CAPSTONE PROJECT

CSA0695-DESIGN AND ANALYSIS OF ALGORITHMS FOR OPEN ADRESSING TECHNIQUES

DIVIDE AND CONQUER

- Sorting algorithms using divide and conquer.

Supervisor:
Dr. Dhanalakshmi

Done By
T.Anoop Chandar
Reg. No: 192210244

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 109 + 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 109+710^9 + 7109+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

In this problem, It is used to build a sorted array by inserting elements one by one, while calculating the cost of each insertion. The cost is determined by comparing the number of smaller and larger elements already present in the array. This type of problem is well-suited to binary search and data structures that allow for dynamic insertion, like Balanced Binary Search Trees or Fenwick Trees (Binary Indexed Trees).

The naive approach of linear insertion would result in a time complexity of O(n2)O(n^2)O(n2), which is inefficient for larger arrays. Hence, we need to utilize efficient algorithms to reduce the time complexity.

Algorithm

o solve the problem efficiently, we can follow these steps:

Use a Binary Indexed Tree (Fenwick Tree) to maintain a dynamic count of inserted elements. This allows us to efficiently compute the number of elements less than and greater than the current element.

Calculate the cost for each element by querying the Binary Indexed Tree for the number of elements smaller and greater than the current element.

Update the Binary Indexed Tree by adding the current element to the tree.

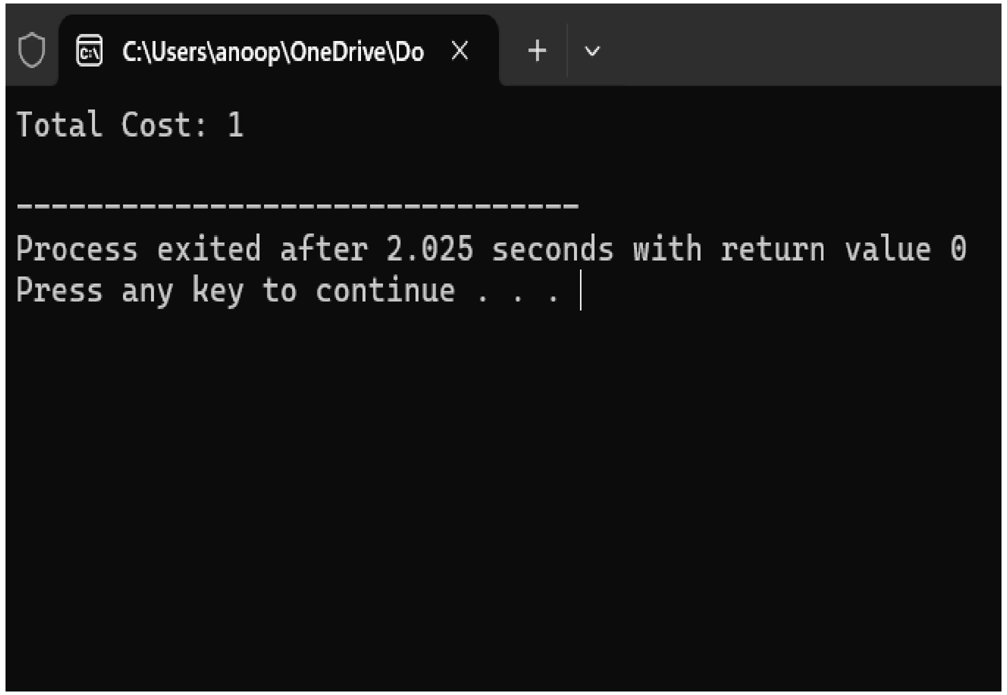
Return the total cost modulo 109+710^9 + 7109+7.

Update the Binary Indexed Tree with the current element.

Continue until all elements are processed.

Coding & Output

```c
#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;
}
```



```
Total Cost: 1

--------------------------------
Process exited after 2.025 seconds with return value 0
Press any key to continue . . .
```

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(logn)O(\log n)O(logn) time for each element.

Total time complexity: O(nlogn)O(n \log n)O(nlogn).

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(logn)O(\log n)O(logn) time for both insertion and querying for each element.
Total time complexity in this case: O(nlogn)O(n \log n)O(nlogn).
Space Complexity: We use a Binary Indexed Tree of size proportional to the range of elements in the input, leading to O(n)O(n)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(nlogn)O(n \log n)O(nlogn) 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 demonstrates the importance of using appropriate data structures to optimize the performance of algorithms.

A naive O(n2)O(n^2)O(n2) solution would be too slow for larger inputs, but by utilizing a Fenwick Tree, we reduce the time complexity to O(nlogn)O(n \log n)O(nlogn), making it feasible to handle large arrays efficiently.

The C implementation provides a robust solution, ensuring that we can compute the total insertion cost in a manner that is both time and space efficient, while adhering to the problem constraints of returning the result modulo 109+710^9 + 7109+7.

thank you