

*A curated list of solutions to the most useful questions
for software engineering interviews.*

Coding Interview Essentials

Davide Spataro

*Clear, bite-sized solutions to coding interview
problems.*

Coding Interview Essentials

Davide Spataro

Copyright ©2021 Davide Spataro

Permission is granted to copy, distribute and/or modify this document under the terms of the LGPL, Version 3.0 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section “GNU LESSER GENERAL PUBLIC LICENSE” at page 390.

Contents

Preface	xiv
Aknowledgments	xv
About the author	xvi
A note from the author	xvii
1 Power set generation	1
1.1 Problem statement	1
1.2 Clarification Questions	1
1.3 Discussion	2
1.3.1 Bruteforce - Backtracking-like approach	2
1.3.2 Bit Manipulation	3
2 Square root of an integer	6
2.1 Problem statement	6
2.2 Clarification Questions	6
2.3 Discussion	7
2.3.1 Brute-Force	7
2.3.2 Logarithmic Solution	7
3 Two string anagram	9
3.1 Problem statement	9
3.2 Clarification Questions	10
3.3 Discussion	10
3.3.1 Brute-Force	10
3.3.2 Sorting	11
3.3.3 Histograms	12
4 Two numbers sum problem	14
4.1 Problem statement	14
4.2 Clarification Questions	14
4.3 Discussion	15
4.3.1 Brute-force	15
4.3.2 Hashing	16
4.3.3 Sorting and binary search	17
4.3.4 Sorting and two pointers technique	18

4.3.5	Problem statement	19
4.3.6	Naïve $O(n^4)$ solution	19
4.3.7	$O(n^3)$ solution	20
4.3.8	$O(n^2)$ solution using hashing	20
4.3.9	Problem statement	22
4.3.10	Clarification Questions	22
4.3.11	Discussion	22
5	Unique Elements in a collection	25
5.1	Problem statement	25
5.2	Clarification Questions	25
5.3	Discussion	25
5.3.1	Brute Force	25
5.3.2	Linear time - Hashset	26
6	Greatest element on the right side	28
6.1	Problem statement	28
6.2	Clarification Questions	28
6.3	Discussion	28
6.3.1	Brute Force	28
6.3.2	Linear solution	29
7	String to Integer	31
7.1	Problem statement	31
7.2	Clarification Questions	31
7.3	Discussion	31
7.3.1	Common Variation	32
8	Climb the Stairs	33
8.1	Problem statement	33
8.2	Clarification Questions	33
8.3	Discussion	34
8.4	Common Variation	35
8.4.1	Arbitrary step lengths	35
9	Wave Array	36
9.1	Problem statement	36
9.2	Clarification Questions	38
9.3	Discussion	38
9.3.1	Brute-force	38
9.3.2	Sorting	39
9.3.3	Linear time solution	40
9.4	Common Variations - Return the lexicographically smallest	40
9.4.1	Problem statement	41
9.5	Conclusions	41

10	First positive missing	42
10.1	Problem statement	42
10.2	Clarification Questions	42
10.3	Discussion	43
10.3.1	Brute-force	43
10.3.2	Sorting	43
10.3.3	Linear time and space solution	45
10.3.4	Linear time and constant space solution	46
11	Exponentiation	49
11.1	Problem statement	49
11.2	Clarification Questions	49
11.3	Discussion	49
11.3.1	Using recursion	50
11.3.2	Binary fast exponentiation	50
11.3.3	Iterative solution using bit manipulation	51
11.4	Common Variations	53
11.4.1	Fibonacci numbers - Problem statement	53
12	Largest sum in contiguous subarray	54
12.1	Problem statement	54
12.2	Clarification Questions	54
12.3	Discussion	55
12.3.1	Brute-force	55
12.3.2	Improving the Brute-force	55
12.3.3	Kadane's Algorithm	56
12.4	Common Variations	59
12.4.1	Minimum sum contiguous sub-array	59
12.4.2	Longest positive/negative contiguous sub-array	59
13	String Reverse	60
13.1	Problem statement	60
13.2	Clarification Questions	60
13.3	Discussion	61
13.4	Common Variation	62
13.4.1	Out-of-place solution	62
13.4.2	Recursive solution	62
14	Find the odd occurring element	64
14.1	Problem statement	64
14.2	Clarification Questions	64
14.3	Discussion	64
14.3.1	Brute-force	64
14.3.2	Linear time and space solution	66
14.3.3	Linear time and constant space solution	66

15	Capitalize the first letters of every words	68
15.1	Problem statement	68
15.2	Discussion	69
15.3	Common Variations	71
15.3.1	Apply an user provided function	71
15.3.2	Modify the every k^{th} character of every word	71
16	Trapping Water	73
16.1	Problem statement	73
16.2	Discussion	74
16.2.1	Brute-force	74
16.2.2	Dynamic Programming	76
16.2.3	Two pointers solution	77
16.2.4	Stack based solution	79
17	Minimum element in rotated sorted array	82
17.1	Problem statement	82
17.2	Clarification Questions	83
17.3	Discussion	83
17.3.1	Brute-force	83
17.3.2	Logarithmic solution	83
18	Search in sorted and rotated array	87
18.1	Problem statement	87
18.2	Clarification Questions	87
18.3	Discussion	87
18.3.1	Brute-force	87
18.3.2	Logarithmic time solution	88
19	Verify BST property	90
19.1	Problem statement	90
19.2	Clarification Questions	91
19.3	Discussion	92
19.3.1	A common mistake	92
19.3.2	Top Down approach	93
19.3.3	Brute force	94
20	Clone a linked list with random pointer	97
20.1	Problem statement	97
20.2	Clarification Questions	97
20.3	Discussion	98
20.3.1	Linear memory solution	98
20.3.2	Constant memory solution	99

21	Delete duplicates from Linked List	102
21.1	Problem statement	102
21.2	Clarification Questions	102
21.3	Discussion	102
21.3.1	Brute-force	102
21.3.2	In-place $O(1)$ space solution	104
21.4	Common Variations and follow-up questions	105
22	Generate points in circle uniformly	106
22.1	Problem statement	106
22.2	Clarification Questions	106
22.3	Discussion	106
22.3.1	Polar Coordinates - The wrong approach	106
22.3.2	Loop approach	107
22.3.3	Polar Coordinates - The right approach	109
22.3.4	Conclusion	110
23	Best time to buy and sell stock	112
23.1	Problem statement	112
23.2	Clarification Questions	112
23.3	Discussion	112
23.3.1	Brute-force	112
23.3.2	Linear time solution	113
23.4	Common Variations - Multiple Transactions	113
23.4.1	Problem statement	113
23.5	Discussion	114
23.6	Brute force solution	114
23.7	Linear time solution	115
23.8	Common Variations - Best profit with exactly two transactions	116
23.8.1	Problem statement	116
23.9	Discussion	116
23.10	DP - Linear time solution	116
23.10.1	Linear time and constant space	118
23.11	Variation - Best profit with at most k transactions	118
23.11.1	Problem statement	118
23.12	Discussion	119
23.13	$O(n^2K)$ time and $O(nK)$ space	119
23.14	$O(nK)$ time and space	120
23.15	$O(P K)$ time and $O(P)$ and space	121
24	Find the cycle in a Linked list	123
24.1	Problem statement	123


24.2	Discussion	124
24.2.1	Linear time and space solution	124
24.2.2	Slow and fast pointer solution - Floyd's algorithm	125
25	Reverse a singly linked list	129
25.1	Problem statement	129
25.2	Clarification Questions	129
25.3	Discussion	130
25.3.1	Constant space	131
25.4	Conclusion	132
25.5	Common variation - Reverse a sublist	132
25.5.1	Problem statement	132
26	Min stack	134
26.1	Problem statement	134
26.2	Clarification Questions	135
26.3	Discussion	135
26.3.1	Linear Space solutions	135
26.3.2	Constant space	137
26.3.3	Common Variations	139
27	Find the majority element	140
27.1	Problem statement	140
27.2	Clarification Questions	140
27.3	Discussion	141
27.3.1	Brute-force	141
27.3.2	Hash-map approach	141
27.3.3	Sorting - Counting	142
27.3.4	Sorting - Median	142
27.3.5	Boyer-Moore algorithm	143
27.4	Find the element repeated $\frac{n}{k}$ times.	144
27.4.1	Boyer-Moore algorithm extended	144
28	n^{th} node from the end	145
28.1	Problem statement	145
28.2	Clarification Questions	145
28.3	Discussion	145
28.3.1	Brute-force	145
28.3.2	Two pointers	146
28.3.3	Common Variation	147
29	Validate Parenthesized String	149
29.1	Problem statement	149
29.2	Clarification Questions	149

29.3	Discussion	150
29.3.1	Brute-force	150
29.3.2	Dynamic Programming	151
29.3.3	Greedy - Linear time	153
30	Tree Diameter	156
30.1	Problem statement	156
30.2	Discussion	156
30.2.1	Brute-force	156
30.2.2	Linear time and space	158
31	Largest square in a binary matrix	159
31.1	Problem statement	159
31.2	Discussion	159
31.2.1	Brute-force	160
31.2.2	Dynamic programming	163
31.3	Conclusion	168
32	Sudoku	169
32.1	Problem statement	169
32.2	Clarification Questions	169
32.3	Discussion	171
32.3.1	Backtracking	171
32.4	Conclusion	175
33	Jump Game	176
33.1	Problem statement	176
33.2	Backtracking	176
33.3	DFS	178
33.4	Greedy	179
33.5	Jump Game 2	180
33.6	Problem statement	180
33.6.1	Discussion	180
33.7	Jump Game 3	181
33.7.1	Discussion	182
33.8	Jump game 4	183
33.9	Discussion	184
33.10	Jump Game 5	185
33.11	Discussion	186
34	k^{th} largest in a stream	189
34.1	Problem statement	189
34.2	Clarification Questions	190

34.3	Discussion	190
34.3.1	Array based solution	191
34.3.2	Ordered set	192
35	Find the K closest elements	195
35.1	Problem statement	195
35.2	Clarification Questions	195
35.3	Sorting	196
35.3.1	Binary Search	196
36	Binary Tree mirroring	200
36.1	Problem statement	200
36.2	Discussion	201
37	Count the number of islands	206
37.1	Problem statement	206
37.2	Clarification Questions	206
37.3	Discussion	206
38	Median of two sorted arrays	212
38.1	Problem statement	212
38.2	Clarification Questions	212
38.3	Discussion	213
38.3.1	Brute-force	213
38.3.2	Brute-force improved	214
38.3.3	Logarithmic solution	215
39	BST Lowest Common Ancestor	219
39.1	Problem statement	219
39.2	Clarification Questions	219
39.3	Discussion	220
40	Distance between nodes in BST	224
40.1	Problem statement	224
40.2	Clarification Questions	224
40.3	Discussion	224
40.4	Conclusion	226
41	Counts the items in the containers	227
41.1	Problem statement	227
41.2	Clarification Questions	228
41.3	Discussion	228
41.3.1	Brute-force	228
41.3.2	Linear time solution	229

42	Minimum difficulty job schedule	232
42.1	Problem statement	232
42.2	Clarification Questions	233
42.3	Discussion	233
42.3.1	Brute-force	233
42.3.2	Dynamic Programming	236
42.3.3	Top-down	236
42.3.4	Bottom-up	237
42.4	Conclusion	239
43	Max in manhattan neighborhood^K	240
43.1	Problem statement	240
43.2	Clarification Questions	242
43.3	Discussion	242
43.3.1	Brute-force	242
43.3.2	Dynamic Programming	244
44	Coin Change Problem	251
44.1	Problem statement	251
44.2	Clarification Questions	251
44.3	Discussion	252
44.3.1	The greedy approach and why it is incorrect	252
44.3.2	Formulation as an optimization problem	253
44.3.3	Brute-force	253
44.3.4	Dynamic Programming - Top-Down	254
44.3.5	Bottom-up	257
44.3.6	Conclusion	258
44.4	Common Variations	258
44.4.1	Count the number of ways to give change.	258
45	Number of Dice Rolls With Target Sum	259
45.1	Problem statement	259
45.2	Clarification Questions	259
45.3	Discussion	259
45.3.1	Brute-force	260
45.3.2	Dynamic Programming - Recursive top-down	262
45.4	Dynamic programming - Iterative bottom-up	264
46	Remove duplicates in sorted array	266
46.1	Problem statement	266
46.2	Clarification Questions	266
46.3	Discussion	266
46.3.1	Linear space solution	267
46.3.2	Constant Space	268

46.4	Common Variations	269
46.4.1	Max 2 duplicates allowed	269
46.4.2	Discussion	271
46.4.3	Max k duplicates allowed	272
47	Remove all occurrences - unsorted array	273
47.1	Problem statement	273
47.2	Clarification Questions	273
47.3	Discussion	274
47.3.1	Linear time and linear space solution	274
47.3.2	Linear time and constant space solution	274
48	Sort the chunks, sort the array.	276
48.1	Problem statement	276
48.2	Clarification Questions	276
48.3	Discussion	276
48.4	Brute-force	276
48.5	Linear time	278
49	Palindrome Partitioning II	280
49.1	Problem statement	280
49.2	Discussion	280
49.2.1	Brute-force	280
49.3	Dynamic Programming	282
49.3.1	Top-down	282
49.3.2	Bottom-up	286
50	Find the largest gap	288
50.1	Problem statement	288
50.2	Clarification Questions	288
50.3	Trivial Solution	289
50.4	Radix Sort	289
50.5	Buckets and the pigeonhole principle	290
51	Smallest Range I and II	294
51.1	Problem statement	294
51.2	Clarification Questions	294
51.3	Discussion	294
51.4	Common Variations	296
51.4.1	Smallest range II	296
51.5	Discussion	297

52	Next Greater Element I	301
52.1	Problem statement	301
52.2	Clarification Questions	302
52.2.1	Brute-force	302
52.3	$O(B \log(B))$ time, $O(B)$ space solution	302
52.4	Common Variation	303
52.4.1	First next greater element	303
52.5	Discussion	304
53	Count the bits	307
53.1	Problem statement	307
53.2	Clarification Questions	307
53.2.1	Naïve approach solution	307
53.2.2	DP solution	308
53.2.3	Another efficient approach	309
54	Decode the message	312
54.1	Problem statement	312
54.2	Clarification Questions	312
54.2.1	Recursive solution	313
54.2.2	Iterative solution	314
55	N-Queens	317
55.1	Problem statement	317
55.2	Discussion	317
55.2.1	Brute-force	318
55.2.2	One row one queen	320
55.2.3	One queen per column	321
55.2.4	Brute-force revisited	322
56	Gas Station 	324
56.1	Problem statement	324
56.2	Clarification Questions	325
56.2.1	Brute-force	326
56.2.2	Linear time	327
56.3	Common Variation - Fuel tank with limited capacity	329
57	Merge Intervals	330
57.1	Problem statement	330
57.2	Clarification Questions	330
57.3	Discussion	331
57.3.1	Brute-Force	331
57.3.2	$n\log(n)$ sorting solution	332

57.4	Common Variation - Add a new interval	333
57.4.1	Problem statement	333
57.4.2	Discussion	334
57.5	Common Variation - How many meeting rooms are needed?	336
57.5.1	Problem statement	336
57.6	Brute-force	336
57.7	$n\log(n)$ - Intervals endpoints	337
58	Least Recently Used Cache	340
58.1	Problem statement	340
58.2	Clarification Questions	340
58.2.1	Brute-force	342
58.2.2	Constant time solution	345
59	Longest consecutive sequence	348
59.1	Problem statement	348
59.2	Clarification Questions	348
59.2.1	Solution using Sorting	348
59.2.2	Linear time and space solution	349
60	Merge k sorted lists	354
60.1	Problem statement	354
60.2	Discussion	354
60.2.1	Brute-force	354
60.2.2	Priority-queue approach	357
61	k^{th} smallest element in a sorted matrix	359
61.1	Problem statement	359
61.2	Clarification Questions	360
61.3	Discussion	360
61.3.1	Brute-force	360
61.3.2	Brute-force constant space	360
61.3.3	Binary Search	361
62	Mini Problems	363
62.1	Greatest Common Divisor	363
62.1.1	C++ Brute-force	363
62.1.2	Log-time solution. Euclidean Algorithm	363
62.1.3	C++ Compile-time	364
62.2	Maximum Depth of N-ary Tree	365
62.2.1	Discussion	365
62.2.2	Recursive solution	366
62.2.3	Iterative Solution	366
62.3	Assigning cookies 🍪	367
62.3.1	Discussion	368

62.4	Maximize Sum Of Array After K Negations	368
62.4.1	Discussion	369
62.5	Pairs of Songs With Total Durations Divisible by k	369
62.5.1	Discussion	370
62.6	Trim text	370
62.6.1	Discussion	371
62.7	Items and bags	372
62.7.1	Discussion	372
62.8	Coupons	373
62.8.1	Discussion	373
63	C++ questionnaire	375
64	C++ questionnaire solutions	379
	Appendices	381
	Bibliography	387
	GNU LESSER GENERAL PUBLIC LICENSE	392

Preface

Landing a lucrative job as a software engineer at FAANG is an increasingly competitive endeavour. It is not uncommon to have to go through and impress in 5 or more technical rounds, each of which will require you to solve complex coding problems in a high pressure environment.

A common preparation strategy is to binge code on one of the many online coding platforms that cater to this need. This however, is not the optimal approach as the quality of the resources on these websites is insufficiently high or consistent to ensure the solid grasp of solution fundamentals needed to quickly adapt to the specifics of any question that may come up in real life.

For this reason, I have decided to take a different approach and, having spent several years studying the most common interview problems based on surveys from candidates, I have compiled a subset of problems whose solutions can be applied across a broad array of actual interview questions. These problems and their solutions are presented in bite-sized quality lessons which will hopefully provide you with the tools you need to go on and succeed at interview.

Davide Spataro
Amsterdam, The Netherlands, November 28, 2021

Acknowledgements

About the author

Davide Spataro was born and brought up in Southern Italy. He discovered a love of coding early on and, after attending a high school that focused on humanities, he moved to the University of Calabria in 2008 where he studied and worked for 3 years collaborating with researchers of the Department of Mathematics and Computer Science on the modelling and simulation of complex natural systems. He obtained his BSc in Computer Science in 2011, his MSc (summa cum laude) in 2014 and, in 2018, he successfully defended his Ph.D. thesis with the title: “Acceleration of numerical regular grid methods on manycore systems”. He has worked as a Software Engineer at ASML working on TWINSCAN photolithography systems and at DEGIRO as senior software engineer. He has been involved in programming competitions since age 12 and is passionate about C++ , parallel programming and GPGPU. When absolutely forced to do something other than think about coding problems, Davide can be found either arguing with gravity on a racing bike, paddling in the Mediterranean, listening to or playing classical and jazz piano, or drinking vast amounts of coffee. It’s probably fair to say he thinks about coding when doing these things too.



A note from the author

I started writing what would eventually become this manuscript in 2018 but in truth, its origins lie in a habit I formed years earlier.

Whilst still a student, I began a routine of solving at least one serious coding interview or competitive programming challenge per day and, for those I considered most interesting, putting together a markdown document with a summary of the problem and all the solutions I could identify as well as the thought processes that had led me to them. Essentially I created a series of short essays made up of written code and notes that I could use to bootstrap my understanding of the problem months or years later with the idea that, when it was time to throw myself into a real interviews, I could use these as a reference and sharpen my preparation with material that I knew was correct and that was in a format that I could absorb and understand quickly.

Over the years I accumulated a substantial amount of material and I eventually started sharing it with colleagues at the university and at work. Many of them found the content and the format of my notes useful and convinced me to polish, add illustrations, and organize them into a proper collection which ultimately formed the basis of this text book.

I won't claim that writing this text book has been easy. In fact, whilst the whole purpose of both this book and the study routine that begat it was preparation, I will freely admit I was unprepared for the amount of work involved. From dealing with making quality illustrations to managing a large \LaTeX document and going through the sometimes interminable rounds of revisions and proofreading (and there are likely still some linguistic errors for which I must beg the reader's forgiveness) I have spent many many hours on this project. I have, however, learned a great deal and I hope that the finished text may at least prove useful to you in your interview preparation.

Some notes on the text:

C++ as the language of choice

Almost all the solution code in this book is written in C++ (C++ 14/17/20). I have been using C++ academically and professionally since I started programming and the original source material I created was almost entirely C++ code. I chose not to rewrite it as C++ remains an extremely popular language, frequently adopted in the competitive programming community. Also, engineers skilled in this language continue to be in high demand making C++ , in my opinion, the ideal language to use during the type of interviews the book is intended to prepare you for.

I have tried my best to stick to standard C++ and not to use any compiler-specific features so that it is easy to port every piece of code in this book to different mainstream imperative languages like C#, Python, or Java.

Of course, eventually during an interview you will need to code solutions; preferably elegant, fast and legible ones. Absent the right algorithm, however, all that written code will be pointless. As such the focus of this book is less on the particular features of a

specific coding language and more on understanding the insights and core ideas need to reach the optimal solutions to the problems presented. This does not mean the code you will find in this book is sloppy or C++ unidiomatic. Quite the opposite. I have tried to make full use of its potential at all points.

All the solutions have been compiled and tested using `gcc (GCC)11.1.0` and `clang version 12.0.1` but I am confident they will compile and work on another toolchain/system.

Organization

I always try to be consistent in the way I approach a problem.

1. I begin by looking at a few examples and considering any edge cases or ambiguities in the problem statement that I could clarify by asking questions to the interviewer.
2. When I am comfortable with my understanding of the problem, I try to consider it holistically to either identify similarities with other problems I may have solved in the past or puzzle out a lower bound that can help direct my thinking.
3. I then move into summarizing (usually only on paper) the simplest solution I can think of. This is often the slowest and usually some sort of brute-force solution that can be derived from the problem statement directly.
4. From there I consider whether the brute-force solution can be sped up enough to be considered good or if I need to shift my thinking in an entirely new direction.

These steps are not set in stone and sometimes it can happen that I know the optimal solution immediately but, for the majority of the cases, I would go through most of the steps above. Given that fact, it seemed sensible for the structure of this book to mirror this approach.

Each chapter is broadly organized in the following manner:

- *“Introduction”*, where I set the stage for the problem.
- *“Problem statement”*, enunciating the formal statement for the problem along with a few examples.
- *“Clarification questions”*, where I list all the questions I would ask an interviewer to ensure my understanding.
- *“Discussion”*, where I make the first observations and deductions that I can use later when crafting solutions. The rest of this Section is usually followed by a number of subsections each describing a solution in detail. These subsections are sorted by decreasing time complexity, space used, simplicity/readability, and ease of implementation.
- *“Common variation”*, containing one or more variations of the problem that have been known to be used as an interview question. This section is not present in every chapter.

It’s my opinion that that coding interviews have a lot in common with sports competitions. If you wish to become a good tennis player you will rehearse the same movement over and over again until it becomes muscle memory, but you will also consider rationally when and how to deploy each movement depending on the specific context of your current match. Coding interviews are no different. Interviewers are not expecting candidates to be silent and just write code as if it were muscle memory. A good candidate should be able to show their thought process as they go through the interview, clearly explaining the steps and demonstrating their ability to think strategically about the best solution in a high pressure environment.

The structured approach taken in the chapters of this book are functional to this goal and can be used to rehearse solving a problem as if you were guiding the interviewer along a journey from the slowest to the best solution during your interview.

Audience

The intended audience of this book is intermediate/experienced software engineers with a degree of prior exposure to coding interviews or competitive programming who wish to prepare further for coding interviews. A decent working knowledge of C++ and especially with the Standard Template Library (which is extensively used) is required even if the code presented is not particularly hard or obscure C++ . Indeed, I am sure the code is perfectly understandable to those with experience in C-like languages.

Having an acquaintance with the most common algorithm design techniques (like Dynamic programming for instance) is recommended but not required as, despite the fact this is not an algorithm design book, the solutions are explained in detail. However, being familiar with the concept of complexity analysis is necessary as these concepts are used as a tool to analyze each and every solution but are not explicitly explained.

1. Power set generation

Introduction

Most readers will already be familiar with power sets, i.e. the set of all the subsets of a set S . Being asked to generate the power set of a given set, both directly and indirectly, is a frequently asked question in coding interviews and therefore, in this chapter, we will investigate two different solutions.

1. The first approach derives straightforwardly from the recursive definition of power-set, which states “the power-set of an empty set is a set containing one element only: the empty set itself”. For a non-empty set S , let e be an element of S and T be the original set S set minus e ($T = S \setminus e$), then the power set of S is defined as the union of two distinct power sets:
 - the power set of T
 - and the power set of T modified in a such way that e is added to all of its elements (See Equation 1.1).
2. The second approach is based on a bit-distribution property in the integers binary representation from 0 to the size of the power set.

The two proposed solutions have similar performance in terms of time space required, but the latter is easier to explain and results in shorter and simpler code.

$$\mathcal{P}(S) = \begin{cases} \{\{\}\} & \text{if } S = \{\} \\ \mathcal{P}\{T\} \cup \{t \cup \{e\} : t \in \mathcal{P}\{T\}\} & \text{where } T = S \setminus \{e\} \forall e \in S, \text{ otherwise} \end{cases} \quad (1.1)$$

1.1 Problem statement

Problem 1 Write a function that given a set S returns its power set. The power-set of S ($\mathcal{P}(S)$) is the set of all its subsets including the empty subset (\emptyset) and S itself.

■ Example 1.1

Given the set $S = \{a, b, c\}$, the following is a correct output for this problem:

$$\{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$$

■

■

1.2 Clarification Questions

Q.1.

Q.2. What is the maximum input size?

The maximum number of elements in S is strictly less than 32.

Q.3. Are all the elements in the collection distinct?

No, the elements are not necessarily distinct. S might contain duplicates.

Q.4. Can the elements of the power-set appear in any order?

Yes, subsets can appear in any order. For example the following is also a valid output for the input shown in Example 1.1: $\{\{\}, \{b, c\}, \{a\}, \{a, b\}, \{a, b, c\}, \{b\}, \{a, c\}, \{c\}\}$

1.3 Discussion

The first key point to note is that the power set of a collection of n elements has size 2^n . The proof is straightforward and based on the fact that a subset of S can be uniquely identified by a list $X = \{x_0, x_1, \dots, x_{|S|-1}\}$ of $|S|$ binary variables each carrying the information about whether S_i is part of the subset; the variable x_i is crucial to answer the question **should S_i be part of this subset?**: If x_i is true the answer is yes, otherwise, the answer is no. We have two possible choices for every element of S (either take it or not), then the total number of distinct X s is: $2 \times 2 \times \dots \times 2 = 2^{|S|}$. Two choices for the first element, two for the second, and so on until the last element of S .

This, together with the constraint on $|S|$ ($|S| < 32$) is a strong indicator that an **exponential time and space** solution is expected. After all, we are required to output all the elements of the power set, and thus the number of operations of an algorithm designed for this task cannot be less than the size of the power set itself.

1.3.1 Bruteforce - Backtracking-like approach

The first approach to solving this problem is based on the fact that, during the generation of one of the power set's elements, a decision must be made for each element e of S , on whether or not to include e into the subset. Once this is determined, what we are left with are $|S| - 1$ decisions before we have created a valid subset of $|S|$.

This process is inherently recursive and therefore easily visualized with a tree (see the figure below): a node at level i represents a decision for the i^{th} element of S and a path from the root to a leaf uniquely identifies a subset of S ; after having traversed all the levels down to a leaf, n decisions have been made: one for each of the elements of S .

Collectively, all the paths from the root to the leaves are the power set, and therefore, in order to solve the problem, we have to visit the entire tree.

A common route to solving such problems is to use a backtracking-like approach in order to try all possible decisions (or equivalently to visit every path from the root to a leaf).

The idea is that, for all elements of S , from the first to the last one, we are going to explore the two available possibilities: either take or exclude it from the subset.

We start by making a decision for the first element: From there, we continue to generate all possible subsets where the first decision is never changed. When there are no more subsets to generate, we *backtrack* and change our first decision and repeat the process of generating all possible subsets.

For instance, given $S = \{1, 2, 3\}$, we might start by deciding to use the element 1, and include it in all possible subsets from the remaining elements $\{2, 3\}$ only. Once we are done with it, we can repeat the same process, only this time excluding 1. What we do is: $\mathcal{P}(S) = \{\{1\} \cup \mathcal{P}(\{2, 3\})\} \cup \{\mathcal{P}(\{2, 3\})\}$

The proposed solution will incrementally construct one subset at a time, using an integer variable to keep track of which element we are currently taking the decision for. This type of problem is naturally solved recursively, with a base case of the recursion happening when there is no more decision to make, meaning that the current subset is ready to be included in the solution (it has been produced after n decision steps).

Here below we can see how the C++ code implements this idea. The complexity of this solution is exponential i.e. $O(2^{|S|})$ which as already stated, is as good as it gets.

```

1 void power_set_backtracking_helper(const std::vector<int> &S,
2                                   const int idx,
3                                   std::vector<int> &curr,
4                                   std::vector<std::vector<int>> &ans)
5 {
6     if (idx >= S.size())
7         // base case
8     {
9         ans.push_back(curr);
10        return;
11    }
12
13    // include element S[idx]
14    curr.push_back(S[idx]);
15    power_set_backtracking_helper(S, idx + 1, curr, ans);
16
17    // exclude element S[idx]
18    curr.pop_back();
19    power_set_backtracking_helper(S, idx + 1, curr, ans);
20 }
21
22 std::vector<std::vector<int>> power_set_backtracking(const std::vector<int> &S)
23 {
24     std::vector<std::vector<int>> ans;
25     std::vector<int> current;
26     power_set_backtracking_helper(S, 0, current, ans);
27     return ans;
28 }

```

Listing 1.1: "C++ to the power set generation using backtracking"

Using a backtracking-like approach is convenient because, once we identify that a problem can be solved by fully exploring the associated search space tree, we can immediately start writing the code and rely on our experience as backtracking expert writers to implement a correct solution. It is also concise and short when written in a recursive form (fewer chances to make mistakes, and less code to debug and explain), as well as easily understood. The downside is that, if we decide to use it, an iterative implementation can be a little harder and verbose to write.

Regardless of which type we decide to write, the interviewer will be pleased with the code provided, although it remains important to get to the final solution without making too many implementation mistakes such as forgetting to handle the base case.

1.3.2 Bit Manipulation

The second approach to solving this problem is based on the fact that the values of the bits of the numbers $\{0, 1, 2, \dots, 2^n - 1\}$ already provide all the information necessary to decide whether or not to include an element from S into a subset. The main principle is that the binary representation of all the numbers ($2^{|S|}$ of them) from 0 to $2^{|S|} - 1$ is the power set of n bits. This means that the binary representation of any of those numbers carries the necessary information that can be used to build one subset of $\mathcal{P}(S)$.

For example, given the input $S = \{a, b, c\}$ the table below shows numbers from 0 to $2^3 - 1 = 7$ and their binary representation (second column), as well as how the information about which bit is set can be used to construct one subset of $\mathcal{P}(S)$ (third column). **When the i^{th} bit is set (its value is 1), it means that corresponding i^{th} element of S is chosen, while an unset bit (with value 0) means it is excluded**

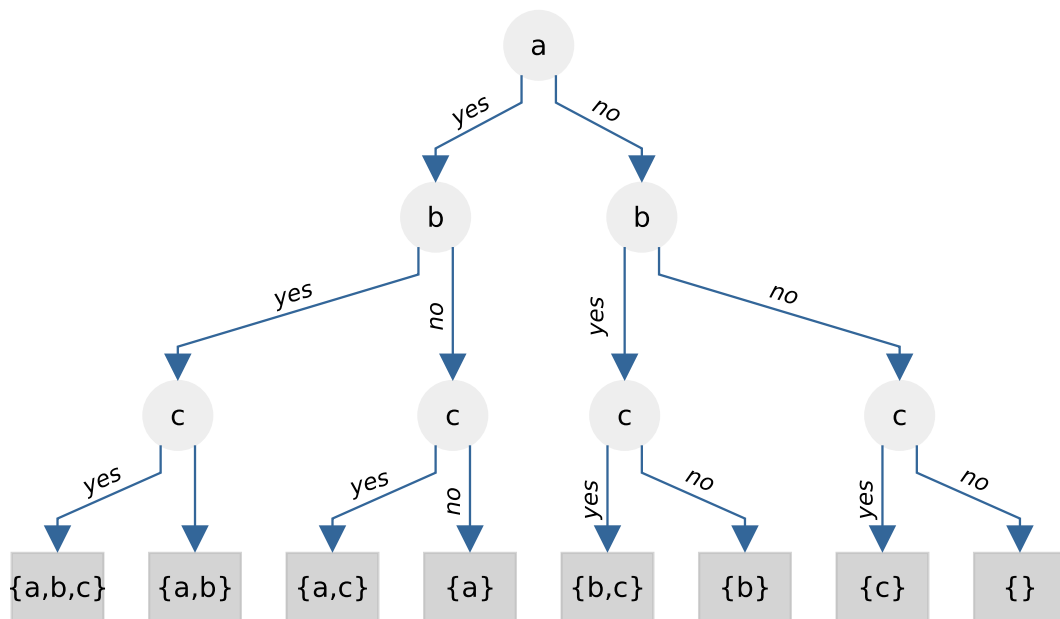


Figure 1.1: Decision tree for the power-set generation using a backtracking-like brute-force approach. At level i are the decisions for the element i in S . A label marked with “yes” identifies the decision to add the corresponding element into the subset, while a node labeled with “no” identifies the opposite. Each path from the root to a leaf is an element of the power set. At the last level is the power set.

Number	Value	Bits	Subset
0		000	$\{\}$
1		001	$\{c\}$
2		010	$\{b\}$
3		011	$\{b,c\}$
4		100	$\{a\}$
5		101	$\{a,c\}$
6		110	$\{a,b\}$
7		111	$\{a,b,c\}$

Table 1.1: This table shows a 1-to-1 mapping between integer values, their binary representation and an element of the power set.

This can be used to write an algorithm in which all the numbers in the range $\{0, 1, 2, \dots, 2^{|S|} - 1\}$ are considered and each of them is used to generate a subset of the final solution. Every number from this range maps uniquely to a subset of $\mathcal{P}(S)$.

The approach is easily understood when we think about the meaning of a bit in the binary representation of integers. One can “build” a number k by summing up powers of 2 where the bits contain the information about whether a certain power of two should be added to the final value. With n bits one can represent 2^n numbers, each corresponding to one and only one subset of the power set of those n bits. Listing 1.2 shows a possible C++ implementation of the idea above.

```

1 constexpr inline bool is_bit_set(const int n, const unsigned p)
2 {
3     return (n >> p) & 1;
4 }
5
6 std::vector<std::vector<int>> power_set_bit_manipulation(
7     const std::vector<int> &A)
8 {
9     const size_t limit = (1ll << A.size()) - 1;
10    std::vector<std::vector<int>> PS;
11    PS.reserve(limit + 1);
12
13    for (int i = 0; i < limit; i++)
14    {
15        std::vector<int> subset = {};
16        for (int p = 0; p < 32; p++)
17            if (is_bit_set(i, p))
18            {
19                subset.push_back(A[p]);
20            }
21        PS.push_back(subset);
22    }
23
24    PS.push_back(A);
25    return PS;
26 }
```

Listing 1.2: “Solution using bit manipulation.”

The complexity of this function is, not surprisingly, $O(2^{|S|})$. We also pay a constant price of 32 for each number we loop through given that we need to inspect all of its bits. This proposed implementation assumes that the size of `int` is 4 bytes, which is true for most systems.

It is important to note the usage `std::reserve` as it should be used in all those scenarios when we already know the final size of the collection we are building. This saves time because it avoids intermediate allocations and copies that must happen during the resize of the vector.

2. Square root of an integer

Introduction

The square root is not only one of the central operations in mathematics, used almost as often as addition, multiplication, or division, but is also at the core of much of our essential technology such as radio and GPS systems. Despite the fact that almost every programming language has dedicated libraries that are optimized such that no serious developer would ever need to write a function to calculate the square root from scratch, interviewers still regularly ask this question to ensure that the candidate can see past the trivial solution embedded in the definition of square root and use divide and conquer concepts effectively.

The square root of a number x , denoted with the \sqrt{x} symbol, is formally defined to be a number y such that $y^2 = y \times y = x$. For example: $\sqrt{4} = 2$ and $\sqrt{1253} \approx 35.3977$. The square root is defined for every positive real number but the most common question in coding interviews focuses on deriving an algorithm for calculating the square root for integers (to avoid the complexity associated with the precision of the answer associated with floating arithmetic and algorithms like the Newton's or bisection method).

As with all such coding interview problems, there are several possible solutions and approaches but in this chapter we will focus on how to write a simple and yet sub-optimal solution that runs in $O(\sqrt{n})$ time that comes straight out of the formal problem statement and the definition of square-root, as well as a classical and much faster and elegant logarithmic time solution.

2.1 Problem statement

Problem 2 Write a function that calculates the integral part of the square root of an integer n i.e. $\lfloor \sqrt{n} \rfloor$. **You cannot use any library functions.**

■ **Example 2.1**

Given $n = 9$ the function returns 3: $\lfloor \sqrt{9} \rfloor = 3$ ■

■ **Example 2.2**

Given $n = 11$ the function returns 3: $\lfloor \sqrt{11} \rfloor \approx \lfloor 3.316624 \rfloor = 3$ ■

■ **Example 2.3**

Given $n = 18$ the function returns 4: $\lfloor \sqrt{18} \rfloor \approx \lfloor 4.242640 \rfloor = 4$ ■

2.2 Clarification Questions

Q.1. What is the maximum value the parameter n can take?

The greatest input is guaranteed to be smaller than 2^{32} .

Q.2. Is n guaranteed to be always positive?

Yes, there is no need to check for invalid input.

0	1	2	$\lfloor \sqrt{n} \rfloor$	$\lfloor \sqrt{n} \rfloor + 1$...	n
0	0	...	1	1	...	1

Table 2.1: Partition of the search space according to the function in Eq. 2.1

2.3 Discussion

A brute-force solution is quickly derivable from the definition of square root given above ($\sqrt{x} = y$ where $y^2 = x$.) and the interviewer will expect to see it identified within the first few minutes of the interview.

2.3.1 Brute-Force

We know that if $y = \sqrt{x}$ then $y^2 = x$. Moreover, y is an integer only when x is a perfect square^①. If x is not a perfect square, then y is a real number and the following holds true: $\lfloor y \rfloor^2 \leq x$ and $\lceil y \rceil^2 > x$. For instance, $\sqrt{5} \approx 2.2360$ and $2^2 = 4 \leq 5$ and $3^2 = 9 > 5$.

We can use this last property to blindly loop through all the integers $k = 0, 1, 2, \dots$ until the following is true: $k^2 \leq n$ and $(k+1)^2 > n$. A solution is guaranteed because eventually, k will be equal to $\lfloor y \rfloor$. Moreover, it is clear that no more than \sqrt{n} numbers will be tested, which proves that the time complexity of this approach is $O(\sqrt{n})$.

Listing 2.1 shows a C++ implementation of this idea.

```

1 int square_root_brute_force(const int n)
2 {
3     long i = 0;
4     while ((i * i) <= n)
5         i++;
6     // i at this point is the smallest element s.t. i*i > n
7     return i - 1;
8 }

```

Listing 2.1: $O(\sqrt{n})$ solution to the problem of finding the square root of an integer.

It is worth noting that the variable i has a type that is larger in size than an `int`. This is necessary in order to prevent overflows during the calculation of i^2 (see the highlighted line). One of the constraints of the problem is that the largest input can be $n = 2^{32} - 1$; The square of that number does not fit in a 4 bytes `int`.

2.3.2 Logarithmic Solution

Binary search can also be effectively used to solve this problem: in order to demonstrate this, we need to look at the problem from a different angle. Let

$$F(k) = \begin{cases} 0 & : k^2 \leq n \\ 1 & : k^2 > n \end{cases} \quad (2.1)$$

be a piece-wise function that partition the search space $[0 \dots n]$ into two parts, as shown in Table 2.1:

1. the numbers less or equal than \sqrt{n}
2. the numbers strictly greater or equal than \sqrt{n}

Clearly, the answer we are looking for is **the greatest value k s.t. $F(k) = 0$** . Note that every number in the left part of the search space, $0 \leq l \leq \lfloor n \rfloor$ has $F(l) = 0$, while the values in the right side, $\lfloor n \rfloor + 1 \leq r \leq n$, have $F(r) = 1$.

^①an integer x is a perfect square if its square root is also an integer

Because the function $F(k)$ splits the search space into two parts, we can use binary search to find the end of the first partition (this is true in general and if we ever recognize a problem that presents these characteristics we can apply binary search to it). We can do that because if we pick an integer from in $[0, n]$, say k , and $F(k) = 1$ we know that k is not the solution and **crucially, also that all the values greater than k are not good candidates because they all belong to the right partition**. On the other hand, if $F(k) = 0$, we know that k might be the solution but also that, **all the values smaller than k are not good candidates as k is already a better answer than any of those numbers would be**. The idea above is implemented in Listing 2.2.

```

1  int square_root_binary_search(const int A)
2  {
3      long l = 0, r = A;
4      int ans = 0;
5      while (l <= r)
6      {
7          const long long mid = l + (r - l) / 2;
8          if ((long)(mid * mid) == (long)A)
9              return mid;
10         if (mid * mid > A)
11         {
12             r = mid - 1;
13         }
14         else
15         {
16             l = mid + 1;
17             ans = mid;
18         }
19     }
20     return ans;
21 }

```

Listing 2.2: $O(\log_2(n))$ solution to the problem of finding the square root of an integer.

The algorithm works by maintaining an interval (defined by the variables `l` and `r`): inside of it lies the solution, which is initially set to be the entire search space $[0, n]$. It iteratively shrinks this range by testing the middle element of $[l, r]$ (value hold by `middle`), and this can lead to one of the following three scenarios:

1. $\text{middle}^2 = n$: `middle` is the solution and also that n is a perfect square.
2. $\text{middle}^2 > n$: `middle` is **not** the solution and we can also exclude all numbers $k \geq \text{middle}$ from the search (by setting `r = middle-1`).
3. $\text{middle}^2 < n$: `middle` is the best guess we have found so far (it might be the solution). We can, however, exclude every number $k < \text{middle}$ (by assigning `l = middle+1`) as when squared, they would also be smaller than middle^2 .

Note the way the midpoint between l and r is calculated. It is common to see it calculated by using the following formula: $(l+r)/2$, however, this can lead to overflow problems when $l+r$ does not fit in an `int`.

Finally, the time and space complexities of this algorithm are $O(\log(n))$ and $O(1)$, respectively. A good improvement with regard to the complexity of the brute-force solution.

3. Two string anagram

Introduction

Anagrams are words that share the same character set. This makes it possible to create multiple words by rearranging the letters in a single source word. For example, the letters in the word “*alerting*” can be reordered to create 4 new words:

- “*altering*”
- “*integral*”
- “*relating*”
- “*triangle*”.

The creation of anagrams, especially ones that reflect or comment on the source words (for instance turning “*Madam Curie*” into “*Radium came*”) is difficult. As such, computers are often used to find anagrams in longer texts, as well as to generate the so-called anagram dictionaries: a specific kind of dictionary, where all the letters in a word and all their transposition are arranged in alphabetical order. Such alphabet dictionaries are often used in games like Scrabble^①. Often, at the core of such applications lies an efficient algorithm for determining if a word is an anagram of another word.

In this chapter, we are going to consider anagrams and, more specifically, how to determine the number of modifications needed to make a word into a valid anagram of another word. Although this type of question is considered straightforward in the context of coding interviews as all it really requires is a basic understanding of the concept of an anagram; nevertheless it is worth studying as it is often asked during the preliminary interview stages and provides an opportunity to demonstrate more than one neat and elegant approach leading to an efficient solution to the problem.

We will examine three possible solutions, beginning with the slow but easy to understand brute-force in Section 3.3.1, moving on to a faster approach using sorting in Section 3.3.2, and finally addressing the optimal solution running in linear time in Section 3.3.3.

3.1 Problem statement

Problem 3 Write a function that given two string, a and b of length n and m , respectively, determines the minimum number of character substitution, $C(s, i, c)$, necessary to make the string a an anagram of the string b .

Two strings are said to be anagrams of one another if you can turn the first string into the second by rearranging its letters.

A substitution operation $C(s, i, c)$ modifies the string s , by changing its i^{th} character into c . Notice that deletions or additions of characters are not allowed. The only operation you can do is change a character of the first string into another one.

In other words, what is the minimum number of characters of the input strings that need to be modified (no addition or deletion) so that a becomes an anagram of b ?

^①<https://en.wikipedia.org/wiki/Scrabble>

■ Example 3.1

- $a = \text{"aaa"}$
- $b = \text{"bbb"}$

The function returns 3. All the characters of a need to be changed into ' b '. ■

■ Example 3.2

- $a = \text{"tear"}$
- $b = \text{"fear"}$

The function returns 1. All that is necessary is turning the first letter ' t ' into a ' f '. ■

■ Example 3.3

- $a = \text{"Protectional"}$
- $b = \text{"Lactoprotein"}$

The answer for this case is 0 because *Protectional* is already an anagram of *Lactoprotein*. ■

3.2 Clarification Questions

Q.1. Are the letters of the string always only letters from the English alphabet?

Yes, letters are always from the English alphabet.

Q.2. Should the function be case sensitive?

No. You can assume the input letters are always lower case.

Q.3. Can the input string be modified? No, the input is immutable.

No, the input strings are immutable.

Q.4. What value should be returned when there is no solution?

In such case you can return -1 .

3.3 Discussion

Let's start by reviewing what the word anagram means in the context of this problem. First note that both a and b contain a single word (which can be fairly long). Moreover, for a to be an anagram of b , it has to be the case that there exists an arrangement of characters in a that is equal to b . In other words, the question we need to answer is: is it possible to shuffle the character of a so that we obtain b ? For this to be the case, it must be that a and b contain the same set of characters meaning that sorting both a and b would make them equal. In addition, as a consequence of the fact that no addition or deletion is allowed, **a and b must have the same length**. On the other hand, if they have the same length then it is always possible to solve this problem because in the worst case, we can modify every letter of a (see Example 3.1). Thus, the only case when the problem has no solution has been isolated: when $n \neq m$ we must return -1 otherwise we can proceed with our calculation knowing that a solution exists.

3.3.1 Brute-Force

One of the first options to consider is a solution where we generate all possible arrangements of the letters in a , and for each of these arrangements, calculate the number of modifications necessary to convert it into b . The key idea is that the cost of transforming

a string into another is equal to the number positions having different letters. For instance, the cost of transforming “*abcb*” into “*bbbb*” is 2 because the two strings differ in the first and third letters.

Although it is simple to explain, this approach must be considered sub-optimal because the number of arrangements of a set of n letters grows as fast as $n!$. Moreover, enumerating all the arrangements is no trivial task, unless we use a library function capable of doing it (for instance, the C++ standard library provides the function `std::next_permutation`^② devoted to this purpose).

Listings 3.1 shows a C++ implementation of the idea above.

```
1 #include <algorithm>
2 #include <limits>
3 #include <string>
4
5 int count_different_letters(const std::string &a_perm, const std::string &b)
6 {
7     assert(a_perm.size() == b.size());
8
9     int count = 0;
10    for (size_t i = 0; i < a_perm.length(); i++)
11    {
12        if (a_perm[i] != b[i])
13            ++count;
14    }
15    return count;
16 }
17
18 int solution_brute_force(const std::string &a, const std::string &b)
19 {
20     if (a.length() != b.length())
21         return -1;
22
23     std::string a_perm(a);
24     sort(a_perm.begin(), a_perm.end());
25     int ans = std::numeric_limits<int>::max();
26     do
27     {
28         ans = std::min(ans, count_different_letters(a_perm, b));
29         if (ans == 0)
30             break;
31     } while (std::next_permutation(a_perm.begin(), a_perm.end()));
32
33     return ans;
34 }
```

Listing 3.1: “Brute force.”

3.3.2 Sorting

This brute-force solution does a lot of superfluous work, because it tries to find a permutation of the string *a* requiring minimal modifications to be morphed into *b*. But is it really necessary to turn *a* into **exactly** *b*, or is it sufficient to modify *a* so that it is equal to a particular permutation of *b*? After all, being an anagram is a transitive property: if *a* is a permutation of *b* and *b* is a permutation of *c*, then *a* must also be a permutation of *c*.

By definition, an anagram of *b* is any permutation of its characters, and therefore, the particular permutation in which the characters of *b* are sorted is a valid anagram on

^②https://en.cppreference.com/w/cpp/algorithm/next_permutation

its own. It is much easier than checking all possible permutations, to modify a into the “sorted” anagram of b (where all of its characters are sorted), rather than to exactly b because all we need to do is to create a copy of both a and b , sort both of them and then calculate the character-by-character difference. **This approach works because if x is an anagram of b then x is also an anagram of ‘sort(b)’.** In other words, it does not matter how the characters are arranged in a and b , as the only thing that matters is the set of the characters appearing in a and b : the order in which characters in both a and b appear does not matter.

Listings 3.2 shows how we can take advantage of this fact and write a fast solution for this problem.

```

1
2 int solution_sorting(const std::string &a, const std::string &b)
3 {
4     if (a.length() != b.length())
5         return -1;
6
7     std::string aa(a);
8     std::string bb(b);
9
10    std::sort(aa.begin(), aa.end());
11    std::sort(bb.begin(), bb.end());
12    return count_different_letters(aa, bb);
13 }
```

Listing 3.2: “Solution based on sorting.”

Note that, if the input was mutable, then, the additional space occupied by the copies of the string `aa` and `bb` could have been avoided.

The time complexity of Listing 3.2 is $O(n \log(n))$ (because of sorting). The space complexity is $O(n)$ (we create copies of the input strings).

3.3.3 Histograms

There is another piece of information that we have not used yet: **the alphabet** from which the letters of a and b are taken from **is small**. If the only thing that matters is the set of characters appearing in a and b (and not their order, as discussed above), then we can use the same idea at the core of the bucket sort algorithm to achieve a linear time complexity solution.

The key idea here is to pre-process a and b so as to calculate their per-character frequencies, denoted here as F_a and F_b , respectively. An entry of $F_a[c]$ and $F_b[c]$, where $c \in \{a, b, \dots, z\}$ (a letter of the alphabet), contains the frequency of character c , in a and b , respectively.

If F_a and F_b are the same, then a and b have exactly the same character set and a is an anagram of b . Otherwise, it must be the case that some characters of a appear in b a different number of times. In this case, we can fix a in such a way as to make sure that its frequencies F_a match the ones in F_b . But the main question remains unanswered: how many operations are necessary to do so? In order to get this answer, it is useful to look at the difference (D) of F_a and F_b .

$$D = F_a - F_b = \{D[a] = (F_a[a] - F_b[a]), D[b] = (F_a[b] - F_b[b]), D[c] = (F_a[c] - F_b[c]), \dots, D[z] = (F_a[z] - F_b[z])\}$$

$D[c]$ (where $c \in \{a, b, \dots, z\}$) contains the difference between the number of occurrences of the character c in the string a and b . Depending on whether the value of $D[c]$ is greater or smaller than 0, a has an excess or a deficit of the letter c , respectively.

Firstly, note that $\sum_{c=a}^z D[c] = 0$. This observation stems from the fact that $|a| = n = m = |b|$ (a and b must have equal length for this problem to have a solution as noted above) and that if a has an excess of a certain character c then there must exist another character $d \neq c$ that the string a has a shortage of. If that is not the case, it is impossible for a and b to have equal length.

We can use this fact to modify the excesses of the letters of a , the ones having a positive value of D into some of the letters there is a shortage of so that eventually, every single value of D is zero. If $D[c] = x$ is going to take x modifications to transform the excess of characters c . The answer to this problem is, therefore, the sum of all the positive numbers of D .

Listings 3.3 shows a possible implementation of the solution above.

```

1  int solution_histogram(const std::string &a, const std::string &b)
2  {
3      if (a.length() != b.length())
4          return -1;
5
6      std::array<int, 128> F = {0};
7      for (int i = 0; i < a.size(); i++)
8      {
9          F[a[i] - 'a']++;
10         F[b[i] - 'a']--;
11     }
12
13     int ans = 0;
14     for (const auto x : F)
15         if (x > 0)
16             ans += x;
17
18     return ans;
19 }
```

Listing 3.3: "C++ solution to the two string anagram problem using the histogram approach."

Note how the array of differences of frequencies D can be calculated easily without explicitly computing the frequencies for the characters of a and b but simply by adding 1 to $D[c]$ when the letter c appears in a and subtracting 1 when it does in b .

The time and space complexity of the code above is $O(n)$ and $O(1)$ in space (we are using an array of 128 integers regardless of the size of the input). We cannot do any better than this, as all characters in the input strings must be read at least once.

4. Two numbers sum problem

Introduction

This chapter addresses one of the most frequently posed problems during the early stages of the coding interview process: two numbers sums. The problem is hard enough to require non-trivial insights in order to be able to write a non-trivial solution but, at the same time, it is not so hard that it would take a candidate hours to come up with something meaningful to say or to write. It's ubiquity also means that any interviewer will expect a candidate to be at least familiar with the issues and able to present multiple paths to solution.

We are going to look at several possible solutions.

First, the inefficient brute force approach which we will subsequently refine it into a fast and time-optimal one). Then, a radically different approach based on sorting which [AGAIN SOMETHING ABOUT WHY THIS?]. Finally, condensing the strengths of all the previous solutions into a time and space optimal solution that will likely perform best in an interview context. As we will see, this final solution is efficient and not terribly difficult to write and explain; key elements for success in any coding interview.

4.1 Problem statement

Problem 4 Write a function that takes an array of integers A of size n and an integer T , and returns **true** if the sum of any two distinct elements I is equal to T , **false** otherwise.

More formally: Given an array $= \{a_1, \dots, a_n\}$ and T , where $a_i, T \in \mathcal{N}$, return:

- **true** if $\exists i, j \ i \neq j$ s.t. $a_i + a_j = T$
- **false** otherwise

■ Example 4.1

Given $A = \{9, 4, 17, 42, 36, -3, 15\}$ and $T = 14$, the function returns **true** because we can obtain 14 by summing up together the elements 17 and -3 . If $T = 17$ the answer is **false**. ■

■ Example 4.2

Given $A = \{1, 3, 7\}$ and $T = 8$, the function returns **true** because we can obtain 8 by summing up together the elements 7 and 1. If $T = 6$ the answer is **false**. ■

4.2 Clarification Questions

Q.1. Is the input array modifiable?

Yes, the input array can be modified.

Q.2. Are the integers guaranteed to be all positive or all negative?

No, A can contain positive or negative numbers.

Q.3. Are the values in A guaranteed to be from a given range?

No, the input is arbitrary. No assumption can be made on the magnitude of the elements of A .

Q.4. Can a pair be made from an element and itself?

No, the pair's elements should be distinct. You cannot use the same element a_i twice. You can however use two elements at indices i and j s.t. $i \neq j$ and $a_i = a_j$.

Q.5. Are all elements in the array unique?

No, duplicates are allowed.

Q.6. Is the input sorted?

No, the ordering of A is arbitrary.

Q.7. Shall the function integer overflow be considered when performing the sum of two integers? Is it possible for two elements of A to sum up to a value that does not fit in a standard `int`?

No, you do not need to worry about overflow.

4.3 Discussion

4.3.1 Brute-force

The brute force solution is straightforward because it consists of a direct application of the formal problem statement. The solution space consists of all possible ordered pairs (a_i, a_j) , $i < j$. Two nested loops can be used to enumerate all those pairs, and, for each of them, we can check whether their sum is equal to T : if that is the case, then **true** can be immediately returned, otherwise, if we have checked every possible pair and none of them was good, then we can return **false**. You will find an a fomalization and an implementation of this idea in Algorithm 1 and Listing 4.1), respectively.

Algorithm 1: Two loops, quadratic solution to the question in Section 4

```
Input:  $A$  // An array  $A$  of length  $n$ 
Input:  $T$  // An integer  $T$ 
Function solveQuadratic( $A, T$ )
    for  $i \leftarrow 0$  to  $n - 1$  do
        for  $j \leftarrow i + 1$  to  $n$  do
            if  $a_i + a_j = T$  then
                return True ;
            end
        end
    end
    return False ;
End Function
```

```
1 bool two_numers_sum_bruteforce(const std::vector<int> &A, const int T)
2 {
3     const size_t size = A.size();
4     for (int i = 0; i < size - 1; i++)
5         for (int j = i + 1; j < size; j++)
6             if (A[i] + A[j] == T)
7                 return true;
8     return false;
9 }
```

Listing 4.1: "C++ solution of the two number sum problem with a brute force approach."

The time complexity of this solution is $O(n^2)$ because there is a quadratic number of ordered pairs and in the worst case, we will look at **all** of them.

The number of iterations of the internal loop depends on the value of i and it is described by the function: $f(i) = n - i - 1$. The total number of iterations the second loop runs in the worst case is the the sum of $f(i)$ for all values of i : $\sum_{i=0}^{n-2} f(i) = (n-1) + (n-2) + (n-3) \dots + 1 = \sum_{x=1}^{n-1} x = \frac{n(n-1)}{2} = O(n^2)$

The space complexity is $O(1)$.

4.3.2 Hashing

The internal loop of the brute force solution above can be eliminated entirely with the help of a hash table. The key insight is that if a solution exists involving a_i then it must be the case that exists another element $a_j = a_i - T$ with $i > j$.

What this means in practice is that we can loop through A one element at a time and keep track in a lookup table of all the elements seen so far so that the lookup operation for the aforementioned element a_j can be performed in constant time.

Algorithm 2 and Listing 4.2 shows this idea in code.

Algorithm 2: Hashset, linear solution to the *two number sum* question in Section 4.

Input: A // An array A of length n
Input: T // An integer T
Output: true if two distinct element of A sum to T , False otherwise
Function solveHashSet(A, T)
 $H \leftarrow \text{CreateHashSet}\langle \text{int} \rangle$;
 for $i \leftarrow 0$ **to** n **do**
 $\text{target} \leftarrow (T - a_i)$ **if** $H.\text{find}(\text{target})$ **then**
 return True
 else
 $H.\text{insert}(a_i)$
 end
 end
 return False;
End Function

```
1 bool two_numers_sum_hashset(const std::vector<int> &A, const int T)
2 {
3     std::unordered_set<int> H;
4     const size_t size = A.size();
5     for (int i = 0; i < size; i++)
6     {
7         if (H.find(T - A[i]) != end(H))
8             return true;
9         H.insert(A[i]);
10    }
11    return false;
12 }
```

Listing 4.2: "C++ solution of the two number sum problem using hashing."

The time complexity of this approach is $O(n)$ (technically it is linear on average due to the complexity of lookups in hash tables) because the input array is scanned once and for each of its elements, only one lookup and insertion are performed in the hash table (both operations costing constant time on average).

The space complexity is also $O(n)$ as, in the worst case scenario, the whole input array is stored in the lookup table.

A common mistake when solving this problem using this approach is to insert the whole input array into the lookup table, and only after searching for $(T - a_i)$. The mistake becomes evident when T is an even number ($2|T$) and $\frac{T}{2}$ appears in A exactly once, at index k i.e. $a_k = \frac{T}{2}$ causing `H.find(T-a_k)` to return **true**, which is wrong because this corresponds to a solution where we sum a_k twice to obtain T .

For instance, when $A = \{1, 2, 5, 4\}$ and $T = 10$ this approach wrongly returns **true**, even if there are not two elements at distinct indices in A whose sum is T (we would use 5 twice to obtain 10).

■ **Example 4.3**

$A = \{1, 2, 5, 4\}$
 $T = 10$

Algorithm 3 wrongly return true even if there are not two distinct elements whose sum is 10. ■

Algorithm 3: Hashset, linear solution to the *two number sum* question in Section 4

```

input : An array  $A$  of length  $n$ 
input : An integer  $T$ 
output: true if two distinct element of  $A$  sum to  $T$ 
Function solveHashSet( $A, T$ )
     $H \leftarrow \text{CreateHashSet}<\text{int}>;$ 
    // Add the whole array in the hashset
    for  $i \leftarrow 0$  to  $n$  do
        |  $H.\text{insert}(a_i);$ 
    end
    for  $i \leftarrow 0$  to  $n$  do
        |  $\text{target} \leftarrow T - a_i;$ 
        | if  $H.\text{find}(\text{target})$  then
        | | return True
        | end
    end
    return False;
End Function

```

4.3.3 Sorting and binary search

As with countless other problems on arrays, sorting the input often leads to a faster and more efficient solution.

We can start by asking ourselves how the problem changes if A is sorted. Sorted arrays are naturally associated with binary search as many problems can be solved efficiently by pairing sorting and binary search on arrays. This problem is no different therefore we can use binary search if A is sorted to substitute the internal loop of the brute force solution presented [above](). This way, we lower the overall complexity to $O(n \log(n))$; it costs $O(n \log(n))$ to sort the input array in the first place, and the actual search consists of n binary searches, each of them costing $O(\log(n))$.

The space complexity is $O(1)$ because no additional space is required as the array is sorted in place.

Listing 4.3 shows a C++ implementation of this idea. Note that it uses `std::binary_search` from the C++ standard library and that a possible follow-up question might be to show your own version of the binary search algorithm.

```
1 bool two_numers_sum_sorting(std::vector<int> &A, const int T)
2 {
3     std::ranges::sort(A);
4     const size_t size = A.size();
5     for (int i = 0; i < size - 1; i++)
6         if (std::binary_search(begin(A) + i + 1, end(A), T - A[i]))
7             return true;
8     return false;
9 }
```

Listing 4.3: "C++ solution of the two number sum problem with sorting and binary search."

4.3.4 Sorting and two pointers technique

There is a variation to the to the approach described in Section 4.3.3 which still involves sorting but uses a two-pointers technique instead of binary search to finish the job.

The key idea is that, once A is sorted, the algorithm initializes two pointers: one starting at the beginning (p_s) and the other at the end (p_e) of the array respectively. It continues by looking at the sum of the two elements pointed by the two pointers and moving one of the two at each step using the following logic:

- if $a[p_s] + a[p_e] = T$ a solution has been found. The algorithm returns true.
- if $a[p_s] + a[p_e] > T$, $p_e = p_e - 1$. The right pointer is moved to the left. Moving p_e to the left has the effect of making the sum of the values pointed by the two pointers smaller (this has an effect at the next iteration).
- if $a[p_s] + a[p_e] < T$, $p_s = p_s + 1$. The right end pointer is moved to the left. Moving p_s to the right has the effect of making the sum of the values pointed by the two pointers larger.

Listing 4.4 shows an implementation of the idea above. Note that compared to the solution using the binary search, this one is shorter and simpler to write. Moreover, it does not use library functions.

```
1 bool two_numers_sum_two_pointers(const std::vector<int> &A, const int T)
2 {
3     int s = 0, e = A.size() - 1;
4     while (s < e)
5     {
6         const int sum = A[s] + A[e];
7         if (sum < T)
8             s++;
9         else if (sum > T)
10            e--;
11        else
12            return true;
13    }
14    return false;
15 }
```

Listing 4.4: "C++ solution of the two number sum problem with the two pointers technique."

Despite the fact that the overall time complexity is still $O(n\log(n))$, this solution is likely to be faster than using binary search due to the fact that the array is scanned linearly (which makes caches happier) by the two pointers and not in the scattered way of binary search.

Variation: Four numbers sum problem

4.3.5 Problem statement

Problem 5 Write a function that takes four arrays of integers, A, B, C, D and a integer T , and returns how many distinct tuple (i, j, k, l) where exist such that $A_i + B_j + C_k + D_l = Y$.

■ Example 4.4

Given:

- $A = \{1, 2\}$,
- $B = \{-2, -1\}$,
- $C = \{-1, 2\}$,
- $D = \{0, 2\}$, and
- $T = 0$

The answer is 2 because the only two valid tuples are:

1. $(0, 0, 0, 1): A_0 + B_0 + C_0 + D_1 = 1 + (-2) + (-1) + 2 = T = 0$
2. $(1, 1, 0, 0): A_1 + B_1 + C_0 + D_0 = 2 + (-1) + (-1) + (-1) = T = 0$

4.3.6 Naïve $O(n^4)$ solution

We can solve this problem very easily by using the same approach we have described in Section 4.3.1. The idea is that we can use four nested loops and enumerate all possible 4-elements tuples of indices. Listing 4.5 shows how this can be implemented. This is not, however, the fastest solution we can come up with as it has a time complexity of $O(n^4)$

```

1  int four_sum_bruteforce(const std::vector<int>& A,
2                          const std::vector<int>& B,
3                          const std::vector<int>& C,
4                          const std::vector<int>& D,
5                          const int T)
6  {
7      int ans = 0;
8      for (size_t i = 0; i < A.size(); i++)
9          for (size_t j = 0; j < B.size(); j++)
10             for (size_t k = 0; k < C.size(); k++)
11                 for (size_t l = 0; l < D.size(); l++)
12                     {
13                         const long sum = (long)A[i] + (long)B[j] + (long)C[k] + (long)D[l];
14                         if (sum == T)
15                             ans++;
16                     }
17
18     return ans;
19 }
```

Listing 4.5: "Brute force naïve solution to the four numbers sum problem."

Needless to say, that this is not the fastest solution we can come up with, considering it has a time complexity of $O(n^4)$.

4.3.7 $O(n^3)$ solution

The trivial solution shown in Listing 4.5 can be improved by using a similar approach to the one we used to improve the brute-force quadratic time solution for the two numbers problem in Listing 4.1 and to the linear time (and space) in Listing 4.2.

The idea is that the inner-most loop is searching for a value $D_l = x$ s.t. if it summed to $A_i + B_j + C_k$ gives us T ; in other words: $x + (A_i + B_j + C_k) = T$. Therefore $x = T - (A_i + B_j + C_k)$. If there is a way of avoiding a linear search in the array D for such a value, then we could bring down the complexity from $O(n^4)$ to $O(n^3)$.

This is possible if we use a hash map. If we create a hashmap mapping the value of D and to their frequencies, the inner-most loop of the $O(n^4)$ solution above can be substituted with a query to the hashmap which runs in constant time (on average).

Listing 4.6 shows an implementation of this idea. Note that, in order to obtain the maximum saving in terms of work avoided, the arrays are rearranged in such a way that D is the longest of the four input arrays.

```
1  int four_sum_cubic(std::vector<int>& A,
2                    std::vector<int>& B,
3                    std::vector<int>& C,
4                    std::vector<int>& D,
5                    const int T)
6  {
7      if (A.size() > D.size())
8          std::swap(A, D);
9      if (B.size() > D.size())
10         std::swap(B, D);
11      if (C.size() > D.size())
12         std::swap(C, D);
13
14      // D is now the longest
15      std::unordered_map<int, int> Dmap; // frequencies map for D
16      for (const auto d : D)
17         Dmap[d]++;
18
19      int ans = 0;
20      for (size_t i = 0; i < A.size(); i++)
21         for (size_t j = 0; j < B.size(); j++)
22             for (size_t k = 0; k < C.size(); k++)
23             {
24                 const long sum = (long)A[i] + (long)B[j] + (long)C[k];
25                 if (auto it = Dmap.find(T - sum); it != Dmap.end())
26                 {
27                     ans += it->second;
28                 }
29             }
30
31      return ans;
32 }
```

Listing 4.6: "Brute force cubic time solution to the four numbers sum problem."

4.3.8 $O(n^2)$ solution using hashing

This problem can be however be solved more efficiently in quadratic time if we use hashmaps by holding all the frequencies of all the values we can obtain by summing up any two elements of A and B and of C and D . The key idea is that we can build two distinct hashmaps:

- *AB*: holding the frequencies of the values obtainable by summing any two elements of *A* and *B*
- *CD*: holding the frequencies of the values obtainable by summing any two elements of *C* and *D*.

The space required for both *AB* and *CD* is quadratic, which is more than the space used by any of the previous solutions, but this extra space also enables us to solve this variation in quadratic time.

The idea is that we are going to spend $O(n^2)$ time to construct both *AB* and *CD* and then again $O(n^2)$ to calculate the final answer by searching into *CD* for the value $T - y$ where y is an element of *AB*. If such a value exists in *CD* it means that there exists one element in *A* and one in *B* such that they sum up to y and one element *C* and one in *D* such that they sum up to $T - y$. Summing all these elements up gives: $y + T - y = T$. This approach is shown in Listing 4.7.

```

1  int four_sum_hashing(const std::vector<int>& A,
2                      const std::vector<int>& B,
3                      const std::vector<int>& C,
4                      const std::vector<int>& D,
5                      const int T)
6  {
7      const size_t size = A.size();
8      std::unordered_map<int, int> ab;
9      for (size_t i = 0; i < size; i++)
10         for (size_t j = 0; j < size; j++)
11             ab[A[i] + B[j]]++;
12
13     std::unordered_map<int, int> cd;
14     for (size_t i = 0; i < size; i++)
15         for (size_t j = 0; j < size; j++)
16             cd[C[i] + D[j]]++;
17
18     int ans = 0;
19     for (const auto [k, v] : ab)
20
21         if (auto it = cd.find(T - k); it != cd.end())
22             ans += v * it->second;
23
24     return ans;
25 }
```

Listing 4.7: "Quadratic time solution to the four numbers sum problem."

Note that the first thing we do is to fill *AB* by looping over all possible pairs of elements from *A* and *B*. We then do the same thing for *CD*, and finally, in the last loop, we take care of calculating the answer by searching, for each element (k, v) of *AB*, where k is the sum obtained by one element of *A* and one of *B*, and v is the number of ways we can obtain it, into *CD* for the target value $T - k$. If such a value exists into *CD* then we know we can obtain T . The number of times that is possible is dictated by the frequencies of k and of the target value in *CD*.

However, you might have already noticed that we do not really need to explicitly create the map *CD*. When we create *CD* we already have all the values of *AB* and therefore for a given $C_i + D_j$ we can already find out how many pairs in *AB* exist that we can use to get a total sum of T . This optimization does not really change the overall space complexity but in practice it means that we use half the memory and we avoid doing $O(n^2)$ work by eliminating the last loop.

Listing 4.8 shows this optimized version.

```

1  int four_sum_hashing_space_optimized(const std::vector<int>& A,
2                                     const std::vector<int>& B,
3                                     const std::vector<int>& C,
4                                     const std::vector<int>& D,
5                                     const int T)
6  {
7      const size_t size = A.size();
8      std::unordered_map<int, int> ab;
9      for (size_t i = 0; i < size; i++)
10         for (size_t j = 0; j < size; j++)
11             ab[A[i] + B[j]]++;
12
13     int ans = 0;
14     for (size_t i = 0; i < size; i++)
15         for (size_t j = 0; j < size; j++)
16             if (auto it = ab.find(T - (C[i] + D[j])); it != ab.end())
17                 ans += it->second;
18
19     return ans;
20 }

```

Listing 4.8: "Space optimized quadratic time solution to the four numbers sum problem."

Variation: Max triplet sum

4.3.9 Problem statement

Problem 6 Write a function that, given an array I of length n , returns the maximum value obtainable by summing 3 distinct elements of I : I_i , I_j and I_k such that $0 \leq i < j < k \leq n - 1$ and $I_i < I_j < I_k$.

■ Example 4.5

Given $I = \{2, 5, 3, 1, 4, 9\}$ the function returns 16. The max value of 16 is obtainable by summing together the elements of I at indices: 0, 1 and 5 : $I_0 + I_1 + I_5 = 2 + 5 + 9 = 16$.

Note that there is another way of obtaining the max sum of 16; that is by using the elements at indices 2, 4 and 5: $I_2 + I_4 + I_5 = 3 + 4 + 9 = 16$. ■

■ Example 4.6

Given $I = \{3, 2, 1\}$ the function returns -1 as there is no valid triplet in I . ■

■ Example 4.7

Given $I = \{1, 3, 2\}$ the function returns -1 as there is no valid triplet in I . ■

■ Example 4.8

Given $I = \{1, 2, 3\}$ the function returns 6. There is only one valid triplet in I . ■

4.3.10 Clarification Questions

Q.1. Is it guaranteed that I contains at least three elements?

No. When $n < 3$ the function should return -1 .

Q.2. Is the answer guaranteed to fit a standard `int`?

Yes you can assume the the answer always fits a standard 4-bytes `int`

4.3.11 Discussion

This problem is asking us to find the largest possible sum obtainable by summing up three distinct elements of I with the additional constraint that when ordered according to their

indices they form a sorted sequence. You can form such a triplet by selecting an element at index i , then another element at index j that appears after and is larger than the element at index i and finally, a third element at index k which appears after and is larger than the element at index j .

4.3.11.1 Brute-force

We can solve this problem in a brute-force manner by trying all possible triplets of ordered indices $i < j < k$ and keeping track of the triplet yielding the maximum value. Three simple nested loops are enough to implement this idea as shown in Listing 4.9. The time complexity of this approach is $O(|I|^3)$ which is far from optimal. The space complexity is $O(1)$ as no additional space is used.

```

1 int max_triplet_sum_bruteforce(const std::vector<int>& I)
2 {
3     const auto n = std::ssize(I);
4     int ans = -1;
5     for (int i = 0; i < n; i++)
6     {
7         for (int j = i + 1; j < n; j++)
8         {
9             if (!(I[i] < I[j]))
10                continue;
11
12            for (int k = j + 1; k < n; k++)
13            {
14                if (!(I[j] < I[k]))
15                    continue;
16                // here: i < j < k and I[i] < I[j] < I[k]
17                ans = std::max(ans, I[i] + I[j] + I[k]);
18            }
19        }
20    }
21    return ans;
22 }
```

Listing 4.9: Cubic time complexity bruteforce solution.

4.3.11.2 Pre-calculation and Binary Search

The cubic time complexity approach discussed in Section 4.3.11.1 can be dramatically improved if we approach the problem a little differently. Imagine we would be able to efficiently calculate L_j and G_j for an element at index j where:

1. L_j is the **largest** value among any of the elements of I appearing at any index **smaller** than j which is **smaller** than I_j ;
2. G_j is the **largest** value among any of the elements of I appearing at any index **higher** than j which is **larger** than I_j .

When these values are available we can calculate the value of the largest sum obtainable by any triplet having I_j as the middle element. The triplet (L_j, I_j, G_j) yields the largest sum as, if that was not the case, it would mean that either a larger element than L_j existed that is also smaller than I_j in any of the positions before j or that an element exists that is larger than G_j in any of the positions after j . Both of these two scenarios are impossible because L_j is by definition the largest element that is smaller than I_j and appears before index j and similarly, G_j is defined to be the largest element appearing after index j that is larger than I_j .

We can use this fact to calculate the answer to this problem by looping over I and for each element I_j calculating $L_j + I_j + G_j$. The largest of the sums calculated this way is the

final answer. But how can we calculate L_j and G_j for I_j ?

L_j can be calculated efficiently by keeping a sorted list of all the values appearing before index j and using binary search to find L_j in the list; while G_j can be pre-calculated using a similar method as that used to solve the problem in Chapter 6 where we loop from the right to the left of I to keep track of the largest element (M) seen so far. If M is larger than the element we are currently examining (I_x) then M is also the largest element larger than I_x appearing after x . If not, it means that I_x is the largest element so far and that I_x does not have any larger element on its right: thus $M = I_x$ (see Section 6.3.2 and Listing 6.2).

```
1
2 constexpr auto MIN_INT = std::numeric_limits<int>::min();
3 auto find_largest_smaller_than(const std::set<int>& N, const int n)
4 {
5     auto it = N.lower_bound(n);
6     if (N.size() == 0 || it == std::begin(N))
7         return std::make_tuple(MIN_INT, false);
8     return std::make_tuple(*(--it), true);
9 }
10
11 int max_triplet_sum_prefix_binary_search(const vector<int>& A)
12 {
13     std::set<int> N;
14     std::vector<int> L;
15
16     L.resize(A.size(), MIN_INT);
17     int M = A[A.size() - 1];
18     for (int i = std::ssize(A) - 2; i >= 0; i--)
19     {
20         L[i] = A[i] < M ? M : MIN_INT;
21         M = std::max(A[i], M);
22     }
23
24     int ans = -1;
25     for (size_t i = 0; i < A.size(); i++)
26     {
27         auto larger = L[i];
28         auto [smaller, exists] = find_largest_smaller_than(N, A[i]);
29         if (larger != MIN_INT && exists)
30             ans = std::max(ans, A[i] + larger + smaller);
31         N.insert(A[i]);
32     }
33     return ans;
34 }
```

Listing 4.10: $O(n\log(n))$ solution to the max triplet sum problem.

5. Unique Elements in a collection

Introduction

The problem presented in this chapter is very popular in coding interviews, possibly because it features an incredibly simple statement and is therefore easily understood. We will look first at the intuitive brute-force solution that can be coded in a few minutes and then examine how it can be refined and optimized into a short and efficient one.

5.1 Problem statement

Problem 7 Given a string s of length n , determine whether it does **not** contain duplicate characters.

■ **Example 5.1**

- Given $s = \text{"graph"}$ the function returns `true`. There are no duplicates in s . ■

■ **Example 5.2** Given $s = \text{"tree"}$ the function returns `false`. Characters at indices 2 and 3 are the same. ■

5.2 Clarification Questions

Q.1. What is the maximum size of the input?

The maximum length for the input string is 10^6 .

Q.2. Are all characters upper or lower case?

No, both upper and lower case might be present.

Q.3. Is the function case-sensitive?

Yes.

Q.4. Can I assume only alphanumeric characters are present in the input?

Yes. Upper and lower case Latin letters and numbers only.

5.3 Discussion

As this problem so popular, the interviewer will expect a good solution in a short time frame.

For this reason the obvious $O(n^2)$ solution should be immediately put on the whiteboard or verbally explained.

5.3.1 Brute Force

The easy approach to solving this problem works by looping over each character at index i , and checking if s_i is present in any of the elements of s appearing at positions higher than i . In other words we want to check whether the following is true: $\exists j \text{ s.t. } s_j = s_i \text{ and } j > i$. This idea can be implemented as shown in Listing 5.1 using two simple nested loops.

```

1 bool unique_elements_brute_force(const std::string &s)
2 {
3     for (size_t i = 0; i < s.size(); i++)
4         for (size_t j = i + 1; j < s.size(); j++)
5             if (s[i] == s[j])
6                 return false;
7
8     return true;
9 }

```

Listing 5.1: "C++ solution for determining all characters in a string are unique."

As a stylistic improvement to the code in Listing 5.1, Listing 5.2 uses the C++ standard library function `std::find` to search for a duplicate of the character s_i . This not only makes the code shorter and cleaner but also shows to the interviewer that you are able to use the standard library and do not try to reinvent the wheel.

```

1 bool unique_elements_brute_force_std(const std::string &s)
2 {
3     for (auto it = s.begin(); it != s.end(); it++)
4         if (std::find(it + 1, s.end(), *it) != s.end())
5             return false;
6     return true;
7 }

```

Listing 5.2: "C++ solution for determining if all characters in a string are unique using `std::find`"

5.3.2 Linear time - Hashset

In Listing 5.1 the internal loop is doing the hard work of searching for a duplicate of the character at index i . We can trade space for time and reduce the complexity of the search for a duplicate of s_i down to $O(1)$. The idea is that we can use a set to keep track, as we loop over the characters of s , of all the distinct characters seen so far. A search for a duplicate of s_i becomes a query into this set. If the query is positive then we know we have seen this character before, otherwise we insert s_i into the set and can continue processing the rest of s . Listing 5.3 shows how this idea can be implemented.

```

1 bool unique_elements_map(const std::string &s)
2 {
3     std::unordered_set<char> L;
4     for (size_t i = 0; i < s.size(); i++)
5     {
6         if (L.contains(s[i]))
7             return false;
8         L.insert(s[i]);
9     }
10    return true;
11 }

```

Listing 5.3: "C++ solution for determining all characters in a string are unique in $O(n)$ using an hashset."

This approach effectively lowers the time complexity down to linear, but at the cost of some space. But how much space exactly? Intuition would suggest $O(n)$ as that is the size of the input string and, after all, we might be inserting into the hashset all of the characters of s . But intuition is wrong in this case as the string is made of characters from an alphabet Σ which has a (very) limited size, at most 128 (which is the size of the ASCII

set) elements. The insert instruction will not be executed more than $|\Sigma|$ times. Because of this the space complexity of this solution is $O(1)$.

We can expand further on this as follows: **Every string with more than $|\Sigma|$ character contains at least one duplicate**(follows from the pigeon principle^①). The longest string with only unique characters is one of the permutations of “*abcde...zABCD ...Z123 ...9*”. Thus the solution using the hashset has complexity of $O(1)$ because in the worst case we can have $|\Sigma|$ negative lookups. For this reason we can limit our investigation to only strings that have size smaller or equal to $|\Sigma|$ character. For all other strings we can immediately return false. Note that, in light of these new facts, the brute-force approach also has a complexity of $O(1)$ if i and j in Listing 5.1 are forced to stay below $|\Sigma|$.

It therefore follows that the most efficient solution to present during an interview need only use an array of booleans of size $|\Sigma|$ storing the information regarding the presence of a character in the portion of s considered so far. If at any time the currently examined character has been already seen, then it is a duplicate. Listing 5.4 shows an implementation of this idea.

```
1 bool unique_elements_final(const std::string &s)
2 {
3     constexpr size_t ALPH_SIZE = 128;
4
5     if (s.size() > ALPH_SIZE)
6         return false;
7
8     std::array<bool, ALPH_SIZE> F = {};
9     for (size_t i = 0; i != s.size(); i++)
10    {
11        // index in F
12        const int idx = s[i] - 'a';
13        if (F[idx])
14            return false;
15        F[idx] = true;
16    }
17    return true;
18 }
```

Listing 5.4: "C++ solution for determining all characters in a string are unique in $O(n)$ using an hashset."

^①The pigeonhole principle (https://en.wikipedia.org/wiki/Pigeonhole_principle) states that if n items are put into m containers, with $n > m$, then at least one container must contain more than one item.

6. Greatest element on the right side

Introduction

This chapter discusses a problem that is known for having been asked during on-site interviews at Amazon. It is a relatively easy problem on arrays where, in a nutshell, we are given one as input and we are asked to find for each element of its element the value of the largest element among the ones to its right.

Since, as we shall see, it is not a particularly challenging problem as all the information to come up with a good solution are hiding in plain sight in its statement, it is essential to focus our efforts towards making a good impression on the interviewer by showing clean reasoning, clear and simple communication as well as an elegant implementation of the solution.

6.1 Problem statement

Problem 8 You are given an array A of size n . You have to modify A in place s.t. $A[i] = \max(A[i+1], A[i+2], \dots, A[n-1])$. In other words $A[i]$ should be substituted with the maximum value among all elements $A[j], j > i$. If such element does not exist set $A[i] = -1$.

■ Example 6.1

Given the input array $A = \{15, 22, 12, 13, 12, 19, 0, 2\}$, the output of the function in this case should be $A = \{22, 19, 19, 19, 19, 2, 2, -1\}$. ■

■ Example 6.2

Given the input array $A = \{2, 3, 1, 9\}$, the output of the function in this case should be $A = \{9, 9, 9, -1\}$. ■

6.2 Clarification Questions

Q.1. Are the element of the array sorted?

No, the input array is not sorted.

Q.2. Are the element always positive or negative?

The elements can be either positive or negative.

Q.3. Is $n > 0$?

Not necessarily; the input array A can be empty.

6.3 Discussion

6.3.1 Brute Force

A brute-force solution for this problem is not difficult to conceive because all it takes is to follow the instructions given in the formal problem statement. We can think of processing

A from left to right and to find the value associated to $A[i]$ by scanning all of the elements to its right.

This can be implemented using a double loop or more conveniently in C++ using the `std::max_element()` function as shown in Listing 6.1.

```
1 void greatest_right_bruteforce(std::vector<int> &A)
2 {
3     for (size_t i = 0; i < A.size(); i++)
4     {
5         const auto greatest = max_element(begin(A) + i + 1, end(A));
6         A[i]                = (greatest != end(A)) ? *greatest : -1;
7     }
8 }
```

Listing 6.1: C++ brute-force solution using `std::max_element()` from the STL.

Listing 6.1 works by looping through A from left to right and for each element $A[i]$ issuing a call to `std::max_element()`. The search for the maximum is enforced to be performed only on the elements to the right of $A[i]$ by using as starting point `begin(A)+i+1`^①. It should be highlighted that for the very last element of A , `begin(A)+i+1` correspond to the element past the end and therefore it is always modified into -1 ; this is the only element not having any other fellow elements to its right.

The complexities of this approach are quadratic and constant for time and space, respectively. This solution is to be considered poor as a much faster and more efficient solution exists.

6.3.2 Linear solution

The approach used in Listing 6.1 can be greatly improved if instead of looping from left to right, the scan is performed from right to left. We can start inspecting the A from index $A.size() - 2$ to 0 because, as it was mentioned above, the element at index $A.size() - 1$ is always turned into -1 . This shift in the order we inspect A allows us to keep track of the maximum element (M) on the right side of an element to be calculated in constant time as:

- at first $M = A[A.size() - 1]$ (the largest element to the right of element at index $A.size() - 2$ is always the element at the back of A);
- if M is maintained properly, we can update $A[i]$ by simply copying M in it;
- crucially, we can update M by only using the **old** value of $A[i]$ (we can remember it by saving it in a temporary before the updated happens): $M = \max(M, A_{old}[i])$;

This idea above is implemented in Listing 6.2

```
1 void greatest_right_final1(std::vector<int> &V)
2 {
3     if (V.size() <= 0)
4         return;
5     // max so far
6     int M = -1;
7     int i = V.size() - 1;
8     do
9     {
10         const int m = std::max(M, V[i]);
11         V[i]         = M;
12         M             = m;
13         i--;
```

^①The `template< class ForwardIt > ForwardIt max_element(ForwardIt first, ForwardIt last);` functions operates on a range of elements specified by `first` and `last` [7].

```

14     } while (i >= 0);
15 }

```

Listing 6.2: C++ linear time and constant space solution.

The code works by scanning A from right to left (i is initialized to $A.size() - 1$ which allows the last element of A to be modified into -1 even if we do not set -1 explicitly) using M as a placeholder for the maximum value among the elements with index strictly higher than i . m , instead, contains the value of the largest element among all the elements with index higher or equal to i (it also considers the element being currently processed during the active iteration). Every element $A[i]$ is overwritten with the current value of M which is itself subsequently overwritten with the value held m .

An alternative and marginally more condensed implementation of Listing 6.2 is shown in Listing 6.3.

```

1 void greatest_right_final2(std::vector<int> &V)
2 {
3     if (V.size() > 0)
4     {
5         for (int i = V.size() - 2, M = V.back(); i >= 0; i--)
6         {
7             const int m = std::max(M, V[i]);
8             V[i]        = M;
9             M            = m;
10        }
11        V.back() = -1;
12    }
13 }

```

Listing 6.3: Alternative implementation of Listing 6.2.

The time and space complexities of this approach are linear and constant, respectively. These are optimal figures, as we need to at least read and write every element of A once.

7. String to Integer

Introduction

The problem discussed in this chapter is often used as a warm-up question during onsite interviews or is part of pre-selections online assessments. The question is about converting a string to an integer, a familiar operations in the day of a programmer. Being a straightforward problem, even a small bug in the actual code or conceptual flaw in the algorithm used in the solution can kill the chance to continue our journey in the hiring process. Therefore, it is imperative to ask good clarification questions to ensure the details of the problem as well as all corner cases are well understood. For example, the interviewer might want us to take care of negative numbers, or to take care of invalid input, but that may not be explicitly mentioned when the problem is presented.

7.1 Problem statement

Problem 9 Write a function that given a string s containing only numbers (characters from the range $[0-9]$), parse it into its integer representation without using any library specific functions (like `atoi()` in C++ or `Integer.parseInt()` in Java).

■ **Example 7.1**

If $s = "12345"$, then the function returns the integer 12345. ■

7.2 Clarification Questions

Q.1. Does the function need to handle integer overflow?

No, the input will never cause overflow. You might assume the integer always fits a standard integer.

Q.2. Can the input string have leading zeros?

Yes, the string might have one or more leading zeros.

■ **Example 7.2**

If $s = "0000012345"$, then the function should return the integer 12345. ■

7.3 Discussion

An elegant solution presents itself if we use the idea behind the decimal positional numeral systems. In any positional number system, the ultimate numeric value of a digit is also determined by the position it holds and not only by the digit itself. Take as an example the number 427: although 7 is thought of as a larger number than 4, the 7 is worth less than the 4 in this instance because of its respective position within the number. The value of a digit d at position i is equal to $d \times 10^i$. Thus the value of a number represented by the string ($k+1$ characters long) $s = d_0d_1 \dots d_k$ is equal to $(d_0 \times 10^0) + (d_1 \times 10^1) + \dots + (d_k \times 10^k)$.

All we need to do to write a solution for this problem is to go through each digit of the number and calculate the answer using the formula above. For example, given the string

$s = "22498"$ then its decimal value is equal to: $(2 \times 10^4) + (2 \times 10^3) + (4 \times 10^2) + (9 \times 10^1) + (8 \times 10^0) = 20000 + 2000 + 400 + 90 + 8 = 22498$

It is worth highlighting that using this approach leading zeros are not a problem because they clearly do not contribute to the final result as $0 \times 10^x = 0$. Let's consider $s = "00022498"$ for which we can calculate its decimal value as follows: $(0 \times 10^7) + (0 \times 10^6) + (0 \times 10^5) + (2 \times 10^4) + (2 \times 10^3) + (4 \times 10^2) + (9 \times 10^1) + (8 \times 10^0) = 0 + 0 + 0 + 20000 + 2000 + 400 + 90 + 8 = 22498$

The concept above can be implemented by looping through the string from right to left and summing up each digit of the string at position i multiplied by 10^i as shown in Listing 7.1.

```
1 int string_to_int1(const std::string &s)
2 {
3     int ans = 0;
4     int p = 1; // 10^i
5     for (int i = s.size() - 1; i >= 0; i--)
6     {
7         const int char_to_int = (int)(s[i] - '0');
8         ans += char_to_int * p;
9         p *= 10;
10    }
11    return ans;
12 }
```

Listing 7.1: C++ linear time and constant space solution.

Listing 7.1 is considered to be good as its time complexity is linear in the length of the input string. We cannot do better than this figure as we need to at least inspect each digit once.

7.3.1 Common Variation

As mentioned above, this problem might be prone to having many variations and below you will find a list of the most common ones:

- Add support for negative numbers. One optional char which could either be `+` or `-`, at the beginning of the string signals the sign. A solution for this variation is shown in Listing 7.2.
- Handle overflow and return 0 when the answer does not fit into an `int`.
- Raise an exception (or return a certain value) in case of bad input. For instance when invalid digits are present in the string e.g. $s = 123f456$.
- Perform the conversion by interpreting s as being represented in base b . For instance when Example 7.1 is interpreted as a number in base 8, the function returns 5349.

```
1 int string_to_int_negative(const std::string &s)
2 {
3     if (s.size() == 0)
4         return 0;
5     const int sign = s[0] == '-' ? -1 : 1;
6     // skip the first char if sign is specified
7     const int start = (s[0] == '-') || (s[0] == '+') ? 1 : 0;
8     return sign * string_to_int1(std::string(begin(s) + start, end(s)));
9 }
```

Listing 7.2: C++ solution to the string to integer problem with negative number support. It uses Listing 7.1 as subroutine.

8. Climb the Stairs

Introduction

This chapter deals with a classic problem often set during the interview process for big tech companies like Amazon or Google. It also shares the same underlying structure, key properties and solutions as many other standard interview problems (for example, the coin change problem in Chapter X) and therefore we can apply the techniques discussed in this chapter to all question where the problem statement is structured: *Given a target find minimum (maximum) cost / path / sum to reach the target.*

The basic approach to solving this problem is problem: *Choose minimum/maximum path among all possible paths before the current state, then add the value for the current state.* but we will examine this in more detail in order to clarify what is meant by words such as “current” and “state” and also address a common variation on the problem.

8.1 Problem statement

Problem 10 You are climbing a stair case and it takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

■ Example 8.1

Given $n = 3$ the answer is 3 because there are three ways (See Figure 8.1 to climb to the top of the stairs:

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

■ Example 8.2

Given $n = 4$ the answer is 5 because there are five ways (See Figure 8.2 to climb to the top of the stairs:

1. 1 step + 1 step + 1 step + 1 step
2. 2 steps + 1 step + 1 step
3. 1 step + 1 step + 2 steps
4. 1 step + 2 steps + 1 step
5. 2 steps + 2 steps

8.2 Clarification Questions

Q.1. Can the size of the staircase be zero?

Yes, the staircase can be made of zero steps.

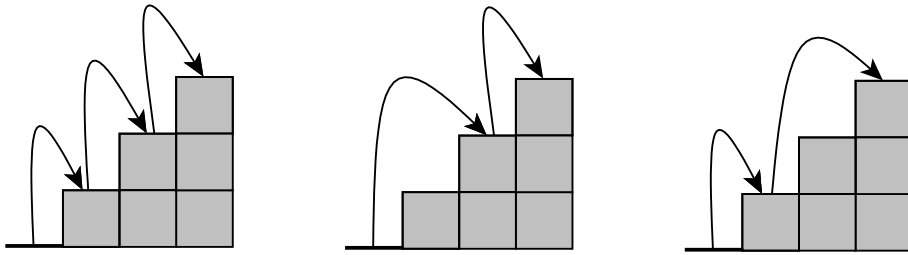


Figure 8.1: All different ways to climb a 3 stairs staircase using steps of size 1 or 2.

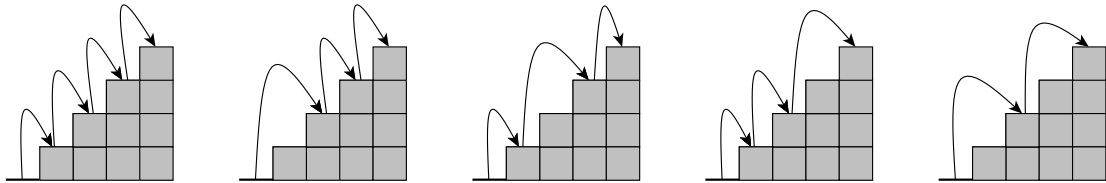


Figure 8.2: All different ways to climb a four stairs staircase using steps of size 1 or 2.

Q.2. It is guaranteed that the answer will fit a built-in integer?

Yes, do not worry about overflow.

8.3 Discussion

First, let's examine a few examples in order to identify the relevant patterns. Table ?? shows how many ways there are to climb a stair of length n up to $n = 7$.

Looking at the table one thing should be immediately apparent: the number of ways to climb the stair of size n is equal to the n^{th} element of the **Fibonacci** sequence (starting with two 1). Once that is clear then the solution is straightforward as shown in Listing 8.1.

```

1  /*int fibonacci(int k)
2  {
3      int p = 0, c = 1;
4      while(k--)
5      {
6          const int tmp = c;
7          c = p+tmp;

```

n	Ways
0	0
1	1
2	2
3	3
4	5
5	8
6	13
7	21

Table 8.1: All the ways to climb a stair of length $n \leq 7$


```

8         p = tmp;
9     }
10    return p;
11 }*/
12
13 int fibonacci(int k)
14 {
15     if (k <= 1)
16         return 1;
17     return fibonacci(k - 1) + fibonacci(k - 2);
18 }
19
20 int stair_climbing_fibonacci(const int n)
21 {
22     if (n <= 1)
23         return n;
24     return fibonacci(n);
25 }

```

Listing 8.1: Solution to the stairs climbing problem with steps of size 1 and 2 using Fibonacci.

Now, let's have a look at why the seemingly unrelated fibonacci sequence plays a role in this problem. If the problem is looked at as an iterative process in which at each step a certain number of stairs are climbed. For instance if $n = 3$ and:

- 1 step is hopped then the number of remaining steps is $3 - 1 = 2$.
- 2 steps are hopped then the number of remaining steps is $3 - 2 = 1$.

When one step is hopped, the problem changes from climbing n stairs to $n - 1$ stairs. At this point the problem is seemingly unchanged except for the number of stairs left to climb and the same reasoning can be applied again:

- 1 step is hopped then the number of remaining steps is $(n - 1) - 1 = n - 2$.
- 2 steps are hopped then the number of remaining steps is $(n - 1) - 2 = (n - 3)$.

As can be seen, two decisions are possible i.e. climbing one or two stairs, exactly as in the fibonacci sequence, until either the n step or a point past it is reached.

8.4 Common Variation

8.4.1 Arbitrary step lengths

What happens when the step sizes allowed are not just 1 or 2 but an array of k positive values $A = \{s_1 < s_2 < \dots < s_k\}$. The problem statement for this harder variant is as follows:

Problem 11 You are climbing a stair case and it takes n steps to reach the top.
 Each time you can either climb s_1 or s_2 or ... or s_k steps where $0 < s_1 < s_2 < \dots < s_k$.
 In how many distinct ways can you climb to the top? ■

The solution to this problem variant is equivalent to the easier version described in Section 8.1 when the allowed step sizes are $s_i = 1$ and $s_2 = 2$.

9. Wave Array

Introduction

We are used to talking about sorting in terms of arranging items in either ascending or descending order, but in fact, sorting is simply the process of arranging items systematically according to a criterion that can be purely arbitrary.

The problem discussed in this Chapter concerns writing an algorithm for sorting the elements of a collection in an unusual way by placing them at even indices such that each is surrounded by either greater or smaller elements. For example, the collection: $\{1, 3, -1, 3, 2, 4\}$ is properly sorted on these terms whilst $\{1, 3, -1, 1, 2, 4\}$ is not.

This question often arises at first on-site interview stage for companies like *Adobe*, and *Google*, mostly as it is not considered particularly difficult and should be solvable by writing just a few lines code. It can, however, prove challenging to get right the first time under pressure therefore, once a working draft of the solution is finished, our advice is to still take time to make sure the code is behaving as expected, especially in edge case scenarios.

9.1 Problem statement

Problem 12 Given an array A of n integers, arrange the numbers in a wave-like fashion. A valid wave array X has its elements arranged in one of the two following ways:

1. $x_0 \geq x_1 \leq x_2 \geq x_3 \leq x_4 \geq \dots$ where $x_{2i-1} \geq x_{2i} \leq x_{2i+1}$
2. $x_1 \leq x_2 \geq x_3 \leq x_4 \geq x_5 \leq \dots$ where $x_{2i-1} \leq x_{2i} \geq x_{2i+1}$

■ Example 9.1

Given $A = \{10, 5, 6, 3, 2, 20, 100, 80\}$ the followings are all valid output (see Figure 9.1a):

- $\{20, 5, 10, 2, 80, 6, 100, 3\}$
- $\{10, 5, 6, 2, 20, 3, 100, 80\}$

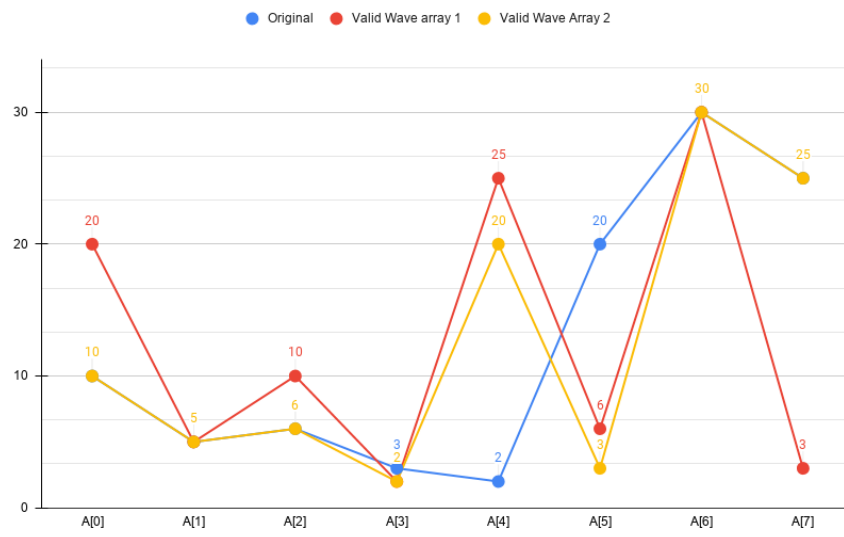
■ Example 9.2

Given $A = \{20, 10, 8, 6, 4, 2\}$ the followings are all valid output (see Figure 9.1a):

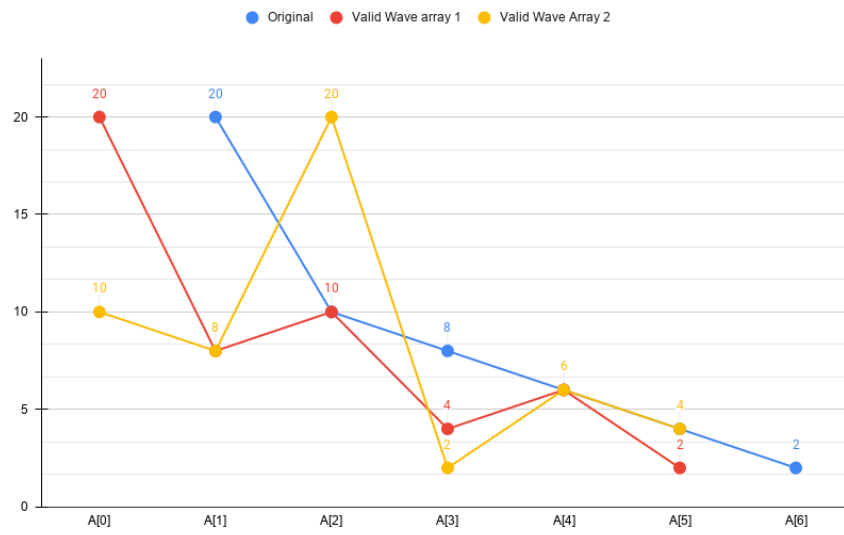
- $\{20, 8, 10, 4, 6, 2\}$
- $\{10, 8, 20, 2, 6, 4\}$

■ Example 9.3

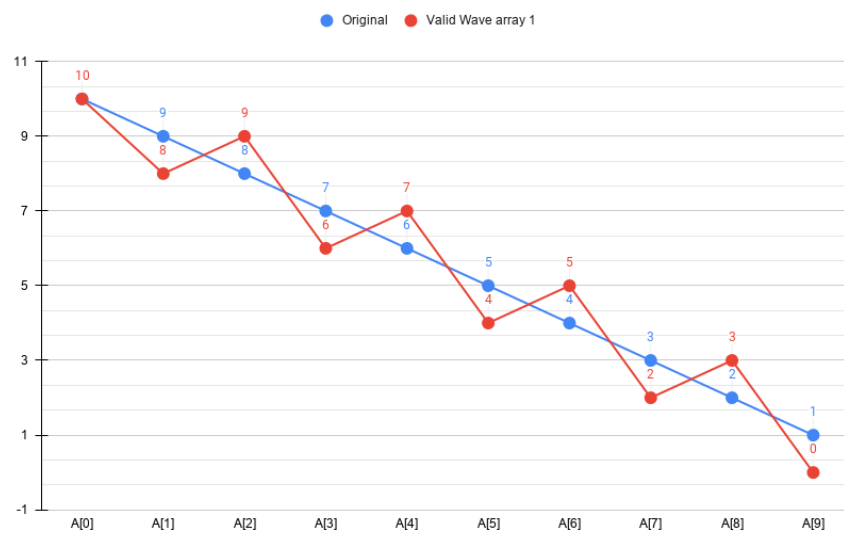
Given $A = \{10, 9, 8, 7, 6, 5, 4, 3, 2, 1\}$ the following is a output: $\{10, 8, 9, 6, 7, 4, 5, 2, 3, 0, 1\}$ (see Figure 9.1a).



(a) Input and solutions for Example 9.1.



(b) Input and solutions for Example 9.2.



(c) Input and solutions for Example 9.3.

9.2 Clarification Questions

Q.1. Does the array *A* only contain positive numbers?

No, the input numbers can be positive or negative.

Q.2. Are duplicates in *A* allowed?

Yes, duplicates might be present.

Q.3. Do the numbers in *A* lie in a given particular range? If yes which one?

*No; no assumptions can be made on the values in *A*.*

9.3 Discussion

The challenge confronting us here is the creation of an entirely new array *X* (we, therefore, know from the very beginning we must make a copy of *A* at some point) that contains the same elements in *A*, arranged in a form that reminds a wave. An array of this type has its elements arranged so that they produce a zig-zig-like pattern when plotted on a graph.

Sequences of numbers of this type can be described as having the property that all of their elements located at even indices are **all** either *local minima* or *maxima*^①. Identifying a local minimum/maximum is easy but it is only helpful when we want to test whether a sequence is a valid wave array.

9.3.1 Brute-force

One way to attack this problem is by enumerating every possible arrangement of the elements of *A* and applying the criteria of wave-array validity discussed above to find a solution. We can enumerate all permutations of an array quite easily by using a function like `std::next_permutation(Iterator first, Iterator last)` which (taken from the docs): “Rearranges the elements in the range `[first,last)` into the next lexicographically greater permutation”.

This idea is implemented in Listing 9.1.

```
1  template <typename It, typename Cmp_fn>
2  bool is_valid_wave_array(It begin, It end, Cmp_fn fn = std::greater<int>())
3  {
4      It curr = begin;
5      while (curr != end)
6      {
7          if (const It prev = curr - 1; prev >= begin && !cmp_fn(curr, prev))
8              return false;
9
10         if (const It next = curr + 1; next < end && !cmp_fn(curr, next))
11             return false;
12     }
13     return true;
14 }
15
16 std::vector<int> wave_brute_force(const std::vector<int> &A)
17 {
18     std::vector<int> B(A);
19     std::sort(begin(B), end(B));
20
21     do
22     {
23         if (is_valid_wave_array(B.begin(), B.end(), std::greater<int>())
24             || is_valid_wave_array(B.begin(), B.end(), std::less<int>()))
25             return B;
```

^①An element is a local minimum/maximum if it is lower/higher than its two immediate neighbors.

```

26 } while (std::next_permutation(B.begin(), B.end()));
27
28 throw std::runtime_error("Should never happen");
29 }

```

Listing 9.1: Brute-force time solution to the wave array problem.

