NAME – **Trisha Dilip Katole**
CLASS - A5
ROLL NO - **47**
BATCH – B3

PRACTICAL – 4

Aim: Implement maximum sum of subarray for the given scenario of resource allocation using

the divide and conquer approach.

Problem Statement: A project requires allocating resources to various tasks over a period of time. Each task requires

a certain amount of resources, and you want to maximize the overall efficiency of resource

usage. You're given an array resources where resources[i] represents the amount of resources

required for the I th task. Your goal is to find the contiguous subarray of tasks that maximizes

the total resources utilized without exceeding a given resource constraint.

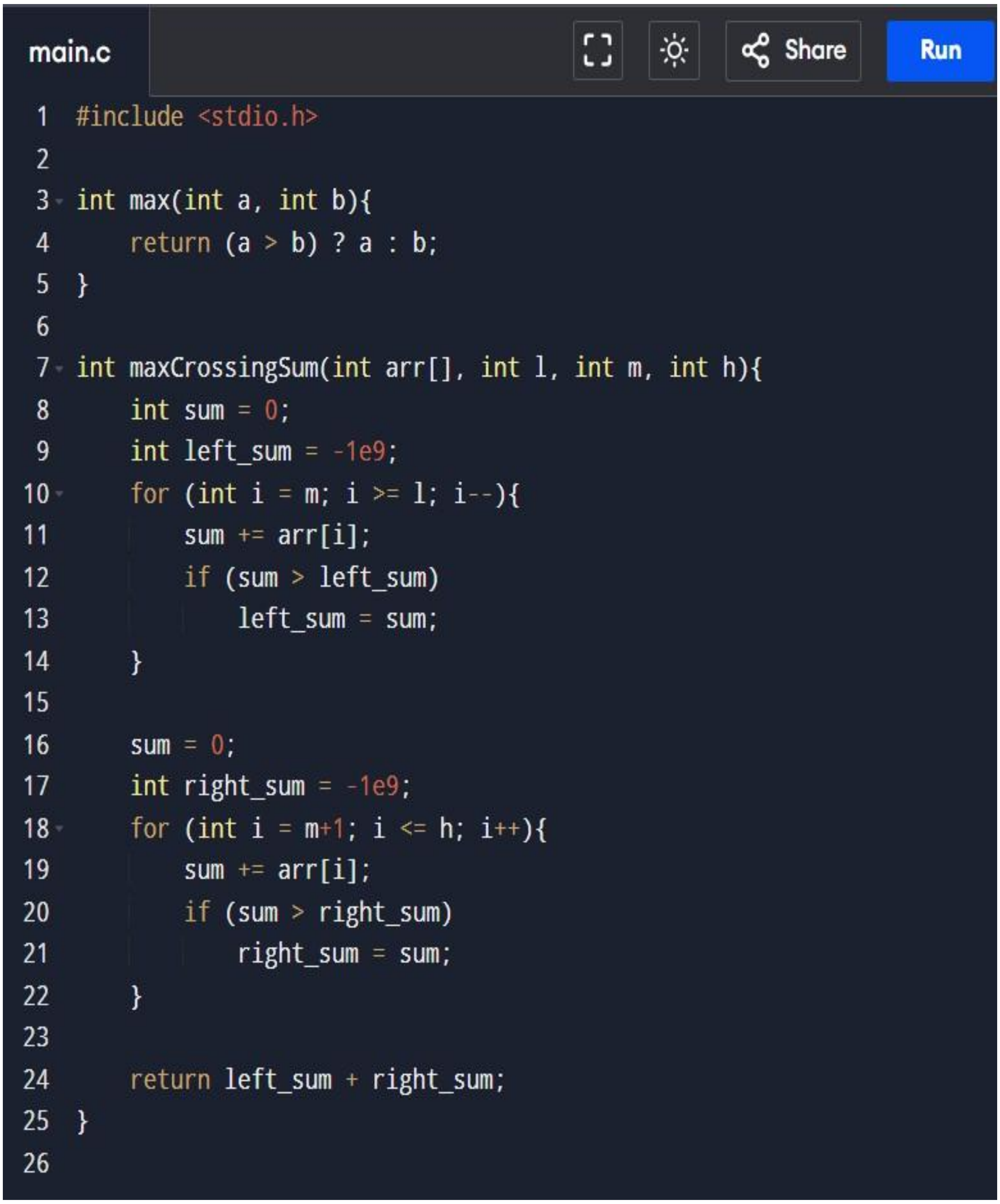
Handle cases where the total resources exceed the constraint by adjusting the subarray window

accordingly. Your implementation should handle various cases, including scenarios where

