Out of print; full text available for free at http://www.gustavus.edu/+max/concrete-abstractions.html

CHAPTER FOUR

Orders of Growth and Tree Recursion

4.1 Orders of Growth

In the previous chapters we've concerned ourselves with one aspect of how to design procedures: making sure that the generated process calculates the desired result. Although this is clearly important, there are other design considerations as well. If we compare our work to that of an aspiring automotive designer, we've learned how to make cars that get from here to there. That's important, but customers expect more. In this chapter we'll focus on considerations more akin to speed and gas mileage. Along the way we'll also add another style of process to our repertoire, alongside linear recursion and iteration.

At first glance, comparing the speed of two alternative procedures for solving the same problem should be easy. Pull out your stopwatch, time how long one takes, and then time how long the other takes. Nothing to it: one wins, the other loses. This approach has three primary weaknesses:

1. It can't be used to decide which procedure to run, because it requires running both. Similarly, you can't tell in advance that one process is going to take a billion years, and hence isn't worth waiting for, whereas the other one will be done tomorrow if you'll just be patient and wait that long.
2. It doesn't tell you how long other instances of the same general problem are going to take or even which procedure will be faster for them. Maybe method A calculates 52! in 1 millisecond, whereas procedure B takes 5 milliseconds. Now you want to compute 100!. Which method should you use? Maybe A, maybe B; sometimes the method that is faster on small problems is slower on large problems, like a sprinter doing poorly on long-distance races.

Excerpted from Concrete Abstractions; copyright © 1999 by Max Hailperin, Barbara Kaiser, and Karl Knight

3. It doesn't distinguish performance differences that are flukes of the particular hardware, Scheme implementation, or programming details from those that are deeply rooted in the two problem-solving strategies and will persist even if the details are changed.

Computer scientists use several different techniques to cope with these difficulties, but the primary one is this:

> **The asymptotic outlook:** Ask not which takes longer, but rather which is more rapidly taking longer as the problem size increases.

This idea is exceptionally hard to grasp. We are all much more experienced with feeling what it's like for something to be slow than we are with feeling something quickly growing slower. Luckily we have developed a foolproof experiment you can use to get a gut feeling of a process quickly growing slow.

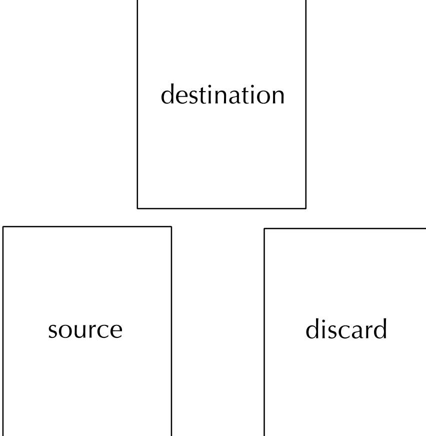
The idea of this experiment is to compare the speeds of two different methods for sorting a deck of numbered cards. To get a feeling for which method becomes slow more rapidly, you will sort decks of different sizes and time yourself. Before you begin, you'll need to get a deck of 32 numbered cards; the web site for this book has sheets of cards that you can print out and cut up, or you could just make your own by writing numbers on index cards. Ask a classmate, friend, or helpful stranger to work with you, because timing your sorting is much easier with a partner. One of you does the actual sorting of the cards. The other keeps track of the rules of the sorting method, provides any necessary prompting, and points out erroneous moves. This kibitzer is also in charge of measuring and recording the time each sort takes (a stopwatch is really helpful for this).

The two sorting methods, or sorting *algorithms*, are described in sidebars that follow. (The word *algorithm* is essentially synonymous with *procedure* or *method*, with the main technical distinction being that only a procedure that is guaranteed to terminate may be called an algorithm. The connotation, as with method, is that one is referring to a general procedure independent of any particular embodiment in a programming language. This distinguishes algorithms from programs.) Before you begin, make sure that both you and your partner understand these two algorithms. You might want to try a practice run using a deck of four cards for selection sorting and a deck of eight cards for merge sorting, because the pattern of that sort isn't so discernible with only four cards.

Now that you're ready to begin, make a deck of four cards by shuffling all the cards well and then taking the top four as the deck to sort. Sort them using selection sort, keeping track of how long the sorting took. Do this again using a deck of 8 cards, then a deck of 16 cards, and finally all 32. Be sure to shuffle all the cards each time. Finally, try sorting decks of 4, 8, 16, and 32 cards using the merge sort algorithm.

Selection Sort

You will use three positions for stacks of cards:



Initially you should put all the cards, face down, on the source stack, with the other two positions empty. Now do the following steps repeatedly:

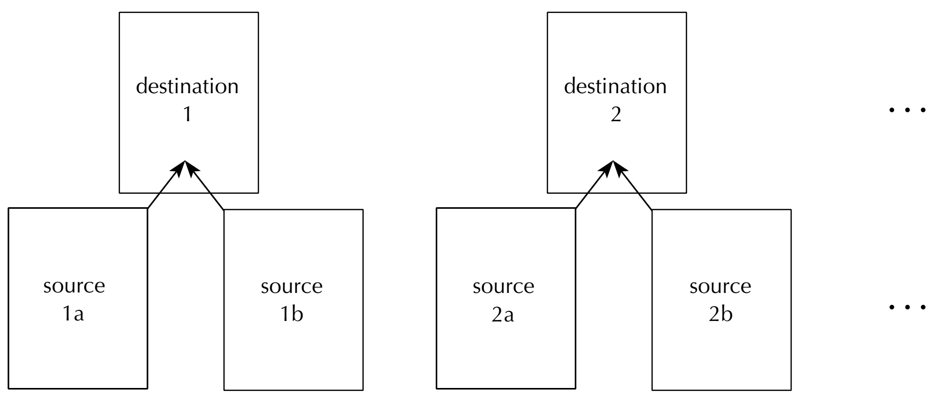
1. Take the top card off the source stack and put it face-up on the destination stack.
2. If that makes the source stack empty, you are done. The destination stack is in numerical order.
3. Otherwise, do the following steps repeatedly until the source stack is empty:
   (a) Take the top card off the source stack and compare it with the top of the destination stack.
   (b) If the source card has a larger number,
      i. Take the card on top of the destination stack and put it face down on the discard stack.
      ii. Put the card you took from the source stack face up on the destination stack.

      Otherwise, put the card from the source stack face down on the discard stack.
4. Slide the discard stack over into the source position, and start again with step 1.

Merge Sort

You will need lots of space for this sorting procedure—enough to spread out all the cards—so it might be best done on the floor. (There are ways to do merge sort with less space, but they are harder to explain.) The basic skill you will need for merge sorting is merging two stacks of cards together, so first refer to the sidebar titled "Merging" (on the following page) for instructions on how to merge. Once you know how to merge, the actual merge sorting process is comparatively easy.

To do the actual merge sort, lay out the cards face down in a row. We will consider these to be the initial source "stacks" of cards, even though there is only one card per stack. The merge sort works by progressively merging pairs of stacks so that there are fewer stacks but each is larger; at the end, there will be a single large stack of cards.



Repeat the following steps until there is a single stack of cards:

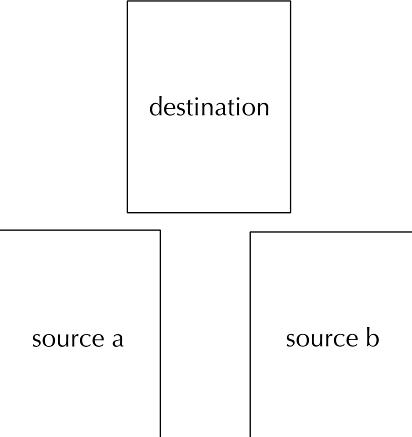
1. Merge the first two face-down stacks of cards.
2. As long as there are at least two face-down stacks, repeat the merging with the next two stacks.
3. Flip each face-up stack over.

The key question is: Suppose we now asked you to sort sixty-four cards. How would you feel about doing it using selection sort? We allowed some space there for you to groan. That's the feel of a process that is quickly becoming slow.

