A PROJECT REPORT

On

NUMBER OF WAYS TO REORDER ARRAY TO GET SAME BST USING DYNAMIC PROGRAMMING

SUBMITTED TO

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

In partial fulfillment of the award of the course of

CSA0697-DESIGN AND ANALYSIS OF ALGORITHMS FOR LOWER BOUND THEORY

By

J.DHARSHINI(192210257)

SUPERVISOR

Dr. GNANA SOUNDARI

SAVEETHA SCHOOL OF ENGINEERING, SIMATS CHENNAI- 602105

SEPTEMBER-2024

BONAFIDE CERTIFICATE

Certified that this project report titled **"MAXIMUM NUMBER OF NON-OVERLAPPING SUBSTRINGS"** is the bonafide work **LAVANYA R (192210663)** , who carried out the project work under my supervision as a batch. Certified further, that to the best of my knowledge, the work reported here in does not form any other project report.

Project Supervisor

Date:

Head of the Department

Date:

ABSTRACT:

This paper introduces a dynamic programming solution for calculating the number of ways to reorder an array to obtain the same Binary Search Tree (BST) that the original array constructs. Given an array of unique elements, the order in which the elements are inserted into the tree determines its structure.

The challenge is to determine how many different permutations of the original array would result in the exact same BST. The approach leverages dynamic programming to break down the problem into smaller subproblems involving left and right subtrees.

By employing combinatorial mathematics, this method efficiently calculates the number of possible reorderings that yield the same BST. A C++ program is provided to demonstrate the implementation of this solution.

PROBLEM STATEMENT AND ASSUMPTIONS:

Given an array of unique elements, a Binary Search Tree (BST) is constructed by inserting the elements one by one.

The objective is to find the total number of ways to reorder the array such that the resulting BST remains the same.

For example, if the array is `[3, 1, 4, 2]`, we can construct a BST by inserting `3`, followed by `1`, then `4`, and finally `2`. This BST has the same structure regardless of the order of certain permutations.

Input: An array of unique integers.

Output: The number of different permutations of the array that result in the same BST.

Example:

Input: `[3, 1, 2, 5, 4]`

Output: `6`

Explanation:The original array constructs a specific BST, and there are exactly 6 permutations of this array that lead to the same BST structure.

Assumptions:

1. The values in the input array are unique.

2. The tree structure is defined by the order in which elements are inserted.

3. The result must be computed programmatically to ensure efficiency for larger arrays.

INTRODUCTION:

Binary Search Trees (BSTs) play a fundamental role in various computer science applications, such as searching, sorting, and dynamic data storage.

The process of constructing a BST from an array depends on the order of insertion.

Specifically, for each element in the array, values smaller than the element are placed in the left subtree, while values larger are placed in the right subtree.

Given this structure, the problem of finding the number of reorderings of an array that result in the same BST structure can be viewed as a combinatorial problem.

This has practical implications in database management, cryptography, and tree-based data structures.

Dynamic programming offers a solution by breaking the problem into smaller overlapping subproblems, allowing for efficient computation.

By solving the subproblems involving the left and right subtrees independently, we can combine their results to compute the total number of reorderings.

RECURSIVE FORMULATION AND DYNAMIC PROGRAMMING APPROACH:

The solution hinges on recursive calculations and combinatorial mathematics. To solve the problem for a given array, we divide it into two parts:

1. Left subtree (elements smaller than the root).

2. Right subtree (elements larger than the root).

The number of ways to reorder the array while preserving the BST structure depends on how we can interleave the elements of the left and right subtrees while maintaining their internal order.

This is equivalent to computing the number of ways to arrange the elements such that the relative order within each subtree is preserved.

Recursive Formula:

Let `L` and `R` represent the sizes of the left and right subtrees, respectively. The total number of ways to reorder the array is given by:

\[
T(n) = C(L + R, L) \times T(L) \times T(R)
\]

Where:

- `T(n)` is the total number of ways to reorder an array of size `n` while preserving the BST structure.

- `C(L + R, L)` is the binomial coefficient, representing the number of ways to interleave `L` elements from the left subtree with `R` elements from the right subtree.

Dynamic Programming Table:

We use a dynamic programming table to store intermediate results for the number of reorderings possible for smaller arrays.

This allows for efficient reuse of previously computed values, reducing the overall computational complexity.

PROGRAM IMPLEMENTATION:

The following C++ program implements the dynamic programming solution to compute the number of ways to reorder an array to obtain the same BST:

cpp

```
#include <iostream>
#include <vector>
using namespace std;

// Helper function to compute binomial coefficients
long long binomialCoeff(int n, int k) {
    vector<vector<long long>> C(n + 1, vector<long long>(k + 1, 0));
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= min(i, k); j++) {
            if (j == 0 || j == i)
                C[i][j] = 1;
            else
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
        }
    }
    return C[n][k];
}
```

```
// Recursive function to count the number of ways to reorder the array
long long countWays(vector<int>& arr) {
    if (arr.size() <= 1) return 1;  // Base case: Single element array or empty array



  vector<int> left, right;
    int root = arr[0];

    // Partition the array into left and right subtrees
    for (int i = 1; i < arr.size(); i++) {
        if (arr[i] < root)
            left.push_back(arr[i]);
        else
            right.push_back(arr[i]);
    }

    // Compute number of ways to interleave left and right subtrees
    long long leftWays = countWays(left);
    long long rightWays = countWays(right);
    long long interleavingWays = binomialCoeff(left.size() + right.size(),
left.size());

    return interleavingWays * leftWays * rightWays;
}

int main() {
    vector<int> arr = {3, 1, 2, 5, 4};
```

```
    cout << "Number of ways to reorder the array to get the same BST: " <<
countWays(arr) << endl;

    return 0;

}
```

COMPLEXITY ANALYSIS:

The time complexity of the dynamic programming solution is dominated by the recursive calls for each subtree and the computation of the binomial coefficients. Specifically:

Time Complexity: \(O(n^2)\) due to the computation of binomial coefficients and the recursive traversal of the tree.

Space Complexity: \(O(n^2)\) to store the binomial coefficients and intermediate results in the dynamic programming table.

FUTURE SCOPE:

Potential future work could include optimizing the space complexity of the algorithm, particularly for larger arrays.

Additionally, parallelizing the recursive calls for the left and right subtrees could lead to performance improvements.

Further exploration may also focus on generalizing this approach to trees with duplicate elements or other types of binary trees.

CONCLUSION:

This paper presents a dynamic programming approach to calculate the number of reorderings of an array that generate the same Binary Search Tree.

By leveraging recursive solutions and combinatorial mathematics, the method efficiently computes the total number of valid reorderings.

This has significant implications for understanding the behavior of BSTs and can be extended to other applications in computational tree structures.