# 1. 01_Arrays_&_Hashing/01_Contains_Duplicate/0217-contains-duplicate.cpp

```
/*
Problem: LeetCode 217 - Contains Duplicate

Intuition:
To solve this problem, we can utilize the property of a hash set. By storing each element encountered in the set and checking for collisions, we can efficiently determine if any duplicates exist in the array. If a collision occurs, it indicates the presence of a duplicate element.

Approach:
1. Initialize an empty hash set.
2. Iterate through each element num in the input array nums:
    - If num is already present in the hash set, return true as we have found a duplicate.
    - Otherwise, add num to the hash set.
3. If no duplicates are found after iterating through all elements, return false.

Time Complexity:
The time complexity of this approach is O(n), where n is the size of the input array nums. This is because we iterate through the array once and perform constant-time operations for each element.

Space Complexity:
The space complexity is O(n), as the hash set can potentially store all elements of the input array.
*/

class Solution {
  public:
    bool containsDuplicate(vector<int> &nums) {
        unordered_set<int> seen;

        for (int num : nums) {
            if (seen.count(num) > 0) {
                return true; // Duplicate found
            }

            seen.insert(num);
        }

        return false; // No duplicates found
    }
};

/*
auto init = [](){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    return 0;
}();

class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        size_t count = nums.size();

        // Sort using LSD Radix
        Sort(reinterpret_cast<unsigned int*>(&(nums[0])), count);

        --count;
        for(size_t i = 0; i < count; ++i)
            if(nums[i] == nums[i+1])
```

```
                return true;

        return false;
    }

private:
    void Sort(unsigned int* start, size_t len) {
        unsigned int* buffer = new unsigned int[len];
        LSDRadix(start, buffer, len);
        delete[] buffer;
    }

    void LSDRadix(unsigned int* input, unsigned int* buffer, size_t len) {
        for(int bits = 0; bits < 16; bits += 8) {
            size_t count[256] = {0};
            for(int x = 0; x < len; ++x)
                ++count[(input[x] >> bits) & 0xff];
            for(int x = 0; x < 255; ++x)
                count[x + 1] += count[x];
            for(int x = len - 1; x >= 0; --x)
                buffer[--count[(input[x] >> bits) & 0xff]] = input[x];
            unsigned int* temp = input;
            input = buffer;
            buffer = temp;
        }
    }
};
*/
```

## 2. 01_Arrays_&_Hashing/02_Valid_Anagram/0242-valid-anagram.cpp

```
/*
Problem: LeetCode 242 - Valid Anagram

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

Intuition:
To determine if two strings are valid anagrams, we can compare the counts of each character in
both strings. If the counts of all characters are equal, the strings are valid anagrams.

Approach:
1. If the lengths of the two input strings s and t are not equal, return false.
2. Initialize an array of size 26 to store the counts of each character.
3. Iterate through each character in string s and increment its count in the array.
4. Iterate through each character in string t and decrement its count in the array.
5. If any count in the array is not zero, return false as the characters do not match.
6. Return true if all counts in the array are zero.

Time Complexity:
The time complexity of this approach is O(n), where n is the length of the input strings s and t.
This is because we iterate through both strings once to update the counts of each character.

Space Complexity:
The space complexity is O(1), as we use a fixed-size array of size 26 to store the counts.
*/

auto init = []() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    return 0;
}
();

class Solution {
  public:
    bool isAnagram(string s, string t) {
        if (s.length() != t.length()) {
            return false;
        }

        int count[26] = {0};

        for (char ch : s) {
            count[ch - 'a']++;
        }

        for (char ch : t) {
            count[ch - 'a']--;
        }

        for (int i = 0; i < 26; i++) {
            if (count[i] != 0) {
                return false;
            }
        }

        return true;
    }
};
```

```
// class Solution {
// public:
//     bool isAnagram(string s, string t) {
//         if(s.size() != t.size())
//                 return false;

//         sort(s.begin(), s.end());
//         sort(t.begin(), t.end());

//         return s == t;
//     }
// };
```

## 3. 01_Arrays_&_Hashing/03_Two_Sum/0001-two-sum.cpp

```
/*
Problem: LeetCode 1 - Two Sum

Description:
Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

Intuition:
To find the pair of numbers that add up to the target, we can utilize a hash map. By iterating through the array and checking if the complement (target - current number) exists in the hash map, we can efficiently find the desired pair.

Approach:
1. Initialize an empty hash map to store the elements and their indices.
2. Iterate through each element num and its index in the input array nums:
    - Calculate the complement as target - num.
    - If the complement exists in the hash map, return the indices [map[complement], index].
    - Otherwise, add the current element num and its index to the hash map.
3. If no solution is found, return an empty vector or handle it as per the problem's requirement.

Time Complexity:
The time complexity of this approach is O(n), where n is the size of the input array nums.
This is because we iterate through the array once and perform constant-time operations for each element.

Space Complexity:
The space complexity is O(n), as the hash map can potentially store all elements of the input array.
*/

class Solution {
  public:
    vector<int> twoSum(vector<int> &nums, int target) {
        unordered_map<int, int> numMap;

        for (int i = 0; i < nums.size(); i++) {
            int complement = target - nums[i];

            if (numMap.count(complement) > 0) {
                return {numMap[complement], i};
            }

            numMap[nums[i]] = i;
        }

        return {}; // Handle no solution as per problem's requirement
    }
};

// class Solution {
// public:
//     vector<int> twoSum(vector<int>& nums, int target) {
//         unordered_map<int, int> M;
//         vector<int> res;
//         for(int i = 0; i < nums.size(); i++) {
//             int compliment = target - nums[i];

//             // If we find the compliment
//             if(M.find(compliment) != M.end()) {
//                 // returning i and index of compliment
```

```
//                 res.push_back(M[compliment]);
//                 res.push_back(i);
//             }
//             else // insert pair
//                 M.insert( {nums[i], i});
//         }
//         return res;
//     }
// };
```

## 4. 01_Arrays_&_Hashing/04_Group_Anagrams/0049-group-anagrams.cpp

```
/*
Problem: LeetCode 49 - Group Anagrams

Description:
Given an array of strings strs, group the anagrams together. You can return the answer in any order.

Intuition:
An anagram is a word formed by rearranging the letters of another word. To group anagrams together, we can utilize a hash map. By sorting each string and using the sorted string as a key in the hash map, we can efficiently group the anagrams.

Approach:
1. Initialize an empty hash map to store the groups of anagrams.
2. Iterate through each string str in the input array strs:
    - Sort the characters of str to create a sorted string.
    - If the sorted string is already present in the hash map, add str to its corresponding group.
    - Otherwise, create a new group in the hash map with the sorted string as the key and str as the initial value.
3. Return the values of the hash map, which represent the grouped anagrams.

Time Complexity:
The time complexity of this approach is O(n * k * log k), where n is the size of the input array strs and k is the maximum length of a string in strs.
This is because we iterate through the array and sort each string, which takes O(k * log k) time for each string.

Space Complexity:
The space complexity is O(n * k), as we store all the strings in the hash map, where n is the number of groups and k is the maximum length of a string.
*/

class Solution {
  public:
    vector<vector<string>> groupAnagrams(vector<string> &strs) {
        // To store the sorted string and all its anagrams
        unordered_map<string, vector<string>> M;

        /*
            Consider example 1 : strs = ["eat","tea","tan","ate","nat","bat"]
            After the below opeartion of for loop map will contain -
            aet -- eat, tea, ate
            ant -- tan, nat
            abt -- bat
        */
        for (int i = 0; i < strs.size(); i++) {
            string T = strs[i];
            sort(T.begin(), T.end());
            M[T].push_back(strs[i]);
        }

           // Now pushing all the anagrams(vector<string>) of one word, one by one, into result vector
        vector< vector<string> > result;

        for (auto i : M) {
            result.push_back(i.second);
        }

        return result;
```

```
    }
};
```

## 5. 01_Arrays_&_Hashing/05_Top_K_Frequent_Elements/0347-top-k-frequent-element

```
/*
Problem: LeetCode 347 - Top K Frequent Elements

Description:
Given an integer array nums and an integer k, return the k most frequent elements. You may return
the answer in any order.

Intuition:
To find the k most frequent elements, we can utilize a combination of a hash map and bucket sort.
By counting the frequencies of elements using the hash map and using bucket sort to group elements
by their frequencies, we can efficiently find the k most frequent elements.

Approach:
1. Initialize an empty hash map, m, to store the frequencies of elements.
2. Iterate through each element num in the input array nums:
    - Increment the frequency count of num in the hash map.
3. Create a vector of vectors, buckets, to act as buckets for grouping elements based on their
frequencies.
4. Iterate through the key-value pairs in the hash map:
    - Place each key (element) in the corresponding bucket based on its frequency.
5. Create an empty vector, result, to store the k most frequent elements.
6. Iterate from the highest bucket index down to 0:
    - If the result vector size reaches k, break the loop.
    - If the current bucket is not empty, append its elements to the result vector.
7. Return the result vector, which represents the k most frequent elements.

Time Complexity:
The time complexity of this approach is O(n), where n is the size of the input array nums. This is
because we iterate through the array once to count the frequencies and place elements in the
buckets.

Space Complexity:
The space complexity is O(n), as we store the frequencies of elements in the hash map and the
bucket vectors.
*/

class Solution {
  public:
    vector<int> topKFrequent(vector<int> &nums, int k) {
        int n = nums.size();
        unordered_map<int, int> m;

        for (int i = 0; i < n; i++) {
            m[nums[i]]++;
        }

        vector<vector<int>> buckets(n + 1);

        for (auto it = m.begin(); it != m.end(); it++) {
            buckets[it->second].push_back(it->first);
        }

        vector<int> result;

        for (int i = n; i >= 0; i--) {
            if (result.size() >= k) {
                break;
            }

            if (!buckets[i].empty()) {
```

```
                result.insert(result.end(), buckets[i].begin(), buckets[i].end());
            }
        }

        return result;
    }
};

/*
! Using Priority Queue
Problem: LeetCode 347 - Top K Frequent Elements

Description:
Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.

Intuition:
To find the k most frequent elements, we can utilize a combination of a hash map and a priority queue (min-heap). By counting the frequencies of elements using the hash map and maintaining a min-heap of size k, we can efficiently find the k most frequent elements.

Approach:
1. Initialize an empty hash map to store the frequencies of elements.
2. Iterate through each element num in the input array nums:
    - Increment the frequency count of num in the hash map.
3. Initialize a min-heap to store the k most frequent elements based on their frequencies.
4. Iterate through the elements and frequencies in the hash map:
    - Push the current element into the min-heap.
      - If the size of the min-heap exceeds k, remove the smallest element (with the lowest frequency).
5. Return the elements in the min-heap, which represent the k most frequent elements.

Time Complexity:
The time complexity of this approach is O(n log k), where n is the size of the input array nums. This is because we iterate through the array once to count the frequencies and perform log k operations for each element insertion and removal in the min-heap.

Space Complexity:
The space complexity is O(n), as we store the frequencies of elements in the hash map and the k most frequent elements in the min-heap.
*/
/*
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> freqMap;
        for (int num : nums) {
            freqMap[num]++;
        }

        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> minHeap;
        for (auto& kvp : freqMap) {
            minHeap.push({kvp.second, kvp.first});
            if (minHeap.size() > k) {
                minHeap.pop();
            }
        }

        vector<int> result;
        while (!minHeap.empty()) {
            result.push_back(minHeap.top().second);
            minHeap.pop();
        }
```

```
        return result;
    }
};
*/

// ! O(n log n) solution
/*
class Solution {
public:
     // Approach - First make a frequency map normally, then insert key value pairs(frequency first, value second) in vector and sort in descending order
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> M;
        vector< pair<int, int> > arr;
        vector<int> result;

        for(int i = 0; i < nums.size(); i++) {
            // it reaches end of map if it didn't find the element
            if(M.find(nums[i]) == M.end())
                M[nums[i]] = 1;
            else
                M[nums[i]]++;
        }

        for(auto i : M) {
            arr.push_back( make_pair(i.second, i.first) );
        }

        sort(arr.rbegin(), arr.rend());

        for(int i = 0; i < k; i++) {
            result.push_back(arr[i].second);
        }
        return result;
    }
};
*/
```

## 6. 01_Arrays_&_Hashing/06_Product_of_Array_Except_Self/0238-product-of-array-ex

```
/*
Problem: LeetCode 238 - Product of Array Except Self

Description:
Given an integer array nums, return an array output such that output[i] is equal to the product of
all the elements of nums except nums[i].

Intuition:
To find the product of all elements except self, we can utilize prefix and suffix products. By
calculating the prefix product from the left and the suffix product from the right, we can obtain
the desired result efficiently.

Approach:
1. Initialize an output array of the same size as the input array nums.
2. Calculate the prefix product from the left side of the array and store it in the output array.
3. Calculate the suffix product from the right side of the array and multiply it with the
corresponding element in the output array.
4. Return the resulting output array.

Time Complexity:
The time complexity of this approach is O(n), where n is the size of the input array nums.
This is because we iterate through the array twice, once for calculating the prefix product and
once for calculating the suffix product.

Space Complexity:
The space complexity is O(1), as we are reusing the input array for storing the output.
*/

class Solution {
  public:
    vector<int> productExceptSelf(vector<int> &nums) {
        int n = nums.size();
        vector<int> output(n, 1);
        // Calculate prefix product
        int prefix = 1;

        for (int i = 0; i < n; i++) {
            output[i] *= prefix;
            prefix *= nums[i];
        }

        // Calculate suffix product and multiply with prefix product stored in the output array
        int suffix = 1;

        for (int i = n - 1; i >= 0; i--) {
            output[i] *= suffix;
            suffix *= nums[i];
        }

        return output;
    }
};

// class Solution {
// public:
//     vector<int> productExceptSelf(vector<int>& nums) {
//         vector<int> res(nums.size(), 1);
//         // First, compute the prefix product and store in res
//         // res[i] = product of elements in nums from index 0, 1, ... to i - 1
//         for (int i = 0; i < nums.size() - 1; i++) {
```

```
//             res[i+1] = nums[i] * res[i];
//         }
//         // Second, compute the final result
//         int suffixProduct = 1;
//         for(int j = nums.size()-1; j > 0; j--) {
//             suffixProduct *= nums[j];
//             res[j-1] *= suffixProduct;
//         }
//         return res;
//     }
// };
```

## 7. 01_Arrays_&_Hashing/07_Valid_Sudoku/0036-valid-sudoku.cpp

```
/*
Problem: LeetCode 36 - Valid Sudoku

Description:
Given a 9 x 9 Sudoku board, determine if it is a valid Sudoku. The board is only partially filled, and each digit from 1 to 9 must appear exactly once in each row, column, and 3 x 3 sub-grid.

Intuition:
To check if a Sudoku board is valid, we need to verify that each digit appears exactly once in each row, column, and 3 x 3 sub-grid. We can use three separate 2D arrays to keep track of the digits already used in each row, column, and sub-grid.

Approach:
1. Initialize three 2D arrays, usedRows, usedCols, and usedSubgrids, with all values set to 0.
2. Iterate through each cell in the Sudoku board:
   - If the current cell is not empty:
     - Convert the character to an integer and subtract 1 to get the corresponding number index.
     - Calculate the sub-grid index based on the current cell's position.
      - Check if the number is already used in the current row, column, or sub-grid by looking at the corresponding index in the usedRows, usedCols, and usedSubgrids arrays.
     - If the number is already used, return false as the Sudoku board is not valid.
        - Mark the number as used in the current row, column, and sub-grid by setting the corresponding index to 1 in the usedRows, usedCols, and usedSubgrids arrays.
3. If all cells pass the checks, return true as the Sudoku board is valid.

Time Complexity:
The time complexity of this approach is O(1) since the Sudoku board has a fixed size of 9 x 9, and the iteration is constant.

Space Complexity:
The space complexity is O(1) since the arrays used for tracking the used digits (usedRows, usedCols, and usedSubgrids) have a fixed size of 9 x 9.
*/

class Solution {
  public:
    bool isValidSudoku(vector<vector<char>> &board) {
        int usedRows[9][9] = {0};
        int usedCols[9][9] = {0};
        int usedSubgrids[9][9] = {0};

        for (int i = 0; i < board.size(); ++i) {
            for (int j = 0; j < board[i].size(); ++j) {
                if (board[i][j] != '.') {
                    int num = board[i][j] - '0' - 1;
                    int subgridIndex = (i / 3) * 3 + j / 3;

                    if (usedRows[i][num] || usedCols[j][num] || usedSubgrids[subgridIndex][num]) {
                        return false;
                    }

                    usedRows[i][num] = usedCols[j][num] = usedSubgrids[subgridIndex][num] = 1;
                }
            }
        }

        return true;
    }
};
```

## 8. 01_Arrays_&_Hashing/08_Encode_and_Decode_Strings/0271-encode-and-decode-

```
/*
Problem: LeetCode 271 - Encode and Decode Strings

Description:
Design an algorithm to encode a list of strings to a string and decode the string to the original
list of strings. The encoded string should be as compact as possible.

Intuition:
To encode and decode a list of strings, we need a way to separate individual strings and handle
any special characters. One approach is to use a delimiter to separate the strings, and escape any
occurrences of the delimiter within the strings.

Approach:
Encoding:
1. Iterate through the input list of strings:
   - For each string, append its length followed by a delimiter (e.g., '#') to the encoded string.
   - Append the actual string content to the encoded string.
2. Return the encoded string.

Decoding:
1. Initialize an empty result list of strings.
2. While the encoded string is not empty:
   - Extract the length of the next string from the encoded string until the delimiter.
   - Convert the length string to an integer.
   - Extract the next substring of length characters from the encoded string.
   - Append the substring to the result list.
3. Return the result list.

Time Complexity:
The time complexity for encoding is O(n), where n is the total number of characters in all the
strings. For decoding, the time complexity is also O(n), as we need to iterate through the encoded
string to extract and reconstruct the original strings.

Space Complexity:
The space complexity is O(n), where n is the total number of characters in all the strings. This
is because we need to store the encoded string, which contains all the characters from the input
strings.
*/

class Codec {
  public:
    // Encodes a list of strings to a single string.
    string encode(vector<string> &strs) {
        string encodedStr;

        for (const string &str : strs) {
            encodedStr += to_string(str.length()) + "#" + str;
        }

        return encodedStr;
    }

    // Decodes a single string to a list of strings.
    vector<string> decode(string s) {
        vector<string> decodedStrs;
        int i = 0;

        while (i < s.length()) {
            int delimiterIndex = s.find("#", i);
            int strLength = stoi(s.substr(i, delimiterIndex - i));
```

```
            decodedStrs.push_back(s.substr(delimiterIndex + 1, strLength));
            i = delimiterIndex + strLength + 2;
        }

        return decodedStrs;
    }
};
```

## 9. 01_Arrays_&_Hashing/09_Longest_Consecutive_Sequence/0128-longest-consecut

```
/*
Problem: LeetCode 128 - Longest Consecutive Sequence

Description:
Given an unsorted array of integers nums, return the length of the longest consecutive elements sequence.

Intuition:
To find the longest consecutive sequence, we can sort the array in ascending order and then iterate through it to count the longest consecutive sequence. However, this approach would have a time complexity of O(n log n) due to the sorting operation. We can instead use a HashSet to efficiently find consecutive elements.

Approach:
1. Create a HashSet `numSet` and insert all the numbers from the input array `nums`.
2. Initialize a variable `maxLength` to store the maximum length of consecutive elements.
3. Iterate through each number `num` in the `numSet`:
   - Check if the current number `num - 1` is not present in the `numSet`. If true, it means the current number is the start of a consecutive sequence.
   - Initialize a variable `length` to 1 to count the current consecutive sequence length.
   - Iterate through consecutive numbers starting from the current number `num + 1` until a number is not present in the `numSet`, incrementing the `length` accordingly.
   - Update `maxLength` with the maximum value between `maxLength` and `length`.
4. Return `maxLength`, which represents the length of the longest consecutive sequence.

Time Complexity:
The time complexity of this approach is O(n), where n is the number of elements in the input array. It is determined by the iteration through the `numSet`, which contains all the unique numbers.

Space Complexity:
The space complexity is O(n) as we need to store all the numbers from the input array in the `numSet`.
*/

class Solution {
  public:
    int longestConsecutive(vector<int> &nums) {
        unordered_set<int> numSet(nums.begin(), nums.end());
        int maxLength = 0;

        for (int num : numSet) {
            // Check if the current number is the start of a consecutive sequence
            if (numSet.count(num - 1) == 0) {
                int length = 1;

                // Iterate through consecutive numbers starting from the current number
                while (numSet.count(num + length) != 0) {
                    length++;
                }

                maxLength = max(maxLength, length);
            }
        }

        return maxLength;
    }
};

// auto init = [](){
```

```
//     ios::sync_with_stdio(false);
//     cin.tie(0);
//     cout.tie(0);
//     return 0;
// }();

// class Solution {
// public:
//     int longestConsecutive(vector<int>& nums) {
//         if(nums.empty())
//             return 0;
//         else if(nums.size() == 1)
//             return 1;

//         sort(nums.begin(),nums.end());
//         int ans = 1;
//         int cnt = 1;

//         for(int i = 0; i < nums.size(); i++) {
//             if(i > 0 && nums[i-1]+1 == nums[i]) {
//                 ++cnt;
//                 ans = max(ans, cnt);
//             }
//             else if(i > 0 && nums[i-1] == nums[i])
//                 continue;
//             else
//                 cnt = 1;
//         }
//         return ans;
//     }
// };

/*
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        if(nums.size() == 0)
            return 0;

        // Using set because its sorted and takes O(log N) time
        set<int> S;
        int Max = 0, local_Max = 0;

        for(int i = 0; i < nums.size(); i++)
            S.insert(nums[i]);

        vector<int> V(S.begin(), S.end());

        for(int i = 1; i < V.size(); i++) {
            if(V[i] - V[i-1] == 1) {
                local_Max++;
                if(Max < local_Max)
                    Max = local_Max;
            }
            else
                local_Max = 0;
        }
        return Max + 1;
    }
};
*/
```

## 10. 02_Two_Pointers/01_Valid_Palindrome/0125-valid-palindrome.cpp

```
/*
Problem: LeetCode 125 - Valid Palindrome

Description:
Given a string s, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

Intuition:
To check if a string is a palindrome, we need to compare characters from both ends of the string. We can ignore non-alphanumeric characters and treat uppercase and lowercase letters as the same.

Approach:
1. Initialize two pointers, `left` pointing to the start of the string, and `right` pointing to the end of the string.
2. Iterate while `left` is less than `right`:
     - Skip non-alphanumeric characters by incrementing `left` or decrementing `right` if the character at that position is not alphanumeric.
   - Convert both characters to lowercase and compare them. If they are not equal, return false.
   - Increment `left` and decrement `right` to move to the next pair of characters.
3. If the loop completes without finding any non-matching characters, return true.

Time Complexity:
The time complexity is O(n), where n is the length of the input string. We iterate through the string once to check if it is a valid palindrome.

Space Complexity:
The space complexity is O(1) since we are using a constant amount of space to store the pointers and perform the comparison.
*/

class Solution {
  public:
    bool isPalindrome(string s) {
        int left = 0, right = s.length() - 1;

        while (left < right) {
            // isalnum -> Is alphabet or number
            if (!isalnum(s[left])) {
                left++;
                continue;
            }

            if (!isalnum(s[right])) {
                right--;
                continue;
            }

            if (tolower(s[left]) != tolower(s[right])) {
                return false;
            }

            left++;
            right--;
        }

        return true;
    }
};
```

## 11. 02_Two_Pointers/02_Two_Sum_II_-_Input_Array_Is_Sorted/0167-two-sum-ii-input

```
/*
Problem: LeetCode 167 - Two Sum II - Input array is sorted

Description:
Given an array of integers numbers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number.
Return the indices of the two numbers (1-indexed) as an integer array answer of size 2, where 1 <= answer[0] < answer[1] <= numbers.length.
You may assume that each input would have exactly one solution and you may not use the same element twice.

Intuition:
Since the input array is sorted, we can use a two-pointer approach to find the two numbers that sum up to the target. By maintaining two pointers, one pointing to the start of the array and the other pointing to the end, we can narrow down the search space based on the comparison of the current sum with the target.

Approach:
1. Initialize two pointers, `left` pointing to the start of the array (index 0) and `right` pointing to the end of the array.
2. Iterate while `left` is less than `right`:
   - Calculate the current sum of the elements at `left` and `right`.
    - If the sum is equal to the target, return the indices (`left + 1` and `right + 1`) as the answer.
   - If the sum is less than the target, increment `left` to consider a larger element.
   - If the sum is greater than the target, decrement `right` to consider a smaller element.
3. If no solution is found, return an empty vector.

Time Complexity:
The time complexity is O(n), where n is the number of elements in the input array. In the worst case, we iterate through the entire array once.

Space Complexity:
The space complexity is O(1) as we are using a constant amount of space to store the pointers and perform the comparison.
*/

class Solution {
  public:
    vector<int> twoSum(vector<int> &numbers, int target) {
        int left = 0, right = numbers.size() - 1;

        while (left < right) {
            int sum = numbers[left] + numbers[right];

            if (sum == target) {
                return {left + 1, right + 1};
            } else if (sum < target) {
                left++;
            } else {
                right--;
            }
        }

        return {};
    }
};
```

## 12. 02_Two_Pointers/03_3Sum/0015-3sum.cpp

```
/*
Problem: LeetCode 15 - 3Sum

Description:
Given an array nums of n integers, find all unique triplets in the array which gives the sum of zero.
The solution set must not contain duplicate triplets.

Intuition:
To find unique triplets that sum up to zero, we can leverage the two-pointer approach. By sorting the array and iterating through each element, we can convert the problem into finding two numbers that sum up to the negation of the current element.

Approach:
1. Sort the input array `nums` in non-decreasing order.
2. Iterate through each element `nums[i]` from 0 to n-3 (exclusive):
    - If `i > 0` and `nums[i]` is equal to `nums[i-1]`, skip the current iteration to avoid duplicate triplets.
    - Initialize two pointers, `left` pointing to the element next to `nums[i]`, and `right` pointing to the last element of the array.
   - Iterate while `left` is less than `right`:
     - Calculate the current sum of the elements `nums[i]`, `nums[left]`, and `nums[right]`.
      - If the sum is equal to zero, add the triplet `[nums[i], nums[left], nums[right]]` to the result.
       - Increment `left` and decrement `right` to search for the next pair of elements.
       - Skip any duplicate values of `nums[left]` and `nums[right]` to avoid duplicate triplets.
     - If the sum is less than zero, increment `left`.
     - If the sum is greater than zero, decrement `right`.
3. Return the result, which contains all unique triplets that sum up to zero.

Time Complexity:
The time complexity is O(n^2), where n is the number of elements in the input array. Sorting the array takes O(nlogn) time, and iterating through the array with two pointers takes O(n^2) time.

Space Complexity:
The space complexity is O(1) as we are using a constant amount of space to store the pointers and the result.
*/

class Solution {
  public:
    vector<vector<int>> threeSum(vector<int> &nums) {
        vector<vector<int>> result;
        int n = nums.size();
        sort(nums.begin(), nums.end());

        for (int i = 0; i < n - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }

            int left = i + 1, right = n - 1;

            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];

                if (sum == 0) {
                    result.push_back({nums[i], nums[left], nums[right]});
                    left++;
                    right--;
```

```
                while (left < right && nums[left] == nums[left - 1]) {
                    left++;
                }

                while (left < right && nums[right] == nums[right + 1]) {
                    right--;
                }
            } else if (sum < 0) {
                left++;
            } else {
                right--;
            }
        }
    }

    return result;
  }
};
```

## 13. 02_Two_Pointers/04_Container_With_Most_Water/0011-container-with-most-wate

```
/*
Problem: LeetCode 11 - Container With Most Water

Description:
Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai).
n vertical lines are drawn such that the two endpoints of the line i is at (i, ai) and (i, 0).
Find two lines, which, together with the x-axis, forms a container, such that the container contains the most water.

Intuition:
To maximize the container's area, we need to find two vertical lines that enclose the most water.
The area is determined by the height of the shorter line and the distance between the lines.
We can start with the maximum width and move the pointers inward, always choosing the next height that is greater than the current one.

Approach:
1. Initialize two pointers, `left` pointing to the start of the array (index 0), and `right` pointing to the end of the array.
2. Initialize a variable `maxArea` to store the maximum container area.
3. Iterate while `left` is less than `right`:
     - Calculate the current container area as the minimum height between `height[left]` and `height[right]` multiplied by the width (`right - left`).
   - Update `maxArea` with the maximum value between `maxArea` and the current area.
     - Move the pointer with the smaller height inward, as moving the pointer with the greater height does not increase the area.
4. Return `maxArea`, which represents the maximum container area.

Time Complexity:
The time complexity is O(n), where n is the number of elements in the input array.
We only need to iterate through the array once from both ends.

Space Complexity:
The space complexity is O(1) as we are using a constant amount of space to store the pointers and the maximum area.
*/

class Solution {
  public:
    int maxArea(vector<int> &height) {
        int left = 0, right = height.size() - 1;
        int maxArea = 0;

        while (left < right) {
            int currArea = min(height[left], height[right]) * (right - left);
            maxArea = max(maxArea, currArea);

            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return maxArea;
    }
};

// class Solution {
// public:
```

```
//     int maxArea(vector<int>& height) {
//         int leftPointer = 0, rightPointer = height.size() - 1;
//         int result = -1, currResult = 0;

//         while(leftPointer < rightPointer){
//             int dist = rightPointer - leftPointer;
//             currResult = min(height[leftPointer], height[rightPointer]) * dist;

//             result = max(result, currResult);

//             if( height[leftPointer] > height[rightPointer] )
//                 rightPointer--;
//             else
//                 leftPointer++;
//         }
//         return result;
//     }
// };

/*
class Solution {
public:
    int maxArea(vector<int>& height) {
        int water = 0;
        int i = 0, j = height.size() - 1;
        while (i < j) {
            int h = min(height[i], height[j]);
            water = max(water, (j - i) * h);
            while(height[i] <= h && i < j)
                i++;
            while(height[j] <= h && i < j)
                j--;
        }
        return water;
    }
};
*/
```

## 14. 02_Two_Pointers/05_Trapping_Rain_Water/0042-trapping-rain-water.cpp

```
/*
Problem: LeetCode 42 - Trapping Rain Water

Description:
Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Intuition:
To determine the amount of water that can be trapped, we need to consider the height of each bar and the width between the bars.
The amount of water trapped at a particular position depends on the minimum height of the tallest bars on its left and right sides minus the elevation.

Approach:
1. Initialize two pointers, `left` pointing to the start of the array (index 0), and `right` pointing to the end of the array.
2. Initialize variables `leftMax` and `rightMax` to keep track of the maximum heights encountered on the left and right sides.
3. Initialize a variable `water` to store the total trapped water.
4. Iterate while `left` is less than `right`:
   - If the height at `left` is less than or equal to the height at `right`:
     - Update `leftMax` with the maximum value between `leftMax` and the current height at `left`.
      - Calculate the amount of water that can be trapped at `left` by subtracting the height at `left` from `leftMax`.
     - Add the calculated water to the total `water` variable.
     - Increment `left`.
   - If the height at `left` is greater than the height at `right`:
       - Update `rightMax` with the maximum value between `rightMax` and the current height at `right`.
      - Calculate the amount of water that can be trapped at `right` by subtracting the height at `right` from `rightMax`.
     - Add the calculated water to the total `water` variable.
     - Decrement `right`.
5. Return the total trapped `water`.

Time Complexity:
The time complexity is O(n), where n is the number of elements in the input array. We iterate through the array once from both ends.

Space Complexity:
The space complexity is O(1) as we are using a constant amount of space to store the pointers and variables.
*/

class Solution {
  public:
    int trap(vector<int> &height) {
        int left = 0, right = height.size() - 1;
        int leftMax = 0, rightMax = 0;
        int water = 0;

        while (left < right) {
            if (height[left] <= height[right]) {
                leftMax = max(leftMax, height[left]);
                water += leftMax - height[left];
                left++;
            } else {
                rightMax = max(rightMax, height[right]);
                water += rightMax - height[right];
                right--;
```

```
            }
        }

        return water;
    }
};

/*
Solution -
The key is we can calculate the amount of water at any given index by
-> taking minimum of (max of left nd right so far)
-> that will give us water in between them
-> then subtract the height to remove the elevation

SO make two array to store max height so far from left and right
and then just calculate, min(maxleft, maxright) - height[i];
*/

// class Solution {
// public:
//     int trap(vector<int>& height) {
//         int heightSize = height.size();
//         vector<int> leftMax(heightSize), rightMax(heightSize);
//         int result = 0;

//         // Filling the left and right max arrays
//         for(int i = 1, j = heightSize - 2; i < heightSize, j >= 0; i++, j--) {
//             if(leftMax[i-1] <= height[i-1])
//                 leftMax[i] = height[i-1];
//             else
//                 leftMax[i] = leftMax[i-1];

//             if(rightMax[j+1] <= height[j+1])
//                 rightMax[j] = height[j+1];
//             else
//                 rightMax[j] = rightMax[j+1];
//         }

//         for(int i = 0; i < heightSize; i++) {
//             int temp = min(leftMax[i], rightMax[i]) - height[i];
//             if(temp < 0)
//                 temp = 0;

//             result += temp;
//         }
//         return result;
//     }
// };

/*
class Solution {
public:
    int trap(vector<int>& H) {

        int n = H.size(), mx = 0, ans = 0;
        int idx = max_element(begin(H), end(H)) - begin(H);

        for(int i = 0; i < idx; i++) {
            ans += max(0, mx - H[i]);
            mx = max(mx, H[i]);
        }
        mx = 0;
        for(int i = n - 1; i > idx; i--) {
            ans += max(0, mx - H[i]);
```

```
            mx = max(mx, H[i]);
        }

        return ans;
    }
};
*/
```

## 15. 03_Sliding_Window/01_Best_Time_to_Buy_and_Sell_Stock/0121-best-time-to-buy

```
/*
Problem: LeetCode 121 - Best Time to Buy and Sell Stock

Description:
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.

Intuition:
To maximize the profit, we need to find the largest difference between any two prices, where the lower price comes before the higher price.
We can track the minimum price seen so far and calculate the maximum profit by subtracting the minimum price from each subsequent price.

Approach:
1. Initialize variables `minPrice` and `maxProfit`.
   - Set `minPrice` to the maximum possible value and `maxProfit` to 0.
2. Iterate through each price in the `prices` array:
   - Update `minPrice` to the minimum value between `minPrice` and the current price.
   - Calculate the potential profit by subtracting `minPrice` from the current price.
   - Update `maxProfit` to the maximum value between `maxProfit` and the potential profit.
3. Return `maxProfit`, which represents the maximum profit achievable.

Time Complexity:
The time complexity is O(n), where n is the number of elements in the input array. We iterate through the array once to calculate the maximum profit.

Space Complexity:
The space complexity is O(1) as we are using a constant amount of space to store the variables.
*/

class Solution {
  public:
    int maxProfit(vector<int> &prices) {
        int minPrice = INT_MAX;
        int maxProfit = 0;

        for (int price : prices) {
            minPrice = min(minPrice, price);
            maxProfit = max(maxProfit, price - minPrice);
        }

        return maxProfit;
    }
};

//   -> Calculate LeastSoFar
//   -> Measure profit, if greater than result, update result

// class Solution {
// public:
//     int maxProfit(vector<int>& prices) {
//         int leastSoFar = 100000, profit = 0, result = 0;

//         for(int i = 0; i < prices.size(); i++) {
//             if(prices[i] < leastSoFar)
//                 leastSoFar = prices[i];
```

```
//             profit = prices[i] - leastSoFar;
//             if(result < profit)
//                 result = profit;
//         }
//         return result;
//     }
// };

/*
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        ios_base::sync_with_stdio(0);
        cout.tie(0);

        int profit = 0;
        int minbuy = prices[0];

        for(int x: prices) {
            int diff = x - minbuy;
            profit = max(profit, diff);
            minbuy = min(minbuy, x);
        }

        return profit;
    }
};
*/
```

## 16. 03_Sliding_Window/02_Longest_Substring_Without_Repeating_Characters/0003-

```
/*
Problem: LeetCode 3 - Longest Substring Without Repeating Characters

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

Intuition:
To find the longest substring without repeating characters, we can use a sliding window approach.
We can maintain a window that contains unique characters and keep expanding it until we encounter
a repeating character.
At each step, we update the maximum length of the non-repeating substring.

Approach:
1. Initialize variables `left` and `right` to represent the left and right pointers of the sliding
window.
   - Set both `left` and `right` to 0 initially.
2. Initialize a variable `maxLen` to store the maximum length of the non-repeating substring.
3. Initialize a set or map to keep track of the unique characters in the window.
4. Iterate while `right` is less than the length of the string:
   - Check if the character at `right` is already present in the set/map:
     - If it is present, remove the character at `left` from the set/map and increment `left`.
      - If it is not present, add the character at `right` to the set/map, calculate the current
length of the non-repeating substring as `right - left + 1`, and update `maxLen` if necessary.
   - Increment `right` to expand the window.
5. Return `maxLen`, which represents the length of the longest substring without repeating
characters.

Time Complexity:
The time complexity is O(n), where n is the length of the input string. We iterate through the
string once with the sliding window approach.

Space Complexity:
The space complexity is O(min(n, m)), where n is the length of the input string and m is the size
of the character set.
In the worst case, the character set can be as large as the input string, but it can also be
limited by the number of distinct characters.
*/

class Solution {
  public:
    int lengthOfLongestSubstring(string s) {
        int left = 0, right = 0;
        int maxLen = 0;
        unordered_set<char> charSet;

        while (right < s.length()) {
            if (charSet.count(s[right])) {
                charSet.erase(s[left]);
                left++;
            } else {
                charSet.insert(s[right]);
                maxLen = max(maxLen, right - left + 1);
                right++;
            }
        }

        return maxLen;
    }
};
```

```
/*
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int n = s.length();
        int ans = 0;
        vector<int> M(256, -1);
        int start = 0, end = 0;
        while(end < n) {
            if(M[s[end]] != -1) {
                start = max(start, M[s[end]] + 1);
            }
            M[s[end]] = end;
            ans = max(ans, end-start + 1);
            end++;
        }
        return ans;
    }
};
*/
```

## 17. 03_Sliding_Window/03_Longest_Repeating_Character_Replacement/0424-longes

```
/*
Problem: LeetCode 424 - Longest Repeating Character Replacement

Description:
Given a string `s` that consists of only uppercase English letters, you can perform at most `k` operations on that string.
In one operation, you can choose any character of the string and change it to any other uppercase English letter.
Find the length of the longest substring containing repeating letters you can get after performing the above operations.

Intuition:
To find the longest substring with repeating characters, we can use a sliding window approach.
We can maintain a window that contains the most frequent character and update it as we expand the window.
The maximum length of the substring with repeating characters can be obtained by considering the count of the most frequent character within the window.

Approach:
1. Initialize variables `maxCount` and `maxLength` to track the maximum count of a character and the maximum length of the substring with repeating characters.
2. Initialize a vector `count` to keep track of the count of each character.
3. Initialize variables `left` and `right` to represent the left and right pointers of the sliding window.
   - Set both `left` and `right` to 0 initially.
4. Iterate while `right` is less than the length of the string:
   - Increment the count of the character at `right` in the `count` vector.
   - Update `maxCount` with the maximum value between `maxCount` and the current count.
   - Calculate the length of the current substring as `right - left + 1`.
   - If the length of the current substring minus `maxCount` is greater than `k`:
     - Decrement the count of the character at `left` in the `count` vector.
     - Increment `left` to shrink the window.
    - Update `maxLength` with the maximum value between `maxLength` and the current substring length.
   - Increment `right` to expand the window.
5. Return `maxLength`, which represents the length of the longest substring with repeating characters that can be obtained.

Time Complexity:
The time complexity is O(n), where n is the length of the input string. We iterate through the string once with the sliding window approach.

Space Complexity:
The space complexity is O(1) as we are using a fixed-size vector `count` to store the count of characters.
*/

class Solution {
  public:
    int characterReplacement(string s, int k) {
        vector<int> count(26, 0);
        int left = 0, right = 0;
        int maxCount = 0, maxLength = 0;

        while (right < s.length()) {
            count[s[right] - 'A']++;
            maxCount = max(maxCount, count[s[right] - 'A']);

            if ((right - left + 1) - maxCount > k) {
                count[s[left] - 'A']--;
```

```
                left++;
            }

            maxLength = max(maxLength, right - left + 1);
            right++;
        }

        return maxLength;
    }
};


//   -> Have two pointer L and R start at 0
//   -> Keep incrementing R until size of the window(R-L) minus
//      the most frequent element is lesser than or equal to k.
//      i.e Window Size - Most Frequent Element count <= K
//      Because K is the max number of changes we can make
//   -> If Window Size - Most Frequent Element count > K, then increment L

// class Solution {
// public:
//     int characterReplacement(string s, int k) {
//         int L = 0, R = 0;
//         int maxFreq = 0, maxLength = 0;
//         unordered_map<char, int> M; //For storing chars and their frequency

//         while(R < s.size()) {
//             // int winLen = R - L + 1; // Sliding window length
//             M[s[R]]++;      // Updating frequency in Map
//             maxFreq = max(maxFreq, M[s[R]]); // if curr freq is greater than maxFreq, update
//             if(R-L+1 - maxFreq > k) {
//                  M[s[L]]--;      // Decrementing count of L'th element because we're gonna move
L
//                 L++;
//             }
//             // Technically we will never need to reduce the maxFreq,
//             // because answer will be set around around it
//             // We can adjust the length, so basically the ans is maxFreq + K
//             maxLength = max(maxLength, R-L+1);
//             R++;                //Moving R
//         }
//         return maxLength;
//     }
// };

/*
class Solution {
public:
    int characterReplacement(string s, int k) {
        if(s.size() == 1)
            return 1;
        int i{}, j{}, res{}, maxCount{0};
        vector<int> a(26, 0);
        while(j < s.size()) {
            a[s[j] - 'A']++;
            if(maxCount < a[s[j]-'A'])
                maxCount = a[s[j]-'A'];
            if((j - i + 1) - maxCount > k){
                a[s[i]-'A']--;
                i++;
            }
            res = max(res, j-i+1);
            j++;
        }
```

```
    return res;
    }
};
*/
```

## 18. 03_Sliding_Window/04_Permutation_in_String/0567-permutation-in-string.cpp

```
/*
Problem: LeetCode 567 - Permutation in String

Description:
Given two strings `s1` and `s2`, return `true` if `s2` contains the permutation of `s1`.
In other words, one of the first string's permutations is the substring of the second string.

Intuition:
To check if `s2` contains a permutation of `s1`, we can use a sliding window approach.
- We initialize two frequency maps, `freq1` and `freq2`, to store the frequencies of characters in `s1` and the sliding window of `s2`.
- If the frequencies in `freq1` and `freq2` are equal, then it means `s2` contains a permutation of `s1`.
- By sliding the window through `s2`, we can keep track of the frequencies of characters in the current window and compare them with the frequencies in `freq1`.

Approach:
1. We start by checking if the length of `s1` is greater than `s2`. If it is, then it's not possible for `s2` to contain a permutation of `s1`, so we return `false`.
2. We initialize two arrays, `freq1` and `freq2`, to store the frequencies of characters in `s1` and the sliding window of `s2`.
   - Both arrays should have a size of 26 to represent the lowercase English letters.
3. We iterate through the first `s1.size()` characters of `s2` to initialize the frequencies in both arrays.
4. We initialize two pointers, `L` and `R`, representing the left and right ends of the window, respectively.
5. If the frequencies in `freq1` and `freq2` are equal, we return `true` because `s2` contains a permutation of `s1`.
6. We iterate from index `s1.size()` to `s2.size()` using the sliding window approach:
    - If the frequency of the character at index `L` in `freq2` is 1, we remove it from `freq2` since we will increment `L` later.
   - Otherwise, we decrement the frequency of the character at index `L` in `freq2` since we will increment `L`.
   - We increment the frequency of the character at index `R` in `freq2` since we have added a new character to the window.
   - We increment `L` to slide the window to the right.
   - If the frequencies in `freq1` and `freq2` are equal, we return `true`.
7. If no permutation is found after iterating through `s2`, we return `false`.

Time Complexity:
The time complexity is O(n), where n is the length of `s2`. We iterate through `s2` once using the sliding window approach.

Space Complexity:
The space complexity is O(1) as both `freq1` and `freq2` have a fixed size of 26, representing lowercase English letters.
*/

class Solution {
  public:
    bool checkInclusion(string s1, string s2) {
        if (s1.size() > s2.size()) {
            return false;
        }

        vector<int> freq1(26, 0);
        vector<int> freq2(26, 0);

        for (int i = 0; i < s1.size(); i++) {
            freq1[s1[i] - 'a']++;
```

```
            freq2[s2[i] - 'a']++;
        }

        int L = 0;

        if (freq1 == freq2) {
            return true;
        }

        for (int R = s1.size(); R < s2.size(); R++) {
            if (freq2[s2[L] - 'a'] == 1) {
                freq2[s2[L] - 'a'] = 0;
            } else {
                freq2[s2[L] - 'a']--;
            }

            freq2[s2[R] - 'a']++;
            L++;

            if (freq1 == freq2) {
                return true;
            }
        }

        return false;
    }
};
```

## 19. 03_Sliding_Window/05_Minimum_Window_Substring/0076-minimum-window-sub

```
/*
Problem: LeetCode 76 - Minimum Window Substring

Description:
Given two strings `s` and `t`, return the minimum window in `s` that contains all the characters of `t`.
If there is no such window in `s` that covers all characters in `t`, return an empty string.

Intuition:
To find the minimum window substring, we can use the sliding window technique.
The idea is to maintain two pointers, `left` and `right`, to create a window in `s`.
By expanding and contracting the window, we can find the minimum window substring that contains all the characters of `t`.

Approach:
1. We start by initializing two pointers, `left` and `right`, to the first index of `s`.
2. We initialize two frequency maps, `targetFreq` and `windowFreq`, to store the frequencies of characters in `t` and the current window of `s`, respectively.
3. We initialize variables, `count` and `minLen`, to keep track of the count of characters in `t` that are present in the current window and the minimum window length found so far.
4. We iterate through `s` using the `right` pointer:
   - Increment the frequency of the character at `s[right]` in `windowFreq`.
   - If the frequency of the character at `s[right]` in `windowFreq` is less than or equal to the frequency of the same character in `targetFreq`, increment the `count`.
   - If `count` is equal to the length of `t`, it means we have found a valid window that contains all the characters of `t`.
     - Update `minLen` if the current window length is smaller.
     - Move the `left` pointer to contract the window:
       - Decrement the frequency of the character at `s[left]` in `windowFreq`.
         - If the frequency of the character at `s[left]` in `windowFreq` becomes less than the frequency in `targetFreq`, decrement the `count`.
     - Move the `left` pointer to the right to search for the next valid window.
5. Repeat steps 4 and 5 until the `right` pointer reaches the end of `s`.
6. Return the minimum window substring found based on `minLen`. If no valid window is found, return an empty string.

Time Complexity:
The time complexity is O(n), where n is the length of `s`. We iterate through `s` once using the sliding window approach.

Space Complexity:
The space complexity is O(1) as both `targetFreq` and `windowFreq` have a fixed size of 128 to represent ASCII characters.
*/

class Solution {
  public:
    string minWindow(string s, string t) {
        vector<int> targetFreq(128, 0);
        vector<int> windowFreq(128, 0);

        for (char ch : t) {
            targetFreq[ch]++;
        }

        int left = 0;
        int right = 0;
        int count = 0;
        int minLen = INT_MAX;
        int minStart = 0;
```

```
        while (right < s.length()) {
            windowFreq[s[right]]++;

            if (windowFreq[s[right]] <= targetFreq[s[right]]) {
                count++;
            }

            while (count == t.length()) {
                if (right - left + 1 < minLen) {
                    minLen = right - left + 1;
                    minStart = left;
                }

                windowFreq[s[left]]--;

                if (windowFreq[s[left]] < targetFreq[s[left]]) {
                    count--;
                }

                left++;
            }

            right++;
        }

        return (minLen == INT_MAX) ? "" : s.substr(minStart, minLen);
    }
};

// class Solution {
// public:
//     string minWindow(string s, string t) {
//         if(t.length() > s.length())
//             return "";

//         unordered_map<char, int> M;
//         for(int i = 0; i < t.size(); i++)
//             M[t[i]]++;

//         int L = 0, R = 0;
//         int count = t.size();  // No. of chars in t that need to be n S
//         int minLength = INT_MAX;
//           int minStart = 0; // We have to return string, so this variable is for storing teh
starting index

//         while(R < s.size()) {
//             if(M[s[R]] > 0)     // If we find an element from T(which we put in M)
//                 count--;

//             // We will decrease the value here also, since count only counts unique elements
//             // Elements in M will decrease till 0(which then decreases count)
//             // and elements that dont exist in M will be -ve
//             M[s[R]]--;
//             R++;
//             while(count == 0) {
//                 if(R-L < minLength) {
//                     minLength = R-L;
//                     minStart = L;
//                 }
//                 M[s[L]]++; // Updating the value in Map
//                 // If the current L element is from T, then increase count
//                 if(M[s[L]] > 0) {
//                     count++;
```

```
//                 }
//                 L++;
//             }
//         }
//         // If the ans is empty string, minLength(= MAX_INT) will not be changed
//         if (minLength != INT_MAX) {
//             return s.substr(minStart, minLength);
//         }
//         return "";
//     }
// };

/*
class Solution {
public:
    string minWindow(string s, string t) {
        int cnt[128] = {}, diff = t.size();
        for(int i = 0; i < t.size(); i++)
            cnt[t[i]]++;
        int left = 0, right = 0, idx = 0, size = 2e5;
        while(right < s.size()) {
            if(cnt[s[right++]]-- > 0)
                diff--;
            while(diff == 0) {
                if(right-left < size)
                    size = right-(idx = left);
                if(cnt[s[left++]]++ == 0)
                    diff++;
            }
        }
        return (size == 2e5) ? "" : s.substr(idx, size);
    }
};
*/
```

