Exercises (cont.)

4. Write a function perfect3 that computes all 3-perfect numbers. (A 3-perfect number is such that the sum of its divisors equals three times the number. For example, 120 is 3-perfect since

\[
\text{In}[1]:= \text{Apply}[\text{Plus}, \text{Divisors}[120]]
\]
\[
\text{Out}[1]= 360
\]

which is of course, 3(120). Find the only other 3-perfect number under 1000.

5. Write a function perfect4 and find the only 4-perfect number less than 2,200,000.

6. Write a function perfectK that accepts as input a number \( k \), and two numbers \( a \) and \( b \), and computes all \( k \)-perfect numbers in the range from \( a \) to \( b \). For example, perfectK[1, 30, 2] would compute all 2-perfect numbers in the range from 1 to 30 and hence, would output \( \{6, 28\} \).

7. If \( \sigma(n) \) is defined to be the sum of the divisors of \( n \), then a number \( n \) is called superperfect if \( \sigma(\sigma(n)) = 2n \). Write a function superperfect[a, b] that finds all superperfect numbers in the range from \( a \) to \( b \).

4.5 Anonymous Functions

An anonymous function is a function that does not have a name and that can be used on the spot, at the moment it is created. This is often convenient, especially if the function is only going to be used once or as an argument to a higher-order function, such as Map, Fold, or Nest. The built-in function Function is used to create an anonymous function.

The basic form of an anonymous function is Function[x, body] for an anonymous function with a single variable \( x \) (note that any symbol can be used for the variable), and Function[{x, y, ...}, body] for an anonymous function with more than one variable. The body looks like the rhs of a user-defined function definition, with the variables \( x, y, \ldots \), where argument names would be.
As an example, the first user-defined function we created:

\[ \text{In}[1]:= \text{square}[x_] := x^2 \]

can be written as an anonymous function:

\[ \text{In}[2]:= \text{Function}[z, z^2] \]
\[ \text{Out}[2]= \text{Function}[2, 2] \]

There is also a standard input form that can be used in writing an anonymous function which is easier to write and read than the Function notation. In this form, the rhs of the function definition is placed inside parentheses which are followed by the ampersand symbol (&), and the argument name is replaced by the pound symbol (#). If there is more than one variable, #1, #2, and so on are used. Using this notation, the square function:

\[ \text{square}[x_] := x^2 \]

becomes

\[ (#^2)& \]

An anonymous function is applied in the usual way, by following the function with the argument values enclosed in square brackets. For example,

\[ \text{In}[3]:= (#^2)&[6] \]
\[ \text{Out}[3]= 36 \]

We can, if we wish, give an anonymous function a name and then use the name to call the function later. This has the same effect as defining the function in the normal way. For example,

\[ \text{In}[4]:= \text{squared} = (#^2)&; \]
\[ \text{In}[5]:= \text{squared}[6] \]
\[ \text{Out}[5]= 36 \]

The best way to become comfortable with anonymous functions is to see them in action, so we will convert some of the functions we defined earlier into anonymous functions (we'll show both the \( \ldots \# \ldots \)& and the Function forms so that you can decide which you prefer to use):

- The function that tests whether all the elements of a list are even:
  \[ \text{areEltsEven}[\text{lis}_] := \text{Apply}[\text{And}, \text{Map}[\text{EvenQ}, \text{lis}]] \]
becomes
(Apply[And, Map[EvenQ, #]]) &

or
Function[x, Apply[And, Map[EvenQ, x]]]

- The function that returns each element in the list greater than all previous elements:
  maxima[x_] := Union[Rest[FoldList[Max, 0, x]]]

  becomes
  (Union[Rest[FoldList[Max, 0, #]]]) &

  or
  Function[y, Union[Rest[FoldList[Max, 0, y]]]]

- The function that removes a randomly-chosen element from a list:
  removeRand[lis_] := Delete[lis, Random[Integer, {1, Length[lis]}]]

  becomes
  (Delete[#, Random[Integer, {1, Length[##]}]]) &

  or
  Function[x, Delete[x, Random[Integer, {1, Length[x]}]]]

We can also create nested anonymous functions. For example:

In[8] := (Map[(#^2)&, #]) & [3, 2, 7]
Out[8] := {9, 32, 49}

When dealing with nested anonymous functions, the shorthand notation can be used for each of the anonymous functions but care needs to be taken to avoid confusion as to which variable belongs to which anonymous function. This can be avoided by using Function, in which case different variables names can be used.

