Experiment 2

Student Name: Shubham **UID: 22BCS15853**
Branch: BE-CSE **Section/Group: KRG 2 B**
Semester: 6th **Date of Performance: 15/01/2025**
Subject Name: Advanced Programming Lab-2 **Subject Code: 22CSP-351**

Program 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. **Objective:** The goal is to identify two distinct elements in the given array of integers whose sum equals the specified target value. The program must return the indices of these two elements, ensuring that each input has exactly one solution. The indices can be returned in any order, and the same array element cannot be used more than once. This problem emphasizes efficient searching techniques to achieve an optimal solution.

3. **Algorithm:**

a. **Initialize HashMap:** Create a HashMap to store numbers as keys and their indices as values for quick lookups.
b. **Iterate Through the Array:** Traverse the array using a loop with index i.
c. **Calculate Complement:** For each element nums[i], calculate its complement as complement = target - nums[i].
d. **Check HashMap for Complement:**
   - If the complement exists as a key in the HashMap, it means a pair is found.
   - Return the indices of the complement (from the HashMap) and the current index i.
e. **Update HashMap**: If the complement is not found, store the current element nums[i] along with its index i in the HashMap.
f. **Continue Searching:** Repeat the process for all elements in the array until a solution is found.
g. **Handle Edge Case:** If the loop completes without finding a solution (although guaranteed in the problem), throw an exception.

4. Code:

```
import java.util.HashMap;

public class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> map = new HashMap<>();

       for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];

            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i };
            }

            map.put(nums[i], i);
       }

 throw new IllegalArgumentException("No two sum solution");
       }
}
```

5. Output:

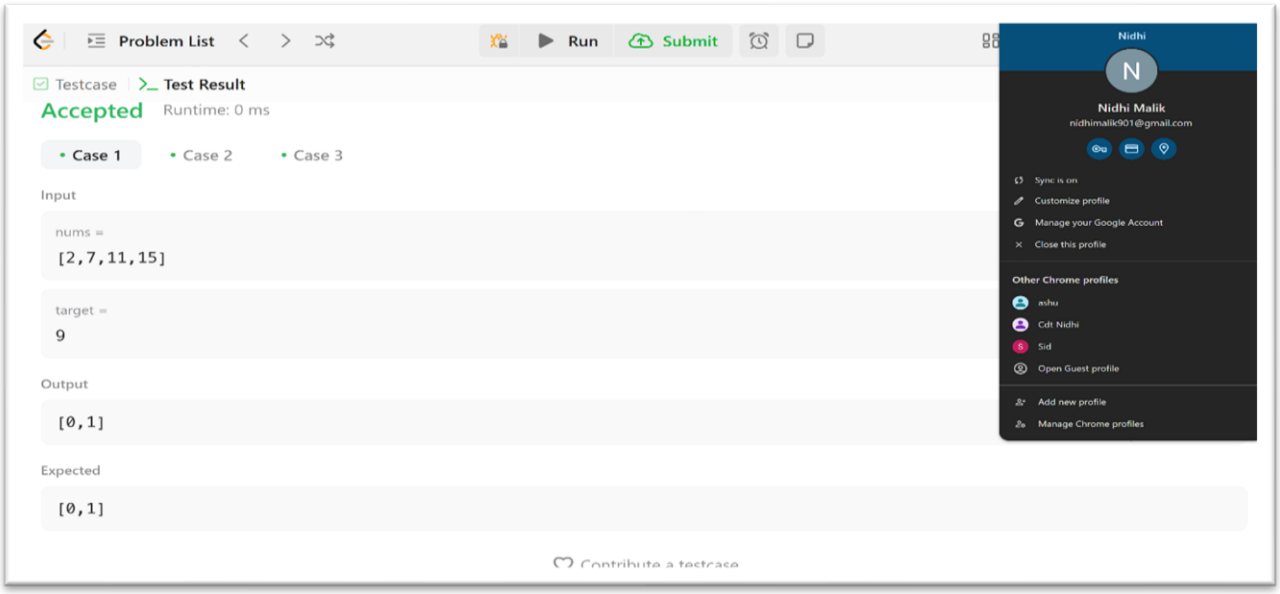

Figure.1

Program 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. **Objective:** The objective is to find the minimum number of jumps needed to reach the last index of the array. Each element in the array represents the maximum forward jump length from that index. You start at the first index and aim to reach the last index using the least number of jumps. Use a greedy approach to track the farthest reachable index and increment the jump count whenever the current jump's range ends. Stop when the last index is within the current jump range. Return the total jump count.

