2/24

7 Recursive Data Types

Recursive data types play a central role in programming, and induction is really all about them.

Recursive data types are specified by *recursive definitions* that say how to contruct new data elements from previous ones. Along with each recursive data type there are recursive definitions of properties or functions on the data type. Most importantly, based on a recursive definition, there is a *structural induction* method for proving that all data of the given type have some property.

This chapter examines a few examples of recursive data types and recursively defined functions on them:

- strings of characters,
- the "balanced" strings of brackets,
- the nonnegative integers, and
- arithmetic expressions.

7.1 Recursive Definitions and Structural Induction

We'll start off illustrating recursive definitions and proofs using the example of character strings. Normally we'd take strings of characters for granted, but it's informative to treat them as a recursive data type. In particular, strings are a nice first example because you will see recursive definitions of things that are easy to understand or you already know, so you can focus on how the definitions work without having to figure out what they are for.

Why?

Definitions of recursive data types have two parts:

- **Base case(s)** specifying that some known mathematical elements are in the data type, and
- **Constructor case(s)** that specify how to construct new data elements from previously contructed element or from base elements.

The definition of strings over a given character set, A, follows this pattern:

? like a recursive function?

Definition 7.1.1. Let A be a nonempty set called an *alphabet*, whose elements are referred to as *characters*, *letters*, or *symbols*. The recursive data type, A^*, of strings over alphabet, A, are defined as follows:

- **Base case:** the empty string, λ, is in A^*.
- **Constructor case:** If $a \in A$ and $s \in A^*$, then the pair $\langle a, s\rangle \in A^*$.

So $\{0, 1\}^*$ are supposed to be the binary strings.

The usual way to treat binary strings is as sequences of 0's and 1's. For example, we have identified the length-4 binary string 1011 as a sequence of bits, of a 4-tuple, namely, $(1, 0, 1, 1)$. But according to the recursive Definition 7.1.1, this string would be represented by nested pairs, namely

$$\langle 1, \langle 0, \langle 1, \langle 1, \lambda\rangle\rangle\rangle\rangle\,.$$

exactly

These nested pairs are definitely cumbersome, and may also seem bizarre, but they actually reflect the way lists of characters would be represented in programming languages like Scheme or Python, where $\langle a, s\rangle$ would be correspond to cons(a, s).

Notice that we haven't said exactly how the empty string is represented. It really doesn't matter as long as we can recognize the empty string and not confuse it with any nonempty string.

Continuing the recursive approach, let's define the length of a string.

Definition 7.1.2. The length, $|s|$, of a string, s, is defined recursively based on the definition of $s \in A^*$:

Base case: $|\lambda| ::= 0$.

Constructor case: $|\langle a, s\rangle| ::= 1 + |s|$.

This definition of length follows a standard pattern: functions on recursive data types can be defined recursively using the same cases as the data type definition. Namely, to define a function, f, on a recursive data type, define the value of f for the base cases of the data type definition, and then define the value of f in each constructor case in terms of the values of f on the component data items.

Let's do another example: the *concatenation* $s \cdot t$ of the strings s and t is the string consisting of the letters of s followed by the letters of t. This is a perfectly clear mathematical definition of concatenation (except maybe for what to do with the empty string), and in terms of Scheme/Python lists, $s \cdot t$ would be the list append(s, t). Here's a recursive definition of concatenation.

Definition 7.1.3. The *concatenation* $s \cdot t$ of the strings $s, t \in A^*$ is defined recursively based on the definition of $s \in A^*$:

Base case:

$$\lambda \cdot t ::= t.$$

Constructor case: what is $\langle a, s \rangle$ here?

$$\langle a, s \rangle \cdot t ::= \langle a, s \cdot t \rangle .$$

Structural induction is a method for proving that all the elements of a recursively ~~defined data type have~~ some property. A structural induction proof has two parts corresponding to the recursive definition:

- Prove that each base case element has the property.
- Prove that each constructor case element has the property, when the constructor is applied to elements that have the property.

For example, we can verify the familiar fact that the length of the concatenation of two strings is the sum of their lengths using structural induction:

Theorem 7.1.4. *For all $s, t \in A^*$,*

$$|s \cdot t| = |s| + |t|.$$

Proof. By structural induction on the definition of $s \in A^*$. The induction hypothesis is

$$P(s) ::= \forall t \in A^*. \, |s \cdot t| = |s| + |t|.$$

Base case ($s = \lambda$):

$$\begin{aligned} |s \cdot t| &= |\lambda \cdot t| \\ &= |t| && \text{(def }\cdot\text{, base case)} \\ &= 0 + |t| \\ &= |s| + |t| && \text{(def length, base case)} \end{aligned}$$

Constructor case: Assume the induction hypothesis, $P(s)$, and let $r = \langle a, s \rangle \cdot t$. We must show that $P(r)$ holds:

$$\begin{aligned} |r \cdot t| &= |\,\langle a, s \rangle \cdot t| \\ &= |\,\langle a, s \cdot t \rangle\,| && \text{(concat def, constructor case)} \\ &= 1 + |s \cdot t| && \text{(length def, constructor case)} \\ &= 1 + (|s| + |t|) && \text{since } P(s) \text{ holds} \\ &= (1 + |s|) + |t| \\ &= |\,\langle a, s \rangle\,| + |t| && \text{(length def, constructor case)} \\ &= |r| + |t|. \end{aligned}$$

This completes the proof of the constructor case, so by structural induction we conclude that $P(s)$ holds for all strings $s \in A^*$. ∎

This proof illustrates the general principle:

The Principle of Structural Induction.
Let P be a predicate on a recursively defined data type R. If

- $P(b)$ is true for each base case element, $b \in R$, and
- for all two argument constructors, **c**,

$$[P(r) \text{ AND } P(s)] \text{ IMPLIES } P(\mathbf{c}(r,s))$$

for all $r, s \in R$,
and likewise for all constructors taking other numbers of arguments,

then

$$P(r) \text{ is true for all } r \in R.$$

The number, $\#_c(s)$, of occurrences of the character $c \in A$ in the string s has a simple recursive definition based on the definition of $s \in A^*$:

Definition 7.1.5.
Base case: $\#_c(\lambda) ::= 0$.
Constructor case:

$$\#_c(\langle a, s\rangle) ::= \begin{cases} \#_c(s) & \text{if } a \neq c, \\ 1 + \#_c(s) & \text{if } a = c. \end{cases}$$

So simple I would not have thought of it

We'll need the following lemma in the next section:

Lemma 7.1.6.

$$\#_c(s \cdot t) = \#_c(s) + \#_c(t).$$

The easy proof by structural induction is an exercise (Problem 7.6).

7.2 Strings of Matched Brackets

Let $\{], [\}^*$ be the set of all strings of square brackets. For example, the following two strings are in $\{], [\}^*$:

$$\texttt{[]][[[[[]]} \quad \text{and} \quad \texttt{[[[]][]][]} \tag{7.1}$$

A string, $s \in \{],[\}^*$, is called a *matched string* if its brackets "match up" in the usual way. For example, the left hand string above is not matched because its second right bracket does not have a matching left bracket. The string on the right is matched.

We're going to examine several different ways to define and prove properties of matched strings using recursively defined sets and functions. These properties are pretty straightforward, and you might wonder whether they have any particular relevance in computer scientist. The honest answer is "not much relevance, *any more*." The reason for this is one of the great successes of computer science as explained in the text box below.

Expression Parsing

During the early development of computer science in the 1950's and 60's, creation of effective programming language compilers was a central concern. A key aspect in processing a program for compilation was expression parsing. One significant problem was to take an expression like

$$x + y * z^2 \div y + 7$$

and *put in* the brackets that determined how it should be evaluated —should it be

$$[[x + y] * z^2 \div y] + 7, \text{ or,}$$
$$x + [y * z^2 \div [y + 7]], \text{ or,}$$
$$[x + [y * z^2]] \div [y + 7], \text{ or}\ldots?$$

The Turing award (the "Nobel Prize" of computer science) was ultimately bestowed on Robert W Floyd, for, among other things, being discoverer of a simple program that would insert the brackets properly.

process described somewhere?

In the 70's and 80's, this parsing technology was packaged into high-level compiler-compilers that automatically generated parsers from expression grammars. This automation of parsing was so effective that the subject no longer demanded attention. It largely disappeared from the computer science curriculum by the 1990's.

The matched strings can be nicely characterized as a recursive data type:

Definition 7.2.1. Recursively define the set, RecMatch, of strings as follows:

- **Base case:** $\lambda \in$ RecMatch.

- **Constructor case:** If $s, t \in \text{RecMatch}$, then
$$[\, s\,]\, t \in \text{RecMatch}.$$

Here $[\, s\,]\, t$ refers to the concatenation of strings which would be written in full as
$$[\, \cdot (s \cdot (] \cdot t)).$$

From now on, we'll usually omit the "$\cdot$'s."

Using this definition, $\lambda \in \text{RecMatch}$ by the Base case, so letting $s = t = \lambda$ in the constructor case implies
$$[\, \lambda\,]\, \lambda = [\,] \in \text{RecMatch}.$$

Now,
$$\begin{aligned} [\, \lambda\,]\, [\,] &= [\,]\, [\,] \in \text{RecMatch} && (\text{letting } s = \lambda, t = [\,]) \\ [\, [\,]\,]\, \lambda &= [\, [\,]\,] \in \text{RecMatch} && (\text{letting } s = [\,], t = \lambda) \\ & [\, [\,]\,]\, [\,] \in \text{RecMatch} && (\text{letting } s = [\,], t = [\,]) \end{aligned}$$

are also strings in RecMatch by repeated applications of the Constructor case; and so on.

It's pretty obvious that in order for brackets to match, there better be an equal number of left and right ones. For further practice, let's carefully prove this from the recursive definitions.

Lemma. *Every string in RecMatch has an equal number of left and right brackets.*

Proof. The proof is by structural induction with induction hypothesis
$$P(s) ::= \#_{[}\,(s) = \#_{]}\,(s).$$

Base case: $P(\lambda)$ holds because
$$\#_{[}\,(\lambda) = 0 = \#_{]}\,(\lambda)$$

by the base case of Definition 7.1.5 of $\#_c()$.

Constructor case: By structural induction hypothesis assume $P(s)$ and $P(t)$ and must show $P([\,s\,]\,t)$:

$$
\begin{aligned}
\#_{[}\,([\,s\,]\,t) &= \#_{[}\,([\,) + \#_{[}\,(s) + \#_{[}\,(]\,) + \#_{[}\,(t) && \text{(Lemma 7.1.6)}\\
&= 1 + \#_{[}\,(s) + 0 + \#_{[}\,(t) && (\text{def } \#_{[}\,())\\
&= 1 + \#_{]}\,(s) + 0 + \#_{]}\,(t) && (\text{by } P(s) \text{ and } P(t))\\
&= 0 + \#_{]}\,(s) + 1 + \#_{]}\,(t) \\
&= \#_{]}\,([\,) + \#_{]}\,(s) + \#_{]}\,(]\,) + \#_{]}\,(t) && (\text{def } \#_{]}\,())\\
&= \#_{]}\,([\,s\,]\,t) && \text{(Lemma 7.1.6)}
\end{aligned}
$$

This completes the proof of the constructor case. We conclude by structural induction that $P(s)$ holds for all $s \in \text{RecMatch}$. ■

Warning: When a recursive definition of a data type allows the same element to be constructed in more than one way, the definition is said to be *ambiguous*. We were careful to choose an *un*ambiguous definition of RecMatch to ensure that functions defined recursively on its definition would always be well-defined. Recursively defining a function on an ambiguous of a data type definition usually will not work. To illustrate the problem, here's another definition of the matched strings.

Definition 7.2.2. Define the set, AmbRecMatch $\subseteq \{\,]\,,[\,\}^*$ recursively as follows:

- **Base case:** $\lambda \in \text{AmbRecMatch}$,
- **Constructor cases:** if $s, t \in \text{AmbRecMatch}$, then the strings $[\,s\,]$ and st are also in AmbRecMatch.

It's pretty easy to see that the definition of AmbRecMatch is just another way to define RecMatch, that is AmbRecMatch = RecMatch (see Problem 7.10). The definition of AmbRecMatch is arguably easier to understand, but we didn't use it becaused it's ambiguous, while the trickier definition of RecMatch is unambiguous. Here's why this matters. Let's define the number of operations, $f(s)$, to construct a matched string s recursively on the definition of $s \in \text{AmbRecMatch}$:

$$
\begin{aligned}
f(\lambda) &::= 0, && (f \text{ base case})\\
f([\,s\,]\,) &::= 1 + f(s), \\
f(st) &::= 1 + f(s) + f(t). && (f \text{ concat case})
\end{aligned}
$$

This definition may seem ok, but it isn't: $f(\lambda)$ winds up with two values, and consequently:

$$\begin{aligned}
0 &= f(\lambda) && (f \text{ base case)})\\
&= f(\lambda \cdot \lambda) && \text{(concat def, base case)}\\
&= 1 + f(\lambda) + f(\lambda) && (f \text{ concat case}),\\
&= 1 + 0 + 0 = 1 && (f \text{ base case}).
\end{aligned}$$

This is definitely not a situation we want to be in!

7.3 Recursive Functions on Nonnegative Integers

The nonnegative integers can be understood as a recursive data type.

Definition 7.3.1. The set, $\mathbb{N}$, is a data type defined recursivly as:

- $0 \in \mathbb{N}$.
- If $n \in \mathbb{N}$, then the *successor*, $n + 1$, of n is in $\mathbb{N}$. add one

This of course makes it clear that ordinary induction is simply the special case of structural induction on the recursive Definition 7.3.1. This also justifies the familiar recursive definitions of functions on the nonnegative integers.

7.3.1 Some Standard Recursive Functions on $\mathbb{N}$

The Factorial function. This function is often written "$n!$." You will see a lot of it in later cahpters. chapters Here we'll use the notation fac(n):

- $\text{fac}(0) ::= 1$.
- $\text{fac}(n + 1) ::= (n + 1) \cdot \text{fac}(n)$ for $n \geq 0$.

The Fibonacci numbers. Fibonacci numbers arose out of an effort 800 years ago to model population growth. They have a continuing fan club of people captivated by their extraordinary properties. The nth Fibonacci number, fib, can be defined recursively by:

$$\begin{aligned}
F(0) &::= 0,\\
F(1) &::= 1,\\
F(n) &::= F(n-1) + F(n-2) && \text{for } n \geq 2.
\end{aligned}$$

Here the recursive step starts at $n = 2$ with base cases for 0 and 1. This is needed since the recursion relies on two previous values.

What is $F(4)$? Well, $F(2) = F(1) + F(0) = 1$, $F(3) = F(2) + F(1) = 2$, so $F(4) = 3$. The sequence starts out $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

Sum-notation. Let "$S(n)$" abbreviate the expression "$\sum_{i=1}^{n} f(i)$." We can recursively define $S(n)$ with the rules

- $S(0) ::= 0$.
- $S(n+1) ::= f(n+1) + S(n)$ for $n \geq 0$.

7.3.2 Ill-formed Function Definitions

There are some other blunders to watch out for when defining functions recursively. The main problems come when recursive definitions don't follow the recursive definition of the underlying data type. Below are some function specifications that resemble good definitions of functions on the nonnegative integers, but they aren't.

$$f_1(n) ::= 2 + f_1(n-1). \tag{7.2}$$

This "definition" has no base case. If some function, f_1, satisfied (7.2), so would a function obtained by adding a constant to the value of f_1. So equation (7.2) does not uniquely define an f_1.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_2(n+1) & \text{otherwise.} \end{cases} \tag{7.3}$$

This "definition" has a base case, but still doesn't uniquely determine f_2. Any function that is 0 at 0 and constant everywhere else would satisfy the specification, so (7.3) also does not uniquely define anything.