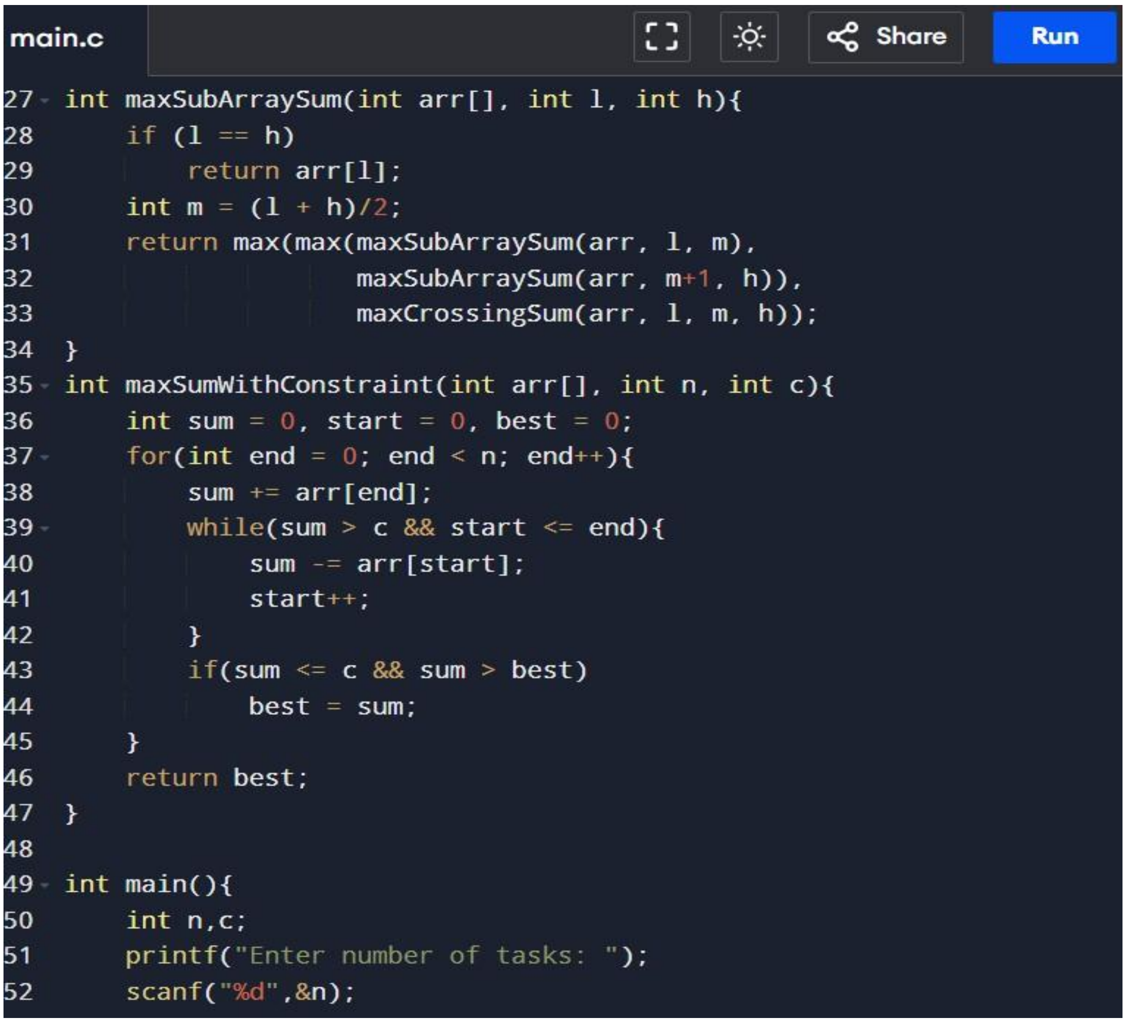
there's no feasible subarray given the constraint and scenarios where multiple subarrays yield

the same maximum resource utilization.

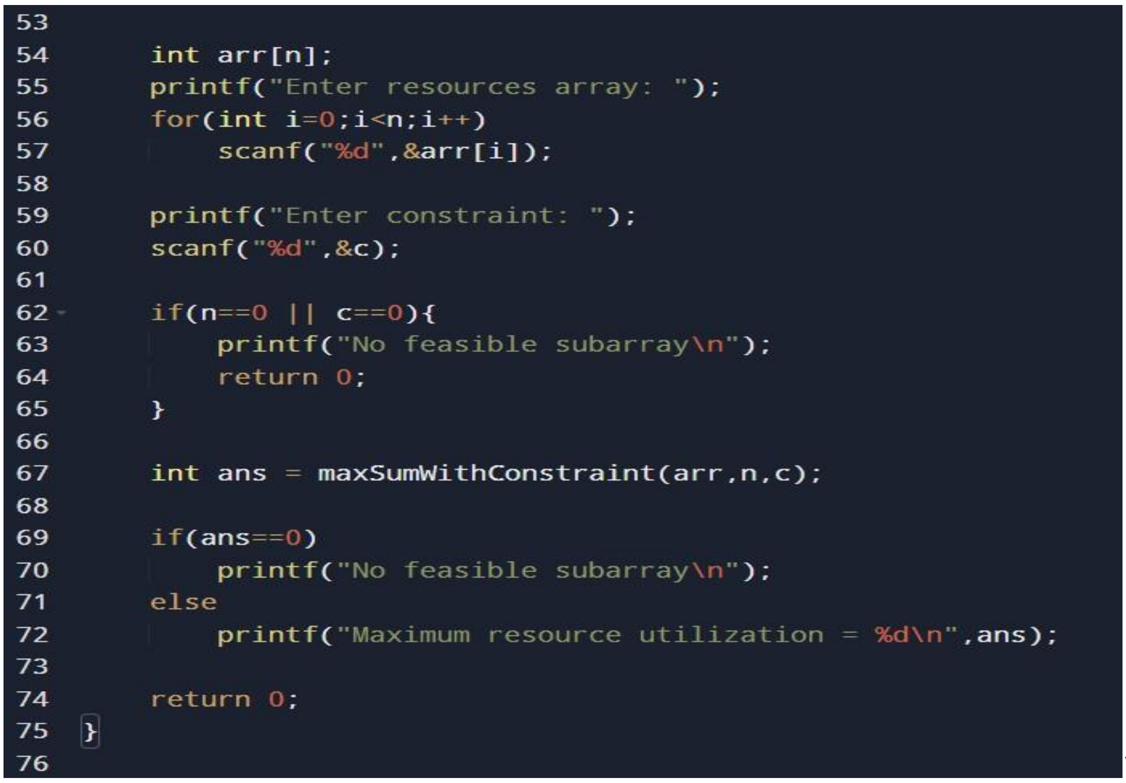
CODE –



```c
#include <stdio.h>

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

int maxCrossingSum(int arr[], int l, int m, int h){
    int sum = 0;
    int left_sum = -1e9;
    for (int i = m; i >= l; i--){
        sum += arr[i];
        if (sum > left_sum)
            left_sum = sum;
    }

    sum = 0;
    int right_sum = -1e9;
    for (int i = m+1; i <= h; i++){
        sum += arr[i];
        if (sum > right_sum)
            right_sum = sum;
    }

    return left_sum + right_sum;
}

