Practical -4

Name-Huzaif Baig

Roll no-40

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.

```
#include <iostream>
#include <vector>
using namespace std;

// 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(const vector<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(const vector<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(const vector<int>& arr, int constraint) {
    if (arr.empty()) return 0;
    return maxSubArraySumUtil(arr, 0, (int)arr.size() - 1, constraint);
}

// === Test ===
int main() {
    vector<int> arr1 = {2, 1, 3, 4};
    int constraint1 = 5;
    cout << "Test 1: Max sum = " << maxSubArraySum(arr1,
constraint1) << "\n"; // Expected: 4

    vector<int> arr2 = {2, 2, 2, 2};
    int constraint2 = 4;
    cout << "Test 2: Max sum = " << maxSubArraySum(arr2,
constraint2) << "\n"; // Expected: 4

    vector<int> arr3 = {1, 5, 2, 3};
    int constraint3 = 5;
    cout << "Test 3: Max sum = " << maxSubArraySum(arr3,
constraint3) << "\n"; // Expected: 5

    vector<int> arr4 = {6, 7, 8};
    int constraint4 = 5;
    cout << "Test 4: Max sum = " << maxSubArraySum(arr4,
constraint4) << "\n"; // Expected: 0

    vector<int> arr5 = {1, 1, 1};
    int constraint5 = 5;
    cout << "Test 5: Max sum = " << maxSubArraySum(arr5,
constraint5) << "\n"; // Expected: 3

    return 0;
}
```

Output

```
Test 1: Max sum = 4
Test 2: Max sum = 4
Test 3: Max sum = 5
Test 4: Max sum = 0
Test 5: Max sum = 3
```