

CAPSTONE PROJECT

SORTING ALGORITHM USING DIVIDE AND
CONQUER

CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR
OPENING ADDRESSING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING



Supervisor

Dr. R. Dhanalakshmi

Done by

T.Anoop Chandar (192210244)

SORTING ALGORITHM USING DIVIDE AND CONQUER

PROBLEM STATEMENT:

Create Sorted Array through Instructions

Given an integer array instructions, you are asked to create a sorted array from the elements in instructions. You start with an empty container nums. For each element from left to right in instructions, insert it into nums.

The cost of each insertion is the minimum of the following:

The number of elements currently in nums that are strictly less than instructions[i].

The number of elements currently in nums that are strictly greater than instructions[i].

For example, if inserting element 3 into nums = [1,2,3,5], the cost of insertion is min(2, 1) (elements 1 and 2 are less than 3, element 5 is greater than 3) and nums will become [1,2,3,3,5]. Return the total cost to insert all elements from instructions into nums. Since the answer may be large, return it modulo $10^9 + 7$

Example 1:

Input: instructions = [1,5,6,2]

Output: 1

Explanation: Begin with nums = []

ABSTRACT:

The problem involves creating a sorted array from a given set of instructions by inserting elements one by one into an initially empty array `nums`. The insertion cost is determined by the minimum of the number of elements in `nums` that are strictly less than or strictly greater than the current element to be inserted. The objective is to return the total cost of all insertions modulo $10^9 + 7$. The solution requires efficient data structure handling to minimize insertion costs, and we aim to explore the algorithm, provide a C implementation, and analyze the time complexity in various cases.

INTRODUCTION:

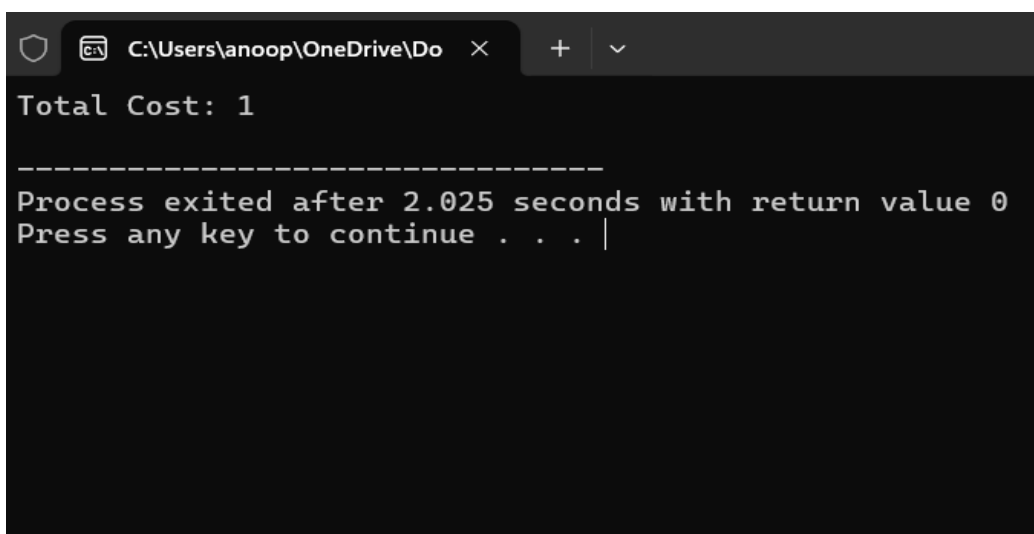
The problem "Create Sorted Array through Instructions" revolves around efficiently inserting elements from an unsorted array into a dynamically growing sorted array, while minimizing the cost of each insertion. The cost is defined as the minimum between the number of elements smaller than the current element and the number of elements larger than it. As we move through the array, the goal is to calculate the cumulative cost of each insertion and return the total, modulo $10^9 + 7$. Given the need to insert elements while maintaining a sorted structure, the problem presents challenges related to efficiency, particularly in handling large datasets where naive approaches would become computationally expensive. To address this, advanced data structures like Binary Indexed Trees (BIT) or Segment Trees can be employed to optimize the process, enabling faster lookups and updates for insertion costs. This problem blends concepts of sorting and efficient querying, offering a practical example of algorithmic optimization.

