1.1 Administrative Stuff

Welcome to Algorithms! In this class you will learn the basics of the theory of algorithms. Most importantly, you will learn how to design and analyze algorithms with an eye towards provable performance.

Some administrative trivia for the class:

- The course website is `http://www.cs.jhu.edu/~mdinitz/IntroAlgorithms/`. All materials can be found there, including the syllabus with the official class policies and a tentative schedule. All announcements will be posted there.
- Homeworks will go out on Tuesdays and will be due by the beginning of class. Can work in groups of at most 3, but everyone needs to do their own, independent writeup. That is, collaboration is limited to talking and working on the problems, and cannot include writing up the solutions.
- Turn in homeworks on Blackboard. Different page for each question, your name and group members on each question. We **strongly** prefer homeworks written up using LaTeX (resources on course webpage), but will accept handwritten and scanned solutions or solutions written up using other software (e.g. Word). But if we can't read it, no credit.
- Two midterms and a final exam, dates posted on web.
- We will use Piazza for content questions. If you have a question about a lecture, homework, book chapter, etc., please post it on Piazza instead of emailing the instructor, TA, or CAs. Not only does this get multiple eyes on your question, it also avoids duplication since other students might have the same question.

1.2 Course Overview

This course is about the theory of algorithms. Note the word *theory*: there will be no programming assignments in this class. Instead, we will do formal mathematical proofs about algorithms and, at the end of the course, their converse: complexity theory. We will learn how to design efficient and correct algorithms, and also how to analyze correctness, running time, and other properties of algorithms.

What is an algorithm? A method for solving computational problems, sometimes explained as a recipe. Always at least want to have correctness: the algorithm does correctly solve the problem, i.e. its outputs are what you think they are. Many time we also want other guarantees, e.g. that it runs in time at most $f(n)$ on any input of size n. This course will focus on both aspects: how to design an algorithm, and how to prove that they meet the desired specification.

1.3 Why?

Obvious why we want to prove correctness. But why prove bounds on running time? Why not just try on a bunch of examples to test experimentally if the algorithm is fast? Many reasons, including but not limited to:

1. How do you know that your test instances are an accurate representation of "real-life" instances? Particularly important for "low-level" algorithms – if the algorithm will be a subroutine for many different, larger algorithms, might encounter a huge variety of different instances with different properties.
2. We will care about how running time changes with respect to the instance size, i.e. how the algorithm *scales*. Hard (but not necessarily impossible) to determine scaling behavior experimentally.
3. Perhaps most importantly, when we prove something about an algorithm, we *understand* it. Experimental evidence does not provide any understanding – it wouldn't be able to tell us *why* the algorithm exhibits the behavior, just that it does. Forcing ourselves to prove bounds forces us to really understand what's going on. This is particularly true when paired with complexity theory, which lets us provide lower bounds on algorithms. If we can prove matching upper and lower bounds, we really understand a problem.

1.4 Karatsuba Multiplication

One of the reasons that it is interesting to study algorithms is that often, the "obvious" way to do something from the definition is in fact quite bad. As an example, consider vanilla multiplication. Suppose we want to multiply two n-bit numbers X and Y (so each number is between 0 and $2^n - 1$). From the definition of multiplication, we could add X to itself Y times to get $X \times Y$. But this takes $\Theta(2^n)$ additions, so at least that much time!

Better idea: grade-school algorithm. Suppose we want to multiply 54 and 41:

```
          110110  = 54
x         101001  = 41
----------------
          110110
       110110
+    110110
----------------
    100010100110  = 2 + 4 + 32 + 128 + 2048 = 2214
```

Algorithmically, we scan the second number from right to left, and each time we see a 1 we add the first number (padded with an appropriate number of 0's) to the total. So n additions, and each addition takes $O(n)$ time. Total time: $O(n^2)$.

