Chapter 7

Time

Time makes more converts than reason.

Thomas Paine

The previous chapter introduced notations for conveniently describing the growth rates of functions. To use them to understand the time it takes to evaluate a procedure application, we need to devise a function that predicts the running time of the evaluation. That function should take as input a number that measures the size of the output, and produce as output a number that captures the running time. The first section of this chapter explains how to measure input sizes and running times. The next six sections analyze some procedures with different growth rates, from slowest to fastest. 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. The final section presents an extended example.

7.1 Measuring Running Time

To understand the growth rate of a procedure's running time, we need a function that maps the size of the inputs to the procedure to the amount of time it takes to evaluate the application. First we consider how to measure the input size; then, we consider how to measure the running time.

7.1.1 Input Size

The inputs to our procedures may be many different things: numbers, lists of numbers, lists of lists of numbers, procedures, etc. Our goal is to characterize the input size with a single number that doesn't depend on understanding the details of the different input types.

We will use the number of characters it takes to write down the input.[1] The characters can be from any fixed-size alphabet, such as the 10 decimal digits, or the letters of the alphabet. Computer scientists typically think about the binary alphabet where there are only two different characters since at the lower levels in the computer, all data is represented using only two different symbols. The number of different characters in the alphabet does not matter for our analysis, since we will be concerned with orders of growth, not with absolute values. Because we will use the O, Ω, and Θ operators from the previous chapter to describe our cost functions, we do not need the output to be an absolute measure of the running time, but instead we need a function that grows at the same rate as the running time grows when the input size increases.

7.1.2 Worst Case Input

A procedure may have different running times for inputs of the same size. An extreme example would be a procedure that takes a list as input and outputs the first positive number in the list:

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

If the first element in the input list is positive, evaluating `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,

[1]In Chapter **??**, we will see more reasons why this is a good choice.

the procedure needs to test each element in the list until it reaches the end of the list (and reports an error).

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

In later chapters, we will sometimes also consider the *average case* input. This requires understanding the distribution of possible inputs, to consider an "average" input.

7.1.3 Running Time

To estimate the running time of an evaluation, we need to consider the number of steps required to perform the evaluation. The actual number of steps depends on the details of how the interpreter is implemented, and what instructions the processor provides. Further, not all steps take the same amount of time. Even the same low-level instruction, such as reading the value in some memory location, may take different amounts of time depending on where the location is and the state of the machine. When we analyze procedures, however, we usually don't want to deal with these details. Instead, what we care about is how the asymptotic running time scales with the input size. This means we can count anything we want as a "step", as long as each step is the approximately same size (that is, the time a step requires does not depend on the size of the input).

One possibility is to count the number of times an evaluation rule is used.[2] The amount of work in each evaluation rule may vary slightly (for example, is it more work to apply the primitive rule or the if-expression rule?) but does not typically depend on the input size. For this chapter, we will consider 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, plus all the work required to

[2] In Chapter **??**, we will define a "step" more universally

evaluate the subexpressions. Because we use the O, Ω, and Θ operators, the actual time needed for each step does not matter — it is hidden in the factor which is hidden by the operator. What matters is how the number of steps required grows as the size of the input grows.

7.2 No Growth

If the running time of a procedure does not increase when the size of the input increases, it means the procedure must be able to produce its output without even looking at the entire input. Procedures whose running time does not increase with the size of the input are known as *constant time* procedures. Their running time is in $O(1)$ — it does not grow at all.

We cannot do much in constant time, since we cannot 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. For example, consider the built-in procedure `car`. It takes a pair as input and evaluates to the first component of that pair. When it 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` should be[3] in $O(1)$.

Other built-in procedures that involve lists and pairs that should have running times in $O(1)$ include `cons`, `cdr`, `null?`, and `pair?`. None of these procedures should depend on examining more than the first pair of the list.

7.3 Linear Growth

If 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 procedures grows *linearly* with

[3]Since we are speculating based on what `car` does, not examining how `car` is actually implemented, we cannot say definitively that its running time is in $O(1)$. This would depend on how the Scheme implementation in question implements `car`. But, it would be rather shocking for an implementation to have a running time that is not in $O(1)$. The implementation of `car` in Section 5.1.1 is constant time. Evaluating an application of it involves evaluating a single application expression, and then evaluating an if-expression.

the input. 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.

