Chapter 6

Cost

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

Alan Perlis

The evaluation rules in Chapter 3 explain how to determine the value of every expression that can be produced from our Scheme grammar subset. In this chapter, we begin our exploration of how to predict the *cost* of evaluating a given expression. That cost is most immediately measured in the amount of time it will take the evaluation to complete. Other measures of cost include the amount of memory the processor will need to use to complete the evaluation and the amount of energy consumed by the processor to complete the evaluation. Indirectly, these costs can often be translated into money: the value of the time for the person waiting for the program to produce an answer, or the price of the computers needed to solve a problem. In this chapter, we introduce tools for understanding the cost of evaluating an expression. The following chapter uses these tools to characterize procedures and make predictions about long evaluations of different applications of those procedures will take.

6.1 Measuring Cost

The most obvious way to measure the cost of evaluating a given expression is to just evaluate it. If we are primarily concerned with time, we could just use a stopwatch to measure the time it takes to complete the evaluation. For more accurate results, we can use the built-in `(time` *`expression`*`)` special form to find the processor time used to evaluate the expression.[1] Evaluating `(time` *`expression`*`)` produces the value of *expression*, but also prints out the time required to evaluate the expression (shown in our examples using *slanted* font).

The output printed by `time` provides three 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 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 reflects just the amount of time the processor was working on evaluating this expression.
- *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. We will explain how garbage collection works and why it is necessary in Chapter **??**.

For example, assuming the definitions from Chapter 5:

```
> (time (car (append (intsto 100) (intsto 1000))))
cpu time: 188 real time: 188 gc time: 0
1
> (time (car (append (intsto 1000) (intsto 100))))
cpu time: 406 real time: 406 gc time: 219
1
> (time (car (append (intsto 1000) (intsto 100))))
cpu time: 250 real time: 250 gc time: 0
1
```

The last two expressions are identical, but the time taken is quite different (mainly because garbage collection happened to be needed for the first evaluation, but

[1]The `time` construct is not part of the standard Scheme language, but is an extension provided by the MzScheme language implemented by DrScheme.

not for the second). Timings are inexact and many properties unrelated to our expression (such as what else is running on the computer and where things happen to be stored in memory) affect the actual time needed for any particular evaluation. Hence, it is dangerous to draw conclusions based on a few timings.

Exercise 6.1. Explain why `time` must be a special form, and cannot be implemented as a normal procedure. ♢

Exercise 6.2. 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 `(append short long)` or `(append long short)`? (Hint: the examples above provide some data, but definitely do not provide enough information to answer this question well.) ♢

Example 6.1: 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 long arithmetic computations we learn in grade school. It also included the problem for which *Fibonacci* numbers are named[2]:

> 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 *Fibonacci(n)* that gives the number of pairs of rabbits at the beginning of the n^{th} month as:

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

[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.

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

For example,

$$\begin{array}{l}
Fibonacci(1) = 1 \\
Fibonacci(2) = 1 \\
Fibonacci(3) = Fibonacci(2) + Fibonacci(1) = 2 \\
Fibonacci(4) = Fibonacci(3) + Fibonacci(2) = 3 \\
Fibonacci(5) = Fibonacci(4) + Fibonacci(3) = 5 \\
\ldots
\end{array}$$

The sequence produced is known as the Fibonacci sequence:

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

Translating the definition of *Fibonacci* above into a Scheme procedure is straightforward:

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

Unlike other recursive definitions we have seen, this one has two base cases, for inputs `1` and `2`.[3] We need both base cases, since the recursive case involves applications of `fibo` to both `(- n 1)` and `(- n 2)`.

Our definition of `fibo` appears to be correct, but when we use it to try and determine the number of rabbits in five years by computing `(fibo 60)`, our interpreter just hangs without producing a value.[4]

[3]This is the way Bonacci defined the sequence, and the most standard mathematical definition of it. It is sometimes defined with the base case inputs being `0` and `1`, as is the case in Abelson & Sussman, *Structure and Interpretation of Computer Programs*.

[4]Try evaluating this yourself to see what happens. If you get bored waiting for a result, you can use the **Stop** button in the upper right hand corner to terminate the evaluation.

