2.Medium\01.2_sum_problem.cpp

```cpp
/*
QUESITON:-
Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.
You may assume that each input would have exactly one solution, and you may not use the same element twice.
You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
Example 2:

Input: nums = [3,2,4], target = 6
Output: [1,2]
*/

/*
Approach:

-> Create an empty map to store the elements and their corresponding indices.
>  Iterate through the input array, nums, and for each element:
    Calculate the complement by subtracting the current element from the target value.
    Check if the complement exists in the map.
    If the complement exists, return the indices of the current element and the complement.
    If the complement does not exist, add the current element and its index to the map.
-> If no solution is found, return an empty vector or a message indicating no solution exists.
*/

// CODE:-
vector<int> twoSum(vector<int> &nums, int target)
{
    unordered_map<int, int> mp;
    for (int i = 0; i < nums.size(); i++)
    {
        int remain = target - nums[i];
        if (mp.find(remain) != mp.end() && mp[remain] != i)
            return {i, mp[remain]};
        mp[nums[i]] = i;
    }
    return {-1, -1};
    // If the question asks to just return whether pair exists or not, not the indexes in that case we can sort and easily find the pair sum without extra space
}

// TIME COMPLEXITY = O(N)
// SPACE COMPLEXITY = O(N)
```

2.Medium\02.Sort_0_1_2.cpp

```cpp
/*
QUESTION:-
Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.
We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.
You must solve this problem without using the library's sort function.

Example 1:

Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]

Example 2:

Input: nums = [2,0,1]
Output: [0,1,2]
*/

/*
APPROACH:-
-> Initialize three pointers: low at the beginning of the array, mid at the beginning of the array, and high at the end of the array.
-> Iterate through the array while the mid pointer is less than or equal to the high pointer:
1. If the current element at the mid pointer is 0 (red), we swap it with the element at the low pointer and increment both low and mid pointers. This ensures that red elements are moved to the left side of the array.
2. If the current element at the mid pointer is 1 (white), we simply increment the mid pointer. This keeps white elements in the middle of the array.
3. If the current element at the mid pointer is 2 (blue), we swap it with the element at the high pointer and decrement the high pointer. This ensures that blue elements are moved to the right side of the array.

Repeat step 2 until the mid pointer crosses the high pointer.
At the end of the algorithm, the array will be sorted in the desired order.
*/

// CODE:-
void sortColors(vector<int> &nums)
{
    int low = 0, mid = 0, high = nums.size() - 1;
    while (mid <= high)
    {
        if (nums[mid] == 0)
            swap(nums[mid++], nums[low++]);
        else if (nums[mid] == 1)
            mid++;
        else
            swap(nums[mid], nums[high--]);
    }
}

```

```
// TIME COMPLEXITY = O(N)
// SPACE COMPLEXITY = O(0)
```

2.Medium\03.Majority_element.cpp

```cpp
/*
QUESTION:-
Given an array nums of size n, return the majority element.
The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Example 1:

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

Example 2:

Input: nums = [2,2,1,1,1,2,2]
Output: 2
*/

/*
APROACH:-
-> Initialize two variables: candidate and count. Set candidate to the first element of the array, and count to 1.
-> Iterate through the array starting from the second element:
    If the current element is equal to the candidate, increment the count by 1.
    If the current element is different from the candidate, decrement the count by 1.
    If the count becomes 0, update the candidate to the current element and set the count to 1 again.
-> After the iteration, the candidate variable will hold the majority element.
Return the candidate as the result.
*/

// CODE:-
int majorityElement(vector<int> &nums)
{
    int candidate = nums[0];
    int vote = 1;
    for (int i = 1; i < nums.size(); i++)
    {
        if (vote <= 0)
            candidate = nums[i];
        if (nums[i] == candidate)
            vote++;
        else
            vote--;
    }
    return candidate;
}

// TIME COMPLEXITY = O(N)
// SPACE COMPLEXITY = O(0)
```

