Prime numbers have fascinated mathematicians for many thousands of years. Ancient Greek, Egyptian, and Chinese philosophers realized that most integers are composite numbers, meaning they are the product of two smaller integers, but some special numbers, the prime numbers, are not evenly divisible by any smaller number except 1. These mathematicians developed many important theories about prime numbers and their relationships. For most of the last two thousand years prime numbers were only of theoretical interest, but recently they have become important elements in a variety of important “real world” applications in computer science. The security of messages transmitted using public key cryptography, the most widely used method for transferring sensitive information on the Internet, relies heavily on properties of prime numbers that were discovered thousands of years ago.

The Sieve of Eratosthenes, one of the earliest algorithms invented, is still used to generate lists of prime numbers. To see why the algorithm is called a “sieve,” imagine numbers are like rocks, where the shape of each rock is determined by its factors, and that we have a magic bowl with adjustable holes in the bottom. To find prime numbers, we could put a bunch of rocks in the bowl, and then adjust the holes on the bottom of the bowl to match powers of 2. When we shake the bowl, all the rocks that are multiples of 2 (i.e. the even numbers) will fall out. We could then repeat to shake out multiples of 3, and so on. The goal is to shake the bowl long enough so that only prime numbers are left.

The way the algorithm really works is to start by picking an upper limit. We'll call this limit $n$, and our goal is to make a list of all the prime numbers less than $n$. The first step is make a list of all the integers from 2 to $n$ (Figure 3.1a shows an example for $n = 20$). The first number on the list, 2, is prime, but all the multiples of 2 are not, so scan through the list and remove all the multiples of 2 (Figure 3.1b). The lowest remaining number, 3, is also a prime, so next we go through and remove all the multiples of 3 (Figure 3.1c). Keep repeating this process of marking the lowest remaining number as prime and removing its multiples until all that remains are prime numbers.

The method is fairly straightforward, but this description leaves out a few details. The first detail is how to know when we can stop searching for multiples. If you do the next few steps of the example, where $n = 20$, you'll soon notice that you aren't crossing out any more numbers. For example, after you “shake the bowl” to cross out multiples of 11 the only remaining numbers are 13, 17, and 19. You could continue and search for multiples of 13, but you can tell at a glance that there aren't any – the smallest composite number that
is a multiple of 13 is $2 \times 13 = 26$, and this list only goes up to 20. So your intuition is that you're done – the only remaining numbers are prime.

We also need to define what it means to “mark a number as prime” or “remove a number.” How we define these operations depends on the technology we have available. If we use paper and pencil we can do something like what's shown in Figure 3.1 and circle prime numbers as they are found and cross out their multiples. If we're using a whiteboard it might be neater and easier to follow if we erase multiples instead of crossing them out. If we have a computer, and can figure out how to create lists of numbers and selectively remove numbers, and can spell out all the other operations (including when to stop) then we can get the computer to generate the list for us.

This chapter contains a set of exercises that will explore the Sieve of Eratosthenes algorithm. We will replace the paper and pencil technology shown in Figure 3.1 with a computer-based environment that uses the Ruby programming language to create and manipulate lists of numbers. We will see how Ruby provides a workbench where we can create lists of numbers and write expressions in Ruby that will scan lists to remove non-primes. Letting the computer do the “housekeeping” allows us to focus on the essential concepts in the algorithm.

The Sieve of Eratosthenes is a simple algorithm, and using Ruby to set up experiments on lists of numbers might seem like overkill. But this same approach will be used in later chapters to investigate more challenging problems and more complex algorithms. Consider this chapter a “warm-up exercise” that will introduce Ruby and several techniques that will be used throughout the book.
3.1 The mod Function

Tutorial Project

Take some time to make sure you understand the basic idea of the Sieve of Eratosthenes. Make a copy of the list in Figure 3.1c, then carry out the next few rounds of marking a number as prime and sifting out its multiples. How many rounds did you have to do before you were left with all prime numbers? Can you think of a formula based on the upper limit $n$ that describes a general rule for when you can stop?

3.1 The mod Function

If we are going to use Ruby to generate a list of prime numbers one of the first questions we need to address is how to use Ruby to tell if a number is prime. In the last chapter we saw that arithmetic expressions in Ruby can be written with the four basic arithmetic operators (+, −, *, and /) plus an operator for finding exponents (**). Ruby has another operator for specifying divisions involving integers. The new operator, which uses the symbol %, returns the remainder of a division operation. Here is an example:

```ruby
>> 10 % 3
=> 1
```

The result is 1 because $10 \div 3 = 3 \text{ R } 1$.

