The Problem Solver Guide To Coding

Nhut Nguyen, Ph. D.

Jun 25, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

Welcome to the world of coding challenges, where you'll be tested to solve exciting problems using your programming skills. Whether you're a beginner or an experienced programmer, this book will help you to challenge and enhance your coding skills.

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 test your abilities and help you develop your skills.

Coding challenges can take many forms. They could be programming puzzles, riddles, 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. This book includes coding challenges that cover a wide range of topics, including data structures, algorithms, mathematical problems, and more.

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.

The challenges in this book are picked from leetcode.com. I keep the challenges' titles unchanged so you can easily find it there. I chose the ones that are not too difficult and do not require much effort and determination to solve. Each challenge comes with a detailed solution, explaining the logic behind the solution and how to implement it in C++. You can find most of this book's content on my blog leetsolve.com[1].

1.1 Leetcode

LeetCode.com[2] 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.

[1] https://leetsolve.com
[2] https://leetcode.com

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 users to gain practical experience in problem-solving.

One feature that makes LeetCode stand out is its extensive discussion forum, where users can interact, share their solutions, and learn from one another. This fosters community and collaboration, as users 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 users prepare for technical interviews, sharpen their skills, and advance their 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 Algorithm complexity

Algorithm complexity, also known as time 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 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

2.1 Shift 2D Grid

2.1.1 Problem statement[3]

Given a 2D `grid` of size `m x n` and an integer `k`. You need to shift the grid `k` times.

In one shift operation:

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

Return the 2D grid after applying shift operation `k` times.

Example 1

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

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

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

Example 2

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

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

```
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

- `m == grid.length`.
- `n == grid[i].length`.
- `1 <= m <= 50`.
- `1 <= n <= 50`.
- `-1000 <= grid[i][j] <= 1000`.
- `0 <= k <= 100`.

2.1.2 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

```cpp
#include <vector>
#include <iostream>
using namespace std;
vector<vector<int>> shiftGrid(vector<vector<int>>& grid, int k) {
    vector<int> v;
    for (auto& r : grid) {
        v.insert(v.end(), r.begin(), r.end());
    }
    const int m = grid.size();
    const int n = grid[0].size();
    int p = v.size() - (k % v.size());
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (p == v.size()) {
                p = 0;
            }
            grid[i][j] = v[p++];
        }
    }
    return grid;
}
void printResult(vector<vector<int>>& grid) {
    cout << "[";
    for (auto& r : grid) {
        cout << "[";
        for (int a: r) {
            cout << a << ",";
        }
        cout << "]";
    }
```

(continues on next page)

(continued from previous page)

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

Complexity

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

2.1.3 C++ notes

1. To convert a 2D matrix into a 1D vector, you can use the vector's function `insert()`[4].
2. The modulo operator `%` is usually used to index an array to ensure the index is inbound.

2.2 Find All Numbers Disappeared in an Array

2.2.1 Problem statement[Page 12, 5]

Given an array `nums` of `n` integers where `nums[i]` is in the range `[1, n]`, return an array of all the integers in the range `[1, n]` that do not appear in `nums`.

[4] https://en.cppreference.com/w/cpp/container/vector/insert
[5] https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/

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

Could you do it without extra space and in `O(n)` runtime? You may assume the returned list does not count as extra space.

2.2.2 Solution 1: Marking the appearances

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(vector<int>& nums) {
    vector<bool> exist(nums.size() + 1, false);
    for (auto& i : nums) {
        exist[i] = true;
    }
    vector<int> result;
    for (int i = 1; i <= nums.size(); i++) {
```

(continues on next page)

(continued from previous page)

```
        if (!exist[i]) {
            result.push_back(i);
        }
    }
    return result;
}
void print(vector<int>& nums) {
    cout << "[";
    for (auto& n : nums) {
        cout << n << ",";
    }
    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,]
```

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: much less than `O(n)`. `vector<bool>`[6] is optimized for space efficiency; it stores single bits.

[6] https://en.cppreference.com/w/cpp/container/vector_bool

2.2.3 Solution 2: Follow up

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

One way of marking the appearance of an index `j` is making the element `nums[j]` to be negative. Then the indices `j`'s whose `nums[j]` are unchanged (still positive) are the ones that do not appear in `nums`.

Code

```cpp
#include <vector>
#include <iostream>
using namespace std;
vector<int> findDisappearedNumbers(vector<int>& nums) {
    int j;
    for (int i{0}; i < nums.size(); i++) {
        j = abs(nums[i]);
        nums[j - 1] = -abs(nums[j - 1]);
    }
    vector<int> result;
    for (int i{1}; i <= nums.size(); i++) {
        if (nums[i - 1] > 0) {
            result.push_back(i);
        }
    }
    return result;
}
void print(vector<int>& nums) {
    cout << "[";
    for (auto& n : nums) {
        cout << n << ",";
    }
    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,]
```

Complexity

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

2.2.4 Conclusion

- 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 `vector<bool>` is optimized for space efficiency. Moreover, it is easier to understand than Solution 2.

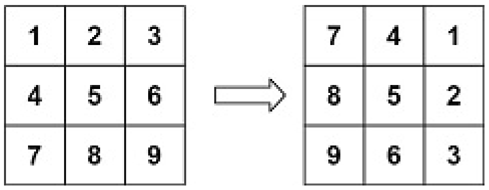
2.3 Rotate Image

2.3.1 Problem statement[Page 16, 7]

You are given an `n x n` 2D `matrix` representing an image. Rotate the image by 90 degrees (clockwise).

You have to rotate the image in-place[8], which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.

Example 1

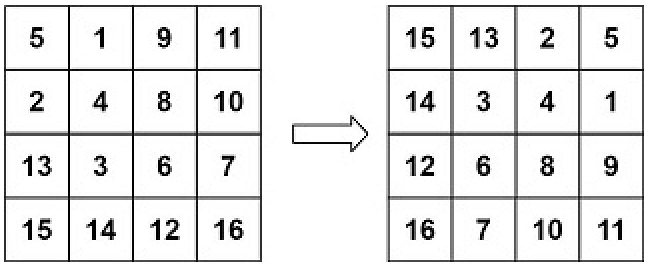


```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[7,4,1],[8,5,2],[9,6,3]]
```

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

[8] https://en.wikipedia.org/wiki/In-place_algorithm

Example 2



```
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`.

2.3.2 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.

2.3.3 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
```

(continues on next page)

(continued from previous page)

```
    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(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);
    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)`.

2.3.4 Implementation notes

1. The function `std::swap`[9] can be used to exchange two values.
2. When doing the transpose or mirroring, you could visit over one-half of the matrix.

2.4 Can Place Flowers

2.4.1 Problem statement[Page 19, 10]

You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in **adjacent** plots.

Given an integer array `flowerbed` containing `0`'s and `1`'s, where `0` means empty and `1` means not empty, and an integer `n`, return `true` if `n` new flowers can be planted in the `flowerbed` without violating the no-adjacent-flowers rule.

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
```

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`.

[9] https://en.cppreference.com/w/cpp/algorithm/swap
[10] https://leetcode.com/problems/can-place-flowers/

2.4.2 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) {
            flowerbed[i] = 1;
            n--;
            i+=2;
        } else {
            i++;
        }
    }
    return n <= 0;
}
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
```

Complexity

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

2.4.3 Implementation note

- 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`[11] and `push_back`[12] of a `std::vector`[13].

2.5 Daily Temperatures

2.5.1 Problem statement[Page 21, 14]

Given an array of integers `temperatures` represents the daily temperatures, return an array `answer` such that `answer[i]` is the number of days you have to wait after the `i-th` day to get a warmer temperature. If there is no future day for which this is possible, keep `answer[i] = 0` instead.

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]
```

[11] https://en.cppreference.com/w/cpp/container/vector/insert
[12] https://en.cppreference.com/w/cpp/container/vector/push_back
[13] https://en.cppreference.com/w/cpp/container/vector
[14] https://leetcode.com/problems/daily-temperatures/

Example 3

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

Constraints

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

2.5.2 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

```cpp
#include <vector>
#include <iostream>
using namespace std;
vector<int> dailyTemperatures(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;
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
    return answer;
}
void print(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:
[1,1,4,2,1,1,0,0,]
[1,1,1,0,]
[1,1,0,]
```

Complexity

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

2.5.3 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

```cpp
#include <vector>
#include <iostream>
using namespace std;
vector<int> dailyTemperatures(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]) {
            // there is some temperature 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(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:
[1,1,4,2,1,1,0,0,]
[1,1,1,0,]
[1,1,0,]
```

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

2.5.4 Key takeaway

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

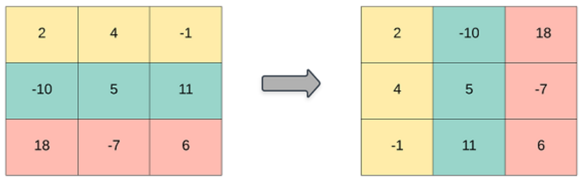
In this particular problem, it can be achieved by doing it in the reversed order.

2.6 Transpose Matrix

2.6.1 Problem statement[15]

Given a 2D integer array `matrix`, return the transpose of `matrix`.

The transpose of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices.



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

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]]
```

Constraints

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

2.6.2 Solution

Code

```
#include <iostream>
#include <vector>
using namespace std;
vector<vector<int>> transpose(vector<vector<int>>& matrix) {
    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(vector<vector<int>>& matrix) {
    cout << "[";
    for (auto row : matrix) {
        cout << "[";
```

(continues on next page)

(continued from previous page)

```
        for (int m : row) {
            cout << m << ",";
        }
        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`, `n = matrix[i].length`.
- Extra space: `O(1)`.

2.6.3 Implementation note

Note that the matrix might not be square, you cannot just swap the elements using for example the function `std::swap`[16].

2.7 Valid Mountain Array

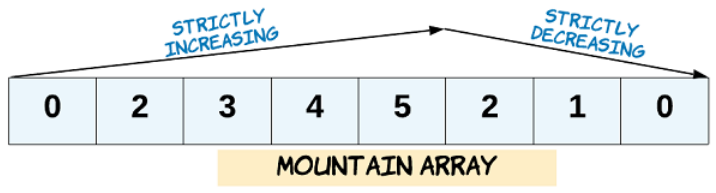
2.7.1 Problem statement[Page 28, 17]

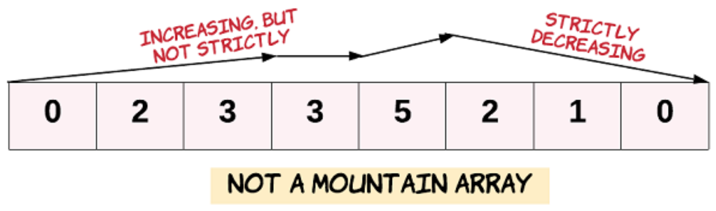
Given an array of integers `arr`, return `true` if and only if it is a valid *mountain array*.

Recall that arr is a **mountain array** if and only if:

[16] https://en.cppreference.com/w/cpp/algorithm/swap
[17] https://leetcode.com/problems/valid-mountain-array/

- `arr.length >= 3`
- There exists some `i` with `0 < i < arr.length - 1` such that:
  - `arr[0] < arr[1] < ... < arr[i - 1] < arr[i]`
  - `arr[i] > arr[i + 1] > ... > arr[arr.length - 1]`





Example 1

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

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`.

2.7.2 Solution

Following the conditions, we have the following implementation.

Code

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

Complexity

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

CHAPTER THREE

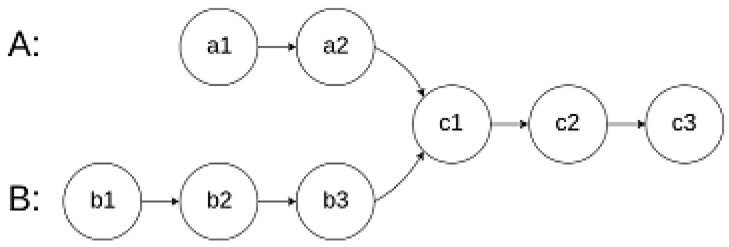
LINKED LIST

3.1 Intersection of Two Linked Lists

3.1.1 Problem statement[Page 33, 18]

Given the heads of two singly linked-lists `headA` and `headB`, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return `null`.

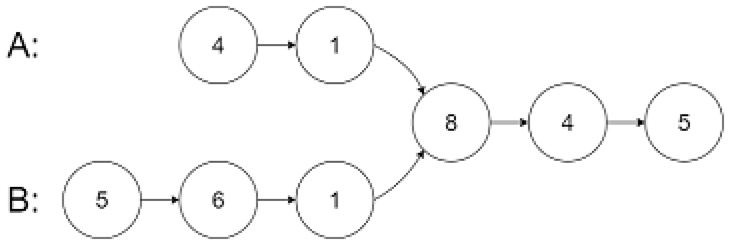
For example, the following two linked lists begin to intersect at node `c1`:



The test cases are generated such that there are no cycles anywhere in the entire linked structure.

Note that the linked lists must retain their original structure after the function returns.

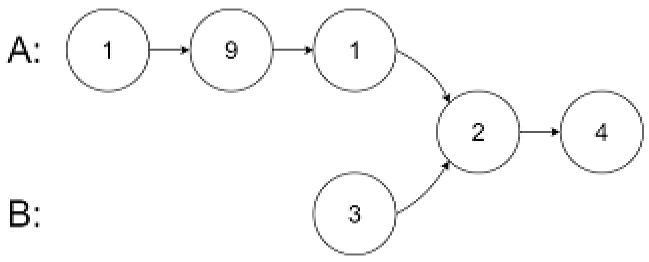
Example 1



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

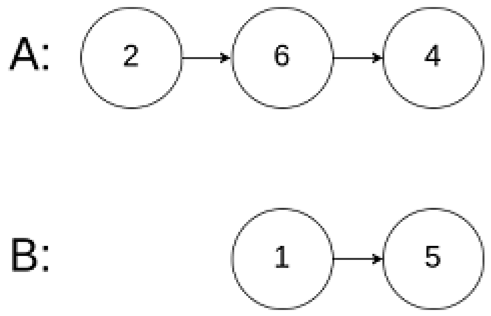
```
Input: listA = [4,1,8,4,5], listB = [5,6,1,8,4,5].
Output: Intersected at '8'
```

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?

