Chapter 4

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 integers evenly divide one another and which integers are prime.

4.1 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.

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. Zero is a factor of only one number: zero.
- $(-3) \mid 12$ because $12 = 3 \cdot -4$
- $3 \mid (-12)$ because $-12 = 3 \cdot -4$

An integer 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.

4.2 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 20 *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, $a \mid (b + c)$. □

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

Claim 22 *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.

4.3 Stay in the Set

Students are sometimes tempted to rephrase the definition of $a \mid b$ as "$\frac{b}{a}$ is an integer." This is not a good idea because it introduces a non-integer rational number into a problem that only needs to involve integers. Stepping outside the set of interest in this way is occasionally useful, but more often it leads to inelegant and/or buggy solutions. This happens for three reasons:

- The purely integer proofs are typically simpler.
- When constructing math from the ground up, the integers are typically constructed first and the rationals built from them. So using rationals to prove facts about integers can lead to circular proofs.
- On computers, integer operations yield exact answers but floating point operations are only approximate. So implementing an integer calculation using real numbers often introduces errors.

4.4 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. Numbers smaller than 2 are neither prime nor composite.

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, $260 = 2 \cdot 2 \cdot 5 \cdot 13$ 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 haven't seen yet.

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).

4.5 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, $140 = 2^2 \cdot 5 \cdot 7$ and $650 = 2 \cdot 5^2 \cdot 13$. So $\gcd(140, 6500)$ is $2 \cdot 5 = 10$.

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 positive number for which this is true. The lcm can be computed using the formula:

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

For example, $\text{lcm}(140, 650) = \frac{140 \cdot 650}{10} = 9100$.

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

If k is a non-zero integer, then k divides zero. the largest common divisor of k and zero is k. So $\gcd(k, 0) = \gcd(0, k) = k$. However, $\gcd(0, 0)$ isn't defined. All integers are common divisors of 0 and 0, so there is no greatest one.

4.6 The division 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 fast method, called the Euclidean algorithm. Before presenting this algorithm, we need some background about integer division.

In general, when we divide one integer by another, we get a quotient and a remainder:

Theorem 1 (Division Algorithm) *The Division Algorithm: For any integers a and b, where b is positive, there are unique integers q (the quotient) and r (the remainder) such that $a = bq + r$ and $0 \leq r < b$.*

For example, if 13 is divided by 4, the quotient is 3 and the remainder is

1. Notice that the remainder is required to be non-negative. So -10 divided by 7 has the remainder 4, because $(-10) = 7 \cdot (-2) + 4$. This is the standard convention in mathematics. The word “unique” in this theorem simply means that only one pair of number q and r satisfy the pair of equations.

Now, notice the following non-obvious fact about the gcd:

Claim 23 *For any integers a, b, q and r, where b is positive, 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.

If r is the remainder when a is divided by b, then $a = bq + r$, for some integer q. So we can immediately conclude that:

> Corollary: Suppose that a and b are integers and b is positive. Let r be the remainder when a is divided by b. Then $\gcd(a, b) = \gcd(b, r)$.

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

4.7 Euclidean algorithm

We can now give a fast algorithm for computing gcd, which dates back to Euclid. Suppose that $remainder(a, b)$ returns the remainder when a is divided by b. Then we can compute the gcd as follows:

```
gcd(a,b: positive integers)
   x := a
   y := b
   while (y > 0)
      begin
      r := remainder(x,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	$r = \text{remainder}(x, y)$
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.

4.8 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.

If you have taken a programming course, pseudocode is typically easy to read. Many small details are not standardized, e.g. is the test for equality written = or ==? However, it's usually easy for a human (though not a computer) to figure out what the author must have intended.

A common question is how much detail to use. Try to use about the same amount as in the examples shown in the notes. And think about how easily your pseudocode could be read by a classmate. Actual C or Java code is almost never acceptable pseudocode, because it is way too detailed.

4.9 A recursive version of gcd

We can also write gcd as a recursive algorithm

```
procedure gcd(a,b: positive integers)
r := remainder(a,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.

