# 1 On Algorithms

An algorithm is a recipe or a well-defined procedure for transforming some input into a desired output. Perhaps the most familiar algorithms are those those for adding and multiplying integers. Here is a multiplication algorithm that is different the algorithm you learned in school: write the multiplier and multiplicand side by side. Repeat the following operations - divide the first number by 2 (throw out any fractions) and multiply the second by 2, until the first number is 1. This results in two columns of numbers. Now cross out all rows in which the first entry is even, and add all entries of the second column that haven't been crossed out. The result is the product of the two numbers.

In this course we will ask a number of basic questions about algorithms:

- Does it halt?

  The answer for the algorithm given above is clearly yes, *provided* we are multiplying positive integers. The reason is that for any integer greater than 1, when we divide it by 2 and throw out the fractional part, we always get a smaller integer which is greater than or equal to 1.

- Is it correct?

  To see that the algorithm correctly computes the product of the integers, observe that if we write a 0 for each crossed out row, and 1 for each row that is not crossed out, then reading from bottom to top just gives us the first number in binary. Therefore, the algorithm is just doing the standard multiplication in binary.

- Is it fast?

  It turns out that the algorithm is as fast as the standard algorithm. Later in the course, we will study a faster algorithm for multiplying integers.

- How much memory does it use?

The history of algorithms for simple arithmetic is quite fascinating. Although we take them for granted, their widespread use is surprisingly recent. Today algorithms for arithmetic touch so many facets of our lives that it is hard to imagine a World without them — what would

computers, or science and technology or commerce be like without arithmetic algorithms? The key to good algorithms for arithmetic was the positional number system (such as the decimal system). The positional number system was first invented by the Mayan Indians in Central America about 2000 years ago. The decimal system we use today and the algorithms for performing arithmetic were invented in India about 600 A.D. These were transmitted via Persia to the rest of the World. It was around this time that the Persian mathematician Al-Khwarizmi compiled these and other algorithms into his arabic textbook on the subject. The word "algorithm" comes from Al-Khwarizmi's name. Al-Khwarizmi's work was translated into Latin around 1200 AD, and the positional number system was propagated throughout Europe from 1200 to 1600 AD.

With the invention of computers in this century, the field of algorithms has seen explosive growth. Perhaps the day is not far when people will have as much difficulty imagining a World without some of these new algorithms, as we have today trying to imagine a World without arithmetic algorithms. Here is a list of some of the most important successes in the field of algorithms:

- Parsing algorithms - these form the basis of the field of programming languages (CS164)
- Fast fourier transform - the field of digital signal processing is built upon this algorithm. (EE)
- Linear programming - this algorithm is extensively used in resource scheduling. (IEOR)
- sorting algorithms - until recently, sorting used up the bulk of computer cycles. (CS61B)
- string matching algorithms - these are extensively used in computational biology.
- number theoretic algorithms - these algorithms make it possible to implement cryptosystems such as the RSA public key cryptosystem. (CS276)
- Numerical algorithms for evaluating physical systems - they replace expensive or impossible physical experiments (example climate modeling) (Ma128ab)

Recall that in computer science (as in any science or engineering discipline), we must carefully choose the level of abstraction that we work at. For example, in the case of computers, we could think of operation of a computer in terms of code in a high level computer language such as Pascal or Java, or in terms of an assembly language or machine code. We could dip further down, and think of the computer at the gate level — AND and NOT gates, flip-flops,

etc. Or go beyond the digital abstraction and think of a computer as consisting of wires and transistors and resistors, with voltages and currents that are real numbers — not just 0's and 1's. The field of algorithms provides us with another level of abstraction from which we can view computers. This time we will abstract away even the details of the high level programming language, and write our algorithms in "pseudo-code", without worrying about certain details of implementation. Sometimes we have to be careful that we do not abstract away essential features of the problem. To illustrate this, let us consider an example:

## 2 Computing the $n$th Fibonacci Number

Remember the famous sequence of numbers invented in the 15th century by the Italian mathematician Leonardo Fibonacci? The sequence is represented as $F_0, F_1, F_2, \ldots$, where $F_0 = 0$, $F_1 = 1$, and for all $n \geq 2$ $F_n$ is defined as $F_{n-1} + F_{n-2}$. The first few numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$. $F_{30}$ is greater than a million! The Fibonacci numbers grow exponentially. Not quite as fast as $2^n$, but close: $F_n$ is about $2^{.694\ldots n}$.
Suppose we want to compute the number $F_n$, for some given large integer $n$. Our first algorithm is the one that slavishly implements the definition:

function $F(n$ : integer): integer
if $n = 0$ then return 0
elseif $n = 1$ then return 1
else return $F(n-1) + F(n-2)$

It is obviously correct, and always halts. The problem is, it is too slow. Since it is a recursive algorithm, we can express its running time on input $n$, $T(n)$, by a *recurrence equation*, in terms of smaller values of $T$ (we shall talk a lot about such recurrences later in this class). So, what is $T(n)$? We shall be interested in the *order of growth* of $T(n)$, ignoring constants. If $n \leq 2$, then we do constant amount of work to obtain $F_n$; since we are suppressing constants, we can write $T(n) = 1$ for $n \leq 2$. Otherwise, if $n \geq 3$, we have:

$$T(n) = T(n-1) + T(n-2),$$

because in this case the running time of $F$ on input $n$ is just the running time of $F$ on input $n-1$ —which is $T(n-1)$— plus $T(n-2)$. This is the Fibonacci equation! In other words, the running time of our Fibonacci program *grows as the Fibonnaci numbers* —that is to say, way too fast.