3.1.2 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) {}
};

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
```

(continues on next page)

(continued from previous page)

```
        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);
        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);
```

(continues on next page)

(continued from previous page)

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

```
Output:
1
1
1
```

Complexity

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

3.1.3 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;
    }
    if (nodeA != nodeB) {
        return nullptr;
    }
    nodeA = headA;
    nodeB = headB;
    while (lengthA > lengthB) {
        nodeA = nodeA->next;
        lengthA--;
    }
    while (lengthB > lengthA) {
        nodeB = nodeB->next;
        lengthB--;
    }
    while (nodeA != nodeB) {
        nodeA = nodeA->next;
        nodeB = nodeB->next;
    }
    return nodeA;
}
```

(continues on next page)

(continued from previous page)

```
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);
        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);
```

(continues on next page)

(continued from previous page)

```
        two.next = &six;

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

```
Output:
1
1
1
```

Complexity

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

3.1.4 Implementation note

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.

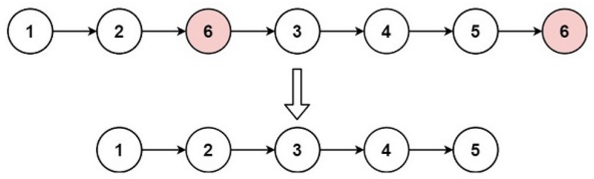
3.2 Remove Linked List Elements

3.2.1 Problem statement[Page 40, 19]

Given the `head` of a linked list and an integer `val`, remove all the nodes of the linked list that has `Node.val == val`, and return *the new head*.

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

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: []
```

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

```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) {
    while (head && head->val == val) {
        head = head->next;
    }
    if (head == nullptr) return nullptr;
    ListNode* pre = head;
    while (pre->next) {
        if (pre->next->val == val) {
            pre->next = pre->next->next;
        } else {
            pre = pre->next;
        }
    }
    return head;
}
void print(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,]
[]
[]
```

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.

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* removeElements(ListNode* head, int val) {
    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(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);
    ListNode two(2, &sixa);
    ListNode head(1, &two);
```

(continues on next page)

(continued from previous page)

```
    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,]
[]
[]
```

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;
        }
```

(continues on next page)

(continued from previous page)

```
    }
    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.

3.3 Merge Two Sorted Lists

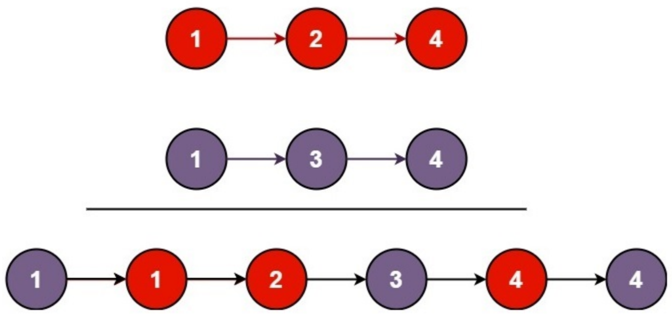
3.3.1 Problem statement[Page 46, 20]

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists in a one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Example 1



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

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

Example 2

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

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.

3.3.2 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
    ListNode* head = list1;
    if (list2->val < head->val) {
        head = list2;
        list2 = list2->next;
    } else {
        list1 = list1->next;
    }
    ListNode* node = head;
    while (list1 && list2) {
        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() {
    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);
```

(continues on next page)

(continued from previous page)

```
    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

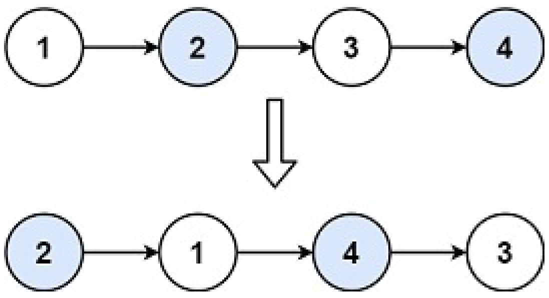
- Runtime: `O(N)`, where `N = list1.length + list2.length`.
- Extra space: `O(1)`.

3.4 Swap Nodes in Pairs

3.4.1 Problem statement[Page 49, 21]

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

Example 1



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

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

Example 2

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

Example 3

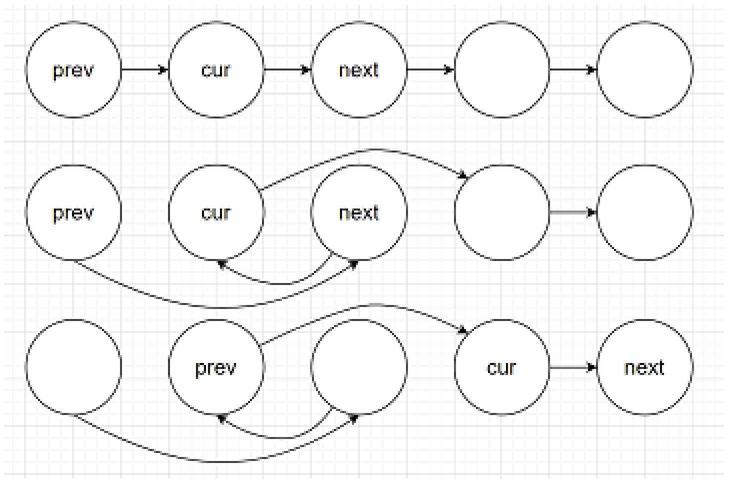
```
Input: head = [1]
Output: [1]
```

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) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
    ListNode* preNode = nullptr;
    ListNode* curNode = head;
    ListNode* nextNode = head->next;
    head = nextNode;
    while (curNode != nullptr && nextNode != nullptr) {
        curNode->next = nextNode->next;
        nextNode->next = curNode;
        if (preNode) {
            preNode->next = nextNode;
        }
        preNode = curNode;
        curNode = curNode->next;
        if (curNode) {
            nextNode = curNode->next;
        }
    }
    return head;
}
void print(ListNode* head) {
    ListNode* node = head;
    std::cout << "[";
    while (node != nullptr) {
        std::cout << node->val << ",";
        node = node->next;
    }
    std::cout << "]" << std::endl;
}
```

(continues on next page)

(continued from previous page)

```
int main() {
    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)`.

3.5 Add Two Numbers

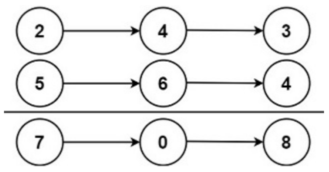
3.5.1 Problem statement[Page 52, 22]

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

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

Example 1



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

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

ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode prehead;     // dummy node to hook the head of the list
    ListNode* node = l1;  // store result on l1
    prehead.next = node;
    int sum = 0;
    while (node) {
        if (l1) {
            sum += l1->val;
            l1 = l1->next;
        }
        if (l2) {
            sum += l2->val;
            l2 = l2->next;
        }
        node->val = sum % 10;
        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;
    }
    return prehead.next;
}
void printResult(ListNode* l) {
    std::cout << "[";
    while (l) {
```

(continues on next page)

(continued from previous page)

```
        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);
        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)`.

CHAPTER

FOUR

HASH TABLE

4.1 Roman to Integer

4.1.1 Problem statement[Page 57, 23]

Roman numerals are represented by seven symbols: `I`, `V`, `X`, `L`, `C`, `D`, and `M`.

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

For example, `2` is written as `II` in Roman numeral, just two ones added together. `12` is written as `XII`, which is simply `X + II`. The number `27` is written as `XXVII`, which is `XX + V + II`.

Roman numerals are usually written from largest to smallest from left to right. However, the numeral for four is not `IIII`. Instead, the number four is written as `IV`. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as `IX`. There are six instances where subtraction is used:

- `I` can be placed before `V` (`5`) and `X` (`10`) to make `4` and `9`.
- `X` can be placed before `L` (`50`) and `C` (`100`) to make `40` and `90`.
- `C` can be placed before `D` (`500`) and `M` (`1000`) to make `400` and `900`.

Given a roman numeral, convert it to an integer.

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

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

```
#include <iostream>
#include <unordered_map>
using namespace std;
int romanToInt(string s) {
    unordered_map<char, int> value = {
        {'I', 1},
        {'V', 5},
```

(continues on next page)

(continued from previous page)

```
        {'X', 10},
        {'L', 50},
        {'C', 100},
        {'D', 500},
        {'M', 1000}
    };
    int result = value[s.back()];
    for (int i = s.length() - 2; i >= 0; i--) {
        result = value[s[i]] < value[s[i+1]] ? result - value[s[i]] :␣
↪result + value[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)` (the map `value` is very small).

4.2 Maximum Erasure Value

4.2.1 Problem statement[Page 59, 24]

You are given an array of positive integers `nums` and want to erase a subarray containing unique elements. The score you get by erasing the subarray is equal to the sum of its elements.

Return the maximum score you can get by erasing exactly one subarray.

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

An array b is called to be a subarray of a if it forms a contiguous subsequence of a, that is, if it is equal to a[l],a[l+1],...,a[r] for some (l,r).

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(vector<int>& nums) {
    vector<int> sum(nums.size(), 0);
    sum[0] = nums[0];
    int maxSum = sum[0];
    unordered_map<int, int> position;
    position[nums[0]] = 0;
```

(continues on next page)

(continued from previous page)

```
    int start = -1;
    for (int i = 1; i < nums.size(); i++) {
        sum[i] = sum[i - 1] + nums[i];
        if (position.find(nums[i]) != position.end()) {
            start = max(start, position[nums[i]]);
        }
        position[nums[i]] = i;
        maxSum = (start == -1) ? sum[i] : max(maxSum, sum[i] -␣
↪sum[start]);
    }
    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)`.

CHAPTER

FIVE

STRING

5.1 Compare Version Numbers

5.1.1 Problem statement[Page 63, 25]

Given two version numbers, `version1` and `version2`, compare them.

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

For example `2.5.33` and `0.1` are valid version numbers.

To compare version numbers, 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 an index, then treat the 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 < 1`.

Return the following:

- If `version1 < version2`, return `-1`.
- If `version1 > version2`, return `1`.
- Otherwise, return `0`.

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

Example 1

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

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.

5.1.2 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;

vector<int> toVector(const string& version) {
    vector<int> revisions;
    string revision;
    for (char c : version) {
        if (c != '.') {
            revision += c;
        } else {
            revisions.push_back(stoi(revision));
            revision = "";
        }
    }
    revisions.push_back(stoi(revision));
    return revisions;
}

int compareVersion(string version1, string version2) {
    vector<int> revisions1 = toVector(version1);
    vector<int> revisions2 = toVector(version2);

    int i = 0;
    while (i < revisions1.size() && i < revisions2.size()) {
        if (revisions1[i] < revisions2[i]) {
            return -1;
        } else if (revisions1[i] > revisions2[i]) {
            return 1;
        }
        i++;
    }
```

(continues on next page)

(continued from previous page)

```
    int remain1 = accumulate(revisions1.begin() + i, revisions1.end(), 0);
    int remain2 = accumulate(revisions2.begin() + i, revisions2.end(), 0);
    if (remain1 < remain2) {
        return -1;
    } else if (remain1 > remain2) {
        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)`.

5.1.3 C++ Notes

- `std::stoi(string)`[26] is used to convert a `string` to an `int`. It ignores the leading zeros for you.
- `std::accumulate(firstIter, lastIter, initValue)`[27] is used to compute the sum of a container.

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

5.2 Valid Anagram

5.2.1 Problem statement[Page 67, 28]

Given two strings `s` and `t`, return `true` if `t` is an anagram of `s`, and `false` otherwise.

An **Anagram** is a word or phrase formed by rearranging the letters of a different 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?

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

Code

```
#include <iostream>
#include <algorithm>
using namespace std;
bool isAnagram(string s, string t) {
    if (s.length() != t.length()) {
        return false;
```

(continues on next page)

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

(continued from previous page)

```
    }
    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
```

Complexity

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

5.2.3 Solution 2: Count the appearances of each letter

Code

```
#include <iostream>
using namespace std;
bool isAnagram(string s, string t) {
    if (s.length() != t.length()) {
        return false;
    }
    int alphabet[26];
    for (int i = 0; i < 26; i++) {
        alphabet[i] = 0;
    }
    for (char& c : s) {
        alphabet[c - 'a']++;
    }
    for (char& c : t) {
        alphabet[c - 'a']--;
        if (alphabet[c - 'a'] < 0) {
```

(continues on next page)

(continued from previous page)

```
            return false;
        }
    }
    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(1)`.

5.2.4 Solution 3: If the inputs contain Unicode characters

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

Code

```
#include <iostream>
#include <unordered_map>
using namespace std;
bool isAnagram(string s, string t) {
    if (s.length() != t.length()) {
        return false;
    }
    unordered_map<char, int> alphabet;
    for (char& c : s) {
        alphabet[c]++;
    }
    for (char& c : t) {
        alphabet[c]--;
        if (alphabet[c] < 0) {
```

(continues on next page)

(continued from previous page)

```
            return false;
        }
    }
    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(1)`.

5.3 Longest Substring Without Repeating Characters

5.3.1 Problem statement[Page 70, 29]

Given a string `s`, find the length of the longest substring without repeating characters.

Example 1

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

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

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.
```

Constraints

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

5.3.2 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 consist 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

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;
int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> position;
    int maxLen = 0;
    int start = -1;
    for (int i = 0; i < s.length(); i++) {
        if (position.find(s[i]) != position.end()) {
            start = max(start, position[s[i]]);
        }
        position[s[i]] = i;
        maxLen = max(maxLen, i - start);
    }
    return maxLen;
}
int main() {
    cout << lengthOfLongestSubstring("abcabcbb") << endl;
    cout << lengthOfLongestSubstring("bbbbb") << endl;
    cout << lengthOfLongestSubstring("pwwkew") << endl;
}
```

```
Output:
3
1
3
```

Complexity

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

5.4 Detect Capital

5.4.1 Problem statement[Page 73, 30]

We define the usage of capitals in a word to be right when one of the following cases holds:

- All letters in this word are capitals, like `"USA"`.
- All letters in this word are not capitals, like `"leetcode"`.
- Only the first letter in this word is capital, like `"Google"`.