So this shows that sometimes the "obvious" algorithm from the definition might not be the right one. But is the grade-school algorithm the best possible? It turns out that the answer is no: better algorithms are possible! The following algorithm is due to Anatoli Karatsuba, from 1962. Suppose we want to multiple X and Y, both of which are n-bit numbers. We first rewrite them:

$$X = 2^{n/2}A + B$$
$$Y = 2^{n/2}C + D$$

Then we get that

$$XY = (2^{n/2}A + B)(2^{n/2}C + D) = 2^n AC + 2^{n/2}AD + 2^{n/2}BC + BD \tag{1.4.1}$$

Computing XY with this formula takes four $n/2$-bit multiplications, three shifts, and three $O(n)$-bit adds. If we let $T(n)$ be the time necessary for this algorithm, we get the recurrence relation

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

where c is a constant that handles the cost of this shifts and adds. When we solve this recurrence, we get that $T(n) = O(n^2)$, so we unfortunately have not made any progress. But now let's rewrite Equation (1.4.1):

$$XY = 2^{n/2}(A + B)(C + D) + (2^n - 2^{n/2})AC + (1 - 2^{n/2})BD$$

This looks a lot more complicated, but when we count the operations we see that there are only three $n/2$-bit multiplications, together with a constant number of shifts and $O(n)$-bit additions. So now the recurrence relation looks like

$$T(n) = 3T(n/2) + cn$$

(where the new c is larger than the old one, but still a constant). When we solve this, we get that $T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$.

It turns out that while fast than the grade-school $O(n^2)$-time algorithm, this still is not the fastest possible (or even known). Using the Fast Fourier Transform (which we may discuss at the end of the semester, time permitting), it is possible to design an $O(n \log^2 n)$-time algorithm (this was first done by Dick Karp). Even this has been improved a few times, and the best result known is an $O(n(\log n)2^{O(\log^* n)})$-time algorithm due to Fürer in 2007, where $\log^* n$ is a very slowly-growing function of n which we will discuss next week.

1.5 Matrix Multiplication

Another famous and important example of the "obvious" algorithm not being optimal is matrix multiplication. Suppose we want to multiply matrix X and matrix Y, both of which are $n \times n$ (so each contains n^2 entries). The normal algorithm (which you should all know) computes the output (i, j) entry by computing the dot product of the i'th row of X with the j'th column of Y. Each

dot product computation involves n multiplies and adds, so takes time $O(n)$[1]. Since we do this computation for each entry of the output matrix, the total time is $O(n^3)$.

It turns out that there are faster algorithms for matrix multiplication that use the same basic ideas as Karatsuba's algorithm. The first, and most famous, of these is due to Volker Strassen, in 1969. We start by breaking each of X and Y into four $(n/2) \times (n/2)$ matrices:

$$X = \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array} \qquad\qquad Y = \begin{array}{|c|c|}\hline E & F \\ \hline G & H \\ \hline \end{array}$$

It's not hard to see that we can write XY using these 8 smaller $(n/2) \times (n/2)$ matrices:

$$XY = \begin{array}{|c|c|}\hline AE + BG & AF + BH \\ \hline CE + DG & CF + DH \\ \hline \end{array}$$

This algorithm recursively computes 8 products of $(n/2) \times (n/2)$ matrices, and does 4 additions of these matrices. Each addition takes time $O(n^2)$, so the running time for this algorithm is

$$T(n) = 8T(n/2) + cn^2.$$

Solving this recurrence gives $T(n) = O(n^3)$, which is not an improvement over the previous algorithm. However, Strassen realized that, like with Karatsuba, there is a way to compute the same product using fewer multiplications. In particular, we compute the following 7 products:

$$\begin{aligned} M_1 &= (A + D)(E + H) \\ M_2 &= (C + D)E \\ M_3 &= A(F - H) \\ M_4 &= D(G - E) \\ M_5 &= (A + B)H \\ M_6 &= (C - A)(E + F) \\ M_7 &= (B - D)(G + H) \end{aligned}$$

You can check that these seven matrices let us compute XY as follows:

$$XY = \begin{array}{|c|c|}\hline M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ \hline M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \\ \hline \end{array}$$

So now we've won! The recurrence relation for the running time is $T(n) = 7T(n/2) + cn^2$, which solves to $T(n) = O(n^{\log_2 7}) \approx O(n^{2.8074})$.

While Strassen's algorithm was the first to break the n^3 barrier, it too can be improved. The first major improvement was due to Coppersmith and Winograd, who in 1990 gave an algorithm

[1]Note that here, unlike the previous example, we are assuming that a single add or multiply takes a constant amount of time. This is an example of where the costs we use are determined from context and history.

with running time $O(n^{2.375477})$. This was the best result known until recently: in 2011 Virginia Vassilevska Williams showed how to extend the Coppersmith-Winograd framework to get an $O(n^{2.3728642})$, which is currently the best known. But as far as we know, there is an $O(n^2)$-time algorithm for matrix multiplication that no one has figured out yet!