Number Theory

Margaret M. Fleck

10 Sept 2010

These notes cover concepts from elementary number theory, including Euclid's algorithm, corresponding to Rosen sections 3.4, 3.5, and 3.6.

1 Number Theory

We've now covered most of the basic techniques for writing proofs. So we're going to start applying them to specific topics in mathematics, starting with number theory.

Number theory is a branch of mathematics concerned with the behavior of integers. It has very important applications in cryptography and in the design of randomized algorithms. Randomization has become an increasingly important technique for creating very fast algorithms for storing and retrieving objects (e.g. hash tables), testing whether two objects are the same (e.g. MP3's), and the like. Much of the underlying theory depends on facts about which numbers evenly divide one another and which numbers are prime.

2 Factors and multiples

You've undoubtedly seen some of the basic ideas (e.g. divisibility) somewhat informally in earlier math classes. However, you may not be fully clear on what happens with special cases, e.g. zero, negative numbers. We also need clear formal definitions in order to write formal proofs. So, let's start with

Definition: Suppose that a and b are integers. Then a divides b if $b = an$ for some integer n. a is called a factor or divisor of b. b is called a multiple of a.

If b is non-zero and a is zero, clearly we can't have $b = an$. However, there is disagreement about how to handle the case where both a and b are both zero. Some authors allow zero to be a divisor and/or multiple of itself; some explicitly restrict one or both definitions so as to require a to be non-zero. I'll leave the definition in its simpler form, which disagrees with Rosen. This won't be a big problem because we won't have much use for this special case.

The shorthand for a divides b is $a \mid b$. Be careful about the order. The divisor is on the left and the multiple is on the right.

Some examples:

- $7 \mid 77$
- $77 \not\mid 7$
- $7 \mid 7$ because $7 = 7 \cdot 1$
- $7 \mid 0$ because $0 = 7 \cdot 0$, zero is divisible by any integer.
- $0 \not\mid 7$ because $0 \cdot n$ will always give you zero, never 7.
- $(-3) \mid 12$ because $12 = 3 \cdot -4$
- $3 \mid (-12)$ because $-12 = 3 \cdot -4$

A number p is even exactly when $2 \mid p$. The fact that zero is even is just a special case of the fact that zero is divisible by any integer.

3 Direct proof with divisibility

We can prove basic facts about divisibility in much the same way we proved basic facts about even and odd.

Claim 1 *For any integers a,b,and c, if $a \mid b$ and $a \mid c$ then $a \mid (b+c)$.*

> Proof: Let a,b,and c and suppose that $a \mid b$ and $a \mid c$.
>
> Since $a \mid b$, there is an integer k such that $b = ak$ (definition of divides). Similarly, since $a \mid c$, there is an integer j such that $c = aj$. Adding these two equations, we find that $(b+c) = ak + aj = a(k+j)$. Since k and j are integers, so is $k+j$. Therefore, by the definition of divides, $a \mid (b+c)$. $\square$

When we expanded the definition of divides for the second time, we used a fresh variable name. If we had re-used k, then we would have wrongly forced b and c to be equal.

The following two claims can be proved in a similar way:

Claim 2 *For any integers a,b,and c, if $a \mid b$ and $b \mid c$ then $a \mid c$. (Transitivity of divides.)*

Claim 3 *For any integers a,b, and c, if $a \mid b$, then $a \mid bc$.*

You've probably seen "transitivity" before in the context of inequalities. E.g. if $a < b$ and $b < c$, then $a < c$. We'll get back to the general notion of transitivity later in the term.

4 Prime numbers

We're all familiar with prime numbers from high school. Firming up the details:

> Definition: an integer $q \geq 2$ is prime if the only positive factors of q are q and 1. An integer $q \geq 2$ is composite if it is not prime.

For example, among the integers no bigger than 20, the primes are 2, 3, 5, 7, 11, 13, 17, and 19.

A key fact about prime numbers is

Fundanmental Theorem of Arithmetic: Every integer ≥ 2 can be written as the product of one or more prime factors. Except for the order in which you write the factors, this prime factorization is unique.

The word "unique" here means that there is only one way to factor each integer.

For example, $260 = 2 \cdot 5 \cdot 13$ and $180 = 2 \cdot 2 \cdot 3 \cdot 3 \cdot 5$.

We won't prove this theorem right now, because it requires a proof technique called "induction," which we won't see for three weeks.

There are quite fast algorithms for testing whether a large integer is prime. However, even once you know a number is composite, algorithms for factoring the number are all fairly slow. The difficulty of factoring large composite numbers is the basis for a number of well-known cryptographic algorithms (e.g. the RSA algorithm).

5 gcd and lcm

If c divides both a and b, then c is called a **common divisor** of a and b. The largest such number is the **greatest common divisor** of a and b. Shorthand for this is $\gcd(a, b)$.

