8 Time

Time makes more converts than reason.
Thomas Paine

The previous chapter introduced notations for conveniently describing the asymptotic growth rates of functions. Chapter 6 introduced the Turing Machine model and argued that it can simulate any reasonable computer. In this chapter, we combine the computing model with the asymptotic operators to reason about the running time of procedures.

The first section of this chapter explains how to measure input sizes and running times. Section 8.2 provides examples of procedures with different growth rates. The growth rate of a procedure's running time gives us an understanding of how the running time increases as the size of the input increases. In the next chapter, we provide an extended example.

8.1 Measuring Running Time

To understand the growth rate of a procedure's running time, we need a function that maps the size of the inputs to the procedure to the amount of time it takes to evaluate the application. First we consider how to measure the input size; then, we consider how to measure the running time. In Section 8.1.3 we consider *which* input of a given size should be used to reason about the cost of applying a procedure.

8.1.1 Input Size

Procedure inputs may be many different types: Numbers, Lists of Numbers, Lists of Lists, Procedures, etc. Our goal is to characterize the input size with a single number that does not depend on the types of the input.

We use the Turing machine to model a computer, so the way to measure the size of the input is the number of characters needed to write done the input

on the tape. The characters can be from any fixed-size alphabet, such as the ten decimal digits, or the letters of the alphabet. As we saw in Chapter 1, we can use a sequence of bits to uniquely represent a member of any finite set.

The number of different symbols in the tape alphabet does not matter for our analysis, since we are concerned with orders of growth, not with absolute values. Within the O, Ω, and Θ operators, a constant factor does not matter. For example, $\Theta(n)$ defines exactly the same set as $\Theta(17n + 523)$. This means whether we use an alphabet with two symbols or an alphabet with 256 symbols, the effective input size is not impacted; with two symbols the input may be 8 times as long as it is with a 256-symbol alphabet, but the constant factor of 8 does not matter in the asymptotic operator.

Thus, we measure the size of the input is the number of symbols required to write the number on a Turing Machine input tape. To figure out the input size of a given type, we need to think about how many symbols it would require to write down inputs of that type.

Booleans. There are only two Boolean values: true and false. Hence, we can encode each value using a single symbol. The length of a Boolean input is fixed.

Numbers. Using the decimal number system (that is, 10 tape symbols), we can write a number of magnitude n using $\log_{10} n$ digits. Using the binary number system (that is, 2 tape symbols), we can write it using $\log_2 n$ digits. Withing the asymptotic operators, the base of the logarithm does not matter (as long as it is a constant). We can see this from the argument above — changing the number of symbols in the input alphabet changes the input length by a constant factor which has no impact within the asymptotic operators. Another way to see this is to use the formula for change logarithm bases:

$$\log_b x = \frac{1}{\log_a b} \cdot \log_a x$$

Changing the base of the logarithm multiplies the value by $\frac{1}{\log_a b}$, a constant factor if both logarithm bases are constants.

Lists. If the input is a List, the size of the input is related to the number of elements in the list. If each element is a constant size (for example, a list of numbers where each number is between 0 and 100), then the size of the input list is some constant multiple of the number of elements in the list. Hence, the size of an input that is a list of n elements is cn for some constant c. Since $\Theta(cn) = \Theta(n)$, the size of a List input is $\Theta(n)$ where n is the number of elements in the List. If List elements can vary in size, then we need to account for that in the input size. For example, suppose the

input is a List of Lists, where there are n elements in each inner List, and there are n List elements in the main List. Then, there are n^2 total elements, so the input size is in $\Theta(n^2)$.

8.1.2 Running Time

As discussed in Section 7.1, timing the actual time of particular executions of a procedure does not, by itself, provide a very useful way to understand important properties of the cost of a procedure. Instead, we want a measure of the running time of a procedure that satisfies two properties: (1) it should be robust to ephemeral properties of a particular execution or computer, and (2) it should provide insights into how long it will take to evaluate the procedure on a wide range of inputs.

To estimate the running time of an evaluation, we need to consider the number of steps required to perform the evaluation. The actual number of steps depends on the details of how much work can be done on each step. If we count instructions on a particular processor, the amount of work that can be done in one instruction varies. Further, not all instructions take the same amount of time of time to execute. Even the same simple instruction, such as reading the value in some memory location, may take vastly different amounts of time depending on the state of the machine.[1]

When we analyze procedures, however, we usually don't want to deal with these details. Instead, what we care about is how the running time changes as the input size increases. This means we can count anything we want as a "step" as long as each step is the approximately same size (that is, the time a step requires does not depend on the size of the input).

The clearest and simplest definition of a step is to use one Turing Machine step. We have a precise definition of exactly what a Turing Machine can do in one step: it can read the symbol in the current square, write a symbol into that square, transition its internal state number, and move one square to the left or right. Counting Turing Machine steps is very precise, but difficult because we do not usually start with a Turing Machine description of a procedure and creating one could be tedious.

Instead, we can usually reason directly from a Scheme procedure (or any precise description of a procedure) using larger steps. As long as we can claim that whatever we consider a step could be simulated using a constant number of steps on a Turing Machine, our larger steps will produce

[1] If the same memory location has been read recently, it is likely to be available in a memory cache that is built-in to the processor, so the access time is very quick. If it has not been read recently, it may take several thousand times as long to read it.

the same answer within the asymptotic operators. One possibility when we are analyzing a Scheme procedure is to count the number of times an evaluation rule is used in an evaluation of an application of the procedure. The amount of work in each evaluation rule may vary slightly (for example, the evaluation rule for an if-expression seems more complex than the rule for a primitive) but does not depend on the input size.

Hence, it is reasonable to assume all the evaluation rules to take constant time. This does not include any additional evaluation rules that are needed to apply one rule. For example, the evaluation rule for application expressions includes evaluating every subexpression. Evaluating an application constitutes one work unit for the application rule itself, plus all the work required to evaluate the subexpressions. In cases where the bigger steps are unclear, we can always return to our precise definition of a step as one step of a Turing Machine.