The `fibo` procedure is definied in a way that guarantees it will complete when applied to a non-negative whole number: each recursive call reduces the input by one or two, so both inputs get closer to the base cases than the original input. Hence, we always make progress and must eventually reach the base case, unwind the recursive applications, and produce a value. So, we know it always eventually finishes. To understand why the evaluation of `(fibo 60)` did not finish in our interpreter, we need to consider how much work is involved in evaluating 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 `(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 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 6.1. 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. 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 trillion applications of `fibo`!

Although our recursive definition is *correct*, it is ridiculously inefficient. It involves a tremendous amount of duplicated work: for the `(fibo 60)` example, over a trillion evaluations each of `(fibo 1)` and `(fibo 2)`.

A more efficient definition would avoid this duplicated effort. We can do this 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 below 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. This is a form of what is known as *dynamic programming*. The definition is still recursive, but unlike the original

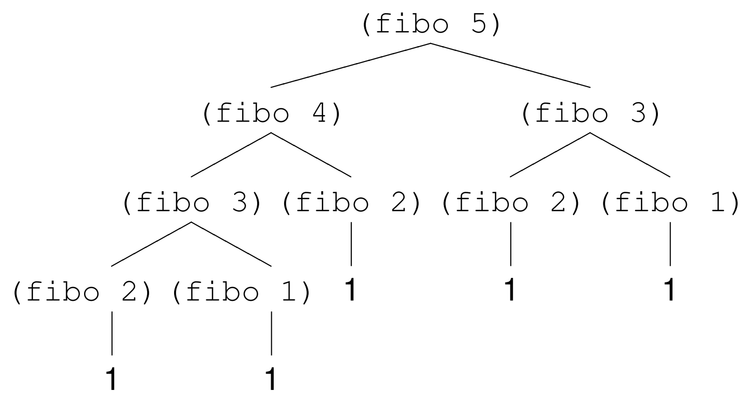

Figure 6.1: Evaluation of `(fibo 5)`.

definition the problem is broken down differently. Instead of breaking the problem down into a slightly smaller instance of the original problem (or in the case of Fibonacci, two slightly smaller instances of the original problem), the `fast-fibo` procedure builds up from the base cases until reaching the desired answer:

```
(define (fast-fibo n)
  (define (fib-helper a b left)
    (if (zero? left)
        b
        (fib-helper b (+ a b) (- left 1))))
  (fib-helper 1 1 (- n 2)))
```

The helper procedure, `fib-helper`, 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 numbers needed before reaching the target. The initial call to `fib-helper` passed in `1` as `a` (the value of $\mathit{Fibonacci}(1)$), and `1` as `b` (the value of $\mathit{Fibonacci}(2)$), and `(- n 2)` as `left` (we have $n - 2$ more numbers to go to reach the target, since the first two Fibonacci numbers were passed in as `a` and `b` we are now working on $\mathit{Fibonacci}(2)$).

The body of `fib-helper` first checks if we have reached the target number. This happens when `left` is `0`, and the value is the previous Fibonacci number (which was passed in as the value of the `b` parameter). If we have not reached the target number, we make progress by recursively calling `fib-helper`, but advancing the numbers: the value that was previously `b` (the previous Fibonacci number) will now be the first parameter (the previous-previous Fibonacci number), the value of the previous Fibonacci number is the sum of the previous two, `(+ a b)`, and since we have advanced one number the value passed in as `left` is decremented by `1`.

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 `fast-fibo` needed to evaluate `(fast-fibo 60)` is now only 59. The value passed in as `left` for the first application of `fib-helper` 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 as 1548008755920 (over 1.5 Trillion[5]).

6.2 Orders of Growth

From the Fibonacci example, we see that 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, even on the world's most powerful supercomputer, but the fast Fibonacci procedure finishes (apparently instantly) on a typical laptop.

The important difference is the number of recursive applications: for the original procedure, we need $Fibonacci(n+1)$ applications to compute `(fibo` n`)`; for the fast procedure, we need $n-2$ applications to compute `(fast-fibo` n`)`. Although the amount of time each application takes is different for the two procedures, the actual time needed does not matter too much for understanding the resources required to evaluate the procedure applications. The actual time will depend on the computer we have, as well as on other factors like what other programs are running on the computer at the same time, and how things happen to be

[5]Perhaps Bonacci's assumptions are not a good model for actual rabbit procreation. They suggest 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!

arranged in memory.