The code above has a time complexity that is proportional to $n!$ and it is, therefore, impractical to use even for smaller sized arrays as the factorial of 10 is already greater than 3 million.

9.3.2 Sorting

As per all array problems, the first thing to consider is: *does sorting the elements (we are referring here to a canonical sorting in increasing order) change the difficulty of the problem?* Incrementally sorted sequences are simpler to approach as they provide strong and clear guarantees on how elements relate to each other. More importantly, the same problem is very often easier to solve on a sorted collection than on an unsorted one.

If we apply the wave-array validity criterion (discussed above on local minima/maxima) on a sorted array $S = \{s_0 \leq s_1 \leq \dots \leq s_{n-1}\}$ we notice that S fails the test as there is only one local minimum and local maximum i.e. s_0 and s_{n-1} (which also happen to be the global minimum and maximum).

The question is then how does S change if every element that is located at an even index is swapped with its subsequent neighbor? When all elements at indices $2i$ and $2i+1$ ($i = 0, 1, \dots$) are swapped, then: $S = \{s_1 \geq s_0 \leq s_3 \geq s_2 \leq s_5 \geq s_4 \leq s_7 \geq \dots\}$ which is now in better shape to pass the wave-array validity test as every element at even index is surrounded by smaller (or equal) elements.

Notice that the elements of S have been shuffled around and that the element a_i is not located at index i anymore (contrary to its original position). We can see that a_3 is now located at index 2 and is surrounded by a_0 and a_2 , which are both smaller or equal to a_3 . Similarly, a_5 is now placed at index 4 and is surrounded by the elements a_2 and a_4 , both known to be smaller or equal than a_5 .

We can use this observation to solve the problem efficiently and elegantly as shown in Listing 9.2.

```

1  std::vector<int> wave_sorting(const std::vector<int> &A)
2  {
3      std::vector<int> B(A);
4      std::sort(begin(B), end(B));
5      auto it = B.begin();
6      for (auto it = B.begin(); it + 1 < B.end(); it += 2)
7      {
8          std::swap(*it, *(it + 1));
9      }
10     return B;
11 }

```

Listing 9.2: Solution to the wave array problem using sorting.

Solution 9.2 works by creating a copy of the input array A named B , which is subsequently sorted. The code then proceeds to swap every element located at an even location with the element after it. You can see the `swap` operation is applied to the iterators `it` and `it+1`, and that at the end of each iteration `it` is incremented by 2. This, together with the fact `it` initially pointer to the first even element at location 0, means that only pairs of items at indices of the form $(2i, 2i+1)$ are swapped.

This solution is considered good as its time and space complexity are $O(n\log(n))$ and $O(n)$ respectively.

Before proceeding with this option, it is worth noting that, should the interviewer ask you to return the lexicographical minimum arrangement amongst all possible arrangements, a Brute Force sorting solution won't work. In such cases you should consider the solutions proposed at section [FILL THIS IN] instead.

9.3.3 Linear time solution

Although the solution using sorting presented in Section 9.3.2 is likely sufficient for the interview, there is also a solution that works in linear time and that is as easy to implement and explain. The core idea remains the same: elements at even index should always be greater (or smaller, equivalently) than their adjacent neighbors. The difference here is that, on this occasion, we will enforce it in a single pass on the array by swapping elements at even indices with their direct neighbors (to the left and to the right) if they happen to be smaller in such a way that the largest element among $x_{2i-1}, x_{2i}, x_{2i+1}$ always ends up going to the location $2i$.

We do this by iterating over all even indices and performing the following operations:

1. if the current element a_{2i} is smaller than the element a_{2i-1} then swap them;
2. if the current element a_{2i} (possibly newly assigned from the previous step) is smaller than the element a_{2i+1} then swap them.

At this point we have effectively placed the largest among $x_{2i-1}, x_{2i}, x_{2i+1}$ at the location $2i$ and we can proceed to the next even element $a_{2(i+1)}$.

See Listing 9.3 for a possible implementation of this idea.

```
1  std::vector<int> wave_linear(const std::vector<int> &A)
2  {
3      if (A.size() <= 2)
4          return A;
5
6      std::vector<int> B(A);
7      for (size_t i = 0; i < B.size(); i += 2)
8      {
9          if (i > 0 && B[i - 1] > B[i])
10             std::swap(B[i - 1], B[i]);
11
12             if (i < B.size() - 1 && B[i + 1] > B[i])
13                 std::swap(B[i + 1], B[i]);
14     }
15     return B;
16 }
```

Listing 9.3: Linear time solution to the wave array problem.

Note that the code above performs some checks on the corner elements so that we do not perform out-of-bound accesses. This solution is optimal as it runs in $O(n)$ space and time.

As with the Brute Force sorting solution above unfortunately, the linear time solution also does not work when the lexicographical minimum arrangement should be returned so we will address this common variation now.

9.4 Common Variations - Return the lexicographically smallest

One of the most common variations of this problem is the one where you are required to return the lexicographically minimum wave-like arrangement of A . In this exercise, you

have the chance to apply everything we have learned about this problem so far and write an efficient solution for this variation.

9.4.1 Problem statement

Problem 13 Given an array A of n integers, arrange the numbers in a wave-like fashion (see Section 12 for a definition of wave-array). If there are multiple valid answers, the function returns the one that is lexicographically smallest.

1. $x_0 \geq x_1 \leq x_2 \geq x_3 \leq x_4 \geq \dots$ where $x_{2i-1} \geq x_{2i} \leq x_{2i+1}$
2. $x_1 \leq x_2 \geq x_3 \leq x_4 \geq x_5 \leq \dots$ where $x_{2i-1} \leq x_{2i} \geq x_{2i+1}$

■ **Example 9.4**

Given $A = \{1, 2, 3, 4\}$ the function returns $\{2, 1, 4, 3\}$. Notice that the sequence $\{4, 1, 3, 2\}$ is also a valid wave-array but it is not lexicographically minimal. ■



9.5 Conclusions

10. First positive missing

Introduction

This chapter addresses a fairly common problem posed during on-site interviews for which there are a number of solutions which vary widely in terms of time and space complexity.

Finding what most interviewers would consider the “best” solution in terms of asymptotic complexity can be challenging therefore it needs a more in depth analysis than some other problems posed in this book.

It is common for interviewers to pose this problem using a short and purposely vague statement. It is, therefore, important to ask questions to ensure all aspects of the problem are well understood before attempting a solution. ^①

10.1 Problem statement

Problem 14 Write a function that, given an unsorted integer array A , returns the smallest positive integer not contained in A .

■ **Example 10.1**

Given $A = \{1, 0, -1, -2\}$ the answer is 2. ■

■ **Example 10.2**

Given $A = \{2, 3, -7, 6, 8, 1, -10, 15\}$ the answer is 4. ■

■ **Example 10.3**

Given $A = \{1, 0, -1, -2\}$ the answer is 2. ■

10.2 Clarification Questions

Q.1. Are the input numbers always positive?

No, the array contains positive and negative numbers.

Q.2. Are all the elements distinct?

No, the array might contain duplicates.

Q.3. Can the input array be modified?

Yes.

Q.4. Can the size of the array be zero? In other words, can the array be empty?

No, the input array contains at least one element.

Q.5. Is 0 a valid output?

No, only strictly positive numbers should be returned.

^①We think a good way of doing this is to repeat out loud a summary of your understanding of the problem to the interviewer.

10.3 Discussion

This problem has a solid real-life application. Consider how an OS might assign PID^②s to processes. One approach would be to keep a list of all the PIDs for all the processes running, and once a new one is fired up, the OS will assign it the smallest PID **not already assigned** to any other process.

In a highly dynamic environment like the OS with thousands of applications active at the same time; the focus of the solution should be speed as you want the process to be up and running as quickly as possible.

10.3.1 Brute-force

One of the simplest approaches is to simply search *A* for the missing number incrementally, starting from 1. The practical reality of this approach is that we have to perform a search operation in *A* for each number from 1 onward **until the search fails**. This algorithm is always guaranteed to return the smallest missing number given that we perform the searches in order, with the smallest numbers being searched first.

Listing 10.1 shows an implementation of this using `std::find()` as a means to do the actual search in the *A*.

```
1  int first_positive_missing_bruteforce1(const std::vector<int> A)
2  {
3      int ans = 0;
4      // until ans is found
5      while (std::find(begin(A), end(A), ans) != end(A))
6          ans++;
7      return ans;
8  }
9
10 int first_positive_missing_bruteforce2(const std::vector<int> A)
11 {
12     for (int i = 0;; i++)
13     {
14         // not found
15         if (std::find(begin(A), end(A), i) == end(A))
16             return i;
17     }
18 }
```

Listing 10.1: Two bruteforce solution implementations the problem of finding the smallest missing positive integer in an array.

This is often considered a poor solution (as a rule of thumb, in the context of coding interviews, brute-force solutions always are) as it has a complexity of $O(n^2)$ time and $O(1)$ space.

It does, however, have some advantage in being easy and fast to write, and avoiding implementations mistakes due to simple logic and small amount of code involved.

10.3.2 Sorting

The second most intuitive approach (after brute-force) is sorting the input as having the numbers sorted is helpful for easily coming up with a faster solution.

^②A number used (in UNIX part of the process control block) to uniquely identify a process within the OS. In Unix, process IDs are usually allocated on a sequential basis, beginning at 0 and rising to a maximum value (usually 65535) which varies from system to system. Once this limit is reached, allocation restarts at zero and again increases. However, for this and subsequent passes any PIDs still assigned to processes are skipped.

When the A is sorted, we know the positive numbers in it will all be appearing **in an ordered fashion** from a certain index $k \geq 0$ onwards (the positions from index 0 to $k - 1$ are occupied by negatives or zeros).

We also know that, if no number is missing in $A[k \dots n - 1]$, then we would expect to see:

- $A[k] = 1$
- $A[k + 1] = 2$
- $A[k + 2] = 3$
- ...
- $A[n - 1] = n - k + 1$

i.e. all numbers from 1 onward appear in their natural order $(1, 2, 3, \dots, (n - k + 1))$ from the cell at index k to the end of A . If any of these numbers are missing then we would not be able to see such a sequence.

The goal of this problem is to find the first number that is missing from that sequence. We can do that by finding the first element among $A[k \dots n - 1]$ where the condition $A[k + i] = i + 1$ with $(i = 0, 1, 2, \dots)$ is **false**. When this happens, we can conclude the missing number is $i + 1$. If such a cell does not exist (every cell satisfies the condition above), then we know that the missing number is $A[n - 1] + 1$.

For example, consider the array $A = \{9, -7, 0, 4, 5, 2, 0, 1\}$. When sorted, the array becomes $A = \{-7, 0, 0, 1, 2, 4, 5, 9\}$. The positives start at index $k = 3$:

- $A[3 + 0] = 1$ (test passes)
- $A[3 + 1] = 2$ (test passes)
- $A[3 + 2] = 4$ (**test fails**)

As we can see, the test fails after three tries and therefore we can conclude the missing number is 3.

Now let's consider the array $B = \{3, -7, 0, 4, 5, 2, 0, 1\}$ which is exactly the same as in the previous example, with the exception that we have swapped a 9 for a 3. When sorted, the array becomes $B = \{-7, 0, 0, 1, 2, 3, 4, 5\}$ which contains no gaps between any of the positive numbers. As before, the positives start at index $k = 3$ but this time every element passes the test:

- $A[3 + 0] = 1$ (test passes)
- $A[4 + 1] = 2$ (test passes)
- $A[4 + 2] = 3$ (test passes)
- $A[4 + 3] = 4$ (test passes)
- $A[4 + 4] = 5$ (test passes)

We can clearly see that the missing number is $6 = A[8] + 1 = A[n - 1] + 1$.

An implementation of this idea is shown in Listing 10.2.

```

1  int first_positive_missing_sorting(std::vector<int> A)
2  {
3      std::sort(begin(A), end(A));
4
5      auto it =
6          std::find_if(begin(A), end(A), [](const auto &x) { return x >= 0; });
7
8      int expected = 0;
9      while (it != end(A) && (*it) == expected)
10     {
11         expected++;
12         it++;
13     }
14     return expected;

```

15 }

Listing 10.2: Solution to the problem of finding the smallest missing positive integer in an array.

Note that:

- the iterator `it` always points at the currently evaluated element. It is initialized to either:
 - the **smallest positive** in the sorted array;
 - to one element past the end of the array if no positives are present.`it` is moved to its initial location by using the `std::find_if` function from the STL which runs in linear time. We might have used binary search to perform this task, but that would not have significantly helped in lowering the overall asymptotic time complexity as the sorting operation itself costs $O(n\log(n))$ and the subsequent `while` loop runs in linear time.
- `expected` is a variable holding the value that is expected to be found where `it` is pointing to (the value $i + 1$ mentioned above).
- if the `while` runs to completion because we have examined every element of A (`it == std::end(A)`) then `expected` points to `A.front()+1`.
- if no positives are present, then the `while` does not even run once and 1 is returned.

This is considered a good solution with an overall time and space complexity of $O(n\log(n))$ and $O(n)$ respectively. It is, however, not optimal, solutions with better time and space complexities exist.

10.3.3 Linear time and space solution

Examining the problem more closely we immediately notice that the missing number will always be in the range $[1, n]$, where n is the size of the input array. **Why is this the case?** We can draw this conclusion by considering which input can possibly lead to the largest possible output: among all possible arrays of size n , only one configuration leads to the highest missing number: $A = \{1, 2, 3, 4, \dots, n\}$ i.e. the configuration where all numbers from 1 to n are present. All the other configurations contain duplicates, negative or numbers higher than n which forces the input to have “holes” (i.e. missing numbers in the range $[1, n]$).

This fact can be exploited to keep track of which positive numbers from 1 to n are present in A . We can use an array of booleans flags of size n to store this information. Therefore, all we have to do is set the x^{th} flag to true for each number x in A in the range $[1, n]$. Eventually, this array of flags contains the answer, which can be found by scanning through it linearly to find the first **false** element which signals that this is the first missing element.

An implementation of the idea above is found in Listing 10.3.

```
1 int first_positive_missing_linear_space(std::vector<int> A)
2 {
3     std::vector<bool> F(A.size(), false);
4
5     for (const auto &x : A)
6     {
7         if (x >= 0 && x < A.size())
8             F[x] = true;
9     }
10    for (size_t i = 0; i < F.size(); i++)
11        if (!F[i])
12            return i;
```

```

13
14     return A.size();
15 }

```

Listing 10.3: Solution to the problem of finding the smallest missing positive integer in an array.

The code in Listing 10.4 works in two phases:

1. For each number x of A in the range $[1, n]$, we remember we have found it by marking the cell at index $x-1$ in F .
2. We find if exists the first cell in F containing false which means the corresponding element was not found in A . If a false cell does not exist then we can conclude the missing number is $n + 1$.

This approach is considered good and it has time and space complexity both equal to $O(n)$.

As an alternative, instead of an array, we can use a hashmap to keep track of which element has already been seen as the array is scanned from left to right. This might have advantages in some cases, especially when n is large but there are many duplicates in A .

10.3.4 Linear time and constant space solution

As mentioned above, the optimal solution does not use any additional space but shares the idea of keeping track of which element has been found with the solution described in Section 10.3.3.

In order to avoid using additional space we have to somehow implement the functionality the array F (in the code above) provides using the input array itself. Doing so sounds harder than it is, especially before we realize that we can store the information about a positive number being present

The previous solutions have already demonstrated that we can safely ignore every negative number in A , and that the largest output we can ever hope to get is always less than the number of positives in A . For instance, if we have an array of size n with x negatives and y positives ($n = x + y$), then the largest possible output we can get is: $y + 1$.

As an example, let's consider the array $A = \{-1, -2, -3, 0, 1, 2, 3\}$ which has $x = 4$ negatives and $y = 3$ positives ($n = 7$). We can see that the missing number is $4 = y + 1$ and also that, if we substitute any positive (or negative or zeros for that matter) number with a different positive, the output of the function will **not increase**.

The idea is to loop through A and, for each number $x > 0$, store the information about x being present in A in some cell of A itself. But which one? What we want is to have a mapping from the positives in A to indices of A . We can use this mapping to choose the cell in which we remember whether a number is present or not by changing its sign.

Note that, if A contains y positive, then we have y cells to which we can change the sign from positive to negative. In our quest to create this 1-to-1 mapping one problem, the problem we face is that A is unsorted and positives and negatives are all shuffled together. Therefore, the first step would be to rearrange the elements so that all the positives appear before all the negatives. In this way creating the mapping becomes much easier. If all y positive are located from index 0 to $y - 1$, then every time we process an element of x of A , which value is between 1 and y ($1 \leq x \leq y$), we can change the value of the cell at index x to remember the fact the value x is present in A . The remaining problem at this point is to rearrange A so that all positives appear before all negatives and zeros in an efficient manner.

Given an unsorted array, we can rearrange its elements so that all positives are before all negatives by using a two-pointer technique where we use - unsurprisingly - two pointers

s and e pointing initially to index 0 and $n - 1$, respectively. The idea is to keep moving s towards the end of the array and e towards the start, until s points to a positive and e points to a negative. At this point, s points to a value that should appear after the element pointed by e and therefore we can swap those values. If we keep repeating this process until $s \geq e$, eventually all the pairs of misplaced elements (a negative appearing before a positive) are swapped and the final array is arranged so that all negatives appear in the first x positions of the array.

Consider for instance the array $A = \{-1, 1, 2, -20, 3, -3\}$. Initially $s = 0$ and $e = 6$. We first move s to the first positive which happens to appear at index 1. Similarly, we move e to the left towards the first negative which appears at index 6. The elements pointed by s and e are swapped, and $A = \{-1, -3, 2, -20, 3, 1\}$.

The same process is repeated and after they are moved, $s = 2$ and $e = 4$. The values they point to are swapped leaving $A = \{-1, -3, 0, -22, 3, 1\}$. When the pointers are next moved, they would cross, and this signals the rearrangement is finally complete. It is important to note that the aim of this process is not to sort the array, but to simply make all the negatives appear before all the positives (therefore still allowing the positives and the negatives to appear in any order).

Once all the numbers in A are processed this way, similar to what we did in step 2 of the solution running in linear time and space (where we used the array 'F'), we can scan the portion of A from index 0 to y looking for positives. If we find one at index k , it means that the element $k + 1$ is missing from the array as, if it was present, it would have undergone a sign change. If all those cells contain negative, then it means that the missing number is $n + 1$.

This idea is implemented in Listing 10.4 shown below.

```

1  int divide_pos_neg(std::vector<int> N)
2  {
3      int s = 0;
4      int e = N.size() - 1;
5      while (s <= e)
6      {
7          while (s <= e && N[s] > 0)
8              s++;
9          while (s <= e && N[e] <= 0)
10             e--;
11         if (s >= e)
12             break;
13
14         std::swap(N[s], N[e]);
15     }
16     return s;
17 }
18
19 int first_positive_missing_constant_space(std::vector<int> N)
20 {
21     const int num_pos = divide_pos_neg(N);
22     for (int i = 0; i < num_pos; i++)
23     {
24         const int ni = abs(N[i]);
25         if (ni > 0 && ni - 1 < num_pos)
26             N[ni - 1] *= -1;
27     }
28
29     for (int i = 1; i < num_pos; i++)
30         if (N[i - 1] >= 0)
31             return i;

```

```
32     return num_pos + 1;
33 }
```

Listing 10.4: Linear time and constantspace solution to the problem of finding the smallest missing positive integer in an array.

The two phases of this solution are packaged into two functions:

- `first_positive_missing_constant_space` builds on it and uses the rearrangement to mark the presence of any element x in the range $[1, y]$ by changing the sign of the cell at index $x - 1$. When it is done with it, it proceeds in finding the answer by searching for the smallest index in i containing a positive.
- `divide_pos_neg` is responsible for rearranging the input array as discussed above

The complexity of this approach is linear in time and constant in space which is optimal.

11. Exponentiation

Introduction

In this chapter we will review the common problem of implementing a function in order to calculate the power of an integer. Although it is relatively easy to find a solid solution that works in linear time which, if implemented correctly can be sufficient for an interviewer, if the goal is to impress we should look to deliver something more sophisticated and efficient. The key to creating this efficient, and impressive solution is the well-refined concept of *exponentiation by squaring*, which can be applied not only to integers but also to many other mathematical objects, such as polynomials or matrices (reinsert the bracket that I deleted by accident).

11.1 Problem statement

Problem 15 Implement a function that given two positive integers n and k calculates n^k .

■ **Example 11.1**

Given $n = 2$ and $k = 3$ the function returns 8. ■

■ **Example 11.2**

Given $n = 5$ and $k = 2$ the function returns 25. ■

11.2 Clarification Questions

Q.1. Should the function handle the case where $k = 0$?

Yes $k = 0$ is a valid input.

Q.2. Should the function handles integer overflow?

No overflow should not be accounted for. ^①

11.3 Discussion

Exponentiation, the calculation of powers by means of performing consecutive multiplications, is a well-understood operation and the method below is a direct application of this concept. It involves two numbers n (the base) and k (the exponent) and it is usually written as n^k (pronounced as *" n raised to the power of k "* or *the k^{th} power of n *):

^①This clarification question may lead to a follow-up discussion on how such scenarios can be handled, for example:

- how to represent and manipulate infinite precision numbers,
- examples of production libraries providing infinite precision, etc. (the GMP library[13] probably being the best known).
- how overflow errors can be handled? (Exceptions, error codes, UB[12]?)

$n^k = n \times n \times n \dots \times n$ where we multiply the base exactly k times with itself to obtain the result.

This simple algorithm embedded in the definition can be coded in just a few lines and a possible iterative implementation is shown in Listing 11.1.

```
1 unsigned exponentiation_linear(const unsigned n, unsigned k)
2 {
3     unsigned ans = 1;
4     while (k > 0)
5     {
6         ans *= n;
7         k--;
8     }
9     return ans;
10 }
```

Listing 11.1: Iterative linear time solution.

The code calculates the answer, stored in the variable `ans`, by multiplying `ans` itself and `n`, `k` times as we would do on a blackboard and exactly as stated in the definition of exponentiation above. Note that:

- Listing 11.1 assumes $k \geq 0$,
- when $k = 0$ the while loop is not executed at all and the final result is 1 (which is correct as the result of raising any positive to the power of 0 is 1.)
- the time complexity is $O(k)$ as the while loop decreases the value of k by 1 at each iteration;
- the space complexity is constant.

11.3.1 Using recursion

When discussing this solution, the interviewer may explicitly request a recursive solution. The definition of power by repetitive multiplication provides all the information needed to write such a solution, and it should be noted that we can regroup the operations in the definition above so that: $n^k = n \times n^{k-1}$ which shows that the k^{th} power of n is function of its $(k-1)^{th}$ power.

Listing 11.2 shows a recursive code solution.

```
1 unsigned exponentiation_linear_recursive(const unsigned n, unsigned k)
2 {
3     if (k == 0)
4         return 1;
5     // n * n^{k-1}
6     return n * exponentiation_linear_recursive(n, k - 1);
7 }
```

Listing 11.2: Recursive linear solution.

Listing 11.2 runs in $O(k)$ time and space as:

- the base case $k = 0$ is reached only after k steps as each recursive call decreases the value of k by 1 and each call costs constant time;
- we need to use space for the activation record of each of the $O(k)$ recursive calls.

11.3.2 Binary fast exponentiation

The recursive solution above was based on the fact that the k^{th} power of n is function of its $k-1^{th}$ power. We obtain this result by simply regrouping the definition of exponentiation given in the introduction of this chapter. This is not, however, the only possible way of

regrouping these multiplications. For instance, we can calculate n^k as $n^4 \times n^{k-4}$. This is possible thanks to the following two well-known properties of powers:

1. if $x + y = k$ then, $n^k = n^x n^y = n^{x+y}$
2. if $x \times y = k$ then, $n^k = (n^x)^y$

The question then, is how can we use these properties to speed up the exponentiation process? Let's consider what happens when k is a power of 2. All it takes to calculate n^k is knowing the value of $n^{\frac{k}{2}}$, and in turn, all it takes to calculate $n^{\frac{k}{2}}$ is $n^{\frac{k}{4}}$ and so on, ... Taken to its full conclusion, eventually the exponent would be one, and at that point we know the answer.

Given that, at every step the exponent is divided by 2, after $\log_2(k)$ steps we have all the input needed to calculate the answer as $n^k : n^{\frac{k}{2}} \times n^{\frac{k}{2}} = (n^{\frac{k}{4}} \times n^{\frac{k}{4}}) \times (n^{\frac{k}{4}} \times n^{\frac{k}{4}}) = ((n^{\frac{k}{8}} \times n^{\frac{k}{8}}) \times (n^{\frac{k}{8}} \times n^{\frac{k}{8}})) \times ((n^{\frac{k}{8}} \times n^{\frac{k}{8}}) \times (n^{\frac{k}{8}} \times n^{\frac{k}{8}})) = \dots$

We have effectively reduced the time complexity down to $\log_2(k)$ **when k is a power of two**. But what about the general case? Note that we can split k in half every time k is even (and that when k is a power of 2 all the intermediate division lead to an even number) As such, we can start by applying the idea above only when k is even and relying on the $(1, k-1)$ split in all the other cases.

In other words, the idea is to calculate the answer by multiplying two smaller powers: n^p and n^q with $p, q < k$. The value of p and q depends on the parity of k (whether k is even or odd) and more in particular we want:

$$n^k = \begin{cases} p = q \implies n^{\frac{k}{2}} \times n^{\frac{k}{2}}, & \text{if } k \text{ even} \\ p = 1, q = k - 1 \implies n \times n^{k-1}, & \text{if } k \text{ odd} \end{cases}$$

This allows for the number of multiplication to be reduced by half each time that k is even but also, crucially, when k is odd as n^{k-1} can be calculated by reducing the number of multiplications by half because $k-1$ is even.

Clearly, this approach is inherently recursive and can be easily coded in this way as shown in Listing 11.3.

```

1 unsigned exponentiation_fast(unsigned n, unsigned k)
2 {
3     if (k == 0)
4         return 1;
5     if (k % 2 == 0)
6     {
7         const auto ans = exponentiation_fast(n, k / 2);
8         return ans * ans;
9     }
10    return n * exponentiation_fast(n, k - 1);
11 }
```

Listing 11.3: Recursive $O(\log_2)$ solution to the exponentiation problem.

The code works similarly to the linear time recursive solution at [INCLUDE REFERENCE?] except for the special treatment k receives when it is even.

This solution has a time complexity of $\log_2(k)$. The basic idea is that in the worst-case scenario, for every two invocations of the `exponentiation_fast` function we split the value of k in half anyway.

11.3.3 Iterative solution using bit manipulation

Another way to tackle this problem is by looking at the binary representation of the exponent $k = b_0 \times 2^0 + b_1 \times 2^1 + \dots + b_l \times 2^l$ where b_i as a binary digit. When plugging k

into the formula for the calculation of n^k , the following is obtained (and by applying the properties of powers shown above):

$$\begin{aligned}
 n^k &= n^{b_0 \times 2^0 + b_1 \times 2^1 + \dots + b_l \times 2^l} \\
 &= n^{b_0 \times 2^0} \times n^{b_1 \times 2^1} \times \dots \times n^{b_l \times 2^l} \\
 &= (n^{2^0})^{b_0} \times (n^{2^1})^{b_1} \times \dots \times (n^{2^l})^{b_l} \\
 &= (n^2)^{b_0} \times (n^2)^{b_1} \times \dots \times (n^2)^{b_l}
 \end{aligned}$$

It is clear that the term i^{th} in the final multiplication chain contributes to the final result only when the corresponding value of b_i is set to 1, because if it is 0 then the term contribution is 1 (which is the neutral element for multiplication). Additionally, as i increases, n gets squared at each step, as i is in the formula an exponent for 2.

This approach can be used to implement a fast exponentiation iterative solution by looking only at the bits of k , and using the formula above to calculate the answers accordingly. Listing 11.4 and 11.5 shows two possible ways of doing this.

```

1 unsigned exponentiation_fast_iterative_simple(unsigned n, unsigned k)
2 {
3     if (k == 0)
4         return 1;
5
6     int ans = 1;
7     for (int i = 0; i < std::numeric_limits<int>::digits; i++)
8     {
9         const bool bit = (k >> i) & 1;
10        if (bit)
11            ans *= n;
12        n *= n;
13    }
14    return ans;
15 }

```

Listing 11.4: Logarithmic solution based on the analysis of the bits of the exponent.

The code in Listing 11.4 works by keeping track of two quantities:

1. `ans`: a variable holding the partial calculations of the answers along the multiplication chain and,
2. `n` which stores the value of n raised to 2 to the power of the current iteration index i . The value of n is squared at each iteration.

The loop is used to inspect the i^{th} bit of k and when it is set, `ans` is multiplied with the current value hold by `n`.

The complexity of this approach is $O(\log_2(k))$ as the algorithm does not perform more iterations than the number of bits of the exponent k . Thus, at most $\lfloor \log(k) \rfloor$ squarings and multiplications are performed. If native types like `unsigned` are used, then the complexity is constant as these types have a finite precision and therefore a fixed number of bits (the same reasoning holds for all the solutions discussed in this chapter).

Listing 11.5 shows an alternative implementation which is slightly more sophisticated and efficient as it stops as soon as it notices there is no bit set in k anymore (when k is zero), while Listing 11.4 iterates blindly over all the bits of k . In practice, this might not be a real or even measurable advantage.

```

1 unsigned exponentiation_fast_iterative(unsigned n, unsigned k)
2 {
3     if (k == 0)
4         return 1;
5

```

```

6  int ans = 1;
7  while (k > 1)
8  {
9      if (k & 1) // bit set
10         ans *= n;
11
12         k >>= 1;
13         n *= n;
14     }
15     return n * ans;
16 }

```

Listing 11.5: Alternative implementation of Listing 11.4.

Finally, we should note that, because one of the constraints of the problem is that overflow is guaranteed to never occur, we know that k is a relatively small number and we can safely assume it is smaller than 64, otherwise we would need data types with a capacity of more than 64 bits to store the answer. Under this constraint, the logarithmic time solution may not provide a measurable speed-up or may even be slower, due to the fact that the linear time solution features a simple loop with basic operations that can be well optimized by the compiler optimizer.

It is worth remember that, as discussed in the introduction to this Chapter, exponentiation not only applies to numbers but that can be applied to any larger class of objects such as matrices. As such the codes discussed in above can easily be extended so that they work on any of these types of objects providing the `operator*()`, `operator>>()` and the `operator&()` operators by using `templates`.

11.4 Common Variations

11.4.1 Fibonacci numbers - Problem statement

Problem 16 Write a function that given an integer n returns the n^{th} Fibonacci number. Your solution should run in $O(\log(n))$ time.

■ Example 11.3

Given $n = 44$ the function returns 701408733

12. Largest sum in contiguous subarray

Introduction

When it comes to coding interviews, dynamic programming questions are among the most feared and challenging: one of the most famous and iconic of this category is the *Largest sum in contiguous subarray* problem. This is not only a still frequently asked question, but it also has many real-life applications in science including, but not limited to, genomic sequence analysis (to identify certain protein sequences) and computer vision (to identify the brightest or darkest part of an image).

In this chapter, we will investigate how to efficiently solve this problem and its most popular variations, starting from the inefficient brute-force solution, which will serve as a starting point for our journey towards efficiency. This solution will then be improved by using the concept of avoiding duplicate calculation which is central to DP[1] (see Section 64). Finally, we will study and develop an intuitive idea of what is considered to be the reference algorithm for this problem which allows us to solve this problem efficiently and elegantly.

12.1 Problem statement

Problem 17 Write a function that finds the largest sum of a contiguous sub-array (containing at least one element) within an array A of length n .

Formally, the task is to find two indices $0 \leq i \leq j < n$ s.t. the following sum is maximized:

$$\sum_{x=i}^j A[x] = A[i] + A[i+1] + \dots + A[j]$$

■ **Example 12.1**

Given $A = \{-2, -5, \underline{6, -2, -3, 1, 5}, -6\}$ then the answer is 7 which can be obtained by summing all elements from index 2 to 6 i.e. $\sum_{i=2}^7 A[i] = 7$ ■

■ **Example 12.2**

Given $A = \{-2, 1, -3, \underline{4, -1, 2, 1}, -5, 4\}$ then the answer is 6 which can be obtained by summing all elements from index 3 to 6 i.e. $\sum_{i=3}^6 A[i] = 6$ ■

12.2 Clarification Questions

Q.1. Are the elements all positive or negative?

No, the input numbers can be either positive, or negative.

Q.2. Is the array sorted?

No, the array is not sorted.

Q.3. Is it guaranteed the final result fits an `int`?

Yes, you should not worry about integer overflow.

12.3 Discussion

A couple of important observations can be made after reading the problem statement:

- If the array only contains non-negative numbers, then the problem becomes trivial, because the answer is the sum of the whole array;
- If, on the contrary, the array contains only numbers lower than or equal to zero, then the answer coincides with the largest number in A (if the constraint on the non-empty size of the sub-array is relaxed, then the answer in this case is always zero.);
- The answer is unique, but more than one sub-array might sum up to that value.

12.3.1 Brute-force

One way to tackle this problem is to look at the sum of all possible sub-arrays and return the largest. The idea is that, for all elements $A[i]$, the sum of **all** sub-arrays having it as starting element can be calculated as shown in Listing 12.1.

```
1 int max_sum_contiguous_subarray_bruteforce(const std::vector<int> &A)
2 {
3     int ans = std::numeric_limits<int>::min();
4     for (auto i = begin(A); i != end(A); i++)
5     {
6         for (auto j = i; j != end(A); j++)
7         {
8             const int subarray_sum = std::accumulate(i, j + 1, 0);
9             ans = std::max(ans, subarray_sum);
10        }
11    }
12    return ans;
13 }
```

Listing 12.1: Cubic time brute-force solution

The code in Listing 12.1 enumerates all possible pairs of indices $i < j$, and for each of them it calculates the sum of the elements of A between i and j . Among all of those sums, the largest is returned.

This approach is correct but it is unnecessarily slow, and it has a time complexity of $O(n^3)$: there are $O(n^2)$ ordered pairs (i, j) each identifying a sub-array, and calculating the sum of a single sub-array costs $O(n)$ (the call to `std::accumulate`), for a grand total of $O(n^3)$. This is considered a poor solution: it is quite far off from the optimal linear time complexity solution that exists for this problem.

12.3.2 Improving the Brute-force

We can improve the brute-force solution proposed in the Section 12.3.1 above if we explicitly avoid calculating the sum of a sub-array over and over again. Given two indices i and j the corresponding sub-array sum can be obtained in constant time by using a pre-calculated prefix sum (see Section ??) of the input array. This allows for constant-time computation of the sum of any sub-array in A . Given an array Y containing the prefix sum for the array A then the sub-array sum between indices i and j is equal to: $Y[j] - Y[i - 1]$ where $Y[-1] = 0$.

The code below shows how this idea can be used to bring the time complexity of the brute-force solution above down to $O(n^2)$.

Notice how the call to `std::accumulate` is substituted with a simple $O(1)$ operation on the prefix sum that is pre-calculated using the function `prefix_sum` (which crucially runs in linear time).

```
1 std::vector<int> prefix_sum(const std::vector<int> &A)
2 {
3     assert(A.size() > 0);
4
5     std::vector<int> Y(A.size());
6     Y[0] = A[0];
7     for (size_t i = 1; i < A.size(); i++)
8         Y[i] = Y[i - 1] + A[i];
9
10    return Y;
11 }
12
13 int max_sum_contiguous_subarray_bruteforce_prefix_sum(const std::vector<int> &A
14 )
15 {
16     const std::vector<int> Y = prefix_sum(A);
17
18     int ans = std::numeric_limits<int>::min();
19     for (size_t i = 0; i < A.size(); i++)
20     {
21         for (size_t j = i; j < A.size(); j++)
22         {
23             int subarray_sum = Y[j]; // 0 to j
24             if (i > 0)
25                 subarray_sum -= Y[i - 1]; // 0 to i
26
27             ans = std::max(ans, subarray_sum);
28         }
29     }
30     return ans;
31 }
```

Listing 12.2: Brute-force quadratic time and linear space solution using prefix-sum.

Despite the dramatic improvement in time complexity obtained with this new solution, the code in Listing 12.2 is still considered a rather poor solution: storing the prefix sum costs linear space, and if we consider how common this question is during interviews, the interviewer is likely expecting the known linear time and constant space solution.

12.3.3 Kadane's Algorithm

To solve this problem efficiently by using a dynamic programming approach, there is an ad-hoc developed algorithm named *Kadane's algorithm*: in its simplest form uses an additional array B storing at each position j the largest sum for a sub-array ending (and including) at $A[j]$. Once B is filled, the solution to the problem simply boils down to finding the maximum element in B . An implementation of this algorithm is shown in Listing 12.3.

```
1 int max_sum_contiguous_subarray_kadane_space(const std::vector<int> &A)
2 {
3     std::vector<int> B(A.size(), std::numeric_limits<int>::min());
4
5     B[0] = A[0];
6     for (int i = 1; i < A.size(); i++)
7         B[i] = std::max(A[i], B[i - 1] + A[i]);
8
9     return *max_element(begin(B), end(B));
10 }
```

Listing 12.3: Linear space Kadane's algorithm.

The core of this solution is in how B is filled up. DP is used to calculate the value of each element $B[i]$ by reusing the information in the cell at index $i - 1$. Clearly $B[0]$ cannot be anything different than $A[0]$ (there is only one sub-array ending at the first element of A), while for all the other locations we use `std::max(A[i], B[i-1]+A[i])` to decide the value of $B[i > 0]$. The function call to `std::max` really means that the maximum sub-array sum ending at (and including) $A[i]$ comes from either:

1. the sub-array starting and ending at index i i.e. only containing $A[i]$;
2. extending the best sub-array ending at index $i - 1$ by adding $A[i]$ to it. Notice that the sum of the best sub-array ending at $i - 1$ is already computed and stored in $B[i - 1]$. By doing so we are effectively avoiding a lot of re-computation, and this is the reason why Kadane's algorithm is so fast.

If we think about it, it makes sense to construct B this way. After all, when processing an element $A[i]$ we can either use it to extend a sub-array ending at index $i - 1$ (and if we are going to extend one, we are better off extending the one which gives us the largest sum up to that point) or start a new sub-array from the cell at position i . We choose one of the two options based on which choice leads to the largest value.

Figure 12.1 shows an example of execution of this idea on the array $A = \{-2, -5, 6, -2, -3, -2, 5, 2\}$ where it is depicted for each of the eight steps of the algorithm how the value for the corresponding cells of B are calculated as well as which cells of A contribute to it.

12.3.3.1 Linear time and constant space Kadane's algorithm

But, is the additional space used by B really necessary? A closer look at the implementation of the linear space Kadane's algorithm above provides the answer: no.

In fact, every value of B is only used **once**, and then ignored for the rest of the execution (count how many times $B[i]$ is read). Thus, the algorithm can be modified to take advantage of this fact, so it only uses a single variable to store the latest calculated value of B i.e. the value for the index $i - 1$ as shown in Listing 12.4.

```

1  int max_sum_contiguous_subarray_kadane(const std::vector<int> &A)
2  {
3      assert(A.size() > 0);
4
5      int ans          = A[0];
6      int max_ending_here = A[0];
7      for (int i = 1; i < A.size(); i++)
8      {
9          max_ending_here = std::max(A[i], max_ending_here + A[i]);
10         ans              = std::max(ans, max_ending_here);
11     }
12     return ans;
13 }
```

Listing 12.4: Constant space and linear time Kadane's algorithm

Notice how the variable `max_ending_here` is doing the job that the variable `B[i-1]` was doing in the implementation for the linear space Kadane's algorithm, and also how the final answer is the maximum among all values taken by `max_ending_here`.

The complexity of this approach is $O(n)$ in time and $O(1)$ in space, and very likely is the sort of complexity the interviewer expects.

A	-2	-5	6	-2	-3	-2	5	2
---	----	----	---	----	----	----	---	---

B	-2	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
---	----	-----------	-----------	-----------	-----------	-----------	-----------	-----------

(a) $i = 0$: $B[0]$ is equal to the first element of A

A	-2	-5	6	-2	-3	-2	5	2
---	----	----	---	----	----	----	---	---

B	-2	-5	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
---	----	----	-----------	-----------	-----------	-----------	-----------	-----------

(b) $i = 1$: $B[1]$ is equal to $A[1]$ only. $B[0]$ is negative and therefore $B[0] + A[1] < A[1]$.

A	-2	-5	6	-2	-3	-2	5	2
---	----	----	---	----	----	----	---	---

B	-2	-5	6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
---	----	----	---	-----------	-----------	-----------	-----------	-----------

(c) $i = 2$: $B[2]$ is equal to $A[2]$ only. $B[1]$ is negative and therefore $B[1] + A[2] < A[2]$.

A	-2	-5	6	-2	-3	-2	5	2
---	----	----	---	----	----	----	---	---

B	-2	-5	6	4	$-\infty$	$-\infty$	$-\infty$	$-\infty$
---	----	----	---	---	-----------	-----------	-----------	-----------

(d) $i = 3$: $B[3]$ is equal to $A[2] + A[3]$. $B[2] > 0$ and therefore $B[2] + A[3] > A[3]$.

A	-2	-5	6	-2	-3	-2	5	2
---	----	----	---	----	----	----	---	---

B	-2	-5	6	4	1	$-\infty$	$-\infty$	$-\infty$
---	----	----	---	---	---	-----------	-----------	-----------

(e) $i = 4$: $B[4]$ is equal to $A[3] + A[4]$. $B[3] > 0$ and therefore $B[3] + A[4] > A[4]$.

A	-2	-5	6	-2	-3	-2	5	2
---	----	----	---	----	----	----	---	---

B	-2	-5	6	4	1	-1	$-\infty$	$-\infty$
---	----	----	---	---	---	----	-----------	-----------

(f) $i = 5$: $B[5]$ is equal to $A[4] + A[5]$. $B[4] > 0$ and therefore $B[4] + A[5] > A[5]$

A	-2	-5	6	-2	-3	-2	5	2
---	----	----	---	----	----	----	---	---

B	-2	-5	6	4	1	-1	5	$-\infty$
---	----	----	---	---	---	----	---	-----------

(g) $i = 6$: $B[6]$ is equal to $A[6]$ only. $B[5]$ is negative and therefore $B[5] + A[6] < A[6]$.

A	-2	-5	6	-2	-3	-2	5	2
---	----	----	---	----	----	----	---	---

B	-2	-5	6	4	1	-1	5	7
---	----	----	---	---	---	----	---	---

(h) $i = 7$: $B[7]$ is equal to $A[6] + A[7]$. $B[6] > 0$ and therefore $B[6] + A[7] > A[7]$

Figure 12.1: This figure shows the array B in the Kadane's algorithm is calculated. Gray cells are part of the sub-array that gives the value to the corresponding cell in B coloured in cyan.

12.4 Common Variations

12.4.1 Minimum sum contiguous sub-array

A very common variation of this problem is to find the smallest sum instead of the largest. This variation is very quickly solved by just changing the way the variables in Kadane's algorithm `min_ending_here` (the variable is named **min** instead of max so to reflect the goal of this variation) and `ans` are updated. Since the minimum is to be returned, then they should be updated using `min_ending_here = std::min(A[i] , max_ending_here+A[i]);` and `std::min(ans, min_ending_here)`, respectively.

12.4.2 Longest positive/negative contiguous sub-array

Another common variation of the max/min sum contiguous subarray is to find the longest subarray only containing positive or negative numbers.

The same core behind the Kadane's algorithm can be used for the this variant too. At each iteration i the variable `longest_ending_here` represents the longest positive subarray up to that iteration and including the element $A[i]$ and can be updated as follows:
`longest_ending_here = A[i] >= 0 ? longest_ending_here+1 : 0;`

13. String Reverse

Introduction

Reversing a collection of items or a string is a ubiquitous operation. For this reason it is also a common coding interview question although, due to its simplicity, it is most often used as one of a set of warm-up problems. As the interviewer is expecting us to have seen (or solved) this problem more than once before, the real goal in answering should be to demonstrate how quickly and clearly we can explain our reasoning and present a concise and elegant (as well as correct) solution. In this chapter we will examine how best to do this.

First, it is worth noting that there are really only two variations of this problem:

1. in-place where we are asked explicitly not to use any additional space and to modify the input string;
2. out-of-place, where we are free to return a brand new collection.

If the problem statement is not clear on which variation is being used, this should be clarified as soon as possible.

Let's first consider how popular and difficult this problem is: it is important to focus on getting the solution right at the first try and in a relatively short time frame; in addition, we have to make sure the communication is clear, concise, and precise and that we engage the interviewer into following our reasoning process. He expects us to have seen (or solved) this question already and thus more than on the algorithm itself, in order to be able to stand out among all the other candidates, our communication and implementation should be spot-on.

13.1 Problem statement

Problem 18 Write a function that takes a string s of length n and reverses it.

■ **Example 13.1**

Given $s = "abcde"$ the function produces $s = "edcba"$. ■

■ **Example 13.2**

Given $s = "programming"$ the function produces $s = "gnimmargorp"$. ■

13.2 Clarification Questions

Q.1. Should the function reverse the string in place?

Yes, a copy of the input cannot be created.

Q.2. Is the empty string a valid input?

Yes, the input string might be empty.

Index in Input	Index in Output
0	5
5	0
1	4
4	1
3	2
2	3

Table 13.1: Indices shuffling for the reversal of $s = a_0a_1a_2a_3a_4a_5$.

13.3 Discussion

We have to reverse a string in place but what is the actual meaning of *in-place* here? It means that no auxiliary storage is allowed; that the input itself will be processed and modified by the function; and that it will be eventually transformed into the output. As the original content of the input is lost once the function is terminated, in-place algorithms are also called *destructive*. However, having to use no additional storage space does not literally mean that not even a single additional byte of space can be utilized. This constraint should be interpreted as meaning a copy of the input is disallowed, or that the function should work in $O(1)$ (constant) space.

To develop an intuitive approach to solving this problem it is useful to take a deeper look at what happens to each letter of s once is reversed. For example, consider the string $s = a_0a_1a_2a_3a_4a_5$ which is transformed into $s' = a_5a_4a_3a_2a_1a_0$. The subscript i in a_i identifies the position in which the letter a_i appears **in the original input string** s . The core of the solution is to establish how the letters are shuffled around from their original position to their final location in the reversed string. In order to do this, let's look at how the indices are moved during the reverse process by comparing the positions of a letter in the original and in the reversed string.

Table 13.1 shows how indices of string $s = a_0a_1a_2a_3a_4a_5$ are shuffled around and contains all the information that is necessary to deduce the function that maps the indices of the original string into the indices of the reversed string. An index i gets mapped to an index j s.t. $i + j = n$ (index 2 goes to 3 and $2 + 3 = 5$ for instance). A quick manipulation of that equation shows that j (the index in the reversed string where the letter at index i in the original string is mapped to) is equal to: $j = n - i$. We now know which elements to swap in order to obtain a reversed list. This information can be used to reverse the entire string as shown in Listing 13.1.

```

1 void reverse_string_inplace(std::string &s)
2 {
3     const int len = s.length();
4     for (int i = 0; i < len / 2; i++)
5         swap(s[i], s[len - 1 - i]);
6 }
```

Listing 13.1: Linear time constant space iterative solution.

An important detail to note in Listing 13.1 is how the loop only terminates after $\frac{n}{2}$ iterations. This is necessary because a swap operation on the index $i < \frac{n}{2}$ involves two elements: the element at index i , but also its symmetrical sibling at index $n - i$ in the second half of the string. If the loop would not terminate at $\frac{n}{2}$, then each element a_i would be involved in **two** swap operations. For instance, for the letter at index 0, the following swaps would occur:

- `swap(0, n-1)`

- `swap(n-1,0)`

Applying two (or any even number) swap operations on the same indices is equivalent to a no-op and it results in having the letters involved in the swaps stay at their original locations. Therefore, if the loop does not terminate after $\frac{n}{2}$ iterations, then the function would not modify the original string at all!

This solution is considered good because, besides being short and expressive, it has a time and space complexity of $O(n)$ and $O(1)$ respectively which is optimal.

13.4 Common Variation

13.4.1 Out-of-place solution

Sometimes the interviewer might ask for an easier version of this problem where the memory constraint is relaxed and we are allowed to allocate linear space. In this variation we can simply construct the reversed string by looping the original string backward, as shown in Listing 13.2 and Listing 13.3

```

1
2 std::string reverse_string_outplace_raw_loop(const std::string &s)
3 {
4     std::string ans;
5     ans.reserve(s.size());
6
7     for (int i = s.size() - 1; i >= 0; i--)
8         ans.push_back(s[i]);
9
10    return ans;
11 }
```

Listing 13.2: Linear time and space iterative out-of-place solution using raw loops.

```

1 std::string reverse_string_outplace_iterator(const std::string &s)
2 {
3     std::string ans;
4     ans.reserve(s.size());
5
6     auto it_s = std::prev(end(s));
7     while (it_s >= s.begin())
8     {
9         ans.push_back(*it_s);
10        --it_s;
11    }
12
13    return ans;
14 }
```

Listing 13.3: Linear time and space iterative iterative out-of-place solution using iterators.

13.4.2 Recursive solution

The interviewer may also explicitly ask for a recursive implementation as this problem is well suited for recursion. In fact, a look at the iterative linear time and constant space solution above shows that at any given point in the loop, the status of the string is the following: $a_{n-1}a_{n-2}\dots a_k a_{k+1}\dots a_l a_{l-1}a_{l-2}\dots a_0$. There is always a portion of the string delimited by two indices k and l , $k \leq l$, which is yet to be processed (i.e. with the letters un-swapped). This fact can be used to write a recursive solution; At first $k = 0$ and $l = n - 1$ and the string can be reversed by swapping $a_k = 0$ and $a_l = n - 1$ and by recursively reversing the inner part of the string i.e. in the range $k = 1$ and $l = n - 2$. The

inner part can be reversed using the same logic, and this reasoning can be applied for all the recursive calls right until $k \geq l$. At that point, the function can simply terminate and the input string is reversed successfully. Equation 13.1 formalizes this idea and a possible implementation is shown in Listing 13.4.

$$R(s, k, l) = \begin{cases} \text{swap}(s[k]s[l]) \wedge R(s, k+1, l-1) & \text{if } k \geq l \\ \text{return} & \text{otherwise} \end{cases} \quad (13.1)$$

```

1 void reverse_string_inplace_recursive_helper(std::string &s,
2                                             const int k,
3                                             const int l)
4 {
5     if (k >= l)
6         return;
7
8     swap(s[k], s[l]);
9     reverse_string_inplace_recursive_helper(s, k + 1, l - 1);
10 }
11
12 void reverse_string_inplace_recursive(std::string &s)
13 {
14     reverse_string_inplace_recursive_helper(s, 0, s.size() - 1);
15 }

```

Listing 13.4: Recursive in-place solution.

The complexity analysis for this approach can be a bit controversial, in particular the one concerning space, as we also have to consider the stack space utilized by all the recursive calls, which can theoretically amount to $O(n)$. However, a decent compiler optimizer would optimize it so as to use constant space. It is, however, important to clarify this point with the interviewer when presenting this solution.

Discussing this topic may lead to discussions about recursion, and especially the Tail Call Optimization (TCO)^①, so it is best to be prepared and ready to answer any questions that arise.

The time complexity is linear.

^①TCO (Tail Call Optimization) is the process by which a smart compiler can make a call to a function and take no additional stack space. The allocation of a new stack frame for a function can be avoided because the calling function will simply return the value that it gets from the called function. The most common use is tail-recursion, where a recursive function written to take advantage of tail-call optimization can use constant stack space.

14. Find the odd occurring element

Introduction

In this chapter we will deal with a problem on arrays and on the XOR (also known as *disjunctive-or* and usually identified by the symbol \oplus)^① operation.

There is a very simple, intuitive yet inefficient brute-force solution to the problem, however, as it is conceptually very different from other, faster, approaches it is difficult to use even as a starting point during interview to iteratively improve on to reach optimal time and space complexity. In this instance, it is more effective to begin by reading the problem statement carefully and looking for the right insight immediately rather than getting carried away towards a dead-end by the brute-force approach.

14.1 Problem statement

Problem 19 Write a function that, given an array A of positive integers where all elements except one appear an even number of times, returns the one and only one element appearing an odd number of times.

■ **Example 14.1**

Given the array $A = \{4, 3, 6, 2, 4, 2, 3, 4, 3, 3, 6\}$ the function returns 4 because it appears 3 times while all the other elements appear an even number of times.

■

■

14.2 Clarification Questions

Q.1. Is the input array always valid. Does it always contain only one element appearing an odd number of times?

Yes the input array can be assumed to be valid.

Q.2. Is the range of the input integers known?^②

No it is not. The values of the elements of A is arbitrary.

14.3 Discussion

14.3.1 Brute-force

As mentioned above, the brute-force solution to this problem is very intuitive. We simply have to count the occurrences of each of the elements of A until we find one appearing an odd number of times. Provided that a counting function (which counts the occurrences of a given element in an array) is available, it is only a matter of using that function for all the elements in the array, and return as soon as it returns an odd number.

^① \oplus is a boolean binary operator that returns true only when its two inputs have different values i.e. when one is true and the other is false.

^②This is very good question because if the answer is yes we can use an approach similar to the counting sort to keep track using only constant space and linear time, of the number of times an element appears.

Listing 14.1 shows a possible implementation in C++ which uses the `std::count` function from the STL to count the number of occurrences of a given number in `A`.

```
1 inline constexpr bool is_odd(const int x)
2 {
3     return x & 1; // only odd number have the leftmost bit set
4 }
5
6 int odd_appearing_element_bruteforce_rawloop(const std::vector<int>& A)
7 {
8     for (const auto& x : A)
9     {
10         // count how many times x appears in A
11         const size_t number_occurrences = std::count(begin(A), end(A), x);
12         if (is_odd(number_occurrences))
13             return x;
14     }
15     throw std::invalid_argument(
16         "Invalid input array. No elements appear an odd number of times");
17 }
```

Listing 14.1: Brute force solution using a counting function.

What the code above is really trying to do is **find** the element appearing an odd number of times. Instead of using a raw loop for doing so, the code can be made much more expressive (which is always appreciated by interviewers) by using the standard `find_if` metafunction as shown in the Listing 14.2.

```
1 int odd_appearing_element_bruteforce_standard(const std::vector<int> &A)
2 {
3     return *std::find_if(begin(A), end(A), [&](const auto x) {
4         return is_odd(std::count(begin(A), end(A), x));
5     });
6 }
```

Listing 14.2: Brute force solution using standard libraries functions `std::count` and `std::find_if`.

This is, however, a poor solution as the time complexity is $O(n^2)$ which is far from optimal, while the space complexity is constant.

Note that, in the first brute-force solution (Listing 14.2), we dereference the iterator returned by `find_if` directly without checking if it is valid or not. `find_if(InputIt first, InputIt last, UnaryPredicate p)` returns an iterator to the element satisfying the search criteria `p` (in the form of a lambda) only if such an element exists, and that otherwise it would return `last` which is equal to `std::end(A)`. Dereferencing `std::end(A)` would cause UB, but we can guarantee this won't happen as an odd occurring element is **always present** in `A`^③.

In the second implementation (Listing 14.1), we took a different approach to handling a bad input and decided to explicitly throw an exception in case all elements appear an even number of times or `A` is empty. Even if the interviewer does not ask for this, it is good to show that we thought about this and also that we can handle it without big penalties in expressiveness and performance: we can rest assured this certainly adds a bonus point to our final evaluation. Moreover, we can argue that a throw statement makes it explicit that the function is expecting certain characteristics from the input without incurring performance penalties: ^④ when the input is good (which is safe to assume would be the

^③How could we change Listing 14.2 so that it handles bad input safely?

^④Throwing an exception is cheap when the exception is not raised. This is the case in the main exception model used nowadays (Itanium ABI, VC++ 64 bits Zero-Cost model exceptions)[3].)

majority of the times the function gets invoked).

14.3.2 Linear time and space solution

In order to speed up the process of keeping count of how many times each element appear in the input array, we can adopt a map-like structure where the keys are the numbers in A and the values are integers representing the number of times each element appears in the array. If a hash-based map is used to store this key-value information then this effectively reduces the time complexity of the brute-force approach down to $O(n)$ (on average) at the expense of space that increases to linear as well.

Keeping track of the actual number of times an element appears in A is actually unnecessary as all we need is the information about whether or not the number of times it appears is even or odd. We do not care about the actual number therefore a single bit is sufficient to store this information. The map structure would then associate integers to booleans for a substantial saving in the space used. However big the reduction is the space used remains linear. This idea is implemented in Listing 14.3.

```
1 int odd_appearing_element_linear_space(const std::vector<int>& A)
2 {
3     // true = even
4     // false = odd
5     std::unordered_map<int, bool> M;
6     for (const auto& x : A)
7         M[x] = !M[x];
8
9     for (const auto& kv : M)
10         if (kv.second) // kv is a pair<key, value>
11             return kv.first;
12
13     throw std::invalid_argument(
14         "Invalid input array. No elements appear an odd number of times");
15 }
```

Listing 14.3: Linear time and space solution using a map.

The code works in two phases:

1. the map M is filled in such a way that for each key x the corresponding value is 1 if and only if x appears in A an odd number of times.
2. the map is scanned to find the one element having a value of 1.

The time and space complexity are $O(n)$.

14.3.3 Linear time and constant space solution

However, there is a way to solve this problem in constant space and linear time. This solution is based on the XOR operation which can be thought of as the equivalent of the sum for bits and has several interesting properties that are useful in constructing a solution to this problem:

1. it is a commutative, distributive and associative operation;
2. its neutral element is the 0. What it means is that applying the XOR to a number $x \neq 0$ and 0 always results in x i.e. $x \oplus 0 = x$ and $0 \oplus x = x$
3. xor-ing an element with itself always results in 0 i.e. $x \oplus x = 0$.

The practical consequence of these facts is that when xor-ing an element x with itself an odd number of times, the result is x as $(x \oplus x) \oplus x = (0 \oplus x) = x$, but doing so an even number of times results in 0 because $(x \oplus x) \oplus (x \oplus x) = 0 \oplus 0 = 0$.

This is useful because we know that all input integers except one are occurring an even number of times, therefore when all numbers are xor-ed together, all that is left at

the end is the number appearing an odd number of times: every number except the answer will be xor-ed an even number of times with itself, resulting in 0.

For instance if we try to XOR all the elements of the example 14.1 above where $A = \{4, 3, 6, 2, 4, 2, 3, 4, 3, 3, 6\}$ we obtain: $4 \oplus 3 \oplus 6 \oplus 2 \oplus 4 \oplus 2 \oplus 3 \oplus 4 \oplus 3 \oplus 3 \oplus 6 = 4$. At this point we can use commutativity, associativity and distributivity properties to rearrange it as follows (this would be equivalent to first sort A and then XOR all the elements):

$$\underbrace{(2 \oplus 2)}_0 \oplus \underbrace{(3 \oplus 3 \oplus 3 \oplus 3)}_0 \oplus \underbrace{(4 \oplus 4 \oplus 4)}_4 \oplus \underbrace{(6 \oplus 6)}_0 = 4$$

which clearly show the only value remaining is the one of the element appearing an odd number of times.

An implementation of the idea above is shown in Listings 14.4 where we explicitly loop over A and 14.5 where instead, we use `std::accumulate` to perform the array reduction^⑤.

```
1 int odd_appearing_element_final(const std::vector<int> &A)
2 {
3     int ans = 0; // 0 is the neutral element for XOR
4     for (const int x : A)
5         ans ^= x;
6     return ans;
7 }
```

Listing 14.4: Linear time and constnat space using XOR and a raw loop.

```
1 int odd_appearing_element_final_std(const std::vector<int> &A)
2 {
3     return std::accumulate(std::begin(A),
4                             std::end(A), // range
5                             0,           // initial value
6                             [](const int acc, const int x) {
7                                 return acc ^ x;
8                             } // binary reduction operation
9 );
10 }
```

Listing 14.5: Linear time and constant space solution using XOR \oplus and the `std::accumulate` function from the STL.

Both implementations 14.4 and 14.5 have very similar characteristics in terms of asymptotic performance, as they both use linear time and constant space.

^⑤The process of reducing the array to a single value. Can be thought of as an aggregation of the values of an array that results in a single value. The terms reduction comes from the fact that this operation in its general form can be applied to a multi-dimensional object (imagine a 3D matrix for instance) which are aggregated across a dimension and results in a value without that dimension (into a 2D matrix), practically reducing the number of dimensions of that object by one. In the case of an array, we go from a one-dimensional object to a scalar. Calculating the average, sum, or variance of an array are all examples of reduction operations.

15. Capitalize the first letters of every words

Introduction

Text editing is one of the most basic and common operations computers are used for. There are vast numbers of text editors out there, some of them specialized for a particular type of users (e.g. specific editors for programmers such as 1. vi, 2. GNU Emacs, 3. gedit, 4. TextPad, 5. Visual Studio Code, 6. Eclipse, 7. Sublime Text, 8. Qt Creator, 9. etc.), while others are intended for a broader audience and use case such as MS Word or LibreOffice writer.

A problem that is often posed during coding interviews asks us to put ourselves in the place of a software engineer working on a feature for the newest version of Word that is supposed to make the tedious tasks of converting a particular piece of text into a variant of the title case:^①. The idea is that the user would highlight a portion of text and then have the text modified in place by simply pressing a button instead of manually changing every single letter.

These types of questions often appear as a warm up during the preliminary stages as it isn't inherently complex. As such, the main focus of this chapter is to ensure that the final solution is readable and easy to understand, rather than creating a smarter algorithm.

We will first discuss how the core feature can be implemented and then examine a number of possible implementations and solutions.

15.1 Problem statement

Problem 20 Write a function that given a string *s*, modifies it so that every first letter of every word in *s* is in upper case while leaving the rest of the characters untouched.

■ Example 15.1

Given the string "arturo benedetti michelangeli is the best pianist ever". The function should turn it into: "Arturo Benedetti Michelangeli Is The Best Pianist Ever"^a

■

■ Example 15.2

Given the string:

```
"Truth May Seem BUt Cannot be;  
Beauty brag but 'tis not she;  
TruTh and beauty buried be."
```

The function should turn it into:

```
"Truth May Seem BUt Cannot Be;  
Beauty Brag But 'tis Not She;  
TruTh And Beauty Buried Be."
```

^①All words are capitalized, except non-initial articles like "a", "the", "and", etc.

^aA.B.M (5 January 1920 - 12 June 1995) was an Italian classical pianist considered one of the greatest pianists of all time. He was perhaps the most reclusive, enigmatic and obsessive among the handful of the world's legendary pianists.

15.2 Discussion

This problem does not require coming up with a smart algorithm in order to get the job done. Our goal is to be able to put a working implementation on the table in a reasonably short amount of time and spend the rest of the time polishing it so that it is clean and easy to understand.

What are the practical implications of having to capitalize only the first letter of every word? Let's start by first looking at what makes a letter the first letter of a word. A character is the beginning of a word if any of the following is true:

- is not space and it is preceded by a space;
- is not space and it is the first character of the string.

Any other character is either space (for which the notion of lower/upper case is not defined) or is in the middle of a word. Given this definition, all we need to do to solve this problem is to search for any character in the input string satisfying any of the criteria above as shown in Listing 15.1.

```
1 void capitalize_words_first_letter_simple(std::string& s)
2 {
3     if (s.empty())
4         return;
5
6     // first char: to upper if not a space
7     if (!std::isspace(s[0]))
8         s[0] = std::toupper(s[0]);
9
10    // rest of the string
11    for (int i = 1; i < std::ssize(s); i++)
12    {
13        if (!std::isspace(s[i])
14            && std::isspace(s[i - 1])) // if the previous char is a space
15            s[i] = std::toupper(s[i]);
16    }
17 }
```

Listing 15.1: Linear time constant space solution.

Listing 15.1 works in two phases:

1. makes sure that the first character of *s* is handled properly (depending on whether it is a space or not);
2. takes care of the rest of the characters from the position 1 (skipping the very first one) onward.

15.2.0.1 `std::adjacent_find`

The same idea discussed above and shown in Listing 15.1 can be implemented using the function `std::adjacent_find[9]` from the STL which can be used to search, in a range, for a pair of subsequent elements satisfying user-provided criteria. In the context of this solution we can use it to find all pairs composed by a space followed by a letter; which we know is the letter that has to be capitalized as it marks the beginning of a word. Listing 15.2 implements this idea.

```

1 void capitalize_words_first_letter_adj_find(std::string& s)
2 {
3     if (s.size() > 0 && !std::isspace(s.front()))
4         s.front() = std::toupper(s.front());
5
6     const auto p = [](const auto& x, auto& y) {
7         return std::isspace(x) && !std::isspace(y);
8     };
9     auto it = std::adjacent_find(begin(s), end(s), p);
10    while (it != std::end(s))
11    {
12        it++;
13        *it = std::toupper(*it);
14        it = std::adjacent_find(it, std::end(s), p);
15    }
16 }

```

Listing 15.2: Linear time constant space solution using `std::adjacent_find`[9]. by

The complexity of the Listing 15.2 is linear in time and constant in space and it has the same asymptotical complexity profile as the one presented in Listing 15.1 with the added benefit of being more expressive.

15.2.0.2 Recursive solution

Another way to solve this problem is to adopt a recursive approach as follows:

1. find the first character in the string;
2. transform it in uppercase;
3. ignore all the subsequent non-space characters until a space or the end of the string is reached.

When we reach a space we repeat the process from the beginning; otherwise we stop. At that point the whole string is modified so that only the first character of every word is in uppercase and the rest of the string is untouched. These rules can also be easily turned into code as shown in Listing 15.3.

```

1 void capitalize_words_first_letter_iterator(std::string &s)
2 {
3     auto it = begin(s);
4     while (it != end(s))
5     {
6         //(1) skip all spaces
7         while (it != end(s) && *it == ' ')
8             it++;
9         //(2) to_upper
10        if (it != end(s))
11            *it = toupper(*it);
12
13        //(3) skip the rest of the word
14        while (it != end(s) && *it != ' ')
15            it++;
16    }
17 }

```

Listing 15.3: Linear time constant space solution.

The code is clearly divided into three distinct blocks; each performing one of the tasks listed above (see code comments). The variable `it` is an iterator pointing to the element currently under examination and it is used by the outer loop to determine whether the string has been completely processed. `it` is moved inside the body of the loop which, by processing the text from left to right, ignores all spaces until a letter is found (first inner

loop). This letter is then capitalized and `it` is moved forward so that all the non-space intra-word characters are ignored (second inner loop). This process repeats until the text is fully processed.

Note how we use short-circuit evaluation^②[18] in the `while (it != end(s)&& *it == '')` expression so as to always be sure `it` is pointing to a valid element when we dereference it.

The complexity of this solution is linear in time as every letter is read or modified at most once. The space complexity is constant.

15.3 Common Variations

15.3.1 Apply an user provided function

Sometimes this problem can be posed such that the operation to be applied to the letter is different than capitalization; or we can be even asked to write a high order function that takes the operation to be performed as an additional parameter (as a lambda for instance).

We can use the same core ideas discussed above even if we decide to go for a generic solution where we accept a function from the user. The complications are only syntactical as shown in the Listing 15.4, where we can see how a generic solution can be implemented. The code takes as an input a string and, in addition to the other solutions discussed above, a function `char f(char)` which takes as an input a character and returns a character. This function is used in place of the `std::to_upper`.

```

1  template <class Fn>
2  void capitalize_words_first_letter_adj_find(std::string& s, Fn f)
3  {
4      if (s.size() > 0 && !std::isspace(s.front()))
5          s.front() = f(s.front());
6
7      const auto p = [](const auto& x, auto& y) {
8          return std::isspace(x) && !std::isspace(y);
9      };
10     auto it = std::adjacent_find(begin(s), end(s), p);
11     while (it != std::end(s))
12     {
13         it++;
14         *it = f(*it);
15         it = std::adjacent_find(it, std::end(s), p);
16     }
17 }
```

Listing 15.4: Generic version of Listing 15.2

15.3.2 Modify the every k^{th} character of every word

Another common variation is where we are asked to modify every k^{th} character of a word if it exists. For example, you might be asked to solve the exercise below:

Problem 21 Given a string s , modify s such that every 3^{rd} letter of every word in s is modified according to a function provided by the user. The rest of the string should remain untouched. ■

^②Also known as *minimal evaluation* or *McCarthy evaluation* refers to the semantic of certain boolean operators in which the second argument is executed or evaluated only if the first argument does not suffice to determine the value of the expression.

This variation can be solved by using any of the codes shown above as a starting point and you have the chance of solving it yourself in the next exercise.

16. Trapping Water

Introduction

Imagine being the only survivor of a plane crash in the middle of the Pacific ocean. You manage to get ashore on Manra Island^① but the only cargo that survived the crash was a large number of plastic cubic boxes of size $1m^3$ and two long, wide and rigid sheets of acrylic glass^②.

One of your first tasks should be to ensure you have sufficient water to survive until you are rescued. You come up with the brilliant idea of arranging the boxes into piles of different heights each oriented in the same direction at a certain distance from each other so that they form concave structures which, sealed with the help of the plastic sheets, collect rainwater as depicted in Figure 16.1.

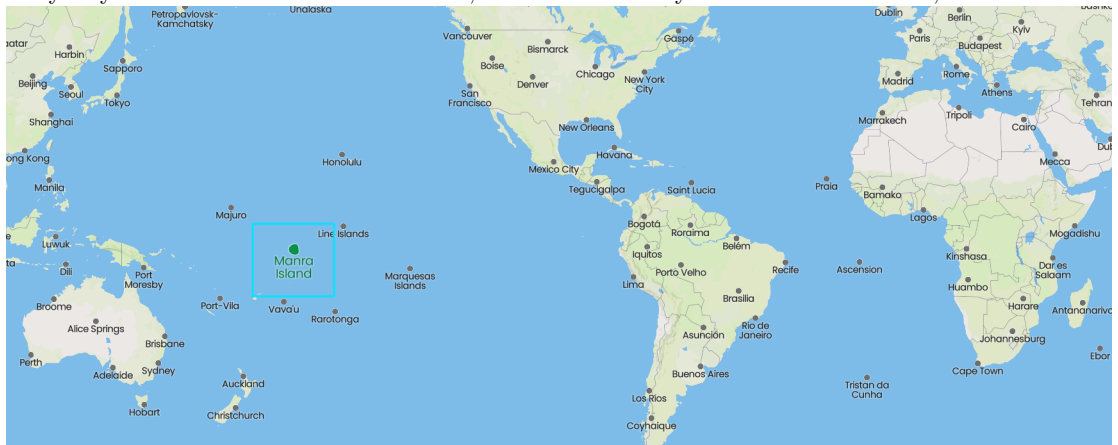
In order to figure out the best ways of arranging boxes so that the structure collects as much water as possible given the scarce rainfall, you need to calculate the total amount of water each possible arrangement of the boxes can hold.

In this chapter we will investigate how this calculation can be carried out efficiently. As we shall see there are a number of valid approaches and it is important to master the core concepts of each of them as these solutions have a broader application than just this specific problem. Moreover, this type of question remains very popular with interviewers at major tech companies.

16.1 Problem statement

Problem 22 Write a function that takes as input an array of length n of non-negative integers representing an elevation map (the height of each pile of boxes) where the width of each bar is 1. The function should return the maximum amount of water that

^① Also called Sydney Island and abandoned since 1963; it is almost entirely covered in scrub forest, herbs



and grasses.

^② Poly (methyl methacrylate) (PMMA), is a transparent thermoplastic often used in sheet form as a lightweight or shatter-resistant alternative to glass.

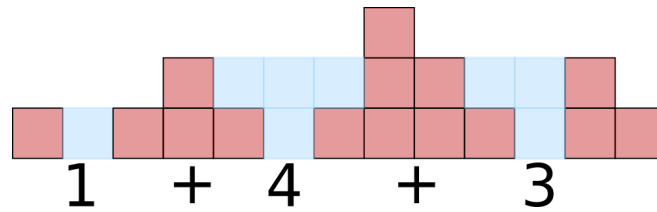


Figure 16.1: Visual representation of the heightmap $H = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]$ in the Example 16.1. Blue squares represent water, while the red ones, boxes.

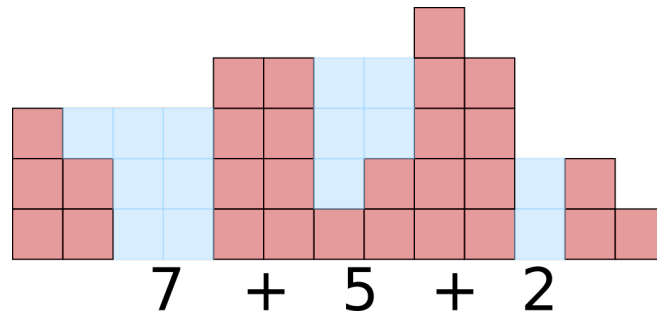


Figure 16.2: Visual representation of the heightmap $H = [3, 2, 0, 0, 4, 4, 1, 2, 5, 4, 0, 2, 1]$ in the Example 16.2. Blue squares represent water, while the red ones, boxes.

can be potentially trapped in between all the piles.

■ Example 16.1

Given the array $H = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]$ (see Figure 16.1), the function returns 8

■

■ Example 16.2

Given the array $H = [3, 2, 0, 0, 4, 4, 1, 2, 5, 4, 0, 2, 1]$ (see Figure 16.2), the function returns 14

■

■

16.2 Discussion

There are at least three techniques we can use to solve this problem satisfactorily:

1. Dynamic programming (Section 16.2.2)
2. Two pointers (Section 16.2.3)
3. Stack (Section 16.2.4)

But we will begin our analysis by looking at a possible brute-force solution before moving on to more sophisticated and faster solutions.

16.2.1 Brute-force

A possible brute-force solution to this problem stems from the fact that each element of the array (pile) can potentially hold some water provided that there are two other bars surrounding it (one at its left and one at its right), with equal or higher height. If that is the case then we can safely add enough water so that its level reaches a height equal to the smallest of the two surrounding piles. For instance in 16.2, we can see that, for the element at index 6 (having height 1), the highest bars on its left and right have height 3 and 4, respectively. We can add water on top of the pile at index 6 up to a height of 3

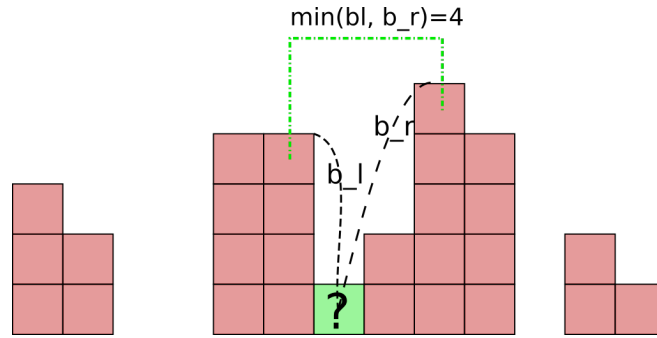


Figure 16.3: Calculation of the water we can fit on top of a pile using the information about the highest bars surrounding it.

(the minimum between 3 and 4) without risking the water spilling out. Figure 16.3 depicts how the pile marked with the question mark can be processed using this approach i.e. by calculating the minimum between b_l and b_r (the height of the highest bars on its left and right, respectively).

If on the other hand, a pile is higher than both the highest bars on its left and right side, then it is impossible to add any water to it (this is always the case when processing the highest bar on the histogram).

To summarize, we can find the answer by calculating the amount of water we can place on top of each pile $H[i]$ by:

1. find the height of the highest bars on the left (b_l) and right (b_r) of $H[i]$;
2. add $\text{std::max}(0, \text{std::min}(b_l, b_r) - H[i])$ to the final answer.

b_l and b_r can be implemented with a simple linear search which is performed for all the elements of H causing the complexity of the full algorithm to reach $O(n^2)$. Moreover, we should note that the first and the last element will never be able to hold any water as those elements have no bars on their left and right sides (any water placed on top of them will inevitably spill outside the structure).

Listing 16.1 shows a possible implementation of this idea. Notice how the `std::max_element` function from C++ STL can be employed elegantly to calculate b_l and b_r .

```

1  int trapping_water_brute_force(const std::vector<int> &height)
2  {
3      const int size = height.size();
4      int ans = 0;
5      for (int i = 1; i < size - 1; i++)
6      {
7          const int b_l =
8              *std::max_element(std::begin(height), std::begin(height) + i);
9          const int b_r =
10             *std::max_element(std::begin(height) + i + 1, std::end(height));
11
12             const int min_max_on_side = std::min(b_l, b_r);
13             // equivalent to
14             // if(min_max_on_side - height[i] > 0) ans+=min_max_on_side - height[i]
15             ans += std::max(0, min_max_on_side - height[i]);
16         }
17     return ans;
18 }
```

Listing 16.1: Brute-force solution.

16.2.2 Dynamic Programming

The solution proposed in Section 16.2.1 is far from optimal, but it can be transformed into a better one if we can somehow precalculate the values of b_l and b_r in linear time. We have discussed in great detail how this can be achieved in Chapter 6, and it can indeed be accomplished in linear time.

Therefore all that's necessary is to use pre-calculate (reusing the logic in Listing 6.2 for instance) R , and L each of length n (same as $|H|$) where:

- $R[i]$ contains the value of the highest bar among all elements of the input with index $j > i$ (on the right of, and not considering, i).
- symmetrically, $L[i]$ contains the value of the highest bar among all elements of the input with index $j < i$ (on the left of, and not considering, i).

Armed with these two arrays, the same algorithm used in Section 16.2.1 can be turned into an elegant and efficient solution as shown in Listing 16.2 which will impress most interviewers.

```
1 std::vector<int> max_left_it(auto begin, auto end)
2 {
3     std::vector<int> L(std::distance(begin, end), 0);
4
5     int i = 0;
6     int cmax = *begin;
7     while (++begin != end)
8     {
9         L[++i] = cmax;
10        cmax = std::max(cmax, *begin);
11    }
12    return L;
13 }
14
15 int trapping_water_DP(const std::vector<int> &height)
16 {
17     const size_t len = height.size();
18     if (len < 2)
19         return 0;
20
21     int ans = 0;
22     std::vector<int> L(max_left_it(height.begin(), height.end()));
23     // reversed input to calculate
24     std::vector<int> R(max_left_it(height.rbegin(), height.rend()));
25     std::reverse(R.begin(), R.end());
26
27     for (size_t i = 0; i < height.size(); i++)
28     {
29         ans += std::max(0, std::min(R[i], L[i]) - height[i]);
30     }
31
32     return ans;
33 }
```

Listing 16.2: Dynamic programming, $O(n)$ time and space solution.

The code is clearly divided into two separate and somewhat independent steps each of time complexity $O(n)$.

1. vectors R and L are filled up by using the function `max_left_it`. It is worth noting here how we are able to calculate R only using the function `max_left_it` (that, as its name suggests, calculates the maximum value to the left of each element of the input array), by providing the input to `max_left_it` reversing what comes out of it;

Figures 16.4a and 16.4b show a representation of L and R . If we superimpose them we can visualize the amount of water we can trap between the piles as shown in Figure 16.4c.

2. the answer calculated by summing up the amount of water that can stand on top of a pile by using the values of `R[i]`, `L[i]` and `height[i]` as also discussed in Section 16.2.

Listing 16.2 has a time complexity of $O(n)$ because the computation of L and R can be done in linear time, while calculating the final answer can be done in a single pass over the array. The space complexity is linear as well as the arrays L and R both have a size proportional to n .

16.2.3 Two pointers solution

Despite the fact that the solution presented in Section 16.2.2 is already good time-complexity-wise, we can definitely do better and lower the space complexity down to $O(1)$. The key here is that we do not really need to store the information of L and R in their entirety. Whenever we calculate the contribution of a pile we only need a single value from both L and R . Once we are done with that pile we can freely discard the corresponding elements in L and R we have used because they won't be used in the future as they do not play any role in the calculation of the contribution to the final answer of any other any other element of the input array H . The solution proposed in this Section uses a two-pointers technique which maintains two *rolling* maximum values, m_l and m_r , one for the left and one for the right of a single pile. When processing the pile at index i , m_l and m_r contain the maximum value to the its left and right, respectively (the same information stored in $L[i]$ and $R[i]$).

The input array is traversed from left to right using two pointers l and r , which are initialized to the beginning and the end of the input array, respectively. m_l and m_r are initialized to 0. The idea is that we process one of the elements pointed to by l or r depending on which is larger. m_l and m_r always contain the largest element to the left of l and, symmetrically m_r contains the largest element to the right of r .

We process the smallest element between $H[l]$ and $H[r]$ and in particular if $H[l] \leq H[r]$ is:

true: then the contribution of $H[l]$ is bounded by m_l (because $H[r] \geq H[l]$). l is moved towards the center (we have considered the contribution of this cell and we can move forward).

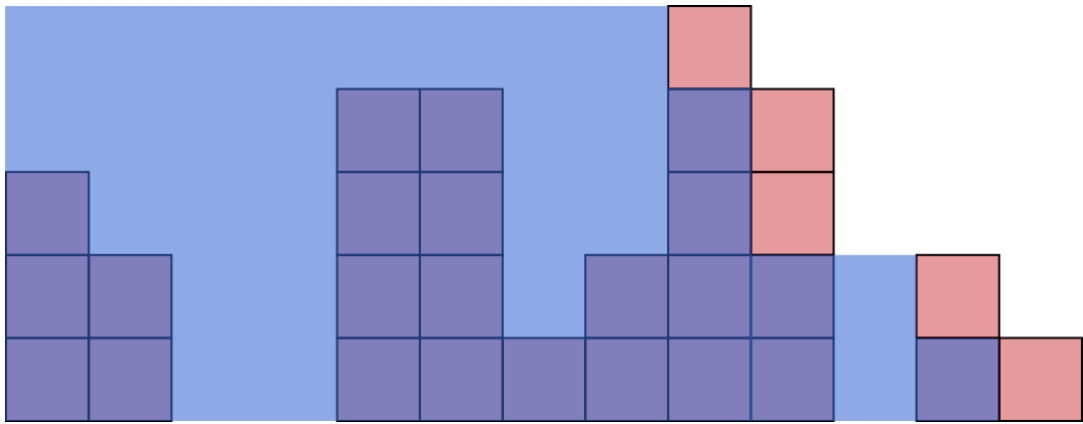
false: then the contribution of $H[r]$ is bounded by m_r (because $H[l] \geq H[r]$). r is moved towards the center (we have considered the contribution of this cell and we can move forward).

An implementation of the idea above is shown in Listing 16.3.

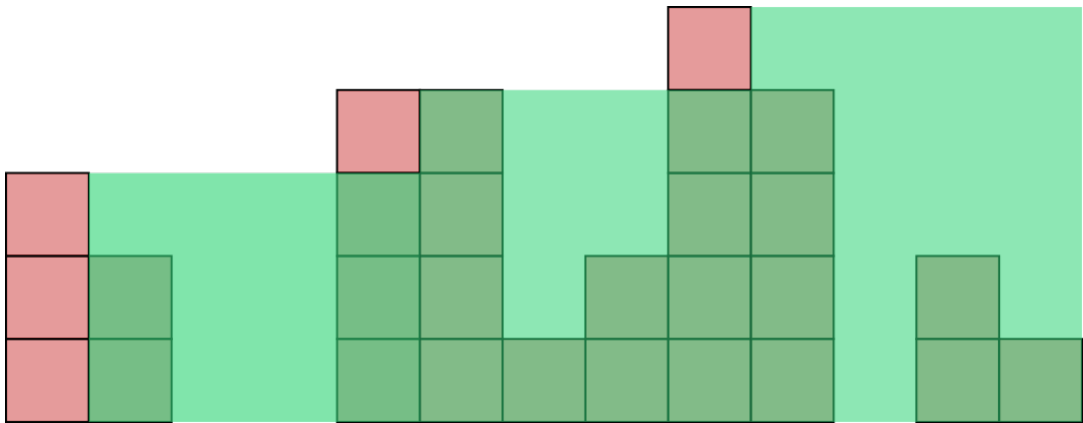
```

1  int trapping_water_two_pointers(const std::vector<int>& H)
2  {
3      int m_l, m_r;
4      m_l = m_r = 0;
5
6      int l = 0;
7      int r = H.size() - 1;
8
9      int ans = 0;
10     while (l < r)
11     {

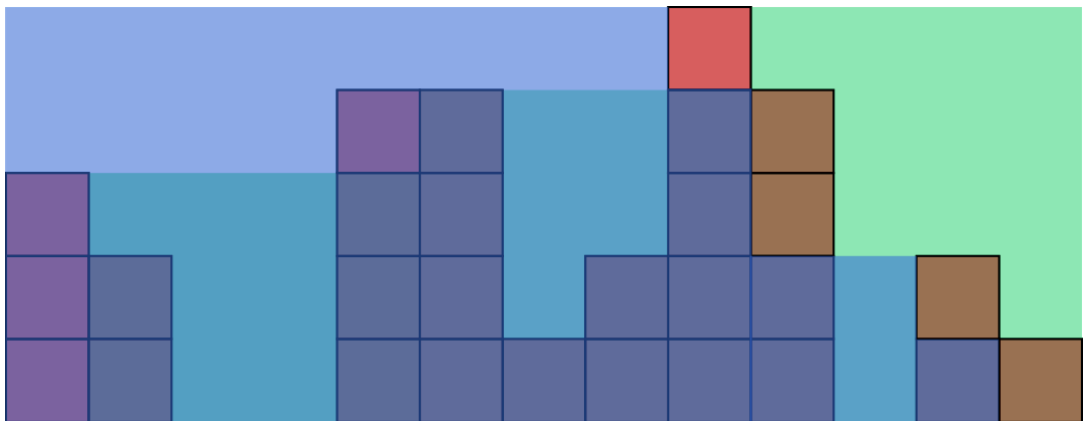
```



(a) Representation of the highest value to the right of a pile. The level of color ■ on top of each pile marks the maximum height of another pile to its **right**.



(b) Representation of the highest value to the left of a pile. The level of color ■ on top of each pile marks the maximum height of another pile to its **left**.



(c) Superimposition of Figures 16.4a and 16.4b. Cells colored in ■ represent the intersection between cells colored in ■ in Figure 16.4a and cells colored in ■ in Figure 16.4b. Those are the cells that can trap water.

Figure 16.4

```

12     if (H[l] <= H[r])
13     {
14         m_l = std::max(m_l, H[l]);
15         ans += m_l - H[l];
16         l++;
17     }
18     else
19     {
20         m_r = std::max(m_r, H[r]);
21         ans += m_r - H[r];
22         r--;
23     }
24 }
25 return ans;
26 }

```

Listing 16.3: Two pointers solution, $O(n)$ time and $O(1)$ space, solution to the problem of calculating the amount of water trapped between buildings.

The code is fairly self explanatory but it is worth noting that all the elements of the input array will be considered eventually because either l or r are moved one step closer to each other at each iteration.

This approach has a time complexity of $O(n)$ (we cannot do better than this because we have to touch all the elements of the input at least once) and a space complexity of $O(1)$ which is optimal.

We believe this is the solution the interviewer is hoping to see.

16.2.4 Stack based solution

There is another way of solving this problem in linear time that makes good use of a stack. The main idea is to loop through H one element p_i (pile at index i in H) at a time and try to place p_i on the stack in such a way that the values in the stack are always in decreasing order (from top to bottom). We start with an empty stack S and for each pile $p_i \in H$ we do one of the two following operations depending on whether p_i is lower than the current top of the stack s_{top} :

$p_i < s_{top}$ the current top of the stack is higher than p_i which means that s_{top} bounds p_i from the left. In this case we simply add it to the top of the stack. At this point the stack is still ordered in a decreasing fashion.

$p_i \geq s_{top}$ we have a pile that is higher or equal to the s_{top} . Therefore if we want to place p_i on the stack and maintain the ordering, we need to remove as many elements as necessary until we are left with a top of the stack that is higher than p_i , or the stack is fully empty (this happens for instance, but not only, when p_i is the highest pile in H). However each pile that is removed together with p_i forms a rectangular area that can trap some water. This is because the removed pile is bounded from the left by some other pile (this is guaranteed by the way in which elements are inserted in the stack, or more simply by the ordering the stack maintains) and to the right by p_i itself. How much water exactly? The contribution of the removed element S_{oldtop} is calculated as the area of a rectangle of base equal to the distance between the two bounding bars (p_i and the element before S_{oldtop}) and height that is equal to the minimum of the heights between the two bounding bars **minus** S_{oldtop} . Clearly if there is no bar left in the stack once we remove S_{oldtop} , there is no way we can trap water and therefore we add nothing to the final answer.

Figure 16.5 shows a step-by-step execution of this algorithm.

A possible implementation of this idea is shown in Listing 16.4. Note that the real

work happens inside the `while` loop and that the pile where we calculate the contribution is named `old_top` while the bar on the left and right are named `bar_left` and `bar_right` (equivalent to p_i) respectively. Moreover, no matter whether the stack is unwound or not, `bar_right` is **added** to the stack because when controls reaches line 21 it will be either the only bar in the stack of it will be smaller than the current top (order is restored).

The complexity of this approach is linear in both time and space .

```
1 int trapping_water_stack(const std::vector<int> &H)
2 {
3     int ans = 0, current = 0;
4     // w store indexes of the bars
5     std::stack<int> w;
6
7     for (size_t bar_right = 0; bar_right < H.size(); bar_right++)
8     {
9         while (!w.empty() && H[bar_right] > H[w.top()])
10        {
11            const int old_top = w.top();
12            w.pop();
13            if (w.empty())
14                break;
15            const int bar_left = w.top();
16            const int distance = bar_right - bar_left - 1;
17            const int bounded_height =
18                std::min(H[bar_right], H[bar_left]) - H[old_top];
19            ans += distance * bounded_height;
20        }
21        w.push(bar_right);
22    }
23    return ans;
24 }
```

Listing 16.4: Stack based solution, $O(n)$ time and space solution.

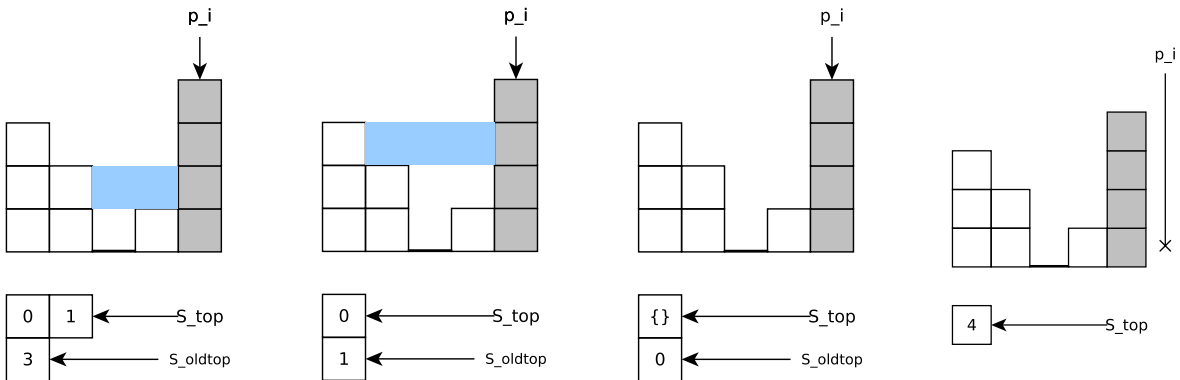
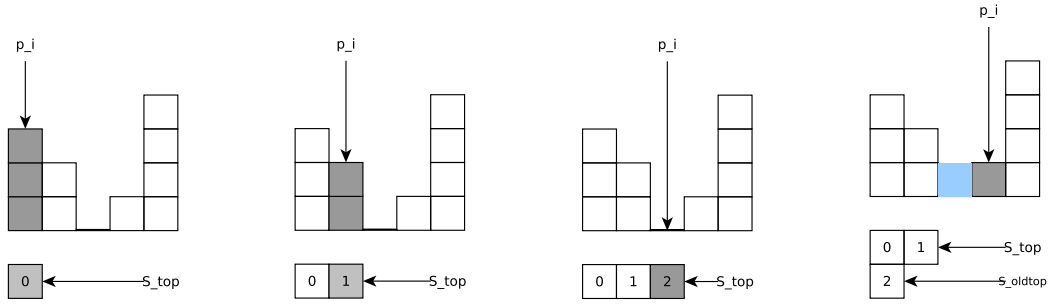


Figure 16.5: Execution of the algorithm discussed in Section 16.2.4 and implemented in Listing 16.4 on the input $H = \{3, 2, 0, 1, 4\}$. The final answer is the sum of the light blue cells, for a total of 6 squares worth of water that can be trapped by this particular pattern of piles.

17. Minimum element in rotated sorted array

Introduction

In this chapter, we will tackle a very popular interview question that has a surprisingly short statement and an obvious linear time solution. Given, however, the aim of impressing an interviewer, we will focus on developing an elegant and efficient solution which requires more thought and careful coding.

This problem is based upon the concept of array rotations. To develop an intuitive understanding of this concept, imagine that we want to “rotate” the elements of an array; that is to shift all of them to the right by a certain number k of positions. The element that used to be at position 0 is now at position k and the element that was at position one is now at $k + 1$ etc. (see Figure 17.1).

17.1 Problem statement

Problem 23 Given an array A sorted in ascending order with no duplicates and rotated around a pivot, return the smallest element.

■ **Example 17.1**

Given the rotated array $\{3, 4, 5, 6, 1, 2\}$ the function returns 1. ■

■ **Example 17.2**

Given the rotated array $\{0, 2, 3\}$ the function returns 0. ■

■ **Example 17.3**

Given the rotated array $\{3, 2, 1\}$ the function returns 1. ■

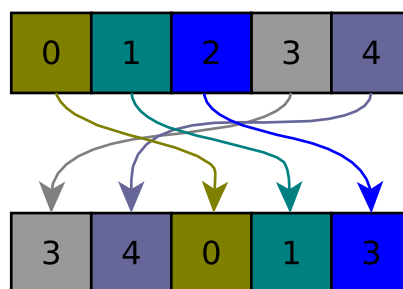


Figure 17.1: Example of array rotation where every element is moved to the right by 2 positions. Note how the elements at position 3 and 4 are wrapped around to positions 1 and 2, respectively.

17.2 Clarification Questions

Q.1. Are all the elements unique?

Yes, you can assume all the elements are unique

Q.2. Can the input array be empty?

No, you might assume the array contains at least one element.

17.3 Discussion

What does it actually mean for a sorted array to be rotated around an element? Given a sorted array $A = \{a_0, a_1, \dots, a_{n-1}\}$ s.t. $\forall 0 \leq i < n : a_i < a_{i+1}$, rotating A around the pivot element at index p results in: $A_p = \{a_p, a_{p+1}, \dots, a_{n-1}, a_0, a_1, \dots, a_{p-1}\}$. In a nutshell, all the elements are rotated in such a way that the element at index p becomes the first element of the array. For instance, rotating the array $X = \{1, 2, 3, 4, 5\}$ around the element at index 2, results in $X = \{3, 4, 5, 1, 2\}$. We would obtain the same result by applying a offset of either -2 or $3 = 5 - 2 = (|A| - 2)$ positions to each element of X .

This way of performing rotation is so common that there is an algorithm in the C++ STL[10] adopting such API.

17.3.1 Brute-force

The brute-force solution to this problem is trivial and consists of simply looping through the array and keeping a record of the smallest element encountered. In C++ this can be implemented with a one-liner as shown in Listings ?? and 17.2 both having $O(n)$ time $O(1)$ space complexity.

```
1 template <typename T>
2 auto min_rotated_array_brute_force_1(const std::vector<T>& V)
3 {
4     auto ans = std::numeric_limits<T>::max();
5     for (const auto v : V)
6         ans = std::min(ans, v);
7     return ans;
8 }
```

Listing 17.1: Brute force solution using an explicit loop.

```
1 template <typename T>
2 auto min_rotated_array_brute_force_1(const std::vector<T>& V)
3 {
4     auto ans = std::numeric_limits<T>::max();
5     for (const auto v : V)
6         ans = std::min(ans, v);
7     return ans;
8 }
```

Listing 17.2: One-liner brute force solution.

It is worth mentioning this approach during the interview however, no time should be spent in its actual implementation as the interviewer will assume you know how to carry this out and is looking for you to present a more advanced solution that takes advantage of the fact that the array is sorted (even if provided in a rotated form).

17.3.2 Logarithmic solution

As usual, when the word “sorted” makes its appearance in a problem statement our first thought should be **binary search** see Appendix ??). Indeed, in this problem we are

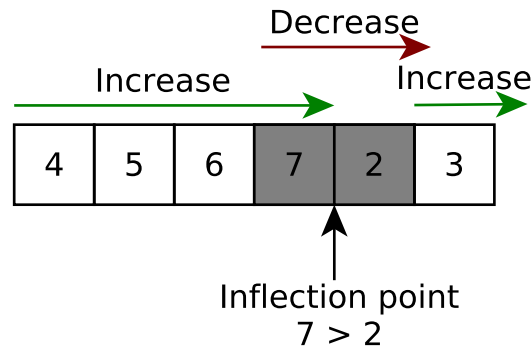


Figure 17.2: Inflection point in a rotated sorted array. When the binary search examines both element 7 and 2 it is able to determine the inflection point (element 2).

almost forced to consider binary search as it not only involves a sorted input, but it is also about searching.

How can we use binary search to actually solve this problem, given the fact we have an oddly sorted array? First note that, despite the fact that the array is not sorted in a canonical manner, it is still very much sorted as there is an index i of the array holding the smallest value from which we could iterate the array forward (and eventually continue from the start when we reach the end) and all we would see is a sorted sequence.

In order to be able to apply binary search effectively to any problem we need to be able to:

1. keep track of a range of elements that are currently under examination. Binary search works by cutting off parts of a search range until it becomes empty or a solution is found. Usually such range is initialized to be the closed interval: $[l = 0, r = A.size() - 1]$ i.e. the entire array;
2. analyze the element in the middle of this range;
3. if the middle element is the one we are looking for we are done;
4. otherwise, the search proceeds either to the left or to the right of the range.

The core challenges of this specific problem lie at steps 2 and 4 because we need to be able to:

- test whether an element is the minimum or not (2)
- decide how to split the space range into two and whether to proceed with the search on the right-hand or on the left-hand side (4).

17.3.2.1 Test if an element is the minimum

In order to decide whether an element a_k at index k is the minimum, it is useful to look at one property that differentiates it from all the other values in the collection. The minimum element is the only element s.t. both the elements on its right and left are **greater** than it (sometimes this element is referred to as an inflection point). Another useful property that can aid in the identification of the minimum is that the element on its left is always the maximum element of the array (see examples in Section 17.3 and Figure 17.2). Thus, whenever $a_{k-1} > a_k$ (meaning that a_k is the minimum and a_{k+1} the maximum) or $a_k > a_{k+1}$ (meaning that a_k is the maximum element and a_{k+1} the minimum) we can stop and return because we have found the answer.

In short, Listing 17.3 shows the condition that can be used in the binary search to test

whether a_k is the answer to the problem. Note how the modulo operation is used in order to avoid having to specialize this test for the elements at the beginning and at the end of the array (positions 0 and $A.size() - 1$, respectively).

```

1  const int curr = A[k];
2
3  const int prec = A[(k-1+A.size())]    //+A.size() due to negative modulo
4  const int succ = A[(k+1)%A.size()];
5  if( (curr <= prec) || (curr >= succ))
6      return min({prec, curr, succ});
7  }

```

Listing 17.3: Test to verify whether the binary search can stop because an answer has been found.

17.3.2.2 Binary search range split

The last part of the algorithm to be addressed is how and in which split of the array to continue the binary search if the element in the middle of the range is not good to determine the answer. An useful property of the sorted rotated array is that, when the smallest element is at position i then **all the elements on its right side are smaller than the very first element of the array** (at index 0) i.e. the following is always true:

- $(a_i < a_0) \wedge (a_{i+1} < a_0) \wedge \dots (a_{n-1} < a_0)$
- $(a_{i-1} \geq a_0) \wedge (a_{i-2} \geq a_0) \wedge \dots (a_0 \geq a_0)$

For instance, consider the sorted rotated array $\{8, 9, 10, 5, 6, 7\}$: the minimum element 5 is at index 3 and all the elements located between index 3 and 5 are strictly smaller than the first element 8 while all the elements to the left of 5 are larger than or equal to 8.

This is the last piece of information that is needed in order to make the binary search work as we can use it to determine which portion of the two subarrays (the one to the left or to the right of *middle*) to discard. Therefore, given an element at position i is not the answer, we will continue the binary search on the subarray to the left of i if $a_i > a_0$, otherwise we will use the right side.

By being able to test whether an element is the smallest element in the array and, if not, how to split the array and continue the binary search, we have all the ingredients necessary to solve this problem efficiently.

An implementation of this idea is shown in the Listing 17.4.

```

1  template <typename T>
2  auto min_rotated_array_brute_force_log(const std::vector<T> &V)
3  {
4      const auto size = V.size();
5
6      int l = 0, r = size - 1;
7      while (l <= r)
8      {
9          const int mid = l + (r - l) / 2;
10         const auto curr = V[mid];
11         const auto prec = V[(mid - 1 + size) % size];
12         const auto succ = V[(mid + 1) % size];
13
14         if ((curr <= prec) || (curr >= succ))
15         {
16             return std::min(std::min(curr, succ), prec);
17         }
18
19         if (curr > V[0])
20             l = mid + 1;

```

```
21     else
22         r = mid - 1;
23     }
24     return -1;
25 }
```

Listing 17.4: Logarithmic solution implemented using a standard iterative binary search.

This solution has a complexity of $O(\log(n))$ time and $O(1)$ space. The code is a straightforward implementation of the binary search where `l` and `r` determine the range under examination, `middle` is the element in the middle of `l` and `r` while `prec` and `succ` are the elements preceeding and succeeding `mid`, respectively. Note how the modulo operation is used to make sure that both `prec` and `succ` always point to a valid element.

18. Search in sorted and rotated array

Introduction

The problem presented in this chapter is another classic that often appears during interviews and that can be considered to be an evolution of the problem of finding the minimum element in a sorted and rotated array which was covered in Chapter 17 (at page 82). The two problems are so deeply linked that it is actually possible to solve this problem by using the other's solution structure.

18.1 Problem statement

Problem 24 Write a function that given an ascending sorted array A of length n with no duplicates and rotated around a pivot p (meaning that the array has been rotated such that the smallest element of A ends up at index p), and an integer t , returns:

- if t does not exist in A it returns -1
- otherwise the index of A where t appears.

■ Example 18.1

Given $A = \{3, 4, 5, 6, 1, 2\}$ and $t = 5$ the function returns 2.

■ Example 18.2

Given $A = \{3, 4, 5, 6, 1, 2\}$ and $t = 7$ the function returns -1 .

18.2 Clarification Questions

Q.1. Are all the elements unique?

Yes, you can assume all the elements are unique

Q.2. Can the input array be empty?

No, you might assume the array contains at least one element.

18.3 Discussion

18.3.1 Brute-force

As was the case for the problem of finding the minimum in a sorted and rotated array (Chapter 17) the brute-force solution to this current problem is trivial and consists of simply running a linear search in the entire array as shown in Listing 18.1. Not surprisingly, the complexity of this implementation is linear in time and constant in space.

```
1 int search_sorted_rotated_array_bruteforce(const std::vector<int>& A,  
2                                           const int t)  
3 {  
4     const auto it = std::find(std::begin(A), std::end(A), t);  
5     return it != std::end(A) ? std::distance(std::begin(A), it) : -1;
```

6 }

Listing 18.1: Brute force solution (linear search) to the problem of finding an element in a sorted and potentially rotated array.

18.3.2 Logarithmic time solution

The solution presented in Section 18.3.1 above is far from optimal given we can solve this problem in logarithmic time and constant space (as we did for the problem in Chapter 17).

The main point is that we need to take advantage of the fact that the array is sorted and that, if we know the pivot location, p then we can logically divide the array into two subarrays, starting and ending at indices:

1. 0 and $p - 1$
2. p and $n - 1$ (n is the size of the array)

Both arrays are sorted and thus binary search can be applied in each of the subarrays separately. This is why we can reuse the solution to the problem of finding the minimum element in a sorted and rotated array to solve this one.

To summarize the algorithm, proceed as follows:

- search for the pivot index p
- search for t in $A[0, p - 1]$. If the search is successful return the found index for t .
- search for t in $A[p, n - 1]$. If the search is successful return the found index for t .
- None of the searches had success. t is not in the array. Return -1 .

This algorithm can be implemented as shown in Listing 18.2. Note that the function `find_idx_min` is almost identical to the one in Listing 17.4 and has been modified so as to return the index instead of the value for the smallest element in the array.

Also note that the function `midpoint` is not implemented as $(l+r)/2$ because that might cause overflow during the computation of $(l+r)$ even if the final result fits in a `int`. Specifically, it fails if the sum of low and high is greater than the maximum positive `int` value ($2^{31} - 1$ in most C++ implementation). The sum overflows to a negative value and the value stays negative when divided by two. In C++ this causes an array index out of bounds with unpredictable results. In Java, it throws `ArrayIndexOutOfBoundsException`.

