Fibonacci Numbers

The Fibonacci numbers are defined by the recurrence

$$F_n = F_{n-1} + F_{n-2} \quad \forall n \geq 2, F_0 = 0, F_1 = 1.$$

The Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$.

In this note, we discuss algorithms for computing the n^{th} Fibonacci number and prove some related results.

1 A naïve recursion

Algorithm 1 A naïve recursive algorithm to compute F_n

```
int fib(int n)
if n is 0 then
  return 0
end if
if n is 1 then
  return 1
end if
return fib(n-1)+fib(n-2)
```

This algorithm is (woefully) inefficient for at least a couple of reasons. From an implementation perspective, without tail recursion, it is easy for this implementation to exhaust stack space allocated to the program. From a broader theoretical perspective, the number of addition operations required is as high as F_n itself. *(Why?)*

We can bound F_n as follows using mathematical induction (see Section 4 for the detailed proof):

$$\forall n \geq 6 \quad 2^{n/2} \leq F_n \leq 2^n.$$ [1]

[1] Also note the use of the universal quantifier $\forall$ in this example.

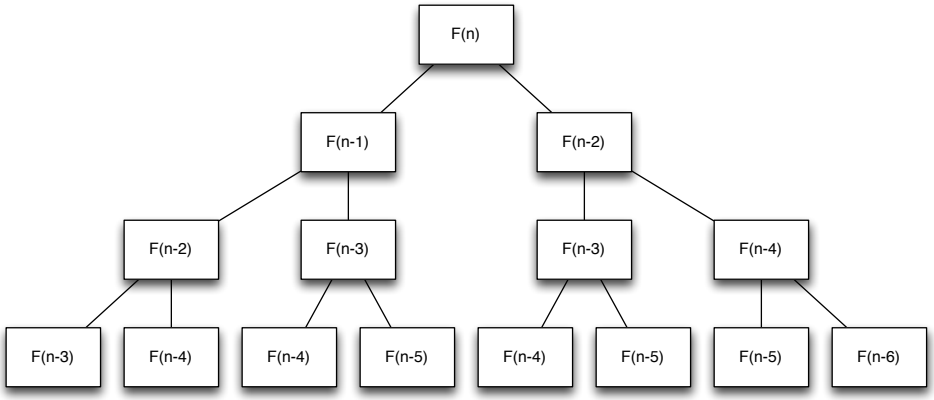

Figure 1: The Fibonacci recursion

This bound on F_n implies that, at the least, $2^{n/2}$ addition operation are required to compute F_n using this algorithm. To compute F_{200}, for example, we would need more addition operations than there are atoms in the universe.

The weakness in this naïve implementation is the number of re-computations that are required (Figure 1) because the algorithm does not 'remember' prior computations.

We can easily improve on the algorithm by using a technique called *memoization*.

2 Improved algorithm for computing F_n

Algorithm 2 An iterative algorithm to compute F_n

int fib(int n)
Use an array $F[0 \ldots n]$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
  $F[i] = F[i-1] + F[i-2]$
end for
return $F[n]$

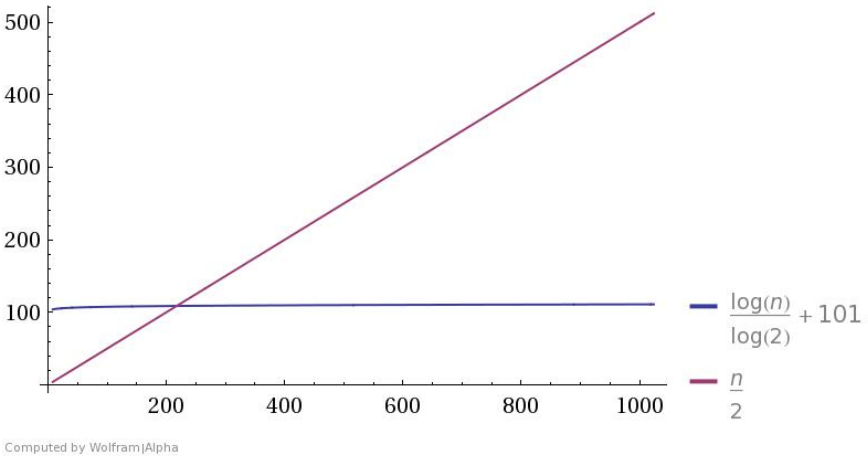

Figure 2: Complexity of computing F_n. $n/2$ grows much faster than $S+1+\lg n$. In this example, $S = 100$. This plot captures the variation with respect to n and one could also plot a similar graph to capture the variation with respect to $\log n$ to visualize the exponential growth in complexity.

