4

Problems and Procedures

A great discovery solves a great problem, but there is a grain of discovery in the solution of any problem. Your problem may be modest, but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by your own means, you may experience the tension and enjoy the triumph of discovery.
George Pólya, *How to Solve It*

Computers are tools for performing computations to solve problems. In this chapter, we consider what it means to solve a problem and explore some strategies for constructing procedures that solve problems.

4.1 Solving Problems

Traditionally, a problem is an obstacle to overcome or some question to answer. Once the question is answered or the obstacle circumvented, the problem is solved and we can declare victory and move on to the next one.

When we talk about writing programs to solve problems, though, we usually have a larger goal. We don't just want to solve *one* instance of a problem, we want a procedure that can solve *all* instances of a problem. For example, we don't just want to find the best route between Charlottesville and Washington, we want to find a procedure that can find the best route between any two locations on a map. The inputs to the procedure are the map, the start location, and the end location, and the procedure should generate the best route as its output.[1] There are infinitely many possible inputs that each specify different instances of the problem of finding the best route between two locations on a map; a general solution to the problem is a procedure that can always find the best route for any possible inputs.

A *problem* is defined by its inputs and the desired property of the output. Recall from Chapter 1 that a procedure is a precise description of a process. To define a procedure that can solve a problem, we need to define a pro-

problem

[1] Actually finding a general procedure that does this is a challenging and interesting problem, that we will return to in Chapter 18.

cedure that takes inputs describing the problem instance and produces a different information process depending on the actual values of its inputs.

A procedure takes zero or more inputs, and produces one output or no outputs[2], as shown in Figure 4.1.

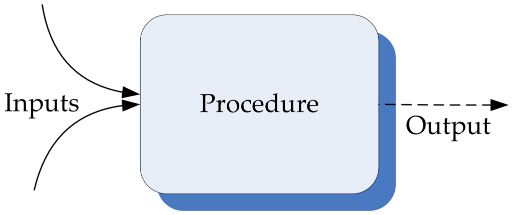


Figure 4.1. A procedure maps inputs to an output.

Our goal in solving a problem is to devise a procedure that takes inputs that define a problem instance, and produces as output the solution to the problem. This means every application of the procedure must eventually finish evaluating and produce an output value.

algorithm

A procedure is guaranteed to always finish is called an *algorithm*. The name algorithm is a Latinization of the name of the Persian mathematician and scientist, Muhammad ibn Mūsā al-Khwārizmī, who published a book in 825 on calculation with Hindu numerals. Al-Khwārizmī was also the responsible for defining algebra.

al-Khwārizmī

Although the name algorithm was not adopted until after al-Khwārizmī's book, algorithms go back much further than that. The ancient Babylonians had algorithms for finding square roots more than 3500 years ago (see Example 4).

divide and conquer

There is no magic wand for solving all problems, but at its core most problem solving involves breaking problems you do not yet know how to solve into simpler and simpler problems until you find problems simple enough that you already know how to solve them. The trick is to find the right subproblems so that they can be combined to solve the original problem. This approach of solving problems by breaking them into simpler parts is known as *divide and conquer*.

The following sections describe a few techniques for solving problems, and illustrate them with some simple examples. We will use these same problem-solving techniques over and over throughout this book. In these

[2] Although procedures can produce more than one output, we limit our discussion here to procedures that produce no more than one output. In the next chapter, we introduce ways to construct complex data, so any number of output values can be packaged into a single output.

examples, we limit ourselves to simple data: all of the problems have one or more numbers as their input, and produce one number as output. In the next chapter, we introduce structured data and revisit these problem solving techniques.

4.2 Composing Procedures

One way to divide a problem is to split it into steps where the output of the first step is the input to the second step, and the output of the second step is the solution to the problem. Each step can be defined by one procedure, and the two procedures can be combined to create one procedure that solves the problem.

Figure 4.2 shows a composition of two functions, *f* and *g*. The output of *f* is used as the input to *g*.

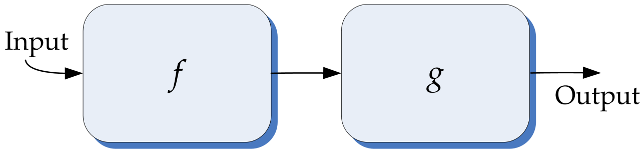


Figure 4.2. Composition.

We can express this composition with the Scheme expression (*g* (*f x*)) where *x* is the input. The written order appears to be reversed from the picture in Figure 4.2. This is because we apply a procedure to the values of its subexpressions: the values of the inner subexpressions must be computed first, and then used as the inputs to the outer applications. So, although *f* appears to the right of *g* in the expression, the subexpression (*f x*) is evaluated first since the evaluation rule for the outer application expression is to first evaluate all the subexpressions.

We can define a procedure that implements the composed procedure by making *x* a parameter:

```
(define fog (lambda (x) (g (f x))))
```

This defines *fog* as a procedure that takes one input and produces as output the composition of *f* and *g* applied to the input parameter. The works for any two procedures, as long as both procedures take a single input parameter.

For example, we could compose the *square* and *cube* procedures from Chapter 3 as:

```
(define sixth-power (lambda (x) (cube (square x))))
```

Then, (*sixth-power* 2) evaluates to 64.

4.2.1 Procedures as Inputs and Outputs

higher-order procedure

So far, all the procedure inputs and outputs we have seen have been numbers. But, the subexpressions of an application can be any expression including a procedure. A *higher-order procedure* is a procedure that takes other procedures as inputs or that produces a procedure as its output. Higher-order procedures give us the ability to write procedures that behave differently based on the procedures that are passed in as inputs.

For example, we can create a generic composition procedure by making *f* and *g* parameters:

```
(define fog (lambda (f g x) (g (f x))))
```

The *fog* procedure takes three parameters. The first two are both procedures that take one input. The third parameter is a value that can be the input to the first procedure.

For example,

```
> (fog square cube 2)
64
> (fog (lambda (x) (+ x 1)) square 2)
9
```

In the second example the first parameter is the procedure produced by the lambda-expression (**lambda** (*x*) (+ *x* 1)). This procedure takes a number as input and produces as output that number plus one. We define *inc* (short for increment) as this procedure:

```
(define inc (lambda (x) (+ x 1)))
```

A more useful composition procedure would separate the input value, *x*, from the composition. The *fcompose* procedure takes two procedures as inputs and produces as output a procedure that is their composition:[3]

[3] We name our composition procedure *fcompose* to avoid collision with the built-in *compose* procedure that behaves similarly.

```
(define fcompose
    (lambda (f g) (lambda (x) (g (f x)))))
```

The body of the *fcompose* procedure is a lambda expression that makes a procedure! Hence, the result of applying *fcompose* to two procedures is not a simple value, but a procedure. The resulting procedure can then be applied to a value.

