Experiment 1.4

Student Name: Aditya Patel **UID:** 22BCS11543

Branch: BE-CSE **Section/Group:** 22BCS-IOT-640(B)

Semester: 6th **Date of Performance:** 14/02/25

Subject Name: ADVANCED PROGRAMMING LAB - 2 **Subject Code:** 22CSP-351

PROGRAM-1

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

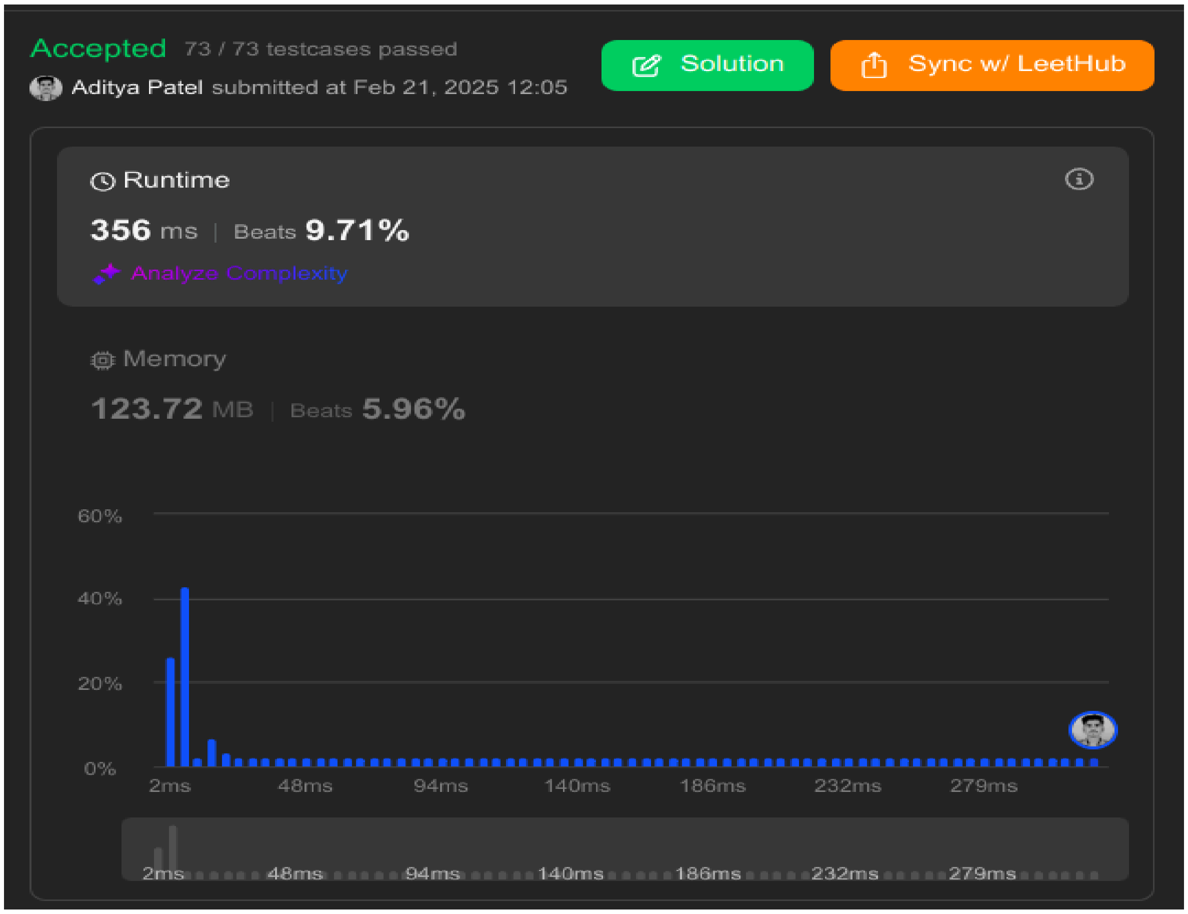
2) **Objective:** The objective of the program is to identify the longest "nice" substring in a given string, where a "nice" substring contains both uppercase and lowercase versions of every character. It achieves this by checking all possible substrings and validating their character case presence.

