Programming with Mathematica
An Introduction

Solutions to exercises

Solutions to the exercises in *Programming with Mathematica: An Introduction* are given here. The exercises to every section are listed first, followed by the solutions. Solutions are provided both as a PDF file and in notebook form at www.cambridge.org/wellin.

2
The *Mathematica* language

2.1 *Expressions*

1. Give the full (internal) form of the expression `a (b + c)`.
2. What is the traditional representation of `Times[a, Power[Plus[b, c], -1]]`.
3. What is the part specification of the `b` in the expression `a x^2 + b x + c`?
4. What do you expect to be the result of the following operations? Use the `FullForm` of the expressions to understand what is going on.
   a. `((x^2 + y) z / w)[[2, 1, 2]]`.
   b. `(a / b)[[2, 2]]`.

2.1 *Solutions*

1. The expression `a (b + c)` is given in full form as `Times[a, Plus[b, c]]`.
2. This is simply $\frac{a}{b+c}$ as can be seen by evaluating the full form expression.

```
In[1]:= Times[a, Power[Plus[b, c], -1]]

           a
Out[1]= -----
         b + c
```

3. There are three elements in the expression, with the term `b x` being the second.

```
In[2]:= expr = a x^2 + b x + c;

In[3]:= FullForm[expr]

Out[3]//FullForm= Plus[c, Times[b, x], Times[a, Power[x, 2]]]
```

The first element of `Times[b, x]` is `b`, so the part specification is `2, 1`.

```
In[4]:= expr[[2]]

Out[4]= b x

In[5]:= expr[[2, 1]]

Out[5]= b
```

4. Looking at the internal representation of this expression with `FullForm` helps to unwind the part specification.

```
In[6]:= FullForm[(x^2 + y) z / w]

Out[6]//FullForm= Times[Power[w, -1], Plus[Power[x, 2], y], z]

In[7]:= (x^2 + y) z / w [[2, 1, 2]]

Out[7]= 2
```

From the `FullForm` of `a / b`, you can see that the second part is `Power[b, -1]` and the second part of that is `-1`. Note the need for parentheses here as the `Part` function has higher precedence than `Power`. For more information on operator precedence, see Operator Input Forms (WMDC).

```
In[8]:= FullForm[a / b]

Out[8]//FullForm= Times[a, Power[b, -1]]

In[9]:= (a / b)[[2, 2]]

Out[9]= -1
```

2.2 *Definitions*

1. What rules are created by each of the following functions? Check your predictions by evaluating them and then querying *Mathematica* with `?`*function_name*.

a. `randLis1[n_] := RandomReal[1, {n}]`

b. `randLis2[n_] := (x = RandomReal[]; Table[x, {n}])`

c. `randLis3[n_] := (x := RandomReal[]; Table[x, {n}])`

d. `randLis4[n_] = Table[RandomReal[], {n}]`

2. Consider two functions `f` and `g`, which are identical except that one is written using an immediate assignment and the other using a delayed assignment.

```
In[1]:=  f[n_] = Sum[(1 + x)^j, {j, 1, n}];

In[2]:=  g[n_] := Sum[(1 + x)^j, {j, 1, n}]
```

Explain why the outputs of these two functions *look* so different. Are they in fact different?

```
In[3]:=  f[2]
```

Out[3]= $\dfrac{(1+x)\left(-1+(1+x)^2\right)}{x}$

```
In[4]:=  g[2]
```

Out[4]= $1 + x + (1 + x)^2$

3. Write rules for a function `log` (note lowercase) that encapsulate the following identities:

$$\log(a\,b) = \log(a) + \log(b);$$
$$\log\left(\tfrac{a}{b}\right) = \log(a) - \log(b);$$
$$\log(a^n) = n\log(a).$$

4. Create a piecewise-defined function $g(x)$ based on the following and then plot the function from -2 to 0.

$$g(x) = \begin{cases} -\sqrt{1-(x+2)^2} & -2 \le x \le -1 \\ \sqrt{1-x^2} & x < 0 \end{cases}$$

2.2 *Solutions*

1. This exercise focuses on the difference between immediate and delayed assignments.

   a. This will generate a list of *n* random numbers.

```
In[1]:=  randLis1[n_] := RandomReal[1, {n}]

In[2]:=  randLis1[3]

Out[2]= {0.988991, 0.213663, 0.475922}
```

   b. Since the definition for `x` is an immediate assignment, its value does not change in the body of `randLis2`. But each time `randLis2` is called, a new value is assigned to `x`.

```
In[3]:=  randLis2[n_] := (x = RandomReal[]; Table[x, {n}])

In[4]:=  randLis2[3]

Out[4]= {0.759267, 0.759267, 0.759267}
```

```
In[5]:=  randLis2[3]

Out[5]=  {0.979382, 0.979382, 0.979382}
```

c. Because the definition for `x` is a delayed assignment, the definition for `randLis3` is functionally equivalent to `randLis1`.

```
In[6]:=  randLis3[n_] := (x := RandomReal[]; Table[x, {n}])

In[7]:=  randLis3[3]

Out[7]=  {0.7941, 0.736074, 0.254351}
```

d. In an immediate assignment, the right-hand side of the definition is evaluated first. But in this case, `n` does not have a value, so `Table` is not able to evaluate properly.

```
In[8]:=  randLis4[n_] = Table[RandomReal[], {n}]

   Table::iterb : Iterator {n} does not have appropriate bounds. >>

Out[8]=  Table[RandomReal[], {n}]

In[9]:=  Clear[x]
```

2. The definition for `f` given in the exercise evaluates the sum first (immediate assignment), giving a symbolic expression for the general sum from 1 to *n*. When `f[2]` is evaluated, the argument 2 is then substituted into this expression for *n*. In the case of `g`, the value of *n* is substituted and then the sum is evaluated. Although the resulting expressions output by these two functions look different at first, expanding them gives the same result.

```
In[10]:=  f[n_] = Sum[(1 + x)^j, {j, 1, n}]
```

$$\text{Out[10]=}\quad \frac{(1+x)\,(-1+(1+x)^n)}{x}$$

```
In[11]:=  g[n_] := Sum[(1 + x)^j, {j, 1, n}]

In[12]:=  Expand[f[2]]
```

Out[12]= $2 + 3\,x + x^2$

```
In[13]:=  Expand[g[2]]
```

Out[13]= $2 + 3\,x + x^2$

3. The rules for the logarithm function are as follows. Note, there is no need to program the division rule separately. Do you see why? (Look at `FullForm[x / y]`.)

```
In[14]:=  log[a_ * b_] := log[a] + log[b]
```

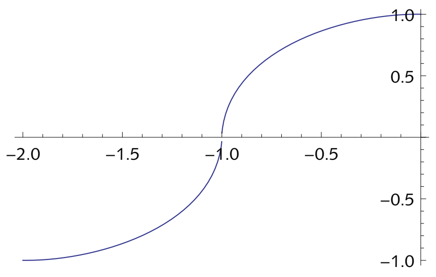
In[15]:= `log[`$a\_^{n\_}$`] := n log[a]`

```
In[16]:= log[x y^2 z^3]

Out[16]= log[x] + 2 log[y] + 3 log[z]

In[17]:= log[x / y]

Out[17]= log[x] - log[y]
```

4. Using `Piecewise`, we have:

```
In[18]:= g[x_] := Piecewise[{{-1 Sqrt[1 - (x + 2)^2], -2 ≤ x ≤ -1}, {Sqrt[1 - x^2], x < 0}}]

In[19]:= Plot[g[x], {x, -2, 0}]
```

Out[19]=



2.3 *Predicates and Boolean operations*

1. Create a predicate function that returns a value of `True` if its argument is between –1 and 1.
2. Define a predicate function `CharacterQ[`*str*`]` that returns true if its argument *str* is a single string character, and returns false otherwise.
3. Write a predicate function `NaturalQ[`*n*`]` that returns a value of `True` if *n* is a natural number and `False` otherwise, that is, `NaturalQ[`*n*`]` is `True` if *n* is among 0, 1, 2, 3, ….
4. Create a predicate function `SubsetQ[`lis_1`, `lis_2`]` that returns a value of `True` if lis_1 is a subset of lis_2. Remember, the empty set, `{}`, is a subset of every set.
5. Create a predicate function `CompositeQ` that tests whether its argument is a nonprime integer.

2.3 *Solutions*

1. There are several ways to define this function, either using the relational operator for less than, or with the absolute value function.

```
In[1]:= f[x_] := -1 < x < 1

In[2]:= f[x_] := Abs[x] < 1

In[3]:= f[4]

Out[3]= False
```

```
In[4]:=  f[-0.35]

Out[4]=  True
```

2. The requirements here are that the argument be both a string (`StringQ`) and have length (`StringLength`) one.

```
In[5]:=  CharacterQ[ch_] := StringQ[ch] && StringLength[ch] == 1

In[6]:=  CharacterQ["v"]

Out[6]=  True

In[7]:=  CharacterQ["vi"]

Out[7]=  False

In[8]:=  CharacterQ[vi]

Out[8]=  False
```

3. A number *n* can be considered a natural number if it is both an integer and greater than or equal to zero. There is some disagreement in the mathematics community about 0, but for our purposes, we will adopt the convention that 0 is a natural number.

```
In[9]:=  NaturalQ[n_] := IntegerQ[n] && n ≥ 0

In[10]:=  NaturalQ[0]

Out[10]=  True

In[11]:=  NaturalQ[-4]

Out[11]=  False
```

4. The empty set is a subset of every set. So first we need a definition to cover this case.

```
In[12]:=  SubsetQ[{}, lis2_] := True
```

The intersection of `lis1` and `lis2` will be identical to `lis1` whenever `lis1` is a subset of `lis2`.

```
In[13]:=  SubsetQ[lis1_, lis2_] := Intersection[lis1, lis2] == lis1

In[14]:=  A = {a, b, c};
          B = {a, b, c, d, e};

In[16]:=  SubsetQ[A, B]

Out[16]=  True
```

We can also give a definition in terms of the subset character ⊂ which can be entered by typing ESC-sub-ESC or by using one of the palettes.

```
In[17]:=  lis1_ ⊂ lis2_ := Intersection[lis1, lis2] == lis1
```

```
In[18]:=  A ⊂ B

Out[18]=  True
```

5. There are three tests that have to be satisfied: integer, greater than 1, not prime.

```
In[19]:=  CompositeQ[n_] := IntegerQ[n] && n > 1 && Not[PrimeQ[n]]

In[20]:=  CompositeQ[2^31 - 1]

Out[20]=  False

In[21]:=  CompositeQ[2^31 + 1]

Out[21]=  True
```

This is more neatly done using conditional pattern matching. See, for example, Section 4.1 on patterns.

2.4 *Attributes*

1. Ordinarily, when you define a function, it has no attributes. *Mathematica* evaluates the arguments before passing them up to the calling function. So, in the following case, `2 + 3` is evaluated before it is passed to `f`.

```
In[1]:=  f[x_ + y_] := x^2 + y^2

In[2]:=  f[2 + 3]

Out[2]=  f[5]
```

Use one of the `Hold` attributes to give `f` the property that its argument is not evaluated first. The resulting output should look like this:

```
In[3]:=  f[2 + 3]

Out[3]=  13
```

2. Define a function that takes each number in a vector of numbers and returns that number if it is within a certain interval, say $-0.5 < x < 0.5$, and returns $\sqrt{x}$ otherwise. Then make your function listable so that it can operate on vectors (lists) directly.

2.4 *Solutions*

1. First clear any definitions and attributes that might be associated with `f`.

```
In[1]:=  ClearAll[f]
```

Then set the `HoldAll` attribute to prevent initial evaluation of the argument of this function.

```
In[2]:=  SetAttributes[f, HoldAll]

In[3]:=  f[x_ + y_] := x^2 + y^2
```

```
In[4]:=  f[a + b]

Out[4]=  a^2 + b^2

In[5]:=  f[2 + 3]

Out[5]=  13
```

2. Here is a small list of random numbers to use.

```
In[6]:=  vec = RandomReal[{-1, 1}, 10]

Out[6]=  {-0.0606576, -0.245491, 0.612118, 0.904241, -0.598884,
          0.546108, 0.145979, -0.491207, 0.327699, 0.810952}
```

The function could be set up to take two arguments, the number and the bound.

```
In[7]:=  fun[x_?NumberQ, bound_] := If[-bound < x < bound, x, √x]
```

Make `fun` listable.

```
In[8]:=  SetAttributes[fun, Listable]

In[9]:=  fun[vec, 0.5]

Out[9]=  {-0.0606576, -0.245491, 0.78238, 0.950916, 0. + 0.773876 ⅈ,
          0.738991, 0.145979, -0.491207, 0.327699, 0.900529}
```

3
Lists

3.1 *Creating and displaying lists*

1. Generate the list `{{0}, {0, 2}, {0, 2, 4}, {0, 2, 4, 6}, {0, 2, 4, 6, 8}}` in two different ways using the `Table` function.
2. A table containing ten random 1s and 0s can be created using `RandomInteger[1, {10}]`. Create a ten-element list of random 1s, 0s and −1s.
3. Create a ten-element list of random 1s and −1s. This list can be viewed as the steps taken in a random walk along the *x*-axis, where a step can be taken in either the positive *x* direction (corresponding to 1) or the negative *x* direction (corresponding to −1) with equal likelihood.

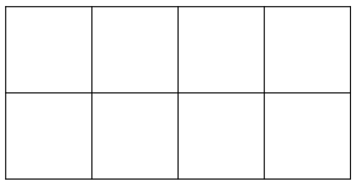
   The random walk in one, two, three (and even higher) dimensions is used in science and engineering to represent phenomena that are probabilistic in nature. We will use a variety of random walk models throughout this book to illustrate many different programming concepts.
4. Generate both of the following arrays using the `Table` function.

```
In[1]:= Array[f, 5]

Out[1]= {f[1], f[2], f[3], f[4], f[5]}

In[2]:= Array[f, {3, 4}]

Out[2]= {{f[1, 1], f[1, 2], f[1, 3], f[1, 4]},
          {f[2, 1], f[2, 2], f[2, 3], f[2, 4]},
          {f[3, 1], f[3, 2], f[3, 3], f[3, 4]}}
```

5. Construct an integer lattice graphic like the one below. Start by creating pairs of coordinate points to connect with lines – here we have written the coordinates explicitly but you should generate them programmatically. Once you have your coordinate pairs, you can display the graphic as follows:

```
In[3]:= coords = {{{-2, -1}, {2, -1}}, {{-2, 0}, {2, 0}},
           {{-2, 1}, {2, 1}}, {{-2, -1}, {-2, 1}}, {{-1, -1}, {-1, 1}},
           {{0, -1}, {0, 1}}, {{1, -1}, {1, 1}}, {{2, -1}, {2, 1}}};
        Graphics[Line[coords]]
```

Out[4]=



6. Import six images, resize them to the same dimensions, then display them inside a 3 × 2 grid using options for `Grid` to format the output.

3.1 *Solutions*

1. You can take every other element in the iterator list, or encode that in the expression `2 j`.

```
In[1]:=  Table[j, {i, 0, 8, 2}, {j, 0, i, 2}]

Out[1]=  {{0}, {0, 2}, {0, 2, 4}, {0, 2, 4, 6}, {0, 2, 4, 6, 8}}

In[2]:=  Table[2 j, {i, 0, 4}, {j, 0, i}]

Out[2]=  {{0}, {0, 2}, {0, 2, 4}, {0, 2, 4, 6}, {0, 2, 4, 6, 8}}
```

2. Here is probably the simplest way to generate random -1s, 0s, and 1s.

```
In[3]:=  RandomInteger[{-1, 1}, {10}]

Out[3]=  {-1, -1, -1, 1, 0, 1, 1, 0, -1, -1}
```

Or use `RandomChoice`:

```
In[4]:=  RandomChoice[{-1, 0, 1}, {10}]

Out[4]=  {0, 1, -1, 0, 1, 0, 1, -1, 1, 0}
```

3. Here are three ways to generate the list.

```
In[5]:=  2 RandomInteger[1, {10}] - 1

Out[5]=  {-1, 1, 1, -1, -1, 1, -1, -1, -1, -1}
```

In[6]:= $(-1)^{\texttt{RandomInteger[1,\{10\}]}}$

```
Out[6]=  {1, 1, 1, 1, 1, -1, 1, 1, -1, -1}
```

The most direct way to do this is to use `RandomChoice`.

```
In[7]:=  RandomChoice[{-1, 1}, {10}]

Out[7]=  {-1, 1, 1, 1, -1, 1, 1, -1, -1, -1}
```

4. These lists can be generated with `Table`, using two iterators for the second example.

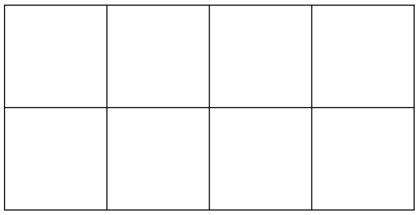
```
In[8]:=  Table[f[i], {i, 5}]

Out[8]=  {f[i], f[i], f[i], f[i], f[i]}

In[9]:=  Table[f[i, j], {i, 3}, {j, 4}]

Out[9]=  {{f[i, j], f[i, j], f[i, j], f[i, j]},
          {f[i, j], f[i, j], f[i, j], f[i, j]},
          {f[i, j], f[i, j], f[i, j], f[i, j]}}
```

5. Some thought is needed to get the iterators right using `Table`.

```
In[10]:= xmin = -2; xmax = 2; ymin = -1; ymax = 1;
         hlines = Table[{{xmin, y}, {xmax, y}}, {y, ymin, ymax}]

Out[11]= {{{-2, -1}, {2, -1}}, {{-2, 0}, {2, 0}}, {{-2, 1}, {2, 1}}}

In[12]:= vlines = Table[{{x, ymin}, {x, ymax}}, {x, xmin, xmax}]

Out[12]= {{{-2, -1}, {-2, 1}}, {{-1, -1}, {-1, 1}},
          {{0, -1}, {0, 1}}, {{1, -1}, {1, 1}}, {{2, -1}, {2, 1}}}
```

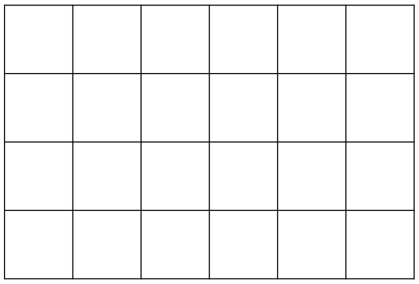
Join the two sets of lines and then flatten to remove one set of braces.

```
In[13]:= pairs = Flatten[{hlines, vlines}, 1]

Out[13]= {{{-2, -1}, {2, -1}}, {{-2, 0}, {2, 0}},
          {{-2, 1}, {2, 1}}, {{-2, -1}, {-2, 1}}, {{-1, -1}, {-1, 1}},
          {{0, -1}, {0, 1}}, {{1, -1}, {1, 1}}, {{2, -1}, {2, 1}}}

In[14]:= Graphics[Line[pairs]]

Out[14]=
```



Here is a function that puts all this together:

```
In[15]:= Lattice[{xmin_, xmax_}, {ymin_, ymax_}] :=
          Module[{hlines, vlines, coords},
           hlines = Table[{{xmin, y}, {xmax, y}}, {y, ymin, ymax}];
           vlines = Table[{{x, ymin}, {x, ymax}}, {x, xmin, xmax}];
           coords = Flatten[{hlines, vlines}, 1];
           Graphics[Line[coords]]]

In[16]:= Lattice[{-3, 3}, {-2, 2}]

Out[16]=
```



6. Here are some sample images to work with.

```
In[17]:= RandomSample[ExampleData["TestImage"], 6]

Out[17]= {{TestImage, Peppers}, {TestImage, Clock}, {TestImage, Airplane},
          {TestImage, Numbers}, {TestImage, Lena}, {TestImage, Ruler}}

In[18]:= images = Map[ExampleData,
            {{"TestImage", "Girl2"}, {"TestImage", "Peppers"},
             {"TestImage", "Aerial"}, {"TestImage", "Moon"},
             {"TestImage", "Tank2"}, {"TestImage", "Ruler"}}];
```

Get their dimensions.

```
In[19]:= Map[ImageDimensions, images]

Out[19]= {{256, 256}, {512, 512}, {256, 256},
          {256, 256}, {512, 512}, {512, 512}}
```

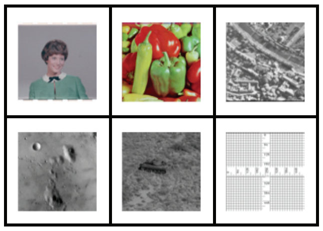
Resize the larger images.

```
In[20]:= img1 = images[[1]];
         img2 = ImageResize[images[[2]], 256];
         img3 = images[[3]];
         img4 = images[[4]];
         img5 = ImageResize[images[[5]], 256];
         img6 = ImageResize[images[[6]], 256];
```

Finally, put in a grid with some formatting.

```
In[26]:= Grid[{
           {img1, img2, img3},
           {img4, img5, img6}
          }, Frame → All, Spacings → {1, 1}, ItemSize → {3, 3}]
```

Out[26]=



3.2 *The structure of lists*

1. Given a list of integers such as the following, count the number of 0s. Find a way to count all those elements of the list which are not 1s.

```
In[1]:= ints = RandomInteger[{-5, 5}, 30]

Out[1]= {-2, -2, 2, -1, -1, -3, -5, 3, -4, -4, -3, 4, -3,
         4, 2, -2, -3, 1, 2, 3, -2, -4, 1, -1, 1, 1, 5, -2, 0, 3}
```

2. Given the list `{{{1, a}, {2, b}, {3, c}}, {{4, d}, {5, e}, {6, f}}}`, determine its dimensions. Use the `Dimensions` function to check your answer.
3. Find the positions of the 9s in the following list. Confirm using `Position`.

```
{{2, 1, 10}, {9, 5, 7}, {2, 10, 4}, {10, 1, 9}, {6, 1, 6}}
```

3.2 *Solutions*

1. Here is the list of integers to use.

```
In[1]:= ints = RandomInteger[{-5, 5}, 30]

Out[1]= {-3, 0, -4, 5, -5, -1, -5, 0, -3, 5, 4, -3, -5, 1,
         -4, 4, 3, 2, 3, -2, 5, 4, -1, 0, 2, -3, -4, -1, -1, -3}
```

Count all elements that match 0.

```
In[2]:= Count[ints, 0]

Out[2]= 3
```

Count all integers in `ints` that do not match 1.

```
In[3]:= Count[ints, Except[1]]

Out[3]= 29
```

2. From the top level, there are two lists, each consisting of three sublists, each sublist consisting of two elements.

```
In[4]:= Dimensions[{{{1, a}, {2, b}, {3, c}}, {{4, d}, {5, e}, {6, f}}}]

Out[4]= {2, 3, 2}
```

3. The `Position` function tells us that the 9s are located in the second sublist, first position, and in the fourth sublist, third position.

```
In[5]:= Position[{{2, 1, 10}, {9, 5, 7}, {2, 10, 4}, {10, 1, 9}, {6, 1, 6}}, 9]

Out[5]= {{2, 1}, {4, 3}}
```

3.3 *Operations on lists*

1. Given a list of data points, $\{\{x_1, y_1\}, \{x_2, y_2\}, \{x_3, y_3\}, \{x_4, y_4\}, \{x_5, y_5\}\}$, separate the *x* and *y* components to get:

$$\{\{x_1, x_2, x_3, x_4, x_5\}, \{y_1, y_2, y_3, y_4, y_5\}\}$$

2. Consider a two-dimensional random walk on a square lattice. (A square lattice can be envisioned as a two-dimensional grid, just like the lines on graph paper.) Each step can be in one of the four directions: `{1, 0}`, `{0, 1}`, `{-1, 0}`, `{0, -1}`, corresponding to steps in the compass directions east, north, west and south, respectively. Use the list `{{1, 0}, {0, 1}, {-1, 0}, {0, -1}}` to create a list of the steps of a ten-step random walk.

3. Extract elements in the even-numbered locations in the list `{a, b, c, d, e, f, g}`.

4. Given a matrix, use list component assignment to swap any two rows.

5. Create a function `AddColumn[`*mat*`, `*col*`, `*pos*`]` that inserts a column vector *col* into the matrix *mat* at the column position given by *pos*. For example:

```
In[1]:= mat = RandomInteger[9, {4, 4}];
        MatrixForm[mat]
Out[2]//MatrixForm=
         5  0  9  1
         0  0  0  5
         9  6  2  5
         1  2  2  2

In[3]:= AddColumn[mat, {a, b, c, d}, 3] // MatrixForm
Out[3]//MatrixForm=
         5  0  a  9  1
         0  0  b  0  5
         9  6  c  2  5
         1  2  d  2  2
```

6. Suppose you are given a list `S` of length *n*, and a list `P` containing *n* different numbers between 1 and *n*, that is, `P` is a permutation of `Range[`*n*`]`. Compute the list `T` such that for all *k* between 1 and *n*, `T⟦k⟧ = S⟦P⟦k⟧⟧`. For example, if `S = {a, b, c, d}` and `P = {3, 2, 4, 1}`, then `T = {c, b, d, a}`.

7. Given the lists `S` and `P` in the previous exercise, compute the list `U` such that for all *k* between 1 and *n*, `U⟦P⟦k⟧⟧ = S⟦k⟧`, that is, `S⟦i⟧` takes the value from position `P⟦`*i*`⟧` in `U`. Thus, for `S = {a, b, c, d}` and `P = {3, 2, 4, 1}`, `U = {d, b, a, c}`. Think of it as moving `S⟦1⟧` to position `P⟦1⟧`, `S⟦2⟧` to position `P⟦2⟧`, and so on. Hint: start by pairing the elements of `P` with the elements of `S`.

8. How would you perform the same task as `Prepend[{`*x*`, `*y*`}, `*z*`]` using the `Join` function?

9. Starting with the lists `{1, 2, 3, 4}` and `{a, b, c, d}`, create the list `{2, 4, b, d}`.

10. Starting with the lists `{1, 2, 3, 4}` and `{a, b, c, d}`, create the list `{1, a, 2, b, 3, c, 4, d}`.

11. Given two lists, find all those elements that are not common to the two lists. For example, starting with the lists, `{a, b, c, d}` and `{a, b, e, f}`, your answer would return the list `{c, d, e, f}`.

12. One of the tasks in computational linguistics involves statistical analysis of text using what are called *n*-grams. These are sequences of *n* adjacent letters or words and their frequency distribution in a body of text can be used to predict word usage based on the previous history or usage. Import a file consisting of some text and find the twenty most frequently occurring word combinations. Pairs of words that are grouped like this are called *bigrams*, that is, *n*-grams for $n = 2$.

    Use the following `StringSplit` code to split long strings into a list of words that can then be operated on with the list manipulation functions. Regular expressions are discussed in detail in Section 9.4.

```
In[4]:= words =
         StringSplit["Use StringSplit to split long strings into words.",
          RegularExpression["\\W+"]]

Out[4]= {Use, StringSplit, to, split, long, strings, into, words}
```

13. Based on the previous exercise, create a function `NGrams[`*str*`,` *n*`]` that takes a string of text and returns a list of *n*-grams, that is a list of the *n* adjacent words. For example:

```
In[5]:= text = "Use StringSplit to split long strings into words.";
        NGrams[text, 3]

Out[6]= {{Use, StringSplit, to}, {StringSplit, to, split},
         {to, split, long}, {split, long, strings},
         {long, strings, into}, {strings, into, words}}
```

3.3 *Solutions*

1. This is a straightforward use of the `Transpose` function.

```
In[1]:= Transpose[{{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}, {x5, y5}}]

Out[1]= {{x1, x2, x3, x4, x5}, {y1, y2, y3, y4, y5}}
```

2. Here is one way to do it. First create a list representing the directions.

```
In[2]:= NSEW = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
```

`RandomChoice` chooses with replacement.

```
In[3]:= RandomChoice[NSEW, {10}]

Out[3]= {{0, -1}, {0, -1}, {0, 1}, {1, 0},
         {0, 1}, {0, 1}, {0, 1}, {0, 1}, {1, 0}, {-1, 0}}
```

3. Start by dropping the first element in the list, then create a nested list of every other element in the remaining list, and finally unnest the resulting list.

```
In[4]:= Rest[{a, b, c, d, e, f, g}]

Out[4]= {b, c, d, e, f, g}

In[5]:= Partition[%, 1, 2]

Out[5]= {{b}, {d}, {f}}

In[6]:= Flatten[%]

Out[6]= {b, d, f}
```

This can also be done directly in one step using `Part` with `Span`. The expression `2 ;; -1 ;; 2` indicates the range from the second element to the last element in increments of 2.

```
In[7]:= Part[{a, b, c, d, e, f, g}, 2 ;; -1 ;; 2]

Out[7]= {b, d, f}
```

4. The standard procedural approach is to use a temporary variable to do the swapping.

```
In[8]:= mat = RandomInteger[9, {4, 4}];
        MatrixForm[mat]
Out[9]//MatrixForm=
        ( 1  8  6  9 )
        ( 4  8  4  1 )
        ( 2  9  7  5 )
        ( 6  3  0  7 )

In[10]:= temp = mat[[1]];
         mat[[1]] = mat[[2]];
         mat[[2]] = temp;
         MatrixForm[mat]
Out[13]//MatrixForm=
        ( 4  8  4  1 )
        ( 1  8  6  9 )
        ( 2  9  7  5 )
        ( 6  3  0  7 )
```

But you can use parallel assignments to avoid the temporary variable.

```
In[14]:= mat = RandomInteger[9, {4, 4}];
         MatrixForm[mat]
Out[15]//MatrixForm=
        ( 2  9  8  7 )
        ( 8  8  8  4 )
        ( 2  4  7  7 )
        ( 7  4  7  2 )

In[16]:= {mat[[2]], mat[[1]]} = {mat[[1]], mat[[2]]};
         MatrixForm[mat]
Out[17]//MatrixForm=
        ( 8  8  8  4 )
        ( 2  9  8  7 )
        ( 2  4  7  7 )
        ( 7  4  7  2 )
```

In fact you can make this a bit more compact.

```
In[18]:= mat = RandomInteger[9, {4, 4}];
         MatrixForm[mat]
Out[19]//MatrixForm=
        ( 5  7  2  6 )
        ( 2  0  3  6 )
        ( 4  2  4  8 )
        ( 5  8  2  0 )
```

```
In[20]:= mat[[{2, 1}]] = mat[[{1, 2}]];
         MatrixForm[mat]
Out[21]//MatrixForm=
         ( 2  0  3  6 )
         ( 5  7  2  6 )
         ( 4  2  4  8 )
         ( 5  8  2  0 )
```

A key point to notice is that in this exercise, the matrix `mat` was overwritten in each case; in other words, these were destructive operations. Section 5.5 discusses how to handle row and column swapping properly so that the original matrix remains untouched.