Many procedures that take a list as input have linear time growth. A procedure that takes a list as input, and does something that takes constant time with every element in the list, has running time that grows linearly with the size of the input list. Consider the `length` procedure from Chapter 5:

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

The procedure makes one recursive application of `length` for each element in the input `p`. If the input has n elements, there will be $n + 1$ total applications of `length` to evaluate (one for each element, and one for the `null`). To determine the running time, we need to determine how much work is involved in each application. Evaluating an application of `length` consists of evaluating its body, which is an if-expression. To evaluate the if-expression first the predicate expression, `(null? p)`, must be evaluated. This requires constant time, since it is an application of the built-in procedure `null?` which should be implemented with running time in $O(1)$ — the time it takes to determine is `p` is null does not depend on the length of the list `p`.

If the predicate expression evaluates to true, the consequent expression must be evaluated. It is the primitive expression, `0`, which can be evaluated in constant time. Otherwise, the alternate expression, `(+ 1 (length (cdr p)))` is evaluated. The time required to evaluated the `length` application is not included now, since we know there are $n + 1$ total applications of `length` to evaluate. It is usually easiest to account for the work involved in all of the recursive applications required, so we will consider the work for all the recursive applications of `length` later.

The other work needed here is evaluating `(cdr p)` and evaluating the + application. Both of these are constant time operations. So, the total time needed to evaluate one application of `length`, excluding the recursive application, is constant. It does not depend on the length of the input list.

There are $n+1$ total applications of `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 constant time needed to evaluate the if-expression in `length`). The set $\Theta(c(n+1))$ is identical to the set $\Theta(n)$, so we can say that the running time for the length procedure is in $\Theta(n)$ where n is the length of the input list. Equivalently, the running time for the length procedure scales linearly with the length of the input list. Less formally, sometimes it is phrased as "`length` is order n".

Exercise 7.1. Explain why the `sum-list`, `product-list`, `map`, and `filter` procedures from Section 5.3.1 all have running times that are linear in the size of their list inputs. (For `map` and `filter`, you should assume the procedure input has constant running time.) $\diamondsuit$

Exercise 7.2. What is the running time of the `append` procedure (from Exercise 5.3.3):

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

Your answer should be in terms of n_p, the number of inputs in the `p` input, and n_q, then number of inputs in the `q` input. $\diamondsuit$

7.4 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.

A procedure that takes a list as input will have quadratic time growth if its evaluation involves going 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 with $n(n-1)$ comparisons (n elements, each is compared with every element besides itself). This simplifies to $n^2 - n$, but $\Theta(n^2 - n)$ is equivalent to $\Theta(n^2)$ (see Exercise 6.3).

The `find-matching-pair` procedure, defined below, takes as inputs a test procedure and a list. The test procedure is a procedure that takes two inputs and produces a Boolean value. If there are any pairs of elements in the list for which the test procedure evaluates to true, the output of `find-matching-pair` is a pair of two elements in the list for which the test procedure evaluates to true.

Here are some example evaluations:

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

To define `find-matching-pair`, we first define a procedure that takes two inputs, a list and a number `n`, and produces as output a list with the same elements as the input list except with the $\mathtt{n}^{th}$ element removed (where we start counting the elements in the list from 0):

```
(define (remove-nth-element lst n)
  (if (null? lst)
      (error "No element n")
      (if (zero? n)
          (cdr lst)
          (cons
           (car lst)
           (remove-nth-element (cdr lst)
                               (- n 1))))))
```

If the input list is null, there are no elements to remove, so `remove-nth-element` evaluates to the error produced by `(error "No element")`. If the value of `n` is 0, we are removing the first element in the list, so the result is the rest of the list. Otherwise, we should keep the first element of the list, and remove the $n-1^{th}$ element from the rest of the list.

The `remove-nth-element` procedure has running time in $\Theta(m)$ where m is the number of elements in the input list (we use m here to avoid confusing with the parameter `n`). The worst case input is when we are removing the last element in the list, when `n` is `(- (length lst) 1)`. There is one recursive application of `remove-nth-element` for every element in the input list. Otherwise, each application has constant running time since it involves applications of `null?`, `zero?`, `cdr`, `cons`, `cdr`, and `-` (with a constant input), all of which have running times in $O(1)$.

The `find-matching-pair` procedure traverses through the elements of the list in order. For each element, it creates a list of all elements except that element using `remove-nth-element`. Then, it compares the current element to every element in that list. Once it finds a match, that is, a pair of inputs for which the test procedure evaluates to true, it produces a pair of those elements as the output and no further recursive calls are needed.

```
(define (find-matching-pair-helper test orig left others)
  (if (null? left)
      #f ; no matching pair found
      (if (null? others)
          (if (null? (cdr left))
              #f
              (find-matching-pair-helper
               test orig (cdr left)
                (remove-nth-element
                 orig (- (length orig)
                         (length (cdr left))))))
          (if (test (car left) (car others))
              (cons (car left) (car others))
              (find-matching-pair-helper
               test orig left (cdr others))))))

(define (find-matching-pair test lst)
  (find-matching-pair-helper test lst lst (cdr lst)))
```

We consider the worst case input in which no pair of elements satisfies the test procedure. This is the worst case input, since it is necessary to check all pairs of elements to determine that there is no pair that satisfies the test procedure. In the best case, the first two elements satisfy the test procedure, and no more elements need to be tested.

