7 Cost

A LISP programmer knows the value of everything, but the cost of nothing.
Alan Perlis

I told my dad that someday I'd have a computer that I could write programs on. He said that would cost as much as a house. I said, "Well, then I'm going to live in an apartment."
Steve Wozniak

This chapter develops tools for reasoning about the cost of evaluating a given expression. Predicting the cost of executing a procedure has practical value (for example, we can estimate how much computing power is needed to solve a particular problem or decide between two possible implementations), but also provides deep insights into the nature of procedures and problems.

The most commonly used cost metric is time. Other measures of cost include the amount of memory needed and the amount of energy consumed. Indirectly, these costs can often be translated into money: the rate of transactions a service can support, or the price of the computer needed to solve a problem.

7.1 Empirical Measurements

We can measure the cost of evaluating a given expression empirically. If we are primarily concerned with time, we could just use a stopwatch to measure the evaluation time. For more accurate results, we use the built-in (*time Expression*) special form.[1] Evaluating (*time Expression*) produces the value of the input expression, but also prints out the time required to evaluate the expression (shown in our examples using *slanted* font). It prints out three time values:

cpu time
: The time in milliseconds the processor ran to evaluate the expression. CPU is an abbreviation for "central processing unit", the computer's main processor.

real time
: The actual time in milliseconds it took to evaluate the expression. Since other processes may be running on the computer while this expression is evaluated, the real time may be longer than the CPU time, which only counts the time the processor was working on evaluating this expression.

[1] The *time* construct must be a special form, since the expression is not evaluated before entering *time* as it would be with the normal application rule. If it were evaluated normally, there would be no way to time how long it takes to evaluate, since it would have already been evaluated before *time* is applied.

gc time
: The time in milliseconds the interpreter spent on garbage collection to evaluate the expression. Garbage collection is used to reclaim memory that is storing data that will never be used again.

For example, using the definitions from Chapter 5,

```
(time (solve-pegboard (board-remove-peg (make-board 5)
                                        (make-position 1 1))))
```

prints: *cpu time: 141797 real time: 152063 gc time: 765*. The real time is 152 seconds, meaning this evaluation took just over two and a half minutes. Of this time, the evaluation was using the CPU for 142 seconds, and the garbage collector ran for less than one second.

Here are two more examples:

```
> (time (car (list-append (intsto 1000) (intsto 100))))
cpu time: 531 real time: 531 gc time: 62
1
> (time (car (list-append (intsto 1000) (intsto 100))))
cpu time: 609 real time: 609 gc time: 0
1
```

The two expressions evaluated are identical, but the reported time varies. Even on the same computer, the time needed to evaluate the same expression varies. Many properties unrelated to our expression (such as where things happen to be stored in memory) impact the actual time needed for any particular evaluation. Hence, it is dangerous to draw conclusions about which procedure is faster based on a few timings.

Another limitation of this way of measuring cost is it only works if we wait for the evaluation to complete. If we try an evaluation and it has not finished after an hour, say, we have no idea if the actual time to finish the evaluation is sixty-one minutes or a quintillion years. We could wait another minute, but if it still hasn't finished we don't know if the execution time is sixty-two minutes or a quintillion years. The techniques we develop allow us to predict the time an evaluation needs without waiting for it to execute.

There's no sense in being precise when you don't even know what you're talking about.
John von Neumann

Finally, measuring the time of a particular application of a procedure does not provide much insight into how long it will take to apply the procedure to different inputs. We would like to understand how the evaluation time scales with the size of the inputs so we can understand which inputs the procedure can sensibly be applied to, and can choose the best procedure to use for different situations. The next section introduces mathematical tools that are helpful for capturing how cost scales with input size.

Exercise 7.1. Suppose you are defining a procedure that needs to append two lists, one short list, *short* and one very long list, *long*, but the order of elements in the resulting list does not matter. Is it better to use (*list-append short long*) or (*list-append long short*)? (A good answer will involve both experimental results and an analytical explanation.)

Exploration 7.1: Multiplying Like Rabbits

Filius Bonacci was an Italian monk and mathematician in the 12th century. He published a book, *Liber Abbaci*, on how to calculate with decimal numbers that introduced Hindu-Arabic numbers to Europe (replacing Roman numbers) along with many of the algorithms for doing arithmetic we learn in elementary school. It also included the problem for which *Fibonacci* numbers are named:[2]

Filius Bonacci

> *A pair of newly-born male and female rabbits are put in a field. Rabbits mate at the age of one month and after that procreate every month, so the female rabbit produces a new pair of rabbits at the end of its second month. Assume rabbits never die and that each female rabbit produces one new pair (one male, one female) every month from her second month on. How many pairs will there be in one year?*

We can define a function that gives the number of pairs of rabbits at the beginning of the n^{th} month as:

$$\mathit{Fibonacci}(n) = \begin{cases} 1 & : \quad n = 1 \\ 1 & : \quad n = 2 \\ \mathit{Fibonacci}(n-1) + \mathit{Fibonacci}(n-2) & : \quad n > 1 \end{cases}$$

The third case follows from Bonacci's assumptions: all the rabbits alive at the beginning of the previous month are still alive (the $\mathit{Fibonacci}(n-1)$ term), and all the rabbits that are at least two months old reproduce (the $\mathit{Fibonacci}(n-2)$ term).

The sequence produced is known as the Fibonacci sequence:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

After the first two 1s, each number in the sequence is the sum of the previous two numbers. Fibonacci numbers occur frequently in nature, such as the arrangement of florets in the sunflower (34 spirals in one direction and 55 in the other) or the number of petals in common plants (typically 1, 2, 3, 5, 8, 13, 21, or 34), hence the rarity of the four-leaf clover.