This new operator, called the “mod” operator, gives us a very useful tool when we’re looking for prime numbers. 10 is not prime because it is evenly divisible by 2 and 5 (or, put another way, the two factors of 10 are 2 and 5). 13 is prime because it has no factors; the only way to write 13 as the product of two other numbers is as $1 \times 13$. The reason the mod operator is useful for finding prime numbers is that the remainder of dividing a number by one of its factors is 0, e.g. $10 \div 5 = 2 \text{ R } 0$ and $10 \div 2 = 5 \text{ R } 0$. We can determine whether or not an integer is prime by trying to divide it by every smaller integer; if the remainder is non-zero after every attempt then we have found a prime number. The mod operator will come in handy in a surprisingly large number of projects throughout this book.

Tutorial Project

1. Use IRB to evaluate $18 \div 3$.
2. What does IRB print for the value of $19 \div 3$? Can you explain why?
3. What does IRB print for $19.0 \div 3$?
4. What is the result of $19 \% 3$ of $18 \% 3$?
5. Try some experiments on your own. Pick any pair of numbers $n$ and $m$ such that $n$ is a multiple of $m$ (for example 25 and 5). Ask Ruby to calculate the quotient using the / operator and to find the remainder using the % operator. Is the remainder always 0 when $n$ is a multiple of $m$?
6. Try some experiments involving pairs of numbers $n$ and $m$ where $n$ is not a multiple of $m$. Is the remainder always non-zero?
3.2 Containers

The main goal for this chapter is to use Ruby to explore an algorithm for generating a list of prime numbers. We've seen how we can manipulate numbers and numeric expressions, and now we are ready to tackle the problem of how to create a list of numbers.

In everyday life a list is a collection of items. Some lists are ordered, but many are just random collections of information. The list of ingredients in a recipe are typically ordered because cooks like to have them presented in the order in which they are used, but shopping lists are often just random or semi-random collections of information.

Mathematicians also deal with collections. The concept of a set is one of the fundamental concepts in mathematics. Usually items in a set are not given in any particular order, but if order is important mathematicians say they are working with an “ordered set” or a “sequence.”

Computer scientists have a wide variety of ways of organizing collections of data, including sets, sequences, graphs, trees, and many other structures. In programming language terminology, a structure that holds a collection of data items is known as a container. Containers are going to be used in almost every project in this book. This chapter is no exception, and it introduces a very simple but useful container that we will use to implement lists of numbers used in the Sieve of Eratosthenes algorithm.

In Ruby the container we are going to use to implement a list of numbers is called an “array”. Ruby arrays are ordered collections of data. The items in a list can be numbers, strings, or any other type of data object we can create in Ruby.

The simplest way to make a list in Ruby is to write the items in the list between square brackets, separated by commas. Thus to make a list of the numbers from 1 to 5 we would write

```ruby
>> a = [1, 2, 3, 4, 5]
=> [1, 2, 3, 4, 5]
```

The expression on the first line above is an assignment statement. Like the assignments in the previous chapter, it has the name of a variable on the left side; the only difference here is the value on the right side is an array of numbers. Ruby handles this assignment like it did the others: if the variable a does not exist it is created, and the name a is associated with the list of elements (Figure 3.2).

---

**Arrays vs. Lists**

In the computer science and programming literature the words “array” and “list” have subtle differences. Both are names for ordered collections of data, but lists are more flexible than arrays. Arrays typically have a predetermined size, but lists can grow and shrink.

The designers of the Ruby language had historical reasons for choosing the term “array”, and the name has stuck. For the most part in this book we do not need to be concerned with the differences between arrays and lists, and we can simply use Ruby's arrays to implement lists of items.

© 2008 John S. Conery. Please do not distribute without permission.
3.2 Containers

Figure 3.2: The object store after creating a list of numbers. The array object is a “container” that holds references to other objects.

Note that Ruby responds to this assignment the same way it did the others, by printing the value of the new variable. To verify that \( a \) is an array, we can just ask Ruby to print its value:

\[
\text{\texttt{\textgreater\textless a}}
\Rightarrow \text{\texttt{[1, 2, 3, 4, 5]}}
\]

There are all sorts of interesting things we can do with our new array. We can add items, delete items, invert the order, make copies, and carry out dozens of other useful operations. For this project, though, we only need to know how to do a few of these things, and we’ll put off investigating the other operations until we need them.

