Name - Abhishek Jagam
Class - A4_B2_21
Practical - 4

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.
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.

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

Code:

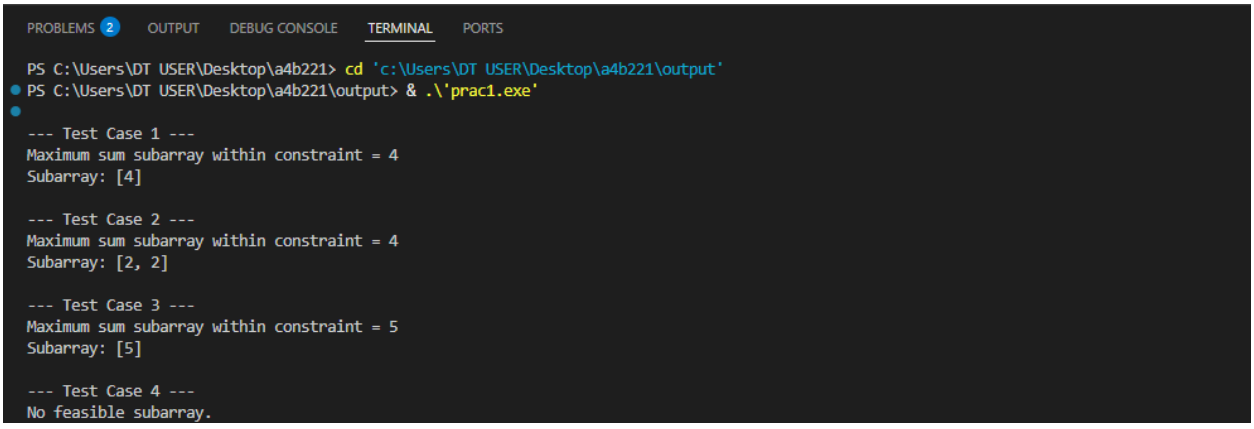
```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

typedef struct {
    int start, end;
    long long sum;
} Result;

typedef struct {
    long long sum;
    int idx;
} Pair;
```

```
int cmpPair(const void *a, const void *b) {
    Pair *pa = (Pair*)a;
    Pair *pb = (Pair*)b;
    if (pa->sum < pb->sum) return -1;
    if (pa->sum > pb->sum) return 1;
    return 0;
}

Result make_result(int s, int e, long long sum) {
    Result r;
    r.start = s; r.end = e; r.sum = sum;
    return r;
}

Result findMaxCrossSubArray(int *arr, int low, int mid, int high, long long constraint) {
    int capL = mid - low + 1;
    int capR = high - mid;
    Pair *Lpairs = malloc(sizeof(Pair) * capL);
    Pair *Rpairs = malloc(sizeof(Pair) * capR);
    int Lcount = 0, Rcount = 0;

    long long sum = 0;
    for (int i = mid; i >= low; --i) {
        sum += arr[i];
        if (sum <= constraint) {
            Lpairs[Lcount].sum = sum;
            Lpairs[Lcount].idx = i;
            Lcount++;
        }
    }

    sum = 0;
    for (int j = mid + 1; j <= high; ++j) {
        sum += arr[j];
        if (sum <= constraint) {
            Rpairs[Rcount].sum = sum;
            Rpairs[Rcount].idx = j;
            Rcount++;
        }
    }

    Result res = make_result(-1, -1, LLONG_MIN);
    if (Lcount == 0 || Rcount == 0) {
```

```
        free(Lpairs); free(Rpairs);
        return res;
    }

    qsort(Lpairs, Lcount, sizeof(Pair), cmpPair);
    qsort(Rpairs, Rcount, sizeof(Pair), cmpPair);

    int i = 0, j = Rcount - 1;
    long long best = LLONG_MIN;
    int bestLi = -1, bestRj = -1;

    while (i < Lcount && j >= 0) {
        long long cur = Lpairs[i].sum + Rpairs[j].sum;
        if (cur > constraint) {
            j--;
        } else {
            if (cur > best) {
                best = cur;
                bestLi = Lpairs[i].idx;
                bestRj = Rpairs[j].idx;
            }
            i++;
        }
    }

    if (best != LLONG_MIN) {
        res.start = bestLi;
        res.end = bestRj;
        res.sum = best;
    }

    free(Lpairs); free(Rpairs);
    return res;
}

Result findMaxSubArray(int *arr, int low, int high, long long constraint) {
    if (low > high) return make_result(-1, -1, LLONG_MIN);
    if (low == high) {
        if (arr[low] <= constraint) return make_result(low, low, arr[low]);
        else return make_result(-1, -1, LLONG_MIN);
    }
    int mid = (low + high) / 2;
    Result left = findMaxSubArray(arr, low, mid, constraint);
    Result right = findMaxSubArray(arr, mid + 1, high, constraint);
```

```
    Result cross = findMaxCrossSubArray(arr, low, mid, high, constraint);

    Result best = left;
    if (right.sum > best.sum) best = right;
    if (cross.sum > best.sum) best = cross;
    return best;
}

void runTest(int *arr, int n, long long constraint, int testCase) {
    printf("\n--- Test Case %d ---\n", testCase);
    Result ans = findMaxSubArray(arr, 0, n - 1, constraint);

    if (ans.sum == LLONG_MIN) {
        printf("No feasible subarray.\n");
    } else {
        printf("Maximum sum subarray within constraint = %lld\n", ans.sum);
        printf("Subarray: [");
        for (int i = ans.start; i <= ans.end; ++i) {
            printf("%d", arr[i]);
            if (i < ans.end) printf(", ");
        }
        printf("]\n");
    }
}

int main() {
    // Test 1
    int arr1[] = {2, 1, 3, 4};
    runTest(arr1, 4, 5, 1);

    // Test 2
    int arr2[] = {2, 2, 2, 2};
    runTest(arr2, 4, 4, 2);

    // Test 3
    int arr3[] = {1, 5, 2, 3};
    runTest(arr3, 4, 5, 3);

    // Test 4
    int arr4[] = {6, 7, 8};
    runTest(arr4, 3, 5, 4);

    // Test 5
    int arr5[] = {1, 2, 3, 2, 1};
```

```
    runTest(arr5, 5, 5, 5);

    // Test 6
    int arr6[] = {1, 1, 1, 1, 1};
    runTest(arr6, 5, 4, 6);

    // Test 7
    int arr7[] = {4, 2, 3, 1};
    runTest(arr7, 4, 5, 7);

    // Test 8 (empty array)
    runTest(NULL, 0, 10, 8);

    // Test 9
    int arr9[] = {1, 2, 3};
    runTest(arr9, 3, 0, 9);

    // Test 10 (large stress test)
    int n = 100000;
    int *arr10 = malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++) arr10[i] = i + 1;
    runTest(arr10, n, 1000000000LL, 10);
    free(arr10);

    return 0;
}
```



```
--- Test Case 5 ---
Maximum sum subarray within constraint = 5
Subarray: [2, 3]

--- Test Case 6 ---
Maximum sum subarray within constraint = 4
Subarray: [1, 1, 1, 1]

--- Test Case 7 ---
Maximum sum subarray within constraint = 5
Subarray: [2, 3]

--- Test Case 8 ---
No feasible subarray.

--- Test Case 9 ---
No feasible subarray.
```

Leetcode :

https://leetcode.com/problems/maximum-subarray/submissions/1782426164/