Name :	Swati Kundu
Section :	A4-B3
Roll No :	48

Practical 4

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.

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.

2. Exact match to constraint

• resources = [2, 2, 2, 2], constraint = 4

o Best subarray: [2, 2] → sum = 4

o Tests exact utilization.

3. Single element equals constraint

• resources = [1, 5, 2, 3], constraint = 5

o Best subarray: [5] → sum = 5

o Tests one-element solution.

4. All elements smaller but no combination fits

• resources = [6, 7, 8], constraint = 5

o No feasible subarray.

o Tests "no solution" case.

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).

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.

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.

8. Empty array

• resources = [], constraint = 10

o Output: no subarray.

o Edge case: empty input.

9. Constraint = 0

• resources = [1, 2, 3], constraint = 0

o No subarray possible.

o Edge case: zero constraint.

CODE:

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

// Struct to represent the result of a subarray
typedef struct {
    int left;
    int right;
    int sum;
} Subarray;
```

```
// Return an empty result
Subarray emptyResult() {
   Subarray res = {-1, -1, 0};
   return res;
}

// Function to find the best crossing subarray within constraint
Subarray maxCrossingSubarray(int arr[], int left, int mid, int right, int
constraint) {
   int leftSum = 0, maxLeftSum = INT_MIN;
   int maxLeft = mid, sum = 0;

   for (int i = mid; i >= left; i--) {
      sum += arr[i];
      if (sum <= constraint && sum > maxLeftSum) {
         maxLeftSum = sum;
         maxLeft = i;
      }
   }

   int rightSum = 0, maxRightSum = INT_MIN;
   int maxRight = mid + 1;
   sum = 0;
```

```
    for (int i = mid + 1; i <= right; i++) {
        sum += arr[i];
        if (sum <= constraint && sum > maxRightSum) {
            maxRightSum = sum;
            maxRight = i;
        }
    }

    if (maxLeftSum == INT_MIN && maxRightSum == INT_MIN) {
        return emptyResult();
    }

    int totalSum = 0;
    if (maxLeftSum != INT_MIN) totalSum += maxLeftSum;
    if (maxRightSum != INT_MIN) totalSum += maxRightSum;

    if (totalSum <= constraint) {
        Subarray result = {maxLeft, maxRight, totalSum};
        return result;
    }

    return emptyResult();
}

// Divide and conquer function to find max subarray under constraint
```

```
Subarray maxSubarrayDC(int arr[], int left, int right, int constraint) {
  if (left > right) return emptyResult();

  if (left == right) {
    if (arr[left] <= constraint) {
      Subarray result = {left, left, arr[left]};
      return result;
    } else {
      return emptyResult();
    }
  }

  int mid = (left + right) / 2;

  Subarray leftRes = maxSubarrayDC(arr, left, mid, constraint);
  Subarray rightRes = maxSubarrayDC(arr, mid + 1, right, constraint);
  Subarray crossRes = maxCrossingSubarray(arr, left, mid, right,
constraint);

  Subarray best = emptyResult();
  if (leftRes.sum > best.sum) best = leftRes;
  if (rightRes.sum > best.sum) best = rightRes;
  if (crossRes.sum > best.sum) best = crossRes;

  return best;
```

```
}

// Safely handles empty or null input
void findMaxSubarray(int arr[], int size, int constraint) {
    if (arr == NULL || size <= 0 || constraint <= 0) {
        printf("No feasible subarray found under constraint %d.\n", constraint);
        return;
    }

    Subarray result = maxSubarrayDC(arr, 0, size - 1, constraint);
    if (result.left == -1) {
        printf("No feasible subarray found under constraint %d.\n", constraint);
    } else {
        printf("Best subarray (sum = %d): [", result.sum);
        for (int i = result.left; i <= result.right; i++) {
            printf("%d", arr[i]);
            if (i < result.right) printf(", ");
        }
        printf("]\n");
    }
}
int main() {
    // Test Case 1: Basic small array
    printf("Test Case 1:\n");
    int arr1[] = {2, 1, 3, 4};
```

```
findMaxSubarray(arr1, 4, 5);

// Test Case 2: Exact match
printf("Test Case 2:\n");
int arr2[] = {2, 2, 2, 2};
findMaxSubarray(arr2, 4, 4);

// Test Case 3: Single element equals constraint
printf("Test Case 3:\n");
int arr3[] = {1, 5, 2, 3};
findMaxSubarray(arr3, 4, 5);

// Test Case 4: All elements too large
printf("Test Case 4:\n");
int arr4[] = {6, 7, 8};
findMaxSubarray(arr4, 3, 5);

// Test Case 5: Multiple optimal subarrays
printf("Test Case 5:\n");
int arr5[] = {1, 2, 3, 2, 1};
findMaxSubarray(arr5, 5, 5);

// Test Case 6: Large window valid
printf("Test Case 6:\n");
int arr6[] = {1, 1, 1, 1, 1};
```

```
    findMaxSubarray(arr6, 5, 4);

    // Test Case 7: Shrinking window needed
    printf("Test Case 7:\n");
    int arr7[] = {4, 2, 3, 1};
    findMaxSubarray(arr7, 4, 5);

    // Test Case 8: Empty array
    printf("Test Case 8:\n");
    int *arr8 = NULL;
    findMaxSubarray(arr8, 0, 10);

    // Test Case 9: Constraint = 0
    printf("Test Case 9:\n");
    int arr9[] = {1, 2, 3};
    findMaxSubarray(arr9, 3, 0);

    return 0;
}
```

Output:

Output Clear

```
Test Case 1:
Best subarray (sum = 4): [4]
Test Case 2:
Best subarray (sum = 4): [2, 2]
Test Case 3:
Best subarray (sum = 5): [5]
Test Case 4:
No feasible subarray found under constraint 5.
Test Case 5:
Best subarray (sum = 3): [1, 2]
Test Case 6:
Best subarray (sum = 3): [1, 1, 1]
Test Case 7:
Best subarray (sum = 4): [4]
Test Case 8:
No feasible subarray found under constraint 10.
Test Case 9:
No feasible subarray found under constraint 0.


=== Code Execution Successful ===
```