ASSIGNMENT -5

NAME : LAKSHAY BASKOTRA   UID:22BCS15016

1. **Find the Difference**
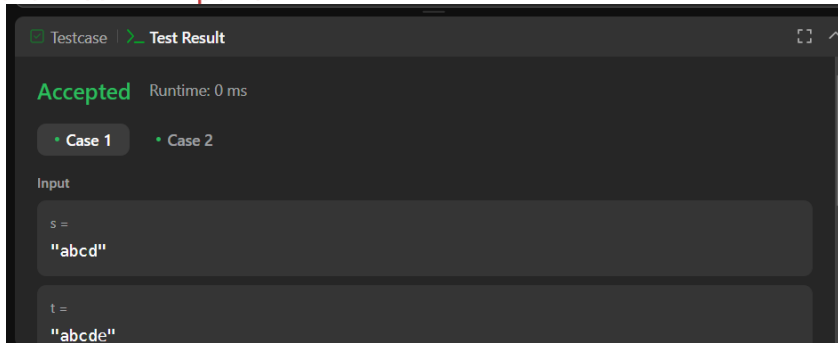
You are given two strings s and t.

String t is generated by random shuffling string s and then add one more letter at a random position.

Return the letter that was added to t.

SOLUTION:

```
class Solution {
    public char findTheDifference(String s, String t) {
        char miss = 0;
        int len = t.length();

        for(int i = 0; i < len; i++) {
            if(i < s.length()) {
                miss ^= s.charAt(i);
            }
            miss ^= t.charAt(i);
        }
        return miss;
    }
}
```
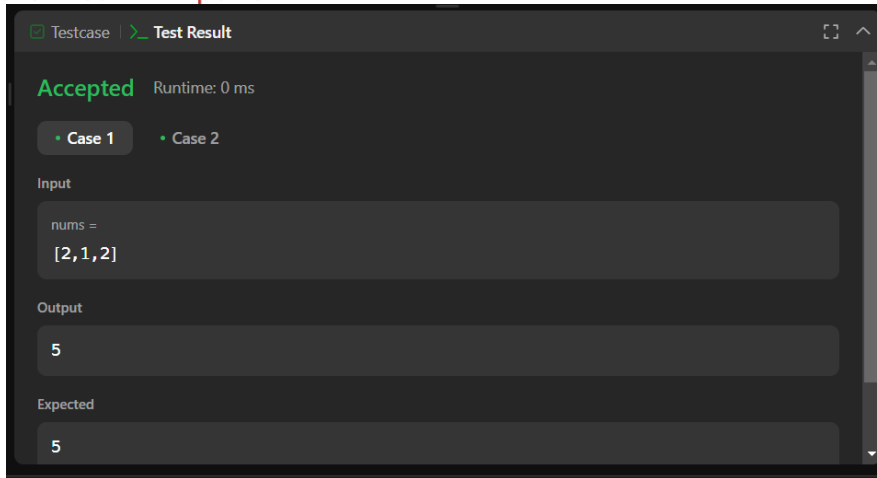
Testcase | Test Result

Accepted Runtime: 0 ms

• Case 1   • Case 2

Input

s =
"abcd"

t =
"abcde"

2. Largest Perimeter Triangle

Given an integer array nums, return the largest perimeter of a triangle with a non-zero area, formed from three of these lengths. If it is impossible to form any triangle of a non-zero area, return 0.

SOLUTION:

```
class Solution {
    public int largestPerimeter(int[] nums) {
        Arrays.sort(nums);
        int n=nums.length;
        for(int i=n-1;i>=2;i--){
            if(nums[i-2]+nums[i-1]>nums[i]){
                return nums[i-2]+nums[i-1]+nums[i];
            }
        }
        return 0;
    }
}
```
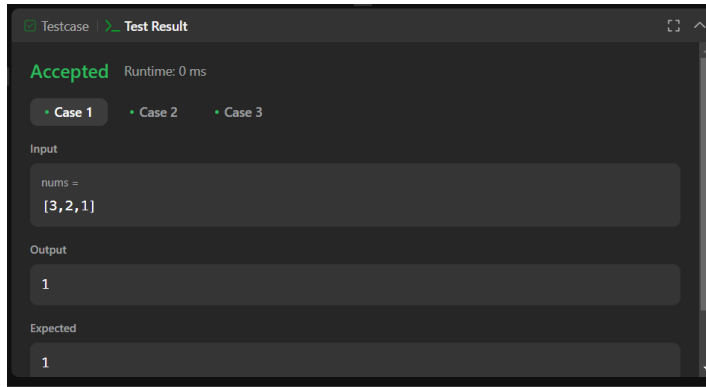
☑ Testcase | >_ **Test Result**

Accepted   Runtime: 0 ms

• **Case 1**   • Case 2

Input

```
nums =
 [2,1,2]
```

Output

```
5
```

Expected

```
5
```

3. Third Maximum Number

Given an integer array nums, return *the **third distinct maximum** number in this array. If the third maximum does not exist, return the **maximum** number*.