Translating the definition of the *Fibonacci* function into a Scheme procedure is straightforward; we combine the two base cases using the *or* special form:

```
(define (fibo n)
  (if (or (= n 1) (= n 2)) 1
      (+ (fibo (- n 1)) (fibo (- n 2)))))
```

Applying *fibo* to small inputs works fine:

```
> (time (fibo 10))
cpu time: 0 real time: 0 gc time: 0
55
> (time (fibo 30))
cpu time: 2156 real time: 2187 gc time: 0
832040
```

[2] Although the sequence is named for Bonacci, it was probably not invented by him. The sequence was already known to Indian mathematicians with whom Bonacci studied.

But when we try to determine the number of rabbits in five years by computing (*fibo* 60), our interpreter just hangs without producing a value.

The *fibo* procedure is defined in a way that guarantees it eventually completes when applied to a non-negative whole number: each recursive call reduces the input by 1 or 2, so both recursive calls get closer to the base case. Hence, we always make progress and must eventually reach the base case, unwind the recursive applications, and produce a value. To understand why the evaluation of (*fibo* 60) did not finish in our interpreter, we need to consider how much work is required to evaluate the expression.

To evaluate (*fibo* 60), the interpreter follows the if expressions to the recursive case, where it needs to evaluate (+ (*fibo* 59) (*fibo* 58)). To evaluate (*fibo* 59), it needs to evaluate (*fibo* 58) again and also evaluate (*fibo* 57). To evaluate (*fibo* 58) (which needs to be done twice), it needs to evaluate (*fibo* 57) and (*fibo* 56). So, there is one evaluation of (*fibo* 60), one evaluation of (*fibo* 59), two evaluations of (*fibo* 58), and three evaluations of (*fibo* 57).

The total number of evaluations of the *fibo* procedure for each input is itself the Fibonacci sequence! To understand why, consider the evaluation tree for (*fibo* 4) shown in Figure 7.1. The only direct number values are the 1 values that result from evaluations of either (*fibo* 1) or (*fibo* 2). Hence, the number of 1 values must be the value of the final result, which just sums all these numbers. For (*fibo* 4), there are 5 leaf applications, and 3 more inner applications, for 8 (= $Fibonacci(5)$) total recursive applications. The number of evaluations of applications of *fibo* needed to evaluate (*fibo* 60) is the 61st Fibonacci number — 2,504,730,781,961 — over two and a half trillion applications of *fibo*!

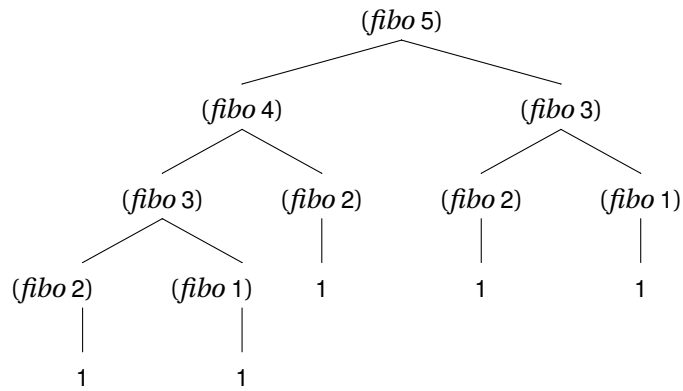


Figure 7.1. Evaluation of *fibo* procedure.

Although our *fibo* definition is *correct*, it is ridiculously inefficient and only finishes for input numbers below about 40. It involves a tremendous amount of duplicated work: for the (*fibo* 60) example, there are two evaluations of (*fibo* 58) and over a trillion evaluations of (*fibo* 1) and (*fibo* 2).

We can avoid this duplicated effort by building up to the answer starting from the base cases. This is more like the way a human would determine the numbers in the Fibonacci sequence: we find the next number by adding the previous two

numbers, and stop once we have reached the number we want.

The *fast-fibo* procedure computes the n^{th} Fibonacci number, but avoids the duplicate effort by computing the results building up from the first two Fibonacci numbers, instead of working backwards.

```
(define (fast-fibo n)
  (define (fibo-iter a b left)
    (if (<= left 0) b
        (fibo-iter b (+ a b) (- left 1))))
  (fibo-iter 1 1 (- n 2)))
```

This is a form of what is known as *dynamic programming*. The definition is still recursive, but unlike the original definition the problem is broken down differently. Instead of breaking the problem down into a slightly smaller instance of the original problem, with dynamic programming we build up from the base case to the desired solution. In the case of Fibonacci, the *fast-fibo* procedure builds up from the two base cases until reaching the desired answer. The additional complexity is we need to keep track of when to stop; we do this using the *left* parameter.

dynamic programming

The helper procedure, *fibo-iter* (short for iteration), takes three parameters: a is the value of the previous-previous Fibonacci number, b is the value of the previous Fibonacci number, and *left* is the number of iterations needed before reaching the target. The initial call to *fibo-iter* passes in 1 as a (the value of $Fibonacci(1)$), and 1 as b (the value of $Fibonacci(2)$), and $(- \; n \; 2)$ as *left* (we have $n-2$ more iterations to do to reach the target, since the first two Fibonacci numbers were passed in as a and b we are now working on $Fibonacci(2)$). Each recursive call to *fibo-iter* reduces the value passed in as *left* by one, and advances the values of a and b to the next numbers in the Fibonacci sequence.

The *fast-fibo* procedure produces the same output values as the original *fibo* procedure, but requires far less work to do so. The number of applications of *fibo-iter* needed to evaluate (*fast-fibo* 60) is now only 59. The value passed in as *left* for the first application of *fibo-iter* is 58, and each recursive call reduces the value of *left* by one until the zero case is reached. This allows us to compute the expected number of rabbits in 5 years is 1548008755920 (over 1.5 Trillion)[3].

