Lecture 12: Dynamic Programming Part 1
10:00 AM, Feb 21, 2018

Contents

1 Introduction 1
2 Fibonacci 2

Objectives

By the end of these notes, you will know:

- the top-down and bottom-up approaches to dynamic programming

By the end of these notes, you will be able to:

- apply dynamic programming to implement faster solutions to problems with recursive solutions than a naive recursive implementation

1 Introduction

Dynamic programming (DP) is a technique for solving problems whose solutions can be derived from the solutions to their subproblems. “Dynamic” refers to the idea that what the algorithm computes at each step is based on its earlier computations; “program” refers not to a computer program, but rather to a schedule of activities. DP is not one specific algorithm. It is an algorithmic strategy (like divide-and-conquer is an algorithmic strategy) that is often applied to solving optimization problems: e.g., path planning.

DP is applicable to problems with recursive solutions, specifically when the subproblems into which they decompose overlap. Subproblems overlap when they can be used to solve multiple “super” problems. For example, the \( n - 2 \)nd Fibonacci number is used to compute both the \( n - 1 \)st and the \( n \)th Fibonacci number. The key insight that makes DP efficient is that overlapping subproblems are not solved and re-solved repeatedly; rather intermediate solutions are calculated once, and then stored, so they can be accessed later as needed.

DP comes in two flavors: top-down and bottom-up. When using the top-down approach, we use recursion as usual, with two straightforward enhancements: (i) before solving a problem, we check our store (i.e., the data structure in which we are storing intermediate solutions) to see whether we have already solved it; and (ii) after solving a problem, we make sure to update our store with the new solution. This storing of intermediate solutions for potential later use (rather than recalculating them as needed) is called memoization.

On the other hand, the bottom-up approach is not recursive; it is iterative. Instead of solving larger problems in terms of smaller ones, as a recursive solution would, we first solve the smaller problems,
and then use those solutions to build up solutions to larger problems. The solutions to smaller problems are again stored, and the solutions to the bigger problems are calculated by looking up those stored solutions.

2 Fibonacci

To introduce this new topic, we’re going to revisit to an old (but good) topic: the Fibonacci numbers. Remember how the Fibonacci sequence is defined:

\[
F(n) = \begin{cases} 
0, & \text{if } n = 0 \\
1, & \text{if } n = 1 \\
F(n - 1) + F(n - 2), & \text{otherwise}
\end{cases}
\] (1)

Take a careful look at this recurrence relation. To compute \(F(n)\), we must compute \(F(n - 1)\) and \(F(n - 2)\). And then to compute \(F(n - 1)\) we must compute \(F(n - 2)\) and \(F(n - 3)\). So the computation of both \(F(n)\) and \(F(n - 1)\) require that we compute \(F(n - 2)\). But in our naive implementation of \(\text{fib}\) in CS 17, we forgot all about \(F(n - 2)\) after computing \(F(n - 1)\), and just recomputed it when we needed it for \(F(n)\). Pretty silly. And not just silly: expensive!

Without DP:

With memoization:
The calls with asterisks are the only ones that require computation since all the non-marked calls will have their values stored already. The $\text{fib}(0)$ and $\text{fib}(1)$ calls on the left branch are base cases, so we don’t need to find their value, and we’ve already found the $\text{fib}(2)$ call on the right branch from the call to $\text{fib}(2)$ on the left branch.

Below, we try two different dynamic-programming approaches, one top-down and the other bottom-up, to calculate the $n$th Fibonacci number. Both of these programs use an array to store solutions to subproblems. As per the Fibonacci recurrence relation, the 0th and 1st elements of the array are initialized to 0 and 1, respectively. Rather than make an array that holds integers, and use a sentinel value to indicate when a subproblem has yet to be computed, we’ll use Java’s `OptionalInt` type, which either contains an `int` or a special empty value. If we call the `get` method on an `OptionalInt` that is empty, Java will throw an exception—preventing a “silent failure” condition that might arise from using sentinel values. (We’ll see other ways to initialize the array in the next lecture.)

Both our top-down and bottom-up implementations implement `IFibonacci`, an interface for Fibonacci numbers, which declares only one method `fib`, with parameter $n$, which returns the $n$th Fibonacci number.

**Top-Down Fibonacci**  
Our naive implementation of `fib` calls itself an exponential number of times; but it only computes $n$ Fibonacci numbers! This redundancy is clearly unnecessary. A better solution would simply remember the answers to the questions it was asked previously and return those answers (in constant time) when those same questions are asked again. This idea is called *memoization*; it is a technique for improving the run time of an algorithm.

An important step when taking a top-down approach to DP is to figure out what to memoize: i.e., what information to store for later use. In the case of `fib`, this step is pretty clear: we should memoize solutions to smaller problems. Since smaller problems are characterized by integers from 0 to $n - 1$, we’ll just use an array of size $n$.

