Name: Jhaveri Varun Nimitt

UID: 2023800042

Batch: CSE A Batch C

Experiment No.: 1

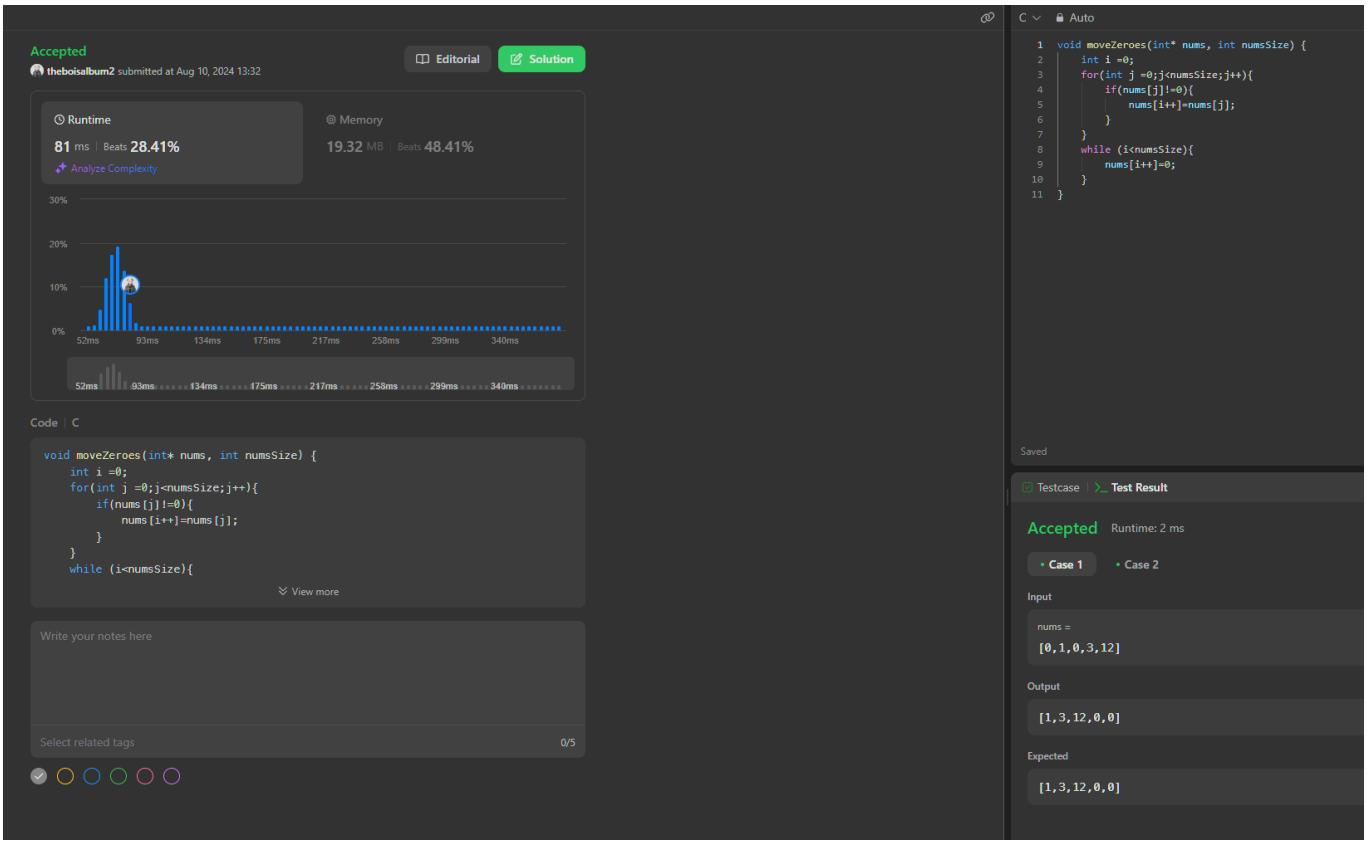
Aim: Arrays as Data Structure

Problem: **283**. Move Zeroes : Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Program:

```
void moveZeroes(int* nums, int numsSize) {
    int i =0;
    for(int j =0;j<numsSize;j++){
        if(nums[j]!=0){
            nums[i++]=nums[j];
        }
    }
    while (i<numsSize){
        nums[i++]=0;
    }
}
```

Output:



Initialization:

- i = 0 is initialized to track the position where the next non-zero element should be placed.

