Home » Wiki » Tutorial for Dynamic Programming

Tutorial For Dynamic Programming

Table of Contents [hide]

Introduction

Dynamic programming (usually referred to as **DP**) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again.. shortly *'Remember your Past'* :) . If the given problem can be broken up in to smaller sub-problems and these smaller subproblems are in turn divided in to still-smaller ones, and in this process, if you observe some over-lapping subproblems, then its a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem (referred to as the Optimal Substructure Property).

There are two ways of doing this.

1.) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memoization***.

2.) Bottom-Up : Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as ***Dynamic Programming***.

Note that divide and conquer is slightly a different technique. In that, we divide the problem in to non-overlapping subproblems and solve them independently, like in mergesort and quick sort.

In case you are interested in seeing visualizations related to Dynamic Programming try this out.

Complementary to Dynamic Programming are Greedy Algorithms which make a decision once and for all every time they need to make a choice, in such a way that it leads to a near-optimal solution. A Dynamic Programming solution is based on the principal of Mathematical Induction greedy algorithms require other kinds of proof.

Cold War Between Systematic Recursion And Dynamic Programming

Recursion uses the top-down approach to solve the problem i.e. It begin with core(main) problem then breaks it into subproblems and solve these subproblems similarily. In this approach same subproblem can occur multiple times and consume more CPU cycle ,hence increase the time complexity. Whereas in Dynamic programming same subproblem will not be solved multiple times but the prior result will be used to optimise the solution. eg. In fibonacci series :-

Fib(4) = Fib(3) + Fib(2)

= (Fib(2) + Fib(1)) + Fib(2)

l"> =((Fib(1) + Fib(0)) + Fib(1)) + Fib(2)

=((Fib(1) + Fib(0)) + Fib(1)) + (Fib(1) + Fib(0))

Here, call to Fib(1) and Fib(0) is made multiple times.In the case of Fib(100) these calls would be count for million times. Hence there is lots of wastage of resouces(CPU cycles & Memory for storing information on stack).

In dynamic Programming all the subproblems are solved even those which are not needed, but in recursion only required subproblem are solved. So solution by dynamic programming should be properly framed to remove this ill-effect.

For ex. In combinatorics, C(n.m) = C(n-1,m) + C(n-1,m-1).

```
         1
       1   1
     1   2   1
   1   3   3   1
 1   4   6   4   1
1  5  10  10  5  1
```

In simple solution, one would have to construct the whole pascal triangle to calcute C(5,4) but recursion could save a lot of time.

Dynamic programming and recursion work in almost similar way in the case of non overlapping subproblem. In such problem other approaches could be used like “divide and conquer” .

Even some of the high-rated coders go wrong in tricky DP problems many times. DP gurus suggest that DP is an art and its all about Practice. The more DP problems you solve, the easier it gets to relate a new problem to the one you solved already and tune your thinking very fast. It looks like a magic when you see some one solving a tricky DP so easily. Its time for you to learn some magic now :). Lets start with a very simple problem.

Problem : Minimum Steps To One

Problem Statement: On a positive integer, you can perform any one of the following 3 steps. ***1.)*** Subtract 1 from it. (n = n - 1) , ***2.)*** If its divisible by 2, divide by 2. (if n % 2 == 0 , then n = n / 2) , ***3.)*** If its divisible by 3, divide by 3. (if n % 3 == 0 , then n = n / 3). Now the question is, given a positive integer *n*, find the minimum number of steps that takes *n* to 1

eg: 1.)For n = 1 , output: 0 2.) For n = 4 , output: 2 (4 **/2** = 2 **/2** = 1) 3.) For n = 7 , output: 3 (7 **-1** = 6 **/3** = 2 **/2** = 1)

Approach / Idea: One can think of greedily choosing the step, which makes *n* as low as possible and conitnue the same, till it reaches 1. If you observe carefully, the greedy strategy doesn't work here. Eg: Given *n* = 10 , Greedy --> 10 /2 = 5 -1 = 4 /2 = 2 /2 = 1 (4 steps). But the optimal way is --> 10 -1 = 9 /3 = 3 /3 = 1 (3 steps). So, we need to try out all possible steps we can make for each possible value of *n* we encounter and choose the minimum of these possibilities.

