Two Sum : Check if a pair with given sum exists in Array

Problem Statement: Given an array of integers nums[] and an integer target, return *indices of the two numbers such that their sum is equal to the target*.
Note: Assume that there is ***exactly* one solution**, and you are not allowed to use the *same* element twice. Example: If target is equal to 6 and num[1] = 3, then nums[1] + nums[1] = target is not a solution.
Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]

Explanation: Because nums[0] + nums[1] == 9, which is the required target, we return
indexes [0,1]. (0-based indexing)
```

Example 2:

```
Input Format: nums = [3,2,4,6], target = 6
Output: [1,2]

Explanation: Because nums[1] + nums[2] == 6, which is the required target, we return
indexes [1,2].
```

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Naive Approach (Brute Force)
Intuition: For each element, we try to find an element such that the sum of both elements is equal to the given target.

Approach: We traverse through the array, and for each element **i**, we try to find another element amongst the remaining elements, such that the sum of both the elements equals the target.
First Element: **i**
So the required second element will be, **target – i**
If we find both the elements, we break the loop and return the indices.
Dry Run: Given array, nums = [2,1,3,4], target = 4
Using the naive approach, we first select one element and then find the second elements.
For index 0, i = 2
Then, we iterate through index 1 to 3 to find if **target – i,** i.e. 4 – 2 = 2 exists. We find that it does not exist, so we move forward.
For index 1, i=1
Then, we iterate through index 2 to 3 to find if **target – i,** i.e. 4 – 1 = 3 exists. We find that it does exist at index 2, so we store the indices 1 and 2, break the loop, and return the indices.

Code:

```
vector<int> twoSum(vector<int>& nums, int target) {
    vector<int> res;
    for (int i = 0; i < nums.size(); ++i) {
        for (int j = i + 1; j < nums.size(); ++j) {
               if (nums[i] + nums[j] == target) {
                      res.emplace_back(i);
                      res.emplace_back(j);
                      break;
               }
        }
        if (res.size() == 2)
               break;
    }
    return res;
}
```

Time Complexity: O(N2)
Space Complexity: O(1)

Solution 2: Two-Pointer Approach
Intuition: Think about, what if the array is sorted? If the array is sorted, is it possible to reach a sum by traversing the array from both sides simultaneously?
We sort the array, use two variables, each will start from one end of the array, and traverse in both directions till we get the required sum.

Approach: We traverse through the array, and for each element **i**, we try to find another element amongst the remaining elements, such that the sum of both the elements equals the target.
First Element: **i**
So the required second element will be, **target – i**
If we find both the elements, we break the loop and return the indices.
Dry Run: Given array, nums = [2,1,3,4], target = 4
First we sort the array. So nums after sorting is [1,2,3,4]
We take two pointers, i and j. i points to index 0 and j points to index 3.
Now we check if nums[i] + nums[j] == target. In this case, they don't sum up, and nums[i] + nums[j] > target, so we will reduce j by 1.
Now, i = 0, j=2
Here, nums[i] + nums[j] == 1 + 3 == 4, which is the required target, so we store both the elements and break the loop.
Now, we run another loop to find the indices of the elements in the actual array, i.e [2,1,3,4]
Then, return the actual indices, [1,2].

Code:
```
vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> res,store;
        store = nums;
        sort(store.begin(), store.end());
        int left=0,right=nums.size()-1;
        int n1,n2;
        while(left<right){
                if(store[left]+store[right]==target){
                n1 = store[left];
                n2 = store[right];
                break;
                }
                else if(store[left]+store[right]>target)
                    right--;
                else
                    left++;
        }
        for(int i=0;i<nums.size();++i){
                if(nums[i]==n1)
                    res.emplace_back(i);
                else if(nums[i]==n2)
                    res.emplace_back(i);
        }
            return res;
        }
```

Time Complexity: O(NlogN)
Space Complexity: O(N)

Solution 3: Hashing (Most efficient)
Approach: We can solve this problem efficiently by using **hashing**. We'll use a **hash-map** to see if there's a value **target – i** that can be added to the current array value i to get the sum equals to target. If **target – i** is found in the map, we return the current index, and index stored at **target – nums[index]** location in the map.
This can be done in constant time.

Dry Run: Given array, nums = [2,3,1,4], target = 4
For index 0, i = 2
We try to find if **target – i = 4 – 2 = 2** is present in the map. It is not present in this case, so we put 0 for key 2 in the map.
For index 1, i = 3
We try to find if **target – i = 4 – 3 = 1** is present in the map. We find that it is also not present, so we put 1 for key 3 in the map.
For index 2, i = 1
We try to find if **target – i = 4 – 1 = 3** is present in the map. We find that it is present, so we store index 2 and value stored for key 3 in the map, and break the loop.
And return [1,2].

Code:
```
vector<int> twoSum(vector<int>& nums, int target) {
    vector<int> res;
    unordered_map<int, int> mp;
    for (int i = 0; i < nums.size(); ++i) {
        if (mp.find(target - nums[i]) != mp.end()) {

