

# PRACTICAL NO. : 04

NAME : RANVEER M. BHORTEKAR

SECTION/BATCH : A4/B1

ROLL NO. : 06

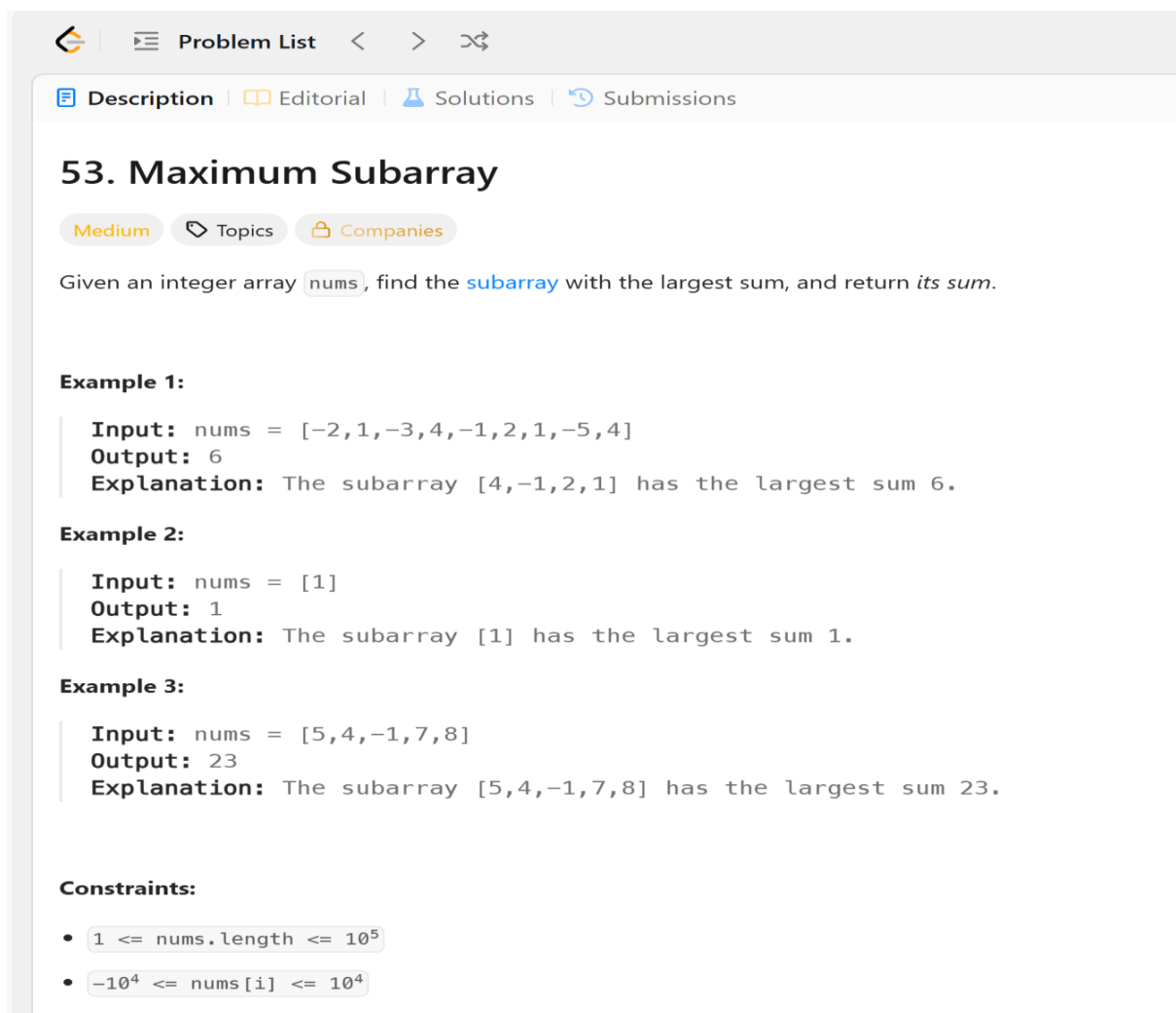
SUBJECT : DAA

LEETCODE ID : [Bhortekar](#) [Coder - LeetCode Profile](#)

**AIM** : Implement maximum sum of subarray for the given scenario of resource allocation using the divide and conquer approach.

