Experiment 4

Student Name: Mohsin Jamil **UID: 22BET10188**
Branch: BE-IT **Section/Group: 22BET_IOT-703/A**
Semester: 6th **Date of Performance: 13/02/25**
Subject Name: Advanced programming Lab II **Subject Code: 22ITP-351**

PROBLEM 1:

1. **Aim:** Longest Nice Substring **(Easy)**

2. **Objective:** 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.

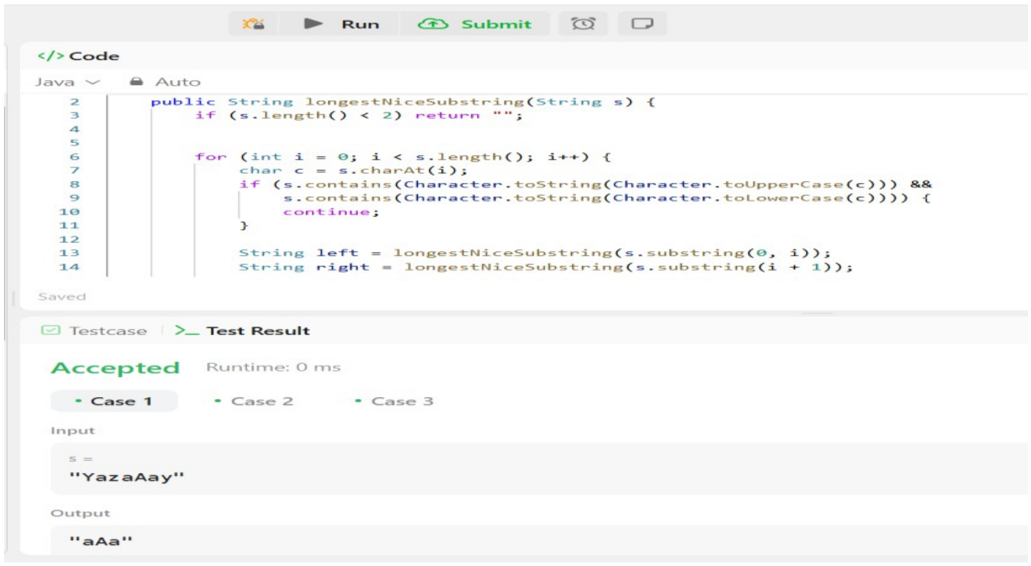
3. **Code:**

```
class Solution {
   public String longestNiceSubstring(String s)
      { if (s.length() < 2) return "";
      for (int i = 0; i < s.length(); i++)
         { char c = s.charAt(i);
         if (s.contains(Character.toString(Character.toUpperCase(c))) && s.contains (Character.toString
(Character.toLowerCase(c)))) {
            continue;
         }
         String left = longestNiceSubstring(s.substring(0, i));
         String right = longestNiceSubstring(s.substring(i + 1));
         return left.length() >= right.length() ? left : right;
      }
      return s;
   }
}
```

4. **Time Complexity:**
   Best Case (String is already "nice") = O(n)
   Average Case (Some splits, but balanced) = O (n log n)
   Worst Case (Unbalanced splits at every character) = O(n^2)

   Space complexity is O(n)

5. **Output:**



```java
    public String longestNiceSubstring(String s) {
        if (s.length() < 2) return "";


        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (s.contains(Character.toString(Character.toUpperCase(c))) &&
                s.contains(Character.toString(Character.toLowerCase(c)))) {
                continue;
            }

            String left = longestNiceSubstring(s.substring(0, i));
            String right = longestNiceSubstring(s.substring(i + 1));
```

```
Accepted   Runtime: 0 ms
• Case 1   • Case 2   • Case 3
Input
s =
"YazaAay"
Output
"aAa"
```

<u>PROBLEM 2</u>:

1. **Aim:** Maximum Subarrray **(Medium)**.

2. **Objective:** Given an integer array nums, find the subarray with the largest sum, and return its sum.

