Dynamic Programming

Content

Introduction

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. For example, if we write simple recursive solutions for Fibonacci numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

Applications

1. **The Fibonacci numbers are the numbers in the following integer sequence. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …….. Given a number n, print n-th Fibonacci Number.**

   In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$

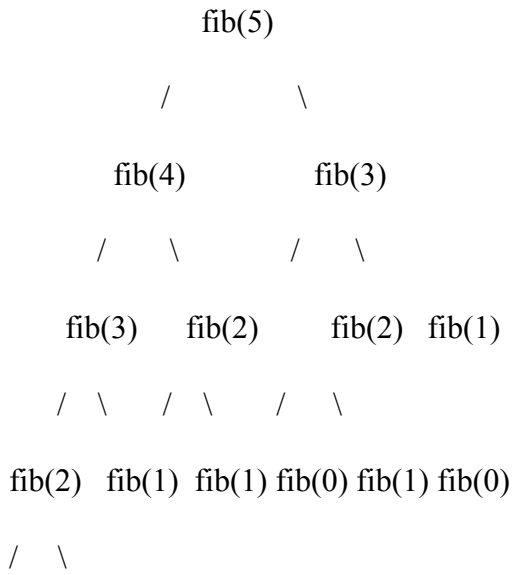
   with seed values

   $F_0 = 0$ and $F_1 = 1$.

   Recursive Approach - A simple method that is a direct recursive implementation mathematical recurrence relation given above.

   Time Complexity **-** T(n) = T(n-1) + T(n-2) which is exponential.

   We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.



Extra Space - O(n) if we consider the function call stack size, otherwise O(1).

Approach - Dynamic Programming

We can avoid the repeated work done in method 1 by storing the Fibonacci numbers calculated so far.

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/fibonacci_dp.cpp

2. **Let 1 represent ‘a’, 2 represents ‘b’, etc. Given a digit sequence, count the number of possible decodings of the given digit sequence.**

   Algorithm -

   Input: digits[] = "121"
   Output: 3
   // The possible decodings are "aba", "au", "la"

   Input: digits[] = "1234"

Output: 3

// The possible decodings are "abcd", "lcd", "awd”

An empty digit sequence is considered to have one decoding. It may be assumed that the input contains valid digits from 0 to 9 and there are no leading 0’s, no extra trailing 0’s, and no two or more consecutive 0’s.

This problem is recursive and can be broken into sub-problems. We start from the end of the given digit sequence. We initialize the total count of decodings as 0. We recur for two subproblems.

1. If the last digit is non-zero, recur for the remaining (n-1) digits and add the result to the total count.
2. If the last two digits form a valid character (or smaller than 27), recur for the remaining (n-2) digits and add the result to the total count.

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/alpha_code.cpp

3. Count ways to reach the nth stair using step 1 or 2

A child is running up a staircase with n steps and can hop either 1 step or 2 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

1. Recursive Method
2. Dynamic Programming

Method 1: Recursive

There are n stairs, and a person is allowed to jump next stair, skip one stair or skip two stairs. So there are n stairs. So if a person is standing at i-th stair, the person can move to i+1, i+2-th stair. A recursive function can be formed where at current index i the function is recursively called for i+1 and i+2 th stair.

There is another way of forming the recursive function. To reach stair i, a person has to jump either from i-1 or i-2 th stair or i is the starting stair.

Algorithm**:**

1. Create a recursive function (count(int n)) that takes only one parameter.
2. Check the base cases. If the value of n is less than 0 then return 0, and if the value of n is equal to zero then return 1 as it is the starting stair.
3. Call the function recursively with values n-1, n-2 and sum up the values that are returned, i.e. sum = count(n-1) + count(n-2)Return the value of the sum

Time Complexity - $O(3^n)$.

The time complexity of the above solution is exponential, a close upper bound will be $O(3^n)$. From each state, 3 recursive functions are called. So the upper bound for n states is $O(3^n)$.

Space Complexity **-** O(1).

As no extra space is required.

Note: The Time Complexity of the program can be optimized using Dynamic Programming.

Method 2: Dynamic Programming

The idea is similar, but it can be observed that there are n states but the recursive function is called 2 ^ n times. That means that some states are called repeatedly. So the idea is to store the value of states. This can be done in two ways.

1. *Top-Down Approach:* The first way is to keep the recursive structure intact and just store the value in a HashMap and whenever the function is called again return the value store without computing ().

2. *Bottom-Up Approach:* The second way is to take an extra space of size n and start computing values of states from 1, 2 .. to n, i.e. compute values of i, i+1, i+2 and then use them to calculate the value of i+3.

Algorithm

1. Create an array of size n + 1 and initialize the first 3 variables with 1, 1, 2. The base cases.
2. Run a loop from 3 to n.
3. For each index i, computer value of ith position as dp[i] = dp[i-1] + dp[i-2] + dp[i-3].
4. Print the value of dp[n], as the Count of the number of ways to reach n th step.

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/staircase_problem.cpp

Time Complexity - The time complexity of this recursive approach is exponential as there is a case of overlapping subproblems.

Auxiliary Space - O(1). No external space is used for storing values apart from the internal stack space.

4. Maximum Rectangle sum in a 2D array

A matrix is given. We need to find a rectangle (sometimes square) matrix, whose sum is maximum.

The idea behind this algorithm is to fix the left and right columns and try to find the sum of the element from the left column to the right column for each row, and store it temporarily. We will try to find top and bottom row numbers. After getting the temporary array, we can apply Kadane's Algorithm to get the maximum sum subarray. With it, the total rectangle will be formed.

Implementation - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Dynamic%20Programming/Codes/maximum_rectangle_sum_in_2D_array.cpp