               res.emplace_back(i);
               res.emplace_back(mp[target - nums[i]]);
               return res;
        }
        mp[nums[i]] = i;
    }
    return res;
}
```

Time Complexity: O(N)
Space Complexity: O(N)

4 Sum | Find Quads that add up to a target value

Problem Statement: Given an array of N integers, your task is to find unique quads that add up to give a target value. In short, you need to return *an array of all the unique quadruplets* [arr[a], arr[b], arr[c], arr[d]] such that their sum is equal to a given *target*.

Pre-req: Binary Search and 2-sum problem
Note:

- 0 <= a, b, c, d < n
- a, b, c, and d are distinct.
- arr[a] + arr[b] + arr[c] + arr[d] == target
-

Example 1:

```
Input Format: arr[] = [1,0,-1,0,-2,2], target = 0

Result: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]


Explanation: We have to find unique quadruplets from
the array such that the sum of those elements is
equal to the target sum given that is 0.

The result obtained is such that the sum of the
quadruplets yields 0.
```

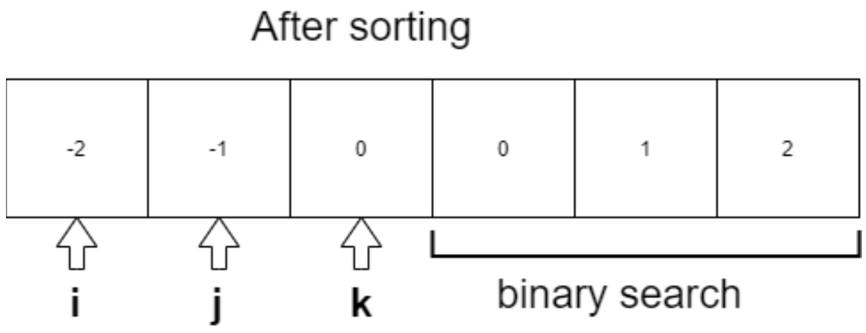
Example 2:

```
Input Format: arr[] = [4,3,3,4,4,2,1,2,1,1], target = 9

Result: [[1,1,3,4],[1,2,2,4],[1,2,3,3]]
```

Solution:
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Using 3 pointers and Binary Search
Intuition:



Approach:

The main idea is to sort the array, and then we can think of searching in the array using the binary search technique. Since we need the 4 numbers which sum up to target. So we will fix three numbers as **nums[i], nums[j] and nums[k]**, and search for the remaining **(target – (nums[i] + nums[j] + nums[k]))** in the array. Since we sorted the array during the start, we can apply **binary search** to search for this value, and if it occurs we can store them. In order to get the unique quads, we use a set data structure to store them.
Code:

```
#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        int n = nums.size();
        sort(nums.begin(),nums.end());
       set<vector<int>> sv;
```

```
        for(int i=0;i<n;i++)
        {
            for(int j=i+1;j<n;j++)
            {
                for(int k=j+1;k<n;k++)
                {
                   int x = (long long)target -
                           (long long)nums[i]-
                           (long long)nums[j]-(long long)nums[k];

            if(binary_search(nums.begin()+k+1,nums.end(),x)){
                            vector<int> v;
                            v.push_back(nums[i]);
                            v.push_back(nums[j]);
                            v.push_back(nums[k]);
                            v.push_back(x);
                            sort(v.begin(),v.end());
                            sv.insert(v);
                        }
                }
            }
        }
        vector<vector<int>> res(sv.begin(),sv.end());
        return res;
    }
};
int main()
{    Solution obj;
    vector<int> v{1,0,-1,0,-2,2};
    vector<vector<int>> sum=obj.fourSum(v,0);
    cout<<"The unique quadruplets are"<<endl;
    for (int i = 0; i < sum.size(); i++) {
        for (int j = 0; j < sum[i].size(); j++)
            cout << sum[i][j] << " ";
        cout << endl;
    }}
```

Output:

```
The unique quadruplets are
-2 -1 1 2
-2 0 0 2
-1 0 0 1
```

Time Complexity: O(**N log N + N^3 logN**)

Reason: Sorting the array takes **N log N** and three nested loops will be taking **N^3** and for binary search, it takes another log N.

Space Complexity: O(M * 4), where M is the number of quads

Solution 2: Optimized Approach

Intuition: In the previous approach we fixed three-pointers and did a binary search to find the fourth. Over here we will fix two-pointers and then find the remaining two elements using two pointer technique as the array will be sorted at first.

Approach: Sort the array, and then fix two pointers, so the remaining sum will be **(target – (nums[i] + nums[j]))**. Now we can fix two pointers, one front, and another back, and easily use a two-pointer to find the remaining two numbers of the quad. In order to avoid duplications, we jump the duplicates, because taking duplicates will result in repeating quads. We can easily jump duplicates, by skipping the same elements by running a loop.

Dry-run: In case you want to see the dry run of this approach, please watch the video at the bottom.
Code:
```
#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& num, int target) {
         vector<vector<int> > res;
        if (num.empty())
            return res;
        int n = num.size();
        sort(num.begin(),num.end());
        for (int i = 0; i < n; i++) {
            int target_3 = target - num[i];
            for (int j = i + 1; j < n; j++) {
                int target_2 = target_3 - num[j];
                int front = j + 1;
                int back = n - 1;
                while(front < back) {
                    int two_sum = num[front] + num[back];
                    if (two_sum < target_2) front++;
                    else if (two_sum > target_2) back--;
                    else {
                        vector<int> quadruplet(4, 0);
                        quadruplet[0] = num[i];
                        quadruplet[1] = num[j];
                        quadruplet[2] = num[front];
                        quadruplet[3] = num[back];
                        res.push_back(quadruplet);
                        // Processing the duplicates of number 3
                        while (front < back && num[front] == quadruplet[2]) ++front;

                        // Processing the duplicates of number 4
                        while (front < back && num[back] == quadruplet[3]) --back;
                    }
                }
                // Processing the duplicates of number 2
                while(j + 1 < n && num[j + 1] == num[j]) ++j;
            }
            // Processing the duplicates of number 1
            while (i + 1 < n && num[i + 1] == num[i]) ++i;
        }
        return res;
    }
};
int main()
{
    Solution obj;
    vector<int> v{1,0,-1,0,-2,2};

    vector<vector<int>> sum=obj.fourSum(v,0);
    cout<<"The unique quadruplets are"<<endl;
    for (int i = 0; i < sum.size(); i++) {
        for (int j = 0; j < sum[i].size(); j++)
            cout << sum[i][j] << " ";
        cout << endl;
    }
}
```

Longest Consecutive Sequence in an Array

Problem Statement: You are given an array of 'N' integers. You need to find the length of the longest sequence which contains the consecutive elements.

Examples:

```
Example 1:

Input: [100, 200, 1, 3, 2, 4]

Output: 4

Explanation: The longest consecutive subsequence is 1, 2, 3, and 4.

Input: [3, 8, 5, 7, 6]

Output: 4

Explanation: The longest consecutive subsequence is 5, 6, 7, and 8.
```

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Brute force

Approach: We can simply sort the array and run a for loop to find the longest consecutive sequence.

Code:

```
#include<bits/stdc++.h>
using namespace std;
int longestConsecutive(vector<int> nums) {
        if(nums.size() == 0 ){
            return 0;
        }
        sort(nums.begin(),nums.end());
        int ans = 1;
        int prev = nums[0];
        int cur = 1;
        for(int i = 1;i < nums.size();i++){
            if(nums[i] == prev+1){
                cur++;
            }
            else if(nums[i] != prev){
                cur = 1;
            }
            prev = nums[i];
            ans = max(ans, cur);
        }
        return ans;
    }
    int main()
    {
        vector<int> arr{100,200,1,2,3,4};
        int lon=longestConsecutive(arr);
        cout<<"The longest consecutive sequence is "<<lon<<endl;
    }
```

Output:

The longest consecutive sequence is 4

Time Complexity: We are first sorting the array which will take O(N * log(N)) time and then we are running a for loop which will take O(N) time. Hence, the overall time complexity will be O(N * log(N)).

Space Complexity: The space complexity for the above approach is O(1) because we are not using any auxiliary space

Optimal Approach

Approach: We will first push all are elements in the HashSet. Then we will run a for loop and check for any number(x) if it is the starting number of the consecutive sequence by checking if the HashSet contains (x-1) or not. If 'x' is the starting number of the consecutive sequence we will keep searching for the numbers y = x+1, x+2, x+3, ….. And stop at the first 'y' which is not present in the HashSet. Using this we can calculate the length of the longest consecutive subsequence.

Code:

```
#include<bits/stdc++.h>
using namespace std;
int longestConsecutive(vector < int > & nums) {
  set < int > hashSet;
  for (int num: nums) {
    hashSet.insert(num);
  }
  int longestStreak = 0;
  for (int num: nums) {
    if (!hashSet.count(num - 1)) {
      int currentNum = num;
      int currentStreak = 1;
      while (hashSet.count(currentNum + 1)) {
        currentNum += 1;
        currentStreak += 1;
      }
      longestStreak = max(longestStreak, currentStreak);
    }
  }
  return longestStreak;
}
int main() {
  vector<int> arr{100,200,1,2,3,4};
  int lon = longestConsecutive(arr);
  cout << "The longest consecutive sequence is " << lon << endl;
}
```

Output:

The longest consecutive sequence is 4

Time Complexity: The time complexity of the above approach is O(N) because we traverse each consecutive subsequence only once.

Space Complexity: The space complexity of the above approach is O(N) because we are maintaining a HashSet.

Length of the longest subarray with zero Sum

Array Part IV

Problem Statement: Given an array containing both positive and negative integers, we have to find the length of the longest subarray with the sum of all elements equal to zero.

Example 1:

```
Input Format: N = 6, array[] = {9, -3, 3, -1, 6, -5}

Result: 5

Explanation: The following subarrays sum to zero:
{-3, 3} , {-1, 6, -5}, {-3, 3, -1, 6, -5}
Since we require the length of the longest subarray, our answer is 5!
```

Example 2:

```
Input Format: N = 8, array[] = {6, -2, 2, -8, 1, 7, 4, -10}

Result: 8

Subarrays with sum 0 : {-2, 2}, {-8, 1, 7}, {-2, 2, -8, 1, 7}, {6, -2, 2, -8, 1, 7, 4,
-10}
Length of longest subarray = 8
```

Example 3:

```
Input Format: N = 3, array[] = {1, 0, -5}

Result: 1

Subarray : {0}
Length of longest subarray = 1
```

Example 4:

```
Input Format: N = 5, array[] = {1, 3, -5, 6, -2}

Result: 0

Subarray: There is no subarray that sums to zero
```

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1 (Naive approach) :

Intuition:

We are required to find the longest subarray that sums to zero. So our initial approach could be to consider all possible subarrays of the given array and check for the subarrays that sum to zero. Among these valid subarrays (sum equal to zero) we'll return the length of the largest subarray by maintaining a variable, say max.

Approach :

1. Initialise a variable max = 0, which stores length of longest subarray with sum 0.
2. Traverse the array from start and initialise a variable sum = 0 which stores sum of subarray starting with current index
3. Traverse from next element of current_index up to end of the array, each time add the element to sum and check if it is equal to 0.
    1. If sum = 0, check if length of subarray so far is > max and if yes update max
4. Now keep adding elements and repeat step 3 a.
5. After the outer loop traverses all elements return max

Time Complexity: O(N^2) as we have two loops for traversal
Space Complexity : O(1) as we aren't using any extra space.
Can this be done in single traversal? Let's check

Solution 2 (Optimised Approach) :
Intuition: Now let's say we know that the sum of subarray(i, j) = S, and we also know that sum of subarray(i, x) = S where i < x < j. We can conclude that the sum of subarray(x+1, j) = 0.
Let us understand the above statement clearly

So in this problem, we'll store the prefix sum of every element, and if we observe that 2 elements have same prefix sum, we can conclude that the 2nd part of this subarray sums to zero
Now let's understand the approach
Approach:

1. First let us initialise a variable say **sum = 0** which stores the sum of elements traversed so far and another variable say **max = 0** which stores the length of longest subarray with sum zero.
2. Declare a HashMap<Integer, Integer> which stores the prefix sum of every element as key and its index as value.
3. Now traverse the array, and add the array element to our sum.

(i) If sum = 0, then we can say that the subarray until the current index has a sum = 0, so we update max with the maximum value of (max, current_index+1)
(ii) If sum is not equal to zero then we check hashmap if we've seen a subarray with this sum before
if HashMap contains sum -> this is where the above-discussed case occurs (subarray with equal sum), so we update our max
else -> Insert (sum, current_index) into hashmap to store prefix sum until current index

4. After traversing the entire array our max variable has the length of longest substring having sum equal to zero, so return max.

NOTE : we do not update the index of a sum if it's seen again because we require the length of the ***longest*** subarray

Dry Run: Let us dry run the algorithm for a better understanding
```
Input : N = 5, array[] = {1, 2, -2, 4, -4}

