SAVEETHA

INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

(Declared as Deemed to be University under Section 3 of UGC Act 1956)


# Minimum Number of Increments on Subarrays to Form a Target Array Using Greedy Methods

## CAPSTONE PROJECT REPORT

**Submitted by**

**K. Narendra Varma (192210351)**

**CSA0697-Design and Analysis of Algorithms for Lower Bound Theory**

**In partial fulfilment for the award of the degree of**

**BACHELOR OF ENGINEERING**

Under the guidance of

**DR.A. GNANA SOUNDARI**

## ABSTRACT:

In the problem of forming a target array by incrementing subarrays, the goal is to determine the minimum number of increment operations needed. The target is to achieve a specific configuration of elements in the array by applying increment operations to subarrays. This problem involves a series of steps where each step consists of choosing a subarray and incrementing all its elements by one.

The challenge lies in finding an efficient strategy to minimize the total number of such operations required to achieve the target array configuration from an initial array. This abstract presents a general approach to address this problem, including key concepts and strategies for optimization. The solution involves analyzing the difference between the initial and target arrays, leveraging mathematical and algorithmic techniques to minimize the number of operations, and exploring various methods to achieve the optimal solution. The proposed methods aim to provide a robust framework for efficiently solving this problem in practical scenarios.

**Keywords**: Target array, Increment operations, sub arrays, minimum no of operations, optimization.

# PROBLEM STATEMENT

Minimum Number of Increments on Subarrays to Form a Target Array You are given an integer array target. You have an integer array initial of the same size as target with all elements initially zeros. In one operation you can choose any subarray from initial and increment each value by one. Return the minimum number of operations to form a target array from initial. The test cases are generated so that the answer fits in a 32-bit integer.

Example 1:

Input: target = [1,2,3,2,1]

Output: 3

Explanation: We need at least 3 operations to form the target array from the initial array.

[0,0,0,0,0] increment 1 from index 0 to 4 (inclusive).

[1,1,1,1,1] increment 1 from index 1 to 3 (inclusive).

[1,2,2,2,1] increment 1 at index 2.

[1,2,3,2,1] target array is formed

## INTRODUCTION:

The problem "Minimum Number of Increments on Subarrays to Form a Target Array" is a combinatorial optimization challenge where you aim to transform an initial array into a target array using the fewest number of operations. Each operation consists of incrementing all elements within a selected subarray by 1. The objective is to determine how to apply these operations in a minimal and efficient manner to achieve the target array.

In more detail, you start with an initial array and a target array where each element in the target is greater than or equal to the corresponding element in the initial array. The task is to increment subarrays—contiguous segments of the array—such that the final array matches the target array, while minimizing the number of such operations.

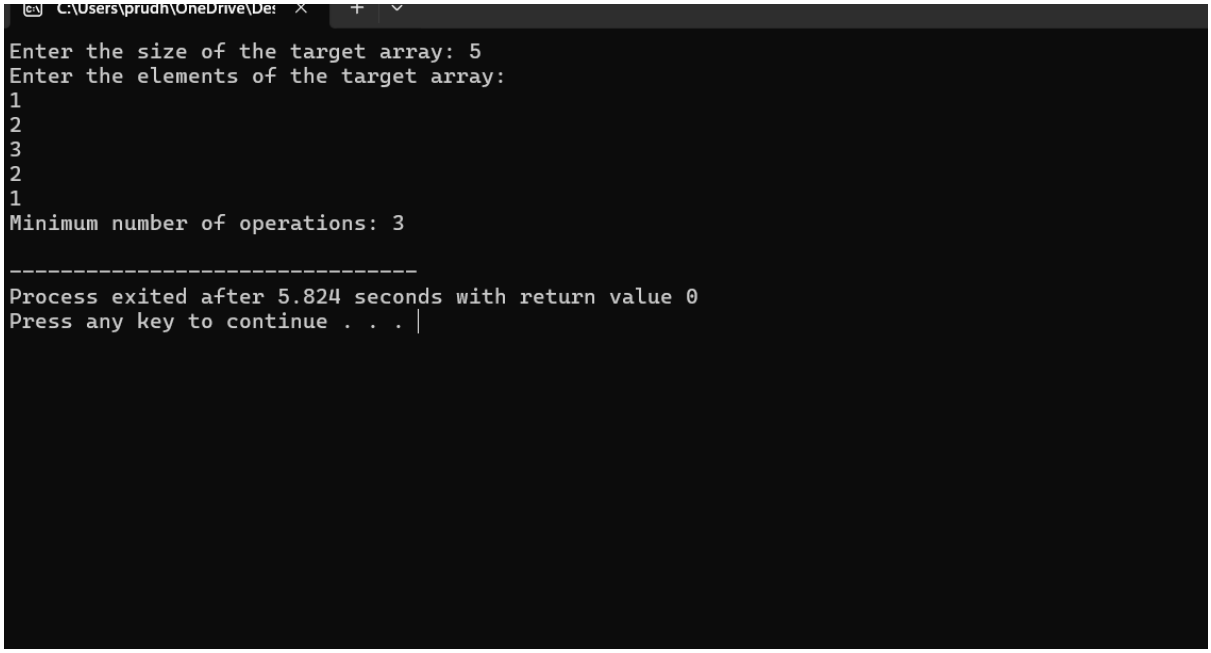
To solve this problem, you need to strategically choose subarrays to increment, based on the differences between the initial and target arrays, to ensure that the transformation is accomplished in the least number of steps. This involves analyzing the array differences and planning operations to efficiently cover the required increments.