5. You need to first transpose the matrix to operate on the columns as rows.

```
In[22]:= mat = RandomInteger[9, {4, 4}];
         MatrixForm[mat]
Out[23]//MatrixForm=
         ( 7  0  8  8 )
         ( 8  3  3  8 )
         ( 3  0  8  4 )
         ( 7  5  0  6 )

In[24]:= Transpose[mat]

Out[24]= {{7, 8, 3, 7}, {0, 3, 0, 5}, {8, 3, 8, 0}, {8, 8, 4, 6}}
```

Now insert the column vector at the desired position. Then transpose back.

```
In[25]:= Insert[Transpose[mat], {a, b, c, d}, 3] // MatrixForm
Out[25]//MatrixForm=
         ( 7  8  3  7 )
         ( 0  3  0  5 )
         ( a  b  c  d )
         ( 8  3  8  0 )
         ( 8  8  4  6 )

In[26]:= Transpose@Insert[Transpose[mat], {a, b, c, d}, 3] // MatrixForm
Out[26]//MatrixForm=
         ( 7  0  a  8  8 )
         ( 8  3  b  3  8 )
         ( 3  0  c  8  4 )
         ( 7  5  d  0  6 )
```

Here then is the function, with some basic argument checking to make sure the number of elements in the column vector is the same as the number of rows of the matrix.

```
In[27]:= AddColumn[mat_, col_, pos_] /; Length[col] == Length[mat] :=
          Transpose[Insert[Transpose[mat], col, pos]]
```

6. We want those elements in `s` given by the positions in `P`.

```
In[28]:= S = {a, b, c, d};
         P = {3, 2, 4, 1};

In[30]:= S[[{3, 2, 4, 1}]]

Out[30]= {c, b, d, a}
```

Or, more compactly:

```
In[31]:= S⟦P⟧

Out[31]= {c, b, d, a}
```

7. Without resorting to a functional approach as discussed in Chapter 5, this requires several steps.

```
In[32]:= S = {a, b, c, d};
         P = {3, 2, 4, 1};

In[34]:= Transpose[{P, S}]

Out[34]= {{3, a}, {2, b}, {4, c}, {1, d}}

In[35]:= Sort[%]

Out[35]= {{1, d}, {2, b}, {3, a}, {4, c}}

In[36]:= Transpose[%]

Out[36]= {{1, 2, 3, 4}, {d, b, a, c}}

In[37]:= %⟦2⟧

Out[37]= {d, b, a, c}
```

Using `Map`, this can be done in one step.

```
In[38]:= Map[Last, Sort[Transpose[{P, S}]]]

Out[38]= {d, b, a, c}
```

8. `Join` expects lists as arguments.

```
In[39]:= Join[{z}, {x, y}]

Out[39]= {z, x, y}
```

9. Joining the two lists and then using `Part` with `Span` is the most direct way to do this.

```
In[40]:= expr = Join[{1, 2, 3, 4}, {a, b, c, d}]

Out[40]= {1, 2, 3, 4, a, b, c, d}

In[41]:= expr[[2 ;; -1 ;; 2]]

Out[41]= {2, 4, b, d}
```

10. The function `Riffle` is designed for this task.

```
In[42]:=  Riffle[{1, 2, 3, 4}, {a, b, c, d}]

Out[42]=  {1, a, 2, b, 3, c, 4, d}
```

This can also be done in two steps by first transposing the two lists and then flattening.

```
In[43]:=  Transpose[{{1, 2, 3, 4}, {a, b, c, d}}]

Out[43]=  {{1, a}, {2, b}, {3, c}, {4, d}}

In[44]:=  Flatten[%]

Out[44]=  {1, a, 2, b, 3, c, 4, d}
```

11. This is another way of asking for all those elements that are in the union but not the intersection of the two sets.

```
In[45]:=  A = {a, b, c, d};
          B = {a, b, e, f};

In[47]:=  Complement[A ⋃ B, A ⋂ B]

Out[47]=  {c, d, e, f}

In[48]:=  Complement[Union[A, B], Intersection[A, B]]

Out[48]=  {c, d, e, f}
```

12. We will use Darwin's *On the Origin of Species* text, built into *Mathematica* via `ExampleData`.

```
In[49]:=  darwin = ExampleData[{"Text", "OriginOfSpecies"}];
          words = StringSplit[darwin, RegularExpression["\\W+"]]

          A very large output was generated. Here is a sample of it:

          {INTRODUCTION, When, on, board, H, M, S, ≪151 191≫,
Out[50]=   wonderful, have, been, and, are, being, evolved}

          Show Less   Show More   Show Full Output   Set Size Limit...
```

First, partition the list of words into pairs with overlap one. Then tally them and sort by the frequency count, the last element in each sublist. Finally, take the last twenty expressions, those bigrams occurring the most frequently.

```
In[51]:= tally = SortBy[Tally[Partition[words, 2, 1]], Last];
         Take[tally, -20]

Out[52]= {{{has, been}, 190}, {{each, other}, 195},
          {{species, of}, 202}, {{of, life}, 229}, {{with, the}, 230},
          {{natural, selection}, 236}, {{it, is}, 238}, {{and, the}, 240},
          {{by, the}, 243}, {{from, the}, 256}, {{in, a}, 256},
          {{to, be}, 267}, {{of, a}, 268}, {{have, been}, 432},
          {{that, the}, 438}, {{on, the}, 501}, {{to, the}, 582},
          {{the, same}, 716}, {{in, the}, 1028}, {{of, the}, 1995}}
```

Here are the next twenty most frequently occurring bigrams.

```
In[53]:= Take[tally, -40 ;; -20]

Out[53]= {{{to, have}, 120}, {{which, are}, 121}, {{conditions, of}, 124},
          {{between, the}, 125}, {{do, not}, 128}, {{and, in}, 132},
          {{the, case}, 133}, {{can, be}, 137}, {{will, be}, 138},
          {{as, the}, 141}, {{would, be}, 142}, {{number, of}, 144},
          {{all, the}, 150}, {{at, the}, 157}, {{the, most}, 160},
          {{the, other}, 167}, {{for, the}, 171}, {{the, species}, 172},
          {{I, have}, 187}, {{may, be}, 189}, {{has, been}, 190}}
```

13. This is a straightforward extension of the previous exercise.

```
In[54]:= NGrams[text_, n_] := Partition[
           StringSplit[text, RegularExpression["\\W+"]], n, 1]

In[55]:= sentence = "Use StringSplit to split long strings into words.";
         NGrams[sentence, 3]

Out[56]= {{Use, StringSplit, to}, {StringSplit, to, split},
          {to, split, long}, {split, long, strings},
          {long, strings, into}, {strings, into, words}}
```

4 Patterns and rules

4.1 Patterns

1. Use conditional patterns to find all those numbers in a list of integers that are divisible by 2 or 3 or 5.
2. Write down five conditional patterns that match the expression `{4, {a, b}, "g"}`.
3. Write a function `Collatz` that takes an integer n as an argument and returns $3n + 1$ if n is an odd integer and returns $n/2$ if n is even.
4. Write the `Collatz` function from the above exercise, but this time you should also check that the argument to `Collatz` is positive.
5. Use alternatives to write a function `abs[x]` that returns x if $x \geq 0$, and $-x$ if $x < 0$, whenever x is an integer or a rational number. Whenever x is complex, `abs[x]` should return $\sqrt{\mathrm{re}(x)^2 + \mathrm{im}(x)^2}$.
6. Create a function `swapTwo[lis]` that returns *lis* with only its first two elements interchanged; for example, the input `swapTwo[{a, b, c, d, e}]` should return `{b, a, c, d, e}`. If *lis* has fewer than two elements, `swapTwo` just returns *lis*. Write `swapTwo` using three clauses: one for the empty list, one for one-element lists, and one for all other lists. Then write it using two clauses: one for lists of length 0 or 1 and another for all longer lists.

4.1 Solutions

1. Start by creating a list of integers with which to work.

```
In[1]:= lis = RandomInteger[1000, {20}]

Out[1]= {480, 430, 509, 848, 842, 760, 785, 769, 579,
         96, 754, 241, 840, 180, 849, 707, 347, 333, 613, 67}
```

`IntegerQ` is a predicate; it returns `True` or `False`, so we need to use the logical OR to separate clauses here.

```
In[2]:= Cases[lis, n_ /; IntegerQ[n / 2] || IntegerQ[n / 3] || IntegerQ[n / 5]]

Out[2]= {480, 430, 848, 842, 760, 785, 579, 96, 754, 840, 180, 849, 333}
```

This is a bit more compact and direct.

```
In[3]:= Cases[lis, n_ /; Mod[n, 2] == 0 || Mod[n, 3] == 0 || Mod[n, 5] == 0]

Out[3]= {480, 430, 848, 842, 760, 785, 579, 96, 754, 840, 180, 849, 333}
```

Once you are familiar with pure functions (Section 5.6), you can also do this with `Select`.

```
In[4]:= Select[lis, Mod[#, 2] == 0 || Mod[#, 3] == 0 || Mod[#, 5] == 0 &]

Out[4]= {480, 430, 848, 842, 760, 785, 579, 96, 754, 840, 180, 849, 333}
```

2. `FullForm` should help to guide you.

```
In[5]:= FullForm[{4, {a, b}, "g"}]

Out[5]//FullForm= List[4, List[a, b], "g"]

In[6]:= MatchQ[{4, {a, b}, "g"}, x_List /; Length[x] == 3]

Out[6]= True

In[7]:= MatchQ[{4, {a, b}, "g"}, {_, y_, _} /; y[[0]] == List]

Out[7]= True

In[8]:= MatchQ[{4, {a, b}, "g"}, {x_, y_, z_} /; AtomQ[z]]

Out[8]= True

In[9]:= MatchQ[{4, {a, b}, "g"}, {x_, _, _} /; EvenQ[x]]

Out[9]= True
```

3. The Collatz function has a direct implementation based on its definition. There is no need to check explicitly that the argument is an integer since `OddQ` and `EvenQ` handle that.

```
In[10]:= Collatz[n_?OddQ] := 3 n + 1
```

In[11]:= `Collatz[n_?EvenQ] :=` $\frac{n}{2}$

Here we iterate the Collatz function fifteen times starting with an initial value of 23.

```
In[12]:= NestList[Collatz, 23, 15]

Out[12]= {23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

Check for arguments that do not match the patterns above.

```
In[13]:= Collatz[24.0]

Out[13]= Collatz[24.]
```

4. Here again is the Collatz function, but this time using a condition on the right-hand side of the definition.

```
In[14]:= Clear[Collatz]

In[15]:= Collatz[n_] := 3 n + 1 /; OddQ[n] && Positive[n]
```

In[16]:= `Collatz[n_] :=` $\frac{n}{2}$ `/; EvenQ[n] && Positive[n]`

```
In[17]:= Collatz[4.3]
Out[17]= Collatz[4.3]

In[18]:= Collatz[-3]
Out[18]= Collatz[-3]

In[19]:= NestList[Collatz, 22, 15]
Out[19]= {22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

You could also put the conditions inside the pattern on the left-hand side if you prefer.

```
In[20]:= Clear[Collatz]

In[21]:= Collatz[n_ /; OddQ[n] && Positive[n]] := 3 n + 1

In[22]:= Collatz[n_ /; EvenQ[n] && Positive[n]] := n/2

In[23]:= NestList[Collatz, 22, 15]
Out[23]= {22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

5. Using alternatives, this gives the definition for real, integer, or rational arguments.

```
In[24]:= abs[x_Real | x_Integer | x_Rational] := If[x ≥ 0, x, -x]
```

Here is the definition for complex arguments.

```
In[25]:= abs[x_Complex] := Sqrt[Re[x]^2 + Im[x]^2]
```

Note that these rules are not invoked for symbolic arguments.

```
In[26]:= Map[abs, {-3, 3 + 4 I, -4/5, a}]
Out[26]= {3, 5, 4/5, abs[a]}
```

6. We first have to consider the base cases. Given a list with no elements, `swapTwo` should return the empty list. And, given a list with one element, swapping should give that one element back.

```
In[27]:= swapTwo[{}] := {}
         swapTwo[{x_}] := {x}
```

Now, we use the triple-blank to indicate that `r` could be a sequence of zero or more elements.

```
In[29]:= swapTwo[{x_, y_, r___}] := {y, x, r}

In[30]:= Map[swapTwo, {{}, {a}, {a, b, c, d}}]
Out[30]= {{}, {a}, {b, a, c, d}}
```

Notice in this second definition for `swapTwo` that the second clause covers both the situation where the argument is the empty list and when it contains only one element.

```
In[31]:= swapTwo2[{x_, y_, r___}] := {y, x, r}
         swapTwo2[x_] := x

In[33]:= Map[swapTwo2, {{}, {a}, {a, b, c, d}}]

Out[33]= {{}, {a}, {b, a, c, d}}
```

4.2 *Transformation rules*

1. Here is a rule designed to switch the order of each pair of expressions in a list. It works fine on the first example, but fails on the second.

```
In[1]:= {{a, b}, {c, d}, {e, f}} /. {x_, y_} :→ {y, x}

Out[1]= {{b, a}, {d, c}, {f, e}}

In[2]:= {{a, b}, {c, d}} /. {x_, y_} :→ {y, x}

Out[2]= {{c, d}, {a, b}}
```

Explain what has gone wrong and rewrite this rule to correct the situation, that is, so that the second example returns `{{b, a}, {d, c}}`.

2. The following compound expression returns a value of 14. Describe the evaluation sequence that was followed. Use the `Trace` function to check your answer.

```
In[3]:= z = 11;
        a = 9;
        z + 3 /.z → a

Out[5]= 14
```

Then use the `Hold` function in the compound expression to obtain a value of 12.

3. Create a function to compute the area of any triangle, given its three vertices. The area of a triangle is one-half the base times the altitude. For arbitrary points, the altitude requires a bit of computation that does not generalize. The magnitude of the cross product of two vectors gives the area of the parallelogram that they determine. The cross product is only defined for three-dimensional vectors, so to compute the area of a two-dimensional triangle using the cross product you will need to embed the edges (vectors) in three-dimensional space, say, in the plane $z = 0$. Try a second implementation using determinants instead of cross products.

4. Use pattern matching to extract all negative solutions of the following polynomial:

$$x^9 + 3.4\,x^6 - 25\,x^5 - 213\,x^4 - 477\,x^3 + 1012\,x^2 + 111\,x - 123$$

Then extract all real solutions, that is, those which are not complex.

5. Create a rewrite rule that uses a repeated replacement to "unnest" the nested lists within a list.

```
In[6]:= unNest[{{α, α, α}, {α}, {{β, β, β}, {β, β}}, {α, α}}]

Out[6]= {α, α, α, α, β, β, β, β, β, α, α}
```

6. Define a function using pattern matching and repeated replacement to sum the elements of a list.

7. Using the built-in function `ReplaceList`, write a function `cartesianProduct` that takes two lists as input and returns the Cartesian product of these lists.

```
In[7]:=  cartesianProduct[{x1, x2, x3}, {y1, y2}]

Out[7]=  {{x1, y1}, {x1, y2}, {x2, y1}, {x2, y2}, {x3, y1}, {x3, y2}}
```

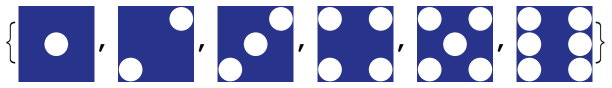
8. Write a function to count the total number of multiplications in any polynomial expression. For example, given a power, your function should return one less than the exponent.

```
In[8]:=  MultiplyCount[t^5]

Out[8]=  4

In[9]:=  MultiplyCount[a x y t]

Out[9]=  3

In[10]:= MultiplyCount[a x y t^4 + w t]

Out[10]= 7
```

9. Create six graphical objects, one each to represent the faces of a standard six-sided die. `Dice[n]` should display the face of the appropriate die, as below.

```
In[11]:= Table[Dice[n], {n, 1, 6}]
```

Out[11]=



One way to approach this problem is to think of a die face as a grid of nine elements, some of which are turned on (white) and some turned off (blue above). Then create one set of rules for each die face. Once your rules are defined, you could use something like the following graphics code (a bit incomplete as written here) to create your images.

```
Dice[n_] := GraphicsGrid[
  Map[Graphics, Partition[Range[9], 3] /. rules[[n]], {2}]]
```

4.2 *Solutions*

1. The problem here is that the pattern is too general and has been matched by the entire expression, which has the form `{x_, y_}`, where `x` is matched by `{a, b}` and `y` is matched by `{c, d}`. To fix this, use patterns to restrict the expressions that match.

```
In[1]:=  {{a, b}, {c, d}} /. {x_Symbol, y_Symbol} :→ {y, x}

Out[1]=  {{b, a}, {d, c}}

In[2]:=  {{a, b}, {c, d}, {e, f}} /. {x_Symbol, y_Symbol} :→ {y, x}

Out[2]=  {{b, a}, {d, c}, {f, e}}
```

2. The evaluation sequence can be seen directly from the `Trace` of this compound expression.

```
In[3]:=  Trace[
          z = 11;
          a = 9;
          z + 3 /. z → a
         ]

Out[3]= {z = 11; a = 9; z + 3 /. z → a,
          {z = 11, 11}, {a = 9, 9}, {{{z, 11}, 11 + 3, 14},
           {{z, 11}, {a, 9}, 11 → 9, 11 → 9}, 14 /. 11 → 9, 14}, 14}
```

First make sure that a and z have no values associated with them.

```
In[4]:=  Clear[a, z]

In[5]:=  Hold[z = 11];
         a = 9;
         z + 3 /.z → a

Out[7]=  12

In[8]:=  Clear[a]
```

3. The cross product is only defined for three dimensions, so first we need to embed the two-dimensional vectors in 3-space; in this case, in the plane $z = 0$.

```
In[9]:=  {x₁, y₁} /.{x_, y_} :→ {x, y, 0}

Out[9]=  {x₁, y₁, 0}
```

We need to compute the cross product of two vectors that span the triangle.

```
In[10]:= Cross[{x₂, y₂} - {x₁, y₁} /.{x_, y_} :→ {x, y, 0},
          {x₃, y₃} - {x₁, y₁} /.{x_, y_} :→ {x, y, 0}]

Out[10]= {0, 0, -x₂ y₁ + x₃ y₁ + x₁ y₂ - x₃ y₂ - x₁ y₃ + x₂ y₃}
```

Here are the coordinates for a triangle.

```
In[11]:= a = {0, 0};
         b = {5, 0};
         c = {3, 2};
```

And here is the computation for the cross product.

```
In[14]:= Cross[b - a /.{x_, y_} :→ {x, y, 0}, c - a /.{x_, y_} :→ {x, y, 0}]

Out[14]= {0, 0, 10}
```

So the given area is then just half the magnitude of the cross product.

```
In[15]:= Norm[%]
         -------
            2

Out[15]= 5
```

This is done more simply using determinants. Note the change here: each vector (edge of triangle) is embedded in the plane $z = 1$.

```
In[16]:= TriangleArea[tri : {v1_, v2_, v3_}] :=
          1
          - Det[tri /.{x_, y_} :> {x, y, 1}]
          2

In[17]:= TriangleArea[{a, b, c}]

Out[17]= 5

In[18]:= Clear[a, b, c]
```

4. First, get the solutions to this polynomial.

```
In[19]:= soln =
          Solve[x^9 + 3.4 x^6 - 25 x^5 - 213 x^4 - 477 x^3 + 1012 x^2 + 111 x - 123 == 0, x]

Out[19]= {{x → -2.80961}, {x → -1.85186 - 2.15082 i},
          {x → -1.85186 + 2.15082 i}, {x → -0.376453},
          {x → 0.323073}, {x → 1.06103 - 3.12709 i},
          {x → 1.06103 + 3.12709 i}, {x → 1.30533}, {x → 3.13931}}
```

The pattern needs to match an expression consisting of a list with a rule inside where the value on the right-hand side of the rule should pass the `Negative` test.

```
In[20]:= Cases[soln, {x_ → _?Negative}]

Out[20]= {{x → -2.80961}, {x → -0.376453}}
```

Here are two solutions for the noncomplex roots.

```
In[21]:= Cases[soln, {_ → _Real}]

Out[21]= {{x → -2.80961}, {x → -0.376453},
          {x → 0.323073}, {x → 1.30533}, {x → 3.13931}}

In[22]:= DeleteCases[soln, {_ → _Complex}]

Out[22]= {{x → -2.80961}, {x → -0.376453},
          {x → 0.323073}, {x → 1.30533}, {x → 3.13931}}
```

5. The transformation rule unnests lists within a list.

```
In[23]:= unNest[lis_] := Map[(# //.{x__} :> x &), lis]

In[24]:= unNest[{{a, a, a}, {a}, {{b, b, b}, {b, b}}, {a, a}}]

Out[24]= {a, a, a, a, b, b, b, b, b, a, a}
```

6. Note the need to put `y` in a list on the right-hand side of the rule. Also, an immediate rule is required here.

```
In[25]:= sumList[lis_] := First[lis //.{x_, y___} → x + {y}]

In[26]:= sumList[{1, 5, 8, 3, 9, 3}]

Out[26]= 29
```

7. The triple blank is required both before and after the variables `x` and `y`.

```
In[27]:= cartesianProduct[lis1_, lis2_] :=
          ReplaceList[{lis1, lis2}, {{___, x_, ___}, {___, y_, ___}} :> {x, y}]
```

We should also have a rule for the base case.

```
In[28]:= cartesianProduct[{}] := {}

In[29]:= Clear[x, y, z, a, b, c]

In[30]:= cartesianProduct[{a, b, c}, {x, y, z}]

Out[30]= {{a, x}, {a, y}, {a, z}, {b, x}, {b, y}, {b, z}, {c, x}, {c, y}, {c, z}}

In[31]:= cartesianProduct[{}]

Out[31]= {}
```

8. For an expression of the form `Power[a, b]`, the number of multiplies is $b - 1$.

```
In[32]:= Cases[{x^4}, Power[_, exp_] :> exp - 1]

Out[32]= {3}
```

For an expression of the form `Times[a, b, c, …]`, the number of multiplications is given by one less then the number of arguments.

```
In[33]:= Cases[{a b c d e}, fac_Times :> Length[fac] - 1]

Out[33]= {4}
```

For a mix of terms of these two cases, we will need to total up the counts from the respective terms. Here is a function that puts this all together. Use `Infinity` as a third argument to `Cases` to make sure the search goes all the way down the expression tree.

```
In[34]:= MultiplyCount[expr_?PolynomialQ] :=
          Total@Cases[{expr}, Power[_, exp_] :> exp - 1, Infinity] +
           Total@Cases[{expr}, fac_Times :> Length[fac] - 1, Infinity]
```

```
In[35]:= MultiplyCount[a b^2 c d^5]

Out[35]= 8

In[36]:= poly = Expand[(x + y - z)^3]
```

Out[36]= $x^3 + 3 x^2 y + 3 x y^2 + y^3 - 3 x^2 z - 6 x y z - 3 y^2 z + 3 x z^2 + 3 y z^2 - z^3$

```
In[37]:= MultiplyCount[poly]

Out[37]= 28
```

9. First, we create a grid of the nine locations on the die.

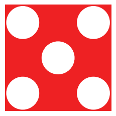
```
In[38]:= lis = Partition[Range[9], 3];
         Grid[lis]

         1  2  3
Out[39]= 4  5  6
         7  8  9
```

Next, use graphics primitives to indicate if a location on the grid is colored (on) or not (off).

```
In[40]:= off = {Red, Disk[]};
         on = {White, Disk[]};
```

Here are the rules for a five.

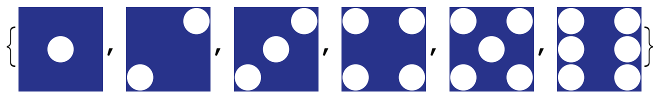
```
In[42]:= GraphicsGrid[Map[Graphics,
           lis /. {1 → on, 2 → off, 3 → on,
             4 → off, 5 → on, 6 → off, 7 → on, 8 → off, 9 → on},
           {2}], Background → Red, Spacings → 10, ImageSize → 50]

Out[42]=
```



The five other rules are straightforward. Here then is a function that wraps up the code. Note the use of the `Background` option to `GraphicsGrid` to pick up the color from the value of `off`.

```
In[43]:= Dice[n_] :=
          Module[{rules, off = {Darker@Blue, Disk[]}, on = {White, Disk[]}},
           rules = {
             {1 → off, 2 → off, 3 → off, 4 → off,
              5 → on, 6 → off, 7 → off, 8 → off, 9 → off},
             {1 → off, 2 → off, 3 → on, 4 → off, 5 → off,
              6 → off, 7 → on, 8 → off, 9 → off},
             {1 → off, 2 → off, 3 → on, 4 → off, 5 → on,
              6 → off, 7 → on, 8 → off, 9 → off},
             {1 → on, 2 → off, 3 → on, 4 → off, 5 → off,
              6 → off, 7 → on, 8 → off, 9 → on},
             {1 → on, 2 → off, 3 → on, 4 → off, 5 → on,
              6 → off, 7 → on, 8 → off, 9 → on},
             {1 → on, 2 → off, 3 → on, 4 → on, 5 → off,
              6 → on, 7 → on, 8 → off, 9 → on}
            };
           GraphicsGrid[Map[Graphics,
             Partition[Range[9], 3] /. rules[[n]],
             {2}], Background → First[off], Spacings → 10, ImageSize → 40]
          ]
```

```
In[44]:= Table[Dice[n], {n, 1, 6}]
```

Out[44]= 

4.3 *Examples and applications*

1. The function `FindSubsequence` defined in this section suffers from the limitation that the arguments `digits` and `subseq` must both be lists of numbers. Write another definition of `FindSubsequence` that takes two integers as arguments. So, for example, the following should work:

```
In[1]:= n = RandomInteger[10^200]

Out[1]= 99 886 364 225 785 890 637 248 382 678 171 952 235 146 647 070 036 321 ⋱
         273 192 078 968 865 572 610 676 045 767 583 093 169 497 891 617 017 225 ⋱
         261 830 124 007 777 401 603 464 795 137 556 513 541 607 966 794 013 354 ⋱
         513 861 062 656 302 896 471 480 157 720 676 043 512

In[2]:= FindSubsequence[n, 22]

Out[2]= {{9, 10}, {35, 36}, {105, 106}}
```

2. Plot the function $\sin(x)$ over the interval $[-2\pi, 2\pi]$ and then reverse the x- and y-coordinates of each point by means of a transformation rule to display a reflection in the line $y = x$.

3. Given a two-column array of data,

```
In[3]:= data = RandomReal[{0, 10}, {5, 2}];
        MatrixForm[data, TableAlignments → "."]
Out[4]//MatrixForm=
             2.75703  8.36575
             7.99197  4.86756
             1.90927  5.59835
             7.76051  2.29443
             3.87192  8.11463
```

create a new array that consists of three columns where the first two columns are identical to the original, but the third column consists of the norm of the two numbers from the first two columns.

```
             2.75703  8.36575  8.80835
             7.99197  4.86756  9.35761
             1.90927  5.59835  5.91497
             7.76051  2.29443  8.09258
             3.87192  8.11463  8.99105
```

4. Occasionally, when collecting data from an instrument, the collector fails or returns a bad value. In analyzing the data, the analyst has to make a decision about what to use to replace these bad values. One approach is to replace them with a column mean. Given an array of numbers such as the following, create a function to replace each `"NAN"` with the mean of the numbers that appear in that column.

$$\text{data} = \begin{pmatrix} 0.9034 & \text{"NAN"} & 0.7163 & 0.8588 \\ 0.3031 & 0.5827 & 0.2699 & 0.8063 \\ 0.0418 & 0.8426 & \text{"NAN"} & 0.8634 \\ \text{"NAN"} & 0.8913 & 0.0662 & 0.8432 \end{pmatrix};$$

4.3 *Solutions*

1. Here is the function `FindSubsequence` as given in the text.

