The Problem Solver's Guide To Coding

Nhut Nguyen, Ph. D.

February, 2024

The Problem Solver's Guide To Coding

First edition. February, 2024.

ISBN 9788797517413 (PDF)

Copyright © 2024 Nhut Nguyen.

All rights reserved.

www.nhutnguyen.com

To my dearest mother, Nguyen Thi Kim Sa.

PREFACE

Welcome to *The Problem Solver's Guide To Coding*, a comprehensive journey to master problem-solving, data structures, algorithms, and programming techniques using C++. This book results from my experiences, challenges, failures and successes in my programming career.

One of the most critical phases in software developer hiring process is the **coding interview**, a rigorous process where candidates are tested on their problem-solving skills and technical proficiency. Companies like FAANG (Facebook, Amazon, Apple, Netflix, and Google) often subject candidates to three to four rounds of interviews, making preparation essential for success.

My journey with coding interviews began during a pivotal career change in the summer of 2021. During this transitional period, I discovered LeetCode, a platform that soon became my daily companion in honing my programming skills. What started as a means to practice new languages (Golang and C#) quickly evolved into a deep exploration of my strongest language, C++.

One day, I decided to write an article for each challenge and share it on my blog leetsolve.com. Over time, my daily practice transformed into more than 70 challenges, each accompanied by a detailed article encompassing problem statements, various approaches, C++ code implementations, thorough explanations, and complexity analyses.

As I delved into these coding challenges, I realized their potential to empower aspiring programmers, students, and junior developers **to excel in interviews and master problem-solving and algorithmic thinking**.

Overview of the book

The Problem Solver's Guide to Coding presents challenges covering **fundamental** data structures, algorithms, and mathematical problems. Challenges are grouped in topics, starting with the simplest data structure - Array. Most are arranged in order of increasing difficulty, but you can pick any chapter or any challenge to start since I write each independently to the other.

Challenges in this book are curated from LeetCode.com, focusing on those that are **not difficult** but provide valuable learning experiences. You might encounter some simple challenges I go directly to the code without saying much about the idea (intuition) since their solution is straightforward.

I also keep the problems' original constraints (inputs' size, limits, etc.) as the code in this book is the ones I submitted on Leetcode.com. It explains why I usually focus on the core algorithm and do not consider/handle corner cases or invalid inputs.

The problems in each chapter comes with a detailed solution, explaining the logic behind the solution and how to implement it in C++, my strongest programming language.

At the end of some problems, I also provide similar problems on leetcode.com for you to solve on your own, as practicing is essential for reinforcing understanding and mastery of the concepts presented in the book. By engaging in problem-solving exercises, you can apply what you have learned, develop your problem-solving skills, and gain confidence in your ability to tackle real-world challenges.

In this book, I focus on readable code rather than optimal one, as most of you are at the beginner level. Some of my solutions might need to be in better runtime or memory. But I keep my code in my style or coding convention, where readability is vital.

Moreover, my weekly sharing of articles with various developer communities has refined the content and established a connection with a diverse group of programming enthusiasts.

Who is this book for?

This book is tailored to benefit a wide audience, from **students** beginning their programming journey to **experienced developers** looking to enhance their skills. Regardless of your experience level, whether you're **preparing for coding interviews** or simply seeking to **improve your problem-solving abilities**, this book is designed to meet your needs.

As a minimum requirement, you are supposed to have some basic background in C++ programming language, data structures and algorithms like a second-year undergraduate in Computer Science.

What sets this book apart is its focus on practicality. The challenges presented here are not just exercises; they mirror **real coding interviews** from top companies like FAANG.

As you work through the coding challenges in this book, you'll learn new skills, improve your problem-solving abilities, and develop your confidence as a programmer.

Acknowledgement

I am deeply grateful to my invaluable previewers, especially Alexander Volkodav, Phung Phu Tai, Le Nhat Tung, Tran Anh Tuan A, Bui Nguyen Cong Duy, Cao Minh Thinh, Nguyen Trung Tuan, Nguyen Tuan Hung, Nguyen Hoang Nhat Minh, Nhan Nguyen, Phan Dinh Thai and Nguyen Quang Trung, whose dedication and insights have shaped this book into its final form. Your contributions and unwavering support are truly appreciated.

I would like to express my heartfelt gratitude to Ninh Pham, Hoang Thanh Lam, Dinh Thai Minh Tam and Tran Anh Tuan B, whose invaluable feedback contributed to the refinement of this book in future versions. Your insightful comments and constructive criticism have played a pivotal role in shaping its content and enhancing its quality. Thank you for your dedication and generosity in sharing your expertise. Your input will undoubtedly pave the way for future improvements and iterations of this work.

Students and developers! By immersing yourself in the challenges and insights shared in this book, you will not only prepare for coding interviews but also cultivate a mindset beyond the scope of a job interview. You will become a problem solver, a strategic thinker, and a proficient C++ programmer.

As you embark on this journey, remember that every challenge you encounter is an opportunity for growth. Embrace the complexities, learn from each solution, and let the knowledge you gain propel you to new heights in your programming career.

Thank you for joining me on this expedition.

May your code be elegant, your algorithms efficient, and your programming journey genuinely transformative.

Happy coding!

Copenhagen, February 2024.

Nhut Nguyen, Ph.D.

CONTENTS

1 Introduction **1**
- 1.1 Why LeetCode? 1
- 1.2 A brief about algorithm complexity 2
- 1.3 Why readable code? 4

2 Array **7**
- 2.1 Transpose Matrix 8
- 2.2 Valid Mountain Array 11
- 2.3 Shift 2D Grid 14
- 2.4 Find All Numbers Disappeared in an Array 19
- 2.5 Rotate Image 24
- 2.6 Spiral Matrix II 27
- 2.7 Daily Temperatures 31

3 Linked List **39**
- 3.1 Merge Two Sorted Lists 40
- 3.2 Remove Linked List Elements 44
- 3.3 Intersection of Two Linked Lists 51
- 3.4 Swap Nodes in Pairs 60
- 3.5 Add Two Numbers 65

4 Hash Table **71**
- 4.1 Roman to Integer 71
- 4.2 Maximum Erasure Value 75
- 4.3 Find and Replace Pattern 79

5 String 83
5.1 Valid Anagram . . . 84
5.2 Detect Capital . . . 89
5.3 Unique Morse Code Words . . . 92
5.4 Unique Email Addresses . . . 95
5.5 Longest Substring Without Repeating Characters . . . 101
5.6 Compare Version Numbers . . . 105

6 Stack 111
6.1 Baseball Game . . . 112
6.2 Valid Parentheses . . . 116
6.3 Backspace String Compare . . . 119
6.4 Remove All Adjacent Duplicates in String II . . . 123

7 Priority Queue (Heap) 129
7.1 Last Stone Weight . . . 130
7.2 Kth Largest Element in a Stream . . . 132
7.3 Kth Smallest Element in a Sorted Matrix . . . 138
7.4 Construct Target Array With Multiple Sums . . . 143

8 Bit Manipulation 151
8.1 Hamming Distance . . . 152
8.2 Power of Four . . . 154
8.3 Find the Duplicate Number . . . 159
8.4 Maximum Product of Word Lengths . . . 164

9 Sorting 171
9.1 Majority Element . . . 172
9.2 Merge Sorted Array . . . 176
9.3 Remove Covered Intervals . . . 182
9.4 My Calendar I . . . 188
9.5 Remove Duplicates from Sorted Array II . . . 192

10 Greedy Algorithm 199
10.1 Can Place Flowers . . . 200
10.2 Minimum Deletions to Make Character Frequencies Unique . . . 203
10.3 Wiggle Subsequence . . . 207
10.4 Partitioning Into Minimum Number Of Deci-Binary Numbers . . . 211
10.5 Maximum Units on a Truck . . . 214

11 Dynamic Programming **219**
11.1 Fibonacci Number 220
11.2 Unique Paths 225
11.3 Largest Divisible Subset 232
11.4 Triangle 240
11.5 Unique Paths II 245

12 Counting **249**
12.1 Single Number 250
12.2 First Unique Character in a String 254
12.3 Max Number of K-Sum Pairs 258

13 Prefix Sums **263**
13.1 Running Sum of 1d Array 264
13.2 Maximum Subarray 268
13.3 Product of Array Except Self 271
13.4 Subarray Sum Equals K 276

14 Two Pointers **285**
14.1 Middle of the Linked List 286
14.2 Linked List Cycle 291
14.3 Sort Array By Parity II 298
14.4 Container With Most Water 304
14.5 Remove Nth Node From End of List 309
14.6 Shortest Unsorted Continuous Subarray 315

15 Mathematics **321**
15.1 Excel Sheet Column Number 322
15.2 Power of Three 326
15.3 Best Time to Buy and Sell Stock 329
15.4 Subsets 335
15.5 Minimum Moves to Equal Array Elements II 338
15.6 Array Nesting 343
15.7 Count Sorted Vowel Strings 347
15.8 Concatenation of Consecutive Binary Numbers 353
15.9 Perfect Squares 356

16 Conclusion **365**

A Coding challenge best practices **367**
A.1 Read the problem carefully 367
A.2 Plan and pseudocode 367
A.3 Test your code 367
A.4 Optimize for time and space complexity 368
A.5 Write clean, readable code 368
A.6 Submit your code and learn from feedback 368
A.7 Keep practicing 368

CHAPTER ONE

INTRODUCTION

1.1 Why LeetCode?

Coding challenges are a great way to practice problem-solving, algorithm development, and logical thinking. They showcase your creativity and innovation while improving your coding techniques. This book offers diverse coding challenges to help you develop your skills.

Coding challenges could be programming puzzles or mathematical problems that require coding solutions. Each challenge requires different coding skills and is designed to challenge and develop a particular set of skills.

The coding challenges in this book are picked from LeetCode. It is a popular online platform for programmers and software engineers that provides many coding challenges and problems. The website was launched in 2015 and has since grown to become one of the go-to resources for coding practice, technical interview preparation, and skills enhancement.

LeetCode offers diverse coding challenges, ranging from easy to hard, covering a wide range of topics such as algorithms, data structures, databases, system design, and more. The problems are created by industry experts and are designed to simulate real-world scenarios, allowing you to gain practical experience in problem-solving.

One feature that makes LeetCode stand out is its extensive discussion forum, where you can interact, share your solutions, and learn from one another. This fosters community and collaboration, as you can receive feedback on their solutions and ask for clarification on difficult problems.

LeetCode also provides premium services like mock interviews with real-world companies, career coaching, and job postings. These premium services are designed to help you prepare for technical interviews, sharpen your skills, and advance your careers.

LeetCode has become a popular resource for technical interview preparation, as many companies use similar problems to screen and evaluate potential candidates. The platform has helped many users to secure job offers from top companies in the technology industry, including Google, Microsoft, and Facebook.

In summary, LeetCode is a valuable resource for programmers and software engineers looking to improve their coding skills, prepare for technical interviews, and advance their careers. Its extensive collection of coding challenges, community discussion forums, and premium services make it an all-in-one platform for coding practice and skills enhancement.

1.2 A brief about algorithm complexity

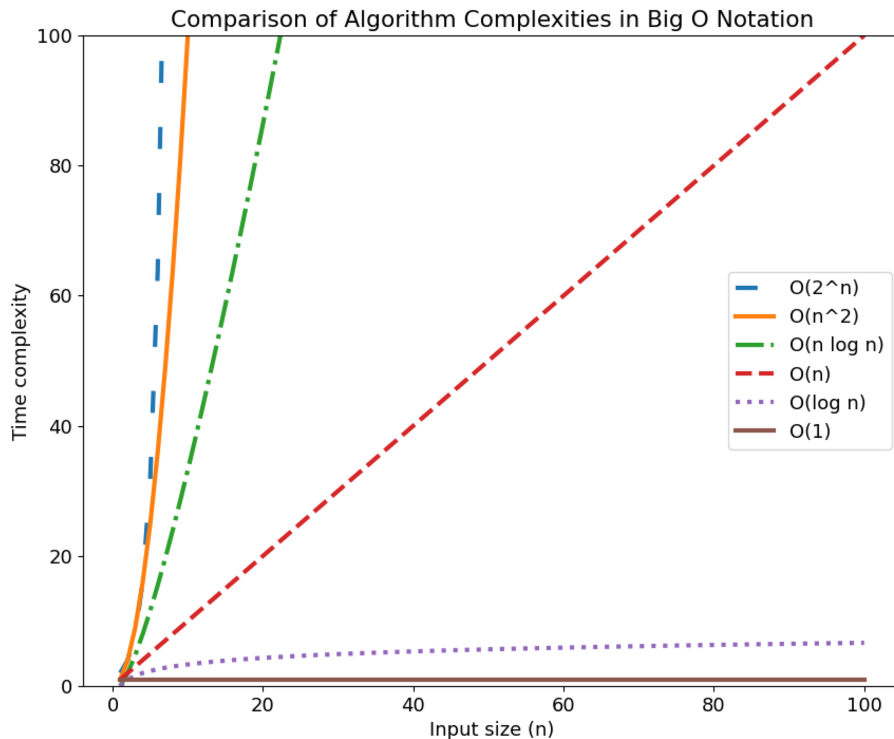
Algorithm complexity, also known as *runtime complexity*, is a measure of how the running time of an algorithm increases as the input size grows. It is an essential concept in computer science, as it helps programmers evaluate and optimize their algorithms' performance.

The complexity of an algorithm is usually measured in terms of its *Big O notation*, which describes the upper bound of the algorithm's running time as a function of the input size. For example, an algorithm with a time complexity of `O(n)` will have a running time proportional to the input size. In contrast, an algorithm with a time complexity of `O(n^2)` will have a running time proportional to the square of the input size.

Algorithm complexity is important because it helps programmers determine their algorithms' efficiency and scalability. In general, algorithms with lower complexity are more efficient, as they require less time and resources to process larger inputs. By analyzing the time complexity of an algorithm, programmers can identify potential performance bottlenecks and optimize their code accordingly.

In addition to time complexity, algorithms may also have space complexity, which measures the memory required to execute the algorithm. Space complexity is also measured in Big O notation and is important for optimizing the memory usage of

an algorithm.



While it is important to optimize the performance of algorithms, it is also important to balance this with readability and maintainability. A highly optimized algorithm may be difficult to understand and maintain, which can lead to problems in the long run. Therefore, it is important to balance performance and readability when designing and implementing algorithms.

In summary, algorithm complexity is an essential concept in computer science that helps programmers evaluate and optimize their algorithms' performance. By analyzing an algorithm's time and space complexity, programmers can identify potential performance bottlenecks and optimize their code to improve efficiency and scalability.

1.3 Why readable code?

Readable code is code that is easy to understand, maintain, and modify. It is an essential aspect of programming, as it ensures that code is accessible to other programmers and helps to prevent errors and bugs. Readable code is important for several reasons.

Firstly, readable code makes it easier for other programmers to understand and modify it. This is particularly important in collaborative projects where multiple programmers work on the same codebase. If the code is not readable, it can lead to confusion and errors, making it difficult for others to work on it.

Secondly, readable code helps to prevent bugs and errors. When code is easy to understand, it is easier to identify and fix potential issues before they become problems. This is important for ensuring the code is reliable and performs as expected.

Thirdly, readable code can improve the overall quality of the codebase. When code is easy to understand, it is easier to identify areas for improvement and make changes to improve the code. This can help improve the codebase's efficiency and maintainability, leading to a better overall product.

Finally, readable code can save time and money. When code is easy to understand, it is easier to maintain and modify. This can help reduce the time and resources required to make changes to the codebase, leading to cost savings in the long run.

In conclusion, readable code is an essential aspect of programming that ensures that code is accessible, error-free, and efficient. By focusing on readability when designing and implementing code, programmers can improve the quality and reliability of their code, leading to a better overall product.

I hope this book is an enjoyable and educational experience that will challenge and inspire you. Whether you want to enhance your skills, prepare for a technical interview, or just have fun, this book has something for you. So, get ready to put your coding skills to the test and embark on a challenging and rewarding journey through the world of coding challenges!

CHAPTER TWO

ARRAY

This chapter will explore the basics of **arrays** - collections of elements organized in a sequence. While they may seem simple, you can learn many concepts and techniques from arrays to improve your coding skills. We'll cover topics like **indexing**, **iteration**, and **manipulation**, as well as dynamic arrays (`std::vector`) and time/space complexity.

Along the way, we'll tackle challenging problems like **searching**, **sorting**, and **subarray** problems, using a structured approach to break down complex tasks into manageable steps.

What this chapter covers:

1. **Fundamentals of Arrays:** Gain a solid understanding of arrays, their properties, and how to access and manipulate elements efficiently.
2. **Array Operations:** Learn essential array operations like insertion, deletion, and updating elements, and understand their trade-offs.
3. **Dynamic Arrays:** Explore dynamic arrays, their advantages over static arrays, and the mechanics of resizing.
4. **Time and Space Complexity:** Grasp the importance of analyzing the efficiency of algorithms and how to evaluate the time and space complexity of array-related operations.
5. **Common Array Algorithms:** Discover classic algorithms such as searching, sorting, and various techniques for tackling subarray problems.
6. **Problem-Solving Strategies:** Develop systematic strategies to approach

array-related challenges, including how to break down problems, devise algorithms, and validate solutions.

2.1 Transpose Matrix

2.1.1 Problem statement

[1]You are given a 2D integer array `matrix`, and your objective is to find the transpose of the given matrix.

The transpose of a matrix involves flipping the matrix over its main diagonal, effectively swapping its row and column indices.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Example 1

```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[1,4,7],[2,5,8],[3,6,9]]
```

Example 2

```
Input: matrix = [[1,2,3],[4,5,6]]
Output: [[1,4],[2,5],[3,6]]
```

[1] https://leetcode.com/problems/transpose-matrix/

Constraints

- `m == matrix.length`.
- `n == matrix[i].length`.
- `1 <= m, n <= 1000`.
- `1 <= m * n <= 10^5`.
- `-10^9 <= matrix[i][j] <= 10^9`.

2.1.2 Solution

Code

```cpp
#include <iostream>
#include <vector>
using namespace std;
vector<vector<int>> transpose(const vector<vector<int>>& matrix) {
    // declare the transposed matrix mt of desired size, i.e.
    // mt's number of rows = matrix's number of columns
    // mt's number of columns = matrix's number of rows
    vector<vector<int>> mt(matrix[0].size(),
                           vector<int>(matrix.size()));
    for (int i = 0; i < mt.size(); i++) {
        for (int j = 0; j < mt[i].size(); j++) {
            mt[i][j] = matrix[j][i];
        }
    }
    return mt;
}
void printResult(const vector<vector<int>>& matrix) {
    cout << "[";
    for (auto& row : matrix) {
        cout << "[";
        for (int m : row) {
            cout << m << ",";
```

(continues on next page)

(continued from previous page)

```
        }
        cout << "]";
    }
    cout << "]\n";
}
int main() {
    vector<vector<int>> matrix = {{1,2,3},{4,5,6},{7,8,9}};
    auto result = transpose(matrix);
    printResult(result);
    matrix = {{1,2,3},{4,5,6}};
    result = transpose(matrix);
    printResult(result);
}
```

```
Output:
[[1,4,7,][2,5,8,][3,6,9,]]
[[1,4,][2,5,][3,6,]]
```

Complexity

- Runtime: `O(m*n)`, where `m = matrix.length` and `n = matrix[i].length`.
- Extra space: `O(1)`.

2.1.3 Implementation note

Note that the matrix might not be square, you cannot just swap the elements using for example the function `std::swap`.

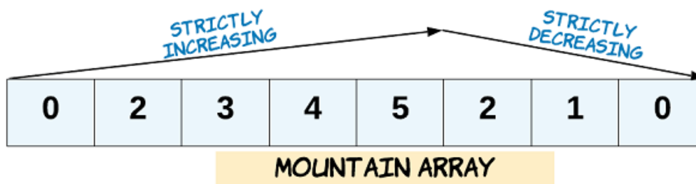
2.2 Valid Mountain Array

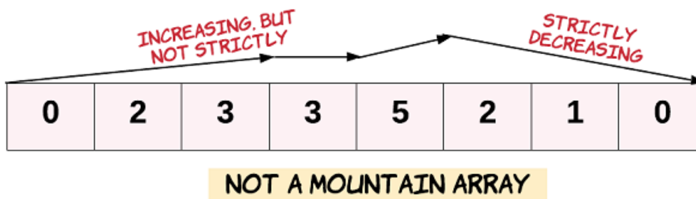
2.2.1 Problem statement

[1]You are given an array of integers `arr`, and your task is to determine whether it is a valid *mountain array*.

A valid *mountain array* must meet the following conditions:

1. The length of `arr` should be greater than or equal to `3`.
2. There should exist an index `i` such that `0 < i < arr.length - 1`, and the elements up to `i` (`arr[0]` to `arr[i]`) should be in strictly ascending order, while the elements starting from `i` (`arr[i]` to `arr[arr.length-1]`) should be in strictly descending order.





Example 1

```
Input: arr = [2,1]
Output: false
```

[1] https://leetcode.com/problems/valid-mountain-array/

Example 2

```
Input: arr = [3,5,5]
Output: false
```

Example 3

```
Input: arr = [0,3,2,1]
Output: true
```

Constraints

- `1 <= arr.length <= 10^4.`
- `0 <= arr[i] <= 10^4.`

2.2.2 Solution

Following the conditions, we have the following implementation.

Code

```
#include <vector>
#include <iostream>
using namespace std;
bool validMountainArray(const vector<int>& arr) {
    if (arr.size() < 3) {
        return false;
    }
    const int N = arr.size() - 1;
    int i = 0;
    // find the top of the mountain
    while (i < N && arr[i] < arr[i + 1]) {
        i++;
```

(continues on next page)

(continued from previous page)

```
    }
    // condition: 0 < i < N - 1
    if (i == 0 || i == N) {
        return false;
    }
    // going from the top to the bottom
    while (i < N && arr[i] > arr[i + 1]) {
        i++;
    }
    return i == N;
}
int main() {
    vector<int> arr{2,1};
    cout << validMountainArray(arr) << endl;
    arr = {3,5,5};
    cout << validMountainArray(arr) << endl;
    arr = {0,3,2,1};
    cout << validMountainArray(arr) << endl;
    arr = {9,8,7,6,5,4,3,2,1,0};
    cout << validMountainArray(arr) << endl;
}
```

```
Output:
0
0
1
0
```

This solution iteratively checks for the two slopes of a mountain array, ensuring that the elements to the left are strictly increasing and the elements to the right are strictly decreasing. If both conditions are met, the function returns `true`, indicating that the input array is a valid mountain array; otherwise, it returns `false`.

Complexity

- Runtime: `O(N)`, where `N = arr.length`.
- Extra space: `O(1)`.

2.2.3 Coding best practices

Breaking down the problem into distinct stages, like finding the peak of the mountain and then traversing down from there, can simplify the logic and improve code readability. This approach facilitates a clear understanding of the algorithm's progression and helps in handling complex conditions effectively.

2.2.4 Exercise

- Beautiful Towers I

2.3 Shift 2D Grid

2.3.1 Problem statement

[1]You are given a 2D `grid` with dimension `mxn` and an integer `k`. Your task is to perform `k` shift operations on the grid.

In each shift operation:

- The element at `grid[i][j]` moves to `grid[i][j+1]`.
- The element at `grid[i][n-1]` moves to `grid[i+1][0]`.
- The element at `grid[m-1][n-1]` moves to `grid[0][0]`.

After performing `k` shift operations, return the updated 2D grid.

[1] https://leetcode.com/problems/shift-2d-grid/

Example 1

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix} \longrightarrow \begin{bmatrix}9 & 1 & 2\\3 & 4 & 5\\6 & 7 & 8\end{bmatrix}$$

```
Input: grid = [[1,2,3],[4,5,6],[7,8,9]], k = 1
Output: [[9,1,2],[3,4,5],[6,7,8]]
```

Example 2

$$\begin{bmatrix}3 & 8 & 1 & 9\\19 & 7 & 2 & 5\\4 & 6 & 11 & 10\\12 & 0 & 21 & 13\end{bmatrix} \to \begin{bmatrix}13 & 3 & 8 & 1\\9 & 19 & 7 & 2\\5 & 4 & 6 & 11\\10 & 12 & 0 & 21\end{bmatrix} \to \begin{bmatrix}21 & 13 & 3 & 8\\1 & 9 & 19 & 7\\2 & 5 & 4 & 6\\11 & 10 & 12 & 0\end{bmatrix}$$

$$\to \begin{bmatrix}0 & 21 & 13 & 3\\8 & 1 & 9 & 19\\7 & 2 & 5 & 4\\6 & 11 & 10 & 12\end{bmatrix} \to \begin{bmatrix}12 & 0 & 21 & 13\\3 & 8 & 1 & 9\\19 & 7 & 2 & 5\\4 & 6 & 11 & 10\end{bmatrix}$$

```
Input: grid = [[3,8,1,9],[19,7,2,5],[4,6,11,10],[12,0,21,13]], k = 4
Output: [[12,0,21,13],[3,8,1,9],[19,7,2,5],[4,6,11,10]]
```

Example 3

```
Input: grid = [[1,2,3],[4,5,6],[7,8,9]], k = 9
Output: [[1,2,3],[4,5,6],[7,8,9]]
```

Constraints

- `1 <= grid.length <= 50`.
- `1 <= grid[i].length <= 50`.
- `-1000 <= grid[i][j] <= 1000`.

- `0 <= k <= 100`.

2.3.2 Solution: Convert a 2D array into a 1D one

You can convert the 2D `grid` into a 1D vector `v` to perform the shifting easier. One way of doing this is concatenating the rows of the matrix.

- If you shift the grid `k = i*N` times where `N = v.size()` and `i` is any non-negative integer, you go back to the original grid; i.e. you did not shift it.
- If you shift the grid `k` times with `0 < k < N`, the first element of the result starts from `v[N-k]`.
- In general, the first element of the result starts from `v[N - k%N]`.

Example 1

For `grid = [[1,2,3],[4,5,6],[7,8,9]]`:

- It can be converted into a 1D vector `v = [1,2,3,4,5,6,7,8,9]` of size `m*n = 9`.
- With `k = 1` the shifted `grid` now starts from `v[9-1] = 9`.
- The final result is `grid = [[9,1,2][3,4,5][6,7,8]]`.

Code

```
#include <vector>
#include <iostream>
using namespace std;
vector<vector<int>> shiftGrid(vector<vector<int>>& grid, int k) {
    vector<int> v;
    // store the 2D grid values into a 1D vector v
    for (auto& r : grid) {
        v.insert(v.end(), r.begin(), r.end());
    }
    const int N = v.size();
```

(continues on next page)

(continued from previous page)

```
    // perform the shifting
    int p = N - k % N;

    // number of rows
    const int m = grid.size();

    // number of columns
    const int n = grid[0].size();

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (p == N) {
                p = 0;
            }
            // reconstruct the grid
            grid[i][j] = v[p++];
        }
    }
    return grid;
}
void printResult(const vector<vector<int>>& grid) {
    cout << "[";
    for (auto& r : grid) {
        cout << "[";
        for (int a: r) {
            cout << a << ",";
        }
        cout << "]";
    }
    cout << "]\n";
}
int main() {
    vector<vector<int>> grid{{1,2,3},{4,5,6},{7,8,9}};
    auto result = shiftGrid(grid, 1);
    printResult(result);
```

(continues on next page)

(continued from previous page)

```
    grid = {{3,8,1,9},{19,7,2,5},{4,6,11,10},{12,0,21,13}};
    result = shiftGrid(grid, 4);
    printResult(result);
    grid = {{1,2,3},{4,5,6},{7,8,9}};
    result = shiftGrid(grid, 9);
    printResult(result);
}
```

```
Output:
[[9,1,2,][3,4,5,][6,7,8,]]
[[12,0,21,13,][3,8,1,9,][19,7,2,5,][4,6,11,10,]]
[[1,2,3,][4,5,6,][7,8,9,]]
```

This solution flattens the 2D `grid` into a 1D vector `v`, representing the grid's elements in a linear sequence. Then, by calculating the new position for each element after the shift operation, it reconstructs the grid by placing the elements back into their respective positions based on the calculated indices. This approach avoids unnecessary copying or shifting of elements within the grid, optimizing both memory and time complexity.

Complexity

- Runtime: `O(m*n)` (the nested `for` loops), where `m = grid.length` and `n = grid[i].length`.
- Extra space: `O(m*n)` (the vector `v`).

2.3.3 Key takeaway

1. To convert a 2D matrix into a 1D vector, you can use the `std::vector`'s function `insert()`.
2. The modulo operator `%` is usually used to ensure the index is inbound.

2.4 Find All Numbers Disappeared in an Array

2.4.1 Problem statement

[1]You are given an array `nums` of `n` integers, where each integer `nums[i]` falls within the range `[1, n]`. Your task is to find and return an array containing all the integers in the range `[1, n]` that are not present in the given array `nums`.

Example 1

```
Input: nums = [4,3,2,7,8,2,3,1]
Output: [5,6]
```

Example 2

```
Input: nums = [1,1]
Output: [2]
```

Constraints

- `n == nums.length`.
- `1 <= n <= 10^5`.
- `1 <= nums[i] <= n`.

Follow up

Can you solve the problem without using additional memory and achieve a linear runtime complexity? You can assume that the list you return does not count as extra space.

[1] https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/

2.4.2 Solution 1: Marking the appearances by bool

You can use a vector of bool to mark which value appeared in the array.

Code

```
#include <vector>
#include <iostream>
using namespace std;
vector<int> findDisappearedNumbers(const vector<int>& nums) {

    const int n = nums.size();
    vector<bool> exist(n + 1, false);
    for (auto& a : nums) {
        exist[a] = true;
    }
    vector<int> result;
    for (int i = 1; i <= n; i++) {
        if (!exist.at(i)) {
            result.push_back(i);
        }
    }
    return result;
}
void print(const vector<int>& nums) {
    cout << "[";
    for (auto& a : nums) {
        cout << a << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> nums = {4,3,2,7,8,2,3,1};
    auto result = findDisappearedNumbers(nums);
    print(result);
    nums = {1,1};
    result = findDisappearedNumbers(nums);
```

(continues on next page)

(continued from previous page)

```
    print(result);
}
```

```
Output:
[5,6,]
[2,]
```

This code declares a vector named `exist` of type `bool` and initializes all of its values to `false`. Its size is declared as `n + 1` where `n = nums.size()` so it can mark the values ranged from `1` to `n`.

Then it performs the marking of all `nums`'s elements to `true`. The ones that are `false` will belong to the result.

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: much less than `O(n)`. `vector<bool>` is optimized for space efficiency; it stores single bits.

2.4.3 Solution 2: Marking the appearances by sign

You could use the indices of the array `nums` to mark the appearances of its elements because they are just a shift (`[1, n]` vs. `[0, n-1]`).

One way of marking the appearance of a value `j` (`1 <= j <= n`) is making the element `nums[j-1]` to be negative. Then the indices `j`'s whose `nums[j-1]` are still positive are the ones that do not appear in `nums`.

Example 1

With `nums = [4,3,2,7,8,2,3,1]`:

- To indicate `4` is present, make `nums[4-1]` is negative, i.e. changing `nums[4-1] = nums[3]` to `-7`.
- To indicate `3` is present, make `nums[3-1]` is negative, i.e. changing `nums[3-1] = nums[2]` to `-2`.
- And so on.
- `nums` becomes `[-4,-3,-2,-7,8,2,-3,-1]`.
- The positive values `8` corresponds to `nums[4] = nums[5-1]`, indicates `5` was not present in `nums`.
- Similarly, the positive values `2` corresponds to `nums[5] = nums[6-1]`, indicates `6` was not present in `nums`.

Code

```cpp
#include <vector>
#include <iostream>
using namespace std;
vector<int> findDisappearedNumbers(vector<int>& nums) {
    const int n = nums.size();
    int j;
    for (int i{0}; i < n; i++) {
        // make sure j is positive since nums[i] might be
        // changed to be negative in previous steps
        j = abs(nums.at(i));

        // Mark nums[j - 1] as negative to indicate its presence
        nums[j - 1] = -abs(nums.at(j - 1));
    }
    vector<int> result;
    for (int j{1}; j <= n; j++) {
        // If nums[j - 1] is positive, it means j is missing
        if (nums.at(j - 1) > 0) {
```

(continues on next page)

(continued from previous page)

```
            result.push_back(j);
        }
    }
    return result;
}
void print(const vector<int>& nums) {
    cout << "[";
    for (auto& a : nums) {
        cout << a << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> nums = {4,3,2,7,8,2,3,1};
    auto result = findDisappearedNumbers(nums);
    print(result);
    nums = {1,1};
    result = findDisappearedNumbers(nums);
    print(result);
}
```

```
Output:
[5,6,]
[2,]
```

The key to this solution is that it utilizes the array to mark the presence of numbers. Negating the value at the index corresponding to each number found in the input array effectively marks that number as present. Then, by iterating through the modified array, it identifies the missing numbers by checking which indices still hold positive values.

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(1)` (the returned list does not count as extra space).

2.4.4 Readable code

- Solution 2 helps to avoid allocating extra memory but it is not straightforward to understand.
- Though Solution 1 requires some extra space, that memory is not much since `std::vector<bool>` is optimized for space efficiency. Moreover, it is easier to understand than Solution 2.

2.4.5 Exercise

- Find All Duplicates in an Array

2.5 Rotate Image

2.5.1 Problem statement

[1]Given an `n x n` 2D matrix representing an image, your task is to rotate the image by 90 degrees clockwise. The rotation must be performed in-place, meaning you need to modify the original input 2D matrix directly. It is not allowed to create another 2D matrix for the rotation.

[1] https://leetcode.com/problems/rotate-image/

Example 1

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix} \longrightarrow \begin{bmatrix}7 & 4 & 1\\8 & 5 & 2\\9 & 6 & 3\end{bmatrix}$$

```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[7,4,1],[8,5,2],[9,6,3]]
```

Example 2

$$\begin{bmatrix}5 & 1 & 9 & 11\\2 & 4 & 8 & 10\\13 & 3 & 6 & 7\\15 & 14 & 12 & 16\end{bmatrix} \longrightarrow \begin{bmatrix}15 & 13 & 2 & 5\\14 & 3 & 4 & 1\\12 & 6 & 8 & 9\\16 & 7 & 10 & 11\end{bmatrix}$$

```
Input: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
```

Constraints

- `n == matrix.length == matrix[i].length`.
- `1 <= n <= 20`.
- `-1000 <= matrix[i][j] <= 1000`.

2.5.2 Solution: The math behind

For any square matrix, the rotation 90 degrees clockwise can be performed in two steps:

1. Transpose the matrix.
2. Mirror the matrix vertically.

Code

```
#include <iostream>
#include <vector>
using namespace std;
void rotate(vector<vector<int>>& matrix) {
    const int n = matrix.size();
    // transpose
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
    // vertical mirror
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n / 2; j++ ) {
            swap(matrix[i][j], matrix[i][n - 1 - j]);
        }
    }
}
void printMatrix(const vector<vector<int>>& matrix) {
    cout << "[";
    for (auto& row: matrix) {
        cout << "[";
        for (auto& a: row) {
            cout << a << ",";
        }
        cout << "],";
    }
    cout << "]\n";
}
int main() {
    vector<vector<int>> matrix{{1,2,3},{4,5,6},{7,8,9}};
    rotate(matrix);
    printMatrix(matrix);
    matrix = {{5,1,9,11},{2,4,8,10},{13,3,6,7},{15,14,12,16}};
    rotate(matrix);
```

(continues on next page)

(continued from previous page)

```
    printMatrix(matrix);
}
```

```
Output:
[[7,4,1,],[8,5,2,],[9,6,3,],]
[[15,13,2,5,],[14,3,4,1,],[12,6,8,9,],[16,7,10,11,],]
```

Complexity

- Runtime: `O(n^2)`, where `n = matrix.length`.
- Extra space: `O(1)`.

2.5.3 Implementation tips

1. The function std::swap[2] can be used to exchange two values.
2. When doing the transpose or mirroring, you could visit over only one-half of the matrix.

2.5.4 Exercise

- Determine Whether Matrix Can Be Obtained By Rotation

2.6 Spiral Matrix II

2.6.1 Problem statement

[1]Given a positive integer `n`, generate an `n x n` matrix filled with elements from `1` to `n^2` in spiral order.

[2] https://en.cppreference.com/w/cpp/algorithm/swap
[1] https://leetcode.com/problems/spiral-matrix-ii/

Example 1

```
Input: n = 3
Output: [[1,2,3],[8,9,4],[7,6,5]]
```

Example 2

```
Input: n = 1
Output: [[1]]
```

Constraints

- `1 <= n <= 20`.

2.6.2 Solution

1. Starting from the top left of the matrix.
2. Going along the spiral direction.
3. Put the value to the matrix, starting from `1`.

Code

```
#include <vector>
#include <iostream>
using namespace std;
enum Direction {RIGHT, DOWN, LEFT, UP};
vector<vector<int>> generateMatrix(int n) {
    vector<vector<int>> m(n, vector<int>(n));
    int bottom = n - 1;
    int right = n - 1;
    int top = 0;
    int left = 0;
    int row = 0;
    int col = 0;
    Direction d = RIGHT;
    int a = 1;
    while (top <= bottom && left <= right) {
        m[row][col] = a++;
        switch (d) {
            case RIGHT: if (col == right) {
                            top++;
                            d = DOWN;
                            row++;
                        } else {
                            col++;
                        }
                        break;
            case DOWN:  if (row == bottom) {
                            right--;
                            d = LEFT;
                            col--;
                        } else {
                            row++;
                        }
                        break;
            case LEFT:  if (col == left) {
                            bottom--;
```

(continues on next page)

(continued from previous page)

```
                        d = UP;
                        row--;
                    } else {
                        col--;
                    }
                    break;
            case UP:    if (row == top) {
                        left++;
                        d = RIGHT;
                        col++;
                    } else {
                        row--;
                    }
                    break;
        }
    }
    return m;
}
void printResult(const vector<vector<int>>& m) {
    cout << "[";
    for (auto& r : m) {
        cout << "[";
        for (int a : r) {
            cout << a << ",";
        }
        cout << "]";
    }
    cout << "]\n";
}
int main() {
    auto m = generateMatrix(3);
    printResult(m);
    m = generateMatrix(1);
    printResult(m);
}
```

```
Output:
[[1,2,3,][8,9,4,][7,6,5,]]
[[1,]]
```

This solution uses a `Direction` `enum` and boundary variables to iteratively fill the matrix in a spiral pattern. Updating the direction of movement based on the current position and boundaries efficiently populates the matrix with sequential values, traversing in a clockwise direction from the outer layer to the inner layer.

Complexity

- Runtime: `O(n^2)`, where `n x n` is the size of the matrix.
- Extra space: `O(1)`.

2.6.3 Key Takeaway

Enumerating directions with an enum (like `Direction`) can enhance code readability and maintainability, especially in algorithms involving traversal or movement. It aids in clearly defining and referencing the possible directions within the problem domain.

2.6.4 Exercise

- Spiral Matrix

2.7 Daily Temperatures

2.7.1 Problem statement

[1]You are given an array of integers `temperatures`, which represents the daily temperatures. Your task is to create an array `answer` such that `answer[i]` represents the

[1] https://leetcode.com/problems/daily-temperatures/

number of days you need to wait after the i-th day to experience a warmer temperature. If there is no future day with a warmer temperature, then `answer[i]` should be set to `0`.

Example 1

```
Input: temperatures = [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]
```

Example 2

```
Input: temperatures = [30,40,50,60]
Output: [1,1,1,0]
```

Example 3

```
Input: temperatures = [30,60,90]
Output: [1,1,0]
```

Constraints

- `1 <= temperatures.length <= 10^5`.
- `30 <= temperatures[i] <= 100`.

2.7.2 Solution 1: Starting from the first day

For each `temperatures[i]`, find the closest `temperatures[j]` with `j > i` such that `temperatures[j] > temperatures[i]`, then `answer[i] = j - i`. If not found, `answer[i] = 0`.

Example 1

For `temperatures = [73,74,75,71,69,72,76,73]`:

- `answer[0] = 1` since the next day is warmer (`74 > 73`).
- `answer[1] = 1` since the next day is warmer (`75 > 74`).
- `answer[2] = 4` since only after 4 days it is warmer (`76 > 75`).
- And so on.

Code

```
#include <vector>
#include <iostream>
using namespace std;
vector<int> dailyTemperatures(const vector<int>& temperatures) {
    vector<int> answer(temperatures.size());
    for (int i = 0; i < temperatures.size(); i++) {
        answer[i] = 0;
        for (int j = i + 1; j < temperatures.size(); j++) {
            if (temperatures[j] > temperatures[i]) {
                answer[i] = j - i;
                break;
            }
        }
    }
    return answer;
}
void print(const vector<int>& answer) {
    cout << "[";
    for (auto& v : answer ) {
        cout << v << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> temperatures{73,74,75,71,69,72,76,73};
```

(continues on next page)

(continued from previous page)

```
    auto answer = dailyTemperatures(temperatures);
    print(answer);
    temperatures = {30,40,50,60};
    answer = dailyTemperatures(temperatures);
    print(answer);
    temperatures = {30,60,90};
    answer = dailyTemperatures(temperatures);
    print(answer);
}
```

```
Output:
[1,1,4,2,1,1,0,0,]
[1,1,1,0,]
[1,1,0,]
```

This solution iterates through the `temperatures` array and, for each temperature, iterates through the remaining temperatures to find the next higher temperature. Storing the time difference between the current day and the next higher temperature day constructs the resulting array representing the number of days until warmer temperatures.

Complexity

- Runtime: `O(N^2)`, where `N = temperatures.length`.
- Extra space: `O(1)`.