In[9] := Function[y, Map[Function[x, x^2], y]] [3, 2, 7]
Out[9] := {9, 32, 49}

We can use the anonymous function version of the removeRand function in the deal function definition.
In[8] := Clear[deal]

In[9] := deal[n_] := Module[{carddeck},
   carddeck = Flatten[{Outer[List, {c, d, h, s},
      Join[Range[2, 10], {j, q, k}], 1];
   Complement[carddeck, Nest[(Delete[#, Random[Integer, {1, Length[##]})] &,
      carddeck, n)]}
   ]
   ]

We see that this works fine:

Out[10]= {{c, j}, {h, 5}, {h, 8}, {h, 10}, {s, k}}

Finally, we can generalize the deal function to work with any list

In[11] := chooseWithoutReplacement[lis_, n_] :=
   Complement[lis, Nest[(Delete[#, Random[Integer, {1, Length[##]})] &,
      lis, n)]
   ]

Notice that it is not necessary to use the Module function in the above
definition because the only quantities on the rhs of the function definition are
anonymous functions, built-in functions and the names of the arguments of the
function. Functions that have this form are called one-liners.

Exercises

1. Write a function to sum the squares of the elements of a numeric list.

2. Write a function to sum the digits of any integer. You will need the
   IntegerDigits function (use ?IntegerDigits, or look in the Function Browser
   or the manual [Wol91] to find out about this function).

3. Write a function setOfDistances[l] that, given a list l of points in the plane,
   finds the set of all distances between the points. So for example, given a
   list of 5 points in the plane

   In[10] := points = Table[{Random[], Random[]}, {5}]
   Out[10]= {{0.780739, 0.807045}, {0.541554, 0.512377},
      {0.226454, 0.563833}, {0.797958, 0.261694},
      {0.27265, 0.669689}}
Exercises (cont.)

setOfDistances will return the \( \binom{5}{2} = 10 \) distances between the 5 points (omitting of course the 0-distance from any point to itself):

\[
\text{In[2]} := \text{setOfDistances[points]}
\]

\[
\text{Out[2]} = \{0.379543, 0.605334, 0.545621, 0.52146, 0.319283, \\
0.355591, 0.322219, 0.646454, 0.134257, 0.677717\}
\]

Consider adapting the code from Exercise 3 on page 79 by making it an anonymous function inside your setOfDistances function.

For an interesting discussion of the many facts concerning the set of distances determined by \( n \) points in the plane, see Chapter 12 of [Hon76].

4. Write an anonymous function that moves a random walker from one location on a square lattice to one of the four adjoining locations with equal probability. For example, given argument \{6, 2\}, the function should return either \{6, 3\}, \{8, 1\}, \{7, 2\} or \{5, 2\} with equal likelihood. Now, use this anonymous function with NestList to generate the list of step locations for an \( n \)-step random walk.

4.6 One-Liners

In the simplest version of a user-defined function, there are no value declarations or auxiliary function definitions; the rhs is a single nested function call whose arguments are the names of the arguments on the lhs, without the blanks. What we want to show now is how to construct one-liners from scratch.

4.6.1 The Josephus Problem

Flavius Josephus was a Jewish historian during the Roman-Jewish war of the first century. Through his writings comes the following story:

The Romans had chased a group of 10 Jews into a cave and were about to attack. Rather than die at the hands of their enemy, the group chose to commit suicide one by one. Legend has it though, that they decided to go around their circle of 10 individuals and eliminate every other person until no-one was left. Who was the last to survive?