Here are some examples using *fcompose*:

```
> (fcompose inc inc)
#<procedure>
> ((fcompose inc inc) 1)
3
> (define sixth-power (fcompose square cube))
> (sixth-power 3)
729
> (((fcompose inc square) 2)
9
> ((fcompose square inc) 2)
5
```

Note that the order in which procedures are composed matters. For example, (((*fcompose inc square*) 2)) evaluates to 9 since the input is incremented first, then squared; but, ((*fcompose square inc*) 2) evaluates to 5.

Exercise 4.1. Evaluating procedures. For each expression, give the value to which the expression evaluates. Assume *fcompose* and *inc* are defined as above.

a. (*fcompose* (**lambda** (*x*) (* *x* 2)) (**lambda** (*x*) (/ *x* 2)))

b. ((*fcompose* (**lambda** (*x*) (* *x* 2)) (**lambda** (*x*) (/ *x* 2))) 150)

c. ((*fcompose* (*fcompose inc inc*) *inc*) 2)

Exercise 4.2. Suppose we define *self-compose* as a procedure that composes a procedure with itself:

```
(define (self-compose f) (fcompose f f))
```

Explain how (((*fcompose self-compose self-compose*) *inc*) 1) is evaluated.

Exercise 4.3. Define a procedure *fcompose3* that takes three procedures as input, and produces as output a procedure that is the composition of the three input procedures. For example, ((*fcompose3 abs inc square*) -5) should evaluate to 36. Try to define *fcompose3* two different ways: once without using *fcompose*, and once using *fcompose*.

Exercise 4.4. The *fcompose* procedure only works when both input procedure take one input. Define a *f2compose* procedure that composes two procedures where the first procedure takes two inputs, and the second procedure takes one input. For example, ((*f2compose add abs*) 3 -5) should evaluate to 2.

4.3 Recursive Problem Solving

In the previous section, we used functional composition to break a problem into two procedures that can be composed to produce the desired output. A particularly useful variation on this is when we can break a problem into a smaller version of the original problem.

The goal is to be able to feed the output of one application of the procedure back into the same procedure as its input for the next application, as shown in Figure 4.3.

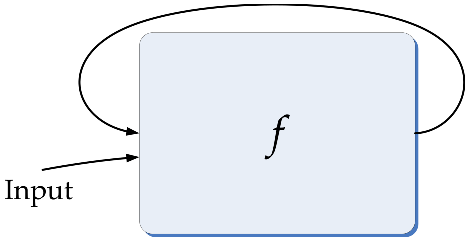


Figure 4.3. Circular Composition.

Here's a corresponding Scheme procedure:

(**define** *f* (**lambda** (*n*) (*f n*)))

Of course, this doesn't work very well![4] Every time an application of *f* is evaluated, it results in another application of *f* to evaluate. This never stops, stop so no output is ever produced and the interpreter will keep evaluating applications of *f* until it is stopper or runs out of memory.

[4] Curious readers should try entering this definition into a Scheme interpreter and evaluating (*f* 0). If you get tired of waiting for an output, in DrScheme you can click the **Stop** button in the upper right corner to interrupt the evaluation.

What we need is a way of making progress and eventually stopping, instead of going around in circles. To make progress, each subsequent application should have a smaller input. Then, the applications can stop when the input to the procedure is simple enough that we know the output directly. This is called the *base case*, similarly to the grammar rules in Section 2.5. In our grammar examples, the base case involved replacing the nonterminal with nothing (e.g., *MoreDigits* ::⇒ ϵ) or with a terminal (e.g., *Noun* ::⇒ **Alice**). In recursive procedures, the base case will provide a solution for some input for which the problem is so simple we already know the answer. When the input is a number, this is often (but not necessarily) when the input is zero or one.

base case

To define a recursive procedure, we need to use an if-expression to test if the input matches the base case input. If it does, the consequent expression is the known answer for the base case. Otherwise, we enter the recursive case and apply the procedure again but with a different input. Each time we apply the procedure we need to make progress towards reaching the base case. This means, the input has to change in a way that gets closer to the base case input. If the base case is for 0, and the original input is a positive number, one way to get closer to the base case input is to subtract 1 from the input value with each recursive application.

This evaluation spiral is depicted in Figure 4.4. With each subsequent recursive call, the input gets smaller, eventually reaching the base case. For the base case application, a result is returned to the previous application. This is passed back up the spiral to produce the final output.

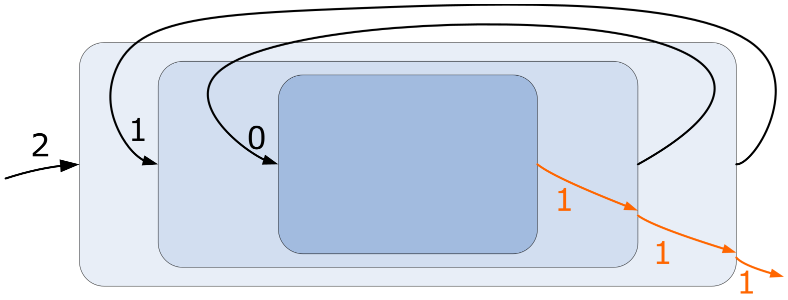


Figure 4.4. Recursive Composition.

Here is the corresponding procedure:

```
(define g
  (lambda (n)
    (if (= n 0) 1 (g (− n 1)))))
```

Unlike the earlier circular *f* procedure, if we apply *g* to any non-negative integer it will eventually produce an output. For example, consider evaluating (g 2). When we evaluate the first application, the value of the parameter n is 2, so the predicate expression (= n 0) evaluates to false and the value of the procedure body is the value of the alternate expression, (g (− n 1)). The subexpression, (− n 1) evaluates to 1, so the result is the result of applying g to 1. As with the previous application, this leads to the application, (g (− n 1)), but this time the value of n is 1, so (− n 1) evaluates to 0. The next application leads to the application, (g 0). This time, the predicate expression evaluates to true and we have reached the base case. The consequent expression is just 1, so no further applications of g are performed and this is the result of the application (g 0). This is returned as the result of the (g 1) application in the previous recursive call, and then as the output of the original (g 2) application.