7.2 Orders of Growth

As illustrated by the Fibonacci exploration, the same problem can be solved by procedures that require vastly different resources. The important question in understanding the resources required to evaluate a procedure application is *how the required resources scale with the size of the input.* For small inputs, both Fibonacci procedures work using with minimal resources. For large inputs, the first Fibonacci procedure never finishes, but the fast Fibonacci procedure finishes effectively instantly.

In this section, we introduce three functions computer scientists use to capture

[3] Perhaps Bonacci's assumptions are not a good model for actual rabbit procreation. This result suggests that in about 10 years the mass of all the rabbits produced from the initial pair will exceed the mass of the Earth, which, although scary, seems unlikely!

the important properties of how resources required grow with input size. Each function takes as input a function, and produces as output a set of functions:

$O(f)$ ("big oh")
: The set of functions that grow *no faster* than f grows.

$\Theta(f)$ (theta)
: The set of functions that grow *as fast* as f grows.

$\Omega(f)$ (omega)
: The set of functions that grow *no slower* than f grows.

These functions capture the asymptotic behavior of functions, that is, how they behave as the inputs get arbitrarily large. To understand how the time required to evaluate a procedure increases as the inputs to that procedure increase, we need to know the asymptotic behavior of a function that takes the size of input to the target procedure as its input and outputs the number of steps to evaluate the target procedure on that input.

Remember that accumulated knowledge, like accumulated capital, increases at compound interest: but it differs from the accumulation of capital in this; that the increase of knowledge produces a more rapid rate of progress, whilst the accumulation of capital leads to a lower rate of interest. Capital thus checks its own accumulation: knowledge thus accelerates its own advance. Each generation, therefore, to deserve comparison with its predecessor, is bound to add much more largely to the common stock than that which it immediately succeeds.
Charles Babbage, 1851

Figure 7.2 depicts the sets O, Θ, Ω for some function f. Next, we define each function and provide some examples. Section 7.3 illustrates how to analyze the time required to evaluate applications of procedures using these notations.

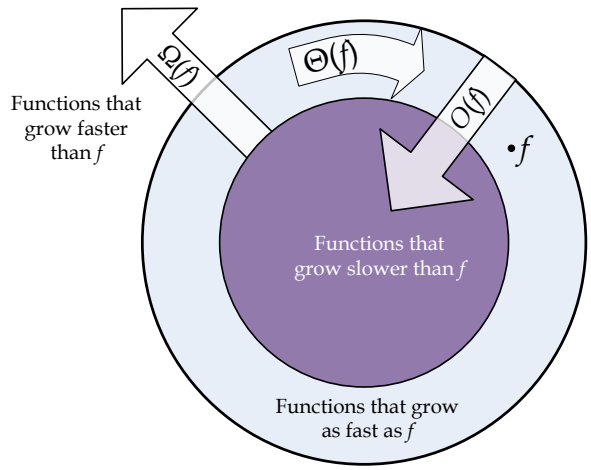


Figure 7.2. Visualization of the sets $O(f)$, $\Omega(f)$, and $\Theta(f)$.

7.2.1 Big *O*

The first notation we introduce is O, pronounced "big oh". The O function takes as input a function, and produces as output the set of all functions that grow no faster than the input function. The set $O(f)$ is the set of all functions that grow as fast as, or slower than, f grows. In Figure 7.2, the $O(f)$ set is represented by everything inside the outer circle.

To define the meaning of O precisely, we need to consider what it means for a function to *grow*. We want to capture how the output of the function increases as the input to the function increases. First, we consider a few examples; then we provide a formal definition of O.

$f(n) = n + 12$ and $g(n) = n - 7$
No matter what n value we use, the value of $f(n)$ is greater than the value of $g(n)$. This doesn't matter for the growth rates, though. What matters is how the difference between $g(n)$ and $f(n)$ changes as the input values increase. No matter what values we choose for n_1 and n_2, we know $g(n_1) - f(n_1) = g(n_2) - f(n_2) = -19$. Thus, the growth rates of f and g are identical and $n - 7$ is in the set $O(n + 12)$, and $n + 12$ is in the set $O(n - 7)$.

$f(n) = 2n$ and $g(n) = 3n$
The difference between $g(n)$ and $f(n)$ is n. This difference increases as the input value n increases, but it increases by the same amount as n increases. So, the growth rate as n increases is $\frac{n}{n} = 1$. The value of $2n$ is always within a constant multiple of $3n$, so they grow asymptotically at the same rate. Hence, $2n$ is in the set $O(3n)$ and $3n$ is in the set $O(2n)$. x

$f(n) = n$ and $g(n) = n^2$
The difference between $g(n)$ and $f(n)$ is $n^2 - n = n(n-1)$. The growth rate as n increases is $\frac{n(n-1)}{n} = n - 1$. The value of $n - 1$ increases as n increases, so g grows faster than f. This means n^2 is *not* in $O(n)$ since n^2 grows faster than n. The function n is in $O(n^2)$ since n grows slower than n^2 grows.

$f(n) = \mathit{Fibonacci}(n)$ and $g(n) = n$
The *Fibonacci* function grows very rapidly. The value of $\mathit{Fibonacci}(n+2)$ is more than *double* the value of $\mathit{Fibonacci}(n)$ since

$$\mathit{Fibonacci}(n+2) = \mathit{Fibonacci}(n+1) + \mathit{Fibonacci}(n)$$

and $\mathit{Fibonacci}(n+1) > \mathit{Fibonacci}(n)$. The rate of increase is multiplicative, and must be at least a factor of $\sqrt{2} \approx 1.414$ (since increasing by one twice more than doubles the value). (In fact, the rate of increase is a factor of $\phi = (1+\sqrt{5})/2 \approx 1.618$, also known as the "golden ratio". This is a rather remarkable result, but explaining why is beyond the scope of this book.) This is much faster than the growth rate of n, which increases by one when we increase n by one. So, n is in the set $O(\mathit{Fibonacci}(n))$, but $\mathit{Fibonacci}(n)$ is not in the set $O(n)$.