## 20. 03_Sliding_Window/06_Sliding_Window_Maximum/0239-sliding-window-maximum

```
/*
Problem: LeetCode 239 - Sliding Window Maximum

Description:
Given an array of integers `nums` and an integer `k`, return the maximum sliding window of size `k` in the array.

Intuition:
To find the maximum sliding window of size `k`, we can use a sliding window and a deque (double-ended queue).
The deque will store the indices of elements in decreasing order of their values.
By maintaining this order, the maximum element in the window will always be at the front of the deque.

Approach:
1. We start by initializing an empty deque to store the indices of elements.
2. We iterate through the array `nums` from left to right:
   - Remove indices from the back of the deque if the corresponding elements are smaller than the current element.
   - Add the current element's index to the back of the deque.
   - If the index at the front of the deque is outside the current window, remove it.
   - If the current index is greater than or equal to `k - 1` (i.e., the window size), it means we have processed `k` elements and can start collecting the maximum values.
       - Add the maximum value (which is the element at the front of the deque) to the result vector.
3. Return the result vector.

Time Complexity:
The time complexity is O(n), where n is the length of the input array `nums`. We iterate through `nums` once.

Space Complexity:
The space complexity is O(k), where k is the size of the sliding window. The deque stores at most `k` indices.
*/

class Solution {
  public:
    vector<int> maxSlidingWindow(vector<int> &nums, int k) {
        vector<int> result;
        deque<int> indices;

        for (int i = 0; i < nums.size(); i++) {
            // Remove indices from the back of the deque if the corresponding elements are smaller than the current element
            while (!indices.empty() && nums[indices.back()] < nums[i]) {
                indices.pop_back();
            }

            indices.push_back(i);

            // Remove the index at the front of the deque if it is outside the current window
            if (indices.front() <= i - k) {
                indices.pop_front();
            }

            // Add the maximum value to the result if the current index is in the window
            if (i >= k - 1) {
                result.push_back(nums[indices.front()]);
            }
```

```
        }

        return result;
    }
};

/*
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        int offset = 1e4;
        std::ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0);
        int mp[(int)(2 * 1e4 + 1)] = {0};
        int maxval = INT_MIN;

        for(int i = 0; i < k; i++) {
            mp[offset + nums[i]]++;
            maxval = max(maxval, nums[i]);
        }

        vector<int> ans;
        ans.push_back(maxval);

        for(int j = k; j < nums.size(); j++) {
            mp[offset + nums[j - k]]--;
            mp[offset + nums[j]]++;
            maxval = max(maxval, nums[j]);
            int mpos = maxval + offset;
            while(mp[mpos] == 0) {
                mpos--;
            }
            maxval = mpos - offset;
            ans.push_back(maxval);
        }

        return ans;

    }
};
*/
```

## 21. 04_Stack/01_Valid_Parentheses/0020-valid-parentheses.cpp

```
/*
Problem: LeetCode 20 - Valid Parentheses

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

Intuition:
To determine if a string of parentheses is valid, we can utilize a stack data structure. We iterate through the characters of the string and perform the following steps:
- If we encounter a closing bracket, we check if the stack is empty or if the top of the stack does not correspond to the current closing bracket. If either condition is true, the string is not valid.
- If we encounter an opening bracket, we push it onto the stack.
After iterating through all the characters, if the stack is empty, the string is valid; otherwise, it is not.

Approach:
1. Initialize an empty stack to store opening brackets.
2. Iterate through each character in the string `s`:
    - If the current character is a closing bracket, check if the stack is empty or if the top of the stack does not match the current closing bracket. Return false if either condition is true.
   - If the current character is an opening bracket, push it onto the stack.
3. After iterating through all characters, check if the stack is empty:
   - If the stack is empty, return true, indicating that all brackets have been matched.
   - If the stack is not empty, return false, indicating unmatched brackets.
4. The function returns true if all brackets are matched; otherwise, it returns false.

Time Complexity:
The time complexity is O(n), where n is the length of the input string `s`. We iterate through each character once.

Space Complexity:
The space complexity is O(n), where n is the length of the input string `s`. In the worst case, the stack may store all opening brackets.
*/

class Solution {
  public:
    bool isValid(string s) {
        stack<char> bracketStack;

        for (char ch : s) {
            if (ch == ')' || ch == '}' || ch == ']') {
                if (bracketStack.empty() || !isMatchingPair(bracketStack.top(), ch)) {
                    return false;
                }

                bracketStack.pop();
            } else {
                bracketStack.push(ch);
            }
        }

        return bracketStack.empty();
    }

  private:
    bool isMatchingPair(char opening, char closing) {
        return (opening == '(' && closing == ')') ||
```

```
            (opening == '{' && closing == '}') ||
            (opening == '[' && closing == ']');
    }
};

// class Solution {
// public:
//     bool isValid(string s) {
//         stack<char> stag;
//         for(int i = 0; i < s.size(); i++) {
//             if(s[i] == ')' || s[i] == '}' || s[i] == ']') {
//                 if(stag.empty())
//                     return false;
//                 if(s[i] == ')' && stag.top() != '(')
//                     return false;
//                 if(s[i] == '}' && stag.top() != '{')
//                     return false;
//                 if(s[i] == ']' && stag.top() != '[')
//                     return false;
//                 stag.pop();
//             }
//             else {
//                 stag.push(s[i]);
//             }
//         }

//         if (!stag.empty()) {
//             return false;
//         }

//         return true;
//     }
// };
```

## 22. 04_Stack/02_Min_Stack/0155-min-stack.cpp

```
/*
Problem: LeetCode 155 - Min Stack

Description:
Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Intuition:
To efficiently retrieve the minimum element in constant time, we need to keep track of the minimum value at each step. We can achieve this by using an additional stack to store the minimum values.

Approach:
1. Initialize two stacks: a main stack to store the actual values and a minimum stack to store the minimum values.
2. When pushing a value, check if it is smaller than or equal to the top of the minimum stack. If it is, push the value onto both stacks.
   - If it is larger, push the value only onto the main stack.
3. When popping a value, check if the top of the main stack is equal to the top of the minimum stack.
   - If they are equal, pop both values from both stacks.
   - If they are not equal, pop only from the main stack.
4. When retrieving the top element, simply return the top of the main stack.
5. When retrieving the minimum element, simply return the top of the minimum stack.
6. The implementation ensures that the minimum stack always has the minimum value at the top, reflecting the minimum value at each step.

Time Complexity:
All operations - push, pop, top, and getMin - have a time complexity of O(1). They are all performed in constant time.

Space Complexity:
The space complexity is O(n), where n is the number of elements pushed into the stack. The two stacks store the same number of elements as the main stack.
*/

class MinStack {
  private:
    std::stack<int> mainStack;
    std::stack<int> minStack;

  public:
    MinStack() {
    }

    void push(int val) {
        mainStack.push(val);

        if (minStack.empty() || val <= minStack.top()) {
            minStack.push(val);
        }
    }

    void pop() {
        if (mainStack.top() == minStack.top()) {
            minStack.pop();
        }

        mainStack.pop();
    }

    int top() {
```

```
        return mainStack.top();
    }

    int getMin() {
        return minStack.top();
    }
};
```

## 23. 04_Stack/03_Evaluate_Reverse_Polish_Notation/0150-evaluate-reverse-polish-no

```
/*
Problem: LeetCode 150 - Evaluate Reverse Polish Notation

Description:
Evaluate the value of an arithmetic expression in Reverse Polish Notation (RPN).
Valid operators are +, -, *, and /. Each operand may be an integer or another expression.

Intuition:
Reverse Polish Notation (RPN) eliminates the need for parentheses in arithmetic expressions by using a postfix notation. To evaluate an RPN expression, we can utilize a stack to perform the necessary operations.

Approach:
1. Initialize an empty stack.
2. Iterate through each token in the given list:
    - If the token is an operator ('+', '-', '*', '/'), pop the top two values from the stack, perform the corresponding operation, and push the result back to the stack.
   - If the token is an operand (integer), convert it to an integer and push it onto the stack.
3. After iterating through all tokens, the stack will contain the final result.
4. Pop the result from the stack and return it.

Time Complexity:
The time complexity is O(n), where n is the number of tokens in the input list. We iterate through each token once.

Space Complexity:
The space complexity is O(n), where n is the number of tokens in the input list. In the worst case, the stack may store all the operands.
*/

class Solution {
  public:
    int evalRPN(std::vector<std::string> &tokens) {
        std::stack<int> stack;

        for (const auto &token : tokens) {
            if (isOperator(token)) {
                int operand2 = stack.top();
                stack.pop();
                int operand1 = stack.top();
                stack.pop();
                int result = performOperation(operand1, operand2, token);
                stack.push(result);
            } else {
                stack.push(std::stoi(token));
            }
        }

        return stack.top();
    }

  private:
    bool isOperator(const std::string &token) {
        return token == "+" || token == "-" || token == "*" || token == "/";
    }

    int performOperation(int operand1, int operand2, const std::string &operatorStr) {
        if (operatorStr == "+") {
            return operand1 + operand2;
        } else if (operatorStr == "-") {
```

```
            return operand1 - operand2;
        } else if (operatorStr == "*") {
            return operand1 * operand2;
        } else if (operatorStr == "/") {
            return operand1 / operand2;
        }

        return 0; // Invalid operator
    }
};
```

## 24. 04_Stack/04_Generate_Parentheses/0022-generate-parentheses.cpp

```
/*
Problem: LeetCode 22 - Generate Parentheses

Description:
Given an integer n, generate all combinations of well-formed parentheses of length 2n.

Intuition:
To generate all combinations of well-formed parentheses, we can use a backtracking approach. At each step, we have two choices: either to place an opening parenthesis or a closing parenthesis.

Approach:
1. Initialize an empty result vector to store all valid combinations.
2. Define a helper function, backtrack, that takes the current combination, the count of opening parentheses, the count of closing parentheses, and the result vector.
3. In the backtrack function:
   - If the length of the current combination is equal to 2n, add it to the result vector.
   - If the count of opening parentheses is less than n, recursively call the backtrack function with the current combination appended with an opening parenthesis and an incremented count of opening parentheses.
     - If the count of closing parentheses is less than the count of opening parentheses, recursively call the backtrack function with the current combination appended with a closing parenthesis and an incremented count of closing parentheses.
4. Call the backtrack function initially with an empty combination, 0 opening parentheses, 0 closing parentheses, and the result vector.
5. Return the result vector containing all valid combinations.

Time Complexity:
The time complexity is O(4^n / sqrt(n)), as there are Catalan numbers of well-formed parentheses combinations.

Space Complexity:
The space complexity is O(4^n / sqrt(n)), as there can be a total of 4^n / sqrt(n) combinations generated.
*/

class Solution {
  public:
    vector<string> generateParenthesis(int n) {
        vector<string> result;
        generate(n, 0, 0, "", result);
        return result;
    }
  private:
    void generate(int n, int open, int close, string str, vector<string> &result) {
        if (open == n && close == n) {
            result.push_back(str);
            return;
        }

        if (open < n) {
            generate(n, open + 1, close, str + '(', result);
        }

        if (open > close) {
            generate(n, open, close + 1, str + ')', result);
        }
    }
};
```

## 25. 04_Stack/05_Daily_Temperatures/0739-daily-temperatures.cpp

```
/*
Problem: LeetCode 739 - Daily Temperatures

Description:
Given a list of daily temperatures, produce a list that, for each day in the input, tells you how many days you would have to wait until a warmer temperature. If there is no future day with a warmer temperature, put 0 instead.

Intuition:
To determine the number of days until a warmer temperature, we can use a stack to keep track of the previous temperatures. By iterating through the temperatures in reverse order, we can compare each temperature with the temperatures in the stack to find the next warmer day.

Approach:
1. Initialize an empty stack to store the indices of temperatures.
2. Initialize a result vector with 0s of the same size as the input temperatures.
3. Iterate through the temperatures in reverse order:
   - While the stack is not empty and the current temperature is greater than the temperature at the index at the top of the stack:
      - Pop the index from the stack and calculate the number of days until a warmer temperature (current index - popped index).
     - Update the result vector at the popped index with the number of days.
   - Push the current index onto the stack.
4. Return the result vector.

Time Complexity:
The time complexity is O(n), where n is the number of temperatures. We iterate through the temperatures once.

Space Complexity:
The space complexity is O(n), where n is the number of temperatures. In the worst case, the stack may store all the indices of the temperatures.
*/

#include<bits/stdc++.h>
using namespace std;

class Solution {
  public:
    vector<int> dailyTemperatures(vector<int> &temperatures) {
        int n = temperatures.size();
        vector<int> result(n, 0);
        stack<int> stack;

        for (int i = n - 1; i >= 0; --i) {
            while (!stack.empty() && temperatures[i] >= temperatures[stack.top()]) {
                stack.pop();
            }

            if (!stack.empty()) {
                result[i] = stack.top() - i;
            }

            stack.push(i);
        }

        return result;
    }
};
```

## 26. 04_Stack/06_Car_Fleet/0853-car-fleet.cpp

```
/*
Problem: LeetCode 853 - Car Fleet

Description:
N cars are going to the same destination along a one-lane road. The destination is target miles away.
Each car i has a constant speed speed[i] (in miles per hour), and initial position position[i] miles towards the target along the road.

A car can never pass another car ahead of it, but it can catch up to it and drive bumper to bumper at the same speed.
The distance between these two cars is ignored - they are assumed to have the same position.

A car fleet is some non-empty set of cars driving at the same position and same speed. Note that a single car is also a car fleet.

If a car catches up to a car fleet right at the destination point, it will still be considered as one car fleet.

Intuition:
To determine the number of car fleets that reach the destination, we can simulate the car movement and track the time it takes for each car to reach the destination.

Approach:
1. Create a vector of pairs to store the positions and speeds of the cars.
2. Sort the vector of pairs based on the positions in descending order.
3. Initialize the count of car fleets to 0.
4. Iterate through each car in the sorted vector:
   - Calculate the time it takes for the car to reach the destination (target - position) / speed.
    - If the current car's time is greater than the previous car's time (car fleet is not possible), increment the count of car fleets.
    - Update the previous car's time with the maximum of the current car's time and the previous car's time.
5. Return the count of car fleets.

Time Complexity:
The time complexity is O(n log n), where n is the number of cars. Sorting the vector of pairs takes O(n log n) time.

Space Complexity:
The space complexity is O(n), where n is the number of cars. We store the positions and speeds of the cars in a vector of pairs.
*/

#include<bits/stdc++.h>
using namespace std;

class Solution {
  public:
    int carFleet(int target, vector<int> &position, vector<int> &speed) {
        int n = position.size();
        vector<pair<int, int>> cars;

        // Create vector of pairs to store positions and speeds
        for (int i = 0; i < n; ++i) {
            cars.push_back({position[i], speed[i]});
        }

        // Sort the vector of pairs based on positions in descending order
        sort(cars.begin(), cars.end(), [](const pair<int, int> &a, const pair<int, int> &b) {
```

```
            return a.first > b.first;
        });
        int count = 0;
        double prevTime = 0.0;

        for (int i = 0; i < n; ++i) {
            double currTime = static_cast<double>(target - cars[i].first) / cars[i].second;

            if (currTime > prevTime) {
                count++;
                prevTime = currTime;
            }
        }

        return count;
    }
};
```

## 27. 04_Stack/07_Largest_Rectangle_In_Histogram/0084-largest-rectangle-in-histogra

```
/*
Problem: LeetCode 84 - Largest Rectangle in Histogram

Description:
Given an array of integers heights representing the histogram's bar height where the width of each
bar is 1, return the area of the largest rectangle in the histogram.

Intuition:
To find the largest rectangle in a histogram, we can utilize a stack to keep track of the indices
of increasing heights. By iterating through the histogram, we can calculate the area of each
rectangle formed by the heights.

Approach:
1. Initialize a stack to store the indices of increasing heights.
2. Initialize the maximum area to 0.
3. Iterate through each bar in the histogram:
    - While the stack is not empty and the current bar's height is less than the height at the
index at the top of the stack:
      - Pop the index from the stack and calculate the area of the rectangle formed by the popped
bar.
     - Update the maximum area if the calculated area is greater.
   - Push the current index onto the stack.
4. After iterating through all bars, there might be remaining bars in the stack. Process them
similarly to step 3 to calculate the areas and update the maximum area.
5. Return the maximum area.

Time Complexity:
The time complexity is O(n), where n is the number of bars in the histogram. We iterate through
each bar once.

Space Complexity:
The space complexity is O(n), where n is the number of bars in the histogram. In the worst case,
all bars are stored in the stack.
*/

#include<bits/stdc++.h>
using namespace std;

class Solution {
  public:
    int largestRectangleArea(vector<int> &heights) {
        int n = heights.size();
        stack<int> stack;
        int maxArea = 0;

        for (int i = 0; i <= n; ++i) {
            while (!stack.empty() && (i == n || heights[i] < heights[stack.top()])) {
                int height = heights[stack.top()];
                stack.pop();
                int width = stack.empty() ? i : i - stack.top() - 1;
                maxArea = max(maxArea, height * width);
            }

            stack.push(i);
        }

        return maxArea;
    }
};
```

## 28. 05_Binary_Search/01_Binary_Search/0704-binary-search.cpp

```
/*
Problem: LeetCode 704 - Binary Search

Description:
Given a sorted (in ascending order) integer array nums of n elements and a target value, write a function to search target in nums. If target exists, then return its index, otherwise return -1.

Intuition:
Binary search is a widely used algorithm to efficiently search for a target element in a sorted array. It works by repeatedly dividing the search space in half until the target element is found or determined to be not present.

Approach:
1. Initialize left and right pointers to the start and end of the array.
2. While the left pointer is less than or equal to the right pointer:
   - Calculate the middle index as (left + right) / 2.
   - If the middle element is equal to the target, return the middle index.
   - If the middle element is greater than the target, update the right pointer to middle - 1.
   - If the middle element is less than the target, update the left pointer to middle + 1.
3. If the target is not found after the while loop, return -1.

Time Complexity:
The time complexity of binary search is O(log n), where n is the number of elements in the array. At each step, the search space is divided in half.

Space Complexity:
The space complexity is O(1), as the algorithm uses a constant amount of extra space to store the left and right pointers.
*/

class Solution {
  public:
    int search(vector<int> &nums, int target) {
        int left = 0;
        int right = nums.size() - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] > target) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        return -1;
    }
};
```

## 29. 05_Binary_Search/02_Search_a_2D_Matrix/0074-search-a-2d-matrix.cpp

```
/*
Problem: LeetCode 74 - Search a 2D Matrix

Description:
Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:
- Integers in each row are sorted in ascending order from left to right.
- Integers in each column are sorted in ascending order from top to bottom.

Intuition:
To search for a value efficiently in the given matrix, we can utilize the sorted property of the matrix and perform binary search on both rows and columns. By narrowing down the search space, we can quickly find the target value or determine that it does not exist.

Approach:
1. Initialize the number of rows (m) and columns (n) in the matrix.
2. Perform a binary search on the rows:
   - Set the left pointer to 0 and the right pointer to m - 1.
   - While the left pointer is less than or equal to the right pointer:
     - Calculate the middle row index as (left + right) / 2.
     - If the target value is found in the middle row, return true.
      - If the target value is less than the first element of the middle row, update the right pointer to middle - 1.
      - If the target value is greater than the last element of the middle row, update the left pointer to middle + 1.
3. Perform a binary search on the columns:
   - Set the top pointer to 0 and the bottom pointer to n - 1.
   - While the top pointer is less than or equal to the bottom pointer:
     - Calculate the middle column index as (top + bottom) / 2.
     - If the target value is found in the middle column, return true.
     - If the target value is less than the first element of the middle column, update the bottom pointer to middle - 1.
      - If the target value is greater than the last element of the middle column, update the top pointer to middle + 1.
4. If the target value is not found after both binary searches, return false.

Time Complexity:
The time complexity is O(log m + log n), where m is the number of rows and n is the number of columns in the matrix. Binary search is performed on both rows and columns.

Space Complexity:
The space complexity is O(1), as the algorithm uses a constant amount of extra space.
*/

class Solution {
  public:
    bool searchMatrix(std::vector<std::vector<int>> &matrix, int target) {
        int m = matrix.size();
        int n = matrix[0].size();
        int leftRow = 0;
        int rightRow = m - 1;
        int topCol = 0;
        int bottomCol = n - 1;

        while (leftRow <= rightRow) {
            int midRow = leftRow + (rightRow - leftRow) / 2;

            if (matrix[midRow][0] <= target && target <= matrix[midRow][n - 1]) {
                while (topCol <= bottomCol) {
                    int midCol = topCol + (bottomCol - topCol) / 2;
```

```
                if (matrix[midRow][midCol] == target) {
                    return true;
                } else if (matrix[midRow][midCol] < target) {
                    topCol = midCol + 1;
                } else {
                    bottomCol = midCol - 1;
                }
            }

            break;
        } else if (matrix[midRow][0] > target) {
            rightRow = midRow - 1;
        } else {
            leftRow = midRow + 1;
        }
    }

    return false;
  }
};
```

## 30. 05_Binary_Search/03_Koko_Eating_Bananas/0875-koko-eating-bananas.cpp

```
/*
Problem: LeetCode 875 - Koko Eating Bananas

Description:
Koko loves to eat bananas. There are n piles of bananas, the ith pile has piles[i] bananas. The guards have gone and will come back in h hours.
Koko can decide her bananas-per-hour eating speed of k. Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour.
Koko likes to eat slowly but still wants to finish eating all the bananas before the guards come back.
Return the minimum integer k such that she can eat all the bananas within h hours.

Intuition:
To find the minimum eating speed that allows Koko to eat all the bananas within h hours, we can use binary search. By defining the search space and adjusting the speed based on the time taken to eat bananas, we can find the optimal speed.

Approach:
1. Set the left and right pointers to 1 and the maximum number of bananas in piles, respectively.
2. While the left pointer is less than the right pointer:
   - Calculate the middle value as (left + right) / 2, which represents the eating speed.
   - Initialize the total hours as 0.
   - Iterate through each pile of bananas:
        - Calculate the time taken to eat the pile as ceil(piles[i] / (double)middle), which represents the number of hours needed to eat the bananas in the pile.
     - Add the time taken to the total hours.
   - If the total hours is less than or equal to h, update the right pointer to middle to search for a lower eating speed.
   - If the total hours is greater than h, update the left pointer to middle + 1 to search for a higher eating speed.
3. Return the left pointer, which represents the minimum eating speed.

Time Complexity:
The time complexity is O(n log m), where n is the number of piles of bananas and m is the maximum number of bananas in piles. Binary search is performed on the search space of eating speeds.

Space Complexity:
The space complexity is O(1), as the algorithm uses a constant amount of extra space.
*/

#include <cmath>

class Solution {
  public:
    int minEatingSpeed(vector<int> &piles, int h) {
        int left = 1;
        int right = *max_element(piles.begin(), piles.end());

        while (left < right) {
            int middle = left + (right - left) / 2;
            int totalHours = 0;

            for (int bananas : piles) {
                // handling integer overflow by using ceil to round up the division result
                     // and ensuring the result is accurate by casting bananas to double before performing division
                totalHours += ceil(static_cast<double>(bananas) / middle);
            }
```

```
            if (totalHours <= h) {
                right = middle;
            } else {
                left = middle + 1;
            }
        }

        return left;
    }
};

/*
class Solution {
public:
    // Helper function to check if a given eating speed works within the given time constraint
    bool isSpeedFeasible(vector<int>& piles, int k, int h) {
        double hours = 0;
        for (int bananas : piles) {
            hours += ceil(static_cast<double>(bananas) / k);
        }
        return hours <= h;
    }

    int minEatingSpeed(vector<int>& piles, int h) {
        int start = 1;
        int end = *max_element(piles.begin(), piles.end());
        int bestSpeed = end;

        while (start <= end) {
            int mid = start + (end - start) / 2;

            if (isSpeedFeasible(piles, mid, h)) {
                bestSpeed = mid;
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }

        return bestSpeed;
    }
};
*/
```

## 31. 05_Binary_Search/04_Find_Minimum_In_Rotated_Sorted_Array/0153-find-minimu

```
/*
Problem: LeetCode 153 - Find Minimum in Rotated Sorted Array

Description:
Suppose an array of length n is sorted in ascending order, rotated at some pivot unknown to you
beforehand. You are given a target value to search for. If found in the array, return its index,
otherwise return -1.

Intuition:
To find the minimum element in the rotated sorted array, we can utilize the property that the
minimum element is the only element that is smaller than its adjacent elements. By performing a
modified version of binary search, we can efficiently locate the minimum element.

Approach:
1. Initialize left and right pointers to the start and end of the array.
2. While the left pointer is less than the right pointer:
   - Calculate the middle index as (left + right) / 2.
   - If the middle element is greater than the last element, the minimum element is on the right
side of the middle. Update the left pointer to middle + 1.
   - If the middle element is less than or equal to the last element, the minimum element is on
the left side of the middle. Update the right pointer to middle.
3. Return the element at the left pointer, which represents the minimum element.

Time Complexity:
The time complexity is O(log n), where n is the length of the array. At each step, the search
space is divided in half.

Space Complexity:
The space complexity is O(1), as the algorithm uses a constant amount of extra space.
*/

class Solution {
  public:
    int findMin(vector<int> &nums) {
        int left = 0; // Initialize the left pointer to the start of the array
        int right = nums.size() - 1; // Initialize the right pointer to the end of the array

          while (left < right) { // Perform binary search until left pointer is less than right
pointer
            int middle = left + (right - left) / 2; // Calculate the middle index

            if (nums[middle] > nums[right]) {
                // If the middle element is greater than the last element,
                // it means the rotation point is on the right side of the middle.
                // Update the left pointer to middle + 1 to search in the right half.
                left = middle + 1;
            } else {
                // If the middle element is less than or equal to the last element,
                  // it means the rotation point is on the left side of the middle or the middle
element itself.
                // Update the right pointer to middle to search in the left half.
                right = middle;
            }
        }

        return nums[left]; // Return the element at the left pointer, which represents the minimum
element.
    }
};
```

```
// class Solution {
// public:
//     int findMin(vector<int>& nums) {
//         int low = 0;
//         int high = nums.size() - 1;

//         // not low <= high since not searching for target
//         while (low < high) {
//             int mid = low + (high - low) / 2;
//             // ex. [3,4,5,6,7,8,9,1,2], mid = 4, high = 8
//             // nums[mid] > nums[high], min must be right
//             if (nums[mid] > nums[high]) {
//                 // never consider mid bc know for sure not min
//                 low = mid + 1;
//             // ex. [8,9,1,2,3,4,5,6,7], mid = 4, high = 8
//             // nums[mid] <= nums[right], min must be left
//             } else {
//                 // consider mid still bc could be min
//                 high = mid;
//             }
//         }

//         // low lands on correct value, never disqualified mins
//         return nums[low];
//     }
// };
```

## 32. 05_Binary_Search/05_Search_In_Rotated_Sorted_Array/0033-search-in-rotated-so

```
/*
Problem: LeetCode 33 - Search in Rotated Sorted Array

Description:
Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. You
are given a target value to search for. If found in the array, return its index, otherwise return
-1.
You may assume no duplicate exists in the array.

Intuition:
To search for a target value efficiently in the rotated sorted array, we can utilize a modified
version of binary search. By comparing the target value with the array elements and adjusting the
search space based on the rotation point, we can find the target value or determine its absence.

Approach:
1. Initialize the left and right pointers to the start and end of the array.
2. While the left pointer is less than or equal to the right pointer:
   - Calculate the middle index as (left + right) / 2.
   - If the middle element is equal to the target value, return the middle index.
   - If the left half of the array is sorted:
      - If the target value is within the range of the left half, update the right pointer to
middle - 1.
     - Otherwise, update the left pointer to middle + 1.
   - If the right half of the array is sorted:
      - If the target value is within the range of the right half, update the left pointer to
middle + 1.
     - Otherwise, update the right pointer to middle - 1.
3. If the target value is not found after the while loop, return -1.

Time Complexity:
The time complexity is O(log n), where n is the size of the array. At each step, the search space
is divided in half.

Space Complexity:
The space complexity is O(1), as the algorithm uses a constant amount of extra space.
*/

class Solution {
  public:
    int search(vector<int> &nums, int target) {
        int left = 0;
        int right = nums.size() - 1;

        while (left <= right) {
            int middle = left + (right - left) / 2;

               if (nums[middle] == target) { // If middle element is equal to the target value,
return the middle index
                return middle;
            }

            if (nums[left] <= nums[middle]) { // If left half of the array is sorted
                 if (target >= nums[left] && target < nums[middle]) { // If target value is within
the range of the left half
                    right = middle - 1; // Update the right pointer to search in the left half
                } else {
                    left = middle + 1; // Update the left pointer to search in the right half
                }
            } else { // If right half of the array is sorted
                 if (target > nums[middle] && target <= nums[right]) { // If target value is within
```

```
the range of the right half
                    left = middle + 1; // Update the left pointer to search in the right half
                } else {
                    right = middle - 1; // Update the right pointer to search in the left half
                }
            }
        }

        return -1; // Target value not found, return -1
    }
};
```

## 33. 05_Binary_Search/06_Time_Based_Key_Value_Store/0981-time-based-key-value-

```
/*
Problem: LeetCode 981 - Time Based Key-Value Store

Description:
Design a time-based key-value data structure that can store multiple values for the same key and retrieve the value of a key at a certain time.
Implement the TimeMap class:
- TimeMap() Initializes the object of the data structure.
- void set(String key, String value, int timestamp) Stores the key, value pair in the data structure.
- String get(String key, int timestamp) Returns a value such that set(key, value, timestamp_prev) was called previously, with timestamp_prev <= timestamp.
     If there are multiple such values, it returns the value associated with the largest timestamp_prev.
   If there are no values, it returns an empty string ("").

Intuition:
To implement the TimeMap, we can utilize a hashmap data structure to store the key-value pairs. Each key will map to a list of timestamps and corresponding values.
By using binary search within the list of timestamps, we can efficiently retrieve the value associated with a given key and timestamp.

Approach:
1. Implement the TimeMap class.
2. Create a hashmap, where each key maps to a list of pairs containing timestamps and values.
3. Implement the set function:
     - If the key does not exist in the hashmap, create a new list with the first pair of (timestamp, value).
   - If the key already exists, append the new pair of (timestamp, value) to the existing list.
4. Implement the get function:
   - If the key does not exist in the hashmap, return an empty string.
   - If the key exists, perform binary search within the list of pairs:
        - If the timestamp at the middle index is equal to the target timestamp, return the corresponding value.
     - If the timestamp at the middle index is greater than the target timestamp, update the right pointer to middle - 1 and continue searching in the left half.
      - If the timestamp at the middle index is less than the target timestamp, update the left pointer to middle + 1 and continue searching in the right half.
5. If the binary search does not find an exact match, return the value at the index of the right pointer.

Time Complexity:
The time complexity of set is O(1), as it only involves hashmap operations.
The time complexity of get is O(log n), where n is the number of entries for a given key in the hashmap. The binary search is performed within the list of timestamps.

Space Complexity:
The space complexity is O(n), where n is the total number of entries in the hashmap. Each entry occupies space for the key and a list of timestamp-value pairs.
*/

class TimeMap {
  private:
    unordered_map<string, vector<pair<int, string>>> data; // Hashmap to store key-value pairs

  public:
    TimeMap() {}

    void set(string key, string value, int timestamp) {
          data[key].emplace_back(timestamp, value); // Append the pair (timestamp, value) to the
```

```
list for the key
    }

    string get(string key, int timestamp) {
        if (data.find(key) == data.end()) {
            return ""; // Key does not exist, return empty string
        }

         const vector<pair<int, string>> &entries = data[key]; // Get the list of entries for the
key
        int left = 0;
        int right = entries.size() - 1;

        while (left <= right) { // Perform binary search
            int middle = left + (right - left) / 2; // Calculate the middle index

            if (entries[middle].first == timestamp) {
                return entries[middle].second; // Exact match found, return the value
            } else if (entries[middle].first > timestamp) {
                   right = middle - 1; // Target timestamp is in the left half, update the right
pointer
            } else {
                   left = middle + 1; // Target timestamp is in the right half, update the left
pointer
            }
        }

        if (right >= 0) {
             return entries[right].second; // No exact match found, return the value at the right
pointer index
        }

        return ""; // No value found, return empty string
    }
};
```

## 34. 05_Binary_Search/07_Median_of_Two_Sorted_Arrays/0004-median-of-two-sorted

```
/*
Problem: LeetCode 4 - Median of Two Sorted Arrays

Description:
Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.

Intuition:
To find the median of two sorted arrays, we need to divide the combined array into two equal halves. The median is the middle element or the average of the two middle elements, depending on whether the total number of elements is odd or even. Since the arrays are sorted, we can leverage binary search and partitioning to find the median efficiently.

Approach:
1. Ensure that nums1 is the smaller array. If not, swap nums1 and nums2.
2. Use binary search to find the correct partitioning point in the smaller array (nums1).
3. The partition point i divides nums1 into two parts: elements before i form the left half, and elements after i form the right half.
4. Calculate the partition point j in nums2 based on i, such that j = (m + n + 1) / 2 - i, where m is the size of nums1 and n is the size of nums2.
5. Check if the partitioning is valid by comparing the maximum element of the left half (maxLeft) with the minimum element of the right half (minRight).
6. If the partitioning is valid, calculate the median based on the array sizes and the partition points:
   - If the total number of elements (m + n) is odd, the median is maxLeft.
   - If the total number of elements is even, the median is the average of maxLeft and minRight.
7. If the partitioning is not valid, adjust the partition points using binary search until a valid partitioning is found.
8. Return the median.

Time Complexity:
The time complexity of the approach is O(log(min(m, n))), where m and n are the sizes of the input arrays. The binary search is performed on the smaller array.

Space Complexity:
The space complexity is O(1) as the approach uses only a constant amount of extra space.

*/

class Solution {
  public:
    double findMedianSortedArrays(vector<int> &nums1, vector<int> &nums2) {
        // Ensure nums1 is the smaller array
        if (nums1.size() > nums2.size()) {
            nums1.swap(nums2);
        }

        int m = nums1.size();
        int n = nums2.size();
        int left = 0;
        int right = m;
        int halfLen = (m + n + 1) / 2;

        while (left <= right) {
            int i = left + (right - left) / 2; // Partition point in nums1
            int j = halfLen - i; // Partition point in nums2

            if (i < m && nums2[j - 1] > nums1[i]) {
                // i is too small, increase it to the right half
                left = i + 1;
```

```
            } else if (i > 0 && nums1[i - 1] > nums2[j]) {
                // i is too large, decrease it to the left half
                right = i - 1;
            } else {
                // Found the correct partitioning
                int maxLeft = 0;

                if (i == 0) {
                    maxLeft = nums2[j - 1];
                } else if (j == 0) {
                    maxLeft = nums1[i - 1];
                } else {
                    maxLeft = max(nums1[i - 1], nums2[j - 1]);
                }

                if ((m + n) % 2 == 1) {
                    return maxLeft; // Odd number of elements, median is the max of the left half
                }

                int minRight = 0;

                if (i == m) {
                    minRight = nums2[j];
                } else if (j == n) {
                    minRight = nums1[i];
                } else {
                    minRight = min(nums1[i], nums2[j]);
                }

                    return (maxLeft + minRight) / 2.0; // Even number of elements, median is the
average of max left and min right
            }
        }

        // Should never reach this point
        return 0.0;
    }
};

// class Solution {
// public:
//     double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
//         if (nums1.size() > nums2.size()) {
//             // if nums1 is bigger we swap it because we're gonna assume nums1 is smaller
//             return findMedianSortedArrays(nums2, nums1);
//         }

//         int m = nums1.size();
//         int n = nums2.size();

//         double Answer = 0.0;

//         // Now that nums1 is smaller
//         int L = 0, R = m;

//         while(L <= R) {
//             // Mid of nums1 & nums2
//             int MidM = L + (R-L)/2;
//             int MidN = (m+n+1)/2 - MidM;

//             int a = (MidM > 0) ? nums1[MidM - 1] : INT_MIN;
//             int b = (MidM < m) ? nums1[MidM] : INT_MAX;
//             int c = (MidN > 0) ? nums2[MidN - 1] : INT_MIN;
//             int d = (MidN < n) ? nums2[MidN] : INT_MAX;
```

```
//             // If both halves are correctly sorted
//             if(a <= d && c <= b) {
//                 // Checking if the merged array has even elements or odd
//                 if( (m + n) % 2 == 0)
//                     Answer = (max(a, c) + min(b, d)) / 2.0;
//                 else
//                     Answer = max(a, c);

//                 break;
//             }

//               // If not correctly sorted and right part of nums2 is lesser than left part of
nums1
//             else if(d < a)
//                 R = MidM - 1;
//               // If not correctly sorted and right side of nums1 is lesser than left part of
nums2 i.e. c < b
//             else
//                 L = MidM + 1;
//         }

//         return Answer;
//     }
// };
```

## 35. 06_Linked_List/01_Reverse_Linked_List/0206-reverse-linked-list.cpp

```
/*
Problem: LeetCode 206 - Reverse Linked List

Description:
Reverse a singly linked list.

Intuition:
To reverse a linked list, we need to change the direction of each node's "next" pointer. We can do this iteratively by keeping track of three nodes: current, previous, and next. We start with current pointing to the head of the original list, previous as nullptr (since there is no node before the head), and next as the next node of the current node. During each iteration, we update the "next" pointer of the current node to point to the previous node. Then, we move the previous node to be the current node and the current node to be the next node. We repeat this process until the end of the original list is reached.

Approach:
1. Create three pointers: current, previous, and next.
2. Initialize current to the head of the original list and previous as nullptr.
3. While the current node is not nullptr:
   a. Update the "next" pointer of the current node to point to the previous node.
   b. Move previous to be the current node and current to be the next node.
4. At the end of the loop, the previous node will be the new head of the reversed list.

Time Complexity:
The time complexity is O(N), where N is the number of nodes in the linked list. We visit each node once during the reversal process.

Space Complexity:
The space complexity is O(1) since we are using a constant amount of extra space for the three pointers.
*/

class Solution {
  public:
    ListNode *reverseList(ListNode *head) {
        ListNode *current = head;
        ListNode *previous = nullptr;
        ListNode *next;

        while (current) {
            next = current->next;
            current->next = previous;
            previous = current;
            current = next;
        }

        return previous;
    }
};
```

## 36. 06_Linked_List/02_Merge_Two_Sorted_Lists/0021-merge-two-sorted-lists.cpp

```
/*
Problem: LeetCode 21 - Merge Two Sorted Lists

Description:
Merge two sorted linked lists and return it as a new sorted list.
The new list should be made by splicing together the nodes of the first two lists.

Intuition:
To merge two sorted linked lists, we can use a simple approach where we compare the values of the nodes at the head of both lists.
We create a dummy node to serve as the new merged list's head. Then, we compare the values of the current nodes from both lists.
Whichever node's value is smaller, we append that node to the merged list and move its pointer to the next node.
We repeat this process until we have traversed both input lists completely.

Approach:
1. Create a dummy node as the merged list's head.
2. Initialize two pointers, "curr" and "dummy," both pointing to the dummy node.
3. While both input lists are not empty:
   a. Compare the values of the current nodes from both lists.
   b. Append the node with the smaller value to the merged list.
    c. Move the pointer of the merged list and the pointer of the selected node's list to their next nodes.
4. If any of the input lists still has nodes remaining, append the rest of that list to the merged list.
5. Return the merged list's head, which is the next node of the dummy node.

Time Complexity:
The time complexity is O(N+M), where N and M are the number of nodes in the input lists, as we need to visit each node once.

Space Complexity:
The space complexity is O(1), as we only use a constant amount of extra space for the pointers.
*/

class Solution {
  public:
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        ListNode *dummy = new ListNode(0);
        ListNode *curr = dummy;

        while (l1 && l2) {
            if (l1->val < l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }

            curr = curr->next;
        }

        if (l1) {
            curr->next = l1;
        } else {
            curr->next = l2;
        }
```

```
        return dummy->next;
    }
};
```

## 37. 06_Linked_List/03_Reorder_List/0143-reorder-list.cpp

```
/*
Problem: LeetCode 143 - Reorder List

Description:
Given a singly linked list L: L0 -> L1 -> ... -> Ln-1 -> Ln,
reorder it to: L0 -> Ln -> L1 -> Ln-1 -> L2 -> Ln-2 -> ...

You may not modify the values in the list's nodes, only nodes itself may be changed.

Intuition:
To reorder the linked list, we can divide the problem into three main steps:
1. Find the middle of the list using the slow and fast pointer approach.
2. Reverse the second half of the list.
3. Merge the first half and the reversed second half together to form the reordered list.

Approach:
1. Use the slow and fast pointer approach to find the middle of the list.
2. Reverse the second half of the list.
3. Merge the first half and the reversed second half together to form the reordered list.

Time Complexity:
The time complexity is O(N), where N is the number of nodes in the linked list, as we need to
traverse the list twice (once to find the middle and once to reverse the second half).

Space Complexity:
The space complexity is O(1) as we are using constant extra space for the pointers.
*/

class Solution {
  public:
    void reorderList(ListNode *head) {
        if (!head || !head->next || !head->next->next) {
            return;
        }

        // Step 1: Find the middle of the list
        ListNode *slow = head;
        ListNode *fast = head;

        while (fast->next && fast->next->next) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Step 2: Reverse the second half of the list
        ListNode *prev = nullptr;
        ListNode *curr = slow->next;
        slow->next = nullptr;

        while (curr) {
            ListNode *nextNode = curr->next;
            curr->next = prev;
            prev = curr;
            curr = nextNode;
        }

        // Step 3: Merge the first half and the reversed second half
        ListNode *first = head;
        ListNode *second = prev;
```

```
        while (second) {
            ListNode *nextFirst = first->next;
            ListNode *nextSecond = second->next;
            first->next = second;
            second->next = nextFirst;
            first = nextFirst;
            second = nextSecond;
        }
    }
};
```

## 38. 06_Linked_List/04_Remove_Nth_Node_From_End_of_List/0019-remove-nth-node

```
/*
Problem: LeetCode 19 - Remove Nth Node From End of List

Description:
Given the head of a linked list, remove the nth node from the end of the list and return its head.

Intuition:
To remove the nth node from the end of the list, we can use the two-pointer approach. We will use two pointers, first and second, with a gap of n nodes between them. When the second pointer reaches the end of the list, the first pointer will be pointing to the nth node from the end. We can then remove that node.

Approach:
1. Create a dummy node and set it as the head of the list to handle cases where we need to remove the head.
2. Initialize both first and second pointers to the dummy node.
3. Move the second pointer n+1 times to create a gap of n nodes between first and second.
4. Move both first and second pointers simultaneously until the second pointer reaches the end.
5. Remove the nth node by updating the next pointer of the previous node to skip the nth node.

Time Complexity:
The time complexity is O(N), where N is the number of nodes in the linked list, as we need to traverse the list once to find the nth node from the end.

Space Complexity:
The space complexity is O(1) as we are using constant extra space for the pointers.
*/

class Solution {
  public:
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        ListNode *dummy = new ListNode(0);
        dummy->next = head;
        ListNode *first = dummy;
        ListNode *second = dummy;

        // Move second pointer n+1 steps ahead
        for (int i = 0; i <= n; i++) {
            second = second->next;
        }

        // Move both pointers simultaneously until second reaches the end
        while (second) {
            first = first->next;
            second = second->next;
        }

        // Remove the nth node by updating the next pointer of the previous node
        ListNode *temp = first->next;
        first->next = first->next->next;
        delete temp;
        return dummy->next;
    }
};
```

## 39. 06_Linked_List/05_Copy_List_With_Random_Pointer/0138-copy-list-with-random-

```
/*
Problem: LeetCode 138 - Copy List with Random Pointer

Description:
A linked list is given such that each node contains an additional random pointer that could point to any node in the list or null.
Return a deep copy of the list.

Intuition:
To create a deep copy of the given linked list, we can use a hashmap to keep track of the mapping between the original nodes and their copies. We will traverse the original list, create a copy of each node, and store the mapping in the hashmap. Then we will traverse the list again to connect the copied nodes with their corresponding random pointers.

Approach:
1. Traverse the original list, create a copy of each node, and store the mapping in the hashmap.
2. Traverse the original list again, and for each node, connect its copy with the corresponding random pointer using the hashmap.

Time Complexity:
The time complexity is O(N), where N is the number of nodes in the linked list, as we need to traverse the list twice.

Space Complexity:
The space complexity is O(N) as we use extra space to store the hashmap, where N is the number of nodes in the linked list.
*/

class Solution {
  public:
    Node *copyRandomList(Node *head) {
        if (!head) {
            return nullptr;
        }

        unordered_map<Node *, Node *> nodeMap;
        Node *current = head;

        // Traverse the original list and create a copy of each node
        while (current) {
            nodeMap[current] = new Node(current->val);
            current = current->next;
        }

        current = head;

        // Traverse the original list again to connect the copied nodes with their random pointers
        while (current) {
            nodeMap[current]->next = nodeMap[current->next];
            nodeMap[current]->random = nodeMap[current->random];
            current = current->next;
        }

        return nodeMap[head];
    }
};
```

## 40. 06_Linked_List/06_Add_Two_Numbers/0002-add-two-numbers.cpp

```
/*
Problem: LeetCode 2 - Add Two Numbers

Description:
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.

Intuition:
We can traverse the two linked lists simultaneously, adding the digits at the same position and keeping track of the carry. As we move forward, we create a new node for the sum and update the carry for the next iteration. If any linked list has more digits left, we continue the process until both lists are fully traversed.

Approach:
1. Initialize a dummy node to the head of the result list.
2. Initialize variables `carry` and `sum` to 0.
3. Traverse both linked lists simultaneously until both are fully traversed.
4. At each step, compute the sum of digits and the carry for the next iteration.
5. Create a new node with the sum and attach it to the result list.
6. Move the current pointers of both input lists to the next nodes.
7. If any list has remaining digits, continue adding them to the result.
8. Return the head of the result list after skipping the dummy node.

Time Complexity:
The time complexity is O(max(N, M)), where N and M are the number of nodes in the input linked lists. We traverse both lists once.

Space Complexity:
The space complexity is O(max(N, M)), where N and M are the number of nodes in the input linked lists. The extra space is used to store the result list.
*/

class Solution {
  public:
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
        ListNode *dummy = new ListNode(0);
        ListNode *current = dummy;
        int carry = 0;

        while (l1 || l2) {
            int sum = carry;

            if (l1) {
                sum += l1->val;
                l1 = l1->next;
            }

            if (l2) {
                sum += l2->val;
                l2 = l2->next;
            }

            carry = sum / 10;
            current->next = new ListNode(sum % 10);
            current = current->next;
        }

        if (carry) {
            current->next = new ListNode(carry);
```

```
        }

        return dummy->next;
    }
};
```

## 41. 06_Linked_List/07_Linked_List_Cycle/0141-linked-list-cycle.cpp

```
/*
Problem: LeetCode 141 - Linked List Cycle

Description:
Given a linked list, determine if it has a cycle in it.
To represent a cycle in the given linked list, we use an integer pos,
which represents the position (0-indexed) in the linked list where the tail connects to.
If pos is -1, there is no cycle in the linked list.

Intuition:
To detect if there is a cycle in a linked list, we can use the two-pointer technique.
We maintain two pointers, slow and fast. The slow pointer moves one step at a time,
while the fast pointer moves two steps at a time. If there is a cycle in the linked list,
the fast pointer will eventually catch up to the slow pointer, and they will meet.

Approach:
1. Initialize two pointers, slow and fast, to the head of the linked list.
2. Move the slow pointer one step and the fast pointer two steps at a time.
3. If there is a cycle, the fast pointer will eventually catch up to the slow pointer.
4. If the fast pointer becomes null, there is no cycle in the linked list.
5. Return true if the two pointers meet, otherwise return false.

Time Complexity:
The time complexity is O(n), where n is the number of nodes in the linked list.
In the worst case, the fast pointer traverses the linked list twice.

Space Complexity:
The space complexity is O(1) since we only use two pointers to detect the cycle.
*/

class Solution {
  public:
    bool hasCycle(ListNode *head) {
        ListNode *slow = head;
        ListNode *fast = head;

        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;

            if (slow == fast) {
                return true;
            }
        }

        return false;
    }
};
```