Output : 5
```
1. Initially sum = 0, max = 0
2. HashMap<Integer, Integer> h = [ ];
3. => **i=0** -> **sum=1**, sum!=0 so check in hashmap if we've seen a subarray with this sum before, in our case hashmap does not contain sum (1), so we add (sum, i) to hashmap.
h = [[1,0]]
=> **i=1** -> **sum=3**, sum!=0 so check in hashmap if we've seen a subarray with this sum before, in our case hashmap does not contain sum (3), so we add (sum, i) to hashmap.
h=[[1,0], [3,1]]
=> **i=2** -> **sum=1**, sum!=0 so check in hashmap if it contains sum, in our case hashmap contains sum (1)
Here we can observe that our current sum is seen before which concludes that it's a zero subarray!
So we now update our max as maximum(max, 2-0) => **max = 2**
h=[[1,0], [3,1]]
=> **i=3** -> **sum=5** , sum!=0 so check in hashmap if it contains sum, in our case hashmap does not contain sum (5), so we add (sum, i) to hashmap.
h=[[1,0], [3,1], [5,3]]
=> **i=4** -> **sum=1**, sum!=0 so check in hashmap if it contains sum, in our case hashmap contains sum (1)
Here we can again observe our above discussed case
So we now update our max as maximum(max, 4-0) => **max = maximum(2, 4) = 4**
h=[[1,0], [3,1], [5,3]]
4. Now that we have traversed our whole array, our answer is max = 4
Subarray = {2, -2, 4, -4}
```
int maxLen(int A[], int n)
{
    // Your code here
    unordered_map<int,int> mpp;
    int maxi = 0;
    int sum = 0;
    for(int i = 0;i<n;i++) {
        sum += A[i];
        if(sum == 0) {
            maxi = i + 1;
        }
        else {
            if(mpp.find(sum) != mpp.end()) {
                maxi = max(maxi, i - mpp[sum]);
            }
            else {
                mpp[sum] = i;
            }
        }
    }

    return maxi;
}
```

Time Complexity: O(N), as we are traversing the array only once
Space Complexity: O(N), in the worst case we would insert all array elements prefix sum into our hashmap

Count the number of subarrays with given xor K

Problem Statement: Given an array of integers A and an integer B. Find the total number of subarrays having bitwise XOR of all elements equal to B.
Examples:

```
Input Format:  A = [4, 2, 2, 6, 4] , B = 6
Result: 4
Explanation: The subarrays having XOR of their elements as 6 are  [4, 2], [4, 2, 2, 6, 4], [2, 2, 6], [6]
Input Format: A = [5, 6, 7, 8, 9], B = 5
Result: 2
Explanation:The subarrays having XOR of their elements as 2 are [5] and [5, 6, 7, 8, 9]
```

Solution:
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute Force

Intuition: The brute force solution is to generate all possible subarrays. For each generated subarray we get the respective XOR and then check if this XOR is equal to B. If it is then we increment the count. In the end, we will get the count of all possible subarrays that have XOR equal to B.
Approach:

1. Generate subarrays: To generate all possible subarrays, we use the same old technique of nested loops. For each value of the outer loop (i loop), the inner loop(j loop) runs from i till the last element. Each iteration of the inner loop gives a new subarray.
2. Maintain XOR: Since each iteration of the inner loop gives a new subarray, we maintain a variable X, in which we keep the XOR of the current subarray.

3. Check and Count: Before moving to the next iteration of the inner loop (that is before going to the next subarray), we check if the current XOR is equal to B, if it is then we increment the counter (counter also has to be maintained).

Dry Run:

Code:

```
#include<bits/stdc++.h>
using namespace std;
class Solution{
    public:
    int solve(vector<int> &A, int B) {
    long long c=0;
    for(int i=0;i<A.size();i++){
        int current_xor = 0;
        for(int j=i;j<A.size();j++){
            current_xor^=A[j];
            if(current_xor==B) c++;
        }
    }
    return c;
}
};

