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:

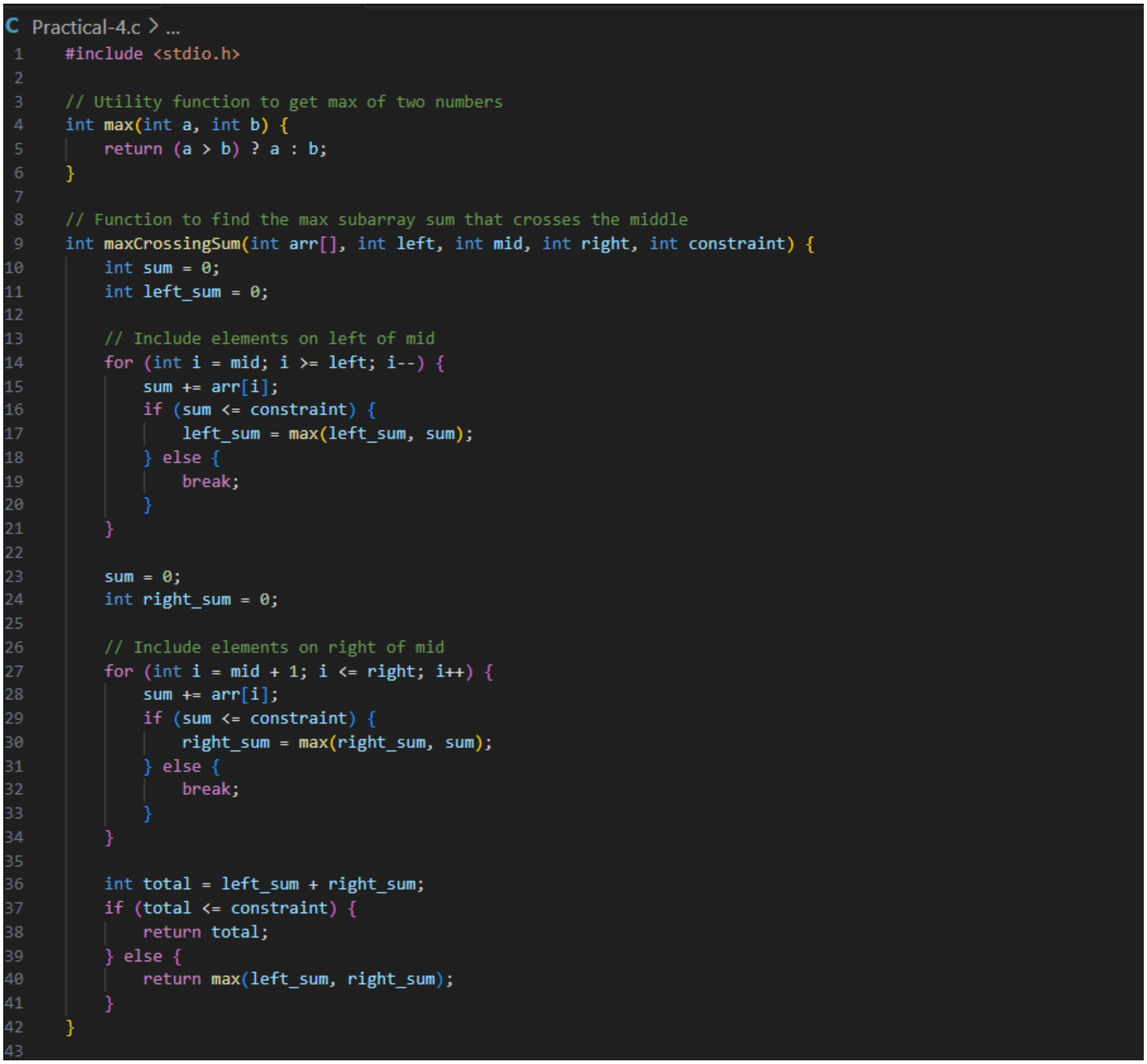
A project requires allocating resources to various tasks over a period of time. Each task requires a certain amount of resources, and you want to maximize the overall efficiency of resource usage. You're given an array of resources where resources[i] represents the amount of resources required for the ith 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.