Finally, the function `binary_search` implements a simple and canonical recursive binary search. The complexity of the overall implementation is $O(\log(n))$ in time and $O(1)$ in space.

```
1 using Range = std::pair<int, int>;
2 inline unsigned midpoint(const unsigned l, const unsigned r)
3 {
4     return l + (r - l) / 2;
5 }
6
7 inline int positive_modulo(const int n, const int m)
8 {
9     return (n + m) % m;
10 }
11
12 int find_idx_min(const vector<int>& A)
13 {
14     const int size = A.size();
15     int l = 0;
16     int r = A.size() - 1;
17     if (A[l] < A[r])
18         return l;
19     while (l <= r)
```

```

20 {
21     const int mid = midpoint(l, r);
22
23     const int curr = A[mid];
24     const int next = A[positive_modulo(mid + 1, size)];
25     const int prec = A[positive_modulo(mid - 1, size)];
26
27     if (curr <= next && curr <= prec)
28         return mid;
29     if (curr < A[0])
30         r = mid - 1;
31     else
32         l = mid + 1;
33 }
34 return l;
35 }
36
37 int binary_search(const std::vector<int>& A, const Range& range, const int t)
38 {
39     auto [l, r] = range;
40     if (l > r)
41         return -1;
42
43     const int mid = midpoint(l, r);
44     if (A[mid] == t)
45         return mid;
46
47     if (A[mid] < t)
48         l = mid + 1;
49     else
50         r = mid - 1;
51
52     return binary_search(A, {l, r}, t);
53 }
54
55 int search_sorted_rotated_array_log(const vector<int>& A, int t)
56 {
57     if (A.size() == 0)
58         return -1;
59     const int idx = find_idx_min(A);
60
61     int ans = binary_search(A, Range(0, idx - 1), t);
62     if (ans == -1)
63         ans = binary_search(A, Range(idx, A.size() - 1), t);
64     return ans;
65 }

```

Listing 18.2: Log time solution (using binary search) to the problem of finding an element in a sorted and rotated array.

19. Verify BST property

Introduction

Data structures is a topic that lies at the heart of the entire field of computer science and of virtually every computer code running around the globe. Algorithms are built around particular data arrangements and there are some arrangements that are more convenient than others and often choosing the right one could mean the difference between waiting years for a particular algorithm to come to completion versus seconds.

Among the vast number of the mainstream data structures, trees, and especially the binary kind, are probably one of the most used because they naturally allow representing hierarchical data which is ubiquitous and at the basis, DOM^① (XML,HTML) and JSON documents, which lies at the heart of the Wolrd Wide Web. Trees are also fundamental for compilers as they are used to represent the syntactic structure of a source code for programming languages.

Trees can be defined recursively as a collection of nodes, which contains some data and a list of references to other nodes, the “children”. There is a special node called the root with the property that no other nodes have reference to it. Moreover, a node can only be referenced once (i.e. it can have one and only one father). See Figure 19.1a for an example of a generic tree.

Binary search trees (BST) are a special kind of trees that are extremely useful when we need to arrange data on which the following operations need to be performed

- insert
- delete
- search
- ceil/floor

. In this chapter we are going to look at a common interview question in which we will have to determine whether a given tree is a valid BST or not. Studying this classic problem is important as the structure of and the insights behind it solutions can be transferred to many others trees problems.

19.1 Problem statement

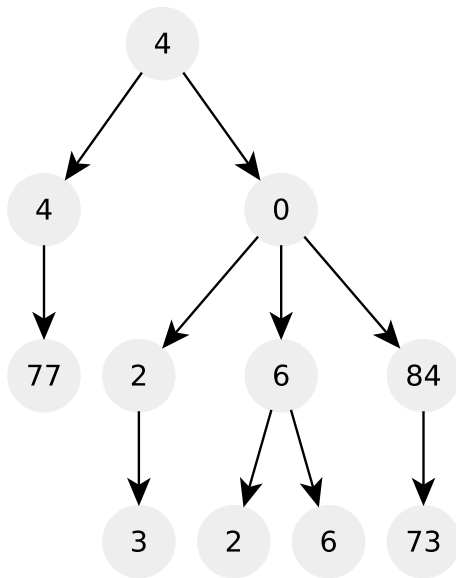
Problem 25 Given a binary tree [16], determine if it is a valid binary search tree.

Assume a BST is defined as follows:

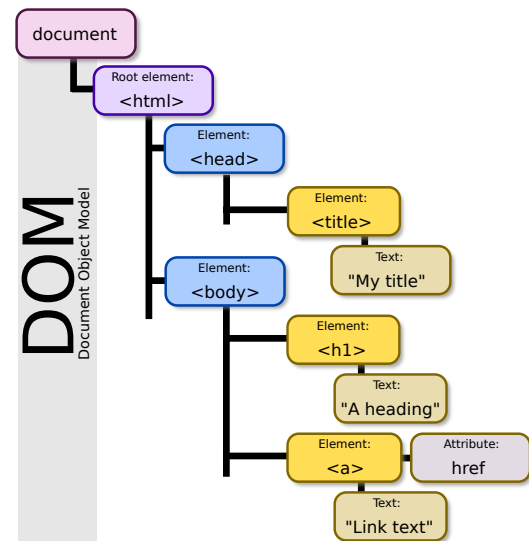
- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.

You can assume the input tree is given as a pointer or a reference to a structure named `TreeNode` which definition is in Listing 19.1.

^①Document Object Model is a way of representing documents as trees wherein each node is an object represents a part of the document (See Figure 19.1b).



(a) Example of a generic tree.



(b) Example of DOM in a HTML document.

Figure 19.1

```

1
2 struct TreeNode {
3     int val;
4     TreeNode *left;
5     TreeNode *right;
6     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
7 };
  
```

Listing 19.1: Binary tree definition used in this exercise.

■ Example 19.1

For the tree shown in Figure 19.2a the function should return **false**.

■ Example 19.2

For the tree shown in Figure 19.2b the function should return **true**.

■ Example 19.3

For the tree shown in Figure 19.2c the function should return **true**.

19.2 Clarification Questions

Q.1. Is it guaranteed the input pointer or reference is valid tree?

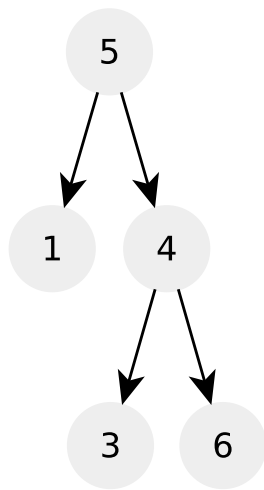
Yes, the input is a valid tree. There is a root, and all the other nodes have exactly one father.

Q.2. Are all elements in the tree distinct?

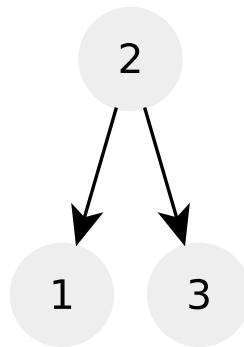
Yes, you can assume all elements are distinct.

Q.3. How many nodes does the tree contain?

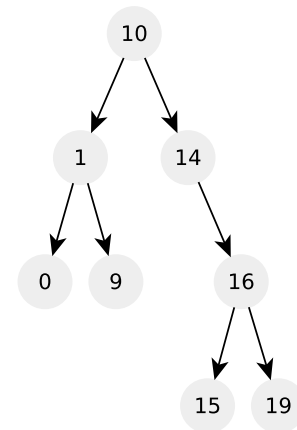
Up to 10^6 nodes.



(a) Tree in Example 19.1



(b) Tree in Example 19.2



(c) Tree in Example 19.3

Figure 19.2

19.3 Discussion

This problem is asking for a function that verifies whether a given tree is a binary search tree or not, therefore we should begin by defining what that actually means. A tree T is a binary search tree if:

1. Every node has two subtrees (named left and right, respectively) i.e. T is a binary tree
2. given a node n in the tree **all** the nodes in its left subtree are smaller than the value in n .
3. additionally, **all** nodes in the right subtree are larger.

For instance, the tree in Figure 19.3b is not a valid BST because node 15 is a right descendant of the root but is not greater than it. On the other hand, the trees in Figure 19.2 and 19.3 are valid BST. The tree in Figure 19.1a is not a valid BST because it is not a binary tree as node 0 has three children.

Therefore, in order to solve this problem we must be able to prove the conditions above hold for each and every node of the input tree.

19.3.1 A common mistake

Most candidates can recall write down the BST properties rather quickly and they dive immediately into writing a greedy algorithm that works as follows: for each node n of T check that `n.val > n->left.val && n.val < n->right.val` i.e. that the payload of the node currently analyzed is greater than the value of its left child but smaller than its right one. This algorithm might even return the correct answer for some input trees but it fails on others, and it is, therefore, not correct. For example, it returns true if we execute on the tree in Figure 19.3b. Usually approaching the problem this way means the end of the round and very likely rejection.

It becomes clear at this point, that the problem with this approach is that when verifying if the BST property holds for node n we only look n and its children. We must be able to check every node in the left and right subtrees of n and also to somehow, make sure the information in n travels down to n 's children and descendants so they can use it (to carry on the check for n itself). We will see how this can be done in the next sections.

19.3.2 Top Down approach

When talking about trees, one should immediately think about a top down approach and recursion. This problem is no different and in-fact becomes almost simple if approached using recursion and the following key considerations are addressed:

1. every node can be thought of as the root of a tree for which the BST property needs to hold (and thus verified).
2. empty trees satisfy the BST property
3. every node must be within a certain range that is determined by its parent. For instance, given the node 15 in the example ??, in order for the tree to be a valid it must be within the range (14,16). Why is that? Simply put, because its parent, the node 16, must be within the range (14, $+\infty$) and additionally node 15, being the left subtree of node 16, must be lower than its parent. The same reasoning can be applied recursively up to the root of the tree where the range of the value is simply $(-\infty, +\infty)$ (no constraints).

The node 9 in the example ?? must be within the range (1,10) for similar reasons.

To summarize, we can visit the tree in a top-down fashion and maintain a range that the current node must satisfy starting with a range equal to $(-\infty, +\infty)$ for the root (meaning that de-facto there is no restriction on the value the root can take). Once the value is checked against the range, then the same function can be applied to the right and left children but making sure that the range is modified accordingly when recurring on the children.

The fulcrum of the problem seems to be the update of such a range change when visiting down the tree. But how does such a range change when visiting down the tree? The idea is simple: given a node n with parent p and range (l_p, u_p) then:

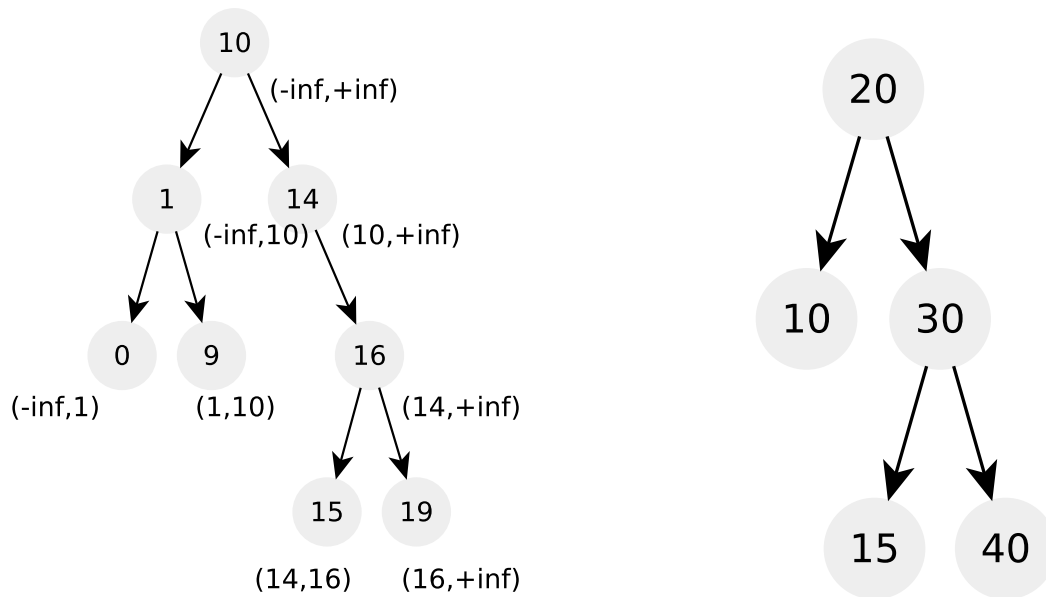
- if n is the right child of p , then the range for n is: (p, u_p) : all nodes in the right subtree of p must be **higher** than p . Note that $p > l_p$ (otherwise the BST property would be violated when checking p) and thus the range for n becomes smaller; meaning that all the constraints coming from the ancestors of p will also be satisfied with the new range (p, u_p) .
- A similar reasoning applies if n is the left child of p . The range for n is then : (l_u, p) .

See Figure 19.3a showing what the ranges look like for the tree in Figure ??.

An implementation of the idea above is shown in Listing 19.2.

```
1
2 inline bool isValidBST_helper(const TreeNode* const root,
3                               const long lower,
4                               const long upper)
5 {
6     if (!root)
7         return true;
8
9     return (root->val > lower) && (root->val < upper)
10         && isValidBST_helper(root->left, lower, root->val)
11         && isValidBST_helper(root->right, root->val, upper);
12 }
13
14 bool isValidBST_top_down(TreeNode* root)
15 {
16     static constexpr long INF = std::numeric_limits<long>::max();
17     static constexpr long mINF = std::numeric_limits<long>::min();
18     return isValidBST_helper(root, mINF, INF);
19 }
```

Listing 19.2: Linear time recursive solution.



(a) Example of tree where each node is label with the range of values it should lie within.

(b) Example of a binary tree that is not a BST.

Figure 19.3

The function `isValidBST_top_down` is a simple entry point for the recursive function `isValidBST_top_down_helper` which uses the parameters `lower` and `upper` to keep track of the range the node `root` must respect. The interesting bits in this function are the ways the recursive calls are made and specifically in what values are provided to the parameters `lower` and `upper`. When performing the first recursive call on `root->left` the parameter `lower` is kept unchanged while the `upper` is set to be equal to the current value of `root` because every element in the left subtree of `n` must indeed be smaller than `root->val`.

Symmetrically for the second recursive call on `root->right`, `upper` stays unchanged and `lower` get the value of `root->val` because all the nodes in `n`'s right subtree must be larger than `root->val`.

The time and space complexities of Listing 19.2 are $O(n)$ and $O(1)$, respectively space (if we do not keep into consideration the space on the stack taken by the recursion).

19.3.3 Brute force

We can look at the problem from a different perspective and try to prove the input tree is not BST rather than trying to prove it is. Clearly, if we fail at proving it is not a BST, then it must indeed be one! A tree is not a BST if we are able to find a node n s.t. its left subtree contains any element greater than n and or its right subtree contains any element smaller than n . This task is trivial when the two following functions are available:

1. `tree_min(TreeNode* root)`
2. `tree_max(TreeNode* root)`

. which return the minimum and the maximum value of a tree, respectively.

We can verify the BST property of a node n is not violated by making sure that the maximum value in its left subtree is not larger or equal than n and that the minimum value in its right subtree is not smaller or equal than n .

Luckily implementing `tree_min(TreeNode* root)` and `tree_max(TreeNode* root)` is easy to do in linear time as all we need to do is traverse the tree (in any order really) and keep track of the smallest/largest element seen.

The idea above can be implemented as shown in Listing 19.4

```
1 int tree_min(TreeNode *root)
2 {
3     if (!root)
4         return std::numeric_limits<int>::max();
5     return std::min({root->val, tree_min(root->left), tree_min(root->right)});
6 }
7
8 int tree_max(TreeNode *root)
9 {
10    if (!root)
11        return std::numeric_limits<int>::min();
12    return std::max({root->val, tree_max(root->left), tree_max(root->right)});
13 }
14 bool isValidBST_min_max(TreeNode *root)
15 {
16     if (!root)
17         return true;
18
19     bool left_ok = true;
20     if (root->left)
21         left_ok = tree_max(root->left) <= root->val;
22     bool right_ok = true;
23     if (root->right)
24         right_ok = tree_min(root->right) > root->val;
25
26     return (left_ok && right_ok)
27         && (isValidBST_min_max(root->left) && isValidBST_min_max(root->right))
28         ;
29 }
```

Listing 19.3: Quadratic solution based on the two functions returning the minimum and maximum value of a tree.

Listing 19.4 has a quadratic time complexity because for each and every node of the tree we perform a linear amount of work. The space complexity is also linear if we consider the space on the stack to perform the recursion.

Notice that we can do substantially better than quadratic time if we memoize `tree_min` and `tree_max` at the expense of using linear space. We can, for instance, use a map to store the information for each `TreeNode*` and its associated min and max values. An implementation of this idea is shown in Listing ??.

```
1 #include <iostream>
2
3 using Cache = std::unordered_map<TreeNode*, int>;
4
5 int tree_min_memoized(TreeNode* root, Cache& cache)
6 {
7     if (!root)
8         return std::numeric_limits<int>::max();
9
10    if (cache.contains(root))
11    {
12        return cache[root];
13    }
14
15    const auto ans = std::min({root->val,
16                               tree_min_memoized(root->left, cache),
17                               tree_min_memoized(root->right, cache)});
18    // update the cache
```

```

19     cache.insert({root, ans});
20     return ans;
21 }
22
23 int tree_max_memoized(TreeNode* root, Cache& cache)
24 {
25     if (!root)
26         return std::numeric_limits<int>::min();
27     if (cache.contains(root))
28     {
29         return cache[root];
30     }
31
32     const auto ans = std::max({root->val,
33                               tree_max_memoized(root->left, cache),
34                               tree_max_memoized(root->right, cache)});
35     // update the cache
36     cache[root] = ans;
37     return ans;
38 }
39 bool isValidBST_min_max_memoized_helper(TreeNode* root,
40                                         Cache& min_cache,
41                                         Cache& max_cache)
42 {
43     if (!root)
44         return true;
45
46     bool left_ok = true;
47     if (root->left)
48         left_ok = tree_max_memoized(root->left, max_cache) <= root->val;
49     bool right_ok = true;
50     if (root->right)
51         right_ok = tree_min_memoized(root->right, min_cache) > root->val;
52
53     return (left_ok && right_ok)
54         && (isValidBST_min_max_memoized_helper(
55             root->left, min_cache, max_cache)
56             && isValidBST_min_max_memoized_helper(
57                 root->right, min_cache, max_cache));
58 }
59
60 bool isValidBST_min_max_memoized(TreeNode* root)
61 {
62     Cache min_cache, max_cache;
63     return isValidBST_min_max_memoized_helper(root, min_cache, max_cache);
64 }

```

Listing 19.4: Linear time solution obtained by memoizing Listing ??.

20. Clone a linked list with random pointer

Introduction

This chapter discusses a very interesting problem on (an unconventional type of) linked lists. The kind of linked list we are dealing with here is a singly linked, with an additional pointer that **might** point to another node in the list. The C++ definition of said list is given in Listing 21.1 where we can notice the additional field `random` which differentiates it from the other more canonical linked list definitions seen in other chapters (see Chapter 21 and Listing 21.1).

```
1
2 template <class T>
3 class Node
4 {
5     public:
6     T val{};
7     Node *next{nullptr}; //points to the next element in the list
8     Node *random{nullptr}; //nullptr or points to any other node in the list.
9
10    Node(const T &_val)
11    {
12        val = _val;
13        next = nullptr;
14        random = nullptr;
15    }
16 };
```

Listing 20.1: Definition of a linked list with a pointer to a *random* node.

20.1 Problem statement

Problem 26 Write a function that, given a linked list L of the type defined in Listing ?? returns a deep-copy of L .

In this chapter we will be graphically representing a list as a sequence of pairs of integers. A pair (v, r) represent a node of the list where:

- v is the payload of the node
- r is the index of the node that the random pointer points to. -1 represents `nullptr`.

For instance the sequence $[(7, -1), (13, 0), (11, 4), (10, 2), (1, 0)]$ represent the list shown in Figure ??.

20.2 Clarification Questions

The problem is intended to test list manipulation skills and it is not really about complex algorithm design. As such, questions about the size of the input aimed at getting a feel for the type of time complexity expected does not help much. Rather, it is better to ask

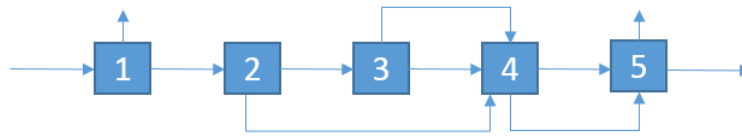


Figure 20.1: Linked list with random pointer. Each node has two outgoing arrows representing. One for the *next* and the other for the *random* node.

questions relating to the structure of the list itself in order to identify any pattern in the list we could take advantage of.

Q.1. Is it guaranteed that at least one not-null random pointer exists?

No, all random pointer might be null.

Q.2. Can a random pointer point to itself?

Yes, you can have a node pointing to itself.

Q.3. Does the random pointer always point to a node ahead in the list?

No, the random pointer can point to any node.

20.3 Discussion

In the following section we are going to have a look at two solutions that are fundamentally different from each other in terms of time and space complexity. The first of the two, the one in Section 20.3.1, uses additional memory (linear amount) while the second one (see Section 20.3.2) works in constant time but, at the cost of being more complex and significantly harder to come up with during an interview.

20.3.1 Linear memory solution

The core idea behind the solution presented in this section can be conceptually divided into 4 steps:

1. We start by creating as many empty nodes as in the original list. We save the pointers to these new nodes in a vector `std::vector<Node<T>*> ptrs;`
2. In the next step we want to map the pointers to the nodes in the original list to their indices (the distance from the head node). We can do this while traversing the list from head to tail. We do this because we want to remember for each node its index in the original list.
3. We can proceed now in fixing all the `next` pointers of the nodes in `ptrs` so that `ptrs[i]->next` points to `ptrs[i+1]`. At this point we have a new singly linked list with broken random pointers. Basically an half-cloned list.
4. We can traverse the original list once again, and if the current node `c` has a not-null random pointer `c->random` we can query the map we filled in step 2 to know the index of the node `c->random`. Once we have this information, we can fix the random pointer of the copy of `c` in `ptrs`.

An implementation of this idea is shown in Listing 20.2

```

1  template <typename T>
2  Node<T> *clone_random_list_map(Node<T> *head)
3  {
4      // empty list case
5      if (!head)
6          return nullptr;
7
8      Node<T> *ans = nullptr;
```



```

9  std::unordered_map<Node<T> *, int> P;
10 std::vector<Node<T> *> ptrs;
11
12 Node<T> *t = head;
13 int idx = 0;
14 while (t)
15 {
16     Node<T> *n = new Node<T>(t->val);
17     ptrs.push_back(n);
18     if (!ans)
19         ans = n;
20     // remember the index of this node t
21     P[t] = idx;
22     t = t->next;
23     idx++;
24 }
25
26 // connect the copy list forward
27 for (int i = 0; i < ptrs.size() - 1; i++)
28     ptrs[i]->next = ptrs[i + 1];
29
30 t = head;
31 idx = 0;
32 while (t)
33 {
34     // which index does t->random has in the original list?
35     // connect the current node with the P[t->random]-th node in the copy list
36     Node<T> *rnd = P.find(t->random) != P.end() ? ptrs[P[t->random]] : nullptr;
37     ptrs[idx]->random = rnd;
38     idx++;
39     t = t->next;
40 }
41 return ans;
42 }

```

Listing 20.2: Linear memory solution.

The first `while` takes care of creating the copy nodes and to fill-up the maps P containing the information about the index of a node.

The subsequence `for` connects the `next` pointers in the copy nodes and the final `while` traverses the list from head to tail and takes care of fixing the copy nodes random pointers using the information in the map P .

Listing 20.2 has linear time and space complexity. The time complexity is already optimal as we cannot do better than linear time considering that in order to create a copy we need to look at all the nodes at least once. The space complexity can be improved though, and as we will see in Section 20.3.2, can actually be brought down to constant.

20.3.2 Constant memory solution

The idea behind the solution presented in this section is to construct the copy such that its nodes are interleaved with the ones from the original list. We are going to embed the copy inside the original list. The solution in this section is divided into three steps where:

1. we create an initial copy interleaved in the original list (see Section 20.3.2.1);
2. then, the random pointers are fixed (see Section 20.3.2.2);
3. finally, the cloned list is extracted-out from the original list and returned (see Section 20.3.2.3).

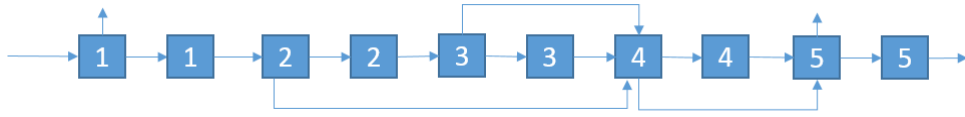


Figure 20.2: Intermediate interleaved list.

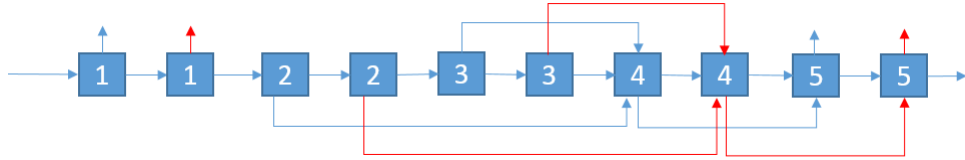


Figure 20.3: Interleaved list with fixed random pointers.

20.3.2.1 Copy interleaved with the original list

This is the easiest of the three steps and it only requires to create a copy of a node at index i and place it at index $i+1$. For instance given the input list^①: $A = [(7, -1), (13, 0), (11, 4), (10, 2), (1, 0)]$ we want to have an interleaved list that look like the following: $A' = [(7, -1), (7, -1), (13, 0), (13, -1), (11, 4), (11, -1), (10, 2), (10, -1), (1, 0), (1, -1)]$ (see Figure 20.2). Every node at even indices is a copy of its predecessor except that it has the random pointer set to `nullptr`.

This step is implemented in the function `fix_random_pointers` in Listing 20.3.

Given the original list is $A = a_0 \rightarrow a_1 \rightarrow \dots \rightarrow a_{n-1}$ and the copy of a A is $B = b_0 \rightarrow b_1 \rightarrow \dots \rightarrow b_{n-1}$ what we achieved in this step a new list I of size $2n$: $I = a_0 \rightarrow b_0 \rightarrow a_1 \rightarrow b_1 \rightarrow a_2 \rightarrow b_2 \rightarrow \dots \rightarrow a_{n-1} \rightarrow b_{n-1}$.

20.3.2.2 Fix the random pointers in the interleaved list

Having the two lists A (the original) and B (the copied) arranged this way is quite useful because we can still visit the original lists and at the same time operate on its mirror by simply modifying the nodes at odd indexes.

An interleaved list always has an even number of nodes. All the ones at even positions $(0, 2, \dots)$ belong to the original lists while all the nodes with odd indexes $(1, 3, \dots)$ to the copy. Given a node $n_{2k} = (x, r)$ at an even index $2k$, we can fix the random pointer for the copy of this node $n_{2k+1} = (x, -1)$ at index $2k+1$ by simply fixing its random pointer to the value pointed by $n_{2k} \rightarrow \text{random} \rightarrow \text{next}$. See Figure 20.3 where the red lines represent the mirrored for the random pointers in the original list and function `split_fix_random_pointers` in Listing 20.3.

20.3.2.3 Extract the cloned list

Once we reach this point, we have basically two copies of the original list (with random pointers fixed) interleaved with each other. All it is necessary at this point is to pull out the cloned list. This is easily achievable as all we need to do is to remove all the odd nodes, and in the process stitch the nodes at even indices together. This step is implemented in the function `split_list` in Listing 20.3).

The full implementation of this solution is shown in Listing 20.3.

```
1 template <typename T>
2 Node<T> *interleave_list(Node<T> *head)
3 {
4     Node<T> *h = head;
5     while (h)
```

^①Remember that $(x, -1)$ is a node containing the value x and with random pointer set to `nullptr`

```

6   {
7       Node<T> *next    = h->next;
8       Node<T> *h_copy = new Node<T>(h->val);
9
10      h_copy->next = next;
11      h->next      = h_copy;
12      h            = next;
13  }
14  return head;
15  }
16
17  template <typename T>
18  void fix_random_pointers(Node<T> *head)
19  {
20      Node<T> *o = head;
21      while (o)
22      {
23          Node<T> *c = o->next;
24          if (o->random)
25          {
26              Node<T> *pointed    = o->random;
27              Node<T> *pointed_c = pointed->next;
28              c->random            = pointed_c;
29          }
30          o = o->next->next;
31      }
32  }
33
34  template <typename T>
35  Node<T> *split_list(Node<T> *head)
36  {
37      Node<T> *o    = head;
38      Node<T> *ans = head->next;
39      while (o)
40      {
41          Node<T> *c      = o->next;
42          Node<T> *cnext = c->next;
43
44          o->next = c->next;
45          if (c->next)
46              c->next = c->next->next;
47          o = cnext;
48      }
49      return ans;
50  }
51
52  template <typename T>
53  Node<T> *clone_random_list_interleave_lists(Node<T> *head)
54  {
55      if (!head)
56          return nullptr;
57
58      interleave_list(head);
59      fix_random_pointers(head);
60      return split_list(head);
61  }

```

Listing 20.3: Constant memory solution.

21. Delete duplicates from Linked List

Introduction

This chapter deals with a fairly simple problem on linked lists that is popular during the preliminary interview stage. Given its ubiquity, it is important to have a good understanding of the solution so that you can implement it quickly and, more importantly, flawlessly during a real interview.

21.1 Problem statement

Problem 27 Given a singly linked list L (see definition at Listing 21.1), return a linked list with no duplicates.

■ **Example 21.1**

Given the input list in Figure 21.1a the function returns the list in Figure 21.1b ■

■ **Example 21.2**

Given the input list in Figure 21.1c the function returns the list in Figure 21.1d ■

```
1  template<typename T>
2  struct Node {
3      T val;
4      Node *next;
5      Node(T x) : val(x), next(nullptr) {}
6  };
```

Listing 21.1: Singly Linked list definition

21.2 Clarification Questions

Q.1. Can the input list be modified?

Yes.

Q.2. Is it guaranteed the input list to be a valid list?

The input list is always a valid singly linked list.

21.3 Discussion

There are several approaches to solving this problem but we will focus on two, the second of which is a refinement of the first intended to allow you to present an elegant and efficient solution in a short time frame.

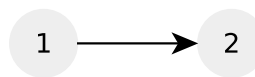
21.3.1 Brute-force

The easiest solution possible is as follows:

1. Create a vector that contains a copy of the list;



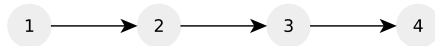
(a) Input list for Example 21.1



(b) List shown in Figure 21.1a with duplicates removed.



(c) Input list for Example 21.2



(d) List shown in Figure 21.1c with duplicates removed.

Figure 21.1: Input and output for the Examples 21.1 and 21.2

2. Remove duplicates from the vector;

3. Create a brand new list with the content of the duplicate-free vector.

This solution is straightforward but not optimal as, while it is optimal in time we are not taking advantage of the fact that we can modify the list in place and thereby avoid creating and returning a brand new list.

A possible implementation is shown in Listing 21.2 where for the step 2 the remove-erase idiom[14] is used to remove duplicates from the vector(the erase part is actually not necessary in this case).

```

1  template <typename T>
2  std::vector<T> list_to_vector(Node<T> *head)
3  {
4      std::vector<T> ans;
5      for (; head; head = head->next)
6          ans.push_back(head->val);
7
8      return ans;
9  }
10
11 template <typename T>
12 Node<T> *list_from_vector(const std::vector<T> &Vs)
13 {
14     Node<T> *tail = nullptr;
15     Node<T> *head = nullptr;
16     for (const auto v : Vs)
17     {
18         Node<T> *n = new Node<T>(v);
19         if (!tail)
20             head = n;
21         else
22             tail->next = n;
23         tail = n;
24     }
25     return head;
26 }
27
28 template <typename T>
29 Node<T> *remove_duplicates_from_linked_list_1(Node<T> *head)
30 {

```

```

31 auto vec_list = list_to_vector<int>(head);
32 // std::sort(std::begin(vec_list), std::end(vec_list)); //not necessary. List
33 // is sorted already
34 vec_list.erase(std::unique(std::begin(vec_list), std::end(vec_list)),
35               std::end(vec_list));
36
37 return list_from_vector(vec_list);
38 }

```

Listing 21.2: C++ solution $O(n)$ time and $O(n)$ space solution to the problem of removing duplicates from a Linked List using `std::remove`.

21.3.2 In-place $O(1)$ space solution

In the solution describe in Section 21.3 used additional space to both remove the duplicates and also to avoid the pain of rearranging the input list by creating a brand new list containing no duplicates. It is, however, possible to write a in-place solution that uses no additional space.

The main idea is that since the list is **sorted**, duplicate elements will be one after the other. We can take advantage of this by simply ignoring pairs of consecutive nodes that have the same payload. Ignored nodes can therefore be deleted. The only complications is that we need to make sure to connect the first occurrence of every Node in the list with each other. It is for this reason that we need to remember the first Node of a stride of the same value.

This idea is demonstrated in Listing 21.3. Note that:

- The base case `if(!head || !head->next) return head;` is making sure that if we are examining the last element of the list (or an empty list) then there is no duplicate to ignore and thus we can return this element.
- otherwise, we are looking at a list with **at least** two elements. These two elements can be potentially the start of a stride of equal elements that we want to ignore. We keep a pointer, `Node<T>* head;` (which is never modified) to the first element of the stride and advance a second pointer, `Node<T>* head_n;`, until we either reach the end of the list or we find an element that is different from the first one, `while (head_n && head -> val == head_n -> val)`. All the advanced elements are deleted. At the end of the loop the second pointer is pointing to either:
 - An element different from the element pointed to by the first one. The stride of equal elements is processed and `head->next` can now point to the second pointer.
 - `nullptr`. We have reached the end of the input list. We are done.

The time and space complexity for this approach are $O(n)$ and $O(1)$, respectively. All the nodes of the list are visited at most once.

```

1 template <typename T>
2 Node<T> *remove_duplicates_from_linked_list_linear_space(Node<T> *head)
3 {
4     if (!head || !head->next)
5         return head;
6
7     Node<T> *head_n = head->next;
8
9     while (head && head_n && head->val == head_n->val)
10    {
11        const auto head_n_n = head_n->next;
12        delete head_n;
13        head_n = head_n_n;

```

```
14     }  
15  
16     head->next = remove_duplicates_from_linked_list_linear_space(head_n);  
17     return head;  
18 }
```

Listing 21.3: C++ solution $O(n)$ time and $O(1)$ space solution to the problem of removing duplicates from a Linked.

21.4 Common Variations and follow-up questions

One follow-up question that an interviewer may ask relates to the deletion of the duplicate nodes. In the Listing 21.3 we see that the nodes are deleted using the `operator delete`, but what if the list was not allocated using `operator new`? The question is left for the reader.

Problem 28 How would the solution change if the nodes were allocated using a custom allocator? (spoiler, a custom deleter is also needed.) ■

22. Generate points in circle uniformly

Introduction

This chapter's problem concerns uniformly generating a (potentially large) number of random points in a circle of a certain radius. Despite its simplicity the problem poses some unexpected challenges. We will discuss the best approach to this problem as well as one solution that many candidates provide which, whilst initially appearing correct actually fails in one crucial aspect (spoiler: it does not distribute points uniformly).

22.1 Problem statement

Problem 29 Write a function that, given a circle of radius r and centered at (x,y) where $r,x,y \in \mathcal{R}$ returns a uniformly distributed point in the circle. ■

22.2 Clarification Questions

Q.1. What exactly does it mean for the point to be uniformly distributed?

It means that every point of the circle has the same probability of being picked/-generated by the function.

22.3 Discussion

Before discussing solutions it is worth mentioning that the fact that the circle is centered at (x,y) makes very little difference and we can continue our discussion as if it were centered at $(0,0)$. This is the case because all the points we generate can then be translated to (x,y) by simply adding x and y to the x -coordinate and y -coordinate of the generated point.

22.3.1 Polar Coordinates - The wrong approach

Let's start by discussing an intuitive, but ultimately incorrect, approach. One might think that in order to pick a point in the circle it is sufficient to

1. Pick a random angle $\theta \in [0, 2\pi[$
2. Pick a random radius $\bar{r} \in [0, r]$
3. Generate the Cartesian coordinates of the point given the radius and the angle (polar coordinates [17]) as (see Figure ??):

$$x = \bar{r} \sin(\theta)$$

$$y = \bar{r} \cos(\theta)$$

Unfortunately, despite its appealing simplicity, this approach is wrong as it fails to produce points that are distributed uniformly in the circle. Before examining the mathematical proof it is instructive to have a look at Figure ?? which is drawing a large number of points on the circle generated using this incorrect solution. As you can see, the points are not generated uniformly as their density is higher towards the center. The bottom

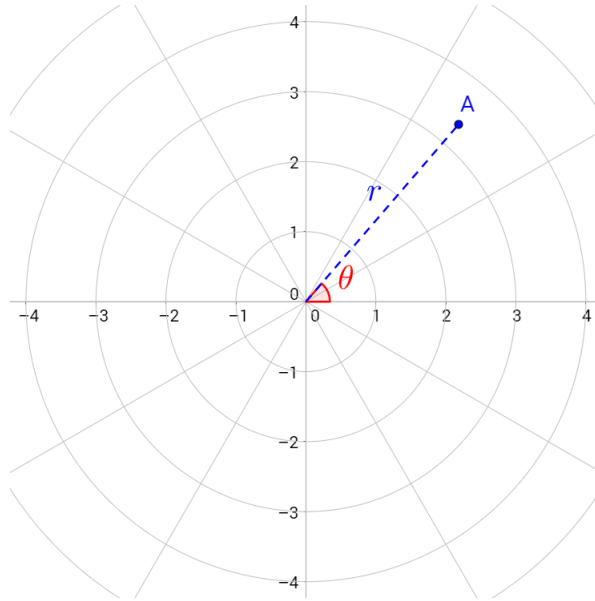


Figure 22.1: Generation of a random point in polar coordinates given a random angle θ and a random radius r .

line is, do not use this solution in an interview. A possible matlab implementation of this buggy approach is shown at 22.1

```
1 function [px, py] = buggy_random_point(radius, x,y)
2     r = rand()*radius;
3     theta = rand()*2*pi;
4     px = r * sin(theta);
5     py = r * cos(theta);
6 endfunction
```

Listing 22.1: Non-uniform random point in a circle generation using Matlab

22.3.2 Loop approach

A good way to ensure that the point density is uniform across on the surface of the circle is to pick a point randomly in an enclosing square and make sure that we discard all the points that lie outside the circle. In other words, we keep asking for a random point ($p_x = \text{rand}(), p_y = \text{rand}()$) in the enclosing square until the following is true: $p_x^2 + p_y^2 \leq r$. In this way we are guaranteed to generate uniformly distributed points because we pick those points from a set of points that are already uniformly distributed in a square, and we exclude those which are not inside the circle. This method is also known as the exclusion method. Figure ?? depicts a large number of points generated with this method.

The downside here is that we might need to generate a number of points in the square before getting lucky and picking one lying in the circle. We need to make on average ≈ 1.2732 tries before getting a point in the circle. This number is the ratio between the are of enclosing square and the area of the enclosed circle i.e. $\frac{(2r)^2}{\pi r^2} = \frac{4}{\pi}$.

A Matlab implementation of this approach is shown in Listing 22.2.

```
1 function [px, py, t] = random_point_loop(radius, x,y)
2     px = 100;
3     py = 100;
4     t=0;
5     while (px*px + py*py > 1)
```

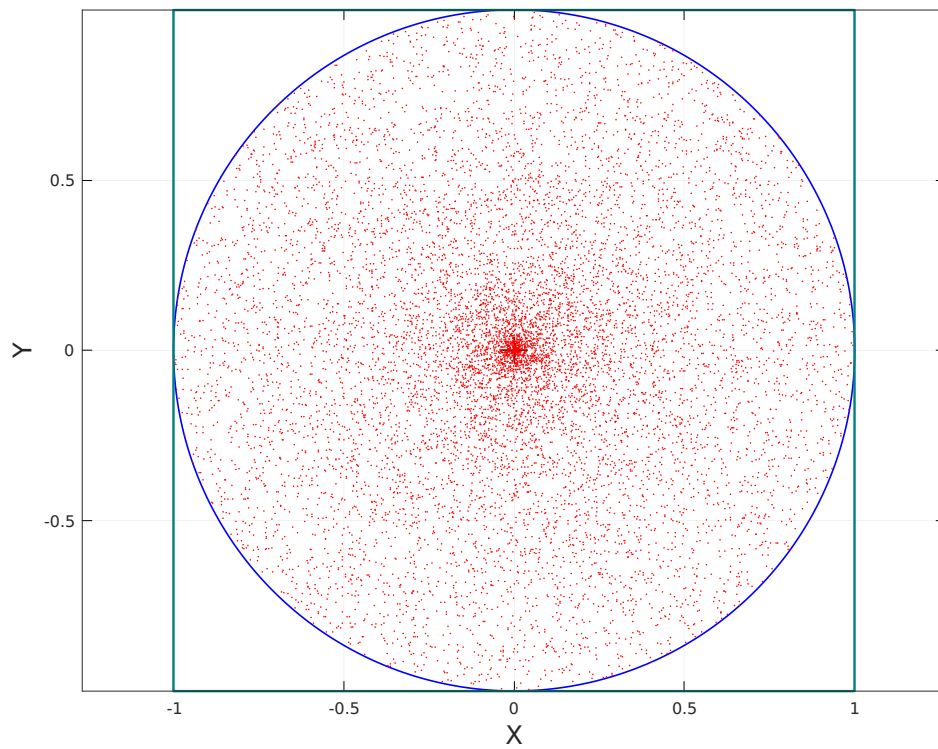


Figure 22.2: Large number of points generated using the approach described in Section 22.3.1. Note that the density of points is not uniform as more points are packed around the center.

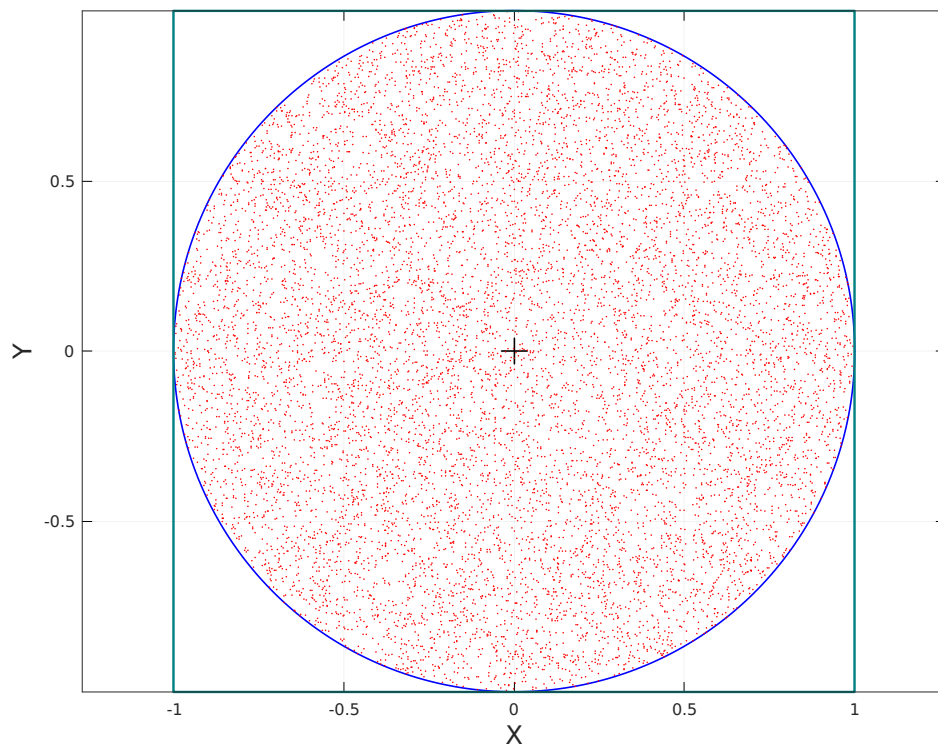


Figure 22.3: Large number of points generated using the approach described in Section 22.3.2.

```

6     signx = 2*randi([0,1])-1;
7     signy = 2*randi([0,1])-1;
8     px = rand()*radius*signx;
9     py = rand()*radius*signy;
10    t=t+1;
11    endwhile
12
13
14    endfunction

```

Listing 22.2: Random point in a circle generation using the exclusion method.

22.3.3 Polar Coordinates - The right approach

In order for the points to be distributed uniformly it is necessary that the average distance between the points be the same regardless of how far they lie from the center of the circle. This means that, looking at the points generated on a circumference of radius 2, there have to be twice as many points as the the number of points on a circumference of radius 1. A circumference that is twice as long translates to needing twice as many points to maintain the same density. Another intuitive way to understand why simply picking a random angle and a random radius is not enough would be to think about having to distribute 10 points at random on a circle of radius 1 and 2. It is clear that the circumference of radius 2 would look emptier than the one with radius 1 simply because there is more circumference to be filled but a constant amount of points.

The fundamental problem with the appraoch described in Section 22.3.1 is that the area of the circle is not uniformly covered. The random radius cuts through the area of the circle and this is the only parameter that affects how the points are going to be distributed across the full area of the circle. Therefore we should focus our attention how we can pick a better radius by making sure that larger radii are picked more often to accommodate for the larger area they define. In other words, we need to ensure that our random function for picking the radius takes the area of our circle into account.

Consider the area A of a circle of radius r i.e. $A = \pi r^2$. We can rearrange the formula so that $r = \sqrt{\frac{A}{\pi}}$. What this formula is really telling us is that the radius is proportional to \sqrt{A} . Now we have a way of choosing the radius that depends on the area of the circle. We can simply pick an area at random and then calculate the radius accordingly. This will make sure that the radius is picked taking into consideration the area of the circle. Figure ?? shows many points generated using this method. As you can see the points are generated uniformly across the area of the circle and the picture looks similar to Figure ??.

A C++ implementation of this method is shown in Listing 22.3. Details on the random number generation in Modern C++ can be found in [8].

```

1  auto generate_random_point_in_circle()
2  {
3      static std::uniform_real_distribution<double> dist_radius(0, 1);
4      static std::uniform_real_distribution<double> dist_angle(0, 2 * M_PI);
5      const auto r      = radius * sqrt(dist_radius(rnd));
6      const auto theta  = dist_angle(rnd);
7      const auto x      = r * cos(theta);
8      const auto y      = r * sin(theta);
9      return std::make_pair{x + x_center, y + y_center};
10 }

```

Listing 22.3: C++ implementation of the function for generating a random point in a circle described in Section 22.3.3

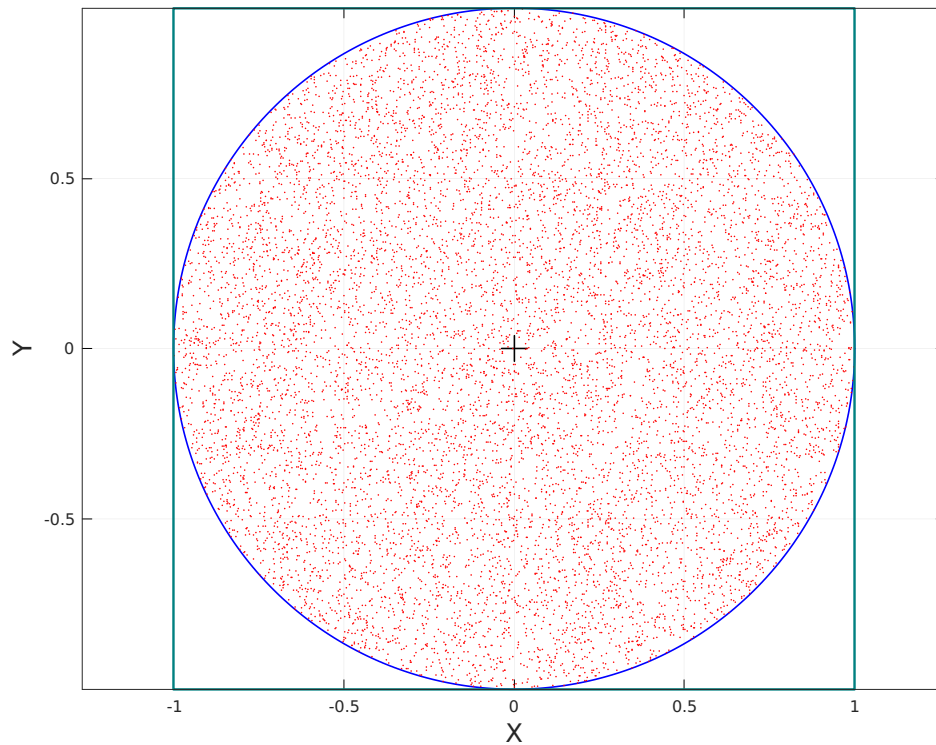


Figure 22.4: Large number of points generated using the approach described in Section 22.3.3.

A Matlab implementation of this approach is shown in Listing 22.4.

```

1 function [px, py] = random_sqrt_area(radius, x,y)
2     area = pi*radius*radius*rand(); %random area
3     r = sqrt(area/pi);
4     theta = rand()*2*pi;
5     px = r * sin(theta);
6     py = r *cos(theta);
7 endfunction

```

Listing 22.4: Random point in a circle generation using polar coordinates and the $\approx \sqrt{A}$ dependency of the radius on the area of the circle.

22.3.4 Conclusion

For both the viable methods for generating random points within a circle which we have discussed the time and space complexity is constant although the one presented in Section 22.3.3 will probably have better performance when included in a hot path i.e. in a loop for the generation of many random points.

All the code used to generate the Figures in this chapter is shown in Listing 22.5.

```

1 function draw_points(n)
2     clf(1);
3     % n is the total number of points
4     px = zeros(1,n);
5     py = zeros(1,n);
6
7     tries = 0;
8     for i =0:n

```

```

9      % [x,y] = buggy_random_point(1,0,0);
10     % [x,y,t] = random_point_loop(1,0,0);
11     [x,y] = random_sqrt_area(1,0,0);
12     % tries = tries + t;
13     px(i+1) = x;
14     py(i+1) = y;
15 endfor
16
17 average = tries/n
18
19
20 % Plot a circle.
21 angles = linspace(0, 2*pi, n);
22 radius = 1;
23 xCenter = 0;
24 yCenter = 0;
25 cx = radius * cos(angles) + xCenter;
26 cy = radius * sin(angles) + yCenter;
27 % Plot circle.
28 plot(cx, cy, 'b-', 'LineWidth', 2);
29 % Plot center.
30 hold on;
31 plot(xCenter, yCenter, 'k+', 'LineWidth', 2, 'MarkerSize', 16);
32 grid on;
33 axis equal;
34 xlabel('X', 'FontSize', 16);
35 ylabel('Y', 'FontSize', 16);
36
37
38
39 % Plot random points.
40 plot(px, py, 'r.', 'MarkerSize', 1);
41
42
43 rectangle('Position', [-1 -1 2 2], 'LineWidth', 3, 'EdgeColor', [0 .5 .5])
44
45 endfunction

```

Listing 22.5: Matlab driver code for the generation of all figures in Chapter 22

23. Best time to buy and sell stock

Introduction

The problem discussed in this chapter is not particularly difficult as it is easily solvable in quadratic time using a brute-force algorithm. However, a more efficient solution is possible and, given that this is exactly the type of question for which interviewers expect fast and elegant solutions, it's worth taking the time to become familiar with the problem structure and the best approaches to solving it.

23.1 Problem statement

Problem 30 You are given prices for a stock for a number n of days. The prices are stored in an array P of length n where each cell i of the array contains the price for the stock on the i^{th} day. You are only permitted to perform **one** buy and **one** sell operations. What is the maximum profit you can achieve given the prices for the stock in P ?

You have to perform the buy operation **before** the sell operation. You cannot buy the stock on the 10th day and sell on the 9th.

■ Example 23.1

Given the array of prices for the stock is: $[7, 1, 5, 3, 6, 4]$, the answer is 5. You can buy on the 2nd day and sell on the 5th. ■

■ Example 23.2

Given the array of prices for the stock is: $[6, 5, 4, 3, 2, 1]$, the answer is 0. There is no way you can make a profit higher than 0 i.e. not buying and not selling. ■

23.2 Clarification Questions

Q.1. Can you perform the buy and sell operation on the same day?

Yes, that is possible.

23.3 Discussion

A profit is achieved when a buy and sell transaction are performed with prices p_b and p_s respectively and $p_b \leq p_s$. In other words, our goal is to buy at a lower price than we sell. The maximum profit is obtained whenever the spread between those two prices is maximum i.e. $\max(p_s - p_b)$

23.3.1 Brute-force

The brute force approach is very straightforward as the only thing we need to do is apply the definition of maximum profit we discussed earlier. For all pairs of ordered index $i \leq j$ we can calculate $P_i - P_j$ and return the maximum among all those profit values. Listing

23.1 shows an implementation of this approach. Note that a profit of 0 is always possible by either not performing any transaction or simply performing the buy and sell on the same day. Thus $j = i + 1$, because it is pointless to calculate the profit for the same day as we know already it will always be 0. For this reason we also limit the buy operation to the day before ($i < n - 1$) the last, because if we want to have any chance of making a profit we need to at least have one day left after the buy to perform the sell operation.

```

1 int buy_sell_stocks_bruteforce(std::vector<int> &P)
2 {
3     const int n = P.size();
4     int ans = 0;
5     for (int i = 0; i < n; i++)
6         for (int j = i + 1; j < n; j++)
7             ans = std::max(ans, P[j] - P[i]);
8     return ans;
9 }

```

Listing 23.1: Brute force $O(n^2)$ solution to the problem of buying and selling stock.

23.3.2 Linear time solution

The solution above can be improved if we look at the problem from slightly different angle. The idea is that we can process the array from the last day to the first and, for each of the days, calculate the **best** profit to be made by selling on any of the days already processed (which occurs later in time).

We keep a variable b with the maximum price seen so far which is initially $-\infty$. The algorithm starts from day n and for each day checks whether buying that day and selling at the price b (the highest price seen so far) would improve the profit found thus far. This approach is correct because the maximum profit happens when the spread between sell and buy price is maximum. The implementation of the idea above is shown in Listing 23.2.

```

1 int buy_sell_stocks_DP(std::vector<int> &P)
2 {
3     const int days = P.size();
4     int highest_so_far = std::numeric_limits<int>::min();
5     int ans = 0;
6     for (int i = days - 1; i >= 0; i--)
7     {
8         highest_so_far = std::max(highest_so_far, P[i]);
9         ans = std::max(ans, highest_so_far - P[i]);
10    }
11    return ans;
12 }

```

Listing 23.2: Dynamic programming linear time, constant space solution to the problem of buying and selling stock.

23.4 Common Variations - Multiple Transactions

23.4.1 Problem statement

Problem 31 You are given an integer array P where $P[i]$ contains the price of a given stock on the i^{th} day.

On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any given time. However, you might engage in multiple transactions over the course of time i.e. you repeat the process of buying a share then

selling it after a while (also the next day) multiple times.

Write a function that given P returns the maximum profit achievable.

Notice that you may not engage in multiple transactions at the same time i.e., you must sell the stock before you buy it again.

■ Example 23.3

Given the array of prices for the stock is $[7, 1, 5, 3, 6, 4]$, the answer is 7. You can buy on the 2nd day and sell on the 3rd and then engage on a second transaction where you buy on the 4th day and sell on the 5th. ■

23.5 Discussion

This might seem like a harder problem at first than the version presented in Section 23.1 but in reality as we will see in Section 23.7 its solution is actually easier.

23.6 Brute force solution

As usual, we start our discussion by quickly presenting the brute force solution. In this case, this means trying all possible sets of transactions (a valid pair of buying and selling operations not overlapping with any other transaction). We can try all possible sets by using recursion cleverly. However, this approach will not take us far because the number of possible sets of transactions grows exponentially. We are showing this approach in Listing 23.3 only because we think its implementation can be somehow instructive.

```
1 int buy_sell_stocks_multiple_transactions_exp_helper(const std::vector<int> &P,
2                                                     const int start)
3 {
4     const int *x = nullptr;
5     int ans = 0;
6     for (int buy_day = start; buy_day < std::ssize(P) - 1; buy_day++)
7     {
8         for (int sell_day = buy_day + 1; sell_day < std::ssize(P); sell_day++)
9         {
10             if (P[buy_day] < P[sell_day]) // pointless to sell otherwise
11             {
12                 x = &P[0];
13                 const int selling_profit = P[sell_day] - P[buy_day];
14                 const int profit_rest_transactions =
15                     buy_sell_stocks_multiple_transactions_exp_helper(P, sell_day + 1);
16                 ans = std::max(ans, selling_profit + profit_rest_transactions);
17             }
18             else
19             {
20                 ans = x != nullptr ? *x : 0;
21             }
22         }
23     }
24     return ans;
25 }
26
27 int buy_sell_stocks_multiple_transactions_exp(const std::vector<int> &P)
28 {
29     return buy_sell_stocks_multiple_transactions_exp_helper(P, 0);
30 }
```

Listing 23.3: Brute force exponential solution to the problem of buying and selling stock with no limits on the number of transactions.

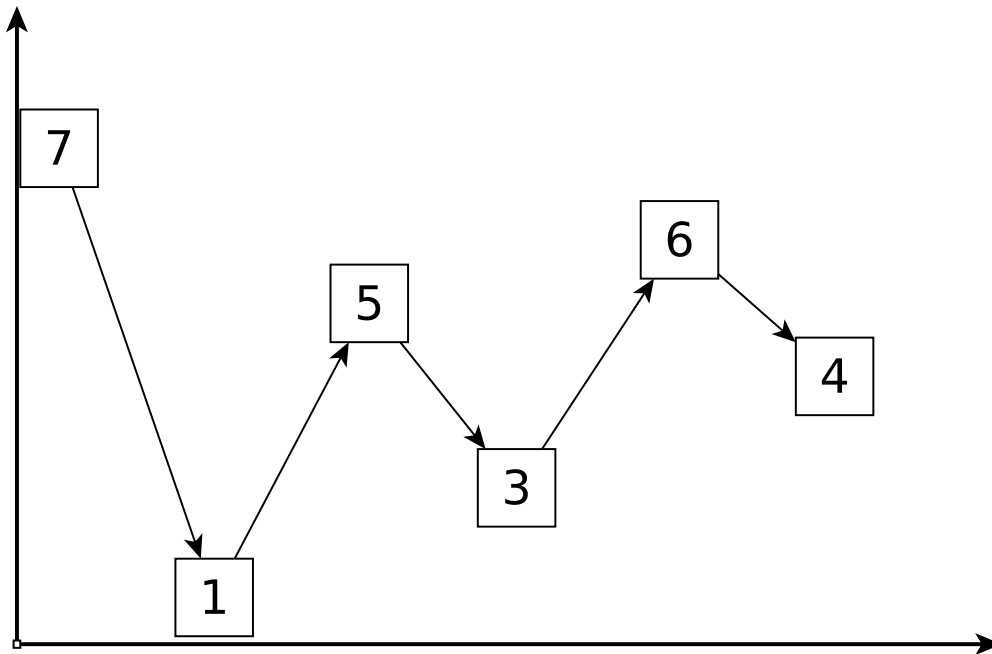


Figure 23.1: Visual representation of Example 23.3

23.7 Linear time solution

The idea is simple and it is clearer once we look at prices plotted on a graph. As you can see in Figure 23.1, the data for Example 23.3 is made of peaks and valleys (unless the data is fully increasing or decreasing). Those are the points of interest because if we buy at valleys and sell at peaks we are able to obtain the maximum profit. One can simply loop through the array and identify those peaks and valleys and calculate the total profit as the sum of the profits along with those points of interest. For instance w.r.t. the example 23.3 there are two pairs valley-peak happening at days 2 and 3 and days 4 and 5, respectively. But, what is a valley and/or a peak exactly? A day i is a valley if $P_i < P_{i-1}$ and $P_i < P_{i+1}$ while is a peak if $P_i > P_{i-1}$ and $P_i > P_{i+1}$. So all it is needed is to identify those pairs of valleys and peaks and we are done.

But do we really need to find peaks and valleys? The answer is not as all it is necessary is to make sure we cash at **all** opportunities we have i.e. in all those cases where we can buy at a lower price we sell. Thus we can process days two at a time and, since there is no limit on the number of transactions, simply buy and sell whenever the spread between buying and selling price is convenient.

The idea above can be implemented as shown in Listing 23.4.

```

1 int buy_sell_stocks_multiple_transactions_lin(std::vector<int> &P)
2 {
3     int ans = 0;
4     for (int i = 0; i < (int)P.size() - 1; i++)
5     {
6         if (P[i] < P[i + 1])
7             ans += (P[i + 1] - P[i]);
8     }

```

```

9     return ans;
10 }

```

Listing 23.4: $O(n)$ time and $O(1)$ space solution to the problem of buying and selling stock with no limits on the number of transactions.

23.8 Common Variations - Best profit with exactly two transactions

23.8.1 Problem statement

Problem 32 You are given an array P where $P[i]$ is the price of a given stock on the i^{th} day. Write a function that given P finds the max profit you can achieve by performing at most two transactions.

Notice that you may not engage in multiple transactions simultaneously, meaning that you must sell the stock before you buy it again.

■ Example 23.4

Given $P = \{3, 3, 5, 0, 0, 3, 1, 4\}$ the function outputs 6. You can buy on the 4th day and sell on the 6th. This transaction yields a profit of 3. You can then perform another transaction with buy and sell dates being the 7th and 8th days, respectively, for a total profit of 6. ■

■ Example 23.5

Given $P = \{1, 2, 3, 4, 5\}$ the function outputs 4. The best strategy here is to perform a single transaction where you buy the first and sell the last day. Notice that you can achieve the same total profit by also performing two transactions. ■

■ Example 23.6

Given $P = \{7, 6, 4, 3, 1\}$ the function outputs 0. It is best in this case not to trade this stock at all, as all possible transaction leads to a loss. ■

23.9 Discussion

This variation might seem at first easier than the one presented in Section 23.4. However, not having a limit on the number of transactions you can make allows us to adopt the strategy in which we make all possible transactions that result in a profit. When we have a constraint on the maximum number of transactions we can make, we are suddenly forced to discard some and keep only the best (two in this specific case, but the same reasoning will apply to the variation in Section 23.12). This makes solving this problem significantly harder.

23.10 DP - Linear time solution

A possible way of solving this problem is by noticing that if you complete the first transaction at day i (day of the sale) then you must have made the buying part of the transaction when the price was at its minimum between day 0 and $i - 1$. Say we made a profit of t_1^i (t_y^x represent the profit of the y^{th} transaction completed on day x). At this point, we still have one more transaction we can make from day $i + 1$ to $n - 1$. Say that the profit of the best second transaction is t_2^i then you end up with a total profit of $t_1^i + t_2^i$. If we have a way of quickly determining t_2 for each i then this problem can be solved relatively easy as we need to return the maximum among $t_1^i + t_2^i$ for all days.

What is exactly t_2^i ? It is the maximum profit we can make by making a **single** between days $i+1$ and $n-1$. Luckily we can calculate t_2^i for all $0 \leq i < n$ using DP. We know that the value of t_2^{n-1} is zero. For every other day i we can calculate the answer to t_2^i if we know the answer for t_2^{i+1} because the value of t_2^i can either be:

- t_2^{i+1}
- $M(j > i) - P[i]$ where $M(j)$ is the highest price of the stock for some day after i .

The reasoning behind this is that the best single transaction you can make with the prices from day i to $n-1$ is either the one you make by buying exactly at day i (and therefore selling at the highest price later) or a transaction you make with the prices for the days ahead i.e. from $i+1$ to $n-1$ for which, crucially, we already have an answer.

Equation 23.1 formalizes this idea where the final answer is the maximum among all values of $T(i)$. $m(i)$ and $M(i)$ contain the information about the smallest and largest element to the left and to the right of index i , respectively. $B(i)$ instead carries the value of the best single transaction that you can make with any values to the left of index i (inclusive). In other words, what Equation 23.1 states is that the value of the final answer is the value of the best transaction you can make by selling at exactly index $i^{\textcircled{1}}$ plus the value of the best transaction you can make with any of the prices to the right of i . If you take the maximum among all indices, then it is clear that this quantity is indeed the value you can achieve by performing two transactions at most (when $i=0$ or $i=n-1$ we are in practice making only a single transaction.).

$$\begin{cases} B(|I|-1) = 0 \\ B(i) = \max(B(i+1), M(i) - P(i)) \\ M(i) = \max(P(i), P(i+2), \dots, P(n-1)) \\ m(i) = \min(P(0), P(1), \dots, P(i)) \\ T(i) = (P[i] - m(i)) + B(i) \end{cases} \quad (23.1)$$

Listing 23.5 shows an implementation of this idea.

```

1
2 auto best_transaction_right(const std::vector<int>& prices)
3 {
4     const auto size = prices.size();
5     std::vector<int> ans(size, 0);
6     int max_right = prices[size - 1];
7     ans[size - 1] = (max_right - prices[size - 1]);
8     for (int i = size - 2; i >= 0; i--)
9     {
10         max_right = std::max(max_right, prices[i]);
11         ans[i] = std::max(ans[i + 1], max_right - prices[i]);
12     }
13     return ans;
14 }
15
16 int buy_sell_stocks3_DP(vector<int>& prices)
17 {
18     const auto best_right = best_transaction_right(prices);
19     const auto size = prices.size();
20
21     int ans = 0;
22     int min_left = prices[0];
23     for (int i = 0; i < size; i++)

```

^①For which you must have bought at the minimum between 0 and $i-1$ to make the best profit.

```

24 {
25     min_left = std::min(min_left, prices[i]);
26     ans      = std::max(ans, prices[i] - min_left + best_right[i]);
27 }
28 return ans;
29 }

```

Listing 23.5: $O(n)$ time and space DP solution.

The code works by calculating the values of the best transaction we can make with the values to the right of each index i and stores this info in an array of size n (this is B in Equation 23.1). The code then proceeds in calculating the answer by looping over all days and maintaining a variable `min_left` which contains the minimum element seen so far: this value is useful in calculating the profit for the first transaction we can make by selling at index i . The loop goal is to calculate $T(i)$ of Equation 23.1 and remember the maximum value ever calculated (in the variable `ans` which is eventually returned).

Listing 23.5 has linear time and space complexity.

23.10.1 Linear time and constant space

Suppose we make some profit p_1 by doing our first transaction. When it is time to make the second transaction at the day i we are going to of course pay $P[i]$. Now, for us, the net effective price that we are spending from our pocket is actually $P[i] - p_1$, because we already have p_1 currency unit in our hand. When it is time to sell our second purchase at time $j > i$ we will do at price $P[j]$ and the net profit p_2 for the second transaction will be $p_2 = P[j] - (P[i] - p_1)$. All we have to do is maximixing the value of p_2 as shown in Listing 23.6

```

1 int buy_sell_stock3_linear_space(const std::vector<int>& P)
2 {
3     if (P.empty())
4         return 0;
5
6     int buy1      = std::numeric_limits<int>::max();
7     int profit1   = std::numeric_limits<int>::min();
8     int buy2      = std::numeric_limits<int>::max();
9     int profit2   = std::numeric_limits<int>::min();
10    for (int i = 0; i < P.size(); i++)
11    {
12        buy1      = std::min(buy1, P[i]);
13        profit1   = std::max(profit1, P[i] - buy1);
14        buy2      = std::min(buy2, P[i] - profit1);
15        profit2   = std::max(profit2, P[i] - buy2);
16    }
17    return profit2;
18 }

```

Listing 23.6: $O(n)$ time and constnt space solution.

23.11 Variation - Best profit with at most k transactions

23.11.1 Problem statement

Problem 33 You are given an integer K and an array P where $P[i]$ is the price of a given stock on the i^{th} day. Write a function that given P and K finds the maximum profit you can achieve by performing at most K transactions.

Notice that you may not engage in multiple transactions simultaneously, meaning

that you must sell the stock before you buy it again.

■ **Example 23.7**

Given $K = 2$ and $P = \{2, 4, 1\}$ the function outputs 2. You can buy on the 1st day and sell on the 2nd. This transaction yields a profit of 2. Notice that you only use one of the two transactions allowed. ■

■ **Example 23.8**

Given $K = 2$ and $P = \{3, 2, 6, 5, 0, 3\}$ the function outputs 7. You can buy on the 2nd day and sell on the 3rd. You can then make another transaction where you buy on the 5th and sell on the 6th day for a total profit of $4 + 3 = 7$. ■

■ **Example 23.9**

Given $K = 4$ and $P = \{4[3, 2, 6, 5, 0, 3, 3, 8, 2, 3, 5, 5, 9]\}$ the function outputs 19. Notice that the function would output 19 even when $K = 3$. ■

23.12 Discussion

The variation discussed here is a generalization of the one discussed in Section 23.8 where we are allowed to make up to K transaction where K is given to us as a parameter. Clearly when $K = 2$ this variation is equivalent to the one in Section 23.8. However, not knowing precisely the upper bound on the number of possible transactions complicates things a bit (but not too much).

Let's start with a simple observation: if $K > \frac{|P|}{2}$ then there is no limit on the number of transactions we can make and we can immediately fall back on the same approach used Listing 23.4. Despite being interesting is not key to solving this problem in its generality albeit it might, in practice, speed up the actual runtime for these specific cases.

23.13 $O(n^2K)$ time and $O(nK)$ space

We are going to attempt to write a recursive formula similar to Equation 23.1 for the variation number 3 of this problem, that describes the answer in terms of subproblems. Let $DP(i, j)$ be the maximum profit possible by only considering prices up to index i and by using at most $j \leq K$ transactions. $DP(0, j) = DP(i, 0) = 0$, as it is impossible to complete a transaction in just one day (the very first) as well as when you can't even make a single transaction. When calculating $DP(i, j)$ for the general case, we should consider that its value can either be:

- the same as $DP(i-1, k)$ i.e. the value of the maximum profit you can make up to index $i-1$ with at most k transaction (in other words we are saying we ignore the price at index i);
- or the maximum profit we can make by performing a transaction that ends in a sell at exactly index i . To calculate this value we need to find the best place to perform the buy-side of the transaction which can take place at every index $l < i$. This can be calculate with the following formula: $\max DP(l, j-1) + (P[i] - P[j])$.

We can rewrite the general case $DP(i, j)$ in Equation 23.2 as

$$DP(i, j) = \max(DP(i-1, k), P[i] + \max(DP(l, j-1) - P[j])) \quad \forall 0 \leq l < i$$

as $P[i]$ is constant in the innermost max expression (l is the only variable there).

Equation 23.2 summarises what we have discussed so far for DP .

$$\begin{cases} DP(0, j) = 0 \\ DP(i, 0) = 0 \\ DP(i, j) = \max \left\{ DP(i-1, k), P[i] + \underbrace{\max \{ DP(l, j-1) - P[l] \}}_{\forall 0 \leq l < i} \right\} \end{cases} \quad (23.2)$$

We could already proceed in turning Equation 23.2 naively into code and we would obtain a working solution with $O(nK)$ space and $O(n^2K)$ time complexities, respectively, as shown in Listing 23.7.

```

1 int buy_sell_stock4_DP_unoptimized(const int K, const std::vector<int>& P)
2 {
3     const int n = P.size();
4     if (K == 0 || n <= 1)
5         return 0;
6     std::vector<std::vector<int>>> DP(n + 1, std::vector<int>(K + 1, 0));
7
8     for (int k = 1; k <= K; k++)
9     {
10         for (int i = 1; i < n; i++)
11         {
12             int best_l = 0;
13             for (int l = 0; l < i; l++)
14             {
15                 best_l = std::max(best_l, DP[l][k - 1] + P[i] - P[l]);
16             }
17             DP[i][k] = std::max(DP[i - 1][k], best_l);
18         }
19     }
20     return DP[n - 1][K];
21 }
```

Listing 23.7: $O(n^2K)$ time and $O(nK)$ space DP solution.

Listing 23.7 is correct and it would actually not be that bad if we could come up with it during an actual interview.

23.14 $O(nK)$ time and space

However, the solution discussed above can be improved dramatically by noticing that we do not need to calculate the value of the innermost max in the third case of Equation 23.2 as shown in Listing 23.8.

```

1 int buy_sell_stock4_DP_time_optimized(const int K, const std::vector<int>& P)
2 {
3     const int n = P.size();
4     if (K == 0 || n <= 1)
5         return 0;
6     std::vector<std::vector<int>>> DP(n + 1, std::vector<int>(K + 1, 0));
7
8     for (int k = 1; k <= K; k++)
9     {
10         int max_left = DP[0][k - 1] - P[0];
11         for (int i = 1; i < n; i++)
12         {
13             DP[i][k] = std::max(DP[i - 1][k], max_left + P[i]);
14             max_left = std::max(max_left, DP[i][k - 1] - P[i]);
15         }
16     }
17 }
```

```

16     }
17     return DP[n - 1][K];
18 }

```

Listing 23.8: $O(nK)$ time and $O(nK)$ space DP solution.

The important and most challenging part of this solution is to make sure that the quantity $L = \underbrace{\max\{DP(l, j-1) - P[j]\}}_{\forall 0 \leq l < i}$ is calculated as we iterate incrementally over all values of i . To understand why let's look at the particular values of L for some incremental values of i :

- if $i = 1$ then $L_1 = \max\{DP(0, j-1) - P[0]\}$
- if $i = 2$ then $L_2 = \max\{DP(0, j-1) - P[0], DP(1, j-1) - P[1]\}$; but crucially $DP(0, j-1) - P[0] = L_1$
- if $i = 3$ then $L_3 = \max\{DP(0, j-1) - P[0], DP(1, j-1) - P[1], DP(2, j-1) - P[2]\}$; thanks to the fact that $L_2 = \max\{DP(0, j-1) - P[0], DP(1, j-1) - P[1]\}$ we can simply this expression as $L_3 = \max\{L_2, DP(2, j-1) - P[2]\}$. crucially $DP(0, j-1) - P[0] = L_1$
- ...
- In general $L_i = \max\{L_{i-1}, DP(i, j-1) - P[i]\}$

This approach allows us to avoid the loop over l each time we calculate an entry in $DP(i, j)$ and to lower the time complexity $O(nK)$: a good improvement w.r.t. the previous solutions!

23.15 $O(|P|K)$ time and $O(|P|)$ and space

If we pay attention to either the main loop in Listing 23.7 or Equation 23.2 we notice that in the innermost loop for i we never ever reference in DP any value of k that is lower than $k-1$. This observation opens for the an space optimization opportunity where the size of DP goes down from $n \times K$ to $n \times 2$. We can use one column of DP to refer to the current value of k and the other to $k-1$ and after the innermost loop is completed we can swap these two columns. For example at the first iteration of the outermost loop ($k=1$), $DP[_][1]$ refers to the values for $k=1$ and $DP[_][0]$ to the values for $k-1=0$. When the innermost loop ends and the outermost starts again, the two columns are swapped and therefore $DP[_][0]$ refers to $k=2$ while $DP[_][1]$ to $k-1=1$. This process continues until both loops end.

Listing 23.9 shows an implementation of this idea.

```

1  int buy_sell_stock4_DP_time_and_space_optimized(const int K,
2                                                    const std::vector<int>& P)
3  {
4      const int n = P.size();
5      if (K == 0 || n <= 1)
6          return 0;
7
8      std::vector<std::vector<int>> DP(n + 1, std::vector<int>(2, 0));
9
10     int curr_k = 1;
11     int prec_k = 0;
12     for (int k = 1; k <= K; k++)
13     {
14         int max_kminus1_to_left_of_i = DP[0][prec_k] - P[0];
15         for (int i = 1; i < n; i++)
16         {
17             DP[i][curr_k] =
18                 std::max(DP[i - 1][curr_k], max_kminus1_to_left_of_i + P[i]);

```

```

19     max_kminus1_to_left_of_i =
20         std::max(max_kminus1_to_left_of_i, DP[i][prec_k] - P[i]);
21     }
22     swap(curr_k, prec_k);
23 }
24 return DP[n - 1][prec_k];
25 }

```

Listing 23.9: $O(nK)$ time and $O(n)$ space DP solution.

The code is extremely similar to Listing 23.8, with the only difference being the size of `DP` is now $O(n)$ and we use two variables `curr_k` and `prec_k` to keep track of the column assigned to the “current” and “previous” values of k . Notice how at the end of each innermost loop, the two columns are swapped by simply swapping around the values of `curr_k` and `prec_k`.

24. Find the cycle in a Linked list

Introduction

The topic of this chapter is linked-lists i.e. linear collections of elements whose order, unlike an array, is not dictated by their ordering in memory. As they are one of the most simple and commonly data structures it is reasonable to assume that they will come up during interview and should, therefore, form part of your preparation.

The major benefit that lists offer over conventional arrays is that elements in the list can be efficiently (in constant time) removed and inserted without the need to reorganize and perform a complete restructuring of all the data.^① As a result, linked lists are often used to implement more complex data structures where this insertion and deletion cost is crucial; for example, associative arrays. They do, however, also have quite a number of drawbacks. For instance: 1. memory consumption (as for each node of the list you also pay a price as has to remember the next and/or previous nodes). 2. they offer sequential access. Accessing a node costs linear time. 3. cache unfriendly.

A linked list is, at a high level of abstraction, a collection of so-called nodes or elements each of which (except the last) contains a pointer to the next one. Each node also carries payload data which is the information you ultimately want to be stored. The standard linked list has two special nodes:

- the head that is not pointed to by other elements and is the first of the elements.
- the tail, which is a node that has no next element, and is, not surprisingly, the last of the elements.

In some particular cases during the manipulation of the list you might end up with a broken or corrupted list where the tail node no longer exists, meaning that each of the elements in the list is pointing to some other node. In this situation a loop forms and the list becomes what it known as a *circular* list. In this chapter we will investigate how we can find out whether: 1. a list is circular and if it is; 2. how to identify the first element of the loop..

24.1 Problem statement

Problem 34 Given a singly linked list (definition in Listing 21.1 at page 102) determine whether the list contains a loop.

- If it does, return the the node where the loop starts
- otherwise, return `nullptr`

For the rest of the chapter we will use an array of integers to represents the nodes of the list and a single integer to represent the node that the last element of the list connects to in order to create a cycle or `-1` if there is no cycle. For instance the array $L = [1, 2, 3, 4]$ and the integer 2 represent the list shown in Figure 24.1.

^①For arrays, the cost of inserting or deleting an element is linear as you need to: 1. possibly enlarge the allocated space for the array 2. copy all the elements (minus or plus the element you want to remove or insert) in the new memory space.

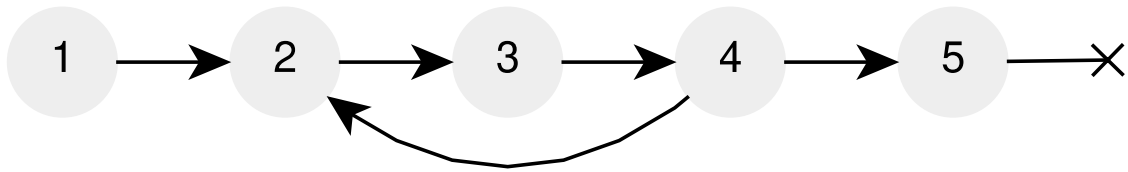


Figure 24.1: Example of linked list with a cycle.

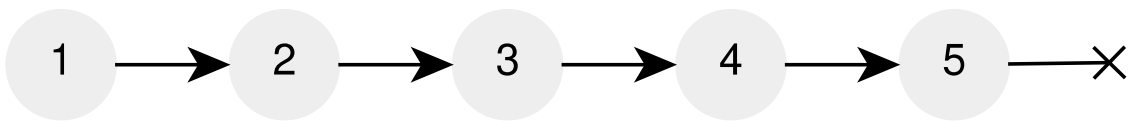


Figure 24.2: Example of linked list with no cycle.

■ Example 24.1

Given the List $\{[1,2,3,4,5],2\}$, the function returns the address of the node 2. See Figure 24.1. ■

■ Example 24.2

Given the List $\{[1,2,3,4,5],-1\}$, the function returns `nullptr`. See Figure ??.

24.2 Discussion

Considering this is a very well-known problem we will not spent time on the obvious brute-force solution. Instead we will concentrate first on an optimal in time solution with linear space, and then examine how to improve it by lowering the space complexity to constant.

All solution implementations in this chapter uses the Linked list definition shown in Listing 24.1;

```

1  template <typename T>
2  struct Node
3  {
4      T val;
5      Node *next;
6      Node() = default;
7      Node(T x) : val(x), next(nullptr)
8      {
9      }
10 };

```

Listing 24.1: Singly linked-list node definition.

24.2.1 Linear time and space solution

This problem has many similarities with the problem of finding a duplicate in a collection and, as such, we can approach it in a similar way. The idea is to visit the list and store in a hash-set the address of the node **already** visited. While visiting a new node, we first

check if that node was already visited, and if the answer is yes then we can stop as we have found the starting point of a loop. If we reach the end of the list and we were not be able to find a duplicate then there is no loop and we can return `nullptr`. A possible implementation of this idea is shown in Listing 24.2.

```

1  template <typename T>
2  Node<T>* detect_cycle_linear_time_space(Node<T>* head)
3  {
4      using Node_ptr = Node<T>*;
5      std::unordered_set<Node_ptr> visited;
6
7      while (head)
8      {
9          // has the current node already been visited?
10         if (visited.find(head) != visited.end())
11             return head;
12         // if not, then remember that we did now
13         visited.insert(head);
14         // advance one node in the list
15         head = head->next;
16     }
17     return nullptr;
18 }
```

Listing 24.2: Linear time and space solution to the problem of detecting a cycle in a linked list where an hashset is used to remember already visited nodes.

24.2.2 Slow and fast pointer solution - Floyd's algorithm

This algorithm[[cit::wiki::floyd](#)] uses the fact that, like clock hands, things iterating on a cycle at different speeds will eventually meet at some point in the future. Consider two runners R_1 and R_2 with velocities V_1 and $V_2 = 2V_1$ respectively (R_2 goes twice as fast as R_1), starting their run from the same point in a circular stadium. They will meet again when the slower runner reaches the starting point for the second time. This occurs because, by the time the slower runner has completed a half lap of the track, the faster runner will have completed a full lap; and by the time the slower finishes a full lap, arriving at the starting point again, the faster will have completed a second full lap. We can use this fact to detect a cycle in a linked list even if for the cycle detection problem things might be a bit more complicated because the two runners might start going in a loop at the same time (the list potentially has a first part the is not part of the loop as can be seen in Figure 24.1).

The rest of this section will outline the technical specifics so, if you are pressed for time, it is possible to skip to the implementation shown in Listing 24.3 which is quite self-explanatory given the underlying algorithm works fairly intuitively. If you are interested in the details of why it works, read along.

Consider two iterators p, q with velocities $v_p = 1, v_q = 2$ respectively. Suppose the **cycle**(not the entire list) has length n . We can have two scenarios depending on the index A of the starting node of the cycle:

1. the cycle starts at $A < n$.
2. or starts at $A \geq n$.

For the case (1) when the slower iterator reaches A the faster is at location $2A$ (which might mean that the faster iterator looped around the cycle already). How many iterations k will it take before they meet and at which node will this meeting occur? The situation is

described by the following congruences:

$$A + kv_p \equiv 2A + k2v_p \pmod{n} \quad (24.1)$$

$$2A + k2v_p \equiv A + kv_p \pmod{n} \quad (24.2)$$

$$A + k2v_p \equiv kv_p \pmod{n} \quad (24.3)$$

$$A + kv_p \equiv 0 \pmod{n} \quad (24.4)$$

$$A + k \equiv 0 \pmod{n} \quad (24.5)$$

which has solution $k = n - A$. This means that they will meet after $k = n - A$ iterations of the slower iterator, i.e. at A nodes before the beginning of the cycle and we can use this fact to count A nodes from the beginning of the list in order to find the starting point of the cycle.

Once the iterators meet **in the cycle** we can move the fast iterator back to the beginning of the list and iterate forward one node per step with both iterators until they match again. When we move the fast iterator back at the head of the list, **both iterators are A nodes away from the beginning of the cycle**. Because of this, when we move both of them by one, they will eventually meet exactly at that node A i.e. the beginning of the cycle.

Let's consider now the case (2) i.e. when $A \geq n$. This means that by the time the slower iterator reaches the beginning of the cycle the faster one has completed more than one cycle. What will then be the starting point for the faster one? We argue that once p reaches A , q is at node $2A$ but since $A > n$, this means that it will be at position $A + (A \pmod{n})$. We can now use similar arguments to the previous example and write:

$$A + kv_p \equiv A + (A \pmod{n}) + (k2v_p \pmod{n}) \quad (24.6)$$

$$A + (A \pmod{n}) + k2v_p \equiv A + kv_p \pmod{n} \quad (24.7)$$

$$(A \pmod{n}) + kv_p \equiv 0 \pmod{n} \quad (24.8)$$

$$(A \pmod{n}) + k \equiv 0 \pmod{n} : \text{because } v_p = 1 \quad (24.9)$$

$$(24.10)$$

which has solution $k = n - (A \pmod{n})$. This means that the meeting point is $A \pmod{n}$ nodes before the beginning of the cycle. If we do the same operations as previously (when $A < n$), we obtain the same result. Iterators will meet at the beginning of the cycle. This happens because advancing q makes p cycle possibly several times (remember that $A \geq n$) and it will clearly stop at $A + (n - A \pmod{n}) + A \pmod{n} = A + n \pmod{n} = A$. In other words; the slower pointer is at first at node number $A + (n - A \pmod{n})$. We can write $A = bn + r$ where $r = A \pmod{n}$. After A advancing steps it will be at location $A + (n - A \pmod{n}) + bn + r \pmod{n}$. Since $bn \pmod{n} = 0$ the result follows.

As an example, consider a list with a cycle of length $n = 4$ starting at node number 10. The first part of the algorithm tells us that the nodes will meet at node $10 + 4 - 10 \pmod{4} = 12$. Moving the fast pointer back to the head of the list and iterating one node per time; both iterators will lead the slower pointer to node:

Figure 24.3 depicts how the algorithm works on a list of 8 nodes with a cycle of length 4 starting at node number 4. After 5 steps the slow (p) and fast (q) iterators point to the same node i.e. node number 6. After a new phase starts, with the slow pointer being moved to the head of the list and continues with both iterators moving forward by 1 until they meet again. They will meet again at the beginning of the cycle.

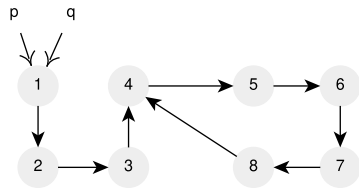
An implementation of the Floyd's algorithm is shown in Listing 24.3.

```

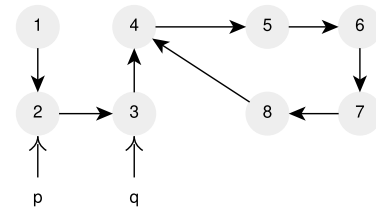
1  template <typename T>
2  Node<T> *detect_cycle_constant_time(Node<T> *head)
3  {
4      Node<T> *n1, *n2;
5      n1 = n2 = head;
6
7      while (n1 && n2)
8      {
9          n1 = n1->next;
10         n2 = n2->next;
11         if (n2)
12             n2 = n2->next;
13         else
14             break;
15
16         if (n1 == n2)
17             break;
18     }
19     // second phase floyds's algorithm
20     if (n1 == n2)
21     {
22         n2 = head;
23         while (n1 != n2)
24         {
25             n1 = n1->next;
26             n2 = n2->next;
27         }
28         return n1;
29     }
30     return nullptr;
31 }

```

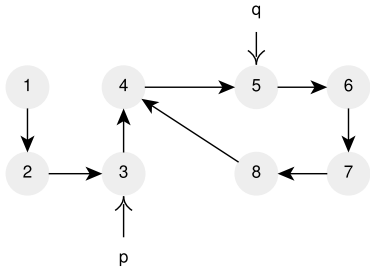
Listing 24.3: Floyd's algorithm, linear time, constant space solution to the problem of detecting a cycle in a linked list.



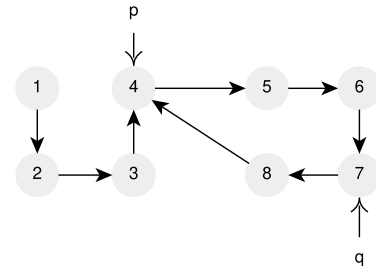
(a) At the beginning $p = q = 1$. The slow and fast forward: $p = p + 1$, $q = q + 2$.



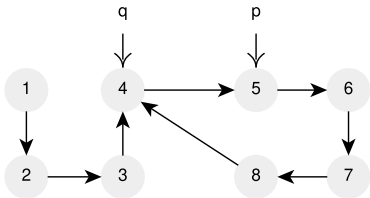
(b) $p \neq q$, thus: $p = p + 1$, $q = q + 2$



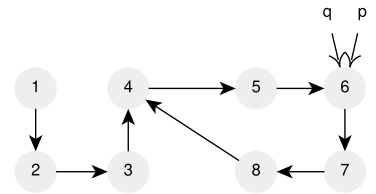
(c) $p \neq q$, thus: $p = p + 1$, $q = q + 2$



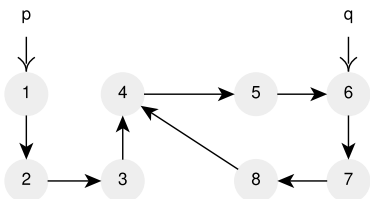
(d) $p \neq q$, thus: $p = p + 1$, $q = q + 2$



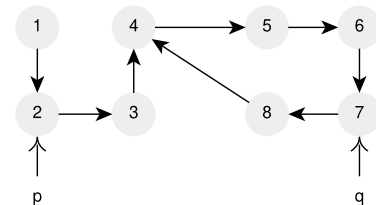
(e) $p \neq q$, thus: $p = p + 1$, $q = q + 2$



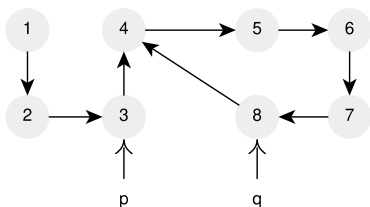
(f) $p = q$. The fast and slow movements stop.



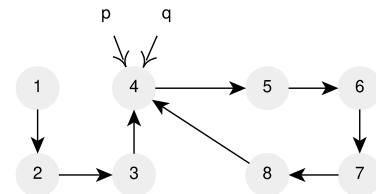
(g) p is reset to the beginning of the list. q is not moved. From now on p and q are moved one step at the time.



(h) $p \neq q$, thus: $p = p + 1$, $q = q + 1$



(i) $p \neq q$, thus: $p = p + 1$, $q = q + 1$



(j) $p = q$. The algorithm stops, and both p and q point to the beginning of the cycle.

Figure 24.3: Execution of the Floyd's algorithm. The slow and fast pointers are initialized to the head of the list (see Figure 24.3a) and immediately moved forward at different speeds (Figure 24.3b). They continue to move forward at different speed until their values mismatch (from Figure 24.3b to 24.3f). At this point p is moved back to the head of the list (Figure 24.3g). From now on the pointers are moved at the same speed of 1 and they continue to move forward until they match again (from Figure 24.3h to 24.3j). p and q now point to the beginning of the cycle in the list.

25. Reverse a singly linked list

Introduction

In this chapter we are going to have a look at a problem based on reversing a singly linked list. Despite the fact that this is one of the fundamental structures in computer science and is also an extremely popular interview question, it often trips up prospective candidates and is usually a cause for immediate rejection. As such, it is worth spending a bit of time on it to ensure a solid grasp of the optimal solutions.

The problem has a simple definition as all it asks us to do is reverse a given list. We will discuss how we can approach this problem both a recursive and an iterative manner. We will also examine a slightly harder variation that is often asked as a follow-up although we leave the solution to that one for the reader.

25.1 Problem statement

Problem 35 Create a function that, given a singly linked list L , reverses it and return the pointer to the new head of L . L is given as a pointer to the first node of the list. The definition of the node is given in Listing 25.1.

■ Example 25.1

Given the $L = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$, the function modifies it into $L = 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$ and returns a pointer to the node 5, the new head of the list. ■

```
1  template <typename T>
2  struct Node
3  {
4      T val;
5      Node *next;
6      Node() = default;
7      Node(T x) : val(x), next(nullptr)
8      {
9      }
10 };
```

Listing 25.1: Singly linked-list node definition.

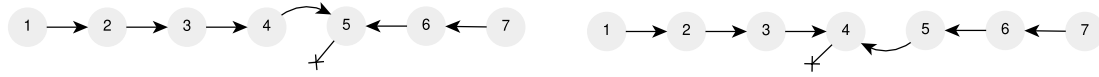
25.2 Clarification Questions

Q.1. Can the input list be empty?

Yes.

Q.2. Can I assume L is not corrupted by e.g. containing cycles?

Yes, the input list is a singly linked list with no cycles.



(a) Nodes arrangements in the middle of the recursive process for node 5 (b) Nodes arrangements after performing the recursive call for node 5.

25.3 Discussion

Solving this problem using linear additional space is trivial as we can iterate over the list and for each push the address of each of its nodes in a stack. We can then pop them one at a time while making sure they are connected in the same order they are popped out. Listing 25.2 shows a possible implementation of this idea. The time and space complexity of this approach is $O(n)$.

```

1  template <typename T>
2  Node<T>* list_reverse_linear_space(Node<T>* L)
3  {
4      if (!L)
5          return nullptr;
6
7      std::stack<Node<T>*> nodes;
8      Node<T>* it = L;
9      while (it)
10     {
11         nodes.push(it);
12         it = it->next;
13     }
14
15     Node<T>* new_head = nodes.top();
16     nodes.pop();
17
18     it = new_head;
19     while (!nodes.empty())
20     {
21         const auto it_next = nodes.top();
22         nodes.pop();
23         it->next = it_next;
24         it      = it_next;
25     }
26     it->next = nullptr;
27     return new_head;
28 }
  
```

Listing 25.2: Linear time and space complexity solution using a stack to reverse the nodes in the list.

We can, however, avoid using additional space in the form of a `std::stack` and rely on the implicit stack we get when we perform recursive calls. In order to take advantage of it, however, it is convenient to shift our view of the problem as follows:

Imagine we have a list such that it is already reversed after its k^{th} node. How can we then reverse the rest of it? Let's have a look at Figure 25.1a depicting this scenario where $k = 4$. As we can see the list is already reversed from node 5 onwards and all we have to do is to have it pointing to node 4 and make 4 point to nothing. More generically what we want to achieve is to make the node $k + 1$ (L_{k+1}) point to the node k (L_k). We can achieve this by doing: $L_k \rightarrow next \rightarrow next = L_k(L_{k+1})$. With regard to Figure 25.1a $L_k \rightarrow next$ is 5 and $L_k \rightarrow next$ is pointing to nothing. After these operations what we are left with is the list shown in Figure 25.1b. If we do that for each of the nodes eventually we are left with a reversed list.

But what about the new head of the list? What should each recursive call return? This is actually fairly straightforward. Whenever we reach the end of the list^① we return the current node - which is effectively the new head of the reversed list - and we keep propagating that value for all the recursive calls.

To summarise, for each recursive call we first reverse the rest of the list and we get back the head of the reversed list. We can now take care of reversing the link from the current node to the next and return the head we got back from the recursive call. Listing 25.3 shows a possible implementation of this idea. Note that despite this solution not explicitly using any additional space, it still requires spaces for the activation frames of all the recursive calls. As such, its complexity remains equivalent to the one in Listing 25.2.

```

1  template <typename T>
2  Node<T>* list_reverse_recursive(Node<T>* L)
3  {
4      if (!L || !(L->next))
5          return L;
6
7      auto reverse_next_head = list_reverse_recursive(L->next);
8      L->next->next          = L;
9      L->next                = nullptr;
10     return reverse_next_head;
11 }

```

Listing 25.3: Recursive linear time and space complexity solution to reverse the nodes in the list.

25.3.1 Constant space

It is impossible to solve this problem faster than linear time as each node must be accessed at least once; but we can bring the space complexity down to constant. We do this by reversing the list, two nodes at a time, from the head to the tail. Assuming *L* has at least two nodes (if not we are in a trivial case in which the list is already reversed and *L* is also the head of the reversed list) then we can always maintain two pointers to the current element *curr* and its next *curr_next*. We know that *curr* points to *curr_next* but what we really want to achieve is *curr_next* pointing to *curr*. We can take care of that and proceed with moving *curr* and *curr_next* forward and repeat the process. Eventually we will have reversed all nodes in the list. This process ends whenever we reach the last node of the list; which also happen to be the new head of the list.

An implementation of such idea is shown in Listing 25.4. Note that while making *curr_next* point to *curr* we must also remember the element *curr_next* points to, otherwise it would be impossible to move *curr* and *curr_next* forward. Figure 25.2 shows the execution of the algorithm in Listing 25.4 on a list of 7 elements.

```

1  template <typename T>
2  Node<T>* list_reverse_constant_space_iterative(Node<T>* L)
3  {
4      if (!L || !(L->next))
5          return L;
6
7      auto curr      = L;
8      auto curr_next = curr->next;
9      curr->next     = nullptr; // the first node is the new tail
10     while (curr && curr_next)
11     {

```

^①Which is when either the current node is null or the current node does not have any node next to it.

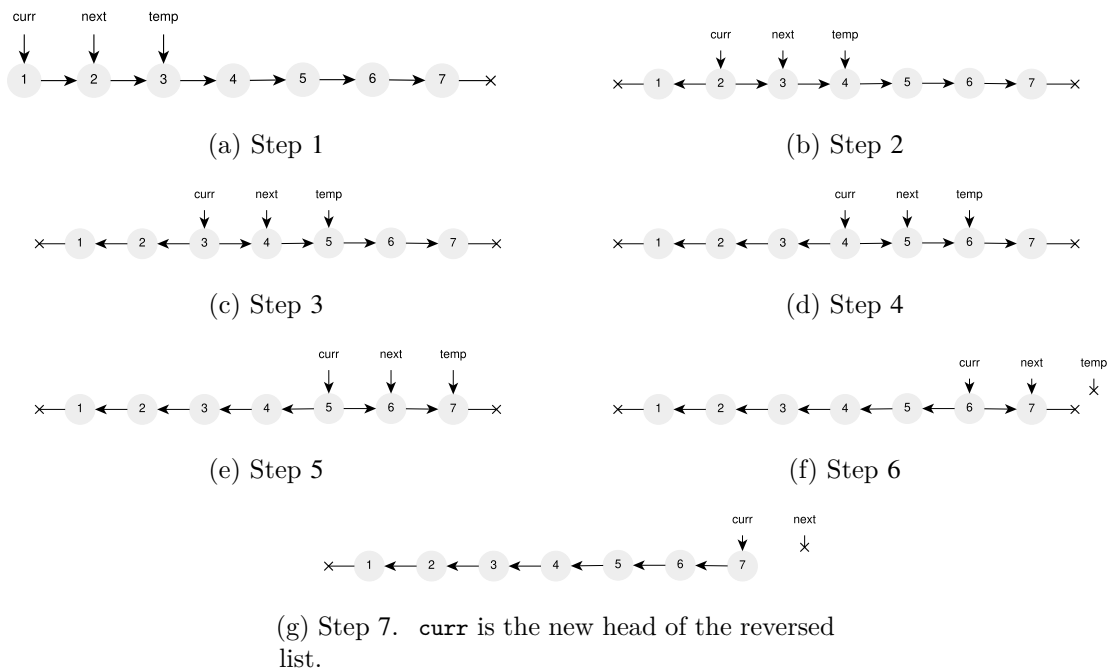


Figure 25.2: Execution of the algorithm implemented in Listing 25.4 on the list $L = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$

```

12     auto temp      = curr_next->next; // needed to move forward curr_next
13     curr_next->next = curr;
14     curr           = curr_next;
15     curr_next      = temp;
16 }
17 return curr;
18 }

```

Listing 25.4: Iterative constant space solution to the problem of reversing a list.

25.4 Conclusion

We have discussed three possible approaches to the problem of reversing a singly-linked list. We saw that it is almost trivial when using an iterative approach together with a stack to store the addresses of the list's nodes. This approach is based on the fact that the ordering of n elements popped from a stack is the reverse of the ordering the elements have been pushed on to.

We then examined an alternative solution that, whilst based on the same stack idea, does not use an explicit stack to store the nodes but rather stores the same information in the call stack of the recursive function.

Finally, we discussed a solution where only a constant space is required. This approach works iteratively from the head to the tail of the list by reversing two nodes at a time.

25.5 Common variation - Reverse a sublist

25.5.1 Problem statement

Problem 36 Create a function that, given a singly linked list L and two integers $n \geq m \geq 0$, reverses only its nodes from the m^{th} to the n^{th} and return the pointer to the new head of L . As in the other version of this problem discussed above the definition of a node is the same and the list L is given as a pointer to the first node of the list itself.

■ **Example 25.2**

Given the $L = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$, $m = 3$, $n = 5$ the function modifies it into $L = 1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3$ and returns a pointer to the node 1. ■

■ **Example 25.3**

Given the $L = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$, $m = 1$, $n = 2$ the function modifies it into $L = 1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3$ and returns a pointer to the node 1. ■

26. Min stack

Introduction

This chapter introduces a very popular question among companies like Yahoo, Amazon, Adobe and Microsoft. The question is simple and concerns designing a data structure for performing stack operations that is also able to keep track of the minimum element that is currently present in the stack. There is a simple, short and elegant solution for this problem, however, it is important understand the approach thoroughly as it is likely you may be asked a similar problem during the on-line screening steps of the interview process or during the first on-site.

26.1 Problem statement

Problem 37 Design a stack that supports:

- `push(x)`: the element x is pushed onto the stack
- `pop()`: removes the top of the stack
- `top()`: retrieve the top of the stack
- `get_min()`: retrieve the minimum among all elements present in the stack.

■ Example 26.1

Suppose the following set of operation on the stack are performed on a newly constructed and empty stack S :

- `push(1)`: $S = [1]$
- `push(5)`: $S = [5, 1]$
- `push(3)`: $S = [3, 5, 1]$
- `top()`: $S = [3, 5, 1]$, returns 3
- `pop()`: $S = [5, 1]$
- `get_min()`: $S = [5, 1]$, return 1
- `push(0)`: $S = [0, 5, 1]$
- `get_min()`: $S = [0, 5, 1]$, returns 0

■

■ Example 26.2

Suppose the following set of operations on the stack are performed on a newly constructed and empty stack S :

- `push(3)`: $S = [3]$
- `push(5)`: $S = [5, 3]$
- `push(1)`: $S = [1, 5, 3]$
- `get_min`: $S = [1, 5, 3]$, return 1
- `pop()`: $S = [5, 3]$, returns 3
- `get_min()`: $S = [5, 3]$, return 3
- `pop()`: $S = [1]$, return 1
- `pop()`: $S = []$

```
- pop(): raise std::logic_error
```

26.2 Clarification Questions

Q.1. What should be done when `get_min()` or `top()` or `pop()` are performed on an empty stack?

You should throw a `std::logic_error` exception with a sensible and short description.

26.3 Discussion

This problem can become quite tricky if approached from the wrong angle. We will discuss two solutions both of which are good options to use during an actual interview.

26.3.1 Linear Space solutions

26.3.1.1 Stack of pairs

This first solution uses an additional space ($2\times$) to store for each element of the stack the information about the minimum among the elements still present in the stack. In order to do so we use a stack of **pairs**, where the first item of each pair is the actual element we want to push into our data structure and the second is the minimum value among the elements we have seen so far. Given this set-up the operations can then be implemented as follows:

- `push(x)`. We will store on top of the stack of pair either the pair $\{x, x\}$ if the stack is empty or $\{\text{std::min}(x, \text{get_min}())\}$.
- `top(x)` returns the **first** element of the top of the stack of pair if the stack is not empty, otherwise it throws an exception.
- `pop(x)` will call `pop` on the stack of pairs if the stack is not empty, otherwise raises an exception.
- `get_min(x)` returns the **second** element of the top of the stack of pair if the stack is not empty, otherwise it throws an exception.

Listing 26.1 shows a possible implementation of this idea.

```
1  template <class T>
2  class min_stack_stack_pair
3  {
4  public:
5      void push(const T& x)
6      {
7          const auto nm = q.size() > 0 ? std::min(x, getMin()) : x;
8          q.push({x, nm});
9      }
10
11     void pop()
12     {
13         guard_empty_stack();
14         q.pop();
15     }
16
17     T top()
18     {
19         guard_empty_stack();
20         return q.top().first;
```

```

21     }
22
23     T getMin()
24     {
25         guard_empty_stack();
26         return q.top().second;
27     }
28
29 protected:
30     void guard_empty_stack()
31     {
32         if (q.size() <= 0)
33             throw std::logic_error("Invalid operation on an empty stack");
34     }
35
36 private:
37     std::stack<std::pair<T, T>> q;
38 };

```

Listing 26.1: Solution to the problem of designing a min stack using a stack of pairs.

26.3.1.2 Two stacks

The solution presented in Section 26.3.1.1 can be improved upon (even though will remain asymptotically the same in the worst case) by realizing that there is no need to have a copy of the minimum element for each element of the stack. What we really need is to have a second stack that contains the sequence of minimum elements as they are inserted.

We can achieve that by using an additional stack to store the minimums. Every time we try to push an element that is lower than the top of this stack, we will push it into. Given this additional stack we can implement all the operations as follows:

- `push(x)`. We will store x on top of the 1st stack (where we store the actual elements are they are given to us) and only if $x \leq \text{get_min}()$ we will also push x to the stack of minimums. This way we keep the information about the current minimum without losing the information about the previous ones which will be useful whenever in the future it will be removed from the stack (if x is the new minimum).
- `top(x)` returns the **first** element of the top of the 1st stack if it is not empty, otherwise it throws an exception.
- `pop(x)` if the stack is not empty, it will call `pop` on the 1st stack, and if the element we are popping is equal to the top of the stack of minimums we will also pop from that stack. We need to react to the fact that the current minimum is changing.
- `get_min(x)` returns the top of the 2nd stack (the stack of minimums) if the stack is not empty, otherwise it throws an exception.