```



```c
int maxSubArraySum(int arr[], int l, int h){
    if (l == h)
        return arr[l];
    int m = (l + h)/2;
    return max(max(maxSubArraySum(arr, l, m),
                   maxSubArraySum(arr, m+1, h)),
                   maxCrossingSum(arr, l, m, h));
}
int maxSumWithConstraint(int arr[], int n, int c){
    int sum = 0, start = 0, best = 0;
    for(int end = 0; end < n; end++){
        sum += arr[end];
        while(sum > c && start <= end){
            sum -= arr[start];
            start++;
        }
        if(sum <= c && sum > best)
            best = sum;
    }
    return best;
}

int main(){
    int n,c;
    printf("Enter number of tasks: ");
    scanf("%d",&n);
```



```c

    int arr[n];
    printf("Enter resources array: ");
    for(int i=0;i<n;i++)
        scanf("%d",&arr[i]);

    printf("Enter constraint: ");
    scanf("%d",&c);

    if(n==0 || c==0){
        printf("No feasible subarray\n");
        return 0;
    }

    int ans = maxSumWithConstraint(arr,n,c);

    if(ans==0)
        printf("No feasible subarray\n");
    else
        printf("Maximum resource utilization = %d\n",ans);

    return 0;
}

```

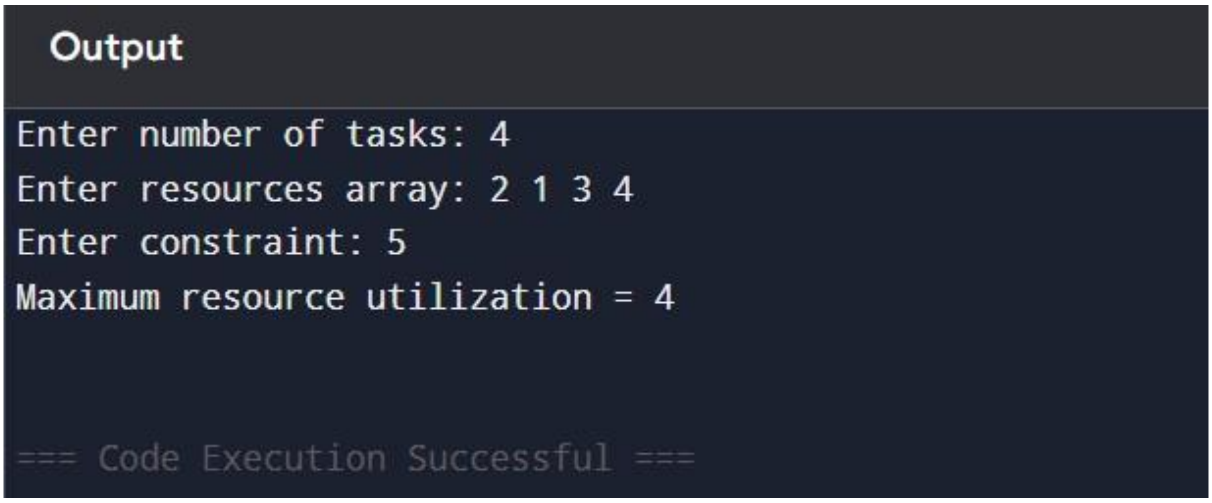
\

OUTPUT -

TASK-1:

1. Basic small array

- resources = [2, 1, 3, 4], constraint = 5
  - Best subarray: [2, 1] or [1, 3] → sum = 4
  - Checks simple working.

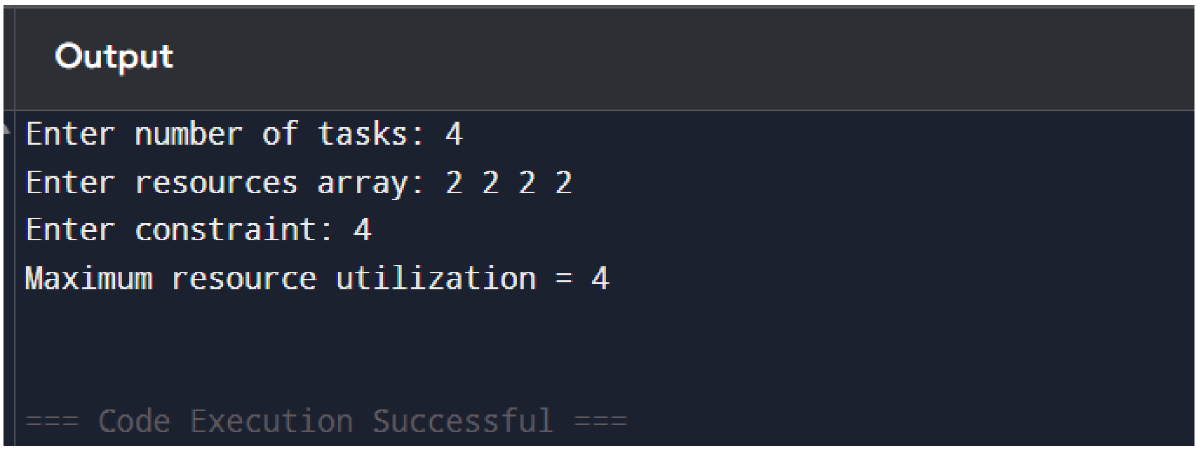


Output

```
Enter number of tasks: 4
Enter resources array: 2 1 3 4
Enter constraint: 5
Maximum resource utilization = 4


=== Code Execution Successful ===
```

2. Exact match to constraint

- resources = [2, 2, 2, 2], constraint = 4
  - Best subarray: [2, 2] → sum = 4
  - Tests exact utilization.