To determine the running time of `find-matching-pair-helper`, we need to determine the number of recursive applications, and the amount of work required for each application. The first two inputs, `test` and `orig`, do not change in the recursive applications. They just keep track of the test procedure and the original input list. The other two parameters are `left`, which is the list of elements remaining to try as the first operand to the test procedure, and `others`, which is a list of elements in the original list that remain to be tested with the current element (the first element of `left`).

There are two different recursive applications in `find-matching-pair-helper`. The first one is done when the `others` list is empty. This happens after it has tested the first element of `left` with every other element in the original list without finding any element that satisfies the test procedure. So, it needs to move on to checking the next element of `left`. If there are no more elements (the consequent expression of the `(null? (cdr left))` predicate), it evaluates to #f. It has tried all possible pairs of elements without finding any that satisfy the test procedure.

Otherwise, the alternate expression applies `find-matching-pair-helper` recursively:

```
(find-matching-pair-helper
 test orig (cdr left)
 (remove-nth-element
  orig (- (length orig)
          (length (cdr left)))))
```

This will happen n times, since each time the new value passed in as the `left` parameter is the value of `(cdr left)`. The value passed in as the `others` parameter is the result of applying `remove-nth-element` to the original list and the number of the current element, which is found by subtracting the length of the remaining elements from the length of the original list.

As analyzed earlier, the running time of `remove-nth-element` is in $\Theta(m)$

where m is the length of the input list. In this case, the input list is the original input to `find-matching-pair`, so the running time for the `remove-nth-element` application is in $\Theta(n)$ where n is the number of elements in the original input list. The running time for everything else in `find-matching-pair` is constant. Other than the recursive applications, it involves applications of `null?`, `cdr`, `cons`, `car`, `-` (with two parameters whose values are always below n), and the test procedure (which we assume has constant running time). So, the total running time resulting from these recursive applications is in $\Theta(n^2)$ — there are n applications, and each one requires time in $\Theta(n)$.

We also need to consider the other recursive application at the bottom of the definition of `find-matching-pair`:

```
(find-matching-pair-helper
 test orig left (cdr others))
```

This occurs up to $n - 1$ times for each value of `left`. Every time we apply the first recursive call, the value of `others` passed in is a list with $n - 1$ elements. As long as there are elements remaining in `others`, and the test procedure does not evaluate to a true value, this recursive application is evaluated. It reduces the length of the `others` list by one, since it passes in `(cdr others)` as the new value of `others`. Hence, there will be $n-1$ recursive applications for each value of `left`. But, the original value of `left` is the original input list. So, there are n different values of `left`, each of which involves $n - 1$ evaluations of the second recursive call, for $n(n - 1) = n^2 - n$ total applications. Each application requires constant time, so the running time of these applications is in $\Theta(n^2 - n)$ which is equivalent to $\Theta(n^2)$.

The total running time for `find-matching-pair` is the sum of the running times resulting from the first recursive application, and from the second recursive application: $\Theta(n^2) + \Theta(n^2)$. But, $\Theta(2n^2)$ is equivalent to $\Theta(n^2)$, so the running time of `find-matching-pair` is in $\Theta(n^2)$.

Exercise 7.3. Analyze the running time of the `reverse` procedure from Section 5.3.2:

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

$\diamondsuit$

Exercise 7.4.($\star\star$) Is it possible to define a `reverse` procedure that has running time in $\Theta(n)$ where n is the number of elements in the input list? Either define such a procedure, or explain why it cannot be done. $\diamondsuit$

7.5 Polynomial Growth

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

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

is a polynomial where the variable is n and the coefficients are $a_0, a_1, ..., a_k$. Within our O, Ω and Θ operators, only the term with the largest exponent matters. For high enough n, all the other terms become insignificant. So, the polynomial above is in $\Theta(n^k)$ which is equivalent to $\Theta(a_k n^k + ... + a_3 n^3 + a_2 n^2 + a_1 n + a_0)$ for all positive constants $a_0, \ldots, a_k$.

