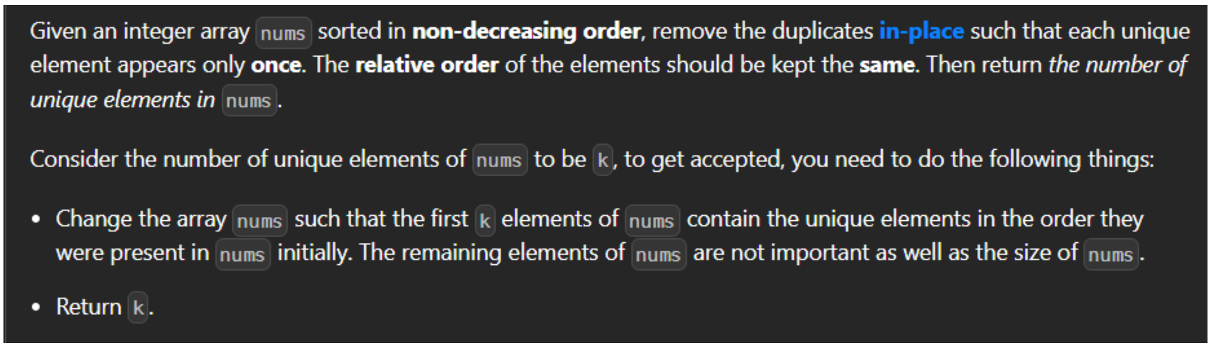
Two Pointers-Sliding window

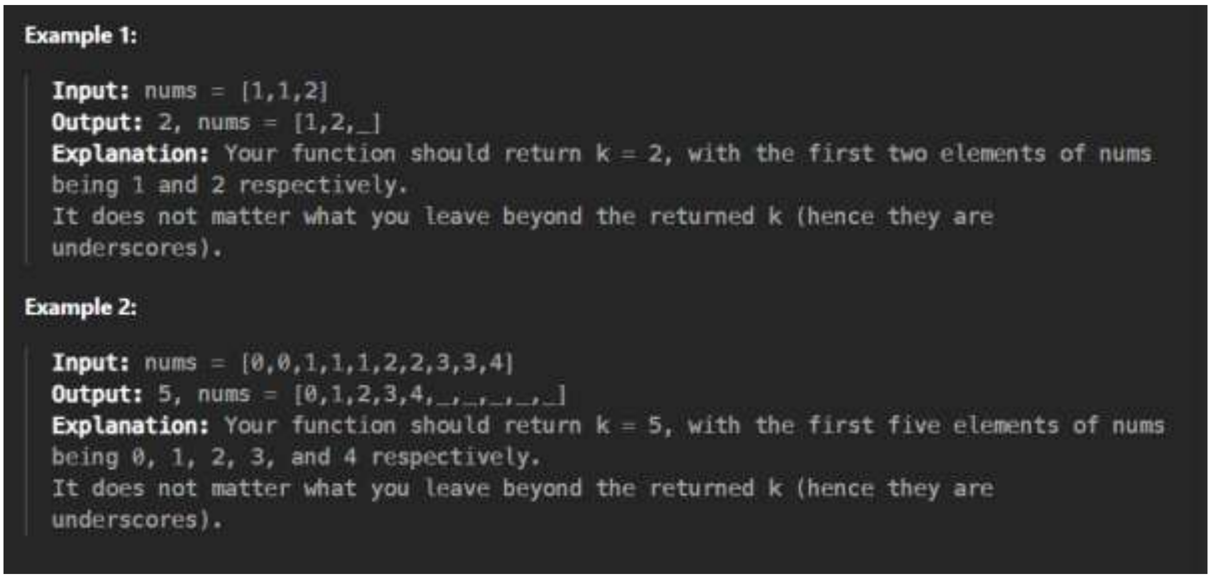
1.Remove duplicate from sorted array



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`.

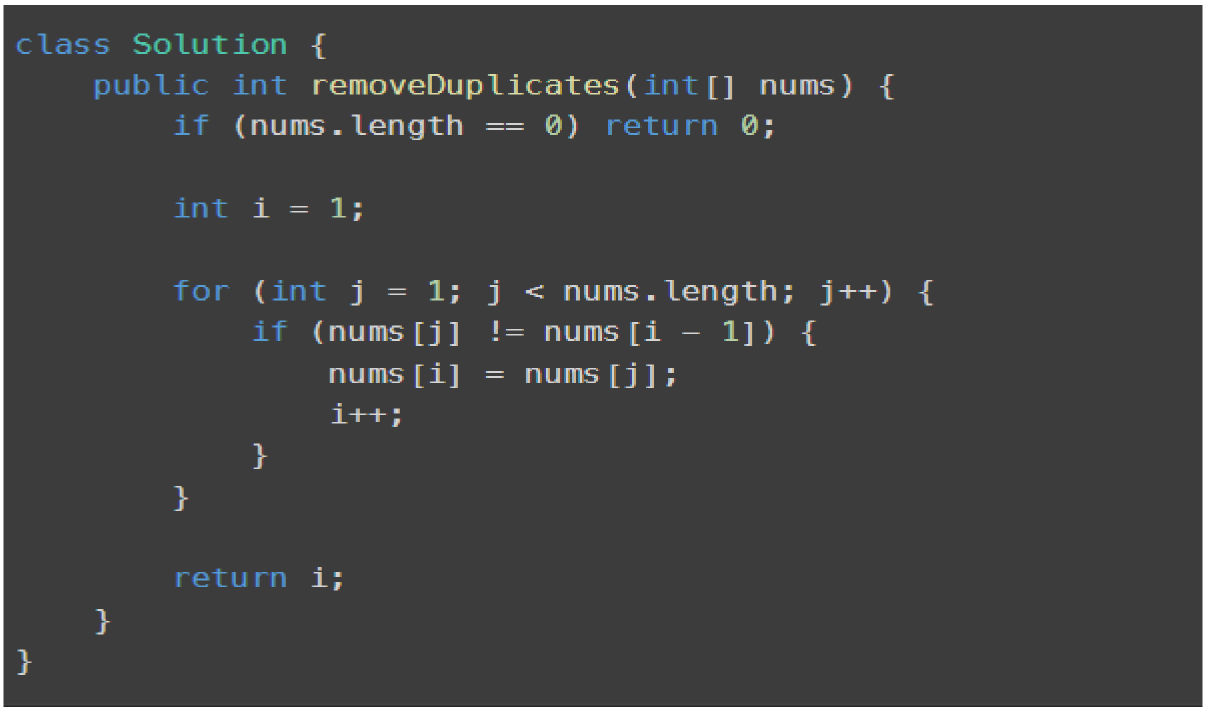


Example 1:

```
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).
```

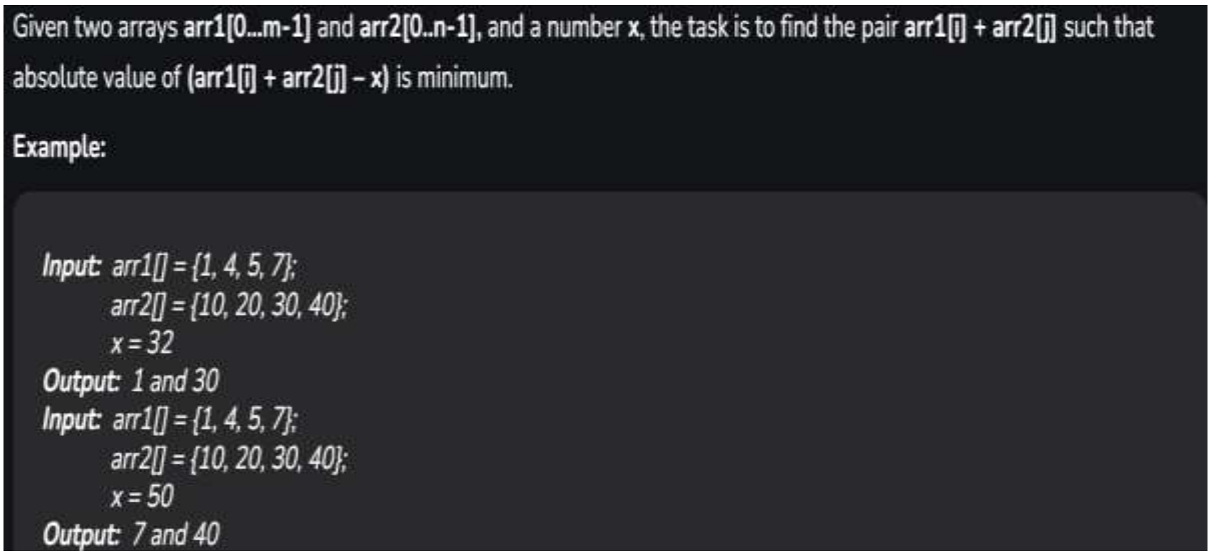
Example 2:

```
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).
```



```
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;

        int i = 1;

        for (int j = 1; j < nums.length; j++) {
            if (nums[j] != nums[i - 1]) {
                nums[i] = nums[j];
                i++;
            }
        }

        return i;
    }
}
```

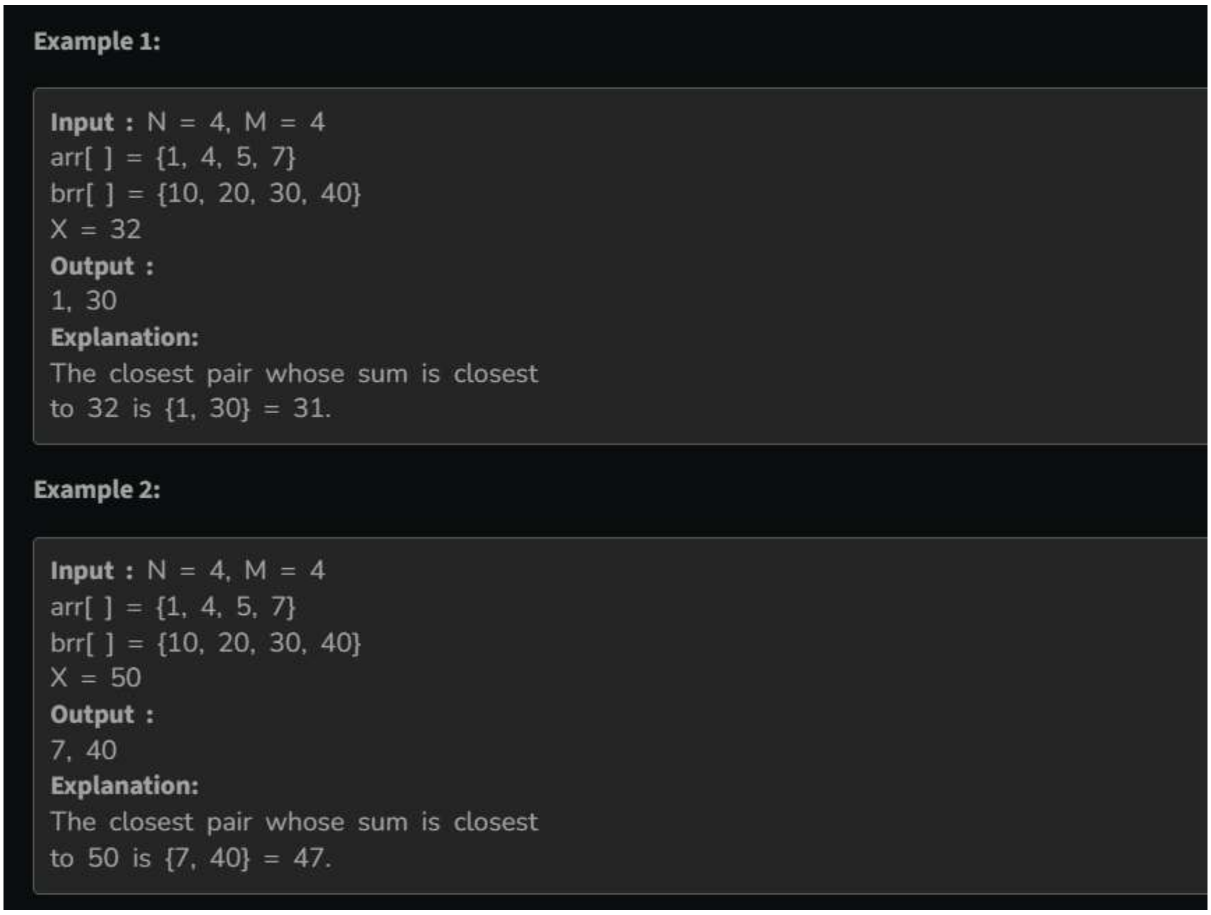
2.Find the closest pair from two sorted array



Given two arrays **arr1[0...m-1]** and **arr2[0..n-1]**, and a number **x**, the task is to find the pair **arr1[i] + arr2[j]** such that absolute value of **(arr1[i] + arr2[j] – x)** is minimum.

Example:

```
Input:  arr1[] = {1, 4, 5, 7};
        arr2[] = {10, 20, 30, 40};
        x = 32
Output:  1 and 30
Input:  arr1[] = {1, 4, 5, 7};
        arr2[] = {10, 20, 30, 40};
        x = 50
Output:  7 and 40
```

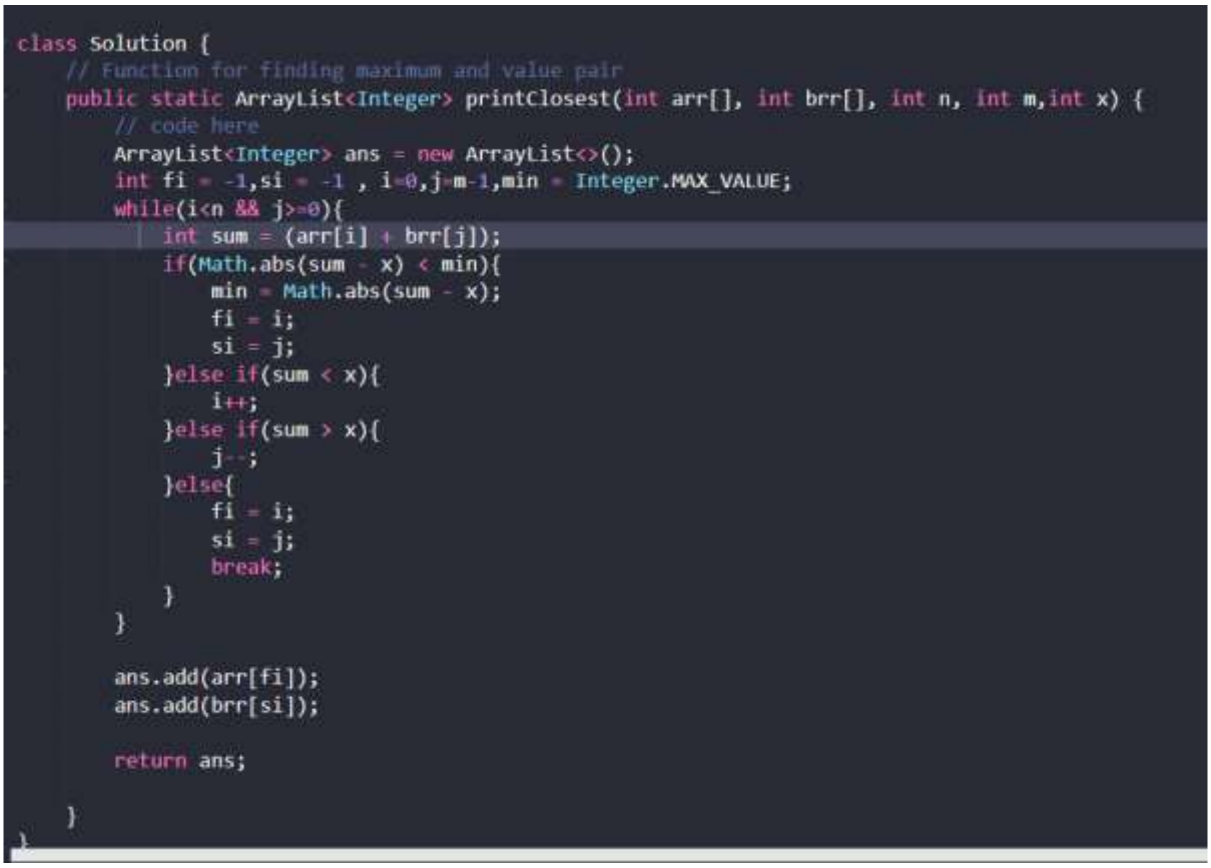


Example 1:

```
Input : N = 4, M = 4
arr[ ] = {1, 4, 5, 7}
brr[ ] = {10, 20, 30, 40}
X = 32
Output :
1, 30
Explanation:
The closest pair whose sum is closest
to 32 is {1, 30} = 31.
```

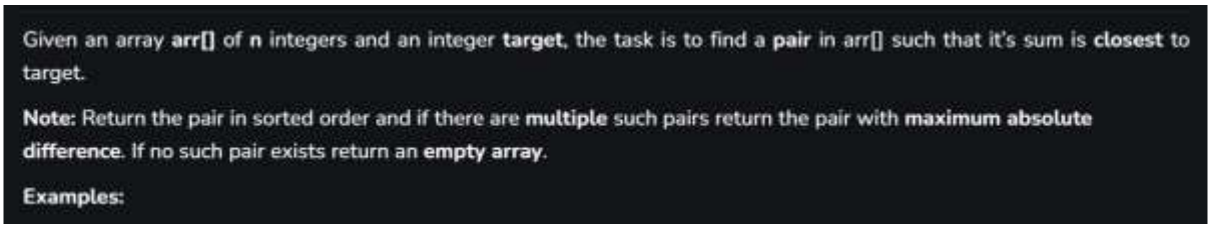
Example 2:

```
Input : N = 4, M = 4
arr[ ] = {1, 4, 5, 7}
brr[ ] = {10, 20, 30, 40}
X = 50
Output :
7, 40
Explanation:
The closest pair whose sum is closest
to 50 is {7, 40} = 47.
```



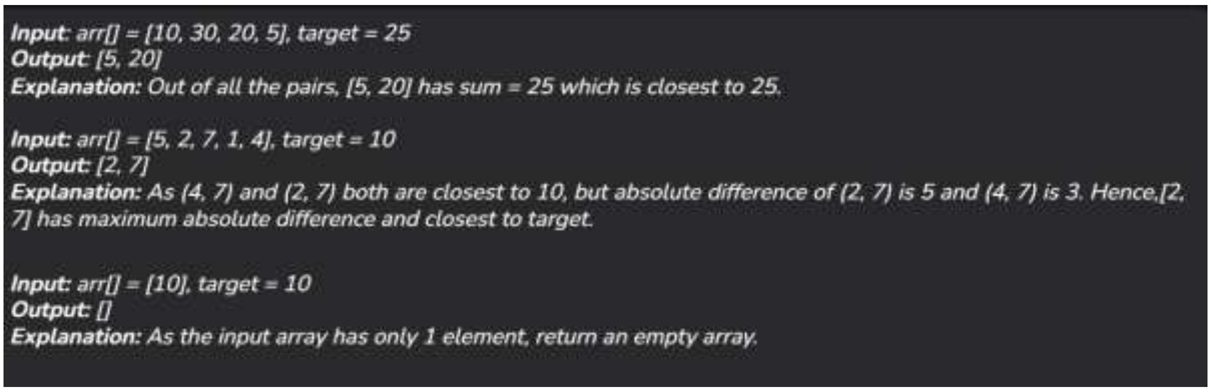
```java
class Solution {
    // Function for finding maximum and value pair
    public static ArrayList<Integer> printClosest(int arr[], int brr[], int n, int m,int x) {
        // code here
        ArrayList<Integer> ans = new ArrayList<>();
        int fi = -1,si = -1 , i=0,j=m-1,min = Integer.MAX_VALUE;
        while(i<n && j>=0){
            int sum = (arr[i] + brr[j]);
            if(Math.abs(sum - x) < min){
                min = Math.abs(sum - x);
                fi = i;
                si = j;
            }else if(sum < x){
                i++;
            }else if(sum > x){
                j--;
            }else{
                fi = i;
                si = j;
                break;
            }
        }

        ans.add(arr[fi]);
        ans.add(brr[si]);

        return ans;

    }
}
```

3. 2 Sum –pair sum closest to target using Binary Search


Given an array **arr[]** of **n** integers and an integer **target**, the task is to find a **pair** in arr[] such that it's sum is **closest** to target.

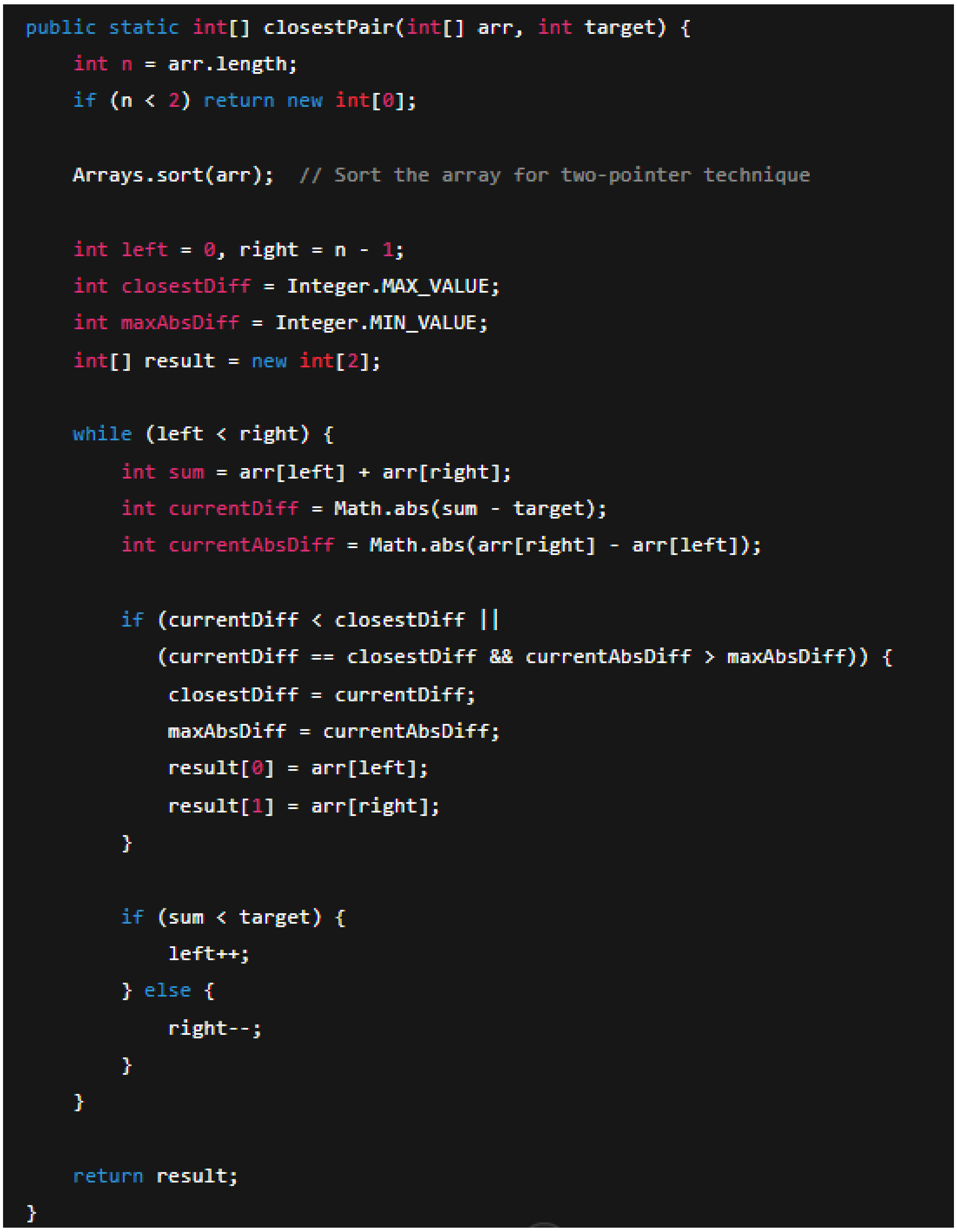
Note: Return the pair in sorted order and if there are **multiple** such pairs return the pair with **maximum absolute difference**. If no such pair exists return an **empty array**.

Examples:


Input: *arr[] = [10, 30, 20, 5], target = 25*
Output: *[5, 20]*
Explanation: *Out of all the pairs, [5, 20] has sum = 25 which is closest to 25.*

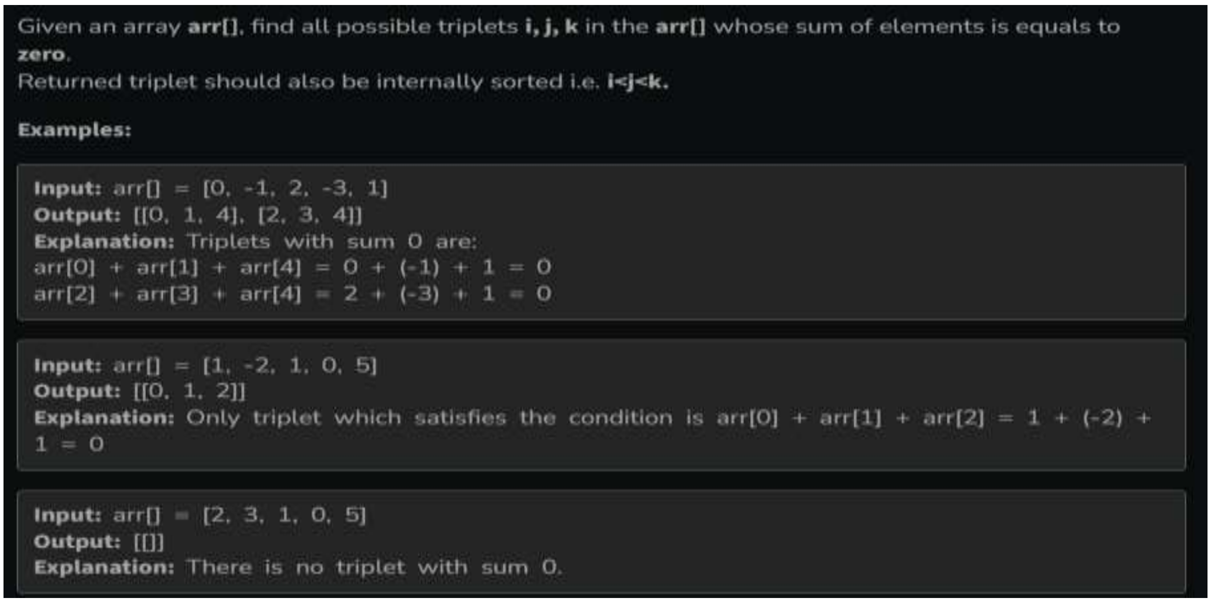
Input: *arr[] = [5, 2, 7, 1, 4], target = 10*
Output: *[2, 7]*
Explanation: *As (4, 7) and (2, 7) both are closest to 10, but absolute difference of (2, 7) is 5 and (4, 7) is 3. Hence,[2, 7] has maximum absolute difference and closest to target.*

Input: *arr[] = [10], target = 10*
Output: *[]*
Explanation: *As the input array has only 1 element, return an empty array.*



```java
public static int[] closestPair(int[] arr, int target) {
    int n = arr.length;
    if (n < 2) return new int[0];

    Arrays.sort(arr);  // Sort the array for two-pointer technique

    int left = 0, right = n - 1;
    int closestDiff = Integer.MAX_VALUE;
    int maxAbsDiff = Integer.MIN_VALUE;
    int[] result = new int[2];

    while (left < right) {
        int sum = arr[left] + arr[right];
        int currentDiff = Math.abs(sum - target);
        int currentAbsDiff = Math.abs(arr[right] - arr[left]);

        if (currentDiff < closestDiff ||
           (currentDiff == closestDiff && currentAbsDiff > maxAbsDiff)) {
            closestDiff = currentDiff;
            maxAbsDiff = currentAbsDiff;
            result[0] = arr[left];
            result[1] = arr[right];
        }

        if (sum < target) {
            left++;
        } else {
            right--;
        }
    }

    return result;
}
```

4. 3-Sum ,Find all triplet with zero sum

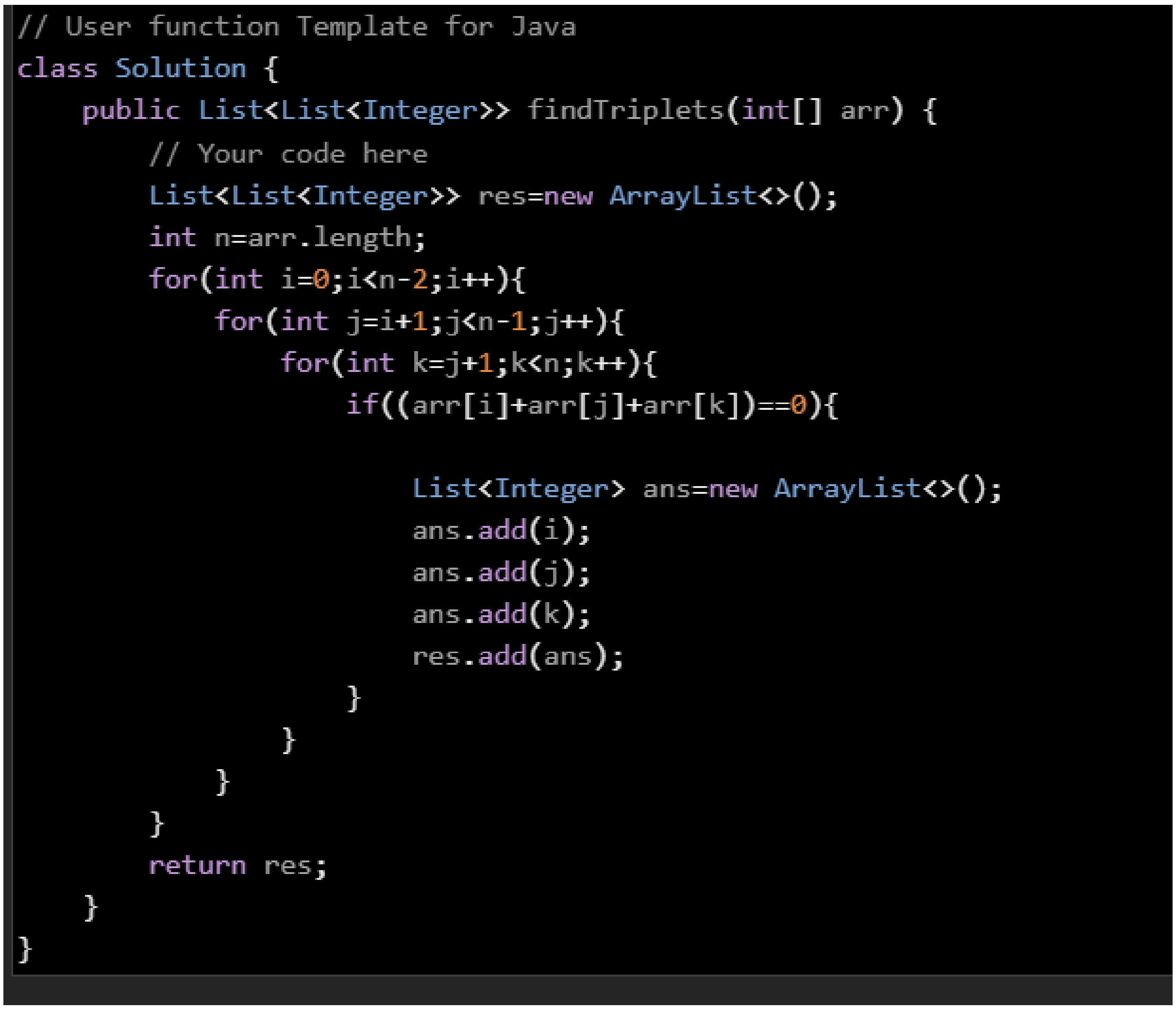

Given an array **arr[]**, find all possible triplets **i, j, k** in the **arr[]** whose sum of elements is equals to **zero**.
Returned triplet should also be internally sorted i.e. **i<j<k.**

Examples:

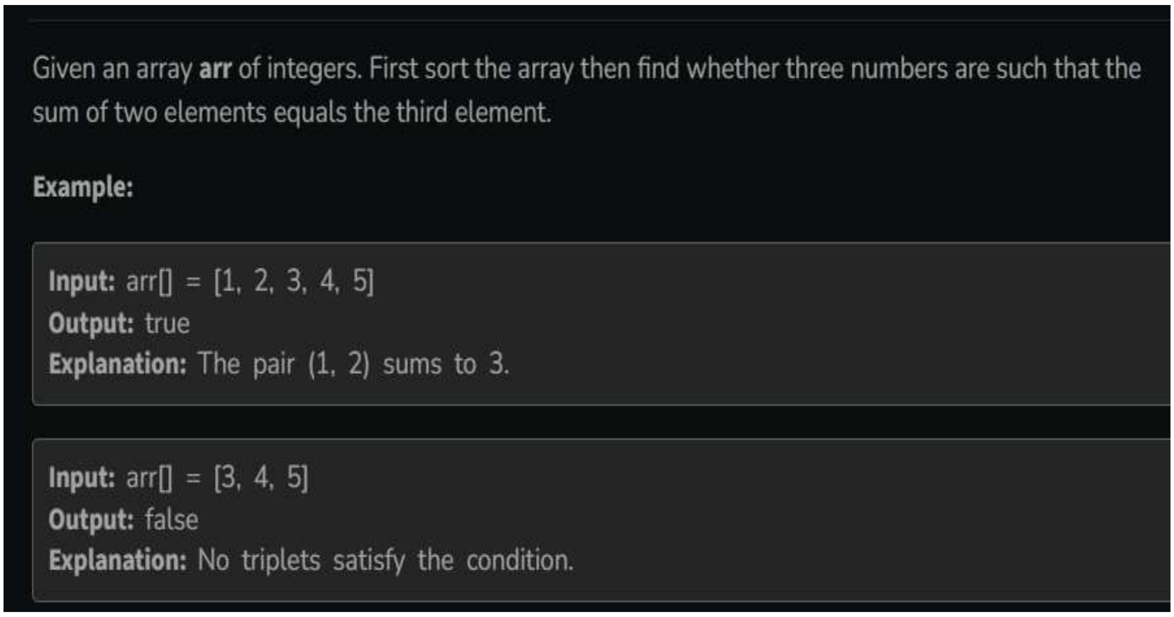
```
Input: arr[] = [0, -1, 2, -3, 1]
Output: [[0, 1, 4], [2, 3, 4]]
Explanation: Triplets with sum 0 are:
arr[0] + arr[1] + arr[4] = 0 + (-1) + 1 = 0
arr[2] + arr[3] + arr[4] = 2 + (-3) + 1 = 0
```

```
Input: arr[] = [1, -2, 1, 0, 5]
Output: [[0, 1, 2]]
Explanation: Only triplet which satisfies the condition is arr[0] + arr[1] + arr[2] = 1 + (-2) +
1 = 0
```

```
Input: arr[] = [2, 3, 1, 0, 5]
Output: [[]]
Explanation: There is no triplet with sum 0.
```


```java
// User function Template for Java
class Solution {
    public List<List<Integer>> findTriplets(int[] arr) {
        // Your code here
        List<List<Integer>> res=new ArrayList<>();
        int n=arr.length;
        for(int i=0;i<n-2;i++){
            for(int j=i+1;j<n-1;j++){
                for(int k=j+1;k<n;k++){
                    if((arr[i]+arr[j]+arr[k])==0){

                        List<Integer> ans=new ArrayList<>();
                        ans.add(i);
                        ans.add(j);
                        ans.add(k);
                        res.add(ans);
                    }
                }
            }
        }
        return res;
    }
}
```

5.Find the triplet such that sum of two equals to third element(Triplet Family)

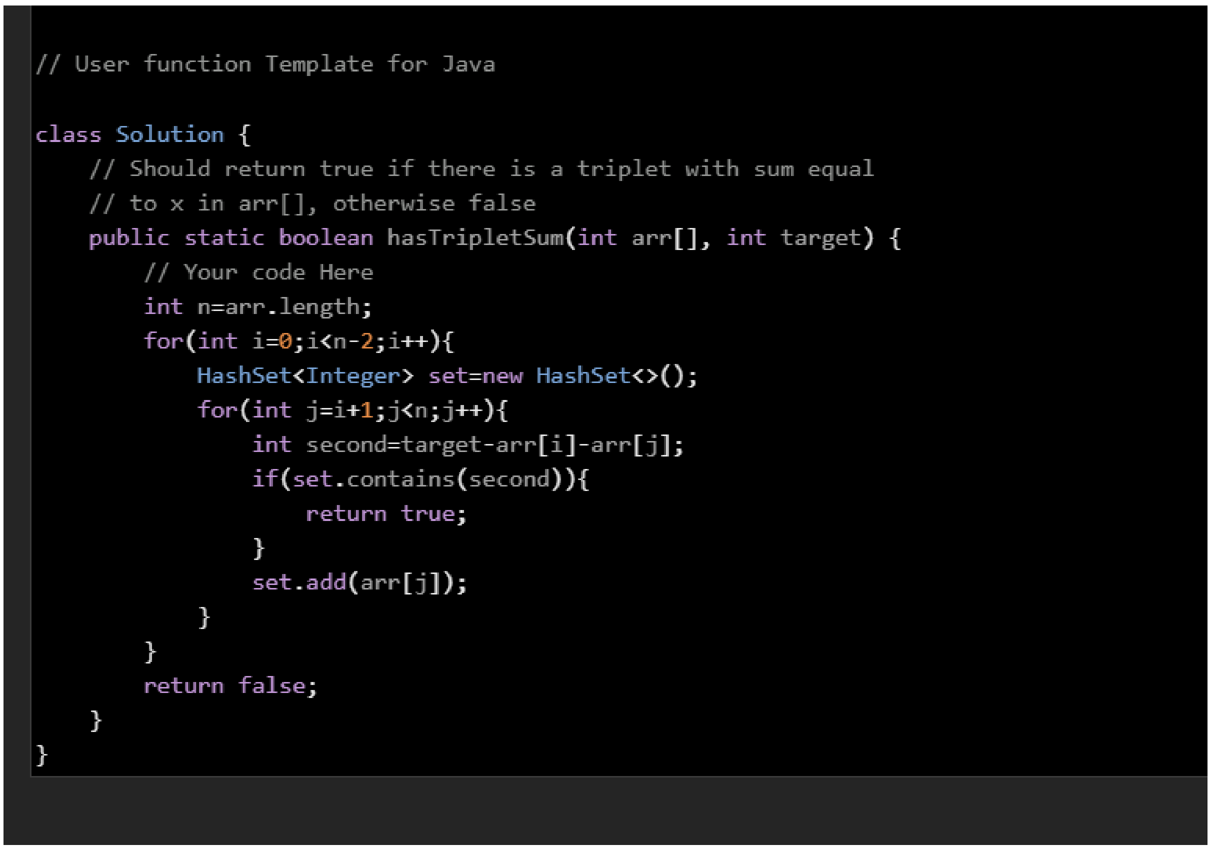


Given an array **arr** of integers. First sort the array then find whether three numbers are such that the sum of two elements equals the third element.

Example:

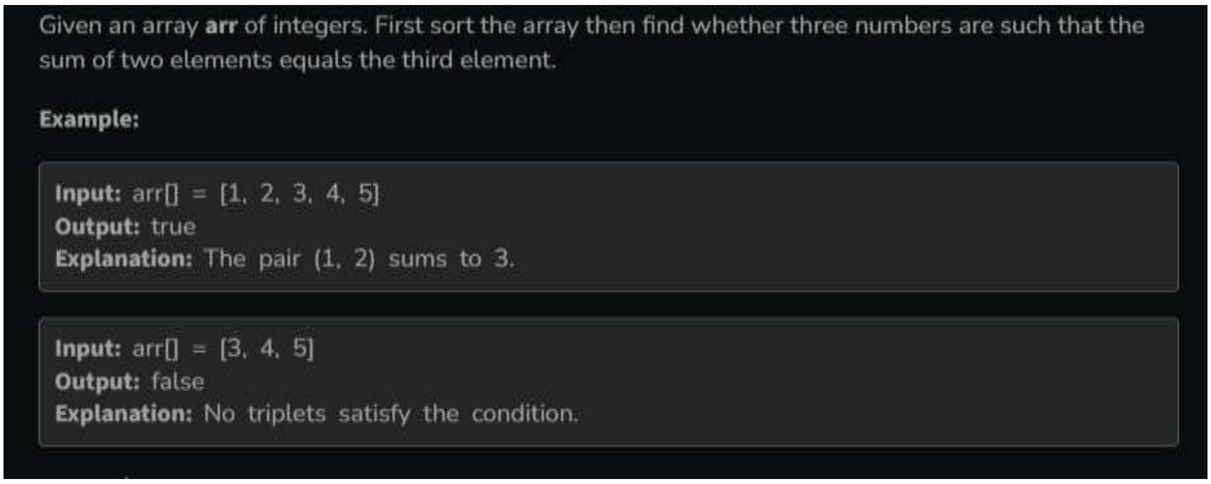
Input: arr[] = [1, 2, 3, 4, 5]
Output: true
Explanation: The pair (1, 2) sums to 3.

Input: arr[] = [3, 4, 5]
Output: false
Explanation: No triplets satisfy the condition.



```java
// User function Template for Java

class Solution {
    // Should return true if there is a triplet with sum equal
    // to x in arr[], otherwise false
    public static boolean hasTripletSum(int arr[], int target) {
        // Your code Here
        int n=arr.length;
        for(int i=0;i<n-2;i++){
            HashSet<Integer> set=new HashSet<>();
            for(int j=i+1;j<n;j++){
                int second=target-arr[i]-arr[j];
                if(set.contains(second)){
                    return true;
                }
                set.add(arr[j]);
            }
        }
        return false;
    }
}
```

6.Find a triplet such that sum of two equals to third element’

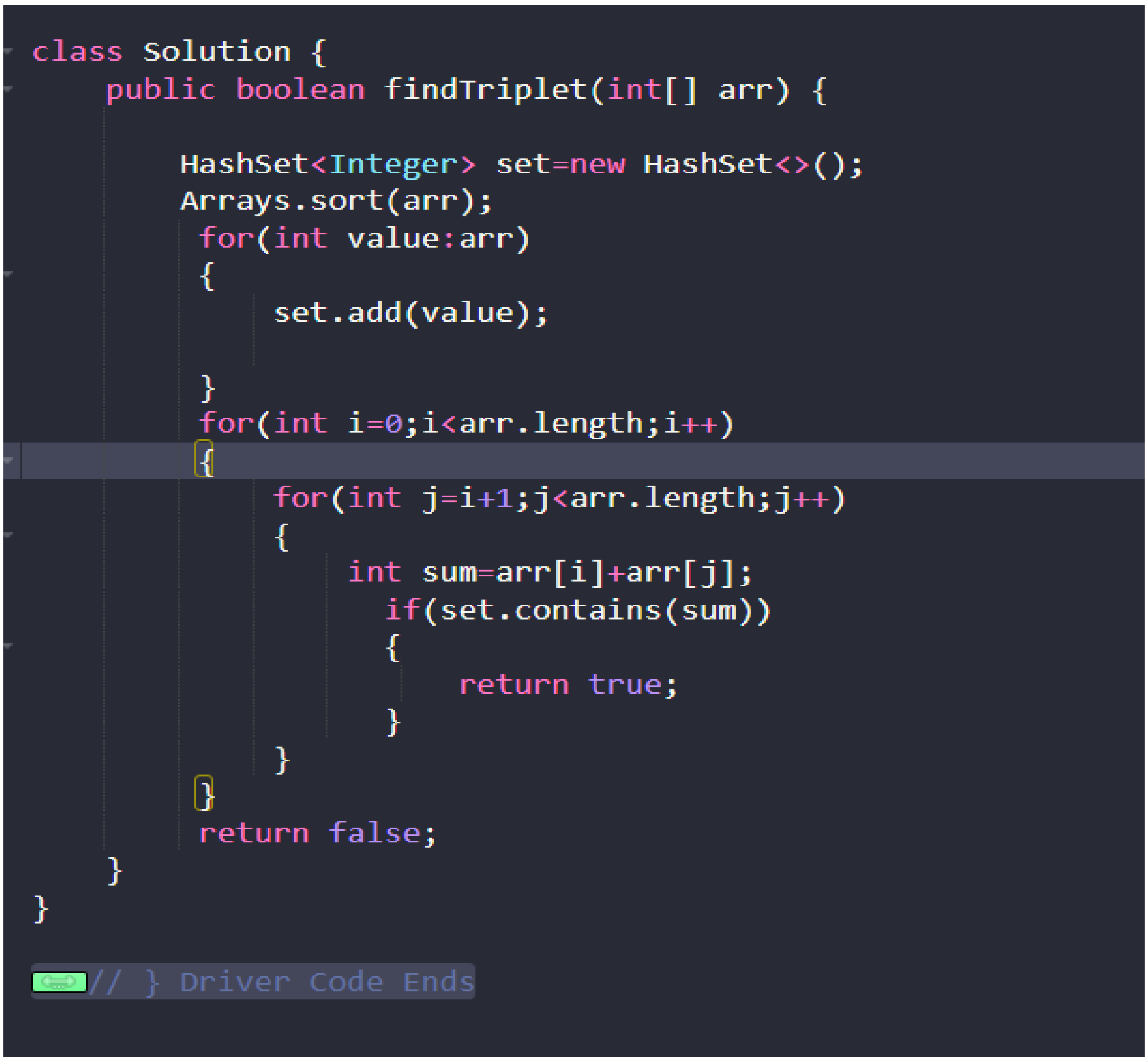


Given an array **arr** of integers. First sort the array then find whether three numbers are such that the sum of two elements equals the third element.

Example:

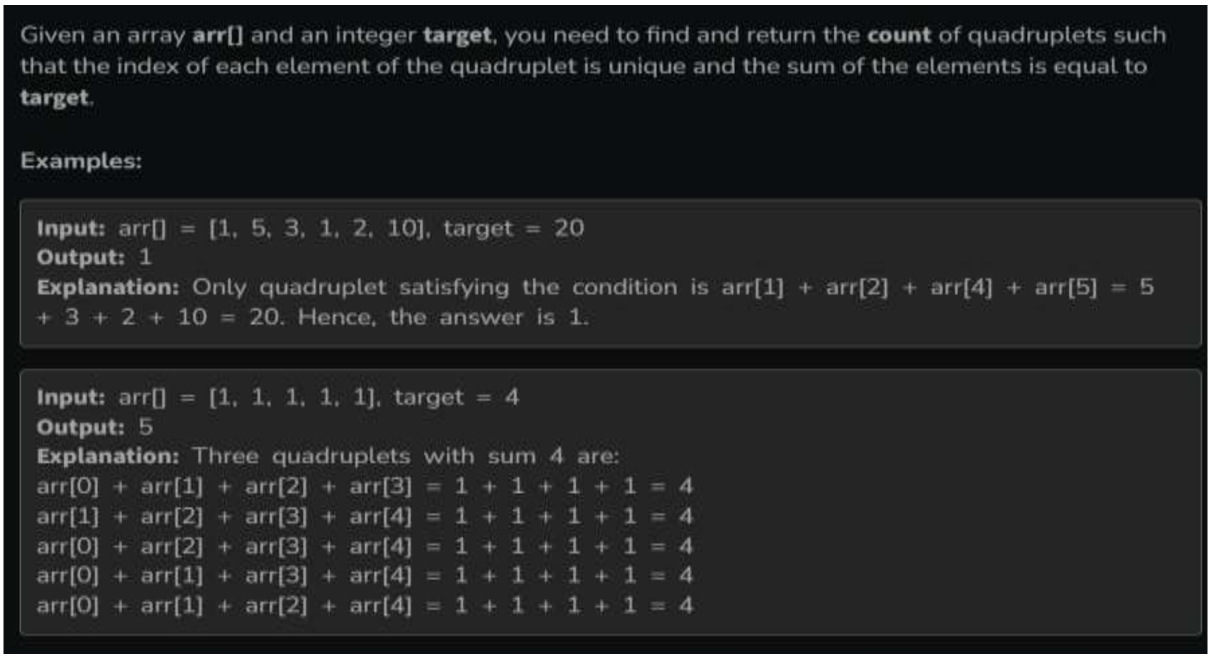
```
Input: arr[] = [1, 2, 3, 4, 5]
Output: true
Explanation: The pair (1, 2) sums to 3.
```

```
Input: arr[] = [3, 4, 5]
Output: false
Explanation: No triplets satisfy the condition.
```



```java
class Solution {
    public boolean findTriplet(int[] arr) {
        
        HashSet<Integer> set=new HashSet<>();
        Arrays.sort(arr);
         for(int value:arr)
         {
             set.add(value);
             
         }
         for(int i=0;i<arr.length;i++)
         {
             for(int j=i+1;j<arr.length;j++)
             {
                 int sum=arr[i]+arr[j];
                   if(set.contains(sum))
                   {
                       return true;
                   }
             }
         }
         return false;
    }
}

// } Driver Code Ends
```

7. 4sum –Count quadruplet with given sum

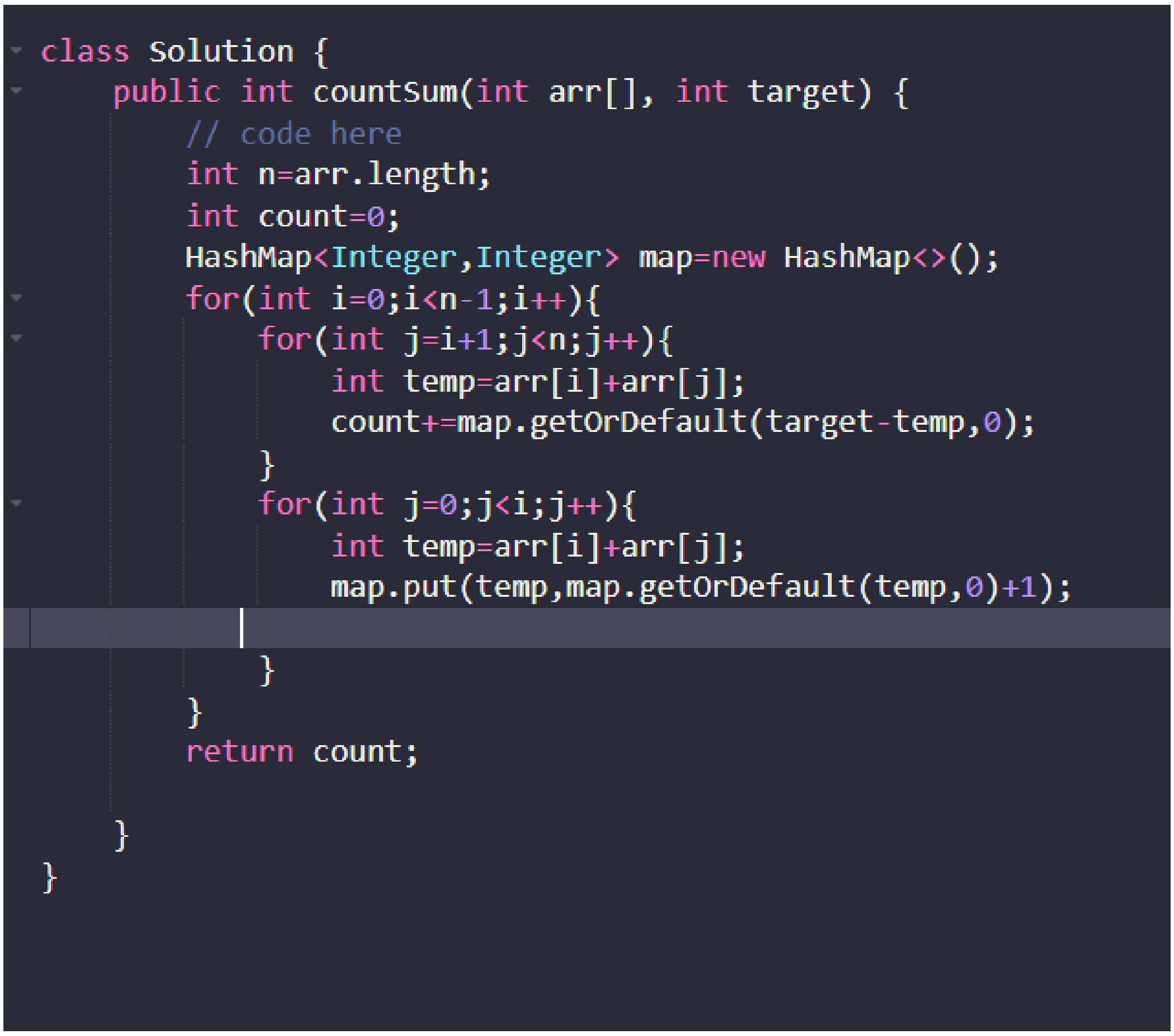


Given an array **arr[]** and an integer **target**, you need to find and return the **count** of quadruplets such that the index of each element of the quadruplet is unique and the sum of the elements is equal to **target**.

Examples:

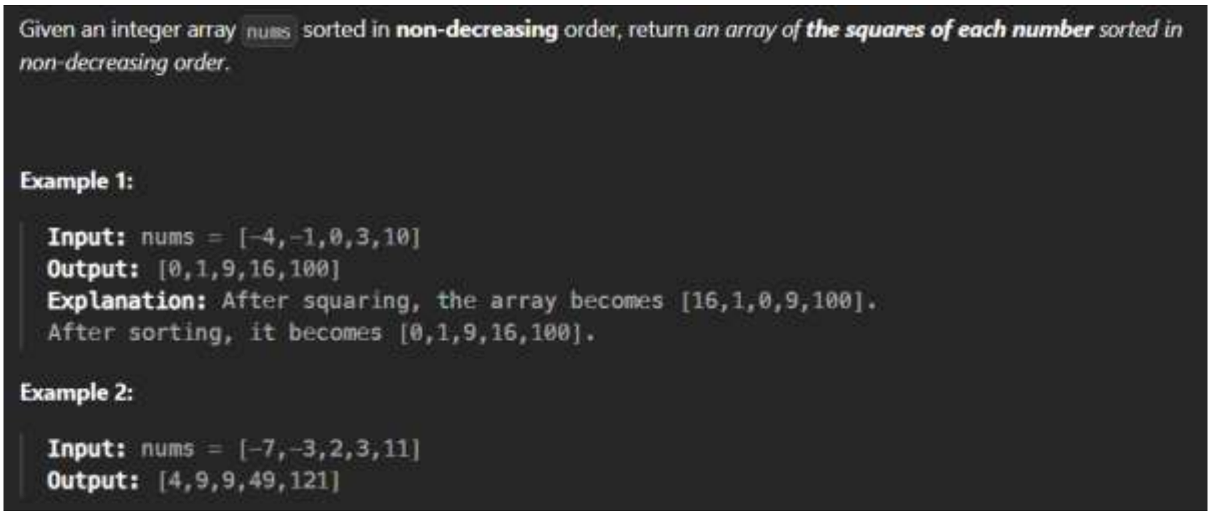
Input: arr[] = [1, 5, 3, 1, 2, 10], target = 20
Output: 1
Explanation: Only quadruplet satisfying the condition is arr[1] + arr[2] + arr[4] + arr[5] = 5 + 3 + 2 + 10 = 20. Hence, the answer is 1.

Input: arr[] = [1, 1, 1, 1, 1], target = 4
Output: 5
Explanation: Three quadruplets with sum 4 are:
arr[0] + arr[1] + arr[2] + arr[3] = 1 + 1 + 1 + 1 = 4
arr[1] + arr[2] + arr[3] + arr[4] = 1 + 1 + 1 + 1 = 4
arr[0] + arr[2] + arr[3] + arr[4] = 1 + 1 + 1 + 1 = 4
arr[0] + arr[1] + arr[3] + arr[4] = 1 + 1 + 1 + 1 = 4
arr[0] + arr[1] + arr[2] + arr[4] = 1 + 1 + 1 + 1 = 4



```
class Solution {
    public int countSum(int arr[], int target) {
        // code here
        int n=arr.length;
        int count=0;
        HashMap<Integer,Integer> map=new HashMap<>();
        for(int i=0;i<n-1;i++){
            for(int j=i+1;j<n;j++){
                int temp=arr[i]+arr[j];
                count+=map.getOrDefault(target-temp,0);
            }
            for(int j=0;j<i;j++){
                int temp=arr[i]+arr[j];
                map.put(temp,map.getOrDefault(temp,0)+1);

            }
        }
        return count;

    }
}
```

8.Squares of a sorted array



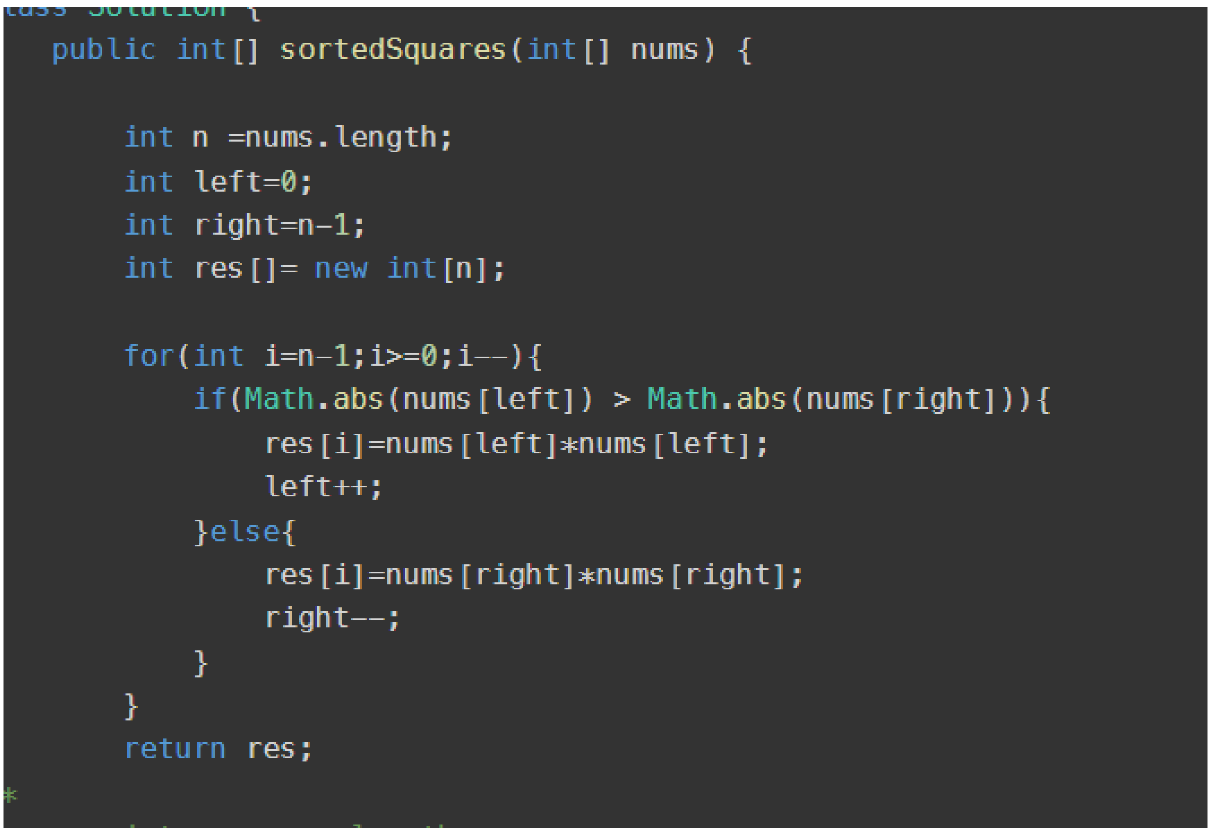
Given an integer array `nums` sorted in **non-decreasing** order, return *an array of **the squares of each number** sorted in non-decreasing order.*

Example 1:

```
Input: nums = [-4,-1,0,3,10]
Output: [0,1,9,16,100]
Explanation: After squaring, the array becomes [16,1,0,9,100].
After sorting, it becomes [0,1,9,16,100].
```

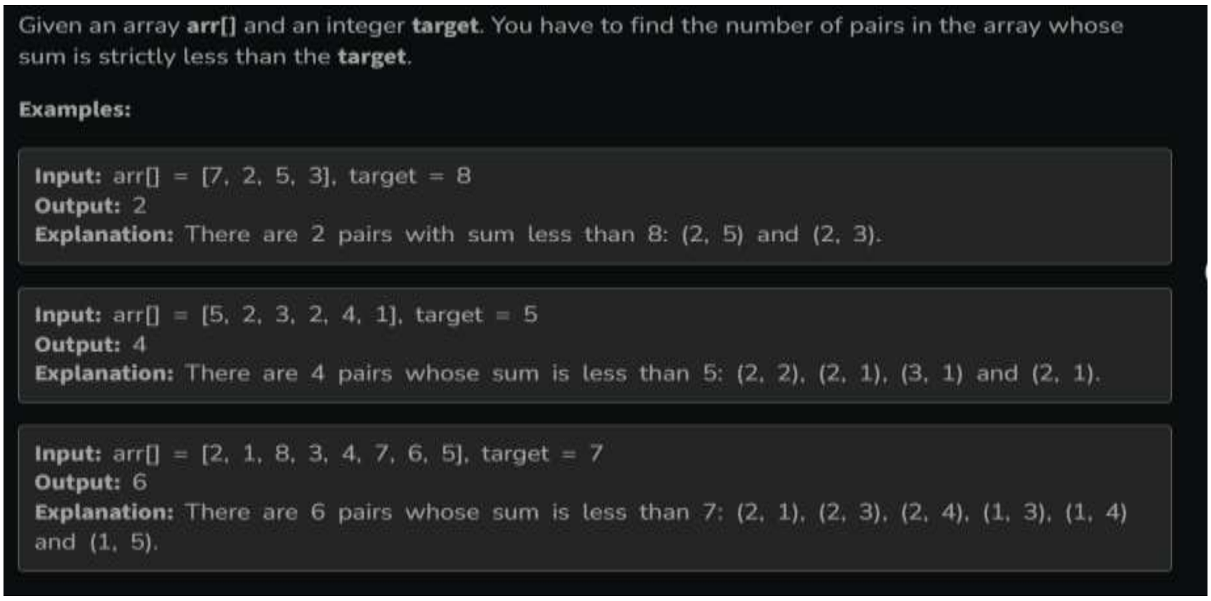
Example 2:

```
Input: nums = [-7,-3,2,3,11]
Output: [4,9,9,49,121]
```



```
  public int[] sortedSquares(int[] nums) {

      int n =nums.length;
      int left=0;
      int right=n-1;
      int res[]= new int[n];

      for(int i=n-1;i>=0;i--){
          if(Math.abs(nums[left]) > Math.abs(nums[right])){
              res[i]=nums[left]*nums[left];
              left++;
          }else{
              res[i]=nums[right]*nums[right];
              right--;
          }
      }
      return res;
*
```

9.Count pair whose sum is less than target

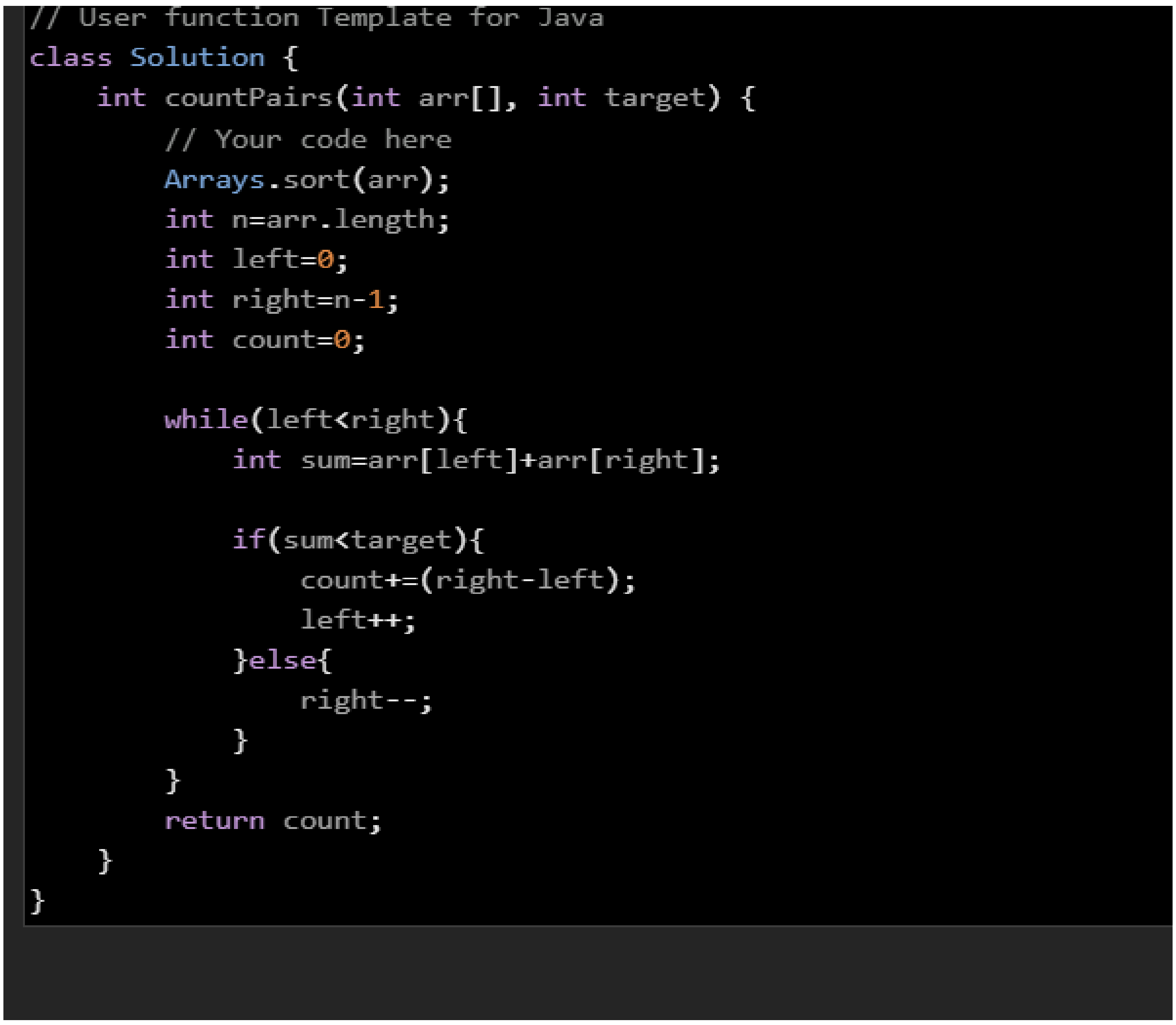

Given an array **arr[]** and an integer **target**. You have to find the number of pairs in the array whose sum is strictly less than the **target**.