Although the most important point was that gut feeling of how quickly selection sort was becoming a stupid way to sort, we can try extracting some more value from

Merging

You will have the two sorted stacks of cards to merge side by side, face down. You will be producing the result stack above the other two, face up:



Take the top card off of each source stack—source *a* in your left hand, source *b* in your right hand. Now do the following repeatedly, until all the cards are on the destination stack:

1. Compare the two cards you are holding.
2. Place the one with the larger number on it onto the destination stack, face-up.
3. With the hand you just emptied, pick up the next card from the corresponding source stack and go back to step 1. If there is no next card in the empty hand's stack because that stack is empty, put the other card you are holding on the destination stack face-up and continue flipping the rest of the cards over onto the destination stack.

all your labor. Make a table showing your timings, or better yet graph them, or best yet pool them together with timings from everyone else you know and make a graph that shows the average and range for each time. Figure 4.1 is a graph like that for ten pairs of our students; the horizontal ticks are the averages, and the vertical bars represent the range.

If you look very closely at Figure 4.1, you'll notice that the fastest selection sorters can sort four cards faster than the slowest merge sorters. Therefore, if you only have four cards to sort, neither method is intrinsically superior: Either method might turn out faster, depending on the particular skills of the person doing the sorting. On

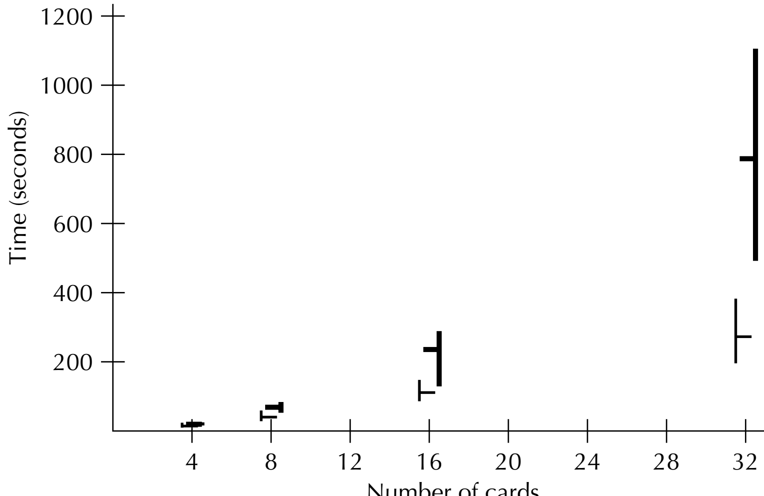


Figure 4.1 Times for selection sort and merge sort. The vertical bars indicate the range of times observed, and the horizontal marks are the averages. The lighter lines with the ranges shown on the left side are for merge sort, and the darker ones with the ranges shown on the right are for selection sort.

the other hand, for 32 cards even the clumsiest merge sorter can outdo even the most nimble-fingered selection sorter. This general phenomenon occurs whenever two methods get slow at different rates. Any initial disparity in speed, no matter how great, will always be eventually overcome by the difference in the intrinsic merits of the algorithms, provided you scale the problem size up far enough. If you were to race, using your bare hands, against a blazingly fast electronic computer programmed to use selection sort, you could beat it by using merge sort, provided the contest involved sorting a large enough data set. (Actually, the necessary data set would be so large that you would be dead before you won the race. Imagine passing the race on to a child, grandchild, etc.)

Another thing we can see by looking at the graph is that if we were to fit a curve through each algorithm's average times, the shapes would be quite different. Of course, it's hard to be very precise, because four points aren't much to go on, but the qualitative difference in shape is rather striking. The merge sort numbers seem to be on a curve that has only a very slight upward bend—almost a straight line. By contrast, no one could mistake the selection sort numbers for falling on a line; they clearly are on a curve with a substantial upward curvature. This corresponds to your gut feeling that doubling the number of cards was going to mean a lot more work—much more than twice as much.

Gathering any more empirical data would strain the patience of even the most patient students, so we use another way to describe the shape of these curves. We will

find a function for each method that describes the relationship between deck size and the number of steps performed. We concentrate on the number of steps rather than how long the sorting takes because we don't know how long each step takes. Indeed, some may take longer than others, and the length may vary from person to person. However, doing the steps faster (or slower) simply results in a rescaled curve and does not change the basic shape of it.

First consider what you did in selection sorting n cards. On the first pass through the deck, you handled all n cards, once or twice each. On the second pass you handled only $n - 1$ of them, on the third pass $n - 2$ of them, and so forth. So, the total number of times you handled a card is somewhere between $n + (n - 1) + (n - 2) + \cdots + 1$ and twice that number. How big is $n + (n - 1) + (n - 2) + \cdots + 1$? It's easy to see that it is no bigger than n^2, because n numbers are being added and the largest of them is n. We can also see that it is at least $n^2/4$, because there are $n/2$ of the numbers that are $n/2$ or larger. Thus we can immediately see that the total number of times you handled a card is bounded between $n^2/4$ and $2n^2$. Because both of these are multiples of n^2, it follows that the basic shape of the touches versus n curve for selection sort must be roughly parabolic. In symbols we say that the number of times you handle a card is $\Theta(n^2)$. (The conventional pronunciation is "big theta of en squared.") This means that for all but perhaps finitely many exceptions it is known to lie between two multiples of n^2. (More particularly, between two *positive* multiples of n^2, or the lower bound would be too easy.)

With a bit more work, we could produce a simple exact formula for the sum $n + (n - 1) + \cdots + 1$. However, this wouldn't really help any, because we don't know how often you only touch a card once in a pass versus twice, and we don't know how long each touch takes. Therefore, we need to be satisfied with a somewhat imprecise answer. On the other hand, we can confidently say that you take at least one hundredth of a second to touch each card, so you take at least $n^2/400$ seconds to sort n cards, and similarly we can confidently say that you take at most 1000 seconds to touch each card, so you take at most $2000n^2$ seconds. Thus, the imprecision that the Θ notation gives us is exactly the kind we need; we're able to say that not only is the number of touches $\Theta(n^2)$ but also the time you take is $\Theta(n^2)$. Our answer, though imprecise, tells the general shape or *order of growth* of the function. Presuming that we do wind up showing merge sort to have a slower order of growth, as the empirical evidence suggests, the difference in orders of growth would be enough to tell us which method must be faster for large enough decks of cards.

Exercise 4.1

Go ahead and figure out exactly what $n + (n - 1) + \cdots + 2 + 1$ is. Do this by adding the first term to the last, the second to the second from last, and so forth. What does each pair add up to? How many pairs are there? What does that make the sum?

Moving on to merge sort, we can similarly analyze how many times you handled the cards by breaking it down into successive passes. In the first pass you merged the n initial stacks down to $n/2$; this involved handling every card. How about the second pass, where you merged $n/2$ stacks down to $n/4$; how many cards did you handle in that pass? Stop and think about this.

If you've thought about it, you've realized that you handled all n cards in every pass. This just leaves the question of how many passes there were. The number of stacks was cut in half each pass, whereas the number of cards per stack doubled. Initially each card was in a separate stack, but at the end all n were in one stack. So, the question can be paraphrased as "How many times does 1 need to be doubled to reach n?" or equivalently as "How many times does n need to be halved to reach 1?" As the sidebar on logarithms explains, the answer is the logarithm to the base 2 of n, written $\log_2 n$, or sometimes just $\lg n$. Putting this together with the fact that you handled all cards in each round, we discover that you did $n \log_2 n$ card handlings. This doesn't account for the steps flipping the stacks over between each pass, and of course there is still the issue of how much time each step takes. Therefore, we're best off again being intentionally imprecise and saying that the time taken to merge sort n cards is $\Theta(n \log n)$.

One interesting point here is that we left the base off of the logarithm. This is because inside a Θ, the base of the logarithm is irrelevant, because changing from one base to another is equivalent to multiplying by a constant factor, as the sidebar explains. Remember, saying that the time is big theta of some function simply means it is between two multiples of that function, without specifying which particular multiples. The time would be between two multiples of $2n^2$ if and only if

Logarithms

If $x^k = y$, we say that k is the logarithm to the base x of y. That is, k is the exponent to which x needs to be raised to produce y. For example, 3 is the logarithm to the base 2 of 8, because you need to multiply three 2s together to produce 8. In symbols, we would write this as $\log_2 8 = 3$. That is, $\log_x y$ is the symbol for the logarithm to the base x of y. The formal definition of logarithm specifies its value even for cases like $\log_2 9$, which clearly isn't an integer, because no number of 2s multiplied together will yield 9. For our purposes, all that you need to know is that $\log_2 9$ is somewhere between 3 and 4, because 9 is between 2^3 and 2^4.

