

## PROBLEM SHEET 2 Solution 1:

This screenshot shows the LeetCode platform interface for a Java solution to problem 173. The code implements a modified quicksort algorithm to find the k-th smallest element in a matrix. It uses partitioning and recursion to efficiently search through the sorted matrix rows.

```

1 class Solution {
2     public int kthSmallest(int[] arr, int k) {
3         int l = 0, r = arr.length - 1;
4
5         while (l <= r) {
6             int p = partition(arr, l, r);
7
8             if (p == k - 1) return arr[p];
9             else if (p > k - 1) r = p - 1;
10            else l = p + 1;
11        }
12        return -1;
13    }
14
15    private int partition(int[] a, int l, int r) {
16        int pivot = a[r], i = l;
17
18        for (int j = l; j < r; j++) {
19            if (a[j] <= pivot) {
20                int t = a[i]; a[i] = a[j]; a[j] = t;
21                i++;
22            }
23        }
24        int t = a[i]; a[i] = a[r]; a[r] = t;
25        return i;
26    }
27
28}

```

## Solution 2:

This screenshot shows the LeetCode platform interface for a Java solution to problem 173. The code uses a different approach, likely a binary search on the columns to find the k-th smallest element. It sorts the array and then iterates through it to calculate the minimum difference between the current element and the target k-th element.

```

1 import java.util.Arrays;
2
3 class Solution {
4     int getMinDiff(int[] arr, int k) {
5
6         int n = arr.length;
7         Arrays.sort(arr);
8
9         int ans = arr[n - 1] - arr[0];
10
11         int smallest = arr[0] + k;
12         int largest = arr[n - 1] - k;
13
14         for (int i = 0; i < n - 1; i++) {
15
16             int min = Math.min(smallest, arr[i + 1] - k);
17             int max = Math.max(largest, arr[i] + k);
18
19             if (min < 0) continue;
20
21             ans = Math.min(ans, max - min);
22
23         }
24
25         return ans;
26     }
27
28}

```

## Solution 3:

This screenshot shows the LeetCode platform interface for a Java solution to problem 173. The code uses a dynamic programming approach to calculate the minimum number of jumps required to reach the k-th smallest element. It maintains a DP array where each index represents the minimum jumps needed to reach that index.

```

1 class Solution {
2     static int minJumps(int[] arr) {
3
4         int n = arr.length;
5
6         if (n == 1) return 0;
7
8         if (arr[0] == 0) return -1;
9
10        int maxReach = arr[0];
11        int steps = arr[0];
12        int jumps = 1;
13
14        for (int i = 1; i < n; i++) {
15
16            if (i == n - 1) return jumps;
17
18            maxReach = Math.max(maxReach, i + arr[i]);
19
20            steps--;
21
22            if (steps == 0) {
23                jumps++;
24
25                if (i >= maxReach) return -1;
26
27                steps = maxReach - i;
28            }
29        }
30
31        return jumps;
32    }
33
34}

```

## Solution 4:

This screenshot shows the LeetCode problem details for Solution 4. The status is Accepted with 59 / 59 testcases passed. The code is written in Java and implements the Floyd's Tortoise and Hare (Cycle Detection) algorithm to find a duplicate number in an array. The runtime is 4 ms (Beats 90.89%) and memory usage is 83.24 MB (Beats 25.70%). A histogram indicates that 13.82% of solutions used 5 ms of runtime.

```
1 class Solution {
2     public int findDuplicate(int[] nums) {
3         int slow = nums[0];
4         int fast = nums[0];
5
6         do {
7             slow = nums[slow];
8             fast = nums[nums[fast]];
9         } while (slow != fast);
10
11         slow = nums[0];
12         while (slow != fast) {
13             slow = nums[slow];
14             fast = nums[fast];
15         }
16
17         return slow;
18     }
19 }
```

