Lecture 26: Dynamic Programming Part 1

11:00 AM, Mar 22, 2021

Contents

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 a dynamic programming solution to reuse repeated computations

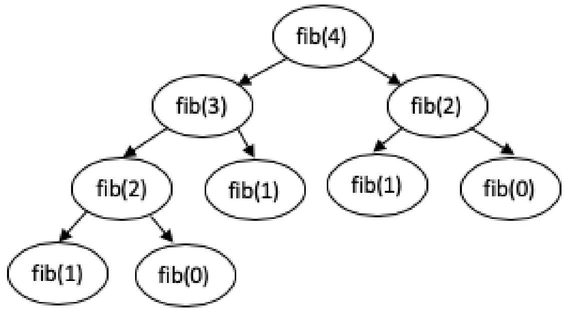
1 A Motivating Problem: Fibonacci Numbers

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: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...$

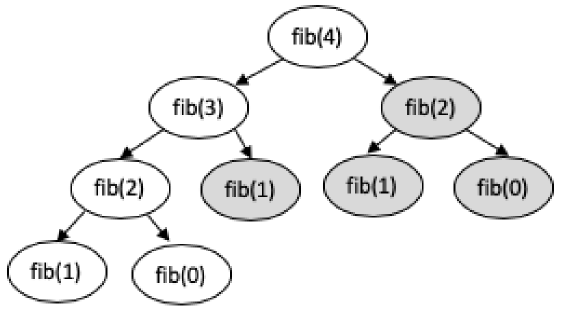
Computing the n^{th} Fibonacci number is a standard recursive problem in computer science:

```
def fib(n: Int): Int = {
    if (n == 0) 0
    else if (n == 1) 1
    else fib(n - 1) + fib(n - 2)
  }
}
```

What calls get made if we use this definition to compute `fib(4)`?



Notice that we repeat several calls. Since there are no assignment operations in this code, `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 `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 `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 *dynamic programming* that is used to implement such an optimization for recursive functions.

2 Dynamic Programming: The Core Idea

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.

This raises a question: what data structure should we use for storing the computed values? The data structure needs to let us map function inputs to their computed outputs. This sounds like a hashtable, and a hashtable would indeed work here. However, in the specific case of fibonacci, we can observe that the inputs are consecutive integers ranging from 0 to the initial input to the function. In this case, we could also use an array: each input corresponds to an array index, and we could store the result of `fib(i)` in the i^{th} index of the array. We will use an array in these notes, noting that the hashtable is the more general solution.

3 Top-Down Dynamic Programming

Let's rework our original `fib` function to maintain an array of previously computed values. As an overivew, we have to extend the code in the following way:

```
// set up an array (call it computed) to hold the computed results

def fib(n: Int): Int = {
    // if there is a result in computed(n), return it
    // otherwise run the original computation, saving it as \code{result}
    if (n == 0) 0
    else if (n == 1) 1
    else fib(n - 1) + fib(n - 2)
    // store result in computed(n)
    // return result
  }
}
```

We'll put all of this code in a class (which puts the array along with the function). We'll present the code, then discuss the nuances:

```
 class FibonacciTopDown(val maxInput: Int) {
  // make a data structure to store computed values of fib
  private val computed: Array[Option[Int]] = Array.fill(maxInput + 1)(None)

  def fib(n: Int): Int = {
    // ask whether we've computed answer for n yet
    computed(n) match {
      case Some(result) => result
      case None => {
        println("Computed " + n)
        val result =
          if (n == 0) 0
          else if (n == 1) 1
          else fib(n - 1) + fib(n - 2)
        computed(n) = Some(result)
        result
      }
    }
  }
}
```

Notes:

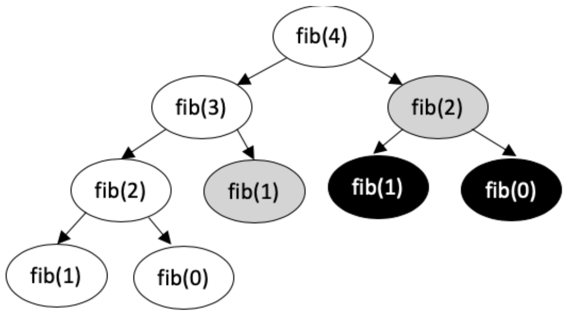
- The class takes an input `maxInput`, which is the largest value for which we will optimize the fibonacci computation (we aren't dealing here with larger inputs).
- The array contains `Option[Int]`, which we have discussed as a common way to represent values that may or may not yet exist.
- The `Array.fill` initializes every space in the array to `None`.

- The array has size `maxInput + 1` because we need to be able to store computed results for 0...maxInput.
- We use a match expression to distinguish Some and None values when checking whether a value has previously been computed.

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 (we see here that the body of the original `fib` function is verbatim within the `None` case).

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.

Does the top-down solution achieve our original goal of calling `fib` at most once on each input? Recalling our trees of calls to `fib` from earlier, do we indeed only make the non-shaded calls? Actually, the following image shows what happens here: the gray-shaded calls are made, but the if-expression (and its recursive calls) doesn't get evaluated. The black-shaded calls aren't made at all.



4 Bottom-Up Dynamic Programming

Another approach to populating the array would be to work from the base cases up. We'd fill in the base cases, then fill every other value in the array as an initial step. Then, when someone asks for a specific value of `fib`, we just look it up in the table. Here's the corresponding code:

```
class FibonacciBottomUp(val maxInput: Int) extends IFib {
  val computed: Array[Option[Int]] = Array.fill(maxInput+1)(None)

  // setup the computed values
  if (maxInput >= 0)
    computed(0) = Some(0)
  if (maxInput >= 1)
    computed(1) = Some(1)
  if (maxInput >= 2) {
    for (k <- 2 to maxInput) {
      computed(k) = Some(computed(k - 2).getOrElse(-1) + computed(k - 1).
          getOrElse(-1))
    }
  }

  // lookup answers as called
  def fib(n: Int): Int = {
    computed(n) match {
      case Some(result) => result // already computed, so return that answer
      case None => {
        throw new RuntimeException("fib -- value not computed")
      }
    }
  }
}
```

This is called *bottom-up dynamic programming*. The version makes only the un-shaded (gray or black) calls from our `fib` tree diagrams.

The bottom-up version looks less like the top-down version, which retained the general naive-recursive solution code. The latter can be less error-prone to implement as a result, but which version is better depends on the details of the function you are trying to optimize.

For CS18, all we care about is that you can produce *some* implementation of dynamic programming. We'll guide you through specific examples in class, 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).

5 The Architecture of a DP Solution

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, though we have not shown that here)
3. Create a table (array or hashtable) 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

We will work more on this in the next lecture.

6 Study Questions

1. Explain why the gray-shaded calls in the fib-call tree happen in the top down solution, but not the black-shaded ones
2. In big-O terms, is there a difference between the number of calls to fib in the top-down and bottom-up approaches?
3. If you switched the recursive calls to compute `fib(n-2)` before `fib(n-1)`, in what order would the indices of computed get filled?