Most operations on arrays are performed by calling a method. In the previous chapter, we saw examples of methods that implement math functions, and we saw how a method is called by writing its name in an expression. But for arrays, most methods are called by writing the name of the array, a period, and then the name of the operation. For example, there is an operation named \textit{length} that will compute the number of items in an array. After making the array named \( a \) in the previous example, we can ask Ruby how many items are in it by typing

\[
\text{\texttt{\textgreater\textless a.length}}
\Rightarrow 5
\]

Two other examples are \textit{first}, which returns the value at the front of the list, and \textit{last}, which returns the value at the end:

\[
\begin{align*}
\text{\texttt{\textgreater\textless a.first}} &= 1 \\
\text{\texttt{\textgreater\textless a.last}} &= 5
\end{align*}
\]

We can pass parameters to array methods, just like we passed parameters to methods in the \texttt{Math} module. An example of an array method that has a parameter is \textit{include?}, which returns \texttt{true} or \texttt{false}, depending on whether the value passed as a parameter is contained somewhere in the array:

\[
\begin{align*}
\text{\texttt{\textgreater\textless a.include?(5)}} &= \texttt{true} \\
\text{\texttt{\textgreater\textless a.include?(6)}} &= \texttt{false}
\end{align*}
\]
Note also that, as was the case with the methods we saw in the previous chapter, the values returned by array methods are plugged in to the surrounding expression where they are used just like any other values:

```ruby
global = [2, 4, 6, 8]
print global
```

```plaintext
2, 4, 6, 8
```

Some array methods are invoked when we use an operator in an expression. A method we want to use for this project is identified by the `<<` symbol (two less-than signs in a row, with no space between them). This method attaches a new value to the end of an array. To call this method we write it as an operator in an expression, with an array on the left side and an item to append to the array on the right side. For example, to append the number 6 to the end of `a`:

```ruby
global << 6
```

```plaintext
[1, 2, 3, 4, 5, 6]
```

In all of the examples in this section IRB is following the same general set of rules it did in the previous chapter: it waits for you to type an expression, it evaluates the expression, and it prints the result. What we’ve seen in this section is that:

- values in an expression can be arrays as well as numbers and strings
- operations on arrays are performed by methods
- when Ruby sees an array method in an expression it calls the method to perform an operation that uses the array, and the method returns a value that is plugged in to the expression
- most array methods are called by giving the name of an array, a period, and the name of the operation, but a few, such as `<<`, are called when they appear as operators in an expression.

**Tutorial Project**

7. Use IRB to create an array containing a few even numbers:

```ruby
 >> a = [2, 4, 6, 8]
 => [2, 4, 6, 8]
```

8. Use the `length` operation to find out how many items are in the array:

```ruby
 >> a.length
 => 4
```

9. Add a new number to the end of the array:

```ruby
 >> a << 10
 => [2, 4, 6, 8, 10]
```
10. An empty array is an array that has 0 items. It’s analogous to the empty set in math. A common operation is to create an initially empty array and add items to it during the course of an algorithm, or start with an array of items and repeatedly delete items until the array is empty. Type this expression to make an empty array in Ruby:

```ruby
b = []
=> []
```

11. Use the `length` method to find out how many elements are in `b`. Did you get 0?

12. There is a method named `empty?` that tests whether an array is empty (note the question mark is part of the name of the operation). If we invoke this operation on the new array `b` Ruby tells us the array is empty:

```ruby
b.empty?
=> true
```

13. Use this method to see if `a` is empty.

14. At the beginning of this section there was a claim that arrays can hold any type of item. Let’s test this by making an array of strings:

```ruby
colors = ["green", "yellow", "white"]
```

15. Use the `<<` operator to add a new string to the end of `colors`:

```ruby
colors << "black"
```

16. Use the `length` method to figure out how many items are now in `colors`.

17. What do you think will happen if you try to add a string to the end of `a`, which is an array of numbers? Or if you try to add a number to the end of `colors`, which is an array of strings? Will Ruby complain, or will it make a mixed array? Use IRB to test your hypothesis. What did you learn?

### 3.3 Iterators

Recall from the introduction to this chapter that the basic outline for the Sieve of Eratosthenes is to first make a list of integers, and then to go through the list to discard numbers that are not primes. We’ve seen how to make a list of numbers, and now we’re ready to see how we can work through a list to perform operations on individual items.

The general technique for performing an operation on every item in a list is known as iteration. The word comes from the Latin *iter*, which means “road” or “path”. We often talk about iterating over a list, which means we start at the front of the list and walk our way through, one item at a time, in order to perform some operation. Ruby and other object-oriented languages have several methods known as iterators that apply an operation to each item in the container.

The simplest iterator for arrays is named `each`. This operation is invoked just like `length` and other methods, by giving the name of the array we want to work on, a period, and then the name of the operation. But in this case there is a twist: we need also need to specify the expression we want to evaluate on each element in the list. We do this by writing that expression in braces (curly parentheses) following the word `each`. Here is an example:

```ruby
>> a = [1,2,3]
=> [1, 2, 3]
>> a.each { |x| p x * 2 }
2
4
```
Displaying Output in the Terminal Window

Ruby has a special method named \texttt{p}, which stands for "print." It always returns the same thing: a special value named \texttt{nil}, which is Ruby's way of representing "nothing." Why call a method that always returns nothing? The answer is that we are interested in this method's \textbf{side effects}. The useful side effect of this method is that it print the values of its parameters on the terminal.