Some of the example functions are plotted in Figure 7.3. The O notation reveals the asymptotic behavior of functions. The functions plotted are the same in both graphs, but the scale of the horizontal axis is different. In the first graph, the

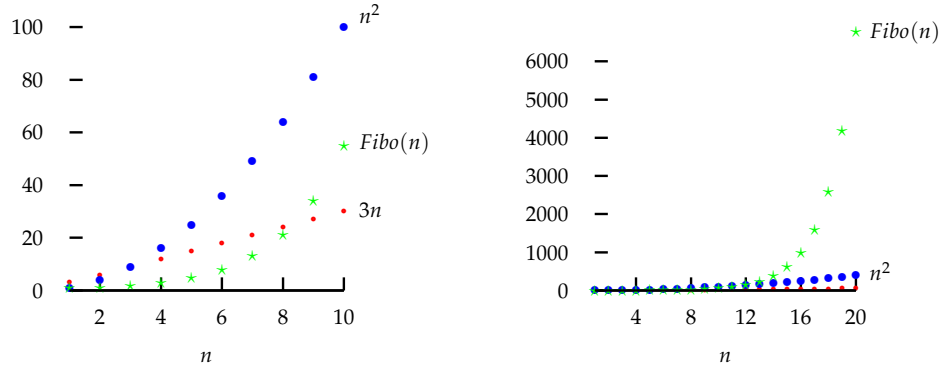


Figure 7.3. Orders of Growth.

rightmost value of n^2 is greatest; for higher input values, the value of $Fibonacci(n)$ is greatest. In the second graph, the values of $Fibonacci(n)$ for input values up to 20 are so large that the other functions appear as nearly flat lines on the graph.

Definition of O. The function g is a member of the set $O(f)$ if and only if there exist positive constants c and n_0 such that, for all values $n \geq n_0$,

$$g(n) \leq cf(n).$$

We can show g is in $O(f)$ using the definition of $O(f)$ by choosing positive constants for the values of c and n_0, and showing that the property $g(n) \leq cf(n)$ holds for all values $n \geq n_0$. To show g is not in $O(f)$, we need to explain how, for any choices of c and n_0, we can find values of n that are greater than n_0 such that $g(n) \leq cf(n)$ does not hold.

Example 7.1: O Examples

We now show the claimed properties are true using the formal definition.

$n-7$ is in $O(n+12)$
: Choose $c = 1$ and $n_0 = 1$. Then, we need to show $n - 7 \leq 1(n+12)$ for all values $n \geq 1$. This is true, since $n - 7 > n + 12$ for all values n.

$n+12$ is in $O(n-7)$
: Choose $c = 2$ and $n_0 = 26$. Then, we need to show $n + 12 \leq 2(n-7)$ for all values $n \geq 26$. The equation simplifies to $n + 12 \leq 2n - 14$, which simplifies to $26 \leq n$. This is trivially true for all values $n \geq 26$.

$2n$ is in $O(3n)$
: Choose $c = 1$ and $n_0 = 1$. Then, $2n \leq 3n$ for all values $n \geq 1$.

$3n$ is in $O(2n)$
: Choose $c = 2$ and $n_0 = 1$. Then, $3n \leq 2(2n)$ simplifies to $n \leq 4/3n$ which is true for all values $n \geq 1$.

n is in $O(n^2)$
: Choose $c = 1$ and $n_0 = 1$. Then $n \leq n^2$ for all values $n \geq 1$.

n^2 is **not** in $O(n)$
: We need to show that no matter what values are chosen for c and n_0, there are values of $n \geq n_0$ such that the inequality $n^2 \leq cn$ does not hold. For any value of c, we can make $n^2 > cn$ by choosing $n > c$.

n is in $O(Fibonacci(n))$
: Choose $c = 1$ and $n_0 = 3$. Then $n \leq Fibonacci(n)$ for all values $n \geq n_0$.

$Fibonacci(n)$ is **not** in $O(n-2)$
: No matter what values are chosen for c and n_0, there are values of $n \geq n_0$ such that $Fibonacci(n) > c(n)$. We know $Fibonacci(12) = 144$, and, from the discussion above, that:

$$Fibonacci(n+2) > 2 * Fibonacci(n)$$

This means, for $n > 12$, we know $Fibonacci(n) > n^2$. So, no matter what value is chosen for c, we can choose $n = c$. Then, we need to show

$$Fibonacci(n) > n(n)$$

The right side simplifies to n^2. For $n > 12$, we know $Fibonacci(n) > n^2$. Hence, we can always choose an n that contradicts the $Fibonacci(n) \leq cn$ inequality by choosing an n that is greater than n_0, 12, and c.

For all of the examples where g is in $O(f)$, there are many acceptable choices for c and n_0. For the given c values, we can always use a higher n_0 value than the selected value. It only matters that there is some finite, positive constant we can choose for n_0, such that the required inequality, $g(n) \leq cf(n)$ holds for all values $n \geq n_0$. Hence, our proofs work equally well with higher values for n_0 than we selected. Similarly, we could always choose higher c values with the same n_0 values. The key is just to pick any appropriate values for c and n_0, and show the inequality holds for all values $n \geq n_0$.