2.7.3 Solution 2: Starting from the last day

The straightforward solution above is easy to understand, but the complexity is `O(N^2)`.

The way starting from the first day to the last day does not make use of the knowledge of the `answer[i]` values.

- The value `answer[i] > 0` tells you that `temperatures[i + answer[i]]` is the next temperature that is warmer than `temperatures[i]`.

- The value `answer[i] = 0` tells you that there is no warmer temperature than `temperatures[i]`.

When computing `answer[i]` in the reversed order, you can use that knowledge more efficiently.

Suppose you already know the future values `answer[j]`. To compute an older value `answer[i]` with `i < j`, you need only to compare `temperatures[i]` with `temperatures[i + 1]` and its **chain** of warmer temperatures.

Example 1

For `temperatures = [73,74,75,71,69,72,76,73]`.

Suppose you have computed all `answer[j]` with `j > 2`, `answer = [?,?,?,2,1,1,0, 0]`.

To compute `answer[i = 2]` for `temperatures[2] = 75`, you need to compare it with

- `temperatures[3] = 71 (< 75)`. Go to the next warmer temperature than `temperatures[3]`, which is `temperatures[3 + answer[3]] = temperatures[3 + 2]`.
- `temperatures[5] = 72 (< 75)`. Go to the next warmer temperature than `temperatures[5]`, which is `temperatures[5 + answer[5]] = temperatures[5 + 1]`.
- `temperatures[6] = 76 (> 75)`. Stop.
- `answer[i = 2] = j - i = 6 - 2 = 4`.

Code

```
#include <vector>
#include <iostream>
using namespace std;
vector<int> dailyTemperatures(const vector<int>& temperatures) {
    vector<int> answer(temperatures.size(), 0);
    for (int i = temperatures.size() - 2; i >= 0 ; i--) {
        int j = i + 1;
```

(continues on next page)

(continued from previous page)

```
        while (j < temperatures.size() &&
               temperatures[j] <= temperatures[i]) {
            // some temperature is bigger than temperatures[j],
            // go to that value
            if (answer[j] > 0) {
                j += answer[j];
            } else {
                j = temperatures.size();
            }
        }
        if (j < temperatures.size()) {
            answer[i] = j - i;
        }
    }
    return answer;
}
void print(const vector<int>& answer) {
    cout << "[";
    for (auto& v : answer ) {
        cout << v << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> temperatures{73,74,75,71,69,72,76,73};
    auto answer = dailyTemperatures(temperatures);
    print(answer);
    temperatures = {30,40,50,60};
    answer = dailyTemperatures(temperatures);
    print(answer);
    temperatures = {30,60,90};
    answer = dailyTemperatures(temperatures);
    print(answer);
}
```

```
Output:
```

(continues on next page)

(continued from previous page)

```
[1,1,4,2,1,1,0,0,]
[1,1,1,0,]
[1,1,0,]
```

The key to this solution lies in its optimized approach to finding the next higher temperature. It utilizes a `while` loop to traverse the `temperatures` array efficiently, skipping elements if they are not potential candidates for a higher temperature. Updating the index based on previously calculated values stored in the `answer` array avoids unnecessary iterations, resulting in improved performance compared to the straightforward nested loop approach.

This improved solution reduces the time complexity to `O(N)` as it iterates through the `temperatures` vector only once, resulting in a more efficient algorithm for finding the waiting periods for each day.

Complexity

Worse cases for the `while` loop are when most `temperatures[j]` in their chain are cooler than `temperatures[i]`.

In these cases, the resulting `answer[i]` will be either `0` or a big value `j - i`. Those extreme values give you a huge knowledge when computing `answer[i]` for other older days `i`.

The value `0` would help the `while` loop terminates very soon. On the other hand, the big value `j - i` would help the `while` loop skips the days `j` very quickly.

- Runtime: `O(N)`, where `N = temperatures.length`.
- Extra space: `O(1)`.

2.7.4 Tips

In some computations, you could improve the performance by using the knowledge of the results you have computed.

In this particular problem, it can be achieved by doing it in the reversed order.

2.7.5 Exercise

- Next Greater Element I

CHAPTER THREE

LINKED LIST

In this chapter, we'll learn about **linked list** - a unique and dynamic data structure that challenges our understanding of sequential data.

Unlike *arrays*, linked lists do not impose a fixed size or continuous memory block. Rather, they consist of nodes that contain data and a reference to the next node. This seemingly simple concept unlocks many possibilities, from creating efficient insertions and deletions to creatively solving problems that may seem specifically designed for linked list manipulation.

Our exploration of linked lists will encompass a variety of variations and intricacies, including singly linked lists. By delving into these lists, you'll discover how they empower us to tackle problems that may initially appear complicated.

What this chapter covers:

1. **Introduction to Linked Lists:** Gain a comprehensive understanding of linked lists, their advantages, and their role in problem-solving.
2. **Singly Linked Lists:** Explore the mechanics of singly linked lists, mastering the art of traversal, insertion, and deletion.
3. **Advanced Linked List Concepts:** Learn about sentinel nodes, dummy nodes, and techniques to handle common challenges like detecting cycles and reversing lists.
4. **Problem-Solving Strategies:** Develop strategies to approach linked list problems systematically, including strategies for merging lists, detecting intersections, and more.

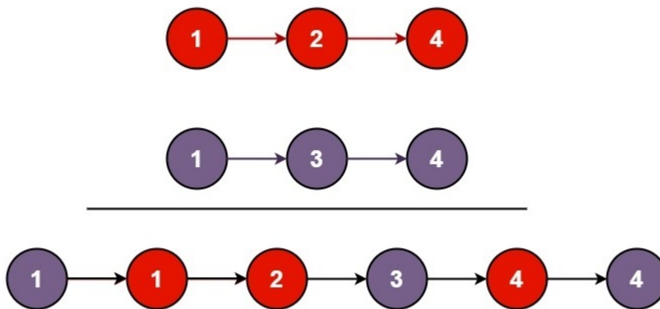
3.1 Merge Two Sorted Lists

3.1.1 Problem statement

[1]Given the starting nodes of two sorted linked lists, `list1` and `list2`, your task is to combine these lists into a single sorted linked list.

This merged list should be created by connecting the nodes from both `list1` and `list2`. Finally, you should return the starting node of the resulting merged linked list.

Example 1



```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Example 2

```
Input: list1 = [], list2 = []
Output: []
```

[1] https://leetcode.com/problems/merge-two-sorted-lists/

Example 3

```
Input: list1 = [], list2 = [0]
Output: [0]
```

Constraints

- The number of nodes in both lists is in the range `[0, 50]`.
- `-100 <= Node.val <= 100`.
- Both `list1` and `list2` are sorted in non-decreasing order.

3.1.2 Solution: Constructing a new list

For each pair of nodes between the two lists, pick the node having smaller value to append to the new list.

Code

```
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
    if (list1 == nullptr) {
        return list2;
    } else if (list2 == nullptr) {
        return list1;
    }
```

(continues on next page)

(continued from previous page)

```
    // identify which list is head of the merged one
    ListNode* head = list1;
    if (list2->val < head->val) {
        head = list2;
        list2 = list2->next;
    } else {
        list1 = list1->next;
    }
    ListNode* node = head;
    while (list1 && list2) {
        // pick the smaller node to append to the new list.
        if (list1->val < list2->val) {
            node->next = list1;
            list1 = list1->next;
        } else {
            node->next = list2;
            list2 = list2->next;
        }
        node = node->next;
    }
    if (list1 == nullptr) {
        node->next = list2;
    } else {
        node->next = list1;
    }
    return head;
}
void printResult(ListNode* head) {
    std::cout << "[";
    while (head) {
        std::cout << head->val << ",";
        head = head->next;
    }
    std::cout << "]\n";
}
int main() {
```

(continues on next page)

(continued from previous page)

```
    ListNode four1(4);
    ListNode two1(2, &four1);
    ListNode one1(1, &two1);
    ListNode four2(4);
    ListNode three2(3, &four2);
    ListNode one2(1, &three2);
    auto newOne = mergeTwoLists(&one1, &one2);
    printResult(newOne);

    auto empty = mergeTwoLists(nullptr, nullptr);
    printResult(empty);

    ListNode zero(0);
    auto z = mergeTwoLists(nullptr, &zero);
    printResult(z);
}
```

```
Output:
[1,1,2,3,4,4,]
[]
[0,]
```

Complexity

- Runtime: `O(N)`, where `N = list1.length + list2.length`.
- Extra space: `O(1)`.

3.1.3 Conclusion

This solution merges two sorted linked lists efficiently without using extra space.

It identifies the head of the merged list by comparing the values of the first nodes of the input lists. Then, it iterates through both lists, linking nodes in ascending order until one list is exhausted.

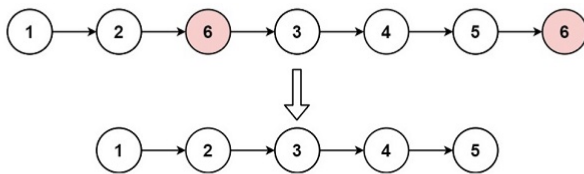
Finally, it appends the remaining nodes from the non-empty list to the merged list, ensuring the resulting list remains sorted.

3.2 Remove Linked List Elements

3.2.1 Problem statement

[1]You are given the starting node, `head`, of a linked list, and an integer `val`. Your task is to eliminate all nodes from the linked list that have a value equal to `val`. After removing these nodes, return the new starting node of the modified linked list.

Example 1




```
Input: head = [1,2,6,3,4,5,6], val = 6
Output: [1,2,3,4,5]
```

Example 2

```
Input: head = [], val = 1
Output: []
```

[1] https://leetcode.com/problems/remove-linked-list-elements/

Example 3

```
Input: head = [7,7,7,7], val = 7
Output: []
```

Constraints

- The number of nodes in the list is in the range `[0, 10^4]`.
- `1 <= Node.val <= 50`.
- `0 <= val <= 50`.

Linked list data structure

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
```

3.2.2 Solution 1: Consider the special case for head

Removing a node `A` in a linked list means instead of connecting the previous node `A.pre` to `A`, you connect `A.pre` to `A.next`.

Code

```
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
```

(continues on next page)

(continued from previous page)

```
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
ListNode* removeElements(ListNode* head, int val) {
    // remove head if its value matches val
    while (head && head->val == val) {
        head = head->next;
    }
    if (head == nullptr) return nullptr;
    ListNode* pre = head;
    while (pre->next) {
        if (pre->next->val == val) {
            // remove pre->next
            pre->next = pre->next->next;
        } else {
            pre = pre->next;
        }
    }
    return head;
}
void print(const ListNode* head) {
    ListNode* node = head;
    std::cout << "[";
    while (node) {
        std::cout << node->val << ",";
        node = node->next;
    }
    std::cout << "]\n";
}
int main() {
    ListNode sixb(6);
    ListNode five(5, &sixb);
    ListNode four(4, &five);
    ListNode three(3, &four);
    ListNode sixa(6, &three);
```

(continues on next page)

(continued from previous page)

```
    ListNode two(2, &sixa);
    ListNode head(1, &two);
    ListNode* newHead = removeElements(&head, 6);
    print(newHead);
    newHead = removeElements(nullptr, 1);
    print(newHead);
    ListNode seven4(7);
    ListNode seven3(7, &seven4);
    ListNode seven2(7, &seven3);
    ListNode seven1(7, &seven2);
    newHead = removeElements(&seven1, 7);
    print(newHead);
}
```

```
Output:
[1,2,3,4,5,]
[]
[]
```

This solution efficiently removes nodes with a specified value `val` from a linked list by using two pointers (`head` and `pre`) to traverse the list and update the `next` pointers to bypass nodes with the specified value.

Complexity

- Runtime: `O(N)`, where `N` is the number of nodes.
- Memory: `O(1)`.

3.2.3 Solution 2: Create a previous dummy node for head

`head` has no `pre`. You can create a dummy node for `head.pre` whose values is out of the contraints.

Code

```cpp
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
ListNode* removeElements(ListNode* head, int val) {

    // create a new head of value 2023 > 50
    ListNode preHead(2023, head);
    ListNode* pre = &preHead;
    while (pre->next) {
        if (pre->next->val == val) {
            pre->next = pre->next->next;
        } else {
            pre = pre->next;
        }
    }
    return preHead.next;
}
void print(const ListNode* head) {
    ListNode* node = head;
    std::cout << "[";
    while (node) {
        std::cout << node->val << ",";
        node = node->next;
    }
    std::cout << "]\n";
}
int main() {
    ListNode sixb(6);
    ListNode five(5, &sixb);
    ListNode four(4, &five);
```

(continues on next page)

(continued from previous page)

```
    ListNode three(3, &four);
    ListNode sixa(6, &three);
    ListNode two(2, &sixa);
    ListNode head(1, &two);
    ListNode* newHead = removeElements(&head, 6);
    print(newHead);

    newHead = removeElements(nullptr, 1);
    print(newHead);

    ListNode seven4(7);
    ListNode seven3(7, &seven4);
    ListNode seven2(7, &seven3);
    ListNode seven1(7, &seven2);
    newHead = removeElements(&seven1, 7);
    print(newHead);
}
```

```
Output:
[1,2,3,4,5,]
[]
[]
```

This solution creates a `preHead` node with a value of `2023` (an arbitrary value larger than `50`) and sets its `next` pointer to point to the original `head` of the linked list.

The purpose of this `preHead` node is to serve as the dummy or sentinel node at the beginning of the linked list. Having a `preHead` node simplifies the code because it eliminates the need to handle the special case of removing nodes from the beginning of the list separately.

The remaining code is the same.

Complexity

- Runtime: `O(N)`, where `N` is the number of nodes.
- Memory: `O(1)`.

Attention!

Depending on your real situation, in practice, you might need to deallocate memory for the removed nodes; especially when they were allocated by the `new` operator.

```
ListNode* removeElements(ListNode* head, int val) {
    ListNode preHead(2022, head);
    ListNode* pre = &preHead;
    while (pre->next) {
        if (pre->next->val == val) {
            ListNode* node = pre->next;
            pre->next = node->next;
            delete node;
        } else {
            pre = pre->next;
        }
    }
    return preHead.next;
}
```

3.2.4 Key takeaway

- In some linked list problems where `head` needs to be treated as a special case, you can create a previous dummy node for it to adapt the general algorithm.
- Be careful with memory leak when removing nodes of the linked list containing pointers.

3.2.5 Exercise

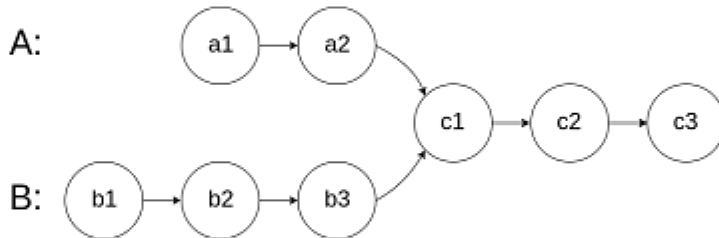
- Delete Node in a Linked List

3.3 Intersection of Two Linked Lists

3.3.1 Problem statement

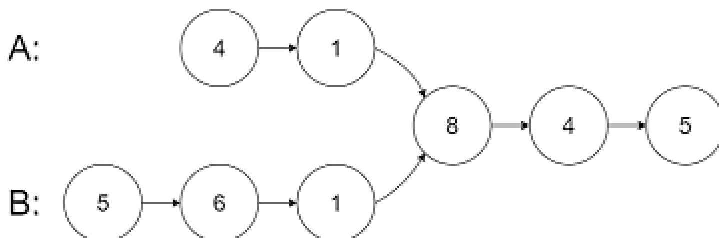
[1]You are provided with the starting nodes of two singly linked lists, `headA` and `headB`. Your task is to find the node where these two lists intersect. If there is no point of intersection, return `null`.

For example, the following two linked lists begin to intersect at node `c1`:



Note that the linked lists do not have any cycles, and you must ensure that the original structure of the linked lists remains unchanged after solving this problem.

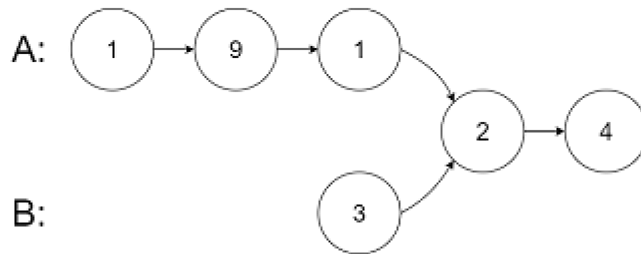
Example 1



[1] https://leetcode.com/problems/intersection-of-two-linked-lists/

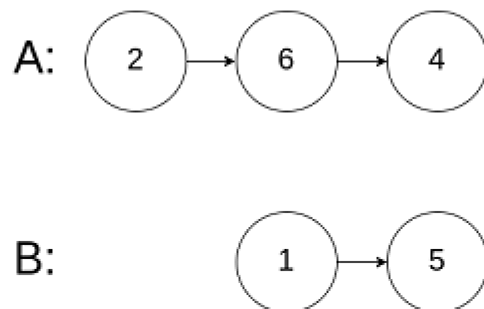
```
Input: listA = [4,1,8,4,5], listB = [5,6,1,8,4,5].
Output: Intersected at '8'
```

Example 2




```
Input: listA = [1,9,1,2,4], listB = [3,2,4]
Output: Intersected at '2'
```

Example 3




```
Input: listA = [2,6,4], listB = [1,5]
Output: No intersection.
```

Constraints

- The number of nodes of `listA` is in the `m`.
- The number of nodes of `listB` is in the `n`.
- `1 <= m, n <= 3 * 10^4`.
- `1 <= Node.val <= 10^5`.

Follow up

- Could you write a solution that runs in `O(m + n)` time and use only `O(1)` memory?

3.3.2 Solution 1: Store the nodes

You can store all nodes of `listA` then iterate `listB` to determine which node is the intersection. If none is found, the two lists have no intersection.

Example 1

- Store all nodes of `listA = [4,1,8,4,5]` in a map.
- Iterate `listB` and found node `'8'` was stored.
- Return `'8'`.

Code

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
```

(continues on next page)

(continued from previous page)

```
};

ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    unordered_map<ListNode*, bool> m;
    ListNode *node = headA;
    while (node != nullptr) {
        m[node] = true;
        node = node->next;
    }
    node = headB;
    while (node != nullptr) {
        if (m.find(node) != m.end()) {
            return node;
        }
        node = node->next;
    }
    return nullptr;
}
int main() {
    {   // Example 1
        ListNode five(5);
        ListNode four(4);
        four.next = &five;
        ListNode eight(8);
        eight.next = &four;

        ListNode one1(1);
        one1.next = &eight;
        ListNode four1(4);
        four1.next = &one1;

        ListNode one2(1);
        one2.next = &eight;
        ListNode six2(6);
        six2.next = &one2;
        ListNode five2(5);
```

(continues on next page)

(continued from previous page)

```
        five2.next = &six2;
        cout << (getIntersectionNode(&four1, &five2) == &eight) << endl;
    }
    {   // Example 2
        ListNode four(4);
        ListNode two(2);
        two.next = &four;

        ListNode one12(1);
        one12.next = &two;
        ListNode nine1(9);
        nine1.next = &one12;
        ListNode one11(1);
        one11.next = &nine1;

        ListNode three2(3);
        three2.next = &two;
        cout << (getIntersectionNode(&one11, &three2) == &two) << endl;
    }
    {   // Example 3
        ListNode four(4);
        ListNode six(6);
        six.next = &four;
        ListNode two(2);
        two.next = &six;

        ListNode five(5);
        ListNode one(1);
        one.next = &five;
        cout << (getIntersectionNode(&two, &one) == nullptr) << endl;
    }
}
```

```
Output:
1
1
```

(continues on next page)

(continued from previous page)

```
1
```

This code uses an unordered map to store the nodes of `headA` while traversing it. Then, it traverses `headB` and checks if each node in `headB` exists in the map of nodes from `headA`. If a common node is found, it returns that node as the intersection point; otherwise, it returns `nullptr` to indicate no intersection.

Complexity

- Runtime: `O(m + n)`, where `m`, `n` are the number of nodes of `listA` and `listB`.
- Extra space: `O(m)`.

3.3.3 Solution 2: Reiterating the two lists at the same time

If the two lists do not share the same tail, they have no intersection. Otherwise, they must intersect at some node.

After iterating to find the tail node, you know the length of the two lists. That information gives you a hint of how to reiterate to find the intersection node.

Example 1

- After iterating `listA = [4,1,8,4,5]`, you find the tail node is `'5'` and `listA.length = 5`.
- After iterating `listB = [5,6,1,8,4,5]`, you find the tail node is the last `'5'` and `listB.length = 6`.
- The two lists share the same tail. They must intersect at some node.
- To find that intersection node, you have to reiterate the two lists.
- Since `listB.length = 6 > 5 = listA.length`, you can start iterating `listB` first until the number of its remaining nodes is the same as `listA`. In this case, it is the node `'6'` of `listB`.
- Now you can iterate them at the same time to find which node is shared.

- Found and return the intersection node '8'.

Code

```
#include <iostream>
#include <unordered_map>
using namespace std;
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB)
{
    int lengthA = 0;
    ListNode *nodeA = headA;
    while (nodeA->next != nullptr) {
        lengthA++;
        nodeA = nodeA->next;
    }
    int lengthB = 0;
    ListNode *nodeB = headB;
    while (nodeB->next != nullptr) {
        lengthB++;
        nodeB = nodeB->next;
    }
    // not the same tail -> no intersection
    if (nodeA != nodeB) {
        return nullptr;
    }
    nodeA = headA;
    nodeB = headB;
    // find the nodeA in listA and nodeB in listB
    // that make two lists have the same length
    while (lengthA > lengthB) {
        nodeA = nodeA->next;
        lengthA--;
```

(continues on next page)

(continued from previous page)

```
    }
    while (lengthB > lengthA) {
        nodeB = nodeB->next;
        lengthB--;
    }
    // find the intersection
    while (nodeA != nodeB) {
        nodeA = nodeA->next;
        nodeB = nodeB->next;
    }
    return nodeA;
}
int main() {
    {   // Example 1
        ListNode five(5);
        ListNode four(4);
        four.next = &five;
        ListNode eight(8);
        eight.next = &four;

        ListNode one1(1);
        one1.next = &eight;
        ListNode four1(4);
        four1.next = &one1;

        ListNode one2(1);
        one2.next = &eight;
        ListNode six2(6);
        six2.next = &one2;
        ListNode five2(5);
        five2.next = &six2;
        cout << (getIntersectionNode(&four1, &five2) == &eight) << endl;
    }
    {   // Example 2
        ListNode four(4);
        ListNode two(2);
```

(continues on next page)

(continued from previous page)

```
        two.next = &four;

        ListNode one12(1);
        one12.next = &two;
        ListNode nine1(9);
        nine1.next = &one12;
        ListNode one11(1);
        one11.next = &nine1;

        ListNode three2(3);
        three2.next = &two;
        cout << (getIntersectionNode(&one11, &three2) == &two) << endl;
    }
    {   // Example 3
        ListNode four(4);
        ListNode six(6);
        six.next = &four;
        ListNode two(2);
        two.next = &six;

        ListNode five(5);
        ListNode one(1);
        one.next = &five;
        cout << (getIntersectionNode(&two, &one) == nullptr) << endl;
    }
}
```

```
Output:
1
1
1
```

This improved solution finds the intersection of two linked lists by first determining their lengths and adjusting the pointers so that they start from the same relative position to the intersection point. Then, it iterates through both linked lists until it finds the common intersection node.

Complexity

- Runtime: `O(m + n)`, where `m`, `n` are the number of nodes of `listA` and `listB`.
- Extra space: `O(1)`.

3.3.4 Implementation tip

- The technique used in Solution 2 is known as the *Two-pointer* technique since you use two pointers to iterate the list at the same time.

3.3.5 Exercise

- Minimum Index Sum of Two Lists

3.4 Swap Nodes in Pairs

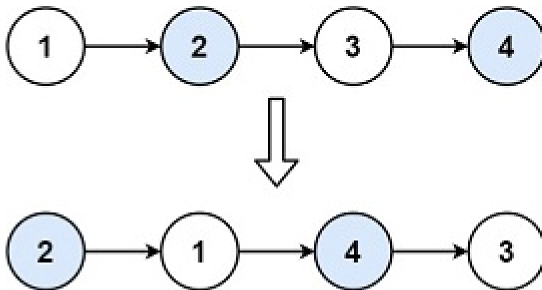
3.4.1 Problem statement

[1]You are provided with a linked list. Your goal is to exchange every two adjacent nodes in the list and then return the head of the modified list.

You must solve this problem without altering the values within the nodes; you should only modify the arrangement of the nodes themselves.

[1] https://leetcode.com/problems/swap-nodes-in-pairs/

Example 1




```
Input: head = [1,2,3,4]
Output: [2,1,4,3]
```

Example 2

```
Input: head = []
Output: []
```

Example 3

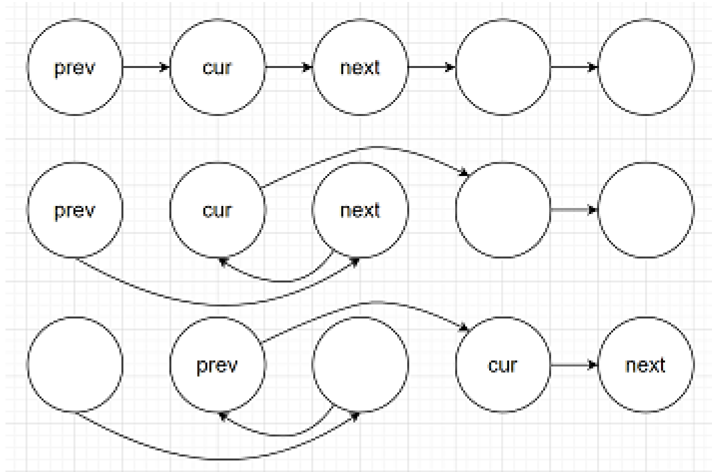
```
Input: head = [1]
Output: [1]
```

Constraints

- The number of nodes in the list is in the range `[0, 100]`.
- `0 <= Node.val <= 100`.

3.4.2 Solution

Draw a picture of the swapping to identify the correct order of the update.



Denote `(cur, next)` the pair of nodes you want to swap and `prev` be the previous node that links to `cur`. Here are the steps you need to perform for the swapping.

1. Update the links between nodes.
2. Go to the next pair.

Code

```
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
ListNode* swapPairs(ListNode* head) {
    // the list does not have enough nodes to swap
    if (head == nullptr || head->next == nullptr) {
```

(continues on next page)

(continued from previous page)

```
        return head;
    }
    ListNode* preNode = nullptr;
    ListNode* curNode = head;
    ListNode* nextNode = head->next;
    head = nextNode;
    while (curNode != nullptr && nextNode != nullptr) {

        // swap curNode and nextNode
        curNode->next = nextNode->next;
        nextNode->next = curNode;

        // update links/pointers after swap
        if (preNode) {
            preNode->next = nextNode;
        }

        // update nodes for next step
        preNode = curNode;
        curNode = curNode->next;
        if (curNode) {
            nextNode = curNode->next;
        }
    }
    return head;
}
void print(const ListNode* head) {
    ListNode* node = head;
    std::cout << "[";
    while (node != nullptr) {
        std::cout << node->val << ",";
        node = node->next;
    }
    std::cout << "]" << std::endl;
}
int main() {
```

(continues on next page)

(continued from previous page)

```
    ListNode four(4);
    ListNode three(3, &four);
    ListNode two(2, &three);
    ListNode one(1, &two);
    print(swapPairs(&one));
    ListNode five(5);
    print(swapPairs(nullptr));
    print(swapPairs(&five));
}
```

```
Output:
[2,1,4,3,]
[]
[5,]
```

Complexity

- Runtime: `O(N)`, where `N` is the number of nodes.
- Extra space: `O(1)`.

3.4.3 Conclusion

This solution swaps pairs of nodes in a linked list by adjusting the pointers accordingly.

It initializes pointers to the current node (`curNode`), its next node (`nextNode`), and the previous node (`preNode`). Then, it iterates through the list, swapping pairs of nodes by adjusting their `next` pointers and updating the `preNode` pointer.

This approach efficiently swaps adjacent nodes in the list without requiring additional space, effectively transforming the list by rearranging pointers.

3.4.4 Exercise

- Swapping Nodes in a Linked List

3.5 Add Two Numbers

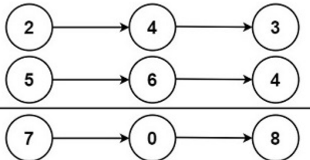
3.5.1 Problem statement

[1]You have two linked lists that represent non-negative integers. The digits of these numbers are stored in reverse order, with each node containing a single digit.

Your task is to add the two numbers represented by these linked lists and return the result as a new linked list.

You can assume that the two numbers don't have leading zeros, except for the number 0 itself.

Example 1




```
Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
Explanation: 342 + 465 = 807.
```

[1] https://leetcode.com/problems/add-two-numbers/

Example 2

```
Input: l1 = [0], l2 = [0]
Output: [0]
```

Example 3

```
Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
Output: [8,9,9,9,0,0,0,1]
```

Constraints

- The number of nodes in each linked list is in the range `[1, 100]`.
- `0 <= Node.val <= 9`.
- It is guaranteed that the list represents a number that does not have leading zeros.

3.5.2 Solution: Addition With Remember

Perform the school addition calculation and store the result in one of the lists.

Without loss of generality, let us store the result in `l1`. Then you might need to extend it when `l2` is longer than `l1` and when the result requires one additional node (Example 3).

Code

```cpp
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
```

(continues on next page)

(continued from previous page)

```
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    // dummy node to hook the head of the list
    ListNode prehead;

    // let's use l1's nodes to store result
    ListNode* node = l1;
    prehead.next = node;
    int sum = 0;
    while (node) {

        // perform the addition
        if (l1) {
            sum += l1->val;
            l1 = l1->next;
        }
        if (l2) {
            sum += l2->val;
            l2 = l2->next;
        }
        node->val = sum % 10;

        // keep track the carry
        sum /= 10;
        if (!l1) {      // l1 ends
            if (l2) {   // l1 is shorter than l2
                node->next = l2;
            } else if (sum == 1) {
                // both l1 and l2 end but the remember is not zero
                ListNode* newNode = new ListNode(sum);
                node->next = newNode;
            }
        }
        node = node->next;
```

(continues on next page)

(continued from previous page)

```
    }
    return prehead.next;
}
void printResult(ListNode* l) {
    std::cout << "[";
    while (l) {
        std::cout << l->val << ",";
        l = l->next;
    }
    std::cout << "]\n";
}
int main() {
    {
        ListNode three(3);
        ListNode four1(4, &three);
        ListNode two(2, &four1);
        ListNode four2(4);
        ListNode six(6, &four2);
        ListNode five(5, &six);
        printResult(addTwoNumbers(&two, &five));
    }
    {
        ListNode zero1(0);
        ListNode zero2(0);
        printResult(addTwoNumbers(&zero1, &zero2));
    }
    {
        ListNode nine0(9);
        ListNode nine1(9, &nine0);
        ListNode nine2(9, &nine1);
        ListNode nine3(9, &nine2);
        ListNode nine4(9, &nine3);
        ListNode nine5(9, &nine4);
        ListNode nine6(9, &nine5);
        ListNode nine7(9);
        ListNode nine8(9, &nine7);
```

(continues on next page)

(continued from previous page)

```
        ListNode nine9(9, &nine8);
        ListNode nine10(9, &nine9);
        printResult(addTwoNumbers(&nine6, &nine10));
    }
}
```

```
Output:
[7,0,8,]
[0,]
[8,9,9,9,0,0,0,1,]
```

Complexity

- Runtime: `O(N)`, where `N = max(l1.length, l2.length)`.
- Extra space: `O(1)`.

3.5.3 Conclusion

This solution leverages a dummy node (`prehead`) to simplify the handling of edge cases and to hook the head of the resulting list.

By iterating through both input lists simultaneously and performing addition digit by digit while keeping track of carry, it efficiently computes the sum without the need for additional checks for the head of the resulting list.

This approach streamlines the addition process, resulting in a concise and straightforward implementation.

3.5.4 Exercise

- Double a Number Represented as a Linked List

CHAPTER

FOUR

HASH TABLE

This chapter is about the C++ Standard Template Library's `std::unordered_map` and how it can help your programming.

With hash-based data structures, you can store and retrieve information quickly, like a well-organized library. Hash tables allow you to efficiently manage data by inserting, locating, and removing elements, even from large datasets. C++'s `std::unordered_map` makes it easy to use hash tables without manual implementation.

What this chapter covers:

1. **Exploring std::unordered_map:** Dive into the C++ Standard Template Library's *std::unordered_map* container, learning how to use it effectively for mapping keys to values.
2. **Problem-Solving with Hash Tables:** Learn strategies for solving many problems using hash tables, including frequency counting, anagram detection, and more.

4.1 Roman to Integer

4.1.1 Problem statement

[1]Roman numerals utilize seven symbols: `I`, `V`, `X`, `L`, `C`, `D`, and `M` to represent numbers.

[1] https://leetcode.com/problems/roman-to-integer/

```
Symbol       Value
I             1
V             5
X             10
L             50
C             100
D             500
M             1000
```

For example, `2` is denoted as `II`, which is essentially two ones added together. Similarly, `12` is represented as `XII`, indicating `X + II`. The number `27` is written as `XXVII`, which stands for `XX + V + II`.

Roman numerals are generally written from the largest value to the smallest value, moving from left to right. However, there are exceptions to this pattern. For instance, the numeral for `4` is `IV` instead of `IIII`, where `I` is placed before `V` to subtract `1` from `5`. Similarly, `9` is `IX`, representing the subtraction of `1` from `10`. There are six such subtraction instances:

- `I` before `V` (`5`) or `X` (`10`) forms `4` and `9`.
- `X` before `L` (`50`) or `C` (`100`) forms `40` and `90`.
- `C` before `D` (`500`) or `M` (`1000`) forms `400` and `900`.

Your task is to convert a given Roman numeral into its equivalent integer value.

Example 1

```
Input: s = "III"
Output: 3
Explanation: III = 3.
```

Example 2

```
Input: s = "LVIII"
Output: 58
Explanation: L = 50, V= 5, III = 3.
```

Example 3

```
Input: s = "MCMXCIV"
Output: 1994
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

Constraints

- `1 <= s.length <= 15`.
- s contains only the characters `'I'`, `'V'`, `'X'`, `'L'`, `'C'`, `'D'`, `'M'`.
- It is guaranteed that s is a valid Roman numeral in the range `[1, 3999]`.

4.1.2 Solution: Mapping and summing the values

To treat the subtraction cases easier you can iterate the string s backward.

Code

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;
const unordered_map<char, int> value = {
    {'I', 1},   {'V', 5},
    {'X', 10},  {'L', 50},
    {'C', 100}, {'D', 500},
    {'M', 1000}
```

(continues on next page)

(continued from previous page)

```
};
int romanToInt(const string& s) {

    // starting from the end character of the string s
    int i = s.length() - 1;
    int result = value.at(s[i--]);
    while (i >= 0) {
        // In cases of subtraction
        if (value.at(s[i]) < value.at(s[i+1])) {
            result -= value.at(s[i--]);
        } else {
            result += value.at(s[i--]);
        }
    }
    return result;
}
int main() {
    cout << romanToInt("III") << endl;
    cout << romanToInt("LVIII") << endl;
    cout << romanToInt("MCMXCIV") << endl;
}
```

```
Output:
3
58
1994
```

Complexity

- Runtime: `O(N)` where `N = s.length`.
- Extra space: `O(1)`.

4.1.3 Conclusion

This problem can be solved using a map to store the values of each Roman numeral character. This solution iterates through the string from right to left, accumulating the integer value based on the corresponding Roman numeral characters.

By comparing the current character's value with the previous one, the solution handles cases of subtraction (e.g., IV, IX, etc.) by subtracting the value if it's smaller and adding it otherwise.

4.1.4 Exercise

- Integer to Roman

4.2 Maximum Erasure Value

4.2.1 Problem statement

[1]You have an array of positive integers called `nums`, and you wish to remove a subarray from it that consists of distinct elements. The score you achieve by removing this subarray is the sum of its elements.

Your goal is to determine the highest possible score attainable by erasing exactly one subarray from the provided array.

A subarray, denoted as `b`, is considered part of another array, `a`, if it appears consecutively within `a`, i.e., if it is equivalent to `a[l], a[l+1], ..., a[r]` for some indices `(l, r)`.

[1] https://leetcode.com/problems/maximum-erasure-value/

Example 1

```
Input: nums = [4,2,4,5,6]
Output: 17
Explanation: The optimal subarray here is [2,4,5,6].
```

Example 2

```
Input: nums = [5,2,1,2,5,2,1,2,5]
Output: 8
Explanation: The optimal subarray here is [5,2,1] or [1,2,5].
```

Constraints

- `1 <= nums.length <= 10^5.`
- `1 <= nums[i] <= 10^4.`

4.2.2 Solution: Store the position of the visited elements

You can use a map to store the position of the elements of `nums`. Then when iterating `nums` you can identify if an element has been visited before. That helps you to decide if a subarray contains unique elements.

Code

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;
int maximumUniqueSubarray(const vector<int>& nums) {
    // sum stores the running sum of nums
    // i.e., sum[i] = nums[0] + ... + nums[i]
    vector<int> sum(nums.size(), 0);
```

(continues on next page)

(continued from previous page)

```
    sum[0] = nums.at(0);

    // store the maximum sum of the maximum subarray
    int maxSum = sum.at(0);

    // position[a] keeps track of latest index i whose nums[i] = a
    unordered_map<int, int> position;
    position[nums.at(0)] = 0;

    // the starting index of the current subarray
    int start = -1;
    for (int i = 1; i < nums.size(); i++) {
        sum[i] = sum.at(i - 1) + nums.at(i);

        // check if the current subarray's elements are still distinct
        auto it = position.find(nums.at(i));

        // found the value nums[i] in the map position
        if (it != position.end()) {
            // a new subarray now starts from i
            start = max(start, it->second);

            // clear the latest index of the value nums[i]
            // by updating it to i
            it->second = i;
        } else {
            // current subarray continues with nums[i]
            position.insert({nums.at(i), i});
        }
        if (start == -1) {
            // still on the first subarray nums[0]..nums[i]
            maxSum = sum.at(i);
        } else {
            // update maxSum with the sum of subarray nums[start]..
↪nums[i]
            maxSum = max(maxSum, sum.at(i) - sum.at(start));
```

(continues on next page)

(continued from previous page)

```
        }
    }
    return maxSum;
}
int main() {
    vector<int> nums{4,2,4,5,6};
    cout << maximumUniqueSubarray(nums) << endl;
    nums = {5,2,1,2,5,2,1,2,5};
    cout << maximumUniqueSubarray(nums) << endl;
}
```

```
Output:
17
8
```

Complexity

- Runtime: `O(N)`, where `N = nums.length`.
- Extra space: `O(N)`.

4.2.3 Conclusion

This solution computes the maximum sum of a subarray containing unique elements.

It uses a sliding window approach to maintain a running sum of the elements encountered so far and a hashmap to keep track of the positions of previously seen elements. By updating the starting index of the window when a repeated element is encountered, it ensures that the current subarray contains only unique elements.

This approach optimizes the computation of the maximum sum by handling the sliding window and updating the sum accordingly, resulting in an overall efficient solution.

4.3 Find and Replace Pattern

4.3.1 Problem statement

[1]You are provided with a list of strings named `words` and a string named `pattern`. Your task is to find the strings from `words` that match the given `pattern`. The order in which you return the answers does not matter.

A word is considered to match the pattern if there is a mapping `p` of the letters such that, when each letter `x` in the pattern is replaced with `p(x)`, the word is formed.

Keep in mind that a permutation of letters is a one-to-one correspondence from letters to letters, where each letter is mapped to a distinct letter, and no two letters are mapped to the same letter.

Example 1

```
Input: words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"
Output: ["mee","aqq"]
Explanation: "mee" matches the pattern because there is a permutation
↪{a -> m, b -> e, ...}.
"ccc" does not match the pattern because {a -> c, b -> c, ...} is not a␣
↪permutation, since a and b map to the same letter.
```

Example 2

```
Input: words = ["a","b","c"], pattern = "a"
Output: ["a","b","c"]
```

[1] https://leetcode.com/problems/find-and-replace-pattern/

Constraints

- `1 <= pattern.length <= 20`.
- `1 <= words.length <= 50`.
- `words[i].length == pattern.length`.
- `pattern` and `words[i]` are lowercase English letters.