Given a string `word`, return `true` if the usage of capitals in it is right.

Example 1

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

Example 2

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

Constraints

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

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

5.4.2 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

```
#include <string>
#include <iostream>
using namespace std;
bool isValidCase(const char& c, const bool isLower) {
    if (isLower) {
        return 'a' <= c && c <= 'z';
    }
    return 'A' <= c && c <= 'Z';
}
bool detectCapitalUse(string word) {
    if (word.length() == 1) {
        return true;
    }
    bool isLower = true;
    if (isValidCase(word[0], false) && isValidCase(word[1], false)) {
        isLower = false;
    }
    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)`.

5.5 Baseball Game

5.5.1 Problem statement[Page 75, 31]

You are keeping score for a baseball game with strange rules. The game consists of several rounds, where the scores of past rounds may affect future rounds' scores.

At the beginning of the game, you start with an empty record. You are given a list of strings `ops`, where `ops[i]` is the `i-th` operation you must apply to the record and is one of the following:

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

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.
```

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

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.

5.5.2 Solution

Code

```
#include <vector>
#include <iostream>
#include <string>
#include <numeric>
using namespace std;
int calPoints(vector<string>& ops) {
```

(continues on next page)

(continued from previous page)

```
    vector<int> stk;
    for (string& 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 {
            stk.push_back(stoi(s));
        }
    }
    return accumulate(stk.begin(), stk.end(), 0);
}
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
```

Complexity

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

5.5.3 Implementation notes

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`[32]:

   - `std::vector` has also methods `push_back(value)`[33] and `pop_back()`[34] like the ones in stack.

[32] https://en.cppreference.com/w/cpp/container/vector
[33] https://en.cppreference.com/w/cpp/container/vector/push_back
[34] https://en.cppreference.com/w/cpp/container/vector/pop_back

  - 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)`[35] computes the sum of the vector `stk`.

5.6 Unique Morse Code Words

5.6.1 Problem statement[Page 78, 36]

International Morse Code defines a standard encoding where each letter is mapped to a series of dots and dashes, as follows:

- `'a'` maps to `".-"`,
- `'b'` maps to `"-..."`,
- `'c'` maps to `"-.-."`, and so on.

For convenience, the full table for the `26` letters of the English alphabet is given below:

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

Given an array of strings `words` where each word can be written as a concatenation of the Morse code of each letter.

- For example, `"cab"` can be written as `"-.-..--..."`, which is the concatenation of `"-.-."`, `".-"`, and `"-..."`. We will call such a concatenation the transformation of a word.

Return the number of different transformations among all words we have.

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 "--...--.".
```

35 https://en.cppreference.com/w/cpp/algorithm/accumulate
36 https://leetcode.com/problems/unique-morse-code-words/

Example 2

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

Constraints

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

5.6.2 Solution: Store the transformations in a set

Code

```cpp
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;
int uniqueMorseRepresentations(vector<string>& words) {
    vector<string> morse{".-", "-...", "-.-.", "-..", ".", "..-.", "--.",
                         "....", "..", ".---", "-.-", ".-..", "--", "-.",
                         "---", ".--.", "--.-", ".-.", "...", "-", "..-",
                         "...-", ".--", "-..-", "-.--", "--.."};
    unordered_set<string> transformations;
    for (string& w : words) {
        string s{""};
        for (char& c : w) {
            s += morse[c - 'a'];
        }
        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)`.

5.7 Find and Replace Pattern

5.7.1 Problem statement[Page 80, 37]

Given a list of strings `words` and a string `pattern`, return a list of `words[i]` that match `pattern`. You may return the answer in any order.

A word matches the pattern if there exists a permutation of letters `p` so that after replacing every letter `x` in the pattern with `p(x)`, we get the desired word.

Recall that a permutation of letters is a bijection from letters to letters: every letter maps to another letter, and no two letters map 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.
```

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

Example 2

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

Constraints

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

5.7.2 Solution: Construct the bijection and check the condition

Code

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
vector<string> findAndReplacePattern(vector<string>& words, string␣
↪pattern) {
    vector<string> result;
    // need two maps for the bijection
    unordered_map<char,char> w_to_p, p_to_w;
    int i;
    for (string& 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()) {
```

(continues on next page)

(continued from previous page)

```
                    // w[i] was not mapped to any letter yet
                    // 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 (const string& 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(N*L)`, where `N = words.length` and `L = pattern.length`.
- Extra space: `O(1)` if `N` or `L` is very larger than 26. The maps `w_to_p` and `p_to_w` just map between 26 lowercase English letters.

5.8 Unique Email Addresses

5.8.1 Problem statement[Page 83, 38]

Every valid email consists of a local name and a domain name, separated by the `'@'` sign. Besides lowercase letters, the email may contain one or more `'.'` or `'+'`.

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

If you add periods `'.'` between some characters in the local name part of an email address, mail sent there will be forwarded to the same address without dots in the local name. Note that this rule does not apply to domain names.

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

If you add a plus `'+'` in the local name, everything after the first plus sign will be ignored. This allows certain emails to be filtered. Note that this rule does not apply to domain names.

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 we send one email to each `emails[i]`, return the number of different addresses that actually receive mails.

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.
```

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

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.

5.8.2 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(vector<string>& emails) {
    unordered_set<string> s;
    for (auto e: emails) {
        auto apos = e.find('@');
```

(continues on next page)

(continued from previous page)

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

```
Output:
2
3
2
```

Complexity

- Runtime: `O(N*M^2)`, where `N = emails.length`, `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).

5.8.3 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

```cpp
#include<string>
#include<iostream>
#include<vector>
#include <unordered_set>
using namespace std;
int numUniqueEmails(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`, `M = max(emails[i].length)`.
- Extra space: `O(N*M)`.

5.8.4 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)[39] 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)[40] returns the position of the first `char` which appears in the string `string` starting from `pos`.

5.8.5 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[41] or std::unordered_map[42] 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`.

[39] https://en.cppreference.com/w/cpp/string/basic_string/substr
[40] https://en.cppreference.com/w/cpp/string/basic_string/find
[41] https://en.cppreference.com/w/cpp/container/unordered_set
[42] https://en.cppreference.com/w/cpp/container/unordered_map

CHAPTER

SIX

STACK

6.1 Remove All Adjacent Duplicates in String II

6.1.1 Problem statement[Page 89, 43]

You are given a string s and an integer k, a k duplicate removal consists of choosing k adjacent and the same letters from s and removing them, causing the left and the right side of the deleted substring to concatenate together.

We repeatedly make k duplicate removals on s until we no longer can.

Return the final string after all such duplicate removals have been made. It is guaranteed that the answer is 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"
```

[43] 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.

6.1.2 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) {
    vector<string> stk;
    int i = 0;
    while (i < s.length()) {
        string a;   // to store adjacent equal letters
        // perform the merge
        if (!stk.empty() && s[i] == stk.back().back()) {
            a = move(stk.back());
            stk.pop_back();
        }
        int j = i;
        while (j < s.length() && s[j] == s[i]) {
            a += s[j];
            // remove the k-duplicate
            if (a.length() == k) {
                a = "";
            }
            j++;
        }
        if (!a.empty()) {
            stk.push_back(a);
        }
        i = j;
    }
    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
```

(continues on next page)

(continued from previous page)

```
aa
ps
```

Complexity

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

6.1.3 Implementation notes

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`[44]:

- `std::vector` also has methods `push_back(value)`[45] and `pop_back()`[46] like the ones in a stack.
- On the other hand, it is faster for a vector to perform the string concatenation at the end.

6.2 Valid Parentheses

6.2.1 Problem statement[Page 92, 47]

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

[44] https://en.cppreference.com/w/cpp/container/vector
[45] https://en.cppreference.com/w/cpp/container/vector/push_back
[46] https://en.cppreference.com/w/cpp/container/vector/pop_back
[47] https://leetcode.com/problems/valid-parentheses/

Example 1

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

Example 2

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

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

```
#include <iostream>
#include <stack>
using namespace std;
bool isValid(string s) {
    stack<char> stk;
    for (char c : s) {
        if (c == '(' || c == '[' || c == '{') {
            stk.push(c);
        } else if (stk.empty()) {
            return false;
        } else if  (c == ')' && stk.top() != '(' ||
                    c == ']' && stk.top() != '[' ||
                    c == '}' && stk.top() != '{') {
            return false;
        } else {
            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/2)`.

6.3 Backspace String Compare

6.3.1 Problem statement[Page 95, 48]

Given two strings `s` and `t`, return `true` if they are equal when both are typed into empty text editors. `'#'` means a backspace character.

Note that after backspacing an empty text, the text will continue empty.

Example 1

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

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?

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

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

Code

```
#include <iostream>
#include <vector>
using namespace std;
string cleanString(string &s) {
    vector<char> v;
    for (int i = 0; i < s.length(); i++) {
        if (s[i] != '#') {
            v.push_back(s[i]);
        } else {
            if (!v.empty()) {
                v.pop_back();
            }
        }
    }
    return string(v.begin(), v.end());
}
bool backspaceCompare(string s, 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
```

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`[49] and `pop`[50] of the data structure `stack`[51] to build and clean the strings.

But `vector`[52] has also such methods: `push_back`[53] and `pop_back`[54].

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`[55] is not constant.

[49] https://en.cppreference.com/w/cpp/container/stack/push
[50] https://en.cppreference.com/w/cpp/container/stack/pop
[51] https://en.cppreference.com/w/cpp/container/stack
[52] https://en.cppreference.com/w/cpp/container/vector
[53] https://en.cppreference.com/w/cpp/container/vector/push_back
[54] https://en.cppreference.com/w/cpp/container/vector/pop_back
[55] https://en.cppreference.com/w/cpp/string/basic_string/erase

CHAPTER SEVEN

PRIORITY QUEUE

7.1 Last Stone Weight

7.1.1 Problem statement[Page 99, 56]

You are given an array of integers `stones` where `stones[i]` is the weight of the `i-th` stone.

We are playing a game with the stones. On each turn, we choose the heaviest two stones and smash them together. Suppose the heaviest two stones have weights `x` and `y` with `x <= y`. The result of this smash is:

- If `x == y`, both stones are destroyed, and
- If `x != y`, the stone of weight `x` is destroyed, and the stone of weight `y` has new weight `y - x`.

At the end of the game, there is at most one stone left.

Return the smallest possible weight of the left stone. 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.
```

[56] 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`[57].

Code

```
#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();
        if (y != x) {
            q.push(y - x);
        }
    }
    return q.empty() ? 0 : q.top();
}
int main() {
    vector<int> stones{2,7,4,1,8,1};
    cout << lastStoneWeight(stones) << endl;
    stones = {1};
```

(continues on next page)

[57] https://en.cppreference.com/w/cpp/container/priority_queue

(continued from previous page)

```
    cout << lastStoneWeight(stones) << endl;
}
```

```
Output:
1
1
```

Complexity

- Runtime: worst case `O(NlogN)`, on average `O(N)`, where `N = stones.length`.
- Extra space: `O(N)`.

7.2 Construct Target Array With Multiple Sums

7.2.1 Problem statement[Page 101, 58]

You are given an array `target` of `n` integers. From a starting array `arr` consisting of `n` 1's, you may perform the following procedure:

- Let `x` be the sum of all elements currently in your array `arr`.
- Choose any index `i` such that `0 <= i < n` and set the value `arr[i] = x`.
- You may repeat this procedure as many times as needed.

Return `true` if it is possible to construct the `target` array from `arr`, 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
```

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

Example 2

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

Example 3

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

Constraints

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

7.2.2 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(vector<int>& target) {
    unsigned long sum = accumulate(target.begin(), target.end(),␣
↪(unsigned long) 0);
    auto pmax = max_element(target.begin(), target.end());
    while (*pmax > 1) {
        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) {
            return false;
        }
        if (sum == 0) {
            return false;
        }
        *pmax %= sum;
        if (*pmax == 0) {
            return false;
        }
        sum += *pmax;
        pmax = max_element(target.begin(), target.end());
    }
    return sum == target.size();
}
int main() {
```

(continues on next page)

(continued from previous page)

```
    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(1)`.

7.2.3 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[59].

Code

```
#include <iostream>
#include <numeric>
#include <queue>
#include <vector>
using namespace std;
bool isPossible(vector<int>& target) {
    priority_queue<int> q(target.begin(), target.end());
    unsigned long sum = accumulate(target.begin(), target.end(),␣
↪(unsigned long) 0);
    while (q.top() > 1) {
        sum -= q.top();
```

(continues on next page)

[59] https://en.cppreference.com/w/cpp/container/priority_queue

(continued from previous page)

```
        if (sum == 1) {
            return true;
        }
        if (q.top() <= sum) {
            return false;
        }
        if (sum == 0) {
            return false;
        }
        int pre = q.top() % sum;
        if (pre == 0) {
            return false;
        }
        q.pop();
        q.push(pre);
        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`.

7.3 Kth Smallest Element in a Sorted Matrix

7.3.1 Problem statement[Page 106, 60]

Given an `n x n` matrix where each of the rows and columns is sorted in ascending order, return the `k-th` smallest element in the matrix.

Note that it is the `k-th` smallest element in the sorted order, not the `k-th` distinct element.

You must find a solution with a memory complexity better than `O(n^2)`.

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`.

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

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[61] fun.