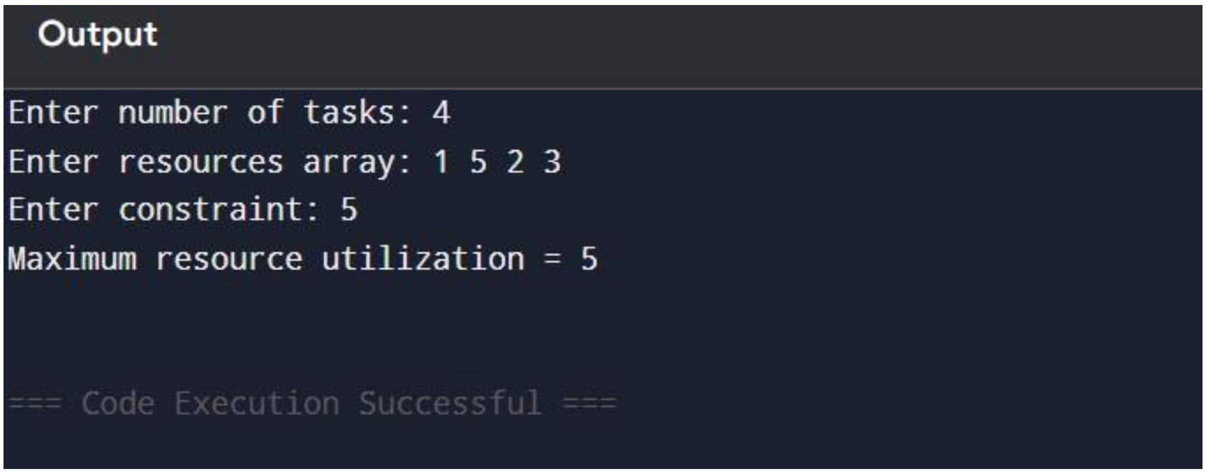


Output

```
Enter number of tasks: 4
Enter resources array: 2 2 2 2
Enter constraint: 4
Maximum resource utilization = 4


=== Code Execution Successful ===
```

3. Single element equals constraint

- resources = [1, 5, 2, 3], constraint = 5
  - Best subarray: [5] → sum = 5
  - Tests one-element solution.

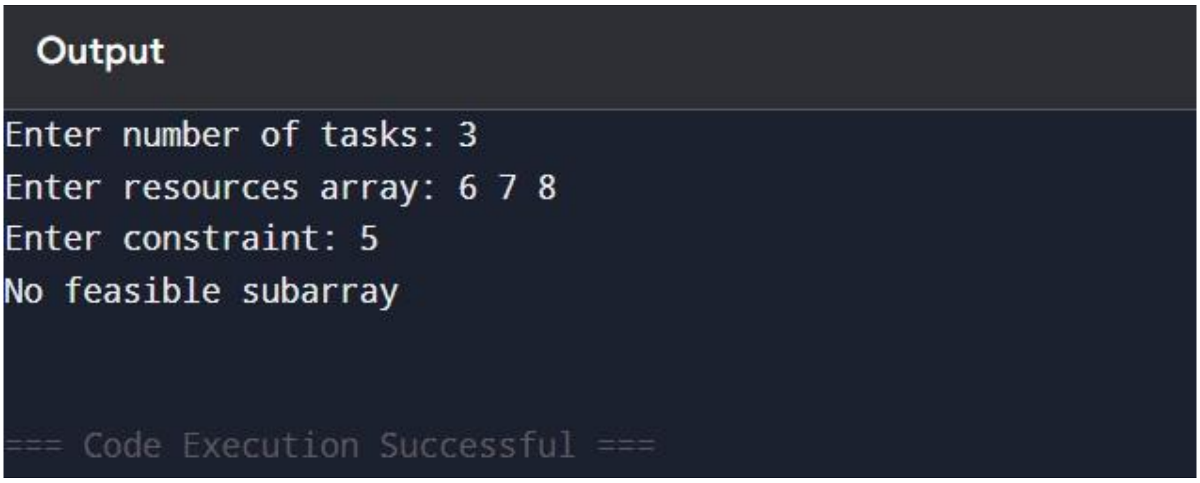


Output

```
Enter number of tasks: 4
Enter resources array: 1 5 2 3
Enter constraint: 5
Maximum resource utilization = 5


=== Code Execution Successful ===
```

4. All elements smaller but no combination fits

- resources = [6, 7, 8], constraint = 5
  - No feasible subarray.
  - Tests "no solution" case.

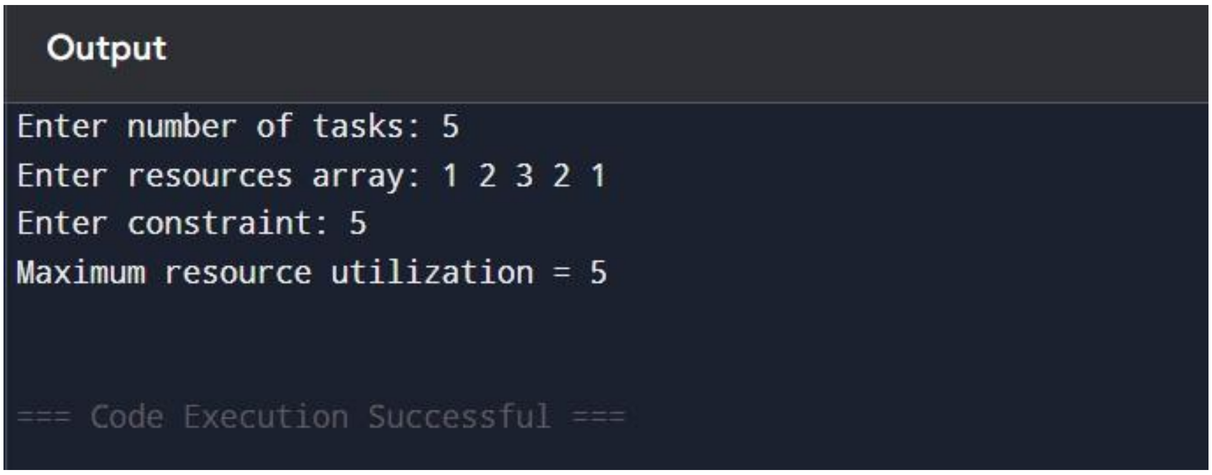


Output

```
Enter number of tasks: 3
Enter resources array: 6 7 8
Enter constraint: 5
No feasible subarray


=== Code Execution Successful ===
```

5. Multiple optimal subarrays

- resources = [1, 2, 3, 2, 1], constraint = 5
  - Best subarrays: [2, 3] and [3, 2] → sum = 5
  - Tests tie-breaking (should return either valid subarray).

