Good morning.

This is Computer Science 301, “Analysis of Algorithms.” I’m the instructor, John Stone. The sheet I’m handing out contains the usual facts about the course. Please read it carefully between now and Monday.

In today’s lecture, I’ll delimit the subject of the course by defining the term ‘algorithm’ and explaining the role of algorithms in human intellectual history. Then we’ll look in some detail at one particular algorithm to get a sense of the kinds of questions that we need to ask in order to arrive at a satisfying understanding of an algorithm.

The study of algorithms began with, and is motivated by, the human desire for answers to questions. For practical reasons, people prefer answers that are, in the first place, true (since one’s actions are more likely to have satisfactory results if one understands the world as it is) and, secondly, arrived at by objective methods, which make it possible to secure the agreement and cooperation of others (since others can apply the same objective methods to confirm the truth of one’s answers).

For instance, at some point before history began, people discovered an accurate and objective way to determine which, if either, of two groups of discrete things is more numerous than the other. One arranges the members of the groups in pairs, each pair comprising a different member from each group, until the members of one group are exhausted. If the other group is not yet exhausted, it is more numerous; otherwise, neither group is more numerous.

In many cases it is easy to perform the pairing operation correctly and easy to check that someone else is performing it correctly, so that the same answer is likely to be obtained no matter who performs the pairing and regardless of the circumstances in which it is conducted. Moreover, one needs neither great skill nor great faith in mathematics to understand why the method gives the right answer. So this ritual provides an impartial and commonly acceptable way to settle disputes.

In cases where the groups to be compared cannot be brought together to be paired off, one can pair off the members of either group with counters, such as pebbles, shells, or slashes on a wooden stick, and then carry the counters to the other group and pair again, with each counter representing the member of the absent group with which it was formerly paired. This method, too, produces reliable, credible, and reproducible answers.

Initially, such rituals arise only when the need to determine which of two groups of things is more numerous occurs frequently. They arise because sufficiently experienced people can perceive that the individual comparisons have a common structure — that the question whether Kohath has more sheep than Merari, and the question whether the tribe of Enan or the tribe of Ocran is larger, and the question whether Bezaleel or Aholiab possesses more items of silver plate, are all instances of the same problem, the problem of determining which of two groups of things is more numerous. The pairing ritual is a general way to address this problem, to solve any instance of it.

Whenever a number of questions have a similar structure, they are instances of the same problem, and frequently there is a common method by which any of them can be
answered. Given a question of the form “Which, if either, of two groups of discrete objects is more numerous?” the pairing ritual tells you, step by step, what to do in order to answer that question. And that’s what an algorithm is: an effective, step-by-step, computational method for solving any instance of a specified problem.

Perhaps there was a time when pairing was the only form of computation. However, if one performs the pairing ritual often, especially with the help of counters, the configurations of counters become familiar and take on identities and names of their own: one begins to recognize them as numbers. It then becomes possible to ask and answer a slightly more abstract question: How numerous is this group? How many things are in it? Furthermore, once one is accustomed to asking and answering this question, it becomes unnecessary to carry the counters around in order to compare groups. Instead, one need only record or remember the number of things in one group and use the same ritual for counting the things in the other group. The abstract sequence of numbers takes the place of physical counters.

In spite of this higher level of abstraction, counting remains an objective procedure. It delivers true answers that can be checked and confirmed by skeptics, and everyone who understands the counting ritual can agree on those answers. Counting, too, is an algorithm.

Next, having learned to perceive numbers as individual entities, a person can observe that they also stand in relationships and exemplify repeated structures and patterns—that there are general truths about numbers themselves, and objective ways to find those truths. On this basis one can formulate other questions and answer them by performing more elaborate computational rituals—addition, for instance, or the extraction of cube roots. Unlike pairing, which anyone can do with a little attention and patience, the rituals of arithmetic require training and practice. Some of them are intricate, long, and hard to understand. In such cases it is possible to separate the ritual from the understanding that grounds it, so that accepting the answers it delivers becomes an act of faith rather than reason. Also, faults in performance are common, even for well-trained practitioners.

Successive steps of abstraction and generalization lead from positive integers to ratios, negative numbers, irrational numbers, complex numbers, quaternions and octonions, Conway numbers, vectors, matrices, sequences, sets, functions, graphs, groups, fields, categories, and so on. In these rich mathematical universes we also find models and encodings for all kinds of non-numeric and even non-mathematical data, from individual Booleans and characters to texts, graphics, sounds, animations, and the like. Since we can specify the encoding processes themselves as computational rituals, the notion of computation that we now use is no longer tied to numbers, but applies to the processing of data of any kind.