Proving that a function is not in $O(f)$ is usually tougher. The key to these proofs is that the value of n that invalidates the inequality is selected *after* the values of c and n_0 are chosen. One way to think of this is as a game between two adversaries. The first player picks c and n_0, and the second player picks n. To show the property that g is not in $O(f)$, we need to show that no matter what values the first player picks for c and n_0, the second player can always find a value n that is greater than n_0 such that $g(n) > cf(n)$.

Exercise 7.2. For each of the g functions below, answer whether or not g is in the set $O(n)$. Your answer should include a proof. If g is in $O(n)$ you should identify values of c and n_0 that can be selected to make the necessary inequality hold. If g is not in $O(n)$ you should argue convincingly that no matter what values are chosen for c and n_0 there are values of $n \geq n_0$ such the inequality in the definition of O does not hold.

a. $g(n) = n + 5$

b. $g(n) = .01n$

c. $g(n) = 150n + \sqrt{n}$

d. $g(n) = n^{1.5}$

e. $g(n) = n!$

Exercise 7.3. [⋆] Given f is some function in $O(h)$, and g is some function not in $O(h)$, which of the following must always be true:

a. For all positive integers m, $f(m) \leq g(m)$.

b. For some positive integer m, $f(m) < g(m)$.

c. For some positive integer m_0, and all positive integers $m > m_0$,

$$f(m) < g(m).$$

7.2.2 Omega

The set $\Omega(f)$ (omega) is the set of functions that grow no *slower* than f grows. So, a function g is in $\Omega(f)$ if g grows as fast as f or faster. Constrast this with $O(f)$, the set of all functions that grow no *faster* than f grows. In Figure 7.2, $\Omega(f)$ is the set of all functions outside the darker circle.

The formal definition of $\Omega(f)$ is nearly identical to the definition of $O(f)$: the only difference is the $\leq$ comparison is changed to $\geq$.

Definition of $\Omega(f)$. The function g is a member of the set $\Omega(f)$ if and only if there exist positive constants c and n_0 such that, for all values $n \geq n_0$,

$$g(n) \geq cf(n).$$

Example 7.2: Ω Examples

We repeat selected examples from the previous section with Ω instead of O. The strategy is similar: we show g is in $\Omega(f)$ using the definition of $\Omega(f)$ by choosing positive constants for the values of c and n_0, and showing that the property $g(n) \geq cf(n)$ holds for all values $n \geq n_0$. To show g is not in $\Omega(f)$, we need to explain how, for any choices of c and n_0, we can find a choice for $n \geq n_0$ such that $g(n) < cf(n)$.

$n-7$ is in $\Omega(n+12)$
: Choose $c = \frac{1}{2}$ and $n_0 = 26$. Then, we need to show $n - 7 \geq \frac{1}{2}(n+12)$ for all values $n \geq 26$. This is true, since the inequality simplifies $\frac{n}{2} \geq 13$ which holds for all values $n \geq 26$.

$2n$ is in $\Omega(3n)$
: Choose $c = \frac{1}{3}$ and $n_0 = 1$. Then, $2n \geq \frac{1}{3}(3n)$ simplifies to $n \geq 0$ which holds for all values $n \geq 1$.

n is not in $\Omega(n^2)$
: Whatever values are chosen for c and n_0, we can choose $n \geq n_0$ such that $n \geq cn^2$ does not hold. Choose $n > \frac{1}{c}$ (note that c must be less than 1 for the inequality to hold for any positive n, so if c is not less than 1 we can just choose $n \geq 2$). Then, the right side of the inequality cn^2 will be greater than n, and the needed inequality $n \geq cn^2$ does not hold.

n is not in $\Omega(\mathit{Fibonacci}(n))$
: No matter what values are chosen for c and n_0, we can choose $n \geq n_0$ such that $n \geq \mathit{Fibonacci}(n)$ does not hold. The value of $\mathit{Fibonacci}(n)$ more than doubles every time n is increased by 2 (see Section 7.2.1), but the value of $c(n)$ only increases by $2c$. Hence, if we keep increasing n, eventually $\mathit{Fibonacci}(n+1) > c(n-2)$ for any choice of c.

Exercise 7.4. Repeat Exercise 7.2 using Ω instead of O.

Exercise 7.5. For each part, identify a function g that satisfies the property.

a. g is in $O(n^2)$ but not in $\Omega(n^2)$.

b. g is not in $O(n^2)$ but is in $\Omega(n^2)$.

c. g is in both $O(n^2)$ and $\Omega(n^2)$.

7.2.3 Theta

The function $\Theta(f)$ denotes the set of functions that grow at the same rate as f. It is the intersection of the sets $O(f)$ and $\Omega(f)$. Hence, a function g is in $\Theta(f)$ if and only if g is in $O(f)$ and g is in $\Omega(f)$. In Figure 7.2, $\Theta(f)$ is the ring between the outer and inner circles.

An alternate definition combines the inequalities for O and Ω:

Definition of $\Theta(f)$. The function g is a member of the set $\Theta(f)$ if any only if there exist positive constants c_1, c_2, and n_0 such that, for all values $n \geq n_0$,

$$c_1 f(n) \geq g(n) \geq c_2 f(n).$$

If $g(n)$ is in $\Theta(f(n))$, then the sets $\Theta(f(n))$ and $\Theta(g(n))$ are identical. If $g(n) \in \Theta(f(n))$ then g and f grow at the same rate,

Example 7.3: Θ Examples

Determining membership in $\Theta(f)$ is simple once we know membership in $O(f)$ and $\Omega(f)$.

$n-7$ is in $\Theta(n+12)$
: Since $n-7$ is in $O(n+12)$ and $n-7$ is in $\Omega(n+12)$ we know $n-7$ is in $\Theta(n+12)$. Intuitively, $n-7$ increases at the same rate as $n+12$, since adding one to n adds one to both function outputs. We can also show this using the definition of $\Theta(f)$: choose $c_1 = 1$, $c_2 = \frac{1}{2}$, and $n_0 = 38$.