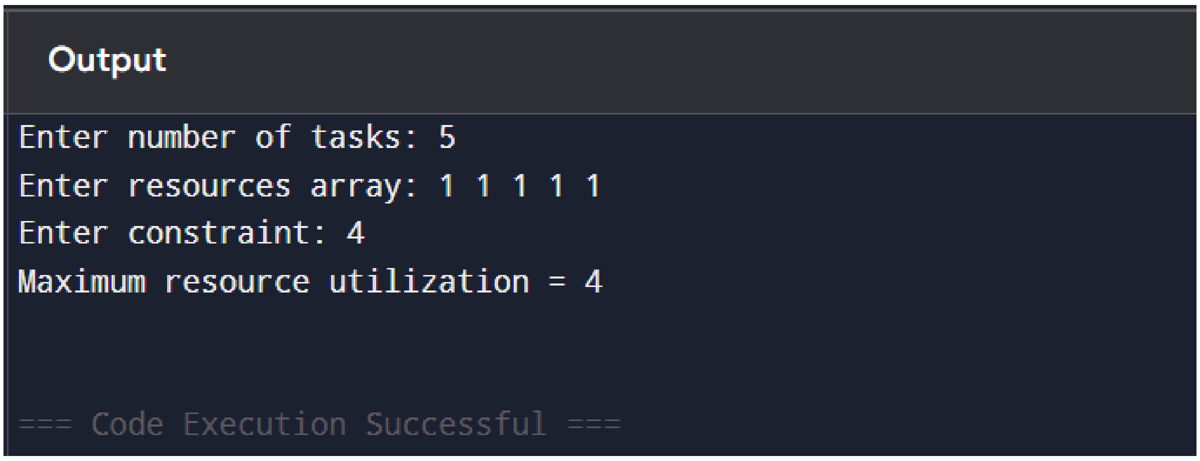


Output

```
Enter number of tasks: 5
Enter resources array: 1 2 3 2 1
Enter constraint: 5
Maximum resource utilization = 5


=== Code Execution Successful ===
```

6. Large window valid

- resources = [1, 1, 1, 1, 1], constraint = 4
  - Best subarray: [1, 1, 1, 1] → sum = 4
  - Ensures long window works.

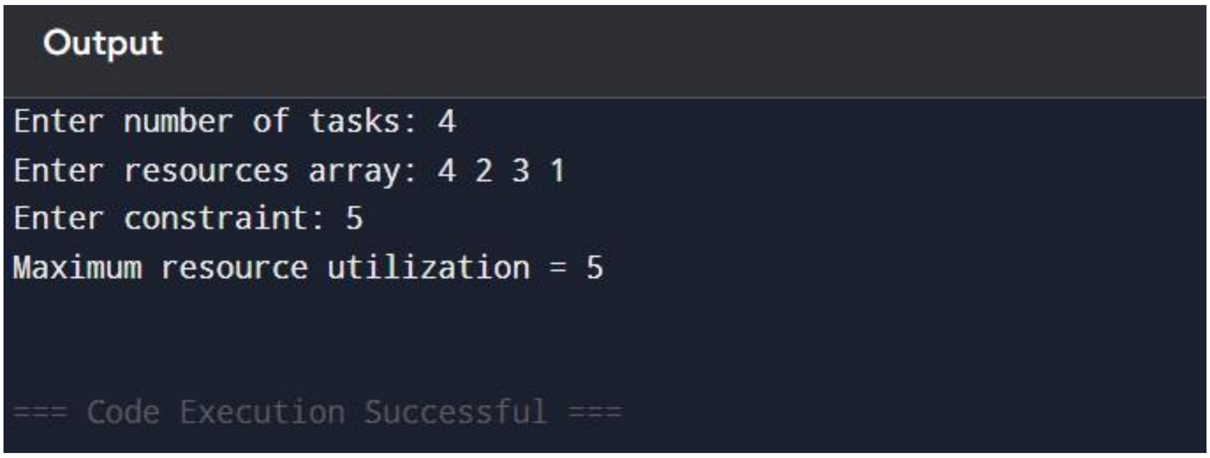


Output

```
Enter number of tasks: 5
Enter resources array: 1 1 1 1 1
Enter constraint: 4
Maximum resource utilization = 4


=== Code Execution Successful ===
```

7. Sliding window shrink needed

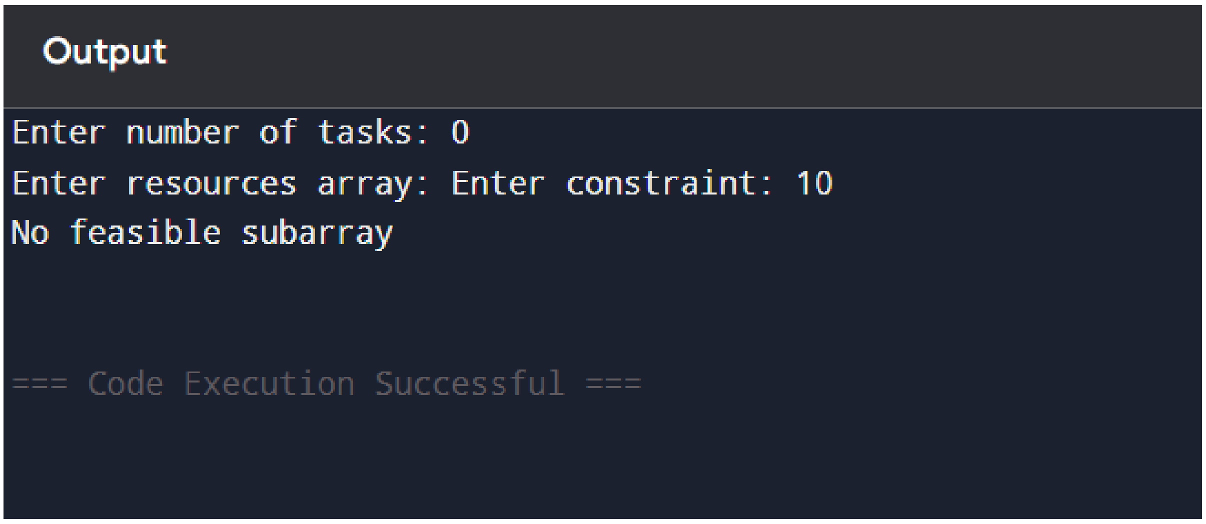
- resources = [4, 2, 3, 1], constraint = 5
    - Start [4,2] = 6 (too big) → shrink to [2,3] = 5.
    - Tests dynamic window adjustment.



```
Output
Enter number of tasks: 4
Enter resources array: 4 2 3 1
Enter constraint: 5
Maximum resource utilization = 5


=== Code Execution Successful ===
```