SOLUTION:

```java
import java.util.*;

class Solution {
    public int thirdMax(int[] nums) {
        Set<Integer> s = new HashSet<>();
        for (int n : nums) s.add(n);

        if (s.size() < 3) return Collections.max(s);

        s.remove(Collections.max(s));
        s.remove(Collections.max(s));

        return Collections.max(s);
    }
}
```

}

☑ Testcase | >_ **Test Result**

Accepted   Runtime: 0 ms

• **Case 1**   • Case 2   • Case 3

Input

nums =
[3,2,1]

Output

1

Expected

1

4. Given a string s, sort it in **decreasing order** based on the **frequency** of the characters. The **frequency** of a character is the number of times it appears in the string.

Return *the sorted string*. If there are multiple answers, return *any of them*.

SOLUTION:

```
class Solution {
    public String frequencySort(String s) {
        Map<Character , Integer> mp = new HashMap<>();
        StringBuilder st = new StringBuilder();
        for(char c : s.toCharArray())
        {
            mp.put(c , mp.getOrDefault(c,0)+1);


        }


        List<Map.Entry<Character,Integer>> lst  = new
        ArrayList<>(mp.entrySet());
```

```
        lst.sort((entry1 ,entry2)->

    entry2.getValue().compareTo(entry1.getValue()));

       for(Map.Entry<Character,Integer> entry: lst)

        {

st.append(Character.toString(entry.getKey()).repeat(entry.getValue()));

        }


    return st.toString();



    }
}
```
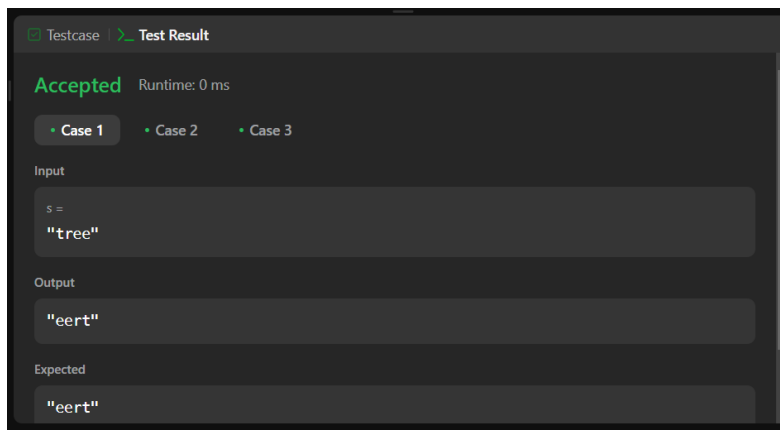

5. [Minimum Number of Arrows to Burst Balloons](#)

There are some spherical balloons taped onto a flat wall that represents the
XY-plane. The balloons are represented as a 2D integer
array points where points[i] = [x_{start}, x_{end}] denotes a balloon
whose **horizontal diameter** stretches between x_{start} and x_{end}. You do not
know the exact y-coordinates of the balloons.

Arrows can be shot up **directly vertically** (in the positive y-direction) from
different points along the x-axis. A balloon with x_{start} and x_{end} is **burst** by an
arrow shot at x if $x_{start} <= x <= x_{end}$. There is **no limit** to the number of

arrows that can be shot. A shot arrow keeps traveling up infinitely, bursting any balloons in its path.

Given the array points, return *the **minimum** number of arrows that must be shot to burst all balloons*.

SOLUTION:

```
class Solution {
    public int findMinArrowShots(int[][] segments) {
        Arrays.sort(segments, (a, b) -> Integer.compare(a[1], b[1]));
        int ans = 0, arrow = 0;
        for (int i = 0; i < segments.length; i ++) {
            if (ans == 0 || segments[i][0] > arrow) {
                ans ++;
                arrow = segments[i][1];
            }
        }
        return ans;
    }
}



        return false ;


    }
}
```

Accepted Runtime: 0 ms

• **Case 1**   • Case 2   • Case 3

Input

```
points =
[[10,16],[2,8],[1,6],[7,12]]
```

Output

```
2
```

Expected

```
2
```

6. Boats to Save People

You are given an array people where people[i] is the weight of the ith person, and an **infinite number of boats** where each boat can carry a maximum weight of limit. Each boat carries at most two people at the same time, provided the sum of the weight of those people is at most limit.

