Design and Analysis of Algorithms

Lab - 7

Dynamic Programming

Dynamic programming is a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution.

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

A. Write a C/C++ program for counting unique binary search trees using numbers 1 through n.

Perform the comparative complexity analysis of optimized solution using dynamic Programming with respect to plain recursion solution for different size values, n.

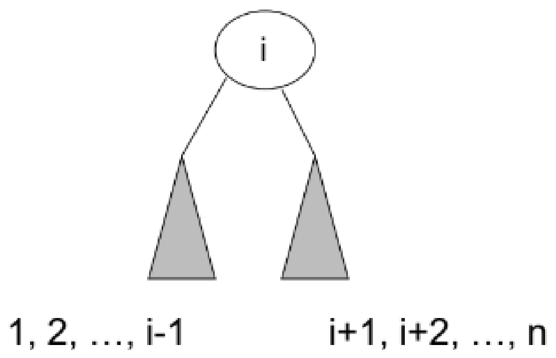
B. Write a C/C++ program for the implementation of Knapsack Problem using dynamic programming.

Do the run time analysis and time complexity analysis with the different values of input size. Maintain the tabular data (n, execution time) and plot it graphically using data plotting tools.

Suggestion:

Counting binary search tree

For counting the unique binary tree using the numbers from 1 to n (given). Pick a root, say i, and find the unique binary search tree using 1 to i -1 (left subtree) and i + 1 to n (right subtree).



See it as subproblem and formulate as $T(i) = T(i-1) * T(n-i)$ and solve it using dynamic programming (memorization in bottom-up approach).

Compare the complexity analysis with the plain recursion solution

Knapsack Problem

For given N items, each item having some weights and values associated with it, say, N = 3, weights = {4, 5, 1} and val = {1, 2, 3}. Also given a Knapsack with the capacity W. Say, W = 4. The Knapsack problem is to find the set of items (to be put in the knapsack) such that its maximum total value and the sum of their weights within the knapsack capacity, W.

So, find out the maximum value subset of val[] such that the sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don’t pick it (0–1 property). Here answer is {3}