```
In[1]:= FindSubsequence[lis_List, subseq_List] :=
         Module[{p, len = Length[subseq]},
          p = Partition[lis, len, 1];
          Position[p, subseq] /. {num_?IntegerQ} :→ {num, num + len - 1}]
```

This creates another rule associated with `FindSubsequence` that simply takes each integer argument, converts it to a list of integer digits, and then passes that off to the rule above.

```
In[2]:= FindSubsequence[n_Integer, subseq_Integer] := Module[
          {nlist = IntegerDigits[n], sublist = IntegerDigits[subseq]},
          FindSubsequence[nlist, sublist]
         ]
```

Create the list of the first 100 000 digits of π.

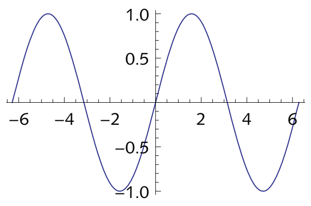
```
In[3]:= pi = FromDigits[RealDigits[N[Pi, 10^5] - 3][[1]]];
```

The subsequence 1415 occurs seven times at the following locations in this digit expansion of π.

```
In[4]:= FindSubsequence[pi, 1415]

Out[4]= {{1, 4}, {6955, 6958}, {29 136, 29 139}, {45 234, 45 237},
         {79 687, 79 690}, {85 880, 85 883}, {88 009, 88 012}}
```

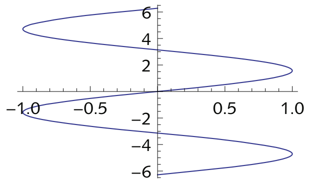
2. Here is the plot of the sine function.

```
In[5]:= splot = Plot[Sin[x], {x, -2 π, 2 π}]
```

Out[5]=


This replacement rule interchanges each ordered pair of numbers. Note the need to modify the plot range here.

```
In[6]:=  Show[splot /. {x_?NumberQ, y_?NumberQ} :→ {y, x},
          PlotRange → {-2 π, 2 π}]
```

Out[6]=


Although this particular example may have worked without the argument checking (`_?NumberQ`), it is a good idea to include it so that pairs of arbitrary expressions are not pattern matched here. We only want to interchange pairs of numbers, not pairs of options or other expressions that might be present in the underlying expression representing the graphic.

3. We are embedding the two-dimensional data into a three-dimensional array. The embedding function is written directly as a transformation rule.

```
In[7]:=  data = RandomReal[{0, 1}, {8, 2}]

Out[7]=  {{0.430685, 0.720811}, {0.258206, 0.143087},
          {0.41837, 0.180368}, {0.038807, 0.860367}, {0.759872, 0.532175},
          {0.404326, 0.769468}, {0.530604, 0.795417}, {0.345516, 0.0340149}}

In[8]:=  data /. {x_, y_} :→ {x, y, Norm[{x, y}]} // MatrixForm

Out[8]//MatrixForm=
         ⎛0.430685  0.720811   0.839678⎞
         ⎜0.258206  0.143087   0.295203⎟
         ⎜ 0.41837  0.180368   0.455594⎟
         ⎜0.038807  0.860367   0.861242⎟
         ⎜0.759872  0.532175   0.927694⎟
         ⎜0.404326  0.769468    0.86923⎟
         ⎜0.530604  0.795417   0.956153⎟
         ⎝0.345516  0.0340149  0.347186⎠
```

4. First, here is the data with which we will work.

```
                  ⎛0.9034   "NAN"   0.7163  0.8588  0.1228⎞
                  ⎜0.3031  0.5827  0.2699  0.8063   "NAN" ⎟
In[9]:=  array =  ⎜0.0418  0.8426   "NAN"   0.8634  0.9682⎟;
                  ⎜0.9163  0.8913  0.0662  0.8432  0.0547⎟
                  ⎝0.7937  0.6905  0.9105  0.5589  0.8993⎠
```

Get only the numeric values from the second column.

```
In[10]:=  col2 = array[[All, 2]];
          Cases[col2, _?NumberQ]

Out[11]=  {0.5827, 0.8426, 0.8913, 0.6905}
```

Compute the mean of the second column.

```
In[12]:=  Mean[Cases[col2, _?NumberQ]]

Out[12]=  0.751775
```

Replace the string with the column mean.

```
In[13]:=  col2 /. "NAN" → Mean[Cases[col2, _?NumberQ]] // MatrixForm

Out[13]//MatrixForm=
          ( 0.751775 )
          (  0.5827  )
          (  0.8426  )
          (  0.8913  )
          (  0.6905  )
```

Turn it into a function.

```
In[14]:=  fixcolumn[col_] := array[[All, col]] /.
            "NAN" :→ Mean[Cases[array[[All, col]], _?NumberQ]]
```

Try this function out on column 1 of our matrix.

```
In[15]:=  fixcolumn[2]

Out[15]=  {0.751775, 0.5827, 0.8426, 0.8913, 0.6905}
```

Map this function across all the columns.

```
In[16]:=  Map[fixcolumn, Range[Length[First[array]]]] // MatrixForm

Out[16]//MatrixForm=
          (  0.9034    0.3031    0.0418    0.9163  0.7937 )
          ( 0.751775   0.5827    0.8426    0.8913  0.6905 )
          (  0.7163    0.2699   0.490725   0.0662  0.9105 )
          (  0.8588    0.8063    0.8634    0.8432  0.5589 )
          (  0.1228   0.51125    0.9682    0.0547  0.8993 )
```

This operated on the columns, so the array is a list of the transformed column vectors. Transpose it back to put things right.

```
In[17]:=  MatrixForm[Transpose[%]]

Out[17]//MatrixForm=
          ( 0.9034  0.751775   0.7163    0.8588   0.1228 )
          ( 0.3031   0.5827    0.2699    0.8063  0.51125 )
          ( 0.0418   0.8426   0.490725   0.8634   0.9682 )
          ( 0.9163   0.8913    0.0662    0.8432   0.0547 )
          ( 0.7937   0.6905    0.9105    0.5589   0.8993 )
```

Next, turn this into a reusable function, `FixArray`.

```
In[18]:=  FixArray[mat_] := Module[{fixcolumn},
            fixcolumn[col_] := mat[[All, col]] /.
              "NAN" :→ Mean[Cases[mat[[All, col]], _?NumberQ]];
            Transpose[Map[fixcolumn, Range[Length[First[mat]]]]]]
```

```
In[19]:= FixArray[array] // MatrixForm
Out[19]//MatrixForm=
         ⎛0.9034  0.751775   0.7163    0.8588   0.1228 ⎞
         ⎜0.3031   0.5827    0.2699    0.8063  0.51125 ⎟
         ⎜0.0418   0.8426   0.490725   0.8634   0.9682 ⎟
         ⎜0.9163   0.8913    0.0662    0.8432   0.0547 ⎟
         ⎝0.7937   0.6905    0.9105    0.5589   0.8993 ⎠
```

5
Functional programming

5.2 *Functions for manipulating expressions*

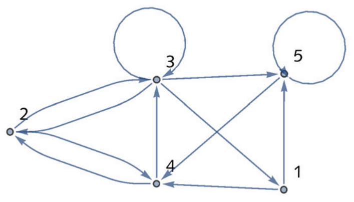
1. Rewrite the definition of `SquareMatrixQ` given in Section 4.1 to use `Apply`.
2. Given a set of points in the plane (or 3-space), find the maximum distance between any pair of these points. This is often called the *diameter* of the pointset.
3. An adjacency matrix can be thought of as representing a graph of vertices and edges where a value of 1 in position a_{ij} indicates an edge between vertex i and vertex j, whereas $a_{ij} = 0$ indicates no such edge between vertices i and j.

```
In[1]:= mat = RandomInteger[1, {5, 5}];
        MatrixForm[mat]
Out[2]//MatrixForm=
```

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

```
In[3]:= AdjacencyGraph[mat, VertexLabels → "Name"]
```

Out[3]=



Compute the total number of edges for each vertex in both the adjacency matrix and graph representations. For example, you should get the following edge counts for the five vertices represented in the above adjacency matrix. Note: self-loops count as two edges each.

```
{3, 4, 7, 5, 5}
```

4. Create a function `ToGraph`[*lis*] that takes a list of pairs of elements and transforms it into a list of graph (directed) edges. For example:

```
In[4]:= lis = RandomInteger[9, {12, 2}]
Out[4]= {{4, 3}, {6, 4}, {0, 1}, {6, 0}, {5, 2}, {4, 7},
         {6, 4}, {7, 1}, {7, 6}, {7, 8}, {4, 0}, {3, 4}}
```

```
In[5]:= ToGraph[lis]

Out[5]= {4 ⟷ 3, 6 ⟷ 4, 0 ⟷ 1, 6 ⟷ 0, 5 ⟷ 2,
         4 ⟷ 7, 6 ⟷ 4, 7 ⟷ 1, 7 ⟷ 6, 7 ⟷ 8, 4 ⟷ 0, 3 ⟷ 4}
```

Make sure that your function also works in the case where its argument is a single list of a pair of elements.

```
In[6]:= ToGraph[{3, 6}]

Out[6]= 3 ⟷ 6
```

5. Create a function `RandomColor[]` that generates a random RGB color. Add a rule for `RandomColor[`*n*`]` to create a list of *n* random colors.
6. Create a graphic that consists of *n* circles in the plane with random centers and random radii. Consider using `Thread` or `MapThread` to thread `Circle[…]` across the lists of centers and radii. Use `RandomColor` from the previous exercise to give each circle a random color.
7. Use `MapThread` and `Apply` to mirror the behavior of `Inner`.
8. While matrices can easily be added using `Plus`, matrix multiplication is a bit more involved. The `Dot` function, written as a single period, is used.

   ```
   In[7]:= {{1, 2}, {3, 4}}.{x, y}

   Out[7]= {x + 2 y, 3 x + 4 y}
   ```

   Perform matrix multiplication on `{{1, 2}, {3, 4}}` and `{x, y}` without using `Dot`.
9. `FactorInteger[`*n*`]` returns a nested list of prime factors and their exponents for the number *n*.

   ```
   In[8]:= FactorInteger[3 628 800]

   Out[8]= {{2, 8}, {3, 4}, {5, 2}, {7, 1}}
   ```

   Use `Apply` to reconstruct the original number from this nested list.
10. Repeat the above exercise but instead use `Inner` to reconstruct the original number *n* from the factorization given by `FactorInteger[`*n*`]`.
11. Create a function `PrimeFactorForm[`*n*`]` that formats its argument *n* in prime factorization form. For example:

    ```
    In[9]:= PrimeFactorForm[12]
    ```

    Out[9]= $2^2 \cdot 3^1$

    You will need to use `Superscript` and `CenterDot` to format the factored integer.
12. The Vandermonde matrix arises in Lagrange interpolation and in reconstructing statistical distributions from their moments. Construct the Vandermonde matrix of order *n*, which should look like the following:

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix}$$

13. Using `Inner`, write a function `div[`*vecs*`, `*vars*`]` that computes the divergence of an *n*-dimensional vector field, *vecs* = $\{e_1, e_2, \ldots, e_n\}$ dependent upon *n* variables, *vars* = $\{v_1, v_2, \ldots, v_n\}$. The divergence is given by the sum of the pairwise partial derivatives.

$$\frac{\partial e_1}{\partial v_1} + \frac{\partial e_2}{\partial v_2} + \cdots + \frac{\partial e_n}{\partial v_n}$$

14. The example in the section on `Select` and `Pick` found those Mersenne numbers $2^n - 1$ that are prime doing the computation for all exponents *n* from 1 to 100. Modify that example to only use prime exponents (since a basic theorem in number theory states that a Mersenne number with composite exponent must be composite).

5.2 Solutions

1. First, here is the definition given in Section 4.1.

```
In[1]:= SquareMatrixQ[mat_?MatrixQ] :=
         Dimensions[mat][[1]] == Dimensions[mat][[2]]
```

For a matrix, `Dimensions` returns a list of two integers. Applying `Equal` to the list will return `True` if the two dimensions are identical, that is, if the matrix is square.

```
In[2]:= SquareMatrixQ[mat_?MatrixQ] := Apply[Equal, Dimensions[mat]]
```

2. First create a set of points with which to work.

```
In[3]:= points = RandomReal[1, {100, 2}];
```

The set of all two-element subsets is given by:

```
In[4]:= pairs = Subsets[points, {2}];
```

Apply the distance function to `pairs`. Note the need to apply `EuclideanDistance` at level 1.

```
In[5]:= Apply[EuclideanDistance, pairs, {1}];
```

The maximum distance (diameter) is given by `Max`.

```
In[6]:= Max[%]

Out[6]= 1.22117
```

Here is a function that puts it all together.

```
In[7]:= PointsetDiameter[pts_List] :=
         Max[Apply[EuclideanDistance, Subsets[pts, {2}], {1}]]
```

```
In[8]:=  PointsetDiameter[points]

Out[8]=  1.22117
```

In fact, this function works on *n*-dimensional point sets.

```
In[9]:=  points3D = RandomReal[1, {5, 3}]

Out[9]=  {{0.909071, 0.903087, 0.299482},
          {0.696615, 0.317125, 0.391343}, {0.544775, 0.465739, 0.847848},
          {0.368606, 0.928498, 0.77199}, {0.582795, 0.993437, 0.74005}}

In[10]:= PointsetDiameter[points3D]

Out[10]= 0.791365
```

3. Here is a test matrix.

```
In[11]:= mat = RandomInteger[1, {5, 5}];
         MatrixForm[mat]
Out[12]//MatrixForm=
```

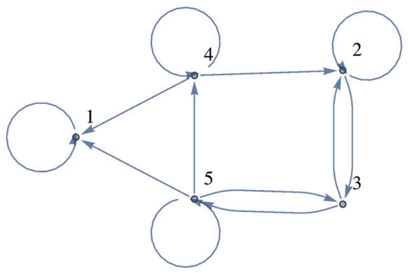
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

A bit of thought should convince you that adding the matrix to its transpose and then totaling all the 1s in each row will give the correct count.

```
In[13]:= Map[Total, mat + Transpose[mat]]

Out[13]= {4, 5, 4, 5, 6}
```

Using graphs you can accomplish the same thing.

```
In[14]:= gr = AdjacencyGraph[mat, VertexLabels → "Name"]

Out[14]=
```



```
In[15]:= VertexDegree[gr]

Out[15]= {4, 5, 4, 5, 6}
```

4. Applying `DirectedEdge` at level 1 will do the trick.

```
In[16]:=  ToGraph[lis : {{_, _} ..}] := Apply[DirectedEdge, lis, {1}]

In[17]:=  lis = RandomInteger[9, {12, 2}];
          ToGraph[lis]

Out[18]=  {3 ⧴ 8, 6 ⧴ 9, 9 ⧴ 8, 2 ⧴ 7, 0 ⧴ 7,
           1 ⧴ 0, 2 ⧴ 5, 1 ⧴ 0, 8 ⧴ 6, 2 ⧴ 6, 7 ⧴ 7, 6 ⧴ 0}
```

This rule fails for the case when the argument is a single flat list of a pair of elements.

```
In[19]:=  ToGraph[{3, 6}]

Out[19]=  ToGraph[{3, 6}]
```

A second rule is needed for this case.

```
In[20]:=  ToGraph[lis : {_, _}] := Apply[DirectedEdge, lis]

In[21]:=  ToGraph[{3, 6}]

Out[21]=  3 ⧴ 6
```

5. RGBColor takes a sequence of three values between 0 and 1. So you only need to apply RGBColor to this list.

```
In[22]:=  RandomColor[] := Apply[RGBColor, RandomReal[1, {3}]]
```

A second rule uses pattern matching to make sure the argument, *n*, to RandomColor is a positive integer; then create a list of *n* triples of random reals before applying RGBColor at level 1.

```
In[23]:=  RandomColor[n_Integer?Positive] :=
           Apply[RGBColor, RandomReal[1, {n, 3}], {1}]
```

6. First, create the random centers and radii.

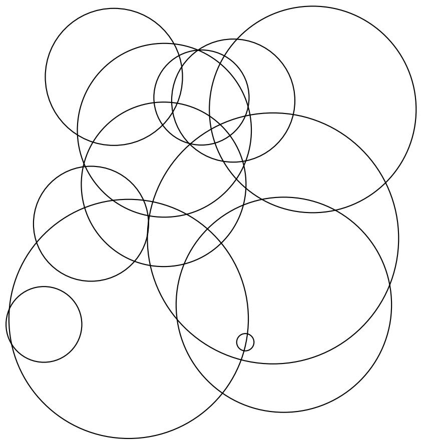
```
In[24]:=  n = 12;
          centers = RandomReal[{-1, 1}, {n, 2}]

Out[25]=  {{-0.33306, -0.757053}, {-0.604635, -0.0691777},
           {0.995937, 0.75683}, {0.787243, -0.654724},
           {-0.439541, 0.992207}, {0.192561, 0.843047},
           {-0.944553, -0.796825}, {0.509232, -0.925483},
           {-0.0752457, 0.606451}, {0.709054, -0.175166},
           {0.422127, 0.82078}, {-0.0804006, 0.215504}}

In[26]:=  radii = RandomReal[1, {n}]

Out[26]=  {0.864193, 0.41542, 0.746293, 0.777527, 0.495235, 0.343537,
           0.273533, 0.0624348, 0.62805, 0.906858, 0.444684, 0.59442}
```

MapThread is perfect for the task of grabbing one center, one radii, and wrapping Circle around them.

```
In[27]:=  (circles = MapThread[Circle, {centers, radii}]) // Short

Out[27]//Short=  {Circle[≪1≫], ≪10≫, Circle[{-≪18≫, ≪20≫}, ≪19≫]}
```

```
In[28]:= Graphics[circles]
```

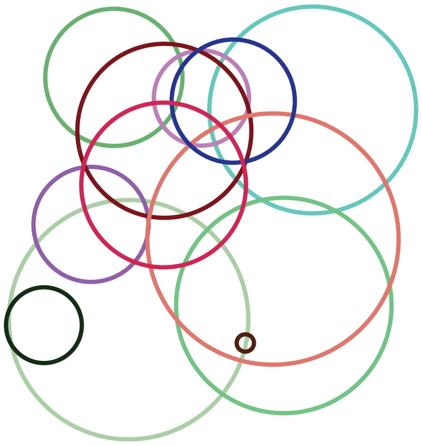
Out[28]=



And here is a rule to transform each circle into a scoped list that includes `Thick` and `RandomColor`. Note the need for the delayed rule (:→).

```
In[29]:= Graphics[circles /. Circle[x__] :→ {Thick, RandomColor[], Circle[x]}]
```

Out[29]=



7. Here is the `Inner` example from the text.

```
In[30]:= Inner[f, {a, b, c}, {d, e, f}, g]

Out[30]= g[f[a, d], f[b, e], f[c, f]]
```

Using `MapThread`, we zip together the two lists and wrap `f` around each pair. Then apply `g`.

```
In[31]:= MapThread[f, {{a, b, c}, {d, e, f}}]

Out[31]= {f[a, d], f[b, e], f[c, f]}
```

```
In[32]:=  Apply[g, %]

Out[32]=  g[f[a, d], f[b, e], f[c, f]]
```

8. This can be done either in two steps, or by using the `Inner` function.

```
In[33]:=  Transpose[{{1, 2}, {3, 4}}] {x, y}

Out[33]=  {{x, 3 x}, {2 y, 4 y}}

In[34]:=  Total[%]

Out[34]=  {x + 2 y, 3 x + 4 y}

In[35]:=  Inner[Times, {{1, 2}, {3, 4}}, {x, y}, Plus]

Out[35]=  {x + 2 y, 3 x + 4 y}
```

9. To get down to the level of the nested lists, you have to use a second argument to `Apply`.

```
In[36]:=  facs = FactorInteger[3 628 800]

Out[36]=  {{2, 8}, {3, 4}, {5, 2}, {7, 1}}

In[37]:=  Apply[Power, facs, {1}]

Out[37]=  {256, 81, 25, 7}
```

One more use of `Apply` is needed to multiply these terms.

```
In[38]:=  Apply[Times, %]

Out[38]=  3 628 800
```

Here is a function that puts this all together.

```
In[39]:=  ExpandFactors[lis_] := Apply[Times, Apply[Power, lis, {1}]]

In[40]:=  FactorInteger[295 232 799 039 604 140 847 618 609 643 520 000 000]

Out[40]=  {{2, 32}, {3, 15}, {5, 7}, {7, 4}, {11, 3},
           {13, 2}, {17, 2}, {19, 1}, {23, 1}, {29, 1}, {31, 1}}

In[41]:=  ExpandFactors[%]

Out[41]=  295 232 799 039 604 140 847 618 609 643 520 000 000
```

10. Here is a factorization we can use to work through this problem.

```
In[42]:=  facs = FactorInteger[3 628 800]

Out[42]=  {{2, 8}, {3, 4}, {5, 2}, {7, 1}}
```

Another approach uses `Transpose` to separate the bases from their exponents, then uses `Inner` to put things back together.

```
In[43]:= {base, exponents} = Transpose[facs]
Out[43]= {{2, 3, 5, 7}, {8, 4, 2, 1}}

In[44]:= Inner[Power, base, exponents, Times]
Out[44]= 3 628 800
```

Since `Tranpose` returns a list of two lists in this example, we need to strip the outer list. This is done by applying `Sequence`.

```
In[45]:= ExpandFactors2[lis_] :=
           Inner[Power, Sequence @@ Transpose[lis], Times]

In[46]:= ExpandFactors2[facs]
Out[46]= 3 628 800
```

11. First, here is the prime factorization of a test integer:

```
In[47]:= lis = FactorInteger[10 !]
Out[47]= {{2, 8}, {3, 4}, {5, 2}, {7, 1}}
```

Apply `Superscript` at level 1 to each of the sublists:

```
In[48]:= Apply[Superscript, lis, {1}]
```

Out[48]= $\{2^8, 3^4, 5^2, 7^1\}$

Finally, apply `CenterDot` to this list.

```
In[49]:= Apply[CenterDot, %]
```

Out[49]= $2^8 \cdot 3^4 \cdot 5^2 \cdot 7^1$

Put it all together (using shorthand notation for `Apply`) and `Apply` at level 1.

```
In[50]:= PrimeFactorForm[p_] :=
           CenterDot @@ (Superscript @@@ FactorInteger[p])

In[51]:= PrimeFactorForm[20 !]
```

Out[51]= $2^{18} \cdot 3^8 \cdot 5^4 \cdot 7^2 \cdot 11^1 \cdot 13^1 \cdot 17^1 \cdot 19^1$

Unfortunately, this rule fails for numbers that have only one prime factor.

```
In[52]:= PrimeFactorForm[9]
```

Out[52]= CenterDot$[3^2]$

A second rule is needed for this special case.

```
In[53]:= PrimeFactorForm[p_ ? PrimePowerQ] :=
           First[Superscript @@@ FactorInteger[p]]
```

```
In[54]:= PrimeFactorForm[9]
```

```
Out[54]= 3^2
```

A subtle point is that *Mathematica* has automatically ordered these two rules, putting the one involving prime powers first.

```
In[55]:= ? PrimeFactorForm
```

```
Global`PrimeFactorForm

PrimeFactorForm[p_?PrimePowerQ] :=
 First[Apply[Superscript, FactorInteger[p], {1}]]

PrimeFactorForm[p_] :=
 CenterDot @@ Apply[Superscript, FactorInteger[p], {1}]
```

This reordering (we evaluated the rules in a different order) is essential for this function to work properly. If the general rule was checked first, it would apply to arguments that happen to be prime powers and it would give wrong answers.

One final point: the expressions returned by `PrimeFactorForm` will not evaluate like ordinary expressions due to the use of `CenterDot` which has no evaluation rules associated with it. You could add an "interpretation" to such expressions by using `Interpretation[`*disp*`, `*expr*`]` as follows.

```
In[56]:= PrimeFactorForm[p_Integer] := With[{fp = FactorInteger[p]},
           Interpretation[
            CenterDot @@ (Superscript @@@ fp),
            Times @@ (Power @@@ fp)]]
```

Now the output of the following expression can be evaluated directly to get an interpreted result.

```
In[57]:= PrimeFactorForm[12 !]
```

```
Out[57]= 2^10 · 3^5 · 5^2 · 7^1 · 11^1
```

12. This is a straightforward application of the `Outer` function.

```
In[58]:= VandermondeMatrix[n_, x_] :=
          Outer[Power, Table[x_i, {i, 1, n}], Range[0, n - 1]]
```

```
In[59]:= VandermondeMatrix[4, x] // MatrixForm
```

Out[59]//MatrixForm=

$$\begin{pmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \end{pmatrix}$$

13. If we first look at a symbolic result, we should be able to see how to construct our function. For three vectors and three variables, here is the divergence (think of `d` as the derivative operator).

```
In[60]:= Inner[d, {e1, e2, e3}, {v1, v2, v3}, Plus]

Out[60]= d[e1, v1] + d[e2, v2] + d[e3, v3]
```

So for arbitrary-length vectors and variables, we have:

```
In[61]:= div[vecs_, vars_] := Inner[D, vecs, vars, Plus]
```

As a check, we can compute the divergence of the standard gravitational or electric force field, which should be zero.

```
In[62]:= div[{x, y, z} / (x^2 + y^2 + z^2)^(3/2), {x, y, z}]
```

$$\text{Out[62]= } -\frac{3\,x^2}{\left(x^2+y^2+z^2\right)^{5/2}}-\frac{3\,y^2}{\left(x^2+y^2+z^2\right)^{5/2}}-\frac{3\,z^2}{\left(x^2+y^2+z^2\right)^{5/2}}+\frac{3}{\left(x^2+y^2+z^2\right)^{3/2}}$$

```
In[63]:= Simplify[%]

Out[63]= 0
```

Finally, we should note that this definition of divergence is a bit delicate as we are doing no argument checking at this point. For example, it would be sensible to insure that the length of the vector list is the same as the length of the variable list before starting the computation. Refer to Chapter 4 for a discussion of how to use pattern matching to deal with this issue.

14. First create a table of primes and then use that list for values of p in the second table.

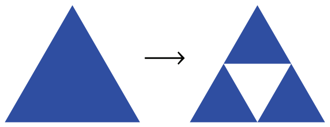
```
In[64]:= primes = Table[Prime[n], {n, 1, 50}]

Out[64]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
          47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
          109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
          173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229}

In[65]:= Select[Table[2^p - 1, {p, primes}], PrimeQ]

Out[65]= {3, 7, 31, 127, 8191, 131 071, 524 287, 2 147 483 647,
          2 305 843 009 213 693 951, 618 970 019 642 690 137 449 562 111,
          162 259 276 829 213 363 391 578 010 288 127,
          170 141 183 460 469 231 731 687 303 715 884 105 727}
```

Or you could do the same thing more directly.

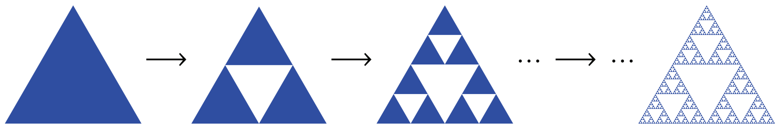
```
In[66]:= Select[Table[2^Prime[n] - 1, {n, 1, 50}], PrimeQ]

Out[66]= {3, 7, 31, 127, 8191, 131 071, 524 287, 2 147 483 647,
          2 305 843 009 213 693 951, 618 970 019 642 690 137 449 562 111,
          162 259 276 829 213 363 391 578 010 288 127,
          170 141 183 460 469 231 731 687 303 715 884 105 727}
```

5.3 *Iterating functions*

1. Determine the locations after each step of a ten-step one-dimensional random walk. (Recall that you have already generated the step *directions* in Exercise 3 at the end of Section 3.1.)
2. Create a list of the step locations of a ten-step random walk on a square lattice.
3. Using `Fold`, create a function `fac[n]` that takes an integer n as argument and returns the factorial of n, that is, $n(n-1)(n-2)\cdots 3\cdot 2\cdot 1$.
4. The Sierpinski triangle is a classic iteration example. It is constructed by starting with an equilateral triangle (other objects can be used) and removing the inner triangle formed by connecting the midpoints of each side of the original triangle.



The process is iterated by repeating the same computation on each of the resulting smaller triangles.



One approach is to take the starting equilateral triangle and, at each iteration, perform the appropriate transformations using `Scale` and `Translate`, then iterate. Implement this algorithm, but be careful about nesting large graphical structures too deeply.

5.3 *Solutions*

1. First generate the step directions.

```
In[1]:= Table[(-1)^Random[Integer], {10}]

Out[1]= {-1, -1, -1, -1, -1, 1, -1, -1, 1, 1}
```

Or the following also works.

```
In[2]:= steps = 2 RandomInteger[1, {10}] - 1

Out[2]= {-1, -1, -1, -1, 1, 1, -1, 1, 1, 1}
```

Then, starting at 0, the fold operation generates the locations.

```
In[3]:= FoldList[Plus, 0, steps]

Out[3]= {0, -1, -2, -3, -4, -3, -2, -3, -2, -1, 0}
```

2. Use the method of generating a list of step locations that was shown in an earlier exercise.