Although a bit macabre, this problem has a definite mathematical component and lends itself well to a functional style of programming. We'll start by
changing the problem a bit (the importance of rewording a problem can hardly
be overstated; the key to most problem solving resides in turning something we
can’t work with into something we can work with). We’ll restate the problem as
follows: $n$ people are lined up. The first person is moved to the end of the line,
the second person is removed from the line, the third person is moved to the
end of the line, the fourth person is ... and so on until only one person remains
in the line.

The statement of the problem indicates that there is a repetitive action,
performed over and over again. It involves the use of the RotateLeft function
(move the person at the front of the line to the back of the line) followed by the
use of the Rest function (remove the next person from the line). We can write
an anonymous function to make this nested function call:

$$(\text{Rest}(\text{RotateLeft}[\#]))\&$$

At this point it is already pretty clear where this computation is headed.
We want to take a list and using the Nest function, perform the anonymous
function call $(\text{Rest}(\text{RotateLeft}[\#]))\&$ on the list until only one element remains.
A list of $n$ elements will need $n - 1$ calls. So we can now write the function, to
which we give the apt name survivor:

$$\text{survivor}[\text{lis}_0] :=$$
$$\text{Nest}[(\text{Rest}(\text{RotateLeft}[\#]))\&, \text{lis}, \text{Length}[\text{lis}] - 1]$$

Trying out the survivor function on a list of 10:

$$\text{survivor}[	ext{Range}[10]]$$

we see that the fifth position will be the position of the survivor.

Tracing the applications of RotateLeft in this example gives a very clear
picture of what is going on. The following form of TracePrint shows only
the results of the applications of RotateLeft that occur during evaluation of the
expression survivor[Range[10]]:
In[3] := TracePrint[survivor[Range[10]], RotateLeft]
    RotateLeft
    {2, 3, 4, 5, 6, 7, 8, 9, 10, 1}
    RotateLeft
    {4, 5, 6, 7, 8, 9, 10, 1, 3}
    RotateLeft
    {6, 7, 8, 9, 10, 1, 3, 5}
    RotateLeft
    {8, 9, 10, 1, 3, 5, 7}
    RotateLeft
    {10, 1, 3, 5, 7, 9}
    RotateLeft
    {3, 5, 7, 9, 1}
    RotateLeft
    {7, 9, 1, 5}
    RotateLeft
    {1, 5, 9}
    RotateLeft
    {9, 5}

Out[3] = {5}

4.6.2 | Pocket Change

As another example, we will write a program to perform an operation most
of us do every day. We will calculate how much change we have in our pocket.
As an example, suppose we have the following collection of coins:

In[1] := coins = {p, p, q, n, d, p, q, q, p}

where p, n, d, and q represent pennies, nickels, dimes, and quarters, respectively. Let's start by using the Count function to determine the number of pennies we have:

In[2] := Count[coins, p]
Out[2] = 4

This works. So let's do the same thing for all of the coin types:

In[3] := {Count[coins, p],
        Count[coins, n],
        Count[coins, d],
        Count[coins, q]}
Out[3] = {4, 1, 2, 3}
Looking at this list, it's apparent that there ought to be a more compact way of writing the list. If we Map an anonymous function involving Count and coins onto the list \{p, n, d, q\}, it should do the job.

\[
\text{In}[4] := \text{Map}[(\text{Count}[\text{coins}, \#])&, \{p, n, d, q\}]
\]
\[
\text{Out}[4] = \{4, 1, 2, 3\}
\]

Now that we know how many coins of each type we have, we want to calculate how much change we have. We first do the calculation manually to see what we get for an answer (so we'll know when our program works).

\[
\text{In}[5] := (4 \ 1) + (1 \ 5) + (2 \ 10) + (3 \ 25)
\]
\[
\text{Out}[5] = 104
\]