This solution has the advantage of not using as much space as the one presented in Section 26.3.1.1 when e.g. a sequence of increasing number is pushed. In that case the minimum will be the first element, and it will never change. Also note that because of the way the stack of minimums is used, it will always contains a decreasing sequence of values (after all we only push to it if the new element is smaller than the top).

This idea can be implemented as shown in Listing 26.2

```

1  template <class T>
2  class min_stack_two_stacks
3  {
4  public:
5      void push(const T& x)
6      {
7          if (x <= getMin() || minimums.size() == 0)

```

```

8     minimums.push(x);
9     elements.push(x);
10 }
11
12 void pop()
13 {
14     guard_empty_stack();
15     if (getMin() == elements.top())
16         minimums.pop();
17     elements.pop();
18 }
19
20 T top()
21 {
22     guard_empty_stack();
23     return elements.top();
24 }
25
26 T getMin()
27 {
28     guard_empty_stack();
29     return minimums.top();
30 }
31
32 protected:
33 void guard_empty_stack()
34 {
35     if (elements.size() <= 0)
36         throw std::logic_error("Invalid operation on an empty stack");
37 }
38
39 private:
40     std::stack<int> elements;
41     std::stack<int> minimums;
42 };

```

Listing 26.2: Solution to the problem of designing a min stack using a two stacks.

26.3.2 Constant space

Despite the fact that the solution presented in Section 26.3.1.2 is likely already sufficient for a coding interview, it is worth considering an additional solution that works only for integral types and that works in constant space. This is a big improvement relative to the other solutions above, however, the downside is that it only works for a very limited number of types.

If, as already discussed, the key challenge of this problem is how to retrieve the current minimum for each element in the stack and store such information without using additional then the answer may be that we need to encode such information into the elements themselves.

The idea is simple: we will store the elements in a `std::stack`, S , and we will also keep track of the **current** minimum element in a variable, `min_el`. The operations on the data structure can be implemented as follows:

- `push(x)` we have two cases here:
 1. if the stack is empty: push x to S and set `min_el = x`
 2. otherwise:
 - if $x \geq \text{min_el}$ just push x to S leaving `min_el` untouched.
 - if $\text{min_el} > x$, push $2*x - \text{min_el}$ to S and set `min_el = x`.

- `pop()` two cases here as well depending on the element to be removed `y` (at the top of the stack):

1. if $y \geq \text{min_el}$, `y` is removed from the stack leaving `min_el` untouched
2. otherwise: if $y < \text{min_el}$, set $\text{min_el} = 2 * \text{min_el} - y$

The key idea here is that we can retrieve the previous minimum element given the current one and the value that is currently on the stack.

- `top()` very similar to the `pop` operation, without the update on the variable `min_el`
- `get_min(x)`, returns `min_el`

When an element `x` is less than the current minimum i.e. $x < \text{min_el}$, the value $2 * x - \text{min_el}$ will be inserted in the stack and the minimum set to `x`. The fact that $2 * x - \text{min_el} < x$ (remember that $\text{min_el} > x$) is important given that when this element will be popped out from the stack we will be able to tell that the minimum is changing because we will see that $2 * x - \text{min_el} < x$ and therefore we will update the minimum element accordingly.

The idea above is shown in Listing 26.3. Note how the template class will only compile for integral types thanks to `std::enable_if`[11].

```

1  template <class T,
2      typename = typename std::enable_if<std::is_integral<T>::value>::type>
3  class min_stack_int_constant_time
4  {
5  public:
6      void push(const T& x)
7      {
8          T new_min_el = min_el;
9          if (elements.size() == 0)
10         {
11             elements.push(x);
12             new_min_el = x;
13         }
14
15         if (x < elements.top())
16         {
17             elements.push(2 * x - min_el);
18             new_min_el = x;
19         }
20         else
21         {
22             elements.push(x);
23         }
24         min_el = new_min_el;
25     }
26
27     void pop()
28     {
29         guard_empty_stack();
30         const T top_el = elements.top();
31         elements.pop();
32
33         if (top_el < min_el)
34         {
35             min_el = 2 * min_el - top_el;
36         }
37     }
38
39     T top()
40     {
41         guard_empty_stack();
42         const T top_el = elements.top();

```



```

43     if (top_el >= min_el)
44         return top_el;
45     else
46         return min_el;
47 }
48
49 T getMin()
50 {
51     guard_empty_stack();
52     return min_el;
53 }
54
55 protected:
56 void guard_empty_stack()
57 {
58     if (elements.size() <= 0)
59         throw std::logic_error("Invalid operation on an empty stack");
60 }
61
62 private:
63     std::stack<T> elements;
64     T min_el;
65 };

```

Listing 26.3: Solution to the problem of designing a min stack of integer working in constant space and linear time.

26.3.3 Common Variations

Problem 38 Design and implement a max stack data structure supporting the following operations:

- `push(x)`: the element x is pushed onto the stack
- `pop()`: removes the top of the stack
- `top()`: retrieve the top of the stack
- `get_max()`: retrieve the maximum among all elements present in the stack.

27. Find the majority element

Introduction

It is election time and we are hired to make sure the vote counting is free of mistakes and quick. Our job is to determine the winner of this year election. Votes are collected as an unordered list and our task is to determine whether there is a clear winner i.e. someone with the majority of the votes (i.e. with more than 50% of them) or a new voting session is necessary.

This problem has been asked at Google and Yahoo interviews for software engineering positions and it is considered medium difficulty. Infact, despite the fact it is almost trivial to solve it in linear space, doing so in constant space proves to be quite a bit more challenging and requires non-trivial insights.

As we will see in the coming sections, this problem (and its solution) is a specialization of a more general problem where we need to find out if there is an element in the input list that appears more than $\frac{n}{k}$ times. Clearly, under this definition, we have the majority element problem when $k = 2$.

27.1 Problem statement

Problem 39 Given an array N of size n , find the majority element i.e. that element that appears more than $\lfloor \frac{n}{2} \rfloor$ times. If such element does not exist, return -1 .

■ **Example 27.1**

Given the array $[1, 2, 3, 2, 2, 1, 1, 1]$, the function returns 1 because it appears 4 times in an array of length 8.

■ **Example 27.2**

Given the input array $[2, 1, 2]$ the function return 2 because it is greater than $\frac{3}{2}$.

■ **Example 27.3**

Given the input array $[2, 1, 2, 3, 4, 5]$ the function return -1 no element appear more than 3 times.

27.2 Clarification Questions

Q.1. What are the minimum and maximum values an element of the array can take?

The minimum and maximum values are $[-10^9, 10^9]$, respectively. This is a good question to ask because if the range is small then we can apply a solution based on bucket counting.

Q.2. Can the input array N be modified or shuffled?

Yes, the input array can be modified.

27.3 Discussion

We will examine three different solutions for this problem. We begin by looking at the brute-force approach in section 27.3.1. Section 27.3.3 will then describe an approach that uses sorting to improve the time complexity of the brute-force approach. Finally, in section 27.3.5 we will examine the optimal approach using the Boyer-Moore algorithm.

27.3.1 Brute-force

The brute force solution is very simple and consist of looping through the array and for each element counting how many times it occurs in the input array. Although this approach is simple, it will not serve you well in an interview scenario as it is far from the optimum and the interviewer is certainly expecting a more sophisticated solution. Listing 27.1 shows a possible implementation of this approach.

```
1 int find_majority_element_brute_force(const std::vector<int>& N)
2 {
3     const size_t threshold = N.size() / 2;
4     for (const auto x : N)
5     {
6         const size_t countx = std::count(begin(N), end(N), x);
7         if (countx > threshold)
8             return x;
9     }
10    return -1;
11 }
```

Listing 27.1: Sample Caption

27.3.2 Hash-map approach

A simple improvement on the solution in the section 27.3.1 can be made by using a hash-map to store the number of occurrence of each element in the input array. There cannot be more than n different numbers in the the array N , thus with a single pass of the input and with a linear cost in space we can calculate the number of occurrence of each element and check if any of the counters at any points gets higher than $\lfloor \frac{n}{2} \rfloor$.

A possible implementation of this approach is shown in Listing 27.2. The complexity of this approach is $O(n)$ for both space and time. In-fact, even in the worst case all the elements of the input array are only read and stored once.

```
1 int find_majority_element_hash_map(const std::vector<int>& N)
2 {
3     const size_t threshold = N.size() / 2;
4
5     std::unordered_map<int, int> C;
6     std::pair<int, int> max_val = {0, 0};
7     for (const auto x : N)
8     {
9         int& countx = C[x];
10        countx++;
11        if (countx > threshold)
12            return x;
13    }
14    return -1;
15 }
```

```
15 }
```

Listing 27.2: Solution to the problem of finding the majority element in an array using hash-map.

27.3.3 Sorting - Counting

The approach described in section 27.3.2 is definitely faster than the quadratic brute-force but at a linear price in space. In order not to pay the price in space and to lower the time complexity down from quadratic, we could rely on the fact that in a sorted collection of elements all equal elements appear grouped together e.g. in $[1, 1, 2, 2, 3, 3, 3, 4, 4, 9, 9]$, all the 1s appear at the beginning of the array, followed by all the 2s, etc. We can then count the number of occurrences of each element in constant space as shown in Listing 27.3. The complexity of this approach is bounded by the sorting which costs $O(n \log(n))$ time.

```
1 int find_majority_element_sorting(std::vector<int>& N)
2 {
3     const size_t threshold = N.size() / 2;
4     if (N.size() == 0)
5         return -1;
6
7     std::sort(std::begin(N), std::end(N));
8
9     // current.first = number
10    // number.second = occurrences
11    std::pair<int, int> current;
12    for (size_t i = 0; i < N.size(); i++)
13    {
14        if (N[i] != current.first || i == 0)
15            current = {N[i], 1};
16        else
17            current.second++;
18
19        if (current.second > threshold)
20            return current.first;
21    }
22    return -1;
23 }
```

Listing 27.3: Solution to the problem of finding the majority element in an array using sorting.

27.3.4 Sorting - Median

We can, however, make even better use of the fact that we have a sorted collection. The key idea here is that if a majority element exists then this element **must be the median**. After all, by definition, the median element is right in the middle of the sorted collection. Since the majority element will occupy **more** than half of the positions of the array it must also occupy the median position. All that is necessary after sorting the array is to check if the median value appears more than $\lfloor \frac{n}{2} \rfloor$ times. This idea is implemented in Listing 27.4 and has a complexity of $O(n \log(n))$ due to sorting.

```
1 int find_majority_element_median(std::vector<int>& N)
2 {
3     if (N.size() == 0)
4         return -1;
5     std::sort(std::begin(N), std::end(N));
6     const int midpoint = N.size() / 2;
```

```

7   const int el      = N[midpoint];
8
9   const size_t threshold = N.size() / 2;
10  if (std::count(begin(N), end(N), el) > threshold)
11      return el;
12  return -1;
13 }

```

Listing 27.4: Linear time constant space solution to the problem of finding the majority element in an array.

27.3.5 Boyer-Moore algorithm

The best approach to solving this problem in linear time and constant space is, however, to use the Boyer-Moore algorithm[2].

The algorithm uses two variables to maintain a candidate element *el* of the sequence and its current count *count* = 0 (initialized to 0). It processes the elements one by one and will perform the following operations:

- if we are processing the very first element of the sequence or *count*=0, it will set *count* = 1 and *el* to that element (this is our first candidate).
- otherwise, if the element currently processed is equal to *el* it increments the counter i.e. *count* = *count*+1
- if the element currently processed is different, then it decrements the counter i.e. *count* = *count* -1;

At the end of this process the variable *el* will contain a candidate majority element. If the array contains a majority element then *el* is the one. The algorithm correctness can be derived from the fact that the counter will be incremented more times than it will be decremented for the majority element. If we cannot assume that a majority element **always** exists then a second pass on the array is necessary in order to count the number of occurrences of *el* in the input array.

Listing 27.5 shows a possible implementation of the Boyer-Moore algorithm. The complexity of this approach is $O(n)$ time and $O(1)$ space because the input array is scanned twice and only two additional integer variables are used.

```

1  int find_majority_element_linear(const std::vector<int>& nums)
2  {
3      if (nums.size() <= 0)
4          return -1;
5
6      int el      = nums.front();
7      int count = 0;
8      for (size_t i = 0; i < nums.size(); i++)
9      {
10         if (nums[i] == el)
11         {
12             count++;
13         }
14         else
15         {
16             count--;
17         }
18         if (count == 0)
19         {
20             el      = nums[i];
21             count = 1;
22         }
23     }

```

```

24 // check that el appears > n/2 times
25 if (std::count(begin(nums), end(nums), el) > nums.size() / 2)
26     return el;
27 return -1;
28 }

```

Listing 27.5: Linear time constant space solution to the problem of finding the majority element in an array.

27.4 Find the element repeated $\frac{n}{k}$ times.

Problem 40 Write a function that, given an array A of n integers and an integer k , returns any of its element that occurs more than $\frac{n}{k}$. If such an element does not exist the function returns -1 .

■ Example 27.4

Given $A = \{1, 2, 1, 3, 1\}$ and $k = 3$, the function return 1 as it occurs $3 > \left\lfloor \frac{|A|}{3} \right\rfloor = 2$ times.

■

27.4.1 Boyer-Moore algorithm extended

Solution approach:

if you have three distinct elements in the array the solution does not change. You can ignore all three of them.

Therefore, just keep track of the frequencies of two elements. When you process a new element you can do the following:

1. if you do not have three elements in the frequency list. add it with frequency one
2. if the element is equal to another element in the list. increase its frequency
3. if it is a new element (not in the list), decrease the frequency of all in the list by one. Remove any element with frequency zero.

Listing 27.6: Sample Caption

28. n^{th} node from the end

Introduction

The problem presented in this chapter is a particularly interesting one on linked lists. It is often asked during interviews for major tech companies like Amazon and Google and it is therefore worth taking time to ensure we understand and master the solution to this problem.

28.1 Problem statement

Problem 41 Given a linked list L (which definition is shown in Listing 21.1 at page 102) and an integer n remove the n^{th} node from the end of list.

■ **Example 28.1**

Given $L = [1, 2, 3, 4]$, and $n = 2$ the function returns: $L = [1, 2, 4]$. See Figure ??.

■ **Example 28.2** Given $L = [1, 2, 3, 4]$, and $n = 0$ the function returns: $L = [1, 2, 4]$. See Figure ??.

28.2 Clarification Questions

Q.1. Is n guaranteed to be a valid node in the list?

Yes you can assume that n is the index of a valid node in the list.

28.3 Discussion

This problem can be broken down into two parts:

1. Finding out the index of the n -to last node
2. Removing a node from the list

These tasks are separate and thus we can solve each of them separately and then use the solution to these two sub-problems to obtain our final answer.

28.3.1 Brute-force

Finding out the the node to be deleted is easy once we know how long the list is. The brute-force approach simply performs a first pass in the list and counts how many nodes

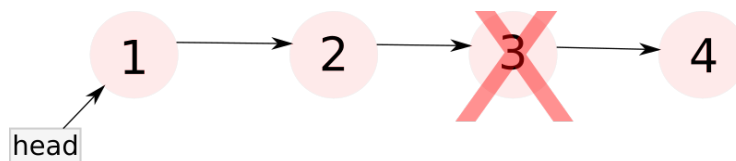


Figure 28.1: Removal of the 2nd to last element in a singly linked list of length 4.

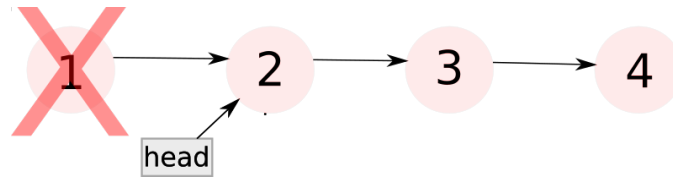


Figure 28.2: Removal of the 4th to last element in a singly linked list of length 4. The head pointer needs to be updated.

it is made of i.e. l . Then it performs another pass but it stops at node $l - n$ (the n -to-last node) and removes it. Please note that in order to correctly remove a node from a singly linked list we need to have a pointer to the node we want to remove as well as a pointer to its predecessor (variable `pred` in the code). This approach can be implemented as shown in Listing 28.1, and it has a time and space complexity of $O(n)$ and $O(1)$, respectively.

```

1  int list_length(ListNode* head)
2  {
3      int ans = 0;
4      while (head)
5      {
6          ans++;
7          head = head->next;
8      }
9      return ans;
10 }
11
12 ListNode* remove_nth_node_from_end_bruteforce(ListNode* head, int n)
13 {
14     const int length = list_length(head);
15     // we can assume it is always valid/positive
16
17     ListNode *prec = nullptr, *curr = head;
18
19     int index = length - n - 1;
20     while (index--)
21     {
22         prec = curr;
23         curr = curr->next;
24     }
25
26     ListNode* next = curr->next;
27     ListNode* ans = head;
28     if (!prec)
29         ans = next; // we are removing the first node
30     else
31         prec->next = next;
32
33     return ans;
34 }

```

Listing 28.1: Sample Caption

28.3.2 Two pointers

There is however another way of solving this problem that is slightly better even if not in terms of asymptotic complexity. The key issue we have with this problem is that we have to delete a node at index $l - n$ but we have no idea what l is and we do not want to compute it explicitly. What we can do is to loop forward with a pointer s from the head

of the list for n nodes. At this point s will be at n node distance from the head and at $l - n$ from the tail. We now have a way of counting $n - l$. Let f be a pointer to the head of the list: we can advance both f and s until s reaches the end of the list. At that point s had advanced $l - n$ times and f crucially will be pointing at the node $l - n$ i.e. at the n -to-last node.

This idea is implemented in Listing 28.2. Please note that the second `while`, as in the brute-force approach (Listing 28.1) also needs to keep a pointer to the node preceding the one that needs to be deleted (pointer p in the code).

The complexity of this implementation is linear in time and constant in space.

```

1  ListNode *remove_nth_node_from_end_two_pointers(ListNode *head, int n)
2  {
3      if (n == 0)
4          return head;
5      ListNode *s, *f, *p = nullptr;
6      s = f = head;
7
8      // advance s n times
9      while (n)
10     {
11         s = s->next;
12         n--;
13     }
14
15     // now s is at a distance of l-n from the tail
16     while (s)
17     {
18         ListNode *oldf = f;
19         f = f->next;
20         s = s->next;
21         p = oldf;
22     }
23     // f points to the node l-n of the list
24
25     ListNode *next = f ? f->next : nullptr;
26     if (!p)
27     {
28         head = next;
29     }
30     else
31     {
32         p->next = next;
33     }
34
35     return head;
36 }

```

Listing 28.2: Sample Caption

28.3.3 Common Variation

28.3.3.1 List midpoint

Problem 42 Given a linked list L of length l (which definition is shown in Listing 21.1 at page 102), return the value of the node at position $\frac{l}{2}$.

■ Example 28.3

Given $L = [1, 2, 3, 4]$, the function returns 2. ■

■ **Example 28.4** Given $L = [1, 5, 7, 8, 9, 4, 5, 6, 1, 2, 4, 9, 7]$, the function returns 5. ■

This is a very popular variation of the problem described in this chapter. It can be solved using the same methods described in Sections 28.3.1 and 28.3.2 or using an ad-hoc solution (hint: a fast and a slow pointers)

29. Validate Parenthesized String

Introduction

Analyzing strings is an important operation for computer languages and it lies at the heart of programming languages. For example a calculator would look at an input such as `33 * 4 + (125 - 22*sqrt(2))` and before proceeding in performing the calculation would check that the input string forms an allowable expression.

In the problem discussed in this chapter we will study how we can write an efficient parser for a simple grammar on an alphabet consisting only three characters describing a particular kind of well parenthesized strings. What is cool about this problem is that the techniques as well as the structure of the solutions presented here can be adapted and exploited for other string analysis problems.

29.1 Problem statement

Problem 43 Given a string s containing only three types of characters:

1. (
2.)
3. *

write a function to check whether a string is valid. A string is valid if the following holds:

- Any left parenthesis (must have a corresponding right parenthesis).
- Any right parenthesis) must have a corresponding left parenthesis (.
- Left parenthesis (must appear before the corresponding right parenthesis).
- The character * could be treated as a jolly, and can be modified into a single right parenthesis) or a single left parenthesis (or deleted.

■ Example 29.1

Given the input string $s = "(**)"$ the function returns **true** because it is possible to obtain from s the string $(())$ by deleting the first * and by turning the second one into a left parenthesis (.

■ Example 29.2

Given the input string $s = "**(*)()()"$ the function returns **false** because no matter how the * are arranged there is no way to obtain a well balanced string of parenthesis.

29.2 Clarification Questions

Q.1. Is an empty string considered valid?

An empty string is also valid.

29.3 Discussion

This is an extremely interesting and challenging problem that can be solved in several ways. We start with the brute-force solution in Section 29.3.1 from which we can develop a more effective dynamic programming solution. Section 29.3.3 examines a different solution based on a greedy technique that can dramatically improve the time and space complexity compared to the previous approaches. Section ?? presents a linear time and space clever solution based on stacks.

The solution shown in Section 29.3.3 is likely to most effective and therefore we advise to us this as the reference point during an actual interview.

29.3.1 Brute-force

If the input string does not contains wild-cards, this problem is quite simple and is easily solvable using a stack. When wild-cards are present things are complicated because now for each of them there are three options. In the brute-force approach we will try all possible options for all wild-cards. The idea is that the input string s is traversed from left to right. As we traverse the string we will keep track of how many open `open` and closed `closed` parenthesis we have encountered. We do this because if at any moment we find that the number of closed parenthesis is greater than the number of open ones, the string is invalid (it violates the constraint that any left parenthesis should appear before any right one). Depending on the character c we are processing:

1. If c is a `(` then we increase the number of open parenthesis `open++` found so far and we recursively check the rest of the string.
2. Similarly, if c is a `)` then we increase the number of closed parenthesis and proceed checking the rest of the string.
3. If the current character is a `*` then we have the option to:
 - consider it as an open parenthesis
 - consider it as a closed parenthesis
 - ignore it

The recursion terminates when either:

- the number of closed parenthesis is larger than the number of open ones
- we have processed the whole string. In this case we return `true` only if the number of open parenthesis so far is equal to the closed ones (a necessary condition for a well balanced string).

Listing 29.1 shows a possible recursive implementation of the idea above. The complexity of this approach is exponential in the number of `*`, i.e. $O(3^n)$, where n is the length of s .

```
1 bool validate_parenthesized_string_bruteforce_helper(std::string s,
2                                                       const size_t pos,
3                                                       const int open,
4                                                       const int closed)
5 {
6     if (pos == s.size())
7         return open == closed;
8
9     if (closed > open)
10        return false;
11
12     const char curr = s[pos];
13     bool ans = false;
14     if (curr != '{') // either } or *: add a right parenthesis
15         ans = validate_parenthesized_string_bruteforce_helper(
16             s, pos + 1, open, closed + 1);
```

```

17
18     if (curr != '}' && !ans) // either {} or *: add a left parenthesis
19         ans = validate_parenthesized_string_bruteforce_helper(
20             s, pos + 1, open + 1, closed);
21
22     if (curr == '*' && !ans) // if neither { nor } worked, then ignore this *
23         ans = validate_parenthesized_string_bruteforce_helper(
24             s, pos + 1, open, closed);
25
26     return ans;
27 }
28
29 bool validate_parenthesized_string_bruteforce(std::string s)
30 {
31     return validate_parenthesized_string_bruteforce_helper(s, 0, 0, 0);
32 }

```

Listing 29.1: Brute-force, exponential time solution to the problem of validating a string of parenthesis with wild-cards.

29.3.2 Dynamic Programming

Another way of solving this problem is to still try all possibilities as in the brute-force solution, but to streamline that process by ensuring no work is done more than once. In a string of length n there are $O(n^2)$ possible substrings. Given a substring starting at i and ending at j , from now on identified by $s(i, j)$ we can solve this problem by processing one character c at the time. A substring $s(i, j)$ is valid when

- if c is an $*$ and $s(i+1, j)$ is valid. We try to ignore the character c .
- if c is either $*$ or $($, then we search for a character k in $s(i+1, k)$ s.t. it can be turned into a closing parenthesis. If k exists (in case multiple k exists, then we try all of them) then $s(i, j)$ is valid if $s(i+1, k-1)$ and $s(k+1, j)$ are valid. What we are doing here is matching an open parenthesis with a closing one that appears further in the range. Remember that each open parenthesis must be paired up with a closing one.
- if c is $)$ we return false because in this case we have an unmatched closing parenthesis.

The result for a substring $s(i, j)$ is saved in a map, and when the algorithm asks for the validation of the same substring again, the value returned in the map is returned instead of doing the computation again. This technique is called memoization, and allows us to avoid the repeated re-computation of the same subproblem. The approach described in this section can be implemented as shown in Listing 29.2 and it has a time complexity of $O(n^3)$. There are $O(n^2)$ possible substrings and for each of them a work proportional to n is performed. The space complexity is bound by the amount of substrings that we can potentially store in the Hash-set i.e. $O(n^2)$.

Note that this solution is, at it's core, just another way of solving this problem by brute-force. We just use Dynamic Programming techniques to speed it up by remembering the result of intermediate subproblems (in this case the substring of the input string s) thereby avoiding repetitive work. As such, although the DP solution is definitely better than simple brute-force, it is still far from optimal. In the next section we will investigate a much faster solution that has the additional benefit of being also much shorter in length and therefore less error prone than the ones presented so far.

```

1 // a pair of indices identifying a substring
2 using pii = std::pair<int, int>;
3
4 struct pair_hash
5 {

```

```

6  template <class T1, class T2>
7  std::size_t operator()(const std::pair<T1, T2>& pair) const
8  {
9      return std::hash<T1>()(pair.first) ^ std::hash<T2>()(pair.second);
10 }
11 };
12
13 bool validate_parenthesized_string_DP_helper(
14     const std::string& s,
15     std::unordered_map<pii, bool, pair_hash>& DP,
16     const pii& substr)
17 {
18     const auto [i, j] = substr;
19     if (i > j)
20     {
21         // empty string is valid
22         return true;
23     }
24
25     if (DP.find(substr) != DP.end())
26         return DP[substr];
27
28     bool ans = false;
29     const char c = s[i];
30
31     if (c == ')')
32         ans = false;
33
34     if (!ans && c == '*') // try ignoring this character
35         ans = validate_parenthesized_string_DP_helper(s, DP, {i + 1, j});
36
37     if (!ans && c != ')') // either * or open brackets. Try turning it into a (
38     {
39         // find a something that can be turned into a ) further ahead in the
40         // string
41         for (int k = i + 1; !ans && k <= j; k++)
42         {
43             if (s[k] == ')') || s[k] == '*')
44             {
45                 // validate the two resulting substring from pairing char i and k
46                 ans = validate_parenthesized_string_DP_helper(s, DP, {i + 1, k - 1})
47                     && (validate_parenthesized_string_DP_helper(s, DP, {k + 1, j}));
48             }
49         }
50     }
51
52     DP[substr] = ans;
53     return ans;
54 }
55
56 bool validate_parenthesized_string_DP(const std::string& s)
57 {
58     std::unordered_map<pii, bool, pair_hash> DP;
59     const int size = s.size() - 1;
60     return validate_parenthesized_string_DP_helper(s, DP, {0, size});
61 }

```

Listing 29.2: Dynamic programming solution to the problem of validating a string of parenthesis with wild-cards.

Also note that the structure `pair_hash` is necessary so that the `std::unordered_map` knows

how to correctly calculate a hash value for `std::pairs`.

29.3.3 Greedy - Linear time

The two previous approaches do much more than the problem asks for. They not only determine whether the input string is valid or not; they also calculate one. They solve the problem by manufacturing a valid string by trying out all possibilities until either one or none of them is good. A more efficient approach would determine whether the input string can be turned into a good one without having to actually come up with a specific valid string obtainable from the input. This is the rationale behind the solution presented in this section.

The main idea is that for a valid string it has to be true that the balance between open o and closed c parenthesis is perfect i.e. $o - c = 0$ has to hold. $o - c = 0$ represents the number of open unmatched open parenthesis. Because the `*` can be either a `(` or a `)` we cannot simply loop through the string and count the number of open and closed parenthesis. What we can do is to store all possible values for $o - c$ parenthesis that can be obtained. As we will see, these values change in a fairly predictable way depending on what character we process. For instance given the input string `s="(*)"` when processing the i^{th} character, the number of $c - o$ possible values P can be:

1. $P = \{1\}$
2. $P = \{0, 1, 2\}$ because considering only the first two character of `s` `(*` we can obtain
 - `((` by turning the `*` into a `(`.
 - `(` by deleting the `*`.
 - `()` by turning the `*` into a `)`.
3. $P = \{1, 2, 3\}$ because considering only the first three character of `s` `(*(` we can obtain
 - `((((` by turning the `*` into a `(`
 - `((` by deleting the `*`.
 - `((()` by turning the `*` into a `(`
4. $P = \{0, 1, 2\}$ because considering only the first four character of `s` `(*(()` we can obtain
 - `((())` by turning the `*` into a `)`
 - `((()` by deleting the `*`
 - `((()` by turning the `*` into a `(`.
5. $P = \{-1, 0, 1, 2, 3\}$ because considering only the first four character of `s` `(*(())` we can obtain
 - `((())` by turning the first `*` into a `)` and deleting the second one.
 - `((())` by deleting both the `*`
 - `((())` by turning both the `*` into a `(`.
 - finally we can obtain -1 with this string `((())` by turning both the `*` into a `)`.

Note that the values in the list P can be obtained with different combinations of substitutions and that P is always made of a contiguous element. This last piece of information is important because it allows us to describe P by only using its maximum and minimum value. We are interested in seeing whether at the end of the process we can obtain a

value of 0 meaning that the string is balanced. If the maximum value at any point goes under 0 it means that we reached a place where we have an excess of closed parenthesis that we cannot fix using all the asterisks we encountered so far. For instance consider the string `s="(())"`. When processing the element number the max and min values for the difference between the open and closed parenthesis will be:

1. (1,1)
2. (0,2)
3. (-1,1)
4. (-2,0)
5. (-2,-1)

When we reach the 6th character the maximum number of open parenthesis we can obtain in the best case is -1, meaning that we are short by one for a balanced string. The string is, therefore, invalid because there is an excess of closed parenthesis. This is a violation of the rule stating that every closed parenthesis must have a proceeding open one.

We can use the idea described above to derive an algorithm that works as follows: Let `min`, `max` respectively be the smallest and largest possible number of open left brackets after processing the i^{th} character in the string `s`.

If we encounter:

- a left parenthesis `(`, then we can increment both `min` and `max`.
- similarly, a right parenthesis `)`, then we can decrement both `min` and `max`.
- an asterisk `*` we can choose to either consider this as an open parenthesis (increasing the max number of obtainable open ones), delete it (leaving the balance unvaried) or convert it into a closed parenthesis (reducing `min`)

If at any point the maximum number of open parenthesis falls under 0 then we are forced to consider the string invalid, for the reason we pointed out earlier (at least an unmatched closed parenthesis). Similarly we need to make sure that `min` does not go below zero, because we do not need to consider strings which have this count of open parenthesis as they would be invalid for the same reason as above.

When all the characters are processed, the only thing that is left to check is that 0 is contained in the range defined by `min` and `max`. If it is, then there is a way to turn the string into a valid one, otherwise it is impossible (imagine the case where `min > 0` meaning that no matter what we do, the minimum amount of open parenthesis we end up with is still more than one meaning that there is at least one unmatched open parenthesis).

Listing 29.3 shows a possible implementation of this idea. Note how every decrement of `min` is guarded by a check, so to avoid that it goes below 0.

```

1 bool validate_parenthesized_string_linear(std::string s)
2 {
3     int min, max;
4     min = max = 0;
5     for (const char c : s)
6     {
7         if (c == '(')
8         {
9             min++;
10            max++;
11        }
12        if (c == ')')
13        {
14            if (min > 0)
15                min--;
16            max--;

```



```
17     }
18     if (c == '*')
19     {
20         if (min > 0)
21             min--;
22         max++;
23     }
24     if (max < 0)
25         return false;
26 }
27 return min == 0;
28 }
```

Listing 29.3: Linear time constant space solution to the problem of validating a string of parenthesis with wild-cards.

30. Tree Diameter

Introduction

The problem described in this chapter is quite simple and can be solved elegantly in just a handful of lines of code. As such, it is really important we understand all the pieces that make up the solution so we will can present it quickly during an interivew.

30.1 Problem statement

Problem 44 Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. The length of path between two nodes is the number of edges you need to traverse to go from one to the other. The definition of the tree is shown in Listing 19.1.

■ **Example 30.1**

Given the binary tree shown in Figure 30.1 the function returns 7. One path of such length is from node 10 to node 7 or 8. ■

30.2 Discussion

30.2.1 Brute-force

In order to solve this problem we need to first tackle a different one i.e. finding the depth of a tree. We will then use the solution to this problem to compute the solution for the main one. Why is the depth of the binary tree important for determining the tree diameter? Let's start by saying that the height of a binary tree is the longest path from the root to a leaf. The tree diameter can be found by visiting the tree one node at the time and for each node n calculating the longest path between two leaves by a path passing n . For instance considering the Figure 30.1 the height of the subtree rooted at node 5 is 2 while the height for the node rooted at 6 is 1. Given the heights for node 5 and 6 we can calculate the length of the longest path between between two leaves passing through node 2: 3(height of node 5) + 2(height of node 6). So given a node n and the height of its left and right subtrees h_l and h_r , respectively, the length of the longest path between two leaves passing through n can be calculated as follows:

- calculate h_l , height of the left subtree of n
- calculate h_r , height of the right subtree of n
- $d = h_l + h_r$
- if the left subtree of n is not null, add 1 to d : $d = d + 1$ (we need to account for the arc going from n to the left subtree)
- similarly for the right subtree, if it is not null, add 1 to d : $d = d + 1$.

The height of a tree can be easily calculated using the recursive function `height` in Listing 30.1.

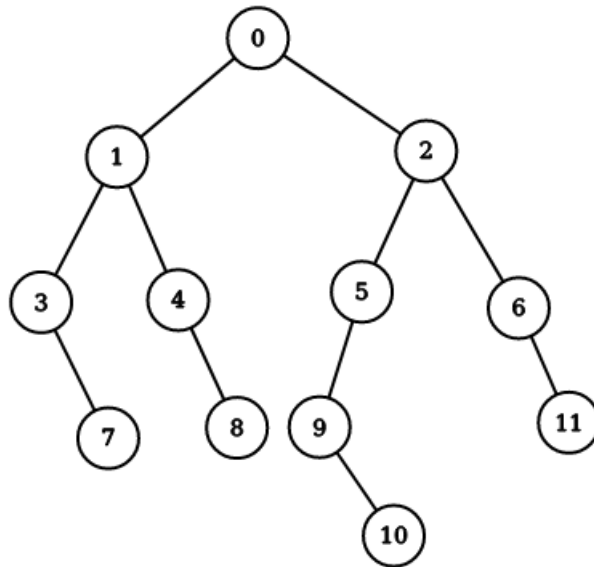


Figure 30.1: Visual representation of the example 1 of the problem calculating the tree of a diameter.

To summarize: the diameter of a tree T is the largest of the following quantities:

- the diameter of T 's left subtree
- the diameter of T 's right subtree
- the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Listing 30.1 shows a recursive implementation of this idea. The complexity of this approach is $O(n^2)$ where n is the number of nodes in the tree. Why is that the case? Consider a list like tree T_1 . It's height is n , and each call to `diameter` involves a call to `height` which has a linear complexity. So for each node we need to do linear work. We can do better than this, however, if we are willing to sacrifice some space for time.

```

1  int depth(Node<int>* root)
2  {
3      if (!root)
4          return 0;
5      int ans = 0;
6      if (root->left)
7          ans = 1 + depth(root->left);
8      if (root->right)
9          ans = std::max(ans, 1 + depth(root->right));
10
11     return ans;
12 }
13 int diameter_of_binary_tree_quadratic(Node<int>* root)
14 {
15     if (!root)
16         return 0;
17     int l, r;
18     l = r = 0;
19     if (root->left)
20         l = 1 + depth(root->left);
21     if (root->right)
22         r = 1 + depth(root->right);

```

```

23     return std::max(
24         std::max(l + r, diameter_of_binary_tree_quadratic(root->left)),
25         diameter_of_binary_tree_quadratic(root->right));
26 }

```

Listing 30.1: Sample Caption

30.2.2 Linear time and space

The key idea allowing us to go from quadratic to linear time is the realization that in order to calculate the height of a node we also need to calculate the height of all of its descendants. Therefore while we calculate the height of a node, we can save the height for all its descendants so that we do not have to repeatedly recalculate it. If we inspect the function `depth` in Listing 30.1 we can see that in order to calculate the height of the current node we also calculate the height of its left and right child. All that is necessary, therefore, is to cache the result of height and use the cache as shown in Listing 30.2. This solution has linear complexity because the full visit of the tree will only be done once and then all subsequent queries to `height` will be available in the cache.

```

1  int depth(Node<int>* root)
2  {
3      if (!root)
4          return 0;
5      int ans = 0;
6      if (root->left)
7          ans = 1 + depth(root->left);
8      if (root->right)
9          ans = std::max(ans, 1 + depth(root->right));
10
11     return ans;
12 }
13 int diameter_of_binary_tree_quadratic(Node<int>* root)
14 {
15     if (!root)
16         return 0;
17     int l, r;
18     l = r = 0;
19     if (root->left)
20         l = 1 + depth(root->left);
21     if (root->right)
22         r = 1 + depth(root->right);
23     return std::max(
24         std::max(l + r, diameter_of_binary_tree_quadratic(root->left)),
25         diameter_of_binary_tree_quadratic(root->right));
26 }

```

Listing 30.2: Sample Caption

31. Largest square in a binary matrix

Introduction

Imagine you are given a black and white image represented as a boolean matrix of size $N \times M$ where 0 and 1 in the matrix correspond to a black and a white pixel respectively. Such images are more common than may be expected as they are often the output of digital image processing algorithms such as masking or thresholding. Further analysis of this kind of image often requires identifying homogeneous portions of the image. The problem described in this chapter deals with a simple type of image processing algorithm to determine the size of the largest square area of white pixels of a binary bitmap. We will walk through a number of solutions, starting from a naive brute-force one and ultimately moving to a more sophisticated, more complex and definitely more efficient one.

31.1 Problem statement

Problem 45 Given a 2D boolean matrix M , return the area of the largest square containing only one cell.

■ Example 31.1

Given the following matrix the function returns 9. The largest square has side length of 3 and the coordinates of the top-left corner are (2,1). Cells belonging to the largest square are highlighted.

0	0	1	1	1
0	0	1	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	0

■ Example 31.2

Given the following matrix the function returns 4. The side of the largest square is 2 and the top-left coordinates are (2,2). Cells belonging to the largest square are highlighted.

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

31.2 Discussion

In the next section we will analyze a number of possible approaches to this problem. We start by looking at a few brute-force approaches so to then move towards more elaborate and more time and space efficient dynamic programming solutions.

31.2.1 Brute-force

31.2.1.1 Incremental side

The first brute-force approach consists of trying to find the largest square made entirely of set (i.e. holding a value of 1) cells by visiting each set cell and treating it as if it was the top-left corner of a square. Given that calculating the largest square having that cell as the top-left corner is easy the answer to the problem is just the largest value over all the set cells in the matrix. In order to find out what the value of the largest square having cell (x,y) as the top-left corner we can try to build squares of incrementally larger sides around it, starting from side length 1. At first we try to build a square of size 1. If that is possible we try size 2, then 3, and so on, until it is impossible or we hit the boundaries of the matrix. The answer for the cell (x,y) is the last value for a side for which we were able to construct a square. Consider for example Figure ?? where, in order to find the value of the largest square that can be built from cell $(0,1)$, all squares highlighted have to be fully checked. This approach is clearly correct because eventually we find all squares in the matrix, and it has a complexity of (assuming, with no loss in generality, $N \leq M$) $O(N^4M)$. This is because there are $O(NM)$ possible starting point for a square, $O(N)$ possible values for the side value and checking whether a square is valid costs $O(N^2)$ (all cells in the square needs to be checked). A possible implementation of this idea is shown in the Listing 31.1.

```
1
2 [[nodiscard]] int largerSquareFrom(
3     const vector<vector<int>>& matrix,
4     const std::pair<size_t, size_t>& top_left_corner,
5     const size_t rows,
6     const size_t cols)
7 {
8     const auto [x, y] = top_left_corner;
9
10    int k = 0;
11    bool good = true;
12    while (good && ((x + k) < rows) && ((y + k) < cols))
13    {
14        for (size_t i = x; good && i <= x + k; i++)
15        {
16            for (size_t j = y; good && j <= y + k; j++)
17            {
18                if (!matrix[i][j])
19                {
20                    return k;
21                }
22            }
23        }
24        ++k;
25    }
26    return k;
27 }
28
29 [[nodiscard]] int maximal_square_brute_force_1(
30     const vector<vector<int>>& matrix)
31 {
32     if (matrix.size() <= 0 || matrix[0].size() <= 0)
33         return 0;
34
35     const auto rows = matrix.size();
36     const auto cols = matrix[0].size();
37     int ans = 0;
38     for (size_t i = 0; i < rows; i++)
```

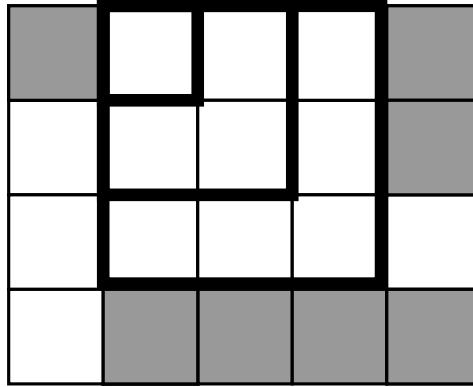


Figure 31.1: This figure shows the squares that are checked by the brute-force approach for solving the square in matrix problem. From the cell (0,1) we first try to build a square of side 2, and when that is verified to be possible, a square of size 3 is tried. This also succeeds and so a square of side 4 is checked, with a negative outcome. Thus 3 is the largest square having cell (0,1) as top left corner.

```

39     for (size_t j = 0; j < cols; j++)
40         if (matrix[i][j])
41             ans = std::max(ans, largerSquareFrom(matrix, {i, j}, rows, cols));
42
43     return ans * ans;
44 }
```

Listing 31.1: Brute force solution to the *square in matrix* problem using incremental side probing.

31.2.1.2 Walking diagonally

The idea presented in Section 31.2.1.1 can be significantly improved by realising it is not really necessary to check, given a cell (x,y) , squares of all possible side lengths having it as the top left corner fully and consecutively. The idea is that we can walk diagonally (towards the bottom-right cell) from a cell (x,y) (by incrementing both x and y) and, for every step i we take, we check whether all the elements to the left of $(x+i,y+i)$ and to the right of y are set and also whether all the cells in the columns above $(x+i,y+i)$ and below the cell $(x,y+i)$ are set (see Figure ??). If both conditions are true it means that we can construct a square of side i . We can then proceed one step further until a 0 is found among the checked cells on the left or above of (x,y) . If we were able to perform t diagonal steps, it means we have found out that the largest square having (x,y) as the top-left corner has an area of t^2 . The final answer is the largest value we calculated this way across all cells that are set. See Figure ?? where the numbers represents the cells that are checked during the corresponding step. Every highlighted square depicts one of the squares that is checked by the algorithm.

The time complexity of this approach is $O(N^3M)$, lower than the previous solution. As shown in Section 31.2.1.1, there are $O(NM)$ potential top-left corners for a square and for each of them $O(N)$ diagonal steps. Each diagonal steps costs $O(N)$ as, in the worst case scenario, we must check one entire row and column. Thus the complexity of calculating the value of the largest square having a certain cell as the top-left corner is $O(N^2)$ (See Figure ?? where you can see that no cell is checked twice during this step).

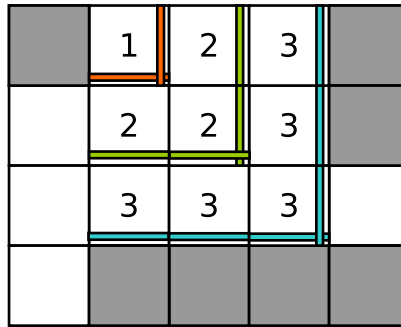


Figure 31.2: This figure depicts the process of calculating the value of the side of the largest square having as a top-left corner cell (0,1). Each cell is labeled with a number representing the step at which that cell is checked. Note that no cell is checked twice.

Listing 31.2 shows a possible implementation of the idea described here. Note how Listings 31.2 and 31.1 for both the solutions proposed so far are very similar, with the only difference being in how the size of the largest square constructible from a certain top-left cell is computed.

```

1
2 [[nodiscard]] int largerSquareFrom(
3     const vector<vector<int>>& matrix,
4     const std::pair<size_t, size_t>& top_left_corner,
5     const size_t rows,
6     const size_t cols)
7 {
8     const auto [x, y] = top_left_corner;
9
10    int k = 0;
11    bool good = true;
12    while (good && ((x + k) < rows) && ((y + k) < cols))
13    {
14        for (size_t i = x; good && i <= x + k; i++)
15        {
16            for (size_t j = y; good && j <= y + k; j++)
17            {
18                if (!matrix[i][j])
19                {
20                    return k;
21                }
22            }
23        }
24        ++k;
25    }
26    return k;
27 }
28
29 [[nodiscard]] int maximal_square_brute_force_1(
30     const vector<vector<int>>& matrix)
31 {
32     if (matrix.size() <= 0 || matrix[0].size() <= 0)
33         return 0;
34
35     const auto rows = matrix.size();
36     const auto cols = matrix[0].size();

```

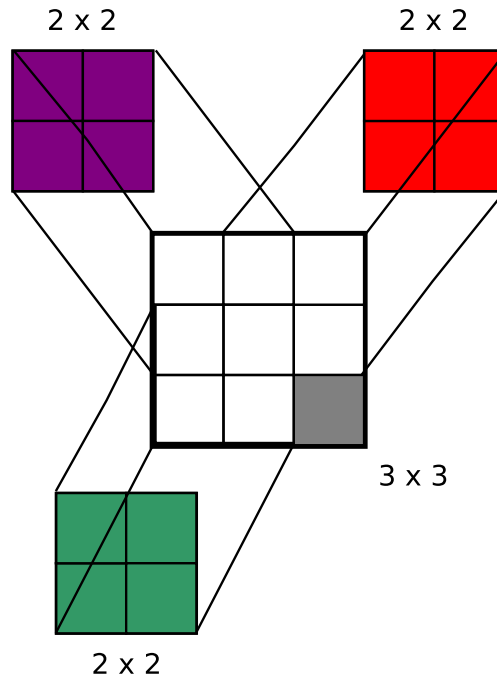



Figure 31.3: This figure shows how a square of side 3 can be decomposed into three smaller subsquares of side 2.

```

37  int ans          = 0;
38  for (size_t i = 0; i < rows; i++)
39      for (size_t j = 0; j < cols; j++)
40          if (matrix[i][j])
41              ans = std::max(ans, largerSquareFrom(matrix, {i, j}, rows, cols));
42
43  return ans * ans;
44  }

```

Listing 31.2: C++ brute force solution using diagonal steps for solving the *square in matrix* problem.

31.2.2 Dynamic programming

31.2.2.1 General Idea

This problem can be solved faster than $O(N^3M)$ time with the help of dynamic programming. The solution is based on the fact that for any square of size $k \times k$ having as bottom-left corner the cell (x, y) it also has a top, top-left and left subsquares of size $(k-1) \times (k-1)$. As an example see Figure ?? which shows a square of size 3×3 decomposed into 3, 2×2 subsquares.

Suppose $DP(i, j)$ is a function returning the size of the largest square having the cell (i, j) as its bottom-right corner (what is discussed in this Section can be easily adapted so that (i, j) is the top-left corner). Clearly the values of $DP(0, j) : 0 \leq j \leq M$ (all the cells belonging to the first row) and $DP(i, 0) : 0 \leq i \leq N$ (all the cells of the first column) are the same as the values in the input matrix (either 1 or 0 depending on the corresponding value in the input matrix M). This is explained by the fact that a cell in the first row or

Input Matrix					DP				
0	1	1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	2	2	0
0	1	1	1	1	0	1	2	3	1
0	1	1	1	1	0	1	1	3	2
0	0	1	1	1	0	0	1	2	1

Figure 31.4: This figure shows the values of each cell in the original matrix and the corresponding values for the largest square having that cell as the bottom-right corner. Colors are used to highlight the cells that are part of the same square. Note how the cell holding a 3 in the bold frame denotes that a square of side 3 can be constructed from it with cells belonging to the top (in red), top-left (purple) and green (right).

column lacks one or more of the subsquares described above. For instance for a cell in the first row, the top subsquare is missing (as there are no cells above it) and thus it is impossible to construct a square having a side larger than 1 starting from it. For all the other (internal) cells the value of DP can be easily calculated by using the Equation 31.1. The formula is basically stating that if we have a cell (i, j) set to 1 then from it we can construct a larger square whose size depends on the size of the smallest square among the neighboring subsquares.

$$DP(i, j) = \min\{DP(i-1, j), DP(i-1, j-1), DP(i, j-1)\} + 1 \quad (31.1)$$

Figure ?? shows the idea above in practice. The value 2 in $DP(1, 3)$, $DP(1, 2)$ and $DP(2, 2)$ signifies that there is a square of size 2×2 up to (having that cell as bottom-right corner) those cells in the original matrix. By combining those 3 squares with the set cell at location $(2, 3)$ we can build a larger square of size 3×3 . Now consider the value of $DP(3, 4) = 3$. The entries for the neighboring cells $DP(3, 4) = 3$ and $DP(3, 4) = 3$ imply that a square of side 3×3 exists up to their indices, but the entry at location $DP(2, 4) = 1$ indicates that up to that cell only a square of size 1×1 exists and this prevents cell $(3, 4)$ having a maximum square size larger than 2 (in other words, making a square of size 3 from $(3, 4)$ is limited by the cells above it).

The function DP in Equation 31.1 is recursive and when drawing its recursion tree, as shown in Figure ?? (which depicts part of the recursion tree for $DP(3, 3)$), we can easily see that:

- the tree is complete and therefore has an exponential number of nodes.
- there are duplicate nodes.

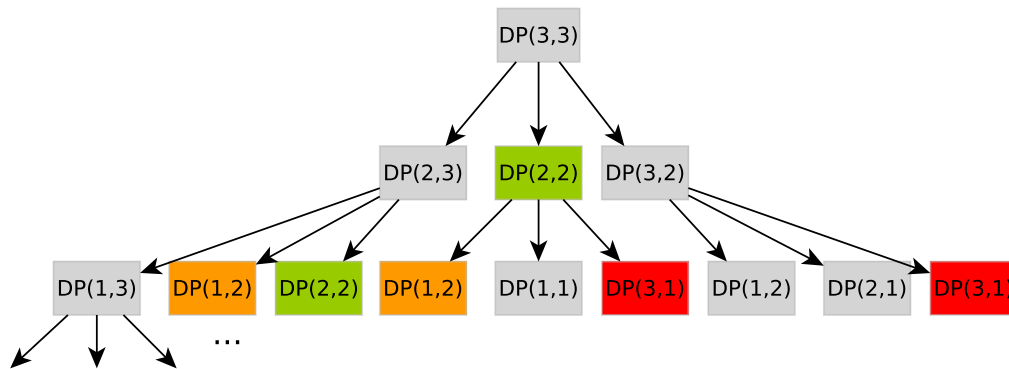


Figure 31.5: This figure is an example of the recursion tree for the Equation 31.1. Note that the nodes are duplicates (these are denoted by the same color).

The number of possible unique function calls to DP is bounded by the values of its parameters which is far less than exponential. In fact, it is proportional to $N \times M$ (the size of the input matrix) as there are only N possible values for i and M possible values for j in $DP(i, j)$. Therefore the only way for the recursion tree to have an exponential number of nodes is for some of them to be duplicates. Given this fact we can conclude that the problem exposes both the property of **optimal substructure**, because it can be solved by optimally solving smaller subproblems, and has **overlapping subproblems**. As such, we can employ dynamic programming and solve each subproblem only once. In the Sections 31.2.2.2 31.2.2.3 we will go through the details of the two ways of implementing dynamic programming algorithm

- top-down
- bottom-up

31.2.2.2 Top-Down

This is probably the easiest way of implementing the dynamic programming solution for the problem described in Section 31.2.2.1 as we can directly translate the Equation 31.1 to a recursive function. The important fact that allows us to implement it efficiently is to remember the solution to a subproblem by using a cache (which can easily be a 2D matrix or anything that allows us to map (i, j) to a integer, like a hashmap) as shown in Listing 31.3. As you can see, in the implementation shown here we use as a cache a `std::unordered_map<Cell, int>` (where `Cell` is just an alias for `std::tuple<int, int>`) where the function `CellHash` is the type we provide to `std::unordered_map` so it knows how to hash a `Cell`. The main driver function is the function `int maximal_square_in_matrix_top_down(const vector<vector<int>>& matrix)` which operates in a similar manner as in the other solutions seen so far and calculates the final results by looping over all cells of the matrix and calculating the largest square having that cell as a bottom-right corner for each of them. The most important part of the code is the recursive function `int maximal_square_in_matrix_top_down_helper(const vector<vector<int>>& matrix, Cache& cache, const`

`Cell cell, const size_t rows, const size_t cols)` which takes as input the original matrix, the cache (by reference because it needs to update it along the way) the `Cell` which it operates on and the `rows` and `cols` of the input matrix (we are passing it along so we avoid retrieving it from the `matrix` object for every invocation).

This function which has two base cases:

1. when the current cell has value of 0 no square can have it as the bottom right corner so 0 is returned.
2. when we ask for the maximal square from a cell that is outside the bounds of the original matrix we simply return 0. Such cell does not exist so we cannot have a square having it as the bottom-right corner.
3. when the value for a cell has already been calculated and it is thus already in the cache we avoid the expensive work and simply return the value in the cache. This is how duplicate work is avoided.

If none of the base-case conditions are true then, as per the description of the Equation 31.1, we calculate the maximal square recursively calling `maximal_square_in_matrix_top_down_helper` on the cells immediately:

- above: `Cell(i-1,j)`
- to the left: `Cell(i,j-1)`
- to the top-left: `Cell(i-1,j-1)`

When the values for all the recursive calls above is finally calculated we save it in the cache before returning it to the caller so it will be available on subsequent calls.

The complexity of this implementation is $O(NM)$ because the function `maximal_square_in_matrix_top_down_helper` is executed only $O(NM)$ times (you can verify this by printing the `cell` after the bases cases in `maximal_square_in_matrix_top_down_helper` and see that no duplicates appear in the list).

```

1
2 using Cell = std::tuple<int, int>;
3
4 struct CellHash : public std::unary_function<Cell, std::size_t>
5 {
6     std::size_t operator()(const Cell& k) const
7     {
8         return std::get<0>(k) ^ std::get<1>(k);
9     }
10 };
11 using Cache = std::unordered_map<Cell, int, CellHash>;
12
13 int maximal_square_in_matrix_top_down_helper(const vector<vector<int>>& matrix,
14                                             Cache& cache,
15                                             const Cell cell,
16                                             const size_t rows,
17                                             const size_t cols)
18 {
19     auto [i, j] = cell;
20
21     if ((i >= rows || j >= cols) || (!matrix[i][j]))
22         return 0;
23
24     if (cache.contains(cell))
25         return cache[cell];
26
27     // uncomment the line below to verify no work for the same cell is done
28     // twice std::format("Recursive call for ({0:d},{1:d})\n", i,j);
29
30     const int ans = std::min({maximal_square_in_matrix_top_down_helper(
31                             matrix, cache, Cell{i - 1, j}, rows, cols),
32                             maximal_square_in_matrix_top_down_helper(
33                                 matrix, cache, Cell{i - 1, j - 1}, rows, cols),
34                             maximal_square_in_matrix_top_down_helper(
35                                 matrix, cache, Cell{i, j - 1}, rows, cols)})
36         + 1;

```

```

37     cache[cell] = ans;
38     return ans;
39 }
40
41 int maximal_square_in_matrix_top_down(const vector<vector<int>>& matrix)
42 {
43     if (matrix.size() <= 0 || matrix[0].size() <= 0)
44         return 0;
45
46     const auto rows = matrix.size();
47     const auto cols = matrix[0].size();
48     Cache cache;
49
50     int ans = 0;
51     for (size_t i = 0; i < rows; i++)
52         for (size_t j = 0; j < cols; j++)
53             ans = std::max(ans,
54                             maximal_square_in_matrix_top_down_helper(
55                                 matrix, cache, Cell{i, j}, rows, cols));
56     return ans * ans;
57 }

```

Listing 31.3: C++ dynamic programming top-down solution for solving the *square in matrix* problem.

31.2.2.3 Bottom-up

The other way of implementing a dynamic programming algorithm is to use a bottom-up approach. The idea in this case is to start filling the cache with values we know upfront without doing any work. We have already mentioned some of those values; namely the ones belonging to cells of the first row and columns. Once those values are in the cache we can then move on to calculating the values for the second row. According to the Equation 31.1 in order to calculate $DP(1, 1)$, the values for $DP(0, 1)$, $DP(1, 0)$ and $DP(0, 0)$ are needed. Because they all belong to either the first or second row, and because values for cells in those locations are already in the cache we can calculate $DP(1, 1)$. When $DP(1, 1)$ is in the cache, then we can also calculate $DP(1, 2)$ and so on for all the cells in the row. The same reasoning can be applied to the rest of the rows. Eventually the cache will be filled completely and thus the answer is just the largest value in the cache.

Listing 31.4 shows a possible implementation of such idea.

```

1 int maximal_square_in_matrix_bottom_up(const vector<vector<int>>& matrix)
2 {
3     if (matrix.size() <= 0 || matrix[0].size() <= 0)
4         return 0;
5
6     const auto rows = matrix.size();
7     const auto cols = matrix[0].size();
8     // first row and first column have the same values as in the original
9     // input matrix
10    std::vector<vector<int>> cache(matrix);
11
12    // is there a 1 in the first row?
13    int ans =
14        std::find(begin(matrix[0]), end(matrix[0]), 1) != end(matrix[0]) ? 1 : 0;
15
16    // is there a 1 in the first column?
17    for (size_t i = 1; i < rows; i++)
18    {
19        if (matrix[i][0])

```

```

20     {
21         ans = 1;
22         break;
23     }
24 }
25
26 for (size_t i = 1; i < rows; i++)
27 {
28     for (size_t j = 1; j < cols; j++)
29     {
30         if (matrix[i][j])
31         {
32             cache[i][j] =
33                 std::min({cache[i - 1][j], cache[i][j - 1], cache[i - 1][j - 1]})
34                 + 1;
35         }
36         ans = std::max(ans, cache[i][j]);
37     }
38 }
39 return ans * ans;
40 }

```

Listing 31.4: C++ dynamic programming bottom-up solution for solving the *square in matrix* problem.

This implementation initializes the `ans` variable with 1 or 0 depending on if there is a cell set in the first row or column or not. The rest of the code loops through the the rest of the cells starting from cell (1,1) and avoiding the first row and column because - as stated before - the values for these cells are known upfront. For each of these cells the final value is calculated by using Equation 31.1.

The complexity of the code in Linst 31.4 is clearly $O(NM)$ (probably more obvious in here than in the top-down solution).

31.3 Conclusion

32. Sudoku

Introduction

The game of *Sudoku*^① has become hugely popular in the last 20 years to There are now countless websites and magazines dedicated to these mathematical-logic-based number-placement puzzles. The objective of this is to fill a nine-by-nine (9x9) grid (subdivided in 3×3 subgrids) with digits so that each:

- **row**,
- **column**,
- 3×3 **subsquare section**

contains a number between 1 and 9, with the constraint that each number can appear only once in each section. The puzzle is given as a incomplete grid where only some of the cells are filled.

This chapter describes how to write a very basic and simple sudoku solver based on backtracking that can be implemented fast enough for a programming interview. Having played this puzzle before might help during the interview but it is not essential as the rules are easy enough to understand in a few minutes.

32.1 Problem statement

Problem 46 Write a function that takes as an input a sudoku grid and returns its solution. The input sudoku grid is given as a string of length 81 representing the grid in a row-major manner^a where empty cells are represented by the character '0'. ■

^aIn row-major order, the rows of the grid are stored next to each other in the string.

■ Example 32.1

Given the input string ``000060280709001000860320074900040510007190340003006002002970000300800905500000021`` the function returns ``431567289729481653865329174986243517257198346143756892612975438374812965598634721``. See Figures 32.1 and 32.2 for their 2D representation. ■

32.2 Clarification Questions

Q.1. Is the input string guaranteed to only contains numeric characters and be the right size?

Yes the string is guaranteed to be encoding a valid sudoku

^①The literal meaning of “Su-doku” in Japanese is “the number that is single”.

		4	3			2		9
		5			9			1
	7			6			4	3
		6			2		8	7
1	9				7	4		
	5			8	3			
6						1		5
		3	5		8	6	9	
	4	2	9	1		3		

Figure 32.1: Example of a 9×9 sudoku.

8	6	4	3	7	1	2	5	9
3	2	5	8	4	9	7	6	1
9	7	1	2	6	5	8	4	3
4	3	6	1	9	2	5	8	7
1	9	8	6	5	7	4	3	2
2	5	7	4	8	3	9	1	6
6	8	9	7	3	4	1	2	5
7	1	3	5	2	8	6	9	4
5	4	2	9	1	6	3	7	8

Figure 32.2: Solution to the puzzle shown in Figure 32.1.

32.3 Discussion

The general problem of solving a sudoku (of size $n \times m$) is NP-complete^② and thus an efficient (polynomial-time) solution is not yet known. The simple brute-force algorithm would have to try each available number across all empty cells and therefore would have a runtime complexity of $O(N^{N^2})$, where N is size of the Sudoku puzzle. For a classic 9×9 puzzle $N = 9$ and the number of operations required would be at most 2×10^{77} operations to find a solution which would make this approach impractical.

In practice the number of operations varies hugely according to the difficulty of the puzzle itself and especially according to the number of given clues which, in turn, limit the options for each empty cell. Clues reduce the number of possible states the grid can be and in which the rules of the puzzle are not violated. The more clues there are, the higher the number of invalid states. An algorithm can take advantage of that fact to avoid those invalid states. For example, a 17-clue puzzle with diagonal symmetry is one of the hardest to solve due to the large number of candidates and branches^③.

32.3.1 Backtracking

Backtracking is a good approach to use to solve this problem considering that it has the following characteristics:

- potentially large puzzle-states search space
- many *invalid* states we can skip visiting

For a more detailed explanation of backtracking see [\[backtracking\]](#).

In a nutshell the solution proposed in this section works by visiting the empty cells starting from the first one from the left, filling it in with a feasible digit (i.e. a digit that does not take the grid to an invalid state) and then doing the same for every other empty cell. If at any point there is no available digit for an empty cell then a backtracking step occurs. The choice for the previous cell is then changed and the whole process repeats until either all the empty cells are filled (in this case we have a valid solution) or there are no more options for the first cell (in this case the puzzle has no solution and it is invalid). A backtracking solution would solve a puzzle by placing the digit '1' in the first empty cell and checking if it is allowed to be there (i.e. that no rules are broken). If there are no violations (checking row, column, and box constraints) then the algorithm advances to the next cell and places a '1' in the next empty cell. When checking for violations, if it is discovered that the "1" is not allowed, the value is advanced to "2". If a cell is discovered where none of the 9 digits is allowed, then the algorithm leaves that cell blank and moves **back** to the previous cell. The value in that cell is then incremented by one and the whole process repeats. Clearly, this method will eventually find a solution if the puzzle is valid because all possible valid states for the grid will be tested.

Listing 32.1 shows a possible implementation of the backtracking idea described above. The public interface of the `SudokuSolver` class consists only of a constructor `SudokuSolver::SudokuSolver(std::string)` taking a sole input a `std::string`, the problem input, and the `std::string` `SudokuSolver::solve()` function that is responsible for returning the solution. The constructor is responsible for analyzing the input and storing the indices of all the empty cells (i.e. the cells the backtracking function is going to try to fill) in a

^②NP stands for Non-deterministic Polynomial time. A problem that can be solved in polynomial time (efficiently) by a non-deterministic Turing machine and for which the solution can be efficiently verified to be correct by a deterministic Turing machine. A problem in NP is complete if by solving it you are able to solve every other problem in NP. This means that an NP-complete problem is at least as hard as every other problem in NP.

^③17 clues has been proven to be the lower-bound for having a puzzle with a **unique** solution

`vector(std :: vector < int > blankCells).`

The core implementation function is the `bool solve_helper(const int i)` recursive function that takes as input an integer `cell` representing the index of an empty cell in the input string. The base case for this function is when `i >= blankCells . size ()` i.e. there are no more empty cells to be filled. The rest of the function is straightforward because it only consists of a loop trying all possible numbers for that cell from '1' to '9'. The `canInsert(char x, int pos)` function is responsible for deciding whether a character `x` can be placed in a certain cell `pos`. The check is performed by examining whether any of the rules described above would be broken by having `x` at cell `pos`. If no rules are broken then the function `solve_helper` calls itself recursively on the **next** empty cells i.e. `cell+1`. If none of the values tried in the loop yield a valid solution then the function returns false (no value can be inserted at this location without violating one or more rules).

Because the input is a linear representation of a grid, which is a 2D structure, and the constraints of the puzzle are for the 2D portion of the grid itself, the code is further complicated by calculations that are necessary for the functions `canInsertInRow`, `canInsertInCol` and `canInsertInSquare` to be able to map the cells belonging to the same row, column or subsquare to the input 1D input string. The functions `getRow`, `getCol`, `getSubsquare` are used to - given a index in the 1D input string - retrieve the corresponding row, column and subsquare index in the 2D grid. These functions are used in the `canInsertInRow`, `canInsertInCol` and `canInsertInSquare` functions that are responsible for verifying that the constraints on the row, column and subsquare, respectively, are not violated when we try to insert a certain value in a cell. In order to do this they need to be able to calculate the indices of all cells belonging to the same row, columns and subsquare. Specifically:

- the `canInsertInRow` function checks all the cells belonging to the same row. Given a row r then all 9 cells belonging to it have indices in the range $[9r, 9(r+1)]$ (See Figure ??).
- It becomes more complex when it comes to checking cells in the same column, in the function `canInsertInRow`. The column c to which a cell in the input string with index x belongs can be found by using the following formula: $c = x \pmod{9}$. This means that the very first cells in the input belonging to column c is located at index c and all subsequent cells of the column are distanced 9 cells from each other. More formally, the index for the k^{th} cell of the column in the input string is: $P(k, c) = 9k + c$.
- The hardest check is the one for subsquares in the `canInsertInSquare(char x, int s)` function because, in order to check whether it is possible to insert the value x in the subsquare s , it has to compare x to all the other non-empty cells of the same subsquare. This goal is accomplished in two steps:
 1. First, the index of the $F(s)$ top left corner of the subsquare s is calculated by using the following formula: $F(s) = (27 \lfloor \frac{s}{3} \rfloor + (3 \times (s \pmod{3})))$. In order to understand the formula, we need first to note that the subsquares are organized into 3 rows each of size 3 (for a total of 9 subsquares, see Figure ??). Clearly each subsquare contains 9 cells, and thus, a full row of subsquares contains $3 \times 9 = 27$ cells. $\frac{s}{3}$ is a value representing how many full subsquare rows come before s . Clearly we can skip all the cells belonging to those subsquares, because all cells in them come before $F(s)$. The value $(27 \lfloor \frac{s}{3} \rfloor)$ is thus an index pointing to a cell at the beginning of the row where $F(s)$ is located. All we need to do now is to advance to the correct subsquare in the row and we can do that by looking at the position of the subsquare **in the row** which clearly is $(s \pmod{3})$; s can either be either on the left $((s \pmod{3}) = 0)$, center $((s \pmod{3}) = 1)$ or on the right side of the row $((s \pmod{3}) = 2)$. Given each subsquare has width

of 3 we can jump to the correct location by using $3 \times (s \bmod 3)$).

2. once $F(s)$ is known then it is easy to retrieve the indices of all the cells in the subsquare by using the ideas adopted for `canInsertInRow` and `canInsertInCol`.

```
1 #include <optional>
2 class SudokuSolver
3 {
4 public:
5     SudokuSolver(std::string _problem) : problem(std::move(_problem))
6     {
7         assert(problem.size() == 81);
8     }
9     auto solve()
10    {
11        printSudoku();
12        getBlankCells();
13        solve_helper(0);
14        printSudoku();
15        return problem;
16    }
17
18 private:
19     void getBlankCells()
20     {
21         for (int i = 0; i < problem.size(); i++)
22             if (problem[i] == '0')
23                 blankCells.push_back(i);
24     }
25
26     char intToChar(const char num)
27     {
28         assert(num >= '0' && num <= '9');
29         return num;
30     }
31
32     bool canInsertInRow(const auto x, const auto row)
33     {
34         assert(row >= 0 && row < 9);
35         auto start = std::begin(problem) + 9 * row;
36         auto end = start + 9;
37         return find(start, end, intToChar(x)) == end;
38     }
39
40     bool canInsertInCol(const auto x, const auto column)
41     {
42         int curr = column;
43         while (curr < 81)
44         {
45             if (problem[curr] == intToChar(x))
46                 return false;
47             curr += 9;
48         }
49         return true;
50     }
51
52     bool canInsertInSquare(const auto x, const auto square)
53     {
54         int start_cell = (3 * 9 * (square / 3)) + (3 * (square % 3));
55         for (int i = 0; i < 3; i++)
56         {
57             const bool found = (problem[start_cell + i * 9] == intToChar(x))
```

```

58         || (problem[start_cell + i * 9 + 1] == intToChar(x))
59         || (problem[start_cell + i * 9 + 2] == intToChar(x));
60     if (found)
61         return false;
62 }
63 return true;
64 }
65 bool canInsert(const auto x, const auto pos)
66 {
67     const auto row    = pos / 9;
68     const auto col    = pos % 9;
69     const auto square = (row / 3) * 3 + (col / 3);
70     return canInsertInRow(x, row) && canInsertInCol(x, col)
71         && canInsertInSquare(x, square);
72 }
73
74 void printSudoku()
75 {
76     for (int i = 0; i < 9; i++)
77     {
78         for (size_t j = 0; j < 9; j++)
79         {
80             cout << problem[i * 9 + j] << " ";
81         }
82         cout << endl;
83     }
84 }
85
86 bool solve_helper(const int i)
87 {
88     if (i >= blankCells.size())
89     {
90         return true;
91     }
92     auto pos = blankCells[i];
93     cout << pos << " ++++++ " << endl;
94     // printSudoku();
95     // cout<<endl;
96     for (char x = '1'; x <= '9'; x++)
97     {
98         problem[pos] = '0';
99         /* if(pos == 27)
100             cout<<"here";*/
101         if (canInsert(x, pos))
102         {
103             problem[pos] = x;
104             if (solve_helper(i + 1))
105                 return true;
106         }
107     }
108     problem[pos] = '0';
109     return false;
110 }
111
112 std::string problem;
113 std::vector<int> blankCells;
114 };
115
116 std::string solve_sudoku_backtracking(std::string& sudoku)
117 {
118     SudokuSolver solver(sudoku);

```

```
119     solver.solve();  
120     return solver.solve();  
121 }
```

Listing 32.1: Backtracking solution to the Sudoku problem.

32.4 Conclusion

33. Jump Game

Introduction

In this chapter, we will investigate whether a solution exists for a game played in an array where you are the only player and you are initially located at the first cell of the array. Your goal is to get to the last cell by jumping from one cell to another a specified number of times. The array contains information about the length of the jump you can take from a cell.

There are several possible different solutions to this problem and, in this Chapter, we will discuss the most common. In particular:

- In Section 33.2 we take a look at the most intuitive approach where we try all possible jumps in a backtracking-like manner;
- In Section 33.3 we will refine the solution of Section 33.2 into one that uses a clever insight to visit the cells efficiently.
- Finally, in Section 33.4 we will discuss an efficient and concise greedy solution.

33.1 Problem statement

Problem 47 Write a function that takes as input an array I of non-negative integers. You are initially positioned at the beginning of the array (at index 0) and your goal is to jump from cell to cell to the end of the array (cell $|I| - 1$). Each cell i of the array contains the maximum length for a jump that can be made from cell i . If you are at index j you are allowed to jump to any cell within the following range: $[j - I_i, j + I_i]$. The function should return true if you can reach the last cell of the array, otherwise, it should return false.

■ Example 33.1

Given $I = [2, 3, 1, 1, 4]$ the function returns **true**. You jump from cell 0 to 1 and then take a 3 cells wide jump to the end of the array. See Figure 33.1. ■

■ Example 33.2

Given $I = [3, 2, 1, 0, 4]$ the function returns **false** because it is impossible to reach any cells with index higher than 3. See Figure 33.2: there is no incoming edge for the node with label 4. ■

33.2 Backtracking

The first solution that we will investigate is based on an idea similar to the DFS where I is treated as an implicit graph where each cell can be thought of as being a node of a graph and is connected to all the other cells that can be reached by jumping from it. The set of cells you can reach from a given cell c is identified by the length of the jump you can perform from c (a value that is stored within c itself). The idea is to use DFS to check

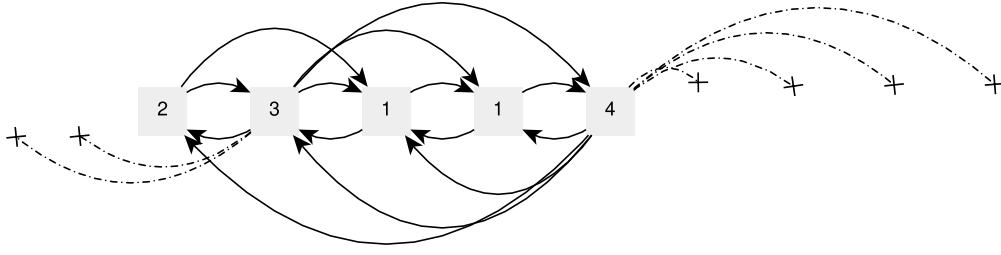


Figure 33.1: Visual representation (implicit graph) of the problem instance of Example 33.1.

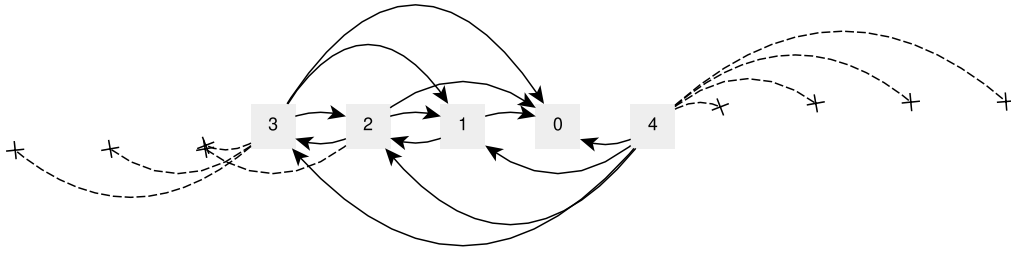


Figure 33.2: Visual representation (implicit graph) of the problem instance of Example 33.2.

whether the last node of the graph is connected with the first one. In other words, we want to answer the following question: is there a path from the first to the last node?

We can proceed by adopting a recursive approach where we try to visit all the nodes that we can reach from the node we currently occupy and to continue this process until either we have reached the last node or there is no more jump to try; in the latter case, there is no way to reach the last node (i.e. the last node is disconnected).

As the implicit graph is not guaranteed to be acyclic, to make this approach work we need to ensure that we do not jump back and forth from one cell to another in a cycle. This can happen if, for instance, you jump from a cell 0 to cell 1 and then back to the cell 0. To overcome this issue, we can only perform forward jumps so that it will be impossible to be stuck in a cycle. When you jump to a cell i from a cell j s.t. $j < i$ (you performed a forward jump) we know that we can also visit all cells $j \leq k \leq i$ (all the cells in between j and i) from j . If we only jump forward, we are not going to need to visit any cell $j \leq k \leq i$ using backward jumps as these cells are visited anyway when processing cells j by performing forward jumps from it.

An implementation of this idea is shown in Listing 33.1. This approach is correct and it will eventually find a solution but it is extremely inefficient. Its complexity is exponential

in time as potentially the same cells are visited over and over^① and constant in space^②.

```

1 bool can_jump_DFS_forward_only_helper(const vector<int>& nums, const int n)
2 {
3     const int tgt = nums.size() - 1;
4     if (n == tgt)
5         return true;
6
7     int r = std::min(tgt, n + nums[n]);
8     for (int i = n + 1; i <= r; i++)
9     {
10         if (can_jump_DFS_forward_only_helper(nums, i))
11             return true;
12     }
13     return false;
14 }
15
16 bool can_jump_DFS_forward_only(const vector<int>& nums)
17 {
18     return can_jump_DFS_forward_only_helper(nums, 0);
19 }

```

Listing 33.1: Exponential time solution to the *jump game* problem where only forward jumps are performed.

33.3 DFS

Another option for solving the cycle problem arising from the algorithm described in Section 33.2 (this solution can be in-fact thought of as optimized backtracking) is to keep track of the cells that we have already visited and every time we are about to perform a jump to a cell we first check whether that cell has already been visited and - if it has - the jump is discarded and not performed. As such, no cell is visited twice, and consequently, the complexity in this case is $O(|I|^2)$. In the worst-case scenario, you must check for each cell whether all the other cells have been already visited. Listing 33.2 shows an implementation of this idea.

```

1 bool can_jump_DFS_helper(const vector<int>& nums,
2                          vector<bool>& visited,
3                          const int n)
4 {
5     const int tgt = nums.size() - 1;
6     if (n == tgt)

```

^①Suppose $W(x)$ is the number of possible ways you can jump from position x to the end of the array at index N . We know that $T(N) = 1$ (the only way to jump from cell N to itself is not to jump at all). For all other cells we have that:

$$\begin{aligned}
 W(x) &= \sum_{i=x+1}^N W(i) \\
 &= W(x+1) + \sum_{i=x+2}^N W(i) \\
 &= W(x+1) + W(x+1)
 \end{aligned}$$

So to calculate $W(X)$ we need the values $W(x+1)$ twice. The recursive tree for W is binary and complete and has height N and therefore contains $O(2^N)$ number of nodes.

^②if we do not consider the spaces utilized by the stack frames during the recursive calls, otherwise, it is linear


```

7     return true;
8
9     visited[n] = true;
10
11     int l      = std::max(0, n - nums[n]);
12     int r      = std::min(tgt, n + nums[n]);
13     bool sol_found = (r == tgt);
14     for (int i = l; i <= r && !sol_found; i++)
15     {
16         if (visited[i])
17             continue;
18
19         sol_found = can_jump_DFS_helper(nums, visited, i);
20     }
21     return sol_found;
22 }
23
24 bool can_jump_DFS(const vector<int>& nums)
25 {
26     std::vector<bool> visited(nums.size(), false);
27     return can_jump_DFS_helper(nums, visited, 0);
28 }

```

Listing 33.2: Quadratic time and linear space DFS solution to the *jump game* problem using a visited array.

Note that one optimization from which this solution (and perhaps also Listing 33.1) can benefit would be to always try to jump the longest distance possible. Although this won't change their asymptotic complexity in practice it might be faster.

33.4 Greedy

There is, however, a much faster solution to this problem using the idea that we can return true if we can jump from the cell at index 0 to a cell from which we can reach the end of the array. If we apply the same reasoning to generic index i we end up with what is essentially a dynamic programming approach that - given $G(x)$ is 1 if you can reach the end of the array from the cell x and 0 otherwise - is based on the following recursive formula:

$$\begin{cases} G(|I| - 1) = 1 \\ G(x) = 1 \text{ if } \exists y > x \text{ s.t. } y < (x + I_x) \text{ and } G(y) = 1 \\ \text{otherwise } G(x) = 0 \end{cases} \quad (33.1)$$

Equation 33.1 shows that a possible implementation would start processing cells from the last to the first and that for each element a linear time lookup for a suitable cell y might be needed. Therefore the complexity of this solution is quadratic in time. However, we can drastically lower its complexity by noting that when processing cell x all we care about is whether the closest cell to the right from which you can reach the end of the array is reachable from x . We can carry this information into a variable m down from cell $|I| - 1$ to cell 0 and update it after a cell is processed and this would effectively allow us to have a linear time solution.

To summarize, the linear time solution for this problem works as follows: We iterate the array I right-to-left and for each cell x we check whether we can reach m jumping from x . If we can then x is the new leftmost cell from which we can reach the end of the array, thus $m = x$. Otherwise, we continue by processing cell $x - 1$ in a similar manner. Eventually, we will have processed all cells and therefore we can return true if $m = 0$ meaning that cell 0 is the leftmost cell from which we can jump to location $|I| - 1$, and false otherwise.

```

1 bool can_jump_linear(const vector<int>& nums)
2 {
3     const int size = nums.size();
4     int m          = size - 1;
5     for (auto i = size - 2; i >= 0; i--)
6     {
7         const int max_reach = i + nums[i];
8         if (max_reach >= m)
9             m = i;
10    }
11    return m == 0;
12 }

```

Listing 33.3: Greedy solution where we use the fact that the DP solution described by Equation 33.1 can be optimized if we only consider if it is possible to reach the closest cell from which we can jump to the end of the array.

33.5 Jump Game 2

33.6 Problem statement

Problem 48 Given an array of non-negative integers I , you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Your goal is to reach the last index in the minimum number of jumps.

You can assume that you can always reach the last index.

■ Example 33.3

Given $I = [2, 3, 0, 1, 4]$ the function returns **2**. You jump from cell 0 to 1 and then take a 3 cells wide jump to the end of the array. See Figure 33.1. ■

33.6.1 Discussion

The key difference this variation has w.r.t. the version in Section 33.1 is that here we are guaranteed that it is possible to reach the last location of the array by starting from the beginning and performing some combination of forward jumps.

When performing the first jump, we know we can reach cells in the range $[0, 0 + I[0]]$. Which of these cells we should jump to? The answer is, always jump to the cell at index j in $[0, 0 + I[0]]$ that gets us the farthest! In other words, choose the cell j s.t. $j + I[j]$ is maximum. Why is this the case? The reasoning behind it is that, jumping to any other cell other j , say cell $0 \leq k \leq 0 + I[0]$ with $k \neq j$ does not decrease the overall number of steps to get to the final cell because from cell j we can reach every cell we can reach from cell k plus potentially some more cells that are unreachable from cell j . For example let's examine Figure 33.3. Among all the cells we can reach from cells 0 (in **red**) the cell at index 1 is the one through which we can travel the farthest. If we decide to jump at the cell at index 2 nothing would change as we would not be able to reach more cells than the ones we can reach from the cell at index 1.

Another way of looking at this problem is by thinking of the cells being divided into levels and to solve this problem we need to apply a BFS visit to the cells. The cells in the interval $[0, I[0]]$ would belong to level 0. Level 1 cells would consist of all cells not in level 0 and that can be reached from any cell of level 0. In general, cells in level i are cells that are not in level $i - 1$ and can be reached by jumping from a cell at level $i - 1$. using this definition is it easier to see how the cell that jumps the farthest at level $i - 1$ would be able

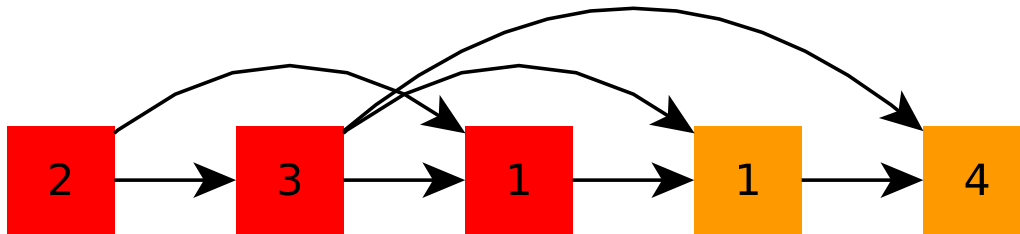


Figure 33.3: An instance of the problem with cells divided by color into their respective levels.

to reach all cells of level i ! There is no other cell in level $i - 1$ from which we can reach more cells at the next level. Therefore, the min number of jumps necessary is equivalent to the level of the last cell.

An implementation of this idea is shown in Listing 33.4.

```

1  int can_jump2_levels(const vector<int>& nums, int pos)
2  {
3      const int last_pos = nums.size() - 1;
4      if (pos == last_pos)
5          return 0;
6      int last_reachable = pos + nums[pos];
7      if (last_reachable >= last_pos)
8          return 1;
9      int next = last_reachable;
10     for (int i = pos + 1; i <= pos + nums[pos]; i++)
11     {
12         if (i + nums[i] > last_reachable)
13         {
14             last_reachable = i + nums[i];
15             next = i;
16         }
17         if (last_reachable >= last_pos)
18         {
19             break;
20         }
21     }
22     return 1 + jumps(nums, next);
23 }
24 int jump(vector<int>& nums)
25 {
26     return jumps(nums, 0);
27 }
```

Listing 33.4: Linear time and space solution.

33.7 Jump Game 3

Problem 49 You are given an array of non-negative integers I of size n representing jump lengths and a start position s . You are initially located at the start index (0) of the array. When you are at index i , you can jump to index $i + I[i]$ or $i - I[i]$.

Write a function that given such an array I and start position s returns true if you

can reach a cell of I containing the value 0 and false otherwise by performing zero or more jump starting from s .

Clearly, you can not jump outside of the array at any time.

■ **Example 33.4**

Given $I = [4, 2, 3, 0, 3, 1, 2]$ and $s = 5$ the function returns **true**. One way to reach cell at index 3 is by starting from index 5, then jump to cell 4, then to cell 1 and finally to cell 3. ■

■ **Example 33.5**

Given $I = [3, 0, 2, 1, 2]$ and $s = 2$ the function returns **false**. There is no combination of jumps you can make that will ever make you land at index 1. ■

33.7.1 Discussion

In the previous variation of this problem discussed in Sections 33.6 and 33.7 we were allowed to jump from index i to **any** cell in the range $[i - I[i], i + I[i]]$, but the variation discussed in this section adds a constraints that forces each jump to be in either of the following two locations:

- $i - I[i]$
- $i + I[i]$

. Another difference is that our target destination is not the end of the array and we will be happy to land in any cell containing a 0.

In our opinion these constraints do not add significant complexity to the problem as at a closer look, like for the other variation, we are dealing with a graph problem where we are asked to check whether we can reach a certain node. In general, to be able to reach a node v from another node u , we need for v and u to be connected: there must be a path you can take (a series of jumps in this case) that take you from node v to node u . This condition is easily checkable by performing a DFS or BFS from node u . For this problem, we are not really interested in a particular node v , and provided it contains the value 0 we are happy.

Therefore we can reframe the problem and ask ourselves whether exist a node with value 0 in it that is connected to node s . This problem is not particularly difficult as all is necessary is to start a visit of the graph from the node s and stop as soon as either of the following is true:

- we have landed to a node with value 0;
- we have visited every node reachable from s (every node in the connected component of s).

If the first condition is true, then we can stop the visit and return true as we have indeed managed to find a way to jump from s to a node with value 0 in it; otherwise, we can return false, because we have visited every possible node reachable from s , but none of them is of the type we want.

Listing 33.5 shows a possible implementation of this idea.

```
1 bool can_jump3_DFS(const std::vector<int>& I, const int s)
2 {
3     const auto size = I.size();
4     std::vector<bool> visited(size, false);
5     std::stack<int> S;
6     S.push(s); // start the visit from s
7     while (!S.empty())
8     {
9         const auto curr = S.top();
```

```

10     S.pop();
11     if (I[curr] == 0)
12     {
13         return true; // we have reached a node of the type we want
14     }
15     visited[curr] = true;
16     if (const auto right = curr + I[curr]; right < size && !visited[right])
17     {
18         S.push(right);
19     }
20     if (const auto left = curr - I[curr]; left >= 0 && !visited[left])
21     {
22         S.push(left);
23     }
24 }
25 return false;
26 }

```

Listing 33.5: DFS solution.

The code is nothing more than a simple DFS implementation on the implicit $|I|$ nodes and edges defined by the content of I and the jump rules of the problem statement (from each node i there are at most two edges to nodes at indices $i - I[i]$ and $i + I[i]$, respectively). Notice that the implicit graph is not guaranteed to be acyclic and some care needs to be taken in order not to visit the same node twice. This is taken care of by the `visited` vector of bools, where we store the information on whether a node has been visited already.

The time complexity of Listing 33.5 is $O(|I|)$ (the number of edges in the implicit graph is also proportional to $|I|$ as each node can have at most two outgoing edges). Its space complexity is likewise linear in the size of I .

Finally, notice that if we knew already that a cell in I can ever have a certain value (for instance because we are told that each element of the input array is non-negative) then we could use I itself to mark a cell as visited, thus lowering to space complexity to constant.

33.8 Jump game 4

Problem 50 Given an array of integers I , you are initially positioned at the first index of the array. In one step you can jump from index i to index:

- $i + 1$;
- $i - 1$;
- any other index $j \neq i$ s.t. $I[i] == I[j]$

Clearly, you can not jump outside of the array at any time.

■ Example 33.6

Given $I = [100, -23, -23, 404, 100, 23, 23, 23, 3, 404]$ the function returns **3**. You can jump from index 0 to 4, then jump to 3 and finally to cell at index 9 which is the last cell of the array. ■

■ Example 33.7

Given $I = [7]$ the function returns **0**. We are already at the last cell of the array. ■

■ Example 33.8

Given $I = [7, 6, 9, 6, 9, 6, 9, 7]$ the function returns **1**. You can jump directly to the last cell. ■

■ Example 33.9

Given $I = [6, 1, 9]$ the function returns **2**. ■

■ **Example 33.10**

Given $I = [11, 22, 7, 7, 7, 7, 7, 7, 22, 13]$ the function returns **3**. ■

33.9 Discussion

This variation of the problem differs from the others because now the content of a cell in I does not tell us how far we can jump. From a given cell i we can jump now to any other cell in I , irrespective of the jump we have to take to reach it, provided that the landing and start cells have the very same value.

The goal is to tell how many jumps at minimum we need to make to reach the end of the array.

At first, this problem looks quite different than the one discussed in Section 33.6 (the other of this series where we needed to return the minimum number of steps to reach the last cell). But is it really that different? The answer to this question is: not really. The idea of visiting the array in a per-level fashion is still valid.

The idea behind the solution presented here is that we will perform a normal BFS visit on a graph having as nodes the cells of I and an edge between each cell at index i and both $i - 1$ and $i + 1$. In addition to these edges we have also edges going from i to any other cell j in the array where $i \neq j$ and $I[i] = I[j]$.

Whenever we visit a node with a certain value x we know we will also have to visit all the indices with the same value and it is, therefore, useful to store the indices of those cells sharing the value in a map having as key the a value and as value a list of indices.

Using this map we can perform a BFS by starting from the index 0. Whenever we are visiting a node i we need to make sure we will visit its immediate neighbors and also all the other indices of the array that have value $I[i]$.

An implementation of this approach is shown in Listing 33.6.

```
1 using Index          = size_t;
2 using EqualIndicesMap = std::unordered_map<int, std::vector<Index>>>;
3
4 EqualIndicesMap build_equal_map(const std::vector<int>& I)
5 {
6     EqualIndicesMap ans;
7     for (size_t i = 0; i < I.size(); i++)
8     {
9         if (ans.find(I[i]) == ans.end())
10         {
11             ans.insert({I[i], {i}});
12         }
13         else
14         {
15             ans[I[i]].push_back(i);
16         }
17     }
18     return ans;
19 }
20
21 int can_jump4(const std::vector<int>& I)
22 {
23     const auto size = I.size();
24     EqualIndicesMap equals_map = build_equal_map(I);
25     std::unordered_set<int> added;
26 }
```

```

27 std::queue<std::pair<int, int>> S;
28 S.push({0, 0});
29 while (!S.empty())
30 {
31     const auto [idx, level] = S.front();
32     if (idx == size - 1)
33         return level;
34     S.pop();
35     if (added.find(I[idx]) == added.end())
36     {
37         for (const auto i : equals_map[I[idx]])
38         {
39             if (i != idx)
40             {
41                 S.push({i, level + 1});
42             }
43         }
44     }
45     added.insert(I[idx]);
46     if (const auto next = idx + 1;
47         next < size && added.find(I[next]) == added.end())
48     {
49         S.push({next, level + 1});
50     }
51     if (const auto prev = idx - 1;
52         prev >= 0 && added.find(I[prev]) == added.end())
53     {
54         S.push({prev, level + 1});
55     }
56 }
57
58 return 0;
59 }

```

Listing 33.6: Graph based solution.

The code works by first calling the function `build_equal_map` that is responsible for constructing the map mentioned above where indices of those cells sharing the same value are grouped together.

The rest of the code performs an iterative BFS visit from cell 0 and the interesting bit of code is possible when at line 35 we check whether we have already processed the value of the currently visited cell (`I[curr]`) and if not, we are going to insert in the visit queue all the indices at which we can find such value (in the loop at line 37). The rest of the loop code takes care of inserting the immediate neighbors of cell `curr`: after all those two cells are also neighbors of `curr`.

The time and space complexities of Listing 33.6 are $O(|I|)$ as we never visit the same node twice and in the `EqualIndicesMap` will never find the same indices in vectors belonging to two keys.

33.10 Jump Game 5

Problem 51 You are given an array of integers I and an integer d . In one step you can jump from index i to index:

- $i + x$ where: $i + x < |I|$ and $0 < x \leq d$.
- $i - x$ where: $i - x \geq 0$ and $0 < x \leq d$.

In addition, you can only jump from index i to index j if and only if $I[i]$ is greater than

$I[k]$ for all indices between i and j ;

Write a function that returns the maximum number of cells you can visit by starting from any cell of I .

Clearly, you can not jump outside of the array at any time.

■ **Example 33.11**

Given $I = [6, 4, 14, 6, 8, 13, 9, 7, 10, 6, 12]$ and $d = 2$ the function returns **4**. One possible jump sequence is the following: $10 \Rightarrow 8 \Rightarrow 6 \Rightarrow 7$

Notice that despite the fact $d = 2$ would allow you in theory to perform a jump of length 2, if you start from the cell at index 6 you can only reach the cell at index 7 as $I[8] > I[6]$. Moreover, because $I[5] > I[6]$, you are also disallowed to reach cell at index 4 despite the fact $I[6] > I[4]$. All cells between the start and end location of a jump must form a strictly decreasing sequence. See Figure 33.4a. ■

■ **Example 33.12**

Given $I = [7, 6, 5, 4, 3, 2, 1]$ and $d = 2$ the function returns **7**. Starting from index 0 you can visit all the nodes by making contiguous jumps of length 1. ■

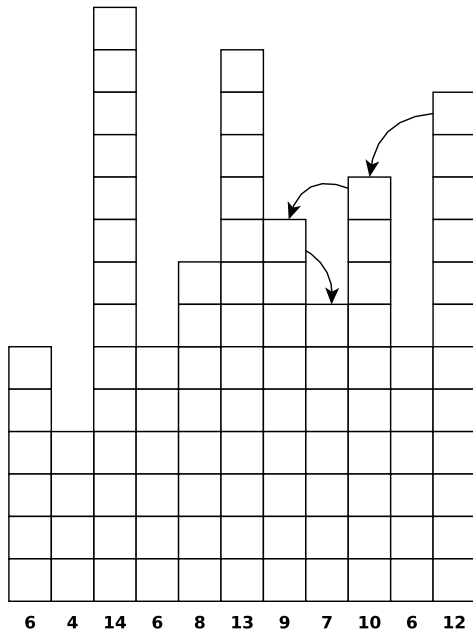
33.11 Discussion

One of the most useful things we can do to solve this variant of the “jump game” problem is (again) to model it as a graph problem. First of all, if we had a graph representation of the problem instance at hand then we would be able to solve this problem by just trying to start a visit from each and every node of the graph and find out which leads to the most visited nodes. This approach is conceptually simple but its efficiency solely depends on and is proportional to the size of such graph. So, how many nodes and edges would this graph have? If $d = O(|I|)$ theoretically, we would be able to reach all nodes by starting from any node. If that is the case then the number of edges would be $O(|I|^2)$ (in general we have $O(|I|d)$ edges). As an example, let’s consider Figure 33.4b which shows how the resulting graph from an instance of this problem which consist of $d = 6$ and $I = \{6, 5, 4, 3, 2, 1\}$ ends up having a quadratic number of edges (a total of $5 + 4 + 3 + 2 + 1$).

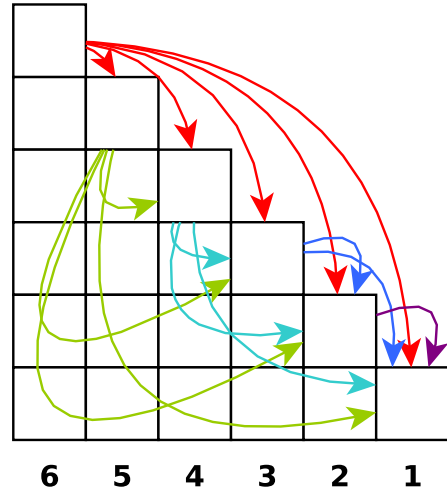
Suppose $R(i)$ is the maximum number of nodes we can visit by starting from node i . We can avoid performing a full-fledged visit to the graph from i twice if we are willing to store values of $R(i)$ somewhere in a cache. When a visit from a node i is complete, we know that we are able to reach $R(i)$ nodes and, if we save this value somewhere, then next time we need the count of nodes we can reach from i , we are not going to perform a complete visit of the graph starting from i but instead, we return the cached value. Because there only so many nodes the size of this cache will never exceed $|I|$.

Listing 33.7 shows a recursive implementation of this idea.

```
1 using Index = size_t;
2 using Graph = std::unordered_map<Index, std::vector<Index>>>;
3
4 Graph build_graph(vector<int>& arr, int d)
5 {
6     Graph ans;
7     for (int i = 0; i < arr.size(); i++)
8     {
9         int curr = arr[i];
10        const int left = (i >= d) ? i - d : 0;
11        for (int j = i - 1; j >= left; j--)
12        {
13            auto next = arr[j];
```

(a) Visual representation of the problem instance of Example 33.11. The arrow represent the jumps one can make from the last cell.



(b) Possible jumps in an instance with $d = 6$. Each color groups jumps allowed from a given cell.

```

14     if (curr <= next)
15         break;
16
17     ans[i].push_back(j);
18 }
19
20 for (int j = i + 1; j < arr.size() && j <= i + d; j++)
21 {
22     auto next = arr[j];
23     if (curr <= next)
24         break;
25     ans[i].push_back(j);
26 }
27 }
28 return ans;
29 }
30
31 using Cache = std::unordered_map<Index, int>;
32 int count = 0;
33 int visit(const Graph& g, const size_t pos, Cache& c)
34 {
35     if (g.find(pos) == g.end())
36         return 0;
37     if (c.find(pos) != c.end())
38         return c[pos];
39
40     int ans = 0;
41     const auto& neighbors = (g.find(pos))->second;
42     for (const auto& n : neighbors)
43     {

```

```

44     count++;
45     ans = std::max(ans, 1 + visit(g, n, c));
46 }
47 c[pos] = ans;
48 return ans;
49 }
50
51 int maxJumps(vector<int>& arr, int d)
52 {
53     const auto g = build_graph(arr, d);
54
55     Cache c;
56     int ans = 0;
57     for (size_t i = 0; i < arr.size(); i++)
58     {
59         ans = std::max(ans, visit(g, i, c));
60     }
61     std::cout << count << std::endl;
62     return 1 + ans;
63 }

```

Listing 33.7: Recursive graph based solution using memoization to avoid performing full-fledged visit for already visited nodes.

The main driver function `max_jumps5__memoized` is responsible for starting a visit from each and every node and keeping track of the maximum number each visit was able to reach. The function `int visit(const Graph& g, const size_t pos, Cache& c)` is solely responsible for traversing the graph and counting the number of hops we can make. It does so by looping through each and every neighbor `n` of the node `pos` recursively calculating the number of nodes reachable from `n`. If we can reach $R(n)$ nodes from `n` then we can reach $R(pos) = 1 + R(n)$ nodes from `pos`. When the function has finally finished with node `pos`, it saves the answer into a `Cache` (a simple map memoizing calls to `visit`). This allows for the first second `if` statement to immediately stop the recursion if we have already performed a visit from node `pos`.

The time and space complexities of this approach are $O(|I|d)$ and $O(|I|)$, respectively. From each node i (out of the $|I|$ nodes) we have to potentially issue $O(|I|)$ calls to the `visit` function (one for each neighbor; these calls return immediately since the moment the cache is full).

34. k^{th} largest in a stream

Introduction

This chapter deals with a problem where the input data is not statically provided all at once but is instead given as a continuous stream of data. These kind of algorithms are common in real life and they are becoming increasingly important in fields like medicine (where data from wearable devices is used to provide real-time insights on the patient's health conditions), or finance where an enormous amount of data (usually provided from the stock exchanges) is used to perform high-frequency trading. We are going to study a coding interview question that has been popular during the last few years and that asks you to design a data structure that is able to deal with a stream of integers and can keep track of the k^{th} largest element seen so far. We will present and discuss three solution based on the same fundamental idea (discussed in Section 34.3) but which are built around three different data structures:

1. a simple array (in Section 34.3.1), 2. a self balancing binary search tree (in Section 34.3.2) and, finally 3. a heap (in Section ??)..

34.1 Problem statement

Problem 52 Design a class which task is to accept a stream of integers one number at the time and to return the k^{th} largest integer seen so far in the stream. The class has two public functions having the following signatures:

1. `void initialize(const std::vector<int>& I, const unsigned K)`: the array `I` contains the first elements of the stream. This function will be called only once upon initialization.
2. `int add(int val)`: this function accepts a new element from the stream and returns the k^{th} largest numbers seen so far.

Note that you can assume that the function `void initialize(const std::vector<int>& I, const unsigned K)` is called only once before any call to the function `int add(int val)`.

■ Example 34.1

Given $K = 4$ and the initial array $I = \{1, 2, 3\}$, the function `int add(int)` behaves as follows:

- `add(4)` returns 1
- `add(4)` returns 1
- `add(0)` returns 1
- `add(2)` returns 2
- `add(200)` returns 2

■

■ Example 34.2

Given $K = 4$ and the initial array $I\{1, 2, 3, 4, 50, 100, 150, 200\}$ the function `int add(int)` behave as follows:

- `add(20)` returns 50
- `add(250)` returns 100
- `add(50)` returns 100
- `add(110)` returns 110
- `add(180)` returns 150
- `add(500)` returns 180

34.2 Clarification Questions

Q.1. What should the function `add` return if the stream counted less than K elements?

You can assume that the largest k^{th} elements exists when `add` is called.

Q.2. Is there a limit to the size of the array I ?

No.

34.3 Discussion

There are two phases associated with this class:

1. the initialization phase where an initial input array I is provided to the class. Because when calling `add` the k^{th} largest value exists, then we can deduce that the size of the vector I is at least $K - 1$ otherwise the first call of `add` could not possibly return the correct value. This operations is guaranteed to happen one time only before any call to `add`.
2. the stream elements processing phase where the class is ready to accept a new number from the stream and return the answer.

The key to attacking this problem is to understand that during the initialization phase when the initialization array comes in, we are forced to remember the largest elements within it. In particular, if $|I| \geq K$ then we can throw away all the elements that are not among the K largest and keep the rest (this should be self-explanatory as those elements will never be used as a return value of `add` as there are already K values larger than all of them), otherwise we can remember I as it is (and in this case we know that $|I| = K - 1$). One might think that it isn't necessary to remember all K largest numbers seen so far and that it is in fact only necessary to remember the K^{th} largest element. We will use Example 34.2 as a simple counterexample to demonstrate why this leads to incorrect results. First of all, after the initialization phase the 4th largest number ($K = 4$ in this example) is 50. Then, after the call to `add(20)` the 4th largest number is not changed and the function still returns 50. But when `add(250)` is called, then 50 suddenly becomes the 5th largest number and it is at this point that remembering the other numbers larger than 50 becomes important. Without them we would not now be able to return the correct value i.e. 100.

In short, in order to be able to always give an answer we need to store and keep track of all the K largest numbers seen so far. This naturally leads to the question of where and how we can actually do that? Let's name the set of the largest K numbers seen so far L^K . Moreover let m be the smallest element in L^K ; $m = \min(L^K)$. When a new number n arrives, we can do one of the following operations depending on its value:

- if $|L^K| < K$ we simply insert n in L^K and return.
- otherwise, if $n \leq m$ then, n can be ignored as it has no influence among the elements of L^K .

- otherwise ($n > m$), m can be safely removed from L^K as it would become the $K - 1^{th}$ largest after the addition of n . The new element n can be inserted in L^K .

Note that the size of L^K never changes after it reaches K . The way we decide to store L^K has a big influence on the cost of the operations listed above, namely: 1. find the minimum element (m) 2. remove the minimum element (m) 3. insert a new element (n). In the following Section we will investigate three data structures that can be used to hold the values of L^K .

34.3.1 Array based solution

In this section we present a solution where the elements of L^K are stored in a sorted array. We will see that this is probably not the best idea as, when a new element n arrives the whole array needs to be rearranged and that can be a costly operation. In particular let's have a look at how both the `initialize` and `add` function can be implemented:

initialize(I, K): The initialization phase has to filter the largest K elements out of the initialization array I . This can be done by sorting I in ascending order first and then copying in L^K only the first of its K elements (the K largest). The same outcome can be obtained by using a partial sort algorithm which makes sure that the largest K elements are at the front of the array but given no guarantees on the relative ordering of the rest of the array. The complexity of this operation is $O(|I|\log(|I|))$ if the whole array I is sorted but when a partial sort algorithm is used then the costs becomes $O(|I|\log(K))$.

add(n): When $|L^K| < K$ or n is inserted in L^K . But if $n > m$ then m is substituted by n (thus effectively removing m) and subsequently the ordering of L^K is restored. When this happens we will be in a situation where the L^K is sorted except for n which might not be in the right position (for instance when n would be the largest element of L^K). So restoring the order on L^K can be achieved either by:

- fully sorting L^K . In this case the complexity of `add` is $O(K\log(K))$
- by moving the newly inserted element from the first location of the array up, by swapping it with its subsequent element, until it reaches the correct position. This operation is analogous to the way the insertion sort algorithm operates. The cost of this operation is $O(K)$. In the worst case scenario we need to bubble up n up to the last cell of the array (when n is effectively the largest element of L^K) by performing $K - 1$ swap operations.

