CS 124	**Course Notes 1**	Spring 2015

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.

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

- Does the algorithm halt?
- Is it correct?

  That is, does the algorithm's output always satisfy the input to output specification that we desire?
- Is it efficient?

  Efficiency could be measured in more than one way. For example, what is the running time of the algorithm? What is its memory consumption?

Perhaps the most familiar algorithms are those those for adding and multiplying integers. We all know the grade school algorithm for addition from kindergarten: write the two numbers on top of each other, then add digits right to left while keeping track of carries. If the two numbers being added each have n digits, the total number of steps (for some reasonable definition of a "step") is n. Thus, addition of two n-digit numbers can be performed in n steps. Can we take fewer steps? Well, no, because the answer itself can be n digits long and thus simply writing it down, no matter what method you used to obtain it, would take n steps.

How about integer multiplication? Here we will describe several different algorithms to multiply two n-digit integers x and y.

Algorithm 1. We can multiply $x \cdot y$ using repeated addition. That is, add x to itself y times. Note all intermediate sums throughout this algorithm are between x and $x \cdot y$, and thus these intermediate sums can be represented between n and $2n$ digits. Thus one step of adding x to our running sum takes at most $2n$ steps. Thus the total number of steps is at most $2n \cdot y$. If we want an answer purely in terms of n, then an n-digit number y cannot be bigger than $10^n - 1$ (having n 9's). Thus $2n \cdot y \leq 2n \cdot (10^n - 1) \approx 2n \cdot 10^n$. In fact usually in this course we will ignore leading constant factors and lower order terms (more on that next lecture!), so this bound is at most roughly $n \cdot 10^n$, or as we will frequently say starting next lecture, $O(n \cdot 10^n)$. This bound is quite terrible; it means multiplying 10-digit numbers already takes about 100 billion steps!

$$
\begin{array}{cccccc}
 & & & 1 & 7 & 8 \\
 & & \times & 2 & 1 & 3 \\
\hline
 & & & 5 & 3 & 4 \\
 & & 1 & 7 & 8 & \\
+ & 3 & 5 & 6 & & \\
\hline
 & 3 & 7 & 9 & 1 & 4
\end{array}
$$

Figure 1.1: Grade school multiplication.

Algorithm 2. A second algorithm for integer multiplication is the one we learned in grade school, shown in Figure 1.1. That is, we run through the digits of y, right to left, and multiply them by x. We then sum up the results after multiplying them by the appropriate powers of 10 to get the final result. What's the running time? Multiplying one digit of y by the n-digit number x takes n steps. We have to do this for each of the n digits of y, thus totaling n^2 steps. Then we have to add up these n results at the end, taking n^2 steps. Thus the total number of steps is $O(n^2)$. As depicted in Figure 1.1 we also use $O(n^2)$ memory to store the intermediate results before adding them. Note that we can reduce the memory to $O(n)$ without affecting the running time by adding in the intermediate results to a running sum as soon as we calculate them.

Now, it is important at this point to pause and observe the difference between two items: (1) the problem we are trying to solve, and (2) the algorithm we are using to solve a problem. The problem we are trying to solve is integer multiplication, and as we see above, there are multiple algorithms which solve this problem. Prior to taking this class, it may have been tempting to equate integer multiplication, the *problem*, with the grade school multiplication procedure, the *algorithm*. However, they are not the same! And in fact, as algorithmists, it is our duty to understand whether the grade school multiplication algorithm is in fact the most efficient algorithm to solve this problem. In fact, as we shall soon see, it isn't!

Algorithm 3. Let's assume that n is a power of 2. If this is not the case, we can pad each of x, y on the left with enough 0's so that it does become the case (and doing so would not increase n by more than a factor of 2, and recall we will ignore constant factors in stating our final running times anyway). Now, imagine splitting each number x and y into two parts: $x = 10^{n/2}a + b, y = 10^{n/2}c + d$. Then

$$xy = 10^n ac + 10^{n/2}(ad + bc) + bd.$$

The additions and the multiplications by powers of 10 (which are just shifts!) can all be done in linear time. We have therefore reduced our multiplication problem into four smaller multiplications problems. Thus if we let $T(n)$ be a function which tells us the running time to multiply two n-digit numbers. Then $T(n)$ satisfies the equation