8.1.3 Worst Case Input

A procedure may have different running times for inputs of the same size. One example is this procedure that takes a List as input and outputs the first positive number in the list:

```
(define (list-first-pos p)
  (if (null? p)
      (error "No positive element found")
      (if (> (car p) 0)
          (car p)
          (list-first-pos (cdr p)))))
```

If the first element in the input list is positive, evaluating the application of *list-first-pos* requires very little work. It is not necessary to consider any other elements in the list if the first element is positive. On the other hand, if none of the elements are positive, the procedure needs to test each element in the list until it reaches the end of the list (where the base case reports an error).

In our analyses we usually consider the *worst case* input. This is the input of a given size for which evaluating the procedure takes the most work. By focusing on the worst case input, we know the maximum running time for the procedure. Without knowing something about the possible inputs to the procedure, it is safest to be pessimistic about the input and not assume any properties that are not known (such as that the first number in the list is positive for the *first-pos* example).

In some cases, we need to also consider the *average case* input. Since most

procedures can take infinitely many inputs, this requires understanding the distribution of possible inputs to determine an "average" input. This is often necessary when we are analyzing the running time of a procedure that uses another helper procedure. If we use the worst-case running time for the helper procedure, we will grossly overestimate the running time of the main procedure. Instead, since we know how the main procedure uses the helper procedure, we can more precisely estimate the actual running time by considering the actual inputs. We see an example of this in the analysis of how the + procedure is used by *list-length* in Section 8.2.2.

8.2 Growth Rates

Our goal is to understand how the running time of an application of a procedure is related to the size of the input. To do this, we want is to devise a function that predicts the running time of the procedure application. That function should take as input a number that represents the size of the input, and produces as output a number that gives the maximum number of steps required to complete the evaluation on an input of that size. Symbolically, we can think of this function as:

$$\textit{Number-Of-Steps}_{\textit{Procedure}}\colon \textit{Number} \rightarrow \textit{Number}$$

Because the output represents the *maximum* number of steps required, we need to consider the worst-case input of the given size.

Because of all the issues with counting steps exactly, and the uncertainty about how much work can be done in one step on a particular machine, we cannot usually determine the exact function for $\textit{Number-Of-Steps}_{\textit{Procedure}}$. Instead, we characterize the running time of a procedure with a set of functions produce by an asymptotic operator. Inside the O, Ω, and Θ, the actual time needed for each step does not matter since the constant factors are hidden by the operator; what matters is how the number of steps required grows as the size of the input grows.

Hence, we will characterize the running time of a procedure using a set of functions produced by one of the asymptotic operators. The Θ operator provides the most information. Recall that $O(f)$ is the set of functions that grow *no faster* than f, $\Omega(f)$ is the set of functions that grow *no slower* than f, and $\Theta(f)$ is the set of functions that grow *as fast as* f. Since $\Theta(f)$ is the intersection of $O(f)$ and $\Omega(f)$, knowing that the running time of a procedure is in $\Theta(f)$ for some function f provides much more information than just knowing it is in $O(f)$ or just knowing that it is in $\Omega(f)$. Hence, our goal is

to characterize the running time of a procedure using the set of functions defined by $\Theta(f)$ of some function f.

The rest of this section provides examples of procedures with different growth rates, from slowest (no growth) through increasingly rapid growth rates. The growth classes described are important classes that are commonly encountered when analyzing procedures, but these are only examples of growth classes. Between each pair of classes described here, there are an unlimited number of different growth classes.

8.2.1 No Growth: Constant Time

If the running time of a procedure does not increase when the size of the input increases, it means the procedure must be able to produce its output by looking at only a constant number of symbols in the input.

constant time

Procedures whose running time does not increase with the size of the input are known as *constant time* procedures. Their running time is in $O(1)$ — it does not grow at all. By convention, we use $O(1)$ instead of $\Theta(1)$ to describe constant time. Since there is no way to grow slower than no growth, $O(1)$ and $\Theta(1)$ are equivalent.

We cannot do much in constant time, since we cannot even examine the whole input. A constant time procedure must be able to produce its output by examining only a fixed-size part of the input. Recall that the input size measures the number of squares needed to represent the input. A constant time procedure can look at no more than C squares on the tape where C is some constant. If the input is larger than C, a constant time procedure can not even read parts of the input.

List procedures. An example of a constant time procedure is the built-in procedure *car*. When *car* is applied to a non-empty list, it evaluates to the first element of that list. No matter how long the input list is, all the *car* procedure needs to do is extract the first component of the list. So, the running time of *car* is in $O(1)$.[2]

Other built-in procedures that involve lists and pairs that have running times in $O(1)$ include *cons*, *cdr*, *null?*, and *pair?*. None of these procedures need to examine more than the first pair of the list.

[2]Since we are speculating based on what *car* does, not examining how *car* a particular Scheme interpreter actually implements it, we cannot say definitively that its running time is in $O(1)$. It would be rather shocking, however, for an implementation to implement *car* in a way such that its running time that is not in $O(1)$. The implementation of *scar* in Section 5.2.1 is constant time: regardless of the input size, evaluating an application of it involves evaluating a single application expression, and then evaluating an if-expression.

Boolean procedures. All sensible procedures that have only Booleans as inputs are constant time. This is because the size of a Boolean input is a constant. Since there is no way to grow the size of a Boolean input, the running time of a procedure that takes only Boolean inputs should not grow either. For example, the *logical-and* procedure from Chapter 6 takes two Boolean inputs. Evaluating it involves evaluating one if-expression, so the running time is constant.