## 42. 06_Linked_List/08_Find_The_Duplicate_Number/0287-find-the-duplicate-number.c

```
/*
Problem: LeetCode 287 - Find the Duplicate Number

Description:
Given an array nums containing n + 1 integers where each integer is between 1 and n (inclusive),
prove that at least one duplicate number must exist in the array.
Assume that there is only one duplicate number, find the duplicate one.

Intuition:
To find the duplicate number, we can use the concept of cycle detection in a linked list.
Since the numbers in the array are between 1 and n, we can treat the array as a linked list,
where each element points to the value at its index.
If there is a duplicate number, it means there is a cycle in the linked list.

Approach:
1. Use the two-pointer technique to detect the cycle in the array.
2. Initialize slow and fast pointers to the first element (index 0) of the array.
3. Move the slow pointer one step at a time and the fast pointer two steps at a time.
4. When the two pointers meet, reset one of the pointers to the first element (index 0).
5. Move both pointers one step at a time until they meet again. The meeting point is the duplicate
number.

Time Complexity:
The time complexity is O(n), where n is the length of the array.
The loop to detect the cycle takes at most n steps.

Space Complexity:
The space complexity is O(1) since we are not using any additional data structures.

Note:
The given array is assumed to have at least one duplicate number, and the duplicate number will
always exist.
*/

class Solution {
  public:
    int findDuplicate(vector<int> &nums) {
        int slow = nums[0];     // Tortoise
        int fast = nums[0];     // Hare

        // Step 1: Detect the cycle
        do {
            slow = nums[slow];
            fast = nums[nums[fast]];
        } while (slow != fast);

        // Step 2: Find the entrance to the cycle (duplicate number)
        slow = nums[0];

        while (slow != fast) {
            slow = nums[slow];
            fast = nums[fast];
        }

        return slow;
    }
};
```

## 43. 06_Linked_List/09_LRU_Cache/0146-lru-cache.cpp

```
/*
Problem: LeetCode 146 - LRU Cache

Description:
Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.
Implement the LRUCache class:
- LRUCache(int capacity) Initialize the LRU cache with positive size capacity.
- int get(int key) Return the value of the key if the key exists, otherwise return -1.
- void put(int key, int value) Update the value of the key if the key exists.
  Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation,
  evict the least recently used key.

Intuition:
To efficiently implement an LRU cache, we can use a combination of a hashmap and a doubly linked list.
The hashmap will store the key-value pairs, and the doubly linked list will maintain the order of the elements based on their usage.

Approach:
1. We use a hashmap to store the key-value pairs for quick access.
2. We use a doubly linked list to maintain the order of the elements based on their usage.
3. When getting a value from the cache, we first check if the key exists in the hashmap.
    If it does, we update the usage order in the doubly linked list by moving the corresponding node to the front (most recently used).
4. When putting a key-value pair into the cache, we check if the key already exists in the hashmap.
    If it does, we update the value and move the corresponding node to the front (most recently used).
   If it doesn't, we add a new node to the front of the doubly linked list and update the hashmap.
    If the number of keys exceeds the capacity, we remove the least recently used node from the back of the doubly linked list
   and also remove the corresponding key from the hashmap.

Time Complexity:
Both get and put operations have a time complexity of O(1) since all operations (insert, delete, and access) on the doubly linked list are constant time.

Space Complexity:
The space complexity is O(capacity) to store the key-value pairs and doubly linked list nodes.
*/

class LRUCache {
  private:
    // Structure to represent a doubly linked list node
    struct ListNode {
        int key;         // The key of the cache item
        int value;       // The value of the cache item
        ListNode *prev;  // Pointer to the previous node in the list
        ListNode *next;  // Pointer to the next node in the list

        ListNode(int k, int v) : key(k), value(v), prev(nullptr), next(nullptr) {}
    };

    int capacity;                     // Maximum capacity of the LRUCache
    unordered_map<int, ListNode *> hashmap; // Hashmap to store key-node mappings
    ListNode *head;                   // Dummy head node for the doubly linked list
    ListNode *tail;                   // Dummy tail node for the doubly linked list

    // Helper function to add a node to the front of the linked list
```

```
    void addToFront(ListNode *node) {
        node->next = head->next;
        node->prev = head;
        head->next->prev = node;
        head->next = node;
    }

    // Helper function to remove a node from the linked list
    void removeFromList(ListNode *node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }

  public:
    // Constructor to initialize the LRUCache with a given capacity
    LRUCache(int capacity) {
        this->capacity = capacity;
        head = new ListNode(-1, -1); // Initialize dummy head with key and value -1
        tail = new ListNode(-1, -1); // Initialize dummy tail with key and value -1
        head->next = tail;           // Connect head to tail initially (empty list)
        tail->prev = head;           // Connect tail to head initially (empty list)
    }

    // Function to get the value of the given key from the cache
    int get(int key) {
        if (hashmap.find(key) != hashmap.end()) {
            ListNode *node = hashmap[key];
            removeFromList(node); // Move the accessed node to the front
            addToFront(node);
            return node->value;   // Return the value associated with the key
        }

        return -1; // Key not found, return -1
    }

    // Function to put a key-value pair in the cache
    void put(int key, int value) {
        if (hashmap.find(key) != hashmap.end()) {
            ListNode *node = hashmap[key];
            removeFromList(node);  // Remove the existing node from the list
            node->value = value;   // Update the value
            addToFront(node);      // Move the updated node to the front
        } else {
            if (hashmap.size() >= capacity) {
                ListNode *removedNode = tail->prev;
                removeFromList(removedNode); // Remove the last node (least recently used)
                hashmap.erase(removedNode->key); // Remove the key from the hashmap
                delete removedNode; // Free the memory of the removed node
            }

             ListNode *newNode = new ListNode(key, value); // Create a new node for the key-value
pair
            addToFront(newNode); // Add the new node to the front of the list
            hashmap[key] = newNode; // Add the key-node mapping to the hashmap
        }
    }

    // Destructor to properly deallocate memory for the ListNode objects
    ~LRUCache() {
        ListNode *curr = head;

        while (curr) {
            ListNode *temp = curr;
            curr = curr->next;
```

```
            delete temp;
        }
    }
};
```

## 44. 06_Linked_List/10_Merge_K_Sorted_Lists/0023-merge-k-sorted-lists.cpp

```
/*
Problem: LeetCode 23 - Merge k Sorted Lists

Description:
You are given an array of k linked-lists, each linked-list is sorted in ascending order.
Merge all the linked-lists into one sorted linked-list and return it.

Intuition:
To merge k sorted lists, we can use a priority queue (min heap). We start by pushing the head nodes of all k linked lists into the priority queue.
Then, while the priority queue is not empty, we pop the node with the minimum value, append it to the result list, and push its next node (if exists) back into the priority queue.
This way, we always pick the smallest element to add to the result list, which ensures that the final list is sorted.

Approach:
1. Create a struct ListNode to represent the nodes of the linked list.
2. Create a custom comparator function for the priority queue to compare ListNode pointers based on their values.
3. Initialize a priority queue (min heap) using the custom comparator.
4. Push the head nodes of all k linked lists into the priority queue.
5. Initialize a dummy ListNode to build the merged list.
6. While the priority queue is not empty, pop the node with the minimum value, append it to the dummy list, and push its next node (if exists) back into the priority queue.
7. Finally, return the next node of the dummy list, which will be the head of the merged sorted list.

Time Complexity:
The overall time complexity is O(n*log(k)), where n is the total number of nodes and k is the number of linked lists.
The priority queue operations take O(log(k)) time, and we do this for all n nodes.

Space Complexity:
The space complexity is O(k), as we store the head nodes of all k linked lists in the priority queue.
*/

struct ListNodeComparator {
    bool operator()(const ListNode *a, const ListNode *b) const {
        return a->val > b->val;
    }
};

class Solution {
  public:
    ListNode *mergeKLists(vector<ListNode *> &lists) {
        priority_queue<ListNode *, vector<ListNode *>, ListNodeComparator> pq;

        for (ListNode *head : lists) {
            if (head) {
                pq.push(head);
            }
        }

        ListNode dummy(0);
        ListNode *current = &dummy;

        while (!pq.empty()) {
            ListNode *smallest = pq.top();
            pq.pop();
```

```
            current->next = smallest;
            current = current->next;

            if (smallest->next) {
                pq.push(smallest->next);
            }
        }

        return dummy.next;
    }
};
```

## 45. 06_Linked_List/11_Reverse_Nodes_In_K_Group/0025-reverse-nodes-in-k-group.c

```
/*
Problem: LeetCode 25 - Reverse Nodes in k-Group

Description:
Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.
k is a positive integer and is less than or equal to the length of the linked list.
If the number of nodes is not a multiple of k, then the remaining nodes should remain as it is.

Intuition:
To reverse the nodes in k-groups, we can use a recursive approach.
We start by finding the k+1-th node in the linked list, as it will be the new head of the reversed k-group.
Then we reverse the current k-group and connect it to the next reversed k-group.
We repeat this process until we reach the end of the linked list.

Approach:
1. Create a struct ListNode to represent the nodes of the linked list.
2. Implement a helper function to reverse a k-group of nodes, which takes the head and tail of the group as input and returns the new head of the reversed group.
3. Initialize a dummy ListNode to build the final result list.
4. Create a pointer current to iterate through the linked list.
5. Find the k+1-th node from the current node.
6. If the k+1-th node exists, reverse the current k-group and update the pointers accordingly.
7. Append the reversed k-group to the dummy list and move the current pointer to the k+1-th node.
8. Repeat steps 5 to 7 until we reach the end of the linked list.
9. Finally, return the next node of the dummy list, which will be the head of the modified list.

Time Complexity:
The time complexity is O(n), where n is the number of nodes in the linked list. We visit each node once during the reversal process.

Space Complexity:
The space complexity is O(1), as we use only a constant amount of extra space throughout the process.
*/

class Solution {
  public:
    ListNode *reverseKGroup(ListNode *head, int k) {
        ListNode *dummy = new ListNode(0);
        dummy->next = head;
        ListNode *current = dummy;

        while (current) {
            ListNode *groupStart = current->next;
            ListNode *groupEnd = current;

            // Find k+1-th node
            for (int i = 0; i < k && groupEnd; i++) {
                groupEnd = groupEnd->next;
            }

            if (!groupEnd) {
                break; // Remaining nodes are less than k
            }

            ListNode *nextGroup = groupEnd->next;
            // Reverse the current k-group
            reverseGroup(groupStart, groupEnd);
            // Connect reversed k-group to the previous group
```

```
            current->next = groupEnd;
            groupStart->next = nextGroup;
            // Move the current pointer to the next group
            current = groupStart;
        }

        return dummy->next;
    }

  private:
    // Helper function to reverse a k-group of nodes
    void reverseGroup(ListNode *head, ListNode *tail) {
        ListNode *prev = nullptr;
        ListNode *current = head;

        while (current != tail) {
            ListNode *nextNode = current->next;
            current->next = prev;
            prev = current;
            current = nextNode;
        }

        tail->next = prev;
    }
};
```

## 46. 07_Trees/01_Invert_Binary_Tree/0226-invert-binary-tree.cpp

```
/*
Problem: LeetCode 226 - Invert Binary Tree

Description:
Given the root of a binary tree, invert the tree and return its root.

Intuition:
To invert a binary tree, we need to swap the left and right subtrees of each node. This can be done recursively, starting from the root node and swapping the children of each node.

Approach:
1. Implement a recursive function to invert the binary tree.
2. If the root is null, return null.
3. Swap the left and right children of the root node.
4. Recursively invert the left and right subtrees.
5. Return the modified root.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We visit each node once.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. This is the space used by the recursive call stack.
*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    TreeNode *invertTree(TreeNode *root) {
        // Base case: if the root is null, return null
        if (root == nullptr) {
            return nullptr;
        }

        // Swap the left and right children of the root node
        swap(root->left, root->right);
        // Recursively invert the left and right subtrees
        invertTree(root->left);
        invertTree(root->right);
        return root;
    }
};
```

## 47. 07_Trees/02_Maximum_Depth_of_Binary_Tree/0104-maximum-depth-of-binary-tre

```
/*
Problem: LeetCode 104 - Maximum Depth of Binary Tree

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

Intuition:
The maximum depth of a binary tree is the number of edges in the longest path from the root node
to any leaf node. We can find the maximum depth by recursively traversing the tree and keeping
track of the depth at each level.

Approach:
1. Implement a recursive function to find the maximum depth of the binary tree.
2. If the root is null, return 0.
3. Recursively calculate the maximum depth of the left subtree.
4. Recursively calculate the maximum depth of the right subtree.
5. Return the maximum depth among the left and right subtrees, plus 1 for the current level.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We
visit each node once.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. This is the space used by
the recursive call stack.
*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    int maxDepth(TreeNode *root) {
        // Base case: if the root is null, return 0
        if (root == nullptr) {
            return 0;
        }

        // Recursively calculate the maximum depth of the left and right subtrees
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
          // Return the maximum depth among the left and right subtrees, plus 1 for the current
level
        return max(leftDepth, rightDepth) + 1;
    }
};


// class Solution {
// public:
//     int maxDepth(TreeNode* root) {
```

```
//         if(!root)
//             return 0;

//         return 1 + max(maxDepth(root->left), maxDepth(root->right));
//     }
// };
```

## 48. 07_Trees/03_Diameter_of_Binary_Tree/0543-diameter-of-binary-tree.cpp

```
/*
Problem: LeetCode 543 - Diameter of Binary Tree

Description:
Given the root of a binary tree, return the length of the diameter of the tree.
The diameter of a binary tree is defined as the length of the longest path between any two nodes in the tree.
This path may or may not pass through the root.

Intuition:
To find the diameter of a binary tree, we need to determine the length of the longest path between any two nodes.
This can be achieved by calculating the maximum depth of each node's left and right subtrees and summing them up.
We can use a recursive approach to traverse the tree and update the diameter as we go.

Approach:
1. Implement a recursive function to find the diameter of the binary tree.
2. If the root is null, return 0 as the diameter.
3. Recursively calculate the maximum depth of the left subtree.
4. Recursively calculate the maximum depth of the right subtree.
5. Update the diameter by taking the maximum of the current diameter and the sum of the maximum depths of the left and right subtrees.
6. Return the maximum depth among the left and right subtrees, plus 1 for the current level.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We visit each node once.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. This is the space used by the recursive call stack.
*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

// Definition for a binary tree node
// struct TreeNode {
//     int val;
//     TreeNode* left;
//     TreeNode* right;
//     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
// };

class Solution {
  public:
    int diameterOfBinaryTree(TreeNode *root) {
        int diameter = 0;
        maxDepth(root, diameter);
        return diameter;
```

```
    }

  private:
    // Depth First Search
    int maxDepth(TreeNode *root, int &diameter) {
        // Base case: if the root is null, return 0
        if (root == nullptr) {
            return 0;
        }

        // Recursively calculate the maximum depth of the left and right subtrees
        int leftDepth = maxDepth(root->left, diameter);
        int rightDepth = maxDepth(root->right, diameter);
         // Update the diameter by taking the maximum of the current diameter and the sum of the
maximum depths of the left and right subtrees
        diameter = max(diameter, leftDepth + rightDepth);
          // Return the maximum depth among the left and right subtrees, plus 1 for the current
level
        return max(leftDepth, rightDepth) + 1;
    }
};
```

## 49. 07_Trees/04_Balanced_Binary_Tree/0110-balanced-binary-tree.cpp

```
/*
Problem: LeetCode 110 - Balanced Binary Tree

Description:
Given a binary tree, determine if it is height-balanced.
A height-balanced binary tree is defined as a binary tree in which the left and right subtrees' heights differ by at most one.

Intuition:
To determine if a binary tree is height-balanced, we need to check if the heights of its left and right subtrees differ by at most one.
We can calculate the height of each subtree recursively and compare the heights at each level to check for balance.

Approach:
1. Implement a recursive function to check if the binary tree is height-balanced.
2. If the root is null, return true as it is considered height-balanced.
3. Recursively calculate the height of the left subtree.
4. Recursively calculate the height of the right subtree.
5. Check if the heights of the left and right subtrees differ by more than one. If so, return false.
6. If the heights are balanced, return true if both the left and right subtrees are also balanced; otherwise, return false.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We visit each node once.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. This is the space used by the recursive call stack.
*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    bool isBalanced(TreeNode *root) {
        return checkHeight(root) != -1;
    }

  private:
    int checkHeight(TreeNode *root) {
        // Base case: if the root is null, return 0
        if (root == nullptr) {
            return 0;
        }

        // Recursively calculate the height of the left and right subtrees
        int leftHeight = checkHeight(root->left);
```

```
        int rightHeight = checkHeight(root->right);

        // If the left or right subtree is not balanced, return -1
        if (leftHeight == -1 || rightHeight == -1) {
            return -1;
        }

        // If the heights of the left and right subtrees differ by more than one, return -1
        if (abs(leftHeight - rightHeight) > 1) {
            return -1;
        }

         // Return the maximum height between the left and right subtrees, plus 1 for the current
level
        return max(leftHeight, rightHeight) + 1;
    }
};
```

## 50. 07_Trees/05_Same_Tree/0100-same-tree.cpp

```
/*
Problem: LeetCode 100 - Same Tree

Description:
Given the roots of two binary trees p and q, write a function to check if they are the same or not.
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Intuition:
To determine if two binary trees are the same, we need to compare their structures and values.
We can use a recursive approach to check if the current nodes of both trees are equal and if their left and right subtrees are also equal.

Approach:
1. Implement a recursive function to check if two binary trees are the same.
2. If both roots are null, return true as they are considered the same.
3. If one of the roots is null and the other is not, or if the values of the roots are different, return false.
4. Recursively check if the left subtrees of both trees are the same.
5. Recursively check if the right subtrees of both trees are the same.
6. Return the logical AND of the above checks.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We visit each node once.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. This is the space used by the recursive call stack.
*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        // Base case: if both roots are null, return true
        if (p == nullptr && q == nullptr) {
            return true;
        }

        // Check if either root is null or their values are different
        if (p == nullptr || q == nullptr || p->val != q->val) {
            return false;
        }

        // Recursively check if the left subtrees and right subtrees are the same
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
```

```
};
```

## 51. 07_Trees/06_Subtree_of_Another_Tree/0572-subtree-of-another-tree.cpp

```
/*
Problem: LeetCode 572 - Subtree of Another Tree

Description:
Given the roots of two binary trees, `s` and `t`, check if `t` is a subtree of `s`.
A subtree of a tree is a tree consisting of a node in `s` and all of its descendants.
The trees `s` and `t` have the same structure, and all the values of the nodes in `t` are also present in `s`.

Intuition:
To check if `t` is a subtree of `s`, we can perform a depth-first search (DFS) on `s` to find a node with the same value as the root of `t`.
Once we find a matching node, we can recursively check if the subtree rooted at that node is identical to `t`.

Approach:
1. Implement a recursive function to check if `t` is a subtree of `s`.
2. If `s` is null, return false as `t` cannot be a subtree of an empty tree.
3. Check if the current node in `s` is identical to `t`. If so, check if the subtree rooted at this node is identical to `t`.
4. If the subtree is identical, return true.
5. If not, recursively check if `t` is a subtree of the left subtree or the right subtree of `s`.
6. Return the logical OR of the above checks.

Time Complexity:
The time complexity of the approach is O(m * n), where m and n are the number of nodes in `s` and `t`, respectively. In the worst case, we may have to compare each node of `s` with `t`.

Space Complexity:
The space complexity is O(max(m, n)), where m and n are the number of nodes in `s` and `t`, respectively. This is the space used by the recursive call stack.

*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    bool isSubtree(TreeNode *s, TreeNode *t) {
        // Base case: if `s` is null, return false as `t` cannot be a subtree of an empty tree
        if (s == nullptr) {
            return false;
        }

        // Check if the current node in `s` is identical to `t`, and if the subtree rooted at this node is identical to `t`
        if (isIdentical(s, t)) {
            return true;
        }
```

```
        // Recursively check if `t` is a subtree of the left subtree or the right subtree of `s`
        return isSubtree(s->left, t) || isSubtree(s->right, t);
    }

  private:
    bool isIdentical(TreeNode *s, TreeNode *t) {
        // Base cases: if either `s` or `t` is null, return true only if both are null
        if (s == nullptr && t == nullptr) {
            return true;
        }

        if (s == nullptr || t == nullptr) {
            return false;
        }

         // Check if the current nodes have the same value and recursively check if their left and
right subtrees are identical
            return (s->val == t->val) && isIdentical(s->left, t->left) && isIdentical(s->right,
t->right);
    }
};
```

## 52. 07_Trees/07_Lowest_Common_Ancestor_of_a_Binary_Search_Tree/0235-lowest-

```
/*
Problem: LeetCode 235 - Lowest Common Ancestor of a Binary Search Tree

Description:
Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.
According to the definition of LCA on Wikipedia: "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)."

Intuition:
The key property of a binary search tree (BST) is that for every node, its left subtree contains values smaller than the node's value, and its right subtree contains values greater than the node's value.
To find the lowest common ancestor (LCA) of two nodes in a BST, we can take advantage of this property to navigate through the tree and determine the LCA based on the values of the nodes.

Approach:
1. Start from the root of the BST.
2. If both `p` and `q` are smaller than the current node, the LCA lies in the left subtree. Recurse on the left subtree.
3. If both `p` and `q` are greater than the current node, the LCA lies in the right subtree. Recurse on the right subtree.
4. If neither of the above conditions is true, then the current node is the LCA.

Time Complexity:
The time complexity of the approach is O(h), where h is the height of the BST. In the worst case, we need to traverse the entire height of the BST.

Space Complexity:
The space complexity is O(h), where h is the height of the BST. This is the space used by the recursive call stack.

*/

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */

class Solution {
  public:
    TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *p, TreeNode *q) {
        // If both `p` and `q` are smaller than the current node, recurse on the left subtree
        if (p->val < root->val && q->val < root->val) {
            return lowestCommonAncestor(root->left, p, q);
        }
        // If both `p` and `q` are greater than the current node, recurse on the right subtree
        else if (p->val > root->val && q->val > root->val) {
            return lowestCommonAncestor(root->right, p, q);
        }
        // If neither of the above conditions is true, return the current node as the LCA
        else {
            return root;
        }
```

```
    }
};
```

## 53. 07_Trees/08_Binary_Tree_Level_Order_Traversal/0102-binary-tree-level-order-tra

```
/*
Problem: LeetCode 102 - Binary Tree Level Order Traversal

Description:
Given the root of a binary tree, return the level order traversal of its nodes' values.
(i.e., from left to right, level by level).

Intuition:
To perform a level order traversal of a binary tree, we can utilize a breadth-first search (BFS)
algorithm.
By visiting the nodes in a breadth-first manner, we can easily track the nodes at each level and
store their values.

Approach:
1. Create a result vector to store the level order traversal.
2. Create a queue to perform the BFS.
3. Enqueue the root node into the queue.
4. While the queue is not empty:
     - Get the current size of the queue to represent the number of nodes at the current level.
     - Create a level vector to store the values of the nodes at the current level.
     - Iterate through the nodes at the current level:
         - Dequeue a node from the queue.
         - Add the value of the dequeued node to the level vector.
         - Enqueue the left and right children of the dequeued node, if they exist.
     - Add the level vector to the result vector.
5. Return the result vector.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We
visit each node once during the BFS.

Space Complexity:
The space complexity is O(m), where m is the maximum number of nodes at any level in the binary
tree. This is the space used by the queue and the result vector.

*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    vector<vector<int>> levelOrder(TreeNode *root) {
        vector<vector<int>> result;

        if (root == nullptr) {
            return result;
        }

        queue<TreeNode *> q;
        q.push(root);
```

```
        while (!q.empty()) {
            int levelSize = q.size();
            vector<int> level;

            for (int i = 0; i < levelSize; i++) {
                // Dequeue a node from the queue
                TreeNode *node = q.front();
                q.pop();
                // Add the value of the dequeued node to the level vector
                level.push_back(node->val);

                // Enqueue the left and right children of the dequeued node, if they exist
                if (node->left != nullptr) {
                    q.push(node->left);
                }

                if (node->right != nullptr) {
                    q.push(node->right);
                }
            }

            // Add the level vector to the result vector
            result.push_back(level);
        }

        return result;
    }
};
```

## 54. 07_Trees/09_Binary_Tree_Right_Side_View/0199-binary-tree-right-side-view.cpp

```
/*
Problem: LeetCode 199 - Binary Tree Right Side View

Description:
Given a binary tree, imagine yourself standing on the right side of it, return the values of the
nodes you can see ordered from top to bottom.

Intuition:
To obtain the right side view of a binary tree, we can perform a level order traversal and keep
track of the last node at each level.
Since we traverse the tree level by level, the last node we encounter at each level from left to
right will be visible from the right side.

Approach:
1. Create a vector, `result`, to store the right side view of the binary tree.
2. Create an empty queue of `TreeNode*` to perform the level order traversal.
3. Enqueue the root node into the queue.
4. While the queue is not empty:
   - Get the current size of the queue to represent the number of nodes at the current level.
   - Create a variable, `lastValue`, to store the value of the last node at the current level.
   - Iterate through the nodes at the current level:
     - Dequeue a node from the queue.
     - Update `lastValue` with the value of the dequeued node.
     - Enqueue the left and right children of the dequeued node, if they exist.
   - Add `lastValue` to the `result` vector.
5. Return the `result` vector containing the right side view.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We
visit each node once during the level order traversal.

Space Complexity:
The space complexity is O(m), where m is the maximum number of nodes at any level in the binary
tree. This is the space used by the queue and the `result` vector.

*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    vector<int> rightSideView(TreeNode *root) {
        vector<int> result;

        if (root == nullptr) {
            return result;
        }

        queue<TreeNode *> q;
        q.push(root);
```

```
        while (!q.empty()) {
            int levelSize = q.size();
            int lastValue;

            for (int i = 0; i < levelSize; i++) {
                TreeNode *node = q.front();
                q.pop();
                lastValue = node->val;

                if (node->left != nullptr) {
                    q.push(node->left);
                }

                if (node->right != nullptr) {
                    q.push(node->right);
                }
            }

            result.push_back(lastValue);
        }

        return result;
    }
};
```

## 55. 07_Trees/10_Count_Good_Nodes_In_Binary_Tree/1448-count-good-nodes-in-bina

```
/*
Problem: LeetCode 1448 - Count Good Nodes in Binary Tree

Description:
Given a binary tree root, a node X in the tree is named good if in the path from root to X there are no nodes with a value greater than X.
Return the number of good nodes in the binary tree.

Intuition:
To count the number of good nodes in a binary tree, we can perform a depth-first search (DFS) and keep track of the maximum value seen so far.
For each node, if its value is greater than or equal to the maximum value seen so far, we increment the count of good nodes.

Approach:
1. Create a helper function, `countGoodNodesHelper`, to perform the DFS traversal and count the good nodes.
2. Initialize a count variable to keep track of the number of good nodes.
3. Start the DFS traversal from the root node with the initial maximum value as negative infinity.
4. In the `countGoodNodesHelper` function:
   - Check if the current node is nullptr. If so, return.
   - Update the maximum value seen so far to be the maximum of the current node's value and the current maximum value.
    - If the current node's value is greater than or equal to the maximum value seen so far, increment the count of good nodes.
   - Recursively call the `countGoodNodesHelper` function for the left and right children of the current node.
5. Return the count of good nodes.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We visit each node once during the DFS traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. This is the space used by the recursive call stack.
*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    int goodNodes(TreeNode *root) {
        return countGoodNodesHelper(root, INT_MIN);
    }

  private:
    int countGoodNodesHelper(TreeNode *node, int maxSoFar) {
        if (node == nullptr) {
            return 0;
```

```
        }

        int count = 0;

        if (node->val >= maxSoFar) {
            count++;
        }

        int newMax = max(node->val, maxSoFar);
        count += countGoodNodesHelper(node->left, newMax);
        count += countGoodNodesHelper(node->right, newMax);
        return count;
    }
};
```

## 56. 07_Trees/11_Validate_Binary_Search_Tree/0098-validate-binary-search-tree.cpp

```
/*
Problem: LeetCode 98 - Validate Binary Search Tree

Description:
Given the root of a binary tree, determine if it is a valid binary search tree (BST).

Intuition:
A binary search tree (BST) is a binary tree in which the value of each node is greater than all the values in its left subtree and less than all the values in its right subtree.
To validate a BST, we can perform an in-order traversal and check if the values are in ascending order.

Approach:
1. Initialize a previous value to store the last visited value during the in-order traversal.
2. Create a helper function, `isValidBSTHelper`, to perform the in-order traversal and validate the BST.
3. In the `isValidBSTHelper` function:
   - Check if the current node is `nullptr`. If so, return true since it does not violate the BST property.
   - Recursively call the `isValidBSTHelper` function for the left subtree. If it returns false, return false.
   - Check if the current node's value is less than or equal to the previous value. If so, return false.
   - Update the previous value to be the current node's value.
   - Recursively call the `isValidBSTHelper` function for the right subtree. If it returns false, return false.
4. Return true if the entire tree has been traversed without any violations.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We visit each node once during the in-order traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. This is the space used by the recursive call stack.
*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    bool isValidBST(TreeNode *root) {
        long long prev = LLONG_MIN; // Use long long to handle edge case with INT_MIN
        return isValidBSTHelper(root, prev);
    }

  private:
    bool isValidBSTHelper(TreeNode *node, long long &prev) {
        if (node == nullptr) {
            return true;
```

```
        }

        if (!isValidBSTHelper(node->left, prev)) {
            return false;
        }

        if (node->val <= prev) {
            return false;
        }

        prev = node->val;
        return isValidBSTHelper(node->right, prev);
    }
};
```

## 57. 07_Trees/12_Kth_Smallest_Element_In_a_BST/0230-kth-smallest-element-in-a-bst

```
/*
Problem: LeetCode 230 - Kth Smallest Element in a BST

Description:
Given the root of a binary search tree (BST), return the kth smallest element in the BST.

Intuition:
In a binary search tree (BST), the left subtree contains smaller elements, and the right subtree contains larger elements.
To find the kth smallest element, we can perform an in-order traversal and keep track of the count of visited nodes.

Approach:
1. Create a helper function, `kthSmallestHelper`, to perform the in-order traversal and find the kth smallest element.
2. Initialize a count variable to keep track of the number of visited nodes.
3. Start the in-order traversal from the root node.
4. In the `kthSmallestHelper` function:
   - Check if the current node is `nullptr`. If so, return -1 to indicate an invalid result.
   - Recursively call the `kthSmallestHelper` function for the left subtree. If the result is not -1, return the result.
   - Increment the count of visited nodes. If the count equals k, return the current node's value.
   - Recursively call the `kthSmallestHelper` function for the right subtree. If the result is not -1, return the result.
5. Return -1 if the kth smallest element is not found.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary search tree. We visit each node once during the in-order traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the binary search tree. This is the space used by the recursive call stack.

*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    int kthSmallest(TreeNode *root, int k) {
        int count = 0;
        return kthSmallestHelper(root, k, count);
    }

  private:
    int kthSmallestHelper(TreeNode *node, int k, int &count) {
        if (node == nullptr) {
            return -1;
        }
```

```
        int result = kthSmallestHelper(node->left, k, count);

        if (result != -1) {
            return result;
        }

        count++;

        if (count == k) {
            return node->val;
        }

        return kthSmallestHelper(node->right, k, count);
    }
};
```

## 58. 07_Trees/13_Construct_Binary_Tree_from_Preorder_and_Inorder_Traversal/0105

```
/*
Problem: LeetCode 105 - Construct Binary Tree from Preorder and Inorder Traversal

Description:
Given two integer arrays `preorder` and `inorder` representing the preorder and inorder traversal
of a binary tree, construct the binary tree.

Intuition:
In a preorder traversal, the root node is visited first, followed by the left subtree and then the
right subtree.
In an inorder traversal, the left subtree is visited first, followed by the root node and then the
right subtree.
We can utilize these properties to construct the binary tree.

Approach:
1. Create a helper function, `buildTreeHelper`, to construct the binary tree recursively.
2. In the `buildTreeHelper` function:
   - Check if the preorder array is empty. If so, return `nullptr`.
   - Extract the root value from the preorder array and create a new node.
   - Find the index of the root value in the inorder array.
   - Split the inorder array into left and right subtrees based on the root index.
   - Recursively call the `buildTreeHelper` function for the left subtree using the corresponding
sections of the preorder and inorder arrays.
   - Recursively call the `buildTreeHelper` function for the right subtree using the corresponding
sections of the preorder and inorder arrays.
   - Assign the left and right subtrees to the root node.
   - Return the root node.
3. Call the `buildTreeHelper` function with the entire preorder and inorder arrays.
4. Return the constructed binary tree.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We
visit each node once during the construction.

Space Complexity:
The space complexity is O(n), where n is the number of nodes in the binary tree. This is the space
used by the recursive call stack.

*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {
        return buildTreeHelper(preorder, inorder, 0, 0, inorder.size() - 1);
    }

  private:
      TreeNode  *buildTreeHelper(vector<int>  &preorder,  vector<int>  &inorder,  int  preStart,  int
```

```
inStart, int inEnd) {
        if (preStart >= preorder.size() || inStart > inEnd) {
            return nullptr;
        }

        int rootVal = preorder[preStart];
        TreeNode *root = new TreeNode(rootVal);
        int rootIndex;

        for (int i = inStart; i <= inEnd; i++) {
            if (inorder[i] == rootVal) {
                rootIndex = i;
                break;
            }
        }

        int leftSize = rootIndex - inStart;
        root->left = buildTreeHelper(preorder, inorder, preStart + 1, inStart, rootIndex - 1);
         root->right = buildTreeHelper(preorder, inorder, preStart + leftSize + 1, rootIndex + 1,
inEnd);
        return root;
    }
};
```

## 59. 07_Trees/14_Binary_Tree_Maximum_Path_Sum/0124-binary-tree-maximum-path-s

```
/*
Problem: LeetCode 124 - Binary Tree Maximum Path Sum

Description:
Given a non-empty binary tree, find the maximum path sum.
A path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections.
The path must contain at least one node and does not need to go through the root.

Intuition:
The maximum path sum can be found by considering the maximum sum path for each node in the binary tree.
A path can include nodes from the left subtree, the current node, and nodes from the right subtree.
We need to keep track of the maximum sum encountered so far as we traverse the tree.

Approach:
1. Create a helper function, `maxPathSumHelper`, to calculate the maximum path sum for each node.
2. In the `maxPathSumHelper` function:
   - Check if the current node is `nullptr`. If so, return 0 to represent an empty path.
   - Recursively call the `maxPathSumHelper` function for the left subtree and store the result in `leftSum`.
   - Recursively call the `maxPathSumHelper` function for the right subtree and store the result in `rightSum`.
   - Calculate the maximum sum path that includes the current node:
     - Calculate `maxChildSum` as the maximum between `leftSum` and `rightSum`, or 0 if they are negative.
     - Calculate `maxSum` as the maximum between `leftSum + rightSum + node->val` and `node->val`.
   - Update the maximum sum encountered so far by comparing `maxSum` with the current maximum.
     - Return the maximum sum path that includes the current node by adding `node->val` to `maxChildSum`.
3. Call the `maxPathSumHelper` function with the root node.
4. Return the maximum sum encountered during the traversal.

Time Complexity:
The time complexity of the approach is O(n), where n is the number of nodes in the binary tree. We visit each node once during the traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. This is the space used by the recursive call stack.

*/

/**
 * Definition for a binary tree node.
 * 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) {}
 * };
 */

class Solution {
  public:
    int maxPathSum(TreeNode *root) {
        int maxSum = INT_MIN;
```

```
        maxPathSumHelper(root, maxSum);
        return maxSum;
    }

  private:
    int maxPathSumHelper(TreeNode *node, int &maxSum) {
        if (node == nullptr) {
            return 0;
        }

        // Recursively calculate the maximum path sum for the left and right subtrees
        int leftSum = maxPathSumHelper(node->left, maxSum);
        int rightSum = maxPathSumHelper(node->right, maxSum);
        // Calculate the maximum sum path that includes the current node
        int maxChildSum = max(max(leftSum, rightSum), 0);
        int maxSumWithNode = max(maxChildSum + node->val, leftSum + rightSum + node->val);
        maxSum = max(maxSum, maxSumWithNode);
         // Return the maximum sum path that includes the current node by adding it to the maximum
child sum
        return max(maxChildSum + node->val, node->val);
    }
};
```

## 60. 07_Trees/15_Serialize_and_Deserialize_Binary_Tree/0297-serialize-and-deserializ

```
/*
Problem: LeetCode 297 - Serialize and Deserialize Binary Tree

Description:
Design an algorithm to serialize and deserialize a binary tree. Serialize means to encode a tree
structure into a string representation, and deserialize means to decode the string representation
back into a binary tree structure. The encoded string should be as compact as possible.

Intuition:
To serialize a binary tree, we can perform a pre-order traversal and encode the tree structure
into a string. We use a special character to represent nullptr or empty nodes. During
deserialization, we can use the encoded string to reconstruct the binary tree.

Approach:
1. Serialization:
   - Perform a pre-order traversal of the binary tree.
   - When encountering a non-empty node, append its value to the serialized string, followed by a
separator.
     - When encountering a nullptr or empty node, append a special character (e.g., 'N') to
represent it.
   - Use a separator character (e.g., ',') to separate the values in the serialized string.

2. Deserialization:
   - Split the serialized string by the separator to obtain an array of values.
   - Create a queue to hold the values from the array.
   - Recursively build the binary tree using a helper function:
     - If the queue is empty or the current value is the special character, return nullptr.
     - Create a new node with the current value.
     - Pop the queue to move to the next value.
     - Set the left child of the current node by recursively calling the helper function.
     - Set the right child of the current node by recursively calling the helper function.
     - Return the current node.
   - Call the helper function with the queue to build the binary tree.
   - Return the root of the binary tree.

Time Complexity:
- Serialization: O(n), where n is the number of nodes in the binary tree. We perform a pre-order
traversal to serialize the tree.
- Deserialization: O(n), where n is the number of nodes in the binary tree. We iterate through the
serialized string to deserialize and reconstruct the tree.

Space Complexity:
- Serialization: O(n), where n is the number of nodes in the binary tree. The serialized string
requires space proportional to the number of nodes.
- Deserialization: O(n), where n is the number of nodes in the binary tree. We use a queue to
store the values during deserialization.

*/

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */

class Codec {
```

```
  public:
    string serialize(TreeNode *root) {
        string serialized = "";
        PreOrder(root, serialized);
        return serialized;
    }

    TreeNode *deserialize(string data) {
        queue<string> Q;
        split(data, Q);
        return MakeTree(Q);
    }

  private:
    void PreOrder(TreeNode *root, string &str) {
        if (!root) {
            str.push_back('N');
            str.push_back(',');
            return;
        }

        str += to_string(root->val) + ",";
        PreOrder(root->left, str);
        PreOrder(root->right, str);
    }

    TreeNode *MakeTree(queue<string> &Q) {
        string S = Q.front();
        Q.pop();

        if (S == "N") {
            return nullptr;
        }

        // stoi -> string to integer
        TreeNode *root = new TreeNode(stoi(S));
        root->left = MakeTree(Q);
        root->right = MakeTree(Q);
        return root;
    }

    void split(const string &data, queue<string> &Q) {
        size_t start = 0;
        size_t pos = data.find(",");

        while (pos != string::npos) {
            Q.push(data.substr(start, pos - start));
            start = pos + 1;
            pos = data.find(",", start);
        }
    }
};


// class Codec {
// public:
//     // Encodes a tree to a single string.
//     string serialize(TreeNode* root) {
//         string serialized = "";
//         PreOrder(root, serialized);
//         return serialized;
//     }

//     // Decodes your encoded data to tree.
```

```
//     TreeNode* deserialize(string data) {
//         queue<string> Q;
//         string S;
//         for(int i = 0; i < data.size(); i++) {
//             // If it's a comma, push string into queue and reset string
//             if(data[i] == ',') {
//                 Q.push(S);
//                 S = "";
//                 // Continuing so we go to next iteration without pushing comma
//                 continue;
//             }
//             // pushing back char to string if its not a comma
//             S.push_back(data[i]);
//         }
//         // Making tree after decoding
//         return MakeTree(Q);
//     }

// private:
//     // Function to make string using pre-order traversal
//     void PreOrder(TreeNode* root, string &str) {
//         // If root is null then insert N into string
//         if(!root) {
//             str.push_back('N');
//             str.push_back(',');
//             return;
//         }
//         // Converting int to string and appending
//         // Another way of appending
//         str += to_string(root->val) + ",";

//         PreOrder(root->left, str);
//         PreOrder(root->right, str);
//     }

//     TreeNode* MakeTree(queue<string> &Q) {
//         string S = Q.front();
//         Q.pop();

//         if(S == "N")
//             return NULL;

//         // stoi -> string to integer
//         TreeNode* root = new TreeNode(stoi(S));

//         root->left = MakeTree(Q);
//         root->right = MakeTree(Q);
//         return root;
//     }
// };
```

# 61. 08_Tries/01_Implement_Trie_Prefix_Tree/0208-implement-trie-prefix-tree.cpp

```
/*
Problem: LeetCode 208 - Implement Trie (Prefix Tree)

Description:
Implement a trie with insert, search, and startsWith methods.

Intuition:
A trie, also known as a prefix tree, is a tree-like data structure that stores a set of strings. Each node in the trie represents a prefix or a complete word. The trie allows efficient insertion, search, and prefix matching operations.

Approach:
1. TrieNode:
   - Define a TrieNode class that represents each node in the trie.
   - Each TrieNode has an array of pointers to child nodes, representing the 26 lowercase letters of the English alphabet.
   - Each TrieNode also has a boolean flag to indicate if it represents a complete word.

2. Trie:
   - Define a Trie class that contains the root of the trie.
   - Implement the insert method to insert a word into the trie:
     - Start from the root and iterate over each character in the word.
     - For each character, check if the corresponding child node exists. If not, create a new node and link it to the current node.
     - Move to the child node and repeat the process for the next character.
     - After iterating through all characters, mark the last node as a complete word.
   - Implement the search method to search for a word in the trie:
     - Start from the root and iterate over each character in the word.
     - For each character, check if the corresponding child node exists. If not, the word is not in the trie.
     - Move to the child node and repeat the process for the next character.
     - After iterating through all characters, check if the last node represents a complete word.
   - Implement the startsWith method to check if there is any word in the trie that starts with the given prefix:
     - Start from the root and iterate over each character in the prefix.
     - For each character, check if the corresponding child node exists. If not, there are no words with the given prefix.
     - Move to the child node and repeat the process for the next character.
     - After iterating through all characters, return true, indicating that there are words with the given prefix.

Time Complexity:
- Insert: O(m), where m is the length of the word being inserted.
- Search: O(m), where m is the length of the word being searched.
- StartsWith: O(m), where m is the length of the prefix being checked.

Space Complexity:
- The space complexity is O(n*m), where n is the number of words inserted into the trie and m is the average length of the words.

*/

class TrieNode {
  public:
    bool isWord;
    TrieNode *children[26];

    TrieNode() {
        isWord = false;
```

```
        for (int i = 0; i < 26; i++) {
            children[i] = nullptr;
        }
    }
};

class Trie {
  private:
    TrieNode *root;

  public:
    Trie() {
        root = new TrieNode();
    }

    void insert(string word) {
        TrieNode *node = root;

        for (char c : word) {
            int index = c - 'a';

            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }

            node = node->children[index];
        }

        node->isWord = true;
    }

    bool search(string word) {
        TrieNode *node = root;

        for (char c : word) {
            int index = c - 'a';

            if (!node->children[index]) {
                return false;
            }

            node = node->children[index];
        }

        return node->isWord;
    }

    bool startsWith(string prefix) {
        TrieNode *node = root;

        for (char c : prefix) {
            int index = c - 'a';

            if (!node->children[index]) {
                return false;
            }

            node = node->children[index];
        }

        return true;
    }
};
```

```
/*
class Trie {
private:
    // Defining TrieNode Datatype
    struct TrieNode {
        // Can have 26 diffenet childen because thats the amt of alphabets
        TrieNode *child[26];
        // To check where the word ends
        bool isWord;

        //Constructor to initialise values
        TrieNode() {
            isWord = false;
            for (auto &c : child)
                c = nullptr;
        }
    };
    // Root pointer
    TrieNode *root;

public:
    Trie() {
        root = new TrieNode();
    }

    void insert(string word) {
        // Pointer to point the character we insert in trie
        TrieNode* current = root;
        for(auto i: word) {
            // index of the character i.e. a = 0, z = 25
            int index = i - 'a';
            if(current->child[index] == NULL)
                current->child[index] = new TrieNode();

            // Pointing current to the character we just inserted
            current = current->child[index];
        }
        current->isWord = true;
    }

    bool search(string word) {
        TrieNode *current = root;
        for(auto i: word) {
            int index = i - 'a';

            // If the child isn't there, return false. Else keep going.
            if(current->child[index] == NULL)
                return false;

            current = current->child[index];
        }
        // Will return true if it's a word
        return current->isWord;
    }

    // Same code as search but here we dont need to check if its a word
    bool startsWith(string prefix) {
        TrieNode *current = root;
        for(auto i: prefix) {
            int index = i - 'a';

            // If the child isn't there, return false. Else keep going.
            if(current->child[index] == NULL)
                return false;
```

```
            current = current->child[index];
        }
        return true;
    }
};
*/
```

## 62. 08_Tries/02_Design_Add_and_Search_Words_Data_Structure/0211-design-add-a

```
/*
Problem: LeetCode 211 - Design Add and Search Words Data Structure

Description:
Design a data structure that supports adding new words and finding if a string matches any previously added string.
The word may contain only lowercase alphabets '.' or be an empty string.

Intuition:
To solve this problem, we can use a Trie data structure to store the words. The Trie allows efficient insertion and search operations. For the '.' character, we need to consider all possible characters at that position.

Approach:
1. TrieNode:
   - Define a TrieNode class that represents each node in the Trie.
   - Each TrieNode has an array of pointers to child nodes, representing the lowercase alphabets and the '.' character.
   - Each TrieNode also has a boolean flag to indicate if it represents a complete word.

2. WordDictionary:
   - Define a WordDictionary class that contains the root of the Trie.
   - Implement the addWord method to add a word to the Trie:
     - Start from the root and iterate over each character in the word.
     - For each character, check if the corresponding child node exists. If not, create a new node and link it to the current node.
     - Move to the child node and repeat the process for the next character.
     - After iterating through all characters, mark the last node as a complete word.
   - Implement the search method to search for a word in the Trie:
     - Start from the root and iterate over each character in the word.
     - For each character, check if the corresponding child node exists. If not, return false.
     - Move to the child node and repeat the process for the next character.
     - After iterating through all characters, check if the last node represents a complete word.
   - Implement the searchWithWildcard method to search for a word with wildcard characters ('.') in the Trie:
     - Use a recursive approach to search for the word.
      - If the current character is a wildcard ('.'), iterate over all possible child nodes and recursively search for the remaining word.
      - If the current character is not a wildcard, check if the corresponding child node exists and recursively search for the remaining word.
     - Return true if any of the recursive searches return true.
     - Return false if no matching word is found.

Time Complexity:
- Adding a word: O(m), where m is the length of the word being added.
- Searching a word: O(m), where m is the length of the word being searched.
- Searching a word with wildcard: O(n*m), where n is the number of words in the Trie and m is the length of the word being searched.

Space Complexity:
- The space complexity is O(n*m), where n is the number of words added to the Trie and m is the average length of the words.

*/

class TrieNode {
  public:
    bool isWord;
    TrieNode *children[26];
```

```
    TrieNode() {
        isWord = false;

        for (int i = 0; i < 26; i++) {
            children[i] = nullptr;
        }
    }
};

class WordDictionary {
  private:
    TrieNode *root;

  public:
    WordDictionary() {
        root = new TrieNode();
    }

    void addWord(string word) {
        TrieNode *node = root;

        for (char c : word) {
            int index = c - 'a';

            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }

            node = node->children[index];
        }

        node->isWord = true;
    }

    bool search(string word) {
        return searchHelper(word, root, 0);
    }

    bool searchHelper(string word, TrieNode *node, int index) {
        if (index == word.length()) {
            return node->isWord;
        }

        char c = word[index];

        if (c != '.') {
            int childIndex = c - 'a';

            if (node->children[childIndex]) {
                return searchHelper(word, node->children[childIndex], index + 1);
            } else {
                return false;
            }
        } else {
            for (int i = 0; i < 26; i++) {
                if (node->children[i] && searchHelper(word, node->children[i], index + 1)) {
                    return true;
                }
            }

            return false;
        }
    }
};
```

```
/*
class WordDictionary {
private:
    struct TrieNode {
        TrieNode* child[26];
        bool isWord;

        TrieNode() {
            for(auto &i: child)
                i = nullptr;
            isWord = false;
        }
    };
    TrieNode* root;

    bool searchInNode(string& word, int i, TrieNode* node) {
        if (node == NULL)
            return false;

        if (i == word.size())
            return node->isWord;

        // if its an alphabet and not .
        if (word[i] != '.')
            return searchInNode(word, i + 1, node->child[word[i] - 'a']);

        // If the current character is a dot, we need to check all children of the current node
         // recursively by skipping over the dot character and moving to the next character of the
word
        for (int j = 0; j < 26; j++)
            if (searchInNode(word, i + 1, node->child[j]))
                return true;

        return false;
    }

public:
    WordDictionary() {
        root = new TrieNode();
    }

    void addWord(string word) {
        TrieNode *current = root;

        for(auto c: word) {
            int i = c - 'a';
            if(!current->child[i])
                current->child[i] = new TrieNode();

            current = current->child[i];
        }
        current->isWord = true;
    }

    bool search(string word) {
        TrieNode* node = root;
        return searchInNode(word, 0, node);
    }
};
*/
```

