Calculating Fibonacci numbers

The well-known Fibonacci numbers are defined recursively in the box to the right. The sequence of Fibonacci starts off like this:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Fibonacci numbers
F(0) = 0. F(1) = 1.
For n > 1, F(n) = F(n-1) + F(n-2)

This discussion of the Fibonacci numbers provides an interesting look at the development of algorithms and data structures, with interesting historical tidbits. Moreover, Fibonacci numbers are connected with numbers called the golden ratio and golden angle, they have connections with architecture, and they appear in various ways in nature. Another pdf file in this JavaHyperText treats discusses these topics.

Some of this material is taken from en.wikipedia.org/wiki/Fibonacci_number. Also, there is so much interest in Fibonacci numbers that there is a journal devoted to it. *The Fibonacci Quarterly*, started in 1963, is an official publication of the Fibonacci Association. All but the past 5 volumes are free. Here is its website: www.fq.math.ca/list-of-issues.html.

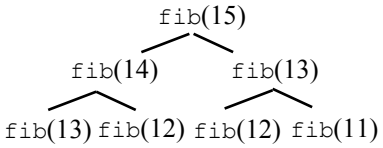
Many people think that Fibonacci numbers first appeared in Fibonacci's historic book on arithmetic, *Liber Abaci* (*The Book of Calculation*), in 1202. Actually, Fibonacci was not his real name! That name was given to him by a writer well after his death. When he was living, he was known as Leonardo of Pisa. Moreover, the Fibonacci numbers were used and discussed in ancient Sanskrit texts in India as early as 450BC–200BC. A paper by Parmanand Singh, written in 1985, on the early introduction of Fibonacci numbers in Sanskrit texts can be found in the JavaHyperText entry for Fibonacci.

A naïve implementation of Fibonacci numbers

Function `fib` to the right is the obvious translation of the definition of function F into Java. It is also the worst way to compute it. Consider computing fib(15), as shown in the tree to the right. That call requires calling `fib`(13) and `fib`(14). These two calls then require calling `fib`(13), `fib`(12) twice, and `fib`(11). You can imagine the next level. In total,

```
/** = F(n).
  * Precondition: n ≥ 0. */
public static int fib(int n) {
  if (n <= 1) return n;
  return fib(n-1) + fib(n-2);
}
```

`fib`(13) is called 2 times,
`fib`(12) is called 3 times,
`fib`(11) is called 5 times,
`fib`(10) is called 8 times



Hey, the number of times each is called forms the Fibonacci sequence!

We prove that the time complexity of `fib`(`n`) is in O(2^n). We first write a (recursive) function `f` that gives an upper bound on the number of basic steps taken in computing `fib`(`n`). Constant `c` is an upper bound on the number of basic steps in the case `n` ≤ 1 and the basic steps needed besides the recursive calls in the case `n` ≥ 2.

`f`(0) = `c`
`f`(1) = `c`
For `n` ≥ 2, `f`(`n`) = `c` + `f`(`n`–1) + `f`(`n`–2)

To the left is a proof by mathematical induction that this function is in O(2^n).

Actually, O(2^n) is not the tightest bound for recursive function `fib`. The tightest bound is O(φ^n), where φ is the golden ratio:

$\varphi = (1 + \sqrt{5})/2 = 1.6180339887\ldots$

Fibonacci numbers and the golden ratio are intricately connected, and you will hear more about them later.

Theorem. fib(n) ∈ O(2^n).
Proof. We prove that f(n) ≤ c 2^n for n ≥ 0, where c is given in the definition of f(n) to the left. The proof is by induction on n. We have:

f(0) = c ≤ c 2^0 (since $2^0 = 1$).
f(1) = c ≤ c 2^1 (since $2^1 = 2$).

