PRACTICAL NUMBER 4

Name:Yash Rai

Class : A7-B3

Roll Number:35

Date: 1 September 2025

Time: 3:00pm

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 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.

Code:

```c
#include <stdio.h>

struct Subarray {
    int start;
    int end;
    int sum;
};

struct Subarray maxSubarray(struct Subarray a, struct Subarray b) {
    if (a.sum > b.sum) return a;
    if (b.sum > a.sum) return b;
    return a;
}

struct Subarray maxCrossingSubarray(int arr[], int low, int mid, int high, int
constraint) {
    int left_sum = 0, right_sum = 0;
    int sum = 0, i;
    int best_left = -1, best_right = -1;

    sum = 0;
    for (i = mid; i >= low; i--) {
        sum += arr[i];
        if (sum <= constraint && sum > left_sum) {
            left_sum = sum;
            best_left = i;
        }
    }

    sum = 0;
    for (i = mid + 1; i <= high; i++) {
        sum += arr[i];
        if (sum <= constraint && sum > right_sum) {
            right_sum = sum;
            best_right = i;
        }
    }

    struct Subarray result;
    if (best_left != -1 && best_right != -1 && left_sum + right_sum <=
constraint) {
        result.start = best_left;
        result.end = best_right;
        result.sum = left_sum + right_sum;
    } else {
        result.start = -1;
        result.end = -1;
        result.sum = 0;
    }
    return result;
}

struct Subarray maxSubarrayDC(int arr[], int low, int high, int constraint) {
    if (low == high) {
        struct Subarray base;
        if (arr[low] <= constraint) {
            base.start = low;
            base.end = low;
            base.sum = arr[low];
        } else {
            base.start = -1;
            base.end = -1;
            base.sum = 0;
        }
        return base;
    }

    int mid = (low + high) / 2;
    struct Subarray left = maxSubarrayDC(arr, low, mid, constraint);
    struct Subarray right = maxSubarrayDC(arr, mid + 1, high, constraint);
    struct Subarray cross = maxCrossingSubarray(arr, low, mid, high, constraint);

    struct Subarray best = maxSubarray(left, right);
    best = maxSubarray(best, cross);

    return best;
}

int main() {
    int n, constraint;
    printf("Enter the size of array:\n");
    scanf("%d", &n);
    int resources[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the %d index element of the array:\n", i);
        scanf("%d", &resources[i]);
    }

    printf("The arrays is {");
    for (int i = 0; i < n; i++) {
        printf("%d",resources[i]);
    }
    printf("}.\n");



    printf("Enter the size of constraint:\n");
    scanf("%d", &constraint);

    if (n == 0 || constraint <= 0) {
        printf("No feasible subarray.\n");
        return 0;
    }

    struct Subarray result = maxSubarrayDC(resources, 0, n - 1, constraint);

    if (result.start == -1) {
        printf("No feasible subarray.\n");
    } else {
        printf("Best subarray (indices %d to %d): ", result.start, result.end);
        for (int i = result.start; i <= result.end; i++) {
            printf("%d ", resources[i]);
        }
        printf("\nMaximum sum = %d\n", result.sum);
    }

    return 0;
}
```

Output:

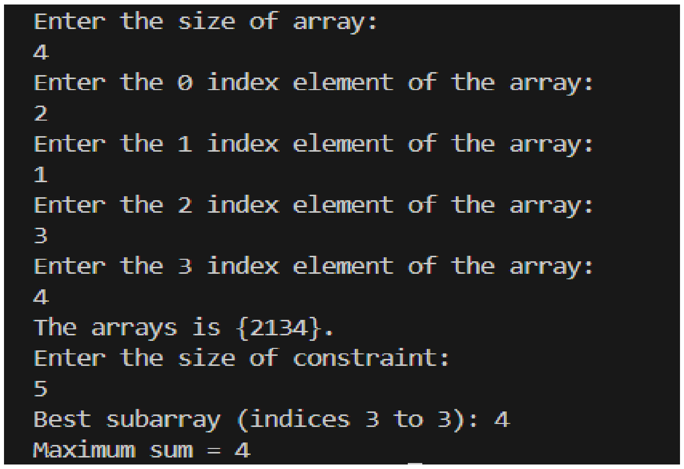
Problem Statement 1:

