



### Experiment 1.4

**Student Name:** Yuvraj Tripathi  
**Branch:** BE-CSE  
**Semester:** 6<sup>th</sup>  
**Subject Name:** Advanced programming  
**Lab II**

**UID:** 22BET10140  
**Section/Group:** 22BET\_IOT-703/ B  
**Date of Performance:** 13/02/25  
**Subject Code:** 22ITH-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:**

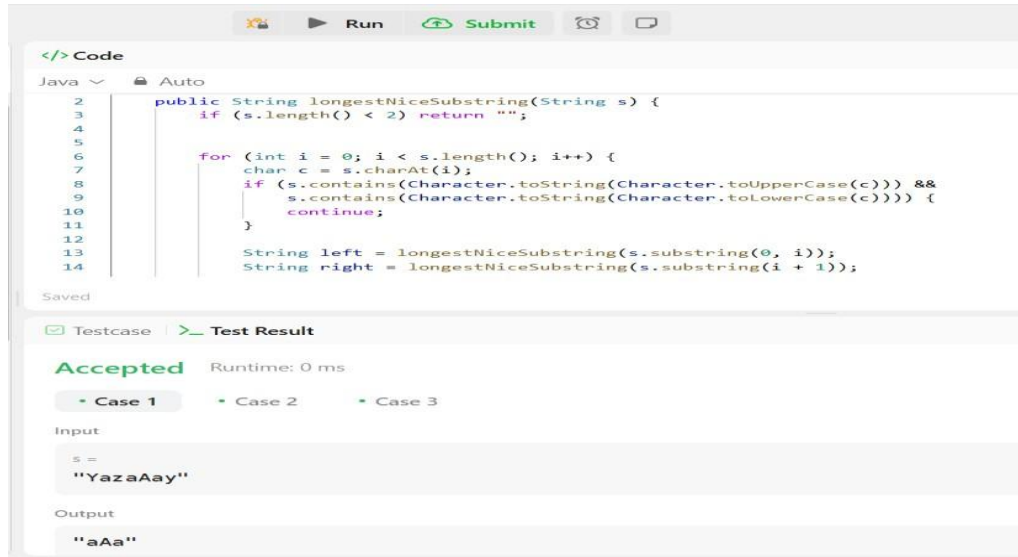
```
class Solution {
    public String longestNiceSubstring(String s) {
        if (s.length() < 2) return "";
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (s.contains(Character.toString(Character.toUpperCase(c))) && s.contains(Character.toString(Character.toLowerCase(c)))) {
                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:



The screenshot shows a code editor with a Java program and its test results. The code is as follows:

```
2 public String longestNiceSubstring(String s) {  
3     if (s.length() < 2) return "";  
4  
5  
6     for (int i = 0; i < s.length(); i++) {  
7         char c = s.charAt(i);  
8         if (s.contains(Character.toString(Character.toUpperCase(c))) &&  
9             s.contains(Character.toString(Character.toLowerCase(c)))) {  
10             continue;  
11         }  
12  
13         String left = longestNiceSubstring(s.substring(0, i));  
14         String right = longestNiceSubstring(s.substring(i + 1));
```

The test results show that the code is **Accepted** with a runtime of 0 ms. The input is "YazaAay" and the output is "aAa".

## PROBLEM 2:

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

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

3. **Code:**

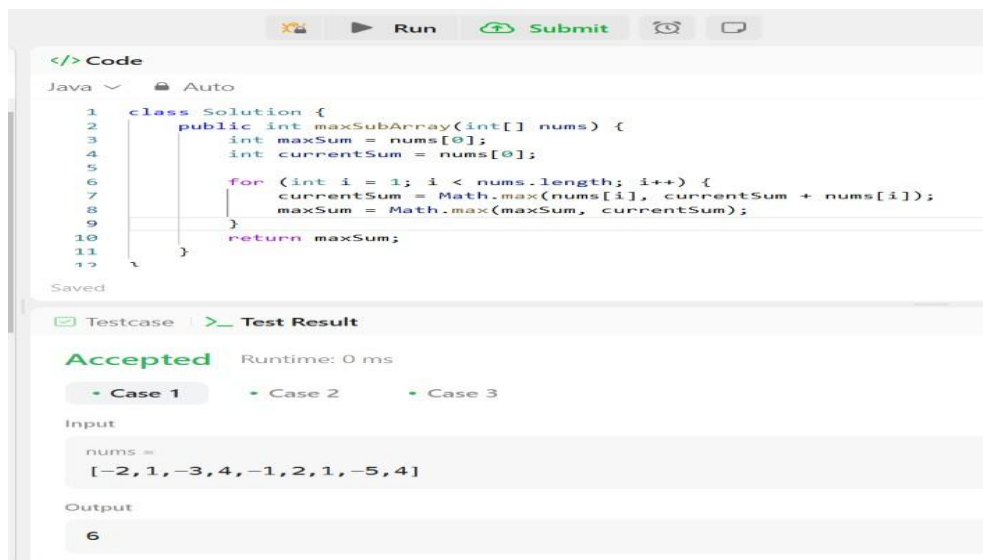
```
class Solution {  
    public int maxSubArray(int[] nums) {  
        int maxSum = nums[0];  
        int currentSum = nums[0];  
  
