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

Contents

1 A Motivating Problem: Fibonacci Numbers 1
2 Top-Down Dynamic Programming 2
3 Bottom-Up Dynamic Programming 3
4 The Architecture of a DP Solution 4

Objectives

By the end of these notes, you will know:

- what dynamic programming is
- the difference between top-down and bottom-up dynamic programming

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

- convert a naive recursive program to a version that uses dynamic programming solution to reuse repeated computations

1 A Motivating Problem: Fibonacci Numbers

The Fibonacci sequence is a well-known sequence of numbers in mathematics in which each number is the sum of the two numbers before it (the sequence starts with 0 and 1 as its base cases). Specifically, the sequence looks like: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Computing the $n^{th}$ Fibonacci number is a standard recursive problem in computer science:

```java
public int fib(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

What calls get made if we use this definition to compute $\text{fib}(4)$?
Notice that we repeat several calls. Since there are no assignment operations in this code, \texttt{fib} returns the same output every time it receives the same input. Thus, there is potential for a lot of wasted computation here. We could make our implementation more time-efficient by storing the results of \texttt{fib} each time it is called, then re-using the stored result if we call the function with the same inputs again later. When computing \texttt{fib(4)}, for example, each of the shaded calls below could get looked up, rather than recomputed.

In this lecture and the next, we are going to look at a technique called \textit{dynamic programming} that is used to implement such an optimization for recursive functions.

## 2 Top-Down Dynamic Programming

At the heart of dynamic programming is a simple idea: as the program runs, maintain an additional data structure that stores the output computed for each unique input. If an input is requested a second time, return the stored value.

Here is a version of the \texttt{fib} function with this added table. Since the inputs to \texttt{fib} are numbers, we can use an array to implement the table. The contents of the \texttt{i}th slot in the array would be the output (if any) previously computed on input \texttt{i}.

Within the table, we need a way to distinguish between values that have been stored and values that have not been set. This means we need a default value for the array. While \texttt{null} might seem like the right approach here, many languages instead use something called an \textit{option} or \textit{option type} to say “a specific type for which a value may not have been provided yet”. This carries more information than \texttt{null}, and saves some error checking around using \texttt{null}. The source code for today shows the \texttt{null} version as well so you can see the differences.

```java
OptionalInt[] fibTable2;
```
```java
public int fibTable1OptInt(int n) {
    fibTable2 = new OptionalInt[n+1]; // set the table size
    // put default values in the table
    for (int i=0; i <= n; i++) {
        fibTable2[i] = OptionalInt.empty();
    }
    return fibTable1HelperOptInt(n);
}

private int fibTable1HelperOptInt(int n) {
    if (fibTable2[n].isPresent()) // value has been computed before
        return fibTable2[n].getAsInt();
    else {
        int result;
        if (n == 0)
            result = 0;
        else if (n == 1)
            result = 1;
        else
            result = fibTable1HelperOptInt(n-1) +
                    fibTable1HelperOptInt(n-2);
        fibTable2[n] = OptionalInt.of(result);
        return result;
    }
}
```

The approach in this code, which keeps the original recursive solution structure but holds onto the results as they get computed, is called top-down dynamic programming. The top-down part comes from letting the algorithm run from the top-most input down to the base cases, filling in the table as values are completed. Top-down dynamic programming requires few modifications to the original recursive source code.

**Note to those from CS19:** This looks a lot like memoization, as you covered in CS19. The difference is that memoization should not require you to modify the code to add the storage. Instead, you leave the function to memoize intact, and pass it as an argument to another function that wraps the table infrastructure around the function. Passing an arbitrary method as an argument is very difficult in Java, so we modify the code as shown here.

### 3 Bottom-Up Dynamic Programming

As the algorithm runs, the table gets populated with already-computed values. As an alternative, we could also populate the table from the bottom up, starting from the base cases and progressing upwards to the final answer. This is called bottom-up dynamic programming, and it has the following code:

```java
OptionalInt[] fibTable3;

public int fibTableBU(int n) {
    fibTable3 = new OptionalInt[n+1]; // set the table size
    if (n == 0)
        ...
The bottom-up version looks less like the previous version, which retained the general naive-recursive solution code. There are tradeoffs between these two approaches: top-down may be less error-prone to implement (in languages like Racket and OCaml that allow functions as arguments the original code doesn’t have to get edited at all). Bottom-up can use less space, however, as sometimes only part of the table needs to be maintained.

For CS18, all we care about is that you can produce some implementation of dynamic programming. We’ll guide you through a specific one in lab, but you may use either one on the homework. Both versions yield the same table (though top-down may skip some entries if they aren’t encountered during execution).

4 The Architecture of a DP Solution

Whether you work top-down or bottom-up, the heart of one of these problems lies in the recurrence relation that summarizes the relationship between subproblems. For Fibonacci, that recurrence relation looks like the following:

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

In the case of Fibonacci, the recurrence looks a lot like the code for the recursive solution. For more complicated problems, an implementation would add more detail than the recurrence, but the heart of the problem would be captured in the recurrence.

All DP solutions share a common set of methods, and common steps for writing them. Here’s a summary:
1. Write out a couple of small examples by hand to identify the (recursive) computation that lets you build up partial solutions from smaller solutions

2. Capture the relationship between subproblems in a naive recursive program or recurrence relation

3. Create a table to hold the results of subproblems

4. Initialize the table cells to default values

5. Fill the table, using either a top-down or bottom-up approach
   - top-down follows the recursive computation from the original input, saving intermediate values as they are computed
   - bottom-up iteratively fills in the table from the base cases, working backwards to the original input

6. Look up the final answer from within the table

In lab, we will give you a template of how to structure a Java class with a collection of methods that implement these steps.

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: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)