Basic small array

resources = [2, 1, 3, 4], constraint = 5

Best subarray: [2, 1] or [1, 3] → sum = 4

Checks simple working.



```
Enter the size of array:
4
Enter the 0 index element of the array:
2
Enter the 1 index element of the array:
1
Enter the 2 index element of the array:
3
Enter the 3 index element of the array:
4
The arrays is {2134}.
Enter the size of constraint:
5
Best subarray (indices 3 to 3): 4
Maximum sum = 4
```

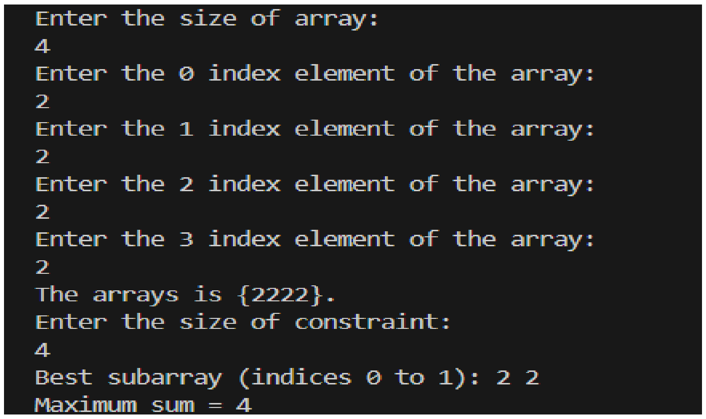
Problem Statement 2:

Exact match to constraint

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

Best subarray: [2, 2] → sum = 4

Tests exact utilization.



```
Enter the size of array:
4
Enter the 0 index element of the array:
2
Enter the 1 index element of the array:
2
Enter the 2 index element of the array:
2
Enter the 3 index element of the array:
2
The arrays is {2222}.
Enter the size of constraint:
4
Best subarray (indices 0 to 1): 2 2
Maximum sum = 4
```

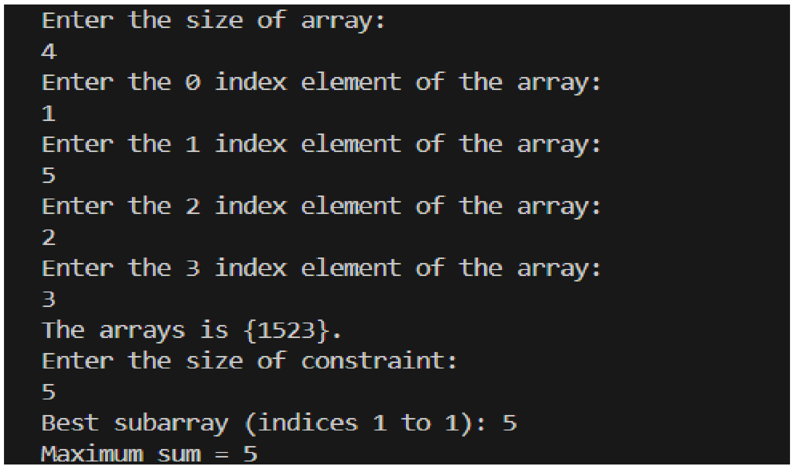
Problem Statement 3:

Single element equals constraint

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

Best subarray: [5] → sum = 5

Tests one-element solution.



```
Enter the size of array:
4
Enter the 0 index element of the array:
1
Enter the 1 index element of the array:
5
Enter the 2 index element of the array:
2
Enter the 3 index element of the array:
3
The arrays is {1523}.
Enter the size of constraint:
5
Best subarray (indices 1 to 1): 5
Maximum sum = 5
```

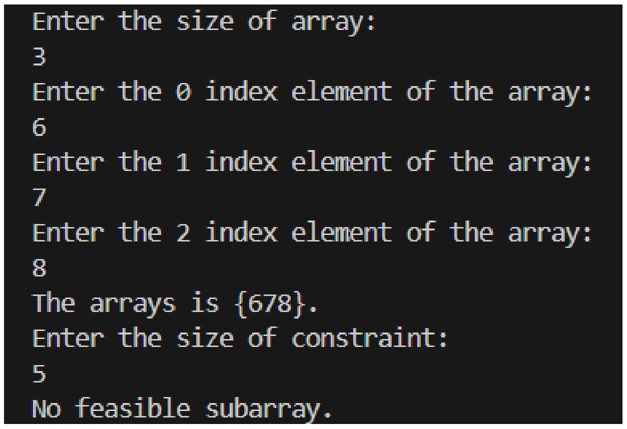
Problem Statement 4:

All elements smaller but no combination fits

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

No feasible subarray.

