An Algorithmic Approach to Algebra

John Perry

November 7, 2018

Contents

Chapter 1

Modular arithmetic

1.1 Sets and relations

The material in this section is fundamental! Much of mathematics depends entirely on the definitions.

Sets

A *set* is a collection of *elements*. If S is a set then we write $s \in S$ to say that "s is an element of S," and we write $t \notin S$ to say that "t is not an element of S." In this text we *usually* use capital letters to indicate a set, and minuscule letters to indicate elements of a set, and an element of a set is often the minuscule letter of the set's capital letter.

If a set S has finitely many elements, then the *size* of a set is the number of elements. We write $|S|$ for the size of S

We often use multiple elements of a set, and employ the following conventions when considering successive elements of a set S.

- If we only consider one or two, we'll start with $s \in S$, then consider $t \in S$. Similarly we may start with $a \in A$, then consider $b \in A$.
- If there's some intuitive relationship between $s \in S$ and the next element we consider, we may place a decoration on s and thus consider it a different element, such as $\hat{s}$, s', and so forth.
- If we take a longer sequence of elements, we'll write the first one as s_0, the next one as s_1, the one after that as s_2, and so forth. We call the number a

subscript and sometimes we'll use a letter when we don't necessarily know *which* element of the sequence we mean; for instance, s_i is the ith element in s_1, s_2, ...

The basic sets of school mathematics are

- the *positive numbers* 1, 2, 3, ..., written $\mathbb{N}^+$ for short;
- the *natural numbers* 0, 1, 2, ..., written $\mathbb{N}$ for short;
- the *integers* ..., -2, -1, 0, 1, 2, ..., written $\mathbb{Z}$ for short;
- the *rational numbers* a/b where a, b are integers and $b \neq 0$, written $\mathbb{Q}$ for short;
- the *real numbers*, which we can describe intuitively as "any length of a line segment," written $\mathbb{R}$ for short.[1]

Whenever every element a set S is also an element of a set T, we say that S is a *subset of* T, written $S \subseteq T$ for short. We say that two sets are *equal* if each is a subset of the other, written $S = T$ for short. If we know that $S \subseteq T$ but either don't know or don't care whether $S = T$, we write $S \subsetneq T$.

Example 1. Every natural number is an integer, allowing us to write $\mathbb{N} \subseteq \mathbb{Z}$. The negatives are not natural, so $\mathbb{Z} \subsetneq \mathbb{N}$ and so $\mathbb{N} \neq \mathbb{Z}$.

Example 2. Every rational number of the form $a/1$ is in identified with the integer a, so every integer is a rational number, allowing us to write $\mathbb{Z} \subseteq \mathbb{Q}$. Some rationals, like $1/2$, are not integers, so $\mathbb{Q} \subsetneq \mathbb{Z}$ and so $\mathbb{Q} \neq \mathbb{R}$.

Example 3. Every rational number corresponds to the length of a line, allowing us to write $\mathbb{Q} \subseteq \mathbb{R}$.

Example 4. Is every rational number a real number? This question is said to have vexed the Pythagoreans. Certainly $\sqrt{2}$ would be a real number, as we can show that $\sqrt{2}$ is the length of a line segment (see the exercises). But can we write $\sqrt{2}$ as a/b, where a and b are integers? If we can do this for every real number, then we could write $\mathbb{R} \subseteq \mathbb{Q}$. We postpone the resolution of this question until Section 1.4.

[1]This is not really important, but $\mathbb{Z}$ it from the German word for "number" and $\mathbb{Q}$ is from the Italian word for "quotient."

Sets are often written using braces and a notation called *set-builder notation*. For instance, we could have written the definition of $\mathbb{Q}$ above as

$$\mathbb{Q} = \{{}^{a}\!/\!{}_{b} : a, b \in \mathbb{Z} \text{ and } b \neq 0\} \ .$$

We would read this as, "$\mathbb{Q}$ is the set of all ${}^{a}\!/\!{}_{b}$ such that a and b are integers and b is not 0."

There are three common ways to build one set from two others.

- The *union* of S and T is the set of elements that are members of S or T,[2] written $S \cup T$. In set-builder notation,

$$S \cup T = \{x : x \in S \text{ or } x \in T\} \ .$$

- The *intersection* of S and T is the set of elements that are members of S and T, written $S \cap T$. In set-builder notation,

$$S \cap T = \{x : x \in S \text{ and } x \in T\} \ .$$

- The *difference* of S and T is the set of elements that are members of S but not of T,[3] written $S \backslash T$. In set-builder notation,

$$S \backslash T = \{x : x \in S \text{ and } x \notin T\} \ .$$

Relations

From here until the end of the section, S and T are sets.

The *Cartesian product* of S and T is the set of all ordered pairs whose first entry is an element of S and whose second entry is an element of T. Written symbolically,

$$S \times T = \{(s, t) : s \in S, t \in T\} \ .$$

[2]When we say "or" in common English we typically mean "either-or," or "exclusive-or." That is, "You can have your cake or you can eat it, but you can't both have your cake and eat it." When we say "or" in mathematics, however, we always mean "inclusive-or." A mathematician understands that the correct answer to, "Would you like cake or pie?" is "Yes."

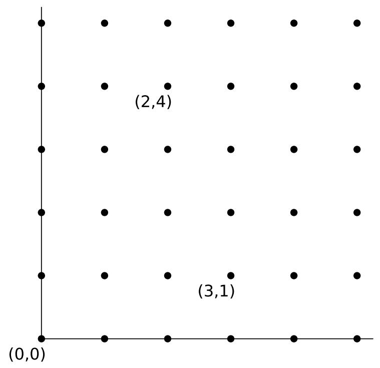
[3]As the set-builder notation shows, "but" and "and" have the same logical meaning in mathematics, and one can usually interchange them. In common English, "but" and "and" are not typically interchangeable.

Example 5. The Cartesian product of $\mathbb{N}$ with itself is

$$\mathbb{N} \times \mathbb{N} = \{(0,0), (0,1), (0,2), \ldots, (1,0), (1,1), (1,2), \ldots\} \ .$$

In a case like this we will write $\mathbb{N}^2$ instead of $\mathbb{N} \times \mathbb{N}$. We call $\mathbb{N}^2$ the *lattice* of natural numbers.

An interesting aspect of the lattice of natural numbers is that you can plot its elements rather easily using the first quadrant of the real plane, typically drawn on a grid:



The only points allowed on the lattice are the ones marked with dots. Unlike an ordinary graph, *no points lie between them!* We have diagrammed the lattice points $(0,0)$, $(2,4)$, and $(3,1)$.

A *relation* R *between* S *and* T is a subset of $S \times T$. Given a relation R, we say that $s \in S$ and $t \in T$ are related if $(s,t) \in R$. However, it is more common to write relations differently: we typically use symbols such as $\leq$, $\sim$, $\equiv$, and write $s \leq t$ or $s \sim t$ or $s \equiv t$.

If $S = T$ then we call R a *relation on* S.

Example 6. Consider the relation on $\mathbb{N}$

$$R = \left\{(a,b) \in \mathbb{N}^2 : b - a \in \mathbb{N}^+\right\} \ .$$

Elements of this set include

$$(0,1), \quad (0,2), \quad (2,4), \quad (8,15)$$

but *not* elements of the form

$$(0,0), \quad (2,0), \quad (-3,7) \ .$$

You are more likely to think of this as the *less-than* relation:

$$0 < 1 \quad 0 < 2 \quad \ldots \quad 1 < 2 \quad 2 < 3 \quad \ldots \quad 2 < 3 \quad 3 < 4 \quad \ldots .$$

Example 7. Here's another relation on $\mathbb{N}$:

$$R = \{(0,0),(1,1),(2,2),\ldots\} .$$

What familiar relation are you looking at?

Many relations belong to an important class of relations called *equivalence relations*. Equivalence relations satisfy three important properties. To describe them, we need a set S and a relation $\sim$ on S.

- The *reflexive* property states that $s \sim s$ for every $s \in S$.
- The *symmetric* property states that for every $s, t, u \in S$ if $s \sim t$ and $t \sim u$, then $s \sim u$.
- The *transitive* property states that for every $s, t, u \in S$ if $s \sim t$ and $t \sim u$, then $s \sim u$.

Example 8. Look back at Example 6. It does *not* satisfy the reflexive property, because $0 \not< 0$, or, $(0,0) \notin R$. It is therefore not an equivalence relation. It also does not satisfy the symmetric property, because $0 < 1$ but $1 \not< 0$, or, $(0,1) \in R$ but $(1,0) \notin R$. On the other hand, it does satisfy the transitive property, because if $a < b$ and $b < c$ then $a < c$, or, if $(a,b),(b,c) \in R$ then $(a,c) \in R$.

Example 9. Look back at Example 7. You hopefully noticed that R is really the equality relation: every element of R has the form (a,b) where $a = b$. We rely on this to show that R is an equivalence relation.

- For every $a \in \mathbb{N}$, we see that $(a,a) \in R$, so R is symmetric.
- For every $a, b \in \mathbb{N}$, we see that if $(a,b) \in R$, then $a = b$, in which case $b = a$, so $(b,a) \in R$.
- For every $a, b, c \in \mathbb{N}$, if $(a,b),(b,c) \in R$, then $a = b$ and $b = c$, so $a = c$, in which case $(a,c) \in R$.

Exercises

Exercise 10. For the sets $S = \{1,2,3\}$ and $T = \{2,3,4\}$, compute the following sets.

(a) $S \cup T$

(b) $S \cap T$

(c) $S \backslash T$

(d) $S \times T$

Exercise 11. Use the intuitive definition of real number as the "length of a line segment" to explain how we know that $3/8$ is a real number. Be as specific as possible; if you know the geometric constructions with ruler and compass, describe how to construct $3/8$.

Exercise 12. Define a relation on $\mathbb{Q}$ in the following way. For any $a/b, c/d \in \mathbb{Q}$, we say that $a/b \sim c/d$ if $ad = bc$.

(a) Show that $4/6 \sim 6/9$.

(b) Show that $-2/5 \sim 10/-25$.

(c) Show that $a/b \sim a/b$.

(d) Show that if $a/b \sim c/d$, then $c/d \sim a/b$.

(e) Show that if $a/b \sim c/d$ and $c/d \sim e/f$, then $a/b \sim e/f$.

(f) Is $\sim$ an equivalence relation?

Remember to use the *definition* of the relation in each part! If you aren't rewriting fractions as multiplication, then you aren't doing it right!

Exercise 13. The *Euclidean distance* between $(a,b), (c,d) \in \mathbb{N}^2$ is the value determined by the ordinary distance formula, $\sqrt{(a-c)^2 + (b-d)^2}$. This is not a natural number. It is possible to compute distance a different way, so that it is a natural number. In this case we consider the distance from (a,b) to (c,d) to be $|(a-c)| + |(b-d)|$. We'll call this the *sidewalk* distance between two points, because it indicates the number of sidewalks you'd have to travel when walking through a city laid out with perpendicular streets from point (a,b) to point (c,d).

(a) Compute the sidewalk distance between $(0,0)$ and $(3,4)$.

(b) Compute the sidewalk distance between $(1,5)$ and $(7,2)$.

(c) Explain why we can find the sidewalk distance between (a,b) and (c,d) by tracing north-south and east-west lines from (a,b) to (c,d), and determining the shortest path.

Sage supplement

This section shows how to perform some elementary operations in Sage.

Some sets are already defined in Sage. For instance, you can type `NN`, `ZZ`, and `QQ` to obtain the sets $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$. If you type them into Sage, it will display a funny name:

```
sage: NN
Non negative integer semiring
sage: ZZ
Integer Ring
sage: QQ
Rational Field
```

Don't worry too much about the names; we explain later what the names "ring" and "field" mean.

You can define a set using either braces `{}` or the `set()` command.

```
sage: { 3, 5, 7 }
set([3, 5, 7])
sage: set( [ 7, 5, 3, 7, 7, 3, 5 ] )
set([3, 5, 7])
```

Notice how elements are automatically ordered, and no element can appear more than twice.

It is also possible to define a set using something akin to set-builder notation, making use of `for` and `if` statements:

```
sage: { i^2 for i in { 3, 5, 7 } }
set([9, 49, 25])
sage: { i^2 for i in { 3, 4, 5, 6, 7 } if is_even(
      i ) }
set([16, 36])
```

Notice how all three numbers were squared in the first assignment, and how only the even numbers were taken and squared in the second assignment.

Another useful command for generating a set is `range()`.

- With just one integer between the parentheses, it returns a list of all the numbers that are between 0 and the specified number, including 0 but not the specified number.
- With two integers between the parentheses, it returns a list of all the numbers between the two, including the first but not the last.

```
sage: range( 8 )
[0, 1, 2, 3, 4, 5, 6, 7]
sage: range( 3, 8 )
[3, 4, 5, 6, 7]
```

You can assign a name to a set using the = operator. You can assign a name to any object in this way. You can even assign several objects at a time.

```
sage: a, b = 3, 4
sage: a + b
7
sage: S = { a, b, 7, 3 }
sage: S
set([3, 4, 7])
```

Notice that Sage does not display any messages after a successful assignment.

You can test whether two objects are equal using the == operator. This is *not* the same as the = operator; comparison uses two equality signs instead of one. Be careful when doing this; if you type only one sign, you may accidentally overwrite an object. In other cases, Sage will report an error. What happens depends on the context.

```
sage: 3 + 3 == 6
True
sage: 3 + 3 = 6
Error in lines 1-1
Traceback (most recent call last):
  File "/cocalc/lib/python2.7/site-packages/
smc_sagews/sage_server.py", line 1188, in execute
flags=compile_flags) in namespace, locals
  File "<string>", line 1
SyntaxError:  can't assign to operator
```

Simple set operations are possible: use "dot commands" to accomplish them.

```
sage: S = { i^2 for i in range(20) if is_even(i) }
sage: T = { 4*i for i in range(100) }
sage: S.intersection(T)
set([0, 64, 4, 16, 256, 144, 196, 324, 36, 100])
sage: S.union(T)
set([0, 256, 392, 4, 8, ...  252])
sage: S.difference(T)
set([])
```

That last output is how Sage indicates that it has computed an empty set.

You can often discover commands available for an object by typing the object's name, adding a period, then pressing the "tab" key. If you do this with `S`, the version of Sage I am using will return 17 commands:

`add`, `clear`, `copy`, `difference`, `difference_update`, `discard`, `intersection`, `intersection_update`, `isdisjoint`, `issubset`, `issuperset`, `pop`, `remove`, `symmetric_difference`, `symmetric_difference_update`, `union`, `update`

You should see commands for union, intersection, and difference. To learn more about a command, you can continue by typing the command after the period, followed a question mark, then executing the line, Sage will give you some help on the command. If I do this with pop, the version of Sage I am using will display the following:

```
sage: S.pop?
File:
Docstring :  Remove and return an arbitrary set
element.  Raises KeyError if the set is empty.
```

This gives you an idea of what happens when you pop an element from a set.

You may have noticed that set objects lack a dot command for a Cartesian product. A command to compute Cartesian products actually exists; it just isn't a dot command. Try this:

```
sage: CP = cartesian_product((S,T))
sage: CP
The Cartesian product of ({0, 64, 4, 100, 324,
144, 256, 16, 196, 36}, {0, 256, 4, 8, ...  136,
252})
```

Notice that we used double parentheses; the cartesian_product command expects as input *one* argument, which is a pair or list of sets. Here we used parentheses to give a pair.

It may not seem especially useful to have a "Cartesian product" that displays itself only as a "Cartesian product" and not as a set of points, but trust us when we say that it is *very* useful.[4] In any case, we can use a set builder to transform CP into a set of ordered pairs: (that's an x between the S and the T below)

```
sage: SxT = { P for P in CP }
sage: SxT
set([(100, 36), (100, 324), (4, 264), ...  (16,
328), (36, 76), (324, 352)])
```

[4]Explaining why is beyond the scope of these notes.

(We omitted a *lot* of output this time.)

It is possible to define new commands in Sage. This is not a textbook on programming; we assume that if you are reading this, then you have some experience with programming, so we won't delve into the details of how various control structures work, but the following will define a command that computes the sidewalk distance between two points of the lattice, described in Exercise 13.

```
sage: def sidewalk_dist(P, Q):
          return abs(P[0] - Q[0]) + abs(P[1] - Q[1])
```

The `def` keyword defines a new command, in this case named `sidewalk_dist`. Parentheses always follow, and contain arguments that `sidewalk_dist` requires. In this case, it requires two arguments, `P` and `Q`.[5] The first line ends with a colon, and subsequent lines are indented; these two signals indicate that the indented lines depend on the line that ends with a colon. You will see this in all Sage's control structures.

The procedure then computes $|p_0 - q_0| + |p_1 - q_1|$, where p_0 is the first entry of P and p_1 is its second entry; we use `abs(...)` to compute the absolute value of whatever is in parentheses. The `return` command indicates that whatever follows on that line is the result of `sidewalk_dist`.

Once we define the command, we can use it as follows.

```
sage: sidewalk_dist((3,5),(7,2))
7
```

What just happened inside the computer? First it assigned the values $(3, 5)$ to `P` and $(7, 2)$ to `Q`. It then computed

- `abs( P[0] - Q[0] )` $= |3 - 7| = 4$;
- `abs( P[1] - Q[1] )` $= |5 - 2| = 3$;
- the sum of these numbers, 7;

and then returned the result.

[5] Sage is like Python 2 or Perl, and unlike Python 3, in that it does not allow you to specify an argument's type of an argument. It is very much unlike C or Java, where you *must* specify the type.

Exercises

Exercise 14. Use Sage to verify your answers in Exercise 10.

Exercise 15. Use Sage to verify your answers to Exercise 13.

1.2 Integer division

This class will make heavy use of integer division, so it is important to develop a solid understanding of this topic.

A *function from a set* S *to a set* T is a relation F between S and T such that if $(a,b) \in F$ then $(a,c) \in F$ only if $b = c$. That is, each "input" to F can have only one "output". It is common to write $F(a) = b$ instead of $(a,b) \in F$, so we will do that from now on.

An *operation on a set* S is a function from S^2 to S. It is more common to give an operation a symbol, such as $\diamond$, so that instead of saying $((s,t),u)$ is in the operation, we say $s \diamond t = u$, so we will do that from now on.

An operation is *closed* if for every $s,t \in S$ there exists $u \in S$ such that $s \diamond t = u$.

Example 16.

- Addition is an operation on $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$. In fact, it is closed on all four sets.
- Subtraction is an operation on $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$. However, closed only on the latter three; it is not closed on $\mathbb{N}$, since $3 - 4 \notin \mathbb{N}$.
- Multiplication is an operation on $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$. In fact, it is closed on all four sets.
- Division, however, is a little odd. You will probably believe that it is an operation on $\mathbb{Q}$ and $\mathbb{R}$; after all, you've done it many times. However, we show below that division does not satisfy our definition of an operation in $\mathbb{N}$ and $\mathbb{Z}$. On these sets it's a strange little creature.