$$T(n) = 4T(n/2) + Cn$$

for some constant C. Also when $n = 1$, $T(1) = 1$ (we imagine we have hardcoded the 10×10 multiplication table for all digits in our program). This equation where we express $T(n)$ as a function of T evaluated on small numbers is is what's called a *recurrence relation*; we'll see more about this next lecture. Thus what we are saying is that the time to multiply two n-digit numbers equals that to multiply $n/2$ digit numbers, 4 times (in each of our recursive subproblems), plus an additional Cn time to combine these problems into the final solution (using additions and shifts). If we draw a recursion tree, we find that at the root we have to do all the work of the 4 subtrees, plus Cn work at the root itself to combine results from these recursive calls. At the next level of the tree, we have 4 nodes. Each one, when receivng the results from its subtrees, does $Cn/2$ work to combine those results. Since there are 4 nodes at this level, the total work at this level is $4 \cdot Cn/2 = 2Cn$. In general, the total work in the kth level (with the root being level 0) is $2^k \cdot Cn$. The height of the tree is $h = \log_2 n$, and thus the total work is

$$Cn + 2Cn + 2^2Cn + \ldots + 2^hCn = Cn(2^{h+1} - 1) = Cn(2n - 1) = O(n^2).$$

Thus, unfortunately we've done nothing more than give yet another $O(n^2)$ algorithm more complicated than Algorithm 2.

Algorithm 4. Algorithm 3, although it didn't give us an improvement, can be modified to give an improvement. The following algorithm is called *Karatsuba's algorithm* and was discovered by the Russian mathematician Anatolii Alexeevitch Karatsuba in 1960. The basic idea is the same as Algorithm 3, but with one clever trick. The key thing to notice here is that four multiplications is too many. Can we somehow reduce it to three? It may not look like it is possible, but it is using a simple trick. The trick is that *we do not need to compute ad and bc separately; we only need their sum $ad + bc$.* Now note that

$$(a+b)(c+d) = (ad+bc) + (ac+bd).$$

So if we calculate ac, bd, and $(a+b)(c+d)$, we can compute $ad+bc$ by the subtracting the first two terms from the third! Of course, we have to do a bit more addition, but since the bottleneck to speeding up this multiplication algorithm is the number of smaller multiplications required, that does not matter. The recurrence for $T(n)$ is now

$$T(n) = 3T(n/2) + Cn.$$

Then, drawing the recursion tree again, there are only 3^k nodes at level k instead of 4^k, and each one requires doing $Cn/2^k$ work. Thus the total work of the algorithm is, again for $h = \log_2 n$,

$$Cn + \frac{3}{2}Cn + \left(\frac{3}{2}\right)^2 Cn + \ldots + \left(\frac{3}{2}\right)^h Cn = Cn \cdot \frac{\left(\frac{3}{2}\right)^{h+1} - 1}{\frac{3}{2} - 1},$$

where we used the general fact that $1 + p + p^2 + \ldots + p^m = (p^{m+1} - 1)/(p - 1)$ for $p \neq 1$. Now, using the general fact that we can change bases of logarithms, i.e. $\log_a m = \log_b m / \log_b a$, we can see that $(3/2)^{\log_2 n} = (3/2)^{\log_{3/2} n / \log_{3/2} 2} = n^{1/\log_{3/2} 2}$. Then changing bases again, $\log_{3/2} 2 = \log_2 2 / \log_2 (3/2) = 1/(\log_2 3 - 1)$. Putting everything together, our final running time is then $O(n^{\log_2 3})$, which is roughly $O(n^{1.585})$, much better than the grade school algorithm! Now of course you can ask: is this the end? Is $O(n^{\log_2 3})$ the most efficient number of steps for multiplication? In fact, it is not. The Schönhage-Strassen algorithm, discovered in 1971, takes a much smaller $O(n \log_2 n \log_2 \log_2 n)$ steps. The best known algorithm to date, discovered in 2007 by Martin Fürer, takes $O(n \log_2 n 2^{C \log^* n})$ for some constant C. Here, $\log^* n$ is the number of base-2 logarithms one must take of n to get down to a result which is at most 1. The point is, it is a *very* slow growing function. If one took $\log^*$ of the number of particles in the universe, the result would be at most 5!

Also, Karatsuba's algorithm is not just a source of fun for theorists, but actually is used in practice! For example, it is used for the implementation of integer multiplication in Python. If you want to check it out for yourself, here's what I did on my Ubuntu machine:

```
apt-get source python3.2-dev
emacs python3.2-3.2.3/Objects/longobject.c
```

Now look through that file for mentions of Karatsuba! Most of the action happens in the function `k_mul`.

Now that we've seen just how cool algorithms can get. 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 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.

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* $\Omega(n^2)$ *?* We saw in the first lecture that indeed this is possible, using Karatsuba's algorithm!

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.