4.3.2 Solution: Construct the bijection and check the condition

Code

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
vector<string> findAndReplacePattern(const vector<string>& words, const␣
↪string& pattern) {
    vector<string> result;
    // need two maps for the bijection
    unordered_map<char,char> w_to_p, p_to_w;
    int i;
    for (auto& w : words) {
        w_to_p.clear();
        p_to_w.clear();
        i = 0;
        while (i < w.length()) {
            if (w_to_p.find(w[i]) != w_to_p.end()) {
                // w[i] was mapped to some letter x
                // but x != pattern[i]
                if (w_to_p[w[i]] != pattern[i]) {
                    break;
                }
            } else {
                if (p_to_w.find(pattern[i]) != p_to_w.end()) {
                    // w[i] was not mapped to any letter yet
```

(continues on next page)

(continued from previous page)

```
                // but pattern[i] was already mapped to some letter
                break;
            }
            // build the bijection w[i] <-> pattern[i]
            w_to_p[w[i]] = pattern[i];
            p_to_w[pattern[i]] = w[i];
        }
        i++;
    }
    if (i == w.length()) {
        result.push_back(w);
    }
}
return result;
}
void printResult(const vector<string>& result) {
    cout << "[";
    for (auto& s : result) {
        cout << s << ",";
    }
    cout << "]\n";
}
int main() {
    vector<string> words{"abc","deq","mee","aqq","dkd","ccc"};
    auto result = findAndReplacePattern(words, "abb");
    printResult(result);
    words = {"a", "b", "c"};
    result = findAndReplacePattern(words, "abb");
    printResult(result);
}
```

```
Output:
[mee,aqq,]
[a,b,c,]
```

Complexity

- Runtime: `O(NL)`, where `N = words.length` and `L = pattern.length`.
- Extra space: `O(1)`. The maps `w_to_p` and `p_to_w` just map between 26 lowercase English letters.

4.3.3 Conclusion

This solution efficiently finds and returns words from a vector of strings that match a given pattern in terms of character bijection. It uses two unordered maps to establish and maintain the bijection while iterating through the characters of the words and the pattern.

CHAPTER FIVE

STRING

In this chapter, we'll learn about the importance of **strings** in programming. Strings help us work with text and are essential for many tasks, from processing data to creating better communication between programs and people. By understanding strings, you'll be better equipped to solve problems and make things easier for users.

What this chapter covers:

1. **Understanding Strings:** Lay the groundwork by comprehending the nature of strings, character encoding schemes, and the basics of representing and storing textual data.
2. **String Manipulation:** Explore the art of string manipulation, covering operations like concatenation, slicing, reversing, and converting cases.
3. **String Searching and Pattern Matching:** Delve into strategies for finding substrings, detecting patterns, and performing advanced search operations within strings.
4. **Anagrams and Palindromes:** Tackle challenges related to anagrams and palindromes, honing your ability to discern permutations and symmetric constructs.
5. **Problem-Solving with Strings:** Learn how to approach coding problems that involve string manipulation, from simple tasks to intricate algorithms.

5.1 Valid Anagram

5.1.1 Problem statement

[1]You are given two strings, `s` and `t`. Your task is to determine whether `t` is an anagram of `s`. If `t` is an anagram of `s`, return `true`; otherwise, return `false`.

An **anagram** is a term that describes a word or phrase formed by rearranging the letters of another word or phrase, typically using all the original letters exactly once.

Example 1

```
Input: s = "anagram", t = "nagaram"
Output: true
```

Example 2

```
Input: s = "rat", t = "car"
Output: false
```

Constraints

- `1 <= s.length, t.length <= 5 * 10^4`.
- `s` and `t` consist of lowercase English letters.

Follow up

- What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

[1] https://leetcode.com/problems/valid-anagram/

5.1.2 Solution 1: Rearrange both s and t into a sorted string

Code

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
bool isAnagram(string& s, string& t) {
    // anagrams must have the same length
    if (s.length() != t.length()) {
        return false;
    }
    sort(s.begin(), s.end());
    sort(t.begin(), t.end());
    return s == t;
}
int main() {
    cout << isAnagram("anagram", "nagaram") << endl;
    cout << isAnagram("rat", "car") << endl;
}
```

```
Output:
1
0
```

This solution determines if two strings are anagrams by comparing their sorted versions. If the sorted versions are equal, the original strings are anagrams, and the function returns `true`. Otherwise, it returns `false`.

Complexity

- Runtime: `O(NlogN)`, where `N = s.length`.
- Extra space: `O(1)`.

5.1.3 Solution 2: Count the appearances of each letter

Code

```
#include <iostream>
using namespace std;
bool isAnagram(const string& s, const string& t) {
    if (s.length() != t.length()) {
        return false;
    }
    // s and t consist of only lowercase English letters
    // you can encode 0: 'a', 1: 'b', .., 25: 'z'.
    int alphabet[26];
    for (int i = 0; i < 26; i++) {
        alphabet[i] = 0;
    }
    // count the frequency of each letter in s
    for (auto& c : s) {
        alphabet[c - 'a']++;
    }
    for (auto& c : t) {
        alphabet[c - 'a']--;
        // if s and t have the same length but are not anagrams,
        // there must be some letter in t having higher frequency than␣
↪s
        if (alphabet[c - 'a'] < 0) {
            return false;
        }
    }
    return true;
}
int main() {
    cout << isAnagram("anagram", "nagaram") << endl;
    cout << isAnagram("rat", "car") << endl;
}
```

```
Output:
```

(continues on next page)

(continued from previous page)

```
1
0
```

This solution efficiently determines if two strings are anagrams by counting the frequency of each character in both strings using an array. If the character frequencies match for both strings, they are anagrams.

Complexity

- Runtime: `O(N)`, where `N = s.length`.
- Extra space: `O(1)`.

5.1.4 Solution 3: If the inputs contain Unicode characters

Replace the array `alphabet` in Solution 2 with a map.

Code

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;
bool isAnagram(const string& s, const string& t) {
    if (s.length() != t.length()) {
        return false;
    }
    // this alphabet can store all UTF-8 characters
    unordered_map<char, int> alphabet;
    for (auto& c : s) {
        alphabet[c]++;
    }
    for (auto& c : t) {
        alphabet[c]--;
        if (alphabet[c] < 0) {
            return false;
```

(continues on next page)

(continued from previous page)

```
        }
    }
    return true;
}
int main() {
    cout << isAnagram("anagram", "nagaram") << endl;
    cout << isAnagram("rat", "car") << endl;
}
```

```
Output:
1
0
```

Complexity

- Runtime: `O(N)`, where `N = s.length`.
- Extra space: `O(c)` where `c` represents the number of unique characters present in both strings `s` and `t`.

5.1.5 Key Takeaway

Instead of relying on a fixed-size array like the ASCII-based solutions, Solution 3 uses an `unordered_map` to store character frequencies. Each character is used as a key in the map, and the count of occurrences is stored as the associated value.

Unicode characters values are not restricted to a specific range. The `unordered_map` approach accommodates this variability by allowing any character to be a key.

5.1.6 Exercise

- Find Resultant Array After Removing Anagrams

5.2 Detect Capital

5.2.1 Problem statement

[1]The task is to determine if the usage of capital letters in a given string, `word`, is correct according to the following rules:

1. All letters in the word are capital, like "USA".
2. All letters in the word are not capital, like "leetcode".
3. Only the first letter in the word is capital, like "Google".

If the capitalization in the given word adheres to these rules, the function should return `true`; otherwise, it should return `false`.

Example 1

```
Input: word = "USA"
Output: true
```

Example 2

```
Input: word = "FlaG"
Output: false
```

[1] https://leetcode.com/problems/detect-capital/

Constraints

- `1 <= word.length <= 100`,
- `word` consists of lowercase and uppercase English letters.

5.2.2 Solution

Only when the first two characters of the `word` are uppercase, the rest must be the same. Otherwise, the rest is always lowercase.

Code

```cpp
#include <string>
#include <iostream>
using namespace std;
//! @return true if (c is lowercase and isLower is true)
//!                  or (c is uppercase and isLower is false).
//!          false, otherwise.
bool isValidCase(const char& c, const bool isLower) {
    if (isLower) {
        return 'a' <= c && c <= 'z';
    }
    return 'A' <= c && c <= 'Z';
}
bool detectCapitalUse(const string& word) {
    if (word.length() == 1) {
        return true;
    }
    bool isLower = true;

    // if the first two characters are uppercase,
    // the rest must be uppercase, too.
    if (isValidCase(word[0], false) && isValidCase(word[1], false)) {
        isLower = false;
    }
```

(continues on next page)

(continued from previous page)

```
    for (int i = 1; i < word.length(); i++) {
        if (!isValidCase(word[i], isLower)) {
            return false;
        }
    }
    return true;
}
int main() {
    cout << detectCapitalUse("USA") << endl;
    cout << detectCapitalUse("FlaG") << endl;
    cout << detectCapitalUse("leetcode") << endl;
    cout << detectCapitalUse("Google") << endl;
}
```

```
Output:
1
0
1
1
```

Complexity

- Runtime: `O(N)`, where `N = word.length`.
- Extra space: `O(1)`.

5.2.3 Conclusion

This solution efficiently checks whether a given word follows one of the specified capitalization rules by iterating through the characters of the word and using the `isValidCase` function to validate each character's capitalization based on the current capitalization type (`isLower`). If no violations are found, the word is considered valid, and the function returns `true`.

5.2.4 Exercise

- Capitalize the Title

5.3 Unique Morse Code Words

5.3.1 Problem statement

[1]The problem involves the International Morse Code, which defines a standard way to encode letters with dots and dashes. Each English letter corresponds to a specific sequence in Morse Code, and a full table mapping each letter is provided.

For instance, `'a'` is encoded as `".-"`, `'b'` as `"-..."`, and so on.

The full table for the `26` letters of the English alphabet is given below:

```
[".-", "-...", "-.-.", "-..", ".", "..-.", "--.",
"....", "..", ".---", "-.-", ".-..", "--", "-.",
"---", ".--.", "--.-", ".-.", "...", "-", "..-",
"...-", ".--", "-..-", "-.--", "--.."]
```

You are given an array of strings named `words`, where each word can be represented as a concatenation of the Morse code for each of its letters. For example, the word `"cab"` can be represented as `"-.-..--..."`, which is the concatenation of `"-.-."`, `".-"`, and `"-..."`. This concatenated Morse code representation is referred to as the "transformation" of a word.

Your task is to count the number of different transformations that can be obtained from all the words in the given array.

[1] https://leetcode.com/problems/unique-morse-code-words/

Example 1

```
Input: words = ["gin","zen","gig","msg"]
Output: 2
Explanation: The transformation of each word is:
"gin" -> "--...-."
"zen" -> "--...-."
"gig" -> "--...--."
"msg" -> "--...--."
There are 2 different transformations: "--...-." and "--...--.".
```

Example 2

```
Input: words = ["a"]
Output: 1
```

Constraints

- `1 <= words.length <= 100`.
- `1 <= words[i].length <= 12`.
- `words[i]` consists of lowercase English letters.

5.3.2 Solution: Store the transformations in a set

Code

```
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;
const vector<string> morse{
    ".-", "-...", "-.-.", "-..", ".", "..-.", "--.",
    "....", "..", ".---", "-.-", ".-..", "--", "-.",
```

(continues on next page)

(continued from previous page)

```
    "---", ".--.", "--.-", ".-.", "...", "-", "..-",
    "...-", ".--", "-..-", "-.--", "--.."
};

int uniqueMorseRepresentations(const vector<string>& words) {
    unordered_set<string> transformations;
    for (auto& w : words) {
        string s{""};
        for (auto& c : w) {
            // concatnate the letter c's Morse code
            s += morse[c - 'a'];
        }
        // only insert the transformation s to the set
        // if the set did not consist s yet.
        transformations.insert(s);
    }
    return transformations.size();
}
int main() {
    vector<string> words{"gin","zen","gig","msg"};
    cout << uniqueMorseRepresentations(words) << endl;
    words = {"a"};
    cout << uniqueMorseRepresentations(words) << endl;
}
```

```
Output:
2
1
```

Complexity

- Runtime: `O(N*M)`, where `N = words.length` and `M = words[i].length`.
- Extra space: `O(N)`.

5.3.3 Conclusion

This solution converts each word into Morse code based on a predefined mapping and uses an unordered set to keep track of unique representations. By inserting each representation into the set, it automatically filters out duplicates. The final result is the size of the set, which represents the number of unique Morse code representations among the input words.

5.4 Unique Email Addresses

5.4.1 Problem statement

[1]Each valid email address is composed of a local name and a domain name, separated by the `'@'` sign. The local name may contain lowercase letters, one or more `'.'` characters, and a plus `'+'` sign. However, the rules for dots and the plus sign do not apply to the domain name.

For example, in the email `"alice@leetcode.com"`, `"alice"` is the local name, and `"leetcode.com"` is the domain name.

If you insert periods `'.'` between certain characters in the local name, the email will still be forwarded to the same address without the dots in the local name. This rule does not apply to the domain name.

For example, `"alice.z@leetcode.com"` and `"alicez@leetcode.com"` both forward to the same email address.

If you include a plus `'+'` sign in the local name, everything after the first plus sign is ignored, allowing for email filtering. This rule also does not apply to the domain name.

For example, `"m.y+name@email.com"` will be forwarded to `"my@email.com"`.

It is possible to use both of these rules at the same time.

Given an array of strings `emails`, where each element is an email address to which an email is sent, your task is to determine the number of different addresses that will actually receive the emails after applying the rules described above.

[1] https://leetcode.com/problems/unique-email-addresses/

Example 1

```
Input: emails = ["test.email+alex@leetcode.com","test.e.mail+bob.
↪cathy@leetcode.com","testemail+david@lee.tcode.com"]
Output: 2
Explanation: "testemail@leetcode.com" and "testemail@lee.tcode.com"␣
↪actually receive mails.
```

Example 2

```
Input: emails = ["a@leetcode.com","b@leetcode.com","c@leetcode.com"]
Output: 3
```

Constraints

- `1 <= emails.length <= 100`.
- `1 <= emails[i].length <= 100`.
- `emails[i]` consist of lowercase English letters, `'+'`, `'.'` and `'@'`.
- Each `emails[i]` contains exactly one `'@'` character.
- All local and domain names are non-empty.
- Local names do not start with a `'+'` character.
- Domain names end with the `".com"` suffix.

5.4.2 Solution 1: Removing the ignored characters

Do exactly the steps the problem describes:

1. Extract the local name.
2. Ignore all characters after `'+'` in it.
3. Ignore all `'.'` in it.
4. Combine the local name with the domain one to form the clean email address.

Code

```
#include<string>
#include<iostream>
#include<vector>
#include <unordered_set>
using namespace std;
int numUniqueEmails(const vector<string>& emails) {
    unordered_set<string> s;
    for (auto& e: emails) {
        auto apos = e.find('@');

        // extract the local name
        string local = e.substr(0, apos);

        // ignore all characters after '+'
        local = local.substr(0, local.find('+'));
        auto it = local.find('.');
        while (it != string::npos) {
            // remove each '.' found in local
            local.erase(it, 1);
            it = local.find('.');
        }
        // combine local name with domain one
        s.insert(local + e.substr(apos));
    }
    return s.size();
}
int main() {
    vector<string> emails{"test.email+alex@leetcode.com",
                        "test.e.mail+bob.cathy@leetcode.com",
                        "testemail+david@lee.tcode.com"};
    cout << numUniqueEmails(emails) << endl;
    emails = {"a@leetcode.com","b@leetcode.com","c@leetcode.com"};
    cout << numUniqueEmails(emails) << endl;
    emails = {"test.email+alex@leetcode.com","test.email.leet+alex@code.
↪com"};
```

(continues on next page)

(continued from previous page)

```
    cout << numUniqueEmails(emails) << endl;
}
```

```
Output:
2
3
2
```

This solution parses a list of email addresses, normalizes each email address by removing periods and ignoring characters after the plus sign in the local name, and then counts the number of unique email addresses. The use of an unordered set ensures that only unique email addresses are counted.

Complexity

- Runtime: `O(N*M^2)`, where `N = emails.length` and `M = max(emails[i].length)`. Explanation: you loop over `N` emails. Then you might loop over the length of each email, `O(M)`, to remove the character `'.'`. The removal might cost `O(M)`.
- Extra space: `O(N*M)` (the set of emails).

5.4.3 Solution 2: Building the clean email addresses from scratch

The runtime of removing characters in `std::string` is not constant. To avoid that complexity you can build up the clean email addresses from scratch.

Code

```
#include<string>
#include<iostream>
#include<vector>
#include <unordered_set>
using namespace std;
```

(continues on next page)

(continued from previous page)

```
int numUniqueEmails(const vector<string>& emails) {
    unordered_set<string> s;
    for (auto& e: emails) {
        string address;
        int i = 0;
        // the local name ends here
        while (e[i] != '@' && e[i] != '+') {
            // ignore each '.' found
            if (e[i++] == '.') {
                continue;
            }
            // add valid characters to local name
            address += e[i++];
        }
        // combine local name with domain one
        address += e.substr(e.find('@', i));
        s.insert(address);
    }
    return s.size();
}
int main() {
    vector<string> emails{"test.email+alex@leetcode.com",
                          "test.e.mail+bob.cathy@leetcode.com",
                          "testemail+david@lee.tcode.com"};
    cout << numUniqueEmails(emails) << endl;
    emails = {"a@leetcode.com","b@leetcode.com","c@leetcode.com"};
    cout << numUniqueEmails(emails) << endl;
    emails = {"test.email+alex@leetcode.com","test.email.leet+alex@code.
↪com"};
    cout << numUniqueEmails(emails) << endl;
}
```

```
Output:
2
3
2
```

Complexity

- Runtime: `O(N*M)`, where `N = emails.length` and `M = max(emails[i].length)`.
- Extra space: `O(N*M)`.

5.4.4 C++ Notes

- A `string` can be concatenated with a `char` and another `string` by + operator.

```
std::string address = "name";
address += '@';              // "name@"
address += "domain.com";     // "name@domain.com"
```

- string::substr(pos = 0, count = npos) returns the substring of length `count` starting from the position `pos` of the string `string`.

```
std::string address = "name@domain.com";
cout << address.substr(address.find('.'));      // ".com"
cout << address.substr(0, address.find('@'));   // "name"
```

- string::find(char, pos=0) returns the position of the first `char` which appears in the string `string` starting from `pos`.

5.4.5 High-performance C++

- Do not use `std::set` or `std::map` unless you want the keys to be *in order* (*sorted*). Use *unordered containers* like std::unordered_set or std::unordered_map instead. They use hashed keys for faster lookup.
- Do not blindly/lazily use `string.find(something)`. If you know where to start the search, use `string.find(something, pos)` with a **specific** `pos`.

5.5 Longest Substring Without Repeating Characters

5.5.1 Problem statement

[1]Given a string s, your task is to determine the length of the longest substring within s that does not contain any repeating characters.

Example 1

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with a length of 3.
```

Example 2

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with a length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and␣
↪not a substring.
```

[1] https://leetcode.com/problems/longest-substring-without-repeating-characters/

Constraints

- `0 <= s.length <= 5 * 10^4`.
- `s` consists of English letters, digits, symbols and spaces.

5.5.2 Solution: Store the position of the visited characters

Whenever you meet a visited character `s[i] == s[j]` for some `0 <= i < j < s.length`, the substring "`s[i]...s[j - 1]`" might be valid, i.e., it consists of only nonrepeating characters.

But in case you meet another visited character `s[x] == s[y]` where `x < i < j < y`, the substring "`s[x]...s[y - 1]`" is not valid because it consists of repeated character `s[i] == s[j]`.

That shows the substring "`s[i]...s[j - 1]`" is not always a valid one. You might need to find the right starting position `start >= i` for the valid substring "`s[start]...s[j - 1]`".

Example 4

For the string `s = "babba"`:

- When you visit the second letter `'b'`, the substring `"ba"` is a valid one.
- When you visit the third letter `'b'`, the substring of interest should be started by the second letter `'b'`. It gives you the substring `"b"`.
- When you visit the second letter `'a'`, the substring `"abb"` is not a valid one since `'b'` is repeated. To ensure no repetition, the starting position for this substring should be the latter `'b'`, which leads to the valid substring `"b"`.
- The final longest valid substring is `"ba"` with length 2.

Example 4 shows the starting position `start` for the substring of interest "`s[i]...s[j - 1]`" should be:

```
this_start = max(previous_start, i).
```

Code

```
#include <iostream>
#include <unordered_map>
using namespace std;
int lengthOfLongestSubstring(const string& s) {
    // keep track latest index of a character in s
    unordered_map<char, int> position;

    // posible maximum length of the longest substring
    int maxLen = 0;

    // starting index of current substring
    int start = -1;
    for (int i = 0; i < s.length(); i++) {
        auto it = position.find(s.at(i));
        // found this s[i] has appeared in the map
        if (it != position.end()) {
            // start a new substring from this index i
            start = max(start, it->second);

            // update latest found position of character s[i] to be i
            it->second = i;
        } else {

            // keep track index of s[i] in the map
            position.insert({s.at(i), i});
        }
        // update the maximum length
        maxLen = max(maxLen, i - start);
    }
    return maxLen;
}
int main() {
    cout << lengthOfLongestSubstring("abcabcbb") << endl;
    cout << lengthOfLongestSubstring("bbbbb") << endl;
    cout << lengthOfLongestSubstring("pwwkew") << endl;
```

(continues on next page)

(continued from previous page)

```
}
```

```
Output:
3
1
3
```

Complexity

- Runtime: `O(N)`, where `N = s.length`.
- Extra space: `O(N)`.

5.5.3 Conclusion

This solution utilizes a sliding window approach to track the starting index of the current substring and an unordered map to store the position of the characters encountered so far. By updating the starting index when a repeating character is encountered, it ensures that the current substring contains only unique characters.

This approach optimizes the computation of the length of the longest substring by handling the sliding window and updating the length accordingly, resulting in an overall efficient solution.

5.5.4 Exercise

- Optimal Partition of String

5.6 Compare Version Numbers

5.6.1 Problem statement

[1]Given two version numbers, `version1` and `version2`, your task is to compare them.

Version numbers consist of one or more revisions joined by a dot `'.'`. Each revision is composed of digits and may contain leading zeros. Each revision has at least one character. Revisions are indexed from left to right, with the leftmost revision being revision 0, the next revision being revision 1, and so on.

For instance, `2.5.33` and `0.1` are valid version numbers.

To compare version numbers, you should compare their revisions in left-to-right order. Revisions are compared using their integer value, ignoring any leading zeros. This means that revisions `1` and `001` are considered equal. If a version number does not specify a revision at a particular index, treat that revision as `0`. For example, version `1.0` is less than version `1.1` because their revision 0s are the same, but their revision 1s are `0` and `1` respectively, and `0` is less than `1`.

The function should return the following:

- If `version1` is less than `version2`, return `-1`.
- If `version1` is greater than `version2`, return `1`.
- If `version1` and `version2` are equal, return `0`.

Example 1

```
Input: version1 = "1.01", version2 = "1.001"
Output: 0
Explanation: Ignoring leading zeroes, both "01" and "001" represent the
↪same integer "1".
```

[1] https://leetcode.com/problems/compare-version-numbers/

Example 2

```
Input: version1 = "1.0", version2 = "1.0.0"
Output: 0
Explanation: version1 does not specify revision 2, which means it is␣
↪treated as "0".
```

Example 3

```
Input: version1 = "0.1", version2 = "1.1"
Output: -1
Explanation: version1's revision 0 is "0", while version2's revision 0␣
↪is "1". 0 < 1, so version1 < version2.
```

Constraints

- `1 <= version1.length, version2.length <= 500`.
- `version1` and `version2` only contain digits and `'.'`.
- `version1` and `version2` are valid version numbers.
- All the given revisions in `version1` and `version2` can be stored in a 32-bit integer.

5.6.2 Solution

Each version can be considered as an array of revisions.

```
version = revisions[0].revisions[1].revisions[2]....
```

The problem is to compare each `revisions[i]` between two versions.

For example, `revisions[0]` of `version1` is less than of `version2` in Example 3. So the result is `-1`.

All `revisions[i]` of `version1` and `version2` are equal in Example 1. So the result is `0`.

The number of revisions between the versions might not be equal (like in Example 2).

If all revisions of the shorter version are equal to the corresponding revisions of the longer one, the version having extra revisions and there exists a non-zero revision among them is the bigger one. Otherwise, the two versions are equal.

Code

```
#include <iostream>
#include <vector>
#include <string>
#include <numeric>
using namespace std;
//! @return the vector of revisions of the version
//! @example if version = "1.02.11", return {1,2,11}
vector<int> toVector(const string& version) {
    vector<int> revisions;
    string revision;
    for (auto& c : version) {
        if (c != '.') {
            // continue to build current revision
            revision += c;
        } else {
            // current revision completes
            // uses stoi() to ignore leading zeros
            revisions.push_back(stoi(revision));

            // start a new revision
            revision = "";
        }
    }
    revisions.push_back(stoi(revision));
    return revisions;
}

int compareVersion(const string& version1, const string& version2) {
```

(continues on next page)

(continued from previous page)

```
    vector<int> r1 = toVector(version1);
    vector<int> r2 = toVector(version2);

    int i = 0;
    // perform the comparison on the revisions
    while (i < r1.size() && i < r2.size()) {
        if (r1[i] < r2[i]) {
            return -1;
        } else if (r1[i] > r2[i]) {
            return 1;
        }
        i++;
    }
    if (i == r1.size()) {
        // if version1 is not longer than version2
        // and version2 still has some valid revisions remain
        if (accumulate(r2.begin() + i, r2.end(), 0) > 0) {
            return -1;
        }
    } else if (accumulate(r1.begin() + i, r1.end(), 0) > 0) {
        // if version2 is not longer than version1
        // and version1 still has some valid revisions remain
        return 1;
    }
    return 0;
}
int main() {
    cout << compareVersion("1.01", "1.001") << endl;
    cout << compareVersion("1.0", "1.0.0") << endl;
    cout << compareVersion("0.1", "1.1") << endl;
}
```

```
Output:
0
0
-1
```

Complexity

- Runtime: `O(N)` where `N = max(version1.length, version2.length)`.
- Extra space: `O(N)`.

5.6.3 Conclusion

This solution first converts the version strings into vectors of integers representing the individual components of the version numbers. This conversion is done by iterating through each character of the version string, accumulating digits until encountering a dot, at which point the accumulated integer is added to the revisions vector.

Once both version strings are converted into vectors, the function iterates through the vectors, comparing corresponding elements to determine the relationship between the versions. Additionally, it accounts for any remaining digits in the longer version string after the common components by summing them up and comparing the totals.

This approach simplifies the comparison process by breaking down the version strings into easily comparable components.

C++ Notes

- std::stoi[2] is used to convert a `string` to an `int`. It ignores the leading zeros for you.
- std::accumulate[3] is used to compute the sum of a container.

---

[2] https://en.cppreference.com/w/cpp/string/basic_string/stol
[3] https://en.cppreference.com/w/cpp/algorithm/accumulate

CHAPTER SIX

STACK

This chapter explores the **stack** data structure, a useful tool for managing data in a Last-In-First-Out (LIFO) way. We'll investigate the basics of stacks and examine how they work using C++'s *std::stack*` and `std::vector` from the Standard Template Library (STL).

Stacks in programming are like a stack of books where you add and remove books from the top. They provide a structured way to manage data, making them ideal for handling temporary information, tracking function calls, and solving various algorithmic challenges.

What this chapter covers:

1. **Introduction to Stacks:** Begin by understanding the core principles of stacks, their fundamental operations, and their real-world applications.
2. **Leveraging `std::stack`:** Dive into the STL's powerful *std::stack* container, mastering its usage and versatility for stack-based operations.
3. **Exploring `std::vector`:** Discover the capabilities of *std::vector* in context with stacks, exploiting its dynamic array nature to create flexible stack structures.
4. **Stack Operations:** Explore operations such as **push** and **pop**, understanding their impact on the stack's state and memory usage.
5. **Balancing Parentheses:** Tackle the classic problem of parentheses balancing using stacks, a prime example of their utility in parsing and validation.

As you progress through this chapter, you'll learn about the importance of stacks and how *std::stack* and *std::vector* can help you solve problems more efficiently. By

the end of the chapter, you'll thoroughly understand the stack data structure's Last-In-First-Out (LIFO) principle and how you can leverage *std::stack* and *std::vector* to manage data effectively. Let's embark on this enlightening journey through stacks and uncover their potential for simplifying complex operations and algorithmic problems!

6.1 Baseball Game

6.1.1 Problem statement

[1]You are responsible for keeping score in a unique baseball game with special rules. The game involves multiple rounds where the scores of previous rounds can influence the scores of future rounds.

At the beginning of the game, your record is empty. You are given a list of operations called `ops`, where each `ops[i]` is one of the following:

1. An integer `x` - This represents recording a new score of `x`.
2. `"+"` - This represents recording a new score that is the sum of the previous two scores. It is guaranteed that there will always be two previous scores.
3. `"D"` - This represents recording a new score that is double the previous score. It is guaranteed that there will always be a previous score.
4. `"C"` - This represents invalidating the previous score, removing it from the record. It is guaranteed that there will always be a previous score.

Your task is to calculate and return the sum of all the scores in the record after performing all the operations.

[1] https://leetcode.com/problems/baseball-game/

Example 1

```
Input: ops = ["5","2","C","D","+"]
Output: 30
Explanation:
"5" - Add 5 to the record; the record is now [5].
"2" - Add 2 to the record; the record is now [5, 2].
"C" - Invalidate and remove the previous score; the record is now [5].
"D" - Add 2 * 5 = 10 to the record; the record is now [5, 10].
"+" - Add 5 + 10 = 15 to the record, record is now [5, 10, 15].
The total sum is 5 + 10 + 15 = 30.
```

Example 2

```
Input: ops = ["5","-2","4","C","D","9","+","+"]
Output: 27
Explanation:
"5" - Add 5 to the record; the record is now [5].
"-2" - Add -2 to the record; the record is now [5, -2].
"4" - Add 4 to the record; the record is now [5, -2, 4].
"C" - Invalidate and remove the previous score; the record is now [5, -
↪2].
"D" - Add 2 * -2 = -4 to the record; the record is now [5, -2, -4].
"9" - Add 9 to the record; the record is now [5, -2, -4, 9].
"+" - Add -4 + 9 = 5 to the record, record is now [5, -2, -4, 9, 5].
"+" - Add 9 + 5 = 14 to the record, record is now [5, -2, -4, 9, 5, 14].
The total sum is 5 + -2 + -4 + 9 + 5 + 14 = 27.
```

Example 3

```
Input: ops = ["1"]
Output: 1
```

Constraints

- `1 <= ops.length <= 1000`.
- `ops[i]` is `"C"`, `"D"`, `"+"`, or a string representing an integer in the range `[-3 * 10^4, 3 * 10^4]`.
- For operation `"+"`, there will always be at least two previous scores on the record.
- For operations `"C"` and `"D"`, there will always be at least one previous score on the record.

6.1.2 Solution

Code

```
#include <vector>
#include <iostream>
#include <string>
#include <numeric>
using namespace std;
int calPoints(const vector<string>& ops) {
    vector<int> stk;
    for (auto& s : ops) {
        if (s == "C") {
            stk.pop_back();
        } else if (s == "D") {
            stk.push_back(stk.back()*2);
        } else if (s == "+") {
            stk.push_back(stk[stk.size() - 1] + stk[stk.size() - 2]);
        } else { // s is an integer
            stk.push_back(stoi(s));
        }
    }
    // compute the sum
    return accumulate(stk.begin(), stk.end(), 0);
}
```

(continues on next page)

(continued from previous page)

```cpp
int main() {
    vector<string> ops{"5","2","C","D","+"};
    cout << calPoints(ops) << endl;
    ops = {"5","-2","4","C","D","9","+","+"};
    cout << calPoints(ops) << endl;
}
```

```
Output:
30
27
```

This solution simulates the baseball game by processing each round's operation and maintaining a stack of valid points. It accurately calculates the final sum of valid points based on the given operations.

Complexity

- Runtime: `O(N)`, where `N = ops.length`.
- Extra space: `O(N)`.

6.1.3 Implementation tips

1. The data structure `stk` you might need to solve this problem is a stack. But here are the reasons you had better use `std::vector`:
   - `std::vector` has also methods `push_back(value)` and `pop_back()` like the ones in stack.
   - On the other hand, a stack does not give easy access to the second last element for the operator `"+"` in this problem.
2. `accumulate(stk.begin(), stk.end(), 0)` computes the sum of the vector `stk`.

6.1.4 Exercise

- Crawler Log Folder

6.2 Valid Parentheses

6.2.1 Problem statement

[1]You are given a string s containing only the characters '(', ')', '{', '}', '[', and ']'. Your task is to check if the input string is valid.

A string is considered valid if the following conditions are satisfied:

1. Opening brackets must be closed by the same type of brackets.
2. Opening brackets must be closed in the correct order, meaning that the innermost opening bracket should be closed before its surrounding brackets.

Example 1

```
Input: s = "()"
Output: true
```

Example 2

```
Input: s = "()[]{}"
Output: true
```

[1] https://leetcode.com/problems/valid-parentheses/

Example 3

```
Input: s = "(]"
Output: false
```

Constraints

- `1 <= s.length <= 10^4`.
- `s` consists of parentheses only `'()[]{}'`.

6.2.2 Solution: Using a stack

For each character `c` of `s`:

1. If it is an open parenthesis (`'('`, `'{'`, or `'['`), push it into the stack.
2. If it is a closed parenthesis (`')'`, `'}'`, or `']'`) but its previous character is not the corresponding open one, return `false`. End.
3. Otherwise (i.e. match open-closed), erase the pair.
4. Continue the process until all characters of `s` are visited.
5. Return `true` if the stack is empty, i.e. all valid pairs are erased.

Code

```cpp
#include <iostream>
#include <stack>
using namespace std;
bool isValid(const string& s) {
    stack<char> stk;
    for (auto& c : s) {
        if (c == '(' || c == '[' || c == '{') {
            stk.push(c);
        } else if (stk.empty()) {
```

(continues on next page)

(continued from previous page)

```
            // start with a non-open parenthesis is invalid
            return false;
        } else if (c == ')' && stk.top() != '('
              || c == ']' && stk.top() != '['
              || c == '}' && stk.top() != '{') {
            // the last open parenthesis does not match this closed one
            return false;
        } else {
            // open-close match
            stk.pop();
        }
    }
    return stk.empty();
}
int main() {
    cout << isValid("()") << endl;
    cout << isValid("(){}[]") << endl;
    cout << isValid("(]") << endl;
    cout << isValid("([)]") << endl;
}
```

```
Output:
1
1
0
0
```

Complexity:

- Runtime: `O(N)`, where `N = s.length`.
- Extra space: `O(N)`.

6.2.3 Conclusion

This solution efficiently checks the validity of a string of parentheses, brackets, and curly braces by using a stack to ensure that each opening bracket is correctly matched with its corresponding closing bracket.

6.2.4 Exercise

- Check If Word Is Valid After Substitutions

6.3 Backspace String Compare

6.3.1 Problem statement

[1]You are provided with two strings, `s` and `t`. Your task is to determine if these two strings are equal when typed into an empty text editor, where the character `'#'` represents a backspace action.

Note that applying a backspace action to an empty text does not change the text; it remains empty. Your function should return `true` if the two strings become equal after considering the backspace actions, otherwise return `false`.

Example 1

```
Input: s = "ab#c", t = "ad#c"
Output: true
Explanation: Both s and t become "ac".
```

[1] https://leetcode.com/problems/backspace-string-compare/

Example 2

```
Input: s = "ab##", t = "c#d#"
Output: true
Explanation: Both s and t become "".
```

Example 3

```
Input: s = "a#c", t = "b"
Output: false
Explanation: s becomes "c" while t becomes "b".
```

Constraints

- `1 <= s.length, t.length <= 200.`
- `s` and `t` only contain lowercase letters and `'#'` characters.

Follow up

- Can you solve it in `O(n)` time and `O(1)` space?

6.3.2 Solution: Build and clean the string using the `stack`'s behaviors

Code

```
#include <iostream>
#include <vector>
using namespace std;
string cleanString(const string &s) {
    vector<char> v;
    for (int i = 0; i < s.length(); i++) {
        if (s[i] != '#') {
```

(continues on next page)

(continued from previous page)

```
            // s[i] is a normal letter
            v.push_back(s[i]);
        } else {
            if (!v.empty()) {
                // perform the backspace
                v.pop_back();
            }
        }
    }
    // create a string from a vector of char
    return string(v.begin(), v.end());
}
bool backspaceCompare(const string& s, const string& t) {
    return cleanString(s) == cleanString(t);
}
int main() {
    cout << backspaceCompare("ab#c", "ad#c") << endl;
    cout << backspaceCompare("ab##", "c#d#") << endl;
    cout << backspaceCompare("a#c", "b") << endl;
}
```

```
Output:
1
1
0
```

This solution effectively handles backspace characters ('#') in input strings s and t by constructing cleaned versions of the strings and then comparing the cleaned strings for equality.

Complexity

- Runtime: `O(n)`, where `n = max(s.length, t.length)`.
- Extra space: `O(n)`.

6.3.3 Implementation notes

Why `vector` instead of `stack`?

You can use the methods `push` and `pop` of the data structure `stack` to build and clean the strings.

But `vector` has also such methods: `push_back` and `pop_back`.

On the other hand, using `vector` it is easier to construct a `string` by constructor than using `stack` after cleaning.

Can you solve it in `O(n)` time and `O(1)` space?

Yes, you can.

The simplest way is just to perform the erasure directly on strings `s` and `t`. But the run time complexity of `string::erase` is not constant.

6.3.4 Exercise

- Removing Stars From a String

6.4 Remove All Adjacent Duplicates in String II

6.4.1 Problem statement

[1]You are given a string s and an integer k. A k duplicate removal operation involves selecting k adjacent and identical letters from s and removing them, causing the remaining portions on the left and right of the removed substring to join together.

You need to perform the k duplicate removal operation on s repeatedly until it is no longer possible. After completing all such operations, return the resulting string. It is guaranteed that the answer will be unique.

Example 1

```
Input: s = "abcd", k = 2
Output: "abcd"
Explanation: There is nothing to delete.
```

Example 2

```
Input: s = "deeedbbcccbdaa", k = 3
Output: "aa"
Explanation:
First delete "eee" and "ccc", get "ddbbbdaa"
Then delete "bbb", get "dddaa"
Finally delete "ddd", get "aa"
```

[1] https://leetcode.com/problems/remove-all-adjacent-duplicates-in-string-ii/

Example 3

```
Input: s = "pbbcggttciiippooaais", k = 2
Output: "ps"
```

Constraints

- `1 <= s.length <= 10^5.`
- `2 <= k <= 10^4.`
- s only contains lower case English letters.

6.4.2 Solution: Strings of adjacent equal letters

Construct a stack of strings that has adjacent equal letters and perform the removal during building those strings.

Example 2

For `s = "deeedbbcccbdaa"` and `k = 3`:

- The first built string is `"d"`.
- Then `"eee"` with the exact length `k`, remove this string.
- The next character is `'d'`, which equals the last character of the last string `"d"`, merge them together. The first string becomes `"dd"`.
- The next string is `"bb"`.
- Then `"ccc"` is removed.
- The next character `'b'` is merged with the last string (`"bb"`) to become `"bbb"` and be removed.
- The next character `'d'` is merged with the last string (`"dd"`) to become `"ddd"` and be removed.
- The remaining string is `"aa"`.

Code

```
#include <iostream>
#include <vector>
using namespace std;
string removeDuplicates(string& s, int k) {
    // stk is used as a stack
    // all letters in each string a of stk are equal
    // every a's length is less than k
    vector<string> stk;
    int i = 0;
    while (i < s.length()) {
        // a represents the current string with duplicate letters
        string a;

        if (!stk.empty() && s[i] == stk.back().back()) {
            // let a be the latest string in stk
            // because its letters are equal to s[i]
            a = move(stk.back());
            stk.pop_back();
        }
        int j = i;
        // iterate all adjacent duplicates of s[i]
        while (j < s.length() && s[j] == s[i]) {
            a += s[j];
            // remove the k-duplicate
            if (a.length() == k) {
                a = "";
            }
            j++;
        }
        // after the loop, the number of duplicates in a is less than k
        if (!a.empty()) {
            stk.push_back(a);
        }
        i = j;
    }
```

(continues on next page)

(continued from previous page)

```
    // create the final result from stk
    s = "";
    for (auto& str : stk) {
        s += str;
    }
    return s;
}
int main() {
    cout << removeDuplicates("abcd", 2) << endl;
    cout << removeDuplicates("deeedbbcccbdaa", 3) << endl;
    cout << removeDuplicates("pbbcggttciiippooaais", 2) << endl;
}
```

```
Output:
abcd
aa
ps
```

This solution efficiently removes substrings of consecutive duplicate characters of length `k` from a given string by using a stack to keep track of the adjacent characters.

Complexity

- Runtime: `O(N)`, where `N = s.length`.
- Extra space: `O(N)`.

6.4.3 Implementation tips

- The data structure `stk` you might need to solve this problem is a stack. But here are the reasons you had better use `std::vector`:
- `std::vector` also has methods `push_back(value)` and `pop_back()` like the ones in a stack.
- On the other hand, it is faster for a vector to perform the string concatenation at the end.

- In the expression `stk.back().back()`: `stk.back()` is the latest string `a` of `stk`. Then `stk.back().back() = a.back()` is the last character of `a`.

6.4.4 Exercise

- Remove All Adjacent Duplicates In String

CHAPTER

SEVEN

PRIORITY QUEUE (HEAP)

This chapter explores **priority queues** (or **heaps**), the fascinating data structures designed to manage elements with distinct levels of importance. In this chapter, we'll focus on harnessing the capabilities of C++'s `std::priority_queue` from the Standard Template Library (STL).

Think of a priority queue as a line at a theme park, where individuals with priority passes are served before others. Similarly, a priority queue ensures that elements with higher priority are processed ahead of those with lower priority, enabling us to address a wide range of problems that involve ordering and selection.

What this chapter covers:

1. **Understanding Priority Queues:** Begin by grasping the essence of priority queues, their underlying mechanisms, and the significance of their unique ordering.
2. **Leveraging std::priority_queue:** Dive into the versatile `std::priority_queue` container provided by the STL, mastering its usage for managing priorities effectively.
3. **Operations and Methods:** Explore the operations available in `std::priority_queue`, including insertion, and extraction while maintaining optimal order.
4. **Custom Comparators:** Customize the behavior of your priority queue by utilizing custom comparators, tailoring it to handle diverse data types and priority criteria.
5. **Problem-Solving with Priority Queues:** Learn strategies for tackling problems where prioritization is key, from scheduling tasks to efficient data re-

trieval.

7.1 Last Stone Weight

7.1.1 Problem statement

[1]You are given an array of integers called `stones`, where each `stones[i]` represents the weight of the `i-th` stone.

A game is played with these stones as follows: In each turn, we choose the two heaviest stones and smash them together. Let us say the weights of the two heaviest stones are `x` and `y`, where `x <= y`. The outcome of this smash operation is:

1. If `x` is equal to `y`, both stones are destroyed.
2. If `x` is not equal to `y`, the stone with weight `x` is destroyed, and the stone with weight `y` now has a new weight of `y - x`.

The game continues until there is at most one stone left. Your task is to determine the smallest possible weight of the remaining stone after the game ends. If there are no stones left, return `0`.

Example 1

```
Input: stones = [2,7,4,1,8,1]
Output: 1
Explanation:
We combine 7 and 8 to get 1, so the array converts to [2,4,1,1,1] then,
we combine 2 and 4 to get 2, so the array converts to [2,1,1,1] then,
we combine 2 and 1 to get 1, so the array converts to [1,1,1] then,
we combine 1 and 1 to get 0, so the array converts to [1] then that's
↪the value of the last stone.
```

[1] https://leetcode.com/problems/last-stone-weight/

Example 2

```
Input: stones = [1]
Output: 1
```

Constraints

- `1 <= stones.length <= 30`.
- `1 <= stones[i] <= 1000`.

7.1.2 Solution: Keeping the heaviest stones on top

The only things you want at any time are the two heaviest stones. One way of keeping this condition is by using std::priority_queue.

Code

```cpp
#include <vector>
#include <iostream>
#include <queue>
using namespace std;
int lastStoneWeight(vector<int>& stones) {
    priority_queue<int> q(stones.begin(), stones.end());
    while (q.size() >= 2) {
        int y = q.top();
        q.pop();
        int x = q.top();
        q.pop();
        // compare two heaviest stones
        if (y != x) {
            q.push(y - x);
        }
    }
    return q.empty() ? 0 : q.top();
```

(continues on next page)

(continued from previous page)

```
}
int main() {
    vector<int> stones{2,7,4,1,8,1};
    cout << lastStoneWeight(stones) << endl;
    stones = {1};
    cout << lastStoneWeight(stones) << endl;
}
```

```
Output:
1
1
```

Complexity

- Runtime: `O(n*logn)`, where `n = stones.length`.
- Extra space: `O(n)`.

7.1.3 Conclusion

This solution efficiently simulates the process of smashing stones and finding the last remaining stone by using a max-heap (priority queue) to always select the heaviest stones to smash together.

7.2 Kth Largest Element in a Stream

7.2.1 Problem statement

[1]Create a class that can find the `k-th` largest element in a stream of integers. This is the `k-th` largest element when the elements are arranged in sorted order, not the `k-th` distinct element.

[1] https://leetcode.com/problems/kth-largest-element-in-a-stream/

The `KthLargest` class needs to support the following operations:

1. `KthLargest(int k, int[] nums)`: This initializes the object with an integer `k` and a stream of integers `nums`.
2. `int add(int val)`: This method adds the integer `val` to the stream and returns the element representing the `k-th` largest element in the stream.

Example 1

```
Input
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
Output
[null, 4, 5, 5, 8, 8]

Explanation
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3);   // return 4
kthLargest.add(5);   // return 5
kthLargest.add(10);  // return 5
kthLargest.add(9);   // return 8
kthLargest.add(4);   // return 8
```

Constraints

- `1 <= k <= 10^4`.
- `0 <= nums.length <= 10^4`.
- `-10^4 <= nums[i] <= 10^4`.
- `-10^4 <= val <= 10^4`.
- At most `10^4` calls will be made to add.
- It is guaranteed that there will be at least `k` elements in the array when you search for the `k-th` element.

7.2.2 Solution 1: Sort and Append

Sort the stream when initialization. And keep it sorted whenever you append a new value.

Example 1

For `nums = [4, 5, 8, 2]` and `k = 3`.

- Sort `nums = [8, 5, 4, 2]`.
- Adding `3` to `nums`. It becomes `[8, 5, 4, 3, 2]`. The third largest element is `4`.
- Adding `5` to `nums`. It becomes `[8, 5, 5, 4, 3, 2]`. The third largest element is `5`.
- Adding `10` to `nums`. It becomes `[10, 8, 5, 5, 4, 3, 2]`. The third largest element is `5`.
- So on and so on.

Code

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
class KthLargest {
    vector<int> _nums;
    int _k;
public:
    KthLargest(int k, vector<int>& nums) : _nums(nums), _k(k) {
        // sort the nums when constructed
        sort(_nums.begin(), _nums.end(), std::greater());
    }

    int add(int val) {
        auto it = _nums.begin();
```

(continues on next page)

(continued from previous page)

```
        // find the position to insert val
        while (it != _nums.end() &&  val < *it) {
            it++;
        }
        _nums.insert(it, val);
        // return the k-th largest element
        return *(_nums.begin() + _k - 1);
    }
};
int main() {
    vector<int> nums{4,5,8,2};
    KthLargest a(3, nums);
    cout << a.add(3) << endl;
    cout << a.add(5) << endl;
    cout << a.add(10) << endl;
    cout << a.add(9) << endl;
    cout << a.add(4) << endl;
}
```

```
Output:
4
5
5
8
8
```

This solution maintains a sorted vector `_nums` in non-ascending order upon initialization, which stores the elements. When adding a new element `val`, it inserts it into `_nums` while maintaining the sorted order.