For example, once computed, the 0th entry in such an array would contain 0; the 1st entry would contain 1; $\ldots$, the 5th entry would contain 5; the 6th entry would contain 8; and so on.
We implement this top-down approach to DP below. As per the Fibonacci recurrence relation, the 0th and 1st elements of the array are initialized to 0 and 1, respectively.

Now, when `fib` is called on input `n`, if the `n`th Fibonacci number has been memoized (i.e., if `fibComputed[n]`), then the memoized value is returned immediately. If it has not, the `n`th Fibonacci number is calculated (by calling `fib(n-1)` and `fib(n-2)`, which may have been stored in `fibArray` for later use), and then returned.

This DP algorithm runs in time $O(n)$, because `fib` is never called twice on the same input, and there are only $n$ possible inputs.

```java
package lec12;

import java.util.OptionalInt;

/**
 * A class that computes the Fibonacci numbers using top-down dynamic programming.
 */
public class TDFibonacci implements IFibonacci {

    @Override
    public int fib(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        OptionalInt fibTable[] = new OptionalInt[n+1];
        for (int i=0; i<=n; i++) {
            fibTable[i] = OptionalInt.empty();
        }
        fibTable[0] = OptionalInt.of(0);
        fibTable[1] = OptionalInt.of(1);
        fibTable[2] = OptionalInt.of(1);

        return fibHelper(n, fibTable);
    }

    int fibHelper(int n, OptionalInt[] fibTable) {
        if (!fibTable[n].isPresent()) {
            fibTable[n] = OptionalInt.of(fibHelper(n - 1, fibTable) + fibHelper(n - 2, fibTable));
        }
        return fibTable[n].getAsInt();
    }
}
```
Caching is a generalization of memoization. In English, a cache is a hiding place for storing valuables, like jewelry or ammunition. In CS, a cache is not a hiding place, but it is a place for storing valuables—not jewelry or ammunition, but values you expect to need again later, and don’t want to have to retrieve again from their original source (e.g., a computation or an image retrieved from a web server). By caching values you save time precisely because you do not have to retrieve those values again. Because a cache is of finite size, the problem of what values to store in a cache is an important, and well-studied, optimization problem. (Take CSCI 167 for more information about caching strategies.)

**Bottom-Up Fibonacci** So far, we have taken a top-down approach: we have used memoization to speed up the original recursive algorithm. What we will now do is take a bottom-up approach. To do so, we will design a brand new algorithm that works, as the name suggests, from the bottom up. This means that we will solve the subproblems first, store their solutions in a table, and when all the subproblems have been solved, we will use that table to solve the problem itself.

In the recursive solution with memoization, we created a table and let the recursive algorithm store values in that table in whatever order naturally arose from its sequence of recursive calls. Now, we will pick the order in which values are calculated (and hence stored) more intelligently. That is, we will ask the question: “in what order should we fill in the table?” In the Fibonacci case, the answer is rather trivial: since the value at each index depends on the availability of the values at the two previous indices, we can simply compute the values in increasing index order, starting from 0.

This is a very natural way to compute $fib(n)$: Create an array of size $n + 1$, set the 0th element to be 0 and the 1st element to be 1, then compute the 2nd from those, then compute the 3rd from the entries you have already computed, and so on. Here is the corresponding implementation:

```java
package lec12;

import java.util.OptionalInt;

/**
 * A class that computes the Fibonacci numbers using bottom-up dynamic programming.
 */
class BUFibonacci implements IFibonacci {

    @Override
    public int fib(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;

        OptionalInt[] fibTable = new OptionalInt[n+1];
        for (int i=0; i<=n; i++) {
            fibTable[i] = OptionalInt.empty();
        }
        fibTable[0] = OptionalInt.of(0);
        fibTable[1] = OptionalInt.of(1);
        fibTable[2] = OptionalInt.of(1);
        for (int i = 2; i <= n; i++)
            fibTable[i] = OptionalInt.of(fibTable[i - 1]).
```
Computer scientists often trade off time efficiency for space efficiency. Our DP implementations of \texttt{fib} run much faster than a pure and simple recursive implementation would, but they use more space. But if we take a closer look at our bottom-up DP solution to \texttt{fib}, we find that we only actually need the previous two numbers to compute the next number. (Recall fast-fib, the CS 17 program for computing Fibonacci numbers that used an accumulator.) Thus, there is an even more space-efficient implementation that doesn’t require an entire array, but only 2 variables! When taking a bottom-up DP approach to solving a problem, if you find that you don’t continually need the solutions to all subproblems, you should feel free to throw away the ones you don’t need any more.

At this point, you might be wondering: Which approach to DP is better: top-down or bottom-up? The answer is unclear. Theoretically, the top-down approach is always at least as good as the bottom-up approach in terms of its asymptotic time complexity. For example, suppose our recurrence relation were \( F(n) = F(n - 10) + F(n - 20). \) Here, the top-down approach only computes the necessary values, whereas the bottom-up approach computes all values. But the bottom-up approach can often reveal opportunities for improved space-efficiency.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: \url{http://cs.brown.edu/courses/cs018/feedback}.