Number procedures. Most procedures that take numbers are inputs are not constant time, since most operations on numbers depend on the whole number. For example, there is no way to correctly add two numbers without completely examining all the digits in both numbers. There are some operations we can do on numbers in constant time, however.

For example, the *zero?* procedure takes a number as its input and outputs a Boolean indicating whether or not the input number is zero. Whether it is possible to implement *zero?* in constant time depends on how numbers are represented. If numbers are represented with no leading zeros (that is, 0043 is not a valid number), then all *zero?* needs to do is look at the first digit, so it can be done in constant time. We will more examples in the next section involving arithmetic operations where one of the numbers is a constant.

8.2.2 Linear Growth

When the running time of a procedure increases by a constant amount when the size of the input grows by one, the running time of the procedure grows *linearly* with the input size. If the input size is n, the running time is in $\Theta(n)$. If a procedure has running time in $\Theta(n)$, doubling the size of the input will approximately double the execution time. *linearly*

Many procedures that take a List as input have linear time growth. A procedure that does something that takes constant time with every element in the input List, has running time that grows linearly with the size of the input since adding one element to the list increases the number of steps by a constant amount. Next, we examine three example list procedures, all of which have running times that scale linearly with the size of their input.

Example 8.1: Append. Consider the *list-append* procedure (from Example 5.6):

```
(define (list-append p q)
  (if (null? p) q
      (cons (car p) (list-append (cdr p) q))))
```

We want to understand how the running time of *list-append* scales with the size of its input. Since *list-append* takes two inputs, we need to be careful about how we refer to the input size. We use n_p to represent the number of elements in the first input, and n_q to represent the number of elements in the second input.

To analyze the running time of *list-append*, we examine its body which is an if-expression. The predicate expression applies the *null?* procedure. As we argued in the previous section, *null?* can be applied in constant time since the effort required to determine if a list is *null* does not depend on the length of the list. When the predicate expression evaluates to true, the alternate expression is just q, which can also be evaluated in constant time.

Next, we consider the alternate expression. It includes a recursive application of *list-append*. Hence, the running time of the alternate expression is the time required to evaluate the recursive application plus the time required to evaluate everything else in the expression. The other expressions to evaluate are applications of *cons*, *car*, and *cdr*, each of which is a constant time procedure.

So, we could express the total running time as,

$$\textit{Running-Time}_{\textit{list-append}}(n_p, n_q) = C + \textit{Running-Time}_{\textit{list-append}}(n_p - 1, n_q)$$

where C is some constant (that is, the time for all the operations besides the recursive call does not depend on the length of any of the inputs). Note that the value of n_q does not matter, so we can simplify this to:

$$\textit{Running-Time}_{\textit{list-append}}(n_p) = C + \textit{Running-Time}_{\textit{list-append}}(n_p - 1).$$

This indicates that the running time to evaluate an application of *list-append* when the first input is a list with n_p elements is the time required to evaluate *list-append* on an input with one fewer element plus some constant.

This does not yet provide a useful characterization of the running time of *list-append* though, since it is a circular definition. To make it a recursive definition, we need a base case. The base case for the running time definition is the same as the base case for the procedure: when the input is *null*. For the base case, the running time is constant. So, we can also define:

$$\textit{Running-Time}_{\textit{list-append}}(0) = C_0$$

where C_0 is some constant (not necessarily the same as C).

To better characterize the running time of *list-append*, we want a closed form solution. For a given input n, *Running-Time*(n) is $C + C + C + C + \ldots + C + C_0$ where there are $n - 1$ of the C terms in the sum. This simplifies to

$(n-1)C + C_0 = nC - C + C_0 = nC + C_2$. We do not know what the values of C and C_2 are, but since we use the asymptotic notations to describe the running time, it doesn't matter. The important thing is that the running time scales linearly with the value of its input. Thus, the running time of *list-append* is in $\Theta(n_p)$ where n_p is the number of elements in the first input.

Usually, we do not need to reason at quite this low a level. Instead, to analyze the running time of a recursive procedure it is enough to determine the amount of work involved in each recursive call (excluding the recursive application itself) and multiply this by the number of recursive calls. For this example, there are n_p recursive calls since each call reduces the length of the p input by one until the base case is reached. Each call involves only constant-time procedures (other than the recursive application), so the amount of work involved in each call is constant. Hence, the running time is in $\Theta(n_p)$. This is equivalent to stating that the running time for the *list-append* procedure scales linearly with the length of the first input list.

Example 8.2: Length. Consider the *list-length* procedure from Chapter 5:

```
(define (list-length p)
  (if (null? p)
      0
      (+ 1 (list-length (cdr p)))))
```

This procedure makes one recursive application of *list-length* for each element in the input p. If the input has n elements, there will be $n + 1$ total applications of *list-length* to evaluate (one for each element, and one for the *null*). So, the total work is in $\Theta(n \cdot \text{work for each recursive application})$.

To determine the running time, we need to determine how much work is involved in each application. Evaluating an application of *list-length* consists of evaluating its body, which is an if-expression. To evaluate the if-expression, the predicate expression, (*null? p*), must be evaluated first. This requires constant time since the *null?* procedure has constant running time (see Section 8.2.1).

If the predicate expression evaluates to true, the consequent expression must be evaluated. It is the primitive expression, 0, which can be evaluated in constant time.

When the predicate expression evaluates to false, the alternate expression, (+ 1 (*list-length* (*cdr p*))) is evaluated. We analyze the running time of evaluating this expression without including the recursive *list-length* application. The reason we do not include this know is that we know there are $n + 1$ total applications of *length* to evaluate. Once we know the running time for each application (other than the recursive application itself), we can just

multiply this by the number of recursive calls to get the total running time.

The remaining work is evaluating (*cdr p*) and evaluating the + application. The *cdr* procedure is constant time. Analyzing the running time of the + procedure application is more complicated.

