



Experiment 1

Student Name: Shristi Rai

Branch: CSE

Semester: 6th

Subject Name: AP-II LAB

UID: 22BCS15330

Section/Group: NTPP-602-A

Date of Performance: 20-01-24

Subject Code: 22CSP-352

PROBLEM 1

1. Aim:

Two Sum Problem

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:

- Understand how to identify and retrieve indices of two numbers in an array that sum up to a given target.
- Develop an efficient approach to solve the problem using a hash map for improved time complexity.
- Learn to handle constraints like not reusing the same element and ensuring a unique solution.

3. Algorithm:

1. Initialize an empty hash map (dict).
2. Iterate through the `nums` array:
 - For each element `num`, calculate the complement: `complement = target - num`.
 - Check if the complement exists in the hash map:

- If it does, return the indices of the complement and the current number.
 - 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.

4. Implementation/Code:

```
import java.util.HashMap;

class Solution {

    public static int[] twoSum(int[] nums, int target) {

        // Create a HashMap to store the numbers and their indices
        HashMap<Integer, Integer> numMap = new HashMap<>();

        // Iterate through the array
        for (int i = 0; i < nums.length; i++) {

            int complement = target - nums[i];

            // Check if the complement exists in the HashMap
            if (numMap.containsKey(complement)) {

                return new int[] { numMap.get(complement), i };

            }

            // Add the current number and its index to the HashMap
            numMap.put(nums[i], i);

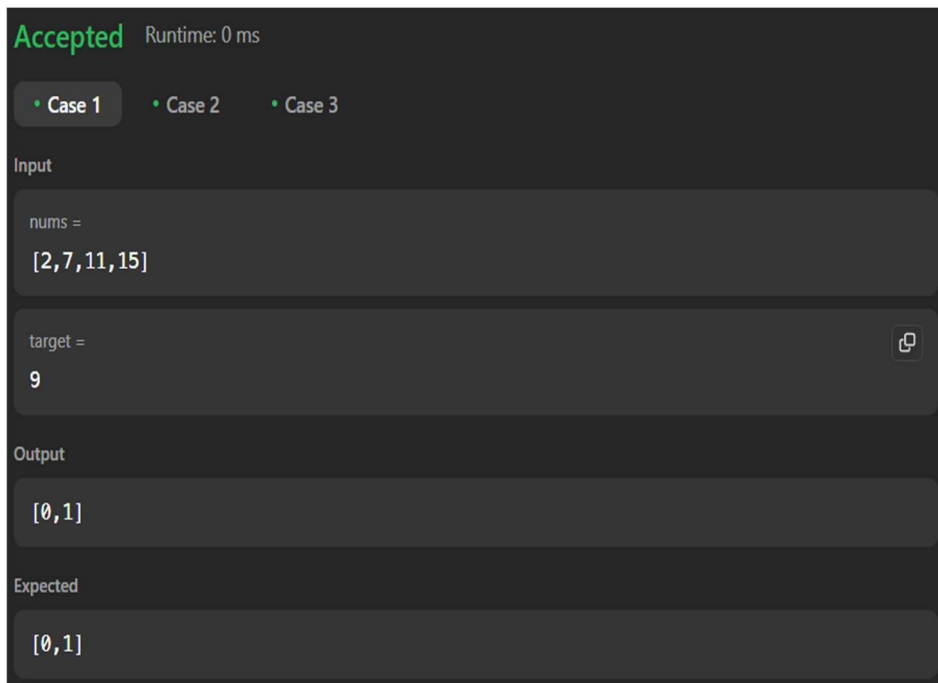
        }

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

}
```

```
public static void main(String[] args) {  
    int[] nums1 = {2, 7, 11, 15};  
    int target1 = 9;  
    int[] result1 = twoSum(nums1, target1);  
    System.out.println("Output: [" + result1[0] + ", " + result1[1] + "]);  
  
    int[] nums2 = {3, 2, 4};  
    int target2 = 6;  
    int[] result2 = twoSum(nums2, target2);  
    System.out.println("Output: [" + result2[0] + ", " + result2[1] + "]);  
}  
}
```

5. Output:



The screenshot shows a code execution interface with a dark theme. At the top, it says "Accepted" in green and "Runtime: 0 ms". Below this, there are three tabs: "Case 1", "Case 2", and "Case 3", with "Case 1" being the active tab. The "Input" section contains two fields: "nums =" with the value "[2,7,11,15]" and "target =" with the value "9". The "Output" section shows the result "[0,1]". The "Expected" section also shows "[0,1]".

```
Accepted Runtime: 0 ms  
• Case 1 • Case 2 • Case 3  
Input  
nums =  
[2,7,11,15]  
target =  
9  
Output  
[0,1]  
Expected  
[0,1]
```

• Case 1 • Case 2 • Case 3

Input

nums =
[3,2,4]

target =
6

Output

[1,2]

Expected

[1,2]

6. Time Complexity: $O(n)$, where n is the number of elements in the nums array.

7. Space Complexity: $O(n)$, since we store each number and its index in the hash map.

8. Learning Outcomes:

- Ability to implement an optimized solution for the Two Sum problem with a time complexity of $O(n)$.
- Confidence in using hash maps for lookup-based operations in algorithm design.
- Enhanced problem-solving skills for tackling array-based challenges with constraints.

PROBLEM 2

1. Aim:

Implement a FIFO queue using two stacks. The queue should support all the functions of a normal queue: push, peek, pop, and empty.