Observation

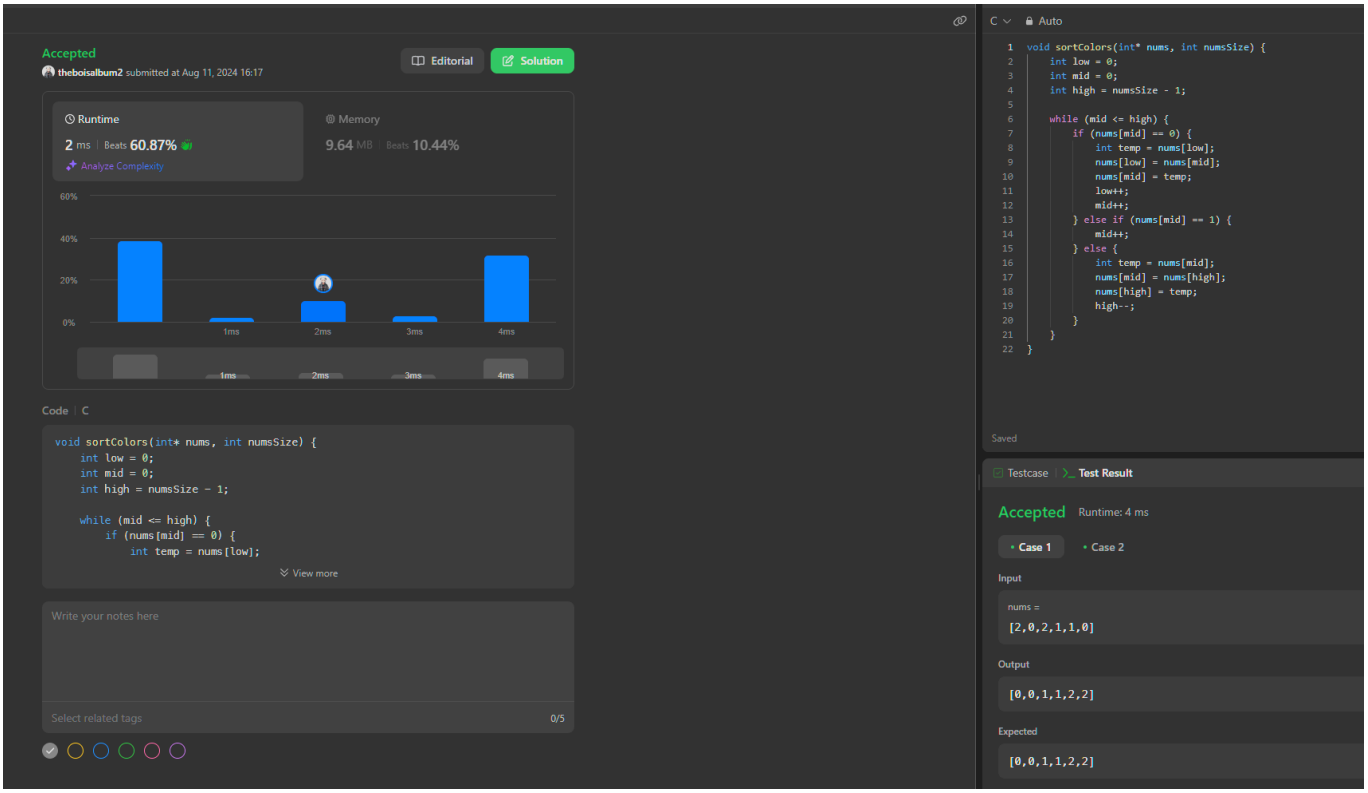
- my code successfully moves all zeros to the end of the array while preserving the order of non-zero elements.
- The final output for the given input is [1, 3, 12, 0, 0], achieved in a single pass with a subsequent fill of zeros, which is as expected in **testcase1**

Problem: **75**. Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.You must solve this problem without using the library's sort function.

Program:

```
void sortColors(int* nums, int numsSize) {
    int low = 0;
    int mid = 0;
    int high = numsSize - 1;

    while (mid <= high) {
        if (nums[mid] == 0) {
            if (mid != low) {
                int temp = nums[low];
                nums[low] = nums[mid];
                nums[mid] = temp;
            }
            low++;
            mid++;
        } else if (nums[mid] == 1) {
            mid++;
        } else {
            if (mid != high) {
                int temp = nums[mid];
                nums[mid] = nums[high];
                nums[high] = temp;
            }
            high--;
        }
    }
}
```

Output:



solution uses dutch flag algorithm

Observations:

- ➔ Initialization:
  - ◆ low = 0, mid = 0, and high = numsSize - 1
  - ◆ low tracks the position where the next 0 should be placed.
  - ◆ mid is the current element being evaluated.
  - ◆ high tracks the position where the next 2 should be placed.