You can find the GCD of two numbers by inspecting their prime factorizations and extracting the shared factors. For example, $70 = 2 \cdot 5 \cdot 7$ and $130 = 2 \cdot 5 \cdot 13$. So $\gcd(70, 130)$ is $2 \cdot 5 = 10$. Next week, we'll see a more efficient way to compute the GCD.

Similarly, a common multiple of a and b is a number c such that $a|c$ and $b|c$. The least common multiple (lcm) is the smallest such number. The lcm can be computed using the formula:

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

For example, $\text{lcm}(70, 130) = \frac{70 \cdot 130}{10} = 910$.

If two integers a and b share no common factors, then $\gcd(a, b) = 1$. Such a pair of integers are called **relatively prime**.

6 There are infinitely many prime numbers

We can now prove a classic result about prime numbers. So classic, in fact, that it goes back to Euclid, who lived around 300 B.C.

> Euclid's Theorem: There are infinitely many prime numbers.

This is a lightly disguised type of non-existence claim. The theorem could be restated as "there is no largest prime" or "there is no finite list of all primes." So this is a good situation for applying proof by contradiction.

> Proof: Suppose not. That is, suppose there were only finitely many prime numbers. Let's call them p_1, p_2, up through p_n.
>
> Consider $Q = p_1 p_2 \cdots p_n + 1$.
>
> If you divide Q by one of the primes on our list, you get a remainder of 1. So Q isn't divisible by any of the primes p_1, p_2, up through p_n. However, by the Fundamental Theorem of Arithmetic, Q must have a prime factor (which might be either itself or some smaller number). This contradicts our assumption that p_1, p_2,... p_n was a list of all the prime numbers. $\square$

Notice one subtlety. We're not claiming that Q must be prime. Rather, we're making the much weaker claim that Q isn't divisible by any of the first n primes. It's possible that Q might be divisible by another prime larger than p_n.

7 The division theorem

Now, suppose that you have an integer a and an integer b that might not evenly divide a. We can still divide a by b, but we may be left with a remainder. For example, if 13 is divided by 4, the quotient is 3 and the remainder is 1. When both inputs are positive, most people agree about how to do this, but it's less obvious what's supposed to happen when one or both of the inputs is negative.

Number theorists have a very precise idea of how to compute quotients and remainders and it's specified by the following theorem.

> The Division Algorithm: For any integers a and b, where b is positive, there are unique integers q and r such that $a = bq + r$ and $0 \leq r < b$.

The name of this theorem is traditional but misleading, since neither the theorem nor its proof is an algorithm in the modern sense.

The important constraint here is that the remainder cannot be negative and must be in the range $[0, b)$. This forces the equation $a = bq + r$ to have only one solution (the "unique" qualifier in the theorem). The remainder of a divided by b is written $a \bmod b$.

Computing $a \bmod b$ for some representative inputs, we get:

- $11 \mod 3 = 2$.
- $60 \mod 7 = 4$
- $-11 \mod 3 = 1$. (The quotient is -4.)
- $0 \mod 7 = 0$

Programming languages typically contain one or more arithmetic operators similar to mod, named things like mod, modulo, rem, or %. These frequently allow negative inputs for the divisor (b) as well as the dividend a. They don't always agree with the mathematical definition even when the divisor is positive. Worse yet, in some cases, the output is implementation dependent when either of the inputs is negative. The "modulo operation" entry on wikipedia has a nice table of what different languages do. This is sad, very sad. And it can cause hard-to-find bugs.

There are three take-home messages here. First, in this class, we will always use the number theory definition for mod. Second, always call this operation mod, to help make clear that it's the mathematical notion rather than (say) the C or Java function. Third, when programming, always check what your language does before feeding negative numbers to mod-type operators.

8 Proof of the division theorem

Proving the division theorem requires proving two separate claims. First, ("existence") that there are numbers q and r satisfying these equations. Second, ("uniqueness") that there is only one solution. Let's prove the existence part.

To do this, we need to make explicit an important property of the integers

> Well-ordering: Every non-empty set of natural numbers has a smallest element.

"Non-empty" simply means that the set actually contains something.

The well-ordering property of the integers probably seems obvious to you. However, it is important to state it explicitly, because it's not true for the real numbers. You've probably seen examples in calculus of infinite sequences of real numbers that approach a limit without reaching it. Well-ordering typically fails for such sequences. This is a key property that distinguishes the integers ("discrete") from the reals ("continuous").

Many variants of the well-ordering axiom are also true. E.g. as long as all the integers in the set are above some number b, then the set has a lowest integer. If all the integers in the set are below b, then the set must have a largest integer.

And now, to prove existence:

> Proof: Let a and b be integers, where b is positive. Consider the set of integers $a - bq$, for all integers q.