Examples:

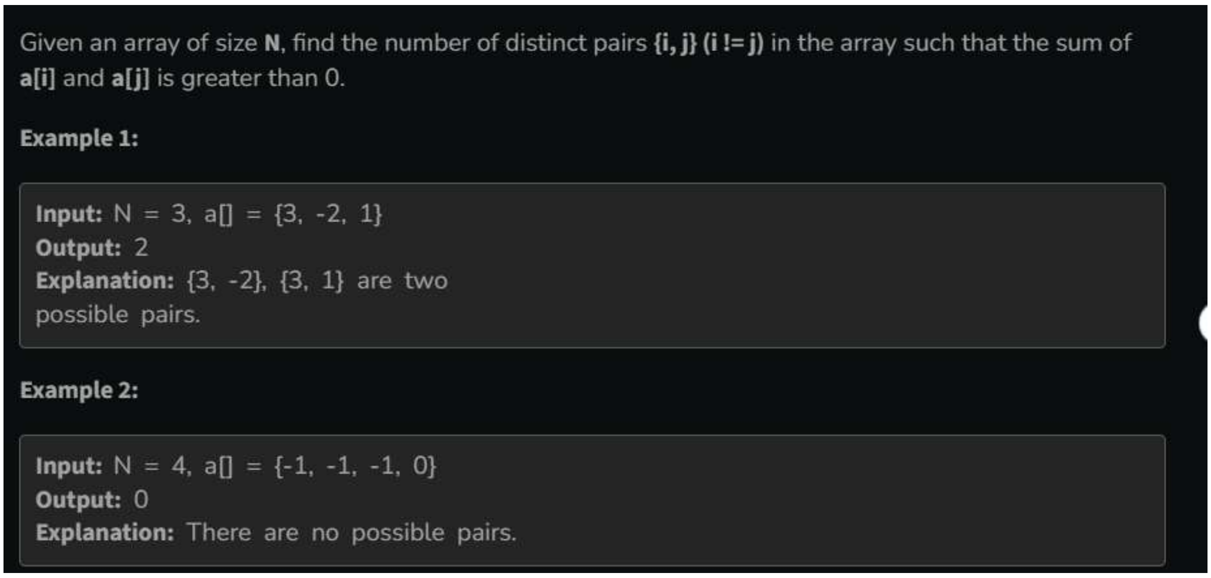
Input: arr[] = [7, 2, 5, 3], target = 8
Output: 2
Explanation: There are 2 pairs with sum less than 8: (2, 5) and (2, 3).

Input: arr[] = [5, 2, 3, 2, 4, 1], target = 5
Output: 4
Explanation: There are 4 pairs whose sum is less than 5: (2, 2), (2, 1), (3, 1) and (2, 1).

Input: arr[] = [2, 1, 8, 3, 4, 7, 6, 5], target = 7
Output: 6
Explanation: There are 6 pairs whose sum is less than 7: (2, 1), (2, 3), (2, 4), (1, 3), (1, 4) and (1, 5).


```java
// User function Template for Java
class Solution {
    int countPairs(int arr[], int target) {
        // Your code here
        Arrays.sort(arr);
        int n=arr.length;
        int left=0;
        int right=n-1;
        int count=0;

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

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

10.Valid Pair Sum

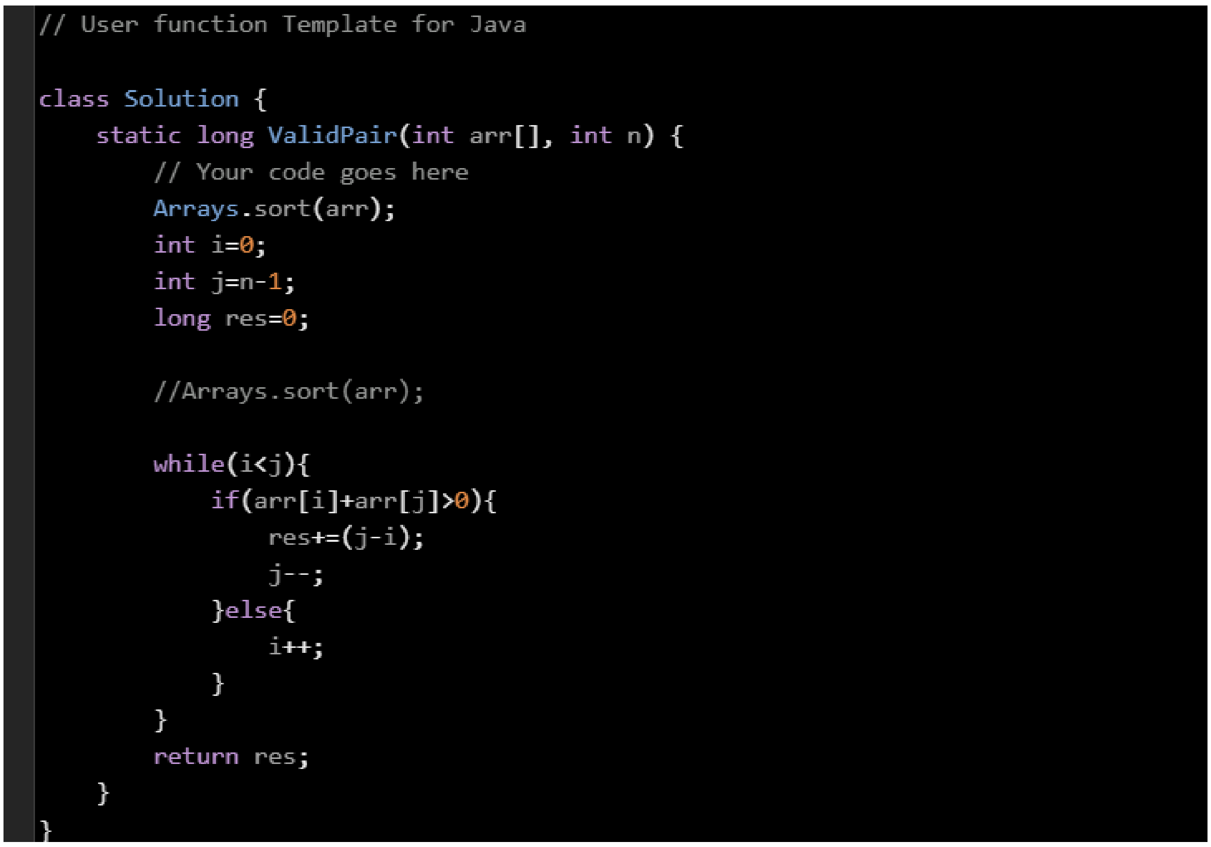

Given an array of size **N**, find the number of distinct pairs **{i, j} (i != j)** in the array such that the sum of **a[i]** and **a[j]** is greater than 0.

Example 1:

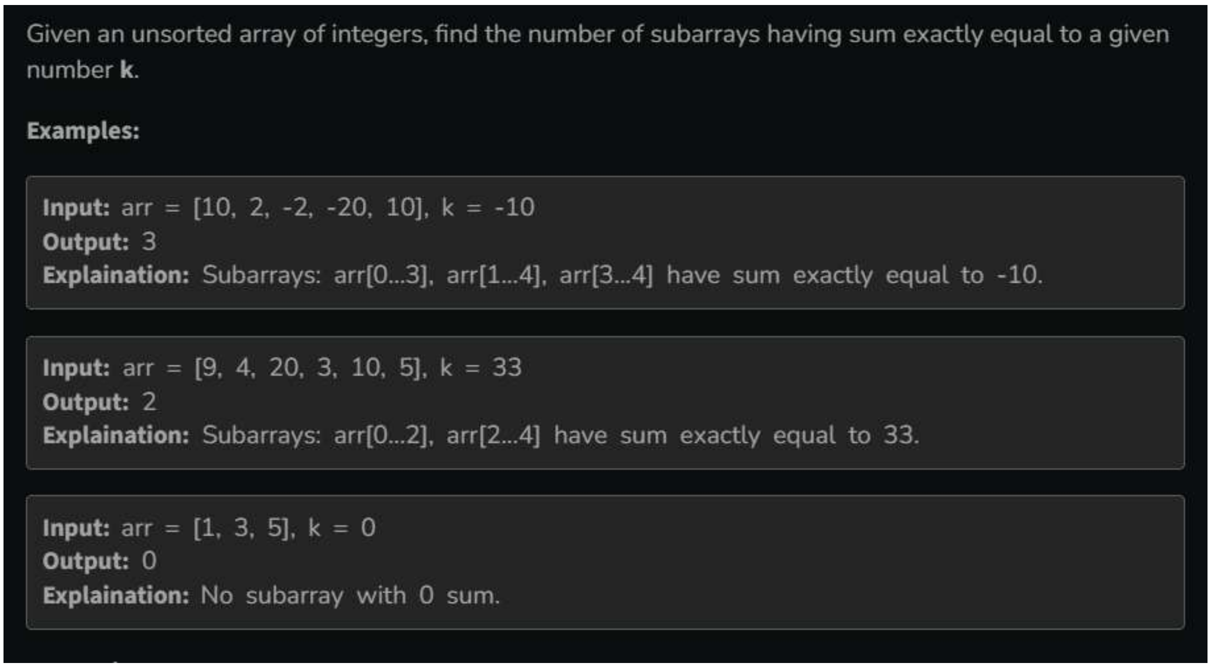
```
Input: N = 3, a[] = {3, -2, 1}
Output: 2
Explanation: {3, -2}, {3, 1} are two
possible pairs.
```

Example 2:

```
Input: N = 4, a[] = {-1, -1, -1, 0}
Output: 0
Explanation: There are no possible pairs.
```


```java
// User function Template for Java

class Solution {
    static long ValidPair(int arr[], int n) {
        // Your code goes here
        Arrays.sort(arr);
        int i=0;
        int j=n-1;
        long res=0;

        //Arrays.sort(arr);

        while(i<j){
            if(arr[i]+arr[j]>0){
                res+=(j-i);
                j--;
            }else{
                i++;
            }
        }
        return res;
    }
}
```

11.Count Subarray having sum k

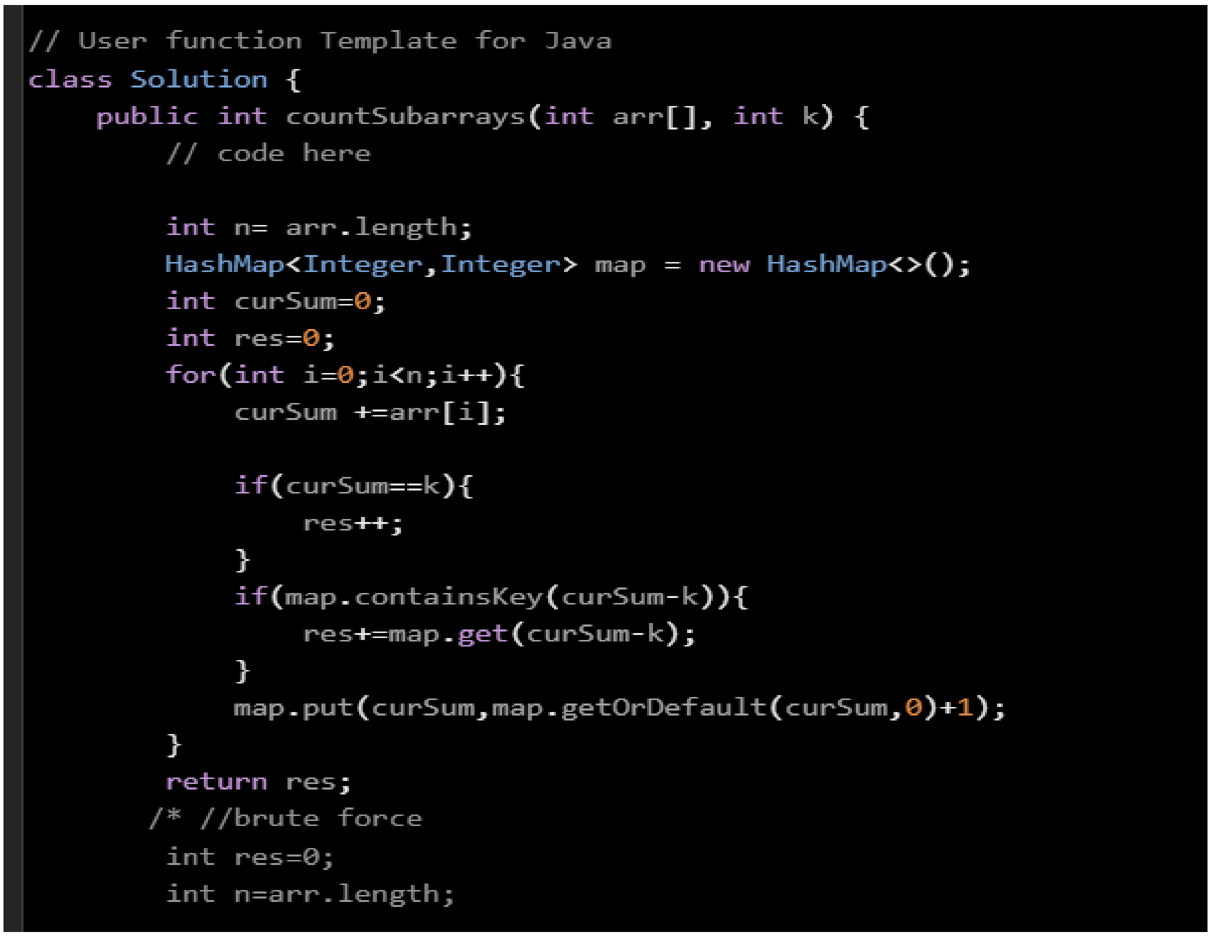

Given an unsorted array of integers, find the number of subarrays having sum exactly equal to a given number **k**.

Examples:

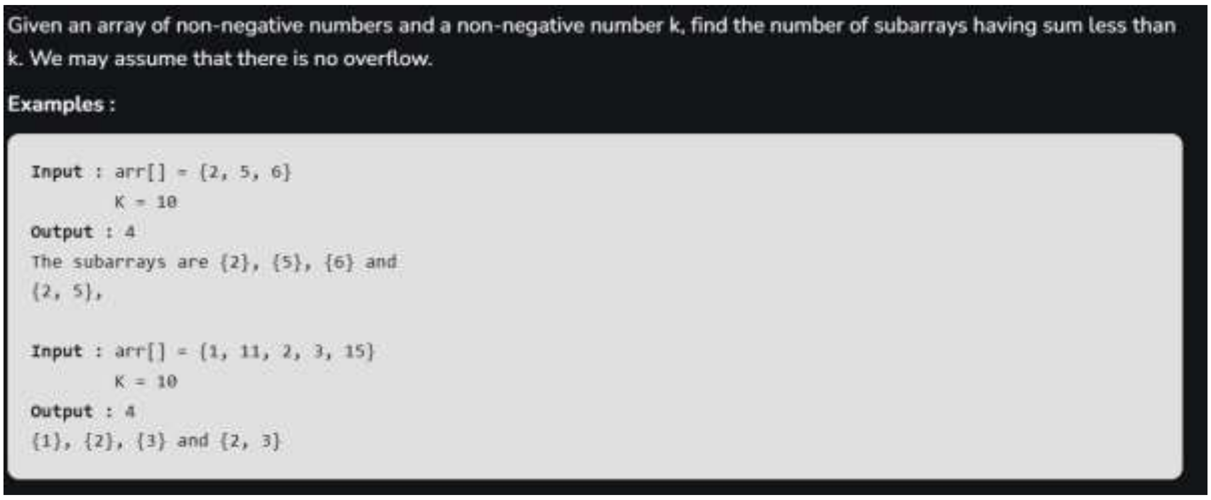
```
Input: arr = [10, 2, -2, -20, 10], k = -10
Output: 3
Explaination: Subarrays: arr[0...3], arr[1...4], arr[3...4] have sum exactly equal to -10.
```

```
Input: arr = [9, 4, 20, 3, 10, 5], k = 33
Output: 2
Explaination: Subarrays: arr[0...2], arr[2...4] have sum exactly equal to 33.
```

```
Input: arr = [1, 3, 5], k = 0
Output: 0
Explaination: No subarray with 0 sum.
```


```java
// User function Template for Java
class Solution {
    public int countSubarrays(int arr[], int k) {
        // code here

        int n= arr.length;
        HashMap<Integer,Integer> map = new HashMap<>();
        int curSum=0;
        int res=0;
        for(int i=0;i<n;i++){
            curSum +=arr[i];

            if(curSum==k){
                res++;
            }
            if(map.containsKey(curSum-k)){
                res+=map.get(curSum-k);
            }
            map.put(curSum,map.getOrDefault(curSum,0)+1);
        }
        return res;
       /* //brute force
        int res=0;
        int n=arr.length;
```

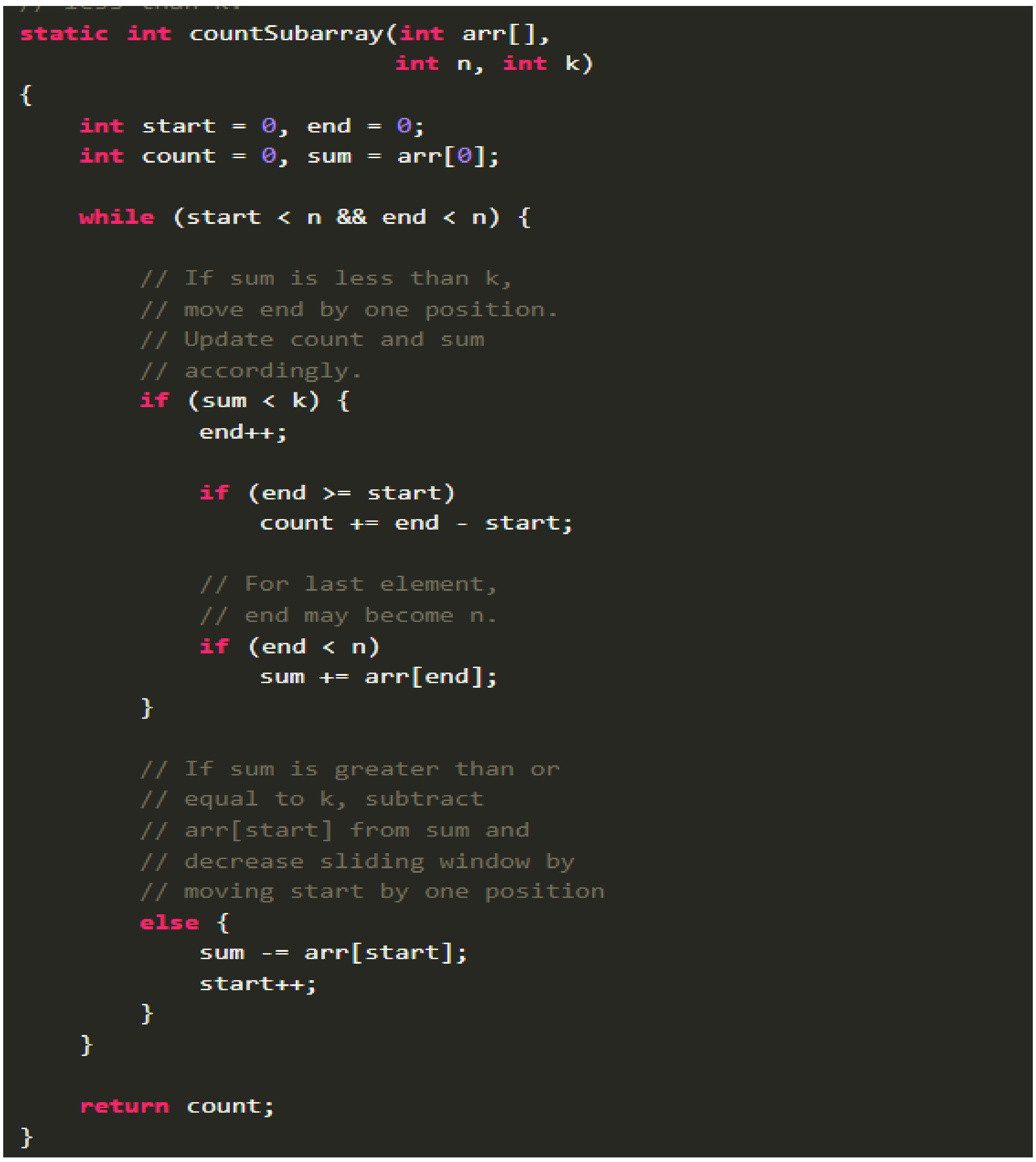
12.Number of subarray having sum less than k



Given an array of non-negative numbers and a non-negative number k, find the number of subarrays having sum less than k. We may assume that there is no overflow.

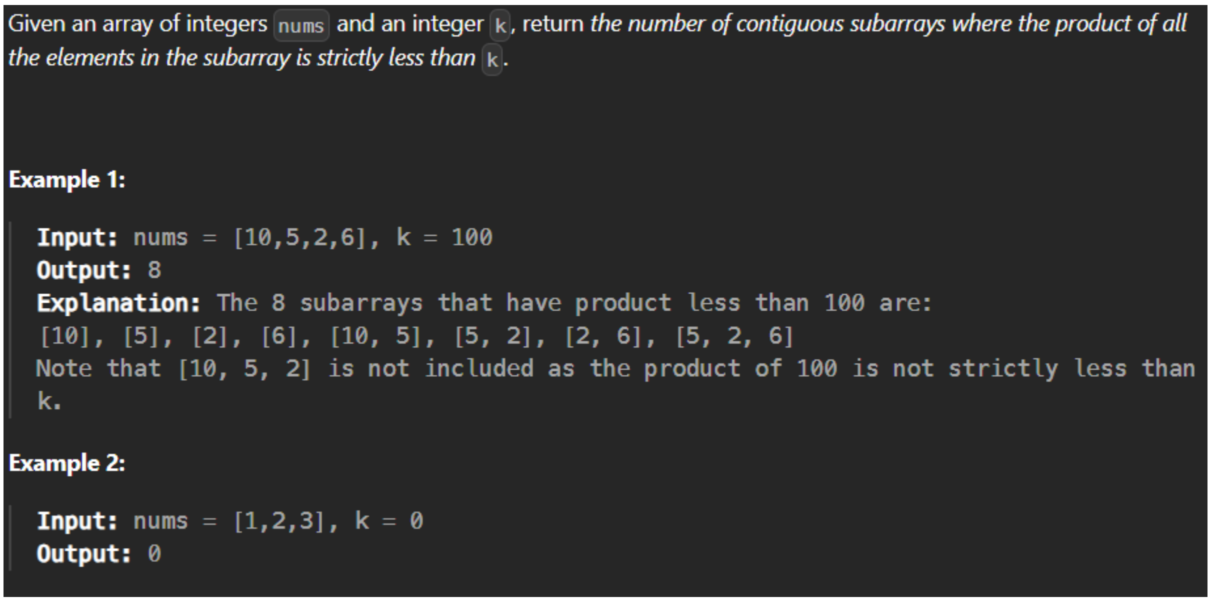
Examples :

```
Input : arr[] = {2, 5, 6}
        K = 10
Output : 4
The subarrays are {2}, {5}, {6} and
{2, 5},

Input : arr[] = {1, 11, 2, 3, 15}
        K = 10
Output : 4
{1}, {2}, {3} and {2, 3}
```



```
// less than k.
static int countSubarray(int arr[],
                         int n, int k)
{
    int start = 0, end = 0;
    int count = 0, sum = arr[0];

    while (start < n && end < n) {

        // If sum is less than k,
        // move end by one position.
        // Update count and sum
        // accordingly.
        if (sum < k) {
            end++;

            if (end >= start)
                count += end - start;

            // For last element,
            // end may become n.
            if (end < n)
                sum += arr[end];
        }

        // If sum is greater than or
        // equal to k, subtract
        // arr[start] from sum and
        // decrease sliding window by
        // moving start by one position
        else {
            sum -= arr[start];
            start++;
        }
    }

    return count;
}
```

13.Subarray product less than k



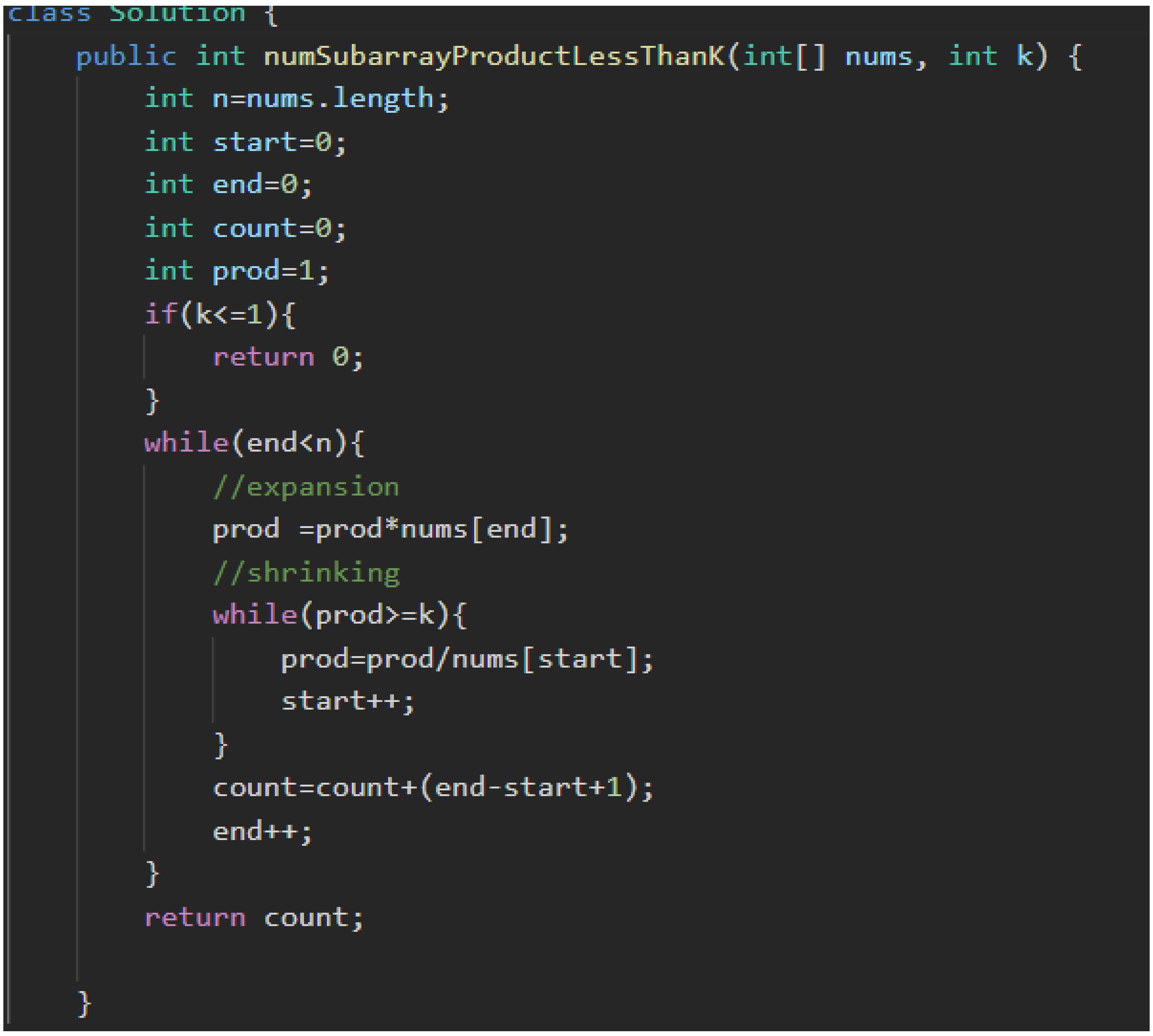
Given an array of integers `nums` and an integer `k`, return *the number of contiguous subarrays where the product of all the elements in the subarray is strictly less than* `k`.

Example 1:

```
Input: nums = [10,5,2,6], k = 100
Output: 8
Explanation: The 8 subarrays that have product less than 100 are:
[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]
Note that [10, 5, 2] is not included as the product of 100 is not strictly less than
k.
```

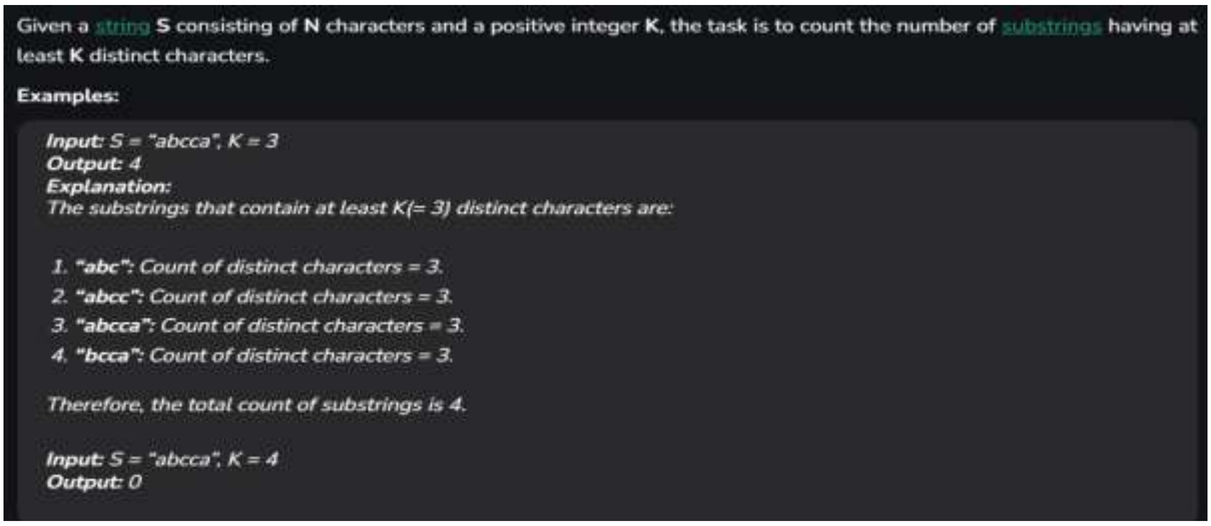
Example 2:

```
Input: nums = [1,2,3], k = 0
Output: 0
```



```
class Solution {
    public int numSubarrayProductLessThanK(int[] nums, int k) {
        int n=nums.length;
        int start=0;
        int end=0;
        int count=0;
        int prod=1;
        if(k<=1){
            return 0;
        }
        while(end<n){
            //expansion
            prod =prod*nums[end];
            //shrinking
            while(prod>=k){
                prod=prod/nums[start];
                start++;
            }
            count=count+(end-start+1);
            end++;
        }
        return count;

