CS 124 Lecture 2

January 29, 2020

Reminder

- math reviews: Sat/Sun 10-12, MD
- sections start next week
- notes, HW1, videos up

Two-Digit Multiplication: Doubling Method

÷2 ↓	↓ ×2
23	45
11	90
5	180
~~2~~	~~360~~
1	720
	1035

remove even numbers

multiplying in binary

$$\begin{array}{r} 45 \\ \times\, 10111 \\ \hline \end{array}$$

even #s are 0 in binary

Does the algorithm halt? Yes, finite # of divisions until 1

Is the algorithm correct? Yes

Is the algorithm fast?

- doubling operations are fast because simply bit shifting
- normal n digit × n digit is $O(n^2)$ digits because addition is $O(n)$
- binary multiplication n digits × $n \cdot$ const # digits is $O(n^2)$

What are the resources my algorithm requires? Space / Memory

Is it optimal / Are there lower bounds?

Computing the nth Fibonacci

$F_0 = 0$ $F_n = F_{n-1} + F_{n-2}$ $F_n \geq 2^{n/2}$ for $n \geq 10$

$F_1 = 1$ $n \geq 2$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

FIB1(n): recursively

```
    if $n = 0$   return 0
    if $n = 1$   return 1
    else   return FIB1($n-1$) + FIB1($n-2$)
```

proof by induction

Runtime: $T(n) =$ # additions this algorithm does

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

$T(0) = 0$ similar to actual sequence: $T(n) \geq 2^{n/2}$

$T(1) = 1$ for n sufficiently larg

FIB2(n): iteratively

```
    Array A
    A[0] = 0, A[1] = 1
    for(i=2; i<=n; i++)
        A[i] = A[i-1] + A[i-2]
```

Runtime: ≈ n additions

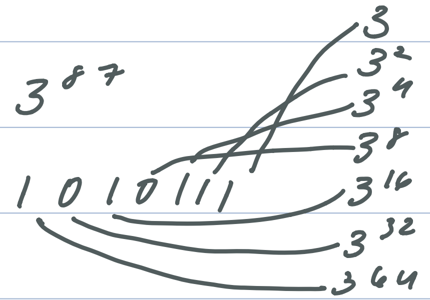
FIB3(n): reduction to matrix multiplication

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} \longrightarrow \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

repeated squaring:

$3 \times 3 \times 3 \times \dots \times 3$. EX: 3^{64}

$3 \times 3 \to 3^2 \times 3^2 \to 3^4 \times 3^4 \to \dots 3^{64}$



$T(n) = T(n/2) + 1$

$O(\log n) \le 2\log n$?

Slow! The numbers grow!

Reality: $O(n^2$ ~~logn~~$)$ $n^2 + n^2/4 + n^2/16 + \dots$

Fibonacci Formula

eigenvalues of $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$ → essentially the same as the matrix diagonalized

Order Notation

- count operations?
- worst case

  bound performance
- speed/complexity depends on hardware

O-Notation

f: positive integers → positive numbers (integers)

input size → # operations of alg

Def. For nonnegative functions $f(n)$, $g(n)$, we say $f(n)$ is $O(g(n))$ if $\exists$ positive constants c, N s.t.

$f(n) \leq c \cdot g(n) \quad \forall \ \boxed{n \geq N.}$ ← We care about asymptotics

EX: $2n^3 + 4n^2$ is $O(n^3)$? Yes.

Proof. $2n^3 + 4n^2 \leq 2n^3 + 4n^3$ for $n \geq 1$

$= 6n^3$

By the definition of $N=1$ and $c=6$, the statement is true.

EX: $10 \cdot \log_2(n)$ is $O(\ln(n))$? Yes.

$10 \cdot \log_2(n) = c \cdot \ln(n)$

EX: $n^{4/100}$ is $O((\log n)^{100})$? No.

CS 124	**Course Notes 1**	**Spring 2011**

An algorithm is a recipe or a well-defined procedure for performing a calculation, or in general, 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 from the standard 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.

75	**29**
37	**58**
~~18~~	~~116~~
9	**232**
~~4~~	~~464~~
~~2~~	~~928~~
1	**1856**
	2175

$$\begin{array}{r} 29 \\ \underline{\text{x }\ 1001011} \\ 29 \\ 58 \\ 232 \\ \underline{1856} \\ 2175 \end{array}$$

Figure 1.1: A different multiplication algorithm.

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. Hence our first number is eventually reduced to 1 and the process halts.

- 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 standard multiplication, with the multiplier written in binary.

- Is it fast?

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

- How much memory does it use?

  The memory used by this algorithm is also about the same as that of standard algorithm.

The history of algorithms for simple arithmetic is quite fascinating. Although we take these algorithms for granted, their widespread use is surprisingly recent. The key to good algorithms for arithmetic was the positional number system (such as the decimal system). Roman numerals (I, II, III, IV, V, VI, etc) are just the wrong data structure for performing arithmetic efficiently. The positional number system was first invented by the Mayan Indians in Central America about 2000 years ago. They used a base 20 system, and it is unknown whether they had invented algorithms for performing arithmetic, since the Spanish conquerors destroyed most of the Mayan books on science and astronomy.