Because we know that three 2s multiplied together produce 8, and two 8s multiplied together produce 64, it follows that six 2s multiplied together will produce 64. In other words, $\log_2 64 = \log_2 8 \times \log_8 64 = 3 \times 2 = 6$. This illustrates a general property of logarithms, namely, $\log_b x = \log_b c \times \log_c x$. So, no matter what x is, its logarithms to the bases b and c differ by the factor $\log_b c$.

it were between two multiples of n^2, so we never say $\Theta(2n^2)$, only the simpler $\Theta(n^2)$. This reason is the same as that for leaving the base of the logarithm unspecified.

In conclusion, note that our analytical results are consistent with our empirical observations. The function $n \log n$ grows just a bit faster than linearly, whereas quadratics are noticeably more upturned. This difference makes merge sort a decidedly superior sorting algorithm; we'll return to it in Chapter 7, when we have the apparatus needed to program it in Scheme.

4.2 Tree Recursion and Digital Signatures

If you watch someone merge sort cards as described in the previous section, you will see that the left-hand and the right-hand halves of the cards don't interact at all until the very last merge step. At that point, each half of the cards is already sorted, and all that is needed is to merge the two sorted halves together. Thus, the merge sort algorithm can be restructured in the following way.

To merge sort a deck of n cards:

1. If $n = 1$, it must already be in order, so you're done.
2. Otherwise:
   a. Merge sort the first $n/2$ cards.
   b. Merge sort the other $n/2$ cards.
   c. Merge together the two sorted halves.

When formulated this way, it is clear that the algorithm is recursive; however, it is not the normal kind of linear recursion we are used to. Rather than first solving a slightly smaller version of the problem and then doing the little bit of work that is left, merge sort first solves *two* much smaller versions of the problem (half the size) and then finishes up by combining their results. This strategy of dividing the work into two (or equally well into three or four) subproblems and then combining the results into the overall solution is known as *tree recursion*. The reason for this name is that the main problem branches into subproblems, each of which branches into sub-subproblems, and so forth, much like the successive branching of the limbs of a tree.

Sometimes this tree-recursive way of thinking can lead you to an algorithm with a lower order of growth than you would otherwise have come up with. This reduced order of growth can be extremely important if you are writing a program designed to be used on very large inputs. To give an example of this, we are going to consider the problem of *digital signatures*.

If you receive a paper document with a signature on it, you can be reasonably sure it was written (or at least agreed to) by the person whose signature it bears. You

can also be sure no one has reattached the signature to a different document, at least as long as each page is individually signed. Finally, you can convince an impartial third party, such as a judge, of these facts, because you are in no better position to falsify the signature than anyone else.

Now consider what happens when you get a digital document, such as an electronic mail message or a file on a disk. How do you know it is authentic? And even if it is authentic, how do you prevent the writer from reneging on anything he agreed to, because he can always claim you forged the agreement? Digital signatures are designed to solve these problems. As such, they are going to be of utmost importance as we convert to doing business in a comparatively paperless manner.

Digital signature systems have three components: a way to reduce an entire message down to a single identifying number, a way to sign these numbers, and a way to verify that any particular signed message is valid. The identifying numbers are called *message digests*; they are derived from the messages by a publicly available *digest function*. The message digests are of some agreed-upon limited size, perhaps 40 digits long. Although a lot of 40 digit numbers exist, far more possible messages do; therefore the digest function is necessarily many to one. However, the digest function must be carefully designed so that it is not feasible to find a message that will have a particular digest or to find two messages that share the same digest. So the validity of a message is effectively equivalent to the validity of its digest. Thus, we have reduced the task of signing messages to the easier mathematical task of signing 40-digit numbers. Although digest functions are interesting in their own right, we'll simplify matters by assuming that the messages we're trying to send are themselves numbers of limited size, so we can skip the digesting step and just sign the message itself. In other words, our messages will be their own digests.

The second part of a digital signature system is the way to sign messages. Each person using the system has a private signature function. If you are sending a message, you can sign it by applying your signature function to the message. Each signature function is designed so that different messages have different signatures and so that computing the signature for a particular message is virtually impossible without knowing the signature function. Because only you, the sender, know the signature function, no one else could forge your signature. When you send a message, you also send along the signature for that message. This pair of numbers is called a *signed message*.

What happens when you receive a signed message? This is the third part of the digital signature system. As receiver, you want to verify that the signature is the right one. To do this you look up the sender's verification function in a public directory and apply it to the signature. This will give you a 40-digit number, which should be equal to the message. Because no one other than the sender can compute the signature for a particular message, you can be reasonably sure that you received a valid signed message. Note that the signature and verification functions are closely related to each other; mathematically speaking, they are inverses. Figure 4.2 shows a

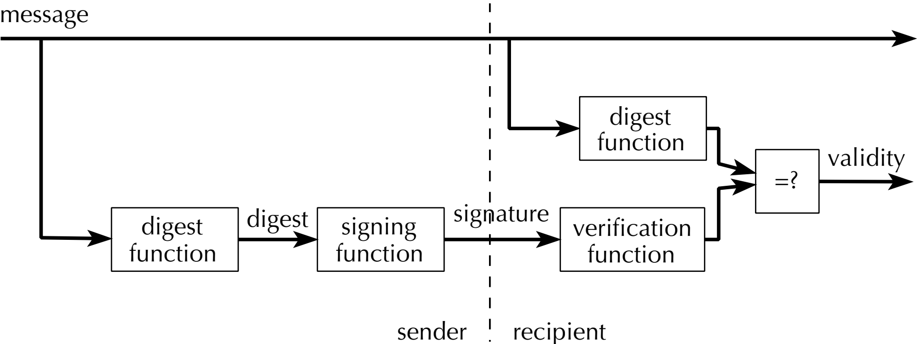


Figure 4.2 The full digital signature system, including the digest function

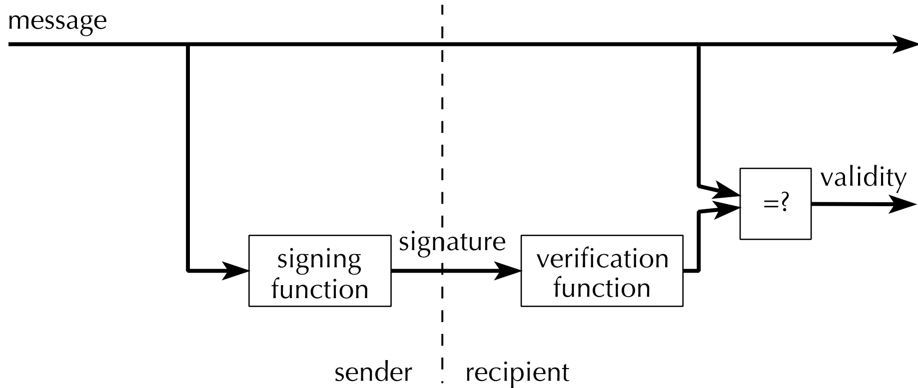


Figure 4.3 Our simplified digital signature system, without a digest function

block diagram of the full version of the digital signature system, including the digest function, and Figure 4.3 similarly depicts the simplified version we're using.

One popular signature and verification strategy is based on an idea known as *modular arithmetic*, which is explained in the accompanying sidebar. In this system, each person has a large number called the *modulus* listed in a public directory under his or her name. The verification function for that person consists of computing the remainder when the cube of the signature is divided by that person's modulus. The `verify` procedure below does this in a straightforward way, using the built-in procedures `remainder` and `expt`. `Expt` raises its first argument to the power specified by its second argument, just like the `power` procedure you wrote in Exercise 2.1.

```
(define verify
  (lambda (signature modulus)
    (remainder (expt signature 3)
               modulus)))
```

Note that we have not yet given the signature function that is the inverse of the verification function above. Before doing that, let us consider an example illustrating how a given signed message is verified.