4.10 Congruence mod k

Many applications of number theory, particularly in computer science, use modular arithmetic. In modular arithmetic, there are only a finite set of numbers and addition "wraps around" from the highest number to the lowest one. This is true, for example, for the 12 hours on a standard US clock: 3 hours after 11 o'clock is 2 o'clock, not 14 o'clock.

The formal mathematical definitions of modular arithmetic are based on the notion of congruence. Specifically, two integers are "congruent mod k" if they differ by a multiple of k. Formally:

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

Notice that $k \mid (a-b)$ if and only if $k \mid (b-a)$. So it doesn't matter which number is subtracted from the other.

For example:

- $17 \equiv 5 \pmod{12}$ (Familiar to those of us who've had to convert between US 12-hour clocks and European/military 24-hour clocks.)
- $3 \equiv 10 \pmod{7}$
- $3 \equiv 38 \pmod{7}$ (Since $38 - 3 = 35$.)
- $38 \equiv 3 \pmod{7}$
- $-3 \equiv 4 \pmod{7}$ (Since $(-3) + 7 = 4$.)
- $-3 \not\equiv 3 \pmod{7}$
- $-3 \equiv 3 \pmod{6}$
- $-29 \equiv -13 \pmod{8}$ (Since $(-13) - (-29) = 16$.)

Congruence mod k is a relaxation of our normal rules for equality of numbers, in which we agree that certain pairs of numbers will be considered interchangeable.

4.11 Proofs with congruence mod k

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

Claim 24 *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 (above) 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 $a + c \equiv b + d \pmod{k}$. $\square$

This proof can easily be modified to show that

Claim 25 *For any integers a, b, c, d, and k, k positive, if $a \equiv b \pmod{k}$ and $c \equiv d \pmod{k}$, then $ac \equiv bd \pmod{k}$.*

So standard arithmetic operations interact well with our relaxed notion of equality.

4.12 Equivalence classes

The true power of modular conguence comes when we gather up a group of conguent integers and treat them all as a unit. Such a group is known as a **congruence class** or an **equivalence class**. Specifically, suppose that we fix a particular value for k. Then, if x is an integer, the equivalence class of x (written $[x]$) is the set of all integers congruent to x mod k. Or, equivalently, the set of integers that have remainder x when divided by k.

For example, suppose that we fix k to be 7. Then

$$[3] = \{3, 10, -4, 17, -11, \ldots\}$$
$$[1] = \{1, 8, -6, 15, -13, \ldots\}$$
$$[0] = \{0, 7, -7, 14, -14, \ldots\}$$

Notice that $[-4]$, and $[10]$ are exactly the same set as $[3]$. That is $[-4] = [10] = [3]$. So we have one object (the set) with many different names (one per integer in it). This is like a student apartment shared by Fred, Emily, Ali, and Michelle. The superficially different phrases "Emily's apartment" and "Ali's apartment" actually refer to one and the same apartment.

Having many names for the same object can become confusing, so people tend to choose a special preferred name for each object. For the k equivalence classes of integers mod k, mathematicians tend to prefer the names $[0], [1], \ldots, [k-1]$. Other names (e.g. $[30]$ when $k = 7$) tend to occur only as intermediate results in calculations.