7.3.2 Solution 1: Transform the 2-D matrix into an 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(vector<vector<int>>& matrix, int k) {
    vector<int> m;
    for (auto& row : matrix) {
        m.insert(m.end(), row.begin(), row.end());
    }
    sort(m.begin(), m.end());
    return m[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
```

[61] http://www.cse.yorku.ca/~andy/pubs/X+Y.pdf

Complexity

- Runtime: `O(n^2*logn)`, where `n x n` is the size of the matrix. Note that `log(n^2) = 2logn`.
- Extra space: `O(n^2)`.

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`[62] can be used for this purpose.

Code

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
int kthSmallest(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]);
            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;
    matrix = {{-5}};
    cout << kthSmallest(matrix, 1) << endl;
}
```

[62] https://en.cppreference.com/w/cpp/container/priority_queue

```
Output:
13
-5
```

Complexity

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

7.3.4 Solution 3: Binary search

Since the matrix is somehow sorted, you can perform the binary search algorithm on it.

But the criteria for the searching is not the value of the element `x` of interest; it is the number of elements that less than or equal to `x` must be exactly `k`. You can use `std::upper_bound`[63] for this purpose.

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int countLessOrEqual(const vector<vector<int>>& matrix, int x) {
    int count = 0;
    for (const auto& row : matrix) {
        count += upper_bound(row.begin(), row.end(), x) - row.begin();
    }
    return count;
}
int kthSmallest(vector<vector<int>>& matrix, int k) {
    int left = matrix.front().front();
    int right = matrix.back().back();
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (countLessOrEqual(matrix, mid) >= k) {
            right = mid - 1;
        } else {
            left = mid + 1;
```

(continues on next page)

[63] https://en.cppreference.com/w/cpp/algorithm/upper_bound

(continued from previous page)

```
        }
    }
    return left;
}
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
```

Complexity

- Runtime: `O(nlognlogM)`, where `n x n` is the size of the matrix, `M` is the difference between the maximum element and the minimum element of the matrix.
- Extra space: `O(1)`.

7.4 Kth Largest Element in a Stream

7.4.1 Problem statement[Page 110, 64]

Design a class to find the `k-th` largest element in a stream. Note that it is the `k-th` largest element in the sorted order, not the `k-th` distinct element.

Implement `KthLargest` class:

- `KthLargest(int k, int[] nums)` Initializes the object with the integer `k` and the stream of integers `nums`.
- `int add(int val)` Appends the integer `val` to the stream and returns the element representing the `k-th` largest element in the stream.

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

7.4.2 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(_nums.begin(), _nums.end(), std::greater());
    }

    int add(int val) {
        auto it = _nums.begin();
        while (it != _nums.end() &&  val < *it) {
            it++;
        }
        _nums.insert(it, val);
        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
```

Complexity

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

7.4.3 Solution 2: Priority queue

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

It is `std::priority_queue`[65], 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` 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 queue.

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) :
        _q(nums.begin(), nums.end()), _k(k) {
    }

    int add(int val) {
        _q.push(val);
        while (_q.size() > _k) {
            _q.pop();
        }
        return _q.top();
    }
};
int main() {
```

(continues on next page)

[65] https://en.cppreference.com/w/cpp/container/priority_queue

(continued from previous page)

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

CHAPTER

EIGHT

BINARY TREE

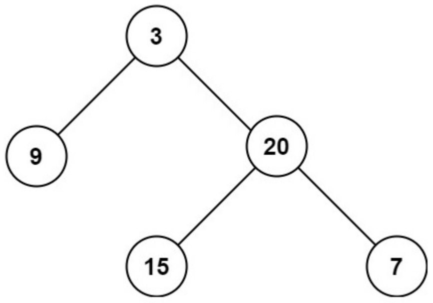
8.1 Maximum Depth of Binary Tree

8.1.1 Problem statement[?]

Given the `root` of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1



```
Input: root = [3,9,20,null,null,15,7]
Output: 3
```

[66] https://leetcode.com/problems/maximum-depth-of-binary-tree/

Example 2

```
Input: root = [1,null,2]
Output: 2
```

Constraints

- The number of nodes in the tree is in the range `[0, 10^4]`.
- `-100 <= Node.val <= 100`.

8.1.2 Solution

You have the following recursive relationship between the `root` and its children.

```
maxDepth(root) = max(maxDepth(root->left), maxDepth(root->right))
```

Code

```
#include <iostream>
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
↪ right(right) {}
};

int maxDepth(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    return 1 + std::max(maxDepth(root->left), maxDepth(root->right));
}

int main() {
    TreeNode fifteen(15);
    TreeNode seven(7);
    TreeNode twenty(20, &fifteen, &seven);
```

(continues on next page)

(continued from previous page)

```
    TreeNode nine(9);
    TreeNode three(3, &nine, &twenty);
    std::cout << maxDepth(&three) << std::endl;
    TreeNode two(2);
    TreeNode one(1, nullptr, &two);
    std::cout << maxDepth(&one) << std::endl;
}
```

```
Output:
3
2
```

Complexity

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

8.2 Lowest Common Ancestor of a Binary Search Tree

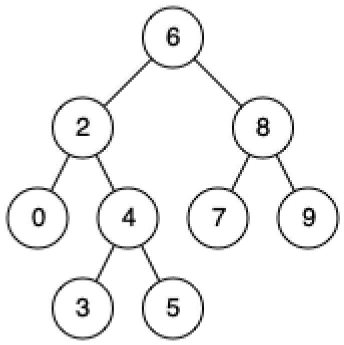
8.2.1 Problem statement[?]

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

According to the definition of LCA on Wikipedia[68]: "The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow a node to be a descendant of itself)."

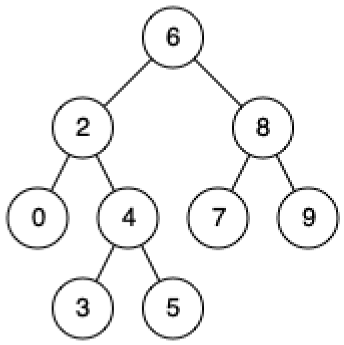
[67] https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/
[68] https://en.wikipedia.org/wiki/Lowest_common_ancestor

Example 1



```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
```

Example 2



```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a␣
↪descendant of itself according to the LCA definition.
```

Example 3

```
Input: root = [2,1], p = 2, q = 1
Output: 2
```

Constraints

- The number of nodes in the tree is in the range `[2, 10^5]`.
- `-10^9 <= Node.val <= 10^9`.
- All `Node.val` are unique.
- `p != q`.
- `p` and `q` will exist in the BST.

8.2.2 Solution: Recursion

Note that in a BST, the values of a `node` and its children `left` and `right` satisfy

```
left.value < node.value < right.value.
```

It lets you know which branch (left or right) of the `root` the nodes `p` and `q` belong to.

Code

```
#include <iostream>
using namespace std;
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (p->val < root->val && q->val < root->val) {
        return lowestCommonAncestor(root->left, p, q);
    } else if (root->val < p->val && root->val < q->val) {
        return lowestCommonAncestor(root->right, p, q);
    }
    return root;
}
```

(continues on next page)

(continued from previous page)

```

int main() {
    TreeNode zero(0);
    TreeNode three(3);
    TreeNode five(5);
    TreeNode four(4);
    four.left = &three;
    four.right = &five;
    TreeNode two(2);
    two.left = &zero;
    two.right = &four;
    TreeNode seven(7);
    TreeNode nine(9);
    TreeNode eight(8);
    eight.left = &seven;
    eight.right = &nine;
    TreeNode six(6);
    six.left = &two;
    six.right = &eight;

    cout << lowestCommonAncestor(&six, &two, &eight)->val << endl;
    cout << lowestCommonAncestor(&six, &two, &four)->val << endl;
    cout << lowestCommonAncestor(&two, &two, &zero)->val << endl;
}
```

```
Output:
6
2
2
```

Complexity

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

CHAPTER

NINE

GRAPH

9.1 Is Graph Bipartite?

9.1.1 Problem statement[?]

There is an undirected graph with `n` nodes, where each node is numbered between `0` and `n - 1`. You are given a 2D array `graph`, where `graph[u]` is an array of nodes that node `u` is adjacent to. More formally, for each `v` in `graph[u]`, there is an undirected edge between node `u` and node `v`. The graph has the following properties:

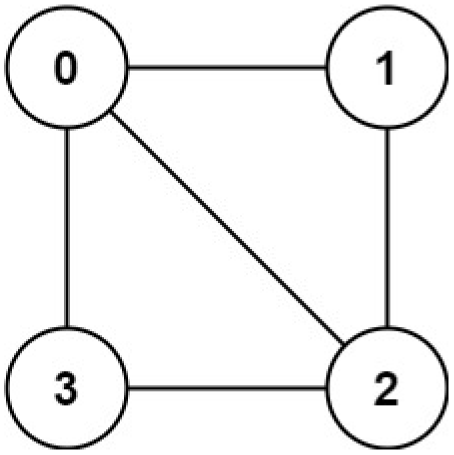
- There are no self-edges (`graph[u]` does not contain `u`).
- There are no parallel edges (`graph[u]` does not contain duplicate values).
- If `v` is in `graph[u]`, then `u` is in `graph[v]` (the graph is undirected).
- The graph may not be connected, meaning there may be two nodes `u` and `v` such that there is no path between them.

A graph is **bipartite** if the nodes can be partitioned into two independent sets `A` and `B` such that every edge in the graph connects a node in set `A` and a node in set `B`.

Return `true` if and only if it is bipartite.

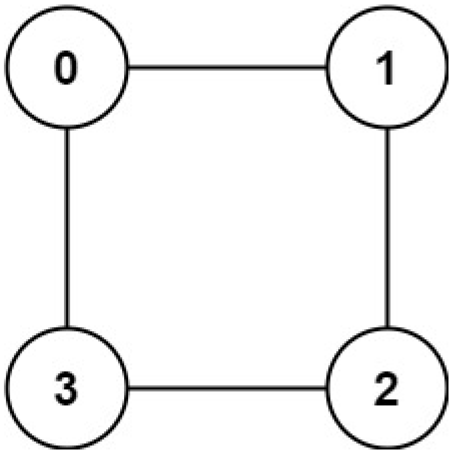
[69] https://leetcode.com/problems/is-graph-bipartite/

Example 1



```
Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]
Output: false
Explanation: There is no way to partition the nodes into two independent␣
↪sets such that every edge connects a node in one and a node in the␣
↪other.
```

Example 2



```
Input: graph = [[1,3],[0,2],[1,3],[0,2]]
Output: true
Explanation: We can partition the nodes into two sets: {0, 2} and {1, 3}.
```

Constraints

- `graph.length == n`.
- `1 <= n <= 100`.
- `0 <= graph[u].length < n`.
- `0 <= graph[u][i] <= n - 1`.
- `graph[u]` does not contain `u`.
- All the values of `graph[u]` are unique.
- If `graph[u]` contains v, then `graph[v]` contains `u`.

9.1.2 Solution: Coloring the nodes by Depth First Search

You could color the nodes in set A with one color and those in B with another color. Then two ends of every edge have different colors.

Now you can use the DFS algorithm[70] to perform the coloring on each connected component of the graph.

During the traversal, if there is an edge having the same color at two ends then return `false`.

Code

```cpp
#include <vector>
#include <iostream>
using namespace std;
bool isBipartite(vector<vector<int>>& graph) {
    vector<int> color(graph.size(), 0);
    for (int i = 0; i < graph.size(); i++) {
        if (color[i] != 0) continue;
        vector<int> s;
        s.push_back(i);
        color[i] = 1;
        while (!s.empty()) {
            int u = s.back();
            s.pop_back();
            for (int v : graph[u]) {
                if (color[v] == 0) {
                    color[v] = -color[u];
```

(continues on next page)

[70] https://en.wikipedia.org/wiki/Depth-first_search

(continued from previous page)

```
                    s.push_back(v);
                } else if (color[v] == color[u]) {
                    return false;
                }
            }
        }

    }
    return true;
}
int main() {
    vector<vector<int>> graph{{1,2,3},{0,2},{0,1,3},{0,2}};
    cout << isBipartite(graph) << endl;
    graph = {{1,3},{0,2},{1,3},{0,2}};
    cout << isBipartite(graph) << endl;
}
```

```
Output:
0
1
```

Complexity

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

9.1.3 Implementation note

- This is the non-recursive implementation of DFS algorithm where you could use the stack data structure to avoid the recursion.
- The stack's methods needed in the DFS algorithm are only **push** and **pop**. There are similar ones in `std::vector`[71], which are `push_back`[72] and `pop_back`[73] which you could use well.

[71] https://en.cppreference.com/w/cpp/container/vector
[72] https://en.cppreference.com/w/cpp/container/vector/push_back
[73] https://en.cppreference.com/w/cpp/container/vector/pop_back

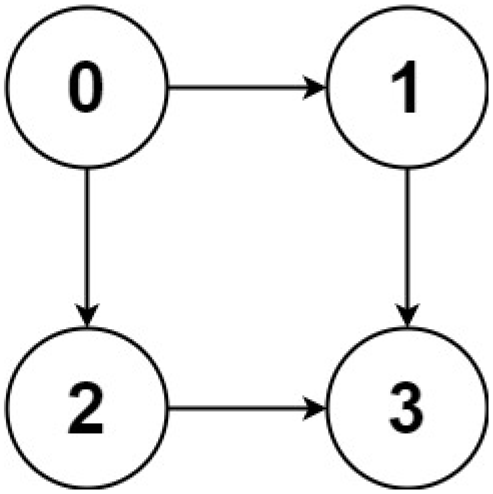
9.2 All Paths From Source to Target

9.2.1 Problem statement[74]

Given a *directed acyclic graph* (DAG) of `n` nodes labeled from `0` to `n - 1`, find all possible paths from node `0` to node `n - 1` and return them in any order.

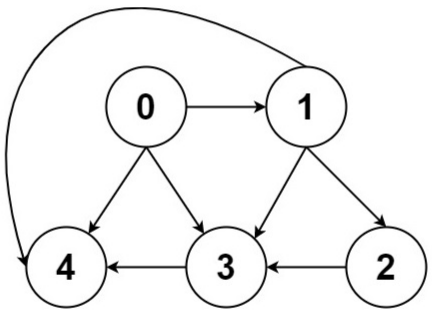
The graph is given as follows: `graph[i]` is a list of all nodes you can visit from node `i` (i.e., there is a directed edge from node `i` to node `graph[i][j]`).

Example 1