This method isn't very useful in interactive Ruby because IRB prints the results of expressions, but it's going to come in very handy when we start experimenting with more complicated expressions. At various points in an algorithm we are going to want to call \texttt{p} to print the values of selected variables so we can keep track of how the algorithm is progressing.

Here are two examples of using \texttt{p} to print values on the terminal:

\begin{verbatim}
>> p(4)
4
=> nil

>> p "Hello"
"Hello"
=> nil
\end{verbatim}

In the first example, Ruby simply uses \texttt{p} to print the number 4. The second example is similar, except \texttt{p} prints a string instead of a number. The second example also shows how \texttt{p} is most often written, without parentheses around the parameter.

Look closely at both examples and note that \texttt{p} has printed a value on the terminal -- the string that shows up on the second line of each example -- and that the third line in each example is what is printed by IRB as the result of the call to \texttt{p}. Since \texttt{p} always returns \texttt{nil} the third line in these two examples is the same.

\begin{verbatim}
6
=> [1, 2, 3]
\end{verbatim}

In this expression, Ruby selects each item from \texttt{a}, one at a time. When an item is selected it is put in a variable named \texttt{x}, and then Ruby evaluates the expression \texttt{p x * 2} (see the sidebar on "Displaying Output" for an explanation of how \texttt{p} works). Since \texttt{p} is the method that prints a value on the terminal, evaluating this expression causes Ruby to print the value of \texttt{x * 2}. There are three items in \texttt{a}, so the expression \texttt{p x * 2} is evaluated three times, once for each item in \texttt{a}. As a result three lines are printed on the terminal, each showing the result of a call to \texttt{p x * 2}.

A useful mnemonic for the notation used with the \texttt{each} iterator is to think of how mathematicians specify the members of a set. For example, in math, if we want to describe the set of numbers between 1 and 10, we might write

\[
\{ x \mid 1 \leq x \leq 10 \}
\]

When read aloud, this expression is "the set of all \textit{x} such that \textit{x} is greater than or equal to 1"
and less than or equal to 10.” The syntax of the Ruby expression is very similar – inside the
braces we give the name of a variable between a pair of | symbols and an expression that
contains this variable. The role of the iterator is to assign the variable to each item in the
list, one after the other, and for each item Ruby evaluates the expression to the right of the
| symbols.

In computer science iteration goes beyond the idea of walking through the elements in
a list. The more general notion of iteration is that of repeating an operation or group of
operations, i.e. iteration is often synonymous with repetition. An algorithm might require a
set of steps to be repeated a fixed number of times, or until some condition is met, and the
algorithm might need to do this regardless of whether there is a list to iterate over.

As an example of an iteration that repeats an operation a specified number of times we
can use an iterator named (appropriately enough) times. In Ruby n.times { ... } means
“evaluate the expression between the braces n times”. For example, this expression causes
Ruby to print the word “hello” three times:

```ruby
>> 3.times { |x| p "hello" }
"hello"
"hello"
"hello"
=> 3
```

In this example the variable x was not used by the expression between the braces. The next
element shows that for a call of the form n.times the variable is set to the values between
0 and n-1. We can make a list containing the numbers from 0 to 9 by first creating an empty
list and then using the times iterator to repeatedly add a value to the end of the list:

```ruby
>> a = []
=> []
>> 10.times { |x| a << x }
=> 10
>> a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

An iterator named delete_if will be very useful for the Sieve of Eratosthenes projects. As
its name implies, it is used to delete items from a list. This iterator puts an item from
the list into the iterator variable and evaluate the expression; if that expression evaluates to
true the item is removed from the list. The tutorial exercises in this section will have some
examples of how to use the delete_if iterator.

**Tutorial Project**

18. Before we experiment with the delete_if iterator let’s review expressions that return true
or false (called Boolean expressions). Note how Ruby evaluates expressions involving com-
parison operators:

```ruby
>> 3 < 4
=> true
>> 3 > 4
=> false
```

19. To see if two values are equal, use the == operator. Note that there are two =’s in this
eexpression! If you use just one = Ruby will think you want to do an assignment, but in this
case you don’t, you want to make a comparison:
20. One way to see if a number is even is to ask Ruby to check the remainder after dividing by 2 – an even number has a remainder of zero:

```ruby
>> 3 % 2
=> 1
>> 3 % 2 == 0
=> false
>> 4 % 2
=> 0
>> 4 % 2 == 0
=> true
```

21. Make a new list of the numbers from 0 to 9:

```ruby
>> a = []
=> []
>> 10.times { |x| a << x }
=> 10
```

22. Use the `each` iterator to print true or false depending on whether an item in `a` is even or not:

```ruby
>> a.each { |n| p n % 2 == 0 }
true
false
true
...
```

23. We can turn the list of integers into a list of odd integers by deleting each even number. To do this, just call the `delete_if` method and it will delete every item for which the expression evaluates to `true`:

```ruby
>> a.delete_if { |n| n % 2 == 0 }
=> [1, 3, 5, 7, 9]
```

24. How many items are in `a` after you called `delete_if`?

25. What is the first item in `a`? The last?

### 3.4 Exploring the Algorithm

We now have all the pieces we need to put together a program that will generate prime numbers using the Sieve of Eratosthenes: we know something about how Ruby operates on numbers, both integers and reals; we know how to make lists of numbers; and we know how to iterate over the lists. In this section we’ll use those pieces to build a list of primes between 2 and 100. We’ll be using IRB as a “scratchpad” – we’ll be making a list pretty much the same way we did it by hand in the introduction to this chapter, but instead of writing a list of numbers on a piece of paper and crossing them off we’ll let Ruby take care of all the clerical work. We’ll create Ruby arrays to keep track of the numbers, and iterate over the arrays to remove composite (non-prime) numbers. In the next section we’ll see how to collect all these pieces into a single working program so Ruby can do everything, not only the clerical work with the number lists but also the decision-making about which numbers to cross off the list and which to add to the growing list of prime numbers.

The first thing to do is to make the initial lists. We want a list of all integers between 2 and 100, and a second list to hold the prime numbers. This second list will initially be empty. Before we make the lists, we need to decide on names for them. The examples of how to
create and use arrays all used simple names like a. That's fine for these examples – the focus is on the array operations, and choosing a more meaningful name might be a distraction. But now we want to create arrays that are going to be used in several different steps, and choosing names like a and b could lead to trouble later on. Better to choose names that make it clear what each array is going to be used for. A good choice is worklist for the array that holds values we iterate over, and primes for the array of numbers that gets new prime numbers as they are discovered.

Here are the main steps in the algorithm presented in the first section, rewritten using terminology introduced since then and with the names of the two arrays we'll use:

1. copy the first number in worklist to primes
2. iterate over worklist to remove every number that is a multiple of the first number in the list
3. if all the numbers in worklist are prime we can stop, otherwise go back to step 1

The second step is the key operation, and it brings up an important detail about how we need to initialize the working list. If we're looking for every prime between 1 and 100, we might be tempted to set worklist to the array [1, 2, ... 100]. But if we do, the iteration will remove every element in worklist because every number is a multiple of 1! So what we want to do is put 99 numbers in the list, ranging in value from 2 to 100.

Our first attempt is:

```ruby
> worklist = []
=> []
> 99.times { |i| worklist << i }
=> 99
```

Now let's ask Ruby to print the array to verify it has what we want:

```ruby
> worklist
=> [0, 1, 2, 3, ... 98]
```

Oops. We have 99 numbers, but they range from 0 to 98 instead of 2 to 100. But we can fix this easily enough by telling Ruby to attach i+2 to the end of the list instead of i. We need to reset worklist to the empty list and retype the times iteration, and then we have what we want:

```ruby
> worklist = []
=> []
> 99.times { |i| worklist << i+2 }
=> 99
> worklist
=> [2, 3, 4, ... 99, 100]
```

Perfect. Creating the initial empty list of primes is trivial:

```ruby
> primes = []
=> []
```
Now we're ready to try the first step of the algorithm. Recall that to get the first item in an array we can call the method named `first`, so to copy the first item from `worklist` to `primes` just type

```ruby
>> primes << worklist.first
=> [2]
```

Note that this step does not remove the 2 from the front of the work list; all it does is make a copy of the first item. We can check the contents of the arrays after this step:

```ruby
>> primes
=> [2]
>> worklist
=> [2, 3, 4, ... 99, 100]
```

Now comes the key step. What we want to do is figure out how to use the `delete_if` method to remove multiples of the most recent prime from `worklist`. Recall from the exercises in the previous section that we can tell if a number `x` is even (i.e. if it's a multiple of 2) by seeing if `x % 2` is 0. We can use the same idea here – to see if a number is a multiple of any value `n` check to see if `x % n` is 0. The following expression will scan the entire working list to remove every value that is a multiple of the last item in `primes` (found by calling `primes.last`):

```ruby
>> worklist.delete_if { |x| x % primes.last == 0 }
=> [3, 5, 7, 9, ... 97, 99]
```

Note that every even number has been deleted from `worklist` because when this expression was evaluated `primes.last` was 2.

Since we're looking for primes from 2 to 100 we aren't done yet – we have to keep filtering numbers out of `worklist` until only prime numbers are left. If we go back and retype the expression that copies the first item in `worklist` to `primes` and the expression that filters the multiples of that item this is what we'll see:

```ruby
>> primes << worklist.first
=> [2, 3]
>> worklist.delete_if { |x| x % primes.last == 0 }
=> [5, 7, 11, 13, ... 95, 97]
```

If we repeat these steps three more times we'll end up with a complete set of prime numbers, with the first five in `primes` and the rest still in `worklist`. Verify this claim by asking Ruby to print the current values of each array:

```ruby
>> primes
=> [2, 3, 5, 7, 11]
>> worklist
=> [13, 17, 19, 23, ... 89, 97]
```

**Tutorial Project**

Go back to the beginning of this section and type all of the expressions in your IRB environment. As you evaluate the expressions make sure you understand how each of the two main steps works: how the first item in the working list is copied to the end of the list of primes, and how the `delete_if` iterator removes multiples of the most recent prime. You can repeat these two steps either by cut-and-paste, or, if your IRB environment supports it, by hitting the uparrow key on your keyboard until you see the expression you want to evaluate again.
3.5 The **sieve** Method

The main goal for this chapter was to use Interactive Ruby to gain some insight into the Sieve of Eratosthenes algorithm. The project in the previous section accomplished this, and after doing the exercises in that section you should have a fairly good understanding of how each cycle of the algorithm removes all the multiples of the most recently discovered prime number. There is only one thing left to do in order to put these operations into a new method named `sieve`: we have to figure out how to automate these steps, so that when the method is called Ruby repeats the two key operations until only prime numbers are left in the working list.

The first step in making the `sieve` method is to create the file named `sieve.rb` and enter the outline of the method:

```ruby
def sieve(n)
  # code goes here
end
```

This definition tells Ruby that we are making a new method named `sieve`, and that when the method is called we will be passing a parameter value. We will be able to use this value inside the body of the method, e.g. to make the initial `worklist` we will use the expression `n.times` to make a list of numbers from 2 to `n`.

Next we can enter the lines that create the two arrays and an expression that will return the final result. At this point we don't really have a list of prime numbers, but don't worry, we'll be adding more statements later. This is what the method looks like at this point:

```ruby
def sieve(n)
  worklist = []
  n.times { |i| worklist << i+2 }

  primes = []
  return primes
end
```

A good habit to get into is to save this version and load it into IRB. It won't do much if you call it:

```bash
>> load "sieve.rb"
=> true
>> sieve(50)
=> []
```

Can you see why the result of the call to `sieve` is the empty list? Even though we don't have a fully working method yet, this exercise of saving and then loading a partially completed version was very helpful: it showed us that the outline of the method is correct, and that we haven't made a silly mistake like leaving off the end or typing in the wrong name for the method.

When we used IRB as a workbench to explore how to use the `delete_if` method to filter composite numbers from `worklist` we simply repeated the execution of two expressions until it was “obvious” there were no more composite numbers left in `worklist`. It may be obvious to us, but not to the computer – we need to figure out how to specify a terminating condition that we can write as an expression in Ruby.
A straightforward method is to just let Ruby keep transferring numbers from worklist to primes until worklist is empty. Ruby will do too much work, but let’s go ahead and implement this version anyway, because it is a good way to introduce a new technique for describing iteration.

Earlier in this chapter we saw two other forms of iteration – one, implemented by the each method, was used to iterate over an array, and the other, implemented by the times method, was used to repeat an operation a fixed number of times. What we want for this project is a form of iteration that repeats a set of operations until a certain condition is met. In Ruby, this type of iteration is written as a while expression:

```ruby
while worklist.length > 0
  primes << worklist.first
  worklist.delete_if { |x| x % primes.last == 0 }
end
```

What these lines tell Ruby to do is to evaluate the boolean expression next to the word while (while is another keyword). If the value of this expression is true, Ruby evaluates the expressions on the following lines, up to the line that has the word end. Ruby then goes back to the line with while and checks the boolean expression again. In this example, as long as there are items in worklist Ruby will keep copying the first item from worklist to the end of primes and sifting multiples of that value from worklist.

An old term to describe iteration, dating from the first programming languages, is loop; the Ruby code above is an example of a “while loop.” An important fact about every while loop is that eventually the expression that controls the loop must evaluate to false, otherwise the program never terminates – it is caught in an infinite loop.

So does the while loop in our sieve method terminate? The answer is “yes,” because every iteration removes at least one number from worklist, and eventually the list shrinks down to the empty list. To see why, remember what happened the first time you typed in an expression with the delete_if iterator:

```ruby
>> worklist
=> [2, 3, 4, ..., 99, 100]
>> primes << worklist.first
=> [2]
>> worklist.delete_if { |x| x % primes.last == 0 }
=> [3, 5, 7, 9, ..., 97, 99]
```

The call to delete_if removes the first item from worklist because every number is its own multiple, i.e. \( x \mod x = 0 \) for any integer \( x \).

To verify that this while expression works, go back to the file and insert the four-line while loop just before the line that has the return statement (the complete method should now look like the definition in Figure 3.3). Go back and reload the file and test the method:

```ruby
>> load "sieve.rb"
=> true
>> sieve(30)
=> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Success! We have a method that creates a list of prime numbers! At this point you might want to try a few more calls, perhaps getting a list of primes up to 100 or even 1,000. But
def sieve(n)
    worklist = []
    (n-1).times { |i| worklist << i+2 }
    primes = []
    while worklist.length > 0
        primes << worklist.first
        worklist.delete_if { |x| x % primes.last == 0 }
    end
    return primes
end

Figure 3.3: The first version of the `sieve` method, using a while loop that tests for an empty list.

don't get too carried away – if you ask for a list of primes up to 10,000 you'll start to see Ruby taking a lot longer to build the list. In the next section we'll improve the method so it requires far fewer cycles around the iteration in the while expression.

**Tutorial Project**

If you have not already done so, create a file named `sieve.rb` and type in the method shown in Figure 3.3. Load it into IRB and test the method by calling it a few times. Once the method is producing lists of prime numbers you are ready to move on to the next section, but if you are still not sure how it works consider doing some of the exercises shown below.

26. One way to track the progress of an algorithm is to insert print statements into the code. Ruby has an operation it calls “interpolation” that makes it easy to construct helpful strings to print in the middle of a method. If, when you are making a string, you enclose an expression between curly braces somewhere inside the string, Ruby will replace the name of the variable with its value. Here is an example:

```ruby
>> n = 5
=> 5
>> p "the value of n is #{n}"
  "the value of n is 5"
```

Notice how Ruby replaced the `#{n}` with the value 5.

27. Add this line immediately before the while loop:

```ruby
p "making a list of prime numbers from 2 to #{n}"
```

Reload the method and call it again. You should see a message from Ruby telling you that the `sieve` method has been called.

28. Add this line at the very end of your while loop, so that the while loop has three lines:

```ruby
p worklist
```

Reload the method and run it again. Can you see how Ruby prints the current value of `worklist` on each iteration, and that `worklist` shrinks each time the body of the loop is executed?

**3.6 A Better Sieve**

If you took the challenge set out in the first section, of trying to devise a formula for the largest number that you need to filter out of the working list, you should have come up
with \( m = \sqrt{n} \). What that means for our algorithm is that as soon as we move a number \( m \) that is greater than \( \sqrt{n} \) from the front of the working list to the end of \( \text{primes} \) we can stop scanning. To see why, consider a useful fact about multiplication. Any number that will be removed from the list is a composite number, which means it is the product of two smaller numbers \( a \) and \( b \). At least one of these numbers must be smaller than \( \sqrt{n} \) because the product of two numbers that are both greater than or equal to \( \sqrt{n} \) will be greater than \( n \). In the case of \( n = 20 \) the algorithm can terminate after finding that 5 is prime because \( 5 > \sqrt{20} \approx 4.42 \).

Ruby has a square root function. It’s name is \( \text{sqrt} \), but before we use it we have to tell Ruby we want to use its math library:

```
>> sqrt(20)
NoMethodError: undefined method ‘sqrt’ for main:Object
>> include Math
=> Object
>> sqrt(20)
=> 4.47213595499958
```

Go back to your text editor, and add the line with the `include` statement to the front of your file, before the definition of the `sieve` method. It’s easy to change the while statement so it checks to see if the last value added is greater than \( \sqrt{n} \), but unfortunately this change introduces a few “bugs” into our method. Let’s track them down, one at a time. First, change the line with the while expression so it looks like this:

```
while primes.last < sqrt(n)
```

This new terminating condition for the loop ensures the loop will terminate: every time the loop body is executed a new number is added to the end of \( \text{primes} \), and the new number is always larger than the previous number, so eventually a number greater than \( \sqrt{n} \) will be added to \( \text{primes} \).

Load your program and call `sieve(50)`. If you see an error message that says something about the method \( \text{sqrt} \) being undefined it means you forgot to add the include statement for the Math library. But if you got this message you have encountered the first bug we need to fix:

```
NoMethodError: undefined method ‘<’ for nil:NilClass
```

To understand this message a programmer needs to do a bit of detective work. Here is what happened: when Ruby evaluated the expression in the while statement, it called `primes.last`. But the primes array is empty, so `primes.last` returned `nil`, the special symbol Ruby uses to mean “nothing.” Ruby then tried to compare nothing with the square root of \( n \), and that’s when it complained, saying it didn’t know how to compare `nil` to anything.

