Chapter 0

Prologue

Look around you. Computers and networks are everywhere, enabling an intricate web of complex human activities: education, commerce, entertainment, research, manufacturing, health management, human communication, even war. Of the two main technological underpinnings of this amazing proliferation, one is obvious: the breathtaking pace with which advances in microelectronics and chip design have been bringing us faster and faster hardware.

This book tells the story of the other intellectual enterprise that is crucially fueling the computer revolution: *efficient algorithms*. It is a fascinating story.

Gather 'round and listen close.

0.1 Books and algorithms

Two ideas changed the world. In 1448 in the German city of Mainz a goldsmith named Johann Gutenberg discovered a way to print books by putting together movable metallic pieces. Literacy spread, the Dark Ages ended, the human intellect was liberated, science and technology triumphed, the Industrial Revolution happened. Many historians say we owe all this to typography. Imagine a world in which only an elite could read these lines! But others insist that the key development was not typography, but *algorithms*.

Today we are so used to writing numbers in decimal, that it is easy to forget that Gutenberg would write the number 1448 as MCDXLVIII. How do you add two Roman numerals? What is MCDXLVIII + DCCCXII? (And just try to think about multiplying them.) Even a clever man like Gutenberg probably only knew how to add and subtract small numbers using his fingers; for anything more complicated he had to consult an abacus specialist.

The decimal system, invented in India around AD 600, was a revolution in quantitative reasoning: using only 10 symbols, even very large numbers could be written down compactly, and arithmetic could be done efficiently on them by following elementary steps. Nonetheless these ideas took a long time to spread, hindered by traditional barriers of language, distance, and ignorance. The most influential medium of transmission turned out to be a textbook, written in Arabic in the ninth century by a man who lived in Baghdad. Al Khwarizmi laid out the basic methods for adding, multiplying, and dividing numbers—even extracting square roots and calculating digits of π. These procedures were precise, unambiguous, mechanical, efficient, correct—in short, they were *algorithms*, a term coined to honor the wise man after the decimal system was finally adopted in Europe, many centuries later.

Since then, this decimal positional system and its numerical algorithms have played an enormous role in Western civilization. They enabled science and technology; they accelerated industry and commerce. And when, much later, the computer was finally designed, it explicitly embodied the positional system in its bits and words and arithmetic unit. Scientists everywhere then got busy developing more and more complex algorithms for all kinds of problems and inventing novel applications—ultimately changing the world.

0.2 Enter Fibonacci

Al Khwarizmi's work could not have gained a foothold in the West were it not for the efforts of one man: the 15th century Italian mathematician Leonardo Fibonacci, who saw the potential of the positional system and worked hard to develop it further and propagandize it.

But today Fibonacci is most widely known for his famous sequence of numbers

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

each the sum of its two immediate predecessors. More formally, the Fibonacci numbers F_n are generated by the simple rule

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0\ . \end{cases}$$

No other sequence of numbers has been studied as extensively, or applied to more fields: biology, demography, art, architecture, music, to name just a few. And, together with the powers of 2, it is computer science's favorite sequence.

In fact, the Fibonacci numbers grow *almost* as fast as the powers of 2: for example, F_{30} is over a million, and F_{100} is already 21 digits long! In general, $F_n \approx 2^{0.694n}$ (see Exercise 0.3).

But what is the precise value of F_{100}, or of F_{200}? Fibonacci himself would surely have wanted to know such things. To answer, we need an algorithm for computing the nth Fibonacci number.