We can think of the recursive evaluation as winding until the base case is reached, and then unwinding the outputs back to the original application. For this procedure, the output is not very interesting: no matter what positive number we apply g to, the eventual result is 1. To solve interesting problems with recursive procedures, we need to accumulate results as the recursive applications wind or unwind. Examples 1 and 2 illustrate recursive procedures that accumulate the result during the unwinding process. Example 3 illustrates a recursive procedure that accumulates the result during the winding process.

Example 4.1: Factorial. How many different arrangements are there of a deck of 52 playing cards? The top card in the deck can be any of the 52 cards, so there are 52 possible choices for the top card. The second card can be any of the cards except for the card that is the top card, so there are 51 possible choices for the second card. The third card can be any of the 50 remaining cards, and so on, until the last card for which there is only one choice remaining. To determine the total number of possible arrangements we need to multiply the number of choices for each card:

$$52 * 51 * 50 * \cdots * 2 * 1$$

This is known as the *factorial* function (denoted in mathematics using the exclamation point, e.g., 52!). It is defined as:

$$n! = \left\{ \begin{array}{rcl} 1 & : & n = 0 \\ n * (n-1)! & : & n > 0 \end{array} \right.$$

The mathematical definition of factorial is recursive, so it is natural that we can define a recursive procedure that computes factorials:

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

We can now determine the number of deck arrangements by evaluating (*factorial* 52) to get a sixty-eight digit number starting with an 8.

The *factorial* procedure has structure very similar to our earlier definition of the useless recursive *g* procedure. The only difference is the alternative expression for the if-expression: in *g* we used (*g* (− *n* 1)); in *factorial* we added the outer application of ∗: (∗ *n* (*factorial* (− *n* 1))). Instead of just evaluating to the result of the recursive application, we are now combining the output of the recursive evaluation with the input *n* using a multiplication application.

Exercise 4.5. How many different ways are there of choosing an unordered 5-card hand from a 52-card deck?

This is an instance of the "*n* choose *k*" problem (also known as the binomial coefficient): how many different ways are there to choose a set of *k* items from *n* items. There are *n* ways to choose the first item, $n - 1$ ways to choose the second, . . ., and $n - k + 1$ ways to choose the k^{th} item. But, since the order does not matter, some of these ways are equivalent. The number of possible ways to order the *k* items is $k!$, so we can compute the number of ways to choose *k* items from a set of *n* items as:

$$\frac{n * (n-1) * \cdots * (n-k+1)}{k!} = \frac{n!}{(n-k)!k!} \tag{4.1}$$

a. [⋆] Define a procedure *choose* that takes two inputs, *n* (the size of the item set) and *k* (the number of items to choose), and outputs the number of possible ways to choose *k* items from *n*.

b. [⋆] Compute the number of possible 5-card hands that can be dealt from a 52-card deck.

c. [⋆] Compute the likelihood of being dealt a flush (5 cards all of the same suit). In a standard 52-card deck, there are 13 cards of each of the four suits. Hint: divide the number of possible flush hands by the number of possible hands.

Exercise 4.6. When Karl Gauss was in elementary school, his teacher assigned the class the task of summing the integers from 1 to 100 (e.g., $1 + 2 + 3 + \cdots + 100$) to keep them busy. Being the (future) "Prince of Mathematics", Gauss developed the formula for calculating this sum, that

Karl Gauss

is now known as the *Gauss sum*. Had he been a computer scientist, however, and had access to a Scheme interpreter in the late 1700s, he might have instead defined a recursive procedure to solve the problem.

a. [⋆] Define a recursive procedure, *gauss-sum*, that takes a number *n* as its input parameter, and evaluates to the sum of the integers from 1 to *n* as its output. For example, (*gauss-sum* 100) should evaluate to 5050.

b. [⋆] Define a higher-order procedure, *accumulate*, that can be used to make both *gauss-sum* and *factorial*. The *accumulate* procedure should take two inputs: the first is the function used for accumulation (e.g., $*$ for *factorial*, $+$ for *gauss-sum*); the second is the base case value (that is, the value of the function when the input is 0). With your *accumulate* procedure, ((*accumulate* $+$ 0) 100) should evaluate to 5050 and ((*accumulate* $*$ 1) 3) should evaluate to 6.

Hint: since your procedure should produce a procedure as its output, it could start like this:

```
(define (accumulate f base)
  (lambda (n)
     ...
```

Example 4.2: Maximum. In Chapter 3, we saw a procedure, *max*, that takes two inputs and evaluates to the greater input. Here, we consider the problem of defining a procedure that takes as its input a procedure, a low value, and a high value, and outputs the maximum value the input procedure produces when applied to an integer value between the low value and high value input. We will name the inputs *f*, *low*, and *high*. To find the maximum, the *find-maximum* procedure should evaluate the input procedure *f* at every integer value between the *low* and *high*, and produce as output the greatest value found.

To define the procedure, we need to think about this in a way that allows us to combine results from simpler problems to find the result. For the base case, we need to identify a case so simple we already know the answer. Consider the case when *low* and *high* are equal. Then, there is only one value to use, and we know the value of the maximum is (*f low*). So, the base case is (**if** (= *low high*) (*f low*) . . .).

How do we make progress towards the base case? Suppose the value of *high* is equal to the value of *low* plus 1. Then, the maximum value is either the value of (*f low*) or the value of (*f* (+ *low* 1)). We could select it using the *max* procedure:

```
(max (f low) (f (+ low 1)))
```

Of course, we can extend this to the case where *high* is equal to the value of *low* plus 2:

```
(max (f low) (max (f (+ low 1)) (f (+ low 2))))
```

The second operand for the outer *max* evaluation is the maximum value of the input procedure between the low value plus one and the high value input. If we name the procedure we are defining *find-maximum*, then this is the result of (*find-maximum f* (+ *low* 1) *high*). This works whether *high* is equal to (+ *low* 1), or (+ *low* 2), or any other value greater than *high*! Putting things together, we have our recursive definition of *find-maximum*:

```
(define (find-maximum f low high)
  (if (= low high)
    (f low)
    (max (f low)
        (find-maximum f (+ low 1) high)))))
```

Here are a few examples:

```
> (find-maximum (lambda (x) x) 1 20)
20
> (find-maximum (lambda (x) (− 10 x)) 1 20)
9
> (find-maximum (lambda (x) (∗ x (− 10 x))) 1 20)
25
```

Exercise 4.7. [⋆] To find the maximum of a continuous function, we need to evaluate at all numbers in the range, not just the integers. There are infinitely many numbers between any two numbers, however, so this is impossible. We can approximate this, however, by evaluating the function at many numbers in the range.

