Name : Lakshay Baskotra

UID: 22BCS15016

Ques1:

Longest Nice Substring

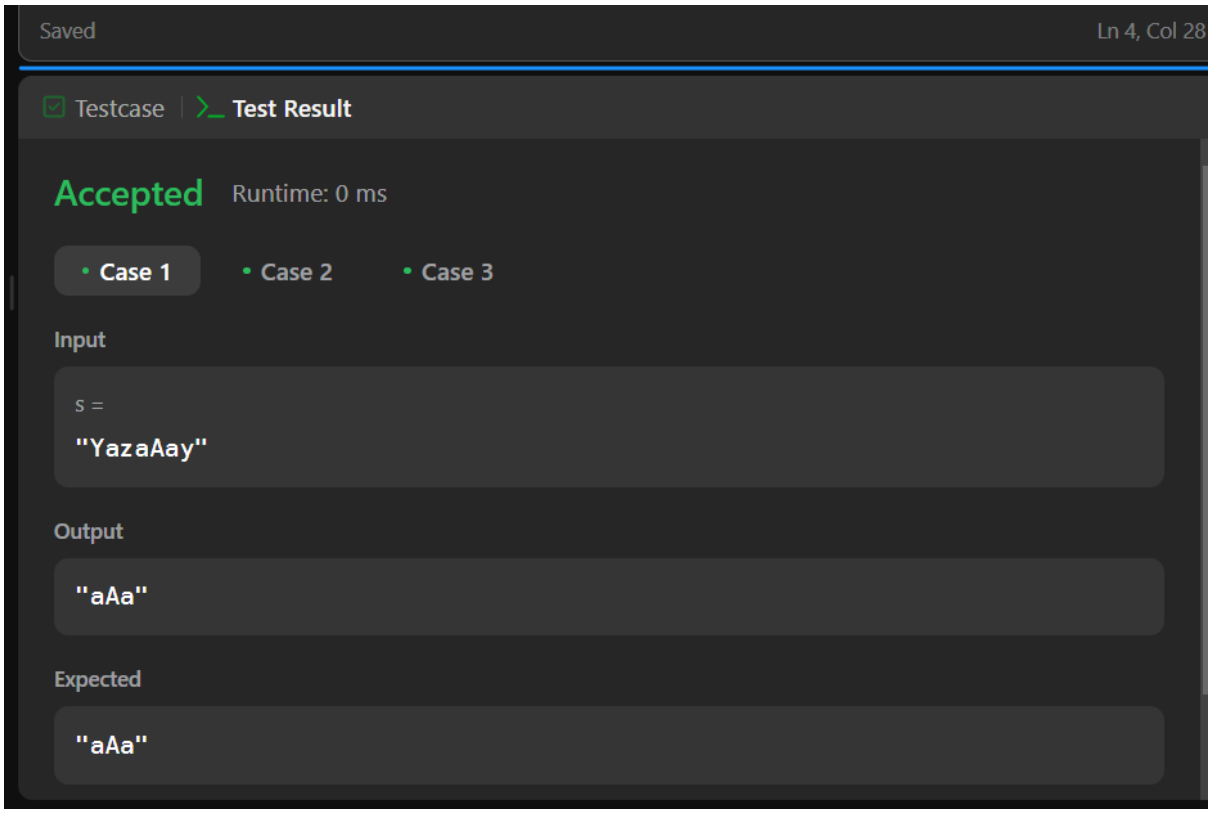
A string s is **nice** if, for every letter of the alphabet that s contains, it appears **both** in uppercase and lowercase. For example, "abABB" is nice because 'A' and 'a' appear, and 'B' and 'b' appear. However, "abA" is not because 'b' appears, but 'B' does not.

Given a string s, return *the longest **substring** of s that is **nice**. If there are multiple, return the substring of the **earliest** occurrence. If there are none, return an empty string*.

SOLUTION :

```
class Solution {
    public String longestNiceSubstring(String s) {
        if (s.length() < 2)
            return "";
        Set<Character> set = new HashSet<>();
        for (char ch : s.toCharArray())
            set.add(ch);
        for (int i=0; i<s.length(); i++) {
            char current = s.charAt(i);
            if (set.contains(Character.toUpperCase(current)) &&
set.contains(Character.toLowerCase(current)))
                continue;
            String str1 = longestNiceSubstring(s.substring(0, i));
            String str2 = longestNiceSubstring(s.substring(i + 1));
            return str1.length() >= str2.length() ? str1 : str2;
        }
        return s;
```

```
    }
}
```



QUES 2 : Reverse bits of a given 32 bits unsigned integer.

