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

In this chapter, we begin our exploration of how to predict the *cost* of evaluating a given expression. Predicting the cost of executing a procedure has practical value (for example, we can determine whether it is worth waiting for a computer to produce a response, or estimate how large a computer we need to solve a given problem instance), but also provides deep insights into the nature of procedures and problems.

The most commonly use cost metric is the amount of time it will take for an execution to complete. Other measures of cost include the amount of memory the processor needs and the amount of energy consumed. 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, the number of transactions per second a service can support, or the price of the computer needed to solve a problem.

In this chapter, we introduce tools for understanding the cost of evaluating a procedure application. Our goal is to reason about the cost of a procedure in a way that does not depend on ephemeral details of a particular computer. From the Turing Machine model in the previous chapter, we know that a simple model computer is enough to execute any algorithm, and that our simple model can simulate any reasonable computer using a constant multiple of the number of steps. Hence, the tools we use to measure cost are robust to details that would be altered by using a different computer. The following chapter uses these tools to characterize procedures and make predictions about how long evaluations of different applications of those procedures will take.

7.1 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 the input expression, but also prints out the time required to evaluate the expression (shown in our examples using *slanted* font).

The output printed by *time* contains 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 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 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.

For example, using the definitions from Chapter 5,

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

produces:

```
cpu time: 141797 real time: 152063 gc time: 765
```

followed by the output:

```
((Move (Position 3 1) (Direction −1 0)) . . . )
```

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

From the printed output generated by *time*, we see the real time is 152 seconds, so 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 are identical, but the time taken is different. 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 doesn't 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 depends on the procedure 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.

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.)

Example 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:

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

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

For example,

$$\begin{aligned}
&Fibonacci(1) = 1\\
&Fibonacci(2) = 1\\
&Fibonacci(3) = Fibonacci(2) + Fibonacci(1) = 2\\
&Fibonacci(4) = Fibonacci(3) + Fibonacci(2) = 3\\
&Fibonacci(5) = Fibonacci(4) + Fibonacci(3) = 5\\
&Fibonacci(6) = Fibonacci(5) + Fibonacci(4) = 8\\
&\ldots
\end{aligned}$$

Sunflower

The sequence produced is known as the Fibonacci sequence:

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

These numbers occur frequently in nature, such as the arrangement of florets in a 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.

[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.

