PRACTICAL NO. 4

Name: Isha Raut
Section: A4
Batch: B2
Roll No: 23

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.

```c
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#define MAX_SIZE 100000
int maximum_crossSubarray(int n, int arr[n], int low, int mid, int high)
{
    int left_sum = INT_MIN;
    int max_left;
    int sum = 0;

    for (int i = mid; i >= low; i--)
    {
        sum += arr[i];
        if (sum > left_sum)
        {
            left_sum = sum;
            max_left = i;
        }
    }

    int right_sum = INT_MIN;
    int max_right;
    sum = 0;

    for (int j = mid + 1; j <= high; j++)
    {
        sum += arr[j];
        if (sum > right_sum)
        {
            right_sum = sum;
            max_right = j;
        }
    }

    return left_sum + right_sum;
}
```

```
int maximum_subarray(int n, int arr[n], int low, int high)
{
    if (high == low)
    {
        return arr[low];
    }
    else
    {
        int mid = (low + high) / 2;

        int left_sum = maximum_subarray(n, arr, low, mid);

        int right_sum = maximum_subarray(n, arr, mid + 1, high);

        int cross_sum = maximum_crossSubarray(n, arr, low, mid, high);

        if (left_sum >= right_sum && left_sum >= cross_sum)
        {
            return left_sum;
        }
        else if (right_sum >= left_sum && right_sum >= cross_sum)
        {
            return right_sum;
        }
        else
        {
            return cross_sum;
        }
    }
}
```

```c
// Sliding window approach for max subarray sum <= constraint
int max_subarray_sum_with_constraint(int arr[], int n, int constraint, int* start_index,
    int max_sum = 0;
    int current_sum = 0;
    int left = 0;
    *start_index = 0;
    *end_index = -1;

    for (int right = 0; right < n; right++) {
        current_sum += arr[right];

        // Shrink window from left while sum > constraint
        while (current_sum > constraint && left <= right) {
            current_sum -= arr[left];
            left++;
        }

        // Update max_sum and indices if current_sum valid and better (tie-break last)
        if (current_sum <= constraint && current_sum >= max_sum) {
            max_sum = current_sum;
            *start_index = left;
            *end_index = right;
        }
    }

    return max_sum;
}
```

```c
int main()
{
    int arr[] = {13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    int max_sum = maximum_subarray(n, arr, 0, n - 1);
    printf("Maximum subarray sum: %d\n", max_sum);
```

Test Cases:

1. Basic small array
• resources = [2, 1, 3, 4], constraint = 5
o Best subarray: [2, 1] or [1, 3] → sum = 4
o Checks simple working.

```c
// Test case 1: Basic small array
int arr1[] = {2, 1, 3, 4};
int n1 = sizeof(arr1) / sizeof(arr1[0]);
int constraint = 5;
int start_index, end_index;
int max_sum1 = max_subarray_sum_with_constraint(arr1, n1, constraint, &start_index, &end_index);

printf("Test case 1 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum1);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr1[i]);
}
printf("\n");
```

2. Exact match to constraint
• resources = [2, 2, 2, 2], constraint = 4
o Best subarray: [2, 2] → sum = 4
o Tests exact utilization.

```c
// Test case 2: Exact match to constraint
int arr2[] = {2, 2, 2, 2};
int n2 = sizeof(arr2) / sizeof(arr2[0]);

int max_sum2 = max_subarray_sum_with_constraint(arr2, n2, constraint, &start_index, &end_index);

printf("Test case 2 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum2);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr2[i]);
}
printf("\n");
```

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

```c
// Test case 3: Single element equals constraint
int arr3[] = {1, 5, 2, 3};
int n3 = sizeof(arr3) / sizeof(arr3[0]);

int max_sum3 = max_subarray_sum_with_constraint(arr3, n3, constraint, &start_index, &end_index);

printf("Test case 3 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum3);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr3[i]);
}
printf("\n");
```

4. All elements smaller but no combination fits
• resources = [6, 7, 8], constraint = 5
o No feasible subarray.
o Tests "no solution" case.

```c
// Test case 4: All elements smaller but no combination fits
int arr4[] = {6, 7, 8};
int n4 = sizeof(arr4) / sizeof(arr4[0]);

int max_sum4 = max_subarray_sum_with_constraint(arr4, n4, constraint, &start_index, &end_index);

printf("Test case 4 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum4);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr4[i]);
}
printf("\n");
```

