Ramdeobaba University, Nagpur
Department of Computer Science and Engineering
Session: 2025-26

Design and Analysis of Algorithms Lab III Semester

PRACTICAL NO. 4

Name : Ananya Biyani
Class : A4 B1
Roll : 02

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.

1. Basic small array

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

2. Exact match to constraint

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

3. Single element equals constraint

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

4. All elements smaller but no combination fits

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

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).

6. Large window valid

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

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.

8. Empty array

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

9. Constraint = 0

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

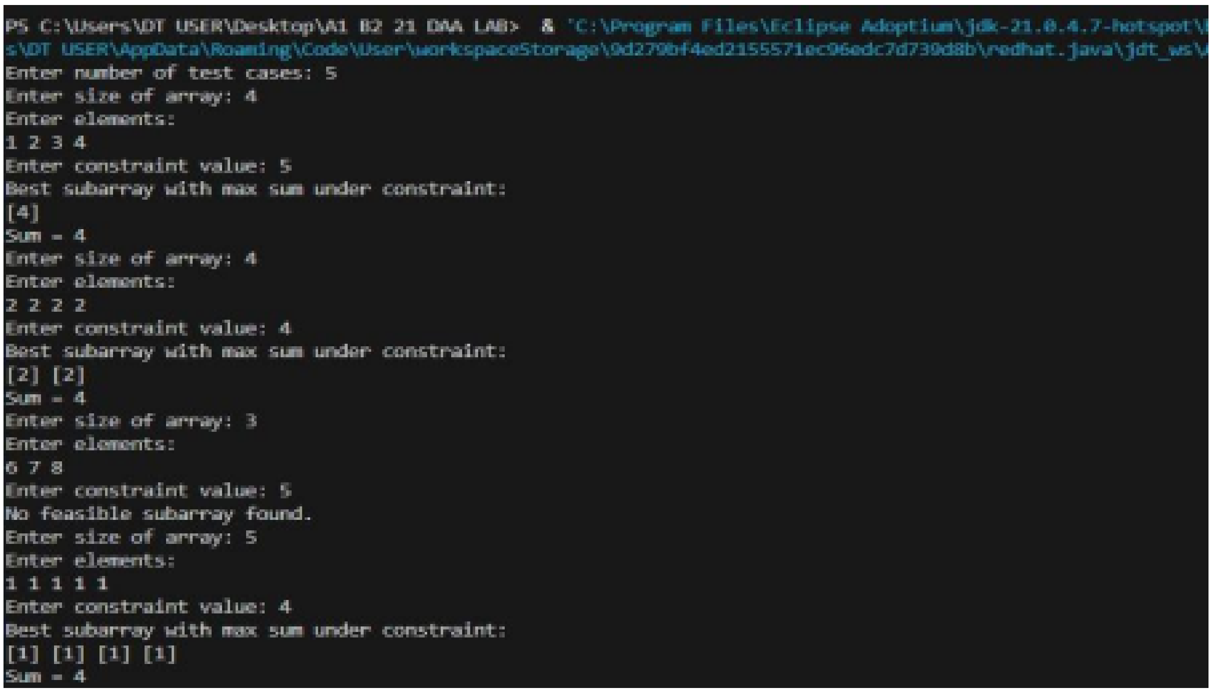
10. Very large input (stress test)

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

CODE:

```
import java.util.*;


public class Practical_04_AdarshJha {


    static class Result {

        int sum;

        int start;

        int end;


        Result(int sum, int start, int end) {

            this.sum = sum;

            this.start = start;

            this.end = end;

        }

    }


    public static Result findMaxSum(int[] arr, int s, int e, int
constraint) {

        if (s == e) {

            if (arr[s] <= constraint){
                return new Result(arr[s], s, s);

            }

            else{

                return new Result(Integer.MIN_VALUE, -1, -1);

            }
        }

        int mid = s + (e - s) / 2;
        Result left = findMaxSum(arr, s, mid, constraint);
        Result right = findMaxSum(arr, mid + 1, e, constraint);
        Result cross = crossSum(arr, s, mid, e, constraint);
        Result maxResult = left;
        if (right.sum > maxResult.sum){
            maxResult = right;
        }
        if (cross.sum > maxResult.sum){
            maxResult = cross;
        }

        return maxResult;
    }
    public static Result crossSum(int[] arr, int s, int mid, int e, int
constraint) {
        int leftSum = Integer.MIN_VALUE;
        int sum = 0;
        int maxLeft = mid;
        for (int i = mid; i >= s; i--) {
            sum += arr[i];
           if (sum <= constraint && sum > leftSum) {
                leftSum = sum;
                maxLeft = i;
            }
        }
        int rightSum = Integer.MIN_VALUE;
        sum = 0;
        int maxRight = mid + 1;
        for (int i = mid + 1; i <= e; i++) {
            sum += arr[i];
            if (leftSum != Integer.MIN_VALUE && leftSum + sum <=
constraint && leftSum + sum > rightSum) {
                rightSum = leftSum + sum;
                maxRight = i;
            }
        }
        if (leftSum != Integer.MIN_VALUE && rightSum !=
Integer.MIN_VALUE) {
            return new Result(rightSum, maxLeft, maxRight);
        }
        if (leftSum != Integer.MIN_VALUE && leftSum > rightSum) {
            return new Result(leftSum, maxLeft, mid);
        }

        return new Result(Integer.MIN_VALUE, -1, -1);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of test cases: ");
        int t = sc.nextInt();

        while (t-- > 0) {
            System.out.print("Enter size of array: ");
            int n = sc.nextInt();

            int[] arr = new int[n];
            System.out.println("Enter elements:");
            for (int i = 0; i < n; i++) {
                arr[i] = sc.nextInt();
            }

            System.out.print("Enter constraint value: ");
            int constraint = sc.nextInt();

            Result result = findMaxSum(arr, 0, n - 1, constraint);

            if (result.sum == Integer.MIN_VALUE) {
                System.out.println("No feasible subarray found.");
            }
            else {
                System.out.println("Best subarray with max sum
under constraint:");
                for (int i = result.start; i <= result.end; i++) {
                    System.out.print("["+arr[i]+"]"+ "  ");
                }
                System.out.println("\nSum = " + result.sum);
            }
        }
    }
}
```