A Division Algorithm

What is division, and how do we accomplish it? The basic idea is that, given a set of n distinct objects, we would like to divide it into subsets of d distinct objects.

We may not be able to do this perfectly, in which case we'll be greedy and take as many subsets as we can, and identify the number of objects left over as the *remainder*.

If division were an operation on $\mathbb{N}$, we would map from $\mathbb{N}^2$ to $\mathbb{N}$. Instead, division maps from $\mathbb{N}^2$ (the dividend n and the divisor d) to $\mathbb{N}^2$ (the quotient q and the remainder r). So division is merely a function on $\mathbb{N}^2$ and not an operation on $\mathbb{N}$!

Example 17. Given a set of 51 elements, we can divide it into 8 sets of 6 elements, with a remainder of 3.

How can we carry out division? One rather simplistic way is via the following algorithm.

Algorithm 1.1 Simplistic Division Algorithm

```
input
    • $n \in \mathbb{N}$
    • $d \in \mathbb{N}^+$
output
    • $q, r \in \mathbb{N}$ such that $n = qd + r$ and $r < d$
do
    1. let $r = n$, $q = 0$
    2. while $r \geq d$
        (a) increment $q$ by 1
        (b) decrement $r$ by $d$
    3. return $q$ and $r$
```

Let's see how this algorithm produces the result of the previous example.

Example 18. We want to divide $n = 51$ by $d = 6$. Step 1 of the algorithm assigns $r = 51$ and $q = 0$.

We proceed to step 2. A "while" statement means that as long as the condition is true, we perform the steps indented underneath it. Since $r = 51$ and $d = 6$, we have $r \geq d$. In step 2(a), we increment q by 1, obtaining $q = 1$. In step 2(b), we decrement r by d, obtaining $r = 45$.

We remain in step 2 because $r \geq d$. Increment q to 2 and decrement r to 39.

We remain in step 2 because $r \geq d$. Increment q to 3 and decrement r to 33.

$\ldots$

The algorithm continues until q rises to 8 and r falls to 3. At this point, $r < d$, so the "while" statement's condition is false, and the algorithm ends.

Whenever we describe a new algorithm, we have to be sure of two important properties.

1. Termination: The algorithm eventually produces *some* result.
2. Correctness: The algorithm's result is the *claimed* result.

Unfortunately, we do not yet have enough theory to explain why Algorithm 1.1 terminates correctly. We need to consider the natural numbers a little more carefully.

The Well-Ordering Property and some of its consequences

You are familiar with the natural ordering of numbers; in Example 6 we saw that for any $a, b \in \mathbb{N}$

$$a < b \quad \text{if and only if} \quad b - a \in \mathbb{N}^+ .$$

This characterization works if $a, b \in \mathbb{Z}$, as well.

An interesting property of $\mathbb{Z}$ is that it has no smallest element. After all, for any $z \in \mathbb{Z}$ we know that $z - 1 \in \mathbb{Z}$ and then

$$z - (z - 1) = 1 \in \mathbb{N}^+ \quad \text{so} \quad z - 1 < z .$$

What about $\mathbb{Z}$'s subsets? Many of its sets do not have a smallest element. Consider $S = \{-3, -4, \ldots\}$; the same argument we applied to $\mathbb{Z}$ applies to S.

On the other hand, $\mathbb{N}$ has 0 as a smallest element: for any nonzero $n \in \mathbb{N}$, we have $n - 0 = n \in \mathbb{N}^+$, so $0 < n$. Let's highlight this as an important and useful fact.

Lemma 19. *Under the natural ordering, zero is the smallest element of* $\mathbb{N}$.

What about $\mathbb{N}$'s subsets? Intuitively, this seems to be true, but it is not so easy to prove this deductively. In fact, without assuming anything at all, *one cannot prove that all of $\mathbb{N}$'s subsets have smallest elements.* So we take it "on faith" to be true.[6]

Axiom (The Well-Ordering Property)**.** *Every subset of $\mathbb{N}$ has a least element.*

The Well-Ordering Property gives us a very useful technique that we will use repeatedly. A non-increasing sequence of negative numbers like

$$-5, -9, -12, -14, \ldots$$

might stabilize eventually, but it might not. Without being able to check every element of the sequence, we can't say. But a sequence of natural numbers is different; if you see

$$14, 12, 9, 5, \ldots$$

then you feel fairly confident that the sequence will in fact stabilize.

Theorem 20. *Every non-increasing sequence of natural numbers a_1, a_2, … eventually stabilizes at a least element.*

Proof. Let a_1, a_2, … be a non-increasing sequence of natural numbers. Let $A = \{a_1, a_2, \ldots\}$. Every $a_i \in \mathbb{N}$, so $A \subseteq \mathbb{N}$. By the Well-Ordering property, A has a least element; call it $\hat{a}$. By definition of A, there exists i such that $\hat{a} = a_i$.

We claim that the sequence stabilizes at a_i. By hypothesis, the sequence is non-increasing, so $a_i \geq a_{i+1} \geq a_{i+2} \geq \cdots$. By the transitive property, $a_i \geq a_j$ for every $j \leq i$. On the other hand, a_i is the least element of A, and by definition of A $a_j \in A$ for every $j \geq i$, so we also have $a_i \leq a_j$. Now, if

$$a_i \geq a_j \quad \text{and} \quad a_i \leq a_j$$

then in fact

$$a_i = a_j \ .$$

(You will prove this in Exercise 24.) This is true for all the $j \geq i$, so the sequence has $a_i = a_{i+1} = a_{i+2} = \cdots$. In other words, the sequence has stabilized at a_i. □

[6]This is something we try to avoid in mathematics if at all possible, but in some cases it's unavoidable. This is one of those cases.

Theorem 20 gives us the information we need to prove that our simplistic division algorithm both produces output a result and produces the claimed result.

Corollary 21. *Algorithm 1.1 terminates correctly.*

Proof. First we show that the algorithm terminates. If the algorithm does not execute step 2 at all, it certainly terminates, so suppose it continues to step 2. Enumerate each value of r computed in step 2 as r_1, r_2, and so forth. By definition, $r_{i+1} = r_i - d$. Rewrite this as $r_i - r_{i+1} = d$; since $d \in \mathbb{N}^+$, we have $r_i > r_{i+1}$. The sequence of r's is thus a non-increasing sequence, and by Theorem 20 it must stabilize eventually at a least element, say r_k. If $r_k \geq d$, the algorithm would create a smaller r_{k+1}, contradicting our observation that the sequence has a least element. Hence $r_k < d$, in which case the algorithm has terminated.

Now we show that the algorithm's final q and r are correct. As before, enumerate the r's of step 2 as $r_1, r_2, \ldots, r_k$, and put $r_0 = n$. This means the algorithm repeated step 2 k times. We consider the criteria slightly out of order.

- Is $0 \leq r < d$?
  - Certainly $r < d$; otherwise, the algorithm wouldn't have terminated.
  - If $r \notin \mathbb{N}$, then $r < 0$. Since its initial value in step 1 is $r_0 = n$, a natural number, the algorithm must have performed step 2 at least once. In particular, it performed step 2 on r_{k-1}, so

    $$r_k = r_{k-1} = d \quad \Longrightarrow \quad r_{k-1} = r_k + d = r + d < 0 + d = d\,.$$

    That is, $r_{k-1} < d$. But the fact that the algorithm performed step 2 on r_{k-1} means that $r_{k-1} \geq d$, a contradiction. So $0 \leq r < d$, as claimed.
- Is $n = qd + r$?
  - If the algorithm does not perform step 2 at all, then
    - $r = n$ and $q = 0$, so $qd + r = 0 \times d + n = n$, satisfying the claim that $qd \leq n$; and
    - since the algorithm did not perform step 2, we must have $r < d$, and $r = n \in \mathbb{N}$ implies that $0 \leq r$.

- Suppose that the algorithm continues to step 2. Step 2(a) increments q by 1 each time, so $q = k$. Also observe that

$$\begin{aligned} r_1 &= n - d \\ r_2 &= r_1 - d = n - 2d \\ &\vdots \\ r_k &= n - kd\,. \end{aligned}$$

By substitution,

$$qd + r = kd + (n - kd) = n.$$

We have shown that $n = qd + r$ and $0 \leq r < d$, so the proof is complete. □

The Division Theorem

Algorithm 1.1 applies to natural numbers only. We can extend this fairly easily to all integers.

Theorem 22 (The Division Theorem)**.** *Let $n, d \in \mathbb{Z}$ with $d \neq 0$. There exist $q, r \in \mathbb{Z}$ such that*

- *$n = qd + r$, and*
- *$0 \leq r < |d|$.*

In addition, q and r are uniquely determined by n and d.

Proof. If $n, d \in \mathbb{N}$, then Corollary 21 proves the result. Otherwise, at least one of $n, d < 0$. We consider this in three cases. For each case we consider an example.

Case 1. Suppose $n < 0$ but $d > 0$.

Example. Suppose $n = -51$ and $d = 6$. Algorithm 1.1 requires non-negative numbers, so what happens if we consider 51 and 6? We get $q = 8$ and $r = 3$. We have $51 = 8 \times 6 + 3$, but we need an expression for -51 rather than 51. If we multiply both sides by -1, we have $-51 = -(8 \times 6 + 3) = (-8) \times 6 + (-3)$. The theorem allows $q < 0$, but not $r < 0$. Can we fix this somehow?

We can, and here's how: add $0 = 6 + (-6)$ on the right hand side. We have $-51 = (-8) \times 6 + 3 + [6 + (-6)]$, which we can rewrite as $-51 = (-8 - 1) \times 6 + (-3 + 6) = -9 \times 6 + 3$. Now we can set $q = -9$ and $r = 3$ and they satisfy the theorem!

This insight allows us to prove the theorem. If $n < 0$, then $-n \in \mathbb{N}$. Divide that by d; Corollary 21 tells us that Algorithm 1.1 will give us $\hat{q}, \hat{r} \in \mathbb{N}$ such that $-n = \hat{q}d + \hat{r}$ and $\hat{r} < d$. Multiply both sides by -1 and we have

$$n = -\left(\hat{q}d + \hat{r}\right) = \left(-\hat{q}\right)d + \left(-\hat{r}\right) \ .$$

If we set $q = -\hat{q}$ and $r = \hat{r}$, then we have $n = qd + r$, but $r \leq 0$. This will work in the theorem only if $\hat{r} = 0$. Otherwise, we try the workaround of the example: let $q = -\hat{q} - 1$ and $r = d - \hat{r}$. By substitution,

$$\begin{aligned} qd + r &= \left(-\hat{q} - 1\right) \times d + \left(d - \hat{r}\right) \\ &= \left(-\hat{q}d - d\right) + \left(d - \hat{r}\right) \\ &= -\hat{q}d - \hat{r} \\ &= -\left(\hat{q}d + \hat{r}\right) \\ &= -\left(-n\right) \\ &= n \ . \end{aligned}$$

So $n = qd + r$, satisfying the first requirement.

As for the second, if $\hat{r} \neq 0$, then $\hat{r} \in \mathbb{N}^+$, so $0 < \hat{r} < d$. Multiply through by -1 to obtain $0 > -\hat{r} > -d$. Add d to every item to obtain $0 + d > -\hat{r} + d > d + (-d)$, or $d > d - \hat{r} > 0$. Recall that $r = d - \hat{r}$, so $r \in \mathbb{N}$ and $r < d$, satisfying the second requirement.

Case 2. Suppose $n > 0$ but $d < 0$.

Example. Suppose $n = 51$ and $d = -6$. Algorithm 1.1 requires non-negative numbers, so what happens if we consider 51 and 6? We get $d = 8$ and $r = 3$. We have $51 = 8 \times 6 + 3$, but we need an expression for -6 rather than 6. Instead of multiplying both sides by -1, however, we notice that $51 = (-8) \times (-6) + 3$. Now we can set $q = -8$ and $r = 3$ and satisfy the theorem!

We leave the generalization of this example to a proof as an exercise for the reader.

Case 3. Suppose both $n, d < 0$.

> **Example.** We leave the creation of an example as an exercise for the reader.
>
> We leave the generalization of this example to a proof as an exercise for the reader.

The three cases listed cover all possibilities; in each case we can find $q, r \in \mathbb{Z}$ such that $n = qd + r$ and $0 \leq r < d$, proving the existence of q and r.

We still have to show that q and r are unique. To that end, suppose there exist $q, \hat{q}, r, \hat{r} \in \mathbb{Z}$ such that $n = qd + r$ and $n = \hat{q}d + \hat{r}$ and $0 \leq r, \hat{r} < |d|$. By substitution, $qd + r = \hat{q}d + \hat{r}$. Rewrite this equation as $(q - \hat{q})\, d = \hat{r} - r$. Since d divides the left hand side, it also divides the right. On the other hand, we can rewrite $0 \leq r, \hat{r} < |d|$ as $0 - |d| < r - \hat{r} < |d| - 0$, or $-|d| < r - \hat{r} < |d|$. Recall that d divides $r - \hat{r}$; the only multiple of d that lies between $-|d|$ and $|d|$ is 0, so $r - \hat{r} = 0$, or $r = \hat{r}$. Substitute into $(q - \hat{q})\, d = \hat{r} - r$ to see that $(q - \hat{q})\, d = 0$. This is possible only if $q - \hat{q} = 0$ or $d = 0$. By the theorem's hypothesis, $d \neq 0$, so we must have $q - \hat{q} = 0$, or $q = \hat{q}$, showing that there is only one possible choice for q and r to satisfy the theorem. □

Exercises

Exercise 23. Show that the set $S = \{-3, -6, -9, \ldots\}$ has no smallest element.

Exercise 24. Show that for any natural numbers a and b, if $a \leq b$ and $b \leq a$ then $a = b$.
Hint: Use the fact that $a \leq b$ implies $b - a \in \mathbb{N}$, and $b \leq a$ implies $a - b \in \mathbb{N}$. If both $b - a$ and its opposite are natural numbers, what does that tell you about $b - a$?

Exercise 25. Generalize the example for Case 2 of the proof of Theorem 20 to a proof for arbitrary $n > 0$ and $d < 0$.

Exercise 26. Suppose $n, d < 0$.

(a) Explain how you can rewrite the expression $51 = 8 \times 6 + 3$ to find q, r such that $-51 = q \times (-6) + r$ and $0 \leq r < 6$.

(b) Generalize your work in part (a) to a proof for arbitrary $n < 0$ and $d < 0$.

Exercise 27. The Well-Ordering Property is not true for the rational numbers. One reason is that $\mathbb{Z} \subseteq \mathbb{Q}$, so if every subset of $\mathbb{Q}$ had a least element, then $\mathbb{Z}$ would, too, but it does not.

However, the Well-Ordering Property fails even if we consider only nonnegative rational numbers. To see why, describe a sequence of rational numbers that does not have a least element. Try to choose a sequence whose elements are all decreasing and never stabilizes. Be sure to prove that the elements really are decreasing.

Hint: To show the elements are decreasing, it might help to consider that $a/b < c/d$ if and only if $ad < bc$.

Exercise 28. We have only considered the natural ordering of integers, but there are other ways to order them. For instance, define the relation

$$a \lessdot b \quad \text{if and only if} \quad \begin{cases} |a| < |b|\,, \text{or} \\ |a| = |b| \text{ and } a < b. \end{cases}$$

(Remember that when we write $a < b$ with no dot, we mean the natural ordering.)

(a) Order the integers -5, -3, -1, 2, 4, 9 according to $\lessdot$.

(b) Explain why $\mathbb{Z}$ has a smallest element according to the $\lessdot$ ordering. (It will help to name it explicitly.)

(c) Show that every subset of $\mathbb{Z}$ has a smallest element according to the $\lessdot$ ordering.

In other words, $\mathbb{Z}$ satisfies the well-ordering property if you use the $\lessdot$ ordering!

Exercise 29. Another way to prove the Division Theorem, albeit less algorithmically, is as follows. Fill in the blanks to complete the proof.

- Let $n, d \in \mathbb{Z}$ and assume $d \neq 0$. Let $S = \{n - qd : q \in \mathbb{Z}\}$. Let $T = S \cap \mathbb{N}$.
- By ____, T has a least element; call it r.
- By ____, $r = n - qd$ for some $q \in \mathbb{Z}$.
- Rewrite to obtain ____, satisfying the theorem's first criterion.
- It remains to show that $0 \le r < d$.

- By ____, $r \in \mathbb{N}$.
- By ____, $0 \leq r$.
- By way of contradiction, assume $d \leq r$.
  - Rewrite to obtain $0 \leq$ ____.
  - By ____, $r - d \in T$.
  - On the other hand, $r - d < r$ because ____ $<$ ____.
  - This contradicts ____.
  - Hence, $r < d$.

Exercise 30. Recall the lattice of natural numbers. Suppose we order its elements in the following way: for any $(a,b), (c,d) \in \mathbb{N}^2$ we have

$$(a,b) \prec (c,d) \qquad \text{if and only if} \qquad \begin{cases} a+b < c+d, \text{ or} \\ a+b = c+d \text{ and } a < b \,. \end{cases}$$

(a) Order the points $(3,7)$, $(2,5)$, $(4,1)$, $(0,0)$, $(4,6)$, $(3,2)$ according to $\prec$.

(b) Explain why $\mathbb{N}^2$ has a smallest element according to the $\prec$ ordering. (It will help to name it explicitly.)

(c) Show that every subset of $\mathbb{N}^2$ has a smallest element according to the $\prec$ ordering.

In other words, $\mathbb{N}^2$ satisfies the well-ordering property if you use the $\prec$ ordering!

Sage supplement

You can divide integers using the / operator in Sage, but that gives you a rational number.

```
sage: 7 / 3
7/3
sage: type( _ )
<type 'sage.rings.rational.Rational'>
```

The `type` command gives us the type of an object in Sage; here we get a long string that, for all practical purposes, means that the result of `2 / 3` is something that Sage considers a rational number.

This works if you want rational division, but what if we're interested in integer division, as we were in this section? Sage offers a different command for that, `.quo_rem`. From the dot that precedes the command you would be right to conclude that it is a dot command, and is used accordingly.

```
sage: 7.quo_rem(3)
(2, 1)
```

This indicates that the quotient is 2 and the remainder is 1. You can assign names to these values if you like.

```
sage: q, r = 7.quo_rem(3)
```

After this, `q` and `r` would have the values 2 and 1, respectively.

This command is perfectly fine, but to illustrate some more aspects of Sage, we define a new command that implements the simplistic division algorithm (Algorithm 1.1).

```
sage: def simplistic_division(n, d):
          r, q = n, 0
          while r >= d:
             q += 1
             r -= d
          return q, r
```

Before trying it out, let's consider what it should do when we execute it. First, you should compare it to Algorithm 1.1 and verify that it looks extremely similar to it. Next, observe the use of keywords you already know: `def` and `return`. As for the lines themselves:

- The first line defines the function and ends with a colon. Subsequent lines are indented.