An exponential algorithm

One idea is to slavishly implement the recursive definition of F_n. Here is the resulting algorithm, in the "pseudocode" notation used throughout this book:

```
function fib1($n$)
if $n = 0$:  return $0$
if $n = 1$:  return $1$
return fib1($n - 1$) + fib1($n - 2$)
```

Whenever we have an algorithm, there are three questions we always ask about it:

1. Is it correct?

2. How much time does it take, as a function of n?

3. And can we do better?

The first question is moot here, as this algorithm is precisely Fibonacci's definition of F_n. But the second demands an answer. Let $T(n)$ be the number of *computer steps* needed to compute `fib1`(n); what can we say about this function? For starters, if n is less than 2, the procedure halts almost immediately, after just a couple of steps. Therefore,

$$T(n) \leq 2 \text{ for } n \leq 1.$$

For larger values of n, there are two recursive invocations of `fib1`, taking time $T(n-1)$ and $T(n-2)$, respectively, plus three computer steps (checks on the value of n and a final addition). Therefore,

$$T(n) = T(n-1) + T(n-2) + 3 \text{ for } n > 1.$$

Compare this to the recurrence relation for F_n: we immediately see that $T(n) \geq F_n$.

This is very bad news: the running time of the algorithm grows as fast as the Fibonacci numbers! *$T(n)$ is exponential in n*, which implies that the algorithm is impractically slow except for very small values of n.

Let's be a little more concrete about just how bad exponential time is. To compute F_{200}, the `fib1` algorithm executes $T(200) \geq F_{200} \geq 2^{138}$ elementary computer steps. How long this actually takes depends, of course, on the computer used. At this time, the fastest computer in the world is the NEC Earth Simulator, which clocks 40 trillion steps per second. Even on this machine, `fib1`(200) would take at least 2^{92} seconds. This means that, if we start the computation today, it would still be going long after the sun turns into a red giant star.

But technology is rapidly improving—computer speeds have been doubling roughly every 18 months, a phenomenon sometimes called *Moore's law*. With this extraordinary growth, perhaps `fib1` will run a lot faster on next year's machines. Let's see—the running time of `fib1`(n) is proportional to $2^{0.694n} \approx (1.6)^n$, so it takes 1.6 times longer to compute F_{n+1} than F_n. And under Moore's law, computers get roughly 1.6 times faster each year. So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}. And the year after, F_{102}. And so on: just one more Fibonacci number every year! Such is the curse of exponential time.

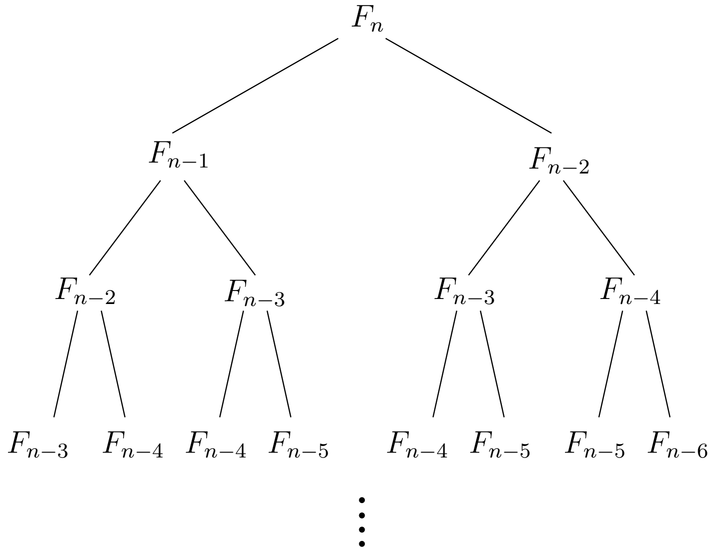
In short, our naive recursive algorithm is correct but hopelessly inefficient. *Can we do better?*

A polynomial algorithm

Let's try to understand why `fib1` is so slow. Figure 0.1 shows the cascade of recursive invocations triggered by a single call to `fib1`(n). Notice that many computations are repeated!

A more sensible scheme would store the intermediate results—the values $F_0, F_1, \ldots, F_{n-1}$—as soon as they become known.

