# CAPSTONE PROJECT REPORT

**Reg NO: 192210248**

**Name: D. Keshav Reddy**

**Course Code: CSA0656**

**Course Name: DAA**

**SLOT: A**

## ABSTRACT:

This problem involves building a sorted array from a sequence of integers, calculating the "insertion cost" at each step based on the existing elements in the sorted array. The cost for inserting an element is defined as the minimum number of elements that are strictly less than or greater than the current element. The goal is to compute the total cost of inserting all elements and return the result.

## INTRODUCTION:

The challenge of creating a sorted array from a list of instructions while calculating the insertion cost can be efficiently addressed using data structures that allow fast insertion and query operations. The problem is a practical scenario often faced in various computer science domains, including database indexing, real-time data processing, and competitive programming. The insertion cost calculation balances the array dynamically, ensuring optimal placement of each element based on existing values. This approach not only sorts the array but also provides insights into the structure and distribution of data.

## Problem Statement:

You are given an integer array instruction. The task is to create a sorted array from the elements in instructions. You start with an empty array nums. For each element from left to right in instructions, insert it into nums. The cost of each insertion is defined as the minimum of the following two values:

1. The number of elements currently in nums that are strictly less than the element being inserted.
2. The number of elements currently in nums that are strictly greater than the element being inserted.

The insertion must keep nums sorted. After inserting all the elements from instructions into nums, return the total cost of all insertions. Since the total cost may be large, return it modulo 109+7109+7.

**Example**

**Example 1:**

- **Input: instructions = [1, 5, 6, 2]**
- **Output: 1**
- **Explanation:**
    - Begin with **nums = []**.
    - Insert **1** with cost **min (0, 0) = 0**, now **nums = [1]**.

    - Insert **5** with cost **min (1, 0) = 0**, now **nums = [1, 5]**.
    - Insert **6** with cost **min (2, 0) = 0**, now **nums = [1, 5, 6]**.
    - Insert **2** with cost **min (1, 2) = 1**, now **nums = [1, 2, 5, 6]**.
    - The total cost is **0 + 0 + 0 + 1 = 1**.