5. Multiple optimal subarrays
• resources = [1, 2, 3, 2, 1], constraint = 5
o Best subarrays: [2, 3] and [3, 2] → sum = 5
o Tests tie-breaking (should return either valid subarray).

```c
// Test case 5: Multiple optimal subarrays
int arr5[] = {1, 2, 3, 2, 1};
int n5 = sizeof(arr5) / sizeof(arr5[0]);

int max_sum5 = max_subarray_sum_with_constraint(arr5, n5, constraint, &start_index, &end_index);

printf("Test case 5 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum5);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr5[i]);
}
printf("\n");
```

6. Large window valid
• resources = [1, 1, 1, 1, 1], constraint = 4
o Best subarray: [1, 1, 1, 1] → sum = 4
o Ensures long window works.

```c
// Test case 6: Large window valid
int arr6[] = {1, 1, 1, 1, 1};
int n6 = sizeof(arr6) / sizeof(arr6[0]);
constraint = 4;
int max_sum6 = max_subarray_sum_with_constraint(arr6, n6, constraint, &start_index, &end_index);

printf("Test case 6 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum6);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr6[i]);
}
printf("\n");
```

7. Sliding window shrink needed

• resources = [4, 2, 3, 1], constraint = 5
o Start [4,2] = 6 (too big) → shrink to [2,3] = 5.
o Tests dynamic window adjustment.

```c
// Test case 7: Sliding window shrink needed
int arr7[] = {4, 2, 3, 1};
int n7 = sizeof(arr7) / sizeof(arr7[0]);
constraint = 5;

int max_sum7 = max_subarray_sum_with_constraint(arr7, n7, constraint, &start_index, &end_index);

printf("Test case 7 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum7);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr7[i]);
}
printf("\n");
```

8. Empty array
• resources = [], constraint = 10
o Output: no subarray.
o Edge case: empty input.

```c
// Test case 8: Empty array
int arr8[] = {};
int n8 = sizeof(arr8) / sizeof(arr8[0]);
constraint = 10;
int max_sum8 = max_subarray_sum_with_constraint(arr8, n8, constraint, &start_index, &end_index);

printf("Test case 8 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum8);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr8[i]);
}
printf("\n");
```

9. Constraint = 0
• resources = [1, 2, 3], constraint = 0
o No subarray possible.
o Edge case: zero constraint.

```c
// Test case 9: Constraint zero, no valid subarray
int arr9[] = {1, 2, 3};
int n9 = sizeof(arr9) / sizeof(arr9[0]);
constraint = 0;
int max_sum9 = max_subarray_sum_with_constraint(arr9, n9, constraint, &start_index, &end_index);

printf("Test case 9 -> Maximum subarray sum under constraint %d: %d\n", constraint, max_sum9);
printf("Best subarray: ");
for (int i = start_index; i <= end_index; i++) {
    printf("%d ", arr9[i]);
}
printf("\n");
```

10. Very large input (stress test)
• resources = [1, 2, 3, ..., 100000], constraint = 10^9
o Valid subarray near full array.
o Performance test.

```c
    // Test case 10: Very large input (stress test) without dynamic allocation

        static int arr10[MAX_SIZE];
        int n10 = MAX_SIZE;
        int constraint10 = 1000000000; // 10^9
        int start_index10, end_index10;

        // Initialize arr10 with values 1, 2, 3, ..., 100000
        for (int i = 0; i < n10; i++) {
            arr10[i] = i + 1;
        }

        int max_sum10 = max_subarray_sum_with_constraint(arr10, n10, constraint10, &start_index10, &end_index10);

        printf("Test case 10 -> Maximum subarray sum under constraint %d: %d\n", constraint10, max_sum10);
        printf("Best subarray starts at index %d and ends at index %d\n", start_index10, end_index10);
        printf("Subarray length: %d\n", end_index10 - start_index10 + 1);


    return 0;
}
```

Output:

```
Maximum subarray sum: 43
Test case 1 -> Maximum subarray sum under constraint 5: 4
Best subarray: 4
Test case 2 -> Maximum subarray sum under constraint 5: 4
Best subarray: 2 2
Test case 3 -> Maximum subarray sum under constraint 5: 5
Best subarray: 2 3
Test case 4 -> Maximum subarray sum under constraint 5: 0
Best subarray:
Test case 5 -> Maximum subarray sum under constraint 5: 5
Best subarray: 3 2
Test case 6 -> Maximum subarray sum under constraint 4: 4
Best subarray: 1 1 1 1
Test case 7 -> Maximum subarray sum under constraint 5: 5
Best subarray: 2 3
Test case 8 -> Maximum subarray sum under constraint 10: 0
Best subarray:
Test case 9 -> Maximum subarray sum under constraint 0: 0
Best subarray:
Test case 10 -> Maximum subarray sum under constraint 1000000000: 1000000000
Best subarray starts at index 56187 and ends at index 71811
Subarray length: 15625
```