Listing 34.1 shows an implementation where the initialization phase is performed using a normal sort and the `add` is implemented by using an approach á la insertion-sort.

```

1  class KthLargestInStreamArray : IKThLargestInStream
2  {
3  public:
4      KthLargestInStreamArray() = default;
5      void initialize(const std::vector<int>& initArray, const size_t K) override
6      {
7          m_values = initArray;
8          m_k      = K;
9          std::sort(begin(m_values), end(m_values));
10         const auto start = begin(m_values);
11         const auto end   = m_values.size() >= K
12                             ? m_values.begin() + (m_values.size() - K)
13                             : m_values.begin();
14         m_values.erase(start, end);
15     };
16
17     int add(const int n) override

```

```

18 {
19     assert(m_values.size() == m_k);
20     if (n < m_values.front())
21         return m_values.front();
22     m_values.front() = n;
23     auto it          = begin(m_values);
24     auto itn         = std::next(it);
25     while (itn != end(m_values) && *it > *itn)
26     {
27         std::iter_swap(it, itn);
28         it = itn;
29         itn = std::next(it);
30     }
31     return m_values.front();
32 };
33
34 private:
35     std::vector<int> m_values;
36     size_t m_k = 0;
37 };

```

Listing 34.1: Solution to the k^{th} largest element in a stream problem using arrays

34.3.2 Ordered set

If instead of an array we use a data structure ^①that allows us to perform insert/search and min operations runs in \log and constant time, respectively, then we can substantially improve the time complexity of the solution shown in Section 34.3.1.

Let's have a more detailed look at what the two operations would look like in this case: **initialize(I, K)**: Recall that the final goal of this operation is to keep only the largest K elements of I . This can be easily achieved in $O(|I|\log(K))$ by looking at each element of I , say I_j individually, and inserting it into L^K if:

- the size of $|L^K| < K$ or
- if I_j is greater than the smallest element of L^K . Additionally in this case, if after the insertion we have that $|L^K| > K$ then the current smallest element in L^K is removed to make sure that $|L^K| = K$.

Because there are $O(|I|)$ elements that can potentially go in L^K and each insertion costs $O(\log(K))$ then the final complexity is $O(|I|\log(K))$. Note that it is not much better than the array solution in this case.

add(n): This is where we see the advantages of having the elements of L^K not stored in a plain array. Like in the array solution, we compare n with the smallest element in L^K , m , and insert n in L^K only if $n > m$. But because the ordered multiset supports the `insert` and `min` operations in $O(\log)$ and $O(1)$ time, respectively, then the complexity of this operation is $O(\log(K))$. Quite an improvement compared with the array solution.

Listing 34.2 shows a possible implementation of this idea using a ordered multiset (we use the C++ STL implementation named `std::multiset`).

```

1
2 #include "IKThLargestInStream.h"
3
4 class KthLargestInStreamMap : IKThLargestInStream
5 {

```

^①Example of such data structures are: 1. ordered multiset (for instance implemented as a self-balancing binary search tree) 2. heap or priority queue.

```

6 public:
7   KthLargestInStreamMap() = default;
8   void initialize(const std::vector<int>& initArray, const size_t K) override
9   {
10    assert(K <= initArray.size());
11    m_k = K;
12    m_values.insert(begin(initArray), end(initArray));
13
14    auto it = begin(m_values);
15    while (m_values.size() > K)
16    {
17      it = m_values.erase(it);
18    }
19    assert(m_k == K);
20    assert(K == m_values.size());
21  }
22  int add(const int n) override
23  {
24    assert(m_k == m_values.size());
25
26    if (n > *(m_values.begin()))
27    {
28      m_values.insert(n);
29      m_values.erase(m_values.begin());
30    }
31    assert(m_k == m_values.size());
32    return *(m_values.begin());
33  }
34
35 private:
36   std::multiset<int> m_values;
37   size_t m_k;
38 };

```

Listing 34.2: Solution to the k^{th} largest element in a stream problem using `std::multiset`

In Listing 62.1 you can see an implementation using a heap instead. Note that there is no dedicated class in C++ for heaps but instead, we can use an array as a container for the elements and then manipulate it by using the heap dedicated functions:

- `make_heap`, to arrange the elements of an array into a heap
- `push_heap`, to add an element to an array assembled by `make_heap`
- `pop_heap`, to remove the smallest element from the heap.

Also note that Listing 34.3 uses a slightly different strategy for implementing the two class functions. Specifically

initialize(I, K): We insert the first K elements of I into an array that we immediately turn into a heap (using `make_heap`). At this point we call `add(n)` for all the remaining elements of I .

add(n): as for the other solution when $n < m$ the function simply returns the smallest element of L^K . On the other hand when this is not the case, it removes the smallest element of the heap by calling `pop_heap`^② and inserts n by using `push_heap`^③.

```

1 #include "IKThLargestInStream.h"
2
3 class KthLargestInStreamHeap : IKThLargestInStream

```

^② `pop_heap` does not really erase anything from the heap. It moves the head of the heap to the end of the array and rearranges all the elements from the begin of the array to the one before the last into a valid heap thus effectively reducing the size of the heap by one.

^③ `push_heap` expects the elements to be inserted into the heap to be placed at the very end of the array.

```

4 {
5 public:
6   KthLargestInStreamHeap() = default;
7   void initialize(const std::vector<int>& initArray, const size_t K) override
8   {
9       assert(K <= initArray.size());
10      m_k = K;
11
12      auto s = begin(initArray);
13      auto e = begin(initArray) + K;
14      m_values_heap.insert(begin(m_values_heap), s, e);
15      std::make_heap(begin(m_values_heap), end(m_values_heap), std::greater<>());
16
17      assert(K == m_values_heap.size());
18
19      while (e != end(initArray))
20      {
21          add(*e);
22          ++e;
23      }
24      assert(K == m_values_heap.size());
25  }
26
27  int add(const int n) override
28  {
29      assert(m_k == m_values_heap.size());
30      if (n <= m_values_heap.front())
31          return m_values_heap.front();
32      std::pop_heap(begin(m_values_heap), end(m_values_heap), std::greater<>());
33      m_values_heap.back() = n;
34      std::push_heap(begin(m_values_heap), end(m_values_heap), std::greater<>());
35
36      assert(m_k == m_values_heap.size());
37      return m_values_heap.front();
38  }
39
40 private:
41     std::vector<int> m_values_heap;
42     size_t m_k;
43 };

```

Listing 34.3: Solution to the k^{th} largest element in a stream problem using a heap

35. Find the K closest elements

Introduction

In this chapter we will discuss a problem that asks you to return a subset of a given input array. We will investigate two solutions: one that is based on sorting and the other on binary search with the latter being more efficient as we will make good use of the fact that the input is provided already sorted. By contrast, the solution based on sorting appears almost trivial to and we can derive it directly from the problem statement while the solution based on binary search requires slightly more insight and typing to get right. We will present two different implementations of the binary search solution: 1. the first based entirely the C++ STL, 2. and the other where we will code the binary search algorithm explicitly.

35.1 Problem statement

Problem 53 Write a function that takes as input • a sorted array I and two integers • k and • x , and returns an array, sorted in ascending order, containing the k elements that are closest to x in I . Note that: given two elements y , and z , y is closer to x than z if:

$$|x - y| < |x - z| \quad (35.1)$$

■ **Example 35.1**

Given • $I = \{1, 2, 3, 4, 5\}$, • $k = 4$ and • $x = 3$, the function returns: $\{2, 3, 4, 5\}$ ■

■ **Example 35.2**

Given • $I = \{1, 2, 3, 4, 5\}$, • $k = 4$ and • $x = -1$, the function returns: $\{1, 2, 3, 4\}$ ■

■ **Example 35.3**

Given • $I = \{12, 16, 26, 30, 35, 39, 42, 46, 48, 50, 53, 55, 56\}$, • $k = 5$ and • $x = 36$, the function returns: $\{26, 30, 35, 39, 42\}$ ■

35.2 Clarification Questions

Q.1.

Q.2. What should the function behavior be when resolving ties? What should it do when you have two elements that are at the same distance from x ?

The function should always favor the smaller element in case of a tie.

Q.3. Is I guaranteed to be sorted in ascending order?

Yes you can assume I is always sorted in ascending order.

35.3 Sorting

A solution that almost immediately follows from the problem statement is based on sorting the elements of I according to the criteria shown in Equation 35.1. The idea is that if I is sorted according to the absolute value of the difference between each number of I and x then the closest number to x will be located after the sorting at the front of I . All that is necessary at that point is to copy the first K element of I into the return array. Listing 35.1 shows a possible implementation of this idea.

```
1  std::vector<int> kth_closest_in_array_sorting(vector<int>& I,
2                                              const int k,
3                                              const int x)
4  {
5      assert(I.size() >= k);
6      std::sort(begin(I), end(I), [x](const auto y, const auto z) {
7          return std::abs(x - y) < std::abs(x - z);
8      });
9
10     std::vector<int> ans{begin(I), begin(I) + k};
11     std::sort(begin(ans), end(ans));
12     return ans;
13 }
```

Listing 35.1: Solution to the problem of finding the k closest element using sorting.

Please note that - as in all cases where you actually do not need to have the whole array sorted - you can use partial sorting instead of fully-fledged sorting. In all cases where k is smaller than n the complexity is going to be slightly better as we will go from the $O(n\log(n))$ of the normal sorting to $O(n\log(k))$ of the partial sort. Fortunately, making this change in C++ is easily as it is only a matter of calling `std::partial_sort` instead of `std::sort` as shown in Listing 35.2.

```
1  std::vector<int> kth_closest_in_array_partial_sorting(vector<int>& I,
2                                              const int k,
3                                              const int x)
4  {
5      assert(I.size() >= k);
6      std::partial_sort(
7          begin(I), begin(I) + k, end(I), [x](const auto y, const auto z) {
8              return std::abs(x - y) < std::abs(x - z);
9          });
10
11     std::vector<int> ans{begin(I), begin(I) + k};
12     std::sort(begin(ans), end(ans));
13     return ans;
14 }
```

Listing 35.2: Solution to the problem of finding the k closest element using sorting.

35.3.1 Binary Search

The problem description clearly states that the input array is sorted but the solution we devised in Section 35.3 does not take advantage of that fact. All it does is invalidate the original ordering in order to enforce a different one. Every time the problem statement mentions that some input is sorted, we should think how to use that to devise a more efficient solution rather than questioning whether that information is useful or not. It is always wise to assume that there will be no useless information in the problem statement. When sorted input is involved, there are several algorithms that should immediately come

to mind and, out of this set, binary search is probably first. All that is left is to ask ourselves how binary search could be applied to this problem?

But first, let's take a step back and try to analyze the problem for a slightly different angle. Specifically, let's discuss the case where $x \in I$. In this case we know for sure that x is going to be part of the output vector. As the input is sorted we can use binary search to search for x in I . Once we have identified the index j such that $I_j = x$ we know that the closest element to x must either be at index $j + 1$ or $j - 1$ ^①. Therefore once x has been identified we can select a range of k elements "centered" at j . Said range can be found by using a two pointers technique. We start by initializing two pointers $l = j$ and $r = j$. Then until $r - l + 1 < k$ we do one of the following operations:

- if $l = 0$ then $r = r + 1$
- if $r = |I|$ then $l = l - 1$
- if $x - I_{l-1} > I_{r+1} - x$ then $r = r + 1$. The range is enlarged at its right end side.
- symmetrically for the left side: if $x - I_{l-1} > I_{r+1} - x$ then $l = l + 1$. The range is enlarged at its left end side.

In other words once x has been found, we incrementally include elements around it by always choosing between the closest numbers to x between the numbers pointed by the two pointers.

This approach can easily be extended to the case where x is not present in the input array as it also works when we try to build the range of elements to be returned around the closest element to x in the array. Turns out that binary search can be used to find such an element (we even have STL support for such an operation). In particular we can use it to identify the index of the first element that is larger or equal than x : a value that is commonly known as *lower bound*. Armed with this information let's have a look at Listing 35.3 showing the implementation of the idea above where we use the STL `std::lower_bound` function to find the index o of the first element greater or equal than x . We then compare such value with the value at index $p = o - 1$ (if it exists) and we promote o or p to be the index closest element to x in the array depending on their absolute difference to x . The closest of the two to x is chosen to be the designated starting value for the algorithm described above.

```

1  template <typename Iterator>
2  auto find_range(
3      Iterator begin, Iterator end, Iterator closest, int k, const int x)
4  {
5      assert(begin < end);
6      assert(closest >= begin);
7      assert(closest < end);
8
9      auto l = closest;
10     auto r = l + 1;
11     auto difference = [](const auto x, const auto y) { return std::abs(x - y); };
12
13     k--; // closest is already included in the range
14     while (k && l > begin && r < end)
15     {
16         if (difference(*(l - 1), x) <= difference(*r, x))

```

^①If that was not true it would mean that:

- $I_k = I_{j+1}$. In this case, picking K would not be an improvement to $j - 1$ or $j + 1$.
- otherwise either:
 - $\exists k < j - 1 : I_k < I_{j-1}$ and $x - I_k < x - I_{j-1}$ or
 - $\exists k > j + 1 : I_k > I_{j+1}$ and $I_k - x < I_{j+1} - x$
 which is impossible because the input is sorted.

```

17     {
18         l--;
19     }
20     else
21     {
22         r++;
23     }
24     k--;
25 }
26 while (k && l > begin)
27 {
28     l--;
29     k--;
30 }
31 while (k && r < end)
32 {
33     r++;
34     k--;
35 }
36 assert(k == 0);
37
38 return std::make_tuple(l, r);
39 }
40 std::vector<int> kth_closest_in_array_binary_search_lower_bound(
41     const vector<int>& I, const int k, const int x)
42 {
43     auto closest = std::lower_bound(begin(I), end(I), x);
44     if (auto prec = std::prev(closest);
45         closest != begin(I) && closest != end(I)
46         && (std::abs(*closest - x) >= std::abs(*prec - x)))
47     {
48         closest = prec;
49     }
50     // if no element is larger than x
51     // then the closest to it is the largest in I
52     if (closest == end(I))
53         closest = std::prev(end(I));
54
55     auto [l, r] = find_range(begin(I), end(I), closest, k, x);
56     return std::vector<int>(l, r);
57 }

```

Listing 35.3: Solution to the problem of finding the k closest element using `std::lower_bound`.

For completeness, in Listing 35.4 we also show an implementation of “in-house” version of `std::lower_bound`. You might be asked to show you can code binary search.

```

1
2 template <typename It, typename Target = typename It::value_type>
3 It my_lower_bound(const It& begin, const It& end, const Target& target)
4 {
5     auto l = begin;
6     auto r = end;
7     while (l < r)
8     {
9         auto mid = l + std::distance(l, r) / 2;
10
11         if (*mid < target)
12         {
13             l = mid + 1;
14         }

```

```
15     else
16     {
17         r = mid;
18     }
19 }
20 return 1;
21 }
```

Listing 35.4: Implementation of a function for the calculation of the *lower_bound* that can be using in substitution of `std::lower_bound` in Listing 35.2.

36. Binary Tree mirroring

Introduction

Binary trees are one of the most taught and discussed data structures in computer science courses. A binary tree is a tree-like data structure where each node has at most two children, which we refer to as right and left children. Trees have long been used in computer science as a way to access data stored within nodes that are usually arranged in a particular order intended to make operations like searching for sorting more efficient. Examples of these special types of trees are: • binary search tree, • binary heap

Binary trees are also often used to model data with an inherently bifurcating structure, i.e. where the organization of data into left and right is also part of the information we are representing^①. A tree is a recursive data structure because it can be thought of as either being:

- an empty tree
- or a node having a binary tree as left and right children.

There are many recursive fundamental algorithms on trees described in the literature that build around this definition, and, as such, recursive solutions to questions about trees are an effective tool in coding interviews. The problem discussed in this chapter focuses on the manipulation of a binary tree into another binary tree so that the latter is a mirror image of the former. As we will see, the question is quite vague as it can be unclear what being a mirror actually means in this context so it is important to ask relevant questions really means and perhaps even create a few examples cases (which we provide later in this chapter) that clarify what the interviewer is expecting.

The tree definition that we will use throughout the chapter is shown in Listing 36.1.

```
1 #ifndef TEST_MIRROR_BINARY_TREE_BINARY_TREE
2 #define TEST_MIRROR_BINARY_TREE_BINARY_TREE
3
4 template <typename T>
5 struct Node
6 {
7     Node() = default;
8     Node(const T& val) : payload(val), left(nullptr), right(nullptr){};
9     Node *left = nullptr, *right = nullptr;
10    T payload{};
11 };
12
13 #endif /* TEST_MIRROR_BINARY_TREE_BINARY_TREE */
```

Listing 36.1: Definition of the tree data structure using in Chapter 36.

36.1 Problem statement

^①In such case changing the arrangements of the node would change the meaning of the data.

Problem 54 Write a function that given a binary tree, return a mirror copy of it.

■ **Example 36.1**

Given the binary tree shown in Figure 36.1a the function returns a tree like the one in Figure 36.1b . ■

■ **Example 36.2**

Given the binary tree shown in Figure 36.1c the function returns a tree like the one in Figure 36.1d ■

■ **Example 36.3**

Given the binary tree shown in Figure 36.2a the function returns a tree like the one in Figure 36.2b ■

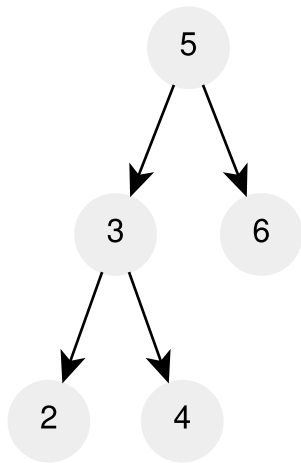
36.2 Discussion

Let's start our discussion by trying to understand what a mirror copy of a tree really looks like. If we have a tree T rooted at node n then its mirror image L can be defined as follows:

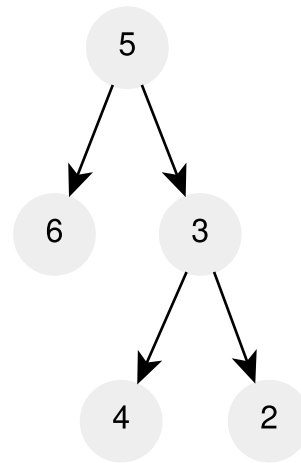
- **if n has no children:** return T . See Figures 36.3a and 36.3b.
- **if n has one only the left child n_l :** return T having as left child the a mirrored copy of n_l . See Figures 36.3e and 36.3f.
- **if n has one only the right child n_r :** return T having as right child the a mirrored copy of n_r . See Figures 36.3c and 36.3d.
- **if n has both children:** return T having as left child the mirrored copy of its right child n_r and, as right child the mirrored copy of its left child n_l . See Figures 36.3g and 36.3h. Another example of this case can be found in the node 5 in the Example 36.1 where its left and right children are first mirrored individually and then swapped.

This recursive definition can be refined into the following simple idea: In order to create the mirror image of a tree T rooted at n we first mirror its children individually and only after do we swap them. Given that we can turn a tree into its mirror image all that is necessary is to first create a copy of the origin tree and then mirror it (remember that the problem is asking to return a copy). Listing 36.2 shows a recursive implementation of this idea.

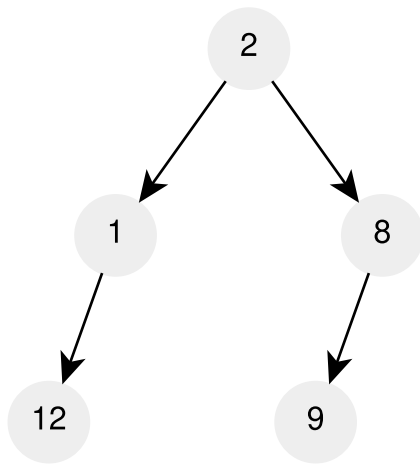
```
1 #include "binary_tree.h"
2
3 template <typename T>
4 Node<T>* copy_binary_tree(const Node<T>* const root)
5 {
6     if (!root)
7         return nullptr;
8
9     auto root_copy = new Node<T>(root->payload);
10    root_copy->left = copy_binary_tree(root->left);
11    root_copy->right = copy_binary_tree(root->right);
12    return root_copy;
13 }
14
15 template <typename T>
16 void mirror_binary_tree_in_place(Node<T>* const node)
17 {
18     using std::swap;
19 }
```



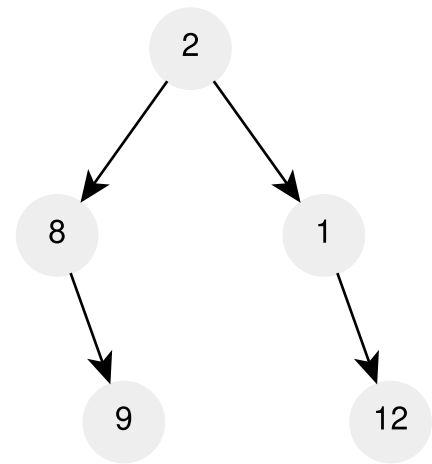
(a) Input binary tree for the Example 36.1.



(b) Output binary tree for the Example 36.1.

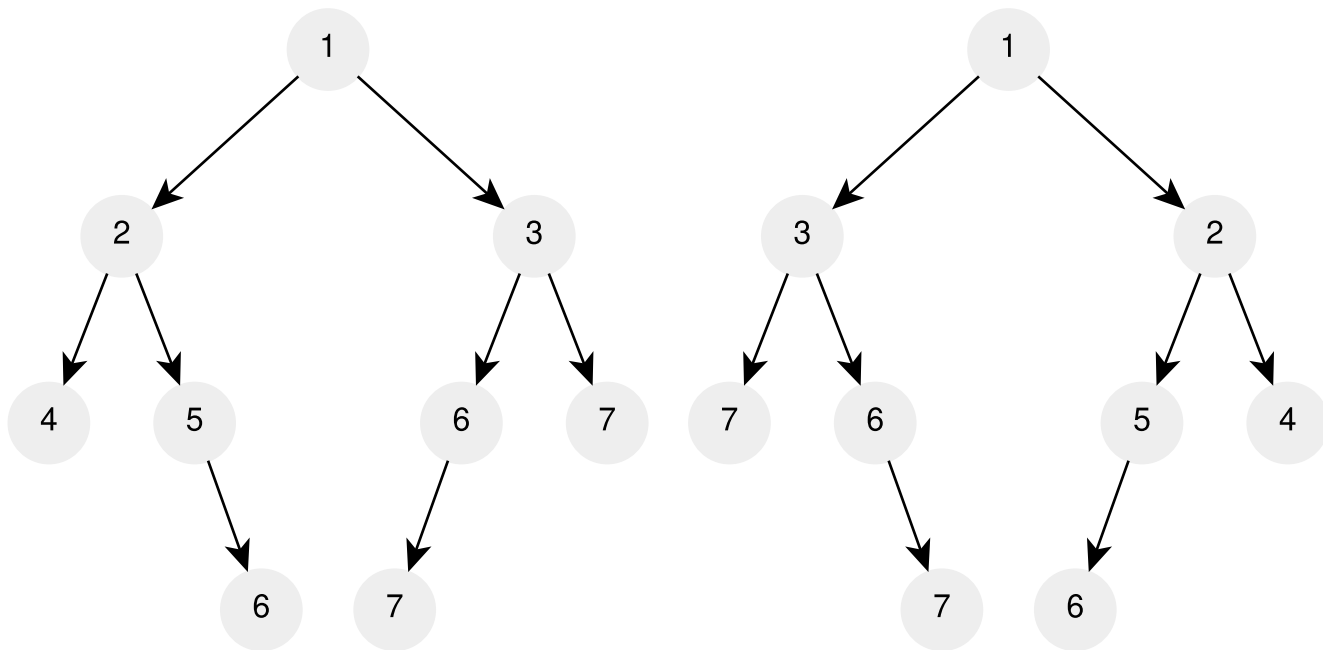


(c) Input binary tree for the Example 36.2.



(d) Output binary tree for the Example 36.2.

Figure 36.1: Input and output for Examples 36.1 and 36.2



(a) Input binary tree for the Example 36.3.

(b) Output binary tree for the Example 36.3.

Figure 36.2: Input and output for Example 36.3

```

20  if (!node)
21      return;
22
23  mirror_binary_tree_in_place(node->left);
24  mirror_binary_tree_in_place(node->right);
25  swap(node->left, node->right);
26  }
27
28  template <typename T>
29  Node<T>* mirror_binary_tree(const Node<T>* const root)
30  {
31      auto&& tree_copy = copy_binary_tree(root);
32      mirror_binary_tree_in_place(tree_copy);
33      return tree_copy;
34  }

```

Listing 36.2: Solution to the problem of creating a mirror of a binary tree. Works by first creating a copy of the original tree and only then performing the mirroring.

The complexity of the code above is $O(n)$ where n is the number of nodes in T . However, despite the fact that splitting the copying and the mirroring steps simplifies the reasoning and the implementation this is not optimal as we need to traverse the whole tree twice. We can create the copy on the fly as we visit T as shown in Listing 36.3. This approach does not lower the asymptotic complexity but it effectively means that we only need to traverse the original tree once instead of twice.

```

1  template <typename T>
2  Node<T>* mirror_binary_tree_on_the_fly(Node<T>* const node)
3  {

```

```

4  if (!node)
5      return nullptr;
6
7  auto root_mirror    = new Node<T>(node->payload);
8  root_mirror->right = mirror_binary_tree_on_the_fly(node->left);
9  root_mirror->left  = mirror_binary_tree_on_the_fly(node->right);
10 return root_mirror;
11 }

```

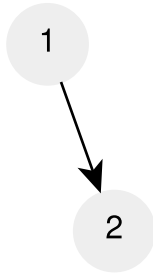
Listing 36.3: Solution to the problem of creating a mirror of a binary tree. The copy and the mirroring are performed simultaneously while visiting T .



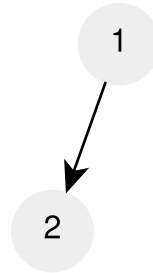
(a) Example of single node tree.



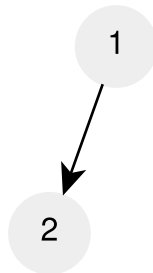
(b) Mirror image of the tree in Figure 36.3a.



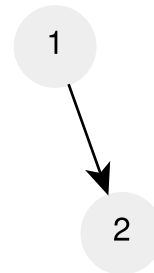
(c) Example of node with a single child: the right one.



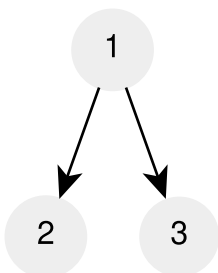
(d) Mirror image of the tree in Figure 36.3c.



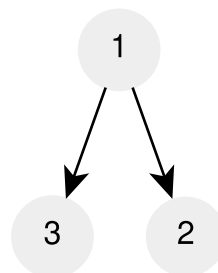
(e) Example of node with a single child: the left one.



(f) Mirror image of the tree in Figure 36.3c.



(g) Example of node with a single child: the left one.



(h) Mirror image of the tree in Figure 36.3g.

Figure 36.3: Examples of various types of binary trees and their associated mirror images.

37. Count the number of islands

In this chapter we will discuss a classic interview problem question which requires that we count the number of islands on a map provided as a 2D boolean matrix. There are many versions of the statements for this problem, some with a wordy and playful background story and others where the map is just given to you as graph of some sort. Thankfully all the versions can be solved with the approaches below which are based on standard graph visiting algorithms.

37.1 Problem statement

Problem 55 Write a function that given a 2D boolean matrix counts the number of islands in the grid. Cells in the input matrix containing a 1 represent land while 0 water. An island is a group of adjacent land cells. Every cell in the input matrix can be adjacent to the 4 cells that are next to it on the same row or column.

The input grid is a 2D `std::vector<std::vector<bool>>` of size $n \times m$.

■ Example 37.1

Given the following input grid (depicted in Figure 37.1a) the function returns 4. The text representation of this example is given below where 1 represents land, and a 0 water:

```
1110000
0100000
0110110
0010110
0100000
0110010
0011000
```

37.2 Clarification Questions

Q.1. Can n or m be 0?

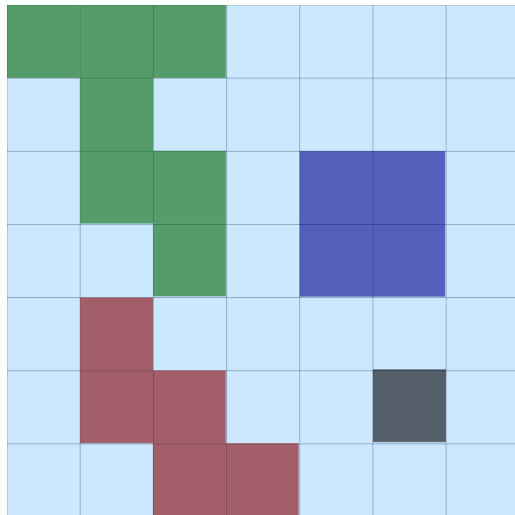
Yes, the map can be empty

Q.2. Can the input grid be modified?

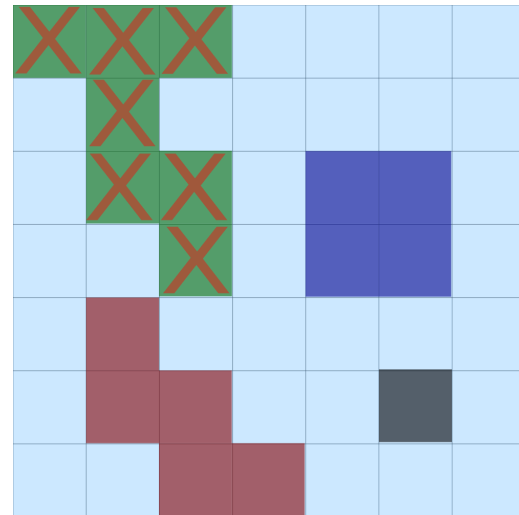
Yes, the input matrix is not read-only.

37.3 Discussion

Essentially, what this problem is asking us to do is identify the number of clusters of 1s in the input matrix. One way to do this is by looping through the map one cell at a time



(a) Visual representation of the example 1 for the problem of counting the number of islands in a map. Cells belonging to the same island share the same color.



(b) Visual representation of the example 1 after visiting the cells of the first island.

until we find a 1, let's say at cell (i, j) . Because this particular 1 must be part of an island, what we can then do is start exploring the island one cell at a time, moving from a 1 to an adjacent one, until there are no 1s we have not already visited. When we visit a land cell we mark it as “visited”. This is useful because when we resume our normal linear scanning of the map we want to make sure we do not count the visited cells as being the starting point of an uncounted island. For instance in the example in Figure 37.1a we can start our visit at cell $(0,0)$ which is a 1 and is not yet visited. This means that this particular cell is part of an island that we did not count yet. At this point we can start visiting the cells adjacent to $(0,0)$ i.e. cells: $(0,0), (0,1), (0,2), (1,1), (2,1), (2,2), (3,2)$. When a cell is visited it is marked as shown in Figure ?? by the red cross \times and after that all of its neighboring land cell are visited similarly in a recursive manner. When we have exhausted all the cells of the island $(0,0)$ is part of we can resume our linear search remembering we have explored an additional island.

The visit can be performed using a BFS or DFS approach. In the following section we will look at a recursive and iterative implementation of the DFS approach. We prefer the DFS approach over BSF mostly because it is easier to code recursively. The iterative version (in Listing 37.3.0.2) can, however, be turned into a BFS quite easily just by changing the policy of the order in which cells are to be visited.

37.3.0.1 DFS iterative

Listing 37.1 shows a possible iterative implementation of the DFS approach as described above. Note that the core of the algorithm is the function `visit` that uses a stack to keep track of the cells that are still to be visited. For each cell that is actually visited we will also try to visit all pieces of yet unvisited land in all four directions (up, down, left and right). We do this by adding them to the pile of cells to be visited. When a cell is actually visited, it is marked as such in its corresponding cell of the array `visited`. When there is no more land left in the stack it means that the island has been completely visited and we can return. Once it is complete its execution function `visit` returns the control back in the double loop of the function `count_island_iterative_DFS` which will skip all the visited cells and will trigger another invocation of `visit` as soon as another unvisited 1

cell is found. That 1 has to be part of a not yet unaccounted island together with all its adjacent land cells.

Also please note how the if at line 22 takes care of not visiting cells that are outside the boundaries of the map, cells that are not land or already visited because this would lead to out-of-bound errors, incorrect results and infinite loops, respectively.

The complexity of this implementation in Listing 37.1 is $O(n \times m)$ for both time and space because we visit the whole map at least once and we use space proportional to $n \times m$ for the array `visited`.

```
1 using cell = std::pair<int, int>;
2 void visit(const cell& c,
3           const std::vector<std::vector<bool>>& grid,
4           std::vector<std::vector<bool>>& visited)
5 {
6     const int n = grid.size();
7     const int m = grid[0].size();
8
9     std::stack<cell> S;
10    S.push(c);
11    while (!S.empty())
12    {
13        auto p = S.top();
14        S.pop();
15
16        const auto [x, y] = p;
17        visited[x][y] = true;
18
19        constexpr std::array<cell, 4> cross = {
20            cell{-1, 0}, cell{1, 0}, cell{0, -1}, cell{0, 1}};
21        for (const auto& inc : cross)
22        {
23            const auto nx = x + inc.first;
24            const auto ny = y + inc.second;
25            if (nx >= 0 && nx < n && ny >= 0 && ny < m && grid[nx][ny]
26                && !visited[nx][ny])
27            {
28                S.push({nx, ny});
29            }
30        }
31    }
32 }
33
34 int count_island_iterative_DFS(const std::vector<std::vector<bool>>& grid)
35 {
36     if (grid.size() == 0 || grid[0].size() == 0)
37         return 0;
38
39     const int n = grid.size();
40     const int m = grid[0].size();
41     int ans = 0;
42
43     std::vector<std::vector<bool>> visited(n, std::vector<bool>(m, false));
44     // search for a piece of unvisited land
45     for (int i = 0; i < n; i++)
46     {
47         for (int j = 0; j < m; j++)
48         {
49             if (!grid[i][j] || visited[i][j])
50                 continue;
51             // found one, mark as visited all the piece of
```

```

52     // land you can reach from here
53     ans++;
54     visit({i, j}, grid, visited);
55 }
56 }
57 return ans;
58 }

```

Listing 37.1: Iterative DFS solution to the problem of counting the number of islands in a map.

Do we, however, really need to have a dedicated matrix just to store the information about which cell is visited? In fact no, as we can store that information in-place in the input matrix. All we have to do when marking a cell visited is turn the value in the input grid (which is modifiable) for that cell from land (1) to water (0) meaning that cell can never be considered as part of an island in the future. If we do that, the space complexity does not change because we still use space to store the cells to be visited in the stack, but the amount of space used will be lower in practice and the overall solution will look cleaner and simpler which is always a plus during an interview.

```

1  using cell = std::pair<int, int>;
2
3  void visit(const cell& c, std::vector<std::vector<bool>>& grid)
4  {
5      const int n = grid.size();
6      const int m = grid[0].size();
7
8      std::stack<cell> S;
9      S.push(c);
10     while (!S.empty())
11     {
12         auto p = S.top();
13         S.pop();
14
15         const auto [x, y] = p;
16         grid[x][y] = false; // mark the original map
17
18         constexpr std::array<cell, 4> cross = {
19             cell{-1, 0}, cell{1, 0}, cell{0, -1}, cell{0, 1}};
20         for (const auto& inc : cross)
21         {
22             const auto nx = x + inc.first;
23             const auto ny = y + inc.second;
24             if (nx >= 0 && nx < n && ny >= 0 && ny < m && grid[nx][ny])
25             {
26                 S.push({nx, ny});
27             }
28         } // for
29     } // while
30 }
31
32 int count_island_iterative_DFS_improved(std::vector<std::vector<bool>>& grid)
33 {
34     if (grid.size() == 0 || grid[0].size() == 0)
35         return 0;
36
37     const int n = grid.size();
38     const int m = grid[0].size();
39     int ans = 0;
40

```

```

41 for (int i = 0; i < n; i++)
42 {
43     for (int j = 0; j < m; j++)
44     {
45         // visited cells are turned into water during the visit
46         if (!grid[i][j])
47             continue;
48         ans++;
49         visit({i, j}, grid);
50     }
51 }
52 return ans;
53 }

```

Listing 37.2: Alternative iterative DFS solution, without dedicated space for marking visited cells, to the problem of counting the number of islands in a map.

37.3.0.2 DFS recursive

The same idea can be implemented recursively which, in our opinion, makes the overall implementation more expressive, shorter and easier to reason about as well as to explain. As such, our first choice is always to use this approach if possible when dealing with problems similar to the one presented here. Listing 37.3 shows a possible implementation of a recursive DFS solution for this problem. Note that the recursion only happens during the DFS itself and that the driver function `count_island_recursive_DFS` is basically the same as the ones shown in the previous two solutions 37.1 and 37.2.

```

1 using cell = std::pair<int, int>;
2 void visit_recursive(const cell& c, std::vector<std::vector<bool>>& grid)
3 {
4     const int n = grid.size();
5     const int m = grid[0].size();
6
7     const auto [x, y] = c;
8     // base case: a cell is out of the map or already visited or water
9     if (x < 0 || y < 0 || x >= n || y >= m || !grid[x][y])
10         return;
11
12     // mark as visited
13     grid[x][y] = false;
14
15     // visit all cells that can potentially extend this island
16     visit_recursive({x + 1, y}, grid);
17     visit_recursive({x - 1, y}, grid);
18     visit_recursive({x, y + 1}, grid);
19     visit_recursive({x, y - 1}, grid);
20 }
21 int count_island_recursive_DFS(std::vector<std::vector<bool>>& grid)
22 {
23     if (grid.size() == 0 || grid[0].size() == 0)
24         return 0;
25
26     const int n = grid.size();
27     const int m = grid[0].size();
28     int ans = 0;
29
30     for (int i = 0; i < n; i++)
31         for (int j = 0; j < m; j++)
32             if (grid[i][j])
33                 {

```



```
34     ans++;  
35     visit_recursive({i, j}, grid);  
36 }  
37  
38 return ans;  
39 }
```

Listing 37.3: Recursive DFS solution, to the problem of counting the number of islands in a map.

38. Median of two sorted arrays

Introduction

The median is one of the most basic and important concepts in statistics and probability theory with applications in almost every field of science. It is defined as the value that splits a certain dataset into two equally sized halves: the higher and the lower half. For example the median of the dataset $\{1, 3, 4, 6, 10, 12, 19\}$ is 6 because we have 3 elements greater and 3 elements smaller than 6. When the size of the dataset is even, no such element exists and thus the median is defined as the mean of the two middle elements; For instance given the dataset $\{1, 3, 4, 6, 8, 10, 12, 19\}$, the median is $\frac{6+8}{2} = 7$.

The problem covered in this chapter concerns finding the median from a dataset provided as two separate input lists of values (you can imagine, for instance, that each input set comes from a separate thread as part of a multithreaded application to analyze a large dataset). Although an obvious solution can be derived from the definition of median, this problem is still considered difficult to solve optimally in a coding interview context as it requires non-trivial insights and careful implementation.

We will start by dissecting the problem statement before we then take a deeper dive into a number of possible solutions beginning with a naive and inefficient approach and working up to the more more sophisticated and optimal approach.

38.1 Problem statement

Problem 56 You are given two **sorted** arrays A and B of size m and n , respectively. Your task is to write a function that takes as input A and B and returns the median of the two sorted arrays. A and B can be considered to be proper subsets of a third dataset $C = A \cup B$.

■ Example 38.1

Given two sorted arrays:

- $A = [1, 4, 6, 10, 15]$
- $B = [2, 3, 5, 6]$

The median is 5 (see Figure ??). ■

■ Example 38.2

Given two sorted arrays:

- $A = [1, 4, 6, 10]$
- $B = [2, 3, 5, 6]$

The median is $\frac{5+4}{2} = 4.5$ (see Figure ??). ■

38.2 Clarification Questions

Q.1. Can A or B be empty?

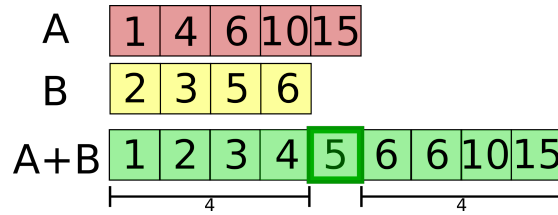


Figure 38.1: Example of median of two sorted arrays where the total number of elements is odd.

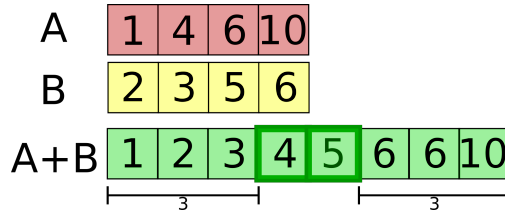


Figure 38.2: Example of median of two sorted arrays where the total number of elements is even.

Yes, but you can assume that $|A \cup B| > 0$ i.e. at most one of the input array can be empty.

38.3 Discussion

Let's start our discussion by reviewing the concept of the median. The median of a collection C of n elements is (C_i represents the i^{th} element of C):

- $C_{\frac{n}{2}}$ if n is odd (see Figure ??)
- $\frac{C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil}}{2}$ if n is even (see Figure ??)

In simpler terms the median of a sorted collection is the element which divides the collections into two equally sized halves, left and right, each with the same number of elements. If n is even, clearly such element does not exist and thus the median is defined to be the mean of the two middle elements as shown in Figure ?. Additionally, note that because the collection is sorted then all the elements in the left half are smaller than or equal to the median and all the elements on the right half are larger.

38.3.1 Brute-force

Armed with the definition of median, we can immediately devise a simple and effective approach to find it given the two input sorted arrays. The only difference between the problem statement and the definition of median is that we are given two sorted arrays and not just one. Therefore the very first thing that should come to mind is to:

1. create a third array $C = A \cup B$, which is the combination of the two input arrays
2. proceed by sorting C ,
3. calculate the median (not forgetting to take into consideration the parity of $|C|$)

This approach is clearly correct as it is a direct consequence and application of the definition of the median given above; but it is also far from being optimal. Listing 38.1 shows a C++ implementation of this idea. Time and space complexities of this approach are $O((n+m)\log(n+m))$ (because of sorting) and $O(s+m)$ (space required by the third array), respectively. Despite being sub-optimal this solution does have the benefit of being very short (only a few lines) and easy to read, explain and understand.

```

1 double mediam_sorted_arrays_naive(const std::vector<int> &A,
2                                   const std::vector<int> &B)
3 {
4     std::vector<int> C(A);
5     C.insert(std::end(C), std::begin(B), std::end(B));
6     std::sort(std::begin(C), std::end(C));
7
8     const auto mid = C.size() / 2;
9     return (C.size() & 1) ? C[mid] : (C[mid] + C[mid + 1]) / 2.0;
10 }

```

Listing 38.1: Naive implementation of solution to the problem of finding the median of two sorted arrays.

38.3.2 Brute-force improved

The brute-force approach can be improved somewhat if we use the fact that the arrays are already sorted. In the approach described in Section 38.3.1 we do not use this fact and therefore we are forced to sort the entire array *C* that we created by blindly juxtaposing *A* and *B* one after the other. By taking advantage of the fact that the inputs are sorted we can create the array *C* in a smarter way, so that it is already sorted. In order to do this we will use the fact that you can merge two sorted array into a third sorted array in linear time. You may already be familiar with this idea if you know how the famous merge-sort algorithm[[wiki:mergesort](#)] works as the same operation is one of its two basic building blocks. Listing 38.2 shows how this idea can be coded in C++. Note how most of the code is now taken by the `std::vector<T> mergeSortedArrays(const std::vector<T> &A, const std::vector<T> &B)` function that is responsible for taking two sorted arrays (pay attention to the `assert`) as input and returning a third sorted one.

```

1 #include <cassert>
2 template <typename T>
3 std::vector<T> mergeSortedArrays(const std::vector<T> &A,
4                                   const std::vector<T> &B)
5 {
6     assert(std::is_sorted(std::begin(A), std::end(A)));
7     assert(std::is_sorted(std::begin(B), std::end(B)));
8
9     const int size = A.size() + B.size();
10    std::vector<int> C;
11    C.reserve(size);
12
13    auto itA = std::begin(A);
14    auto itB = std::begin(B);
15
16    while (itA != std::end(A) && itB != std::end(B))
17    {
18        if (*itA < *itB)
19            C.push_back(*itA++);
20        else
21            C.push_back(*itB++);
22    }
23
24    while (itA != std::end(A))
25        C.push_back(*itA++);
26
27    while (itB != std::end(B))
28        C.push_back(*itB++);
29    return C;
30 }

```

```

31
32 double mediam_sorted_arrays_merge(const std::vector<int> &A,
33                                   const std::vector<int> &B)
34 {
35     std::vector<int> C = mergeSortedArrays(A, B);
36
37     const int mid = C.size() / 2;
38     return (C.size() & 1) ? C[mid] : (C[mid] + C[mid + 1]) / 2.0;
39 }

```

Listing 38.2: Naive implementation of solution to the problem of finding the median of two sorted arrays using the merge part of merge-sort algorithm.

The time and space complexities of this version are both $O(n+m)$, much better than for the solution presented in Section 38.3.1. It is, however, still sub-optimal as this problem can be solved in logarithmic time. We are going to see how in Section 38.3.3.

38.3.2.1 Merge sorted arrays in linear time

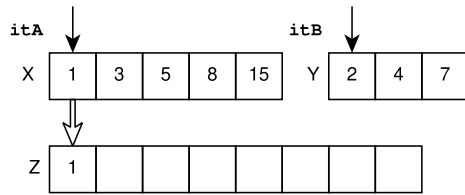
How exactly can we merge two sorted arrays X and Y into a third sorted array Z in linear time? The basic idea is that we can build Z incrementally starting from an empty array and at each step of the process inserting one of the elements of X or Y depending on which one of the two contains the smallest element at that moment. In the implementation of `mergeSortedArrays` this is achieved by using two iterators, `itX` and `itY` each pointing to the next element of X and Y to be inserted in Z , respectively. The `while` loop is responsible for comparing the two elements pointed by the iterators and always inserting the smallest one into Z . Once an element is merged in the final array, the corresponding iterator is incremented so the next value will be considered at the next iteration. When one of the two iterators reaches the end of its array all we are left with are the remaining elements of the other collection that we can at this point blindly insert into Z because they are sorted (see the last two `while` loops in the code). Figure 38.3 shows all the steps that are necessary to perform the merging for the arrays: $X = \{1, 3, 5, 8, 15\}$ and $Y = \{2, 4, 7\}$. At step 1 (Figure 38.3a) Z is initially empty and `itA` and `itB` point to the beginning of X and Y , respectively. Because the element pointed by `itA` is smaller, it is selected for merging and thus `itA` is incremented. At step 2 (Figure 38.3b) the element pointed by `itB` is smaller, and as in the previous step, it is merged in Z and `itB` is incremented. The same operations are performed for all the steps depicted in Figures 38.3c, 38.3d, 38.3e, 38.3f and 38.3g. Eventually it goes out of range signalling that all the elements in Y have been processed (see Figure 38.3g). At this point then, as shown in Figure 38.3h we can safely merge all the elements in X into Z that is now ready to be returned (see Figure 38.3i).

38.3.3 Logarithmic solution

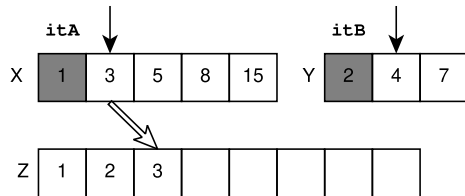
If we want to improve the $O(n+m)$ solution we have at hand at this point we need to abandon the idea of constructing a third array containing all the elements from the inputs. There is no way this can be done in less than linear time as one must at least access the input element once. In reality, we do not actually need to have the array C at all.

The key insights are:

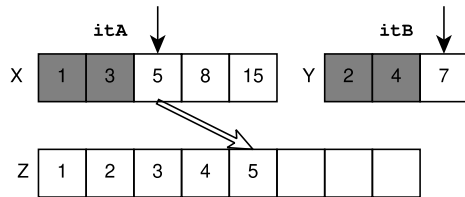
- we know exactly what the size of the merged array C would be: $n+m$.
- we also know that the part of C to the left of the median, C_l would be made up from elements among the smallest values of A and B . These values also lie in the left portions of A and B . For instance, in the example in Figure ?? we can see that the left half (the first 5 elements) of $A \cup B$ is made up from the first two elements



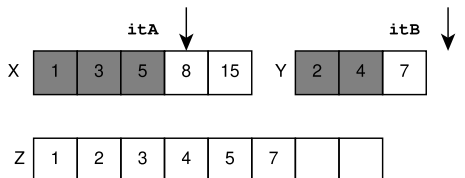
(a) Step 1: Z is initially empty. itX and itY points to the beginning of X and Y , respectively. The element pointed by itX is merged as it is smaller. itX is advanced by one position.



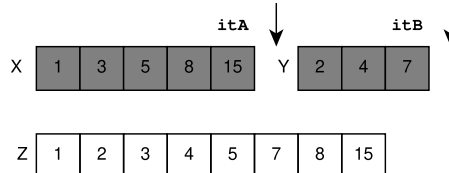
(c) Step 3: itX is smaller than itY , thus it is the one being merged. itX is also advanced by one position.



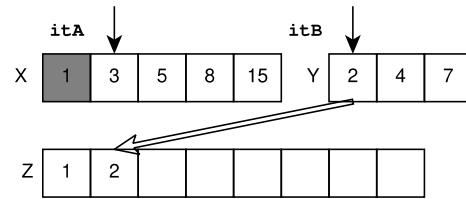
(e) Step 5: itX is smaller than itY , thus it is the one being merged. itX is also advanced by one position.



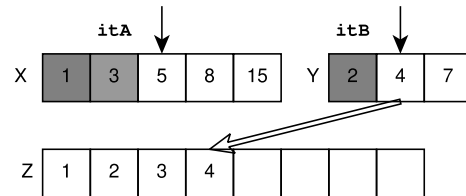
(g) itY now points to the past-the-end element of Y . There are no more elements of Y to merge.



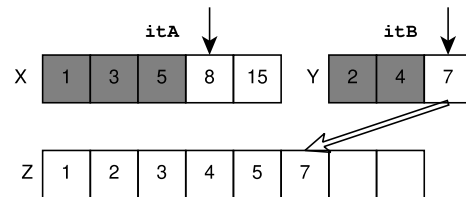
(i) All the elements of X and Y have been merged. Z contains the element of $X \cup Y$ and it is sorted.



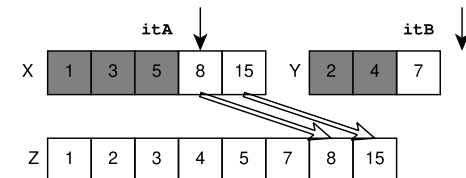
(b) Step 2: itY is smaller than itX , thus it is the one being merged. itB is also advanced by one position.



(d) Step 4: itY is smaller than itX , thus it is the one being merged. itB is also advanced by one position.



(f) Step 6: itY is smaller than itX , thus it is the one being merged. itB is also advanced by one position.



(h) Step 7: All the remaining elements from the current location of itX to the end of X are merged into Z .

Figure 38.3: This figure shows how two sorted arrays can be merged into a third sorted array in linear time. The hollow indicates which element at that step is selected to go into the third array. Notice that the iterator associated with that element is then moved forward. Already merged cells are

of A and the smallest 3 elements of B . Because C will be sorted, only the smallest elements of A and B will can be part of C_l .

The problem here is that we do not know exactly how many elements of A will be part C_l but if we do then we also know immediately how many elements of B go to C_l and at that point we can calculate the median. We cannot directly find how many elements of A contribute to C_l , but we can test fairly easily if the first i elements do. Let's suppose we try to make C_l by using i elements from the left portion of A . Because $|C| = n + m$ then $|C_l| = \frac{n+m}{2} = i + j$ where j is the number of elements from the left part of B contributing to C_l . Thus if we take i elements from A we need to take $j = (n+m) - i$ elements from B . Once i and j are decided, we also know that the last element of C_l , will be the maximum element among the first i elements of A and the first j elements of B . From these arguments it follows that the right half of C , C_r must contain all the remaining $n - i$ elements of A and $m - j$ of B , and also that the first element of C_r will be the smallest element among them. Given:

- M_l is the largest elements among the $A[i] \ B[j]$
- m_r is the smallest element among the $A[i+1] \ B[j+1]$

then if i is indeed the right amount of elements from A belonging to C_l then, $M_l \leq m_r$. If $M_l > m_r$ then we need to understand whether we took too many or too few elements from A to be part of C_l . We can check this by checking whether M_l belongs to B or A , respectively. Thus if $A[i] > B[j]$ we reduce or increase i by doing $r = r - 1$. Conversely if $A[i] < B[j]$ then i is increased by moving the left boundary of the binary search range: $l = l + 1$.

Listing 38.3 shows an implementation of the idea above.

```

1  size_t midpoint(const size_t l, const size_t r)
2  {
3      assert(l <= r);
4      return l + (r - l) / 2;
5  }
6  double mediam_sorted_arrays_binary_search(const std::vector<int> &A,
7                                             const std::vector<int> &B)
8  {
9      size_t l = 0, r = A.size() - 1;
10     const size_t size_C = A.size() + B.size();
11     const size_t half_size_C = size_C / 2;
12
13     auto median = 0.0;
14     while (l <= r)
15     {
16         const size_t i = midpoint(l, r);
17         const size_t j = half_size_C - i;
18         /*
19          const int idx_i = i - 1;
20          const int idx_j = j - 1;
21          if (A[i - 1] <= B[j] && B[j - 1] <= A[i])
22              if (size % 2 == 0)
23                  return (std::max(A[i - 1], B[j - 1]) + std::min(A[i], B[j])) / 2.0;
24              else
25                  return std::max(A[i - 1], B[j - 1]);
26
27          if (A[i - 1] > B[j])
28              r = i - 1;
29          else
30              l = i - 1;*/
31     }
32     return median;

```

Listing 38.3: Binary search solution to the *median of two sorted arrays* problem.

<https://leetcode.com/problems/median-of-two-sorted-arrays/> complete this code first use binary search to find i l , r is the range of elements of A initially $l = 0$ $r = \min(n, (n+m)/2)$ $i = l+r/2$ $j = n+m-i$ if \max among $A[i]$ and $B[j] \leq \min A[i+1], B[j+1]$ we have a median otherwise if $A[i] > B[j]$ then $r = i-1$ else $l = i+1$

39. BST Lowest Common Ancestor

Introduction

The lowest common ancestor is an important concept in graph theory and is often a topic or a fundamental building block of coding interview questions. For example, LCA has important applications in graph theory in:

- computation of *minimum spanning tree*,
- finding a *dominator tree*, or
- as a stepping stone for algorithm for *network routing*, or *range searching*.

Given a tree and two nodes p and q , the lowest common ancestor of p and q , ($LCA(p, q)$) is defined as the lowest or deepest node that has both p and q as descendants. In other words the LCA is the shared ancestor of p and q that is the farthest from the root of the tree.

There are several known algorithms for finding the LCA efficiently on a generic tree; one of the most fundamental being the one from Harel and Tarjan [5, 6]. In this chapter, however, we will focus on the (simpler) problem of finding the LCA for trees of a particular kind: binary search trees. This constraint greatly simplifies the general problem of finding LCAs.

39.1 Problem statement

Problem 57 Write a function that takes a binary search tree T , and two nodes p and q and returns their lowest common ancestor.

■ Example 39.1

Given the tree in Figure 39.1 the lowest common ancestor for nodes:

- $p = 2, q = 12$ is node 3
- $p = 12, q = 3$ is node 12
- $p = 18, q = 1$ is the root node 13

39.2 Clarification Questions

Q.1. What should we return in case the input tree is empty?

You can return an empty tree.

Q.2. Should I check for the validity of the binary search property for T ?

No, you can assume T to always be a valid binary search tree.

Q.3. Can I assume p and q to always be always present in T ?

Yes

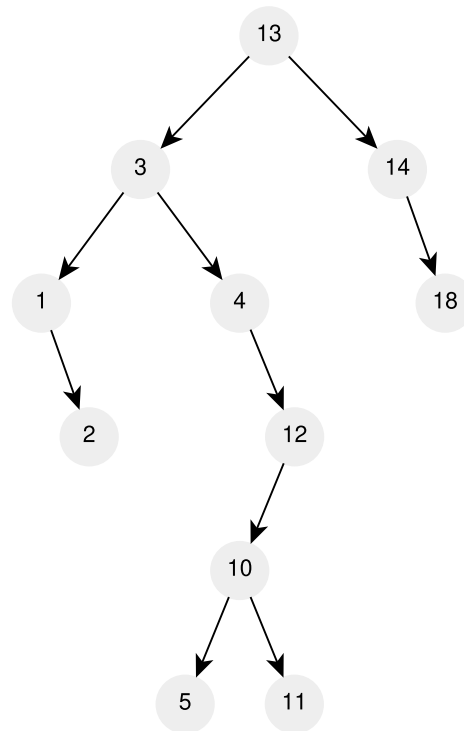


Figure 39.1: Binary Search tree of the Example 39.1

39.3 Discussion

Based on the definition of LCA one of the simplest solution possible would be to compute the path from the root to p and q and store them as two separate lists of nodes: P_q and P_p . We can then compare these lists knowing that they will match up until a certain point, say up to the k^{th} node of the path. After that the lists do not match anymore. Therefore the k^{th} element of the list is the LCA for p and q . For instance in the tree in Figure 39.1 the paths from the root to the nodes 5 and 2 are the following:

$$P_5 = \{\mathbf{13}, \mathbf{3}, 4, 12, 10, 5\} \quad (39.1)$$

$$P_2 = \{\mathbf{13}, \mathbf{3}, 1, 2\} \quad (39.2)$$

As you can see they match up to node 3 which is indeed their LCA.

If we try the same approach for the nodes 5 and 11 their respective paths from the root are:

$$P_5 = \{\mathbf{13}, \mathbf{3}, \mathbf{4}, \mathbf{12}, \mathbf{10}, 5\} \quad (39.3)$$

$$P_{11} = \{\mathbf{13}, \mathbf{3}, \mathbf{4}, \mathbf{12}, \mathbf{10}, 11\} \quad (39.4)$$

P_5 and P_{11} match up to the penultimate node (10). Therefore, their LCA is node 10.

This approach is correct and is easily implementable. Its time and space complexity is $O(k)$ where k is the height of T (which for unbalanced trees might be proportional to the number of nodes in T). Listing 39.1 show an implementation of the idea above.

```
1 template <typename T>
2 void find_path_helper(Node<T>* root,
```

```

3         const T& target,
4         std::vector<Node<T>*>& path)
5     {
6         assert(root); // because target is guaranteed to be in the tree.
7
8         // visited a new node. remember it
9         path.push_back(root);
10        if (root->val == target)
11        {
12            // found the target element. we can stop as the path is complete
13            return;
14        }
15
16        // classic BST search
17        if (target <= root->val)
18            find_path_helper(root->left, target, path);
19        else
20            find_path_helper(root->right, target, path);
21    }
22
23    template <typename T>
24    std::vector<Node<T>*> find_path(Node<T>* root, const T& node)
25    {
26        std::vector<Node<T>*> path = {};
27        find_path_helper(root, node, path);
28        return path;
29    }
30
31    template <typename T>
32    Node<T>* findLeastCommonAncestor_paths(Node<T>* root, const T& p, const T& q)
33    {
34        std::vector<Node<T>*> P_p = find_path(root, p);
35        std::vector<Node<T>*> P_q = find_path(root, q);
36
37        // find the point up to which P_p and P_q are the same
38        auto itp = begin(P_p);
39        auto itq = begin(P_q);
40        Node<T>* ans = *itp;
41        while ((itp != end(P_p) && itq != end(P_q)) && (*itp == *itq))
42        {
43            ans = *itp;
44            itp++;
45            itq++;
46        }
47        return ans;
48    }

```

Listing 39.1: LCA solution based on the difference of paths from the root.

This approach is, however, not optimal as we waste space by storing entire paths which is not necessary as we only really care about the last common node. One way to avoid memorizing the entire paths is to find the path for both p and q simultaneously and only remember the last node we visited. If at some point during this visit we find that the the next node to visit for p is different from the direction that the search for q requires, we can stop as this is the point where the paths for p and q diverge and return the last element that was common. Despite the fact that this approach does not improve the time complexity we have only a constant space usage which is a very good improvement compared to the linear space complexity for Listing 39.1. This optimized version is shown in Listing 39.2.

```

1  template <typename T>
2  void find_path_optimized_helper(Node<T>* root,
3                                const T& p,
4                                const T& q,
5                                Node<T>*& last_common)
6  {
7      assert(root); // because target is guaranteed to be in the tree.
8
9      // LCA is the current node. Either p is descendant of q or the other way
10     // around
11     if (root->val == p || root->val == q)
12     {
13         last_common = root;
14         return;
15     }
16     last_common = root;
17
18     // paths for p and q take different direction from here
19     if ((p <= root->val && q > root->val) || (p > root->val && q <= root->val))
20         return;
21
22     // they are both lower or equal than val or both higher
23     if (p <= root->val)
24     {
25         find_path_optimized_helper(root->left, p, q, last_common);
26     }
27     else
28         find_path_optimized_helper(root->right, p, q, last_common);
29 }
30
31 template <typename T>
32 Node<T>* findLeastCommonAncestor_paths_optimized(Node<T>* root,
33                                                  const T& p,
34                                                  const T& q)
35 {
36     Node<T>* ans = root;
37     find_path_optimized_helper(root, p, q, ans);
38     return ans;
39 }

```

Listing 39.2: Space optimized version of Listing 39.1

We can also simplify the implementation shown in Listing 39.2 further by rewriting it such that it runs iteratively rather than recursively. Listing 39.3 shows this iterative version which starts at the root of T and keeps navigating the tree by moving left or right until the direction of the search for both p and q is the same. When the path diverges we can stop and return the current node which is the lowest shared node in the paths from the root to p and q .

```

1  template <typename T>
2  Node<T>* findLeastCommonAncestor_reference(Node<T>* root,
3                                             const T& p,
4                                             const T& q)
5  {
6      while (root)
7      {
8          const auto& payload = root->val;
9          if (payload > p && payload > q)
10             {
11                 root = root->left;
12             }

```

```
13     else if (payload < p && payload < q)
14     {
15         root = root->right;
16     }
17     else
18     {
19         return root;
20     }
21 }
22 return root;
23 }
```

Listing 39.3: Iterative solution to the problem of finding the LCA in a binary search tree.

40. Distance between nodes in BST

Introduction

In the problem described in this chapter we will investigate how to find the distance between two nodes in a binary search tree. As we will see this problem can be approached and solved easily if we are able identify the key underlying concepts. This will become apparent after we look at a few examples and our advice for approaching this problem (and to be honest all problems on graphs and trees) is to draw and discuss quite a few examples with your interviewer. This help you get a much better intuitive understanding of what the problem is really about, which will eventually lead to the eureka moment.

This is a relatively short chapter because the solution is built on top of the solution for the problem discussed in chapter [REFERENCE]. In Section 40.1 we will have a look at the formal problem statement and in Section 40.3 we discuss the solution approach and we will look into two possible different implementations: recursive and iterative.

40.1 Problem statement

Problem 58 Write a function that takes as input a binary search tree T and two nodes p and q and returns the distance between p and q . The distance between two nodes $D(p,q)$ is defined as the number of edges you need to traverse to get from p to q .

■ **Example 40.1**

Given the tree shown in Figure 40.1b, $p = 1$ and $q = 3$, the function returns $D(1,3) = 2$

If $p = 3$ and $q = 2$ the function returns 1. ■

■ **Example 40.2**

Given the tree shown in Figure 40.1a, $p = 5$ and $q = 2$, the function returns $D(5,2) = 6$ ■

40.2 Clarification Questions

Q.1. Can p be equal to q ?

Yes, this is a valid case.

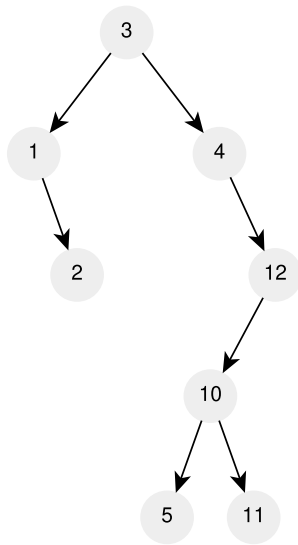
Q.2. Is it guaranteed for p and q to be present in T ?

Yes, you can assume T always contains both p and q .

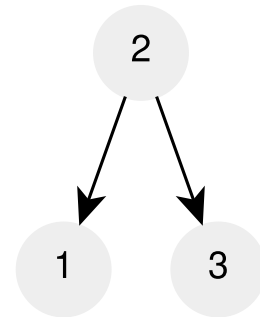
40.3 Discussion

As already mentioned in the introduction the key to solving this problem lies in clearly identifying the correct approach from the problem statement by using several examples, solving them by hand and then identifying similarities in their solutions.

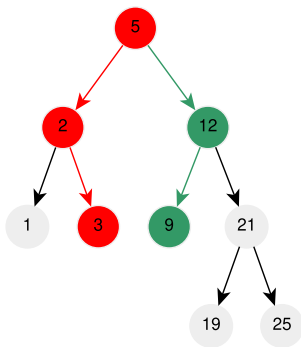
Let's have a look at some instances of this problem and their solution. If we consider T to the tree depicted in Figure 40.1c then the distance between nodes $p = 3$ and $q = 9$ is



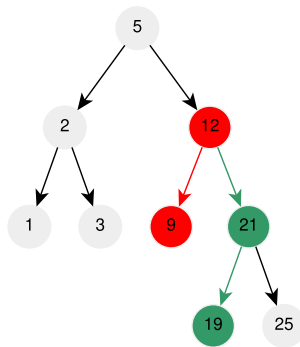
(a) Binary Search tree of the Example 40.2



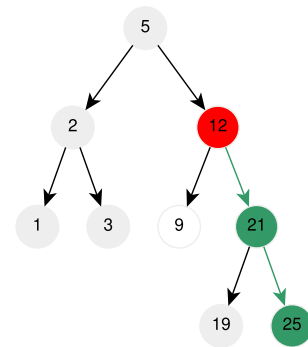
(b) Binary Search tree of the Example 40.1



(c)



(d)



(e)

Figure 40.1: Various instances of the problem of finding the distance between two nodes in a BST and their solutions. Figures 40.1c, 40.1d, and 40.1e have nodes and arcs in red and green to highlight the paths between the p and q and their LCA.

4 and can be found by walking up the tree from node 3 to a node 5 (follow the red arcs) from which we can descend and reach node 9 (green arcs). Note that node 5 is the first node from which we can travel down the tree and reach both nodes 3 and 9. You can also see that the same reasoning applies to the trees shown in Figures 40.1d and 40.1e where you get the minimum distance by traveling up to the first node that allows you to reach the destination node. Notice that in Figure 40.1e the path upwards has length zero as from p you can already reach q by traveling down the tree.

At this point it should be clear that the minimum distance between two nodes can be calculated as the sum of distances from p and q their lowest common ancestor (LCA). The LCA is the lowest node from which it is possible to walk in a downward direction and reach both p and q . In order to go from p to q one must pass through their LCA. If you need to refresh your memory on the topic of finding the LCA on binary search trees you can read Chapter 39. Listing 40.1 shows a possible implementation of the idea described above.

```

1  /**
2  * Calculates the distance between T and a node with payload `val`
3  * Perform a classic BST visit/search (downward) from T for val.
4  *
5  * @param T is valid binary search tree
6  * @param val is the value to be searched in T
7  * @return the distance between T and val
8  */
9  template <typename U>
10 int find_distance_down(const Node<U>* const T, const U val)
11 {
12     assert(T && "node val exists and is reachable from T");
13     const auto& payload = T->val;
14     if (payload == val)
15         return 0;
16     if (val <= payload)
17         return 1 + find_distance_down(T->left, val);
18     return 1 + find_distance_down(T->right, val);
19 }
20
21 /**
22 * Find the distance between two nodes a tree
23 *
24 * @param T is valid binary search tree
25 * @param p is the payload of a node in T
26 * @param q is the payload of a node in T
27 * @return the minimum number of edges to traverse to get from p to q
28 */
29 template <typename U>
30 int min_distance_nodes_BST(Node<U>* T, const U p, const U q)
31 {
32     const Node<U>* const lca = find_LCA(T, p, q);
33     return find_distance_down(lca, p) + find_distance_down(lca, q);
34 }

```

Listing 40.1: Solution to the problem of finding the distance between two nodes in a binary search tree.

40.4 Conclusion

In this chapter we have seen how we can efficiently solve the problem of finding the distance between two nodes in a binary search tree by using the concept of LCA (discussed more in details in Chapter 39). The general strategy is that we can calculate the distance between the LCA and both p and q . The sum of these two distances is the final answer. The distance between two nodes in a binary search tree can be found by slightly modifying the standard search algorithm for BSTs so that we return the number of recursive calls made instead of a boolean value signaling whether the element to be searched was found or not.

41. Counts the items in the containers

Introduction

Imagine you are the owner of a successful online store. You would like to be able to know the number of items you still have in the warehouse. The problem is that you cannot just walk into the warehouse and count the items as they are stored in closed containers. Thankfully, the warehouse is equipped with sensors and is able to produce a string representing the state of the warehouse and single containers. The problem described in this chapter investigates how we can write an algorithm that takes such a string (the state of all the containers in the warehouse) and be able to answer queries on the number of elements that are present in some portions of the warehouse itself.

Unsurprisingly, this problem has been reported as being asked during Amazon interviews and can be considered as of a medium difficulty. We will investigate two solutions:

- brute-force based on relatively straightforward logic (blindly count the items in the string) and easy to code (in Section 41.3.1),
- a more sophisticated solution with optimal asymptotic complexity where the input string is pre-processed so that queries can be answered faster.

41.1 Problem statement

Problem 59 You are given a string s representing the current state of a warehouse. The string contains only two kinds of characters:

‘*’(ASCII 42) : representing an item

‘|’(ASCII 124) : representing the boundaries of a container.

A container is a closed space within the warehouse and it is represented in s by a pair of ‘|’. Items within a container c are represented as ‘*’ appearing within the two ‘|’ defining c . You are also given an array of pairs $Q = \{(s_0, e_0), (s_1, e_2), \dots, (s_{n-1}, e_{e-1}) : 0 \leq s_i \leq e_i \leq |s|\}$, where each pair in Q identifies a substring in s . Each element of Q is a query you must answer to.

Your task is to write a function that returns an array A of length n , containing the answers to all of the queries in Q , where each element A_i is the number of items contained in all the **closed** compartments between (s_i, e_i) .

■ Example 41.1

Given $s = `|**|*|*|`$ and $Q = \{(0, 4), (0, 5)\}$ the function returns $A = \{2, 3\}$. s has a total of 2 closed containers the first with 2 and 1 item inside respectively.

The first query asks you to find the number of elements in the substring $s[0, 4] = `|**|*|`$ where three items are represented but only two are within a closed container (the first two).

The second query refers to the substring $s[0, 5] = `|**|*|`$. The items are the same as in the previous query but this time all of them are in closed containers.

■

■ Example 41.2

Given $s = \texttt{`*|*|'}$ and $Q = \{(0,2), (1,3)\}$ the function returns $A = \{0,1\}$. s has a total of two items and only 1 closed container containing only a single item.

The first query refers to the substring $s[0,2] = \texttt{`*|*|'}$. No closed containers are represented in such a substring thus the answer in this case must be 0. However, the second question refers to $s[1,3] = \texttt{`|*|'}$ where we can see we have a valid container. We can therefore counts the elements in it. ■

41.2 Clarification Questions

Q.1. Is it guaranteed for the input string s to only contains valid characters?

Yes, you do not need to worry about the sanity of the input.

41.3 Discussion

41.3.1 Brute-force

This problem has a straightforward solution that essentially loops over all the elements specified in a query $(s, e) \in Q$ and counts all the elements inside the containers. Because the $|s - e|$ is $O(|s|)$ the complexity of this approach is $O(|s| * |Q|)$. Listing 41.1 shows an implementation of this idea. Note that most of the code complexity of this solution is in the `count_items_in_substring` function that has to make sure only to count items that are within a closed container. It does so by first finding the first container wall appearing after the start of the query interval. We can safely skip all those items because they are not inside a container. Once we have found the beginning of the first container, we can proceed by counting the elements one container at a time.

```
1 using Query = std::pair<int, int>;
2 constexpr char kContDelimiter = '|';
3 constexpr char kItem = '*';
4
5 int count_items_in_substring(const std::string& s, const Query& query)
6 {
7     const auto& [start, end] = query;
8     assert(start <= end);
9     assert(end <= std::ssize(s));
10
11     auto curr_char = start;
12     // find the first container
13     while (curr_char <= end && s[curr_char] != kContDelimiter)
14         curr_char++;
15     curr_char++;
16
17     int ans = 0;
18     int cont_counter = 0;
19     while (curr_char <= end)
20     {
21         if (s[curr_char] == kItem)
22         {
23             cont_counter++;
24         }
25         else if (s[curr_char] == kContDelimiter)
26         {
27             ans += cont_counter;
```

```

28     cont_counter = 0;
29     }
30     curr_char++;
31 }
32 return ans;
33 }
34 std::vector<int> items_in_containers_naive(const std::string& curr_char,
35                                           const std::vector<Query>& Q)
36 {
37     std::vector<int> ans;
38     ans.reserve(Q.size());
39     for (const auto& q : Q)
40         ans.push_back(count_items_in_substring(curr_char, q));
41     return ans;
42 }

```

Listing 41.1: Naïve solution to the *items in the container* problem.

41.3.2 Linear time solution

There is, however, a much faster solution that can be easily implemented provided we have come pre-computed values. Performing some pre-computation in order to speed-up an algorithm is a common idea that is useful in solving many coding interview questions. In this particular problem, we are going to calculate two values for each character of the input string:

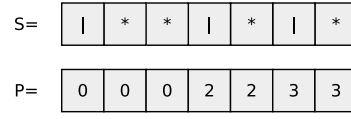
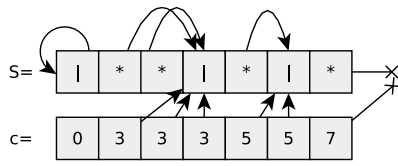
C_i , the closest delimiter to the right : for each character s_i of the input string s we want to have information about the index of the closest container delimiter appearing after it. In other words we are looking for the index $j > i$ such that $s[j] = '|'$. If such index does not exist then we assume it is the index of the last character of s , i.e. $|s| - 1$.

P_i , the number of elements in all containers to the left : this value should answer the question: given a character at index i of s , how many items are placed into all the closed containers appearing to the left of i ?

When this information is available for each and every position of s then we can answer each query in constant time. Given a query (l, r) , we can calculate the answer to it by using the information about the closest container delimiter to the right of l , $c \geq l$, to find the beginning of the first container in the range (s, e) . All the elements between l and c can be ignored. So now that we have transformed our query from (l, r) to (c, r) we are ready to use the prefix sum of the number of elements in the containers.

We can calculate the answer to (c, r) by simply returning $P_r - P_l$: the number of elements in the all container up to index r minus the number of elements in all containers up to the index l .

Figure ?? shows the values of P and C for the input string is $s = "|**|*|*"$. Each value of the array P_i contains the count of the items inside all the containers in the prefix of s up to and including index i . For instance $P_4 = 2$ because the substring of s between indices 0 and 4 only contains one container with two elements in it while $P_5 = 3$ because between indices 0 and 5 we have two containers with 2 and 1 items inside, respectively. The values in C_i contains the indices of the first character $'|'$ in the suffix of s from index i . For instance $C_1 = 3$ because the first $'|'$ after index 1 appears at index 3 in s while $C_3 = 3$ because $s[3]$ contains $'|'$ itself. Note that the last element of s always contains $|s| - 1$ regardless of whether $s_{|s|-1}$ is $'|'$ or not. Listing 41.2 shows an implementation of the idea above. The main driver function is `items_in_containers_lineartime` which first calls two other functions: 1. `find_closest_bars_right` and, 2. `prefix_sum_containers_items` that



(a) Each element at location i of the array C contains an integer corresponding to the smallest index j of s larger than i such that $s[j] = '|'$ is a delimiter of a container.

(b) Each element at index i of the array P contains the number of elements in all the closed containers appearing before i in s .

are responsible for the pre-computation of C and P , respectively. Note that in Listing 41.2, for the sake of clarity, C and P , are named `closest_bars_right` and `prefix_sum_count_items`, respectively.

```

1
2 std::vector<int> prefix_sum_containers_items(const std::string& s)
3 {
4     std::vector<int> cont_prefix_sum;
5     cont_prefix_sum.reserve(s.size());
6
7     auto it = std::begin(s);
8     while (it != std::end(s) && *it != '|')
9     {
10         cont_prefix_sum.push_back(0);
11         it = std::next(it);
12     }
13     cont_prefix_sum.push_back(0);
14     it = std::next(it);
15
16     int cont_curr_countainer = 0;
17     while (it != std::end(s))
18     {
19         const auto count_prev_containers =
20             (cont_prefix_sum.size() > 0 ? cont_prefix_sum.back() : 0);
21         if (*it == '|')
22         {
23             // sum of the previous and previous container items
24             cont_prefix_sum.push_back(count_prev_containers + cont_curr_countainer);
25             cont_curr_countainer = 0;
26         }
27         else
28         {
29             cont_prefix_sum.push_back(count_prev_containers);
30             cont_curr_countainer++;
31         }
32         it = std::next(it);
33     }
34     return cont_prefix_sum;
35 }
36
37 std::vector<int> find_closest_bars_right(const std::string& s)
38 {
39     std::vector<int> ans(s.size());
40     int idx_last_bar = std::ssize(s) - 1;
41     for (int i = std::ssize(s) - 1; i >= 0; i--)
42     {
43         if (s[i] == '|')
44             idx_last_bar = i;

```

```

45     ans[i] = idx_last_bar;
46 }
47 return ans;
48 }
49 std::vector<int> items_in_containers_linear_time(const std::string& s,
50                                                  const std::vector<Query>& Q)
51 {
52     const std::vector<int>& prefix_sum_count_items =
53         prefix_sum_containers_items(s);
54     const std::vector<int>& closestBarsRight = find_closest_bars_right(s);
55
56     std::vector<int> ans;
57     ans.reserve(Q.size());
58     for (const auto& [start, end] : Q)
59     {
60         const auto& new_start = closestBarsRight[start];
61         if (new_start >= end)
62         {
63             ans.push_back(0);
64         }
65         else
66         {
67             const auto& count_before_start =
68                 (new_start <= 0) ? 0 : prefix_sum_count_items[new_start];
69             ans.push_back(prefix_sum_count_items[end] - count_before_start);
70         }
71     }
72     return ans;
73 }

```

Listing 41.2: Linear time and linear space solution to the *items in the container* problem.

42. Minimum difficulty job schedule

Introduction

Imagine you are part of a team currently busy doing beta testing on your new cool feature. The testing consists of executing several tasks. Each task has dependencies on other tasks and is assigned a certain amount of complexity points (a measure of how difficult a task is to perform; it is not a measure of time). The dependencies between the tasks have already been worked out i.e. the order in which the tasks are going to be executed is decided. The problem in this chapter is about creating a schedule plan for the execution of these tasks spanning across a given number of days. Among all possible schedules, we need to make an effort to calculate the minimum possible complexity achievable for the schedule that will eventually make sure all tasks are executed and also that there is at least one task executed every day.

42.1 Problem statement

Problem 60 Write a function that takes as an input a list of tasks I and an integer d . The elements in I are dependent on each other and to schedule a certain task I_i all the tasks $I_j : j < i$ have to be completed. The function should return the minimum complexity among all possible schedules of length exactly d days. The complexity of a job is calculated as the sum of the complexity of every single day of the schedule. The complexity of a day of the schedule is defined as the maximum complexity of the tasks planned for that day.

As an additional constraint, you have to make sure that there is at least one task scheduled for each day.

■ Example 42.1

Given:

- $I = \{6, 5, 4, 3, 2, 1\}$
- $d = 2$

the function returns 7. You can schedule tasks 0 to 4 during the first day and the last task during the second day. You cannot just schedule all tasks during the first day because then you would have a day in the schedule without planned tasks which is not permitted. ■

■ Example 42.2

Given:

- $I = \{10, 10, 10\}$
- $d = 4$

the function returns -1 . There is no way to schedule tasks for 4 days when there are only 3 tasks available for scheduling. ■

■ Example 42.3

Given:

- $I = \{7, 1, 7, 1, 7, 1\}$
- $d = 3$

the function returns 15. You can schedule the first 4 tasks the first day for a total complexity of 7. Tasks at index 4 and 5 can be scheduled for days 2 and 3 respectively.

Notice that in this case if $d = 2$ then the function would return 8. ■

■ Example 42.4

Given:

- $I = \{11, 111, 22, 222, 33, 333, 44, 444\}$
- $d = 6$

the function returns 843. You can schedule tasks 0, 1, 2, 3, 4 in the first 5 days and the rest during the 6th. ■

42.2 Clarification Questions

Q.1. What should the function return in the case where it is not possible to make a valid schedule? For instance when $|I| < d$?

You can return -1 in that case.

Q.2. Is it guaranteed for the complexity values to be positive (≥ 0)?

Yes you can assume complexities are always positive.

42.3 Discussion

This is a classic example of a problem that can be easily solved via dynamic programming but can be very challenging if you try to approach it differently. Fortunately, the statement is full of hints that this problem can be solved using DP. For instance: 1. it is an optimization problem, and, 2. you are not asked to find an actual schedule, but only the value of the best possible one.. Very often those are the two most common ingredients in a DP problem. It's important, therefore, to be able to quickly identify the clues within the statement that point to a DP based solution.

42.3.1 Brute-force

If you do not immediately think about DP one of the possible approaches to this problem would be to try out all possible schedules, and for each of them calculate its cost, and return the smallest. The problem explicitly mentions a case where a solution does not exist. This is an easy case as there is only one scenario where you cannot schedule jobs for d days: when the number of jobs to be scheduled is strictly less than d . The core of the problem is really about the case where $|I| \geq d$. You can think about a schedule as a way of splitting I into d non-empty sub-arrays. You can split an array into d parts by placing $d - 1$ splitting-points in I at different locations. A different placing of the splitting-points leads univocally to a different schedule. There is, therefore, a one-to-one correspondence between a subset of size $d - 1$ of $\{0, 1, 2, \dots, |I| - 2\}$ (the splitting point locations) and schedules (see Equation 42.1). We can therefore generate all possible schedules by generating all possible combinations of $d - 1$ elements from $\{0, 1, 2, \dots, |I| - 2\}$ where each number of a combination $\{e_0, \dots, e_{d-1}\}$ represents a splitting point in I and e_i identifies the following subarray of I : $\{A_{e_i-1+1}, A_{e_i-1+1}, \dots, A_{e_i}\}$.

In order to solve this problem we can calculate the costs for each of the schedules represented by a combination of $d - 1$ elements of $\{0, 1, 2, \dots, |I| - 1\}$, and return the cost

of the best (the one having minimum cost overall). The cost of a schedule - as shown in the problem statement - is the sum of the costs for each of the d day where the cost of a single day is the cost of the most expensive job scheduled for that particular day. So given a schedule represented by the combination $e = \{e_1, \dots, e_d\}$ we can easily calculate its cost, $C(e)$, by using:

$$C(e) = \underbrace{\max(A_0, A_1, \dots, A_{e_1})}_{\text{cost for the 1st day}} + \underbrace{\max(A_{e_1+1}, A_{e_1+2}, \dots, A_{e_2})}_{\text{cost for the 2nd day}} + \dots + \underbrace{\max(A_{e_{d-1}+1}, A_{e_{d-1}+2}, \dots, A_{|I|-1})}_{\text{cost for the } d^{\text{th}} \text{ day}} \quad (42.1)$$

42.3.1.1 Generate all combinations

The real challenge at this point concerns the generation of combinations in groups of $d - 1$ elements. We can generate all the combinations one at a time by using a backtracking algorithm where we try to construct one combination of elements at a time. A possible recursive implementation of such an algorithm is shown in Listing 42.1. The function `generate_all_combination` takes as a input two integers k and l . k represents the size of the combination and l identifies the elements of the combinations i.e. $0, 1, \dots, l - 1$. If you had to write a generic function for generating combinations you would also most likely have a parameter containing the list of elements from which to generate the combinations. In this case, such a list is implicit as we need to generate combinations of splitting points and can be uniquely identified by a single integer. For instance `generate_all_combination(3,10)` generates all combinations of three elements from $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and `generate_all_combination(2,4)` generates all combinations of 2 elements from $\{0, 1, 2, 3\}$. `generate_all_combination_helper` is a recursive function which enumerates all combinations. It takes the following parameters:

- `std::vector<std::vector<int>>> & all_combinations`: the output list of all generated combination,
- `std::vector<int>& combination`: the work-in-progress combination,
- `const unsigned k`: the size of the combinations
- `const unsigned l`: the last number we can add to the work-in-progress combination
- `const unsigned curr_el`: the first number we can add to the combination

Each call tries to place a number in the work-in-progress `combination` at a location specified by `curr_el`. Initially `curr_el = 0` and each recursive call increases it by 1. Eventually `curr_el = d` and we can stop the recursion and return. At that point the combination is ready and saved into `all_combinations`. After each recursive call, the last inserted element is removed from the work-in-progress combination and another number is pushed. The process repeats until there are no more numbers to be pushed.

```

1 using Combination = std::vector<int>;
2 using CombinationList = std::vector<Combination>;
3
4 void all_combinations_helper(CombinationList& all_combinations, Combination&
   combination, const unsigned d, const int curr_idx, const unsigned limit)
5 {
6     if(combination.size() == d){
7         //combination is ready
8         all_combinations.push_back(combination);
9         return;
10    }
11
12    for(size_t i = curr_idx; i < limit ; i++)
13    {
14        combination.push_back(i);

```



```

15     all_combinations_helper(all_combinations, combination, d, i+1, limit);
16     combination.pop_back();
17 }
18
19 }
20
21 auto all_combinations(const unsigned d, const unsigned limit)
22 {
23     Combination work_in_progress;
24     work_in_progress.reserve(d);
25     CombinationList all_combinations;
26     //limit -1 because the last cut cannot be empty
27     all_combinations_helper(all_combinations, work_in_progress, d, 0, limit-1 );
28     return all_combinations;
29 }

```

Listing 42.1: Function that generates all the combinations of size k from the elements $\{0, 1, \dots, I\}$

42.3.1.2 Wrapping-up

Once we are able to generate all the possible schedules we are going to evaluate the cost associated with each of them, and pick the one with the smallest difficulty overall. All that is left to do at this point is to come up with a way to evaluate a given schedule c . We have already seen in Equation 42.1 how a certain combination of $d - 1$ splitting points maps directly to subarrays of I . The function `calculate_cost_schedule` in Listing 42.1 uses this idea to evaluate a schedule and calculate its difficulty by summing up the difficulties of each of the tasks scheduled each day. Note that `start` and `finish` identify the elements of I in the following range: $[start, finish]$ (the element pointed by `finish` is included). The function `min_difficulty_scheduler_combinations` is the driver that is responsible for keeping track of the minimum difficulty among all the processed schedules.

```

1  int calculate_cost_schedule(const std::vector<int>& I,
2                             const std::vector<int>& cutpoints_combo)
3  {
4      int ans = 0;
5      auto start = std::begin(I);
6      for (const auto& cutpoint : cutpoints_combo)
7      {
8          const auto finish = std::begin(I) + cutpoint + 1;
9          ans += *std::max_element(start, finish);
10         start = finish;
11     }
12     ans += *std::max_element(start, std::end(I));
13     return ans;
14 }
15
16 int min_difficulty_scheduler_combinations(const std::vector<int>& I,
17                                           const unsigned d)
18 {
19     if (I.size() < d)
20         return -1;
21
22     auto all_combinations_cutpoints = all_combinations(d - 1, I.size());
23     int ans = std::numeric_limits<int>::max();
24     for (const auto& cutpoints_combo : all_combinations_cutpoints)
25     {
26         ans = std::min(ans, calculate_cost_schedule(I, cutpoints_combo));
27     }
28     return ans;

```

Listing 42.2: Brute-force solution that works by evaluating all the possible schedules generated using Listing 42.1

42.3.2 Dynamic Programming

The key insight needed to solve this problem with DP is that given that you have decided on a set of tasks that are scheduled in the first day, say the first i tasks, then the minimum difficulty of a schedule across d days having the first i elements scheduled the first day is the sum of 1. the largest difficulty among the first i tasks, and 2. the cost of the best possible schedule of the last $|I| - i$ tasks across $d - 1$ days. More formally, if $C(I, d)$ is a function returning the minimum cost of a schedule of the tasks in I across d days and can be defined as follows:

$$\left\{ \begin{array}{l} C(\emptyset, 0) = 0 : \text{the cost of scheduling 0 task in 0 days is 0} \\ C(\emptyset, d > 0) = +\infty : \text{it is impossible to schedule 0 tasks in 1 or more days} \\ C(|I|, 0) = +\infty : \text{it is impossible to schedule 1 or more tasks in 0 days} \\ C(|I|, d) = \min_{\substack{\forall j \in \{0, 1, \dots, |I|-1\} \\ \forall \text{ schedule of the } d^{\text{th}} \text{ day}}} \left(\max I_j + \underbrace{C(I - \{0, 1, \dots, j\}, d - 1)}_{\text{optimal solution to a subproblem}} \right) \end{array} \right. \quad (42.2)$$

$C(I, d)$ has a recursive definition and we can quickly see that the problem has both the properties any DP problem has:

optimal substructure: can be solved by solving and combining together various **optimal** solutions to **smaller** subproblems.

overlapping subproblems: the same problems are solved over and over again. (try to draw the recursion tree for C if you are not entirely convinced)

42.3.3 Top-down

Without any optimization, the function that we obtain by translating the recursive definition of Equation 42.2 is extremely inefficient due to the fact that problems are recalculated over and over (See Appendix 64). In order to make good use of DP, we can therefore use memoization to avoid unnecessary recomputation. Listing 42.3 shows a possible implementation of this idea where a `std::unordered_map` is used to remember the calls to `min_difficulty_helper` (the equivalent of the function C). Note that given I and d the function can “only” be invoked in $|I| \times d$ ways. Therefore, in the worst case scenario, by using memoization we will never make more than $|I| \times d$ calls to `min_difficulty_helper`. Because the cost of a single call to `min_difficulty_helper` is linear in $|I|$ the complexity of the whole algorithm is $O(|I|^2 d)$

```

1 struct KeyHash
2 {
3     std::size_t operator()(const std::tuple<int, int>& key) const
4     {
5         return std::get<0>(key) ^ std::get<1>(key);
6     }
7 };
8
9 using Cache = std::unordered_map<std::tuple<int, int>, int, KeyHash>;
10
```

```

11 long min_difficulty_scheduler_DP_topdown_helper(const std::vector<int>& I,
12                                                  const size_t start,
13                                                  const int d,
14                                                  Cache& cache)
15 {
16     if (start >= I.size() && d == 0)
17         return 0;
18
19     const size_t remaining = I.size() - start;
20     if (remaining < d)
21         return std::numeric_limits<int>::max();
22
23     auto t = std::make_tuple(start, d);
24     if (auto it = cache.find(t); it != cache.end())
25         return it->second;
26
27     int M = I[start];
28     long ans = std::numeric_limits<int>::max();
29     for (size_t i = start; i < I.size(); i++)
30     {
31         M = std::max(M, I[i]);
32         ans = std::min(
33             ans,
34             M + min_difficulty_scheduler_DP_topdown_helper(I, i + 1, d - 1, cache))
35         ;
36     }
37     cache[t] = ans;
38     return ans;
39 }
40 int min_difficulty_scheduler_DP_topdown(const vector<int>& I, int d)
41 {
42     Cache cache;
43     auto ans = min_difficulty_scheduler_DP_topdown_helper(I, 0, d, cache);
44     if (ans >= std::numeric_limits<int>::max())
45         return -1;
46     return ans;
47 }

```

Listing 42.3: Dynamic programming top-down solution.

42.3.4 Bottom-up

In this section, we are going to have a look at how we can implement the DP idea in a bottom-up fashion. Like many bottom-up solutions, we have to come up with a way of filling out a table of values of some sort. For this problem we can use a table T of size $d \times |I|$ where each element of the table $T[i][j]$ will eventually contain the solution to the problem of scheduling the elements of $|I|$ up to and including the task at index j in exactly i days. Clearly, filling some cells of T is easier than others. For instance, all the values of the first column of T are all filled with a value indicating that the problem has no solution because you cannot schedule any task in 0 days. An exception should be made for $T[0][0]$ that is filled with 0 as the cost of scheduling 0 tasks in 0 days is equivalent to the cost of doing nothing. To mark that a subproblem is impossible we can use a large value, or perhaps the largest value a cell of T can hold.

Additionally, the values of the first row are also relatively easy to fill in as they contain values that are symmetrically equal to the cells in the first column. Each value of the first row represents a solution to the problem of scheduling some task in 0 days and there is clearly no way that can be done (except for the case when you have 0 task to schedule).

An element of the first row $T[1][j]$ corresponds to the subproblem of scheduling the first j of I in exactly one day. Its cost is clearly the maximum difficulty among the elements of I from 0 to j as there is only one way of scheduling all the $j + 1$ tasks in a single day.

Things get a bit more interesting when looking at the second row. When we have two days at our disposal to schedule j tasks we have more freedom over which task to schedule on the first day and which on the second. The values we just filled for the first column and row can be helpful in making the best decision for the elements of the second row. We can, in fact, fill $T[2][j]$ by scheduling one task on the first day, and $j - 1$ on the second. Or 2 tasks the first day and $j - 2$ in the second and so on. But which of these divisions is the best? Easy! let's try them all and see which one yields the smallest difficulty overall. Therefore we can calculate $T[2][j]$ as shown in Equation 42.3:

$$T[2][j] = \min_{1 \leq k \leq j-1} \left(T[1][k] + \left(\max_{k+1 \leq l \leq j} I_l \right) \right) \quad (42.3)$$

Equation 42.4 is really saying that we can calculate the minimum difficulty of scheduling the tasks in I up to the one having index j by calculating the minimum among the costs of scheduling the tasks up to the index $k \leq j - 1$ on the first day and the rest on the second. The trick here is that we have already calculated all the possible costs of scheduling all possible number of tasks, and thus all we have to do at this step is to calculate the costs of scheduling the tasks from the one having index $k + 1$ to task j . This can be done by simply returning the maximum costs among those tasks.

This exact same reasoning can be applied to all the other rows and we can therefore come up with a general formula that can be used to fill the entire table of values as shown in Equation 42.4.

$$T[i][j] = \min_{i-1 \leq k \leq j-1} \left(T[i-1][k] + \left(\max_{k+1 \leq l \leq j} I_l \right) \right) \quad (42.4)$$

Clearly the solution to the entire problem is in $T[d][|I| - 1]$: the cost of scheduling all the elements in I in exactly d days. Listing 42.4 shows an implementation of this idea.

```

1
2 using DPTable = std::vector<std::vector<int>>>;
3
4 int min_difficulty_scheduler_DP_bottomup(const std::vector<int>& I, int d)
5 {
6     const int num_tasks = I.size();
7     const int INF = std::numeric_limits<int>::max();
8
9     if (num_tasks < d)
10         return -1;
11
12     DPTable T(d, std::vector<int>(num_tasks, INF));
13
14     // initializing values for the first day
15     int maxV = std::numeric_limits<int>::min();
16     for (int j = 0; j < num_tasks; j++)
17     {
18         maxV = std::max(maxV, I[j]);
19         T[0][j] = maxV;
20     }
21
22     for (int i = 1; i < d; i++)
23     {
24         for (int j = 0; j < num_tasks; j++)
25         {

```

```

26 // l is the number of tasks scheduled the previous days i-1 days
27 // l must be at least i-1 (it is impossible to schedule them otherwise)
28 for (int l = i - 1; l < j; l++)
29 {
30     // elements from [0,l] the scheduled the days before and [l+1,j] today
31     const auto start_task_dth_day = std::begin(I) + l + 1;
32     const auto end_task_dth_day   = std::begin(I) + j + 1;
33     auto max_tasks_second_day =
34         *std::max_element(start_task_dth_day, end_task_dth_day);
35
36     T[i][j] = std::min(T[i][j], T[i - 1][l] + max_tasks_second_day);
37 }
38 }
39 }
40
41 return T[d - 1][num_tasks - 1];
42 }

```

Listing 42.4: Dynamic programming bottom-up solution.

This implementation has a space complexity of $O(d * |I|)$, but a closer inspection of the code and Equation 42.4 should make clear that we do not really need to keep all the values for T in memory all the time. In fact, all we need is two rows with the values for days i and $i - 1$. This way the complexity goes down to $O(|I|)$. The Listing can be easily modified so that it implements this memory saving strategy. We will leave this as an exercise for the reader.

42.4 Conclusion

43. Max in manhattan neighborhood^K

Introduction

This chapter discusses a very interesting problem based on the concept of the *Manhattan distance*^①; an alternative way to measure distances that is particularly useful in real life. Imagine you need to measure the distance between two points on a map. You can use the Euclidean distance and come up with a number that in reality is not going to be super helpful unless you can fly. This is because that number is not going to tell you the actual distance you need to cover if you want to get to your destination by moving on land. For example, what is the distance between the Empire State building and Times Square in New York? If you are not naturally equipped with wings then what you actually do is to jump in a car or a cab or a bike and follow the grid pattern of streets of Manhattan. This means that you would probably have to cover around 15 blocks north and 3 south (See Figure 43.1). The idea of measuring the distance by counting the number of steps we take in the north-south or west-east directions underlies what is known as the taxicab distance. In this framework, the distance is not represented as a straight line going from point A to point B (like it would for the Euclidean distance) but it is a zig-zagged sequence of vertical and horizontal segments, representing movements along the north-south and east-west axis. Therefore, the formula for measuring the taxicab distance is all about measuring the length of the horizontal and vertical segments. The formula for measuring the Manhattan distance in Equation 43.1 .

$$d = |x_1 - x_2| + |y_1 - y_2| \quad (43.1)$$

The problem in this chapter will challenge you to find, for each cell of a given matrix, the largest value in any cell sitting at a Manhattan distance below a certain threshold.

43.1 Problem statement

Problem 61 Write a function that given 1. a matrix I of n rows and m columns and 2. an integer $K > 0$ returns a new matrix M of size $n \times m$ where $M[i][j]$ contains the maximum value among the elements in the Manhattan neighborhood of size K for the cell (i, j) . The Manhattan neighborhood of size K for a cell (i, j) is composed of all cell (p, q) such that:

$$N(i, j, K) = \{(p, q) \mid |i - p| + |j - q| \leq K\} \quad (43.2)$$

■ Example 43.1

Given: $I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ and $K = 1$ the function return $I = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 8 & 9 & 9 \end{bmatrix}$

^①Also known as *taxicab distance*.

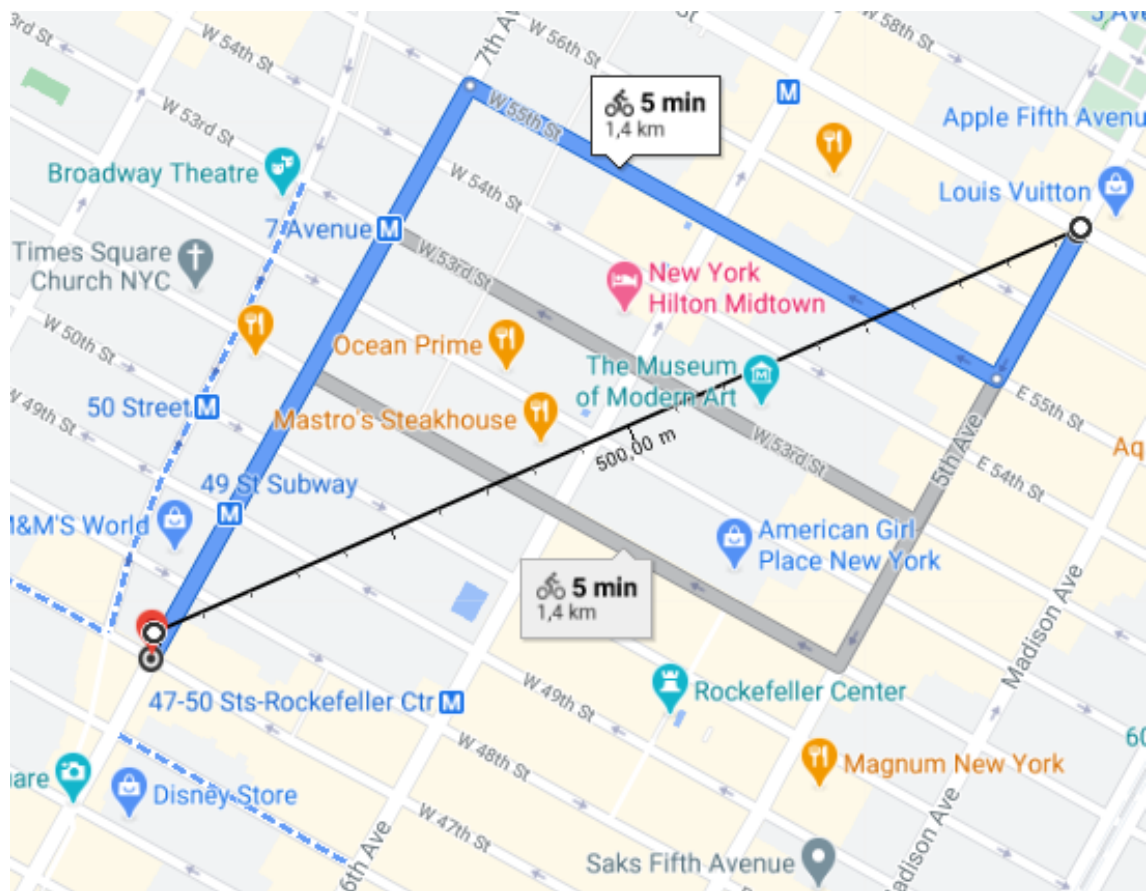


Figure 43.1: Taxicab distance from times square to the Trump's tower. Notice that the black straight-line is depicting the Euclidean distance, and that the latter is shorter than the actual taxicab distance between the two points.