With each new development, new abstractions made it possible to ask new questions, and new methods of determining the answers to those questions were developed. Until about 1890, however, four related difficulties or limitations severely constrained the performance of computational rituals:

- In the absence of any unambiguous general-purpose notation for recording algorithms, it was often difficult to figure out exactly how to enact a particular computation. Descriptions of algorithms in ordinary prose often omitted details, failed to explain
how to deal with exceptional cases, or required the performer to guess how to proceed at key points. (Consider, for instance, the process of computing the next digit of a quotient in the long-division ritual.)

- Since an algorithm could be, and often was, separated from the understanding that grounded and justified it, performers habitually followed rules on faith (“just because it works”). Unfortunately, like other kinds of faith, computational faith is extremely error-prone: If you don’t know exactly what you’re doing or why you’re doing it, there’s a much greater chance that you’re doing it wrong and that the results will be unsatisfactory.

- To make matters worse, there was frequently no simple way to detect a fault in the performance of an intricate computational ritual or, having detected a fault, to remedy it.

- Performing an algorithm that involved a large number of steps or a large quantity of data required extraordinary patience, care, and tenacity and even so usually yielded incorrect results. In the middle of the nineteenth century, a monomaniac hobbyist named William Shanks devoted twenty years of his free time to the computation of a high-precision value for \( \pi \), obtaining 707 digits after the decimal point. This stood as a sort of record for a single computation until the 1950s, when it was discovered that Shanks made a mistake that affected the 528th and all subsequent digits. The record for a collection of related computations performed without mechanical aids is probably the 1880 census of the United States, which took seven or eight years to complete and was almost certainly riddled with incorrect results.

In our time, the invention and development of stored-program computers have lifted the last of these constraints. We can now expect to compute 707 digits of \( \pi \) in a fraction of a second, while the 1880 census computations, which would probably be I/O-bound, might take a most of a minute, disk drives being slower than processors. A “long” computation today would be something like computing five hundred billion digits of \( \pi \), and we would expect all five hundred billion to be correct. A “large” data set today would be measured in terabytes or petabytes.

Similarly, the invention and development of high-level programming languages have largely removed the first constraint. It is now possible, indeed common, for the creator of an algorithm to express it exactly, completely, and unambiguously in the form of a program. There are still some difficulties. For instance, some authors naively and erroneously believe that floating-point representations obey the same arithmetic laws as the real numbers they approximate, and some fail to distinguish between integers, on one hand, and integers modulo some power of two on the other. But the standards of expression are so much higher today than they were before 1950 that I think we can boast of having almost solved this difficult intellectual problem.

We have made less progress with the other two difficulties. In fact, our increasing reliance on machines to perform computational rituals has exacerbated them. Millions of people now have calculators with square-root keys; few of them could either describe or perform an algorithm for computing a square root without mechanical aids, and only a fraction of those could explain why their algorithm works. Calculators are so reliable that
we seldom notice their limitations, and when they contain defects that produce incorrect answers we are apt not to notice the failures even when they would be obvious to a someone performing the same computation manually. (And when we are convinced that a calculator has failed, there is nothing to do about it except buy a new calculator.)

We have come to rely in daily life on the correctness of immense numbers of intricate computations that we do not understand. On one hand, this reliance reflects and underscores the fragile complexity of our civilization, with its extreme division of labor. It is enough that just a few people really understand the computation of square roots and can embody their understanding in calculating machines. Everyone else in the world who can afford a calculator then gets all the benefits of being able to compute square roots while remaining free to understand and do other useful things instead. On the other hand, those who take square roots completely on faith enter a kind of intellectual slavery, a tyranny imposed by the calculators and their makers — benevolent, or at least mutually profitable, when the computational rituals are correctly enacted, but as despicable as any other kind of tyranny when they are not.

Since some of you belong, or will belong, to the technological elite that imposes this tyranny, it is your responsibility to make sure that the algorithms on which others rely are flawless, and it is my responsibility to ensure that you have the knowledge and the tools that you’ll need to meet this standard. More generally, I hope to free all of you, as much as possible, from the ignorance and ineptitude that leads so many people to make disgraceful excuses: “There’s nothing I can do about it. The computer just does it that way.”

Let me illustrate the kind of study we shall make and the kind of knowledge that we shall obtain by taking, as an example, Euclid’s algorithm for finding the greatest common divisor of two positive integers — “the oldest nontrivial algorithm that has survived to the present day,” as Donald Knuth observes.

The reference is to section 4.5.2 of *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming* (Reading, Massachusetts: Addison–Wesley Publishing Company). The results that I’ll be presenting below are derived from Knuth’s discussion of Euclid’s algorithm in sections 4.5.2 and 4.5.3 of this book.