Tests "no solution" case.



```
Enter the size of array:
3
Enter the 0 index element of the array:
6
Enter the 1 index element of the array:
7
Enter the 2 index element of the array:
8
The arrays is {678}.
Enter the size of constraint:
5
No feasible subarray.
```

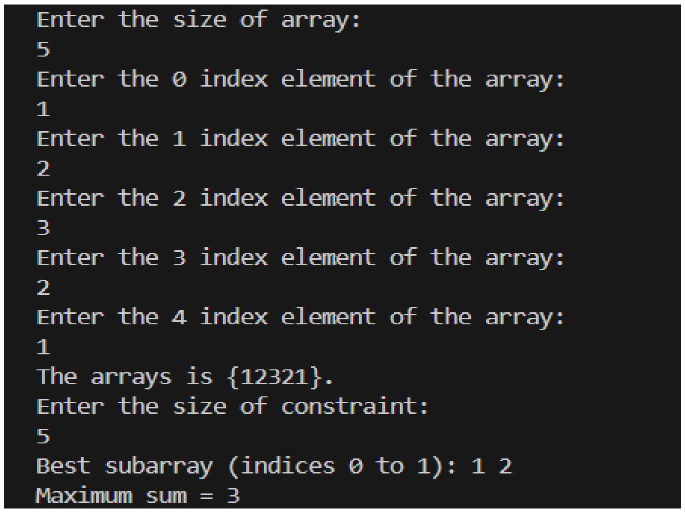
Problem Statement 5:

Multiple optimal subarrays

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

Best subarrays: [2, 3] and [3, 2] → sum = 5

Tests tie-breaking (should return either valid subarray).



```
Enter the size of array:
5
Enter the 0 index element of the array:
1
Enter the 1 index element of the array:
2
Enter the 2 index element of the array:
3
Enter the 3 index element of the array:
2
Enter the 4 index element of the array:
1
The arrays is {12321}.
Enter the size of constraint:
5
Best subarray (indices 0 to 1): 1 2
Maximum sum = 3
```

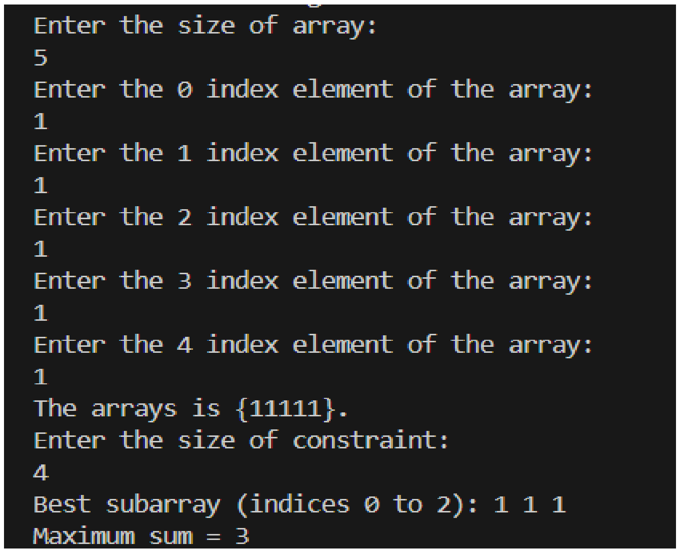
Problem Statement 6:

Large window valid

resources = [1, 1, 1, 1, 1], constraint = 4

Best subarray: [1, 1, 1, 1] → sum = 4

Ensures long window works.



```
Enter the size of array:
5
Enter the 0 index element of the array:
1
Enter the 1 index element of the array:
1
Enter the 2 index element of the array:
1
Enter the 3 index element of the array:
1
Enter the 4 index element of the array:
1
The arrays is {11111}.
Enter the size of constraint:
4
Best subarray (indices 0 to 2): 1 1 1
Maximum sum = 3
```