## LEETCODE QUESTION ID 53. MAXIMUM SUBARRAY :

### PROBLEM STATEMENT :



The screenshot shows the LeetCode interface for the problem '53. Maximum Subarray'. At the top, there's a navigation bar with 'Problem List', '<', '>', and a refresh icon. Below it, tabs for 'Description', 'Editorial', 'Solutions', and 'Submissions' are visible. The problem title '53. Maximum Subarray' is prominently displayed, followed by difficulty tags 'Medium', 'Topics', and 'Companies'. The problem description states: 'Given an integer array `nums`, find the **subarray** with the largest sum, and return *its sum*.' Three examples are provided: Example 1 with input `nums = [-2,1,-3,4,-1,2,1,-5,4]` and output 6; Example 2 with input `nums = [1]` and output 1; and Example 3 with input `nums = [5,4,-1,7,8]` and output 23. Each example includes an explanation of the subarray that yields the maximum sum. At the bottom, the constraints are listed: `1 <= nums.length <= 105` and `-104 <= nums[i] <= 104`.

Problem List < > ↺

Description | Editorial | Solutions | Submissions

### 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:**

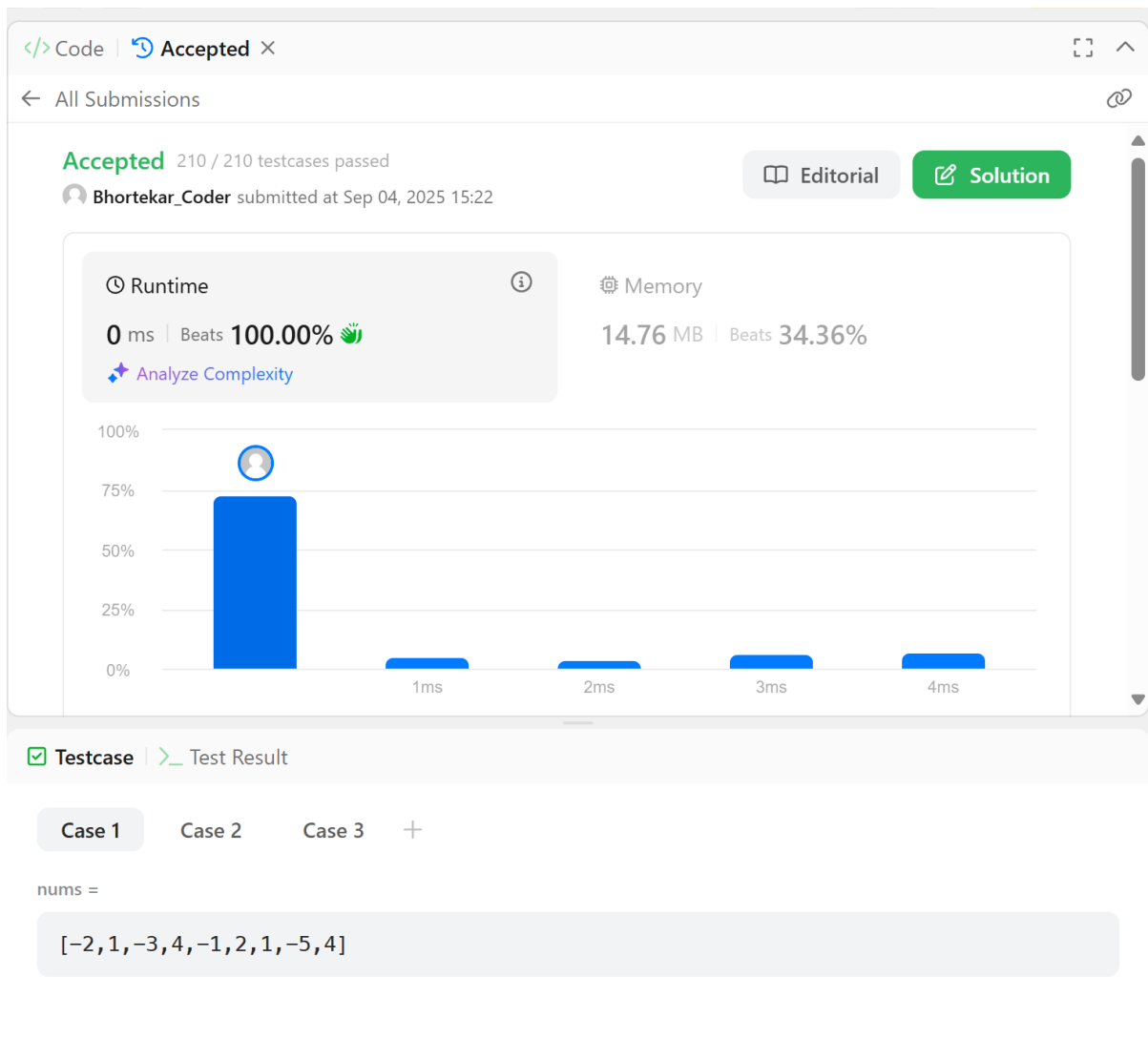
**Input:** `nums = [5,4,-1,7,8]`  
**Output:** 23  
**Explanation:** The subarray `[5,4,-1,7,8]` has the largest sum 23.