- The second line assigns the values `n` and `0` to `r` and `q`, as in the first line of Algorithm 1.1's instructions.
- The third line begins a repetition of statements, called a *loop*. Sage offers several kinds of loops; this one is called a `while` loop, and performs the indented statements only if, and as long as, the stated condition remains true. Here, the condition is that $r \geq d$.
- The fourth line is the first line repeated in the loop. The `+=` symbol is an operator, and it tells Sage to increment the value before it by the value after it. In this case, it increments `q` by `1`.
- The fifth line is the second line repeated in the loop. The `-=` symbol behaves just like the `+=` symbol, except it decrements the value before it. In this case, it decrements `r` by `d`.
- The sixth line is indented but not as much as the ones before. It lines up with the `while` statement. That means that it should be the first command performed after the `while` statement terminates. In this case, it is a `return` statement, so it indicates that the procedure `simplistic_division` should terminate and give the result `q,r`.

One aspect of Sage that it shares with its roots in Python is that you can return multiple values from a procedure. Here; we return `q` and `r`.

Let's go ahead and try this.

```
sage: simplistic_division(7, 3)
(2, 1)
```

We end up with the same result as the `quo_rem` command.

The `simplistic_division` command also reveals the importance of the Well-Ordering Property. If we try to execute it with numbers that are not natural, strange things will result. For instance:

```
sage: simplistic_division(-7, 3)
(0, -7)
```

Here the result is $q = 0$ and $r = -7$. This is incorrect according to our definition of division, because we require $r \geq 0$, but it is true that $-7 = 0 \times 3 + (-7)$.

That said, things can get worse. If you try the following, an "infinite loop" will result. The `while` loop's condition never becomes false, because we initialize $r = 7$ and every time we decrement it by $d = -3$ the value of r *increases,* first to 10, then to 13, then to 16, and so forth. You can force the command to stop either by holding `control` and pressing `C` (if you're using Sage via a command line terminal), or by pushing the `Stop` button (if you're using it via a graphical interface). You will then encounter an error message similar to the one shown. Go ahead and try it.

```
sage: simplistic_division(7, -3)
Error in lines 1-1
Traceback (most recent call last):
  File "/cocalc/lib/python2.7/site-packages/
smc_sagews/sage_server.py", line 1188, in execute
flags=compile_flags) in namespace, locals
  File "", line 1, in <module> File "", line 4, in
simplistic_division
  File "src/cysignals/signals.pyx", line 265, in
cysignals.signals.python_check_interrupt
  File "src/cysignals/signals.pyx", line 98, in
cysignals.signals.sig_raise_exception
KeyboardInterrupt
```

The phenomenon of the infinite loop illustrates why we must always prove that an algorithm terminates. So long as the algorithm is described properly, we should only need to worry about "while" statements; all other statements should either be clearly one step, such as an assignment or simple operation, or otherwise depend on an algorithm already proved to terminate.

You should remember that the proof of the Division Theorem (Theorem 22) explained how to use Algorithm for unnatural values.[7] For instance, Case 1 says that if $n < 0$ but $d > 0$, divide $|n|$ by d, obtaining quotient $\hat{q}$ and $\hat{r}$. If $\hat{r} = 0$, use $q = -\hat{q}$ and $r = 0$. Otherwise, use the quotient $q = -\hat{q} - 1$ and the remainder $r = d - \hat{r}$.

[7] Technically, it explained this for *some* unnatural values. Others were in the section's exercises. Guess what's coming in the exercises to this supplement?

Let's write a new command that takes care of this case. Sage has a convenient `if` statement that, like the `while` statement, allows us to execute some lines only if the condition is true. Unlike the `while` statement, an `if` statement does not loop! We'll test if the condition of Case 1 is true; if it is, we'll use the `simplistic_division` command as specified to compute $\hat{q}$ and $\hat{r}$, then adjust them as Case 1 indicates, and return the adjusted values.

```
sage: def unnatural_division(n, d):
sage:    # case 1
         if n < 0 and d > 0:
            q_hat, r_hat = simplistic_division(abs(n), d)
            if r_hat == 0:
               q = -q_hat
               r = 0
            else:
               q = -q_hat - 1
               r = d - r_hat
         # additional cases would go here
         return q, r
```

If you try this with -7 and 3, you obtain the desired result!

```
sage: unnatural_division(-7, 3)
(-3, 2)
sage: unnatural_division(-6, 3)
(-2, 0)
```

In fact, $-7 = (-3) \times 3 + 2$.

Exercises

Exercise 31. While Sage allows you to create new commands using the `def` keyword, they will not usually be very efficient. To see why, compare how long it takes perform the following commands:

```
sage: 100000000.quo_rem(2)
(50000000, 0)
sage: simplistic_division(100000000, 2)
(50000000, 0)
```

How long does each command take?

Exercise 32. The proof of Theorem 22 describes three cases where division by negative numbers can happen. The new `unnatural_division` command implemented only the first case. Add additional lines in place of the comment `# additional cases go here` to implement the other cases.
Hint: First implement only Case 2, then make sure it works properly. Use your answers to Exercises 25 and 26.

1.3 Common divisors

Let $a, b, c \in \mathbb{N}$. We say that c is a *common divisor* of a and b if $c \mid a$ and $c \mid b$.

Example 33. The numbers $a = 12$ and $b = 16$ have common divisors 1, 2, 4.

It is not hard to show that none of a's divisors is larger than a itself.

Lemma 34. *Let $a \in \mathbb{N}^+$. If $d \mid a$, then $d \leq a$.*

Proof. Assume $d \mid a$. By definition, there exists $q \in \mathbb{N}$ such that $qd = a$. By way of contradiction, suppose $d > a$. Then $2d = d + d > a$, $3d = 2d + d > a$, … $qd > a$. By substitution, $a > a$, a contradiction. The assumption that $d > a$ must be invalid; we conclude that $d \leq a$. □

There are only finitely many numbers smaller than a, and only finitely many smaller than b, so if $a, b \neq 0$ they can have only finitely many common divisors. We call the largest of these the *greatest common divisor*, written $\gcd(a, b)$.

Example 35. $\gcd(12, 16) = 4$.

Over two thousand years ago, Euclid described a very nice way to use compute the greatest common divisor via division. This is still considered the most efficient general method to compute a gcd.

Algorithm 1.2 The Euclidean Algorithm

```
Input
  • $a, b \in \mathbb{N}^+$
Output
  • $\gcd(a, b)$
Do
  1. let $m = \max(a, b)$, $n = \min(a, b)$
  2. while $n \neq 0$
     (a) determine $q, r$ that satisfy the Division Theorem
     (b) replace $m$ by $n$, then replace $n$ by $r$
  3. return $m$
```

As with Algorithm 1.1, we'll have to prove that Algorithm 1.2 terminates correctly. However, let's look at an example to get an idea for how it works.

Example 36. We compute $\gcd(142, 64)$. Step 1 of the algorithm assigns $m = 142$, $n = 64$.

Since $n \neq 0$, we proceed to step 2, compute $q = 2$ and $r = 14$, and replace m by 64 and n by 14.

Since $n \neq 0$, we repeat step 2, compute $q = 4$ and $r = 8$, and replace m by 14 and n by 8.

Since $n \neq 0$, we repeat step 2, compute $q = 1$ and $r = 6$, and replace m by 8 and n by 6.

Since $n \neq 0$, we repeat step 2, compute $q = 1$ and $r = 2$, and replace m by 6 and n by 2.

Since $n \neq 0$, we repeat step 2, compute $q = 3$ and $r = 0$, and replace n by 2 and n by 0.

We now have $n = 0$, so the algorithm terminates with $\gcd(142, 64) = 2$. You can confirm this result by listing all the divisors of 142 and 64.

Theorem 37. *The Euclidean Algorithm terminates correctly.*

Proof. Enumerate each m and n in steps 1 and 2 as m_0, m_1, … and n_0, n_1, n_2, …. For convenience, write $d = \gcd(a, b)$.

To show that it terminates, consider that for $i > 0$ we know that n_i is the remainder of dividing m_{i-1} by n_{i-1}, so $n_i \geq n_{i+1}$ for each i. This is a non-increasing sequence of natural numbers; by Theorem 20, it must stabilize at a least element, say n_k. If $n_k \neq 0$, then the algorithm would perform step 2 again, and obtain a remainder from dividing m_k by n_k, and assign it to n_{k+1}. That makes $n_k > n_{k+1}$, contradicting the observation that the sequence stabilized at the least element n_k. The assumption that $n_k \neq 0$ must have been wrong; we conclude that $n_k = 0$. Yet once $n_k = 0$ the algorithm terminates.

To show that the algorithm terminates correctly, we claim that $\gcd(m_i, n_i) = \gcd(m_{i+1}, n_{i+1})$ for each i. If this is true, then the last pair is $\gcd(r, 0) = r$, and we would have $\gcd(a, b) = \gcd(m_0, n_0) = \gcd(m_{\text{last}}, n_{\text{last}}) = r$. So it suffices to show the claim.

To that end, let $d = \gcd(m_i, n_i)$, and choose $x, y \in \mathbb{N}$ such that $m_i = xd$ and $n_i = yd$. The algorithm assigns $m_{i+1} = n_i$ and n_{i+1} to be the remainder of dividing m_i by n_i. Let q be the quotient of that division, so that

$$m_i = qn_i + n_{i+1} .$$

By substitution,

$$xd = q(yd) + n_{i+1} .$$

We rewrite this as

$$d(x - qy) = n_{i+1} .$$

By definition, $d \mid n_{i+1}$, so d is a common divisor of n_{i+1} and $n_i = m_{i+1}$. So $d \leq \gcd(m_{i+1}, n_{i+1})$.

Now let $d' = \gcd(m_{i+1}, n_{i+1}) = \gcd(n_i, n_{i+1})$, and choose $u, v \in \mathbb{N}$ such that $n_i = ud'$ and $n_{i+1} = vd'$. By substitution,

$$m_i = qn_i + n_{i+1} \quad \Longrightarrow \quad m_i = q(ud') + vd' \quad \Longrightarrow \quad m_i = d'(qu + v) .$$

By definition, $d' \mid m_i$, so d' is a common divisor of m_i and n_i. So $d' \leq \gcd(m_i, n_i)$. Putting this all together,

$$d \leq \gcd(m_{i+1}, n_{i+1}) = d' \leq \gcd(m_i, n_i) = d .$$

In short,

$$d \leq d' \text{ and } d' \leq d ;$$

by Exercise 24, $d = d'$. By substitution,

$$\gcd(m_i, n_i) = \gcd(m_{i+1}, n_{i+1}) \ .$$

□

The various quotients and remainders turn out to be even more useful.

Theorem 38 (Extended Euclidean Algorithm)**.** *For any $a, b \in \mathbb{N}$, there exist $x, y \in \mathbb{Z}$ such that $ax + by = \gcd(a, b)$.*

Example 39. For $a = 142$ and $b = 64$, we have $142 \times (-9) + 64 \times 20 = 2$.

We call the equation

$$ax + by = \gcd(a, b)$$

the *Bézout identity*, and we call x and y *Bézout coefficients* of $\gcd(a, b)$. There are infinitely many Bézout coefficients, but it suffices to find only one pair. We can find these coefficients by back-substituting through the various divisions of the Euclidean Algorithm, in reverse order.

Example 40. We found $\gcd(142, 64) = 2$ via the divisions

$$\begin{aligned} 142 &= 2 \times 64 + 14 && (1.1) \\ 64 &= 4 \times 14 + 8 && (1.2) \\ 14 &= 1 \times 8 + 6 && (1.3) \\ 8 &= 1 \times 6 + 2 \ . \end{aligned}$$

First we isolate $\gcd(142, 64)$ in the last equation,

$$2 = 8 + (-1) \times 6 \ . \tag{1.4}$$

Similarly, equation (1.3) tells us that

$$6 = 14 + (-1) \times 8 \ .$$

Substitute that into equation (1.4) and we have

$$2 = 8 + (-1) \times [14 + (-1 \times 8)] \ ,$$

or

$$2 = 2 \times 8 + (-1) \times 14 \ . \tag{1.5}$$

Equation (1.2) tells us that

$$8 = 64 + (-4) \times 14 \ .$$

Substitute that into equation (1.5) and we have

$$2 = 2 \times [64 + (-4) \times 14] + (-1) \times 14 \ ,$$

or

$$2 = 2 \times 64 + (-9) \times 14 \ . \tag{1.6}$$

Equation (1.1) tells us that

$$14 = 142 + (-2) \times 64 \ .$$

Substitute that into equation (1.6) and we have

$$2 = 2 \times 64 + (-9) \times [142 + (-2) \times 64] \ ,$$

or

$$2 = 20 \times 64 + (-9) \times 142 \ ,$$

as claimed.

We can use this technique to prove the Extended Euclidean Algorithm.

Proof of Theorem 38. Enumerate the various divisions performed during the Euclidean Algorithm as

$$m_0 = q_0 n_0 + r_0 \quad , \quad m_1 = q_1 n_1 + r_1, \quad \ldots, \quad m_k = q_k n_k + r_k \ ,$$

where $r_k = \gcd(a, b)$ is the last non-zero remainder. Rewrite the last division as

$$\gcd(a, b) = m_k - q_k n_k \ . \tag{1.7}$$

Recall that $m_k = n_{k-1}$ and $n_k = r_{k-1}$, so rewrite this equation as

$$\gcd(a, b) = n_{k-1} - q_k r_{k-1} \ .$$

Rewrite the previous division as

$$r_{k-1} = m_{k-1} - q_{k-1} n_{k-1} \ .$$

Substitute into equation (1.7) to obtain

$$\gcd(a,b) = n_{k-1} - q_k(m_{k-1} - q_{k-1}n_{k-1}) = (1 + q_kq_{k-1})\,n_{k-1} + (-q_k)\,m_{k-1} .$$

By repeating this process, we eventually obtain an expression

$$\gcd(a,b) = Q_0m_0 + Q_1n_0 ,$$

which by substitution becomes

$$\gcd(a,b) = Q_0a + Q_1b .$$

The values $x = Q_0$ and $y = Q_1$ satisfy the theorem. □

Algorithm 1.3 describes a step-by-step method for the Extended Euclidean Algorithm.

Theorem 41. *Algorithm 1.3 terminates correctly.*

Proof. Termination? Step 1 terminates by Theorem 37. Steps 2, 3, 4, 5(a,b), and 6 are simple statements, so they will not inhibit termination. That brings us to step 5, a while statement. The statement continues as long as i is a natural number; it starts at $i = k - 1$ and is changed only by step 5(b), which decreases it by 1. In other words, step 5 will consider the values $i = k-1, k-2, \dots, 1, 0, -1$, at which point the condition $i \in \mathbb{N}$ is no longer true, and the algorithm proceeds to step 6. Hence the algorithm terminates.

Correctness? The computations of steps 3 and 5(a) replicate the proof of Theorem 38. □

Exercises

Exercise 42. For each pair $a, b \in \mathbb{N}$, use the Euclidean Algorithm to compute $\gcd(a,b)$. Then use the Extended Euclidean Algorithm to compute the Bézout coefficients of a and b.

(a) $a = 4, b = 9$

(b) $a = 100, b = 112$

(c) $a = 255, b = 51$

Algorithm 1.3 Extended Euclidean Algorithm

Inputs

- $a, b \in \mathbb{N}^+$

Outputs

- $s, t \in \mathbb{Z}$ such that $as + bt = \gcd(a, b)$

Do

1. apply the Euclidean Algorithm, enumerating the divisions as $m_i = q_i n_i + r_i$
2. let k be the number of the last division with a nonzero remainder
3. solve $m_k = q_k n_k + r_k$ for r_k, obtaining an expression
$$\gcd(a, b) = m_k s_k + n_k t_k \tag{1.8}$$
(in the first case we have $s_k = 1$ and $t_k = -q_k$)
4. let $i = k - 1$
5. while $i \in \mathbb{N}$
   (a) substitute $r_i = m_i - q_i n_i$ in place of n_{i+1} in (1.8)
   (b) decrement i by 1
6. return $s = s_0, t = t_0$

Exercise 43. Let $n \geq 2$.

(a) Show that $\gcd(n+1, n) = 1$.

(b) What is $\gcd(n+2, n)$? Explain why.
Hint: It depends on the value of n. Try a few examples before deciding and explaining.

Exercise 44. In this section we have discussed common divisors only of positive integers, and the greatest common divisor of two positive integers.

(a) Explain why it makes sense to speak of common divisors of negative numbers, as well.

(b) How would you compute the greatest common divisor of two integers if they are negative?

(c) How would you define $\gcd(a, 0)$ where a is nonzero?

(d) Why does $\gcd(0, 0)$ not make sense?

Exercise 45. Suppose $\gcd(a, n) = 1$.

(a) Show that if $\gcd(b, n) = 1$, then $\gcd(ab, n) = 1$.

(b) Show that $\gcd\left(a^k, n\right) = 1$ for any $k \in \mathbb{N}$.
Hint: It can be easy if you use part (a) and induction.

Exercise 46. Let $a, b, c \in \mathbb{N}$. Suppose $\gcd(a, b) = 1$ and both $a \mid c$ and $b \mid c$. Show that $(ab) \mid c$.

Exercise 47. Another way to compute the Bézout coefficients of $a, b \in \mathbb{Z}$ is by the algorithm below.

(a) Compute the Bézout coefficients of $\gcd(255, 51)$ using this algorithm, and compare your result to Exercise 42.

(b) Prove that Algorithm 1.4 terminates. (Don't worry about correctness. It is correct, but we won't consider those details here.)

Algorithm 1.4 Alternate Extended Euclidean Algorithm

Inputs

- $a, b \in \mathbb{N}^+$

Outputs

- $s, t \in \mathbb{Z}$ such that $as + bt = \gcd(a, b)$

Do

1. set up the following table, which will probably extend by more rows

i	s_i	t_i	m_i	n_i	q_i	r_i
-1	1	0				
0	0	1	$\max(a, b)$	$\min(a, b)$		

   ($m_{-1}, n_{-1}, q_{-1}, r_{-1}$ will remain undefined)

2. let $i = 0$
3. repeat…
   (a) compute q_i, r_i to satisfy the Division Theorem for dividing m_i by n_i
   (b) let $s_{i+1} = s_{i-1} - q_i s_i$, $t_{i+1} = t_{i-1} - q_i t_i$, $m_{i+1} = n_i$, $n_{i+1} = r_i$
   (c) increment i by 1

   …until $r_{i-1} = 0$
4. return $s = s_i$, $t = t_i$

Sage supplement

Sage will compute the greatest common divisor of two integers via the `gcd` command. It will compute the Bézout coefficients at the same time as the gcd via the `xgcd` command.

```
sage: gcd(132, 72)
12
sage: xgcd(132, 72)
(12, -1, 2)
```

The result of `xgcd(a,b)` is (d, x, y) where $\gcd(a, b) = d = ax + by$.

In the last section we illustrated basic programming in Sage by implementing a simplistic division algorithm, even though Sage already has a division operator (which is much faster, anyway). In this section we illustrate how to implement the Extended Euclidean Algorithm, even though Sage already has the `xgcd` command. In this case we implement Algorithm 1.4, the Alternate Extended Euclidean Algorithm. One reason is to show how one can keep track of the results from an iteration.

