3.Hard\01.Pascal_triangle.cpp

```
/*
**Question:**

Given an integer `rowIndex`, return the `rowIndex`th (0-indexed) row of Pascal's triangle.

In Pascal's triangle, each number is the sum of the two numbers directly above it.

Example:

Input: `rowIndex = 3`

Output: `[1, 3, 3, 1]`
*/

/*
**APPROACH:**
To generate the `rowIndex`th row of Pascal's triangle, we can use the property that each number is the sum of the two numbers directly above it. We start with the base case of the first row, which is `[1]`. Then, for each subsequent row, we calculate the elements using the formula `C(n, k) = C(n-1, k-1) * (n-k+1) / k`, where `C(n, k)` represents the binomial coefficient.

**CODE:**
*/

vector<int> getRow(int rowIndex) {
    vector<int> row(rowIndex + 1, 1); // Initialize the row with 1s
    long long coefficient = 1;

    for (int col = 1; col <= rowIndex; col++) {
        coefficient = coefficient * (rowIndex - col + 1) / col;
        row[col] = coefficient;
    }

    return row;
}

/*
**COMPLEXITY ANALYSIS:**
- Time Complexity: O(rowIndex)
  - We iterate over each element in the row and calculate its value using the binomial coefficient formula.
- Space Complexity: O(rowIndex)
  - We use additional space to store the row of Pascal's triangle.

Overall, the algorithm has a linear time complexity and linear space complexity.
*/

```

3.Hard\02.Majority_element_2.cpp

```cpp
/*
QUESTION:
Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

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

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

APPROACH:
To find all elements that appear more than ⌊ n/3 ⌋ times, we can use the Boyer-Moore Majority Vote algorithm. This algorithm helps us find potential candidates that could appear more than ⌊ n/3 ⌋ times in a single pass. After finding the candidates, we count their occurrences and return the elements that meet the criteria.

1. Initialize two candidate variables, c1 and c2, and their corresponding vote counters, vote1 and vote2.
2. Iterate through the array:
   - If the current element matches c1, increment vote1.
   - Else if the current element matches c2, increment vote2.
   - Else if vote1 is 0, assign the current element to c1 and set vote1 to 1.
   - Else if vote2 is 0, assign the current element to c2 and set vote2 to 1.
   - Else, decrement both vote1 and vote2.
3. After finding the potential candidates, count the occurrences of each candidate using cnt1 and cnt2.
4. If cnt1 is greater than ⌊ n/3 ⌋, add c1 to the result vector.
5. If cnt2 is greater than ⌊ n/3 ⌋ and c2 is different from c1, add c2 to the result vector.
6. Return the result vector containing the elements that appear more than ⌊ n/3 ⌋ times.


*/

vector<int> majorityElement(vector<int> &nums)
{
    int c1 = 0, c2 = 0, vote1 = 0, vote2 = 0;

    // Finding potential candidates
    for (int i = 0; i < nums.size(); i++)
    {
        if (c1 == nums[i])
        {
            vote1++;
        }
        else if (c2 == nums[i])
        {
            vote2++;
        }
        else if (vote1 == 0)
        {
```

```cpp
            c1 = nums[i];
            vote1 = 1;
        }
        else if (vote2 == 0)
        {
            c2 = nums[i];
            vote2 = 1;
        }
        else
        {
            vote1--;
            vote2--;
        }
    }

    vector<int> ans;
    int cnt1 = 0, cnt2 = 0;

    // Counting occurrences of potential candidates
    for (auto it : nums)
    {
        if (it == c1)
        {
            cnt1++;
        }
        if (it == c2)
        {
            cnt2++;
        }
    }

    // Checking if candidates appear more than ⌊ n/3 ⌋ times
    if (cnt1 > nums.size() / 3)
    {
        ans.push_back(c1);
    }
    if (cnt2 > nums.size() / 3 && c2 != c1)
    {
        ans.push_back(c2);
    }

    return ans;
}

// TIME COMPLEXITY: O(n), where n is the size of the input array.
// SPACE COMPLEXITY: O(1), as we are using a constant amount of extra space.

