CS 70	Discrete Mathematics for CS	
Fall 2003	Wagner	Lecture 10

The next sequence of lectures in on the topic of *Arithmetic Algorithms*. We shall build up to an understanding of the RSA public-key cryptosystem.

Primality and Factoring

You are given a natural number — say, 307131961967 — and you are asked: *Is it a prime number?* You must have faced this familiar kind of question in the past. How does one decide if a given number is prime?

There is, of course, an algorithm for deciding this:

```
algorithm prime(x)
  y := 2
  repeat
    if x mod y = 0 then return(false);
    y := y + 1
  until y = x
  return(true)
```

Here by $x \bmod y$ we mean the remainder of the division of x by $y \neq 0$ ($x\%y$ in the C family). This algorithm correctly determines if a natural number $x > 2$ is a prime. It implements the definition of a prime number by checking exhaustively all possible divisors, from 2 up to $x-1$. But it is not a useful algorithm: it would be impractical to apply it even to the relatively modest specimen 307131961967 (and, as we shall see, modern cryptography requires that numbers with several hundreds of digits be tested for primality). It takes a number of steps that is proportional to its argument x —and, as we shall see, this is bad.

```
algorithm fasterprime(x)
  y := 2
  repeat
    if x mod y = 0 then return(false);
    y := y + 1
  until y * y ≥ x
  return(true)
```

Now, this is a little better. This algorithm checks all possible divisors up to the square root of x. And this suffices, because, if x had any divisors besides 1 and itself, then consider the smallest among these, and call it y. Thus, $x = y \cdot z$ for some integer z which is also a divisor of x other than one and itself. And since y is the smallest divisor, $z \geq y$. It follows that $y \cdot y \leq y \cdot z = x$, and hence y is no larger than the square root of x. And the second algorithm does indeed look for such a y.

Still, this algorithm is not satisfactory: in a certain well-defined sense it is "as exponential" as the exhaustive algorithm for satisfiability. To see why, we must understand how we evaluate the running time of algorithms with arguments that are natural numbers.

And you know such algorithms: e.g., the methods you learned in elementary school for adding, multiplying, and dividing whole numbers. To add two numbers, you have to carry out several elementary operations (adding two digits, remembering the carry, etc.), and the number of these operations is proportional to the *number of digits* n in the input: we express this by saying that the number of such operations is $O(n)$ (pronounced "big-Oh of n"). To multiply two numbers, you need a number of elementary operations (looking up the multiplication table, remembering the carry, etc.) that is proportional to the *square* of the number of digits, i.e., $O(n^2)$. (Make sure you understand why it is n^2).[1] In contrast, the first primality algorithm above takes time at least proportional to x, which is about 10^n, where n is the number of digits in x; the second one takes time at least proportional to $10^{\frac{n}{2}}$ — also exponential in n.

So, *in analyzing algorithms taking whole numbers as inputs, it is most informative to express the running time as a function of the number of digits in the input, not of the input itself.* It does not matter if n is the number of bits of the input or the number of decimal digits: As you know, these two numbers are within a small constant factor of each other —in particular, $\log_2 10 = 3.30\ldots$

We have thus arrived at a most important question: Is there a primality algorithm whose time requirements grow as a *polynomial* (like n, n^2, n^3, etc.) in the number n of digits of the input? As we shall see later in this class, the answer is "yes," such an algorithm does indeed exist. This algorithm has the following remarkable property: It determines whether or not x is prime *without* discovering a factor of x whenever x is composite (i.e., not prime). In other words, we would not find this algorithm by looking further down the path we started with our two algorithms above: it is not the result of clever ways of examining fewer and fewer possible divisors of x. And there is a good reason why our fast primality algorithm has to be like this: *There is no known polynomial algorithm for discovering the factors of a whole number*. Indeed, this latter problem, known as *factoring*, is strongly suspected to be hard, i.e., of *not* being solvable by *any* algorithm whose running time is polynomial (in n, of course). This combination of mathematical facts sounds almost impossible, but it is true: *Factoring is hard, primality is easy!* In fact, as we shall see, modern cryptography is based on this subtle but powerful distinction.

To understand algorithms for primality, factoring and cryptography, we first need to develop some more basic algorithms for manipulating natural numbers.

Computing the Greatest Common Divisor

The *greatest common divisor* of two natural numbers x and y, denoted $\gcd(x,y)$, is the largest natural number that divides them both. (Recall, 0 divides no number, and is divided by all.) How does one compute the gcd? By *Euclid's algorithm*, perhaps the first algorithm ever invented:

[1] The algorithm we are thinking of here is "long multiplication", as you learned in elementary school. There is in fact a recursive algorithm with running time about $O(n^{1.58})$, which you will see in CS170. The state of the art is a rather complex algorithm that achieves $O(n \log n \log\log n)$, which is only a little slower than linear in n.

```
algorithm gcd(x,y)
  if y = 0 then return(x)
  else return(gcd(y,x mod y))
```

This algorithm assumes that $x \geq y \geq 0$ and $x > 0$.