3) **Implementation/Code:**

```
class Solution {
public:
    bool isNiceSubstring(string& s, int i, int j) {
        unordered_map<int, bool> mp;
        bool isTrue = true;
        for (int x = i; x <= j; x++) {
            mp[s[x]] = true;
        }
        while (i <= j) {
            if (s[i] < 97) {
                if (!mp[s[i] + 32]) {
                    return false;
                }
            } else {
                if (!mp[s[i] - 32]) {
                    return false;
                }
            }
            i++;
        }
        return true;
    }
```

```
        string longestNiceSubstring(string s) {
            int n = s.length();
            int maxLength = 0;
            string result;
            for (int i = 0; i < n; i++) {
                for (int j = i; j < n; j++) {
                    if (isNiceSubstring(s, i, j)) {
                        if (j - i + 1 > maxLength) {
                            maxLength = j - i + 1;
                            result = s.substr(i, j - i + 1);
                        }
                    }
                }
            }
            return result;
        }
    };
```

4) Output:


Accepted 73 / 73 testcases passed
Aditya Patel submitted at Feb 21, 2025 12:05

Runtime
356 ms | Beats 9.71%

Memory
123.72 MB | Beats 5.96%

5) Learning Outcomes:

- **Identify Longest Substring:** Learners will be able to effectively identify the longest substring without repeating characters in a given string.
- **Implement Algorithms:** Participants will understand and implement various algorithms for substring searching, including sliding window and last index tracking methods.
- **Character Case Validation:** Learners will apply techniques for validating character cases within substrings, enhancing their string manipulation skills.
- **Enhance Problem-Solving Skills:** Participants will improve their problem-solving abilities by tackling challenges related to string manipulation and algorithm optimization.
- **Analyze Time and Space Complexity:** Learners will analyze the time and space complexity of different approaches to substring problems, enabling them to choose the most efficient solution.

PROGRAM-3

1) Aim: Reverse Pairs.

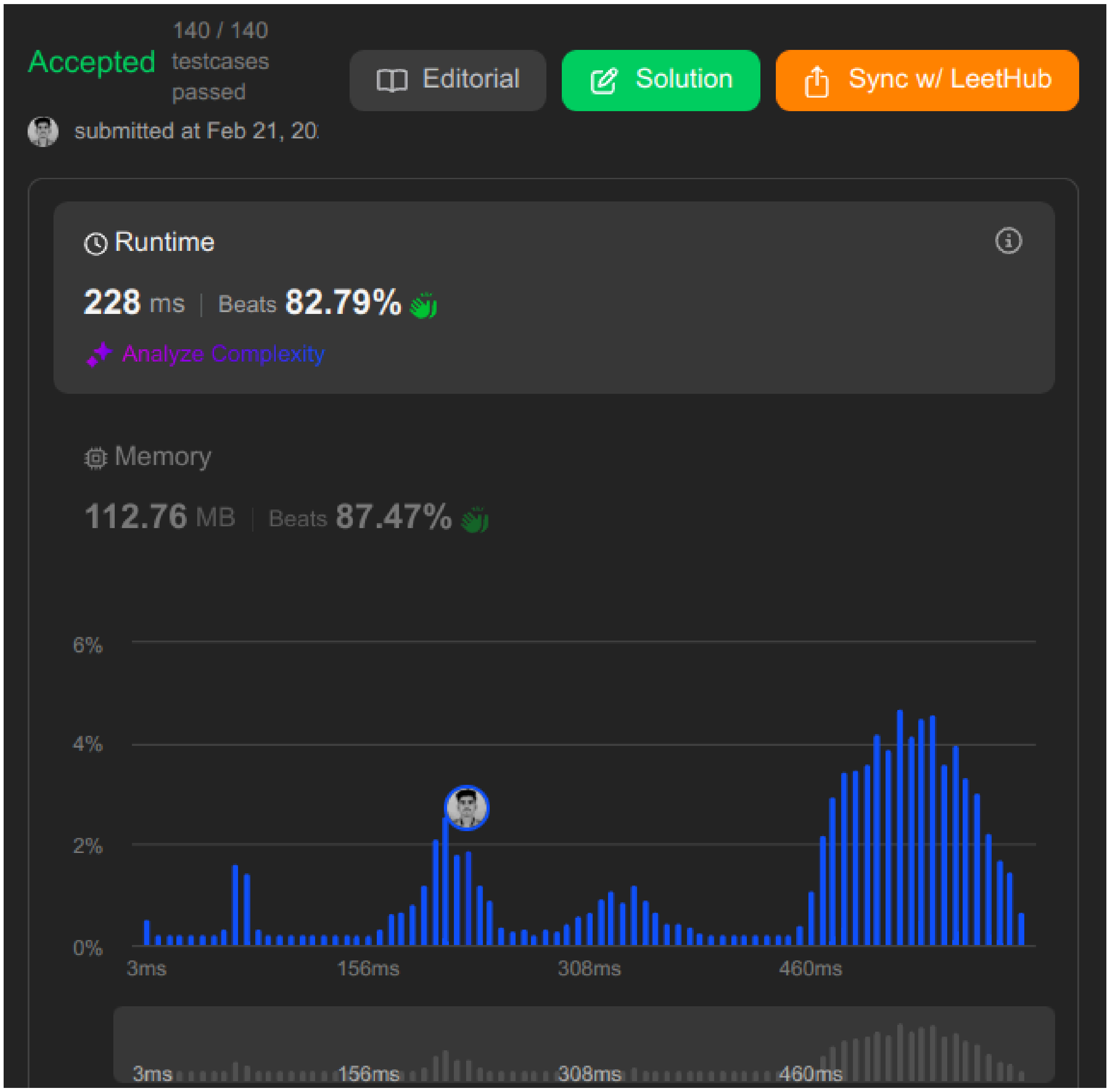
2) Objective: The objective of the reversePairs function is to count the number of reverse pairs in a given array of integers, where a reverse pair is defined as a pair of indices ((i, j)) such that (i < j) and (nums[i] > 2 \times nums[j]). This is achieved using a modified merge sort algorithm.