Let’s begin, in this case, with the ideas in which the algorithm is grounded. By definition, the greatest common divisor of two positive integers \( m \) and \( n \) is the greatest integer that evenly divides both of them. If \( n \) evenly divides \( m \), then \( n \) is the greatest common divisor, since obviously \( n \) is divisible by itself and by no greater integer. On the other hand, if \( n \) does not divide \( m \) evenly, but instead leaves a non-zero remainder \( r \), then \( \text{gcd}(m, n) = \text{gcd}(n, r) \).

This last idea may not be completely obvious, so let’s look at it a little, first with a picture (Figure 0).

Here \( m \), \( n \), and \( r \) are the widths of the pictured bars, and the lines along the bar representing \( m \) indicate that \( n \) goes into \( m \) three times with \( r \) left over. Now let \( d \) be any common divisor of \( m \) and \( n \), so that we get the following picture (Figure 1).

Since \( d \) goes evenly into \( n \) and \( m \), removing any multiple of \( n \) from \( m \) leaves a remainder that is also divisible by \( d \). Similarly, if \( d' \) is any common divisor of \( n \) and \( r \), then one can build up \( m \) out of multiples of \( d' \), so that \( m \) itself is a multiple of \( d' \).
It’s easy to develop this diagrammatic intuition into an algebraic proof. To say that dividing $m$ by $n$ leaves a remainder of $r$ means that $m = qn + r$ for some non-negative integer $q$. To say that $d$ is a common divisor of $m$ and $n$ means that $m = cd$ and $n = c'd$ for some positive integers $c$ and $c'$. So, by substitution, $cd = qc'd + r$; so $r = d(c - qc')$, where $c - qc'$ is an integer. So $r$ is divisible by $d$. Similarly, if $d'$ is a common divisor of $n$ and $r$, then, for some integers $e$ and $e'$, $m = qed' + e'd' = d'(qe + e')$, so $m$ is divisible by $d'$.

So every common divisor of $m$ and $n$ is a common divisor of $n$ and $r$, and vice versa. So the greatest common divisor of each pair of numbers is the same.

What makes it possible to develop a working algorithm out of these ideas is the fact that the remainder $r$ resulting from a division by $n$ is less than $n$, so that the equation $\gcd(m, n) = \gcd(n, r)$ provides a way to convert a difficult greatest-common-divisor question, involving very large values of $m$ and $n$, into an easier one. This should be a familiar pattern to all of you graduates of Computer Science 151: It’s a direct recursion.
(define (Euclidean-gcd m n)
  (let ((r (remainder m n)))
    (if (zero? r)
        n
        (Euclidean-gcd n r))))

If $r$ is zero, then $m$ is divisible by $n$, so that gcd$(m, n) = n$ by our first idea. If $r$ is not zero, then gcd$(m, n) =$ gcd$(n, r)$, by our second idea. So it seems that the Euclidean-gcd procedure should always give the right answer.

To qualify as an algorithm, this procedure must terminate, regardless of the values of $m$ and $n$. We can prove that it does. Whenever Euclidean-gcd invokes itself recursively, the value of the second argument is a positive integer resulting from the division of $m$ by $n$ — hence, a positive integer strictly less than $n$. In a succession of recursive calls, then, the successive values of the second argument form a monotonically decreasing series of positive integers. Any such sequence is finite (its length cannot be greater than the original argument $n$), so there must be a last recursive call.

If $m$ and $n$ are sufficiently large, it is not practical to use this algorithm even though we know that it will terminate with the correct answer. For instance, the enactment of the Euclidean-gcd procedure for some particular choices of $m$ and $n$ would require so many recursive calls that it would take a trillion years to complete them all on a present-day computer. It is therefore useful to know, as a function $f$ of $m$ and $n$, how many times the Euclidean-gcd procedure is called during the computation of their greatest common divisor.

The only way I know of to compute this quantity is to run a procedure like Euclidean-gcd except that it tallies the calls as they are made. Fortunately, it is relatively easy to establish an upper-bound function $g$ of $m$ and $n$ that serves as a worst-case guarantee, in the sense that the actual number of calls, $f(m, n)$, is less than or equal to the upper-bound number $g(m, n)$.

