Experiment 9

Student Name: Akshant Kumar **UID:22BCS13418**
Branch: CSE **Section/Group:NTPP-IOT603/B**
Semester: 6th **Date of Performance:11-5-25**
Subject Name: AP Lab-2 **Subject Code:** 22CSP-351

Problem -1

1. Aim: Two Sum:

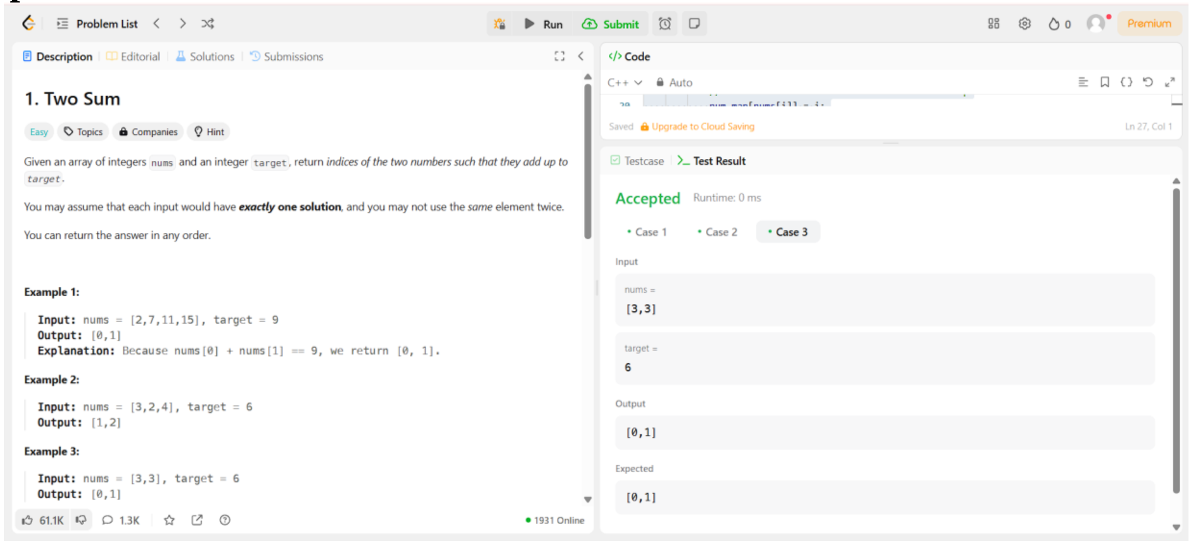
2. Objective:

- **Find Indices of Two Numbers**: Locate two distinct numbers in the input array (nums) whose sum equals a given target value (target).
- **Efficient Algorithm**: Use a hash map (unordered_map) to achieve fast lookups while iterating through the array. This avoids the need for nested loops, making the solution efficient with a time complexity of O(n).
- **Correct and Unique Solution**

3. Implementation/Code:

```
#include <unordered_map>
#include <vector>

class Solution {
public:
    std::vector<int> twoSum(std::vector<int>& nums, int target) {
        // Create a hash map to store the indices of the numbers
        std::unordered_map<int, int> num_map;

        for (int i = 0; i < nums.size(); i++) {
            // Calculate the complement needed to reach the target
            int complement = target - nums[i];

            // Check if the complement exists in the hash map
            if (num_map.find(complement) != num_map.end()) {
                return {num_map[complement], i};
            }

            // Add the number and its index to the hash map
            num_map[nums[i]] = i;
        }

        // Return an empty vector if no solution is found (shouldn't happen per problem constraints)
        return {};
    }
};
```

4. Output:



1. Two Sum

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to `target`*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Example 3:

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

```
Accepted   Runtime: 0 ms
Case 1   Case 2   Case 3
Input
nums =
[3,3]
target =
6
Output
[0,1]
Expected
[0,1]
```

Figure 1

5. Learning Outcome:

- **Algorithm Design**: Learn to break down problems logically and optimize solutions for efficiency.
- **Problem-Solving Skills**: Build the ability to analyze constraints and select suitable data structures for implementation.
- **C++ Practical Implementation**: Gain hands-on experience with C++ features like vectors, loops, and STL (e.g., unordered_map).
- **Coding Interview Preparation**: Strengthen coding skills and familiarity with algorithmic problems commonly asked in interviews.