Since `_nums` is sorted in non-ascending order, the `k`-th largest element is always at index `_k - 1`. Thus, upon adding a new element, it returns the value at index `_k - 1` as the `k`-th largest element in the collection.

This approach optimizes the `add` operation by leveraging the sorted nature of the data structure, resulting in efficient retrieval of the `k`-th largest element.

Complexity

- Runtime: for the constructor `O(N*logN)`, where `N = nums.length`. For the `add` method, `O(N)`.
- Extra space: `O(1)`.

7.2.3 Solution 2: Priority queue

There is a data structure that has the property you want in this problem.

It is `std::priority_queue`, which keeps its top element is always the largest one according to the comparison you define for the queue.

By default, the "less than" comparison is used for std::priority_queue (heap) and the top one is always the biggest element.

If you want the top one is always the smallest element, you can use the comparison "greater than" for your heap.

Code

```
#include <vector>
#include <queue>
#include <iostream>
using namespace std;
class KthLargest {
    priority_queue<int, vector<int>, greater<int>> _q;
    int _k;
public:
    KthLargest(int k, vector<int>& nums)
    // create the heap when constructed
    : _q(nums.begin(), nums.end()), _k(k) {

    }

    int add(int val) {
        _q.push(val);
```

(continues on next page)

(continued from previous page)

```
        // remove elements until _q remains k elements
        while (_q.size() > _k) {
            _q.pop();
        }
        return _q.top();
    }
};
int main() {
    vector<int> nums{4,5,8,2};
    KthLargest a(3, nums);
    cout << a.add(3) << endl;
    cout << a.add(5) << endl;
    cout << a.add(10) << endl;
    cout << a.add(9) << endl;
    cout << a.add(4) << endl;
}
```

```
Output:
4
5
5
8
8
```

Complexity

- Runtime: for the constructor, `O(N*logN)`, where `N = nums.length`. For the `add` method, `O(logN)`.
- Extra space: `O(1)`.

7.2.4 Conclusion

The key insight of Solution 2 is utilizing a min-heap (priority queue with the `greater` comparator) to find the kth largest element in a collection.

Upon initialization, the constructor populates the priority queue with the elements from the input vector `nums`. When adding a new element `val`, it inserts it into the priority queue and then removes elements until the size of the priority queue is reduced to `_k`, ensuring that only the k largest elements are retained in the queue.

Finally, it returns the top element of the priority queue, which represents the kth largest element. This approach leverages the properties of a min-heap to track the kth largest element in the collection, resulting in an overall efficient solution.

7.2.5 Exercise

- Kth Largest Element in an Array

7.3 Kth Smallest Element in a Sorted Matrix

7.3.1 Problem statement

[1]You are given an `n x n` matrix where each row and column is sorted in ascending order. Your task is to find the `k-th` smallest element in this matrix.

Please note that we are looking for the `k-th` smallest element based on its position in the sorted order, and not counting distinct elements.

Additionally, it is required to find a solution with a memory complexity better than `O(n^2)`.

[1] https://leetcode.com/problems/kth-smallest-element-in-a-sorted-matrix/

Example 1

```
Input: matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8
Output: 13
Explanation: The elements in the matrix are [1,5,9,10,11,12,13,13,15],␣
↪and the 8th smallest number is 13
```

Example 2

```
Input: matrix = [[-5]], k = 1
Output: -5
```

Constraints

- `n == matrix.length == matrix[i].length`.
- `1 <= n <= 300`.
- `-10^9 <= matrix[i][j] <= 10^9`.
- All the rows and columns of `matrix` are guaranteed to be sorted in non-decreasing order.
- `1 <= k <= n^2`.

Follow up

- Could you solve the problem with a constant memory (i.e., `O(1)` memory complexity)?
- Could you solve the problem in `O(n)` time complexity? The solution may be too advanced for an interview but you may find reading this paper fun.

7.3.2 Solution 1: Transform the 2-D matrix into a 1-D vector then sort

You can implement exactly what Example 1 has explained.

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int kthSmallest(const vector<vector<int>>& matrix, int k) {
    vector<int> m;
    // transform the 2D matrix into a 1D array m
    for (auto& row : matrix) {
        m.insert(m.end(), row.begin(), row.end());
    }
    // sort the array m
    sort(m.begin(), m.end());
    return m.at(k - 1);
}
int main() {
    vector<vector<int>> matrix{{1,5,9},{10,11,13},{12,13,15}};
    cout << kthSmallest(matrix, 8) << endl;
    matrix = {{-5}};
    cout << kthSmallest(matrix, 1) << endl;
}
```

```
Output:
13
-5
```

The core idea behind this solution is to transform the 2D matrix into a 1D sorted array, making it easier to find the `k`-th smallest element efficiently. The time complexity of this solution is dominated by the sorting step, which is `O(N*logN)`, where `N` is the total number of elements in the matrix.

Complexity

- Runtime: `O(N*logN)`, where `N = n^2` is the total number of elements in the matrix.
- Extra space: `O(N)`.

7.3.3 Solution 2: Build the max heap and keep it ungrown

Instead of sorting after building the vector in Solution 1, you can do the other way around. It means building up the vector from scratch and keeping it sorted.

Since you need only the `k-th` smallest element, std::priority_queue can be used for this purpose.

Code

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
int kthSmallest(const vector<vector<int>>& matrix, int k) {
    priority_queue<int> q;
    for (int row = 0; row < matrix.size(); row++) {
        for (int col = 0; col < matrix[row].size(); col++) {
            q.push(matrix[row][col]);
            // maintain q's size does not exceed k
            if (q.size() > k) {
                q.pop();
            }
        }
    }
    return q.top();
}
int main() {
    vector<vector<int>> matrix{{1,5,9},{10,11,13},{12,13,15}};
    cout << kthSmallest(matrix, 8) << endl;
```

(continues on next page)

(continued from previous page)

```
    matrix = {{-5}};
    cout << kthSmallest(matrix, 1) << endl;
}
```

```
Output:
13
-5
```

Complexity

- Runtime: `O(N*logk)`, where `N = n^2` is the total number of elements of the matrix.
- Extra space: `O(k)`.

7.3.4 Conclusion

Solution 2 maintains a priority queue of size `k`, allowing it to efficiently keep track of the `k`-th smallest element encountered while iterating through the matrix.

This approach is handy for large matrices, as it doesn't require sorting the entire matrix.

7.3.5 Exercise

- Find K Pairs with Smallest Sums

7.4 Construct Target Array With Multiple Sums

7.4.1 Problem statement

[1]You are provided with an array of integers called `target` with `n` elements. You start with another array, `arr`, consisting of `n` elements, all initialized to `1`. You have the ability to perform the following operation:

1. Calculate the sum of all elements in your current array `arr`, let's call it `x`.
2. Choose an index `i` where `0 <= i < n`, and update the value at index `i` in `arr` to be `x`.

You can repeat this operation as many times as needed. Your task is to determine whether it's possible to transform the initial array `arr` into the given `target` array using this operation. If it's possible, return `true`; otherwise, return `false`.

Example 1

```
Input: target = [9,3,5]
Output: true
Explanation: Start with arr = [1, 1, 1]
[1, 1, 1], sum = 3 choose index 1
[1, 3, 1], sum = 5 choose index 2
[1, 3, 5], sum = 9 choose index 0
[9, 3, 5] Done
```

Example 2

```
Input: target = [1,1,1,2]
Output: false
Explanation: Impossible to create target array from [1,1,1,1].
```

[1] https://leetcode.com/problems/construct-target-array-with-multiple-sums/

Example 3

```
Input: target = [8,5]
Output: true
```

Constraints

- `n == target.length`.
- `1 <= n <= 5 * 10^4`.
- `1 <= target[i] <= 10^9`.

7.4.2 Solution 1: Going backward

If you start from `arr = [1,1,...,1]` and follow the required procedure, the new element `x` you get for the next state is always the max element of `arr`.

To solve this problem, you can start from the max element of the given `target` to compute its previous state until you get the `arr = [1,1,...,1]`.

Example 1

For `target = [9,3,5]`:

- The max element is `9`, subtract it from the remaining sum: `9 - (3 + 5) = 1`, you get `target = [1,3,5]`.
- The max element is `5`, subtract it from the remaining sum: `5 - (1 + 3) = 1`, you get `target = [1,3,1]`.
- The max element is `3`, subtract it from the remaining sum: `3 - (1 + 1) = 1`, you get `target = [1,1,1]`.
- Return `true`.

Notes

- If `target = [m,1]` or `target = [1,m]` for any `m >= 1`, you can always turn it to `arr = [1,1]`.
- If the changed value after the subtraction is still the max element of the previous state, you need to redo the subtraction at the same position. In this case, the modulo might be used instead of subtraction.

Code

```
#include <iostream>
#include <numeric>
#include <algorithm>
#include <vector>
using namespace std;
bool isPossible(const vector<int>& target) {
    // compute sum of all target's elements
    unsigned long sum = accumulate(target.begin(),
                                   target.end(),
                                   (unsigned long) 0);
    // find the max element in the target
    // pmax is the pointer to the max element,
    // *pmax is the value that pointer points to
    auto pmax = max_element(target.begin(), target.end());
    while (*pmax > 1) {
        // compute the remaining sum
        sum -= *pmax;
        if (sum == 1) {
            // This is the case target = [m,1],
            // which you can always turn it to [1,1].
            return true;
        }
        if (*pmax <= sum) {
            // the next subtraction leads to non-positive values
            return false;
        }
```

(continues on next page)

(continued from previous page)

```
        if (sum == 0) {
            // cannot change target
            return false;
        }
        // perform the subtraction as much as possible
        // and update new value for the pointer pmax
        *pmax %= sum;
        if (*pmax == 0) {
            return false;
        }
        // compute the sum of the subtracted target
        sum += *pmax;
        // find the max element in the subtracted target
        pmax = max_element(target.begin(), target.end());
    }
    // if the final target = [1, .., 1],
    // its sum equals to its length
    return sum == target.size();
}
int main() {
    vector<int> target{9,3,5};
    cout << isPossible(target) << endl;
    target = {1,1,1,2};
    cout << isPossible(target) << endl;
    target = {8,5};
    cout << isPossible(target) << endl;
}
```

```
Output:
1
0
1
```

This solution iteratively reduces the maximum element in the `target` array while keeping track of the total sum. It checks various conditions to determine whether it's possible to reach an array consisting of only 1s. If all conditions are satisfied, it returns `true`; otherwise, it returns `false`.

Complexity

- Runtime: `O(log N)`, where `N = max(target)`.
- Extra space: `O(1)`.

7.4.3 Solution 2: Using priority_queue

In the solution above, the position of the max element in each state is not so important as long as you update exactly it, not the other ones.

That might lead to the usage of the std::priority_queue.

Code

```
#include <iostream>
#include <numeric>
#include <queue>
#include <vector>
using namespace std;
bool isPossible(const vector<int>& target) {
    // create a heap from the target
    priority_queue<int> q(target.begin(), target.end());
    // compute the sum of all elements
    unsigned long sum = accumulate(target.begin(),
                                   target.end(),
                                   (unsigned long) 0);
    while (q.top() > 1) {
        // compute the remaining sum
        sum -= q.top();
        if (sum == 1) {
            return true;
        }
        if (q.top() <= sum) {
            return false;
        }
        if (sum == 0) {
```

(continues on next page)

(continued from previous page)

```
            return false;
        }
        // perform the subtraction as much as possible
        int pre = q.top() % sum;
        if (pre == 0) {
            return false;
        }
        // remove the old max element
        q.pop();
        // add subtracted element to the heap
        q.push(pre);
        // compute the sum of the subtracted target
        sum += pre;
    }
    return sum == target.size();
}
int main() {
    vector<int> target{9,3,5};
    cout << isPossible(target) << endl;
    target = {1,1,1,2};
    cout << isPossible(target) << endl;
    target = {8,5};
    cout << isPossible(target) << endl;
}
```

```
Output:
1
0
1
```

Complexity

- Runtime: `O(logN)`, where `N = max(target)`.
- Extra space: `O(n)`, where `n = target.length`.

7.4.4 Conclusion

Solution 2 uses a max heap (`priority_queue`) to efficiently find and process the maximum element in the `target` array while keeping track of the total sum. It checks various conditions to determine whether it's possible to reach an array consisting of only 1s.

7.4.5 Exercise

- Minimum Amount of Time to Fill Cups

CHAPTER EIGHT

BIT MANIPULATION

In this chapter, we're diving deep into **Bit Manipulation**, a fascinating computer science and programming area that manipulates individual bits within data.

Bit Manipulation is crucial in various programming tasks, from optimizing algorithms to working with hardware-level data. Whether you're a seasoned programmer looking to expand your skill set or a newcomer eager to delve into the intricacies of bits and bytes, this chapter has something valuable for you.

Here's what you can expect to explore in this chapter:

1. **Understanding the Basics**: We'll start by demystifying bits and binary numbers, ensuring you're comfortable with the fundamentals. You'll learn to convert between decimal and binary, perform basic bit operations, and understand two's complement representation.
2. **Bitwise Operators**: We'll delve into the world of bitwise operators in programming languages like C++. You'll get hands-on experience with AND, OR, XOR, and other essential operators, seeing how they can be applied to practical coding problems.
3. **Bit Hacks**: Discover the art of Bit Hacks – clever and often elegant tricks programmers use to solve specific problems efficiently. You'll learn to perform tasks like counting bits, finding the rightmost set bit, and swapping values without temporary variables.
4. **Bit Manipulation Techniques**: We'll explore techniques and patterns for common bit manipulation tasks, such as setting, clearing, or toggling specific bits, checking if a number is a power of two, or extracting subsets of bits from a larger number.

By the end of this chapter, you'll have a solid foundation in Bit Manipulation and the ability to harness the power of bits to optimize your code and tackle complex problems. So, let's embark on this exciting journey into the realm of Bit Manipulation and discover how the smallest data units can have a massive impact on your coding skills and efficiency!

8.1 Hamming Distance

8.1.1 Problem statement

[1]The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Given two integers `x` and `y`, return the **Hamming distance** between them.

Example 1

```
Input: x = 1, y = 4
Output: 2
Explanation:
1   (0 0 0 1)
4   (0 1 0 0)
       ^   ^
The above arrows point to positions where the corresponding bits are␣
↪different.
```

Example 2

```
Input: x = 3, y = 1
Output: 1
```

[1] https://leetcode.com/problems/hamming-distance/

Constraints

- `0 <= x, y <= 2^31`.

8.1.2 Solution: Using bitwise operator XOR

You could use bitwise XOR (`^`) to get the bit positions where `x` and `y` are different. Then use bitwise AND operator (`&`) at each position to count them.

Example 1

```
x = 1   (0 0 0 1)
y = 4   (0 1 0 0)
z = x^y (0 1 0 1)
```

Code

```cpp
#include <iostream>
int hammingDistance(int x, int y) {
    // compute the bit difference
    int z = x ^ y;
    int count = 0;
    while (z) {
        count += z & 1; // e.g. '0101' & '0001'
        // shift z to the right one position
        z = z >> 1; // e.g. z = '0101' >> '0010'
    }
    return count;
}
int main() {
    std::cout << hammingDistance(1,4) << std::endl; // 2
    std::cout << hammingDistance(1,3) << std::endl; // 1
}
```

```
Output:
2
1
```

Complexity

- Runtime: `O(1)` as the number of bits is at most 32 as constrained.
- Extra space: `O(1)`.

8.1.3 Conclusion

Utilizing bitwise operations, such as XOR (`^`) and bitwise AND (&), allows for efficient computation of the Hamming distance between two integers. This approach provides a straightforward and efficient method for calculating the Hamming distance without the need for complex logic or additional data structures.

8.1.4 Exercise

- Number of 1 Bits

8.2 Power of Four

8.2.1 Problem statement

[1]Given an integer `n`, return `true` if it is a power of four. Otherwise, return `false`.

An integer `n` is a power of four if there exists an integer `x` such that `n == 4^x`.

[1] https://leetcode.com/problems/power-of-four/

Example 1

```
Input: n = 16
Output: true
```

Example 2

```
Input: n = 5
Output: false
```

Example 3

```
Input: n = 1
Output: true
```

Constraints

- `-2^31 <= n <= 2^31 - 1.`

Follow up

- Could you solve it without loops/recursion?

8.2.2 Solution 1: Division by four

Code

```cpp
#include <iostream>
using namespace std;
bool isPowerOfFour(int n) {
    // perform the divison by 4 repeatedly
    while (n % 4 == 0 && n > 0) {
```

(continues on next page)

(continued from previous page)

```
        n /= 4;
    }
    // if n % 4 != 0, then n > 1
    return n == 1;
}
int main()
{
    cout << isPowerOfFour(16) << endl;
    cout << isPowerOfFour(5) << endl;
    cout << isPowerOfFour(1) << endl;
}
```

```
Output:
1
0
1
```

This solution repeatedly divides the given number `n` by 4 until `n` becomes either 1 or a number that is not divisible by 4. If `n` becomes 1 after this process, it means that `n` was originally a power of 4.

Complexity

- Runtime: `O(logn)`.
- Extra space: `O(1)`.

8.2.3 Solution 2: Binary representation

You can write down the binary representation of the powers of four to find the pattern.

```
1    : 1
4    : 100
16   : 10000
```

(continues on next page)

(continued from previous page)

```
64  : 1000000
256 : 100000000
...
```

You might notice the patterns are **n is a positive integer having only one bit 1 in its binary representation and it is located at the odd positions** (starting from the right).

How can you formulate those conditions?

If `n` has only one bit `1` in its binary representation `10...0`, then `n - 1` has the complete opposite binary representation `01...1`.

You can use the bit operator AND to formulate this condition

```
n & (n - 1) == 0
```

Let `A` be the number whose binary representation has only bits `1` at all odd positions, then `n & A` is never `0`.

In this problem, `A < 2^31`. You can choose`A = 0x55555555`, the hexadecimal of `0101 0101 0101 0101 0101 0101 0101 0101`.

Code

```cpp
#include <iostream>
using namespace std;
bool isPowerOfFour(int n) {
    // the condition of the pattern "n is a positive integer
    // having only one bit 1 in its binary representation and
    // it is located at the odd positions"
    return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
}
int main() {
    cout << isPowerOfFour(16) << endl;
    cout << isPowerOfFour(5) << endl;
    cout << isPowerOfFour(1) << endl;
}
```

```
Output:
1
0
1
```

Complexity

- Runtime: `O(1)`.
- Extra space: `O(1)`.

8.2.4 Conclusion

Recognizing the unique properties of powers of four, such as their binary representation, can lead to efficient solutions. Solution 2 leverages bitwise operations to check if a number meets the criteria of being a power of four.

By examining the binary representation and ensuring that the only set bit is located at an odd position, Solution 2 effectively determines whether the number is a power of four in constant time complexity, without the need for division operations.

But in term of readable code, Solution 2 is not easy to understand like Solution 1, where complexity of `O(logn)` is not too bad.

8.2.5 Exercise

- Power of Two

8.3 Find the Duplicate Number

8.3.1 Problem statement

[1]You have an array of integers called `nums` that contains `n + 1` integers. Each integer in the array falls within the range `[1, n]` inclusive.

Within this array, there is only one number that appears more than once. Your task is to find and return this repeated number.

Importantly, you must solve this problem without making any modifications to the original array `nums`, and you are only allowed to use a constant amount of extra space.

Example 1

```
Input: nums = [1,3,4,2,2]
Output: 2
```

Example 2

```
Input: nums = [3,1,3,4,2]
Output: 3
```

Constraints

- `1 <= n <= 10^5`.
- `nums.length == n + 1`.
- `1 <= nums[i] <= n`.
- All the integers in `nums` appear only once except for precisely one integer which appears two or more times.

[1] https://leetcode.com/problems/find-the-duplicate-number/

Follow up

- How can we prove that at least one duplicate number must exist in `nums`?
- Can you solve the problem in linear runtime complexity?

8.3.2 Solution 1: Sorting

Code

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int findDuplicate(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    for (int i = 0; i < nums.size() - 1; i++) {
        if (nums[i] == nums[i + 1]) {
            return nums[i];
        }
    }
    return 0;
}
int main() {
    vector<int> nums{1,3,4,2,2};
    cout << findDuplicate(nums) << endl;
    nums = {3,1,3,4,2};
    cout << findDuplicate(nums) << endl;
}
```

```
Output:
2
3
```

The code relies on sorting to bring duplicate elements together, making it easy to identify them during the linear pass.

Complexity

- Runtime: `O(n*logn)` (sorting).
- Extra space: `O(1)`.

8.3.3 Follow up

How can we prove that at least one duplicate number must exist in `nums`?

Due to Pigeonhole principle:

Here there are `n + 1` pigeons in `n` holes. The pigeonhole principle says that at least one hole has more than one pigeon.

Can you solve the problem in linear runtime complexity?

Here are a few solutions.

8.3.4 Solution 2: Marking the visited numbers

Code

```
#include <vector>
#include <iostream>
using namespace std;
int findDuplicate(const vector<int>& nums) {
    // initialize n + 1 elements false
    vector<bool> visited(nums.size());
    for (auto& a : nums) {
        if (visited.at(a)) {
            return a;
        }
        visited[a] = true;
    }
    return 0;
```

(continues on next page)

(continued from previous page)

```
}
int main() {
    vector<int> nums{1,3,4,2,2};
    cout << findDuplicate(nums) << endl;
    nums = {3,1,3,4,2};
    cout << findDuplicate(nums) << endl;
}
```

```
Output:
2
3
```

This solution utilizes a boolean array `visited` to track visited elements.

It iterates through the elements of the input vector `nums`, marking each element as visited by setting the corresponding index in the `visited` array to true. Upon encountering a visited element, it immediately returns it as the duplicate element.

This approach efficiently identifies the duplicate element in the vector by tracking visited elements without requiring additional space proportional to the size of the input vector.

Complexity

- Runtime: `O(n)`.
- Extra space: much less than `O(n)`. `std::vector<bool>` is optimized for space-efficient.

8.3.5 Solution 3: Marking with std::bitset

Since `n <= 10^5`, you can use this size for a `std::bitset` to do the marking.

Code

```
#include <vector>
#include <iostream>
#include <bitset>
using namespace std;
int findDuplicate(const vector<int>& nums) {
    // initialize visited = '000..0' with 100001 bits 0
    bitset<100001> visited;
    for (auto& a : nums) {
        if (visited[a]) {
            return a;
        }
        // set a-th bit to 1
        visited[a] = 1;
    }
    return 0;
}
int main() {
    vector<int> nums{1,3,4,2,2};
    cout << findDuplicate(nums) << endl;
    nums = {3,1,3,4,2};
    cout << findDuplicate(nums) << endl;
}
```

```
Output:
2
3
```

This code uses a bitset to keep track of visited elements and quickly detects any duplicate element encountered during the iteration.

Complexity

- Runtime: `O(n)`.
- Extra space: `O(1)`.

8.3.6 Key Takeaway

Different strategies can be employed to detect duplicates in an array, such as sorting followed by linear search, using a hash set to track visited elements, or utilizing a bitset to mark visited indices.

Each approach has its trade-offs in terms of time complexity, space complexity, and implementation simplicity. To me, Solution 2 balances all the trade-offs. What do you think?

8.3.7 Exercise

- Missing Number

8.4 Maximum Product of Word Lengths

8.4.1 Problem statement

[1]Given a string array `words`, return the maximum value of `length(word[i]) * length(word[j])` where the two words do not share common letters. If no such two words exist, return `0`.

[1] https://leetcode.com/problems/maximum-product-of-word-lengths/

Example 1

```
Input: words = ["abcw","baz","foo","bar","xtfn","abcdef"]
Output: 16
Explanation: The two words can be "abcw", "xtfn".
```

Example 2

```
Input: words = ["a","ab","abc","d","cd","bcd","abcd"]
Output: 4
Explanation: The two words can be "ab", "cd".
```

Example 3

```
Input: words = ["a","aa","aaa","aaaa"]
Output: 0
Explanation: No such pair of words.
```

Constraints

- `2 <= words.length <= 1000`.
- `1 <= words[i].length <= 1000`.
- `words[i]` consists only of lowercase English letters.

8.4.2 Solution 1: Bruteforce

For each `words[i]`, for all `words[j]` with `j > i`, check if they do not share common letters and compute the product of their lengths.

Code

```
#include <vector>
#include <iostream>
using namespace std;
int maxProduct(const vector<string>& words) {
    int maxP = 0;
    for (int i = 0; i < words.size(); i++) {
        // visited marks all letters that appear in words[i]
        // words[i] consists of only 26 lowercase English letters
        vector<bool> visited(26, false);
        for (auto& c : words[i]) {
            // map 'a'->0, 'b'->1, .., 'z'->25
            visited[c - 'a'] = true;
        }
        // compare with all other words[j]
        for (int j = i + 1; j < words.size(); j++) {
            bool found = false;
            for (auto& c : words[j]) {
                if (visited[c - 'a']) {
                    // this words[j] has common letter with words[i]
                    found = true;
                    break;
                }
            }
            // if words[j] disjoints words[i]
            if (!found) {
                // compute and update max product of their lengths
                maxP = max(maxP, (int) (words[i].length() * words[j].
↪length()));
            }
        }
    }
    return maxP;
}
int main() {
    vector<string> words{"abcw","baz","foo","bar","xtfn","abcdef"};
```

(continues on next page)

(continued from previous page)

```
    cout << maxProduct(words) << endl;
    words = {"a","ab","abc","d","cd","bcd","abcd"};
    cout << maxProduct(words) << endl;
    words = {"a","aa","aaa","aaaa"};
    cout << maxProduct(words) << endl;
}
```

```
Output:
16
4
0
```

This solution checks for common characters between pairs of words to determine their product of lengths.

It iterates through each pair of words in the input vector `words`, maintaining a boolean array `visited` to mark the presence of characters in each word. By comparing the characters of each pair of words, it identifies whether there are any common characters. If no common characters are found, it computes the product of the lengths of the two words and updates the maximum product accordingly.

This approach optimizes the computation of the maximum product by efficiently checking for common characters between pairs of words without requiring additional space proportional to the length of the words.

Complexity

- Runtime: `O(n^2 * m)`.
- Extra space: `O(1)`.

8.4.3 Solution 2: Checking common letters using bit masking

You can map a `words[i]` to the bit representation of an integer `n` by their characters like the following:

- If the word `words[i]` contains the letter `'a'`, the bit at position `0` of `n` is `1`.
- If the word `words[i]` contains the letter `'b'`, the bit at position `1` of `n` is `1`.
- ...
- If the word `words[i]` contains the letter `'z'`, the bit at position `25` of `n` is `1`.

Then to check if two words have common letters, you just perform the bitwise operator AND on them.

Example 1:

- The word `"abcw"` is mapped to `00010000000000000000000111`.
- The word `"baz"` is mapped to `10000000000000000000000011`.
- `"abcw" & "baz" = 00000000000000000000000011`. This value is not zero, which means they have common letters.

This technique is called bit masking.

Code

```
#include <vector>
#include <iostream>
using namespace std;
int maxProduct(const vector<string>& words) {
    int maxP = 0;
    // initialize all elements of mask to 0
    vector<int> mask(words.size());
    for (int i = 0; i < words.size(); i++) {
        // mark all characters of word[i]
        for (auto& c : words[i]) {
            // map 'a'->0, 'b'->1, .., 'z'->25
```

(continues on next page)

(continued from previous page)

```
            // set the bit at that mapped position of mask[i] to 1
            mask[i] |= 1 << (c - 'a');
        }
        for (int j = 0; j < i; j++) {
            if ((mask[j] & mask[i]) == 0) {
                // there is no common bit between mask[j] and mask[i]
                maxP = max(maxP, (int) (words[i].length() * words[j].
↪length()));
            }
        }
    }
    return maxP;
}
int main() {
    vector<string> words{"abcw","baz","foo","bar","xtfn","abcdef"};
    cout << maxProduct(words) << endl;
    words = {"a","ab","abc","d","cd","bcd","abcd"};
    cout << maxProduct(words) << endl;
    words = {"a","aa","aaa","aaaa"};
    cout << maxProduct(words) << endl;
}
```

```
Output:
16
4
0
```

This solution represents each word in the input vector `words` as a bitmask, where each bit represents the presence or absence of a character in the word.

By iterating through the words and constructing their corresponding bitmasks, it encodes the character information. Then, by comparing the bitmasks of pairs of words, it identifies whether there are any common characters between them. If no common characters are found, it computes the product of the lengths of the two words and updates the maximum product accordingly.

This approach optimizes the computation of the maximum product by using bitwise operations to efficiently check for common characters between pairs of words

without requiring additional space proportional to the length of the words.

Complexity

- Runtime: `O(n^2)`, where `n` is the number of words in the input vector.
- Extra space: `O(n)`.

8.4.4 Tips

Utilizing bit manipulation techniques, such as bitmasking, can significantly optimize the comparison process for determining common characters between words.

Solution 2 reduces the time complexity compared to brute force methods, particularly when dealing with large datasets, as it avoids nested loops and unnecessary character comparisons.

CHAPTER NINE

SORTING

The arrangement of the elements in an array can hold the key to improved efficiency and a deeper understanding of your code, which is explored in this chapter as it delves into the usage of sorting algorithms.

Sorting is similar to putting puzzle pieces to reveal the overall structure. Rearranging elements makes it possible to retrieve data more quickly, conduct searches more quickly, and even discover patterns and linkages that might otherwise go unnoticed.

What this chapter covers:

1. **Introduction to Sorting:** Establish a strong foundation by understanding the significance of sorting, its impact on algorithmic performance, and the role of ordering in data analysis.
2. **Stability and Uniqueness:** Learn about the concepts of stability and uniqueness in sorting and how they can impact the integrity and usefulness of sorted data.
3. **Insights through Sorting:** Discover scenarios where sorted data provides valuable insights, such as identifying trends, anomalies, or patterns that inform decision-making.

9.1 Majority Element

9.1.1 Problem statement

[1]You're given an array `nums` with a total of `n` elements. Your task is to find and return the majority element.

The majority element is the element that appears more frequently in the array than any other element, specifically, it appears more than `n / 2` times.

You can assume that the majority element always exists in the given array.

Example 1

```
Input: nums = [3,2,3]
Output: 3
```

Example 2

```
Input: nums = [2,2,1,1,1,2,2]
Output: 2
```

Constraints

- `n == nums.length`.
- `1 <= n <= 5 * 10^4`.
- `-2^31 <= nums[i] <= 2^31 - 1`.

[1] https://leetcode.com/problems/majority-element/

Follow-up:

Could you solve the problem in linear time and in `O(1)` space?

9.1.2 Solution 1: Counting the frequency

Code

```
#include <vector>
#include <iostream>
#include <unordered_map>
using namespace std;
int majorityElement(const vector<int>& nums) {
    unordered_map<int,int> freq;
    const int HALF = nums.size() / 2;
    for (auto& a : nums) {
        // count a's occurrences
        freq[a]++;
        if (freq[a] > HALF) {
            return a;
        }
    }
    return nums[0];
}
int main() {
    vector<int> nums{3,2,3};
    cout << majorityElement(nums) << endl;
    nums = {2,2,1,1,1,2,2};
    cout << majorityElement(nums) << endl;
}
```

```
Output:
3
2
```

The code effectively counts the occurrences of each integer in the array and checks if any integer appears more than `n/2` times. If so, it returns that integer as the

majority element; otherwise, it defaults to the first element of the array.

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(n)`.

9.1.3 Solution 2: Sorting and picking the middle element

Code

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int majorityElement(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    return nums[nums.size()/2];
}
int main() {
    vector<int> nums{3,2,3};
    cout << majorityElement(nums) << endl;
    nums = {2,2,1,1,1,2,2};
    cout << majorityElement(nums) << endl;
}
```

```
Output:
3
2
```

This code leverages the property of a majority element, which guarantees that it occupies the middle position in the sorted list of elements. Sorting the array allows us to easily access this middle element.

Complexity

- Runtime: `O(n*logn)`, where `n = nums.length`.
- Extra space: `O(1)`.

9.1.4 Solution 3: Partial sort

Since you are interested in only the middle element after sorting, the partial sorting algorithm `std::nth_element` can be used in this case to reduce the cost of the full sorting.

