Calculating Fibonacci (and related) numbers efficiently

Martin Hock
mhock@cs.wisc.edu

January 5, 2004

1 Introduction

The Fibonacci sequence $F = (1, 1, 2, 3, 5, \ldots)$, where $F_0 = F_1 = 1$ and F_{n+2} is given by the recurrence $F_{n+2} = F_{n+1} + F_n$, represents a wide range of phenomena, from rabbit breeding to stair climbing to domino tiling. We wish to be able to calculate individual members of the sequence in a computationally efficient manner. What is the lowest asymptotic complexity we can achieve? At what points do slower but simpler algorithms become less efficient then equivalent ones which are more complicated but faster in the long run? We examine the four most common algorithms used to compute Fibonacci.

2 Algorithms

- The simple recursive algorithm based on the recurrence itself. This algorithm is not worth considering as the nth number will take $2(F_n - 1) \in O(\phi^n)$ recursive calls, so this algorithm will always be beaten by the simple iterative algorithm.

- The simple iterative algorithm which calculates each of $F_0, F_1, \ldots, F_{n-1}$ in turn in order to calculate F_n, but remembering only the previous two elements at any one time.

  Since $F(n) \in O(\phi^n)$, F_n takes $O(n)$ bits to store. Since the iterative algorithm produces all Fibonacci numbers leading up to F_n, it produces $O(n) * O(n-1)/2 = O(n^2)$ bits. Since addition takes time linear in the number of bits being operated upon, and each bit is involved in at most two addition operations, $O(n^2)$ is the total time that this algorithm takes. Since we only need to keep two numbers in memory at once, $O(n)$ is the space needed.

- The matrix multiplication algorithm which, to calculate F_n, calculates the upper left entry of

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n.$$

  Rather than simply multiplying the matrix by itself n times, we can repeatedly square the matrix $\lfloor \log_2 n \rfloor$ times, seperately multiplying together each matrix which corresponds to a 1 bit in the binary representation of n. Since this separate multiplication will only involve a subset of the numbers multiplied in the repeated squaring, the time complexity will be dominated by that of the repeated squaring. After k such squares, our primary matrix will contain a number which represents F_{2^k}, which will have $O(2^k)$ bits. We only need to hold

two matrices at once, the squaring matrix and the separate multiplication matrix, for a total of $O(n)$ space.

For multiplication, we have three main choices: standard grade school, Karatsuba, or Strassen. Folklore states that Strassen multiplication is only the fastest method when dealing with numbers on the order of at least ten thousand digits, though for the purposes of asymptotic analysis it clocks in at an impressive $O(n \log n \log\log n)$, where n is the number of bits. This algorithm is very complicated, though a simpler one based on FFT takes time For simplicity, we can bound this at $O(n \log^{2+\epsilon} n)$. Then the time spent squaring will be

$$\begin{aligned} O(\sum_{i=0}^{\log n} 2^i \log^{2+\varepsilon} 2^i) &= O(\sum_{i=0}^{\log n} 2^i i^{2+\varepsilon} \\ &= O(n \log^{2+\varepsilon} n). \end{aligned}$$

Karatsuba takes $O(n^{\log_2 3})$ ($\simeq n^{1.58}$) and the time spent squaring will also be $O(n^{\log_2 3})$. Similarly, if we use grade school multiplication at $O(n^2)$, the time spent squaring will be $O(n^2)$ also, so in order to beat the simple iterative algorithm, we need to use a more advanced multiplication algorithm.

- Binet's formula

$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

We can work in the field $\mathbb{Q} + \sqrt{5}\mathbb{Q}$ (notation?). It is easy to see that this is a field; each element is of the form

$$\frac{a+b\sqrt{5}}{c+d\sqrt{5}}$$

where a, b, c, d are integers and that adding, multiplying, or dividing two elements takes only a constant number of integer multiplications and additions if we ignore unique representation (lowest terms) issues. But as we can see, as with the matrix representation, we are still raising an algebraic object to the nth power, which will require $O(\log n)$ integer squaring operations. Since as before the intermediate representations represent Fibonacci numbers (albeit not necessarily in lowest terms), as before, after k operations we will be working with a number containing at least $O(k)$ bits. Due to the additional overhead of dealing with the division step and eventually simplifying the final number to an integer, the matrix method is a better choice.

3 Conclusion

The matrix method of generating Fibonacci numbers is more efficient than the simple iterative algorithm, though in order to see its benefits, you will probably have to work with numbers consisting of hundreds of bits or more. For smaller numbers, the simplicity of the iterative algorithm is preferable.