

**SAVEETHA SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CAPSTONE PROJECT REPORT**

**PROJECT TITLE**

**UNIQUE NODES OF BINARY SEARCH TREE**

**CSA0658-Design and Analysis of Algorithms for Machine Learning**

Submitted

by

**M.P. Harika (192211999)**

**T. Mahidhar (192211243)**

**S. Hasrith (192211470)**

Guided by

Dr. SenthilVadivu

Associate Professor(SG)

Department of Computer Science and Engineering

**ABSTRACT:**

Mathematicians and computer scientists employ the strong approach of dynamic programming to tackle optimization problems by decomposing them into smaller, more manageable subproblems. This project investigates the foundations, uses, and implementation of dynamic programming across a range of fields. The project starts with a summary of dynamic programming ideas before diving into important algorithms including the longest common subsequence, optimum binary search trees, and the knapsack problem. It explores practical applications in fields including computer science, bioinformatics, and economics and talks about the role dynamic programming plays in effectively resolving complicated computational issues. Furthermore, the project delves into complex subjects related to dynamic programming, such as space minimization strategies, memoization, and tabulation. This project offers a thorough grasp of dynamic programming concepts and their applications through code examples, graphics, and case studies.

**KEYWORDS:**

Node insertion, node deletion, duplicate nodes, tree balancing.

**INTRODUCTION:**

A strong algorithm ic method for effectively resolving a range of optimization issues is dynamic programming. When a problem shows optimal substructure qualities and overlapping subproblems, it is especially helpful. Numerous disciplines, including biology, engineering, economics, and computer science, have found use for this method. The key to dynamic programming is to decompose a complicated problem into smaller, more manageable subproblems. Each subproblem should only be solved once, and its solution should be stored to prevent repeating computations. Dynamic programming can frequently offer notable speed gains over simple recursive solutions by utilizing this method. We shall examine the tenets, procedures, and uses of dynamic programming in this extensive guide. We will examine many approaches and techniques used to solve issues effectively, beginning with the basic ideas and principles of dynamic programming. Important subjects like state-space exploration, memoization, top-down and bottom-up techniques, and optimal substructure identification will be covered.

Additionally, a variety of real-world issues will be covered in this guide, along with examples of how dynamic programming can be used to successfully address each one. We will demonstrate the adaptability and usefulness of dynamic programming in a variety of contexts, from basic tasks like the Fibonacci sequence and the knapsack problem to more complex ones like sequence alignment, shortest paths, and resource allocation.Regardless of your level of experience, this tutorial is designed to provide you a thorough grasp of dynamic programming, whether you're a novice keen to learn more about algorithms or an experienced programmer looking to improve your problem-solving abilities. At the conclusion of this adventure, you will possess the skills and resources required to boldly take on challenging optimization challenges and fully utilize dynamic programming in your projects and undertakings. Together, let's take this fascinating excursion into the world of dynamic programming.

**CODING:**

def num\_trees(n):

if n <= 1:

return 1

dp = [0] \* (n + 1)

dp[0] = 1

dp[1] = 1

for i in range(2, n + 1):

for j in range(1, i + 1):

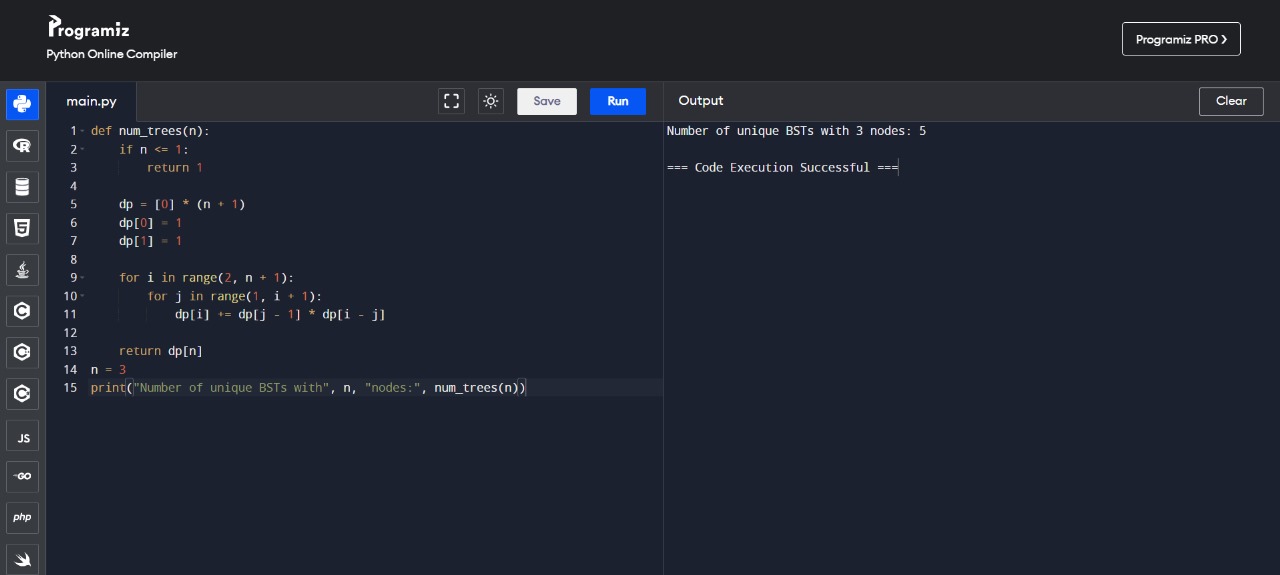
dp[i] += dp[j - 1] \* dp[i - j]