```
Input: graph = [[1,2],[3],[3],[]]
Output: [[0,1,3],[0,2,3]]
Explanation: There are two paths: `0 -> 1 -> 3` and `0 -> 2 -> 3`.
```

Example 2



[74] https://leetcode.com/problems/all-paths-from-source-to-target/

```
Input: graph = [[4,3,1],[3,2,4],[3],[4],[]]
Output: [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]
```

Example 3

```
Input: graph = [[1],[]]
Output: [[0,1]]
```

Example 4

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

Example 5

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

Constraints

- `n == graph.length`.
- `2 <= n <= 15`.
- `0 <= graph[i][j] < n`.
- `graph[i][j] != i` (i.e., there will be no self-loops).
- All the elements of `graph[i]` are unique.
- The input graph is guaranteed to be a DAG.

9.2.2 Solution: Depth-first search (DFS)

This problem is exactly the Depth-first search algorithm[75].

[75] https://en.wikipedia.org/wiki/Depth-first_search

Code

```cpp
#include <vector>
#include <iostream>
using namespace std;
void DFS(vector<vector<int>>& graph, vector<vector<int>>& paths, vector
↪<int>& path) {
    for (auto& node : graph[path.back()]) {
        path.push_back(node);
        if (node == graph.size() - 1) {
            paths.push_back(path);
            path.pop_back();
        } else {
            DFS(graph, paths, path);
        }
    }
    path.pop_back();
}
vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
    vector<vector<int>> paths;
    vector<int> path = {0};
    DFS(graph, paths, path);
    return paths;
}
void printPaths(vector<vector<int>>& paths) {
    cout << "[";
    for (auto& p : paths) {
        cout << "[";
        for (auto& node : p) {
            cout << node << ",";
        }
        cout << "],";
    }
    cout << "]\n";
}
int main() {
    vector<vector<int>> graph = {{1,2},{3},{3},{}};
    auto paths = allPathsSourceTarget(graph);
    printPaths(paths);
    graph = {{4,3,1},{3,2,4},{3},{4},{}};
    paths = allPathsSourceTarget(graph);
    printPaths(paths);
}
```

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

Complexity

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

CHAPTER

TEN

SORTING

10.1 Remove Covered Intervals

10.1.1 Problem statement[76]

Given an array `intervals` where `intervals[i] = [li, ri]` represent the interval `[li, ri)`, remove all intervals that are covered by another interval in the list.

The interval `[a, b)` is covered by the interval `[c, d)` if and only if `c <= a` and `b <= d`.

Return the number of remaining 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
```

[76] https://leetcode.com/problems/remove-covered-intervals/

Constraints

- `1 <= intervals.length <= 1000`.
- `intervals[i].length == 2`.
- `0 <= li <= ri <= 10^5`.
- All the given intervals are unique.

10.1.2 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

```cpp
#include <vector>
#include <iostream>
using namespace std;
inline bool isCovered(vector<int>& i, 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])) {
                intervals.erase(intervals.begin() + i);
                erase_i = true;
                break;
```

(continues on next page)

(continued from previous page)

```
            } else if (isCovered(intervals[j], intervals[i])) {
                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}};
    cout << removeCoveredIntervals(intervals) << endl;
    intervals = {{1,4},{2,3}};
    cout << removeCoveredIntervals(intervals) << endl;
}
```

```
Output:
2
1
```

Complexity

- Runtime: `O(N^3)`, where `N = intervals.length`.
- Extra space: `O(1)`.

10.1.3 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(intervals.begin(), intervals.end());
    int count = 0;
    int maxRight = -1;
    int preLeft = -1;
    for (auto& i : intervals) {
        if (i[1] <= maxRight) { // i is covered by some previous interval
            count++;
        } else if (i[0] == preLeft) { // i covers its exact previous one
            count++;
        } else {
            preLeft = i[0];
        }
        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(NlogN)`, where `N = intervals.length`.
- Extra space: `O(1)`.

10.1.4 Key takeaway

- Two `std::vectors` can be compared using dictionary order.

10.2 Minimum Deletions to Make Character Frequencies Unique

10.2.1 Problem statement[77]

A string `s` is called good if there are no two different characters in `s` that have the same frequency.

Given a string `s`, return the minimum number of characters you need to delete to make `s` good.

The frequency of a character in a string is the number of times it appears in the string. For example, in the string `"aab"`, the frequency of `'a'` is `2`, while the frequency of `'b'` is `1`.

Example 1

```
Input: s = "aab"
Output: 0
Explanation: s is already good.
```

[77] https://leetcode.com/problems/minimum-deletions-to-make-character-frequencies-unique/

Example 2

```
Input: s = "aaabbbcc"
Output: 2
Explanation: You can delete two 'b's resulting in the good string "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 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 to 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) {
    vector<int> freq(26, 0);
    for (char& c: s) {
        freq[c - 'a']++;
    }
    sort(freq.begin(), freq.end(), greater<int>());
    int deletion = 0;
    int currentFreq = freq[0];
    for (int i = 1; i < freq.size() && freq[i] > 0; i++) {
        if (currentFreq == 0) {
            deletion += freq[i];
        } else if (freq[i] >= currentFreq) {
            deletion += freq[i] - currentFreq + 1;
            currentFreq--;
        } else {
            currentFreq = freq[i];
        }
    }
    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)` since `26` is not a big number.

10.3 Majority Element

10.3.1 Problem statement[78]

Given an array `nums` of size `n`, return the majority element.

The majority element is the element that appears more than `n / 2` times. You may assume that the majority element always exists in the 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`.

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

Follow-up:

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

10.3.2 Solution 1: Counting the frequency

Code

```
#include <vector>
#include <iostream>
#include <unordered_map>
using namespace std;
int majorityElement(vector<int>& nums) {
    unordered_map<int,int> freq;
    const int HALF = nums.size() / 2;
    for (int a : nums) {
        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
```

Complexity

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

10.3.3 Solution 2: Sorting and picking the middle element

Code

```cpp
#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
```

10.3.4 Solution 3: Partial sort

Since you are interested in only the middle element after sorting, the partial sorting algorithm `std::nth_element`[79] can be used in this case to reduce the cost of the full sorting.

Code

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int majorityElement(vector<int>& nums) {
    const int mid = nums.size() / 2;
    std::nth_element(nums.begin(), nums.begin() + mid, nums.end());
    return nums[mid];
}
```

(continues on next page)

[79] https://en.cppreference.com/w/cpp/algorithm/nth_element

(continued from previous page)

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

```
Output:
3
2
```

Complexity

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

10.3.5 Modern C++ notes

In the code of Solution 3, the partial sorting algorithm `std::nth_element`[80] 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.

10.4 Find the Duplicate Number

10.4.1 Problem statement[?]

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only one repeated number in `nums`, return this repeated number.

You must solve the problem without modifying the array `nums` and uses only constant extra space.

[80] https://en.cppreference.com/w/cpp/algorithm/nth_element
[81] https://leetcode.com/problems/find-the-duplicate-number/

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.

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?

10.4.2 Solution 1: Sorting

```
#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() {
```

(continues on next page)

(continued from previous page)

```
    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
```

Complexity

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

10.4.3 Follow up

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

Due to Pigeonhole principle[82]:

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.

10.4.4 Solution 2: Marking the visited numbers

Code

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

(continues on next page)

[82] https://en.wikipedia.org/wiki/Pigeonhole_principle

(continued from previous page)

```
    vector<bool> visited(nums.size());
    for (int a : nums) {
        if (visited[a]) {
            return a;
        }
        visited[a] = true;
    }
    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
```

Complexity

- Runtime: `O(N)`, where `N = nums.length`.
- Extra space: `O(logN)`. `std::vector<bool>`[83] is optimized for space-efficient.

10.4.5 Solution 3: Marking with `std::bitset`

Since `n <= 10^5`, you can use this size for a `std::bitset`[84] to do the marking.

Code

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

(continues on next page)

[83] https://en.cppreference.com/w/cpp/container/vector_bool
[84] https://en.cppreference.com/w/cpp/utility/bitset

(continued from previous page)

```
    bitset<100001> visited;
    for (int a : nums) {
        if (visited[a]) {
            return a;
        }
        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
```

Complexity

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

10.5 My Calendar I

10.5.1 Problem statement[85]

You are implementing a program to use as your calendar. We can add a new event if adding the event will not cause a double booking.

A double booking happens when two events have some non-empty intersection (i.e., some moment is common to both events.).

The event can be represented as a pair of integers `start` and `end` that represents a booking on the half-open interval `[start, end)`, the range of real numbers `x` such that `start <= x < end`.

Implement the `MyCalendar` class:

[85] https://leetcode.com/problems/my-calendar-i/

- `MyCalendar()` Initializes the calendar object.
- `boolean book(int start, int end)` Returns `true` if the event can be added to the calendar successfully without causing a double booking. Otherwise, return `false` and do not add the event 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.
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.

10.5.2 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;
```

(continues on next page)

(continued from previous page)

```
public:
    MyCalendar() {}
    bool book(int start, int end) {
        for (auto& e : _events) {
            if (!(e.second <= start || end <= e.first)) {
                return false;
            }
        }
        _events.push_back({start, end});
        return true;
    }
};
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

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

10.5.3 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`[86] to store the sorted unique events.

[86] https://en.cppreference.com/w/cpp/container/set

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:
    set<Event, EventCmp> _events;
public:
    MyCalendar() {}
    bool book(int start, int end) {
        auto result = _events.insert({start, end});
        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

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

10.6 Remove Duplicates from Sorted Array II

10.6.1 Problem statement[?]

Given an integer array `nums` sorted in **non-decreasing order**, remove some duplicates in-place[88] such that each unique element appears **at most twice**. The **relative order** of the elements should be kept the **same**.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array `nums`. More formally, if there are `k` elements after removing the duplicates, then the first `k` elements of `nums` should hold the final result. It does not matter what you leave beyond the first `k` elements.

Return `k` after placing the final result in the first `k` slots of `nums`.

Do **not** allocate extra space for another array. You must do this by **modifying the input array in-place** with `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.
It does not matter what you leave beyond the returned k (hence they are␣
↪underscores).
```

[87] https://leetcode.com/problems/remove-duplicates-from-sorted-array-ii/
[88] https://en.wikipedia.org/wiki/In-place_algorithm

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.
It does not matter what you leave 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.

10.6.2 Solution 1: Erasing the duplicates

In order for each unique element appears **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

```cpp
#include <vector>
#include <iostream>
using namespace std;
int removeDuplicates(vector<int>& nums) {
    int i = 2;
    while (i < nums.size()) {
        if (nums[i] == nums[i-2]) {
            int j = i;
            while (j < nums.size() && nums[j] == nums[i]) {
                j++;
            }
            nums.erase(nums.begin() + i, nums.begin() + j);
        } else {
            i++;
        }
```

(continues on next page)

(continued from previous page)

```
    }
    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";
}
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,]
```

Complexity

- Runtime:
  - Worst case `O(N*N/3)`, where `N = nums.size()`. The complexity of the `erase()`[89] 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)`.

[89] https://en.cppreference.com/w/cpp/container/vector/erase

10.6.3 Solution 2: Reassigning the satisfying elements

You might need to avoid the `erase()` method in the solution above to reduce the complexity. Moreover, the problem only cares about the first `k` elements of the array `nums` after removing the duplicates.

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 satisified 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]) {
            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] << ",";
    }
    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)`.

10.7 Merge Sorted Array

10.7.1 Problem statement[?]

You are given two integer arrays `nums1` and `nums2`, sorted in non-decreasing order, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in non-decreasing order.

The final sorted array should not be returned by the function, but instead be stored inside the array `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

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.
```

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

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?

10.7.2 Solution 1: Store the result in a new container

Code

```
#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;
    int j = 0;
    while (i < m || j < n) {
```

(continues on next page)

(continued from previous page)

```
        if (j == n) {
            result.push_back(nums1[i++]);
        } else if (i == m) {
            result.push_back(nums2[j++]);
        } else {
            result.push_back(nums1[i] < nums2[j] ? nums1[i++] :␣
↪nums2[j++]);
        }
    }
    nums1.swap(result);
}
void printResult(vector<int>& nums1) {
    cout << "[";
    for (int 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 + n = nums1.length, n = nums2.length`.
- Extra space: `O(m + n)`.

10.7.3 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) {
            nums1[k--] = nums1[i--];
        } else if (i < 0) {
            nums1[k--] = nums2[j--];
        } else {
            nums1[k--] = nums1[i] > nums2[j] ? nums1[i--] : nums2[j--];
        }
    }
}
void printResult(vector<int>& nums1) {
    cout << "[";
    for (int 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);
```

(continues on next page)

(continued from previous page)

```
    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 + n = nums1.length, n = nums2.length`.
- Extra space: `O(1)`.

CHAPTER ELEVEN

BINARY SEARCH

11.1 Single Element in a Sorted Array

11.1.1 Problem statement[91]

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return the single element that appears only once.

Your solution must run in `O(log n)` time and `O(1)` space.

Example 1

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

Example 2

```
Input: nums = [3,3,7,7,10,11,11]
Output: 10
```

[91] https://leetcode.com/problems/single-element-in-a-sorted-array/

Constraints

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

11.1.2 Solution 1: Bruteforce

Code

```
#include <vector>
#include <iostream>
using namespace std;
int singleNonDuplicate(vector<int>& nums) {
    for (int i = 0; i < nums.size() - 1; i += 2) {
        if (nums[i] != nums[i + 1]) {
            return nums[i];
        }
    }
    return nums[0];
}
int  main() {
    vector<int> nums{1,1,2,3,3,4,4,8,8};
    cout << singleNonDuplicate(nums) << endl;
    nums = {3,3,7,7,10,11,11};
    cout << singleNonDuplicate(nums) << endl;
    nums = {3};
    cout << singleNonDuplicate(nums) << endl;
}
```