3. **Code:**

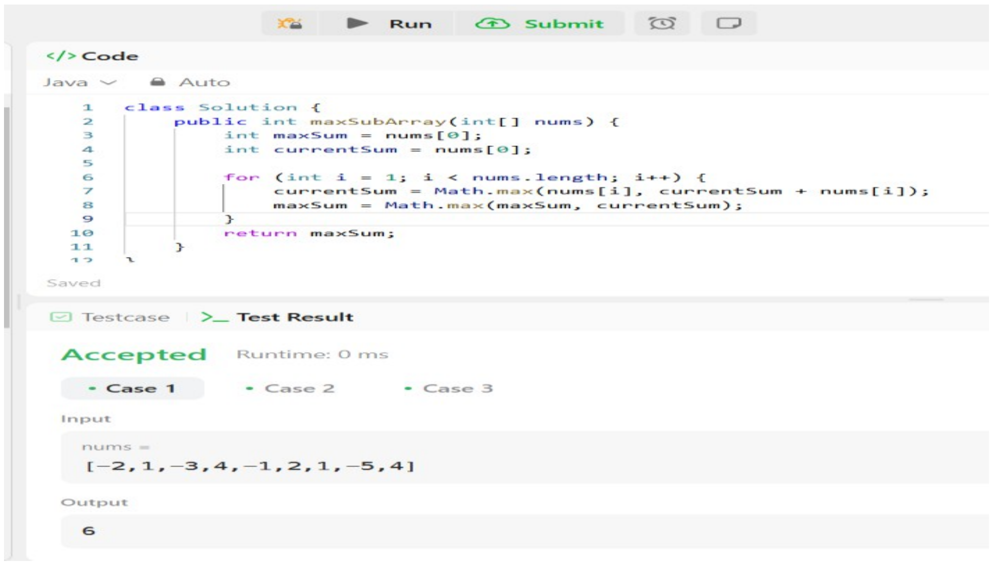
```java
class Solution {
    public int maxSubArray(int[] nums)
        { int maxSum = nums[0];
        int currentSum = nums[0];

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

4. **Time Complexity:**

   Time Complexity: O(n) (linear time)
   Space Complexity: O(1) (constant space)

5. **Output:**



```java
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;
    }
}
```

```
Accepted    Runtime: 0 ms
• Case 1    • Case 2    • Case 3
Input
nums =
[-2,1,-3,4,-1,2,1,-5,4]
Output
6
```

<u>PROBLEM 3</u>:

1. **Aim:** Reverse Pairs **(Hard)**.

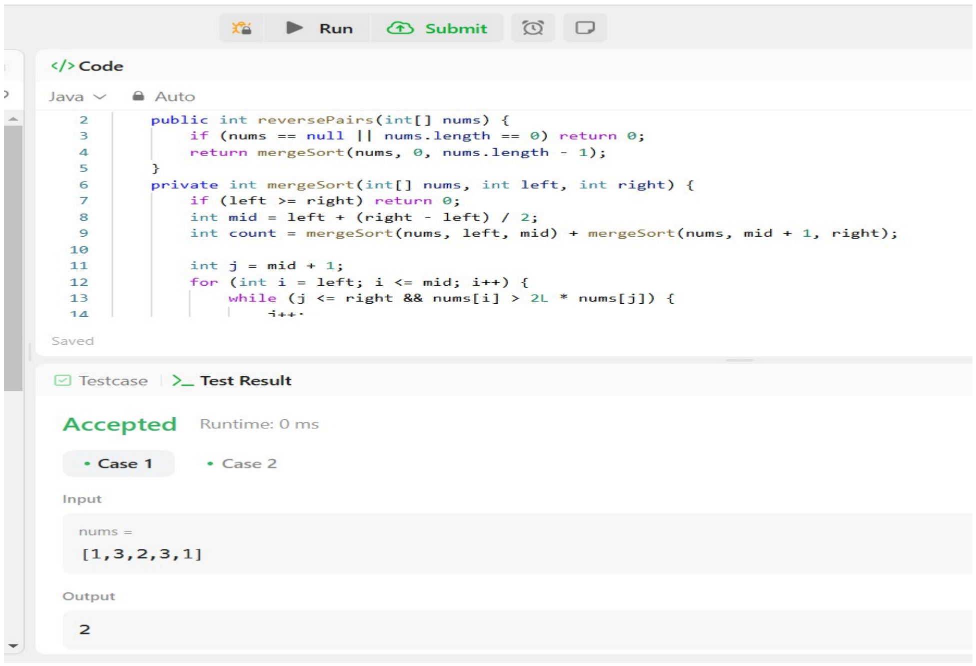
2. **Objective:** Given an integer array nums, return the number of reverse pairs in the array. A reverse pair is a pair (i, j) where:
   - 0 <= i < j < nums.length and
   - nums[i] > 2 * nums[j].

3. **Code:**

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

```
        merge(nums, left, mid, right);
        return count;
    }
    private void merge(int[] nums, int left, int mid, int right)
        { int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, 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);
    }
}
```

4. Output:



```
    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);

        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
            while (j <= right && nums[i] > 2L * nums[j]) {
                j++;
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
nums =
[1,3,2,3,1]
```

Output