Problem-2

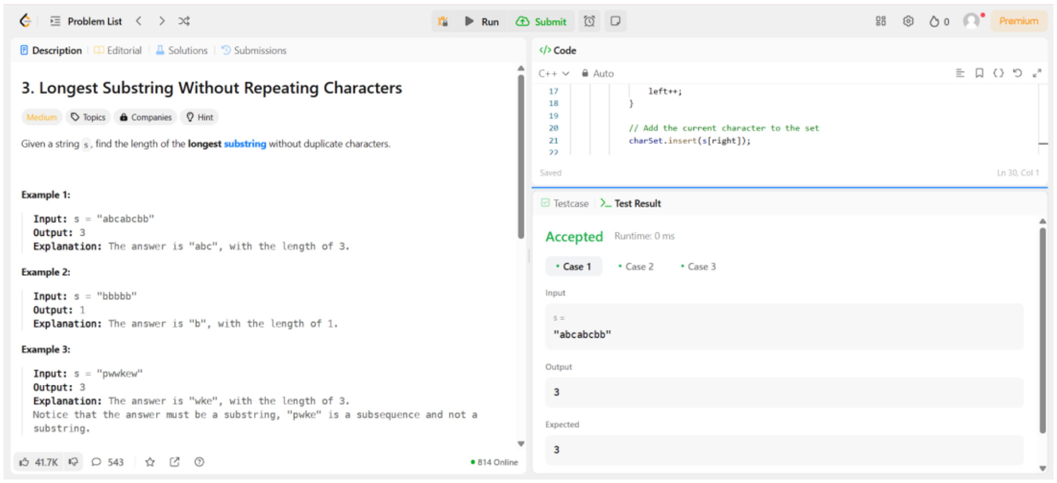
1. **Aim:** Longest Substring Without Repeating Characters
2. **Objectives:**

   The objective of this code is to find the length of the longest substring in a given string s that contains no duplicate characters. It achieves this by efficiently utilizing the sliding window technique and a hash set, ensuring optimal performance with a time complexity of **O(n)**

3. **Implementation/Code:**

```
#include <unordered_set>
#include <string>
using namespace std;

class Solution {
public:
   int lengthOfLongestSubstring(string s) {
      // Use a sliding window approach with a set to track characters
      unordered_set<char> charSet;
      int maxLength = 0;
      int left = 0; // Left pointer of the sliding window

      for (int right = 0; right < s.size(); right++) {
         // If the current character is already in the set, remove from the left until it's no longer a duplicate
         while (charSet.find(s[right]) != charSet.end()) {
            charSet.erase(s[left]);
            left++;
         }

         // Add the current character to the set
         charSet.insert(s[right]);

         // Calculate the length of the current substring and update maxLength
         maxLength = max(maxLength, right - left + 1);
      }

      return maxLength;
   }
};
```

4. Output:



3. Longest Substring Without Repeating Characters

Given a string `s`, find the length of the **longest substring** without duplicate characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a
substring.
```

```
            left++;
        }

        // Add the current character to the set
        charSet.insert(s[right]);
```

Saved   Ln 30, Col 1

Accepted   Runtime: 0 ms

Case 1   Case 2   Case 3

Input