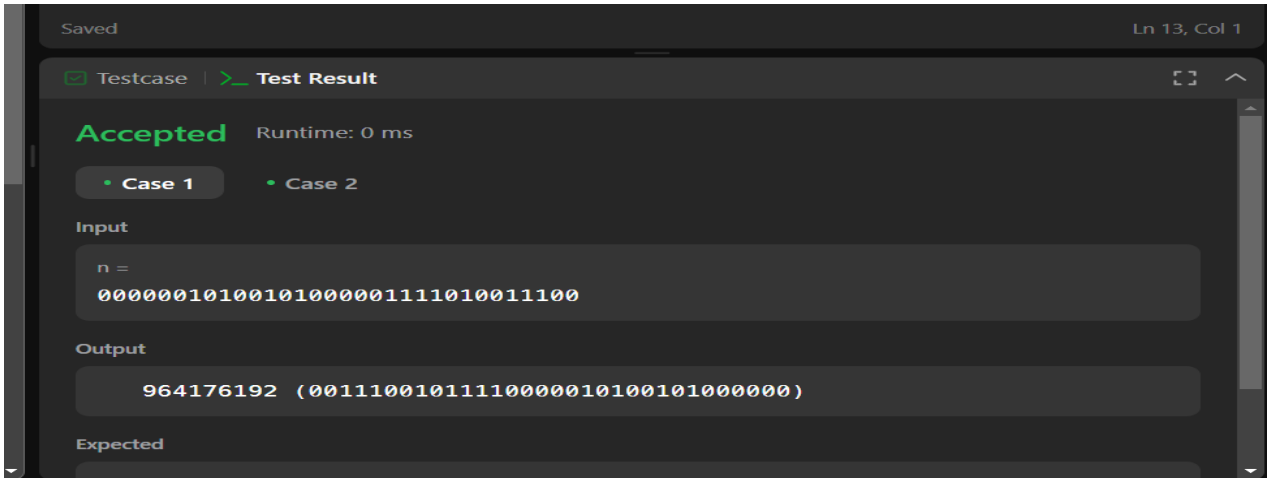
Note:

- Note that in some languages, such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 2** above, the input represents the signed integer -3 and the output represents the signed integer -1073741825.

SOLUTION:

```
public class Solution {
    public int reverseBits(int n) {
        int result = 0;

        for (int i = 0; i < 32; i++) {
            result |= (n & 1) << (31 - i);
```

```
            n >>>= 1;
        }

        return result;
    }
}
```



QUES 3: Given a positive integer n, write a function that returns the number of

set bits

in its binary representation (also known as the Hamming weight).

Example 1:

Input: n = 11

Output: 3

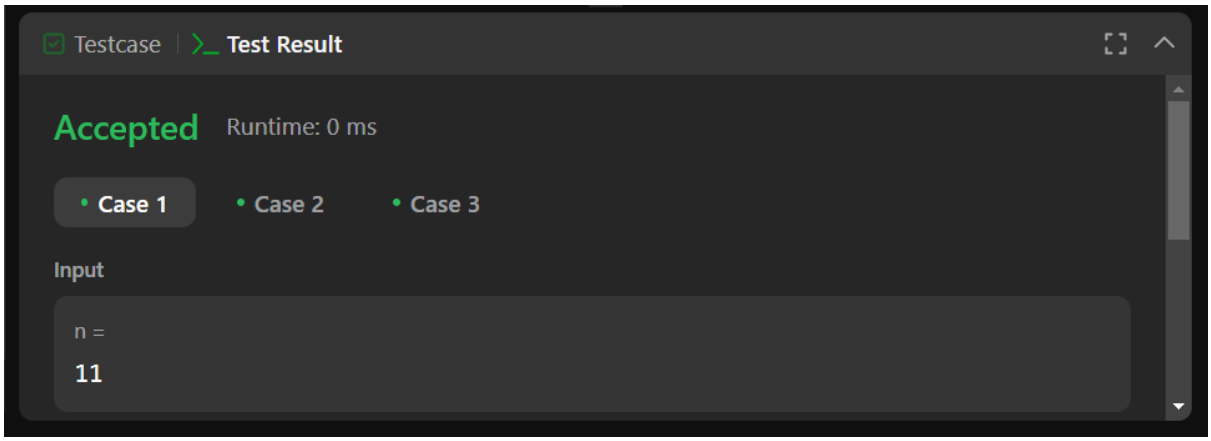
Explanation:

The input binary string **1011** has a total of three set bits.

SOLUTION:

```
public class Solution {
    public int hammingWeight(int n) {
        int count = 0;
```

```
        while (n != 0) {
            count += (n & 1);
            n >>>= 1;
        }
        return count;
    }
}
```



QUES 4: Given an integer array nums, find the

subarray

with the largest sum, and return *its sum*.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