Suppose that you get a message purporting to come from the authors of this book and containing a partial solution to Exercise 3.8 on page 64. You suspect that the message is actually from a friend playing a prank on you, so you want to check it. The message says that the numerator of the result of `(approximate-golden-ratio (expt 10 -79))` and its signature are as follows:

```
(define gold-num 5972304273877744135569338397692020533504)
(define signature 14367622178330772814011855673053282570996235969
51473988726330337289482255409401120915769529658684452651613736161
53020167902900930324840824269164789456142215776895016041636987254
848119449940440885630)
```

What you need to do is feed that second number into our verification function and see if you get back the first number. If so, you can be sure the message was genuine, because nobody but us knows how to arrange for this to be the case (yet; we're going to give the secret away in a bit). Suppose you looked us up in a directory and found that our modulus is:

```
(define modulus 6716294880486034006153652581749856549007659719419
61654084193604750896012182890124354255484422321487634816640987992
31759689309995696195638345433333958485027650558453766363029391294
0840460009374858969)
```

At this point, you would do the following to find out that we really did send you a personal hint (only the second expression need be evaluated; we evaluate both so that you can see the number returned by the verification function):

```
(verify signature modulus)
5972304273877744135569338397692020533504

(= gold-num
   (verify signature modulus))
#t
```

Having seen how to verify signed messages, we're ready to consider how to generate the signatures. Recall that the signature and verification functions are inverses of each other. So the signature of a given message is an integer s such that the remainder you get when you divide s^3 by the modulus is the original message. In some sense, the signature function might be called the "modular cube root." You should keep in mind, however, that this is quite different from the ordinary cube root. For example,

Modular Arithmetic

The basic operations of arithmetic are +, −, *, and /. These are ordinarily considered as operating on integers, or more generally, real numbers. There are, of course, restrictions as to their applicability; for example, the quotient of two integers is not in general an integer, and division by 0 is not allowed. On the other hand, these operations satisfy a number of formal properties. For example, we say that addition and multiplication are *commutative*, meaning that for all numbers x and y,

$$x + y = y + x$$
$$x * y = y * x$$

Other formal properties are *associativity:*

$$(x + y) + z = x + (y + z)$$
$$(x * y) * z = x * (y * z)$$

and *distributivity:*

$$x * (y + z) = x * y + x * z$$

Are there other types of number systems whose arithmetic operations satisfy the properties given above? One such example is *modular arithmetic*, which might also be called *remainder* or *clock arithmetic*. In modular arithmetic, a specific positive integer m called the *modulus* is chosen. For each nonnegative integer x, we let $x \bmod m$ denote the remainder of x when divided by m; this is just `(remainder x m)` in Scheme. Note that $x \bmod m$ is the unique integer r satisfying $0 \leq r < m$ and for which there is another integer q such that $x = qm + r$. The integer q is the integer quotient of x by m (i.e., `(quotient x m)` in Scheme).

If two integers differ by a multiple of m, they have the same remainder mod m. We can use this fact to show that for all integers x and y,

$$xy \bmod m = (x \bmod m)(y \bmod m) \bmod m$$
$$(x + y) \bmod m = ((x \bmod m) + (y \bmod m)) \bmod m$$

To show that these equalities hold, let $x = q_1m + r_1$ and $y = q_2m + r_2$. Then $xy = (q_1r_2 + q_2r_1 + q_1q_2m)m + r_1r_2$ and $x + y = (q_1 + q_2)m + (r_1 + r_2)$.

The set of all possible remainders mod m is $\{0, 1, \ldots, m - 1\}$. We can define $+_m$ and $*_m$ on this set by

(Continued)

Modular Arithmetic (Continued)

$$x +_m y \stackrel{\text{def}}{=} (x + y) \bmod m$$

$$x *_m y \stackrel{\text{def}}{=} (x * y) \bmod m$$

(The symbol $\stackrel{\text{def}}{=}$ denotes "is defined as.") In Scheme they would be defined as follows:

```
(define mod+
  (lambda (x y)
    (remainder (+ x y) modulus)))

(define mod*
  (lambda (x y)
    (remainder (* x y) modulus)))
```

(We assume `modulus` has been defined.) It is not hard to show that $+_m$ and $*_m$ satisfy commutativity, associativity, and distributivity.

Other operations, such as modular subtraction, division, and exponentiation can be defined in terms of $+_m$ and $*_m$. We'll only consider modular subtraction here, because exponentiation is investigated in the text and division poses theoretical difficulties (namely, it can't always be done).

How should we define modular subtraction? Formally, we would want

$$x - y = x + (-y)$$

where $-y$ is the additive inverse of y (i.e., the number z such that $y + z = 0$). Does such a number exist in modular arithmetic? And if so, is it uniquely determined for each y? The answer to both of these questions is Yes: The (modular) additive inverse of y is $(m-y) \bmod m$, because that is the unique number in $\{0, 1, \ldots, m-1\}$ whose modular sum with y is 0. For example, if $m = 17$ and $y = 5$, then $-y = 12$ because $(5 + 12) \bmod 17 = 0$. This allows us to define modular subtraction as follows:

```
(define mod-
  (lambda (x y)
    (remainder (+ x (- modulus y)) modulus)))
```

if the modulus were a smaller number such as 17, the modular cube root of 10 would be 3. (Why?) In particular, the signature must be an integer.

Signature functions use the same mathematical process as verification functions. The underlying mathematics is somewhat deeper than you have probably encountered and is partly explained in the notes at the end of this chapter. Briefly, for the types of moduli used in this strategy, there is an exponent called the *signing exponent*, depending on the modulus, that is used to calculate the signature. The signature of a number is calculated by raising the number to the signing exponent and then finding the remainder when the result is divided by the modulus. Mathematically, this means that if m is the modulus, e is the signing exponent, x is the message, and s is its signature,

$$s = x^e \bmod m$$

$$x = s^3 \bmod m$$

The fact that this works follows from the fact that for all choices of nonnegative integers $x < m$,

$$x = (x^e \bmod m)^3 \bmod m$$

Thus, the only difference between signing and verifying is the exponent—that's our secret. Only we (so far) know what exponent to use in the signing so that an exponent of 3 will undo it. In fact, for a 198-digit modulus like ours, no one knows how to find the signing exponent in any reasonable amount of time, without knowing something special about how the modulus was chosen.

What is our secret signing exponent? It is

```
(define signing-exponent 4477529920324022670769101721166571032671
77314627974436056129069833930674788593416236170322948214322483305
17527801279310239221589593147057716354461360014347167979987666468
6423606429437389098641670667)
```

That's a big number, in case you didn't notice (again, 198 digits long). This poses a bit of a problem. From a strictly mathematical standpoint, all we would have to do to sign the numerator is

```
(remainder (expt gold-num signing-exponent) modulus)
```

However, this isn't practical, because the result of the exponentiation would be an extremely large number. We don't even want to tell you how large it would be by telling how many digits are in it, because even the number of digits is itself a 200-digit number. This means that if the computer were to evaluate the expression above, it couldn't possibly have enough memory to store the intermediate result produced by

the exponentiation. Keep in mind that there are only about 10^{79} subatomic particles in the universe. This means that if each one of those particles was replaced by a whole universe, complete with 10^{79} particles of its own, and the computer were to store a trillion trillion trillion digits of the intermediate result on each of the particles in this universe of universes, it wouldn't have enough memory.

Luckily there is an out, based on a property noted in the sidebar on modular arithmetic. Namely,

$$xy \bmod m = (x \bmod m)(y \bmod m) \bmod m$$

In other words, we are allowed to do the mod operation before the multiplication as well as after without changing the answer. This is important, because taking a number mod m reduces it to a number less than m. For exponentiation, the important point is this: Rather than multiplying together lots of copies of the base and then taking the result mod m, we can do the mod operation after each step along the way, so the numbers involved never grow very big. Here is a Scheme procedure that does this, based on the observations that $b^0 = 1$ and $b^e = b^{e-1}b$:

```
(define mod-expt
  (lambda (base exponent modulus)
    (define mod*
      (lambda (x y)
        (remainder (* x y) modulus)))
    (if (= exponent 0)
        1
        (mod* (mod-expt base (- exponent 1) modulus)
              base))))
```

We can try this out by using it to reverify the original signature:

```
(mod-expt signature 3 modulus)
5972304273877744135569338397692020533504
```

It works. So now we're all set to use it to sign a new message. Let's sign a nice small number, like 7:

```
(mod-expt 7 signing-exponent modulus)
```

What happens if you try this?