It all starts with recursion :). F(n) = 1 + min{ F(n-1) , F(n/2) , F(n/3) } if (n>1) , else 0 (i.e., F(1) = 0) . Now that we have our recurrence equation, we can right way start coding the recursion. Wait.., does it have over-lapping subproblems ? YES. Is the optimal solution to a given input depends on the optimal solution of its subproblems ? Yes... Bingo ! its DP :) So, we just store the solutions to the subproblems we solve and use them later on, as in memoization.. or we start from bottom and move up till the given *n*, as in dp. As its the very first problem we are looking at here, lets see both the codes.

Memoization

[code]

int memo[n+1]; // we will initialize the elements to -1 (-1 means, not solved it yet)

```
int getMinSteps ( int n )

{

if ( n == 1 )  return 0;  // base case

if( memo[n] != -1 ) return memo[n];  // we have solved it already :)

int r = 1 + getMinSteps( n - 1 );  // '-1' step .  'r' will contain the optimal answer finally

if( n%2 == 0 )   r  =  min( r , 1 + getMinSteps( n / 2 ) ) ;  //  '/2' step

if( n%3 == 0 )   r  =  min( r , 1 + getMinSteps( n / 3 ) ) ;  //  '/3' step

memo[n] = r ;  // save the result. If you forget this step, then its same as plain recursion.

return r;

}
```

[/code]

Bottom-Up DP

[code]

```
int getMinSteps ( int n )

{

int dp[n+1] , i;

dp[1] = 0;  // trivial case

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

{

dp[i] = 1 + dp[i-1];

if(i%2==0) dp[i] = min( dp[i] , 1+ dp[i/2] );

if(i%3==0) dp[i] = min( dp[i] , 1+ dp[i/3] );

}

return dp[n];

}
```

[/code]

Both the approaches are fine. But one should also take care of the lot of over head involved in the function calls in Memoization, which may give StackOverFlow error or TLE rarely.

Identifying The State

Problem : Longest Increasing Subsequence

The Longest Increasing Subsequence problem is to find the longest increasing subsequence of a given sequence. Given a sequence S= {a_1 , a_2 , a_3, a_4, ..., a_{n-1}, a_n } we have to find a longest subset such that for all j and i, j<i in the subset a_j<a_i.

First of all we have to find the value of the longest subsequences(LS_i) at every index i with last element of sequence being a_i. Then largest LS_i would be the longest subsequence in the given sequence. To begin LS_i is assigned to be one since a_i is element of the sequence(Last element). Then for all j such that j<i and a_j<a_i ,we find Largest LS_j and add it to LS_i. Then algorithm take O(n^2) time.

Pseudo-code for finding the length of the longest increasing subsequence:

This algorithms complexity could be reduced by using better data structure rather than array. Storing predecessor array and variable like largest_sequences_so_far and its index would save a lot time.

Similar concept could be applied in finding longest path in Directed acyclic graph.

```
for i=0 to n-1
        LS[i]=1
        for j=0 to i-1
                if (a[i] >  a[j] and LS[i]<LS[j])
                        LS[i] = LS[j]+1
 for i=0 to n-1
        if (largest < LS[i])
                largest = LS[i]
```

Problem : Longest Common Subsequence (LCS)

Longest Common Subsequence - Dynamic Programming - Tutorial and C Program Source code

Given a sequence of elements, a subsequence of it can be obtained by removing zero or more elements from the sequence, preserving the relative order of the elements. Note that for a substring, the elements need to be contiguous in a given string, for a subsequence it need not be. Eg: S1="ABCDEFG" is the given string. "ACEG", "CDF" are subsequences, where as "AEC" is not. For a string of lenght n the total number of subsequences is 2^n (Each character can be taken or not taken). Now the question is, what is the length of the longest subsequence that is common to the given two Strings S1 and S2. Lets denote length of S1 by N and length of S2 by M.

BruteForce : Consider each of the 2^N subsequences of S1 and check if its also a subsequence of S2, and take the longest of all such subsequences. Clearly, very time consuming.

Recursion : Can we break the problem of finding the LCS of S1[1...N] and S2[1...M] in to smaller subproblems ?