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 20))
cpu time: 16 real time: 16 gc time: 0
6765
> (time (fibo 30))
cpu time: 2156 real time: 2187 gc time: 0
832040
```

Our definition of *fibo* appears to be correct, but when we use it to try to determine the number of rabbits in five years by computing (*fibo* 60), our interpreter just hangs without producing a value.[3]

The *fibo* procedure is defined 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 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 number of evaluations of the *fibo* procedure for each input is itself the Fibonacci sequence!

[3]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.

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*!

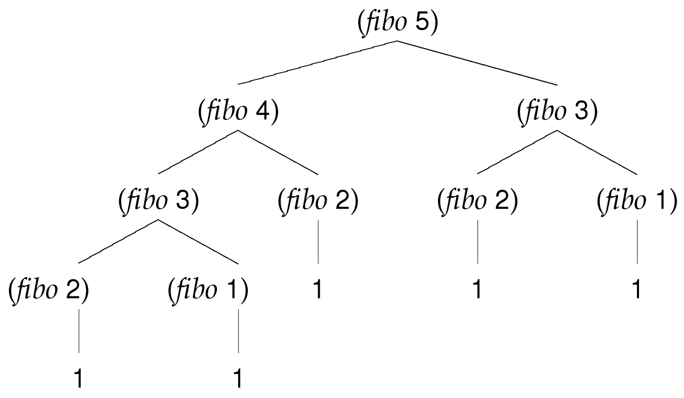

Figure 7.1. Evaluation of *fibo* procedure.

Although our *fibo* definition is *correct*, it is ridiculously inefficient. In practice, it only works 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).

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 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.

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)).

The body of *fibo-iter* 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 *fibo-iter*, but advancing the inputs: the value that was previously *b* (the previous Fibonacci number) will now be the first input (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 completed one iteration 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 *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)[4].

7.2 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 ap-

[4]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!

plication 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 (nearly instantly) on a typical laptop (the time reported by (*time* (*fast-fibo* 60)) is 0 milliseconds).

The important difference is the number of recursive applications: for the original *fibo* procedure, we need *Fibonacci*$(n+1)$ recursive applications to compute (*fibo* n); for the *fast-fibo* procedure, we only need $n-2$ applications of *fibo-iter* to evaluate (*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 depends 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 arranged in memory.

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

In this section, we introduce three functions computer scientists use to capture 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") is the set of functions that grow *no faster* than f grows.
- $\Theta(f)$ (theta) is the set of functions that grow *as fast* as f grows.
- $\Omega(f)$ (omega) is the set of functions that grow *no slower* than f grows.

The point of these functions is to 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 want to understand 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 required to evaluate the target procedure on that input.

Figure 7.2 depicts the sets O, Θ, Ω for some function f. Next, we define each function and provide some examples. Chapter 8 illustrates how to analyze the time required to evaluate applications of procedures using these notations.

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

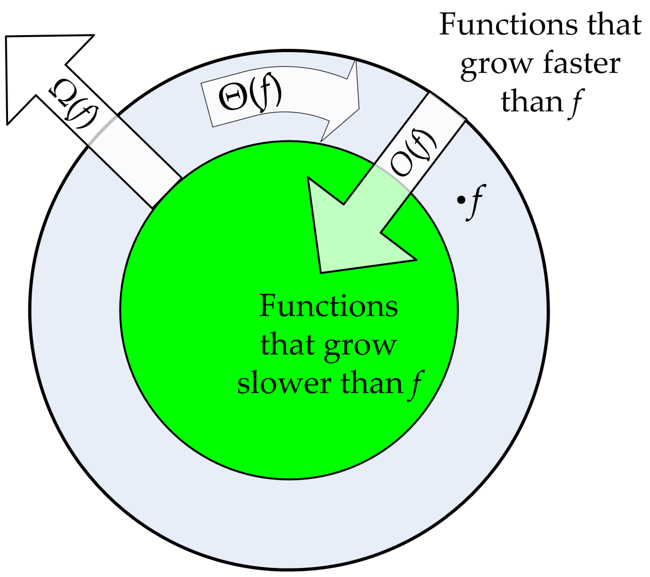
Functions that grow faster than f
Ω(f)
Θ(f)
O(f)
•f
Functions that grow slower than f

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

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*. 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 $g(n)$. 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 $g(n_1) - f(n_1) = g(n_2) - f(n_2) = -19$. 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 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 $O(2n)$.