In this section, we introduce three notations computer scientists use to capture the important properties of how resources required grow with input size:

- $O(f)$ is the set of functions that grow *no faster* than f grows.
- $\Theta(f)$ is the set of functions that grow *as fast* as f grows.
- $\Omega(f)$ is the set of functions that grow *no slower* than f grows.

Figure 6.2 depicts the sets O, Θ, Ω for some function. The next three subsections define these sets and provide some examples. Chapter 7 illustrates how to analyze the time required to evaluate applications of procedures using these notations.

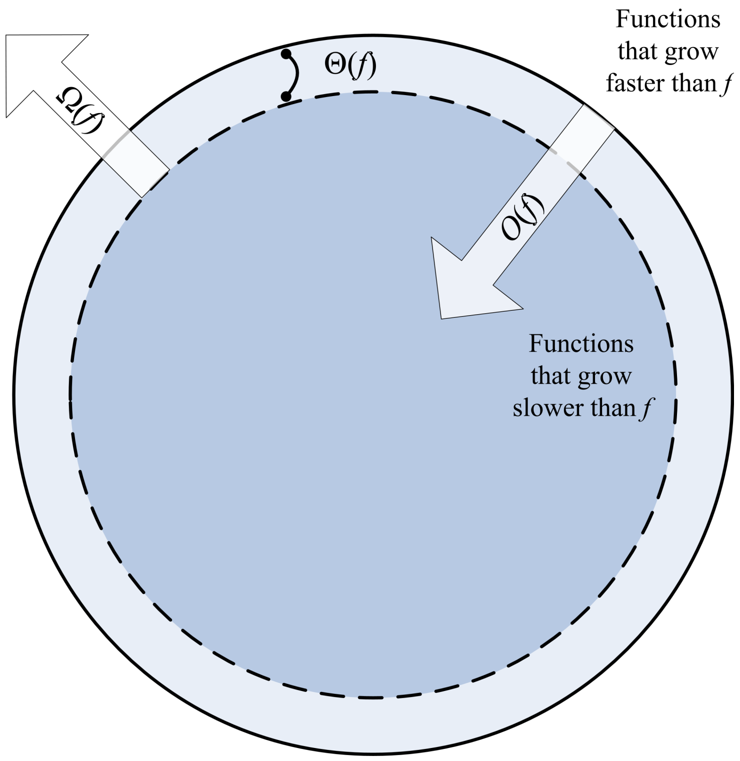
Functions that grow faster than f
$\Theta(f)$
$\Omega(f)$
$O(f)$
Functions that grow slower than f

Figure 6.2: Visualization of the sets $O(f)$, $\Omega(f)$, and $\Theta(f)$.

6.2.1 Big *O*

The first notation we introduce is O, pronounced "big oh". O is a mathematical function that 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 6.2, 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*. What we want to capture is 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.

Consider two functions, $f(n) = n + 12$ and $g(n) = n - 7$. No matter what input value we try for n, the value of $f(n)$ is greater than the value of $g(n)$, but this doesn't matter for the growth rates. What matters is the difference between $g(n)$ and $f(n)$ as the input values change. No matter what values we choose for n_1 and n_2, we know $g(n_1) - f(n_1) = g(n_2) - f(n_2) = -19$. So, the growth rates of f and g are identical. Hence, $n - 7$ is in the set $O(n + 12)$, and $n + 12$ is in the set $O(n - 7)$.

Suppose the functions are $f(n) = 2n$ 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 arount as n increases. So, the growth rate as n increases is $n/n = 1$. Hence, $2n$ is in the set $O(3n)$ and $3n$ is in the set $O(2n)$.

Now, consider $f(n) = n$ 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 $n(n - 1)/n = n - 1$. The value of $n - 1$ increases as n increases, so g grows faster than f. This means n^2 is not in $O(n)$, but n is in $O(n^2)$ since n grows slower than n^2 grows.