2.Medium\04.Kadane's_algorithm.cpp

```cpp
/*
QUESTION:-
Given an integer array nums, find the subarray with the largest sum, and return its sum.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:

Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
*/

/*
APPROACH:-
-> Initialize two variables: maxSum and currentSum. Set both variables to the first element of the array.
-> Iterate through the array starting from the second element:
    Update currentSum by adding the current element to it.
    If currentSum becomes negative, reset it to 0. This step ensures that we consider only the subarrays with positive sums.
    Update maxSum by taking the maximum value between maxSum and currentSum. This keeps track of the maximum subarray sum encountered so far.
-> After the iteration, the maxSum variable will hold the largest sum of any subarray.
-> Return the maxSum as the result.
*/

// CODE:-
int maxSubArray(vector<int> &nums)
{
    int curr_sum = 0;
    int ans = INT_MIN;
    for (int i = 0; i < nums.size(); i++)
    {
        curr_sum += nums[i];
        ans = max(ans, curr_sum);
        if (curr_sum < 0)
            curr_sum = 0;
    }
    return ans;
}

// TIME COMPLEXITY = O(N)
// SPACE COMPLEXITY = O(0)
```

2.Medium\05.Number_of_subarray_sum_equal_k.cpp

```cpp
/*
QUESTION:
Given an array of integers nums and an integer k, return the total number of subarrays whose sum equals to k.

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

APPROACH:
To find the total number of subarrays with sum equal to k, we can use the technique of prefix sum along with a hashmap.
1. Initialize a variable `count` to keep track of the count of subarrays with sum equal to k.
2. Initialize a variable `prefixSum` to keep track of the prefix sum while iterating through the array.
3. Initialize a hashmap `sumCount` to store the frequency of prefix sums encountered so far.
4. Set the initial prefix sum to 0 and set its count to 1 in the `sumCount` hashmap.
5. Iterate through the array and update the prefix sum by adding each element.
6. Check if the current prefix sum minus k exists in the `sumCount` hashmap. If it does, add the count of that prefix sum to the `count` variable.
7. Increment the count of the current prefix sum in the `sumCount` hashmap.
8. Finally, return the `count` variable as the total number of subarrays with sum equal to k.

CODE:
*/

int subarraySum(vector<int> &nums, int k)
{
    int pref_sum = 0;
    unordered_map<int, int> mp;
    int ans = 0;

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

        if (pref_sum == k)
            ans++;

        if (mp.find(pref_sum - k) != mp.end())
        {
            ans += mp[pref_sum - k];
        }

        mp[pref_sum]++;
    }

    return ans;
}

/*
TIME COMPLEXITY: O(n), where n is the size of the input array nums.
```

```
SPACE COMPLEXITY: O(n), as we are using a hashmap to store the prefix sums and their corresponding counts.
*/

```

2.Medium\06.Stock_buy_sell.cpp

```cpp
/*
QUESTION:-
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.
Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Example 1:

Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.
*/

/*
APPROACH:-
Initialize two variables: min_price and max_profit.

-> min_price = minimum price in the array.
-> max_profit = 0.

Iterate through the array, and for each price:

-> Update min_price to the minimum price seen so far.
-> Update max_profit to the maximum profit seen so far, or the current price minus min_price, whichever is greater.

Return max_profit.
*/

// CODE:-
int maxProfit(vector<int> &prices)
{
    int minprice = prices[0];
    int ans = 0;
    for (int i = 1; i < prices.size(); i++)
    {
        ans = max(ans, prices[i] - minprice);
        minprice = min(minprice, prices[i]);
    }
    return ans;
}