Line 3(b) of the Alternate Extended Euclidean Algorithm requires us to track several values of s and t. We can do this using a list. We can create a list in Sage using the `list` command or brackets `[]`; we add elements to the end of a list using the `.append` command.

```
sage: L = []
sage: L.append(3)
sage: L.append(5)
sage: L.append(-2)
sage: L.append(5)
[3, 5, -2, 5]
```

Unlike a set, a list can contain multiple copies of an element, so we see 5 twice in `L`.

We access elements using the `[]` operator. Proper usage of the `[]` operator might be a little counterintuitive if you aren't accustomed to languages like C and Python where the first element is element 0, not element 1.

```
sage: L
sage: L[0]
3
sage: L[1]
5
sage: L[2]
-2
sage: L[-1]
-2
```

As indicated above `L[0]` gives you 3, the list's first element. Another curiosity is that negative indices refer to elements from the back of the list. Just as `L[-1]` gives us the last element, `L[-2]` gives us the element before that, and so on.

In a manner similar to sets, we can build lists using Sage's analogy to set-builder notation.

```
sage: L2 = [ i^2 for i in range(20) if is_even(i)
      ]
sage: L2
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

There are a number of useful operations you can perform on a list, but discussing them lies beyond the scope of our current motivation, which is to implement the Alternative Extended Euclidean Algorithm. Let's remind ourselves why we need a list anyway: the instructions in Algorithm 1.4 require us to keep track of previously computed s and t values. To do this, we will maintain two lists, `s` and `t`. We will initialize them with the values indicated at the beginning of Algorithm 1.4. The last values in the lists, `s[-1]` and `t[-1]`, will correspond to s_i and t_i. We can thus compute the next values — the ones we have to add, or `.append` to the lists — by translating

$$s_{i+1} = s_{i-1} - q_i s_i \quad \longrightarrow \quad \texttt{s.append(s[-2] - q*s[-1])}$$

and similarly for t_{i+1}. Thus, translating Algorithm 1.4 into Sage code yields the following.

```
sage: def alternate_euclidean(a, b):
          s = [ 1, 0 ]
          t = [ 0, 1 ]
          m, n = max(a, b), min(a, b)
          r = n
          while r != 0:
             q, r = m.quo_rem(n)
             s.append( s[-2] - q*s[-1] )
             t.append( t[-2] - q*t[-1] )
             m, n = n, r
          return s[-2], t[-2]
```

Before testing it, let's point out an important difference. First, Algorithm 1.4 uses a "repeat…until" construction in step 3. This tells a computer to perform steps 3(a)–3(c) *at least once,* test the subsequent condition ($r_{i-1} = 0$), and if it is false repeat the loop. Sage does not have a command that does this, so when implementing it we had to convert it to a "while" construction. This typically involves setting up the variable(s) in the condition so that the while condition will be true at least once (in this case, `r = n` should do the trick).

Another thing to notice is the use of the != operator. This is how we tell Sage to test whether two values are *not* equal to each other. The statement `r != 0` will be true so long as $r \neq 0$, and becomes false only if $r = 0$.

Now let's try the algorithm.

```
sage: alternate_euclidean(132, 72)
(-1, 2)
```

These are precisely the Bézout coefficients that `xgcd` gave us. While `alternate_euclidean` does not return the gcd itself, we can obtain it as follows.

```
sage: _[0] * 132 + _[1] * 72
12
```

The _ symbol asks Sage for the result of the last statement. In our case, the last statement's result was the pair `(-1, 2)`, so `_[0]` gives us the number -1, and `_[1]` gives us the number 2.

Exercises

Exercise 48. Use Sage's `gcd` and `xgcd` commands to verify your answers to Exercises 42 and 43.

Exercise 49. Define a procedure that implements Algorithm 1.2 in Sage. Don't call your procedure `gcd`, as that would "overwrite" Sage's `gcd` command. Rather, call it `euclidean`, then test it with several examples, such as `euclidean(132,72)`.

1.4 Prime and composite numbers

Except for where we explicitly state otherwise, the rest of this chapter deals only with natural numbers.

The previous section described for us a division algorithm, and proved that it had several nice properties. The case where the remainder is 0 has several interesting possibilities. If n, d, q are all integers and $n = qd$, we say that d *divides* n, and write $n \mid d$. We also say that n is *divisible* by d, and that d is a *divisor* of n. If d does not divide n, but has a remainder, we write $d \nmid n$.

Example 50. $4 \mid 8$ but $4 \nmid 6$.

An important example of divisibility occurs when 2 divides a number; we call such a number *even*.

Prime and composite numbers

Some numbers are very hard to divide evenly. For instance, if you have 5 chocolates, you can only divide them without remainder if you have 1 friend or 5; any other number of friends leaves a remainder. We say that a natural number is *prime* if it has exactly two natural divisors: itself and 1. We can also call a prime number *irreducible*, because it does not "reduce" by factorization. If a number is greater than 1 and not prime, we call it *composite*. These definitions deliberately exclude 0 and 1 from consideration for either.

Example 51. The numbers 2 and 5 are prime. The numbers 4 and 6 are composite. The numbers 0 and 1 are neither prime nor composite, because they don't have exactly two natural divisors (0 has infinitely many while 1 has only itself), and they are smaller than 2.

How can we find prime numbers?

Algorithm 1.5 The Sieve of Eratosthenes

Inputs

- $n > 2$

Outputs

- every prime $p \leq n$

Do

```
1. write the numbers from 2 to $n$
2. let $i = 2$
3. while $i \leq \sqrt{n}$
    (a) if $i$ is not itself crossed out, cross out all multiples of $i$ except for $i$ itself
    (b) increment $i$ by 1
4. return the numbers that are not crossed out
```

Theorem 52. *Algorithm 1.5 terminates correctly.*

Proof. Termination? Steps 1, 2, and 4 can be done in finite time. Step 3 repeats as long as $i < \sqrt{n}$, but i starts at 1 and increments by 1 at step 3(b) each time, so eventually it rises above $\sqrt{n}$. Hence the algorithm terminates.

Correctness? As the algorithm crosses out numbers that are obviously prime, we need merely show that the remaining numbers are all prime. Suppose $2 < a \leq n$ and a is not prime. By Exercise 61, it has a prime divisor, say p. By Lemma 34, $p < a$. Choose $q \in \mathbb{N}$ such that $a = pq$. By Lemma 34, we also have $q < a$. If $p \leq \sqrt{n}$, then we would have encountered p in step 3(a) of the algorithm, and crossed out a as a result. Otherwise, $p > \sqrt{n}$. If $q > \sqrt{n}$ also, then

$$a = pq > \sqrt{n}^2 = n \geq a \ ,$$

a contradiction. So we must have $q < \sqrt{n}$. If q is prime, then we would have encountered q in step 3(a) of the algorithm, and crossed out a as a result. Otherwise, q is composite, and as before it must have a prime divisor, say r; again,

$r < q$. We now have $a = (pq)\,r$, and since $r < q$ and $q < \sqrt{n}$ we have $r < \sqrt{n}$, so we would have encountered r in step 3(a) of the algorithm, and crossed out a as a result.

No matter how we go about it, we crossed out the composite number a, so the algorithm could not return it in step 4. Hence the algorithm returns only prime numbers, and is correct. □

Prime numbers enjoy a special property that composite numbers do not. Consider the product $4 \times 3 = 12$, and the fact that $6 \mid 12$. It is *not* the case that $6 \mid 4$ or $6 \mid 3$. That probably doesn't surprise you.

On the other hand, suppose we know of an even number ab where $a, b \in \mathbb{N}$. If $2 \nmid a$ and $2 \nmid b$, then we can find q_a, q_b such that $a = 2q_a + 1$ and $b = 2q_b + 1$. By substitution,

$$ab = (2q_a + 1)(2q_b + 1) = 4q_aq_b + 2q_a + 2q_b + 1 = 2(2q_aq_b + q_a + q_b) + 1\,.$$

This is strange: we started with ab even, but now ab is odd! That's a contradiction, so one of our assumptions must be wrong. Our only assumptions are "ab is even" and "$2 \nmid a$ and $2 \nmid b$." The first assumption is certainly reasonable; lots of products are even. The second assumption, however, is not at all clear; it must have been an invalid assumption. In other words, at least one of a or b is even.

A big difference between 6 and 2 is that 2 is prime. Indeed, this property holds true for any prime divisor, and fails for every composite divisor.

Theorem 53 (Euclid's Lemma). *Let $d \in \mathbb{N}^+$. Then d is prime if and only if any time d divides a product ab, we also have $d \mid a$ or $d \mid b$.*

Proof. Assume d is prime and $d \mid ab$. If $d \mid a$, then the statement "$d \mid a$ or $d \mid b$" is true, and we're done. Otherwise, $d \nmid a$, and the definition of prime tells us that d's only divisors are 1 and itself; inasmuch as $d \nmid a$, the two can have no common divisor except 1. Thus $\gcd(a, d) = 1$ and the Euclidean Algorithm tells us we can find $x, y \in \mathbb{Z}$ such that $ax + dy = 1$. Multiply both sides by b, and we have $(ab)\,x + d\,(by) = b$. By hypothesis, $d \mid ab$, so we can choose $z \in \mathbb{Z}$ such that $ab = dz$. By substitution, $(dz)\,x + d\,(by) = b$, or $d\,(xz + by) = b$. By definition, $d \mid b$. We have shown that if d is prime, then $d \mid a$ or $d \mid b$.

Conversely, assume that any time $d \mid ab$, we also have $d \mid a$ or $d \mid b$. Choose any $x, y \in \mathbb{N}$ such that $d = xy$. By Lemma 34, $x, y \leq d$. On the other hand, $d \cdot 1 = xy$ as well. By definition, $d \mid xy$. By hypothesis, $d \mid x$ or $d \mid y$; let's say $d \mid x$. By Lemma 34, $d \leq x$. We now have $x \leq d \leq x$; by Exercise 24, $x = d$, and thus $y = 1$. Since xy was an arbitrary factorization of d, the only factors of d are 1 and itself. Hence d is prime. □

Euclid's Lemma is actually a special case of a more general fact.

Theorem 54. *Let $d \in \mathbb{Z}$ with $d \neq 0$ and $a, b \in \mathbb{Z}$. If $d \mid ab$ and $\gcd(a, d) = 1$, then $d \mid b$.*

We leave the proof to Exercise 59.

Theorem 54 is just one example of how numbers that are not prime can on some occasions interact with other numbers as if they were prime. You will see more examples of this later on. The relationship is so important that we make the following definition: If $\gcd(a, b) = 1$ then we call a and b *relatively prime.*

Factorization

In Exercise 61 you will show that every composite number has at least one prime divisor. We can actually say something much stronger.

Theorem 55 (The Fundamental Theorem of Arithmetic)**.** *Let $n > 2$ be an integer. There exist prime numbers $p_1, \ldots, p_k$ such that $n = p_1 \cdots p_k$. (The p's might not be distinct.) Moreover, if $p_1 \leq \cdots \leq p_k$, then this expression is unique.*

Example 56. We can factor $40 = 2 \times 2 \times 2 \times 5$. Here $p_1 = p_2 = p_3 = 2$ and $p_4 = 5$.

Proof. If n is prime, then put $p_1 = n$ and we are done.

Otherwise, let $m_1 = n$ and $i = 1$. While m_i is composite, Exercise 61 tells us that m_i has a prime divisor; call it q_i, and choose m_{i+1} such that $m_i = m_{i+1}q_i$; then increment i by 1. By Lemma 34, $m_{i+1} \leq m_i$ for each $i = 1, \ldots$; since q_i is prime and m_i is not, we see that $m_{i+1} < m_i$. By Theorem 20, the sequence of m's stabilizes at a least value, m_k. Were m_k composite, we could prolong the sequence, but the sequence has now stabilized, so m_k must be prime; let $q_k = m_k$.

We claim that each q_i divides $n/(q_1 \cdots q_{i-1})$. To see why, notice that $q_1 \mid m_1$ and $m_2 = {}^{m_1}/_{q_1} = {}^{n}/_{q_1}$, so $n = q_1 m_2$. Then if $q_i \mid m_i$ and $m_{i+1} = {}^{m_i}/_{q_i}$, rewriting and substitution gives us

$$n = q_1 \cdots q_{i-1} m_i = q_1 \cdots q_{i-1} (q_i m_{i+1}) = (q_1 \cdots q_i)\, m_{i+1},$$

which we can rewrite again as

$$\frac{n}{q_1 \cdots q_{i-1}} = q_i m_{i+1},$$

as claimed. Moreover,

$$n = q_1 \cdots q_{k-1} m_k = q_1 \cdots q_k \ ,$$

so n factors into prime numbers.

The list of $q's$ now consists of prime numbers. Let $p_1 = \min(q_1, \ldots, q_k)$ and for $i = 2, \ldots, k$ let $p_i = \min(\{q_1, \ldots, q_k\} \setminus \{p_1, \ldots, p_{i-1}\})$. By construction, $p_1 \leq \ldots \leq p_k$. Moreover, each of these corresponds to a unique q_i, so $n = p_1 \cdots p_k$.

It remains to show uniqueness. Suppose we factor n twice and obtain $p_1 \cdots p_k$ and $q_1 \cdots q_\ell$, with the p's and q's prime and not necessarily distinct. Hence $p_1 \cdots p_k = q_1 \cdots q_\ell$. By Euclid's Lemma, $p_1 \mid q_i$ for some $i = 1, \ldots, \ell$; similarly, $q_1 \mid p_j$ for some $j = 1, \ldots, k$. Both p_1 and q_1 have minimal value on each side, so $p_1 \mid q_1$ and $q_1 \mid p_1$. This forces $p_1 = q_1$. Divide both sides by p_1 and we have $p_2 \cdots p_k = q_2 \cdots q_\ell$. Continuing in this fashion we find that $p_2 = q_2$, …, $p_k = q_k$, and $k = \ell$. The factorization is unique. □

Irrational numbers

Example 4 of Section 1.1 mentions that the following question vexed the ancient Greeks:

Is $\sqrt{2}$ rational?

We can now answer this question.

Theorem 57. $\sqrt{2} \notin \mathbb{Q}$.

Proof. By way of contradiction, suppose $\sqrt{2} \in \mathbb{Q}$. By definition, there exists $a/b \in \mathbb{Q}$ such that $\sqrt{2} = a/b$. We may suppose that a/b is in simplest form, because if it weren't, then we could factor a and b, divide common factors from numerator and denominator, and obtain thereby a representation in simplest form.

Rewrite $\sqrt{2} = a/b$ as $2 = a^2/b^2$, and again as $2b^2 = a^2$. Notice that $2 \mid a^2$. Since 2 is prime, $2 \mid a$. Choose q such that $a = 2q$ and substitute into our equation to obtain $\sqrt{2} = 2q/b$. Rewrite as $2 = 4q^2/b^2$, and again as $2b^2 = 4q^2$, or $b^2 = 2q^2$. Notice that $2 \mid b^2$. Since 2 is prime, $2 \mid b$. This contradicts our assumption that a/b is in simplest form. However, this assumption is perfectly reasonable, so it can't be mistaken. The mistake must be in assuming that $\sqrt{2} \in \mathbb{Q}$. □

We conclude that real numbers that are not rational exist, and call them *irrational numbers.*

Exercises

Exercise 58. Use the Sieve of Eratosthenes to compute all the prime numbers smaller than 200.

Exercise 59. Prove Theorem 54. The proof should be similar to that of the first part of Euclid's Lemma.

Exercise 60. Show that $\sqrt{p} \notin \mathbb{Q}$ for any prime number p.

Exercise 61. Show that every composite number has at least one prime divisor. *Do not* use the Fundamental Theorem of Arithmetic, or anything after that.

Sage supplement

To compute the prime factorization of an integer, use the `factor()` command. The `.divides()` and `.is_prime()` dot commands return `True` or `False` to indicate what their names imply: whether one number divides another, and whether a number is prime.

```
sage: factor(100)
2^2 * 5^2
sage: 2.divides(5)
False
sage: 2.divides(4)
True
sage: 2.is_prime()
True
sage: 4.is_prime()
False
```

Sage will produce a list of primes up to n using a command fittingly called `eratosthenes()`.

```
sage: eratosthenes(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

That said, we will do here the same thing we've done in previous sections: implement the Sieve of Eratosthenes as a procedure. This will give us some more practice manipulating lists.

The Sieve requires only one input, an integer n. Algorithm 1.5 first instructs us to "write the numbers from 2 to n." We could ask Sage to `print` the numbers from to to n, but that won't actually help us in our task; the algorithm tells us to write the numbers so that we can manipulate them. To do that in Sage, we want a list or a set. In step 3(a) we see that we need to test if a number i is in our list or set of numbers, and cross out its multiples. For both of these types, the `in` command tests for membership and the dot command `.remove` removes an element. [8] However, sets have the convenient *difference* operation, implemented in Sage as `.difference` or `.difference_update`; using that makes the code easier to follow, so we'll go with that.

```
sage: def sieve(n):
          S = set(i for i in range(2,n+1)
          i = 2
          while i <= sqrt(n):
             if i in S:
                S.difference_update(
                  i*j for j in range(2,(n+i)/i)
                )
             i += 1
          return S
```

Let's look at what this procedure attempts to do.

- The first line defines a procedure named `sieve`, which takes one input, `n`.
- The second line creates a set `S`, which contains the numbers from 2 to n. (Recall that the `range` command does *not* include the last number!) This corresponds to step 1 of Algorithm 1.5.
- The third line corresponds to set 2 of Algorithm 1.5.
- The fourth line begins the same loop that we see in step 3 Algorithm 1.5.

[8]It's also more efficient to test for membership in a set. Depending on the set's structure, it may also be more efficient to remove items from a set than from a list.

- Lines 5–8 implement Step 3(a) of Algorithm 1.5:
  - The fifth line tests if `i` is a member of `S`. This corresponds to the beginning of step 3(a) of Algorithm 1.5!
  - Lines 6–8 creates a new set that consists of all multiples of `i` (computed using `i*j`) starting from `2*i` until `(n+i)/i`, then use the `.difference_update` dot command to remove its elements from `S`. We have "split up" the invocation of `.difference_update` over three lines in part because we didn't have much space in the text here, but also to show that Sage suspends its rules on indentation between parentheses.
- The ninth line corresponds to step 3(b) of Algorithm 1.5.
- The tenth line corresponds to step 4 of Algorithm 1.5.

Let's try the command.

```
sage: sieve(100)
set([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89,
97])
```

The only difference between this result and the one we obtained from `eratosthenes` is that this one comes back as a set rather than a list. It's always good to get the same result!

Exercises

Exercise 62. Does Sage factor negative numbers? Try it and see if the answer makes sense.