## Solution 5:

This screenshot shows the LeetCode problem details for Solution 5. The status is Problem Solved Successfully. The code is written in Java and implements the merge step of the merge sort algorithm to merge two sorted arrays (a and b) into one. It uses a gap-based approach to merge elements from both arrays. The test cases passed are 1111 / 1111, accuracy is 100%, and time taken is 0.57 seconds.

```
1 class Solution {
2     public void mergeArrays(int a[], int b[]) {
3         int n = a.length;
4         int m = b.length;
5
6         int gap = nextGap(n + m);
7
8         while (gap > 0) {
9             int i = 0;
10            int j = gap;
11
12            while (j < n + m) {
13
14                if (i < n && j < n) {
15                    if (a[i] > a[j]) {
16                        int temp = a[i];
17                        a[i] = a[j];
18                        a[j] = temp;
19                    }
20                }
21
22                else if (i < n && j >= n) {
23                    if (a[i] > b[j - n]) {
24                        int temp = a[i];
25                        a[i] = b[j - n];
26                        b[j - n] = temp;
27                    }
28                }
29            }
30            gap = nextGap(gap / 2);
31        }
32    }
33 }
```

## Solution 6:

This screenshot shows the LeetCode problem details for Solution 6. The status is Accepted with 172 / 172 testcases passed. The code is written in Java and implements the merge step of the merge sort algorithm to merge overlapping intervals. It sorts the intervals by their start value and then iterates through them to merge overlapping ones. The runtime is 7 ms (Beats 98.63%) and memory usage is 49.31 MB (Beats 21.33%).

```
1 import java.util.*;
2
3 class Solution {
4     public int[][] merge(int[][] intervals) {
5
6         Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
7
8         List<int[]> result = new ArrayList<>();
9
10        int[] current = intervals[0];
11        result.add(current);
12
13        for (int i = 1; i < intervals.length; i++) {
14
15            if (current[1] < intervals[i][0]) {
16                current = intervals[i];
17                result.add(current);
18            }
19            else if (current[1] >= intervals[i][0] && current[1] <= intervals[i][1]) {
20                current[1] = intervals[i][1];
21            }
22            else if (current[1] >= intervals[i][0] && current[1] > intervals[i][1]) {
23                current[1] = intervals[i][1];
24            }
25        }
26
27        return result.toArray(new int[result.size()][]);
28    }
29 }
```

## Solution 7:

The screenshot shows a Java code editor interface. The code in the editor is:

```
1 import java.util.*;
2
3 public class Solution {
4     public List<Integer> commonElements(List<Integer> arr1, List<Integer> arr2, List<Integer> arr3) {
5         List<Integer> result = new ArrayList<>();
6         int i = 0, j = 0, k = 0;
7         int n1 = arr1.size(), n2 = arr2.size(), n3 = arr3.size();
8
9         while (i < n1 && j < n2 && k < n3) {
10             if (arr1.get(i).equals(arr2.get(j)) && arr2.get(j).equals(arr3.get(k))) {
11                 if (result.isEmpty() || !result.get(result.size() - 1).equals(arr1.get(i))) {
12                     result.add(arr1.get(i));
13                 }
14             }
15             i++;
16             j++;
17             k++;
18         }
19         if (arr1.get(i) < arr2.get(j)) {
20             i++;
21         } else if (arr2.get(j) < arr3.get(k)) {
22             j++;
23         } else {
24             k++;
25         }
26     }
27     return result;
28 }
```

The output window shows "Problem Solved Successfully" with a green checkmark. Statistics: Test Cases Passed 1215 / 1215, Attempts: Correct / Total 1 / 1, Accuracy: 100%, Points Scored 2 / 2, Time Taken 3.82, and Your Total Score: 29.

## Solution 8:

The screenshot shows a Java code editor interface. The code in the editor is:

```
1 import java.util.*;
2
3 public class Solution {
4     public ArrayList<Integer> factorial(int n) {
5         ArrayList<Integer> result = new ArrayList<>();
6         result.add(1);
7
8         for (int i = 2; i <= n; i++) {
9             int carry = 0;
10            for (int j = result.size() - 1; j >= 0; j--) {
11                int prod = result.get(j) * i + carry;
12                result.set(j, prod % 10);
13                carry = prod / 10;
14            }
15            while (carry > 0) {
16                result.add(0, carry % 10);
17                carry /= 10;
18            }
19        }
20        return result;
21    }
22 }
```

The output window shows "Problem Solved Successfully" with a green checkmark. Statistics: Test Cases Passed 1111 / 1111, Attempts: Correct / Total 1 / 1, Accuracy: 100%, Points Scored 4 / 4, Time Taken 0.62, and Your Total Score: 33.