```
In[4]:= steps = RandomChoice[{{1, 0}, {-1, 0}, {0, 1}, {0, -1}}, {10}]

Out[4]= {{1, 0}, {0, 1}, {1, 0}, {0, 1}, {1, 0},
         {0, -1}, {0, 1}, {0, -1}, {1, 0}, {0, 1}}

In[5]:= FoldList[Plus, {0, 0}, steps]

Out[5]= {{0, 0}, {1, 0}, {1, 1}, {2, 1}, {2, 2},
         {3, 2}, {3, 1}, {3, 2}, {3, 1}, {4, 1}, {4, 2}}
```

3. Starting with 1, fold the `Times` function across the first *n* integers.

```
In[6]:= fac[n_] := Fold[Times, 1, Range[n]]

In[7]:= fac[10]

Out[7]= 3 628 800
```

4. First create the vertices of the triangle. Wrapping them in `N[…]` helps to keep the graphical structures small (see Section 10.2 for more on this).

```
In[8]:= vertices = N[{{0, 0}, {1, 0}, {1 / 2, 1}}];
```

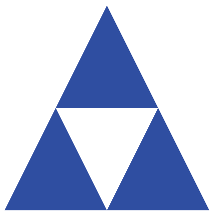
This gives the three different translation vectors.

```
In[9]:= translateVecs = 0.5 vertices

Out[9]= {{0., 0.}, {0.5, 0.}, {0.25, 0.5}}
```

Here is the set of transformations of the triangle described by `vertices`, scaled by 0.5, and translated according to the translation vectors.

```
In[10]:= tri = Polygon[vertices];
         Graphics[{
           Blue, Translate[Scale[tri, 0.5, {0., 0.}], translateVecs]
          }]
```

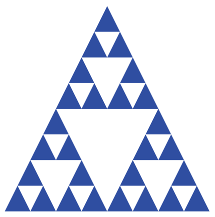
Out[11]=



Finally, iterate the transformations by wrapping them in `Nest`.

```
In[12]:= Graphics[{Blue,
           Nest[{Blue, Translate[Scale[#, 0.5, {0., 0.}], translateVecs]} &,
            Polygon[vertices], 3]}]
```

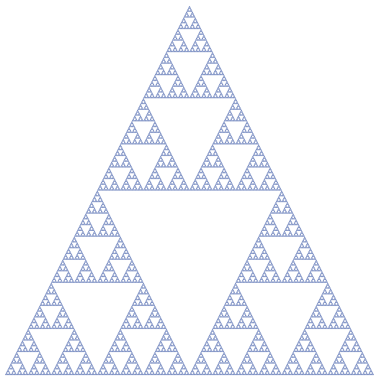
Out[12]=



Once you have been through the rest of this chapter, you should be able to turn this into a reusable function, scoping local variables, using pure functions, and adding options.

```
In[13]:= SierpinskiTriangle[iter_, opts : OptionsPattern[Graphics]] :=
          Module[{vertices, vecs},
           vertices = N[{{0, 0}, {1, 0}, {1 / 2, 1}}];
           vecs = 0.5 vertices;
           Graphics[
            {Blue, Nest[{Blue, Translate[Scale[#, 0.5, {0., 0.}], vecs]} &,
              Polygon[vertices], iter]}, opts]]

In[14]:= SierpinskiTriangle[8, ImageSize → Small]
```

Out[14]=



5.4 *Programs as functions*

1. Using `Total`, create a function to sum the first *n* positive integers.
2. Rewrite the `listEvenQ` function from this section using `MemberQ`.
3. Using the `shuffle` function developed in this section, how many shuffles of a deck of cards (or any list, for that matter) are needed to return the deck to its original order?

4. Many lotteries include games that require you to pick several numbers and match them against the "house." The numbers are independent, so this is essentially random sampling with replacement. The built-in `RandomChoice` does this. For example, here are five random samples from the integers 0 through 9.

```
In[1]:= RandomChoice[Range[0, 9], 5]

Out[1]= {4, 1, 8, 7, 4}
```

   Write your own function `randomChoice`[*lis*, *n*] that performs a random sampling with replacement, where *n* is the number of elements being chosen from the list *lis*. Here is a typical result using a list of symbols.

```
In[2]:= randomChoice[{a, b, c, d, e, f, g, h}, 12]

Out[2]= {g, c, a, a, d, h, c, a, c, f, c, a}
```

5. Use `Trace` on the rule-based `maxima` from Section 4.2 and `maxima` developed in this section to explain why the functional version is much faster than the pattern matching version.
6. Write your own user-defined functions using the `Characters` and `StringJoin` functions to perform the same operations as `StringInsert` and `StringDrop`.
7. Write a function `interleave` that interleaves the elements of two lists of unequal length. (You have already seen how to interleave lists of equal length using `Partition` earlier in this section with the `shuffle` function.) Your function should take the lists `{a, b, c, d}` and `{1, 2, 3}` as inputs and return `{a, 1, b, 2, c, 3, d}`.
8. Write nested function calls using `ToCharacterCode` and `FromCharacterCode` to perform the same operations as the built-in `StringJoin` and `StringReverse` functions.

5.4 *Solutions*

1. Generate the list of integers 1 through *n*, then total that list.

```
In[1]:= sumInts[n_] := Total[Range[n]]

In[2]:= sumInts[100]

Out[2]= 5050

In[3]:= sumInts[1000]

Out[3]= 500 500
```

   We have not been careful to check that the arguments are positive integers here. See Section 5.6 for a proper definition to check arguments.

2. Use `MemberQ` to check if any elements of the list pass the `OddQ` test. If they do, `True` is returned and so we take the Boolean negation of that. In other words, if the list contains an odd number, `False` is returned, indicating that the list does not consist of even numbers exclusively.

```
In[4]:= listEvenQ2[lis_] := Not[MemberQ[lis, _?OddQ]]
```

```
In[5]:=  listEvenQ2[{2, 4, 6, 4, 8}]

Out[5]=  True

In[6]:=  listEvenQ2[{2, 4, 6, 5, 8}]

Out[6]=  False
```

Alternatively, you could have `FreeQ` check to see if the list is free of numbers that are equal to 1 mod 2.

```
In[7]:=  listEvenQ3[lis_] := FreeQ[lis, p_ /; Mod[p, 2] == 1]

In[8]:=  listEvenQ3[{2, 4, 6, 4, 8}]

Out[8]=  True

In[9]:=  listEvenQ3[{2, 4, 6, 5, 8}]

Out[9]=  False
```

3. Some simple experiments iterating the `shuffle` function shows that the number of shuffles to return the deck to its original state is dependent upon the number of cards in the deck. For a deck of 52 cards, eight such perfect (Faro) shuffles will return the deck to its original state.

```
In[10]:= shuffle[lis_] := Module[{len = Ceiling[Length[lis] / 2]},
           Apply[Riffle, Partition[lis, len, len, 1, {}]]]

In[11]:= Nest[shuffle, Range[52], 8]

Out[11]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
          21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
          37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52}
```

4. The obvious way to do this is to take the list and simply pick out elements at random locations. The right-most location in the list is given by `Length[lis]`, using `Part` and `RandomInteger`.

```
In[12]:= randomChoice[lis_, n_] := lis⟦RandomInteger[{1, Length[lis]}, {n}]⟧

In[13]:= randomChoice[{a, b, c, d, e, f, g, h}, 12]

Out[13]= {h, b, c, f, c, f, b, g, h, e, e, b}
```

5. The pattern matched function is slower because it repeatedly applies transformation rules.

```
In[14]:= maxima[lis_List] :=
          lis //.{a___, b_, c___, d_, e___} /; d ≤ b :→ {a, b, c, e}

In[15]:= maximaF[lis_] := Rest[DeleteDuplicates[FoldList[Max, -∞, lis]]]
```

```
In[16]:= Trace[maxima[{3, 5, 2, 6, 1, 8, 4, 9, 7}]]

Out[16]= {maxima[{3, 5, 2, 6, 1, 8, 4, 9, 7}], {3, 5, 2, 6, 1, 8, 4, 9, 7} //.
           {a___, b_, c___, d_, e___} /; d ≤ b :→ {a, b, c, e},
          {{a___, b_, c___, d_, e___} /; d ≤ b :→ {a, b, c, e},
           {a___, b_, c___, d_, e___} /; d ≤ b :→ {a, b, c, e}},
          {3, 5, 2, 6, 1, 8, 4, 9, 7} //.
           {a___, b_, c___, d_, e___} /; d ≤ b :→ {a, b, c, e},
          {5 ≤ 3, False}, {2 ≤ 3, True}, {5 ≤ 3, False}, {6 ≤ 3, False},
          {1 ≤ 3, True}, {5 ≤ 3, False}, {6 ≤ 3, False}, {8 ≤ 3, False},
          {4 ≤ 3, False}, {9 ≤ 3, False}, {7 ≤ 3, False}, {6 ≤ 5, False},
          {8 ≤ 5, False}, {4 ≤ 5, True}, {5 ≤ 3, False}, {6 ≤ 3, False},
          {8 ≤ 3, False}, {9 ≤ 3, False}, {7 ≤ 3, False}, {6 ≤ 5, False},
          {8 ≤ 5, False}, {9 ≤ 5, False}, {7 ≤ 5, False}, {8 ≤ 6, False},
          {9 ≤ 6, False}, {7 ≤ 6, False}, {9 ≤ 8, False}, {7 ≤ 8, True},
          {5 ≤ 3, False}, {6 ≤ 3, False}, {8 ≤ 3, False}, {9 ≤ 3, False},
          {6 ≤ 5, False}, {8 ≤ 5, False}, {9 ≤ 5, False}, {8 ≤ 6, False},
          {9 ≤ 6, False}, {9 ≤ 8, False}, {3, 5, 6, 8, 9}}

In[17]:= Trace[maximaF[{3, 5, 2, 6, 1, 8, 4, 9, 7}]]

Out[17]= {maximaF[{3, 5, 2, 6, 1, 8, 4, 9, 7}], Rest[
           DeleteDuplicates[FoldList[Max, -∞, {3, 5, 2, 6, 1, 8, 4, 9, 7}]]],
          {{{{∞, ∞}, -∞, -∞}, FoldList[Max, -∞, {3, 5, 2, 6, 1, 8, 4, 9, 7}],
             {Max[-∞, 3], Max[3, -∞], 3}, {Max[3, 5], 5}, {Max[5, 2],
              Max[2, 5], 5}, {Max[5, 6], 6}, {Max[6, 1], Max[1, 6], 6},
             {Max[6, 8], 8}, {Max[8, 4], Max[4, 8], 8}, {Max[8, 9], 9},
             {Max[9, 7], Max[7, 9], 9}, {-∞, 3, 5, 5, 6, 6, 8, 8, 9, 9}},
           DeleteDuplicates[{-∞, 3, 5, 5, 6, 6, 8, 8, 9, 9}],
           {-∞, 3, 5, 6, 8, 9}},
          Rest[{-∞, 3, 5, 6, 8, 9}], {3, 5, 6, 8, 9}}
```

6. Here is our user-defined `stringInsert`.

```
In[18]:= stringInsert[str1_, str2_, pos_] := StringJoin@Join[
             Take[Characters[str1], pos - 1],
             Characters[str2],
             Drop[Characters[str1], pos - 1]
           ]

In[19]:= stringInsert["Joy world", "to the ", 5]

Out[19]= Joy to the world

In[20]:= stringDrop[str_, pos_] := StringJoin[Drop[Characters[str], pos]]

In[21]:= stringDrop["ABCDEF", -2]

Out[21]= ABCD
```

The idea in these two examples is to convert a string to a list of characters, operate on that list using list manipulation functions like Join, Take, and Drop, then convert back to a string. More efficient approaches use string manipulation functions directly (see Chapter 9).

7. We assume that lis1 is longer than lis2 and pair off the corresponding elements in the lists and then tack on the leftover elements from lis1.

```
In[22]:= interleave[lis1_, lis2_] :=
          Flatten[Join[Transpose[{Take[lis1, Length[lis2]], lis2}],
            Take[lis1, Length[lis2] - Length[lis1]]]]

In[23]:= interleave[{a, b, c, d}, {1, 2, 3}]

Out[23]= {a, 1, b, 2, c, 3, d}
```

Compare with the built-in Riffle.

```
In[24]:= Riffle[{a, b, c, d}, {1, 2, 3}]

Out[24]= {a, 1, b, 2, c, 3, d}
```

8. First, here is how we might write our own StringJoin.

```
In[25]:= FromCharacterCode[Join[
           ToCharacterCode["To be, "], ToCharacterCode["or not to be"]
          ]]

Out[25]= To be, or not to be
```

And here is a how we might implement a StringReverse.

```
In[26]:= FromCharacterCode[Reverse[ToCharacterCode[%]]]

Out[26]= eb ot ton ro ,eb oT
```

5.5 *Scoping constructs*

1. Write a compound function definition for the location of steps taken in an *n*-step random walk on a square lattice. The step directions can be taken to be the compass directions with north represented by {1, 0}, south by {-1, 0}, and so on. Hint: consider using the Accumulate function.

2. The PerfectSearch function defined in Section 1.1 is impractical for checking large numbers because it has to check all numbers from 1 through *n*. If you already know the perfect numbers below 500, say, it is inefficient to check all numbers from 1 to 1000 if you are only looking for perfect numbers in the range 500 to 1000. Modify PerfectSearch so that it accepts two numbers as input and finds all perfect numbers between the inputs. For example, PerfectSearch[*a*, *b*] will produce a list of all perfect numbers in the range from *a* to *b*.

3. A number, *n*, is *k*-perfect if the sum of its proper divisors equals *k n*. Redefine PerfectSearch from the previous exercise so that it accepts as input two numbers *a* and *b*, a positive integer *k*, and computes all *k*-perfect numbers in the range from *a* to *b*. Use your rule to find the only three 4-perfect numbers less than 2 200 000.

4. Often in processing files you are presented with expressions that need to be converted into a format that can be more easily manipulated inside *Mathematica*. For example, a file may contain dates in the form 20120515 to represent May 15, 2012. *Mathematica* represents its dates as a list in the form {*year*, *month*, *day*, *hour*, *minutes*, *seconds*}. Write a function `convertToDate[n]` to convert a number consisting of eight digits such as `20120515` into a list of the form `{2012, 5, 15}`.

```
In[2]:= convertToDate[20 120 515]

Out[2]= {2012, 5, 15}
```

5. Create a function `zeroColumns[mat, m ;; n]` that zeros out columns *m* through *n* in matrix *mat*. Include rules to handle the cases of zeroing out one column or a list of nonconsecutive columns.

5.5 *Solutions*

1. In the first definition, we only use one auxiliary function inside the `Module`.

```
In[1]:= latticeWalk2D[n_] :=
         Module[{NSEW = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}},
          Accumulate[RandomChoice[NSEW, n]]]

In[2]:= latticeWalk2D[10]

Out[2]= {{1, 0}, {1, -1}, {1, -2}, {0, -2}, {0, -3},
         {0, -2}, {0, -3}, {1, -3}, {0, -3}, {0, -4}}
```

2. The following function creates a local function `perfectQ` using the `Module` construct. It then checks every other number between *n* and *m* by using a third argument to the `Range` function.

```
In[3]:= PerfectSearch[n_, m_] := Module[{perfectQ},
          perfectQ[j_] := Total[Divisors[j]] == 2 j;
          Select[Range[n, m, 2], perfectQ]]

In[4]:= PerfectSearch[2, 10 000]

Out[4]= {6, 28, 496, 8128}
```

This function does not guard against the user supplying “bad” inputs. For example, if the user starts with an odd number, then this version of `PerfectSearch` will check every other odd number, and, since it is known that there are no odd perfect numbers below at least 10^{300}, none is reported.

```
In[5]:= PerfectSearch[1, 10 000]

Out[5]= {}
```

You can fix this situation by using the (as yet unproved) assumption that there are *no* odd perfect numbers. This next version first checks that the first argument is an even number.

```
In[6]:= Clear[PerfectSearch]

In[7]:= PerfectSearch[n_?EvenQ, m_] := Module[{perfectQ},
          perfectQ[j_] := Total[Divisors[j]] == 2 j;
          Select[Range[n, m, 2], perfectQ]]
```

Now, the function only works if the first argument is even.

```
In[8]:=  PerfectSearch[2, 10 000]

Out[8]=  {6, 28, 496, 8128}

In[9]:=  PerfectSearch[1, 1000]

Out[9]=  PerfectSearch[1, 1000]
```

3. This function requires a third argument.

```
In[10]:= Clear[PerfectSearch];
         PerfectSearch[n_, m_, k_] := Module[{perfectQ},
           perfectQ[j_] := Total[Divisors[j]] == k j;
           Select[Range[n, m], perfectQ]]
```

The following computation can be quite time consuming and requires a fair amount of memory to run to completion. If your computer's resources are limited, you should split up the search intervals into smaller units or try running this in parallel. See Section 12.3 for a discussion on how to set up parallel computation.

```
In[12]:= PerfectSearch[1, 2 200 000, 4] // AbsoluteTiming

Out[12]= {30.775270, {30 240, 32 760, 2 178 540}}
```

We also give a speed boost by using `DivisorSigma[1, j]` which gives the sum of the divisors of *j*.

```
In[13]:= PerfectSearchParallel[n_, m_, k_] :=
          Module[{perfectQ}, perfectQ[j_] := DivisorSigma[1, j] == k j;
           DistributeDefinitions[perfectQ];
           Parallelize[Select[Range[n, m, 2], perfectQ]]]

In[14]:= PerfectSearchParallel[2, 2 200 000, 4] // AbsoluteTiming

Out[14]= {5.698599, {30 240, 32 760, 2 178 540}}
```

4. Many implementations are possible for `convertToDate`. The task is made easier by observing that `DateList` handles this task directly if its argument is a string.

```
In[15]:= DateList["20120515"]

Out[15]= {2012, 5, 15, 0, 0, 0.}
```

The string is necessary otherwise `DateList` will interpret the integer as an absolute time (from Jan 1 1900).

```
In[16]:= DateList[20 120 515]

Out[16]= {1900, 8, 21, 21, 1, 55.}
```

So we need to convert the integer to a string first,

```
In[17]:= DateList[ToString[20 120 515]]

Out[17]= {2012, 5, 15, 0, 0, 0.}
```

and then take the first three elements.

```
In[18]:= Take[%, 3]

Out[18]= {2012, 5, 15}
```

Here is the function that puts these steps together.

```
In[19]:= convertToDate[n_Integer] := Take[DateList[ToString[n]], 3]

In[20]:= convertToDate[20 120 515]

Out[20]= {2012, 5, 15}
```

With a bit more manual work, you could also do this with `StringTake`.

```
In[21]:= convertToDate2[n_Integer /; Length[IntegerDigits[n]] == 8] :=
          Module[{str = ToString[n]}, {StringTake[str, 4],
            StringTake[str, {5, 6}], StringTake[str, -2]}]

In[22]:= convertToDate2[20 120 515]

Out[22]= {2012, 05, 15}
```

You could avoid working with strings by making use of `FromDigits`. This uses `With` to create a local constant `d`, as this expression never changes throughout the body of the function.

```
In[23]:= convertToDate3[num_] := With[{d = IntegerDigits[num]},
           {FromDigits[Take[d, 4]],
            FromDigits[Take[d, {5, 6}]],
            FromDigits[Take[d, {7, 8}]]}]

In[24]:= convertToDate3[20 120 515]

Out[24]= {2012, 5, 15}
```

5. The computation of zeroing out one or more columns of a matrix can be handled with list component assignment. We need to use a local variable here to avoid changing the original matrix.

```
In[25]:= mat = RandomReal[1, {5, 5}];
         MatrixForm[mat]

Out[26]//MatrixForm=
          0.451062  0.891983  0.184897   0.144232   0.568668
          0.575134  0.116466  0.194857   0.715349   0.0186447
          0.813932 0.0560997  0.13639  0.00183425   0.450973
          0.165999  0.585373  0.302838  0.0688475   0.804879
          0.639574  0.215239  0.147306  0.0116693   0.677564
```

Here is a rule for zeroing out one column:

```
In[27]:= zeroColumns[mat_, n_Integer] := Module[{lmat = mat},
           lmat[[All, n]] = 0;
           lmat]
```

This next rule is for zeroing out a range of columns:

```
In[28]:=  zeroColumns[mat_, Span[m_, n_]] := Module[{lmat = mat},
             lmat[[All, m ;; n]] = 0;
             lmat]
```

We also need a final rule for zeroing out a discrete set of columns whose positions are given by a list.

```
In[29]:=  zeroColumns[mat_, lis : {__}] := Module[{lmat = mat},
             lmat[[All, lis]] = 0;
             lmat]
```

```
In[30]:=  zeroColumns[mat, 3] // MatrixForm
Out[30]//MatrixForm=
          ( 0.451062  0.891983  0   0.144232    0.568668  )
          ( 0.575134  0.116466  0   0.715349   0.0186447  )
          ( 0.813932 0.0560997  0  0.00183425   0.450973  )
          ( 0.165999  0.585373  0   0.0688475   0.804879  )
          ( 0.639574  0.215239  0   0.0116693   0.677564  )
```

```
In[31]:=  zeroColumns[mat, 1 ;; 2] // MatrixForm
Out[31]//MatrixForm=
          ( 0  0  0.184897   0.144232    0.568668  )
          ( 0  0  0.194857   0.715349   0.0186447  )
          ( 0  0   0.13639  0.00183425   0.450973  )
          ( 0  0  0.302838   0.0688475   0.804879  )
          ( 0  0  0.147306   0.0116693   0.677564  )
```

```
In[32]:=  zeroColumns[mat, {1, 3, 5}] // MatrixForm
Out[32]//MatrixForm=
          ( 0   0.891983  0   0.144232   0 )
          ( 0   0.116466  0   0.715349   0 )
          ( 0  0.0560997  0  0.00183425  0 )
          ( 0   0.585373  0   0.0688475  0 )
          ( 0   0.215239  0   0.0116693  0 )
```

5.6 *Pure functions*

1. Write a function to sum the squares of the elements of a numeric list.
2. In Exercise 2 from Section 5.2 you were asked to create a function to compute the diameter of a set of points in *n*-dimensional space. Modify that solution by instead using the `Norm` function and pure functions to find the diameter.
3. Rewrite the code from Section 5.3 for finding the next prime after a given integer so that it uses pure functions instead of relying upon auxiliary definitions `addOne` and `CompositeQ`.
4. Create a function `RepUnit[`*n*`]` that generates integers of length *n* consisting entirely of ones. For example `RepUnit[7]` should produce 1111111.
5. Given a set of numerical data, extract all those data points that are within one standard deviation of the mean of the data.

```
In[1]:=  data = RandomVariate[NormalDistribution[0, 1], {2500}];
```

6. Write a pure 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, starting at `{0, 0}`, the function should return `{0, 1}`, `{0, -1}`, `{1, 0}`, or `{-1, 0}` with equal likelihood. Now, use this pure function with `NestList` to generate the list of step locations for an *n*-step random walk starting at `{0, 0}`.

7. Find all words in the dictionary that start with the letter *q* and are of length five. Here is the list of words in the dictionary that comes with *Mathematica*.

```
In[2]:=  words = DictionaryLookup[];
         RandomSample[words, 24]

Out[3]=  {leafage, uncorrupted, cocci, disadvantaged, inflicter, Moira,
          interpolates, squander, archer, tricking, lithosphere,
          deforested, throb, soapboxes, monopolies, advisedly, silencer,
          tames, satanists, individuals, snorter, huh, noised, WWW}
```

8. A naive approach to polynomial arithmetic would require three additions and six multiplications to carry out the arithmetic in the expression $a x^3 + b x^2 + c x + d$. Using Horner's method for fast polynomial multiplication, this expression can be represented as $d + x(c + x(b + a x))$, where there are now half as many multiplications. You can see this using the `MultiplyCount` function developed in Exercise 8 of Section 4.2.

```
In[4]:=  MultiplyCount[a x^3 + b x^2 + c x + d]

Out[4]=  6

In[5]:=  MultiplyCount[d + x (c + x (b + a x))]

Out[5]=  3
```

In general, the number of multiplications in an *n*-degree polynomial is given by:

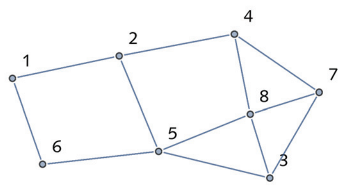
```
In[6]:=  Binomial[n + 1, 2]

Out[6]=  1/2 n (1 + n)
```

This, of course, grows quadratically with *n*, whereas Horner's method grows linearly. Create a function `Horner[`*lis*`,` *var*`]` that gives a representation of a polynomial in Horner form. Here is some sample output that your function should generate.

```
In[7]:=  Horner[{a, b, c, d}, x]

Out[7]=  d + x (c + x (b + a x))

In[8]:=  Expand[%]

Out[8]=  d + c x + b x^2 + a x^3
```

9. Graphs that are not too dense are often represented using *adjacency structures* which consist of a list for each vertex v_i that includes those other vertices that v_i is connected to. Create an adjacency structure for any graph, directed or undirected. For example, consider the graph `gr` below.

```
In[9]:=  gr = RandomGraph[{8, 12}, VertexLabels → "Name"]

Out[9]=
```



Start by creating an adjacency list for any given vertex; that is, a list of those vertices to which the given vertex is connected. For example, the adjacency list for vertex 8 in the above graph would be:

```
{3, 4, 5, 7}
```

The adjacency structure is then the list of adjacency lists for every vertex in that graph. It is common to prepend each adjacency list with its vertex; typically the adjacency structure takes the following form where this syntax indicates that vertex 1 is connected to vertices 2 and 6; vertex 2 is connected to vertices 1, 4, and 5; and so on.

```
{{1, {2, 6}}, {2, {1, 4, 5}}, {3, {5, 7, 8}}, {4, {2, 7, 8}},
 {5, {2, 3, 6, 8}}, {6, {1, 5}}, {7, {3, 4, 8}}, {8, {3, 4, 5, 7}}}
```

10. Use `FoldList` to compute an exponential moving average of a list `{x1, x2, x3}`. You can check your result against the built-in `ExponentialMovingAverage`.

```
In[10]:=  ExponentialMovingAverage[{x1, x2, x3}, α]

Out[10]=  {x1, x1 + α (-x1 + x2), x1 + α (-x1 + x2) + α (-x1 - α (-x1 + x2) + x3)}
```

11. A well-known programming exercise in many languages is to generate Hamming numbers, sometimes referred to as *regular numbers*. These are numbers that divide powers of 60 (the choice of that number goes back to the Babylonians who used 60 as a number base). Generate a sorted sequence of all Hamming numbers less than 1000. The key observation is that these numbers have only 2, 3, and 5 as prime factors.

5.6 *Solutions*

1. This function adds the squares of the elements in a list.

```
In[1]:=  elementsSquared[lis_] := Total[lis^2]

In[2]:=  elementsSquared[{1, 3, 5, 7, 9}]

Out[2]=  165
```

Using a pure function, this becomes:

```
In[3]:=  Function[lis, Total[lis^2]][{1, 3, 5, 7, 9}]

Out[3]=  165
```

or simply,

```
In[4]:= Total[#^2] &[{1, 3, 5, 7, 9}]

Out[4]= 165
```

2. To compute the distance between two points, use either `EuclideanDistance` or `Norm`.

```
In[5]:= pts = RandomReal[1, {4, 2}]

Out[5]= {{0.928713, 0.605799}, {0.479095, 0.758771},
          {0.860938, 0.239588}, {0.539415, 0.1972}}

In[6]:= Norm[pts[[1]] - pts[[2]]]

Out[6]= 0.474928

In[7]:= EuclideanDistance[pts[[1]], pts[[2]]]

Out[7]= 0.474928
```

Now we need the distance between every pair of points. So we first create the set of pairs.

```
In[8]:= pairs = Subsets[pts, {2}]

Out[8]= {{{0.928713, 0.605799}, {0.479095, 0.758771}},
          {{0.928713, 0.605799}, {0.860938, 0.239588}},
          {{0.928713, 0.605799}, {0.539415, 0.1972}},
          {{0.479095, 0.758771}, {0.860938, 0.239588}},
          {{0.479095, 0.758771}, {0.539415, 0.1972}},
          {{0.860938, 0.239588}, {0.539415, 0.1972}}}
```

Then we compute the distance between each pair and take the `Max`.

```
In[9]:= Apply[Norm[#1 - #2] &, pairs, {1}]

Out[9]= {0.474928, 0.37243, 0.564364, 0.64448, 0.564802, 0.324305}

In[10]:= Max[%]

Out[10]= 0.64448
```

Or, use `Outer` on the set of points directly, but not the need to get the level correct.

```
In[11]:= Max@Outer[Norm[#1 - #2] &, pts, pts, 1]

Out[11]= 0.64448
```

Now put it all together using a pure function in place of the distance function. The `diameter` function operates on lists of pairs of numbers, so we need to specify them in our pure function as `#1` and `#2`.

```
In[12]:= diameter[lis_] := Max[Apply[Norm[#1 - #2] &, Subsets[lis, {2}], {1}]]

In[13]:= diameter[pts]

Out[13]= 0.64448
```

`EuclideanDistance` is a bit faster here, but for large datasets, the difference is more pronounced.

```
In[14]:= Max[Apply[EuclideanDistance, Subsets[pts, {2}], {1}]]

Out[14]= 0.64448

In[15]:= pts = RandomReal[1, {1500, 2}];
         Max[Apply[Norm[#1 - #2] &, Subsets[pts, {2}], {1}]] // Timing

Out[16]= {6.86021, 1.38635}

In[17]:= Max[Apply[EuclideanDistance, Subsets[pts, {2}], {1}]] // Timing

Out[17]= {1.67334, 1.38635}
```

3. Pure functions are needed to replace both `addOne` and `CompositeQ`:

```
In[18]:= nextPrime[n_Integer /; n > 1] :=
          NestWhile[# + 1 &, n, Not[PrimeQ[#]] &]
```

Here is a quick check for correctness.