```

3.Hard\03.3_sum.cpp

```cpp
/*
QUESTION:
Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Example:
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not matter.
*/

/*
APPROACH:
To find all triplets that sum up to zero, we can follow these steps:
1. Sort the input array in non-decreasing order.
2. Iterate through the array and fix the first element as nums[k] (where k = 0 to n-1).
3. Use two pointers (i and j) to find the other two elements such that nums[i] + nums[j] = -nums[k].
4. Move the pointers accordingly to find all possible triplets.
5. Skip duplicate elements to avoid duplicate triplets.
6. Return the resulting triplets.
*/

vector<vector<int>> threeSum(vector<int> &nums)
{
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());

    for (int k = 0; k < nums.size(); k++)
    {
        int i = k + 1;
        int j = nums.size() - 1;
        int target = -nums[k];

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

            if (sum == target)
            {
                ans.push_back({nums[k], nums[i], nums[j]});
                i++;
                j--;

                // Skip duplicate elements
                while (i < j && nums[i] == nums[i - 1])
                    i++;
```

```cpp
                while (i < j && nums[j] == nums[j + 1])
                    j--;
            }
            else if (sum < target)
            {
                i++;
            }
            else
            {
                j--;
            }
        }

        // Skip duplicate elements
        while (k + 1 < nums.size() && nums[k + 1] == nums[k])
            k++;
    }

    return ans;
}

/*
TIME COMPLEXITY: O(n^2), where n is the size of the input array.
The sorting step takes O(n log n), and the two-pointer traversal takes O(n^2) in the worst case.
Hence, the overall time complexity is O(n^2).

SPACE COMPLEXITY: O(1), as we are using a constant amount of extra space for storing the output and variables.
*/

```

3.Hard\04.4_sum.cpp

```
/*
QUESTION:
Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]] such that:
- 0 <= a, b, c, d < n
- a, b, c, and d are distinct.
- nums[a] + nums[b] + nums[c] + nums[d] == target

Example:
Input: nums = [1,0,-1,0,-2,2], target = 0
Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

APPROACH:
To find the unique quadruplets that sum up to the target, we can use a similar approach as the threeSum problem. We will fix two elements (nums[a] and nums[b]) and use two pointers to find the remaining two elements (nums[c] and nums[d]) that sum up to the target.

1. Sort the input array nums in ascending order.
2. Iterate through the array with two pointers: a and b.
3. For each pair of elements nums[a] and nums[b], use two pointers c and d to find the remaining two elements that sum up to the target.
   - Initialize c as b + 1 and d as the last index of the array.
   - Calculate the target sum as trgt = target - (nums[a] + nums[b]).
   - While c < d, compare the sum of nums[c] and nums[d] with the target sum.
     - If the sum is equal to the target sum, we found a quadruplet. Add it to the answer and move the pointers c and d.
       - Important: Skip any duplicate elements while moving c and d.
     - If the sum is greater than the target sum, decrement d.
     - If the sum is less than the target sum, increment c.
4. Skip any duplicate elements for pointers a and b to avoid duplicate quadruplets.
5. Return the answer array containing unique quadruplets.


CODE:
*/

vector<vector<int>> fourSum(vector<int> &nums, int target)
{
    vector<vector<int>> ans;
    long long trgt = (long long)(target); // to handle overflow
    sort(nums.begin(), nums.end());

    for (int a = 0; a < nums.size(); a++)
    {
        for (int b = a + 1; b < nums.size(); b++)
        {
            if (a == b)
                continue;

            int c = b + 1;
            int d = nums.size() - 1;
            long long tar = trgt - (nums[a] + nums[b]);
```

```cpp

            while (c < d)
            {
                long long sum = nums[c] + nums[d];

                if (sum == tar)
                {
                    ans.push_back({nums[a], nums[b], nums[c], nums[d]});
                    c++;
                    d--;

                    // Skip duplicate elements
                    while (c < d && nums[c] == nums[c - 1])
                        c++;
                    while (c < d && nums[d] == nums[d + 1])
                        d--;
                }
                else if (sum > tar)
                {
                    d--;
                }
                else
                {
                    c++;
                }
            }

            // Skip duplicate elements
            while (b + 1 < nums.size() && nums[b + 1] == nums[b])
                b++;
        }

        // Skip duplicate elements
        while (a + 1 < nums.size() && nums[a + 1] == nums[a])
            a++;
    }

    return ans;
}

/*
TIME COMPLEXITY: O(n^3), where n is the size of the input array nums.
SPACE COMPLEXITY: O(1), as we are using a constant amount of extra space.
*/