For our final example, consider the number of applications of our Fibonacci procedures. For the first procedure, the number of applications is $Fibonacci(n + 1)$; for the second procedure, the number of applications is $n - 2$. The $Fibonacci$ function grows very rapidly. The value of $Fibonacci(n + 2)$ is more than *double* the value of $Fibonacci(n)$ since

$$Fibonacci(n + 2) = Fibonacci(n + 1) + Fibonacci(n)$$

and $Fibonacci(n + 1) > Fibonacci(n)$. 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).[6] This is much faster than the growth rate of $n - 2$, which increases by one when we increase n by one. So, $n - 2$ is in the set $O(Fibonacci(n + 1))$, but $Fibonacci(n + 1)$ is not in the set $O(n - 2)$.

Some of the example functions are plotted in Figure 6.2.1. Recall that we are concerned with the running time of programs as input sizes increase. The O notation reveals the asymptotic behavior of functions. Note in the first graph, the rightmost value of n^2 is greatest, followed by $3n$, $n+12$ and $Fibonacci(n)$. For higher input values, however, eventually the value of $Fibonacci(n)$ will be greatest. For the third graph, the values of $Fibonacci(n)$ for input values up to 20 are so high, 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

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

for all values $n \geq n_0$.

Example 6.2: O Examples. We now show the properties claimed earlier are true using the formal definition. 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.

a. $n - 7$ 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.

b. $n + 12$ 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 $n \geq 26$.

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

[6] 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.