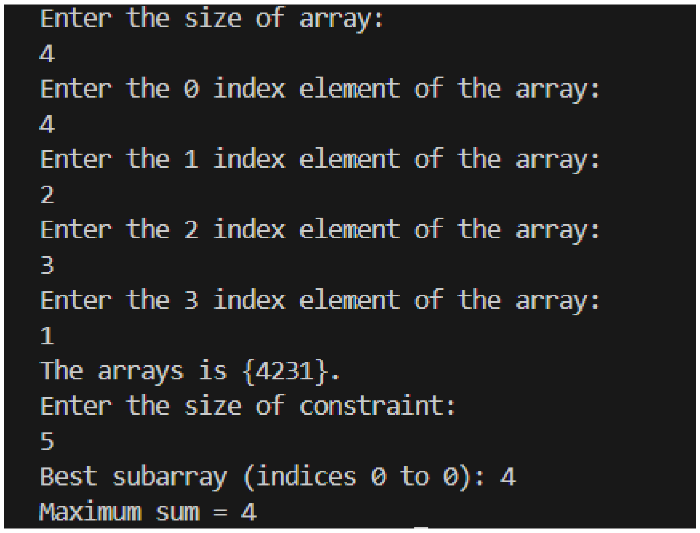
Problem Statement 7:

Sliding window shrink needed

resources = [4, 2, 3, 1], constraint = 5

Start [4,2] = 6 (too big) → shrink to [2,3] = 5.

Tests dynamic window adjustment.



```
Enter the size of array:
4
Enter the 0 index element of the array:
4
Enter the 1 index element of the array:
2
Enter the 2 index element of the array:
3
Enter the 3 index element of the array:
1
The arrays is {4231}.
Enter the size of constraint:
5
Best subarray (indices 0 to 0): 4
Maximum sum = 4
```

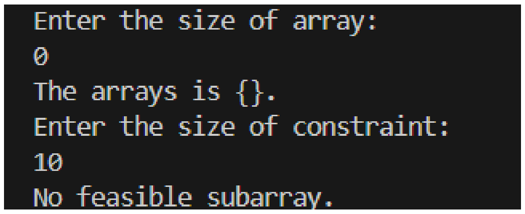
Problem Statement 8:

Empty array

resources = [], constraint = 10

Output: no subarray.

Edge case: empty input.



```
Enter the size of array:
0
The arrays is {}.
Enter the size of constraint:
10
No feasible subarray.
```

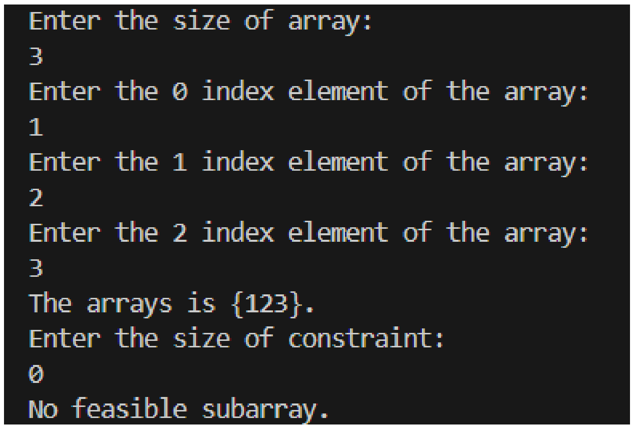
Problem Statement 9:

Constraint = 0

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

No subarray possible.

Edge case: zero constraint.



```
Enter the size of array:
3
Enter the 0 index element of the array:
1
Enter the 1 index element of the array:
2
Enter the 2 index element of the array:
3
The arrays is {123}.
Enter the size of constraint:
0
No feasible subarray.
```

Problem Statement 10:

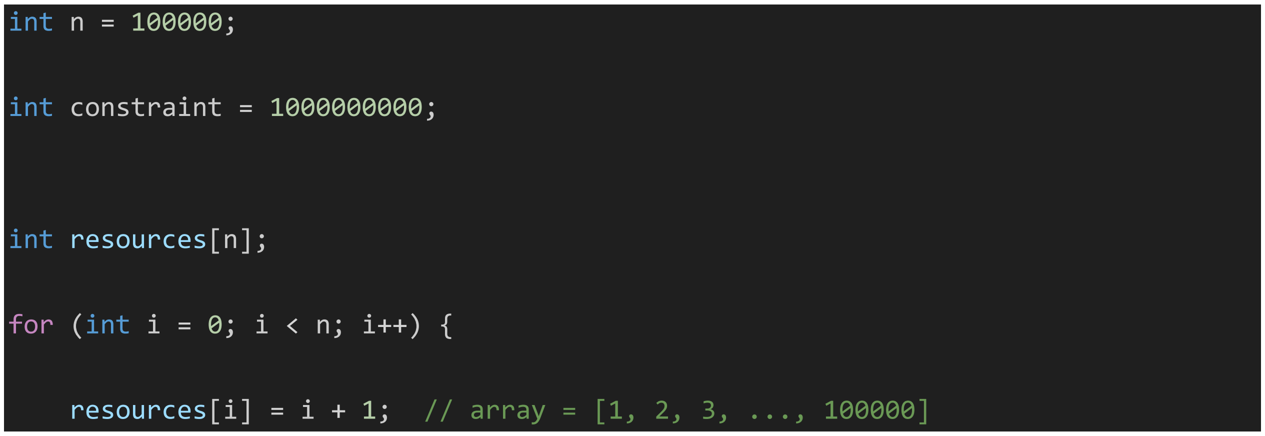
Very large input (stress test)

resources = [1, 2, 3, ..., 100000], constraint = 10^9

Valid subarray near full array.