```

3.Hard\05.Largest_subarray_with_0sum.cpp

```cpp
/*
QUESTION:
Given an array with both positive and negative integers, we need to compute the length of the largest subarray with a sum of 0.

Example:
Input:
N = 8
A[] = {15, -2, 2, -8, 1, 7, 10, 23}
Output: 5
Explanation: The largest subarray with a sum of 0 will be -2, 2, -8, 1, 7.

APPROACH:
To find the length of the largest subarray with a sum of 0, we can use a technique called prefix sum.
1. Create a prefix sum array of the same size as the input array.
2. Initialize a map to store the prefix sum and its corresponding index. Initialize it with an entry for prefix sum 0 and index -1.
3. Iterate through the input array and calculate the prefix sum by adding each element.
4. For each prefix sum encountered, check if it exists in the map. If it does, update the answer by taking the maximum of the current answer and the difference between the current index and the index stored in the map for that prefix sum.
5. If the prefix sum is not found in the map, add it to the map with its corresponding index.
6. Finally, return the answer as the length of the largest subarray with a sum of 0.

CODE:
*/

int maxLen(vector<int> &A, int n)
{
    unordered_map<int, int> mp;
    mp[0] = -1;
    int pref_sum = 0;
    int ans = 0;

    for (int i = 0; i < n; i++)
    {
        pref_sum += A[i];
        if (mp.find(pref_sum) != mp.end())
        {
            ans = max(ans, i - mp[pref_sum]);
        }
        else
        {
            mp[pref_sum] = i;
        }
    }

    return ans;
}

/*
```

```
TIME COMPLEXITY: O(n), where n is the size of the input array A.
SPACE COMPLEXITY: O(n), as we are using a map to store the prefix sums and their corresponding indices.
*/

```

3.Hard\06.Subarrays_with_xor_k.cpp

```cpp
/*QUESTION:
Given an array 'A' consisting of 'N' integers and an integer 'B', find the number of subarrays of array 'A' whose bitwise XOR of all elements is equal to 'B'.

Example:
Input: 'N' = 4, 'B' = 2
'A' = [1, 2, 3, 2]
Output: 3
Explanation: Subarrays have bitwise xor equal to '2' are: [1, 2, 3, 2], [2], [2].

APPROACH:
To find the number of subarrays with bitwise XOR equal to B, we can use the technique of prefix XOR along with a hashmap.
1. Initialize a variable `prefixXOR` to keep track of the prefix XOR while iterating through the array.
2. Initialize a variable `count` to keep track of the count of subarrays with XOR equal to B.
3. Initialize a hashmap `xorCount` to store the frequency of prefix XOR values encountered so far.
4. Set the initial prefix XOR to 0 and set its count to 1 in the `xorCount` hashmap.
5. Iterate through the array and update the prefix XOR by XOR-ing each element.
6. Check if the current prefix XOR is equal to B. If it is, increment the `count` variable.
7. Check if the XOR of the current prefix XOR with B exists in the `xorCount` hashmap. If it does, add the count of that XOR value to the `count` variable.
8. Increment the count of the current prefix XOR in the `xorCount` hashmap.
9. Finally, return the `count` variable as the number of subarrays with XOR equal to B.

CODE:
*/

int subarraysWithSumK(vector<int> a, int b) {
    int pref_xr = 0;
    int ans = 0;
    unordered_map<int, int> mp;

    for(int i = 0; i < a.size(); i++){
        pref_xr = pref_xr ^ a[i];

        if(pref_xr == b)
            ans++;

        if(mp.find(pref_xr ^ b) != mp.end()){
            ans += mp[pref_xr ^ b];
        }

        mp[pref_xr]++;
    }

    return ans;
}

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

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