Define a procedure *find-maximum-continuous* that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *inc*, and produces as output the maximum value of *f* in the range between *low* and *high* where *f* is evaluated at *low*, (+ *low inc*), (+ *low inc inc*), . . ., *high*.

As the value of increment decreases, we expect to find a more accurate maximum value. For example,

```
(find-maximum-continuous (lambda (x) (∗ x (− 5.5 x))) 1 10 1)
```

should evaluate to 7.5. And,

```
(find-maximum-continuous (lambda (x) (* x (− 5.5 x))) 1 10 0.0001)
```

should evaluate to 7.5625.

Exercise 4.8. [⋆] The *find-maximum* procedure we defined evaluates to the maximum value of the input function in the range, but does not provide the input value that produces that maximum output value. Define a procedure that finds the input in the range that produces the maximum output value.

Exercise 4.9. [⋆] Define a *find-area* procedure that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *inc*, and produces as output an estimate for the area under the curve produced by the function *f* between *low* and *high* using the *inc* value to determine how many points to evaluate.

Example 4.3: Euclid's Algorithm. In Book 7 of the *Elements*, Euclid describes an algorithm for finding the greatest common divisor of two non-zero integers. The greatest common divisor is the greatest integer that divides both of the input numbers without leaving any remainder. For example, the greatest common divisor of 150 and 200 is 50 since (/ 150 50) evaluates to 3 and (/ 200 50) evaluates to 4, and there is no number greater than 50 which can divide both 150 and 200 without leaving a remainder.

The *modulo* primitive procedure takes two integers as its inputs and evaluates to the remainder when the first input is divided by the second input. For example, (*modulo* 6 3) evaluates to 0 and (*modulo* 7 3) evaluates to 1.

Euclid's algorithm stems from two properties of integers:

1. If (*modulo a b*) evaluates to 0 then *b* is the greatest common divisor of *a* and *b*.
2. If (*modulo a b*) evaluates to a non-zero integer *r*, then the greatest common divisor of *a* and *b* is the greatest common divisor of *b* and *r*.

We can define a recursive procedure for finding the greatest common divisor closely following Euclid's algorithm:

```
(define (gcd a b)
  (if (= (modulo a b) 0)
      b
      (gcd b (modulo a b))))
```

The structure of the definition is similar to the *factorial* definition: the procedure body is an if-expression and the predicate tests for the base case. For the *gcd* procedure, the base case corresponds to the first property above. It occurs when *b* divides *a* evenly, and the consequent expression is *b*. The alternate expression, (*gcd b* (*modulo a b*)), is the recursive application.

It differs from the alternate expression in the *factorial* definition in that there is no outer application expression (the $*$ application). We do not need to combine the result of the recursive application with some other value as was done in the *factorial* definition, the result of the recursive application is the final result. Unlike the *factorial* and *find-maximum* examples, the *gcd* procedure produces the result in the base case, and no further computation is necessary to produce the final result. When no further evaluation is necessary to get from the result of the recursive application to the final result, a recursive definition is said to be *tail recursive*.

tail recursive

Exercise 4.10. Show the structure of the applications of *gcd* in evaluating (*gcd* 6 9).

Exercise 4.11. [⋆] Provide a convincing argument why the evaluation of (*gcd a b*) will always finish when the inputs are both positive integers.

Exercise 4.12. [⋆] Provide an alternate definition of *factorial* that is tail recursive. To be tail recursive, the expression containing the recursive application cannot be part of another application expression.

Hint: define a *factorial-helper* procedure that takes an extra parameter, and then define *factorial* as:

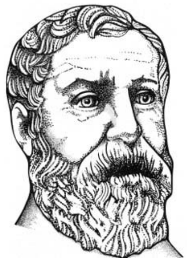
```
(define (factorial n) (factorial-helper n 1))
```

Exercise 4.13. [⋆] Provide an alternate definition of *find-maximum* that is tail recursive.

Exercise 4.14. [⋆⋆] Provide a convincing argument why it is always possible to transform a recursive procedure into an equivalent procedure that is tail recursive.

Example 4.4: Square Roots. One of the earliest known algorithms is a method for computing square roots. It is known as Heron's method after the Greek mathematician Heron of Alexandria who lived in the first century AD who described the method, although it was also known to the Babylonians many centuries earlier. Issac Newton developed a more general method for estimating functions based on their derivatives known as Netwon's method, of which Heron's method is a specialization.

Square root is a mathematical function that take a number, *a*, as input and outputs a value *x* such that $x^2 = a$. For many numbers (for example, 2), the square root is irrational, so the best we can hope for with is a good approx-



Heron of Alexandria

imation. In this example, we will define a procedure *find-sqrt* that takes the target number as input and outputs an approximation for its square root.

Heron's method works by starting with an arbitrary guess, g_0. Then, with each iteration, compute a new guess (g_n is the n^{th} guess) that is a function of the previous guess (g_{n-1}) and the target number (a):

$$g_n = \frac{g_{n-1} + \frac{a}{g_{n-1}}}{2}$$

As n increases, g_n will get closer and closer to the square root of a.

The definition is recursive since we compute g_n as a function of g_{n-1}, so we can define a recursive procedure that computes Heron's method. First, we define a procedure for computing the next guess from the previous guess and the target:

```
(define (heron-next-guess a g)
  (/ (+ g (/ a g)) 2))
```

Next, we define a recursive procedure to compute the n^{th} guess using Heron's method. It takes three inputs: the target number, a, the number of guesses to make, n, and the value of the first guess, g.

```
(define (heron-method a n g)
  (if (= n 0)
     g
     (heron-method a (− n 1) (heron-next-guess a g))))
```

To start, we need a value for the first guess. The choice doesn't really matter — the method works with any starting guess (but will reach a closer estimate quicker if the starting guess is good). We will use 1 as our starting guess. So, we can define a *find-sqrt* procedure that takes two inputs, the target number and the number of guesses to make, and outputs an approximation of the square root of the target number.

```
(define (find-sqrt a guesses)
  (heron-method a guesses 1))
```

Heron's method converges to a good estimate very quickly. For example,

```
> (square (find-sqrt 2 0))
1
> (square (find-sqrt 2 1))
2 1/4
> (square (find-sqrt 2 2))
```

```
2 1/144
> (square (find-sqrt 2 3))
2 1/166464
> (square (find-sqrt 2 4))
2 1/221682772224
> (square (find-sqrt 2 5))
2 1/393146012008229658338304
> (exact->inexact (find-sqrt 2 5))
1.4142135623730951
```

The actual square root of 2 is 1.414213562373095048 . . . so our estimate is correct to 16 digits after only five guesses.

