LeetCode Experience Report

Introduction

LeetCode is a platform that provides a collection of coding problems to enhance programming skills, particularly in preparation for technical interviews. This report summarizes my LeetCode experience, focusing on three problems of varying difficulty levels: one easy, one medium, and one hard. For each problem, I will describe the problem statement, my approach, and provide the solution in C++.

Problem 1: Easy - Two Sum

Problem Statement: Given an array of integers `nums` and an integer `target`, return the indices of the two numbers such that they add up to `target`. Assume that each input would have exactly one solution, and you may not use the same element twice.

Approach:

1. Create an unordered map to store the value and its index as we iterate through the array.
2. For each element in the array, calculate the complement (i.e., `target - nums[i]`).
3. Check if the complement exists in the map:
   - If it does, return the indices of the current element and the complement.
   - If it doesn't, add the current element and its index to the map.
4. This approach ensures that each element is processed only once, achieving O(n) time complexity.

Solution in C++:

```
#include <vector>
#include <unordered_map>
class Solution {
public:
  std::vector<int> twoSum(std::vector<int>& nums, int target) {
    std::unordered_map<int, int> map;
    for (int i = 0; i < nums.size(); ++i) {
      int complement = target - nums[i];
      if (map.find(complement) != map.end()) {
        return {map[complement], i};
      }
      map[nums[i]] = i;
    }
    return {};
  }
};
```

Problem 2: Medium - Longest Substring Without Repeating Characters

Problem Statement: Given a string `s`, find the length of the longest substring without repeating characters.

Approach:

1. Use a sliding window approach with two pointers, `left` and `right`, to represent the current substring.
2. Use an unordered map to store characters and their indices.
3. Iterate through the string with the `right` pointer:
   - If the character at `right` is already in the map and its index is greater than or equal to `left`, move the `left` pointer to `map[s[right]] + 1`.
   - Update the character's index in the map.
   - Calculate the length of the current substring and update the maximum length.
4. This approach ensures that each character is processed only once, achieving O(n) time complexity.

Solution in C++:

```
#include <string>
#include <unordered_map>

class Solution {
public:
  int lengthOfLongestSubstring(std::string s) {
    std::unordered_map<char, int> map;
    int maxLength = 0, left = 0;
    for (int right = 0; right < s.size(); ++right) {
      if (map.find(s[right]) != map.end() && map[s[right]] >= left) {
        left = map[s[right]] + 1;
      }
      map[s[right]] = right;
      maxLength = std::max(maxLength, right - left + 1);
    }
    return maxLength;
  }
};
```

Problem 3: Hard - Median of Two Sorted Arrays

Problem Statement: Given two sorted arrays `nums1` and `nums2` of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Approach:

1. Ensure that `nums1` is the smaller array. If not, swap `nums1` and `nums2`.
2. Use binary search on the smaller array to find the partition:
   - Calculate `partitionX` and `partitionY` to divide the arrays into left and right halves.
   - Adjust the partitions based on the values at the boundaries to ensure that all elements in the left half are less than or equal to those in the right half.
3. Once the correct partition is found, calculate the median based on the total number of elements (even or odd).
4. This approach achieves the desired O(log (m+n)) time complexity.

Solution in C++:

```
#include <vector>
#include <algorithm>

class Solution {
public:
  double findMedianSortedArrays(std::vector<int>& nums1, std::vector<int>& nums2) {
    if (nums1.size() > nums2.size()) {
      return findMedianSortedArrays(nums2, nums1);
    }

    int x = nums1.size();
    int y = nums2.size();
    int low = 0, high = x;

    while (low <= high) {
      int partitionX = (low + high) / 2;
      int partitionY = (x + y + 1) / 2 - partitionX;

      int maxX = (partitionX == 0) ? INT_MIN : nums1[partitionX - 1];
      int minX = (partitionX == x) ? INT_MAX : nums1[partitionX];

      int maxY = (partitionY == 0) ? INT_MIN : nums2[partitionY - 1];
      int minY = (partitionY == y) ? INT_MAX : nums2[partitionY];

      if (maxX <= minY && maxY <= minX) {
        if ((x + y) % 2 == 0) {
          return (std::max(maxX, maxY) + std::min(minX, minY)) / 2.0;
        } else {
                return std::max(maxX, maxY);
            }
        } else if (maxX > minY) {
            high = partitionX - 1;
        } else {
            low = partitionX + 1;
        }
    }

    throw std::invalid_argument("Input arrays are not sorted");
  }
};
```