■ Example 43.2

Given: $I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ and $K = 2$ the function return $I = \begin{bmatrix} 7 & 8 & 9 \\ 8 & 9 & 9 \\ 9 & 9 & 9 \end{bmatrix}$

■

43.2 Clarification Questions

43.3 Discussion

43.3.1 Brute-force

This problem can be tackled by using a brute-force approach that blindly calculates the answer for each cell as per the problem statement. All that is needed to find the answer for the cell (i, j) is to calculate the maximum among the cells that are at a Manhattan distance of at most K from it (the cells that are part of the Manhattan neighborhood of size K). Therefore, the problem boils down to figuring out the neighborhood for a given cell. Figure 43.2 shows an example of such a neighborhood where the numbers inside the cells represent the distance from the cell (i, j) at the center. Given a cell (i, j) , figuring out exactly which cells are part of the neighborhood and which are not is not particularly difficult. Instead of coming up with a way of generating such a list of cells, it is easier to loop over all the cells within a square of size $2K$ having cell (i, j) at its center, and to ignore the ones that do not satisfy Equation 43.2. Because both 1. the number of cells in the neighborhood and 2. the number of cell in such square is quadratic in K , then, asymptotically speaking, the additional work required by visiting cells that are inside the area of cells we are looping on but not part of the neighborhood does not really make any difference. Listing 43.1 shows an implementation of this idea. This approach is clearly correct and has a time complexity of $O(nmK^2)$ as for each of the nm cells of the matrix we do exactly K^2 work. The space complexity is $O(1)$ as no additional space is used other than the second matrix we must return.

```

1  //! This function returns the max value among the cells of I that are part of
2  //! the
3  //! manhattan neighborhood of size K for cell at (i,j)
4  /*!
5   \param I the input matrix
6   \param cell the coordinate of the cell for which the max value is to be
7   calculated \param K the size of the manhattan neighborhood \return the max
8   value for (i,j) among all cells (p,q) satisfying: |i-p|+|j-q| <= K
9  */
10 using Matrix = std::vector<std::vector<int>>>;
11 using Cell    = std::pair<int, int>;
12
13 auto find_max_in_manhattan_neigh_k(const Matrix& I,
14                                     const Cell& cell,
15                                     const int K)
16 {
17     const int rows    = I.size();
18     const int cols    = I.back().size();
19     const auto [i, j] = cell;
20     assert(i >= 0 && i < rows && j >= 0 && j < cols);
21
22     int ans = I[i][j];
23     for (int p = std::max(0, i - K); p <= std::min(rows - 1, i + K); p++)
24     {
25         for (int q = std::max(0, j - K); q <= std::min(cols - 1, j + K); q++)
26         {

```

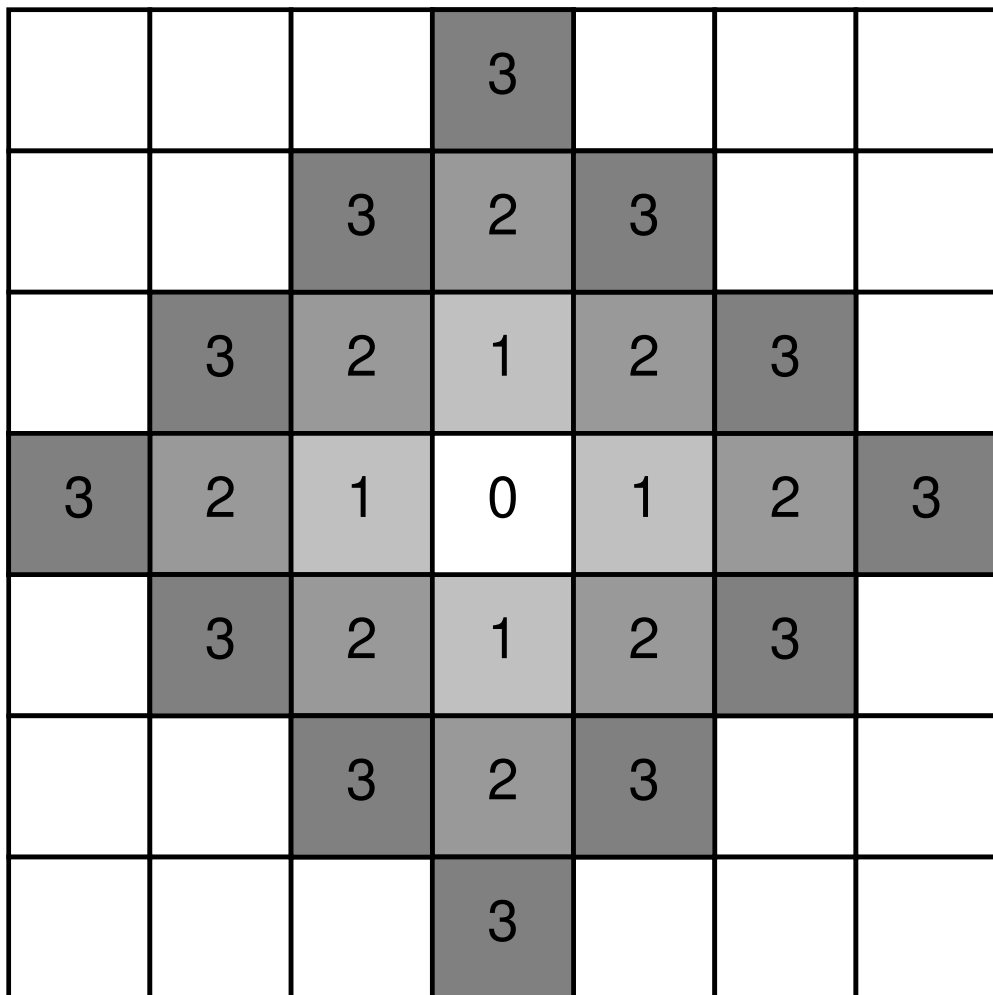



Figure 43.2: Cells in the Manhattan neighborhood of size 3. The numbers in each cell represent the Manhattan distance from the central cell.

```

27     if (std::abs(i - p) + std::abs(j - q) <= K)
28     {
29         ans = std::max(ans, I[p][q]);
30     }
31 }
32 }
33 return ans;
34 }
35
36 Matrix max_manhattan_matrix_k_bruteforce(const Matrix& I, const unsigned K)
37 {
38     const int rows = I.size();
39     const int cols = I.back().size();
40
41     Matrix M(I);
42     for (int i = 0; i < rows; i++)
43     {
44         for (int j = 0; j < cols; j++)
45         {
46             M[i][j] = find_max_in_manhattan_neigh_k(I, Cell(i, j), K);
47         }
48     }
49     return M;
50 }

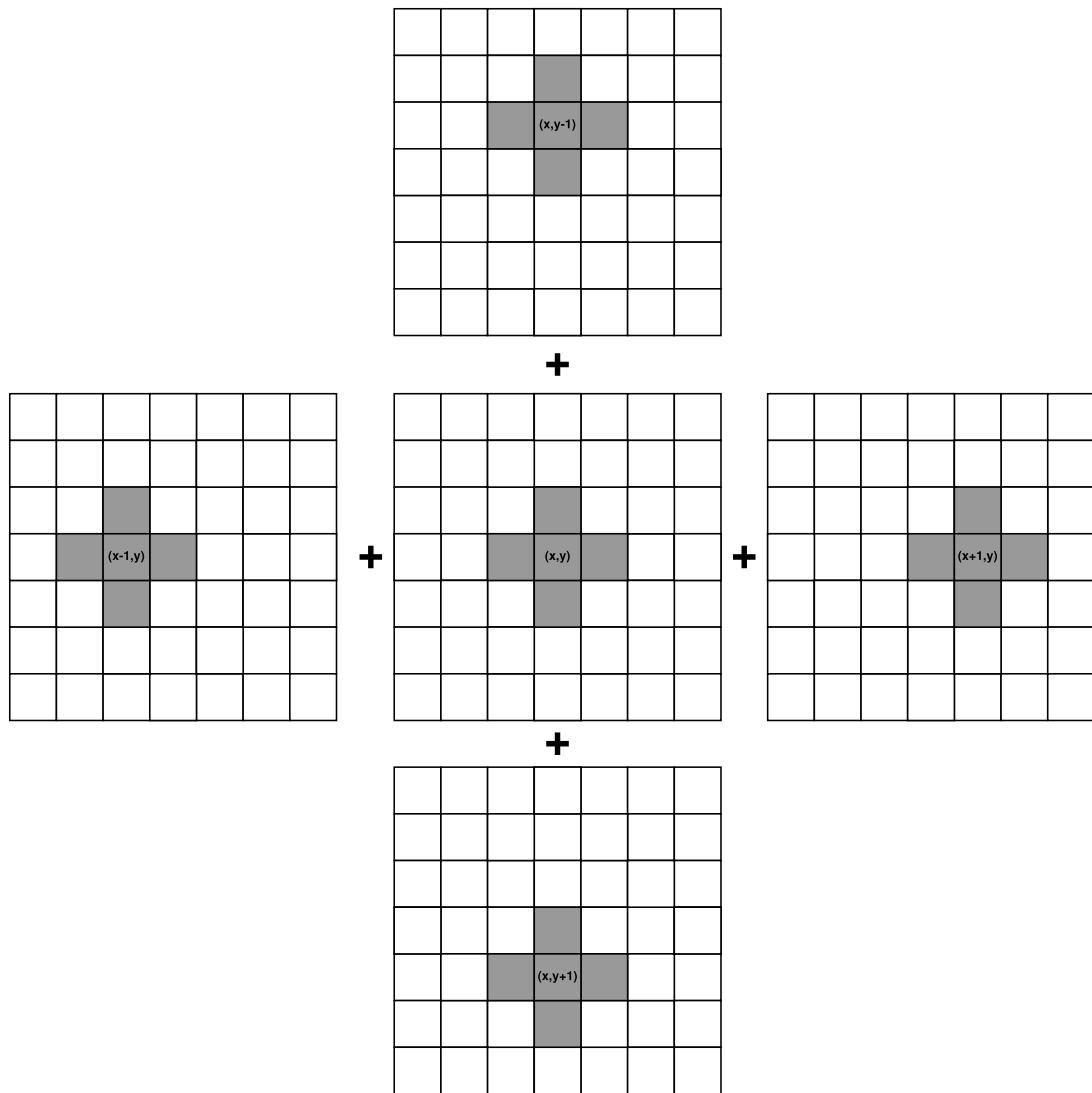
```

Listing 43.1: Brute Force solution to the problem of finding the max cell within the manhattan neighborhood of size K .

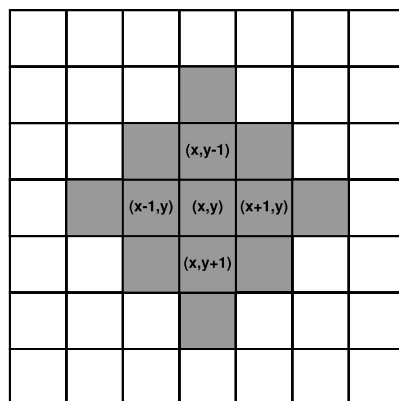
43.3.2 Dynamic Programming

As previously stated, the brute-force solution works by blindly finding the solution for each cell without taking into consideration whether we can use the information already calculated for other cells. Consider the matrix shown in Figure 43.3a showing the neighborhood of size 1 for the cells • (x,y) (center) • $(x+1,y)$ (right) • $(x-1,y)$ (left) • $(x,y-1)$ (top) • $(x,y+1)$ (down) and Figure 43.3b showing the neighborhood of size 2 for the cell (x,y) . Note that the latter is composed by the union of the neighborhoods of size 1 shown in Figure 43.3a. Therefore, if we know the answer for $K = 1$ for the cells • (x,y) • $(x+1,y)$ • $(x-1,y)$ • $(x,y-1)$ • $(x,y+1)$ we can easily calculate the answer for $K = 2$ for the cell (x,y) without having to look at all the 12 cells composing its Manhattan neighborhood of size 2.

We can apply the same line of reasoning to find the answer for $K = 3$. Figure 43.4a shows the neighborhoods of size 2 for the cells • (x,y) • $(x+1,y)$ • $(x-1,y)$ • $(x,y-1)$ • $(x,y+1)$ and Figure 43.4b shows the neighborhood of size 3 for the cell (x,y) . Also in this case the latter can be obtained by the union of all the cells in Figure 43.4a and therefore, if we have the answer for all the sub-problems where $K = 2$ we can obtain the answer for the sub-problems where $K = 3$ without having to scan the entirety of the neighborhood of size 3 for (x,y) . We only have to find the maximum among 5 elements instead of having to look into 25 cells. This idea can be generalized and its formalization is shown in Equation 43.3. The formula is saying that we can obtain the answer for $K = 0$ by simply returning the value of the cell (i,j) . For $K > 0$, we only have to return the max among the neighboring north,south, west and east cells of (i,j) for the same subproblem where $K - 1$.



(a) Manhattan neighborhoods of size 1 for the cells: (x, y) (center), $(x+1, y)$ (right), $(x-1, y)$ (left), $(x, y-1)$ (top) and $(x, y+1)$ (down).



(b) Neighborhood of size 2 for the cell (x, y) obtained by the union of the neighborhood depicted in Figure 43.3a.

Figure 43.3

$$\begin{cases} S(0, i, j) = I[i][j] \\ S(K, i, j) = \max \left[S(K-1, i, j), S(K-1, i+1, j), S(K-1, i-1, j), S(K-1, i, j+1), S(K-1, i, j-1) \right] \end{cases} \quad (43.3)$$

As with all dynamic programming problems, we have two ways to write the solution:

1. top-down, where we use memoization to avoid making duplicate work
2. bottom-up, where we solve subproblems in an ordered manner from the simplest to the hardest.

43.3.2.1 Top-down

The top-down approach is possibly the easiest to write as we can translate $S(K, i, j)$ (see Equation 43.3) into a C++ recursive function, and we can use memoization (in the form of a cache of size $K \times n \times m$) to store intermediate values of S and avoid duplicate work. Listing 43.2 shows an implementation of this approach where such a cache is implemented via a hashmap, which maps the arguments of S , all triplets (K, i, j) , to integers. Notice that the function `hash_combine` and the function `TupleHash` are the machinery that makes us use tuples of type `std::tuple<int, int, int>` as keys in the Cache (which has type `std::unordered_map`). `TupleHash` uses `hash_combine` to calculate the hash value for a given `std::tuple<int, int, int>`. The main driver function is named `max_manhattan_matrix_k_DP_topdown` and the sole purpose is to create the memoization cache and then to call the C++ equivalent of function S (see Equation 43.3): `max_manhattan_matrix_k_DP_topdown_helper`. The latter is a recursive function that takes as input 1. the original input matrix I (which is never modified and is only passed along as a reference), 2. K , the max distance at which we search for the max value in I , 3. `cell`, which contains the coordinate of the cell for which we are finding an answer and finally 4. the memoization cache where we store the answers for a given K and `cell`.

The first thing we do is to unpack the cell into two separate variables representing the row and the column of a cell: i, j . If the coordinates of the cell are outside the boundaries of I then there is no answer for such a cell and we return the smallest possible int. Moreover if $K = 0$, as per Equation 43.3, the answer is the value of the cell (i, j) . When we have already calculated the solution to this problem (i.e. for the same values of K, i, j) then we simply return the memoized value. In all other cases we have to do the actual work and perform the recursive calls to get the answer for the subproblems for the neighboring cells at the previous value of K :

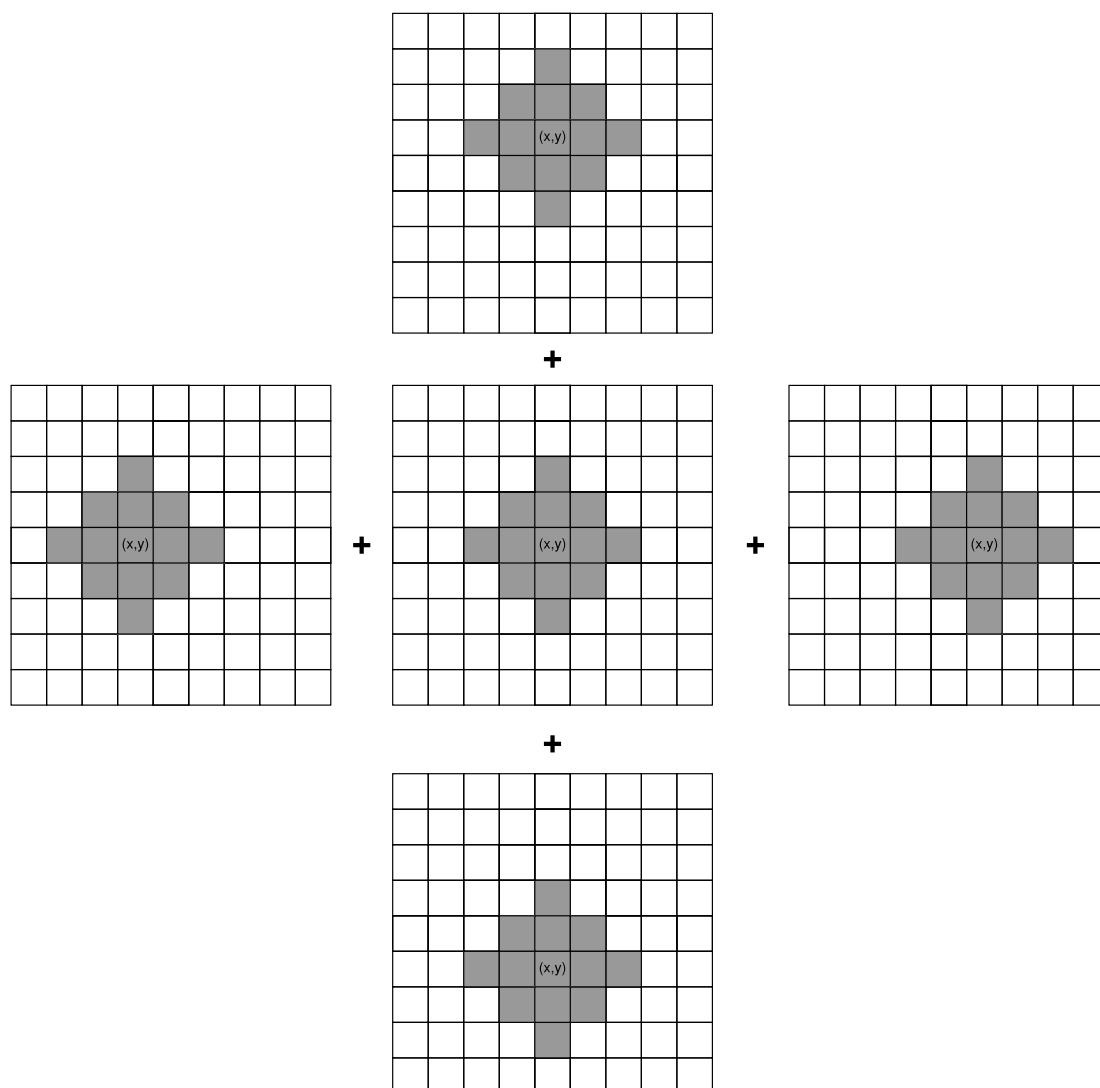
- `max_manhattan_matrix_k_DP_topdown_helper(K-1, i-1, j)`
- `max_manhattan_matrix_k_DP_topdown_helper(K-1, i+1, j)`
- `max_manhattan_matrix_k_DP_topdown_helper(K-1, i, j+1)`
- `max_manhattan_matrix_k_DP_topdown_helper(K-1, i, j-1)`

The time and space complexity of this approach is $O(nmK)$. The proof is quite simple and it boils down to the following facts:

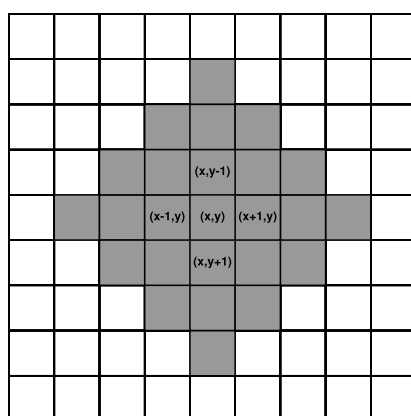
1. there are exactly $n \times m \times K$ different unique ways we can call the recursive function.
2. each function call is memoized. This means that redundant work is avoided, therefore we do not do the work for a recursive call that has already been fully executed.

Each entry in the memoization cache costs 4 integers for a total of $O(nmK)$.

```
1 using Matrix = std::vector<std::vector<int>>>;
2 using Cell   = std::pair<int, int>;
3
4 template <typename SeedType, typename T, typename... Rest>
5 void hash_combine(SeedType& seed, const T& v, const Rest&... rest)
```



(a) Manhattan neighborhoods of size 1 for the cells: (x,y) (center), $(x+1,y)$ (right), $(x-1,y)$ (left), $(x,y-1)$ (top) and $(x,y+1)$ (down).



(b) Neighborhood of size 3 for the cell (x,y) obtained by the union of the neighborhood depicted in Figure 43.4a.

Figure 43.4

```

6 {
7     seed ^= std::hash<T>{}(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
8     (hash_combine(seed, rest), ...);
9 }
10 struct TupleHash
11     : public std::unary_function<std::tuple<int, int, int>, std::size_t>
12 {
13     std::size_t operator()(const std::tuple<int, int, int>& k) const
14     {
15         size_t seed = 0;
16         hash_combine(seed, std::get<0>(k), std::get<1>(k), std::get<2>(k));
17         return seed;
18     }
19 };
20
21 using Cache = std::unordered_map<std::tuple<int, int, int>, int, TupleHash>;
22
23 int max_manhattan_matrix_k_DP_topdown_helper(const Matrix& I,
24                                              const unsigned K,
25                                              const Cell& cell,
26                                              Cache& cache)
27 {
28     const auto [i, j] = cell;
29     if (i < 0 || j < 0 || i >= I.size() || j >= I.back().size())
30         return std::numeric_limits<int>::min();
31
32     if (K == 0)
33         return I[i][j];
34
35     const auto key = std::make_tuple(K, i, j);
36     if (const auto& it = cache.find(key); it != cache.end())
37         return it->second;
38
39     const auto ans = std::max(
40         {I[i][j],
41          max_manhattan_matrix_k_DP_topdown_helper(I, K - 1, {i - 1, j}, cache),
42          max_manhattan_matrix_k_DP_topdown_helper(I, K - 1, {i + 1, j}, cache),
43          max_manhattan_matrix_k_DP_topdown_helper(I, K - 1, {i, j + 1}, cache),
44          max_manhattan_matrix_k_DP_topdown_helper(I, K - 1, {i, j - 1}, cache)});
45
46     cache[key] = ans;
47     return ans;
48 }
49
50 Matrix max_manhattan_matrix_k_DP_topdown(const Matrix& I, const unsigned K)
51 {
52     const int rows = I.size();
53     const int cols = I.back().size();
54     Cache cache;
55
56     Matrix M(I);
57     for (int i = 0; i < rows; i++)
58         for (int j = 0; j < cols; j++)
59             M[i][j] = max_manhattan_matrix_k_DP_topdown_helper(I, K, {i, j}, cache);
60
61     return M;
62 }

```

Listing 43.2: DP top-down solution to the problem of finding the max cell within the manhattan neighborhood of size K .

43.3.2.2 Bottom-up

If we pay closer attention to Equation 43.3 or, equivalently to the top-down implementation in Listing 43.2 we immediately notice that in order to calculate all the values of $S(K, i, j)$ for a given K we only need the values of S for $K - 1$. Because we know the solution to the sub-problems where $K = 0$, we can immediately solve all the problems where $K = 1$. At this point the values for the sub-problems where $K = 0$ are not needed anymore and we can throw them away and use that space to store the solution for the sub-problems where $K = 1$. Now that we have the solution for all sub-problems where $K = 1$, we can proceed and calculate the solutions for $K = 2$. We apply the same line of reasoning to the rest of the sub-problems until we reach the value of K we need.

The bottom-up approach is built on this idea and works by iteratively computing the answers for sub-problems where $K - 1$ before moving on to calculating the answer for the sub-problems for the next value of K . This can be implemented by using two matrices of the same size of I :

- M_{K-1} : storing the values of the sub-problems for the previous value of K we are trying to compute during this step.
- M_K which is the space where we write the answers for the sub-problems we calculate during this step.

When M_K is full and ready, it can be copied into M_{K-1} and continue to process the next value of K . In other words, M_K is a working space where the solutions to the sub-problems for the current K are stored, and M_{K-1} contains all the answers for the sub-problems necessary to calculate the answers at the step.

The computation of a value of M_K uses the same idea as the top-down approach: the value of $M_K[i][j]$ is the maximum among the following five values: 1. $M_{K-1}[i][j]$ 2. $M_{K-1}[i+1][j]$ 3. $M_{K-1}[i-1][j]$ 4. $M_{K-1}[i][j+1]$ 5. $M_{K-1}[i][j-1]$

Note that at any time all the space we need is the space for storing the solution for the sub-problems for two different values of K . This translates into a significant reduction in space complexity compared to the top-down approach described in Section 43.3.2.1 which is $O(nm)$ in this approach.

Listing 43.3 shows an implementation of this idea. Note that in the actual code M_{K-1} and M_K are the variables `previous` and `current`, respectively.

```
1 Matrix max_manhattan_matrix_k_DP_bottomup(const Matrix& I, const unsigned K)
2 {
3     const auto rows = I.size();
4     assert(rows > 0);
5     const auto cols = I.back().size();
6     assert(cols > 0);
7
8     std::array<Matrix, 2> cache = {I, I};
9     Matrix& previous          = cache[0];
10    Matrix& current           = cache[1];
11
12    for (int k = 1; k <= K; k++)
13    {
14        for (int i = 0; i < rows; i++)
15        {
16            for (int j = 0; j < cols; j++)
17            {
18                auto ans = previous[i][j];
19                if (i - 1 >= 0)
20                    ans = std::max(ans, previous[i - 1][j]);
21                if (i + 1 < rows)
22                    ans = std::max(ans, previous[i + 1][j]);
```

```

23     if (j - 1 >= 0)
24         ans = std::max(ans, previous[i][j - 1]);
25     if (j + 1 < cols)
26         ans = std::max(ans, previous[i][j + 1]);
27
28     current[i][j] = ans;
29 }
30 }
31 std::swap(current, previous);
32 }
33 return previous;
34 }

```

Listing 43.3: DP bottom-down solution to the problem of finding the max cell within the manhattan neighborhood of size K .

44. Coin Change Problem

Introduction

The problem discussed in this chapter is considered by many to be a fundamental stepping stone for anyone on the path towards mastering Dynamic Programming (see Section 64). This reputation originates from the fact that this problem encompasses all the crucial ingredients of any DP algorithm with the additional benefit

of having a very intuitive statement as it features things like coins and change that are concepts we are all familiar with.

This problem addresses the question of finding the minimum number of coins that add up to a given amount of money. Many people, when reading the statement of this problem, are tempted to approach it greedily but, as we will see, this does not always (despite it actually often does) lead to the correct answer.

The coin change problem can be seen as an archetype for a whole bunch of DP optimization problems that can be reduced and solved, using the techniques shown in this section (see Chapter 45 and 33, for instance).



44.1 Problem statement

Problem 62 Write a function that, given an array of coin denominations I and an integer t representing an amount of money, returns the minimum number of coins (of any denomination in I) that are necessary obtain t . You have an infinite amount of coins of each denomination.

■ Example 44.1

Given $I = \{1, 2, 5\}$ and $t = 11$, the function returns 3. We can change 11 in many ways, but none of them uses less than 3 coins: • **two** coins of denomination 5, and • **one** coin of denomination 1. ■

■ Example 44.2

Given $I = \{1, 3, 4, 5\}$ and $t = 7$, the function returns 2. We can change 7 by using • **one** coin of value 3 and • **one** of value 4. ■

■ Example 44.3

Given $I = \{1, 5, 8\}$ and $t = 12$, the function returns 4. We can change 12 by using • **two** coins of value 1 and • **two** of value 5. ■

44.2 Clarification Questions

Q.1. Can I be empty?

Yes.

Q.2. Is I sorted?

No, denominations in I are not sorted.

Q.3. Can we assume we can always change t using the denomination in I ?

No, and if that is the case the function should return -1 .

44.3 Discussion

44.3.1 The greedy approach and why it is incorrect

This is one of those problems that can trick inexperienced candidates into thinking about a greedy approach, especially when nudged by the examples given along with the statement that are crafted so that a greedy approach produces the optimal answer.

A greedy algorithm for this problem works by repeatedly picking the largest coin I_k that is smaller than t , and repeating the process on a new target amount $t - I_k$ until we reach 0. Listing 44.1 shows a possible implementation of this algorithm.

If we apply this algorithm to the Example 44.4 we see that initially $t = 11$ and that the largest denomination 5 is smaller than t . Therefore we pick it (in the code this is reflected in assigning the variable `greedy_choice = *it`) and we decrease t by the same amount. Now, $t = 6$ which is still larger than 5. We pick 5 again and $t = 1$. At this point neither 5 nor 2 are smaller or equal than 1 and we choose 1 which is the only denomination that is smaller or equal than the current value of t . Now $t = 0$ and we can stop, after having used 3 coins in total, which is optimal.

However if we try the same algorithm on the Example 44.3 we get the answer 6 which is quite far off from the optimum 4. This approach is also not complete as it fails to find a valid solution like in the case where $I = \{2, 5, 8\}$ and $t = 12$. In this case the greedy algorithm returns -1 , when it is perfectly possible to change the amount 12 by using

- two coins of value 5 and,
- one coin of value 2.

```
1 // Note: This algorithm is neither correct nor complete
2 int change_ways_bruteforce(const std::vector<int>& I, int t)
3 {
4     int ans = 0;
5
6     while (t > 0)
7     {
8         int greedy_choice = 0;
9         for (auto it = I.rbegin(); it != I.rend(); it++)
10         {
11             if (*it <= t)
12             {
13                 greedy_choice = *it;
14                 break;
15             }
16         }
17         if (greedy_choice == 0) // no element smaller or equal to t found
18             return -1;
19
20         t -= greedy_choice;
21         ans++; // used one coin
22     }
23     return ans;
24 }
```

Listing 44.1: Greedy solution which always try to use the largest coin we can. Notice that this approach is incorrect and should not be used during an interview.

44.3.2 Formulation as an optimization problem

This problem can be formalized as an optimization problem where the solution is a set of number $X = \{x_0, x_1, \dots, x_{|I|-1}\}$ of size $|I|$ with each x_j representing how many coins of the denomination I_j are used. Given this formulation, the answer is simply the minimum of Equation 44.1a subject to Equation 44.1b.

$$W(t) = \sum_{j=0}^{|I|-1} X_j \quad (44.1a)$$

$$\sum_{j=0}^{|I|-1} X_j I_j = t \quad (44.1b)$$

$W(t)$ (Equation 44.1a) is the total number of coins used and the constraint $W(t)$ is subject to (Equation 44.1b) forces their collective value to be exactly equal to the target amount t .

44.3.3 Brute-force

The brute-force approach is conceptually straightforward and consists in enumerating and checking every single possible valid combination of coins while keeping track of the one with the fewest number of coins adding up to t . A valid combination is described by an instance of the array X mentioned above in Section 44.3.2.

The enumeration process can be implemented using recursion and backtracking. The idea is that we fill X (which initially is zeroed) incrementally, by starting with the first position, X_0 . A value in X at position j (x_j) represents the number of coins of the denomination I_j (contributing for a total value of $I_j X_j$).

When we try a new value k for x_0 , we know we are adding kI_0 to the overall value of all the coins in X , and of course also that we used k more coins. Once a decision regarding the number of coins of value I_0 we use is made, we can continue and try to fill the next position of X knowing that we have to make up for $t - (kI_0)$ and that we have used already k coins.

This process can be repeated until either we reach a point where we have nothing to make up for anymore, or we still have some amount left to change but, no available denominations to use. In the former case we return the number of coins used up to that point (or we compare it to the current minimum so far), while in the latter, we return a value indicating that there is no solution (usually a large number).

An implementation of this idea is shown below in Listing 44.2. Notice that the function `change_ways_bruteforce_backtracking_helper` takes 4 parameters:

1. `I`: a read-only parameter containing the denominations;
2. `t`: the amount we need to make up for;
3. `j`: the index of the denomination in I we are currently processing;
4. `coin_used`: the number of coin used so far.

```
1 int change_ways_bruteforce_backtracking_helper(  
2     const std::vector<int>& I,  
3     const int t,  
4     const size_t j, /*current denomination*/  
5     const int coin_used /*number of coin in X*/)   
6 {  
7     if (t == 0)  
8         return coin_used;
```

```

9
10 // Either we added more coin than necessary
11 // or we do not have any more denomination to use
12 if (t < 0 || j >= I.size())
13     return std::numeric_limits<int>::max();
14
15 int ans = std::numeric_limits<int>::max();
16 for (int k = 0, new_t = t; new_t >= 0; k++, new_t = t - (I[j] * k))
17 {
18     ans = std::min(ans,
19                     change_ways_bruteforce_backtracking_helper(
20                         I, new_t, j + 1, coin_used + k));
21 }
22 return ans;
23 }
24
25 int change_ways_bruteforce(const std::vector<int>& I, const int t)
26 {
27     return change_ways_bruteforce_backtracking_helper(I, t, 0, 0);
28 }

```

Listing 44.2: Backtracking recursive brute-force solution

The function is initially called with $j = 0$ (indicating we start with the first denomination), $\text{coin_used} = 0$ (no coins are used) and t is set to be equal to the original amount (the one coming from the main driver function `change_ways_bruteforce`). As the execution and the recursion unfold, t is changed accordingly to the value of the k coins of denomination I_j we are trying to use, coin_used is incremented by k , and j is incremented by one.

Figure 44.1 shows the recursion tree of `change_ways_bruteforce_backtracking_helper` when the input is the one shown in Example 44.4; As we can see, there are 4 ways (highlighted in green) of changing 4 by using coin of denominations $\{1, 2, 3\}$:

1. $2 + 2$ (two coins of value 2),
2. $1 + 3$, (one and three coins of values 1 and 3, respectively)
3. $1 + 1 + 2$, (two and one coins of values 1 and 2, respectively), and
4. $1 + 1 + 1 + 1$ (4 coins of value 1).

The time complexity of this approach is exponential in $|I|$. As an informal proof of this fact consider that for each denomination we at least try either to use zero or one coin. Therefore for each element of I we have two choices resulting in $2^{|I|}$ possibilities. The space complexity is linear in $|I|$ as in the worst case the depth of the recursive calls do not go deeper than $|I|$. This is a direct consequence of the base case, checking for $j \geq |I|$.

44.3.4 Dynamic Programming - Top-Down

Like all DP problems, one of the first things we need to do, is to try to define the solution to the problem in terms of solutions to sub-problems^①. Once that is in place, we need to make sure that our formulation satisfies the *optimal substructure* property and, crucially, that also requires the solutions to the same sub-problems more than once (see Appendix 64). Only then we are ready to unleash the full power of DP.

Consider the denominations listed in $I = \{I_0 < I_1 < \dots\}$ and the function $C(x)$ which returns the minimum number of coins necessary to obtain the amount x using I . We can

^①The concept of sub-problem, in the context of this problem might seem quite fuzzy; You can think of it as a problem exactly equal to the main one except it operates on an input that is somehow “smaller” and it is therefore easier to solve. In this specific case it means t is smaller.

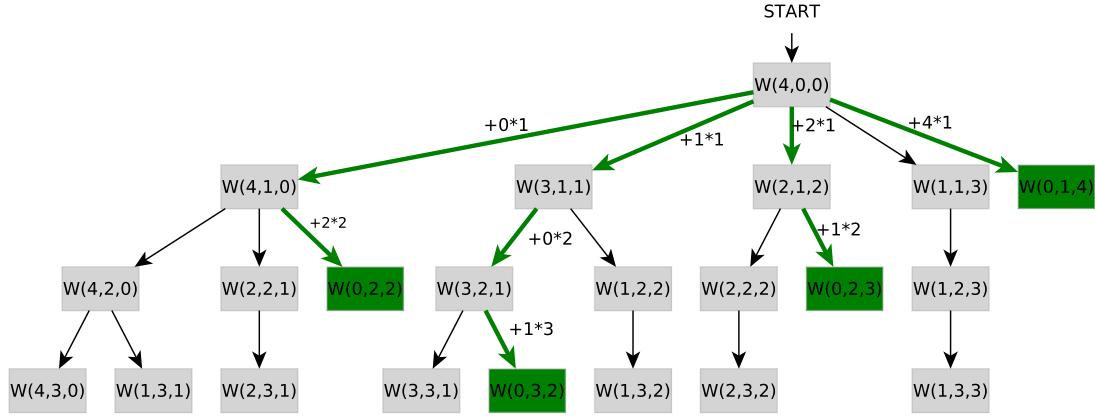


Figure 44.1: This figure shows the call tree for the recursive function `change_ways_bruteforce_backtracking_helper` on the following input: $I = \{1, 2, 3\}$, and $t = 4$. Each node contains the only three varying parameters of `change_ways_bruteforce_backtracking_helper` (shortened here as W): the first is the current t (the amount that we still need to make up for). The second is the index to an element of I for the denomination we are considering and the third is the number of coins used so far. Moreover, the highlighted paths shows all the valid ways of changing 4. Note that all the green nodes have the first number equal to zero.

calculate the value of $C(y)$ where $y > x$ very easily by using Equation 44.2:

$$\begin{cases} C(0) = 0 \\ C(y) = +\infty \text{ if } y < 0 \\ C(y) = 1 + \min_{d \in I} C(y - d) \end{cases} \quad (44.2)$$

We can see that, the answer for the amount y can be expressed in terms of answers to amount strictly smaller than y and in particular, when:

- $y = 0$, the answer is 0 as there is only one way of making up for the amount 0 i.e. using zero coins;
- $y < 0$ the answer is $+\infty$, signalling it is impossible to obtain a negative amount by only using positive denominations;
- in all the other cases, you can calculate the answer by using the answers to sub-problems for smaller amounts that you can obtain by subtracting the current amount with one of the coin denomination in I .

The key point here is that $C(y)$, as it is defined in Equation 44.2 satisfies the optimal substructure property. In-fact we can see that we can obtain the optimal answer to $C(y)$ from the optimal solution to smaller sub-problems.

Moreover, if we apply Equation 44.2 to the Example 44.4 we see that solution to sub-problems are required over and over again:

- $C(11) = \min(C(10), C(9), C(6))$
- $C(10) = \min(C(9), C(8), C(5))$
- $C(9) = \min(C(8), C(7), C(4))$
- $C(8) = \min(C(7), C(6), C(3))$
- $C(7) = \min(C(6), C(5), C(2))$
- ...

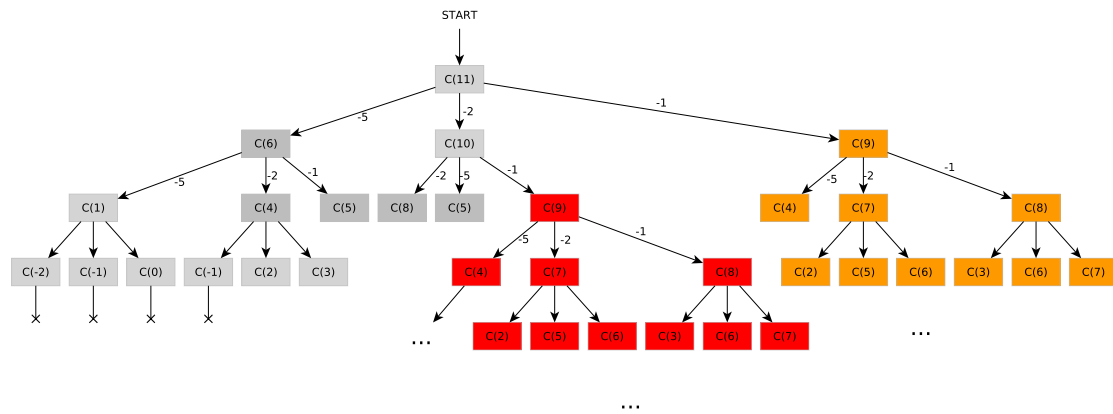


Figure 44.2: Initial layers of the recursion tree for $C(11)$

Figure 44.2 shows the initial layers of the recursion tree for $C(11)$, from which it is clear that the whole work described by the subtree $C(9)$ is done twice: once from $C(11)$ when using a coin of value 2 (red nodes) and a second time from $C(10)$ when using a coin of value 1 (orange nodes).

Therefore, it seems that this problem also satisfies the overlapping subproblem property and we can very easily turn Equation 44.2 into an efficient DP solution by translating it into a memoized recursive function implementation as shown in Listing 44.3. The code works by blindly following what dictated by Equation 44.2 with the only addition of the function `change_ways_DP_topdown_helper` being memoized via a cache, which takes the shape of a standard `std::unordered_map`.

```

1 using DPCache = std::unordered_map<int, int>;
2
3 int change_ways_DP_topdown_helper(const std::vector<int>& I,
4                                   const int t,
5                                   DPCache& cache)
6 {
7     if (t == 0)
8         return 0;
9     if (t < 0)
10        return std::numeric_limits<int>::max();
11
12     if (cache.contains(t))
13         return (cache.find(t))->second;
14
15     int ans = std::numeric_limits<int>::max();
16     for (const auto d : I)
17     {
18         ans = std::min(ans, change_ways_DP_topdown_helper(I, t - d, cache));
19     }
20     ans += 1;
21     cache.insert({t, ans});
22     return ans;
23 }
24
25 int change_ways_DP_topdown(const std::vector<int>& I, const int t)
26 {
27     DPCache cache;
28     return change_ways_DP_topdown_helper(I, t, cache);

```

Listing 44.3: Dynamic Programmin top-down solution.

The time complexity of Listing 44.3 is $\Theta(t|I|)$: There are $t + 1$ possible distinct calls to `change_ways_DP_topdown_helper` and each of them costs $\Theta(|I|)$. The space complexity is $O(t)$ (if you consider the additional space occupied the by the stack during the recursive process, otherwise it is constant); if $1 \in I$ then we get calls to `change_ways_DP_topdown_helper` for every value from t to 0.

44.3.5 Bottom-up

In Section 44.3.4 we have seen how it is possible to use DP to attach and solve this problem efficiently by adopting a top-down approach. All DP solutions can be also implemented in a bottom-up fashion where we explicitly fill in a DP table (T) starting with the known values, usually corresponding to the base cases of the recursion for the top-down formulation.

Let's start by isolating the values of t for which the solution is known. A good starting point seems to be the base cases of Listing 44.3 where $t = 0$ and we return 0 immediately. The next question we want to ask ourselves is, how can we fill cells of the DP table corresponding to higher values of t starting from the value for the cell at $t = 0$? The key idea here is that from a given t we can obtain all the amounts corresponding to: $t + I_0, t + I_1, \dots$, with a number of coins equal to the number of coins you needed to obtain t plus 1.

For instance if $I = \{1, 2, 5\}$ the DP table T initially is as follows: $T = \{0, +\infty, +\infty, \dots\}$. From the value 0 we can update cells for $t = 1, 2, 5$ with the value 1 and the the table becomes: $T = \{0, 1, 1, +\infty, +\infty, 1, +\infty, \dots\}$. We can now repeat the process from $t = 1$ and update all the values you can achieve from $t = 1$ i.e. 2, 3, 6. Notice that we can skip values 2 and 3 because they have already been obtained from the amount 1 with fewer coins. T is now: $T = \{0, 1, 1, 2, +\infty, 1, 1, +\infty, \dots\}$. This process can continue until we have finished processing all the values up to t and the final answer will be stored in the DP table cell for the amount t .

More generally, assuming the table is filled up to (and including) cell at index $x + 1$ (corresponding to the amount x) you can update the cell of T at index $0 \leq k < |I|$ as follows:

$$T_{x+I_k} = \min(T_{x+I_k}, T_x + 1)$$

Listing 44.4 shows an implmentation of this idea. The time and space complexities are both $O(|I| \times t)$.

```

1  int change_min_ways_bottom_up(const int amount, const vector<int>& coins)
2  {
3      constexpr int INF = std::numeric_limits<int>::max();
4      std::vector<int> T(amount + 1, INF);
5      T[0] = 0;
6
7      for (const auto c : coins)
8          for (int x = 0; x <= amount; x++)
9              if ((x + c) <= amount)
10                 T[x + c] = std::min(T[x + c], T[x] + 1);
11     return T[amount];
12 }
```

Listing 44.4: Dynamic Programmin bottom-up solution.

44.3.6 Conclusion

In this chapter we have seen how to solve the Coin change problem which is a classical DP problem.

The nice thing about this approach is that, we can reuse it virtually for any DP problem, provided we came up with a suitable recursive definition for the solutions to the subproblem. All it is necessary is to code such a definition into a recursive function and use memoization to save precious computation steps. You can see more examples of problems solved with a similar techniques in Sections 64,45.3.2, 43.3.2.1, 42.3.3, 49.3.1.1 and 31.2.2.2

44.4 Common Variations

44.4.1 Count the number of ways to give change.

There is a very common variation of this problem where you are asked to return the total count of the possible ways you can change a given amount t . The solution approach is the same and you can apply everything we have covered in this Chapter so far to solve this variant.

Problem 63 Write a function that given an array of coin denominations I and an integer t representing an amount of money, returns the number of ways you can make up that amount by using coins of the denomination specified by the array I . You have an infinite amount of coins of each denomination.

■ Example 44.4

Given $I = 1, 5, 10$ and $t = 8$, the function returns 2. We can change 8 in two distinct ways:

1. eight coins of denomination 1, or
2. three and one coin of denomination 1 and 5, respectively.

■

■ Example 44.5

Given $I = 2, 5, 3, 6$ and $t = 10$, the function returns 5. We can change 8 in the following ways:

1. five coins of denomination 2,
2. two and three coins of denomination 2 and 3, respectively,
3. two and one coin of denomination 2 and 6, respectively,
4. one coin of the denominations 2, 3 and 5, and finally,
5. two coins of denomination 5.

■

■

45. Number of Dice Rolls With Target Sum

Introduction

Dices have been around for centuries. The oldest known dice were discovered in Iran as part of a staggering five-thousand-year-old Backgammon^① set.

If we think about what a die is, the first thing that comes to mind is the fair 6 faced-die (see Figure 45.1b), but potentially any polyhedron can be a die: consider for instance a 20-faced icosahedron (see Figure 45.1c), or a 12-faced dodecahedron (see Figure 45.1a). In this chapter's problem, we will use up to 30 dice at the same time, each with up to 30 faces, and we are going to calculate the number of ways we can obtain a certain target value when we throw them all at the same time.

45.1 Problem statement

Problem 64 You have d dice, and each die has f faces numbered $1, 2, \dots, f$. Write a function that returns the number of possible ways to roll the dice so the sum of the upper faces equals a given target number t . Because this number can be very large, the answer should be returned modulo $10^9 + 7$.

■ **Example 45.1**

Given $d = 1$, $f = 6$ and $t = 6$ the function should return 1. Clearly, there is only one way to obtain 6 when rolling a common 6-face cubic die. ■

■ **Example 45.2**

Given $d = 2$, $f = 6$ and $t = 7$ the function should return 6. Table 45.1 lists all the possible ways of obtaining 7 from two common 6-face dice. ■

■ **Example 45.3**

Given $d = 2$, $f = 3$ and $t = 7$ the function should return 0 because the highest number obtainable by rolling two dices with three faces is 6. ■

45.2 Clarification Questions

Q.1. What is the maximum number of dices, faces, and the highest target value possible?

30, 30 and 1000, respectively.

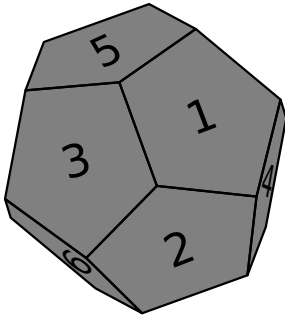
45.3 Discussion

Let's start by noting that the answer can be astronomically high, but more importantly, that the number of possible value combinations resulting from rolling d dices is even larger.

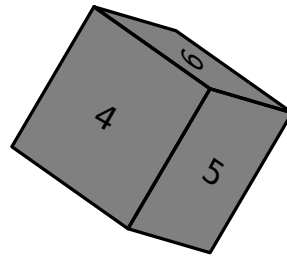
^①One of the oldest known board games, Backgammon is a two-player game where pieces are moved around between twenty-four triangles according to the roll of two 6 faced-dice. The goal of each player is to remove all of their 15 pieces before the other player

First die	Second die
1	6
2	5
3	4
4	3
5	2
6	1

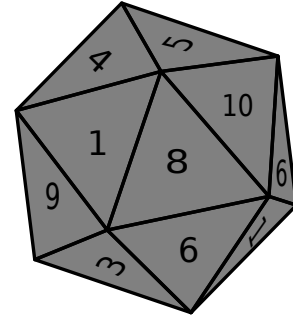
Table 45.1: Possible valide arrangements of two dice for the Example 45.2



(a) Example of dice with 12 faces.



(b) Example of common 6 faces dice.



(c) Example of dice with 20 faces.

If each die has f faces, then we are looking at f^d possible distinct roll outcomes! During an interview, a brute-force approach, where we go over each and every possible roll outcome of the d dice, is completely out of question (considering the constraints on the maximum number of dices and faces we might get for input) unless we are willing to wait around 6×10^{18} years. Even if we implement this algorithm so that it can run on the fastest supercomputer available today (which is capable of a staggering ≈ 450 ^② operations per second), it would still require $\frac{30^{30}}{10^{15}}$ s to run to completion. By that time, humanity will be long gone, the universe will be a dark and cold place, but most importantly, the position you are dreaming of will have gone to somebody else.

This type of "counting" questions is usually (and crucially more efficiently) solved by using a dynamic programming approach. In fact, this question shares a lot of similarities with the classical dynamic programming *Coin change* problem, to the point that we could solve this one using the solution to the other. In fact we can stretch this reasoning so as to consider the problem addressed in this chapter to be a proper specialization of the *Coin change* problem, where the number of available coins is equal to d and the denomination of the coins are $1, 2, \dots, f$: we have coins of the same denomination as the dice faces.

45.3.1 Brute-force

For science's sake, let's see how a brute-force approach would be applied here. As we all know, a brute-force approach evaluates all possible outcomes of rolling d dice and keeps track of how many yield a total sum of t . When dealing with this kind of task, where you have to enumerate/generate the elements of a given set, recursion is usually the way to go, especially if the elements (in this case a combination of face values) have a recursive definition. In this specific case, we can generate all possible combinations of d faces we

^②**petaflop**: A petaflop is a measure of a computer's processing speed and can be expressed as: A quadrillion (thousand trillion) floating-point operations per second (FLOPS). A thousand teraflops. 10 to the 15th power FLOPS. 2 to the 50th power FLOPS. A huge number of operations per second.

can obtain from rolling d dices by doing the following:

- generate the combinations from rolling $d - 1$ dice;
- create f copies of them;
- prepend 1 to the items of the first copy;
- prepend 2 to the items of the second copy;
- ...
- prepend f to the items of the last copy.

For instance, we can generate the all rolls outcome of 3 six-faced dice by:

1. Generate all the outcomes for only 2 of them C_2 :

$$C_2 = \{(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4),$$

$$(3, 1), (3, 2), (3, 3), (3, 4), (4, 1), (4, 2), (4, 3), (4, 4)\}$$
2. Append 1 to each of the elements of C_2 :

$$C_3^1 = \{(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 1, 4), (1, 2, 1), (1, 2, 2), (1, 2, 3), (1, 2, 4),$$

$$(1, 3, 1), (1, 3, 2), (1, 3, 3), (1, 3, 4), (1, 4, 1), (1, 4, 2), (1, 4, 3), (1, 4, 4)\}$$
3. Append 2 to each of the elements of C_2 :

$$C_3^2 = \{(2, 1, 1), (2, 1, 2), (2, 1, 3), (2, 1, 4), (2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 2, 4),$$

$$(2, 3, 1), (2, 3, 2), (2, 3, 3), (2, 3, 4), (2, 4, 1), (2, 4, 2), (2, 4, 3), (2, 4, 4)\}$$
4. Append 3 to each of the elements of C_2 :

$$C_3^3 = \{(3, 1, 1), (3, 1, 2), (3, 1, 3), (3, 1, 4), (3, 2, 1), (3, 2, 2), (3, 2, 3), (3, 2, 4),$$

$$(3, 3, 1), (3, 3, 2), (3, 3, 3), (3, 3, 4), (3, 4, 1), (3, 4, 2), (3, 4, 3), (3, 4, 4)\}$$
5. Prepend 4 to each of the elements of C_2 :

$$C_3^4 = \{(4, 1, 1), (4, 1, 2), (4, 1, 3), (4, 1, 4), (4, 2, 1), (4, 2, 2), (4, 2, 3), (4, 2, 4),$$

$$(4, 3, 1), (4, 3, 2), (4, 3, 3), (4, 3, 4), (4, 4, 1), (4, 4, 2), (4, 4, 3), (4, 4, 4)\}$$
6. Finally, return $C_3 = \{C_3^1 \cup C_3^2 \cup C_3^3 \cup C_3^4\}$

The definition above is correct but not very useful in practice. It requires making many copies of a potentially very large (indeed exponential) set of items. We will therefore use a different approach that will still run in exponential time (this section is named brute-force after all) but that can be used as a basis for developing a more efficient DP solution.

We start by rolling the first die. Clearly we have f possible values we can get, but once the value for this specific die is set (say we got the value x) we are left with $d - 1$ dices to roll and we still have to make up for $t - x$ with the remaining $d - 1$ dice in order to reach our target value t . Once a die is rolled, we are left with exactly the original problem on a smaller number of dice and target value. This is why recursion is handy as we can describe the solution to the entire problem in terms of solutions to sub-problems. We can continue this recursive process - rolling one die at a time - until we reach one of the following cases:

1. $d < 0$ or $t < 0$ the answer is 0. There is no solution to the problem when the number of dice to use is negative or the target number is negative.
2. $t = 0$. We have reached the target value t . If we have used **all** dice then we have a solution, otherwise we do not. In other words:
 - if $d = 0$, we have used all d dice and the sum is exactly equal to t . This is a valid combination. We have rolled d dice and the sum of their faces is exactly equal to t .
 - if $d > 0$, we have not rolled all the dice, yet we have already reached our target value. If we continue to roll, we will generate a combination with a total sum higher than t . This is not a good combination.

The idea above can be better expressed using the recurrence relation shown in Equation 45.1 where $S(d, t, f)$ is the number of ways one can obtain a target value t by throwing d dice. Note that the third parameter never changes and thus it does not play a dynamic

role in the recurrence.

$$S(d, t, f) = \begin{cases} 1 & \text{if } d = t = 0 \\ 0 & \text{if } d = 0, t > 0 \\ \sum_1^{\min(f, t)} S(d-1, t-j, f) & \text{otherwise} \end{cases} \quad (45.1)$$

Listing 45.1 shows a possible implementation of such idea. Please note that this code is remarkably similar to the brute-force solution for the Coin Change problem in Chapter 44. Section ?? also discusses the problem of generating combinations and the material discussed there can be adapted and applied here.

```

1  int num_rolls_to_target_bruteforce(const int dices,
2                                     const int f,
3                                     const int target)
4  {
5      constexpr unsigned long MOD = 1e9 + 7;
6      // ops. overreached
7      if (target < 0 || dices < 0)
8          return 0;
9
10     // no more die to roll. Have we reached the target value?
11     if (dices == 0)
12         return target == 0 ? 1 : 0;
13
14     // for each possible face
15     int ans = 0;
16     for (int i = 1; i <= f; i++, ans %= MOD)
17     {
18         // we assume we rolled the face with value i and solve the associated
19         // subproblem
20         ans += num_rolls_to_target_bruteforce(dices - 1, f, target - i);
21     }
22     return ans;
23 }
```

Listing 45.1: Brute-force (enumerating all possible combinations) solution for the problem of counting the number of dice rolls summing up to a target number t .

The time and space complexity of this approach are exponential and constant, respectively.

The proof of this can be derived from the solution of the recurrence relation shown in Equation 45.2:

$$S(d, t) = S(d-1, t-1) + S(d-1, t-2) \quad (45.2)$$

where $S(d, t)$ expresses the number of invocations of the function `num_rolls_to_target_bruteforce` for a given number of dice d and target value t when $f = 2$. The resulting invocation tree is complete and has height $h = t$ (assuming $d \leq t$, but the same reasoning can be applied in the other cases). The cost of each node of the tree is $O(1)$. The number of nodes in such a tree is exponentially proportional to its height.

45.3.2 Dynamic Programming - Recursive top-down

The brute-force solution we laid down in Section 45.3.1 can be turned into a nice and efficient one with the help of DP, and in particular of memoization. As with many other

problems solvable with DP, the brute-force solution above can be turned into a much more efficient one by simply realizing that the same sub-problems are solved over and over again. For instance, imagine the case where $f = 5$. We might end up solving the sub-problem where $d = 1$ and $t = 3$ from the sub-problem where $d = 2$, $t = 4$ (by rolling the face with 1), or from the sub-problem $d = 2$, $t = 5$ (by rolling the face with 2).

However, the maximum number of *distinct* invocations for the function `num_rolls_to_target_bruteforce` is not larger than $30 \times 1000 = 30000$: the maximum number of dice multiplied by the largest target value as these are the only function parameters varying during the execution. If we can somehow guarantee that no duplicate work is done for a given d and t , then we can get away with only $O(d \times t)$ function calls.

Consider, for instance, what happens during the execution of the code in Listing 45.1 for the following input:

- $d = 3$
- $f = 6$
- $t = 12$

The function `num_rolls_to_target_bruteforce(1,5,6)` is called 6 times and the sub-problem `num_rolls_to_target_bruteforce(1,4,6)` is solved 5 times. You can verify this yourself if you draw the recursion tree for `num_rolls_to_target_bruteforce(3,6,12)` or simply add a print statement at the beginning of the function. All these duplicate calls are superfluous and they represent work that can be saved if the result of each sub-problem is stored in a cache as shown in the following Listing.

The function subproblem `num_rolls_to_target_bruteforce(1,5,6)` is solved 6 times and the subproblem `num_rolls_to_target_bruteforce(1,4,6)` is solved 5 times. All these superfluous execution can be avoided if the result of each of the subproblem is stored into a cache as shown in Listing 45.2.

```

1 using Cache = std::vector<std::vector<unsigned long>>>;
2
3 int num_rolls_to_target_memoization_helper(const int target,
4                                           const int dices,
5                                           const int f,
6                                           Cache& cache)
7 {
8     constexpr unsigned long MOD = 1e9 + 7;
9
10    if (target < 0 || dices < 0)
11        return 0;
12
13    if (dices == 0)
14        return target == 0;
15
16    if (cache[dices][target] != -1)
17        return cache[dices][target];
18
19    unsigned long ans = 0;
20    for (int i = 1; i <= f; i++, ans %= MOD)
21        ans +=
22            num_rolls_to_target_memoization_helper(target - i, dices - 1, f, cache)
23            ;
24    cache[dices][target] = ans;
25    return ans;
26 }
27
28 int num_rolls_to_target_memoization(int d, int f, int target)
29 {

```

```

30     Cache DP(d + 1, std::vector<unsigned long>(target + 1, -1));
31     return num_rolls_to_target_memoization_helper(target, d, f, DP);
32 }

```

Listing 45.2: Dynamic programming with memoization top-down recursive solution for the problem of counting the number of dice rolls summing up to a target number t .

This implementation is an almost identical copy of the brute-force solution, except for the addition of a cache. Note how **before** actually trying to compute the answer we first look into the cache (see highlighted lines) to see if it is already present in the cache. If not, we solve the problem and before returning the answer we **save** the result in the cache.

45.4 Dynamic programming - Iterative bottom-up

Turning the top-down-up solution shown in Section 45.3.2 into a bottom-up one is relatively easy. The recursive definition of the solution (see Equation 45.1) clearly shows that we can calculate the answer for a given number of dice d and a target value t if we have already calculated the values for targets $t-1, t-2, \dots, t-f$ and $d-1$. We also know how to easily calculate the answer for all possible target values when $d=0$ and $d=1$ and from there we can apply the reasoning above. This is exactly the strategy that Listing 45.3 implements.

```

1  using Cache = std::vector<std::vector<unsigned long>>;
2
3  int num_rolls_to_target_bottom_up(const int d, const int f, const int target)
4  {
5      Cache DP(d + 1, std::vector<unsigned long>(target + 1, 0));
6
7      for (int j = 1; j <= f && j <= target; j++)
8      {
9          // only one way to make a given value with 1 dice
10         DP[1][j] = 1;
11     }
12
13     // 1 way to make 0 with 0 dice
14     DP[0][0] = 1;
15
16     // num dices
17     for (int i = 2; i <= d; i++)
18     {
19         // target value
20         for (int t = 1; t <= target; t++)
21         {
22             // face value for the ith die
23             for (int j = 1; j <= f && j <= t; j++)
24             {
25                 DP[i][t] += DP[i - 1][t - j];
26             }
27         }
28     }
29     return DP[d][target];
30 }

```

Listing 45.3: Dynamic programming bottom-up solution for the problem of counting the number of dice rolls summing up to a target number t .

We use a 2D table `DP` (initialized with zeros) with $d+1$ rows and $t+1$ columns where each cell `DP[i][j]` corresponds to the solution of a sub-problem where $d=i$ and $t=j$. The first loop takes care of filling the table with "known" values for all sub-problems where $d=1$. If we only have a die with f faces, there is only one way we can achieve the target

values $1, 2, \dots, f$ and no way to obtain any higher values. The rest of the code fills the table one row at a time (one die at a time) by using the values of the previous row.

The time and space complexity of this algorithm are $\Theta(dtf)$ and $\Theta(dt)$, respectively. However, the space complexity can be easily lowered to $\Theta(t)$ because, as already mentioned, we only need space for two rows of the DP table: one for the values of the current d and one for the values at $d - 1$.

46. Remove duplicates in sorted array

Introduction

Sorting and duplicates are the bread and butter of coding interview questions. There are countless problems that ask you to perform some task, or calculate an answer, where you are given either some form of sorted input or there are duplicates involved.

In the problem described in this chapter, we are going to investigate how we can remove duplicates from an already sorted collection of elements. This problem is easily solvable when you can use linear space but doing it *in-place* and by only using constant space is slightly more challenging.

46.1 Problem statement

Problem 65 Write a function that takes a sorted array I as input and returns the number of unique elements u in it. The function should also cause all the unique elements of I to appear in the first u positions.

■ **Example 46.1**

Given $I = \{1, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6, 6, 7\}$ the function returns 7 and I is rearranged such that its first 7 elements are $\{1, 2, 3, 4, 5, 6, 7\}$. ■

■ **Example 46.2**

Given $I = \{1, 2, 3, 4\}$ the function returns 4 and I is rearranged such that its first 4 elements are $\{1, 2, 3, 4\}$. ■

46.2 Clarification Questions

Q.1. Is the input array guaranteed to contain integers?

Yes you can assume I is an array of integers, but only if you are free to produce a generic solution that works for any type.

46.3 Discussion

This problem behavior is remarkably similar to the function `std::unique` from the STL library: it does not really remove any element from the input collection, instead, it rearranges the elements to divide the initial collection. The official documentation for `std::unique` says that:

It eliminates all except the first element from every consecutive group of equivalent elements from a range and returns a past-the-end iterator for the new logical end of the range. Removing is done by shifting the elements in the range in such a way that elements to be erased are overwritten. The relative order of the elements that remain is preserved and the physical size of the container is unchanged. Iterators pointing to an element between the new logical end and the physical end of the range are still dereferenceable, but the elements themselves have unspecified values.

As we can see, `std::unique` does not really remove or erase any element from the input collection. What it does instead is rearrange the elements such that the initial collection is divided into two parts:

1. the first (from the left) containing only the unique elements;
2. the second where the duplicate elements are moved to (possibly empty).

This function is often used in real-life applications paired with `std::erase` to delete the second part of the newly arranged collection when you actually want the duplicates removed.

Listing 46.1 shows how we can solve this problem with a one-liner solution using `std::unique` and `std::distance` from the STL.

```
1 template <typename T>
2 int remove_duplicates_STL(std::vector<T>& I)
3 {
4     return std::distance(std::begin(I), std::unique(std::begin(I), std::end(I)));
5 }
```

Listing 46.1: One-liner solution using `std::unique`.

The code works by first invoking `std::unique` and the entire array, which causes `I` to be split into two parts as described above and returns an iterator to the first element of the second part. `std::distance` is then used to calculate the number of elements in the first part which is the final answer.

Being able to show you can use the standard library to solve a relatively complex problem is something any interviewer is going to appreciate, however, as important as making a good first impression is, this is unlikely to be enough to clear the interview round entirely. If you use this approach during an actual interview, the interviewer is likely to ask you to implement `std::unique` and `std::distance` yourself.

46.3.1 Linear space solution

As mentioned in the introduction, it is quite easy to implement this problem when you can use linear additional space. You can think of building a list U of unique elements of I by:

1. insert the first element of I ;
2. insert at the back of U every element of I that is not equal to the last element of U .

It is important to note that U does not at any moment contain duplicates. Eventually, at the end of this process, U contains an ordered list of all the unique elements in I . All we have to do is copy U into the first $|U|$ positions of $|I|$ and return $|U|$. The complexity of this approach is linear in time and space, as in the worst-case scenario (when I does not contain duplicates) we move the entire array I into U , and then immediately copy U back into I . An implementation of this approach is shown below in Listing 46.2.

```
1 template <typename T>
2 int remove_duplicates_linear_space(std::vector<T>& I)
3 {
4     const auto num_elements = I.size();
5     if (num_elements <= 1)
6         return num_elements;
7
8     std::unordered_set<T> inserted;
9     std::vector<T> I_uniques;
10    for (const auto& x : I)
11    {
12        if (!inserted.contains(x))
13        {
```

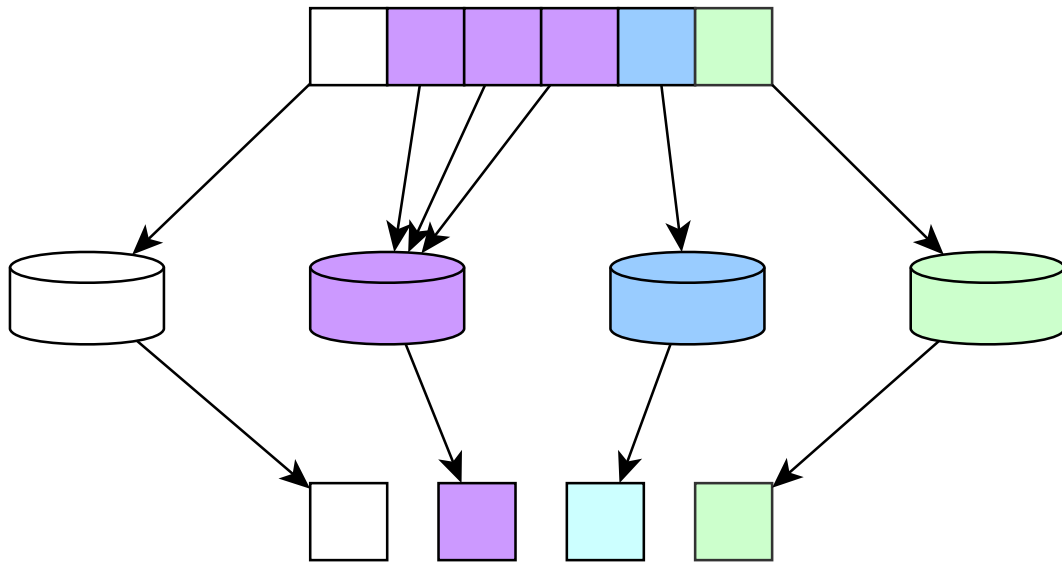


Figure 46.1

```

14     inserted.insert(x);
15     I_uniques.push_back(x);
16 }
17 }
18 std::copy(std::begin(I_uniques), std::end(I_uniques), std::begin(I));
19 return I_uniques.size();
20 }

```

Listing 46.2: Linear time and space solution using `std::unordered_set` to remember what elements have been already encountered.

46.3.2 Constant Space

Although we cannot do much better than spending linear time we can improve on the space used to the point where we only need a constant amount of it. The key idea is that, because the array is sorted, equal elements will be next to one another, therefore forming clusters of the same value. Eventually, I has to be logically divided into two subarrays where we only care about the content of the first part containing unique elements (no duplicates), as there are no constraints on the content of the second part.

The algorithm proposed in this section uses a two pointer technique with which we build the first half of I one element at a time by looping through the elements of I and keeping track of two pointers:

- x : a pointer to the last element of the first part of I ;
- y : a pointer to the next element to be processed.

When the element pointed by y is different from the element pointed by x , we know that we can add y to the first part of I . We can do that by copying I_y into I_{x+1} and incrementing both y and x so that the next comparison would be among the last inserted element and a brand new unprocessed one. If they are equal, however, the first part of I is not going to grow and we can safely ignore the element pointed by y as we already have an instance of it (the value pointed by x) in the first half of I already.

When all the elements of I are processed ($y \geq |I|$) then the algorithm can be stopped.

At this point we know that x is marking the end of the part of I containing only unique elements. All we have to do is calculate and return its length.

Listing 46.3 shows an implementation of this idea.

```
1  template <typename T>
2  int remove_duplicates_constant_space(std::vector<T>& I)
3  {
4      const auto num_elements = I.size();
5      if (num_elements <= 1)
6          return num_elements;
7
8      int x = 0, y = 1;
9      while (y < num_elements)
10     {
11         if (I[x] != I[y])
12             I[++x] = std::move(I[y]);
13         y++;
14     }
15     return x + 1;
16 }
```

Listing 46.3: Linear time constant space solution.

Note that with x we have two important invariants:

1. x there are no duplicates among all the elements to the left of (including) x ;
2. y is always larger than x .

These invariants are true prior to entering the `while` loop and they are true at the end of each and every iteration. It is also essential to note that the cells strictly between x and y can be overwritten as they must contain duplicates.

As already stated at the beginning of this section, the time complexity is linear but now, as opposed to the other solutions discussed so far, we use only constant space.

Moreover, because we do not have to care about the state of the elements of I after x , we can use `std::move`^① to (potentially) avoid expensive copies.

Figure 46.2 depicts the execution of this algorithm on the input of Example 46.1, where the shaded part (the left side) of the array contains all the unique elements found so far (among all the elements to the left of y): x is a pointer to the last element of this sequence and y is a pointer to the element currently processed.

46.4 Common Variations

46.4.1 Max 2 duplicates allowed

In this section, we will have a look at a common variation of the main problem of this lesson, which differs from it on the basis that each element can now appear at **most twice** in the final rearrangement of I .

Problem 66 Write a function that - given a sorted array I - removes all the duplicates in such a way that an element appears **at most twice** and with all the valid elements being located at the beginning of I itself. The function returns the number of valid elements in I .

■ Example 46.3

Given $I = \{1, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6, 6, 7\}$ the function returns 11 and I is rearranged such

^① `std::move` to indicate that an object t may be “moved from”, i.e. allowing the efficient transfer of resources from t to another object without the need for an explicit copy.

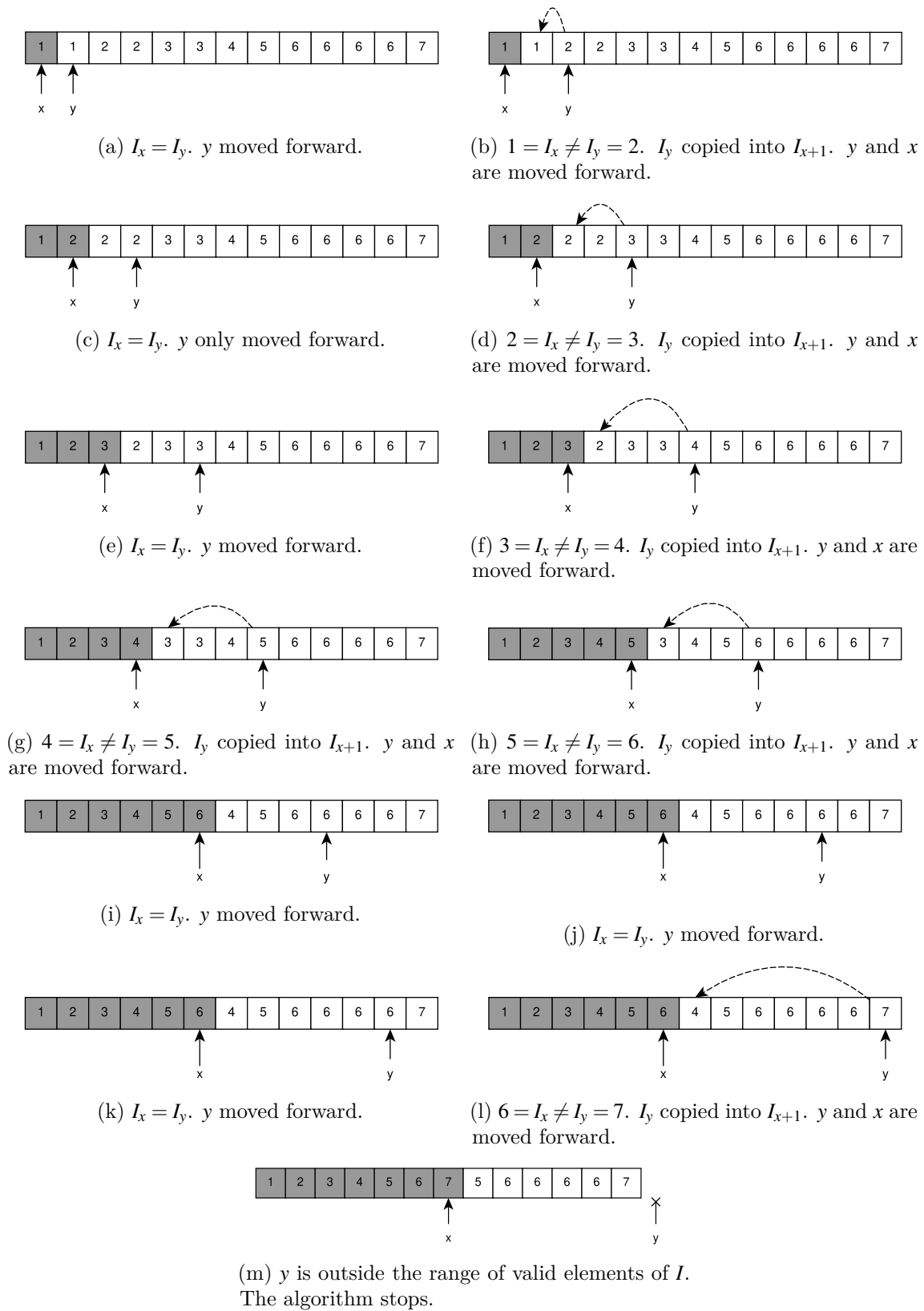


Figure 46.2: Execution of the algorithm implemented in Listing 46.3 on the input of the Example 46.1. The shaded part of the array contains all the unique elements processed so far. x is a pointer to the last element of this sequence. y is a pointer to the element currently processed.

that itself first 11 elements are {1,1,2,2,3,3,4,5,6,6,7}. ■

■ Example 46.4

Given $I = \{1,2,3,4\}$ the function returns 4 and I is rearranged such that its first 4 elements are {1,2,3,4}. ■

46.4.2 Discussion

This variant can be solved with minimal changes to the solution presented for the main problem. We can modify the code shown in the Section 46.3.2 so that we keep track of the number of repetitions we have already inserted for a given element. This can be implemented as shown in Listing 46.4.

```
1  /**
2   * @input A : Integer array
3   * @input n1 : Integer array's ( A ) length
4   *
5   * @Output Integer
6   */
7  int removeDuplicates(int* A, int n1)
8  {
9      if (n1 <= 1)
10         return n1;
11
12     int x          = 0;
13     int y          = 1;
14     int consecutive = 1;
15     while (y < n1)
16     {
17         if (A[x] == A[y] && consecutive == 1)
18         {
19             A[++x] = A[y];
20             consecutive = 2;
21         }
22         if (A[x] != A[y])
23         {
24             A[++x] = A[y];
25             consecutive = 1;
26         }
27         y++;
28     }
29     return x + 1;
30 }
```

Listing 46.4: Linear time constant space solution to the variation where at most two duplicates are allowed.

Note that the meaning of the variables x and y did not change, and that here we use the variable `consecutive` to keep track of the number of times the element pointed by x appears in the array A . If the element pointed by y is equal to the element pointed by x (we have a duplicate), then we decide whether to insert it or not based on the value of the variable `consecutive`:

- If it appears already more than 1 times we discard it;
- otherwise, we copy it to the cell at index $x+1$ and increment `consecutive`.

The time and space complexity of this approach is $O(|I|)$ and $O(1)$, respectively.

46.4.3 Max k duplicates allowed

This variation is also a quite common and is basically a generalization of the problems above where now each element can appear k times. Note that when $k = 1$ and $k = 2$ this problem is equivalent to the Problems 65 and 66. The solution for this variation is not discussed here as it can be easily derived from the solution to the Problem 65.

Problem 67 Write a function that - given a sorted array I - removes all the duplicates in such a way an element appears at most twice and with all the valid elements being located at the beginning of the I itself. The function returns the number of valid elements in I .

■ **Example 46.5**

Given $I = \{1, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6, 6, 7\}$ and $k = 3$ the function returns 12 and I is rearranged such that its first 12 elements are $\{1, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6, 7\}$. Notice the extra 6 w.r.t. the Example 46.3. ■

■ **Example 46.6**

Given $I = \{1, 1, 1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 4, 4\}$ and $k = 5$ the function returns 13 and I is rearranged such that its first 13 elements are $\{1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 4, 4\}$. ■

■

47. Remove all occurrences - unsorted array

Introduction

The problem covered in this chapter asks us to implement a common operations: removing all elements satisfying specific criterium from a collection. This problem has many similarities to the one discussed in Chapter 46 and as a consequence they share the same general approach to their solution.

There are many variations of this problem but the most common being where the collection is a simple array or a vector of integers and we are asked to remove all the elements equal to a given integer. On this occasion, however, we will discuss a more generalized version where the collection is of a generic type T and the criterium is given in the form of a unary function returning a boolean^①.

If you are asked to solve this particular problem version during an interview, you should be able to easily specialise what is discussed here in the moment.

47.1 Problem statement

Problem 68 Write a function that - given a collection I of elements of type T and a predicate function p with signature `bool(const T&)` - rearranges I in such a way that all the $0 \leq k \leq |I|$ elements satisfying p in I are moved to the front. The function should return k .

Moreover, the relative order of the elements satisfying p should be preserved. If both elements at indices n and m satisfy the predicate p and I_n comes before I_m then when the function returns, their relative order is unchanged albeit they both might be moved to new locations.

■ Example 47.1

Given $I = \{4, 1, 1, 2, 1, 3\}$ and a function p returning true if its input argument is an even number, false otherwise, the function returns 4. The first 4 elements of I are $\{1, 1, 1, 3\}$. ■

■ Example 47.2

Given $I = \{4, 1, 1, 2, 1, 3\}$ and a function p returning true if its input argument is odd, the function returns 2. At this point, the first 2 elements of I are $\{4, 2\}$. ■

47.2 Clarification Questions

Q.1. What should the content be of I from index $k+1$ and after?

There are no constraints on the content of those cells of I .

^①This type of function is commonly known as *predicates*.

47.3 Discussion

This problem could be restated as: *Implement the `remove_if`^② function from the C++ STL.* So it would not be surprising if, during an interview, it could come-up as a one-liner as shown in Listing 47.1^③.

```
1 template <typename T>
2 int remove_elements_unsorted_array_remove_STL(std::vector<T>& A, auto predicate
3 )
4 {
5     return std::distance(std::begin(A),
6                          std::remove_if(std::begin(A), std::end(A), predicate));
7 }
```

Listing 47.1: One-liner solution using the STL functions `distance` and `remove_if`

This algorithm has a linear time and constant space complexity, which is pretty much as good as it gets considering you must at least read all the elements in the input array. It is worth mentioning, however, that if you present this solution to an interviewer you can expect to be asked to implement the logic behind `std::remove_if` itself as this is the core of the problem.

47.3.1 Linear time and linear space solution

There is a straightforward way of solving this problem that has the added benefit of occupying only a couple of lines and of being very clear and simple. The idea is that we can use an additional linear amount of space to temporarily store the valid (dissatisfying the predicate `p`) elements, and in a subsequent phase move them to the front of `I`. Listing 47.2 shows a possible implementation of this idea.

```
1 template <typename T>
2 int remove_elements_unsorted_array_linear_space(std::vector<T>& A,
3                                                  auto predicate)
4 {
5     std::vector<int> temp;
6     std::copy_if(std::begin(A),
7                 std::end(A),
8                 std::back_inserter(temp),
9                 std::not_fn(predicate));
10    std::copy(std::begin(temp), std::end(temp), std::begin(A));
11    return temp.size();
12 }
```

Listing 47.2: Linear space solution using the `std::copy` family of functions from the STL.

This solution only works correctly for types that can be copied. As such, the interviewer could ask you to fix this; in which case you can loop over `I` and `temp` and `move`^④ the elements around instead.

47.3.2 Linear time and constant space solution

The idea discussed in Section 47.3.1 can quite easily be modified so that we avoid using additional linear space. We could use `std::move` the elements of `I` into `I` itself in the same way as we did for the problem covered in Chapter 46 while discussing the solution 46.3.

Here we use exactly the same approach of two pointers `x` and `y`:

^②<https://en.cppreference.com/w/cpp/algorithm/remove>

^③Notice the similarities with Listing 46.1 for the problem in Chapter 46

^④`std::move` is used to indicate that an object `t` may be “moved from”, i.e. allowing the efficient transfer of resources from `t` to another object without the need for an explicit copy.

1. x keeps track of the new front to I . It points to the end of the portion of I (starting at index 0 and ending at $x - 1$) containing all the valid elements found so far;
2. y is a pointer to the next element not yet processed in I .

Listing 47.3 implements this idea.

```
1 int remove_elements_unsorted_array(std::vector<int>& A, auto predicate)
2 {
3     size_t x = 0;
4     const auto size = A.size();
5     while (!predicate(A[x]) && x < size)
6         x++;
7     size_t y = x + 1;
8     while (y < size)
9     {
10         if (!predicate(A[y]))
11         {
12             A[x] = A[y];
13             x++;
14         }
15         y++;
16     }
17     return x;
18 }
```

Listing 47.3: Constant space solution using a two pointer approach.

At the beginning of the execution the algorithm moves x forward. All the valid elements that are already at the front of I stay untouched as they are already in the right locations. At this point, x points either to the first invalid element in I or to one element past I . In the second scenario, there is no more work to do. All the elements are valid to begin with, and the second `while`

will not even start. I is left unchanged. In the first scenario, we will use y to scan the remaining elements of I past x , and we move each valid element we encounter into the location pointed by x . When this happens x is moved forward as the portion of valid elements grew by one element.

Notice that the invariant $x \leq y$ is always respected as:

- it holds before the beginning of the loop;
- x is incremented at the same rate or less compared to y .

At the end of this process, we are left with y pointing to the one element past I and x pointing to one cell after the last valid element of the newly rearranged I .

48. Sort the chunks, sort the array.

Introduction

Sorting is a popular topic in computer science and programming interviews. Its usefulness is beyond dispute and there are countless research papers and algorithms devoted to the topic.

In this problem however, we are not going to devise a novel sorting algorithm. Instead we will investigate how we can sort an entire array by sorting a number of its sub-arrays. The idea is that we want to be able to split an array into pieces such that if each of the pieces is sorted individually then the final result is equivalent to having sorted the entire array.

It is necessary to uncover a key insights to solve this problem efficiently. As such, asking the right questions and looking at a certain number of good examples is fundamental. In the next section we will explore how these insights can be gained and then turned into efficient code.

48.1 Problem statement

Problem 69 Write a function that - given an array I of integers - returns the maximum number of sub-arrays (or chunks) of I such that , if each of the sub-array is sorted individually, then I as a whole is sorted.

■ **Example 48.1**

Given $I = \{45, 88, 1, 9, 90\}$ then the function return 1.

■ **Example 48.2**

Given $I = \{4, 3, 2, 1, 5, 9, 10\}$ then the function return 4. We can sort the following sub-arrays: • $[0, 3]$ • $[4, 4]$ • $[4, 4]$ • $[4, 4]$

48.2 Clarification Questions

Q.1. Can the chunks overlap?

No. If you choose to sort two sub-arrays of I $s_1 = [p, q], p \leq q$ and $s_2 = [x, y], x \leq y$ then either $x > q$ or $p > y$.

48.3 Discussion

48.4 Brute-force

Let's start our discussion by considering a brute-force solution. One possible approach would be to try to divide the array into $|I|$ non-empty and non-overlapping parts (only one way of performing such division), sort them individually and then check if I is sorted. If it is not, then we can try to divide I into $|I| - 1$ sub-arrays, and check whether by

sorting the resulting individual pieces I turns out to be sorted. This line of reasoning can be generalized producing a general brute-force approach that works by progressively trying to split I into less and less numbers of sub-arrays $k < |I|$. For each of the possible valid divisions of I into k sub-arrays, we can then check whether we can obtain a complete sorting of I by only sorting the individual k sub-arrays. Eventually when $k = 1$, I would be fully sorted as this is equivalent to sorting I entirely.

Clearly this algorithm is complete and correct as all possible valid partitions of I are checked. Its complexity is however exponential in time as, given a certain k , there are $\binom{n}{k}$ possible ways we can divide I into k sub-arrays. Listing 48.1 shows a C++ implementation of such idea.

```

1
2 void sort_subarrays(auto begin, auto end, std::vector<int>& subarrays)
3 {
4     auto it = begin;
5     for (const auto n : subarrays)
6     {
7         auto itn = begin + n + 1;
8         std::sort(it, itn);
9         it = itn;
10    }
11    std::sort(it, end);
12 }
13
14 bool check_all_k_combinations(const std::vector<int>& I,
15                               const int offset,
16                               const int k,
17                               std::vector<int> combination)
18 {
19     if (k == 0)
20     {
21         auto I_copy(I);
22         sort_subarrays(std::begin(I_copy), std::end(I_copy), combination);
23         return std::ranges::is_sorted(I_copy);
24     }
25
26     for (int i = offset; i < I.size() - k; ++i)
27     {
28         combination.push_back(i);
29         if (check_all_k_combinations(I, i + 1, k - 1, combination))
30             return true;
31         combination.pop_back();
32     }
33     return false;
34 }
35
36 int max_chunks_to_sorted_bruteforce(const std::vector<int>& I)
37 {
38     for (int k = std::ssize(I); k >= 2; k--)
39     {
40         std::vector<int> splitting_points{};
41         if (check_all_k_combinations(I, 0, k - 1, splitting_points))
42         {
43             return k;
44         }
45     }
46     return 1;
47 }

```

Listing 48.1: Bruteforce solution to the problem *Sort the chunks, sort the array*.

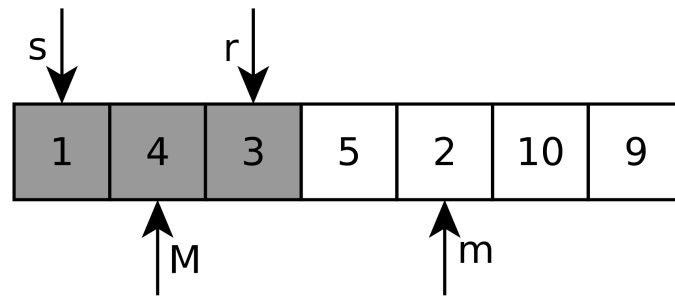


Figure 48.1: Input array from the Example 48.1. If the sub-array identified by s and r is sorted in isolation, I as a whole cannot be sorted (no matter how elements from r to $|I| - 1$ are sorted) because M will always appear before m , despite the fact $M > m$ and it should appear after.

48.5 Linear time

This problem can be sorted in linear time if we consider that if we sort a sub-array of I containing elements from index s to $r < |I| - 1$ (there is at least an element after this sub-array) then I cannot be fully sorted if the maximum element among the elements I_s, I_{s+1}, \dots, I_r is smaller than any of the elements of I after r (or smaller than the smallest among those elements after r). For instance, imagine I is the input array of the example 48.2. If we choose $s = 0$ and $r = 2$ then I can never be properly sorted, no matter how we divide the elements after the one at index $r = 2$ into subarrays because the element at index 3 will always appear after the value 4 (see Figure 48.1).

This insight makes it possible to derive a greedy approach to this problem that is based on the idea that we are going to split I into as many pieces as possible, such that the largest element of a sub-array is smaller than all the subsequent elements.

Listing 48.2 shows an implementation of this that works by keeping a sorted list N of all the elements not yet processed in I . N initially contains all the values in I . We start a new chunk at index 0 and we keep including elements into this chunk until the largest of its elements is still larger than the smallest element in N . When an element is included in the chunk then it is removed from N . If the largest element in the current chunk is indeed smaller than the smallest in N , then this signals the fact we can sort this chunk independently without causing I as a whole not to be sorted (all the elements of this chunk appear before the rest of the elements when I is sorted). At this point, we can start a new chunk and repeat the process until we are left with no element to process (or equivalently N is empty). Listing 48.2 has a complexity of $O(|I|)$ for both space and time.

```

1  int max_chunks_to_sorted_lineartime(std::vector<int>& arr)
2  {
3      constexpr int INF = std::numeric_limits<int>::min();
4
5      if (arr.size() <= 0)
6          return 0;
7
8      std::set<int> N(std::begin(arr), std::end(arr));
9      int ans      = 0;
10     int curr_max = INF;
11     for (int i = 0; i < std::ssize(arr); i++)
12     {
13         N.erase(arr[i]);
14         const int new_max      = std::max(curr_max, arr[i]);

```

```
15     const auto& smallest_among_rest = *(N.begin());
16     if (N.size() > 0 && new_max >= smallest_among_rest)
17     {
18         curr_max = new_max;
19     }
20     else
21     {
22         ans++;
23         curr_max = INF;
24     }
25 }
26 return ans;
27 }
```

Listing 48.2: Linear time solution to the problem *Sort the chunks, sort the array*.

49. Palindrome Partitioning II

Introduction

In this chapter we will investigate a problem involving strings. It features a short yet complex statement which requires some care to internalize and understand fully. This is another problem on palindromes where we are asked to calculate the cost of breaking down an input string into chunks such that each of the individual chunks is a palindrome.

49.1 Problem statement

Problem 70 Write a function that - given a string s - partitions it in such a way that every resulting substring is a palindrome. A partition for a string s is a collection of cut-points $1 \leq c_0 < c_1 \dots < c_k < |s|$ splitting the string s into $k+1$ non empty substrings:

- $s(0 \dots c_0)$
- $s(c_0 + 1 \dots c_1)$
- ...
- $s(k - 1 \dots c_k)$

The function should return the minimum number of cuts needed so that the resulting partition consists only of palindrome substrings.

■ Example 49.1

Given $s = \text{"aab"}$ the function returns 1. 0 cut-points are not enough as s itself is not a palindrome but with one cutpoint at index 1 we can obtain the following partitioning $[\text{"aa"}, \text{"b"}]$ where both aa and b are palindromes. ■

■ Example 49.2

Given $s = \text{"itopinonavevanonipoti"}$ the function returns 0 because s is itself a palindrome. ■

■ Example 49.3

Given $s = \text{"ababbbabbababa"}$ the function returns 3. One possible partition that could be produced with 3 cuts is: $[\text{"a"}, \text{"babbbab"}, \text{"b"}, \text{"ababa"}]$. ■

49.2 Discussion

49.2.1 Brute-force

The obvious solution would be to try to all possible partitions of the input string, from the ones splitting it into 1 piece, then all the ones splitting it into 2 pieces, and so on in a similar fashion until we eventually find a partition that splits s into palindromes. Such a partition must exist as if we split s into $|s|$ pieces, down to its individual characters, the resulting substrings of length one are all palindromes. This approach is basically the same adopted for the brute-force (see Section 42.3.1) solution of the problem discussed in Chapter 42 where the bulk of the complexity is in the generation of the partitions of incremental size. In order to do that, we could use the algorithm for the generation of

all combinations of size k shown in Listing 42.1 to generate all possible cut-points and from there get the associated sub-strings. For each partition size $l = 1, 2, \dots, |s|$ we can use Listing 42.1 to generate the combination of $\{1, 2, \dots, |s| - 1\}$ in groups of size l and for each of them evaluate whether the resulting substrings are all palindromes. We can return l as soon as we find a combination which does.

Listing 42.1 shows an implementation of this idea which has a time and space complexity of $O(2^{|s|})$. The work done is the sum of all the work necessary to generate the combinations of sizes $1, 2, \dots, |s| - 1$ i.e. $\sum_{k=1}^{|s|-1} \binom{|s|}{k} = 2^n$. The union of all combinations of size $k = 1, 2, \dots, |s|$ is equivalent to the power-set (see Section 1 at page 2) which has size 2^n .

```

1  #include "generate_combinations.h"
2
3  /**
4   * Returns
5   *   true iff the substring of s starting at start and ending at end is
6   *   palindrome false otherwise.
7   */
8  bool is_palindrome(const std::string& s, const size_t start, const size_t end)
9  {
10     assert(start <= end);
11     assert(end < s.length());
12
13     auto l = start, r = end;
14     while (l < r)
15         if (s[l++] != s[r--])
16             return false;
17     return true;
18 }
19
20 int palindrome_partitioning2_bruteforce(const std::string s)
21 {
22     if (is_palindrome(s, 0, s.size() - 1))
23         return 0;
24
25     for (int k = 1; k <= std::ssize(s) - 1; k++)
26     {
27         // generate combinations in groups of k from [0...s.size()-2]
28         const auto& cutpoints = all_combinations(k, s.size());
29         // is there a partition of size k such that all the associated substrings
30         // in
31         // s are palindrome?
32         const auto found = std::any_of(
33             std::begin(cutpoints), std::end(cutpoints), [&](const auto& combo) {
34                 auto substring_start = 0;
35                 for (const auto substring_end : combo)
36                 {
37                     if (!is_palindrome(s, substring_start, substring_end))
38                         return false;
39                     substring_start = substring_end + 1;
40                 }
41                 return is_palindrome(s, substring_start, s.size() - 1);
42             });
43         if (found)
44             return k;
45     }
46     return s.size() - 1;

```

Listing 49.1: Exponential time solution to the palindrome partition problem using Listing 42.1 at page as a sub-routine for the generation of the combinations of size k .

49.3 Dynamic Programming

This problem has however a solution that is much better than exponential time. As the title of this section suggests, we can use DP to effectively tackle it. The key insight that allows us to develop an effective DP solution is to think about how we can break down the problem into subproblems on a suffix of s . What I mean is that we can think about a function $P(s, i)$ which returns the answer to the problem for the substring of s from the character at index i to its end. Clearly $P(s, 0)$ is the answer to the general question. However given this formulation we can express the solution of a subproblem for a starting index i in terms of optimal solutions for the smaller sub-problems at indices $j > i$. Equation 49.1 captures this idea in a recurrence relation. It states that the answer to a subproblem for an empty substring of s is zero: no cuts are necessary as an empty string is already palindrome. If the whole string from index i to the end is a palindrome, then no cuts are necessary. For any other sub-problems $P(s, i)$ what we can do is to make a cut at an index $k \geq i$ provided that the substring of s from index i to k resulting from the cut is a palindrome and then add to it the answer to the subproblem $P(s, k + 1)$ which gives the optimal solution for the unprocessed part of s : from index $k + 1$ to the end.

$$P(s, i) = \begin{cases} 0 & \text{if } i \geq |s| \\ 0 & \text{if } s[i \dots |s| - 1] \text{ is palindrome} \\ \min_{k \geq |s|} (1 + P(s, k + 1)) & \text{if } s[i \dots k] \text{ is palindrome} \end{cases} \quad (49.1)$$

49.3.1 Top-down

The solution outlined in Section 49.3 and formalized in Equation 49.1 can be easily translated into a recursive solution as shown in Listing 49.2. The recursive function `palindrome_partitioning2_DP_helper` has an almost 1-to-1 mapping to the Equation 49.1 except for the code responsible for the memoization optimization which allows the answer for a given subproblem that has been previously solved to be returned immediately. We have used this optimization already in other problems such as: 1. *Number of Dice Rolls With Target Sum* in Section 45 at page 259 or 2. *Minimum difficulty job schedule* in Section 42 at page 232.

```

1 using Cache = std::unordered_map<int, int>;
2
3 int palindrome_partitioning2_DP_topdown_helper(const std::string s,
4                                                const int start_idx,
5                                                Cache& memoization_cache)
6 {
7     if (start_idx >= std::ssize(s)
8         || is_palindrome(s, start_idx, std::ssize(s) - 1))
9         return 0;
10
11     if (memoization_cache.contains(start_idx))
12         return memoization_cache[start_idx];
13
14     int ans = std::numeric_limits<int>::max();
15     for (int i = start_idx; i < std::ssize(s); i++)

```



```

16 {
17     if (is_palindrome(s, start_idx, i))
18         ans = std::min(ans,
19                        1
20                        + palindrome_partitioning2_DP_topdown_helper(
21                            s, i + 1, memoization_cache));
22 }
23
24 assert(ans <= std::ssize(s) - start_idx);
25 memoization_cache[start_idx] = ans;
26 return ans;
27 }
28
29 size_t palindrome_partitioning2_DP_topdown(const std::string s)
30 {
31     Cache memoization_cache;
32     return palindrome_partitioning2_DP_topdown_helper(s, 0, memoization_cache);
33 }

```

Listing 49.2: Quadratic time dynamic programming top-down solution to the palindrome partition problem.

The complexity of this solution is $O(|s|^3)$ because each of the $O(|s|)$ distinct (for which we might have to execute the whole function code) calls to `palindrome_partitioning2_DP_helper` and each performs $O(|s|^2)$ work: the for loops runs $O(|s|)$ times, and the function `is_palindrome` has a complexity of $O(|s|)$.

49.3.1.1 Top-down improved

The complexity of code in Listing 49.2 can be lowered to $O(|s|^2)$ if we are able to answer the question about whether a given substring of s from index i to $j > i$ is a palindrome or not. The current implementation blindly processes the whole substring to find that information. What we can do instead is to use DP again to build a table B where each of its elements $B[i][j]$ contains the information about whether the substring of s from index i to $j > i$ is a palindrome. The key idea here is that we can build such a table in $O(|s|^2)$ time and store it using $O(|s|^2)$ space.

A palindrome is a word having the same first and last character, for which the substring obtained by removing the first and the last character is itself a palindrome. We are going to use this property to build B which is reflected by the fact that an entry of B , $B[i][j]$ contains true if and only if $s[i] = s[j]$ and $B[i + 1][j - 1]$ is true. There are certain cells of B that we can fill immediately: for instance all the cells where $i = j$ can be set to true as those map to sub-strings of s of length 1 which are palindromes by definition. We can fill the table by using a recursive function and memoization as shown in Listing 49.3. This shows an implementation of a class which will provide the same information in the table B but wrapped in a class with a simple API: a constructor taking a `std::string` as input and a function `bool is_palindrome(const size_t start, const size_t end)` for answering queries on sub-strings of s . Note that the constructor will immediately call the function `buildMap` which will fill the table B fully before any call to `is_palindrome`. With minimal change we can also make the class `PalindromeSubstringCacheRecursive`, by removing the `buildMap` function completely via implementing `is_palindrome` in terms of `is_palindrome_substring_helper`. The file `"hash_pair.h"`^① contains some code whose only purpose is to allow us to use `std::pair<int, int>` as keys in the `std::unordered_map`. The same functionality can be also implemented iteratively by using a bottom-up DP strategy. Listing 49.4 illustrates how this can be done.

^①See Listing 64.3 at page 387.

```

1  #include "hash_pair.h"
2
3  class PalindromeSubstringCacheRecursive
4  {
5  public:
6      PalindromeSubstringCacheRecursive(const std::string& s) :mStr_size(s.size())
7          {
8              buildMap(s);
9          }
10
11     [[nodiscard]] bool is_palindrome(const size_t start, const size_t end)
12         const
13     {
14         const std::pair<size_t,size_t> p(start, end);
15         return mB.contains(p) && mB.at(p);
16     }
17
18     [[nodiscard]] size_t size() const {
19         return mStr_size;
20     }
21 private:
22     bool is_palindrome_substring_helper(const string&s, const size_t start,
23         const size_t end)
24     {
25         if(start > end || start >= s.size() || end < 0)
26             return true;
27
28         const std::pair<int,int> p(start, end);
29         if(mB.contains(p))
30             return mB.at(p);
31
32         const bool ans = (start == end) || (
33             (s[start]==s[end]) &&
34             is_palindrome_substring_helper(s,start+1, end-1)
35         );
36         mB.insert({p, ans});
37         return ans;
38     }
39
40     void buildMap(const std::string&s)
41     {
42         for(size_t i = 0 ; i < std::size(s) ; i++)
43         {
44             for (size_t j = i; j < std::size(s); j++)
45             {
46                 mB[{i,j}] = is_palindrome_substring_helper(s,i,j);
47             }
48         }
49     }
50
51     std::unordered_map<std::pair<int,int>, bool, PairHasher> mB;
52     const size_t mStr_size;
53 };

```

Listing 49.3: Recursive implementation of a class which allows to answer queries about whether a given substring of a given string is palindrome or not in constant time.

```

1  class PalindromeSubstringCacheIterative
2  {

```

```

3 public:
4     PalindromeSubstringCacheIterative(const std::string& s) : mStr_size(s.size())
5     {
6         buildMap(s);
7     }
8     [[nodiscard]] bool is_palindrome(const size_t start, const size_t end)
9     const
10    {
11        return start < mStr_size && end >= 0 &&
12            mB[start][end] != -1 && mB[start][end];
13    }
14
15    [[nodiscard]] size_t size() const {
16        return mStr_size;
17    }
18
19 private:
20
21     void buildMap(const std::string&s)
22     {
23         mB.resize(mStr_size, std::vector<int>(mStr_size,-1));
24         for(int i = mStr_size-1; i >= 0 ; i--)
25         {
26             for(int j = i ; j < mStr_size ; j++ )
27             {
28                 mB[i][j]=(s[i]==s[j]) && ((j-i<=2) || mB[i+1][j-1]);
29                 //mB[{i,j}]=s[i]==s[j] && ((j-i<=2) || mB[{i+1,j-1}]);
30             }
31         }
32     }
33
34     std::vector<vector<int>>> mB;
35     const size_t mStr_size;
36
37 };

```

Listing 49.4: Iterative implementation of a class which allows to answer queries about whether a given substring of a given string is palindrome or not in constant time.

Finally in Listing 49.5 we can see how such a *Substring Palindrome Cache* can be used in order to implement a quadratic solution for the problem. Notice that the code for `palindrome_partitioning2_DP_topdown_optimized_helper` is almost identical to the one of `palindrome_partitioning2_DP_topdown_helper` in Listing 49.2 with the difference being that the former takes B , the substring palindrome cache, as an additional parameter and that the call to `is_palindrome` is substituted with a query into B which runs in constant time. The complexity of this solution is now $O(|s|^2)$ which is a big improvement from $O(|s|^3)$. The space complexity is also $O(|s|^2)$, because of the space used by B .

```

1 #include "PalindromeSubstringCacheRecursive.h"
2
3 size_t palindrome_partitioning2_DP_topdown_optimized_helper(
4     const PalindromeSubstringCacheRecursive& B,
5     const size_t start_idx,
6     Cache& memoization_cache)
7 {
8     const auto size = B.size();
9     if (start_idx >= size || B.is_palindrome(start_idx, size - 1))
10         return 0;

```

```

11
12     if (memoization_cache.contains(start_idx))
13         return memoization_cache[start_idx];
14
15     // cout<<start_idx<<std::endl;
16     size_t ans = std::numeric_limits<int>::max();
17     for (size_t i = start_idx; i < size; i++)
18     {
19         if (B.is_palindrome(start_idx, i)) // O(1)
20             ans = std::min(ans,
21                             1
22                             + palindrome_partitioning2_DP_topdown_optimized_helper
23                               (
24                                   B, i + 1, memoization_cache));
25     }
26     assert(ans <= size - start_idx);
27     memoization_cache[start_idx] = ans;
28     return ans;
29 }
30
31 size_t palindrome_partitioning2_DP_topdown_optimized(const std::string s)
32 {
33     PalindromeSubstringCacheRecursive B(s);
34     Cache memoization_cache;
35     return palindrome_partitioning2_DP_topdown_optimized_helper(
36         B, 0u, memoization_cache);
37 }

```

Listing 49.5: Quadratic time dynamic programming bottom-up solution to the palindrome partition problem.

49.3.2 Bottom-up

In this section we will discuss how we can implement the DP approach shown in Section 49.3 in a bottom-up fashion.

The idea is that we can start processing progressively longer portions of s starting from the last character at index $|s| - 1$. Each of these portions, starting at index i to the end of s , correspond to a sub-problem that can be uniquely identified by its starting index i . For instance subproblem where $i = 3$ corresponds to a substring of s from index 3 to its end. When solving a given sub-problem i , we will use the information about sub-problems related to smaller portions of s starting at higher indices $j > i$ to determine the minimum number of cuts necessary to split the substring $s[i \dots |s| - 1]$ into several palindromes.

The substring of s starting at index $|s| - 1$ has size 1 and therefore is already a palindrome and does not need any cuts. For any other substring starting at index $i = |s| - 1 - k$ where $k \geq 0$, we have two options:

1. if $s[i \dots |s| - 1]$ is already a palindrome, then we know this subproblem has a solution equal to 0. No cuts are necessary.
2. otherwise, we can try to split $s[i \dots |s| - 1]$ at index $i + 1 \leq j \leq |s| - 1$. If $s[i \dots j]$ is a palindrome, then we know we can turn $s[i \dots |s| - 1]$ into a partition of palindromes by using one cut (the one we just performed at index j) plus all the cuts necessary to turn the substring $s[j \dots |s| - 1]$ into a partition of palindromes. The crucial point is that we have already solved the sub-problem $j > i$ and therefore we can reuse its solution. The final answer for this sub-problem starting at index i is the smallest value we can obtain among all the cuts (at index $j > i$) we can make.

The sub-problem at index 0 contains the answer for the entire problem i.e. the smallest size of a palindrome partition of s starting at index 0. Listing 49.6 implements this idea where we use an array `DP` having size $|s| + 1$ to store the answers to all sub-problems.

```
1
2
3 int palindrome_partitioning2_DP_bottomup(const std::string s)
4 {
5     std::vector<int> DP(s.size() + 1, s.size() - 1);
6     DP[s.size() - 1] = DP[s.size()] = 0;
7     for (int i = std::ssize(s) - 2; i >= 0; i--)
8     {
9         if (is_palindrome(s, i, s.size() - 1))
10         {
11             DP[i] = 0;
12             continue;
13         }
14         for (int j = i; j < std::ssize(s); j++)
15         {
16             if (is_palindrome(s, i, j))
17             {
18                 const auto cost_partition_rest = j < std::ssize(s) - 1 ? DP[j + 1] : 0;
19                 DP[i] = std::min(DP[i], 1 + cost_partition_rest);
20             }
21         }
22     }
23     return DP[0];
24 }
```

Listing 49.6: Quadratic time dynamic programming bottom-up solution to the palindrome partition problem.

50. Find the largest gap

Introduction

This chapter discusses another sorting problem. The statement is quite simple and the only input given is an unsorted array from which we are asked to calculate a value that would be simple to find if the input was sorted. Therefore, the real challenge of this problem is to find a solution that does not require explicit sorting.

Particular attention should be paid to the examples as well as to the problem statement because it is easy to misinterpret the real requirements of the function you are asked to write if you dive straight into coding. The problem asks you to return the largest distance between any element in the input array provided they appear one next to the other when I is sorted. You might misinterpret the problem by thinking that you need to return the largest distance between any two elements of the original input array but this is incorrect. This should be obvious if we consider that the we are only being asked to find the minimum and the maximum values of the input array. You can expect any coding interview question to be harder than that. An imaginative effort (or some pen and paper work) is therefore necessary to understand each of the examples provided.

50.1 Problem statement

Problem 71 Write a function that given a unsorted array of non-negative integers I of length n returns the largest gap between two elements that would appear one next to the other if I was sorted. A gap between x and y is defined as the absolute value of the difference between x and y : $|x - y|$.

■ Example 50.1

Given $I = \{5, 3, 1, 8, 9, 2, 4\}$ the function returns 3. Sorting I changes it into: $\text{sort}(I) = \{1, 2, 3, 4, 5, 8, 9\}$, and the largest gap between any two consecutive elements is 3. In this case between 5 and 8. ■

■ Example 50.2

Given $I = \{7, 1, 8, 9, 15\}$ the function returns 6. $\text{sort}(I) = \{1, 7, 8, 9, 15\}$, and the largest gap between any two of its consecutive elements is 6 e.g. between 1 and 7 or between 15 and 9. ■

50.2 Clarification Questions

Q.1. Is the input I modifiable?

