CS 161, Lecture 1 Introduction
Scribes: V.V. Williams, J. Su (2015), W. Yang, G. Valiant **Date:** January 9, 2017

Optional reading for this lecture: Kleinberg-Tardos: p. 278-280 Sec 5.5, 6.9

1 Course information

- The class website is `http://cs161.stanford.edu`. Additionally, there is a Piazza site which you should sign up for — the link is on the class website. Final grades for the class will be based on
  - Homework (35%)
  - Midterm (25%): Monday, October 31, in class, 1:30 pm - 2:50 pm
  - Final (40%): Wednesday, December 14, 3:30 pm - 6:30 pm (Location TBA)
- We will use two textbooks for this course:
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press

    This is the main textbook that we use. It is available online through the Stanford library.
  - Jon Kleinberg, Éva Tardos, *Algorithm Design*, Pearson/Addison-Wesley

    This is another textbook that we will usually occasionally.

2 Why are you here?

Many of you are here because the class is required. But why is it required?

1. **Algorithms are fundamental to all areas of CS:** Algorithms are the backbone of computer science. Wherever computer science reaches, an algorithm is there, and the classical algorithms algorithmic design paradigms that we cover re-occur throughout all areas of CS. For example, CS 140 and 143 (operating systems and compiles) leverages scheduling algorithms and efficient data structures, CS 144 (networking) crucially uses shortest-path algorithms, CS 229 (machine learning) leverages fast geometric algorithms and similarity search, CS 255 (cryptography) leverages fast number theoretic and algebraic algorithms, and CS 262 (computational biology) leverages algorithms that operate on strings–and often employs the dynamic programming paradigm. We'll discuss applications to all these subjects in this class.

   Algorithms and the computational perspective (the "computational lens") has also been fruitfully applied to other areas, such as physics (e.g. quantum computing), economics (e.g. algorithmic game theory), and biology (e.g. for studying evolutions, as a surprisingly efficient algorithm that searches the space of genotypes).

2. **Algorithms are useful:** Much of the progress that has occurred in tech/industry is due to the dual developments of improved hardware (a la "Moore's Law"—a prediction made in 1965 by the co-founder of Intel that the density of transistors on integrated circuits would double every year or two), and improved algorithms. In fact, the faster computers get, the bigger the discrepancy is between what can be accomplished with fast algorithms vs what can be accomplished with slow algorithms. . . . Industry needs to continue developing new algorithms for the problems of tomorrow, and you can help contribute.

3. **Algorithms are fun!** The design and analysis of algorithms requires a combination of creativity and mathematical precision. It is both an art and a science, and hopefully at least some of you will come to love this combination. One other reason it is so much fun is that algorithmic surprises abound. Hopefully this class will make you re-think what you thought was algorithmically possible, and cause you to constantly ask "is there a better algorithm for this task?". Part of the fun is that Algorithms is still a young area, and there are still many mysteries, and many problems for which (we suspect that) we still do not know the best algorithms. This is what makes research in Algorithms so fun and exciting, and hopefully some of you will decide to continue in this direction.

3 Karatsuba Integer Multiplication

3.1 The problem

Suppose you have two large numbers, and you want to multiply them. In general, we will focus on algorithms that work well for *large* instances ("Only large instances of a problem are interesting")—think about developing an algorithm for multiplying 100-digit or 1000-digit numbers, this is when the problem starts to get interesting. We all know how to multiply numbers, but when the numbers get large, the grade-school algorithm becomes slow. Question: can we do better?

The quality of an algorithm can be measured by how long it takes—in this setting, we care about the number of basic operations that must be performed, where we define a "basic operation" as the multiplication or addition of 1-digit numbers. In the grade-school multiplication algorithm, we need to multiply each digit in the first number by each digit in the second number. So if we are multiplying two n-digit numbers, the total number of one-digit multiplications we need to perform is $n \cdot n$, and then we need to perform some additions, though lets not worry about the additions for the time being. Thus the total amount of work you need to do to multiply these numbers using the grade-school multiplication algorithm is at least n^2 1-digit operations. Can we do better?

To improve this algorithm, we could consider storing the products of all pairs of t-digit numbers. This does result in performance gains, however, and also leads to exponential storage costs. (For example, if $t = 100$, we would need to store a table of $10^{2t} = 10^{200}$ products. Note that the number of atoms in the universe is only $\approx 10^80 \ldots$.)

3.2 Divide and Conquer

The "Divide and Conquer" algorithm design paradigm is a very useful and widely applicable technique. We will see a variety of problems to which it can be fruitfully applied. The high-level idea is just to split a given problem up into smaller pieces, and the solve the smaller pieces, often recursively.

How can we apply Divide and Conquer to integer multiplication? Lets try splitting up the numbers. For example, if we were multiplying 1234×5678, we could express this as $((12 \cdot 100) + 34) \cdot ((56 \cdot 100) + 78)$. In general, if we are multiplying two n-digit numbers x and y, we can write $x = 10^{n/2} \cdot a + b$ and $y = 10^{n/2} \cdot c + d$. So

$$x \cdot y = (10^{n/2}a + b) \cdot (10^{n/2}c + d) = 10^n ac + 10^{n/2}(ad + bc) + bd.$$

Now we can split this problem into four subproblems, where each subproblem is similar to the original problem, but with half the digits. This gives rise to a recursive algorithm.