Figure 0.1 The proliferation of recursive calls in `fib1`.



```
function fib2($n$)
if $n = 0$ return $0$
create an array f[$0 \ldots n$]
f[$0$] $= 0$, f[$1$] $= 1$
for $i = 2 \ldots n$:
   f[$i$] = f[$i-1$] + f[$i-2$]
return f[$n$]
```

As with `fib1`, the correctness of this algorithm is self-evident because it directly uses the definition of F_n. How long does it take? The inner loop consists of a single computer step and is executed $n-1$ times. Therefore the number of computer steps used by `fib2` is *linear in* n. From exponential we are down to *polynomial*, a huge breakthrough in running time. It is now perfectly reasonable to compute F_{200} or even $F_{200,000}$.[1]

As we will see repeatedly throughout this book, the right algorithm makes all the difference.

More careful analysis

In our discussion so far, we have been counting the number of *basic computer steps* executed by each algorithm and thinking of these basic steps as taking a constant amount of time. This is a very useful simplification. After all, a processor's instruction set has a variety of basic primitives—branching, storing to memory, comparing numbers, simple arithmetic, and

[1] To better appreciate the importance of this dichotomy between exponential and polynomial algorithms, the reader may want to peek ahead to *the story of Sissa and Moore,* in Chapter 8.

so on—and rather than distinguishing between these elementary operations, it is far more convenient to lump them together into one category.

But looking back at our treatment of Fibonacci algorithms, we have been too liberal with what we consider a basic step. It is reasonable to treat addition as a single computer step if small numbers are being added, 32-bit numbers say. But the nth Fibonacci number is about $0.694n$ bits long, and this can far exceed 32 as n grows. Arithmetic operations on arbitrarily large numbers cannot possibly be performed in a single, constant-time step. We need to audit our earlier running time estimates and make them more honest.

We will see in Chapter 1 that the addition of two n-bit numbers takes time roughly proportional to n; this is not too hard to understand if you think back to the grade-school procedure for addition, which works on one digit at a time. Thus `fib1`, which performs about F_n additions, actually uses a number of *basic steps* roughly proportional to nF_n. Likewise, the number of steps taken by `fib2` is proportional to n^2, still polynomial in n and therefore exponentially superior to `fib1`. This correction to the running time analysis does not diminish our breakthrough.

But can we do even better than `fib2`*?* Indeed we can: see Exercise 0.4.

0.3 Big-O notation

We've just seen how sloppiness in the analysis of running times can lead to an unacceptable level of inaccuracy in the result. But the opposite danger is also present: it is possible to be *too* precise. An insightful analysis is based on the right simplifications.

Expressing running time in terms of *basic computer steps* is already a simplification. After all, the time taken by one such step depends crucially on the particular processor and even on details such as caching strategy (as a result of which the running time can differ subtly from one execution to the next). Accounting for these architecture-specific minutiae is a nightmarishly complex task and yields a result that does not generalize from one computer to the next. It therefore makes more sense to seek an uncluttered, machine-independent characterization of an algorithm's efficiency. To this end, we will always express running time by counting the number of basic computer steps, as a function of the size of the input.

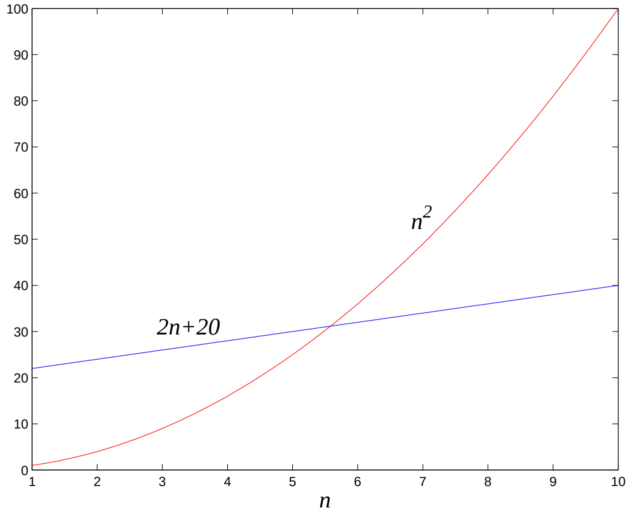
And this simplification leads to another. Instead of reporting that an algorithm takes, say, $5n^3 + 4n + 3$ steps on an input of size n, it is much simpler to leave out lower-order terms such as $4n$ and 3 (which become insignificant as n grows), and even the detail of the coefficient 5 in the leading term (computers will be five times faster in a few years anyway), and just say that the algorithm takes time $O(n^3)$ (pronounced "big oh of n^3").