```

3.Hard\07.Merge_overlapping_subinterval.cpp

```
/*
QUESTION:
Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals and return an array of non-overlapping intervals that cover all the intervals in the input.

Example 1:
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

APPROACH:
To merge overlapping intervals, we can follow these steps:
1. Sort the intervals based on the start time.
2. Initialize a vector `ans` to store the merged intervals.
3. Add the first interval from the sorted intervals to the `ans` vector.
4. Iterate through the remaining intervals:
   - If the start time of the current interval is less than or equal to the end time of the last interval in the `ans` vector, it means they overlap. Update the end time of the last interval in the `ans` vector if necessary.
   - If the start time of the current interval is greater than the end time of the last interval in the `ans` vector, it means they don't overlap. Add the current interval to the `ans` vector.
5. Return the `ans` vector as the merged non-overlapping intervals.

CODE:
*/

vector<vector<int>> merge(vector<vector<int>>& intervals) {
    sort(intervals.begin(), intervals.end());
    vector<vector<int>> ans;
    ans.push_back(intervals[0]);

    for(int i = 1; i < intervals.size(); i++){
        if(ans.back()[1] >= intervals[i][0]){
            ans.back()[1] = max(ans.back()[1], intervals[i][1]);
        }
        else{
            ans.push_back(intervals[i]);
        }
    }

    return ans;
}

/*
TIME COMPLEXITY: O(nlogn), where n is the number of intervals in the input.
The sorting step takes O(nlogn) time, and the merging step takes O(n) time.
Overall, the time complexity is dominated by the sorting step.
SPACE COMPLEXITY: O(n), where n is the number of intervals in the input.
We are using additional space to store the merged intervals in the `ans` vector.
*/
```

```
47
48
```

3.Hard\08.Merge_2_sorted_array_without_space.cpp

```
/*
QUESTION:
You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.
Merge nums1 and nums2 into a single array sorted in non-decreasing order.
The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

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

APPROACH:
To merge two sorted arrays, nums1 and nums2, into nums1, we can use a two-pointer approach.
1. Initialize three pointers: i, j, and k, where i points to the last valid element of nums1, j points to the last element of nums2, and k points to the last index of the merged array nums1.
2. Start from the end of the arrays and compare the elements at i and j.
3. If the element at nums1[i] is greater than the element at nums2[j], swap it with the element at nums1[k], decrement i and k.
4. Otherwise, swap the element at nums2[j] with the element at nums1[k], decrement j and k.
5. Repeat steps 3 and 4 until all elements in nums1 and nums2 have been merged.
6. If there are still elements remaining in nums2 after merging, copy them to the remaining positions in nums1.

CODE:
*/

void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    int i = m - 1; // Pointer for nums1
    int j = n - 1; // Pointer for nums2
    int k = m + n - 1; // Pointer for merged array nums1

    while (i >= 0 && j >= 0) {
        if (nums1[i] > nums2[j]) {
            swap(nums1[i], nums1[k]);
            i--;
            k--;
        } else {
            swap(nums2[j], nums1[k]);
            j--;
            k--;
        }
    }

    // Copy remaining elements from nums2 to nums1 if any
    while (j >= 0) {
        swap(nums2[j], nums1[k]);
```

```
        j--;
        k--;
    }
}

/*
TIME COMPLEXITY: O(m + n), where m and n are the lengths of nums1 and nums2 respectively.
The merging process requires iterating through both arrays once.
SPACE COMPLEXITY: O(1)
The merge is performed in-place without using any additional space.

*/


```

3.Hard\09.Repeating_and_missing_numbers.cpp

```
/*
QUESTION:
Given an unsorted array Arr of size N of positive integers. One number 'A' from set {1, 2,....,N} is missing and one number 'B' occurs twice in the array. Find these two numbers.

Example:
Input:
N = 2
Arr[] = {2, 2}
Output: 2 1
Explanation: Repeating number is 2 and the smallest positive missing number is 1.

APPROACH:
To find the missing and repeating numbers in the given unsorted array, we can utilize the properties of summation and sum of squares. Let's denote the missing number as 'x' and the repeating number as 'y'.

1. Calculate the optimal sum 'optSum' using the formula: optSum = N * (N + 1) / 2, where N is the size of the array.
2. Calculate the optimal sum of squares 'opt2Sum' using the formula: opt2Sum = N * (N + 1) * (2 * N + 1) / 6.
3. Calculate the actual sum 'actSum' and actual sum of squares 'act2Sum' of the given array.
4. Find the difference between the optimal sum and the actual sum: xMinusY = optSum - actSum.
5. Find the difference between the optimal sum of squares and the actual sum of squares: x2MinusY2 = opt2Sum - act2Sum.
6. Calculate the sum of 'x' and 'y': xPlusY = x2MinusY2 / xMinusY.
7. Calculate 'x' and 'y' using the equations: x = (xPlusY + xMinusY) / 2 and y = xPlusY - x.

CODE:
*/

vector<int> findTwoElement(vector<int> arr, int N) {
    long long n = N;
    long long optSum = n * (n + 1) / 2; // Sum if all elements are present once
    long long opt2Sum = n * (n + 1) * (2 * n + 1) / 6; // Optimum sum of squares
    long long actSum = 0; // Actual sum of the given array
    long long act2Sum = 0; // Actual sum of squares

    for (auto it : arr) {
        actSum += it;
        act2Sum += (long long)it * (long long)it;
    }

    long long xMinusY = optSum - actSum;
    long long x2MinusY2 = opt2Sum - act2Sum;
    long long xPlusY = x2MinusY2 / xMinusY;

    long long x = (xPlusY + xMinusY) / 2;
    long long y = xPlusY - x;

    return {(int)y, (int)x};
}
```

```
47
48 /*
49 TIME COMPLEXITY: O(N), where N is the size of the array.
50 SPACE COMPLEXITY: O(1).
51 */
52
```