Since + is a built-in procedure, we need to think about how it might be implemented. Following the elementary school addition algorithm (from Section 6.2.3), we know we can add any two numbers by walking down the digits. The work required for each digit is constant; we just need to compute the corresponding result and carry bits using a simple formula or lookup table. The number of digits to add is the maximum number of digits in the two input numbers. Thus, if there are b digits to add, the total work is in $\Theta(b)$. In the worst case, we need to look at all the digits in both numbers. In general, we cannot do asymptotically better than this, since adding two arbitrary numbers might require looking at all the digits in both numbers.

But, in the *list-length* procedure the + is used in only a very limited way: (+ 1 (*list-length* (*cdr p*))). One of the inputs is always 1. We might be able to add 1 to a number without looking at all the digits in the number. Recall the addition algorithm: we start at the rightmost (least significant) digit, add that digit, and continue with the carry. If one of the input numbers is 1, then once the carry is zero we know now of the more significant digits will need to change. In the worst case, adding one requires changing every digit in the other input. For example, (+ 99999 1) is 100000. In the best case (when the last digit is below 9), adding one requires only examining and changing one digit.

Worst Case

Figuring out the average case is more difficult, but necessary to get a good estimate of the running time of *list-length*. We assume the numbers are represented in binary, so instead of decimal digits we are counting bits (this is both simpler, and closer to how numbers are actually represented in the computer). Approximately half the time, the least significant bit is a 0, so we only need to examine one bit. When the last bit is not a 0, we need to examine the second least significant bit (the second bit from the right): if it is a 0 we are done; if it is a 1, we need to continue. So, we need to examine two or more bits in the case where the least significant bit is a 1.

Thus, we always need to examine one bit, the least significant bit. Half the time we need to also examine the second least significant bit. Of those times, half the time we need to continue and examine the next least significant bit. This continues through the whole number. Thus, the expected number of bits we need to examine is,

$$1+\frac{1}{2}\left(1+\frac{1}{2}\left(1+\frac{1}{2}\left(1+\frac{1}{2}\left(1+\ldots\right)\right)\right)\right)$$

where the number of terms is the number of bits in the input number, b. Simplifying the equation, we get:

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots + \frac{1}{2^b}$$

No matter how large b gets, this value is always less than 2. So, on average, the number of bits to examine to add 1 is constant: it does not depend on the length of the input number. Although adding two arbitrary values cannot be done in constant time, adding 1 to an arbitrary value can be done in constant time.

This result generalizes to addition where one of the inputs is any constant. One way to see this is that adding any constant C to a number n is equivalent to adding one C times. Since adding one is a constant time procedure, adding one C times can also be done in constant time for any constant C. Another way to see this is to note that the constant C can be written in $\log_2 C$ bits. Hence, we always need to look at the rightmost $\log_2 C$ bits of the input number, but only need to look beyond those bits when there are carries. The frequency of carries after the next most significant bit is the same as if we are adding 1. So, on average, we only need to look at a constant number of bits to add a constant to any number.

Excluding the recursive application, the *list-length* application involves applications of two constant time procedures: *cdr* and adding one using $+$. Hence, the total time needed to evaluate one application of *list-length*, excluding the recursive application, is constant. It does not depend on the length of the input list.

There are $n + 1$ total applications of *list-length* to evaluate total, so the total running time is $c(n + 1)$ where c is the amount of time needed for each application. The set $\Theta(c(n + 1))$ is identical to the set $\Theta(n)$, so we can say that the running time for the length procedure is in $\Theta(n)$ where n is the length of the input list.

Example 8.3: Accessing List Elements. Consider the *list-get-element* procedure from Example 5.3:

```
(define (list-get-element p n)
    (if (= n 1)
        (car p)
        (list-get-element (cdr p) (- n 1))))
```

The procedure takes two inputs, a List and a Number selecting the element of the list to get. Since there are two inputs, we need to think carefully about the input size. We can use variables to represent the size of each input, for

example s_p and s_n for the size of p and n respectively. In this case, however, we will see that only the size of the first input really matters.

The procedure body is an if-expression. The predicate uses the built-in $=$ procedure to compare n to 1. The worst case running time of the $=$ procedure is linear in the size of the input: it potentially needs to look at all bits in the input numbers to determine if they are equal. Similarly to $+$, however, if one of the inputs is a constant, the comparison can be done in constant time. To compare a number of any size to 1, it is enough to look at a few bits. If the least significant bit of the input number is not a 1, we know the result is false. If it is a 1, we need to examine a few other bits of the input number to determine if its value is different from 1 (the exact number of bits we need to look at depends on how numbers are represented in more detail; if there are no leading zeros, which is not the case with most computer representations, it would only be necessary to examine one more bit). So, the $=$ comparison can be done in constant time.

If the predicate is true, the base case applies the *car* procedure, which has constant running time. The alternate expression involves the recursive calls, as well as evaluating (*cdr p*), which requires constant time, and ($-$ n 1). The $-$ procedure is similar to $+$: for arbitrary inputs, its worst case running time is linear in the input size, but when one of the inputs is a constant the running time is constant. This follows from a similar argument to the one we used for the $+$ procedure. To subtract a constant of length C bits, we need to examine C bits of the input number, and further bits only some of the time (Exercise 4 asks for a more detailed analysis of the running time of subtraction). So, the work required for each recursive call is constant.

The number of recursive calls is determined by the value of n and the number of elements in the list p. In the best case, when n is 1, there are no recursive calls and the running time is constant. It does not depend on the number of elements in the list, since the procedure only needs to examine the first element. Each recursive call reduces the value passed in as n by 1, so the number of recursive calls scales linearly with n (the actual number is $n - 1$ since the base case is when n equals 1). But, there is a limit on the value of n for which this is true. If the value passed in as n exceeds the number of elements in p, the procedure will produce an error when it attempts to evaluate (*cdr p*) for the empty list. This happens after s_p recursive calls, where s_p is the number of elements in p. Hence, the running time of *list-get-element* does not grow with the length of the input passed as n; after the value of n exceeds the number of elements in p it does not matter how much bigger it gets, the running time does not continue to increase.