Interestingly enough, **this algorithm isn't actually better!** Intuitively this is because if we expand the recursion, we still have to multiply every pair of digits, just like we did before. But in order to prove this formally, we need to formally define the runtime of an algorithm, and prove that these algorithms are not very different in runtime.

3.3 Recurrence Relation

We can analyze the runtime of the algorithm as follows.

Let $T(n)$ be the runtime of the algorithm, given an input of size n (two n-digit numbers). Because we are breaking up the problem into four subproblems with half the digits, plus some addition with linear cost, we have the equation $T(n) = 4T(n/2) + O(n)$. (Don't worry if you haven't seen big-O notation before; we'll go over this in detail in the coming lectures.)

Although **in general you should pay attention to the $O(n)$ term**, today we will just ignore it because the term doesn't matter in this case.

By repeatedly breaking up the problem into subproblems, we find that

$$T(n) = 4T(n/2) = 16T(n/4) = \ldots = 2^{2t}T\left(\frac{n}{2^t}\right) = n^2T(1).$$

Since $T(1)$ is the time it takes to multiply two digits, we see that the above suggestion does not reduce the number of 1-digit operations.

3.4 Divide and conquer (take 2)

Karatsuba found a better algorithm by noticing that we only need the sum of ad and bc, not their actual values. So he improved the algorithm by computing ac and bd as before, and computing $(a + b) \cdot (c + d)$. It turns out that if $t = (a+b) \cdot (c+d)$, then $ad+bc = t-ac-bd$. Now instead of solving four subproblems, we only need to solve three! (This idea goes back to Gauss, who found a similar efficient way to multiply two complex numbers.)

Sure, we need to do more additions. (But as we will see in a few lectures, we can formally show that the additions are cheap.) To do a quick-and-dirty analysis of the number of operations required by Karatsuba multiplication, first assume that $n = 2^s$ for some integer s. (Note that we can always add 0's to the front of a number until the length is a power of two, so this assumption holds without loss of generality.) Letting $T(n)$ denote the number of multiplications of pairs of 1-digit numbers required to compute the product of two n-digit numbers, Karatsuba's algorithm gives $T(n) = 3T(n/2)$, since we've divided the problem into three recursive calls to multiplication of length $n/2$ numbers. [Note, we are cheating a bit here, since $(a + b)$ and $(c + d)$ might actually be $n/2 + 1$ digit numbers, but lets ignore this for now...] Hence we have the following:

$$T(n) = 3T(n/2) = 3^2T(n/4) = \ldots = 3^sT(n/2^s).$$

Since we assumed $n = 2^s$, we have that $T(n/2^s) = T(1) = 1$, since multiplying two 1-digit numbers counts as 1 basic operation. Hence $T(n) = 3^s$, where $n = 2^s$. Solving for s yields $s = \log_2 n$, and hence we get

$$T(n) = 3^{\log_2 n} = 2^{(\log_2 3)(\log_2 n)} = n^{\log_2 3} \leq n^{1.6}.$$

If you didn't follow this arithmetic, don't worry about it, we will see a more systematic way of analyzing the total runtime of these recursive algorithms next class. Still, you should be surprised/excited about the result. This algorithm seems MUCH better than the n^2 operation multiplication algorithm that we learned in grade school!

4 Etymology of the word "Algorithm"

As a round-about way of describing the etymology of the word "Algorithm", pause for a minute and consider how remarkable it is that 3rd graders can actually multiply large numbers. Its really amazing that anyone, let alone a 8-yr old, can multiply two 10-digit numbers. One reason multiplication is so easy for us is because we have a great datastructure for numbers—we represent numbers using base-10 (Arabic) numerals, and this lends itself to easy arithmetic.

Why were romans so bad at multiplication? Well, imagine multiplying using roman numerals. What is LXXXIX times CM? The only way I can imagine computing this is to first translate the numbers into Arabic numerals [$LXXXIX = 50 + 10 + 10 + 10 + (-1) + 10 = 89$, and $CM = (-100) + 1000 = 900$] then multiplying those the standard way. Roman numerals seem like a pretty lousy data structure if you want to do arithmetic.

The word "Algorithm" is a mangled transliteration of the name "al-Khwarizmi", who was a Persian mathematician (and geographer, astronomer, scholar, etc.) who lived around 800AD. He wrote several influencial books, including one with the title [something like] "On the Calculation with Hindu Numerals", which described how to do arithmetic using Arabic numerals (aka Arabic-Hindu, or just Hindu numerals). The original manuscript was lost, though a Latin translation from the 1100's introduced this number system to Europe, and is responsible for why we use Arabic numerals today. (You can imagine how happy a 12th century tax collector would have been with this new ability to easily do arithmetic....) [The old French word *algorisme* meant "the Arabic numerals system", and only later did it come to mean a general recipe for solving computational problems.]