Practical No. 4: DAA LAB

Name: Mohit Deepak Agrawal
Section: A2-B2
Roll No: 33

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 :

```
#include <iostream>

using namespace std;


int maxSubArr(int a[], int n, int limit, int &start, int &end) {

   int left = 0, sum = 0, maxSum = -1;


   start = -1;

   end = -1;
```

```
    for (int right = 0; right < n; right++) {
        sum += a[right];

        while (left <= right && sum > limit) {
            sum -= a[left];
            left++;
        }

        if (sum <= limit && sum > maxSum) {
            maxSum = sum;
            start = left;
            end = right;
        }
    }

    return maxSum;
}

int main() {
    int n, limit;
    cout << "Enter the size of the array : ";
    cin >> n;

    if (n <= 0) {
        cout << "No tasks given." << endl;
        return 0;
    }
```

```
    int a[n];
    cout << "Enter the resources :" << endl;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    cout << "Enter resource constraint : ";
    cin >> limit;

    int start, end;
    int result = maxSubArr(a, n, limit, start, end);

    if (result == 0) {
        cout << "No feasible subarray within constraint." << endl;
    } else {
        cout << "Sum: " << result << endl;
        cout << "Subarray: [";
        for (int i = start; i <= end; i++) {
            cout << a[i];
            if (i != end) cout << ", ";
        }
        cout << "]" << endl;
    }

    return 0;
}
```

Output :

```
Enter the size of the array : 4
Enter the resources :
2
1
3
4
Enter resource constraint : 5
Sum: 4
Subarray: [1, 3]
```

```
Enter the size of the array : 4
Enter the resources :
2
2
2
2
Enter resource constraint : 4
Sum: 4
Subarray: [2, 2]
```

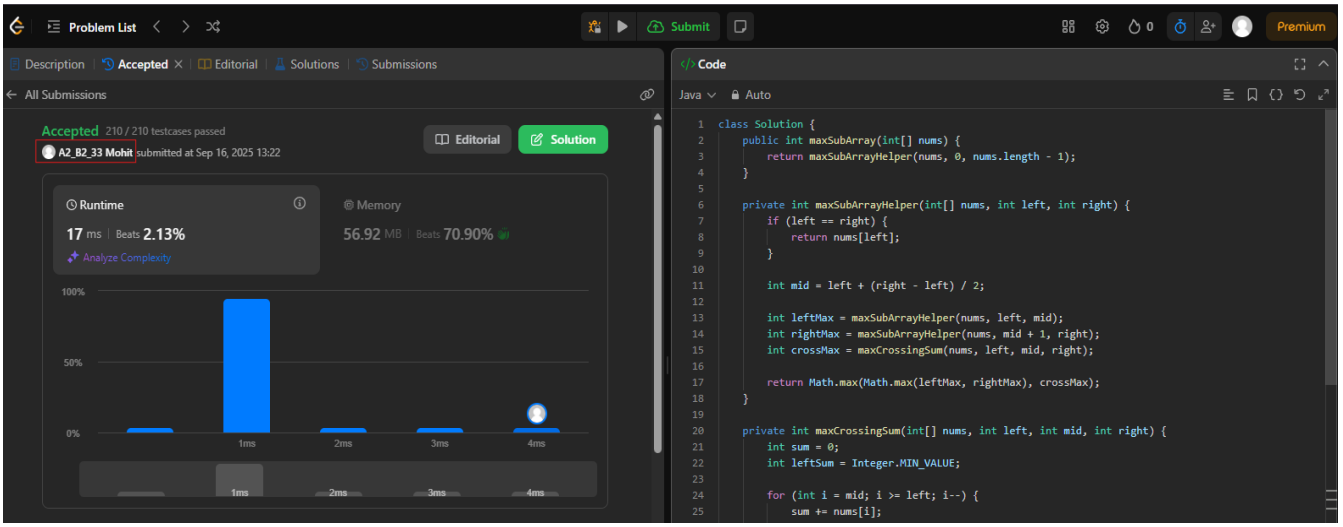
```
Enter the size of the array : 4
Enter the resources :
1
5
2
3
Enter resource constraint : 5
Sum: 5
Subarray: [5]
```

```
Enter the size of the array : 3
Enter the resources :
1
2
3
Enter resource constraint : 0
No feasible subarray within constraint.
```

```
Enter the size of the array : 3
Enter the resources :
6
7
8
Enter resource constraint : 5
No feasible subarray within constraint.
```

```
Enter the size of the array : 5
Enter the resources :
1
2
3
2
1
Enter resource constraint : 5
Sum: 5
Subarray: [2, 3]
```

Leet Code:




```
Code
Java  Auto
```

```java
class Solution {
    public int maxSubArray(int[] nums) {
        return maxSubArrayHelper(nums, 0, nums.length - 1);
    }

    private int maxSubArrayHelper(int[] nums, int left, int right) {
        if (left == right) {
            return nums[left];
        }

        int mid = left + (right - left) / 2;

        int leftMax = maxSubArrayHelper(nums, left, mid);
        int rightMax = maxSubArrayHelper(nums, mid + 1, right);
        int crossMax = maxCrossingSum(nums, left, mid, right);

        return Math.max(Math.max(leftMax, rightMax), crossMax);
    }

    private int maxCrossingSum(int[] nums, int left, int mid, int right) {
        int sum = 0;
        int leftSum = Integer.MIN_VALUE;

        for (int i = mid; i >= left; i--) {
            sum += nums[i];
            if (sum > leftSum) leftSum = sum;
        }

        sum = 0;
        int rightSum = Integer.MIN_VALUE;
```

```
31          for (int i = mid + 1; i <= right; i++) {
32              sum += nums[i];
33              if (sum > rightSum) rightSum = sum;
34          }
35
36          return leftSum + rightSum;
37      }
38
39      public int maxSubArrayKadane(int[] nums) {
40          int maxSoFar = nums[0];
41          int currentMax = nums[0];
42
43          for (int i = 1; i < nums.length; i++) {
44              currentMax = Math.max(nums[i], currentMax + nums[i]);
45              maxSoFar = Math.max(maxSoFar, currentMax);
46          }
47          return maxSoFar;
48      }
49  }
50
```