```
In[19]:= nextPrime[2^123] == NextPrime[2^123]

Out[19]= True
```

Compare timing with the built-in function.

```
In[20]:= Timing[nextPrime[2^2500];]

Out[20]= {0.342968, Null}

In[21]:= Timing[NextPrime[2^2500];]

Out[21]= {0.321423, Null}
```

4. This function is ideally written as an iteration.

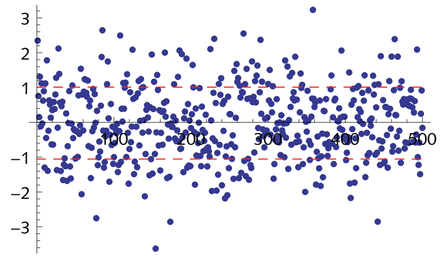
```
In[22]:= RepUnit[n_] := Nest[(10 # + 1) &, 1, n - 1]

In[23]:= RepUnit[7]

Out[23]= 1 111 111

In[24]:= Map[RepUnit[#] &, Range[12]]

Out[24]= {1, 11, 111, 1111, 11 111, 111 111, 1 111 111, 11 111 111,
          111 111 111, 1 111 111 111, 11 111 111 111, 111 111 111 111}
```

5. Here are some sample data taken from a normal distribution.

```
In[25]:= data = RandomVariate[NormalDistribution[0, 1], {500}];
```

Quickly visualize the data together with dashed lines drawn one standard deviation from the mean.

```
In[26]:= mean = Mean[data];
         sd = StandardDeviation[data];
         len = Length[data];
         ListPlot[data,
          Epilog → {Dashed, Red,
            Line[{{0, mean + sd}, {len, mean + sd}}],
            Line[{{0, mean - sd}, {len, mean - sd}}]}
         ]
```

Out[29]=


Select those data elements whose distance to the mean is less than one standard deviation.

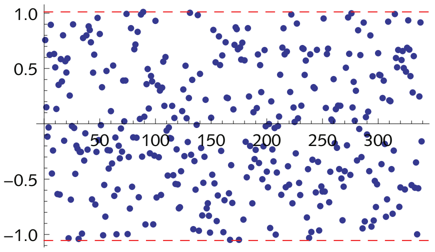
```
In[30]:= filtered = Select[data, (Abs[(# - mean)] < sd &)];
```

Here is a quick check that we get about the value we might expect (we would expect about 68% for normally distributed data).

In[31]:= $N\left[\frac{\texttt{Length[filtered]}}{\texttt{Length[data]}}\right]$

```
Out[31]= 0.678
```

```
In[32]:= ListPlot[filtered, PlotRange → All,
          Epilog → {Dashed, Red,
            Line[{{0, mean + sd}, {len, mean + sd}}],
            Line[{{0, mean - sd}, {len, mean - sd}}]}
         ]
```

Out[32]=


6. Using the list of step increments in the north, south, east, and west directions, this ten-step walk starts at the origin.

```
In[33]:= SeedRandom[0];
         NestList[
          #1 + RandomChoice[{{1, 0}, {-1, 0}, {0, 1}, {0, -1}}] &, {0, 0}, 10]

Out[34]= {{0, 0}, {0, 1}, {0, 2}, {-1, 2}, {-1, 1},
          {0, 1}, {1, 1}, {1, 2}, {2, 2}, {3, 2}, {3, 3}}
```

Except for the initial value, you can get the same result with `Accumulate` which generates cumulative sums.

```
In[35]:= SeedRandom[0];
         Accumulate[RandomChoice[{{1, 0}, {-1, 0}, {0, 1}, {0, -1}}, 10]]

Out[36]= {{0, 1}, {0, 2}, {-1, 2}, {-1, 1},
          {0, 1}, {1, 1}, {1, 2}, {2, 2}, {3, 2}, {3, 3}}
```

7. Here are the words from the built-in *Mathematica* dictionary.

```
In[37]:= words = DictionaryLookup[];
```

Here are those words that start with the letter *q*.

```
In[38]:= DictionaryLookup["q" ~~ __];
         RandomSample[%, 20]

Out[39]= {quadruples, quaffs, quark, quarantining, quahogs, quaff, quickie,
          quadrupled, quires, quit, quell, quints, quizzes, quadriplegia,
          quotation, quacking, quilter, queered, quintuple, quarterfinals}
```

And here are those words that start with the letter *q* and are of length 5. Note the need for `StringLength`, not `Length`.

```
In[40]:= Select[DictionaryLookup["q" ~~ __], StringLength[#] == 5 &]

Out[40]= {quack, quads, quaff, quail, quake, quaky, qualm,
          quark, quart, quash, quasi, quays, queen, queer,
          quell, quern, query, quest, queue, quick, quids, quiet,
          quiff, quill, quilt, quins, quint, quips, quire, quirk,
          quirt, quite, quits, quoin, quoit, quota, quote, quoth}
```

8. Using `Fold`, this pure function requires two arguments. The key is to start with an initial value of 0.

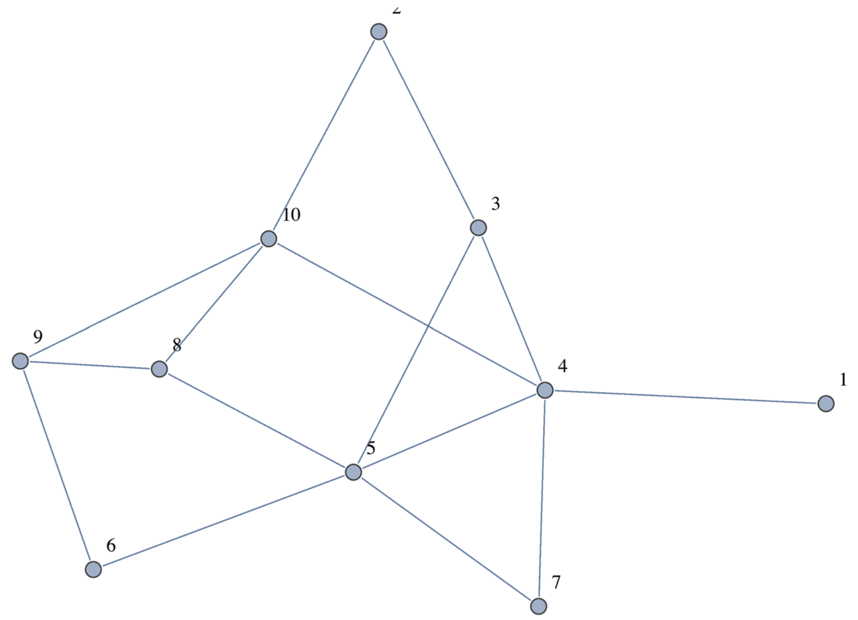
```
In[41]:= Horner[list_List, var_] := Fold[var #1 + #2 &, 0, list]

In[42]:= Horner[{a, b, c, d, e}, x]

Out[42]= e + x (d + x (c + x (b + a x)))

In[43]:= Expand[%]
```

Out[43]= $e + d\,x + c\,x^2 + b\,x^3 + a\,x^4$

9. Here is the prototype graph we will work with:

```
In[44]:= SeedRandom[16];
         gr = RandomGraph[{10, 15}, VertexLabels → "Name"]
```

Out[45]=



And here are its edges and its vertices:

```
In[46]:= EdgeList[gr]

Out[46]= {2 ⟷ 10, 2 ⟷ 3, 3 ⟷ 5, 4 ⟷ 5, 4 ⟷ 1, 4 ⟷ 7, 4 ⟷ 3,
          5 ⟷ 7, 5 ⟷ 8, 6 ⟷ 5, 9 ⟷ 8, 9 ⟷ 6, 10 ⟷ 9, 10 ⟷ 8, 10 ⟷ 4}

In[47]:= VertexList[gr]

Out[47]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Below are those edges from vertex 3 to any other vertex. In other words, this gives the adjacency list for vertex 3.

```
In[48]:= With[{u = 3},
          Select[VertexList[gr], (EdgeQ[gr, UndirectedEdge[u, #]] &)]
         ]

Out[48]= {2, 4, 5}
```

The case for directed graphs is similar. Here then is a function that returns the adjacency list for a given vertex u in graph `gr`.

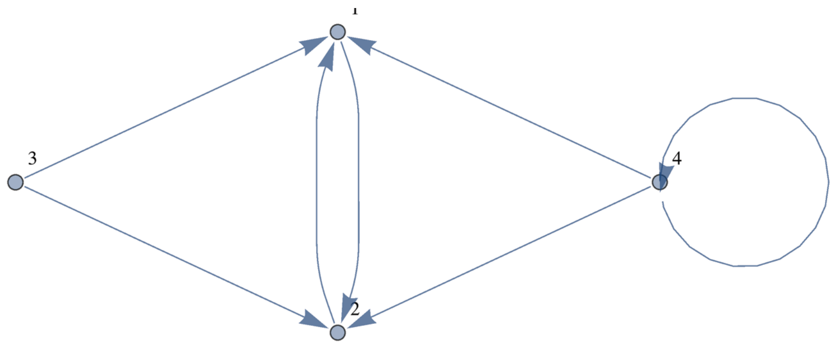
```
In[49]:= adjacencyList[gr_, u_] := If[DirectedGraphQ[gr],
            Select[VertexList[gr], EdgeQ[gr, DirectedEdge[u, #]] &],
            Select[VertexList[gr], EdgeQ[gr, UndirectedEdge[u, #]] &]
          ]
```

The adjacency structure is then given by mapping the above function across the vertex list.

```
In[50]:= AdjacencyStructure[gr_Graph] :=
          Map[{#, adjacencyList[gr, #]} &, VertexList[gr]]

In[51]:= AdjacencyStructure[gr]

Out[51]= {{1, {4}}, {2, {3, 10}}, {3, {2, 4, 5}}, {4, {1, 3, 5, 7, 10}},
          {5, {3, 4, 6, 7, 8}}, {6, {5, 9}}, {7, {4, 5}},
          {8, {5, 9, 10}}, {9, {6, 8, 10}}, {10, {2, 4, 8, 9}}}
```

Check that it works for a directed graph also.

```
In[52]:= gr2 = Graph[{1 ↔ 2, 2 ↔ 1, 3 ↔ 1, 3 ↔ 2, 4 ↔ 1, 4 ↔ 2, 4 ↔ 4},
           VertexLabels → "Name"]
```

Out[52]=


```
In[53]:= AdjacencyStructure[gr2]

Out[53]= {{1, {2}}, {2, {1}}, {3, {1, 2}}, {4, {1, 2, 4}}}
```

10. The key to solving this problem is thinking carefully about the initial value for `FoldList`.

```
In[54]:= FoldList[#1 + α (#2 - #1) &, x₁, {x₂, x₃}]

Out[54]= {x₁, x₁ + α (-x₁ + x₂), x₁ + α (-x₁ + x₂) + α (-x₁ - α (-x₁ + x₂) + x₃)}
```

If you were defining your own function, you would need to extract the first element of the (data) list as the initial value of `FoldList`.

```
In[55]:= expMovingAverage[lis_, α_] :=
          FoldList[#1 + α (#2 - #1) &, First[lis], Rest[lis]]

In[56]:= expMovingAverage[{a, b, c}, α]

Out[56]= {a, a + (-a + b) α, a + (-a + b) α + α (-a + c - (-a + b) α)}
```

11. A first, naive implementation will use the fact that the factors are all less than 6. Here are the factors for a single integer.

```
In[57]:= facs = FactorInteger[126]
Out[57]= {{2, 1}, {3, 2}, {7, 1}}
```

This extracts only the prime factors.

```
In[58]:= Map[First, facs]
Out[58]= {2, 3, 7}
```

In this case, they are not all less than 6.

```
In[59]:= Max[%] < 6
Out[59]= False
```

Putting these pieces together, here are the Hamming numbers less than 1000.

```
In[60]:= Select[Range[1000], Max[Map[First, FactorInteger[#]]] < 6 &]
Out[60]= {1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32,
          36, 40, 45, 48, 50, 54, 60, 64, 72, 75, 80, 81, 90, 96, 100, 108,
          120, 125, 128, 135, 144, 150, 160, 162, 180, 192, 200, 216, 225,
          240, 243, 250, 256, 270, 288, 300, 320, 324, 360, 375, 384, 400,
          405, 432, 450, 480, 486, 500, 512, 540, 576, 600, 625, 640, 648,
          675, 720, 729, 750, 768, 800, 810, 864, 900, 960, 972, 1000}
```

Factoring is slow for large integers and so this implementation does not scale well. This finds the 507 Hamming numbers less than 10^6.

```
In[61]:= With[{n = 10^6},
           Select[Range[n], Max[Map[First, FactorInteger[#]]] < 6 &]
          ]; // Timing
Out[61]= {7.80972, Null}
```

See Dijkstra (1981) for a different implementation that starts with h = {1}, then builds lists 2 h, 3 h, 5 h, merges these lists, and iterates.

```
In[62]:= HammingNumberList[n_] := Module[{lim},
           lim =
            If[n < 100, Ceiling[Log2[n]], Ceiling[Log2[n/(2 × 3 × 5)] Log2[n]]];
           Join[{1}, Take[Union @@ NestList[
               Union @@ Outer[Times, {2, 3, 5}, #] &, {2, 3, 5}, lim], n - 1]
           ]]

In[63]:= HammingNumber[n_] := Part[HammingNumberList[n], n]

In[64]:= HammingNumberList[20]
Out[64]= {1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36}
```

```
In[65]:= HammingNumber[1691] // Timing

Out[65]= {0.120575, 2 125 764 000}
```

This gives the one-millionth Hamming number.

```
In[66]:= HammingNumber[10^6] // Timing

Out[66]= {15.1734,
           519 312 780 448 388 736 089 589 843 750 000 000 000 000 000 000 000 000 000 
             000 000 000 000 000 000 000 000 000}
```

5.7 *Options and messages*

1. In Section 5.5 we developed a function `switchRows` that interchanged two rows in a matrix. Create a message for this function that is issued whenever a row index greater than the size of the matrix is used as an argument. For example,

```
In[1]:= mat = RandomInteger[{0, 9}, {4, 4}];
        MatrixForm[mat]
Out[2]//MatrixForm=
```

$$\begin{pmatrix} 3 & 5 & 1 & 8 \\ 5 & 9 & 7 & 4 \\ 5 & 0 & 7 & 1 \\ 4 & 2 & 3 & 0 \end{pmatrix}$$

```
In[3]:= switchRows[mat, {5, 2}]

        switchRows::badargs : The absolute value of the row indices
            5 and 2 in switchRows[mat,{5,2}] must be between 1 and 4, the size of the matrix.

Out[3]= {{3, 5, 1, 8}, {5, 9, 7, 4}, {5, 0, 7, 1}, {4, 2, 3, 0}}
```

You should also trap for a row index of 0.

```
In[4]:= switchRows[mat, {0, 2}]

        switchRows::badargs : The absolute value of the row indices
            0 and 2 in switchRows[mat,{0,2}] must be between 1 and 4, the size of the matrix.

Out[4]= {{3, 5, 1, 8}, {5, 9, 7, 4}, {5, 0, 7, 1}, {4, 2, 3, 0}}
```

2. Create an error message for `StemPlot`, developed in this section, so that an appropriate message is issued if the argument is not a list of numbers.

5.7 *Solutions*

1. The message will slot in the values of the row indices being passed to the function `switchRows`, as well as the length of the matrix, that is, the number of matrix rows.

```
In[1]:= switchRows::badargs =
          "The absolute value of the row indices `1` and
            `2` in switchRows[mat,`1`,`2`] must be
            between 1 and `3`, the size of the matrix.";
```

The message is issued if either of the row indices have absolute value greater than the length of the matrix or if either of these indices is equal to 0.

```
In[2]:= switchRows[mat_, {r1_Integer, r2_Integer}] :=
         Module[{lmat = mat, len = Length[mat]},
          If[Abs[r1] > len || Abs[r2] > len || r1 r2 == 0,
           Message[switchRows::badargs, r1, r2, len],
           lmat[[{r1, r2}]] = lmat[[{r2, r1}]]];
          lmat]

In[3]:= mat = RandomInteger[9, {4, 4}];
        MatrixForm[mat]
Out[4]//MatrixForm=
```

$$\begin{pmatrix} 0 & 0 & 8 & 5 \\ 4 & 6 & 9 & 9 \\ 0 & 0 & 0 & 3 \\ 2 & 7 & 0 & 8 \end{pmatrix}$$

```
In[5]:= switchRows[mat, {0, 4}]

  switchRows::badargs : The absolute value of the row indices
       0 and 4 in switchRows[mat,0,4] must be between 1 and 4, the size of the matrix.

Out[5]= {{0, 0, 8, 5}, {4, 6, 9, 9}, {0, 0, 0, 3}, {2, 7, 0, 8}}

In[6]:= switchRows[mat, {2, 8}]

  switchRows::badargs : The absolute value of the row indices
       2 and 8 in switchRows[mat,2,8] must be between 1 and 4, the size of the matrix.

Out[6]= {{0, 0, 8, 5}, {4, 6, 9, 9}, {0, 0, 0, 3}, {2, 7, 0, 8}}
```

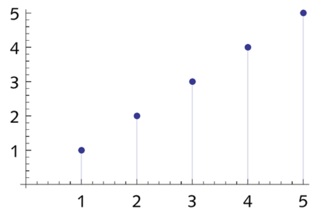
2. If the first argument is not a list containing numbers, then issue a message.

```
In[7]:= MatchQ[{1, 2, a}, {__?NumericQ}]

Out[7]= False
```

Here is the message:

```
In[8]:= StemPlot::badarg =
          "The first argument to StemPlot must be a list of numbers.";

In[9]:= Options[StemPlot] = Options[ListPlot];

In[10]:= StemPlot[lis_, opts : OptionsPattern[]] :=
          If[MatchQ[lis, {__?NumericQ}],
           ListPlot[lis, opts, Filling → Axis],
           Message[StemPlot::badarg]
          ]

In[11]:= StemPlot[4]

  StemPlot::badarg : The first argument to StemPlot must be a list of numbers.
```

```
In[12]:= StemPlot[{1, 2, c}]

   StemPlot::badarg : The first argument to StemPlot must be a list of numbers.

In[13]:= StemPlot[{1, 2, 3, 4, 5}]

Out[13]=
```



5.8 Examples and applications

1. Write a version of the function that computes Hamming distance by using `Count` to find the number of nonidentical pairs of corresponding numbers in two binary signals.
2. Write an implementation of Hamming distance using the `Total` function and then compare running times with the other versions discussed in this chapter.
3. Extend the survivor function developed in this section to a function of two arguments, so that `survivor[n, m]` returns the survivor starting from a list of *n* people and executing every *m*th person.
4. Create a function `median[lis]` that computes the median of a one-dimensional list. Create one rule for the case when *lis* has an odd number of elements and another rule for the case when the length of *lis* is even. In the latter case, the median is given by the average of the middle two elements of *lis*.
5. 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. Deconstruct these programs to see what they do and then reconstruct them as compound functions without any pure functions.

   a. Tally the frequencies with which distinct elements appear in a list.

```
In[1]:= tally[lis_] := Map[({#, Count[lis, #]}) &, Union[lis]]

In[2]:= tally[{a, a, b, b, b, a, c, c}]

Out[2]= {{a, 3}, {b, 3}, {c, 2}}

In[3]:= Tally[{a, a, b, b, b, a, c, c}]

Out[3]= {{a, 3}, {b, 3}, {c, 2}}
```

   b. Divide up a list such that the length of each part is given by the second argument.

```
In[4]:= split1[lis_, parts_] :=
          (Inner[Take[lis, {#1, #2}] &, Drop[#1, -1] + 1, Rest[#1], List] &)[
           FoldList[Plus, 0, parts]]

In[5]:= split1[Range[10], {2, 5, 0, 3}]

Out[5]= {{1, 2}, {3, 4, 5, 6, 7}, {}, {8, 9, 10}}
```

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

```
In[6]:= split2[lis_, parts_] := Map[Take[lis, #1 + {1, 0}] &,
          Partition[FoldList[Plus, 0, parts], 2, 1]]
```

6. In Section 4.2 we created a function `CountChange`[*lis*] that took a list of coins and, using transformation rules, returned the monetary value of that list of coins. Rewrite `CountChange` to use a purely functional approach. Consider using `Dot`, or `Inner`, or `Tally`.
7. Write a function that generates a one-dimensional off-lattice, random walk, that is, a walk with step positions any real number between −1 and 1. Then do the same for two- and three-dimensional off-lattice walks.
8. Extend the range of `ReplaceElement` developed in this section to accept a list of strings considered as nonnumeric matrix entries, each of which should be replaced by a column mean.
9. Extend the visualization of PPI networks from this section by coloring vertices according to the biological process in which they are involved. The built-in `ProteinData` contains this information, for example:

```
 In[7]:= ProteinData["KLKB1", "BiologicalProcesses"]

Out[7]= {BloodCoagulation, Fibrinolysis,
          InflammatoryResponse, Proteolysis}
```

10. Create a function `TruthTable`[*expr*, *vars*] that takes a logical expression such as $A \wedge B$ and outputs a truth table similar to those in Section 2.3. You can create a list of truth values using `Tuples`. For example,

```
 In[8]:= Tuples[{True, False}, 2]

Out[8]= {{True, True}, {True, False}, {False, True}, {False, False}}
```

You will also find it helpful to consider threading rules over the tuples using `MapThread` or `Thread`.

11. Given a list of expressions, *lis*, create a function `NearTo`[*lis*, *elem*, *n*] that returns all elements of *lis* that are exactly *n* positions away from *elem*. For example:

```
 In[9]:= chars = CharacterRange["a", "z"]

Out[9]= {a, b, c, d, e, f, g, h, i, j, k,
          l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}

In[10]:= NearTo[chars, "q", 3]

Out[10]= {{n}, {t}}
```

Write a second rule, `NearTo`[*lis*, *elem*, {*n*}] that returns all elements in *lis* that are within *n* positions of *elem*.

```
In[11]:= NearTo[chars, "q", {4}]

Out[11]= {{m, n, o, p, q, r, s, t, u}}
```

Finally, create you own distance function (`DistanceFunction`) and use it with the built-in `Nearest` to do the same computation.

Two useful functions for these tasks are `Position` and `Extract`. `Extract[`*expr*`, `*pos*`]` returns elements from *expr* whose positions *pos* are given by `Position`.

12. A *Smith number* is a composite number such that the sum of its digits is equal to the sum of the digits of its prime factors. For example, the prime factorization of 852 is $2^2 \cdot 3^1 \cdot 71^1$, and so the sum of the digits of its prime factors is $2 + 2 + 3 + 7 + 1 = 15$ which is equal to the sum of its digits, $8 + 5 + 2 = 15$. Write a program to find all Smith numbers less than 10 000.

5.8 Solutions

1. Here are two sample lists.

```
In[1]:=  l1 = {1, 0, 0, 1, 1};
         l2 = {0, 1, 0, 1, 0};
```

First, pair them.

```
In[3]:=  ll = Transpose[{l1, l2}]

Out[3]=  {{1, 0}, {0, 1}, {0, 0}, {1, 1}, {1, 0}}
```

Here is the conditional pattern that matches any pair where the two elements are *not* identical. The Hamming distance is the number of such nonidentical pairs.

```
In[4]:=  Count[ll, {p_, q_} /; p ≠ q]

Out[4]=  3
```

Finally, here is a function that puts this all together.

```
In[5]:=  HammingDistance3[lis1_List, lis2_List] :=
          Count[Transpose[{lis1, lis2}], {p_, q_} /; p ≠ q]

In[6]:=  HammingDistance3[l1, l2]

Out[6]=  3
```

The running times of this version of `HammingDistance` are quite a bit slower than those where we used bit operators. This is due to additional computation (`Transpose`, `Length`) and the use of pattern matching and comparisons at every step.

```
In[7]:=  HammingDistance2[lis1_, lis2_] := Total[BitXor[lis1, lis2]]
In[8]:=  data1 = RandomInteger[1, {10^6}];
In[9]:=  data2 = RandomInteger[1, {10^6}];

In[10]:= Timing[HammingDistance2[data1, data2]]

Out[10]= {0.011924, 501 049}
```

```
In[11]:=  Timing[HammingDistance3[data1, data2]]

Out[11]=  {0.718988, 501 049}
```

2. Using `Total`, which simply gives the sum of the elements in a list, Hamming distance can be computed as follows:

```
In[12]:=  HammingDistance4[lis1_, lis2_] := Total[Mod[lis1 + lis2, 2]]
```

Timing tests show that the implementation with `Total` is quite a bit more efficient than the previous versions, although still slower than the version that uses bit operators.

```
In[13]:=  sig1 = RandomInteger[1, {10^6}];

In[14]:=  sig2 = RandomInteger[1, {10^6}];

In[15]:=  HammingDistance1[lis1_, lis2_] :=
            Count[MapThread[SameQ, {lis1, lis2}], False]

In[16]:=  Map[{#, Timing[#[sig1, sig2]]} &, {HammingDistance1,
             HammingDistance2, HammingDistance3, HammingDistance4}] // Grid

          HammingDistance1  {0.500941, 499 991}
          HammingDistance2  {0.007204, 499 991}
Out[16]=  HammingDistance3  {0.695777, 499 991}
          HammingDistance4  {0.02221, 499 991}
```

3. Just one change is needed here: add a second argument to `RotateLeft` that specifies the number of positions to rotate. We have used `NestList` to display the intermediate steps.

```
In[17]:=  survivor[n_, m_] :=
            NestList[Rest[RotateLeft[#, m - 1]] &, Range[n], n - 1]

In[18]:=  survivor[11, 3]

Out[18]=  {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}, {4, 5, 6, 7, 8, 9, 10, 11, 1, 2},
           {7, 8, 9, 10, 11, 1, 2, 4, 5}, {10, 11, 1, 2, 4, 5, 7, 8},
           {2, 4, 5, 7, 8, 10, 11}, {7, 8, 10, 11, 2, 4},
           {11, 2, 4, 7, 8}, {7, 8, 11, 2}, {2, 7, 8}, {2, 7}, {7}}
```

4. The median of a list containing an odd number of elements is the middle element of the sorted list.

```
In[19]:=  median[lis_List /; OddQ[Length[lis]]] :=
            Part[Sort[lis], Ceiling[Length[lis] / 2]]
```

When the list has an even number of elements, take the mean of the middle two.

```
In[20]:=  median[lis_List /; EvenQ[Length[lis]]] :=
            Module[{len = Length[lis] / 2},
             Mean[Part[Sort[lis], len ;; len + 1]]
            ]
```

Check the two cases – an even number of elements, and an odd number of elements. Then compare with the built-in `Median`.

```
In[21]:=  dataE = RandomInteger[10 000, 100 000];
```

```
In[22]:= dataO = RandomInteger[10 000, 100 001];

In[23]:= median[dataE] // Timing

Out[23]= {0.019613, 4977}

In[24]:= Median[dataE] // Timing

Out[24]= {0.018717, 4977}

In[25]:= median[dataO] // Timing

Out[25]= {0.019516, 4962}

In[26]:= Median[dataO] // Timing

Out[26]= {0.019463, 4962}
```

The two rules given here should be more careful about the input, using pattern matching to insure that these rules only apply to one-dimensional lists. The following modifications handle that more robustly.

```
In[27]:= Clear[median]

In[28]:= median[lis : {__} /; OddQ[Length[lis]]] :=
          Part[Sort[lis], Ceiling[Length[lis] / 2]]

In[29]:= median[lis : {__} /; EvenQ[Length[lis]]] :=
          Module[{len = Length[lis] / 2},
           Mean[Part[Sort[lis], len ;; len + 1]]]
```

5.

a. The pure function `{#, Count[lis, #]} &` is replaced with the function `pair` below.

```
In[30]:= frequencies[lis_] := Module[{pair},
           pair[x_] := {x, Count[lis, x]};
           Map[pair, Union[lis]]]

In[31]:= frequencies[{a, a, b, b, b, a, c, c}]

Out[31]= {{a, 3}, {b, 3}, {c, 2}}
```

b.

```
In[32]:= split1[lis_, parts_] := Module[{lis1, lis2},
           lis1[y_, z_] := Take[lis, {y, z}];
           lis2[x_] := Inner[lis1, Drop[x, -1] + 1, Rest[x], List];
           lis2[FoldList[Plus, 0, parts]]]

In[33]:= split1[Range[10], {2, 5, 0, 3}]

Out[33]= {{1, 2}, {3, 4, 5, 6, 7}, {}, {8, 9, 10}}
```

```
In[34]:= split2[lis_, parts_] := Module[{lis1},
           lis1[x_] := Take[lis, x + {1, 0}];
           Map[lis1, Partition[FoldList[Plus, 0, parts], 2, 1]]]

In[35]:= split2[Range[10], {2, 5, 0, 3}]

Out[35]= {{1, 2}, {3, 4, 5, 6, 7}, {}, {8, 9, 10}}
```

c.

```
In[36]:= lotto1[lis_, n_] := Module[{lis1, lis2, lis3}, lis1[x_] :=
            Flatten[Rest[MapThread[Complement, {RotateRight[x], x}, 1]]];
           lis2[y_] := Delete[y, RandomInteger[{1, Length[y]}]];
           lis3[z_] := NestList[lis2, z, n];
           lis1[lis3[lis]]]

In[37]:= lotto1[Range[10], 5]

Out[37]= {4, 3, 6, 8, 9}

In[38]:= lotto2[lis_, n_] := Take[Transpose[
             Sort[Transpose[{RandomReal[1, {Length[lis]}], lis}]]][[2]], n]

In[39]:= lotto2[Range[10], 5]

Out[39]= {3, 5, 4, 7, 10}
```

6. Here is a list of coins (modify for other currencies).

```
In[40]:= coins = {p, p, q, n, d, d, p, q, q, p};
```

First count the occurrences of each.

```
In[41]:= Map[Count[coins, #] &, {p, n, d, q}]

Out[41]= {4, 1, 2, 3}
```

Then a dot product of this count vector with a value vector does the trick.

```
In[42]:= %.{.01, .05, .10, .25}

Out[42]= 1.04

In[43]:= CountChange[lis_] :=
          Dot[Map[Count[lis, #] &, {p, n, d, q}], {.01, .05, .10, .25}]

In[44]:= CountChange[coins]

Out[44]= 1.04

In[45]:= CountChange2[lis_] :=
          Inner[Times, Map[Count[lis, #] &, {p, n, d, q}],
           {.01, .05, .10, .25}, Plus]
```

```
In[46]:=  CountChange2[coins]

Out[46]=  1.04
```

And here is a rule-based approach.

```
In[47]:=  Tally[coins] /. {d → .10, n → .05, p → .01, q → .25}

Out[47]=  {{0.01, 4}, {0.25, 3}, {0.05, 1}, {0.1, 2}}

In[48]:=  Total[Apply[Times, %, {1}]]

Out[48]=  1.04

In[49]:=  CountChange3[lis_] := Module[{freq},
            freq = Tally[lis] /. {p → .01, n → .05, d → .10, q → .25};
            Total[Apply[Times, freq, {1}]]]

In[50]:=  CountChange3[coins]

Out[50]=  1.04
```

7. The two-dimensional implementation insures steps of unit length by mapping the pure function `{Cos[#], Sin[#]} &` over the angles.

```
In[51]:=  walk1DOffLattice[steps_] := Accumulate[RandomReal[{-1, 1}, steps]]

In[52]:=  walk2DOffLattice[steps_] :=
           Accumulate[Map[{Cos[#], Sin[#]} &, RandomReal[{0, 2 π}, steps]]]
```

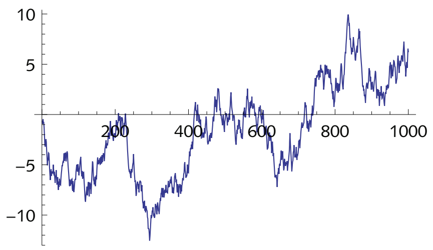
The three-dimensional walk requires two angles, θ in the interval $[0, 2\pi)$ and ϕ in the interval $[-1, 1]$. See Section 13.1 for a discussion of the three-dimensional off-lattice walk.

```
In[53]:=  walk3DOffLattice[t_] := Accumulate[
            Table[Function[{θ, ϕ}, {Cos[θ] √(1 - ϕ²), Sin[θ] √(1 - ϕ²), ϕ}] @@
              {RandomReal[{0, 2 π}], RandomReal[{-1, 1}]}, {t}]]
```

With the one-dimensional walk, the vertical axis gives displacement from the origin and the horizontal axis shows the number of steps.