In a typical programming language, evaluation of $f_2(1)$ would begin with a recursive call of $f_2(2)$, which would lead to a recursive call of $f_2(3)$, ... with recursive calls continuing without end. This "operational" approach interprets (7.3) as defining a *partial* function, f_2, that is undefined everywhere but 0.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by 2,} \\ 1, & \text{if } n \text{ is divisible by 3,} \\ 2, & \text{otherwise.} \end{cases} \tag{7.4}$$

This "definition" is inconsistent: it requires $f_3(6) = 0$ and $f_3(6) = 1$, so (7.4) doesn't define anything.

Mathematicians have been wondering about this function specification for a while:

$$f_4(n) ::= \begin{cases} 1, & \text{if } n \leq 1, \\ f_4(n/2) & \text{if } n > 1 \text{ is even,} \\ f_4(3n+1) & \text{if } n > 1 \text{ is odd.} \end{cases} \tag{7.5}$$

For example, $f_4(3) = 1$ because

$$f_4(3) ::= f_4(10) ::= f_4(5) ::= f_4(16) ::= f_4(8) ::= f_4(4) ::= f_4(2) ::= f_4(1) ::= 1.$$

The constant function equal to 1 will satisfy (7.5) (why?), but it's not known if another function does too. The problem is that the third case specifies $f_4(n)$ in terms of f_4 at arguments larger than n, and so cannot be justified by induction on $\mathbb{N}$. It's known that any f_4 satisfying (7.5) equals 1 for all n up to over a billion.

A final example is Ackermann's function, which is an extremely fast-growing function of two nonnegative arguments. Its inverse is correspondingly slow-growing —it grows slower than $\log n$, $\log\log n$, $\log\log\log n$, ..., but it does grow unboundly. This inverse actually comes up analyzing a useful, highly efficient procedure known as the *Union-Find algorithm.* This algorithm was conjectured to run in a number of steps that grew linearly in the size of its input, but turned out to be "linear" but with a slow growing coefficient nearly equal to the inverse Ackermann function. This means that pragmatically *Union-Find* is linear since the theoretically growing coefficient is less than 5 for any input that could conceivably come up.

Ackermann's function can be defined recursively as the function, A, given by the following rules:

$$A(m,n) = 2n, \qquad \text{if } m = 0 \text{ or } n \leq 1, \tag{7.6}$$
$$A(m,n) = A(m-1, A(m, n-1)), \qquad \text{otherwise.} \tag{7.7}$$

Now these rules are unusual because the definition of $A(m,n)$ involves an evaluation of A at arguments that may be a lot bigger than m and n. The definitions of f_2 above showed how definitions of function values at small argument values in terms of larger one can easily lead to nonterminating evaluations. The definition of Ackermann's function is actually ok, but proving this takes some ingenuity (see Problem 7.12).

7.4 Arithmetic Expressions

Expression evaluation is a key feature of programming languages, and recognition of expressions as a recursive data type is a key to understanding how they can be processed.

To illustrate this approach we'll work with a toy example: arithmetic expressions like $3x^2 + 2x + 1$ involving only one variable, "x." We'll refer to the data type of such expressions as Aexp. Here is its definition:

Definition 7.4.1.

- **Base cases:**
  - The variable, x, is in Aexp.
  - The arabic numeral, k, for any nonnegative integer, k, is in Aexp.
- **Constructor cases:** If $e, f \in$ Aexp, then
  - $[\,e + f\,] \in$ Aexp. The expression $[\,e + f\,]$ is called a *sum*. The Aexp's e and f are called the *components* of the sum; they're also called the *summands*.
  - $[\,e * f\,] \in$ Aexp. The expression $[\,e * f\,]$ is called a *product*. The Aexp's e and f are called the *components* of the product; they're also called the *multiplier* and *multiplicand*.
  - $-[\,e\,] \in$ Aexp. The expression $-[\,e\,]$ is called a *negative*.

Notice that Aexp's are fully bracketed, and exponents aren't allowed. So the Aexp version of the polynomial expression $3x^2+2x+1$ would officially be written as

$$[\,[\,3 * [\,x * x\,]\,] + [\,[\,2 * x\,] + 1\,]\,]. \tag{7.8}$$

These brackets and $*$'s clutter up examples, so we'll often use simpler expressions like "$3x^2+2x+1$" instead of (7.8). But it's important to recognize that $3x^2+2x+1$ is not an Aexp; it's an *abbreviation* for an Aexp.

7.4.1 Evaluation and Substitution with Aexp's

Evaluating Aexp's

Since the only variable in an Aexp is x, the value of an Aexp is determined by the value of x. For example, if the value of x is 3, then the value of $3x^2 + 2x + 1$ is obviously 34. In general, given any Aexp, e, and an integer value, n, for the variable, x, we can evaluate e to finds its value, $\text{eval}(e, n)$. It's easy, and useful, to specify this evaluation process with a recursive definition.

Definition 7.4.2. The *evaluation function*, eval(:, Aexp) $\times \mathbb{Z} \to \mathbb{Z}$, is defined recursively on expressions, $e \in$ Aexp, as follows. Let n be any integer.

- **Base cases:**

$$\text{eval}(x, n) ::= n, \qquad \text{(value of variable } x \text{ is } n.) \qquad (7.9)$$

$$\text{eval}(\mathtt{k}, n) ::= k, \qquad \text{(value of numeral } \mathtt{k} \text{ is } k, \text{ regardless of } x.) \qquad (7.10)$$

- **Constructor cases:**

$$\text{eval}([\, e_1 + e_2\,], n) ::= \text{eval}(e_1, n) + \text{eval}(e_2, n), \qquad (7.11)$$

$$\text{eval}([\, e_1 * e_2\,], n) ::= \text{eval}(e_1, n) \cdot \text{eval}(e_2, n), \qquad (7.12)$$

$$\text{eval}(-[\, e_1\,], n) ::= -\text{eval}(e_1, n). \qquad (7.13)$$

For example, here's how the recursive definition of eval(,w)ould arrive at the value of $3 + x^2$ when x is 2:

$$\begin{aligned}
\text{eval}([\,3 + [\,x * x\,]\,], 2) &= \text{eval}(3, 2) + \text{eval}([\,x * x\,], 2) && \text{(by Def 7.4.2.7.11)} \\
&= 3 + \text{eval}([\,x * x\,], 2) && \text{(by Def 7.4.2.7.10)} \\
&= 3 + (\text{eval}(x, 2) \cdot \text{eval}(x, 2)) && \text{(by Def 7.4.2.7.12)} \\
&= 3 + (2 \cdot 2) && \text{(by Def 7.4.2.7.9)} \\
&= 3 + 4 = 7.
\end{aligned}$$

Substituting into Aexp's

Substituting expressions for variables is a standard operation used by compilers and algebra systems. For example, the result of substituting the expression $3x$ for x in the expression $x(x-1)$ would be $3x(3x-1)$. We'll use the general notation subst(f, e) for the result of substituting an Aexp, f, for each of the x's in an Aexp, e. So as we just explained,

$$\text{subst}(3x, x(x-1)) = 3x(3x-1).$$

This substitution function has a simple recursive definition:

Definition 7.4.3. The *substitution function* from Aexp $\times$ Aexp to Aexp is defined recursively on expressions, $e \in$ Aexp, as follows. Let f be any Aexp.

- **Base cases:**

$$\text{subst}(f, x) ::= f, \qquad \text{(subbing } f \text{ for variable, } x \text{, just gives } f\text{)} \tag{7.14}$$
$$\text{subst}(f, \mathrm{k}) ::= \mathrm{k} \qquad (subbing into a numeral does nothing.) \tag{7.15}$$

- **Constructor cases:** ?later error

$$\text{subst}(f, [\, e_1 + e_2]) ::= [\ \text{subst}(f, e_1) + \text{subst}(f, e_2)] \tag{7.16}$$
$$\text{subst}(f, [\, e_1 * e_2])) ::= [\ \text{subst}(f, e_1) * \text{subst}(f, e_2)] \tag{7.17}$$
$$\text{subst}(f, -[\, e_1]) ::= -[\ \text{subst}(f, e_1)]\,. \tag{7.18}$$

Here's how the recursive definition of the substitution function would find the result of substituting $3x$ for x in the $x(x-1)$:

$$\begin{aligned}
&\text{subst}(3x, [\, x[\, x-1]\,]) \\
&= \text{subst}(3x, [\, x * [\, x + -[\, 1]\,]\,]) && \text{(unabbreviating)} \\
&= [\ \text{subst}(3x, x) * \text{subst}(3x, [\, x + -[\, 1[\,]\,]) && \text{(by Def 7.4.3 7.17)} \\
&= [\, 3x * \text{subst}(3x, [\, x + -[\, 1]\,]\,)] && \text{(by Def 7.4.3 7.14)} \\
&= (3x * [\ \text{subst}(3x, x) + \text{subst}(3x, -[\, 1])\,]\,] && \text{(by Def 7.4.3 7.16)} \\
&= [\, 3x * [\, 3x + -[\ \text{subst}(3x, 1)]\,]\,] && \text{(by Def 7.4.3 7.14 \& 7.18)} \\
&= [\, 3x * [\, 3x + -[\, 1]\,]\,] && \text{(by Def 7.4.3 7.15)} \\
&= 3x[\, 3x - 1] && \text{(abbreviation)}
\end{aligned}$$

Now suppose we have to find the value of $\text{subst}(3x, x(x-1))$ when $x = 2$. There are two approaches.

First, we could actually do the substitution above to get $3x(3x-1)$, and then we could evaluate $3x(3x-1)$ when $x = 2$, that is, we could recursively calculate $\text{eval}(3x(3x-1), 2)$ to get the final value 30. In programming jargon, this would be called evaluation using the *Substitution Model*. Because the formula $3x$ appears twice after substitution, it the multiplication $3 \cdot 2$ to computes its value gets performed twice.

The other approach is called evaluation using the *Environment Model*. Namely, to compute

$$\text{eval}(\text{subst}(3x, x(x-1)), 2) \tag{7.19}$$

we evaluate $3x$ when $x = 2$ using just 1 multiplication to get the value 6. Then we evaluate $x(x-1)$ when x has this value 6 to arrive at the value $6 \cdot 5 = 30$. So the Environment Model only computes the value of $3x$ once and so requires one fewer multiplication than the Substitution model to compute (7.19). But how do we know that these final values reached by these two approaches always agree? We can prove this easily by structural induction on the definitions of the two approaches. More precisely, what we want to prove is

Theorem 7.4.4. *For all expressions* $e, f \in$ *Aexp and* $n \in \mathbb{Z}$,

$$\text{eval}(\text{subst}(f, e), n) = \text{eval}(e, \text{eval}(f, n)). \tag{7.20}$$

Proof. The proof is by structural induction on e.[1]

Base cases:

- Case$[x]$

  The left hand side of equation (7.20) equals eval(f, n) by this base case in Definition 7.4.3 of the substitution function, and the right hand side also equals eval(f, n) by this base case in Definition 7.4.2 of eval(,.)

- Case[k].

  The left hand side of equation (7.20) equals k by this base case in Definitions 7.4.3 and 7.4.2 of the substitution and evaluation functions. Likewise, the right hand side equals k by two applications of this base case in the Definition 7.4.2 of eval(,.)

Constructor cases:

- Case$[[\, e_1 + e_2 \,]]$

  By the structural induction hypothesis (7.20), we may assume that for all $f \in$ Aexp and $n \in \mathbb{Z}$,

$$\text{eval}(\text{subst}(f, e_i), n) = \text{eval}(e_i, \text{eval}(f, n)) \tag{7.21}$$

  for $i = 1, 2$. We wish to prove that

$$\text{eval}(\text{subst}(f, [\, e_1 + e_2 \,]), n) = \text{eval}([\, e_1 + e_2 \,], \text{eval}(f, n)) \tag{7.22}$$

  But the left hand side of (7.22) equals

$$\text{eval}([\, \text{subst}(f, e_1) + \text{subst}(f, e_2) \,], n)$$

[1] This is an example of why it's useful to notify the reader what the induction variable is—in this case it isn't n.

by Definition 7.4.3.7.16 of substitution into a sum expression. But this equals

$$\text{eval}(\text{subst}(f, e_1), n) + \text{eval}(\text{subst}(f, e_2), n)$$

by Definition 7.4.2.(7.11) of eval(,f)or a sum expression. By induction hypothesis (7.21), this in turn equals

$$\text{eval}(e_1, \text{eval}(f, n)) + \text{eval}(e_2, \text{eval}((, f), n)).$$

Finally, this last expression equals the right hand side of (7.22) by Definition 7.4.2.(7.11) of eval(,f)or a sum expression. This proves (7.22) in this case.

- Case[[$e_1 * e_2$]] Similar.
- Case[$-$[e_1]] Even easier.

This covers all the constructor cases, and so completes the proof by structural induction.

■

7.5 Induction in Computer Science

Induction is a powerful and widely applicable proof technique, which is why we've devoted two entire chapters to it. Strong induction and its special case of ordinary induction are applicable to any kind of thing with nonnegative integer sizes –which is a awful lot of things, including all step-by-step computational processes.

Structural induction then goes beyond natural number counting by offering a simple, natural approach to proving things about recursive data types and recursive computation. This makes it a technique every computer scientist should embrace.

Problems for Section 7.1

Class Problems

Problem 7.1.
Prove that for all strings $r, s, t \in A^*$

$$(r \cdot s) \cdot t = r \cdot (s \cdot t).$$

Problem 7.2.
The *reversal* of a string is the string written backwards, for example, $\text{rev}(abcde) = edcba$.

(a) Give a simple recursive definition of $\text{rev}(s)$ based on the recursive definition 7.1.1 of $s \in A^*$ and using the concatenation operation 7.1.3.

(b) Prove that
$$\text{rev}(s \cdot t) = \text{rev}(t) \cdot \text{rev}(s),$$
for all strings $s, t \in A^*$.

Problem 7.3.
The Elementary 18.01 Functions (F18's) are the set of functions of one real variable defined recursively as follows:

Base cases:

- The identity function, $\text{id}(x) ::= x$ is an F18,
- any constant function is an F18,
- the sine function is an F18,

Constructor cases:
If f, g are F18's, then so are

1. $f + g$, fg, e^g (the constant e),
2. the inverse function $f^{(-1)}$,
3. the composition $f \circ g$.

(a) Prove that the function $1/x$ is an F18.

Warning: Don't confuse $1/x = x^{-1}$ with the inverse, $\text{id}^{(-1)}$ of the identity function $\text{id}(x)$. The inverse $\text{id}^{(-1)}$ is equal to id.

(b) Prove by Structural Induction on this definition that the Elementary 18.01 Functions are *closed under taking derivatives*. That is, show that if $f(x)$ is an F18, then so is $f' ::= df/dx$. (Just work out 2 or 3 of the most interesting constructor cases; you may skip the less interesting ones.)

Problem 7.4.
Here is a simple recursive definition of the set, E, of even integers:

Definition. **Base case**: $0 \in E$.
Constructor cases: If $n \in E$, then so are $n + 2$ and $-n$.

Provide similar simple recursive definitions of the following sets:

(a) The set $S ::= \{2^k 3^m 5^n \mid k, m, n \in \mathbb{N}\}$.

(b) The set $T ::= \{2^k 3^{2k+m} 5^{m+n} \mid k, m, n \in \mathbb{N}\}$.

(c) The set $L ::= \{(a, b) \in \mathbb{Z}^2 \mid 3 \mid (a - b)\}$.

Let L' be the set defined by the recursive definition you gave for L in the previous part. Now if you did it right, then $L' = L$, but maybe you made a mistake. So let's check that you got the definition right.

(d) Prove by structural induction on your definition of L' that

$$L' \subseteq L.$$

(e) Confirm that you got the definition right by proving that

$$L \subseteq L'.$$

(f) See if you can give an *unambiguous* recursive definition of L.

Problem 7.5.