Users of square roots don't really care about the method used to find the square root (or how many guesses are used). Instead, what is important to a square root user is how accurate the estimate is. Can we change our *find-sqrt* procedure so that instead of taking the number of guesses to make as its second input it takes a minimum tolerance value?

Since we don't know the actual square root value (otherwise, of course, we could just return that), we need to measure tolerance as how close the square of the approximation is to the target number. Hence, we can stop when the square of the guess is close enough to the target value.

```
(define (close-enough? a tolerance g)
  (<= (abs (− a (square g))) tolerance))
```

The stopping condition for the recursive definition is now when the guess is close enough. Otherwise, our definitions are the same as before.

```
(define (heron-method-tolerance a tolerance g)
  (if (close-enough? a tolerance g)
     g
     (heron-method-tolerance a tolerance (heron-next-guess a g))))
```

```
(define (find-sqrt-approx a tolerance)
  (heron-method-tolerance a tolerance 1))
```

Note that the value passed in as *tolerance* does not change with each recursive call. We are making the problem smaller by making each successive guess closer to the required answer.

Here are some example interactions with *find-sqrt-approx*:

```
> (exact->inexact (square (find-sqrt-approx 2 0.01)))
2.0069444444444446
> (exact->inexact (square (find-sqrt-approx 2 0.0000001)))
2.000000000004511
```

Excursion 4.1: Square Roots. Scheme provides a built-in *sqrt* procedure. How accurate is it? Can you produce more accurate square roots than the built-in *sqrt* procedure? (Why doesn't the built-in procedure do better?)

4.4 Evaluating Recursive Applications

Evaluating an application of a recursive procedure follows the evaluation rules just like any other expression evaluation. It may be confusing, however, to see that this works because of the apparent circularity of the procedure definition. Here, we show in detail the evaluation steps for evaluating (*factorial* 2). The evaluation and application rules refer to the rules summary in Section 3.8. We first show the complete evaluation following the substitution model evaluation rules in full gory detail, and later consider a subset containing the most revealing steps. Stepping through even a fairly simple evaluation using the evaluation rules is quite tedious, and not something humans should do very often (that's what we have computers for!) but instructive to do once to understand exactly how an expression is evaluated.

The evaluation rule for an application expression does not specify the order in which the subexpressions are evaluated. A Scheme interpreter is free to evaluate them in any order. Here, we choose to evaluate the subexpressions in the order that is most readable. As long as none of the expressions have side effects, the value produced by an evaluation does not depend on the order in which the subexpressions are evaluated.

In the evaluation steps, we use `typewriter` font for uninterpreted Scheme expressions and sans-serif font to show values. So, `2` represents the Scheme expression that evaluates to the number 2.

1. `(factorial 2)`

   Evaluation Rule 3(a): Application subexpressions

2. `(factorial 2)`

   Evaluation Rule 2: Name

3. `((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 2)`

   Evaluation Rule 4: Lambda

4. (((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) `2`)

Evaluation Rule 1: Primitive

5. ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 2)
Evaluation Rule 3(b): Application, Application Rule 2

6. (if (= 2 0) 1 (* 2 (factorial (- 2 1))))
Evaluation Rule 5(a): If predicate

7. (if (= 2 0) 1 (* 2 (factorial (- 2 1))))
Evaluation Rule 3(a): Application subexpressions

8. (if (= 2 0) 1 (* 2 (factorial (- 2 1))))
Evaluation Rule 1: Primitive

9. (if (= 2 0) 1 (* 2 (factorial (- 2 1))))
Evaluation Rule 3(b): Application, Application Rule 1

10. (if false 1 (* 2 (factorial (- 2 1))))
Evaluation Rule 5(b): If alternate

11. (* 2 (factorial (- 2 1)))
Evaluation Rule 3(a): Application subexpressions

12. (* 2 (factorial (- 2 1)))
Evaluation Rule 1: Primitive

13. (* 2 (factorial (- 2 1)))
Evaluation Rule 3(a): Application subexpressions

14. (* 2 (factorial (- 2 1)))
Evaluation Rule 3(a): Application subexpressions

15. (* 2 (factorial (- 2 1)))
Evaluation Rule 1: Primitive

16. (* 2 (factorial (- 2 1)))
Evaluation Rule 3(b): Application, Application Rule 1

17. (* 2 (factorial 1))
Continuing Evaluation Rule 3(a); Evaluation Rule 2: Name

18. (* 2 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 1))
Evaluation Rule 4: Lambda

19. (* 2 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 1))
Evaluation Rule 3(b): Application, Application Rule 2

20. (* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))
Evaluation Rule 5(a): If predicate

21. (* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))
Evaluation Rule 3(a): Application subexpressions

22. (* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))
Evaluation Rule 1: Primitives

23. (* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))
Evaluation Rule 3(b): Application Rule 1

24. (* 2 (if false 1 (* 1 (factorial (- 1 1)))))
Evaluation Rule 5(b): If alternate

25. (* 2 (* 1 (factorial (- 1 1))))
Evaluation Rule 3(a): Application

26. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 1: Primitives
27. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 3(a): Application
28. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 3(a): Application
29. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 1: Primitives
30. `(* 2 (* 1 (factorial (- 1 1))))`
    Evaluation Rule 3(b): Application, Application Rule 1
31. `(* 2 (* 1 (factorial 0)))`
    Evaluation Rule 2: Name
32. `(* 2 (* 1 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 0)))`
    Evaluation Rule 4, Lambda
33. `(* 2 (* 1 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 0)))`
    Evaluation Rule 3(b), Application Rule 2
34. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 5(a): If predicate
35. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 3(a): Application subexpressions
36. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 1: Primitives
37. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 3(b): Application, Application Rule 1
38. `(* 2 (* 1 (if true 1 (* 0 (factorial (- 0 1))))))`
    Evaluation Rule 5(b): If consequent
39. `(* 2 (* 1 1))`
    Evaluation Rule 1: Primitives
40. `(* 2 (* 1 1))`
    Evaluation Rule 3(b): Application, Application Rule 1
41. `(* 2 1)`
    Evaluation Rule 3(b): Application, Application Rule 1
42. **2**
    Evaluation finished, no unevaluated expressions remain.