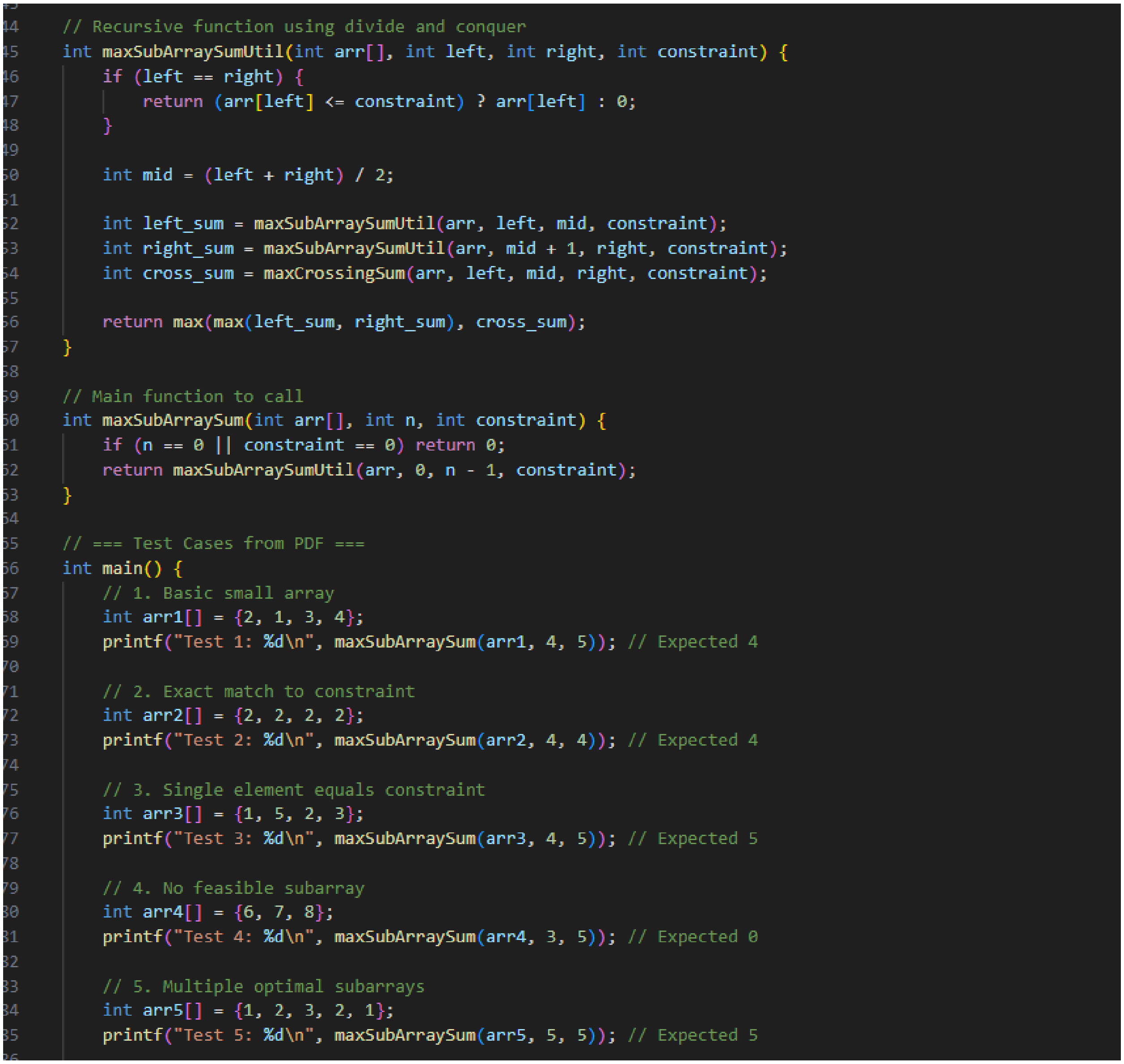
Code in text format:

```
#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;
}
```