3) Implementation/Code:

```
class Solution {
private:
    void merge(vector<int>& nums, int low, int mid, int high, int& reversePairsCount){
        int j = mid+1;
        for(int i=low; i<=mid; i++){
            while(j<=high && nums[i] > 2*(long long)nums[j]){
                j++;
            }
            reversePairsCount += j-(mid+1);
        }
        int size = high-low+1;
        vector<int> temp(size, 0);
        int left = low, right = mid+1, k=0;
        while(left<=mid && right<=high){
            if(nums[left] < nums[right]){
                temp[k++] = nums[left++];
            }
            else{
                temp[k++] = nums[right++];
            }
        }
        while(left<=mid){
            temp[k++] = nums[left++];
        }
        while(right<=high){
            temp[k++] = nums[right++];
        }
        int m=0;
        for(int i=low; i<=high; i++){
            nums[i] = temp[m++];
        }
    }

    void mergeSort(vector<int>& nums, int low, int high, int& reversePairsCount){
        if(low >= high){
            return;
        }
        int mid = (low + high) >> 1;
        mergeSort(nums, low, mid, reversePairsCount);
        mergeSort(nums, mid+1, high, reversePairsCount);
        merge(nums, low, mid, high, reversePairsCount);
    }
public:
    int reversePairs(vector<int>& nums) {
        int reversePairsCount = 0;
        mergeSort(nums, 0, nums.size()-1, reversePairsCount);
        return reversePairsCount;
    }
};
```

4) Output:



Accepted 140 / 140 testcases passed

submitted at Feb 21, 20

Runtime

228 ms | Beats 82.79%

Memory

112.76 MB | Beats 87.47%

5) Learning Outcomes:

- **Implement Merge Sort:** Learners will understand and implement the merge sort algorithm to efficiently sort an array while counting specific conditions (reverse pairs).
- **Count Reverse Pairs:** Participants will learn how to count reverse pairs during the merge step of the merge sort, enhancing their understanding of algorithmic problem-solving.
- **Analyze Algorithm Efficiency:** Learners will analyze the time and space complexity of the merge sort approach, gaining insights into the efficiency of divide-and-conquer algorithms.

PROGRAM-3

1) Aim: Reverse Pairs.