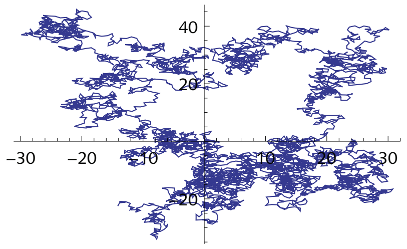
```
In[54]:=  ListLinePlot[walk1DOffLattice[1000]]
```

Out[54]=



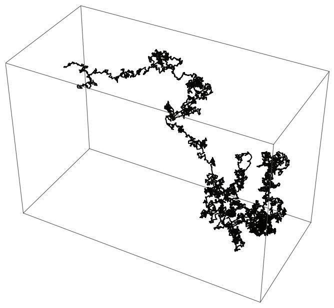
```
In[55]:= ListLinePlot[walk2DOffLattice[5000]]
```

Out[55]=



```
In[56]:= Graphics3D[Line[walk3DOffLattice[5000]]]
```

Out[56]=



8. Column 4 of this matrix contains several different nonnumeric values.

```
In[57]:= mat3 = {{0.796495, "N/A", 0.070125, "nan", 0.806554},
             {"nn", -0.100365, 0.992736, -0.320560, -0.0805351},
             {0.473571, 0.460741, 0.030060, -0.412400, 0.788522},
             {0.614974, -0.503201, 0.615744, 0.966053, -0.011776},
             {-0.828415, 0.035514, 0.8911617, "N/A", -0.453926}};
         MatrixForm[col4 = mat3[[All, 4]]]
```

Out[58]//MatrixForm=

$$\begin{pmatrix} \text{nan} \\ -0.32056 \\ -0.4124 \\ 0.966053 \\ \text{N/A} \end{pmatrix}$$

To pattern match on either `"N/A"` or `"nan"`, use `Alternatives` (|).

```
In[59]:= col4 /. "N/A" | "nan" → Mean[Cases[mat3[[All, 4]], _?NumberQ]] //
          MatrixForm
```

Out[59]//MatrixForm=

$$\begin{pmatrix} 0.0776977 \\ -0.32056 \\ -0.4124 \\ 0.966053 \\ 0.0776977 \end{pmatrix}$$

Convert the list of strings to a set of alternatives.

```
In[60]:= Apply[Alternatives, {"N/A", "nan", "nn"}]

Out[60]= N/A | nan | nn
```

Here is a third set of definitions, including a new rule for `ReplaceElement` where the second argument is a list of strings. And another rule for `ReplaceElement` accommodates the new argument structure of `colMean`.

```
In[61]:= colMean[col_, {strings___String}] := col /.
           Apply[Alternatives, {strings}] → Mean[Cases[col, _?NumberQ]]

In[62]:= ReplaceElement[mat_, {strings__}] :=
          Transpose[Map[colMean[#, {strings}] &, Transpose[mat]]]

In[63]:= ReplaceElement[mat3, {"N/A", "nan", "nn"}] // MatrixForm

Out[63]//MatrixForm=
```

$$\begin{pmatrix} 0.796495 & -0.0268277 & 0.070125 & 0.0776977 & 0.806554 \\ 0.264156 & -0.100365 & 0.992736 & -0.32056 & -0.0805351 \\ 0.473571 & 0.460741 & 0.03006 & -0.4124 & 0.788522 \\ 0.614974 & -0.503201 & 0.615744 & 0.966053 & -0.011776 \\ -0.828415 & 0.035514 & 0.891162 & 0.0776977 & -0.453926 \end{pmatrix}$$

9. Start with a prototype logical expression.

```
In[64]:= Clear[A, B]

In[65]:= expr = (A || B) ⇒ C;

In[66]:= vars = {A, B, C};
```

List all the possible truth value assignments for the variables.

```
In[67]:= tuples = Tuples[{True, False}, Length[vars]]

Out[67]= {{True, True, True}, {True, True, False}, {True, False, True},
          {True, False, False}, {False, True, True}, {False, True, False},
          {False, False, True}, {False, False, False}}
```

Next, create a list of rules, associating each of the triples of truth values with a triple of variables.

```
In[68]:= rules = Map[Thread[vars → #] &, tuples]

Out[68]= {{A → True, B → True, C → True}, {A → True, B → True, C → False},
          {A → True, B → False, C → True}, {A → True, B → False, C → False},
          {A → False, B → True, C → True}, {A → False, B → True, C → False},
          {A → False, B → False, C → True}, {A → False, B → False, C → False}}
```

Replace the logical expression with each set of rules.

```
In[69]:= expr /. rules

Out[69]= {True, False, True, False, True, False, True, True}
```

Put these last values at the end of each "row" of the tuples.

```
In[70]:= table = Transpose@Join[Transpose[tuples], {expr /. rules}]

Out[70]= {{True, True, True, True}, {True, True, False, False},
          {True, False, True, True}, {True, False, False, False},
          {False, True, True, True}, {False, True, False, False},
          {False, False, True, True}, {False, False, False, True}}
```

Create a header for `table`.

```
In[71]:= head = Append[vars, TraditionalForm[expr]]

Out[71]= {A, B, C, A ⋁ B ⇒ C}
```

Prepend `head` to `table`.

```
In[72]:= Prepend[table, head]

Out[72]= {{A, B, C, A ⋁ B ⇒ C},
          {True, True, True, True}, {True, True, False, False},
          {True, False, True, True}, {True, False, False, False},
          {False, True, True, True}, {False, True, False, False},
          {False, False, True, True}, {False, False, False, True}}
```

Pour into a grid.

```
In[73]:= Grid[Prepend[table, head]]

            A       B       C     A ⋁ B ⇒ C
          True    True    True     True
          True    True    False    False
          True    False   True     True
Out[73]=  True    False   False    False
          False   True    True     True
          False   True    False    False
          False   False   True     True
          False   False   False    True
```

Replace `True` with `"T"` and `False` with `"F"`.

```
In[74]:= Grid[Prepend[table /. {True → "T", False → "F"}, head]]

          A  B  C  A ⋁ B ⇒ C
          T  T  T      T
          T  T  F      F
          T  F  T      T
Out[74]=  T  F  F      F
          F  T  T      T
          F  T  F      F
          F  F  T      T
          F  F  F      T
```

Add formatting via options to `Grid`.

```
In[75]:= Grid[Prepend[table /. {True → "T", False → "F"}, head],
          Dividers → {{1 → Black, -1 → Black, -2 → LightGray},
            {1 → Black, 2 → LightGray, -1 → Black}},
          BaseStyle → {FontFamily → "Times"}]
```

Out[75]=

A	B	C	$A \vee B \Rightarrow C$
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	T

Put the pieces together.

```
In[76]:= TruthTable[expr_, vars_] :=
          Module[{len = Length[vars], tuples, rules, table, head},
           tuples = Tuples[{True, False}, len];
           rules = Thread[vars → #1] & /@ tuples;
           table = Transpose@Join[Transpose[tuples], {expr /. rules}];
           head = Append[vars, TraditionalForm[expr]];
           Grid[Prepend[table /. {True → "T", False → "F"}, head],
            Dividers → {{1 → {Thin, Black}, -1 → {Thin, Black},
                -2 → {Thin, LightGray}}, {1 → {Thin, Black},
                2 → {Thin, LightGray}, -1 → {Thin, Black}}},
            BaseStyle → {FontFamily → "Times"}]]
```

```
In[77]:= TruthTable[A ⋀ B ⇒ ¬ C, {A, B, C}]
```

Out[77]=

A	B	C	$A \wedge B \Rightarrow \neg C$
T	T	T	F
T	T	F	T
T	F	T	T
T	F	F	T
F	T	T	T
F	T	F	T
F	F	T	T
F	F	F	T

10. `Position`[*lis*, *elem*] returns a list of positions at which *elem* occurs in *lis*. `Extract`[*lis*, *pos*] returns those elements whose positions are specified by `Position`.

```
In[78]:= NearTo[lis_List, elem_, n_] :=
          Module[{pos = Position[lis, elem]}, Extract[lis, {pos - n, pos + n}]]

In[79]:= NearTo[lis_List, elem_, {n_}] := Module[
           {pos = Position[lis, elem]}, Extract[lis, Range[pos - n, pos + n]]]

In[80]:= chars = CharacterRange["a", "z"];

In[81]:= NearTo[chars, "q", 3]

Out[81]= {{n}, {t}}

In[82]:= NearTo[chars, "q", {4}]

Out[82]= {{m, n, o, p, q, r, s, t, u}}
```

The key to writing the distance function is to observe that it must be a function of two variables and return a numeric value (the distance metric). We are finding the difference of the positions of a target element in the list with the element in question, `y` and `x`, respectively in the pure function. The use of `[[1, 1]]` is to strip off extra braces returned by `Position`.

```
In[83]:= NearToN[lis_, elem_, n_] :=
          Nearest[lis, elem, {2 n + 1, n}, DistanceFunction → Function[
             {x, y}, Abs[(Position[lis, y] - Position[lis, x])[[1, 1]]]]]

In[84]:= NearToN[chars, "q", 4]

Out[84]= {q, p, r, o, s, n, t, m, u}
```

11. Rather than try to incorporate all the conditions into one rule, it is cleaner and more efficient to write separate rules for the cases where the input is prime or less than or equal to one.

```
In[85]:= SmithNumberQ[n_ /; n ≤ 1] := False

In[86]:= SmithNumberQ[n_?PrimeQ] := False
```

Given the factorization of a number, separate the prime bases from their exponents using `Transpose`.

```
In[87]:= lis = FactorInteger[852]

Out[87]= {{2, 2}, {3, 1}, {71, 1}}

In[88]:= Transpose[lis]

Out[88]= {{2, 3, 71}, {2, 1, 1}}
```

This multiplies the integer digits of each base by its multiplicity.

```
In[89]:= MapThread[IntegerDigits[#1] #2 &, %]

Out[89]= {{4}, {3}, {7, 1}}
```

Here is the sum.

```
In[90]:= Total[Flatten[%]]

Out[90]= 15
```

Check that it equals the sum of the digits of the original number:

```
In[91]:= Total[IntegerDigits[852]]

Out[91]= 15
```

This puts the pieces together for the general rule.

```
In[92]:= SmithNumberQ[n_] := With[{lis = FactorInteger[n]},
           Total[
             Flatten[MapThread[IntegerDigits[#1] #2 &, Transpose[lis]]]] ==
            Total[IntegerDigits[n]]]
```

Here are the Smith numbers less than 100.

```
In[93]:= Select[Range[100], SmithNumberQ]

Out[93]= {4, 22, 27, 58, 85, 94}
```

There are 376 Smith numbers less than 10000.

```
In[94]:= Select[Range[10^4], SmithNumberQ] // Length

Out[94]= 376
```

As an interesting aside, you can also generate Smith numbers using rep units (see Exercise 5 in Section 5.6). For example, multiply any prime repunit by a suitable factor, e.g., 1540. For details of the relationship between repunits and Smith numbers, see Hoffman (1999).

```
In[95]:= RepUnit[n_] := Nest[(10 # + 1) &, 1, n - 1]

In[96]:= PrimeQ[RepUnit[19]]

Out[96]= True

In[97]:= SmithNumberQ[1540 RepUnit[23]]

Out[97]= True
```

6
Procedural programming

6.1 *Loops and iteration*

1. Compare the use of a `Do` loop with using the function `Nest` (see Section 5.3). In particular, compute the square root of 2 using `Nest`.
2. `Do` is closely related to `Table`, the main difference being that `Do` does not return any value, whereas `Table` does. Use `Table` instead of `Do` to rewrite one of the `findRoot` functions given in this section. Compare the efficiency of the two approaches.
3. Compute Fibonacci numbers iteratively. Fibonacci numbers consist of the sequence 1, 1, 2, 3, 5, 8, 13, …, where, after the first two 1s, each Fibonacci number is the sum of the previous two numbers in the sequence. You will need to have two variables, say `this` and `prev`, giving the two most recent Fibonacci numbers, so that after the *i*th iteration, `this` and `prev` have the values F_i and F_{i-1}, respectively.
4. One additional improvement can be made to the `findRoot` program developed in this section. Notice that the derivative of the function `fun` is recomputed each time through the loop. This is quite inefficient. Rewrite `findRoot` so that the derivative is computed only once and that result is used in the body of the loop.
5. Another termination criterion for root-finding is to stop when $| x_i - x_{i+1} | < \epsilon$, that is, when two successive estimates are very close. The idea is that if you are not getting much improvement, you must be very near the root. The difficulty in programming this is that you need to remember the *two* most recent estimates computed. (It is similar to computing Fibonacci numbers iteratively, as in Exercise 3.) Program `findRoot` this way.
6. The built-in `FindRoot` function is set up so that you can monitor intermediate computations using the option `EvaluationMonitor` and `Reap` and `Sow`. For example, the following sows the values of x and $f(x)$ and when `FindRoot` is done, `Reap` displays the sown expressions.

```
In[1]:=  f[x_] := x^2 - 2

In[2]:=  Reap[
          FindRoot[f[x], {x, 1}, EvaluationMonitor :> Sow[{x, f[x]}]]
         ]

Out[2]=  {{x → 1.41421}, {{{1., -1.}, {1.5, 0.25},
            {1.41667, 0.00694444}, {1.41422, 6.0073 × 10^-6},
            {1.41421, 4.51061 × 10^-12}, {1.41421, 4.44089 × 10^-16}}}}
```

Modify each of the versions of `findRoot` presented in the text that uses a `Do` or `While` loop to produce a similar output to that above.

7. To guard against starting with a poor choice of initial value, modify your solution to the previous exercise to take, as an argument, a *list* of initial values, and simultaneously compute approximations for each until one converges; then return that one.
8. The *bisection method* is quite useful for finding roots of functions. If a continuous function $f(x)$ is such that $f(a) < 0$ and $f(b) > 0$ for two real numbers a and b, then, as a consequence of the Intermediate Value Theorem of calculus, a root of f must occur between a and b. If f is now evaluated at the midpoint of a and b, and if $f(a+b)/2 < 0$, then the root must occur between $(a+b)/2$ and b; if not, then it occurs between a and $(a+b)/2$. This bisection can be repeated until a root is found to a specified tolerance.

   Define `bisect[f, {x, a, b}, ϵ]` to compute a root of f, within ϵ, using the bisection method. You should give it two initial values a and b and assume that $f(a) \cdot f(b) < 0$, that is, $f(a)$ and $f(b)$ differ in sign.
9. Using a `While` loop, write a function `gcd[m, n]` that computes the greatest common divisor (gcd) of m and n. The Euclidean algorithm for computing the gcd of two positive integers m and n, sets $m = n$ and $n = m \bmod n$. It iterates this process until $n = 0$, at which point the gcd of m and n is left in the value of m.
10. Create a procedural definition for each of the following functions. For each function, create a definition using a `Do` loop and another using `Table`.

    For example, the following function first creates an array consisting of 0s of the same dimension as `mat`. Then inside the `Do` loop it assigns the element in position `{j, i}` in `mat` to position `{i, j}` in `matA`, effectively performing a transpose operation. Finally, it returns `matA`, since the `Do` loop itself does not return a value.

```
In[3]:= transposeDo[mat_] :=
         Module[{matA, rows = Length[mat], cols = Length[mat[[1]]], i, j},
          matA = ConstantArray[0, {rows, cols}];
          Do[matA⟦i, j⟧ = mat⟦j, i⟧,
           {i, 1, rows},
           {j, 1, cols}];
          matA]

In[4]:= mat1 = {{a, b, c}, {d, e, f}, {g, h, i}};

In[5]:= MatrixForm[mat1]
Out[5]//MatrixForm=
```

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

```
In[6]:= MatrixForm[transposeDo[mat1]]
Out[6]//MatrixForm=
```

$$\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

This same computation could be performed with a *structured iteration* using `Table`.

```
In[7]:= transposeTable[mat_?MatrixQ] := Module[{matA, rows, cols},
          {rows, cols} = Dimensions[mat];
          matA = ConstantArray[0, {rows, cols}];
          Table[matA[[i, j]] = mat[[j, i]], {i, rows}, {j, cols}]
         ]

In[8]:= transposeTable[mat1] // MatrixForm
Out[8]//MatrixForm=
```

$$\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

a. Create the function `reverse[`*vec*`]` that reverses the elements in the list *vec*.

b. Create a function `rotateRight[`*vec*`,` *n*`]`, where *vec* is a vector and *n* is a (positive or negative) integer.

c. Create a procedural implementation of `rotateRows`, which could be defined in this functional way:

```
In[9]:= rotateRows[mat_] :=
         Map[rotateRight[mat[[#]], # - 1] &, Range[1, Length[mat]]]
```

That is, it rotates the *i*th row of `mat` by $i - 1$ places to the right.

d. Create a procedural function `rotateRowsByS`, which could be defined in this functional way:

```
In[10]:= rotateRowsByS[mat_, S_] /; Length[mat] == Length[S] :=
          Map[(rotateRight[mat[[#1]], S[[#1]]] &), Range[1, Length[mat]]]
```

That is, it rotates the *i*th row of `matA` by the amount `S[[`*i*`]]`.

e. Create a function `pick`$[lis_a, \ lis_b]$, where lis_a and lis_b are lists of equal length, and lis_b contains only Boolean values (`False` and `True`). This function selects those elements from lis_a corresponding to `True` in lis_b. For example, the result of the following should be `{a, b, e}`.

```
pick[{a, b, c, d, e}, {True, True, False, False, True}]
```

6.1 Solutions

1. To compute the square root of a number *r*, iterate the following expression.

```
In[1]:= fun[x_] := x^2 - r;
        Simplify[x - fun[x]/fun'[x]]
```

$$\text{Out[2]=}\quad \frac{r + x^2}{2\,x}$$

This can be written as a pure function, with a second argument giving the initial guess. Here we iterate ten times, starting with a high-precision initial value, 2.0 to 30-digit precision.

```
In[3]:=  nestSqrt[r_, init_] := Nest[(r + #^2)/(2 #) &, init, 10]

In[4]:=  nestSqrt[2, N[2, 30]]

Out[4]=  1.41421356237309504880168872
```

2. Here is a first basic attempt to replace the `Do` loop with `Table`.

```
In[5]:=  f[x_] := x^2 - 2

In[6]:=  a = 2;
         Table[a = N[a - f[a]/f′[a]], {10}]

Out[7]=  {1.5, 1.41667, 1.41422, 1.41421, 1.41421,
          1.41421, 1.41421, 1.41421, 1.41421, 1.41421}

In[8]:=  findRoot[fun_Symbol, {var_, init_}, iter_ : 10] :=
          Module[{xi = init},
           Table[xi = N[xi - fun[xi]/fun′[xi]], {iter}];
           {var → xi}]

In[9]:=  findRoot[f, {x, 2}]

Out[9]=  {x → 1.41421}
```

This runs the iteration only three times.

```
In[10]:=  findRoot[f, {x, 2}, 3]

Out[10]=  {x → 1.41422}
```

3. Note that this version of the Fibonacci function is much more efficient than the simple recursive version given in Chapter 7, and is closer to the version there that uses dynamic programming.

```
In[11]:=  fib[n_] := Module[{prev = 0, this = 1, next},
            Do[next = prev + this;
             prev = this;
             this = next,
             {n}];
            prev]

In[12]:=  Table[fib[i], {i, 1, 10}]

Out[12]=  {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

Actually, this code can be simplified a bit by using parallel assignments.

```
In[13]:=  fib2[n_] := Module[{f1 = 0, f2 = 1},
            Do[{f1, f2} = {f2, f1 + f2},
             {n - 1}];
            f2]
In[14]:= Table[fib2[i], {i, 1, 10}]
Out[14]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

Both of these implementations are quite fast and avoid the deep recursion of the classical definition.

```
In[15]:= {Timing[fib[100 000];], Timing[fib2[100 000];]}
Out[15]= {{0.22523, Null}, {0.183665, Null}}
```

4. We compute the derivative `df` inside the `Module` and then use that throughout the body of the function.

```
In[16]:= Clear[findRoot]
In[17]:= findRoot[fun_, {var_, init_}, ϵ_] :=
          Module[{xi = init, funxi = fun[init], df = fun'},
           While[Abs[funxi] > ϵ,
            xi = N[xi - funxi/df[xi]];
            funxi = fun[xi]];
           {var → xi}]
In[18]:= f[x_] := x^2 - 2
In[19]:= findRoot[f, {x, 10}, 0.0001]
```

5. The variable `b` is the current approximation, and the variable `a` is the previous approximation.

```
In[16]:= findRoot[fun_, {var_, init_}, ϵ_] :=
          Module[{a = init, b = fun[init]},
           While[Abs[b - a] > ϵ,
            a = b;
            b = N[b - fun[b]/fun′[b]]];
           {var → b}]
In[17]:= f[x_] := x^2 - 50
In[18]:= findRoot[f, {x, 10}, 0.0001]
Out[18]= {x → 7.07107}
```

6. This solution is based on the solution to Exercise 4 above.

```
In[19]:= findRootList[fun_, init_, ϵ_] :=
          Module[{a = init, b, solns = {init}},
           b = N[a - fun[a]/fun'[a]];
           While[Abs[b - a] > ϵ,
            a = b;
            b = b - fun[b]/fun'[b];
            AppendTo[solns, b]];
           solns]

In[20]:= f[x_] := x^2 - 2

In[21]:= findRootList[f, 1, 10^-6]

Out[21]= {1, 1.41667, 1.41422, 1.41421, 1.41421}
```

There is a numerical issue here that may not be apparent at first. If you were to provide a value for ϵ that is smaller than `1 / MachinePrecision`, this function will have trouble satisfying the test. In fact, this is true for all of the implementations of Newton's method in this chapter. These issues are explored and resolved in Section 8.4.

7. Based on a previous version of `findRoot`, the following adds multiple initial values.

```
In[22]:= findRootList[fun_, inits_List, ϵ_] := Module[{a = inits},
           While[Min[Abs[Map[fun, a]]] > ϵ,
            a = Map[N[# - fun[#]/fun'[#]] &, a]];
           Select[a, Min[Abs[Map[fun, a]]] == Abs[fun[#]] &]]

In[23]:= findRootList[(#^2 - 50) &, {-10, 1, 10}, .001]

Out[23]= {-7.07108, 7.07108}
```

8. A bit of variable swapping is needed here depending on whether or not a sign change occurs.

```
In[24]:= bisect[f_, {var_, a_, b_}, ϵ_] :=
          Module[{midpt = N[(a + b)/2], low = a, high = b},
           While[Abs[f[midpt]] > ϵ,
            If[Sign[f[low]] == Sign[f[midpt]], low = midpt, high = midpt];
            midpt = (low + high)/2];
```

```
        {var → midpt}]
```

```
In[25]:= f[x_] := x^2 - 2
         bisect[f, {x, 0, 2}, .0001]

Out[26]= {x → 1.41418}
```

9. This is a direct implementation of the Euclidean algorithm.

```
In[27]:= gcd[m_, n_] := Module[{a = m, b = n, tmpa},
           While[b > 0,
            tmpa = a;
            a = b;
            b = Mod[tmpa, b]];
           a]

In[28]:= With[{m = 12 782, n = 5 531 207},
          gcd[m, n]]

Out[28]= 11
```

You can avoid the need for the temporary variable `tmpa` by performing a parallel assignment as in the following function. In addition, some argument checking insures that *m* and *n* are integers.

```
In[29]:= gcd[m_Integer, n_Integer] := Module[{a = m, b = n},
           While[b > 0,
            {a, b} = {b, Mod[a, b]}];
           a]

In[30]:= With[{m = 12 782, n = 5 531 207},
          gcd[m, n]]

Out[30]= 11
```

10. Each solution mirrors that of the transpose example in the exercise.

a. Create a list `vecA` of zeros, then use a `Do` loop to set `vecA`⟦*i*⟧ to `vec`⟦*n* − *i*⟧, where *n* is the length of `vec`.

```
In[31]:= Clear[reverse, a, b, c, d, e]

In[32]:= reverse[vec_] := Module[{vecA, n = Length[vec]},
           vecA = ConstantArray[0, {n}];
           Do[vecA[[i]] = vec[[n - i + 1]],
            {i, 1, n}];
           vecA]

In[33]:= reverse[{a, b, c, d, e}]

Out[33]= {e, d, c, b, a}
```

```
In[34]:= reverseStruc[vec_] := Module[{vecA, n = Length[vec]},
           vecA = ConstantArray[0, {n}];
           Table[vecA[[i]] = vec[[n - i + 1]], {i, n}]
          ]

In[35]:= reverseStruc[{a, b, c, d, e}]

Out[35]= {e, d, c, b, a}
```

b. The key to this problem is to use the `Mod` operator to compute the target address for any item from `vec`. That is, the element `vec[i]` must move to, roughly speaking, position $n + i$ mod `Length[vec]`. The “roughly speaking” is due to the fact that the `Mod` operator returns values in the range 0 to `Length[vec] - 1`, whereas vectors are indexed by values 1 up to `Length[vec]`. This causes a little trickiness in this problem.

```
In[36]:= rotateRight[vec_, n_] := Module[{vecA, len = Length[vec]},
           vecA = ConstantArray[0, {len}];
           Do[vecA[[1 + Mod[n + i - 1, len]]] = vec[[i]], {i, 1, len}];
           vecA]

In[37]:= rotateRight[{a, b, c, d, e}, 2]

Out[37]= {d, e, a, b, c}

In[38]:= rotateRightStruc[vec_, n_] := Module[{vecA, len = Length[vec]},
           vecA = ConstantArray[0, {len}];
           Table[vecA[[1 + Mod[n + i - 1, len]]] = vec[[i]], {i, len}];
           vecA
          ]

In[39]:= rotateRightStruc[{a, b, c, d, e}, 3]

Out[39]= {c, d, e, a, b}
```

c. Iterate over the rows of `mat`, setting row *i* to the result of calling `rotateRight`.

```
In[40]:= rotateRows[mat_] := Module[{matA, len = Length[mat]},
           matA = ConstantArray[0, {len}];
           Do[matA[[i]] = rotateRight[mat[[i]], i],
            {i, 1, len}];
           matA]

In[41]:= rotateRows[{{a, b, c}, {d, e, f}, {g, h, k}}]

Out[41]= {{c, a, b}, {e, f, d}, {g, h, k}}
```

d. Similar construction to the previous exercise.

```
In[42]:= rotateRowsByS[mat_, S_] := Module[{matA, len = Length[mat]},
           matA = ConstantArray[0, {len}];
           Do[matA⟦i⟧ = rotateRight[mat⟦i⟧, S⟦i⟧],
            {i, 1, len}];
           matA]

In[43]:= rotateRowsByS[{{a, b, c}, {d, e, f}, {g, h, k}}, {1, 2, 3}]

Out[43]= {{c, a, b}, {e, f, d}, {g, h, k}}
```

e. Create a list `lisC` of correct length, then iterate over `lisA` and `lisB`, moving `lisA`⟦*i*⟧ to `lisC` whenever `lisB`⟦*i*⟧ is `True`. The position in `lisC` that receives this value is not necessarily *i*; we use the variable `last` to keep track of the next position in `lisC` that will receive a value from `lisA`.

```
In[44]:= pick[lisA_, lisB_] :=
          Module[{lisC = Table[0, {Count[lisB, True]}], last = 1},
           Do[
            If[lisB⟦i⟧,
             lisC⟦last⟧ = lisA⟦i⟧; last = last + 1],
            {i, 1, Length[lisB]}];
           lisC]

In[45]:= pick[{a, b, c, d, e}, {True, True, False, False, True}]

Out[45]= {a, b, e}
```

This is doing the same computation as the built-in `Pick` function.

```
In[46]:= Pick[{a, b, c, d, e}, {True, True, False, False, True}]

Out[46]= {a, b, e}
```

6.2 *Flow control*

1. Create a function `UpperTriangularMatrix[{`*m*`, `*n*`}]` that generates an $m \times n$ upper triangular matrix, that is, a matrix containing 1s on and above the diagonal and 0s below the diagonal. Create an alternative rule that defaults to 1 for the upper values, but allows the user to specify a nondefault upper value.

```
In[1]:= UpperTriangularMatrix[{3, 3}] // MatrixForm
Out[1]//MatrixForm=
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

```
In[2]:= UpperTriangularMatrix[{4, 4}, ξ] // MatrixForm
Out[2]//MatrixForm=
```

$$\begin{pmatrix} \xi & \xi & \xi & \xi \\ 0 & \xi & \xi & \xi \\ 0 & 0 & \xi & \xi \\ 0 & 0 & 0 & \xi \end{pmatrix}$$

2. Write a function `signum[x]` which, when applied to an integer x, returns -1, 0, or 1, if x is less than, equal to, or greater than 0, respectively. Write it in four ways: using three clauses, using a single clause with `If`, using a single clause with `Which`, and using `Piecewise`.

3. The definition of the absolute value function in this section does not handle complex numbers properly.