It is time to define this notation precisely. In what follows, think of $f(n)$ and $g(n)$ as the running times of two algorithms on inputs of size n.

> Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ (which means that "f grows no faster than g") if *there is a constant* $c > 0$ *such that* $f(n) \le c \cdot g(n)$.

Saying $f = O(g)$ is a very loose analog of "$f \le g$." It differs from the usual notion of $\le$ because of the constant c, so that for instance $10n = O(n)$. This constant also allows us to

Figure 0.2 Which running time is better?



disregard what happens for small values of n. For example, suppose we are choosing between two algorithms for a particular computational task. One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps (Figure 0.2). Which is better? Well, this depends on the value of n. For $n \leq 5$, f_1 is smaller; thereafter, f_2 is the clear winner. In this case, f_2 *scales* much better as n grows, and therefore it is superior.

This superiority is captured by the big-O notation: $f_2 = O(f_1)$, because

$$\frac{f_2(n)}{f_1(n)} = \frac{2n+20}{n^2} \leq 22$$

for all n; on the other hand, $f_1 \neq O(f_2)$, since the ratio $f_1(n)/f_2(n) = n^2/(2n+20)$ can get arbitrarily large, and so no constant c will make the definition work.

Now another algorithm comes along, one that uses $f_3(n) = n + 1$ steps. Is this better than f_2? Certainly, but only by a constant factor. The discrepancy between f_2 and f_3 is tiny compared to the huge gap between f_1 and f_2. In order to stay focused on the big picture, we treat functions as equivalent if they differ only by multiplicative constants.

Returning to the definition of big-O, we see that $f_2 = O(f_3)$:

$$\frac{f_2(n)}{f_3(n)} = \frac{2n+20}{n+1} \leq 20,$$

and of course $f_3 = O(f_2)$, this time with $c = 1$.

Just as $O(\cdot)$ is an analog of $\leq$, we can also define analogs of $\geq$ and $=$ as follows:

$$f = \Omega(g) \text{ means } g = O(f)$$
$$f = \Theta(g) \text{ means } f = O(g) \text{ and } f = \Omega(g).$$

In the preceding example, $f_2 = \Theta(f_3)$ and $f_1 = \Omega(f_3)$.

Big-O notation lets us focus on the big picture. When faced with a complicated function like $3n^2 + 4n + 5$, we just replace it with $O(f(n))$, where $f(n)$ is as simple as possible. In this particular example we'd use $O(n^2)$, because the quadratic portion of the sum dominates the rest. Here are some commonsense rules that help simplify functions by omitting dominated terms:

1. Multiplicative constants can be omitted: $14n^2$ becomes n^2.

2. n^a dominates n^b if $a > b$: for instance, n^2 dominates n.

3. Any exponential dominates any polynomial: 3^n dominates n^5 (it even dominates 2^n).

4. Likewise, any polynomial dominates any logarithm: n dominates $(\log n)^3$. This also means, for example, that n^2 dominates $n \log n$.

Don't misunderstand this cavalier attitude toward constants. Programmers and algorithm developers are *very* interested in constants and would gladly stay up nights in order to make an algorithm run faster by a factor of 2. But understanding algorithms at the level of this book would be impossible without the simplicity afforded by big-O notation.