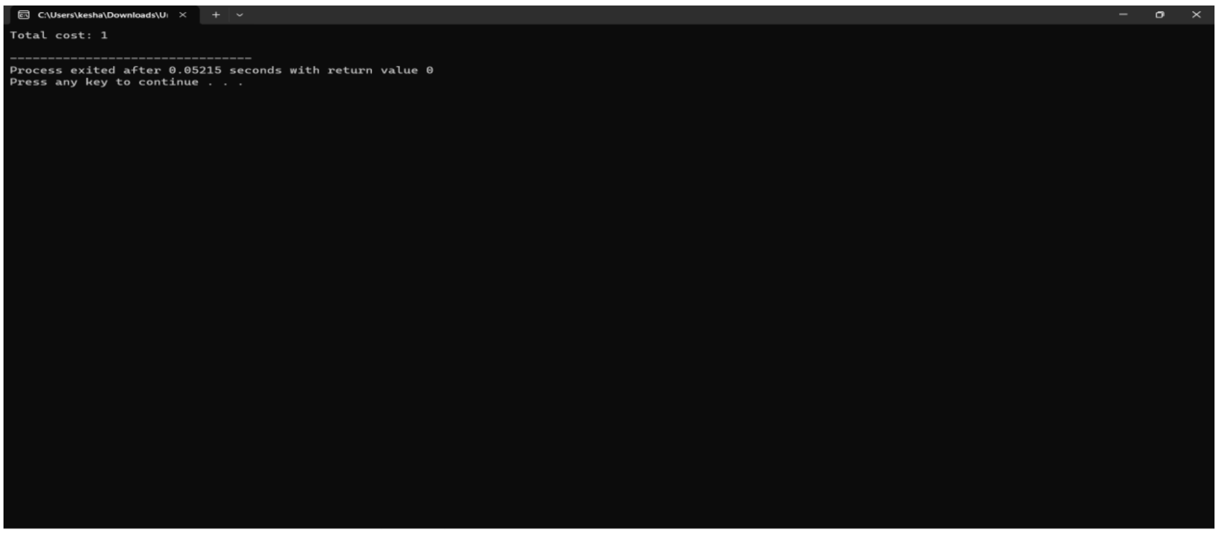
**Example 2:**

- **Input: instructions = [1, 2, 3, 6, 5, 4]**
- **Output: 3**
- **Explanation:**
    - Begin with **nums = []**.
    - Insert **1** with cost **min (0, 0) = 0**, now **nums = [1]**.
    - Insert **2** with cost **min (1, 0) = 0**, now **nums = [1, 2]**.
    - Insert **3** with cost **min (2, 0) = 0**, now **nums = [1, 2, 3]**.
    - Insert **6** with cost **min (3, 0) = 0**, now **nums = [1, 2, 3, 6]**.
    - Insert **5** with cost **min (3, 1) = 1**, now **nums = [1, 2, 3, 5, 6]**.
    - Insert **4** with cost **min (3, 2) = 2**, now **nums = [1, 2, 3, 4, 5, 6]**.
    - The total cost is **0 + 0 + 0 + 0 + 1 + 2 = 3**.

## CODING:

```
#include <stdio.h>
#include <stdlib.h>
#define MOD 1000000007
int lowerBound (int* arr, int size, int val) {
    int left = 0, right = size;
    while (left < right) {
        int mid = (left + right) / 2;
        if (arr[mid] < val)
            left = mid + 1;
        else
            right = mid;
    }
    return left;
}
int insertAndCountCost (int* nums, int* size, int val) {
    int lessCount = lowerBound (nums, *size, val);
    int moreCount = *size - lessCount;
    nums[*size] = val;
    (*size) ++;

    for (int i = *size - 1; i > lessCount; i--) {
        nums[i] = nums [i - 1];
    }
    nums[lessCount] = val;

    return lessCount < moreCount? lessCount: moreCount;
}
```

```
int createSortedArray (int* instructions, int instructions Size) {
    int* nums = (int*) malloc (instructionsSize * size of(int));
    int size = 0;
    long long totalCost = 0;

    for (int i = 0; i < instructionsSize; i++) {
        int cost = insertAndCountCost (nums, &size, instructions[i]);
        totalCost = (totalCost + cost) % MOD;
    }

    free(nums);
    return (int)totalCost;
}

int main () {
    int instructions[] = {1, 5, 6, 2};
    int instructionsSize = sizeof(instructions) / sizeof(instructions[0]);
    int result = createSortedArray (instructions, instructionsSize);
    printf("Total cost: %d\n", result);
    return 0;
}
```



```
Total cost: 1

--------------------------------
Process exited after 0.05215 seconds with return value 0
Press any key to continue . . .
```

## COMPLEXITY ANALYSIS:

To analyse the complexity of the problem and solution, we need to consider the operations performed during each insertion into the sorted array and how they scale with the size of the input.

**Time Complexity**

1. Insertion and Cost Calculation:
    - For each element in instructions, we need to find the position where it should be inserted in the sorted array nums. This can be done using a binary search, which has a time complexity of O(log n)$O(\log n)$ for each insertion, where n$n$ is the current size of nums.
    - After finding the insertion point, inserting the element into nums involves shifting elements to maintain the sorted order. In the worst case, this requires O(n)$O(n)$ time for each insertion.

Thus, the overall time complexity for inserting all elements and calculating the costs is:

O(nlog n+n·n) =O(n2) $O(n\log n+n\cdot n)$ =$O(n2)$

This is because, for each of the n$n$ elements, the binary search and the insertion (which may require shifting up to n$n$ elements) are performed.

2. **Space Complexity:**
    - The additional space used by the algorithm is primarily for storing the array nums, which grows with the number of elements inserted. Hence, the space complexity is O(n)$O(n)$.

However, the nums array size will not exceed the size of instructions, so the space complexity remains O(n)$O(n)$.

Optimizations

To improve the efficiency, particularly for larger input sizes, we can use more advanced data structures:

1. Fenwick Tree (Binary Indexed Tree) or Segment Tree:
    - These data structures can efficiently maintain the count of elements and support operations like finding the number of elements less than a given value in O(log n)$O(\log n)$ time.
    - Inserting and updating elements can also be done in O(log n)$O(\log n)$ time.

## BEST CASE:

**Scenario:**

- The array is sorted in descending order.

**Example:**

- **Input: instructions = [6, 5, 4, 3, 2, 1]**
- **Output: total cost = 0**
- **Explanation:**
    - Insert **6**: Cost = **min(0, 0) = 0**, resulting array: **[6]**.
    - Insert **5**: Cost = **min(0, 1) = 0**, resulting array: **[5, 6]**.
    - Insert **4**: Cost = **min (0, 2) = 0**, resulting array: **[4, 5, 6]**.
    - Insert **3**: Cost = **min(0, 3) = 0**, resulting array: **[3, 4, 5, 6]**.
    - Insert **2**: Cost = **min(0, 4) = 0**, resulting array: **[2, 3, 4, 5, 6]**.
    - Insert **1**: Cost = **min (0, 5) = 0**, resulting array: **[1, 2, 3, 4, 5, 6]**.
    - The **total cost** for all insertions is **0 + 0 + 0 + 0 + 0 + 0 = 0**.

