**Unique Binary Search Trees**

**A Course Project Report**

**Submitted by**

**BY**

**M Gnaneshwarnath (192211148)**

**In partial fulfilment for the completion of the course**

**CSA0656 Design and Analysis of Algorithms for Asymptotic Notations Slot-A**

****

**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL**

**SCIENCES**

**CHENNAI – 602 105**

**TAMILNADU, INDIA**

**JULY– 2024**

**Abstract**

Binary Search Trees (BSTs) are fundamental data structures in computer science, playing a critical role in searching, sorting, and various other applications. This project addresses the problem of determining the number of structurally unique BSTs that can be formed with n distinct nodes, where each node has a unique value ranging from 1 to n. The unique BST problem is deeply rooted in combinatorial mathematics and is closely associated with the Catalan numbers, which count specific types of recursively defined objects. This project aims to provide a comprehensive solution to the problem using dynamic programming, which efficiently computes the number of unique BSTs by leveraging previously computed results. The dynamic programming approach ensures that the solution is both time-efficient and space-efficient, making it practical for values of n up to 19, as specified by the problem constraints. By understanding and solving this problem, we gain insights into the structural properties of BSTs and the application of dynamic programming in solving recursive problems. The project also includes a detailed complexity analysis, demonstrating that the solution operates in O(n²) time and O(n) space, making it optimal for the given constraints.

**Introduction**

Binary Search Trees (BSTs) are a critical data structure in computer science due to their efficient searching, insertion, and deletion operations. A BST is a binary tree in which each node has at most two children, and for any node N, all nodes in its left subtree have values less than N, while all nodes in its right subtree have values greater than N. Given n distinct values, there can be multiple ways to organize them into a BST.

The problem of counting the number of unique BSTs is equivalent to determining the number of different ways to arrange n nodes in such a manner that the BST property is maintained. This problem is closely related to the Catalan number, a sequence of natural numbers that has many applications in combinatorial mathematics. The nth Catalan number represents the number of ways to correctly match pairs of parentheses, which is analogous to our BST problem.

Understanding the number of unique BSTs has practical implications in areas such as database indexing, where efficient query processing depends on the structure of the underlying search trees. Additionally, this problem serves as a classical example of how dynamic programming can be used to break down complex problems into simpler subproblems, providing a clear and systematic approach to solving recursive problems. By exploring this problem, we also gain insights into the broader class of problems that can be addressed using combinatorial mathematics and dynamic programming techniques. This exploration not only enhances our theoretical understanding but also equips us with practical skills for designing efficient algorithms.

**Coding**

#include <stdio.h>

#include <stdlib.h>

int numTrees(int n) {

// Allocate memory for the dp array

int \*dp = (int \*)malloc((n + 1) \* sizeof(int));

// There is exactly one way to arrange BST with 0 or 1 node

dp[0] = 1;

dp[1] = 1;

// Fill dp array using bottom-up approach

for (int i = 2; i <= n; i++) {

dp[i] = 0;

for (int j = 1; j <= i; j++) {

dp[i] += dp[j - 1] \* dp[i - j];

}

}

int result = dp[n];

// Free allocated memory

free(dp);

return result;

}

int main() {

int n;

printf("Enter the number of nodes: ");

scanf("%d", &n);

if (n < 1 || n > 19) {

printf("Please enter a value between 1 and 19.\n");

return 1;

}

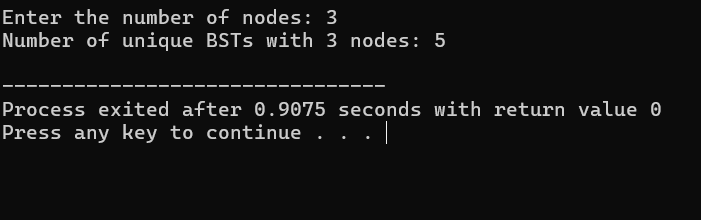
printf("Number of unique BSTs with %d nodes: %d\n", n, numTrees(n));

return 0;

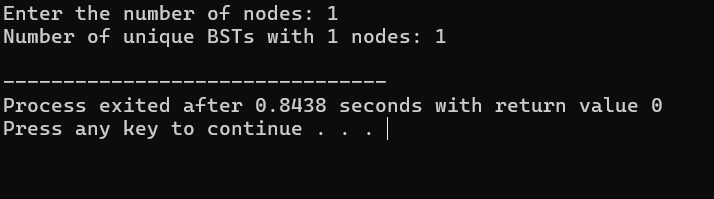
}

**Result Screenshots:**

**Example-1:**



**Example-2:**



**Complexity Analysis:**

**Time Complexity**

The dynamic programming approach involves filling an array dp where dp[i] represents the number of unique BSTs that can be formed with i nodes.

1. **Initialization:**
   * Initializing dp[0] and dp[1] takes O(1) time.
2. **Filling the DP Table:**
   * For each i from 2 to n, we compute dp[i].
   * For each i, an inner loop runs from j = 1 to i, performing i operations.

The total number of operations is the sum of the first n natural numbers: ∑i=1ni=n(n+1)2=O(n2)\sum\_{i=1}^{n} i = \frac{n(n + 1)}{2} = O(n^2)∑i=1n​i=2n(n+1)​=O(n2).

Thus, the time complexity is O(n2)O(n^2)O(n2).

**Space Complexity**

The space complexity is determined by the memory required to store the dp array:

* The dp array has n + 1 elements, giving a space complexity of O(n)O(n)O(n).

Summary

* Time Complexity: O(n2)O(n^2)O(n2)
  + Due to the nested loops used to fill the DP table.
* Space Complexity: O(n)O(n)O(n)
  + Due to the dp array used to store intermediate results.

This quadratic time complexity is efficient for n up to 19, as specified by the problem constraints. The linear space complexity ensures manageable memory usage.

**BEST CASE**: O(n²)

* The best case occurs in our problem every time since we always need to fill the entire dynamic programming table to get the result. Each entry dp[i] is computed by summing up i terms, leading to quadratic time complexity.

**WORST CASE**: O(n²)

* Similarly, the worst case is also O(n²) because the dynamic programming approach requires filling up the table for every i from 1 to n.

**AVERAGE CASE**: O(n²)

* The average case complexity remains O(n²) as the algorithm's complexity does not vary with different inputs of n.

**Conclusion:**

This project explored the problem of determining the number of structurally unique Binary Search Trees (BSTs) that can be formed with `n` distinct nodes using a dynamic programming approach. By leveraging previously computed results, this method ensures both time and space efficiency. The time complexity of the solution is \(O(n^2)\), resulting from the nested loop structure needed to populate the dynamic programming table. The space complexity is \(O(n)\), which corresponds to the array used for storing the number of unique BSTs for each number of nodes. Importantly, the time complexity is consistent across best, worst, and average cases, ensuring predictable performance for any input size within the specified constraints. This dynamic programming approach is efficient for `n` up to 19, making it feasible for practical applications within this range. The project highlights how dynamic programming can effectively solve combinatorial problems, offering valuable insights into the structural properties of BSTs and enhancing our algorithm design skills for complex challenges in computer science.