```
In[3]:= abs[3 + 4 I]

    GreaterEqual::nord : Invalid comparison with 3 + 4 ⅈ attempted. ≫

    Less::nord : Invalid comparison with 3 + 4 ⅈ attempted. ≫

Out[3]= abs[3 + 4 I]
```

Correct this problem by rewriting `abs` to include a specific rule for the case where its argument is complex.

4. Use `If` in conjunction with `Map` or `Fold` to define the following functions:

   a. In a list of numbers, double all the positive numbers, but leave the negative numbers alone.

   b. `remove3Repetitions` alters three or more consecutive occurrences in a list, changing them to two occurrences; if there are only two occurrences to begin with, they are left alone. For example, `remove3Repetitions[{0, 1, 1, 2, 2, 2, 1}]` will return `{0, 1, 1, 2, 2, 1}`.

   c. Add the elements of a list in consecutive order, but never let the sum go below 0.

```
In[4]:= positiveSum[{5, 3, -13, 7, -3, 2}]
Out[4]= 6
```

Since the –13 caused the sum to go below 0, it was instead put back to 0 and the summation continued from there.

5. Rewrite the `median` function from Exercise 4 in Section 5.8 using an `If` control structure.

6. Using `NestWhileList`, write a function `CollatzSequence[n]` that produces the Collatz sequence for any positive integer n. The Collatz sequence is generated as follows: starting with a number n, if it is even, then output $n/2$; if n is odd, then output $3n + 1$. Iterate this process while $n \neq 1$.

6.2 Solutions

1. If, for element a_{ij}, i is bigger than j, then we are below the diagonal and should insert a 0, otherwise insert a 1.

```
In[1]:= UpperTriangularMatrix[{m_, n_}] :=
         Table[If[i ≥ j, 0, 1], {i, m}, {j, n}]
```

A default value can be given for an optional argument that specifies the elements above the diagonal.

```
In[2]:= UpperTriangularMatrix[{m_, n_}, val_: 1] :=
         Table[If[i ≥ j, 0, val], {i, m}, {j, n}]

In[3]:= UpperTriangularMatrix[{5, 5}, α] // MatrixForm
```

Out[3]//MatrixForm=

$$\begin{pmatrix} 0 & \alpha & \alpha & \alpha & \alpha \\ 0 & 0 & \alpha & \alpha & \alpha \\ 0 & 0 & 0 & \alpha & \alpha \\ 0 & 0 & 0 & 0 & \alpha \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

2. Here are the conditional definitions.

```
In[4]:= signum[x_ /; x < 0] := -1
        signum[x_ /; x > 0] := 1
        signum[0] := 0
        signum[0.0] := 0

In[8]:= Map[signum, {-2, 0, 1}]
```

Here is the signum function defined using `If`.

```
In[9]:= signumIf[x_] := If[x < 0, -1, If[x == 0, 0, 1]]

In[10]:= Map[signumIf, {-2, 0, 1}]
```

Here is the signum function defined using `Which`.

```
In[11]:= signumWhich[x_] := Which[x < 0, -1, x > 0, 1, True, 0]

In[12]:= Map[signumWhich, {-2, 0, 1}]
```

Finally, here is the signum function defined using `Piecewise`.

```
In[13]:= Piecewise[{{-1, x < 0}, {1, x > 0}, {0, x == 0}}]
```

3. The test as the first argument of `If` on the right-hand side checks to see if x is an element of the domain of complex numbers and, if it is, then $\sqrt{\mathrm{re}(x)^2 + \mathrm{im}(x)^2}$ is computed. If x is not complex, nothing is done, but then the other definitions for `abs` will be checked.

```
In[14]:= Clear[abs];
         abs[x_] := Sqrt[Re[x]^2 + Im[x]^2] /; x ∈ Complexes;
         abs[x_] := x /; x ≥ 0
         abs[x_] := -x /; x < 0
```

```
In[18]:= abs[3 + 4 I]

Out[18]= 5

In[19]:= abs[-3]

Out[19]= 3
```

The condition itself can appear on the left-hand side of the function definition, as part of the pattern match. Here is a slight variation on the `abs` definition.

```
In[20]:= Clear[abs]
         abs[x_] := If[x ≥ 0, x, -x]
         abs[x_ /; x ∈ Complexes] := Sqrt[Re[x]^2 + Im[x]^2]

In[23]:= abs[3 + 4 I]

Out[23]= 5

In[24]:= abs[-3]

Out[24]= 3
```

4.

a. The pure function doubles its argument if it is greater than zero.

```
In[25]:= doublePos[lis_] := Map[If[# > 0, 2 #, #] &, lis]
```

b.

```
In[26]:= remove3Repetitions[lis_] := Fold[
           If[Length[#1] > 2 && #2 == #1[[-1]] == #1[[-2]], #1, Join[#1, {#2}]] &,
           {}, lis]
```

c.

```
In[27]:= positiveSum[L_] := Fold[If[#1 + #2 < 0, 0, #1 + #2] &, 0, L]
```

5. This is a straightforward conversion from the two rules given in Exercise 4 in Section 5.8 to an `If` statement.

```
In[28]:= medianP[lis : {__}] := Module[{len = Length[lis]},
           If[OddQ[len],
            Part[Sort[lis], Ceiling[len / 2]],
            Mean@Part[Sort[lis], len / 2 ;; len / 2 + 1]
           ]]

In[29]:= dataO = RandomInteger[10 000, 100 001];
         dataE = RandomInteger[10 000, 100 000];

In[31]:= medianP[dataO] // Timing

Out[31]= {0.019074, 5001}
```

```
In[32]:= Median[dataO] // Timing
Out[32]= {0.019326, 5001}

In[33]:= medianP[dataE] // Timing
Out[33]= {0.020737, 4985}

In[34]:= Median[dataE] // Timing
Out[34]= {0.018148, 4985}
```

6. First, define the auxiliary function using conditional statements.

```
In[35]:= collatz[n_] := n/2 /; EvenQ[n]

In[36]:= collatz[n_] := 3 n + 1 /; OddQ[n]
```

Alternatively, use `If`.

```
In[37]:= collatz[n_Integer?Positive] := If[EvenQ[n], n / 2, 3 n + 1]
```

Then iterate `Collatz`, starting with n, and continue while n is not equal to 1.

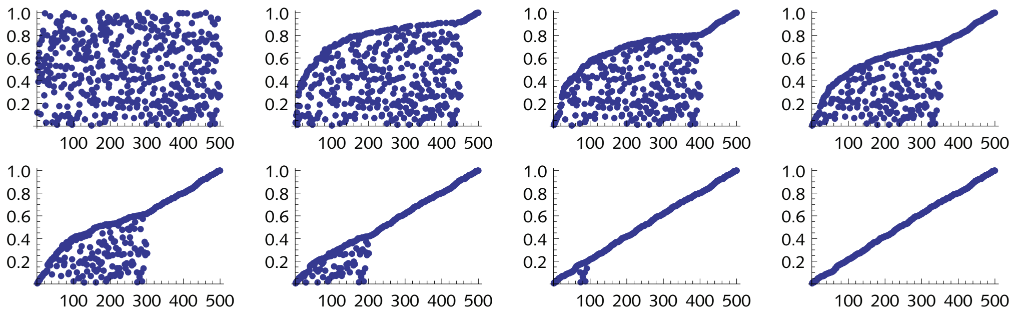
```
In[38]:= CollatzSequence[n_] := NestWhileList[collatz, n, # ≠ 1 &]

In[39]:= CollatzSequence[17]
Out[39]= {17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

6.3 *Examples and applications*

1. Using an `If` function, write a function `gcd[m, n]` that implements the Euclidean algorithm (see Exercise 9 of Section 6.1) for finding the greatest common divisor of *m* and *n*.
2. The *digit sum* of a number is given by adding the digits of that number. For example, the digit sum of 7763 is $7 + 7 + 6 + 3 = 23$. If you iterate the digit sum until the resulting number has only one digit, this is called the *digit root* of the original number. So the digit root of 7763 is $7763 \to 7 + 7 + 6 + 3 = 23 \to 2 + 3 = 5$. Create a function to compute the digit root of any positive integer.
3. Use `Piecewise` to define the `quadrant` function given in this section.
4. In the version of `quadrant` using `If` and `Which` developed in this section, the point `{0.0, 0.0}` is not handled properly because of how *Mathematica* treats the real number 0.0 compared with the integer 0. Write another version of `quadrant` using alternatives (discussed in Section 4.1) to handle this situation and correctly return the 0.
5. Extend `quadrant` to three dimensions, following this rule: for point (x, y, z), if $z \geq 0$, then give the same classification as (x, y), with the exception that 0 is treated as a positive number (so the only classifications are 1, 2, 3, and 4); if $z < 0$, add 4 to the classification of (x, y) (with the same exception). For example, $(1, 0, 1)$ is in octant 1, and $(0, -3, -3)$ is in octant 8. `quadrant` should work for points in two or three dimensions.

Consider a sequence of numbers generated by the following iterative process: starting with the list of odd integers 1, 3, 5, 7, …, the first odd number greater than 1 is 3, so delete every third number from the list; from the list of remaining numbers, the next number is 7, so delete every seventh number; and so on. The numbers that remain after this process has been carried out completely are referred to as *lucky numbers* (Weisstein, Lucky Number). Use a sieving method to find all lucky numbers less than 1000.

7. Create an animation for bubble sort similar to the animation in the text for selection sort.



6.3 *Solutions*

1. Here is the `gcd` function implemented using an `If` structure.

```
In[1]:=  Clear[gcd]

In[2]:=  gcd[m_Integer, n_Integer] :=
          If[m > 0, gcd[Mod[n, m], m], gcd[m, n] = n]

In[3]:=  With[{m = 12 782, n = 5 531 207},
         gcd[m, n]]

Out[3]=  11
```

2. Given an integer, this totals the list of its digits.

```
In[4]:=  Total[IntegerDigits[7763]]

Out[4]=  23
```

To repeat this process until the resulting integer has only one digit, use `While`.

```
In[5]:=  digitRoot[n_Integer?Positive] := Module[{locn = n, lis},
           While[
            Length[lis = IntegerDigits@locn] > 1,
            locn = Total[lis]];
           locn]

In[6]:=  digitRoot[7763]

Out[6]=  5
```

This can also be accomplished without iteration as follows:

```
In[7]:=  digitRoot2[n_Integer?Positive] := If[Mod[n, 9] == 0, 9, Mod[n, 9]]

In[8]:=  digitRoot2[1000!]

Out[8]=  9
```

3. This is a direct implementation using `Piecewise`.

```
In[9]:=  Piecewise[{{0, x == 0 && y == 0}, {-1, y == 0}, {-2, x == 0},
           {1, x > 0 && y > 0}, {2, x < 0 && y > 0}, {3, x < 0 && y < 0}}, 4]

In[10]:= quadrantPW[{x_, y_}] :=
          Piecewise[{{0, x == 0 && y == 0}, {-1, y == 0}, {-2, x == 0},
            {1, x > 0 && y > 0}, {2, x < 0 && y > 0}, {3, x < 0 && y < 0}}, 4]

In[11]:= Map[quadrantPW, {{0, 0}, {4, 0}, {0, 1.3},
           {2, 4}, {-2, 4}, {-2, -4}, {2, -4}, {2, 0}, {3, -4}}]
```

4. The alternatives we need to check for are `0 | 0.0` for both *x* and y.

```
In[9]:=  quadrant[{0 | 0.0, 0 | 0.0}] := 0
         quadrant[{x_, 0 | 0.0}] := -1
         quadrant[{0 | 0.0, y_}] := -2
         quadrant[{x_, y_}] := If[x < 0, 2, 1] /; y > 0
         quadrant[{x_, y_}] := If[x < 0, 3, 4]

In[14]:= quadrant[{0.0, 0}]

Out[14]= 0

In[15]:= quadrant[{1, 0}]

Out[15]= -1
```

5. These rules are basic extensions of the two-dimensional cases.

```
In[19]:= quadrant[{0, 0}] := 0
         quadrant[{x_, 0}] := -1
         quadrant[{0, y_}] := -2
         quadrant[{x_, y_}] := If[x < 0, 2, 1] /; y > 0
         quadrant[{x_, y_}] := If[x < 0, 3, 4]
         quadrant[{x_, y_, z_}] := If[x < 0, 2, 1] /; y ≥ 0 && z ≥ 0
         quadrant[{x_, y_, z_}] := If[x < 0, 3, 4] /; y < 0 && z ≥ 0
         quadrant[{x_, y_, z_}] := If[x < 0, 6, 5] /; y ≥ 0 && z < 0
         quadrant[{x_, y_, z_}] := If[x < 0, 7, 8] /; y < 0 && z < 0

In[28]:= Map[quadrant, {{2, 0, 1}, {-1, 3, -4}}]
```

6. Start with a small list of odd numbers.

```
In[29]:= ints = Range[1, 100, 2]

Out[29]= {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35,
          37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67,
          69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99}
```

On the first iteration, drop every third number, that is, drop 5, 11, 17, and so on.

```
In[30]:= p = ints[[2]];
         ints = Drop[ints, p ;; -1 ;; p]

Out[31]= {1, 3, 7, 9, 13, 15, 19, 21, 25, 27, 31, 33, 37, 39, 43, 45, 49, 51,
          55, 57, 61, 63, 67, 69, 73, 75, 79, 81, 85, 87, 91, 93, 97, 99}
```

Get the next number, 7, in the list `ints`; then drop every seventh number.

```
In[32]:= p = ints[[3]];
         ints = Drop[ints, p ;; -1 ;; p]

Out[33]= {1, 3, 7, 9, 13, 15, 21, 25, 27, 31, 33, 37, 43, 45, 49,
          51, 55, 57, 63, 67, 69, 73, 75, 79, 85, 87, 91, 93, 97, 99}
```

Iterate. You will need to be careful about the upper limit of the iterator `i`.

```
In[34]:= ints = Range[1, 1000, 2];
         Do[
          p = ints[[i]];
          ints = Drop[ints, p ;; -1 ;; p],
          {i, 2, 32}]
         ints

Out[36]= {1, 3, 7, 9, 13, 15, 21, 25, 31, 33, 37, 43, 49, 51, 63, 67, 69,
          73, 75, 79, 87, 93, 99, 105, 111, 115, 127, 129, 133, 135, 141,
          151, 159, 163, 169, 171, 189, 193, 195, 201, 205, 211, 219, 223,
          231, 235, 237, 241, 259, 261, 267, 273, 283, 285, 289, 297, 303,
          307, 319, 321, 327, 331, 339, 349, 357, 361, 367, 385, 391,
          393, 399, 409, 415, 421, 427, 429, 433, 451, 463, 475, 477,
          483, 487, 489, 495, 511, 517, 519, 529, 535, 537, 541, 553,
          559, 577, 579, 583, 591, 601, 613, 615, 619, 621, 631, 639,
          643, 645, 651, 655, 673, 679, 685, 693, 699, 717, 723, 727,
          729, 735, 739, 741, 745, 769, 777, 781, 787, 801, 805, 819,
          823, 831, 841, 855, 867, 873, 883, 885, 895, 897, 903, 925,
          927, 931, 933, 937, 957, 961, 975, 979, 981, 991, 993, 997}
```

It would be more efficient if you did not need to manually determine the upper limit of the iteration. A `While` loop is better for this task. The test checks that the value of the iterator has not gone past the length of the successively shortened lists.

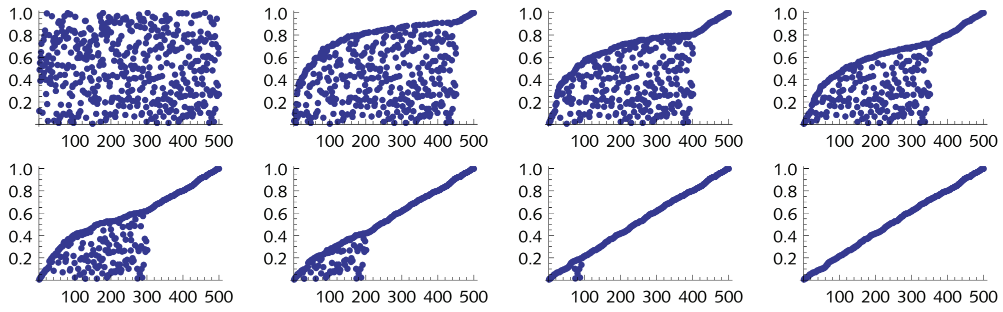
```
In[37]:= LuckyNumbers[n_Integer?Positive] :=
          Module[{p, i = 2, ints = Range[1, n, 2]},
           While[ints[[i]] < Length[ints],
            p = ints[[i]];
            ints = Drop[ints, p ;; -1 ;; p];
            i++];
           ints]

In[38]:= LuckyNumbers[1000]

Out[38]= {1, 3, 7, 9, 13, 15, 21, 25, 31, 33, 37, 43, 49, 51, 63, 67, 69,
          73, 75, 79, 87, 93, 99, 105, 111, 115, 127, 129, 133, 135, 141,
          151, 159, 163, 169, 171, 189, 193, 195, 201, 205, 211, 219, 223,
          231, 235, 237, 241, 259, 261, 267, 273, 283, 285, 289, 297, 303,
          307, 319, 321, 327, 331, 339, 349, 357, 361, 367, 385, 391,
          393, 399, 409, 415, 421, 427, 429, 433, 451, 463, 475, 477,
          483, 487, 489, 495, 511, 517, 519, 529, 535, 537, 541, 553,
          559, 577, 579, 583, 591, 601, 613, 615, 619, 621, 631, 639,
          643, 645, 651, 655, 673, 679, 685, 693, 699, 717, 723, 727,
          729, 735, 739, 741, 745, 769, 777, 781, 787, 801, 805, 819,
          823, 831, 841, 855, 867, 873, 883, 885, 895, 897, 903, 925,
          927, 931, 933, 937, 957, 961, 975, 979, 981, 991, 993, 997}
```

This latter approach is also reasonably fast. Here is the time it takes to compute all lucky numbers less than one million; there are 71918 of them.

```
In[39]:= Length[LuckyNumbers[10^6]] // Timing

Out[39]= {0.265116, 71918}
```

7. Use the same constructs as were used in the text for selection sort.

```
In[40]:= bubbleSortList[lis_] :=
          Module[{slist = lis, len = Length[lis], tmp = {}},
           For[i = len, i > 0, i--,
            AppendTo[tmp, slist];
            For[j = 2, j ≤ i, j++,
             If[slist〚j - 1〛 > slist〚j〛,
              slist〚{j - 1, j}〛 = slist〚{j, j - 1}〛]]];
           tmp]

In[41]:= data = RandomReal[1, 500];
         sort = bubbleSortList[data];
         ListAnimate[ListPlot /@ sort];
```


1.0
0.8
0.6
0.4
0.2
100 200 300 400 500

7 Recursion

7.1 *Fibonacci numbers*

1. For each of the following sequences of numbers, see if you can deduce the pattern and write a *Mathematica* function to compute the general term.

a.
$$\begin{array}{cccccccc} 2, & 3, & 6, & 18, & 108, & 1944, & 209\,952, & \dots \\ A_1 & A_2 & A_3 & A_4 & A_5 & A_6 & A_7 & \dots \end{array}$$

b.
$$\begin{array}{cccccccccc} 0, & 1, & -1, & 2, & -3, & 5, & -8, & 13, & -21, & \dots \\ B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & B_7 & B_8 & B_9 & \dots \end{array}$$

c.
$$\begin{array}{cccccccccc} 0, & 1, & 2, & 3, & 6, & 11, & 20, & 37, & 68, & \dots \\ C_1 & C_2 & C_3 & C_4 & C_5 & C_6 & C_7 & C_8 & C_9 & \dots \end{array}$$

2. The numbers FA_n represent the number of additions that are done in the course of evaluating the Fibonacci function `F[n]` defined in this section.

$$\begin{array}{cccccccccc} 0 & 0 & 1 & 2 & 4 & 7 & 12 & 20 & 33 & \dots \\ FA_1 & FA_2 & FA_3 & FA_4 & FA_5 & FA_6 & FA_7 & FA_8 & FA_9 & \dots \end{array}$$

Write a function `FA` such that `FA[n] =` FA_n.

3. A faster approach to computing Fibonacci numbers uses various identities associated with these numbers (The Fibonacci Sequence 2011). We start the base case at 0 instead of 1 here. The notation $\lfloor number \rfloor$ represents the floor of *number*. You can use `IntegerPart`.

$$f_0 = 0$$
$$f_1 = 1$$

$$f_n = \begin{cases} f[k]\,(f[k] + 2\,f[k-1]) & n \text{ even},\ k = \lfloor n/2 \rfloor \\ (2\,f[k] + f[k-1])\,(2\,f[k] - f[k-1]) + 2 & n \bmod 4 = 1,\ k = \lfloor n/2 \rfloor \\ (2\,f[k] + f[k-1])\,(2\,f[k] - f[k-1]) - 2 & \text{otherwise} \end{cases}$$

Implement this algorithm. Consider using `Which` for the different conditions.

4. The Fibonacci sequence can also be defined for negative integers using the following formula (Graham, Knuth, and Patashnik 1994):

$$F_{-n} = (-1)^{n-1}\,F_n$$

The first few terms are

$$\begin{array}{cccccccccc} 0 & 1 & -1 & 2 & -3 & 5 & -8 & 13 & -21 & \ldots \\ F_0 & F_{-1} & F_{-2} & F_{-3} & F_{-4} & F_{-5} & F_{-6} & F_{-7} & F_{-8} & \ldots \end{array}$$

Write the definitions for Fibonacci numbers with negative integer arguments.

7.1 Solutions

1. The key here is to get the stopping conditions right in each case.

   a. This is a straightforward recursion, multiplying the previous two values to get the next.

```
In[1]:=  a[1] := 2
         a[2] := 3
         a[i_] := a[i - 1] a[i - 2]

In[4]:=  Table[a[i], {i, 1, 8}]

Out[4]=  {2, 3, 6, 18, 108, 1944, 209 952, 408 146 688}
```

   b. The sequence is obtained by taking the difference of the previous two values.

```
In[5]:=  b[1] := 0
         b[2] := 1
         b[i_] := b[i - 2] - b[i - 1]

In[8]:=  Table[b[i], {i, 1, 9}]

Out[8]=  {0, 1, -1, 2, -3, 5, -8, 13, -21}
```

   c. Here we add the previous three values.

```
In[9]:=  c[1] := 0
         c[2] := 1
         c[3] := 2
         c[i_] := c[i - 3] + c[i - 2] + c[i - 1]

In[13]:= Table[c[i], {i, 1, 9}]

Out[13]= {0, 1, 2, 3, 6, 11, 20, 37, 68}
```

2. It is important to get the two base cases right here.

```
In[14]:= FA[1] := 0
         FA[2] := 0
         FA[i_] := FA[i - 2] + FA[i - 1] + 1

In[17]:= Map[FA, Range[9]]

Out[17]= {0, 0, 1, 2, 4, 7, 12, 20, 33}
```

   It is interesting to note that the number of additions needed to compute the *n*th Fibonacci number is one less than the *n*th Fibonacci number itself. As the Fibonacci numbers grow, so too does the computation!

```
In[18]:= Fibonacci /@ Range[9]

Out[18]= {1, 1, 2, 3, 5, 8, 13, 21, 34}
```

3. This is a direct implementation of the traditional mathematical notation given in the exercise. Avoiding the double recursion of the naive implementation reduces the memory required and speeds things up significantly, although it is still too slow for large numbers.

```
In[19]:= Clear[fib, f];
         fib[0] = 0;
         fib[1] = 1;

In[22]:= fib[n_Integer?Positive] := With[{k = IntegerPart[n / 2]},
           Which[
            EvenQ[n], fib[k] (fib[k] + 2 fib[k - 1]),
            Mod[n, 4] == 1, (2 fib[k] + fib[k - 1]) (2 fib[k] - fib[k - 1]) + 2,
            True, (2 fib[k] + fib[k - 1]) (2 fib[k] - fib[k - 1]) - 2
           ]]

In[23]:= Timing@Table[fib[i], {i, 1, 40}]

Out[23]= {0.023411, {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
           233, 377, 610, 987, 1597, 2584, 4181, 6765, 10 946, 17 711,
           28 657, 46 368, 75 025, 121 393, 196 418, 317 811, 514 229,
           832 040, 1 346 269, 2 178 309, 3 524 578, 5 702 887, 9 227 465,
           14 930 352, 24 157 817, 39 088 169, 63 245 986, 102 334 155}}
```

4. You can use your earlier definition of the Fibonacci numbers, or use the built-in `Fibonacci`.

```
In[24]:= f[n_Integer?NonPositive] := (-1)^(n-1) Fibonacci[-n]

In[25]:= f[0] = 0;
         f[-1] = 1;

In[27]:= Table[f[i], {i, 0, -8, -1}]

Out[27]= {0, 1, -1, 2, -3, 5, -8, 13, -21}
```

7.2 *Thinking recursively*

1. Create a recursive function to reverse the elements in a flat list.
2. Create a recursive function to transpose the elements of two lists. Write an additional rule to transpose the elements of three lists.
3. Write a recursive function `sumOddElements[`*lis*`]` that adds up only the elements of the list *lis* that are odd integers. *lis* may contain even integers and nonintegers.
4. Write a recursive function `sumEveryOtherElement[`*lis*`]` that adds up *lis*`[[1]]`, *lis*`[[3]]`, *lis*`[[5]]`, etc. Each of these elements is a number. *lis* may have any number of elements.
5. Write a function `addTriples[`lis_1`, `lis_2`, `lis_3`]` that is like `addPairs` in that it adds up the corresponding elements of the three equal-length lists of numbers.

6. Write a function `multAllPairs[lis]` that multiplies every consecutive pair of integers in the numerical list *lis*. Add a rule that issues an appropriate warning message if the user supplies a list with an odd number of elements.

```
In[1]:= multAllPairs[{3, 9, 17, 2, 6, 60}]

Out[1]= {27, 153, 34, 12, 360}
```

7. Write the function `maxPairs[`lis_1`, `lis_2`]` which, for numerical lists of equal length, returns a list of the larger value in each corresponding pair.

8. The function `riffle[`lis_1`, `lis_2`]`, which merges two lists of equal length, can be defined as follows:

```
In[2]:= riffle[lis1_, lis2_] := Flatten[Transpose[{lis1, lis2}]]

In[3]:= riffle[{a, b, c}, {x, y, z}]

Out[3]= {a, x, b, y, c, z}
```

Rewrite `riffle` using recursion.