```
Output:
2
10
3
```

Complexity

- Runtime `O(n/2)`, where `n = nums.length`.
- Memory `O(1)`.

11.1.3 Solution 2: Binary search

Since `nums` is sorted, you can perform a binary search on it.

Let us divide `nums` into two halves.

If the single element belongs to the right half, all elements of the left half satisfy `nums[2*i] == nums[2*i + 1]`.

Conversely, if the single element belongs to the left half, that condition is violated at the middle element of `nums` (the middle one with an even index).

Example 1

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

- The middle element with even index is `nums[4] = 3`. It is not equal to `nums[4 + 1] = 4`. So the single element must be somewhere in the left half `[1,1,2,3,3]`.
- The middle element of `nums = [1,1,2,3,3]` with even index is `nums[2] = 2`, which is not equal to `nums[2 + 1] = 3`. So the single element must be somewhere in the left half `[1,1,2]`.
- The middle element of `nums = [1,1,2]` with even index is `nums[0] = 1 == nums[0 + 1]`. So the single element must be somewhere in the right half `[2]`.
- `nums = [2]` contains only one element. So `2` is the result.

Code

```cpp
#include <vector>
#include <iostream>
using namespace std;
int singleNonDuplicate(vector<int>& nums) {
    int left = 0;
    int right = nums.size() - 1;
    while (left < right) {
        int mid = (right + left)/4 * 2; // to make sure mid is even
        if (nums[mid] != nums[mid + 1]) {
```

(continues on next page)

(continued from previous page)

```
            right = mid;
        } else {
            left = mid + 2;
        }
    }
    return nums[right];
}
int  main() {
    vector<int> nums{1,1,2,3,3,4,4,8,8};
    cout << singleNonDuplicate(nums) << endl;
    nums = {3,3,7,7,10,11,11};
    cout << singleNonDuplicate(nums) << endl;
    nums = {3};
    cout << singleNonDuplicate(nums) << endl;
}
```

```
Output:
2
10
3
```

Complexity

- Runtime `O(logn)`, where `n = nums.length`.
- Memory `O(1)`.

CHAPTER TWELVE

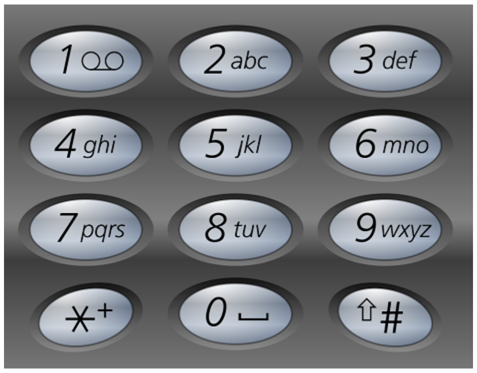
RECURSIVE

12.1 Letter Combinations of a Phone Number

12.1.1 Problem statement[92]

Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent. Return the answer in any order.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that `1` does not map to any letters.



Example 1

```
Input: digits = "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

[92] https://leetcode.com/problems/letter-combinations-of-a-phone-number/

Example 2

```
Input: digits = ""
Output: []
```

Example 3

```
Input: digits = "2"
Output: ["a","b","c"]
```

Constraints

- `0 <= digits.length <= 4`.
- `digits[i]` is a digit in the range `['2', '9']`.

12.1.2 Solution: Recursive

If you know the combinations `result` of a string `digits`, what is the result of extending it one more digit?

Answer: The new result is constructed by adding each letter of the mapping of the new digit to each string of the `result`.

Example 1 and 3

Assume you have computed the result of `digits = "2"`, which is `["a","b","c"]`.

To compute the result of `digits = "23"`, you add each letter of the mapping `'3' -> {'d', 'e', 'f'}` to each string `"a", "b", "c"`.

You get the new result `["ad","ae","af","bd","be","bf","cd","ce","cf"]`.

Code

```cpp
#include <vector>
#include <iostream>
#include <unordered_map>
using namespace std;
const unordered_map<char, vector<char> > m{{'2', {'a', 'b', 'c'}},
```

(continues on next page)

(continued from previous page)

```
                                                    {'3', {'d', 'e', 'f'}},
                                                    {'4', {'g', 'h', 'i'}},
                                                    {'5', {'j', 'k', 'l'}},
                                                    {'6', {'m', 'n', 'o'}},
                                                    {'7', {'p', 'q', 'r', 's'}},
                                                    {'8', {'t', 'u', 'v'}},
                                                    {'9', {'w', 'x', 'y', 'z'}}};

void combination(const string_view& digits, int i, vector<string>&␣
↪result) {
    if (i >= digits.length()) {
        return;
    }
    if (result.empty()) {
        result = {""};
    }
    vector<string> newResult;
    for (string& s : result) {
        for (auto& c : m.at(digits[i])) {
            newResult.push_back(s + c);
        }
    }
    result.swap(newResult);
    combination(digits, i + 1, result);
}
vector<string> letterCombinations(string digits) {
    vector<string> result;
    combination(digits, 0, result);
    return result;
}
void printResult(vector<string>& result) {
    cout << "[";
    for (string& s : result) {
        cout << s << ",";
    }
    cout << "]\n";
}
int main() {
    vector<string> result = letterCombinations("23");
    printResult(result);
    result = letterCombinations("");
    printResult(result);
    result = letterCombinations("2");
```

(continues on next page)

(continued from previous page)

```
    printResult(result);
}
```

```
Output:
[ad,ae,af,bd,be,bf,cd,ce,cf,]
[]
[a,b,c,]
```

Complexity

- Runtime: `O(3^N)`, where `N = digits.length`. In this problem, `N` is very small (`N <= 4`).
- Extra space: `O(1)` (the small map).

12.1.3 Implementation notes

You can use the assignment operator `'='` for `result.swap(newResult)`, i.e. `result = newResult`.

But this assignment allocates additional memory for a copy of `newResult` before assigning it to `result`.

The `std::swap()` algorithm[93] avoids such copying by using `std::move()`[94]. It exchanges the contents of each other without allocating additional memory.

[93] https://en.cppreference.com/w/cpp/string/basic_string/swap
[94] https://en.cppreference.com/w/cpp/utility/move

CHAPTER THIRTEEN

DYNAMIC PROGRAMMING

13.1 Triangle

13.1.1 Problem statement[95]

Given a `triangle` array, return the minimum path sum from top to bottom.

For each step, you may move to an adjacent number of the row below. More formally, if you are on index `i` on the current row, you may move to either index `i` or index `i + 1` on 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).
```

[95] 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?

13.1.2 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.

```
minTotal[i][j] = triangle[i][j] + min(minTotal[i - 1][j - 1], minTotal[i -
↪ 1][j]);
```

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int minimumTotal(vector<vector<int>>& triangle) {
    vector<vector<int>> minTotal(triangle.size());
    minTotal[0] = triangle[0];
    for (int i = 1; i < triangle.size(); i++) {
        const int N = triangle[i].size();
        minTotal[i].resize(N);
        for (int j = 0; j < N; j++) {
            if (j == 0) {
                minTotal[i][j] = triangle[i][j] + minTotal[i - 1][j];
            } else if (j == N - 1) {
```

(continues on next page)

(continued from previous page)

```
                minTotal[i][j] = triangle[i][j] + minTotal[i - 1][j - 1];
            } else {
                minTotal[i][j] = triangle[i][j] + min(minTotal[i - 1][j -␣
↪1], minTotal[i - 1][j]);
            }
        }
    }
    return *min_element(minTotal[triangle.size() - 1].begin(),␣
↪minTotal[triangle.size() - 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
```

Complexity

- Runtime: `O(n*n/2)`, where `n = triangle.length`.
- Extra space: `O(n*n/2)`.

13.1.3 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(vector<vector<int>>& triangle) {
    vector<int> minTotal(triangle.size());
```

(continues on next page)

(continued from previous page)

```
    minTotal[0] = triangle[0][0];
    for (int i = 1; i < triangle.size(); i++) {
        minTotal[i] = triangle[i][i] + minTotal[i - 1];
        for (int j = i - 1; j > 0; j--) {
            minTotal[j] = triangle[i][j] + min(minTotal[j - 1],␣
↪minTotal[j]);
        }
        minTotal[0] = triangle[i][0] + minTotal[0];
    }
    return *min_element(minTotal.begin(), minTotal.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*n/2)`, where `n = triangle.length`.
- Extra space: `O(n)`.

13.2 Largest Divisible Subset

13.2.1 Problem statement[96]

Given a set of **distinct** positive integers `nums`, return the largest subset `answer` such that every pair `(answer[i], answer[j])` of elements in this subset satisfies:

- `answer[i] % answer[j] == 0`, or
- `answer[j] % answer[i] == 0`.

If there are multiple solutions, return any of them.

[96] https://leetcode.com/problems/largest-divisible-subset/

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]
```

Constraints

- `1 <= nums.length <= 1000`.
- `1 <= nums[i] <= 2 * 10^9`.
- All the integers in `nums` are **unique**.

13.2.2 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;

//! @brief compute the maxSubset[i] starting from nums[i]
//!         and store it to _map[i]
//! @note nums is sorted
vector<int> largestDivisibleSubsetOf(vector<int>& nums, int i,
                                    unordered_map<int, vector<int> >& _
↪map) {

    if (_map.find(i) != _map.end()) {
        return _map[i];
    }
    vector<int> maxSubset{nums[i]};
    if (i == nums.size() - 1) {
        _map.insert({i, maxSubset});
        return maxSubset;
    }
    for (int j = i + 1; j < nums.size(); j++) {
        if (nums[j] % nums[i] == 0) {
            auto subset = largestDivisibleSubsetOf(nums, j, _map);
            subset.push_back(nums[i]);
            if (maxSubset.size() < subset.size()) {
                maxSubset = subset;
            }
        }
    }
    _map.insert({i, maxSubset});
    return maxSubset;
}
```

(continues on next page)

(continued from previous page)

```
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()) {
            answer = maxSubset;
        }
    }
    return answer;
}
void printSolution(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);
}
```

```
Output:
[2,1,]
[8,4,2,1,]
```

Complexity

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

13.2.3 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

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
vector<int> largestDivisibleSubset(vector<int>& nums) {
    if (nums.size() <= 1) {
        return nums;
```

(continues on next page)

(continued from previous page)

```
    }
    sort(nums.begin(), nums.end());
    int maxSize = 0;     // the size of the resulting subset
    int maxindex = 0;    // nums[maxindex] is the largest value of the␣
↪resulting subset
    vector<int> subsetSize(nums.size(), 1); // subsetSize[i] stores the␣
↪size
                                            // of the largest subset␣
↪having
                                            // biggest number nums[i]
    vector<int> pre(nums.size(), -1);       // pre[i] stores the previous
                                            // index of i in their␣
↪largest subset
    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(vector<int>& result) {
    cout << "[";
    for (auto& v : result) {
        cout << v << ",";
    }
    cout << "]" << endl;
}
```

(continues on next page)

(continued from previous page)

```
int main() {
    vector<int> nums{2,1,3};
    auto result = largestDivisibleSubset(nums);
    printSolution(result);
    nums = {1,2,4,8};
    result = largestDivisibleSubset(nums);
    printSolution(result);
}
```

```
Output:
[2,1,]
[8,4,2,1,]
```

Complexity

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

13.2.4 Key takeaway

In this interesting problem, we use index mapping to simplify everything. That improves the performance in both runtime and memory.

13.3 Fibonacci Number

13.3.1 Problem statement[97]

The Fibonacci numbers, commonly denoted `F(n)` form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from `0` and `1`. That is,

```
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.
```

Given `n`, calculate `F(n)`.

[97] https://leetcode.com/problems/fibonacci-number/

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.
```

Constraints

- `0 <= n <= 30`.

13.3.2 Solution 1: Recursive

Code

```cpp
#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
```

Complexity

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

13.3.3 Solution 2: Dynamic programming

```cpp
#include <iostream>
#include <vector>
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    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
2
3
```

Complexity

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

13.3.4 Solution 3: Reduce space for dynamic programming

Code

```cpp
#include <iostream>
#include <vector>
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    int f0 = 0;
    int f1 = 1;
    for (int i = 2; i <= n; i++) {
        int f2 = f1 + f0;
        f0 = f1;
        f1 = f2;
    }
    return f1;
}
int main() {
    std::cout << fib(2) << std::endl;
    std::cout << fib(3) << std::endl;
    std::cout << fib(4) << std::endl;
}
```

```
Output:
1
2
3
```

Complexity

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

13.4 Unique Paths

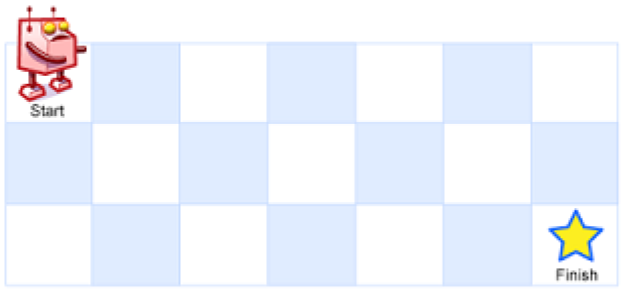
13.4.1 Problem statement[98]

A robot is located at the top-left corner of a `m x n` grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Example 1



```
Input: m = 3, n = 7
Output: 28
```

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
```

(continues on next page)

[98] https://leetcode.com/problems/unique-paths/

(continued from previous page)

```
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`.

13.4.2 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
```

Complexity

- Runtime: `O(2^m + 2^n)`, where `m*n` is the size of the grid.
- Extra space: `O(2^m + 2^n)`.

13.4.3 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) {
    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
```

Complexity

- Runtime: `O(m*n)`, where `m*n` is the size of the grid.
- Extra space: `O(m*n)`.

13.4.4 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) {
    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)`.
- Memory `O(n)`.

13.4.5 Final thought

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.

13.5 Unique Paths II

13.5.1 Problem statement[99]

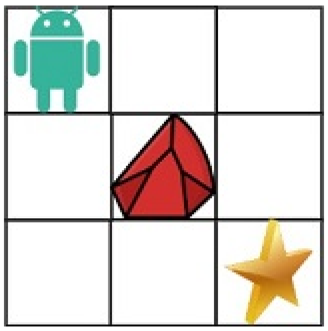
You are given an `m x n` integer array `grid`. There is a robot initially located at the top-left corner (i.e., `grid[0][0]`). The robot tries to move to the bottom-right corner (i.e., `grid[m-1][n-1]`). The robot can only move either down or right at any point in time.