Thus, the worst case running time of *list-get-element* grows linearly with the

length of the input list. Equivalently, the running time of *list-get-element* is in $\Theta(s_p)$ where s_p is the number of elements in the input list.

Exercise 8.1. Explain why the *list-map* procedure from Section 5.4.1 has running time that is linear in the size of its List input. Assume the procedure input has constant running time.

Exercise 8.2. Consider the *list-sum* procedure (from Example 5.2):

```
(define (list-sum p)
  (if (null? p)
      0
      (+ (car p) (list-sum (cdr p)))))
```

What assumptions are needed about the elements in the list for the running time to be linear in the number if elements in the input list?

Exercise 8.3. For the decimal six-digit odometer (shown in the picture on page 8-10), we measure the amount of work to add one as the total number of wheel digit turns required. For example, going from 000000 to 000001 requires one work unit, but going from 000099 to 000100 requires three work units.

Voting Machine Counter

a. What are the worst case inputs?

b. What are the best case inputs?

c. [⋆] On average, how many work units are required for each mile? Assume over the lifetime of the odometer, the car travels 1,000,000 miles.

d. Lever voting machines were used by the majority of American voters in the 1960s, although they are not widely used today. Most level machines used a three-digit odometer to tally votes. Explain why candidates ended up with 99 votes on a machine far more often than 98 or 100 on these machines.

Exercise 8.4. [⋆] The *list-get-element* argued by comparison to +, that the − procedure has constant running time when one of the inputs is a constant. Develop a more convincing argument why this is true by analyzing the worst case and average case inputs for −.

Exercise 8.5. [⋆] Our analysis of the work required to add one to a number argued that it could be done in constant time. Test experimentally if the DrScheme + procedure actually satisfies this property. Note that one + application is too quick to measure well using the *time* procedure, so you will need to design a procedure that applies + many times without doing much other work.

8.2.3 Quadratic Growth

quadratically

If the running time of a procedure scales as the square of the size of the input, the procedure's running time grows *quadratically*. Doubling the size of the input approximately quadruples the running time. The running time is in $\Theta(n^2)$ where n is the size of the input.

A procedure that takes a list as input has running time that grows quadratically if it goes through all elements in the list once for every element in the list. For example, we can compare every element in a list of length n with every other element using $n(n-1)$ comparisons. This simplifies to $n^2 - n$, but $\Theta(n^2 - n)$ is equivalent to $\Theta(n^2)$ since as n increases only the highest power term matters (see Exercise 7.7).

Example 8.4: Reverse. Consider the *list-reverse* procedure defined in Section 5.4.2:

```
(define (list-reverse p)
  (if (null? p)
      null
      (list-append (list-reverse (cdr p)) (list (car p)))))
```

To determine the running time of *list-reverse*, we need to know how many recursive calls there are and how much work is involved in each recursive call. Each recursive application passes in (*cdr p*) as the input, so reduces the length of the input list by one. Hence, applying *list-reverse* to a input list with n elements involves n recursive calls.

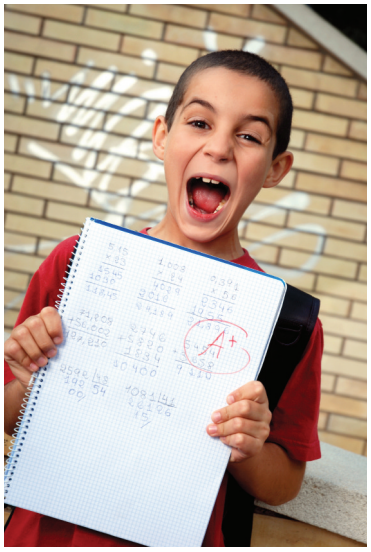
The work for each recursive application, excluding the recursive call itself, is applying *list-append*. The first input to *list-append* is the output of the recursive call. As we argued in Example 8.1, the running time of *list-append* is in $\Theta(n_p)$ where n_p is the number of elements in its first input. So, to determine the running time we need to know the length of the first input list to *list-append*. For the first call, (*cdr p*) is the parameter, with length $n-1$; for the second call, there will be $n-2$ elements; and so forth, until the final call where (*cdr p*) has 0 elements. So, the total number of elements in all of these calls is:

$$(n-1) + (n-2) + \ldots + 1 + 0.$$

The average number of elements in each call is approximately $\frac{n}{2}$. Within the asymptotic operators the constant factor of $\frac{1}{2}$ does not matter, so the average running time for each recursive application is in $\Theta(n)$.

There are n recursive applications, so the total running time of *list-reverse* is n times the average running time of each recursive application: $n \cdot \Theta(n) = \Theta(n^2)$. Thus, the running time is quadratic in the size of the input list.

Example 8.5: Multiplication. Consider the problem of multiplying two numbers. The elementary school long multiplication algorithm works by multiplying each digit in b by each digit in a, aligning the intermediate results in the right places, and summing the results:



$$
\begin{array}{cccccccc}
 & & & & a_{n-1} & \cdots & a_1 & a_0 \\
 & & & \times & b_{n-1} & \cdots & b_1 & b_0 \\
\hline
 & & & & a_{n-1}b_0 & \cdots & a_1b_0 & a_0b_0 \\
 & & & a_{n-1}b_1 & \cdots & a_1b_1 & a_0b_1 & \\
 & + & a_{n-1}b_{n-1} & \cdots & a_1b_{n-1} & a_0b_{n-1} & & \\
\hline
r_{2n-1} & & r_{2n-2} & \cdots & \cdots & r_3 & r_2 & r_1 & r_0
\end{array}
$$

If both input numbers have n digits, there are n^2 digit multiplications, each of which can be done in constant time. The intermediate results will be n rows, each containing n digits. So, the total number of digits to add is n^2: 1 digit in the ones place, 2 digits in the tens place, ..., n digits in the 10^{n-1}s place, ..., 2 digits in the 10^{2n-3}s place, and 1 digit in the 10^{2n-2}s place. Each digit addition requires constant work, so the total work for all the digit additions is in $\Theta(n^2)$. Adding the work for both the digit multiplications and the digit additions, the total running time for the elementary school multiplication algorithm is quadratic in the number of input digits, $\Theta(n^2)$ where n is the number if digits in the inputs.