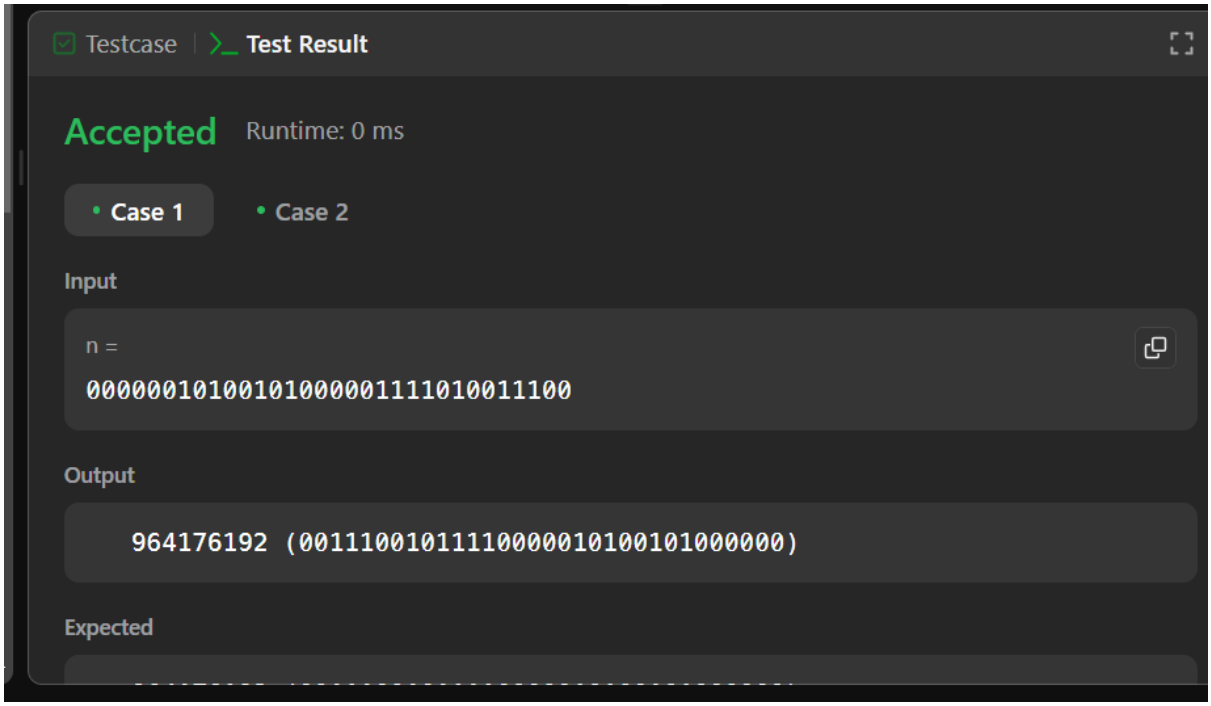
Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Solution:

```
public class Solution {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0];
```

```
        int currentSum = nums[0];

        for (int i = 1; i < nums.length; i++) {
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }

        return maxSum;
    }
}
```



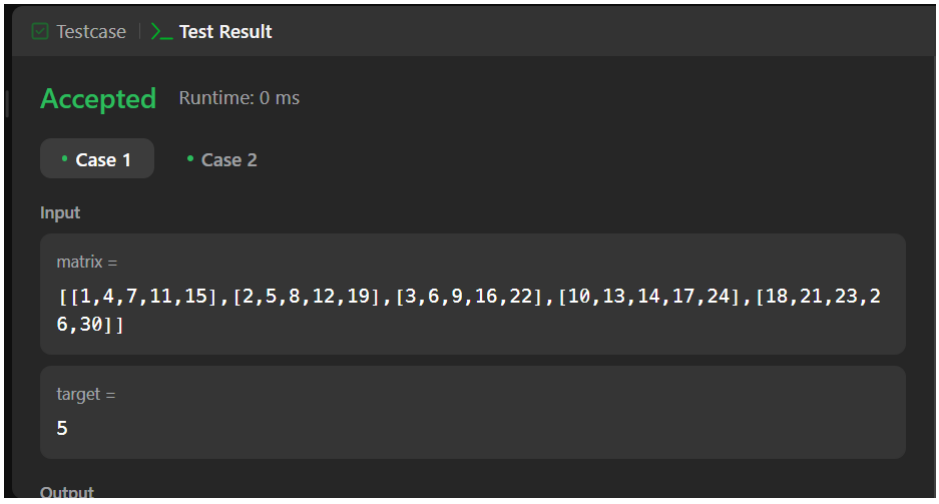
QUES 5:

Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

SOLUTION:

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {

        int n = matrix.length;
        int m = matrix[0].length;
        int row = 0 ;
        int col = m-1;
        while(row<n && col>=0)
        {
            if(matrix[row][col]==target)  return true ;
            else if (matrix[row][col]<target)  row++;
            else {
                col--;
            }
        }

        return false ;