int main()
{   vector<int> A{ 4,2,2,6,4 };
    Solution obj;
    int totalCount= obj.solve(A,6);
    cout<<"The total number of subarrays having a given XOR k is "<<totalCount<<endl;
}
```

Output:

The total number of subarrays having a given XOR k is 4

Time Complexity: O(N2)

Space Complexity: O(1)

Solution 2: Prefix xor and map

Intuition: The main idea is to observe the prefix xor of the array. Prefix Xor is just another array, where each index contains XOR of all elements of the original array starting from index 0 up to that index. In other words
prefix_xor[i] = XOR(a[0], a[1], a[2],……,a[I])
Once we have made the prefix xor array, we observe a fact:
P = xor(a[0], a[1], a[2],……, a[q], a[q+1],….., a[p])
Q = xor(a[0], a[1], a[2],……, a[q])
Therefore,
P^Q = xor(a[q+1],….., a[p]) = M
So, now we understand that from the prefix xor array when we XOR two elements at different indices we get the xor of the elements (in the original array) that were between those indices.
Let's say we did XOR(P, B) and we got Q (B is the integer given in question). What does this mean?
This means that the subarray between q and p is having xor = B. To understand this we just use simple equations:
P^B = Q
=> P^B^P = Q^P
=> B = Q^P
And we already know by fact 1 that Q^P will give xor of all elements between q and p. Therefore, the subarray between q and p has xor = B.
Suppose we did XOR(P, B) and we got Q (B is the integer given in question). But there is more than one Q before p. In this case, there are two subarrays that have XOR = B. Subarrays between q1 to p and q2 to p.

IMP NOTE: although we talk about prefix xor “array”, it should be noted that at a time we need only one element of this array. Hence, we can just use a variable to maintain the prefix xor.
Approach: We need to traverse the array while we maintain variables for current_perfix_xor, counter, and also a map to keep track of visited xors. This map will maintain the frequency count of all previous visited current_prefix_xor values. If for any index current_prefix_xor is equal to B, we increment the counter. If for any index we find that Z is present in the visited map, we increment the counter by visited[Z] (Z=current_prefi_xor^B). At every index, we insert the current_prefix_xor into the visited map with its frequency.
Code:
```
#include<bits/stdc++.h>
using namespace std;
class Solution{
    public:
    int solve(vector<int> &A, int B) {
    unordered_map<int,int>visited;
    int cpx = 0;
    long long c=0;
    for(int i=0;i<A.size();i++){
        cpx^=A[i];
        if(cpx==B) c++;
        int h = cpx^B;
        if(visited.find(h)!=visited.end()){
            c=c+visited[h];
        }
        visited[cpx]++;
    }
    return c;
}
};