## 63. 08_Tries/03_Word_Search_II/0212-word-search-ii.cpp

```
/*
Problem: LeetCode 212 - Word Search II

Description:
Given an m x n board of characters and a list of words, find all words in the board.
Each word must be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring.
The same letter cell may not be used more than once in a word.

Intuition:
To find all the words in the board, we can use the Trie data structure to efficiently search for each word. We perform a depth-first search (DFS) starting from each cell on the board, checking if the current sequence of characters forms a valid word in the Trie.

Approach:
1. TrieNode:
   - Define a TrieNode class that represents each node in the Trie.
    - Each TrieNode has an unordered_map to store the child nodes, representing the lowercase alphabets.
   - Each TrieNode also has a boolean flag to indicate if it represents a complete word.

2. Build the Trie:
   - Construct a Trie by inserting each word from the given list into the Trie.

3. DFS Search:
   - Perform a depth-first search (DFS) starting from each cell on the board.
   - At each cell, check if the current character exists in the Trie and move to the corresponding child node.
   - Mark the current cell as visited.
   - If the current node represents a complete word, add it to the result.
   - Recursively explore the neighboring cells (up, down, left, right).
    - Backtrack by unmarking the current cell and removing the last character from the current sequence.

4. Word Search II:
   - Initialize an empty vector to store the found words.
   - Iterate through each cell on the board and perform the DFS search.
   - Return the found words as the result.

Time Complexity:
- Building the Trie: O(m), where m is the total number of characters in all words.
- DFS Search: O((m*n)*3^l), where m and n are the dimensions of the board and l is the average length of the words.

Space Complexity:
- The space complexity is O(m), where m is the total number of characters in all words (used for constructing the Trie).
- The space complexity of the recursive stack for DFS is O(l), where l is the maximum length of the words.
*/

class TrieNode {
  public:
    bool isWord;
    unordered_map<char, TrieNode *> children; // Map to store the child nodes

    TrieNode() {
        isWord = false;
    }
};
```

```
class Solution {
  public:
    vector<string> findWords(vector<vector<char>> &board, vector<string> &words) {
        TrieNode *root = buildTrie(words); // Build the Trie
        int rows = board.size();
        int cols = board[0].size();
        vector<string> result;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                string currentWord = ""; // Initialize the current word for each cell
                dfs(board, i, j, root, currentWord, result); // Perform DFS search
            }
        }

        return result;
    }

  private:
    TrieNode *buildTrie(vector<string> &words) {
        TrieNode *root = new TrieNode();

        for (string &word : words) {
            TrieNode *node = root;

            for (char c : word) {
                if (node->children.find(c) == node->children.end()) {
                    node->children[c] = new TrieNode();
                }

                node = node->children[c];
            }

            node->isWord = true;
        }

        return root;
    }

     void dfs(vector<vector<char>> &board, int row, int col, TrieNode *node, string &currentWord,
vector<string> &result) {
        if (row < 0 || row >= board.size() || col < 0 || col >= board[0].size() || board[row][col]
== '#') {
            return;
        }

        char c = board[row][col];

        if (node->children.find(c) == node->children.end()) {
            return;
        }

        node = node->children[c];
        currentWord += c;

        if (node->isWord) {
            result.push_back(currentWord); // Add the found word to the result
            node->isWord = false; // Mark the word as visited
        }

        board[row][col] = '#'; // Mark the current cell as visited
        // Explore the neighboring cells (up, down, left, right)
        dfs(board, row - 1, col, node, currentWord, result);
```

```
        dfs(board, row + 1, col, node, currentWord, result);
        dfs(board, row, col - 1, node, currentWord, result);
        dfs(board, row, col + 1, node, currentWord, result);
        board[row][col] = c; // Backtrack: unmark the current cell
        currentWord.pop_back(); // Remove the current character from the current word
    }
};
```

## 64. 09_Heap_Priority_Queues/01_Kth_Largest_Element_in_a_Stream/0703-kth-larges

```
/*
Problem: LeetCode 703 - Kth Largest Element in a Stream

Description:
Design a class to find the kth largest element in a stream.
Note that it is the kth largest element in the sorted order, not the kth distinct element.

Intuition:
To find the kth largest element in a stream efficiently, we can use a min-heap of size k.
As new elements are added to the stream, we compare them with the root of the min-heap.
If the new element is larger than the root, we replace the root with the new element and perform
heapify to maintain the heap property.

Approach:
1. Implement a class KthLargest with the following members:
    - A min-heap to store the k largest elements. Use a priority_queue in C++ with the smallest
element on top.
   - A variable k to store the value of k.

2. In the constructor of KthLargest:
   - Initialize the variable k.
   - Iterate through the given vector of integers and add each element to the min-heap.
   - If the size of the min-heap exceeds k, remove the smallest element from the heap.

3. In the add function:
   - If the size of the min-heap is less than k, simply add the new element to the heap.
    - If the new element is larger than the root of the min-heap, replace the root with the new
element and perform heapify.
   - Return the value of the root of the min-heap, which represents the kth largest element.

Time Complexity:
- Construction: O(n*log(k)), where n is the number of elements in the input vector.
- Adding an element: O(log(k)), as we need to perform heapify after adding an element.

Space Complexity:
- The space complexity is O(k), as we store the k largest elements in the min-heap.
*/

// static int pr = []() {
//     std::ios::sync_with_stdio(false);
//     cin.tie(NULL);
//     return 0;
// }();

class KthLargest {
  private:
    int k;
     priority_queue<int, vector<int>, greater<int>> minHeap; // Min-heap to store the k largest
elements

  public:
    KthLargest(int k, vector<int> &nums) {
        this->k = k;

        for (int num : nums) {
            minHeap.push(num);

            // removing elements till k elements remain
            if (minHeap.size() > k) {
                minHeap.pop();
```

```
            }
        }
    }

    int add(int val) {
        minHeap.push(val);

        if (minHeap.size() > k) {
            minHeap.pop();
        }

        return minHeap.top();
    }
};
```

## 65. 09_Heap_Priority_Queues/02_Last_Stone_Weight/1046-last-stone-weight.cpp

```
/*
Problem: LeetCode 1046 - Last Stone Weight

Description:
You are given an array `stones` where `stones[i]` represents the weight of the ith stone.
In each turn, you choose the two heaviest stones and smash them together.
If the stones have the same weight, they both get destroyed, and if they have different weights,
the heavier stone gets destroyed and the lighter stone's weight is reduced by the difference.
When only one stone remains, return its weight. If no stones remain, return 0.

Intuition:
To find the last stone weight, we can use a max-heap to keep track of the heaviest stones.
In each turn, we remove the two largest stones from the heap, calculate their difference, and add
the result back to the heap.
We repeat this process until the heap contains only one stone.

Approach:
1. Implement a function `lastStoneWeight` that takes the input array `stones` as a parameter.
2. Create a max-heap using `priority_queue<int>` in C++ to store the stone weights. The largest
element will always be at the top of the heap.
3. Populate the max-heap with the elements from the `stones` array.
4. While the heap has more than one stone:
   - Remove the two largest stones from the heap using `pop()`.
   - Calculate the difference between the two stones.
   - If the difference is non-zero, add it back to the heap using `push()`.
5. After the loop ends, the heap will contain only one stone.
   - If the heap is empty, return 0 as no stones remain.
   - If the heap is not empty, return the top element of the heap, which represents the last stone
weight.

Time Complexity:
- Building the max-heap: O(n), where n is the number of elements in the input array.
- Performing heap operations: O(log(n)) per operation.
- Overall time complexity: O(n*log(n)), where n is the number of elements in the input array.

Space Complexity:
- The space complexity is O(n), where n is the number of elements in the input array, for storing
the max-heap.
*/

class Solution {
  public:
    int lastStoneWeight(vector<int> &stones) {
        priority_queue<int> maxHeap; // Max-heap to store the stone weights

        // Populate the max-heap with the elements from the 'stones' array
        for (int stone : stones) {
            maxHeap.push(stone);
        }

        while (maxHeap.size() > 1) {
            int stone1 = maxHeap.top(); // Get the heaviest stone
            maxHeap.pop();
            int stone2 = maxHeap.top(); // Get the second heaviest stone
            maxHeap.pop();
            int diff = stone1 - stone2; // Calculate the difference

            if (diff > 0) {
                maxHeap.push(diff); // Add the difference back to the heap
            }
```

```
        }

        if (maxHeap.empty()) {
            return 0; // No stones remain
        }

        return maxHeap.top(); // Return the last stone weight
    }
};
```

## 66. 09_Heap_Priority_Queues/03_K_Closest_Points_to_Origin/0973-k-closest-points-

```
/*
Problem: K Closest Points to Origin

Description:
Given an array points representing coordinates of N points on a 2D plane, and an integer k, you need to return the k closest points to the origin (0, 0).

Intuition:
To find the K closest points to the origin, we can utilize the concept of a priority queue (min-heap).
We iterate through the points and calculate the distance of each point from the origin.
We add the points to the priority queue, and if the size of the queue exceeds K, we remove the farthest point.
In the end, the priority queue will contain the K closest points to the origin.
Approach:

1.Create a vector distances to store pairs of distances and indices of points.
2.Iterate through each point in the points vector and calculate its distance from the origin using the formula distance = x^2 + y^2, where (x, y) are the coordinates of the point.
3. Store each distance along with its corresponding index in the distances vector.
4. Use the nth_element function to find the k-th smallest distance in the distances vector. This function partially sorts the vector, placing the k-th smallest element in its correct position. Elements before it are smaller or equal, while elements after it are greater.
5. Create a result vector result to store the k closest points.
6. Iterate through the first k elements of the distances vector. For each element, retrieve the index and use it to access the corresponding point in the points vector. Add this point to the result vector.
7. Return the result vector containing the k closest points to the origin.

Time Complexity:
The time complexity of this approach is O(N log k), where N is the number of points. Calculating the distance for each point takes O(N), and the nth_element function takes O(N log k).

Space Complexity:
The space complexity is O(N), where N is the number of points. We use the distances vector to store the distances and indices of points
*/

/*
* Syntax for PQ: priority_queue<data_type, container, comparator> ds;
* Data_type(mandatory) : datatype that we are going to store in priority_queue. (int, float, or even any custom datatype)
* Container(optional) : Container is passed as an underlying container to store the elements. It needs to satisfy some properties, therefore it can be either vector<datatype> or deque<datatype>.
* Comparator(optional) : Comparator decides the ordering of elements.
*/

class Solution {
  public:
    vector<vector<int>> kClosest(vector<vector<int>> &points, int k) {
        // Create a vector to store the distances and indices of points
        vector<pair<int, int>> distances(points.size());

        // Calculate the distance of each point from the origin and store it along with its index
        for (int i = 0; i < points.size(); i++) {
            int distance = points[i][0] * points[i][0] + points[i][1] * points[i][1];
            distances[i] = make_pair(distance, i);
        }

        // Find the k-th smallest distance using nth_element
```

```
        nth_element(distances.begin(), distances.begin() + k - 1, distances.end());
        // Create a result vector to store the k closest points
        vector<vector<int>> result(k);

        // Add the k closest points to the result vector
        for (int i = 0; i < k; i++) {
            int index = distances[i].second;
            result[i] = move(points[index]);
        }

        return result;
    }
};
```

## 67. 09_Heap_Priority_Queues/04_Kth_Largest_Element_in_an_Array/0215-kth-larges

```
/*
Problem: LeetCode 215 - Kth Largest Element in an Array

Description:
Given an integer array `nums` and an integer `k`, return the kth largest element in the array.
Note that it is the kth largest element in the sorted order, not the kth distinct element.

Intuition:
The code uses the Quickselect algorithm to find the kth largest element in the array efficiently.
Quickselect is a variation of the QuickSort algorithm and works by partitioning the array based on a chosen pivot element.
Instead of sorting both sides of the partition like in QuickSort, Quickselect only focuses on the side of the partition that contains the desired kth largest element.

Approach:
1. The code implements the Quickselect algorithm to efficiently find the kth largest element.
   - Quickselect is a variation of the QuickSort algorithm that focuses on the desired kth largest element during partitioning.

2. The `quickselect` function is a helper function that takes the input array `nums`, start index `l`, end index `r`, and the value of `k` as parameters.
   - It returns the kth largest element in the array by recursively partitioning the array.

3. Base case:
   - If the start index `l` is greater than or equal to the end index `r`, it means we have found the kth largest element, so we return the element at index `l`.

4. Partitioning:
   - The code chooses the pivot element as `nums[l + r >> 1]`, which is the element at the middle index between `l` and `r`.
   - It initializes two pointers, `i` and `j`, where `i` starts from `l - 1` and `j` starts from `r + 1`.
    - The code enters a loop where it increments `i` until it finds an element greater than the pivot and decrements `j` until it finds an element smaller than the pivot.
   - If `i` is less than `j`, it means there are elements on the wrong side of the partition, so it swaps `nums[i]` and `nums[j]`.

5. After the loop:
   - The code calculates the size of the smaller partition as `sl = j - l + 1`.

6. Recursion:
    - If `k` is less than or equal to `sl`, it means the desired element is in the smaller partition, so it recursively calls `quickselect` on that partition.
   - If `k` is greater than `sl`, it means the desired element is in the larger partition, so it recursively calls `quickselect` on that partition, adjusting `k` by subtracting `sl`.

7. The `findKthLargest` function is the main function that takes the input array `nums` and integer `k` as parameters.
   - It disables synchronization between C and C++ standard streams for faster I/O.
   - It returns the result of the `quickselect` function called with `nums`, 0 as the start index, `nums.size() - 1` as the end index, and `k`.

Time Complexity:
- On average, the Quickselect algorithm has a time complexity of O(n), where n is the number of elements in the input array.
- However, in the worst case, Quickselect can have a time complexity of O(n^2) if the pivot selection is unbalanced.
- Overall, the average time complexity of Quickselect is O(n), making it an efficient algorithm for finding the kth largest element.
```

```
Space Complexity:
- The space complexity is O(log n) for the recursive call stack of the `quickselect` function, where n is the number of elements in the input array.
- In addition to the input array, the algorithm uses a constant amount of extra space.
- Therefore, the overall space complexity is O(log n).
*/

class Solution {
  public:
    int quickselect(vector<int> &nums, int left, int right, int k) {
        // Base case: if the left and right indices are the same, return the element at that index
        if (left >= right) {
            return nums[left];
        }

        // Choose a pivot element
        int pivot = nums[left + (right - left) / 2];
        // Initialize two pointers, one from the left and one from the right
        int i = left - 1;
        int j = right + 1;

        // Partition the array around the pivot
        while (i < j) {
            // Find the first element greater than the pivot from the left side
            while (nums[++i] > pivot);

            // Find the first element smaller than the pivot from the right side
            while (nums[--j] < pivot);

            // Swap the elements if the pointers haven't crossed each other
            if (i < j) {
                swap(nums[i], nums[j]);
            }
        }

        // Calculate the size of the smaller partition
        int smallerPartitionSize = j - left + 1;

          // If the desired element is in the smaller partition, recursively call quickselect on that partition
        if (k <= smallerPartitionSize) {
            return quickselect(nums, left, j, k);
        }

         // If the desired element is in the larger partition, recursively call quickselect on that partition
        return quickselect(nums, j + 1, right, k - smallerPartitionSize);
    }

    int findKthLargest(vector<int> &nums, int k) {
        // Disable synchronization between C and C++ standard streams for faster I/O
        cin.tie(0);
        ios::sync_with_stdio(false);
        // Call the quickselect algorithm to find the kth largest element
        return quickselect(nums, 0, nums.size() - 1, k);
    }
};
```

## 68. 09_Heap_Priority_Queues/05_Task_Scheduler/0621-task-scheduler.cpp

```
/*
Problem: LeetCode 621 - Task Scheduler

Description:
Given a characters array tasks, representing the tasks a CPU needs to do, where each character represents a different task. Tasks could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU could complete either one task or just be idle.
However, there is a non-negative integer n that represents the cooldown period between two same tasks (the same task cannot be executed in adjacent units of time).
Return the least number of units of times that the CPU will take to finish all the given tasks.

Intuition:
To minimize the overall time, we need to arrange the tasks in such a way that the maximum number of occurrences of any task is spread apart by the cooldown period. We can then fill the gaps with idle cycles if needed.

Approach:
1. Count the frequency of each task and store it in a frequency array.
2. Sort the frequency array in descending order.
3. Find the maximum frequency `maxFreq`.
4. Calculate the number of idle cycles required:
    - Subtract 1 from `maxFreq` to exclude the last occurrence of the most frequent task (as it doesn't need an idle cycle after it).
   - Multiply `maxFreq - 1` by `n` to get the number of slots occupied by the most frequent task and its cooldown periods.
    - Subtract this value from the total number of tasks to get the number of remaining idle cycles.
5. Calculate the minimum number of time units required:
   - Add the total number of tasks to the number of idle cycles calculated in step 4.
   - Return the maximum of this value and the length of the tasks array.

Time Complexity:
The time complexity is O(n log n), where n is the number of tasks. This is due to the sorting operation on the frequency array.

Space Complexity:
The space complexity is O(1) since the frequency array has a fixed size of 26 (assuming tasks only contain uppercase letters).
*/

class Solution {
  public:
    int leastInterval(vector<char> &tasks, int n) {
        // Count the frequency of each task
        vector<int> frequency(26, 0);

        for (char task : tasks) {
            frequency[task - 'A']++;
        }

        // Sort the frequency array in descending order
        sort(frequency.rbegin(), frequency.rend());
        // Find the maximum frequency
        int maxFreq = frequency[0];
        // Calculate the number of idle cycles required
        int idleCycles = (maxFreq - 1) * n;

        // Subtract the remaining tasks from the idle cycles
        for (int i = 1; i < frequency.size(); i++) {
            idleCycles -= min(frequency[i], maxFreq - 1);
```

```
        }

        // Calculate the minimum number of time units required
        int minTime = tasks.size() + max(0, idleCycles);
        return minTime;
    }
};
```

## 69. 09_Heap_Priority_Queues/06_Design_Twitter/0355-design-twitter.cpp

```
/*
Problem: LeetCode 355 - Design Twitter

Description:
Design a simplified version of Twitter where users can post tweets, follow/unfollow another user,
and see the 10 most recent tweets in the user's news feed.

Intuition:
To design Twitter, we need to efficiently handle the following operations:
1. Posting a tweet: We need to store the tweets along with their timestamps.
2. Following/Unfollowing a user: We need to maintain a data structure to track the followers and
followees of each user.
3. Retrieving the news feed: We need to combine the tweets from the user's own timeline along with
the tweets from the users they follow.

Approach:
1. Implement the `User` class to store the user's information, including their tweets and the
users they follow.
2. Use an unordered_map to store the user ID as the key and the corresponding `User` object as the
value.
3. Implement the `Tweet` class to store the tweet's information, including the tweet ID and the
timestamp.
4. Use a deque to store the tweets in the user's timeline, with the most recent tweet at the
front.
5. To post a tweet:
   - Get the `User` object corresponding to the user ID.
   - Create a new `Tweet` object with the tweet ID and the current timestamp.
   - Add the tweet to the user's timeline and update the timestamp.
6. To follow a user:
   - Get the `User` objects corresponding to the follower and followee IDs.
   - Add the followee ID to the follower's list of followees.
7. To unfollow a user:
   - Get the `User` objects corresponding to the follower and followee IDs.
   - Remove the followee ID from the follower's list of followees.
8. To retrieve the news feed:
   - Get the `User` object corresponding to the user ID.
   - Create a priority_queue to store the tweets from the user's timeline and the timelines of the
users they follow.
   - Iterate over the tweets in the user's timeline and add them to the priority queue.
   - Iterate over the followees of the user and add their tweets to the priority queue.
   - Pop the top 10 tweets from the priority queue and return them in reverse order.

Time Complexity:
The time complexity of posting a tweet, following/unfollowing a user, and retrieving the news feed
is O(log n), where n is the number of tweets. This is because we use a priority queue to retrieve
the top 10 tweets in the news feed.

Space Complexity:
The space complexity is O(m + n), where m is the number of users and n is the number of tweets. We
store the user information in the unordered_map and the tweets in the deque.
*/

class Twitter {
  private:
    struct Tweet {
        int tweetId;
        int timestamp;
        Tweet(int id, int time) : tweetId(id), timestamp(time) {}
    };
```

```
    unordered_map<int, vector<Tweet>> userTweets;  // Store tweets of each user
    unordered_map<int, unordered_set<int>> userFollowees;  // Store followees of each user
    int time;  // Keep track of the timestamp

  public:
    Twitter() {
        time = 0;
    }

    void postTweet(int userId, int tweetId) {
           userTweets[userId].emplace_back(tweetId, time++);   // Add the tweet with the current
timestamp
    }

    void follow(int followerId, int followeeId) {
         userFollowees[followerId].insert(followeeId);  // Add followee to the follower's set of
followees
    }

    void unfollow(int followerId, int followeeId) {
         userFollowees[followerId].erase(followeeId);  // Remove followee from the follower's set
of followees
    }

    vector<int> getNewsFeed(int userId) {
        vector<int> newsFeed;
         priority_queue<pair<int, int>> pq;  // Use a priority queue to sort tweets by timestamp
(max-heap)

        // Add the user's own tweets to the priority queue
        for (const auto &tweet : userTweets[userId]) {
            pq.push({ tweet.timestamp, tweet.tweetId });
        }

        // Add tweets from the user's followees to the priority queue
        for (int followeeId : userFollowees[userId]) {
            for (const auto &tweet : userTweets[followeeId]) {
                pq.push({ tweet.timestamp, tweet.tweetId });
            }
        }

        // Retrieve the top 10 tweets from the priority queue
        while (!pq.empty() && newsFeed.size() < 10) {
            newsFeed.push_back(pq.top().second);  // Add the tweet ID to the news feed
            pq.pop();
        }

        return newsFeed;
    }
};
```

## 70. 09_Heap_Priority_Queues/07_Find_Median_from_Data_Stream/0295-find-median-

```
/*
Problem: LeetCode 295 - Find Median from Data Stream

Description:
Design a data structure that supports adding integers to the structure and finding the median of the current elements.
The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the average of the two middle values.

Intuition:
To efficiently find the median of a data stream, we can use two heaps: a max-heap to store the smaller half of the elements, and a min-heap to store the larger half of the elements.
The median will either be the root of the max-heap (if the heaps have equal size) or the average of the roots of both heaps (if the max-heap has one more element than the min-heap).

Approach:
1. Use two priority_queues (heaps) - a max-heap (`maxHeap`) and a min-heap (`minHeap`).
   - The max-heap (`maxHeap`) stores the smaller half of the elements.
   - The min-heap (`minHeap`) stores the larger half of the elements.
2. Maintain the following conditions:
   - The size of the max-heap is either equal to or one more than the size of the min-heap.
   - The root of the max-heap is smaller than or equal to the root of the min-heap.
3. When adding a new element:
    - If the max-heap is empty or the element is less than the root of the max-heap, push the element into the max-heap.
   - Otherwise, push the element into the min-heap.
    - Balance the heaps by moving the root of the max-heap to the min-heap if the sizes are not balanced.
4. To find the median:
   - If the size of the max-heap is greater than the min-heap, return the root of the max-heap.
   - Otherwise, return the average of the roots of both heaps.

Time Complexity:
Adding an element and finding the median both have a time complexity of O(log n), where n is the number of elements in the data stream. This is due to the heap operations.

Space Complexity:
The space complexity is O(n), where n is the number of elements in the data stream. This is because we store the elements in the heaps.
*/

class MedianFinder {
  private:
    priority_queue<int> maxHeap;  // Max-heap to store the smaller half of the elements
    priority_queue<int, vector<int>, greater<int>> minHeap;  // Min-heap to store the larger half of the elements

  public:
    void addNum(int num) {
        if (maxHeap.empty() || num <= maxHeap.top()) {
            maxHeap.push(num);
        } else {
            minHeap.push(num);
        }

        // Balance the heaps
        if (maxHeap.size() > minHeap.size() + 1) {
            minHeap.push(maxHeap.top());
            maxHeap.pop();
        } else if (maxHeap.size() < minHeap.size()) {
```

```
            maxHeap.push(minHeap.top());
            minHeap.pop();
        }
    }

    double findMedian() {
        if (maxHeap.size() > minHeap.size()) {
            return maxHeap.top();
        } else {
            return (maxHeap.top() + minHeap.top()) / 2.0;
        }
    }
};
```

## 71. 10_Backtracking/01_Subsets/0078-subsets.cpp

```
/*
Problem: LeetCode 78 - Subsets

Description:
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.

Intuition:
To find all possible subsets of a given array, we can use a backtracking approach. We start with an empty subset and gradually add elements to it, generating all possible combinations.

Approach:
1. Initialize an empty vector `subset` to store the current subset.
2. Initialize an empty vector `result` to store all subsets.
3. Define a helper function `generateSubsets`:
    - If the index is equal to the size of the input array `nums`, add the current subset to the `result` vector.
   - Otherwise:
     - Include the current element at the current index in the subset.
     - Recursively call `generateSubsets` with the next index.
     - Exclude the current element from the subset.
     - Recursively call `generateSubsets` with the next index.
4. Call the `generateSubsets` function with the initial index 0.
5. Return the `result` vector containing all possible subsets.

Time Complexity:
The time complexity is O(2^n), where n is the size of the input array `nums`. This is because there are 2^n possible subsets, and we generate all of them.

Space Complexity:
The space complexity is O(n), where n is the size of the input array `nums`. This is because we store the subsets in the `result` vector.
*/

class Solution {
  public:
    vector<vector<int>> subsets(vector<int> &nums) {
        vector<vector<int>> result;
        vector<int> subset;
        generateSubsets(nums, 0, subset, result);
        return result;
    }

  private:
          void generateSubsets(const vector<int> &nums, int index, vector<int> &subset, vector<vector<int>> &result) {
        // Base case: If we have processed all elements, add the current subset to the result
        if (index == nums.size()) {
            result.push_back(subset);
            return;
        }

        subset.push_back(nums[index]);  // Include the current element
        generateSubsets(nums, index + 1, subset, result);  // Recursively call with the next index
        subset.pop_back();  // Exclude the current element
        generateSubsets(nums, index + 1, subset, result);  // Recursively call with the next index
    }
};
```

## 72. 10_Backtracking/02_Combination_Sum/0039-combination-sum.cpp

```
/*
Problem: LeetCode 39 - Combination Sum

Description:
Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target.
You may return the combinations in any order.
The same number may be chosen from candidates an unlimited number of times.
Note: The solution set must not contain duplicate combinations.

Intuition:
To find all unique combinations that sum to the target, we can use a backtracking approach. The idea is to generate all possible combinations by trying out different candidate elements at each step. We explore each candidate, including it in the current combination if its value does not exceed the remaining target. By backtracking, we generate all valid combinations that sum up to the target.

Approach:
1. Sort the input array `candidates` to handle duplicates and generate combinations in a non-decreasing order.
2. Initialize an empty vector `combination` to store the current combination.
3. Initialize an empty vector `result` to store all valid combinations.
4. Define a helper function `backtrack`:
   - If the remaining target is equal to 0, add the current combination to the `result` vector.
   - Otherwise:
     - Iterate through the candidates from the current index to the end:
       - If the current candidate is greater than the remaining target, break the loop (as the remaining candidates will be larger and will not fit).
       - Include the current candidate in the combination.
         - Recursively call `backtrack` with the updated remaining target (target - current candidate) and the same index (to allow choosing the same candidate multiple times).
       - Exclude the current candidate from the combination (backtrack).
5. Call the `backtrack` function with the initial target and index 0.
6. Return the `result` vector containing all valid combinations.

Time Complexity:
The time complexity is determined by the number of valid combinations. In the worst case, the algorithm explores all possible combinations. The time complexity can be approximated as O(N^target), where N is the number of candidates and target is the given target integer.

Space Complexity:
The space complexity is determined by the recursion stack and the `result` vector that stores all valid combinations. The space complexity is O(target) to store the recursion stack, and the `result` vector may contain multiple combinations.
*/

class Solution {
  public:
    vector<vector<int>> combinationSum(vector<int> &candidates, int target) {
        vector<vector<int>> result;
        vector<int> combination;
        sort(candidates.begin(), candidates.end());  // Sort the candidates in ascending order
        backtrack(candidates, target, 0, combination, result);  // Call the backtrack function
        return result;
    }

  private:
    // Backtracking function to find combinations that sum up to the target
    void backtrack(const vector<int> &candidates, int target, int index, vector<int> &combination, vector<vector<int>> &result) {
```

```
        // Base case: if the target is reached, add the current combination to the result
        if (target == 0) {
            result.push_back(combination);
            return;
        }

        // Iterate through the candidates starting from the given index
        for (int i = index; i < candidates.size(); ++i) {
            // If the current candidate is greater than the target, skip the remaining candidates
            if (candidates[i] > target) {
                break;
            }

            // Include the current candidate in the combination
            combination.push_back(candidates[i]);
            // Recursively call the backtrack function with the updated target and the same index
            backtrack(candidates, target - candidates[i], i, combination, result);
            // Exclude the current candidate from the combination (backtrack)
            combination.pop_back();
        }
    }
};
```

## 73. 10_Backtracking/03_Permutations/0046-permutations.cpp

```
/*
Problem: LeetCode 46 - Permutations

Description:
Given an array nums of distinct integers, return all possible permutations of the elements in nums.
You can return the answer in any order.

Intuition:
To find all possible permutations of a given array, we can use a backtracking approach. The idea is to generate all possible arrangements by trying out different elements at each step.
We explore each element, including it in the current permutation if it hasn't been used before. By backtracking, we generate all valid permutations.

Approach:
1. Initialize an empty vector `permutation` to store the current permutation.
2. Initialize an empty vector `result` to store all valid permutations.
3. Define a helper function `backtrack`:
    - If the size of the current permutation is equal to the size of the input array `nums`, add the current permutation to the `result` vector.
   - Otherwise:
     - Iterate through the elements in the input array `nums`:
       - If the current element is already in the permutation, skip it to avoid duplicates.
       - Include the current element in the permutation.
       - Recursively call `backtrack` with the updated permutation.
       - Exclude the current element from the permutation (backtrack).
4. Call the `backtrack` function with an empty permutation.
5. Return the `result` vector containing all valid permutations.

Time Complexity:
The time complexity is O(N!), where N is the size of the input array `nums`. This is because there are N! possible permutations, and we generate all of them.

Space Complexity:
The space complexity is O(N), where N is the size of the input array `nums`. This is because we store the permutations in the `result` vector.
*/

class Solution {
  public:
    vector<vector<int>> permute(vector<int> &nums) {
        vector<vector<int>> result;
        vector<int> permutation;
        vector<bool> used(nums.size(), false);  // Track used elements to avoid duplicates
        backtrack(nums, used, permutation, result);
        return result;
    }

  private:
        void backtrack(const vector<int> &nums, vector<bool> &used, vector<int> &permutation, vector<vector<int>> &result) {
        if (permutation.size() == nums.size()) {
            result.push_back(permutation);
            return;
        }

        for (int i = 0; i < nums.size(); ++i) {
            if (used[i]) {
                continue;  // Skip already used elements
            }
```

```
            used[i] = true;  // Mark the current element as used
            permutation.push_back(nums[i]);  // Include the current element in the permutation
                backtrack(nums, used, permutation, result);   // Recursively call with updated
permutation
            permutation.pop_back();  // Exclude the current element (backtrack)
            used[i] = false;  // Mark the current element as unused for other permutations
        }
    }
};

// class Solution {
// public:
//     vector<vector<int>> permute(vector<int>& nums) {
//         vector<vector<int>> result;
//         backtrack(result, nums, 0);
//         return result;
//     }
// private:
//     void backtrack(vector<vector<int>> &result, vector<int> &nums, int start) {
//         if(start == nums.size()) {
//             result.push_back(nums);
//             return;
//         }

//         for(int i = start; i < nums.size(); i++) {
//             // swapping allows us to explore all possible permutations by considering different
elements at the current position.
//             // since start is zero we will swap indices, [0, 1], 0, 2], ... , [0, n-1]
//             swap(nums[i], nums[start]);
//             backtrack(result, nums, start+1);
//             // Once we come out of the recursion, we backtrack by swapping back the elements at
indices start and i. This is necessary to restore the original order of elements and avoid
duplicates.
//             swap(nums[i], nums[start]);
//         }
//     }
// };
```

## 74. 10_Backtracking/04_Subsets_II/0090-subsets-ii.cpp

```
/*
Problem: LeetCode 90 - Subsets II

Description:
Given an integer array nums that may contain duplicates, return all possible subsets (the power set).
The solution set must not contain duplicate subsets. Return the solution in any order.

Intuition:
To find all possible subsets of a given array that may contain duplicates, we can use a backtracking approach. The idea is to generate all possible combinations by trying out different elements at each step, while avoiding duplicate subsets. We sort the array to handle duplicates and skip adding duplicate elements to the current subset.

Approach:
1. Sort the input array `nums` to handle duplicates and generate combinations in a non-decreasing order.
2. Initialize an empty vector `subset` to store the current subset.
3. Initialize an empty vector `result` to store all subsets.
4. Define a helper function `backtrack`:
   - Add the current subset to the `result` vector.
   - Iterate through the elements in the input array `nums`:
       - If the current element is a duplicate (not the first occurrence), skip it to avoid duplicate subsets.
     - Include the current element in the subset.
     - Recursively call `backtrack` with the next index.
     - Exclude the current element from the subset.
5. Call the `backtrack` function with the initial index 0.
6. Return the `result` vector containing all subsets.

Time Complexity:
The time complexity is O(2^n), where n is the size of the input array `nums`. This is because there are 2^n possible subsets, and we generate all of them.

Space Complexity:
The space complexity is O(n), where n is the size of the input array `nums`. This is because we store the subsets in the `result` vector.
*/

class Solution {
  public:
    vector<vector<int>> subsetsWithDup(vector<int> &nums) {
        vector<vector<int>> result;
        vector<int> subset;
        sort(nums.begin(), nums.end());  // Sort the array to handle duplicates
        backtrack(nums, 0, subset, result);
        return result;
    }

  private:
     void backtrack(const vector<int> &nums, int index, vector<int> &subset, vector<vector<int>> &result) {
        result.push_back(subset);  // Add the current subset to the result vector

        for (int i = index; i < nums.size(); ++i) {
            if (i > index && nums[i] == nums[i - 1]) {
                continue;  // Skip duplicate elements to avoid duplicate subsets
            }

            subset.push_back(nums[i]);  // Include the current element in the subset
```

```
            backtrack(nums, i + 1, subset, result);  // Recursively call with the next index
            subset.pop_back();  // Exclude the current element (backtrack)
        }
    }
};
```

## 75. 10_Backtracking/05_Combination_Sum_II/0040-combination-sum-ii.cpp

```
/*
Problem: LeetCode 40 - Combination Sum II

Description:
Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.
Each number in candidates may only be used once in the combination.
Note: The solution set must not contain duplicate combinations.

Intuition:
To find all unique combinations that sum to the target, we can use a backtracking approach. The idea is to generate all possible combinations by trying out different candidate elements at each step.
However, we need to skip duplicate combinations to avoid duplicates in the solution set.

Approach:
1. Sort the input array `candidates` to handle duplicates and generate combinations in a non-decreasing order.
2. Initialize an empty vector `combination` to store the current combination.
3. Initialize an empty vector `result` to store all valid combinations.
4. Define a helper function `backtrack`:
    - If the current sum is equal to the target, add the current combination to the `result` vector.
   - Otherwise:
     - Iterate through the candidates from the current index to the end:
        - If the current candidate is greater than the remaining target, break the loop (as the remaining candidates will be larger and won't fit).
        - If the current candidate is a duplicate (not the first occurrence), skip it to avoid duplicate combinations.
       - Include the current candidate in the combination.
       - Recursively call `backtrack` with the updated sum and the next index.
       - Exclude the current candidate from the combination.
5. Call the `backtrack` function with the initial sum of 0 and index 0.
6. Return the `result` vector containing all valid combinations.

Time Complexity:
The time complexity is O(2^n), where n is the size of the input array `candidates`. This is because there are 2^n possible combinations, and we generate all of them.

Space Complexity:
The space complexity is O(n), where n is the size of the input array `candidates`. This is because we store the combinations in the `result` vector.
*/

class Solution {
  public:
    vector<vector<int>> combinationSum2(vector<int> &candidates, int target) {
        vector<vector<int>> result;
        vector<int> combination;
        sort(candidates.begin(), candidates.end());  // Sort the candidates
        backtrack(candidates, target, 0, combination, result);
        return result;
    }

  private:
    void backtrack(const vector<int> &candidates, int target, int index, vector<int> &combination, vector<vector<int>> &result) {
        if (target == 0) {
            result.push_back(combination);
            return;
```

```
        }

        for (int i = index; i < candidates.size(); ++i) {
            if (candidates[i] > target) {
                break;  // Skip remaining candidates since they will be larger and won't fit
            }

            if (i > index && candidates[i] == candidates[i - 1]) {
                continue;  // Skip duplicate candidates to avoid duplicate combinations
            }

            combination.push_back(candidates[i]);  // Include the current candidate
                 backtrack(candidates, target - candidates[i], i + 1, combination, result);   //
Recursively call with updated target and next index
            combination.pop_back();  // Exclude the current candidate (backtrack)
        }
    }
};
```

## 76. 10_Backtracking/06_Word_Search/0079-word-search.cpp

```
/*
Problem: LeetCode 79 - Word Search

Description:
Given an m x n grid of characters board and a string word, return true if word exists in the grid.
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring.
The same letter cell may not be used more than once.

Intuition:
To determine if a given word exists in the grid, we can use a backtracking approach. We start from each cell and explore all possible paths to find the word.
At each step, we check if the current cell matches the current character of the word. If it does, we continue the search in the neighboring cells until the word is found or all paths have been explored.

Approach:
1. Iterate through each cell in the grid:
   - If the current cell matches the first character of the word, call the `backtrack` function.
2. Define a helper function `backtrack`:
   - If the index of the current character equals the length of the word, return true (the entire word has been found).
   - If the current cell is out of bounds or does not match the current character, return false.
   - Mark the current cell as visited (e.g., change its value to a special character to indicate it has been used).
   - Recursively call `backtrack` for the neighboring cells (up, down, left, right) with the next character of the word.
   - Restore the original value of the current cell (backtrack).
3. Return the result obtained from the `backtrack` function.

Time Complexity:
The time complexity is O(M * N * 4^L), where M is the number of rows, N is the number of columns, and L is the length of the word. In the worst case, we explore all possible paths from each cell.

Space Complexity:
The space complexity is O(L), where L is the length of the word. This is the maximum depth of the recursion stack.
*/

class Solution {
  public:
    bool exist(vector<vector<char>> &board, string word) {
        int m = board.size();
        int n = board[0].size();

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (board[i][j] == word[0] && backtrack(board, word, i, j, 0)) {
                    return true;
                }
            }
        }

        return false;
    }

  private:
    bool backtrack(vector<vector<char>> &board, const string &word, int row, int col, int index) {
        if (index == word.length()) {
            return true;  // The entire word has been found
```

```
        }

        int m = board.size();
        int n = board[0].size();

        if (row < 0 || row >= m || col < 0 || col >= n || board[row][col] != word[index]) {
                return false;  // Out of bounds or the current cell does not match the current
character
        }

        char temp = board[row][col];
        board[row][col] = '#';  // Mark the current cell as visited
        // Recursively call for the neighboring cells
        bool found = backtrack(board, word, row - 1, col, index + 1) ||
                     backtrack(board, word, row + 1, col, index + 1) ||
                     backtrack(board, word, row, col - 1, index + 1) ||
                     backtrack(board, word, row, col + 1, index + 1);
        board[row][col] = temp;  // Mark the current cell as unvisited (backtrack)
        return found;
    }
};
```

# 77. 10_Backtracking/07_Palindrome_Partitioning/0131-palindrome-partitioning.cpp

```
/*
Problem: LeetCode 131 - Palindrome Partitioning

Description:
Given a string s, partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s.
A palindrome string is a string that reads the same backward as forward.

Intuition:
To find all possible palindrome partitioning of a given string, we can use a backtracking approach. The idea is to generate all possible partitions by trying out different substrings at each step.
We check if the current substring is a palindrome and continue the search for the remaining part of the string. By backtracking, we generate all valid palindrome partitionings.

Approach:
1. Initialize an empty vector `partition` to store the current partition.
2. Initialize an empty vector `result` to store all valid partitions.
3. Define a helper function `backtrack`:
    - If the start index is equal to the length of the string, add the current partition to the `result` vector.
   - Otherwise:
     - Iterate through the substring starting from the current index:
       - If the substring is a palindrome:
         - Include the substring in the partition.
             - Recursively call `backtrack` with the updated start index (next index after the substring).
         - Exclude the substring from the partition (backtrack).
4. Call the `backtrack` function with the initial start index of 0.
5. Return the `result` vector containing all valid partitions.

Time Complexity:
The time complexity is O(N * 2^N), where N is the length of the string `s`. This is because there can be up to 2^N possible palindrome partitions, and for each partition, we check if each substring is a palindrome.

Space Complexity:
The space complexity is O(N) for the recursion stack, where N is the length of the string `s`. The `result` vector may contain multiple partitions, but the overall space complexity is dominated by the recursion stack.
*/

class Solution {
  public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> partition;
         backtrack(s, 0, partition, result);  // Call the backtrack function to generate all valid partitions
        return result;
    }

  private:
    // Depth First Search
     void backtrack(const string &s, int start, vector<string> &partition, vector<vector<string>> &result) {
        if (start == s.length()) {
            result.push_back(partition);  // Add the current partition to the result
            return;
        }
```

```
        for (int i = start; i < s.length(); ++i) {
            if (isPalindrome(s, start, i)) {  // Check if the substring is a palindrome
                  partition.push_back(s.substr(start, i - start + 1));  // Include the palindrome
substring in the partition
                backtrack(s, i + 1, partition, result);  // Recursively call with the next index
                partition.pop_back();  // Exclude the last added substring (backtrack)
            }
        }
    }

    bool isPalindrome(const string &s, int start, int end) {
        while (start < end) {
            if (s[start] != s[end]) {  // If characters don't match, it's not a palindrome
                return false;
            }

            ++start;
            --end;
        }

        return true;  // All characters matched, it's a palindrome
    }
};
```

## 78. 10_Backtracking/08_Letter_Combinations_of_a_Phone_Number/0017-letter-comb

```
/*
Problem: LeetCode 17 - Letter Combinations of a Phone Number

Description:
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.

Intuition:
To find all possible letter combinations of a given phone number, we can use a backtracking approach. The idea is to generate all possible combinations by trying out different letters at each step, based on the digit mapping on a phone keypad.

Approach:
1. Define a mapping from each digit to the corresponding letters on a phone keypad.
2. Initialize an empty vector `combinations` to store the current combination.
3. Initialize an empty vector `result` to store all valid combinations.
4. Define a helper function `backtrack`:
   - If the index is equal to the length of the input digits, add the current combination to the `result` vector.
   - Otherwise:
     - Get the letters corresponding to the current digit.
     - Iterate through the letters:
       - Include the current letter in the combination.
       - Recursively call `backtrack` with the updated index.
       - Exclude the current letter from the combination (backtrack).
5. Call the `backtrack` function with the initial index of 0.
6. Return the `result` vector containing all valid combinations.

Time Complexity:
The time complexity is O(3^N * 4^M), where N is the number of digits that map to 3 letters (e.g., 2, 3, 4, 5, 6, 8) and M is the number of digits that map to 4 letters (e.g., 7, 9). For each digit, there can be up to 3 mappings (3^N) or 4 mappings (4^M), and we generate all possible combinations.

Space Complexity:
The space complexity is O(N + M), where N is the number of digits that map to 3 letters and M is the number of digits that map to 4 letters. This is the space used to store the combinations.
*/

class Solution {
  public:
    vector<string> letterCombinations(string digits) {
        vector<string> result;

        if (digits.empty()) {
            return result;  // If the input is empty, return an empty result
        }

        vector<string> mapping = {
            "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"
        };
        string combination;  // Store the current combination
          backtrack(digits, mapping, 0, combination, result);  // Call the backtrack function to generate all valid combinations
        return result;
    }

  private:
    // Depth First Search (DFS) Backtracking function
```

```
       void backtrack(const string &digits, const vector<string> &mapping, int index, string &combination, vector<string> &result) {
        if (index == digits.length()) {
            result.push_back(combination);  // Add the current combination to the result
            return;
        }

         string letters = mapping[digits[index] - '0'];  // Get the corresponding letters for the current digit

        for (char letter : letters) {
            combination.push_back(letter);  // Add the letter to the current combination
             backtrack(digits, mapping, index + 1, combination, result);  // Recursively call with the next index
            combination.pop_back();  // Remove the last added letter (backtrack)
        }
    }
};

/*
// Breadth First Search((BFS)
class Solution {
public:
    vector<string> letterCombinations(string digits) {
        vector<string> result;
        if (digits.empty()) {
            return result;
        }

        vector<string> mapping = {
            "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"
        };

        queue<string> combinations;
        combinations.push("");

        while (!combinations.empty()) {
            string curr = combinations.front();
            combinations.pop();

            if (curr.length() == digits.length()) {
                result.push_back(curr);
                continue;
            }

            string letters = mapping[digits[curr.length()] - '0'];
            for (char letter : letters) {
                combinations.push(curr + letter);
            }
        }

        return result;
    }
};
*/

/*
class Solution {
public:
    vector<string> letterCombinations(string digits) {
        if(digits.size() == 0) return vector<string>{};
        vector<string> ans;
        string curr;
        backtrack(curr, 0, digits, ans);
```

```
        return ans;
    }

private:
    unordered_map<char, vector<char>> dig2char = {
        {'2', {'a', 'b', 'c'}},
        {'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 backtrack(string & curr, int index, string & digits, vector<string> & ans) {
        if(curr.size() == digits.size()) {
            ans.push_back(curr);
            return;
        }
        vector<char> possibleLetters = dig2char[digits[index]];
        for(char c: possibleLetters) {
            curr.push_back(c);
            backtrack(curr, index + 1, digits, ans);
            curr.pop_back();
        }
    }
};
*/
```

## 79. 10_Backtracking/09_N_Queens/0051-n-queens.cpp

```
/*
Problem: LeetCode 51 - N-Queens

Description:
The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.
Given an integer n, return all distinct solutions to the n-queens puzzle.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Intuition:
The N-Queens problem can be solved using backtracking. The idea is to place queens on the board row by row, ensuring that no two queens attack each other. We can use a recursive approach to explore all possible placements and backtrack when conflicts arise.

Approach:
1. Define a vector of vectors `board` to represent the chessboard.
2. Define a vector `queens` to store the column index of the queens placed in each row.
3. Define a helper function `backtrack`:
    - If the current row is equal to `n`, it means all queens have been placed successfully. Add the current `board` configuration to the `result` vector.
   - Otherwise:
     - Iterate through the columns from 0 to `n`:
       - Check if placing a queen at the current position (row, col) is valid (i.e., no conflicts with previously placed queens).
        - If it is valid, mark the current position on the `board` as a queen ('Q') and add the current column to `queens`.
       - Recursively call `backtrack` for the next row.
       - Remove the queen from the `board` and backtrack by removing the last queen from `queens`.
4. Call the `backtrack` function with the initial row 0.
5. Return the `result` vector containing all distinct board configurations.

Time Complexity:
The time complexity is O(N!), where N is the size of the chessboard (n x n). This is because there are N! possible placements for the queens.

Space Complexity:
The space complexity is O(N), where N is the size of the chessboard (n x n). This is because we store the `board`, `queens`, and the `result` vector.
*/

class Solution {
  public:
    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> result;
        vector<string> board(n, string(n, '.'));  // Initialize the board with empty spaces
        vector<int> queens;  // Column indices of the queens in each row
         backtrack(n, 0, board, queens, result);  // Call the backtrack function to generate all valid solutions
        return result;
    }

  private:
    // Backtracking function to generate all valid solutions
          void backtrack(int n, int row, vector<string> &board, vector<int> &queens, vector<vector<string>> &result) {
        if (row == n) {
            result.push_back(board);  // Add the current valid solution to the result
            return;
        }
```

```
        for (int col = 0; col < n; ++col) {
            if (isValidPlacement(row, col, queens)) {
                board[row][col] = 'Q';  // Place the queen at the current position
                queens.push_back(col);  // Store the column index of the queen in the current row
                // Recursively call for the next row
                backtrack(n, row + 1, board, queens, result);
                queens.pop_back();  // Remove the last queen from the current row
                board[row][col] = '.';  // Restore the empty space
            }
        }
    }

    // Function to check if placing a queen at the current position is valid
    bool isValidPlacement(int row, int col, const vector<int> &queens) {
        for (int i = 0; i < queens.size(); ++i) {
            int rowDiff = abs(row - i);
            int colDiff = abs(col - queens[i]);

            if (rowDiff == 0 || colDiff == 0 || rowDiff == colDiff) {
                return false;  // Found a queen in the same row, same column, or diagonal
            }
        }

        return true;  // No conflicting queens found, placement is valid
    }
};
```