If you tried the above experiment, you probably waited for a while and then got a message like we did:

```
;Aborting!: out of memory
```

The problem is that the definition given above for `mod-expt` does one recursive step for each reduction of the exponent by 1. Because each step takes some amount of memory to hold the main problem while working on the subproblem, this means that the total memory consumption is $\Theta(e)$, where e is the exponent. Given that our signing exponent is nearly 200 digits long, it is hardly surprising that the computer ran out of memory (again, even all the particles in a universe of universes wouldn't be enough). We could fix this problem by switching to a linear iterative version of the procedure, much as in Section 3.2 where we wrote an iterative version of the `power` procedure from Exercise 2.1. This procedure would just keep a running product as it modularly multiplied the numbers one by one. This would reduce the memory consumption to $\Theta(1)$ (i.e., constant).

Unfortunately, the time it would take to do the exponentiation would still be $\Theta(e)$, so even though there would be ample memory, it would all have crumbled to dust before the computation was over. (In fact, there wouldn't even be dust left. The fastest computers that even wild-eyed fanatics dream of would take about 10^{-12} seconds to do a multiplication; there are about 10^7 seconds in a year. Therefore, even such an incredibly fast machine would take something like 10^{180} years to do the calculation. For comparison, the earth itself is only around 10^9 years old, so the incredibly fast computer would require roughly a trillion earth-lifetimes for each particle in our hypothetical universe of universes.)

So, because chipping away at this huge signing exponent isn't going anywhere near fast enough, we are motivated to try something drastic, something that will in one fell swoop dramatically decrease it. Let's cut it in half, using a tree recursive strategy. Keep in mind that b^e means e b's multiplied together. Provided that e is even, we could break that string of multiplications right down the middle into two, each of which is only half as big:

```
(define mod-expt
  (lambda (base exponent modulus)
    (define mod*
      (lambda (x y)
        (remainder (* x y) modulus)))
    (if (= exponent 0)
        1
        (if (even? exponent)
            (mod* (mod-expt base (/ exponent 2) modulus)
                  (mod-expt base (/ exponent 2) modulus))
            (mod* (mod-expt base (- exponent 1) modulus)
                  base)))))
```

Does this help any? Unfortunately not—although at least this version won't run out of memory, because the recursion depth is only $\Theta(\log e)$. Consider what would

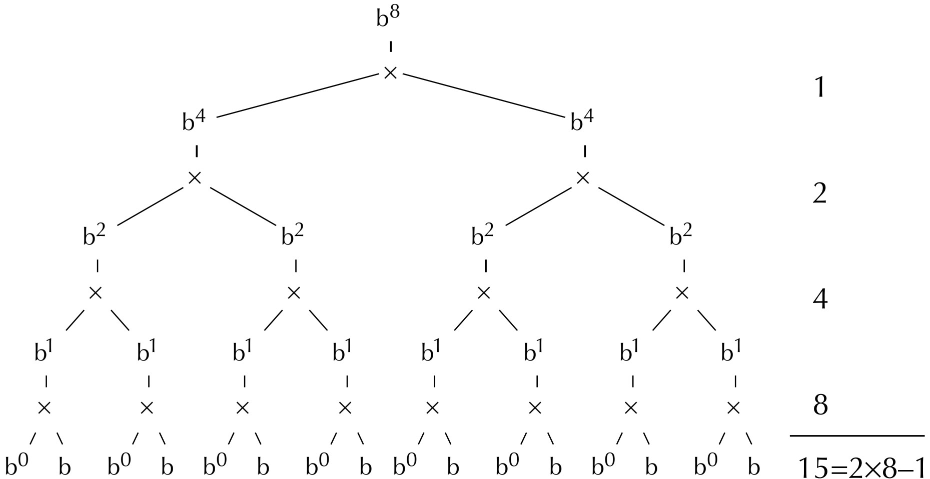


Figure 4.4 A tree recursive exponentiation still does $\Theta(e)$ multiplications.

happen in the simplest possible case, which is when the exponent is a power of 2, so it is not only initially even but in fact stays even after each successive division by 2 until reaching 1. The multiplications form a tree, with one multiplication at the root, two at the next level, four at the next, eight at the next, and so on down to e at the leaves of the tree. This totals $2e - 1$ multiplications when all the levels are added up; Figure 4.4 illustrates this for $e = 8$. The details would be slightly different for an exponent that wasn't a power of 2, but in any case the number of multiplications is still $\Theta(e)$.

Exercise 4.2

In this exercise you will show that this version of `mod-expt` does $\Theta(e)$ multiplications, as we claimed.

a. Use induction to prove each of the following about this latest version of `mod-expt`:

(1) e is a nonnegative integer, `(mod-expt` b e m`)` does at least e multiplications.

(2) When e is a positive integer, `(mod-expt` b e m`)` does at most $2e - 1$ multiplications.

b. To show that the number of multiplications is $\Theta(e)$, it would have sufficed to show that it lies between e and $2e$. However, rather than having you prove that the number of multiplications was at most $2e$, we asked you prove more, namely, that the number of multiplications is at most $2e - 1$. Try using induction to prove that when e is a positive integer, at most $2e$ multiplications are done. What goes wrong? Why is it easier to prove more than you need to, when you're using induction?

You may be wondering why we went down this blind alley. The reason is that although the tree-recursive version is not itself an improvement, it serves as a stepping stone to a better version. You may have already noticed that we compute `(mod-expt base (/ exponent 2) modulus)` twice, yet clearly the result is going to be the same both times. We could instead calculate the result once and use it in both places. By doing this, we'll only need to do one computation for each level in the tree, eliminating all the redundancy. We can make use of `let` to allow us to easily reuse the value:

```
(define mod-expt
  (lambda (base exponent modulus)
    (define mod*
      (lambda (x y)
        (remainder (* x y) modulus)))
    (if (= exponent 0)
        1
        (if (even? exponent)
            (let ((x (mod-expt base (/ exponent 2) modulus)))
              (mod* x x))
            (mod* (mod-expt base (- exponent 1) modulus)
                  base)))))
```

Although this is only a small change from our original tree-recursive idea, it has a dramatic impact on the order of growth of the time the algorithm takes, as illustrated in Figure 4.5. The exponent is cut in half at worst every other step, because 1 less

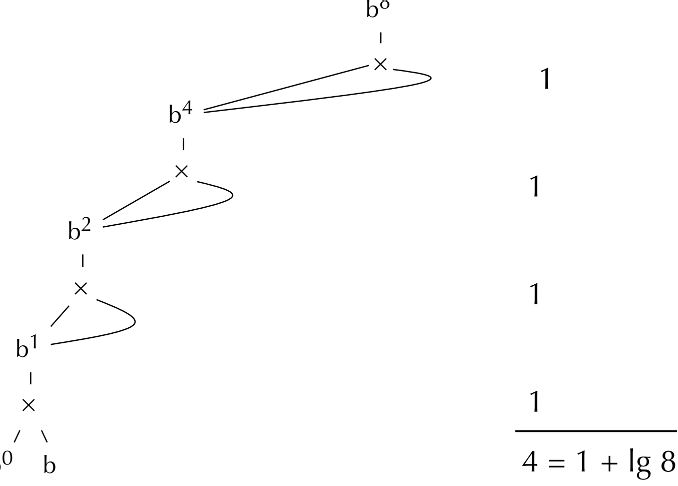


Figure 4.5 Eliminating redundant computations makes a big difference.

than an odd number is an even number. Therefore, the number of steps (and time taken) is $\Theta(\log e)$. Because a logarithmic function grows *much* more slowly than a linear one, the computation of `(mod-expt 7 signing-exponent modulus)` can now be done in about 7 seconds on our own modest computer, as opposed to 10^{180} years on an amazingly fast one. To give you some appreciation for the immense factor by which the computation has been speeded up, consider that the speed of a ... no, even *we* are at a loss for a physical analogy this time.