int main()
{   vector<int> A{ 4,2,2,6,4 };
    Solution obj;
    int totalCount= obj.solve(A,6);
    cout<<"The total number of subarrays having a given XOR k is "<<totalCount<<endl;
}
```
Output:
```
The total number of subarrays having a given XOR k is 4
```
Time Complexity: O(N)
Space Complexity: O(N)
NOTE: the complexity of worst-case searching for an unordered_map can go up to O(N), hence it is safer to use ordered_map. But if we use ordered_map then the time complexity will be O(N logN). Space complexity will be the same in both cases.

Length of Longest Substring without any Repeating Character

Problem Statement: Given a String, find the length of longest substring without any repeating character.
Examples:
`Example 1:`

Input: s = ”abcabcbb”

Output: 3

Explanation: The answer is abc with length of 3.

Example 2:

Input: s = ”bbbbb”

Output: 1

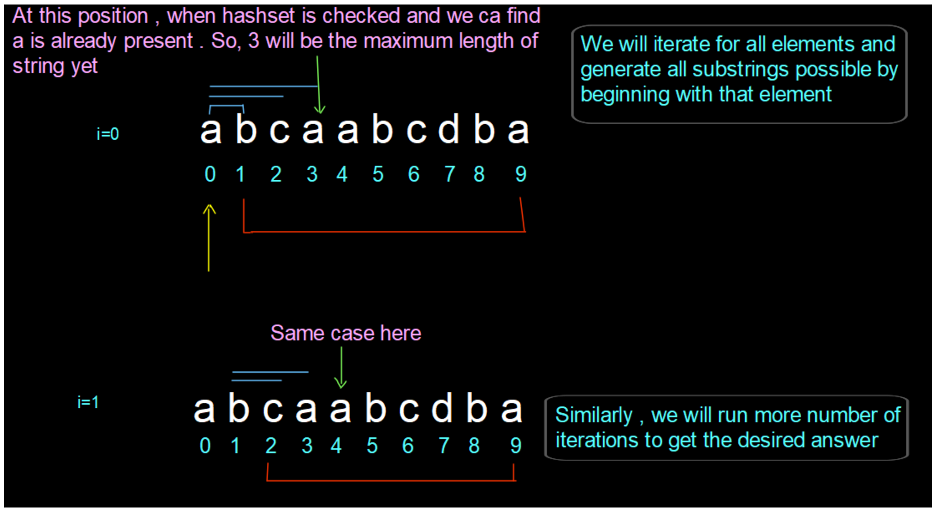
Explanation: The answer is b with length of 1 units.

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute force Approach

Approach: This approach consists of taking the two loops one for traversing the string and another loop – nested loop for finding different substrings and after that, we will check for all substrings one by one and check for each and every element if the element is not found then we will store that element in HashSet otherwise we will break from the loop and count it.


At this position , when hashset is checked and we ca find a is already present . So, 3 will be the maximum length of string yet
We will iterate for all elements and generate all substrings possible by beginning with that element
i=0
a b c a a b c d b a
0 1 2 3 4 5 6 7 8 9
Same case here
i=1
a b c a a b c d b a
0 1 2 3 4 5 6 7 8 9
Similarly , we will run more number of iterations to get the desired answer

Code:

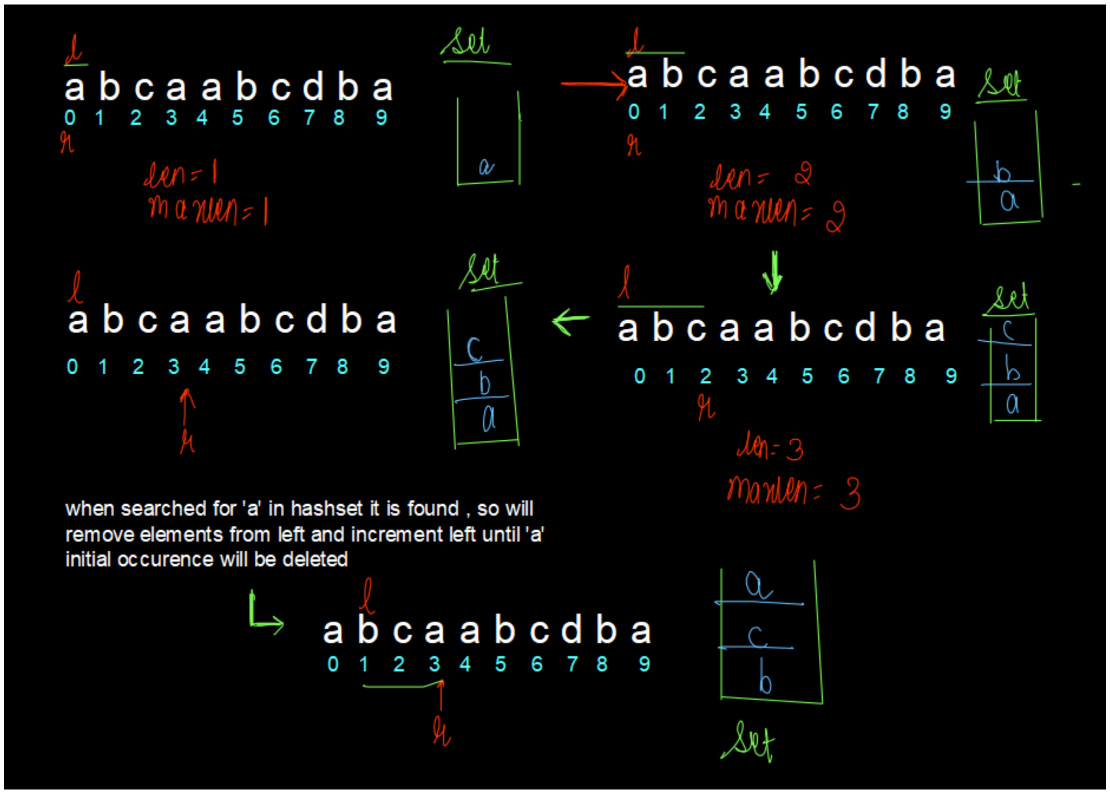
```
#include<bits/stdc++.h>
using namespace std;
int solve(string str) {
  int maxans = INT_MIN;
  for (int i = 0; i < str.length(); i++) // outer loop for traversing the string
  {
    unordered_set < int > set;
    for (int j = i; j < str.length(); j++) // nested loop for getting different string
starting with str[i]
    {
      if (set.find(str[j]) != set.end()) // if element if found so mark it as ans and
break from the loop
      {
        maxans = max(maxans, j - i);
        break;
      }
      set.insert(str[j]);
    }
  }
  return maxans;
}

