1.Easy\01.Largest_element_in_array.cpp

```cpp
/*
QUESTION:-
Given an array A[] of size n. The task is to find the largest element in it.

Example:

Input:
n = 5
A[] = {1, 8, 7, 56, 90}
Output:
90
Explanation:
The largest element of given array is 90
*/

/*
APPROACH:-
-> Intialize the ans with starting element
-> Traverse the entire array and update the ans if the element is greater then ans
-> Finally, return the ans
*/

// CODE:-
int largest(int arr[], int n)
{
    int ans = arr[0];
    for (int i = 1; i < n; i++)
    {
        if (arr[i] > ans)
            ans = arr[i];
    }
    return ans;
}

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

1.Easy\02.Second_largest_element_in_array.cpp

```cpp
/*
QUESTION:-
Given an array Arr of size N, print second largest distinct element from an array.

Example:

Input:
N = 6
Arr[] = {12, 35, 1, 10, 34, 1}
Output: 34
Explanation: The largest element of the
array is 35 and the second largest element
is 34.
*/

/*
APPROACH
-> If the current element is larger than ‘large’ then update second_large and large variables
-> Else if the current element is larger than ‘second_large’ then we update the variable second_large.
-> Once we traverse the entire array, we would find the second largest element in the variable second_large.
*/

// CODE:-
int print2largest(int arr[], int n)
{
    int prev = -1, curr = arr[0];
    for (int i = 1; i < n; i++)
    {
        if (arr[i] > curr)
        {
            prev = curr;
            curr = arr[i];
        }
        else if (arr[i] > prev && arr[i] != curr)
            prev = arr[i];
    }
    return prev;
}

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

1.Easy\03.Check_if_array_is_sorted_and_rotated.cpp

```cpp
/*
QUESTION:-
Given an array nums, return true if the array was originally sorted in non-decreasing order, then rotated some number of positions (including zero). Otherwise, return false.
There may be duplicates in the original array.
Example 1:

Input: nums = [3,4,5,1,2]
Output: true
Explanation: [1,2,3,4,5] is the original sorted array.
You can rotate the array by x = 3 positions to begin on the the element of value 3: [3,4,5,1,2].
Example 2:

Input: nums = [2,1,3,4]
Output: false
Explanation: There is no sorted array once rotated that can make nums.

*/

/*
APPROACH:-
Compare all neignbour elements (a,b) in A,
the case of a > b can happen at most once.

Note that the first element and the last element are also connected.

If all a <= b, A is already sorted so answer is true.
If all a <= b but only one a > b, and the first element is greater than equal to last element we can rotate and make b the first element so answer is true.
Other case, return false.
*/

// CODE:-
bool check(vector<int> &nums)
{
    int cnt = 0;
    int n = nums.size();
    for (int i = 0; i < n - 1; i++)
    {
        if (nums[i] > nums[i + 1])
            cnt++;
    }
    if (cnt == 0)
        return true;
    else if (cnt == 1 && nums[0] >= nums[n - 1])
        return true;
    return false;
}

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

1.Easy\04.Remove_duplicates_from_sorted_array.cpp

```
/*
QUESTION:-

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in nums.
Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:
Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.
Return k.

Example 1:

Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).
Example 2:

Input: nums = [0,0,1,1,1,2,2,3,3,4]
Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).
*/

/*
APPROACH:-
-> The idea, is to use keep a pointer k which signifies that upto here the array is sorted
-> Now travese the entire array and if arr[k]!=arr[j] that is arr[j] is a unique value hence it should be included
   so increment the k and swap arr[k] with arr[j]
-> Return k+1, +1 is because of 0 based indexing
*/

// CODE:-
int removeDuplicates(vector<int> &nums)
{
    int k = 0; // upto k array contains unique elements
    for (int j = 1; j < nums.size(); j++)
    {
        if (nums[k] != nums[j])
        {
            k++;
            swap(nums[k], nums[j]);
        }
    }
    return k + 1;
}
```

1.Easy\05.Rotate_array_left_by_1place.cpp

```cpp
/*
QUESTION:-
Given an array "ARR' containing 'N' elements, rotate this array Left by once means to shift all elements by one place to the left and move the first element to the last position in the array.