Exercise 4.3

Write a procedure that, given the exponent, will compute how many multiplications this latest version of `mod-expt` does.

Although we now have a version of `mod-expt` that takes $\Theta(\log e)$ time and uses $\Theta(\log e)$ memory, both of which are quite reasonable, we could actually do one better and reduce the memory consumption to $\Theta(1)$ by developing an iterative version of the procedure that still cuts the problem size in half. In doing so, we'll be straying even one step further from the original tree-recursive version, which is now serving as only the most vague source of motivation for the algorithm. To cut the problem size in half with a recursive process, we observed that when e was even, $b^e = (b^{e/2})^2$. To cut the problem size in half but generate an iterative process, we can instead observe that when e is even, $b^e = (b^2)^{e/2}$. This is the same as recognizing that when e is even, the string of e b's multiplied together can be divided into $e/2$ pairs of b's multiplied together, rather than two groups containing $e/2$ each.

Exercise 4.4

Develop a logarithmic time iterative version of `mod-expt` based on this concept.

At this point you have seen how an important practical application that involves very large problem sizes can be turned from impossible to possible by devising an algorithm with a lower order of growth. In particular, we successively went through algorithms with the following growth rates:

- $\Theta(e)$ space and time (linear recursion)
- $\Theta(1)$ space and $\Theta(e)$ time (linear iteration)
- $\Theta(\log e)$ space and $\Theta(e)$ time (tree recursion)
- $\Theta(\log e)$ space and time (logarithmic recursion)
- $\Theta(1)$ space and $\Theta(\log e)$ time (logarithmic iteration)

4.3 An Application: Fractal Curves

The tree-recursive `mod-expt` turned out not to be such a good idea because the two half-sized problems were identical to one another, so it was redundant to solve each of them separately. By contrast, the tree-recursive merge sort makes sense, because the two half-sized problems are distinct, although similar. Both are problems of the form "sort these $n/2$ cards," but the specific cards to sort are different. This typifies the situation in which tree recursion is natural: when the problem can be broken into two (or more) equal-sized subproblems that are all of the same general form as the original but are distinct from one another.

Fractal curves are geometric figures that fit this description; we say that they possess *self-similarity*. Each fractal curve can be subdivided into a certain number of subcurves, each of which is a smaller version of the given curve. Mathematicians are interested in the case where this subdividing process continues forever so that the subcurves are quite literally identical to the original except in size and position. Because we can't draw an infinitely detailed picture on the computer screen, we'll stop the subdivision at some point and use a simple geometric figure, such as a line or triangle, as the basis for the curve. We call that basis the *level 0* curve; a level 1 curve is composed out of level 0 curves, a level 2 curve is composed out of level 1 curves, and so forth.

As a first example, consider the fractal curve in Figure 4.6, known as *Sierpinski's gasket*. As indicated in Figure 4.7, the gasket contains three equally sized subgaskets, each of which is a smaller version of the larger gasket. Figure 4.8 shows Sierpinski's gaskets of levels 0, 1, and 2.

A level n Sierpinski's gasket is composed of three smaller Sierpinski's gaskets of level $n - 1$, arranged in a triangular fashion. Furthermore, the level 0 Sierpinski's gasket is itself a triangle. Therefore, triangles play two different roles in Sierpinski's

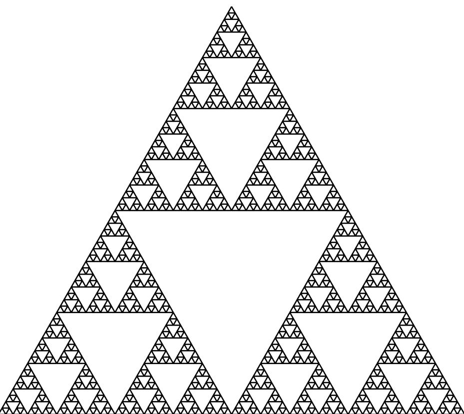


Figure 4.6 An example of Sierpinski's gasket.

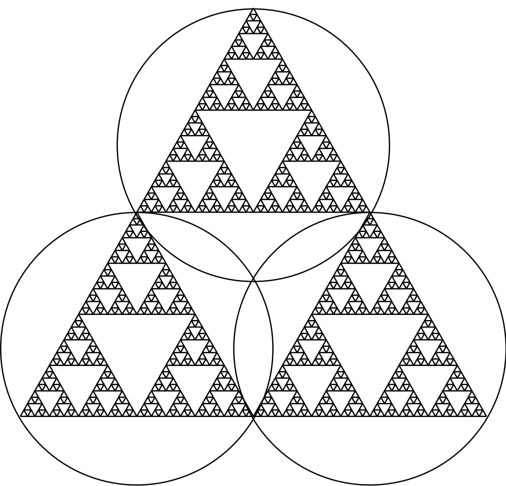


Figure 4.7 An example of Sierpinski's gasket, with the three subgaskets circled.

gasket: the *self-similarity* (i.e., the composition out of lower-level components) is triangular, and the *basis* (i.e., level 0 curve) is also triangular.

In some fractals the self-similarity may differ from the basis. As an example, consider the so-called *c-curve,* which is displayed in Figure 4.9 at levels 6 and 10. The basis of a c-curve is just a straight line. A level n curve is made up of two level $n-1$ c-curves, but the self-similarity is somewhat difficult to detect. We will describe this self-similarity by writing a procedure that produces c-curves. To write this procedure, we'll need to write a procedure that takes five arguments. The first four are the x and y coordinates of the starting and ending points, the fifth is the level of the curve. A level 0 c-curve is simply a line from the starting point, say $(x0, y0)$, to

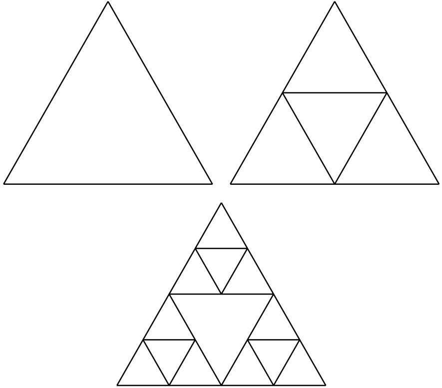


Figure 4.8 Sierpinski's gaskets of levels 0, 1, and 2.

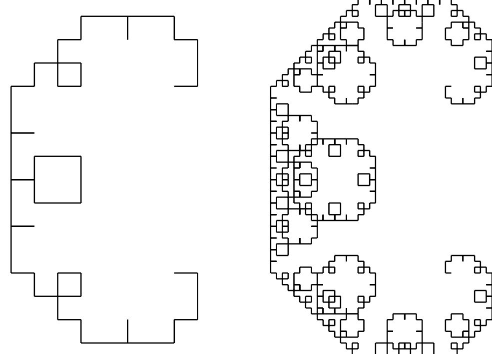


Figure 4.9 C-curves of levels 6 and 10.

the ending point, say $(x1, y1)$. This is what the built-in procedure `line` will produce. For higher level c-curves, we need to join two subcurves together at a point that we'll call (xa, ya). Figure 4.10 illustrates the relationship between the three points; the two subcurves go from point 0 to point *a* and then from point *a* to point 1.

```
(define c-curve
  (lambda (x0 y0 x1 y1 level)
    (if (= level 0)
        (line x0 y0 x1 y1)
        (let ((xmid (/ (+ x0 x1) 2))
              (ymid (/ (+ y0 y1) 2))
              (dx (- x1 x0))
              (dy (- y1 y0)))
          (let ((xa (- xmid (/ dy 2)))
                (ya (+ ymid (/ dx 2))))
            (overlay (c-curve x0 y0 xa ya (- level 1))
                     (c-curve xa ya x1 y1 (- level 1))))))))
```

Try out the `c-curve` procedure with various values for the parameters in order to gain an understanding of their meaning and the visual effect resulting from changing their values. Overlaying two or more c-curves can help you understand what is going on. For example, you might try any (or all) of the following:

```
(c-curve 0 -1/2 0 1/2 0)
(c-curve 0 -1/2 0 1/2 1)
(c-curve 0 -1/2 0 1/2 2)
(c-curve 0 -1/2 0 1/2 3)
(c-curve 0 -1/2 0 1/2 4)
```

(x1, y1)

(xa, ya)

(x0, y0)

Figure 4.10 The three key points in a c-curve of level greater than zero.

```
(c-curve -1/2 0 0 1/2 3)
(c-curve 0 -1/2 -1/2 0 3)
(overlay (c-curve -1/2 0 0 1/2 3)
         (c-curve 0 -1/2 -1/2 0 3))

(c-curve 0 -1/2 0 1/2 6)
(c-curve 0 -1/2 0 1/2 10)

(c-curve 0 0 1/2 1/2 0)
(c-curve 0 0 1/2 1/2 4)
(c-curve 1/2 1/2 0 0 4)
```

Exercise 4.5

A c-curve from point 0 to point 1 is composed of c-curves from point 0 to point *a* and from point *a* to point 1. What happens if you define a d-curve similarly but with the direction of the second half reversed, so the second half is a d-curve from point 1 to point *a* instead?