int main() {
  string str = "takeUforward";
  cout << "The length of the longest substring without repeating characters is " <<
  solve(str);
  return 0;
}
```

Output: The length of the longest substring without repeating characters is 9
Time Complexity: O(N2)
Space Complexity: O(N) where N is the size of HashSet taken for storing the elements

Solution 2: Optimised Approach 1

Approach: We will have two pointers left and right. Pointer 'left' is used for maintaining the starting point of substring while 'right' will maintain the endpoint of the substring.' right' pointer will move forward and check for the duplicate occurrence of the current element if found then 'left' pointer will be shifted ahead so as to delete the duplicate elements.


l
a b c a a b c d b a
0 1 2 3 4 5 6 7 8 9
r
len=1
maxlen=1
set
a
len=2
maxlen=2
b
len=3
maxlen=3
c
when searched for 'a' in hashset it is found , so will remove elements from left and increment left until 'a' initial occurence will be deleted

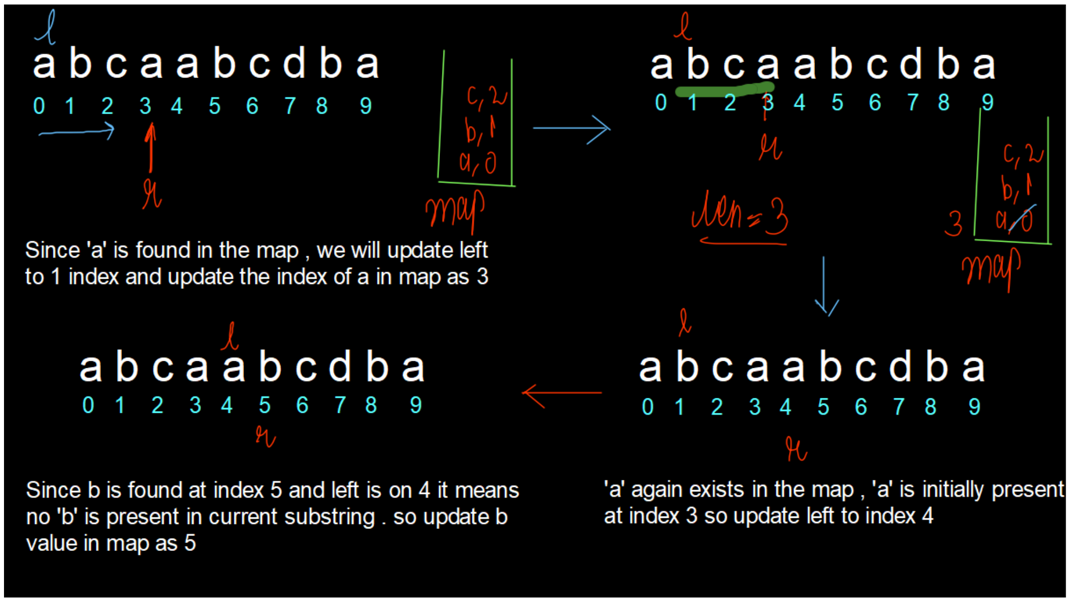
Code:

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

```
int solve(string str) {
  int maxans = INT_MIN;
  unordered_set < int > set;
  int l = 0;
  for (int r = 0; r < str.length(); r++) // outer loop for traversing the string
  {
    if (set.find(str[r]) != set.end()) //if duplicate element is found
    {
      while (l < r && set.find(str[r]) != set.end()) {
        set.erase(str[l]);
        l++;
      }    }
    set.insert(str[r]);
    maxans = max(maxans, r - l + 1);
  }  return maxans;
}
int main() {
  string str = "sahilshaikh";
  cout << "The length of the longest substring without repeating characters is " <<
  solve(str);
  return 0;
}
```

Output: The length of the longest substring without repeating characters is 9
Time Complexity: O(2*N) (sometimes left and right both have to travel complete array)
Space Complexity: O(N) where N is the size of HashSet taken for storing the elements

Solution 3: Optimised Approach 2
Approach: In this approach, we will make a map that will take care of counting the elements and maintaining the frequency of each and every element as a unity by taking the latest index of every element.



l
a b c a a b c d b a
0 1 2 3 4 5 6 7 8 9
r
c, 2
b, 1
a, 0
map
Since 'a' is found in the map , we will update left to 1 index and update the index of a in map as 3
l
a b c a a b c d b a
0 1 2 3 4 5 6 7 8 9
r
len=3
c, 2
b, 1
3 a, 0
map
l
a b c a a b c d b a
0 1 2 3 4 5 6 7 8 9
r
'a' again exists in the map , 'a' is initially present at index 3 so update left to index 4
l
a b c a a b c d b a
0 1 2 3 4 5 6 7 8 9
r
Since b is found at index 5 and left is on 4 it means no 'b' is present in current substring . so update b value in map as 5

Code:

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:
    int lengthofLongestSubstring(string s) {
      vector < int > mpp(256, -1);
      int left = 0, right = 0;
      int n = s.size();
      int len = 0;
      while (right < n) {
        if (mpp[s[right]] != -1)
          left = max(mpp[s[right]] + 1, left);
        mpp[s[right]] = right;
        len = max(len, right - left + 1);
        right++;
      }
      return len;
    }
};
int main() {
  string str = "takeUforward";
  Solution obj;
  cout << "The length of the longest substring without repeating characters is " <<
obj.lengthofLongestSubstring(str);
  return 0;
}
```

Output: The length of the longest substring without repeating characters is 9
Time Complexity: O(N)
Space Complexity: O(N) where N represents the size of HashSet where we are storing our elements