One simple way to fix this problem is to make sure the \( \text{primes} \) list is never empty. We can just initialize it to \([1]\) (we can consider 1 to be a prime number for the moment). Change the line that says

```
primes = []
```

to

```
primes = [1]
```
3.6 A Better Sieve

```ruby
include Math

def sieve(n)
  worklist = []
  (n-1).times { |i| worklist << i+2 }
  primes = [1]
  while primes.last < sqrt(n)
    primes << worklist.first
    worklist.delete_if { |x| x % primes.last == 0 }
  end
  primes.shift
  return primes + worklist
end
```

Figure 3.4: The complete definition of the `sieve` method, which uses the Sieve of Eratosthenes algorithm to generate a list of prime numbers between 2 and an arbitrary upper limit `n`.

```ruby
primes = [1]
```

The second “bug” is that we’re missing a lot of numbers all of a sudden. If you make the change shown above and reload your program and run it again by calling `sieve(100)` you’ll set a list of prime numbers, but they will only go up to 11. Previously, after we removed the last set of multiples, the prime numbers were spread across the two lists – all the primes up to and including the first number greater than \(\sqrt{n}\) were in the `primes` array and all the remaining primes were still in `worklist`. To fix this new bug we have to figure out how to splice together the `primes` list and the working list. In Ruby it’s easy to splice together two lists. If `a` and `b` are arrays, the expression `a + b` creates a new array that has all the items in `a` followed by all the items in `b`. So the final expression in our new method is simply

```ruby
return primes + worklist
```

There is just one final detail to attend to. If you don’t like the fact that the new list includes the number 1 at the front, you can remove it before the line that has the return statement. A method named `shift` deletes the first item in an array, so adding a call to `primes.shift` just before the last statement will result in a list that is just like the one we got with the first version of the method.

After you make these three changes (initializing `primes` to `[1]`, calling the `shift` method, and modifying the return value) your program should look like the one in Figure 3.4.

**Tutorial Project**

After you make the changes to your method described in this section you can try the following optional project. You might be surprised to find how much of an effect this new loop termination condition has on the efficiency of the method: to make a list of prime numbers between 2 and 100,000 requires only 66 iterations. Without this new terminating condition – i.e. if the loop ran until the worklist was empty – the method would make almost 10,000 iterations. Our new version is over 100 times more efficient!

29. To count how many times a loop is executed is simple: just create a variable with an initial value of 0, and then add 1 to the variable in the middle of the loop. Add the following line
to your method, just after the line that initializes the primes array:

```ruby
count = 0
```

30. Add this line to the end of the while loop, just before the line with the word end:

```ruby
count = count + 1
```

This expression may seem strange, but remember how Ruby carries out assignments: it evaluates the expression on the right side, using the current value of count, and then simply stores the new value back in count, overwriting the old value.

31. Add this line just before the line with the return expression:

```ruby
puts "count = #{count}"
```

puts stands for “put string.” It’s like p, except it doesn’t include the quotes around the string, so it produces nicer output when we are printing strings.

32. Save your file, reload the method, and call it to make a list of primes up to 100:

```ruby
>> sieve(100)
  count = 5
```

33. Notice what happened in the last example: you asked Ruby to call your method, passing it the value 100 as the parameter. The method built the list of primes, then printed the value of count, and then returned the list. Ruby printed the list as the value of the call to sieve. Does this count seem accurate? One way to find out: look at the fifth item in the list. Is it greater than √100?

34. If you want to try a more ambitious experiment, call sieve again to make a list from 2 to 100,000, but this time save the result in a variable named p:

```ruby
>> p = sieve(100000)
  count = 66
  => [2, 3, 5, 7, 11, ... 99989, 99991]
```

You might have to scroll pretty far back in your terminal emulator window to find the line that tells you the iteration count was 66.

35. To see if this count is accurate, we can ask Ruby for the 66th item in the list p. To fetch a value from an array, write the location of the item between square brackets. To get the first item (computer scientists like to start counting at 0):

```ruby
>> p[0]
  => 2
```

To get the second item:

```ruby
>> p[1]
  => 3
```

To see the 66th item:

```ruby
>> p[65]
  => 317
```

Is this number correct?

36. As a final challenge, consider the fact that to make a list of prime numbers between 2 and 100,000 we should have to filter out multiples of numbers between 2 and √100,000, which is 316. At first glance it might seem our new improved algorithm should have to do 316 iterations. But we just saw that our method makes only 66 trips around the loop. Can you explain why it does not have to make a full 316 iterations?