This is not the fastest known algorithm for multiplying two numbers, although it was the best algorithm known until 1960. In 1960, Anatolii Karatsuba discovers a multiplication algorithm with running time in $\Theta(n^{\log_2 3})$. Since $\log_2 3 < 1.585$ this is an improvment over the $\Theta(n^2)$ elementary school algorithm. In 2007, Martin Fürer discovered an even faster algorithm for multiplication [FÖ7]. It is not yet known if this is the fastest possible multiplication algorithm, or if faster ones exist.

Exercise 8.6. [⋆] Analyze the running time of the elementary school long division algorithm.

Exercise 8.7. [⋆] Define a Scheme procedure that multiplies two multi-digit numbers (without using the built-in $*$ procedure except to multiply single-digit numbers). Strive for your procedure to have running time in $\Theta(n)$ where n is the total number of digits in the input numbers.

Exercise 8.8. [⋆⋆⋆⋆] Devise a multiplication algorithm that is faster than Fürer's, or prove that no faster algorithm for general multiplication exists.

8.2.4 Polynomial Growth

A *polynomial* function is a function that is the sum of powers of one or more variables multiplied by coefficients. For example,

$$a_k n^k + ... + a_3 n^3 + a_2 n^2 + a_1 n + a_0$$

is a polynomial where the variable is n and the coefficients are $a_0, a_1, ..., a_k$. Within the O, Ω and Θ operators, only the term with the largest exponent matters. For high enough n, all the other terms become insignificant. An easy way to see this is to observe that n^k is $n \cdot n^{k-1}$. So, whatever the a_{k-1} coefficient is, once $n > a_{k-1}$ the value of $n^k > a_{k-1} n^{k-1}$. So, the polynomial above is in $\Theta(n^k)$ which is equivalent to $\Theta(a_k n^k + ... + a_3 n^3 + a_2 n^2 + a_1 n + a_0)$ for all positive constants $a_0, \ldots, a_k$.

All of the slower growth rates in this section are polynomials: constant growth is a polynomial where the highest exponent is 0, linear growth is a polynomial where the highest exponent is 1, and quadratic grow is a polynomial where the highest exponent is 2. We can have procedures where the running time grows as a higher polynomial. The next example analyzes a procedure whose running time grows cubically with the size of its input.

Example 8.6: Satisfying pair. The *list-satisfying-pair* procedure, defined below, takes as inputs a test procedure and a list. The test procedure is a procedure that takes two inputs and produces a Boolean value. If there are any pairs of elements in the list for which the test procedure evaluates to true, the output is a pair of two elements in the list for which the test procedure evaluates to true. Otherwise, the output is false to indicate there is no satisfying pair.

Here are some example evaluations:

```
> (list-satisfying-pair = (list 1 3 2 4 6 1))
(1 . 1)
> (list-satisfying-pair = (list 1 3 2 4 6 7))
false
> (list-satisfying-pair (lambda (a b) (= a (* b 2)))
(list 2 3 5 7 11 13))
false
> (list-satisfying-pair (lambda (a b) (= a (* b b)))
(list 3 4 5 6 7 8 9 10 11))
(9 . 3)
```

We can define *list-satisfying-pair* by using *list-get-element* to select elements in turn to try all possible pairs.

```
(define (list-satisfying-pair cf p)
  (define (list-satisfying-pair-iter i j)
    (if (> i (list-length p))
        false
        (if (> j (list-length p))
            (list-satisfying-pair-iter (+ i 1) 1)
            (if (and (not (= i j))
                     (cf (list-get-element p i) (list-get-element p j)))
                (cons (list-get-element p i) (list-get-element p j))
                (list-satisfying-pair-iter i (+ j 1))))))
  (list-satisfying-pair-iter 1 1))
```

The inner definition defines the *list-satisfying-pair-iter* procedure that takes two inputs, i and j, Numbers used to represent the indices of elements in the list to compare. It also uses the two parameters to the outer *list-satisfying-pair* procedure: *cf*, the Procedure used to test if a pair of inputs satisfy some property, and p, the input List.

To check all possible pairs, we need to evaluate the test procedure for every pair of values i and j that correspond to elements in the input list. We use n to represent the number of elements in the input list. Since there are n possible values for i and n possible values for j, this amounts to n^2 total comparisons. The number of recursive calls to *list-satisfying-pair-iter* should match this. The recursive definition is a bit complex since there are two different recursive calls. The first base case is when i exceeds the length of the input list. When this is true, it means all possible pairs have been tried without finding a satisfying pair, so the output is false. The second base case is when j, the second index, exceeds the length of the list. This happens when the procedure has finished trying the comparison procedure with all pairs for a given first element (the value of i). To continue, the recursive call, (*list-satisfying-pair-iter* (+ i 1) 1) passes in the next value for i and resets the value for the j parameter to 1. This happens n times.

The other recursive call is done when (> j (*list-length* p)) evaluates to false: (*list-satisfying-pair-iter* i (+ j 1)). The value passed in as i does not change, but the value passed as j increases by 1. Since j starts at 1, the number of recursive calls required before the predicate is false is n. After these n calls, the recursive call that increases i by 1 and resets j to 1 is used. Hence, the total number of recursive calls is n^2.