Memory Constrained DP

[to do , fibonacci , LCS etc.,]

Practice Problems

1. Other Classic DP problems : 0-1 KnapSack Problem (tutorial and C Program), Matrix Chain Multiplication (tutorial and C Program), Subset sum, Coin change, All to all Shortest Paths in a Graph (tutorial and C Program), Assembly line joining or topographical sort

You can refer to some of these in the Algorithmist site

2. The lucky draw(June 09 Contest). http://www.codechef.com/problems/D2/

3. Find the number of increasing subsequences in the given subsequence of length 1 or more.

4.SPOJ-

To see problems on DP visit this link

5.TopCoder - ZigZag

6.TopCoder - AvoidRoads - A simple and nice problem to practice

7. For more DP problems and different varieties, refer a very nice collection http://www.codeforces.com/blog/entry/325

8.. TopCoder problem archive

This is not related to Dynamic Programming, but as 'finding the n^{th} [[http://www.thelearningpoint.net/computer-science/learning-python-programming-and-data-structures/learning-python-programming-and-data-structures--tutorial-7--functions-and-recursion-multiple-function-arguments-and-partial-functions|Fibonacci number]]' is discussed, it would be useful to know a very fast technique to solve the same.

Finding N^{th} Finobacci Number In O(Log N)

Note: The method described here for finding the n^{th} Fibonacci number using dynamic programming runs in O(n) time. There is still a better method to find F(n), when n become as large as 10^{18} (as F(n) can be very huge, all we want is to find the F(N)%MOD , for a given MOD).

Consider the Fibonacci recurrence F(n+1) = F(n) + F(n-1). We can represent this in the form a matrix, we shown below.

$\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}\begin{pmatrix}f(n)\\f(n-1)\end{pmatrix} = \begin{pmatrix}f(n+1)\\f(n)\end{pmatrix} \quad \text{where} \begin{pmatrix}f(1)\\f(0)\end{pmatrix} = \begin{pmatrix}1\\1\end{pmatrix}$ Look at the matrix A = [[1 1] [1 0]] . Multiplying A with [F(n) F(n-1)] gives us [F(n+1) F(n)] , so.. we

start with [F(1) F(0)] , multiplying it with A^n gives us [F(n+1) F(n)] , so all that is left is finding the n^{th} power of the matrix A. Well, this can be computed in O(log n) time, by recursive doubling. The idea is, to find A^n , we can do R = $A^{n/2}$ x $A^{n/2}$ and if n is odd, we need do multiply with an A at the end. The following pseudo code shows the same.

[code]

Matrix findNthPower(Matrix M , power n)

{

if(n == 1) return M;

Matrix R = findNthPower (M , n/2);

R = RxR; // matrix multiplication

if(n%2 == 1) R = RxM; // matrix multiplication

return R;

}

[/code]

You can read more about it here

This method is in general applicable to solving any Homogeneous Linear Recurrence Equations, eg: G(n) = a.G(n-1) + b.G(n-2) - c.G(n-3) , all we need to do is to solve it and find the Matrix A and apply the same technique.

Tutorials and C Program Source Codes for Common Dynamic Programming problems

Floyd Warshall Algorithm - Tutorial and C Program source code:http://www.thelearningpoint.net/computer-science/algorithms-all-to-all-shortest-paths-in-graphs---floyd-warshall-algorithm-with-c-program-source-code

Integer Knapsack Problem - Tutorial and C Program source code: http://www.thelearningpoint.net/computer-science/algorithms-dynamic-programming---the-integer-knapsack-problem

Longest Common Subsequence - Tutorial and C Program source code : http://www.thelearningpoint.net/computer-science/algorithms-dynamic-programming---longest-common-subsequence

Matrix Chain Multiplication - Tutorial and C Program source code : http://www.thelearningpoint.net/algorithms-dynamic-programming---matrix-chain-multiplication
Related topics: Operations Research, Optimization problems, Linear Programming, Simplex, LP Geometry
Floyd Warshall Algorithm - Tutorial and C Program source code: http://www.thelearningpoint.net/computer-science/algorithms-all-to-all-shortest-paths-in-graphs---floyd-warshall-algorithm-with-c-program-source-code

Dynamic Programming techniques are primarily based on the principle of Mathematical Induction unlike greedy algorithms which try to make an optimization based on local decisions, without looking at previously computed information or tables. Some classic cases of greedy algorithms are the greedy knapsack problem, huffman compression trees, task scheduling. Insertion sort is an example of dynamic programming, selection sort is an example of greedy algorithms,Merge Sort and Quick Sort are example of divide and conquer. So, different categories of algorithms may be used for accomplishing the same goal - in this case, sorting.

Comments

2★ ***sourcewizard*** *@ 21 Mar 2013 12:11 AM*

yup , its a classic misconception.