```
2
```

5. Time Complexity:

Time Complexity = O(n log n)

Space Complexity = O(n)

PROBLEM 4:

1. **Aim:** Longest Increasing Subsequence II **(Hard)**.

2. **Objective:** 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.

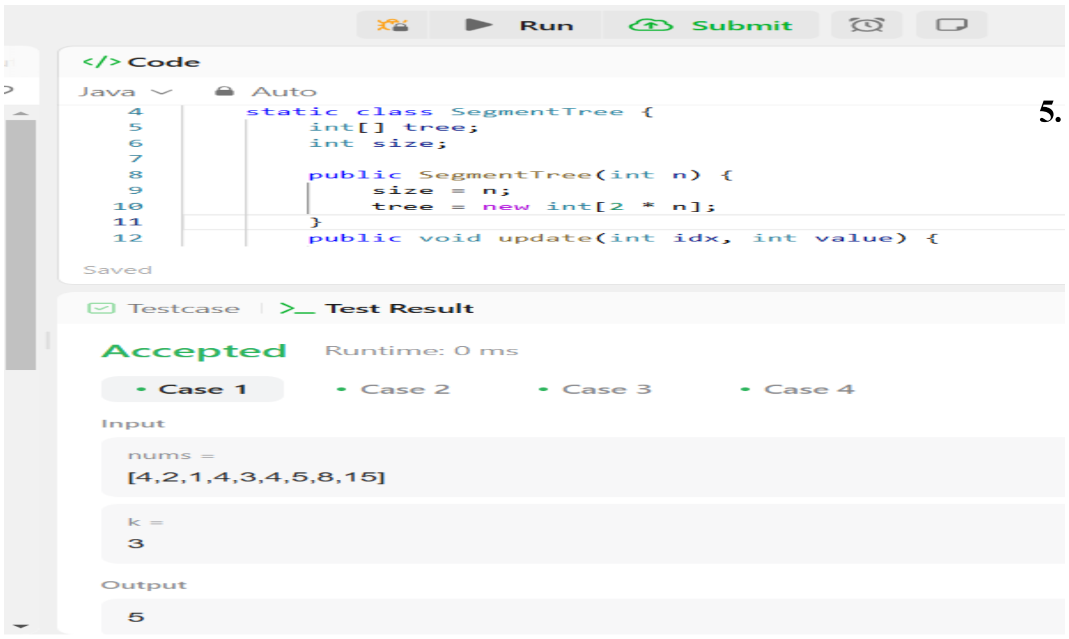
   Return the length of the longest subsequence that meets the requirements.

3. **Code:**

```
import java.util.*;
class Solution {
    static class SegmentTree
        { int[] tree;
        int size;
        public SegmentTree(int n)
            { size = n;
            tree = new int[2 * n];
        }
        public void update(int idx, int value)
            { idx += size;
            tree[idx] = value;
            while (idx > 1) {
                idx /= 2;
                tree[idx] = Math.max(tree[2 * idx], tree[2 * idx + 1]);
            }
        }
        public int query(int left, int right)
            { int res = 0;
            left += size;
            right += size;
            while (left <= right) {
                if ((left & 1) == 1) res = Math.max(res, tree[left++]);
                if ((right & 1) == 0) res = Math.max(res, tree[right--]);
                left /= 2;
                right /= 2;
            }
            return res;
        }
    }
```

```
        public int lengthOfLIS(int[] nums, int k) {
            int maxVal = Arrays.stream(nums).max().getAsInt();
            SegmentTree segTree = new SegmentTree(maxVal + 1);
            int maxLen = 1;
            for (int num : nums) {
                int bestPrev = segTree.query(Math.max(1, num - k), num - 1);
                int newLength = bestPrev + 1;
                segTree.update(num, newLength);
                maxLen = Math.max(maxLen, newLength);
            }
            return maxLen;
        }
    }
```

4. Output:



```
    static class SegmentTree {
        int[] tree;
        int size;

        public SegmentTree(int n) {
            size = n;
            tree = new int[2 * n];
        }
        public void update(int idx, int value) {
```

Accepted   Runtime: 0 ms

• Case 1   • Case 2   • Case 3   • Case 4

Input

nums =
[4,2,1,4,3,4,5,8,15]

k =
3

Output

5

5.

5. Time Complexity:

Time Complexity = O(n log n)   Space Complexity = O(n)

6. Learning Outcome:

a. Learned how different problems have varying complexities, ranging from O(n) (Kadane's Algorithm) to O(n log n) (Merge Sort & Segment Tree).
b. Explored recursion-based solutions (Longest Nice Substring) and Divide & Conquer techniques (Reverse Pairs using Merge Sort).
c. Implemented Segment Tree for optimized Longest Increasing Subsequence II, reducing time complexity to O(n log n).
d. Applied Kadane's Algorithm for Maximum Subarray, Merge Sort-based counting for Reverse Pairs, and Segment Tree-based LIS for efficient computations.