If we pick any $q \leq \frac{a}{b}$, then $a - bq$ is non-negative. So our set does contain some values. Using the well-ordering property, choose the smallest non-negative value of $a - bq$. Let's call it r.

Since $r = a - bq$, then $bq + r = a$. So r satisfies the first equation from the division theorem. The way we chose r guarantees that it's ≥ 0. So we just need to show that $r < b$.

Suppose that r were $\geq b$. Consider $r - b$. $r - b$ is less than r. It's still non-negative. And it must also have the form $a - bq$. (Reduce the size of q by 1.) So this contradicts our choice of r as the smallest non-negative integer of the form $a - bq$. Therefore, it must be the case that $r < b$. $\square$.

To prove the uniqueness part of the theorem, you would pick two possible solutions q_1,r_1 and q_2,r_2 to the pair of equations. Then fiddle with the algebra to show that the two solutions must be equal.

9 Congruence mod k

If two integers a and b have the same remainder mod k, they are said to be "congruent mod k." For example, 47 and 15 are congruent mod 8, because 47 mod 8 = 7 = 15 mod 8. You can do arithmetic on integers but only keep the remainders mod k rather than the full results of operations like addition and multiplication. This is called modular arithmetic. So, if we add 5 and 6 modulo 7, we get 11 mod 7, which is 4.

Modular arithmetic is often useful in describing periodic phenomena. For example, many facts about real-world schedules are determined by the hour of the day and the day of the week (e.g. this class meets Monday, Wednesday, and Friday). To describe these patterns, we use arithmetic mod 12 to describe the fact that hours of the day go up to 12 and then wrap around back to 1. And we can use arithmetic mod 7 to describe the fact that, within a month, days whose numbers are congruent mod 7 lie on the same day of the week.

The shorthand notation for a is congruent to b mod k is $a \equiv b \pmod{k}$. Logically, we really should write something like $a \equiv_k b$, so it looked like the notation for (say) a logarithm with a particular choice of base. In congruence mod k, k is really a modifier on our notation of equality ($\equiv$). However, $a \equiv b \pmod{k}$ has become the standard and we have to live with it.

To get used to this predicate, here are some concrete examples:

- $17 \equiv 5 \pmod{12}$ (Familiar to those of us who've had to convert between US 12-hour clocks and European or military 24-hour clocks.)
- $3 \equiv 10 \pmod{7}$
- $-3 \equiv 4 \pmod{7}$ (Since $(-3) + 7 = 4$.)
- $-3 \not\equiv 3 \pmod{7}$
- $-3 \equiv 3 \pmod{6}$
- $-21 \equiv -13 \pmod{8}$ (For both -21 and -13, the remainder mod 8 is 3.)

10 A digression on types

Argh. We're now using the notation "mod" in two related but quite distinct ways. How horribly confusing! Unfortunately, this "overloading" of the symbol mod has become standard, so you're stuck with just getting used to it.

To keep from being confused, first notice that we have recently seen two quite distinct sorts of operations on pairs of integers.

First, there are arithmetic operators, which output a number. These include functions like $+$, $*$, etc. Also gcd and lcm. And also $a \bmod k$, which returns the remainder if you divide a by k.

Second, there are predicates which output a boolean (true or false). These include $a < b$, $a = b$, $a \mid b$, and $a \equiv b \pmod{k}$.

Both in programming and mathematics, it's very important to be clear about whether a given function returns a number or a boolean, because arithmetic operations and predicates are used in very different ways.

When reading notation, you frequently have to decide whether mod is supposed to be the arithmetic operator or the boolean. Notice that the boolean version has the mod in parentheses and it always directly follows an equation using $\equiv$. When mod is used as an arithmetic operator, it's not in parentheses and there's usually no $\equiv$ nearby.

11 The official formal definition

To get a formal definition that makes proofs easy, we'll have to rephrase this idea slightly.

> Definition: If k is any positive integer, two integers a and b are congruent mod k if $k \mid (a-b)$.

This will be our official definition. It is then possible to prove that (as you might expect) that this definition is equivalent to testing whether the two numbers have the same remainder mod k. Specifically:

> Theorem: For any integers a and b and any positive integer k, $a \equiv b \pmod{k}$ if and only if $a \bmod k = b \bmod k$.