Assume that n ≥ 2. Assuming f(k) ≤ c 2^k for k < n, we prove f(n) ≤ c 2^n. We start with the definition of f(n):

c + f(n-1) + f(n-2)
= <inductive hypotheses>
c + a2^{n-1} + a2^{n-2}
= <arithmetic>
c (2^{n-1} +1 + 2^{n-2})
≤ <for n ≥ 2, 1+2^{n-2} ≤ 2^{n-1}>
c (2^{n-1} + 2^{n-1})
= <arithmetic>
c 2^n

Calculating Fibonacci numbers

Caching

A *cache* is a collection of items stored away in some oft-hidden place. It's a stockpile, a store. One hears, for example, of an *arms cache*. Caches are used in many places in computing. In hardware, for example, a computer may have a *memory cache*, a small place close to the CPU for oft-used words of memory, to make it quicker to retrieve them. Your browser uses a cache of lately used web pages, so they don't have to be retrieved so often.

We can modify function `fib` to save computed values in a cache. This cache is implemented as a static `ArrayList`, shown to the right.

```
/** For 0 ≤ n < cache.size, F(n) is cache[n]. If fibCached(k)
  * has been called, its result in in cache[k] */
public static ArrayList<Integer> cache= new ArrayList<>();
```

The modified method `fibCached` appears to the right. The first if-statement returns the value `F(n)` if it has been computed before. Note that after the value of `F(n-2) + F(n-1)` is calculated and stored in `ans`, it is placed in the cache.

At worst, `fibCached(n)` takes time linear in `n`, but future calls with the same `n` take constant time. However, the space requirement is O(`n`) where `n` is the largest value for which `fibCached (n)` was called, and this space stays there as long as the program is running.

```
/** = F(n).   Pre: n >= 0. Use the cache. */
public static int fibCached(int n) {
   if (n < cache.size()) return cache.get(n);
   if (n == 0) { cache.add(0); return 0; }
   if (n == 1) { cache.add(1); return 1; }

   int ans= fibCached(n-2) + fibCached(n-1);
   cache.add(ans);
   return ans;
}
```

Computing Fibonacci in linear time

There's no need to use recursion to calculate `F(n)`! Instead, write a simple loop as in method `fibit`, which appears to the right. At each iteration, `p` = `F(k-2)` and `c` = `F(k-1)`, so `F(k)` can be calculated as their sum.

```
/** = F(n), for n >= 0. */
public static int fibit(int n) {
  if (n <= 1) return n;
  int k= 2; int p= 0;   int c= 1;
  // invariant: p = F(k-2) and c = F(k-1)
  while (k < n) {
      int Fk= p + c;   p= c;  c= Fk;
      k= k+1;
   }
   return p + c;
}
```

Computing Fibonacci in log time

The first equation to the right shows how to calculate `F(2)` by multiplying the vector containing `F(0)` and `F(1)` by the 2 x 2 matrix. The second equation to the right above shows how to calculate `F(n)` and `F(n+1)` by first raising the matrix to the power `n`. Since `F(0)` = 0 and `F(1)` = 1, we can use the equation to the right to calculate `F(n)` and `F(n+1)`.

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F(0) \\ F(1) \end{pmatrix} = \begin{pmatrix} F(1) \\ F(2) \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F(0) \\ F(1) \end{pmatrix} = \begin{pmatrix} F(n) \\ F(n+1) \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} F(n) \\ F(n+1) \end{pmatrix}$$

You know a logarithmic algorithm to compute something to the power `n`! Look it up in JavaHyperText under entry *Power* or *Exponentiation*. Therefore, we can use this technique to calculate `F(n)` in logarithmic time. Neat! It gets better and better! This computation was shown by David Gries and Gary Levin in a 2-page paper in 1980.

Another logarithmic algorithm to compute Fibonacci numbers depends on the golden ratio φ and its conjugate Φ, sometimes called the silver ratio:

$$\varphi = (1 + \sqrt{5})/2 = 1.6180339887\ldots \qquad \Phi = (1 - \sqrt{5})/2 = -.6180339887\ldots$$

They are the roots of the polynomial $x^2 - x - 1$. They satisfy this equation: $F(n) = (\varphi^n - \Phi^n)/\sqrt{5}$. This equation yields another logarithmic-time algorithm to calculate `F(n)`, since φ^n and Φ^n can each be calculated in logarithmic time.

Interestingly enough, $\lim_{n\to\infty} F(n+1)/F(n) = \varphi$, although the convergence is very slow. Therefore, if `F(n)` is known, `F(n+1)` may be calculated in constant time.