**CSCI 4041: Algorithms and Data Structures**
**Memoization**

*Last revision March 22, 2020*

Here's the agenda.

Dynamic programming.
The Fibonacci series.
Computing the Fibonacci series recursively.
How to compute faster using memoization.

This is the first in a series of lectures about *dynamic programming* and how it can construct *optional binary search trees* (OBST's). For some background about dynamic programming, please read pages 359–360 of Cormen. For OBST's, please read 397–404. The algorithm that constructs OBST's is fairly detailed, so we'll construct it a little at a time, over several lectures. Part of that algorithm will involve a technique called *memoization*, a term Cormen doesn't use, but which is still important. That's what I'll discuss here.

**Dynamic programming.** On page 359, Cormen talks about dynamic programming as being a four-step process. This is true, but I want to talk about it a little differently. A dynamic programming algorithm starts by computing a series of small results, each in an optimal way (for some definition of the word *optimal*). Then it combines the small results into bigger results, also in an optimal way, continuing until only one big optimal result is left—which is what the algorithm was designed to compute.

One problem is that the same result may be computed more than once. If that happens, then the dynamic programming algorithm will waste time computing it, and will run slowly. For the algorithm to run fast, it needs some way to avoid computing the same result repeatedly. That's what memoization does. Here we'll apply memoization to a familiar problem: computing the Fibonacci series recursively.

**The Fibonacci series.** You've seen the *Fibonacci series*, discovered by 13th century Italian mathematician Leonardo Pisano, also called Leonardo Fibonacci (his name means *Leonardo the son of Bonaccio*). It's an infinite series of nonnegative integers. The first two are 0 and 1; each of the others is the sum of the previous two. If $F_n$ is the $n$th Fibonacci number, then we can define it using the following recurrence.

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Here are the first few Fibonacci numbers. We'll stop at $F_{46}$ because it's the largest Fibonacci number that fits in a 32-bit two's complement integer, which is what many programming languages use.

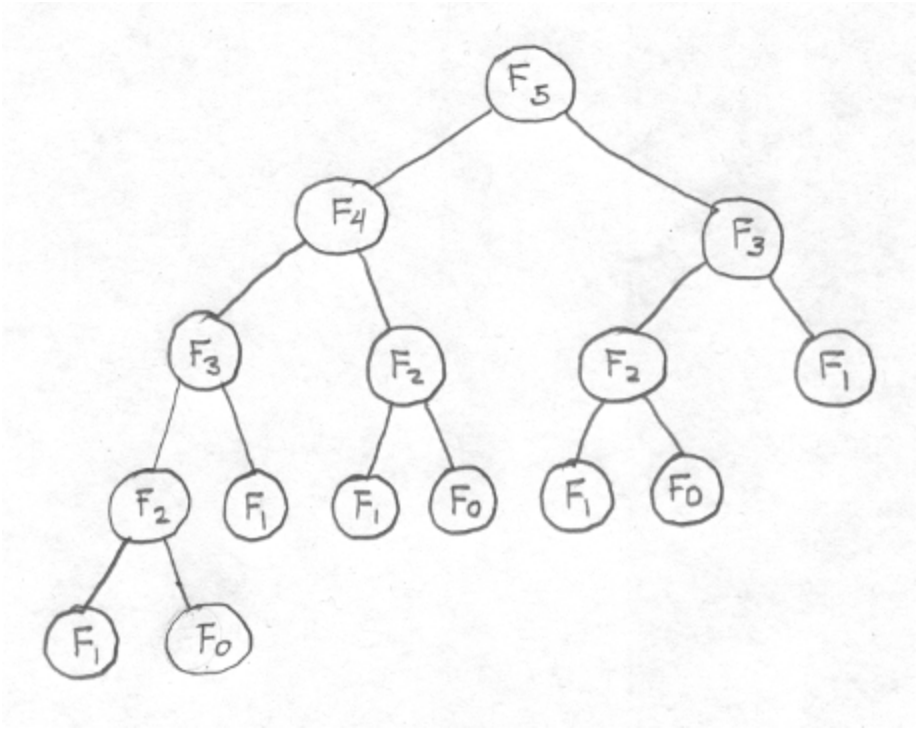
$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_2 &= 1 \\ F_3 &= 2 \\ F_4 &= 3 \\ F_5 &= 5 \\ &\vdots \\ F_{46} &= 1836311903 \end{aligned}$$