Example:
Input: 'N' 5, 'ARR' = [1, 2, 3, 4, 5]
Output: [2, 3, 4, 5, 1]

Explanation:
We moved the 2nd element to the 1st position and 3rd element to the 2nd position and 4th element to the 3rd position and 5th element to the 4th position and move oth element to the 5th position.
*/

/*
APPROACH:-
-> By observing we can the ans is the arr where arr[i] = arr[i+1] and at last place we will have arr[0]
-> Before traversing store the arr[0] in temp and then traverse the array and make arr[i] = arr[i+1]
-> Make arr[n-1] = arr[0], where n is the size of the array
*/

// CODE:-
vector<int> rotateArray(vector<int> &arr, int n)
{
    int temp = arr[0];
    for (int i = 0; i < n - 1; i++)
    {
        arr[i] = arr[i + 1];
    }
    arr[n - 1] = temp;

    return arr;
}

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

1.Easy\06.Rotate_array_left&right_by_k_places.cpp

```
/*
QUESTION:-

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

Example 1:

Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
Example 2:

Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]
*/

/*
APPROACH:-
To rotate the array k places to right follow below steps
-> Reverse first n-k elements
-> Reverse last k elements
-> Reverse the entire array

To rotate the array k places to left follow below steps
-> Reverse first k elements
-> Reverse last n-k elements
-> Reverse the entire array
*/

// CODE:-

// RIGHT ROATATE:-
void rightRotate(int arr[], int n, int k)
{
    k = k % n; // to keep k within the range
    reverse(arr, arr + (n - k));
    reverse(arr + (n - k), arr + n);
    reverse(arr, arr + n);
}

// LEFT ROATATE:-
void leftRotate(int arr[], int n, int k)
{
    k = k % n; // to keep k within the range
    reverse(arr, arr + k);
```

```
    reverse(arr + k, arr + n);
    reverse(arr, arr + n);
}

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

1.Easy\07.Move_0's_to_end.cpp

```cpp
/*
QUESTION:-
Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]
Example 2:

Input: nums = [0]
Output: [0]
*/

/*
APPROACH:-
-> The idea is while traversing the array if we found any zero then we have to swap it with next non-zero
*/

// CODE:-
// function to find the next non-zero element
int next_nonzero(vector<int> &a, int &j)
{
    while (j < a.size())
    {
        if (a[j] != 0)
            return j;
        j++;
    }
    return -1;
}
void moveZeroes(vector<int> &nums)
{
    int j = -1; // is to find the next non zero element
    // i signifies that upto here all elements are non-zero
    for (int i = 0; i < nums.size(); i++)
    {
        if (nums[i] != 0)
            continue;
        if (j == -1)
            j = i + 1;
        int nxt_non0 = next_nonzero(nums, j);
        if (nxt_non0 == -1)
            return;
        swap(nums[i], nums[nxt_non0]);
    }
}

```

1.Easy\09.Union_of_2_sorted_arrays.cpp

```
/*
QUESTION:-
Union of two arrays can be defined as the common and distinct elements in the two arrays.
Given two sorted arrays of size n and m respectively, find their union.


Example 1:

Input:
n = 5, arr1[] = {1, 2, 3, 4, 5}
m = 3, arr2 [] = {1, 2, 3}
Output: 1 2 3 4 5
Explanation: Distinct elements including
both the arrays are: 1 2 3 4 5.


Example 2:

Input:
n = 5, arr1[] = {2, 2, 3, 4, 5}
m = 5, arr2[] = {1, 1, 2, 3, 4}
Output: 1 2 3 4 5
Explanation: Distinct elements including
both the arrays are: 1 2 3 4 5.
*/

/*
APPROACH:-
-> Take two pointer i and j where i is for arr1 and j is for arr2 and traverse
-> While travsersing 3 cases arises
    -> arr1[ i ] == arr2[ j ]
        Here we found a common element, so insert only one element in the union.
        Let’s insert arr[i] in union and whenever we insert element we increment pointer
while pointer is not equal to the inserted element
    -> arr1[i]<arr2[j]
        Here insert arr[i]
    -> arr1[i]>arr2[j]
        Here insert arr2[j]
-> Now check if elements of any array is left to traverse then traverse that array
*/