Exercise 63. Sage has an `.is_prime` dot command, but we can implement our own function to test if an integer n is prime by using the Sieve of Eratosthenes, then testing if any of its elements divides n. Basically, the following algorithm:

Algorithm 1.6 Naïve primality test

```
Inputs
    • $n$, an integer
Outputs
    • True if $n$ is prime; False otherwise
Do
    1. let $P$ be the set of all primes no larger than $\sqrt{n}$
    2. for each $p \in P$
        (a) if $p \mid n$, return False
    3. return True
```

Implement Algorithm 1.6 as a Sage procedure, and test it on several "large" integers, both prime and not-so-prime.

1.5 Congruence

It's 10:00pm and I tell you we should meet in 4 hours. At what time will we meet? Not 14:00pm, but rather 2:00am, for once we move past 12:59 time resets to 1:00. In algebra we generalize this idea to dials with different settings and later to polynomials. For the rest of this chapter, let $n > 2$.

Congruence

Recall that division is not actually an operation on $\mathbb{Z}$, but a function on $\mathbb{Z}^2$. The Division Theorem tells us that any we can divide any integer by any non-zero integer and obtain a unique quotient and remainder. However, we can refine this function to become an operation. Let $\overline{a}$ be the remainder after dividing $a \in \mathbb{Z}$ by n. Here we have

$$(a, n) \to \overline{a}, \quad \text{which is an element of} \quad \mathbb{Z}^2 \times \mathbb{Z} .$$

In other words, $\overline{a}$ is an operation on $\mathbb{Z}$. Given our assumption that $n > 2$, it is also *closed*: after all, the Division Theorem guarantees us a remainder.

In our example above, 14:00 changes to 2:00 because 2 is the remainder of 14 after division by 12. We say that a is *congruent to b, modulo n, written $a \equiv b \pmod{n}$,* if a and b have the same remainder after division by n. The following characterization based on divisibility is often more convenient.

Theorem 64. *Let $a, b \in \mathbb{Z}$. Then $a \equiv b \pmod{n}$ if and only if $n \mid (a - b)$.*

Proof. By definition, a and b have the same remainder after division by n if and only if we can find $q_a, q_b \in \mathbb{Z}$ and $r \in \{0, 1, \ldots, n-1\}$ such that $a = q_a n + r$ and $b = q_b n + r$. These equations are true if and only if $a - q_a n = r$ and $b - q_b n = r$. These equations are true if and only if $a - q_a n = b - q_b n$. This equation is true if and only if $a - b = (q_a - q_b)\, n$. By definition, this is true if and only if $n \mid (a - b)$. □

Example 65. By definition, $14 \equiv 2 \pmod{12}$. As the theorem indicates, $12 \mid (14 - 2)$.

Congruence is a relation! After all, we can write $a \equiv b \pmod{n}$ as the ordered pair (a, b), and define the set

$$R_n = \{(a, b) : a \equiv b \pmod{n}\} .$$

Example 66. Some elements of R_{12} would be $(3, 15)$, $(-24, 144)$, $(38, 2)$.

Henceforth, however, we keep with convention and write $a \equiv b \pmod{n}$ instead of $(a, b) \in R_{12}$.

The congruence symbol $\equiv$ resembles the equality symbol $=$ for a reason: congruence shares many interesting properties with equality.

Theorem 67. *For any $n > 2$, congruence modulo n is an equivalence relation.*

Proof. Let $n > 2$. We have to show three properties: reflexive, symmetric and transitive.

Reflexive: Let $a \in \mathbb{Z}$. We want to show $a \equiv a \pmod{n}$. By Theorem 64, this is true if and only if $n \mid (a - a)$, or $n \mid 0$. The last statement is definitely true, since $n \times 0 = 0$. Hence $a \equiv a \pmod{n}$; or, congruence modulo n is reflexive.

Symmetric: Let $a, b \in \mathbb{Z}$. We want to show that if $a \equiv b \pmod{n}$, then $b \equiv a \pmod{n}$. We leave this as Exercise 82 to the reader.

Transitive: Let $a, b, c \in \mathbb{Z}$. We want to show that if $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$. Assume that $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$. By Theorem 64, $n \mid (b-a)$ and $n \mid (c-b)$. By definition, there exist $x, y \in \mathbb{Z}$ such that $nx = b-a$ and $ny = c-b$.

We need to show that $a \equiv c \pmod{n}$. By Theorem 64, this is true if and only if $n \mid (a-c)$. By definition, this is true if and only if there exists $z \in \mathbb{Z}$ such that $nz = a-c$.

Can we somehow combine $nx = b-a$ and $ny = c-b$ to find the desired z? Indeed we can:

$$nx - ny = (b-a) + (c-b) = c-a\,.$$

If we set $z = x-y$, then

$$nz = n(x-y) = c-a\,.$$

Hence $a \equiv c \pmod{n}$; or, congruence is transitive.

□

Theorem 68. *Let $a, b, c \in \mathbb{Z}$, and suppose $a \equiv b \pmod{n}$.*

(A) $a \pm c \equiv b \pm c \pmod{n}$.

(B) $ac \equiv bc \pmod{n}$.

Proof. By Theorem 64, $n \mid (a-b)$. We leave a proof of (A) to Exercise 83, proving only (B) here.

We want to show that $ac \equiv bc \pmod{n}$. By Theorem 64, this is true if and only if $n \mid (ac-bc)$, or $n \mid [(a-b)\,c]$. We already know that $n \mid (a-b)$; by definition, there exists $q \in \mathbb{Z}$ such that

$$nq = a-b\,. \tag{1.9}$$

Can we find another integer x such that $nx = (a-b)\,c$? Certainly: just multiply both sides of (1.9) to obtain

$$n(qc) = (a-b)\,c\,.$$

Hence $n \mid [(a-b)\,c]$, and $ac \equiv bc \pmod{n}$. □

Example 69. Since $14 \equiv 2 \pmod{12}$, we also have

$$14 - 2 \equiv 2 - 2 \pmod{0} \quad \text{and} \quad 14 \times 2 \equiv 2 \times 2 \pmod{2} .$$

You can verify this by computing their remainders.

Does division preserve congruence?

Theorem 68 means that addition, subtraction, and multiplication preserve congruence. What about division? Unfortunately, division *does not* always preserve congruence.

Example 70. $14 \equiv 2 \pmod{12}$, but if you divide both sides by 2, $7 \not\equiv 1 \pmod{12}$.

On the other hand, sometimes it does preserve congruence.

Example 71. Suppose $n = 5$, and $a \equiv b \pmod{5}$. Suppose further that a and b have a common divisor d, and choose $x, y \in \mathbb{Z}$ such that $a = dx$ and $b = dy$. So

$$dx \equiv dy \pmod{5} .$$

By definition,

$$5 \mid (dx - dy) , \quad \text{or,} \quad 5 \mid [d(x - y)] .$$

Now, 5 is a prime number, so by Euclid's Lemma $5 \mid d$ or $5 \mid (x - y)$.

If $5 \mid d$, then $a \equiv b \equiv 0 \pmod{5}$, and we can't say anything at all about x and y. However, if $5 \nmid d$, then we must have $5 \mid (x - y)$, and by Theorem 64 $x \equiv y \pmod{5}$.

In summary, if $5 \nmid d$ then we can go from $dx \equiv dy$ to $x \equiv y$.

Does this apply to other numbers? The key to our example is that when $5 \nmid d$ we could apply Euclid's Lemma. Euclid's Lemma requires only a prime number. In other words, if p is prime, then we should be able to divide modulo p.

Theorem 72. *The number p is prime if and only if $dx \equiv dy \pmod{p}$ implies $x \equiv y \pmod{p}$ for every $d, x, y \in \mathbb{Z}$ such that $p \nmid d$.*

Proof. Suppose p is prime, and let $d, x, y \in \mathbb{Z}$ such that $dx \equiv dy \pmod{p}$ and $p \nmid d$. By Theorem 64, $p \mid (dx - dy)$, or $p \mid [d(x - y)]$. By hypothesis, $p \nmid d$, so by Euclid's Lemma we must have $p \mid (x - y)$. By definition, $x \equiv y \pmod{p}$.

Conversely, suppose p is not prime. By definition, p is composite, and we can find $a, b \in \mathbb{N}$ such that $ab = p$ and $1 < a, b < p$. Certainly $p \mid (ab - ap)$, so $ab \equiv ap \pmod{p}$, but it is *not* the case that $b \equiv p \equiv 0 \pmod{p}$. □

Just as Euclid's Lemma generalized to a theorem for relatively prime numbers, so does Theorem 72 generalize to a theorem for relatively prime numbers.

Theorem 73. *The numbers d and m are relatively prime if and only if $dx \equiv dy \pmod{m}$ implies $x \equiv y \pmod{m}$ for every $x, y \in \mathbb{Z}$.*

Proof. Suppose d and m are relatively prime, and let $x, y \in \mathbb{Z}$ such that $dx \equiv dy \pmod{m}$. By Theorem 64, $m \mid (dx - dy)$, or $m \mid [d(x-y)]$. By Theorem 54, $m \mid (x-y)$. By Theorem 64, $x \equiv y \pmod{m}$. □

The set $\mathbb{Z}_n$ and its arithmetic

We use our observations in this section to define a new kind of arithmetic. Let

$$\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\} \ .$$

Recall the remainder operation $\overline{a}$ from page 47. Define addition, subtraction, and multiplication on $\mathbb{Z}_n$ in the following way:

$$a \pm b = \overline{a \pm b}$$
$$ab = \overline{ab} \ .$$

The Division Theorem theorem tells us that these operations are *closed;* that is, their result is always in $\mathbb{Z}_n$.

Example 74. In $\mathbb{Z}_{12}$, $10 + 4 = 2$, $2 - 4 = 10$, and $10 \times 4 = 4$.

By which elements can we "divide"? A better way to phrase this might be, which elements have multiplicative inverses? By Theorem 73, only those numbers relatively prime to the modulus.

Example 75. In $\mathbb{Z}_{12}$, only the numbers 1, 5, 7, and 11 have multiplicative inverses: $1^{-1} = 1$, $5^{-1} = 5$, $7^{-1} = 7$, and $11^{-1} = 11$.

Example 76. In $\mathbb{Z}_{14}$, the numbers 1, 3, 5, 9, 11, 13 have multiplicative inverses: $1^{-1} = 1$, $3^{-1} = 5$, $5^{-1} = 3$, $9^{-1} = 11$, $11^{-1} = 9$, and $13^{-1} = 13$.

In small moduli like 12 and 14, it isn't too hard to discover the inverses via brute force. Often, however, the modulus is very large (for example, 32003). In a case like this, how can we find the multiplicative inverse of a number in $\mathbb{Z}_n$?

We can actually just use one method to determine *whether* a has a multiplicative inverse modulo n, as well as *what* the inverse is. Theorem 73 tells us that a has multiplicative inverse if and only if $\gcd(a,n)=1$. To compute the gcd, we perform the Euclidean Algorithm. If in fact $\gcd(a,n)=1$, then by the Extended Euclidean Algorithm we can use its divisions to compute the Bézout coefficients s and t such that

$$as+nt=1\ .$$

Rewrite this as

$$nt=1-as\ .$$

By definition,

$$n \mid (1-as)\,, \quad \text{or} \quad 1\equiv as \pmod n\ .$$

In other words, s is the multiplicative inverse of a, modulo n.

Example 77. In $\mathbb{Z}_{14}$, performing the Euclidean algorithm on 14 and 9 gives us the divisions

$$\begin{aligned}14&=1\times 9+5\\9&=1\times 5+4\\5&=1\times 4+1\\4&=4\times 1+0\ .\end{aligned}$$

Reversing these according to the Extended Euclidean Algorithm, we obtain the equations

$$\begin{aligned}1&=5+(-1)\times 4\\1&=5+(-1)\times[9+(-1)\times 5]\\&=2\times 5+(-1)\times 9\\1&=2\times[14+(-1\times 9)]+(-1)\times 9\\&=2\times 14+(-3)\times 9\ .\end{aligned}$$

So the Bézout coefficient of 9 is -3, and $-3\equiv 11 \pmod{14}$. Hence $11=9^{-1}$ in $\mathbb{Z}_{14}$.

Example 78. In $\mathbb{Z}_{14}$, performing the Euclidean algorithm on 14 and 12 gives us the divisions

$$\begin{aligned}14&=1\times 12+2\\12&=6\times 2+0\ .\end{aligned}$$

This tells us that $\gcd(14,12)\neq 0$, so 12 has no multiplicative inverse modulo 14.

Why is $\mathbb{Z}_n$ interesting? Let a be any integer. By the Division Theorem, there exists a unique remainder r when we divide a by n, and moreover $r \in \mathbb{Z}_n$. Hence, every integer is congruent to some element of $\mathbb{Z}_n$, so by Theorem 68, $\mathbb{Z}_n$ gives us a way to represent arithmetic modulo n in a consistent and unique fashion. We will make use of this in the next section.

Exercises

Exercise 79. In which of the following congruences does the indicated division actually work?

(a) $12 \times a \equiv 12 \times b \pmod 7$ implies $a \equiv b \pmod 7$

(b) $12 \times a \equiv 12 \times b \pmod 6$ implies $a \equiv b \pmod 6$

(c) $12 \times a \equiv 12 \times b \pmod{25}$ implies $a \equiv b \pmod{25}$

Exercise 80. Compute multiplicative inverses of the following numbers, according to the given moduli. If an inverse does not exist, explain why not.

(a) 72^{-1} in $\mathbb{Z}_{101}$

(b) 105^{-1} in $\mathbb{Z}_{539}$

Exercise 81. Show that every nonzero element of $\mathbb{Z}_m$ has a multiplicative inverse if and only if p is prime.

Exercise 82. Complete the proof of Theorem 67 by showing that congruence modulo n is transitive.

Exercise 83. Complete the proof of Theorem 68 by showing that if $a \equiv b \pmod n$, then $a \pm c \equiv b \pm c \pmod n$.

Exercise 84. Recall Euclid's Lemma (Theorem 53).

(a) Use Euclid's Lemma to show that if p is prime and $ab \equiv 0 \pmod p$, then $a \equiv 0 \pmod p$ or $b \equiv 0 \pmod p$.

(b) Find $a, b, m \in \mathbb{N}$ such that $ab \equiv 0 \pmod m$ but neither $a \equiv 0 \pmod m$ nor $b \equiv 0 \pmod m$.

(c) For any fixed $m \in \mathbb{N}$, the numbers a and b of part (b) above are called *zero divisors*. Show that if $m > 2$ is not prime, then you can always find zero divisors $a, b \in \{1, 2, \ldots, m-1\}$.
Hint: Try factoring m.

Exercise 85. Recall the generalization of Euclid's Lemma (Theorem 54).

(a) Use Theorem 54 to show that if $ab \equiv 0 \pmod{m}$ and $\gcd(a, m) = 1$, $b \equiv 0 \pmod{m}$.

(b) Find $a, b, m \in \mathbb{N}^+$ that satisfy part (a). (Notice $b \in \mathbb{N}^+$ means $b \neq 0$.)

This example illustrates further how numbers that are relatively prime interact with each other as if they were prime.

Sage supplement

You already know that we can compute the remainder of a after division by n using `a.quo_rem(n)` and taking the second entry of the result. Another, more convenient way to do this is by using either the `%` operator[9] or the `.mod` dot command.

```
sage: (-10).quo_rem(3)
(-4, 2)
sage: -10 % 3
2
sage: (-10).mod(3)
2
```

Sage doesn't have a command named `is_congruent`, but we can easily create one.

```
sage: def is_congruent(a, b, n):
          return (a % n) == (b % n)
```

[9]You may recognize the `%` operator if you are familiar with the C or Java programming language.

The first line defines our procedure with three arguments, `a`, `b`, and `c`. The second computes the remainders of `a` and `b` after division by `n`, and returns whether they are the same.

```
sage: is_congruent(7, 5, 4)
false
sage: is_congruent(9, 5, 4)
true
```

There's also another way: Sage makes it easy to compute in $\mathbb{Z}_n$! To do this, we must set up every number as an element of $\mathbb{Z}_n$. There are two steps for this: one you have to do only once, but the other you have to do every time you work with a new number.

1. Define `Zn` as the quotient of $\mathbb{Z}$ and the modulus.

2. Initialize a number a as elements of `Zn` by typing `Zn(`a`)`.

```
sage: Z3 = ZZ.quo(3)
sage: a = Z3(-10)
sage: a
2
```

One nice consequence of this is that any arithmetic between `a` and other numbers also occurs in $\mathbb{Z}_n$:

```
sage: a + 7
0
sage: a * 7
2
```

Let's review what happened in these computations.

- We defined `a` as an element of $\mathbb{Z}_3$: originally we set it to -10, but the remainder of dividing -10 by 3 is 2, so Sage sets `a` to 2.

- If `a` has the value 2, how did Sage figure `a + 7` to be 0? The sum is 9, but again Sage automatically reduces it modulo 3 to 0.
- In the same way, for `a * 7` Sage computes the product 14, then reduces it modulo 3 to 2.

Sage similarly coerces numbers involved in a comparison, giving us an easier way to test congruence than the `is_congruent` command we defined earlier. (Make sure you type *two* equality signs in the example below, not one. Otherwise you'll reassign `a`.)

```
sage: a == 5
True
```

Sage will also compute the multiplicative inverse of `a` in a very natural manner: use the exponent -1.

```
sage: a^(-1)
2
```

Recall that not every number in every modulus has a multiplicative inverse. Sage has a way to tell you this, too. We'll look at the same example we conidered in the text, the number 12 in $\mathbb{Z}_{14}$.

```
sage: Z14 = ZZ.quo(14)
sage: a = Z14(12)
sage: a
12
sage: a^(-1)
Error in lines 1-1
Traceback (most recent call last):
  File "/cocalc/lib/python2.7/site-packages/
smc_sagews/sage_server.py", line 1188, in execute
    flags=compile_flags) in namespace, locals
  File "", line 1, in <module>
  File "sage/rings/finite_rings/integer_mod.pyx",
line 2704, in sage.rings.finite_rings.integer_mod.
IntegerMod_int.__pow__ (build/cythonized/sage/
rings/finite_rings/integer_mod.c:30197)
    return ~self._new_c(res)
  File "sage/rings/finite_rings/integer_mod.pyx",
line 2722, in sage.rings.finite_rings.integer_mod.
IntegerMod_int.__invert__ (build/cythonized/sage/
rings/finite_rings/integer_mod.c:30372)
    raise ZeroDivisionError(f"inverse of
Mod({self}, {self.__modulus.sageInteger}) does
not exist")
ZeroDivisionError:  inverse of Mod(12, 14) does
not exist
```

The first line to examine for whenever you encounter an error in Sage is actually the *last* line. It tells you the precise error is, gives a brief explanation, and sometimes even suggests how to fix it. In this case, it gives us a *ZeroDivisionError*, which seems like a strange thing to get when looking for a multiplicative inverse; this will make sense later when we talk about zero divisors.

Exercises

Exercise 86. Use Sage to find the multiplicative inverse of every invertible element of $\mathbb{Z}_{100}$.

Hint: Doing this one number at a time would be tedious. Use a `for` loop to make Sage do them all for you in one go! To avoid a *`ZeroDivisionError`*, use an `if` statement to check whether the number is relatively prime to 100.