        for (int i = 1; i < nums.length; i++) {  
            currentSum = Math.max(nums[i], currentSum + nums[i]);  
            maxSum = Math.max(maxSum, currentSum);  
        }  
        return maxSum;  
    }  
}
```

#### 4. Time Complexity:

Time Complexity:  $O(n)$  (linear time)

Space Complexity:  $O(1)$  (constant space)

#### 5. Output:



```
</> Code
Java Auto
1 class Solution {
2     public int maxSubArray(int[] nums) {
3         int maxSum = nums[0];
4         int currentSum = nums[0];
5
6         for (int i = 1; i < nums.length; i++) {
7             currentSum = Math.max(nums[i], currentSum + nums[i]);
8             maxSum = Math.max(maxSum, currentSum);
9         }
10        return maxSum;
11    }
12}
Saved

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

### PROBLEM 3:

#### 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:

- $0 \leq i < j < \text{nums.length}$  and
- $\text{nums}[i] > 2 * \text{nums}[j]$ .

#### 3. Code:

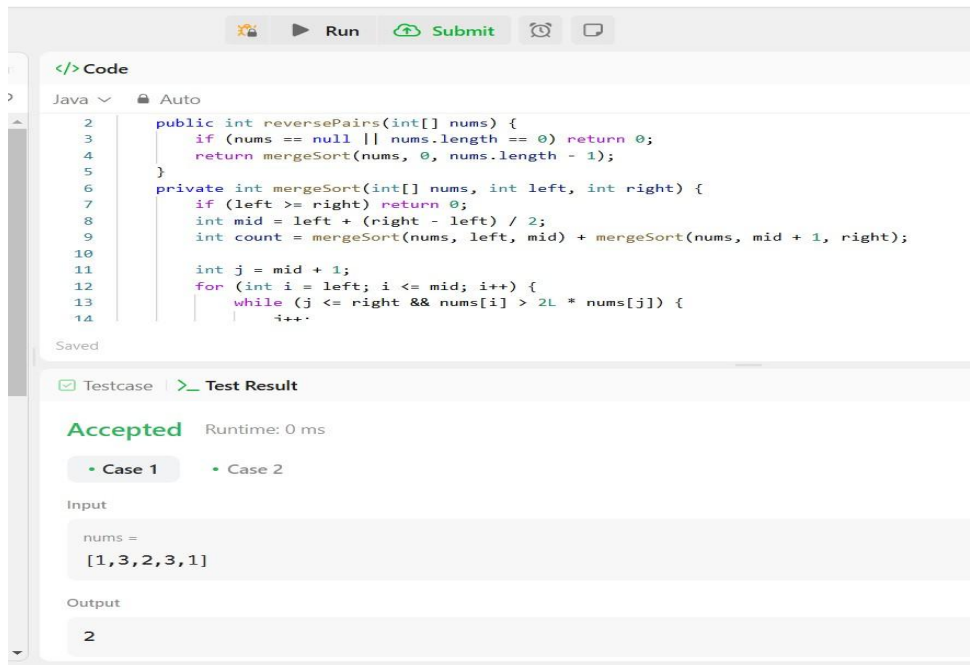
```
class Solution {
    public int reversePairs(int[] nums) {        if
(nums == null || nums.length == 0) return 0;
return mergeSort(nums, 0, nums.length - 1);
    }
    private int mergeSort(int[] nums, int left, int right) {
```

```
        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:



The screenshot shows a Java IDE with a code editor and a test result panel. The code implements a merge sort algorithm to find the number of reverse pairs in an array. The test result panel shows that the code was accepted with a runtime of 0 ms. The input array is [1, 3, 2, 3, 1] and the output is 2.