```

2.Medium\07.Rearange_elements_by_sign.cpp

```cpp
/*
QUESTION:-
You are given a 0-indexed integer array nums of even length consisting of an equal number of positive and negative integers.
You should rearrange the elements of nums such that the modified array follows the given conditions:
Every consecutive pair of integers have opposite signs.
For all integers with the same sign, the order in which they were present in nums is preserved.
The rearranged array begins with a positive integer.
Return the modified array after rearranging the elements to satisfy the aforementioned conditions.


Example 1:

Input: nums = [3,1,-2,-5,2,-4]
Output: [3,-2,1,-5,2,-4]
Explanation:
The positive integers in nums are [3,1,2]. The negative integers are [-2,-5,-4].
The only possible way to rearrange them such that they satisfy all conditions is [3,-2,1,-5,2,-4].
Other ways such as [1,-2,2,-5,3,-4], [3,1,2,-2,-5,-4], [-2,3,-5,1,-4,2] are incorrect because they do not satisfy one or more conditions.
*/

/*
APPROACH:-
Initialize two pointers, pos_ptr and neg_ptr. pos_ptr will point to the first positive integer in the array, and neg_ptr will point to the first negative integer in the array.
Iterate over the array.
If the current integer is positive, swap it with the element at neg_ptr.
Increment pos_ptr by 1.
Increment neg_ptr by 1.
Repeat steps 3-5 until the end of the array is reached.
The array will now be rearranged such that every consecutive pair of integers have opposite signs.
*/

// CODE:-
vector<int> rearrangeArray(vector<int> &nums)
{
    int i = 0; // for +ve integers
    int j = 1; // for -ve integers
    vector<int> ans(nums.size());
    for (int k = 0; k < nums.size(); k++)
    {
        if (nums[k] >= 0)
        {
            ans[i] = nums[k];
            i += 2;
        }
        else
        {
            ans[j] = nums[k];
            j += 2;
        }
    }
    return ans;
}

// TIME COMPLEXITY = O(N)
// SPACE COMPLEXITY = O(0)
```

2.Medium\08.Next_permutation.cpp

```cpp
/*
QUESTION:-

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].
The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

For example, the next permutation of arr = [1,2,3] is [1,3,2].
Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, find the next permutation of nums.

The replacement must be in place and use only constant extra memory.

Example 1:
Input: nums = [1,2,3]
Output: [1,3,2]
*/

/*
APPROACH:-

To find the next permutation of an array, we can follow these steps:

1. Find the first index `i` from the right such that `nums[i] < nums[i+1]`. This is the first element that needs to be swapped.
2. Find the first index `j` from the right such that `nums[j] > nums[i]`. This is the element that will replace `nums[i]`.
3. Swap `nums[i]` and `nums[j]`.
4. Reverse the subarray starting from `i+1` till the end of the array.
5. If step 1 does not find any index `i`, it means the array is in descending order. In that case, reverse the entire array to get the lowest possible order.

*/

// CODE:

void nextPermutation(vector<int> &nums)
{

    int bp = -1;
    // finding the break point
    for (int i = nums.size() - 2; i >= 0; i--)
    {
        if (nums[i] < nums[i + 1])
        {
            bp = i;
            break;
        }
    }
    // first greater element from back
    if (bp != -1)
    {
        for (int i = nums.size() - 1; i >= 0; i--)
        {
            if (nums[i] > nums[bp])
            {
                swap(nums[i], nums[bp]);
                break;
            }
        }
    }
    // reverse the array from bp+1 to end
    reverse(nums.begin() + bp + 1, nums.end());
}

// TIME COMPLEXITY: O(n), where n is the size of the input array.
// SPACE COMPLEXITY: O(1)

```

2.Medium\09.Leaders_in_array.cpp

```cpp
/*
QUESTION:-

Given an array A of positive integers. Your task is to find the leaders in the array. An element of the array is a leader if it is greater than or equal to all the elements to its right side. The rightmost element is always a leader.

Example 1:
Input:
n = 6
A[] = {16,17,4,3,5,2}
Output: 17 5 2
Explanation: The first leader is 17 as it is greater than all the elements to its right. Similarly, the next leader is 5. The rightmost element is always a leader, so it is also included.

*/

/*
APPROACH:-

To find the leaders in the array, we can follow these steps:

1. Initialize a variable `maxRight` with the rightmost element of the array.
2. Iterate the array from right to left:
   - If the current element is greater than or equal to `maxRight`, it is a leader. Print the current element and update `maxRight` to the current element.
3. Finally, print `maxRight` as it is always a leader.

*/

// CODE:

vector<int> leaders(int a[], int n)
{
    vector<int> ans;
    ans.push_back(a[n - 1]);
    int maxi = a[n - 1]; // represent maximum encountered till now

    for (int i = n - 2; i >= 0; i--)
    {
        if (a[i] >= maxi)
        {
            ans.push_back(a[i]);
            maxi = a[i];
        }
    }

    reverse(ans.begin(), ans.end());
    return ans;
}