Definition. The recursive data type, binary-2PTG, of *binary trees* with leaf labels, L, is defined recursively as follows:

- **Base case:** $\langle \texttt{leaf}, l \rangle \in$ binary-2PTG, for all labels $l \in L$.
- **Constructor case:** If $G_1, G_2 \in$ binary-2PTG, then

$$\langle \texttt{bintree}, G_1, G_2 \rangle \in \text{binary-2PTG}.$$

The *size*, $|G|$, of $G \in$ binary-2PTG is defined recursively on this definition by:

- **Base case:**

$$|\langle \texttt{leaf}, l \rangle| ::= 1, \quad \text{for all } l \in L.$$

- **Constructor case:**

$$|\langle \texttt{bintree}, G_1, G_2 \rangle| ::= |G_1| + |G_2| + 1.$$

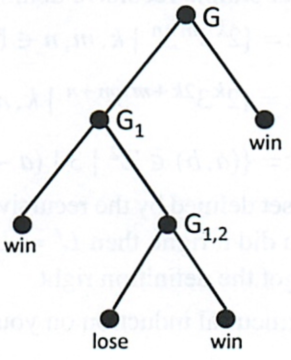


Figure 7.1 A picture of a binary tree w.

For example, for the size of the binary-2PTG, G, pictured in Figure 7.1, is 7.

(a) Write out (using angle brackets and labels `bintree`, `leaf`, etc.) the binary-2PTG, G, pictured in Figure 7.1.

The value of flatten(G) for $G \in$ binary-2PTG is the sequence of labels in L of the leaves of G. For example, for the binary-2PTG, G, pictured in Figure 7.1,

$$\text{flatten}(G) = (\texttt{win}, \texttt{lose}, \texttt{win}, \texttt{win}).$$

(b) Give a recursive definition of flatten. (You may use the operation of *concatenation* (append) of two sequences.)

(c) Prove by structural induction on the definitions of flatten and size that

$$2 \cdot \text{length}(\text{flatten}(G)) = |G| + 1. \tag{7.23}$$

Homework Problems

Problem 7.6.

Definition. Define the number, $\#_c(s)$, of occurrences of the character $c \in A$ in the string s recursively on the definition of $s \in A^*$:

base case: $\#_c(\lambda) ::= 0$.
constructor case:

$$\#_c(\langle a, s\rangle) ::= \begin{cases} \#_c(s) & \text{if } a \neq c, \\ 1 + \#_c(s) & \text{if } a = c. \end{cases}$$

Prove by structural induction that for all $s, t \in A^*$ and $c \in A$

$$\#_c(scdott) = \#_c(s) + \#_c(t).$$

Problem 7.7.
Fractals are example of a mathematical object that can be defined recursively. In this problem, we consider the Koch snowflake. Any Koch snowflake can be constructed by the following recursive definition.

- **base case:** An equilateral triangle with a positive integer side length is a Koch snowflake.
- **constructor case:** Let K be a Koch snowflake, and let l be a line segment on the snowflake. Remove the middle third of l, and replace it with two line segments of the same length as is done below:

  MISSING GRAPHIC

  The resulting figure is also a Koch snowflake.

Prove by structural induction that the area inside any Koch snowflake is of the form $q\sqrt{3}$, where q is a rational number.

Problems for Section 7.2

Practice Problems

Problem 7.8.

Definition. Consider a new recursive definition, MB_0, of the same set of "matching" brackets strings as MB (definition of MB is provided in the Appendix):

- **Base case:** $\lambda \in \text{MB}_0$.
- **Constructor cases:**

  (i) If s is in MB_0, then $[\, s\,]$ is in MB_0.

  (ii) If $s, t \in \text{MB}_0$, $s \neq \lambda$, and $t \neq \lambda$, then st is in MB_0.

(a) Suppose structural induction was being used to prove that $\text{MB}_0 \subseteq \text{MB}$. Circle the one predicate below that would fit the format for a structural induction hypothesis in such a proof.

- $P_0(n) ::= |s| \leq n$ IMPLIES $s \in \text{MB}$.
- $P_1(n) ::= |s| \leq n$ IMPLIES $s \in \text{MB}_0$.
- $P_2(s) ::= s \in \text{MB}$.
- $P_3(s) ::= s \in \text{MB}_0$.
- $P_4(s) ::= (s \in \text{MB}$ IMPLIES $s \in \text{MB}_0)$.

(b) The recursive definition MB_0 is *ambiguous*. Verify this by giving two different derivations for the string "[ ][ ][ ]" according to MB_0.

Class Problems

Problem 7.9.
Let p be the string [ ]. A string of brackets is said to be *erasable* iff it can be reduced to the empty string by repeatedly erasing occurrences of p. For example, here's how to erase the string [[[ ]] [ ]] [ ] :

$$[\,[\,[\,]\,]\,[\,]\,]\,[\,] \rightarrow [\,[\,]\,] \rightarrow [\,] \rightarrow \lambda.$$

On the other hand the string [ ]] [[[[[ ]] is not erasable because when we try to erase, we get stuck:

$$[\,]\,]\,[\,[\,[\,[\,[\,]\,] \rightarrow]\,[\,[\,[\,[\,] \rightarrow]\,[\,[\,[\not\rightarrow$$

Let Erasable be the set of erasable strings of brackets. Let RecMatch be the recursive data type of strings of *matched* brackets given in Definition 7.2.1.

(a) Use structural induction to prove that

$$\text{RecMatch} \subseteq \text{Erasable}.$$

(b) Supply the missing parts of the following proof that

$$\text{Erasable} \subseteq \text{RecMatch}.$$

Proof. We prove by strong induction that every length-n string in Erasable is also in RecMatch. The induction hypothesis is

$$P(n) ::= \forall x \in \text{Erasable}.\, |x| = n \text{ IMPLIES } x \in \text{RecMatch}.$$

Base case:

What is the base case? Prove that P is true in this case.

Inductive step: To prove $P(n+1)$, suppose $|x| = n + 1$ and $x \in$ Erasable. We need to show that $x \in$ RecMatch.

Let's say that a string y is an *erase* of a string z iff y is the result of erasing a *single* occurrence of p in z.

Since $x \in$ Erasable and has positive length, there must be an erase, $y \in$ Erasable, of x. So $|y| = n - 1 \geq 0$, and since $y \in$ Erasable, we may assume by induction hypothesis that $y \in$ RecMatch.

Now we argue by cases:

Case (y is the empty string):

Prove that $x \in$ RecMatch in this case.

Case ($y = [\, s \,]\, t$ for some strings $s, t \in$ RecMatch): Now we argue by subcases.

- **Subcase** (x is of the form $[\, s' \,]\, t$ where s is an erase of s'):
  Since $s \in$ RecMatch, it is erasable by part (b), which implies that $s' \in$ Erasable. But $|s'| < |x|$, so by induction hypothesis, we may assume that $s' \in$ RecMatch. This shows that x is the result of the constructor step of RecMatch, and therefore $x \in$ RecMatch.
- **Subcase** (x is of the form $[\, s \,]\, t'$ where t is an erase of t'):
  Prove that $x \in$ RecMatch in this subcase.

- **Subcase($x = p[\,s\,]\,t$):**
  Prove that $x \in$ RecMatch in this subcase.

The proofs of the remaining subcases are just like this last one. **List these remaining subcases.**

This completes the proof by strong induction on n, so we conclude that $P(n)$ holds for all $n \in \mathbb{N}$. Therefore $x \in$ RecMatch for every string $x \in$ Erasable. That is, Erasable $\subseteq$ RecMatch. Combined with part (a), we conclude that

$$\text{Erasable} = \text{RecMatch}.$$

■

Problem 7.10.
The set of strings, RecMatch, is recursively defined as follows:

- **Base case:** $\lambda \in$ RecMatch.
- **Constructor case:** If $s, t \in$ RecMatch, then

$$[\,s\,]\,t \in \text{RecMatch}.$$

The set of strings, M, is recursively defined as follows:

- **Base case:** $\lambda \in M$,
- **Constructor cases:** if $s, t \in M$, then the strings $[\,s\,]$ and $s \cdot t$ are also in M.

Prove by structural induction that

(a) $M \subseteq$ RecMatch

(b) RecMatch $\subseteq M$

Problem 7.11.

Definition 7.5.1. The set, RecMatch, of strings of matching brackets, is defined recursively as follows:

- **Base case:** $\lambda \in$ RecMatch.

- **Constructor case:** If $s, t \in \text{RecMatch}$, then

$$[\, s\,]\, t \in \text{RecMatch}.$$

One precise way to determine if a string is matched is to start with 0 and read the string from left to right, adding 1 to the count for each left bracket and subtracting 1 from the count for each right bracket. For example, here are the counts for two sample strings:

$$\begin{array}{ccccccccccccccccccccccccc}
 & [& &] & &] & & [& & [& & [& & [& & [& &] & &] & &] & &] & \\
0 & & 1 & & 0 & & -1 & & 0 & & 1 & & 2 & & 3 & & 4 & & 3 & & 2 & & 1 & & 0
\end{array}$$

$$\begin{array}{ccccccccccccccccccccc}
 & [& & [& & [& &] & &] & & [& &] & &] & & [& &] & \\
0 & & 1 & & 2 & & 3 & & 2 & & 1 & & 2 & & 1 & & 0 & & 1 & & 0
\end{array}$$

A string has a *good count* if its running count never goes negative and ends with 0. So the second string above has a good count, but the first one does not because its count went negative at the third step.

Definition 7.5.2. Let

$$\text{GoodCount} ::= \{s \in \{\,]\,,[\,\}^* \mid s \text{ has a good count}\}.$$

The matched strings can now be characterized precisely as this set of strings with good counts.

(a) Prove that GoodCount contains RecMatch by structural induction on the definition of RecMatch.

(b) Conversely, prove that RecMatch contains GoodCount.

Problems for Section 7.3

Homework Problems

Problem 7.12.
Ackermann's function, $A : \mathbb{N}^2 \to \mathbb{N}$, is defined recursively by the following rules:

$$A(m, n) ::= 2n, \qquad \text{if } m = 0 \text{ or } n \le 1 \tag{7.24}$$
$$A(m, n) ::= A(m-1, A(m, n-1)), \qquad \text{otherwise.} \tag{7.25}$$

Prove that if $B : \mathbb{N}^2 \to \mathbb{N}$ is a partial function that satisfies this same definition, then B is total and $B = A$.

Problems for Section 7.4

Practice Problems

Problem 7.13. (a) Write out the evaluation of

$$\text{eval}(\text{subst}(3x, x(x-1)), 2)$$

according to the Environment Model and the Substitution Model, indicating where the rule for each case of the recursive definitions of eval(,) and [:=] or substitution is first used. Compare the number of arithmetic operations and variable lookups.

(b) Describe an example along the lines of part (a) where the Environment Model would perform 6 fewer multiplications than the Substitution model. You need *not* carry out the evaluations.

(c) Describe an example along the lines of part (a) where the Substitution Model would perform 6 fewer multiplications than the Environment model. You need *not* carry out the evaluations.

Homework Problems

Problem 7.14. (a) Give a recursive definition of a function erase(e) that erases all the symbols in the Aexp$e$ but the brackets. For example

$$\text{erase}([\,[\,3 * [\,x * x]\,]\,+[\,[\,2 * x]\,+1]\,]) = [\,[\,[\,]\,]\,[\,[\,2 * x]\,+1]\,].$$

(b) Prove that erase(e) $\in$ RecMatch for all $e \in$ Aexp.

(c) Give an example of a small string $s \in$ RecMatch such that $[\,s] \neq$ erase(e) for any $e \in aexp$.

2/25

8 Number Theory

Number theory is the study of the integers. *Why* anyone would want to study the integers is not immediately obvious. First of all, what's to know? There's 0, there's 1, 2, 3, and so on, and, oh yeah, -1, -2, ... Which one don't you understand? Second, what practical value is there in it?

The mathematician G. H. Hardy expressed pleasure in its impracticality when he wrote:

> [Number theorists] may be justified in rejoicing that there is one science, at any rate, and that their own, whose very remoteness from ordinary human activities should keep it gentle and clean.

g/((

~~Abs~~ abstract/ pure math

Hardy was specially concerned that number theory not be used in warfare; he was a pacifist. You may applaud his sentiments, but he got it wrong: Number Theory underlies modern cryptography, which is what makes secure online communication possible. Secure communication is of course crucial in war—which may leave poor Hardy spinning in his grave. It's also central to online commerce. Every time you buy a book from Amazon, check your grades on WebSIS, or use a PayPal account, you are relying on number theoretic algorithms.

Number theory also provides an excellent environment for us to practice and apply the proof techniques that we developed in previous Chapters. We'll work out properties of greatest common divisors (gcd's) and use them to prove that integers factor uniquely into primes. Then we'll introduce modular arithmetic and work out enough of its properties to explain the RSA public key crypto-system.

Since we'll be focusing on properties of the integers, we'll adopt the default convention in this chapter that *variables range over the set, $\mathbb{Z}$, of integers*.

8.1 Divisibility

The nature of number theory emerges as soon as we consider the *divides* relation, where

remember the notation!

Definition 8.1.1.

$$a \mid b ::= \quad [ak = b \text{ for some } k].$$

(that is an integer — very important for that def'n

The divides relation comes up so frequently that multiple synonyms for it are used all the time. The following phrases all say the same thing:

- $a \mid b$,
- a divides b,
- a is a *divisor* of b,
- a is a *factor* of b,
- b is *divisible* by a,
- b is a *multiple* of a.

diff ways of saying same thing

Some immediate consequences of Definition 8.1.1 are that $n \mid 0, n \mid n$, and $1 \mid n$ for all $n \neq 0$.

Dividing seems simple enough, but let's play with this definition. The Pythagoreans, an ancient sect of mathematical mystics, said that a number is *perfect* if it equals the sum of its positive integral divisors, excluding itself. For example, $6 = 1 + 2 + 3$ and $28 = 1 + 2 + 4 + 7 + 14$ are perfect numbers. On the other hand, 10 is not perfect because $1 + 2 + 5 = 8$, and 12 is not perfect because $1 + 2 + 3 + 4 + 6 = 16$. Euclid characterized all the *even* perfect numbers around 300 BC. But is there an *odd* perfect number? More than two thousand years later, we still don't know! All numbers up to about 10^{300} have been ruled out, but no one has proved that there isn't an odd perfect number waiting just over the horizon.

I would have written a proof on that -ha

So a half-page into number theory, we've strayed past the outer limits of human knowledge! This is pretty typical; number theory is full of questions that are easy to pose, but incredibly difficult to answer.[1]

Some of the greatest insights and mysteries in number theory concern properties of *prime* numbers:

Definition 8.1.2. A *prime* is a number greater than 1 that is divisible only by itself and 1.

Several such problems are included in the box on the following page. Interestingly, we'll see that computer scientists have found ways to turn some of these difficulties to their advantage.

8.1.1 Facts about Divisibility

The following lemma collects some basic facts about divisibility.

Lemma 8.1.3.

[1] *Don't Panic*—we're going to stick to some relatively benign parts of number theory. These super-hard unsolved problems rarely get put on problem sets.

Famous Conjectures in Number Theory

Goldbach Conjecture Every even integer greater than two is equal to the sum of two primes. For example, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, etc. The conjecture holds for all numbers up to 10^{16}. In 1939 Schnirelman proved that every even number can be written as the sum of not more than 300,000 primes, which was a start. Today, we know that every even number is the sum of at most 6 primes. every?

Twin Prime Conjecture There are infinitely many primes p such that $p + 2$ is also a prime. In 1966 Chen showed that there are infinitely many primes p such that $p + 2$ is the product of at most two primes. So the conjecture is known to be *almost* true! 6?

Primality Testing There is an efficient way to determine whether a number is prime. A naive search for factors of an integer n takes a number of steps proportional to $\sqrt{n}$, which is exponential in the *size* of n in decimal or binary notation. All known procedures for prime checking blew up like this on various inputs. Finally in 2002, an amazingly simple, new method was discovered by Agrawal, Kayal, and Saxena, which showed that prime testing only required a polynomial number of steps. Their paper began with a quote from Gauss emphasizing the importance and antiquity of the problem even in his time—two centuries ago. So prime testing is definitely not in the category of infeasible problems requiring an exponentially growing number of steps in bad cases.