3.Hard\10.Count_inversions.cpp

```cpp
/*
QUESTION:
Given an array of integers. Find the Inversion Count in the array.

Inversion Count: For an array, inversion count indicates how far (or close) the array is from being sorted. If the array is already sorted then the inversion count is 0. If an array is sorted in the reverse order then the inversion count is the maximum.
Formally, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

Example 1:
Input: N = 5, arr[] = {2, 4, 1, 3, 5}
Output: 3
Explanation: The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

Example 2:
Input: N = 5, arr[] = {2, 3, 4, 5, 6}
Output: 0
Explanation: As the sequence is already sorted, there is no inversion count.

APPROACH:
To find the inversion count in the array, we can utilize the merge sort algorithm. The idea is to divide the array into two halves, recursively count the inversions in each half, and then merge the two halves while counting the inversions across them.

CODE:
*/

long long int inv_cnt = 0;

long long int merge(long long start, long long mid, long long end, long long arr[]) {
    long long leftsize = mid - start + 1;
    long long rightsize = end - mid;
    long long left[leftsize], right[rightsize];

    for (long long i = 0; i < leftsize; i++) {
        left[i] = arr[start + i];
    }
    for (long long i = 0; i < rightsize; i++) {
        right[i] = arr[mid + 1 + i];
    }

    long long i = 0, j = 0, k = start;
    while (i < leftsize && j < rightsize) {
        if (left[i] > right[j]) {
            inv_cnt += leftsize - i;
            arr[k++] = right[j++];
        } else {
            arr[k++] = left[i++];
        }
    }
    while (i < leftsize) {
        arr[k++] = left[i++];
```

```cpp
    }
    while (j < rightsize) {
        arr[k++] = right[j++];
    }
}

void mergesort(long long start, long long end, long long arr[]) {
    if (start >= end)
        return;
    long long mid = start + (end - start) / 2;
    mergesort(start, mid, arr);
    mergesort(mid + 1, end, arr);
    merge(start, mid, end, arr);
}

long long int inversionCount(long long arr[], long long N) {
    mergesort(0, N - 1, arr);
    return inv_cnt;
}

/*
TIME COMPLEXITY: O(N log N), where N is the size of the array.
SPACE COMPLEXITY: O(N).
*/

```

3.Hard\11.Reverse_pairs.cpp

```cpp
/*
QUESTION:
Given an integer array nums, return the number of reverse pairs in the array.
A reverse pair is a pair (i, j) where:
0 <= i < j < nums.length and
nums[i] > 2 * nums[j].

Example:
Input: nums = [1,3,2,3,1]
Output: 2
Explanation: The reverse pairs are:
(1, 4) --> nums[1] = 3, nums[4] = 1, 3 > 2 * 1
(3, 4) --> nums[3] = 3, nums[4] = 1, 3 > 2 * 1

APPROACH:
To solve this problem, we can use the merge sort algorithm. While merging the two sorted subarrays, we can count the number of reverse pairs.

1. Define a variable 'rev_pair' to store the count of reverse pairs.
2. Implement the 'merge' function to merge two subarrays and count the reverse pairs.
3. Implement the 'mergesort' function to recursively divide the array into subarrays and perform merge sort.
4. Initialize 'rev_pair' to 0 and call the 'mergesort' function on the given array.
5. Return the 'rev_pair' as the result.

CODE:
*/

int rev_pair = 0;

void merge(int start, int mid, int end, vector<int>& nums){
    int left_size = mid - start + 1;
    int right_size = end - mid;
    vector<int> left(left_size);
    vector<int> right(right_size);

    for(int i = 0; i < left_size; i++){
        left[i] = nums[start + i];
    }
    for(int i = 0; i < right_size; i++){
        right[i] = nums[mid + 1 + i];
    }

    // main logic resides here
    int m = 0;
    for(int i = 0; i < left_size; i++){
        while(m < right_size && left[i] > (long long)2 * right[m]){
            m++;
        }
        rev_pair += m;
    }