**Constraints:**

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`

## SUBMISSION :

### CODE



## CODE IN TEXT FORMAT :

```
int maxSubArray(int* nums,int numsSize){
    int maxSum=nums[0];
    int maxSumSoFar=0;

    for(int i=0;i<numsSize;i++){
        maxSumSoFar = maxSumSoFar + nums[i];
        if(maxSumSoFar>maxSum){
            maxSum=maxSumSoFar;
        }
    }
}
```

```

    }

    if(maxSumSoFar<0){
        maxSumSoFar=0;
    }

}

return maxSum;

}

```

## SCREENSHOT OF CODE :

The screenshot displays a LeetCode post-solution page for the "Maximum Subarray" problem. The browser address bar shows the URL: <https://leetcode.com/problems/maximum-subarray/post-solution/?submissionId=1759214829>. The page layout includes a navigation bar with links to Explore, Problems, Contest, Discuss, Interview, and Store. A "Premium" badge is visible in the top right corner.

The main content area is divided into two panels. The left panel contains the C++ code submission, which is a solution for the "Maximum Subarray" problem. The code is as follows:

```

# Intuition
<!-- Describe your first thoughts on how to solve this problem. -->

# Approach
<!-- Describe your approach to solving the problem. -->

# Complexity
- Time complexity:
<!-- Add your time complexity here, e.g. $O(n)$ -->

- Space complexity:
<!-- Add your space complexity here, e.g. $O(n)$ -->

# Code
```c []
int maxSubArray(int* nums, int numsSize){
    int maxSum=nums[0];
    int maxSumSoFar=0;

    for(int i=0; i<numsSize; i++){
        maxSumSoFar = maxSumSoFar + nums[i];
        if(maxSumSoFar>maxSum){
            maxSum=maxSumSoFar;
        }
        if(maxSumSoFar<0){
            maxSumSoFar=0;
        }
    }
    return maxSum;
}
```

```

The right panel shows the "Approach" and "Complexity" sections. The "Approach" section is currently empty. The "Complexity" section lists the time and space complexities:

- Time complexity:
- Space complexity:

Below the complexity section is the "Code" section, which displays the C++ code in a syntax-highlighted format:

```

C

int maxSubArray(int* nums, int numsSize){
    int maxSum=nums[0];
    int maxSumSoFar=0;

    for(int i=0; i<numsSize; i++){
        maxSumSoFar = maxSumSoFar + nums[i];
        if(maxSumSoFar>maxSum){
            maxSum=maxSumSoFar;
        }
        if(maxSumSoFar<0){
            maxSumSoFar=0;
        }
    }
    return maxSum;
}

```

At the bottom of the page, there is a footer that reads "Generated from the chosen submission".