Factoring Given the product of two large primes $n = pq$, there is no efficient way to recover the primes p and q. The best known algorithm is the "number field sieve", which runs in time proportional to:

$$e^{1.9(\ln n)^{1/3}(\ln \ln n)^{2/3}}$$

?? how did they find that?

This is infeasible when n has 300 digits or more.

Fermat's Last Theorem There are no positive integers x, y, and z such that

$$x^n + y^n = z^n$$

for some integer $n > 2$. In a book he was reading around 1630, Fermat claimed to have a proof but not enough space in the margin to write it down. Wiles finally gave a proof of the theorem in 1994, after seven years of working in secrecy and isolation in his attic. His proof did not fit in any margin.

1. *If $a \mid b$ and $b \mid c$, then $a \mid c$.*
2. *If $a \mid b$ and $a \mid c$, then $a \mid sb + tc$ for all s and t.*
3. *For all $c \neq 0$, $a \mid b$ if and only if $ca \mid cb$.*

Proof. These facts all follow directly from Definition 8.1.1, and we'll just prove part 2 for practice:

Given that $a \mid b$, there is some $k_1 \in \mathbb{Z}$ such that $ak_1 = b$. Likewise, $ak_2 = c$, so

$$sb + tc = s(k_1a) + t(k_2a) = (sk_1 + tk_2)a.$$

Therefore $sb + tc = k_3a$ where $k_3 ::= (sk_1 + tk_2)$, which means that

$$a \mid sb + tc.$$

■

A number of the form $sb + tc$ is called an *integer linear combination* of b and c, or a plain *linear combination*, since in this chapter we're only talking integers. So Lemma 8.1.3.2 can be rephrased as

If a divides b and c, then a divides every linear combination of b and c.

We'll be making good use of linear combinations, so let's get the general definition on record:

Definition 8.1.4. An integer n is a *linear combination* of numbers $b_0, \ldots, b_n$ iff

$$n = s_0b_0 + s_1b_1 + \cdots + s_nb_n$$

for some integers $s_0, \ldots, s_n$.

8.1.2 When Divisibility Goes Bad

mod

As you learned in elementary school, if one number does *not* evenly divide another, you get a "quotient" and a "remainder" left over. More precisely:

Theorem 8.1.5. *[Division Theorem][2] Let n and d be integers such that $d > 0$. Then there exists a unique pair of integers q and r, such that*

$$n = q \cdot d + \overset{\text{remainder}}{r} \text{ AND } 0 \leq r < d \leftarrow \text{divisor} \tag{8.1}$$

[2]This theorem is often called the "Division Algorithm," even though it is not what we would call an algorithm. We will take this familiar result for granted without proof.

The number q is called the *quotient* and the number r is called the *remainder* of n divided by d. We use the notation qcnt(n, d) for the quotient and rem(n, d) for the remainder.

$\frac{n}{d}$

For example, qcnt$(2716, 10) = 271$ and rem$(2716, 10) = 6$, since $2716 = 271 \cdot 10 + 6$. Similarly, rem$(-11, 7) = 3$, since $-11 = (-2) \cdot 7 + 3$. There is a remainder operator built into many programming languages. For example, "32 % 5" will be familiar as remainder notation to programmers in in Java, C, and C++; it evaluates to qcnt$(32, 5) = 2$ in all three languages. On the other hand, these languages treat quotients involving negative numbers idiosyncratically, so if you program in one those languages, remember to stick to the definition according to the Division Theorem 8.1.5.

Remainder format? mod -right?

The remainder on division by n is a number in interval from 0 to $n - 1$. Such intervals come up so often that it is useful to have a simple notation for them.

? what is this ?

$$\begin{aligned}
(k, n) &::= \{i \mid k < i < n\} \\
[k, n) &::= (k, n) \cup \{k\} \\
(k, n] &::= (k, n) \cup \{n\} \\
[k, n] &::= (k, n) \cup \{k, n\}
\end{aligned}$$

8.1.3 Die Hard

Die Hard 3 is just a B-grade action movie, but we think it has an inner message: everyone should learn at least a little number theory. In Section 6.2.4, we formalized a state machine for the Die Hard jug-filling problem using 3 and 5 gallon jugs, and also with 3 and 9 gallon jugs, and came to different conclusions about bomb explosions. What's going on in general? For example, how about getting 4 gallons from 12- and 18-gallon jugs, getting 32 gallons with 899- and 1147-gallon jugs, or getting 3 gallons into a jug using just 21- and 26-gallon jugs?

It would be nice if we could solve all these silly water jug questions at once. This is where number theory comes in handy.

Finding an Invariant Property

Suppose that we have water jugs with capacities a and b with $b \geq a$. Let's carry out some sample operations of the state machine and see what happens, assuming

$b \geq a$

the b-jug is big enough:

$$
\begin{aligned}
(0,0) &\to (a,0) && \text{fill first jug}\\
&\to (0,a) && \text{pour first into second}\\
&\to (a,a) && \text{fill first jug}\\
&\to (2a-b,b) && \text{pour first into second (assuming } 2a \geq b)\\
&\to (2a-b,0) && \text{empty second jug}\\
&\to (0,2a-b) && \text{pour first into second}\\
&\to (a,2a-b) && \text{fill first}\\
&\to (3a-2b,b) && \text{pour first into second (assuming } 3a \geq 2b)
\end{aligned}
$$

fill to top

remainder

What leaps out is that at every step, the amount of water in each jug is of a linear combination of a and b. This is easy to prove by induction on the number of transitions:

Lemma 8.1.6 (Water Jugs). *In the* Die Hard *state machine of Section 6.2.4 with jug of sizes a and b, the amount of water in each jug is always a linear combination of a and b.*

Proof. The induction hypothesis, $P(n)$, is the proposition that after n transitions, the amount of water in each jug is a linear combination of a and b.

Base case: ($n = 0$). $P(0)$ is true, because both jugs are initially empty, and $0 \cdot a + 0 \cdot b = 0$.

Inductive step. Suppose the machine is in state (x, y) after n steps, that is, the little jug contains x gallons and the big one contains y gallons. There are two cases:

- If we fill a jug from the fountain or empty a jug into the fountain, then that jug is empty or full. The amount in the other jug remains a linear combination of a and b. So $P(n + 1)$ holds.
- Otherwise, we pour water from one jug to another until one is empty or the other is full. By our assumption, the amount x and y in each jug is a linear combination of a and b before we begin pouring, After pouring, one jug is either empty (contains 0 gallons) or full (contains a or b gallons). Thus, the other jug contains either $x + y$ gallons, $x + y - a$, or $x + y - b$ gallons, all of which are linear combinations of a and b since x and y are. So $P(n + 1)$ holds in this case as well.

be able to do this!

Since $P(n + 1)$ holds in any case, this proves the inductive step, completing the proof by induction. ■

So we have established that the jug problem has an invariant property, namely that the amount of water in every jug is always a linear combination of the capacities of the jugs. Lemma 8.1.6 has an important corollary:

Corollary. *Getting 4 gallons from 12- and 18-gallon jugs, and likewise getting 32 gallons from 899- and 1147-gallon jugs,*

Bruce dies!

Smart!

Proof. By the Water Jugs Lemma 8.1.6, with 12- and 18-gallon jugs, the amount in any jug is a linear combination of 12 and 18. This is always a multiple of 6 by Lemma 8.1.3.2, so Bruce can’t get 4 gallons. Likewise, the amount in any jug using 899- and 1147-gallon jugs is a multiple of 31, so he can’t get 32 either. ■

But the Water Jugs Lemma isn’t very satisfying. One problem is that it leaves the question of getting 3 gallons into a jug using just 21- and 26-gallon jugs unresolved, since the only positive factor of both 21 and 26 is 1, and of course 1 divides 3. A bigger problem is that we’ve just managed to recast a pretty understandable question about water jugs into a complicated question about linear combinations. This might not seem like a lot of progress. Fortunately, linear combinations are closely related to something more familiar, namely greatest common divisors, and these will help us solve the general water jug problem. ?

8.2 The Greatest Common Divisor

A *common divisor* of a and b is a number that divides them both. The *greatest common divisor (gcd)* of a and b is written $\text{gcd}(a, b)$. For example, $\text{gcd}(18, 24) = 6$. The gcd turns out to be a very valuable piece of information about the relationship between a and b and for reasoning about integers in general. So we’ll be making lots of arguments about gcd’s in what follows.

8.2.1 Euclid’s Algorithm

The first thing to figure out is how to find gcd’s. A good way called *Euclid’s Algorithm* has been known for several thousand years. It is based on the following elementary observation.

Lemma 8.2.1.

$$\text{gcd}(a, b) = \text{gcd}(b, \text{rem}(a, b)).$$

rem(18,24)

gcd(24,18) that doesn't help!

Proof. By the Division Theorem 8.1.5,

$$a = qb + r \tag{8.2}$$

defined other way!

where $r = \text{rem}(a, b)$. So a is a linear combination of b and r, which implies that any divisor of b and r is a divisor of a by Lemma 8.1.3.2. Likewise, r is a linear combination, $a - qb$, of a and b, so any divisor of a and b is a divisor of r. This means that a and b have the same common divisors as b and r, and so they have the same *greatest* common divisor. ∎

Lemma 8.2.1 is useful for quickly computing the greatest common divisor of two numbers. For example, we could compute the greatest common divisor of 1147 and 899 by repeatedly it:

$$\begin{aligned}
\gcd(1147, 899) &= \gcd(899, \underbrace{\text{rem}(1147, 899)}_{=248}) \\
&= \gcd(248, \underbrace{\text{rem}(899, 248)}_{=155}) \\
&= \gcd(155, \underbrace{\text{rem}(248, 155)}_{=93}) \\
&= \gcd(93, \underbrace{\text{rem}(155, 93)}_{=62}) \\
&= \gcd(62, \underbrace{\text{rem}(93, 62)}_{=31}) \\
&= \gcd(31, \underbrace{\text{rem}(62, 31)}_{=0}) \\
&= \gcd(31, 0) \\
&= 31
\end{aligned}$$

oh here recursive

The last equation might look wrong, but 31 is a divisor of both 31 and 0 since every integer divides 0. This calculation that $\gcd(1147, 899) = 31$ was how we figured out that with water jugs of sizes 1147 and 899, Bruce dies trying to get 32 gallons.

Euclid's algorithm can easily be formalized as a state machine. The set of states is $\mathbb{N}^2$ and there is one transition rule:

$$(x, y) \longrightarrow (y, \text{rem}(x, y)), \tag{8.3}$$

when does he die again!

for $y > 0$. So by Lemma 8.2.1, the gcd stays the same from one state to the next, which means that started in state (a, b), the predicate $P(x, y)$,

$$\gcd(x, y) = \gcd(a, b),$$

“mcs” — 2011/2/27 — 0:32 — page 189 — #193

is an invariant. By the Invariant Principle, when $y = 0$ the value of x is the gcd because

$$x = \gcd(x, 0) = \gcd(a, b).$$ duh

What’s more, y does get to be 0 pretty fast: it’s easy to check that $y \leq x$ is another invariant, and since x gets divided by y at each step, it gets smaller by more than a factor of 2 until $y \leq 1$, after which the machine terminates in at most two more transitions. It follows that Euclid’s algorithm terminates after at most $2 + \log a$ transtions.

Correction $1+2\log a$

how do you know that?

But applyng Euclid’s algorithm to 26 and 21 gives

$$\gcd(26, 21) = \gcd(21, 5) = \gcd(5, 1) = 1,$$

which is why we left the 21- and 26-gallon jug problem unresolved. To resolve the matter, we will need more number theory.

8.2.2 The Pulverizer

We will get a lot of mileage out of the following key fact:

Theorem 8.2.2. *The greatest common divisor of a and b is a linear combination of a and b. That is,*

$$\gcd(a, b) = sa + tb,$$

for some integers s and t.

with one negative

this is the die hard problem

We already know from Lemma 8.1.3.2 that every linear combination of a and b is divisible by any common factor of a and b, so it is certainly divisible by the greatest of these common divisors. Since any constant multiple of a linear combination is also a linear combination, Theorem 8.2.2 implies that any multiple of the gcd is a linear combination. So we have the immediate corollary:

Corollary 8.2.3. *An integer is a linear combination of a and b iff it is a multiple of* $\gcd(a, b)$.

We’ll prove Theorem 8.2.2 directly by explaining how to find s and t. This job is tackled by a mathematical tool that dates to sixth-century India, where it was called *kuttak*, which means “The Pulverizer”. Today, the Pulverizer is more commonly known as “the extended Euclidean GCD algorithm”, because it is so close to Euclid’s Algorithm.

For example, following Euclid's Algorithm, we can compute the GCD of 259 and 70 as follows:

$$\begin{aligned}
\gcd(259,70) &= \gcd(70,49) && \text{since } \text{rem}(259,70)=49\\
&= \gcd(49,21) && \text{since } \text{rem}(70,49)=21\\
&= \gcd(21,7) && \text{since } \text{rem}(49,21)=7\\
&= \gcd(7,0) && \text{since } \text{rem}(21,7)=0\\
&= 7.
\end{aligned}$$

The Pulverizer goes through the same steps, but requires some extra bookkeeping along the way: as we compute $\gcd(a,b)$, we keep track of how to write each of the remainders (49, 21, and 7, in the example) as a linear combination of a and b. This is worthwhile, because our objective is to write the last nonzero remainder, which is the GCD, as such a linear combination. For our example, here is this extra bookkeeping:

x	y	$(\text{rem}(x,y))$	$=$	$x - q \cdot y$
259	70	49	$=$	$259 - 3 \cdot 70$
70	49	21	$=$	$70 - 1 \cdot 49$
			$=$	$70 - 1 \cdot (259 - 3 \cdot 70)$
			$=$	$-1 \cdot 259 + 4 \cdot 70$
49	21	7	$=$	$49 - 2 \cdot 21$
			$=$	$(259 - 3 \cdot 70) - 2 \cdot (-1 \cdot 259 + 4 \cdot 70)$
			$=$	$\boxed{3 \cdot 259 - 11 \cdot 70}$
21	7	0		

What are we doing here?

keep plugging in for

We began by initializing two variables, $x = a$ and $y = b$. In the first two columns above, we carried out Euclid's algorithm. At each step, we computed $\text{rem}(x,y)$, which can be written in the form $x - q \cdot y$. (Remember that the Division Algorithm says $x = q \cdot y + r$, where r is the remainder. We get $r = x - q \cdot y$ by rearranging terms.) Then we replaced x and y in this equation with equivalent linear combinations of a and b, which we already had computed. After simplifying, we were left with a linear combination of a and b that was equal to the remainder as desired. The final solution is boxed.

duh! be able to see!

This should make it pretty clear how and why the Pulverizer works. Anyone who has doubts can work out Problem 8.8, where the Pulverizer is formalized as a state machine and then verified using an invariant that is an extension of the one used for Euclid's algorithm.

Since the Pulverizer requires only a little more computation than Euclid's algorithm, you can "pulverize" very large numbers very quickly by using this algorithm.

As we will soon see, its speed makes the Pulverizer a very useful tool in the field of cryptography.

Now we can restate the Water Jugs Lemma 8.1.6 in terms of the greatest common divisor:

Corollary 8.2.4. *Suppose that we have water jugs with capacities a and b. Then the amount of water in each jug is always a multiple of* gcd(a, b).

For example, there is no way to form 4 gallons using 3- and 6-gallon jugs, because 4 is not a multiple of gcd$(3, 6) = 3$.

8.2.3 One Solution for All Water Jug Problems

Corollary 8.2.3 says that 3 can be written as a linear combination of 21 and 26, since 3 is a multiple of gcd$(21, 26) = 1$. So the Pulverizer will give us integers s and t such that

$$3 = s \cdot 21 + t \cdot 26 \tag{8.4}$$

Now the coefficient s could be either positive or negative. However, we can readily transform this linear combination into an equivalent linear combination

$$3 = s' \cdot 21 + t' \cdot 26 \tag{8.5}$$

where the coefficient s' is positive. The trick is to notice that if in equation (8.4) we increase s by 26 and decrease t by 21, then the value of the expression $s \cdot 21 + t \cdot 26$ is unchanged overall. Thus, by repeatedly increasing the value of s (by 26 at a time) and decreasing the value of t (by 21 at a time), we get a linear combination $s' \cdot 21 + t' \cdot 26 = 3$ where the coefficient s' is positive. Notice that then t' must be negative; otherwise, this expression would be much greater than 3.