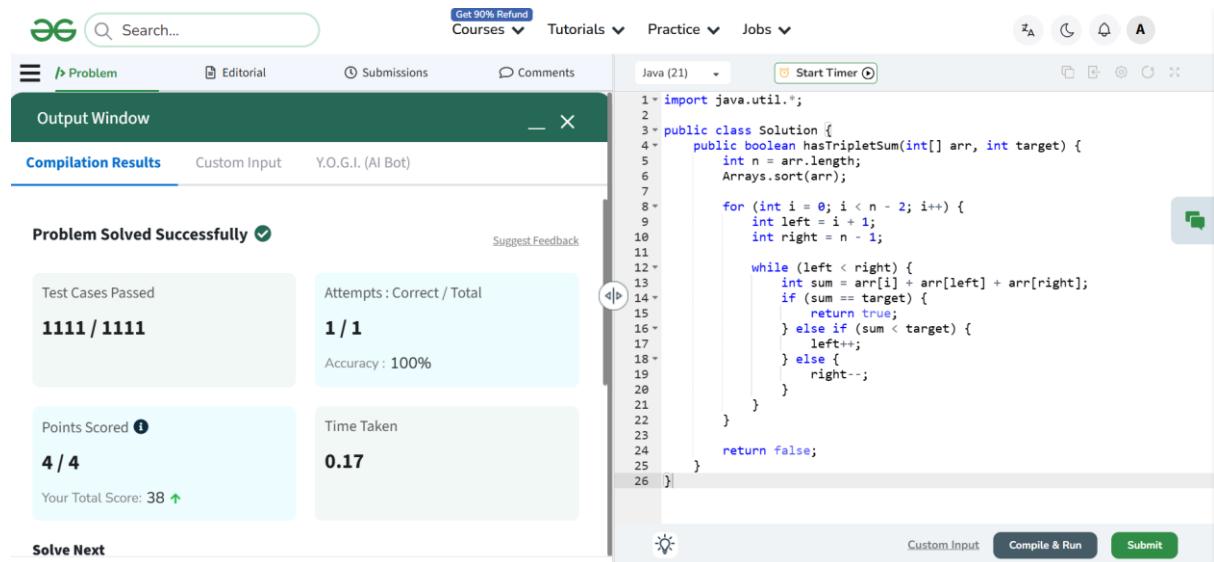
## Solution 9:

The screenshot shows a Java code editor interface. The code in the editor is:

```
1 import java.util.*;
2
3 public class Solution {
4     public boolean isSubset(int[] a, int[] b) {
5         Map<Integer, Integer> countA = new HashMap<>();
6
7         for (int num : a) {
8             countA.put(num, countA.getOrDefault(num, 0) + 1);
9         }
10
11        for (int num : b) {
12            if (!countA.containsKey(num) || countA.get(num) == 0) {
13                return false;
14            }
15            countA.put(num, countA.get(num) - 1);
16        }
17
18        return true;
19    }
20 }
```

The output window shows "Problem Solved Successfully" with a green checkmark. Statistics: Test Cases Passed 1114 / 1114, Attempts: Correct / Total 1 / 1, Accuracy: 100%, Points Scored 1 / 1, Time Taken 0.57, and Your Total Score: 34.

## Solution 10:



A screenshot of a LeetCode problem-solving interface. The top navigation bar includes links for 'Get 90% Refund', 'Courses', 'Tutorials', 'Practice', and 'Jobs'. The main area shows a Java code editor with the following code:

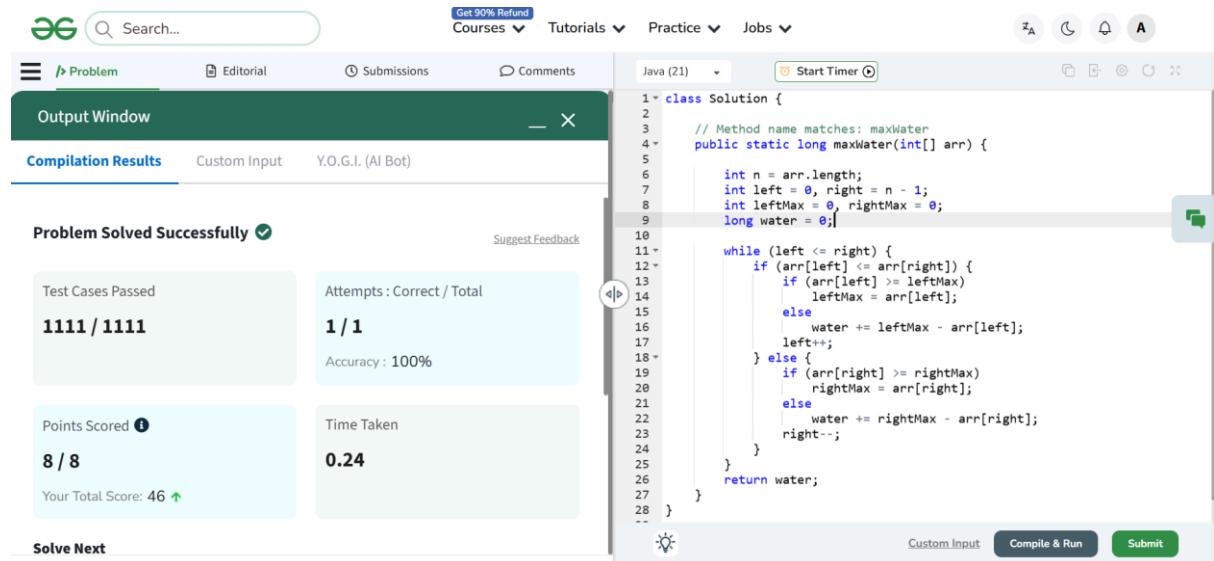
```
1 import java.util.*;
2
3 public class Solution {
4     public boolean hasTripletSum(int[] arr, int target) {
5         int n = arr.length;
6         Arrays.sort(arr);
7
8         for (int i = 0; i < n - 2; i++) {
9             int left = i + 1;
10            int right = n - 1;
11
12            while (left < right) {
13                int sum = arr[i] + arr[left] + arr[right];
14                if (sum == target) {
15                    return true;
16                } else if (sum < target) {
17                    left++;
18                } else {
19                    right--;
20                }
21            }
22        }
23
24        return false;
25    }
26}
```

The 'Output Window' section displays the following results:

- Test Cases Passed: **1111 / 1111**
- Attempts: Correct / Total: **1 / 1** Accuracy: 100%
- Points Scored: **4 / 4**
- Time Taken: **0.17**
- Your Total Score: **38**

Buttons at the bottom include 'Solve Next', 'Custom Input', 'Compile & Run', and 'Submit'.

## Solution 11:



A screenshot of a LeetCode problem-solving interface. The top navigation bar includes links for 'Get 90% Refund', 'Courses', 'Tutorials', 'Practice', and 'Jobs'. The main area shows a Java code editor with the following code:

```
1 class Solution {
2     // Method name matches: maxWater
3     public static long maxWater(int[] arr) {
4
5         int n = arr.length;
6         int left = 0, right = n - 1;
7         int leftMax = 0, rightMax = 0;
8         long water = 0;
9
10
11        while (left <= right) {
12            if (arr[left] < arr[right]) {
13                if (arr[left] >= leftMax)
14                    leftMax = arr[left];
15                else
16                    water += leftMax - arr[left];
17                left++;
18            } else {
19                if (arr[right] >= rightMax)
20                    rightMax = arr[right];
21                else
22                    water += rightMax - arr[right];
23                right--;
24            }
25        }
26
27        return water;
28    }
29}
```

The 'Output Window' section displays the following results:

- Test Cases Passed: **1111 / 1111**
- Attempts: Correct / Total: **1 / 1** Accuracy: 100%
- Points Scored: **8 / 8**
- Time Taken: **0.24**
- Your Total Score: **46**

Buttons at the bottom include 'Solve Next', 'Custom Input', 'Compile & Run', and 'Submit'.