So, to determine the running time of *list-satisfying-pair* we need to estimate the running time of each application of *list-satisfying-pair-iter* (excluding the recursive call) and multiply that by n^2. The worst (and most common) path through *list-satisfying-pair-iter* is the path where the first two predicate expressions are false, and the third one is true. Along this path, we need

to evaluate ($>$ *i* (*list-length p*)), ($>$ *j* (*list-length p*)), (*and* (*not* ($=$ *i j*)) (*cf* (*list-get-element p i*) (*list-get-element p j*))), and (*cons* (*list-get-element p i*) (*list-get-element p j*)). We consider the running time of each expression in turn; the total running time for each *list-satisfying-pair-iter* application is the sum of the running times of each expression.

The first two expressions use the $>$ procedure to compare the index value to the length of the list. The running time of *list-length* is linear in the number of elements in the input list, so this involves $\Theta(n)$ work. The less than comparison can be done by examining the bits of the input numbers from most-significant to least-significant bit. In the worst case, all bits in the input numbers need to be examined. This occurs when all the bits are the same, or all the bits except the last bit are the same. The value of the first input is some number between 1 and $n+1$; the value of the second input is always n. Hence, the size of the input to $<$ is $\Theta(\log n)$ and the running time for evaluating each $>$ expression is in $\Theta(\log n)$. The total running time for evaluating ($>$ *i* (*list-length p*)) is $\Theta(n + \log n)$ which is equivalent to $\Theta(n)$ since as n increases, the $\log n$ term becomes much smaller than n (that is, $1.1n > (n + \log n)$ for $n > 100$). The running time for the second comparison expression is also in $\Theta(n)$.

The third expression involves applications of the Boolean procedure *not*, which is constant time since it takes Boolean inputs; the comparison procedure, *cf*, which we assume is constant time; and two applications of *list-get-element*, which we analyzed in Example 8.3 to have running time that is linear in the number of elements in the input List in the worst case. The running time for *list-get-element* scales linearly with the value of the second input, up to the length of the first input. On average, the values of i and j will be the average length of the list = $n/2$, so the expected running time of each call is also in $\Theta(n)$. This means the total running time for *list-satisfying-pair* is in $\Theta(n^3)$ since there are $\Theta(n^2)$ recursive calls, and each one has running time in $\Theta(n)$. So, the running time for this expression is in $O(1) + O(1) + \Theta(n) + \Theta(n) = \Theta(n)$.

The fourth expression, (*cons* (*list-get-element p i*) (*list-get-element p j*)), also involves two applications of *list-get-element* and the constant time procedures *cons*. Hence, its running time is in $\Theta(n)$.

The total running time to evaluate all four expressions is the sum of their running times: $\Theta(n) + \Theta(n) + \Theta(n) + \Theta(n)$, which is equivalent to $\Theta(n)$. Since there are n^2 recursive calls, the total running time for *list-satisfying-pair* is in $n^2 \cdot \Theta(n) = \Theta(n^3)$.

Exercise 8.9. [⋆⋆] The provided definition of *list-satisfying-pair* has cubic running time, but it is possible to implement a procedure with the same behavior that has running time in $\Theta(n^2)$ where n is the number of ele-

ments in the input list. Write a definition of *list-satisfying-pair* that has quadratic running time, and explain why the running time of your procedure is quadratic.

8.2.5 Exponential Growth

If the running time of a procedure scales as some power of the size of the input, the procedure's running time grows *exponentially*. When the size of the input increases by one, the running time is multiplied by some constant factor. The growth rate of a function whose output is multiplied by w when the input size, n, increases by one is w^n.

A common instance of exponential growth is when the running time is in $\Theta(2^n)$. This occurs when the running time doubles when the input size increases by one. Exponential growth is very fast — if our procedure has running time that is exponential in the size of the input, it is not feasible to evaluate applications of the procedure on large inputs.

For a surprisingly large number of interesting problems, the best known algorithm has exponential running time. Examples of problems like this include finding the best route between two locations on a map (the problem mentioned in Chapter 4), the pegboard puzzle from Chapter 5, solving generalized versions of most other games such as Suduko and Minesweeper, and finding the factors of a number. Whether or not it is possible to design faster algorithms that solve these problems is the most important open problem in omputer science, which we will return to in Chapter 17.

Next, we provide one example of a procedure that has exponential running time. In this case, we know there is no faster than exponential solution, since the size of the output is exponential in the size of the input. Since the most work a Turing Machine can do in one step is write one square, we know if the size of the output to a problem is exponential in the of its input the fastest possible procedure for that problem has exponential running time.

Example 8.7: Power Set. In mathematics, the power set of a set S is the set of all subsets of S. For example, the power set of $\{1, 2, 3\}$ is

$$\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

The number of elements in the power set of S is $2^{|S|}$ (where $|S|$ is the number of elements in the set S).

Here is a procedure that takes a list as input, and produces as output the power set of the elements of the list (unlike mathematical sets we allow duplicate values in our input list, and the resulting list-sets):

```
(define (list-powerset s)
  (if (null? s)
      (list null)
      (list-append (list-map (lambda (t) (cons (car s) t)) (list-powerset (cdr s)))
                   (list-powerset (cdr s)))))
```

The *list-powerset* procedure produces a List of Lists. Hence, for the base case, instead of just producing *null*, it produces a list containing a single element, *null*. In the recursive case, we can produce the power set by appending the list of all the subsets that include the first element, with the list of all the subsets that do not include the first element. For example, the powerset of $\{1, 2, 3\}$ is found by finding the powerset of $\{2, 3\}$, which is {{}, {2}, {3}, {2, 3}}, and taking the union of that set, and the set of all elements in that set unioned with $\{1\}$.