All of the slower growth rates in this section are polynomials: constant growth is a polynomial where the highest exponent is 0, linear growth is a polynomial where the highest exponent is 1, and quadratic grow is a polynomial where the highest exponent is 2. Of course, we can have procedures where the running time grows as a higher polynomial.

Next, we show an alternate definition of the `find-matching-pair` procedure that has running time in $\Theta(n^3)$. It uses the `get-nth` procedure that takes as inputs a list and an index `n`, and evaluates to the `n`th element of the input list (where the first element is element 0):

```
(define (get-nth lst n)
  (if (null? lst)
      (error "No element")
      (if (= n 0)
          (car lst)
          (get-nth (cdr lst) (- n 1)))))
```

The running time of `get-nth` is in $\Theta(m)$ where m is the length of the input list. In the worst case, `n` is the length of the list minus 1, so we need to make $m - 1$ recursive calls to `get-nth`, each of which requires constant time.

We use `get-nth` to define `find-matching-pair`:

```
(define (find-matching-pair test lst)
  (define (find-matching-pair-helper test lst i j)
    (if (= i (length lst))
        #f
        (if (= j (length lst))
            (find-matching-pair-helper test lst (+ i 1) 0)
            (if (and (not (= i j))
                     (test (get-nth lst i)
                           (get-nth lst j)))
                (cons (get-nth lst i) (get-nth lst j))
                (find-matching-pair-helper
                 test lst i (+ j 1))))))
  (find-matching-pair-helper test lst 0 0))
```

The first two parameters to `find-matching-pair-helper` are the test procedure and the input list, as with our previous definition. The other parameters are `i` and `j`, which are both numbers used to index through the elements of the list. We need to evaluate the test procedure for every pair of value `i` and `j` that correspond to elements in the input list. As in our analysis of the earlier `find-matching-pair` definition, there are $\Theta(n^2)$ total recursive applications of `find-matching-pair-helper`.

In this case, however, the cost of each application is not constant. For each application, we may need to evaluate `(get-nth lst i)` and `(get-nth lst j)`. Each of these calls has worst case running time in $\Theta(n)$ where n is the length of the input list to `find-matching-pair`, which is also the input list to `get-nth`. On average, the values of `i` and `j` will be the average length of the list = $n/2$, so the expected running time of each call is also in $\Theta(n)$. This means the total running time for `find-matching-pair` is in $\Theta(n^3)$ since there are $\Theta(n^2)$ recursive calls, and each one has running time in $\Theta(n)$.

7.6 Exponential Growth

If the running time of a procedure scales as some power of the size of the input, the procedure's running time grows *exponentially*. When the size of the input increases by one, the running time is multiplied by some constant factor. The growth rate of a function whose output multiplies by w when the input size increases by one is w^k. A common instance of exponential growth is when the running time is in $\Theta(2^n)$. This occurs when the running time doubles when the input size increases by one. Exponential growth is very fast — if our procedure has running time that is exponential in the size of the input, it is not possible to evaluate applications of the procedure on large inputs.

Example 7.1: Power Set. In mathematics, the power set of a set S is the set of all subsets of S. For example, the power set of $\{1, 2, 3\}$ is

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

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

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

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

The `powerset` procedure produces a list of lists as output. 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.

Evaluating an application of `powerset` involves evaluating an application of `append`, and two recursive applications of `(powerset (cdr s))`. Increasing the size of the input list by one, *doubles* the total number of applications of `powerset`. This is the case because to evaluate `(powerset s)`, we need to evaluate `(powerset (cdr s))` two times. This involves doing all the applications of `powerset` that were needed to evaluate `powerset` for a list of length one less than the length of the input list two times. So, the number off applications of `powerset` is 2^n where n is the length of the input list.

Each application involves applications of `map` and `append` (all the other procedures applied in `powerset` are constant time procedures). Both of these procedures are linear in the length of the input list (the first input, in the case of `append`) (see Section 5.3.3 and Exercise 5.3.3). The length of the input list to `map` is the number of elements in the power set of a size $n-1$ set: 2^{n-1}. But, for each application, the value of n is different. The total length of all the input lists to `map` over all of the `powerset` applications on an input list of length n is:

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

which is equivalent to 2^n. So, the running time for all the `map` applications is in $\Theta(2^n)$.

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

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

Exercise 7.5.($\star\star$) We can reduce the number of recursive applications of the `powerset` procedure by avoiding the duplicate work, similarly to the `fast-fibo` procedure at the beginning of this chapter:

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

Is the running time of the `faster-powerset` procedure still in $\Theta(2^n)$? $\diamondsuit$

Example 7.2: Primes. A prime number is a number greater than 1 that has no positive integer divisors other than 1 and itself. To determine if n is a prime number, we can try dividing n by every number between 2 and $n - 1$. If we find a number that divides n with no remainder, then n is not prime. If none of the numbers divide n with no remainder, then n is prime.

