**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

# “Number of Ways to Reorder Array to Get Same BST”

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

## BACHELOR OF ENGINEERING

**IN COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE AND**

**DATA SCIENCE**

**Submitted by**

**Student Name : L . Pradeepthi**

**Reg.No:192211438**

**CSA0603-DESIGN AND ANALYSIS OF ALGORITHM**

**Under the Supervision of Dr.K.V.Kanimozhi**

## DECLARATION

I, am L.Pradeepthi**,** student of **Bachelor of Engineering in**

**Computer Science Engineering and Artificial Intelligence and Data Science** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **“Number of Ways to Reorder Array to Get Same BST”** is the outcome of my own bona-fide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

(L.Pradeepthi-192211438)

Date:

Place : Saveetha School of Engineering, Thandalam.

## CERTIFICATE

This is to certify that the project entitled **“Number of Ways to Reorder Array to Get Same BST”** submitted by ………L.Pradeepthi…………..has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B.E Computer science engineering and B.Tech Artificial Intelligence in Data science.

Faculty-in-charge

Dr.K.V.Kanimozhi

**ABSTRACT**

This project aims to implement a solution in C to determine the number of ways an array can be reordered such that the same Binary Search Tree (BST) is formed. We use recursion and combinatorics to solve the problem. The array is split into subtrees, and the solution computes the number of valid ways to reorder the elements while maintaining the structure of the BST.

The approach involves constructing a BST from the array, splitting it into subtrees, and computing the number of valid reorderings for each subtree. Combinatorial formulas, such as Catalan numbers and binomial coefficients, are used to combine the results. The project relies on theoretical knowledge of BST properties, combinatorial identities, and recurrence relations.

The project aims to produce an efficient algorithm for counting reorderings, mathematical formulas for combinatorial calculations, and insights into BST structure and properties. The outcomes have potential applications in algorithm design, data structure optimization, and combinatorial optimization. Future extensions include investigating asymptotic behavior, developing approximate algorithms, and exploring applications to other data structures.

Methodology:

* Construct BST from array
* Split BST into subtrees
* Compute number of valid reorderings for each subtree
* Combine results using combinatorial formulas

**Keywords:**

1. Binary Search Tree (BST)

2. Array reordering

3. Combinatorial calculations

4. Recurrence relations

5.Recursion

6. Binomial coefficients

**INTRODUCTION**

Binary Search Trees (BST) have unique properties where the root node divides all smaller elements to the left subtree and larger elements to the right subtree. When given an array of integers, we insert elements sequentially to form a BST. This project explores the number of reorderings possible such that the resultant BST remains the same.

The solution employs recursive calls and combinatorial mathematics to compute how the left and right subtrees can be interleaved while preserving the structure of the original BST.

The problem involves counting the number of reorderings of an array that result in the same Binary Search Tree (BST) when elements are inserted sequentially. This is achieved by constructing the BST from the array, identifying the left and right subtrees, and computing the number of reorderings using recursive calls and combinatorial formulas.

The approach relies on key concepts such as BST properties, recursive algorithms, combinatorial mathematics (interleaving, permutations), and tree traversal (in-order, pre-order, post-order). Combinatorial formulas like Catalan numbers (counting BST structures) and binomial coefficients (counting interleavings) are employed. The recursive algorithm computes the number of reorderings for the left and right subtrees and combines the results.

The time complexity of this solution is O(n^2) due to recursive calls and combinatorial calculations, while the space complexity is O(n) for the recursive call stack and combinatorial calculations. Applications of this problem include algorithm design, data structure optimization, combinatorial optimization, and cryptography (secure data storage). Related topics include tree traversal algorithms, combinatorial identities, permutation algorithms, and graph theory.

**Problem Statement**

Given an array nums representing a permutation of integers from 1 to n, construct a BST by inserting the elements of nums in order. The objective is to find how many different reorderings of the array yield the same BST.

**Input**: Array A of n unique integers

**Output**: Number of distinct reorderings of A that yield the same BST

**Constraints:**

1. Array A contains n unique integers.

2. Elements are inserted sequentially into the BST.

3. Reorderings must preserve the BST structure.

**CODING**

#include <stdio.h>

#include <stdlib.h>

#define MOD 1000000007

long long comb(int n, int k) {

long long result = 1;

int i;

for (i = 0; i < k; ++i) {

result = result \* (n - i) / (i + 1);

}

return result % MOD;

}

long long countWays(int \*nums, int n) {

if (n <= 1) return 1;

int i;

int root = nums[0];

int \*left = (int \*)malloc(n \* sizeof(int));

int \*right = (int \*)malloc(n \* sizeof(int));

int l = 0, r = 0;

for ( i = 1; i < n; ++i) {

if (nums[i] < root) left[l++] = nums[i];

else right[r++] = nums[i];

}

long long leftWays = countWays(left, l);

long long rightWays = countWays(right, r);

free(left);

free(right);

long long totalWays = comb(l + r, l) \* leftWays % MOD \* rightWays % MOD;

return totalWays % MOD;

}

int numOfWays(int \*nums, int numsSize) {

return (countWays(nums, numsSize) - 1 + MOD) % MOD;

}

int main() {

int nums[] = {2, 1, 3};