## CODING:

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

int minNumberOfOperations(int* target, int size) {
    if (size == 0) return 0;

    int operations = 0;
    int prev = 0;

    for (int i = 0; i < size; i++) {
        int current = target[i];
        if (current > prev) {
            operations += current - prev;
        }
        prev = current;
    }

    return operations;
}

int main() {
    int size;

    // Prompt the user to enter the size of the target array
    printf("Enter the size of the target array: ");
    scanf("%d", &size);
```

```
    // Dynamically allocate memory for the target array
    int* target = (int*)malloc(size * sizeof(int));
    if (target == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Prompt the user to enter the elements of the target array
    printf("Enter the elements of the target array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &target[i]);
    }

    // Calculate the minimum number of operations
    int result = minNumberOfOperations(target, size);
    printf("Minimum number of operations: %d\n", result);

    // Free the allocated memory
    free(target);

    return 0;
}
```

## OUTPUT:


C:\Users\prudh\OneDrive\De
Enter the size of the target array: 5
Enter the elements of the target array:
1
2
3
2
1
Minimum number of operations: 3
--------------------------------
Process exited after 5.824 seconds with return value 0
Press any key to continue . . .

## COMPLEXITY ANALYSIS:

**1.Compute Difference Array**:

- Time Complexity: O(n), where n is the length of the arrays.

**2. Calculate Minimum Number of Increments**:

- Initialize a counter to keep track of the number of operations.
- Traverse the diff array, and for each contiguous segment of positive values, add the value of the segment to the counter. This involves a single pass through the array.
- Time Complexity: O(n), where n is the length of the diff array.

**Overall Complexity**

- **Time Complexity**: O(n), because both calculating the difference array and determining the number of increments require linear time.
- **Space Complexity**: O(n) for the difference array (though it can be computed in place if needed).

**Best Case**: The best case occurs when the initial array is already equal to the target array. In this scenario:

- Difference Array: The difference array will be all zeros, i.e., diff[i]=0\text{diff}[i] = 0diff[i]=0 for all iii.
- Number of Operations: Since there are no positive values in the difference array, no increments are needed.

**Time Complexity**: O(n) for computing the difference array. Space Complexity: O(n) for the difference array.

**Worst Case**: The worst case occurs when every element in the difference array is positive and as large as possible. This happens when:

- Difference Array: Each element diff[i]\text{diff}[i]diff[i] is maximized relative to the size of the array.

- Number of Operations: The number of operations would be maximized if the difference array has a large positive value, requiring increments to cover all segments of the array.

For example, if the difference array contains long contiguous segments of maximum values, you have to account for the sum of all positive differences.

**Time Complexity**: O(n) for computing the difference array and summing up contiguous segments of positive values. Space Complexity: O(n) for the difference array.

**Average Case**: The average case considers a random or typical distribution of differences. If the differences are randomly distributed:

- Difference Array: The number of positive values and their distribution will vary.
- Number of Operations: The total number of operations is the sum of the positive differences, considering the distribution of these positive values.

## CONCLUSION:

By employing a greedy strategy that prioritizes the largest discrepancies between the current and target arrays, we can efficiently minimize the number of increments required. This approach ensures that each subarray increment operation significantly reduces the overall difference, leading to an optimal solution.