Now we can form 3 gallons using jugs with capacities 21 and 26: We simply repeat the following steps s' times:

1. Fill the 21-gallon jug.

2. Pour all the water in the 21-gallon jug into the 26-gallon jug. If at any time the 26-gallon jug becomes full, empty it out, and continue pouring the 21-gallon jug into the 26-gallon jug.

At the end of this process, we must have have emptied the 26-gallon jug exactly $|t'|$ times. Here's why: we've taken $s' \cdot 21$ gallons of water from the fountain, and we've poured out some multiple of 26 gallons. If we emptied fewer than $|t'|$ times, then by (8.5), the big jug would be left with at least $3 + 26$ gallons, which is more than it can hold; if we emptied it more times, the big jug would be left containing at most $3 - 26$ gallons, which is nonsense. But once we have emptied the 26-gallon jug exactly $|t'|$ times, equation (8.5) implies that there are exactly 3 gallons left.

Remarkably, we don’t even need to know the coefficients s' and t' in order to use this strategy! Instead of repeating the outer loop s' times, we could just repeat *until we obtain 3 gallons*, since that must happen eventually. Of course, we have to keep track of the amounts in the two jugs so we know when we’re done. Here’s the solution that approach gives:

$$(0,0) \xrightarrow{\text{fill 21}}$$

$$(21,0) \xrightarrow{\text{pour 21 into 26}} (0,21)$$

$$\begin{array}{lllllllll}
\xrightarrow{\text{fill 21}} & (21,21) & \xrightarrow{\text{pour 21 to 26}} & (16,26) & \xrightarrow{\text{empty 26}} & (16,0) & \xrightarrow{\text{pour 21 to 26}} & (0,16) \\
\xrightarrow{\text{fill 21}} & (21,16) & \xrightarrow{\text{pour 21 to 26}} & (11,26) & \xrightarrow{\text{empty 26}} & (11,0) & \xrightarrow{\text{pour 21 to 26}} & (0,11) \\
\xrightarrow{\text{fill 21}} & (21,11) & \xrightarrow{\text{pour 21 to 26}} & (6,26) & \xrightarrow{\text{empty 26}} & (6,0) & \xrightarrow{\text{pour 21 to 26}} & (0,6) \\
\xrightarrow{\text{fill 21}} & (21,6) & \xrightarrow{\text{pour 21 to 26}} & (1,26) & \xrightarrow{\text{empty 26}} & (1,0) & \xrightarrow{\text{pour 21 to 26}} & (0,1) \\
\xrightarrow{\text{fill 21}} & (21,1) & \xrightarrow{\text{pour 21 to 26}} & (0,22) & & & & \\
\xrightarrow{\text{fill 21}} & (21,22) & \xrightarrow{\text{pour 21 to 26}} & (17,26) & \xrightarrow{\text{empty 26}} & (17,0) & \xrightarrow{\text{pour 21 to 26}} & (0,17) \\
\xrightarrow{\text{fill 21}} & (21,17) & \xrightarrow{\text{pour 21 to 26}} & (12,26) & \xrightarrow{\text{empty 26}} & (12,0) & \xrightarrow{\text{pour 21 to 26}} & (0,12) \\
\xrightarrow{\text{fill 21}} & (21,12) & \xrightarrow{\text{pour 21 to 26}} & (7,26) & \xrightarrow{\text{empty 26}} & (7,0) & \xrightarrow{\text{pour 21 to 26}} & (0,7) \\
\xrightarrow{\text{fill 21}} & (21,7) & \xrightarrow{\text{pour 21 to 26}} & (2,26) & \xrightarrow{\text{empty 26}} & (2,0) & \xrightarrow{\text{pour 21 to 26}} & (0,2) \\
\xrightarrow{\text{fill 21}} & (21,2) & \xrightarrow{\text{pour 21 to 26}} & (0,23) & & & & \\
\xrightarrow{\text{fill 21}} & (21,23) & \xrightarrow{\text{pour 21 to 26}} & (18,26) & \xrightarrow{\text{empty 26}} & (18,0) & \xrightarrow{\text{pour 21 to 26}} & (0,18) \\
\xrightarrow{\text{fill 21}} & (21,18) & \xrightarrow{\text{pour 21 to 26}} & (13,26) & \xrightarrow{\text{empty 26}} & (13,0) & \xrightarrow{\text{pour 21 to 26}} & (0,13) \\
\xrightarrow{\text{fill 21}} & (21,13) & \xrightarrow{\text{pour 21 to 26}} & (8,26) & \xrightarrow{\text{empty 26}} & (8,0) & \xrightarrow{\text{pour 21 to 26}} & (0,8) \\
\xrightarrow{\text{fill 21}} & (21,8) & \xrightarrow{\text{pour 21 to 26}} & (3,26) & \xrightarrow{\text{empty 26}} & (3,0) & \xrightarrow{\text{pour 21 to 26}} & (0,3)
\end{array}$$

$3 = 21s + 26t$

3 is a multiple of $\gcd(21,26) = 1$

The same approach works regardless of the jug capacities and even regardless the amount we’re trying to produce! Simply repeat these two steps until the desired amount of water is obtained:

1. Fill the smaller jug.

2. Pour all the water in the smaller jug into the larger jug. If at any time the larger jug becomes full, empty it out, and continue pouring the smaller jug into the larger jug.

By the same reasoning as before, this method eventually generates every multiple of the greatest common divisor of the jug capacities —all the quantities we can possibly produce. No ingenuity is needed at all!

So here the GCD was 1; see before

How/why is this
—reread for

8.3 The Fundamental Theorem of Arithmetic

We now have almost enough tools to prove something that you probably already know, namely, that every number has a unique prime factorization.

Let's state this more carefully. A sequence of numbers is *weakly decreasing* when each number in the sequence is $\geq$ the numbers after it. Note that a sequence of just one number as well as a sequence of no numbers —the empty sequence —is weakly decreasing by this definition.

Theorem 8.3.1 (Fundamental Theorem of Arithmetic). *Every positive integer is a product of a unique weakly decreasing sequence of primes.*

Notice that the theorem would be false if 1 were considered a prime; for example, 15 could be written as $5 \cdot 3$, or $5 \cdot 3 \cdot 1$, or $5 \cdot 3 \cdot 1 \cdot 1$, ...

There is a certain wonder in the Fundamental Theorem, even if you've known it since you were in a crib. Primes show up erratically in the sequence of integers. In fact, their distribution seems almost random:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, \ldots$$

Basic questions about this sequence have stumped humanity for centuries. And yet we know that every natural number can be built up from primes in *exactly one way*. These quirky numbers are the building blocks for the integers.

past
pset did
not show

The Fundamental Theorem is not hard to prove, but we'll need a couple of preliminary facts.

Lemma 8.3.2. *If p is a prime and $p \mid ab$, then $p \mid a$ or $p \mid b$.*

Proof. One case is if $\gcd(a, p) = p$. Then the claim holds, because a is a multiple of p.

Otherwise, $\gcd(a, p) \neq p$. In this case $\gcd(a, p)$ must be 1, since 1 and p are the only positive divisors of p. Since $\gcd(a, p)$ is a linear combination of a and p, we have $1 = sa + tp$ for some s, t. Then $b = s(ab) + (tb)p$, that is, b is a linear combination of ab and p. Since p divides both ab and p, it also divides their linear combination b. ■

A routine induction argument extends this statement to:

The Prime Number Theorem

Let $\pi(x)$ denote the number of primes less than or equal to x. For example, $\pi(10) = 4$ because 2, 3, 5, and 7 are the primes less than or equal to 10. Primes are very irregularly distributed, so the growth of π is similarly erratic. However, the Prime Number Theorem gives an approximate answer:

$$\lim_{x \to \infty} \frac{\pi(x)}{x / \ln x} = 1$$

Thus, primes gradually taper off. As a rule of thumb, about 1 integer out of every $\ln x$ in the vicinity of x is a prime.

The Prime Number Theorem was conjectured by Legendre in 1798 and proved a century later by de la Vallee Poussin and Hadamard in 1896. However, after his death, a notebook of Gauss was found to contain the same conjecture, which he apparently made in 1791 at age 15. (You sort of have to feel sorry for all the otherwise "great" mathematicians who had the misfortune of being contemporaries of Gauss.)

In late 2004 a billboard appeared in various locations around the country:

$$\left\{ \begin{array}{l} \text{first 10-digit prime found} \\ \text{in consecutive digits of } e \end{array} \right\} \textbf{. com}$$

Substituting the correct number for the expression in curly-braces produced the URL for a Google employment page. The idea was that Google was interested in hiring the sort of people that could and would solve such a problem.

How hard is this problem? Would you have to look through thousands or millions or billions of digits of e to find a 10-digit prime? The rule of thumb derived from the Prime Number Theorem says that among 10-digit numbers, about 1 in

$$\ln 10^{10} \approx 23$$

is prime. This suggests that the problem isn't really so hard! Sure enough, the first 10-digit prime in consecutive digits of e appears quite early:

$$\begin{aligned} e = {} & 2.71828182845904523536028747135266249775724709369995957496 6 \\ & 96762772407663035354759457138217852516642\mathbf{7427466391}9320030 \\ & 5992181741359662904357290033429526059563073813232862794 34\ldots \end{aligned}$$

Lemma 8.3.3. *Let p be a prime. If $p \mid a_1 a_2 \cdots a_n$, then p divides some a_i.*

Now we're ready to prove the Fundamental Theorem of Arithmetic.

Proof. Theorem 2.4.1 showed, using the Well Ordering Principle, that every positive integer can be expressed as a product of primes. So we just have to prove this expression is unique. We will use Well Ordering to prove this too.

The proof is by contradiction: assume, contrary to the claim, that there exist positive integers that can be written as products of primes in more than one way. By the Well Ordering Principle, there is a smallest integer with this property. Call this integer n, and let

$$\begin{aligned} n &= p_1 \cdot p_2 \cdots p_j, \\ &= q_1 \cdot q_2 \cdots q_k, \end{aligned}$$

where both products are in weakly decreasing order and $p_1 \le q_1$.

If $q_1 = p_1$, then n/q_1 would be a smaller number than n that was the product of different weakly decreasing sequences of primes, so $p_1 < q_1$. But since the p_i's are weakly decreasing, all the p_i's are less than q_1. But $q_1 \mid n = p_1 \cdot p_2 \cdots p_j$, so Lemma 8.3.3 implies that q_1 divides one of the p_i's, which contradicts the fact that q_1 is bigger than all them. ■

3/2

8.4 Alan Turing

The man pictured in Figure 8.1 is Alan Turing, the most important figure in the history of computer science. For decades, his fascinating life story was shrouded by government secrecy, societal taboo, and even his own deceptions.

At age 24, Turing wrote a paper entitled *On Computable Numbers, with an Application to the Entscheidungsproblem.* The crux of the paper was an elegant way to model a computer in mathematical terms. This was a breakthrough, because it allowed the tools of mathematics to be brought to bear on questions of computation. For example, with his model in hand, Turing immediately proved that there exist problems that no computer can solve—no matter how ingenious the programmer. Turing's paper is all the more remarkable because he wrote it in 1936, a full decade before any electronic computer actually existed.

was not before?

halting?

The word "Entscheidungsproblem" in the title refers to one of the 28 mathematical problems posed by David Hilbert in 1900 as challenges to mathematicians of the 20th century. Turing knocked that one off in the same paper. And perhaps you've heard of the "Church-Turing thesis"? Same paper. So Turing was obviously

Figure 8.1 Alan Turing

a brilliant guy who generated lots of amazing ideas. But this lecture is about one of Turing's less-amazing ideas. It involved codes. It involved number theory. And it was sort of stupid.

Let's look back to the fall of 1937. Nazi Germany was rearming under Adolf Hitler, world-shattering war looked imminent, and—like us—Alan Turing was pondering the usefulness of number theory. He foresaw that preserving military secrets would be vital in the coming conflict and proposed a way *to encrypt communications using number theory*. This is an idea that has ricocheted up to our own time. Today, number theory is the basis for numerous public-key cryptosystems, digital signature schemes, cryptographic hash functions, and electronic payment systems. Furthermore, military funding agencies are among the biggest investors in cryptographic research. Sorry Hardy!

Soon after devising his code, Turing disappeared from public view, and half a century would pass before the world learned the full story of where he'd gone and what he did there. We'll come back to Turing's life in a little while; for now, let's investigate the code Turing left behind. The details are uncertain, since he never formally published the idea, so we'll consider a couple of possibilities.

; not even secretly ;

8.4.1 Turing's Code (Version 1.0)

The first challenge is to translate a text message into an integer so we can perform mathematical operations on it. This step is not intended to make a message harder to read, so the details are not too important. Here is one approach: replace each letter of the message with two digits ($A = 01$, $B = 02$, $C = 03$, etc.) and string all the digits together to form one huge number. For example, the message "victory" could be translated this way:

$$\begin{array}{cccccccc} & \text{"v} & \text{i} & \text{c} & \text{t} & \text{o} & \text{r} & \text{y"} \\ \rightarrow & 22 & 09 & 03 & 20 & 15 & 18 & 25 \end{array}$$

symetric encryption?

Turing's code requires the message to be a prime number, so we may need to pad the result with a few more digits to make a prime. In this case, appending the digits 13 gives the number 2209032015182513, which is prime. ? how do we know what to append?

? I am bringing my old ideas into this Helps me understand

Here is how the encryption process works. In the description below, m is the unencoded message (which we want to keep secret), m^* is the encrypted message (which the Nazis may intercept), and k is the key. is that the right notation

— For rest of class abs. no bg

Beforehand The sender and receiver agree on a secret key, which is a large prime k.

Encryption The sender encrypts the message m by computing:

$$m^* = m \cdot k$$

Again what is modern notation multiple w/ key

Decryption The receiver decrypts m^* by computing:

$$\frac{m^*}{k} = \frac{m \cdot k}{k} = m$$

For example, suppose that the secret key is the prime number $k = 22801763489$ and the message m is "victory". Then the encrypted message is:

$$\begin{aligned} m^* &= m \cdot k \\ &= 2209032015182513 \cdot 22801763489 \\ &= 50369825549820718594667857 \end{aligned}$$

~~[illegible]~~ Knowing message helps deduce key
- repeating key
- crypto analysis

There are a couple of questions that one might naturally ask about Turing's code.

1. How can the sender and receiver ensure that m and k are prime numbers, as required?

   The general problem of determining whether a large number is prime or composite has been studied for centuries, and reasonably good primality tests were known even in Turing's time. In 2002, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena announced a primality test that is guaranteed to work on a number n in about $(\log n)^{12}$ steps, that is, a number of steps bounded by a twelfth degree polynomial in the length (in bits) of the input, n. This definitively places primality testing way below the problems of exponential difficulty. Amazingly, the description of their breakthrough algorithm was only thirteen lines long!

Of course, a twelfth degree polynomial grows pretty fast, so the Agrawal, *et al.* procedure is of no practical use. Still, good ideas have a way of breeding more good ideas, so there's certainly hope that further improvements will lead to a procedure that is useful in practice. But the truth is, there's no practical need to improve it, since very efficient *probabilistic* procedures for prime-testing have been known since the early 1970's. These procedures have some probability of giving a wrong answer, but their probability of being wrong is so tiny that relying on their answers is the best bet you'll ever make. how small?