The decimal system that we use today was invented in India in roughly 600 AD. This positional number system, together with algorithms for performing arithmetic, were transmitted to Persia around 750 AD, when several important Indian works were translated into Arabic. Around this time the Persian mathematician Al-Khwarizmi wrote 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.

The decimal point was not invented until the 10^{th} century AD, by a Syrian mathematician al-Uqlidisi from Damascus. His work was soon forgotten, and five centuries passed before decimal fractions were re-invented by the Persian mathematician al-Kashi.

With the invention of computers in this century, the field of algorithms has seen explosive growth. There are a number of major successes in this field:

- Parsing algorithms - these form the basis of the field of programming languages
- Fast Fourier transform - the field of digital signal processing is built upon this algorithm.
- Linear programming - this algorithm is extensively used in resource scheduling.
- Sorting algorithms - until recently, sorting used up the bulk of computer cycles.
- 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.
- Compression algorithms - these algorithms allow us to transmit data more efficiently over, for example, phone lines.
- Geometric algorithms - displaying images quickly on a screen often makes use of sophisticated algorithmic techniques.

In designing an algorithm, it is often easier and more productive to think of a computer in abstract terms. Of course, we must carefully choose at what level of abstraction to think. For example, we could think of computer operations in terms of a high level computer language such as C or Java, or in terms of an assembly language. We could dip further down, and think of the computer at the level AND and NOT gates.

For more on algorithms generally, you can start with the Wikipedia page: `http://en.wikipedia.org/wiki/Algorithm`.

For most algorithm design we undertake in this course, it is generally convenient to work at a fairly high level. We will usually abstract away even the details of the high level programming language, and write our algorithms in ”pseudo-code”, without worrying about implementation details. (Unless, of course, we are dealing with a programming assignment!) Sometimes we have to be careful that we do not abstract away essential features of the problem. To illustrate this, let us consider a simple but enlightening example.

1.1 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 Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ The value of F_{30} is greater than a million! It is easy to see that the Fibonacci numbers grow exponentially. As an exercise, try to show that $F_n \geq 2^{n/2}$ for sufficiently large n by a simple induction.

The Wikipedia page for Fibonacci numbers is also full of useful information: `http://en.wikipedia.org/wiki/Fibonacci_number`.

Here is a simple program to compute Fibonacci numbers that slavishly follows the definition.

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

The program is obviously correct. However, it is woefully slow. As it is a recursive algorithm, we can naturally express its running time on input n with a *recurrence equation*. In fact, we will simply count the number of addition operations the program uses, which we denote by $T(n)$. To develop a recurrence equation, we express $T(n)$ in terms of smaller values of T. We shall see several such recurrence relations in this class.

It is clear that $T(0) = 0$ and $T(1) = 0$. Otherwise, for $n \geq 2$, we have

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

because to computer $F(n)$ we compute $F(n-1)$ and $F(n-2)$ and do one other addition besides. This is (almost) the Fibonacci equation! Hence we can see that the number of addition operations is growing very large; it is at least $2^{n/2}$ for $n \geq 4$.

Can we do better? This is the question we shall always ask of our algorithms. The trouble with the naive algorithm the wasteful recursion: the function F is called with the same argument over and 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 performance is to avoid repeated calculations. In this case, this can be easily done by avoiding recursion and just calculating successive values:

```
function $F(n$: 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 of course correct. Now, however, we only do $n-1$ additions.

It seems that we have come so far, from exponential to polynomially many operations, that we can stop here. But in the back of our heads, we should be wondering *an we do even better?* Surprisingly, we can. We rewrite our equations in matrix notation. Then

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

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, Similarly,

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

So, in order to compute F_n, it suffices to raise this 2 by 2 matrix to the nth power. Each matrix multiplication takes 12 arithmetic operations, so the question boils down to the following: *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, consider the case where $n > 1$ is a power of 2. To raise X to the nth power, we compute $X^{n/2}$ and then square it. Hence the number of multiplications $T(n)$ satisfies

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

from which we find $T(n) = \log n$. As an exercise, consider what you have to do when n is not a power of 2. (Hint: consider the connection with the multiplication algorithm of the first section; there too we repeatedly halved a number...)

So we have reduced the computation time exponentially again, from $n-1$ arithmetic operations to $O(\log n)$, a great achievement. Well, not really. We got a little too abstract in our model. 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, we often assume that each arithmetic step takes unit time, because the numbers involved will be typically small enough that we can reasonably expect them to fit within a computer's word. Remember, the number n is only $\log n$ bits in length. But in the present case, we are doing arithmetic on huge numbers, with about n bits, where n is pretty large. When dealing with such huge numbers, 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 larger than we actually thought: not really $O(F_n)$ and $O(n)$, but instead $O(nF_n)$ and $O(n^2)$, respectively. The second algorithm 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 comparison between the running times of the second and third algorithms boils down to a most important and ancient issue: *can we multiply two n-bit integers faster than* $\Omega(n^2)$ *?* This would be faster than the method we learn in elementary school or the clever halving method explained in the opening of these notes.

As a final consideration, we might consider the mathematicians' solution to computing the Fibonacci numbers. A mathematician would quickly determine that

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

Using this, how many operations does it take to compute F_n? Note that this calculation would require floating point arithmetic. Whether in practice that would lead to a faster or slower algorithm than one using just integer arithmetic might depend on the computer system on which you run the algorithm.