An obstacle and space are marked as `1` or `0` respectively in `grid`. A path that the robot takes cannot include any square that is an obstacle.

Return the number of possible unique paths that the robot can take to reach the bottom-right corner.

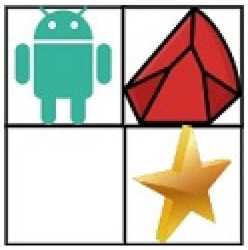
The test cases are generated so that the answer will be less than or equal to `2 * 10^9`.

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
```

Example 2



[99] https://leetcode.com/problems/unique-paths-ii/

```
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`.

13.5.2 Solution: Dynamic programming in place

Let us find the relationship between the positions.

If there is no obstacle at 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(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++) {
        np[i][0] = 1;
    }
    for (int j = 0; j < col && obstacleGrid[0][j] == 0; j++) {
        np[0][j] = 1;
```

(continues on next page)

(continued from previous page)

```
    }
    for (int i = 1; i < row; i++) {
        for (int j = 1; j < col; j++) {
            if (obstacleGrid[i][j] == 0) {
                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 x n` is the size of the `grid`.
- Extra space: `O(m*n)`.

CHAPTER

FOURTEEN

COUNTING

14.1 Single Number

14.1.1 Problem statement[?]

Given a non-empty array of integers `nums`, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant 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
```

[100] https://leetcode.com/problems/single-number/

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.

14.1.2 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(vector<int>& nums) {
    unordered_map<int, int> count;
    for (int 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;
    nums = {4,1,2,1,2};
    cout << singleNumber(nums) << endl;
    nums = {1};
    cout << singleNumber(nums) << endl;
}
```

```
Output:
1
```

(continues on next page)

(continued from previous page)

```
4
1
```

Complexity

- Runtime: `O(N)`, where `N = nums.length`.
- Extra space: `O(N)` (not constant, need another solution).

14.1.3 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(vector<int>& nums) {
    int single = 0;
    for (int 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
```

(continues on next page)

(continued from previous page)

```
4
1
```

Complexity

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

14.2 Max Number of K-Sum Pairs

14.2.1 Problem statement[?]

You are given an integer array `nums` and an integer `k`.

In one operation, you can pick two numbers from the array whose sum equals `k` and remove them from the array.

Return the maximum number of 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.
```

[101] 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`.

14.2.2 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;
int maxOperations(vector<int>& nums, int k) {
    unordered_map<int,int> m;
    int count = 0;
    for (int a : nums) {
        m[a]++;
        if (m[k - a] > 0) {
            if (a != k - a || m[a] >= 2) {
                count++;
                m[a]--;
```

(continues on next page)

(continued from previous page)

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

14.3 First Unique Character in a String

14.3.1 Problem statement[102]

Given a string `s`, find the first non-repeating character in it and return its index. If it does not exist, return `-1`.

Example 1

```
Input: s = "leetcode"
Output: 0
```

[102] https://leetcode.com/problems/first-unique-character-in-a-string/

Example 2

```
Input: s = "loveleetcode"
Output: 2
```

Example 3

```
Input: s = "aabb"
Output: -1
```

Constraints

- `1 <= s.length <= 10^5`.
- s consists of only lowercase English letters.

14.3.2 Solution 1: Using map to store the appearances

Code

```
#include <iostream>
#include <unordered_map>
using namespace std;
int firstUniqChar(string s) {
    unordered_map<char, int> count;
    for (char& 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
```

Complexity

- Runtime: `O(N)`, where `N = s.length`.
- Extra space: `O(1)` if `N` is very larger than `26`.

14.3.3 Solution 2: Using vector to store the appearances

Code

```cpp
#include <iostream>
#include <vector>
using namespace std;
int firstUniqChar(string s) {
    vector<int> count(26);
    for (char& c : s) {
        count[c - 'a']++;
    }
    for (int i = 0; i < s.length(); i++) {
        if (count[s[i] - 'a'] == 1) {
            return i;
        }
    }
    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)` if `N` is very larger than `26`.

CHAPTER

FIFTEEN

PREFIX SUM

15.1 Running Sum of 1d Array

15.1.1 Problem statement[103]

Given an array `nums`. We define a running sum of an array as `runningSum[i] = sum(nums[0]...nums[i])`.

Return the running sum of `nums`.

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].
```

[103] https://leetcode.com/problems/running-sum-of-1d-array/

Example 3

```
Input: nums = [3,1,2,10,1]
Output: [3,4,6,16,17]
```

Constraints

- `1 <= nums.length <= 1000`.
- `-10^6 <= nums[i] <= 10^6`.

15.1.2 Solution 1: Unchange `nums`

Code

```
#include <vector>
#include <iostream>
using namespace std;
vector<int> runningSum(vector<int>& nums) {
    vector<int> rs;
    int s = 0;
    for (int n : nums) {
        s += n;
        rs.push_back(s);
    }
    return rs;
}
void printResult(vector<int>& sums) {
    cout << "[";
    for (int 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);
```

(continues on next page)

(continued from previous page)

```
    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)`.

15.1.3 Solution 2: Change `nums`

If `nums` is allowed to be changed, you could use it to store the result directly.

Code

```
#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;
}
void printResult(vector<int>& sums) {
    cout << "[";
    for (int s: sums) {
        cout << s << ",";
    }
    cout << "]\n";
}
int main() {
    vector<int> nums{1,2,3,4};
```

(continues on next page)

(continued from previous page)

```
    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)`.

15.2 Maximum Subarray

15.2.1 Problem statement[?]

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A **subarray** is a **contiguous** part of an array.

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.
```

[104] https://leetcode.com/problems/maximum-subarray/

Example 2

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

Example 3

```
Input: nums = [5,4,-1,7,8]
Output: 23
```

Constraints

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

15.2.2 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

```cpp
int maxSubArray(vector<int>& nums) {
    int maxSum = -10000;
    int currSum = 0;
    for (auto& num : nums) {
        currSum = currSum < 0 ? num : currSum + num;
        maxSum = max(maxSum, currSum);
    }
    return maxSum;
}
int main() {
    vector<int> nums = {-2,1,-3,4,-1,2,1,-5,4};
```

(continues on next page)

(continued from previous page)

```
    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)`.

15.3 Subarray Sum Equals K

15.3.1 Problem Statement[105]

Given an array of integers `nums` and an integer `k`, return the total number of continuous subarrays whose sum equals to `k`.

Example 1

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

[105] https://leetcode.com/problems/subarray-sum-equals-k/

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`.

15.3.2 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

```cpp
#include <iostream>
#include <vector>
using namespace std;
int subarraySum(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++;
            }
```

(continues on next page)

(continued from previous page)

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

15.3.3 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], nums[j+1], ..., 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(vector<int>& nums, int k) {
    vector<int> sum(nums.size());
    sum[0] = nums[0];
    for (int i = 1; i < nums.size(); i++) {
        sum[i] = sum[i-1] + nums[i];
    }
    int count = 0;
    for (int i = 0; i < nums.size(); i++) {
        if (sum[i] == k) {
            count++;
        }
        for (int j = 0; j < i; j++) {
            if (sum[i] - sum[j] == 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
```

Complexity

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

15.3.4 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

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
int subarraySum(vector<int>& nums, int k) {
    int count = 0;
    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()) {
            count += it->second;
        }
        sums[sumi]++;
    }
    return count;
}
int main() {
    vector<int> nums{1,1,1};
    cout << subarraySum(nums, 2) << endl;
    nums = {1,2,3};
```

(continues on next page)

(continued from previous page)

```
    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)`.

CHAPTER

SIXTEEN

TWO POINTERS

16.1 Container With Most Water

16.1.1 Problem statement[106]

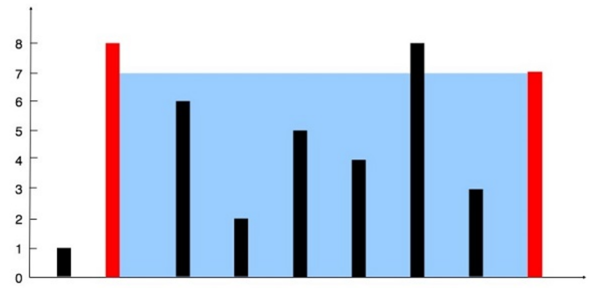
You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `i-th` line are `(i, 0)` and `(i, height[i])`.

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 section) the␣
↪container can contain is 49.
```

[106] 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`.

16.1.2 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(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;
    height = {1,1};
    cout << maxArea(height) << endl;
}
```

```
Output:
49
1
```

Complexity

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

16.1.3 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(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--;
        }
    }
    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)`.

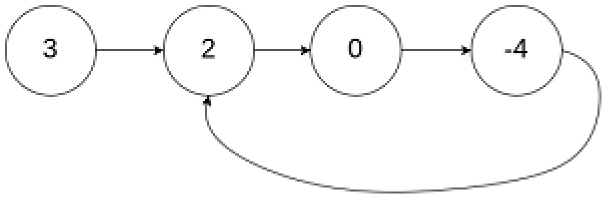
16.2 Linked List Cycle

16.2.1 Problem statement[107]

Given head, the head of a linked list, determine if the linked list has a cycle in it.

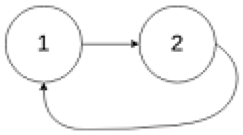
Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1



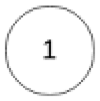
```
Input: head = [3,2,0,-4], where -4 links next to 2.
Output: true
```

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.
```

[107] https://leetcode.com/problems/linked-list-cycle/

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?

16.2.2 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*, int> m;
    while (head) {
        if (m[head] > 0) {
            return true;
        }
        m[head]++;
        head = head->next;
    }
    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;
    }
    {
```

(continues on next page)

(continued from previous page)

```
        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 length of the linked list.
- Extra space: `O(N)`.

16.2.3 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;
    }
    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);
```

(continues on next page)

(continued from previous page)

```
        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

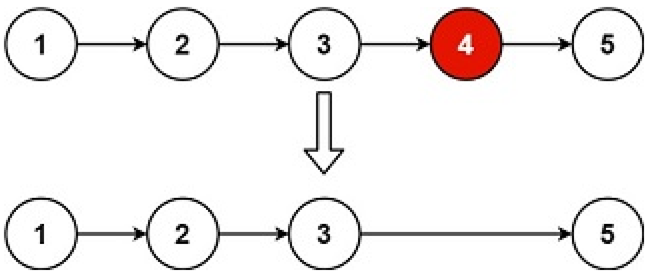
- Runtime: `O(N)`, where `N` is the length of the linked list.
- Extra space: `O(1)`.

16.3 Remove Nth Node From End of List

16.3.1 Problem statement[108]

Given the `head` of a linked list, remove the `n-th` node from the end of the list and return its head.

Example 1



[108] https://leetcode.com/problems/remove-nth-node-from-end-of-list/

```
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]
```

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?

16.3.2 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) {
```

(continues on next page)

(continued from previous page)

```
    vector<ListNode*> nodes;
    ListNode* node = head;
    while (node)
    {
        nodes.push_back(node);
        node = node->next;
    }
    node = nodes[nodes.size() - n];
    if (node == head) {
        head = node->next;
    } else {
        ListNode* pre = nodes[nodes.size() - n - 1];
        pre->next = node->next;
    }
    return head;
}
void printList(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,]
[]
[4,]
```

Complexity

- Runtime: `O(sz)`, where `sz` is the number of nodes in the list.
- Extra space: `O(sz)`.

16.3.3 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;
    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;
    for (int i = 0; i < n; i++) {
        fast = fast->next;
    }
    if (fast == nullptr) {
        return head->next;
    }
    ListNode* slow = head;
    while (fast->next)
    {
        slow = slow->next;
        fast = fast->next;
    }
    slow->next = slow->next->next;
    return head;
```

(continues on next page)

(continued from previous page)

```
}
void printList(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,]
[]
[4,]
```

Complexity

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

16.4 Shortest Unsorted Continuous Subarray

16.4.1 Problem statement[109]

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
```

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?

[109] https://leetcode.com/problems/shortest-unsorted-continuous-subarray/

16.4.2 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 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(vector<int>& nums) {
    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
```

(continues on next page)

(continued from previous page)

```
0
0
```

Complexity

- Runtime: `O(nlogn)`, where `n = nums.length` because of the sorting algorithm.
- Extra space: `O(n)`.

16.4.3 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

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int findUnsortedSubarray(vector<int>& nums) {
    const int n = nums.size();
    int left = n - 1;
    int min = nums[n - 1];
    for (int i = n - 1; i >= 0; i--) {
        if (min < nums[i]) {
            left = i;
        } else {
            min = nums[i];
        }
    }
    int right = 0;
```

(continues on next page)

(continued from previous page)

```
    int max = nums[0];
    for (int i = 0; i < nums.size(); i++) {
        if (max > nums[i]) {
            right = i;
        } else {
            max = nums[i];
        }
    }
    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
```

Complexity

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

16.4.4 Conclusion

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 + mlogm) < O(NlogN)` where `N` is the length of the whole array and `m` is the length of the shortest subarray.

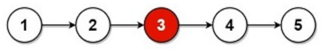
16.5 Middle of the Linked List

16.5.1 Problem statement[110]

Given the head of a singly linked list, return *the middle node of the linked list*.

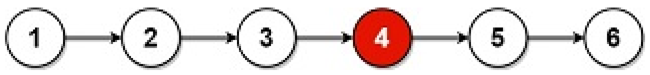
If there are two middle nodes, return *the second middle* node.

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.
```

Constraints

- The number of nodes in the list is in the range `[1, 100]`.
- `1 <= Node.val <= 100`.

[110] https://leetcode.com/problems/middle-of-the-linked-list/

16.5.2 Solution 1: Counting the number of nodes

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* 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(ListNode *head) {
    ListNode *node = head;
    std::cout << "[";
    while (node) {
        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);
```

(continues on next page)

(continued from previous page)

```
    print(result);

    ListNode six(6);
    five.next = &six;
    result = middleNode(&one);
    print(result);
}
```

```
Output:
[3,4,5,]
[4,5,6,]
```

Complexity

- Runtime: `O(N + N/2)`, where `N` is the number of nodes.
- Extra space: `O(1)`.