    (this is dutch flag algorithm concept)

- my code efficiently sorts the array by grouping all 0s, 1s, and 2s together.
- The final sorted output for the given input is [0, 0, 1, 1, 2, 2], achieved in a single pass through the array with constant space; this is as expected as in **testcase1**.

Problem: **188**.You are given an integer array prices where prices[i] is the price of a given stock on the ith day, and an integer k.

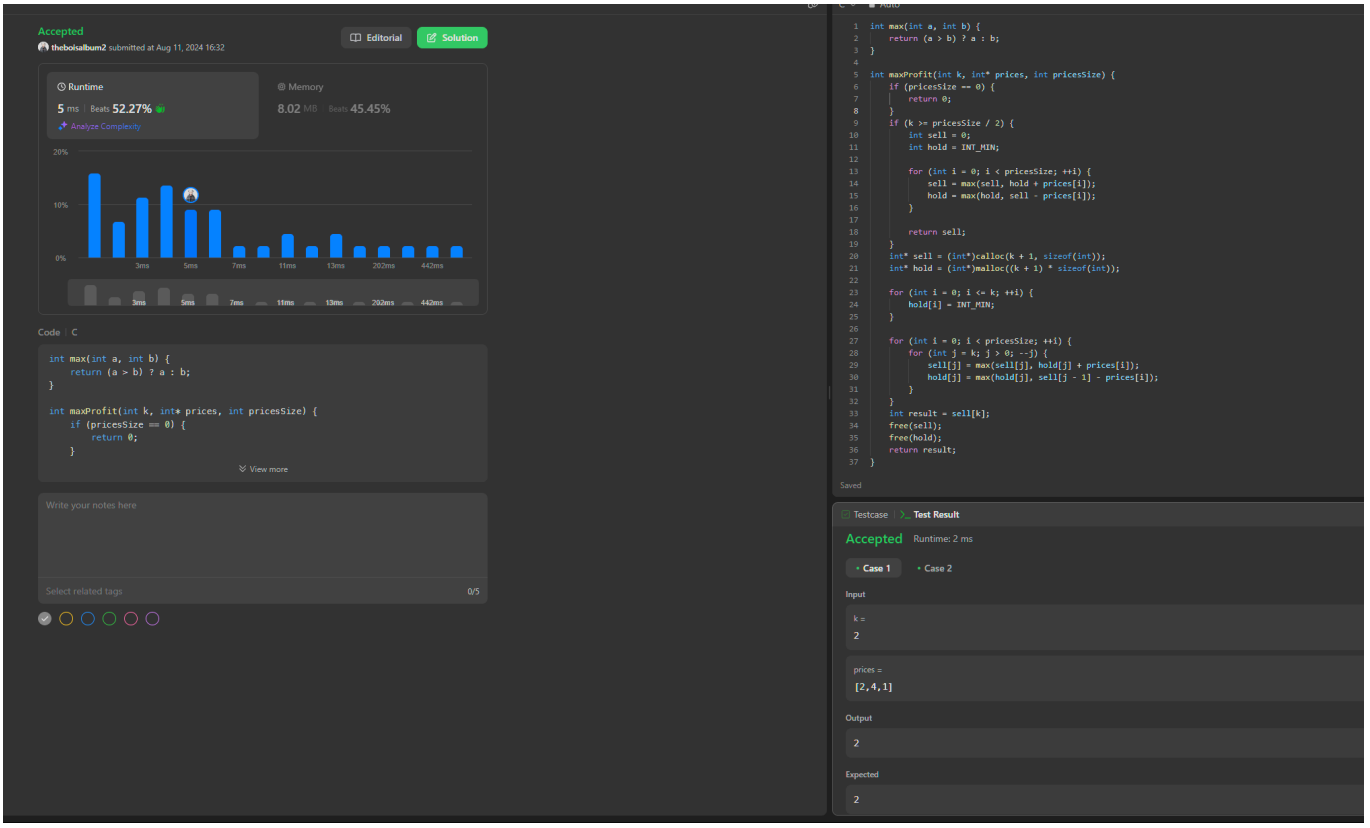
Find the maximum profit you can achieve. You may complete at most k transactions: i.e. you may buy at most k times and sell at most k times.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Program:

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

int maxProfit(int k, int* prices, int pricesSize) {
    if (pricesSize == 0) {
        return 0;
    }
    if (k >= pricesSize / 2) {
        int sell = 0;
        int hold = INT_MIN;

        for (int i = 0; i < pricesSize; ++i) {
            sell = max(sell, hold + prices[i]);
            hold = max(hold, sell - prices[i]);
        }

        return sell;
    }
    int* sell = (int*)calloc(k + 1, sizeof(int));
    int* hold = (int*)malloc((k + 1) * sizeof(int));

    for (int i = 0; i <= k; ++i) {
        hold[i] = INT_MIN;
    }

    for (int i = 0; i < pricesSize; ++i) {
        for (int j = k; j > 0; --j) {
            sell[j] = max(sell[j], hold[j] + prices[i]);
            hold[j] = max(hold[j], sell[j - 1] - prices[i]);
        }
    }
    int result = sell[k];
    free(sell);
    free(hold);
    return result;
}
```

Output:



<u>we use greedy approach if (k >= pricesSize / 2) otherwise just solve normally using DP</u>

Observation:

- my code correctly identifies that only one transaction is optimal, even though two are allowed **(k = 2).**
- The optimal transaction is to buy at 2 and sell at 4, yielding a profit of 2.
- The **greedy approach** simplifies the problem when **k** is large enough, avoiding the need for more complex dynamic programming in **test case 1.**

Problem: **122**. You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

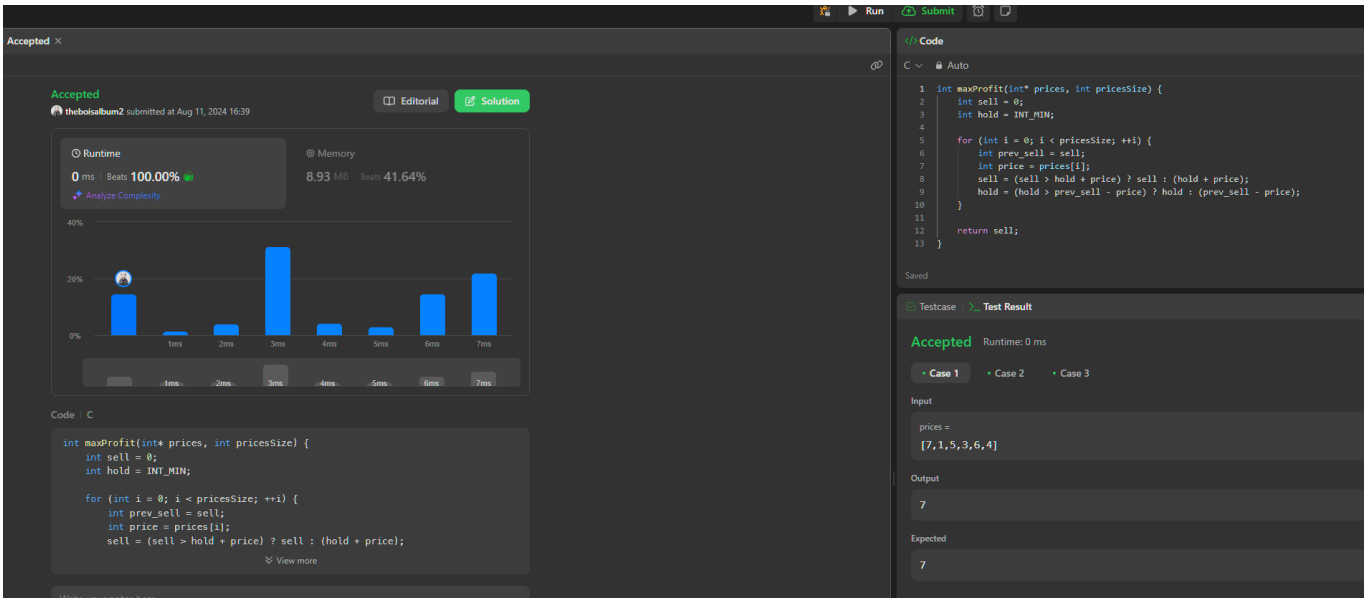
On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return the maximum profit you can achieve.

Program:

```c
int maxProfit(int* prices, int pricesSize) {
    int sell = 0;
    int hold = INT_MIN;

    for (int i = 0; i < pricesSize; ++i) {
        int prev_sell = sell;
        int price = prices[i];
        sell = (sell > hold + price) ? sell : (hold + price);
        hold = (hold > prev_sell - price) ? hold : (prev_sell - price);
    }

    return sell;
}
```

Output:



Observations:

- my code efficiently computes the maximum profit using a greedy approach, allowing multiple transactions.
- The optimal strategy is to buy at 1, sell at 5, buy again at 3, and sell at 6, yielding a total profit of 7.

Problem: **134.** There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

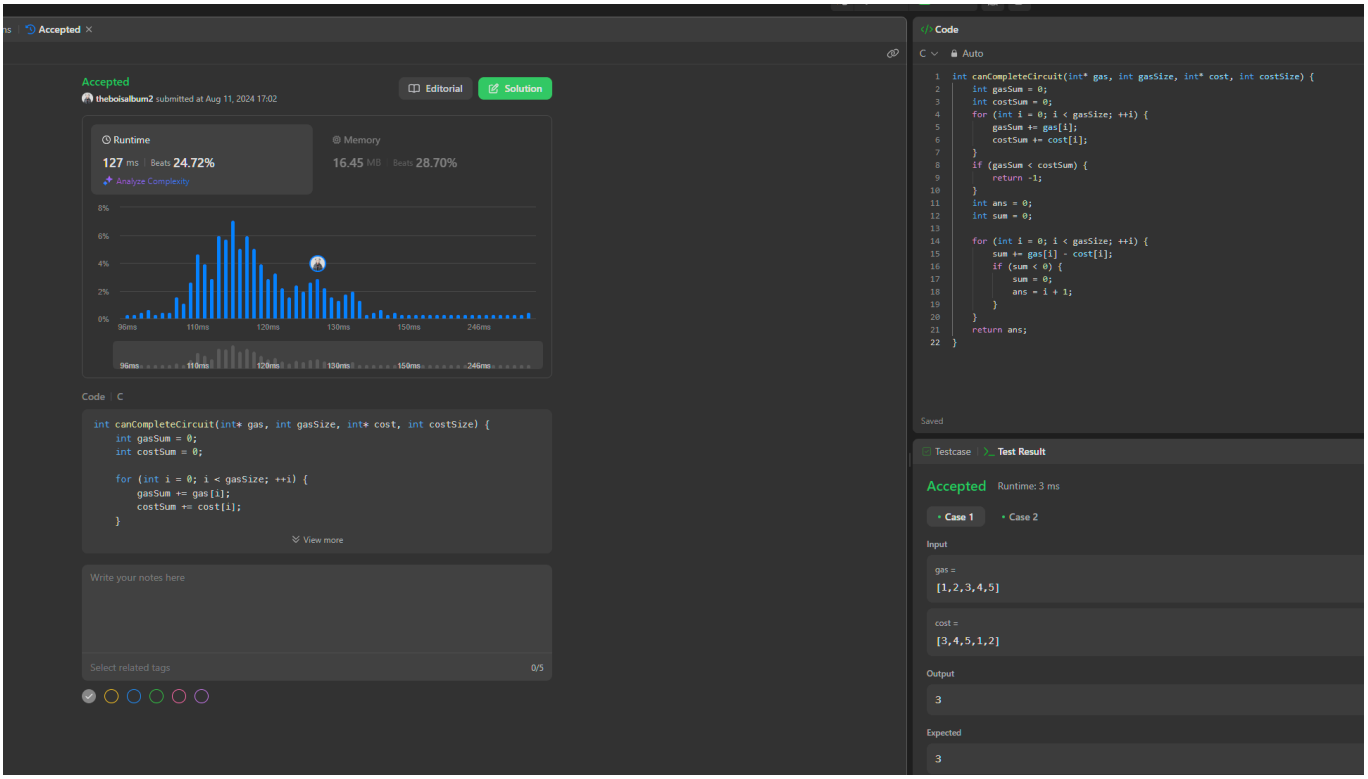
You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1. If there exists a solution, it is guaranteed to be unique

Program:

```
int canCompleteCircuit(int* gas, int gasSize, int* cost, int costSize) {
    int gasSum = 0;
    int costSum = 0;
    for (int i = 0; i < gasSize; ++i) {
        gasSum += gas[i];
        costSum += cost[i];
    }
    if (gasSum < costSum) {
        return -1;
    }
    int ans = 0;
    int sum = 0;

    for (int i = 0; i < gasSize; ++i) {
        sum += gas[i] - cost[i];
        if (sum < 0) {
            sum = 0;
            ans = i + 1;
        }
    }
    return ans;
}
```

Output:



Observations:

- my code correctly identifies that starting from station 3 (4th station) allows the car to complete the circular route with the given gas and cost arrays.
- The result is 3, meaning station 3 is the optimal starting point for completing the circuit