From the above computation we see that the lists \{4, 1, 2, 3\} and \{1, 5, 10, 25\} are first multiplied together element-wise and then the elements of the result are added. This suggests a couple of possibilities:

\[
\text{In}[6] := \text{Apply}[(\text{Plus}, \{4, 1, 2, 3\} \ast \{1, 5, 10, 25\})]
\]
\[
\text{Out}[6] = 104
\]

\[
\text{In}[7] := \{4, 1, 2, 3\} \ast \{1, 5, 10, 25\}
\]
\[
\text{Out}[7] = 104
\]

These two operations are both suitable for the job (to coin a phrase, 'there's not a penny, nickel, quarter, or dime's worth of difference'). We'll write the one-liner using the first method:

\[
\text{In}[8] := \text{pocketChange}[\_] :=
\]
\[
\text{Apply}[(\text{Plus}, \text{Map}[(\text{Count}[(\_), \#])&, \{p, n, d, q\}]) \{1, 5, 10, 25\}]
\]

\[
\text{In}[9] := \text{pocketChange}[\text{coins}]
\]
\[
\text{Out}[9] = 104
\]

**Exercises**

One of the best ways to learn how to write programs is to practice reading code. We list below a number of one-liner function definitions along with a very brief explanation of what these user-defined functions do and a typical input and output. The reader should deconstruct these programs to see what they do and then re-construct them as compound functions without any anonymous functions.
Exercises (cont.)

1. Determine the frequencies with which the distinct elements in a list appear in the list.

   \[ \text{in1} := \text{frequencies} \{ \text{lis}_1 \} := \text{Map}[(\#, \text{Count}[\text{lis}, \#]) \&, \text{Union}[\text{lis}]] \]

   \[ \text{in2} := \text{frequencies} \{\{a, a, b, b, a, c, c\}\} \]
   \[ \text{Out2} = \{(a, 3), (b, 3), (c, 2)\} \]

2. Divvy up a list into parts:

   \[ \text{in1} := \text{split1} \{\text{lis}, \text{parts}\} := \]
   \[ \quad \text{Inner}[\text{Take}[\text{lis}, \text{\#1}, \text{\#2}]] \&,
   \quad \text{Drop}[\text{\#1}, -1] + 1,
   \quad \text{Rest}[\text{\#1}],
   \quad \text{List}[(\text{FoldList}[\text{Plus}, 0, \text{parts}])] \]

   \[ \text{in2} := \text{split1} \{\text{Range}[10], \{2, 5, 0\}\} \]
   \[ \text{Out2} = \{(1, 2), (3, 4, 5, 6, 7), (), (8, 9, 10)\} \]

3. This is the same as the previous program, done in a different way.

   \[ \text{in1} := \text{split2} \{\text{lis}, \text{parts}\} := \]
   \[ \quad \text{Map}[(\text{Take}[\text{lis}, \# + 1, 0])] \&,
   \quad \text{Partition}[\text{FoldList}[\text{Plus}, 0, \text{parts}], 2, 1]] \]

4. Another game in the ILLinois State Lottery is based on choosing \( n \) numbers, each between 0 and \( s \) with no duplicates allowed. Write a user-defined function called \( \text{lotto} \) (after the official lottery names of Little Lotto and Big Lotto) to perform sampling without replacement on an arbitrary list. (Note: the difference between this function and the function \( \text{chooseWithoutReplacement} \), is that the order of selection is needed here).

   \[ \text{in1} := \text{lotto1} \{\text{lis}, \text{n}\} := \]
   \[ \quad \text{Flatten}[\text{Rest}[\text{MapThread}[\text{Complement},
   \quad \{\text{RotateRight}[\#], \#\}, 1]]]] \&
   \quad \text{NestList}[\text{Delete}[\#,
   \quad \text{Random}[\text{Integer}, \{1, \text{Length}[\#]\}]]) \&,
   \quad \text{lis}, \text{n}] \]

   \[ \text{in2} := \text{lotto1} \{\text{Range}[10], 5\} \]
   \[ \text{Out2} = \{3, 8, 6, 5, 4\} \]