2. Is Turing's code secure?

The Nazis see only the encrypted message $m^* = m \cdot k$, so recovering the original message m requires factoring m^*. Despite immense efforts, no really efficient factoring algorithm has ever been found. It appears to be a fundamentally difficult problem, though a breakthrough someday is not impossible. In effect, Turing's code puts to practical use his discovery that there are limits to the power of computation. Thus, provided m and k are sufficiently large, the Nazis seem to be out of luck!

factoring – decomposing into products of objects

factor$(15) = 3 \cdot 5$

If found, we're screwed!

This all sounds promising, but there is a major flaw in Turing's code.

8.4.2 Breaking Turing's Code

Let's consider what happens when the sender transmits a *second* message using Turing's code and the same key. This gives the Nazis two encrypted messages to look at:

$$m_1^* = m_1 \cdot k \qquad \text{and} \qquad m_2^* = m_2 \cdot k$$

The greatest common divisor of the two encrypted messages, m_1^* and m_2^*, is the secret key k. And, as we've seen, the GCD of two numbers can be computed very efficiently. So after the second message is sent, the Nazis can recover the secret key and read *every* message!

A mathematician as brilliant as Turing is not likely to have overlooked such a glaring problem, and we can guess that he had a slightly different system in mind,

review finding GCD

Oh divisor not denominator

confused the two

GCDivisor – largest polynomical that divides both evenly

largest d such that $p \mid d$ AND $q \mid d$

leading up good – straightforward def

perhaps have def in sidebar

Now this stuff getting excited

WP: Modular arametic – # wrap around after reach a certain value – the modulus – like a clock

divisor = factor = an integer that divides n w/o leaving a remainder

one based on *modular* arithmetic.

8.5 Modular Arithmetic

On page 1 of his masterpiece on number theory, *Disquisitiones Arithmeticae*, Gauss introduced the notion of "congruence". Now, Gauss is another guy who managed to cough up a half-decent idea every now and then, so let's take a look at this one. Gauss said that a is *congruent* to b *modulo* n iff $\boxed{n \mid (a-b)}$. This is written

here own def – nothing to do w/ geometry

$$a \equiv b \pmod{n}.$$

remainder $a = \underbrace{q \cdot d}_{\text{integers}} + r$

For example:

1234567 1234567 1234567 1234567 1

$$29 \equiv 15 \pmod 7 \quad \text{because } 7 \mid (29 - 15).$$

ins not what it seems from clock – oh addition is clock thing

There is a close connection between congruences and remainders:

modulo = a small measure
"modulo same" = same except for small measure

Lemma 8.5.1 (Congruences and Remainders).

$$a \equiv b \pmod{n} \quad \textit{iff} \quad \underline{\mathrm{rem}(\overset{\frac{a}{n}}{a}, n) = \mathrm{rem}(\overset{\frac{b}{n}}{b}, n)}.$$

Proof. By the Division Theorem 8.1.5, there exist unique pairs of integers q_1, r_1 and q_2, r_2 such that:

a|b means ak = b for some int k

$$a = q_1 n + r_1$$
$$b = q_2 n + r_2,$$

where $r_1, r_2 \in [0, n)$. Subtracting the second equation from the first gives:

$$a - b = (q_1 - q_2)n + (r_1 - r_2),$$

where $r_1 - r_2$ is in the interval $(-n, n)$. Now $a \equiv b \pmod{n}$ if and only if n divides the left side of this equation. This is true if and only if n divides the right side, which holds if and only if $r_1 - r_2$ is a multiple of n. Given the bounds on $r_1 - r_2$, this happens precisely when $r_1 = r_2$, that is, when $\mathrm{rem}(a, n) = \mathrm{rem}(b, n)$. ■

So we can also see that

$$29 \equiv 15 \pmod 7 \quad \text{because } \mathrm{rem}(29, 7) = 1 = \mathrm{rem}(15, 7).$$

This formulation explains why the congruence relation has properties like an equality relation. In particular, the following properties are follow immediately:

notation weird
diff order than what it means

15 plus some # of 7s will = 29
– its not addition, its congruency

Lemma 8.5.2.

$$a \equiv a \pmod{n} \quad \text{(reflexivity)}$$
$$a \equiv b \pmod{n} b \equiv a \pmod{n} \quad \text{(symmetry)}$$
$$a \equiv bpmodn \text{ and } b \equiv c \pmod{n} \text{ implies } a \equiv c \pmod{n} \quad \text{(transitivity)}$$

Notice that even though "(mod 7)" appears on the end, the $\equiv$ symbol isn't any more strongly associated with the 15 than with the 29. It would really be clearer to write $29 \equiv_7 15$ for example, but the notation with the modulus at the end is firmly entrenched and we'll stick to it.

We'll make frequent use of the following immediate Corollary of Lemma 8.5.1:

Corollary 8.5.3.

$$a \equiv \text{rem}(a, n) \pmod{n}$$

i kinda by def
– at least later

Still another way to think about congruence modulo n is that it *defines a partition of the integers into n sets so that congruent numbers are all in the same set*. For example, suppose that we're working modulo 3. Then we can partition the integers into 3 sets as follows:

$$\{\ \ldots,\ -6,\ -3,\ 0,\ 3,\ 6,\ 9,\ \ldots\ \}$$
$$\{\ \ldots,\ -5,\ -2,\ 1,\ 4,\ 7,\ 10,\ \ldots\ \}$$
$$\{\ \ldots,\ -4,\ -1,\ 2,\ 5,\ 8,\ 11,\ \ldots\ \}$$

all are equivilant
up + down the list

3/10 (never thought of it that way!)

according to whether their remainders on division by 3 are 0, 1, or 2. The upshot is that when arithmetic is done modulo n there are really only n different kinds of numbers to worry about, because there are only n possible remainders. In this sense, modular arithmetic is a simplification of ordinary arithmetic and thus is a good reasoning tool.

The next most useful fact about congruences is that they are *preserved* by addition and multiplication:

Lemma 8.5.4. *For $n \geq 1$, if $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$, then*

1. $a + c \equiv b + d \pmod{n}$,
2. $ac \equiv bd \pmod{n}$.

Proof. We have that n divides $(b - a)$ which is equal to $(b + c) - (a + c)$, so

$$a + c \equiv b + c \pmod{n}.$$

Also, n divides $(d - c)$, so by the same reasoning

$$b + c \equiv b + d \pmod{n}.$$

Combining these according to Lemma 8.5.2, we get

$$a + c \equiv b + d \pmod{n}.$$

The proof for multiplication is virtually identical, using the fact that if n divides $(b - a)$, then it obviously divides $(bc - ac)$ as well. ■

The overall theme is that *congruences work a lot like equations*, though there are a couple of exceptions.

8.5.1 Turing's Code (Version 2.0)

In 1940, France had fallen before Hitler's army, and Britain stood alone against the Nazis in western Europe. British resistance depended on a steady flow of supplies brought across the north Atlantic from the United States by convoys of ships. These convoys were engaged in a cat-and-mouse game with German "U-boats"—submarines—which prowled the Atlantic, trying to sink supply ships and starve Britain into submission. The outcome of this struggle pivoted on a balance of information: could the Germans locate convoys better than the Allies could locate U-boats or vice versa?

Germany lost.

But a critical reason behind Germany's loss was made public only in 1974: Germany's naval code, *Enigma*, had been broken by the Polish Cipher Bureau (see `http://en.wikipedia.org/wiki/Polish_Cipher_Bureau`) and the secret had been turned over to the British a few weeks before the Nazi invasion of Poland in 1939. Throughout much of the war, the Allies were able to route convoys around German submarines by listening in to German communications. The British government didn't explain *how* Enigma was broken until 1996. When it was finally released (by the US), the story revealed that Alan Turing had joined the secret British codebreaking effort at Bletchley Park in 1939, where he became the lead developer of methods for rapid, bulk decryption of German Enigma messages. Turing's Enigma deciphering was an invaluable contribution to the Allied victory over Hitler.

Governments are always tight-lipped about cryptography, but the half-century of official silence about Turing's role in breaking Enigma and saving Britain may be related to some disturbing events after the war. More on that later. Let's get back to number theory and consider an alternative interpretation of Turing's code. Perhaps we had the basic idea right (multiply the message by the key), but erred in using *conventional* arithmetic instead of *modular* arithmetic. Maybe this is what Turing meant:

? public key crypto?

Beforehand The sender and receiver agree on a large prime p, which may be made public. (This will be the modulus for all our arithmetic.) They also agree on a secret key $k \in [1, p)$.

Encryption The message m can be any integer in $[0, p)$; in particular, the message is no longer required to be a prime. The sender encrypts the message m to produce m^* by computing:

$$m^* = \text{rem}(mk, p) \tag{8.6}$$

$\text{rem}\left(\frac{mk}{p}\right)$

Decryption (Uh-oh.)

The decryption step is a problem. We might hope to decrypt in the same way as before: by dividing the encrypted message m^* by the key k. The difficulty is that m^* is the *remainder* when mk is divided by p. So dividing m^* by k might not even give us an integer!

This decoding difficulty can be overcome with a better understanding of arithmetic modulo a prime.

8.6 Arithmetic with a Prime Modulus

8.6.1 Multiplicative Inverses

The *multiplicative inverse* of a number x is another number x^{-1} such that:

$$x \cdot x^{-1} = 1$$

Generally, multiplicative inverses exist over the real numbers. For example, the multiplicative inverse of 3 is $1/3$ since:

$$3 \cdot \frac{1}{3} = 1$$

The sole exception is that 0 does not have an inverse. On the other hand, over the integers, only 1 and -1 have inverses.

Surprisingly, multiplicative inverses do exist when we're working *modulo a prime number*. For example, if we're working modulo 5, then 3 is a multiplicative inverse of 7, since:

$$7 \cdot 3 \equiv 1 \pmod{5}$$

must be prime

(All numbers congruent to 3 modulo 5 are also multiplicative inverses of 7; for example, $7 \cdot 8 \equiv 1 \pmod{5}$ as well.) The only exception is that numbers congruent to 0 modulo 5 (that is, the multiples of 5) do not have inverses, much as 0 does not have an inverse over the real numbers. Let's prove this.

Lemma 8.6.1. *If p is prime and k is not a multiple of p, then k has a multiplicative inverse modulo p.*

Proof. Since p is prime, it has only two divisors: 1 and p. And since k is not a multiple of p, we must have $\gcd(p,k) = 1$. Therefore, there is a linear combination of p and k equal to 1:

$$sp + tk = 1$$

Rearranging terms gives:

$$sp = 1 - tk$$

This implies that $p \mid (1 - tk)$ by the definition of divisibility, and therefore $tk \equiv 1 \pmod{p}$ by the definition of congruence. Thus, t is a multiplicative inverse of k. ■

Multiplicative inverses are the key to decryption in Turing's code. Specifically, we can recover the original message by multiplying the encoded message by the *inverse* of the key:

$$\begin{aligned} m^* \cdot k^{-1} &= \text{rem}(mk, p) \cdot k^{-1} && \text{(the def. (8.6) of } m^*) \\ &\equiv (mk)k^{-1} \pmod{p} && \text{(by Cor. 8.5.3)} \\ &\equiv m \pmod{p}. \end{aligned}$$

This shows that m^*k^{-1} is congruent to the original message m. Since m was in $[0, p)$, we can recover it exactly by taking a remainder:

$$m = \text{rem}(m^*k^{-1}, p).$$

So all we need to decrypt the message is to find a value of k^{-1}. From the proof of Lemma 8.6.1, we know that t is such a value, where $sp + tk = 1$. Finding t is easy using the Pulverizer. Oh this is pulverizer for inverse

8.6.2 Cancellation

did in class

Another sense in which real numbers are nice is that one can cancel multiplicative terms. In other words, if we know that $m_1k = m_2k$, then we can cancel the k's and conclude that $m_1 = m_2$, provided $k \neq 0$. In general, cancellation is *not* valid in modular arithmetic. For example,

$$2 \cdot 3 \equiv 4 \cdot 3 \pmod{6},$$

but canceling the 3’s leads to the *false* conclusion that $2 \equiv 4 \pmod{6}$. The fact that multiplicative terms can not be canceled is the most significant sense in which congruences differ from ordinary equations. However, this difference goes away if we’re working modulo a *prime*; then cancellation is valid.

Lemma 8.6.2. *Suppose p is a prime and k is not a multiple of p. Then*

$$ak \equiv bk \pmod{p} \quad \text{IMPLIES} \quad a \equiv b \pmod{p}.$$

Proof. Multiply both sides of the congruence by k^{-1}. ■

We can use this lemma to get a bit more insight into how Turing’s code works. In particular, the encryption operation in Turing’s code *permutes the set of possible messages*. This is stated more precisely in the following corollary.

Corollary 8.6.3. *Suppose p is a prime and k is not a multiple of p. Then the sequence:*

$$\text{rem}((1 \cdot k), p), \quad \text{rem}((2 \cdot k), p), \quad \ldots, \quad \text{rem}(((p-1) \cdot k), p)$$

is a permutation[3] *of the sequence:*

$$1, \quad 2, \quad \ldots, \quad (p-1).$$

Proof. The sequence of remainders contains $p - 1$ numbers. Since $i \cdot k$ is not divisible by p for $i = 1, \ldots p-1$, all these remainders are in $[1, p)$ by the definition of remainder. Furthermore, the remainders are all different: no two numbers in $[1, p)$ are congruent modulo p, and by Lemma 8.6.2, $i \cdot k \equiv j \cdot k \pmod{p}$ if and only if $i \equiv j \pmod{p}$. Thus, the sequence of remainders must contain *all* of $[1, p)$ in some order. ■

For example, suppose $p = 5$ and $k = 3$. Then the sequence:

$$\underbrace{\text{rem}((1 \cdot 3), 5)}_{=3}, \quad \underbrace{\text{rem}((2 \cdot 3), 5)}_{=1}, \quad \underbrace{\text{rem}((3 \cdot 3), 5)}_{=4}, \quad \underbrace{\text{rem}((4 \cdot 3), 5)}_{=2}$$

re arranged

is a permutation of 1, 2, 3, 4. As long as the Nazis don’t know the secret key k, they don’t know how the set of possible messages are permuted by the process of encryption and thus they can’t read encoded messages.

"really"

[3] A *permutation* of a sequence of elements is a reordering of the elements.

8.6.3 Fermat's Little Theorem

k = thing finding inverse of

An alternative approach to finding the inverse of the secret key k in Turing's code is to rely on Fermat's Little Theorem, which is much easier than his famous Last Theorem.

Theorem 8.6.4 (Fermat's Little Theorem). *Suppose p is a prime and k is not a multiple of p. Then:*

$$k^{p-1} \equiv 1 \pmod{p}$$

Proof. We reason as follows:

$$\begin{aligned}
(p-1)! &::= 1 \cdot 2 \cdots (p-1) \\
&= \text{rem}(k, p) \cdot \text{rem}(2k, p) \cdots \text{rem}((p-1)k, p) && \text{(by Cor 8.6.3)} \\
&\equiv k \cdot 2k \cdots (p-1)k \pmod{p} && \text{(by Cor 8.5.3)} \\
&\equiv (p-1)! \cdot k^{p-1} \pmod{p} && \text{(rearranging terms)}
\end{aligned}$$

Now $(p-1)!$ is not a multiple of p because the prime factorizations of $1, 2, \ldots, (p-1)$ contain only primes smaller than p. So by Lemma 8.6.2, we can cancel $(p-1)!$ from the first and last expressions, which proves the claim. ■

Here is how we can find inverses using Fermat's Theorem. Suppose p is a prime and k is not a multiple of p. Then, by Fermat's Theorem, we know that:

$$k^{p-2} \cdot k \equiv 1 \pmod{p}$$

Therefore, k^{p-2} must be a multiplicative inverse of k. For example, suppose that we want the multiplicative inverse of 6 modulo 17. Then we need to compute rem$(6^{15}, 17)$, which we can do using the fast exponentiation procedure of Section 6.2.5, with all the arithemetic done modulo 17. Namely,

$$\begin{aligned}
(6, 1, 15) &\longrightarrow (36, 6, 7) \equiv (2, 6, 7) \longrightarrow (4, 12, 3) \\
&\longrightarrow (16, 14, 1) \longrightarrow (256, 224, 1) \equiv (1, 3, 0).
\end{aligned}$$

where the $\equiv$'s are modulo 17. Therefore, $6^{15} \equiv 3 \pmod{17}$. Sure enough, 3 is the multiplicative inverse of 6 modulo 17 since

$$3 \cdot 6 = 18 \equiv 1 \pmod{17}.$$

In general, if we were working modulo a prime p, finding a multiplicative inverse by trying every value in $[1, p)$ would require about p operations. However, this approach, like the Pulverizer, requires only about $\log p$ transition, which is far better when p is large.