    }
}
```



QUES 6: Your task is to calculate a^b mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.

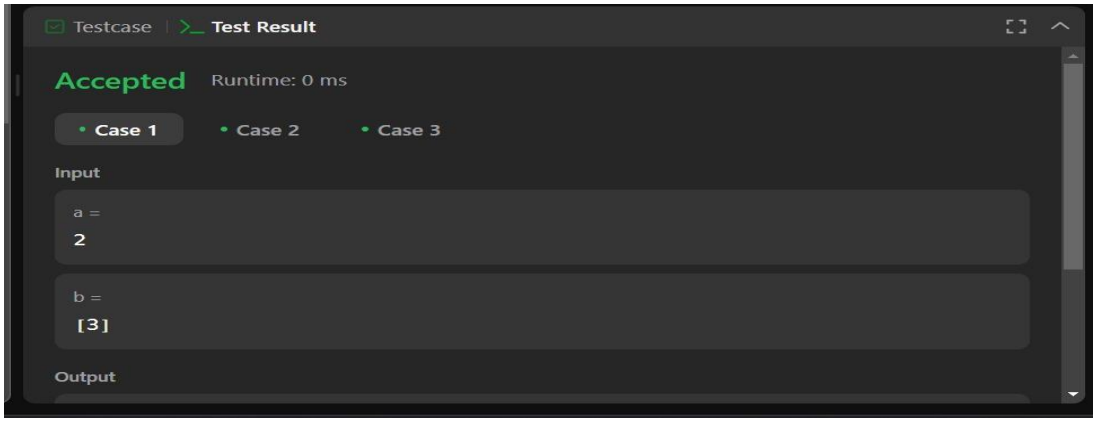
Example 1:

Input: a = 2, b = [3]

Output: 8

SOLUTION:

```
class Solution {


    public int superPow(int a, int[] b) {
        if (a % 1337 == 0) return 0;
        int result = 1;
        for (int digit : b) {
            result = modPow(result, 10) * modPow(a, digit) % 1337;
        }
        return result;
    }


    private int modPow(int base, int exponent) {
        base %= 1337;
        int result = 1;
        for (int i = 0; i < exponent; i++) {
            result = (result * base) % 1337;
        }
        return result;
    }
}
```



QUES 7: An array nums of length n is **beautiful** if:

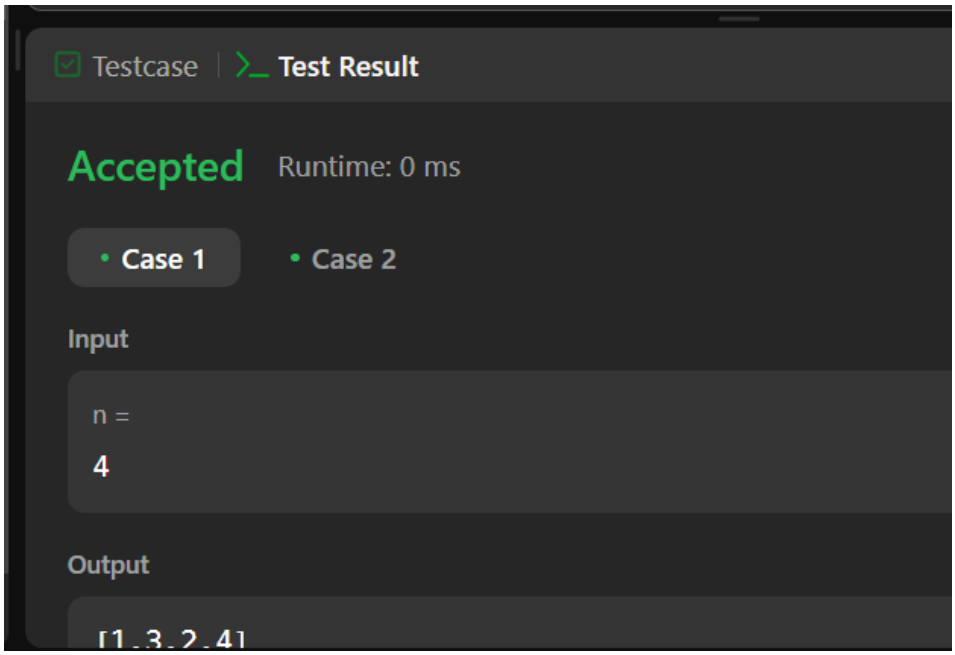
- nums is a permutation of the integers in the range [1, n].
- For every 0 <= i < j < n, there is no index k with i < k < j where 2 * nums[k] == nums[i] + nums[j].

Given the integer n, return *any **beautiful** array* nums *of length* n. There will be at least one valid answer for the given n.

SOLUTION:

```
class Solution {
    public int[] beautifulArray(int N) {
        int[] res = new int[N];
        if (N == 1)
        {
            return new int[] {1};
        }
        else if (N == 2)
        {
            return new int[] {1, 2};
        }
        else
        {
            int[] odds = beautifulArray((N + 1) / 2);
            int[] even = beautifulArray(N / 2);
```