The `is-prime?` procedure takes a positive integer as its input and outputs true if and only if the input number is prime:

```
(define (is-prime? n)
  (define (is-prime-helper? n try)
    (if (= n try)
        #t
        (if (zero? (modulo n try))
            #f
            (is-prime-helper? n (+ try 1)))))
  (is-prime-helper? n 2))
```

In the worst case, `n` is prime, and all numbers must be attempted before the base case is reached. The number of recursive applications of `is-prime-helper?` is `(- n 2)` (since we start with `try` as 2). Each application involves applying =, `zero?`, `modulo`, and + (with a constant value). The applications of `zero?` and + (with a constant) are constant time, but the applications of = and `modulo` scale with the input size. For now, we will assume they are constant time.[4] Hence, the running time of `is-prime?` is in $\Theta(n)$ where n is the *value* of the input.

[4]This could not be the case in reality, however, since the time it takes to compute them scales with the size of the input. Since they are primitive procedures, we do not know exactly how they

As explained in Section 7.1.1, however, we want to know how the running time scales as the *size* of the input increases. If the input is a list of constant-sized elements, the size of the input scales with the number of elements in the list. When the input is a number, the size of the input scales with the number if digits in the input. If our numbers are written in base 10, increasing the input size by one increases the value of the input by a factor of 10. The size of the input is $\log v$ where v is the value of the input. The value of the input, v, can be up to $10^d - 1$ where d is the number of digits in the input. So, if `is-prime?` is in $\Theta(n)$ where n is the value of the input, it is in $\Theta(10^d)$ where d is the size (number of digits) of the input. Hence, the `is-prime?` procedure has running time that grows exponentially in the size of the input.

We could improve our `is-prime?` implementation by stopping once we reach a value greater than the square root of the input. This would reduce the number of recursive applications to $\Theta(\sqrt{n})$ where n is the value of the input. The growth rate is still exponential, however. The square root of $\Theta(10^d)$ is $\Theta((\sqrt{10})^d) = \Theta(\sqrt{3.16}^d)$.

Exercise 7.6.($\star\star\star\star$) Devise a procedure for testing primality whose running time grows slower than exponentially.[5] $\diamondsuit$

Exercise 7.7.($\star\star\star\star\star$) Devise a procedure for factoring an input value whose running time grows slower than exponentially.[6] $\diamondsuit$

are implemented, but we can imagine implementing them. We could implement = by comparing the two numbers one digit at a time. The running time of this procedure would scale with the number of digits. Since `n` is a number in base 10, the number of digits in `n` is $\log n$ (the largest d-digit number is $10^d - 1$). Similarly, we could implement `modulo` by performing long division and keeping just the remainder as the final result. The number of steps required to perform long division scales with the number of digits in the divisor. For each step, we need to perform a multiplication of a single digit times the dividend. If we perform long multiplication, the number of steps scales with the number of digits. So, the running time of a straightforward implementation of `modulo` would be in $\log n \log m$ where n is the value of the first input, and m is the value of the second input.

[5]Until 2002 it was generally believed that a primality test with running time faster than exponential did not exist. Then, Manindra Agrawal and two undergraduate students working on their senior thesis projects, Neeraj Kayal and Nitin Saxena, developed such an algorithm and proved that it was correct. See Folkmar Bornemann, *PRIMES Is in P: A Breakthrough for "Everyman"*, Notices of the AMS, Volume 50, Number 5, May 2003; and Manindra Agrawal, Neeraj Kayal and Nitin Saxena, *Primes is in P*, Annals of Mathematics, Volume 160, Number 2, September 2004.

[6]No such procedure is currently known, but no one has proven it can't be done either. The

7.7 Faster than Exponential Growth

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

7.8 Summary

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

Procedues that can produce an output only touching a fixed amount of input regardless of the input size have constant ($O(1)$) running times. Procedures whose running time increases by a constant amount when the input size increases by one have linear (in $\Theta(n)$) running times. Procedures whose running time quadruples when the input size doubles have quadratic (in $\Theta(n^2)$) running times. Procedures whose running time is multiplied by a constant factor when the input size increases by one have exponential (in $\Theta(k^n)$ for some constant k) running times. If a procedure has exponential running time, it can only be evaluated for small inputs.

security of the most widely used public-key cryptosystems, including those that form the basis of most secure Internet transactions, depends on no such procedure existing.