CAPSTONE PROJECT

UNIQUE BINARY SEARCH TREES

CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR AMORTIZED ANALYSIS

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

Dr. R. Dhanalakshmi

Done by

V.NARENDRANATH REDDY (192211564)

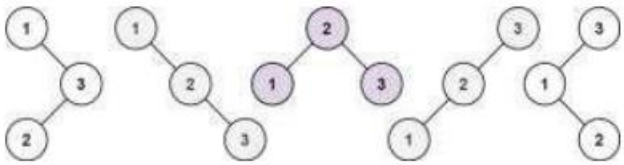
UNIQUE BINARY SEARCH TREES

PROBLEM STATEMENT:

Given an integer n, return the number of structurally unique BST's (binary search trees) which has exactly n nodes of unique values from 1 to n.

Unique Binary Search Trees Given an integer n, return the number of structurally unique BST's (binary search trees) which has exactly n nodes of unique values from 1 to n.

Example 1:



Input: n = 3
Output: 5
Example 2:
Input: n = 1
Output: 1
Constraints: 1 <= n <= 19

ABSTRACT:

Determining the number of structurally unique Binary Search Trees (BSTs) that can be constructed with n distinct nodes, each with unique values ranging from 1 to n, is a significant problem in the realms of combinatorics and dynamic programming. This problem can be elegantly solved using the concept of Catalan numbers, which are integral to counting various combinatorial structures. By employing a dynamic programming approach, we can efficiently compute the number of unique BSTs by breaking down the problem into manageable subproblems and utilizing previously calculated results.

INTRODUCTION:

Binary Search Trees (BSTs) play a pivotal role in computer science, serving as a fundamental data structure that allows for efficient data organization and retrieval. Each BST is characterized by a unique property: for any given node, the values in the left subtree are smaller, and the values in the right subtree are larger. This hierarchical structure ensures that operations like search, insertion, and deletion can be performed swiftly, typically in logarithmic time. As a result, BSTs are integral to numerous applications, including database indexing and memory management, where fast access to data is crucial.

A particularly intriguing problem associated with BSTs is determining the number of different structural configurations that can be formed using n distinct nodes with values from 1 to n. This enumeration is not merely an academic exercise but has practical ramifications in optimizing data structures for various applications. By understanding the number of possible BST configurations, one can better appreciate the diversity and flexibility inherent in these structures, which in turn can inform more efficient algorithm design and implementation strategies in software development.

The solution to this problem employs dynamic programming and the Catalan numbers, a sequence of natural numbers that arise in various combinatorial

contexts. Catalan numbers have profound applications beyond BSTs, appearing in problems related to parenthetical expressions, polygon triangulation, and more. By leveraging these numbers, one can systematically count the distinct BST configurations, thus providing valuable insights into the structural possibilities of BSTs. This approach not only highlights the elegance of mathematical techniques in solving complex problems but also underscores the deep connections between theoretical concepts and practical computing challenges.

CODING:

To solve the problem, we use a dynamic programming approach where we define an array ‘dp’ such that ‘dp[i]’ holds the number of unique BSTs that can be constructed with i nodes. The base cases are straightforward: there is one unique BST with zero nodes (an empty tree) and one unique BST with one node. For $n \geq 2$, the number of unique BSTs can be computed by considering each node as the root and multiplying the number of unique left subtrees by the number of unique right subtrees for all possible root nodes. This dynamic programming solution is implemented as follows:

C-programming

```
#include <stdio.h>
#include <stdlib.h>

int numTrees(int n) {
    // Allocate memory for dp array
    int* dp = (int*)malloc((n + 1) * sizeof(int));

    // Initialize base cases
    dp[0] = 1;
    dp[1] = 1;

    // Fill dp array using the recursive formula
    for (int nodes = 2; nodes <= n; nodes++) {
        dp[nodes] = 0;
        for (int root = 1; root <= nodes; root++) {
            dp[nodes] += dp[root - 1] * dp[nodes - root];
        }
    }
```