Yes you can modify I .

Q.2. Is there any guarantee or constraint on the value of each element of I ?

You can assume each element of I fits in a 4 bytes unsigned integer.

50.3 Trivial Solution

As already discussed, this problem has an extremely simple solution when we can afford to get our hands on a sorted version of the input array. In this specific version of the problem I is not read-only and we are allowed to modify it, therefore, we can sort it directly. If that is not possible all you have to do is to create a copy of I and sort that instead.

Given a sorted collection, the largest gap between any two numbers can be found in linear time by just scanning each pair $p = (I_k, I_{k+1})$ of subsequent elements and for each of them calculating their distance $d_k = I_{k+1} - I_k$. Among all calculated distances we simply return the largest. Listing 50.1 shows an implementation of this idea. Note that:

- we do not need to use the absolute value operation as we are operating on a sorted collection and therefore we are guaranteed that I_{k+1} is larger than or equal to I_k .
- the `for` loop stops when $i = |I| - 1$ in order to avoid accessing an invalid element while executing `I_copy[i+1]`. When $i = |I| - 1$ this would lead to accessing the element at index $|I|$, which does not exist. In C++ this would cause undefined behaviour and the most likely outcome would be a segmentation fault error.

```
1 int max_gap_bruteforce(const std::vector<int>& I)
2 {
3     auto I_copy(I);
4     std::ranges::sort(I_copy);
5
6     int ans = std::numeric_limits<int>::min();
7     for (int i = 0; i < std::ssize(I) - 1; i++)
8     {
9         ans = std::max(ans, I_copy[i + 1] - I_copy[i]);
10    }
11    return ans;
12 }
```

Listing 50.1: Trivial solution to the max gap problem using sorting and linear space.

50.4 Radix Sort

The idea we developed in Section 50.3 can be improved if instead of using a normal comparison-based sorting algorithms we use radix-sort^[cit::wiki::radix_sort]. Radix sort will perform better than a standard $O(n \log n)$ algorithm when there is an upper bound for the values of the input array. If we assume that such bound is the largest value a standard 4 bytes `int` can hold then radix sort will have a complexity of $O(n)$.

Radix sort works by sorting the input array d times, where $d = \lfloor \log_{10} k \rfloor + 1$ and k is the largest number in I . d is just the number of digits of the largest number in the list. For a standard `int` $d = \lfloor \log_{10} (2^{32}) \rfloor + 1 = 10$. The sorting is obtained by repeatedly sorting the input list from the least to the most significant digit where each of the intermediate sorting steps is performed using counting-sort. For instance given $I = \{329, 457, 657, 839, 436, 720, 355\}$ the first pass of radix sort will sort I based on the value of the least significant digits. After this first pass we have $I = \{720, 355, 436, 457, 657, 329, 839\}$. Note how the first digits are sorted. At this point the algorithm proceeds by sorting I further but this time according to their second digit. The resulting list becomes $I = \{720, 329, 436, 839, 355, 457, 657\}$. Finally the third pass will sort all the elements according to the most significant digit resulting in a well sorted list: $I = \{329, 355, 436, 457, 657, 720, 839\}$. This approach needs to be tweaked if you want to apply radix sort to negative numbers^①.

^①You can treat the sign as a special kind of digit. You sort the pile on the units, then the tens, etc. and finally on the sign. This does produce a reversed order for the negatives but you then simply reverse the content of that bucket.

Listing 50.2 shows an implementation of radix-sort and its application to this chapter's problem.

```
1 void count_sort(vector<int>& I, const unsigned base, const unsigned digit_idx)
2 {
3     std::vector<std::vector<int>> counters(base);
4     for (const auto& el : I)
5     {
6         // get the digit_idx th digit
7         const auto digit_value = el / (std::pow(base, digit_idx)) % base;
8         // add this number to the corresponding bucket
9         counters[digit_value].push_back(el);
10    }
11
12    int pos = 0;
13    for (const auto& list : counters)
14    {
15        for (const auto& num : list)
16        {
17            I[pos++] = num;
18        }
19    }
20 }
21
22 void radix_sort(vector<int>& I, const unsigned base = 10)
23 {
24     for (unsigned digit = 0; digit < base; digit++)
25         count_sort(I, base, digit);
26 }
27
28 int max_gap_radix_sort(const std::vector<int>& I)
29 {
30     auto I_copy(I);
31     radix_sort(I_copy);
32
33     int ans = std::numeric_limits<int>::min();
34     for (int i = 0; i < std::ssize(I) - 1; i++)
35     {
36         ans = std::max(ans, I_copy[i + 1] - I_copy[i]);
37     }
38     return ans;
39 }
```

Listing 50.2: Linear time solution to the max gap problem using radix-sort.

Notice how the main driver function `max_gap_radix_sort` is basically the same as `max_gap_bruteforce` from Listing 50.1 except for the sorting procedure used.

50.5 Buckets and the pigeonhole principle

All the solutions we have presented so far rely on sorting. In this section we will discuss an approach that relies on the pigeonhole principle^② and bucketing. The general idea is that we could split the entire array into several buckets and then find the largest gap by only comparing one element of a given bucket to one element of the subsequent bucket.

You can think of I as an array of buckets each containing a single element. $|I|$ is made of b buckets and the total amount of elements in I is $n = b$. Imagine for a moment that you would reduce the number of buckets from b to some integer $k < b$. For the pigeonhole

^②https://en.wikipedia.org/wiki/Pigeonhole_principle

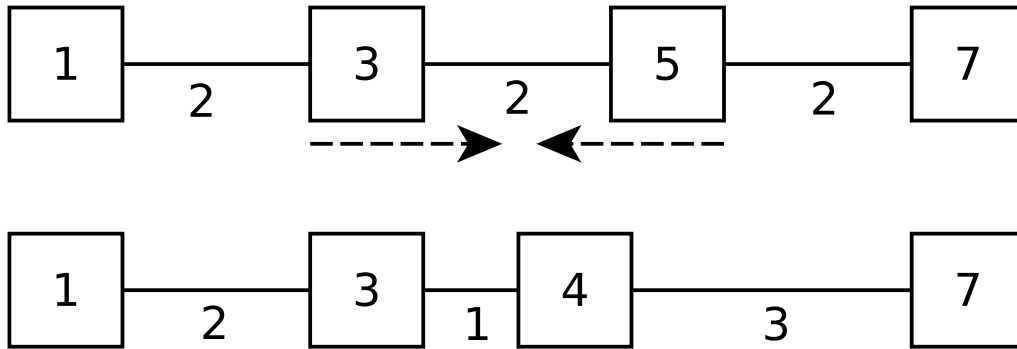


Figure 50.1: Example of how trying to reduce the gap between pairs of subsequent element of a uniformly separated collections makes the largest gap larger

principle then, one or more of the buckets in I has to contain strictly more than one element.

Let's now focus for a moment on the gaps of an ideal collection where each of its elements has the same distance t from its successor in the list. If such a collection is composed of n elements, then there is a total of $n - 1$ gaps, each of size t . t can be easily calculated if the maximum and minimum values are known - in fact $t = \frac{\max - \min}{n - 1}$. For instance for the collection of five elements $\{4, 8, 12, 16, 20\}$ we have $t = \frac{20 - 4}{4}$ and for the $\{-2, 5, 12, 19\}$ we have $t = \frac{19 - (-2)}{3} = 7$. If I was like this ideal collection then the problem would be easily solvable by using the formula above.

I is different to this ideal collection because its elements do not have uniform gaps between them. In this situation we can argue that the maximum gap between any pair of subsequent elements of I is always larger than $t = \frac{\max - \min}{n - 1}$. We can show this by taking an ideal collection C and trying to reduce the gap between any two subsequent elements C_i and C_{i+1} . We do that by moving I_{i+1} closer to I_i . When this happens the gap $(I_{i+1} - I_i)$ becomes smaller than t . So far it all looks promising but what happens to the gap between I_{i+1} and I_{i+2} ? It actually becomes larger than t . In our effort to make the largest gap among two subsequent elements smaller, we obtained the opposite result, we made it larger! Figure 50.1 shows an example of such a scenario where we have a collection of uniformly separated elements where $t = 2$. When the third element is moved by one toward the second, you see that the gap between them is reduced from 2 to 1 but the gap between the third and the fourth element increases from 2 to 3, and now the largest gap between any two pairs of subsequent element is no more 2 but 3 which is larger than what we had to begin with. This shows that the maximum attainable gap can in a collection with uniform gaps can only increase.

We are going to apply the two ideas above to solve this problem in the following way. We will distribute all the elements of I into $n - 1$ buckets. The first bucket will contain all the elements of I in the following range: $[\min, \min + t)$. Similarly the second bucket will contain all the elements in the following range: $[\min + t, \min + 2t)$. In general the i^{th} bucket would contain all the elements of I in the following range: $[\min + (i - 1)t, \min + it)$ (where $1 \leq i$). You can refer to Figure 50.2 for an example of how such a division into buckets would work for the input array in Example 50.2. This allows us to skip comparing all the

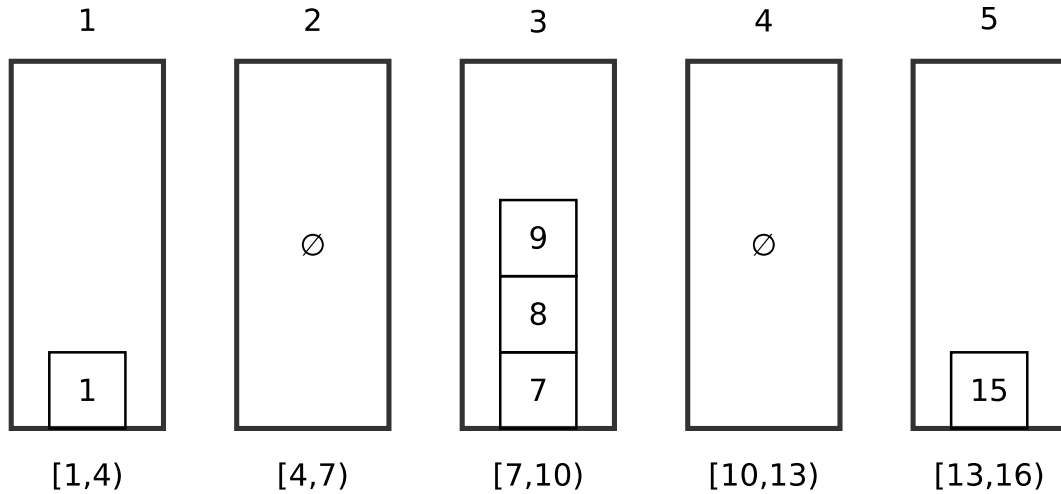


Figure 50.2: Division of the elements of the Example 50.2 into buckets.

element within a bucket because we know for sure they will have a distance that is lower than or equal to t and we can therefore concentrate on comparing elements of subsequent buckets. In particular we should compare the maximum value of the i^{th} bucket with the minimum value of the $(i+1)^{th}$ bucket. This is because they would appear one next to the other if I was sorted. As the number of buckets is always lower or equal to the number of elements in the collection, this approach has a linear time complexity as it requires comparing the number of input elements in I twice at most. The space complexity is also linear as the number of buckets can be proportional to $|I|$.

Listing 50.3 shows an implementation of this idea where we use the `struct Bucket` to model a bucket for which we only need to store three pieces of information: 1. if the buckets contains at least one elements, 2. its minimum 3. and the maximum value. If $|I| < 2$ we can immediately return 0 as there are no possible pairs to calculate the gap for. Otherwise we proceed by calculating t and the number of buckets we need. The first loop takes care of filling each of the buckets an element belongs to. Note that we can calculate such an index for an element el by using the following expression: $\frac{el - \min(I)}{t}$. Once all the buckets are initialized, we can proceed further by calculating the largest gap between them, ensuring we don't consider empty buckets which are ignored during the second loop. We proceed by considering the max element of the first bucket with the minimum element of the next non-empty bucket $j > 0$. Once the gap between them is calculated, we can move on to calculating the gap between the next pair of subsequent buckets which will be made of the bucket at index j and the first non-empty bucket having index larger than j . This process is repeated until all pairs of buckets are processed.

```

1 struct Bucket
2 {
3     bool used = false;
4     int minval = std::numeric_limits<int>::max();
5     int maxval = std::numeric_limits<int>::min();
6 };
7
8 int max_gap_buckets(const std::vector<int>& I)
9 {

```

```

10  if (I.size() < 2)
11      return 0;
12
13  const auto [minEl, maxEl] = [&I]() {
14      const auto p = std::minmax_element(I.begin(), I.end());
15      return std::make_tuple(*p.first, *p.second);
16  }();
17
18  const int t = std::max(
19      11, (maxEl - minEl) / (std::ssize(I) - 1)); // bucket size or capacity
20  const size_t num_buckets = ((maxEl - minEl) / t) + 1; // number of buckets
21  std::vector<Bucket> buckets(num_buckets);
22
23  for (const auto& el : I)
24  {
25      const size_t bucketIdx = (el - minEl) / t; // bucket idx for this element
26      buckets[bucketIdx].used = true;
27      buckets[bucketIdx].minval = std::min(el, buckets[bucketIdx].minval);
28      buckets[bucketIdx].maxval = std::max(el, buckets[bucketIdx].maxval);
29  }
30
31  int prevBucketMax = minEl, ans = 0;
32  for (auto&& bucket : buckets)
33  {
34      if (!bucket.used) // skip empty buckets
35          continue;
36
37      ans = std::max(ans, bucket.minval - prevBucketMax);
38      prevBucketMax = bucket.maxval;
39  }
40
41  return ans;
42 }

```

Listing 50.3: Linear time solution to the max gap problem using bucketing.

51. Smallest Range I and II

Introduction

This chapter discusses a problem and one of its variations on arrays. Both are very common interview questions, with the former having a simple solution, and the latter being slightly more complex. Both variants however, can be solved by starting from the same idea and using just a handful of lines of code.

51.1 Problem statement

Problem 72 Write a function that - given an array of integers I and an integer $K \geq 0$ - returns the smallest possible difference between the smallest and largest value in I after you have added $-K \leq p \leq K$ to each of the elements.

■ **Example 51.1**

Given $I = \{3, 5, 1, 7, 8\}$ and $K = 4$ the function returns 0. You can add I to the following values: $\{1, -1, 3, -3, -4\}$. The modified array becomes: $B = \{4, 4, 4, 4, 4\}$. ■

■ **Example 51.2**

Given $I = \{1, 9, 4\}$ and $K = 2$ the function returns 4. ■

51.2 Clarification Questions

Q.1. Can you add p multiple times to an element of I ?

No, you can only add p once to each element of I .

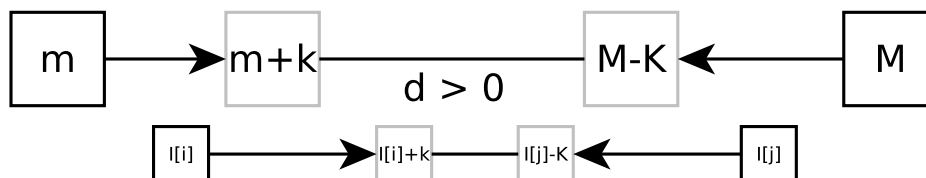
51.3 Discussion

For this problem it isn't valuable to discuss the brute-force solution^① as it is impractical to actually code it during an interview. Moreover, such a solution is conceptually very different and, time complexity-wise, very far from the one that allows us to solve problem most efficiently. Instead, we will go directly to examining a better approach to the solution.

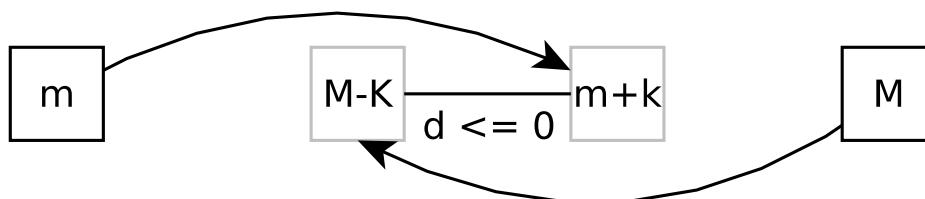
The problem is asking us to minimize the difference between the largest value (M) and the smallest value (m) of I after we have processed it by adding to each and every of its element a value in the range $[-K, +K]$. Let's call B this post-processed version of I . We know that if K is large enough^② so that we can modify M and m to be the same value by subtracting from M and adding to m then we make all the elements of I equal, thus reducing the difference between I 's smallest and the largest element to zero (see Figure 51.1b). This is possible because $M - m$ is the largest difference in I and if we can effectively

^①Constructing and returning the difference between the largest and smallest element among all of the $2K^{|I|}$ possible arrays you can obtain by adding any of the $2K$ between $-K$ and K to each and every element of I .

^②"large enough" in this context means that $(M - K) - (m + K) = M - m - 2K \leq 0$.



(a) m and M are the smallest and largest element of I , respectively. You can bring them closer together by adding K to m and subtracting K to M . d is the difference between these two new values. d will always be larger than any other difference you can obtain in the same way. You can see that any other pair $(I[i], I[j])$ will have a smaller difference because they are closer together to begin with.



(b) m and M are the smallest and largest element of I , respectively. If we add K to m and subtract K to M then in this case $(m+K)$ will be larger than $(M-K)$. This means that we can add to m and subtract to M a number $p \leq K$ such that $m+p = M-p$. Because any other number in I is larger than m and smaller than M we can do the same with them, so that we bring all the elements to the same value.

Figure 51.1

close their gap to 0 then we can do the same with any other difference between any pair of elements of I . On the other hand, if K is not large enough^③, then all we know is that we can reduce the difference between m and M to $d = (M-K) - (m+K)$. Note that in this case $d > 0$ (see Figure 51.1a). Moreover, similar to what we have discussed above, because the difference between any other pair of elements of I is smaller than or equal to $(M-m)$, we also know that their differences can be made at least equal to or smaller than d .

Therefore in order to solve this problem we only have to look at M and m and calculate $d = (M-K) - (m+K)$. If $d \leq 0$ then it means that we can make all the elements of I equal and the function should return 0, otherwise we can safely return d as an answer.

You can find an implementation of this idea in Listing 51.1 which has $O(|I|)$ time and $O(1)$ space complexity. The `std::minmax_element`^④ function returns a pair of iterators pointing to the minimum and maximum element of the input range, respectively.

```

1 int smallest_range_I(const vector<int>& I, const int K)
2 {
3     const auto& [m, M] = std::minmax_element(std::begin(I), std::end(I));
4     const int d = (*M - K) - (*m + K);
5     return std::max(0, d);
6 }

```

Listing 51.1: Solution to the *smallest range* problem.

^③When $(M-m) > 2K$.

^④https://en.cppreference.com/w/cpp/algorithm/minmax_element

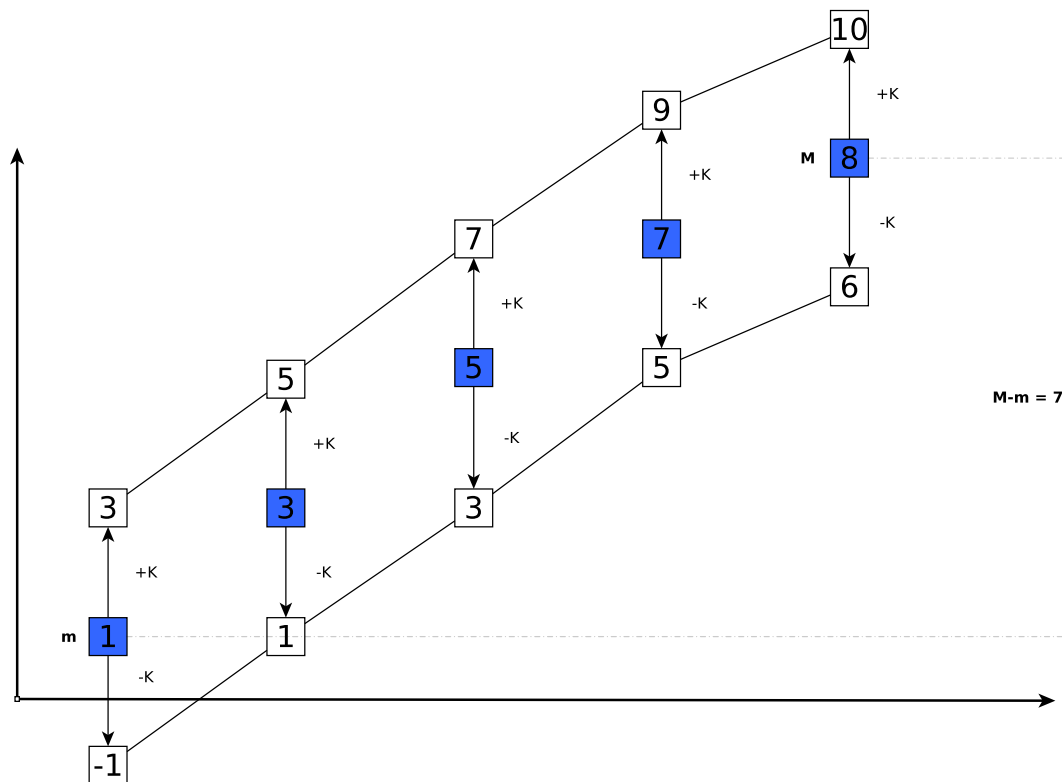


Figure 51.2: This figure is a visual representation of the Example 51.3 where the highlighted boxes represent the input values, the white boxes at the top of the picture represent the values we can get by adding K to the corresponding element and the white boxes at the bottom of the picture shows the values we can get by subtracting K to them.

51.4 Common Variations

51.4.1 Smallest range II

This variant is almost identical to the main problem in this chapter (see Exercise 72) except that this time we are only allowed to add either $-K$ or $+K$ (and not any number in the range $[-K, K)$ to each element of the input array I . As we shall see, this complicates things somewhat but, nevertheless, the core solution strategy remains the same.

Problem 73 Write a function that - given an array of integers I and an integer K - returns the smallest possible difference between the smallest and largest value in I after you have added either $-K$ or K to each of the elements .

■ Example 51.3

Given $I = \{3, 5, 1, 7, 8\}$ and $K = 2$ the function returns 3. You can modify add to I the followings values $\{2, -2, 2, -2, -2\}$. The modified array finally becomes: $I' = \{5, 3, 3, 5, 6\}$.

■

■ Example 51.4

Given $I = \{1, 9, 4\}$ and $K = 3$ the function returns 3. ■

■

■

51.5 Discussion

Let's start by noting that the solution to this problem is always smaller than or equal to the difference between the largest (M) and smaller (m) elements of I . This is the case because in the worst case scenario we can either add or subtract K to all of the elements of I and therefore preserve the relative difference between all the elements of I (including M and m). We have this case when $(M - m) \leq K$ because subtracting and adding K to M and m , respectively, would eventually lead to a larger or equal difference^⑤.

When $(M - m) > K$ then what we can do is to choose one element at index j as a pivot point and add K to all the elements smaller than or equal to I_j and subtract K from all of the elements of I greater than I_j . The new gap depends on the new smallest (m') and largest elements (M'). Given p is the smallest element larger than I_j then M' is the largest among $I_j + K$ and $M - K$ while m' is the smallest among $p - K$ and $m + K$. Therefore for a given j we calculate the maximum gap as $d_j = M' - m'$. The final answer is the smallest of these gaps calculated for each of index of I .

This approach relies on being able to quickly identify elements that are smaller or greater than a given value. If the array is sorted this can be achieved quite efficiently. In-fact if I is sorted then - for a given j - all the elements that are smaller than I_j appear at indices smaller than j and, similarly, all the elements that are larger appear at indices larger than j . Therefore, if I is sorted then m' is the smallest among $I_{j+1} - K$ and $I_0 + K$ and M' is the largest among $I_j + K$ and $I_{|I|-1} - K$.

We can use these observation to derive the following algorithm:

1. sort the input array I ,
2. for each $j = 0 \dots |I| - 1$ calculate $d_j = \max(I_j + K, I_{|I|-1} - K) - \min(I_{j+1} - K, I_0 + K)$,
3. return the smallest d_j .

An execution of this algorithm for the Example 51.3 is shown in the Figure 51.3. The initial input is shown in Figure 51.2 where the smallest and greatest value are $m = 1$ and $M = 8$, respectively. The only way for their gap to become smaller is for M to be decreased and m to be increased by K . Figure 51.3a shows how I would look if we add K to all the elements smaller than or equal to $j = 0$ and subtract K from the others. The highlighted boxes show the new array values while the white boxes show the original values. Note that the gap between the new minimum (1, obtained by subtracting 2 from the original element with value 3) and the new maximum element (6, obtained by subtracting 2 from 8) is now $6 - 3 = 3$. Similarly Figure 51.3b shows the resulting array for $j = 3$. The new array minimum and maximum values are now 3 and 6, respectively. When $j = 2$ or $j = 3$, as we can see in Figures ?? and 51.3d the gap increases where $j = 1$. Finally Figure 51.3e shows the case where all the new elements are obtained by adding K . This scenario leaves the relative distance between the elements unchanged with regard to the original values and therefore, not surprisingly, the gap between the smallest and largest element is 7 (as in Figure 51.2).

Listing 51.2 shows an implementation of this idea. Note that in the code m' is `m_new` and M' is `M_new`.

^⑤The idea behind this is that adding K to m would yield $m' = m + K$ which is greater than or equal to M . Similarly, subtracting K from M would yield $M' = M - K$ which is smaller than or equal to m . The gap between m' and M' is larger than or equal to the gap between m and M . When $(M - m) < K$ then we can express K in terms of $M - m$ as follows: $K = M - m + x$ where $x \geq 0$. Therefore by adding K to m and subtracting K from M we get: $|(M - K) - (m - K)| = |(M - (M - m + x)) - (m + (M - m + x))| = |(m - x) - (M + x)|$ which is at least as large as $M - m$.

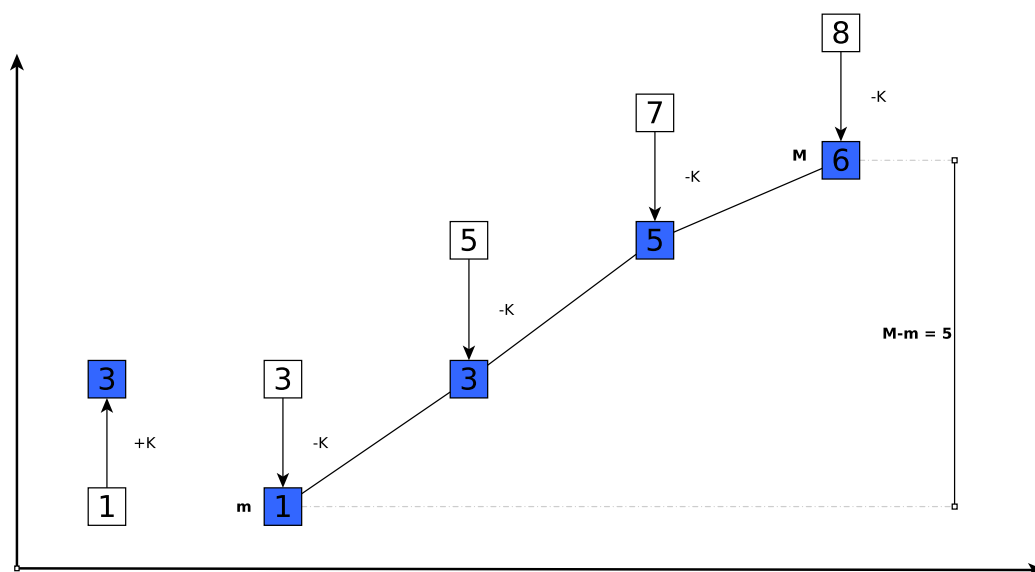
For instance given $I = \{1, 2, 6, 8\}$ and $K = 10 > (8 - 1) = 7$ if we add 10 to the smallest element of I and subtract 10 from its largest element, we end up with: $1 + 10 = 11$ and $8 - 10 = -2$. The difference between these two new values is $11 - (-2) = 13$ which is definitely larger than the difference between 1 and 8.

```

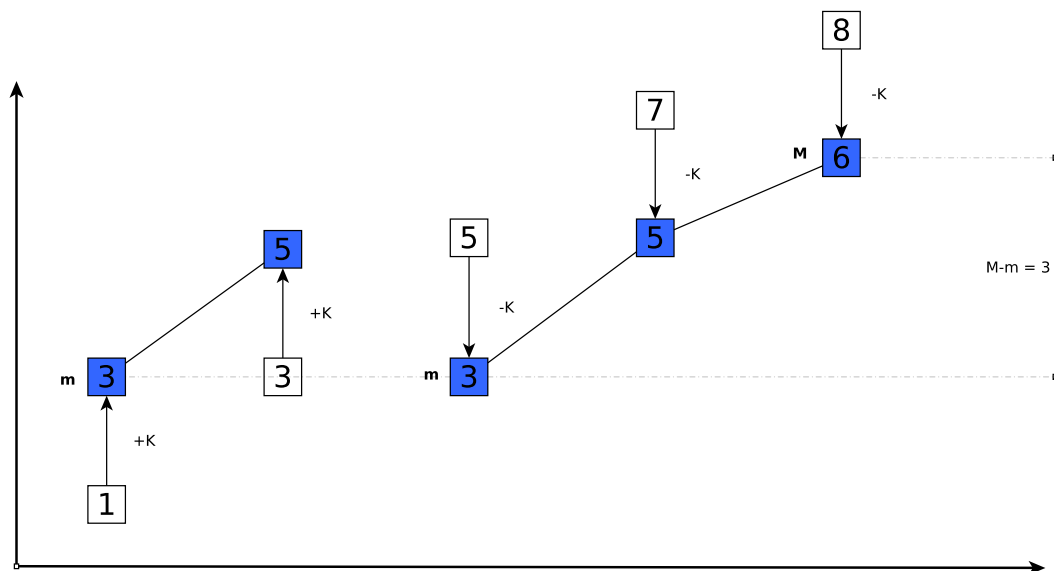
1
2 int smallest_range_II(std::vector<int>& A, const int K)
3 {
4     std::sort(std::begin(A), std::end(A));
5     int ans = A.back() - A.front();
6     for (size_t i = 1; i < A.size(); i++)
7     {
8         const auto m_new = std::min(A.front() + K, A[i] - K);
9         const auto M_new = std::max(A[i - 1] + K, A.back() - K);
10        ans          = std::min(ans, M_new - m_new);
11    }
12    return ans;
13 }

```

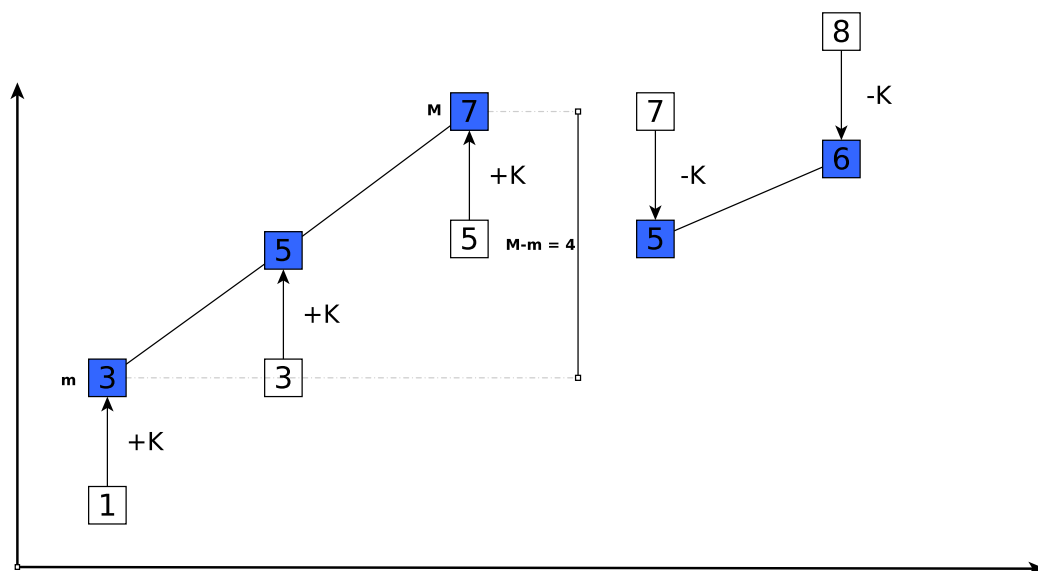
Listing 51.2: Solution to the *smallest range* problem using sorting.



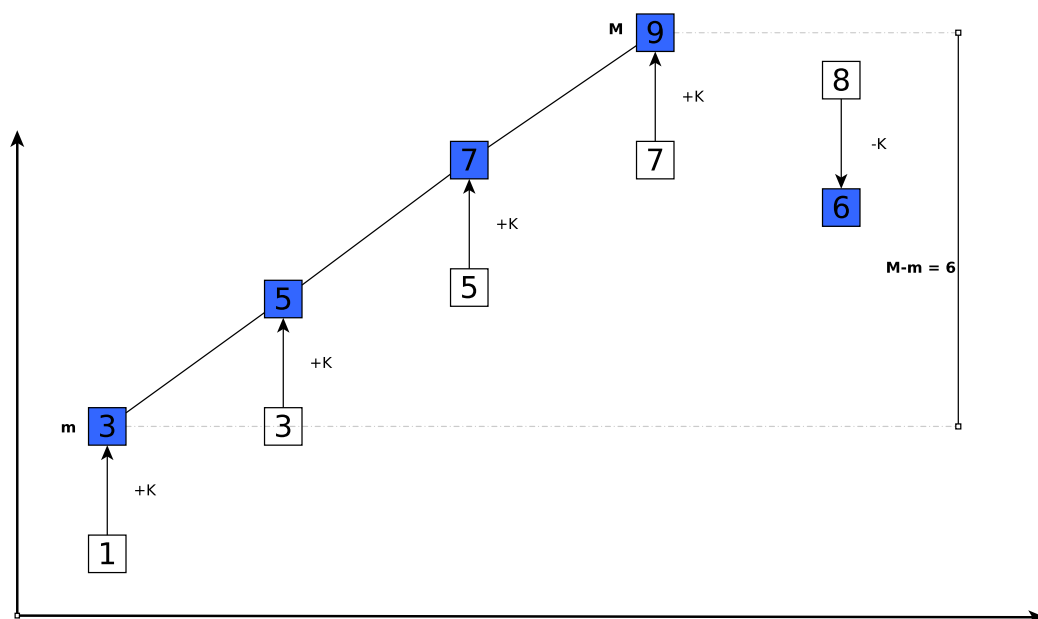
(a) $j = 0$



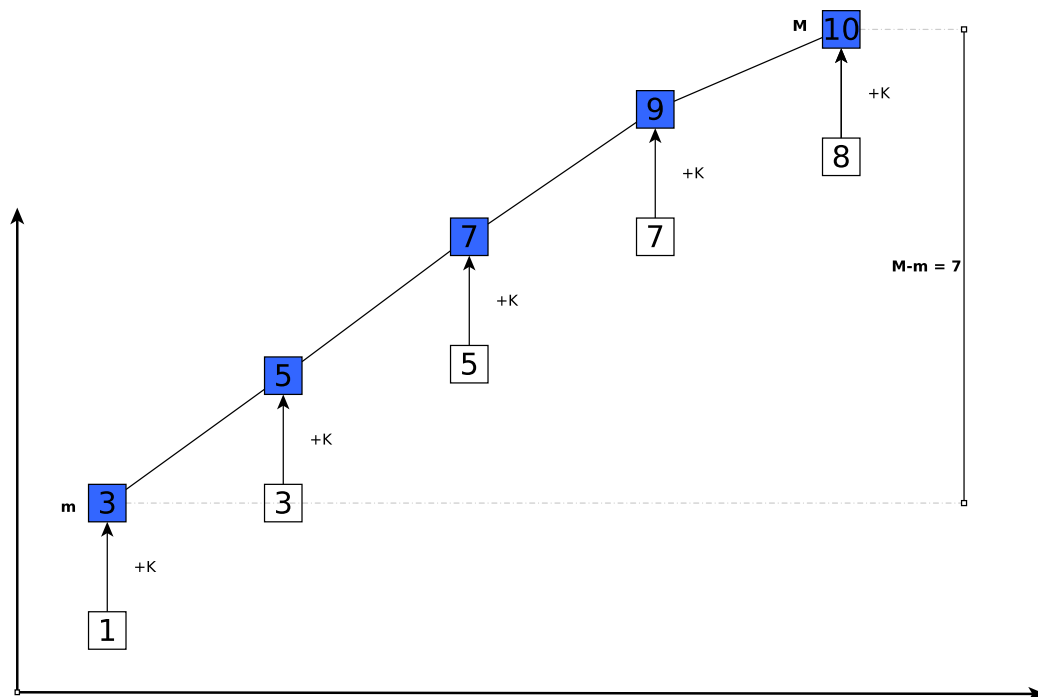
(b) $j = 1$



(c) $j = 1$



(d) $j = 1$



(e) $j = 1$

Figure 51.3: Execution of the algorithm presented in Section 51.5 for $j = 0$ (Figure 51.3a), $j = 1$ (Figure 51.3b), $j = 2$ (Figure ??), $j = 5$ (Figure ??) and $j = 5$ (Figure 51.3d). The highlighted boxes contains the values obtained from the original elements shown in the white boxes.

52. Next Greater Element I

Introduction

One of the key steps in consistent hashing^① is when we have to retrieve the IP of the machine some data o resides in and in a nutshell works by first calculating the hash of the data itself $h(o)$ and then find the smallest hashkey $h(s)$ for a server that is larger than $h(o)$. This operations might be performed thousands of times per seconds in a large system and it is therefore quite important to make it as fast as possible.

In this chapter we will analyze a similar problem where we are given a bag of integers and we need to find the *next greater* element for each of them. The number of applications and variations is high and we feel this problem is a must and that the techniques shown in this chapter can be applicable to a other real-life coding interview problem bein asked out there in the wild.

52.1 Problem statement

Problem 74 Write a function that - given two arrays with no duplicates A and B where $A \subset B$ - returns an array C of size $|A|$ where C_i contains the next greater element of A_i among the elements of B . The next greater element of a number A_i is defined as the smallest element greater than A_i among the elements of B from index j to $|B| - 1$ where $B_j = A_i$

In other words, for each A_i the function finds the smallest element in B that is greater than A_i among the cells that are to the right of the cell in B having the value A_i and places it into C at index i .

■ Example 52.1

Given $A = \{4, 1, 2\}$ and $B = \{1, 3, 4, 2\}$ the function returns $C = \{-1, 2, -1\}$. $C_0 = -1$ because there $A_0 = 4$ appears in B at index 2 and there is no cell to the right of B_2 that is strictly greater than 4. $C_1 = 2$ and because 1 appears in B at index 0 and the smallest element larger than 1 after index 0 in B is the element 2 in the last position. $C_2 = -1$ because $A_2 = 2$ appears in B at index 3 and there is no element to the right of it. Note that there exists a value in B that is larger than 2 but we are not considering it because it appears to the left of the cell in B holding value $A_2 = 2$. ■

■ Example 52.2

Given $A = \{2, 4\}$ and $B = \{9, 2, 1, 4, 12, 8\}$ the function returns $C = \{4, 8\}$. $C_0 = 4$ because

^①A special type of hashing that avoids having to remap every entry in the hash-table when the bucket size changes (to the contrary of what happens when map keys and buckets via a modular operation). The classic example of its usage is in load-balancing or caching in a distributed environment where a distributed hash-map is to be maintained across a number of machines. One way of distributing objects evenly across the n machines is to place data o into the machine $h(o) \pmod{n}$. However, if a server is added/removed, the server assignment of a lot of the data in all machines may change. This is problematic since servers often go up or down and each and that would cause a large amount of cache misses.

there $A_0 = 2$ appears in B at index 1 and the smallest element larger than 2 in B from the cell to the right of the one at index 1 is 4.

$C_1 = 8$ because there $A_0 = 4$ appears in B at index 3 and the smallest element larger than 2 in B from the cell to the right of the one at index 3 is 8, appearing at the very end of B . Note that 12 is also larger than 4 and appears to the right of the index 3 but is not the correct answer because it is not the smallest. ■

52.2 Clarification Questions

Q.1. How should the function behave when an element of A does not have a next greater in B ?

The function can insert -1 in the corresponding cell of C .

52.2.1 Brute-force

This problem has a very intuitive brute-force solution that can be broken down into the following steps:

1. looping through each element at index i of A
2. finding the position j in B where the value A_i appears i.e. $B_j = A_i$ (which exists because $A \subset B$)
3. finding the smallest element larger than A_i in B only considering those positions strictly after j .

An implementation of this approach is shown in Listing 52.1 where we use `std::find` to the location in B (the iterator `it`) where A_j exists. The subsequent `while` is used to scan the remainder of the array and to keep track of the smallest element that is larger than A_i . The complexity of this approach is $O(|A| \times |B|)$ as we could potentially do linear work (proportional to $|B|$) for each element of A . One such case is when the elements of A appear in the first positions of B .

```
1 std::vector<int> next_greater_element_I_bruteforce(const std::vector<int>& A,
2                                                    const std::vector<int>& B)
3 {
4     std::vector<int> C(A.size());
5     for (int i = 0; i < std::ssize(A); i++)
6     {
7         auto it = std::find(std::begin(B), std::end(B), A[i]);
8         int ans_i = -1;
9         while (it != B.end())
10        {
11            if (*it > A[i])
12                ans_i = (ans_i == -1) ? *it : std::min(ans_i, *it);
13            it++;
14        }
15        C[i] = ans_i;
16    }
17    return C;
18 }
```

Listing 52.1: Brute-force solution to the *next smaller element*.

52.3 $O(|B|\log(|B|))$ time, $O(|B|)$ space solution

We can solve this problem much faster than quadratic time if, as is often the case, we are willing to use some additional space. In particular the problem is easily solved if we have

a map containing the information about the next greater element for each of the elements of B . We could then simply loop over all the elements of A and query such a map to get the required answer. However that does raise the question: how can we generate such a map?

The idea is that we can fill the map for each element of B starting from the back and at the same time keep a sorted list of all the elements of B that we have already processed. This list can be used to quickly (by doing a binary search on it) find the upper bound for a given value. The upper bound for an integer x is the first (or smallest) element in a collection that is strictly larger than x . The upper bound operation can be easily implemented on a sorted collection using binary search. We have already implemented a similar operation (the lower bound) in Chapter 35 and you can check Listing 35.4 (at page 198) to have an idea of how you can go about brewing your own version of `upper_bound`.

The idea described above is implemented in Listing 52.2. The `std::set<int> N` contains the sorted list of elements of B that we have already processed while the `std::unordered_map<int,int> C_val` contains the information about the upper bounds for each of the processed elements of B . The first `for` goes through each element j of B (from the back to the front) and calculates the answer for B_j by looking into the sorted `std::set N`^②. The second `for` loop only takes care of copying the relevant information from the map `C_val` to the return array. The time and space complexity of this code are $O(|B|\log(|B|))$ (each of the $|B|$ insertions in N costs $O(\log(|B|))$ and $O(|B|)$, respectively).

```

1  std::vector<int> next_greater_element_I_set(const std::vector<int>& A,
2                                             const std::vector<int>& B)
3  {
4      std::vector<int> C(A.size());
5      std::set<int> N;
6      std::unordered_map<int, int> C_val;
7
8      for (int idx_B = B.size() - 1; idx_B >= 0; idx_B--)
9      {
10         auto it = N.upper_bound(B[idx_B]);
11         C_val[B[idx_B]] = it != N.end() ? *it : -1;
12         N.insert(B[idx_B]);
13     }
14     for (int i = 0; i < std::ssize(A); i++)
15     {
16         C[i] = C_val[A[i]];
17     }
18     return C;
19 }
20 }
```

Listing 52.2: $O(n\log(n))$ time and linear space solution.

52.4 Common Variation

52.4.1 First next greater element

There is a common variation of this problem featuring an almost identical statement to the one shown in Section 62.1 with the only difference being that for an element of A_i we should return the first (and not the necessarily the smallest as in the original variant) element in B that is greater than A_i .

^②Note that we do not use the free function `std::upper_bound` because on non linear data structures (like `std::set`) it operates in linear time instead of logarithmic time.

Problem 75 Write a function that - given two arrays with no duplicates A and B where $A \subset B$ - returns an array C of size $|A|$ where C_i contains the first element greater than A_i among the elements of B strictly after the cell at index j , where $B_j = A_i$.

In other words, for each A_i the function finds the first element in B that is greater than A_i among the cells that are to the right of the cell in B having the value A_i and places it into C at index i .

52.5 Discussion

The difference with the original variation is minimal but big enough such that we have a linear-time solution for this version of the problem. While in solving the original problem we were forced to keep a sorted list of all the already processed elements of B , this time we can simply keep a stack storing only those processed elements of B so that they form an increasing sequence.

Suppose we have a decreasing sequence followed by a greater number. For example, consider the following list: $\{7, 8, 5, 4, 3, 2, 1, 6\}$ (see Figure 52.1); initially the stack is empty and when we process the first number (6) there is clearly no greater element to its right. As the stack is empty, adding 6 to it would still preserve the fact that the numbers contained in it form an increasing sequence (see Figure 52.1a). When the 1 is processed then the stack is not empty and 6 is at the top which is larger than 1. Therefore we can use 6 as an answer for 1 and add 1 to the stack because the sequence 1, 6 is still increasing (see Figure 52.1b). Things however, are a bit different when 2 is processed. This time at the top of the stack we find a 1 which is smaller than 2. As such, the top of the stack cannot be the answer for the element 2. Moreover the sequence 2, 1, 6 would not be increasing and therefore the two cannot be placed on top of the stack as-is. What we do here is remove the elements from the current stack until placing 2 at the top would make the elements in the stack an increasing sequence. So we remove 1 from the stack and the new stack becomes 2, 6 (see Figure 52.1c). The rest of the execution is described in more detail in Figure 52.1.

From this example we can draw a general approach to solving this problem using a stack. When we process an element we try to insert it into the stack paying attention to how this element compares to the top of the stack. If it is larger then we remove the top of the stack and compare it again with the subsequent element. We keep repeating and removing elements from the stack until either the element we are trying to place is smaller than the top of the stack or there are no more elements left in the stack. In the former case then the new top of the stack (after all necessary removals) is going to be the answer associated with the element we are processing. In the latter case the answer does not exist and the element we are trying to place on the stack is therefore the largest processed so far. Listing 52.3 shows an implementation of this idea.

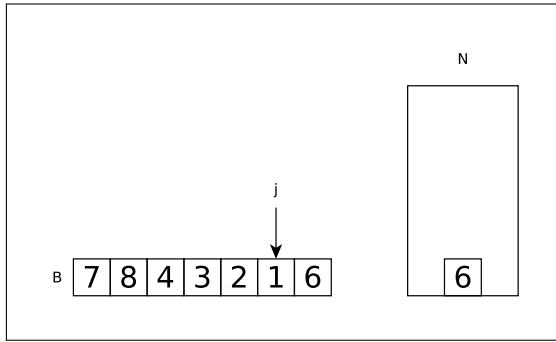
```
1  std::vector<int> next_greater_element_I_stack(const std::vector<int>& A,
2                                              const std::vector<int>& B)
3  {
4      std::vector<int> C(A.size());
5      std::stack<int> N;
6      std::unordered_map<int, int> C_val;
7
8      for (int i = std::ssize(B) - 1; i >= 0; i--)
9      {
10         while (!N.empty() && B[i] > N.top())
11         {
```

```

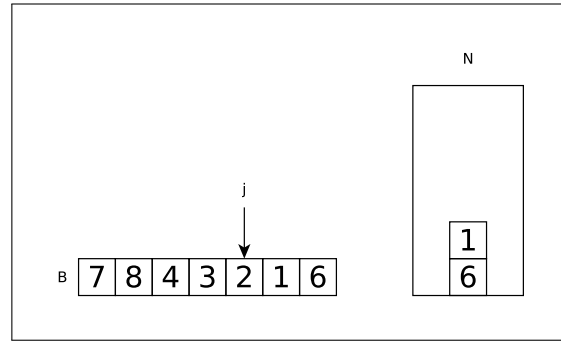
12     N.pop(); // remove smaller elements than *it
13 }
14 // now the stack is either empty or contains an increasing sequence
15 if (!N.empty())
16     C_val[B[i]] = N.top();
17 N.push(B[i]);
18 }
19 for (int i = 0; i < std::ssize(A); i++)
20 {
21     if (C_val.contains(A[i]))
22         C[i] = C_val[A[i]];
23     else
24         C[i] = -1;
25 }
26 return C;
27 }

```

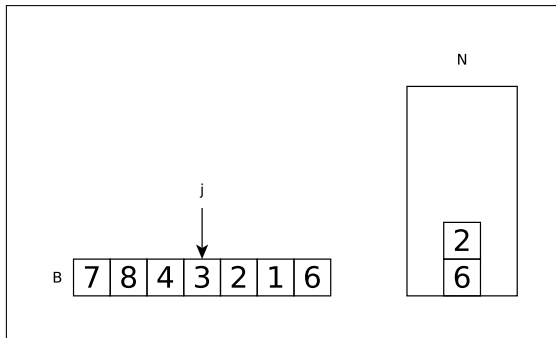
Listing 52.3: linear time solution to the Problem 75 solved using a stack.



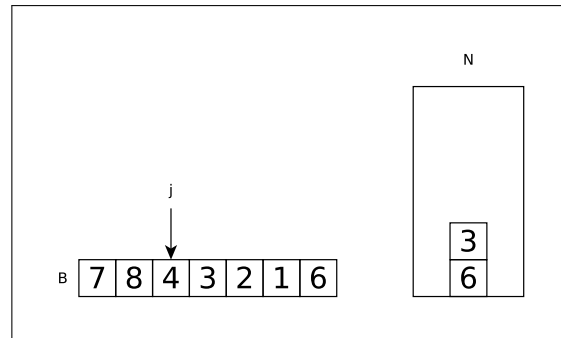
(a) The stack is empty. We place 6 at the top.



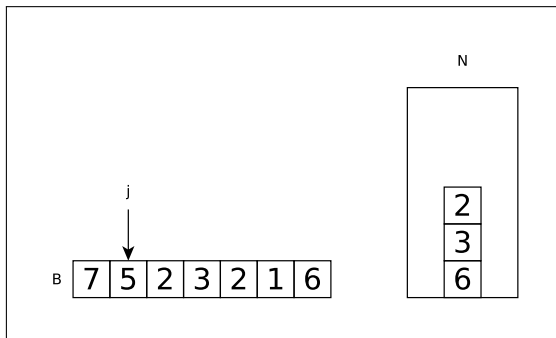
(b) 1 is smaller than the top of the stack therefore 1 is placed at the top. 6 is the answer for 1.



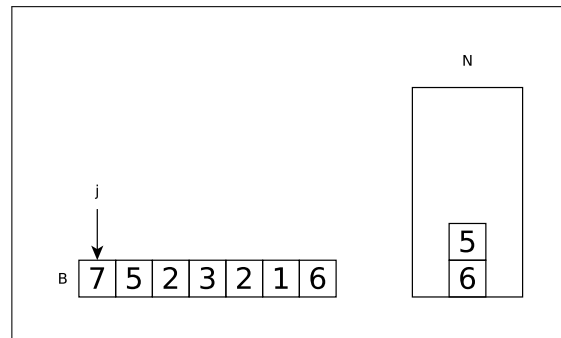
(c) The top of the stack 1 is smaller than 2. We therefore remove 1 and place [WHAT]? at the top. 6, is therefore the answer for 2



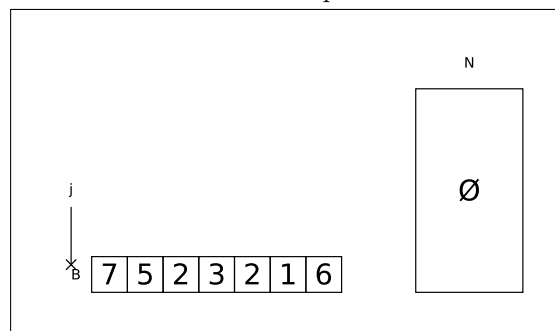
(d) Similarly to what we did in Figure 52.1c we remove all the elements at the top until adding the 3 would preserve the increasing ordering of the stack elements. 2 is removed and 3 is the new top. 6 is the answer for 3.



(e) We add the 2 to the stack and return 3 (the current top of the stack) as the answer for 2.



(f) 5 is larger than the first two elements of the stack which are therefore removed. 5 is the new top and 6 is the answer for 5.



(g) 7 is larger than all the elements currently in the stack. Therefore all the elements are removed and the stack remains empty signalling that 7 has no greater elements to its right. The process ends here as there are no more elements to process.

Figure 52.1

53. Count the bits

introduction

Computers use binary numeral system to store data and perform computations. In this system a digits are called bits and they are the smallest unit of data and can only have value 0 or 1. By using several bits we are able to encode information such as documents, music, books, maps, bitcoins, etc. Ultimately the binary numeral system is just a way of representing numbers and it is not really different from the decimal system we use everyday. A binary number is made up of one or more bits, the same way a decimal number is made up of several 0–9 digits. For instance, the binary number $111111001000011111010110_2$ correspond to the decimal number 16549846.

In programming, binary numbers are often used as bitsets^① as a more efficient and memory cheat substitute to arrays of booleans; such use case is so common that in the C++ STL we even have a dedicated class: `std::bitset`. However a simple `int` can be used as a bitset and in this chapter we will calculate some statistics on the number of bits that are set to `true` in a `int/bitset` (pretty much equivalently to what the function `std::popcount(T x)` does) for a range of numbers.

This problem is aimed at testing our basic bit manipulation skills and all we need to get a woking solution is knowing how to use some basic bit logic operations and functions like *shift* and *AND*. Besides this basic solution we will also have a look at two more sophisticated and efficient solutions: the former based on DP and the second based on a property of powers of two.

53.1 Problem statement

Problem 76 Given a non negative integer number n return an array B of size $n + 1$ where B_i contains the number of bits set in the number i .

■ **Example 53.1**

Given $n = 5$ the function returns $B = \{0, 1, 1, 2, 1, 2, 2\}$ ■

53.2 Clarification Questions

Q.1. Can we assume n is always positive?

Yes, $n \geq 0$.

Discussion

53.2.1 Naïve approach solution

This is an easy problem. All we have to do is to use brute-force to count the number of bits set in each and every number 0 to $n + 1$. Each number has a fixed size which on most

^①An array (usually of fixed size) of bits.

the common modern C++ implementation is 32-bit (`sizeof(int)`) and therefore, we can come up with a $\Theta(32n)$ solution.

Counting the bits of a given integer can even be done with compiler intrinsics as `__builtin_popcount` which can map directly when supported by the hardware to fast machine instructions or by using some simple bit manipulation trickery. From C++ -20 we can also use the `std::popcount` function, together with a several other bit related functions (in the header `<bit>`).

Listing 53.1 shows an implementation of this idea where we use our own version of the bit counting function `my_pop_count` for the sake of showing how `std::popcount` works and to be ready in (likely) case the interviewer asks us to dig deeper in this direction.

The function `my_pop_count` works by repeatedly inspecting the least significant bit of the input `num` to check if it is set or not and then, it shifts it to the right by one position so that the at the next iteration another bit is inspected. The value of the least significant bit of a integer `n` can be retrieved by using the following expression: `n & 1`. The operator `&` performs the bitwise AND between `n` and 1. Because 1 is a number having only the least significant bit set (its binary representation is, assuming integers have size 32 bits `00000000000000000000000000000001`), the result of the aforementioned expression is `true` when the least significant bit of `n` is set, and false otherwise: the bitwise AND between every other bits other than the least significant of `n` and 1 is always 0.

```
1 unsigned my_pop_count(unsigned num)
2 {
3     int ans = 0;
4     while (num)
5     {
6         ans += num & 1; // is the last bit set
7         num >>= 1;      // shift num one bit to the right
8     }
9     return ans;
10 }
11
12 std::vector<int> count_bits_bruteforce(const unsigned n)
13 {
14     std::vector<int> ans;
15     ans.reserve(n + 1);
16     for (unsigned num = 0; num <= n; num++)
17     {
18         // alternatively std::popcount or __builtin_popcount (only on gcc)
19         const int num_bit_set = my_pop_count(num);
20         assert(std::popcount(num) == num_bit_set);
21
22         ans.push_back(num_bit_set);
23     }
24     return ans;
25 }
```

Listing 53.1: Brute-force solution where we manually count the number of bits for each number.

53.2.2 DP solution

This problem can be solved more elegantly and efficiently using dynamic programming. In the approach discussed in this section we will see how we can craft a solution that is correct and does not incur a factor 32 penalty the solution shown in Listing 53.1 costs).

The idea is that the number of bits set for a given number `n` is equal to the number of bits set in `n << 1`, `n` shifted one position to the right (`n` with the last bit removed, a

number that is always smaller or equal than the number we started with), plus one if the removed bit was 1.

For instance consider $x = 2730_{10} = 101010101010_2$. The least significant bit of x is 0 therefore its number of bits set is equal to the number of bits set of $y = 1365_{10} = 10101010101_2$ (last bit of x removed). For the same reasons the number of bits set in y is one (because the last bit of y is 1) plus the number of bits set in $y = 682_{10} = 1010101010_2$ (last bit of y removed). We can follow this line of reasoning until we reach 0 that has zero bits set.

Given that every time we remove a bit we are solving a problem for a smaller number and, because the solution for a number x can be required to count the bits of many numbers $n > x$, we can adopt DP (see Appendix 64): this problem exposes optimal substructures as well as overlapping subproblems properties. In a DP solution we will use a DP table B containing the information about the number of bits set for the numbers, which we initially fill only for the number 0. We will then follow a bottom-up approach where we start solving problems for $x = 1, 2, \dots, n$. When we reach a given number y we have already solved and stored into B the answers for every number less than y , and at that point we are ready to calculate the answer for y . Because the answers for all of these numbers smaller than y are stored in B we do not need to recompute them.

Listing 53.2 shows an implementation of this approach. A slightly shorter possibly less readable version of Listing 53.2 is shown in Listing 53.3.

```

1  std::vector<int> count_bits_DP(const unsigned n)
2  {
3      std::vector<int> B;
4      B.reserve(n + 1);
5      B.push_back(0);
6      for (unsigned num = 1; num <= n; num++)
7      {
8          const unsigned last_bit      = num & 1;
9          const unsigned pop_count_rest = B[num >> 1];
10         B.push_back(last_bit + pop_count_rest);
11     }
12     return B;
13 }
```

Listing 53.2: DP solution where we calculate the bits for a given number from the its last bit and the answer of the number resulting from removing that last bit.

```

1  std::vector<int> count_bits_DP_short(const unsigned n)
2  {
3      std::vector<int> B(n + 1, 0);
4      for (unsigned num = 1; num <= n; num++)
5          B[num] = B[num >> 1] + (num & 1);
6      return B;
7  }
```

Listing 53.3: Shorter version of Listing 53.2.

53.2.3 Another efficient approach

There is another way of approaching this problem that is quite different yet as fast from the DP solution we discussed above.

Let's start by noticing that any power of 2 always has one and only one bit set. For instance, 2^2 has the bit at index 2 set and the rest of the bits not set. The same applies for any other power of two 2^k where only the bit at index k is set. All the numbers from 2^k to $2^{k+1} - 1$ can be obtained by concatenating a bit set (the 1 at index k) as a prefix with

all the binary representations of the numbers from 0 to $2^k - 1$ (all the numbers smaller than 2^k).

For instance let's take $k = 4$ as an example. All the numbers from $2^4 = 16$ to $2^5 - 1 = 31$ can be obtained as shown below:

- $16 = 16 + 0 = 10000_2 + 0_2$
- $17 = 16 + 1 = 10000_2 + 1_2$
- $18 = 16 + 2 = 10000_2 + 10_2$
- $19 = 16 + 3 = 10000_2 + 11_2$
- $20 = 16 + 4 = 10000_2 + 100_2$
- $21 = 16 + 5 = 10000_2 + 101_2$
- $22 = 16 + 6 = 10000_2 + 110_2$
- $23 = 16 + 7 = 10000_2 + 111_2$
- $24 = 16 + 8 = 10000_2 + 1000_2$
- $25 = 16 + 9 = 10000_2 + 1001_2$
- ...
- $31 = 16 + 15 = 10000_2 + 1111_2$

This fact allows us to calculate the answer for all the numbers from 16 to 31 by adding one to the answer of the numbers from 0 to 15 for which, crucially, we already have an answer.

The same applies for smaller k s. For $k = 2$ we have:

- $4 = 4 + 0 = 100_2 + 0_2$
- $5 = 4 + 1 = 100_2 + 1_2$
- $6 = 4 + 2 = 100_2 + 10_2$
- $7 = 4 + 3 = 100_2 + 11_2$

We can use this idea to build a fast and efficient solution as shown in Listing 53.4.

```

1  std::vector<int> count_bits_DP_powers(const unsigned n)
2  {
3      std::vector<int> B(n + 1, 0);
4      for (int i = 1, next_pow = 2, back_idx = 0; i <= n; i++)
5      {
6          if (i == next_pow)
7          {
8              // next power of two.
9              next_pow *= 2;
10             // we use all the solutions from the start to build the next ones
11             back_idx = 0;
12         }
13         B[i] = 1 + B[back_idx++];
14     }
15     return B;
16 }
```

Listing 53.4: Alternative efficient solution where the number of bits set in a integer k is found by using the number of bits set for a smaller integer: $k - (2^{\lfloor \log_2 k \rfloor})$ (see Equation 53.1).

The answers are calculated incrementally starting with the integers 0, 1 and 2 which have 0, 1 and 1 bits set, respectively. Then we can calculate the answer for 2 and 3 (from 2^2 to $2^3 - 1$) by adding 1 to the answers for 0 and 1. For the numbers from 4 to 7 (2^2 to $2^3 - 1$) we add 1 to the answers to 0, 1 and 2 and 3, respectively. For the numbers from 8 to 15 (2^3 to $2^4 - 1$) we add 1 to the answers for 0, 1, ..., 7. We keep doing this, adding 1 to all the numbers from 0 to $2^k - 1$ in order to calculate the answer for all the numbers from

2^k to $2^{k+1} - 1$, until we reach $n + 1$. The complexity of this approach is $\Theta(n)$ and also in this case we do not pay the constant factor associated with a brute-force count of the bits in an integer.

Note that the same approach can be easily adapted to obtain a top-down implementation where we memoize and reuse the answers using a cache. We leave this as an exercise for the reader ^②.

^②The recurrent relation to the number of bits set in k is as shown in Equation 53.1 where $B(x)$ is a function returning the number of bits set in the binary representation of the integer x :

$$\begin{cases} B(0) = 0 \\ B(1) = 1 \\ G(k) = 1 + G(k - (2^{\lfloor \log_2 k \rfloor})) \end{cases} \quad (53.1)$$

54. Decode the message

Introduction

The problem in this chapter resembles the one for decoding a string encoded with the famous *run-length encoding method* (RLE). RLE is a simple form of data compression in which a stream of data is given (e.g. "AAABBCCCC") and the output is a sequence of counts of consecutive data values in a row. (e.g. "3A2B4C"). It is a type of lossless encoding meaning that the encoding process does not lose any information in the original input and therefore the input data can be recovered fully and integrally decoded.

This chapter will deal with a similar algorithm where we will be asked to write a function capable of decoding a string encoded with a run-length-encoding-like algorithm. More than complicated insights, string manipulation skills and attention to details of the implementation and for corner cases are going to be crucial in order to solve this problem during an actual interview.

54.1 Problem statement

Problem 77 Write a function that given an encoded string s decodes it. s is of the form: $k_1[d_1]k_2[d_2][\dots]$ where k is a positive integer and s is another encoded string. The decoded version of s is obtained by appending d_1 k_1 followed by repeating d_2 k_2 times.

■ **Example 54.1**

Given $s = "2[abc]3[ab]"$ the function returns $"abcbabababababab"$. ■

■ **Example 54.2**

Given $s = "2[abc3[ab]]"$ the function returns $"abcbabababababab"$. ■

■ **Example 54.3**

Given $s = "2[abc]3[cd]ef"$ the function returns $"abcbcccdcdcdcdcd"$. ■

54.2 Clarification Questions

Q.1. Is it guaranteed that s is always valid?

Yes, s contains only lower-case letters from the English alphabet, numbers and square brackets and it is a valid encoded string.

Q.2. Is there an upper-bound on the size of the encoded-substrings (the k s in the problem statement)?

Not really, you are only guaranteed their value to fit into a built-in `int`.

Discussion

54.2.1 Recursive solution

The first thing to note about this problem is that the encoded string has a recursive definition. Whenever we encounter a number k followed by the closed square bracket character `']'` we know that we have to decode whatever is inside the brackets and replicate it k times. We can use this fact to simply create a recursive algorithm which follows this definition. The real challenge of this problem actually lies in the implementation more than in the algorithm itself and in our opinion specifically in the correct parsing of the string.

The idea is to construct the final answer by looking at one character of s at a time. We can start from the char at index 0 and depending on what it is:

1. append it to the answer (when s is an alphabetic letter);
2. parse it as a part of a number (when s is a digit);
3. recursively decode the rest of the string (when s is an open square bracket `'['`).

For instance, let's assume we have to decode $s = \text{"xy242[ab3[c]de]"}.$ We start by reading `'x'` and `'y'` which are letters and therefore are just appended to the final answer. We then see a digit which tells us that a number has started. We parse all of its digits into the integer 242 (which assumes we need to perform a conversion from string to int; see Chapter 7 at page 31 where we delved into it for a refresher). The end of the number is signaled by the presence of the char `'['` which also signals that a new encoded substring is starting. So when we see an open square bracket character we recursively call the decode function so that it returns the expansion of whatever is within the brackets. When the recursive call ends (when we find a closed square bracket character `']'`) we are left with the expanded string which we can then replicate 242 times and append to the final answer. When a recursive call ends the caller must continue processing the elements of s from the last unprocessed character. We keep track of the next element to be processed via a integer variable which is passed along to each of the recursive calls. This is necessary because after the recursive call returns we might need to continue processing more characters. Regarding the example above, when the recursive call associated with the substring `3[c]` returns we still have to process `de` for the encoded substring `ab3[c]de`.

Listing 54.1 shows a possible implementation of this idea.

```
1  std::string decode_string_recursive_helper(const std::string& s, std::size_t& i
    )
2  {
3      const auto size = s.size();
4      std::string ans;
5      int multiplier = 0;
6      while (i < size)
7      {
8          const auto curr_char = s[i];
9          if (std::isdigit(s[i]))
10         {
11             // parse the whole number
12             while (i < size && std::isdigit(s[i]))
13             {
14                 multiplier *= 10;
15                 multiplier += s[i] - '0';
16                 i++;
17             }
18         }
19         else if (s[i] == '[')
20         {
21             const std::string nested = decode_string_recursive_helper(s, ++i);
22             for (int k = 0; k < multiplier; k++)
```

```

23     ans += nested;
24     // no increment of i here.
25 }
26 else if (s[i] == ']')
27 {
28     i++;
29     break;
30 }
31 else
32 {
33     ans += s[i];
34     i++;
35 }
36 }
37 return ans;
38 }
39
40 std::string decode_string_recursive(const std::string& s)
41 {
42     std::size_t pos = 0;
43     return decode_string_recursive_helper(s, pos);
44 }

```

Listing 54.1: Recursive implementation of the algorithm described in Section 54.2.1

Notice that the function `decode_string_recursive_helper` takes the second parameter as a reference to an integer. As mentioned above already, we use a reference because we want this number to be updated by each of the recursive calls and this way we do not lose track of what portion of the input is still to be processed. The variable i to keep track of the current character we are examining in s . Figure 54.1 graphically depicts and describes the execution of this algorithm on the input string `ab23[x12[w]]cd`.

The time and complexities of the code in Listing 54.1 is $O(K|s|)$ where k is the largest replication factor we can have.

54.2.2 Iterative solution

The same idea can of course be implemented iteratively. The trick is to *simulate* the call stack of the recursive approach in Section 54.2.1 by using an explicit stack. This stack will contains two pieces of information 1. the replication factor associated with an encoded substring (the value 1 is the default) 2. the decoded substring.

Initially the stack contains only one entry: `(1,"")`. As in the recursive approach, we process s one character at a time and all the operations are performed on the **top** of the stack unless we encounter either a:

- '[' which signals we need to add another element to the stack and start decoding a substring of s .
- ']' which signals that we are done with decoding the substring and we can then replicate it as many time as is necessary, remove the entry from the top of the stack and append the replicated string to the string associated with the new top of the stack.

At the end of this process we are left with the fully decoded string at the top of the stack.

Listing 54.2 implements this idea.

```

1 std::string decode_string(const std::string& s)
2 {
3     std::stack<std::pair<int, std::string>> stack;
4     stack.push({1, std::string()});
5     size_t i = 0;

```

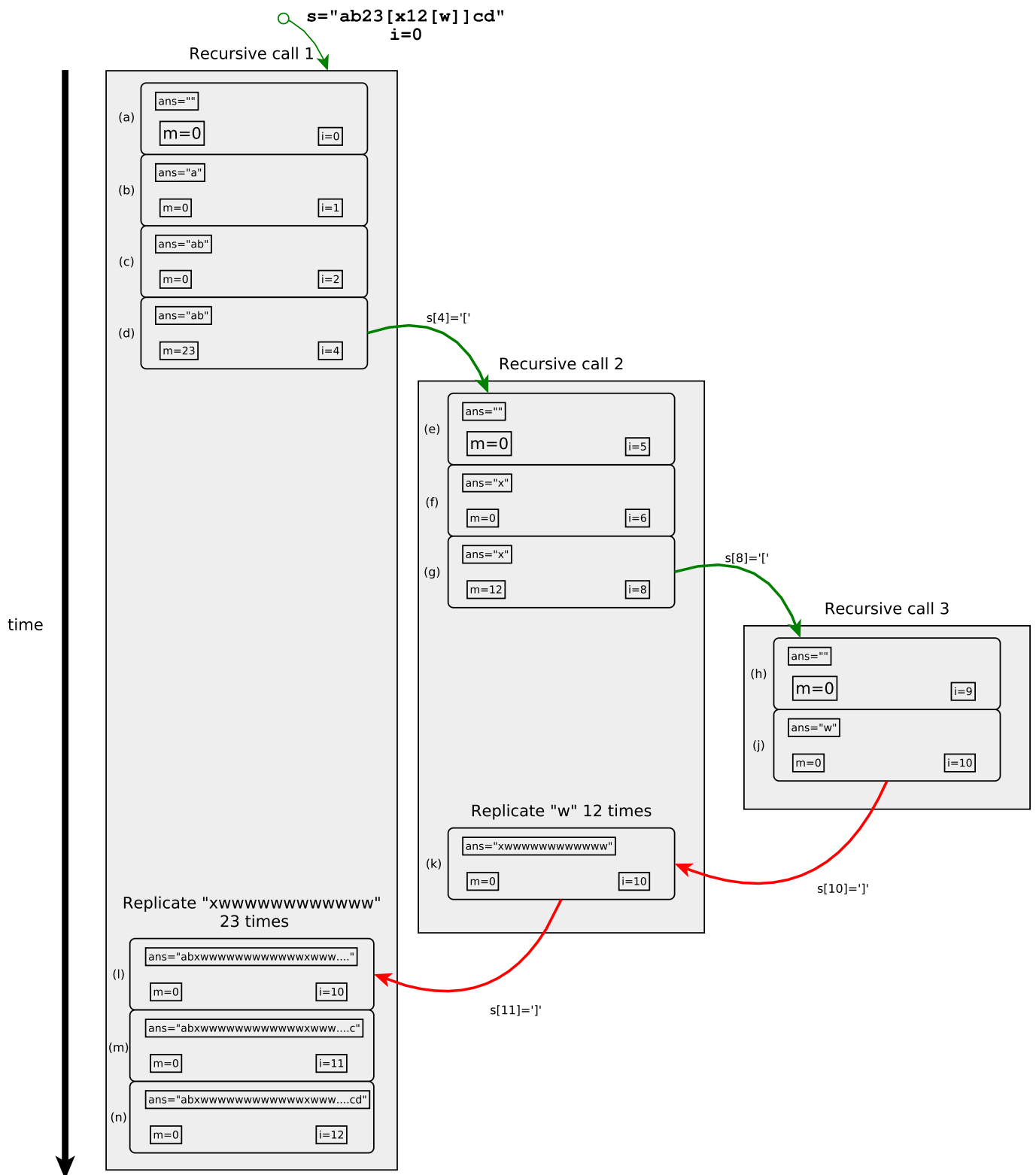



Figure 54.1: Execution of the algorithm in Listing 54.1 for the input string $s = \text{"ab23[x12[w]]cd"}$. The execution starts (Figure (a)) with a first call to `decode_string_recursive_helper` with $i = 0$. The first two characters are letters and therefore they are appended to the instance of `ans` bounded to this recursive call (Figures (a) and (b)). Characters at indices 2 and 3 are numbers and they are parsed and saved into the integer `m` (Figure (c)). The next character is going to be the open square bracket at index 4 and this will cause a new recursive call to happen with $i = 5$ (Figure (d)). The process repeats now and we see at index 5 a letter that we append to the instance of `ans` bound to this call (Figures (e)). We then parse the number 12 at characters 6 and 7 (Figure (f)) and subsequently at index 8 we find an open square bracket that leads to a new recursive call, this time starting from index $i = 9$ (Figure (g)). Index 9 holds a letter which is appended to the (empty) instance of `ans` bound to this call (Figure (h)). The next character is a closed square bracket which means we can terminate the recursive call and return to the caller `ans` (which now holds `"w"`). We are not back (Figure (k)) to the second recursive call where, as you remember, `m` is twelve. Therefore we replicate 12 times the string we received from the recursive call number three and append the result to the current instance of `ans`. We also set `m` to zero. Because the next character is again a closed bracket we return to the caller and repeat the process (Figure (i)). `m = 23` and therefore we replicate `ans` from the second recursive call 23 times. The rest of the characters left are the ones at indices 11 and 12 which are simple letters and are just appended to `ans`.

```

6  int rep_f = 0;
7  while (i < s.size())
8  {
9      if (std::isdigit(s[i]))
10     {
11         rep_f *= 10;
12         rep_f += (s[i] - '0');
13     }
14     else if (s[i] >= 'a' && s[i] <= 'z')
15     {
16         stack.top().second.push_back(s[i]);
17     }
18     else if (s[i] == ']')
19     {
20         const auto [rep, str] = stack.top();
21         stack.pop();
22         for (int i = 0; i < rep; i++)
23             stack.top().second += str;
24     }
25     else if (s[i] == '[')
26     {
27         stack.push({rep_f, std::string()});
28         rep_f = 0;
29     }
30     i++;
31 }
32 assert(stack.size() == 1);
33 return stack.top().second;
34 }

```

Listing 54.2: Iterative solution using a `std::stack`.

Notice that the stack has type `std::stack<std::pair<int, std::string>>` thus reflecting the fact we need to keep two pieces of information for each of the encoded substrings. As for the recursive implementation, both time and space complexities are $O(K|s|)$.

55. N-Queens

Introduction

In this chapter we are going to discuss a problem that is best known for one of its specializations; namely the eight queen puzzle. In the eight queen puzzle the challenge is to place eight chess queens on an 8×8 chessboard so that no two queens threaten each other thereby requiring that no two queens share the same row, column or diagonal.

The n-queens is a classical problem which is used to test various programming techniques such as constraint programming, logic programming, recursion and even genetic algorithms. In the context of a programming interview, you should aim for precision and for outlining the solution strategy clearly while solving it. The interviewer is very likely expecting you to be familiar with the problem already, and possibly also with the solution but they will want to see how well you can explain and materialize in code all the steps between a purely brute-force and a more sophisticated solution.

55.1 Problem statement

Problem 78 The n -queens puzzle is the problem of placing n queens on an $n \times n$ chessboard in such a configuration that no two queens can attack each other. A queen can attack any piece along its row, column or diagonals. Write a function that, given an integer $1 \leq n$, returns all solutions to the n -queens puzzle.

A solution is to be returned as a snapshot of the chessboard. A snapshot of the board is an array of n strings, each representing a row, where an empty cell is denoted by the symbol '.' (dot) and a cell occupied by a queen by the letter 'Q'.

For instance, the following is the snapshot of the board shown in Figure 55.1:

```
["....Q...", "...Q...", "...Q....", "Q.....", "..Q....", ".....Q",  
".....Q..", ".Q....."]
```

■ Example 55.1

Given $n = 4$ the function returns an array containing the following snapshots:

1. [".Q..", "...Q", "Q...", "..Q."] (see Figure 55.2a)
2. ["..Q.", "Q...", "...Q", ".Q.."] (see Figure 55.2b)

55.2 Discussion

The queen (♚) in the game of chess is the most powerful piece as it is able to move any number of unoccupied cells vertically, horizontally and diagonally, as shown in Figure 55.3. Its movement pattern is the combination of the moves of the rook (♖) and the bishop (♗).

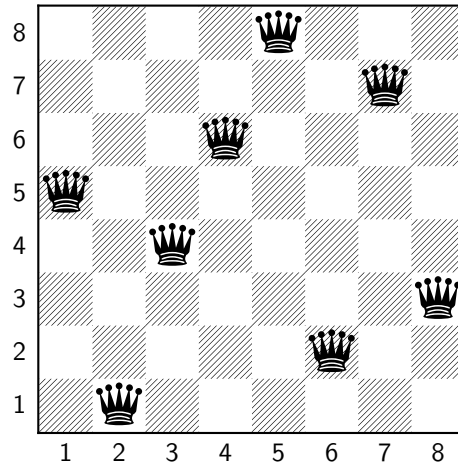


Figure 55.1: Example of solution of the 8-queens problem



Figure 55.2: Solutions to the 4-queens problem.

55.2.1 Brute-force

A purely brute-force approach can be incredibly expensive as there are $\binom{n^2}{n}$ possible ways of placing n queens on a $n \times n$ board. For an 8×8 board that translates to a whopping 4426165368 possible arrangements but only 92 distinct solutions! This approach is extremely easy to implement as the only thing required is to be able to enumerate all the queens' arrangements and filter out the ones in which a queen can attack any of the others. This last step can be done by checking, for each queen, whether another queen is placed in its row, column or diagonals.

Listing 55.1 shows an implementation of this idea where we use the function `nqueen_bruteforce` as a driver, the function `nqueen_bruteforce_helper` to generate all possible arrangements for the queens and the function `is_valid_solution` to evaluate it.

An arrangement is a combination of size n taken from a list of all possible board cells `location`. The function `nqueen_bruteforce_helper` works similarly to the function `all_combinations` in Listing 42.1 (at page 234 in Chapter 42) with the difference that the combination is just evaluated to see if it is a valid solution and only saved if such test is positive.

Function `is_valid_solution` takes care of validating a solution candidate by checking

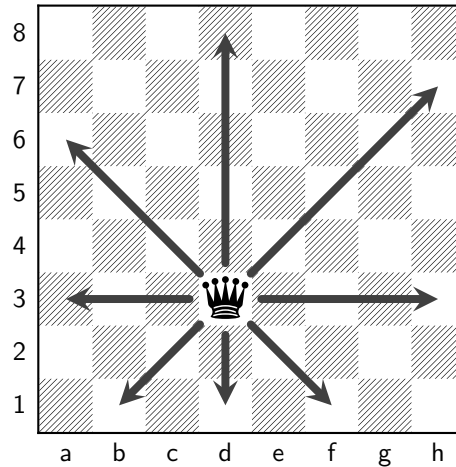


Figure 55.3: Moves of a chess queen.

whether no two queens placed at locations (a,b) and (c,d) share the same:

- row (when $a = c$ or $a - c = 0$)
- column (when $b = d$ or $b - d = 0$)
- diagonal (either when:
 - $a - c = b - d$, meaning that you can reach (c,d) by advancing by the same number of rows and column from (a,b)
 - $|a - c| = |b - d|$ and $\text{sign}(a - c) \leq \text{sign}(b - d)$ which has the geometrical interpretation that you can reach (c,d) by advancing by the same number of rows and columns from (a,b) but in opposite directions: advance k rows and step back k columns or the other way round.

```

1  using ChessBoardSnapshot = std::vector<std::string>;
2  using CellCoordinate     = std::pair<int, int>;
3  using Solution           = std::vector<CellCoordinate>;
4
5  ChessBoardSnapshot make_chessboard_snapshot(const Solution& solution)
6  {
7      ChessBoardSnapshot ans(solution.size(), std::string(solution.size(), '.'));
8
9      for (const auto& queen_location : solution)
10     {
11         const auto [row, col] = queen_location;
12         ans[row][col] = 'Q';
13     }
14     return ans;
15 }
16
17 bool is_valid_solution(const Solution& solution)
18 {
19     const auto sgn = [](auto val) {
20         using T = decltype(val);
21         return (T(0) < val) - (val < T(0));
22     };
23
24     for (int i = 0; i < std::ssize(solution); i++)
25     {
26         for (int j = i + 1; j < std::ssize(solution); j++)

```

```

27     {
28         const auto rows_difference = solution[i].first - solution[j].first;
29         const auto cols_difference = solution[i].second - solution[j].second;
30
31         const bool same_row    = rows_difference == 0;
32         const bool same_col    = cols_difference == 0;
33         const bool same_diag1 = rows_difference == cols_difference;
34
35         const bool same_diag2 =
36             std::abs(rows_difference) == std::abs(cols_difference)
37             && sgn(rows_difference) != sgn(cols_difference);
38         if (same_row || same_col || same_diag1 || same_diag2)
39             return false;
40     }
41 }
42 return true;
43 }
44
45 void nqueen_bruteforce_helper(const unsigned n,
46                             const std::vector<CellCoordinate>& locations,
47                             const int location_idx,
48                             Solution& solution_candidate,
49                             std::vector<ChessBoardSnapshot>& ans)
50 {
51     if (solution_candidate.size() >= n)
52     {
53         if (is_valid_solution(solution_candidate))
54             ans.push_back(make_chessboard_snapshot(solution_candidate));
55
56         return;
57     }
58
59     for (int i = location_idx; i < std::ssize(locations); i++)
60     {
61         solution_candidate.push_back(locations[i]);
62         nqueen_bruteforce_helper(n, locations, i + 1, solution_candidate, ans);
63         solution_candidate.pop_back();
64     }
65 }
66
67 auto nqueen_bruteforce(const unsigned n)
68 {
69     std::vector<CellCoordinate> locations;
70     locations.reserve(n * n);
71     for (unsigned i = 0; i < n; i++)
72         for (unsigned j = 0; j < n; j++)
73             locations.push_back(CellCoordinate(i, j));
74
75     Solution sol_candidate;
76     std::vector<ChessBoardSnapshot> ans;
77     nqueen_bruteforce_helper(n, locations, 0, sol_candidate, ans);
78     return ans;
79 }

```

Listing 55.1: Bruteforce solution where all possible queens arrangements on the board are enumerated.

55.2.2 One row one queen

The purely bruteforce solution goes through a number of arrangements that are by construction invalid. We know that a queen placed on a given row k makes it impossible for

another queen to be placed in the same row. Therefore, we can place a queen at row i and then try to place the remaining queens on the rest of the board without considering row i again. This approach significantly reduces the number of possible arrangements as there are n^n ways of doing so (n choices per each row). Listing 55.2 shows an implementation of this idea.

```

1
2
3 void nqueen_one_per_row_helper(const unsigned n,
4                               const int current_row,
5                               Solution& solution_candidate,
6                               std::vector<ChessBoardSnapshot>& ans)
7 {
8     if (current_row >= n)
9     {
10         if (is_valid_solution(solution_candidate))
11             ans.push_back(make_chessboard_snapshot(solution_candidate));
12         return;
13     }
14
15     // try to place a queen in each of the column of this row
16     for (unsigned column = 0; column < n; column++)
17     {
18         solution_candidate.push_back(CellCoordinate(current_row, column));
19         nqueen_one_per_row_helper(n, current_row + 1, solution_candidate, ans);
20         solution_candidate.pop_back();
21     }
22 }
23
24 auto nqueen_one_per_row(const unsigned n)
25 {
26     Solution sol_candidate;
27     std::vector<ChessBoardSnapshot> ans;
28     nqueen_one_per_row_helper(n, 0, sol_candidate, ans);
29     return ans;
30 }

```

Listing 55.2: Solution to the n-queens problem where we do not place more than one queen on the same row.

55.2.3 One queen per column

The approach discussed in Section 55.2.2 is still generating arrangements that are invalid by construction and the reason is that it tries all those arrangements where two queens are placed on the same column. We can prevent this from happening by using the reusing the idea of placing one queen per row by also placing one queen per column. This can be achieved by using one of the $n!$ permutations of the numbers $0, 1, 2, \dots, n-1$ as indices to place a queen on each row. We can then reject a candidate solution only by looking at the diagonal attacking positions. Listing 55.3 shows an implementation of this idea that uses the `std::next_permutation` function to generate the permutations. This approach is in practice significantly faster than the ones presented above.

```

1 auto nqueen_one_per_row_and_column(const unsigned n)
2 {
3     std::vector<ChessBoardSnapshot> ans;
4     std::vector<int> positions(n);
5     std::iota(std::begin(positions), std::end(positions), 0);
6
7     Solution sol_candidate(n);

```

```

8   do
9   {
10      for (int i = 0; i < std::ssize(positions); i++)
11          sol_candidate[i] = CellCoordinate(i, positions[i]);
12
13      if (is_valid_solution(sol_candidate))
14          ans.push_back(make_chessboard_snapshot(sol_candidate));
15
16  } while (std::next_permutation(std::begin(positions), std::end(positions)));
17
18  return ans;
19 }

```

Listing 55.3: Solution to the n-queens problem where we do not place more than one queen on the same row and column.

55.2.4 Brute-force revisited

The purely brute-force approach shown in Section 55.2.1 can be improved quite dramatically by using an early pruning technique that does not wait to reject a given arrangement until all the queens are placed, but does so as soon as a conflict is found, even on a partial candidate solution. Surprisingly this can be achieved with minimal changes to the Listing 55.1 by adding a call to `is_valid_solution(sol_candidate)` every time an element is added to the candidate solution list.

Listing 55.4 shows how this modified version of the brute-force approach above can be implemented. Note that the function `can_add_queen` is basically a copy and paste of the function `is_valid_solution` with the (crucial) difference that it only checks whether is it legal to add a queen at the position specified by its second argument when you already have a number of queens placed in the board whose positions are defined in its first parameter. This solution has performance that is comparable to the fastest one shown in this chapter, Listing 55.3.

```

1  bool can_add_queen(const Solution& solution,
2                    const CellCoordinate& candidate_position)
3  {
4      const auto sgn = [](auto val) {
5          using T = decltype(val);
6          return (T(0) < val) - (val < T(0));
7      };
8
9      for (int i = 0; i < std::ssize(solution); i++)
10     {
11         const auto rows_difference = solution[i].first - candidate_position.first;
12         const auto cols_difference = solution[i].second - candidate_position.second;
13
14         const bool same_row    = rows_difference == 0;
15         const bool same_col    = cols_difference == 0;
16         const bool same_diag1 = rows_difference == cols_difference;
17
18         const bool same_diag2 =
19             std::abs(rows_difference) == std::abs(cols_difference)
20             && sgn(rows_difference) != sgn(cols_difference);
21         if (same_row || same_col || same_diag1 || same_diag2)
22             return false;
23     }
24     return true;
25 }

```



```

26
27 void nqueen_bruteforce_helper_revised(
28     const unsigned n,
29     const std::vector<CellCoordinate>& locations,
30     const int location_idx,
31     Solution& solution_candidate,
32     std::vector<ChessBoardSnapshot>& ans)
33 {
34     if (solution_candidate.size() >= n)
35     {
36         // all intermediate steps have been checked
37         ans.push_back(make_chessboard_snapshot(solution_candidate));
38
39         return;
40     }
41
42     for (int i = location_idx; i < std::ssize(locations); i++)
43     {
44         if (!can_add_queen(solution_candidate, locations[i]))
45             continue;
46         // adding a queen in locations[i] does not conflict with the rest of the
47         // queen specified in solution_candidate
48         solution_candidate.push_back(locations[i]);
49         nqueen_bruteforce_helper_revised(
50             n, locations, i + 1, solution_candidate, ans);
51         solution_candidate.pop_back();
52     }
53 }
54
55 auto nqueen_bruteforce_revised(const unsigned n)
56 {
57     std::vector<CellCoordinate> locations;
58     locations.reserve(n * n);
59     for (unsigned i = 0; i < n; i++)
60         for (unsigned j = 0; j < n; j++)
61             locations.push_back(CellCoordinate(i, j));
62
63     Solution sol_candidate;
64     std::vector<ChessBoardSnapshot> ans;
65     nqueen_bruteforce_helper_revised(n, locations, 0, sol_candidate, ans);
66     return ans;
67 }

```

Listing 55.4: Revised brute-force solution to the n-queens problem optimized by using an early pruning technique.

56. Gas Station

Introduction

Imagine there we drive along a circular route with a number of gas stations along it. Our goal is to drive across the entire route but before departure we would like to make sure we are not going to get stranded because we run out of gas.

Our car starts with an empty tank and can make 1km per 1l of gas and each gas station has a maximum amount of gas it can deliver. The problem with this setting is that we can end up in a situation where after having refueled we still do not have enough gas to reach the next gas station.

In the problem discussed in this chapter we will discuss how to make sure we always start the journey from a place along the route from where it is not possible to get stranded.

56.1 Problem statement

Problem 79 You are given two arrays of integers G and C both of size n where:

- $G[i]$ represent the amount of gas station i can deliver;
- and $C[i]$ is the amount of liters of gas necessary to reach the next gas station $i + 1$;

You have a car with a tank of unlimited size and, you always start your journey from one of the gas stations and on an empty tank.

You can only move forward from gas station i to the next at index $i + 1$. When you are at gas station $n - 1$ you can travel to gas station 0. A succesfull journey means starting at some gas station k , making n stops for gas and ending up at gas station k .

Write a function that returns the smallest index $0 \leq k < n$ of a gas station from where you can start your journey and complete a loop around the circular route without getting stranded.

■ Example 56.1

Given $G = \{1, 2\}$ and $C = \{2, 1\}$ the function returns 1 (see Figure 56.1).

If we start from index 0, we can fill in the tank with $G[0] = 1$ liters of gas. Now our tank has 1 liter of gas, but we need $C[0] = 2$ gas to travel to station 1.

If instead, we start from the station at index 1, we can fill in $A[1] = 2$ liters of gas and end up with a tank with 2 liters of gas. We need only $B[1] = 1$ liters of gas to get to the next station 0. We make the journey and travel to station 0 and we still have 1 unit of gas left. At this point, we fill the tank again with $A[0] = 1$ liters of additional gas, for a total of 2 liters in the tank. It costs us $B[0] = 2$ liters to get to station 1, which we do and we can then complete the loop succesfully. ■

■ Example 56.2

Given $G = \{7, 1, 0, 11, 4\}$ and $C = \{5, 9, 1, 2, 5\}$ the function returns 1 (see Figure 56.2).

If we start our journey from the station 0 we are stranded before we reach the

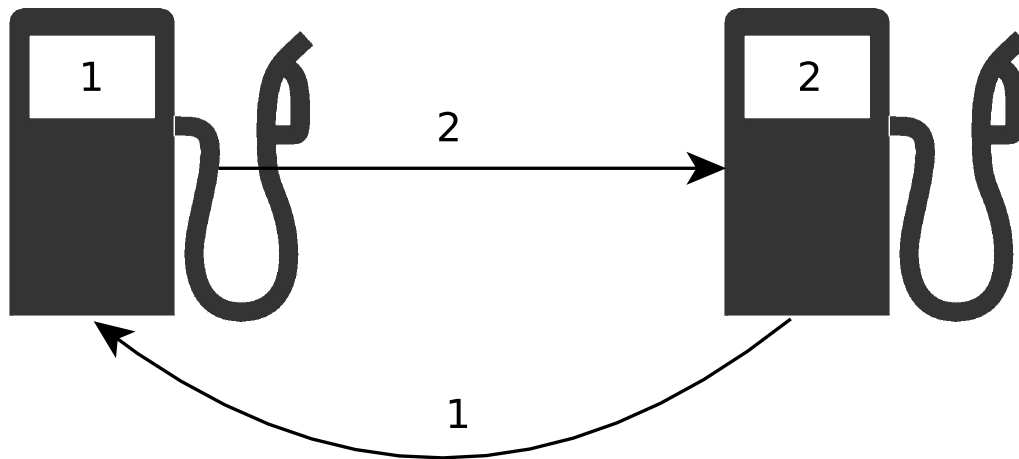


Figure 56.1: Visual representation the problem instance of Example 56.1.

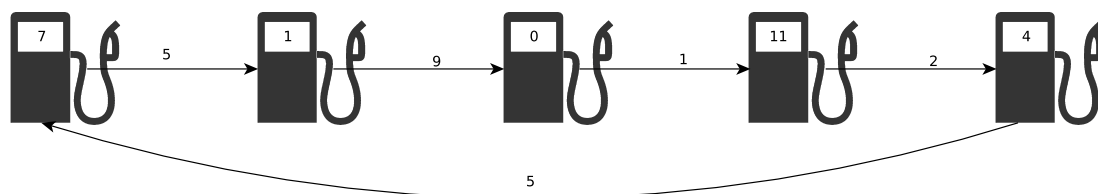


Figure 56.2: Visual representation the problem instance of Example 56.2.

station 2. If we start from station 1 we cannot even make it to the next station as we can only fill the tank with 1 liters but we need 9 to reach station 2. From station 3 it is clear we cannot make it because we cannot even refuel a drop of gas. Station 3 is the good one because we can fill the tank with 11 liters, move to station 4 only using 2. At this point we can refuel 4 liters and we set off with 13 liters in the tank. Once we reach station 0 we used 5 but we can refill with 7 and we are left with 15. On the next leg we use 5 liters of gas and refuel for 1, leaving us with 11 liters. On the next leg we use 11 units but we do not get to refuel at all. At this point we are left with only 2 liters of gas, but fortunately for the last leg of the trip we only need 1 liter. We therefore circle back to the station 3 with still 1 liters left in the tank.

56.2 Clarification Questions

Q.1. What shall the function return when it is not possible to complete a loop starting from any station?

The function returns -1 in this case.

Q.2. What is the maximum possible number of gas stations?

n can be up to 1000000.

Q.3. Can we have negative values for gas refuel and cost?

No, you can assume G and C always contain non-negative integers.

Q.4. Can G and C be empty?

Yes, n can be zero.

Discussion

We can start our discussion by noticing that there are certain cases where it is impossible to perform a full loop, and specifically this is the case when the overall costs are higher than the total sum of available gas along the route. Clearly this signals the fact that we need more gas than it is available to complete the route. In this case we can return -1 .

But are we guaranteed to be able to complete a loop if the available gas is more than the overall cost? The answer is a sound yes.

Moreover when there is only a single station in the route, we can immediately return 0, regardless of the value we have in C , as in order to complete the loop we do not need to move our car at all.

56.2.1 Brute-force

A brute-force solution just simulates the car driving. We can perform this simulation by trying each time a different starting gas station.

The simulation takes care of keeping track of the amount of gas in the tank as we move from station to station and it is implemented in function `can_complete_loop_from_station` in Listing 56.1. This function is called from within a loop feeding it each time with a new starting gas station until either we tried them all or we found one from which is possible to complete a loop. The full implementation is shown in Listing 56.1.

```
1  bool can_complete_loop_from_station(const std::vector<int>& G,
2                                     const std::vector<int>& C,
3                                     const int station)
4  {
5      const auto size = G.size();
6      int curr_station = station;
7      int next_station = (curr_station + 1) % size;
8      int tank         = G[curr_station];
9      while (next_station != station)
10     {
11         if (tank < C[curr_station])
12         {
13             return false;
14         }
15         tank -= C[curr_station];
16         curr_station = next_station;
17         next_station = (curr_station + 1) % size;
18         tank += G[curr_station];
19     }
20     return tank >= C[curr_station]; // make sure you can make the last leg
21 }
22
23 int solve_gas_station_bruteforce(const std::vector<int>& G,
24                                  const std::vector<int>& C)
25 {
26     const auto size = G.size();
27     if (size == 1)
28         return 0;
29
30     for (int i = 0; i < size; i++)
```

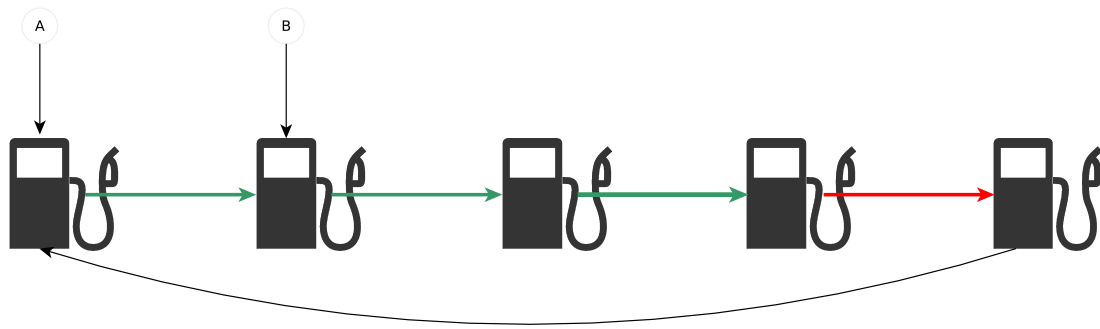


Figure 56.3: Behavior of the brute-force solution.

```

31 {
32     if (can_complete_loop_from_station(G, C, i))
33         return i;
34 }
35 return -1;
36 }

```

Listing 56.1: Brute-force solution.

This solution has a time complexity of $O(n^2)$. The space complexity is constant.

56.2.2 Linear time

Let's have a look at the brute-solution and see if we can find any inefficiencies and fix them. At a closer look it appears that, if we simulate the car driving starting from a station k and we are able to drive it up until station $k+x$ then we stop and we start over simulation choosing as a starting point the station at index $k+1$. Is it really necessary? What if we could somehow start from $k+x$ and still be guaranteed we are not missing any potential good starting gas station?

Let's imagine for a second that we start from station 0 and we are able to drive it up until station 3 but we run out of gas when trying to drive to station 4 as shown in Figure ???. When the brute-force simulation figures that is impossible to continue, it starts the simulation process from the second station (marked as B). But this simulation is useless and it is destined to also fail. Why is that? The reason is that we already know that with the first three legs of the trip we are already in a surplus or at the very least breaking even with the fuel amount. This is because if we were at some kind of deficit we would have not been able to make the three legs journey in the first place.

Therefore starting from station 1 is not useful as in the previous simulation we arrived at station 1 (starting from station 0) with an amount of fuel greater or equal than zero. The same reasoning can be applied to the starting point 2. Because when we simulated the car driving starting from station 0 we were able to reach station 3, this means we reached station 2 from station 1 with a surplus of fuel and therefore if we did not make it back then to station 4, there is no way we can make it now that we start with 0 liters of fuel from station 2.

What does this mean in terms of our algorithm? We know that when performing the simulation from station k , if we run out of fuel at station $k+x$ we can safely start our simulation at index $k+x+1$, saving a lot of work in the process. Moreover, when we finally are able to reach station 0 again, we can immediately return the last starting station. For instance let's analyze the example shown in Figure 56.4. When starting from station 0 we

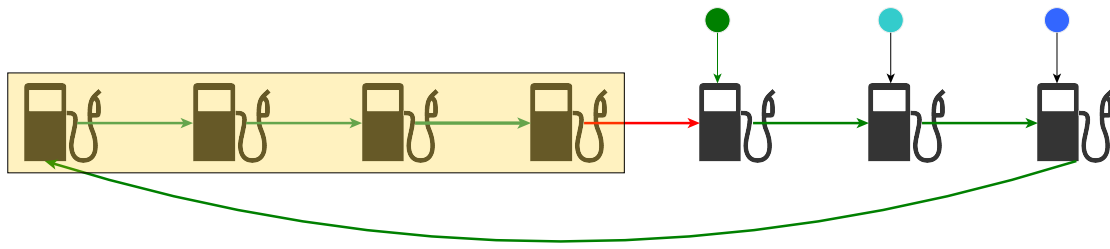


Figure 56.4: Behavior of the brute-force solution.

are able to make it up to station 3. We then use station 4 as starting point and we make it back to station 0. We can at this point conclude that station 4 is the answer because we know that there is an answer for this instance (the sum of the gas is greater or equal than the sum of the costs) and all the previous stations are not valid starting points.

One can however argue that station 4 (marked with a green circle) is not a valid starting point because we did not actually check we can make the journey from station 0 to station 4 (stations highlighted in light yellow) once we circled back and that maybe station 5 (in cyan) or 6 (in blue) are the right station to start from. This reasoning is not quite right as again, if we make it to station 5 or 6 starting from station 4, it means we reach those station with some sort of fuel surplus and therefore starting from station 5 or 6 would not be more beneficial than starting from station 4.

All these insights above are implemented into a solution in Listing 56.1.

```

1  int solve_gas_station_linear_time(const std::vector<int>& G,
2                                  const std::vector<int>& C)
3  {
4      const int n      = G.size();
5      const auto sum_gas = std::reduce(std::begin(G), std::end(G), 0);
6      const auto sum_cost = std::reduce(std::begin(C), std::end(C), 0);
7
8      // if there is not enough gas along the way and we need to make at least one
9      // leg of the trip
10     if ((sum_gas < sum_cost) && (n > 1))
11         return -1;
12
13     if (n <= 0)
14         return -1;
15
16     int ans = 0;
17     int tank = 0;
18     for (int i = 0; i < n; i++)
19     {
20         tank += G[i] - C[i];
21         if (tank < 0)
22         {
23             // i+1 is the new starting point. We can ignore all stations from and to
24             // (i+1) as they are for sure not good starting point
25             ans = (i + 1) % n;
26             tank = 0;
27         }
28     }
29     return ans;
30 }

```

Listing 56.2: Linear time constant space solution.

56.3 Common Variation - Fuel tank with limited capacity

Problem 80 Solve the problem described in Exercise 62.1 with the additional constraint that the fuel tank of the car has a maximum capacity k that is given as an additional parameter to the function you need to write. ■

57. Merge Intervals

Introduction

Intervals are common in programming and they pop up in numerous applications as they are very versatile and are used to represent many things, from segments in geometry applications to time spans for timetable building (e.g. for keeping track of the availability of meeting rooms for instance) or resource scheduling.

Intervals are also quite popular in interview questions; In this chapter will go through one that is commonly asked (still nowadays) at Google where we are given a list of time intervals and we need to produce a new list where none of the elements overlap with one another.

In Section 83, a variation of this problem where we are given a list of time intervals where we are guaranteed none of them overlaps with one another to begin with and our job is to insert a new interval in the list in such a way that the non-overlapping property is maintained.

57.1 Problem statement

Problem 81 Write a function that, given a list of intervals $I = \{(s_0, e_0), (s_1, e_1), \dots, (s_{n-1}, e_{n-1})\}$ represented as a pair of integers, returns a new list I' which is a copy of I except that overlapping intervals are merged together. The resulting list should only contain non-overlapping intervals.

■ **Example 57.1**

Given $I = \{(3, 7), (1, 5), (6, 8), (4, 6)\}$ returns $I' = \{(1, 8)\}$. ■

■ **Example 57.2**

Given $I = \{(1, 5), (6, 7), (4, 4), (9, 12)\}$ returns $I' = \{(1, 5), (6, 7), (9, 12)\}$. ■

57.2 Clarification Questions

Q.1. (If not clear from the examples) Is it guaranteed for the intervals to be sorted? (either by the starting or ending time of the interval)

No, the input is not sorted.

Q.2. Can the input list be modified?

No, I is read-only.

Q.3. Can we always assume that given an interval (x, y) in I , $x \leq y$ always holds?

Yes individual intervals have the first component always smaller or equal than the second.

57.3 Discussion

Let's have a closer look at why Example 57.1 results in $I' = \{(1,8)\}$. Intervals (3,7) and (1,5) overlap and they can be merged into (1,7) which can in turn, be merged with both (6,8) and (4,6). Because $\{(4,6)\}$ lies completely inside (1,7), merging them results still in (1,7). Finally, $\{(1,7)\}$ also overlaps with (6,8) and blending the two of them together yields $\{(1,8)\}$.

We can determine if two intervals $a = (a_s, a_e)$ and $b = (b_s, b_e)$ overlap if any of the following is true:

- a is fully contained in b : this happens when both a_s and a_e are within b i.e. $a_s \geq b_s$ and $a_e \leq b_e$ e.g. (10,19) is fully contained in (9,20);
- b is fully contained in a ;
- a_s is partially contained in b i.e. $a_s \geq b_s$ and $a_s \leq b_e$.
- b_s is partially contained in a i.e. $b_s \geq a_s$ and $b_s \leq a_e$.

If none of the above condition is met then a and b are completely disjoint.

57.3.1 Brute-Force

We can use these facts to build a solution that examines one interval at the time, starting from the first one and greedily tries to merge it with as many other intervals as possible. The idea is that once we picked an interval x we will merge it with all the other overlapping intervals. This results in a interval m which is the combination of x and zero or more other intervals in the list. We can at this point add m to the output list. We can also mark the intervals we merged x with, so that they will be ignored for the remainder of the process: after-all they are already accounted for in m . If we repeat this process for each and every unmarked interval of I we will eventually have an output list I that contains only un-mergable and non-overlapping intervals. This idea is implemented in Listing 57.1.

```
1
2
3 std::vector<Interval> merge_list_intervals_entire_list_bruteforce(
4     std::vector<Interval>& intervals)
5 {
6     if (intervals.size() <= 0)
7         return intervals;
8
9     std::vector<Interval> ans;
10    std::vector<bool> excluded(intervals.size(), false);
11
12    for (size_t i = 0; i < intervals.size(); i++)
13    {
14        if (excluded[i])
15            continue;
16        ans.push_back(intervals[i]);
17        excluded[i] = true;
18        for (size_t j = i + 1; j < intervals.size(); j++)
19        {
20            if (const auto [ok, merged] = merge(ans.back(), intervals[j]);
21                !excluded[j] && ok)
22            {
23                ans.back() = merged;
24                excluded[j] = true;
25            }
26        }
27    }
28
29    return ans;
```

Listing 57.1: Quadratic time solution.