$2n$ is in $\Theta(3n)$
: $2n$ is in $O(3n)$ and in $\Omega(3n)$. Choose $c_1 = 1$, $c_2 = \frac{1}{3}$, and $n_0 = 1$.

n is **not** in $\Theta(n^2)$
: n is not in $\Omega(n^2)$. Intuitively, n grows slower than n^2 since increasing n by one always increases the value of the first function, n, by one, but increases the value of n^2 by $2n+1$, a value that increases as n increases.

n^2 is **not** in $\Theta(n)$: n^2 is not in $O(n)$.

$n-2$ is **not** in $\Theta(\mathit{Fibonacci}(n+1))$: $n-2$ is not in $\Omega(n)$.

$\mathit{Fibonacci}(n)$ is **not** in $\Theta(n)$: $\mathit{Fibonacci}(n+1)$ is not in $O(n-2)$.

Properties of O, Ω, and Θ. Because O, Ω, and Θ are concerned with the asymptotic properties of functions, that is, how they grow as inputs approach infinity, many functions that are different when the actual output values matter generate identical sets with the O, Ω, and Θ functions. For example, we saw $n-7$ is in $\Theta(n+12)$ and $n+12$ is in $\Theta(n-7)$. In fact, every function that is in $\Theta(n-7)$ is also in $\Theta(n+12)$.

More generally, if we could prove g is in $\Theta(an+k)$ where a is a positive constant and k is any constant, then g is also in $\Theta(n)$. Thus, the set $\Theta(an+k)$ is equivalent to the set $\Theta(n)$.

We prove $\Theta(an+k) \equiv \Theta(n)$ using the definition of Θ. To prove the sets are equivalent, we need to show inclusion in both directions.

$\Theta(n) \subseteq \Theta(an+k)$: For any function g, if g is in $\Theta(n)$ then g is in $\Theta(an+k)$. Since g is in $\Theta(n)$ there exist positive constants c_1, c_2, and n_0 such that $c_1 n \geq g(n) \geq c_2 n$. To show g is also in $\Theta(an+k)$ we find d_1, d_2, and m_0 such that $d_1(an+k) \geq g(n) \geq d_2(an+k)$ for all $n \geq m_0$. Simplifying the inequalities, we need $(ad_1)n + kd_1 \geq g(n) \geq (ad_2)n + kd_2$. Ignoring the constants for now, we can pick $d_1 = \frac{c_1}{a}$ and $d_2 = \frac{c_2}{a}$. Since g is in $\Theta(n)$, we know

$$(a\frac{c_1}{a})n \geq g(n) \geq (a\frac{c_2}{a})n$$

is satisfied. As for the constants, as n increases they become insignificant. Adding one to d_1 and d_2 adds an to the first term and k to the second term. Hence, as n grows, an becomes greater than k.

$\Theta(an+k) \subseteq \Theta(k)$: For any function g, if g is in $\Theta(an+k)$ then g is in $\Theta(n)$. Since g is in $\Theta(an+k)$ there exist positive constants c_1, c_2, and n_0 such

that $c_1(an+k) \geq g(n) \geq c_2(an+k)$. Simplifying the inequalities, we have $(ac_1)n + kc_1 \geq g(n) \geq (ac_2)n + kc_2$ or, for some different positive constants $b_1 = ac_1$ and $b_2 = ac_2$ and constants $k_1 = kc_1$ and $k_2 = kc_2$, $b_1 n + k_1 \geq g(n) \geq b_2 n + k_2$. To show g is also in $\Theta(n)$, we find d_1, d_2, and m_0 such that $d_1 n \geq g(n) \geq d_2 n$ for all $n \geq m_0$. If it were not for the constants, we already have this with $d_1 = b_1$ and $d_2 = b_2$. As before, the constants become inconsequential as n increases.

This property also holds for the O and Ω operators since our proof for Θ also proved the property for the O and Ω inequalities.

This result can be generalized to any polynomial. The set $\Theta(a_0 + a_1 n + a_2 n^2 + ... + a_k n^k)$ is equivalent to $\Theta(n^k)$. Because we are concerned with the asymptotic growth, only the highest power term of the polynomial matters once n gets big enough.

Exercise 7.6. Repeat Exercise 7.2 using Θ instead of O.

Exercise 7.7. Show that $\Theta(n^2 - n)$ is equivalent to $\Theta(n^2)$.

Exercise 7.8. [⋆] Is $\Theta(n^2)$ equivalent to $\Theta(n^{2.1})$? Either prove they are identical, or prove they are different.

Exercise 7.9. [⋆] Is $\Theta(2^n)$ equivalent to $\Theta(3^n)$? Either prove they are identical, or prove they are different.

7.3 Analyzing Procedures

By considering the asymptotic growth of functions, rather than their actual outputs, the O, Ω, and Θ operators allow us to hide constants and factors that change depending on the speed of our processor, how data is arranged in memory, and the specifics of how our interpreter is implemented. Instead, we can consider the essential properties of how the running time of the procedures increases with the size of the input.

This section explains how to measure input sizes and running times. 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 7.3.3 we consider *which* input of a given size should be used to reason about the cost of applying a procedure. Section 7.4 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.

7.3.1 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 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. The number of different symbols in the tape

alphabet does not matter for our analysis since we are concerned with orders of growth not absolute values. Within the O, Ω, and Θ operators, a constant factor does not matter (e.g., $\Theta(n) \equiv \Theta(17n + 523)$). This means is doesn't matter whether we use an alphabet with two symbols or an alphabet with 256 symbols. With two symbols the input may be 8 times as long as it is with a 256-symbol alphabet, but the constant factor does not matter inside the asymptotic operator.