1.6 Linear algebra in $\mathbb{Z}_n$

In this section we introduce the reader to linear algebra in $\mathbb{Z}_n$; that is, we consider the question of solving a congruence of the form

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

or, more generally, a system of congruences of the form

$$\begin{cases} ax \equiv b & \pmod{m} \\ cx \equiv d & \pmod{n} \end{cases} .$$

These are called *linear congruences* because the variable, x, is only to the first power.

It turns out that if there is one solution to a linear congruence, there are always *infinitely many solutions.* To see why:

- Suppose y is a solution to $ax \equiv b$. Then

  $$ay \equiv b \pmod{n} ,$$

  but also

  $$a(y+n) = ay + an \equiv b + an \equiv b \pmod{n} ,$$

  because $(b+an) - b = an \equiv 0 \pmod{n}$.

- In a similar fashion, suppose y is a solution to both $ax \equiv b \pmod{m}$ and $cx \equiv d \pmod{n}$. This time, consider the fact that

  $$a(y+mn) = ay + a(mn) \equiv b + (an)m \equiv b \pmod{m} ,$$

  and similarly

  $$c(y+mn) = cy + c(mn) \equiv d + (cm)n \equiv d \pmod{mn} .$$

So *if* there is a solution, there are infinitely many.

That said, the other solutions given can be viewed as not especially interesting; after all:

- For a solution y to $ax \equiv b \pmod{n}$, we know that $y + n \equiv y \pmod{n}$. If we consider y as a solution of $\mathbb{Z}_n$, it might well be unique; we're only guaranteed another solution when we add or subtract a multiple of n. What would be interesting is if we *don't* have a solution that is congruent to y modulo n.

- For a solution y to $ax \equiv b \pmod{m}$ and $cx \equiv d \pmod{n}$, we know that $y + mn \equiv y \pmod{mn}$. If we consider y as a solution of $\mathbb{Z}_{mn}$, it might well be unique; we're only guaranteed another solution when we add or subtract a multiple of mn. What would be interesting is if we *don't* have a solution that is congruent to y modulo mn.

Because of this, this section considers two questions:

- Does a solution exist?
- When it does, is it unique *up to congruence?*

"Up to congruence" means that the solution is the only one in $\mathbb{Z}_n$, where the value of n depends on the problem.

One linear congruence

We begin by looking at $ax \equiv b \pmod{n}$. From the previous section we know that if $\gcd(a, n) = 1$, then a has a multiplicative inverse modulo n; that is, we can find $s \in \mathbb{Z}_n$ such that $as \equiv 1 \pmod{n}$. This leads to a very simple solution to the congruence.

Algorithm 1.7 Solving linear congruences

Inputs

- $ax \equiv b \pmod{n}$, where
  - $a, b, n \in \mathbb{Z}$,
  - $n > 1$, and
  - $\gcd(a, n) = 1$

Outputs

- x satisfying $ax \equiv b$

Do

1. let s be the multiplicative inverse of a modulo n
2. return the remainder of bs after division by n

Theorem 87. *Algorithm 1.7 terminates correctly. The solution is unique up to congruence.*

Proof. Termination? We can find s, t to satisfy step 1 by performing the Extended Euclidean Algorithm (1.3), which terminates by Theorem 41. Step 2 is a return statement, so the algorithm terminates.

Correctness? If we substitute $x = bs$ into the left hand side of $ax \equiv b \pmod{n}$ we have

$$a(bs) = a(sb) = (as)b \equiv 1 \cdot b = b \pmod{n}.$$

So $x = bs$ would be a correct solution. The algorithm actually returns the remainder after division by n; that is, it returns r where $x = qn + r$. Rewrite that equation to $qn = x - r$ and we see that $x \equiv r \pmod{n}$. In other words, n is also a solution.

Is the solution unique up to congruence? Let y, z be solutions to $ax \equiv b \pmod{n}$. Then $ay \equiv b \pmod{n}$ and $az \equiv b \pmod{n}$. By the transitive property of congruence (Theorem 67), $ay \equiv az \pmod{n}$. By hypothesis, $\gcd(a, n) = 1$, so a has a multiplicative inverse modulo n; call it s. Multiply both sides of the congruence by s, and we have

$$(sa)y \equiv (sa)z \quad \Longrightarrow \quad y \equiv z \qquad \pmod{n}.$$

By definition, $n \mid (y-z)$, so $y \equiv z \pmod{n}$, and the solution is unique up to congruence. $\square$

Example 88. Consider the linear congruence $9x \equiv 3 \pmod{14}$. In Example 76 we found that $9^{-1} \equiv 11 \pmod{14}$. Multiply both sides by 11 and we have $99x \equiv 33 \pmod{14}$. However, $99 \equiv 1 \pmod{14}$, and $33 \equiv 5 \pmod{14}$, so

$$99x \equiv 33 \quad \Longrightarrow \quad x \equiv 5 \qquad \pmod{14}\ .$$

If you substitute 5 in for x in $9x \equiv 3 \pmod{14}$, you will see that the equation checks out as true. Moreover, Theorem 87 tells us the solution is unique up to congruence; that is, none of $\{0,1,2,3,4\} \cup \{6,7,\ldots,13\}$ will work for x.

You may have noticed a little wrinkle; Algorithm 1.7 requires $\gcd(a,n) = 1$. What if this is not true? We consider two examples.

Example 89. Algorithm 1.7 cannot solve $6x \equiv 2 \pmod{14}$ directly, because $\gcd(6,14) = 2 \neq 1$. However, x is a solution to the congruence if and only if $14 \mid (6x-2)$, which is true if and only if we can find an integer q such that $14q = 6x - 2$. Both sides of the equation are divisible by 2, so we divide and obtain $7q = 3x - 1$. This is equivalent to $7 \mid (3x-1)$, which is true if and only if x is a solution to $3x \equiv 1 \pmod{7}$. Algorithm 1.7 *can* solve this congruence; it returns $x = 5$. You can easily check that this is also a solution to $6x \equiv 2 \pmod{14}$, precisely what all the equivalences imply.

This solution is not, however, unique! We can add 7 to our solution $x = 3$ to obtain $x = 10$ as another solution to $3x \equiv 1 \pmod{7}$. This is not a new solution modulo 7, but you can easily check that it *is* another solution modulo 14, and it is certainly different from 5 in that modulus!

Example 90. Algorithm 1.7 cannot solve $6x \equiv 1 \pmod{14}$ directly, because $\gcd(6,14) = 2 \neq 1$. In fact, Algorithm 1.7 cannot solve $6x \equiv 1 \pmod{14}$ *at all,* because it is equivalent to the equation $6x = 14q + 1$, or $6x - 14q = 1$, or $2(3x - 7q) = 1$. This latter equation is true if and only if $2 \mid 1$, which it does not!

These results allow us to answer completely the question of solving $ax \equiv b \pmod{n}$.

Theorem 91. *Let $a,b,n \in \mathbb{Z}$, with $n > 1$. Let $d = \gcd(a,b)$.*

- *If $d = 1$, then the linear congruence $ax \equiv b \pmod{n}$ has one solution, and the solution is unique up to congruence.*

- *If $d \neq 1$, then:*
  - *If $d \nmid b$, then there the linear congruence $ax \equiv b \pmod n$ has no solution.*
  - *If $d \mid b$, then the linear congruence $ax \equiv b \pmod n$ has d distinct solutions modulo n, and we can find them by first solving $\left({}^{a}/_{d}\right) x \equiv {}^{b}/_{d}$ $\pmod{}(^{n}/_{d})$, then enumerating the solutions x, $x + {}^{n}/_{d}$, $x + {}^{2n}/_{d}$ $\ldots$, $x + {}^{(d-1)n}/_{d}$.*

Proof. If $d = 1$, then Theorem 87 applies.

If $d \neq 1$ and $d \nmid b$, then we can rewrite $ax \equiv b \pmod n$ as $ax + nq = b$ for some integer q. Then d divides the left hand side, but it does not divide the right, so there can be no solution.

If $d \neq 1$ and $d \mid b$, then choose $\hat{a}, \hat{b}, \hat{n} \in \mathbb{Z}$ such that $\hat{a}d = a$, $\hat{b}d = b$, and $\hat{n}d = n$. The congruence $ax \equiv b \pmod n$ is equivalent to the equation $ax + nq = b$ for some $q \in \mathbb{Z}$. Divide both sides by d to see that this is equivalent to $\hat{a}x + \hat{n}q = \hat{b}$. This latter equation is equivalent to $\hat{a}x \equiv \hat{b} \pmod{\hat{}}n$. We obtained $\hat{a}$ and $\hat{n}$ by dividing a and n by their greatest common divisor, so $\hat{a}$ and $\hat{n}$ can have no common divisor; otherwise, a and n would have a larger one. Hence $\gcd(\hat{a}, \hat{n}) = 1$ and Theorem 87 applies; a solution x to $\hat{a}x \equiv \hat{b} \pmod{\hat{}}n$ exists. As explained above, this congruence is equivalent to $ax \equiv b \pmod n$, so x is a solution to that one, as well. Moreover, $y = x + {}^{kn}/_{d} = x + k\hat{n}$ is likewise a solution to $\hat{a}x \equiv \hat{b} \pmod{\hat{}}n$ for every $k \in \mathbb{N}$, so as with x, y is also a solution to $ax \equiv b \pmod n$.

To show that there are only d distinct solutions modulo n, suppose that $y = x + {}^{in}/_{d}$ and $z = x + {}^{jn}/_{d}$ are congruent modulo n. We have

$$a\left(x + \frac{in}{d}\right) \equiv a\left(x + \frac{jn}{d}\right) \pmod n .$$

A little algebra, and we have

$$ai\left(\frac{n}{d}\right) \equiv aj\left(\frac{n}{d}\right) \pmod n .$$

By definition,

$$n \mid \left[a\left(\frac{n}{d}\right)(i-j)\right] .$$

Recall that $\gcd(a, n) = 1$. By Theorem 54,

$$n \mid \left[\left(\frac{n}{d}\right)(i-j)\right] .$$

Choose $q \in \mathbb{N}$ such that

$$nq = \left(\frac{n}{d}\right)(i-j) \qquad \Longrightarrow \qquad qd = i - j\ .$$

This tells us that $i-j$ is a multiple of d. If we choose only the solutions

$$x\ ,\ x+\frac{n}{d}\ ,\ x+\frac{2n}{d}\ ,\ \ldots\ x+\frac{(d-1)\,n}{d}\ ,$$

then we can never encounter two solutions that are congruent modulo n. □

Several simultaneous congruences

We consider a *system* of congruences

$$\begin{cases} x \equiv a & \pmod{m} \\ x \equiv b & \pmod{n} \end{cases}\ .$$

This section's theorems were a little tough, but they lay the groundwork we need to make the remainder easy (no pun intended).

Theorem 87 tells us that $1 \cdot x \equiv a \pmod{m}$ has a solution $x = a$. Now, a might not be a solution to $x \equiv b \pmod{n}$, but we have pointed out several times that there are infinitely many solutions to $x \equiv a \pmod{m}$, all of them having the form $x = a + km$, where k is any integer. We'd like to find a value of k that makes the second congruence true; in other words, we can treat k as an unknown.

Substitute this expression for x into the second congruence, obtaining

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

Rewrite this as

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

By Theorem 87, we can solve this congruence so long as $\gcd(m,n) \mid (b-a)$. In fact, it is *very* easy to solve if $\gcd(m,n) = 1$, and this is such a useful fact that the result is very ancient.

Theorem 92 (The Chinese Remainder Theorem). *Let* $m, n \in \mathbb{N}$ *such that* $m, n \geq 2$ *and* $\gcd(m,n) = 1$. *The system of linear congruences*

$$\begin{cases} x \equiv a & \pmod{m} \\ x \equiv b & \pmod{n} \end{cases}$$

has a solution, and the solution is unique up to congruence modulo mn.

Proof. As we explained before the theorem, this system has a solution if and only if $\gcd(m,n)=1$. It remains to show that the solution is unique up to congruence modulo mn. Let $y,z\in\mathbb{Z}$ be solutions to the system, so that

$$y\equiv a\equiv z \pmod m \quad \text{and} \quad y\equiv b\equiv z \pmod n\ .$$

By the transitive property, $y\equiv z$ under both moduli. By definition, $m\mid(y-z)$ and $n\mid(y-z)$. By Exercise 46, $mn\mid(y-z)$. By definition, $y\equiv z \pmod m n$. □

What about the more complicated case

$$\begin{cases} ax\equiv b & \pmod m \\ cx\equiv d & \pmod n \end{cases},$$

which we mentioned at the beginning of the section? As with the Chinese Remainder Theorem, we can find solutions to this system using a similar principle.

Example 93. Solve

$$\begin{cases} 5x\equiv 72 & \pmod{99} \\ 3x\equiv 4 & \pmod{101} \end{cases}.$$

It is easy to verify that $\gcd(5,99)=\gcd(3,101)=1$, so we pass through steps 1 and 2 without difficulty. For step 3, we find the multiplicative inverses of 5 modulo 99 and of 3 modulo 101 to rewrite the system as

$$\begin{cases} x\equiv 54 & \pmod{99} \\ x\equiv 35 & \pmod{101} \end{cases}.$$

Again, it is easy to see that $\gcd(99,101)=1$ (use the result of Exercise 43 to make it very easy) so we pass through step 4 without difficulty.

We finally come to something new in step 5. We have to follow step 5(a), which tells us to solve according to the manner outlined in the proof of the Chinese Remainder Theorem. We rewrite the first equation as

$$x=99q+54$$

and substitute this into the second equation,

$$99q+54\equiv 35 \pmod{101}\ .$$

Algorithm 1.8 Solving two linear congruences

Inputs

- $a, b, c, d, m, n \in \mathbb{N}^+$ such that $m, n \geq 2$

Outputs

- a solution to
$$\begin{cases} ax \equiv b & \pmod{m} \\ cx \equiv d & \pmod{n} \end{cases},$$
if it exists; otherwise, $\emptyset$

Do

```
1. if $\gcd(a, m) \nmid b$ or $\gcd(c, n) \nmid d$, return $\emptyset$
2. rewrite the system as
       $\begin{cases} \hat{a}x \equiv \hat{b} & \pmod{\hat{m}} \\ \hat{c}x \equiv \hat{d} & \pmod{\hat{n}} \end{cases}$,
   where $\gcd(\hat{a}, \hat{m}) = \gcd(\hat{c}, \hat{n}) = 1$
3. multiply both sides of the first congruence by $\hat{a}^{-1}$, and both sides of the
   second by $\hat{c}^{-1}$, to obtain the equivalent system
       $\begin{cases} x \equiv b' & \pmod{\hat{m}} \\ x \equiv d' & \pmod{\hat{n}} \end{cases}$,
4. if $\gcd(\hat{m}, \hat{n}) \nmid (b' - d')$, return $\emptyset$
5. else
   (a) if $\gcd(\hat{m}, \hat{n}) = 1$,
       i. compute the unique solution $x$ found by the Chinese Remainder
          Theorem
       ii. return the resulting solution
   (b) else
       i. substitute $x = b' + mk$ into $x \equiv d' \pmod{\hat{n}}$
       ii. return the resulting solution
```

Now we solve this in the usual manner for linear congruences,

$$\begin{aligned} 99q &\equiv 82 \pmod{101} \\ 50 \times 99q &\equiv 50 \times 82 \\ q &\equiv 60\,. \end{aligned}$$

Back-substitute to obtain

$$x = 99 \times 60 + 54 \equiv 5994 \pmod{9999}\,.$$

Verifying this answer is straightforward.

Theorem 94. *Algorithm 1.8 terminates correctly.*

Proof. Termination? Steps 5(a)(ii), 5(b)(i), and 5(b)(iii) do not inhibit termination. By Theorem 37, computing the gcd terminates, so computing the gcd in steps 1, 2, 4, and 5(a) does not inhibit termination. Computing a multiplicative inverse in step 3 requires the Extended Euclidean Algorithm, and by Theorem 41 that terminates, so step 3 does not inhibit termination. By Theorem 1.7, step 5(b)(ii) does not inhibit termination. By the Chinese Remainder Theorem, step 5(a)(i) does not inhibit termination. This covers all the steps, so the algorithm terminates.

Correctness? If the algorithm returns a result in steps 1, 4, or 5(a)(ii), then the Chinese Remainder Theorem guarantees correctness. The only other way it returns a result is in step 5(b)(ii), where Theorem 1.7 and the subsequent discussion imply that the solution is a result to the system. □

Remark. The result obtained from steps 5(a)(ii) and 5(b)(ii) in Algorithm 1.8 are not necessarily unique. Determining *all* distinct solutions is somewhat more difficult than we think is justified at the present time. Nevertheless, see Exercise .

Exercises

Exercise 95. Solve the following linear congruences. If they cannot be solved, explain why not. If there is more than one solution among the canonical residues, list them all.

(a) $4x \equiv 7 \pmod 9$

(b) $100x \equiv 18 \pmod{112}$

(c) $100x \equiv 20 \pmod{112}$

Exercise 96. Solve

$$\begin{cases} 5x \equiv 3 & \pmod{7} \\ 4x \equiv 2 & \pmod{9} \end{cases}.$$

List all distinct solutions modulo 63.

Exercise 97. Solve

$$\begin{cases} 2x \equiv 1 & \pmod{7} \\ 7x \equiv 5 & \pmod{11} \\ 4x \equiv 7 & \pmod{15} \end{cases}.$$

Indicate the modulus in which this solution is unique.
Hint: Divide and conquer. First solve the first two congruences; by the Chinese Remainder Theorem, this gives a unique solution modulo 77. Let's call that solution b; you now know that the solution must satisfy the smaller system

$$\begin{cases} x \equiv b & \pmod{77} \\ 4x \equiv 7 & \pmod{15} \end{cases}.$$

You can solve this the same as before, and find your solution.

Exercise 98. Show that if $\gcd(a, b) \neq 1$ but divides b, then $ax \equiv b \pmod{n}$ still has a solution.
Hint: Consider the equation $ax = nq + b$. "Simplify" this to show that a solution exists. Explain why the solution works for $ax \equiv b \pmod{n}$.

Exercise 99. Solve

$$\begin{cases} x \equiv 12 & \pmod{15} \\ x \equiv 17 & \pmod{20} \end{cases}.$$

Hint: There is one unique solution modulo 60, and distinct 5 solutions modulo 300.

Exercise 100. Consider the system

$$\begin{cases} x \equiv a & \pmod{m} \\ x \equiv b & \pmod{m} \end{cases}$$

where $\gcd(a, m) = 1$ and $\gcd(b, n) = 1$, but $\gcd(m, n) \neq 1$. Suppose that $\gcd(m, n) \mid (b - a)$, as in Exercise 99. In this case, step 5(b)(ii) returns the answer to such a system.

(a) Use the algorithm, as well as insight from Exercise 99, to compute a symbolic formula for one solution.

(b) Explain how to find *all* solutions.

(c) The solutions are distinct modulo what number?