2. Objective:

- Understand how to identify and retrieve indices of two numbers in an array that sum up to a given target.
- Develop an efficient approach to solve the problem using a hash map for improved time complexity.
- Learn to handle constraints like not reusing the same element and ensuring a unique solution.

3. Algorithm:

1. Use two stacks: stack1 for pushing elements, stack2 for popping elements.
2. For push(x), simply push x onto stack1.
3. For pop(), if stack2 is empty, transfer all elements from stack1 to stack2. Then, pop the top element from stack2.
4. For peek(), if stack2 is
5. empty, transfer all elements from stack1 to stack2. Then, return the top element of stack2.
6. For empty(), check if both stack1 and stack2 are empty.

4. Implementation/Code:

```
import java.util.Stack;
```

```
class MyQueue {
```

```
private Stack<Integer> stack1; // Input stack
private Stack<Integer> stack2; // Output stack

public MyQueue() {
    stack1 = new Stack<>();
    stack2 = new Stack<>();
}

// Push element x to the back of the queue
public void push(int x) {
    stack1.push(x);
}

// Removes the element from the front of the queue and returns it
public int pop() {
    if (stack2.isEmpty()) {
        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }
    return stack2.pop();
}

// Get the front element
public int peek() {
    if (stack2.isEmpty()) {
        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }
    return stack2.peek();
}
```

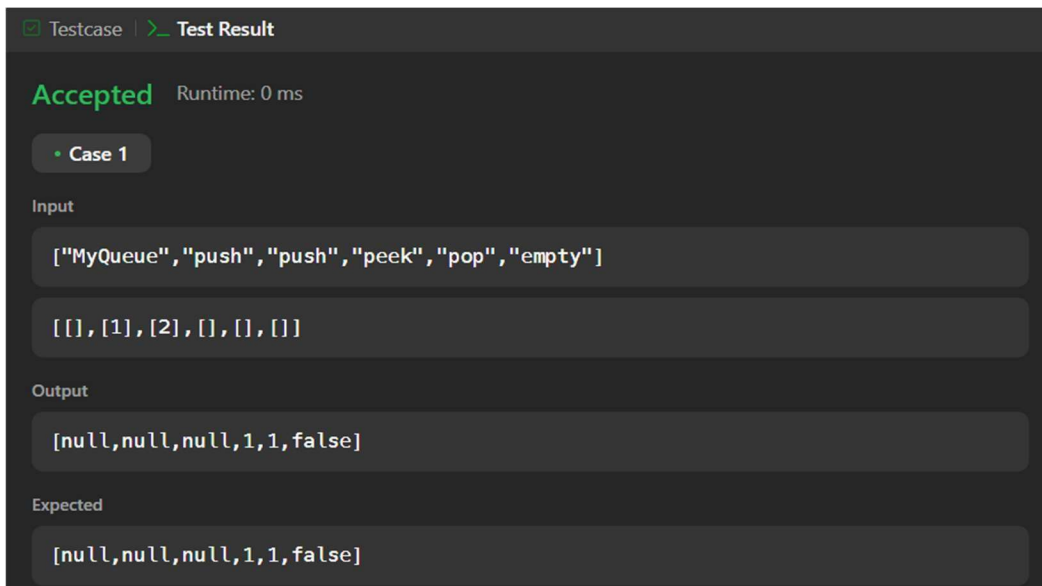
```
// Returns whether the queue is empty
public boolean empty() {
    return stack1.isEmpty() && stack2.isEmpty();
}

}

public class Main {
    public static void main(String[] args) {
        MyQueue myQueue = new MyQueue();
        myQueue.push(1); // queue is: [1]
        myQueue.push(2); // queue is: [1, 2]

        System.out.println(myQueue.peek()); // Output: 1
        System.out.println(myQueue.pop()); // Output: 1
        System.out.println(myQueue.empty()); // Output: false
    }
}
```

5. Output:



The screenshot displays a test result interface with a dark theme. At the top, there are tabs for 'Testcase' and 'Test Result', with 'Test Result' being the active tab. Below the tabs, the word 'Accepted' is shown in green, followed by 'Runtime: 0 ms'. A section titled 'Case 1' is expanded, showing the test details. Under the 'Input' section, there are two input fields: the first contains the command sequence ["MyQueue", "push", "push", "peek", "pop", "empty"] and the second contains the initial state of the queue, represented as an array of lists: [[], [1], [2], [], [], []]. The 'Output' section shows a single output field containing the result array: [null, null, null, 1, 1, false]. The 'Expected' section shows a single expected output field containing the same result array: [null, null, null, 1, 1, false].

Testcase | > Test Result

Accepted Runtime: 0 ms

• Case 1

Input

["MyQueue", "push", "push", "peek", "pop", "empty"]

[[], [1], [2], [], [], []]

Output

[null, null, null, 1, 1, false]

Expected

[null, null, null, 1, 1, false]

6. Time Complexity:

- $\text{push}(x)$: $O(1)$
- $\text{pop}()$: $O(n)$ in the worst case when elements are transferred from stack1 to stack2 .
- $\text{peek}()$: $O(n)$ in the worst case.
- $\text{empty}()$: $O(1)$

7. Space Complexity: $O(n)$, since we use two stacks to store the elements.

8. Learning Outcomes:

- Ability to implement an optimized solution for the Two Sum problem with a time complexity of $O(1)$.
- Confidence in using hash maps for lookup-based operations in algorithm design.
- Enhanced problem-solving skills for tackling array-based challenges with constraints.