```
s =
"abcabcbb"
```

Output

```
3
```

Expected

```
3
```

Figure 2

5. Learning Outcomes:

- **Efficient String Processing**: Learn to optimize algorithms for processing strings, ensuring high performance even for large inputs.
- **Sliding Window Technique**: Master the concept of using two pointers to dynamically adjust the range of a substring or subarray.
- **Hash Set Usage**: Understand how to use a hash set for quick lookups and efficient duplicate detection in a substring.
- **Time Complexity Analysis**: Develop skills to analyze and optimize the time complexity of algorithms, reducing it from O(n^2) to O(n).
- **Edge Case Handling**: Enhance problem-solving abilities by accounting for edge cases, like empty strings or strings with all identical characters.
- **Practical C++ Implementation**: Build hands-on experience with STL components (unordered_set) and explore how they simplify code.
- **Interview Readiness**: Prepare effectively for technical coding interviews by solving a common problem often asked in such settings.

Problem – 3

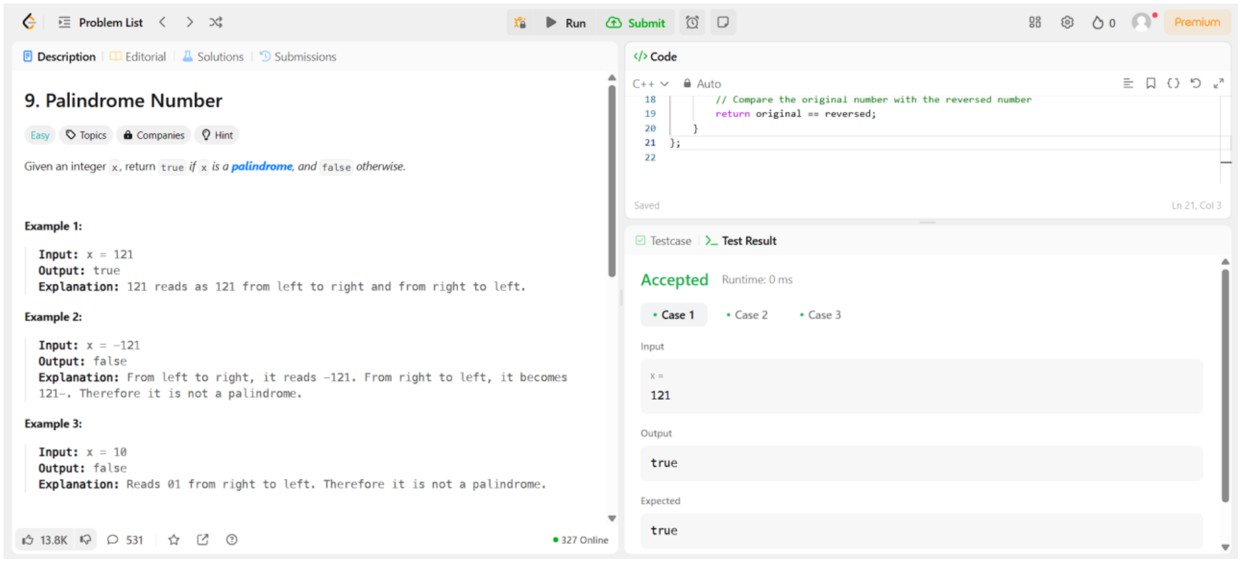
1. **Aim:** Palindrome Number
2. **Objectives:**

   The objective of this code is to determine whether a given integer x is a palindrome. A palindrome is a number that reads the same forwards and backwards. The algorithm achieves this by reversing the digits of the input number and comparing the reversed result with the original number to validate the symmetry. It handles constraints like negative numbers, which cannot be palindromes, and ensures efficiency while avoiding overflow.

3. **Implementation/Code:**

```
class Solution {
public:
    bool isPalindrome(int x) {
        // Negative numbers cannot be palindromes
        if (x < 0) return false;

        // Initialize variables for reverse calculation
        long reversed = 0; // Use 'long' to prevent overflow
        int original = x;  // Store the original number

        // Reverse the digits of the number
        while (x != 0) {
            int digit = x % 10;       // Extract the last digit
            reversed = reversed * 10 + digit; // Append the digit to 'reversed'
            x /= 10;                  // Remove the last digit
        }

        // Compare the original number with the reversed number
        return original == reversed;
    }
};
```

4. **Output:**



9. Palindrome Number

Given an integer `x`, return `true` *if* `x` *is a* ***palindrome****, and* `false` *otherwise.*

Example 1:

```
Input: x = 121
Output: true
Explanation: 121 reads as 121 from left to right and from right to left.
```

Example 2:

```
Input: x = -121
Output: false
Explanation: From left to right, it reads -121. From right to left, it becomes
121-. Therefore it is not a palindrome.
```

Example 3:

```
Input: x = 10
Output: false
Explanation: Reads 01 from right to left. Therefore it is not a palindrome.
```

```cpp
        // Compare the original number with the reversed number
        return original == reversed;
    }
};
```

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
x =
121
```

Output

```
true
```

Expected

```
true
```

Figure 3

5. Learning Outcomes

- **Efficient Peak Finding**: - You will learn how to locate a peak element without scanning the entire array, using a smarter approach with binary search.
- **Mastering Binary Search Variations**: - You will understand how binary search can be adapted for different problems beyond simple number searching.
- **Developing a Logical Approach**: - You will improve your ability to break down problems logically, making it easier to apply efficient solutions in coding interviews and real-world tasks.
- **Understanding Search Space Reduction**: - You will gain insights into how reducing the search space step by step can lead to significant performance improvements.
- **Building Optimized and Scalable Solutions**:- You will develop the skills to write code that is both time-efficient and scalable, a crucial requirement for competitive programming and software development.