## 80. 11_Graphs/01_Number_of_Islands/0200-number-of-islands.cpp

```
/*
Problem: LeetCode 200 - Number of Islands

Description:
Given an m x n grid containing '1's (land) and '0's (water), return the number of islands.
An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically.
You may assume all four edges of the grid are all surrounded by water.

Intuition:
To find the number of islands in the grid, we can use a depth-first search (DFS) or breadth-first search (BFS) approach. The idea is to traverse the grid and whenever we encounter a land ('1'), we explore its neighboring cells and mark them as visited to avoid counting them again.

Approach:
1. Iterate through each cell in the grid:
    - If the current cell is a land ('1'), increment the count of islands and call the `dfs` or `bfs` function to explore its neighboring cells.
2. Define a helper function `dfs` or `bfs`:
   - Check if the current cell is out of bounds or is not a land ('1'). If so, return.
   - Mark the current cell as visited by changing its value to '0'.
   - Recursively call `dfs` or `bfs` for the neighboring cells (up, down, left, right).
3. Return the count of islands.

Time Complexity:
The time complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. We visit each cell once.

Space Complexity:
The space complexity is O(min(m, n)), where m is the number of rows and n is the number of columns in the grid. This is the maximum depth of the recursion stack or the maximum number of cells in the queue during the BFS traversal.
*/

class Solution {
  public:
    int numIslands(vector<vector<char>> &grid) {
        if (grid.empty()) {
            return 0;
        }

        int m = grid.size();  // Number of rows in the grid
        int n = grid[0].size();  // Number of columns in the grid
        int count = 0;  // Counter for the number of islands

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == '1') {  // Found a new island
                    ++count;
                    dfs(grid, i, j);  // Explore the island using DFS
                    // bfs(grid, i, j);  // Alternatively, we can use BFS to explore the island
                }
            }
        }

        return count;
    }

  private:
    void dfs(vector<vector<char>> &grid, int row, int col) {
```

```
        if (row < 0 || row >= grid.size() || col < 0 || col >= grid[0].size() || grid[row][col] !=
'1') {
            return;  // Out of bounds or already visited cell
        }

        grid[row][col] = '0';  // Mark the current cell as visited
        // Recursively call DFS for the neighboring cells (up, down, left, right)
        dfs(grid, row - 1, col);  // Up
        dfs(grid, row + 1, col);  // Down
        dfs(grid, row, col - 1);  // Left
        dfs(grid, row, col + 1);  // Right
    }

    void bfs(vector<vector<char>> &grid, int row, int col) {
        int m = grid.size();  // Number of rows in the grid
        int n = grid[0].size();  // Number of columns in the grid
        queue<pair<int, int>> q;  // Queue to store cell positions
        q.push({row, col});  // Start with the current cell
        grid[row][col] = '0';  // Mark the current cell as visited
          vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};   // Possible
directions (up, down, left, right)

        while (!q.empty()) {
            int r = q.front().first;  // Row of the current cell
            int c = q.front().second;  // Column of the current cell
            q.pop();

            // Explore the neighboring cells
            for (const auto &dir : directions) {
                int nr = r + dir.first;  // Row of the neighboring cell
                int nc = c + dir.second;  // Column of the neighboring cell

                // Check if the neighboring cell is within the grid and is unvisited
                if (nr >= 0 && nr < m && nc >= 0 && nc < n && grid[nr][nc] == '1') {
                    q.push({nr, nc});  // Add the neighboring cell to the queue
                    grid[nr][nc] = '0';  // Mark the neighboring cell as visited
                }
            }
        }
    }
};
```

## 81. 11_Graphs/02_Clone_Graph/0133-clone-graph.cpp

```
/*
Problem: LeetCode 133 - Clone Graph

Description:
Given a reference of a node in a connected undirected graph.
Return a deep copy (clone) of the graph.
Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

Intuition:
To clone a graph, we can use a depth-first search (DFS) or breadth-first search (BFS) approach. The idea is to traverse the original graph and create a copy of each node and its neighbors. We can store the visited nodes in a map to ensure that we don't create duplicate copies.

Approach:
1. Define a helper function `cloneNode`:
   - Create a copy of the current node.
   - Iterate through the neighbors of the current node:
     - If a neighbor is not visited, recursively call `cloneNode` to create a copy of the neighbor and add it to the neighbors of the current node.
     - If a neighbor is already visited, add the corresponding copy from the map to the neighbors of the current node.
2. Create an empty map to store the copies of the nodes.
3. Call the `cloneNode` function with the given reference node.
4. Return the copy of the reference node.

Time Complexity:
The time complexity is O(V + E), where V is the number of nodes (vertices) and E is the number of edges in the graph. We visit each node and each edge once.

Space Complexity:
The space complexity is O(V), where V is the number of nodes (vertices) in the graph. This is the space used to store the copies of the nodes and the recursion stack.
*/

class Solution {
  public:
    Node *cloneGraph(Node *node) {
        if (node == nullptr) {
            return nullptr;
        }

        unordered_map<Node *, Node *> nodeMap; // Map to store copies of nodes
        return cloneNode(node, nodeMap);
    }

  private:
    Node *cloneNode(Node *node, unordered_map<Node *, Node *> &nodeMap) {
        // If a copy of the current node already exists, return it
        if (nodeMap.find(node) != nodeMap.end()) {
            return nodeMap[node];
        }

        Node *newNode = new Node(node->val);  // Create a new copy of the current node
        nodeMap[node] = newNode;  // Add the current node to the map

        // Recursively clone the neighbors of the current node
        for (Node *neighbor : node->neighbors) {
            newNode->neighbors.push_back(cloneNode(neighbor, nodeMap));
        }
```

```
        return newNode;
    }
};
```

## 82. 11_Graphs/03_Max_Area_of_Island/0695-max-area-of-island.cpp

```
/*
Problem: LeetCode 695 - Max Area of Island

Description:
Given a grid of 0's and 1's, find the maximum area of an island in the grid.
An island is a group of connected 1's (horizontally or vertically). You may assume all four edges of the grid are all surrounded by water.

Intuition:
To find the maximum area of an island, we can use a depth-first search (DFS) approach. The idea is to traverse the grid and whenever we encounter a land (1), we explore its neighboring cells and count the number of connected lands.

Approach:
1. Initialize a variable `maxArea` to store the maximum area of an island.
2. Iterate through each cell in the grid:
   - If the current cell is a land (1), call the `dfs` function to explore its neighboring cells and update the maximum area.
3. Define a helper function `dfs`:
   - Check if the current cell is out of bounds or is not a land (1). If so, return 0.
   - Mark the current cell as visited by changing its value to 0.
   - Recursively call `dfs` for the neighboring cells (up, down, left, right) and sum the results.
4. Return the maximum area stored in `maxArea`.

Time Complexity:
The time complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. We visit each cell once.

Space Complexity:
The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. This is the space used for the recursion stack during the DFS traversal.
*/

class Solution {
  public:
    int maxAreaOfIsland(vector<vector<int>> &grid) {
        if (grid.empty()) {
            return 0;
        }

        int m = grid.size();
        int n = grid[0].size();
        int maxArea = 0;

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == 1) {
                    maxArea = max(maxArea, dfs(grid, i, j));
                }
            }
        }

        return maxArea;
    }

  private:
    int dfs(vector<vector<int>> &grid, int row, int col) {
        if (row < 0 || row >= grid.size() || col < 0 || col >= grid[0].size() || grid[row][col] == 0) {
            return 0;
```

```
        }

        grid[row][col] = 0;  // Mark the current cell as visited
        int area = 1;  // Count the current cell as part of the island
        // Recursively explore the neighboring cells (up, down, left, right)
        area += dfs(grid, row - 1, col);  // Up
        area += dfs(grid, row + 1, col);  // Down
        area += dfs(grid, row, col - 1);  // Left
        area += dfs(grid, row, col + 1);  // Right
        return area;
    }
};
```

# 83. 11_Graphs/04_Pacific_Atlantic_Water_Flow/0417-pacific-atlantic-water-flow.cpp

```
/*
Problem: LeetCode 417 - Pacific Atlantic Water Flow

Description:
Given an m x n matrix of non-negative integers representing the height of each unit cell in a continent, the "Pacific ocean" touches the left and top edges of the matrix, and the "Atlantic ocean" touches the right and bottom edges.
Water can only flow in four directions (up, down, left, or right) from a cell to its neighboring cells with equal or lower height.
Find the list of grid coordinates where water can flow to both the Pacific and Atlantic oceans.

Intuition:
To find the cells where water can flow to both the Pacific and Atlantic oceans, we can use a depth-first search (DFS) or breadth-first search (BFS) approach. The idea is to start the traversal from the ocean borders (Pacific and Atlantic) and mark the cells that can be reached by water. Finally, we find the cells that are marked by both traversals.

Approach:
1. Create two boolean matrices `canReachPacific` and `canReachAtlantic`, initialized with false, to track the cells that can be reached by water from the respective oceans.
2. Perform a DFS or BFS traversal from the ocean borders to mark the cells that can be reached by water:
   - For the Pacific ocean:
     - Start from the leftmost column and the topmost row. Traverse all neighboring cells with equal or lower heights and mark them as reachable by water from the Pacific ocean.
   - For the Atlantic ocean:
     - Start from the rightmost column and the bottommost row. Traverse all neighboring cells with equal or lower heights and mark them as reachable by water from the Atlantic ocean.
3. Iterate through all the cells and find the cells that are marked as reachable by both oceans.
4. Return the list of grid coordinates representing the cells that can flow to both oceans.

Time Complexity:
The time complexity is O(m * n), where m is the number of rows and n is the number of columns in the matrix. We perform a DFS or BFS traversal on each cell once.

Space Complexity:
The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the matrix. This is the space used to store the boolean matrices and the recursion stack or the queue during the DFS or BFS traversal.
*/

class Solution {
  public:
    vector<vector<int>> pacificAtlantic(vector<vector<int>> &matrix) {
        vector<vector<int>> result;

        if (matrix.empty()) {
            return result;
        }

        int m = matrix.size();
        int n = matrix[0].size();
        vector<vector<bool>> canReachPacific(m, vector<bool>(n, false));  // Matrix to track cells reachable from the Pacific ocean
           vector<vector<bool>> canReachAtlantic(m, vector<bool>(n, false));  // Matrix to track cells reachable from the Atlantic ocean

           // Traverse the top and bottom borders to mark cells reachable from the Pacific and Atlantic oceans
        for (int col = 0; col < n; ++col) {
```

```
            dfs(matrix, 0, col, INT_MIN, canReachPacific);
            dfs(matrix, m - 1, col, INT_MIN, canReachAtlantic);
        }

           // Traverse the left and right borders to mark cells reachable from the Pacific and
Atlantic oceans
        for (int row = 0; row < m; ++row) {
            dfs(matrix, row, 0, INT_MIN, canReachPacific);
            dfs(matrix, row, n - 1, INT_MIN, canReachAtlantic);
        }

        // Find the cells that are reachable from both the Pacific and Atlantic oceans
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (canReachPacific[i][j] && canReachAtlantic[i][j]) {
                    result.push_back({i, j});
                }
            }
        }

        return result;
    }

  private:
         void  dfs(const  vector<vector<int>>  &matrix,  int  row,  int  col,  int  prevHeight,
vector<vector<bool>> &canReachOcean) {
        int m = matrix.size();
        int n = matrix[0].size();

        // Check if the current cell is out of bounds or has been visited already
           if (row < 0 || row >= m || col < 0 || col >= n || matrix[row][col] < prevHeight ||
canReachOcean[row][col]) {
            return;
        }

        canReachOcean[row][col] = true;  // Mark the current cell as reachable
        // Recursively traverse the neighboring cells
        dfs(matrix, row - 1, col, matrix[row][col], canReachOcean);  // Up
        dfs(matrix, row + 1, col, matrix[row][col], canReachOcean);  // Down
        dfs(matrix, row, col - 1, matrix[row][col], canReachOcean);  // Left
        dfs(matrix, row, col + 1, matrix[row][col], canReachOcean);  // Right
    }
};
```

## 84. 11_Graphs/05_Surrounded_Regions/0130-surrounded-regions.cpp

```
/*
Problem: LeetCode 130 - Surrounded Regions

Description:
Given an m x n matrix board containing 'X' and 'O', capture all regions that are surrounded by 'X'.
A region is captured by flipping all 'O's into 'X's in that surrounded region.

Intuition:
To capture the surrounded regions, we need to identify the regions that are connected to the borders of the matrix. The regions that are not connected to the borders are the ones that need to be captured.
We can use a depth-first search (DFS) approach to identify and mark the regions that are connected to the borders, and then iterate through the matrix to capture the remaining regions.

Approach:
1. Traverse the borders of the matrix and perform a DFS to mark the regions that are connected to the borders:
    - If a cell is 'O', perform a DFS to mark all its neighboring 'O' cells as connected to the border.
2. Iterate through the entire matrix:
   - If a cell is 'O' and not marked as connected to the border, capture it by changing it to 'X'.
   - If a cell is marked as connected to the border, restore it to 'O'.
3. The regions that are not marked as connected to the border are the captured regions.

Time Complexity:
The time complexity is O(m * n), where m is the number of rows and n is the number of columns in the matrix. We visit each cell once.

Space Complexity:
The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the matrix. This is the space used for the recursion stack during the DFS traversal.
*/

class Solution {
  public:
    void solve(vector<vector<char>> &board) {
        if (board.empty()) {
            return;
        }

        int m = board.size();
        int n = board[0].size();

        // Traverse the top and bottom borders
        for (int col = 0; col < n; ++col) {
            dfs(board, 0, col);
            dfs(board, m - 1, col);
        }

        // Traverse the left and right borders
        for (int row = 0; row < m; ++row) {
            dfs(board, row, 0);
            dfs(board, row, n - 1);
        }

        // Capture the remaining regions
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (board[i][j] == 'O') {
```

```
                    board[i][j] = 'X';  // Capture the region
                } else if (board[i][j] == '#') {
                    board[i][j] = 'O';  // Restore the region
                }
            }
        }
    }

  private:
    void dfs(vector<vector<char>> &board, int row, int col) {
        int m = board.size();
        int n = board[0].size();

        if (row < 0 || row >= m || col < 0 || col >= n || board[row][col] != 'O') {
            return;
        }

        board[row][col] = '#';  // Mark the cell as connected to the border
        dfs(board, row - 1, col);  // Up
        dfs(board, row + 1, col);  // Down
        dfs(board, row, col - 1);  // Left
        dfs(board, row, col + 1);  // Right
    }
};
```

## 85. 11_Graphs/06_Rotting_Oranges/0994-rotting-oranges.cpp

```
/*
Problem: LeetCode 994 - Rotting Oranges

Description:
You are given an m x n grid where each cell can have one of three values:
- 0 representing an empty cell.
- 1 representing a fresh orange.
- 2 representing a rotten orange.
Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.
Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1.

Intuition:
To find the minimum number of minutes needed to rot all the oranges, we can use a breadth-first search (BFS) approach. We start with the initial rotten oranges and spread the rot to their adjacent fresh oranges. We continue this process in rounds until no more fresh oranges can be infected. The number of rounds needed corresponds to the minimum minutes required.

Approach:
1. Initialize a queue to store the coordinates of the rotten oranges.
2. Initialize variables to keep track of the number of fresh oranges and the number of minutes passed.
3. Iterate through the grid to find the initial rotten oranges and count the number of fresh oranges.
   - Enqueue the coordinates of the rotten oranges into the queue.
4. Perform a BFS traversal:
   - For each round, process all the rotten oranges in the queue.
   - For each rotten orange, check its adjacent cells (up, down, left, right):
      - If an adjacent cell is a fresh orange, mark it as rotten, decrease the count of fresh oranges, and enqueue its coordinates.
   - Increment the number of minutes.
5. After the BFS traversal, check if there are any remaining fresh oranges. If so, return -1.
6. Return the number of minutes minus one since the last round is not counted as a minute needed to rot the oranges.

Time Complexity:
The time complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. In the worst case, we may need to visit all the cells.

Space Complexity:
The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. This is the space used for the queue to store the coordinates of the rotten oranges.
*/

class Solution {
  public:
    int orangesRotting(vector<vector<int>> &grid) {
        if (grid.empty()) {
            return 0;
        }

        int m = grid.size();
        int n = grid[0].size();
        int freshOranges = 0;
        int minutes = 0;
          queue<pair<int, int>> rottenOranges;  // Queue to store the coordinates of the rotten oranges

        // Find the initial rotten oranges and count the number of fresh oranges
        for (int i = 0; i < m; ++i) {
```

```
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == 2) {
                    rottenOranges.push({i, j});
                } else if (grid[i][j] == 1) {
                    ++freshOranges;
                }
            }
        }

        // Perform a BFS traversal
        while (!rottenOranges.empty() && freshOranges > 0) {
            int size = rottenOranges.size();

            for (int i = 0; i < size; ++i) {
                int row = rottenOranges.front().first;
                int col = rottenOranges.front().second;
                rottenOranges.pop();
                // Check adjacent cells (up, down, left, right)
                vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

                for (const auto &dir : directions) {
                    int newRow = row + dir.first;
                    int newCol = col + dir.second;

                            if (newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
grid[newRow][newCol] == 1) {
                        // Mark the adjacent fresh orange as rotten
                        grid[newRow][newCol] = 2;
                        rottenOranges.push({newRow, newCol});
                        --freshOranges;
                    }
                }
            }

            if (!rottenOranges.empty()) {
                ++minutes;  // Increment the number of minutes
            }
        }

        if (freshOranges > 0) {
            return -1;  // There are remaining fresh oranges
        }

        return minutes;
    }
};
```

## 86. 11_Graphs/07_Walls_and_Gates/0286-walls-and-gates.cpp

```
/*
Problem: LeetCode 286 - Walls and Gates

Description:
You are given an m x n grid rooms initialized with these three possible values:
- -1: A wall or an obstacle.
- 0: A gate.
- INF: Infinity means an empty room. We use the value 231 - 1 = 2147483647 to represent INF, as you may assume that the distance to a gate is less than 2147483647.
Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, leave it as INF.

Intuition:
To fill each empty room with the distance to its nearest gate, we can use a breadth-first search (BFS) approach. We start the BFS from each gate and propagate the distances to the neighboring empty rooms. We repeat this process for each gate to ensure that all empty rooms are filled with their nearest gate distances.

Approach:
1. Initialize a queue to store the coordinates of the gates.
2. Iterate through the grid and enqueue the coordinates of the gates into the queue.
3. Perform a BFS traversal:
   - For each gate, start the BFS and propagate the distances to the neighboring empty rooms.
   - Initialize a distance variable as 0.
   - While the queue is not empty, process the rooms in the queue:
     - For each room, check its neighboring rooms (up, down, left, right):
       - If a neighboring room is an empty room (INF), mark it with the current distance + 1, enqueue its coordinates, and update the grid.
   - Increment the distance by 1.
4. After the BFS traversal, the grid will be filled with the distances to the nearest gates.

Time Complexity:
The time complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. In the worst case, we may need to visit all the rooms.

Space Complexity:
The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. This is the space used for the queue to store the coordinates of the gates.
*/

class Solution {
  public:
    void wallsAndGates(vector<vector<int>> &rooms) {
        if (rooms.empty()) {
            return;
        }

        int m = rooms.size();
        int n = rooms[0].size();
        queue<pair<int, int>> gates;  // Queue to store the coordinates of the gates

        // Enqueue the coordinates of the gates
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (rooms[i][j] == 0) {
                    gates.push({i, j});
                }
            }
        }
```

```
        // Perform a BFS traversal
        while (!gates.empty()) {
            int size = gates.size();

            for (int i = 0; i < size; ++i) {
                int row = gates.front().first;
                int col = gates.front().second;
                gates.pop();
                // Check neighboring rooms (up, down, left, right)
                vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

                for (const auto &dir : directions) {
                    int newRow = row + dir.first;
                    int newCol = col + dir.second;

                            if (newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
rooms[newRow][newCol] == INT32_MAX) {
                        // Mark the neighboring room with the distance to the gate and enqueue its
coordinates
                        rooms[newRow][newCol] = rooms[row][col] + 1;
                        gates.push({newRow, newCol});
                    }
                }
            }
        }
    }
};
```

## 87. 11_Graphs/08_Course_Schedule/0207-course-schedule.cpp

```
/*
Problem: LeetCode 207 - Course Schedule

Description:
There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.
For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return true if you can finish all courses. Otherwise, return false.

Intuition:
This problem can be approached as a graph problem where the courses represent nodes and the prerequisites represent directed edges. To determine if it is possible to finish all courses, we need to check if there is a cycle in the graph. If there is a cycle, it means there is a dependency loop, and we cannot finish all the courses.

Approach:
1. Build an adjacency list representation of the graph using the prerequisites.
2. Initialize a visited array to track the visited nodes during the DFS traversal.
3. Iterate through each node in the graph and perform a DFS traversal to detect cycles:
   - If the current node is being visited, it means there is a cycle, so return false.
   - If the current node is not visited, perform a DFS traversal on its neighbors.
4. If no cycles are detected after the DFS traversal, return true.

Time Complexity:
The time complexity is O(V + E), where V is the number of courses (nodes) and E is the number of prerequisites (edges). We visit each course and prerequisite once.

Space Complexity:
The space complexity is O(V + E), where V is the number of courses (nodes) and E is the number of prerequisites (edges). This is the space used for the adjacency list and the visited array.
*/

class Solution {
  public:
    bool canFinish(int numCourses, vector<vector<int>> &prerequisites) {
        vector<vector<int>> graph(numCourses);  // Adjacency list representation of the graph
        vector<int> visited(numCourses, 0);     // Visited array to track the visited nodes

        // Build the graph
        for (const auto &prerequisite : prerequisites) {
            int course = prerequisite[0];
            int prerequisiteCourse = prerequisite[1];
            graph[course].push_back(prerequisiteCourse);
        }

        // Perform a DFS traversal to detect cycles
        for (int course = 0; course < numCourses; ++course) {
            if (!dfs(course, graph, visited)) {
                return false;
            }
        }

        return true;
    }

  private:
    bool dfs(int course, vector<vector<int>> &graph, vector<int> &visited) {
        // If the current course is being visited, it means there is a cycle
        if (visited[course] == 1) {
```

```
            return false;
        }

        // If the current course is already visited, return true
        if (visited[course] == -1) {
            return true;
        }

        visited[course] = 1;  // Mark the current course as being visited

        // Perform a DFS traversal on the neighbors
        for (const auto &neighbor : graph[course]) {
            if (!dfs(neighbor, graph, visited)) {
                return false;
            }
        }

        visited[course] = -1;  // Mark the current course as visited
        return true;
    }
};

// Topological sort
// Same time and space complexity but this can be more efficient in terms of practical performance
/*
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> graph(numCourses);   // Adjacency list representation of the graph
        vector<int> inDegree(numCourses, 0);      // In-degree of each course

        // Build the graph and calculate the in-degree of each course
        for (const auto& prerequisite : prerequisites) {
            int course = prerequisite[0];
            int prerequisiteCourse = prerequisite[1];
            graph[prerequisiteCourse].push_back(course);
            ++inDegree[course];
        }

        queue<int> q;   // Queue to store courses with in-degree 0

        // Enqueue courses with in-degree 0
        for (int i = 0; i < numCourses; ++i) {
            if (inDegree[i] == 0) {
                q.push(i);
            }
        }

        // Perform topological sorting
        while (!q.empty()) {
            int course = q.front();
            q.pop();
            --numCourses;    // Decrement the number of remaining courses

            // Decrement the in-degree of neighbors and enqueue if their in-degree becomes 0
            for (const auto& neighbor : graph[course]) {
                if (--inDegree[neighbor] == 0) {
                    q.push(neighbor);
                }
            }
        }

        return numCourses == 0;
    }
```

```
};
*/
```

## 88. 11_Graphs/09_Course_Schedule_II/0210-course-schedule-ii.cpp

```
/*
Problem: LeetCode 210 - Course Schedule II

Description:
There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.
For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

Intuition:
This problem can be approached as a graph problem where the courses represent nodes and the prerequisites represent directed edges. To determine the ordering of courses, we can use the topological sorting algorithm. If there is a cycle in the graph, it means there is a dependency loop, and we cannot finish all the courses.

Approach:
1. Build an adjacency list representation of the graph using the prerequisites.
2. Initialize an array to store the in-degree of each course. In-degree represents the number of prerequisites for each course.
3. Create a queue and enqueue all the courses with an in-degree of 0.
4. Perform a topological sorting:
   - While the queue is not empty, dequeue a course:
     - Decrement the in-degree of its neighbors by 1.
     - If any neighbor has an in-degree of 0, enqueue it.
     - Add the dequeued course to the result list.
5. If all the courses have been visited, return the result list. Otherwise, return an empty array.

Time Complexity:
The time complexity is O(V + E), where V is the number of courses (nodes) and E is the number of prerequisites (edges). We visit each course and prerequisite once.

Space Complexity:
The space complexity is O(V + E), where V is the number of courses (nodes) and E is the number of prerequisites (edges). This is the space used for the adjacency list, the in-degree array, the queue, and the result list.
*/

class Solution {
  public:
    vector<int> findOrder(int numCourses, vector<vector<int>> &prerequisites) {
        vector<vector<int>> graph(numCourses);   // Adjacency list representation of the graph
        vector<int> inDegree(numCourses, 0);      // In-degree of each course
        vector<int> result;                       // Result list of the course order

        // Build the graph and calculate the in-degree of each course
        for (const auto &prerequisite : prerequisites) {
            int course = prerequisite[0];
            int prerequisiteCourse = prerequisite[1];
            graph[prerequisiteCourse].push_back(course);
            ++inDegree[course];
        }

        queue<int> q;   // Queue to store courses with in-degree 0

        // Enqueue courses with in-degree 0
        for (int i = 0; i < numCourses; ++i) {
            if (inDegree[i] == 0) {
                q.push(i);
```

```
            }
        }

        // Perform topological sorting
        while (!q.empty()) {
            int course = q.front();
            q.pop();
            result.push_back(course);   // Add the course to the result list

            // Decrement the in-degree of neighbors and enqueue if their in-degree becomes 0
            for (const auto &neighbor : graph[course]) {
                if (--inDegree[neighbor] == 0) {
                    q.push(neighbor);
                }
            }
        }

        // If all the courses have been visited, return the result list
        if (result.size() == numCourses) {
            return result;
        }

        return {};   // Return an empty array if it is impossible to finish all courses
    }
};
```

## 89. 11_Graphs/10_Redundant_Connection/0684-redundant-connection.cpp

```
/*
Problem: LeetCode 684 - Redundant Connection

Description:
In this problem, a tree is an undirected graph that is connected and has no cycles.
You are given a graph that started as a tree with n nodes labeled from 1 to n, with one additional edge added. The added edge has two different vertices chosen from 1 to n, and was not an edge that already existed. The graph is represented as an array edges of length n where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the graph.
Return an edge that can be removed so that the resulting graph is a tree of n nodes. If there are multiple answers, return the answer that occurs last in the input.

Intuition:
The problem is asking to find the redundant connection, which is the edge that creates a cycle in the given graph. We can solve this problem using a Union-Find algorithm.

Approach:
1. Create a parent array of size n+1 to represent each node's parent.
2. Iterate through the edges:
   - Initialize variables to store the parent of each node in the current edge.
   - If the parent of both nodes is the same, it means adding this edge will create a cycle, so return the current edge.
   - Otherwise, union the nodes by setting the parent of one node to be the parent of the other node.
3. If no redundant edge is found, return an empty vector.

Time Complexity:
The time complexity is O(n * ?(n)), where n is the number of nodes and ?(n) is the inverse Ackermann function. In practice, ?(n) is a very slow-growing function and can be considered constant. Thus, the time complexity is effectively O(n).

Space Complexity:
The space complexity is O(n), where n is the number of nodes. This is the space used for the parent array.
*/

class Solution {
  public:
    vector<int> findRedundantConnection(vector<vector<int>> &edges) {
        int n = edges.size();
        vector<int> parent(n + 1);

        // Initialize the parent array
        for (int i = 1; i <= n; ++i) {
            parent[i] = i;
        }

        // Iterate through the edges
        for (const auto &edge : edges) {
            int node1 = edge[0];
            int node2 = edge[1];
            // Find the parent of each node in the current edge
            int parent1 = findParent(parent, node1);
            int parent2 = findParent(parent, node2);

            // If both nodes have the same parent, return the current edge
            if (parent1 == parent2) {
                return edge;
            }
```

```
            // Union the nodes
            parent[parent1] = parent2;
        }

        return {};
    }

  private:
    int findParent(vector<int> &parent, int node) {
        if (parent[node] != node) {
            parent[node] = findParent(parent, parent[node]);
        }

        return parent[node];
    }
};
```

## 90. 11_Graphs/11_Number_of_Connected_Components_in_an_Undirected_Graph/03

```
/*
Problem: LeetCode 323 - Number of Connected Components in an Undirected Graph

Description:
Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of
nodes), write a function to find the number of connected components in an undirected graph.

Intuition:
This problem can be approached as a graph problem where the nodes represent the vertices and the
edges represent the connections between the vertices. We can use depth-first search (DFS) or
breadth-first search (BFS) to explore the graph and count the number of connected components.

Approach:
1. Build an adjacency list representation of the graph using the given edges.
2. Initialize a visited array to track the visited nodes during the graph traversal.
3. Initialize a count variable to keep track of the number of connected components.
4. Iterate through each node in the graph:
   - If the node is not visited, perform a DFS or BFS traversal from that node:
     - Increment the count by 1.
     - Mark all the connected nodes as visited.
5. Return the count, which represents the number of connected components.

Time Complexity:
The time complexity depends on the graph traversal algorithm used. Using DFS or BFS, the time
complexity is O(V + E), where V is the number of nodes (vertices) and E is the number of edges. We
visit each node and edge once.

Space Complexity:
The space complexity is O(V + E), where V is the number of nodes (vertices) and E is the number of
edges. This is the space used for the adjacency list and the visited array.
*/

class Solution {
  public:
    int countComponents(int n, vector<vector<int>> &edges) {
        vector<vector<int>> graph(n);   // Adjacency list representation of the graph
        vector<int> visited(n, 0);      // Visited array to track the visited nodes
        int count = 0;                  // Number of connected components

        // Build the graph
        for (const auto &edge : edges) {
            int node1 = edge[0];
            int node2 = edge[1];
            graph[node1].push_back(node2);
            graph[node2].push_back(node1);
        }

        // Perform graph traversal to count the connected components
        for (int i = 0; i < n; ++i) {
            if (visited[i] == 0) {
                ++count;
                dfs(i, graph, visited);
                // Or use bfs(i, graph, visited) for BFS traversal
            }
        }

        return count;
    }

  private:
```

```
    void dfs(int node, vector<vector<int>> &graph, vector<int> &visited) {
        visited[node] = 1;  // Mark the current node as visited

        // Perform DFS traversal on the neighbors
        for (int neighbor : graph[node]) {
            if (visited[neighbor] == 0) {
                dfs(neighbor, graph, visited);
            }
        }
    }
};
```

## 91. 11_Graphs/12_Graph_Valid_Tree/0261-graph-valid-tree.cpp

```
/*
Problem: LeetCode 261 - Graph Valid Tree

Description:
You have a graph of n nodes labeled from 0 to n - 1. You are given an integer n and a list of edges where edges[i] = [ai, bi] indicates that there is an undirected edge between nodes ai and bi in the graph.
Return true if the edges of the given graph make up a valid tree, and false otherwise.

Intuition:
A valid tree is a connected graph with no cycles. To determine if the given graph is a valid tree, we can perform a depth-first search (DFS) or breadth-first search (BFS) traversal and check if there are any cycles and if all the nodes are reachable from a single source node.

Approach:
1. Build an adjacency list representation of the graph using the given edges.
2. Initialize a visited array to track the visited nodes during the graph traversal.
3. Perform a DFS or BFS traversal from any node in the graph:
   - Mark the current node as visited.
   - Visit all the neighbors of the current node:
     - If a neighbor is already visited and not the parent of the current node, it means there is a cycle, so return false.
     - If a neighbor is not visited, perform a recursive DFS or enqueue it in the BFS queue.
4. After the traversal, check if all the nodes are visited. If not, return false.
5. Return true if no cycle is found and all nodes are visited.

Time Complexity:
The time complexity depends on the graph traversal algorithm used. Using DFS or BFS, the time complexity is O(V + E), where V is the number of nodes (vertices) and E is the number of edges. We visit each node and edge once.

Space Complexity:
The space complexity is O(V + E), where V is the number of nodes (vertices) and E is the number of edges. This is the space used for the adjacency list and the visited array.
*/

class Solution {
  public:
    bool validTree(int n, vector<vector<int>> &edges) {
        vector<vector<int>> graph(n);   // Adjacency list representation of the graph
        vector<int> visited(n, 0);      // Visited array to track the visited nodes

        // Build the graph
        for (const auto &edge : edges) {
            int node1 = edge[0];
            int node2 = edge[1];
            graph[node1].push_back(node2);
            graph[node2].push_back(node1);
        }

        // Perform graph traversal to check for cycles and reachability
        if (!dfs(0, -1, graph, visited)) {
            return false;
        }

        // Check if all nodes are visited
        for (int i = 0; i < n; ++i) {
            if (visited[i] == 0) {
                return false;
            }
```

```
        }

        return true;
    }

  private:
    bool dfs(int node, int parent, vector<vector<int>> &graph, vector<int> &visited) {
        visited[node] = 1;  // Mark the current node as visited

        // Perform DFS traversal on the neighbors
        for (int neighbor : graph[node]) {
            if (visited[neighbor] == 0) {
                if (!dfs(neighbor, node, graph, visited)) {
                    return false;
                }
            } else if (neighbor != parent) {
                return false;
            }
        }

        return true;
    }
};
```

## 92. 11_Graphs/13_Word_Ladder/0127-word-ladder.cpp

```
/*
Problem: LeetCode 127 - Word Ladder

Description:
Given two words, beginWord and endWord, and a dictionary wordList, return the length of the shortest transformation sequence from beginWord to endWord, such that:
- Only one letter can be changed at a time.
- Each transformed word must exist in the wordList.

If there is no such transformation sequence, return 0.

Intuition:
This problem can be solved using a graph traversal algorithm such as breadth-first search (BFS). We can treat each word as a node in the graph and connect the words that differ by a single character. The problem then reduces to finding the shortest path from the beginWord to the endWord.

Approach:
1. Build a set from the wordList for efficient lookup.
2. Create a queue for BFS traversal and initialize it with the beginWord.
3. Initialize a distance variable to track the length of the transformation sequence.
4. Perform BFS:
   - Pop the front word from the queue.
   - For each character position in the word, replace it with each of the 26 alphabets and check if the resulting word is in the wordList.
   - If the resulting word is the endWord, return the current distance + 1.
   - If the resulting word is in the wordList, add it to the queue and remove it from the wordList to avoid revisiting.
5. If no transformation sequence is found, return 0.

Time Complexity:
The time complexity is O(N * M^2), where N is the number of words in the wordList and M is the length of the words. In the worst case, we may need to explore all possible transformations of each word.

Space Complexity:
The space complexity is O(N * M^2), where N is the number of words in the wordList and M is the length of the words. This is the space used for the wordList set, queue, and visited set.
*/

class Solution {
  public:
    int ladderLength(string beginWord, string endWord, vector<string> &wordList) {
        unordered_set<string> wordSet(wordList.begin(), wordList.end());  // Convert wordList to a set for efficient lookup

        if (wordSet.find(endWord) == wordSet.end()) {
            return 0;  // endWord is not in the wordList, no transformation sequence possible
        }

        queue<string> q;
        q.push(beginWord);
        int distance = 1;

        while (!q.empty()) {
            int levelSize = q.size();

            for (int i = 0; i < levelSize; ++i) {
                string currentWord = q.front();
                q.pop();
```

```
                    // Check each character position of the word and replace it with all possible
alphabets
                for (int j = 0; j < currentWord.length(); ++j) {
                    char originalChar = currentWord[j];

                    for (char c = 'a'; c <= 'z'; ++c) {
                        currentWord[j] = c;

                        if (currentWord == endWord) {
                                return distance + 1;  // Transformation sequence found, return the
distance
                        }

                        if (wordSet.find(currentWord) != wordSet.end()) {
                            q.push(currentWord);  // Add the transformed word to the queue
                              wordSet.erase(currentWord);  // Remove the transformed word from the
wordSet
                        }
                    }

                    currentWord[j] = originalChar;  // Revert back the character
                }
            }

            ++distance;  // Increment the distance for each level of BFS traversal
        }

        return 0;  // No transformation sequence found
    }
};

// Bidirectional BFS
/*
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
         unordered_set<string> wordSet(wordList.begin(), wordList.end()); // Convert wordList to a
set for efficient lookup

        if (wordSet.find(endWord) == wordSet.end()) {
            return 0; // endWord is not in the wordList, no transformation sequence possible
        }

          unordered_set<string> beginSet, endSet; // Sets for the current level of traversal from
the beginWord and endWord
        beginSet.insert(beginWord);
        endSet.insert(endWord);

        int distance = 1; // Distance between beginWord and endWord

        while (!beginSet.empty() && !endSet.empty()) {
            // Always expand the smaller set to reduce the search space
            if (beginSet.size() > endSet.size()) {
                swap(beginSet, endSet);
            }

            unordered_set<string> temp; // Temporary set to store the next level of words

            for (const string& word : beginSet) {
                string currentWord = word;

                    // Check each character position of the word and replace it with all possible
alphabets
```

```
                for (int i = 0; i < currentWord.length(); ++i) {
                    char originalChar = currentWord[i];

                    for (char c = 'a'; c <= 'z'; ++c) {
                        currentWord[i] = c;

                        if (endSet.find(currentWord) != endSet.end()) {
                                return distance + 1; // Transformation sequence found, return the
distance
                        }

                        if (wordSet.find(currentWord) != wordSet.end()) {
                                temp.insert(currentWord); // Add the transformed word to the next
level set
                               wordSet.erase(currentWord); // Remove the transformed word from the
wordSet
                        }
                    }

                    currentWord[i] = originalChar; // Revert back the character
                }
            }

            swap(beginSet, temp); // Update the current level set with the next level set
            ++distance; // Increment the distance for each level of BFS traversal
        }

        return 0; // No transformation sequence found
    }
};
*/
```

# 93. 12_Advanced_Graphs/01_Reconstruct_Itinerary/0332-reconstruct-itinerary.cpp

```
/*
Problem: LeetCode 332 - Reconstruct Itinerary

Description:
Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order.
All of the tickets belong to a man who departs from "JFK", thus the itinerary must begin with "JFK". If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string.

Intuition:
To reconstruct the itinerary, we need to find a valid sequence of flights that starts from "JFK" and covers all the tickets. We can approach this problem as a graph traversal, where each airport is a node, and each ticket is an edge from one airport to another.

Approach:
1. Create a graph representation using a hashmap, where the key is the departure airport, and the value is a list of arrival airports. The list should be sorted in lexical order to find the smallest lexical order itinerary.
2. Perform a Depth-First Search (DFS) traversal on the graph, starting from the "JFK" airport.
3. During the DFS, for each airport visited, remove the used ticket (edge) from the graph to avoid using it again.
4. Continue the DFS until all the tickets are used up, and we have a valid itinerary.

Time Complexity:
The time complexity of the DFS approach is O(E * log E), where E is the total number of tickets. Sorting the list of arrival airports for each departure airport takes O(log E) time.

Space Complexity:
The space complexity is O(E), where E is the total number of tickets. We use a hashmap to represent the graph, which can contain E entries.
*/

class Solution {
  public:
    vector<string> findItinerary(vector<vector<string>> &tickets) {
        // Create a graph representation using hashmap
        unordered_map<string, multiset<string>> graph;

        for (const auto &ticket : tickets) {
            graph[ticket[0]].insert(ticket[1]);
        }

        vector<string> itinerary;
        dfs("JFK", graph, itinerary);
        // Reverse the itinerary to get the correct order
        reverse(itinerary.begin(), itinerary.end());
        return itinerary;
    }

       void dfs(string airport, unordered_map<string, multiset<string>> &graph, vector<string> &itinerary) {
        while (!graph[airport].empty()) {
            string nextAirport = *graph[airport].begin();
            graph[airport].erase(graph[airport].begin());
            dfs(nextAirport, graph, itinerary);
        }

        itinerary.push_back(airport);
    }
```

```
};

/*
// Beats 100% Runtime
class Solution {
public:
    // Using a map to represent the graph where key is the departure airport
    // and value is a priority queue to store the arrival airports in sorted order
    unordered_map<string, priority_queue<string, vector<string>, greater<string>>> airportGraph;
    vector<string> itinerary;

    void dfs(string airport) {
        auto& destinations = airportGraph[airport];
        while (!destinations.empty()) {
            string nextAirport = destinations.top();
            destinations.pop();
            dfs(nextAirport);
        }
        itinerary.push_back(airport);
    }

    vector<string> findItinerary(vector<vector<string>>& tickets) {
        // Build the graph
        for (const auto& ticket : tickets) {
            airportGraph[ticket[0]].push(ticket[1]);
        }

        // Start DFS from "JFK" airport
        dfs("JFK");

        // Reverse the itinerary to get the correct order
        reverse(itinerary.begin(), itinerary.end());

        return itinerary;
    }
};
*/
```

## 94. 12_Advanced_Graphs/02_Min_Cost_to_Connect_All_Points/1584-min-cost-to-con

```
/*
Problem: LeetCode 1584 - Min Cost to Connect All Points

Description:
You are given an array points representing integer coordinates of some points on a 2D-plane,
where points[i] = [xi, yi]. The cost of connecting two points [xi, yi] and [xj, yj] is the
manhattan distance between them: |xi - xj| + |yi - yj|, where |val| denotes the absolute value of
val.
Return the minimum cost to make all points connected. All points are connected if there is exactly
one simple path between any two points.

Intuition:
To connect all the points, we need to find the minimum spanning tree (MST) of the graph. The MST
is a
subgraph that connects all vertices together with the minimum possible total edge weight.

Approach:
1. Create a graph representation using an adjacency matrix, where graph[i][j] represents the
manhattan
   distance between point i and point j.
2. Use Prim's algorithm to find the MST of the graph.
3. Start the MST with an arbitrary point (let's say the first point), and keep adding the nearest
   non-visited point until all points are visited.
4. Calculate the total cost of the MST, which will be the minimum cost to connect all points.

Time Complexity:
The time complexity of Prim's algorithm is O(V^2), where V is the number of vertices (points).

Space Complexity:
The space complexity is O(V^2), where V is the number of vertices (points). We use an adjacency
matrix
to represent the graph.
*/

class Solution {
  public:
    int minCostConnectPoints(vector<vector<int>> &points) {
        int n = points.size();
        vector<bool> visited(n, false);
        vector<int> minCost(n, INT_MAX);
        // A lambda function to calculate the Manhattan distance between two points
        auto getManhattanDistance = [](const vector<int> &p1, const vector<int> &p2) {
            return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1]);
        };
        int result = 0;
        minCost[0] = 0; // Start with the first point

        for (int i = 0; i < n; ++i) {
            int u = -1;

            // Find the nearest non-visited point
            for (int j = 0; j < n; ++j) {
                if (!visited[j] && (u == -1 || minCost[j] < minCost[u])) {
                    u = j;
                }
            }

            visited[u] = true;
            result += minCost[u];
```

```
            // Update the minimum cost for the remaining points
            for (int j = 0; j < n; ++j) {
                if (!visited[j]) {
                    minCost[j] = min(minCost[j], getManhattanDistance(points[u], points[j]));
                }
            }
        }

        return result;
    }
};

/*
   Lambda Function in C++:

   A lambda function, also known as an anonymous function or a lambda expression, is a compact and inline way
    to define small functions in C++. It allows you to create function objects (functors) on the fly without
   explicitly defining a named function. Lambda functions are particularly useful when you need a simple function
   that you don't want to define separately.

   The syntax for a lambda function is as follows:

   [ captures ] ( parameters ) -> return_type {
       // function body
   }

    - `captures`: This is an optional part that allows the lambda function to capture and use variables from the
       surrounding scope. It can be used to access local variables, class members, or global variables within the lambda.

    - `parameters`: These are the input parameters of the lambda function, similar to regular function parameters.

    - `return_type`: This specifies the return type of the lambda function. If the return type is not specified, it
     will be deduced automatically by the compiler.

    - `function body`: This is the code that defines the behavior of the lambda function. It is similar to the body
     of a regular function.

   Example of a simple lambda function that adds two integers:

   auto add = [](int a, int b) -> int {
       return a + b;
   };

   int result = add(3, 5); // result will be 8

     Lambda functions provide a concise and efficient way to define short, local functions, improving the readability
   of your code and reducing the need for creating separate named functions.
*/
```

## 95. 12_Advanced_Graphs/03_Network_Delay_Time/0743-network-delay-time.cpp

```
/*
Problem: LeetCode 743 - Network Delay Time

Description:
You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times
as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node,
and wi is the time it takes for a signal to travel from source to target.
We will send a signal from a given node k. Return the time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Intuition:
This problem can be solved using Dijkstra's algorithm, which finds the shortest paths
from a source node to all other nodes in a weighted graph.

Approach:
1. Create a graph representation using an adjacency list, where graph[u] contains a list
   of pairs (v, w), representing an edge from node u to node v with weight w.
2. Use Dijkstra's algorithm to find the shortest path from the source node k to all other
   nodes in the graph.
3. Initialize a min heap (priority queue) to keep track of the nodes to visit, and set the
   distance of the source node k to 0 and distances of all other nodes to infinity.
4. Push the source node k into the min heap.
5. While the min heap is not empty, pop the node with the minimum distance from the heap.
6. For each neighbor of the current node, update the distance if a shorter path is found
   and push the neighbor into the min heap if it hasn't been visited yet.
7. After processing all nodes, the distance array will contain the shortest time to reach
   each node from the source node k. Return the maximum value in the distance array, which
   represents the time it takes for all nodes to receive the signal.
8. If there are nodes that cannot be reached from the source node k, the distance array
   will have some nodes with infinite distance. In this case, return -1.

Time Complexity:
The time complexity of Dijkstra's algorithm is O(E*logV), where E is the number of edges
and V is the number of vertices.

Space Complexity:
The space complexity is O(V+E), where V is the number of vertices and E is the number of edges,
due to the adjacency list representation of the graph.
*/

class Solution {
  public:
    int networkDelayTime(vector<vector<int>> &times, int n, int k) {
        // Create a graph representation using an adjacency list
        unordered_map<int, vector<pair<int, int>>> graph;

        for (const auto &time : times) {
            graph[time[0]].push_back({time[1], time[2]});
        }

        // Initialize a min heap (priority queue) to track nodes to visit
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
        vector<int> distance(n + 1, INT_MAX);
        // Set distance of source node k to 0 and push it into the min heap
        distance[k] = 0;
        pq.push({0, k});

        while (!pq.empty()) {
            auto [dist, node] = pq.top();
```

```
            pq.pop();

            // Check if the node has been visited already
            if (dist > distance[node]) {
                continue;
            }

            // Update the distances for neighbors
            for (const auto &neighbor : graph[node]) {
                int nextNode = neighbor.first;
                int newDist = dist + neighbor.second;

                if (newDist < distance[nextNode]) {
                    distance[nextNode] = newDist;
                    pq.push({newDist, nextNode});
                }
            }
        }

          // Find the maximum distance, which represents the time for all nodes to receive the
signal
        int maxDist = 0;

        for (int i = 1; i <= n; ++i) {
            maxDist = max(maxDist, distance[i]);
        }

        // If some nodes cannot be reached, return -1
        return maxDist == INT_MAX ? -1 : maxDist;
    }
};
```

## 96. 12_Advanced_Graphs/04_Swim_in_Rising_Water/0778-swim-in-rising-water.cpp

```
/*
Problem: LeetCode 778 - Swim in Rising Water

Description:
On an N x N grid, each cell is either empty (0) or blocked (1).
A move consists of walking from one empty cell to another empty cell adjacent to it
in one of the 4 cardinal directions (up, down, left, right).
Time starts at 0, and each time you visit an empty cell, you walk to an adjacent empty
cell and increase time by 1.
The grid is said to be unreachable if we cannot walk from the top-left corner of the grid
(0, 0) to the bottom-right corner of the grid (N-1, N-1) without walking through any
blocked cells.
Return the minimum time required to reach the bottom-right corner of the grid, or -1 if
the grid is unreachable.

Intuition:
This problem can be solved using a binary search approach. We can search for the minimum
time required to reach the bottom-right corner, and then verify if it is possible to
reach the destination using that time.

Approach:
1. Implement a Depth-First Search (DFS) function that explores the grid to check if it is
   possible to reach the destination within a given time t. The DFS function takes the
   current position (x, y), the time t, the grid, and a visited set to track visited cells.
2. In the DFS function, check if the current position is out of bounds or blocked, and
   return false if so.
3. Check if the current position is the destination (bottom-right corner) and return true
   if so.
4. Mark the current position as visited and recursively call the DFS function for all
   adjacent empty cells (up, down, left, right) with time t as the parameter.
5. If any of the recursive calls return true, it means we can reach the destination within
   time t, so return true.
6. If none of the recursive calls return true, return false, indicating that it is not
   possible to reach the destination within time t.
7. Now, use a binary search to find the minimum time required to reach the destination.
   The search range is from 0 to the maximum height in the grid.
8. While the low is less than or equal to the high, calculate the mid as (low + high) / 2
   and call the DFS function with time mid to check if it is possible to reach the
   destination within mid time.
9. If the DFS function returns true, it means it is possible to reach the destination
   within mid time, so set high to mid - 1 to search for smaller time.
10. Otherwise, set low to mid + 1 to search for larger time.
11. After the binary search, return low as the minimum time required to reach the
    destination if reachable, or -1 if unreachable.

Time Complexity:
The time complexity of the binary search is O(log N * N), where N is the side length of
the grid. The DFS function visits all empty cells in the grid, so the overall time
complexity is O(N^2).

Space Complexity:
The space complexity is O(N^2) due to the visited set used in the DFS function.
*/

class Solution {
  public:
    bool dfs(int x, int y, int t, vector<vector<int>> &grid) {
        int n = grid.size();

        // Check if current position is out of bounds or blocked
```

```
        if (x < 0 || x >= n || y < 0 || y >= n || grid[x][y] > t) {
            return false;
        }

        // Check if current position is destination
        if (x == n - 1 && y == n - 1) {
            return true;
        }

        int prev = grid[x][y];
        grid[x][y] = -1; // Mark the cell as visited
        // Recursively call DFS for all adjacent empty cells
        bool canReach = dfs(x + 1, y, t, grid) ||
                        dfs(x - 1, y, t, grid) ||
                        dfs(x, y + 1, t, grid) ||
                        dfs(x, y - 1, t, grid);
        grid[x][y] = prev; // Reset the cell to its original value
        return canReach;
    }

    int swimInWater(vector<vector<int>> &grid) {
        int n = grid.size();
        int low = 0, high = n * n - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            // Check if it is possible to reach destination within time mid
            if (dfs(0, 0, mid, grid)) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }

        return low;
    }
};

/*
// Beats 100% Runtime

class Solution {
public:
    int swimInWater(vector<vector<int>>& grid) {
        int n = grid.size();
         int left = max(grid[0][0], grid[n - 1][n - 1]); // Smallest possible value to start the
binary search.
        int right = n * n - 1; // Largest possible value to end the binary search.

        while (left <= right) {
            int mid = (left + right) / 2;
            vector<vector<bool>> visited(n, vector<bool>(n, false));
            if (dfs(grid, visited, 0, 0, mid)) {
                // If it's possible to reach the destination with the threshold "mid",
                 // try to find a better (smaller) threshold in the left half of the binary search
space.
                right = mid - 1;
            } else {
                // If it's not possible to reach the destination with the threshold "mid",
                   // increase the threshold by searching in the right half of the binary search
space.
                left = mid + 1;
            }
```

```
        }
        // At the end of the binary search, "right" will be the smallest threshold that
        // makes it possible to reach the destination (0-indexed), so return "right + 1".
        return right + 1;
    }

private:
    const vector<int> DIR{0, 1, 0, -1, 0};

       //  Depth-First  Search  function  to  check  if  it's  possible  to  reach  the  destination
(bottom-right cell)
    // from the current cell (i, j) with a maximum threshold of k.
    bool dfs(vector<vector<int>>& grid, vector<vector<bool>>& visited, int i, int j, int k) {
        int n = grid.size();
        if (i == n - 1 && j == n - 1) {
            // We reached the destination, return true.
            return true;
        }
        if (visited[i][j]) {
            // We already visited this cell, return false.
            return false;
        }
        visited[i][j] = true;

        // Explore all possible directions from the current cell.
        for (int l = 0; l < DIR.size() - 1; ++l) {
            int x = i + DIR[l];
            int y = j + DIR[l + 1];
            if (x >= 0 && x < n && y >= 0 && y < n && !visited[x][y] && grid[x][y] <= k) {
                if (dfs(grid, visited, x, y, k)) {
                    return true;
                }
            }
        }
        return false;
    }
};
*/
```