3. **Algorithm:**

a. **Initialize Variables:**
   - jumps to count the number of jumps (initialize as 0).
   - current_end to track the farthest index reachable in the current jump.
   - farthest to track the maximum index reachable at any point.

b. **Iterate Over the Array:**

   Loop through the array from index 0 to n - 2 (excluding the last index, as it is the destination).

c. **Update the Farthest Reachable Index:**At each index i, calculate farthest as the maximum of farthest and i + nums[i].

d. **Check If a Jump is Required:** If the current index i equals current_end, it means the end of the current jump's range is reached.

e. **Increment the Jump Count:** Increase the jumps count by 1 and update current_end to farthest.

f. **Break Condition:** If current_end becomes greater than or equal to n - 1 (the last index), break out of the loop as the destination is reachable.

g. **Return the Result:** After exiting the loop, return the total number of jumps.

4. Code:

```
class Solution {
public:
int jump(vector& nums) {
  int n = nums.size();
  int jumps = 0, current_end = 0, farthest = 0;
  for (int i = 0; i < n - 1; i++) {
      farthest = max(farthest, i + nums[i]);
      if (i == current_end) {
          jumps++;
         current_end = farthest;
         if (current_end >= n - 1) break;
      }
  }
  return jumps;
}
};
```

5. Output:

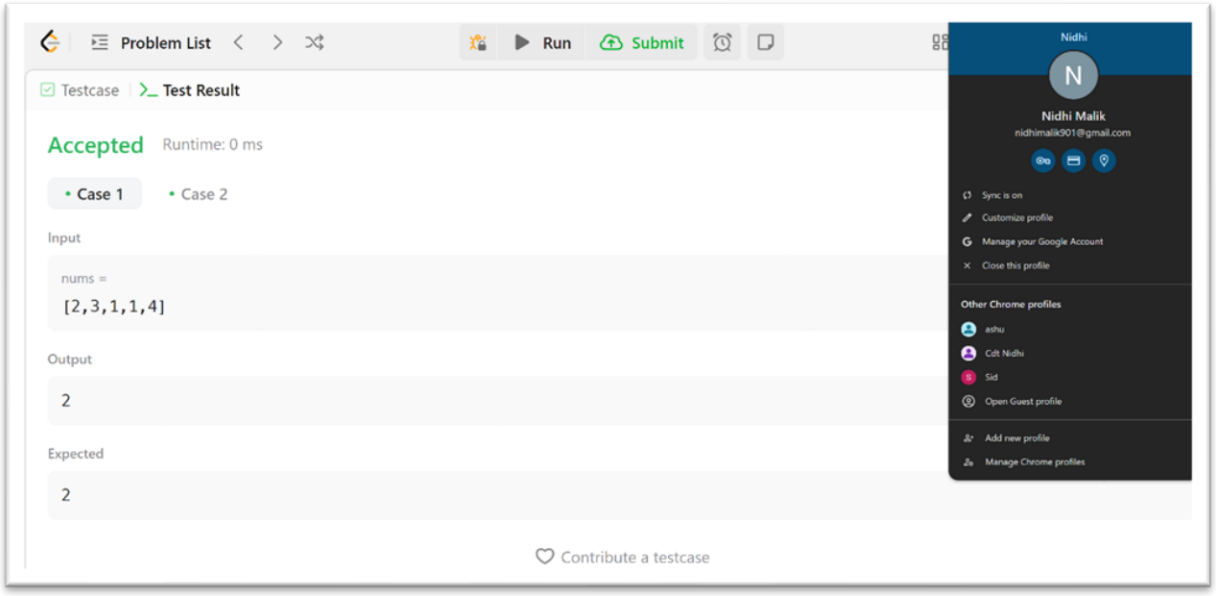

Figure.2

6. **Learning Outcomes:**
   i. **Algorithm Efficiency**: Learn to design efficient algorithms with time complexities of **O(n)** for solving problems like finding indices (Two Sum) and calculating minimum jumps (Jump Game).
   ii. **Array Processing Skills**: Develop skills in manipulating arrays for tasks such as range-based traversal (Jump Game) and complement computation (Two Sum).
   iii. **Techniques Mastery**: Understand and apply **Greedy Algorithms** for range-based problems and **HashMap-based lookups** for quick data access.
   iv. **Problem Decomposition**: Gain the ability to break down complex problems into smaller logical steps, such as tracking ranges or calculating complements.
   v. **Edge Case Handling**: Learn to identify and handle edge cases, such as small array sizes or guaranteed solutions, effectively in both problems.