Exercise 4.6

Using the procedure `c-curve` as a model, define a procedure called `length-of-c-curve` that, when given the same arguments as `c-curve`, returns the length of the path that would be traversed by a pen drawing the c-curve specified.

Exercise 4.7

See what numbers arise when you evaluate the following:

```
(length-of-c-curve 0 -1/2 0 1/2 0)
(length-of-c-curve 0 -1/2 0 1/2 1)
(length-of-c-curve 0 -1/2 0 1/2 2)
(length-of-c-curve 0 -1/2 0 1/2 3)
(length-of-c-curve 0 -1/2 0 1/2 4)
```

Do you see a pattern? Can you mathematically show that this pattern holds?

Exercise 4.8

C-curves can be seen as more and more convoluted paths between the two points, with increasing levels of detours on top of detours. The net effect of any c-curve of any level still is to connect the two endpoints. Design a new fractal that shares this property. That is, a level 0 curve should again be a straight line, but a level 1 curve should be some different shape of detour path of your own choosing that connects up the two endpoints. What does your curve look like at higher levels?

Exercise 4.9

We will now turn to Sierpinski's gasket. To get this started, write a procedure called `triangle` that takes six arguments, namely the *x* and *y* coordinates of the triangle's three vertices. It should produce an image of the triangle. Test it with various argument values; `(triangle -1 -.75 1 -.75 0 1)` should give you a large and nearly equilateral triangle.

Exercise 4.10

Use `triangle` to write a procedure called `sierpinskis-gasket` that again takes six arguments for the vertex coordinates but also takes a seventh argument for the level of curve.

Review Problems

Exercise 4.11

Consider the following procedures:

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))

(define factorial-sum1 ; returns 1! + 2! + ... + n!
  (lambda (n)
    (if (= n 0)
        0
        (+ (factorial n)
           (factorial-sum1 (- n 1))))))

(define factorial-sum2 ; also returns 1! + 2! + ... + n!
  (lambda (n)
    (define loop
      (lambda (k fact-k addend)
        (if (> k n)
            addend
            (loop (+ k 1)
                  (* fact-k (+ k 1))
                  (+ addend fact-k)))))
    (loop 1 1 0)))
```

In answering the following, assume that n is a nonnegative integer. Also, justify your answers.

a. Give a formula for how many multiplications the procedure `factorial` does as a function of its argument n.

b. Give a formula for how many multiplications the procedure `factorial-sum1` does (implicitly through `factorial`) as a function of its argument n.

c. Give a formula for how many multiplications the procedure `factorial-sum2` does as a function of its argument n.

Exercise 4.12

How many ways are there to factor n into two or more numbers (each of which must be no smaller than 2)? We could generalize this to the problem of finding how many ways there are to factor n into two or more numbers, each of which is no smaller than m. That is, we write

```
(define ways-to-factor
  (lambda (n)
    (ways-to-factor-using-no-smaller-than n 2)))
```

Your job is to write `ways-to-factor-using-no-smaller-than`. Here are some questions you can use to guide you:

- If $m^2 > n$, how many ways are there to factor n into two or more numbers each no smaller than m?
- Otherwise, consider the case that n is not divisible by m. Compare how many ways are there to factor n into two or more numbers no smaller than m with how many ways there are to factor n into two or more numbers no smaller than $m + 1$. What is the relationship?
- The only remaining case is that $m^2 \leq n$ and n is divisible by m. In this case, there is at least one way to factor n into numbers no smaller than m. (It can be factored into m and n/m.) There may, however, be other ways as well. The ways of factoring n divide into two categories: those using at least one factor of m and those containing no factor of m. How many factorizations are there in each category?

Exercise 4.13

Consider the following procedure:

```
(define bar
  (lambda (n)
    (cond ((= n 0) 5)
          ((= n 1) 7)
          (else (* n (bar (- n 2)))))))
```

How many multiplications (expressed in Θ notation) will the computation of `(bar n)` do? Justify your answer. You may assume that n is a nonnegative integer.

Exercise 4.14

Consider the following procedures:

```
(define foo
  (lambda (n)        ; computes n! + (n!)^n
    (+ (factorial n) ; that is, (n! plus n! to the nth power)
       (bar n n))))
```

```
(define bar
  (lambda (i j)    ; computes (i!)^j (i! to the jth power)
    (if (= j 0)
        1
        (* (factorial i)
           (bar i (- j 1))))))

(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

How many multiplications (expressed in Θ notation) will the computation of `(foo n)` do? Justify your answer.

Exercise 4.15

Suppose that you have been given n coins that look and feel identical and you've been told that exactly one of them is fake. The fake coin weighs slightly less than a real coin. You happen to have a balance scale handy, so you can figure out which is the fake coin by comparing the weights of various piles of coins. One strategy for doing this is as follows:

- If you only have 1 coin, it must be fake.
- If you have an even number of coins, you divide the coins into two piles (same number of coins in each pile), compare the weights of the two piles, discard the heavier pile, and look for the fake coin in the remaining pile.
- If you have an odd number of coins, you pick one coin out, divide the remaining coins into two piles, and compare the weights of those two piles. If you're lucky, the piles weigh the same and the coin you picked out is the fake one. If not, throw away the heavier pile and the extra coin, and look for the fake coin in the remaining pile.

Note that if you have one coin, you don't need to do any weighings. If you have an even number of coins, the maximum number of weighings is one more than the maximum number of weighings you'd need to do for half as many coins. If you have an odd number of coins, the maximum number of weighings is the same as the maximum number of weighings you'd need for one fewer coins.

a. Write a procedure that will determine the maximum number of weighings you need to do to find the fake coin out of n coins using the above strategy.

b. Come up with a fancier (but more efficient) strategy based on dividing the pile of coins in thirds, rather than in half. (Hint: If you compare two of the thirds, what are the possible outcomes? What does each signify?)

c. Write a procedure to determine the maximum number of weighings using the strategy based on dividing the pile in thirds.

Exercise 4.16

Perhaps you noticed in Section 4.3 that as you increase the value of the *level* parameter in `c-curve` (while keeping the starting and ending points fixed), the c-curve gets larger. Not only is the path larger, but the curve extends further to the left, further to the right, and extends higher and lower. One way to measure this growth would be to ask how far left it extends (i.e., what its minimum *x*-value is). This could be done by defining a procedure called `min-x-of-c-curve`, taking exactly the same arguments as `c-curve`, which returns the minimum *x*-value of the given c-curve.

One strategy for implementing `min-x-of-c-curve` is as follows:

- If $level = 0$, the c-curve is just a line from $(x0, y0)$ to $(x1, y1)$, so return the smaller of $x0$ and $x1$.
- If $level \geq 1$, the given c-curve is built from two c-curves, each of which has $level - 1$ as its level. One goes from $(x0, y0)$ to (xa, ya), and the other from (xa, ya) to $(x1, y1)$. Therefore, you should return the smaller of the min-*x*-values of these two sub-c-curves.

Write the procedure `min-x-of-c-curve`. As an aid in writing it, note that there is a built-in procedure `min` that returns the smaller of its arguments. So we would have

```
(min 1 3)       (min 2 -3)       (min 4 4)
1               -3               4
```

As a hint, note that `min-x-of-c-curve` can be structured in a manner very similar to both `c-curve` and `length-of-c-curve`.