Thus, we measure the size of the input as 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, 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 $\log_2 n$ bits. Within the asymptotic operators, the base of the logarithm does not matter (as long as it is a constant) since it changes the result by a constant factor. 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.

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), 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 and the input size is in $\Theta(n^2)$.

7.3.2 Running Time

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 takes evaluate the procedure on a wide range of inputs.

To estimate the running time of an evaluation, we use 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. For any particular processor, both the time it takes to perform a step and the amount of work that can be done in one step varies. 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 and 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 is tedious.

Time makes more converts than reason. Thomas Paine

Instead, we 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 the same answer within the asymptotic operators. One possibility 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.

7.3.3 Worst Case Input

A procedure may have different running times for inputs of the same size.

For example, consider 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).

worst case In our analyses we usually consider the *worst case* input. For a given size, the worst case input is the input 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 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 7.4.2.

7.4 Growth Rates

Since our goal is to understand how the running time of an application of a procedure is related to the size of the input, we want to devise a function that takes as input a number that represents the size of the input and outputs the maximum number of steps required to complete the evaluation on an input of that size. Symbolically, we can think of this function as:

$$\textit{Max-Steps}_{\textit{Proc}}\text{: } \textit{Number} \rightarrow \textit{Number}$$

where *Proc* is the name of the procedure we are analyzing. 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 $\textit{Max-Steps}_{\textit{Proc}}$. Instead, we characterize the running time of a procedure with a set of functions denoted by an asymptotic operator. Inside the O, Ω, and Θ operators, 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. Since $\Theta(f)$ is the intersection of $O(f)$ (no faster than) and $\Omega(f)$ (no slower than), 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.

7.4.1 No Growth: Constant Time

If the running time of a procedure does not increase when the size of the input increases, the procedure must be able to produce its output by looking at only a constant number of symbols in the input. Procedures whose running time does not increase with the size of the input are known as *constant time* procedures. *constant time*
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 not growing at all, $O(1)$ 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. No matter how long the input is, a constant time procedure can look at no more than some fixed number of squares on the tape, so cannot even read the whole input.

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 $O(1)$.[4] 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.

7.4.2 Linear Growth

linearly

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.

An example of a procedure that has linear growth is the elementary school addition algorithm from Section 6.2.3. To add two d-digit numbers, we need to perform a constant amount of work for each digit. The number of steps required grows linearly with the size of the numbers (recall from Section 7.3.1 that the *size* of a number is the number of input symbols needed to represent the number).

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 analyze three list procedures, all of which have running times that scale linearly with the size of their input.

Example 7.4: 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))))
```

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. So, our goal is to define a function $\mathit{Max\text{-}Steps}_{\mathit{list\text{-}append}}(n_p, n_q)$ that captures how the maximum number of steps required to evaluate an application of *list-append* scales with the size of its input.

[4] 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.

To analyze the running time of *list-append*, we examine its body which is an if expression. The predicate expression applies the *null?* procedure with is 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*, all of which is are constant time procedures.

So, we can defined the total running time recursively as:

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

where C is some constant that reflects the time for all the operations besides the recursive call. Note that the value of n_q does not matter, so we simplify this to:

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

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:

$$\textit{Max-Steps}_{\textit{list-append}}(0) = C_0$$

where C_0 is some constant.

To better characterize the running time of *list-append*, we want a closed form solution. For a given input n, $\textit{Max-Steps}(n)$ 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 within the asymptotic notations the constant values do not matter. The important property 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 $\Theta(n_p)$. Equivalently, the running time for the *list-append* procedure scales linearly with the length of the first input list.

Example 7.5: Length

Consider the *list-length* procedure from Example 5.1:

```
(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* involves 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 7.4.1). The consequent expression is the primitive expression, `0`, which can be evaluated in constant time. The alternate expression, (+ 1 (*list-length* (*cdr p*))), includes the recursive call. There are $n+1$ total applications of *list-length* to evaluate, the total running time is $n+1$ times the work required for each application (other than the recursive application itself).

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.

Cost of Addition. 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 a very limited way: 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.

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.

We always need to examine one bit, the least significant bit. Half the time we also need to examine the second least significant bit. Of those times, half the time we need to continue and examine the next least significant bit. This con-

tinues 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.

This result generalizes to addition where one of the inputs is any constant value. 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.

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.

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 the running time for the length procedure is in $\Theta(n)$ where n is the length of the input list.

Example 7.6: 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, 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 depends on the details of how numbers are represented). 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 (Exercise 7.13 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 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 7.10. 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 7.11. 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 7.12. For the decimal six-digit odometer (shown in the picture on page 142), 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.

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 7.13. [⋆] 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 7.14. [⋆] 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 DrRacket $+$ 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.

7.4.3 Quadratic Growth

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.

quadratically

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 7.7: 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 7.4, 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. 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

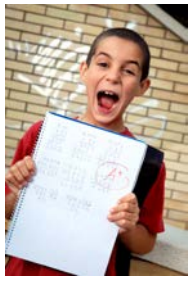
times the average running time of each recursive application:

$$n \cdot \Theta(n) = \Theta(n^2).$$

Thus, the running time is quadratic in the size of the input list.

Example 7.8: 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:



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

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 improvement over the $\Theta(n^2)$ elementary school algorithm. In 2007, Martin Fürer discovered an even faster algorithm for multiplication.[5] It is not yet known if this is the fastest possible multiplication algorithm, or if faster ones exist.

Exercise 7.15. [⋆] Analyze the running time of the elementary school long division algorithm.

Exercise 7.16. [⋆] 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 7.17. [⋆⋆⋆⋆] Devise an asymptotically faster general multiplication algorithm than Fürer's, or prove that no faster algorithm exists.