Code

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int majorityElement(vector<int>& nums) {
    const int mid = nums.size() / 2;
    // rearrange nums such that all elements less than or equal␣
↪nums[mid]
    // are placed before nums[mid]
    nth_element(nums.begin(), nums.begin() + mid, nums.end());
    return nums[mid];
}
int main() {
    vector<int> nums{3,2,3};
    cout << majorityElement(nums) << endl;
    nums = {2,2,1,1,1,2,2};
    cout << majorityElement(nums) << endl;
}
```

```
Output:
3
2
```

The code uses the std::nth_element function to rearrange the elements in the `nums` vector such that the element at index `mid` will be in its correct sorted position, and all elements before it will be less than or equal to it, while all elements after it will be greater than or equal to it.

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(1)`.

9.1.5 Modern C++ tips

In the code of Solution 3, the partial sorting algorithm `std::nth_element` will make sure for all indices `i` and `j` that satisfy `0 <= i <= mid <= j < nums.length`,

```
nums[i] <= nums[mid] <= nums[j].
```

In other words, `nums[mid]` divides the array `nums` into two groups: all elements that are less than or equal to `nums[mid]` and the ones that are greater than or equal to `nums[mid]`.

Those two groups are unsorted. That is why the algorithm is called *partial* sorting.

9.1.6 Exercise

- Most Frequent Even Element

9.2 Merge Sorted Array

9.2.1 Problem statement

[1]You're given two integer arrays, `nums1` and `nums2`, both sorted in non-decreasing order. Additionally, you have two integers, `m` and `n`, representing the number of

[1] https://leetcode.com/problems/merge-sorted-array/

elements in nums1 and nums2, respectively.

Your task is to merge the elements from nums2 into nums1 in a way that the resulting array is sorted in non-decreasing order.

However, the sorted array should not be returned as a separate result. Instead, the merged elements should be stored inside the nums1 array. Here's the setup for that purpose:

- nums1 has a total length of m + n, where the first m elements represent the elements that should be merged, and the last n elements are initialized to 0 and should be ignored.
- The nums2 array has a length of n, representing the elements to be merged from nums2 into the final nums1 array.

Example 1

```
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
Explanation: The arrays we are merging are [1,2,3] and [2,5,6].
The result of the merge is [1,2,2,3,5,6] with the underlined elements␣
↪coming from nums1.
```

Example 2

```
Input: nums1 = [1], m = 1, nums2 = [], n = 0
Output: [1]
Explanation: The arrays we are merging are [1] and [].
The result of the merge is [1].
```

Example 3

```
Input: nums1 = [0], m = 0, nums2 = [1], n = 1
Output: [1]
Explanation: The arrays we are merging are [] and [1].
The result of the merge is [1].
Note that because m = 0, there are no elements in nums1. The 0 is only
↪there to ensure the merge result can fit in nums1.
```

Constraints

- `nums1.length == m + n`.
- `nums2.length == n`.
- `0 <= m, n <= 200`.
- `1 <= m + n <= 200`.
- `-10^9 <= nums1[i], nums2[j] <= 10^9`.

Follow up

- Can you come up with an algorithm that runs in `O(m + n)` time?

9.2.2 Solution 1: Store the result in a new container

Code

```cpp
#include <iostream>
#include <vector>
using namespace std;
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n)
{
    vector<int> result;
    int i = 0;
```

(continues on next page)

(continued from previous page)

```
    int j = 0;
    while (i < m || j < n) {
        if (j == n) {
            // nums2 is done, only nums1 still runs
            result.push_back(nums1[i++]);
        } else if (i == m) {
            // nums1 is done, only nums2 still runs
            result.push_back(nums2[j++]);
        } else if (nums1[i] < nums2[j]) {
            result.push_back(nums1[i++]);
        } else {
            result.push_back(nums2[j++]);
        }
    }
    nums1.swap(result);
}
void printResult(const vector<int>& nums1) {
    cout << "[";
    for (auto& n : nums1) {
        cout << n << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> nums1 = {1,2,3,0,0,0};
    vector<int> nums2 = {2,5,6};
    merge(nums1, 3, nums2, 3);
    printResult(nums1);
    nums1 = {1};
    nums2 = {};
    merge(nums1, 1, nums2, 0);
    printResult(nums1);
    nums1 = {0};
    nums2 = {1};
    merge(nums1, 0, nums2, 1);
    printResult(nums1);
```

(continues on next page)

(continued from previous page)

```
}
```

```
Output:
[1,2,2,3,5,6,]
[1,]
[1,]
```

This solution merges two sorted arrays nums1 and nums2 into nums1 while maintaining sorted order. It iterates through both arrays, comparing elements and adding them to a temporary result vector. After the merging is complete, it replaces the contents of nums1 with the merged result.

Complexity

- Runtime: O(m+n), where m = nums1.length and n = nums2.length.
- Extra space: O(m+n).

9.2.3 Solution 2: Reassigning nums1 backward

Code

```
#include <iostream>
#include <vector>
using namespace std;
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n)
{
    int k = m + n - 1;
    int i = m - 1;
    int j = n - 1;
    while (k >= 0) {
        if (j < 0) {
            // nums2 is done
            nums1[k--] = nums1[i--];
        } else if (i < 0) {
```

(continues on next page)

(continued from previous page)

```
            // nums1 is done
            nums1[k--] = nums2[j--];
        } else if (nums1[i] > nums2[j]) {
            nums1[k--] = nums1[i--];
        } else {
            nums1[k--] = nums2[j--];
        }
    }
}
void printResult(const vector<int>& nums1) {
    cout << "[";
    for (auto& n : nums1) {
        cout << n << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> nums1 = {1,2,3,0,0,0};
    vector<int> nums2 = {2,5,6};
    merge(nums1, 3, nums2, 3);
    printResult(nums1);
    nums1 = {1};
    nums2 = {};
    merge(nums1, 1, nums2, 0);
    printResult(nums1);
    nums1 = {0};
    nums2 = {1};
    merge(nums1, 0, nums2, 1);
    printResult(nums1);
}
```

```
Output:
[1,2,2,3,5,6,]
[1,]
[1,]
```

Complexity

- Runtime: `O(m+n)`, where `m = nums1.length` and `n = nums2.length`.
- Extra space: `O(1)`.

9.2.4 Conclusion

Solution 2 efficiently merges two sorted arrays, `nums1` and `nums2`, into `nums1` while preserving the sorted order. It uses three pointers (`k`, `i`, and `j`) to perform the merge in reverse order, which helps avoid the need for additional space.

9.2.5 Exercise

- Squares of a Sorted Array

9.3 Remove Covered Intervals

9.3.1 Problem statement

[1]You're given an array called `intervals`, where each element `intervals[i]` is a pair `[li, ri]` representing a half-open interval `[li, ri)`.

Your task is to remove all intervals from the list that are completely covered by another interval. An interval `[a, b)` is considered covered by the interval `[c, d)` if and only if `c` is less than or equal to `a`, and `b` is less than or equal to `d`.

After removing the covered intervals, you need to return the number of remaining intervals.

[1] https://leetcode.com/problems/remove-covered-intervals/

Example 1

```
Input: intervals = [[1,4],[3,6],[2,8]]
Output: 2
Explanation: Interval [3,6] is covered by [2,8], therefore it is␣
↪removed.
```

Example 2

```
Input: intervals = [[1,4],[2,3]]
Output: 1
```

Constraints

- `1 <= intervals.length <= 1000`.
- `intervals[i].length == 2`.
- `0 <= li <= ri <= 10^5`.
- All the given intervals are unique.

9.3.2 Solution 1: Bruteforce

For each interval `i`, find if any other interval `j` such that `j` covers `i` or `i` covers `j` then remove the smaller one from `intervals`.

Example 1

For `intervals = [[1,4],[3,6],[2,8]]`.

- With interval `i = [1,4]`, there is no other interval `j` such that covers `i` or `j` covers `i`.
- With interval `i = [3,6]`, there is interval `j = [2,8]` convering `i`. Remove `[3,6]` from `intervals`.

Final intervals = [[1,4],[2,8]].

Code

```
#include <vector>
#include <iostream>
using namespace std;
//! @return true if the interval i is covered by j
inline bool isCovered(const vector<int>& i, const vector<int>& j) {
    return j[0] <= i[0] && i[1] <= j[1];
}
int removeCoveredIntervals(vector<vector<int>>& intervals) {
    int i = 0;
    while (i < intervals.size() - 1) {
        int j = i + 1;
        bool erase_i = false;
        while (j < intervals.size()) {
            if (isCovered(intervals[i], intervals[j])) {
                // remove intervals[i] from intervals
                intervals.erase(intervals.begin() + i);
                erase_i = true;
                break;
            } else if (isCovered(intervals[j], intervals[i])) {
                // remove intervals[j] from intervals
                intervals.erase(intervals.begin() + j);
            } else {
                j++;
            }
        }
        if (!erase_i) {
            i++;
        }
    }
    return intervals.size();
}
int main() {
    vector<vector<int>> intervals{{1,4},{3,6},{2,8}};
```

(continues on next page)

(continued from previous page)

```
    cout << removeCoveredIntervals(intervals) << endl;
    intervals = {{1,4},{2,3}};
    cout << removeCoveredIntervals(intervals) << endl;
}
```

```
Output:
2
1
```

This solution effectively removes covered intervals and retains only those that do not have others covering them. The time complexity of this solution is `O(N^3)`, where `N` is the number of intervals, as it involves nested loops and potential removal of intervals from the list.

Complexity

- Runtime: `O(N^3)`, where `N = intervals.length`.
- Extra space: `O(1)`.

9.3.3 Solution 2: Using dictionary order

You might know how to look up words in a dictionary.

The word `apple` appears before `candy` in the dictionary because the starting letter `a` of `apple` appears before `c` of `candy` in the English alphabet.

And `apple` appears after `animal` since the next letter `p` appears after `n`.

The C++ Standard Library uses that dictionary order to compare two `std::vectors`.

Example 1

Rewriting `intervals = [[1,4],[3,6],[2,8]]` in dictionary order you get `intervals = [[1,4],[2,8],[3,6]]`. In this order, the left bounds of the `intervals` are sorted first.

If `intervals` is sorted like that, you can avoid bruteforce in Solution 1 by a simpler algorithm.

Check if each interval `i` covers or is covered by some of the previous ones.

Remember that the left bound of interval `i` is always bigger than or equal to all left bounds of the previous ones. So,

1. `i` is covered by some previous interval if the right bound of `i` is less than some of the right bounds before.
2. Otherwise `i` can only cover its exact previous one that has the same left bound.

Code

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int removeCoveredIntervals(vector<vector<int>>& intervals) {
    // sort the intervals using dictionary order
    sort(intervals.begin(), intervals.end());
    // count the intervals to be removed
    int count = 0;
    // keep track max right bound of all previous intervals
    int maxRight = -1;
    // log the left bound of the previous interval
    int preLeft = -1;
    for (auto& i : intervals) {
        if (i[1] <= maxRight) {
            // i's right bound is less than some previous one's
            count++;
        } else if (i[0] == preLeft) {
```

(continues on next page)

(continued from previous page)

```
            // i's left bound is the same as exact previous one's
            count++;
        } else {
            // update previous interval's left bound
            preLeft = i[0];
        }
        // update max right bound
        maxRight = max(maxRight, i[1]);
    }
    return intervals.size() - count;
}
int main() {
    vector<vector<int>> intervals{{1,4},{3,6},{2,8}};
    cout << removeCoveredIntervals(intervals) << endl;
    intervals = {{1,4},{2,3}};
    cout << removeCoveredIntervals(intervals) << endl;
}
```

```
Output:
2
1
```

Complexity

- Runtime: `O(N*logN)`, where `N = intervals.length`.
- Extra space: `O(1)`.

9.3.4 Key takeaway

- Two `std::vector`s can be compared using dictionary order.
- Solution 2 first sorts the intervals and then iterates through them while keeping track of whether each interval is covered by others or not. The final result is the count of intervals that are not covered.

9.4 My Calendar I

9.4.1 Problem statement

[1]You're creating a program to use as your calendar. You can add new events to the calendar, but only if adding the event will not lead to a double booking.

A double booking occurs when two events have some time overlap, meaning there's a shared time period between them.

An event is represented as a pair of integers: `start` and `end`, which represent the booking on a half-open interval `[start, end)`. This interval includes all real numbers `x` such that `start <= x < end`.

You need to implement the `MyCalendar` class, which has the following functions:

1. `MyCalendar()`: Initializes the calendar object.
2. `boolean book(int start, int end)`: This function checks if the event with the given `start` and `end` can be added to the calendar without causing a double booking. If it's possible to add the event without a double booking, the function returns `true`. Otherwise, it returns `false`, and the event is not added to the calendar.

Example 1

```
Input
["MyCalendar", "book", "book", "book"]
[[], [10, 20], [15, 25], [20, 30]]
Output
[null, true, false, true]

Explanation
MyCalendar myCalendar = new MyCalendar();
myCalendar.book(10, 20); // return True
myCalendar.book(15, 25); // return False. It can not be booked because time 15 is already booked by another event.
```

(continues on next page)

[1] https://leetcode.com/problems/my-calendar-i/

(continued from previous page)

```
myCalendar.book(20, 30); // return True, The event can be booked, as␣
↪the first event takes every time less than 20, but not including 20.
```

Constraints

- `0 <= start < end <= 10^9.`
- At most `1000` calls will be made to book.

9.4.2 Solution 1: Vector

You can store the booked events in a vector and check the intersection condition whenever you add a new event.

Code

```
#include <iostream>
#include <vector>
using namespace std;
class MyCalendar {
    private:
    vector<pair<int,int>> _events;
public:
    MyCalendar() {}
    bool book(int start, int end) {
        for (auto& e : _events) {
            // check for overlap
            if (!(e.second <= start || end <= e.first)) {
                return false;
            }
        }
        _events.push_back({start, end});
        return true;
    }
```

(continues on next page)

(continued from previous page)

```
};
int main() {
    MyCalendar c;
    std::cout << c.book(10, 20) << std::endl;
    std::cout << c.book(15, 25) << std::endl;
    std::cout << c.book(20, 30) << std::endl;
}
```

```
Output:
1
0
1
```

This code essentially maintains a list of events and checks for overlaps when booking a new event. If no overlaps are found, it adds the new event to the list and allows the booking.

Complexity

For the `book` method:

- Runtime: `O(n)`, where `n= _events.length`.
- Extra space: `O(1)`.

9.4.3 Solution 2: Set

Since the events have no intersection, they can be sorted. You can also consider two events to be the same if they intersect.

With that in mind, you can use std::set to store the sorted unique events.

Code

```
#include <iostream>
#include <set>
using namespace std;
using Event = pair<int,int>;
struct EventCmp {
    bool operator()(const Event& lhs, const Event& rhs) const {
        return lhs.second <= rhs.first;
    }
};
class MyCalendar {
    private:
    // declare a set with custom comparison operator
    set<Event, EventCmp> _events;
public:
    MyCalendar() {}
    bool book(int start, int end) {
        auto result = _events.insert({start, end});
        // result.second stores a bool indicating
        // if the insertion was actually performed
        return result.second;
    }
};
int main() {
    MyCalendar c;
    std::cout << c.book(10, 20) << std::endl;
    std::cout << c.book(15, 25) << std::endl;
    std::cout << c.book(20, 30) << std::endl;
}
```

```
Output:
1
0
1
```

Complexity

For the `book` method:

- Runtime: `O(logn)`, where `n = _events.length`.
- Extra space: `O(1)`.

9.4.4 Key Takeaway

Solution 2 efficiently handles event bookings by maintaining a sorted set of events based on their end times, allowing for quick overlap checks when booking new events.

9.4.5 Exercise

- Determine if Two Events Have Conflict

9.5 Remove Duplicates from Sorted Array II

9.5.1 Problem statement

[1]Given an integer array `nums` already sorted in non-decreasing order, you must remove duplicates so that each unique element appears at most twice. The relative order of the elements should remain unchanged.

Since changing the array's length in some programming languages is impossible, you must place the result in the first part of the `nums` array. In other words, if there are `k` elements after removing the duplicates, the first `k` elements of `nums` should contain the final result. Anything beyond the first `k` elements is not important.

You should return the value of `k` after placing the final result in the first `k` slots of the `nums` array.

[1] https://leetcode.com/problems/remove-duplicates-from-sorted-array-ii/

The key requirement is to accomplish this task without using extra space for another array. It must be done by modifying the input array `nums` in-place, using only `O(1)` extra memory.

Example 1

```
Input: nums = [1,1,1,2,2,3]
Output: 5, nums = [1,1,2,2,3,_]
Explanation: Your function should return k = 5, with the first five␣
↪elements of nums being 1, 1, 2, 2, and 3, respectively.
What you leave does not matter beyond the returned k (hence, they are␣
↪underscores).
```

Example 2

```
Input: nums = [0,0,1,1,1,1,2,3,3]
Output: 7, nums = [0,0,1,1,2,3,3,_,_]
Explanation: Your function should return k = 7, with the first seven␣
↪elements of nums being 0, 0, 1, 1, 2, 3, and 3, respectively.
What you leave does not matter beyond the returned k (hence, they are␣
↪underscores).
```

Constraints

- `1 <= nums.length <= 3 * 10^4`.
- `-10^4 <= nums[i] <= 10^4`.
- `nums` is sorted in **non-decreasing** order.

9.5.2 Solution 1: Erasing the duplicates

In order for each unique element to appear **at most twice**, you have to erase the further appearances if they exist.

Since the array `nums` is sorted, you can determine that existence by checking if `nums[i] == nums[i-2]` for `2 <= i < nums.length`.

Code

```
#include <vector>
#include <iostream>
using namespace std;
int removeDuplicates(vector<int>& nums) {
    int i = 2;
    while (i < nums.size()) {
        // find the element appearing more than twice
        if (nums[i] == nums[i-2]) {
            int j = i;
            // find all duplicates
            while (j < nums.size() && nums[j] == nums[i]) {
                j++;
            }
            // keep nums[i-2] and nums[i-1] remove all later duplicates
            nums.erase(nums.begin() + i, nums.begin() + j);
        } else {
            i++;
        }
    }
    return nums.size();
}
void printResult(const int k, const vector<int>& nums) {
    cout << k << ", [";
    for (int i = 0; i < k ; i++) {
        cout << nums[i] << ",";
    }
    cout << "]\n";
```

(continues on next page)

(continued from previous page)

```
}
int main() {
    vector<int> nums{1,1,1,2,2,3};
    printResult(removeDuplicates(nums), nums);
    nums = {0,0,1,1,1,1,2,3,3};
    printResult(removeDuplicates(nums), nums);
}
```

```
Output:
5, [1,1,2,2,3,]
7, [0,0,1,1,2,3,3,]
```

This solution efficiently removes duplicates from the sorted array by checking for duplicates and erasing the excess occurrences while preserving two instances of each unique element. It then returns the length of the modified array.

Complexity

- Runtime:
  - Worst case: `O(N^2/3)`, where `N = nums.size()`. The complexity of the `erase()` method is linear in `N`. The worst case is when `erase()` is called maximum `N/3` times.

```
Example of the worst case:
nums = [1,1,1,2,2,2,3,3,3,4,4,4,5,5,5,6,6,6].
```

  - On average: `O(N)` since the number of `erase()` calls is `O(1)`.
- Extra space: `O(1)`.

9.5.3 Solution 2: Reassigning the satisfying elements

You might need to avoid the `erase()` method in the solution above to reduce the complexity. Moreover, after removing the duplicates, the problem only cares about the first `k` elements of the array `nums`.

If you look at the final result after removing duplication, the **expected** `nums` satisfies

```
nums[i] > nums[i-2] for 2 <= i < nums.length.
```

You can use this invariant to **reassign** the array `nums` only the satisfied elements.

Code

```
#include <vector>
#include <iostream>
using namespace std;
int removeDuplicates(vector<int>& nums) {
    if (nums.size() <= 2) {
        return nums.size();
    }
    int k = 2;
    int i = 2;
    while (i < nums.size()) {
        if (nums[i] > nums[k - 2]) {
            // make sure nums[k] != nums[k-2]
            nums[k++] = nums[i];
        }
        i++;
    }
    return k;
}
void printResult(const int k, const vector<int>& nums) {
    cout << k << ", [";
    for (int i = 0; i < k ; i++) {
        cout << nums[i] << ",";
    }
```

(continues on next page)

(continued from previous page)

```
    cout << "]\n";
}
int main() {
    vector<int> nums{1,1,1,2,2,3};
    printResult(removeDuplicates(nums), nums);
    nums = {0,0,1,1,1,1,2,3,3};
    printResult(removeDuplicates(nums), nums);
}
```

```
Output:
Output:
5, [1,1,2,2,3,]
7, [0,0,1,1,2,3,3,]
```

Complexity

- Runtime: `O(N)`, where `N = nums.size()`.
- Extra space: `O(1)`.

9.5.4 Conclusion

Solution 2 effectively modifies the input array in-place, removing duplicates that occur more than twice while maintaining the desired order of unique elements. It does so in a single pass through the array, resulting in a time complexity of `O(N)`, where `N` is the number of elements in the array.

9.5.5 Exercise

- Remove Duplicates from Sorted Array

CHAPTER

TEN

GREEDY ALGORITHM

This chapter will explore a fascinating and highly practical problem-solving approach known as **greedy algorithms**. Greedy algorithms are powerful tools for making decisions at each step of an optimization problem, often leading to efficient and near-optimal solutions.

In this chapter, we'll dive deep into the world of greedy algorithms, learning how to apply them to a wide range of real-world scenarios. Here's what you can look forward to:

1. **Understanding Greedy Algorithms**: We'll begin by establishing a solid foundation in greedy algorithms. You'll understand this approach's key principles, advantages, and limitations.
2. **The Greedy Choice Property**: Discover the core characteristic of greedy algorithms—the greedy choice property. Learn how it guides us in making locally optimal decisions at each step.
3. **Greedy for Searching**: Greedy algorithms can also be applied to search problems. We'll delve into graph traversal algorithms and heuristic search methods.
4. **Exercises and Problems**: Reinforce your understanding of greedy algorithms with exercises and LeetCode problems covering a wide range of greedy-based challenges. Practice is essential for mastering this problem-solving technique.

By the end of this chapter, you'll have a comprehensive understanding of greedy algorithms and the ability to apply them to a wide range of problems, from optimization to search. Greedy algorithms are valuable tools in your problem-solving toolkit, and this chapter will equip you with the skills needed to confidently tackle

complex optimization challenges. So, let's dive in and explore the world of greedy algorithms!

10.1 Can Place Flowers

10.1.1 Problem statement

[1]You are presented with a long flowerbed containing plots, some of which are planted with flowers (denoted by `1`) and some are empty (denoted by `0`). Flowers cannot be planted in **adjacent** plots. You are given an integer array `flowerbed` representing the layout of the flowerbed, and an integer `n` representing the number of new flowers you want to plant.

Your task is to determine if it is possible to plant `n` new flowers in the `flowerbed` without violating the rule of no-adjacent-flowers. If it is possible, return `true`; otherwise, return `false`.

Example 1

```
Input: flowerbed = [1,0,0,0,1], n = 1
Output: true
```

Example 2

```
Input: flowerbed = [1,0,0,0,1], n = 2
Output: false
```

[1] https://leetcode.com/problems/can-place-flowers/

Constraints

- `1 <= flowerbed.length <= 2 * 10^4`.
- `flowerbed[i]` is `0` or `1`.
- There are no two adjacent flowers in `flowerbed`.
- `0 <= n <= flowerbed.length`.

10.1.2 Solution: Check the no-adjacent-flowers rule

A new flower can be planted at position `i` only if

```
flowerbed[i - 1] == 0 && flowerbed[i] == 0 && flowerbed[i + 1] == 0.
```

If the condition is satisfied, the flower can be planted at position `i`. `flowerbed[i]` is now assigned to `1`. Then you can skip checking the rule for the position `i + 1` and go directly to position `i + 2`.

Code

```
#include <iostream>
#include <vector>
using namespace std;
bool canPlaceFlowers(vector<int>& flowerbed, int n) {
    if (n == 0) {
        return true;
    }
    flowerbed.insert(flowerbed.begin(), 0);
    flowerbed.push_back(0);
    int i = 1;
    while (i < flowerbed.size() - 1) {
        if (flowerbed[i - 1] == 0 && flowerbed[i] == 0 && flowerbed[i +
↪1] == 0) {
            // plant i if it satisfies the no-adjacent condition
            flowerbed[i] = 1;
            n--;
```

(continues on next page)

(continued from previous page)

```
            i+=2;
        } else {
            i++;
        }
    }
    return n <= 0; // have planted all n
}
int main() {
    vector<int> flowerbed{1,0,0,0,1};
    cout << canPlaceFlowers(flowerbed, 1) << endl;
    flowerbed = {1,0,0,0,1};
    cout << canPlaceFlowers(flowerbed, 2) << endl;
}
```

```
Output:
1
0
```

This solution efficiently iterates through the flowerbed, planting flowers wherever possible while adhering to the constraints. It returns `true` if it's possible to plant all the required flowers and `false` otherwise.

Complexity

- Runtime: `O(N)`, where `N = flowerbed.length`.
- Extra space: `O(1)`.

10.1.3 Implementation tips

- In this implementation, you could insert element `0` to the front and the back of vector `flowerbed` to avoid writing extra code for checking the no-adjacent-flowers rule at `i = 0` and `i = flowerbed.size() - 1`.
- There are a few ways to insert an element to a vector. Here you can see an example of using the methods `insert` and `push_back` of a `std::vector`.

10.1.4 Ecercise

- Teemo Attacking

10.2 Minimum Deletions to Make Character Frequencies Unique

10.2.1 Problem statement

[1]A string s is considered "good" if there are no two different characters in the string that have the same frequency, meaning each character appears a unique number of times.

You're given a string s, and your task is to determine the minimum number of characters you need to delete from s to make it a "good" string.

The frequency of a character in a string is the count of times that character appears in the string. For instance, in the string "aab", the frequency of 'a' is 2, and the frequency of 'b' is 1.

Example 1

```
Input: s = "aab"
Output: 0
Explanation: s is already good.
```

Example 2

```
Input: s = "aaabbbcc"
Output: 2
Explanation: You can delete two 'b's resulting in the good string
```

(continues on next page)

[1] https://leetcode.com/problems/minimum-deletions-to-make-character-frequencies-unique/

(continued from previous page)

```
↪"aaabcc".
Another way is to delete one 'b' and one 'c' resulting in the good␣
↪string "aaabbc".
```

Example 3

```
Input: s = "ceabaacb"
Output: 2
Explanation: You can delete both 'c's resulting in the good string
↪"eabaab".
Note that we only care about characters that are still in the string at␣
↪the end (i.e. frequency of 0 is ignored).
```

Constraints

- `1 <= s.length <= 10^5`.
- `s` contains only lowercase English letters.

10.2.2 Solution: Delete the frequencies in sorted order

Your goal is to make all the frequencies be different.

One way of doing that is by sorting the frequencies and performing the deletion.

Example 4

For `s = "ceaacbb"`, the frequencies of the characters are: `freq['a'] = 2, freq['b'] = 2, freq['c'] = 2` and `freq['e'] = 1`. They are already in sorted order.

- Let the current frequency be the first frequency `freq['a'] = 2`.
- The next frequency is `freq['b'] = 2`, equal to the current frequency. Delete one appearance to make the current frequency be `1`.

- The next frequency is freq['c'] = 2, bigger than the current frequency. Delete two appearances to make the current frequency be 0.
- Because the current frequency is 0, delete all appearances of the remaining frequencies, which is freq['e'] = 1.
- In total, there are 4 deletions.

Code

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int minDeletions(string& s) {
    // map 'a'->0, 'b'->1, ..,'z'->25
    vector<int> freq(26, 0);
    for (char& c: s) {
        // count the frequency of character c
        freq[c - 'a']++;
    }
    // sort freq in descending order
    sort(freq.begin(), freq.end(), greater<int>());
    int deletion = 0;
    int currentFreq = freq.at(0); // start with the max frequency
    for (int i = 1; i < freq.size() && freq.at(i) > 0; i++) {
        if (currentFreq == 0) {
            // delete all remaining characters
            deletion += freq.at(i);
        } else if (freq[i] >= currentFreq) {
            // delete just enough to make the freq[i] < currentFreq
            deletion += freq.at(i) - currentFreq + 1;
            currentFreq--;
        } else {
            // do not delete on freq[i] < currentFreq
            currentFreq = freq.at(i);
        }
    }
```

(continues on next page)

(continued from previous page)

```
    return deletion;
}
int main() {
    cout << minDeletions("aab") << endl;
    cout << minDeletions("aaabbbcc") << endl;
    cout << minDeletions("ceabaacb") << endl;
}
```

```
Output:
0
2
2
```

Complexity

- Runtime: `O(N)`, where `N = s.length`;
- Extra space: `O(1)`.

10.2.3 Conclusion

The problem of determining the minimum number of deletions required to make character frequencies unique can be efficiently solved by counting the frequencies of characters and iteratively adjusting the frequencies to ensure uniqueness.

This solution achieves this by first counting the frequencies of characters and then sorting them in descending order. By iteratively processing the sorted frequencies, the solution ensures that each character frequency is unique while minimizing the number of deletions required.

10.2.4 Exercise

- Minimum Deletions to Make Array Beautiful

10.3 Wiggle Subsequence

10.3.1 Problem statement

[1]A **wiggle sequence** is a sequence where the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with one element and a sequence with two non-equal elements are trivially wiggle sequences.

- For example, `[1, 7, 4, 9, 2, 5]` is a wiggle sequence because the differences `(6, -3, 5, -7, 3)` alternate between positive and negative.
- In contrast, `[1, 4, 7, 2, 5]` and `[1, 7, 4, 5, 5]` are not wiggle sequences. The first is not because its first two differences are positive, and the second is not because its last difference is zero.

A **subsequence** is obtained by deleting some elements (possibly zero) from the original sequence, leaving the remaining elements in their original order.

Given an integer array `nums`, return the length of the longest wiggle subsequence of `nums`.

Example 1

```
Input: nums = [1,7,4,9,2,5]
Output: 6
Explanation: The entire sequence is a wiggle sequence with differences␣
↪(6, -3, 5, -7, 3).
```

[1] https://leetcode.com/problems/wiggle-subsequence/

Example 2

```
Input: nums = [1,17,5,10,13,15,10,5,16,8]
Output: 7
Explanation: There are several subsequences that achieve this length.
One is [1, 17, 10, 13, 10, 16, 8] with differences (16, -7, 3, -3, 6, -
↪8).
```

Example 3

```
Input: nums = [1,2,3,4,5,6,7,8,9]
Output: 2
```

Constraints

- `1 <= nums.length <= 1000`.
- `0 <= nums[i] <= 1000`.

Follow up

- Could you solve this in `O(n)` time?

10.3.2 Solution: Counting the local extrema of `nums`

First, if you pick all local extrema (minima and maxima) of `nums` to form a subsequence `e`, then it is wiggle. Let us call it an **extrema** subsequence.

Example 2

For `nums = [1,17,5,10,13,15,10,5,16,8]`, the local extrema are `[1,17,5,15,5,16,8]`. It is wiggle and called **extrema** subsequence.

Note that if `nums.length = n` then `nums[0]` and `nums[n - 1]` are always the first and the last extremum.

Second, given any two successive local extrema `a` and `b`, you cannot have any wiggle subsequence between them. Because the elements between them are either monotonic increasing or decreasing.

That proves the extrema subsequence is the longest wiggle one.

Code

```
#include <iostream>
#include <vector>
using namespace std;
int wiggleMaxLength(const vector<int>& nums) {
    // nums[0] is always the first extremum
    // start to find the second extremum
    int i = 1;
    while (i < nums.size() && nums[i] == nums[i - 1]) {
        i++;
    }
    if (i == nums.size()) {
        // all nums[i] are equal
        return 1;
    }
    int sign = nums[i] > nums[i - 1] ? 1 : -1;
    int count = 2;
    i++;
    while (i < nums.size()) {
        if ((nums[i] - nums[i - 1]) * sign < 0) {
            // nums[i] is an extremum
            count++;
            sign = -sign;
```

(continues on next page)

(continued from previous page)

```
        }
        i++;
    }
    return count;
}
int main() {
    vector<int> nums{1,7,4,9,2,5};
    cout << wiggleMaxLength(nums) << endl;
    nums = {1,17,5,10,13,15,10,5,16,8};
    cout << wiggleMaxLength(nums) << endl;
    nums = {1,2,3,4,5,6,7,8,9};
    cout << wiggleMaxLength(nums) << endl;
}
```

```
Output:
6
7
2
```

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(1)`.

10.3.3 Conclusion

The problem of finding the length of the longest wiggle subsequence can be efficiently solved using a greedy approach. The solution iterates through the input array, identifying alternating extremums (peaks and valleys) to form the wiggle subsequence.

By keeping track of the current trend direction (increasing or decreasing), the solution efficiently identifies extremums and increments the count accordingly. This greedy approach ensures that each extremum contributes to increasing the length of the wiggle subsequence, maximizing its overall length.

10.4 Partitioning Into Minimum Number Of Deci-Binary Numbers

10.4.1 Problem statement

[1]A decimal number is called deci-binary if each of its digits is either `0` or `1` without any leading zeros. For example, `101` and `1100` are deci-binary, while `112` and `3001` are not.

Given a string `n` that represents a positive decimal integer, return the minimum number of positive deci-binary numbers needed so that they sum up to `n`.

Example 1

```
Input: n = "32"
Output: 3
Explanation: 10 + 11 + 11 = 32
```

Example 2

```
Input: n = "82734"
Output: 8
```

Example 3

```
Input: n = "27346209830709182346"
Output: 9
```

[1] https://leetcode.com/problems/partitioning-into-minimum-number-of-deci-binary-numbers/

Constraints

- `1 <= n.length <= 10^5`.
- `n` consists of only digits.
- `n` does not contain any leading zeros and represents a positive integer.

10.4.2 Solution: Identify the maximum digit of n

Any digit `d` can be obtained by summing the digit `1` `d` times.

The problem turns into identifying the maximum digit of `n`.

Example 2

For `n = "82734"` the answer is 8 because:

```
  82734
= 11111
+ 11111
+ 10111
+ 10101
+ 10100
+ 10100
+ 10100
+ 10000
```

Code

```
#include <iostream>
using namespace std;
int minPartitions(const string& n) {
    char maxDigit = '0';
    for (auto& d : n) {
        maxDigit = max(maxDigit, d);
```

(continues on next page)

(continued from previous page)

```
    }
    return maxDigit - '0';
}
int main() {
    cout << minPartitions("32") << endl;
    cout << minPartitions("82734") << endl;
    cout << minPartitions("27346209830709182346") << endl;
}
```

```
Output:
3
8
9
```

Complexity

- Runtime: `O(N)`, where `N = n.length`.
- Extra space: `O(1)`.

10.4.3 Conclusion

This problem can be efficiently solved by identifying the maximum digit in the string. Since each deci-binary number can only contain digits from 0 to 9, the maximum digit determines the minimum number of deci-binary numbers needed.

By finding the maximum digit in the string and converting it to an integer, the solution effectively determines the minimum number of deci-binary numbers required.

10.5 Maximum Units on a Truck

10.5.1 Problem statement

[1]You are assigned to put some amount of boxes onto one truck. You are given a 2D array `boxTypes`, where `boxTypes[i] = [numberOfBoxes_i, numberOfUnitsPerBox_i]`:

- `numberOfBoxes_i` is the number of boxes of type `i`.
- `numberOfUnitsPerBox_i` is the number of units in each box of the type `i`.

You are also given an integer `truckSize`, which is the maximum number of boxes that can be put on the truck. You can choose any boxes to put on the truck as long as the number of boxes does not exceed `truckSize`.

Return the maximum total number of units that can be put on the truck.

Example 1

```
Input: boxTypes = [[1,3],[2,2],[3,1]], truckSize = 4
Output: 8
Explanation: There are:
- 1 box of the first type that contains 3 units.
- 2 boxes of the second type that contain 2 units each.
- 3 boxes of the third type that contain 1 unit each.
You can take all the boxes of the first and second types, and one box␣
↪of the third type.
The total number of units will be = (1 * 3) + (2 * 2) + (1 * 1) = 8.
```

[1] https://leetcode.com/problems/maximum-units-on-a-truck/

Example 2

```
Input: boxTypes = [[5,10],[2,5],[4,7],[3,9]], truckSize = 10
Output: 91
Explanation: (5 * 10) + (3 * 9) + (2 * 7) = 91.
```

Constraints

- `1 <= boxTypes.length <= 1000.`
- `1 <= numberOfBoxes_i, numberOfUnitsPerBox_i <= 1000.`
- `1 <= truckSize <= 10^6.`

10.5.2 Solution: Greedy algorithm

Put the boxes having more units first.

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
    // sort for the boxes based on their number of units
    sort(boxTypes.begin(), boxTypes.end(),
         [](const vector<int>& a, const vector<int>& b) {
            return a[1] > b[1];
         });
    int maxUnits = 0;
    int i = 0;
    while (truckSize > 0 && i < boxTypes.size()) {
        if (boxTypes[i][0] <= truckSize) {
            // put all boxTypes[i] if there is still room
```

(continues on next page)

(continued from previous page)

```
            maxUnits += boxTypes[i][0] * boxTypes[i][1];
            truckSize -= boxTypes[i][0];
        } else {
            // can put only truckSize < boxTypes[i][0] of boxTypes[i]
            maxUnits += truckSize * boxTypes[i][1];
            break;
        }
        i++;
    }
    return maxUnits;
}
int main() {
    vector<vector<int>> boxTypes{{1,3},{2,2},{3,1}};
    cout << maximumUnits(boxTypes, 4) << endl;
    boxTypes = {{5,10},{2,5},{4,7},{3,9}};
    cout << maximumUnits(boxTypes, 10) << endl;
}
```

```
Output:
8
91
```

This solution optimally loads boxes onto a truck to maximize the total number of units that can be transported, considering both the number of boxes available and their units per box.

Complexity

- Runtime: `O(N*logN)`, where `N = boxTypes.length`.
- Extra space: `O(1)`.

10.5.3 Modern C++ STL notes

Note that two `vectors` can be compared. That is why you can sort them.

But in this case you want to sort them based on the number of units. That is why you need to define the comparison function like the code above. Otherwise, the std::sort algorithm will use the dictionary order to sort them by default.

10.5.4 Exercise

- Maximum Bags With Full Capacity of Rocks

CHAPTER

ELEVEN

DYNAMIC PROGRAMMING

This chapter explains **dynamic programming**, a method for solving complex problems with strategic optimization. Elegant and efficient solutions can be found by breaking down problems into smaller subproblems and using memorization and recursion. It's like solving a puzzle by solving smaller pieces and putting them together to form the larger picture.

What this chapter covers:

1. **Introduction to Dynamic Programming:** Establish a solid foundation by understanding the core principles of dynamic programming, its advantages, and the problems it best suits.
2. **Overlapping Subproblems and Optimal Substructure:** Delve into the key concepts that underlie dynamic programming, namely identifying overlapping subproblems and exploiting optimal substructure.
3. **Fibonacci Series and Beyond:** Begin with classic examples like solving the Fibonacci series and gradually progress to more intricate problems that involve complex optimization.
4. **Efficiency and Trade-offs:** Understand the trade-offs involved in dynamic programming, including the balance between time and space complexity.
5. **Problem-Solving Strategies:** Develop systematic strategies for approaching dynamic programming problems, from identifying subproblems to deriving recurrence relations.

11.1 Fibonacci Number

11.1.1 Problem statement

[1]The Fibonacci numbers make up a sequence denoted as `F(n)`, known as the Fibonacci sequence. Each number in this sequence is the sum of the two preceding numbers, with the sequence starting from `0` and `1`. In other words:

```
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.
```

Your task is to calculate the value of `F(n)` given an integer `n`.

Example 1

```
Input: n = 2
Output: 1
Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.
```

Example 2

```
Input: n = 3
Output: 2
Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.
```

Example 3

```
Input: n = 4
Output: 3
Explanation: F(4) = F(3) + F(2) = 2 + 1 = 3.
```

[1] https://leetcode.com/problems/fibonacci-number/

Constraints

- `0 <= n <= 30`.

11.1.2 Solution 1: Recursive

Code

```
#include <iostream>
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
int main() {
    std::cout << fib(2) << std::endl;
    std::cout << fib(3) << std::endl;
    std::cout << fib(4) << std::endl;
}
```

```
Output:
1
2
3
```

This solution computes the nth Fibonacci number using a recursive approach.

Complexity

The time complexity of this solution is exponential, specifically `O(2^n)`. This is because it repeatedly makes two recursive calls for each `n`, resulting in an exponential number of function calls and calculations. As `n` grows larger, the execution time increases significantly.

The space complexity of the given recursive Fibonacci solution is `O(n)`. This space complexity arises from the function call stack when making recursive calls.

When you call the `fib` function with a value of `n`, it generates a call stack with a depth of `n`, as each call to `fib` leads to two more recursive calls (one for `n - 1` and one for `n - 2`) until `n` reaches the base cases (0 or 1). The space required to store the function call stack is proportional to the depth of the recursion, which is `n`.

Therefore, the space complexity is linearly related to the input value `n`, making it `O(n)`. This can be a concern for large values of `n` because it could lead to a stack overflow if `n` is too large.

- Runtime: `O(2^n)`.
- Extra space: `O(n)`.

11.1.3 Solution 2: Dynamic programming

```
#include <iostream>
#include <vector>
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    // store all computed Fibonacci numbers
    std::vector<int> f(n + 1);
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++) {
        f[i] = f[i - 1] + f[i - 2];
    }
    return f[n];
}
int main() {
    std::cout << fib(2) << std::endl;
    std::cout << fib(3) << std::endl;
    std::cout << fib(4) << std::endl;
}
```

```
Output:
1
```

(continues on next page)

(continued from previous page)

```
2
3
```

This solution uses dynamic programming to avoid redundant calculations by storing and reusing previously computed Fibonacci numbers.

Complexity

- Runtime: `O(n)`.
- Extra space: `O(n)`.

11.1.4 Solution 3: Reduce space for dynamic programming

Code

```cpp
#include <iostream>
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    // store only two previous Fibonacci numbers
    int f0 = 0;
    int f1 = 1;
    for (int i = 2; i <= n; i++) {
        int f2 = f1 + f0;
        // update for next round
        f0 = f1;
        f1 = f2;
    }
    return f1;
}
int main() {
    std::cout << fib(2) << std::endl;
    std::cout << fib(3) << std::endl;
```

(continues on next page)

(continued from previous page)

```
    std::cout << fib(4) << std::endl;
}
```

```
Output:
1
2
3
```

This solution calculates the nth Fibonacci number iteratively using two variables to keep track of the last two Fibonacci numbers.

Complexity

- Runtime: `O(n)`.
- Extra space: `O(1)`.

11.1.5 Conclusion

The Fibonacci sequence can be efficiently computed using various techniques, including recursion with memoization, bottom-up dynamic programming, or even optimizing space usage by storing only the necessary previous Fibonacci numbers.

Solutions 2 and 3 demonstrate dynamic programming approaches, where Fibonacci numbers are computed iteratively while storing intermediate results to avoid redundant computations.

Solution 3 further optimizes space usage by only storing the necessary previous Fibonacci numbers, resulting in a space complexity of `O(1)`. Understanding these different approaches and their trade-offs is essential for selecting the most appropriate solution based on the problem constraints and requirements.

11.1.6 Exercise

- N-th Tribonacci Number

11.2 Unique Paths

11.2.1 Problem statement

[1]A robot starts at the top-left corner of a grid with dimensions `m x n`. It can move either down or right at each step. The robot's goal is to reach the bottom-right corner of the grid.

The problem is to determine the number of unique paths the robot can take to reach the bottom-right corner.

Example 1

```
Input: m = 3, n = 7
Output: 28
```

[1] https://leetcode.com/problems/unique-paths/

Example 2

```
Input: m = 3, n = 2
Output: 3
Explanation:
From the top-left corner, there are a total of 3 ways to reach the
bottom-right corner:
1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down
```

Example 3

```
Input: m = 7, n = 3
Output: 28
```

Example 4

```
Input: m = 3, n = 3
Output: 6
```

Constraints

- `1 <= m, n <= 100`.
- It's guaranteed that the answer will be less than or equal to `2*10^9`.

11.2.2 Solution 1: Recursive

At each point, the robot has two ways of moving: right or down. Let `P(m,n)` is the wanted result. Then you have a recursive relationship:

```
P(m,n) = P(m-1, n) + P(m, n-1)
```

If the grid has only one row or only one column, then there is only one possible path.

```
P(1, n) = P(m, 1) = 1.
```

We have a recursive implementation.

Code

```
#include <iostream>
#include <vector>
using namespace std;
int uniquePaths(int m, int n) {
    if (m == 1 || n == 1) {
        return 1;
    }
    return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
}
int main() {
    std::cout << uniquePaths(3,7) << std::endl;
    std::cout << uniquePaths(7,3) << std::endl;
    std::cout << uniquePaths(3,2) << std::endl;
    std::cout << uniquePaths(3,3) << std::endl;
}
```

```
Output:
28
28
3
6
```

This is a recursive solution that breaks down the problem into two subproblems:

- `uniquePaths(m-1, n)`
- `uniquePaths(m, n-1)`

Each recursive call reduces either the `m` or `n` value by 1.

The base case is when `m == 1` or `n == 1`, where there is only 1 unique path.

Complexity

To calculate the complexity, let's look at the recursion tree:

- `uniquePaths(m,n)` calls:
  - `uniquePaths(m-1,n)`
  - `uniquePaths(m, n-1)`
- Each of those calls two more calls and so on.

The height of the tree will be `max(m,n)`. At each level, there are 2 branches.

So the total number of nodes in the tree will be `2^max(m,n)`.