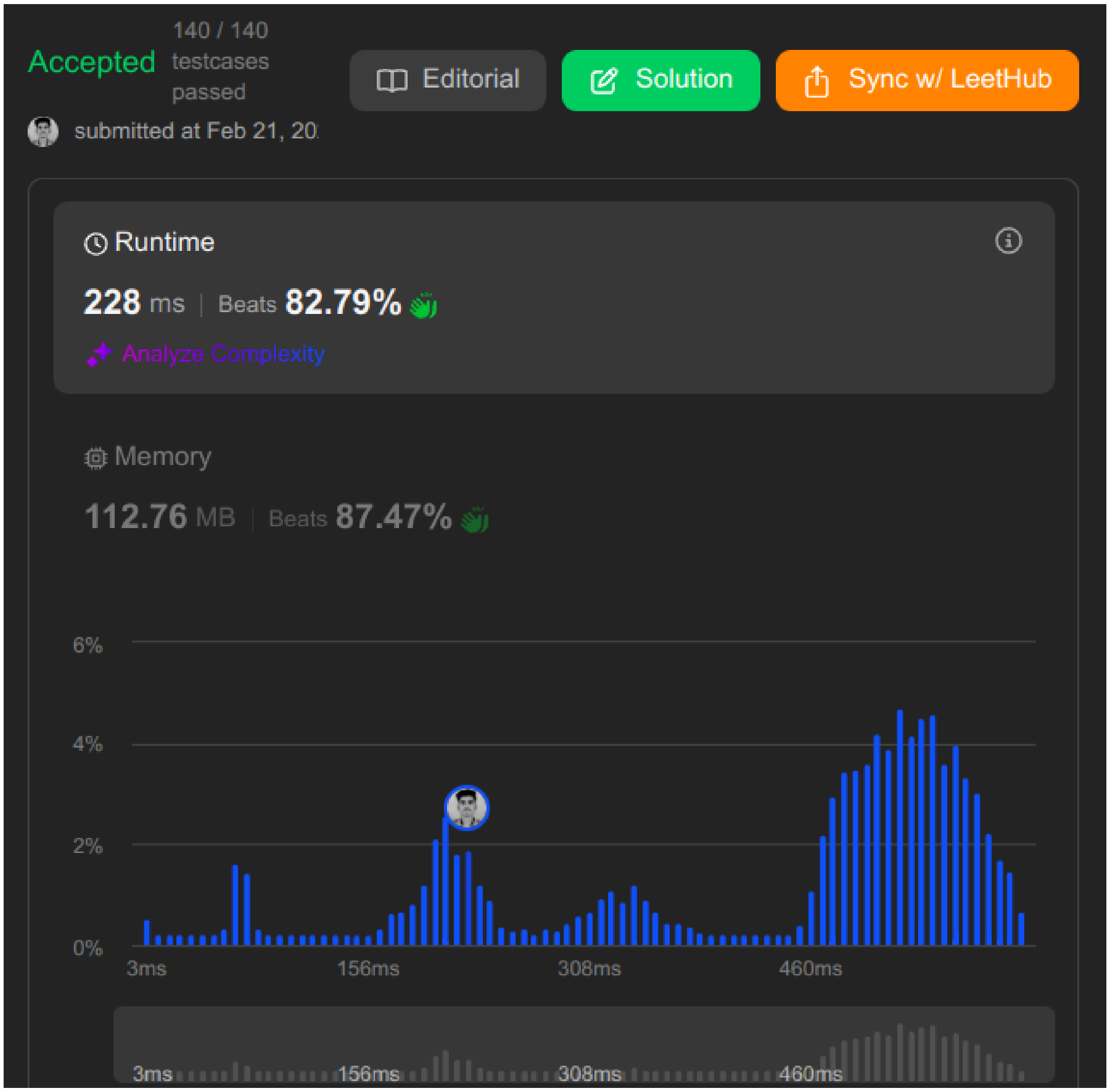
2) Objective: The objective of the reversePairs function is to count the number of reverse pairs in a given array of integers, where a reverse pair is defined as a pair of indices ((i, j)) such that (i < j) and (nums[i] > 2 \times nums[j]). This is achieved using a modified merge sort algorithm.

3) Implementation/Code:

```
class Solution {
private:
    void merge(vector<int>& nums, int low, int mid, int high, int& reversePairsCount){
        int j = mid+1;
        for(int i=low; i<=mid; i++){
            while(j<=high && nums[i] > 2*(long long)nums[j]){
                j++;
            }
            reversePairsCount += j-(mid+1);
        }
        int size = high-low+1;
        vector<int> temp(size, 0);
        int left = low, right = mid+1, k=0;
        while(left<=mid && right<=high){
            if(nums[left] < nums[right]){
                temp[k++] = nums[left++];
            }
            else{
                temp[k++] = nums[right++];
            }
        }
        while(left<=mid){
            temp[k++] = nums[left++];
        }
        while(right<=high){
            temp[k++] = nums[right++];
        }
        int m=0;
        for(int i=low; i<=high; i++){
            nums[i] = temp[m++];
        }
    }

    void mergeSort(vector<int>& nums, int low, int high, int& reversePairsCount){
        if(low >= high){
            return;
        }
        int mid = (low + high) >> 1;
        mergeSort(nums, low, mid, reversePairsCount);
        mergeSort(nums, mid+1, high, reversePairsCount);
        merge(nums, low, mid, high, reversePairsCount);
    }
public:
    int reversePairs(vector<int>& nums) {
        int reversePairsCount = 0;
        mergeSort(nums, 0, nums.size()-1, reversePairsCount);
        return reversePairsCount;
    }
};
```

4) Output:


Accepted 140 / 140 testcases passed
submitted at Feb 21, 20

Runtime
228 ms | Beats 82.79%

Memory
112.76 MB | Beats 87.47%

5) Learning Outcomes:

- **Implement Merge Sort:** Learners will understand and implement the merge sort algorithm to efficiently sort an array while counting specific conditions (reverse pairs).
- **Count Reverse Pairs:** Participants will learn how to count reverse pairs during the merge step of the merge sort, enhancing their understanding of algorithmic problem-solving.
- **Analyze Algorithm Efficiency:** Learners will analyze the time and space complexity of the merge sort approach, gaining insights into the efficiency of divide-and-conquer algorithms.