Experiment 4

Student Name: GAYTRI **UID: 22BET10198**
Branch: BE-IT **Section/Group: 22BET_IOT_701/B**
Semester: 6th **Date of Performance:19-2-25**
Subject Name: AP Lab-II **Subject Code: 22ITP-351**

Problem 1. 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.

Algorithm:

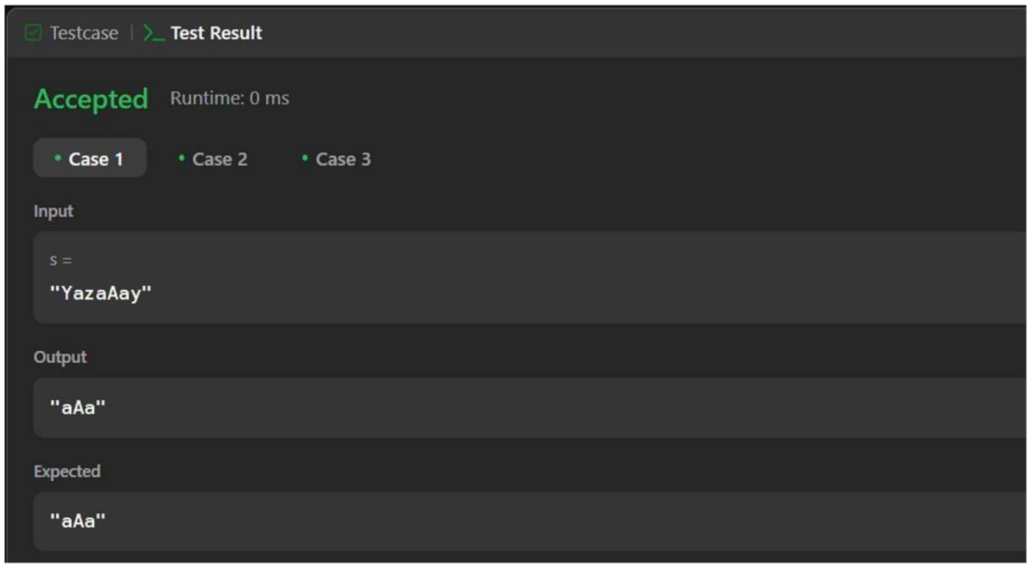
1. **Base Case:**
- If the length of s is less than 2, return an empty string ("") because a single character cannot be "nice".
2. **Create a Set of Characters in s:**
- Store all characters of s in a hash set for quick lookup.
3. **Find a Split Point:**
- Traverse through s. If a character appears in only one case (i.e., either uppercase or lowercase but not both), this means the substring cannot be nice.
- Use this character as a **split point** and break s into two substrings:
  - left = s [0: i]
  - right = s [i+1:]
- Recursively find the longest "nice" substring in both parts.
4. **Compare Substrings:**
- Return the longer of the two substrings found in step 3.
5. **If No Split Occurs:**
- If no character was found that requires splitting, return s itself as it is already "nice".

Code:

```
class Solution {
public:
    string longestNiceSubstring(string s) {
        if (s.length() < 2) return "";

    unordered_set<char> charSet(s.begin(), s.end());

    for (int i = 0; i < s.length(); i++) {
        if (charSet.count(tolower(s[i])) && charSet.count(toupper(s[i]))) {
            continue;
        }

        string left = longestNiceSubstring(s.substr(0, i));
        string right = longestNiceSubstring(s.substr(i + 1));
```

```
            return (left.length() >= right.length()) ? left : right;
        }

        return s;
        }
    };
```

Output:



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

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

Algorithm:

1. **Initialize Variables**
   - maxsum = nums[0] → Stores the maximum subarray sum found so far.
   - currsum = 0 → Tracks the sum of the current subarray.
2. **Iterate Through the Array**
   - If currsum becomes negative, reset it to 0 (since a negative sum reduces the potential maximum).
   - Add the current element num to currsum.
   - Update maxsum = max(maxsum, currsum).

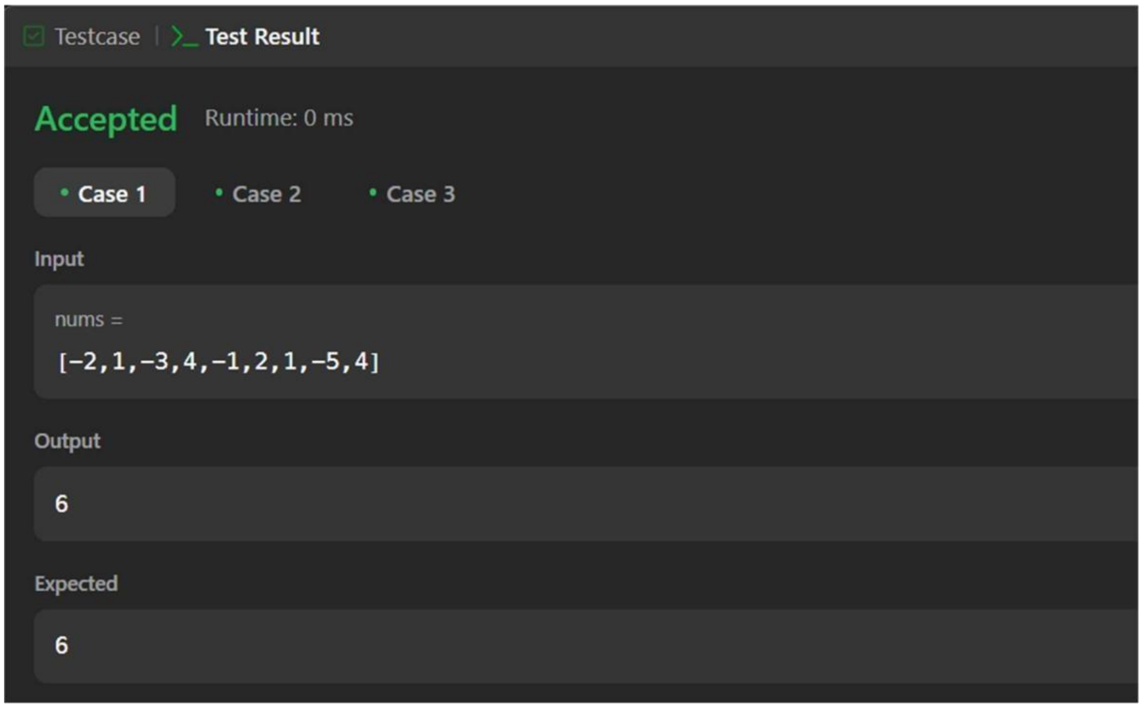
1. **Return maxsum**, which stores the maximum subarray sum.

Code:

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxsum = nums[0];
        int currsum = 0;

    for (int num : nums) {

        if (currsum < 0) {
            currsum = 0;
        }
```

```
            currsum += num;

            maxsum = max(maxsum, currsum);
        }

        return maxsum;
    }
};
```

Output:



```
Accepted   Runtime: 0 ms

Case 1   Case 2   Case 3

Input
nums =
[-2,1,-3,4,-1,2,1,-5,4]

Output
6

Expected
6
```