Since each node represents a function call, the runtime complexity is `O(2^max(m, n))`.

The space complexity is `O(max(m,n))` due to the call stack.

In summary, the complexities are:

- Runtime: `O(2^max(m,n)`, where `m x n` is the size of the grid.
- Extra space: `O(max(m,n)`.

11.2.3 Solution 2: Dynamic programming

The recursive implementation repeats a lot of computations.

For example, `uniquePaths(2,2)` was recomputed in both `uniquePaths(2,3)` and `uniquePaths(3,2)` when you compute `uniquePaths(3,3)`.

One way of storing what has been computed is by using dynamic programming.

Code

```
#include <iostream>
#include <vector>
using namespace std;
int uniquePaths(int m, int n) {
```

(continues on next page)

(continued from previous page)

```
    // store what have been calculated in dp
    vector<vector<int> > dp(m, vector<int>(n,1));
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }
    return dp[m - 1][n - 1];
}
int main() {
    std::cout << uniquePaths(3,7) << std::endl;
    std::cout << uniquePaths(7,3) << std::endl;
    std::cout << uniquePaths(3,2) << std::endl;
    std::cout << uniquePaths(3,3) << std::endl;
}
```

```
Output:
28
28
3
6
```

This solution uses dynamic programming to efficiently compute the number of unique paths in an `m x n` grid. It creates a 2D vector `dp` of size `m x n` and initializes all its values to 1 because there's exactly one way to reach any cell in the first row or first column.

Then, it iterates through the grid, starting from the second row and second column (i.e., indices (1, 1)), and for each cell, it calculates the number of unique paths to reach that cell. This is done by summing the number of paths from the cell above it and the cell to the left of it, as these are the only two possible directions to reach the current cell.

Finally, the value at `dp[m-1][n-1]` contains the total number of unique paths to reach the bottom-right corner of the grid, which is returned as the result.

Complexity

- Runtime: `O(m*n)`, where `m x n` is the size of the grid.
- Extra space: `O(m*n)`.

11.2.4 Solution 3: Reduced dynamic programming

You can rephrase the relationship inside the loop like this:

"new value" = "old value" + "previous value";

Then you do not have to store all values of all rows.

Code

```
#include <iostream>
#include <vector>
using namespace std;
int uniquePaths(int m, int n) {
    // store the number of unique paths for each column in each row
    vector<int> dp(n, 1);
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[j] += dp[j - 1];
        }
    }
    return dp[n - 1];
}
int main() {
    std::cout << uniquePaths(3,7) << std::endl;
    std::cout << uniquePaths(7,3) << std::endl;
    std::cout << uniquePaths(3,2) << std::endl;
    std::cout << uniquePaths(3,3) << std::endl;
}
```

```
Output:
28
28
3
6
```

Complexity

- Runtime `O(m*n)`, where `m x n` is the size of the grid.
- Memory `O(n)`.

11.2.5 Conclusion

Solution 3 uses only a 1D vector `dp` of size `n` to store the number of unique paths for each column.

First, it initializes all elements of `dp` to 1, as there's exactly one way to reach any cell in the first row or first column.

Then, it iterates through the grid, starting from the second row and second column (i.e., indices `(1, 1)`). For each cell, it updates the value in `dp` by adding the value from the cell directly above it and the value from the cell to the left of it. This step efficiently accumulates the number of unique paths to reach the current cell.

Finally, the value at `dp[n-1]` contains the total number of unique paths to reach the bottom-right corner of the grid, which is returned as the result.

A bit of wonder

I am wondering if there is some mathematics behind this problem. Please share your finding if you find a formula for the solution to this problem.

11.2.6 Exercise

- Minimum Path Sum

11.3 Largest Divisible Subset

11.3.1 Problem statement

[1]You have a collection of positive integers called `nums`, where each integer is distinct. Your task is to find the largest subset `answer` from this collection, such that for every pair of elements `(answer[i], answer[j])` within this subset:

- Either `answer[i]` is a multiple of `answer[j]` (i.e., `answer[i] % answer[j] == 0`), or
- `answer[j]` is a multiple of `answer[i]` (i.e., `answer[j] % answer[i] == 0`).

If there are multiple possible solutions, you can return any of them.

Example 1

```
Input: nums = [1,2,3]
Output: [1,2]
Explanation: [1,3] is also accepted.
```

Example 2

```
Input: nums = [1,2,4,8]
Output: [1,2,4,8]
```

[1] https://leetcode.com/problems/largest-divisible-subset/

Constraints

- `1 <= nums.length <= 1000`.
- `1 <= nums[i] <= 2 * 10^9`.
- All the integers in `nums` are **unique**.

11.3.2 Solution 1: Bruteforce with Dynamic programming

Note that the condition `a % b == 0` is called `a` *is divisible by* `b`. In mathematics, it can also be called `b` *divides* `a` and be written as `b | a`.

The symmetry of the divisibility criteria means it does not count the ordering of the `answer`. You could sort the vector `nums` before trying to find the longest subset `answer = [answer[0], answer[1], ..., answer[m]]` where `answer[i] | answer[j]` with all `0 <= i <= j <= m`.

Now assuming the `nums` were sorted. For each `i`, you need to find the largest subset `maxSubset[i]` starting from `nums[i]`. And the final answer is the largest one among them.

Example 3

```
Input: nums = [2, 4, 3, 9, 8].
Sorted nums = [2, 3, 4, 8, 9].
maxSubset[0] = [2, 4, 8].
maxSubset[1] = [3, 9].
maxSubset[2] = [4, 8].
maxSubset[3] = [8].
maxSubset[4] = [9].
Output: [2, 4, 8].
```

Note that for a sorted `nums`, if `nums[i] | nums[j]` for some `i < j`, then `maxSubset[j]` is a subset of `maxSubset[i]`.

For example, `maxSubset[2]` is a subset of `maxSubset[0]` in Example 3 because `nums[0] = 2 | 4 = nums[2]`.

That might lead to some unnecessary recomputing. To avoid it, you could use *dynamic programming* to store the `maxSubset[j]` you have already computed.

Code

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
using namespace std;

//! @return the max divisible subset starting from nums[i]
//!         and store it to _map[i]
//! @param nums a sorted array of unique positive integers
vector<int> largestDivisibleSubsetOf(vector<int>& nums,
            int i, unordered_map<int, vector<int> >& _map) {
    if (_map.find(i) != _map.end()) {
        // already computed!
        return _map[i];
    }
    vector<int> maxSubset{nums[i]}; // start with nums[i]
    if (i == nums.size() - 1) {
        // largest value in nums
        _map.insert({i, maxSubset});
        return maxSubset;
    }
    for (int j = i + 1; j < nums.size(); j++) {
        if (nums[j] % nums[i] == 0) {
            // compute the max divisble subset starting from nums[j]
            auto subset = largestDivisibleSubsetOf(nums, j, _map);

            // add nums[i] to subset as it might become maxSubset
            subset.push_back(nums[i]);
            if (maxSubset.size() < subset.size()) {
                // update maxSubset
                maxSubset = subset;
            }
```

(continues on next page)

(continued from previous page)

```
        }
    }
    // store what have been calculated in _map
    _map.insert({i, maxSubset});
    return maxSubset;
}
vector<int> largestDivisibleSubset(vector<int>& nums) {
    if (nums.size() <= 1) {
        return nums;
    }
    unordered_map<int, vector<int> > _map;
    sort(nums.begin(), nums.end());
    vector<int> answer;
    for (int i = 0; i < nums.size(); i++) {
        auto maxSubset = largestDivisibleSubsetOf(nums, i, _map);
        if (answer.size() < maxSubset.size()) {
            // update the maximal subset
            answer = maxSubset;
        }
    }
    return answer;
}
void printSolution(const vector<int>& result) {
    cout << "[";
    for (auto& v : result) {
        cout << v << ",";
    }
    cout << "]" << endl;
}
int main() {
    vector<int> nums{2,1,3};
    auto answer = largestDivisibleSubset(nums);
    printSolution(answer);
    nums = {1,2,4,8};
    answer = largestDivisibleSubset(nums);
    printSolution(answer);
```

(continues on next page)

(continued from previous page)

```
}
```

```
Output:
[2,1,]
[8,4,2,1,]
```

This solution uses dynamic programming with memoization to find the largest divisible subset of a given set of numbers.

The `largestDivisibleSubsetOf` function recursively computes the largest divisible subset starting from a particular index `i` in the sorted array `nums`. It memoizes the computed subsets in an unordered map `_map` to avoid redundant computations. By iteratively calling `largestDivisibleSubsetOf` for each index `i` in the sorted array and updating the `answer` with the largest subset found so far, the `largestDivisibleSubset` function computes the largest divisible subset of the input array `nums`.

This approach optimizes the computation by avoiding repeated calculations and leveraging dynamic programming techniques to efficiently explore the solution space.

Complexity

- Runtime: `O(n^2)`, where `n` is the number of elements in the `nums` vector.
- Extra space: `O(n^2)`.

11.3.3 Solution 2: Store only the representative of the `maxSubset`

In the brute-force solution above, you used a big `map` to log all `maxSubset[i]` though you need only the largest one at the end.

One way to save memory (and eventually improve performance) is just storing the representative of the chain relationship between the values `nums[i]` of the `maxSubset` through their indices mapping.

That means if `maxSubset[i] = [nums[i0] | nums[i1] | ... | nums[iN1]] | nums[iN]]`, you could log `pre[iN] = iN1`, `...`, `prev[i1] = i0`.

Then all you need to find is only the last index `iN` of the largest `maxSubset`.

Example 3

```
Input: nums = [2, 4, 3, 9, 8].
sorted nums = [2, 3, 4, 8, 9].
pre[0] = -1 (there is no nums[i] | nums[0]).
pre[1] = -1 (there is no nums[i] | nums[1]).
pre[2] = 0 (nums[0] is the only divisor of nums[2]).
pre[3] = 2 (for the largest subset though nums[0] and nums[2] are both
↪divisors of nums[3]).
pre[4] = 1 (nums[1] is the only divisor of nums[4]).
iN = 3 ([2 | 4 | 8] is the largest maxSubset).
Output: [8, 4, 2].
```

Code

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
vector<int> largestDivisibleSubset(vector<int>& nums) {
    if (nums.size() <= 1) {
        return nums;
    }
    sort(nums.begin(), nums.end());
    // the size of the resulting subset
    int maxSize = 0;

    // nums[maxindex] is the largest value of the resulting subset
    int maxindex = 0;

    // subsetSize[i] stores the size of the largest subset
    // having the biggest number nums[i]
    vector<int> subsetSize(nums.size(), 1);
```

(continues on next page)

(continued from previous page)

```
    // pre[i] stores the previous index of i in their largest subset
    vector<int> pre(nums.size(), -1);
    for (int i = 0; i < nums.size(); i++) {
        // find the previous nums[j] that make subsetSize[i] largest
        for (int j = i - 1; j >= 0; j--) {

            if (nums[i] % nums[j] == 0 &&
                subsetSize[j] + 1 > subsetSize[i])
            {
                subsetSize[i] = subsetSize[j] + 1;
                pre[i] = j;
            }
        }
        // update the largest subset
        if (maxSize < subsetSize[i]) {
            maxSize = subsetSize[i];
            maxindex = i;
        }
    }
    vector<int> result;
    while (maxindex != -1) {
        result.push_back(nums[maxindex]);
        maxindex = pre[maxindex];
    }
    return result;
}
void printSolution(const vector<int>& result) {
    cout << "[";
    for (auto& v : result) {
        cout << v << ",";
    }
    cout << "]" << endl;
}
int main() {
    vector<int> nums{2,1,3};
```

(continues on next page)

(continued from previous page)

```
    auto result = largestDivisibleSubset(nums);
    printSolution(result);
    nums = {1,2,4,8};
    result = largestDivisibleSubset(nums);
    printSolution(result);
}
```

```
Output:
[2,1,]
[8,4,2,1,]
```

This solution finds the largest divisible subset of a given set of numbers by dynamically updating the size of the subsets and maintaining the previous index of each element in their largest subset.

It iterates through the sorted array of numbers, updating the size of the largest subset that ends with each element by considering the previous elements that are factors of the current element. By keeping track of the maximum subset size and the index of the largest element in the subset, it constructs the largest divisible subset.

This approach optimizes the computation by avoiding redundant calculations and leveraging dynamic programming techniques to efficiently explore the solution space.

Complexity

- Runtime: `O(n^2)`, where `n` is the number of elements in the `nums` vector. The nested loop searches for previous elements with divisibility relationships, which may lead to quadratic time complexity in the worst case. However, it maintains information about subset sizes and elements, reducing redundant calculations and improving performance.
- Extra space: `O(n)`.

11.3.4 Key takeaway

In this interesting problem, we use index mapping to simplify everything. That improves the performance in both runtime and memory.

11.4 Triangle

11.4.1 Problem statement

[1]You're provided with a `triangle` array. Your goal is to find the smallest possible sum of a path from the top of the triangle to the bottom.

At each step, you have the option to move to an adjacent number in the row below. Specifically, if you're at index `i` in the current row, you can move to either index `i` or index `i + 1` in the next row.

Example 1

```
Input: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
Output: 11
Explanation: The triangle looks like:
   2
  3 4
 6 5 7
4 1 8 3
The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11␣
↪(underlined above).
```

[1] https://leetcode.com/problems/triangle/

Example 2

```
Input: triangle = [[-10]]
Output: -10
```

Constraints

- `1 <= triangle.length <= 200`.
- `triangle[0].length == 1`.
- `triangle[i].length == triangle[i - 1].length + 1`.
- `-10^4 <= triangle[i][j] <= 10^4`.

Follow up

- Could you do this using only `O(n)` extra space, where `n` is the total number of rows in the triangle?

11.4.2 Solution 1: Store all minimum paths

You can store all minimum paths at every positions `(i,j)` so you can compute the next ones with this relationship.

```
minPath[i][j] = triangle[i][j] + min(minPath[i - 1][j - 1], minPath[i -␣
↪1][j]);
```

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int minimumTotal(const vector<vector<int>>& triangle) {
```

(continues on next page)

(continued from previous page)

```
    const int n = triangle.size(); // triangle's height
    vector<vector<int>> minPath(n);
    minPath[0] = triangle[0];
    for (int i = 1; i < n; i++) {
        const int N = triangle[i].size();
        minPath[i].resize(N);
        // left most number
        minPath[i][0] = triangle[i][0] + minPath[i-1][0];
        for (int j = 1; j < N - 1; j++) {
            minPath[i][j] = triangle[i][j] + min(minPath[i-1][j-1],␣
↪minPath[i-1][j]);
        }
        // right most number
        minPath[i][N-1] = triangle[i][N-1] + minPath[i-1][N-2];

    }
    // pick the min path among the ones (begin -> end)
    // go to the bottom (n-1)
    return *min_element(minPath[n-1].begin(), minPath[n-1].end());
}
int main() {
    vector<vector<int>> triangle{{2},{3,4},{6,5,7},{4,1,8,3}};
    cout << minimumTotal(triangle) << endl;
    triangle = {{-10}};
    cout << minimumTotal(triangle) << endl;
}
```

```
Output:
11
-10
```

This solution finds the minimum path sum from the top to the bottom of a triangle, represented as a vector of vectors. It uses dynamic programming to calculate the minimum path sum.

The algorithm initializes a `minPath` vector of vectors to store the minimum path sum for each element in the `triangle`. It starts by setting the first row of `minPath` to be

the same as the first row of the `triangle`.

Then, it iterates through the rows of the `triangle` starting from the second row. For each element in the current row, it calculates the minimum path sum by considering the two possible paths from the previous row that lead to that element. It takes the minimum of the two paths and adds the value of the current element. This way, it accumulates the minimum path sum efficiently.

The algorithm continues this process until it reaches the last row of the `triangle`. Finally, it returns the minimum element from the last row of `minPath`, which represents the minimum path sum from top to bottom.

Complexity

- Runtime: `O(n^2)`, where `n` is the number of rows in the `triangle`.
- Extra space: `O(n^2)`.

11.4.3 Solution 2: Store only the minimum paths of each row

You do not need to store all paths for all rows. The computation of the next row only depends on its previous one.

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int minimumTotal(const vector<vector<int>>& triangle) {
    const int n = triangle.size();
    // store only min path for each row
    vector<int> minPath(n);
    minPath[0] = triangle[0][0];
    for (int i = 1; i < n; i++) {
        // right most number
        minPath[i] = triangle[i][i] + minPath[i - 1];
```

(continues on next page)

(continued from previous page)

```
        for (int j = i - 1; j > 0; j--) {
            minPath[j] = triangle[i][j] + min(minPath[j - 1],␣
↪minPath[j]);
        }
        // left most number
        minPath[0] = triangle[i][0] + minPath[0];
    }
    return *min_element(minPath.begin(), minPath.end());
}
int main() {
    vector<vector<int>> triangle{{2},{3,4},{6,5,7},{4,1,8,3}};
    cout << minimumTotal(triangle) << endl;
    triangle = {{-10}};
    cout << minimumTotal(triangle) << endl;
}
```

```
Output:
11
-10
```

Complexity

- Runtime: `O(n^2)`, where `n` is the number of rows in the triangle.
- Extra space: `O(n)`.

11.5 Unique Paths II

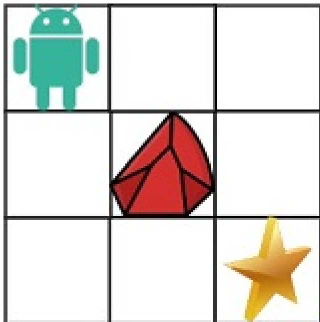
11.5.1 Problem statement

[1]You're given an `m x n` grid represented as an integer array called `grid`. In this grid, there is a robot initially located at the top-left corner (i.e., `grid[0][0]`). The robot's goal is to move to the bottom-right corner (i.e., `grid[m-1][n-1]`). The robot is allowed to move only downwards or to the right at any given point.

Your task is to determine the number of unique paths the robot can take to reach the bottom-right corner while avoiding obstacles.

It's important to note that the test cases are designed in such a way that the answer will always be less than or equal to `2 * 10^9`.

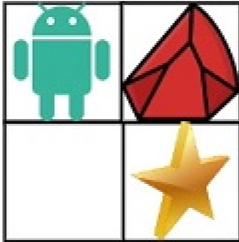
Example 1



```
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
```

[1] https://leetcode.com/problems/unique-paths-ii/ The grid contains obstacles and empty spaces, which are marked as `1` or `0` respectively. The robot cannot pass through squares marked as obstacles.

Example 2



```
Input: obstacleGrid = [[0,1],[0,0]]
Output: 1
```

Constraints

- `m == obstacleGrid.length`.
- `n == obstacleGrid[i].length`.
- `1 <= m, n <= 100`.
- `obstacleGrid[i][j]` is `0` or `1`.

11.5.2 Solution: Dynamic programming in place

Let us find the relationship between the positions.

If there is no obstacle at the position `(row = i, col = j)`, the number of paths `np[i][j]` that the robot can take to reach this position is:

```
np[i][j] = np[i - 1][j] + np[i][j - 1]
```

- As long as there is no obstacle in the first row, `np[0][j] = 1`. Otherwise, `np[0][k] = 0` for all `k >= j0`, where `(0, j0)` is the position of the first obstacle in the first row.
- Similarly, as long as there is no obstacle in the first column, `np[i][0] = 1`. Otherwise, `np[k][0] = 0` for all `k >= i0`, where `(i0, 0)` is the position of the first obstacle in the first column.

Code

```
#include <vector>
#include <iostream>
using namespace std;
int uniquePathsWithObstacles(const vector<vector<int>>& obstacleGrid) {
    const int row = obstacleGrid.size();
    const int col = obstacleGrid[0].size();
    vector<vector<int>> np(row, vector<int>(col, 0));
    for (int i = 0; i < row && obstacleGrid[i][0] == 0; i++) {
        // can move as long as there is no obstacle
        np[i][0] = 1;
    }
    for (int j = 0; j < col && obstacleGrid[0][j] == 0; j++) {
        // can move as long as there is no obstacle
        np[0][j] = 1;
    }
    for (int i = 1; i < row; i++) {
        for (int j = 1; j < col; j++) {
            if (obstacleGrid[i][j] == 0) {
                // can move since there is obstacle
                np[i][j] = np[i - 1][j] + np[i][j - 1];
            }
        }
    }
    return np[row - 1][col - 1];
}
int main() {
    vector<vector<int>> obstacleGrid = {{0,0,0},{0,1,0},{0,0,0}};
    cout << uniquePathsWithObstacles(obstacleGrid) << endl;
    obstacleGrid = {{0,1},{0,0}};
    cout << uniquePathsWithObstacles(obstacleGrid) << endl;
}
```

```
Output:
2
1
```

Complexity

- Runtime: `O(m*n)`, where `m` and `n` are the dimensions of the grid.
- Extra space: `O(m*n)`.

11.5.3 Conclusion

This solution computes the number of unique paths in an `m x n` grid with obstacles using dynamic programming. It initializes a 2D vector `np` of the same size as `obstacleGrid` to store the number of unique paths for each cell.

First, it initializes the top row and left column of `np`. If there are no obstacles in the top row or left column of `obstacleGrid`, it sets the corresponding cells in `np` to 1 because there's only one way to reach any cell in the top row or left column.

Then, it iterates through the grid starting from the second row and second column (i.e., indices (1, 1)). For each cell, if there's no obstacle (`obstacleGrid[i][j] == 0`), it updates the value in `np` by summing up the values from the cell directly above it and the cell to the left of it. This step efficiently accumulates the number of unique paths while avoiding obstacles.

Finally, the value at `np[row-1][col-1]` contains the total number of unique paths to reach the bottom-right corner of the grid, which is returned as the result.

11.5.4 Exercise

- Minimum Path Cost in a Grid

CHAPTER

TWELVE

COUNTING

In this chapter, we will explore the benefits of counting elements and how it can enhance the efficiency of different algorithms and operations. By tallying occurrences, you can gain valuable insights that simplify computations and give you a better understanding of your data.

Counting elements is like organizing a messy room. When you categorize items, it becomes easier to access and make decisions. In algorithms, counting allows you to optimize processes by identifying the most frequent elements or solving complex problems more efficiently.

What this chapter covers:

1. **Introduction to Counting:** Lay the foundation by understanding the significance of counting elements, its role in performance enhancement, and the various scenarios where counting is crucial.
2. **Frequency Counting:** Explore the technique of tallying element occurrences, enabling you to identify the most frequent items within a dataset quickly.
3. **Counting Sort:** Delve into the world of counting sort, a specialized sorting algorithm that capitalizes on the power of element counting to achieve exceptional performance.
4. **Problem-Solving with Counts:** Develop approaches to solve problems that benefit from element counting, from optimizing search operations to identifying anomalies.

12.1 Single Number

12.1.1 Problem statement

[1]You're provided with a non-empty array of integers called `nums`. In this array, every element occurs twice except for one element that appears only once. Your task is to identify and find that unique element.

To solve this problem, your solution needs to have a linear runtime complexity and utilize only a constant amount of extra space.

Example 1

```
Input: nums = [2,2,1]
Output: 1
```

Example 2

```
Input: nums = [4,1,2,1,2]
Output: 4
```

Example 3

```
Input: nums = [1]
Output: 1
```

[1] https://leetcode.com/problems/single-number/description/

Constraints

- `1 <= nums.length <= 3 * 10^4`.
- `-3 * 10^4 <= nums[i] <= 3 * 10^4`.
- Each element in the array appears twice except for one element which appears only once.

12.1.2 Solution 1: Counting the appearances

Count how many times each element appears in the array. Then return the one appearing only once.

Code

```
#include <vector>
#include <iostream>
#include <unordered_map>
using namespace std;
int singleNumber(const vector<int>& nums) {
    unordered_map<int, int> count;
    for (auto& n : nums) {
        count[n]++;
    }
    int single;
    for (auto& pair : count) {
        if (pair.second == 1) {
            single = pair.first;
            break;
        }
    }
    return single;
}
int main() {
    vector<int> nums{2,2,1};
    cout << singleNumber(nums) << endl;
```

(continues on next page)

(continued from previous page)

```
    nums = {4,1,2,1,2};
    cout << singleNumber(nums) << endl;
    nums = {1};
    cout << singleNumber(nums) << endl;
}
```

```
Output:
1
4
1
```

This solution effectively finds the single number by counting the occurrences of each element in the array and selecting the one with a count of 1.

Complexity

- Runtime: `O(N)`.
- Extra space: `O(N)`.

12.1.3 Solution 2: Bitwise exclusive OR

You can also use the bitwise XOR operator to cancel out the duplicated elements in the array. The remain element is the single one.

```
a XOR a = 0.
a XOR 0 = a.
```

Code

```
#include <vector>
#include <iostream>
using namespace std;
int singleNumber(const vector<int>& nums) {
```

(continues on next page)

(continued from previous page)

```
    int single = 0;
    for (auto& n : nums) {
        single ^= n;
    }
    return single;
}
int main() {
    vector<int> nums{2,2,1};
    cout << singleNumber(nums) << endl;
    nums = {4,1,2,1,2};
    cout << singleNumber(nums) << endl;
    nums = {1};
    cout << singleNumber(nums) << endl;
}
```

```
Output:
1
4
1
```

Complexity

- Runtime: `O(N)`.
- Extra space: `O(1)`.

12.1.4 Conclusion

Leveraging bitwise XOR (`^`) operations offers an efficient solution to find the single number in an array. Solution 2 utilizes the property of XOR where XORing a number with itself results in 0.

By XORing all the numbers in the array, Solution 2 effectively cancels out pairs of identical numbers, leaving only the single number behind. This approach achieves a linear time complexity without the need for additional data structures, providing a concise and efficient solution.

12.1.5 Exercise

- Missing Number

12.2 First Unique Character in a String

12.2.1 Problem statement

[1]You have a string called s. Your objective is to locate the index of the first character in the string that does not repeat anywhere else in the string. If such a character doesn't exist, return -1.

Example 1

```
Input: s = "leetcode"
Output: 0
```

Example 2

```
Input: s = "loveleetcode"
Output: 2
```

Example 3

```
Input: s = "aabb"
Output: -1
```

[1] https://leetcode.com/problems/first-unique-character-in-a-string/

Constraints

- `1 <= s.length <= 10^5`.
- s consists of only lowercase English letters.

12.2.2 Solution 1: Using a map to store the appearances

Code

```
#include <iostream>
#include <unordered_map>
using namespace std;
int firstUniqChar(const string& s) {
    unordered_map<char, int> count;
    for (auto& c : s) {
        count[c]++;
    }
    for (int i = 0; i < s.length(); i++) {
        if (count[s[i]] == 1) {
            return i;
        }
    }
    return -1;
}
int main() {
    cout << firstUniqChar("leetcode") << endl;
    cout << firstUniqChar("loveleetcode") << endl;
    cout << firstUniqChar("aabb") << endl;
}
```

```
Output:
0
2
-1
```

This solution finds the index of the first non-repeating character in a string by using an unordered map to count the occurrences of each character.

By iterating through the string and populating the unordered map with the count of each character, it constructs the character count. Then, it iterates through the string again and returns the index of the first character with a count of 1, indicating that it is non-repeating.

This approach optimizes the computation by efficiently tracking the count of each character and identifying the first non-repeating character without requiring additional space proportional to the length of the string.

Complexity

- Runtime: `O(n)`, where `n` is the length of the string `s`.
- Extra space: `O(1)` as the problem considers only 26 lowercase English letters.

12.2.3 Solution 2: Using an array to store the appearances

From the constraints "s consists of only lowercase English letters", you can use an array of 26 elements to store the counts.

Code

```
#include <iostream>
#include <vector>
using namespace std;
int firstUniqChar(const string& s) {
    // map 'a'->0, 'b'->1, .., 'z'->25
    // initializes an array of 26 elements, all set to zero
    std::array<int, 26> count{};
    for (auto& c : s) {
        count[c - 'a']++;
    }
    for (int i = 0; i < s.length(); i++) {
        if (count[s[i] - 'a'] == 1) {
            return i;
        }
```

(continues on next page)

(continued from previous page)

```
    }
    return -1;
}
int main() {
    cout << firstUniqChar("leetcode") << endl;
    cout << firstUniqChar("loveleetcode") << endl;
    cout << firstUniqChar("aabb") << endl;
}
```

```
Output:
0
2
-1
```

Complexity

- Runtime: `O(n)`, where `n = s.length`.
- Extra space: `O(1)` as the array is fixed regardless of how big `n` is.

12.2.4 Conclusion

Utilizing hash maps or arrays to count the frequency of characters in a string provides an efficient way to identify the first unique character. Both solutions use this approach to iterate through the string and count the occurrences of each character.

By storing the counts in a data structure indexed by the character value, the solutions achieve a linear time complexity proportional to the length of the string. Solution 2 further optimizes memory usage by employing an array with a fixed size corresponding to the lowercase English alphabet, avoiding the overhead associated with hash maps.

12.2.5 Exercise

- First Letter to Appear Twice

12.3 Max Number of K-Sum Pairs

12.3.1 Problem statement

[1]You're provided with an array of integers called `nums` and an integer `k`. Each operation involves selecting two numbers from the array whose sum is equal to `k`, and then removing them from the array. Your goal is to determine the maximum count of such operations you can perform on the array.

Example 1

```
Input: nums = [1,2,3,4], k = 5
Output: 2
Explanation: Starting with nums = [1,2,3,4]:
- Remove numbers 1 and 4, then nums = [2,3]
- Remove numbers 2 and 3, then nums = []
There are no more pairs that sum up to 5, hence a total of 2 operations.
```

Example 2

```
Input: nums = [3,1,3,4,3], k = 6
Output: 1
Explanation: Starting with nums = [3,1,3,4,3]:
- Remove the first two 3's, then nums = [1,4,3]
There are no more pairs that sum up to 6, hence a total of 1 operation.
```

[1] https://leetcode.com/problems/max-number-of-k-sum-pairs/

Constraints

- `1 <= nums.length <= 10^5`.
- `1 <= nums[i] <= 10^9`.
- `1 <= k <= 10^9`.

12.3.2 Solution: Count the appearances

You can use a map to count the appearances of the elements of `nums`.

Example 2

For `nums = [3,1,3,4,3]` and `k = 6`:

- Initialize `count = 0`.
- For `i = 0`: `m[3] = 1`; `k - 3 = 3` but `m[3]` is only `1`, not enough to have two numbers.
- For `i = 1`: `m[1] = 1`; `k - 1 = 5` and `m[5] = 0`.
- For `i = 2`: `m[3] = 2`; `k - 3 = 3` and `m[3] = 2` just enough to have two numbers to perform the sum. `count = 1`. Erase those two values `3`'s from the map: `m[3] = 0`.
- For `i = 3`: `m[4] = 1`; `k - 4 = 2` and `m[2] = 0`.
- For `i = 4`: `m[3] = 1`; `k - 3 = 3` but `m[3]` is only `1`, not enough to have two numbers.
- Final `count = 1`.

Code

```
#include <vector>
#include <iostream>
#include <unordered_map>
using namespace std;
```

(continues on next page)

(continued from previous page)

```
int maxOperations(const vector<int>& nums, int k) {
    unordered_map<int,int> m;
    int count = 0;
    for (auto& a : nums) {
        m[a]++; // count a's occurences
        if (m[k - a] > 0) {
            // k-a appears in nums
            if (a != k - a || m[a] >= 2) {
                // if a == k - a, a is required to appear at least twice
                count++;
                m[a]--;
                m[k - a]--;
            }
        }
    }
    return count;
}
int main() {
    vector<int> nums{1,2,3,4};
    cout << maxOperations(nums, 5) << endl;
    nums = {3,1,3,4,3};
    cout << maxOperations(nums, 6) << endl;
}
```

```
Output:
2
1
```

Complexity

- Runtime: `O(n)`, where `n` is the number of elements in the `nums` array.
- Extra space: `O(n)`.

12.3.3 Conclusion

This solution utilizes an unordered map to store the frequency of each element encountered while iterating through `nums`.

By examining each element `a` in `nums`, it checks if `k - a` exists in the map and if its frequency is greater than 0. If so, it increments the count of pairs and decrements the frequency of both `a` and `k - a`, ensuring that each pair is counted only once.

This approach optimizes the computation by efficiently tracking the frequencies of elements and identifying valid pairs whose sum equals the target value without requiring additional space proportional to the size of the array.

12.3.4 Exercise

- Two Sum

CHAPTER THIRTEEN

PREFIX SUMS

This chapter will introduce you to a technique called **prefix sums**. This technique can make calculations much faster and more efficient. The chapter will explain how cumulative aggregation works and can help optimize your operations.

Prefix sums are like building blocks that can create many different algorithms. They make it easier to handle cumulative values and allow you to solve complex problems much more efficiently than before.

What this chapter covers:

1. **Introduction to Prefix Sums:** Establish the groundwork by understanding the essence of prefix sums, their role in performance enhancement, and the scenarios where they shine.
2. **Prefix Sum Array Construction:** Dive into the mechanics of constructing a prefix sum array, unlocking the potential to access cumulative values efficiently.
3. **Range Sum Queries:** Explore how prefix sums revolutionize calculating sums within a given range, enabling quick and consistent results.
4. **Subarray Sum Queries:** Delve into the technique's application in efficiently determining the sum of elements within any subarray of an array.
5. **Prefix Sum Variants:** Discover the versatility of prefix sums in solving problems related to averages, running maximum/minimum values, and more.
6. **Problem-Solving with Prefix Sums:** Develop strategies for solving diverse problems by incorporating prefix sums, from optimizing sequence operations to speeding up specific algorithms.

13.1 Running Sum of 1d Array

13.1.1 Problem statement

[1]Given an array called `nums`, calculate the running sum of its elements and return the resulting array. The running sum at index `i` is the sum of elements from index `0` to `i` in the `nums` array.

Example 1

```
Input: nums = [1,2,3,4]
Output: [1,3,6,10]
Explanation: Running sum is obtained as follows: [1, 1+2, 1+2+3,␣
↪1+2+3+4].
```

Example 2

```
Input: nums = [1,1,1,1,1]
Output: [1,2,3,4,5]
Explanation: Running sum is obtained as follows: [1, 1+1, 1+1+1,␣
↪1+1+1+1, 1+1+1+1+1].
```

Example 3

```
Input: nums = [3,1,2,10,1]
Output: [3,4,6,16,17]
```

[1] https://leetcode.com/problems/running-sum-of-1d-array/

Constraints

- `1 <= nums.length <= 1000`.
- `-10^6 <= nums[i] <= 10^6`.

13.1.2 Solution 1: Unchange nums

Code

```cpp
#include <vector>
#include <iostream>
using namespace std;
vector<int> runningSum(const vector<int>& nums) {
    vector<int> rs;
    int s = 0;
    for (auto& n : nums) {
        s += n;
        rs.push_back(s);
    }
    return rs;
}
void printResult(const vector<int>& sums) {
    cout << "[";
    for (auto& s: sums) {
        cout << s << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> nums{1,2,3,4};
    auto rs = runningSum(nums);
    printResult(rs);
    nums = {1,1,1,1,1};
    rs = runningSum(nums);
    printResult(rs);
    nums = {3,1,2,10,1};
```

(continues on next page)

(continued from previous page)

```
    rs = runningSum(nums);
    printResult(rs);
}
```

```
Output:
[1,3,6,10,]
[1,2,3,4,5,]
[3,4,6,16,17,]
```

This solution iterates through the input array `nums`, calculates the running sum at each step, and appends the running sums to a result vector. This approach efficiently computes the running sums in a single pass through the array.

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(1)`.

13.1.3 Solution 2: Change `nums`

If `nums` is allowed to be changed, you could use it to store the result directly.

Code

```cpp
#include <vector>
#include <iostream>
using namespace std;
vector<int> runningSum(vector<int>& nums) {
    for (int i = 1; i < nums.size(); i++) {
        nums[i] += nums[i - 1];
    }
    return nums;
}
```

(continues on next page)

(continued from previous page)

```
void printResult(const vector<int>& sums) {
    cout << "[";
    for (auto& s: sums) {
        cout << s << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> nums{1,2,3,4};
    auto rs = runningSum(nums);
    printResult(rs);
    nums = {1,1,1,1,1};
    rs = runningSum(nums);
    printResult(rs);
    nums = {3,1,2,10,1};
    rs = runningSum(nums);
    printResult(rs);
}
```

```
Output:
[1,3,6,10,]
[1,2,3,4,5,]
[3,4,6,16,17,]
```

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(1)`.

13.1.4 Conclusion

Solution 2 directly modifies the input array `nums` to store the running sums by iteratively updating each element with the cumulative sum of the previous elements. This approach efficiently calculates the running sums in a single pass through the array.

13.2 Maximum Subarray

13.2.1 Problem statement

[1]You're provided with an array of integers called `nums`. Your task is to identify a subarray (a consecutive sequence of numbers) that has the highest sum. Once you find this subarray, return the sum of its elements.

Example 1

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Example 2

```
Input: nums = [1]
Output: 1
```

[1] https://leetcode.com/problems/maximum-subarray/

Example 3

```
Input: nums = [5,4,-1,7,8]
Output: 23
```

Constraints

- `1 <= nums.length <= 10^5.`
- `-10^4 <= nums[i] <= 10^4.`

13.2.2 Solution

The subarrays you want to find should not have negative prefix sums. A negative prefix sum would make the sum of the subarray smaller.

Example 1

For `nums = [-2,1,-3,4,-1,2,1,-5,4]`, `[-2]` or `[-2,1]` or `[-2,1,-3]` should not be a prefix of the subarrays you want to find. Since it makes the sum of the result smaller.

Code

```
#include <vector>
#include <iostream>
using namespace std;
int maxSubArray(const vector<int>& nums) {
    int maxSum = -10000; // just chose some negative number to start
    int currSum = 0; // sum of current subarray
    for (auto& num : nums) {
        if (currSum < 0) {
            // start a new subarray from this num
            currSum = num;
```

(continues on next page)

(continued from previous page)

```
        } else {
            currSum = currSum + num;
        }
        // update max sum so far
        maxSum = max(maxSum, currSum);
    }
    return maxSum;
}
int main() {
    vector<int> nums = {-2,1,-3,4,-1,2,1,-5,4};
    cout << maxSubArray(nums) << endl;
    nums = {1};
    cout << maxSubArray(nums) << endl;
    nums = {5,4,-1,7,8};
    cout << maxSubArray(nums) << endl;
}
```

```
Output:
6
1
23
```

Complexity

- Runtime `O(n)`, where `n = nums.length`.
- Memory `O(1)`.

13.2.3 Conclusion

This solution is the Kadane's algorithm to find the maximum sum of a contiguous subarray in the given array nums.

It iterates through the elements of the array, updating `currSum` to either the current element or the sum of the current element and the previous `currSum`, whichever is greater. By considering whether adding the current element improves the overall

sum, it effectively handles both positive and negative numbers in the array. Finally, it updates `maxSum` with the maximum value encountered during the iteration, ensuring it holds the maximum sum of any contiguous subarray within the given array.

This approach optimizes the computation by tracking the maximum sum and dynamically updating it as it iterates through the array.

13.2.4 Exercise

- Maximum Product Subarray

13.3 Product of Array Except Self

13.3.1 Problem statement

[1]Given an integer array `nums`, return an array `answer` such that `answer[i]` is equal to the product of all the elements of nums except `nums[i]`.

The product of any prefix or suffix of `nums` is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in `O(n)` time and without using the division operation.

Example 1

```
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
```

[1] https://leetcode.com/problems/product-of-array-except-self/

Example 2

```
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]
```

Constraints

- `2 <= nums.length <= 10^5.`
- `-30 <= nums[i] <= 30.`
- The product of any prefix or suffix of `nums` is guaranteed to fit in a 32-bit integer.

Follow up

- Can you solve the problem in `O(1)` extra space complexity? (The output array does not count as extra space for space complexity analysis.)

13.3.2 Solution 1: Compute the prefix and suffix products

To avoid division operation, you can compute the prefix product and the suffix one of `nums[i]`.

Code

```
#include <vector>
#include <iostream>
using namespace std;
vector<int> productExceptSelf(const vector<int>& nums) {
    const int n = nums.size();
    vector<int> prefix(n);
    prefix[0] = 1;
    // compute all prefix products nums[0]*nums[1]*..*nums[i-1]
    for (int i = 1; i < n; i++) {
```

(continues on next page)

(continued from previous page)

```
        prefix[i] = prefix[i - 1] * nums[i - 1];
    }
    vector<int> suffix(n);
    suffix[n - 1] = 1;
    // compute all suffix products nums[i+1]*nums[i+2]*..*nums[n-1]
    for (int i = n - 2; i >= 0; i--) {
        suffix[i] = suffix[i + 1] * nums[i + 1];
    }
    vector<int> answer(n);
    for (int i = 0; i < n; i++) {
        answer[i] = prefix[i] * suffix[i];
    }
    return answer;
}
void print(const vector<int>& nums) {
    for (auto& v : nums) {
        cout << v << " ";
    }
    cout << endl;
}
int main() {
    vector<int> nums = {1, 2, 3, 4};
    auto answer = productExceptSelf(nums);
    print(answer);
    nums = {-1, 1, 0, -3, 3};
    answer = productExceptSelf(nums);
    print(answer);
}
```

```
Output:
24 12 8 6
0 0 9 0 0
```

This solution computes the product of all elements in an array except for the current element.

It accomplishes this by first computing two arrays: `prefix` and `suffix`. The `prefix`

array stores the product of all elements to the left of the current element, while the `suffix` array stores the product of all elements to the right of the current element. By multiplying the corresponding elements from `prefix` and `suffix` arrays, it effectively computes the product of all elements except for the current element at each index.

This approach optimizes the computation by breaking down the problem into smaller subproblems and leveraging the precomputed prefix and suffix arrays to efficiently compute the final result.