```
        for (int i = 0; i < odds.length; i ++)
        {
            res[i] = odds[i] * 2 - 1;
        }
        for (int j = 0; j < even.length; j ++)
        {
            res[odds.length + j] = even[j] * 2;
        }
    }
    return res;
  }
}
```



QUES 8: A city's **skyline** is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return *the **skyline** formed by these buildings collectively*.

The geometric information of each building is given in the array buildings where buildings[i] = [$left_i$, $right_i$, $height_i$]:

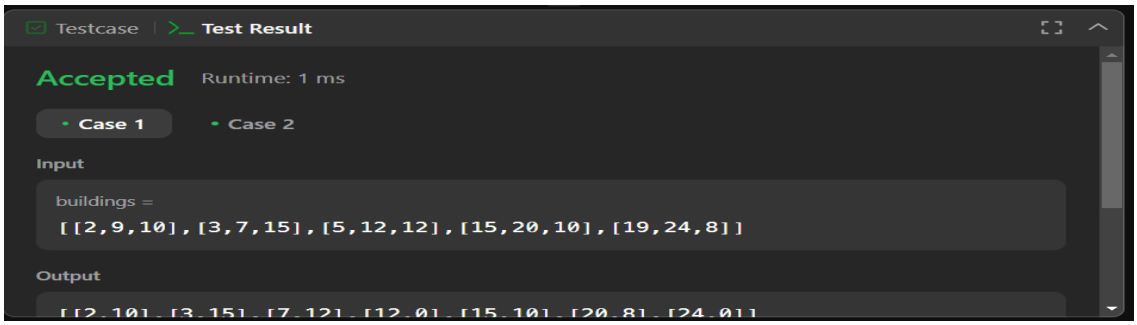
- $left_i$ is the x coordinate of the left edge of the i^{th} building.
- $right_i$ is the x coordinate of the right edge of the i^{th} building.

- $height_i$ is the height of the i^{th} building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

SOLUTION:

```
class Solution {
    public List<List<Integer>> getSkyline(int[][] B) {
        int[][] H = new int[2 * B.length][2];
        for (int i = 0; i < B.length; i++) {
            H[i * 2] = new int[]{B[i][0], -B[i][2]};
            H[i * 2 + 1] = new int[]{B[i][1], B[i][2]};
        }
        Arrays.sort(H, (a, b) -> a[0] != b[0] ? a[0] - b[0] : a[1] - b[1]);
        var map = new TreeMap<Integer, Integer>(Comparator.reverseOrder());
        map.put(0, 1);
        List<List<Integer>> res = new ArrayList<>();
        int prev = 0;
        for (int[] h : H) {
            if (h[1] < 0) map.put(-h[1], map.getOrDefault(-h[1], 0) + 1);
            else {
                map.put(h[1], map.get(h[1]) - 1);
                if (map.get(h[1]) == 0) map.remove(h[1]);
            }
            if (map.firstKey() != prev) {
                prev = map.firstKey();
                res.add(List.of(h[0], prev));
            }
        }
        return res;
    }
}
```



QUES 9: Given an integer array nums, return *the number of **reverse pairs** in the array*.

A **reverse pair** is a pair (i, j) where:

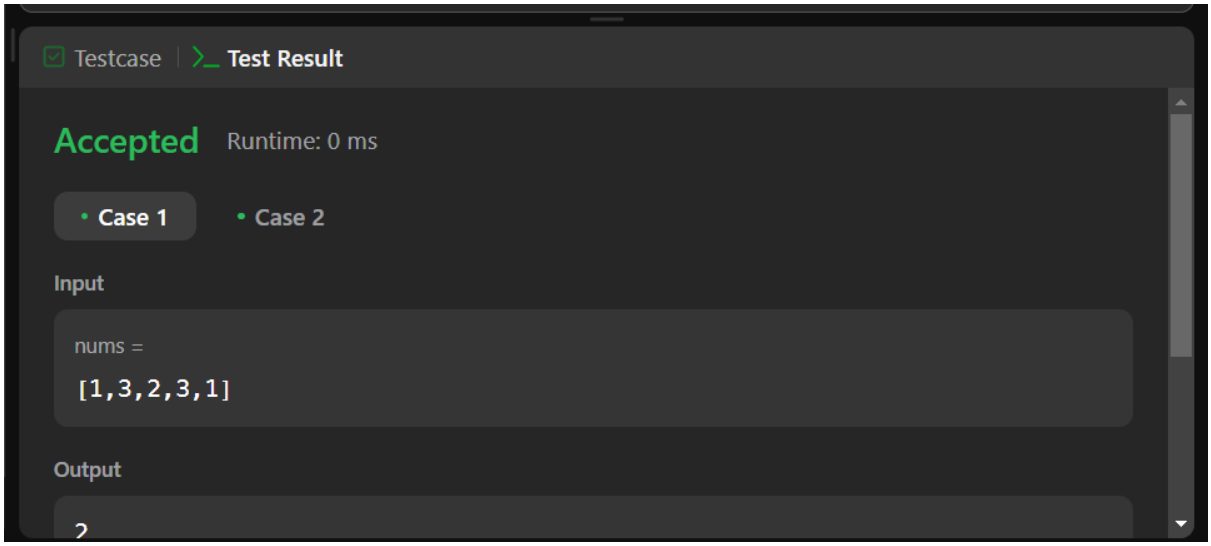
- 0 <= i < j < nums.length and
- nums[i] > 2 * nums[j].

SOLUTION:

```
class Solution {
    public int reversePairs(int[] nums) {
        if(nums == null || nums.length == 0){
            return 0;
        }
        return mergeSort(nums,0,nums.length-1);
    }
    private int mergeSort(int[] nums, int left, int right){
        if(left >= right){
            return 0;
        }
        int mid = left+(right-left)/2;
        int count = mergeSort(nums,left,mid)+mergeSort(nums,mid+1,right);

        count = count+countPairs(nums,left,mid,right);

        merge(nums,left,mid,right);
```

```
    return count;
}
private int countPairs(int[] nums, int left,int mid, int right){
    int count = 0;
    int j= mid+1;
    for(int i=left;i<=mid;i++){

        while(j<=right && nums[i] > 2L * nums[j]){
            j++;
        }
        count += (j-(mid+1));
    }
    return count;
}

private void merge(int[] nums, int left, int mid, int right){
    int[] temp = new int[right-left+1];

    int i = left;
    int j = mid + 1;
    int k = 0;

    while(i<=mid && j<=right){

        if(nums[i]<=nums[j]){
            temp[k++]=nums[i++];
        }else{
            temp[k++]=nums[j++];
        }
    }
```

```
        while(i<=mid){

            temp[k++]=nums[i++];
        }
        while(j<=right){

            temp[k++]=nums[j++];
        }
        System.arraycopy(temp,0,nums,left,temp.length);

    }
}
```



QUES 10: You are given an integer array nums and an integer k.

Find the longest subsequence of nums that meets the following requirements:

- The subsequence is **strictly increasing** and
- The difference between adjacent elements in the subsequence is **at most** k.

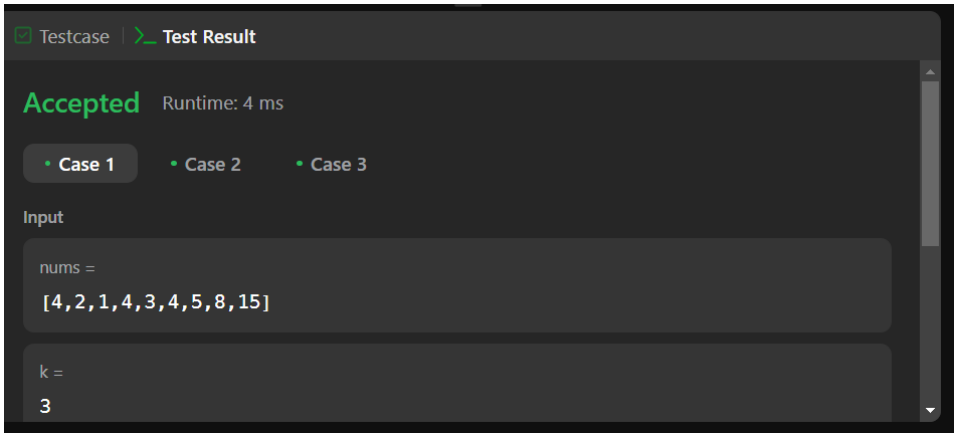
Return *the length of the **longest subsequence** that meets the requirements.*

A **subsequence** is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.

SOLUTION:

```
class Node {
    Node leftChild;
    Node rightChild;
    int start;
    int end;
    int value;

    public Node(int start, int end, int value) {
        this.start = start;
        this.end = end;
        this.value = value;
    }
}

class Solution {
    Node buildSegmentTree(int start, int end) {
        if (start == end)
            return new Node(start, end, 0);
        Node node = new Node(start, end, 0);
        int mid = (start + end) / 2;
        node.leftChild = buildSegmentTree(start, mid);
        node.rightChild = buildSegmentTree(mid + 1, end);
        return node;
    }

    int queryRangeMax(Node node, int l, int r) {
        if (node == null || l > node.end || r < node.start)
            return 0;
        if (l <= node.start && r >= node.end)
            return node.value;
        return Math.max(queryRangeMax(node.leftChild, l, r), queryRangeMax(node.rightChild, l, r));
```

```
    }

    void updateSegmentTree(Node node, int index, int value) {
        if (node == null || index < node.start || index > node.end)
            return;
        node.value = Math.max(value, node.value);
        if (node.start != node.end) {
            updateSegmentTree(node.leftChild, index, value);
            updateSegmentTree(node.rightChild, index, value);
        }
    }