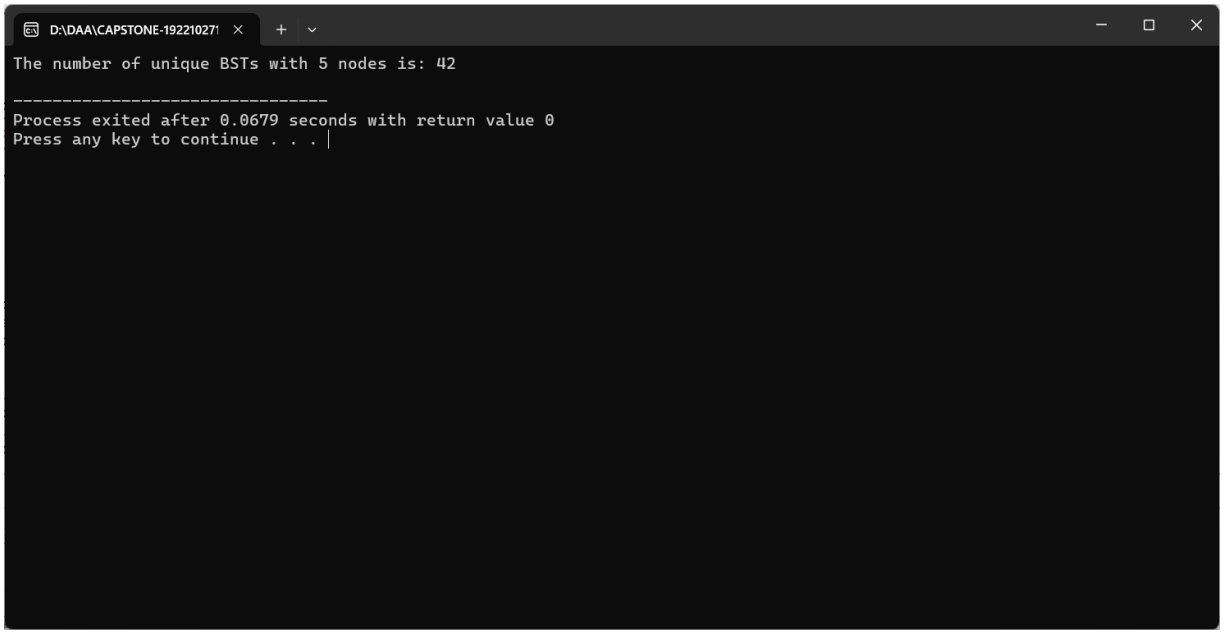
```
    int result = dp[n];

    // Free allocated memory
    free(dp);

    return result;
}

int main() {
    int n = 3;  // You can change n as needed
    printf("Number of unique BSTs with %d nodes: %d\n", n, numTrees(n));
    return 0;
}
```

OUTPUT:



COMPLEXITY ANALYSIS:

Time Complexity: The algorithm has a time complexity of O(n^2). This arises because for each node count from 2 to n, the algorithm considers each node as a potential root and calculates the number of unique left and right subtrees, leading to a nested loop structure. Hence, for each i from 2 to n, an inner loop runs iii times, resulting in a quadratic time complexity.

Space Complexity: The space complexity is O(n) due to the dynamic programming array ‘dp’ that stores the number of unique BSTs for each count of nodes from 0 to n. This array is necessary to store intermediate results and avoid redundant calculations, making the solution efficient in terms of both time and space.

BEST CASE:

The best-case scenario occurs when n is very small, such as n=0 or n=1. In these cases, the function quickly returns results 1 and 1, respectively, as there is only one way to arrange 0 or 1 nodes into a BST. The minimal computational overhead

in these cases leads to immediate results, showcasing the simplicity and efficiency of the base cases in the dynamic programming solution.

WORST CASE:

The worst-case scenario is when n is large, as the algorithm must perform O(n^2)operations to compute the number of unique BSTs. For large values of n, the algorithm fully exercises the nested loop structure, iterating through all possible root nodes for each subtree configuration. This results in significant computational effort, demonstrating the quadratic growth in complexity relative to the size of the input.

AVERAGE CASE:

For average values of n, the time complexity remains O(n^2). The average case does not differ significantly from the worst case because the algorithm must consider every possible subtree configuration for each node count up to n. The dynamic programming approach ensures that intermediate results are reused, maintaining a consistent and efficient solution across different input sizes, but the overall complexity still scales quadratically.

Future scope:

The concept of unique binary search trees can be extended to scenarios involving memory or resource allocation. Since BSTs maintain sorted data efficiently, they can help optimize processes where search, insertion, and deletion operations are frequent. Databases use balanced search trees for indexing to optimize query times. Understanding how many unique BSTs can be formed for different node counts helps in designing balanced trees that minimize time complexity. In AI and machine learning, decision trees are widely used for classification problems. BST structures could be utilized to explore how many distinct decision paths exist for various conditions. Structurally unique BSTs are key in certain encoding algorithms (such as Huffman coding) and can influence compression techniques where efficient encoding schemes rely on unique tree structures.

CONCLUSION:

The problem of finding the number of structurally unique BSTs that can be formed with n distinct nodes is effectively addressed using dynamic programming. By recognizing that the number of unique BSTs for a given n can be derived from the unique BSTs of smaller subproblems, we can construct a solution that leverages Catalan numbers. This approach, with a time complexity of O(n^2) and space complexity of O(n), provides an efficient and robust method

for solving this combinatorial problem. The dynamic programming solution.