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(n)`.

13.3.3 Solution 2: Use directly vector `answer` to store the prefix product

In the solution above you can use directly vector `answer` for `prefix` and merge the last two loops into one.

Code

```
#include <vector>
#include <iostream>
using namespace std;
vector<int> productExceptSelf(const vector<int>& nums) {
    const int n = nums.size();
    vector<int> answer(n);
    answer[0] = 1;
    // compute all prefix products nums[0]*nums[1]*..*nums[i-1]
    for (int i = 1; i < n; i++) {
        answer[i] = answer[i - 1] * nums[i - 1];
    }
    int suffix = 1;
    for (int i = n - 2; i >= 0; i--) {
```

(continues on next page)

(continued from previous page)

```
        // compute suffix product and the final product the same time
        suffix *= nums[i + 1];
        answer[i] *= suffix;
    }
    return answer;
}
void print(const vector<int>& nums) {
    for (auto& v : nums) {
        cout << v << " ";
    }
    cout << endl;
}
int main() {
    vector<int> nums = {1, 2, 3, 4};
    auto answer = productExceptSelf(nums);
    print(answer);
    nums = {-1, 1, 0, -3, 3};
    answer = productExceptSelf(nums);
    print(answer);
}
```

```
Output:
24 12 8 6
0 0 9 0 0
```

This code efficiently calculates the products of all elements in the `nums` vector except for the element at each index using two passes through the array. The first pass calculates products to the left of each element, and the second pass calculates products to the right of each element.

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(1)`.

13.3.4 Conclusion

The problem of computing the product of all elements in an array except the element at the current index can be efficiently solved using different approaches. Solution 1 utilizes two separate passes through the array to compute prefix and suffix products independently. By first computing prefix products from left to right and then suffix products from right to left, this solution efficiently calculates the product of all elements except the one at the current index.

Solution 2 offers a more concise approach by combining the computation of prefix and suffix products into a single pass through the array. By iteratively updating a variable to compute suffix products while simultaneously updating the elements of the answer array, this solution achieves the desired result more efficiently with only one pass through the array.

13.3.5 Exercise

- Construct Product Matrix

13.4 Subarray Sum Equals K

13.4.1 Problem Statement

[1]You have an array of integers called `nums` and an integer `k`. Your task is to determine the count of contiguous subarrays within this array, where the sum of elements in each subarray is equal to the value of `k`.

[1] https://leetcode.com/problems/subarray-sum-equals-k/

Example 1

```
Input: nums = [1,1,1], k = 2
Output: 2
```

Example 2

```
Input: nums = [1,2,3], k = 3
Output: 2
```

Constraints

- `1 <= nums.length <= 2 * 10^4`.
- `-1000 <= nums[i] <= 1000`.
- `-10^7 <= k <= 10^7`.

13.4.2 Solution 1: Bruteforce

For each element, for all subarrays starting from it, choose the satisfied ones.

Example 3

For `nums = [1, -1, 0]` and `k = 0`, you get `3` subarrays for the result:

- There are three subarrays starting from `1`, which are `[1]`, `[1, -1]`, and `[1, -1, 0]`. Only the last two are satisfied.
- There are two subarrays starting from `-1`, which are `[-1]` and `[-1, 0]`. None is satisfied.
- Only `[0]` is the subarray starting from `0`. It is satisfied.

Code

```
#include <iostream>
#include <vector>
using namespace std;
int subarraySum(const vector<int>& nums, int k) {
    int count = 0;
    for (int i = 0; i < nums.size(); i++) {
        int sum = 0;
        for (int j = i; j < nums.size(); j++) {
            sum += nums[j];
            if (sum == k) {
                count++;
            }
        }
    }
    return count;
}
int main() {
    vector<int> nums{1,1,1};
    cout << subarraySum(nums, 2) << endl;
    nums = {1,2,3};
    cout << subarraySum(nums, 3) << endl;
    nums = {1,-1,0};
    cout << subarraySum(nums, 0) << endl;
}
```

```
Output:
2
2
3
```

This solution employs a brute-force approach by considering all possible subarrays and checking whether their sum equals the target k. The time complexity of this solution is relatively high due to the nested loops, resulting in an inefficient algorithm for larger input sizes.

Complexity

- Runtime: O(n^2), where n = nums.length.
- Extra space: O(1).

13.4.3 Solution 2: Prefix sum

In the solution above, many sums can be deducted from the previous ones.

Example 4

For nums = [1, 2, 3, 4]. Assume the sum of the subarrays [1], [1, 2], [1, 2, 3], [1, 2, 3, 4] were computed in the first loop. Then the sum of any other subarray can be deducted from those values.

- sum([2, 3]) = sum([1, 2, 3]) - sum([1]).
- sum([2, 3, 4]) = sum([1, 2, 3, 4]) - sum([1]).
- sum([3, 4]) = sum(1, 2, 3, 4) - sum(1, 2).

In general, assume you have computed the sum sum[i] for the subarray [nums[0], nums[1], ..., nums[i]] for all 0 <= i < nums.length. Then the sum of the subarray [nums[j+1], nums[j+2], ..., nums[i]] for any 0 <= j <= i can be computed as sum[i] - sum[j].

Code

```
#include <iostream>
#include <vector>
using namespace std;
int subarraySum(const vector<int>& nums, int k) {
    const int n = nums.size();
    vector<int> sum(n);
    sum[0] = nums[0];
    // compute all prefix sums nums[0] + .. + nums[i]
    for (int i = 1; i < n; i++) {
```

(continues on next page)

(continued from previous page)

```
        sum[i] = sum[i-1] + nums[i];
    }
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (sum[i] == k) {
            // nums[0] + .. + nums[i] = k
            count++;
        }
        for (int j = 0; j < i; j++) {
            if (sum[i] - sum[j] == k) {
                // nums[j+1] + nums[j+2] + .. + nums[i] = k
                count++;
            }
        }
    }
    return count;
}
int main() {
    vector<int> nums{1,1,1};
    cout << subarraySum(nums, 2) << endl;
    nums = {1,2,3};
    cout << subarraySum(nums, 3) << endl;
    nums = {1,-1,0};
    cout << subarraySum(nums, 0) << endl;
}
```

```
Output:
2
2
3
```

This solution uses the concept of prefix sum to efficiently calculate the sum of subarrays. It then iterates through the array to find subarrays with a sum equal to k, and the nested loop helps in calculating the sum of various subarray ranges. The time complexity of this solution is improved compared to the brute-force approach.

Complexity

- Runtime: `O(n^2)`, where `n = nums.length`.
- Extra space: `O(n)`.

13.4.4 Solution 3: Faster lookup

You can rewrite the condition `sum[i] - sum[j] == k` in the inner loop of the Solution 2 to `sum[i] - k == sum[j]`.

Then that loop can rephrase to *"checking if* `sum[i] - k` *was already a value of* ***some*** *computed* `sum[j]`*"*.

Now you can use an `unordered_map` to store the `sums` as indices for the fast lookup.

Code

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
int subarraySum(const vector<int>& nums, int k) {
    int count = 0;
    // count the frequency of all subarrays' sums
    unordered_map<int, int> sums;
    int sumi = 0;
    for (int i = 0; i < nums.size(); i++) {
        sumi += nums[i];
        if (sumi == k) {
            count++;
        }
        auto it = sums.find(sumi - k);
        if (it != sums.end()) {
            // it->second is the count of j so far
            // having sum[j] = sum[i] - k
            count += it->second;
```

(continues on next page)

(continued from previous page)

```
        }
        // store the count of prefix sum sumi
        sums[sumi]++;
    }
    return count;
}
int main() {
    vector<int> nums{1,1,1};
    cout << subarraySum(nums, 2) << endl;
    nums = {1,2,3};
    cout << subarraySum(nums, 3) << endl;
    nums = {1,-1,0};
    cout << subarraySum(nums, 0) << endl;
}
```

```
Output:
2
2
3
```

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(n)`.

13.4.5 Conclusion

Solution 3 uses an unordered map to efficiently track the frequency of cumulative sums. It utilizes the concept of complement sums to identify subarrays with the desired sum and adds their counts to the overall count. This approach significantly improves the time complexity compared to the brute-force solution.

13.4.6 Exercise

- Find Pivot Index

CHAPTER FOURTEEN

TWO POINTERS

This chapter will explore the **Two Pointers** technique, a strategic approach that can help solve complex problems quickly and effectively. We'll show you how to use simultaneous traversal to streamline operations, optimize algorithms, and extract solutions from complicated scenarios.

The Two Pointers technique is like exploring a cryptic map from both ends to find the treasure. It can enhance your problem-solving skills and help you tackle intricate challenges with a broader perspective.

What this chapter covers:

1. **Introduction to Two Pointers:** Lay the foundation by understanding the essence of the Two Pointers technique, its adaptability, and its role in unraveling complex problems.
2. **Two Pointers Approach:** Dive into the mechanics of the technique, exploring scenarios where two pointers traverse a sequence to locate solutions or patterns.
3. **Collision and Separation:** Discover the duality of the technique, where pointers can converge to solve particular problems or diverge to address different aspects of a challenge.
4. **Optimal Window Management:** Explore how the Two Pointers technique optimizes sliding window problems, facilitating efficient substring or subarray analysis.
5. **Intersection and Union:** Uncover the technique's versatility in solving problems that involve intersecting or uniting elements within different sequences.

6. **Problem-Solving with Two Pointers:** Develop strategies to address diverse problems through the Two Pointers technique, from array manipulation to string analysis.

14.1 Middle of the Linked List

14.1.1 Problem statement

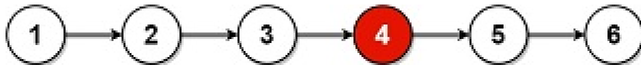
[1]Given the `head` of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return *the second middle* node.

Example 1

```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.
```

Example 2



```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4,␣
↪we return the second one.
```

[1] https://leetcode.com/problems/middle-of-the-linked-list/

Constraints

- The number of nodes in the list is in the range `[1, 100]`.
- `1 <= Node.val <= 100`.

14.1.2 Solution 1: Counting the number of nodes

Code

```
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
ListNode* middleNode(ListNode* head) {
    ListNode *node = head;
    int count = 0;
    while (node) {
        count++;
        node = node->next;
    }
    int i = 1;
    node = head;
    while (i <= count/2) {
        node = node->next;
        i++;
    }
    return node;
}
void print(const ListNode *head) {
    ListNode *node = head;
    std::cout << "[";
    while (node) {
```

(continues on next page)

(continued from previous page)

```
        std::cout << node->val << ",";
        node = node->next;
    }
    std::cout << "]\n";
}
int main() {
    ListNode five(5);
    ListNode four(4, &five);
    ListNode three(3, &four);
    ListNode two(2, &three);
    ListNode one(1, &two);
    auto result = middleNode(&one);
    print(result);

    ListNode six(6);
    five.next = &six;
    result = middleNode(&one);
    print(result);
}
```

```
Output:
[3,4,5,]
[4,5,6,]
```

This solution first counts the total number of nodes in the linked list, and then it iterates to the middle node using the `count` variable.

Complexity

- Runtime: O(N), where N is the number of nodes in the linked list.
- Extra space: O(1).

14.1.3 Solution 2: Fast and Slow pointers

Use two pointers to go through the linked list.

One goes one step at a time. The other goes two steps at a time. When the faster reaches the end, the slower reaches the middle.

Code

```
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
ListNode* middleNode(ListNode* head) {
    ListNode *slow = head;
    ListNode *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
void print(const ListNode *head) {
    ListNode *node = head;
    std::cout << "[";
    while (node) {
        std::cout << node->val << ",";
```

(continues on next page)

(continued from previous page)

```
        node = node->next;
    }
    std::cout << "]\n";
}
int main() {
    ListNode five(5);
    ListNode four(4, &five);
    ListNode three(3, &four);
    ListNode two(2, &three);
    ListNode one(1, &two);
    auto result = middleNode(&one);
    print(result);

    ListNode six(6);
    five.next = &six;
    result = middleNode(&one);
    print(result);
}
```

```
Output:
[3,4,5,]
[4,5,6,]
```

This solution uses two pointers, a slow pointer and a fast pointer, to find the middle node of a linked list. Both pointers start from the head of the list, and in each iteration, the slow pointer moves one step forward while the fast pointer moves two steps forward. This ensures that the slow pointer reaches the middle node of the list when the fast pointer reaches the end.

By advancing the pointers at different speeds, the algorithm identifies the middle node of the linked list. If the list has an odd number of nodes, the slow pointer will be positioned at the middle node. If the list has an even number of nodes, the slow pointer will be positioned at the node closer to the middle of the list.

Finally, the algorithm returns the slow pointer, which points to the middle node of the linked list.

This approach optimizes the computation by traversing the linked list only once and

using two pointers to efficiently locate the middle node.

Complexity

- Runtime: `O(N)`, where `N` is the number of nodes in the linked list.
- Extra space: `O(1)`.

14.1.4 OBS!

- The approach using slow and fast pointers looks very nice and faster. But it is not suitable to generalize this problem to any relative position (one-third, a quarter, etc.). Moreover, long expressions like `fast->next->...->next` are not recommended.
- Though the counting nodes approach does not seem optimized, it is more readable, scalable and maintainable.

14.1.5 Exercise

- Delete the Middle Node of a Linked List

14.2 Linked List Cycle

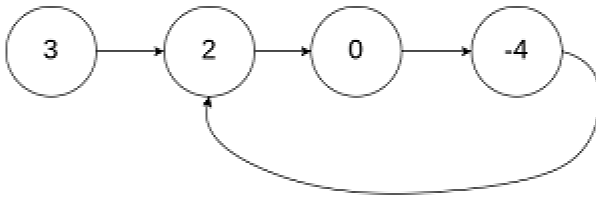
14.2.1 Problem statement

[1]Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

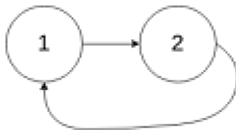
[1] https://leetcode.com/problems/linked-list-cycle/

Example 1



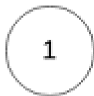

```
Input: head = [3,2,0,-4], where -4 links next to 2.
Output: true
```

Example 2




```
Input: head = [1,2], where 2 links next to 1.
Output: true
```

Example 3




```
Input: head = [1], and 1 links to NULL.
Output: false
Explanation: There is no cycle in this linked list.
```

Constraints

- The number of the nodes in the list is in the range `[0, 10^4]`.
- `-10^5 <= Node.val <= 10^5`.

Follow up

- Can you solve it using `O(1)` (i.e. constant) memory?

14.2.2 Solution 1: Storing the visited nodes

Code

```
#include <unordered_map>
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
bool hasCycle(ListNode *head) {
    std::unordered_map<ListNode*, bool> m;
    while (head) {
        if (m[head]) {
            // found this node marked in the map
            return true;
        }
        m[head] = true; // mark this node visited
        head = head->next;
    }
    return false;
}
int main() {
    {
        ListNode three(3);
```

(continues on next page)

(continued from previous page)

```
        ListNode two(2);
        three.next = &two;
        ListNode zero(0);
        two.next = &zero;
        ListNode four(4);
        zero.next = &four;
        four.next = &two;
        std::cout << hasCycle(&three) << std::endl;
    }
    {
        ListNode one(1);
        ListNode two(2);
        one.next = &two;
        two.next = &one;
        std::cout << hasCycle(&one) << std::endl;
    }
    {
        ListNode one(1);
        std::cout << hasCycle(&one) << std::endl;
    }
}
```

```
Output:
1
1
0
```

This solution uses a hash map to track visited nodes while traversing the linked list.

By iterating through the linked list and marking pointers to visited nodes in the hash map, it detects cycles in the linked list. If a node is found marked `true` in the map, it indicates the presence of a cycle, and the function returns `true`. Otherwise, if the end of the linked list is reached without finding any node marked, it confirms the absence of a cycle, and the function returns `false`.

This approach optimizes the computation by leveraging the hash map to efficiently detect cycles in the linked list without requiring additional space proportional to the length of the list.

Complexity

- Runtime: O(N), where N is the length of the linked list.
- Extra space: O(N).

14.2.3 Solution 2: Fast and Slow runners

Imagine there are two runners both start to run along the linked list from the head. One runs twice faster than the other.

If the linked list has a cycle in it, they will meet at some point. Otherwise, they never meet each other.

Example 1

The slower runs [3,2,0,-4,2,0,...] while the faster runs [3,0,2,-4,0,2,...]. They meet each other at node -4 after three steps.

Example 2

The slower runs [1,2,1,2,...] while the faster runs [1,1,1,...]. They meet each other at node 1 after two steps.

Code

```
#include <iostream>
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
bool hasCycle(ListNode *head) {
    if (head == nullptr) {
        return false;
    }
```

(continues on next page)

(continued from previous page)

```
    ListNode* fast = head;
    ListNode* slow = head;
    while (fast && fast->next) {
        fast = fast->next->next;
        slow = slow->next;
        if (fast == slow) {
            return true;
        }
    }
    return false;
}
int main() {
    {
        ListNode three(3);
        ListNode two(2);
        three.next = &two;
        ListNode zero(0);
        two.next = &zero;
        ListNode four(4);
        zero.next = &four;
        four.next = &two;
        std::cout << hasCycle(&three) << std::endl;
    }
    {
        ListNode one(1);
        ListNode two(2);
        one.next = &two;
        two.next = &one;
        std::cout << hasCycle(&one) << std::endl;
    }
    {
        ListNode one(1);
        std::cout << hasCycle(&one) << std::endl;
    }
}
```

```
Output:
1
1
0
```

Complexity

- Runtime: `O(N)`, where `N` is the number of nodes in the linked list.
- Extra space: `O(1)`.

14.2.4 Conclusion

Solution 2 uses two pointers, a fast pointer and a slow pointer, to detect cycles in a linked list.

Both pointers start from the head of the list, and the fast pointer moves two steps forward while the slow pointer moves one step forward in each iteration. By comparing the positions of the fast and slow pointers, the algorithm detects cycles in the linked list.

If the fast pointer catches up with the slow pointer at any point during traversal, it indicates the presence of a cycle, and the function returns `true`. Otherwise, if the fast pointer reaches the end of the list without intersecting with the slow pointer, it confirms the absence of a cycle, and the function returns `false`.

This approach optimizes the computation by simultaneously advancing two pointers at different speeds to efficiently detect cycles in the linked list.

14.2.5 Exercise

- Linked List Cycle II

14.3 Sort Array By Parity II

14.3.1 Problem statement

[1]Given an array of integers nums, half of the integers in nums are odd, and the other half are even.

Sort the array so that whenever nums[i] is odd, i is odd, and whenever nums[i] is even, i is even.

Return any answer array that satisfies this condition.

Example 1

```
Input: nums = [4,2,5,7]
Output: [4,5,2,7]
Explanation: [4,7,2,5], [2,5,4,7], [2,7,4,5] would also have been
↪accepted.
```

Example 2

```
Input: nums = [2,3]
Output: [2,3]
```

Constraints:

- `2 <= nums.length <= 2 * 10^4`.
- `nums.length` is even.
- Half of the integers in nums are even.
- `0 <= nums[i] <= 1000`.

[1] https://leetcode.com/problems/sort-array-by-parity-ii/

14.3.2 Solution 1: Bubble Sort

For each `0 <= i < nums.length`, if `nums[i]` has the same parity with `i`, you do nothing. Otherwise you need to find another `nums[j]` that has the same parity with `i` to swap with `nums[i]`.

Example 1

For `nums = [4,2,5,7]`:

- `nums[0] = 4` is even like `i = 0`.
- `nums[1] = 2` is even, unlike `i = 1` is odd. Found `nums[2] = 5` is odd. Swap `nums[1] <-> nums[2]`. `nums[2]` becomes `2` while `nums[1]` becomes `5` is odd like `i = 1`.
- `nums[2] = 2` is even, like `i = 2`.
- `nums[3] = 7` is odd like `i = 3`.

Code

```
#include<vector>
#include<iostream>
using namespace std;
vector<int> sortArrayByParityII(vector<int>& nums) {
    for (int i = 0; i < nums.size(); i++) {
        if (i % 2 != nums[i] % 2) {
            // find suitable nums[j] to swap
            for (int j = i + 1; j < nums.size(); j++) {
                if (nums[j] % 2 == i % 2) {
                    swap(nums[i], nums[j]);
                    break;
                }
            }
        }
    }
    return nums;
```

(continues on next page)

(continued from previous page)

```
}
void print(vector<int>& nums) {
    for (auto num : nums) {
        cout << num << " ";
    }
    cout << endl;
}
int main() {
    vector<int> nums = {4,2,5,7};
    auto result = sortArrayByParityII(nums);
    print(result);
    nums = {1,0,7,3,8,9,2,5,4,1,2,4};
    result = sortArrayByParityII(nums);
    print(result);
    nums = {3,4};
    result = sortArrayByParityII(nums);
    print(result);
    nums = {648,831,560,986,192,424,997,829,897,843};
    result = sortArrayByParityII(nums);
    print(result);
}
```

```
Output:
4 5 2 7
0 1 8 3 2 9 4 5 2 1 4 7
4 3
648 831 560 997 192 829 986 897 424 843
```

This solution iteratively scans through the array and swap elements to ensure that the parity (even or odd) of each element matches its index modulo 2.

The algorithm iterates over each index of the array. For each index `i`, if the parity of the element at index `i` does not match `i % 2`, it implies that the element is in the wrong position. In such cases, the algorithm searches for the next element with the correct parity (i.e., even or odd) starting from index `i + 1`. Once found, it swaps the elements at indices `i` and `j`, where `j` is the index of the next element with the correct parity.

By performing these swaps, the algorithm ensures that each element is at the correct position based on its parity.

This approach optimizes the sorting process by performing a single pass through the array and minimizing the number of swaps required to achieve the desired parity arrangement.

Complexity

- Runtime: `O(N^2)`, where `N = nums.length`.
- Extra space: `O(1)`.

14.3.3 Solution 2: Two pointers - Make use of the problem's constraints

In the Bubble Sort approach, you do not make use of the ***constraint*** that **half of the integers in `nums` are even**. Because of that, these are unnecessary things:

1. The loops scan through full `nums`.
2. The loops are nested. That increases the complexity.
3. The `swap(nums[i], nums[j])` happens even when `nums[j]` was already in place, i.e. `nums[j]` had the same parity with `j` (Why to move it?).

Here is a two-pointer approach which takes the important ***constraint*** into account.

Code

```
#include<vector>
#include<iostream>
#include <algorithm>
using namespace std;
vector<int> sortArrayByParityII(vector<int>& nums) {
    int N = nums.size();
    int evenPos = 0;
    int oddPos = N - 1;
    while (evenPos < N) {
```

(continues on next page)

(continued from previous page)

```
        // find the nums[evenPos] that is odd for swapping
        while (evenPos < N && nums[evenPos] % 2 == 0) {
            evenPos += 2;
        }
        // If not found, it means all even nums are in place. Done!
        if (evenPos >= N) {
            break;
        }
        // Otherwise, the problem's constraint makes sure
        // there must be some nums[oddPos] that is even for swapping
        while (oddPos >= 0 && nums[oddPos] % 2 == 1) {
            oddPos -= 2;
        }
        swap(nums[evenPos], nums[oddPos]);
    }
    return nums;
}
void print(vector<int>& nums) {
    for (auto num : nums) {
        cout << num << " ";
    }
    cout << endl;
}
int main() {
    vector<int> nums = {4,2,5,7};
    auto result = sortArrayByParityII(nums);
    print(result);
    nums = {1,0,7,3,8,9,2,5,4,1,2,4};
    result = sortArrayByParityII(nums);
    print(result);
    nums = {3,4};
    result = sortArrayByParityII(nums);
    print(result);
    nums = {648,831,560,986,192,424,997,829,897,843};
    result = sortArrayByParityII(nums);
    print(result);
```

(continues on next page)

(continued from previous page)

```
}
```

```
Output:
4 5 2 7
0 1 8 3 2 9 4 5 2 1 4 7
4 3
648 831 560 997 192 829 986 897 424 843
```

Complexity

- Runtime: `O(N)`, where `N = nums.length`.
- Extra space: `O(1)`.

14.3.4 Conclusion

Solution 2 uses two pointers, one starting from the beginning of the array (`evenPos`) and the other starting from the end (`oddPos`), to efficiently identify misplaced elements.

By incrementing `evenPos` by 2 until an odd element is found and decrementing `oddPos` by 2 until an even element is found, the algorithm can swap these elements to ensure that even-indexed elements contain even values and odd-indexed elements contain odd values. This process iterates until all even and odd elements are correctly positioned.

14.3.5 Exercise

- Rearrange Array Elements by Sign

14.4 Container With Most Water

14.4.1 Problem statement

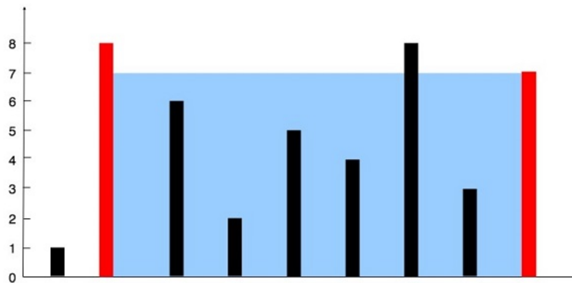
[1]You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `i-th` line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Notice that you may not slant the container.

Example 1




```
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,
↪5,4,8,3,7]. In this case, the max area of water (blue/grey section)␣
↪the container can contain is 49.
```

[1] https://leetcode.com/problems/container-with-most-water/

Example 2

```
Input: height = [1,1]
Output: 1
```

Constraints

- `n == height.length`.
- `2 <= n <= 10^5`.
- `0 <= height[i] <= 10^4`.

14.4.2 Solution 1: Bruteforce

For each line `i`, find the line `j > i` such that it gives the maximum amount of water the container `(i, j)` can store.

Code

```
#include <iostream>
#include <vector>
using namespace std;
int maxArea(const vector<int>& height) {
    int maxA = 0;
    for (int i = 0; i < height.size() - 1; i++) {
        for (int j = i + 1; j < height.size(); j++) {
            maxA = max(maxA, min(height[i], height[j]) * (j - i));
        }
    }
    return maxA;
}
int main() {
    vector<int> height{1,8,6,2,5,4,8,3,7};
    cout << maxArea(height) << endl;
```

(continues on next page)

(continued from previous page)

```
    height = {1,1};
    cout << maxArea(height) << endl;
}
```

```
Output:
49
1
```

This solution computes the maximum area of water that can be trapped between two vertical lines by iterating through all possible pairs of lines. By considering all combinations of lines and calculating the area using the formula `(min(height[i], height[j]) * (j - i))`, where `height[i]` and `height[j]` represent the heights of the two lines and `(j - i)` represents the width between them, it effectively evaluates the area formed by each pair and updates `maxA` with the maximum area encountered.

This approach optimizes the computation by exhaustively considering all possible pairs of lines and efficiently computing the area without requiring additional space.

Complexity

- Runtime: `O(n^2)`, where `n = height.length`. This is because it checks all possible pairs of vertical lines, resulting in a quadratic time complexity.
- Extra space: `O(1)`.

14.4.3 Solution 2: Two pointers

Any container has left line `i` and right line `j` satisfying `0 <= i < j < height.length`. The biggest container you want to find satisfies that condition too.

You can start from the broadest container with the left line `i = 0` and the right line `j = height.length - 1`. Then by moving `i` forward and `j` backward, you can narrow down the container to find which one will give the maximum amount of water it can store.

Depending on which line is higher, you can decide which one to move next. Since you want a bigger container, you should move the shorter line.

Example 1

For `height = [1,8,6,2,5,4,8,3,7]`:

- Starting with `i = 0` and `j = 8`.

```
area = min(height[i], height[j]) * (j - i) = min(1, 7) * (8 - 0) = 
↪8.
maxArea = 8.
```

- `height[i] = 1 < 7 = height[j]`, move `i` to 1.

```
area = min(8, 7) * (8 - 1) = 49.
maxArea = 49.
```

- `height[i] = 8 > 7 = height[j]`, move `j` to 7.

```
area = min(8, 3) * (7 - 1) = 18.
maxArea = 49.
```

- So on and so on. Final `maxArea = 49`.

Code

```
#include <iostream>
#include <vector>
using namespace std;
int maxArea(const vector<int>& height) {
    int maxA = 0;
    int i = 0;
    int j = height.size() - 1;
    while (i < j) {
        if (height[i] < height[j]) {
            maxA = max(maxA, height[i] * (j - i) );
            i++;
        } else {
            maxA = max(maxA, height[j] * (j - i) );
            j--;
```

(continues on next page)

(continued from previous page)

```
        }
    }
    return maxA;
}
int main() {
    vector<int> height{1,8,6,2,5,4,8,3,7};
    cout << maxArea(height) << endl;
    height = {1,1};
    cout << maxArea(height) << endl;
}
```

```
Output:
49
1
```

Complexity

- Runtime: `O(n)`, where `n = height.length`.
- Extra space: `O(1)`.

14.4.4 Conclusion

Solution 2 is the two-pointer approach. By initializing two pointers `i` and `j` at the beginning and end of the array respectively, and iteratively moving them towards each other until they converge, it evaluates all possible pairs of lines. At each step, it calculates the area. By moving the pointer corresponding to the shorter line inward at each step, it ensures that the maximum possible area is considered.

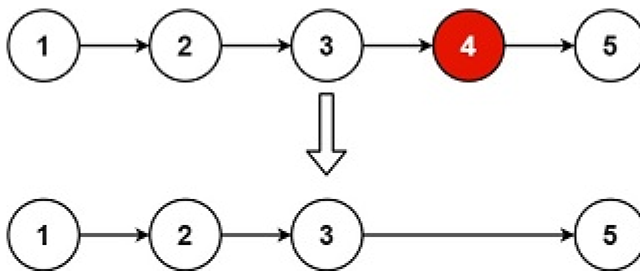
This approach optimizes the computation by avoiding redundant calculations and efficiently exploring the solution space using two pointers.

14.5 Remove Nth Node From End of List

14.5.1 Problem statement

[1]Given the head of a linked list, remove the n-th node from the end of the list and return its head.

Example 1




```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

Example 2

```
Input: head = [1], n = 1
Output: []
```

Example 3

```
Input: head = [1,2], n = 1
Output: [1]
```

[1] https://leetcode.com/problems/remove-nth-node-from-end-of-list/

Constraints

- The number of nodes in the list is `sz`.
- `1 <= sz <= 30`.
- `0 <= Node.val <= 100`.
- `1 <= n <= sz`.

Follow up

- Could you do this in one pass?

14.5.2 Solution 1: Store the nodes

Code

```
#include <iostream>
#include <vector>
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
using namespace std;
ListNode* removeNthFromEnd(ListNode* head, int n) {
    vector<ListNode*> nodes;
    ListNode* node = head;
    while (node)
    {
        nodes.push_back(node);
        node = node->next;
    }
    node = nodes[nodes.size() - n];
```

(continues on next page)

(continued from previous page)

```
    if (node == head) {
        // remove head if n == nodes.size()
        head = node->next;
    } else {
        ListNode* pre = nodes[nodes.size() - n - 1];
        pre->next = node->next;
    }
    return head;
}
void printList(const ListNode *head) {
    ListNode* node = head;
    cout << "[";
    while (node) {
        cout << node->val << ",";
        node = node->next;
    }
    cout << "]\n";
}
int main() {
    ListNode five(5);
    ListNode four(4, &five);
    ListNode three(3, &four);
    ListNode two(2, &three);
    ListNode one(1, &two);
    auto head = removeNthFromEnd(&one, 2);
    printList(head);
    head = removeNthFromEnd(&five, 1);
    printList(head);
    head = removeNthFromEnd(&four, 1);
    printList(head);
}
```

```
Output:
[1,2,3,5,]
[]
```

(continues on next page)

(continued from previous page)

```
[4,]
```

This solution uses a vector to store pointers to all nodes in the linked list, enabling easy access to the node to be removed and its predecessor.

By iterating through the linked list and storing pointers to each node in the vector, it constructs a representation of the linked list in an array-like structure. Then, it retrieves the node to be removed using its index from the end of the vector. Finally, it handles the removal of the node by updating the `next` pointer of its predecessor or updating the `head` pointer if the node to be removed is the head of the linked list.

This approach optimizes the computation by sacrificing space efficiency for simplicity of implementation and ease of manipulation of linked list elements.

Complexity

- Runtime: `O(N)`, where `N` is the number of nodes in the list.
- Extra space: `O(N)`.

14.5.3 Solution 2: Two pointers

The distance between the removed node and the end (`nullptr`) of the list is always `n`.

You can apply the two-pointer technique as follows.

Let the slower runner start after the faster one `n` nodes. Then when the faster reaches the end of the list, the slower reaches the node to be removed.

Code

```cpp
#include <iostream>
#include <vector>
struct ListNode {
    int val;
    ListNode *next;
```

(continues on next page)

(continued from previous page)

```
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
using namespace std;
ListNode* removeNthFromEnd(ListNode* head, int n) {
    ListNode* fast = head;
    // let fast goes ahead n nodes
    for (int i = 0; i < n; i++) {
        fast = fast->next;
    }
    if (fast == nullptr) {
        // remove head if n equals the list's length
        return head->next;
    }
    ListNode* slow = head;
    while (fast->next) {
        slow = slow->next;
        fast = fast->next;
    }
    // remove slow
    slow->next = slow->next->next;
    return head;
}
void printList(const ListNode *head) {
    ListNode* node = head;
    cout << "[";
    while (node) {
        cout << node->val << ",";
        node = node->next;
    }
    cout << "]\n";
}
int main() {
    ListNode five(5);
    ListNode four(4, &five);
```

(continues on next page)

(continued from previous page)

```
    ListNode three(3, &four);
    ListNode two(2, &three);
    ListNode one(1, &two);
    auto head = removeNthFromEnd(&one, 2);
    printList(head);
    head = removeNthFromEnd(&five, 1);
    printList(head);
    head = removeNthFromEnd(&four, 1);
    printList(head);
}
```

```
Output:
[1,2,3,5,]
[]
[4,]
```

Complexity

- Runtime: `O(N)`, where `N` is the number of nodes in the list.
- Extra space: `O(1)`.

14.5.4 Conclusion

Solution 2 uses two pointers, a fast pointer and a slow pointer, to remove the nth node from the end of a linked list.

Initially, both pointers start from the head of the list. The fast pointer moves `n` steps ahead, effectively positioning itself `n` nodes ahead of the slow pointer. Then, while the fast pointer is not at the end of the list, both pointers move forward simultaneously. This ensures that the slow pointer stays `n` nodes behind the fast pointer, effectively reaching the node preceding the nth node from the end when the fast pointer reaches the end of the list. Finally, the nth node from the end is removed by updating the `next` pointer of the node preceding it.

This approach optimizes the computation by traversing the linked list only once and using two pointers to efficiently locate the node to be removed.

14.5.5 Exercise

- Swapping Nodes in a Linked List

14.6 Shortest Unsorted Continuous Subarray

14.6.1 Problem statement

[1]Given an integer array `nums`, you need to find one continuous subarray that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order.

Return the shortest such subarray and output its length.

Example 1

```
Input: nums = [2,6,4,8,10,9,15]
Output: 5
Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to␣
↪make the whole array sorted in ascending order.
```

Example 2

```
Input: nums = [1,2,3,4]
Output: 0
```

[1] https://leetcode.com/problems/shortest-unsorted-continuous-subarray/

Example 3

```
Input: nums = [1]
Output: 0
```

Constraints:

- `1 <= nums.length <= 10^4.`
- `-10^5 <= nums[i] <= 10^5.`

Follow up

- Can you solve it in`O(n)` time complexity?

14.6.2 Solution 1: Sort and compare the difference

Example 1

Comparing `nums = [2,6,4,8,10,9,15]` with its sorted one `sortedNums = [2,4,6,8,9,10,15]`:

- The first position that makes the difference is `left = 1`, where `6 != 4`.
- The last (right) position that makes the difference is `right = 5`, where `9 != 10`.
- The length of that shortest subarray is `right - left + 1 = 5`.

Code

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int findUnsortedSubarray(const vector<int>& nums) {
```

(continues on next page)

(continued from previous page)

```
    vector<int> sortedNums = nums;
    sort(sortedNums.begin(), sortedNums.end());
    int left = 0;
    while (left < nums.size() && nums[left] == sortedNums[left]) {
        left++;
    }
    int right = nums.size() - 1;
    while (right >= 0 && nums[right] == sortedNums[right]) {
        right--;
    }
    return left >= right ? 0 : right - left + 1;
}
int main() {
    vector<int> nums{2,6,4,8,10,9,15};
    cout << findUnsortedSubarray(nums) << endl;
    nums = {1,2,3,4};
    cout << findUnsortedSubarray(nums) << endl;
    nums = {1};
    cout << findUnsortedSubarray(nums) << endl;
}
```

```
Output:
5
0
0
```

This solution compares the original array with a sorted version of itself to identify the unsorted boundaries efficiently.

Complexity

- Runtime: `O(N*logN)` due to the sorting step, where `N` is the number of elements in the `nums` vector.
- Extra space: `O(N)`.

14.6.3 Solution 2: Comparing only maximum and minimum elements

Assume the subarray `A = [nums[0], ..., nums[i - 1]]` is sorted. What would be the wanted `right` position for the subarray `B = [nums[0], ..., nums[i - 1], nums[i]]`?

If `nums[i]` is smaller than `max(A)`, the longer subarray `B` is not in ascending order. You might need to sort it, which means `right = i`.

Similarly, assume the subarray `C = [nums[j + 1], ..., nums[n - 1]]` is sorted. What would be the wanted `left` position for the subarray `D = [nums[j], nums[j + 1], ..., nums[n - 1]]`?

If `nums[j]` is bigger than `min(C)`, the longer subarray `D` is not in ascending order. You might need to sort it, which means `left = j`

Code

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int findUnsortedSubarray(const vector<int>& nums) {
    const int n = nums.size();
    int right = 0;
    int max = nums[0];
    for (int i = 0; i < nums.size(); i++) {
        if (nums[i] < max) {
            right = i;
        } else {
            max = nums[i];
        }
    }
    int left = n - 1;
    int min = nums[n - 1];
    for (int j = n - 1; j >= 0; j--) {
        if (nums[j] > min) {
            left = j;
```

(continues on next page)

(continued from previous page)

```
        } else {
            min = nums[j];
        }
    }
    return left >= right ? 0 : right - left + 1;
}
int main() {
    vector<int> nums{2,6,4,8,10,9,15};
    cout << findUnsortedSubarray(nums) << endl;
    nums = {1,2,3,4};
    cout << findUnsortedSubarray(nums) << endl;
    nums = {1};
    cout << findUnsortedSubarray(nums) << endl;
}
```

```
Output:
5
0
0
```

This solution determines the boundaries of the unsorted subarray by iterating through the array from both ends. It starts by initializing the `right` boundary to the beginning of the array and tracks the maximum element encountered so far. It iterates from the beginning of the array towards the end, updating the `right` boundary whenever an element smaller than the current maximum is encountered. This identifies the rightmost position where the array is unsorted.

Similarly, it initializes the `left` boundary to the end of the array and tracking the minimum element encountered so far. It iterates from the end of the array towards the beginning, updating the `left` boundary whenever an element greater than the current minimum is encountered. This identifies the leftmost position where the array is unsorted.

Finally, it returns the length of the unsorted subarray, calculated as `right - left + 1`, unless the left boundary is greater than or equal to the right boundary, in which case the array is already sorted, and it returns 0.

This approach optimizes the computation by traversing the array only twice, once

from the end and once from the beginning, to efficiently determine the boundaries of the unsorted subarray.

Complexity

- Runtime: `O(N)`, where `N` is the number of elements in the `nums` vector.
- Extra space: `O(1)`.

14.6.4 Key Takeaway

Solution 2 helped you identify the shortest subarray (by the `left` and `right` indices) needed to be sorted in order to sort the whole array.

That means in some cases you can sort an array with complexity `O(N + m*logm) < O(N*logN)` where `N` is the length of the whole array and `m` is the length of the shortest subarray.

CHAPTER

FIFTEEN

MATHEMATICS

This chapter will explore how mathematics and programming create efficient solutions. We'll cover mathematical concepts and show how they can be integrated into coding to enhance problem-solving skills.

Mathematics and programming complement each other and can lead to innovative outcomes. By applying mathematical principles, you can refine algorithms, identify patterns, streamline processes, and better understand your code's underlying logic.

What this chapter covers:

1. **Introduction to Mathematics in Coding:** Set the stage by understanding the symbiotic relationship between mathematics and programming and how mathematical concepts enrich your coding toolkit.
2. **Number Theory and Modular Arithmetic:** Delve into number theory, understanding modular arithmetic and its applications.
3. **Combinatorics and Probability:** Uncover the power of combinatorial mathematics and probability theory in solving problems related to permutations, combinations, and statistical analysis.
4. **Problem-Solving with Mathematics:** Develop strategies for leveraging mathematical concepts to solve problems efficiently and elegantly, from optimization tasks to simulation challenges.

15.1 Excel Sheet Column Number

15.1.1 Problem statement

[1]Given a string `columnTitle` that represents the column title as appears in an Excel sheet, return its corresponding column number.

For example:

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
...
```

Example 1

```
Input: columnTitle = "A"
Output: 1
```

Example 2

```
Input: columnTitle = "AB"
Output: 28
```

[1] https://leetcode.com/problems/excel-sheet-column-number/

Example 3

```
Input: columnTitle = "ZY"
Output: 701
```

Constraints

- `1 <= columnTitle.length <= 7`.
- `columnTitle` consists only of uppercase English letters.
- `columnTitle` is in the range `["A", "FXSHRXW"]`.

15.1.2 Solution: Finding The Pattern

Let us write down some other `columnTitle` strings and its value.