[5] Martin Fürer, Faster Integer Multiplication, *ACM Symposium on Theory of Computing*, 2007.

7.4.4 Exponential Growth

If the running time of a procedure scales as a 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. Exponential growth is very fast—it is not feasible to evaluate applications of an exponential time 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 at the beginning of Chapter 4), the pegboard puzzle (Exploration 5.2, 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 computer science.

Example 7.9: Factoring

A simple way to find a factor of a given input number is to exhaustively try all possible numbers below the input number to find the first one that divides the number evenly. The *find-factor* procedure takes one number as input and outputs the lowest factor of that number (other than 1):

```
(define (find-factor n)
  (define (find-factor-helper v)
    (if (= (modulo n v) 0) v (find-factor-helper (+ 1 v))))
  (find-factor-helper 2))
```

The *find-factor-helper* procedure takes two inputs, the number to factor and the current guess. Since all numbers are divisible by themselves, the *modulo* test will eventually be *true* for any positive input number, so the maximum number of recursive calls is n, the magnitude of the input to *find-factor*. The magnitude of n is exponential in its size, so the number of recursive calls is in $\Theta(2^b)$ where b is the number of bits in the input. This means even if the amount of work required for each recursive call were constant, the running time of the *find-factor* procedure is still exponential in the size of its input.

The actual work for each recursive call is not constant, though, since it involves an application of *modulo*. The *modulo* built-in procedure takes two inputs and outputs the remainder when the first input is divided by the second input. Hence, it output is 0 if n is divisible by v. Computing a remainder, in the worst case, at least involves examining every bit in the input number, so scales at least linearly in the size of its input[6]. This means the running time of *find-factor* is in $\Omega(2^b)$: it grows at least as fast as 2^b.

There are lots of ways we could produce a faster procedure for finding factors: stopping once the square root of the input number is reached since we know there is no need to check the rest of the numbers, skipping even numbers after 2 since if a number is divisible by any even number it is also divisible by 2, or using advanced sieve methods. This techniques can improve the running time by constant factors, but there is no known factoring algorithm that runs in faster than

[6]In fact, it computing the remainder requires performing division, which is quadratic in the size of the input.

exponential time. The security of the widely used RSA encryption algorithm depends on factoring being hard. If someone finds a fast factoring algorithm it would put the codes used to secure Internet commerce at risk.[7]

Example 7.10: Power Set

power set 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:

```
(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 with 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* since we need to evaluate (*list-powerset* (*cdr s*)) twice. The number of applications of *list-powerset* is 2^n where n is the length of the input list.[8]

The body of *list-powerset* is an if expression. The predicate applies the constant-time procedure, *null?*. The consequent expression, (*list null*) is also constant time. The alternate expression is an application of *list-append*. From Example 7.4, 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,

[7] The movie *Sneakers* is a fictional account of what would happen if someone finds a faster than exponential time factoring algorithm.

[8] Observant readers will note that it is not really necessary to perform this evaluation twice since we could do it once and reuse the result. Even with this change, though, the running time would still be in $\Theta(2^n)$.

the running time for all the *list-map* applications is in $\Theta(2^n)$.

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 $\Theta(2^n)$.

In this case, we know there can be no faster than exponential procedure that solves the same problem, 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, the size of the output provides a lower bound on the running time of the Turing Machine. The size of the powerset is 2^n where n is the size of the input set. Hence, the fastest possible procedure for this problem has at least exponential running time.

7.4.5 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 magnitude of the input parameter. The size of a numeric input is the number of bits needed to express it, so the value n can be as high as $2^b - 1$ where b is the number of bits. Hence, the running time of the *fibo* procedure is in $\Theta(\phi^{2^b})$ where b is the size of the input. This is why we are still waiting for (*fibo* 60) to finish evaluating.

7.4.6 Non-terminating Procedures

All of the procedures so far in the section are algorithms: they may be slow, but they are guaranteed to eventually finish if one can wait long enough. 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.

7.5 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 understand is how the work required scales with the size of the input. The asymptotic operators provide a convenient way of understanding the cost involved in evaluating a procedure applications.

Procedures that can produce an output only touching a fixed amount have constant running times. Procedures whose running times increase by a fixed 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 doubles when the input size increases by one have exponential (in $\Theta(2^n)$) running times. Procedures with exponential running time can only be evaluated for small inputs.

Asymptotic analysis, however, must be interpreted cautiously. 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. Without knowing the constants that are hidden by the asymptotic operators, there is no way to accurately predict the actual running time on a given input.

Exercise 7.18. Analyze the asymptotic running time of the *list-sum* procedure (from Example 5.2):

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

You may assume all of the elements in the list have values below some constant (but explain why this assumption is useful in your analysis).

Exercise 7.19. Analyze the asymptotic running time of the *factorial* procedure (from Example 4.1):

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

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

Exercise 7.20. Consider the *intsto* problem (from Example 5.8).

a. [⋆] Analyze the asymptotic running time of this *intsto* procedure:

```
(define (revintsto n)
  (if (= n 0)
      null
      (cons n (revintsto (- n 1)))))
(define (intsto n) (list-reverse (revintsto n)))
```

b. [⋆] Analyze the asymptotic running time of this *instto* procedure:

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

c. Which version is better?

d. [⋆⋆] Is there an asymptotically faster *intsto* procedure?

Exercise 7.21. Analyze the running time of the *board-replace-peg* procedure (from Exploration 5.2):

```
(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 7.22. Analyze the running time of the *deep-list-flatten* procedure from Section 5.5:

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

Exercise 7.23. [⋆] 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 (July 2011), since the current entry contains many fairly obvious errors. Hopefully it will soon become a [⋆⋆⋆] challenge, and perhaps, eventually will become impossible!