**Computing the Fibonacci series recursively.** If we want to compute Fibonacci numbers, then we can easily translate the recurrence into a recursive procedure. Here's such a procedure in Cormen's pseudocode, called FIBONACCI. Let's assume we'll never call FIBONACCI with an integer less than 0.

```
1 FIBONACCI(n)
```

```
2   if n == 0
3      return 0
4   else if n == 1
5      return 1
6   else
7      return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

This procedure is very slow. To see why, let's draw a tree of the procedure calls made by FIBONACCI(5). It looks like this.




To compute $F_5$, we must compute $F_4$ and $F_3$. To compute $F_4$, we must compute $F_3$ and $F_2$, etc. Except for the calls at the bottom of the tree, each Fibonacci number is computed by making two more calls. As a result, we may guess that FIBONACCI runs in $O(2^n)$ time, although it's possible to get a tighter bound, $O(1.6^n)$.

Either way, FIBONACCI runs in exponential time. That means it's far too slow, even on a modern computer, when $n$ is large. About the only thing it's good for is to show students in an elementary programming course how recursion works.

**Memoization.** How can we make FIBONACCI faster? The obvious way is to rewrite it using a loop instead of a recursion. Then we can make it work in only $O(n)$ time. Let's pretend, however, that we don't know how to do that—the only way we know how to write FIBONACCI is to use recursion. Can we still make it faster? (Of course I wouldn't ask unless the answer was *yes*.)

Here's how we do it. If we look at the tree, we see that the same Fibonacci numbers are being computed over and over again. For example, $F_3$ is computed twice, $F_2$ is computed three times, etc. We could make the algorithm run faster if we never computed these more than once.

How could we do that? Suppose that every time we compute a partial result, we store it in a data structure, like an array. Whenever we need a result, we check the array to see if we've computed that result before. If we have, then we just get the result from the array, instead of computing it again.

This is called *memoization,* because it works by remembering previously computed results. (Be careful about spelling: it's not *memo**R**ization:* there's no ***R*** in it.) *Memoization* is a programming technique often used in pure functional programming to speed up certain computations. If you've taken CSCI 2041, then you may have already seen it. We won't use pure functional programming when we discuss dynamic programming and OBST's, but we can still use memoization.

We'll use an array $m[0 : 46]$ to store previously computed Fibonacci numbers, so that $m[n] = F_n$. The procedure SET-UP-MEMY-FIBONACCI initializes $m$. We know $F_0$ and $F_1$ already, so we'll set $m[0]$ and $m[1]$ to them. All the other elements of $m$

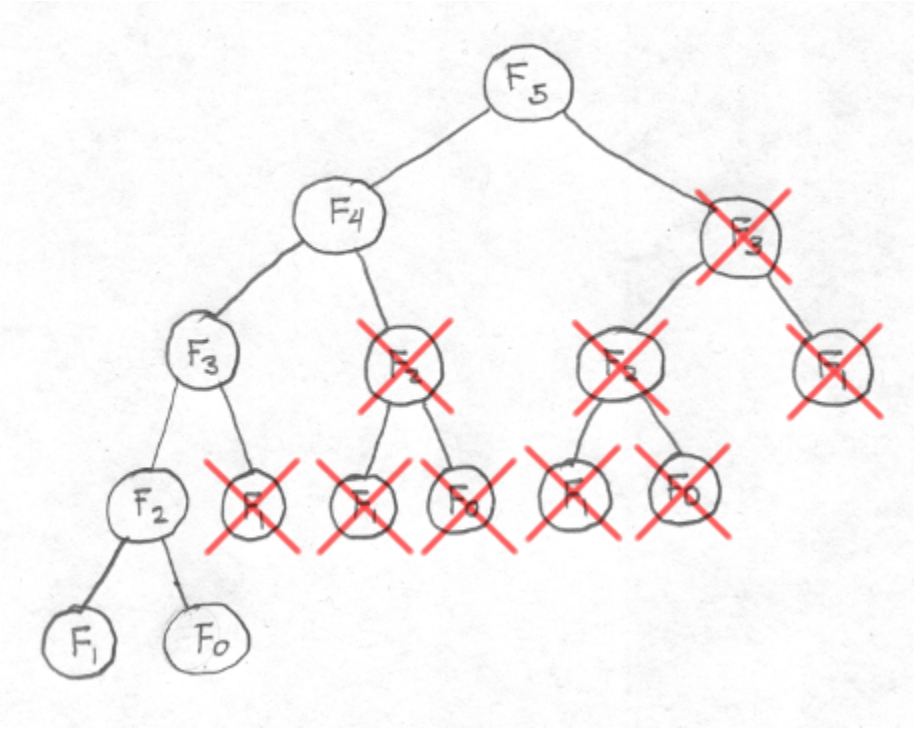
are set to −1, which means that we haven't computed them yet. We use −1 because it can't be a Fibonacci number.

```
1 SET-UP-MEMY-FIBONACCI(m)
2    m[0] = 0
3    m[1] = 1
4    for f = 2 to 46
5       m[f] = −1
```

We can now define MEMY-FIBONACCI, a memoized version of the original procedure FIBONACCI. It assumes we've already called SET-UP-MEMY-FIBONACCI to initialize $m$. When we call MEMY-FIBONACCI with $n$, it tests if we've computed $F_n$ before, so its value is stored in $m$. If it is, then it returns that value without computing $F_n$ again. If it isn't, then MEMY-FIBONACCI computes $F_n$ recursively, but stores it in $m$ to avoid having to compute it again. Note that the base cases of the recursion are also handled by testing $m$.

```
1 MEMY-FIBONACCI(n)
2    if m[n] < 0
3       m[n] = MEMY-FIBONACCI(n − 1) + MEMY-FIBONACCI(n − 2)
4    return m[n]
```

Because of memoization, MEMY-FIBONACCI never computes any Fibonacci number more than once. If we draw its calls in a tree, as we did before, we find that most of those calls now never need to be made. We've drawn scary red X's over them.




This tree suggests that only the calls along a linear path from $F_5$ to $F_1$ and $F_0$ are being made. That in turn suggests that MEMY-FIBONACCI takes $O(n)$ time to compute $F_n$, even though it uses the same recursion that FIBONACCI did. In other words, we've used memoization to convert an exponential time algorithm into a linear time algorithm! It's hard to imagine better results than that, unless we decide not to compute Fibonacci numbers at all.

Okay, I admit it, we don't really care about computing Fibonacci numbers. So why am I talking about them? We'll see later that the dynamic programming algorithm for constructing OBST's (which is what we really care about) will use memoization to run fast.