This is roughly what I said in class on 10 January 2005.

My computer is too slow

I have a problem: my computer is too slow. Maybe you can help me with this problem. You've probably had similar problems: your computers are too slow. Do you have any suggestions for what I should do?

[Suggestion from class: "Stop using the computer."] Okay, that's fix number one. Stop using the computer.

Unfortunately, no matter how hard I try, I can't really stay away from the computer. So I still have this problem: my computer is too slow. Any other suggestions?

[Suggestion from class: "Buy a faster computer."] Okay, that's fix number two. Buy a faster computer. One possibility is to buy a parallel computer, i.e., buy more computers, which will help if we want the computer to do something parallelizable. Of course, this costs money, but it's better to spend money on computers than to waste it on food, for example, or on graduate students.

Well, okay, I *have* a faster computer. Now I've run out of money and my computer is *still* too slow.

What do I mean by "too slow"? My standards for exactly how long I'm willing to wait depend on what I'm doing. I'm always happy if the computer responds instantly, by which I mean within 1/60 of a second. Sometimes I'm willing to wait longer. For example, users seem to be happy to have a web page appear within one second. Or maybe not. Wouldn't it be cool to have web pages appearing the instant you click on them? Maybe users are unhappy but have simply given up complaining.

If I'm doing a computation to announce at a conference in six months then I probably don't need the answer today, or this week, or even this month. I'm happy to let the computer chug along in the background while I do something else. On the other hand, sometimes I do a computation and I need the answer right now—and the computer doesn't give it to me that quickly.

The bottom line is that I tell the computer to do something, let's call it X, and I'm forced to wait for the response. I tell the computer to do something else, let's call it Y, and I'm forced to wait for the response. Et cetera. For these operations X and Y, my computer is too slow. Any other suggestions?

[Suggestion from class: "Make X faster."] Okay, that's fix number three. Let me be careful to distinguish between changing what I want the computer to do—that was fix number one—and changing how the computer does it—that's the new idea. We're still going to perform the same operation X, but we're going to do it in a better way.

The first step is to figure out *why* the current code for X takes as long as it does. This is called "algorithm analysis." The operation X is some function: it has an output, namely the information that I want the computer to provide, and an input, namely the information that the computer already has. The "algorithm" is the sequence of instructions that the computer follows to compute

the output starting from the input. "Analysis" means understanding, or at least trying to understand, the time taken by the algorithm.

The second step is to write better code for X. This is called "algorithm improvement." Sometimes people say "algorithm design," which doesn't really capture what we're trying to do. It's easy to design *an* algorithm for this function; all of the functions that we look at in algorithms courses are obviously computable. The problem is to design a *fast* algorithm. Sometimes people say "optimization," which is going too far: "optimization" means finding the *best* algorithm, whereas people almost always find an algorithm that's merely *better*, not the *best*. We'll see many examples of that in this course.

Then do the same for Y and for any other operations that are too slow.

Goal of this course

This course will focus on some specific operations X, Y, etc.: namely, a few fundamental cryptographic operations.

What does cryptography do? It protects messages sent through the network. Anyone with access to the network can intercept mail messages and web pages and "packets" (little messages sent between computers) to see what the messages say; he can also send fake messages that look like real messages. Cryptography scrambles and unscrambles communications (messages; packets; whatever) to protect against forgery and against espionage. An attacker who forges a message won't be able to scramble it in the right way, so when we unscramble it we'll see that it's a forgery and that we should throw it away. Similarly, an attacker who intercepts a scrambled credit-card number, for example, won't be able to figure out what the credit-card number is.

High-speed cryptography is important for busy network servers. Consider, for example, the following quote (from 2003, I admit, but computers weren't *that* much slower back then): "Verifying signed resource records [i.e., performing cryptographic operations] is computationally intensive [i.e., is painfully slow]. This has slowed deployment of DNS security." In other words, the computer was so slow at cryptographic operations that users felt compelled to resort to the first fix: not doing the computation at all; not deploying the cryptographic tools necessary to protect their communications.

Of course, there are many speed problems, speed bottlenecks, other than cryptography. Probably the most important example is video games. That's something where it's really important to get a response within 1/60 of a second. Anyway, *some* of the techniques in this course are applicable to problems other than cryptography. Other techniques are dedicated to cryptography, and often to the particular functions that I'm computing. Usually it will be obvious, when I describe a technique, how broadly applicable the technique is; sometimes I'll say it explicitly. But the goal of this course isn't to survey techniques with a particular scope. The goal is to compute certain cryptographic functions as quickly as possible. I'll pull together every technique that helps the bottom line.

Other courses

The things I've said shouldn't sound completely new to you. In particular, you're required to have some experience with algorithm analysis and algorithm improvement. The prerequisite for this course is a baby algorithms course, such as MCS 401 or CS 401. (I'll say in a moment what "baby" means here.) I realize that some of you are new graduate students not familiar with our course numbers, so here's a typical quote from 401: "Heapsort takes time $n \lg n$ times, um, some constant? Uh..." You might have used a book by Cormen, Leiserson, Rivest, and Stein, where the authors say that working out constants is "like, totally tedious, dude." If you've seen these quotes before then you've probably taken a baby algorithms course.

The most obvious difference in flavor between 401 and this course is that we *will* pay attention to constant factors. We'll say things like "This takes $3.625(n + 170)$ clock cycles on an Athlon XP" and "Cormen, Leiserson, Rivest, and Stein are ignorant, lazy bums." Clock cycles are a unit of time; for example, a 2000MHz Athlon XP has 2 billion clock cycles per second. You divide one microsecond by the number of megahertz to get a clock cycle.