    public int lengthOfLIS(int[] nums, int k) {
        Node root = buildSegmentTree(0, 100001);
        int ans = 1;
        for (int num : nums) {
            int maxValInRange = queryRangeMax(root, Math.max(0, num - k), num - 1) + 1;
            ans = Math.max(ans, maxValInRange);
            updateSegmentTree(root, num, maxValInRange);
        }
        return ans;
    }
}
```



QUES11: You are given two integer arrays nums1 and nums2, sorted in **non-decreasing order**, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

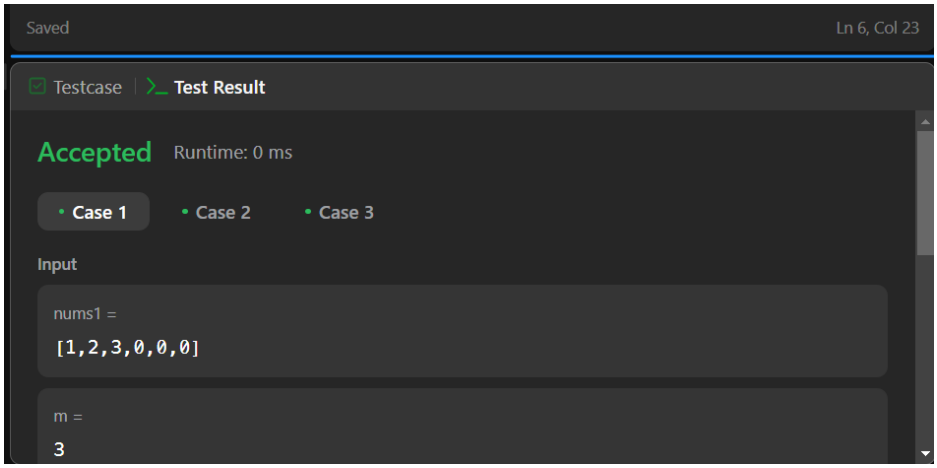
Merge nums1 and nums2 into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

SOLUTION:

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {

        int p1 = m-1;
        int p2 = n-1;
        int i = m+n-1;


        while(p2>=0)
        {
            if(p1>=0 && nums1[p1]>nums2[p2])
            {
                nums1[i--] = nums1[p1--];
            }
            else{
```

```
            nums1[i--] = nums2[p2--];
        }
    }


  }
}
```



QUES 12: You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

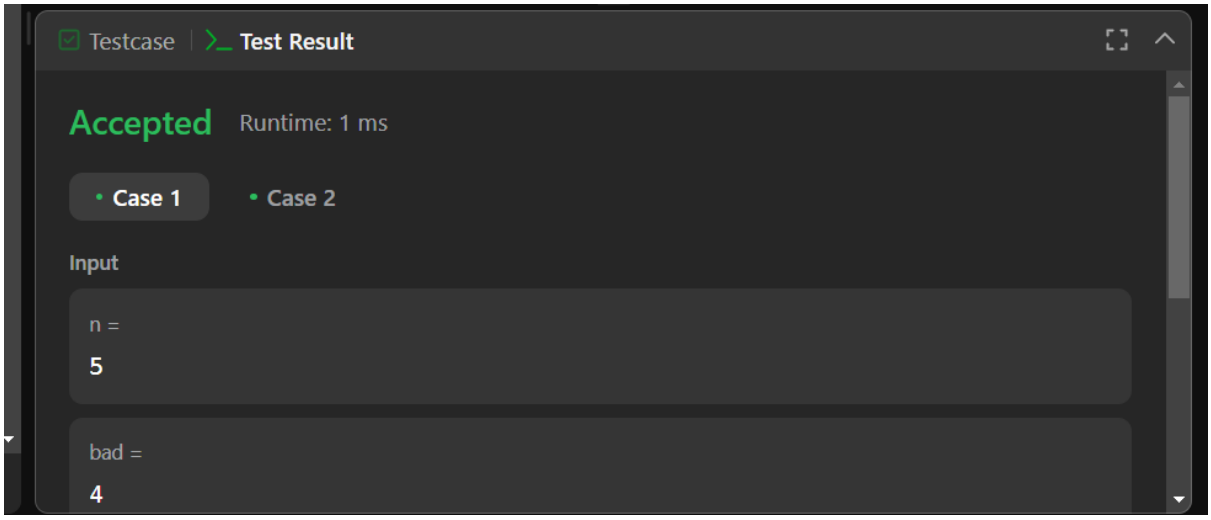
Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

SOLUTION:

```
public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int start = 1;
        int end = n;

        while (start < end) {
            int mid = start + (end - start) / 2;
```

```
            if (isBadVersion(mid)) {
                end = mid;
            } else {
                start = mid + 1;
            }
        }

        return start;
    }
}
```



QUES 13:

Given an array nums with n objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

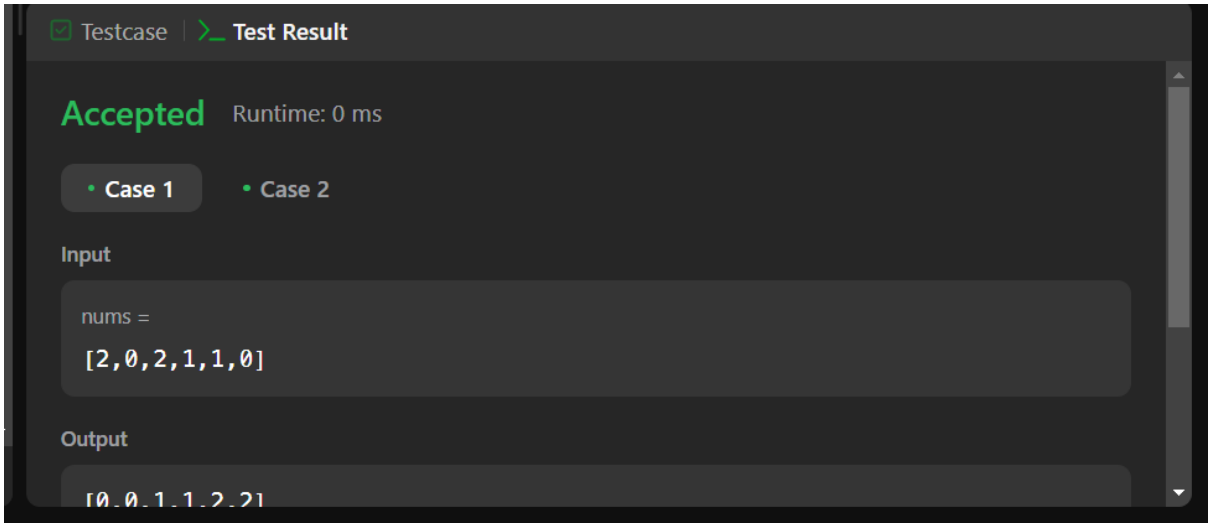
We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