The key to evaluating applications of recursive procedures is the evaluation rule for the if special form. If the if-expression were evaluated like a regular application all subexpressions would be evaluated, and the alternative expression with the recursive call would never finish evaluating! Since the evaluation rule for if requires that the predicate expression is evaluated and the alternative expression is *not* evaluated when the predicate expression is true, the circularity in the definition ends when the predicate expression evaluates to true. This is the base case of the recursion, when (= *n* 0) evaluates to true. This occurs when (*factorial* 0) is evaluated, and instead of producing another recursive call it evaluates to the value of the consequent expression, 1.

Exercise 4.15. These exercises test your understanding of the (*factorial* 2) evaluation. Try to answer them yourself before reading the remainder of this section.

a. In step 5, the second part of the application evaluation rule, Rule 3(b), is used. In which step does this evaluation rule complete?

b. In step 11, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?

c. In step 25, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?

d. How many times is Evaluation Rule 5 (If) used?

e. [⋆] In order to evaluate (*factorial* 3) how many times would Evaluation Rule 2 be used to evaluate the name *factorial*?

Exercise 4.16. For which input values *n* will an evaluation of (*factorial* *n*) eventually reach a value? For values where the evaluation is guaranteed to finish, make a convincing argument why it must finish. For values where the evaluation does not finish, explain why.

4.4.1 The Evaluation Stack

We can see the structure of the evaluation process better if we focus just on the most revealing stems. Here are the evaluation steps that involve applications of the *factorial* procedure extracted from the full evaluation and indented to line up the steps from each application evaluation:

```
1.  (factorial 2)
17.     (* 2 (factorial 1))
31.         (* 2 (* 1 (factorial 0)))
40.         (* 2 (* 1 1))
41.     (* 2 1)
42. 2
```

In Step 1, we start to evaluate (*factorial* 2). The final result of this evaluation is found in Step 42. To evaluate (*factorial* 2), we follow the evaluation rules, eventually reaching the body expression of the if-expression in the factorial definition. This is Step 17: (∗ 2 (*factorial* 1)). To evaluate this expression, we need to evaluate the (*factorial* 1) subexpression.

We can think of each of these application evaluations as a stack, similarly to how we processed RTN subnetworks in Section 2.4.1. A stack has the property that the first item pushed on the stack will be the last item removed—all the items pushed on top of this one must be removed before this item can be removed. For application evaluations, the elements on the stack are expressions to evaluate. To finish evaluating the first expression, all of its component subexpressions must be evaluated. Hence, the first application evaluation started will be the last one to finish.

At Step 17, the first evaluation is in progress, but to complete it we need the value resulting from the second evaluation. The second evaluation results in the body expression, (∗ 1 (*factorial* 0)), shown for Step 31. At this point, the evaluation of (*factorial* 2) is stuck in Evaluation Rule 3, waiting for the value of (*factorial* 1) subexpression. The evaluation of the (*factorial* 1) application leads to the (*factorial* 0) subexpression, which must be evaluated before the (*factorial* 1) evaluation can complete. In Step 40, the (*factorial* 0) subexpression evaluation has completed and produced the value 1. Now, the (*factorial* 1) evaluation can complete, producing 1 as shown in Step 41. Once the (*factorial* 1) evaluation completes, all the subexpressions needed to evaluate the expression in Step 17 are now evaluated, and the evaluation completes in Step 42.

4.5 Developing Complex Programs

To develop and use more complex procedures it will be useful to learn a few techniques that will help you understand what is going on when your procedures are evaluated (and fix problems when things are not working as intended). It is very rare for a first attempt to create a program to be completely correct, even for an expert programmer. Instead, it is best to build programs incrementally, by building and testing small components one at a time.

debugging

The process of fixing broken programs is known as *debugging*. The name comes from the early days of computing when real bugs in the machine could cause problems. The key to debugging effectively is to be systematic and thoughtful. It is a good idea to take notes to keep track of what you have learned and what you have tried. Thoughtless debugging can be very frustrating, and is unlikely to lead to a correct program.

Moth from Grace Hopper's notebook, 1947

A good strategy for debugging is to:

1. First, make sure you understand the intended behavior of your procedure. Think of a few representative inputs, and what the expected output should be.
2. Then, do experiments to observe the actual behavior of your procedure. Does it work correctly for some inputs but not others? What is the relationship between the actual outputs and the desired outputs?
3. Next, make changes to your procedure and retest it. If you are not sure what to do, make changes in small steps and carefully observe the impact of each change.

As your programs get more complex, you'll want to follow this strategy but at the level of sub-components. For example, you can try debugging at the level of one expression before trying the whole procedure. If you break your program down into several procedures, you can test and debug each procedure independently. The smaller the unit you are considering at one time, the easier it will be to understand what the problem is and how to fix it.

DrScheme provides many useful and powerful features to aid debugging, although the most useful tool there is for debugging is using your brain to think carefully about what your program should be doing and how its observed behavior differs from the desired behavior. Next, we describe two simple techniques that are helpful for observing program behavior.