```

```cpp
    int i = 0, j = 0, k = start;
    while(i < left_size && j < right_size){
        if(left[i] > right[j]){
            nums[k++] = right[j++];
        }
        else{
            nums[k++] = left[i++];
        }
    }
    while(i < left_size){
        nums[k++] = left[i++];
    }
    while(j < right_size){
        nums[k++] = right[j++];
    }
}

void mergesort(int start, int end, vector<int>& nums){
    if(start >= end)
        return;
    int mid = start + (end - start) / 2;
    mergesort(start, mid, nums);
    mergesort(mid + 1, end, nums);
    merge(start, mid, end, nums);
}

int reversePairs(vector<int>& nums) {
    rev_pair = 0;
    mergesort(0, nums.size() - 1, nums);
    return rev_pair;
}

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

```

3.Hard\12.Maximum_product_subarray.cpp

```cpp
/*QUESTION:

Given an integer array nums, find a subarray that has the largest product, and return the product.

Example:

Input: nums = [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.

APPROACH:

To find the subarray with the largest product, we iterate through the array while keeping track of the current product. We maintain two variables: `ans` to store the maximum product found so far and `prdct` to store the current product. Since negative numbers can change the sign and potentially result in a larger product, we run the loop twice, once from left to right and once from right to left.

CODE:*/

int maxProduct(vector<int>& nums) {
    int ans = INT_MIN;
    int prdct = 1;

    // Iterate from left to right
    for (int i = 0; i < nums.size(); i++) {
        prdct = prdct * nums[i];
        ans = max(ans, prdct);
        if (prdct == 0)
            prdct = 1;
    }

    prdct = 1;

    // Iterate from right to left
    for (int i = nums.size() - 1; i >= 0; i--) {
        prdct = prdct * nums[i];
        ans = max(ans, prdct);
        if (prdct == 0)
            prdct = 1;
    }

    return ans;
}

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

```

3.Hard\13.Longest_subarray_with_sum_k_containg_+ves_and_-ves.cpp

```
/*Question:
Given an array containing N integers and an integer K, find the length of the longest subarray with the sum of the elements equal to K.

Example:
Input:
A[] = {10, 5, 2, 7, 1, 9}
K = 15
Output:
4
Explanation:
The sub-array is {5, 2, 7, 1}.

Approach:
To solve this problem, we can use a prefix sum approach along with a hashmap to keep track of the prefix sums encountered so far. We iterate through the array and maintain a prefix sum variable. At each index, we check if the prefix sum equals K, in which case we update the maximum length of the subarray found so far. Additionally, we check if the current prefix sum minus K exists in the hashmap. If it does, it means there is a subarray between the previous occurrence of the prefix sum minus K and the current index that sums up to K. We update the maximum length accordingly. We store the prefix sums and their corresponding indices in the hashmap.

Code:
*/
int lenOfLongSubarr(int A[], int N, int K) {
    int pref_sum = 0;
    int ans = 0;
    unordered_map<int, int> mp;

    for (int i = 0; i < N; i++) {
        pref_sum += A[i];
        if (pref_sum == K)
            ans = max(ans, i + 1);
        if (mp.find(pref_sum - K) != mp.end())
            ans = max(ans, i - mp[pref_sum - K]);
        if (mp.find(pref_sum) == mp.end())
            mp[pref_sum] = i;
    }
    return ans;
}
/*
Time Complexity: The code iterates through the array once, resulting in a time complexity of O(N), where N is the size of the array.
Space Complexity: The code uses an unordered map to store the prefix sums and their corresponding indices. In the worst case, all elements of the array could be distinct, leading to a space complexity of O(N) to store the prefix sums in the map.
*/
```