## 97. 12_Advanced_Graphs/05_Alien_Dictionary/0269-alien-dictionary.cpp

```
/*
Problem: LeetCode 269 - Alien Dictionary

Description:
Given a list of words in the alien language, find the order of the characters in this language.

The alien language is represented by a given dictionary, which will have words in an arbitrary order.
Each word consists of lowercase letters ('a' to 'z') and will not have duplicate characters.

It is guaranteed that no two words in the dictionary have the same ordering of letters.

If the order is invalid, return an empty string. There may be multiple valid orderings, you should return the smallest one in lexicographical order.

Intuition:
This problem can be solved using topological sorting. The given dictionary represents the relationship between characters in the alien language. The order of characters in the alien language can be determined by their relative positions in the words of the dictionary.

Approach:
1. Create a graph to represent the relationship between characters in the alien language.
2. Traverse the dictionary and add edges to the graph based on the order of characters in adjacent words.
3. Perform topological sorting on the graph to get the order of characters in the alien language.
4. Return the result as a string.

Time Complexity:
The time complexity of the solution is O(N), where N is the total number of characters in all the words in the dictionary. The reason is that we iterate through each character once to build the graph and perform topological sorting.

Space Complexity:
The space complexity of the solution is O(1) since we are using a fixed-size array of size 26 to represent the graph.

Topological Sorting:
- Intuition: Topological sorting is used to find the linear order of vertices in a directed acyclic graph (DAG) such that for every directed edge (u, v), vertex u comes before v in the ordering.
- Approach: We use a depth-first search (DFS) based approach to perform topological sorting. We start from a source vertex (a vertex with no incoming edges) and visit its neighbors recursively. After visiting all the neighbors, we add the current vertex to the result stack. The result stack will contain the vertices in the correct order.
*/

class Solution {
  public:
    string alienOrder(vector<string> &words) {
        // Initialize the graph and indegree array
        vector<vector<bool>> graph(26, vector<bool>(26, false));
        vector<int> indegree(26, -1);
        // Step 1: Build the graph and calculate indegrees
        buildGraph(words, graph, indegree);
        // Step 2: Perform topological sorting using DFS
        string result = topologicalSort(graph, indegree);
        return result;
    }

  private:
```

```
// Helper function to build the graph and calculate indegrees
void buildGraph(vector<string> &words, vector<vector<bool>> &graph, vector<int> &indegree) {
    for (string &word : words) {
        for (char c : word) {
            indegree[c - 'a'] = 0;
        }
    }

    for (int i = 1; i < words.size(); i++) {
        string prevWord = words[i - 1];
        string currWord = words[i];
        int len = min(prevWord.length(), currWord.length());

        for (int j = 0; j < len; j++) {
            char prevChar = prevWord[j];
            char currChar = currWord[j];

            if (prevChar != currChar) {
                if (!graph[prevChar - 'a'][currChar - 'a']) {
                    graph[prevChar - 'a'][currChar - 'a'] = true;
                    indegree[currChar - 'a']++;
                }

                break;
            }
        }
    }
}

// Helper function for topological sorting using DFS
string topologicalSort(vector<vector<bool>> &graph, vector<int> &indegree) {
    string result = "";
    stack<char> st;

    for (int i = 0; i < 26; i++) {
        if (indegree[i] == 0) {
            st.push('a' + i);
        }
    }

    while (!st.empty()) {
        char curr = st.top();
        st.pop();
        result += curr;

        for (int i = 0; i < 26; i++) {
            if (graph[curr - 'a'][i]) {
                indegree[i]--;

                if (indegree[i] == 0) {
                    st.push('a' + i);
                }
            }
        }
    }

    // If there are still edges in the graph, it means there is a cycle
    for (int i = 0; i < 26; i++) {
        if (indegree[i] > 0) {
            return "";
        }
    }

    return result;
```

```
    }
};

/*
// Another DFS solution

class Solution {
public:
    string alienOrder(vector<string>& words) {
        unordered_map<char, vector<char>> graph;
        unordered_map<char, int> indegree;

        // Step 1: Build the graph and calculate indegrees
        for (string& word : words) {
            for (char c : word) {
                indegree[c] = 0;
            }
        }

        for (int i = 1; i < words.size(); i++) {
            string prevWord = words[i - 1];
            string currWord = words[i];
            int len = min(prevWord.length(), currWord.length());

            for (int j = 0; j < len; j++) {
                char prevChar = prevWord[j];
                char currChar = currWord[j];

                if (prevChar != currChar) {
                    graph[prevChar].push_back(currChar);
                    indegree[currChar]++;
                    break;
                }
            }
        }

        // Step 2: Perform topological sorting using DFS
        string result;
        stack<char> st;

        for (const auto& entry : indegree) {
            if (entry.second == 0) {
                st.push(entry.first);
            }
        }

        while (!st.empty()) {
            char curr = st.top();
            st.pop();
            result += curr;

            for (char next : graph[curr]) {
                if (--indegree[next] == 0) {
                    st.push(next);
                }
            }
        }

        return result.size() == indegree.size() ? result : "";
    }
};
*/

/*
```

```
// BFS Solution

class Solution {
public:
    string alienOrder(vector<string>& words) {
        vector<vector<bool>> adjList(26, vector<bool>(26, false));
        vector<int> indegree(26, -1);

        // Step 1: Build the adjacency list and calculate indegrees
        buildGraph(words, adjList, indegree);

        // Step 2: Perform topological sorting using BFS
        string result = topologicalSort(adjList, indegree);

        return result;
    }

private:
    // Helper function to build the adjacency list and calculate indegrees
    void buildGraph(vector<string>& words, vector<vector<bool>>& adjList, vector<int>& indegree) {
        // Initialize the indegree for all characters
        for (string& word : words) {
            for (char c : word) {
                indegree[c - 'a'] = 0;
            }
        }

        // Compare adjacent words to build the adjacency list and update indegrees
        for (int i = 1; i < words.size(); i++) {
            string prevWord = words[i - 1];
            string currWord = words[i];
            int len = min(prevWord.length(), currWord.length());

            for (int j = 0; j < len; j++) {
                char prevChar = prevWord[j];
                char currChar = currWord[j];

                if (prevChar != currChar) {
                    if (!adjList[prevChar - 'a'][currChar - 'a']) {
                        adjList[prevChar - 'a'][currChar - 'a'] = true;
                        indegree[currChar - 'a']++;
                    }
                    break;
                }
            }
        }
    }

    // Helper function for topological sorting using BFS
    string topologicalSort(vector<vector<bool>>& adjList, vector<int>& indegree) {
        string result = "";

        queue<char> q;
        // Add characters with indegree 0 to the queue
        for (int i = 0; i < 26; i++) {
            if (indegree[i] == 0) {
                q.push('a' + i);
            }
        }

        // Perform BFS to get the topological ordering
        while (!q.empty()) {
            char curr = q.front();
            q.pop();
```

```
            result += curr;

            for (int i = 0; i < 26; i++) {
                if (adjList[curr - 'a'][i]) {
                    indegree[i]--;
                    if (indegree[i] == 0) {
                        q.push('a' + i);
                    }
                }
            }
        }

        // If there are still edges in the graph, it means there is a cycle
        if (result.length() != 26) {
            return "";
        }

        return result;
    }
};
*/
```

## 98. 12_Advanced_Graphs/06_Cheapest_Flights_Within_K_Stops/0787-cheapest-fligh

```
/*
Problem: LeetCode 787 - Cheapest Flights Within K Stops

Description:
There are n cities connected by m flights. Each flight starts from city u and arrives at city v with a price w.
Now given all the cities and flights, together with starting city src and the destination dst, your task is to find the cheapest price from src to dst with at most k stops. If there is no such route, return -1.

Intuition:
This problem can be solved using Dijkstra's algorithm with some modifications. Instead of stopping at the destination city, we need to find the cheapest price to reach the destination city within k stops. Therefore, we will modify Dijkstra's algorithm to allow k stops.

Approach:
1. Create an adjacency list to represent the graph where the key is the source city, and the value is a list of pairs containing the destination city and the price of the flight.
2. Use priority queue to implement Dijkstra's algorithm.
3. Push the starting city "src" into the priority queue with cost 0 and stops 0.
4. While the priority queue is not empty, do the following:
   - Pop the top element from the priority queue.
   - If the popped element is the destination city "dst", return the cost as the answer.
    - If the number of stops is less than or equal to "k", then iterate through the neighbors of the current city, and push them into the priority queue with updated cost and stops.
5. If we reach this point, it means there is no valid route, so return -1.

Time Complexity:
The time complexity of the Dijkstra's algorithm is O((E + V) * log(V)), where E is the number of flights and V is the number of cities. In the worst case, we can have E = V^2, so the time complexity is O((V^2) * log(V)).

Space Complexity:
The space complexity is O(E + V), where E is the number of flights and V is the number of cities. We use an adjacency list to represent the graph.
*/

class Solution {
  public:
    int findCheapestPrice(int n, vector<vector<int>> &flights, int src, int dst, int K) {
        vector<vector<pair<int, int>>> adjList(n);

        for (const auto &flight : flights) {
            adjList[flight[0]].emplace_back(flight[1], flight[2]);
        }

          // Initialize the cheapestCost vector with (n x K+1) dimensions and set all values to infinity
        vector<vector<int>> cheapestCost(n, vector<int>(K + 2, INT_MAX));
        priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;
        pq.push({0, src, 0});
        cheapestCost[src][0] = 0; // Cost to reach source with 0 stops is 0

        while (!pq.empty()) {
            vector<int> cur = pq.top();
            pq.pop();
            int cost = cur[0];
            int city = cur[1];
            int stops = cur[2];
```

```
            if (city == dst) {
                return cost;
            }

            if (stops <= K) {
                for (const auto &neighbor : adjList[city]) {
                    int neighborCity = neighbor.first;
                    int neighborCost = neighbor.second;

                    // Prune if the cost exceeds the current minimum cost
                    if (cost + neighborCost >= cheapestCost[neighborCity][stops + 1]) {
                        continue;
                    }

                    pq.push({cost + neighborCost, neighborCity, stops + 1});
                    cheapestCost[neighborCity][stops + 1] = cost + neighborCost;
                }
            }
        }

        return -1;
    }
};

/*
class Solution {
public:
    int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int k) {
        // Create an adjacency list to represent the graph
        vector<pair<int, int>> adj[n];
        for (int i = 0; i < flights.size(); i++) {
            adj[flights[i][0]].push_back({flights[i][1], flights[i][2]});
        }

        // Create a cost array to store the minimum cost to reach each node
        vector<int> cost(n, 1e9);
        queue<vector<int>> q; // Each entry contains: {stops, node, cost}

        cost[src] = 0; // Initialize the cost to reach the source node as 0
        q.push({0, src, 0}); // Push the source node with 0 stops and cost 0 into the queue

        while (!q.empty()) {
            auto v = q.front();
            q.pop();

            // If the number of stops exceeds k, skip this node
            if (v[0] > k) {
                continue;
            }

            for (auto it : adj[v[1]]) {
                int nbr = it.first; // Neighbor node
                int cst = it.second; // Cost to reach the neighbor node

                // If the new cost is less than the current recorded cost to reach the neighbor
                if (v[2] + cst < cost[nbr]) {
                    cost[nbr] = v[2] + cst; // Update the minimum cost to reach the neighbor
                     q.push({v[0] + 1, nbr, cost[nbr]}); // Push the neighbor with the updated cost
and one more stop into the queue
                }
            }
        }

        // If the cost to reach the destination is not infinity, return the cost
```

```
        if (cost[dst] != 1e9) {
            return cost[dst];
        }

        // Otherwise, return -1 (destination is not reachable)
        return -1;
    }
};
*/
```

# 99. 13_1-D_Dynamic_Programming/01_Climbing_Stairs/0070-climbing-stairs.cpp

```
/*
Problem: LeetCode 70 - Climbing Stairs

Description:
You are climbing a staircase that has n steps. You can either climb 1 or 2 steps at a time.
Return the number of distinct ways to climb to the top.

Intuition:
To reach the nth step, we can either take a single step from the (n-1)th step or take two steps from the (n-2)th step.
This forms the basis of our dynamic programming approach, as we can break down the problem into subproblems and build the solution from there.

Approach:
1. Initialize an array dp of size (n+1) to store the number of distinct ways to reach each step.
2. Set the base cases: dp[0] = 1 (no steps needed) and dp[1] = 1 (one step to reach the first step).
3. Iterate from 2 to n:
    - Compute dp[i] by summing up the number of ways to reach the previous two steps: dp[i] = dp[i-1] + dp[i-2].
4. Return dp[n], which represents the number of distinct ways to reach the top step.

Time Complexity:
The time complexity is O(n) since we iterate through the steps once.

Space Complexity:
The space complexity is O(n) since we use an array of size (n+1) to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is finding the number of distinct ways to reach the current step.
- Recurrence Relation: dp[i] = dp[i-1] + dp[i-2], where dp[i] represents the number of distinct ways to reach the ith step.
- Base Case: dp[0] = 1 and dp[1] = 1, as there is only one way to reach the first two steps.
*/

class Solution {
  public:
    int climbStairs(int n) {
        if (n <= 2) {
            return n;  // Base cases
        }

        vector<int> dp(n + 1);
        dp[0] = 1;
        dp[1] = 1;

        for (int i = 2; i <= n; ++i) {
            dp[i] = dp[i - 1] + dp[i - 2];  // Recurrence relation
        }

        return dp[n];
    }
};

// O(1) Space
// class Solution {
// public:
//     int climbStairs(int n) {
//         if(n <= 2)
```

```
//             return n;

//         int first = 1, second = 2;

//         for(int  i = 2; i < n; i++) {
//             int temp = second;
//             second = first + second;
//             first = temp;
//         }
//         return second;
//     }
// };
```

## 100. 13_1-D_Dynamic_Programming/02_Min_Cost_Climbing_Stairs/0746-min-cost-cli

```
/*
Problem: LeetCode 746 - Min Cost Climbing Stairs

Description:
You are given an integer array cost where cost[i] is the cost of ith step on a staircase.
Once you pay the cost, you can either climb one or two steps.
You can either start from the 0th step or the 1st step.
Return the minimum cost to reach the top of the floor.

Intuition:
To reach the top of the floor with minimum cost, we can consider dynamic programming.
At each step, we have two options: either take one step from the current step or take two steps from the previous step.
We want to minimize the total cost, so we choose the minimum cost between these two options.

Approach:
1. Create an array dp of size (n+1), where n is the size of the cost array.
   - dp[i] represents the minimum cost to reach the ith step.
2. Set the base cases: dp[0] = cost[0] and dp[1] = cost[1].
3. Iterate from 2 to n:
   - Compute dp[i] by taking the minimum cost between the (i-1)th step and the (i-2)th step, plus the cost of the current step:
     dp[i] = min(dp[i-1], dp[i-2]) + cost[i].
4. Return the minimum cost between reaching the (n-1)th step and the nth step: min(dp[n-1], dp[n]).

Time Complexity:
The time complexity is O(n), where n is the size of the cost array. We iterate through the cost array once.

Space Complexity:
The space complexity is O(n). We use an array of size (n+1) to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is finding the minimum cost to reach the current step.
- Recurrence Relation: dp[i] = min(dp[i-1], dp[i-2]) + cost[i], where dp[i] represents the minimum cost to reach the ith step.
- Base Case: dp[0] = cost[0] and dp[1] = cost[1], as the minimum cost to reach the first two steps is the cost of those steps.
*/

class Solution {
  public:
    int minCostClimbingStairs(vector<int> &cost) {
        int n = cost.size();
        vector<int> dp(n + 1);
        dp[0] = cost[0];
        dp[1] = cost[1];

        for (int i = 2; i <= n; ++i) {
            dp[i] = min(dp[i - 1], dp[i - 2]) + cost[i];
        }

        return min(dp[n - 1], dp[n]);
    }
};

// O(1) space
// class Solution {
// public:
```

```
//     int minCostClimbingStairs(vector<int>& cost) {
//         int prev1 = cost[0];
//         int prev2 = cost[1];

//         for (int i = 2; i < cost.size(); i++) {
//             int current = cost[i] + min(prev1, prev2);
//             prev1 = prev2;
//             prev2 = current;
//         }

//         return min(prev1, prev2);
//     }
// };
```

# 101. 13_1-D_Dynamic_Programming/03_House_Robber/0198-house-robber.cpp

```
/*
Problem: LeetCode 198 - House Robber

Description:
You are a professional robber planning to rob houses along a street.
Each house has a certain amount of money stashed, and the only constraint stopping you from robbing each of them is that adjacent houses have a security system connected,
and it will automatically contact the police if two adjacent houses are broken into on the same night.
Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Intuition:
To maximize the amount of money robbed, we can consider dynamic programming.
At each house, we have two options: either rob the current house or skip it.
If we rob the current house, we cannot rob the previous house, so we take the maximum amount between the current house's value and the amount robbed from the previous house.
If we skip the current house, we take the maximum amount robbed from the previous house.
We want to maximize the total amount of money robbed, so we choose the maximum between these two options.

Approach:
1. Create an array dp of size (n+1), where n is the size of the nums array.
   - dp[i] represents the maximum amount of money that can be robbed up to the ith house.
2. Set the base cases: dp[0] = 0 and dp[1] = nums[0].
3. Iterate from 2 to n:
   - Compute dp[i] by taking the maximum amount between the current house's value plus the amount robbed from the house two steps back and the amount robbed from the previous house:
     dp[i] = max(dp[i-2] + nums[i-1], dp[i-1]).
4. Return dp[n], which represents the maximum amount of money that can be robbed.

Time Complexity:
The time complexity is O(n), where n is the size of the nums array. We iterate through the nums array once.

Space Complexity:
The space complexity is O(n). We use an array of size (n+1) to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is finding the maximum amount of money that can be robbed up to the current house.
- Recurrence Relation: dp[i] = max(dp[i-2] + nums[i-1], dp[i-1]), where dp[i] represents the maximum amount of money that can be robbed up to the ith house.
- Base Case: dp[0] = 0 and dp[1] = nums[0], as there is no house to rob initially, and robbing the first house is the only option.
*/

class Solution {
  public:
    int rob(vector<int> &nums) {
        int n = nums.size();
        vector<int> dp(n + 1);
        dp[0] = 0;
        dp[1] = nums[0];

        for (int i = 2; i <= n; ++i) {
            dp[i] = max(dp[i - 2] + nums[i - 1], dp[i - 1]);
        }

        return dp[n];
```

```
    }
};

// Space: O(1)

// class Solution {
// public:
//     int rob(vector<int>& nums) {
//         int prev = 0;
//         int curr = 0;
//         int next = 0;

//         for (int i = 0; i < nums.size(); i++) {
//             next = max(prev + nums[i], curr);
//             prev = curr;
//             curr = next;
//         }

//         return curr;
//     }
// };
```

# 102. 13_1-D_Dynamic_Programming/04_House_Robber_II/0213-house-robber-ii.cpp

```
/*
Problem: LeetCode 213 - House Robber II

Description:
You are a professional robber planning to rob houses along a street.
Each house has a certain amount of money stashed, and the only constraint stopping you from robbing each of them is that adjacent houses have a security system connected,
and it will automatically contact the police if two adjacent houses are broken into on the same night.
Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.
Note: This problem is a variation of LeetCode 198 - House Robber with a circular street, where the first and last houses are adjacent.

Intuition:
To maximize the amount of money robbed, we can consider dynamic programming.
The problem becomes more complex due to the circular nature of the street.
Since the first and last houses are adjacent, we have two possibilities: either rob the first house and skip the last house or skip the first house and rob the last house.
We can solve this problem by splitting it into two separate subproblems:
1. Rob houses from the first to the second-to-last house.
2. Rob houses from the second to the last house.
The maximum amount of money robbed will be the maximum between the two subproblems.

Approach:
1. If the size of the nums array is 1, return nums[0] as it is the only house.
2. Compute the maximum amount of money robbed by considering the two subproblems:
    a. Rob houses from the first to the second-to-last house using the same approach as in the House Robber problem (LeetCode 198).
   b. Rob houses from the second to the last house using the same approach as in the House Robber problem (LeetCode 198).
3. Return the maximum amount between the two subproblems.

Time Complexity:
The time complexity is O(n), where n is the size of the nums array. We iterate through the nums array twice: once for each subproblem.

Space Complexity:
The space complexity is O(1) since we use constant extra space to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is finding the maximum amount of money that can be robbed from a range of houses.
- Recurrence Relation: dp[i] = max(dp[i-2] + nums[i], dp[i-1]), where dp[i] represents the maximum amount of money that can be robbed up to the ith house.
- Base Case: dp[0] = nums[0] and dp[1] = max(nums[0], nums[1]), as the maximum amount to rob the first two houses depends on their values.
*/

class Solution {
  public:
    int rob(vector<int> &nums) {
        int n = nums.size();

        if (n == 1) {
            return nums[0];
        }

        int max1 = robRange(nums, 0, n - 2);
        int max2 = robRange(nums, 1, n - 1);
```

```
        return max(max1, max2);
    }

    int robRange(vector<int> &nums, int start, int end) {
        int prev1 = 0;
        int prev2 = 0;

        for (int i = start; i <= end; ++i) {
            int current = max(prev1 + nums[i], prev2);
            prev1 = prev2;
            prev2 = current;
        }

        return prev2;
    }
};
```

## 103. 13_1-D_Dynamic_Programming/05_Longest_Palindromic_Substring/0005-longes

```
/*
Problem: LeetCode 5 - Longest Palindromic Substring

Description:
Given a string s, return the longest palindromic substring in s.

Intuition:
To find the longest palindromic substring, we can consider dynamic programming.
A palindrome reads the same backward as forward, so we can use this property to solve the problem.
We define a 2D array dp, where dp[i][j] represents whether the substring from index i to j is a palindrome.
Using this definition, we can build the solution by considering smaller subproblems and expanding from there.

Approach:
1. Create a 2D boolean array dp of size (n x n), where n is the length of the input string s.
     - dp[i][j] will be true if the substring from index i to j is a palindrome, and false otherwise.
2. Initialize the base cases:
   - Set dp[i][i] to true, as single characters are palindromes.
     - Set dp[i][i+1] to true if s[i] is equal to s[i+1], as two identical characters are palindromes.
3. Iterate over the substring lengths from 3 to n:
   - Iterate over the starting index i from 0 to n - len:
     - Calculate the ending index j as i + len - 1.
     - Update dp[i][j] to true if s[i] is equal to s[j] and dp[i+1][j-1] is true.
4. Keep track of the longest palindromic substring by updating the start and end indices whenever a longer palindrome is found.
5. Return the substring from the start index to the end index.

Time Complexity:
The time complexity is O(n^2), where n is the length of the input string s. We iterate over the dp array, which has a size of n x n.

Space Complexity:
The space complexity is O(n^2) as well. We use a 2D array dp of size n x n to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is determining whether a substring from index i to j is a palindrome.
- Recurrence Relation: dp[i][j] = (s[i] == s[j]) && dp[i+1][j-1], where dp[i][j] represents whether the substring from index i to j is a palindrome.
- Base Case: dp[i][i] = true (single character is a palindrome) and dp[i][i+1] = (s[i] == s[i+1]) (two identical characters are a palindrome).
*/

class Solution {
  public:
    string longestPalindrome(string s) {
        int n = s.length();
        vector<vector<bool>> dp(n, vector<bool>(n, false));
        int start = 0;
        int maxLen = 1;

        // Base case: single characters are palindromes
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }

        // Base case: two identical characters are palindromes
```

```
        for (int i = 0; i < n - 1; i++) {
            if (s[i] == s[i + 1]) {
                dp[i][i + 1] = true;
                start = i;
                maxLen = 2;
            }
        }

        // Check for palindromes of length greater than 2
        for (int len = 3; len <= n; len++) {
            for (int i = 0; i < n - len + 1; i++) {
                int j = i + len - 1;

                if (s[i] == s[j] && dp[i + 1][j - 1]) {
                    dp[i][j] = true;
                    start = i;
                    maxLen = len;
                }
            }
        }

        return s.substr(start, maxLen);
    }
};

// Expanding around the center approach
// This approach does not utilize dynamic programming

// Consider each element as middle of a palindrome and keep expanding

// class Solution {
// public:
//     string longestPalindrome(string s) {
//         // Two possibilities, palindrome can be even or odd length
//         for(int i = 0; i < s.size() - 1; i++) {
//             // To check odd palindromes
//             helper(s, i, i);
//             // To check even palindromes
//             helper(s, i, i+1);
//         }

//         return s.substr(start, maxLength);
//     }

// private:
//     // need the index and length to cut the string using substr
//     int start = 0, maxLength = 1;
//     void helper(string S, int L, int R) {
//         while(L >= 0 && R < S.size() && S[L] == S[R])
//             L--, R++;

//         int len = R - L - 1;
//         if(len > maxLength) {
//             start = L + 1;
//             maxLength = len;
//         }
//     }
// };
```

## 104. 13_1-D_Dynamic_Programming/06_Palindromic_Substrings/0647-palindromic-su

```
/*
Problem: LeetCode 647 - Palindromic Substrings

Description:
Given a string s, return the number of palindromic substrings in s.
A substring is a contiguous sequence of characters within the string.

Intuition:
To count the number of palindromic substrings, we can use a two-pointer approach.
We iterate through each character in the string and treat it as a potential center of a palindrome.
By expanding from the center, we check if the substring formed is a palindrome and count it as a valid palindrome.

Approach:
1. Initialize a variable count to keep track of the number of palindromic substrings.
2. Iterate through each character in the string:
   - Consider each character as the center of a potential palindrome.
   - Expand around the center using two pointers, one on each side.
   - Count all valid palindromes found during expansion.
3. Return the count of palindromic substrings.

Time Complexity:
The time complexity is O(n^2), where n is the length of the input string s. We iterate through the string and perform expansion for each character.

Space Complexity:
The space complexity is O(1) since we only use a few variables to store indices and counts.
*/

// Better solution but not dynamic programming
class Solution {
  public:
    int countSubstrings(string s) {
        int count = 0;
        int n = s.length();

        for (int i = 0; i < n; i++) {
            count += countPalindromes(s, i, i);      // Odd-length palindromes
            count += countPalindromes(s, i, i + 1);  // Even-length palindromes
        }

        return count;
    }

    int countPalindromes(string s, int left, int right) {
        int count = 0;

        while (left >= 0 && right < s.length() && s[left] == s[right]) {
            count++;
            left--;
            right++;
        }

        return count;
    }
};

/*
Dynamic Programming Approach
```

```
Intuition:
To count the number of palindromic substrings, we can consider dynamic programming.
A palindrome reads the same backward as forward, so we can use this property to solve the problem.
We define a 2D array dp, where dp[i][j] represents whether the substring from index i to j is a palindrome.
Using this definition, we can build the solution by considering smaller subproblems and expanding from there.

Approach:
1. Create a 2D boolean array dp of size (n x n), where n is the length of the input string s.
    - dp[i][j] will be true if the substring from index i to j is a palindrome, and false otherwise.
2. Initialize the base cases:
   - Set dp[i][i] to true, as single characters are palindromes.
    - Set dp[i][i+1] to true if s[i] is equal to s[i+1], as two identical characters are palindromes.
3. Iterate over the substring lengths from 3 to n:
   - Iterate over the starting index i from 0 to n - len:
    - Calculate the ending index j as i + len - 1.
    - Update dp[i][j] to true if s[i] is equal to s[j] and dp[i+1][j-1] is true.
4. Count the number of true values in the dp array, which represents the number of palindromic substrings.

Time Complexity:
The time complexity is O(n^2), where n is the length of the input string s. We iterate over the dp array, which has a size of n x n.

Space Complexity:
The space complexity is O(n^2) as well. We use a 2D array dp of size n x n to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is determining whether a substring from index i to j is a palindrome.
- Recurrence Relation: dp[i][j] = (s[i] == s[j]) && dp[i+1][j-1], where dp[i][j] represents whether the substring from index i to j is a palindrome.
- Base Case: dp[i][i] = true (single character is a palindrome) and dp[i][i+1] = (s[i] == s[i+1]) (two identical characters are a palindrome).
*/

// class Solution {
// public:
//     int countSubstrings(string s) {
//         int n = s.length();
//         vector<vector<bool>> dp(n, vector<bool>(n, false));

//         int count = 0;

//         // Base case: single characters are palindromes
//         for (int i = 0; i < n; i++) {
//             dp[i][i] = true;
//             count++;
//         }

//         // Base case: two identical characters are palindromes
//         for (int i = 0; i < n - 1; i++) {
//             if (s[i] == s[i + 1]) {
//                 dp[i][i + 1] = true;
//                 count++;
//             }
//         }

//         // Check for palindromes of length greater than 2
//         for (int len = 3; len <= n; len++) {
```

```
//             for (int i = 0; i < n - len + 1; i++) {
//                 int j = i + len - 1;

//                 if (s[i] == s[j] && dp[i + 1][j - 1]) {
//                     dp[i][j] = true;
//                     count++;
//                 }
//             }
//         }

//         return count;
//     }
// };
```

## 105. 13_1-D_Dynamic_Programming/07_Decode_Ways/0091-decode-ways.cpp

```
/*
Problem: LeetCode 91 - Decode Ways

Description:
A message containing letters from A-Z can be encoded into numbers using the following mapping:
'A' -> "1", 'B' -> "2", ..., 'Z' -> "26".
Given a string s containing only digits, return the number of ways to decode it.
Note: The grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is not
mapped to any letter.

Intuition:
To count the number of ways to decode a given string, we can use dynamic programming.
The problem can be broken down into smaller subproblems where we consider different lengths of
substrings.
We can build the solution by considering the number of ways to decode smaller substrings and
combine them to get the final answer.

Approach:
1. Initialize an array dp of size n+1, where n is the length of the input string s.
   - dp[i] will represent the number of ways to decode the substring s[0:i].
2. Initialize dp[0] to 1 as there is one way to decode an empty string.
3. Iterate through the characters of the string:
     - If the current character is not '0', update dp[i] by adding dp[i-1] to account for
single-digit decoding.
   - If the current character, along with the previous character, forms a valid two-digit number,
update dp[i] by adding dp[i-2].
4. Return dp[n], which represents the number of ways to decode the entire string.

Time Complexity:
The time complexity is O(n), where n is the length of the input string s. We iterate through each
character of the string once.

Space Complexity:
The space complexity is O(n) as well. We use an array dp of size n+1 to store the intermediate
results.

Dynamic Programming:
- Subproblem: The subproblem is calculating the number of ways to decode a substring of length i.
- Recurrence Relation: dp[i] = dp[i-1] + dp[i-2], if the current character and the previous
character form a valid two-digit number.
- Base Case: dp[0] = 1 (empty string has one way to decode).
*/

class Solution {
  public:
    int numDecodings(string s) {
        int n = s.length();
        vector<int> dp(n + 1, 0);
        dp[0] = 1;  // Base case: empty string has one way to decode

        for (int i = 1; i <= n; i++) {
            // Check if current character is not '0'
            if (s[i - 1] != '0') {
                dp[i] += dp[i - 1];  // Add the number of ways for single-digit decoding
            }

            // Check if current character and the previous character form a valid two-digit number
            if (i > 1 && isValidTwoDigit(s[i - 2], s[i - 1])) {
                dp[i] += dp[i - 2];  // Add the number of ways for two-digit decoding
            }
```

```
        }

        return dp[n];
    }

    bool isValidTwoDigit(char c1, char c2) {
        int num = (c1 - '0') * 10 + (c2 - '0');
        return num >= 10 && num <= 26;
    }
};

// class Solution {
// public:
//     int numDecodings(string s) {
//         int n = s.size();
//         if (n == 0 || s[0] == '0') {
//             return 0;
//         }

//         vector<int> dp(n + 1);
//         dp[0] = 1;
//         dp[1] = 1;

//         for (int i = 2; i <= n; i++) {
//             // Single digit
//             int ones = stoi(s.substr(i - 1, 1));
//             if (0 < ones && ones < 10) {
//                 dp[i] += dp[i - 1];
//             }

//             // Double digit
//             int tens = stoi(s.substr(i - 2, 2));
//             if (tens >= 10 && tens <= 26) {
//                 dp[i] += dp[i - 2];
//             }
//         }
//         return dp[n];
//     }
// };
```

## 106. 13_1-D_Dynamic_Programming/08_Coin_Change/0322-coin-change.cpp

```
/*
Problem: LeetCode 322 - Coin Change

Description:
You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.
Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Intuition:
To find the fewest number of coins needed to make up a given amount, we can use dynamic programming.
The problem can be broken down into smaller subproblems by considering different coin denominations.
We can build the solution by considering the minimum number of coins needed for smaller amounts and combining them to find the answer.

Approach:
1. Initialize an array dp of size amount+1, where dp[i] represents the fewest number of coins needed to make up the amount i.
   - Initialize all elements of dp to infinity, except dp[0] which is set to 0.
2. Iterate through the coin denominations:
   - For each coin, iterate through the amounts from the coin value to the target amount.
   - Update dp[j] by taking the minimum between dp[j] and dp[j-coin] + 1.
3. Return dp[amount], which represents the fewest number of coins needed to make up the target amount.

Time Complexity:
The time complexity is O(amount * n), where amount is the target amount and n is the number of coins. We iterate through all possible amounts and all coin denominations.

Space Complexity:
The space complexity is O(amount) as we use an array dp to store the fewest number of coins needed for each amount.

Dynamic Programming:
- Subproblem: The subproblem is calculating the fewest number of coins needed to make up an amount i.
- Recurrence Relation: dp[i] = min(dp[i], dp[i-coin] + 1) for each coin in the coin denominations.
- Base Case: dp[0] = 0 (zero coins are needed to make up an amount of zero).
*/

class Solution {
  public:
    int coinChange(vector<int> &coins, int amount) {
          vector<int> dp(amount + 1, INT_MAX);  // DP array to store the minimum number of coins needed for each amount
        dp[0] = 0;  // Base case: zero coins are needed to make up an amount of zero

        for (int coin : coins) {  // Iterate through each coin
             for (int j = coin; j <= amount; j++) {  // Iterate through each amount from coin to the target amount
                 if (dp[j - coin] != INT_MAX) {  // Check if there is a valid previous amount (j - coin) that can be made up with coins
                     dp[j] = min(dp[j], dp[j - coin] + 1);  // Update dp[j] with the minimum number of coins needed
                 }
             }
        }
```

```
        return dp[amount] != INT_MAX ? dp[amount] : -1;  // If dp[amount] is still INT_MAX, return
-1 as the amount cannot be made with the given coins
    }
};

/**
 * ! Statement 1 -
 * * Since the coins array is sorted in ascending order, once we encounter a coin that is larger
than i, we can safely break out of the loop.
 *
 * ! Statement 2 -
 * * We are taking mimimum because there could be multiple ways of getting the current amount but
we want the lowest amt of coins
 *     * Ex: coins = [1, 3, 4, 7], so while iterating through first denomination ie 1
 *     * DP[3] will be 3, min(dp[3], dp[2] + 1) => min(MAX_INT, 2 + 1) => 3
 *     * But next iteration we find that 3 denomination exists
 *     * So,  min(DP[i], DP[i-coin] + 1) => min(3, 0 + 1) => DP[i] = 1
*/

// class Solution {
// public:
//     int coinChange(vector<int>& coins, int amount) {
//         // Create a DP array to store the minimum number of coins needed for each amount
//         vector<int> DP(amount + 1, INT_MAX);
//         // Set the base case: 0 coins are needed to make amount 0
//         DP[0] = 0;
//         sort(coins.begin(), coins.end());

//         for (int i = 1; i <= amount; i++) {
//             for (auto coin : coins) {
//                 // ! Statement 1
//                 if (coin > i)
//                     break;
//                 // ! Statement 2
//                 if (DP[i - coin] != INT_MAX)
//                     DP[i] = min(DP[i], DP[i - coin] + 1);
//             }
//         }

//          // If the final amount is still INT_MAX, it means amount cannot be made with the given
coins
//         return (DP[amount] == INT_MAX) ? -1 : DP[amount];
//     }
// };

// int coinChange(vector<int>& coins, int amount) {
//     vector<int> dp(amount + 1, amount + 1);

//     // Base case
//     dp[0] = 0;

//     for (int i = 1; i <= amount; ++i) {
//         for (int j = 0; j < coins.size(); ++j) {
//             if (coins[j] <= i) {
//                 dp[i] = min(dp[i], dp[i - coins[j]] + 1);
//             }
//         }
//     }

//     return dp[amount] > amount ? -1 : dp[amount];
// }
```

## 107. 13_1-D_Dynamic_Programming/09_Maximum_Product_Subarray/0152-maximum

```
/*
Problem: LeetCode 152 - Maximum Product Subarray

Description:
Given an integer array nums, find the contiguous subarray within an array (containing at least one
number) that has the largest product.

Intuition:
To find the maximum product subarray, we can use dynamic programming.
The problem can be broken down into smaller subproblems by considering different subarrays.
We can build the solution by considering the maximum product of subarrays ending at each position.

Approach:
1. Initialize variables maxProduct, minProduct, and result to store the maximum product, minimum
product, and final result, respectively.
2. Iterate through the array nums:
   - For each number, update maxProduct and minProduct:
      - maxProduct is the maximum of the current number, maxProduct * number, and minProduct *
number.
      - minProduct is the minimum of the current number, maxProduct * number, and minProduct *
number.
   - Update the result with the maximum of result and maxProduct.
3. Return the result, which represents the maximum product subarray.

Time Complexity:
The time complexity is O(n), where n is the size of the input array nums. We iterate through each
element of the array once.

Space Complexity:
The space complexity is O(1) as we only need a few variables to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is finding the maximum product of subarrays ending at each position.
- Recurrence Relation:
    - maxProduct = max(nums[i], maxProduct * nums[i], minProduct * nums[i])
    - minProduct = min(nums[i], maxProduct * nums[i], minProduct * nums[i])
- Base Case: Initialize maxProduct and minProduct with the first element of the array.
*/

class Solution {
  public:
    int maxProduct(vector<int> &nums) {
        int n = nums.size();
        int maxProduct = nums[0];  // Maximum product of subarrays ending at each position
        int minProduct = nums[0];  // Minimum product of subarrays ending at each position
        int result = nums[0];      // Final result

        for (int i = 1; i < n; i++) {
            // Update maxProduct and minProduct
            int tempMax = maxProduct;
            maxProduct = max({nums[i], maxProduct *nums[i], minProduct *nums[i]});
            minProduct = min({nums[i], tempMax *nums[i], minProduct *nums[i]});
            // Update the result
            result = max(result, maxProduct);
        }

        return result;
    }
};
```

## 108. 13_1-D_Dynamic_Programming/10_Word_Break/0139-word-break.cpp

```
/*
Problem: LeetCode 139 - Word Break

Description:
Given a string s and a dictionary of strings wordDict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

Intuition:
To determine if a string can be segmented into words from a dictionary, we can use dynamic programming.
The problem can be broken down into smaller subproblems by considering different prefixes of the input string.
We can build the solution by checking if each prefix of the string can be segmented using words from the dictionary.

Approach:
1. Initialize a vector dp of size n+1, where dp[i] represents whether the prefix s[0...i-1] can be segmented using words from the dictionary.
2. Initialize dp[0] as true, indicating that an empty string can be segmented.
3. Iterate through the string from left to right:
   - For each index i, iterate from 0 to i:
     - Check if dp[j] is true and the substring s[j...i-1] is in the dictionary.
     - If both conditions are satisfied, set dp[i] as true.
4. Return dp[n], which represents whether the entire string can be segmented.

Time Complexity:
The time complexity is O(n^2), where n is the length of the input string. We have nested loops to iterate through the string and its prefixes.

Space Complexity:
The space complexity is O(n) as we use an extra array dp of size n+1 to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is determining whether a prefix s[0...i-1] can be segmented using words from the dictionary.
- Recurrence Relation: dp[i] = dp[j] && (s[j...i-1] is in the dictionary) for 0 <= j < i.
- Base Case: dp[0] = true, indicating that an empty string can be segmented.
*/

class Solution {
  public:
    bool wordBreak(string s, vector<string> &wordDict) {
        int n = s.length();
         vector<bool> dp(n + 1, false); // Dynamic programming array to store if substring from 0 to i-1 can be segmented
        dp[0] = true;  // Base case: empty string can be segmented

        for (int i = 1; i <= n; i++) { // Iterate through each position in the string
             for (int j = 0; j < i; j++) { // Iterate over all possible substrings ending at index i
                  // Check if substring from j to i-1 is in the word dictionary and if substring from 0 to j-1 can be segmented
                    if (dp[j] && find(wordDict.begin(), wordDict.end(), s.substr(j, i - j)) != wordDict.end()) {
                    dp[i] = true; // Set dp[i] to true if substring from 0 to i-1 can be segmented
                    break; // No need to check further substrings ending at index i
                }
            }
        }
```

```
        return dp[n]; // Return whether the entire string can be segmented
    }
};

// class Solution {
// public:
//     bool wordBreak(string s, vector<string>& wordDict) {
//         int n = s.length();
//         vector<bool> dp(n, false);
//         dp[0] = true;

//         for (int i = 0; i <= n; i++) {
//             for (auto word: wordDict) {
//                 if (dp[i] && s.substr(i, word.size()).compare(word) == 0) {
//                     dp[i + word.size()] = true;
//                 }
//             }
//         }

//         return dp[n];
//     }
// };
```

## 109. 13_1-D_Dynamic_Programming/11_Longest_Increasing_Subsequence/0300-long

```
/*
Problem: LeetCode 300 - Longest Increasing Subsequence

Description:
Given an integer array nums, return the length of the longest strictly increasing subsequence.

Intuition:
To find the length of the longest increasing subsequence, we can use dynamic programming.
The problem can be broken down into smaller subproblems by considering different subarrays.
We can build the solution by considering the length of the longest increasing subsequence ending
at each position.

Approach:
1. Initialize an array dp of size n, where dp[i] represents the length of the longest increasing
subsequence ending at index i.
2. Initialize dp[i] as 1 for all indices, as the minimum length of an increasing subsequence is 1
(the element itself).
3. Iterate through the array nums:
   - For each index i, iterate from 0 to i-1:
     - If nums[i] is greater than nums[j], update dp[i] as the maximum of dp[i] and dp[j] + 1.
       This means that we consider extending the increasing subsequence ending at index j with the
current element at index i.
4. Return the maximum value in the dp array, which represents the length of the longest increasing
subsequence.

Time Complexity:
The time complexity is O(n^2), where n is the size of the input array nums. We have nested loops
to iterate through the array.

Space Complexity:
The space complexity is O(n) as we use an extra array dp of size n to store the lengths of the
increasing subsequences.

Dynamic Programming:
- Subproblem: The subproblem is finding the length of the longest increasing subsequence ending at
each position.
- Recurrence Relation: dp[i] = max(dp[i], dp[j] + 1) for 0 <= j < i, if nums[i] > nums[j].
- Base Case: Initialize dp[i] as 1 for all indices, as the minimum length of an increasing
subsequence is 1.
*/

class Solution {
  public:
    int lengthOfLIS(vector<int> &nums) {
        int n = nums.size();
        vector<int> dp(n, 1);  // Length of longest increasing subsequence ending at each index

        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
        }

        return *max_element(dp.begin(), dp.end());
    }
};

/**
```

```
 * ! Using Binary Search

class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size(); // Length of the input vector
        vector<int> dp; // Stores the current increasing subsequence

        // Add the first element of nums to the subsequence vector
        dp.push_back(nums[0]);

        // Iterate through the remaining elements of nums
        for(int i = 1; i < n; i++) {
            // If the current element is greater than the last element in dp,
            // it can extend the current increasing subsequence, so add it to dp
            if(nums[i] > dp.back()) {
                dp.push_back(nums[i]);
            } else {
                // If the current element is not greater, find its correct position
                // in dp using binary search (lower_bound)
                auto it = lower_bound(dp.begin(), dp.end(), nums[i]);
                // Update the value at the found position to nums[i]
                *it = nums[i];
            }
        }

        // Return the length of the longest increasing subsequence
        return dp.size();
    }
};
*/
```

## 110. 13_1-D_Dynamic_Programming/12_Partition_Equal_Subset_Sum/0416-partition-

```
/*
Problem: LeetCode 416 - Partition Equal Subset Sum

Description:
Given a non-empty array nums containing only positive integers, determine if it can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Intuition:
To determine if the array can be partitioned into two equal subsets, we can use dynamic programming.
The problem can be broken down into smaller subproblems by considering different subsets of the input array.
We can build the solution by checking if there exists a subset with a target sum equal to half of the total sum of the array.

Approach:
1. Calculate the total sum of the array.
2. If the total sum is odd, return false. It's not possible to partition the array into equal subsets.
3. Initialize a vector dp of size (sum/2) + 1, where dp[i] represents whether a subset with sum i can be formed.
4. Initialize dp[0] as true, indicating that an empty subset with sum 0 can be formed.
5. Iterate through the array nums:
   - For each number num, iterate from (sum/2) down to num:
      - If dp[i - num] is true, set dp[i] as true, indicating that a subset with sum i can be formed.
6. Return dp[sum/2], which represents whether a subset with sum (sum/2) can be formed.

Time Complexity:
The time complexity is O(n * sum), where n is the size of the input array and sum is the total sum of the array.

Space Complexity:
The space complexity is O(sum), as we use an extra array dp of size (sum/2) + 1 to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is determining whether a subset with a target sum can be formed using a subset of the array.
- Recurrence Relation: dp[i] = dp[i - num] for each num in the array, if dp[i - num] is true.
- Base Case: dp[0] = true, indicating that an empty subset with sum 0 can be formed.
*/

class Solution {
  public:
    bool canPartition(vector<int> &nums) {
        int n = nums.size();
        int sum = accumulate(nums.begin(), nums.end(), 0); // Calculate the total sum

        if (sum % 2 != 0) {
              return false; // If the total sum is odd, it's not possible to partition into equal subsets
        }

        int target = sum / 2;
        vector<bool> dp(target + 1, false); // Subset with sum i can be formed
        dp[0] = true; // Base case: Empty subset with sum 0 can be formed

        for (int num : nums) {
            for (int i = target; i >= num; i--) {
```

```
                if (dp[i - num]) {
                    dp[i] = true;
                }
            }
        }

        return dp[target];
    }
};
```

## 111. 14_2-D_Dynamic_Programming/01_Unique_Paths/0062-unique-paths.cpp

```
/*
Problem: LeetCode 62 - Unique Paths

Description:
A robot is located at the top-left corner of a m x n grid. 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.
How many possible unique paths are there?

Intuition:
To find the number of unique paths, we can use dynamic programming.
The problem can be broken down into smaller subproblems by considering different positions in the grid.
We can build the solution by counting the number of unique paths to reach each position.

Approach:
1. Initialize a 2D array dp of size m x n, where dp[i][j] represents the number of unique paths to reach position (i, j).
2. Initialize the first row and first column of dp to 1, as there is only one way to reach each position in the first row and first column.
3. Iterate through the grid starting from position (1, 1):
   - For each position (i, j), set dp[i][j] as the sum of dp[i-1][j] and dp[i][j-1].
     This means that the number of unique paths to reach (i, j) is the sum of the paths from the above position (i-1, j) and the left position (i, j-1).
4. Return dp[m-1][n-1], which represents the number of unique paths to reach the bottom-right corner of the grid.

Time Complexity:
The time complexity is O(m * n), where m and n are the dimensions of the grid.

Space Complexity:
The space complexity is O(m * n), as we use a 2D array to store the number of unique paths for each position.

Dynamic Programming:
- Subproblem: The subproblem is finding the number of unique paths to reach each position in the grid.
- Recurrence Relation: dp[i][j] = dp[i-1][j] + dp[i][j-1].
- Base Case: Initialize the first row and first column of dp to 1, as there is only one way to reach each position in the first row and first column.
*/

class Solution {
  public:
    int uniquePaths(int m, int n) {
          vector<vector<int>> dp(m, vector<int>(n, 1)); // Number of unique paths to reach each position

        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]; // Number of unique paths = paths from above + paths from left
            }
        }

        return dp[m - 1][n - 1]; // Number of unique paths to reach the bottom-right corner
    }
};
```