Theorem 10.1: *The algorithm above correctly computes the gcd of x and y in time $O(n)$, where n is the total number of bits in the input (x,y).*

Proof: Correctness is proved by (strong) induction on y, the smaller of the two input numbers. For each $y \geq 0$, let $P(y)$ denote the proposition that the algorithm correctly computes $\gcd(x,y)$ for all values of x such that $x \geq y$ (and $x > 0$). Certainly $P(0)$ holds, since $\gcd(x,0) = x$ and the algorithm correctly computes this in the `if`-clause. For the inductive step, we may assume that $P(z)$ holds for all $z < y$ (the inductive hypothesis); our task is to prove $P(y)$. The key observation here is that $\gcd(x,y) = \gcd(y, x \bmod y)$ — that is, replacing x by $x \bmod y$ does not change the gcd. This is because a divisor d of y also divides x if and only if it divides $x \bmod y$ (divisibility by d is not affected by adding or subtracting multiples of d, and y *is* a multiple of d). Hence the `else`-clause of the algorithm will return the correct value provided the recursive call `gcd(y,x mod y)` correctly computes the value $\gcd(y, x \bmod y)$. But since $x \bmod y < y$, we know this is true by the inductive hypothesis. This completes our verification of $P(y)$, and hence the induction proof.

Now for the $O(n)$ bound on the running time. It is obvious that the arguments of the recursive calls become smaller and smaller (because $y \leq x$ and $x \bmod y < y$). The question is, how fast? We shall show that, in the computation of $\gcd(x,y)$, after two recursive calls the first (larger) argument is smaller than x by at least a factor of two (assuming $x > 0$). There are two cases:

1. $y \leq \frac{x}{2}$. Then the first argument in the next recursive call, y, is already smaller than x by a factor of 2, and thus in the next recursive call it will be even smaller.
2. $x \geq y > \frac{x}{2}$. Then in two recursive calls the first argument will be $x \bmod y$, which is smaller than $\frac{x}{2}$.

So, in both cases the first argument decreases by a factor of at least two every two recursive calls. Thus after at most $2n$ recursive calls, where n is the number of bits in x, the recursion will stop (note that the first argument is always a natural number). $\square$

Note that the second part of the above proof only shows that the number of recursive calls in the computation is $O(n)$. We can make the same claim for the running time if we assume that each call only requires constant time. Since each call involves one integer comparsion and one mod operation, it is reasonable to claim that its running time is constant. In a more realistic model of computation, however, we should really make the time for these operations depend on the size of the numbers involved: thus the comparison would require $O(n)$ elementary (bit) operations, and the mod (which is really a division) would require $O(n^2)$ operations, for a total of $O(n^2)$ operations in each recursive call. (Here n is the maximum number of bits in x or y, which is just the number of bits in x.) Thus in such a model the running time of Euclid's algorithm is really $O(n^3)$.

Modular Arithmetic

Example: Calculating the day of the week. Suppose that you have mapped the sequence of days of the week (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday) to the sequence of numbers $(0,1,2,3,4,5,6)$ so that Sunday is 0, Monday is 1, etc. Suppose that today is Thursday (=4), and you want to calculate what day of the week will be 10 days from now. Intuitively, the answer is the remainder of $4+10=14$ when divided by 7, that is, 0 —Sunday. In fact, it makes little sense to add a number like 10 in this context, you should probably find *its* remainder modulo 7, namely 3, and then add this to 4, to find 7, which is 0.

What if we want to continue this in 10 day jumps? After 5 such jumps, we would have day $4+3\cdot 5=19$, which gives 5 modulo 7 (Friday).

This example shows that in certain circumstances it makes sense to do arithmetic within the confines of a particular number (7 in this example), that is, to do arithmetic by always finding the remainder of each number modulo 7, say, and repeating this for the results, and so on. As well as being efficient in the sense of keeping intermediate values as small as possible, this actually has some important applications to cryptography, as we shall see later.

We can write equations for this kind of arithmetic. The equations are written, for example:

$$4+10\cdot 5 = 40 \bmod 7.$$

(In certain books, the $=$ sign in such equations is written $\equiv$, and the mod m part often comes within parentheses.) You can read such equations in two ways. One way is to think of them as regular arithmetic equations, in which however the trailing "mod 7" means that we shall end up taking the remainder of both sides modulo 7 when doing the checking.