Performance test.

:Replacing main() with this code because the input will be very big-



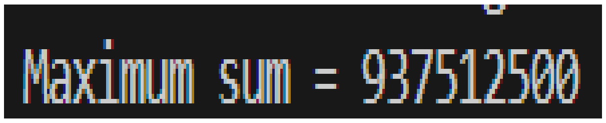
```
int n = 100000;

int constraint = 1000000000;


int resources[n];

for (int i = 0; i < n; i++) {

    resources[i] = i + 1;  // array = [1, 2, 3, ..., 100000]
```

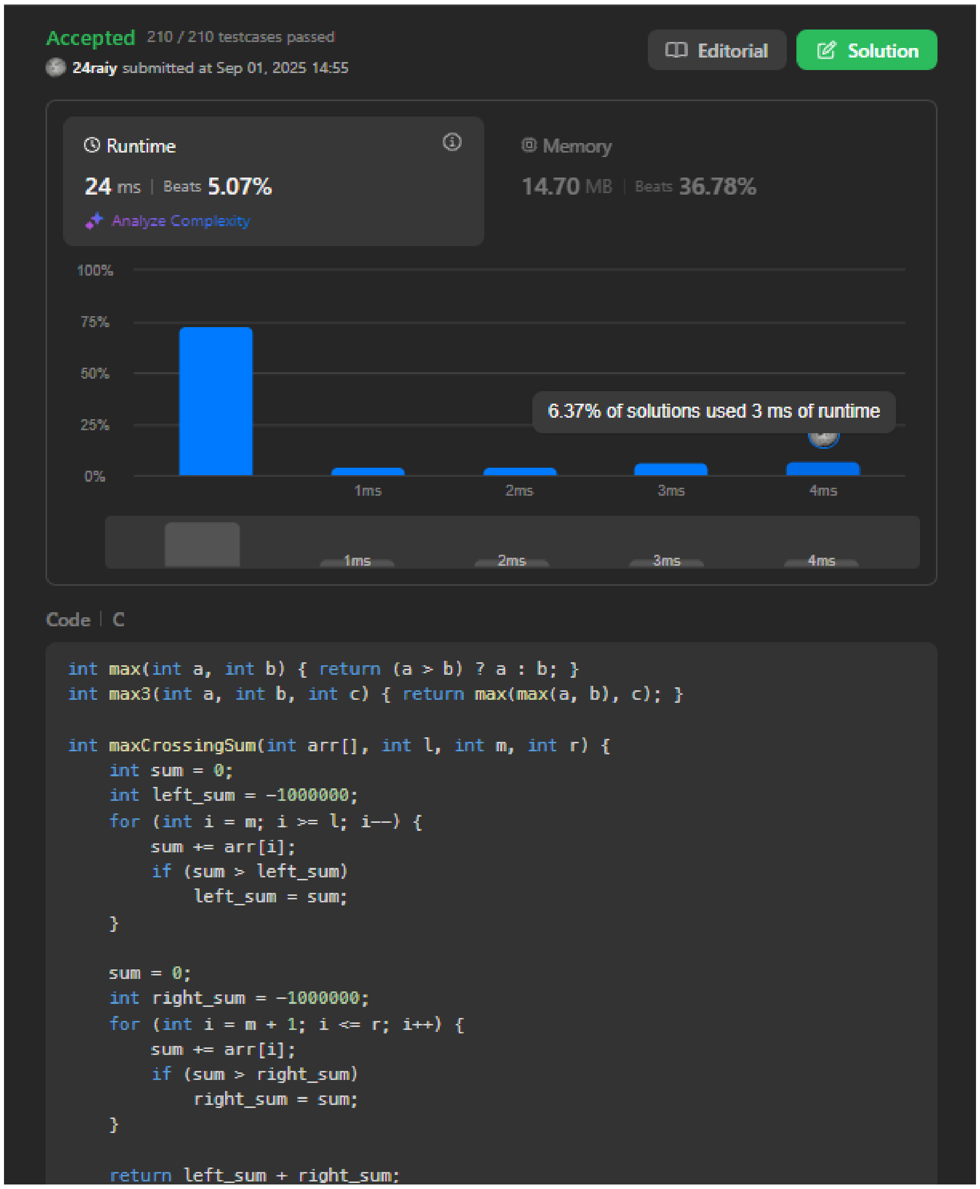


```
Maximum sum = 937512500
```

Conclusion:

1.The program uses the Divide and Conquer method to solve the maximum subarray sum problem with a constraint.
2.The array is recursively divided into two halves until base cases (single elements) are reached.
3.At each step, the algorithm computes three possibilities: maximum subarray in the left half, right half, and a subarray crossing the midpoint.
4.The best among these three is selected using a comparison function.
5.This recursive process ensures that all possible subarrays are considered without brute-force checking.
6.The approach demonstrates how complex problems can be broken into smaller subproblems and then combined for the final solution.
7.It highlights the systematic nature of divide and conquer in solving optimization problems effectively.

Leetcode:

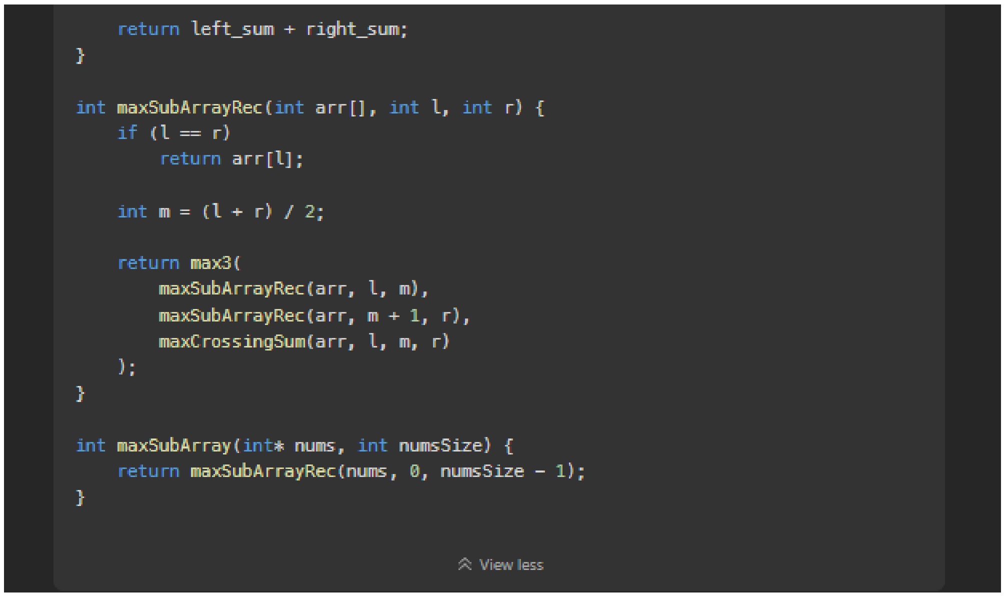


```c
int max(int a, int b) { return (a > b) ? a : b; }
int max3(int a, int b, int c) { return max(max(a, b), c); }

int maxCrossingSum(int arr[], int l, int m, int r) {
    int sum = 0;
    int left_sum = -1000000;
    for (int i = m; i >= l; i--) {
        sum += arr[i];
        if (sum > left_sum)
            left_sum = sum;
    }

    sum = 0;
    int right_sum = -1000000;
    for (int i = m + 1; i <= r; i++) {
        sum += arr[i];
        if (sum > right_sum)
            right_sum = sum;
    }

    return left_sum + right_sum;
```



```
    return left_sum + right_sum;
}

int maxSubArrayRec(int arr[], int l, int r) {
    if (l == r)
        return arr[l];

    int m = (l + r) / 2;

    return max3(
        maxSubArrayRec(arr, l, m),
        maxSubArrayRec(arr, m + 1, r),
        maxCrossingSum(arr, l, m, r)
    );
}

int maxSubArray(int* nums, int numsSize) {
    return maxSubArrayRec(nums, 0, numsSize - 1);
}
```

Link: https://leetcode.com/problems/maximum-subarray/submissions/1755657671/

Github:

https://github.com/volt5123/DAA_pract4/blob/main/pract4.c