## 112. 14_2-D_Dynamic_Programming/02_Longest_Common_Subsequence/1143-longe

```
/*
Problem: LeetCode 1143 - Longest Common Subsequence

Description:
Given two strings text1 and text2, return the length of their longest common subsequence.
A subsequence of a string is a new string generated from the original string with some characters
(can be none) deleted
without changing the relative order of the remaining characters.

Intuition:
To find the longest common subsequence between two strings, we can use dynamic programming.
The problem can be broken down into smaller subproblems by considering different suffixes of the
strings.
We can build the solution by finding the longest common subsequence for each pair of suffixes.

Approach:
1. Initialize a 2D array dp of size (m+1) x (n+1), where dp[i][j] represents the length of the
longest common subsequence
   between the first i characters of text1 and the first j characters of text2.
2. Initialize the first row and first column of dp to 0.
3. Iterate through the characters of text1 and text2:
   - If the current characters are equal, dp[i][j] = dp[i-1][j-1] + 1.
     This means that we include the current character in the longest common subsequence.
   - Otherwise, dp[i][j] = max(dp[i-1][j], dp[i][j-1]).
       This means that we exclude one character either from text1 or text2 to find the longest
common subsequence.
4. Return dp[m][n], which represents the length of the longest common subsequence.

Time Complexity:
The time complexity is O(m * n), where m and n are the lengths of text1 and text2, respectively.

Space Complexity:
The space complexity is O(m * n), as we use a 2D array to store the length of the longest common
subsequence.

Dynamic Programming:
- Subproblem: The subproblem is finding the longest common subsequence for each pair of suffixes
of the strings.
- Recurrence Relation: If text1[i] == text2[j], dp[i][j] = dp[i-1][j-1] + 1. Otherwise, dp[i][j] =
max(dp[i-1][j], dp[i][j-1]).
- Base Case: Initialize the first row and first column of dp to 0.
*/

class Solution {
  public:
    int longestCommonSubsequence(string text1, string text2) {
        int m = text1.length();
        int n = text2.length();
            vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0)); // Length of longest common
subsequence

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (text1[i - 1] == text2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1; // Include the current character
                } else {
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]); // Exclude one character
                }
            }
        }
```

```
        return dp[m][n]; // Length of longest common subsequence
    }
};
```

## 113. 14_2-D_Dynamic_Programming/03_Best_Time_to_Buy_And_Sell_Stock_With_C

```
/*
Problem: LeetCode 309 - Best Time to Buy and Sell Stock with Cooldown

Description:
You are given an array prices where prices[i] is the price of a given stock on the ith 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.
You cannot buy and sell the stock on the same day (i.e., you must sell the stock before buying again).
Additionally, you must not participate in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Intuition:
To maximize the profit while taking into account the cooldown period, we can use dynamic programming.
At each day, we have three possible states: buy, sell, or rest.
The key idea is to track the maximum profit for each state and determine the optimal action to take on each day.

Approach:
1. Initialize three variables: buy (maximum profit if the stock is bought), sell (maximum profit if the stock is sold), and rest (maximum profit if no action is taken).
   Set buy = -prices[0], sell = 0, and rest = 0.
2. Iterate through the prices array starting from the second day:
   - Update the buy variable as the maximum between the previous buy value and the profit from the previous rest state minus the current stock price.
   - Update the sell variable as the maximum between the previous sell value and the profit from the previous buy state plus the current stock price.
   - Update the rest variable as the maximum between the previous sell value and the previous rest value.
3. Return the sell value, which represents the maximum profit at the end.

Time Complexity:
The time complexity is O(n), where n is the length of the prices array.

Space Complexity:
The space complexity is O(1) as we only need three variables to store the maximum profit.

Dynamic Programming:
- Subproblem: The subproblem is determining the maximum profit at each day considering the buy, sell, and rest states.
- Recurrence Relation: buy = max(buy, rest - price), sell = max(sell, buy + price), rest = max(sell, rest).
- Base Case: Initialize buy = -prices[0], sell = 0, and rest = 0.
*/

class Solution {
  public:
    int maxProfit(vector<int> &prices) {
        int n = prices.size();

        if (n <= 1) {
            return 0; // If there are no prices or only one price, no profit can be made
        }

        int buy = -prices[0]; // The maximum profit if the stock is bought at the current day
        int sell = 0; // The maximum profit if the stock is sold at the current day
        int rest = 0; // The maximum profit if no action is taken at the current day

        for (int i = 1; i < n; i++) {
```

```
            int prevBuy = buy; // Store the previous buy value for calculation
            int prevSell = sell; // Store the previous sell value for calculation
            int prevRest = rest; // Store the previous rest value for calculation
            // Calculate the maximum profit if the stock is bought at the current day
             // It is the maximum of the previous buy value (indicating that we didn't buy on the
current day)
            // and the profit of not buying on the previous day minus the current price
            buy = max(prevBuy, prevRest - prices[i]);
            // Calculate the maximum profit if the stock is sold at the current day
             // It is the maximum of the previous sell value (indicating that we didn't sell on the
current day)
            // and the profit of buying on the previous day plus the current price
            sell = max(prevSell, prevBuy + prices[i]);
            // Calculate the maximum profit if no action is taken at the current day
              // It is the maximum of the previous sell value (indicating that we didn't make any
transactions on the current day)
             // and the previous rest value (indicating that we carried over the same profit from
the previous day)
            rest = max(prevSell, prevRest);
        }

        return sell; // Return the maximum profit at the end of all days
    }
};

/*
Each day presents three options or states that can be chosen:
     1. Resting state (resting[i]): In this state, we choose not to take any action related to
buying or selling stocks. The maximum profit in this state is determined by the maximum profit
achieved in the previous day's resting state or selling state. Essentially, we carry over the
maximum profit from the previous day's resting or selling state.

     2. Buying state (buying[i]): In this state, we choose to buy stocks. The maximum profit in
this state is determined by the maximum profit achieved in the previous day's buying state or the
maximum profit achieved in the previous day's resting state minus the current day's stock price.
We select the higher value between these two options to maximize our profit.

    3. Selling state (selling[i]): In this state, we choose to sell stocks. The maximum profit in
this state is determined by adding the stock price of the current day to the maximum profit
achieved in the previous day's buying state. Selling stocks in this state allows us to realize the
profit accumulated from the previous buying state.
*/
/*
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();

        // Create three auxiliary arrays to store the maximum profit at each state
        vector<int> resting(n, 0);  // Maximum profit when in a resting state
        vector<int> buying(n, 0);   // Maximum profit when in a buying state
        vector<int> selling(n, 0);  // Maximum profit when in a selling state

        // Initialize the base cases for the first day
        resting[0] = 0;                 // No profit in resting state
         buying[0] = -prices[0];          // Maximum profit in buying state (buying on the first
day)
         selling[0] = INT_MIN;           // No profit in selling state (impossible to sell on the
first day)


        // In buying state, we can either buy or rest
          // In selling state, we either sell or rest but we can ignore rest to calculate what
profit will be if we sold everyday
```

```
          // In resting state, we carry over the maximum profit from the previous day's selling
state.

        // Iterate through the prices array to update the maximum profit at each state
        for (int i = 1; i < n; i++) {
            resting[i] = max(resting[i - 1], selling[i - 1]);

            buying[i] = max(buying[i - 1], resting[i - 1] - prices[i]);

            // Max profit if sold today
            selling[i] = buying[i - 1] + prices[i];
        }

          // The maximum profit at the end will be the maximum between the resting state and the
selling state
        return max(resting[n - 1], selling[n - 1]);
    }
};
*/
```

# 114. 14_2-D_Dynamic_Programming/04_Coin_Change_II/0518-coin-change-ii.cpp

```
/*
Problem: LeetCode 518 - Coin Change 2

Description:
You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.
Return the number of combinations that make up that amount.
You may assume that you have an infinite number of each kind of coin.
The answer is guaranteed to fit into a signed 32-bit integer.

Intuition:
To find the number of combinations that make up the amount, we can use dynamic programming.
At each step, we can decide whether to include a coin or not.
The key idea is to track the number of combinations for each amount up to the target amount.

Approach:
1. Initialize a 1D array dp of size (amount + 1) and set dp[0] = 1, representing the base case where there is one way to make up an amount of 0.
2. Iterate through the coins array:
   - For each coin, iterate through the amounts from coin to the target amount:
     - Update the dp[j] by adding the number of combinations for the amount j - coin.
3. Return dp[amount], which represents the number of combinations that make up the target amount.

Time Complexity:
The time complexity is O(coins * amount), where coins is the number of coins and amount is the target amount.

Space Complexity:
The space complexity is O(amount) as we only need a 1D array to store the number of combinations.

Dynamic Programming:
- Subproblem: The subproblem is finding the number of combinations for each amount up to the target amount.
- Recurrence Relation: dp[j] = dp[j] + dp[j - coin], where dp[j] represents the number of combinations for the amount j.
- Base Case: Initialize dp[0] = 1, representing one way to make up an amount of 0.
*/

class Solution {
  public:
    int change(int amount, vector<int> &coins) {
        // Create a DP array to store the number of combinations
        // dp[i] represents the number of combinations to make amount i
        vector<int> dp(amount + 1, 0);
        // Base case: There is one combination to make amount 0, using no coins
        dp[0] = 1;

        // Iterate over the coins
        for (int coin : coins) {
            // For each coin, iterate over the amounts starting from the coin value
            for (int i = coin; i <= amount; ++i) {
                  // For each amount, add the number of combinations from the current amount minus the coin value
                // This accounts for using the current coin to make the remaining amount
                   // We sum up these combinations for all the coins to get the total number of combinations for the current amount
                dp[i] += dp[i - coin];
            }
        }
```

```
        // Return the number of combinations to make the given amount using the coins
        return dp[amount];
    }
};

// class Solution {
// public:
//     int change(int amount, vector<int>& coins) {
//         // Create a 2D DP array to store the number of combinations
//           // dp[i][j] represents the number of combinations to make amount j using coins up to
the ith index
//         vector<vector<int>> dp(coins.size() + 1, vector<int>(amount + 1, 0));

//         // Base case: There is one combination to make amount 0, using no coins
//         dp[0][0] = 1;

//         // Iterate over the coins
//         for (int i = 1; i <= coins.size(); ++i) {
//             // For the first column (amount = 0), there is one combination for each coin, i.e.,
not selecting the coin
//             dp[i][0] = 1;

//             for (int j = 1; j <= amount; ++j) {
//                   // If the current coin value is less than or equal to the current amount, we
have two options:
//                    // 1. Use the current coin by subtracting its value from the amount (j) and
considering the number of combinations from the same row (coin)
//                   // 2. Skip the current coin and consider the number of combinations from the
previous row (coin - 1) for the same amount (j)
//                 // We sum up these two options to get the total number of combinations
//                 if (coins[i - 1] <= j) {
//                     dp[i][j] = dp[i][j - coins[i - 1]] + dp[i - 1][j];
//                 } else {
//                      // If the current coin value is greater than the current amount, we cannot
include it in the combination
//                       // So, we only consider the number of combinations from the previous row
for the same amount
//                     dp[i][j] = dp[i - 1][j];
//                 }
//             }
//         }

//         // Return the number of combinations to make the given amount using all the coins
//         return dp[coins.size()][amount];
//     }
// };
```

## 115. 14_2-D_Dynamic_Programming/05_Target_Sum/0494-target-sum.cpp

```
/*
Problem: LeetCode 494 - Target Sum

Description:
You are given an integer array nums and an integer target.
You want to build an expression out of nums by adding or subtracting each element in nums.
Return the number of different expressions that you can build, which evaluates to the target.

Intuition:
To find the number of different expressions that evaluate to the target, we can use dynamic programming.
The problem can be converted to a subset sum problem, where we need to find a subset of elements from the array with a specific sum.
To achieve this, we can transform the problem into a 0/1 knapsack problem.
At each step, we can decide whether to include or exclude an element from the sum.

Approach:
1. Calculate the sum of all elements in the nums array, as it helps to calculate the maximum possible positive sum.
2. Initialize a 2D array dp of size (nums.size() + 1) x (2 * sum + 1), and set dp[0][sum] = 1 to represent the base case where no elements are included, and the sum is 0.
3. Iterate through the nums array and update the dp array as follows:
    - For each element nums[i], update dp[i+1][j] by adding dp[i][j + nums[i]] and dp[i][j - nums[i]], representing including and excluding the element from the sum.
4. Return dp[nums.size()][sum + target], which represents the number of different expressions that evaluate to the target.

Time Complexity:
The time complexity is O(nums.size() * sum), where nums.size() is the number of elements in the array and sum is the sum of all elements.

Space Complexity:
The space complexity is O(nums.size() * sum) as we use a 2D array to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is finding the number of different expressions that evaluate to a specific sum using a subset of elements from the array.
- Recurrence Relation: dp[i+1][j] = dp[i][j + nums[i]] + dp[i][j - nums[i]], where dp[i+1][j] represents the number of different expressions that evaluate to sum j using the first i elements from the array.
- Base Case: Initialize dp[0][sum] = 1 to represent the base case where no elements are included, and the sum is 0.
*/

class Solution {
  public:
    int findTargetSumWays(vector<int> &nums, int target) {
        // Calculate the sum of all elements in the nums array
        int sum = accumulate(nums.begin(), nums.end(), 0);

         // If the target is greater than the sum or less than the negative sum, it's not possible to achieve the target
        if (target > sum || target < -sum) {
            return 0;
        }

        int n = nums.size();
        // Create a 2D dp array to store the number of ways to achieve each possible sum
         // dp[i][j] represents the number of ways to achieve sum j using the first i elements of
```

```
the nums array
        // We use 2 * sum + 1 as the column size to handle negative numbers
        vector<vector<int>> dp(n + 1, vector<int>(2 * sum + 1, 0));
        // Base case: There's one way to achieve a sum of 0 using an empty subset (no elements)
        dp[0][sum] = 1;

        // Loop through each element in the nums array
        for (int i = 0; i < n; i++) {
            // Loop through each possible sum in the dp array
            for (int j = nums[i]; j <= 2 * sum - nums[i]; j++) {
                   // If there's a way to achieve the current sum (dp[i][j]), update the ways to
achieve the sums j + nums[i] and j - nums[i]
                if (dp[i][j]) {
                    dp[i + 1][j + nums[i]] += dp[i][j]; // Add nums[i] to the sum
                    dp[i + 1][j - nums[i]] += dp[i][j]; // Subtract nums[i] from the sum
                }
            }
        }

        // Return the number of ways to achieve the target sum
         // We use dp[n][sum + target] because dp[n][sum] represents the number of ways to achieve
a sum of 0, and we need to adjust it to the target
        return dp[n][sum + target];
    }
};
```

## 116. 14_2-D_Dynamic_Programming/06_Interleaving_String/0097-interleaving-string.c

```
/*
Problem: LeetCode 97 - Interleaving String

Description:
Given strings s1, s2, and s3, find whether s3 is formed by the interleaving of s1 and s2.

Intuition:
To determine if s3 can be formed by interleaving s1 and s2, we can use dynamic programming.
The problem can be broken down into smaller subproblems, where we check if a prefix of s3 can be formed by interleaving prefixes of s1 and s2.

Approach:
1. Create a 2D dp array of size (s1.length()+1) x (s2.length()+1), where dp[i][j] represents whether s3[0...i+j-1] can be formed by interleaving s1[0...i-1] and s2[0...j-1].
2. Initialize dp[0][0] as true, representing the base case where both s1 and s2 are empty, and s3 is also empty.
3. Fill in the dp array using the following recurrence relation:
   - dp[i][j] is true if one of the following conditions is met:
     - dp[i-1][j] is true, and s1[i-1] is equal to s3[i+j-1].
     - dp[i][j-1] is true, and s2[j-1] is equal to s3[i+j-1].
4. Return dp[s1.length()][s2.length()], which represents whether s3 can be formed by interleaving s1 and s2.

Time Complexity:
The time complexity is O(s1.length() * s2.length()), as we fill in the entire dp array.

Space Complexity:
The space complexity is O(s1.length() * s2.length()), as we use a 2D array to store the intermediate results.

Dynamic Programming:
- Subproblem: The subproblem is checking if a prefix of s3 can be formed by interleaving prefixes of s1 and s2.
- Recurrence Relation: dp[i][j] is true if dp[i-1][j] is true and s1[i-1] is equal to s3[i+j-1], or if dp[i][j-1] is true and s2[j-1] is equal to s3[i+j-1].
- Base Case: Initialize dp[0][0] as true, representing the base case where both s1 and s2 are empty, and s3 is also empty.
*/

class Solution {
  public:
    bool isInterleave(string s1, string s2, string s3) {
        int m = s1.length(), n = s2.length();

         // If the lengths of s1 and s2 do not add up to the length of s3, it's not possible to form s3 by interleaving s1 and s2
        if (m + n != s3.length()) {
            return false;
        }

        // Create a 2D dp array to store the results of subproblems
         // dp[i][j] represents whether the first i characters of s1 and the first j characters of s2 can form the first i + j characters of s3
        vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
        // Base case: Both s1 and s2 are empty, and the result is true
        dp[0][0] = true;

        // Loop through each possible combination of i and j
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
```

```
                // Check if the previous characters of s1 and s3 match
                if (i > 0 && s1[i - 1] == s3[i + j - 1]) {
                    dp[i][j] = dp[i][j] || dp[i - 1][j];
                }

                // Check if the previous characters of s2 and s3 match
                if (j > 0 && s2[j - 1] == s3[i + j - 1]) {
                    dp[i][j] = dp[i][j] || dp[i][j - 1];
                }
            }
        }

            // Return the result, whether the last characters of s1 and s2 can form the last
characters of s3
        return dp[m][n];
    }
};
```

# 117. 14_2-D_Dynamic_Programming/07_Longest_Increasing_Path_In_a_Matrix/0329-

```
/*
Problem: LeetCode 329 - Longest Increasing Path in a Matrix

Description:
Given an m x n integers matrix, return the length of the longest increasing path in the matrix.
From each cell, you can either move in four directions: left, right, up, or down. You may not move diagonally or move outside the boundary (i.e., wrap-around is not allowed).

Intuition:
To find the longest increasing path in the matrix, we can use dynamic programming. The problem can be broken down into smaller subproblems, where we find the longest increasing path starting from each cell in the matrix.

Approach:
1. Create a 2D dp array of the same size as the matrix to store the length of the longest increasing path starting from each cell.
2. Initialize the dp array to all zeros, as the minimum path length from any cell is 1 (the cell itself).
3. For each cell (i, j) in the matrix, perform a depth-first search (DFS) to find the longest increasing path starting from that cell.
4. During the DFS, for each neighboring cell (x, y) of the current cell (i, j) that is within bounds and has a greater value than the current cell, calculate the length of the longest increasing path starting from (x, y) using the DFS, and update the dp array accordingly.
5. Return the maximum value in the dp array as the result, which represents the length of the longest increasing path in the matrix.

Time Complexity:
The time complexity is O(m * n), where m and n are the dimensions of the matrix, as we perform a DFS starting from each cell once.

Space Complexity:
The space complexity is also O(m * n) as we use a 2D dp array to store the length of the longest increasing path for each cell.

Dynamic Programming:
- Subproblem: The subproblem is finding the longest increasing path starting from each cell in the matrix.
- Recurrence Relation: For each neighboring cell (x, y) of the current cell (i, j) that is within bounds and has a greater value than the current cell, calculate the length of the longest increasing path starting from (x, y) using the DFS, and update the dp array accordingly.
- Base Case: Initialize the dp array to all zeros, as the minimum path length from any cell is 1 (the cell itself).
*/

class Solution {
  public:
    int longestIncreasingPath(vector<vector<int>> &matrix) {
        int m = matrix.size();
        int n = matrix[0].size();
          vector<vector<int>> dp(m, vector<int>(n, 0)); // dp[i][j] stores the longest increasing path starting at position (i, j)
        int longestPath = 0;

        // Iterate through each element in the matrix
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                  // Find the longest increasing path starting at position (i, j) and update the longestPath
                longestPath = max(longestPath, dfs(matrix, dp, i, j));
            }
```

```
        }

        return longestPath; // Return the overall longest increasing path
    }

    // Helper function to find the longest increasing path starting at position (i, j)
    int dfs(vector<vector<int>> &matrix, vector<vector<int>> &dp, int i, int j) {
        if (dp[i][j] > 0) {
              return dp[i][j];     // If the result is already calculated, return it from the dp
array
        }

        int m = matrix.size();
        int n = matrix[0].size();
        int longest = 1; // The minimum length is 1, considering the current element
        // Check the four neighbors
        vector<pair<int, int>> directions = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

        for (const auto &dir : directions) {
            int x = i + dir.first;
            int y = j + dir.second;

            if (x >= 0 && x < m && y >= 0 && y < n && matrix[x][y] > matrix[i][j]) {
                 // If the neighbor is within the matrix bounds and has a greater value, calculate
the longest path recursively
                longest = max(longest, 1 + dfs(matrix, dp, x, y));
            }
        }

        dp[i][j] = longest; // Save the result in the dp array to avoid redundant computations
        return longest;
    }
};
```

## 118. 14_2-D_Dynamic_Programming/08_Distinct_Subsequences/0115-distinct-subsec

```
/*
Problem: LeetCode 115 - Distinct Subsequences

Description:
Given two strings s and t, return the number of distinct subsequences of s which equals t.
A string's subsequence is a new string formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters.
It is guaranteed the answer fits on a 32-bit signed integer.

Intuition:
To find the number of distinct subsequences of s which equals t, we can use dynamic programming. The problem can be broken down into smaller subproblems, where we find the number of distinct subsequences for each prefix of s and t.

Approach:
1. Create a 2D dp array of size (m+1) x (n+1), where m is the length of string s and n is the length of string t. The dp[i][j] represents the number of distinct subsequences of the first i characters of string s and the first j characters of string t.
2. Initialize the first column dp[i][0] to 1, as there is one way to delete all characters from string s to get an empty string, which matches the empty string t.
3. For each (i, j) in the dp array, calculate the value using the recurrence relation:
    - If the current characters in s and t match (s[i-1] == t[j-1]), dp[i][j] = dp[i-1][j-1] + dp[i-1][j].
   - Otherwise, dp[i][j] = dp[i-1][j].
4. Return dp[m][n], which represents the number of distinct subsequences of s which equals t.

Time Complexity:
The time complexity is O(m * n), where m is the length of string s and n is the length of string t, as we fill the dp array of size (m+1) x (n+1).

Space Complexity:
The space complexity is O(m * n), where m is the length of string s and n is the length of string t, as we use a 2D dp array to store the number of distinct subsequences for each prefix of s and t.

Dynamic Programming:
- Subproblem: The subproblem is finding the number of distinct subsequences of the first i characters of string s and the first j characters of string t.
- Recurrence Relation: For each (i, j) in the dp array, calculate the value using the recurrence relation:
    - If the current characters in s and t match (s[i-1] == t[j-1]), dp[i][j] = dp[i-1][j-1] + dp[i-1][j].
   - Otherwise, dp[i][j] = dp[i-1][j].
- Base Case: Initialize the first column dp[i][0] to 1, as there is one way to delete all characters from string s to get an empty string, which matches the empty string t.
*/

class Solution {
  public:
    int numDistinct(string s, string t) {
        int m = s.length();
        int n = t.length();
        // Create a 2D DP array to store the number of distinct subsequences
        vector<vector<unsigned long long>> dp(m + 1, vector<unsigned long long>(n + 1, 0));

        // Base case: there is one way to delete all characters from s to get an empty string (t = "")
        for (int i = 0; i <= m; i++) {
            dp[i][0] = 1;
        }
```

```
        // Calculate the number of distinct subsequences
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // If the current characters match, we can either include or exclude s[i-1] in the
subsequence
                if (s[i - 1] == t[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
                } else {
                          // If the characters don't match, we can only exclude s[i-1] from the
subsequence
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        // The value at dp[m][n] represents the number of distinct subsequences of t in s
        return dp[m][n];
    }
};
```

## 119. 14_2-D_Dynamic_Programming/09_Edit_Distance/0072-edit-distance.cpp

```
/*
Problem: LeetCode 72 - Edit Distance

Description:
Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2.
You have the following three operations permitted on a word:
1. Insert a character
2. Delete a character
3. Replace a character

Intuition:
The problem can be solved using dynamic programming. We can break down the problem into smaller subproblems and find the minimum number of operations required to transform prefixes of the two strings into each other.

Approach:
1. Create a 2D DP table of size (m+1) x (n+1), where m and n are the lengths of word1 and word2, respectively.
     - dp[i][j] represents the minimum number of operations required to convert the first i characters of word1 to the first j characters of word2.
2. Initialize the first row and first column of the DP table as follows:
     - dp[i][0] represents the minimum number of operations required to convert the first i characters of word1 to an empty string (i deletions).
    - dp[0][j] represents the minimum number of operations required to convert an empty string to the first j characters of word2 (j insertions).
3. Fill in the DP table by considering the following cases:
    - If word1[i-1] is equal to word2[j-1], dp[i][j] will be the same as dp[i-1][j-1] since no operation is required.
   - If word1[i-1] is not equal to word2[j-1], dp[i][j] will be the minimum of the following three cases:
     a. dp[i-1][j] + 1 (deletion in word1)
     b. dp[i][j-1] + 1 (insertion in word1)
     c. dp[i-1][j-1] + 1 (replacement in word1)
4. The final answer will be stored in dp[m][n], representing the minimum number of operations required to convert word1 to word2.

Time Complexity:
The time complexity of the DP solution is O(m*n), where m and n are the lengths of word1 and word2, respectively. We need to fill in the entire DP table.

Space Complexity:
The space complexity of the DP solution is O(m*n), where m and n are the lengths of word1 and word2, respectively. We use a 2D DP table to store the minimum number of operations for each subproblem.

Dynamic Programming:
- Subproblem: The subproblem is finding the minimum number of operations required to convert prefixes of word1 to prefixes of word2.
- Recurrence Relation: The minimum number of operations is determined by three cases: insertion, deletion, or replacement.
- Base Case: The base cases are the first row and the first column of the DP table, representing conversions to and from empty strings.
*/

class Solution {
  public:
    int minDistance(string word1, string word2) {
        if (word1.empty() && word2.empty()) {
            return 0;
```

```
        }

        if (word1.empty() || word2.empty()) {
            return 1;
        }

        int m = word1.size();
        int n = word2.size();
        // Create a 2D DP table to store the minimum edit distance for subproblems
        vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

        // Initialize the first row and first column of the DP table
        // dp[i][0] represents the minimum edit distance between word1[0:i] and an empty string
        // dp[0][j] represents the minimum edit distance between an empty string and word2[0:j]
        for (int i = 0; i <= m; i++) {
            dp[i][0] = i;
        }

        for (int j = 0; j <= n; j++) {
            dp[0][j] = j;
        }

        // Fill in the DP table
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (word1[i - 1] == word2[j - 1]) {
                        // If the characters at the current positions are equal, no operation is
needed
                    // So, the minimum edit distance is the same as the previous subproblem
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                       // If the characters at the current positions are different, we have three
options:
                    // 1. Insert: dp[i][j - 1] represents the minimum edit distance if we insert a
character from word2
                    // 2. Delete: dp[i - 1][j] represents the minimum edit distance if we delete a
character from word1
                      // 3. Replace: dp[i - 1][j - 1] represents the minimum edit distance if we
replace a character in word1
                        // Take the minimum of these three options and add 1 to account for the
current operation
                    dp[i][j] = 1 + min({ dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1] });
                }
            }
        }

          // The final result is stored in dp[m][n], which represents the minimum edit distance
between the two words
        return dp[m][n];
    }
};

// class Solution {
// public:
//     int minDistance(string word1, string word2) {
//         int n = word1.size();
//         int m = word2.size();

//         // Create a 2D DP table to store the minimum edit distance for subproblems
//         int dp[n + 1][m + 1];

//         // Initialize the DP table
//         for (int i = 0; i <= n; i++) {
//             for (int j = 0; j <= m; j++) {
```

```
//                 if (i == 0 || j == 0) {
//                      // Base case: If one of the strings is empty, the minimum edit distance is
the length of the non-empty string
//                     dp[i][j] = max(i, j);
//                 } else if (word1[i - 1] == word2[j - 1]) {
//                       // If the characters at the current positions are equal, no operation is
needed
//                     dp[i][j] = dp[i - 1][j - 1];
//                 } else {
//                     // If the characters at the current positions are different
//                     dp[i][j] = 1 + min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1]));
//                 }
//             }
//         }

//          // The final result is stored in dp[n][m], which represents the minimum edit distance
between the two words
//         return dp[n][m];
//     }
// };
```

# 120. 14_2-D_Dynamic_Programming/10_Burst_Balloons/0312-burst-balloons.cpp

```
/*
Problem: LeetCode 312 - Burst Balloons

Description:
You are given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by an array nums.
You are asked to burst all the balloons.
If you burst the ith balloon, you will get nums[i - 1] * nums[i] * nums[i + 1] coins.
If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.
Return the maximum coins you can collect by bursting the balloons wisely.

Intuition:
The problem can be solved using dynamic programming. We can divide the problem into smaller subproblems and find the maximum coins we can get by bursting balloons in different ranges.

Approach:
1. Create a new array "numsWithBorders" by adding 1 at the beginning and end of the "nums" array. This helps in handling edge cases where i - 1 or i + 1 goes out of bounds.
2. Create a 2D DP table of size (n+2) x (n+2), where n is the length of "numsWithBorders".
   - dp[i][j] represents the maximum coins we can get by bursting balloons in the range [i, j].
3. Initialize the DP table diagonally (for single balloons):
   - dp[i][i] represents the maximum coins we can get from bursting the balloon at index i. Since there are no adjacent balloons, dp[i][i] will be numsWithBorders[i].
4. Fill in the DP table by considering the following cases:
   - For each subarray length "len" (from 2 to n), calculate the maximum coins for each possible range [i, j] of length "len".
    - For each range [i, j], try bursting each balloon "k" in the range and calculate the coins obtained. Update dp[i][j] to the maximum coins obtained from all "k" in the range.
5. The final answer will be stored in dp[0][n+1], representing the maximum coins we can get from bursting all balloons.

Time Complexity:
The time complexity of the DP solution is O(n^3), where n is the length of the "nums" array. We need to fill in the entire DP table, and for each subarray length "len", we consider all possible ranges of length "len".

Space Complexity:
The space complexity of the DP solution is O(n^2), where n is the length of the "nums" array. We use a 2D DP table to store the maximum coins for each subproblem.

Dynamic Programming:
- Subproblem: The subproblem is finding the maximum coins we can get by bursting balloons in different ranges.
- Recurrence Relation: The maximum coins for each range [i, j] can be calculated by trying to burst each balloon "k" in the range and updating dp[i][j] with the maximum coins obtained from all "k" in the range.
- Base Case: The base cases are the diagonals of the DP table, representing single balloons without adjacent balloons.
*/

//
https://leetcode.com/problems/burst-balloons/solutions/892552/for-those-who-are-not-able-to-understand-any-solution-with-diagram/

class Solution {
  public:
    int maxCoins(vector<int> &nums) {
        int n = nums.size();
        // Create a new array with borders containing 1 to simplify the logic
```

```
        vector<int> numsWithBorders(n + 2, 1);

        for (int i = 0; i < n; i++) {
            numsWithBorders[i + 1] = nums[i];
        }

        // Create a 2D DP table to store the maximum coins for each subarray
        vector<vector<int>> dp(n + 2, vector<int>(n + 2, 0));

        // Initialize DP table diagonally for single balloons
        for (int i = 1; i <= n; i++) {
            dp[i][i] = numsWithBorders[i];
        }

        // Fill in DP table for subarrays of length 2 and more
        for (int len = 2; len <= n + 1; len++) {
            for (int i = 0; i <= n + 1 - len; i++) {
                int j = i + len;

                for (int k = i + 1; k < j; k++) {
                    // Calculate the maximum coins for the subarray [i, j] by considering
                    // each possible balloon to be the last one to be bursted in the subarray
                            dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + numsWithBorders[i] *
numsWithBorders[k] * numsWithBorders[j]);
                }
            }
        }

         // The maximum coins that can be obtained from bursting all balloons is stored in dp[0][n
+ 1]
        return dp[0][n + 1];
    }
};

/*
// Beats 99% in Runtime and Memory

class Solution {
public:
    int maxCoins(vector<int>& nums) {
        int n = nums.size();
           int dp[305][305]; // DP table to store the maximum coins obtained for each range of
balloons

        for (int len = 1; len <= n; ++len) { // Iterate over all possible lengths of subarrays
            for (int start = 0; start + len <= n; ++start) { // Iterate over all possible starting
positions of subarrays
                int end = start + len - 1; // Calculate the end position of the current subarray

                  int leftNeighbor = (start == 0) ? 1 : nums[start - 1]; // Get the value of the
left neighbor (if it exists)
                 int rightNeighbor = (end + 1 == n) ? 1 : nums[end + 1]; // Get the value of the
right neighbor (if it exists)

                if (len == 1) {
                     // If the subarray contains only one balloon, the maximum coins obtained will
be just the value of that balloon
                    dp[start][end] = nums[start] * leftNeighbor * rightNeighbor;
                    continue;
                }

                  // Calculate the maximum coins obtained by bursting each balloon in the current
subarray
                        int maxCoinsStart = dp[start + 1][end] + nums[start] * leftNeighbor *
```

```
rightNeighbor;
                int maxCoinsEnd = dp[start][end - 1] + nums[end] * leftNeighbor * rightNeighbor;

                int maxCoinsMiddle = 0;
                for (int k = start + 1; k < end; ++k) {
                    // Calculate the maximum coins obtained by first bursting balloons in the left
subarray,
                        // then bursting balloons in the right subarray, and finally bursting the
balloon in the middle
                    int coinsLeft = dp[start][k - 1];
                    int coinsRight = dp[k + 1][end];
                    int coinsBurst = nums[k] * leftNeighbor * rightNeighbor;
                    maxCoinsMiddle = max(maxCoinsMiddle, coinsLeft + coinsRight + coinsBurst);
                }

                // Store the maximum coins obtained for the current subarray in the DP table
                dp[start][end] = max(maxCoinsStart, max(maxCoinsEnd, maxCoinsMiddle));
            }
        }
         return dp[0][n - 1]; // The result is stored in the top-right corner of the DP table for
the whole array
    }
};
*/
```

## 121. 14_2-D_Dynamic_Programming/11_Regular_Expression_Matching/0010-regular-

```
/*
Problem: LeetCode 10 - Regular Expression Matching

Description:
Given an input string (s) and a pattern (p), implement regular expression matching with support for '.' and '*' where:
- '.' Matches any single character.
- '*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Intuition:
This problem can be solved using dynamic programming. We can break down the problem into smaller subproblems and use a 2D DP table to store the results of these subproblems.

Approach:
1. Create a 2D DP table of size (s.length() + 1) x (p.length() + 1), where s.length() and p.length() are the lengths of the input string "s" and pattern "p", respectively.
   - dp[i][j] represents whether the substring s[0...i-1] matches the pattern p[0...j-1].
2. Initialize the DP table:
    - dp[0][0] represents whether an empty string matches an empty pattern, which is true. So, dp[0][0] = true.
    - dp[i][0] represents whether an empty string matches the pattern p[0...i-1]. For all i > 0, dp[i][0] is false because a non-empty pattern cannot match an empty string.
    - For dp[0][j], we need to handle patterns with '*', which means we should check if p[j-1] is '*' and if there's a valid pattern before it. If yes, then dp[0][j] will have the same result as dp[0][j-2].
3. Fill in the DP table by considering the following cases:
    - If p[j-1] is a regular character or '.', we need to check if s[i-1] matches p[j-1]. If yes, then dp[i][j] will be true if dp[i-1][j-1] is true.
   - If p[j-1] is '*', we have two sub-cases:
      a) If the character before '*' in the pattern (p[j-2]) matches the current character in the string (s[i-1]) or it is '.', then we have two options:
          - Ignore the '*' and the preceding character in the pattern: dp[i][j] will be true if dp[i][j-2] is true.
         - Consider the '*' and the preceding character in the pattern: dp[i][j] will be true if dp[i-1][j] is true and s[i-1] matches the preceding character in the pattern (p[j-2]).
4. The final answer will be stored in dp[s.length()][p.length()], representing whether the entire string "s" matches the entire pattern "p".

Time Complexity:
The time complexity of the DP solution is O(s.length() * p.length()), where s.length() and p.length() are the lengths of the input string "s" and pattern "p", respectively. We fill in the entire DP table of size (s.length() + 1) x (p.length() + 1).

Space Complexity:
The space complexity of the DP solution is O(s.length() * p.length()), where s.length() and p.length() are the lengths of the input string "s" and pattern "p", respectively. We use a 2D DP table to store the results of subproblems.

Dynamic Programming:
- Subproblem: The subproblem is whether a substring s[0...i-1] matches a pattern p[0...j-1].
- Recurrence Relation: The result for dp[i][j] depends on whether s[i-1] matches p[j-1] and the results of previous subproblems (dp[i-1][j-1], dp[i-1][j], dp[i][j-2]).
- Base Case: The base cases are dp[0][0], dp[i][0] (for all i > 0), and dp[0][j] (for patterns with '*').
*/

class Solution {
  public:
```

```
    bool isMatch(string s, string p) {
        int m = s.length();
        int n = p.length();
        // Create a 2D DP table and initialize base cases
        vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
        // Base case: an empty pattern matches an empty string
        dp[0][0] = true;

        // Initialize the first row for pattern p
        for (int j = 1; j <= n; j++) {
            if (p[j - 1] == '*') {
                  // If the current character in pattern is '*', it can match zero or more of the
preceding element
                 // So, we check if it matches two characters back in the pattern (j - 2) for an
empty string
                dp[0][j] = dp[0][j - 2];
            }
        }

        // Fill in the DP table row by row
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (p[j - 1] == s[i - 1] || p[j - 1] == '.') {
                       // If the current pattern character matches the current string character or
it's a '.',
                       // we take the value from the diagonal (dp[i - 1][j - 1]) as the result for
the current cell
                    dp[i][j] = dp[i - 1][j - 1];
                } else if (p[j - 1] == '*') {
                      // If the current pattern character is '*', it can match zero or more of the
preceding element
                       // So, we check if it matches two characters back in the pattern (j - 2) for
an empty string
                    // or if the current string character matches the preceding pattern character
                    // (dp[i - 1][j] && (s[i - 1] == p[j - 2] || p[j - 2] == '.'))
                     dp[i][j] = dp[i][j - 2] || (dp[i - 1][j] && (s[i - 1] == p[j - 2] || p[j - 2]
== '.'));
                }
            }
        }

         return dp[m][n]; // The last cell (dp[m][n]) contains the result for the entire matching
process
    }
};

/*
// Beats 100% Runtime and O(n) space complexity

class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.length();
        int n = p.length();

        // Create a 1D DP array to store the matching results
        vector<bool> dp(n + 1, false);

        // Base case: an empty pattern matches an empty string
        dp[0] = true;

        // Initialize the DP array for the first row (when i = 0)
        for (int j = 1; j <= n; j++) {
            if (p[j - 1] == '*') {
```

```
                  // If the current character is '*', it can match zero or more of the preceding
element
                // So, we check if it matches two characters back in the pattern (j - 2)
                dp[j] = dp[j - 2];
            }
        }

        // Fill in the DP array row by row
        for (int i = 1; i <= m; i++) {
             bool prevDiagonal = dp[0]; // Store the value of the diagonal element (dp[i - 1][j -
1]) before updating it
              dp[0] = false; // Update the base case for each row, as a non-empty pattern cannot
match an empty string

            // Iterate through the pattern characters
            for (int j = 1; j <= n; j++) {
                bool temp = dp[j]; // Store the current value of dp[i][j] before updating it

                if (p[j - 1] == s[i - 1] || p[j - 1] == '.') {
                       // If the current pattern character matches the current string character or
it's a '.',
                       // we take the value from the diagonal (dp[i - 1][j - 1]) as the result for
the current cell
                    dp[j] = prevDiagonal;
                } else if (p[j - 1] == '*') {
                     // If the current pattern character is '*', it can match zero or more of the
preceding element
                    // So, we check if it matches two characters back in the pattern (j - 2) or if
the current string
                     // character matches the preceding pattern character (dp[j] && (s[i - 1] ==
p[j - 2] || p[j - 2] == '.'))
                    dp[j] = dp[j - 2] || (dp[j] && (s[i - 1] == p[j - 2] || p[j - 2] == '.'));
                } else {
                      // If the characters don't match and there's no wildcard, update the cell to
false
                    dp[j] = false;
                }

                prevDiagonal = temp; // Update the diagonal element for the next iteration
            }
        }

          return dp[n]; // The last cell (dp[m][n]) contains the result for the entire matching
process
    }
};
*/
```

## 122. 15_Greedy/01_Maximum_Subarray/0053-maximum-subarray.cpp

```
/*
Problem: LeetCode 53 - Maximum Subarray

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

Intuition:
To find the maximum subarray sum, we can use the Kadane's algorithm.
The idea is to keep track of the maximum sum ending at each position in the array.
At each index i, we calculate the maximum subarray sum that ends at i by considering
two cases:
1. Include the current element in the subarray (by adding it to the sum ending at i-1).
2. Start a new subarray at the current element (by taking the current element itself).

Approach:
1. Initialize two variables, max_sum and current_sum, to track the maximum sum overall
   and the maximum sum ending at the current index, respectively.
2. Iterate through the array.
3. At each index i, update the current_sum by taking the maximum of nums[i] and
   nums[i] + current_sum. This step implements the Kadane's algorithm.
4. Update the max_sum by taking the maximum of max_sum and current_sum.
5. After iterating through the array, max_sum will represent the maximum subarray sum.

Time Complexity:
The time complexity is O(n), where n is the size of the input array. We only iterate
through the array once.

Space Complexity:
The space complexity is O(1), as we use only a constant amount of extra space.
*/

class Solution {
  public:
    int maxSubArray(vector<int> &nums) {
        int max_sum = nums[0]; // Initialize max_sum with the first element
        int current_sum = nums[0]; // Initialize current_sum with the first element

        for (int i = 1; i < nums.size(); i++) {
            // Calculate the maximum subarray sum ending at index i
            current_sum = max(nums[i], nums[i] + current_sum);
            // Update the overall maximum subarray sum
            max_sum = max(max_sum, current_sum);
        }

        return max_sum;
    }
};
```

## 123. 15_Greedy/02_Jump_Game/0055-jump-game.cpp

```
/*
Problem: LeetCode 55 - Jump Game

Description:
Given an array of non-negative integers nums, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Determine if you can reach the last index.

Intuition:
To check if it's possible to reach the last index, we can use a greedy approach.
The idea is to keep track of the furthest position we can reach from the current position.
If at any point, the furthest position we can reach is less than the current position,
we know that we cannot reach the end of the array.

Approach:
1. Initialize a variable, `max_reachable`, to store the furthest position we can reach from the current position.
2. Iterate through the array from the beginning.
3. At each index, update `max_reachable` to be the maximum of `max_reachable` and `i + nums[i]`.
   This represents the furthest position we can reach from the current index.
4. If at any point, `max_reachable` is less than or equal to the current index `i`,
   return false as we cannot reach the end of the array.
5. If we successfully reach the end of the loop, return true as we can reach the last index.

Time Complexity:
The time complexity is O(n), where n is the size of the input array.
We only iterate through the array once.

Space Complexity:
The space complexity is O(1), as we use only a constant amount of extra space for the `max_reachable` variable.
*/

class Solution {
  public:
    bool canJump(vector<int> &nums) {
        int max_reachable = 0; // Initialize the furthest position we can reach

        for (int i = 0; i < nums.size(); i++) {
               // Update max_reachable to represent the furthest position we can reach from the current index
            max_reachable = max(max_reachable, i + nums[i]);

              // If max_reachable is less than or equal to the current index, we cannot reach the end
            if (max_reachable <= i) {
                return false;
            }
        }

        // If we reach the end of the loop, we can reach the last index
        return true;
    }
};
```

## 124. 15_Greedy/03_Jump_Game_II/0045-jump-game-ii.cpp

```
/*
Problem: LeetCode 45 - Jump Game II

Description:
Given an array of non-negative integers nums, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.
If it is not possible to reach the last index, return -1.

Intuition:
To find the minimum number of jumps to reach the last index, we can use a greedy approach.
The idea is to keep track of the farthest position we can reach from the current position,
and the number of jumps needed to reach that position.
We update the current position to be the farthest position and increment the jumps count.

Approach:
1. Initialize `end` to represent the farthest position we can reach in the current jump.
2. Initialize `farthest` to represent the farthest position we can reach from any index in the current jump.
3. Initialize `jumps` to keep track of the number of jumps.
4. Iterate through the array from the beginning.
5. For each index, update `farthest` to be the maximum of `farthest` and `i + nums[i]`.
   This represents the farthest position we can reach from any index in the current jump.
6. If `i` reaches `end`, it means we have reached the end of the current jump.
   Update `end` to be `farthest`, and increment the `jumps` count.
7. If at any point, `end` is greater than or equal to the last index, return the `jumps` count.
8. If we successfully reach the end of the loop and have not yet reached the last index,
   it means we cannot reach the last index. Return -1.

Time Complexity:
The time complexity is O(n), where n is the size of the input array.
We only iterate through the array once.

Space Complexity:
The space complexity is O(1), as we use only a constant amount of extra space for the variables.
*/

class Solution {
  public:
    int jump(vector<int> &nums) {
        int end = 0;
        int farthest = 0;
        int jumps = 0;

        for (int i = 0; i < nums.size() - 1; i++) {
             // Update farthest to represent the farthest position we can reach from any index in the current jump
            farthest = max(farthest, i + nums[i]);

            // If i reaches end, it means we have reached the end of the current jump
            if (i == end) {
                // Update end to be the farthest position for the next jump
                end = farthest;
                // Increment the jumps count
                jumps++;

                // If end is greater than or equal to the last index, we have reached the end
                if (end >= nums.size() - 1) {
                    return jumps;
```

```
                }
            }
        }

        // If we reach the end of the loop and have not yet reached the last index, return -1
        return -1;
    }
};
```

## 125. 15_Greedy/04_Gas_Station/0134-gas-station.cpp

```
/*
Problem: LeetCode 134 - Gas Station

Description:
There are N gas stations along a circular route, where the amount of gas at station i is gas[i].
You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1).
You begin the journey with an empty tank at one of the gas stations.
Return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1.

Intuition:
To determine if there exists a valid starting gas station, we can follow a greedy approach.
If the total gas available is greater than or equal to the total cost of traveling the circuit,
then there must be a starting gas station that allows us to complete the circuit without running out of gas.

Approach:
1. Initialize variables `totalGas` and `totalCost` to store the total gas available and the total cost of traveling the circuit, respectively.
2. Initialize variables `currentGas` and `start` to store the current gas in the tank and the starting gas station index, respectively.
3. Iterate through the gas stations in a circular manner using a for loop.
4. For each gas station `i`, calculate the difference `diff` between the gas available `gas[i]` and the cost to travel to the next station `cost[i]`.
   Add `diff` to `currentGas` to simulate traveling to the next station.
5. If `currentGas` becomes negative at any station, it means we cannot reach the next station.
   In this case, reset `currentGas` to 0 and update `start` to be the next station index.
6. Keep adding `diff` to `totalGas` and `totalCost` as we traverse through the circular route.
7. At the end of the loop, check if `totalGas` is greater than or equal to `totalCost`.
   If true, return `start` as the starting gas station index.
8. If `totalGas` is less than `totalCost`, it means there is no valid starting gas station.
   Return -1.

Time Complexity:
The time complexity is O(n), where n is the number of gas stations. We only traverse through the gas stations once.

Space Complexity:
The space complexity is O(1), as we use only a constant amount of extra space for the variables.
*/

class Solution {
  public:
    int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
        int totalGas = 0; // Total gas available
        int totalCost = 0; // Total cost of traveling the circuit
        int currentGas = 0; // Current gas in the tank
        int start = 0; // Starting gas station index

        for (int i = 0; i < gas.size(); i++) {
            int diff = gas[i] - cost[i]; // Difference between gas available and cost to travel to the next station
            totalGas += gas[i];
            totalCost += cost[i];
            currentGas += diff;

               // If currentGas becomes negative, reset it to 0 and update start to be the next station index
            if (currentGas < 0) {
```

```
                currentGas = 0;
                start = i + 1;
            }
        }

         // If totalGas is greater than or equal to totalCost, there exists a valid starting gas
station
        if (totalGas >= totalCost) {
            return start;
        }

        // If totalGas is less than totalCost, there is no valid starting gas station
        return -1;
    }
};
```