The second way is a little more sophisticated. If m is an integer (such as 7), define the following relation between integers: x and y are called *congruent modulo m*, written $x=y \bmod m$, iff they differ by a multiple of m — that is, $x-y=k\cdot m$ for some integer k (possibly negative). To put it otherwise, x and y are congruent modulo m if they have the same remainder modulo m. Notice that "congruent modulo m" is an *equivalence relation*: it partitions the integers into m equivalence classes $0,1,2,\ldots,m-1$.

Theorem 10.2: *If $a=c \bmod m$ and $b=d \bmod m$, then $a+b=c+d \bmod m$ and $a\cdot b=c\cdot d \bmod m$.*

Proof: We know that $c=a+k\cdot m$ and $d=b+\ell\cdot m$, so $c+d=a+k\cdot m+b+\ell\cdot m=a+b+(k+\ell)\cdot m$, which means that $a+b=c+d \bmod m$. The proof for multiplication is similar and left as an exercise. □

What this theorem tells us is that we can always reduce any arithmetic expression modulo m into a natural number smaller than m. As an example, consider the expresion $(13+11)\cdot 18 \bmod 7)$. Using Theorem 9.2 several times we can write:

$$\begin{aligned}(13+11)\cdot 18 &= (6+4)\cdot 4 \bmod 7\\ &= 10\cdot 4 \bmod 7\\ &= 3\cdot 4 \bmod 7\\ &= 12 \bmod 7\\ &= 5 \bmod 7.\end{aligned}$$

In summary, we can always do calculations modulo m by reducing intermediate results modulo m.

Exponentiation

A standard operation in arithmetic algorithms (including primality testing and RSA) is raising one number to a power modulo another number. I.e., how do we compute $x^y \bmod m$, where x, y, m are natural numbers and $m > 0$? A naïve approach would be to compute the sequence $x \bmod m, x^2 \bmod m, x^3 \bmod m, \ldots$ up to y terms, but this requires time exponential in the number of bits in y. We can do much better using the trick of *repeated squaring*:

```
algorithm mod-exp(x, y, m)
  if y = 0 then return(1)
  else
    z = mod-exp(x, y div 2, m)
    if y mod 2 = 0 then return(z * z mod m)
    else return(x * z * z mod m)
```

This algorithm uses the fact that any $y > 0$ can be written as $y = 2a$ or $y = 2a+1$, where $a = \lfloor \frac{y}{2} \rfloor$ (which we have wrtiten as `y div 2` in the above pseudo-code), plus the facts

$$x^{2a} = (x^a)^2; \qquad \text{and}$$
$$x^{2a+1} = x \cdot (x^a)^2.$$

As an exercise, you should use these facts to construct a formal inductive argument that the algorithm always returns the correct value.

What is its running time? The main task here, as is usual for recursive algorithms, is to figure out how many recursive calls are made. But we can see that the second argument, y, is being (integer) divided by 2 in each call, so the number of recursive calls is exactly equal to the number of bits, n, in y. (The same is true, up to a small constant factor, if we let n be the number of decimal digits in y.) Thus, if we charge only constant time for each arithmetic operation (`div`, `mod` etc.) then the running time of `mod-exp` is $O(n)$.

In a more realistic model, we need to look more carefully at the cost of each recursive call. Note first that the test on y in the `if`-statement just involves looking at the least significant bit of y, and the computation of $\lfloor \frac{y}{2} \rfloor$ is just a shift in the bit representation. Hence each of these operations takes only constant time. The cost of each recursive call is therefore dominated by the mod operation in the final result, which takes time $O(n^2)$, where n is the maximum number of bits in x or y. Thus in the more realistic model the running time of `mod-exp` on n-bit numbers is $O(n^3)$.

Inverses

Another key ingredient of arithmetic algorithms is multiplicative inverses modulo some number m. A *multiplicative inverse* of a positive integer x modulo m is a number a such that $ax = 1 \bmod m$. If we were working with rational numbers, then $a = \frac{1}{x}$ would be the (unique) inverse of x. In our present setting of modular arithmetic, can we be sure that x has an inverse mod m, and if so, is it unique (modulo m) and can we compute it?

As a first example, take $x = 8$ and $m = 15$. Then $2x = 16 = 1 \bmod 15$, so 2 is a multiplicative inverse of 8 mod 15. As a second example, take $x = 12$ and $m = 15$. Then the sequence $\{ax \bmod m : a = 0, 1, 2, \ldots\}$ is periodic, and takes on the values $\{0, 12, 9, 6, 3\}$ (check this!). Thus 12 *has no multiplicative inverse mod* 15.

So when *does* x have a multiplicative inverse modulo m? The answer is: iff $\gcd(m,x) = 1$. This condition means that x and m share no common factors (except 1), and is often expressed by saying that x and m are *relatively prime*. Moreover, when the inverse exists it is unique.