```

2.Medium\10.Longest_consecutive_subsequence.cpp

```cpp
/*
QUESTION:-

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

Example 1:
Input: nums = [100,4,200,1,3,2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

Example 2:
Input: nums = [0,3,7,2,5,8,4,6,0,1]
Output: 9
*/

/*
APPROACH:-

To find the length of the longest consecutive elements sequence, we can follow these steps:

1. Create a set to store all the elements of the array.
2. Iterate through the array and insert each element into the set.
3. For each element, check if its previous consecutive element (num-1) exists in the set. If it does not exist, it means the current element is the starting element of a sequence.
4. For each starting element, keep incrementing the current element (num+1) and checking if it exists in the set. This will help find the consecutive elements in the sequence.
5. Keep track of the maximum length of consecutive elements encountered.
6. Return the maximum length as the result.

*/

// CODE:
int longestConsecutive(vector<int> &nums)
{
    unordered_map<int, int> mp;
    for (int i = 0; i < nums.size(); i++)
    {
        mp[nums[i]]++;
    }
    int ans = 0;
    for (int i = 0; i < nums.size(); i++)
    {
        int start = nums[i];
        // check whehter this can be the start of the subsequence
        if (mp.find(nums[i] - 1) == mp.end())
        {
            int temp = 1;
            int nxt = nums[i];
            while (mp.find(nxt + 1) != mp.end())
            {
                temp++;
                nxt++;
            }
            ans = max(ans, temp);
        }
    }
    return ans;
}

// TIME COMPLEXITY: O(n), where n is the size of the input array.
// SPACE COMPLEXITY: O(n), as we are using a set to store the elements of the array.

```

2.Medium\11.Set_matrix_0's.cpp

```cpp
/*
QUESTION:
Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's.

Example 1:
Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
Output: [[1,0,1],[0,0,0],[1,0,1]]

Example 2:
Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

APPROACH:
To solve this problem in-place, we can follow these steps:
1. Use two boolean variables, firstRowZero and firstColZero, to check if the first row and first column contain zeros initially.
2. Iterate through the matrix and if an element is zero, set the corresponding element in the first row and first column to zero.
3. Iterate through the matrix again, excluding the first row and first column. If an element in the first row or first column is zero, set the current element to zero.
4. Finally, based on the values in firstRowZero and firstColZero, set the first row and first column to zero if needed.

TIME COMPLEXITY: O(m * n), where m and n are the dimensions of the matrix.
SPACE COMPLEXITY: O(1), as we are using constant extra space.

*/

// CODE:
void setZeroes(vector<vector<int>>& matrix) {
    int m = matrix.size();
    int n = matrix[0].size();
    bool firstRowZero = false;
    bool firstColZero = false;

    // Check if the first row contains zero
    for (int j = 0; j < n; j++) {
        if (matrix[0][j] == 0) {
            firstRowZero = true;
            break;
        }
    }

    // Check if the first column contains zero
    for (int i = 0; i < m; i++) {
        if (matrix[i][0] == 0) {
            firstColZero = true;
            break;
        }
    }

    // Mark zeros in the first row and 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 rows to zero
    for (int i = 1; i < m; i++) {
        if (matrix[i][0] == 0) {
            for (int j = 1; j < n; j++) {
                matrix[i][j] = 0;
            }
        }
    }

    // Set columns to zero
    for (int j = 1; j < n; j++) {
        if (matrix[0][j] == 0) {
            for (int i = 1; i < m; i++) {
                matrix[i][j] = 0;
            }
        }
    }

    // Set first row to zero
    if (firstRowZero) {
        for (int j = 0; j < n; j++) {
            matrix[0][j] = 0;
        }
    }

    // Set first column to zero
    if (firstColZero) {
        for (int i = 0; i < m; i++) {
            matrix[i][0] = 0;
        }
    }
}

// TIME COMPLEXITY: O(m * n), where m and n are the dimensions of the matrix.
// SPACE COMPLEXITY: O(1), as we are using constant extra space.