Exercises

0.1. In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case $f = \Theta(g)$).

	$f(n)$	$g(n)$
(a)	$n - 100$	$n - 200$
(b)	$n^{1/2}$	$n^{2/3}$
(c)	$100n + \log n$	$n + (\log n)^2$
(d)	$n \log n$	$10n \log 10n$
(e)	$\log 2n$	$\log 3n$
(f)	$10 \log n$	$\log(n^2)$
(g)	$n^{1.01}$	$n \log^2 n$
(h)	$n^2 / \log n$	$n(\log n)^2$
(i)	$n^{0.1}$	$(\log n)^{10}$
(j)	$(\log n)^{\log n}$	$n / \log n$
(k)	$\sqrt{n}$	$(\log n)^3$
(l)	$n^{1/2}$	$5^{\log_2 n}$
(m)	$n2^n$	3^n
(n)	2^n	2^{n+1}
(o)	$n!$	2^n
(p)	$(\log n)^{\log n}$	$2^{(\log_2 n)^2}$
(q)	$\sum_{i=1}^{n} i^k$	n^{k+1}

0.2. Show that, if c is a positive real number, then $g(n) = 1 + c + c^2 + \cdots + c^n$ is:

(a) $\Theta(1)$ if $c < 1$.

(b) $\Theta(n)$ if $c = 1$.

(c) $\Theta(c^n)$ if $c > 1$.

The moral: in big-Θ terms, the sum of a geometric series is simply the first term if the series is strictly decreasing, the last term if the series is strictly increasing, or the number of terms if the series is unchanging.

0.3. The Fibonacci numbers $F_0, F_1, F_2, \ldots,$ are defined by the rule

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

In this problem we will confirm that this sequence grows exponentially fast and obtain some bounds on its growth.

(a) Use induction to prove that $F_n \geq 2^{0.5n}$ for $n \geq 6$.

(b) Find a constant $c < 1$ such that $F_n \leq 2^{cn}$ for all $n \geq 0$. Show that your answer is correct.

(c) What is the largest c you can find for which $F_n = \Omega(2^{cn})$?

0.4. Is there a faster way to compute the nth Fibonacci number than by `fib2` (page 13)? One idea involves *matrices.*

We start by writing the equations $F_1 = F_1$ and $F_2 = F_0 + F_1$ in matrix notation:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Similarly,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

and in general

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

So, in order to compute F_n, it suffices to raise this 2×2 matrix, call it X, to the nth power.

(a) Show that two 2×2 matrices can be multiplied using 4 additions and 8 multiplications.

But how many matrix multiplications does it take to compute X^n?

(b) Show that $O(\log n)$ matrix multiplications suffice for computing X^n. (*Hint:* Think about computing X^8.)

Thus the number of arithmetic operations needed by our matrix-based algorithm, call it `fib3`, is just $O(\log n)$, as compared to $O(n)$ for `fib2`. *Have we broken another exponential barrier?*

The catch is that our new algorithm involves multiplication, not just addition; and multiplications of large numbers are slower than additions. We have already seen that, when the complexity of arithmetic operations is taken into account, the running time of `fib2` becomes $O(n^2)$.

(c) Show that all intermediate results of `fib3` are $O(n)$ bits long.

(d) Let $M(n)$ be the running time of an algorithm for multiplying n-bit numbers, and assume that $M(n) = O(n^2)$ (the school method for multiplication, recalled in Chapter 1, achieves this). Prove that the running time of `fib3` is $O(M(n) \log n)$.

(e) Can you prove that the running time of `fib3` is $O(M(n))$? (*Hint:* The lengths of the numbers being multiplied get doubled with every squaring.)

In conclusion, whether `fib3` is faster than `fib2` depends on whether we can multiply n-bit integers faster than $O(n^2)$. Do you think this is possible? (The answer is in Chapter 2.)

Finally, there is a formula for the Fibonacci numbers:

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

So, it would appear that we only need to raise a couple of numbers to the nth power in order to compute F_n. The problem is that these numbers are irrational, and computing them to sufficient accuracy is nontrivial. In fact, our matrix method `fib3` can be seen as a roundabout way of raising these irrational numbers to the nth power. If you know your linear algebra, you should see why. (*Hint:* What are the eigenvalues of the matrix X?)