8. Empty array

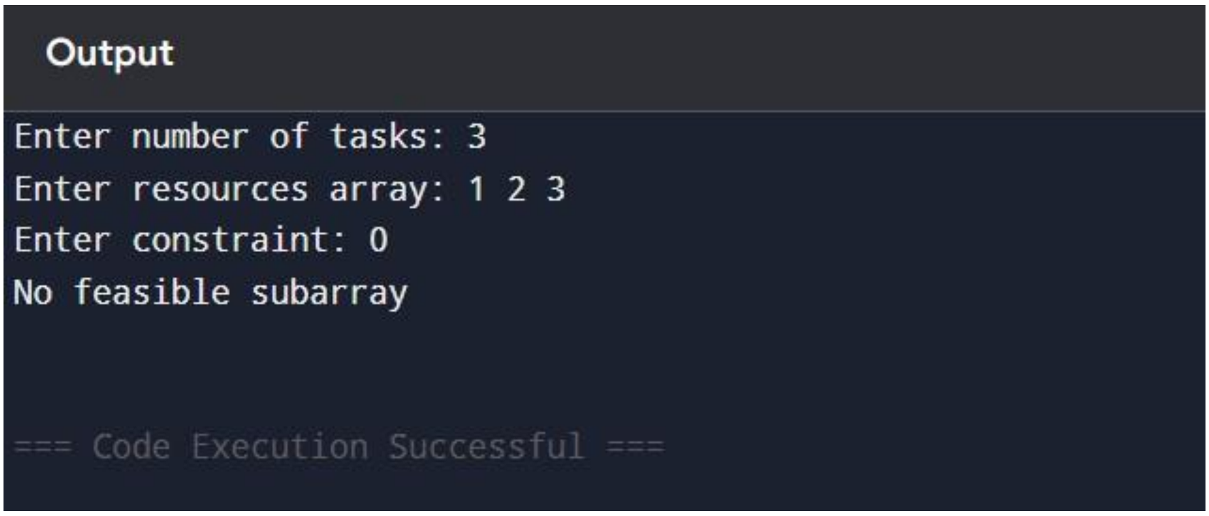
- resources = [], constraint = 10
    - Output: no subarray.
    - Edge case: empty input.



```
Output
Enter number of tasks: 0
Enter resources array: Enter constraint: 10
No feasible subarray


=== Code Execution Successful ===
```

9. Constraint = 0

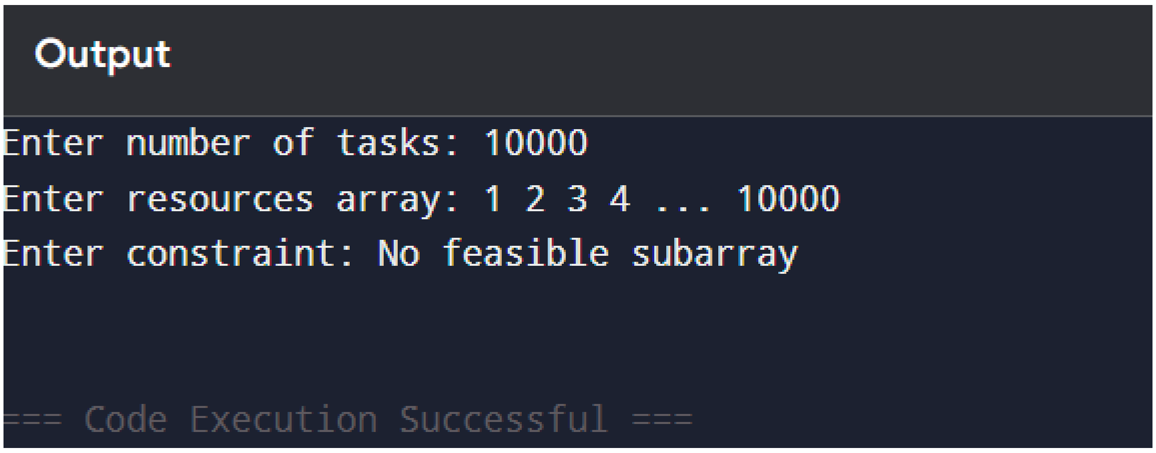
- resources = [1, 2, 3], constraint = 0
  - No subarray possible.
  - Edge case: zero constraint.


Output

```
Enter number of tasks: 3
Enter resources array: 1 2 3
Enter constraint: 0
No feasible subarray


=== Code Execution Successful ===
```

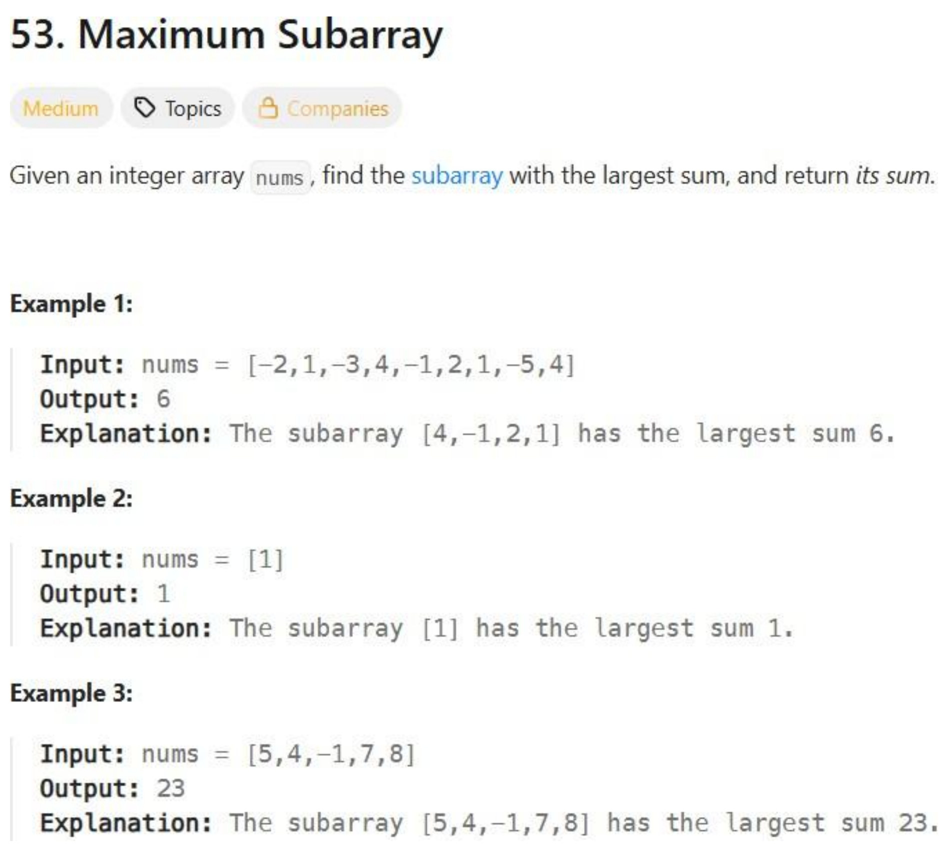
10. Very large input (stress test)