```
2 public int reversePairs(int[] nums) {
3     if (nums == null || nums.length == 0) return 0;
4     return mergeSort(nums, 0, nums.length - 1);
5 }
6 private int mergeSort(int[] nums, int left, int right) {
7     if (left >= right) return 0;
8     int mid = left + (right - left) / 2;
9     int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);
10
11     int j = mid + 1;
12     for (int i = left; i <= mid; i++) {
13         while (j <= right && nums[i] > 2L * nums[j]) {
14             j++;
15         }
16     }
17     return count;
18 }
```

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.

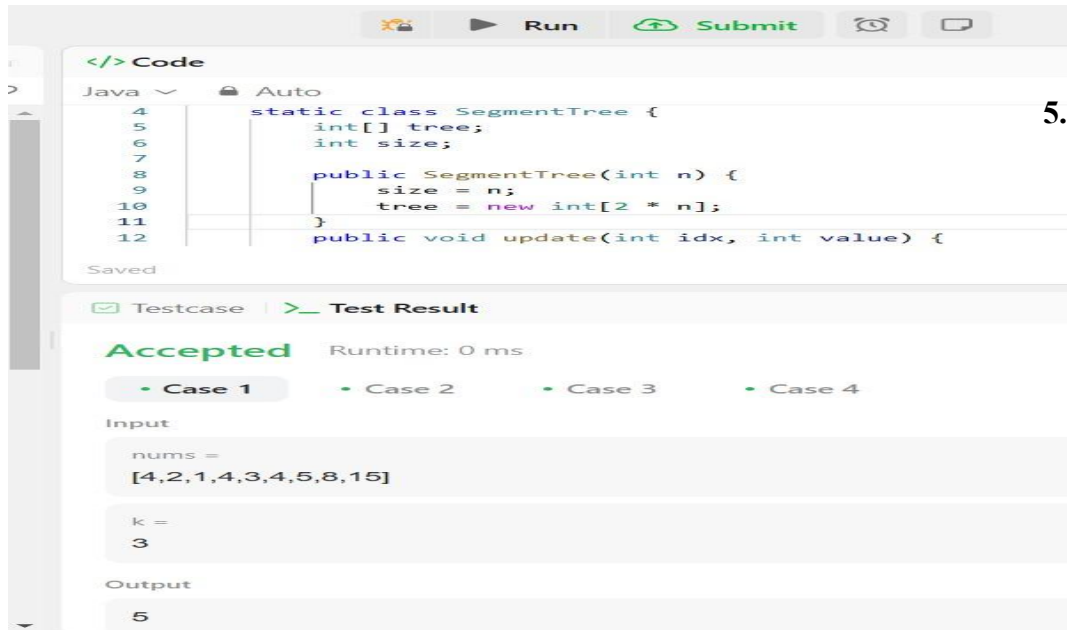
**3. Code:**

```
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:



The screenshot shows a code editor with Java code for a Segment Tree. The code defines a static class `SegmentTree` with attributes `tree` and `size`. It includes a constructor `SegmentTree(int n)` that initializes `size` to `n` and `tree` to a new array of size `2 * n`. It also has a method `update(int idx, int value)`. The code is saved. Below the code, the test results are shown as 'Accepted' with a runtime of 0 ms. The test case details for Case 1 are: Input `nums = [4,2,1,4,3,4,5,8,15]`, `k = 3`, and Output `5`.

```
4 static class SegmentTree {
5     int[] tree;
6     int size;
7
8     public SegmentTree(int n) {
9         size = n;
10        tree = new int[2 * n];
11    }
12    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. Time Complexity:

Time Complexity =  $O(n \log n)$

Space Complexity =  $O(n)$

#### 6. Learning Outcome:

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