### **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 of resources where resources[i] represents the amount of resources required for the ith 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 IN TEXT FORMAT :**

```
#include <stdio.h>

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

int maxCrossingSum(int arr[], int left, int mid, int right, int constraint) {
    int sum = 0;
    int left_sum = 0;
    for (int i = mid; i >= left; i--) {
        sum += arr[i];
        if (sum <= constraint) {
            left_sum = max(left_sum, sum);
        } else {
            break;
        }
    }
    sum = 0;
    int right_sum = 0;
    for (int i = mid + 1; i <= right; i++) {
```

```
    sum += arr[i];  
    if (sum <= constraint) {  
        right_sum = max(right_sum, sum);  
    } else {  
        break;  
    }  
}
```

```
int total = left_sum + right_sum;  
if (total <= constraint) {  
    return total;  
} else {  
    return max(left_sum, right_sum);  
}  
}
```

```
int maxSubArraySumUtil(int arr[], int left, int right, int constraint) {  
    if (left == right) {  
        return (arr[left] <= constraint) ? arr[left] : 0;  
    }  
}
```

```
int mid = (left + right) / 2;  
int left_sum = maxSubArraySumUtil(arr, left, mid, constraint);  
int right_sum = maxSubArraySumUtil(arr, mid + 1, right, constraint);  
int cross_sum = maxCrossingSum(arr, left, mid, right, constraint);  
  
return max(max(left_sum, right_sum), cross_sum);  
}
```

```
int maxSubArraySum(int arr[], int n, int constraint) {  
    if (n == 0) return 0;  
    return maxSubArraySumUtil(arr, 0, n - 1, constraint);  
}
```

```
int main() {  
    int arr1[] = {2, 1, 3, 4};  
    int constraint1 = 5;  
    int n1 = sizeof(arr1) / sizeof(arr1[0]);  
    printf("Test 1: Max sum = %d\n", maxSubArraySum(arr1, n1, constraint1));  
  
    int arr2[] = {2, 2, 2, 2};  
    int constraint2 = 4;  
    int n2 = sizeof(arr2) / sizeof(arr2[0]);  
    printf("Test 2: Max sum = %d\n", maxSubArraySum(arr2, n2, constraint2));  
  
    int arr3[] = {1, 5, 2, 3};  
    int constraint3 = 5;  
    int n3 = sizeof(arr3) / sizeof(arr3[0]);  
    printf("Test 3: Max sum = %d\n", maxSubArraySum(arr3, n3, constraint3));  
  
    int arr4[] = {6, 7, 8};  
    int constraint4 = 5;  
    int n4 = sizeof(arr4) / sizeof(arr4[0]);  
    printf("Test 4: Max sum = %d\n", maxSubArraySum(arr4, n4, constraint4));  
  
    int arr5[] = {1, 2, 3, 2, 1};
```

```
int constraint5 = 5;

int n5 = sizeof(arr5) / sizeof(arr5[0]);

printf("Test 5: Max sum = %d\n", maxSubArraySum(arr5, n5, constraint5));
```

```
int arr6[] = {1, 1, 1, 1, 1};

int constraint6 = 4;

int n6 = sizeof(arr6) / sizeof(arr6[0]);

printf("Test 6: Max sum = %d\n", maxSubArraySum(arr6, n6, constraint6));
```

```
int arr7[] = {4, 2, 3, 1};

int constraint7 = 5;

int n7 = sizeof(arr7) / sizeof(arr7[0]);

printf("Test 7: Max sum = %d\n", maxSubArraySum(arr7, n7, constraint7));
```

```
int arr8[] = {};

int constraint8 = 10;

int n8 = sizeof(arr8) / sizeof(arr8[0]);

printf("Test 8: Max sum = %d\n", maxSubArraySum(arr8, n8, constraint8));
```

```
int arr9[] = {1, 2, 3};

int constraint9 = 0;

int n9 = sizeof(arr9) / sizeof(arr9[0]);

printf("Test 9: Max sum = %d\n", maxSubArraySum(arr9, n9, constraint9));
```

```
int arr10[100000];

for (int i = 0; i < 100000; i++) arr10[i] = i + 1;

int constraint10 = 1000000000;

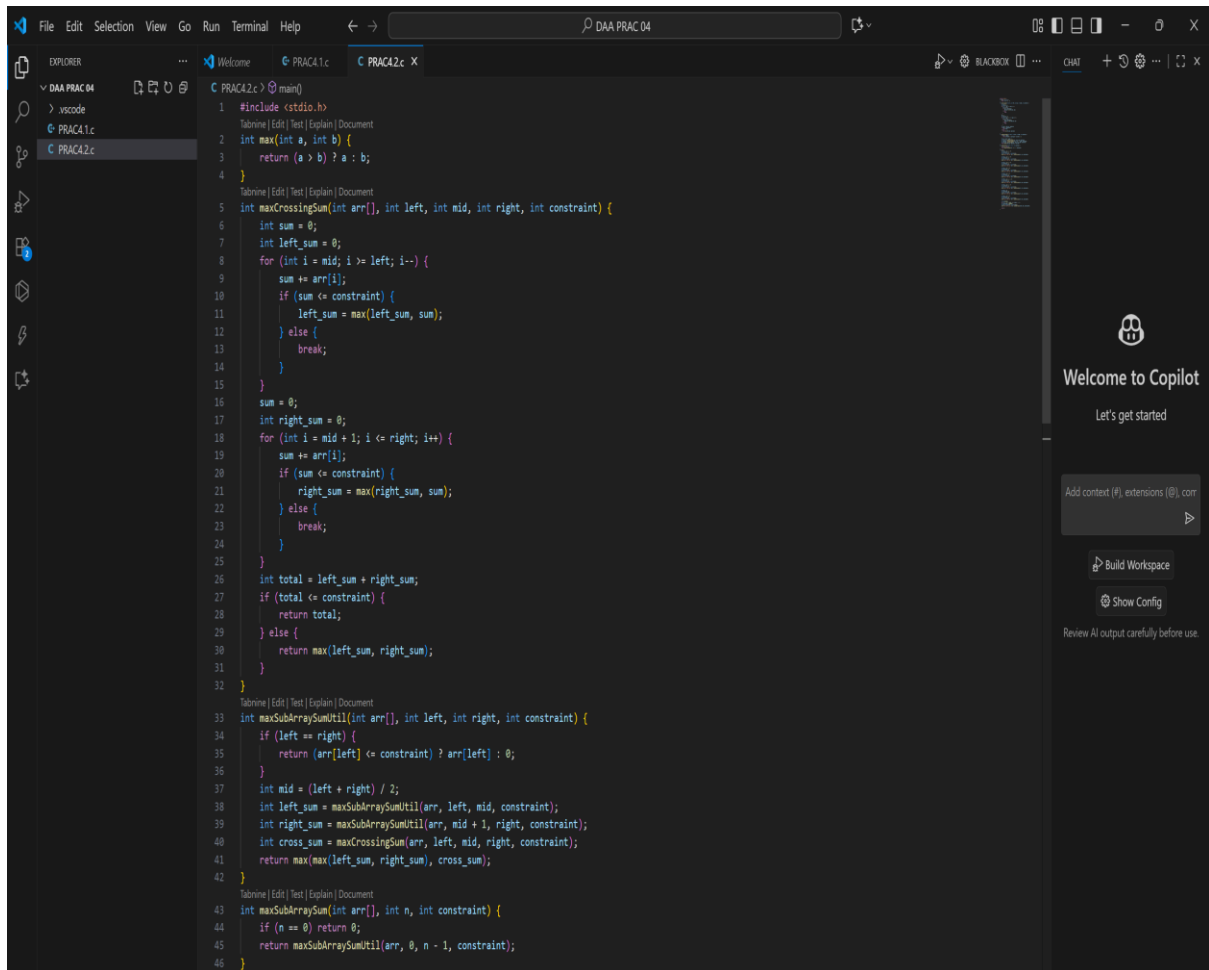
int n10 = sizeof(arr10) / sizeof(arr10[0]);
```

```
printf("Test 10: Max sum = %d\n", maxSubArraySum(arr10, n10, constraint10));
```

```
return 0;
```

```
}
```