Theorem 10.3: *Let m,x be positive integers such that* $\gcd(m,x) = 1$. *Then x has a multiplicative inverse modulo m, and it is unique (modulo m).*

Proof: Consider the sequence of m numbers $0, x, 2x, \ldots (m-1)x$. We claim that these are all distinct modulo m. Since there are only m distinct values modulo m, it must then be the case that $ax = 1 \bmod m$ for exactly one a (modulo m). This a is the unique multiplicative inverse.

To verify the above claim, suppose that $ax = bx \bmod m$ for two distinct values a,b in the range $0 \leq a,b \leq m-1$. Then we would have $(a-b)x = 0 \bmod m$, or equivalently, $(a-b)x = km$ for some integer k (possibly zero or negative). But since x and m are relatively prime, it follows that $a-b$ must be an integer multiple of m. This is not possible since a,b are distinct non-negative integers less than m. □

Actually it turns out that $\gcd(m,x) = 1$ is also a *necessary* condition for the existence of an inverse: i.e., if $\gcd(m,x) > 1$ then x has no multiplicative inverse modulo m. You might like to try to prove this using a similar idea to that in the above proof.

Since we know that multiplicative inverses are unique when $\gcd(m,x) = 1$, we shall write the inverse of x as $x^{-1} \bmod m$. But how do we compute x^{-1}, given x and m? Let's take a slightly roundabout route. For any pair of numbers x,y, suppose we could not only compute $\gcd(x,y)$, but also find integers a,b such that

$$d = \gcd(x,y) = ax + by. \tag{1}$$

(Note that this is not a modular equation; and the integers a,b could be zero or negative.) For example, we can write $1 = \gcd(35,12) = -1 \cdot 35 + 3 \cdot 12$, so here $a = -1$ and $b = 3$ are possible values for a,b.

If we could do this then we'd be able to compute inverses, as follows. Apply the above procedure to the numbers m,x; this returns integers a,b such that

$$1 = \gcd(m,x) = am + bx.$$

But this means that $bx = 1 \bmod m$, so b is a multiplicative inverse of x modulo m. Reducing b modulo m gives us the unique inverse we are looking for. In the above example, we see that 3 is the multiplicative inverse of 12 mod 35.

So, we have reduced the problem of computing inverses to that of finding integers a,b that satisfy equation (1). Now since this problem is a generalization of the basic gcd, it is perhaps not too surprising that we can solve it with a fairly simple extension of Euclid's algorithm. The following algorithm *extended-gcd* takes as input a pair of natural numbers $x \geq y$ as in Euclid's algorithm, and returns a triple of integers (d,a,b) such that $d = \gcd(x,y)$ and $d = ax + by$:

```
algorithm extended-gcd(x,y)
  if y = 0 then return(x, 1, 0)
  else
    (d, a, b) := extended-gcd(y, x mod y)
    return((d, b, a - (x div y) * b))
```

Note that this algorithm has the same form as the basic gcd algorithm we saw earlier; the only difference is that we now carry around in addition the required values a,b. You should hand-turn the algorithm on the input $(x,y) = (35,12)$ from our earlier example, and check that it delivers correct values for a,b.

Let's now look at why the algorithm works. In the base case ($y = 0$), we return the gcd value $d = x$ as before, together with values $a = 1$ and $b = 0$ which satisfy $ax + by = d$. If $y > 0$, we first recursively compute values (d,a,b) such that $d = \gcd(y, x \bmod y)$ and

$$d = ay + b(x \bmod y). \tag{2}$$

Just as in our analysis of the vanilla algorithm, we know that this d will be equal to $\gcd(x,y)$. So the first component of the triple returned by the algorithm is correct.

What about the other two components? Let's call them A and B. What should their values be? Well, from the specification of the algorithm, they must be integers that satisfy

$$d = Ax + By. \tag{3}$$

To figure out what A and B should be, we need to rearrange equation (2), as follows:

$$\begin{aligned} d &= ay + b(x \bmod y) \\ &= ay + b(x - \lfloor x/y \rfloor y) \\ &= bx + (a - \lfloor x/y \rfloor b)y. \end{aligned}$$

(In the second line here, we have used the fact that $x \bmod y = x - \lfloor x/y \rfloor y$ — check this!) Comparing this last equation with equation (3), we see that we need to take $A = b$ and $B = a - \lfloor x/y \rfloor b$. This is exactly what the algorithm does, so we have concluded our proof of correctness.

Since the extended gcd algorithm has exactly the same recursive structure as the vanilla version, its running time will be the same up to constant factors (reflecting the increased time per recursive call). So once again the running time on n-bit numbers will be $O(n)$ arithmetic operations, and $O(n^3)$ bit operations. Combining this with our earlier discussion of inverses, we see that for any x,m with $\gcd(m,x) = 1$ we can compute $x^{-1} \bmod m$ in the same time bounds.