Name :	**Kunal Nayak**
Section :	**A2-B4**
Roll No :	**53**

Practical No. 4 : DAA Lab

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.
```

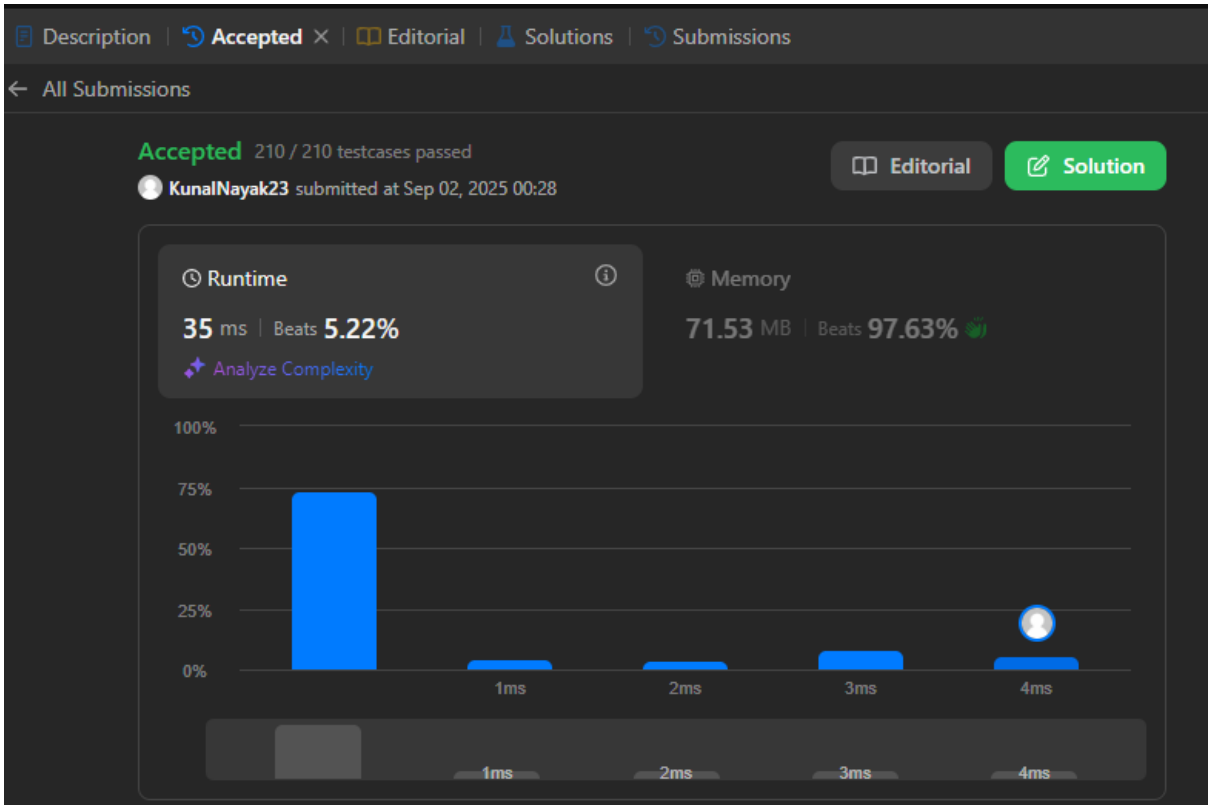
```
Enter the size of the array : 5
Enter the resources :
1
2
3
2
1
Enter resource constraint : 5
Sum: 5
Subarray: [2, 3]
```

Leetcode :

Code

C++ Auto

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>

class Solution {
private:
    int findMaxSum(const std::vector<int>& nums, int left, int right) {
        if (left == right) {
            return nums[left];
        }

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

        int left_half_sum = findMaxSum(nums, left, mid);
        int right_half_sum = findMaxSum(nums, mid + 1, right);

        long long current_sum = 0;
        long long max_left_border_sum = INT_MIN;
        for (int i = mid; i >= left; --i) {
            current_sum += nums[i];
            if (current_sum > max_left_border_sum) {
                max_left_border_sum = current_sum;
            }
        }

        current_sum = 0;
        long long max_right_border_sum = INT_MIN;
        for (int i = mid + 1; i <= right; ++i) {
            current_sum += nums[i];
            if (current_sum > max_right_border_sum) {
                max_right_border_sum = current_sum;
            }
        }
        long long cross_sum = max_left_border_sum + max_right_border_sum;

        return std::max({(long long)left_half_sum, (long long)right_half_sum, cross_sum});
    }

public:
    int maxSubArray(std::vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }
        return findMaxSum(nums, 0, nums.size() - 1);
    }
};
```


Description
Accepted
Editorial
Solutions
Submissions
All Submissions
Accepted 210 / 210 testcases passed
KunalNayak23 submitted at Sep 02, 2025 00:28
Editorial
Solution
Runtime
35 ms | Beats 5.22%
Analyze Complexity
Memory
71.53 MB | Beats 97.63%
100%
75%
50%
25%
0%
1ms
2ms
3ms
4ms


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