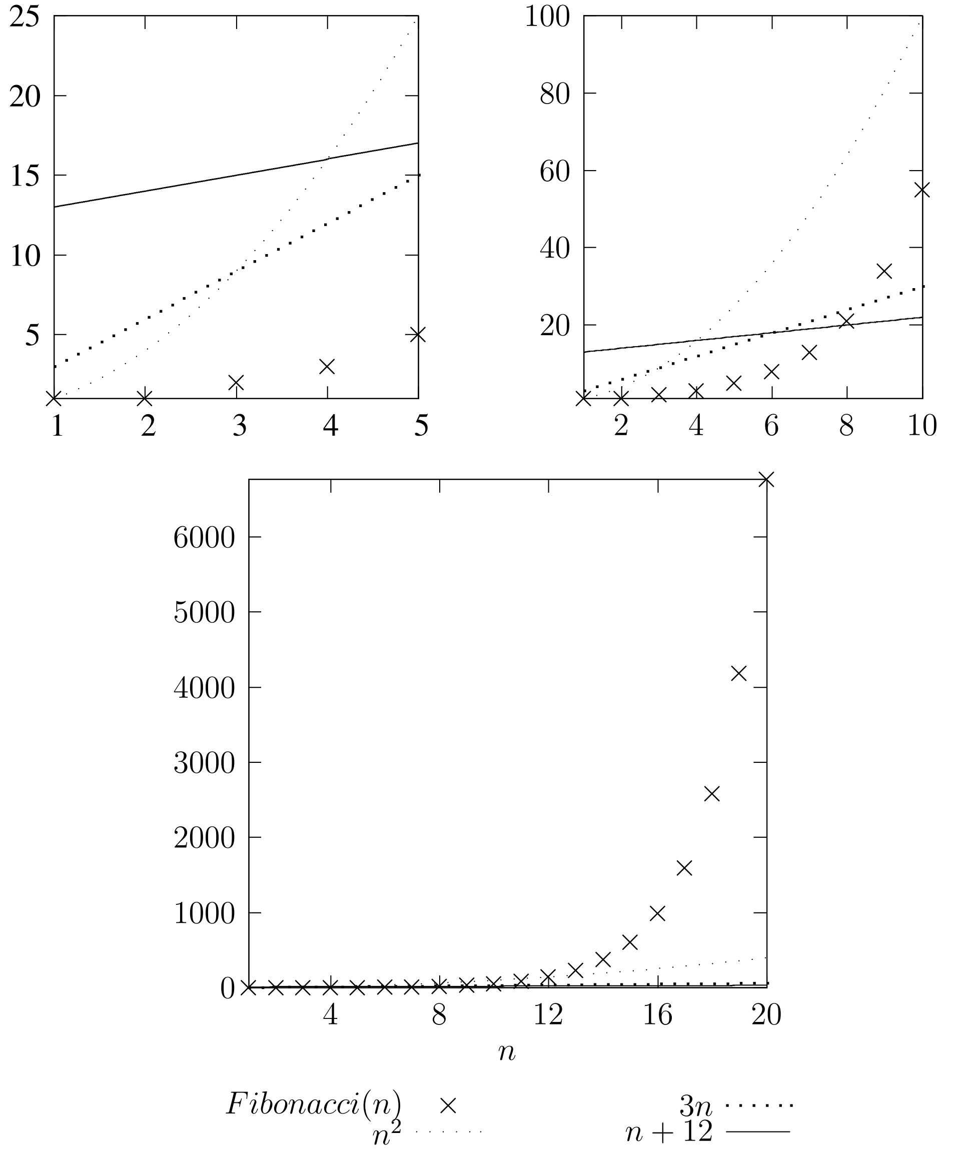

Figure 6.3: Orders of Growth. Each figure shows the same four functions, but for different ranges of input values.

d. $3n$ 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 $n \geq 1$.

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

f. 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 $n > c$ thus invalidating the $n^2 \leq cn$ inequality.

g. $n - 2$ is in $O(Fibonacci(n+1))$ — Choose $c = 1$ and $n_0 = 1$. Then $n - 2 \leq Fibonacci(n+1)$ for all values $n \geq n_0$.

h. $Fibonacci(n+1)$ 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 $Fibonacci(n+1) > c(n-2)$. 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

$$Fibonacci(n+1) > n(n-2)$$

The right side simplifies to $n^2 - 2n$. For $n > 12$, we know $Fibonacci(n) > n^2$, so we also know $Fibonacci(n+1) > n^2 - 2n$. Hence, we can always choose an n that negates the $Fibonacci(n+1) \leq n - 2$ 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 possible choices for c and n_0 that would work. For the given c values, we can always use a higher n_0 value than we choose. 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 would work equally well if we selected higher values for n_0 than we did. 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 $n \geq n_0$.

The key to the proofs showing that g is not in $O(f)$ is that the value of n that invalidates the inequality can be selected *after* the values of c and n_0 are chosen.

One way to think of these 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 6.3. 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. $g(n) = factorial(n)$

$\diamondsuit$

Exercise 6.4.($\star\star$) Given f is some function in $O(h)$, and g is some function not in $O(h)$, which of the following are true (for any choice of h):

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 $m > m_0$, $f(m) < g(m)$.

$\diamondsuit$

6.2.2 Omega (Ω)

The set $\Omega(f)$ is the set of functions that grow no *slower* than f grows. So, a function g is in $\Omega(f)$ if it grows as fast as f or faster. This is different from $O(f)$, which is the set of all functions that grow no *faster* than f grows. In Figure 6.2, $\Omega(f)$ 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 $\leq$ operator 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

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

for all values $n \geq n_0$.

Example 6.3: Ω Examples. We repeat the 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 $g(n) < cf(n)$.

a. $n - 7$ is in $\Omega(n + 12)$ — Choose $c = \frac{1}{2}$ and $n_0 = 38$. Then, we need to show $n - 7 \geq \frac{1}{2}(n + 12)$ for all values $n \geq 38$. This is true, since the inequality simplifies $\frac{n}{2} \geq 19$ which holds for all values $n \geq 38$.

b. $n + 12$ is in $\Omega(n - 7)$ — Choose $c = 1$ and $n_0 = 1$.

c. $2n$ 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 $n \geq 1$.

d. $3n$ is in $\Omega(2n)$ — Choose $c = 1$ and $n_0 = 1$. Then, $3n \geq 2n$ simplifies to $n \geq 0$ which is true for all values $n \geq 1$.

e. n is not in $\Omega(n^2)$ — Whatever values are choosen for c and n_0, we can choose $n \geq n_0$ such that $n \geq cn^2$ does not hold. We can 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.

f. n^2 is in $\Omega(n)$ — Choose $c = 1$ and $n_0 = 0$: $n^2 \geq n$ for all $n \geq 0$.

g. $n - 2$ is not in $\Omega(\mathit{Fibonacci}(n+1))$ — No matter what values are choosen for c and n_0, we can choose $n \geq n_0$ such that $n - 2 \geq \mathit{Fibonacci}(n+1)$ does not hold. The value of $\mathit{Fibonacci}(n+1)$ more than doubles every time n is increased by 2 (see Section 6.2.1), but the value of $c(n-2)$ only increases by $2c$. Hence, if we keep increasing n, eventually $\mathit{Fibonacci}(n+1) > c(n-2)$ for any choice of c.

h. $\mathit{Fibonacci}(n+1)$ is in $\Omega(n-2)$ — choose $c = 1$ and $n_0 = 0$: $\mathit{Fibonacci}(n+1) \geq n - 2$ for all $n \geq 0$.