4.5.1 Printing

One useful procedure built-in to DrScheme is the *display* procedure. It takes one input, and produces no output. Instead of producing an output, it prints out the value of the input (it will appear in purple in the Interactions window). We can use *display* to observe what a procedure is doing before it produces an output. For example, if we add a (*display n*) expression at the beginning of our *factorial* procedure we can see all the intermediate calls. To make each printed value appear on a separate line, we use the *newline* procedure. The *newline* procedure takes not inputs and produces no output, but prints out a new line.

```
(define (factorial n)
  (display "Enter factorial: ") (display n) (newline)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Now, evaluating (*factorial* 3) produces:

```
Enter factorial: 3
Enter factorial: 2
Enter factorial: 1
Enter factorial: 0
6
```

For printing out lots of strings and values, the built-in *printf* procedure is more useful. The *printf* procedure takes one or more inputs. The first input is a string (a sequence of characters enclosed in double quotes). The string can include special `~a` markers that print out values of objects inside the string. Each `~a` marker is matched with a corresponding input, and the value of that input is printed in place of the `~a` in the string. Another special marker, `~n`, prints out a new line inside the string. Using *printf*, we can define our *factorial* procedure with printing as:

```
(define (factorial n)
  (printf "Enter factorial: ~a~n" n)
  (if (= n 0)
     1
     (* n (factorial (− n 1)))))
```

side effects

Note that *display*, *printf*, and *newline* do not produce output values. Instead, they are applied to produce *side effects*. A side effect is something that changes the state of a computation. In this case, the side effect is printing in the Interactions window. Side effects make reasoning about what programs do much more complicated since the order in which events happen now matters. We will mostly avoid using procedures with side effects until Chapter 11, but printing procedures are so useful that we introduce them here.

4.5.2 Tracing

DrScheme provides a more automated way to observe applications of procedures. We can use tracing to observe the beginning of a procedure evaluation (including the procedure inputs), and when the evaluation complete (including the procedure outputs). To use tracing, it is necessary to first load the tracing library by evaluating this expression:

```
(require (lib "trace.ss"))
```

This defines the *trace* procedure that takes one input, a compound procedure (trace does not work for primitive procedures). After evaluating (*trace proc*), every time an application of *proc* is evaluated the tracing will print out the procedure name and its inputs at the beginning of the application evaluation, and the value of the output at the end of the application evaluation. If there are other applications before the first application finishes evaluating, these will be printed too, but indented so it is possible to match up the beginning and end of each application evaluation. For example (the trace outputs are shown in `typewriter` font),

```
> (trace factorial)
> (factorial 3)
|(factorial 3)
| (factorial 2)
| |(factorial 1)
| | (factorial 0)
| | 1
| |1
| 2
|6
6
```

From the trace, one can see that (*factorial* 3) is evaluated first; within its evaluation, (*factorial* 2), then (*factorial* 1), and (*factorial* 0) are evaluated. The outputs of each of these applications is lined up vertically below the application entry trace.

Excursion 4.2: Recipes for π. The value π is the defined as the ratio between the circumference of a circle and its diameter. One way to calculate the value of π is the Gregory-Leibniz series (which was actually discovered by the Indian mathematician Mādhava in the 14^{th} century):

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

This summation converges to π. The more terms that are included, the closer the computed value will be to the actual value of π.

a. [⋆] Define a procedure *compute-pi* that takes as input n, the number of terms to include and outputs an approximation of π computed using the first n terms of the Gregory-Leibniz series. For example, (*compute-pi* 1) should evaluate to 4 and (*compute-pi* 2) should evaluate to 2 2/3 (= $\frac{4}{1} - \frac{4}{3}$). For higher terms, use the built-in procedure *exact->inexact* to see the decimal value. For example (*exact->inexact* (*compute-pi* 10)) should evaluate to 3.0418396189294024 and (*exact->inexact* (*compute-pi* 10000)) evaluates (after a long wait!) to 3.1414926535900434.

The Gregory-Leibniz series is fairly simple, but it takes an awful long time to converge to a good approximation for π — only the first digit is correct after 10 terms, and after summing 10000 terms only the first four digits are correct.

Mādhava discovered another series for computing the value of π that converges much more quickly:

$$\pi = \sqrt{12} * (1 - \frac{1}{3*3} + \frac{1}{5*3^2} - \frac{1}{7*3^3} + \frac{1}{9*3^4} - \ldots)$$

Mādhava computed the first 21 terms of this series, finding an approximation of π that is correct for the first 12 digits: 3.14159265359.

b. [⋆⋆] Define a procedure *cherry-pi* that takes as input n, the number of terms to include and outputs an approximation of π computed using the first n terms of the faster Madhava series. (Continue reading for hints.)

To define *cherry-pi*, you will first want to define two helper functions: *cherry-pi-filling*, that takes one input, n, and computes the sum of the first n terms in the series without the $\sqrt{12}$ factor, and *cherry-pi-term* that takes one input n and computes the value of the n^{th} term in the series (without alternating the adding and subtracting). For example, (*cherry-pi-term* 1) should evaluate to 1 and (*cherry-pi-term* 2) should evaluate to 1/9. Then, you can define *cherry-pi* as:

```
(define (cherry-pi terms) (* (sqrt 12) (cherry-pi-helper terms)))
```

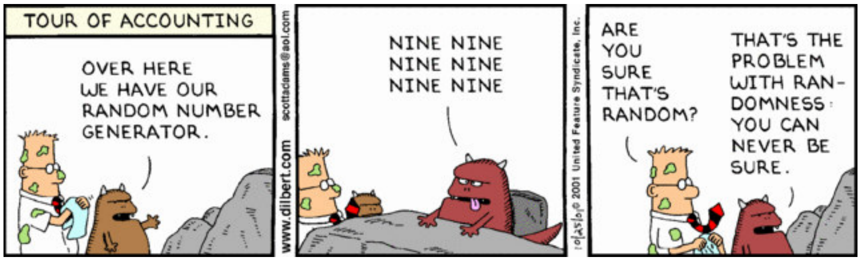
Here, we use the built-in *sqrt* procedure that takes one input and produces as output an approximation of its square root. The accuracy of the *sqrt* procedure[5] limits the number of digits of π that can be correctly computed using this method (see Example 4 for ways to compute a more accurate approximation for the square root of 12). You should be able to get a few more correct digits than Mādhava was able to get without a computer 600 years ago, but to get more digits would need a more accurate *sqrt* procedure or another method for computing π.

To compute the powers ($3^1, 3^2, 3^3, \ldots$), you can use the built-in *expt* procedure (which takes two inputs, a and b, and produces a^b as its output), but

[5] To test its accuracy, try evaluating (*square* (*sqrt* 12)).

you should also be able to figure out a way to define your own procedure to compute a^b for any integer inputs a and b.

c. [⋆⋆⋆] Find a procedure for computing enough digits of π to find the *Feynman point* where there are six consecutive 9 digits. This point is name for Richard Feynman, who quipped that he wanted to memorize π to that point so he could recite it as "... nine, nine, nine, nine, nine, nine, and so on".


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
scottadams@aol.com
www.dilbert.com
NINE NINE NINE NINE NINE NINE
© 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

Producing good random numbers is an important and interesting problem, as is the question of determining whether a given sequence of numbers is random. We consider it in Chapter 19.

Excursion 4.3: Recursive Definitions and Games. Many games can be analyzed by thinking recursively. For this excursion, we consider how to develop a winning strategy for some two-player games. In all the games, we assume player 1 moves first, and the two players take turns until the game ends. The game ends when the player who's turn it is cannot move; the other player wins.

A strategy is a *winning strategy* if it provides a way to always select a move that wins the game, regardless of what the other player does. A good approach for thinking about these games is to work backwards from the winning position (which corresponds to the base case in a recursive definition). If the game reaches a winning position for player 1, then player 1 is guaranteed to win. Moving back one move, if the game reaches a position where it is player 2's move, but all possible moves lead to a winning position for player 1, then player 1 is guaranteed to win. Continuing backwards, if the game reaches a position where it is player 1's move, and there is a move that leads to a position where all possible moves for player 2 lead to a winning position for player 1, then player 1 is guaranteed to win.

The first game we will consider is called *Nim*. Variants on Nim have been played widely over many centuries, and no one is quite sure where the name comes from.

We'll start with a simple variation on the game that was called *Thai 21* when it was used as an Immunity Challenge on *Survivor*. In this version of Nim, the game starts with a pile of 21 stones. One each turn, a player removes one, two, or three stones. The player who removes the last stone wins, since the other player cannot make a valid move on the following turn.

a. [⋆] What should the player who moves first do to ensure she can always win the game? (Hint: start with the base case, and work backwards. Think about a game starting with 5 stones first, before trying 21.)

b. [⋆] Suppose instead of being able to take 1–3 stones with each turn, you can take 1–*n* stones where *n* is some number greater than or equal to 1. For what values of *n* should the first player always win (when the game starts with 21 stones)?

c. [⋆] Define a procedure that plays a winning game of *Thai 21*. Your procedure should take one input, representing the number of stones left, and produce as output a number indicating the number of sticks to take. If there is no winning strategy from the given input number of sticks, your procedure should output 0.

d. [⋆] Generalize your procedure to support the version of the game where the maximum number of stones that can be removed is a parameter. The new procedure should take two inputs, the first is the number of stones left, and the second is the maximum number of stones that can be removed on each turn, and produce as output the number of stones to remove.

In the standard Nim, instead of just having one heap of stones, the game starts with three heaps. At each turn, a player removes any number of stones from any one heap (but may not remove stones from different heaps). We can describe the state of a 3-heap game of Nim using three numbers, representing the number of stones in each heap. For example, the *Thai 21* game starts with the state (21 0 0) (one heap with 21 stones, and two empty heaps). (With the standard Nim rules, this would not be an interesting game since the first player can simply win by removing all 21 stones from the first heap.) (If you get stuck, you'll find many resources about Nim on the Internet; but, you'll get a lot more out of this if you try and solve it yourself.)

e. What should the first player do to win if the starting state is (2 1 0)?

f. Which player should win if the starting state is (2 2 1)?

g. Which player should win if the starting state is (2 2 2)?

h. [⋆] Which player should win if the starting state is (5 6 7)?

i. [⋆⋆] Describe a strategy for always winning a game of Nim starting from any position.

The final game we consider is the "Corner the Queen" game invented by Rufus Isaacs (and described by Martin Gardner [Gar97]). The game is played using a single Queen on a arbitrarily large chessboard as shown in Figure 4.5.

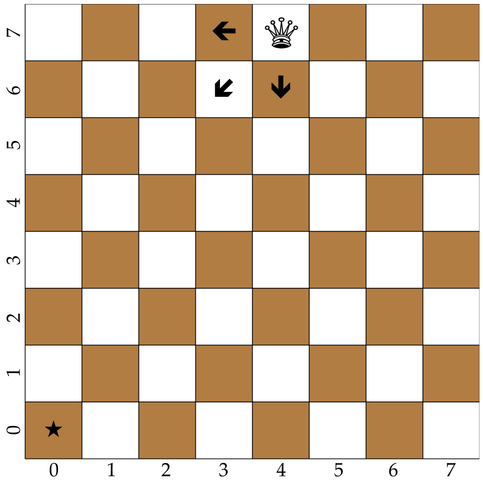


Figure 4.5. Cornering the Queen.

On each turn, a played moves the Queen one or more squares in either the left, down, or diagonally down-left direction (unlike a standard chess Queen, in this game the queen may not move right, up or up-right). As with the other games, the last player to make a legal move wins. For this game, once the Queen reaches the bottom left square marked with the $\star$, there are no moves possible. Hence, the player who moves the Queen onto the $\star$ wins the game. We name the squares using the numbers on the sides of the chessboard with the column number first. So, the Queen in the picture is on square (4 7).

j. Identify all the starting squares for which the first played to move can win right away. (Your answer should generalize to any size square chessboard.)

k. Suppose the Queen is on square (2 1) and it is your move. Explain why there is no way you can avoid losing the game.

l. If the Queen is on square (5 6) and it is your move, how should you move to guarantee you will win the game?

m. [$\star$] Given the shown starting position (with the Queen at (4 7), would you rather be the first or second player?

n. [$\star$] Describe a strategy for winning the game (when possible). Explain from which starting positions it is not possible to win (assuming the other player always makes the right move).

o. [⋆] Define a variant of Nim that is essentially the same as the "Corner the Queen" game. (The game is known as Wythoff's Nim.)

4.6 Summary

By breaking problems down into simpler problems we can develop solutions to complex problems. Many problems can be solved by combining instances of the same problem on simpler inputs. When we define a procedure to solve a problem this way, it needs to have a predicate expression to determine when the base case has been reached, a consequent expression that provides the value for the base case, and an alternate expression that defines the solution to the given input as an expression using a solution to a slightly smaller input.

Our general problem solving strategy is:

1. Be optimistic! Assume you can solve it.
2. Think of the simplest version of the problem, something you can already solve. This is the base case.
3. Consider how you would solve a big version of the problem by using the result for a slightly smaller version of the problem. This is the recursive case.
4. Combine the base case and the recursive case to solve the problem.

For problems involving numbers, the base case will often be when the input value is zero (but not always, as we saw in the *find-maximum* value, where the base case is reached when the difference between two of the input values is zero). The way the problem size is reduced is by subtracting some value (usually 1) from one of the inputs.

In the next chapter, we introduce more complex data structures. For problems involving complex data, the same strategy will often work, but we will use other ways to identify a base case and to shrink the problem size.