Let's try using our official definition to prove a simple fact about modular arithmetic:

Claim 4 *For any integers a, b, c, d, and k, k positive, if* $a \equiv b \pmod{k}$ *and* $c \equiv d \pmod{k}$*, then* $a + c \equiv b + d \pmod{k}$*.*

> Proof: Let a, b, c, d, and k be integers with k positive. Suppose that $a \equiv b \pmod{k}$ and $c \equiv d \pmod{k}$.
>
> Since $a \equiv b \pmod{k}$, $k \mid (a-b)$, by the definition of congruence mod k. Similarly, $c \equiv d \pmod{k}$, $k \mid (c-d)$.
>
> Since $k \mid (a-b)$ and $k \mid (c-d)$, we know by a lemma about divides from last week that $k \mid (a-b)+(c-d)$. So $k \mid (a+c)-(b+d)$
>
> But then the definition of congruence mod k tells us that $a+c \equiv b+d \pmod{k}$. □

12 The Euclidean algorithm

The obvious way to compute the gcd of two integers is to factor both into primes and extract the shared factors. This is easy for small integers. However, it quickly becomes very difficult for larger integers, both for humans and computers. Fortunately, there's a shortcut that allows us to compute the gcd very quickly.

Before launching into the algorithm itself, let's get clear on some useful facts about the gcd. First, what's $\gcd(0, 37)$? Every integer k divides 0, so the largest common divisor of 0 and 37 must be 37. And, in general, $\gcd(0, k) = k$ for any non-zero integer k.

Notice that $\gcd(0, 0)$ isn't defined. All integers are common divisors of 0 and 0, so there is no greatest one.

The algorithm depends on the following fact about the gcd, which is probably not obvious to you:

Claim 5 *For any integers a, b, q and r, if $a = bq + r$, then $\gcd(a, b) = \gcd(b, r)$.*

> Proof: Suppose that n is some integer which divides both a and b. Then n divides bq and so n divides $a - bq$. (E.g. use various lemmas about divides from last week.) But $a - bq$ is just r. So n divides r.
>
> By an almost identical line of reasoning, if n divides both b and r, then n divides a.
>
> So, the set of common divisors of a and b is exactly the same as the set of common divisors of b and r. But $\gcd(a, b)$ and $\gcd(b, r)$ are just the largest numbers in these two sets, so if the sets contain the same things, the two gcd's must be equal.

Notice that $r = a \bmod b$ satisfies the equation $a = bq + r$, so therefore

> Corollary: For any positive integers a and b, then $\gcd(a, b) = \gcd(b, a \bmod b)$.

The term “corollary” means that this fact is a really easy consequence of the preceding claim.

13 Euclidean algorithm

Given this fact, we can give a fast algorithm for computing gcd, which dates back to Euclid (around 300 B.C.).

```
procedure gcd(a,b: positive integers)
x := a
y := b
while y != 0
   begin
   r := x mod y
   x := y
   y := r
   end
return x
```

Let’s trace this algorithm on inputs $a = 105$ and $b = 252$. Traces should summarize the values of the most important variables.

x	y	$r = x \bmod y$
105	252	105
252	105	42
105	42	21
42	21	0
21	0	

Since x is smaller than y, the first iteration of the loop swaps x and y. After that, each iteration reduces the sizes of a and b, because $a \bmod b$ is smaller than b. In the last iteration, y has gone to zero, so we output the value of x which is 21.

To verify that this algorithm is correct, we need to convince ourselves of two things. First, it must halt, because each iteration reduces the magnitude of y. Second, by our corollary above, the value of $\gcd(x, y)$ does not change from iteration to iteration. Moreover, gcd$(x, 0)$ is x, for any non-zero integer x. So the final output will be the gcd of the two inputs a and b.

This is a genuinely very nice algorithm. Not only is it fast, but it involves very simple calculations that can be done by hand (without a calculator). It's much easier than factoring both numbers into primes, especially as the individual prime factors get larger. Most of us can't quickly see whether a large number is divisible by, say, 17.

14 Pseudocode

Notice that this algorithm is written in *pseudocode*. Pseudocode is an abstracted type of programming language, used to highlight the important structure of an algorithm and communicate between researchers who may not use the same programming language. It borrows many control constructs (e.g. the while loop) from imperative languages such as C. But details required only for a mechanical compiler (e.g. type declarations for all variables) are omitted and equations or words are used to hide details that are easy to figure out.

Actual C or Java code is **not** acceptable pseudocode.

If you have taken a programming course, pseudocode is typically easy to read. You won't need to write much pseudocode in this class, but you'll need to write a lot for CS 373. A common question is how much detail to use. Try to use about the same amount as in the examples shown in class.

Appendix 3 of Rosen shows the specific conventions used in the textbook. We'll try to use something similar. However, it's not necessary to adhere religiously to a precise set of conventions, because pseudocode is read by a human, not a computer.

15 A recursive version of gcd

We can also write gcd as a recursive algorithm

```
procedure gcd(a,b: positive integers)
```

```
r := a mod b
if (r = 0) return b
else return gcd(b,r)
```

This code is very simple, because this algorithm has a natural recursive structure. Our corollary allows us to express the gcd of two numbers in terms of the gcd of a smaller pair of numbers. That is to say, it allows us to reduce a larger version of the task to a smaller version of the same task.