Sage supplement

Sage's `solve` command will solve regular equations. You need to specify the name of the variable to solve for.

```
sage: solve( 5*x == 2, x)
[x == (2/5)]
```

The result is a list of equations that satisfy the solution. Higher-degree equations will have multiple solutions.

```
sage: solve( 5*x^2 == 2, x)
[x == -1/5*sqrt(5)*sqrt(2), x ==
1/5*sqrt(5)*sqrt(2)]
```

Sage will also solve a linear congruence, but it requires a different command. Instead of `solve`, we'll use `solve_mod`, which expects an equation as the first argument, and the modulus as the second.

```
sage: solve_mod( 5*x == 7, 12 )
[(11,)]
```

The solution is a list of incongruent values for each variable; when substituted for the variable(s), they make the congruence true. In this case, there is only one solution, and only one variable, so the list `[...]` gives us one solution `(...)` which lists only one number, `11`.

Oftentimes there can be more than one incongruent solution to a linear congruence. In this case, the list will have more entries.

```
sage: solve_mod( 4*x == 8, 12 )
[(8,), (5,), (2,), (11,)]
```

There are four solutions! One of these solutions is obvious: $x = 2$. What about the others? It isn't hard to verify that in fact

$$4 \times 8 \equiv 4 \times 5 \equiv 4 \times 2 \equiv 4 \times 11 \equiv 8 \pmod{12},$$

but how would you know this in advance? You will explore this in the exercises.

There is also a more traditional way to solve linear congruences. Remember that Theorem 64 tells us we can think of a linear congruence $ax \equiv b \pmod{n}$ as $n \mid (ax - b)$. Choose $q \in \mathbb{Z}$ such that $ax - b = nq$ and we're looking at an equation whose solutions must be via integers.[10] Sage gives us a way to specify that via an `assume` command. This command allows us to specify many kinds of constraints on variables. To specify that the variable `x` is an integer, we use `assume(x, 'integer')`. We can then solve an equation with `x`, and only integer solutions will be allowed.

We illustrate this with the congruence $5x \equiv 7 \pmod{12}$. As per the discussion above, we consider the equation $5x = 12q + 7$. We need to define a symbol in Sage for the variable q, which we can do with the `var` command.

```
sage: var( 'q' )
sage: assume( x, 'integer' )
sage: assume( q, 'integer' )
sage: solve( 5*x == 12*q + 7 )
12*t_0 + 35
```

This tells us that for any integer t_0, $x = 12t_0 + 35$ will satisfy $5x = 12q + 7$. We know that this means it should satisfy $5x \equiv 7 \pmod{12}$, and in fact

$$5 \times (12t_0 + 35) = 60t_0 + 175 \equiv 0 + 7 \pmod{12},$$

as desired. So $x \equiv 35 \pmod{12}$, or preferably we use its canonical linear residue, $x \equiv 11 \pmod{12}$.

[10]Equations of the form $ax + by = c$ where every constant variable is an integer are called *Diophantine equations*. Solving them is a major topic in Number Theory.

Solving Chinese Remainder Theorem problems is also fairly easy in Sage; we use the `crt` command. It expects two arguments, and each of those arguments is to be a list of two integers:

$$\texttt{crt([a, b] , [m, n])} \text{ corresponds to } \begin{cases} x \equiv a & \pmod{m} \\ x \equiv b & \pmod{n} \end{cases}.$$

The coefficients of x must be 1; if they are not, we must first rewrite the system.

Example 101. To solve the system in Example 93,

$$\begin{cases} 5x \equiv 72 & \pmod{99} \\ 3x \equiv 4 & \pmod{101} \end{cases},$$

we first have to rewrite it using the multiplicative inverses. We took care of that already in Example 93, obtaining

$$\begin{cases} x \equiv 54 & \pmod{99} \\ x \equiv 35 & \pmod{101} \end{cases}.$$

We can finally apply `crt` to this form.

```
sage: crt( [ 54, 35 ], [ 99, 101 ] )
5994
```

This is the solution we expect.

Exercises

Exercise 102. Use Sage to solve the following linear congruences.

(a) $5123x \equiv 1001 \pmod{32003}$

(b) $7719x \equiv 10017 \pmod{35}$

(c) $1024x \equiv 256 \pmod{65536}$
Hint: In this last case there are a *lot* of solutions. Don't write down all of them: find a pattern $x = ai + b$, where a and b are fixed integers and i ranges from a smallest to a largest value. Be sure to indicate the smallest and largest values of i that describe a solution.

Exercise 103. Try using Sage to solve $6x \equiv 7 \pmod{12}$. What happens? Why?

Exercise 104. Use Sage to solve the following systems of linear congruences. Be sure to list all incongruent solutions if there are more than one. *Be careful with the latter:* Sage will not automatically tell you all incongruent solutions when there are more than one. You'll have to think about it a bit!

(a) $\begin{cases} 38x & \equiv 52 \pmod{101} \\ 82x & \equiv 7 \pmod{103} \end{cases}$

(b) $\begin{cases} x & \equiv 17 \pmod{20} \\ x & \equiv 13 \pmod{32} \end{cases}$

Exercise 105. Sage will not automatically solve systems of linear congruences of the form

$$\begin{cases} ax & \equiv b \pmod{m} \\ cx & \equiv d \pmod{n} \end{cases}.$$

We can however outline an algorithm to solve them, based on this section's discussions and the procedure of Example 93:

Algorithm 1.9 CRT with coefficients

Inputs

- $a, b, c, d \in \mathbb{N}$
- $m, n \in \mathbb{N}^+$

Outputs

- a solution to the system of linear congruences

$$\begin{cases} ax \equiv b & \pmod{m} \\ cx \equiv d & \pmod{n} \end{cases} \tag{1.10}$$

Do

1. if $\gcd(a, m) \nmid b$ or $\gcd(c, n) \nmid d$ then return $\emptyset$
2. find $\hat{a}, \hat{b}$ such that the system (1.10) is equivalent to

$$\begin{cases} x \equiv \hat{a} & \pmod{m} \\ x \equiv \hat{b} & \pmod{n} \end{cases}$$

3. use Sage to solve the system obtained in step 2 and return that solution

Implement this algorithm as a Sage procedure.
Hint: For step 2, you don't have to create any equations; you just have to find $\hat{a}$ and $\hat{b}$ so that you can use them in step 3. You can do this with Sage by computing $\hat{a} = a^{-1}b$ and $\hat{b} = c^{-1}d$.

1.7 Public-key encryption

We end this chapter with a famous application of the ideas we have studied: *secret communication.* Suppose that Person A and Person B want to exchange messages, but are afraid that Person E might overhear.[11] They need a function

[11] Authors often use "Alice," "Bob," and "Eve" instead of A, B, and E. In our internet economy one could well use "Amazon," "Buyer," and "Eavesdropper."

f that transforms a readable message m into an unreadable cipher c, typically using an encryption key e. That is,

$$c = f(m, e) \ .$$

They also need a way to *undo* the encryption.

In the modern age we use computers to do this. Computers work with numbers, so we need some way to turn letters into numbers. We will adopt a very simple method where

$$A \mapsto 0, \quad B \mapsto 1, \quad \ldots \quad Z \mapsto 25, \quad ; \mapsto 26,$$
$$, \mapsto 27, \quad . \mapsto 28, \quad ␣ \mapsto 29, \quad \text{-} \mapsto 30 \ .$$

(The symbol ␣ indicates a space.) This would encode the message

STOP␣-␣DANGER␣AHEAD

as

18 19 14 15 29 30 29 3 0 13 6 4 17 29 0 7 4 0 3 .

The numbers are elements of $\mathbb{Z}_{31}$. We don't *have* to use the modulus 31; we can use any modulus that is sufficiently large to encode the alphabet. The benefit is that we can leverage modular arithmetic to find a way to communicate secretly.

Classical versus public-key encryption

The "classical" approach to encryption requires A and B to know both the encryption method f and the private key e. Only the ciphertext c is public knowledge, so E's challenge is to determine both f and e. Once E determines this information, decryption is a snap, since typically

$$m = f^{-1}(c, e) \ .$$

and it is "easy" to compute f^{-1} from f. For example;

- The *Cæsar cipher* consists of choosing some $k \in \mathbb{Z}$ and using

$$f(m, e) = \overline{m + e} \ ,$$

where the line over $m + e$ means to divide and take the remainder modulo 31 (or whatever modulus one chooses, only both A and B must know it). Decryption consists of computing

$$f^{-1}(c, e) = \overline{c - e} .$$

This cipher takes its name from Julius Cæsar; according to several ancient Romans, he used a version of this method.

- The *Vigenère* cipher consists of choosing a short sequence $e_1, \ldots, e_\ell$ (sometimes corresponding to an easy-to-remember word) and enciphering several characters at a time

  $$f((m_1, \ldots, m_\ell), (e_1, \ldots, e_\ell)) = \left(\overline{m_1 + e_1}, \ldots, \overline{m_\ell + e_\ell}\right) .$$

  Decryption consists of computing

  $$f^{-1}((c_1, \ldots, c_\ell), (e_1, \ldots, e_\ell)) = \left(\overline{c_1 - e_1}, \ldots, \overline{c_\ell - e_\ell}\right) .$$

  It takes its name from Blaise de Vigenère, though Giovan Battista Bellasio discovered it. For a long time people considered the Vigenère cipher indecipherable if E does not know the key.

- A *one-time pad* uses for its key a sequence of random numbers $e_1, \ldots, e_\ell$ in exactly the same fashion as the Vigenère cipher, except that the sequence must be *long,* at least as long as the message. It takes its name from the fact that one uses a one-time pad exactly once, then never again. Because of this, the cipher has been proved indecipherable if E does not know the key. Unfortunately, generating and storing sequences of random numbers is burdensome.

- A *stream cipher* uses for its key a sequence of *pseudo*-random numbers $e_1, e_2, \ldots$ in exactly the same fashion as the one-time pad. Here, "pseudo-random" means that the numbers are generated according to a formula designed to produce numbers that look random, even though they are not — the reasoning being that if you can generate the numbers according to a formula, then they aren't truly random.

- The *Navajo code talkers* were Navajo men who translated English messages to Navajo, which was then radioed between airplanes in the Pacific theater during World War II. The Japanese Navy had never heard anything like it before, and was completely unable to make sense of it.

Again, these techniques require both A and B to know *both* the method *and* the key, and indecipherability depends on keeping at least one of the two secret. This makes classical encryption practically difficult, as both A and B must not only keep a record of the keys — in a codebook, for instance — they must also keep the record hidden from E. Failure either to use a secure method or to keep the method secret forfeits the security.

- The American Department of State at one time used an encryption method so poor that every half-competent intelligence agency was reading our "secret" communications. One history of encryption called us the "laughing stock of the world."
- The German military in World War II used an encryption device called Enigma. The United States and Britain invested heavily into early computer technology precisely to decrypt Nazi communications. These efforts received an enormous boost when the Allies captured a codebook that a captured submarine's commander was unable to destroy before being boarded.

By contrast, public-key encryption works as follows.

- A chooses a method f and a "public encryption key e."
- A broadcasts in public that anyone who wishes to communicate secretly with A should use the method f and the key e.
- B computes and broadcasts $c = f(m, e)$, so that everyone now knows f, c, and e.
- Decryption consists of using a second, private key d which A keeps secret. To decipher the method, E needs to find d.

This is much easier to deal with on a large scale than private encryption: A and B do not need to keep secret, hidden codebooks. Whenever they want to communicate, they simply broadcast clearly each other's encryption key. What's more, *anyone* can send messages securely to A using this method, not just B.

RSA encryption

RSA encryption takes its name from "Rivest, Shamir, Adelman," the mathematicians who first described the technique publicly. One convenient aspect of RSA is that encryption and decryption use the same mathematical operations; the only difference is in the key. Another convenient aspect is that a computer can perform the operations relatively quickly.

A does the following in preparation to receive messages.

- Choose two prime numbers, p and q.
- Compute $N = pq$.
- Let e be a number that is relatively prime to $\phi = (p-1)(q-1)$.
- Let d be the multiplicative inverse of e, modulo ϕ.
- Invite everyone to send messages using RSA, encryption key e, modulo N.

To send a message, B does the following.

- Compute, then broadcast, $c = m^e \pmod{N}$.

To decrypt the message, A does the following.

- Compute $x = c^d$.

Theorem 106. *If A and B perform the steps above, then* $x \equiv m \pmod{m}$.

We postpone the proof until we build up some background theory. To begin with, you are probably wondering why $\phi = (p-1)(q-1)$ is special. To explain this we need a new set: let $\mathbb{Z}_n^*$ be the subset of $\mathbb{Z}_n$ whose elements are all relatively prime to n.

Example 107. $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$ and $\mathbb{Z}_{31}^* = \{1, 2, \ldots, 31\}$.

Lemma 108. *Let* p *and* q *be prime, and* $N = pq$*. The number* $\phi = (p-1)(q-1)$ *counts the elements of* $\mathbb{Z}_N^*$*. That is,* $\phi = \left|\mathbb{Z}_{pq}^*\right|$.

To help understand the proof, we illustrate it with an example.

Example 109. Let $p = 3$ and $q = 5$. We have $N = 15$ and $\phi = 2 \times 4 = 8$. To see why ϕ really does count the number of integers in $\{1, \ldots, 15\}$ that *are* relatively prime to 15, count the number of integers that are *not*.

Since $N = 3 \times 5$, and both 3 and 5 are prime, a number has a common divisor with N only if it is a multiple of 3 or 5. These are

$$\begin{array}{ll} 3, 6, 9, 12, 15 & \text{(multiples of 3)} \\ 5, 10, 15 & \text{(multiples of 5)}\,. \end{array}$$

The first sequence has 5 multiples of 3; the second has 3 multiples of 5. They have no common elements except the last one, 15. In fact, they *should* have nothing in common, since one consists of multiples of 3, the other consists of multiples of 5, and the least common multiple of 3 and 5 is 15. So the number of integers in $\{1, \ldots, 15\}$ that share a common divisor with 15 is

$$\underbrace{5}_{\text{multiples of 3}} + \underbrace{3}_{\text{multiples of 5}} - \underbrace{1}_{\text{extra 15}} .$$

Hence, the number of integers in $\{1, \ldots, 15\}$ that are relatively prime to 15 is

$$15 - (5 + 3 - 1) = 8 ,$$

which is precisely the value of ϕ we computed above.

Proof of Lemma 108. Let $a \in \{0, 1, \ldots, N-1\}$, and suppose that $\gcd(a, N) \neq 1$. By the Fundamental Theorem of Arithmetic, $\gcd(a, N)$ has a unique prime factorization, say

$$\gcd(a, N) = r_1 \ldots r_k .$$

By definition of divisibility, we can find $s \in \mathbb{N}$ such that $N = s \gcd(a, N)$, so by substitution

$$pq = sr_1 \ldots r_k .$$

By Euclid's Lemma, $p \mid s$ or $p \mid r_i$ for some $i = 1, \ldots, k$. Suppose $p \nmid r_i$ for any i; this forces $p \mid s$. We divide both sides by p and obtain the equation

$$q = r_1 \ldots r_k .$$

By Euclid's Lemma, $q \mid r_i$ for some $i = 1, \ldots, k$. Since q is prime, it factors only as $q = q \times 1$; since the r's are prime, we can have only one r, with $q = r_1$. Either way, p or q is a divisor of $\gcd(a, N)$, so p or q is a divisor of a.

We have shown that if $\gcd(a, N) \neq 1$, then p or q divides a. How many such a's are there in $\{0, 1, \ldots, N-1\}$? The multiples of p are

$$p,\ 2p,\ \ldots\ pq ;$$

the multiples of q are

$$q,\ 2q,\ \ldots pq .$$

The number pq appears in both sequences; are there others? Suppose i, j satisfy $ip = jq$. By Euclid's Lemma, $p \mid j$ or $p \mid q$. Now, q is prime, and $p \neq q$, so

$p \mid j$. Similarly, $q \mid i$. So the smallest number that is a multiple of both p and q is pq itself. Hence our sequences above are completely distinct except for pq itself, and there are

$$\underbrace{q}_{\text{multiples of } p} + \underbrace{p}_{\text{multiples of } q} - \underbrace{1}_{\text{extra } pq}$$

common multiples of p and q from $\{0, 1, \ldots, N-1\}$. These are the only numbers in $\mathbb{Z}_N^*$ that share a common divisor with N, so the number of integers in $\{0, 1, \ldots N-1\}$ that are *not* multiples of p or q are

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

as claimed. □

More generally, suppose $\phi(n) = \mathbb{Z}_n^*$ for any integer n. This number has a useful property.

Theorem 110 (Euler's Theorem)**.** *For any* $a \in \mathbb{Z}_n^*$, $a^{\phi(n)} \equiv 1 \pmod{n}$.

Example 111. In $\mathbb{Z}_{15}$, $2^8 \equiv 1 \pmod{15}$. One way to compute this is by evaluating

$$\underbrace{2 \times 2 \times \cdots \times 2}_{8 \text{ times}}\,,$$

but a more clever way to do it is to realize that $8 = 2^3$ and compute

$$\left(\left(2^2\right)^2\right)^2 .$$

If we reduce modulo 15 every chance we get, we see that in fact

$$\begin{aligned} 2^2 &= 4 \\ \left(2^2\right)^2 &= 4^2 = 16 \equiv 1 \\ \left(\left(2^2\right)^2\right)^2 &\equiv 1^2 = 1\,. \end{aligned}$$

Here we encountered 1 at 2^4, illustrating that we might meet it sooner than the $\phi(n)$ power. Nevertheless, we will still always reach it at $\phi(n)$.

Proof of Euler's Theorem. Let $a \in \mathbb{Z}_n$, and suppose a is relatively prime to n. The set $\mathbb{Z}_n$ is finite, so there can be only finitely many distinct powers of a, modulo n. Let $T = \left\{a, a^2, \ldots, a^k\right\}$ be a complete list of the powers of a. We now make two observations.

Claim. The elements of T are all distinct.

Subproof. Suppose $a^i = a^j$ in $\mathbb{Z}_n$. Let i be the smallest power of a for which this occurs, so that $i \leq j$. Recall that a and n are relatively prime; this means that a has a multiplicative inverse, call it s. Multiply both sides of $a^i = a^j$ by s^{i-1} to obtain

$$a = a^{j-i+1} .$$

Remember that i was supposed to be the smallest positive power where repetition occurred, so we have just proved that $i = 1$. By substitution,

$$a = a^j , \quad a^2 = a^{j+1} , \quad \ldots ;$$

that is, all powers from j on simply repeat powers that already appear in T. Repetition cannot occur until after we have reached the jth power, which means $k = j - 1$ and the elements of T are indeed distinct.

Claim. $a^k = 1$ in $\mathbb{Z}_n$.

Subproof. We showed above that $a^{k+1} = a$. Multiply both sides by the inverse of a to see that $a^k = 1$.

