

Practical-4

Name: Krish Parothi

Section: A4

Batch: B3

Roll Number: 49

Aim: 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 of 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 in text format:

```
#include <stdio.h>

// Utility function to get max of two numbers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to find the max subarray sum that crosses the middle
int maxCrossingSum(int arr[], int left, int mid, int right, int
constraint) {
    int sum = 0;
    int left_sum = 0;

    // Include elements on left of mid
    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;

    // Include elements on right of mid
    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);
    }
}

// Recursive function using divide and conquer
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);
}

// Main function to call
int maxSubArraySum(int arr[], int n, int constraint) {
    if (n == 0 || constraint == 0) return 0;
    return maxSubArraySumUtil(arr, 0, n - 1, constraint);
}

// === Test Cases from PDF ===
int main() {
    // 1. Basic small array
    int arr1[] = {2, 1, 3, 4};
    printf("Test 1: %d\n", maxSubArraySum(arr1, 4, 5)); // Expected 4

    // 2. Exact match to constraint

```

```

int arr2[] = {2, 2, 2, 2};
printf("Test 2: %d\n", maxSubArraySum(arr2, 4, 4)); // Expected 4

// 3. Single element equals constraint
int arr3[] = {1, 5, 2, 3};
printf("Test 3: %d\n", maxSubArraySum(arr3, 4, 5)); // Expected 5

// 4. No feasible subarray
int arr4[] = {6, 7, 8};
printf("Test 4: %d\n", maxSubArraySum(arr4, 3, 5)); // Expected 0

// 5. Multiple optimal subarrays
int arr5[] = {1, 2, 3, 2, 1};
printf("Test 5: %d\n", maxSubArraySum(arr5, 5, 5)); // Expected 5

// 6. Large window valid
int arr6[] = {1, 1, 1, 1, 1};
printf("Test 6: %d\n", maxSubArraySum(arr6, 5, 4)); // Expected 4

// 7. Sliding window shrink needed
int arr7[] = {4, 2, 3, 1};
printf("Test 7: %d\n", maxSubArraySum(arr7, 4, 5)); // Expected 5

// 8. Empty array
int arr8[] = {};
printf("Test 8: %d\n", maxSubArraySum(arr8, 0, 10)); // Expected 0

// 9. Constraint = 0
int arr9[] = {1, 2, 3};
printf("Test 9: %d\n", maxSubArraySum(arr9, 3, 0)); // Expected 0

// 10. Very large input (stress test)
int n10 = 100000;
static int arr10[100000];
for (int i = 0; i < n10; i++) arr10[i] = i + 1;
printf("Test 10: %d\n", maxSubArraySum(arr10, n10, 1000000000)); //
Expected large ~ sum of full array

return 0;
}