8.6.4 Breaking Turing's Code—Again

The Germans didn't bother to encrypt their weather reports with the highly-secure Enigma system. After all, so what if the Allies learned that there was rain off the south coast of Iceland? But, amazingly, this practice provided the British with a critical edge in the Atlantic naval battle during 1941.

What did Enigma use?

The problem was that some of those weather reports had originally been transmitted using Enigma from U-boats out in the Atlantic. Thus, the British obtained both unencrypted reports and the same reports encrypted with Enigma. By comparing the two, the British were able to determine which key the Germans were using that day and could read all other Enigma-encoded traffic. Today, this would be called a *known-plaintext attack*.

know plain text get key

Let's see how a known-plaintext attack would work against Turing's code. Suppose that the Nazis know both m and m^* where:

$$m^* \equiv mk \pmod{p}$$

Now they can compute:

multiple both sides by m^{p-2}

$$\begin{aligned}
\boxed{m^{p-2}} \cdot m^* &= m^{p-2} \cdot \text{rem}(mk, p) && \text{(def. (8.6) of } m^*) \\
&\equiv m^{p-2} \cdot mk \pmod{p} && \text{(by Cor 8.5.3)} \\
&\equiv m^{p-1} \cdot k \pmod{p} && \text{(Fermat's Theorem)} \\
&\equiv k \pmod{p} && \text{(Fermat's Little Theorem)}
\end{aligned}$$

oh just exponential math

Finding inverse

Now the Nazis have the secret key k and can decrypt any message!

This is a huge vulnerability, so Turing's code has no practical value. Fortunately, Turing got better at cryptography after devising this code; his subsequent deciphering of Enigma messages surely saved thousands of lives, if not the whole of Britain.

8.6.5 Turing Postscript

A few years after the war, Turing's home was robbed. Detectives soon determined that a former homosexual lover of Turing's had conspired in the robbery. So they arrested him—that is, they arrested Alan Turing—because homosexuality was a British crime punishable by up to two years in prison at that time. Turing was sentenced to a hormonal "treatment" for his homosexuality: he was given estrogen injections. He began to develop breasts.

Three years later, Alan Turing, the founder of computer science, was dead. His mother explained what happened in a biography of her own son. Despite her repeated warnings, Turing carried out chemistry experiments in his own home. Apparently, her worst fear was realized: by working with potassium cyanide while eating an apple, he poisoned himself.

However, Turing remained a puzzle to the very end. His mother was a devoutly religious woman who considered suicide a sin. And, other biographers have pointed out, Turing had previously discussed committing suicide by eating a poisoned apple. Evidently, Alan Turing, who founded computer science and saved his country, took his own life in the end, and in just such a way that his mother could believe it was an accident.

Turing's last project before he disappeared from public view in 1939 involved the construction of an elaborate mechanical device to test a mathematical conjecture called the Riemann Hypothesis. This conjecture first appeared in a sketchy paper by Bernhard Riemann in 1859 and is now one of the most famous unsolved problem in mathematics.

8.7 Arithmetic with an Arbitrary Modulus

Turing's code did not work as he hoped. However, his essential idea—using number theory as the basis for cryptography—succeeded spectacularly in the decades after his death.

In 1977, Ronald Rivest, Adi Shamir, and Leonard Adleman at MIT proposed a highly secure cryptosystem (called **RSA**) based on number theory. Despite decades of attack, no significant weakness has been found. Moreover, RSA has a major advantage over traditional codes: the sender and receiver of an encrypted message need not meet beforehand to agree on a secret key. Rather, the receiver has both a *secret key*, which she guards closely, and a *public key*, which she distributes as widely as possible. The sender then encrypts his message using her widely-distributed public key. Then she decrypts the received message using her closely-held private key. The use of such a *public key cryptography* system allows you and Amazon, for example, to engage in a secure transaction without meeting up beforehand in a dark alley to exchange a key.

Here is public key crypto

Interestingly, RSA does not operate modulo a prime, as Turing's scheme may have, but rather modulo the product of *two* large primes. Thus, we'll need to know a bit about how arithmetic works modulo a composite number in order to understand RSA. Arithmetic modulo an arbitrary positive integer is really only a little more painful than working modulo a prime—though you may think this is like the doctor saying, "This is only going to hurt a little," before he jams a big needle in your arm.

don't get why different for prime

The Riemann Hypothesis

The formula for the sum of an infinite geometric series says:

$$1 + x + x^2 + x^3 + \cdots = \frac{1}{1-x}$$

Substituting $x = \frac{1}{2^s}$, $x = \frac{1}{3^s}$, $x = \frac{1}{5^s}$, and so on for each prime number gives a sequence of equations:

$$1 + \frac{1}{2^s} + \frac{1}{2^{2s}} + \frac{1}{2^{3s}} + \cdots = \frac{1}{1 - 1/2^s}$$
$$1 + \frac{1}{3^s} + \frac{1}{3^{2s}} + \frac{1}{3^{3s}} + \cdots = \frac{1}{1 - 1/3^s}$$
$$1 + \frac{1}{5^s} + \frac{1}{5^{2s}} + \frac{1}{5^{3s}} + \cdots = \frac{1}{1 - 1/5^s}$$

etc.

Multiplying together all the left sides and all the right sides gives:

d multiply by

$$\sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{p \in \text{primes}} \left(\frac{1}{1 - 1/p^s} \right)$$

The sum on the left is obtained by multiplying out all the infinite series and applying the Fundamental Theorem of Arithmetic. For example, the term $1/300^s$ in the sum is obtained by multiplying $1/2^{2s}$ from the first equation by $1/3^s$ in the second and $1/5^{2s}$ in the third. Riemann noted that every prime appears in the expression on the right. So he proposed to learn about the primes by studying the equivalent, but simpler expression on the left. In particular, he regarded s as a complex number and the left side as a function, $\zeta(s)$. Riemann found that the distribution of primes is related to values of s for which $\zeta(s) = 0$, which led to his famous conjecture:

every non-constant single-variable polynomial w/ complex coefficients has at least 1 root

Definition 8.6.5. *The Riemann Hypothesis*: Every nontrivial zero of the zeta function $\zeta(s)$ lies on the line $s = 1/2 + ci$ in the complex plane.

A proof would immediately imply, among other things, a strong form of the Prime Number Theorem.

Researchers continue to work intensely to settle this conjecture, as they have for over a century. It is another of the Millennium Problems whose solver will earn \$1,000,000 from the Clay Institute.

If $\gcd(a,b) = 1$

True for any #, p

Basically no common factors (except 1)

8.7.1 Relative Primality

What does this mean? together?

— Yes it is talking about a pair of #s

First, we need a new definition. Integers a and b are *relatively prime* iff $\gcd(a, b) = 1$. For example, 8 and 15 are relatively prime, since $\gcd(8, 15) = 1$. Note that, except for multiples of p, every integer is relatively prime to a prime number p.

Next we'll need to generalize what we know about arithmetic modulo a prime to work modulo an arbitrary positive integer n. The basic theme is that arithmetic modulo n may be complicated, but the integers *relatively prime* to n remain fairly well-behaved. For example,

Lemma 8.7.1. *Let n be a positive integer. If k is relatively prime to n, then there exists an integer k^{-1} such that:*

$$k \cdot k^{-1} \equiv 1 \pmod{n}.$$

An inverse for any k relatively prime to n is simply the coefficient of k in the linear combination of k and n that equals 1, exactly as in the proof of Lemma 8.6.1.

As a consequence of this lemma, we can cancel a multiplicative term from both sides of a congruence if that term is relatively prime to the modulus:

Corollary 8.7.2. *Suppose n is a positive integer and k is relatively prime to n. If*

$$ak \equiv bk \pmod{n}$$

then

$$a \equiv b \pmod{n}$$

This holds because we can multiply both sides of the first congruence by k^{-1} and simplify to obtain the second.

The following lemma is the natural generalization of Corollary 8.6.3 with much the same proof.

Lemma 8.7.3. *Suppose n is a positive integer and k is relatively prime to n. Let $k_1, \ldots, k_r$ denote all the integers relatively prime to n in the range 1 to $n-1$. Then the sequence:*

$$\text{rem}(k_1 \cdot k, n), \quad \text{rem}(k_2 \cdot k, n), \quad \text{rem}(k_3 \cdot k, n), \ldots \quad, \text{rem}(k_r \cdot k, n)$$

is a permutation of the sequence:

$$k_1, \quad k_2, \ldots, \quad k_r.$$

Proof. We will show that the remainders in the first sequence are all distinct and are equal to some member of the sequence of k_j's. Since the two sequences have the same length, the first must be a permutation of the second.

First, we show that the remainders in the first sequence are all distinct. Suppose that $\text{rem}(k_i k, n) = \text{rem}(k_j k, n)$. This is equivalent to $k_i k \equiv k_j k \pmod{n}$, which implies $k_i \equiv k_j \pmod{n}$ by Corollary 8.7.2. This, in turn, means that $k_i = k_j$ since both are in $[1, n)$. Thus, none of the remainder terms in the first sequence is equal to any other remainder term.

Next, we show that each remainder in the first sequence equals one of the k_i. By assumption, k_i and k are relatively prime to n, and therefore so is $k_i k$ by Unique Factorization. Hence,

$$\begin{aligned} \gcd(n, \text{rem}(k_i k, n)) &= \gcd(k_i k, n) \qquad \text{(Lemma 8.2.1)} \\ &= 1. \end{aligned}$$

Since $\text{rem}(k_i k, n)$ is in $[0, n)$ by the definition of remainder, and since it is relatively prime to n, it must, by their definition, be equal to one of the k_i's. ■

8.7.2 Euler's Theorem

RSA relies heavily on a generalization of Fermat's Theorem known as Euler's Theorem. For both theorems, the exponent of k needed to produce an inverse of k modulo n depends on the number, $\phi(n)$, of integers in $[0, n)$, that are relatively prime to n. This function ϕ is known as Euler's ϕ or *totient function*). For example, $\phi(7) = 6$ since 1, 2, 3, 4, 5, and 6 are all relatively prime to 7. Similarly, $\phi(12) = 4$ since 1, 5, 7, and 11 are the only numbers in $[1, 12]$ that are relatively prime to 12.

Cool

If n is prime, then $\phi(n) = n - 1$ since positive every number less than a prime number is relatively prime to that prime. When n is composite, however, the ϕ function gets a little complicated. We'll get back to it in the next section.

We can now prove Euler's Theorem:

Theorem 8.7.4 (Euler's Theorem)**.** *Suppose n is a positive integer and k is relatively prime to n. Then*

$$k^{\phi(n)} \equiv 1 \pmod{n}$$

Proof. Let $k_1, \ldots, k_r$ denote all integers relatively prime to n where $k_i \in [0, n)$. Then $r = \phi(n)$, by the definition of the function ϕ. The remainder of the proof mirrors the proof of Fermat's Theorem. In particular,

$$\begin{aligned} &k_1 \cdot k_2 \cdots k_r \\ &\quad = \text{rem}(k_1 \cdot k, n) \cdot \text{rem}(k_2 \cdot k, n) \cdots \text{rem}(k_r \cdot k, n) && \text{(by Lemma 8.7.3)} \\ &\quad \equiv (k_1 \cdot k) \cdot (k_2 \cdot k) \cdots\cdot (k_r \cdot k) \pmod{n} && \text{(by Cor 8.5.3)} \\ &\quad \equiv (k_1 \cdot k_2 \cdots k_r) \cdot k^r \pmod{n} && \text{(rearranging terms)} \end{aligned}$$

By Lemma 8.7.2, each of the terms k_i can be cancelled, proving the claim. ■

3/10 reading through again makes things much clearer!

We can find multiplicative inverses using Euler's theorem as we did with Fermat's theorem: if k is relatively prime to n, then $\boxed{k^{\phi(n)-1}}$ is a multiplicative inverse of k modulo n. However, this approach requires computing $\phi(n)$. In the next section, we'll show that computing $\phi(n)$ is easy *if* we know the prime factorization of n. Unfortunately, finding the factors of n can be hard to do when n is large, and so the Pulverizer is generally the best approach to computing inverses modulo n.

8.7.3 Computing Euler's Function

RSA works using arithmetic modulo the product of two large primes, so we begin with an elementary explanation of how to compute $\phi(pq)$ for primes p and q:

Lemma 8.7.5.

$$\phi(\overset{pq}{n}) = (p-1)(q-1)$$

for primes $p \neq q$.

Proof. Since p and q are prime, any number that is not relatively prime to $n = pq$ must be a multiple of p or a multiple of q. Among the pq numbers in $[0, pq)$, there are precisely q multiples of p and p multiples of q. Since p and q are relatively prime, the only number in $[0, pq)$ that is a multiple of both p and q is 0. Hence, there are $p+q-1$ numbers in $[0, pq)$ that are *not* relatively prime to n. This means that

$$\begin{aligned} \phi(n) &= pq - (p+q-1) \\ &= (p-1)(q-1), \end{aligned}$$

as claimed.[4] ■

The following theorem provides a way to calculate $\phi(n)$ for arbitrary n.

Theorem 8.7.6.

[4] This proof provides a brief preview of the kinds of counting arguments that we will explore more fully in Part III.

(a) If p is a prime, then $\phi(p^k) = p^k - p^{k-1}$ for $k \geq 1$.

(b) If a and b are relatively prime, then $\phi(ab) = \phi(a)\phi(b)$.

Here's an example of using Theorem 8.7.6 to compute $\phi(300)$:

$$\begin{aligned}
\phi(300) &= \phi(2^2 \cdot 3 \cdot 5^2) \\
&= \phi(2^2) \cdot \phi(3) \cdot \phi(5^2) && \text{(by Theorem 8.7.6.(a))} \\
&= (2^2 - 2^1)(3^1 - 3^0)(5^2 - 5^1) && \text{(by Theorem 8.7.6.(b))} \\
&= 80.
\end{aligned}$$

To prove Theorem 8.7.6.(a), notice that every pth number among the p^k numbers in $[0, p^k - 1]$ is divisible by p, and only these are divisible by p. So $1/p$ of these numbers are divisible by p and the remaining ones are not. That is,

$$\phi(p^k) = p^k - (1/p)p^k = p^k - p^{k-1}.$$

We'll leave a proof of Theorem 8.7.6.(b) to Problem 8.20.

As a consequence of Theorem 8.7.6, we have

Corollary 8.7.7. *For any number n, if $p_1, p_2, \ldots, p_j$ are the (distinct) prime factors of n, then*

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_j}\right).$$

We'll give another proof of Corollary 8.7.7 in a few weeks based on rules for counting things.

8.8 The RSA Algorithm

Finally, we are ready to see how the *RSA public key encryption scheme* works. The details are in the box on the next page.

It is not immediately clear from the description of the RSA cryptosystem that the decoding of the encrypted message is, in fact, the original unencrypted message. We'll work that out in class.

I don't like this put it in book

Is it hard for someone without the secret key to decrypt the message? No one knows for sure but it is generally believed that if n is a very large number (say, with a thousand digits), then it is difficult to reverse engineer d from e and n. Of course, it is easy to compute d if you know p and q (by using the Pulverizer) but it is not

The RSA Cryptosystem

Beforehand The receiver creates a public key and a secret key as follows.