An application of *list-powerset* involves applying *list-append*, and two recursive applications of (*list-powerset* (*cdr s*)). Increasing the size of the input list by one, *doubles* the total number of applications of *list-powerset*. This is the case because to evaluate (*list-powerset s*), we need to evaluate (*list-powerset* (*cdr s*)) twice. So, the number of applications of *list-powerset* is 2^n where n is the length of the input list.[3]

The body of *list-powerset* is an if-expression. The predicate is a constant-time procedure, *null?*. The consequent expression, (*list null*) is also constant time. The alternate expression is an application of *list-append*. From Example 8.1, we know the running time of *list-append* is $\Theta(n_p)$ where n_p is the number of elements in its first input. The first input is the result of applying *list-map* to a procedure and the List produced by (*list-powerset* (*cdr s*)). The length of the list output by *list-map* is the same as the length of its input, so we need to determine the length of (*list-powerset* (*cdr s*)).

We use n_s to represent the number of elements in s. The length of the input list to *map* is the number of elements in the power set of a size $n_s - 1$ set: 2^{n_s-1}. But, for each application, the value of n_s is different. Since we are trying to determine the total running time, we can do this by thinking about the total length of all the input lists to *list-map* over all of the *list-powerset*. In the input is a list of length n, the total list length is:

$$2^{n-1} + 2^{n-2} + ... + 2^1 + 2^0$$

which is equal to $2^n - 1$. So, the running time for all the *list-map* applications is in $\Theta(2^n)$.

[3] Careful readers will note that it is not really necessary to perform this evaluation twice, since we could do it once and reuse the result. Later, we explain how to avoid needing two evaluations of (*list-powerset* (*cdr s*)).

The analysis of the *list-append* applications is similar. The length of the first input to *list-append* is the length of the result of the *list-powerset* application, so the total length of all the inputs to append is 2^n.

Other than the applications of *list-map* and *list-append*, the rest of each *list-powerset* application requires constant time. So, the running time required for 2^n applications is in $\Theta(2^n)$. The total running time for *list-powerset* is the sum of the running times for the *list-powerset* applications, in $\Theta(2^n)$; the *list-map* applications, in $\Theta(2^n)$; and the *list-append* applications, in $\Theta(2^n)$. Hence, the total running time is in $\Theta(2^n)$.

8.2.6 Faster than Exponential Growth

We have already seen an example of a procedure that grows faster than exponentially in the size of the input: the *fibo* procedure at the beginning of this chapter! Evaluating an application of *fibo* involves $\Theta(\phi^n)$ recursive applications where n is the *value* of the input parameter. The size of a numeric input is the number of digits needed to express it, so the value n can be as high as $10^d - 1$ where d is the number of digits. Hence, the running time of the *fibo* procedure is in $\Theta(\phi^{10^d})$. This is why we are still waiting for (*fibo* 60) to finish evaluating.

8.2.7 Non-terminating Procedures

All of the procedures so far in the section are algorithms: they may be slow, but they are guaranteed to always finish. Some procedures never terminate. For example,

```
(define (run-forever) (run-forever))
```

defines a procedure that never finishes. Its body calls itself, never making any progress toward a base case. The running time of this procedure is effectively infinite since it never finishes.

Exercise 8.10. Analyze the running time of the *intsto* procedure (from Example 5.8):

```
(define (intsto n)
  (if (= n 0) null (list-append (intsto (- n 1)) (list n))))
```

Be careful to describe the running time in terms of the *size* (not the magnitude) of the input.

Exercise 8.11. Analyze the running time of the *factorial* procedure (from Example 4.1):

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

Exercise 8.12. Analyze the running time of the *board-replace-peg* procedure (from Example 5.11):

```
(define (row-replace-peg pegs col val)
  (if (= col 1)
      (cons val (cdr pegs))
      (cons (car pegs) (row-replace-peg (cdr pegs) (- col 1) val))))


(define (board-replace-peg board row col val)
  (if (= row 1)
      (cons (row-replace-peg (car board) col val) (cdr board))
      (cons (car board) (board-replace-peg (cdr board) (- row 1) col val))))
```

Exercise 8.13. [⋆] Find and correct at least one error in the *Orders of Growth* section of the Wikipedia page on *Analysis of Algorithms* (*http://en.wikipedia.org/wiki/Analysis_of_algorithms*). This is rated as [⋆] now (9 February 2009), since the current entry contains many fairly obvious errors. Hopefully it will soon become a [⋆ ⋆ ⋆] challenge, and perhaps, eventually will become impossible!

8.3 Summary

Because the speed of computers varies, and the exact time required for a particular application depends on many details, the most important property to capture is how the amount of work required to evaluate the procedure scales with the size of the input. The asymptotic operator for measuring orders of growth provide a convenient way of understanding the cost involved in evaluating an application of a procedure.

Procedures that can produce an output only touching a fixed amount of input regardless of the input size have constant ($O(1)$) running times. Procedures whose running time increases by a constant amount when the input size increases by one have linear (in $\Theta(n)$) running times. Procedures whose running time quadruples when the input size doubles have quadratic (in $\Theta(n^2)$) running times. Procedures whose running time is multiplied by a constant factor when the input size increases by one have

exponential (in $\Theta(k^n)$ for some constant k) running times. If a procedure has exponential running time, it can only be evaluated for small inputs.

The asymptotic analysis, however, must be considered carefully. For large enough inputs, a procedure with running time in $\Theta(n)$ is always faster than a procedure with running time in $\Theta(n^2)$. But, for an input of a particular size, the $\Theta(n^2)$ procedure may be faster. The asymptotic operators provide useful ways for reasoning about the running times of procedures, but they hide many details including constant factors. Without knowing the constants that are hidden by the asymptotic operators, there is no way to acurately predict the actual running time on a given input.

In this chapter, we have focused on analyzing the running time of a known procedure. A deeper question concerns the running time of the best possible procedure that solves a particular problem. We explore that question in Chapter 17.