Exercise 4.17

Consider the following enumeration problem: How many ways can you choose k objects from n distinct objects, assuming of course that $0 \leq k \leq n$? For example,

how many different three-topping pizzas can be made if you have six toppings to choose from?

The number that is the answer to the problem is commonly written as $C(n, k)$. Here is an algorithm for computing $C(n, k)$:

- As noted above, you may assume that $0 \leq k \leq n$, because other values don't make sense for the problem.
- The base cases are $k = 0$ and $k = n$. It should not be too hard to convince yourself that $C(n, n)$ should equal 1, and similar reasoning can be used to show that $C(n, 0) = 1$ is also the reasonable choice.
- The general case is $0 < k < n$. Here you might argue as follows: Consider one of the objects. Either you select it as one of the k objects, or you don't. If you do select it, then you must select $k - 1$ more objects from the remaining $n - 1$, presumably a simpler problem that you assume you can do recursively. If on the other hand you don't select the first object, you must select k objects from the remaining $n - 1$, which is also a simpler problem whose value is computed recursively. Then the total number of ways to select k objects from these n objects is the sum of the numbers you get from these two subproblems.

Using this algorithm, write a tree-recursive procedure that calculates the numbers $C(n, k)$ described above.

▷ Exercise 4.18

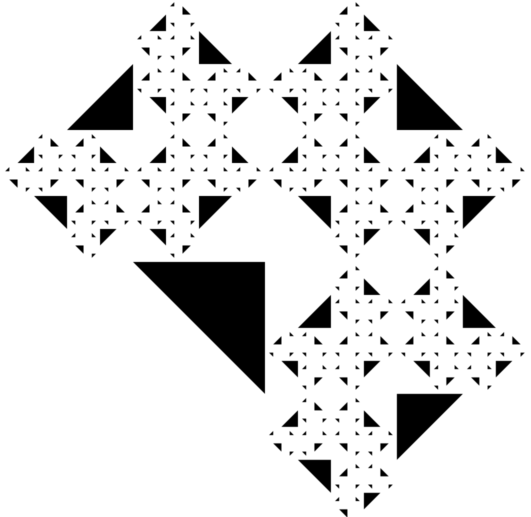
One way to sum the integers from a up to b is to divide the interval in half, recursively sum the two halves, and then add the two sums together. Of course, it may not be possible to divide the interval exactly in half if there are an odd number of integers in the interval. In this case, the interval can be divided as nearly in half as possible.

a. Write a procedure implementing this idea.

b. Let's use n as a name for the number of integers in the range from a up to b. What is the order of growth (in Θ notation) of the number of additions your procedure does, as a function of n? Justify your answer.

▷ Exercise 4.19

The following illustration shows a new kind of image, which we call a *tri-block*:



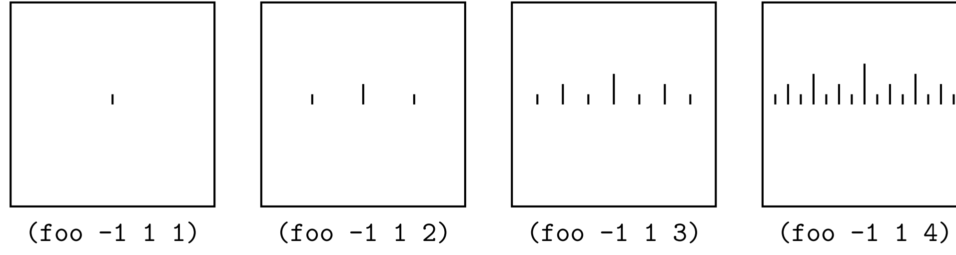
What kind of process do you think was used to generate this image (i.e., was it linear recursive, iterative, or tree recursive)? Write a paragraph carefully explaining *why* you think this.

Exercise 4.20

Consider the following procedure:

```
(define foo
  (lambda (low high level)
    (let ((mid (/ (+ low high) 2)))
      (let ((mid-line (line mid 0 mid (* level .1))))
        (if (= level 1)
            mid-line
            (overlay mid-line
                     (overlay (foo low mid (- level 1))
                              (foo mid high (- level 1))))))))))
```

Examples of the images produced by this procedure are given below:



(foo -1 1 1) (foo -1 1 2) (foo -1 1 3) (foo -1 1 4)

a. What kind of process does `foo` generate (i.e., linear recursive, iterative, or tree recursive)? Justify your answer.

b. Let's call the number of lines in the image `foo` produces $l(n)$, where n is the level. Make a table showing $l(n)$ versus n for $n = 1, 2, 3, 4$. Write a mathematical equation showing how $l(n)$ can be computed from $l(n - 1)$ when n is greater than 1. Explain how each part of your equation relates to the procedure. What is $l(5)$?

Chapter Inventory

Vocabulary

algorithm
selection sort
merge sort
Θ (big theta)
order of growth
logarithm
tree recursion
digital signature
message digest
digest function
modular arithmetic
modulus
commutative
associativity
distributivity
number system
fractal curve
self-similarity
basis (of a fractal)
Sierpinski's gasket
c-curve

Slogans

The asymptotic outlook

New Predefined Scheme Names

The dagger symbol (†) indicates a name that is not part of the R^{4}RS standard for Scheme.

`line`†
`min`

Scheme Names Defined in This Chapter

```
verify
mod+
mod*
mod-
gold-num
signature
modulus
signing-exponent
mod-expt
c-curve
length-of-c-curve
triangle
sierpinskis-gasket
factorial-sum1
```

`factorial-sum2`
`ways-to-factor`
`ways-to-factor-using-no-smaller-than`
`min-x-of-c-curve`

Sidebars

Selection Sort
Merge Sort
Merging
Logarithms
Modular Arithmetic

Notes

The definitive work on sorting algorithms is by Knuth [31]. Knuth reports that merge sort is apparently the first program written for a stored program computer, in 1945.

Our definition of Θ allowed "finitely many exceptions." Other books normally specify that any number of exceptions are allowed, provided that they are all less than some cutoff point. The two definitions are equivalent given our assumption that n is restricted to be a nonnegative integer. If that restriction is lifted, the more conventional definition is needed. In any case, the reason for allowing exceptions to the bounds is to permit the use of bounding functions that are ill-defined for small n. For example, we showed that merge sort was $\Theta(n \log n)$. When $n = 0$, the logarithm is undefined, so we can't make a claim that the time to sort 0 cards is bounded between two positive constant multiples of $0 \log 0$.

The digital signature method we've described is known as the *RSA cryptosystem*, named after the initials of its developers: Ron Rivest, Adi Shamir, and Leonard Adleman. The way we were able to produce our public modulus and secret signing key is as follows. We randomly chose two 100-digit primes, which we've kept secret; call them p and q. We made sure $(p-1)(q-1)$ wasn't divisible by 3. Our modulus is simply the product pq. This means that in principle anyone could discover p and q by factoring our published modulus. However, no one knows how to factor a 200-digit number in any reasonable amount of time. Our secret signing exponent is calculated using p and q. It is the multiplicative inverse of 3 (the verification exponent), mod $(p-1)(q-1)$. That is, it is the number that when multiplied by 3 and then divided by $(p-1)(q-1)$ leaves a remainder of 1. For an explanation of why this works, how the inverse is quickly calculated, and how to find large primes, consult Cormen, Leiserson, and Rivest's superb *Introduction to Algorithms* [14]. For general information on the RSA system, there is a useful publication from RSA Laboratories by Paul Fahn [16]. We should point out that the use of the RSA system for digital signatures is probably covered by several patents; however, the relevant patent holders have indicated that they won't prosecute anyone using the system as an educational exercise. Also, it is worth mentioning that the export from the United

States of any product employing the RSA system is regulated by the International Traffic in Arms Regulation.

There has been a recent explosion of books on fractals, aimed at all levels of audience. Two classics by Mandelbrot, who coined the word *fractal*, are [36] and [37].

Some aspects of our treatment of fractals, such as the c-curve example and the `length-of-c-curve` exercise, are inspired by a programming assignment developed by Abelson, Sussman, and friends at MIT [1].