// CODE:-
vector<int> findUnion(int arr1[], int arr2[], int n, int m)
{
    int i = 0; // i to keep track in arr1
    int j = 0; // j to keep track in arr2
    vector<int> ans;

    while (i < n && j < m)
    {

        if (arr1[i] < arr2[j])
```

```cpp
        {
            ans.push_back(arr1[i++]);
            while (i < n && arr1[i] == arr1[i - 1])
                i++;
        }
        else if (arr2[j] < arr1[i])
        {
            ans.push_back(arr2[j++]);
            while (j < m && arr2[j] == arr2[j - 1])
                j++;
        }
        // means arr1[i] = arr2[j] in that case we can insert anyone
        else
        {
            ans.push_back(arr1[i++]);
            j++;
            while (i < n && arr1[i] == arr1[i - 1])
                i++;
            while (j < m && arr2[j] == arr2[j - 1])
                j++;
        }
    }

    while (i < n)
    {
        ans.push_back(arr1[i++]);
        while (i < n && arr1[i] == arr1[i - 1])
            i++;
    }
    while (j < m)
    {
        ans.push_back(arr2[j++]);
        while (j < m && arr2[j] == arr2[j - 1])
            j++;
    }

    return ans;
}

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

1.Easy\10.Missing_number.cpp

```cpp
/*
QUESTION:-
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.

Example 1:

Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.
Example 2:

Input: nums = [0,1]
Output: 2
Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.
*/

/*
APPROACH:-
-> Calculate the optimum sum i.e. sum when all elements were present
-> Calculate the actual array's sum
-> Return the optimum sum - actual sum
*/

// CODE:-
int missingNumber(vector<int> &nums)
{
    int n = nums.size();
    long long optimum_sum = (n * (n + 1)) / 2; // the sum if no number is absent
    long long actual_sum = 0;
    for (auto it : nums)
    {
        actual_sum += it;
    }
    return optimum_sum - actual_sum;
}

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

1.Easy\11.Max_consecutive_1's.cpp

```
/*
QUESTION:-
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.

Example 1:

Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.
Example 2:

Input: nums = [0,1]
Output: 2
Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.
*/

/*
APPROACH:-
-> Traverse the entire array and within it run a loop while element's are equal to 1 and store the count
-> Update the ans as max(ans,cnt)
*/

// CODE:-
int findMaxConsecutiveOnes(vector<int> &nums)
{
    int ans = 0;
    for (int i = 0; i < nums.size(); i++)
    {
        int cnt = 0; // to store the count of consecutive 1's
        while (i < nums.size() && nums[i] == 1)
        {
            cnt++;
            i++;
        }
        ans = max(ans, cnt);
    }
    return ans;
}

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

```

1.Easy\12.Longest_subarray_with_given_sum.cpp

```cpp
/*
QUESTION:-
You are given an array 'A' of size 'N' and an integer 'K'. You need to print the length of the longest subarray of array 'A' whose sum = 'K'.
Example:
Input: 'N' = 7 'K' = 3
'A' = [1, 2, 3, 1, 1, 1, 1]
Output: 3
Explanation: Subarrays whose sum = '3' are:
[1, 2], [3], [1, 1, 1], [1, 1, 1]
Here, the length of the longest subarray is 3, which is our final answer.
*/

/*
APPROACH:-
-> Use sliding window approach using two pointers start and end
-> Run a loop to traverse the entire array add from end and subtract from start when sum>k
-> If sum==k then, update the ans now, window size = end-start+1
*/

// CODE:-
int longestSubarrayWithSumK(vector<int> a, long long k)
{
    int start = 0;
    int ans = 0;
    long long sum = 0;
    int n = a.size();

    for (int end = 0; end < n; end++)
    {
        sum += a[end];
        while (sum > k)
        {
            sum -= a[start];
            start++;
        }
        if (sum == k)
        {
            ans = max(ans, end - start + 1);
        }
    }
    return ans;
}

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

1.Easy\13.Find_element_present_only_once.cpp

```cpp
/*
QUESTION:-
Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.
You must implement a solution with a linear runtime complexity and use only constant extra space.


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

Example 2:
Input: nums = [4,1,2,1,2]
Output: 4
*/

/*
APPROACH:-
-> We can use XOR operation as we know xor cancles out the same elements
-> Intial xr=0 then traverse the entire array and xor each element with xr
-> Since only one element is present once and all other are present twice so the remaining element would be the
    one which is present only once cause all other gets cancels out
*/

// CODE:-
int singleNumber(vector<int> &nums)
{
    int xr = 0;
    for (int i = 0; i < nums.size(); i++)
    {
        xr = nums[i] ^ xr;
    }
    return xr;
}

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