*Can we do better?* This is the question we shall always ask of our algorithms. The trouble with the naive algorithm is its wasteful use of recursion: The function $F$ is called with the same argument over an over again, exponentially many times (try to see how many times $F(1)$ is called in the computation of $F(5)$). A simple trick for improving this performance is to *memoize* the recursive algorithm, by remembering the results from previous calls. Specifically, we can maintain an array $A[0 \ldots n]$, initially all $\infty$, except that $A[0] = 0$ and $A[1] = 1$. This array is updated before the return step of the $F$ function, which now becomes $A[n] := A(n-1) + A(n-2)$; return $A[n]$. More importantly, *it is consulted in the beginning* of the $F$ function, and, if its value is found to be non-$\infty$ —that is to say, defined— it is immediately returned. But then of course, there would be little point in keeping the recursive structure of the algorithm, we could just write:

function $F(n : \text{integer})$: integer
array $A[0 \ldots n]$ of integer
$A[0] = 0$; $A[1] = 1$;
for $i = 2$ to $n$ do:
  $A[i] := A[i-1] + A[i-2]$
return $A[n]$

This algorithm is correct, because it is just another way of implementing the definition of Fibonacci numbers. The point is that its running time is now much better. We have a single "for" loop, executed $n$ times. And the body of the loop takes only constant time (one addition, one assignment). Hence, the algorithm takes only $O(n)$ operations.

This is usually an important moment in the study of a problem: We have come from the naive but prohibitively exponential algorithm to a polynomial one. But this problem is simple and nonstandard enough that it makes sense to ask: *Can we do even better?* Surprisingly, we can. We start by observing that we can rewrite the equations of $F_1 = F_1$ and $F_2 = F_1 + F_0$ in *matrix notation* as

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

Similarly,

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

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 $2 \times 2$ matrix to the $n$th power. Each matrix multiplication takes 12 arithmetic operations, so the question boils down to *how many multiplications does it take to raise a base (matrix, number, anything) to the nth power?* The answer is, $O(\log n)$.

To see why, just notice that to compute $X^n$, where $X$ is the matrix above, we just have to compute $X^{\lfloor \frac{n}{2} \rfloor}$, and then square it (and possibly multiply it by $X$, in the case $n$ was odd). Thus,

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + O(1),$$

which is the quintessence of $\log n$ —after all, $\log n$ is just the number of times we can take $\lfloor \cdot \rfloor$ —or even $\lceil \cdot \rceil$— before we are down to one. It follows that $T(n) = O(\log n)$. (Incidentally, notice the relationship to the introductory multiplication algorithm: There we repeatedly halved the integer $n$ and used additions to multiply a number by $n$; here we used halving to *raise a number to the nth power.*)

Have we then broken a *second exponential barrier* for the Fibonacci problem, going from $O(n)$ to $O(\log n)$? Not really. In our accounting of the time requirements for all three methods, we have made a grave and common error: We have been too liberal about *what constitutes an elementary step.* In general, in analysing our algorithms we shall assume that each arithmetic step takes unit time, because the numbers involved will be typically small enough —about $n$, the size of the problem— that we can reasonably expect them to fit within a computer's word (remember that if the number is about $n$, then its length is only about $\log n$). But in the present case, we are doing arithmetic on huge numbers, with about $n$ bits (remember, a Fibonacci number has about $.694\ldots n$ bits), —and of course we are interested in the case $n$ is truly large. When dealing with such huge numbers, and if exact computation is required, we have to use sophisticated *long integer packages.* Such algorithms take $O(n)$ time to add two $n$-bit numbers —hence the complexity of the first two methods was not really $O(F_n)$ and $O(n)$, but instead $O(nF_n)$ and $O(n^2)$, respectively; notice that the latter is still exponentially faster.

What is worse, the third algorithm involves multiplications of $O(n)$-bit integers. Let $M(n)$ be the time required to multiply two $n$-bit numbers. Then the running time of the third algorithm is in fact $O(M(n))$. The reason is that the recurrence equation above is really

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + O(M(n)),$$

where the last term accounts for the 4 multiplication and 12 addition steps. So, $T(n)$ is not

asymptotically $\log n$, but instead

$$M(n) + M(\lfloor \frac{n}{2} \rfloor) + M(\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \rfloor) + \cdots,$$

which is a geometric series adding up to $O(M(n))$.

So, the comparison between the running times of the second and third algorithms, $O(n^2)$ vs. $O(M(n))$, boils down to a most important and ancient issue, which we are going to settle in a positive way later in this course: *Can we multiply two $n$-bit integers faster than $O(n^2)$*, faster, that is, than the method we learn at school —or the clever halving method explained in the opening of these notes?