Because standard arithmetic operations interact well with modular congruence, we can set up a system of arithmetic on these equivalence classes. Specifically, we define addition and multiplication on equivalence classes by:

$$[x] + [y] = [x + y]$$
$$[x] * [y] = [x * y]$$

So, (still setting $k = 7$) we can do computations such as

$$[4] + [10] = [4 + 10] = [14] = [0]$$
$$[-4] * [10] = [-4 * 10] = [-40] = [2]$$

This new set of numbers ($[0], [1], \ldots, [k-1]$), with these modular rules of arithmetic and equality, is known as the "integers mod k" or $\mathbb{Z}_k$ for short. For example, the addition and multiplication tables for $\mathbb{Z}_4$ are:

$+_4$	[0]	[1]	[2]	[3]
[0]	[0]	[1]	[2]	[3]
[1]	[1]	[2]	[3]	[0]
[2]	[2]	[3]	[0]	[1]
[3]	[3]	[0]	[1]	[2]

$\times_4$	[0]	[1]	[2]	[3]
[0]	[0]	[0]	[0]	[0]
[1]	[0]	[1]	[2]	[3]
[2]	[0]	[2]	[0]	[2]
[3]	[0]	[3]	[2]	[1]

People making extensive use of modular arithmetic frequently drop the square brackets. We're keeping the brackets for the moment to help you understand more clearly how the modular integers are created from the normal integers.

4.13 Wider perspective on equivalence

Modular arithmetic is often useful in describing periodic phenomena. For example, hours of the day form a circular space that is $\mathbb{Z}_{12}$. However, the tradition in time-keeping is to prefer the names $[1], [2], \ldots, [12]$ rather than the $[0], [2], \ldots, [11]$ traditional in math.

Low-precision integer storage on computers frequently uses modular arithmetic. For example, values in digitized pictures are often stored as 8-bit unsigned numbers. These numbers range from 0 to 255, i.e. the structure is $\mathbb{Z}_{256}$. In $\mathbb{Z}_{256}$, if you add [17] to [242], the result will be [3].

Equivalence classes of a more general type are used to construct rational numbers. That is, rational numbers are basically fractions, except that two fractions $\frac{x}{y}$ and $\frac{p}{q}$ are considered equivalent if $xq = py$. So, for example, the set $[\frac{2}{3}]$ would contain values such as $\frac{2}{3}$, $\frac{6}{9}$, and $\frac{-4}{-6}$. Most people prefer to name a rational number using a fraction in lowest terms.

Further afield, notice that musicians have more than one name for each note. There are seven note names, each of which can be sharped or flatted. However, there are only 12 different notes in a standard Western scale: many of these 21 names actually refer to the same note.[1] For example, A# is the same note as Bb. So, using more mathematical notation than the musicians would find comfortable, we could say that $[A\#]$ contains both A# and Bb.

[1]More precisely, this is true on instruments like pianos. More subtle note distinctions are possible on some other instruments.

4.14 Variation in Terminology

In these notes, a divides b is defined to be true if $b = an$ for some integer n. There is some variation among authors as to what happens when a is zero. Clearly, a non-zero number can't be a multiple of zero. But is zero a multiple of itself? According to our definition, it is, but some authors explicitly exclude this special case. Fortunately, this is a special case that one rarely sees in practice. The greatest common divisor is also known as the highest common factor (HCF).

In the shorthand notation $a \equiv b \pmod{k}$, the notation $\pmod{k}$ is logically a modifier on our notion of equality ($\equiv$). In retrospect, it might have made more sense to write something like $a \equiv_k b$. However, $a \equiv b \pmod{k}$ has become the standard notation and we have to live with it.

There are many variant notations for the quotient and remainder created by integer division, particularly if you include the functions built in to most programming languages. Popular names for the remainder include mod, modulo, rem, remainder, and %. The behavior on negative inputs differs from language to language and, in some cases, from implementation to implementation.[2] This lack of standardization often results in hard-to-find program bugs.

Some authors use $\mathbb{Z}/n\mathbb{Z}$ instead of $\mathbb{Z}_n$ as the shorthand name for the integers mod n. The notation $\mathbb{Z}_n$ may then refer to a set which is structurally the same as the integers mod n, but in which multiplication and exponentiation are used as the basic operations, rather than addition and multiplication. The equivalence class of n is sometimes written $\overline{n}$.

[2]The "modulo operation" entry on wikipedia has a nice table of what happens in different languages.