The improved algorithm (Algorithm 2) uses memoization and requires no more than n additions, which is a significant improvement over the naïve algorithm (Algorithm 1). However this algorithm is not one we would call *efficient*. Any algorithm for computing F_n would need to write out at least $n/2$ bits because $F_n \geq 2^{n/2}$. To understand the time complexity of an algorithm we need to consider the number of operations to be performed relative to the size of the input and the size of the program.

Let us assume, without any loss in generality, that an implementation of the iterative algorithm for F_n requires S bits. The input, n, requires $1+\lfloor \log_2 n \rfloor$ bits[2] (where $\lfloor x \rfloor$ is the greater integer y such that $y \leq x$). The total input size (program and n) is $1+S+\lg n$ bits. Ignoring the complexity of addition, computing F_n requires at least $n/2$ bit-write operations. Therefore, relative to the total input size, the complexity of computing F_n is exponential in input size (Figure 2).

Computing F_n *requires* what one would call a $O(n)$ algorithm. However, since we measure input size in bits, we need $k \approx \lg n$ bits and so computing F_n is a $O(2^k)$ algorithm (it is exponential in the size of the input). The

[2]It is also common to use $\lg n$ in place of $\log_2 n$.

definitions of $O(\cdot)$ will be made precise later. We have also ignored the complexity of addition. We should actually account for the fact that adding two k-bit integers takes $O(k)$ time. Unless k is a constant we cannot treat all additions as single-step operations. (Insight: Adding two 32-bit integers on a 32-bit microprocessor can be thought of as one instruction, but what if we wanted to add two 128-bit integers?)

3 F_n using the closed-form expression

The Fibonacci recurrence $F_n = F_{n-1} + F_{n-2}$ is a difference equation and it is possible to compute a closed-form expression for F_n using the characteristic equation and the exact solution (as one would do with differential equations). Using such methods

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}},$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ (the golden ratio) and $\psi = \frac{1-\sqrt{5}}{2}$.

The challenge in using the closed-form expression to compute F_n, apart from dealing with floating-point computations, is in computing φ^n and ψ^n. Exponentiation (computing a^n) can take n multiplications with a naïve strategy and $\log n$ multiplications using a divide-and-conquer strategy that uses that fact that $a^n = (a^{n/2})^2$ when n is even and $a^n = (a^{(n-1)/2})^2 \times a$ when n is odd. The divide-and-conquer strategy may appear to reduce the number of arithmetic operations to $O(\log n)$ multiplications but we still have to consider the complexity of multiplying n-bit numbers, which is $O(n^2)$ using rudimentary analysis.

We could avoid the problem of dealing with floating point numbers by noticing the linear algebra representation of the Fibonacci recurrence

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix},$$

which can be used to yield

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}.$$

For an integer n, computing A^n when A is a 2×2 matrix is not very different from computing a^n where a is an integer. *(Aside: How would we efficiently multiply two matrices?)*

4 Bounds on the size of F_n

We used the fact that $2^{n/2} \leq F_n \leq 2^n \ \ \forall n \geq 6$ to understand the complexity of computing F_n. We shall now prove this result using mathematical induction. In particular, we shall see that $2^{n/2} \leq F_n$. The proof for the other inequality is left as an exercise.

To show that $\forall n \geq 6 \ \ F_n \geq 2^{n/2}$, we shall first establish the *base case.* Because $F_n = F_{n-1} + F_{n-2}$, we need to show that the propositions $F_6 \geq 2^3 = 8$ and $F_7 \geq 2^{3.5} = 11.313$ hold. This is easy to deduce because $F_6 = 8$ and $F_7 = 13$. Thus the base case holds.

Our *induction hypothesis* is that $F_n \geq 2^{n/2}$ for $n = 8, 9, \ldots, k$. We will now have to show that $F_{k+1} \geq 2^{(k+1)/2}$. This can be demonstrated as follows:

$$\begin{aligned}
F_{k+1} &= F_k + F_{k-1} \text{ (by the definition of } F_{k+1}) \\
&\geq 2^{k/2} + 2^{(k-1)/2} \text{ (using the induction hypothesis)} \\
&\geq 2^{(k-1)/2} + 2^{(k-1)/2} \text{ (using the fact that } 2^{k/2} \geq 2^{(k-1)/2}) \\
&= 2(2^{(k-1)/2}) \\
&= 2^{(k-1)/2+1} \\
&= 2^{(k+1)/2}.
\end{aligned}$$

Using the base case, the induction hypothesis and the induction step, we conclude that $\forall n \geq 6 \ \ F_n \geq 2^{n/2}$. ■