```
"A"   =  1
"Z"   =  26
"AA"  =  27
"AZ"  =  52
"ZZ"  =  702
"AAA" =  703
```

Then try to find the pattern

```
"A"   =  1    =  1
"Z"   =  26   =  26
"AA"  =  27   =  26 + 1
"AZ"  =  52   =  26 + 26
"ZZ"  =  702  =  26*26 + 26
"AAA" =  703  =  26*26 + 26 + 1
```

If you map `'A' = 1, ..., 'Z' = 26`, the values can be rewritten as

```
"A"   =  1    =  'A'
"Z"   =  26   =  'Z'
```

(continues on next page)

(continued from previous page)

```
"AA"  =  27   =  26*'A' + 'A'
"AZ"  =  52   =  26*'A' + 'Z'
"ZZ"  =  702  =  26*'Z' + 'Z'
"AAA" =  703  =  26*26*'A' + 26*'A' + 'A'
```

In general the formula for a string `columnTitle = abcd` is

```
abcd = 26^3*a + 26^2*b + 26*c + d,
```

where `a, b, c, d` are some uppercase English letters `A, ..., Z`.

Longer `columnTitle`s will have bigger leading exponents of `26`.

Code

```
#include <iostream>
using namespace std;
int titleToNumber(const string& columnTitle) {
    int column = 0;
    for (auto& c : columnTitle) {
        // The ASCII value of 'A' is 65.
        column = 26*column + (c - 64);
    }
    return column;
}
int main() {
    cout << titleToNumber("A") << endl;
    cout << titleToNumber("AB") << endl;
    cout << titleToNumber("ZY") << endl;
}
```

```
Output:
1
28
701
```

The solution calculates the decimal representation of the Excel column title by processing each character and updating the result.

Complexity

- Runtime: `O(N)`, where `N = columnTitle.length`.
- Extra space: `O(1)`.

Implementation notes

1. There are many ways to compute the series

   ```
   26^3*a + 26^2*b + 26*c + d.
   ```

   If you write it as

   ```
   26*(26*(26*(0 + a) + b) + c) + d,
   ```

   you get the loop in the code above.
2. To map `'A' = 1, ..., 'Z' = 26`, you can use their ASCII values (`'A' = 65, ..., 'Z' = 90`) minus `64`.
3. The parentheses around `(c - 64)` is needed. Otherwise the value of `columnTitle = "FXSHRXW"` makes `26*column + c` exceed the limit of `int` before it subtracts `64`.

15.1.3 Exercise

- Excel Sheet Column Title

15.2 Power of Three

15.2.1 Problem statement

[1]Given an integer `n`, return `true` if it is a power of three. Otherwise, return `false`.

An integer `n` is a power of three, if there exists an integer `x` such that `n == 3^x`.

Example 1

```
Input: n = 27
Output: true
Explanation: 27 = 3^3.
```

Example 2

```
Input: n = 0
Output: false
Explanation: There is no x where 3^x = 0.
```

Example 3

```
Input: n = -1
Output: false
Explanation: There is no x where 3^x = (-1).
```

[1] https://leetcode.com/problems/power-of-three/

Constraints

- `-2^31 <= n <= 2^31 - 1`.

Follow up

- Could you solve it without loops/recursion?

15.2.2 Solution 1: Repeat the division

Code

```
#include <iostream>
using namespace std;
bool isPowerOfThree(int n) {
    while (n % 3 == 0 && n > 0) {
        n /= 3;
    }
    return n == 1;
}
int main() {
    cout << isPowerOfThree(27) << endl;
    cout << isPowerOfThree(0) << endl;
    cout << isPowerOfThree(-1) << endl;
}
```

```
Output:
1
0
0
```

This solution repeatedly divides the input by 3 until it either becomes 1 (indicating that it was a power of three) or cannot be divided further by 3.

Complexity

- Runtime: `O(logn)`.
- Extra space: `O(1)`.

15.2.3 Solution 2: Mathematics and the constraints of the problem

A power of three must divide another bigger one, i.e. $3^x | 3^y$ where $0 \leq x \leq y$.

Because the constraint of the problem is $n \leq 2^{31} - 1$, you can choose the biggest power of three in this range to test the others.

It is $3^{19} = 1162261467$. The next power will exceed $2^{31} = 2147483648$.

Code

```cpp
#include <iostream>
using namespace std;
bool isPowerOfThree(int n) {
    return n > 0 && 1162261467 % n == 0;
}
int main() {
    cout << isPowerOfThree(27) << endl;
    cout << isPowerOfThree(0) << endl;
    cout << isPowerOfThree(-1) << endl;
}
```

```
Output:
1
0
0
```

This solution effectively checks whether `n` is a power of three by verifying if it is a divisor of the largest power of three that fits within 32 bits. If the condition is met, it returns `true`, indicating that `n` is a power of three; otherwise, it returns `false`.

Complexity

- Runtime: `O(1)`.
- Extra space: `O(1)`.

15.2.4 Readable code

Though Solution 2 offers a direct approach without the need for iteration, it is not easy to understand like Solution 1, where complexity of `O(logn)` is not too bad.

15.2.5 Exercise

- Check if Number is a Sum of Powers of Three

15.3 Best Time to Buy and Sell Stock

15.3.1 Problem statement

[1]You are given an array `prices` where `prices[i]` is the price of a given stock on the `i-th` day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

[1] https://leetcode.com/problems/best-time-to-buy-and-sell-stock/

Example 1

```
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6),␣
↪profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because␣
↪you must buy before you sell.
```

Example 2

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit␣
↪= 0.
```

Constraints

- `1 <= prices.length <= 10^5`.
- `0 <= prices[i] <= 10^4`.

15.3.2 Solution 1: Bruteforce

For each day `i`, find the day `j > i` that gives maximum profit.

Code

```cpp
#include <vector>
#include <iostream>
using namespace std;
int maxProfit(const vector<int>& prices) {
    int maxProfit = 0;
    for (int i = 0; i < prices.size(); i++) {
```

(continues on next page)

(continued from previous page)

```
        for (int j = i + 1; j < prices.size(); j++) {
            if (prices[j] > prices[i]) {
                maxProfit = max(maxProfit, prices[j] - prices[i]);
            }
        }
    }
    return maxProfit;
}
int main() {
    vector<int> prices{7,1,5,3,6,4};
    cout << maxProfit(prices) << endl;
    prices = {7,6,4,3,1};
    cout << maxProfit(prices) << endl;
}
```

```
Output:
5
0
```

This solution uses a brute force approach to find the maximum profit. It compares the profit obtained by buying on each day with selling on all subsequent days and keeps track of the maximum profit found.

Complexity

- Runtime: `O(N^2)`, where `N = prices.length`.
- Extra space: `O(1)`.

15.3.3 Solution 2: Smallest and largest prices

Given a past day `i`, the future day `j > i` that gives the maximum profit is the day that has the largest price which is bigger than `prices[i]`.

Conversely, given a future day `j`, the past day `i < j` that gives the maximum profit is the day with the smallest price.

Code

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int maxProfit(const vector<int>& prices) {
    int maxProfit = 0;
    int i = 0;
    while (i < prices.size()) {
        // while prices are going down,
        // find the bottommost one to start
        while (i < prices.size() - 1 && prices[i] >= prices[i + 1]) {
            i++;
        }
        // find the largest price in the future
        auto imax = max_element(prices.begin() + i, prices.end());
        // find the smallest price in the past
        auto imin = min_element(prices.begin() + i, imax);
        maxProfit = max(maxProfit, *imax - *imin);
        // next iteration starts after the found largest price
        i = distance(prices.begin(), imax) + 1;
    }
    return maxProfit;
}
int main() {
    vector<int> prices{7,1,5,3,6,4};
    cout << maxProfit(prices) << endl;
    prices = {7,6,4,3,1};
    cout << maxProfit(prices) << endl;
    prices = {2,4,1,7};
    cout << maxProfit(prices) << endl;
    prices = {2,4,1};
    cout << maxProfit(prices) << endl;
}
```

```
Output:
```

(continues on next page)

(continued from previous page)

```
5
0
6
2
```

This solution optimally finds the maximum profit by iterating through the array only once, avoiding the need for nested loops.

Complexity

- Runtime: `O(N)`, where `N = prices.length`.
- Extra space: `O(1)`.

15.3.4 Solution 3: Only the smallest price

Given a future day `j`, the past day `i` that gives the maximum profit is the day with minimum price.

Code

```
#include <vector>
#include <iostream>
using namespace std;
int maxProfit(const vector<int>& prices) {
    int maxProfit = 0;
    // keep track the minimum price so fat
    int minPrice = prices[0];
    for (int i = 1; i < prices.size(); i++)  {
        // update the minimum price
        minPrice = min(minPrice, prices[i]);
        maxProfit = max(maxProfit, prices[i] - minPrice);
    }
    return maxProfit;
}
```

(continues on next page)

(continued from previous page)

```
int main() {
    vector<int> prices{7,1,5,3,6,4};
    cout << maxProfit(prices) << endl;
    prices = {7,6,4,3,1};
    cout << maxProfit(prices) << endl;
    prices = {2,4,1,7};
    cout << maxProfit(prices) << endl;
    prices = {2,4,1};
    cout << maxProfit(prices) << endl;
}
```

```
Output:
5
0
6
2
```

This solution efficiently computes the maximum profit by iterating through the array only once, maintaining the minimum buying price and updating the maximum profit accordingly.

Complexity

- Runtime: `O(N)`, where `N = prices.length`.
- Extra space: `O(1)`.

15.3.5 Conclusion

The problem of finding the maximum profit that can be achieved by buying and selling a stock can be efficiently solved using different approaches. Solutions 1, 2, and 3 each offer a different approach to solving the problem, including brute-force iteration, finding local minima and maxima, and maintaining a running minimum price.

Solution 3 stands out as the most efficient approach, achieving a linear time complexity by iterating through the prices only once and updating the minimum price

seen so far. This approach avoids unnecessary comparisons and achieves the desired result in a single pass through the array.

15.3.6 Exercise

- Best Time to Buy and Sell Stock II

15.4 Subsets

15.4.1 Problem Statement

[1]Given an integer array `nums` of unique elements, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1

```
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

Example 2

```
Input: nums = [1]
Output: [[],[1]]
```

[1] https://leetcode.com/problems/subsets/

Constraints

- `1 <= nums.length <= 10`.
- `-10 <= nums[i] <= 10`.
- All the numbers of `nums` are unique.

15.4.2 Solution

You might need to find the relationship between the result of the array `nums` with the result of itself without the last element.

Example 3

```
Input: nums = [1,2]
Output: [[],[1],[2],[1,2]]
```

You can see the powerset of Example 3 was obtained from the one in Example 2 with additional subsets `[2]`, `[1,2]`. These new subsets were constructed from subsets `[]`, `[1]` of Example 2 appended with the new element `2`.

Similarly, the powerset of Example 1 was obtained from the one in Example 3 with the additional subsets `[3]`, `[1,3]`, `[2,3]`, `[1,2,3]`. These new subsets were constructed from the ones of Example 3 appended with the new element `3`.

Code

```
#include <vector>
#include <iostream>
using namespace std;
vector<vector<int>> subsets(const vector<int>& nums) {
    vector<vector<int>> powerset = {{}};
    int i = 0;
    while (i < nums.size()) {
        vector<vector<int>> newSubsets;
```

(continues on next page)

(continued from previous page)

```
        for (auto subset : powerset) {
            subset.push_back(nums[i]);
            newSubsets.push_back(subset);
        }
        powerset.insert(powerset.end(), newSubsets.begin(), newSubsets.
↪end());
        i++;
    }
    return powerset;
}
void print(const vector<vector<int>>& powerset) {
    for (auto& set : powerset ) {
        cout << "[";
        for (auto& element : set) {
            cout << element << ",";
        }
        cout << "]";
    }
    cout << endl;
}
int main() {
    vector<int> nums{1,2,3};
    auto powerset = subsets(nums);
    print(powerset);
    nums = {1};
    powerset = subsets(nums);
    print(powerset);
}
```

```
Output:
[][1,][2,][1,2,][3,][1,3,][2,3,][1,2,3,]
[][1,]
```

Complexity

- Runtime: `O(2^N)`, where `N` is the number of elements in `nums`, as it generates all possible subsets.
- Extra space: `O(2^N)` due to the space required to store the subsets.

15.4.3 Conclusion

This solution generates subsets by iteratively adding each element of `nums` to the existing subsets and accumulating the results.

Note that in `for (auto subset : powerset)` you should not use reference `auto&` because we do not want to change the subsets that have been created.

15.4.4 Exercise

- Subsets II

15.5 Minimum Moves to Equal Array Elements II

15.5.1 Problem statement

[1]Given an integer array `nums` of size `n`, return the minimum number of moves required to make all array elements equal.

In one move, you can increment or decrement an element of the array by `1`.

[1] https://leetcode.com/problems/minimum-moves-to-equal-array-elements-ii/

Example 1

```
Input: nums = [1,2,3]
Output: 2
Explanation:
Only two moves are needed (remember each move increments or decrements␣
↪one element):
[1,2,3]  =>  [2,2,3]  =>  [2,2,2]
```

Example 2

```
Input: nums = [1,10,2,9]
Output: 16
```

Constraints

- `n == nums.length`.
- `1 <= nums.length <= 10^5`.
- `-10^9 <= nums[i] <= 10^9`.

15.5.2 Solution 1: Median - The math behind the problem

You are asked to move all elements of an array to the same value `M`. The problem can be reduced to identifying what `M` is.

First, moving elements of an unsorted array and moving a sorted one are the same. So you can assume `nums` is sorted in some order. Let us say it is sorted in ascending order.

Second, `M` must be in between the minimum element and the maximum one. Apparently!

We will prove that `M` will be the median of `nums`, which is `nums[n/2]` of the sorted `nums`.

In other words, we will prove that if you choose M a value different from nums[n/2], then the number of moves will be increased.

In fact, if you choose M = nums[n/2] + x, where x > 0, then:

- Each element nums[i] that is less than M needs more x moves, while each nums[j] that is greater than M can reduce x moves.
- But the number of nums[i] is bigger than the number of nums[j].
- So the total number of moves is bigger.

The same arguments apply for x < 0.

Example 3

For nums = [0,1,2,2,10]. Its median is 2. The minimum number of moves is 2 + 1 + 0 + 0 + 8 = 11.

If you choose M = 3 (the average value, the mean), the total number of moves is 3 + 2 + 1 + 1 + 7 = 14.

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int minMoves2(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    const int median = nums[nums.size() / 2];
    int moves = 0;
    for (int& a: nums) {
        moves += abs(a - median);
    }
    return moves;
}
int main() {
    vector<int> nums{1,2,3};
```

(continues on next page)

(continued from previous page)

```
    cout << minMoves2(nums) << endl;
    nums = {1,10,2,9};
    cout << minMoves2(nums) << endl;
}
```

```
Output:
2
16
```

This solution leverages the concept of the median to minimize the total absolute differences between each element and the median, resulting in the minimum number of moves to equalize the array.

Complexity

- Runtime: `O(n*logn)` due to the sorting step, where `n` is the number of elements in the `nums` array.
- Extra space: `O(1)`.

15.5.3 Solution 2: Using `std::nth_element` to compute the median

What you only need in Solution 1 is the median value. Computing the total number of moves in the `for` loop does not require the array `nums` to be fully sorted.

In this case, you can use `std::nth_element` to reduce the runtime complexity.

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int minMoves2(vector<int>& nums) {
    const int mid = nums.size() / 2;
```

(continues on next page)

(continued from previous page)

```
    // make sure all elements that are less than or equals to nums[mid]
    // are on the left
    std::nth_element(nums.begin(), nums.begin() + mid, nums.end());
    const int median = nums[mid];
    int moves = 0;
    for (int& a: nums) {
        moves += abs(a - median);
    }
    return moves;
}
int main() {
    vector<int> nums{1,2,3};
    cout << minMoves2(nums) << endl;
    nums = {1,10,2,9};
    cout << minMoves2(nums) << endl;
}
```

```
Output:
2
16
```

This solution efficiently finds the median of the `nums` array in linear time using `std::nth_element` and then calculates the minimum number of moves to make all elements equal to this median.

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: `O(1)`.

15.5.4 Modern C++ tips

In the code of Solution 2, the partial sorting algorithm `std::nth_element` will make sure for all indices `i` and `j` that satisfy `0 <= i <= mid <= j < nums.length`, then

```
nums[i] <= nums[mid] <= nums[j].
```

With this property, if `mid = nums.length / 2`, then the value of `nums[mid]` is unchanged no matter how `nums` is sorted or not.

15.5.5 Exercise

- Minimum Moves to Equal Array Elements

15.6 Array Nesting

15.6.1 Problem statement

[1]You are given an integer array `nums` of length `n` where `nums` is a permutation of the numbers in the range `[0, n - 1]`.

You should build a set `s[k] = {nums[k], nums[nums[k]], nums[nums[nums[k]]], ... }` subjected to the following rule:

- The first element in `s[k]` starts with the element `nums[k]`.
- The next element in `s[k]` should be `nums[nums[k]]`, and then `nums[nums[nums[k]]]`, and so on.
- We stop adding elements before a duplicate element occurs in `s[k]`.

Return the length of the longest set `s[k]`.

[1] https://leetcode.com/problems/array-nesting/

Example 1

```
Input: nums = [5,4,0,3,1,6,2]
Output: 4
Explanation:
nums[0] = 5, nums[1] = 4, nums[2] = 0, nums[3] = 3, nums[4] = 1,␣
↪nums[5] = 6, nums[6] = 2.
One of the longest sets s[k]:
s[0] = {nums[0], nums[5], nums[6], nums[2]} = {5, 6, 2, 0}
```

Example 2

```
Input: nums = [0,1,2]
Output: 1
```

Constraints:

- `1 <= nums.length <= 10^5`.
- `0 <= nums[i] < nums.length`.
- All the values of `nums` are unique.

15.6.2 Solution: Understanding the math behind

A permutation is a one-to-one mapping from a set of integers to itself.

The permutation on the set `nums` in this problem is defined by the mapping `i -> nums[i]`. For instance in Example 1, the permutation is defined as following:

```
0 -> 5,
1 -> 4,
2 -> 0,
3 -> 3,
4 -> 1,
```

(continues on next page)

(continued from previous page)

```
5 -> 6,
6 -> 2.
```

You can always rearrange the definition of a permutation into groups of cyclic chains (factors).

```
0 -> 5, 5 -> 6, 6 -> 2, 2 -> 0,
1 -> 4, 4 -> 1,
3 -> 3
```

The set `s[k]` in this problem is such a chain. In mathematics, it is called a *cycle*; because the chain `(0, 5, 6, 2)` is considered the same as `(5, 6, 2, 0)`, `(6, 2, 0, 5)` or `(2, 0, 5, 6)` in Example 1.

Assume you have used some elements of the array `nums` to construct some cycles. To construct another one, you should start with the unused elements.

The problem leads to finding the longest cycle of a given permutation.

Code

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int arrayNesting(const vector<int>& nums) {
    int maxLen{0};
    vector<bool> visited(nums.size());
    for (auto& i : nums) {
        if (visited[i]) {
            continue;
        }
        int len{0};
        // visit the cycle starting from i
        while (!visited[i]) {
            visited[i] = true;
            i = nums[i];
```

(continues on next page)

(continued from previous page)

```
            len++;
        }
        maxLen = max(len, maxLen);
    }
    return maxLen;
}

int main() {
    vector<int> nums = {5,4,0,3,1,6,2};
    cout << arrayNesting(nums) << endl;
    nums = {0,1,2};
    cout << arrayNesting(nums) << endl;
    nums = {0,2,1};
    cout << arrayNesting(nums) << endl;
    nums = {2,0,1};
    cout << arrayNesting(nums) << endl;
}
```

```
Output:
4
1
2
3
```

Complexity

- Runtime: `O(n)` where `n` is the size of the `nums` array.
- Extra space: much less than `O(n)` since `vector<bool>` is optimized for space efficiency.

15.6.3 Conclusion

The problem of finding the length of the longest cycle in an array can be efficiently solved using a cycle detection approach. This solution efficiently detects cycles in the array by using a boolean array to mark visited elements.

By iterating through each element in the array and visiting the cycle starting from each unvisited element, the solution identifies the length of each cycle and updates the maximum length accordingly. This approach ensures that each cycle is visited only once and maximizes the length of the longest cycle in the array.

15.7 Count Sorted Vowel Strings

15.7.1 Problem statement

[1]Given an integer `n`, return the number of strings of length `n` that consist only of vowels (`a`, `e`, `i`, `o`, `u`) and are lexicographically sorted.

A string `s` is lexicographically sorted if for all valid `i`, `s[i]` is the same as or comes before `s[i+1]` in the alphabet.

Example 1

```
Input: n = 1
Output: 5
Explanation: The 5 sorted strings that consist of vowels only are ["a",
↪"e","i","o","u"].
```

[1] https://leetcode.com/problems/count-sorted-vowel-strings/

Example 2

```
Input: n = 2
Output: 15
Explanation: The 15 sorted strings that consist of vowels only are
["aa","ae","ai","ao","au","ee","ei","eo","eu","ii","io","iu","oo","ou",
↪"uu"].
Note that "ea" is not a valid string since 'e' comes after 'a' in the␣
↪alphabet.
```

Example 3

```
Input: n = 33
Output: 66045
```

Constraints

- `1 <= n <= 50`.

15.7.2 Solution 1: Finding the pattern

Let us find the relationship of the strings between the vowels.

Example 3

For `n = 3`:

- There is (always) only one string starting from `u`, which is `uuu`.
- There are 3 strings starting from `o`: `ooo`, `oou` and `ouu`.
- There are 6 strings starting from `i`: `iii`, `iio`, `iiu`, `ioo`, `iou`, `iuu`.
- There are 10 strings starting from `e`: `eee`, `eei`, `eeo`, `eeu`, `eii`, `eio`, `eiu`, `eoo`, `eou`, `euu`.

- There are 15 strings starting from `a`: `aaa`, `aae`, `aai`, `aao`, `aau`, `aee`, `aei`, `aeo`, `aeu`, `aii`, `aio`, `aiu`, `aoo`, `aou`, `auu`.
- In total: there are 35 strings that satisfy the problem.

Findings

In Example 3, if you ignore the leading vowel of those strings, then the shorted strings of the line above all appear in the ones of the line below and the remaining strings of the line below come from `n = 2`.

More precisely:

- All the shorted strings `oo`, `ou` and `uu` starting from `o` appear on the ones starting from `i`. The remaining `ii`, `io`, `iu` starting from `i` come from the strings of length `n = 2` (see Example 2).
- Similarly, all shorted strings `ii`, `io`, `iu`, `oo`, `ou`, `uu` starting from `i` appear on the ones starting from `e`. The remaining `ee`, `ei`, `eo`, `eu` come from `n = 2`.
- And so on.

That leads to the following recursive relationship.

Let `S(x, n)` be the number of strings of length `n` starting from a vowel `x`. Then

- `S('o', n) = S('o', n - 1) + S('u', n)` for all `n > 1`.
- `S('i', n) = S('i', n - 1) + S('o', n)` for all `n > 1`.
- `S('e', n) = S('e', n - 1) + S('i', n)` for all `n > 1`.
- `S('a', n) = S('a', n - 1) + S('e', n)` for all `n > 1`.
- `S(x, 1) = 1` for all vowels `x`.
- `S('u', n) = 1` for all `n >= 1`.

For this problem, you want to compute

```
S(n) = S('a', n) + S('e', n) + S('i', n) + S('o', n) + S('u', n).
```

Code

```
#include <iostream>
using namespace std;
int countVowelStrings(int n) {
    int a, e, i, o, u;
    a = e = i = o = u = 1;
    while (n > 1) {
        o += u;
        i += o;
        e += i;
        a += e;
        n--;
    }
    return a + e + i + o + u;
}
int main() {
    cout << countVowelStrings(1) << endl;
    cout << countVowelStrings(2) << endl;
    cout << countVowelStrings(33) << endl;
}
```

```
Output:
5
15
66045
```

This solution efficiently computes the count of vowel strings of length `n` using dynamic programming, updating the counts based on the previous lengths to avoid redundant calculations.

Complexity

- Runtime: `O(n)`.
- Extra space: `O(1)`.

15.7.3 Solution 2: The math behind the problem

The strings of length `n` you want to count are formed by a number of `'a'`, then some number of `'e'`, then some number of `'i'`, then some number of `'o'` and finally some number of `'u'`.

So it looks like this

```
s = "aa..aee..eii..ioo..ouu..u".
```

And you want to count how many possibilities of such strings of length `n`.

One way to count it is using combinatorics in mathematics.

If you separate the groups of vowels by `'|'` like this

```
s = "aa..a|ee..e|ii..i|oo..o|uu..u",
```

the problem becomes counting how many ways of putting those 4 separators `'|'` to form a string of length `n + 4`.

In combinatorics, the solution is $\binom{n+4}{4}$, where $\binom{n}{k}$ is the binomial coefficient:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

The final number of strings is

$$\binom{n+4}{4} = \frac{(n+4)!}{4!n!} = \frac{(n+1)(n+2)(n+3)(n+4)}{24}.$$

Code

```cpp
#include <iostream>
using namespace std;
int countVowelStrings(int n) {
    return (n + 1) * (n + 2) * (n + 3) * (n + 4) / 24;
}
int main() {
    cout << countVowelStrings(1) << endl;
    cout << countVowelStrings(2) << endl;
    cout << countVowelStrings(33) << endl;
}
```

```
Output:
5
15
66045
```

Complexity

- Runtime: `O(1)`.
- Extra space: `O(1)`.

15.7.4 Conclusion

The problem of counting the number of strings of length `n` that consist of the vowels 'a', 'e', 'i', 'o', and 'u' in sorted order can be efficiently solved using combinatorial techniques. Solution 1 uses dynamic programming to iteratively calculate the count of strings for each length up to `n`, updating the counts based on the previous counts. This approach efficiently computes the count of sorted vowel strings for the given length `n` without requiring excessive memory usage or computational overhead.

Solution 2 offers a more direct approach by utilizing a combinatorial formula to calculate the count of sorted vowel strings directly based on the given length `n`. By leveraging the combinatorial formula, this solution avoids the need for iterative calculations and achieves the desired result more efficiently.

15.8 Concatenation of Consecutive Binary Numbers

15.8.1 Problem statement

[1]Given an integer `n`, return the decimal value of the binary string formed by concatenating the binary representations of `1` to `n` in order, modulo `10^9 + 7`.

Example 1

```
Input: n = 1
Output: 1
Explanation: "1" in binary corresponds to the decimal value 1.
```

Example 2

```
Input: n = 3
Output: 27
Explanation: In binary, 1, 2, and 3 corresponds to "1", "10", and "11".
After concatenating them, we have "11011", which corresponds to the␣
↪decimal value 27.
```

Example 3

```
Input: n = 12
Output: 505379714
Explanation: The concatenation results in
↪"1101110010111011110001001101010111100".
The decimal value of that is 118505380540.
After modulo 10^9 + 7, the result is 505379714.
```

[1] https://leetcode.com/problems/concatenation-of-consecutive-binary-numbers/

Constraints

- `1 <= n <= 10^5`.

15.8.2 Solution: Recursive

There must be some relationship between the result of `n` and the result of `n - 1`.

First, let us list some first values of `n`.

- For `n = 1`: the final binary string is `"1"`, its decimal value is `1`.
- For `n = 2`: the final binary string is `"110"`, its decimal value is `6`.
- For `n = 3`: the final binary string is `"11011"`, its decimal value is `27`.

Look at `n = 3`, you can see the relationship between the decimal value of `"11011"` and the one of `"110"` (of `n = 2`) is:

```
27 = 6 * 2^2 + 3
Dec("11011") = Dec("110") * 2^num_bits("11") + Dec("11")
Result(3) = Result(2) * 2^num_bits(3) + 3.
```

The same equation for `n = 2`:

```
6 = 1 * 2^2 + 2
Dec("110") = Dec("1") * 2^num_bits("10") + Dec("10")
Result(2) = Result(1) * 2^num_bits(2) + 2.
```

In general, the recursive relationship between `n` and `n - 1` is:

```
Result(n) = Result(n - 1) * 2^num_bits(n) + n.
```

Code

```
#include <cmath>
#include <iostream>
int concatenatedBinary(int n) {
    unsigned long long result = 1;
```

(continues on next page)

(continued from previous page)

```
    for (int i = 2; i <= n; i++) {
        const int num_bits = std::log2(i) + 1;
        result = ((result << num_bits) + i) % 1000000007;
    }
    return result;
}
int main() {
    std::cout << concatenatedBinary(1) << std::endl;
    std::cout << concatenatedBinary(3) << std::endl;
    std::cout << concatenatedBinary(12) << std::endl;
}
```

```
Output:
1
27
505379714
```

Complexity

- Runtime: `O(n*logn)`.
- Extra space: `O(1)`.

15.8.3 Conclusion

This solution efficiently calculates the concatenated binary representation of integers from `1` to `n`, using bitwise operations and modular arithmetic. Note that `a << t` is equivalent to `a * 2^t`.

15.9 Perfect Squares

15.9.1 Problem statement

[1]Given an integer `n`, return the least number of perfect square numbers that sum to `n`.

A **perfect square** is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, `1`, `4`, `9`, and `16` are perfect squares while `3` and `11` are not.

Example 1

```
Input: n = 9
Output: 1
Explanation: 9 is already a perfect square.
```

Example 2

```
Input: n = 13
Output: 2
Explanation: 13 = 4 + 9.
```

Example 3

```
Input: n = 7
Output: 4
Explanation: 7 = 4 + 1 + 1 + 1.
```

[1] https://leetcode.com/problems/perfect-squares/

Example 4

```
Input: n = 12
Output: 3
Explanation: 12 = 4 + 4 + 4.
```

Constraints

- `1 <= n <= 10^4`.

15.9.2 Solution 1: Dynamic Programming

Let us call the function to be computed `numSquares(n)`, which calculates the least number of perfect squares that sum to `n`.

Here are the findings.

1. If `n` is already a perfect square then `numSquares(n) = 1`.
2. Otherwise, it could be written as `n = 1 + (n-1)`, or `n = 4 + (n-4)`, or `n = 9 + (n-9)`, etc. which means `n` is a sum of a perfect square (`1`, `4` or `9`, etc.) and another number `m < n`. That leads to the problems `numSquares(m)` of smaller values `m`.
3. If you have gotten the results of the smaller problems `numSquares(n-1)`, `numSquares(n-4)`, `numSquares(n-9)`, etc. then `numSquares(n) = 1 + the minimum of those results`.

Example 4

`n = 12` is not a perfect square. It can be written as `n = 1 + 11 = 4 + 8 = 9 + 3`.

- For `m = 11`, it is not a perfect square and can be written as `m = 1 + 10 = 4 + 7 = 9 + 2`.
- For `m = 8`, it is not a perfect square and can be written as `m = 1 + 7 = 4 + 4` (matched). You get `numSquares(8) = 2`.
- For `m = 3`, it is not a perfect square and can be written as `m = 1 + 2`.

You can continue to compute `numSquares(m)` for other values `m` in this recursive process. But you can see the case of `m = 8` was already the best solution. And `numSquares(12) = 1 + numSquares(8) = 1 + 2 = 3`, which is the case of `n = 12 = 4 + 4 + 4`.

To improve runtime, you can apply *dynamic programming* to cache the `numSquares(n)` that you have computed.

Code

```
#include <iostream>
#include <cmath>
#include <unordered_map>
using namespace std;
//! @return         the least number of perfect squares that sum to n
//! @param[out] ns  a map stores all intermediate results
int nsq(int n, unordered_map<int, int>& ns) {
    auto it = ns.find(n);
    if (it != ns.end()) {
        return it->second;
    }
    const int sq = sqrt(n);
    if (sq * sq == n) {
        // n is already a perfect square
        ns[n] = 1;
        return 1;
    }
    // if n is written as 1 + 1 + .. + 1,
    // maximum of result is n
    int result = n;
    // finding the minimum nsq(n - i*i) across all i <= sqrt(n)
    for (int i = 1; i <= sq; i++) {
        //
        result = min(result, nsq(n - i*i, ns));
    }
    // write n as imin^2 + (n - imin^2)
    ns[n] = result + 1;
```

(continues on next page)

(continued from previous page)

```
    return ns[n];
}
int numSquares(int n) {
    unordered_map<int, int> ns;
    return nsq(n, ns);
}
int main() {
    cout << numSquares(12) << endl;
    cout << numSquares(13) << endl;
}
```

```
Output:
3
2
```

The key idea of this algorithm is to build the solution incrementally, starting from the smallest perfect squares, and use memoization to store and retrieve intermediate results. By doing this, it efficiently finds the minimum number of perfect squares required to sum up to `n`.

Complexity

- Runtime: `O(n*sqrt(n)) = O(n^(3/2))` due to the nested loops and recursive calls.
- Extra space: `O(n)`.

15.9.3 Solution 2: Number Theory

The dynamic programming solution above is good enough. But for those who are interested in Algorithmic Number Theory, there is a very interesting theorem that can solve the problem directly without recursion.

It is called Lagrange's Four-Square Theorem, which states

> *every natural number can be represented as the sum of four integer squares.*

It was proven by Lagrange in 1770.

Example 4

`n = 12 = 4 + 4 + 4 + 0` or `12 = 1 + 1 + 1 + 9`.

Applying to our problem, **`numSquares(n)` can only be 1, 2, 3, or 4. Not more.**

It turns into the problem of

> *identifying when* `numSquares(n)` *returns 1, 2, 3, or 4.*

Here are the cases.

1. If `n` is a perfect square, `numSquares(n) = 1`.
2. There is another theorem, Legendre's Three-Square Theorem, which states that `numSquares(n)` cannot be 1, 2, or 3 if `n` can be expressed as

$$n = 4^a(8 \cdot b + 7),$$

where a, b are nonnegative integers.

In other words, `numSquares(n) = 4` if `n` is of this form.

Example 3

$n = 7 = 4^0(8 \cdot 0 + 7)$. It can only be written as `7 = 4 + 1 + 1 + 1`.

Code

```cpp
#include <iostream>
#include <cmath>
using namespace std;
bool isSquare(int n) {
    int sq = sqrt(n);
    return sq * sq == n;
}
int numSquares(int n) {
    if (isSquare(n)) {
        return 1;
    }
```

(continues on next page)

(continued from previous page)

```
    // Legendre's three-square theorem
    int m = n;
    while (m % 4 == 0) {
        m /= 4;
    }
    if (m % 8 == 7) {
        return 4;
    }
    const int sq = sqrt(n);
    for (int i = 1; i <= sq; i++) {
        if (isSquare(n - i*i)) {
            return 2;
        }
    }
    return 3;
}
int main() {
    cout << numSquares(12) << endl;
    cout << numSquares(13) << endl;
}
```

```
Output:
3
2
```

This solution finds the minimum number of perfect squares required to sum up to the given integer n by first applying mathematical properties and Legendre's three-square theorem to simplify the problem and then using a loop to find possible combinations of two perfect squares.

Complexity

- Runtime: `O(sqrt(n)) = O(n^(1/2))`, and it does not require dynamic programming or memoization.
- Extra space: `O(1)`.

15.9.4 Solution 3: Further performance improvement

Lagrange's Four-Square Theorem and Legendre's Three-Square Theorem are so powerful to solve this problem. But you can still do a little more algebra to improve further the runtime of the implementation above.

Instead of looping over `sqrt(n)` in the final `for` loop, we will prove that this loop over `sqrt(m)` is enough. That will improve runtime a lot since `m` is much less than `n`.

Let `m` be the reduced value of `n` after the Legendre's `while` loop. It satisfies

$$n = 4^a \cdot m.$$

We will prove that `numSquares(n) = numSquares(m)`.

In fact, if `m` is written as $m = x^2 + y^2 + z^2$, where x, y, z are nonnegative integers. Then

$$n = 4^a \cdot m = (2^a)^2 \cdot m = (2^a \cdot x)^2 + (2^a \cdot y)^2 + (2^a \cdot z)^2.$$

In other words, `numSquares(n) = numSquares(m)`.

Now you can change directly the value `n` during the Legendre's `while` loop without affecting the final result.

Code

```
#include <iostream>
#include <cmath>
using namespace std;
bool isSquare(int n) {
    int sq = sqrt(n);
```

(continues on next page)

(continued from previous page)

```
    return sq * sq == n;
}
int numSquares(int n) {
    if (isSquare(n)) {
        return 1;
    }
    // Legendre's three-square theorem
    while (n % 4 == 0) {
        n /= 4;
    }
    if (n % 8 == 7) {
        return 4;
    }
    const int sq = sqrt(n);
    for (int i = 1; i <= sq; i++) {
        if (isSquare(n - i*i)) {
            return 2;
        }
    }
    return 3;
}
int main() {
    cout << numSquares(12) << endl;
    cout << numSquares(13) << endl;
}
```

```
Output:
3
2
```

Complexity

- Runtime: `O(sqrt(n)) = O(n^(1/2))`.
- Extra space: `O(1)`.

15.9.5 Conclusion

- The title of this coding challenge (*Perfect squares*) gives you a hint it is more about mathematics than coding technique.
- It is amazing from Lagrange's Four-Square Theorem there are only four possibilities for the answer to the problem. Not many people knowing it.
- You can get an optimal solution to a coding problem when you know something about the mathematics behind it.

Hope you learn something interesting from this code challenge.

Have fun with coding and mathematics!

15.9.6 Exercise

- Ways to Express an Integer as Sum of Powers

CHAPTER SIXTEEN

CONCLUSION

Congratulations! You have made it to the end of this book! I hope you have enjoyed and learned from the **coding challenges** and **solutions** presented in this book.

Through these challenges, you have not only improved your **coding skills** but also your **problem-solving abilities**, **logical thinking**, and **creativity**. You have been exposed to different **programming techniques** and **algorithms**, which have broadened your understanding of the programming world. These skills and knowledge will undoubtedly benefit you in your future coding endeavors.

Remember, coding challenges are not only a way to improve your coding skills but also a fun and engaging way to stay up-to-date with the latest technology trends. They can also help you to prepare for technical interviews, which are a crucial part of landing a programming job.

In conclusion, I encourage you to continue exploring the world of coding challenges, as there is always something new to learn and discover. **Keep practicing, keep learning, and keep challenging yourself**. With hard work and dedication, you can become an expert in coding and a valuable asset to any team.

APPENDIX

A

CODING CHALLENGE BEST PRACTICES

Here are some best practices to keep in mind when working on coding challenges:

A.1 Read the problem carefully

Before jumping into writing code, take the time to read and understand the problem statement. Make sure you understand the input and output requirements, any constraints or special cases, and the desired algorithmic approach.

A.2 Plan and pseudocode

Once you understand the problem, take some time to plan and sketch out a high-level algorithmic approach. Write pseudocode to help break down the problem into smaller steps and ensure that your solution covers all cases.

A.3 Test your code

After writing your code, test it thoroughly to make sure it produces the correct output for a range of test cases. Consider edge cases, large inputs, and unusual scenarios to make sure your solution is robust.

A.4 Optimize for time and space complexity

When possible, optimize your code for time and space complexity. Consider the Big O notation of your solution and try to reduce it if possible. This can help your code to run faster and more efficiently.

A.5 Write clean, readable code

Make sure your code is easy to read and understand. Use meaningful variable names, indent properly, and comment your code where necessary. This will make it easier for other programmers to read and understand your code, and will help prevent errors and bugs.

A.6 Submit your code and learn from feedback

Once you have a working solution, submit it for review and feedback. Pay attention to any feedback you receive and use it to improve your coding skills and approach for future challenges.

A.7 Keep practicing

The more coding challenges you complete, the better you will become. Keep practicing and challenging yourself to learn new techniques and approaches to problem-solving.

In conclusion, coding challenges are a great way to improve your coding skills and prepare for technical interviews. By following these best practices, you can ensure that you approach coding challenges in a structured and efficient manner, producing clean and readable code that is optimized for time and space complexity.

THANK YOU!

Thank you for taking the time to read my first published book. I would love to hear your thought about this book.

I hope it has been a valuable experience and that you are excited to continue your coding journey. Best of luck with your coding challenges, and remember to have fun along the way!

ABOUT THE AUTHOR

Nhut Nguyen is a seasoned software engineer and career coach with nearly a decade of experience in the tech industry.

He was born and grew up in Ho Chi Minh City, Vietnam. In 2012, he moved to Denmark for a Ph.D. in mathematics at the Technical University of Denmark. After his study, Nhut Nguyen switched to the industry in 2016 and has worked at various tech companies in Copenhagen, Denmark.

Nhut's passion for helping aspiring software engineers succeed led him to write several articles and books where he shares his expertise and practical strategies to help readers navigate their dream job and work efficiently.

With a strong background in mathematics and computer science and a deep understanding of the industry, Nhut is dedicated to empowering individuals to unlock their full potential, land their dream jobs and live happy lives.

Learn more at https://nhutnguyen.com.

INDEX

A

algorithm complexity, 2

B

binomial coefficient, 351
bit masking, 168
bitwise AND, 153
bitwise XOR, 153, 252

C

Coding challenges, 1

D

dictionary order, 185
dummy node, 47

F

Fast and Slow, 289, 295
Fibonacci Number, 220

K

Kadane's algorithm, 270

L

LeetCode, 1

M

memoization, 236
Morse Code, 92

P

Partial sort, 175
permutation, 344
power set, 335

R

readable code, 4

S

sliding window, 78, 104
Sorting, 160
std::accumulate, 109
std::bitset, 162
std::nth_element, 176
std::priority_queue, 131, 136, 141, 147
std::set, 190
std::sort, 217
std::stoi, 109
std::swap, 27