Return *the minimum number of boats to carry every given person*.
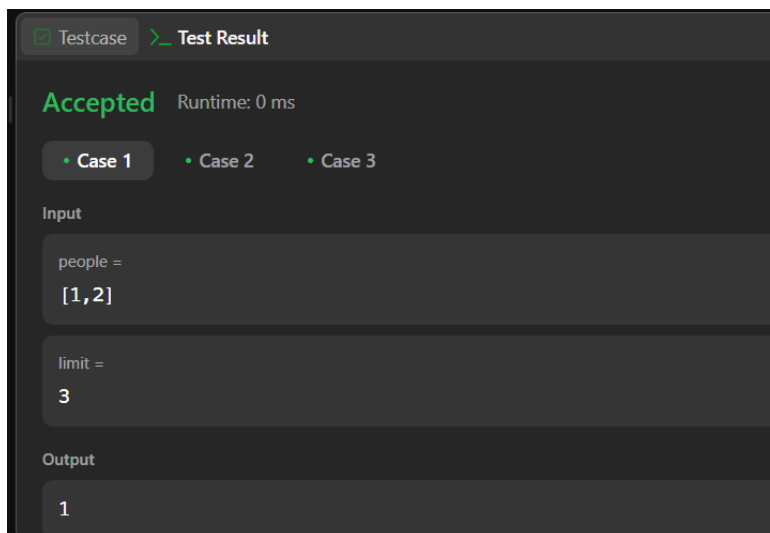
SOLUTION:

```
class Solution {
    public int numRescueBoats(int[] people, int limit) {
        int boatCount = 0;
        Arrays.sort(people);

        int left = 0;
        int right = people.length - 1;

        while(left <= right){
            int sum = people[left] + people[right];
            if(sum <= limit){
                boatCount++;
```

```
            left++;

            right--;

        }

        else{

            boatCount++;

            right--;

        }

    }

    return boatCount;

  }

}
```


7. K Closest Points to Origin

Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).

The distance between two points on the X-Y plane is the Euclidean distance (i.e., $\sqrt{(x1 - x2)2 + (y1 - y2)2}$).

You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in).

SOLUTION:
```java
class Solution {
    static class Point implements Comparable<Point> {
        int x;
        int y;
        int disSq;
        int idx;

        public Point(int x, int y, int disSq, int idx) {
            this.x = x;
            this.y = y;
            this.disSq = disSq;
            this.idx = idx;
        }

        @Override
        public int compareTo(Point p2) {
            return this.disSq - p2.disSq; // Ascending order
        }
    }

    public int[][] kClosest(int[][] points, int k) {
        PriorityQueue<Point> pq = new PriorityQueue<>();

        for (int i = 0; i < points.length; i++) {
```

```java
            int disSq = points[i][0] * points[i][0] + points[i][1] * points[i][1];
            pq.add(new Point(points[i][0], points[i][1], disSq, i));
        }


        int[][] result = new int[k][2];


        for (int i = 0; i < k; i++) {
            Point p = pq.remove();
            result[i][0] = p.x;
            result[i][1] = p.y;
        }


        return result;
    }
}
```

Testcase >_ Test Result

Accepted Runtime: 0 ms

• Case 1   • Case 2

Input

points =
[[1,3],[−2,2]]

k =
1

Output

[[−2,2]]

8. Reduce Array Size to The Half

You are given an integer array arr. You can choose a set of integers and remove all the occurrences of these integers in the array.

Return the minimum size of the set so that at least half of the integers of the array are removed.

SOLUTION:

```java
class Solution {
    public int minSetSize(int[] arr) {
        HashMap<Integer, Integer> map = new HashMap<>();
        PriorityQueue<Integer> pq = new
    PriorityQueue<>(Collections.reverseOrder());

        for(int n:arr){
            map.put(n, map.getOrDefault(n,0)+1);
        }

        if(map.size()==1) return 1;
        for(int n:map.values()){
            pq.add(n);
        }
        int size=0;
        int count=0;
        while(!pq.isEmpty()){
            int a = pq.poll();
            size+=a;
            if(size >= arr.length/2) {
```

```
                count++;

            return count;

        }

        else count++;


    }

    return count;

}

}
```