## 126. 15_Greedy/05_Hand_of_Straights/0846-hand-of-straights.cpp

```
/*
Problem: LeetCode 846 - Hand of Straights

Description:
Alice has a hand of cards, given as an array of integers `hand`, where `hand[i]` is the value of the ith card.
A valid hand is a hand where every group of W cards is made up of cards of the same value.
Return true if and only if she can reorder the cards in her hand to form a valid hand.

Intuition:
To check if Alice can form valid groups of cards, we can use a greedy approach.
We can sort the cards in ascending order and then try to form groups of size W.

Approach:
1. Create a map to store the frequency of each card in the hand.
2. Sort the hand in ascending order.
3. Iterate through the sorted hand and for each card, try to form a group of size W.
4. If the current card frequency in the map is greater than 0, decrement its frequency by 1.
   Then, check if there are W-1 consecutive cards with frequencies greater than 0 after this card.
   If true, decrement the frequencies of these consecutive cards by 1 to form a group.
5. If we can't form a group of size W, return false.
6. If all groups are successfully formed, return true.

Time Complexity:
The time complexity is O(n log n) due to sorting the hand, where n is the number of cards in the hand.

Space Complexity:
The space complexity is O(n) to store the card frequencies in the map.
*/

class Solution {
  public:
    bool isNStraightHand(vector<int> &hand, int W) {
        if (hand.size() % W != 0) {
            return false; // If the hand size is not divisible by W, can't form valid groups
        }

        map<int, int> cardFreq; // Map to store the frequency of each card

        for (int card : hand) {
            cardFreq[card]++;
        }

        sort(hand.begin(), hand.end()); // Sort the hand in ascending order

        for (int card : hand) {
            if (cardFreq[card] > 0) {
                for (int i = 0; i < W; i++) {
                    if (cardFreq[card + i] == 0) {
                        return false; // Can't form a group of size W
                    }

                    cardFreq[card + i]--;
                }
            }
        }

        return true; // All groups of size W are formed successfully
    }
```

```
};

/*
// Beats 99% Runtime and Memory
class Solution {
public:
    bool isNStraightHand(vector<int>& hand, int groupSize) {
        int n = hand.size();
        // Check if the hand size is divisible by the groupSize
        if (n % groupSize != 0)
            return false;

        // Sort the hand array in ascending order
        sort(hand.begin(), hand.end());

        // Iterate through the hand array
        for (int i = 0; i < n; i++) {
            // Skip elements that have been marked as used (-1)
            if (hand[i] == -1)
                continue;

            int k = i;
            // Try to find the next groupSize consecutive elements
            for (int j = 1; j < groupSize; j++) {
                // Search for the next consecutive element by incrementing k
                while (k < n && hand[i] + j != hand[k])
                    k++;

                // If k reaches the end or the next element is not found, return false
                if (k == n)
                    return false;

                // Mark the found element as used by setting it to -1
                hand[k] = -1;
            }
        }

        // If all groups are found successfully, return true
        return true;
    }
};
*/
```

## 127. 15_Greedy/06_Merge_Triplets_to_Form_Target_Triplet/1899-merge-triplets-to-fo

```
/*
Problem: LeetCode 1899 - Merge Triplets to Form Target Triplet

Description:
A triplet is an array of three integers. You are given a 2D integer array `triplets`, where triplets[i] = [ai, bi, ci] describes the ith triplet.
You are also given an integer array `target`, where target = [x, y, z] represents the target triplet.
You want to form the target triplet by choosing three triplets from the triplets array (not necessarily distinct) and bitwise-ORing the elements of each chosen triplet.

Return true if it is possible to form the target triplet, otherwise, return false.

Intuition:
To form the target triplet [x, y, z], we must be able to select three triplets from the given triplets array such that bitwise-OR of their elements results in [x, y, z].

Approach:
1. Initialize three variables `x`, `y`, and `z` as 0.
2. Iterate through each triplet in the `triplets` array.
3. For each triplet, check if it can contribute to `x`, `y`, or `z`.
   - If the triplet's elements are greater than or equal to `x`, `y`, and `z` respectively, then the triplet can contribute to `x`, `y`, or `z`.
    - If it can contribute, update the corresponding `x`, `y`, or `z` value to the triplet's elements.
4. After iterating through all triplets, check if `x`, `y`, and `z` are equal to the target triplet elements.
5. If they are equal, return true, else return false.

Time Complexity:
The time complexity of this approach is O(n), where n is the number of triplets.

Space Complexity:
The space complexity is O(1) as we are using only a constant amount of extra space.
*/

class Solution {
  public:
    bool mergeTriplets(vector<vector<int>> &triplets, vector<int> &target) {
        int x = 0, y = 0, z = 0;

        for (auto &t : triplets) {
            if (t[0] <= target[0] && t[1] <= target[1] && t[2] <= target[2]) {
                x = max(x, t[0]);
                y = max(y, t[1]);
                z = max(z, t[2]);
            }
        }

        return (x == target[0] && y == target[1] && z == target[2]);
    }
};
```

## 128. 15_Greedy/07_Partition_Labels/0763-partition-labels.cpp

```
/*
Problem: LeetCode 763 - Partition Labels

Description:
Given a string s, partition s such that every substring in the partition is a palindrome.
Return a list of integers representing the length of each partition.

Intuition:
To partition the string into palindromic substrings, we need to find the last occurrence of each character in the string.
If we know the last index of each character in the string, we can use that information to determine the boundaries of each partition.
We can do this by iterating through the string and keeping track of the maximum index of each character encountered so far.

Approach:
1. Create a hash map to store the last index of each character in the string.
2. Initialize two variables `start` and `end` to 0, which will represent the current partition's start and end.
3. Initialize an empty vector `result` to store the lengths of each partition.
4. Iterate through the string.
5. For each character encountered, update its last index in the hash map.
6. If the current index equals the maximum index of the character found so far, it means we have reached the end of the current partition.
   - Add the length of the current partition to the `result` vector.
   - Update the `start` variable to the next index (i.e., the index immediately after the end of the current partition).
7. After iterating through the string, return the `result` vector.

Time Complexity:
The time complexity of this approach is O(n), where n is the length of the string.

Space Complexity:
The space complexity is O(1) as the size of the hash map is constant (26 characters).
*/

class Solution {
  public:
    vector<int> partitionLabels(string s) {
        unordered_map<char, int> lastOccurrence;
        vector<int> result;
        int start = 0, end = 0;

        // Store the last occurrence of each character in the hash map
        for (int i = 0; i < s.length(); ++i) {
            lastOccurrence[s[i]] = i;
        }

        // Iterate through the string and find the boundaries of each partition
        for (int i = 0; i < s.length(); ++i) {
            end = max(end, lastOccurrence[s[i]]);

            // If we have reached the end of the current partition
            if (i == end) {
                result.push_back(end - start + 1);
                start = i + 1;
            }
        }

        return result;
```

```
    }
};
```

## 129. 15_Greedy/08_Valid_Parenthesis_String/0678-valid-parenthesis-string.cpp

```
/*
Problem: LeetCode 678 - Valid Parenthesis String

Description:
Given a string s containing only three types of characters: '(', ')', and '*',
return true if s is valid.

The string is valid if the following conditions are met:
- Any left parenthesis '(' must have a corresponding right parenthesis ')'.
- Any right parenthesis ')' must have a corresponding left parenthesis '('.
- Left parenthesis '(' must go before the corresponding right parenthesis ')'.
- '*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string.

Intuition:
To check the validity of the given string, we can use a greedy approach along with two stacks.
We will keep two stacks, one to store the indices of the left parenthesis '(' and another to store the indices of the '*' encountered so far.
At any point, if we encounter a right parenthesis ')', we will first try to match it with the topmost left parenthesis in the left stack.
If the left stack is empty, we will try to match it with the topmost '*' in the '*' stack.
If both stacks are empty, we can't find a matching parenthesis, and the string is invalid.
Otherwise, we continue to pop the matched parenthesis and '*' from their respective stacks until we find a match or both stacks become empty.
If after processing the entire string, both stacks are empty, the string is valid.

Approach:
1. Initialize two stacks to store the indices of left parenthesis '(' and '*' encountered so far.
2. Iterate through the string.
3. If we encounter '(', push its index into the left stack.
4. If we encounter '*', push its index into the '*' stack.
5. If we encounter ')':
   - If the left stack is not empty, pop the topmost index from the left stack as it matches the current ')'.
   - Otherwise, if the '*' stack is not empty, pop the topmost index from the '*' stack as it can act as a right parenthesis.
   - If both stacks are empty, return false as we can't find a matching parenthesis.
6. After processing the entire string, we have some left parenthesis '(' and '*' without matches. Now, we need to match each remaining '(' with a '*' (if possible).
   To do this, we can pop pairs of '(' and '*' from their respective stacks until either of them becomes empty.
   If the left stack becomes empty first, it means we have matched all '(' with '*'. If the '*' stack becomes empty first, it means we have some '*' left without matches.
   We can ignore the '*' after matching all possible '(' as they can act as an empty string.
7. If both stacks become empty during this process, return true, indicating that the string is valid.
8. If any of the stacks still has elements, it means we can't find a match for some left parenthesis, and the string is invalid.

Time Complexity:
The time complexity is O(n), where n is the length of the string.

Space Complexity:
The space complexity is O(n), as in the worst case, both stacks can have all elements of the string.
*/

class Solution {
  public:
    bool checkValidString(string s) {
```

```
        stack<int> leftStack, starStack;

        for (int i = 0; i < s.length(); ++i) {
            if (s[i] == '(') {
                leftStack.push(i);
            } else if (s[i] == '*') {
                starStack.push(i);
            } else {
                if (!leftStack.empty()) {
                    leftStack.pop();
                } else if (!starStack.empty()) {
                    starStack.pop();
                } else {
                    return false;
                }
            }
        }

        while (!leftStack.empty() && !starStack.empty()) {
            if (leftStack.top() > starStack.top()) {
                return false;
            }

            leftStack.pop();
            starStack.pop();
        }

        return leftStack.empty();
    }
};

/*
class Solution {
public:
    bool checkValidString(string s) {
        int leftOpen = 0; // Keep track of possible open parentheses
        int leftMin = 0; // Minimum possible open parentheses
        for (char c : s) {
            if (c == '(') {
                leftOpen++;
                leftMin++;
            } else if (c == ')') {
                leftOpen--;
                leftMin = max(leftMin - 1, 0); // Ensure leftMin doesn't go negative
            } else { // c == '*'
                leftOpen++;
                leftMin = max(leftMin - 1, 0); // Ensure leftMin doesn't go negative
            }

            if (leftOpen < 0) {
                return false; // If there are too many closing parentheses, return false
            }
        }
        return leftMin == 0; // Check if all open parentheses can be matched
    }
};
*/
```

## 130. 16_Intervals/01_Insert_Interval/0057-insert-interval.cpp

```
/*
Problem: LeetCode 57 - Insert Interval

Description:
Given a set of non-overlapping intervals sorted by their start times,
insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Intuition:
The key idea is to find the correct position to insert the new interval and merge any overlapping
intervals if necessary.

Approach:
1. Create an empty result vector to store the merged intervals.
2. Iterate through the given intervals:
    a. If the current interval's end is less than the new interval's start, add the current
interval to the result vector.
   b. If the current interval's start is greater than the new interval's end, it means we have
found the correct position to insert the new interval.
      Add the new interval to the result vector and merge any remaining intervals if needed.
   c. If there is an overlap between the current interval and the new interval, update the new
interval's start and end to cover the overlapping region.
3. Add any remaining intervals to the result vector if needed.
4. Return the result vector.

Time Complexity:
The time complexity is O(n), where n is the number of intervals.

Space Complexity:
The space complexity is O(1) for the result vector, and no additional data structures are used, so
the overall space complexity is also O(1).
*/

class Solution {
  public:
    vector<vector<int>> insert(vector<vector<int>> &intervals, vector<int> &newInterval) {
        vector<vector<int>> result;
        int i = 0;
        int n = intervals.size();

        // Add intervals that end before the new interval starts
        while (i < n && intervals[i][1] < newInterval[0]) {
            result.push_back(intervals[i]);
            i++;
        }

        // Merge overlapping intervals
        while (i < n && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = min(newInterval[0], intervals[i][0]);
            newInterval[1] = max(newInterval[1], intervals[i][1]);
            i++;
        }

        // Add the merged interval
        result.push_back(newInterval);

        // Add remaining intervals
        while (i < n) {
            result.push_back(intervals[i]);
```

```
            i++;
        }

        return result;
    }
};
```

## 131. 16_Intervals/02_Merge_Intervals/0056-merge-intervals.cpp

```
/*
Problem: LeetCode 56 - Merge Intervals

Description:
Given a collection of intervals, merge all overlapping intervals.

Intuition:
To merge overlapping intervals, we can sort the intervals based on their start times. Then, we can iterate through the sorted intervals and merge any overlapping intervals as we encounter them.

Approach:
1. Sort the given intervals based on their start times.
2. Create an empty result vector to store the merged intervals.
3. Iterate through the sorted intervals:
   a. If the result vector is empty or the current interval's start is greater than the end of the last merged interval in the result vector, add the current interval to the result vector.
    b. If there is an overlap between the current interval and the last merged interval in the result vector, update the end of the last merged interval to the maximum of both intervals' ends.
4. Return the result vector.

Time Complexity:
The time complexity is O(n log n), where n is the number of intervals. Sorting the intervals takes O(n log n) time, and merging them takes O(n) time.

Space Complexity:
The space complexity is O(1) for the result vector, and no additional data structures are used, so the overall space complexity is also O(1).
*/

class Solution {
  public:
    vector<vector<int>> merge(vector<vector<int>> &intervals) {
        if (intervals.empty()) return {};

        // Sort the intervals based on their start times
         // Uses a lambda function as the third argument of the sort function to define the custom sorting criteria.
        sort(intervals.begin(), intervals.end(), [](const vector<int> &a, const vector<int> &b) {
            return a[0] < b[0];
        });
        vector<vector<int>> result;
        result.push_back(intervals[0]);

        for (int i = 1; i < intervals.size(); i++) {
            vector<int> &lastInterval = result.back();

            if (intervals[i][0] > lastInterval[1]) {
                // No overlap, add the current interval to the result
                result.push_back(intervals[i]);
            } else {
                // Overlap, update the end of the last merged interval
                lastInterval[1] = max(lastInterval[1], intervals[i][1]);
            }
        }

        return result;
    }
};
```

## 132. 16_Intervals/03_Non_Overlapping_Intervals/0435-non-overlapping-intervals.cpp

```
/*
Problem: LeetCode 435 - Non-overlapping Intervals

Description:
Given an array of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Intuition:
To find the minimum number of intervals to remove, we can use a greedy approach. If two intervals overlap, we should keep the one with the smaller end time, as this will leave more space for other intervals. If we keep intervals with smaller end times, we have a better chance of fitting more intervals in the remaining space.

Approach:
1. Sort the given intervals based on their end times in ascending order.
2. Initialize a variable "end" to store the end time of the last non-overlapping interval.
3. Initialize a variable "count" to keep track of the number of intervals to remove (initialize to 0).
4. Iterate through the sorted intervals:
   a. If the current interval's start time is greater than or equal to "end", it means it doesn't overlap with the last non-overlapping interval, so we update "end" to be the end time of the current interval.
   b. If the current interval's start time is less than "end", it means there is an overlap, and we need to remove one of the intervals. We increment the "count" variable and continue to the next interval.
5. Return the value of "count", which represents the minimum number of intervals to remove.

Time Complexity:
The time complexity is O(n log n), where n is the number of intervals. Sorting the intervals based on their end times takes O(n log n) time.

Space Complexity:
The space complexity is O(1) as we only use a constant amount of additional space.
*/

class Solution {
  public:
    int eraseOverlapIntervals(vector<vector<int>> &intervals) {
        if (intervals.empty()) {
            return 0;
        }

        // Sort the intervals based on their end times in ascending order
        sort(intervals.begin(), intervals.end(), [](const vector<int> &a, const vector<int> &b) {
            return a[1] < b[1];
        });
        int end = intervals[0][1]; // End time of the last non-overlapping interval
        int count = 0; // Number of intervals to remove

        for (int i = 1; i < intervals.size(); i++) {
            // If the current interval's start time is less than "end", there is an overlap
            if (intervals[i][0] < end) {
                count++;
            } else {
                // No overlap, update "end" to be the end time of the current interval
                end = intervals[i][1];
            }
        }

        return count;
```

```
    }
};
```

## 133. 16_Intervals/04_Meeting_Rooms/0252-meeting-rooms.cpp

```
/*
Problem: LeetCode 252 - Meeting Rooms

Description:
Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

Intuition:
To determine if a person could attend all meetings, we need to check if there are any overlapping intervals. If any two intervals overlap, the person cannot attend all meetings.

Approach:
1. Sort the given intervals based on their start times in ascending order.
2. Iterate through the sorted intervals:
   a. For each interval (except the first one), check if its start time is less than the end time of the previous interval. If so, it means there is an overlap, and the person cannot attend all meetings. Return false.
3. If there are no overlaps, return true.

Time Complexity:
The time complexity is O(n log n), where n is the number of intervals. Sorting the intervals based on their start times takes O(n log n) time.

Space Complexity:
The space complexity is O(1) as we only use a constant amount of additional space.
*/

class Solution {
  public:
    bool canAttendMeetings(vector<vector<int>> &intervals) {
        if (intervals.empty()) {
            return true;
        }

        // Sort the intervals based on their start times in ascending order
        sort(intervals.begin(), intervals.end(), [](const vector<int> &a, const vector<int> &b) {
            return a[0] < b[0];
        });

        for (int i = 1; i < intervals.size(); i++) {
            // Check if there is an overlap
            if (intervals[i][0] < intervals[i - 1][1]) {
                return false;
            }
        }

        return true;
    }
};
```

## 134. 16_Intervals/05_Meeting_Rooms_II/0253-meeting-rooms-ii.cpp

```
/*
Problem: LeetCode 253 - Meeting Rooms II

Description:
Given an array of meeting time intervals where intervals[i] = [starti, endi], find the minimum number of conference rooms required.

Intuition:
To find the minimum number of conference rooms required, we can simulate the process of scheduling the meetings in a room.

Approach:
1. Sort the given intervals based on their start times in ascending order.
2. Use a priority queue to keep track of the end times of the meetings currently scheduled in different rooms.
3. Iterate through the sorted intervals:
   a. If the priority queue is empty or the start time of the current interval is greater than the end time of the earliest meeting in the queue, it means we can schedule this meeting in one of the existing rooms. Pop the earliest meeting from the queue and push the end time of the current interval.
    b. If the start time of the current interval is less than or equal to the end time of the earliest meeting in the queue, it means we need to allocate a new room for this meeting. Push the end time of the current interval into the queue.
4. The size of the priority queue at any time represents the number of conference rooms needed to accommodate the meetings.

Time Complexity:
The time complexity is O(n log n), where n is the number of intervals. Sorting the intervals based on their start times takes O(n log n) time, and pushing and popping elements from the priority queue takes O(log n) time for each meeting.

Space Complexity:
The space complexity is O(n) as we use a priority queue to store the end times of the meetings.
*/

class Solution {
  public:
    int minMeetingRooms(vector<vector<int>> &intervals) {
        if (intervals.empty()) {
            return 0;
        }

        // Sort the intervals based on their start times in ascending order
        sort(intervals.begin(), intervals.end(), [](const vector<int> &a, const vector<int> &b) {
            return a[0] < b[0];
        });
        // Use a priority queue to keep track of end times of the meetings in different rooms
        priority_queue<int, vector<int>, greater<int>> pq;
        pq.push(intervals[0][1]); // Push the end time of the first meeting

        for (int i = 1; i < intervals.size(); i++) {
            int start = intervals[i][0];
            int end = intervals[i][1];

            // Check if the current meeting can be accommodated in an existing room
            if (start >= pq.top()) {
                pq.pop();
            }

            // Push the end time of the current meeting into the queue
```

```
            pq.push(end);
        }

        // The size of the priority queue represents the minimum number of conference rooms needed
        return pq.size();
    }
};
```

## 135. 16_Intervals/06_Minimum_Interval_to_Include_Each_Query/1851-minimum-interv

```
/*
Problem: LeetCode 1851 - Minimum Interval to Include Each Query

Description:
You are given a 2D integer array intervals, where intervals[i] = [lefti, righti] describes the ith interval starting from lefti and ending at righti (inclusive). The size of the array is n and 1-indexed.

You are also given an integer array queries, where queries[j] = [leftj, rightj] describes the jth query starting from leftj and ending at rightj (inclusive). The size of the array is m and 1-indexed.

Find the minimum interval for each query such that it is completely covered by an interval in intervals and lci <= leftj <= rightj <= rci.

Return an integer array answer, where answer[j] is the minimum interval length for the jth query.

Intuition:
To solve this problem efficiently, we can use a two-step approach:
1. Sort both intervals and queries in ascending order based on their lengths.
2. Use a priority queue (min heap) to keep track of the right boundaries of the current intervals.

Approach:
1. Create two separate vectors of pairs: one for intervals and one for queries, where the first element of each pair represents the length of the interval/query, and the second element is the index of the interval/query in the original arrays.
2. Sort both vectors in ascending order based on their lengths.
3. Initialize a priority queue (min heap) to keep track of the right boundaries of the current intervals. We'll insert the intervals in the queue based on their right boundary.
4. Iterate through the sorted intervals:
    a. While the current query length is smaller or equal to the current interval length, update the result for the query and remove it from the priority queue.
5. Continue to process the next interval.
6. Return the answer vector containing the minimum interval lengths for each query.

Time Complexity:
The time complexity is O(n log n + m log m), where n is the number of intervals and m is the number of queries. Sorting both vectors of pairs takes O(n log n + m log m) time, and processing intervals and queries takes O(n + m log n) time.

Space Complexity:
The space complexity is O(n + m) to store the vectors of pairs and the priority queue.
*/

class Solution {
  public:
    vector<int> minInterval(vector<vector<int>> &intervals, vector<int> &queries) {
        vector<int> sortedQueries = queries;
        // [size of interval, end of interval]
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
        // {query -> size of interval}
        unordered_map<int, int> m;
        // Also need only valid intervals, so sort by start time & sort queries
        sort(intervals.begin(), intervals.end());
        sort(sortedQueries.begin(), sortedQueries.end());
        vector<int> result;
        int i = 0;

        for (int j = 0; j < sortedQueries.size(); j++) {
            int query = sortedQueries[j];
```

```
            // Push intervals into the min heap whose start time is less than or equal to the
current query value
            while (i < intervals.size() && intervals[i][0] <= query) {
                int left = intervals[i][0];
                int right = intervals[i][1];
                pq.push({right - left + 1, right});
                i++;
            }

             // Pop the invalid intervals from the min heap (intervals whose end time is less than
the current query value)
            while (!pq.empty() && pq.top().second < query) {
                pq.pop();
            }

            // Store the minimum interval size for the current query in the unordered_map
            if (!pq.empty()) {
                m[query] = pq.top().first;
            } else {
                m[query] = -1;
            }
        }

        // Build the result vector using the unordered_map for each query
        for (int j = 0; j < queries.size(); j++) {
            result.push_back(m[queries[j]]);
        }

        return result;
    }
};
```

## 136. 17_Math_&_Geometry/01_Rotate_Image/0048-rotate-image.cpp

```
/*
Problem: LeetCode 48 - Rotate Image

Description:
You are given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise).

Intuition:
To rotate the image by 90 degrees clockwise, we can perform two steps:
1. Transpose the matrix (rows become columns, and vice versa).
2. Reverse each row of the transposed matrix.

Approach:
1. Transpose the matrix in-place:
   - Iterate over the upper triangle of the matrix (i.e., elements above the main diagonal).
   - Swap the element at matrix[i][j] with matrix[j][i].
2. Reverse each row of the transposed matrix:
   - For each row, use two pointers (start and end) and swap the elements until they meet in the middle.

Time Complexity:
The time complexity of this approach is O(n^2), where n is the number of rows (or columns) in the matrix.

Space Complexity:
The space complexity is O(1) as we are performing the rotation in-place without using any additional space.
*/

class Solution {
  public:
    void rotate(vector<vector<int>> &matrix) {
        int n = matrix.size();

        // Step 1: Transpose the matrix
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                swap(matrix[i][j], matrix[j][i]);
            }
        }

        // Step 2: Reverse each row of the transposed matrix
        for (int i = 0; i < n; i++) {
            int start = 0;
            int end = n - 1;

            while (start < end) {
                swap(matrix[i][start], matrix[i][end]);
                start++;
                end--;
            }
        }
    }
};
```

## 137. 17_Math_&_Geometry/02_Spiral_Matrix/0054-spiral-matrix.cpp

```
/*
Problem: LeetCode 54 - Spiral Matrix

Description:
Given an m x n matrix, return all elements of the matrix in spiral order.

Intuition:
We can traverse the matrix in a spiral order by simulating the movement in four directions: right, down, left, and up.
To keep track of the visited elements, we can maintain four boundaries: top, bottom, left, and right.

Approach:
1. Initialize four variables: top = 0, bottom = m - 1, left = 0, right = n - 1.
2. Loop until top <= bottom and left <= right:
   - Traverse from left to right along the top boundary and increment top.
   - Traverse from top to bottom along the right boundary and decrement right.
   - Traverse from right to left along the bottom boundary and decrement bottom.
   - Traverse from bottom to top along the left boundary and increment left.
3. Keep adding the elements encountered during the traversal to the result vector.

Time Complexity:
The time complexity of this approach is O(m * n), where m is the number of rows and n is the number of columns in the matrix.

Space Complexity:
The space complexity is O(1) as we are not using any additional space.
*/

class Solution {
  public:
    vector<int> spiralOrder(vector<vector<int>> &matrix) {
        vector<int> result;

        if (matrix.empty()) {
            return result;
        }

        int m = matrix.size();
        int n = matrix[0].size();
        int top = 0, bottom = m - 1, left = 0, right = n - 1;

        while (top <= bottom && left <= right) {
            // Traverse from left to right along the top boundary
            for (int i = left; i <= right; i++) {
                result.push_back(matrix[top][i]);
            }

            top++;

            // Traverse from top to bottom along the right boundary
            for (int i = top; i <= bottom; i++) {
                result.push_back(matrix[i][right]);
            }

            right--;

            // Traverse from right to left along the bottom boundary
            if (top <= bottom) {
                for (int i = right; i >= left; i--) {
```

```
                result.push_back(matrix[bottom][i]);
            }

            bottom--;
        }

        // Traverse from bottom to top along the left boundary
        if (left <= right) {
            for (int i = bottom; i >= top; i--) {
                result.push_back(matrix[i][left]);
            }

            left++;
        }
    }

    return result;
  }
};
```

## 138. 17_Math_&_Geometry/03_Set_Matrix_Zeroes/0073-set-matrix-zeroes.cpp

```
/*
Problem: LeetCode 73 - Set Matrix Zeroes

Description:
Given an m x n matrix, if an element is 0, set its entire row and column to 0. Do it in-place.

Intuition:
To solve this problem in-place, we can use the first row and the first column of the matrix to keep track of which rows and columns need to be set to 0. We use two additional boolean variables to track whether the first row and first column themselves need to be set to 0.

Approach:
1. First, we check if the first row and the first column need to be set to 0 by iterating through the first row and first column of the matrix.
2. Then, we traverse the matrix starting from the second row and second column.
   - If the current element matrix[i][j] is 0, we set the corresponding elements in the first row and first column to 0.
3. Next, we traverse the matrix again from the second row and second column and set the elements to 0 if the corresponding element in the first row or first column is 0.
4. Finally, we handle the first row and first column based on the boolean variables we used to track them.

Time Complexity:
The time complexity of this approach is O(m * n), where m is the number of rows and n is the number of columns in the matrix. We traverse the matrix twice.

Space Complexity:
The space complexity is O(1) as we are using the first row and first column of the matrix to keep track of which rows and columns need to be set to 0.
*/

class Solution {
  public:
    void setZeroes(vector<vector<int>> &matrix) {
        int m = matrix.size();
        int n = matrix[0].size();
        bool firstRowZero = false;
        bool firstColZero = false;

        // Check if first row needs to be set to 0
        for (int j = 0; j < n; j++) {
            if (matrix[0][j] == 0) {
                firstRowZero = true;
                break;
            }
        }

        // Check if first column needs to be set to 0
        for (int i = 0; i < m; i++) {
            if (matrix[i][0] == 0) {
                firstColZero = true;
                break;
            }
        }

        // Mark rows and columns that need to be set to 0 in the first row and first column
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][j] == 0) {
                    matrix[i][0] = 0;
```

```
                matrix[0][j] = 0;
            }
        }
    }

    // Set elements to 0 based on the first row and first column
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                matrix[i][j] = 0;
            }
        }
    }

    // Set first row and first column to 0 if needed
    if (firstRowZero) {
        for (int j = 0; j < n; j++) {
            matrix[0][j] = 0;
        }
    }

    if (firstColZero) {
        for (int i = 0; i < m; i++) {
            matrix[i][0] = 0;
        }
    }
  }
};
```

## 139. 17_Math_&_Geometry/04_Happy_Number/0202-happy-number.cpp

```
/*
Problem: LeetCode 202 - Happy Number

Description:
A happy number is a number defined by the following process:
- Starting with any positive integer, replace the number by the sum of the squares of its digits.
- Repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1.
- Those numbers for which this process ends in 1 are happy.

Intuition:
To determine if a number is a happy number, we can use a set to keep track of all the numbers we have encountered during the process. If we encounter a number that we have seen before, it means there is a cycle, and the number is not a happy number.

Approach:
1. We start with the given number and calculate the sum of the squares of its digits.
2. We continue this process until the sum becomes 1 or until we encounter a number that we have seen before.
3. If the sum becomes 1, we return true, indicating that the number is a happy number.
4. If we encounter a number we have seen before, we return false, indicating that the number is not a happy number.

Time Complexity:
The time complexity of this approach is difficult to determine as it depends on the number of iterations required to reach 1 or find a cycle. In practice, the process usually converges quickly for happy numbers, so the time complexity is considered to be approximately O(log n).

Space Complexity:
The space complexity is O(log n) as we use a set to keep track of the numbers encountered during the process, and the number of unique numbers encountered is limited.
*/

class Solution {
  public:
    bool isHappy(int n) {
        unordered_set<int> seen;

        while (n != 1 && seen.find(n) == seen.end()) {
            seen.insert(n);
            n = getNextNumber(n);
        }

        return n == 1;
    }

  private:
    int getNextNumber(int n) {
        int sum = 0;

        while (n > 0) {
            int digit = n % 10;
            sum += digit * digit;
            n /= 10;
        }

        return sum;
    }
};
```

## 140. 17_Math_&_Geometry/05_Plus_One/0066-plus-one.cpp

```
/*
Problem: LeetCode 66 - Plus One

Description:
Given a non-empty array of decimal digits representing a non-negative integer, increment one to the integer.
The digits are stored such that the most significant digit is at the head of the list, and each element in the array contains a single digit.
You may assume the integer does not contain any leading zero, except for the number 0 itself.

Intuition:
To increment a number represented as an array of digits, we need to add one to the last digit and handle any carry that may occur.

Approach:
1. Start from the end of the digits array.
2. Add one to the last digit.
3. If the digit becomes 10 (carry occurs), set it to 0 and move to the next digit.
4. Continue this process until there is no more carry or we reach the beginning of the digits array.
5. If there is still a carry after processing all digits, it means the original number was all nines, and we need to add a new digit at the beginning of the array with a value of 1.

Time Complexity:
The time complexity is O(n), where n is the number of digits in the array. In the worst case, we may need to carry the addition all the way to the beginning of the array.

Space Complexity:
The space complexity is O(1) as we are modifying the input array in place and not using any additional data structures.
*/

class Solution {
  public:
    vector<int> plusOne(vector<int> &digits) {
        int n = digits.size();
        // Start from the end and add one to the last digit
        digits[n - 1] += 1;

        // Handle any carry
        for (int i = n - 1; i > 0; i--) {
            if (digits[i] == 10) {
                digits[i] = 0;
                digits[i - 1] += 1;
            } else {
                break; // No more carry, exit the loop
            }
        }

        // If there is still a carry, add a new digit at the beginning of the array
        if (digits[0] == 10) {
            digits[0] = 0;
            digits.insert(digits.begin(), 1);
        }

        return digits;
    }
};

/*
```

```
class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        for(int i=digits.size()-1; i>=0; i--){
            digits[i]++;
            if(digits[i]==10){
                digits[i]=0;
            }
            else{
                break;
            }
        }
        if(digits[0]==0){
            digits.insert(digits.begin(), 1);
        }
        return digits;
    }
};
*/
```

## 141. 17_Math_&_Geometry/06_Pow(x,n)/0050-powx-n.cpp

```
/*
Problem: LeetCode 50 - Pow(x, n)

Description:
Implement pow(x, n), which calculates x raised to the power n (i.e., x^n).

Intuition:
To calculate the power x^n, we can use the concept of recursion and divide-and-conquer. The idea is to break down the problem into smaller subproblems and solve them recursively.

Approach:
1. First, we handle the base case where n is 0 or 1. If n is 0, the result is 1. If n is 1, the result is x itself.
2. If n is negative, we convert the problem to calculate 1/x raised to the power -n.
3. To calculate x^n, we can divide the problem into two parts:
   - Calculate x^(n/2) using recursion.
   - If n is even, the result is (x^(n/2)) * (x^(n/2)).
   - If n is odd, the result is (x^(n/2)) * (x^(n/2)) * x.
4. We use a helper function to perform the recursive calculations.

Time Complexity:
The time complexity of this approach is O(log n) because we are dividing the problem size by 2 in each recursive call.

Space Complexity:
The space complexity is O(log n) as well due to the recursive function calls.
*/

class Solution {
  public:
    double myPow(double x, long long n) {
        // Base case: x^0 = 1
        if (n == 0) {
            return 1.0;
        }

        // Base case: x^1 = x
        if (n == 1) {
            return x;
        }

        // If n is negative, calculate 1/x^-n
        if (n < 0) {
            return 1.0 / myPow(x, -n);
        }

        // Calculate x^n using recursion and divide-and-conquer
        double halfPow = myPow(x, n / 2);
        double result = halfPow * halfPow;

        // If n is odd, multiply with x
        if (n % 2 != 0) {
            result *= x;
        }

        return result;
    }
};

/*
```

```
// Beats 100% runtime

class Solution {
public:
    double myPow(double x, int n) {
        // Handle negative exponent by inverting the base
        if (n < 0) {
            x = 1 / x;
        }

        // Take the absolute value of the exponent to avoid issues with INT_MIN
        long num = labs(n);

        // Initialize the result to 1
        double pow = 1;

        // Exponentiation by squaring algorithm
        while (num) { // While the exponent is not zero
            if (num & 1) { // If the least significant bit of the exponent is 1
                pow *= x; // Multiply the result by the current base value
            }

            x *= x; // Square the base value for the next iteration
            num >>= 1; // Right-shift the exponent to remove the least significant bit
        }

        return pow;
    }
};
*/
```

## 142. 17_Math_&_Geometry/07_Multiply_Strings/0043-multiply-strings.cpp

```
/*
Problem: LeetCode 43 - Multiply Strings

Description:
Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2, also represented as a string.

Intuition:
The basic idea behind multiplying two numbers represented as strings is to simulate the process of multiplication as we do on paper. We start with the least significant digits of both numbers and multiply them. We keep track of the carry and add the product to the corresponding position in the result string. We continue this process for all digits of both numbers, considering the correct position of the result digits based on the multiplication.

Approach:
1. Create a result string to store the final product.
2. Initialize an array to store the intermediate products (i.e., the products of each digit in num1 with each digit in num2).
3. Traverse num1 and num2 from right to left, and calculate all the intermediate products, storing them in the array.
4. Calculate the carry and the sum at each position of the result string, taking into account the intermediate products and any previous carry.
5. After completing the traversal and calculations, the result string will hold the final product.

Time Complexity:
The time complexity of this approach is O(m * n), where m and n are the lengths of num1 and num2, respectively.

Space Complexity:
The space complexity is O(m + n) to store the intermediate products.
*/

class Solution {
  public:
    string multiply(string num1, string num2) {
        int m = num1.size();
        int n = num2.size();
        vector<int> products(m + n, 0); // To store intermediate products
        string result = "";

        // Calculate intermediate products
        for (int i = m - 1; i >= 0; i--) {
            for (int j = n - 1; j >= 0; j--) {
                int mul = (num1[i] - '0') * (num2[j] - '0');
                int sum = mul + products[i + j + 1]; // Add to the existing intermediate product
                products[i + j] += sum / 10; // Carry
                products[i + j + 1] = sum % 10; // Store the digit at correct position
            }
        }

        // Build the result string
        for (int p : products) {
            if (!(result.empty() && p == 0)) {
                result += to_string(p);
            }
        }

        return result.empty() ? "0" : result;
    }
};
```

## 143. 17_Math_&_Geometry/08_Detect_Squares/2013-detect-squares.cpp

```
/*
Problem: LeetCode 2013 - Detect Squares

Intuition:
Precompute the count of points in each row and column and then use this information to find the other three points of the square.

Approach:
1. We use two unordered maps "rowMap" and "colMap" to keep track of the count of points in each row and column, respectively.
2. When we add a new point (x, y), we increment its count in both "rowMap" and "colMap" accordingly.
3. To count the number of squares with a given point (x, y) as the bottom-right corner, we look for other points in the same row "x."
4. For each point found at column "col" in row "x," we calculate the edge length of the potential square. Let's call it "edgeLen," which is the absolute difference between "col" and "y."
5. Then, we check if the points (x - edgeLen, col), (x + edgeLen, col), and (x + edgeLen, y) exist in the maps. If they do, we have found a square with (x, y) as the bottom-right corner.
6. We repeat this process for all points in row "x" to find all squares with (x, y) as the bottom-right corner.
7. Finally, we sum up the counts of all these squares to get the total number of squares with (x, y) as the bottom-right corner.
8. The process is efficient as we use unordered maps to store the counts, making the lookups quick and allowing us to count squares in constant time.

Time Complexity:
The time complexity of this approach is O(1) for `add` and O(n) for `count` (where "n" is the average number of points in the same row as the given point).

Space Complexity:
The space complexity is O(N) (where "N" is the total number of points added).
*/

class DetectSquares {
  public:
    // We maintain two maps for row and column coordinates to store the count of points in each row and column.
    vector<unordered_map<int, int>> rowMap;
    vector<unordered_map<int, int>> colMap;

    DetectSquares() : rowMap(1001), colMap(1001) {
        // Constructor to initialize the rowMap and colMap with size 1001.
    }

    // Function to add a point to the grid.
    void add(vector<int> point) {
        // Increment the count of points in the corresponding row and column maps.
        rowMap[point[0]][point[1]]++;
        colMap[point[1]][point[0]]++;
    }

     // Function to count the number of squares that can be formed with a given point as the bottom-right corner.
    int count(vector<int> point) {
        int result = 0;

        // Iterate through the row map for the given point's row.
        for (auto [col, cCount] : rowMap[point[0]]) {
            int edgeLen = abs(col - point[1]);
```

```
            if (edgeLen == 0) {
                continue;    // Skip points that are at the same position as the given point.
            }

             // Calculate the row coordinates for the other two points of the square based on the
edge length.
            int row = point[0] - edgeLen;

            if (colMap[col].find(row) != colMap[col].end()) {
                auto rCount = colMap[col][row];

                // Check if the other two points exist in the colMap.
                if (colMap[point[1]].find(row) != colMap[point[1]].end()) {
                    // If so, update the result by multiplying the counts of the four points.
                    result += (rCount * cCount * colMap[point[1]][row]);
                }
            }

            row = point[0] + edgeLen;

            if (colMap[col].find(row) != colMap[col].end()) {
                auto rCount = colMap[col][row];

                // Check if the other two points exist in the colMap.
                if (colMap[point[1]].find(row) != colMap[point[1]].end()) {
                    // If so, update the result by multiplying the counts of the four points.
                    result += (rCount * cCount * colMap[point[1]][row]);
                }
            }
        }

        return result;
    }
};
```

## 144. 18_Bit_Manipulation/01_Single_Number/0136-single-number.cpp

```
/*
Problem: LeetCode 136 - Single Number

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

Intuition:
Since all elements appear twice in the array except for one, we can use the XOR operation to find the single element. The XOR operation on two equal numbers results in 0, so XORing all the numbers in the array will give us the single number that appears only once.

Approach:
1. Initialize a variable 'result' to 0.
2. Iterate through the elements in the array 'nums'.
3. For each element, XOR it with the 'result'.
4. The final value of 'result' will be the single number that appears only once.

Time Complexity:
The time complexity is O(n), where n is the number of elements in the array 'nums'. We iterate through the array once to perform the XOR operation.

Space Complexity:
The space complexity is O(1) since we only use a constant amount of extra space for the 'result' variable.
*/

class Solution {
  public:
    int singleNumber(vector<int> &nums) {
        int result = 0;

        // Perform XOR operation on all elements in the array
        for (int num : nums) {
            result ^= num;
        }

        return result;
    }
};
```

## 145. 18_Bit_Manipulation/02_Number_of_1_Bits/0191-number-of-1-bits.cpp

```
/*
Problem: LeetCode 191 - Number of 1 Bits

Description:
Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

Intuition:
To count the number of '1' bits in the binary representation of a number, we can use the bit manipulation technique. The idea is to repeatedly shift the number to the right and check if the least significant bit is '1'. If it is, then we increment a count variable.

Approach:
1. Initialize a variable 'count' to 0.
2. Iterate while the input number 'n' is not equal to 0.
3. Inside the loop, check if the least significant bit of 'n' is '1'.
4. If it is, increment the 'count' variable.
5. Right-shift 'n' by 1 to move on to the next bit.
6. Continue the loop until all bits are processed.
7. Return the 'count' variable, which contains the number of '1' bits in the input number.

Time Complexity:
The time complexity is O(log n), where n is the value of the input number. The number of iterations in the loop is equal to the number of bits in the binary representation of 'n', which is log(n) base 2.

Space Complexity:
The space complexity is O(1) since we only use a constant amount of extra space for the 'count' variable.
*/

class Solution {
  public:
    int hammingWeight(uint32_t n) {
        int count = 0;

        // Iterate while n is not 0
        while (n != 0) {
            // Check if the least significant bit is 1
            if (n & 1) {
                count++;
            }

            // Right-shift n to move to the next bit
            n >>= 1;
        }

        return count;
    }
};

// class Solution {
// public:
//     int hammingWeight(uint32_t n) {
//         int count = 0;
//         while (n != 0) {
//             count++;
//             n &= (n - 1); // Clear the least significant 1 bit
//         }
//         return count;
```

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

## 146. 18_Bit_Manipulation/03_Counting_Bits/0338-counting-bits.cpp

```
/*
Problem: LeetCode 338 - Counting Bits

Description:
Given a non-negative integer num, for every number i in the range 0 ? i ? num,
calculate the number of 1's in their binary representation and return them as an array.

Intuition:
To count the number of '1' bits in the binary representation of each number, we can use dynamic programming.
The idea is to utilize the previously calculated results to build the result for the current number.

Approach:
1. Create a vector 'result' of size num+1 to store the counts for all numbers from 0 to num.
2. Initialize the first element of 'result' to 0 since the number 0 has 0 '1' bits.
3. Use a for loop to iterate from 1 to num.
4. For each number i, the number of '1' bits can be calculated using the expression result[i] = result[i >> 1] + (i & 1).
   - The expression (i >> 1) calculates the number obtained by right-shifting i by one bit, which is equivalent to dividing i by 2.
   - The expression (i & 1) checks if the least significant bit of i is '1'.
   - The count of '1' bits for i is the count of '1' bits for (i >> 1) plus the count of the least significant bit (i & 1).
5. Continue the loop until all numbers from 0 to num are processed.
6. Return the 'result' vector containing the counts for all numbers.

Time Complexity:
The time complexity is O(n), where n is the value of the input number 'num'.
The loop iterates 'num' times to calculate the count of '1' bits for each number.

Space Complexity:
The space complexity is O(n) as we use a vector of size 'num+1' to store the counts for all numbers from 0 to num.
*/

class Solution {
  public:
    vector<int> countBits(int num) {
        vector<int> result(num + 1, 0);

        for (int i = 1; i <= num; ++i) {
            result[i] = result[i >> 1] + (i & 1);
        }

        return result;
    }
};

// class Solution {
// public:
//     vector<int> countBits(int num) {
//         vector<int> result(num + 1, 0);

//         for (int i = 1; i <= num; ++i) {
//             result[i] = result[i & (i - 1)] + 1;
//         }

//         return result;
//     }
```

```
// };
```

## 147. 18_Bit_Manipulation/04_Reverse_Bits/0190-reverse-bits.cpp

```
/*
Problem: LeetCode 190 - Reverse Bits

Description:
Reverse bits of a given 32 bits unsigned integer.

Intuition:
To reverse the bits of the given integer, we can use bit manipulation techniques.

Approach:
1. Initialize a variable 'result' to 0, which will store the reversed bits of the input number.
2. Use a for loop to iterate 32 times (since it is a 32-bit integer).
3. For each iteration, shift the 'result' to the left by 1 bit to make space for the next bit.
4. Extract the least significant bit from the input number using the expression (n & 1).
5. Add the extracted bit to the 'result' using the bitwise OR operation (result |= (n & 1)).
6. Right shift the input number by 1 bit to get rid of the least significant bit (n >>= 1).
7. Continue the loop until all 32 bits are processed.
8. After the loop, the 'result' will contain the reversed bits of the input number.
9. Return the 'result'.

Time Complexity:
The time complexity is O(1) since the number of iterations is fixed (32 iterations for a 32-bit
integer).

Space Complexity:
The space complexity is O(1) as we only use a constant amount of space to store the 'result' and
other variables.
*/

#include <cstdint>

class Solution {
  public:
    uint32_t reverseBits(uint32_t n) {
        uint32_t result = 0;

        for (int i = 0; i < 32; ++i) {
            result <<= 1;
            result |= (n & 1);
            n >>= 1;
        }

        return result;
    }
};
```

## 148. 18_Bit_Manipulation/05_Missing_Number/0268-missing-number.cpp

```
/*
Problem: LeetCode 268 - Missing Number

Description:
Given an array nums containing n distinct numbers in the range [0, n],
return the only number in the range that is missing from the array.

Intuition:
We can use the mathematical concept of finding the sum of the first n natural numbers
and subtract the sum of the elements in the given array to find the missing number.

Approach:
1. Initialize a variable 'expectedSum' to store the sum of the first n natural numbers.
2. Iterate through the array 'nums' and calculate the sum of its elements, storing it in the variable 'actualSum'.
3. Calculate the missing number as 'expectedSum - actualSum' and return it.

Time Complexity:
The time complexity is O(n) as we need to iterate through the entire array once to calculate the sum of its elements.

Space Complexity:
The space complexity is O(1) as we use only a constant amount of extra space to store variables.
*/

class Solution {
  public:
    int missingNumber(std::vector<int> &nums) {
        int n = nums.size();
        int expectedSum = n * (n + 1) / 2;
        int actualSum = 0;

        for (int num : nums) {
            actualSum += num;
        }

        return expectedSum - actualSum;
    }
};
```

## 149. 18_Bit_Manipulation/06_Sum_of_Two_Integers/0371-sum-of-two-integers.cpp

```
/*
Problem: LeetCode 371 - Sum of Two Integers

Description:
Given two integers a and b, return the sum of the two integers without using the '+' and '-' operators.

Intuition:
We can use bitwise operations to perform addition without using the '+' operator.
The bitwise XOR operation (^) will give us the sum of two integers without considering the carry.
To handle the carry, we can use the bitwise AND operation (&) and left shift (<<) to calculate the carry.

Approach:
1. While 'b' is not equal to 0 (there is still a carry):
   a. Calculate the carry as 'carry = a & b'.
   b. Update 'a' as 'a = a ^ b' to get the sum without carry.
   c. Update 'b' as 'b = carry << 1' to prepare for the next iteration.

2. Return 'a' as the final sum.

Time Complexity:
The time complexity is O(1) because we perform a constant number of bitwise operations.

Space Complexity:
The space complexity is O(1) as we use only a constant amount of extra space.
*/

class Solution {
  public:
    int getSum(int a, int b) {
        while (b != 0) {
            int carry = (unsigned int)(a & b) << 1; // Use unsigned int to handle negative numbers
            a = a ^ b;
            b = carry;
        }

        return a;
    }
};
```

## 150. 18_Bit_Manipulation/07_Reverse_Integer/0007-reverse-integer.cpp

```
/*
Problem: LeetCode 7 - Reverse Integer

Description:
Given a signed 32-bit integer x, return x with its digits reversed.
If reversing x causes the value to go outside the signed 32-bit integer range [-2^31, 2^31 - 1],
then return 0.

Intuition:
To reverse an integer, we can repeatedly extract the last digit of the number and add it to the
result after multiplying the result by 10.

Approach:
1. Initialize a variable 'result' to store the reversed number and set it to 0.
2. While the input 'x' is not equal to 0:
   a. Extract the last digit of 'x' using 'x % 10'.
   b. Check if 'result' is going to overflow:
        - If 'result' is greater than INT_MAX/10 or 'result' is equal to INT_MAX/10 and the last
digit is greater than 7, then return 0.
   c. Multiply 'result' by 10 and add the last digit to it.
   d. Update 'x' by removing the last digit using 'x /= 10'.
3. Return the 'result' as the reversed number.

Time Complexity:
The time complexity of this approach is O(log(x)), where x is the given input number.

Space Complexity:
The space complexity is O(1) as we are using only a constant amount of extra space.
*/

class Solution {
  public:
    int reverse(int x) {
        int result = 0;

        while (x != 0) {
            int lastDigit = x % 10;

            if (result > INT_MAX / 10 || (result == INT_MAX / 10 && lastDigit > 7)) {
                return 0;
            }

            if (result < INT_MIN / 10 || (result == INT_MIN / 10 && lastDigit < -8)) {
                return 0;
            }

            result = result * 10 + lastDigit;
            x /= 10;
        }

        return result;
    }
};
```