```

2.Medium\12.Rotate_matrix.cpp

```cpp
/*
QUESTION:-

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

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

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

*/

/*
APPROACH:-

To rotate the image by 90 degrees clockwise in-place, we can follow these steps:

1. Transpose the matrix: Iterate over the matrix and swap each element (i, j) with its corresponding element (j, i). This step transforms rows into columns.

2. Reverse each row: Iterate over each row in the transposed matrix and reverse the elements. This step ensures the rotation in a clockwise direction.

*/

// CODE:

void rotate(vector<vector<int>>& matrix) {
    // Transpose the matrix
    int n = matrix.size();
    int m = matrix[0].size();
    for(int i=0; i<n; i++){
        // note here we move
        for(int j=0; j<i; j++){
            swap(matrix[i][j],matrix[j][i]);
        }
    }

    // Reverse each row
    for(int i=0; i<n; i++){
        reverse(matrix[i].begin(),matrix[i].end());
    }
}

// TIME COMPLEXITY = O(N^2), where N is the size of the matrix.
// SPACE COMPLEXITY = O(1)

```

2.Medium\13.Spiral_traversal.cpp

```cpp
/*
QUESTION:-

Given an m x n matrix, return all elements of the matrix in spiral order.

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

Example 2:
Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]

*/

/*
APPROACH:-

To traverse the matrix in a spiral order, we can use the following steps:

1. Initialize four variables: top, bottom, left, and right to keep track of the boundaries of the current spiral.
2. Create an empty vector called 'ans' to store the elements in spiral order.
3. While the top boundary is less than or equal to the bottom boundary and the left boundary is less than or equal to the right boundary:
   - Traverse the top row from left to right and add each element to 'ans'.
   - Increment the top boundary.
   - Traverse the right column from top to bottom and add each element to 'ans'.
   - Decrement the right boundary.
   - Check if the top boundary is still less than or equal to the bottom boundary:
     - Traverse the bottom row from right to left and add each element to 'ans'.
     - Decrement the bottom boundary.
   - Check if the left boundary is still less than or equal to the right boundary:
     - Traverse the left column from bottom to top and add each element to 'ans'.
     - Increment the left boundary.
4. Return the 'ans' vector containing all the elements in spiral order.

*/

// CODE:

vector<int> spiralOrder(vector<vector<int>>& matrix) {
    int n = matrix.size();
    int m = matrix[0].size();
    int top = 0, bottom = n - 1;
    int left = 0, right = m - 1;
    vector<int> ans;

    while (top <= bottom && left <= right) {
        // Traverse top row
        for (int i = left; i <= right; i++) {
            ans.push_back(matrix[top][i]);
        }
        top++;

        // Traverse right column
        for (int i = top; i <= bottom; i++) {
            ans.push_back(matrix[i][right]);
        }
        right--;

        // Traverse bottom row
        if (top <= bottom) {
            for (int i = right; i >= left; i--) {
                ans.push_back(matrix[bottom][i]);
            }
            bottom--;
        }

        // Traverse left column
        if (left <= right) {
            for (int i = bottom; i >= top; i--) {
                ans.push_back(matrix[i][left]);
            }
            left++;
        }
    }

    return ans;
}

// TIME COMPLEXITY: O(N), where N is the total number of elements in the matrix.
// SPACE COMPLEXITY: O(1)

```