## SCREENSHOT OF CODE :



The screenshot shows a Visual Studio Code editor with a C file named PRAC42.c. The code implements three functions: max, maxCrossingSum, and maxSubArraySumUtil, along with a main function. The max function returns the maximum of two integers. maxCrossingSum calculates the maximum sum of a subarray crossing the middle element, ensuring the sum does not exceed a given constraint. maxSubArraySumUtil uses a recursive divide-and-conquer approach to find the maximum subarray sum within a specified range, also respecting the constraint. The main function tests these functions with an array of 10 elements.

```
1 #include <stdio.h>
2 int max(int a, int b) {
3     return (a > b) ? a : b;
4 }
5 int maxCrossingSum(int arr[], int left, int mid, int right, int constraint) {
6     int sum = 0;
7     int left_sum = 0;
8     for (int i = mid; i >= left; i--) {
9         sum += arr[i];
10        if (sum <= constraint) {
11            left_sum = max(left_sum, sum);
12        } else {
13            break;
14        }
15    }
16    sum = 0;
17    int right_sum = 0;
18    for (int i = mid + 1; i <= right; i++) {
19        sum += arr[i];
20        if (sum <= constraint) {
21            right_sum = max(right_sum, sum);
22        } else {
23            break;
24        }
25    }
26    int total = left_sum + right_sum;
27    if (total <= constraint) {
28        return total;
29    } else {
30        return max(left_sum, right_sum);
31    }
32 }
33 int maxSubArraySumUtil(int arr[], int left, int right, int constraint) {
34     if (left == right) {
35         return (arr[left] <= constraint) ? arr[left] : 0;
36     }
37     int mid = (left + right) / 2;
38     int left_sum = maxSubArraySumUtil(arr, left, mid, constraint);
39     int right_sum = maxSubArraySumUtil(arr, mid + 1, right, constraint);
40     int cross_sum = maxCrossingSum(arr, left, mid, right, constraint);
41     return max(max(left_sum, right_sum), cross_sum);
42 }
43 int maxSubArraySum(int arr[], int n, int constraint) {
44     if (n == 0) return 0;
45     return maxSubArraySumUtil(arr, 0, n - 1, constraint);
46 }
```