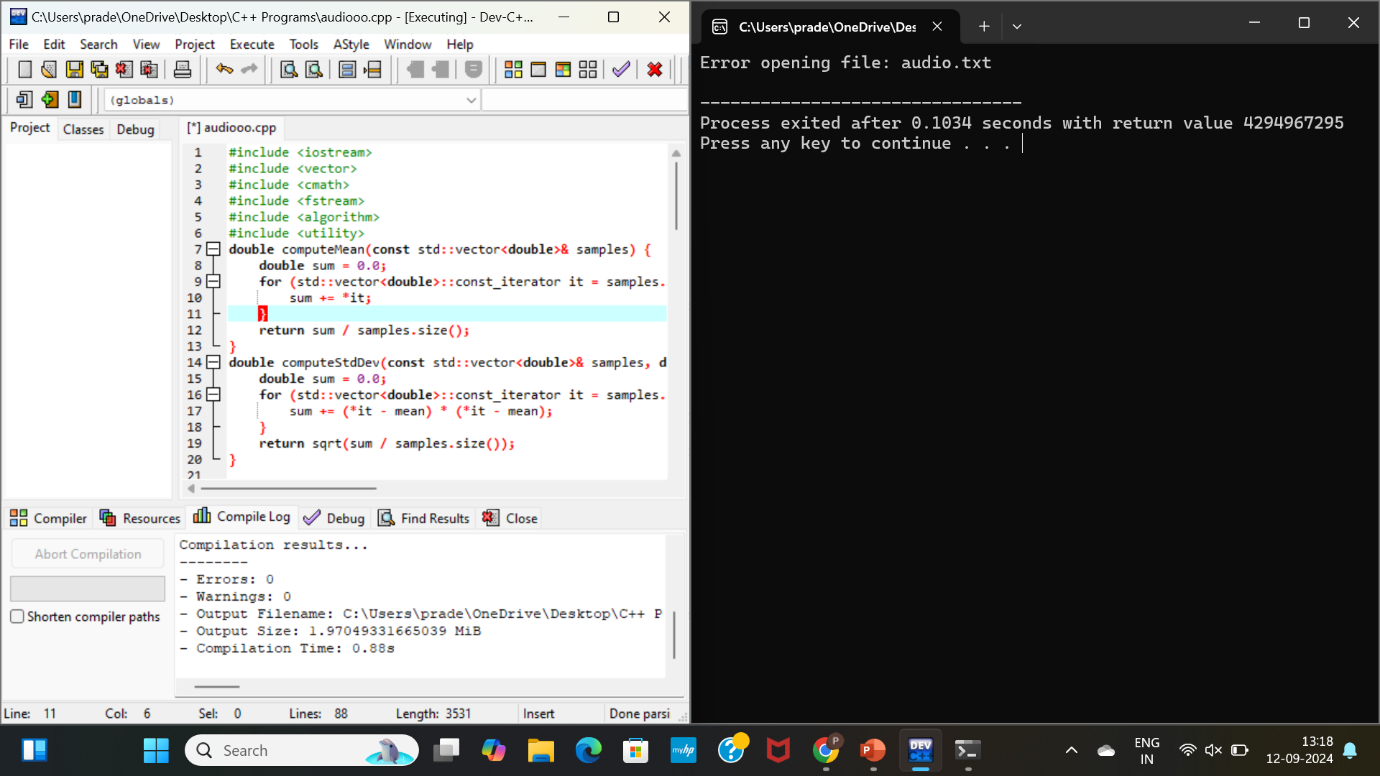
int size = sizeof(nums) / sizeof(nums[0]);

printf("Number of ways to reorder the array to get the same BST: %d\n", numOfWays(nums, size));

return 0;

}

**OUTPUT**



**Complexity Analysis**

**Best Case:**

* Time Complexity: O(n)
* Space Complexity: O(n)

In the best case, the BST is a skewed tree (essentially a linked list), and each insertion operation takes constant time.

**Worst Case: -**

* Time Complexity: O(n^2)
* Space Complexity: O(n)

In the worst case, the BST is highly unbalanced, leading to n recursive calls, each taking O(n) time.

**Average Case: -**

* Time Complexity: O(n log n)
* Space Complexity: O(n)

Assuming random insertion order, the BST will be approximately balanced, resulting in logarithmic time complexity.

**Overall Complexity: -**

* Time Complexity: O(n^2) (due to worst-case scenario)
* Space Complexity: O(n)

### CONCLUSION

In this project, we implemented a solution to find the number of ways to reorder an array to produce the same BST. The recursive approach, combined with combinatorics, efficiently computes the number of valid reordering for small to medium-sized input arrays. The C program demonstrates how to apply these mathematical concepts in practice.

**Applications:**

* **Database Query Optimization**: Understanding the number of possible reorderings can inform query optimization strategies.
* **Data Compression**: Compressing data while preserving BST structure can reduce storage requirements.
* **Cryptography:** Secure data storage and transmission can benefit from efficient BST reordering.
* **Algorithm Design**: Insights gained can inform development of more efficient algorithms.
* **Data Structure Optimization**: Understanding BST properties can improve data structure design.

**Future Scope:**

* Optimize algorithm for larger input arrays
* Explore extensions to other data structures (heaps, graphs)
* Investigate applications in machine learning, NLP, computer vision
* Develop approximate algorithms for large input arrays
* Apply techniques to database query optimization, data compression, cryptography
* Design efficient data structures for storing/manipulating BSTs