The function `mergeNonSorted` contains the logic for merging two intervals and it is used in the main function `merge_list_intervals_entire_list_bruteforce` that loops through the elements of the input list one at the time and carefully uses an array of boolean flags `excluded` to mark intervals that have been already merged. If an interval is not yet merged into the output list `ans` then we place it at the back of `ans` and then we try to merge it with the rest of the unmarked intervals (in the innermost loop). Whenever it overlaps with some other interval the resulting merged interval is substituted at the back of `ans` and the interval it was merged with is marked as to be excluded from further examination in the future iterations. Therefore each and every interval is either used as a starting seed and is tested for overlap and potentially merged with the remaining of the not yet excluded intervals or, it is skipped altogether because was already merged in a previous iteration. Notice that the inner loop starts at $j = i + 1$ as intervals before position i have been already merged (and clearly we do not need to test for overlapping when $i = j$).

Using this strategy, we are sure that the output list will never contain any interval that is not covered by one or more intervals in the input list. Similarly, if a value is covered by an interval in I then, we are also guaranteed it will be contained in the output. In other words, if you imagine that a list of intervals represents colored segments of a line then I and I' would produce the very same exact coloring of the line.

The complexity of this function is quadratic in time; when we have an input list containing only non-overlapping intervals, none of them (except the one we are trying to merge the others with) will ever be excluded and that means a full run of the innerloop will occur.

The space complexity is linear as we use an array of the same size of I for the output list as well as for the array of flags `excluded`.

57.3.2 $n\log(n)$ sorting solution

The quadratic solution presented in Section 57.3.1 is correct but it does quite a lot of unnecessary work. When we start examining an not yet excluded interval at index i we are forced to take a look at all the other subsequent intervals in the list because we have no way of knowing whether there will be one with which interval i overlaps with. However, **this is not true if the intervals are sorted**. Let's assume I is sorted by the *start* field and that ties between intervals (intervals starting at the same value) are resolved using the *end* field. When examining the interval I_i we can simply look at the next interval I_{i+1} and if it does not overlap with I_i then we are sure that none of the elements ahead of it will! The reason is that in order for interval I_i not to overlap with I_{i+1} it has to be that $I_i.end < I_{i+1}.start$. Because intervals are sorted by their *start* fields, any subsequent intervals will have an even higher *start* value. This allows us to save tons of work as we can merge the entire list in linear time once it is sorted.

Listing 57.5 implements this idea.

```

1  std::vector<Interval> merge_list_intervals_entire_list_lin_time(
2      std::vector<Interval>& intervals)
3  {
4      if (intervals.size() <= 0)
5          return intervals;
6
7      std::sort(std::begin(intervals),
8                std::end(intervals),
9                [](const auto& a, const auto& b) {
```

```

10         return (a.start < b.start)
11             || ((a.start == b.start) && (a.end < b.end));
12     });
13
14     std::vector<Interval> ans;
15     ans.push_back(intervals.front());
16     for (size_t i = 1; i < intervals.size(); i++)
17     {
18         if (auto [ok, merged] = merge(ans.back(), intervals[i]); ok)
19             ans.back() = merged;
20         else
21             ans.push_back(intervals[i]);
22     }
23     return ans;
24 }

```

Listing 57.2: $n\log(n)$ time solution using sorting.

Not surprisingly, the first thing this code does is sorting the input list. We then start by pushing the first interval into the output list `ans`. We keep merging subsequent elements that overlap with `ans.back` until we find one that does not. At this point, we are sure that whatever interval is in `ans.back` will definitely not overlap with any other interval in the list. Therefore we can safely push this new interval into `ans` and repeat the process until all intervals have been analyzed.

Notice that here we are using the same `merge` function used in Listing 57.1; However, because we know elements are sorted we could simplify the condition `if ((a.start >= b.start && a.start <= b.end) || (b.start >= a.start && b.start <= a.end))` as we know that `a.start <= b.start` is always true as shown in Listing 57.3.

```

8     auto merge(const Interval& a, const Interval& b)
9     {
10         Interval ans(a);
11         bool ok = false;
12         if (a.start >= b.start && a.start < b.end)
13         {
14             ok = true;
15             ans.start = std::min(a.start, b.start);
16             ans.end = std::max(a.end, b.end);
17         }
18         return std::make_tuple(ok, ans);
19     }

```

Listing 57.3: Function to merge two sorted intervals.

This approach has a time complexity of $n\log(n)$ (due to the sorting) and the space complexity is $O(n)$ (however, we really only use linear space to store the output list and if we do not account for it the complexity is constant).

57.4 Common Variation - Add a new interval

57.4.1 Problem statement

Problem 82 Given a sorted list of disjoint (non-overlapping) intervals I and an interval w , insert w into I so that the resulting list still contains only disjoint intervals. You may assume that the intervals are sorted according to their start times.

■ Example 57.3

Given $I = \{(1, 3), (6, 9)\}$ and $w = (2, 5)$ the function returns $I' = \{(1, 5), (6, 9)\}$ (see Figure 57.5). ■

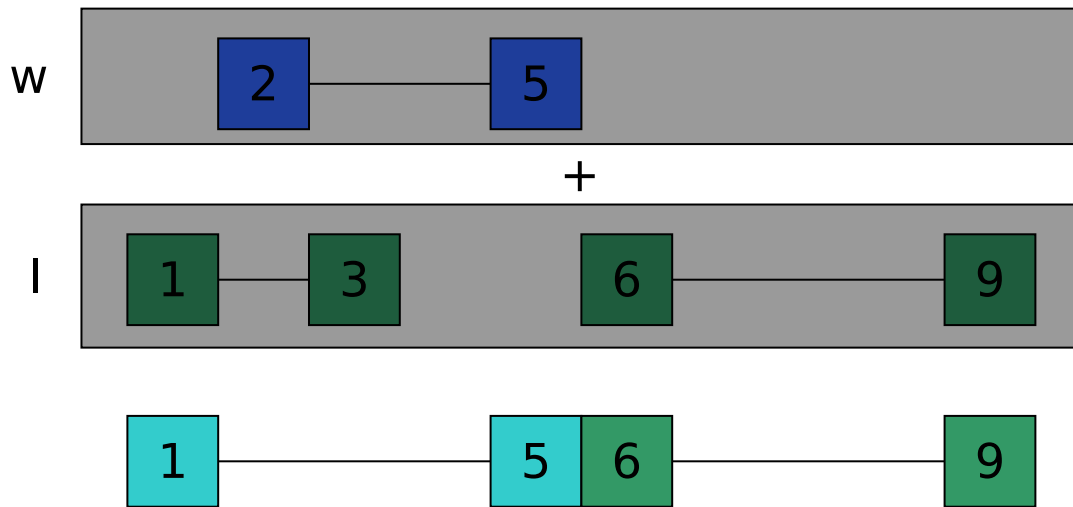


Figure 57.1: Visual representation the problem instance of Example 57.5. The ■ green (1,3) and ■ blue interval (2,5) are merged together into the ■ cyan interval (1,5) (in the bottom row). The interval (6,9) in I does not overlap with either (2,5) not with newly formed (1,5) and it is therefore simply copied over to the answer list.

■ Example 57.4

Given $I = \{(1,2), (3,5), (6,7), (8,10), (12,16)\}$ and $w = (4,9)$ the function returns $I' = \{(1,2), (3,10), (12,16)\}$ ■

57.4.2 Discussion

This problem is really not that different from the one described in Section 81 to the point that you can use almost the same approach to tackle this one. In particular, one can think of simply adding w to I and then proceed to use the algorithms described above to solve the problem. This would grant us a $n \log(n)$ solution for this variation with very little effort. However, there is really no need to do the extra $\log(n)$ work to have the entire I plus w sorted, especially considering that I comes already in a sorted state and we can take advantage of that.

Let's start by imagining that an oracle would tell us the index k (if exists) of the first interval in I having its end field smaller than w 's start field. Clearly every element before k can be simply copied over into the output list as we are sure they do not overlap with w .

What we can do at this point is to insert w in the output list and basically use the same algorithm discussed above for solving the main version of this problem. We can do this because we know for a fact that I_{k+1} does not completely fall behind w , otherwise we would have had inserted it in the previous step. Therefore we are left with two options about the relationship between $I_{k+1}.start$ and w :

1. $I_{k+1}.start$ and w overlap;
2. w falls completely behind $I_{k+1}.start$.

From these facts we can deduce that we are safe at inserting w after having copied element k .

At this point all we are left to do is to try and merge $\{w, I_{k+1}, I_{k+2}, I_{k+2} \dots\}$, which we

can do using the same approach discussed above (minus the sorting step, of course).

An implementation of this idea is shown in Listing 57.4

```
1 static bool overlap(const Interval& i1, const Interval& i2)
2 {
3     return (i1.start >= i2.start && i1.start <= i2.end)
4         || (i2.start >= i1.start && i2.start <= i1.end);
5 }
6
7 std::vector<Interval> merge_intervals_lineartime(
8     const std::vector<Interval>& intervals, Interval newInterval)
9 {
10     std::vector<Interval> ans;
11     if (intervals.empty())
12     {
13         ans.push_back(newInterval);
14         return ans;
15     }
16
17     size_t k = 0;
18     bool inserted = false;
19
20     for (; k < intervals.size(); )
21     {
22         if (intervals[k].end < newInterval.start)
23         {
24             ans.push_back(intervals[k]);
25             k++;
26         }
27         else
28         {
29             inserted = true;
30             ans.push_back(newInterval);
31             break;
32         }
33     }
34     if (!inserted)
35     {
36         ans.push_back(newInterval);
37         return ans;
38     }
39
40     while (k < intervals.size())
41     {
42         if (overlap(intervals[k], ans.back()))
43         {
44             ans.back().start = std::min(ans.back().start, intervals[k].start);
45             ans.back().end = std::max(ans.back().end, intervals[k].end);
46         }
47         else
48         {
49             ans.push_back(intervals[k]);
50         }
51         k++;
52     }
53
54     return ans;
55 }
```

Listing 57.4: Linear time solution.

The code works in two distinct stages represented by the `for` and `while` loops. In the

first loop statement we take care of finding the location k at which we can insert w . Notice that k might also be the end of the list and this is why we have the `if(!inserted)` check. If every element of I ends before $w.start$ then we are sure w should be placed at the very end of the output list and there is no more work to be done.

The second loop, like in Listing 57.5 uses `ans.back` as a working variable to merge w with the rest of the overlapping intervals in I . When we reach a point j where there is no more element overlapping with `ans.back` then the `else` part of the code will always be executed (as we started off with non-overlapping intervals in I) and the remainder of the list gets copied in the output list, completing the exercise.

Both time and space complexities are linear.

57.5 Common Variation - How many meeting rooms are needed?

57.5.1 Problem statement

Problem 83 Given an array of time intervals where each interval $(x,y) : x < y$ represent a meeting starting at time x and ending at time y (excluded), return the minimum number of conference rooms required to accomodate all meetings.

■ Example 57.5

Given $I = \{(9,10), (4,9), (5,17)\}$ the function returns 2. We can schedule $(4,9)$ and $(5,17)$ in two different meeting rooms while $(9,10)$ can be scheduled in the same room of $(4,9)$ after it ends. ■

57.6 Brute-force

Let's start by noticing that if you have k intervals that are **all overlapping with each other**, then you need to have k different meeting rooms to accommodate all k meetings. We can build on this simple observation and calculate the number of concurrent meetings for each and every time unit encompassed by I : from the earliest start to the latest end of a meeting in I (or in other words, from the smallest to the largest among all start and end fields of all intervals). For instance w.r.t. the Example 57.5 we know the smallest time among all intervals is 4 while the largest is 17. Therefore there are $t = 4 - 17$ time units for which we need to calculate the number of concurrent meetings. Our answer will be the largest among these values.

We can keep an array of size t , where index 0 maps to time 4, index 1 to time 5 and so on up to the last index of the array (12) mapping to time 17, to store the information about the number of meetings happening at that time. Such an array can be filled by looping through each and every interval $i = (s_i, e_i)$ in I and incrementing only the cells corresponding to times from s_i to e_i (the time spanned by the interval i).

Figure 57.2 shows how such an array would look like after being filled with the procedure discussed above and using the input of Example 57.5. We can see how the maximum value there is 2 signifying that we need at most 2 meetings rooms to accomodate all meetings.

Listing 57.5 implements this idea. The code is pretty straightforward and we can notice how we have a double loop, one looping through the intervals and the second through the time span of a single interval. We can also notice how we use the variable `shift==min_hour` to map indices of the array `meeting_at_time` to the actual times in I .

```
1 unsigned calculate_number_meeting_rooms1(const std::vector<Interval>& meetings)
2 {
```

time	4	5	6	7	8	9	10	11	12	13	14	15	16	17
meetings	1	2	2	2	2	2	2	1	1	1	1	1	1	1

Figure 57.2: Number of concurrent meetings throughout the entire time spanned by all intervals in Exampled 57.5.

```

3  if (meetings.empty())
4      return 0;
5
6  auto [min_hour, max_hour] = std::tuple(0, 0);
7  for (const auto [start, end] : meetings)
8  {
9      min_hour = std::min(min_hour, start);
10     max_hour = std::max(max_hour, end);
11 }
12
13 std::vector<uint> meeting_at_time(max_hour - min_hour);
14 unsigned ans      = 0;
15 const auto shift = -min_hour; // we want min_hour to be mapped to index 0
16 for (auto [start, end] : meetings)
17 {
18     while (start < end)
19     { // end hour of the meeting is excluded
20         const auto shifted_time = start + shift;
21         meeting_at_time[shifted_time]++;
22         ans = std::max(ans, meeting_at_time[shifted_time]);
23         start++;
24     }
25 }
26 return ans;
27 }

```

Listing 57.5: $O(k^2 + n)$ time and $O(k)$ space solution where k is the time span between the smallest starting and largest ending time of a meeting.

The complexity of this approach is $O(k^2 + n)$ where k is the difference between the smallest and largest time among all intervals in I and n is the number of such intervals. The space complexity is $O(k)$. If k is constant, this is actually not too bad of a solution, but in general, for this problem both k and n can be considered not to have a particularly favorable bound and we need therefore look for a more efficient solution.

57.7 $n \log(n)$ - Intervals endpoints

We can enrich our set of observations about this problem with the fact that, if two meetings overlap, then they must overlap either at the start or at the end of a room's booking interval. This allows us to basically avoid to explicitly mark every time unit in an interval and focus only on its endpoints. For example in the Example 57.2 we see that interval (5,17) and (4,9) overlaps and also that they start doing so from time 4 and also that they stop overlapping at time 9. Why is this useful for us? Because if we can somehow avoid looking at the entire time spanned by all intervals in I and just look at the endpoints then the running time of such algorithm would be time complexity $O(n)$.

Let's imagine for a second we have all the *start* and *end* fields stored and then sorted in

increasing order into two independent arrays, say `startPoints` and `endPoints`, respectively. The idea is to keep track of two separate indices `idxS` and `idxE` pointing to `startPoints` and `endPoints`, respectively, both initialized to zero. We also keep track of the number of concurrent meetings in a variable `concurrentMeetings` also initialized to zero. We can then compare `startPoints[idxS]` with `endPoints[idxE]` and:

1. if `startPoints[idxS] < endPoints[idxE]` then we can conclude that we have found a start point of a meeting that is happening before the next (in time) end of a meeting. Therefore we need another room: `concurrentMeetings++`
2. if, on the other hand, `startPoints[idxS] >= endPoints[idxE]` then we have found that the end of a meeting happens to be before the start pointed by `idxS` and we can decrement `concurrentMeetings`.

This algorithm works because `startPoints` and `endPoints` are sorted for each element of the former there is one that is larger in the latter. This means that `concurrentMeetings` will never be smaller than one in practice after the first iteration (which will always hit the first case above i.e. `startPoints[0] < endPoints[0]`)

This idea is implemented in Listing 57.6.

```
1
2 unsigned calculate_number_meeting_rooms3_touchpoints(
3     const std::vector<Interval>& meetings)
4 {
5     if (meetings.empty())
6         return 0;
7
8     std::vector<int> startPoints;
9     startPoints.reserve(meetings.size());
10
11     std::vector<int> endPoints;
12     endPoints.reserve(meetings.size());
13
14     for (const auto& [s, e] : meetings)
15     {
16         startPoints.push_back(s);
17         endPoints.push_back(e);
18     }
19
20     std::sort(std::begin(startPoints), std::end(startPoints));
21     std::sort(std::begin(endPoints), std::end(endPoints));
22
23     unsigned ans = 1, concurrentMeetings = 0;
24     size_t idxS = 0; // index on startPoints array
25     size_t idxE = 0; // index on endPoints array
26     while (idxS < startPoints.size())
27     {
28         if (startPoints[idxS] < endPoints[idxE])
29         {
30             concurrentMeetings++;
31             ans = std::max(ans, concurrentMeetings);
32             idxS++;
33         }
34         else
35         {
36             assert(concurrentMeetings >= 1);
37             concurrentMeetings--;
38             idxE++;
39         }
40     }
41     return ans;
```


Listing 57.6: $O(n \log(n))$ time and $O(1)$ space solution.

The code works by first making sure `startPoints` and `endPoints` are filled and sorted properly. The next step is to loop until we have took a look at each and every element in `startPoints` and increasing the value of the variable `concurrentMeetings` depending on the value of such comparison.

The time and space complexities of this solution are $O(n)$.

58. Least Recently Used Cache

Introduction

Caches are piece of hardware or software systems responsible to store data that allow the system to respond to future requests faster. Usually such data is the result of earlier computation of it could data that was copied over from somewhere else (following the principle of temporal and spatial locality which states that applications are more likely to access data that has been accessed recently and/or that sit close in memory). Caches are essential piece of nowadays computer systems and we find them at every level: from CPUs and HDDs (see Figure 58.1a) that have dedicated expensive blocks of memory that for temporary storage of data that is likely to be used again, to browsers or webserver (see Figure 58.1b) that use caches to try and lower the latency of your browsing.

Clearly caches have finite size and eventually they get full and therefore we run into the issue of deciding what to delete from it in order to make some space available for the new data. There are many policies that can be employed here like for example:

- LRU: discards the least recently used items first;
- FIFO: evicts the oldest data first (regardless of whether it has been accessed recently);
- RR: (random replacement) that, as the name suggests, removes one or more random cache entries.

The problem we will solve in this chapter is about implementing a LRU cache so that all of its supported operation are carried out with the best time efficiency possible. As we will see, solving this problem by making all operations $\log(n)$ is actually pretty easy, but

58.1 Problem statement

Problem 84 Implement the two two public methods of the `LRUCache` interface below. The class should behave like a LRU cache of capacity `n` that is given as a parameter.

- `std::optional<Value> get(Key k)` returns the value associated with the key `k`. If `k` is not in the cache, the function returns `std::nullopt`.
- `void put(Key k, Value v)` adds or update (depending on whether `k` is already present in the cache or not) the pair (k, v) . If the cache is full i.e. has size `n` the function evicts the least recently used items before inserting (k, v) .

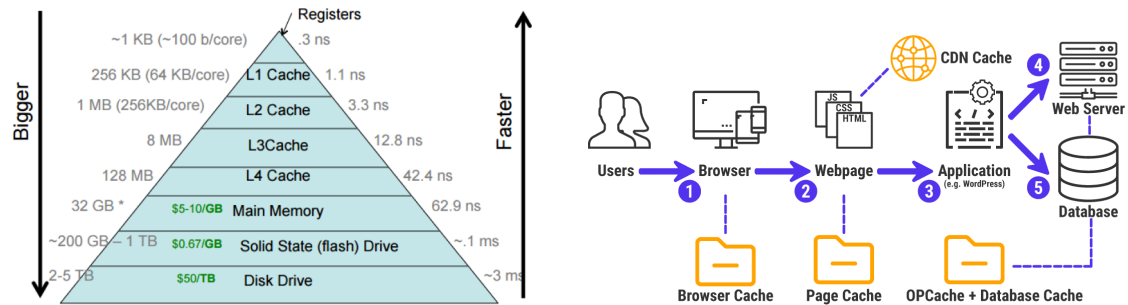
■ Example 58.1

For instance, given a cache `c` of size 2 (with char keys and int values), Table 58.1 shows the content after performing a number of operations on it. ■

58.2 Clarification Questions

Q.1. Is one of the two operations going to be performed more than the other?

No no assumptions like this can be made.



- (a) Caches (notice that there are multiple level of it) are at the top of the memory hierarchy of modern computers. See Table 64.2 for a more detailed account of the latencies figures for each and every memory level of the hierarchy. The figures on size and latency showed are for a typical desktop computer. Use `getconf -a | grep CACHE` or `lscpu` to check the actual cache size in your system.
- (b) Caches in the internet. We can see caches being used at every level. Even the database might cache recently used data in memory.

Figure 58.1

Operation	Cache	Return	Comment
<code>get('a')</code>	<code>{}</code>	<code>std::nullopt</code>	'a' $\notin C$
<code>put('a', 1)</code>	<code>{a=1}</code>	-	Put OK
<code>put('b', 2)</code>	<code>{a=1}, {b=2}</code>	-	Put OK
<code>get('a')</code>	<code>{a=1}, {b=2}</code>	1	'b' is LRU element now
<code>get('b')</code>	<code>{a=1}, {b=2}</code>	2	'a' is LRU element now
<code>put('c', 3)</code>	<code>{a=1}, {c=3}</code>	-	Put OK. 'a' evicted
<code>get('a')</code>	<code>{a=1}, {c=3}</code>	<code>std::nullopt</code>	'a' $\notin C$ anymore
<code>get('b')</code>	<code>{a=1}, {c=3}</code>	2	Get OK
<code>get('c')</code>	<code>{a=1}, {c=3}</code>	3	Get OK

Table 58.1: Example of behavior of the a LRU cache.

58.2.1 Brute-force

A very basic solution can be obtained by just storing the key-value pairs alongside a integer value that marks their last usage time into a vector. If we make such vector is always sorted, then the `get` operation can be implemented as a linear search and the `put` would consist of a potential `pop_back()` (that we only do when the cache is full and removes the oldest entry) followed by a `push_back` and a sort. This is quite inefficient as, the `get` operation would run in $O(n)$ time while the `put` is even more expensive with its $O(n\log(n))$ time complexity.

In order to improve things what we can do is to store the key-value-timestamp tuple into an `std::unordered_map` and keep a vector of pairs `std::vector<std::pair<Timestamp, Key>>` sorted by the first element of the pair. Each pair in the vector contains the information about the last time a given key has been used. If we keep this array sorted by such timestamp then we have a way of keeping track which element needs to be deleted when the cache is full. Moreover, by storing the tuple in a hashmap we are now able to serve the `get` operation in $O(1)$. The `put` is still pretty expensive as it is still $O(n\log(n))$ (actually if we use insertion sort this would most likely be $O(n)$ as we always insert in a almost sorted collection).

The reason why the `put` is expensive is because every time we add a new element we need to sort the entire array again. However if instead of a `vector` we use a `map` then we can continue to keep the pairs of timestamp and Keys sorted, only this time at only $O(\log(n))$ cost! This idea is shown in Listing 58.3

```
1  template <typename Key, typename Value>
2  class LRUCache_logn
3  {
4      using UpdateTime = int;
5
6      // key -> value, updateTime (remember last
7      // time this value was inserted/updated)
8      using ValueMap = std::unordered_map<Key, std::pair<Value, UpdateTime>>;
9      // order keys based on updateTime
10     using TimeKeyMap = std::map<UpdateTime, Key>;
11
12     ValueMap VP;
13     TimeKeyMap TK;
14     UpdateTime time = 0;
15
16 public:
17     LRUCache_logn() = default;
18     LRUCache_logn(const size_t aCapacity) : mCapacity(aCapacity)
19     {
20     }
21
22     std::optional<Value> get(const Key& key)
23     {
24         if (VP.contains(key))
25         {
26             ++time;
27             const std::pair<Value, UpdateTime>& valTime = VP[key];
28             const auto val = valTime.first;
29             const auto oldTime = valTime.second;
30             VP[key] = {val, time};
31             TK.erase(oldTime);
32             TK.insert({time, key});
33             return val;
34         }
```

```

35     return {};
36 }
37
38 void put(const Key& key, const Value& value)
39 {
40     if (VP.contains(key))
41     {
42         const auto& [oldval, oldtimestamp] = VP[key];
43         VP[key] = {value, oldtimestamp};
44         get(key); // update time
45     }
46     else
47     {
48         if (VP.size() >= mCapacity)
49         {
50             auto [timestamp, eraseblekey] = *TK.begin();
51             VP.erase(eraseblekey);
52             TK.erase(TK.begin());
53             put(key, value);
54         }
55         else
56         {
57             ++time;
58             VP[key] = {value, time};
59             TK[time] = key;
60         }
61     }
62 }
63
64 void setCapacity(const size_t aCapacity)
65 {
66     mCapacity = aCapacity;
67 }
68
69 private:
70     size_t mCapacity;
71 };

```

Listing 58.1: Solution using `map` to keep track of the least recently used entry in the cache.

The class `LRUcache_logn` has three class variables:

- `VP` of type `ValueMap` that is responsible for the storing the association between a `Key` and both its value and its timestamp;
- `TK` of type `TimeKeyMap` which basically keeps the `Keys` sorted by `Timestamp`;
- `time` that is the global clock. A variable that we only increase and it is used to mark a cache entry when an operation is performed on it.

The `get` function works by first checking whether the requested `key` has been inserted. If it was it retrieves it alongside its timestamp, then proceeds to update both `VP` and `TK` by making sure now `key` has a new timestamp associated with it (this will make sure it will be brought to the first position in `TK`).

The `put` function is slightly more complicated as we might have a case where the key is already present in the cache and therefore all we need to do is to perform an update without worrying about cache eviction. If on the other hand we are inserting a new key, then we have to further distinguish two cases:

1. **the case is not full:** this is the easiest scenario where we simply add a new entry in both `VP` and `TK`;
2. **the cache is full:** in this case we need to check the `TK` and collect the least recently key. At this point we can proceed in deleting it and safely try again to perform the

put (which this time will succeed as the cache is not full anymore).

One thing to notice is that all operations always increase the global time to make sure that we keep the keys sorted by usage time.

All the operations on `VP` run in constant time while the operations in `TK` in logarithmic time: therefore both `VP` and `VP` run in logarithmic time.

Another implementation of the same idea is shown in Listing 58.2.

```
1 class LRUCache_logn2
2 {
3 private:
4     size_t C;
5     int cnt = 0;
6
7     using Position = int;
8     using Key      = int;
9     using Value     = int;
10
11     std::unordered_map<Key, std::pair<Value, Position>> KV;
12     std::map<Position, Key> PK;
13
14 public:
15     LRUCache_logn2() = default;
16     LRUCache_logn2(const size_t aCapacity) : C(aCapacity)
17     {
18     }
19
20     Value update_value(const Key key, const Value value)
21     {
22         assert(KV.find(key) != KV.end());
23         auto kv_it = KV.find(key);
24
25         const auto [k, pair] = *kv_it;
26         const auto [v, p]    = pair;
27
28         auto pk_it = PK.find(p);
29
30         KV.erase(kv_it);
31         PK.erase(pk_it);
32         put(key, value);
33         return value;
34     }
35
36     std::optional<Value> get(const Key& key)
37     {
38         auto kv_it = KV.find(key);
39         if (kv_it == KV.end())
40             return {};
41
42         return update_value(key, kv_it->second.first);
43     }
44
45     void put(const Key& key, const Value& value)
46     {
47         if (KV.find(key) != KV.end())
48         {
49             // cout<<"key" <<key<<"already present. updating to"<<value<<endl;
50             update_value(key, value);
51             return;
52         }
53         if (KV.size() >= C)
```

```

54     {
55         auto last    = PK.begin();
56         auto [p, k] = *last;
57         PK.erase(last);
58         KV.erase(k);
59     }
60     cnt++;
61     KV.insert({key, {value, cnt}});
62     PK.insert({cnt, key});
63 }
64
65 void setCapacity(const size_t aCapacity)
66 {
67     C = aCapacity;
68 }
69 };

```

Listing 58.2: Alternative implementation of the solution discussed in Section 58.2.1 (see Listing 58.3).

58.2.2 Constant time solution

By using a `map` we were able to bring down the complexity of both operations to logarithmic time. However we can do even better if we use also keep a sorted list of the keys (sorted based on their insertion time). The benefit of using the list is in the fact we can remove an element from the middle of the list in constant time. We can at the same time keep track of the node a given key has been assigned in a `unordered_map`. This way whenever we need to remove an element from the list we just look at the last element of the list and we know immediately the key we need to remove. When it is time to update a key, we can use the new `unordered_map` we have introduced to quickly get the node of the list for that key. This way we can move the node to the front of the list, reflecting the fact this key is now the most recently used! We can operate in a similar fashion when we need to get the value of a key. All we need to do it to perform a lookup operation to retrieve the list's node associated with the key we need the value of and move such node to the front.

We use now only data structures that have constant time complexity operations and therefore we have achieved our goal of implementing both `put` and `get` operation in the most time efficient manner.

An implementation of this idea is shown in Listing ??

```

1  class LRUCache_solution1
2  {
3      using Key    = int;
4      using Value  = int;
5
6      using ValueMap      = std::unordered_map<Key, Value>;
7      using PositionList  = std::list<Key>;
8      using PositionListIterator = PositionList::const_iterator;
9      using PositionMap    = std::unordered_map<Key, PositionListIterator>;
10
11     ValueMap VP;
12     PositionMap PM;
13     PositionList PL;
14
15 public:
16     LRUCache_solution1() = default;
17     LRUCache_solution1(const size_t aCapacity) : mCapacity(aCapacity)
18     {
19

```

```

20
21 std::optional<Value> get(const Key& key)
22 {
23     if (auto it = VP.find(key); it != VP.end())
24     {
25         moveFront(key);
26         return VP[key];
27     }
28     return {};
29 }
30
31 void put(const Key& key, const Value& value)
32 {
33     if (auto it = VP.find(key); it != VP.end())
34     {
35         updateValue(key, value);
36         return;
37     }
38
39     if (VP.size() >= mCapacity)
40     {
41         eraseLeastRecentlyUsed();
42         put(key, value);
43     }
44     else
45     {
46         VP.insert({key, value});
47         PL.push_front(key);
48         PM.insert({key, PL.begin()});
49     }
50 }
51
52 void setCapacity(const size_t aCapacity)
53 {
54     mCapacity = aCapacity;
55 }
56
57 private:
58     size_t mCapacity;
59
60     void moveFront(const Key& key)
61     {
62         auto list_it = PM[key];
63         PL.erase(list_it);
64         PL.push_front(key);
65         PM[key] = PL.begin();
66     }
67
68     void updateValue(const Key& key, const Value& value)
69     {
70         VP[key] = value;
71         moveFront(key);
72     }
73
74     void eraseLeastRecentlyUsed()
75     {
76         auto key = PL.back();
77         PM.erase(key);
78         PL.pop_back();
79         VP.erase(key);
80     }

```



```
81 };
```

Listing 58.3: Alternative implementation of the solution discussed in Section 58.2.1 (see Listing ??).

The code is structurally very similar to both Listing 58.3 and 58.2. The class variable `PositionList PL` is simply a list of Keys and we use the variable `PM` to keep track of the association between a Key and an actual node of `PL`.

The helper function `moveFront` is responsible to mark a key as most recently used by, retrieving the key's list's node, move it to the front of `PL` and update the mapping between Keys and nodes to reflect this change (i.e. `PM[key]=PL.begin()`).

Erasing the least recently used element is also straightforward as all that is necessary is to retrieve the key associated with the last element of the list `PL` and erase all references in the other maps and in the list itself, of course.

59. Longest consecutive sequence

Introduction

The problem discussed in this chapter has been quite popular during virtual on-site interview at Amazon lately as it was reported by interviewees many times on reddit and other forums. The basic idea of this problem is that you are given an unsorted list of numbers and you have to tell how long is the longest sequence of consecutive numbers it contains.

This problem has a super simple solution naive solution which is also not that terrible in terms of time and space complexity but it is not optimal. Many candidates think about this solution right away and never go deeper and investigate whether a faster solution exists and therefore they damage their chances of passing the interview, or at least they make sure they are not passing with full grades. We will have a look at this intuitive sub-optimal solution in Section 59.2.1, but the core of the chapter will be in Section 59.2.2 where we investigate how to solve this problem optimally.

59.1 Problem statement

Problem 85 Write a function that, given a collection L of integers returns the length of the longest sequence of consecutive numbers in L .

■ **Example 59.1**

Given $L = \{45, 31, 46, 235, 28, 30, 29, 47\}$ the function return 4; The longest sequence of consecutive number is $\{28, 29, 30, 31\}$. ■

■ **Example 59.2**

Given $L = \{8, 2, 7, 5, 0, 1, 4, 6, 3\}$ the function return 9; L contains all numbers from 0 to 8. ■

59.2 Clarification Questions

Q.1. Does L contains only positive numbers?

No, it might contain any number that fits a standard `int`.

Q.2. Can L contain duplicates?

Yes, duplicates are allowed.

Q.3. What is the maximum size of L ?

L might contain up to 10^6 elements.

59.2.1 Solution using Sorting

Would this problem be easier if the input array was sorted to begin with? The answer to this question is yes, as, consecutive elements would appear one after the other and therefore, would be easy to check for sequences of consecutive elements with a single linear visit of the array.

Let's imagine for a second that the input of Example 59.1 was sorted i.e. $L = \{28, 29, 30, 31, 45, 46, 47, 235\}$. We can see that we have three sequences of elements and that the first start at the beginning of the array and ends at index 3 ($\{28, 29, 30, 31\}$), the second starting at index 4 ($\{45, 46, 47\}$) and ending at index 6 while the third sequence counts only of the last element, $\{235\}$.

It is easy to see that each sequence ending at at index x is such that either x is the last index of the sequence or $L[x+1] = L[x] + 1$. We can use this simple observation to scan the input array and look for indices satisfying this property and by keeping track of the size of the longest we can solve this problem.

Listing 59.1 shows an implementation of this idea.

```

1  size_t longest_consecutive_sequence_sorting(std::vector<int>& L)
2  {
3      if (const auto size = L.size(); size <= 1)
4          return size;
5
6      std::sort(std::begin(L), std::end(L));
7      auto last = std::unique(L.begin(), L.end());
8      L.erase(last, L.end());
9
10     size_t ans = 1;
11     auto it_curr = std::begin(L);
12     auto it_next = std::next(it_curr);
13     size_t curr_count = 1;
14     while (it_next != std::end(L))
15     {
16         if ((*it_next) == (*it_curr) + 1)
17         {
18             curr_count++;
19             ans = std::max(ans, curr_count);
20         }
21         else
22         {
23             curr_count = 1;
24         }
25         it_curr = it_next;
26         it_next = std::next(it_curr);
27     }
28     return ans;
29 }
```

Listing 59.1: $O(n \log(n))$ time and $O(1)$ space solution using sorting.

The code works by first sorting and removing duplicates in L . Removing duplicates is not strictly necessary but it does makes the code after a little more simpler and clearer and does not make the overall time complexity worse. The variable `curr_count` keeps track of the size of the current sequence of consecutive numbers. This number is incremented everytime the current element is exactly equal to the next minus one and reset to 1 whenever this condition is not met. This would be the point where the sequence ends. The maximum value registered by this counter will be the final answer that is returned.

The time complexity of this solution is $O(n \log(n))$ as we sort the array and the subsequent while loop runs in linear time.

The space complexity is $O(1)$.

59.2.2 Linear time and space solution

We can however avoid sorting and removing duplicates altogether and solve this problem in linear time if we are willing to use also linear space. The idea would be to build an

undirected graph $G = (V, K)$ (V being the nodes and K the edges) from L where we have one node for each element of L . If we carefully connects these nodes in V such that nodes corresponding to consecutive elements are linked together then the result would be a graph with one or more connected components that looks very much like a disconnected doubly linked list. Each connected component is a sequence of consecutive elements and therefore all we have to do is to visit each of these connected components and find out the one with most nodes.

We know that we can visit a graph in linear time and therefore the only issue we have to make this solution work is being able to construct such a graph also in linear time. Luckily also this step is easily achievable in linear time. We can represent G with an hashmap mapping elements of V to their respective successor and predecessor nodes in V using a `std::unordered_map<int, std::pair<std::optional<int>, std::optional<int>>>`; we use the first and second component of the pair to store the predecessor and successor elements, respectively. The fields of the pair are of type `std::optional<int>` to represent the fact that a node might not have a successor or a predecessor.

We can build G bit by bit by scanning L one element at the time and for each of its element l we can check whether $l + 1$ and $l - 1$ are already present in G . When the predecessor $l - 1$ of l is present in G all we have to do is to connect $l - 1$ with l in such a way that we can go from $l - 1$ to l by moving forward (via its successor link in its associated `std::pair`) and that we can also go from l to $l - 1$ following l 's predecessor link. Similarly, when $l + 1$ is in G we connect l and $l + 1$ but this time we make sure we can go from $l + 1$ to l via $l + 1$'s predecessor link and from l to $l + 1$ via l 's successor link. Eventually when we have visited L entirely, all consecutive nodes will be connected and grouped together in a connected component and the resulting graph would be a collection of such connected components.

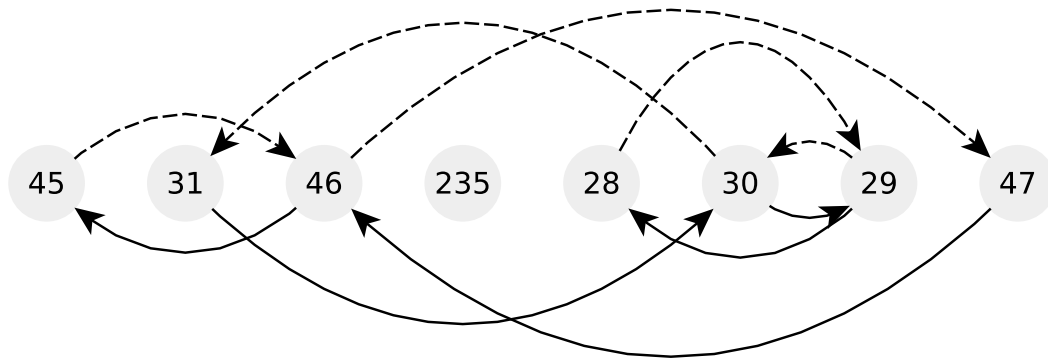
Figure 59.1a show how the graph would look like for the Example 59.1. Figure 59.1b shows the same graph with groups belonging to the same connected components depicted one close to the other. From these two figures it is clear we can visit a connected component and calculate its size and that we can start such a visit from any of its nodes.

An implementation of this strategy is shown in Listing 59.2.

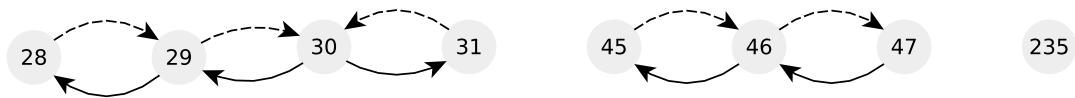
```

1 using pii = std::pair<std::optional<int>, std::optional<int>>;
2 using Graph = std::unordered_map<int, pii>;
3
4 Graph build_graph(std::vector<int>& L)
5 {
6     Graph ans;
7     for (const auto& curr : L)
8     {
9         if (ans.contains(curr))
10         {
11             continue;
12         }
13         ans.insert({curr, {}});
14
15         auto prev = curr - 1;
16         if (ans.contains(prev))
17         {
18             assert(!ans[prev].second);
19             ans[prev].second = curr;
20             ans[curr].first = prev;
21         }
22
23         auto next = curr + 1;
24         if (ans.contains(next))

```



(a) Graph representation of the Example 59.1. Dotted edged represents successors links.



(b) Graph shown in Figure 59.1a with nodes reordered to highlight the connected components. Dotted edged represents successors links.

Figure 59.1

```

25     {
26         assert(!ans[next].first);
27         ans[next].first = curr;
28         ans[curr].second = next;
29     }
30 }
31 return ans;
32 }
33
34 template <bool Direction>
35 size_t find_longest_connected_component(Graph& g, const int start_node)
36 {
37     size_t ans = 0;
38     auto curr_node = start_node;
39     while (g.contains(curr_node))
40     {
41         std::optional<int> connected_node = std::nullopt;
42         if constexpr (Direction)
43         {
44             connected_node = g[curr_node].second;
45         }
46         else
47         {
48             connected_node = g[curr_node].first;
49         }
50         g.erase(curr_node);
51         if (!connected_node.has_value())
52             break;
53         ans++;
54         curr_node = *connected_node;
55     }
56     return ans;
57 }

```

```

58
59 size_t find_longest_connected_component(std::vector<int>& L, Graph& g)
60 {
61     constexpr auto Left = false;
62     constexpr auto Right = true;
63     size_t ans = 0;
64     for (const auto& l : L)
65     {
66         if (!g.contains(l))
67         {
68             continue; // already visited
69         }
70         const auto length_left_from_l =
71             find_longest_connected_component<Left>(g, l);
72         // l is erased now: visit right from the next if exists and add 1 to
           account
73         // for the first hop done here
74         const auto length_right_from_l =
75             g.contains(l + 1)
76             ? find_longest_connected_component<Right>(g, l + 1) + 1
77             : 0;
78         ans = std::max(ans, length_left_from_l + length_right_from_l + 1);
79     }
80     return ans;
81 }
82
83 size_t longest_consecutive_sequence_lineartime(std::vector<int>& L)
84 {
85     if (const auto size = L.size(); size <= 1)
86         return size;
87
88     auto graph = build_graph(L);
89     return find_longest_connected_component(L, graph);
90 }

```

Listing 59.2: Linear time and linear space solution.

The code works in two distinct phases each implemented in its own function:

1. `build_graph`: where we construct the graph discussed above.
2. `find_longest_connected_component` where we take such a graph and we visit it.

The graph building part is pretty straightforward and nothing more than connecting nodes that are consecutive is done here.

The visit of the graph is possibly more interesting because we erase from the graph nodes as we visit them so that a connected component is only visited once (otherwise what would happen in case the entire graph had only a single connected component? we would visit all of the graph for each and every element of L , causing the time complexity to skyrocket to quadratic time). We try to start the visit of a connected component from each node $l \in L$. If l has been already visited, it would not be present in G and therefore we know it has been already considered. In case it yet to be visited then we visit the connected component starting from it by spawning two specialized visit functions (`find_longest_connected_component<Right>` and `find_longest_connected_component<Left>`) that are responsible for visiting the connected components (and keeping the count of the visited nodes) only going by using successors and predecessors links, respectively (effectively visiting the right and left side of the component disjointly). The length of the sequence would be the sum of the visited nodes by these two specialized visit functions plus 1, to account for l itself.

The function `template<bool Direction> find_longest_connected_component<Direction>` starts

a visit of the graph from a given node and only follows successor and predecessor links depending on the `constexpr` value of `Direction`.

Listing 59.2 has linear time and space complexities.

60. Merge k sorted lists

Introduction

The problem discussed in this chapter is quite interesting because it is rooted in the familiar merge-sort algorithm[[wiki:mergesort](#)] which is a divide and conquer algorithm that works by first splitting a list into smaller and smaller ones (see figure 60.1a) and after each of them is individually and separately sorted it merges them a pair at the time preserving the sorting property (see figure 60.1b). In this chapter, we will focus on only one of these phases, specifically the merge phase and, we will try to find an efficient way to augment it so that it will be capable of merging more than only a pair of sorted lists.

Coming up with a brute-force solution for merging k lists is not hard but the resulting end algorithm is rather inefficient. A faster and more efficient approach requires a bit more effort, and in the remainder of the chapter, we will investigate a couple of different approaches that we can take to solve this problem efficiently. In particular, we will have a look at the brute-force solution (in Section 60.2.1) and then in Section 60.2.2 to an approach that allows us to lower the time complexity quite a bit.

60.1 Problem statement

Problem 86 Write a function that, given k lists that are sorted in ascending order, merges them into a new sorted list.

■ **Example 60.1**

Given $L = [[1,4,5], [1,3,4], [2,6]]$ the function returns $[1,1,2,3,4,4,5,6]$ ■

■ **Example 60.2**

Given $L = [[1,2,3], [4,5,6], [7,8,9]]$ the function returns $[1,2,3,4,5,6,7,8,9]$ ■

■ **Example 60.3**

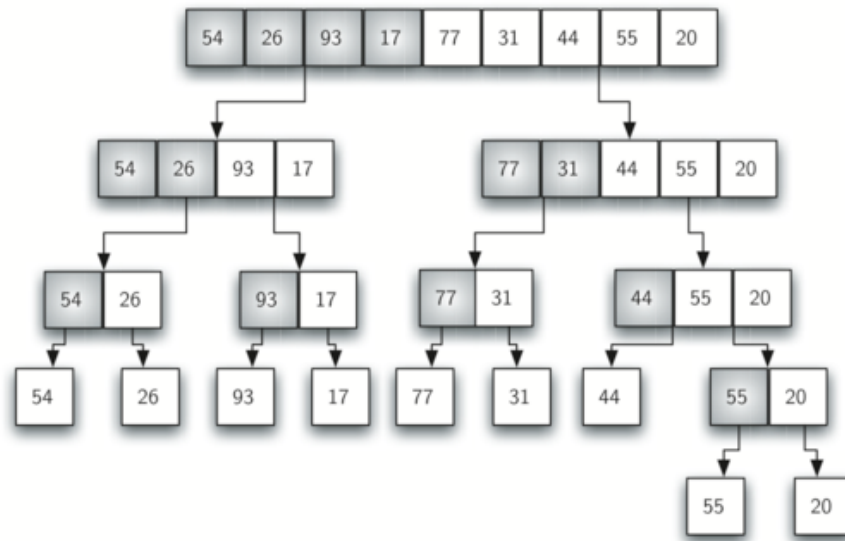
Given $L = [[7,8,9], [4,5,6], [1,2,3]]$ the function returns $[1,2,3,4,5,6,7,8,9]$ ■

60.2 Discussion

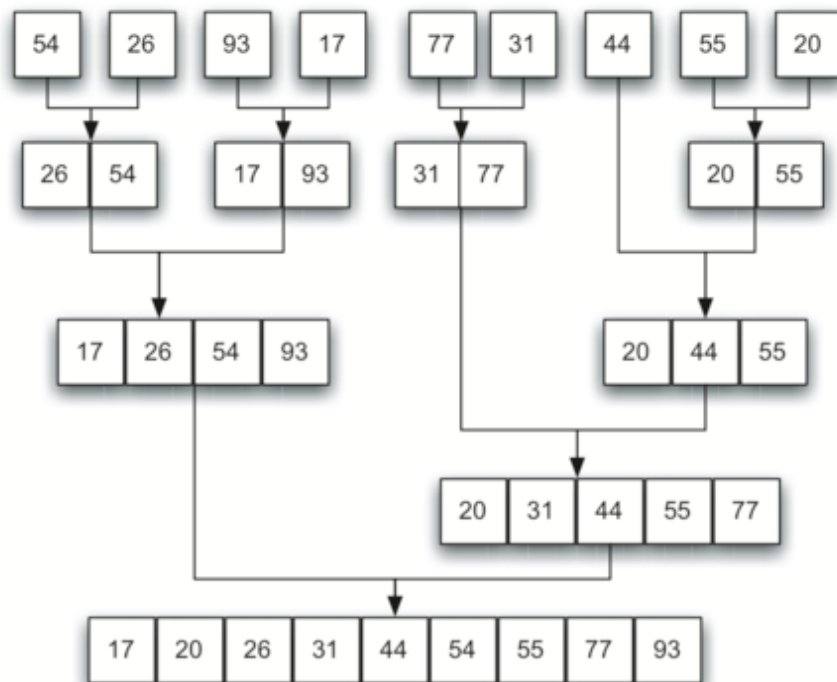
60.2.1 Brute-force

There is clearly a naive way of approaching this problem which involves maintaining a master list (let's refer to it as `sinkList`, which is initially empty), where the content of all k lists will eventually end up, and merging the content of each individual list into it. Merging the `sinkList` with the i^{th} input list L_i can be done exactly in the same way the merge sort does it. Repeating the process of merging the `sinkList` with all the k input lists eventually result in `sinkList` being exactly what we need to return and will contain all the data in the k input lists in the right order.

Listing 60.1 shows an implementation of this idea.



(a) First phase of the merge-sort where the a list is recursively split into smaller ones (half the original length) until we are left with lists of size 1



(b) Second phase of the merge-sort where the split lists are recursively merged preserving the sorting property.

```

1
2 Node<int>* insert_sorted(Node<int>* sinkList, Node<int>* toBeInserted)
3 {
4     Node<int>* ans = sinkList; // head of the merged list
5     if (!sinkList)
6     {
7         sinkList = toBeInserted;
8         return ans;
9     }
10
11     Node<int>* lastNodeSinkList = nullptr;
12     Node<int>* sinkListPrec = nullptr;
13     while (sinkList && toBeInserted)
14     {
15         lastNodeSinkList = sinkList; // remember the last node of the sinkList
16
17         if (sinkList->val <= toBeInserted->val)
18         {
19             sinkListPrec = sinkList;
20             sinkList = sinkList->next;
21             continue;
22         }
23
24         Node<int>* const toBeInsertedNext = toBeInserted->next;
25         toBeInserted->next = sinkList;
26         if (sinkListPrec)
27         {
28             sinkListPrec->next = toBeInserted;
29             // We inserted a value before sinkList. We need to update the prec
30             // pointer
31             sinkListPrec = toBeInserted;
32         }
33         else
34         {
35             // inserting at the front of sinkList.
36             ans = toBeInserted;
37             sinkListPrec = ans;
38         }
39         toBeInserted = toBeInsertedNext;
40     } // while
41
42     // attach the remaining part of toBeInserted list to the end of the sinkList
43     if (toBeInserted)
44     {
45         lastNodeSinkList->next = toBeInserted;
46     }
47     return ans;
48 }
49
50 Node<int>* merge_k_sorted_list_brute_force(std::vector<Node<int>*> lists)
51 {
52     if (lists.empty())
53         return nullptr;
54     if (lists.size() <= 1)
55         return lists.front();
56
57     Node<int>* sinkList = lists.front();
58     for (size_t i = 1; i < lists.size(); i++)
59     {
60         sinkList =
61             insert_sorted(sinkList, lists[i]); // insert list nodes into sinkList

```

```

61     }
62
63     return sinkList;
64 }

```

Listing 60.1: Brute-force solution reusing the two-list merging algorithm from the merge-sort algorithm.

The code work by having a driver function `merge_k_sorted_list_brute_force` issuing $k - 1$ calls to another function called `insert_sorted(Node <int >* sinkList , Node <int >* toBeInserted)`. The latter implements exactly what the merge phase of the merge sort does with the exception that it does so merging the nodes of its second parameter `toBeInserted` directly into the first one `sinkList` (effectively dismantling `toBeInserted` which would not be useable after the function returns).

The complexity of Listing 60.1 is $O(n^2)$ where n is the sum of the sizes of all input lists. The space complexity is $O(1)$ which is optimal as the returned list is constructed from the actual nodes that make up the k input lists.

60.2.2 Priority-queue approach

We can do way better than quadratic time if we are clever in using a priority queue. What we will try to do here is to insert one node at a time into the `sinkList`. At the beginning `sinkList` is empty and the first node will surely be the smallest among all the first nodes of the k input lists. Say list i had the smallest element; we now remove it from list i and add it to the `sinkList`. From where will the second node come from? Again it must be one (the smallest) of the first elements of the k input lists! In other words, because the individual lists are sorted, the “next” element to be inserted in `sinkList` must come from the front of the input lists. Why not use a priority queue to keep track of the smallest element among all fronts of the input lists? If we do so, all it is necessary is to pick whatever node n is at the top of the queue, insert it to the `sinkList` and top up the queue a brand new node which will come from the same list n belonged to.

The advantage of this approach is clear once we realize that now each element we get from the priority queue must be simply appended at the end of `sinkList`. Appending at the end of a list is a simple operation that can be performed in constant time. What about the price we pay for the priority queue? Both `pop` and `insert` operations have $\log(k)$ time complexity (the priority queue will only contain the front element of each of input lists). Therefore the entire algorithm will have $n\log(k)$ and $O(k)$ time and space complexity respectively.

Listing 60.2 shows an implementation of this idea.

```

1  template <typename T>
2  struct NodeWrapper
3  {
4      Node<T>* ptr{nullptr};
5      size_t list_idx{};
6  };
7
8  Node<int>* merge_k_sorted_list_priority_queue(std::vector<Node<int>>* lists)
9  {
10     if (lists.empty())
11         return nullptr;
12     if (lists.size() <= 1)
13         return lists.front();
14
15     auto compareNodeWrapper = [](const NodeWrapper<int>& node1,
16                                 const NodeWrapper<int>& node2) {

```

```

17     assert(node1.ptr && node2.ptr);
18     return node1.ptr->val > node2.ptr->val;
19 };
20
21 std::priority_queue<NodeWrapper<int>,
22                   std::vector<NodeWrapper<int>>,
23                   decltype(compareNodeWrapper)>
24                   queue(compareNodeWrapper);
25
26 std::vector<Node<int>*> lists_pointers(lists.size(), nullptr);
27 for (size_t i = 0; i < lists.size(); i++)
28 {
29     lists_pointers[i] = lists[i];
30     if (lists_pointers[i])
31         queue.push(NodeWrapper<int>{lists[i], i});
32 }
33
34 Node<int>* sinkList_current = nullptr;
35 Node<int>* sinkList_head   = nullptr;
36 while (!queue.empty())
37 {
38     auto [ptr, idx] = queue.top();
39     queue.pop();
40
41     if (sinkList_current)
42     {
43         sinkList_current->next = ptr;
44     }
45     else
46     {
47         sinkList_head = ptr;
48     }
49     sinkList_current = ptr;
50     if (lists_pointers[idx]->next)
51     {
52         lists_pointers[idx] = lists_pointers[idx]->next;
53         queue.push(NodeWrapper<int>{lists_pointers[idx], idx});
54     }
55 }
56 return sinkList_head;
57 }

```

Listing 60.2: Priority-queue-based solution.

The code works by first creating a `std::priority_queue` having as underlying storage a `std::vector` of `NodeWrapper` which, as the name suggests, is a wrapper around a normal `List Node` that also carries the information about the queue it belongs to.

The first loop of the function takes care of adding the first k nodes in the queue. The rest of the code is relatively straightforward with the main `while` loop popping the top element out of the queue (it is done by unpacking a `NodeWrapper` into its components: `ptr` and `idx`, the pointer to the actual list's `Node` and the index of the list this node belongs to) appends it at the end of the `sinkList` and then proceeds in adding a new element into the queue that is drawn from the very same queue the last element popped out the queue belonged to (this info is stored in `idx`). This element could of course not always exist when `ptr` is the last element of the list number `idx`.

When the queue is empty, we have finally inserted every element in the k list and we can return `sinkList` which now contains all the n elements in all input lists in the proper order.

61. Mini Problems

61.1 Greatest Common Divisor

Problem 87 Write a function to calculate the GCD of two integers.

■ **Example 61.1**

Given 35 and 28 the function returns 7. ■

■ **Example 61.2**

Given 15 and 8 the function returns 1. ■

61.1.1 C++ Brute-force

The GCD of two numbers x and y is defined as the largest integer that divides both x and y . A simple and inefficient solution would simply loop over all numbers from the smallest between x and y and would stop as soon as we find one that divides both. We are guaranteed to find such a number as the number 1 will happily divide any number. This solution is shown in Listing 62.1.

```
1 int gcd_bruteforce(const int x, const int y)
2 {
3     if (x < y)
4         return gcd_bruteforce(y, x);
5
6     for (int i = y; i > 1; i--)
7         if ((x % i == 0) && (y % i == 0))
8             return i;
9     return 1;
10 }
```

Listing 61.1: Brute-force, linear time solution.

61.1.2 Log-time solution. Euclidean Algorithm

A much faster solution can be achieved by using the Euclidean algorithm for the GCD. This algorithm is based on the principle that the greatest common divisor of two numbers x and y does not change if the larger number is replaced by the remainder of the integral division between x and y .

For example, 21 is the GCD of 252 and 105 (as $252 = 21 \times 12$ and $105 = 21 \times 5$), and the same number 21 is also the GCD of 105 and $252 \bmod 105 = 42$. Since this replacement reduces the larger of the two numbers, repeating this process gives successively smaller pairs of numbers until we reach a point where the smallest number divides the largest and therefore the remainder is zero (in the worst case the smallest becomes 1).

It was proven by Gabriel Lamé in 1844 that this algorithm always terminates in less steps than five times the number of digits of the smaller number (in base 10) making this algorithm extremely efficient as the number of digits grows logarithmically compared the number it represents.

Listing 62.2 shows a recursive implementation of this algorithm while Listing 62.3 shows an iterative one.

```
1 int gcd_euclide(const int x, const int y)
2 {
3     if ((x % y == 0))
4         return y;
5
6     const auto reminder = x % y;
7     return gcd_euclide(y, reminder);
8 }
```

Listing 61.2: Euclide algorithm, recursive implementation.

```
1 int gcd_euclide_iterative(int x, int y)
2 {
3     while ((x % y) != 0)
4     {
5         const auto reminder = x % y;
6         x = y;
7         y = reminder;
8     }
9     return y;
10 }
```

Listing 61.3: Euclide algorithm, iterative implementation.

61.1.3 C++ Compile-time

It is quite common to see specific requirements on the compile implementation of the GCD algorithm. Therefore in this section we will see how we can calculate the GCD in C++ at compile time.

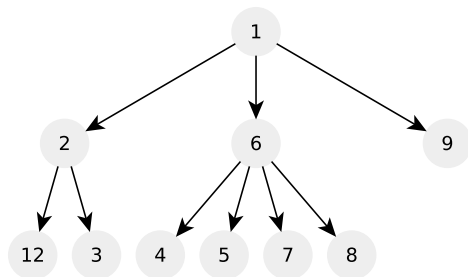
Before C++-11 the only way to do compile time computation was by using templates. In order to calculate GCD using templates we will use a structure with two integral template parameters that we will manipulate in a `static const` variable. An implementation of this idea is shown in 62.4.

```
1 template <int x, int y>
2 struct GCDEuclidePreCpp11
3 {
4     static const int gcd = GCDEuclidePreCpp11<y, x % y>::gcd;
5 };
6
7 template <int x>
8 struct GCDEuclidePreCpp11<x, 0>
9 {
10     static const int gcd = x;
11 };
```

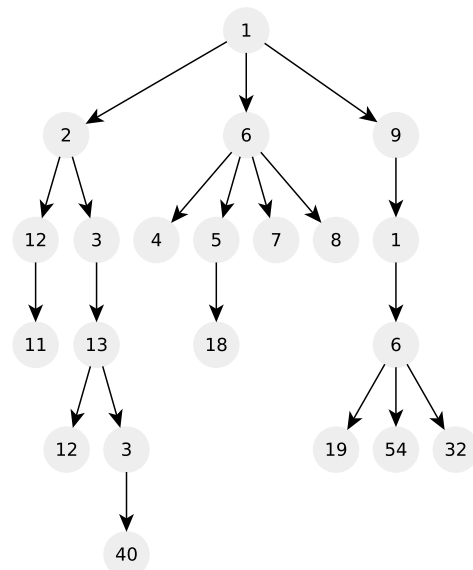
Listing 61.4: Pre C++11, compile-time template based solution.

The code works by have a template class with two integral template parameters and a partial specialization that is used to terminate the recursion which is triggered whenever we request the static field `::gcd`.

C++-11 introduces `constexpr` function that can be used to specify function that can be run at compile-time. In C++-11 there are quite some limitation in what statements and operations we can do in a `constexpr` context: for instance we can only have one return statement. Most of these constraints are related in the subsequent versions of the standard. A `constexpr` recursive solution that works in C++-11 is shown in Listing 62.5.



(a) Input tree for Example 62.8.



(b) Input tree for Example 62.9.

Figure 61.1

```

1 constexpr int gcd_euclide_cpp11(const int x, const int y)
2 {
3     return (x % y == 0) ? y : gcd_euclide_cpp11(y, x % y);
4 }
  
```

Listing 61.5: C++11 constexpr based solution.

Notice that, from C++14 we can decorate Listing 62.3 with `constexpr` so that it can be used in compile-time computation.

61.2 Maximum Depth of N-ary Tree

Problem 88 Given a N-ary tree, return its depth which is defined as the length of the longest path from the tree's root to any of its leaves.

■ Example 61.3

Given the tree depicted in Figure 62.1b the function returns 3 (path from node 1 to node 12). ■

■ Example 61.4

Given the tree depicted in Figure ?? the function returns 6 (path from node 1 to node 40). ■

61.2.1 Discussion

This is a classical problem, mostly asked during phone screening due to its simplicity. What the problem, in other words, is asking us to do, is to return the maximum level of any of the tree's nodes. The level of a node is the number of its ancestors plus one. Therefore to solve this problem, all we have to do is to visit the tree and keep track of the number of ancestors which is equivalent to the number of steps down the tree we took. We

know that the root has 0 ancestors and therefore, we know that each of its children will have 1 ancestor (the root itself) and that any of its grandchildren will have 2 ancestors and so on.

Visiting a tree can be equally easily done recursively and iteratively as shown in Sections 62.1 and 62.1, respectively.

For the remainder of the discussion, we will define the root of a tree to be a pointer to the `Node` structure defined in Listing 62.6.

```
20 class Node {
21 public:
22     int val;
23     vector<Node*> children;
24
25     Node() {}
26
27     Node(int _val) {
28         val = _val;
29     }
30
31     Node(int _val, vector<Node*> _children) {
32         val = _val;
33         children = _children;
34     }
35 };
```

Listing 61.6: Node definition.

61.2.2 Recursive solution

The recursive solution has the advantage of being shorter in terms of lines of code and in our opinion more elegant. It could, however, underperform the iterative solution if the compiler is not able to optimize the code properly or lead to out of memory errors if the tree depth goes over a certain value because at any given time, we keep in the stack a number of activation records that is equal to the level of the node we are visiting.

Listing 62.7 shows an implementation of this approach and has a linear time and space (considering the space occupied by the activation records) in the number of nodes of the tree.

```
1 int n_ary_treedepth_recursive(Node* root)
2 {
3     if (!root)
4         return 0;
5     int max_depth_child = 0;
6     for (auto& childPtr : root->children)
7     {
8         max_depth_child =
9             std::max(max_depth_child, n_ary_treedepth_recursive(childPtr));
10    }
11    return 1 + max_depth_child;
12 }
```

Listing 61.7: Recursive solution.

61.2.3 Iterative Solution

The same idea can be implemented iteratively as shown in Listing 62.8 where we use a stack to keep track of the nodes **and their level**. Whenever we visit a node, we compare its level with the maximum found so far, then we remove it from the stack and insert all of its children in the stack with a level value increased by one.


```

1 int n_ary_treedepth_iterative(Node* root)
2 {
3     if (!root)
4         return 0;
5
6     std::stack<std::pair<Node*, int>> nodes;
7     nodes.push({root, 1});
8
9     int ans = 1;
10    while (!nodes.empty())
11    {
12        auto [node, level] = nodes.top();
13        nodes.pop();
14        ans = std::max(ans, level);
15        for (const auto child : node->children)
16        {
17            nodes.push({child, level + 1});
18        }
19    }
20
21    return ans;
22 }

```

Listing 61.8: Iterative solution.

61.3 Assigning cookies 🍪

Problem 89 You are the parent of n children and you want to make them happy by giving them a cookies (real ones 🍪, not HTTP cookie). However you know that too many cookied are not good for them and therefore you settle for a rule that each child can at most get one cookie. Giving cookies to your children is not easy as they are special and each child i has an integer greed factor $g[i]$ associated, which is the minimum size of a cookie that the child i will be content with; You have at your disposal a number m of cookies, and each cookie j has an integer size $s[j]$. You can give cookie number i to child number i if and only if $s[j] \geq g[i]$. When a child has a cookie assigned is content.

Write a function that given a list (G) of n greed values and a list (C) of m cookie sizes determines the maximum possible of children you can make content.

■ Example 61.5

Given $G = \{1, 2, 3\}$ and $C = \{1, 1\}$ the function returns 1. Despite having two cookies we can only make one child happy because we do not have cookies big enough for children number 2 and 3. ■

■ Example 61.6

Given $G = \{1, 3\}$ and $C = \{1, 4\}$ the function returns 2. We can give the first cookie to the first child and the second cookie to the second child. ■

■ Example 61.7

Given $G = \{1, 2, 3\}$ and $C = \{1, 1, 3\}$ the function returns 2. We can give the first cookie to the first child and the the third cookie to the second child The second cookie remains unassigned and the third child without cookie assigned. ■

61.3.1 Discussion

Let's start by noticing that despite what shown in the Exmaples the input lists are not guaranteed to be sorted and that, sorting actually makes solving this problem much easier. The idea is that we have to find a way to assign to each children the smallest cookie possible that has size higher or equal to his greed. If the input array is not sorted then for each greed value we are forced to search S entirely for a suitable cookie. If on the other hand both G and S are sorted then, we can try to accomodate children by increasing greed and keep track and assign progressively larger cookies to them. If a children with greed i can be assigned cookie number j , then we can try to assign to child number $i+1$ cookie number $j+1$. If that does not work then we can try with cookie number $j+2$ and so on. When we cannot assign a cookie j to a certain child i , cookie j will be unused, but that is not an issue because there is no way we can assign cookie j to any other children because the greed values for children $i+1, i+2, \dots$ will all be greater than the gree of child i .

Therefore in order to solve this problem we can:

- sort G ;
- sort S ;
- use two pointers to keep track of the current child and current cookie
- process one child a the time until we ran out of children or cookies
- if the current cookie size is greater than the current child greed then we can advance both pointers
- otherwise we can only hope the next cookie will be assignable and therefore only advance the cookie pointer.

An implementation of this idea is shown in Listing 62.10. Its time complexity is $O(n \log(n))$ while its the space complexity is $O(1)$.

```
1  int max_content_children(vector<int>& g, vector<int>& s)
2  {
3      std::sort(g.begin(), g.end());
4      std::sort(s.begin(), s.end());
5
6      auto it_g = g.begin();
7      auto it_s = s.begin();
8      size_t ans = 0;
9      while (it_g != g.end() && it_s != s.end())
10     {
11         if (*it_s >= *it_g)
12         {
13             std::advance(it_g, 1);
14             ans++;
15         }
16         std::advance(it_s, 1);
17     }
18     return ans;
19 }
```

Listing 61.9: Solution using sorting.

61.4 Maximize Sum Of Array After K Negations

Problem 90 Given an integer array *nums* and an integer *k*, modify the array in the following way: choose an index *i* and replace *nums*[*i*] with $-nums[i]$.

You should apply this process **exactly** *k* times. You may choose the same index *i* multiple times.

Return the largest possible sum of all the elements of *nums* at end of this process.

■ **Example 61.8**

Given *nums* = {4,2,3} and *k* = 1 the function returns 5. We can choose index 1 and turn the 2 into -2. At the end we will be left with *nums* = {4,-2,3} which totals to 4 - 2 + 3 = 5. ■

■ **Example 61.9**

Given *nums* = {3,-1,0,2} and *k* = 3 the function returns 6. We can choose ■

61.4.1 Discussion

One easy way of solving this problem relies on the fact that all we have to do is to apply the change sign change always on the smallest number of the array. The intuition behind it is that we should aim at first changing the sign of all the negative numbers first and among them we should prioritize the smallest ones: the number with the largest absolute value and negative sign. Changing the sign of those number will bring the best increase in the overall sum of the array.

If after having changed all negatives into positive we are left with more moves to make then we still have to change the sign of the smallest number in the array as many times as necessary. This causes the smallest number of the array to switch sign back and forth until *k* = 0 (this step can be optimized by noticing that if the number of moves left is even then the final value of the smallest number in the array is not going to change, otherwise, it will be negative. We can reach this conclusion without having to actually perform the sign switch).

To always keep track of the smallest number we can use a `std::priority_queue` as shown in Listing 62.1.

```
1 int largestSumAfterKNegations(vector<int>& nums, int k)
2 {
3     std::priority_queue<int, std::vector<int>, std::greater<int>> P(nums.begin(),
4                                                                    nums.end());
5     while (k--)
6     {
7         auto x = P.top();
8         P.pop();
9         P.push(-x);
10    }
11    int ans = 0;
12    while (!P.empty())
13    {
14        ans += P.top();
15        P.pop();
16    }
17    return std::accumulate(P.begin(), P.end(), 0);
18 }
```

Listing 61.10: Solution using sorting.

The code works by applying the *k* modifications always to the smallest element of the queue. At the end of the process we simply sum every element in the queue to obtain the answer. The complexity of this approach is $O(k\log(n) + n\log(n))$ in time and $O(1)$ in space.

61.5 Pairs of Songs With Total Durations Divisible by *k*

Problem 91 You are given a list T of songs where the i^{th} song has a duration of $time[i]$ seconds. Write a function that given T returns the number of distinct unordered pairs of songs for which the sum of their durations is divisible by an integer k .

In other words, the function should count the number of indices of $i < j$ such that $(T[i] + T[j]) \bmod k == 0$.

■ **Example 61.10**

Given $T = \{30, 20, 150, 100, 40\}$ and $k = 60$, the function returns 3. We can pair songs at indices:

- 0 and 2 for a total duration of 180;
- 1 and 3 for a total duration of 120;
- 1 and 4 for a total duration of 60.

■

■

61.5.1 Discussion

This problem is quite similar to the two number sum problem discussed in Chapter 4 and we will therefore use the very same technique to solve it (we will avoid discussing sub-optimal solution as these are discussed already in the two number sum problem). The difference here is that we are only interested in the number modulo k and our main goal is to find two numbers whose remainder sum up to 0. For instance w.r.t. Example 62.10 we can see that $T[0] + T[2] = 180$ which is divisible by 60. If we have a look at their modulus we notice that: $(T[0] \bmod 60) + (T[2] \bmod 60) = 30 + 30 = 60 \bmod 60 = 0$. The same holds for the other two pairs of this example:

- $(T[1] \bmod 60) + (T[3] \bmod 60) = 20 + 40 = 60 \bmod 60 = 0$
- $(T[1] \bmod 60) + (T[4] \bmod 60) = 20 + 40 = 60 \bmod 60 = 0$

Listing 62.11 shows an implementation of this idea.

```

1  int numPairsDivisibleBy60(const vector<int>& T, const int k)
2  {
3      std::vector<int> mod_counters(k, 0);
4      int ans = 0;
5      for (const auto time : T)
6      {
7          const int m = time % k;
8          const int r = (k - m) % k;
9          ans += mod_counters[r];
10         mod_counters[m]++;
11     }
12     return ans;
13 }
```

Listing 61.11: Solution based on the two number sum problem.

61.6 Trim text

Problem 92 You are given a string m of length n and an integer k . m is guaranteed to consist only of English alphabet letters and spaces. Write a function that crops s so that it becomes of length smaller or equal than k . However the cropped messages cannot:

- crop away part of a word;
- have trailing spaces;

- have length greater than k ;

Moreover the cropped string should be as long as possible i.e. any other cropped message satisfying the constraints above should be smaller than the output of your function.

■ Example 61.11

Given the input string `Lorem ipsum dolor sit amet` and $k = 6$ the function returns `Lorem`. If $k = 20$ the function returns `Lorem ipsum dolor`.

When if for instance $k = 2$ or $k = 3$ the function returns an empty string. ■

61.6.1 Discussion

This problem on strings is all about implementation and there is no algorithmic insight that we need to have to solve it efficiently. However, this does not automatically make this problem an easy one as implementation-focused problems are often hard to get right. However, this specific problem can be tackled quite efficiently if we notice that we can remove characters from the back of s if:

- they are spaces, regardless of whether the current size of the string (the original string size minus the characters removed so far) is less than k ;
- they are alphanumeric and the current length is strictly greater than k .

In particular, we can notice that we can safely remove trailing space from the original input as the problem statement clearly states that we cannot return a string with any of them. Moreover, when we remove alphanumeric characters from the back of the string we are doing it only because the current size of s is still strictly larger than k . When this happens we must remove all of these characters up until we reach a space. This is because we are forced not to crop away parts of words and we have to either keep a word or remove it entirely.

Listing 62.12 shows an implementation of this idea.

```

1  #include <cctype>
2
3  template <typename Fn>
4  void skip_if(const std::string& msg, int& pos, Fn fn)
5  {
6      while (pos >= 0 && fn(msg[pos]))
7      {
8          pos--;
9      }
10 }
11
12 std::string trim_text_lineartime(const std::string& message, const size_t K)
13 {
14     int pos = message.size() - 1;
15     while (pos >= 0)
16     {
17         skip_if(message, pos, [](const auto& c) { return std::isspace(c); });
18         if ((pos + 1) <= K)
19             break;
20         skip_if(message, pos, [](const auto& c) { return std::isalnum(c); });
21     }
22     return message.substr(0, pos + 1);
23 }
```

Listing 61.12: Linear time solution.

The function `skip` is used to update the variable `pos`, which is an index in `s`. `pos` keeps track of the portion of the input string we have not yet cropped. `skip` moves `pos` backwards until the user-provided function `fn` returns true (and we have not reached the left limit of `s`). `skip` is used by the main driver function `trim_text_lineartime` which is a function that repeatedly removes any trailing spaces and then, if the length of `s` is still too large, proceed in removing an entire word. The word is also (like for spaces) removed by the function `skip` which takes care of removing any character until it reaches the first non-alphanumeric character which we are assured to be a space (there are no other characters allowed in `s`).

The complexity of this approach is linear in time and constant in space (if we do not consider the space necessary for the output).

61.7 Items and bags

Problem 93 A food bank is developing a system to help reduce its usage of plastic bags. The bank collects food from people that bring it in plastic bags. A plastic bag has a certain capacity c , which indicates the amount of food we can fit in it. In particular, a bag with a capacity c can carry up to c unit of goods. The goal of the system is to rearrange the food in the warehouse into as few bags as possible to help reduce the waste footprint.

You are given two arrays F and B each consisting of N integers where $F[i]$ is the amount of food currently present in the bag i that has capacity specified in $F[i]$. Write a function that returns the minimum number of bags needed to carry all food.

Notice that you are guaranteed $B[i] \geq F[i]$ to always hold.

■ Example 61.12

Given $F = \{1, 4, 1\}$ and $B = \{1, 5, 1\}$ the function returns 2. We could move 1 unit of food from bag 0 and put it in the bag 1 (that currently holds 4 units of food but has capacity 5). ■

■ Example 61.13

Given $F = \{2, 3, 4, 2\}$ and $B = \{2, 5, 7, 2\}$ the function returns 2. All food in the bag 0 can be moved to bag 1 and all the food in bag 3 can go in bag 2. ■

61.7.1 Discussion

This problem can be quickly solved with a greedy approach if we realize that we have a total amount of food units equal to $T = \sum F[i]$ that needs to fit into a subset of the N bags we have at our disposal. The optimal arrangement is to fill the bags to their maximum capacity to avoid waste of space and to minimize the number of bags used we better use larger bags firsts as we can stuff more food units in them. Therefore, all we have to do to solve this problem is to sort the bags by capacity in descending order (larger bags first) and to simulate the process of filling them up until all the units of food are stored.

W.r.t to example 62.13: we have $T = 11$ units of food. If we put the first 7 into the bag 2 we are left with $T = 11 - 7$ units of food not yet inside a bag. The next biggest bag is the one at index 1 with a capacity of 5. This bag can store all the T units of food left. At this point, we can stop the simulation and return 2.

Listing 62.13 shows an implementation of this idea.

```
1 #include <algorithm>
2 #include <numeric>
```

```

3
4 size_t items_and_bags(const std::vector<int> &F, std::vector<int> &B) {
5     std::sort(B.begin(), B.end(), std::greater<int>());
6     auto sum = std::accumulate(F.begin(), F.end(), 0);
7     size_t Bidx = 0;
8     while(sum > 0){
9         sum -= B[Bidx];
10        if(sum <= 0)
11            break;
12        Bidx++;
13    }
14    return Bidx+1;
15 }

```

Listing 61.13: Solution based on sorting.

The code works by first sorting B in descending order and calculating the total amount of units of food we must store. Notice that we use `std::accumulate` to perform this task instead of an explicit raw loop as it is more idiomatic and expressive. The while loop takes care of performing the simulation, and each iteration tries to fill as much food as possible into a bag. The simulation stops when all food is safely inside a bag. Notice that we access B using the variable `Bidx` without checking that `Bidx < B.size()` as the problem statement clearly states that there is always enough space among all bags to store all food. Therefore the while loop is guaranteed to put all food inside a bag before we ran out of bags.

The complexity of this approach is $O(N \log(N))$ due to sorting while the space complexity is $O(1)$.

61.8 Coupons

Problem 94 You want to buy n item from Amazon. Their prices is stored in an array P where $P[i] \geq 0$ contains the price for the item i . Amazon offers you the possibility to apply coupons to each item before the checkout and this causes the price of that item to halve. For instance, if the full price of an item is 4.60\$, then after applying the coupons it costs 2.30\$. You can apply coupons to an item multiple times, this means that if you apply the coupon three times to an item with price X \$, its final price will be $\frac{\frac{X}{2}}{2} = \frac{X}{2^3}$. Your task is to write a function that calculates the minimum number of coupons needed in order to lower the cumulative checkout price for all of the n items by half.

■ Example 61.14

Given $P = \{5, 19, 8, 1\}$ the function should return 3. Initially the cart price is $5 + 19 + 8 + 1 = 33$. By applying two coupons to the item at index 1 the price lowers to $5 + \frac{19}{4} + 8 + 1 = 18.75$ which is still higher than $\frac{33}{2} = 16.5$. We can apply a third coupon to the item at index 0 and the total price becomes $\frac{5}{2} + \frac{19}{4} + 8 + 1 = 16.25$ which is good enough.

■

■

61.8.1 Discussion

When applying a coupon to an item with price X we are going to save $\frac{X}{2}$ from its original price. If we only had one coupon to use, we would naturally apply it to the most expensive item as this will yield the biggest saving. We can extend this reasoning to n coupons and apply the n^{th} coupon to the most expensive item after having used $n - 1$ coupons. To solve

this problem we can simulate the process of applying coupons. As we use the coupons we must keep track of the item in the cart with the highest price, to which we can apply the coupon. We repeat this process until the amount of money saved is higher than half of the original cart price.

A `priority_queue` can be used to keep track of the most expensive item in the cart.

Listing 62.14 implements this idea.

```

1  #include <numeric>
2  #include <priority_queue>
3
4  int coupons_priority_queue(std::vector<int> &P)
5  {
6      double original_cart_total_price = std::accumulate(P.begin(), P.end(), 0.0);
7      const double half_original_cart_total_price = original_cart_total_price /
8              2.0;
9
10     std::priority_queue<double, std::vector<double>> prices_after_coupons(
11         P.begin(), P.end());
12     int ans = 0;
13     while (original_cart_total_price > half_pollution)
14     {
15         const auto best = prices_after_coupons.top();
16         prices_after_coupons.pop();
17         const double money_saved = best / 2.0;
18
19         original_cart_total_price -= money_saved;
20         prices_after_coupons.push(money_saved);
21         ans++;
22     }
23     return ans;

```

Listing 61.14: Priority-queue based vsolution.

The code works by creating a `priority_queue` where prices are sorted in descending order. It is initialized with the prices in P . The while loop continuously pop elements from the top of the queue, calculate the amount of money saved by applying the coupon to the popped-out element, and adds the halved prices back into the queue.

If it is not difficult to show that the while loop runs at most n times as if we half the price of each element in P , we will as a consequence also halve $\sum P[i]$ (the original cart price). Therefore the time complexity is $O(n \log(n))$ (remember that operations on the queue are $O(\log(n))$). The space complexity is $O(n)$ due to the space required by the priority queue itself.

62. C++ questionnaire

C++ Question 1 (Solution 1 at page 379)

Which of the following statements is true for the code below:

```
struct X
{
    A a;
    B b;
    X() : a{}, b{} {}
};

struct Y : X
{
    C c;
    D d;
    Y() : d{}, c{} {}
    ~Y() { }
```

- A. Destruction of type `Y` will call member destructors in the following order `A::~~A()`, `B::~~B()`, `D::~~D()`, `C::~~C()`
- B. Destruction of type `Y` will call member destructors in the following order `A::~~A()`, `B::~~B()`, `C::~~C()`, `D::~~D()`
- C. Destruction of type `Y` will call member destructors in the following order `C::~~C()`, `D::~~D()`, `B::~~B()`, `A::~~A()`
- D. Destruction of type `Y` will call member destructors in the following order `D::~~D()`, `C::~~C()`, `B::~~B()`, `A::~~A()`
- E. Destruction of type `Y` will only call destructors of classes `C` and `D` as the destructor of class `X` is not called from `Y::~~Y()`

C++ Question 2 (Solution 2 at page 379)

What will be the result of the code below?

```
const char* ptr1 = "123456";
const char* ptr2 = "1234567";

if(ptr1 == ptr2)
    printf("same address");
else
    printf("different address");
```

- A. "different address"
- B. "same address"
- C. compilation error
- D. runtime error

C++ Question 3 (Solution 3 at page 379)

Which of the following pointer declarations will allow you to modify the value the pointer points to?

- A. `int* ptr;`
- B. `const int* ptr;`
- C. `const int* const ptr;`
- D. `int const* ptr;`
- E. `int const* const ptr;`
- F. `int* const ptr;`

C++ Question 4 (Solution 4 at page 379)

Which of the following statements are true for the code below?

```
class Account
{
    mutable std::mutex m_;
    unsigned balance_;

public:
    friend void transfer(Account& src, Account& dst, unsigned amount)
    {
        std::lock_guard<std::mutex> lck_src(src.m_);
        std::lock_guard<std::mutex> lck_dst(dst.m_);
        src.balance_ -= amount;
        dst.balance_ += amount;
    }
};
```

- A. Code is thread safe thanks to usage of `lock_guards` that will prevent races and deadlocks.
- B. Mutable `std::mutex` makes this code not thread safe.
- C. Code is not exception safe.
- D. Code is not deadlock-free.
- E. Using `std::lock()` would be better to lock the mutexes.

C++ Question 5 (Solution 5 at page 379)

What will be the address the `ptr` points to after code execution? Assume that `CHAR_BITS` equals 8.

```
int32_t* ptr = (int32_t*) 0x20000004;
ptr += 2;
```

- A. `0x20000000`
- B. `0x20000002`
- C. `0x20000004`
- D. `0x20000006`
- E. `0x20000008`
- F. `0x20000010`
- G. `0x2000000C`

C++ Question 6 (Solution 6 at page 380)

What is the value of `x` after the call to `foo`?

```
uint8_t foo(uint8_t a)
{
    return ++a;
}

int main()
{
    uint8_t x = foo(std::numeric_limits<uint8_t>::max());
    return 0;
}
```

- A. 4294967295
- B. 255
- C. 0
- D. -1
- E. -2147483647 - 1
- F. 129
- G. -128
- H. Compilation error
- I. Undefined behavior

C++ Question 7 (Solution 7 at page 380)

In which of the following statement is Return Value Optimization (RVO) guaranteed to happen? Assume C++-17 standard.

- A. `FooClass a; FooClass b = a;`
- B. `auto a = CreateMyClass();`
- C. `FooClass a{ CreateFooClass()};`
- D. `FooClass a; a = CreateFooClass();`

C++ Question 8 (Solution 8 at page 380)

What is the order in which destructors are called when `a` gets out of scope?

```
struct base0 { ~base0(); };
struct base1 { ~base1(); };
struct member0 { ~member0(); };
struct member1 { ~member1(); };
struct local0 { ~local0(); };
struct local1 { ~local1(); };
struct derived: base0, base1
{
    member0 m0_;
    member1 m1_;
    ~derived()
    {
        local0 l0;
        local1 l1;
    }
}

void userCode()
```

```
{  
    derived d;  
}
```

C++ Question 9 (Solution 9 at page 380)

Which of the following statement results in a vector of 100 values all initialized with the value 0 in C++?

- A. `std::vector<int> v = 100, 0;`
- B. `std::vector<int> v; v.reserve(100);`
- C. `std::vector<int> v; v.resize(100);`
- D. `std::vector<int> v(100, 0);`
- E. `std::vector<int> v{100, 0};`

63. C++ questionnaire solutions

C++ Answer 1 (Question 1 at page 375)

Correct answer is **D**.

The fields of a class are destructed in the **reverse** order they appear in the source. The fields of `y` are destructed first followed by the fields of `x`.

In general, the base class destructors are invoked in reverse order as they appear in the inheritance list.

C++ Answer 2 (Question 2 at page 375)

Correct answer is **A**.

`ptr1` and `ptr2` are two distinct pointers pointing at two distinct string literals.

Usually string literals are places in the so called “read-only-data” section of the binary which gets mapped into the process space as read-only (which is why you can’t change it). It does vary by platform. For example, simpler chip architectures may not support read-only memory segments so the data segment will be writable.

C++ Answer 3 (Question 3 at page 376)

Correct answer are: **A,F**.

- A. `int* ptr;` is a non-const pointer to a non-const integer.
- B. `const int* ptr;` is a non-const pointer to a const int.
- C. `const int* const ptr;` is a const pointer to a const int.
- D. `int const* ptr;` is a non-const pointer to a const int
- E. `int const* const ptr;` is a const pointer to a const int.
- F. `int* const ptr;` is a const pointer to a non-const int.

C++ Answer 4 (Question 4 at page 376)

C++ Answer 5 (Question 5 at page 376)

Correct answer is **G**.

Each element pointed by `ptr` is 4 bytes and therefore we are going to add $8 = 2 \times 4$ to the start address.

On pointers we can perform the following operations:

- Increment `++`, `+`, `+=`
- Decrement `--`, `-`, `-=`
- Comparison `==`
- Assignment `=`

When performing an increment operation on a pointer `ptr` or type `T` like: `ptr += 2` the value of the address pointed by `ptr` will be increased by `2*sizeof(T)`. The decrement operation works in a similar fashion. This means that if we have a pointer `ptr` pointing

to the first element in an array `A` i.e. `A[0]`, then, the following causes `ptr` to point to the fifth element in `A` i.e. `A[4]`: `ptr = ptr + 4;`

It is important to notice that you can only use integer as right parameters for the increase and decrease operations on pointers and that you cannot add a pointer to another pointer.

C++ Answer 6 (Question 7 at page 377)

Correct answer is **C**.

As opposed to overflow/underflow for signed integer where it is indeed undefined behaviour, the standard clearly states that unsigned integers shall obey the arithmetic under 2^n modulo: [basic.fundamental]

“n unsigned integer type has the same width N as the corresponding signed integer type. The range of representable values for the unsigned type is 0 to $2^N - 1$ (inclusive); **arithmetic for the unsigned type is performed modulo 2^N** ”.

C++ Answer 7 (Question ?? at page ??)

Correct answers are: **A,B** and check the rest.

C++ Answer 8 (Question 8 at page 377)

The correct answers is: `~local1()`, `~local0()`, `~member1()`, `~member0()`, `~base1()` and at last `~base0()`.

C++ Answer 9 (Question 9 at page 378)

Correct answers are: **C,D**.

In particular option **B** is wrong because `reserve` only allocate spaces for 100 elements but it does not create any of them. `resize` on the other hand actually increases the size of the array and causes the constructor to be called for each of the 100 elements which, for `int`, mean initialization to the value 0 by default.

Appendices

Dynamic Programming

Dynamic programming (DP) is a popular technique for solving a certain class of optimization problems efficiently and is accredited to the American Scientist Richard Bellman[1]. He coined the term DP in the context of solving problems involving a series of best decision one after the other. The word *programming* can be a bit deceiving for computer scientists or programmers in general but it has really little to do with computer programming and it is in fact intended as a set of rules to follow to solve a certain problem and it is referred specifically to the solution to find an optimal military schedule for logistics (and has more or less the same meaning as linear programming or linear optimization). These rules can of course be coded and executed by a computer but can be easily followed on paper for instance. Dynamic programming is better thought of as an optimization approach rather than a method or framework where a complex optimization problem is transformed into a sequence of smaller (and simpler) problems. The very essence of DP is its multi-stage optimization procedure. DP does not provide directly with the instruction on how to solve a particular problem, but instead provides a general framework that requires creativity and non-trivial effort/insights so that a problem formulation can be adapted and casted within the DP framework bounds. This is possibly the reason why DP is considered a rather hard topic and it is particularly feared during interviews.

This chapter is not intended to be a full treatment of DP, and we will introduce and describe it to the level that is necessary to understand and better tackle DP interview problems. For a more comprehensive material on DP please refer to [1, 4].

The gist of the DP approach is that we aim at breaking down a problem into simpler sub-problems recursively. If it is possible to do so, then the problem at hand is said to have the **optimal substructure** property i.e. it can be solved by using optimal solution to subproblems. But having the optimal substructure property alone is not enough to prefer a DP approach to another when trying to solve the same problem. This is because DP really shines when a problem also exposes the **overlapping subproblems** property i.e. when the subproblems are reused several times. A classic example is the Fibonacci Sequence. In order to calculate $F(n)$ we need to solve two subproblems: $F(n-1)$ and $F(n-2)$ and adding them up. But for solving $F(n-1)$ we need to solve $F(n-2)$ **again**. The value for the subproblem $F(n-2)$ is thus reused and this makes the Fibonacci problem expose the optimal substructure property. Dynamic programming takes care of this fact by making sure of solving each subproblem only once. Usually this can be achieved in two ways:

Top-down This is usually the easiest of the two, by being a direct derivation from the recursive formulation of the problem. If the problem can be formulated recursively in terms of solution then solution to subproblems can be *memoized*^① in a cache. When a subproblem is reused then the (potentially expensive) recursive call is avoided and

^①From the latin word *memorandum* which means to be remembered. It is basically a way of remembering the result of a function for a certain set of inputs by storing it in a cache.

the cached result is returned instead.

Bottom-up We can try to reformulate the problem by twisting and massaging the recursive formulation so that the subproblems are solved first (thus effectively removing the recursion) and build the solution to the bigger problem from the bottom. This is usually done by working in a sort of tabular form where entries of the table for larger problems are filled by using entries for solution to smaller problems that we have already solved. For instance, when solving the problem of finding the 10th Fibonacci number $F(10)$, we can start from the known values for $F(0)$ and $F(1)$ and working our way up to $F(2)$ by using $F(1)$ and $F(2)$. Once $F(2)$ is ready we can move up to $F(3)$, and so on when we have the values for $F(8)$ and $F(9)$ we proceed with calculating $F(10)$.

DP has found application in many field of science such as Control theory, Bioinformatics AI and operations research. There are a number of problems in computer science that can be solved by using DP such as the

- Longest Common (or increasing) Subsequence
- Weighted Interval Scheduling
- Chain Matrix Multiplication
- Subset sub
- String edit distance
- Coin change
- 0/1 knapsack problem
- Graph shortest path

In the next section we will shortly review a number of DP problem focusing on the key ideas that allow a problem to be approached and solved using DP.

Fibonacci Sequence

Computing the n^{th} number of the Fibonacci sequence is probably one of the most common introductory example of DP. The Fibonacci sequence recursive formulation is ready to be solved using a top-down DP approach. Listing 64.1 shows a C++ function that calculated the n^{th} Fibonacci number.

```
1 unsigned F(const unsigned n)
2 {
3     assert(n >= 0);
4     if (n <= 1)
5         return n;
6
7     return F(n - 1) + F(n - 2);
8 }
```

Listing 63.1: Canonical recursive C++ implementation of a function returning the n^{th} Fibonacci number.

Notice that for instance when $F(6)$ a call tree is produced where the same call is repeated more than once as shown in the list below. $F(2)$ has been calculated 5 times!

- $F(6) = F(5) + F(4)$
- $F(6) = (F(4) + F(3)) + (F(3) + F(2))$
- $F(6) = ((F(3) + F(2)) + (F(2) + F(1))) + ((F(2) + F(1)) + (F(1) + F(0)))$
- $F(6) = (((F(2) + F(1)) + (F(1) + F(0))) + ((F(1) + F(0)) + F(1))) + (((F(1) + F(0)) + F(1)) + (F(1) + F(0)))$
- $F(6) = (((((F(1) + F(0)) + F(1)) + (F(1) + F(0))) + ((F(1) + F(0)) + F(1))) + (((F(1) + F(0)) + F(1)) + (F(1) + F(0))) + (F(1) + F(0)))$

Listing 64.2 can be improved dramatically if we memoize the function calls that have been already calculated. This way no duplicate work is done. W.r.t the previous example, from the second time the value of $F(2)$ is needed, no additional work is done, as the value in the cache is returned.

```

1 using Cache = std::unordered_map<unsigned, unsigned>;
2
3 unsigned F_helper(const unsigned n, Cache& c)
4 {
5     if (n <= 1)
6         return n;
7     if (c.contains(n))
8         return cache[n];
9
10    const auto ans = F(n - 1) + F(n - 2);
11    cache[n] = ans;
12    return ans;
13 }
14
15 unsigned F(const unsigned n)
16 {
17     Cache cache;
18     return F_helper(n, cache);
19 }

```

Listing 63.2: Canonical recursive top-down Dynamic Programming C++ implementation of a function returning the n^{th} Fibonacci number.

Data structures Asymptotic complexity cheatsheet

Data Structure	Time Complexities								Space Complexity
	Average case	Worst case			Access	Search	Insertion	Deletion	Worst case
	Access	Search	Insertion	Deletion					
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$O(1)$	N.A.	$O(1)$	$O(1)$	$O(1)$	N.A.	$O(1)$	$O(1)$	$O(n)$
Queue	$O(1)$	N.A.	$O(1)$	$O(1)$	$O(1)$	N.A.	$O(1)$	$O(1)$	$O(n)$
Singly Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Doubly Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Hash Table	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Red-Black Tree	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(n)$
Heap	$O(1)$	N.A.	$O(\log_2(n))$	$O(\log_2(n))$	$O(1)$	N.A.	$O(\log_2(n))$	$O(\log_2(n))$	$O(n)$

Table 63.1: Asymptotic complexities for a number of data structures. For time, both the average and case is reported, while for space only the worst. $O(1) < O(\log_2(n)) < O(\log_2(n)) < O(n) < O(n \log_2(n)) < O(n^2) < O(n^3) \dots < O(2^n) < O(n!) < O(n^n)$; See Figure 64.1

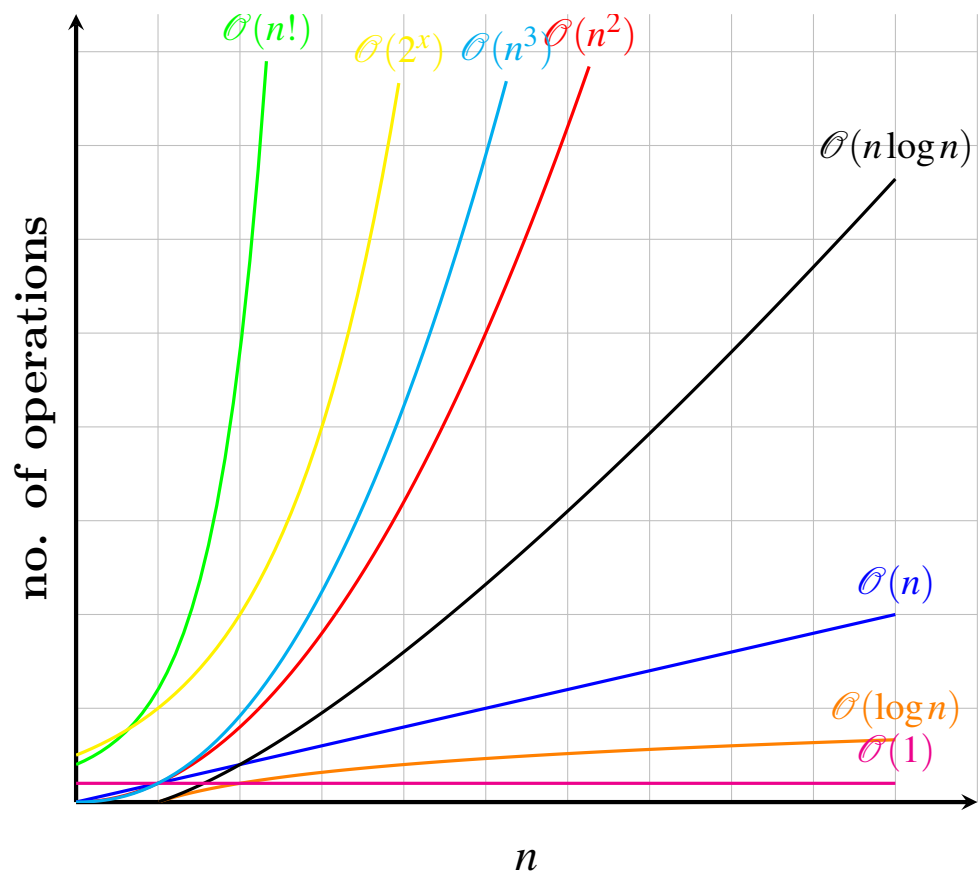


Figure 63.1: Graph showing the relative growth rates of common function used to describe algorithms.

Latencies Reference

Operation	Latency			Notes
	<i>nano</i>	<i>micro</i>	<i>milli</i>	
<i>L1 cache reference</i>	0.5	0.000500000	0.000000500	14 \times L1 cache
<i>Branch mispredict</i>	5	0.005000000	0.000005000	
<i>L2 cache reference</i>	7	0.007000000	0.000007000	
<i>Mutex lock/unlock</i>	25	0.025000000	0.000025000	
<i>Main Memory Reference</i>	100	0.100000000	0.000100000	20 times L2 cache. 200x L1
<i>Compress 1K bytes with Zippy</i>	3000	3.000000000	0.003000000	
<i>Send 1K bytes over 1 Gbps network</i>	10000	10.000000000	0.010000000	
<i>Read 4K randomly from SSD*</i>	150000	150.000000000	0.150000000	~1GB/sec SSD
<i>Round trip within same datacenter</i>	500000	500.000000000	0.500000000	
<i>Read 1 MB sequentially from SSD*</i>	1000000	1000.000000000	1.000000000	~1GB/sec SSD, 4X memory
<i>Disk seek</i>	10000000	10000.000000000	10.000000000	20x datacenter roundtrip
<i>Read 1 MB sequentially from disk</i>	20000000	20000.000000000	20.000000000	80x memory, 20X SSD
<i>Send packet CA->Netherlands->CA</i>	150000000	150000.000000000	150.000000000	

Table 63.2: Latency Comparison Numbers (~2012). Credit to <https://gist.github.com/jboner/2841832>

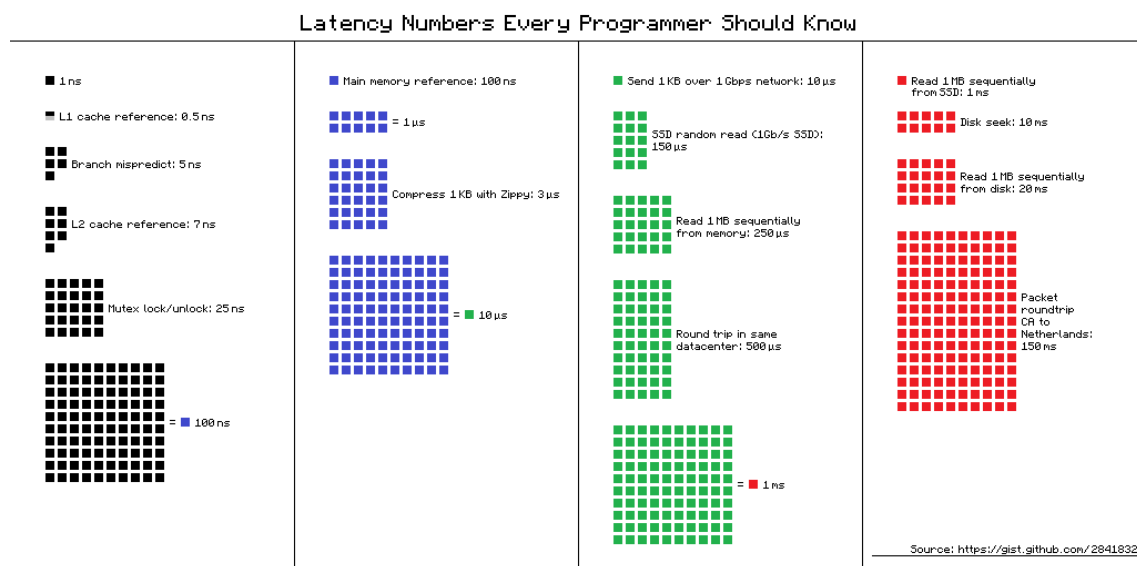


Figure 63.2: Humanized visualization of the data in Table 64.2

Listings

```
1  template <typename SeedType, typename T, typename... Rest>
2  void hash_combine(SeedType& seed, const T& v, const Rest&... rest)
3  {
4      seed ^= std::hash<T>{}(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
5      (hash_combine(seed, rest), ...);
6  }
7  struct PairHasher
8      : public std::unary_function<std::pair<int, int>, std::size_t>
9  {
10     std::size_t operator()(const std::pair<int, int>& k) const
11     {
12         size_t seed = 0;
13         hash_combine(seed, std::get<0>(k), std::get<1>(k));
14         return seed;
15     }
16 };
```

Listing 63.3: Functor used to calculate the hash value for a pair of integers. `hash_combine` is a free function used to mix several input hash values into a new one.

- [1] Richard Bellman. “The theory of dynamic programming”. In: *Bull. Amer. Math. Soc.* 60.6 (Nov. 1954), pages 503–515. URL: <https://projecteuclid.org:443/euclid.bams/1183519147> (cited on pages 54, 381).
- [2] Robert S. Boyer and J. Strother Moore. “MJRTY—A Fast Majority Vote Algorithm”. In: *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Edited by Robert S. Boyer. Dordrecht: Springer Netherlands, 1991, pages 105–117. ISBN: 978-94-011-3488-0. DOI: 10.1007/978-94-011-3488-0_5. URL: https://doi.org/10.1007/978-94-011-3488-0_5 (cited on page 143).
- [3] C++ commitee. *Technical Report on Performance*. [Online]. 2021. URL: <http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf> (cited on page 65).
- [4] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844 (cited on page 381).
- [5] Dov Harel. “A Linear Time Algorithm for the Lowest Common Ancestors Problem”. In: *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*. SFCS ’80. USA: IEEE Computer Society, 1980, pages 308–319. DOI: 10.1109/SFCS.1980.6. URL: <https://doi.org/10.1109/SFCS.1980.6> (cited on page 219).
- [6] Dov Harel and Robert Endre Tarjan. “Fast Algorithms for Finding Nearest Common Ancestors”. In: *SIAM J. Comput.* 13.2 (May 1984), pages 338–355. ISSN: 0097-5397. DOI: 10.1137/0213024. URL: <https://doi.org/10.1137/0213024> (cited on page 219).
- [7] C++ Standard. *max_element*. [Online]. URL: https://en.cppreference.com/w/cpp/algorithm/max_element (cited on page 29).
- [8] C++ Standard. *Random number generation in C++*. [Online]. URL: <https://it.cppreference.com/w/cpp/numeric/random> (cited on page 109).
- [9] C++ Standard. *std::adjacent_find*. [Online]. URL: https://en.cppreference.com/w/cpp/algorithm/adjacent_find (cited on pages 69, 70).
- [10] C++ Standard. *std::rotate*. [Online]. URL: <https://en.cppreference.com/w/cpp/algorithm/rotate> (cited on page 83).
- [11] C++ Standard. *std::enable_if*. [Online]. URL: https://en.cppreference.com/w/cpp/types/enable_if (cited on page 138).
- [12] C++ Standard. *Undefined Behavior*. [Online]. URL: <https://en.cppreference.com/w/cpp/language/ub> (cited on page 49).
- [13] GMP Website. *The GNU Multiple Precision Arithmetic Library*. [Online]. Apr. 2021. URL: https://en.wikipedia.org/wiki/Erase%E2%80%93remove_idiom (cited on page 49).
- [14] Wikipedia. *C++ remove-erase idiom*. [Online]. URL: https://en.wikipedia.org/wiki/Erase%E2%80%93remove_idiom (cited on page 103).

- [15] Wikipedia. *Radix-sort*. [Online]. URL: https://en.wikipedia.org/wiki/Radix_sort.
- [16] Wikipedia, The Free Encyclopedia. *Binary Search Tree*. [Online]. 2020. URL: https://en.wikipedia.org/wiki/Binary_search_tree (cited on page 90).
- [17] Wikipedia, The Free Encyclopedia. *Polar Coordinate System*. [Online]. 2020. URL: https://en.wikipedia.org/wiki/Polar_coordinate_system (cited on page 106).
- [18] Wikipedia, The Free Encyclopedia. *Short Circuit Evaluation*. [Online]. 2020. URL: http://en.wikipedia.org/w/index.php?title=Estimation_lemma&oldid=375747928 (cited on page 71).

GNU LESSER GENERAL PUBLIC LICENSE

Copyright © 2007 Free Software Foundation, Inc. <https://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License. “The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- (a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- (b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- (a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
- (b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- (a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- (b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
- (c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- (d) Do one of the following:
 - . Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
 - i. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
- (e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- (a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms

of this License.

- (b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.
6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation. If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy’s public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.