Notice that these constant factors depend on the computer. Sometimes one computer takes 10 clock cycles for a simple operation while another computer takes just 1 clock cycle; this affects the time for any algorithm built on top that operation.

You might have seen some computer-dependent analysis and optimization in courses on architecture, assembly language, "optimizing" compilers (which I should call "improving" compilers, but then nobody would understand what I'm talking about), supercomputing, etc. [Class feedback at this point: some people have seen architecture and assembly language; some people have had compiler courses; a few people have had supercomputing courses.] None of these are prerequisites. I'll be developing all the material I need as part of this course.

I'll be starting with the UltraSPARC, an overpriced chip with the virtue that its performance is easy to understand. I'll then move on to the PowerPC (the CPU in Macs), the Pentium III, et al. We'll learn how all of these chips work, and we'll see speedups that depend on the chip details.

Another reason to pay attention to constant factors is that many higher-level speedups—in fact, most of the speedups in the algorithms literature—are "only" constant-factor speedups. (When I talk about "level," I'm talking about building more complicated functions on top of simpler functions. A high-level function uses lower-level functions. A machine instruction is the lowest-level function.)

For example, later on we'll spend time looking at the problem of modular exponentiation: "Compute $x^e \bmod n$." This shows up in cryptography in a few important ways that I'll talk about later.

The input is an integer $x \geq 0$, an integer $e \geq 0$, and an integer $n \geq 1$. (Allowing negative exponents creates a quite different problem.) This isn't actually a complete specification of the input, as you'll see if you give this problem to a beginning programmer—he'll say `int x`, which allows only a 32-

bit integer, whereas I want to allow much larger integers. To completely specify the problem, I'll have to tell you how these integers are represented inside the computer, which later on we'll see is an important issue. For the moment, let's imagine integers as being represented as strings of digits, the same way you write them on paper.

Anyway, the output is the remainder when you divide x^e by n; in other words, $x^e - n\lfloor x^e/n \rfloor$, where, as you know, $\lfloor x^e/n \rfloor$ is the greatest integer less than or equal to x^e/n; in other words, the unique integer in $\{0, 1, \ldots, n-1\}$ that is congruent to x^e modulo n, i.e., that is x^e minus some multiple of n. This last characterization is useful for proofs.

This problem can be solved in polynomial time. You might have seen a "fast" algorithm for this problem that takes "like, um, $\lg e$ multiplications, dude." That algorithm actually takes about $1.5 \lg e$ multiplications, and the same number of divisions by n with remainder.

We'll see a better exponentiation algorithm that takes about $(1+1/\lg\lg e)\lg e$ multiplications. A medium-size input might have x, e, n with hundreds of digits, let's say a thousand bits each; then $\lg e$ is about 1000, so $1 + 1/\lg\lg e$ is about 1.1, rather than the previous 1.5. That's a quite noticeable speedup. It's like turning your 2.2GHz Pentium into a 3GHz Pentium, or turning your 3GHz Pentium into a 4GHz Pentium, which is faster than anything Intel makes today.

We'll also see lower-level speedups, such as saving division time compared to multiplication time, and saving time in multiplication. These speedups combine to produce huge improvements. But, even by itself, this improvement from 1.5 to 1.1 is important. You can't see this improvement in the Cormen-Leiserson-Rivest-Stein world: $1.5 \lg e$ and $(1 + 1/\lg\lg e)\lg e$ are both oversimplified into $\Theta(\lg e)$.

The other obvious difference in flavor between 401 and this course is that I'm looking at different functions. 401 focuses on sorting, for example, and various graph operations such as shortest paths. This course focuses on a few cryptographic operations: for example, I'll start with a particular authentication function, a function that scrambles messages to prevent forgery.

You might have seen some of the underlying mathematics in previous courses on algebra, number theory, or cryptography. [Class feedback at this point: most people have had a first cryptography course; some people have had algebra and number theory.] None of these are prerequisites. The bad news, for those of you who've seen some cryptography before, is that a typical introductory cryptography course is probably the least relevant of all the related courses that I've mentioned. I'll be explaining modern cryptography from scratch.

By the way, it's possible for an advanced algorithms course to focus on the same functions as in 401—or just one of those functions! Knuth's famous *Art of computer programming* spends hundreds of pages on sorting algorithms.

A homework problem

I'll sometimes label a homework assignment as being required. Other homework assignments, such as this one, are optional and have nothing to do with your

grade, but they're worth exploring if you'd like to devote extra time to the course material.

On an UltraSPARC—more precisely, a 900MHz UltraSPARC III (specifically `icarus.cc.uic.edu`, which I believe all UIC students have accounts on)—the following program (compiled with `gcc -o uvw uvw.c -O3`; the `gcc` version, as reported by `gcc --version`, was 3.3.1) took almost exactly $4 \cdot 10^9$ clock cycles:

```
main()
{
  double u = 0;
  double v = 0;
  double w = 0;
  int i;
  for (i = 0;i < 1000000000;++i) {
    v *= w;
    u += v;
  }
  printf("%lf\n",u);
}
```

When I changed `v *= w` to `v *= u`, the program slowed down to $8 \cdot 10^9$ clock cycles. Why did it slow down? Can you explain the numbers 4 and 8? Can you explain the performance of the program on another CPU, such as a Pentium III? If that's too easy, try explaining the performance of a more complicated program, such as a heapsort program.