We now perform the following iteration.

```
1. let $U_1 = T$
2. let $i = 2$
3. while $U_1 \cup \cdots \cup U_{i-1} \neq \mathbb{Z}_n^*$
    (a) let $b_i \in \mathbb{Z}_n^* \backslash (U_1 \cup \cdots \cup U_{i-1})$
    (b) let $U_i = \{ab_i, a^2 b_i, \ldots, a_k b_i\}$
    (c) increment $i$ by 1
```

We claim this iteration terminates with $U_1 \cup \cdots \cup U_{\text{last}} = \mathbb{Z}_n^*$, no pair of distinct U's has even one element in common, and the U's all have the same size. We prove each claim individually.

Example 112. Before examining the claims, we illustrate the iteration on a concrete example. Let $n = 15$ and $a = 4$. We have $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$. We start with

$$U_1 = T = \left\{4, 4^2, 4^3, \ldots\right\} = \{4, 1\} \ .$$

Notice that $k = 2$. Since $U_1 \neq \mathbb{Z}_{15}^*$, let $b = 2$ and we have

$$U_2 = \{2 \times 4, 2 \times 1\} = \{8, 2\} \ .$$

Since $U_1 \cup U_2 \neq \mathbb{Z}_{15}^*$, let $b = 7$ and we have

$$U_3 = \{7 \times 4, 7 \times 1\} = \{13, 7\} \ .$$

Since $U_1 \cup U_2 \cup U_3 \neq \mathbb{Z}_{15}^*$, let $b = 11$ and we have

$$U_4 = \{11 \times 4, 11 \times 1\} = \{14, 11\} \ .$$

The iteration has now terminated with $U_1 \cup U_2 \cup U_3 \cup U_4 = \mathbb{Z}_{15}^*$. Distinct U's have no elements in common, and they are all the same size.

Claim. The iteration terminates.

Subproof. Steps 1, 2, 3(a), 3(b), and 3(c) are simple assignments, so by themselves they do not inhibit termination. Only the repetition of step 3 might lead to a never-ending task, but that requires $U_1 \cup \cdots \cup U_{i-1} \neq \mathbb{Z}_n^*$. There are only finitely many elements of $\mathbb{Z}_n^*$, and each time we perform steps 3(a) and 3(b) we move at least one element of $\mathbb{Z}_n^*$ that is not in some U into a new U. Eventually, we run out of elements of $\mathbb{Z}_n^*$, so the iteration must terminate.

Claim. $U_1 \cup \cdots U_{\text{last}} = \mathbb{Z}_n^*$.

Subproof. We showed in the previous claim that the iteration must terminate, but by step 3 the iteration terminates only when $U_1 \cup \cdots \cup U_{\text{last}} = \mathbb{Z}_n$.

Claim. $U_i \cap U_j = \emptyset$ only if $i = j$.

Subproof. Suppose $U_i \cap U_j \neq \emptyset$ and let c be a common element. Without loss of generality, $i \leq j$. By construction, $c \in U_i$ implies that $c = a^\ell b_i$ for some $j = 1, \ldots, k$. Similarly, $c \in U_j$ implies that $c = a^m b_j$ for some $m = 1, \ldots, k$. By substitution,

$$a^\ell b_i = a^m b_j \ .$$

If $m \leq \ell$, then

$$a^{\ell-m} b_i = b_j \ ,$$

which means that $b_j \in T$. This contradicts the choice of b_j as *not* being an element of $U_1 \cup \cdots \cup U_{j-1}$. On the other hand, if $m > \ell$, recall that $a^k = 1$ and $m \leq k$, so

$$a^\ell b_i \times a^{k-m} = a^m b_j \times a^{k-m} = a^k b_j = 1 \times b_j = b_j \ ;$$

in other words,

$$a^{\ell+k-m} b_i = b_j \ .$$

Recall that $\ell < m$, so $\ell + k - m < m + k - m = k$, so $a^{\ell+k-m} b_i \in U_i$, the same contradiction as before.

Claim. The U's all have the same size.

Subproof. It suffices to show that each U has k distinct elements. We have already shown that $U_0 = T$ has k distinct elements. For any other i, suppose there exist $\ell, m \in \{1, \ldots, k\}$ such that $a^\ell b_i = a^m b_i$. Without loss of generality, $\ell \leq m$. Let s be the multiplicative inverse of a in $\mathbb{Z}_n$ and multiply both sides by a^ℓ; we have $b_i = a^{m-\ell} b_i$. Recall that $b_i \in \mathbb{Z}_n^*$; that is, b_i is also relatively prime to n, so it has a multiplicative inverse in $\mathbb{Z}_n$. Let t be the multiplicative inverse of b in $\mathbb{Z}_n$ and multiply both sides by t; we have $1 = a^{m-\ell}$. Since $0 < m, \ell \leq k$ and k is the smallest positive power where $a^k = 1$, we must have $m - \ell = 0$; in other words, $\ell = m$. The elements of U_i are thus all distinct, and it has k elements.

Our three claims show that the U's "divide" $\mathbb{Z}_n^*$ into equally-sized sets. Recall that $\phi = |\mathbb{Z}_n^*|$ If we put $\ell =$ last, then

$$\phi = k \times \ell \ .$$

Hence $a^\phi = a^{k \times \ell} = \left(a^k\right)^\ell = 1^\ell = 1$, as claimed. □

We can now prove RSA's correctness.

Proof of Theorem 106. Observe that

$$x = c^d \equiv (m^e)^d \equiv m^{ed} \pmod{N} \ .$$

If we can show that $m^{ed} \equiv m \pmod{N}$, then we will have proved the theorem.

By construction, $ed \equiv 1 \pmod{\phi}$. By definition, there exists $q \in \mathbb{N}$ such that $ed = 1 + q\phi$. Rewrite

$$x \equiv m^{ed} = m^{1+q\phi} = m \times \left(m^{\phi}\right)^{q} \pmod{N} .$$

Recall that $\phi = (p-1)(q-1)$ is the number of integers $\{0, 1, \ldots, N-1\}$ that are relatively prime to N. By Euler's Theorem, $m^{\phi} \equiv 1 \pmod{N}$. By substitution,

$$x \equiv m \times 1^{q} = m \pmod{N} .$$

The result of A's decryption is B's original message. □

Is RSA secure?

Before we consider this question, let's review what E knows about B's message. E knows that:

- B used RSA with the parameters N and e;
- in RSA, $N = pq$ where p and q are both prime;
- in RSA, the decryption exponent d is e's multiplicative inverse modulo $\phi(N)$;
- by Lemma 108, $\phi(N) = (p-1)(q-1)$;
- applying the Extended Euclidean Algorithm to e and $\phi(N)$ will reveal d.

The one thing E needs is the factorization $N = pq$. Once E knows p and q, E can compute $\phi(N)$ and d, then apply them to decrypt the message.

How would E determine p and q? Again, E *already knows* that N is the product of p and q, which are both prime. So E's task is as "easy" as this:

$$6 = 2 \times 3 ,$$

or as "easy" as this:

$$15 = 3 \times 5 ,$$

or as "easy" as this:

$$33 = 3 \times 11 .$$

This is a grade-school problem! How is RSA secure?

It turns out that factoring is much easier for small numbers than large ones. In real-world RSA encryption, the primes used are quite large. One of the strange quirks of mathematics is that many grade-school problems are easy with small numbers, but unfeasible with large ones. Factoring a number into primes is one of those! In fact, RSA was first described in the late 1970s, and 40 years later there is still no practical way to defeat it. If you could find a practical way to factor two large primes, you would became very famous, possibly very rich, and also possibly very dead, depending on whom you informed first!

Other methods of public-key encryption exist, such as Elgamal or elliptic curve encryption. The long-term security of many of these methods is also not clear in general. Some schemes have been proposed, only to be cracked very quickly: problems that seem difficult to do can sometimes be cracked open quickly once someone finds the right approach — it's just that no one was motivated to find that approach before. Modern cryptography is, therefore, an exciting and active field of research that grows from number theory and algebra — two fields that were once considered as abstract and useless and mathematics could possibly be!

Exercises

Exercise 113. Encode the message `LEAVE␣ME␣ALONE` according to the technique described at the beginning of this section.
Hint: The problem asks you to en*code*, not to en*crypt*. Make sure you understand the difference.

Exercise 114. Another way to encode a message as numbers is to pair consecutive letters together, adding a random letter at the end if needed to get a pair. For example, the message

`STOP␣-␣DANGER␣AHEAD`

pairs up as

`ST, OP, ␣-, ␣D, AN, GE, R␣, AH, EA, DX .`

We then encode each pair `XY` as

$$x \times 31 + y ,$$

where x is the value we'd use for `X` in the encoding described at the beginning of this section, and y is the value we'd use for `Y`. Complete the encoding of the message.

Exercise 115. Use a Cæsar cipher with $k = 3$ to encode `LEAVE␣ME␣ALONE`.

Exercise 116. The message `JUPNKAJADUI` has been encoded using a Cæsar cipher with $k = 9$. Decrypt the message.

Exercise 117. A stream cipher needs a function that generates pseudo-random numbers. One example generator is the following:

$$x_i = \begin{cases} 27, & i = 1 \\ \overline{3x_{i-1}} \in \mathbb{Z}_{31}, & i > 1 \end{cases}.$$

(Here, the notation $\overline{y} \in \mathbb{Z}_{31}$ means to take the remainder of y after dividing by 31.) Compute the first 31 numbers generated by this sequence. Do you think the sequence looks random? Why or why not?

Exercise 118. In Example 111, we showed a shortcut for computing 2^8. Adapt this method to compute the following exponents relatively quickly. If you want to be *really* clever, use Euler's Theorem to make it even faster.

(a) 5^{36} in $\mathbb{Z}_{31}$

(b) 3^{78} in $\mathbb{Z}_{38}$

Exercise 119. Example 112 uses $\mathbb{Z}_{15}^*$ to illustrate the iterative generation of the U's in the proof of Euler's Theorem. Repeat the example with $\mathbb{Z}_{31}^*$ and $a = 2$. Observe how the U's "cover" $\mathbb{Z}_{31}^*$ completely, how they have no elements in common, and how they are all the same size.

Exercise 120. Consider the message (without the period)

`MEET␣AT␣DAWN`.

(a) Encode the message using the encoding described at the beginning of this section. (Don't forget to encode the two spaces!)

(b) Use the RSA algorithm to encrypt the message, using parameters $N = 33$ and $e = 3$.

(c) What value of d would decrypt the message?

Sage supplement

Sage already incorporates fast exponentiation modulo n, *as long as you ask for it.* As it happens, there is a right way and a wrong way.

The *right way* is to define an integer in $\mathbb{Z}_n$. For instance:

```
sage: Z35 = ZZ.quo(35)
sage: Z35(2)^1000000000000
16
```

You'll notice that this computation resolves very quickly. By specifying that $2 \in \mathbb{Z}_{35}$, you have told Sage that you want to compute the power modulo 35. With this information, Sage takes advantage of all the mathematics we have described. On the other hand, suppose we write that second line *only slightly differently:*

```
sage: Z35(2^1000000000000)
```

This takes a lot longer, and might not even work on some machines. The author actually gave up after about a minute passed, so he never saw it produce 16.

What makes the second version take so much longer? The order of operations.

- The first version, explicitly tells Sage that we want $2 \in \mathbb{Z}_{35}$, and only then do we raise it to the enormous exponent. Sage can first divide that exponent by $\phi = 24$, obtaining a remainder of 16. It then computes 2^{16}, dividing by 35 to keep the numbers small.
- The second version tells Sage that we want to compute $2^{1000000000000}$ first, and *only afterwards* should it move the result into $\mathbb{Z}_{35}$. Sage thus tries to compute $2^{1000000000000}$ as a regular integer, which takes a really long time[12] and requires a lot of memory,[13] either of which your machine may lack!

The upshot is that when implementing modular arithmetic, we have to take care to specify that our numbers are in $\mathbb{Z}_n$.

With that in mind, we can illustrate RSA encryption and decryption. We'll use $p = 5$ and $q = 7$, so that $N = 35$ and $\phi = 24$. For an encryption exponent

[12]Try it by hand if you doubt this.

[13]If you do try it by hand, you will probably run out of paper.

we'll choose $e = 7$; then the decryption exponent is $d = 7$,[14] as Sage itself informs us via Bézout coefficients.

```
sage: euler_phi(35)
24
sage: e = 7
sage: xgcd(e, 24)
(1, 7, -2)
```

Encryption and decryption is then a simple matter of encoding the messages and raising them to powers. To encode, we use the Sage command `ord`, which converts a letter to a number. Under the default encoding, the letter `a` has the value 97, so we will subtract its value from `ord(`m`)` in order to obtain numbers between 0 and 35.

```
sage: def encode(m):
         return ord(m) - ord('a'')
```

A few examples:

```
sage: encode('a')
0
sage: encode('m')
12
sage: encode('z')
25
```

Decoding requires us to perform the reverse operation. For this, Sage offers `chr`, which converts a number to a character. As before, we deal with numbers between 0 and 25, inclusive, but the default encoding gives "a" the value 97, so we need to add its value to whatever number comes in.

[14]This is a terrible choice of parameters for the RSA algorithm; in no way should you have $e = d$. Choosing the right parameters is an art form in itself.

```
sage: def decode(n):
          return chr(n + ord('a'))
sage: decode(12)
'm'
sage: decode(25)
'z'
```

Once we have defined these procedures, encryption and decryption is a fairly straightforward matter using a `for` loop inside a list.

- To encrypt, we tell Sage to encode the letters as numbers, put the numbers in $\mathbb{Z}_{35}$, and raise them to the 7th power (e).
- To decrypt, we tell Sage to raise the numbers to the 7th power again (d), then decode the numbers as letters.

We have to take a little care in the second step, because the numbers resulting from operations in $\mathbb{Z}_{35}$ are not integers in Sage's opinion: they're elements of $\mathbb{Z}_{35}$, which are not quite the same thing. Fortunately, we can convert them back into integers using a simple command called `int`.

```
sage: [ Z35( encode(m) )^7 for m in 'secret']
[32, 4, 23, 3, 4, 19]
sage: [ decode( int(n^7) ) for n in _ ]
['s', 'e', 'c', 'r', 'e', 't']
```

If you examine this result carefully, you may wonder whether it is in fact secure. After all, `e` always turns into the same number (in this case, 4). It is well-known that some letters appear more often than others in English text, and "e" typically shows up the most. Hence, a simple *frequency analysis* would tell us which letter corresponded to which number, making it a snap to decrypt.

This skepticism is well warranted; real-life use of the RSA algorithm is not done in quite this fashion. A course on cybersecurity is well beyond the scope of these notes, but one thing we can do to make the algorithm somewhat more secure is to combine several letters at a time. We have to be careful here, as this simultaneously increases the minimum size of the modulus. For instance:

- If we combine two letters at a time, we need $N > 26^2$.

- If we combine three letters at a time, we need $N > 26^3$.
  …
- If we combine ℓ letters at a time, we need $N > 26^\ell$.

This requires us to modify the encoding and decoding algorithms. Instead of encoding or decoding one letter at a time, we'll take ℓ at a time, and multiply each by a power of 26 to move it to the right place.

```
sage: def encode(M):
          result = 0
          for m in M:
             result *= 26
             result += ord('m') - ord('a')
          return result
```

The message `'secret'` now encodes in pairs as:

```
sage: encode('se'), encode('cr'), encode('et')
(472, 69, 123)
```

To encrypt it, we need to choose larger values of p and q, since $N = 35$ is too small to capture numbers like 472. How large *does* N need to be? We are encoding two letters at a time, which means we need

$$N > 26^2 = 676\ .$$

If we choose $p = 29$ and $q = 31$, then $N = 899$ is sufficiently large. We have $\phi = 28 \times 30 = 840$. For an encryption exponent we choose $e = 11$; the decryption exponent will be 611.

```
sage: euler_phi(29*31)
840
sage: xgcd(11, 840)
(1, -229, 3)
sage: 840 - 229
611
```

(We cannot use -229 as an exponent, so we subtract 229 from 840 in order to find a positive multiplicative inverse of 11.)

We can now encrypt as before:

```
sage: Z899 = ZZ.quo(899)
sage: [ Z899(m)^11 for m in [ 'se', 'cr', 'et'] ]
[206, 764, 371]
```

When we encoded `secret` before, the `e`'s repeated. Here there is no repetition, which makes a frequency analysis impossible. With a long enough message, we would encounter some repetition, and some two-letter pairs, such as "an" or "th," appear more frequently than others.

Decryption remains a simple matter of applying the decryption exponent to the result. Decoding, however, requires us to separate the letters which encoding joined; since that involved multiplication, we can decode using the `%` and `/` operators.

```
sage: def decode(N):
          result = ''
          for n in N:
             m = N % 26
             result = chr(m + ord('a')) + result
             N -= m
             N /= 26
          return result
sage: [ decode(int(Z899(n)^611))
          for n in [206, 764, 371] ]
['se', 'cr', 'et']
```

We have successfully decrypted the message!

Exercises

Exercise 121. Reword the encryption of "secret" so that you encrypt and decrypt three letters at a time. This will require you to rewrite the `encode` and `decode` procedures.

Algorithm 1.10 Cæsar cipher

Inputs

- M, a list of numbers corresponding to a message
- $k \in \mathbb{N}$

Outputs

- C, an encryption (or decryption) of M using a Cæsar cipher with an offset of k

Do

```
1. for each $i = 1, 2, \ldots, |M|$
    (a) let $c_i$ be the canonical residue of computing $m_i + k$ modulo 26
2. Return $C = \left(c_1, c_2, \ldots, c_{|M|}\right)$
```

Exercise 122. Implement in Sage the following algorithm to encrypt a message using the Cæsar cipher. Test it against the message `leavemealone`. There are no spaces, and the letters are all lower-case.

Exercise 123. In Exercise 117 you experimented with a pseudo-random number generator which gives us the numbers in the key. The following program will give us the first n numbers in a stream cipher's key.

```
sage: def stream(n):
          ZZ31 = ZZ.quo(31)
          result = [ ZZ31( 27 ) ]
          for each in range(2, n+1):
             result.append( 3*result[-1] )
          return result
```

The command `stream(10)` now gives us the following values:

```
sage: stream(10)
[27, 19, 26, 16, 17, 20, 29, 25, 13]
```

Algorithm 1.11 Encryption via stream cipher

Inputs

- A, a sequence of n letters
- $N \in \mathbb{N}^+$

Outputs

- C, the text M encrypted by a stream cipher

Do

```
1. let $k_1, k_2, \ldots\ k_n$ be the first $n$ numbers of the stream cipher's key
2. let $b_i$ be the encoding of $a_i$ (the $i$th letter of $a$)
3. for $i \in \{1, \ldots, n\}$
    (a) let $c_i = \overline{b_i + k_1}$, where the modulus is $N$
4. return $C = (c_1, \ldots, c_n)$
```

Imagine that you have just exchanged the keys for a stream cipher with a friend, and you want to encrypt the message `secret` using this cipher. Based on the discussion in the text, the following algorithm would do the trick.
Use this algorithm with the pseudo-random generator given above to encrypt the message, `secret`. For extra credit, implement the algorithm in Sage to verify your work.

Chapter 2

Solving polynomial equations

Index