- resources = [1, 2, 3, ..., 100000], constraint = 10^9
  - Valid subarray near full array.
  - Performance test.


Output

```
Enter number of tasks: 10000
Enter resources array: 1 2 3 4 ... 10000
Enter constraint: No feasible subarray


=== Code Execution Successful ===
```

LEETCODE –



53. Maximum Subarray

Medium Topics Companies

Given an integer array `nums`, find the subarray with the largest sum, and return *its sum*.

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.
```

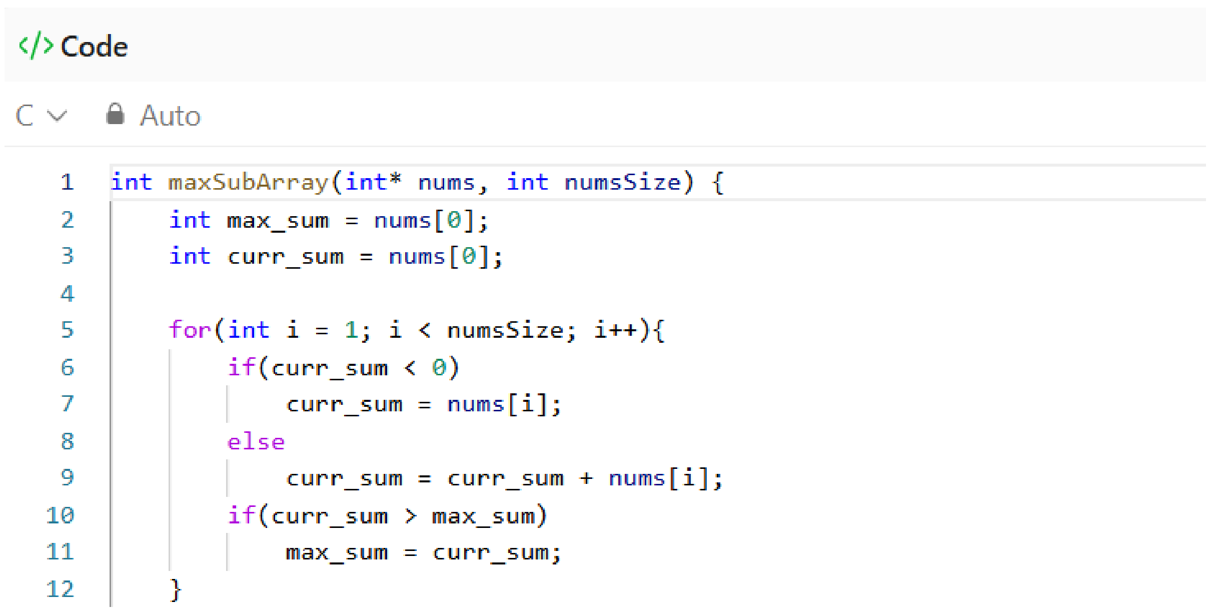
Example 2:

```
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
```

Example 3:

```
Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.
```

CODE –



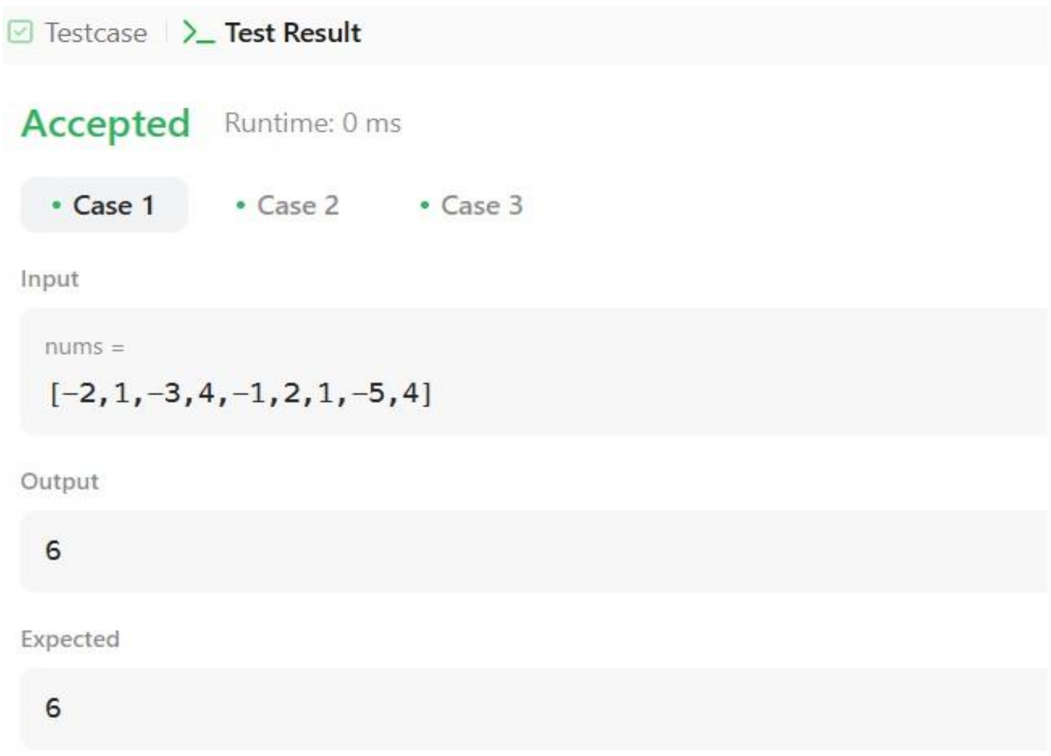
Code

C Auto

