Design and Analysis of Algorithms Lab

Name: Arya Bodkhe

Section-A4 Batch: B-1

Roll No.:09

PRACTICAL NO. 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 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.

Divide and conquer approach CODE:

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

typedef struct {
    int sum;
    int start;
    int end;
} SubarrayResult;

SubarrayResult solve(int arr[], int low, int high, int constraint);
```

```
SubarrayResult max_of_three(SubarrayResult a, SubarrayResult b, SubarrayResult c)
{
    if (a.sum >= b.sum && a.sum >= c.sum) {
        return a;
    } else if (b.sum >= a.sum && b.sum >= c.sum) {
        return b;
    } else {
        return c;
    }
}

SubarrayResult findMaxCrossingSubarray(int arr[], int low, int mid, int high, int
constraint) {
    SubarrayResult result = {-1, -1, -1};
    int current_sum = 0;

    int left_sum = -1;
    int best_left_start = mid;
    current_sum = 0;
    for (int i = mid; i >= low; i--) {
        current_sum += arr[i];
        if (current_sum <= constraint && current_sum > left_sum) {
            left_sum = current_sum;
            best_left_start = i;
        }
    }

    int right_sum = -1;
    int best_right_end = mid + 1;
    current_sum = 0;
    for (int i = mid + 1; i <= high; i++) {
        current_sum += arr[i];
        if (current_sum <= constraint && current_sum > right_sum) {
            right_sum = current_sum;
            best_right_end = i;
        }
    }

    if (left_sum > right_sum) {
        result.sum = left_sum;
        result.start = best_left_start;
        result.end = mid;
```

```
    } else if (right_sum != -1) {
        result.sum = right_sum;
        result.start = mid + 1;
        result.end = best_right_end;
    }

    int left_size = mid - low + 1;
    int right_size = high - mid;
    SubarrayResult* left_options = (SubarrayResult*)malloc(left_size *
sizeof(SubarrayResult));
    SubarrayResult* right_options = (SubarrayResult*)malloc(right_size *
sizeof(SubarrayResult));

    current_sum = 0;
    for (int i = mid; i >= low; i--) {
        current_sum += arr[i];
        left_options[mid - i] = (SubarrayResult){current_sum, i, mid};
    }

    current_sum = 0;
    for (int i = mid + 1; i <= high; i++) {
        current_sum += arr[i];
        right_options[i - (mid + 1)] = (SubarrayResult){current_sum, mid + 1, i};
    }

    int i = 0, j = right_size - 1;
    while (i < left_size && j >= 0) {
        long long combined_sum = (long long)left_options[i].sum +
right_options[j].sum;
        if (combined_sum <= constraint) {
            if (combined_sum > result.sum) {
                result.sum = combined_sum;
                result.start = left_options[i].start;
                result.end = right_options[j].end;
            }
            i++;
        } else {
            j--;
        }
    }

    free(left_options);
    free(right_options);
```

```
    return result;
}

SubarrayResult solve(int arr[], int low, int high, int constraint) {
    if (low == high) {
        if (arr[low] <= constraint) {
            return (SubarrayResult){arr[low], low, high};
        } else {
            return (SubarrayResult){-1, -1, -1};
        }
    }

    int mid = low + (high - low) / 2;

    SubarrayResult left_result = solve(arr, low, mid, constraint);
    SubarrayResult right_result = solve(arr, mid + 1, high, constraint);

    SubarrayResult cross_result = findMaxCrossingSubarray(arr, low, mid, high,
constraint);

    return max_of_three(left_result, right_result, cross_result);
}

int main() {
    int n, constraint;
    printf("Enter number of tasks: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("No tasks to process.\n");
        return 0;
    }

    int *resources = (int *)malloc(n * sizeof(int));
    printf("Enter resource array: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &resources[i]);
    }

    printf("Enter resource constraint: ");
    scanf("%d", &constraint);
```

```
    if (constraint <= 0) {
        printf("No feasible subarray with non-positive constraint.\n");
        free(resources);
        return 0;
    }

    SubarrayResult final_result = solve(resources, 0, n - 1, constraint);

    if (final_result.sum == -1) {
        printf("No feasible subarray found.\n");
    } else {
        printf("Maximum subarray sum = %d\n", final_result.sum);
        printf("Subarray: [ ");
        for (int i = final_result.start; i <= final_result.end; i++) {
            printf("%d ", resources[i]);
        }
        printf("]\n");
    }

    free(resources);
    return 0;
}
```

CODE (for all test cases):

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

int main() {
    int n, constraint;
    printf("Enter number of tasks: ");
    scanf("%d", &n);

    int *resources = (int *)malloc(n * sizeof(int));
    printf("Enter resource array: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &resources[i]);
    }

    printf("Enter resource constraint: ");
    scanf("%d", &constraint);
```

