Experiment 2

Student Name: Umesh
Branch: CSE
Semester: 6th
Subject: AP-II

UID: 22BCS17270
Section: 627-A
DOP: 24/01/2025
Subject Code: 22CSP-351

Problem 1.2.1: Two Sum

1. **Aim-** Given an array of integers nums and an integer target, return the indices of the two numbers such that they add up to target. Each input has exactly one solution, and you cannot use the same element twice.
2. **Algorithm –**
    1. Initialize an empty hash map (dict).
    2. Iterate through the nums array:
        a. For each element num, calculate the complement: complement = target - num.
        b. Check if the complement exists in the hash map:
            i. If it does, return the indices of the complement and the current number.
            ii. If it doesn't, add the current number and its index to the hash map.
    3. Return the indices of the two numbers that add up to the target.

3. Code-

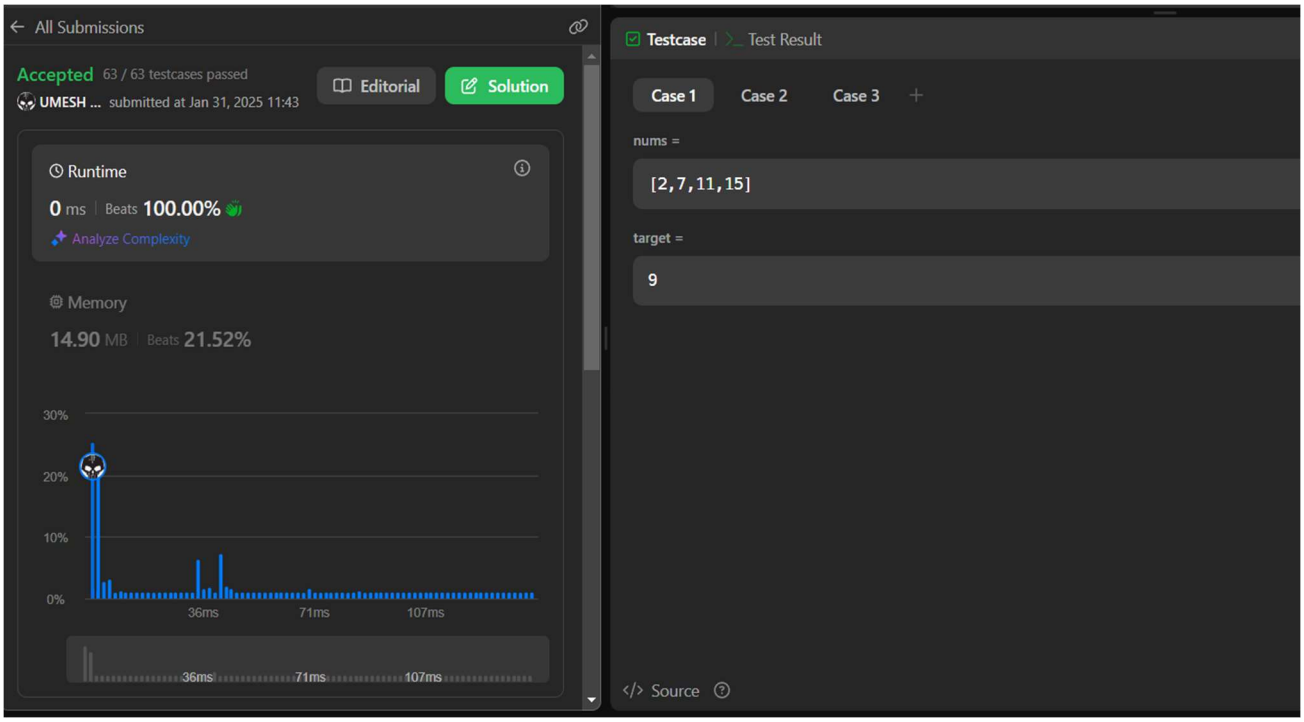
Code

C++ Auto

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

Saved Ln 7, Col 40

4. Output –



5. Time Complexity –

- Time Complexity: O(n), where n is the number of elements in the nums array. This is because we only iterate through the array once.
- Space Complexity: O(n), since we store each number and its index in the hash map.

Problem 1.2.2: Jump Game II

1. Aim - You are given a 0-indexed array nums of length n. You are initially positioned at nums[0]. Each element nums[i] represents the maximum length of a forward jump from index i. Return the minimum number of jumps to reach nums[n - 1].

2. Algorithm-

1. Initialize jumps as 0 (the number of jumps needed), current_end as 0 (the farthest index you can reach in the current jump), and farthest as 0 (the farthest index you can reach so far).
2. Loop through each index in nums (except the last index):
    a. Update farthest to the maximum of farthest and i + nums[i].
    b. If the current index i is the last index of the current jump (i == current_end):
        i. Increment the jumps.
        ii. Set current_end to farthest.
3. Return jumps when current_end reaches or exceeds the last index (n - 1).