```c
int maxSubArray(int* nums, int numsSize) {
    int max_sum = nums[0];
    int curr_sum = nums[0];

    for(int i = 1; i < numsSize; i++){
        if(curr_sum < 0)
            curr_sum = nums[i];
        else
            curr_sum = curr_sum + nums[i];
        if(curr_sum > max_sum)
            max_sum = curr_sum;
    }
```

OUTPUT –

CASE 1


Testcase | >_ Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
nums =
[-2,1,-3,4,-1,2,1,-5,4]
```

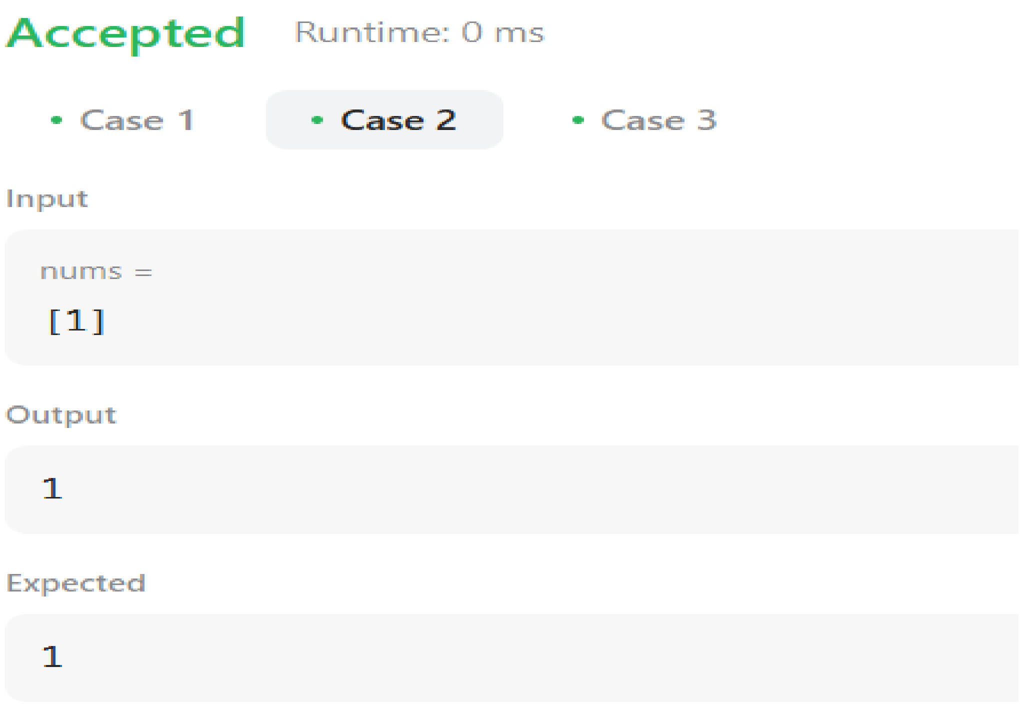
Output

```
6
```

Expected

```
6
```

CASE 2 -


Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
nums =
[1]
```

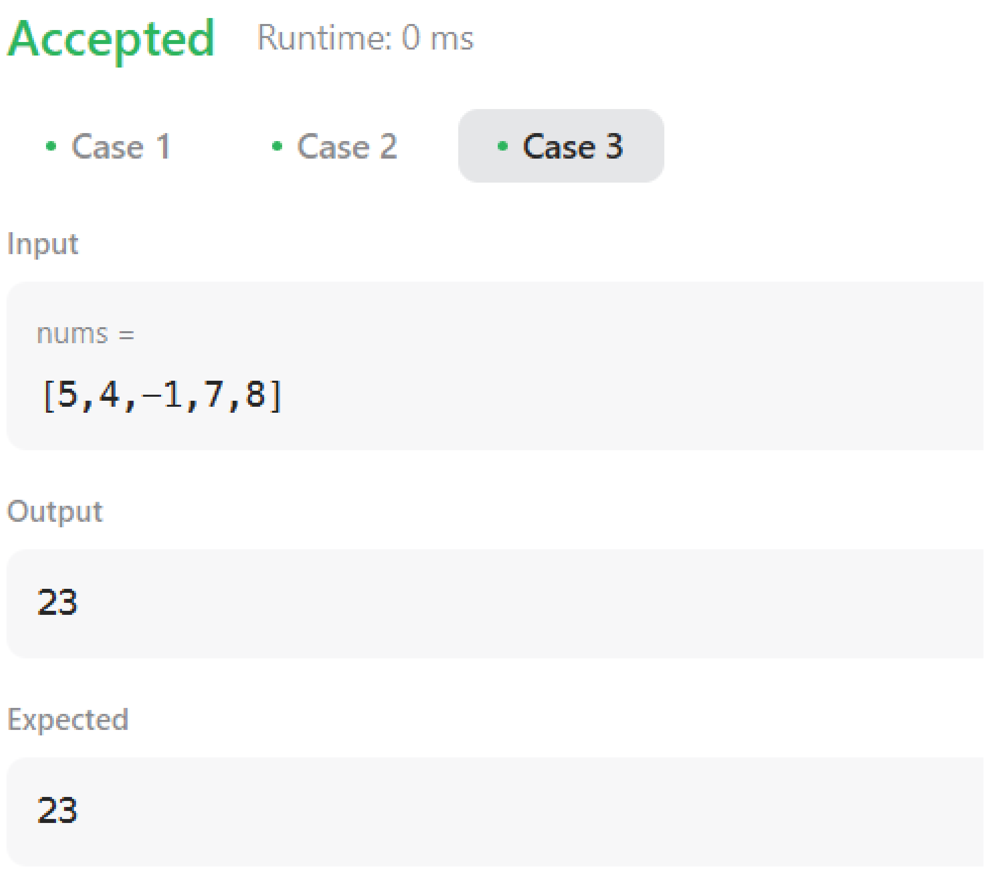
Output

```
1
```

Expected

```
1
```

CASE 3 –


Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

nums =

```
[5,4,-1,7,8]
```

Output

```
23
```

Expected

```
23
```