OUTPUT:

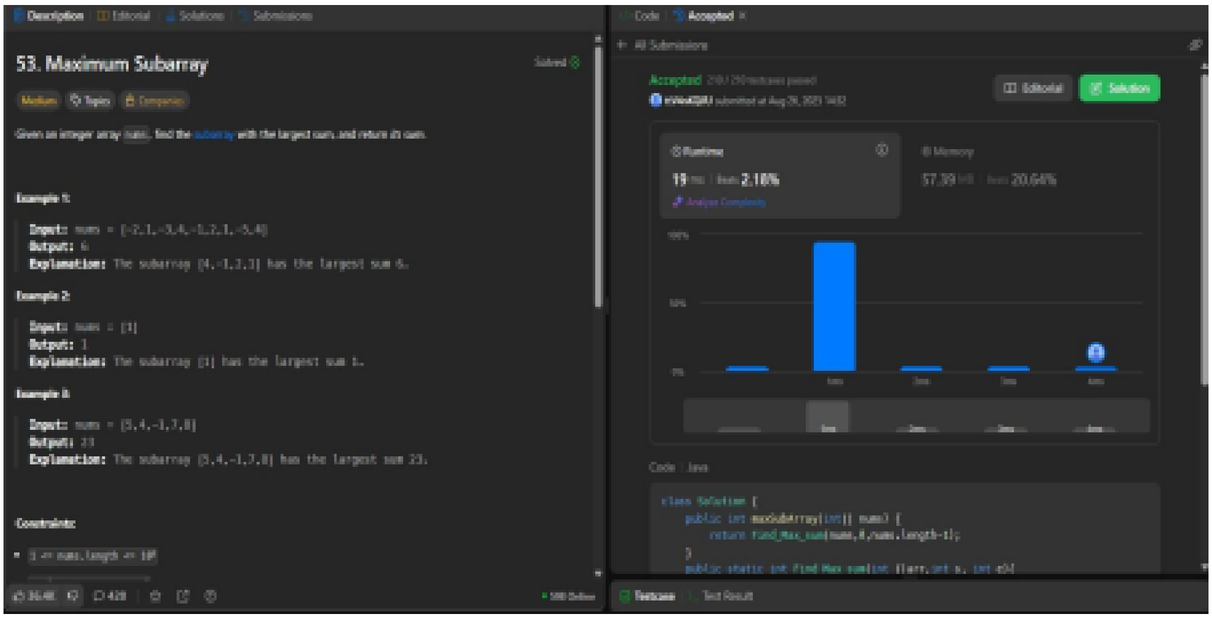


```
PS C:\Users\DT USER\Desktop\A1 B2 21 DAA LAB>  & 'C:\Program Files\Eclipse Adoptium\jdk-21.0.4.7-hotspot\
s\DT USER\AppData\Roaming\Code\User\workspaceStorage\9d279bf4ed2155571ec96edc7d739d8b\redhat.java\jdt_ws\
Enter number of test cases: 5
Enter size of array: 4
Enter elements:
1 2 3 4
Enter constraint value: 5
Best subarray with max sum under constraint:
[4]
Sum = 4
Enter size of array: 4
Enter elements:
2 2 2 2
Enter constraint value: 4
Best subarray with max sum under constraint:
[2] [2]
Sum = 4
Enter size of array: 3
Enter elements:
6 7 8
Enter constraint value: 5
No feasible subarray found.
Enter size of array: 5
Enter elements:
1 1 1 1 1
Enter constraint value: 4
Best subarray with max sum under constraint:
[1] [1] [1] [1]
Sum = 4
```

LEETCODE SUBMISSION:

Submit the same on Leetcode.

https://leetcode.com/problems/maximum-subarray/description/



53. Maximum Subarray

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 <= 10^5

Accepted

Runtime 19 ms | Beats 2.10%

Memory 57.39 MB | Beats 20.64%

Code | Java

```
class Solution {
    public int maxSubArray(int[] nums) {
        return Find_Max_sum(nums,0,nums.length-1);
    }
    public static int Find_Max_sum(int []arr,int s, int e){
```