MIT OpenCourseWare
http://ocw.mit.edu

6.006 Introduction to Algorithms
Spring 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

Recitation 16 Notes

Dynamic Programming

These notes are supposed to be complementary to the lecture notes. Everything that we covered in section is contained either here or there.

Outline

1. Introduction to Dynamic Programming
2. Memoization: computing Fibonacci numbers

Introduction to Dynamic Programming

Good news: Dynamic Programming (DP) is used heavily in optimization problems. Applications range from financial models and operations research to biology and basic algorithm research. So understanding DP is profitable.

Bad news: DP is not an algorithm or a data structure. You don't get to walk away with algorithms that you can memorize.

DP takes advantage of the *optimal sub-structure* of a problem.

Optimal sub-structure

A problem has an optimal sub-structure if the optimum answer to the problem contains optimum answers to smaller sub-problems.

For instance, in the minimum-path problem, suppose s → u → v is a minimum-cost path from s to v. This implies that s → u is a minimum-cost path from s to u, and can be proved by contradiction. If there is a better path between s and u, we can replace s → u with the better path in s → u → v, and this would yield a better path between s and v. But we assumed that s → u → v is an optimal path between s and v, so we have a contradiction.

How do you take advantage of the optimal sub-structure? DP comes as as a bunch of tricks, and I think the only way to master it is to understand DP solutions to problems.

Memoization

In DP, the answer to the same sub-problem is used multiple times. Computing the same answer multiple times can be very inefficient (as demonstrated in the next problem), so DP solutions store and reuse the answers to sub-problems. This is called memoization.

Memoization is very simple to implement in high-level languages like Python. The easiest approach is to use a dictionary that persists across function calls (not a local variable ☺) to store pre-computed results. The dictionary keys are parameters to the function, and the values are the result of the computation.

At the beginning of the function call, check if the parameters (n for the Fibonacci example) exist in the dictionary. If that is the case, return the associated value, bypassing the computation logic completely. Otherwise, execute the computation logic, and store the result in the dictionary before returning from the function.

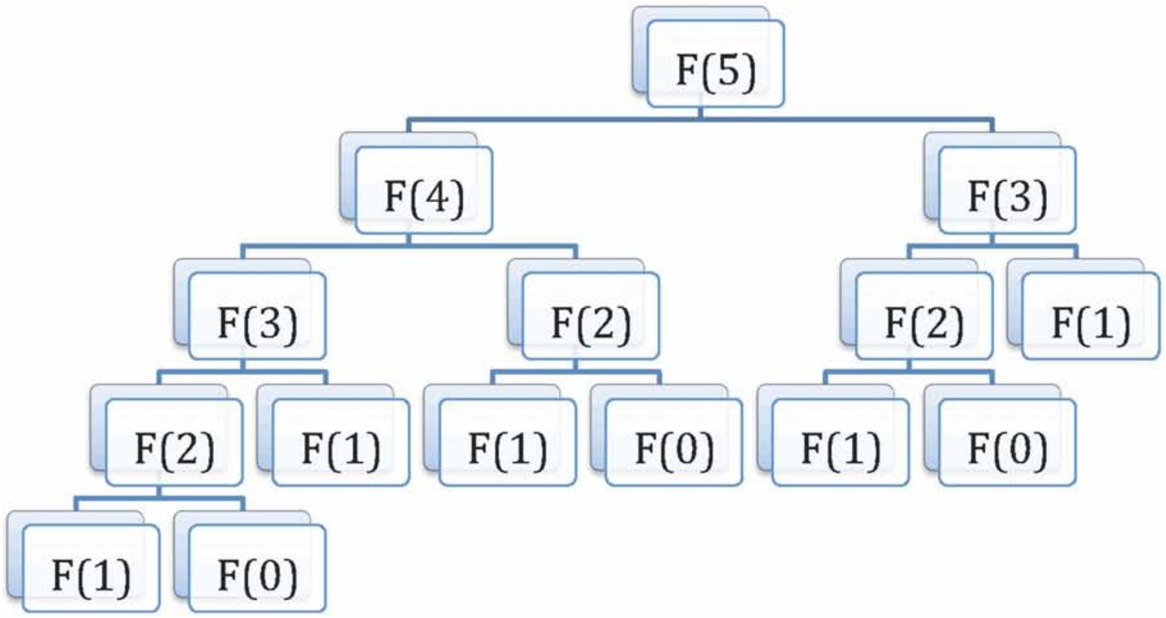
Fibonacci numbers

A dramatic example of the usefulness of memoization is computing Fibonacci numbers. This is done according to the following formula:

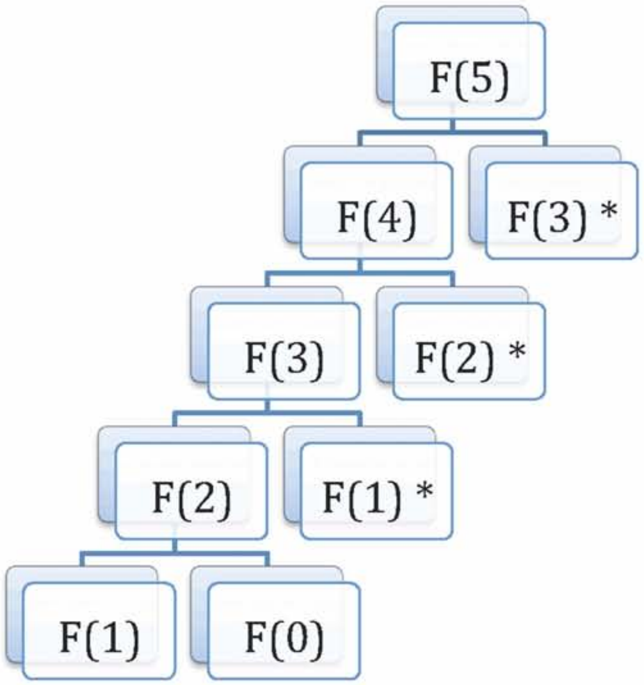
$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n \geq 2 \end{cases}$$

A straightforward implementation of this computation would take very long to run. The running time for F(n) is exponential in n. We can see very quickly this by observing that all the leaves of the recursion tree return 0 or 1, so the tree for F(n) will have at least F(n) leaves, and $F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$.

As Srini said in lecture, exponential = bad, polynomial = good. If we draw the recursion tree for F(5) we see a lot of waste: F(3) is re-computed twice, and F(2) is re-computed 4 times.



Memoization addresses this waste by storing F(n) once it is computed, and reusing its value. With memoization in place, computing F(n) takes O(n). The easiest way to visualize this result is re-drawing the recusion tree above, under the assumption of memoization.



I have used * to denote calls to F(n) that return immediately by re-using a previously computed results.

As a last word on the Fibonacci problem and memoization, notice that both the original and the memoized implementation have a recursion depth of O(n).