16.5.3 Solution 2: Slow and fast 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;
```

(continues on next page)

(continued from previous page)

```
        fast = fast->next->next;
    }
    return slow;
}
void print(ListNode *head) {
    ListNode *node = head;
    std::cout << "[";
    while (node) {
        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,]
```

Complexity

- Runtime: `O(N/2)`, where `N` is the number of nodes.
- Extra space: `O(1)`.

16.5.4 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.

CHAPTER SEVENTEEN

MATHEMATICS

17.1 Best Time to Buy and Sell Stock

17.1.1 Problem statement[111]

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.

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.
```

[111] https://leetcode.com/problems/best-time-to-buy-and-sell-stock/

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`.

17.1.2 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(vector<int>& prices) {
    int maxProfit = 0;
    for (int i = 0; i < prices.size(); i++) {
        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
```

Complexity

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

17.1.3 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(vector<int>& prices) {
    int maxProfit = 0;
    int i = 0;
    while (i < prices.size()) {
        while (i < prices.size() - 1 && prices[i] >= prices[i + 1]) {
            i++;
        }
        auto imax = max_element(prices.begin() + i, prices.end());
        auto imin = min_element(prices.begin() + i, imax);
        maxProfit = max(maxProfit, *imax - *imin);
        i = distance(prices.begin(), imax) + 1;
    }
    return maxProfit;
}
int main() {
    vector<int> prices{7,1,5,3,6,4};
    cout << maxProfit(prices) << endl;
```

(continues on next page)

(continued from previous page)

```
    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
```

Complexity

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

17.1.4 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(vector<int>& prices) {
    int maxProfit = 0;
    int minPrice = prices[0];
    for (int i = 1; i < prices.size(); i++)  {
        minPrice = min(minPrice, prices[i]);
        maxProfit = max(maxProfit, prices[i] - minPrice);
    }
    return maxProfit;
}
int main() {
    vector<int> prices{7,1,5,3,6,4};
```

(continues on next page)

(continued from previous page)

```
    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
```

Complexity

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

17.2 Count Sorted Vowel Strings

17.2.1 Problem statement[112]

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.

[112] https://leetcode.com/problems/count-sorted-vowel-strings/

Example 1

```
Input: n = 1
Output: 5
Explanation: The 5 sorted strings that consist of vowels only are ["a","e
↪","i","o","u"].
```

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`.

17.2.2 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

```cpp
#include <iostream>
using namespace std;
int countVowelStrings(int n) {
    int a, e, i, o, u;
    a = e = i = o = u = 1;
```

(continues on next page)

(continued from previous page)

```
    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
```

Complexity

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

17.2.3 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 `(n + 4 choose 4)`, where `(n choose k)` is the binomial coefficient[113]:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

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

```
#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)`.

[113] https://en.wikipedia.org/wiki/Binomial_coefficient

17.3 Concatenation of Consecutive Binary Numbers

17.3.1 Problem statement[114]

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.
```

[114] https://leetcode.com/problems/concatenation-of-consecutive-binary-numbers/

Constraints

- `1 <= n <= 10^5`.

17.3.2 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

```cpp
#include <cmath>
#include <iostream>
int concatenatedBinary(int n) {
    unsigned long long result = 1;
    for (int i = 2; i <= n; i++) {
        const int num_bits = std::log2(i) + 1;
        result = ((result << num_bits) + i) % 1000000007;
    }
    return result;
}
```

(continues on next page)

(continued from previous page)

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

17.4 Partitioning Into Minimum Number Of Deci-Binary Numbers

17.4.1 Problem statement[115]

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
```

[115] https://leetcode.com/problems/partitioning-into-minimum-number-of-deci-binary-numbers/

Example 2

```
Input: n = "82734"
Output: 8
```

Example 3

```
Input: n = "27346209830709182346"
Output: 9
```

Constraints

- `1 <= n.length <= 10^5`.
- `n` consists of only digits.
- `n` does not contain any leading zeros and represents a positive integer.

17.4.2 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

```cpp
#include <iostream>
using namespace std;
int minPartitions(string n) {
    char maxDigit = '0';
    for (char& d : n) {
        maxDigit = max(maxDigit, d);
    }
    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)`.

17.5 Maximum Units on a Truck

17.5.1 Problem statement[116]

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`.

[116] https://leetcode.com/problems/maximum-units-on-a-truck/

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.
```

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`.

17.5.2 Solution: Greedy algorithm

Put the boxes having more units first.

Code

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
```

(continues on next page)

(continued from previous page)

```
    // 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) {
            maxUnits += boxTypes[i][0] * boxTypes[i][1];
            truckSize -= boxTypes[i][0];
        } else {
            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
```

Complexity

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

17.5.3 Modern C++ STL notes

Note that two `vectors`[117] 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 `sort`[118] algorithm will use the dictionary order to sort them by default.

17.6 Excel Sheet Column Number

17.6.1 Problem statement[?]

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
```

[117] https://en.cppreference.com/w/cpp/container/vector
[118] https://en.cppreference.com/w/cpp/algorithm/sort
[119] https://leetcode.com/problems/excel-sheet-column-number/

Example 2

```
Input: columnTitle = "AB"
Output: 28
```

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"]`.

17.6.2 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'
"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

```cpp
#include <iostream>
using namespace std;
int titleToNumber(string columnTitle) {
    int column = 0;
    for (char 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
```

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 substracts `64`.

17.7 Perfect Squares

17.7.1 Problem statement[?]

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.
```

[120] https://leetcode.com/problems/perfect-squares/

Example 2

```
Input: n = 13
Output: 2
Explanation: 13 = 4 + 9.
```

Example 3

```
Input: n = 7
Output: 4
Explanation: 7 = 4 + 1 + 1 + 1.
```

Example 4

```
Input: n = 12
Output: 3
Explanation: 12 = 4 + 4 + 4.
```

Constraints

- `1 <= n <= 10^4`.

17.7.2 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

```cpp
#include <iostream>
#include <cmath>
#include <unordered_map>
using namespace std;
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) {
        ns[n] = 1;
        return 1;
    }
    int result = n;
    for (int i = 1; i <= sq; i++) {
        result = min(result, nsq(n - i*i, ns));
    }
    ns[n] = result + 1;
    return ns[n];
}
int numSquares(int n) {
    unordered_map<int, int> ns;
    return nsq(n, ns);
```

(continues on next page)

(continued from previous page)

```
}
int main() {
    cout << numSquares(12) << endl;
    cout << numSquares(13) << endl;
}
```

```
Output:
3
2
```

Complexity

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

17.7.3 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[121], 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`.

[121] https://en.wikipedia.org/wiki/Lagrange%27s_four-square_theorem

2. There is another theorem, Legendre's Three-Square Theorem[122], which states that `numSquares(n)` cannot be 1, 2, or 3 if `n` can be expressed as

   ```
   n = 4^a(8*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*0 + 7)`. It can only be written as `7 = 4 + 1 + 1 + 1`.

Code

```
#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;
    }
    // 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;
}
```

(continues on next page)

[122] https://en.wikipedia.org/wiki/Legendre%27s_three-square_theorem

(continued from previous page)

```
int main() {
    cout << numSquares(12) << endl;
    cout << numSquares(13) << endl;
}
```

```
Output:
3
2
```

Complexity

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

17.7.4 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 * 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 * m = (2^a)^2 * m = (2^a * x)^2 + (2^a * y)^2 + (2^a * 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);
    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(logn)`.
- Extra space: `O(1)`.

17.7.5 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 from this code challenge.

Have fun with coding and mathematics!

17.8 Maximum Product of Word Lengths

17.8.1 Problem statement[123]

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`.

Example 1

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

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

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.

17.8.2 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(vector<string>& words) {
    int maxP = 0;
    for (int i = 0; i < words.size(); i++) {
        vector<bool> visited(26, false);
        for (char c : words[i]) {
            visited[c - 'a'] = true;
        }
        for (int j = i + 1; j < words.size(); j++) {
            bool found = false;
            for (char c : words[j]) {
```

(continues on next page)

(continued from previous page)

```
                if (visited[c - 'a']) {
                    found = true;
                    break;
                }
            }
            if (!found) {
                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
```

Complexity

- Runtime: `O(N^2**M)`, where `N = words.length`, `M = avg(words[i].length)`.
- Extra space: `O(1)`.

17.8.3 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.

For 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(vector<string>& words) {
    int maxP = 0;
    vector<int> mask(words.size());
    for (int i = 0; i < words.size(); i++) {
        for (char c : words[i]) {
            mask[i] |= 1 << (c - 'a');
        }
        for (int j = 0; j < i; j++) {
            if ((mask[j] & mask[i]) == 0) {
                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
```

Complexity

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

17.9 Power of Three

17.9.1 Problem statement[124]

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.
```

[124] https://leetcode.com/problems/power-of-three/

Example 3

```
Input: n = -1
Output: false
Explanation: There is no x where 3^x = (-1).
```

Constraints

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

Follow up: Could you solve it without loops/recursion?

17.9.2 Solution 1: Recursion

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
```

Complexity

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

17.9.3 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 <= x <= y`.

Because the constraint of the problem is `n <= 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
```

Complexity

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

17.10 Power of Four

17.10.1 Problem statement[125]

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`.

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?

17.10.2 Solution 1: Division by four

Code

```
#include <iostream>
using namespace std;
bool isPowerOfFour(int n) {
```

(continues on next page)

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

(continued from previous page)

```
    while (n % 4 == 0 && n > 0) {
        n /= 4;
    }
    return n == 1;
}
int main() {
    cout << isPowerOfFour(16) << endl;
    cout << isPowerOfFour(5) << endl;
    cout << isPowerOfFour(1) << endl;
}
```

```
Output:
1
0
1
```

Complexity

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

17.10.3 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
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

```
#include <iostream>
using namespace std;
bool isPowerOfFour(int n) {
    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)`.

17.11 Wiggle Subsequence

17.11.1 Problem statement[126]

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

[126] https://leetcode.com/problems/wiggle-subsequence/

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).
```

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?

17.11.2 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

```cpp
#include <iostream>
#include <vector>
using namespace std;
int wiggleMaxLength(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;
        }
        i++;
```

(continues on next page)

(continued from previous page)

```
    }
    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)`.

17.12 Hamming Distance

17.12.1 Problem statement[?]

The Hamming distance[128] 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.

[127] https://leetcode.com/problems/hamming-distance/
[128] https://en.wikipedia.org/wiki/Hamming_distance

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
```

Constraints

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

17.12.2 Solution: Using bitwise operator XOR

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

Code

```
#include <iostream>
int hammingDistance(int x, int y) {
    int z = x ^ y;
    int count = 0;
    while (z) {
        count += z & 1;
        z = z >> 1;
    }
    return count;
}
int main() {
    std::cout << hammingDistance(1,4) << std::endl; // 2
```

(continues on next page)

(continued from previous page)

```
    std::cout << hammingDistance(1,3) << std::endl; // 1
}
```

```
Output:
2
1
```

Complexity

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

17.13 Minimum Moves to Equal Array Elements II

17.13.1 Problem statement[129]

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.

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]
```

[129] https://leetcode.com/problems/minimum-moves-to-equal-array-elements-ii/

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`.

17.13.2 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[130] 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`.

[130] https://en.wikipedia.org/wiki/Median

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};
    cout << minMoves2(nums) << endl;
    nums = {1,10,2,9};
    cout << minMoves2(nums) << endl;
}
```

```
Output:
2
16
```

Complexity

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

17.13.3 Solution 2: Using `std::nth_element` to compute the median

What you only need in the 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`[131] 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;
    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
```

[131] https://en.cppreference.com/w/cpp/algorithm/nth_element

Complexity

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

17.13.4 Modern C++ notes

In the code of Solution 2, the partial sorting algorithm `std::nth_element`[132] will make sure for all indices `i` and `j` that satisfy `0 <= i <= mid <= j < nums.length`,

```
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.

17.14 Array Nesting

17.14.1 Problem statement[?]

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]`.

[132] https://en.cppreference.com/w/cpp/algorithm/nth_element
[133] 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.

17.14.2 Solution: Understanding the math behind

A permutation[134] 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,
5 -> 6,
6 -> 2.
```

You can always rearrange the definition of a permutation into groups of cyclic chains (factors).

[134] https://en.wikipedia.org/wiki/Permutation

```
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

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int arrayNesting(vector<int>& nums) {
    int maxLen(0);
    vector<bool> visited(nums.size());
    for (auto i : nums) {
        if (visited[i]) {
            continue;
        }
        int len(0);
        while (!visited[i]) {
            visited[i] = true;
            i = nums[i];
            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};
```

(continues on next page)

(continued from previous page)

```
    cout << arrayNesting(nums) << endl;
    nums = {2,0,1};
    cout << arrayNesting(nums) << endl;
}
```

```
Output:
4
1
2
3
```

Complexity

- Runtime: `O(n)`, where `n = nums.length`.
- Extra space: much less than `O(n)` since `vector<bool>`[135] is optimized for space efficiency.

17.15 Subsets

17.15.1 Problem Statement[?]

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]]
```

[135] https://en.cppreference.com/w/cpp/container/vector_bool
[136] https://leetcode.com/problems/subsets/

Example 2

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

Constraints

- `1 <= nums.length <= 10`.
- `-10 <= nums[i] <= 10`.
- All the numbers of `nums` are unique..

17.15.2 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(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(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 = nums.length`.
- Extra space: `O(2^N)`.

CHAPTER EIGHTEEN

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.

18.1 Coding challenge best practices

Here are some best practices to keep in mind when working on coding challenges:

- **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.
- **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.
- **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.

- **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.
- **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.
- **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.
- **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.

CHAPTER NINETEEN

THANK YOU!

Thank you for taking the time to read 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!

Visit my blog leetsolve.com[137] *to read more content.*

I love to hear what you think about my content. Just drop an email to nhut@nhutnguyen.com and share your thoughts.

137 https://leetsolve.com