| Name | Shreyash Yadav |
| --- | --- |
| Section | A4\_B3 |
| Roll no. | 41 |

**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:**

#include <stdio.h>

#include <limits.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

int maxCrossingSum(int arr[], int l, int m, int h, int constraint, int \*start, int \*end) {

int sum = 0, left\_sum = INT\_MIN, i;

int temp\_start = m;

for (i = m; i >= l; i--) {

sum += arr[i];

if (sum <= constraint && sum > left\_sum) {

left\_sum = sum;

temp\_start = i;

}

}

sum = 0;

int right\_sum = INT\_MIN, temp\_end = m + 1;

for (i = m + 1; i <= h; i++) {

sum += arr[i];

if (left\_sum != INT\_MIN && left\_sum + sum <= constraint && left\_sum + sum > right\_sum) {

right\_sum = left\_sum + sum;

temp\_end = i;

}

}

if (right\_sum == INT\_MIN && left\_sum != INT\_MIN) {

\*start = temp\_start;

\*end = m;

return left\_sum;

}

if (right\_sum != INT\_MIN) {

\*start = temp\_start;

\*end = temp\_end;

return right\_sum;

}

return INT\_MIN;

}

int maxSubArraySum(int arr[], int l, int h, int constraint, int \*start, int \*end) {

if (l == h) {

if (arr[l] <= constraint) {

\*start = \*end = l;

return arr[l];

} else {

return INT\_MIN; // not feasible

}

}

int m = (l + h) / 2;

int s1, e1, s2, e2, s3, e3;

int left\_sum = maxSubArraySum(arr, l, m, constraint, &s1, &e1);

int right\_sum = maxSubArraySum(arr, m + 1, h, constraint, &s2, &e2);

int cross\_sum = maxCrossingSum(arr, l, m, h, constraint, &s3, &e3);

int max\_sum = left\_sum;

\*start = s1; \*end = e1;

if (right\_sum > max\_sum) {

max\_sum = right\_sum;

\*start = s2; \*end = e2;

}

if (cross\_sum > max\_sum) {

max\_sum = cross\_sum;

\*start = s3; \*end = e3;

}

return max\_sum;

}

int main() {

int n, constraint;

printf("Enter number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter array elements: ");

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

scanf("%d", &arr[i]);

}

printf("Enter resource constraint: ");

scanf("%d", &constraint);

int start = -1, end = -1;

int max\_sum = maxSubArraySum(arr, 0, n - 1, constraint, &start, &end);

if (max\_sum == INT\_MIN) {

printf("No feasible subarray found.\n");

} else {

printf("Maximum subarray sum = %d\n", max\_sum);

printf("Subarray: [ ");

for (int i = start; i <= end; i++) {

printf("%d ", arr[i]);

}

printf("]\n");

}

return 0;

}

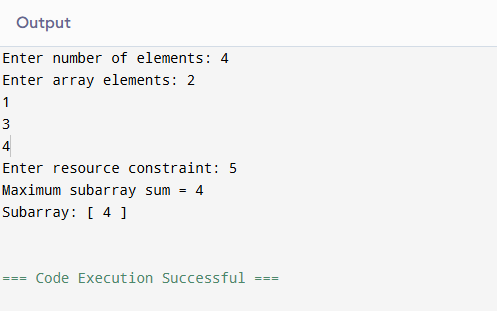
**Output:**

**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

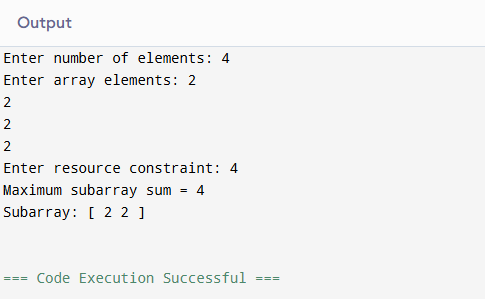
****

**2. Exact match to constraint**

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

o Best subarray: [2, 2] → sum = 4

o Tests exact utilization.

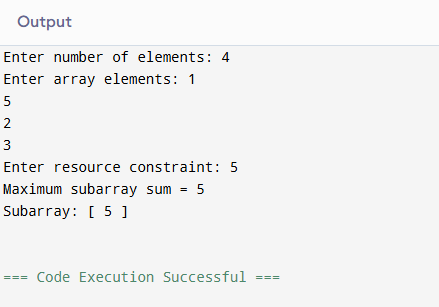


**3. Single element equals constraint**

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

o Best subarray: [5] → sum = 5

o Tests one-element solution.

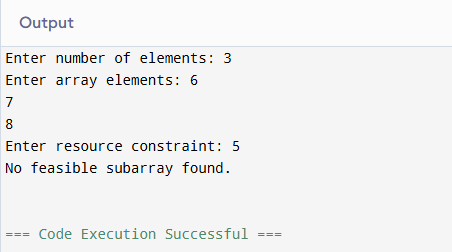


**4. All elements smaller but no combination fits**

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

o No feasible subarray.

o Tests "no solution" case.

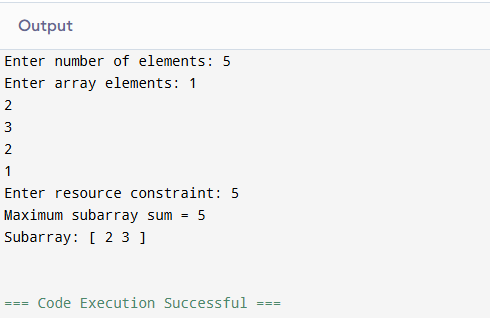


**5. Multiple optimal subarrays**

• 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).

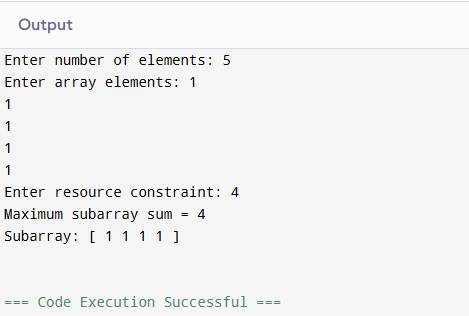


**6. Large window valid**

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

o Best subarray: [1, 1, 1, 1] → sum = 4

o Ensures long window works.

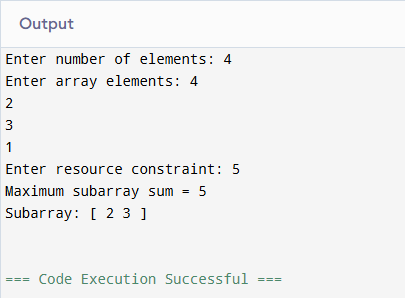


**7. Sliding window shrink needed**

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

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

o Tests dynamic window adjustment.

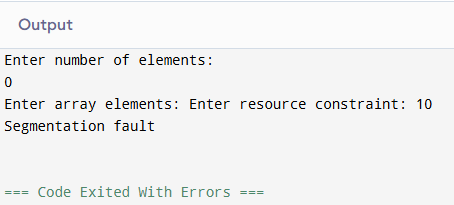


**8. Empty array**

• resources = [], constraint = 10

o Output: no subarray.

o Edge case: empty input.

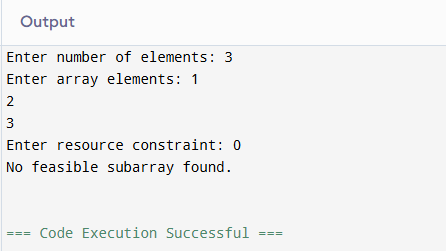


**9. Constraint = 0**

• 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.