1. Generate two distinct primes, p and q. Since they can be used to generate the secret key, they must be kept hidden.
2. Let $n = pq$.
3. Select an integer e such that $\gcd(e, (p-1)(q-1)) = 1$. ← what is this? inverse?
   The *public key* is the pair (e, n). This should be distributed widely.
4. Compute d such that $de \equiv 1 \pmod{(p-1)(q-1)}$. This can be done using the Pulverizer. find multiplicate inverse of e mod $(p-1)(q-1) = d$
   The *secret key* is the pair (d, n). This should be kept hidden!

Encoding Given a message m, the sender first checks that $\gcd(m, n) = 1$.

The sender then encrypts message m to produce m^* using the public key:

$$m^* = \text{rem}(m^e, n).$$

Decoding The receiver decrypts message m^* back to message m using the secret key:

$$m = \text{rem}((m^*)^d, n).$$

m^* = encrypted message

So GCD easy but factoring not

known how to quickly factor n into p and q when n is very large. Maybe with a little more studying of number theory, you will be the first to figure out how to do it. Although, we should warn you that Gauss worked on it for years without a lot to show for his efforts. And if you do figure it out, you might wind up meeting some serious-looking fellows in black suits....

8.9 What has SAT got to do with it?

What is this again? -- reread earlier Chap

So why does the world, or at least the world's secret codes, fall apart if there is an efficient test for satisfiability? To explain this, remember that RSA can be managed computationally because multiplication of two primes is fast, but factoring a product of two primes seems to be overwhelmingly demanding.

Now designing digital multiplication circuits is completely routine. This means we can easily build a digital circuit out of AND, OR, and NOTgates that can take two input strings u, v of length n, and a third input string, z, of length $2n$, and "check" if z represents the product of the numbers represented by u and v. That is, it gives output 1 if z represents the product of u and v, and gives output 0 otherwise.

Now here's how to factor any number with a length $2n$ representation using a SAT solver. Fix the z input to be the representation of the number to be factored. Set the first digit of the u input to 1, and do a SAT test to see if there is a satisfying assignment of values for the remaining bits of u and v. That is, see if the remaining bits of u and v can be filled in to cause the circuit to give output 1. If there is such an assignment, fix the first bit of u to 1, otherwise fix the first bit of u to be 0. Now do the same thing to fix the second bit of u and then third, proceeding in this way through all the bits of u and then of v. The result is that after $2n$ SAT tests, we have found an assignment of values for u and v that makes the circuit give output 1. So u and v represent factors of the number represented by z. This means that if SAT could be done in time bounded by a degree d polynomial in n, then $2n$ digit numbers can be factored in time bounded by a polynomial in n of degree $d + 1$. In sum, if SAT was easy, then so is factoring, and so RSA would be easy to break.

read crypto engineering chap on Primes first & all of the diff formulas
need scope of what is out there

8.10 Problems

Exam Problems

Problem 8.1.
Find the remainder of $26^{1818181}$ divided by 297. *Hint:* $1818181 = (180 \cdot 10101) + 1$; Euler's theorem

Problem 8.2.
Find an integer $k > 1$ such that n and n^k agree in their last three digits whenever n is divisible by neither 2 nor 5. *Hint:* Euler's theorem.

Problems for Section 8.1

Practice Problems

Problem 8.3.
Prove that a linear combination of linear combinations of integers $a_0, \ldots, a_n$ is a linear combination of $a_0, \ldots, a_n$.

Class Problems

Problem 8.4.
A number is *perfect* if it is equal to the sum of its positive divisors, other than itself. For example, 6 is perfect, because $6 = 1 + 2 + 3$. Similarly, 28 is perfect, because $28 = 1 + 2 + 4 + 7 + 14$. Explain why $2^{k-1}(2^k - 1)$ is perfect when $2^k - 1$ is prime.[5]

Problems for Section 8.2

Class Problems

Problem 8.5. (a) Use the Pulverizer to find integers x, y such that

$$x \cdot 50 + y \cdot 21 = \gcd(50, 21).$$

[5]Euclid proved this 2300 years ago. About 250 years ago, Euler proved the converse: *every* even perfect number is of this form (for a simple proof see `http://primes.utm.edu/notes/proofs/EvenPerfect.html`). As is typical in number theory, apparently simple results lie at the brink of the unknown. For example, it is not known if there are an infinite number of even perfect numbers or any odd perfect numbers at all.

(b) Now find integers x', y' with $y' > 0$ such that

$$x' \cdot 50 + y' \cdot 21 = \gcd(50, 21)$$

Problem 8.6.
For nonzero integers, a, b, prove the following properties of divisibility and GCD'S. (You may use the fact that $\gcd(a, b)$ is an integer linear combination of a and b. You may *not* appeal to uniqueness of prime factorization because the properties below are needed to *prove* unique factorization.)

(a) Every common divisor of a and b divides $\gcd(a, b)$.

(b) If $a \mid bc$ and $\gcd(a, b) = 1$, then $a \mid c$.

(c) If $p \mid ab$ for some prime, p, then $p \mid a$ or $p \mid b$.

(d) Let m be the smallest integer linear combination of a and b that is positive. Show that $m = \gcd(a, b)$.

Homework Problems

Problem 8.7.
Let's extend the jug filling scenario of Section 8.1.3 to three jugs and a receptacle. The receptacle can be used to store an unlimited amount of water, but has no measurement markings. Excess water can be dumped into the drain. Among the possible moves are:

1. fill a bucket from the hose,
2. pour from the receptacle to a bucket until the bucket is full or the receptacle is empty, whichever happens first,
3. empty a bucket to the drain,
4. empty a bucket to the receptacle,
5. pour from one bucket to another until either the first is empty or the second is full,

(a) Model this scenario with a state machine. (What are the states? How does a state change in response to a move?)

(b) Prove that Bruce can get $k \in \mathbb{N}$ gallons of water into the receptacle using the above operations only if $\gcd(a, b) \mid k$.

Problem 8.8.
Define the Pulverizer State machine to have:

$$\begin{aligned}
\text{states} &::= \mathbb{N}^7 \\
\text{start state} &::= (a, b, 0, 1, 1, 0) && \text{(where } a \geq b > 0\text{)} \\
\text{transitions} &::= (x, y, s, t, u, v) \longrightarrow \\
&\quad (y, \text{rem}(x, y), u - s\,\text{qcnt}(x, y), v - t\,\text{qcnt}(x, y), s, t) && \text{(for } y > 0\text{)}
\end{aligned}$$

Note that x, y follows the transition rules of the Euclidean algorithm given in equation (8.3), except that this machine stops one step sooner, ensuring that $y \gcd(a, b)$ at the end. So for all inputs x, y, this procedure terminates after at most the same of transitions as the Euclidean algorithm.

(a) Show that the following properties are preserved invariants of the Pulverizer machine:

$$\gcd(x, y) = \gcd(a, b), \tag{8.7}$$
$$sa + tb = y, \text{ and} \tag{8.8}$$
$$ua + vb = x. \tag{8.9}$$

(b) Conclude that the Pulverizer machine is partially correct.

Problems for Section 8.3

Class Problems

Problem 8.9. (a) Let $m = 2^9 5^{24} 11^7 17^{12}$ and $n = 2^3 7^{22} 11^{211} 13^1 17^9 19^2$. What is the $\gcd(m, n)$? What is the *least common multiple*, $\text{lcm}(m, n)$, of m and n? Verify that

$$\gcd(m, n) \cdot \text{lcm}(m, n) = mn. \tag{8.10}$$

(b) Describe in general how to find the $\gcd(m, n)$ and $\text{lcm}(m, n)$ from the prime factorizations of m and n. Conclude that equation (8.10) holds for all positive integers m, n.

Problems for Section 8.5

Class Problems

Problem 8.10.
The following properties of equivalence mod n follow directly from its definition and simple properties of divisibility. See if you can prove them without looking up the proofs in the text.

(a) If $a \equiv b \pmod{n}$, then $ac \equiv bc \pmod{n}$.

(b) If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$.

(c) If $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$, then $ac \equiv bd \pmod{n}$.

(d) $\text{rem}(a, n) \equiv a \pmod{n}$.

Problem 8.11. (a) Why is a number written in decimal evenly divisible by 9 if and only if the sum of its digits is a multiple of 9? *Hint:* $10 \equiv 1 \pmod{9}$.

(b) Take a big number, such as 37273761261. Sum the digits, where every other one is negated:

$$3 + (-7) + 2 + (-7) + 3 + (-7) + 6 + (-1) + 2 + (-6) + 1 = -11$$

Explain why the original number is a multiple of 11 if and only if this sum is a multiple of 11.

Problem 8.12.
At one time, the Guinness Book of World Records reported that the "greatest human calculator" was a guy who could compute 13th roots of 100-digit numbers that were powers of 13. What a curious choice of tasks ....

(a) Prove that

$$d^{13} \equiv d \pmod{10} \tag{8.11}$$

for $0 \leq d < 10$.

(b) Now prove that

$$n^{13} \equiv n \pmod{10} \tag{8.12}$$

for all n.

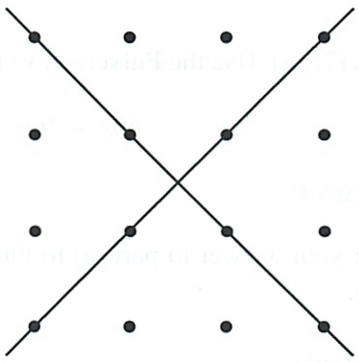
Problems for Section 8.6

Class Problems

Problem 8.13.
Two nonparallel lines in the real plane intersect at a point. Algebraically, this means that the equations

$$\begin{aligned} y &= m_1x + b_1 \\ y &= m_2x + b_2 \end{aligned}$$

have a unique solution (x, y), provided $m_1 \neq m_2$. This statement would be false if we restricted x and y to the integers, since the two lines could cross at a noninteger point:



However, an analogous statement holds if we work over the integers *modulo a prime*, p. Find a solution to the congruences

$$\begin{aligned} y &\equiv m_1 x + b_1 \pmod{p} \\ y &\equiv m_2 x + b_2 \pmod{p} \end{aligned}$$

when $m_1 \not\equiv m_2 \pmod{p}$. Express your solution in the form $x \equiv ? \pmod{p}$ and $y \equiv ? \pmod{p}$ where the ?'s denote expressions involving m_1, m_2, b_1, and b_2. You may find it helpful to solve the original equations over the reals first.

Problem 8.14.
Let $S_k = 1^k + 2^k + \ldots + (p-1)^k$, where p is an odd prime and k is a positive multiple of $p - 1$. Use Fermat's theorem to prove that $S_k \equiv -1 \pmod{p}$.

Homework Problems

Problem 8.15. **(a)** Use the Pulverizer to find the inverse of 13 modulo 23 in the range $\{1, \ldots, 22\}$.

(b) Use Fermat's theorem to find the inverse of 13 modulo 23 in the range $\{1, \ldots, 22\}$.

Problems for Section 8.10

Practice Problems

Problem 8.16. **(a)** Prove that 22^{12001} has a multiplicative inverse modulo 175.

(b) What is the value of $\phi(175)$, where ϕ is Euler's function?

(c) What is the remainder of 22^{12001} divided by 175?

Problem 8.17. **(a)** Use the Pulverizer to find integers s, t such that

$$40s + 7t = \gcd(40, 7).$$

Show your work.

(b) Adjust your answer to part (a) to find an inverse modulo 40 of 7 in the range $\{1, \dots, 39\}$.

Class Problems

Problem 8.18.
Let's try out RSA! There is a complete description of the algorithm at the bottom of the page. You'll probably need extra paper. **Check your work carefully!**

(a) As a team, go through the **beforehand** steps.

- Choose primes p and q to be relatively small, say in the range 10-40. In practice, p and q might contain several hundred digits, but small numbers are easier to handle with pencil and paper.
- Try $e = 3, 5, 7, \dots$ until you find something that works. Use Euclid's algorithm to compute the gcd.
- Find d (using the Pulverizer—see appendix for a reminder on how the Pulverizer works—or Euler's Theorem).

When you're done, put your public key on the board. This lets another team send you a message.

(b) Now send an encrypted message to another team using their public key. Select your message m from the codebook below:

- 2 = Greetings and salutations!
- 3 = Yo, wassup?
- 4 = You guys are slow!
- 5 = All your base are belong to us.
- 6 = Someone on *our* team thinks someone on *your* team is kinda cute.

- 7 = You *are* the weakest link. Goodbye.

(c) Decrypt the message sent to you and verify that you received what the other team sent!

Problem 8.19.
A critical fact about RSA is, of course, that decrypting an encrypted message always gives back the original message! That is, that $\text{rem}((m^d)^e, pq) = m$. This will follow from something slightly more general:

Lemma 8.10.1. *Let n be a product of distinct primes and $a \equiv 1 \pmod{\phi(n)}$ for some nonnegative integer, a. Then*

$$m^a \equiv m \pmod{n}. \tag{8.13}$$

(a) Explain why Lemma 8.10.1 implies that k and k^5 have the same last digit. For example:

$$\underline{2}^5 = 3\underline{2} \qquad 7\underline{9}^5 = 307705639\underline{9}$$

Hint: What is $\phi(10)$?

(b) Explain why Lemma 8.10.1 implies that the original message, m, equals $\text{rem}((m^e)^d, pq)$.

(c) Prove that if p is prime, then

$$m^a \equiv m \pmod{p} \tag{8.14}$$

for all nonnegative integers $a \equiv 1 \pmod{p-1}$.

(d) Prove that if $a \equiv b \pmod{p}_i$ for distinct primes $p_1, p_2, \ldots, p_n$, then $a \equiv b$ $(\text{mod } ()p_1 p_1 \cdots p_n)$.

(e) Combine the previous parts to complete the proof of Lemma 8.10.1.

Homework Problems

Problem 8.20.
Suppose m, n are relatively prime. In the problem you will prove the key property of Euler's function that $\phi(mn) = \phi(m)\phi(n)$.

(a) Prove that if $x \equiv c \pmod{a}b$, then $x \equiv c \pmod{a}$ for all positive a, b.

(b) Prove that for any a, b, there is an x such that

$$x \equiv a \pmod{m}, \tag{8.15}$$
$$x \equiv b \pmod{n}. \tag{8.16}$$

Hint: Let m^{-1} be an inverse of m modulo n and define $e_n ::= m^{-1}m$. Define e_m similarly. Let $x = ae_m + be_n$.

(c) Prove that there is an $x \in [0, mn)$ satisfying (8.15) and (8.16).

(d) Prove that the x satisfying part (c) is unique.

(e) For an integer k, let k^* be the integers in $[1, k)$ that are relatively prime to k. Conclude from part (d) that the function

$$f : (mn)^* \to m^* \times n^*$$

defined by

$$f(x) ::= (\text{rem}(x, m), \text{rem}(x, n))$$

is a bijection.

(f) Conclude from the preceding parts of this problem that

$$\phi(mn) = \phi(m)\phi(n).$$

apparently he added stuff to past sections

3/10

II Structures

Introduction

Structure is fundamental in computer science. Whether you are writing code, solving an optimization problem, or designing a network, you will be dealing with structure. The better you can understand the structure, the better your results will be. And if you can reason about structure, then you will be in a good position to convince others (and yourself) that your results are worthy.

The most important structure in computer science is a *graph*, also known as a *network*). Graphs provide an excellent mechanism for modeling associations between pairs of objects; for example, two exams that cannot be given at the same time, two people that like each other, or two subroutines that can be run independently. In Chapter 9, we study *directed graphs* which model *one-way* relationships such as being bigger than, loving (sadly, it's often not mutual), being a prerequisite for. A highlight is the special case of acyclic digraphs (*DAGs*) that correspond to a class of relations called *partial orders*. Partial orders arise frequently in the study of scheduling and concurrency. Digraphs as models for data communication and routing problems are the topic of Chapter 10.

"Crush on"

In Chapter 11 we focus on *simple graphs* that represent mutual or *symmetric* relationships, such as being congruent modulo 17, being in conflict, being compatible, being independent, being capable of running in parallel.

This part of the text concludes with Chapter 12 which elaborates the use of the *state machines* in program verification and modeling concurrent computation.