3. Code –

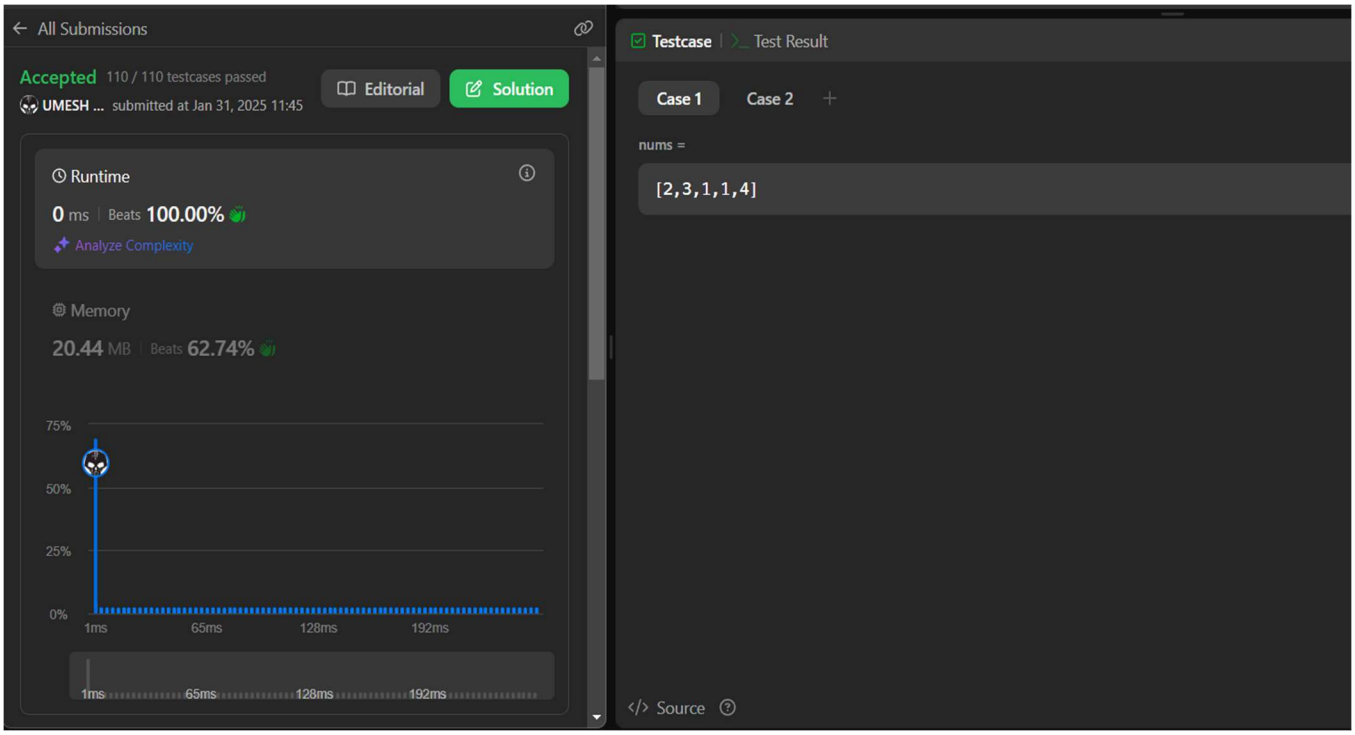
Code

C++ Auto

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

class Solution {
public:
    int jump(vector<int>& nums) {
        int n = nums.size();
        if (n <= 1) return 0;

        int jumps = 0, curEnd = 0, farthest = 0;

        for (int i = 0; i < n - 1; i++) {
            farthest = max(farthest, i + nums[i]);

            if (i == curEnd) {
                jumps++;
                curEnd = farthest;
                if (curEnd >= n - 1) break;
            }
        }

        return jumps;
    }
};
```

Saved Ln 8, Col 31

4. Output-



5. Time Complexity –

- Time Complexity: O(n), where n is the number of elements in the nums array. We iterate through the array once.
- Space Complexity: O(1), since we use only a few extra variables.

6. Learning Outcomes -

i. Using a hash map (unordered_map in C++ / HashMap in Java) to store visited numbers for O(1) average-time complexity lookup.

ii. Understanding how to store values vs indices to return correct results

iii. Understanding Greedy Strategy-Keep track of the farthest reachable index.Iterate through the array, updating the maximum reach.

iv. Using a Single-pass vs Two-pass Approach -Two-pass-Store elements first, then find pairs. Single-pass to Store and check simultaneously.