Problem – 4

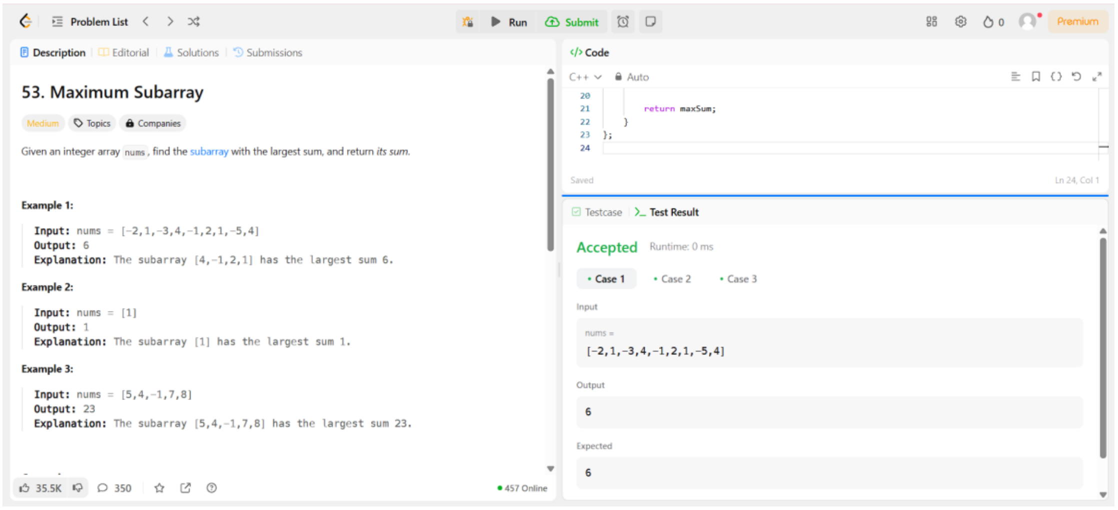
1. **Aim:** Maximum Subarray
2. **Objectives:**

   The objective of this code is to find the sum of the subarray within a given integer array nums that has the largest sum. It uses Kadane's Algorithm to efficiently calculate this, ensuring an optimal solution with a time complexity of **O(n)**. This allows the code to handle large arrays while adhering to the problem constraints.

3. **Implementation/Code:**

```
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        // Initialize the maximum sum and the current sum
        int maxSum = nums[0];
        int currentSum = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            // Update the current sum to include the current element,
            // or restart at the current element if currentSum becomes negative
            currentSum = max(nums[i], currentSum + nums[i]);

            // Update maxSum to be the largest value found so far
            maxSum = max(maxSum, currentSum);
        }

        return maxSum;
    }
};
```

4. **Output:**



53. Maximum Subarray

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

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.
```

Example 2:

```
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
```

Example 3:

```
Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.
```

```
        return maxSum;
    }
};
```

Saved Ln 24, Col 1

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
```

Figure 4

5. Learning Outcomes

- **Kadane's Algorithm**: Understand and apply this algorithm to solve problems involving the maximum subarray sum efficiently.
- **Dynamic Programming Concept**: Learn how to solve problems by breaking them into subproblems, using intermediate results to build the final solution.
- **Optimization Techniques**: Develop skills to optimize brute-force approaches ($O(n^2)$) into linear time complexity ($O(n)$) using a single traversal.
- **Edge Case Management**: Enhance problem-solving by addressing special cases, such as arrays with all negative numbers or a single element.
- **Mathematical Intuition**: Build intuition to decide whether to continue a subarray or start fresh when adding a new element.
- **C++ Practical Application**: Practice coding with common C++ features like vectors, loops, and the max function for concise, readable solutions.
- **Coding Interview Preparation**: Strengthen readiness for technical interviews with exposure to a commonly asked algorithmic problem.