    }
```

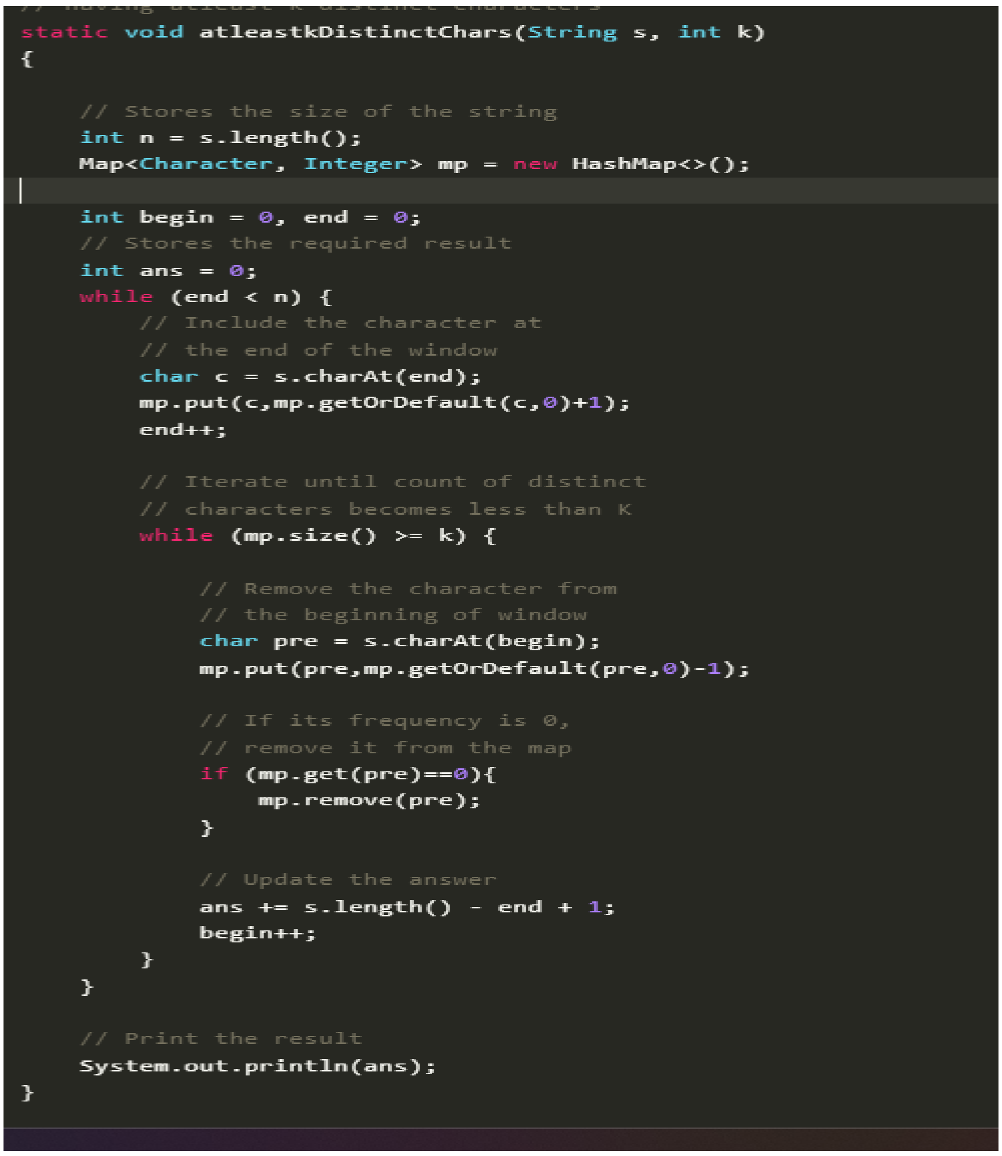
14.Count Number of substring having atleast k distinct character



Given a string S consisting of N characters and a positive integer K, the task is to count the number of substrings having at least K distinct characters.

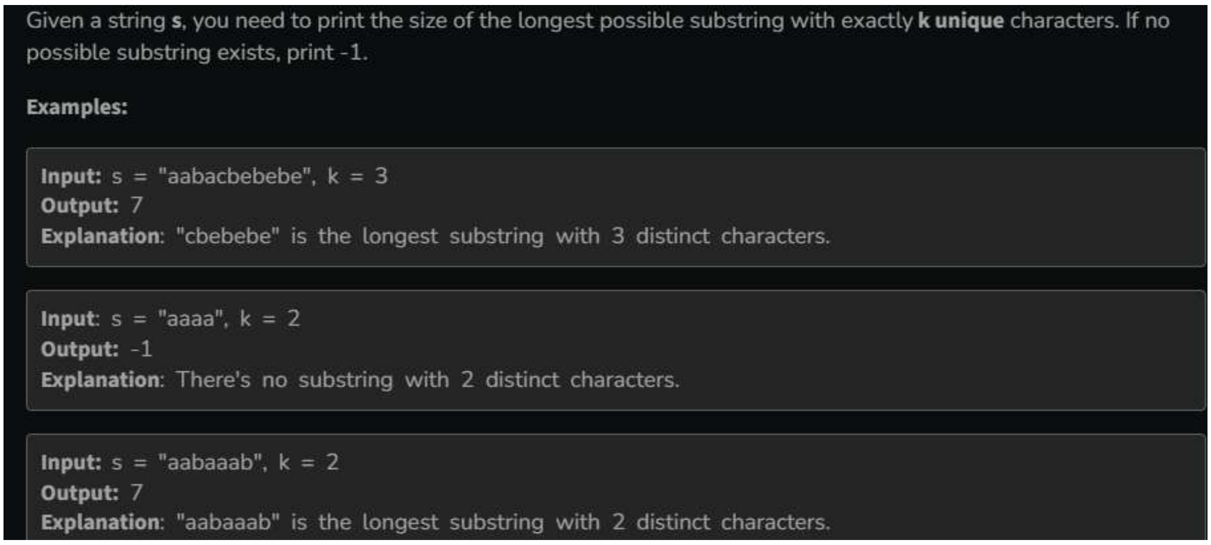
Examples:

> ***Input:*** *S = "abcca", K = 3*
> ***Output:*** *4*
> ***Explanation:***
> *The substrings that contain at least K(= 3) distinct characters are:*
>
> 1. ***"abc"****: Count of distinct characters = 3.*
> 2. ***"abcc"****: Count of distinct characters = 3.*
> 3. ***"abcca"****: Count of distinct characters = 3.*
> 4. ***"bcca"****: Count of distinct characters = 3.*
>
> *Therefore, the total count of substrings is 4.*
>
> ***Input:*** *S = "abcca", K = 4*
> ***Output:*** *0*



```java
static void atleastkDistinctChars(String s, int k)
{

    // Stores the size of the string
    int n = s.length();
    Map<Character, Integer> mp = new HashMap<>();

    int begin = 0, end = 0;
    // Stores the required result
    int ans = 0;
    while (end < n) {
        // Include the character at
        // the end of the window
        char c = s.charAt(end);
        mp.put(c,mp.getOrDefault(c,0)+1);
        end++;

        // Iterate until count of distinct
        // characters becomes less than K
        while (mp.size() >= k) {

            // Remove the character from
            // the beginning of window
            char pre = s.charAt(begin);
            mp.put(pre,mp.getOrDefault(pre,0)-1);

            // If its frequency is 0,
            // remove it from the map
            if (mp.get(pre)==0){
                mp.remove(pre);
            }

            // Update the answer
            ans += s.length() - end + 1;
            begin++;
        }
    }

    // Print the result
    System.out.println(ans);
}
```

15.Longest substring with k unique character



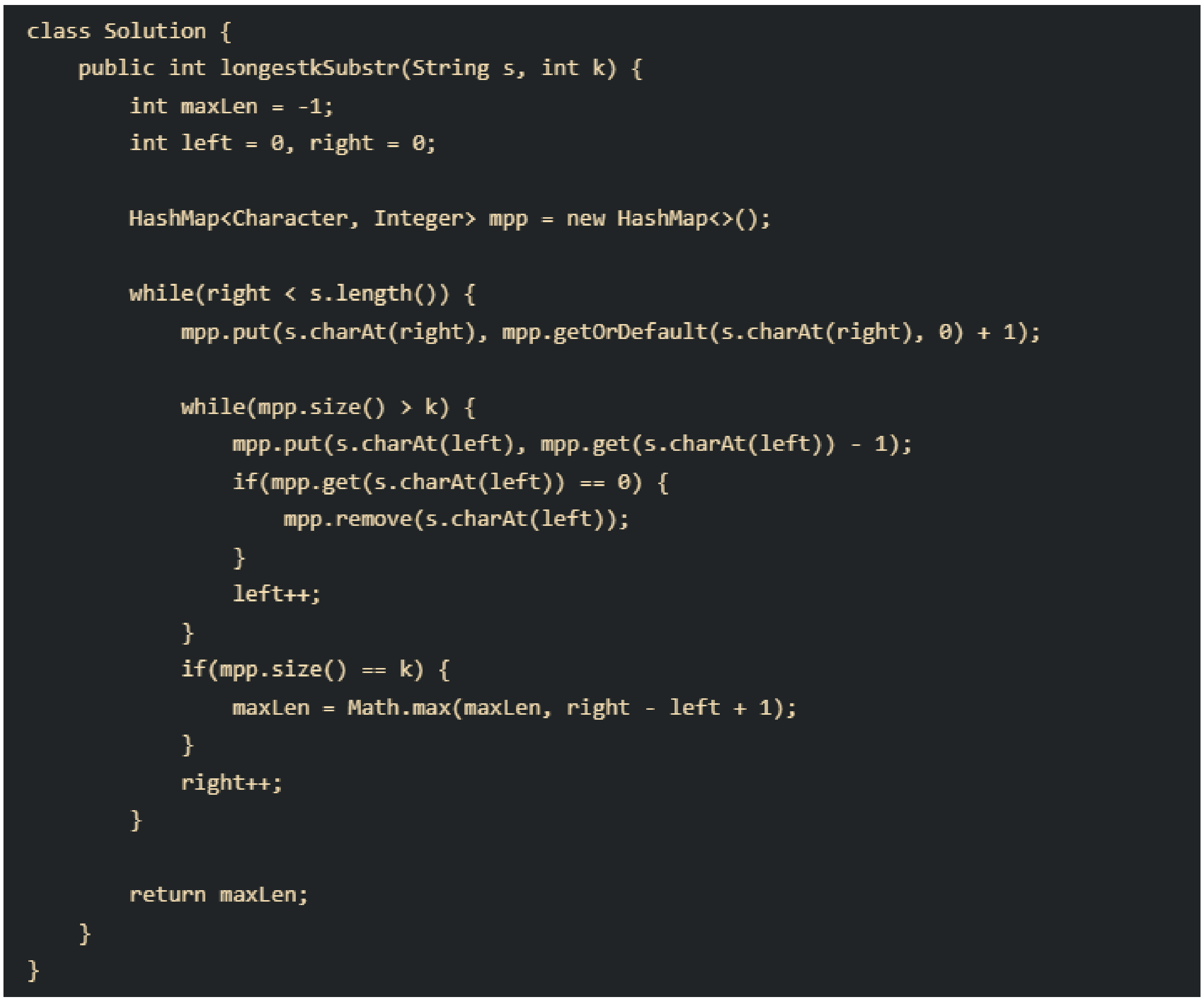
Given a string **s**, you need to print the size of the longest possible substring with exactly **k unique** characters. If no possible substring exists, print -1.

Examples:

```
Input: s = "aabacbebebe", k = 3
Output: 7
Explanation: "cbebebe" is the longest substring with 3 distinct characters.
```

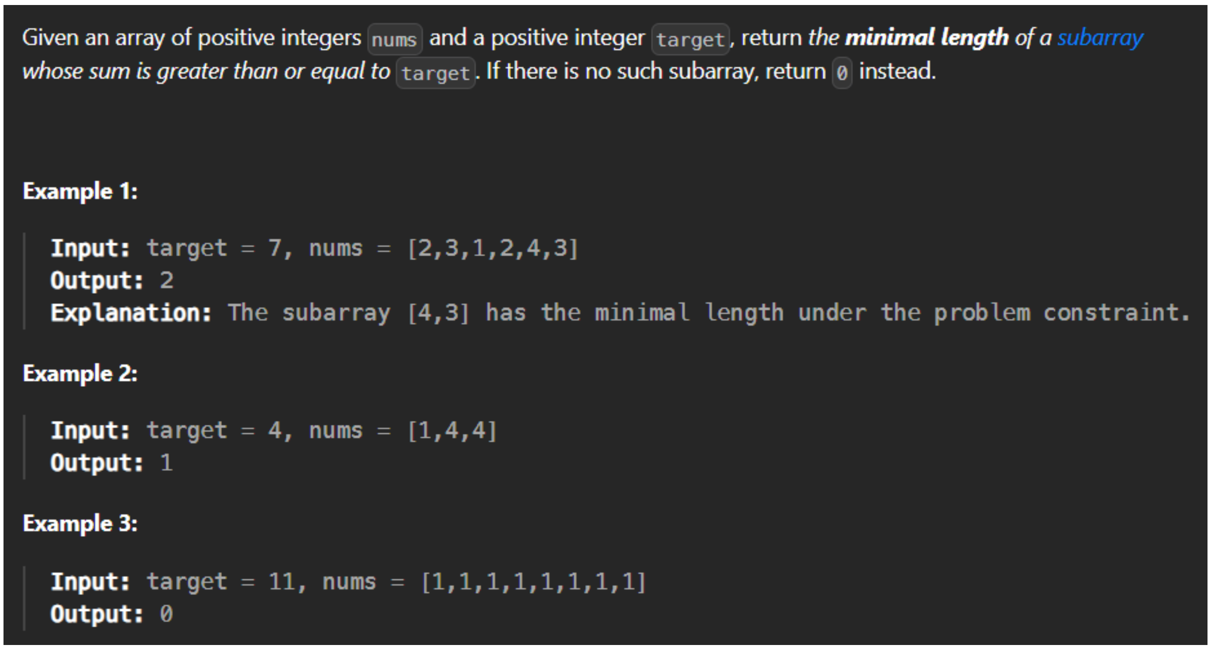
```
Input: s = "aaaa", k = 2
Output: -1
Explanation: There's no substring with 2 distinct characters.
```

```
Input: s = "aabaaab", k = 2
Output: 7
Explanation: "aabaaab" is the longest substring with 2 distinct characters.
```



```java
class Solution {
    public int longestkSubstr(String s, int k) {
        int maxLen = -1;
        int left = 0, right = 0;

        HashMap<Character, Integer> mpp = new HashMap<>();

        while(right < s.length()) {
            mpp.put(s.charAt(right), mpp.getOrDefault(s.charAt(right), 0) + 1);

            while(mpp.size() > k) {
                mpp.put(s.charAt(left), mpp.get(s.charAt(left)) - 1);
                if(mpp.get(s.charAt(left)) == 0) {
                    mpp.remove(s.charAt(left));
                }
                left++;
            }
            if(mpp.size() == k) {
                maxLen = Math.max(maxLen, right - left + 1);
            }
            right++;
        }

        return maxLen;
    }
}
```

16.Minimum size subarray sum



Given an array of positive integers `nums` and a positive integer `target`, return *the **minimal length** of a subarray whose sum is greater than or equal to* `target`. If there is no such subarray, return `0` instead.

Example 1:

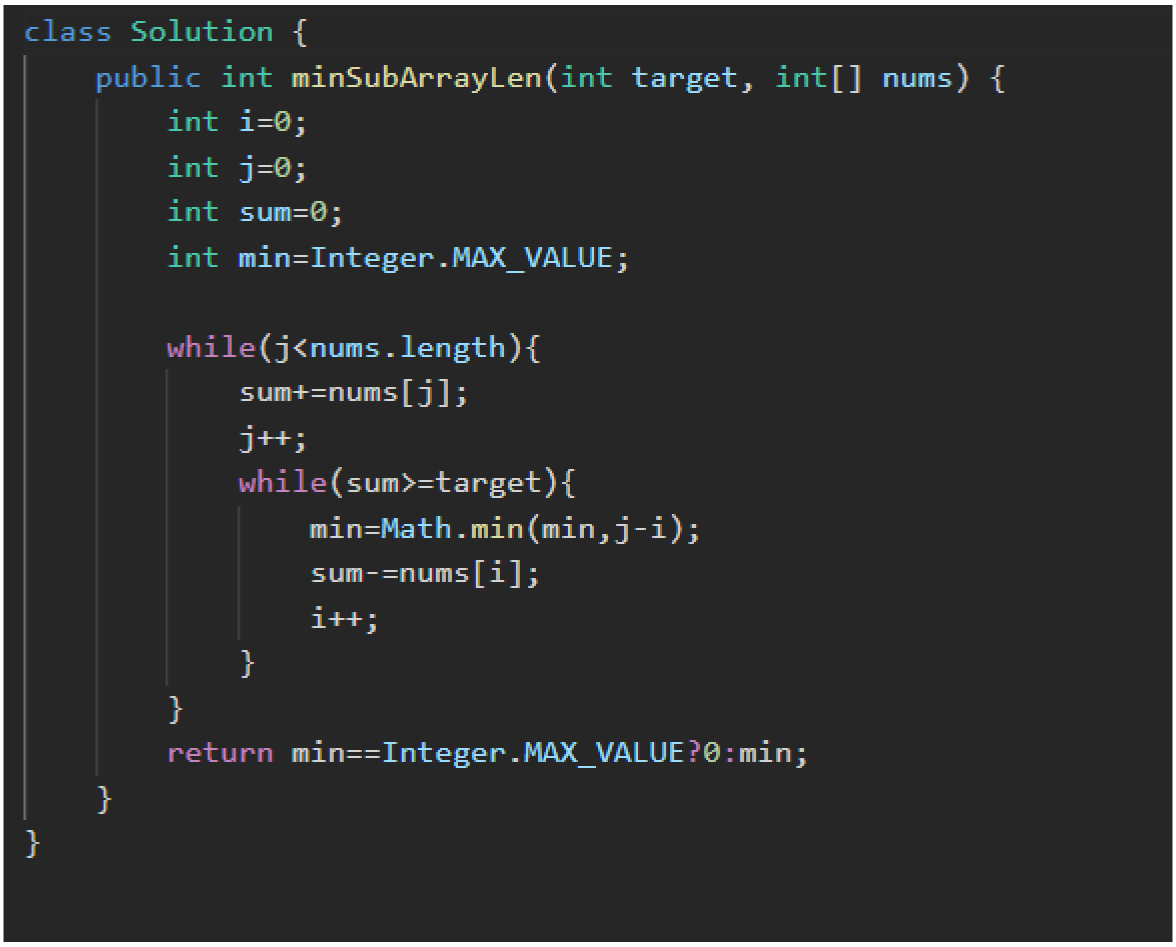
```
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
```

Example 2:

```
Input: target = 4, nums = [1,4,4]
Output: 1
```

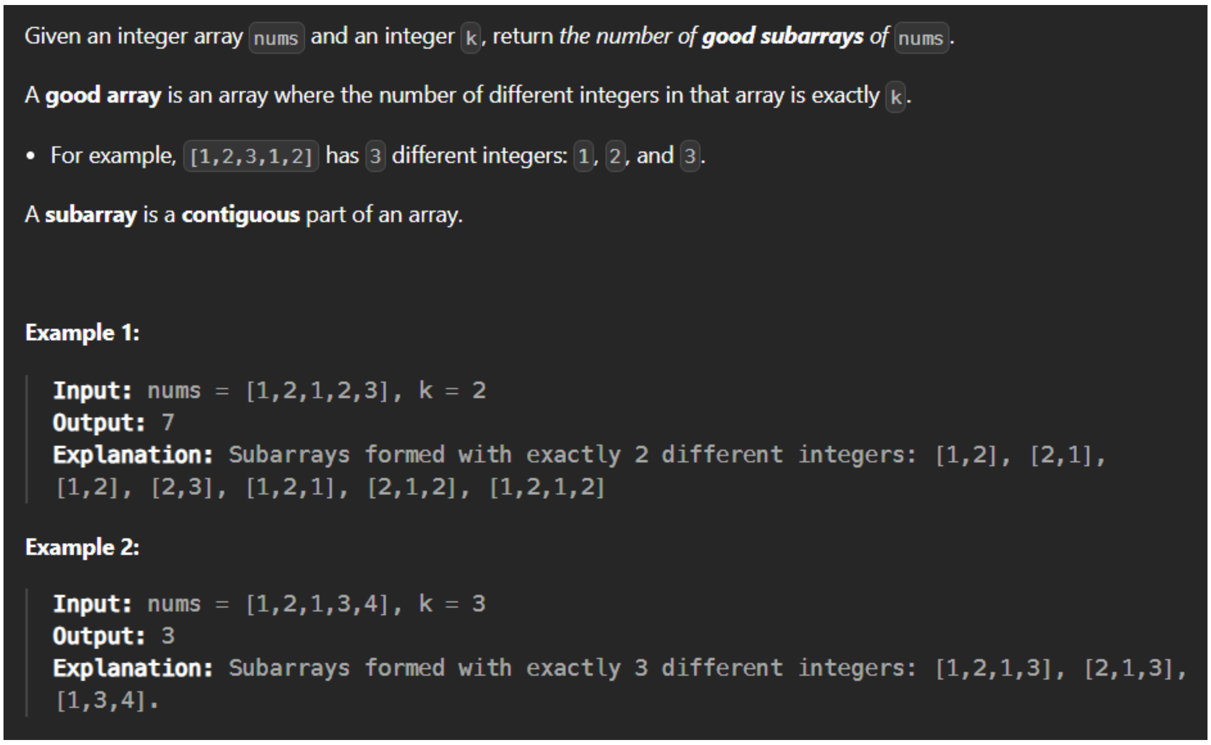
Example 3:

```
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
```



```java
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int i=0;
        int j=0;
        int sum=0;
        int min=Integer.MAX_VALUE;

        while(j<nums.length){
            sum+=nums[j];
            j++;
            while(sum>=target){
                min=Math.min(min,j-i);
                sum-=nums[i];
                i++;
            }
        }
        return min==Integer.MAX_VALUE?0:min;
    }
}
```

17.Subarray with k different Integer



Given an integer array `nums` and an integer `k`, return *the number of **good subarrays** of* `nums`.

A **good array** is an array where the number of different integers in that array is exactly `k`.

- For example, `[1,2,3,1,2]` has `3` different integers: `1`, `2`, and `3`.

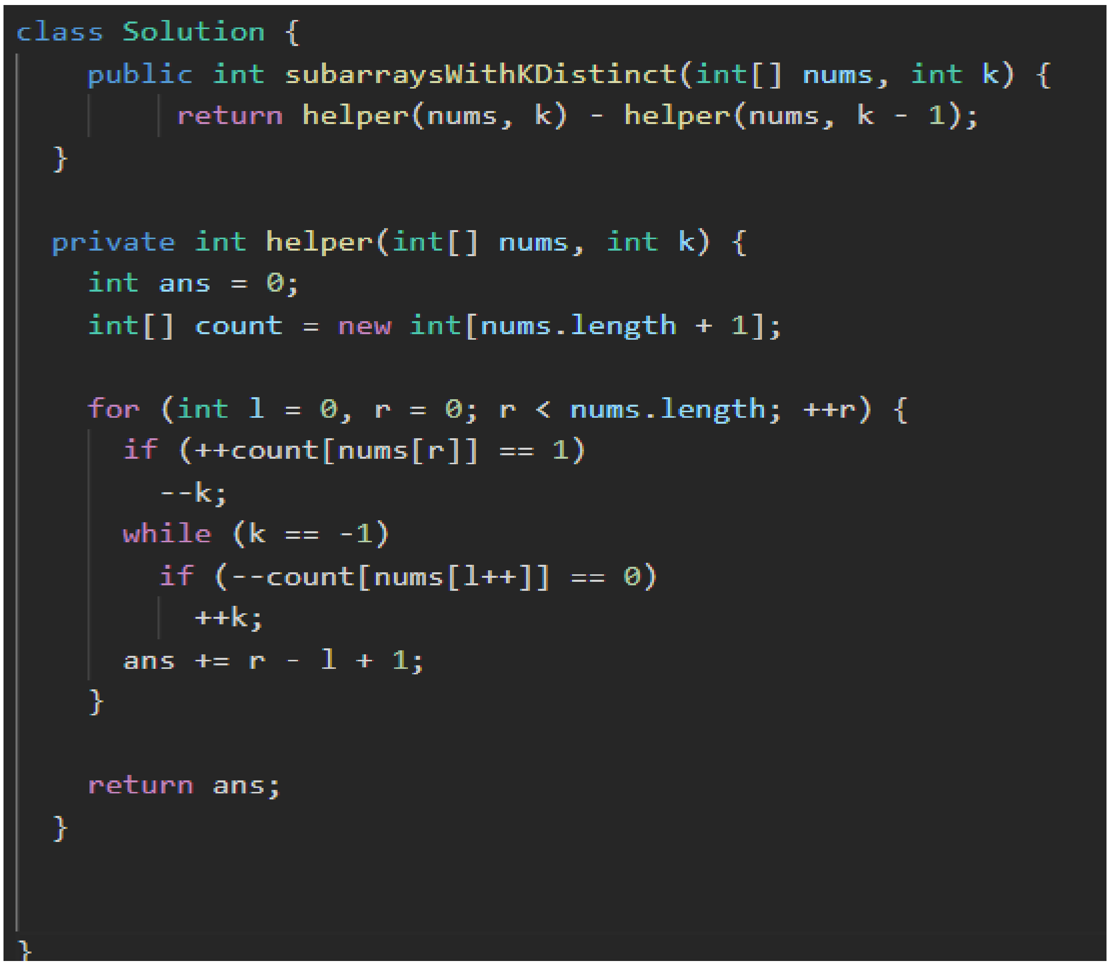
A **subarray** is a **contiguous** part of an array.

Example 1:

```
Input: nums = [1,2,1,2,3], k = 2
Output: 7
Explanation: Subarrays formed with exactly 2 different integers: [1,2], [2,1],
[1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]
```

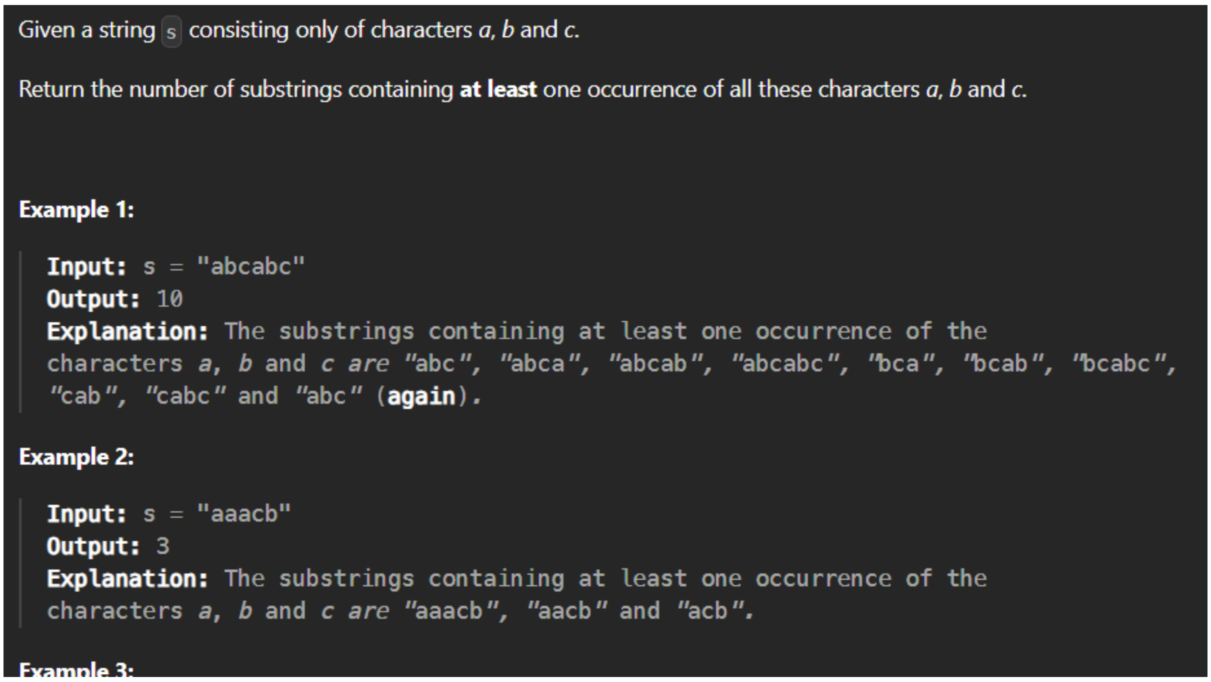
Example 2:

```
Input: nums = [1,2,1,3,4], k = 3
Output: 3
Explanation: Subarrays formed with exactly 3 different integers: [1,2,1,3], [2,1,3],
[1,3,4].
```



```
class Solution {
    public int subarraysWithKDistinct(int[] nums, int k) {
        return helper(nums, k) - helper(nums, k - 1);
  }

  private int helper(int[] nums, int k) {
    int ans = 0;
    int[] count = new int[nums.length + 1];

    for (int l = 0, r = 0; r < nums.length; ++r) {
      if (++count[nums[r]] == 1)
        --k;
      while (k == -1)
        if (--count[nums[l++]] == 0)
          ++k;
      ans += r - l + 1;
    }

    return ans;
  }


}
```

18.Number of substring containing all three character



Given a string `s` consisting only of characters *a*, *b* and *c*.

Return the number of substrings containing **at least** one occurrence of all these characters *a*, *b* and *c*.

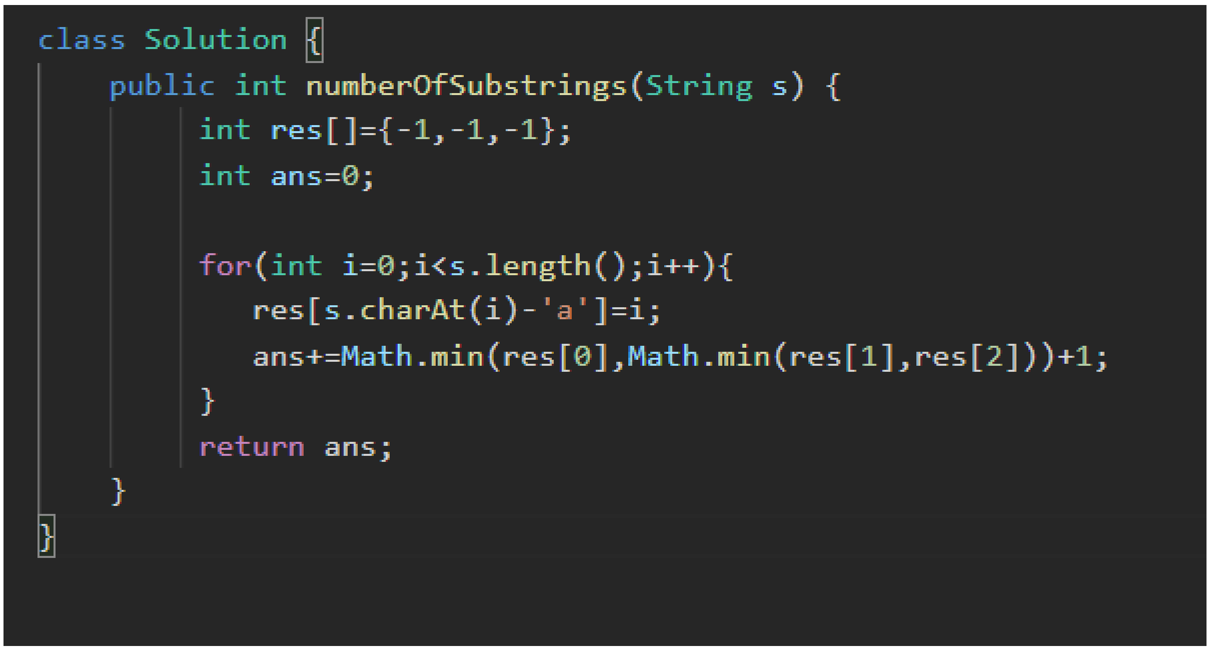
Example 1:

```
Input: s = "abcabc"
Output: 10
Explanation: The substrings containing at least one occurrence of the
characters a, b and c are "abc", "abca", "abcab", "abcabc", "bca", "bcab", "bcabc",
"cab", "cabc" and "abc" (again).
```

Example 2:

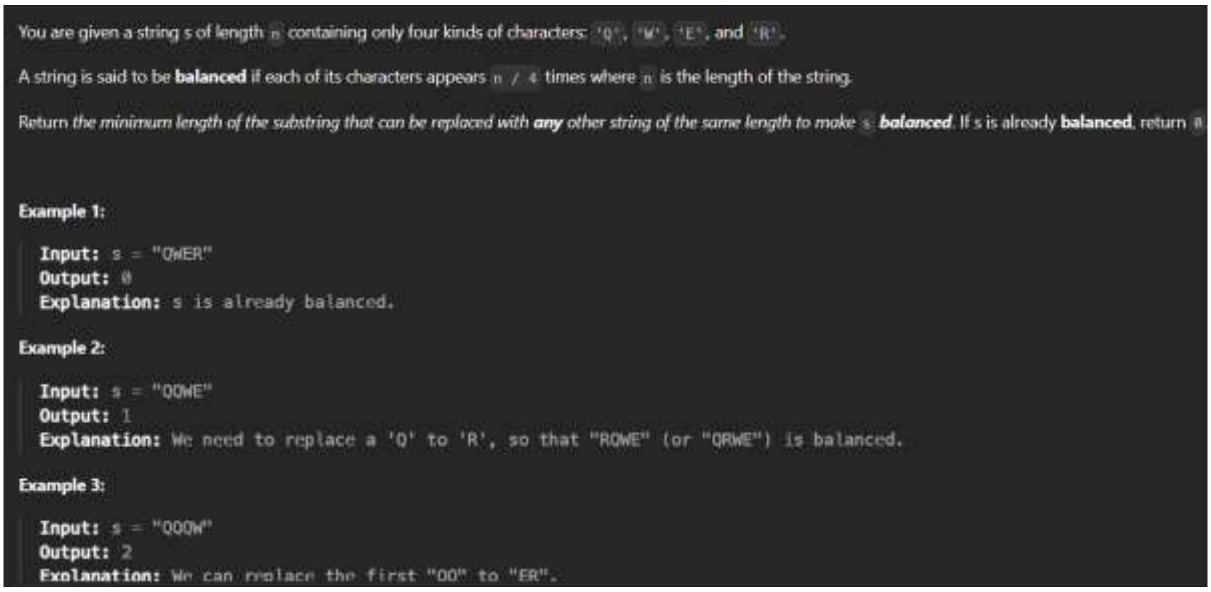
```
Input: s = "aaacb"
Output: 3
Explanation: The substrings containing at least one occurrence of the
characters a, b and c are "aaacb", "aacb" and "acb".
```

Example 3:



```java
class Solution {
    public int numberOfSubstrings(String s) {
        int res[]={-1,-1,-1};
        int ans=0;

        for(int i=0;i<s.length();i++){
            res[s.charAt(i)-'a']=i;
            ans+=Math.min(res[0],Math.min(res[1],res[2]))+1;
        }
        return ans;
    }
}
```

19.Replace the substring for balanced string



You are given a string s of length n containing only four kinds of characters: 'Q', 'W', 'E', and 'R'.

A string is said to be **balanced** if each of its characters appears n / 4 times where n is the length of the string.

*Return the minimum length of the substring that can be replaced with **any** other string of the same length to make* s ***balanced***. If s is already **balanced**, return 0.

Example 1:

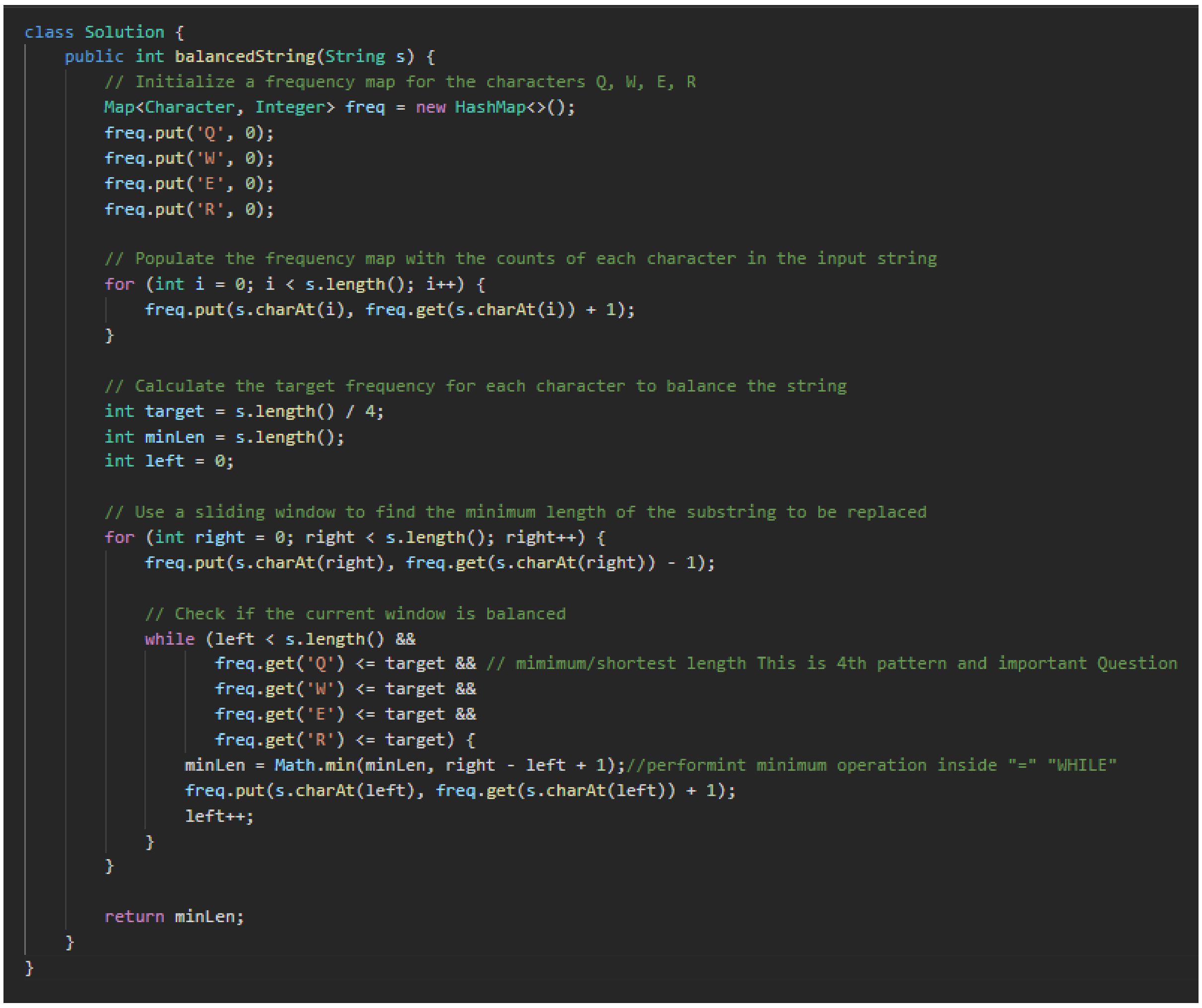
```
Input: s = "QWER"
Output: 0
Explanation: s is already balanced.
```

Example 2:

```
Input: s = "QQWE"
Output: 1
Explanation: We need to replace a 'Q' to 'R', so that "RQWE" (or "QRWE") is balanced.
```

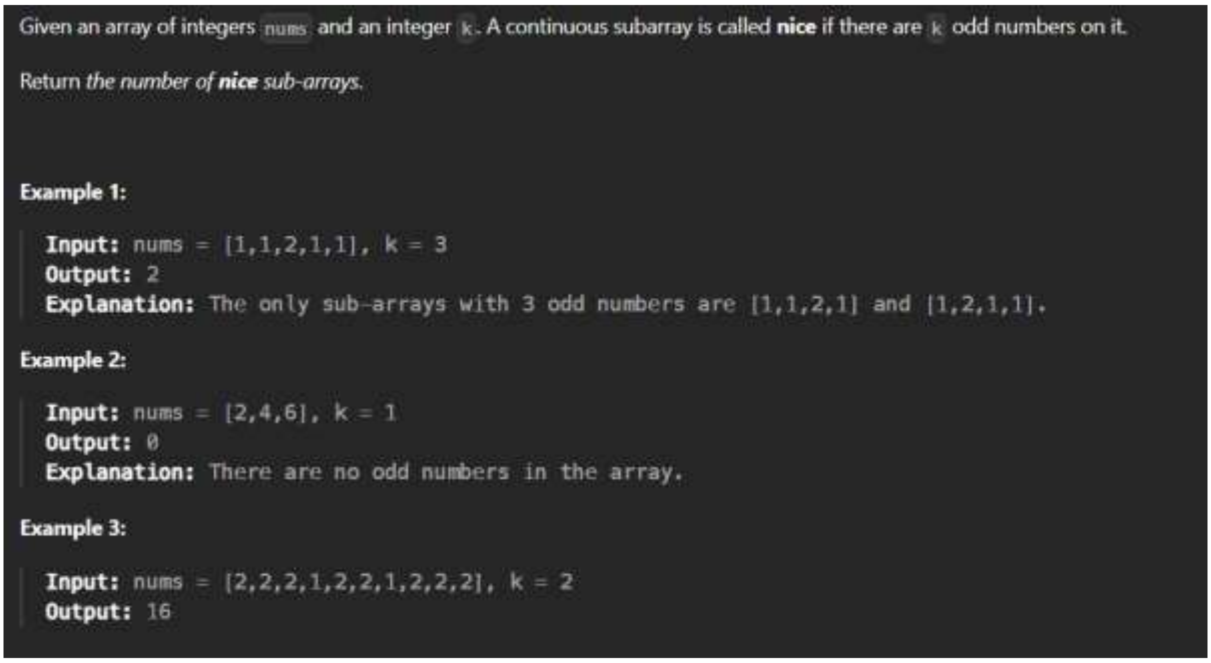
Example 3:

```
Input: s = "QQQW"
Output: 2
Explanation: We can replace the first "QQ" to "ER".
```



```java
class Solution {
    public int balancedString(String s) {
        // Initialize a frequency map for the characters Q, W, E, R
        Map<Character, Integer> freq = new HashMap<>();
        freq.put('Q', 0);
        freq.put('W', 0);
        freq.put('E', 0);
        freq.put('R', 0);

        // Populate the frequency map with the counts of each character in the input string
        for (int i = 0; i < s.length(); i++) {
            freq.put(s.charAt(i), freq.get(s.charAt(i)) + 1);
        }

        // Calculate the target frequency for each character to balance the string
        int target = s.length() / 4;
        int minLen = s.length();
        int left = 0;

        // Use a sliding window to find the minimum length of the substring to be replaced
        for (int right = 0; right < s.length(); right++) {
            freq.put(s.charAt(right), freq.get(s.charAt(right)) - 1);

            // Check if the current window is balanced
            while (left < s.length() &&
                   freq.get('Q') <= target && // mimimum/shortest length This is 4th pattern and important Question
                   freq.get('W') <= target &&
                   freq.get('E') <= target &&
                   freq.get('R') <= target) {
                minLen = Math.min(minLen, right - left + 1);//performint minimum operation inside "=" "WHILE"
                freq.put(s.charAt(left), freq.get(s.charAt(left)) + 1);
                left++;
            }
        }

        return minLen;
    }
}
```

20.Count Number of nice subarray



Given an array of integers `nums` and an integer `k`. A continuous subarray is called **nice** if there are `k` odd numbers on it.

Return *the number of **nice** sub-arrays.*

Example 1:

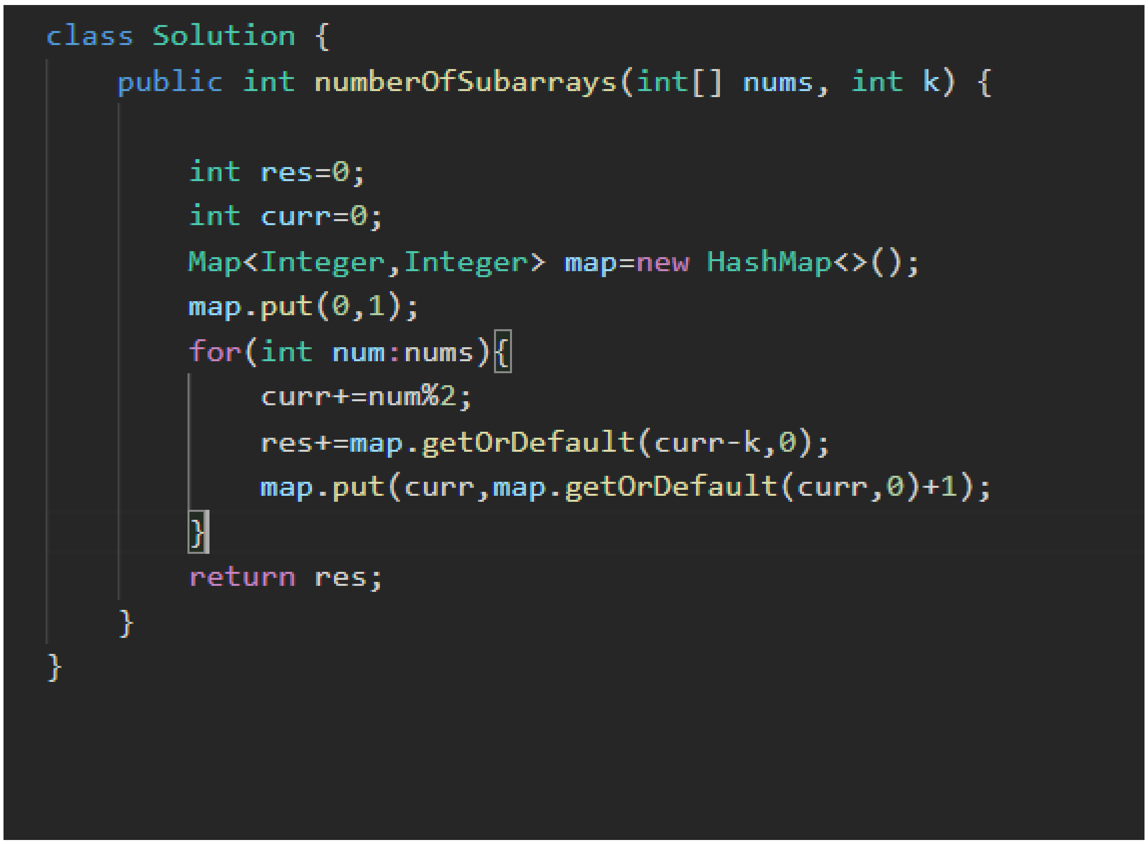
```
Input: nums = [1,1,2,1,1], k = 3
Output: 2
Explanation: The only sub-arrays with 3 odd numbers are [1,1,2,1] and [1,2,1,1].
```

Example 2:

```
Input: nums = [2,4,6], k = 1
Output: 0
Explanation: There are no odd numbers in the array.
```

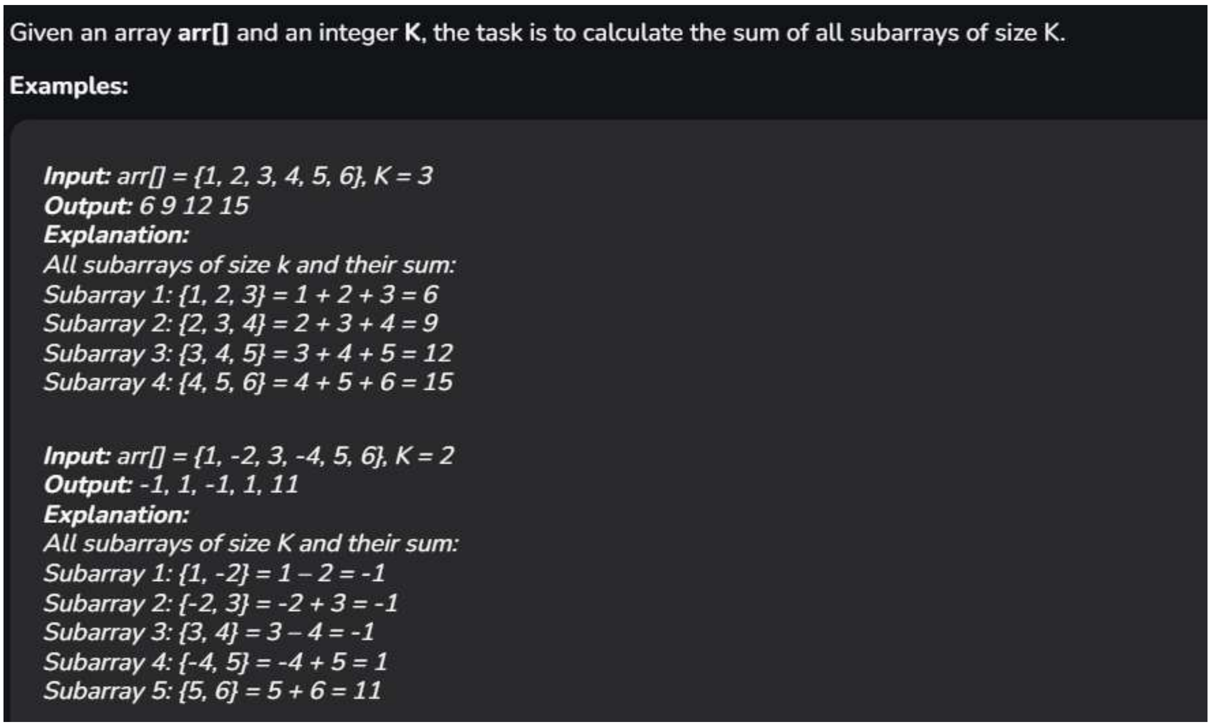
Example 3:

```
Input: nums = [2,2,2,1,2,2,1,2,2,2], k = 2
Output: 16
```



```
class Solution {
    public int numberOfSubarrays(int[] nums, int k) {

        int res=0;
        int curr=0;
        Map<Integer,Integer> map=new HashMap<>();
        map.put(0,1);
        for(int num:nums){
            curr+=num%2;
            res+=map.getOrDefault(curr-k,0);
            map.put(curr,map.getOrDefault(curr,0)+1);
        }
        return res;
    }
}
```

21.Sum of all subarray of size k

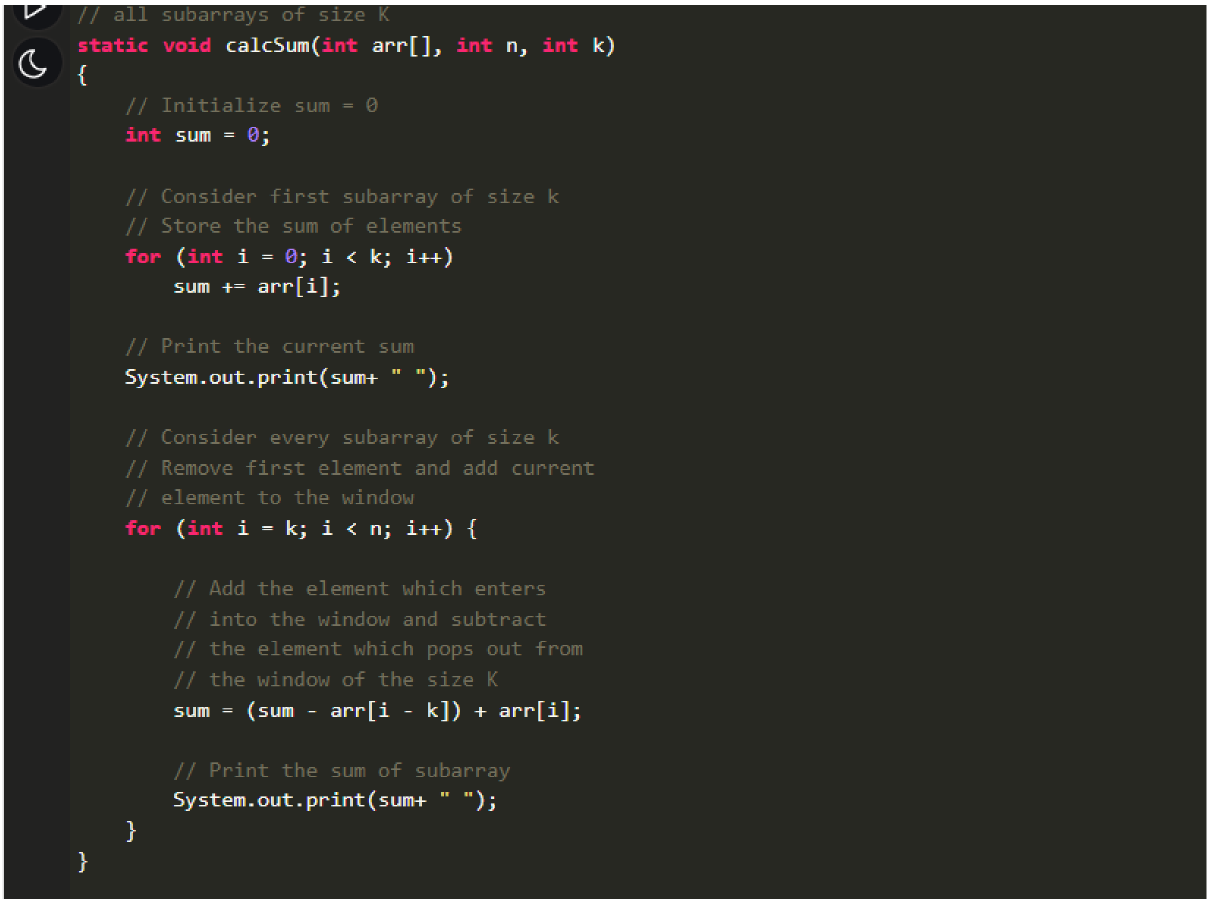


Given an array **arr[]** and an integer **K**, the task is to calculate the sum of all subarrays of size K.

Examples:

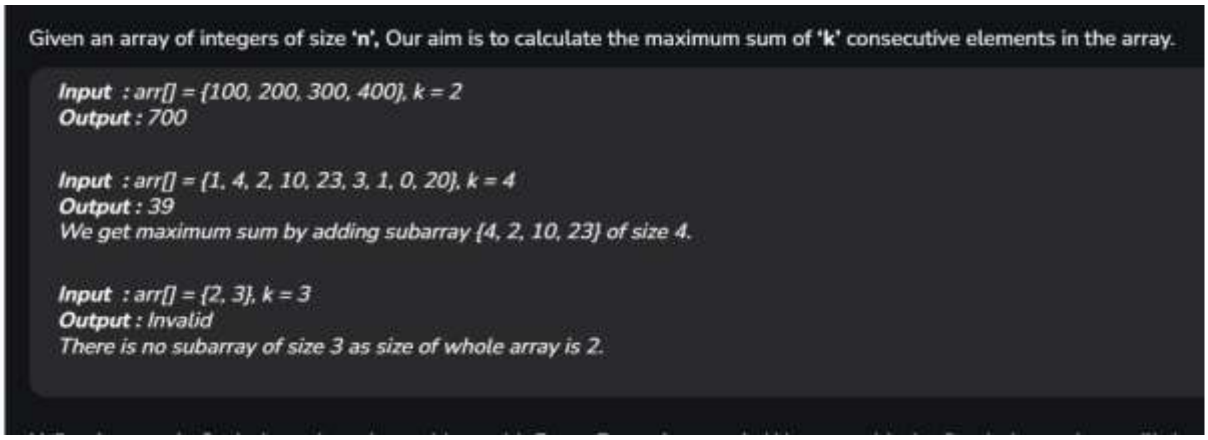
Input: *arr[] = {1, 2, 3, 4, 5, 6}, K = 3*
Output: *6 9 12 15*
Explanation:
All subarrays of size k and their sum:
Subarray 1: {1, 2, 3} = 1 + 2 + 3 = 6
Subarray 2: {2, 3, 4} = 2 + 3 + 4 = 9
Subarray 3: {3, 4, 5} = 3 + 4 + 5 = 12
Subarray 4: {4, 5, 6} = 4 + 5 + 6 = 15

Input: *arr[] = {1, -2, 3, -4, 5, 6}, K = 2*
Output: *-1, 1, -1, 1, 11*
Explanation:
All subarrays of size K and their sum:
Subarray 1: {1, -2} = 1 – 2 = -1
Subarray 2: {-2, 3} = -2 + 3 = -1
Subarray 3: {3, 4} = 3 – 4 = -1
Subarray 4: {-4, 5} = -4 + 5 = 1
Subarray 5: {5, 6} = 5 + 6 = 11



```java
// all subarrays of size K
static void calcSum(int arr[], int n, int k)
{
    // Initialize sum = 0
    int sum = 0;

    // Consider first subarray of size k
    // Store the sum of elements
    for (int i = 0; i < k; i++)
        sum += arr[i];

    // Print the current sum
    System.out.print(sum+ " ");

    // Consider every subarray of size k
    // Remove first element and add current
    // element to the window
    for (int i = k; i < n; i++) {

        // Add the element which enters
        // into the window and subtract
        // the element which pops out from
        // the window of the size K
        sum = (sum - arr[i - k]) + arr[i];

        // Print the sum of subarray
        System.out.print(sum+ " ");
    }
}
```

22.Find the maximum sum of all subarray of size k

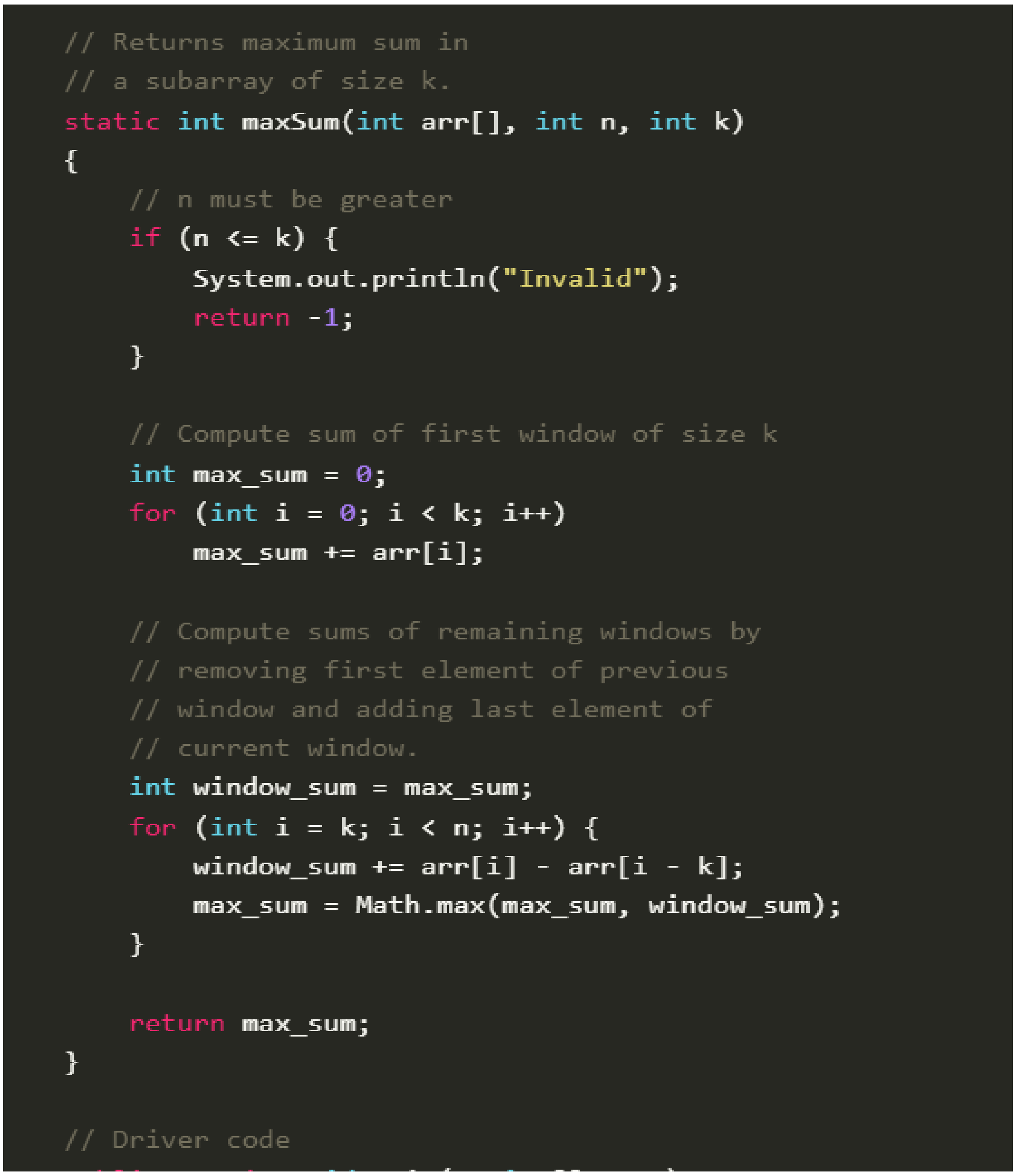


Given an array of integers of size 'n', Our aim is to calculate the maximum sum of 'k' consecutive elements in the array.

***Input** : arr[] = {100, 200, 300, 400}, k = 2*
***Output** : 700*

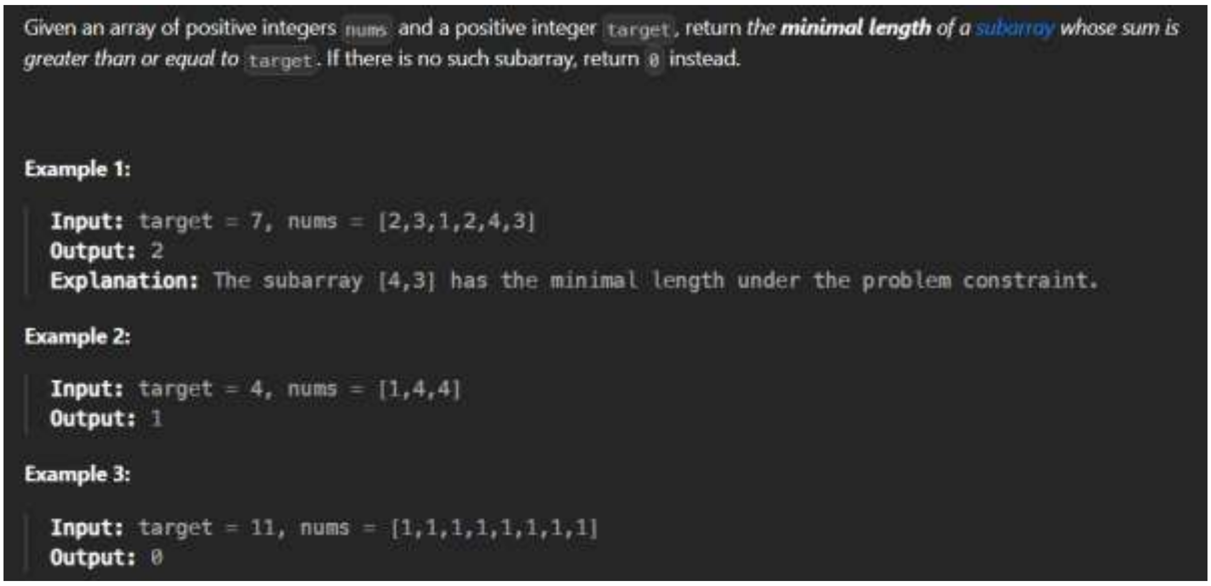
***Input** : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}, k = 4*
***Output** : 39*
We get maximum sum by adding subarray {4, 2, 10, 23} of size 4.

***Input** : arr[] = {2, 3}, k = 3*
***Output** : Invalid*
There is no subarray of size 3 as size of whole array is 2.



```java
// Returns maximum sum in
// a subarray of size k.
static int maxSum(int arr[], int n, int k)
{
    // n must be greater
    if (n <= k) {
        System.out.println("Invalid");
        return -1;
    }

    // Compute sum of first window of size k
    int max_sum = 0;
    for (int i = 0; i < k; i++)
        max_sum += arr[i];

    // Compute sums of remaining windows by
    // removing first element of previous
    // window and adding last element of
    // current window.
    int window_sum = max_sum;
    for (int i = k; i < n; i++) {
        window_sum += arr[i] - arr[i - k];
        max_sum = Math.max(max_sum, window_sum);
    }

    return max_sum;
}

// Driver code
```

23.Minimum size subarray sum



Given an array of positive integers `nums` and a positive integer `target`, return *the* ***minimal length*** *of a subarray whose sum is greater than or equal to* `target`. If there is no such subarray, return `0` instead.

Example 1:

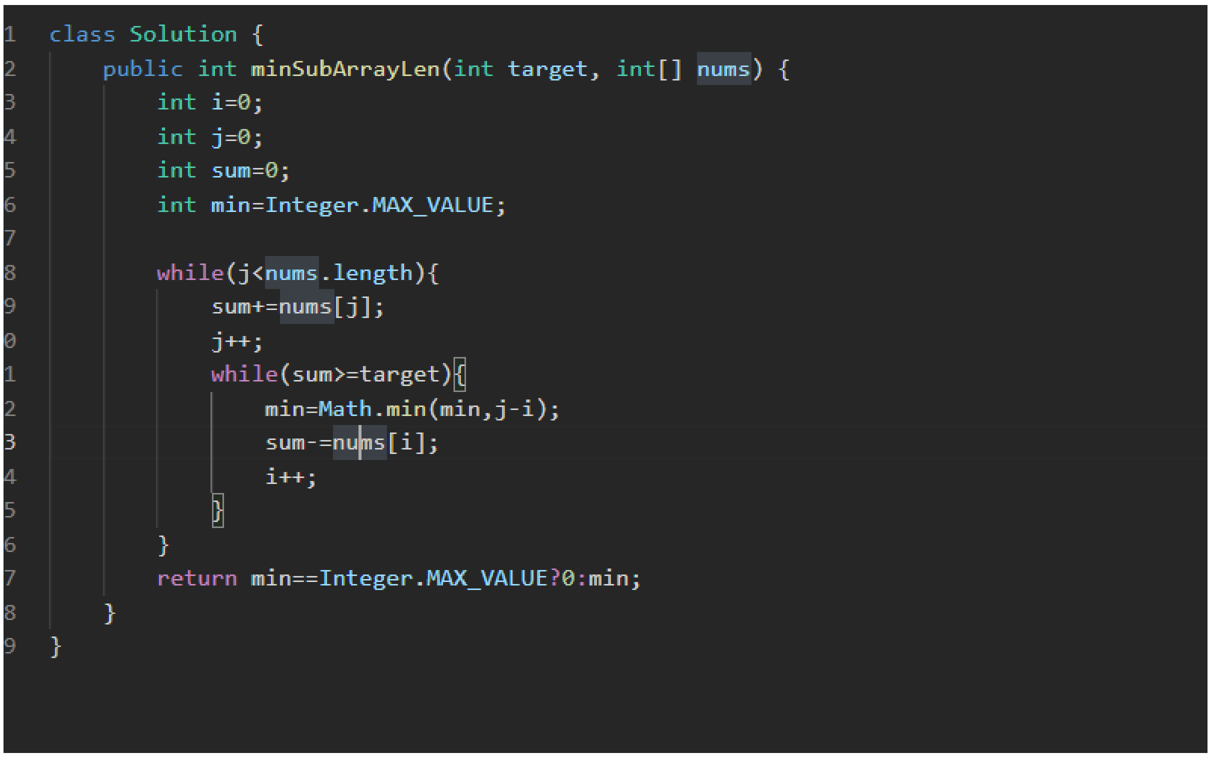
```
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
```

Example 2:

```
Input: target = 4, nums = [1,4,4]
Output: 1
```

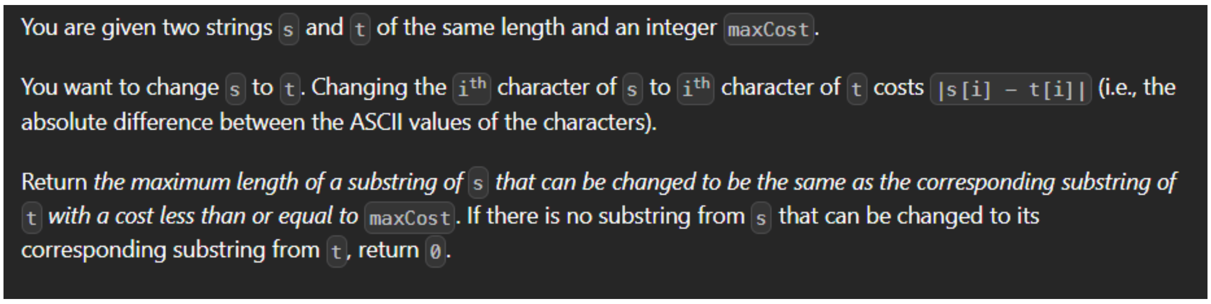
Example 3:

```
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
```



```java
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int i=0;
        int j=0;
        int sum=0;
        int min=Integer.MAX_VALUE;

        while(j<nums.length){
            sum+=nums[j];
            j++;
            while(sum>=target){
                min=Math.min(min,j-i);
                sum-=nums[i];
                i++;
            }
        }
        return min==Integer.MAX_VALUE?0:min;
    }
}
```

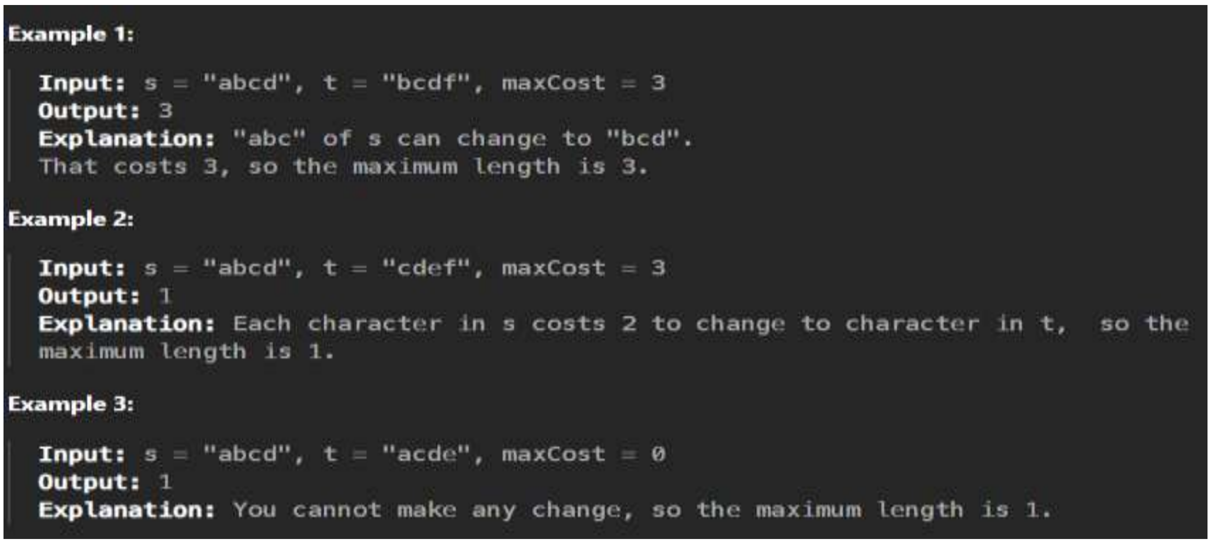
24.Get Equal Substring within budget



You are given two strings `s` and `t` of the same length and an integer `maxCost`.

You want to change `s` to `t`. Changing the `i`th character of `s` to `i`th character of `t` costs `|s[i] - t[i]|` (i.e., the absolute difference between the ASCII values of the characters).

Return *the maximum length of a substring of* `s` *that can be changed to be the same as the corresponding substring of* `t` *with a cost less than or equal to* `maxCost`. If there is no substring from `s` that can be changed to its corresponding substring from `t`, return `0`.



Example 1:

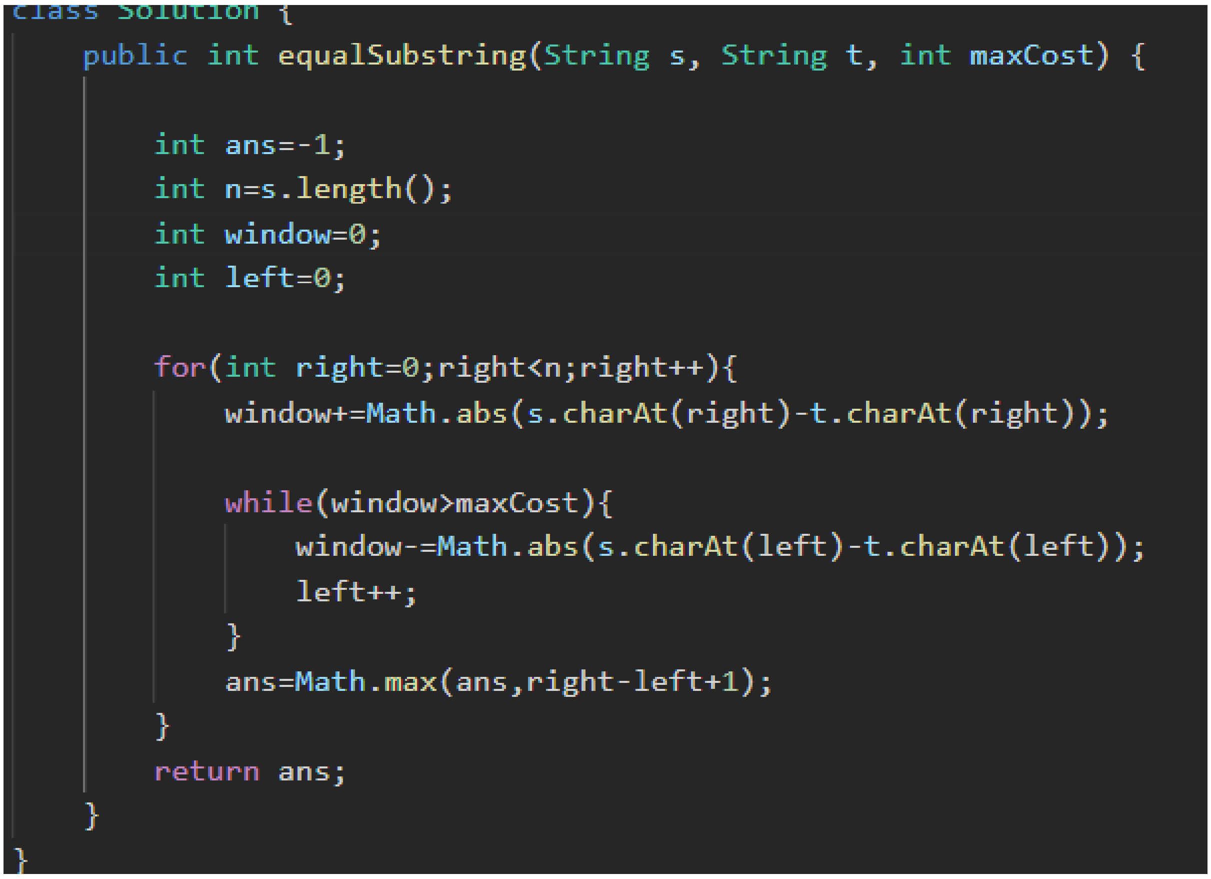
```
Input: s = "abcd", t = "bcdf", maxCost = 3
Output: 3
Explanation: "abc" of s can change to "bcd".
That costs 3, so the maximum length is 3.
```

Example 2:

```
Input: s = "abcd", t = "cdef", maxCost = 3
Output: 1
Explanation: Each character in s costs 2 to change to character in t,  so the
maximum length is 1.
```

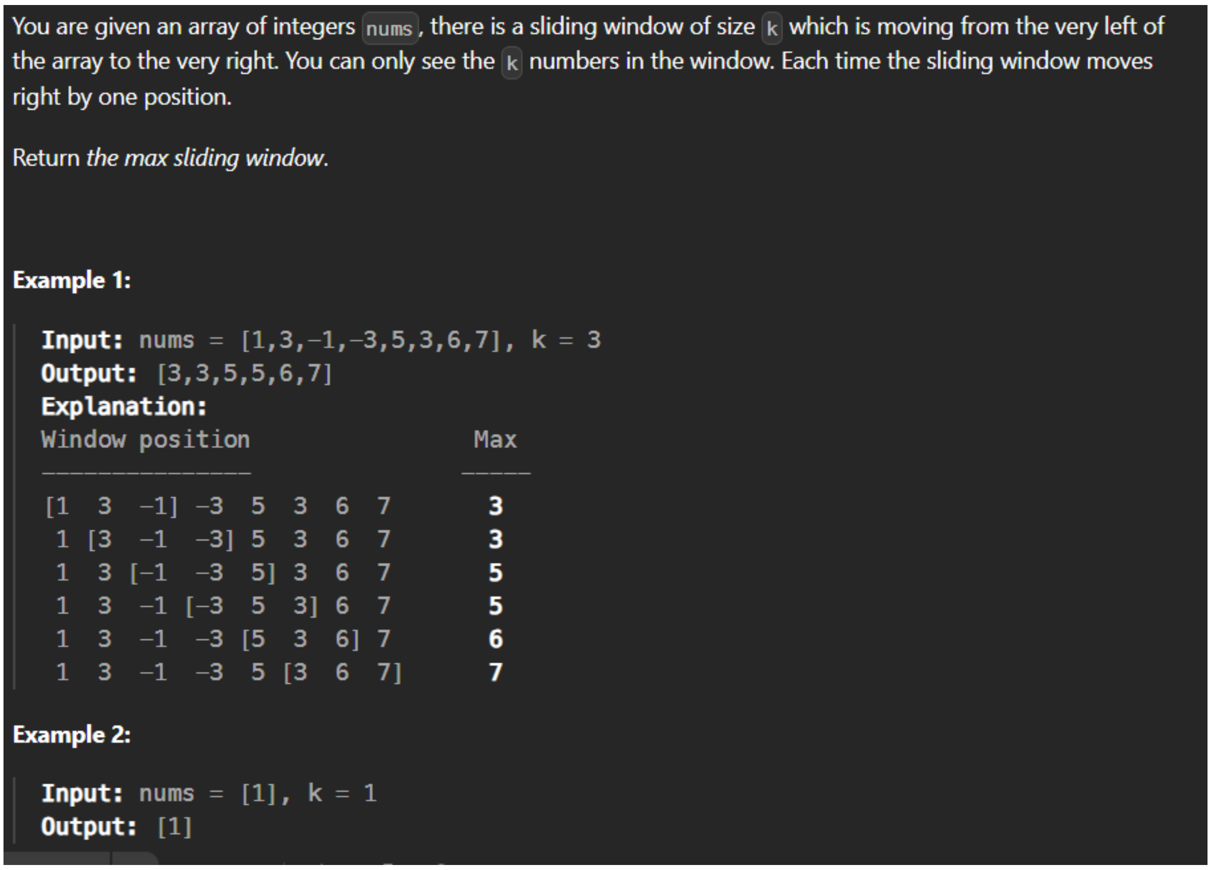
Example 3:

```
Input: s = "abcd", t = "acde", maxCost = 0
Output: 1
Explanation: You cannot make any change, so the maximum length is 1.
```



```java
class Solution {
    public int equalSubstring(String s, String t, int maxCost) {

        int ans=-1;
        int n=s.length();
        int window=0;
        int left=0;

        for(int right=0;right<n;right++){
            window+=Math.abs(s.charAt(right)-t.charAt(right));

            while(window>maxCost){
                window-=Math.abs(s.charAt(left)-t.charAt(left));
                left++;
            }
            ans=Math.max(ans,right-left+1);
        }
        return ans;
    }
}
```

25.Sliding window Maximum



You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

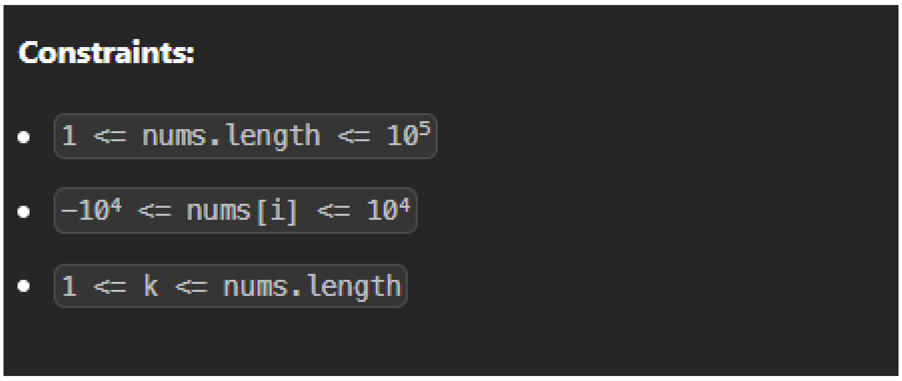
Return *the max sliding window*.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

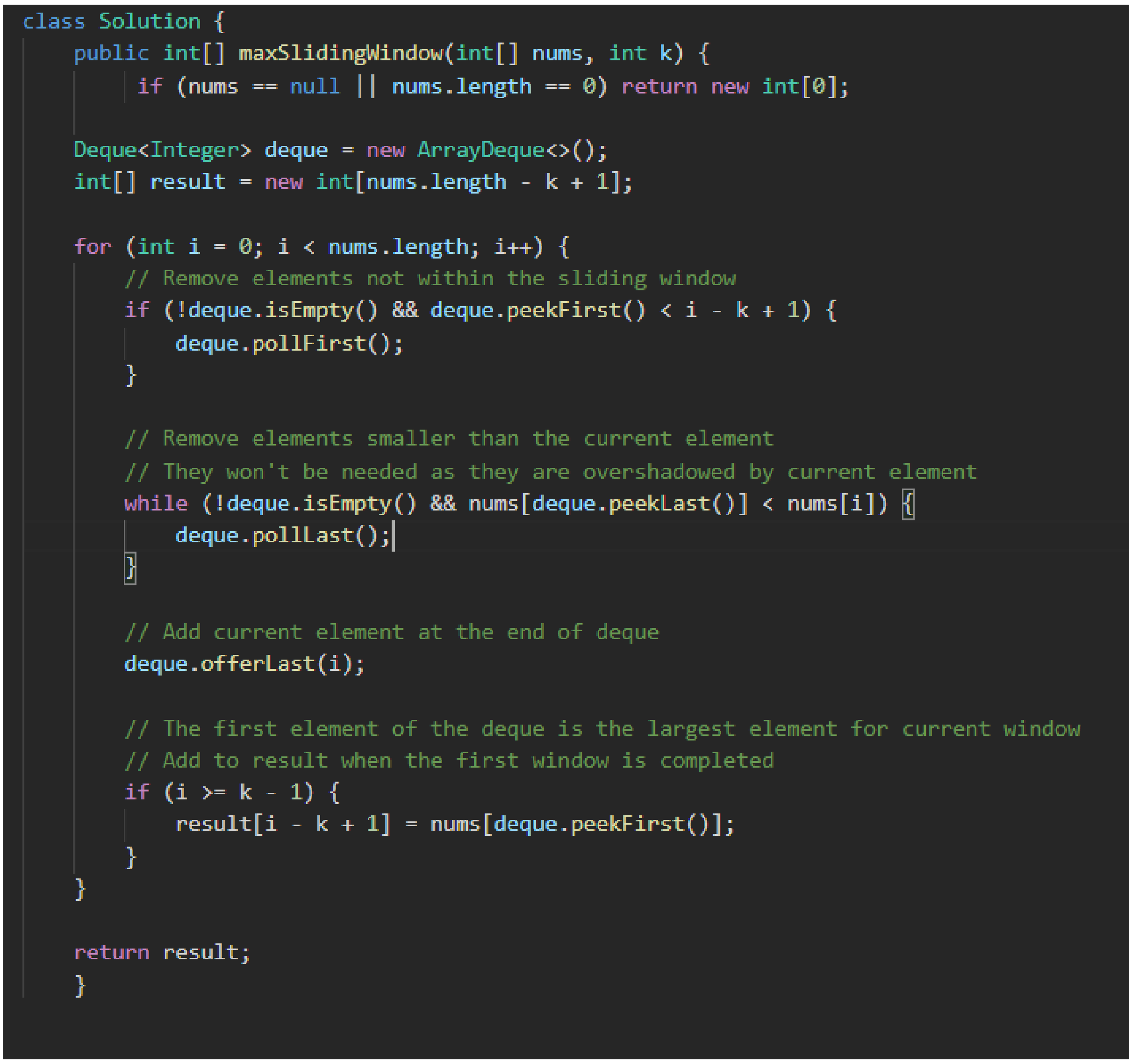
Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

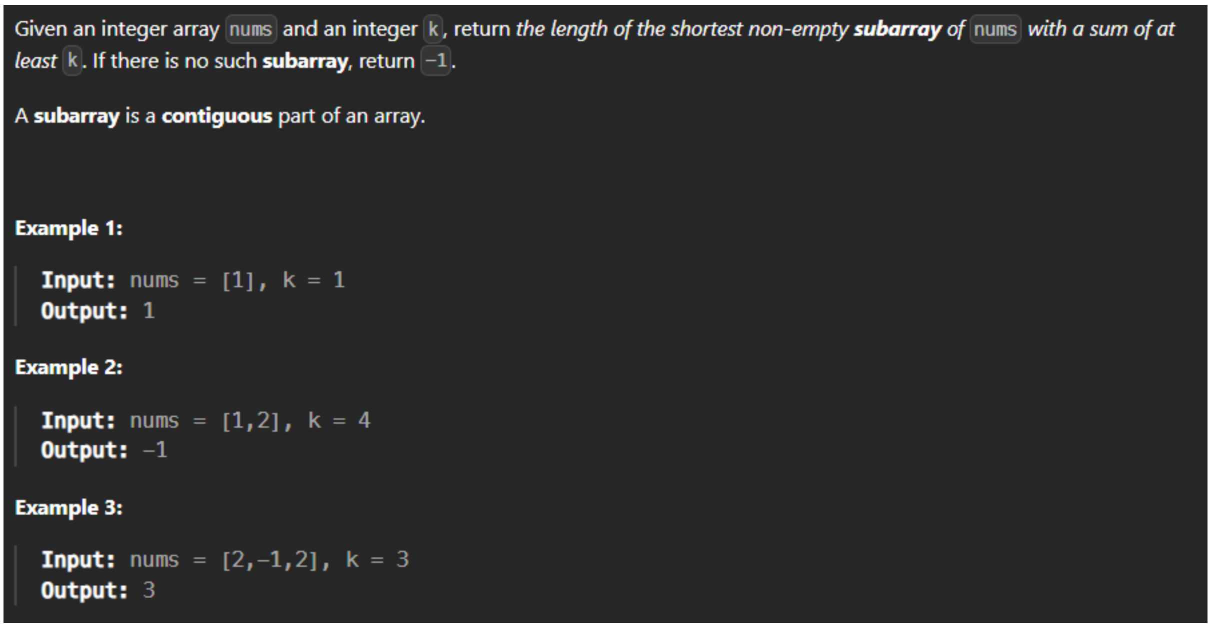


Constraints:

- `1 <= nums.length <=` 10^5
- -10^4 `<= nums[i] <=` 10^4
- `1 <= k <= nums.length`


```java
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums == null || nums.length == 0) return new int[0];

    Deque<Integer> deque = new ArrayDeque<>();
    int[] result = new int[nums.length - k + 1];

    for (int i = 0; i < nums.length; i++) {
        // Remove elements not within the sliding window
        if (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
            deque.pollFirst();
        }

        // Remove elements smaller than the current element
        // They won't be needed as they are overshadowed by current element
        while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
            deque.pollLast();
        }

        // Add current element at the end of deque
        deque.offerLast(i);

        // The first element of the deque is the largest element for current window
        // Add to result when the first window is completed
        if (i >= k - 1) {
            result[i - k + 1] = nums[deque.peekFirst()];
        }
    }

    return result;
    }
```

26.Shortest subarray with sum atleast k



Given an integer array `nums` and an integer `k`, return *the length of the shortest non-empty* ***subarray*** *of* `nums` *with a sum of at least* `k`. If there is no such **subarray**, return `-1`.

A **subarray** is a **contiguous** part of an array.

Example 1:

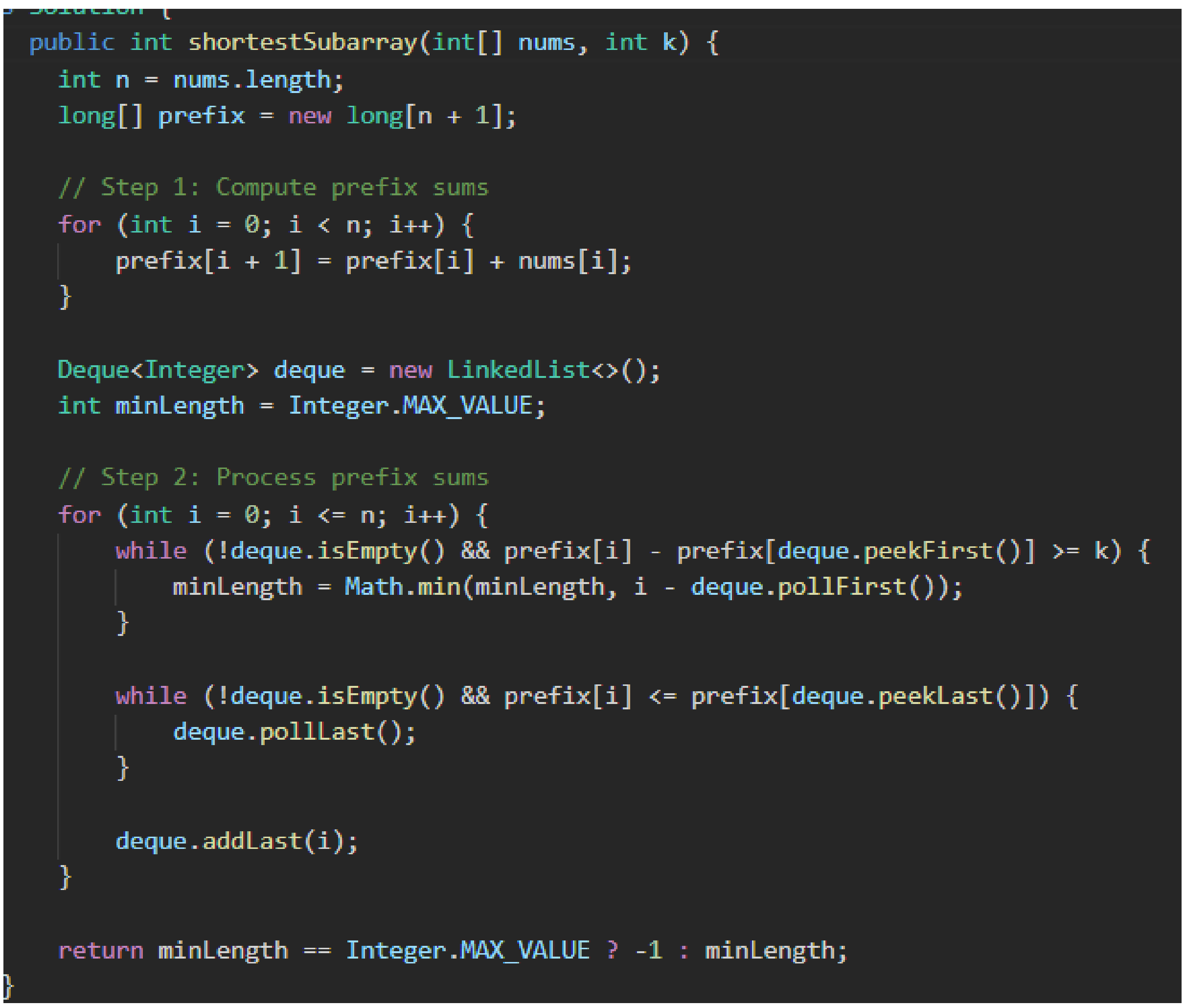
```
Input: nums = [1], k = 1
Output: 1
```

Example 2:

```
Input: nums = [1,2], k = 4
Output: -1
```

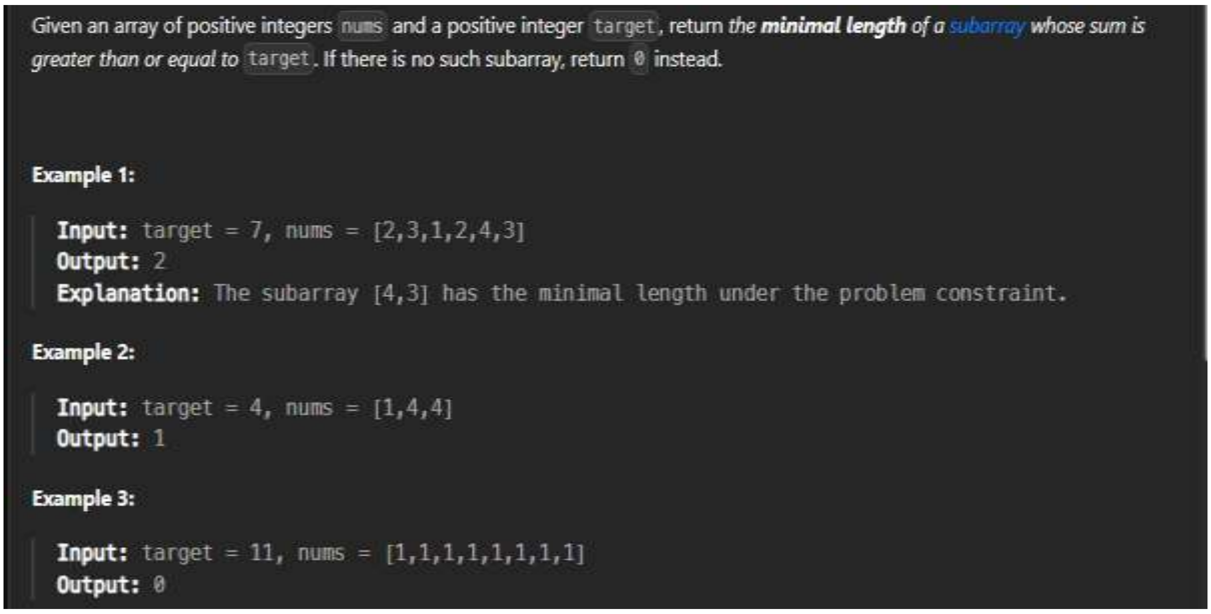
Example 3:

```
Input: nums = [2,-1,2], k = 3
Output: 3
```



```java
  public int shortestSubarray(int[] nums, int k) {
    int n = nums.length;
    long[] prefix = new long[n + 1];

    // Step 1: Compute prefix sums
    for (int i = 0; i < n; i++) {
        prefix[i + 1] = prefix[i] + nums[i];
    }

    Deque<Integer> deque = new LinkedList<>();
    int minLength = Integer.MAX_VALUE;

    // Step 2: Process prefix sums
    for (int i = 0; i <= n; i++) {
        while (!deque.isEmpty() && prefix[i] - prefix[deque.peekFirst()] >= k) {
            minLength = Math.min(minLength, i - deque.pollFirst());
        }

        while (!deque.isEmpty() && prefix[i] <= prefix[deque.peekLast()]) {
            deque.pollLast();
        }

        deque.addLast(i);
    }

    return minLength == Integer.MAX_VALUE ? -1 : minLength;
}
```

27.Minimum size subarray sum



Given an array of positive integers `nums` and a positive integer `target`, return *the **minimal length** of a subarray whose sum is greater than or equal to* `target`. If there is no such subarray, return `0` instead.

Example 1:

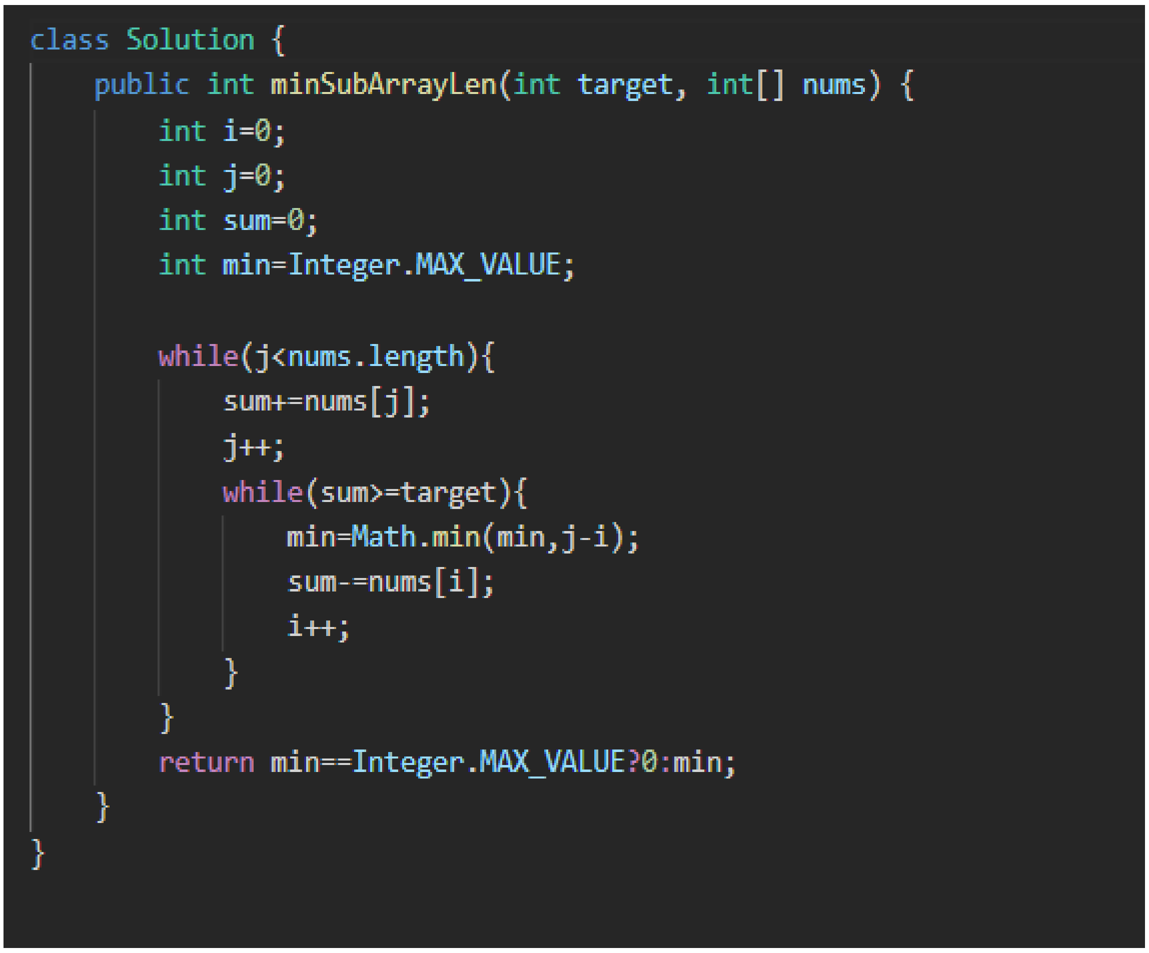
```
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
```

Example 2:

```
Input: target = 4, nums = [1,4,4]
Output: 1
```

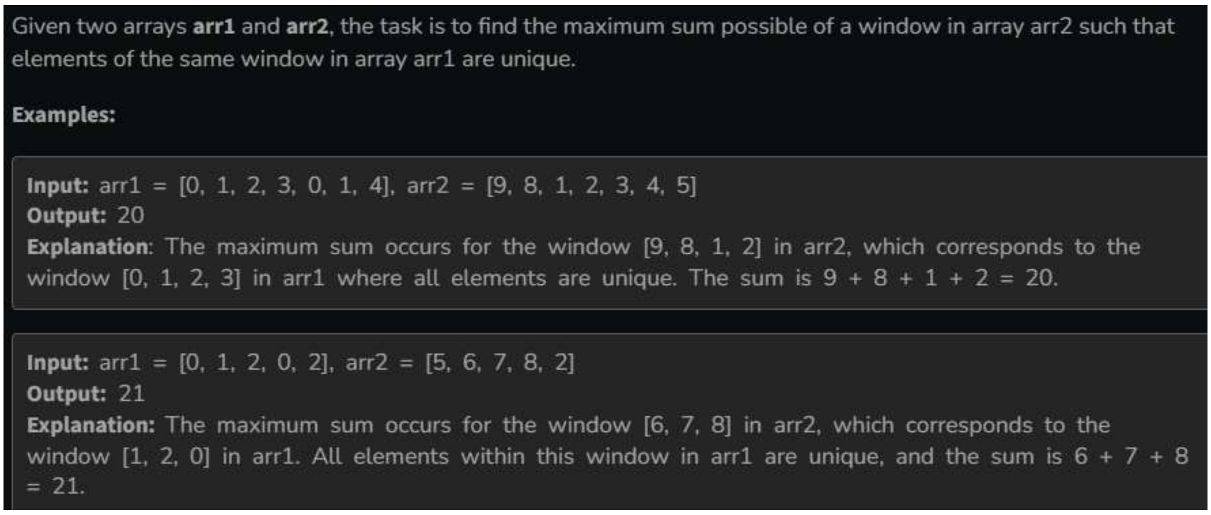
Example 3:

```
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
```



```java
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int i=0;
        int j=0;
        int sum=0;
        int min=Integer.MAX_VALUE;

        while(j<nums.length){
            sum+=nums[j];
            j++;
            while(sum>=target){
                min=Math.min(min,j-i);
                sum-=nums[i];
                i++;
            }
        }
        return min==Integer.MAX_VALUE?0:min;
    }
}
```

28.Maximum Possible sum

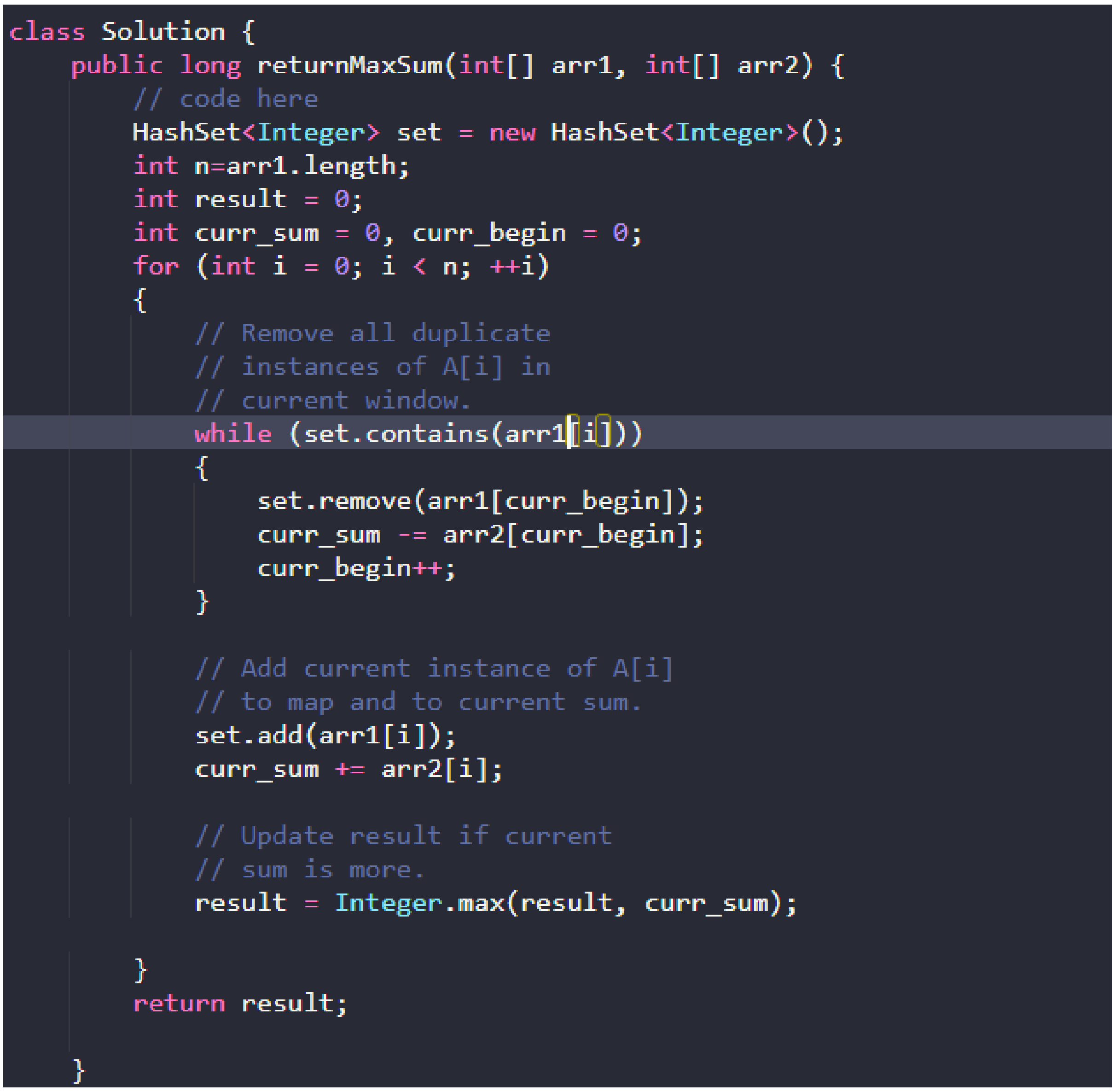


Given two arrays **arr1** and **arr2**, the task is to find the maximum sum possible of a window in array arr2 such that elements of the same window in array arr1 are unique.

Examples:

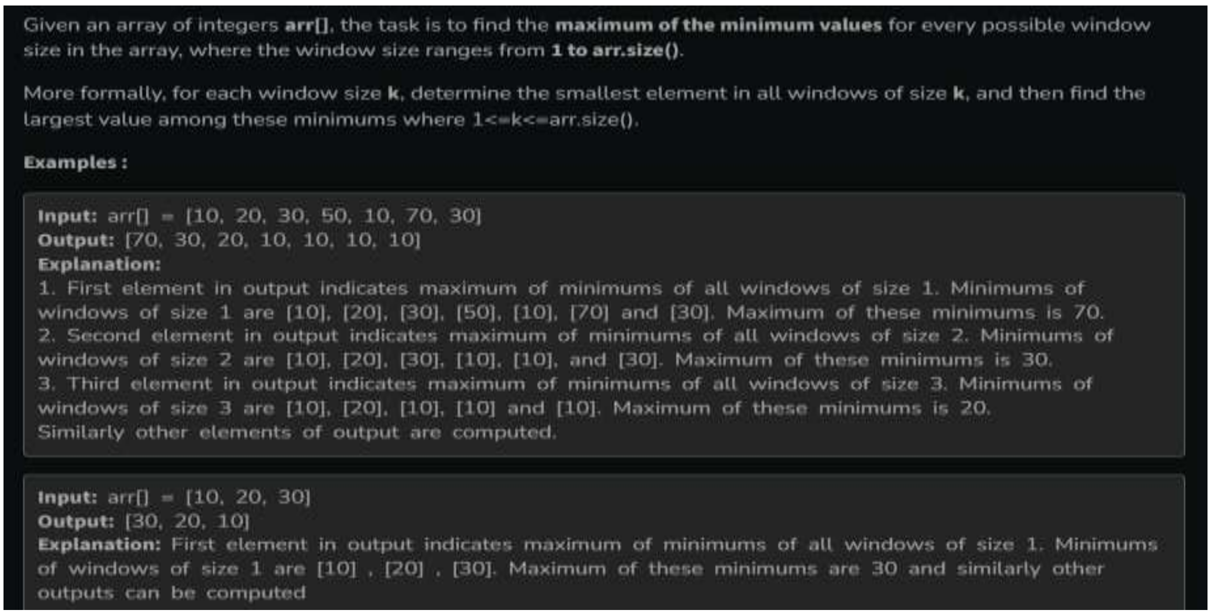
Input: arr1 = [0, 1, 2, 3, 0, 1, 4], arr2 = [9, 8, 1, 2, 3, 4, 5]
Output: 20
Explanation: The maximum sum occurs for the window [9, 8, 1, 2] in arr2, which corresponds to the window [0, 1, 2, 3] in arr1 where all elements are unique. The sum is 9 + 8 + 1 + 2 = 20.

Input: arr1 = [0, 1, 2, 0, 2], arr2 = [5, 6, 7, 8, 2]
Output: 21
Explanation: The maximum sum occurs for the window [6, 7, 8] in arr2, which corresponds to the window [1, 2, 0] in arr1. All elements within this window in arr1 are unique, and the sum is 6 + 7 + 8 = 21.



```java
class Solution {
    public long returnMaxSum(int[] arr1, int[] arr2) {
        // code here
        HashSet<Integer> set = new HashSet<Integer>();
        int n=arr1.length;
        int result = 0;
        int curr_sum = 0, curr_begin = 0;
        for (int i = 0; i < n; ++i)
        {
            // Remove all duplicate
            // instances of A[i] in
            // current window.
            while (set.contains(arr1[i]))
            {
                set.remove(arr1[curr_begin]);
                curr_sum -= arr2[curr_begin];
                curr_begin++;
            }

            // Add current instance of A[i]
            // to map and to current sum.
            set.add(arr1[i]);
            curr_sum += arr2[i];

            // Update result if current
            // sum is more.
            result = Integer.max(result, curr_sum);

        }
        return result;

    }
```

29.Max of min for every window size



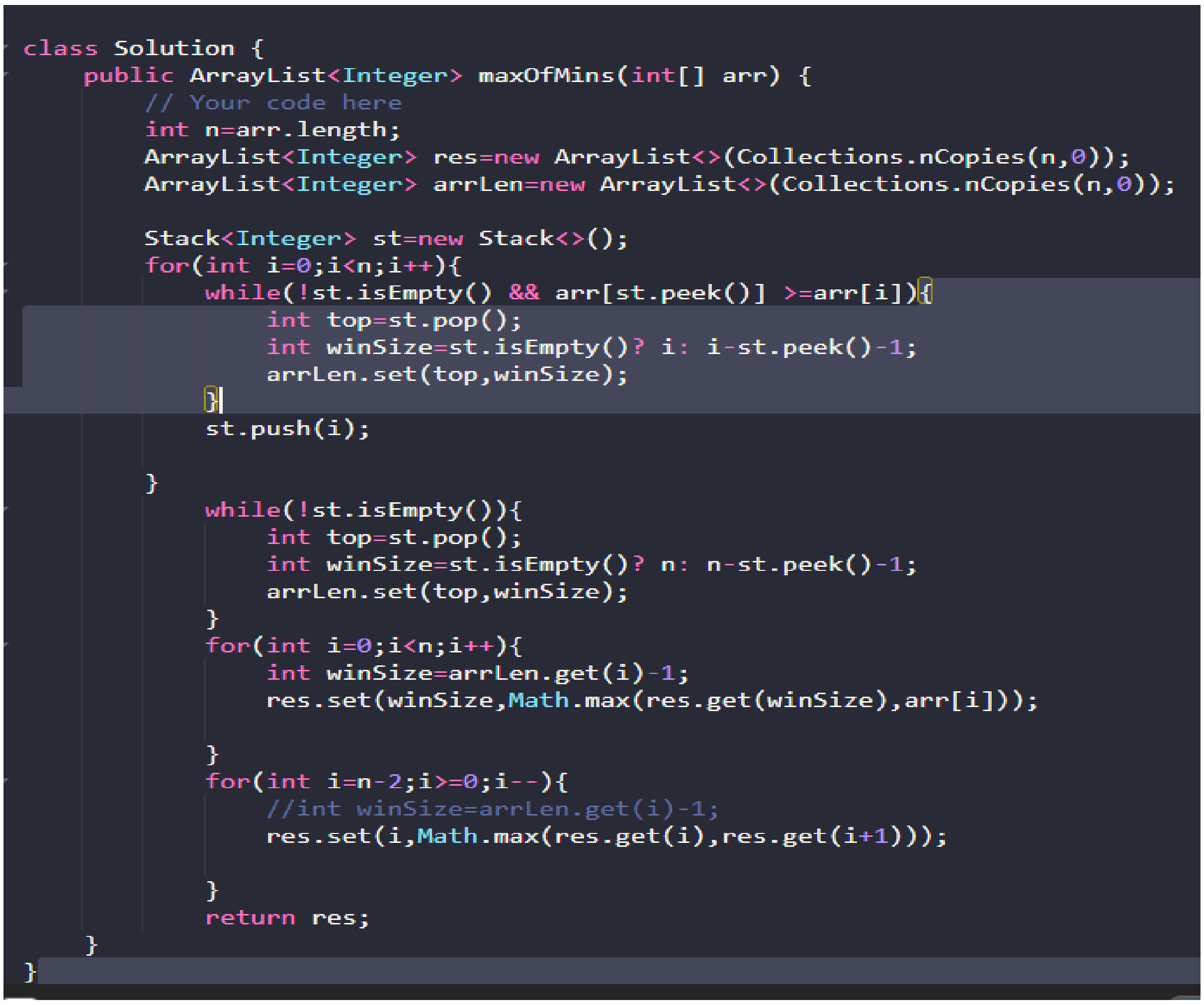
Given an array of integers **arr[]**, the task is to find the **maximum of the minimum values** for every possible window size in the array, where the window size ranges from **1 to arr.size()**.

More formally, for each window size **k**, determine the smallest element in all windows of size **k**, and then find the largest value among these minimums where 1<=k<=arr.size().

Examples :

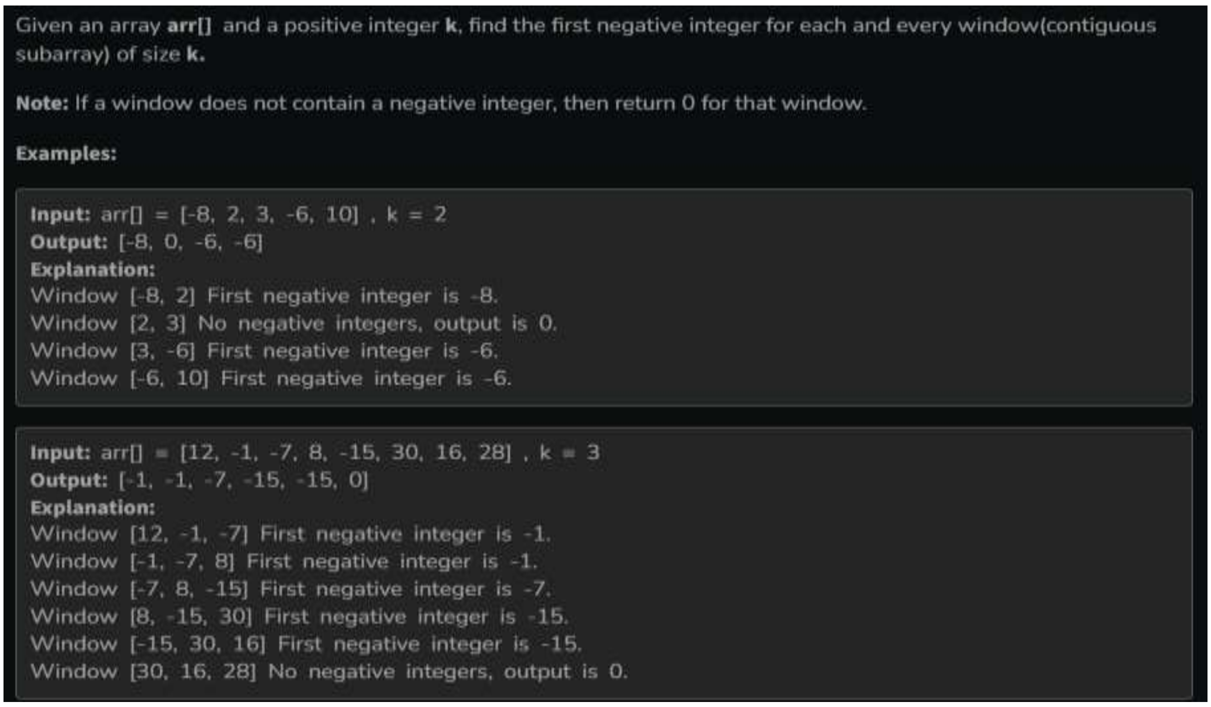
```
Input: arr[] = [10, 20, 30, 50, 10, 70, 30]
Output: [70, 30, 20, 10, 10, 10, 10]
Explanation:
1. First element in output indicates maximum of minimums of all windows of size 1. Minimums of
windows of size 1 are [10], [20], [30], [50], [10], [70] and [30]. Maximum of these minimums is 70.
2. Second element in output indicates maximum of minimums of all windows of size 2. Minimums of
windows of size 2 are [10], [20], [30], [10], [10], and [30]. Maximum of these minimums is 30.
3. Third element in output indicates maximum of minimums of all windows of size 3. Minimums of
windows of size 3 are [10], [20], [10], [10] and [10]. Maximum of these minimums is 20.
Similarly other elements of output are computed.
```

```
Input: arr[] = [10, 20, 30]
Output: [30, 20, 10]
Explanation: First element in output indicates maximum of minimums of all windows of size 1. Minimums
of windows of size 1 are [10] , [20] , [30]. Maximum of these minimums are 30 and similarly other
outputs can be computed
```



```java
class Solution {
    public ArrayList<Integer> maxOfMins(int[] arr) {
        // Your code here
        int n=arr.length;
        ArrayList<Integer> res=new ArrayList<>(Collections.nCopies(n,0));
        ArrayList<Integer> arrLen=new ArrayList<>(Collections.nCopies(n,0));

        Stack<Integer> st=new Stack<>();
        for(int i=0;i<n;i++){
            while(!st.isEmpty() && arr[st.peek()] >=arr[i]){
                int top=st.pop();
                int winSize=st.isEmpty()? i: i-st.peek()-1;
                arrLen.set(top,winSize);
            }
            st.push(i);

        }
            while(!st.isEmpty()){
                int top=st.pop();
                int winSize=st.isEmpty()? n: n-st.peek()-1;
                arrLen.set(top,winSize);
            }
            for(int i=0;i<n;i++){
                int winSize=arrLen.get(i)-1;
                res.set(winSize,Math.max(res.get(winSize),arr[i]));

            }
            for(int i=n-2;i>=0;i--){
                //int winSize=arrLen.get(i)-1;
                res.set(i,Math.max(res.get(i),res.get(i+1)));

            }
            return res;
    }
}
```

30.First negative in every window of size k



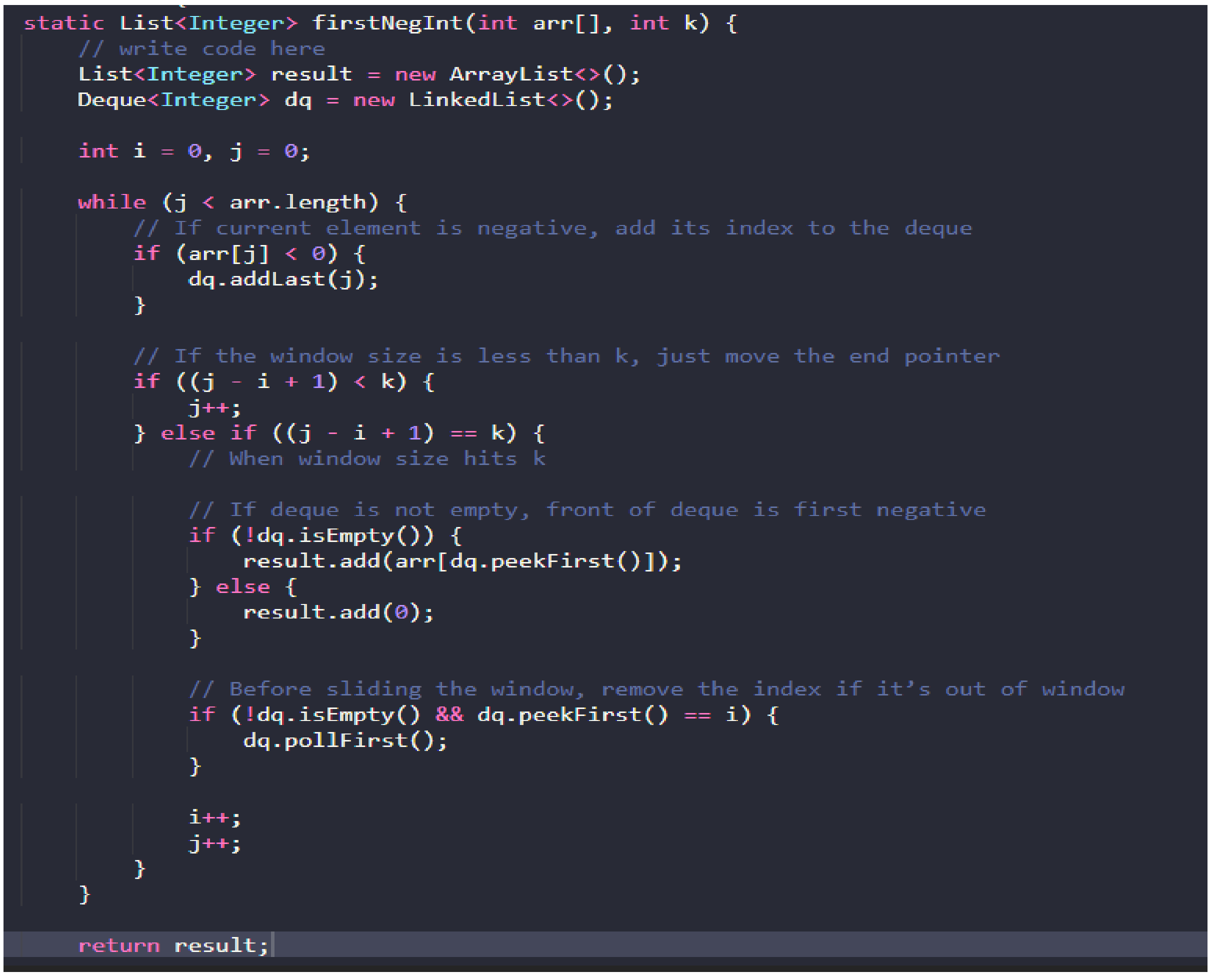
Given an array **arr[]** and a positive integer **k**, find the first negative integer for each and every window(contiguous subarray) of size **k.**

Note: If a window does not contain a negative integer, then return 0 for that window.

Examples:

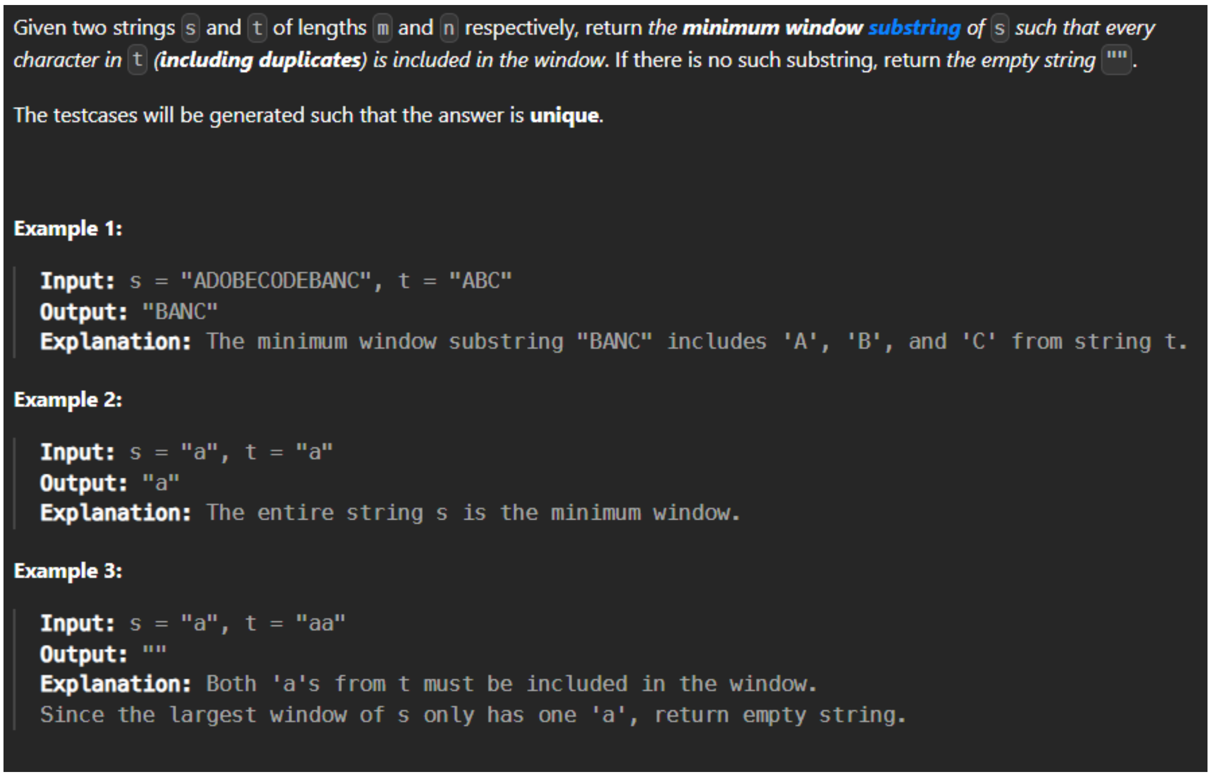
```
Input: arr[] = [-8, 2, 3, -6, 10] , k = 2
Output: [-8, 0, -6, -6]
Explanation:
Window [-8, 2] First negative integer is -8.
Window [2, 3] No negative integers, output is 0.
Window [3, -6] First negative integer is -6.
Window [-6, 10] First negative integer is -6.
```

```
Input: arr[] = [12, -1, -7, 8, -15, 30, 16, 28] , k = 3
Output: [-1, -1, -7, -15, -15, 0]
Explanation:
Window [12, -1, -7] First negative integer is -1.
Window [-1, -7, 8] First negative integer is -1.
Window [-7, 8, -15] First negative integer is -7.
Window [8, -15, 30] First negative integer is -15.
Window [-15, 30, 16] First negative integer is -15.
Window [30, 16, 28] No negative integers, output is 0.
```



```java
static List<Integer> firstNegInt(int arr[], int k) {
    // write code here
    List<Integer> result = new ArrayList<>();
    Deque<Integer> dq = new LinkedList<>();

    int i = 0, j = 0;

    while (j < arr.length) {
        // If current element is negative, add its index to the deque
        if (arr[j] < 0) {
            dq.addLast(j);
        }

        // If the window size is less than k, just move the end pointer
        if ((j - i + 1) < k) {
            j++;
        } else if ((j - i + 1) == k) {
            // When window size hits k

            // If deque is not empty, front of deque is first negative
            if (!dq.isEmpty()) {
                result.add(arr[dq.peekFirst()]);
            } else {
                result.add(0);
            }

            // Before sliding the window, remove the index if it’s out of window
            if (!dq.isEmpty() && dq.peekFirst() == i) {
                dq.pollFirst();
            }

            i++;
            j++;
        }
    }

    return result;
```

31.Minimum window substring



Given two strings `s` and `t` of lengths `m` and `n` respectively, return *the* ***minimum window substring*** *of* `s` *such that every character in* `t` ***(including duplicates)*** *is included in the window*. If there is no such substring, return *the empty string* `""`.

The testcases will be generated such that the answer is **unique**.

Example 1:

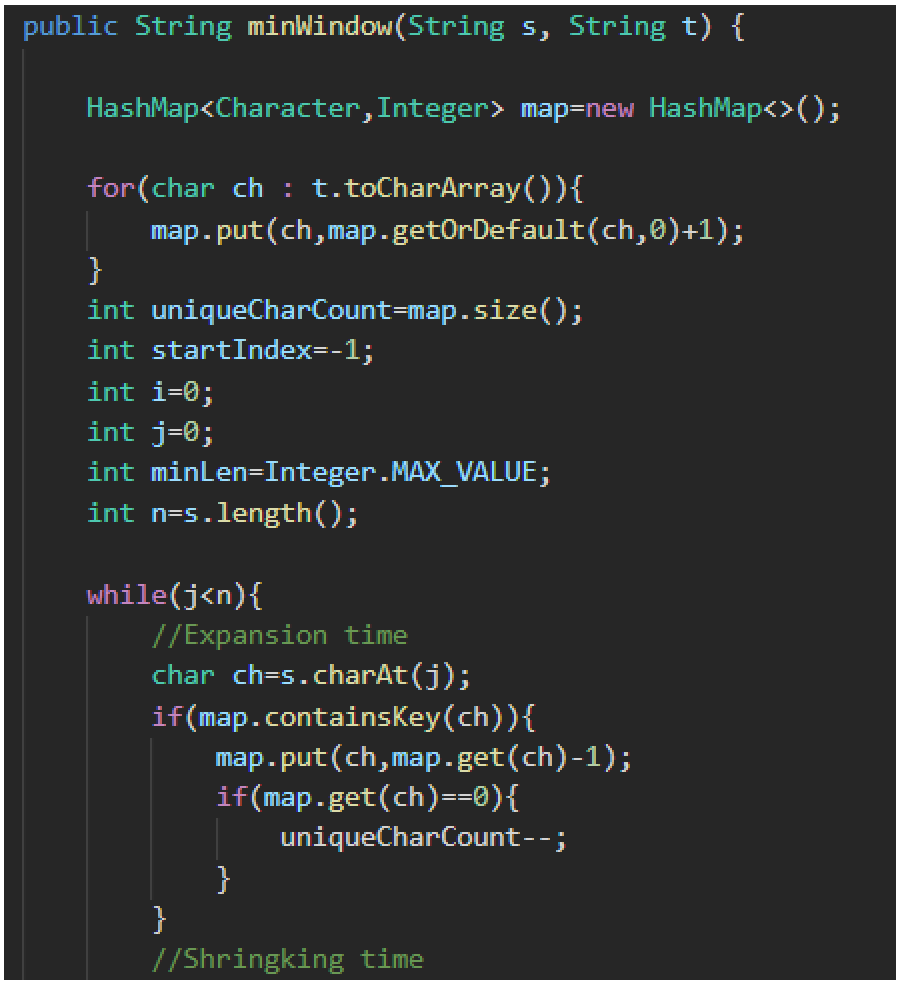
```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
```

Example 2:

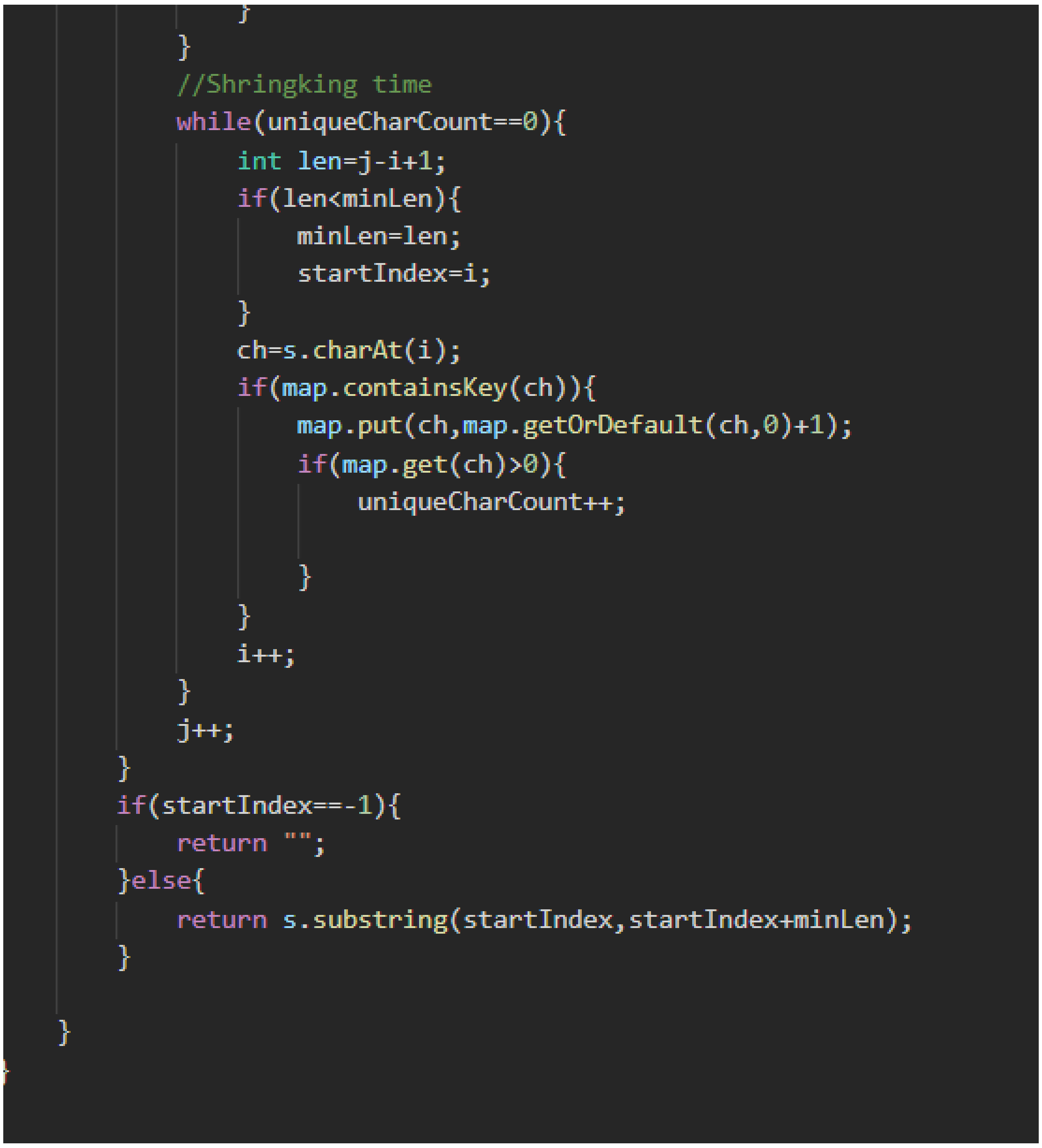
```
Input: s = "a", t = "a"
Output: "a"
Explanation: The entire string s is the minimum window.
```

Example 3:

```
Input: s = "a", t = "aa"
Output: ""
Explanation: Both 'a's from t must be included in the window.
Since the largest window of s only has one 'a', return empty string.
```

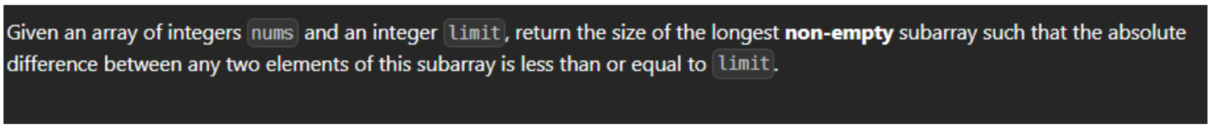