SOLUTION:

```
class Solution {
    public void sortColors(int[] nums) {
    int n = nums.length;
```

```
    int low = 0 ;
    int mid = 0 ;
    int high = n-1;
    while(mid<=high)
    {
        if(nums[mid]==0){
            int temp = nums[low];
            nums[low] = nums[mid];
            nums[mid] = temp;
            low++;
            mid++;
        }
        else if(nums[mid]==1)  mid++;

        else {
            int temp  =nums[high];
            nums[high] = nums[mid];
            nums[mid] =temp;
            high--;
        }
    }

    }
}
```



QUES 14 : A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

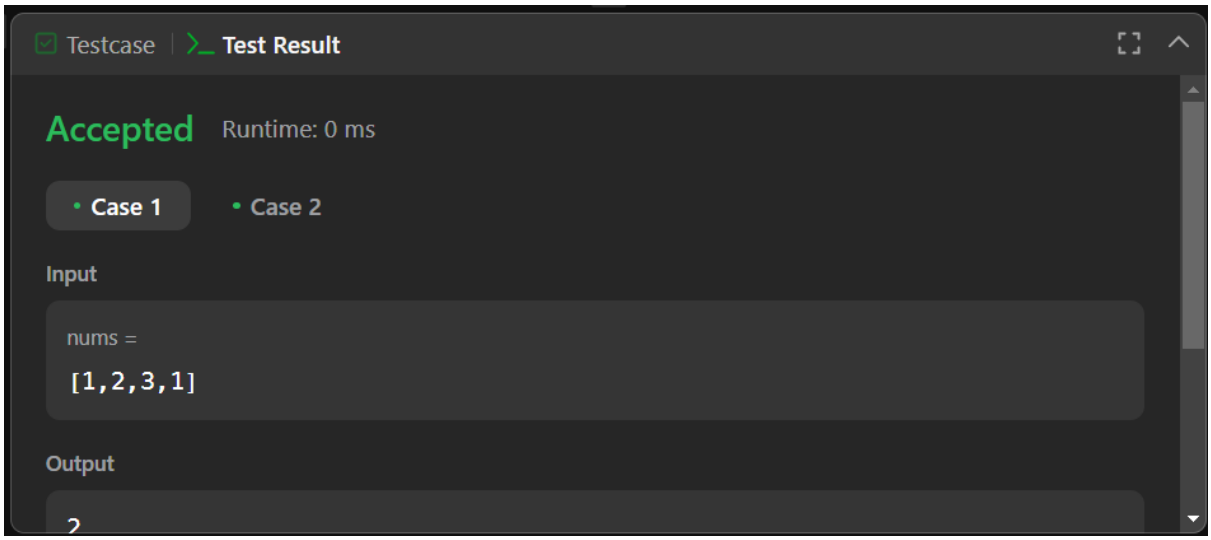
You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

SOLUTION:

```
class Solution {
    public int findPeakElement(int[] nums) {
        int left = 0;
        int right = nums.length - 1;

        while (left < right) {
            int mid = (left + right) / 2;
            if (nums[mid] > nums[mid + 1]) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
```

```
        return left;
    }
}
```



QUES 15: Given an array of intervals where intervals[i] = [$start_i$, end_i], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

Example 1:

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

SOLUTION:

```
import java.util.*;


class Solution {
    public int[][] merge(int[][] intervals) {
        int n = intervals.length;


        Arrays.sort(intervals, new Comparator<int[]>() {
```

```
        public int compare(int[] a, int[] b) {
            return a[0] - b[0];
        }
    });

    List<int[]> ans = new ArrayList<>();

    for (int i = 0; i < n; i++) {
        if (ans.isEmpty() || intervals[i][0] > ans.get(ans.size() - 1)[1]) {
            ans.add(new int[]{intervals[i][0], intervals[i][1]});
        } else {
            ans.get(ans.size() - 1)[1] = Math.max(ans.get(ans.size() - 1)[1], intervals[i][1]);
        }
    }

    // Conveted to int[][]
    return ans.toArray(new int[ans.size()][]);
  }
}
```