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

    int start = 0, best_start = -1, best_end = -1;
    int current_sum = 0, max_sum = -1;

    for (int end = 0; end < n; end++) {
        current_sum += resources[end];

        while (current_sum > constraint && start <= end) {
            current_sum -= resources[start];
            start++;
        }

        if (current_sum <= constraint && current_sum > max_sum) {
            max_sum = current_sum;
            best_start = start;
            best_end = end;
        }
    }

    if (max_sum == -1) {
        printf("No feasible subarray found.\n");
    } else {
        printf("Maximum subarray sum = %d\n", max_sum);
        printf("Subarray: [ ");
        for (int i = best_start; i <= best_end; i++) {
            printf("%d ", resources[i]);
        }
        printf("]\n");
    }

    free(resources);
    return 0;
}
```

OUTPUT:

Case 1: Basic small array

```
Enter number of tasks: 4
Enter resource array: 2 1 3 4
Enter resource constraint: 5
Maximum subarray sum = 4
Subarray: [ 1 3 ]
```

Case 2: Exact match to constraint

```
Enter number of tasks: 4
Enter resource array: 2 2 2 2
Enter resource constraint: 4
Maximum subarray sum = 4
Subarray: [ 2 2 ]
```

Case 3: Single element equals constraint

```
Enter number of tasks: 4
Enter resource array: 1 5 2 3
Enter resource constraint: 5
Maximum subarray sum = 5
Subarray: [ 5 ]
```

Case 4: All elements smaller but no combination fits

```
Enter number of tasks: 3
Enter resource array: 6 7 8
Enter resource constraint: 5
No feasible subarray found.
```

Case-5: Multiple optimal subarrays

```
Enter number of tasks: 5
Enter resource array: 1 2 3 2 1
Enter resource constraint: 5
Maximum subarray sum = 5
Subarray: [ 2 3 ]
```

Case 6: Large window valid

```
Enter number of tasks: 5
Enter resource array: 1 1 1 1 1
Enter resource constraint: 4
Maximum subarray sum = 4
Subarray: [ 1 1 1 1 ]
```

Case-7: Sliding window shrink needed

```
Enter number of tasks: 4
Enter resource array: 4 2 3 1
Enter resource constraint: 5
Maximum subarray sum = 5
Subarray: [ 2 3 ]
```

Case 8: Empty array

```
Enter number of tasks: 0
No tasks to process.
```

Case 9: Constraint = 0

```
Enter number of tasks: 3
Enter resource array: 1 2 3
Enter resource constraint: 0
No feasible subarray with non-positive constraint.
```

Basic code for finding maximum sum subarray

CODE:

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

```c
#include <limits.h>

void findMaxSubarray(int array[], int arraySize) {
    if (arraySize <= 0) {
        printf("Array is empty!\n");
        return;
    }

    int maxSoFar = INT_MIN;
    int bestStartIndex = 0;
    int bestEndIndex = 0;

    int currentMax = 0;
    int currentStartIndex = 0;

    for (int i = 0; i < arraySize; i++) {
        currentMax = currentMax + array[i];

        if (currentMax > maxSoFar) {
            maxSoFar = currentMax;
            bestStartIndex = currentStartIndex;
            bestEndIndex = i;
        }

        if (currentMax < 0) {
            currentMax = 0;
            currentStartIndex = i + 1;
        }
    }

    int resultSize = bestEndIndex - bestStartIndex + 1;

    printf("Maximum subarray sum is: %d\n", maxSoFar);
    printf("Size of that subarray is: %d\n", resultSize);
    printf("Subarray elements are: { ");

    for (int i = bestStartIndex; i <= bestEndIndex; i++) {
        printf("%d", array[i]);
        if (i < bestEndIndex) {
            printf(", ");
        }
    }
```

```
    printf(" }\n");
}

int main() {
    int testArray[] = {-2, 5, 6, -2, -3, 1, 5, -6};
    int arraySize = sizeof(testArray) / sizeof(testArray[0]);

    findMaxSubarray(testArray, arraySize);

    return 0;
}
```

OUTPUT:

```
PS C:\Users\Sarthak\Desktop\Arya_GDG> cd "c:\Users\Sarthak\Desktop\Arya_GDG
Maximum subarray sum is: 12
Size of that subarray is: 6
Subarray elements are: { 5, 6, -2, -3, 1, 5 }
PS C:\Users\Sarthak\Desktop\Arya_GDG>
```