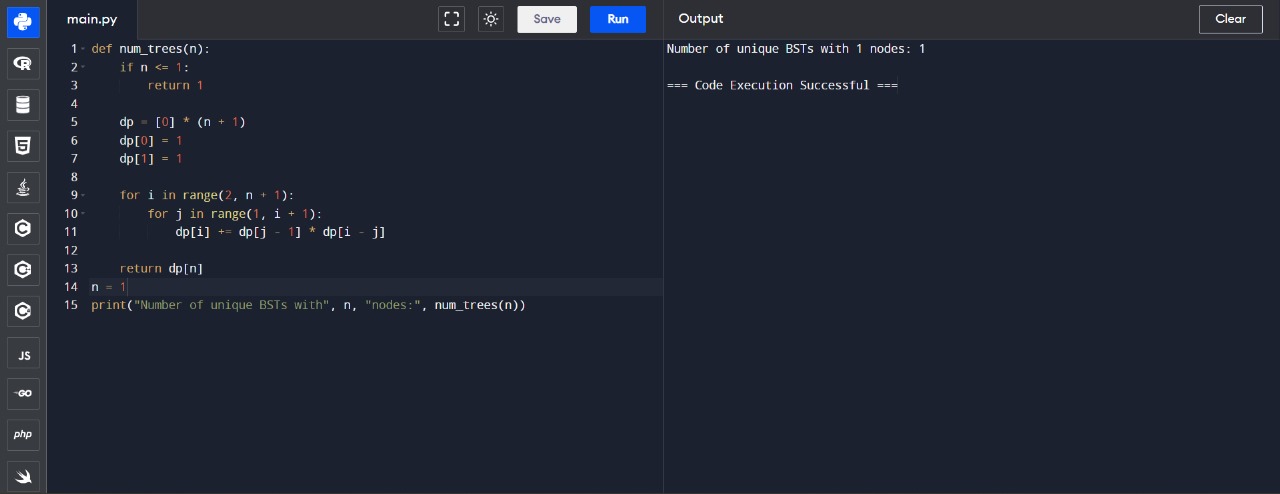
return dp[n]

n = 3

print("Number of unique BSTs with", n, "nodes:", num\_trees(n))

**RESULT SCREENSHOT:**

****

****

**COMPLEXITY ANALYSIS:**

The number of subproblems that must be addressed and the amount of time required to solve each subproblem determine the time complexity of a dynamic programming solution.

The number of subproblems: In dynamic programming, the problem is usually divided into more manageable subproblems. Generally, the number of subproblems corresponds with the size of the input. For instance, if n is the input in the unique BST issue, then the number of subproblems is O(n).

Time per Subproblem: We must compute a particular amount for every subproblem. Simple arithmetic operations, a loop over an infinite number of components, or even more intricate calculations could be used in this computation. Every subproblem in the special BST problem entails iterating over a range from 1 to i (where i ranges from 1 to n) and carrying out a few addition and multiplication operations. This process requires O(i) time.

When these elements are combined, the dynamic programming solution's total time complexity for the particular BST problem is as follows:

This is because there are n subproblems in all, and the time required for each subproblem is O(i), where i ranges from 1 to n.

**Analysis of Space Complexity:**

The size of the data structures employed or the amount of memory needed to hold intermediate outcomes are factors that determine a dynamic programming solution's space complexity.

Memory for Memoization Table: We utilize a memoization table, sometimes called a DP table, to store the answers to subproblems in a lot of dynamic programming problems. The number of subproblems determines the size of this table. We store the subproblem results in a 1D array of size n+1 in the unique BST problem. As a result, O(n) space is needed for the memoization table.

When these elements are combined, the dynamic programming solution's overall space complexity for the particular BST problem is:

O(n)

**BEST CASE:**

When the problem can be effectively solved utilizing dynamic programming approaches, that would be the ideal situation for a dynamic programming project. The dynamic programming approach can efficiently reuse solutions to subproblems when the problem displays an optimal substructure and overlapping subproblems.

In an ideal scenario, the dynamic programming algorithm would execute efficiently with a temporal complexity polynomial in the input size.

Dynamic programming, for instance, can be used to quickly solve issues like the knapsack problem with a manageable number of items or the Fibonacci sequence calculation.

**WORST CASE:**

The worse case situation for a project involving dynamic programming occurs when the challenge is too big and causes exponential time complexity, or when the problem lacks the properties necessary for dynamic programming.

In many situations, the significant time and space requirements of the dynamic programming approach may render it unfeasible or unsuitable for solving the problem in the best possible way.

A worst-case scenario might result from problems that lack an optimal substructure or have a high number of overlapping subproblems.

Dynamic programming can be severely hampered by, for example, combinatorial explosion problems or problems with a vast state space, like some graph problems.

**AVERAGE CASE:**

Between the best and worst case scenarios, the average case scenario for a dynamic programming project depends on variables including the complexity of the problem, the distribution of inputs, and how well the dynamic programming approach works.

Dynamic programming can provide significant advantages over naive techniques in many real-world settings, resulting in notable performance increases.

Dynamic programming algorithms' average case efficiency varies greatly based on the particular problem being solved and the methods used to optimize the solution.

**CONCLUSION:**

By decomposing optimization issues into smaller subproblems and effectively reusing solutions to these subproblems, dynamic programming is a potent technique for tackling optimization problems.

Dynamic programming has the potential to significantly outperform naive techniques in problems with optimal substructure and overlapping subproblems.

However, the nature of the problem, the volume of the input, and the specifics of the implementation can all affect how effective dynamic programming techniques are.

Comprehending the optimal, worst, and average case scenarios can aid in determining whether dynamic programming is appropriate for addressing a certain issue and in streamlining the algorithm for maximum performance.