It turns out that we can define $g(m, n)$ to be $\left\lfloor \log_{\phi}(\sqrt{5}(n + 1)) \right\rfloor - 1$, where $\phi = (1 + \sqrt{5})/2$. I won’t give the complete proof, but I’ll give you the intuition. First, let’s consider only cases in which $n$ is less than $m$. The least such values of $m$ and $n$ for which only one call — the original one — is needed are $m = 2$, $n = 1$. Now, in the smallest case for which two calls are needed, the second argument and the remainder will be the $m$ and $n$ values for the one-call case; in other words, we can take $n$ to be 2 — the $m$ at the next level down — and $m$ to be the least positive integer such that the remainder on dividing $m$ by 2 is 1, which is the $n$ at the next level down. So two calls are needed for $m = 3$, $n = 2$. Next, for three calls, we can again take $n$ to be the $m$ at the next level down, and $m$ to be the least positive integer such that the remainder on dividing $m$ by 3 is 2. So $m = 5$, $n = 3$. Similarly, for four calls, $m = 8$ and $n = 5$; for five calls, $m = 13$ and $n = 8$, and so on.

If we now lift the restriction that $n$ be less than $m$, the new cases that we get are ones in which the only effect of the first call is to reverse the order of the arguments. For instance, if $m = 8$ and $n = 13$, the remainder is 8 and the first recursive call is, in effect,
(Euclidean-gcd 13 8). So actually the least case in which the procedure takes \( k + 1 \) steps is the one we found above as taking \( k \) steps, but with the arguments reversed. (See Table 0.)

<table>
<thead>
<tr>
<th>calls</th>
<th>( m )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 0.

Now, the column headed “calls” gives the number of calls in these “worst cases”; whenever \( n \) is less than or equal to some value that appears in the right-hand column of the table, the number of calls is less than or equal to the corresponding entry in the left-hand column. I expect that you’ll recognize the numbers in the right-hand column as terms of the Fibonacci sequence, which is defined by the recurrence equations

\[
F_0 = 0, \\
F_1 = 1, \\
F_{k+2} = F_k + F_{k+1} \quad \text{for every natural number } k.
\]

There is a closed form for this sequence:

\[
F_k = \left[ \frac{\phi^k}{\sqrt{5}} \right],
\]

where the brackets indicate that the quantity they enclose is to be rounded to the nearest integer.

So we can take \( g(m, n) \) to be one less than the index \( k \) of the greatest term \( F_k \) of the Fibonacci sequence that is less than or equal to \( n \). The closed form just shown enables us to find this index by starting with \( n + 1 \) (to ensure that we get the right result regardless of whether the brackets cause upwards or downwards rounding), multiplying by \( \sqrt{5} \), taking the logarithm to the base \( \phi \), and truncating.

This upper-bound function grows slowly compared to \( m \) and \( n \); for instance, when \( n \) is the largest integer that can be stored in a Java variable of type `long`, \( g(m, n) = 91 \).

One might also be interested in determining the average number of calls required to compute the greatest common divisor of two numbers. Strictly speaking, this makes no sense, because there are infinitely many possible choices of argument; but one can get a
sense of the average-case behavior of the algorithm by setting some upper bound $t$ and considering the average number $h(t)$ of calls over all choices of $m$ and $n$ both less than or equal to $t$, as a function of $t$. Again, I won’t give the details, but Knuth’s analysis leads to the estimate

$$\frac{12\ln 2}{\pi^2} \ln t + 1.06.$$ 

This is another slow-growth function. When $t$ is Java’s `Long.MAX_VALUE`, $h(t)$ is a little less than 38.

The actual running time of the Scheme procedure is not strictly proportional to the number of recursive calls, since multitasking, swapping, paging, and garbage collection all affect it in ways that are difficult to predict. Moreover, the amount of time required for one call to `remainder` is not constant; every Scheme implementation represents sufficiently large integers using data structures with more components for larger values, and the length of time required to enact the division algorithm depends on the sizes of the structures representing the dividend and divisor.

Let me try to summarize the questions that we want to raise about a proposed algorithm. There are some basic ones with which you are already familiar:

- What are the inputs or parameters of the algorithm? What kinds of data does it operate on?
- What problem is the algorithm supposed to solve? What is the result of enacting it?
- What preconditions must be satisfied in order for the algorithm to be enacted correctly?

The further questions that we asked above about Euclid’s algorithm were:

- Does the algorithm, correctly enacted, produce the correct answer whenever the preconditions are met? If so, why? Ideally, we’d like a mathematical proof that the result will always be the one we expect, that the postconditions will always be satisfied.
- How many steps will it take to obtain the answer to a particular question by enacting the ritual? Can we get an exact count? If not, can we demonstrate an upper bound? How many steps will we need on the average, given the size of the input parameters?

Sometimes it’s easy and straightforward to get answers to these questions, and in other cases it’s difficult or impossible, or at least no one has figured out yet how to get the answers. We’ll look at some problems of both kinds.