```

Code Screen shot:

```
C Practical-4.c > ...
1  #include <stdio.h>
2
3  // Utility function to get max of two numbers
4  int max(int a, int b) {
5      return (a > b) ? a : b;
6  }
7
8  // Function to find the max subarray sum that crosses the middle
9  int maxCrossingSum(int arr[], int left, int mid, int right, int constraint) {
10     int sum = 0;
11     int left_sum = 0;
12
13     // Include elements on left of mid
14     for (int i = mid; i >= left; i--) {
15         sum += arr[i];
16         if (sum <= constraint) {
17             left_sum = max(left_sum, sum);
18         } else {
19             break;
20         }
21     }
22
23     sum = 0;
24     int right_sum = 0;
25
26     // Include elements on right of mid
27     for (int i = mid + 1; i <= right; i++) {
28         sum += arr[i];
29         if (sum <= constraint) {
30             right_sum = max(right_sum, sum);
31         } else {
32             break;
33         }
34     }
35
36     int total = left_sum + right_sum;
37     if (total <= constraint) {
38         return total;
39     } else {
40         return max(left_sum, right_sum);
41     }
42 }
43
```

```

44 // Recursive function using divide and conquer
45 int maxSubArraySumUtil(int arr[], int left, int right, int constraint) {
46     if (left == right) {
47         return (arr[left] <= constraint) ? arr[left] : 0;
48     }
49
50     int mid = (left + right) / 2;
51
52     int left_sum = maxSubArraySumUtil(arr, left, mid, constraint);
53     int right_sum = maxSubArraySumUtil(arr, mid + 1, right, constraint);
54     int cross_sum = maxCrossingSum(arr, left, mid, right, constraint);
55
56     return max(max(left_sum, right_sum), cross_sum);
57 }
58
59 // Main function to call
60 int maxSubArraySum(int arr[], int n, int constraint) {
61     if (n == 0 || constraint == 0) return 0;
62     return maxSubArraySumUtil(arr, 0, n - 1, constraint);
63 }
64
65 // === Test Cases from PDF ===
66 int main() {
67     // 1. Basic small array
68     int arr1[] = {2, 1, 3, 4};
69     printf("Test 1: %d\n", maxSubArraySum(arr1, 4, 5)); // Expected 4
70
71     // 2. Exact match to constraint
72     int arr2[] = {2, 2, 2, 2};
73     printf("Test 2: %d\n", maxSubArraySum(arr2, 4, 4)); // Expected 4
74
75     // 3. Single element equals constraint
76     int arr3[] = {1, 5, 2, 3};
77     printf("Test 3: %d\n", maxSubArraySum(arr3, 4, 5)); // Expected 5
78
79     // 4. No feasible subarray
80     int arr4[] = {6, 7, 8};
81     printf("Test 4: %d\n", maxSubArraySum(arr4, 3, 5)); // Expected 0
82
83     // 5. Multiple optimal subarrays
84     int arr5[] = {1, 2, 3, 2, 1};
85     printf("Test 5: %d\n", maxSubArraySum(arr5, 5, 5)); // Expected 5
86

```

```

83 // 5. Multiple optimal subarrays
84 int arr5[] = {1, 2, 3, 2, 1};
85 printf("Test 5: %d\n", maxSubArraySum(arr5, 5, 5)); // Expected 5
86
87 // 6. Large window valid
88 int arr6[] = {1, 1, 1, 1, 1};
89 printf("Test 6: %d\n", maxSubArraySum(arr6, 5, 4)); // Expected 4
90
91 // 7. Sliding window shrink needed
92 int arr7[] = {4, 2, 3, 1};
93 printf("Test 7: %d\n", maxSubArraySum(arr7, 4, 5)); // Expected 5
94
95 // 8. Empty array
96 int arr8[] = {};
97 printf("Test 8: %d\n", maxSubArraySum(arr8, 0, 10)); // Expected 0
98
99 // 9. Constraint = 0
100 int arr9[] = {1, 2, 3};
101 printf("Test 9: %d\n", maxSubArraySum(arr9, 3, 0)); // Expected 0
102
103 // 10. Very large input (stress test)
104 int n10 = 100000;
105 static int arr10[100000];
106 for (int i = 0; i < n10; i++) arr10[i] = i + 1;
107 printf("Test 10: %d\n", maxSubArraySum(arr10, n10, 100000000)); // Expected large ~ sum of full array
108
109 return 0;
110 }
111

```

OUTPUT:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  Filter
[Running] cd "c:\Users\Krish\OneDrive\Desktop\RBU\RBU-Sem-3\LABS\DESIGN ALGORITHM ANALYSIS\Practical-4\" && gcc Pract
"c:\Users\Krish\OneDrive\Desktop\RBU\RBU-Sem-3\LABS\DESIGN ALGORITHM ANALYSIS\Practical-4\"Practical-4
Test 1: 4
Test 2: 4
Test 3: 5
Test 4: 0
Test 5: 5
Test 6: 3
Test 7: 4
Test 8: 0
Test 9: 0
Test 10: 999984681

[Done] exited with code=0 in 2.139 seconds

```