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 $\frac{n(n-1)}{n} = n - 1$. 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.

For our final example, consider the number of applications of our Fibonacci procedures. For the *fibo* procedure, evaluating (*fibo* n) requires $Fibonacci(n+1)$ recursive applications; for the *fast-fibo* procedure, there are $n-2$ applications.

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).[5] 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 7.2.1. 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 $n \geq n_0$.

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

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.

[5]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.

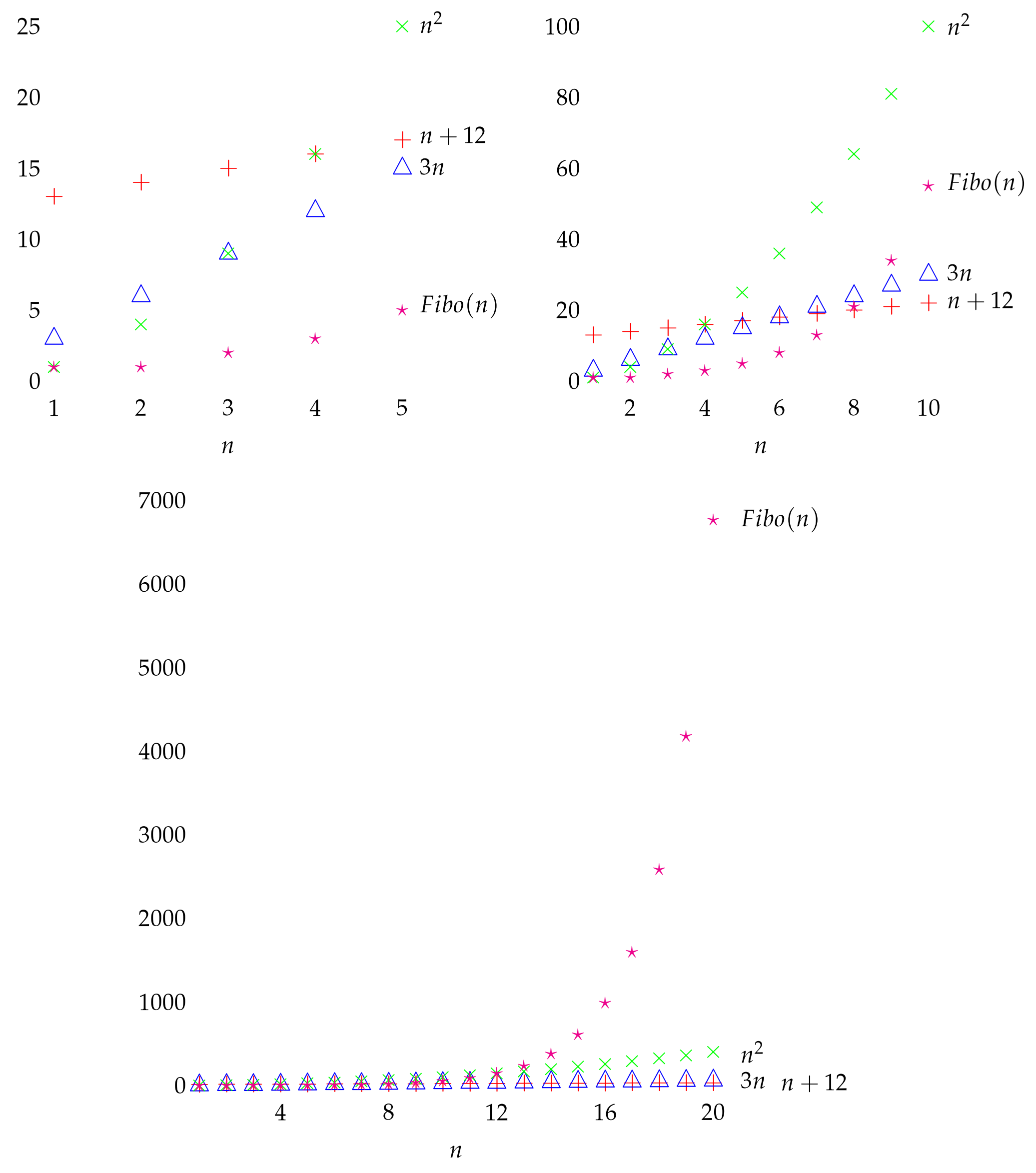

Figure 7.3. Orders of Growth.

Each graph shows the same four functions, but for different input ranges.

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 $n \geq 1$.

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 $n > c$.

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

h. $\mathit{Fibonacci}(n + 1)$ 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 $\mathit{Fibonacci}(n + 1) > c(n - 2)$. We know $\mathit{Fibonacci}(12) = 144$, and, from the discussion above, that:

$$\mathit{Fibonacci}(n + 2) > 2 * \mathit{Fibonacci}(n)$$

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

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

The right side simplifies to $n^2 - 2n$. For $n > 12$, we know $\mathit{Fibonacci}(n) > n^2$, so we also know $\mathit{Fibonacci}(n + 1) > n^2 - 2n$. Hence, we can always choose an n that contradicts the $\mathit{Fibonacci}(n + 1) \leq n - 2$ 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 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 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 $n \geq n_0$.

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. [$\star$] 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 $m > m_0$,

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

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 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 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

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

for all values $n \geq n_0$.

Example 7.3: 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 7.2.1), 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 7.4. Repeat Exercise 7.2, but using Ω instead of O.

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

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

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

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,

Example 7.4: 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 $\Theta(n+12)$: $n-7$ 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 $\Theta(n-7)$: $n+12$ 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 $\Theta(3n)$: $2n$ 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 $\Theta(2n)$: $3n$ is in $O(2n)$ and in $\Omega(2n)$. Choose $c_1 = 2$, $c_2 = 1$, and $n_0 = 1$.

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

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

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

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

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

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

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

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

7.3 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 executing different procedures.