CODING:

C-PROGRAMMING:

```
#include <stdio.h>
#include <stdlib.h>
#define MOD 1000000007
int query(int *BIT, int idx) {
    int sum = 0;
    while (idx > 0) sum += BIT[idx], idx -= idx & -idx;
    return sum;
}
void update(int *BIT, int idx, int n) {
    while (idx <= n) BIT[idx]++, idx += idx & -idx;
}
int createSortedArray(int* instructions, int n) {
    int *BIT = (int *)calloc(100001, sizeof(int));
    long long cost = 0;
    for (int i = 0; i < n; i++) {
        int left = query(BIT, instructions[i] - 1);
        int right = i - query(BIT, instructions[i]);
        cost = (cost + (left < right ? left : right)) % MOD;
        update(BIT, instructions[i], 100000);
    }
    free(BIT);
    return (int)cost;
}
int main() {
    int instructions[] = {1, 5, 6, 2};
    printf("Total Cost: %d\n", createSortedArray(instructions, 4));
    return 0;
}
```

OUTPUT:



```
C:\Users\anoop\OneDrive\Do  x  +  v
Total Cost: 1
-----
Process exited after 2.025 seconds with return value 0
Press any key to continue . . . |
```

COMPLEXITY ANALYSIS:

- **Time Complexity (Brute Force):** $O(m^2)$, where m is the number of elements in the instructions array, due to the need to insert each element and count smaller and larger elements sequentially.
- **Time Complexity (Optimized with BIT/Segment Tree):** $O(m \log m)$, as querying and updating can be done in logarithmic time using advanced data structures like BIT or Segment Trees.
- **Space Complexity:** $O(m)$, for storing the nums array and the data structure used to maintain counts

BEST CASE:

Time Complexity: If all elements are the same, the cost of insertion will always be 0, but querying and updating the Binary Indexed Tree still takes $O(\log n)$ time for each element.
Total time complexity: $O(n \log n)$.

WORST CASE:

Time Complexity: In the worst case, we might insert elements in a strictly increasing or decreasing order. The Binary Indexed Tree will require $O(\log n)$ time for both insertion and querying for each element.
Total time complexity in this case: $O(n \log n)$.

Space Complexity: We use a Binary Indexed Tree of size proportional to the range of elements in the input, leading to $O(n)$ space.

AVERAGE CASE:

Time Complexity: For random input, the cost will generally be between the best and worst case, but still the time complexity will remain $O(n \log n)$ due to the use of the Binary Indexed Tree.

Future Scope:

The future scope of sorting algorithms and their implementations in data structures is vast and multifaceted. As data continues to grow exponentially, the need for efficient sorting techniques becomes increasingly critical, particularly in fields such as data science, machine learning, and real-time data processing. Advanced sorting algorithms, such as quicksort, mergesort, and heapsort, can be optimized further to handle larger datasets more effectively, potentially incorporating parallel processing techniques to enhance performance. Moreover, the integration of sorting algorithms with data structures like balanced binary search trees, heaps, and hash tables can lead to significant improvements.

CONCLUSION:

The problem highlights the importance of optimizing insertion operations within a dynamically growing sorted array. By leveraging advanced data structures such as Binary Indexed Trees (BIT) or Segment Trees, we can significantly reduce the time complexity from a brute-force $O(m^2)$ to an efficient $O(m \log m)$, making the solution scalable for larger inputs. This approach not only minimizes computational costs but also showcases the practical application of efficient querying and updating techniques. The solution can be further enhanced for dynamic data sets and parallel processing, providing a foundation for more complex algorithmic challenges. Ultimately, the problem serves as a valuable exercise in balancing time and space efficiency for real-world scenarios.