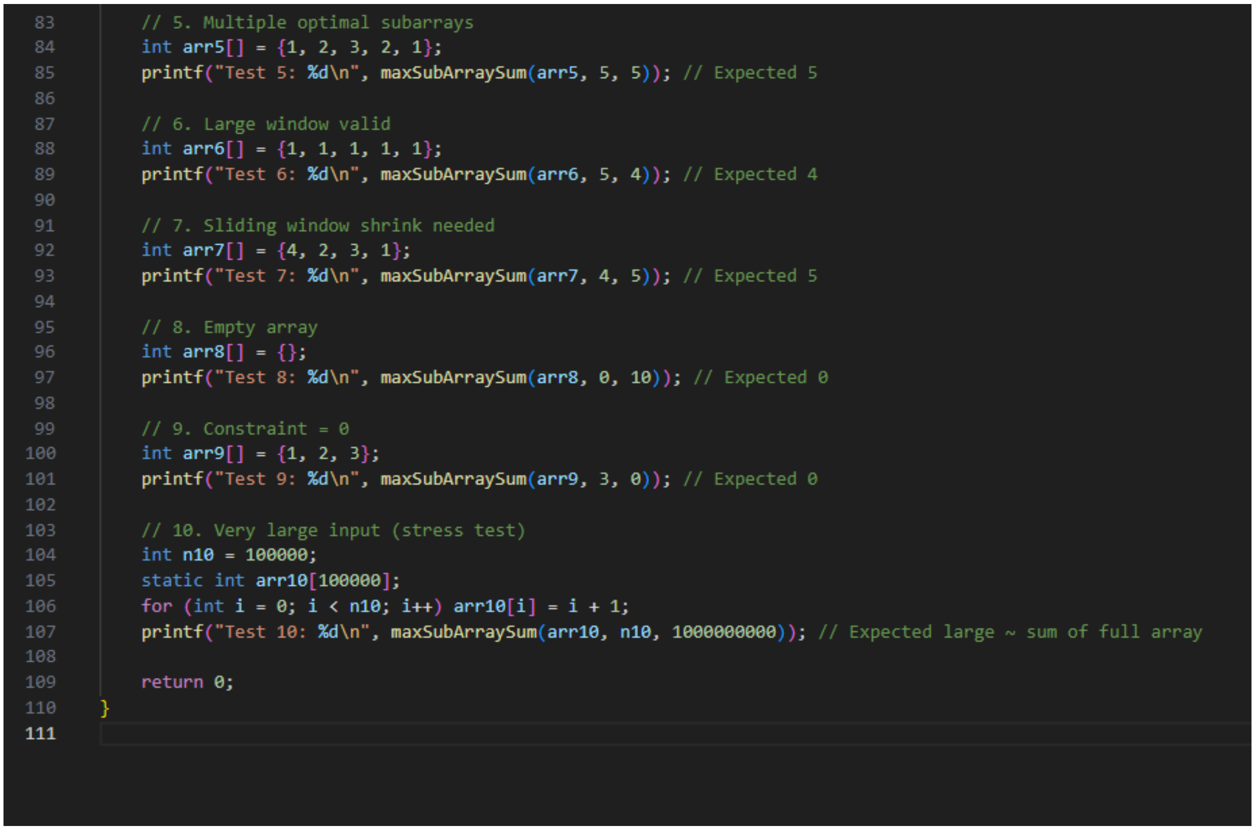
Code Screen shot:



```c
#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);
    }
}
```

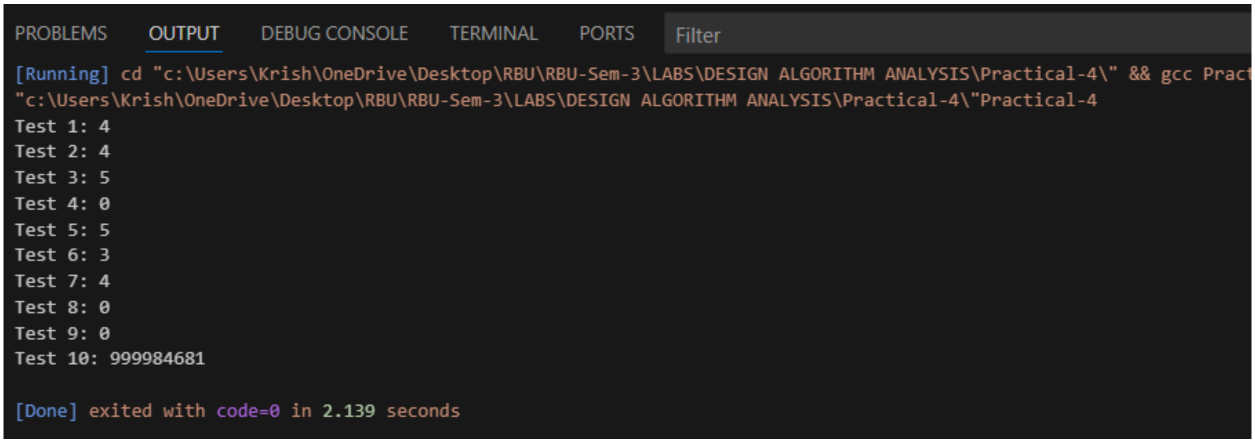


```c
// Recursive function using divide and conquer
int maxSubArraySumUtil(int arr[], int left, int right, int constraint) {
    if (left == right) {
        return (arr[left] <= constraint) ? arr[left] : 0;
    }

    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
```



```c
    // 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;
}
```

OUTPUT:



```
[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
```