Exercise 6.5. Repeat Exercise 6.2.1, but using Ω instead of O. $\diamondsuit$

6.2.3 Theta (Θ)

The notation $\Theta(f)$ is 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 6.2, $\Theta(f)$ is the light grey ring.

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

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

is true for all values $n \geq n_0$.

Example 6.4: Θ Examples. We repeat the previous examples using Θ. Determining membership in $\Theta(f)$ is simple once we know membership in $O(f)$ and $\Omega(f)$.

a. $n - 7$ is in $\Theta(n + 12)$ — It is in $O(n + 12)$ and in $\Omega(n + 12)$. Intuitively, $n - 7$ increases at the same rate as $n + 12$, since adding one to n adds one to both function outputs. Choose $c_1 = 1$, $c_2 = \frac{1}{2}$, and $n_0 = 38$. We can choose our value of c_1 as the value of c in the $O(f)$ proof, c_2 as the value of c in the $\Omega(f)$ proof, and n_0 as the maximum value of the n_0 values from the $O(f)$ and $\Omega(f)$ proofs.

b. $n + 12$ is in $\Theta(n - 7)$ — It is in $O(n - 7)$ and in $\Omega(n - 7)$. Choose $c_1 = 2$, $c_2 = 1$, and $n_0 = 1$.

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

d. $3n$ is in $\Theta(2n)$ — It is in $O(2n)$ and in $\Omega(2n)$. Choose $c_1 = 2$, $c_2 = 1$, and $n_0 = 1$.

e. n is **not** in $\Theta(n^2)$ — It 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.

f. n^2 is **not** in $\Theta(n)$ — It is not in $O(n)$.

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

h. $\mathit{Fibonacci}(n + 1)$ is **not** in $\Omega(n - 2)$ — It is not in $O(n - 2)$.

If $g(n)$ is in $\Theta(f(n))$, then the sets $\Theta(f(n))$ and $\Theta(g(n))$ are identical. We also know $O(f(n)) = O(g(n))$ and $\Omega(f(n)) = \Omega(g(n))$. Intuitively, since $g(n) \in \Theta(f(n))$ means g and f grow at the same rate,

Exercise 6.6. Repeat Exercise 6.2.1, but using Θ instead of O. $\diamondsuit$

6.3 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 then O, Ω, and Θ operators. 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 can prove $\Theta(an + k) \equiv \Theta(n)$ from the definition of Θ. To prove the sets are equivelent, we need to show that (1) any function g which is in $\Theta(n)$ is also in $\Theta(an + k)$; and (2) any function g which is in $\Theta(an + k)$ is also in $\Theta(n)$:

1. Suppose g is in $\Theta(n)$. This means we can find positive constants c_1, c_2, and n_0 such that $c_1 n \geq g(n) \geq c_2 n$. In order to show g is also in $\Theta(an + k)$, we need to show that we can 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 must greater than k.

2. Suppose g is in $\Theta(an + k)$. This means we can find 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 $k_2 = kc_2$, $b_1 n + k_1 \geq g(n) \geq b_2 n + k_2$. In order to show g is also in $\Theta(n)$, we need to show that we can 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 — note that our proof above for Θ also proved the property for the O and Ω inequalities.

The 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 6.7. Show that $\Theta(n^2 - n)$ is equivalent to $\Theta(n^2)$. ♢

Exercise 6.8. ($\star\star$) Is $\Theta(n^2)$ equivalent to $\Theta(n^{2.1})$? Either prove they are identical, or prove they are different. ♢

Exercise 6.9. ($\star\star$) Is $\Theta(2^n)$ equivalent to $\Theta(3^n)$? Either prove they are identical, or prove they are different. ♢

6.4 Summary

By considering the asymptotic growth of functions, rather than their actual outputs, we can better capture the important properties of how the cost of evaluating a procedure application grows with the size of the input. 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 the procedure. In the next chapter, we explore how to use these operators to analyze the costs of evaluating different applications of procedures.