```java
public String minWindow(String s, String t) {

    HashMap<Character,Integer> map=new HashMap<>();

    for(char ch : t.toCharArray()){
        map.put(ch,map.getOrDefault(ch,0)+1);
    }
    int uniqueCharCount=map.size();
    int startIndex=-1;
    int i=0;
    int j=0;
    int minLen=Integer.MAX_VALUE;
    int n=s.length();

    while(j<n){
        //Expansion time
        char ch=s.charAt(j);
        if(map.containsKey(ch)){
            map.put(ch,map.get(ch)-1);
            if(map.get(ch)==0){
                uniqueCharCount--;
            }
        }
        //Shringking time
```

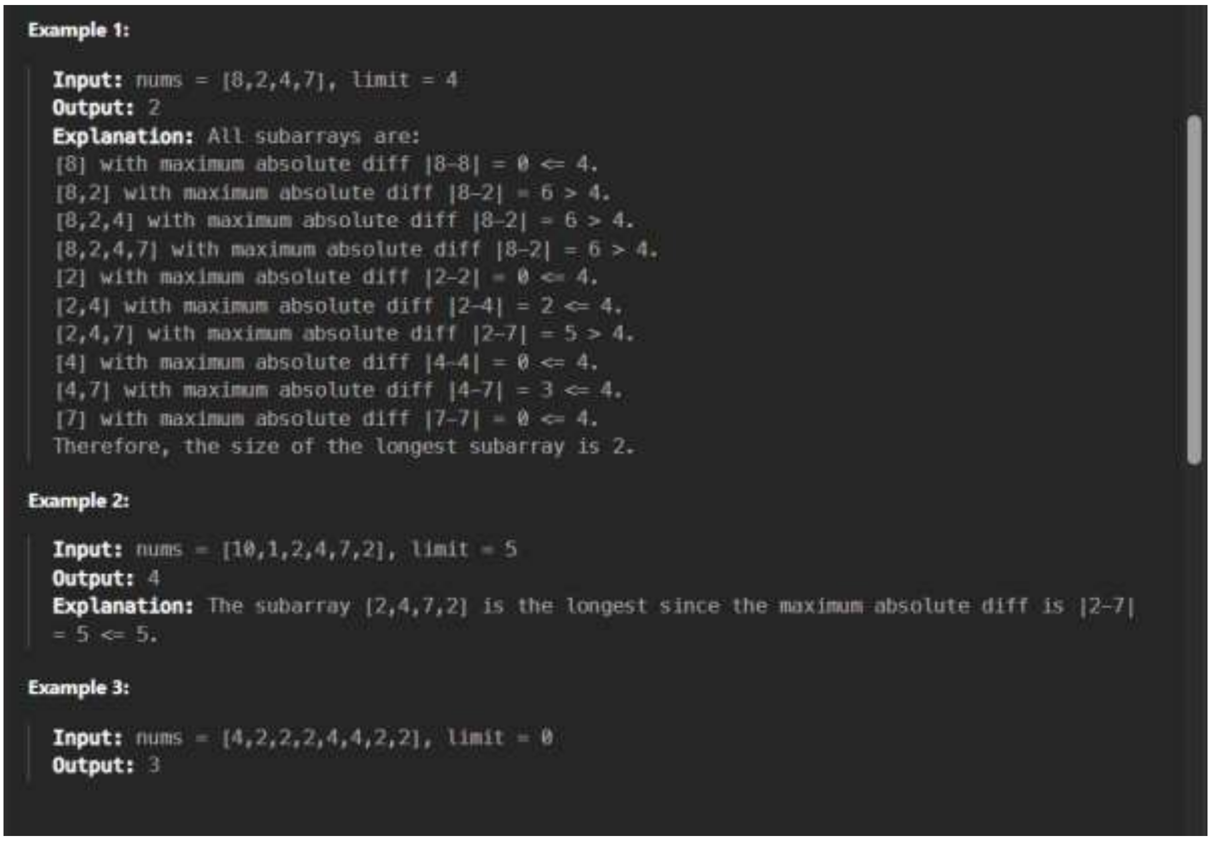


```java
                }
            }
            //Shringking time
            while(uniqueCharCount==0){
                int len=j-i+1;
                if(len<minLen){
                    minLen=len;
                    startIndex=i;
                }
                ch=s.charAt(i);
                if(map.containsKey(ch)){
                    map.put(ch,map.getOrDefault(ch,0)+1);
                    if(map.get(ch)>0){
                        uniqueCharCount++;

                    }
                }
                i++;
            }
            j++;
        }
        if(startIndex==-1){
            return "";
        }else{
            return s.substring(startIndex,startIndex+minLen);
        }

    }
}
```

32.Longest continuous subarray with absolute diff less than or equal to limit



Given an array of integers `nums` and an integer `limit`, return the size of the longest **non-empty** subarray such that the absolute difference between any two elements of this subarray is less than or equal to `limit`.



Example 1:

```
Input: nums = [8,2,4,7], limit = 4
Output: 2
Explanation: All subarrays are:
[8] with maximum absolute diff |8-8| = 0 <= 4.
[8,2] with maximum absolute diff |8-2| = 6 > 4.
[8,2,4] with maximum absolute diff |8-2| = 6 > 4.
[8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.
[2] with maximum absolute diff |2-2| = 0 <= 4.
[2,4] with maximum absolute diff |2-4| = 2 <= 4.
[2,4,7] with maximum absolute diff |2-7| = 5 > 4.
[4] with maximum absolute diff |4-4| = 0 <= 4.
[4,7] with maximum absolute diff |4-7| = 3 <= 4.
[7] with maximum absolute diff |7-7| = 0 <= 4.
Therefore, the size of the longest subarray is 2.
```

Example 2:

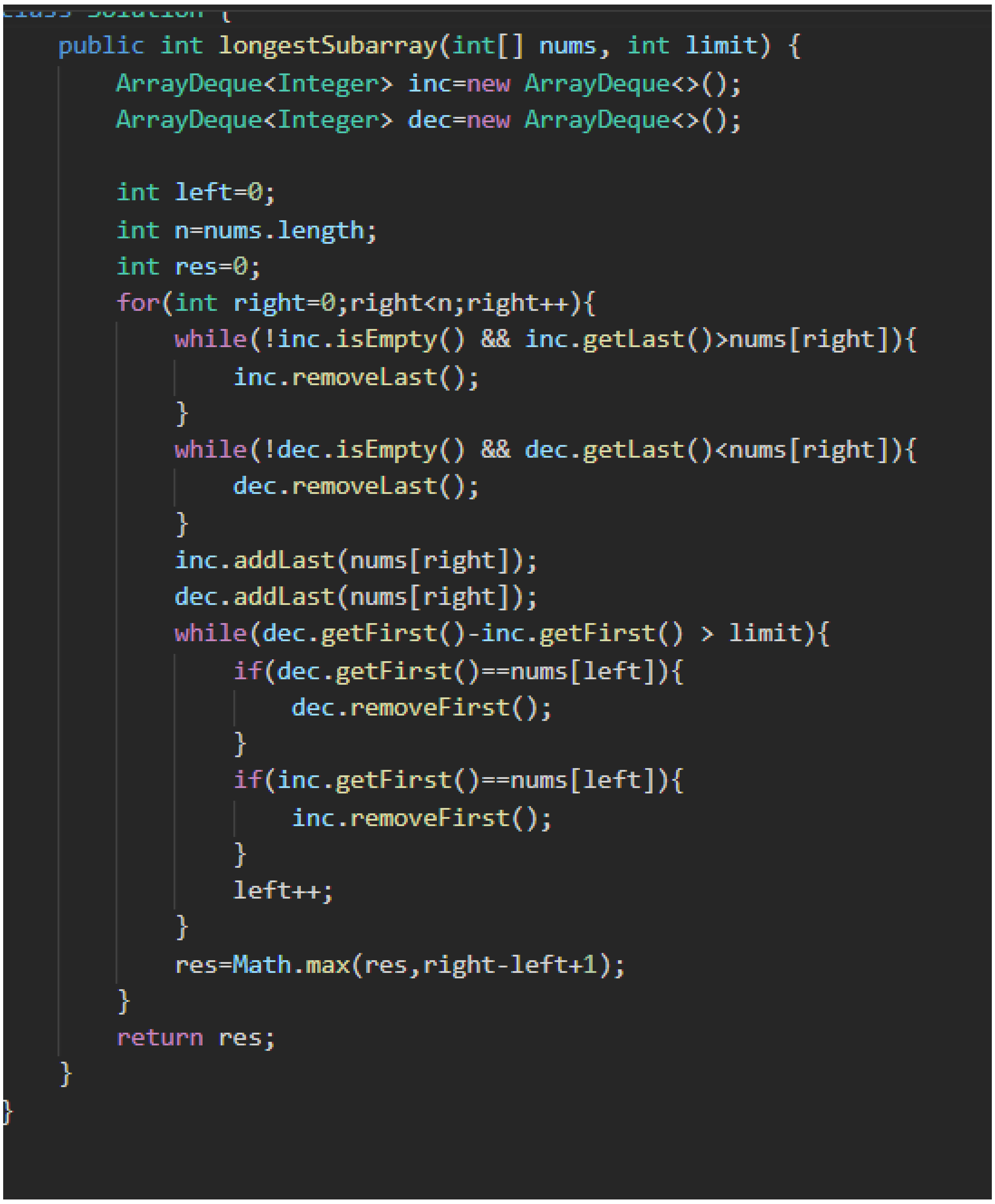
```
Input: nums = [10,1,2,4,7,2], limit = 5
Output: 4
Explanation: The subarray [2,4,7,2] is the longest since the maximum absolute diff is |2-7|
= 5 <= 5.
```

Example 3:

```
Input: nums = [4,2,2,2,4,4,2,2], limit = 0
Output: 3
```

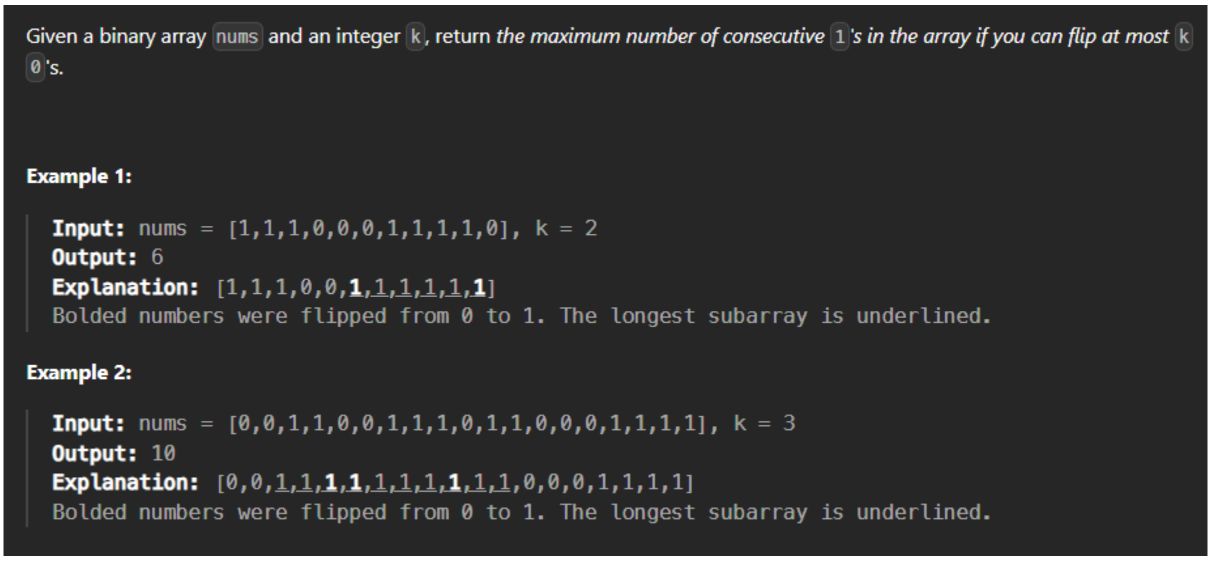
Constraints:

- 1 <= nums.length <= 10^5
- 1 <= nums[i] <= 10^9
- 0 <= limit <= 10^9



```
class Solution {
    public int longestSubarray(int[] nums, int limit) {
        ArrayDeque<Integer> inc=new ArrayDeque<>();
        ArrayDeque<Integer> dec=new ArrayDeque<>();

        int left=0;
        int n=nums.length;
        int res=0;
        for(int right=0;right<n;right++){
            while(!inc.isEmpty() && inc.getLast()>nums[right]){
                inc.removeLast();
            }
            while(!dec.isEmpty() && dec.getLast()<nums[right]){
                dec.removeLast();
            }
            inc.addLast(nums[right]);
            dec.addLast(nums[right]);
            while(dec.getFirst()-inc.getFirst() > limit){
                if(dec.getFirst()==nums[left]){
                    dec.removeFirst();
                }
                if(inc.getFirst()==nums[left]){
                    inc.removeFirst();
                }
                left++;
            }
            res=Math.max(res,right-left+1);
        }
        return res;
    }
}
```

33.Maximum Consecutive ones iii



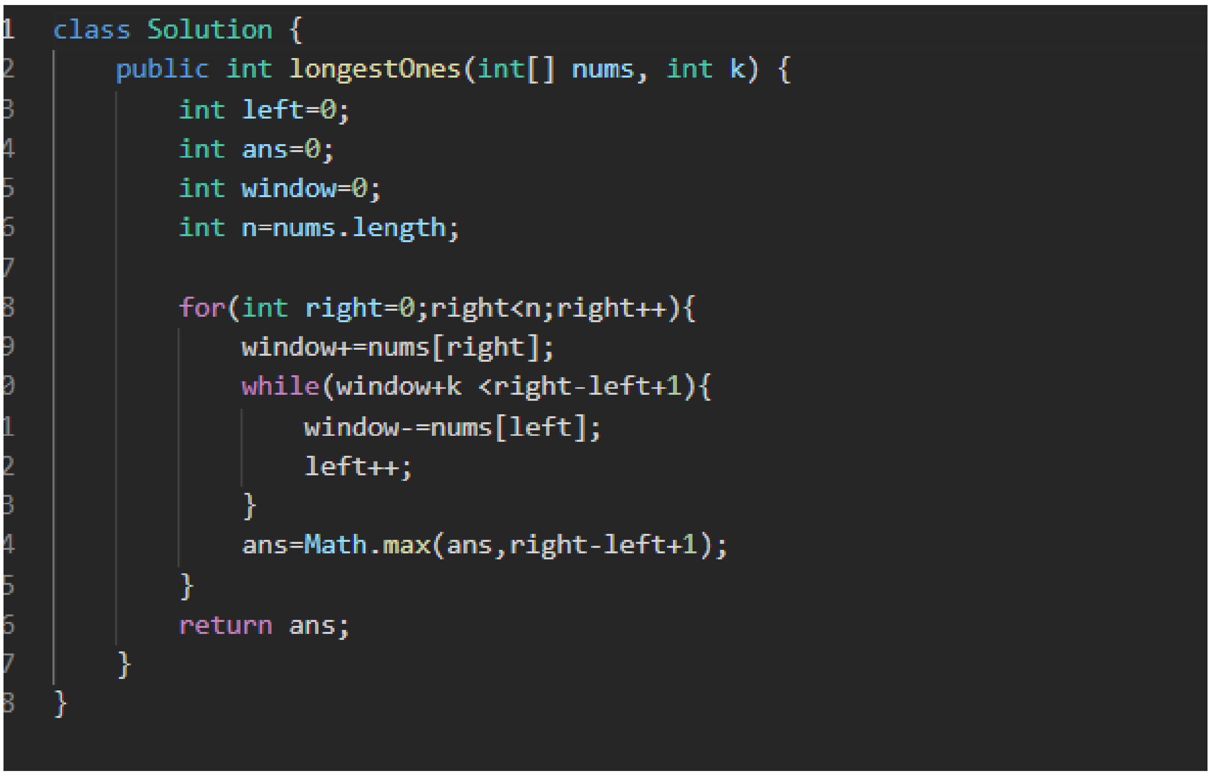
Given a binary array `nums` and an integer `k`, return *the maximum number of consecutive* `1`*'s in the array if you can flip at most* `k` `0`'s.

Example 1:

> **Input:** nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2
> **Output:** 6
> **Explanation:** [1,1,1,0,0,**1**,1,1,1,1,**1**]
> Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

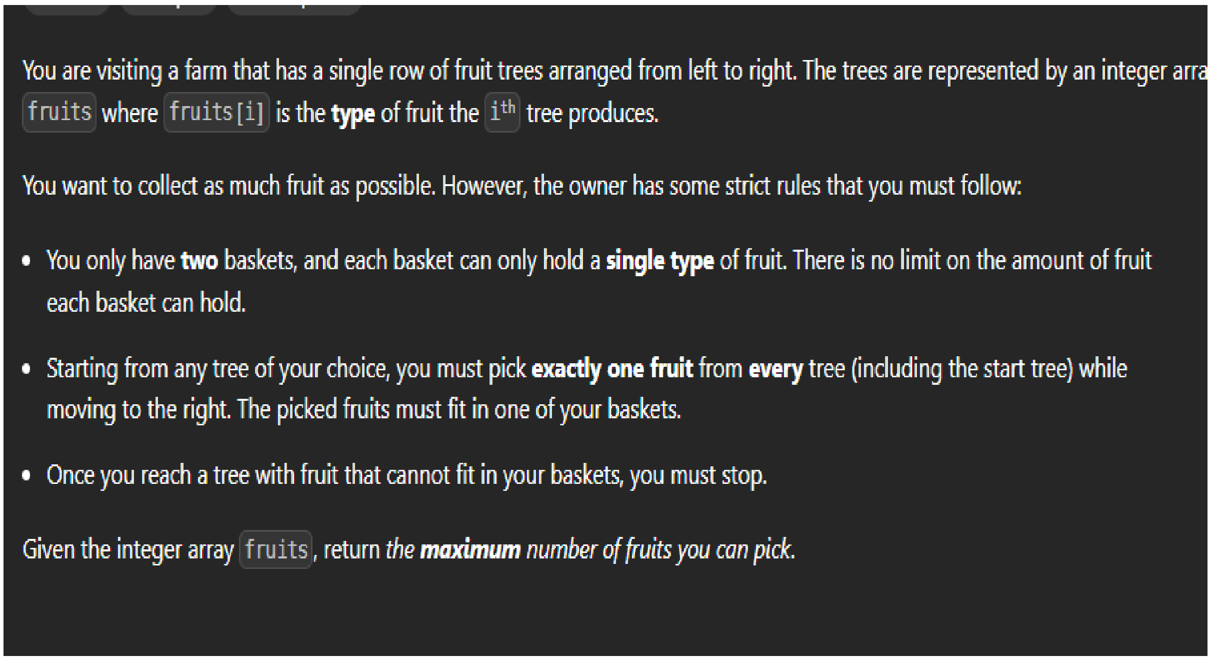
Example 2:

> **Input:** nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], k = 3
> **Output:** 10
> **Explanation:** [0,0,1,1,**1**,**1**,1,1,1,**1**,1,1,0,0,0,1,1,1,1]
> Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.



```java
class Solution {
    public int longestOnes(int[] nums, int k) {
        int left=0;
        int ans=0;
        int window=0;
        int n=nums.length;

        for(int right=0;right<n;right++){
            window+=nums[right];
            while(window+k <right-left+1){
                window-=nums[left];
                left++;
            }
            ans=Math.max(ans,right-left+1);
        }
        return ans;
    }
}
```

34.Fruit into basket

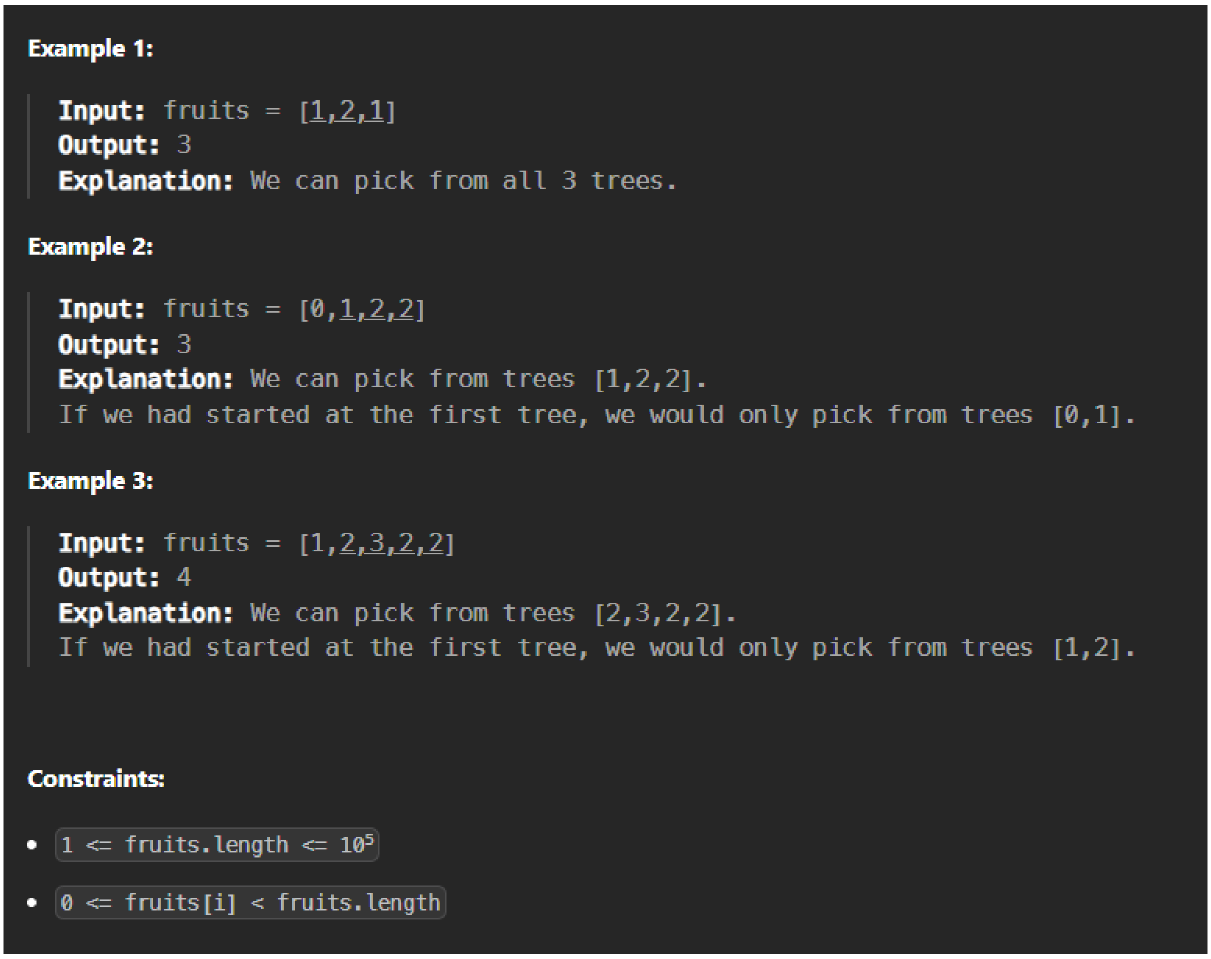


You are visiting a farm that has a single row of fruit trees arranged from left to right. The trees are represented by an integer arra `fruits` where `fruits[i]` is the **type** of fruit the `i`th tree produces.

You want to collect as much fruit as possible. However, the owner has some strict rules that you must follow:

- You only have **two** baskets, and each basket can only hold a **single type** of fruit. There is no limit on the amount of fruit each basket can hold.
- Starting from any tree of your choice, you must pick **exactly one fruit** from **every** tree (including the start tree) while moving to the right. The picked fruits must fit in one of your baskets.
- Once you reach a tree with fruit that cannot fit in your baskets, you must stop.

Given the integer array `fruits`, return *the* ***maximum*** *number of fruits you can pick*.



Example 1:

```
Input: fruits = [1,2,1]
Output: 3
Explanation: We can pick from all 3 trees.
```

Example 2:

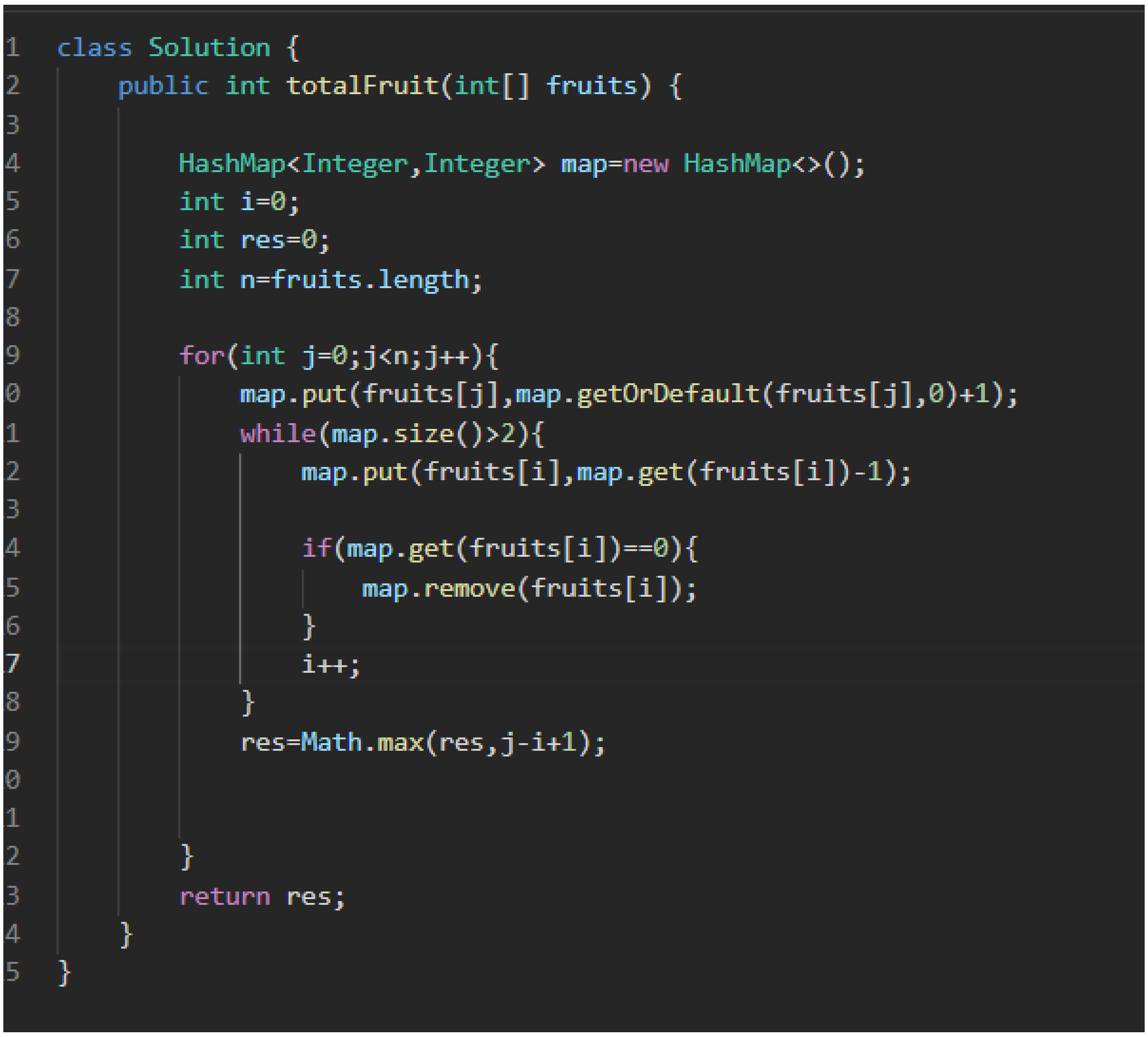
```
Input: fruits = [0,1,2,2]
Output: 3
Explanation: We can pick from trees [1,2,2].
If we had started at the first tree, we would only pick from trees [0,1].
```

Example 3:

```
Input: fruits = [1,2,3,2,2]
Output: 4
Explanation: We can pick from trees [2,3,2,2].
If we had started at the first tree, we would only pick from trees [1,2].
```

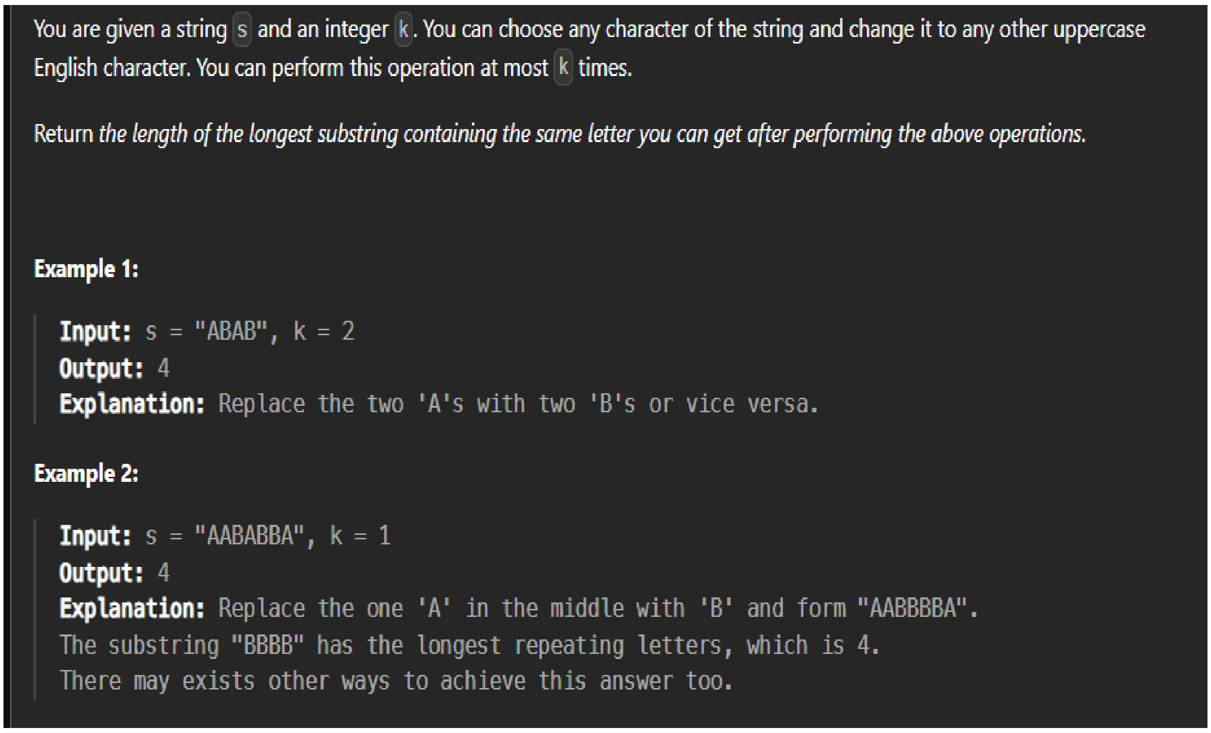
Constraints:

- `1 <= fruits.length <= 10`^{5}
- `0 <= fruits[i] < fruits.length`



```java
class Solution {
    public int totalFruit(int[] fruits) {

        HashMap<Integer,Integer> map=new HashMap<>();
        int i=0;
        int res=0;
        int n=fruits.length;

        for(int j=0;j<n;j++){
            map.put(fruits[j],map.getOrDefault(fruits[j],0)+1);
            while(map.size()>2){
                map.put(fruits[i],map.get(fruits[i])-1);

                if(map.get(fruits[i])==0){
                    map.remove(fruits[i]);
                }
                i++;
            }
            res=Math.max(res,j-i+1);


        }
        return res;
    }
}
```

35.Longest Repeating character Replacement



You are given a string `s` and an integer `k`. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most `k` times.

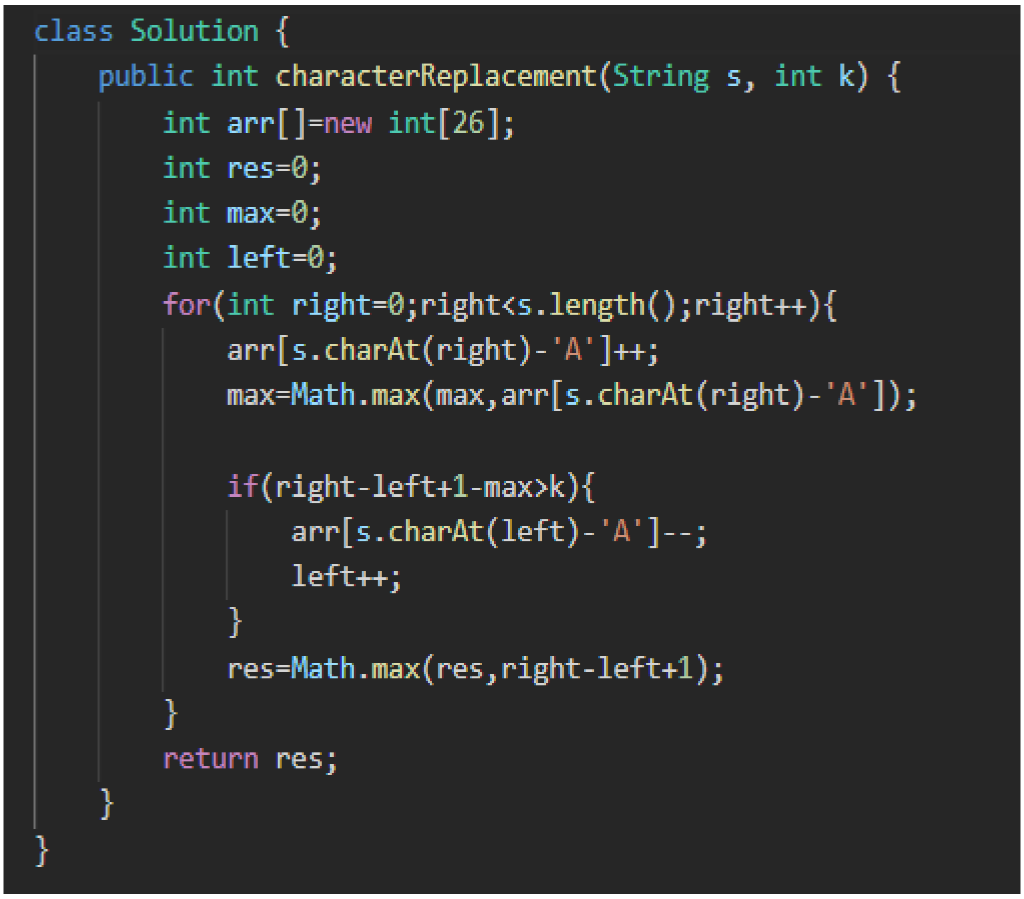
Return *the length of the longest substring containing the same letter you can get after performing the above operations.*

Example 1:

```
Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with two 'B's or vice versa.
```

Example 2:

```
Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBBA".
The substring "BBBB" has the longest repeating letters, which is 4.
There may exists other ways to achieve this answer too.
```



```java
class Solution {
    public int characterReplacement(String s, int k) {
        int arr[]=new int[26];
        int res=0;
        int max=0;
        int left=0;
        for(int right=0;right<s.length();right++){
            arr[s.charAt(right)-'A']++;
            max=Math.max(max,arr[s.charAt(right)-'A']);

            if(right-left+1-max>k){
                arr[s.charAt(left)-'A']--;
                left++;
            }
            res=Math.max(res,right-left+1);
        }
        return res;
    }
}
```