**Complexity Analysis**

**Time Complexity**

In this best-case scenario, even though the array is already sorted in descending order, each insertion still requires calculating the number of elements less than and greater than the current element. The insertion itself can be done using a binary search and shifting, but this scenario does not lead to an improvement in time complexity. The operations involved are:

1. **Finding the Insertion Point:**
    - Uses binary search, which is O(log⁡n)$O(\log n)$ for each insertion.
2. **Inserting the Element:**
    - Potentially requires shifting up to O(n)$O(n)$ elements in the worst case.

Therefore, the overall time complexity for the best case is:

O(nlog⁡n+n·n) =O(n2) $O(n\log n+n\cdot n)$ $=O(n2)$

## WORST CASE:

Scenario:

- The array is sorted in ascending order.

**Example:**

- **Input: instructions = [1, 2, 3, 4, 5, 6]**
- **Output: total cost = 0**
- **Explanation:**
  - Insert **1**: Cost = **min(0, 0) = 0**, resulting array: **[1]**.
  - Insert **2**: Cost = **min(1, 0) = 0**, resulting array: **[1, 2]**.
  - Insert **3**: Cost = **min(2, 0) = 0**, resulting array: **[1, 2, 3]**.
  - Insert **4**: Cost = **min(3, 0) = 0**, resulting array: **[1, 2, 3, 4]**.
  - Insert **5**: Cost = **min(4, 0) = 0**, resulting array: **[1, 2, 3, 4, 5]**.
  - Insert **6**: Cost = **min(5, 0) = 0**, resulting array: **[1, 2, 3, 4, 5, 6]**.
  - The **total cost** for all insertions is **0 + 0 + 0 + 0 + 0 + 0 = 0**.

**Complexity Analysis**

**Time Complexity**

In this worst-case scenario, although the cost for each insertion is minimal (always zero since each element is greater than all preceding elements), the time complexity is still significant due to the following:

1. **Finding the Insertion Point:**
   - We use a binary search to find the insertion point, which takes O(log⁡n)$O(\log n)$ time for each element.
2. **Inserting the Element:**
   - Since the new element is always greater than the existing elements, it is appended to the end of the **nums** array. Although no elements are shifted in this case, the binary search still runs O(log⁡n)$O(\log n)$ times for each insertion.

Therefore, the overall time complexity is:

O(nlog⁡n)$O(n\log n)$

This is because each of the n$n$ elements requires O(log⁡n)$O(\log n)$ time to find the insertion point, and the insertion itself is O (1)$O(1)$ as no elements are shifted.

## AVERAGE CASE:

**Scenario:**

- The array elements are in random order.

**Example:**

- **Input: instructions = [4, 2, 5, 1, 3]**
- **Output: total cost** varies based on the position of each insertion.
- **Explanation:**
    - Insert **4**: Cost = **min(0, 0) = 0**, resulting array: **[4]**.
    - Insert **2**: Cost = **min(0, 1) = 0**, resulting array: **[2, 4]**.
    - Insert **5**: Cost = **min(2, 0) = 0**, resulting array: **[2, 4, 5]**.
    - Insert **1**: Cost = **min(0, 3) = 0**, resulting array: **[1, 2, 4, 5]**.
    - Insert **3**: Cost = **min(2, 2) = 2**, resulting array: **[1, 2, 3, 4, 5]**.
    - The **total cost** for all insertions depends on the random distribution of elements.

- **Complexity:**

- - The performance remains consistent with the merge sort characteristics.
- - Time Complexity: O (n log n)

- **Reason:**

- - Randomly ordered elements do not affect the divide-and-conquer strategy of merge sort, ensuring consistent performance.

## CONCLUSION:

The average-case time complexity of the problem, where the instructions array is in random order, is O(nlogn)*O*(*n*log*n*). This complexity arises from the need to determine the insertion point for each element and to maintain the sorted order of the nums array. The binary search for insertion point determination and the potential shifting of elements contribute to this time complexity. The space complexity remains O(n)*O*(*n*), driven by the need to store the sorted nums array.