9. `maxima` can also be computed more efficiently with an auxiliary function.

```
maxima[{}] := {}
maxima[{x_, r___}] := maxima[x, {r}]
```

The two-argument version has this meaning: `maxima[x, lis]` gives the maxima of the list `Join[{x}, lis]`. Define it. (Hint: the key point about this is that `maxima[x, lis]` is equal to `maxima[x, Rest[lis]]` if $x \geq$ `First[lis]`.) Compare its efficiency with the version in the text.

10. Write recursive definitions for `Fold`, `FoldList`, and `NestList`.

7.2 Solutions

1. This is similar to the `length` function in the text – recursion is on the tail. The base case is a list consisting of a single element.

```
In[1]:= reverse[{x_, y__}] := Join[reverse[{y}], {x}]

In[2]:= reverse[{x_}] := {x}

In[3]:= reverse[{1, β, 3 / 4, "practice makes perfect"}]

                                   3
Out[3]= {practice makes perfect, ─, β, 1}
                                   4
```

2. Recursion is on the tails of the two lists, here denoted `r1` and `r2`.

```
In[4]:= transpose[{{x1_, r1__}, {x2_, r2__}}] :=
         Join[{{x1, x2}}, transpose[{{r1}, {r2}}]]

In[5]:= transpose[{{x_}, {y_}}] := {{x, y}}
```

```
In[6]:=  transpose[{{x₁, x₂}, {y₁, y₂}}]

Out[6]=  {{x₁, y₁}, {x₂, y₂}}

In[7]:=  transpose[%]

Out[7]=  {{x₁, x₂}, {y₁, y₂}}
```

3. Recursion is on the tail.

```
In[8]:=  sumOddElements[{}] := 0
         sumOddElements[{x_, r___}] := x + sumOddElements[{r}] /; OddQ[x]
         sumOddElements[{x_, r___}] := sumOddElements[{r}]

In[11]:= sumOddElements[{2, 3, 5, 6, 7, 9, 12, 13}]

Out[11]= 37
```

4. Again, recursion is on the tail.

```
In[12]:= sumEveryOtherElement[{}] := 0
         sumEveryOtherElement[{x_}] := x
         sumEveryOtherElement[{x_, y_, r___}] :=
          x + sumEveryOtherElement[{r}]

In[15]:= sumEveryOtherElement[{1, 2, 3, 4, 5, 6, 7, 8, 9}]

Out[15]= 25
```

5. This is a direct extension of the `addPairs` function discussed in this section.

```
In[16]:= addTriples[{}, {}, {}] := {}
         addTriples[{x1_, y1___}, {x2_, y2___}, {x3_, y3___}] :=
          Join[{x1 + x2 + x3}, addTriples[{y1}, {y2}, {y3}]]

In[18]:= addTriples[{w₁, x₁, y₁, z₁}, {w₂, x₂, y₂, z₂}, {w₃, x₃, y₃, z₃}]

Out[18]= {w₁ + w₂ + w₃, x₁ + x₂ + x₃, y₁ + y₂ + y₃, z₁ + z₂ + z₃}
```

6. Multiply successive pairs with one element overlap.

```
In[19]:= multAllPairs[{}] := {}
         multAllPairs[{_}] := {}
         multAllPairs[{x_, y_, r___}] := Join[{x y}, multAllPairs[{y, r}]]

In[22]:= multAllPairs[{3, 9, 17, 2, 6, 60}]

Out[22]= {27, 153, 34, 12, 360}
```

7. Recursion is on the tails of each of the two lists.

```
In[23]:= maxPairs[{}, {}] := {}
         maxPairs[{x_, r___}, {y_, s___}] :=
          Join[{Max[x, y]}, maxPairs[{r}, {s}]]
```

```
In[25]:=  maxPairs[{1, 2, 4}, {2, 7, 2}]

Out[25]=  {2, 7, 4}
```

8. Again, we do recursion on the tails of the two lists.

```
In[26]:=  riffle[{}, {}] := {}
          riffle[{x_, r___}, {y_, s___}] := Join[{x, y}, riffle[{r}, {s}]]

In[28]:=  riffle[{a, b, c}, {x, y, z}]

Out[28]=  {a, x, b, y, c, z}
```

Here is the built-in function that does this.

```
In[29]:=  Riffle[{a, b, c}, {x, y, z}]

Out[29]=  {a, x, b, y, c, z}
```

9. Here is `maxima` using an auxiliary function.

```
In[30]:=  maxima[{}] := {}
          maxima[{x_, r___}] := maxima[x, {r}]

In[32]:=  maxima[x_, {}] := {x}
          maxima[x_, {y_, r___}] := maxima[x, {r}] /; x ≥ y
          maxima[x_, {y_, r___}] := Join[{x}, maxima[y, {r}]]
```

10. First, here is the definition for our user-defined `fold`.

```
In[35]:=  fold[f_, x_, {}] := x
          fold[f_, x_, {a_, r___}] := fold[f, f[x, a], {r}]

In[37]:=  fold[Plus, 0, {a, b, c, d, e}]

Out[37]=  a + b + c + d + e

In[38]:=  foldList[f_, x_, {}] := {x}
          foldList[f_, x_, {a_, r___}] :=
           Join[{x}, foldList[f, f[x, a], {r}]]

In[40]:=  foldList[Times, 1, Range[6]]

Out[40]=  {1, 1, 2, 6, 24, 120, 720}
```

And here is `nestList`.

```
In[41]:=  nestList[f_, x_, 0] := {x}
          nestList[f_, x_, n_] := Join[{x}, nestList[f, f[x], n - 1]]

In[43]:=  nestList[Sin, θ, 3]

Out[43]=  {θ, Sin[θ], Sin[Sin[θ]], Sin[Sin[Sin[θ]]]}
```

7.3 *Dynamic programming*

1. An Eulerian number, denoted $\left\langle {n \atop k} \right\rangle$, gives the number of permutations with k increasing runs of elements. For example, for $n = 3$ the permutations of {1,2,3} contain four increasing runs of length 1, namely {1,3,2}, {2,1,3}, {2,3,1}, and {3,1,2}. Hence, $\left\langle {3 \atop 1} \right\rangle = 4$.

```
In[1]:=  Permutations[{1, 2, 3}]
Out[1]=  {{1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, {3, 2, 1}}
```

This can be programmed using the following recursive definition (Graham, Knuth, and Patashnik 1994), where n and k are assumed to be integers:

$$\left\langle {n \atop k} \right\rangle = (k+1)\left\langle {n-1 \atop k} \right\rangle + (n-k)\left\langle {n-1 \atop k-1} \right\rangle, \quad \text{for } n > 0,$$

$$\left\langle {0 \atop k} \right\rangle = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0. \end{cases}$$

Create a function `EulerianNumber[n, k]`. You can check your work against Table 7.1 which displays the first few Eulerian numbers.

TABLE 7.1. *Eulerian number triangle*

	$\left\langle {n \atop 0} \right\rangle$	$\left\langle {n \atop 1} \right\rangle$	$\left\langle {n \atop 2} \right\rangle$	$\left\langle {n \atop 3} \right\rangle$	$\left\langle {n \atop 4} \right\rangle$	$\left\langle {n \atop 5} \right\rangle$	$\left\langle {n \atop 6} \right\rangle$	$\left\langle {n \atop 7} \right\rangle$	$\left\langle {n \atop 8} \right\rangle$
0	1								
1	1	0							
2	1	1	0						
3	1	4	1	0					
4	1	11	11	1	0				
5	1	26	66	26	1	0			
6	1	57	302	302	57	1	0		
7	1	120	1191	2416	1191	120	1	0	
8	1	247	4293	15 619	15 619	4293	247	1	0

Because of the triple recursion, you will find it necessary to use a dynamic programming implementation to compute any Eulerian numbers of even modest size.

Hint: Although the above formulas will compute it, you can add the following rule to simplify some of the computation:

$$\left\langle {n \atop k} \right\rangle = 0, \quad \text{for } k \geq n$$

2. Using dynamic programming is one way to speed up the computation of the Fibonacci numbers, but another is to use a different algorithm. A much more efficient algorithm is based on the following identities.

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_{2n} &= 2F_{n-1}F_n + F_n^2, \quad \text{for } n \geq 1 \\ F_{2n+1} &= F_{n+1}^2 + F_n^2, \qquad \text{for } n \geq 1 \end{aligned}$$

   Program a Fibonacci number generating function using these identities.

3. You can still speed up the code for generating Fibonacci numbers in the previous exercise by using dynamic programming. Do so, and construct tables like those in this section, giving the number of additions performed for various *n* by the two programs you have just written.

4. Calculation of the Collatz numbers, as described in Exercise 6 from Section 6.2, can be implemented using recursion and sped up by using dynamic programming. Using recursion and dynamic programming, create the function `collatz[n, i]`, which computes the *i*th iterate of the Collatz sequence starting with integer *n*. Compare its speed with that of your original solution.

7.3 Solutions

1. Here are the rules translated directly from the formulas given in the exercise.

```
In[1]:= EulerianNumber[0, k_] = 0;
        EulerianNumber[n_Integer, 0] = 1;
        EulerianNumber[n_Integer, k_Integer] /; k ≥ n = 0;

In[4]:= EulerianNumber[n_Integer, k_Integer] :=
         (k + 1) EulerianNumber[n - 1, k] + (n - k) EulerianNumber[n - 1, k - 1]

In[5]:= Table[EulerianNumber[n, k], {n, 0, 7}, {k, 0, 7}] // TableForm
```

```
                    0   0     0     0     0     0     0   0
                    1   0     0     0     0     0     0   0
                    1   1     0     0     0     0     0   0
                    1   4     1     0     0     0     0   0
Out[5]//TableForm=  1   11    11    1     0     0     0   0
                    1   26    66    26    1     0     0   0
                    1   57    302   302   57    1     0   0
                    1   120   1191  2416  1191  120   1   0
```

   Because of the triple recursion, computing larger values is not only time and memory intensive but also bumps up against the built-in recursion limit.

```
In[6]:= EulerianNumber[25, 15] // Timing

Out[6]= {18.4585, 531 714 261 368 950 897 339 996}
```

This is a good candidate for dynamic programming. In the following implementation we have temporarily reset the value of `$RecursionLimit` using `Block`.

```
In[7]:=  Clear[EulerianNumber];

In[8]:=  EulerianNumber[0, k_] = 0;
         EulerianNumber[n_Integer, 0] = 1;
         EulerianNumber[n_Integer, k_Integer] /; k ≥ n = 0;

In[11]:= EulerianNumber[n_Integer, k_Integer] :=
          Block[{$RecursionLimit = Infinity},
           EulerianNumber[n, k] = (k + 1) EulerianNumber[n - 1, k] +
             (n - k) EulerianNumber[n - 1, k - 1]]

In[12]:= EulerianNumber[25, 15] // Timing

Out[12]= {0.00253, 531 714 261 368 950 897 339 996}

In[13]:= EulerianNumber[600, 65]; // Timing

Out[13]= {0.385132, Null}

In[14]:= N[EulerianNumber[600, 65]]

Out[14]= 4.998147102049161 × 10^1091
```

2. This implementation uses the identities given in the exercise together with some pattern matching for the even and odd cases.

```
In[15]:= F[1] := 1
         F[2] := 1

In[17]:= F[n_?EvenQ] := 2 F[n/2 - 1] F[n/2] + F[n/2]^2

         F[n_?OddQ] := F[(n - 1)/2 + 1]^2 + F[(n - 1)/2]^2

In[19]:= Map[F, Range[10]]

Out[19]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}

In[20]:= Timing[F[10^4];]

Out[20]= {0.43854, Null}
```

3. The use of dynamic programming speeds up the computation by several orders of magnitude.

```
In[21]:= FF[1] := 1
         FF[2] := 1

In[23]:= FF[n_?EvenQ] := FF[n] = 2 FF[n/2 - 1] FF[n/2] + FF[n/2]^2

         FF[n_?OddQ] := FF[n] = FF[(n - 1)/2 + 1]^2 + FF[(n - 1)/2]^2
```

```
In[25]:= Map[FF, Range[10]]

Out[25]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

```
In[26]:= Timing[FF[10^5];]

Out[26]= {0.001024, Null}
```

This is fairly fast, even compared with the built-in `Fibonacci` which uses a method based on the binary digits of *n*.

```
In[27]:= Timing[Fibonacci[10^5];]

Out[27]= {0.000275, Null}
```

4. Recursion is on the tail of the iterator *i*.

```
In[28]:= collatz[n_, 0] := n

In[29]:= collatz[n_, i_] :=
          (collatz[n, i] = collatz[n, i - 1] / 2) /; EvenQ[collatz[n, i - 1]]

In[30]:= collatz[n_, i_] :=
          (collatz[n, i] = 3 collatz[n, i - 1] + 1) /; OddQ[collatz[n, i - 1]]
```

Here is the fifth iterate of the Collatz sequence for 27.

```
In[31]:= collatz[27, 5]

Out[31]= 31
```

Here is the Collatz sequence for 27. This sequence takes a while to settle down to the cycle 4, 2, 1.

```
In[32]:= Table[collatz[27, i], {i, 0, 114}]

Out[32]= {27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161,
          484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155,
          466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780,
          890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566,
          283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079,
          3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367,
          4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866,
          433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23,
          70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1}
```

7.4 *Classical examples*

1. Modify one of the `runEncode` functions so that it produces output in the same form as the built-in `Split` function.

```
In[1]:=  Split[{9, 9, 9, 9, 9, 4, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5}]

Out[1]=  {{9, 9, 9, 9, 9}, {4}, {3, 3, 3, 3}, {5, 5, 5, 5, 5, 5}}
```

2. A slightly more efficient version of `runEncode` uses a three-argument auxiliary function.

```
runEncode[{}] := {}
runEncode[{x_, r___}] := runEncode[x, 1, {r}]
```

`runEncode`$[x, k, \{r\}]$ computes the compressed version of $\{x, x, x, \ldots, x, r\}$, where the *x*s are given *k* times. Define this three-argument function. Using the `Timing` function, compare the efficiency of this version with our earlier version; be sure to try a variety of examples, including lists that have many short runs and ones that have fewer, but longer runs. Use `Table` to generate lists long enough to see any difference in speed.

3. Write the function `runDecode`, which takes an encoded list produced by `runEncode` and returns its unencoded form.

```
In[2]:=  runDecode[{{9, 5}, {4, 1}, {3, 4}, {5, 6}}]

Out[2]=  {9, 9, 9, 9, 9, 4, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5}
```

4. The `MergeSort` function defined in this section becomes quite slow for moderately sized lists. Perform some experiments to determine if the bottleneck is caused mostly by the auxiliary `merge` function or the double recursion inside `MergeSort` itself. Once you have identified the cause of the problem, try to rewrite `MergeSort` to overcome the bottleneck issues.

7.4 Solutions

1. Perhaps the most straightforward way to do this is to write an auxiliary function that takes the output from `runEncode` and produces output such as `Split` would generate.

```
In[1]:=  runEncode[{}] := {}
         runEncode[{x_}] := {{x, 1}}
In[3]:=  runEncode[{x_, res___}] := Module[{R = runEncode[{res}], p},
            p = First[R];
            If[x == First[p],
             Join[{{x, p[[2]] + 1}}, Rest[R]],
             Join[{{x, 1}}, R]]]
```

Then our `split` simply operates on the output of `runEncode`. The iterator for the `Table` is the second element in each sublist, that is, the frequency.

```
In[4]:=  sp[lis_] := Map[Table[#[[1]], {#[[2]]}] &, lis]

In[5]:=  sp[{{3, 2}, {4, 1}, {2, 5}}]

Out[5]=  {{3, 3}, {4}, {2, 2, 2, 2, 2}}

In[6]:=  split[lis_] := sp[runEncode[lis]]

In[7]:=  split[{9, 9, 9, 9, 9, 4, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5}]

Out[7]=  {{9, 9, 9, 9, 9}, {4}, {3, 3, 3, 3}, {5, 5, 5, 5, 5, 5}}
```

Check against the built-in function.

```
In[8]:=  Split[{9, 9, 9, 9, 9, 4, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5}]

Out[8]=  {{9, 9, 9, 9, 9}, {4}, {3, 3, 3, 3}, {5, 5, 5, 5, 5, 5}}
```

2. The order of this list of rules is the order in which the *Mathematica* evaluator will search for a pattern match.

```
In[9]:=  runEncode[{}] := {}
         runEncode[{x_, r___}] := runEncode[x, 1, {r}]
         runEncode[x_, k_, {}] := {{x, k}}
         runEncode[x_, k_, {x_, r___}] := runEncode[x, k + 1, {r}]
         runEncode[x_, k_, {y_, r___}] := Join[{{x, k}}, runEncode[y, 1, {r}]]
```

3. Recursion is on the tail.

```
In[14]:= runDecode[{}] := {}
         runDecode[{{x_, k_}, r___}] :=
          Join[Table[x, {k}], runDecode[{r}]]

In[16]:= runDecode[{{9, 5}, {4, 1}, {3, 4}, {5, 6}}]

Out[16]= {9, 9, 9, 9, 9, 4, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5}
```

8 Numerics

8.1 *Numbers in Mathematica*

1. Define a function `complexToPolar` that converts complex numbers to their polar representations. Then, convert the numbers $3 + 3i$ and $e^{\pi i/3}$ to polar form.
2. Using the built-in `Fold` function, write a function `convert[lis, b]` that accepts a list of digits in any base b (less than 20) and converts it to a base 10 number. For example, 1101_2 is 13 in base 10, so your function should handle this as follows:

```
In[1]:= convert[{1, 1, 0, 1}, 2]
Out[1]= 13
```

3. Create a function to compute the sum of the digits of any integer. Write an additional rule to give the sum of the base-*b* digits of an integer. Then use your function to compute the *Hamming weight* of any integer: the Hamming weight of an integer is given by the number of 1s in the binary representation of that number.
4. Write a function `sumsOfCubes[n]` that takes a positive integer argument n and computes the sums of cubes of the digits of n (Hayes 1992).
5. Use `NestList` to iterate the process of summing cubes of digits, that is, generate a list starting with an initial integer of the successive sums of cubes of digits. For example, starting with 4, the list should look like: `{4, 64, 280, 520, 133, …}`. Note, $64 = 4^3$, $280 = 6^3 + 4^3$, etc. Extend the list for at least 15 values and make an observation about any patterns you notice. Experiment with other starting values.
6. Binary shifts arise in the study of computer algorithms because they often allow you to speed up calculations by operating in base 2 or in bases that are powers of 2. Try to discover what a binary shift does by performing the following shift on 24 (base 10). First get the integer digits of 24 in base 2.

```
In[2]:= IntegerDigits[24, 2]
Out[2]= {1, 1, 0, 0, 0}
```

Then, do a binary shift, one place to the right.

```
In[3]:= RotateRight[%]
Out[3]= {0, 1, 1, 0, 0}
```

Finally, construct an integer from these binary digits and convert back to base 10.

```
In[4]:= FromDigits[%, 2]
Out[4]= 12
```

Experiment with other numbers (including both odd and even integers) and make some conjectures.

7. The `survivor[n]` function from Section 5.8 can be programmed using binary shifts. This can be done by rotating the base 2 digits of the number *n* by one unit to the left and then converting this rotated list back to base 10. For example, if $n = 10$, the base 2 representation is 1010_2; the binary shift gives 0101_2; converting this number back to base 10 gives 5, which is the output to `survivor[5]`. Program a new `survivor` function using the binary shift.

8. Using the `Dice` function from Exercise 9 in Section 4.2, create a function `RollDice[]` that "rolls" two dice and displays them side-by-side. Then create an additional rule, `RollDice[n]`, that rolls a pair of dice *n* times and displays the result in a list or row.

9. Create functions `walk2D` and `walk3D` that generate two-dimensional and three-dimensional lattice walks, respectively. For example, the two-dimensional case can use compass directions north, south, east, west that are represented by the list `{{0, 1}, {0, -1}, {1, 0}, {-1, 0}}`.

10. A surprisingly simple pseudorandom number algorithm is the *linear congruential* method. It is quite easy to implement and has been studied extensively. Sequences of random numbers are generated by a formula such as the following:

$$x_{n+1} = x_n\, b + 1 \pmod{m}.$$

The starting value x_0 is the *seed*, *b* is the *multiplier*, and *m* is the *modulus*. Recall that 7 mod 5 is the remainder upon dividing 7 by 5.

```
In[5]:=  Mod[7, 5]

Out[5]=  2
```

Implement the linear congruential method and test it with a variety of numbers *m* and *b*. If you find that the generator gets in a loop easily, try a large value for the modulus *m*. See Knuth (1997) for a full treatment of random number generating algorithms.

11. Implement a *quadratic congruential* random number generator. The iteration is given by the following, where *a*, *b*, and *c* are the parameters, *m* is the modulus, and x_0 is the starting value:

$$x_{n+1} = \left(a\, x_n^2 + b\, x_n + c\right) \bmod m$$

12. John von Neumann, considered by many to be the "father of computer science," suggested a random number generator known as the *middle-square* method. Starting with a ten-digit integer, square the initial integer and then extract its middle ten digits to get the next number in the sequence. For example, starting with 1234567890, squaring it produces 1524157875019052100. The middle digits are 1578750190, so the sequence starts out 1234567890, 1578750190, 4521624250, …. Implement a middle square random number generator and then test it on a 1000-number sequence. Was the "father of computer science" a good random number generator?

13. Information theory, as conceived by Claude Shannon in the 1940s and 1950s, was originally interested in maximizing the amount of data that can be stored and retrieved over some channel such as a telephone line. Shannon devised a measure, now called the *entropy*, that gives the theoretical maxima for such a signal. Entropy can be thought of as the average uncertainty of a single random variable and is computed by the following, where $p(x)$ is the probability of event *x* over a domain *X*:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$

Generate a plot of the entropy (built into *Mathematica* as `Entropy`) as a function of success probability. You can simulate *n* trials of a coin toss with probability *p* using:

```
RandomVariate[BernoulliDistribution[p], n]
```

See (Manning and Schütze 1999) for a discussion of entropy in the context of information theory generally and in natural language processing in particular. Also, see Claude Shannon's very readable original paper on the mathematical theory of communication (Shannon 1948).

8.1 Solutions

1. This function gives the polar form as a list consisting of the magnitude and the polar angle.

```
In[1]:= complexToPolar[z_] := {Abs[z], Arg[z]}

In[2]:= complexToPolar[3 + 3 i]

Out[2]= {3 √2, π/4}

In[3]:= complexToPolar[e^(π i/3)]

Out[3]= {1, π/3}
```

2. This function uses a default value of 2 for the base. (Try replacing `Fold` with `FoldList` to see more clearly what this function is doing.)

```
In[4]:= convert[digits_List, base_ : 2] := Fold[(base #1 + #2) &, 0, digits]
```

Here are the digits for 9 in base 2:

```
In[5]:= IntegerDigits[9, 2]

Out[5]= {1, 0, 0, 1}
```

This converts them back to the base 10 representation.

```
In[6]:= convert[%]

Out[6]= 9
```

Note, this functionality is built into the function `FromDigits[`*lis*`,` *base*`]`.

```
In[7]:= FromDigits[{1, 0, 0, 1}, 2]

Out[7]= 9
```

This function is essentially an implementation of Horner's method for fast polynomial multiplication.

```
In[8]:=  convert[{a, b, c, d, e}, x]

Out[8]=  e + x (d + x (c + x (b + a x)))

In[9]:=  Expand[%]
```

Out[9]= $e + d\,x + c\,x^2 + b\,x^3 + a\,x^4$

3. One rule can cover both parts of this exercise, using a default value of 10 for the base.

```
In[10]:= DigitSum[n_, base_: 10] := Total[IntegerDigits[n, base]]

In[11]:= DigitSum[10!]

Out[11]= 27
```

The Hamming weight of a number is the number of 1s in its binary representation.

In[12]:= `DigitSum[`2^{31}` - 1, 2]`

Out[12]= `31`

Here is a comparison with a built-in function:

In[13]:= `DigitCount[`2^{31}` - 1, 2, 1]`

Out[13]= `31`

4. Here is the `sumsOfCubes` function.

In[16]:= `sumsOfCubes[n_Integer] := Total[IntegerDigits[n]`^{3}`]`

5. Here is the function that performs the iteration.

```
In[17]:= sumsOfSums[n_Integer, iter_] := NestList[sumsOfCubes, n, iter]
```

We see that the number 4 enters into a cycle.

```
In[18]:= sumsOfSums[4, 12]
```

In fact, it appears as if many initial values enter cycles.

```
In[19]:= sumsOfSums[32, 12]

In[20]:= sumsOfSums[7, 12]

In[21]:= sumsOfSums[372, 12]
```

7. Using the number 100 as an example, first get the base two digits.

```
In[22]:= IntegerDigits[100, 2]
```

Perform a binary shift of one unit (actually, the 1 in `RotateLeft` is not needed here as this is the default value to shift by).

```
In[23]:= l = RotateLeft[IntegerDigits[100, 2], 1]
```

This converts back from base 2 to base 10 (using the `convert` function from Exercise 2).

```
In[24]:= convert[l, 2]
```

Now we can put all this code together to make the `survivor` function.

```
In[25]:= survivor[n_] := Module[{p},
           p = RotateLeft[IntegerDigits[n, 2]];
           Fold[(2 #1 + #2) &, 0, p]]

In[26]:= survivor[100]
```

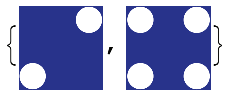
You could of course do the same thing without the symbol p, but it is just a bit less readable.

```
In[27]:= survivor2[n_Integer] :=
          Fold[(2 #1 + #2) &, 0, RotateLeft[IntegerDigits[n, 2]]]

In[28]:= survivor2[100]
```

8. Mapping `Dice` (from Exercise 9 in Section 4.2) over a list of two random integers between 1 and 6 simulates a roll of a pair of dice.

```
In[14]:= Map[Dice, RandomInteger[{1, 6}, {2}]]
```

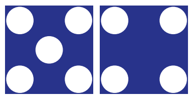
Out[14]=



Here is a function to do that.

```
In[15]:= RollDice[] := GraphicsRow[Map[Dice, RandomInteger[{1, 6}, {2}]]]

In[16]:= RollDice[]
```

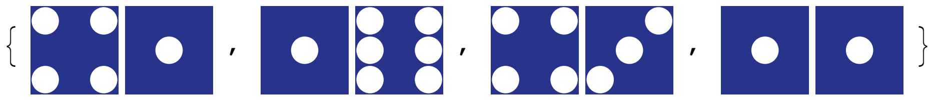
Out[16]=



And here here is the rule for rolling the pair of dice *n* times.

```
In[17]:= RollDice[n_] := Table[RollDice[], {n}]

In[18]:= RollDice[4]
```

Out[18]=



9. Using the hint in the exercise, here are the directions for the two- and three-dimensional cases.

```
In[19]:= NSEW = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

In[20]:= NSEW3 =
           {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {-1, 0, 0}, {0, -1, 0}, {0, 0, -1}};
```

The walk functions follow directly from the one-dimensional case given in the text.

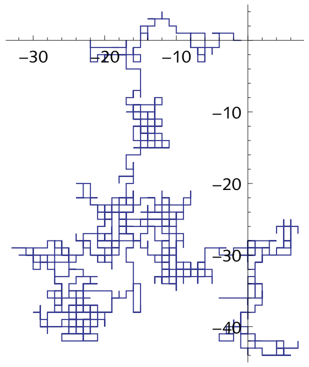
```
In[21]:=  walk2D[t_] := Accumulate[RandomChoice[NSEW, t]]
```

```
In[22]:=  walk3D[t_] := Accumulate[RandomChoice[NSEW3, t]]
```

Exercise the functions and visualize.

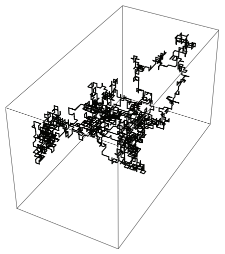
```
In[23]:=  ListLinePlot[walk2D[1500], AspectRatio → Automatic]
```

Out[23]=



```
In[24]:=  Graphics3D[Line[walk3D[2500]]]
```

Out[24]=



For a more complete discussion of these functions, see Section 13.1.

10. Here is the linear congruential generator.

```
In[25]:=  linearCongruential[x_, mod_, mult_, incr_] := Mod[mult x + incr, mod]
```

With modulus 100 and multiplier 15, this generator quickly gets into a cycle.

```
In[26]:=  NestList[linearCongruential[#, 100, 15, 1] &, 5, 10]

Out[26]=  {5, 76, 41, 16, 41, 16, 41, 16, 41, 16, 41}
```

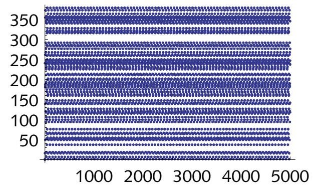
With a larger modulus and multiplier, it appears as if this generator is doing better.

Here are the first 60 terms starting with a seed of 0.

```
In[27]:= data = NestList[linearCongruential[#, 381, 15, 1] &, 0, 5000];
         Take[data, 60]

Out[28]= {0, 1, 16, 241, 187, 139, 181, 49, 355, 373, 262, 121, 292, 190, 184,
          94, 268, 211, 118, 247, 277, 346, 238, 142, 226, 343, 193, 229, 7, 106,
          67, 244, 232, 52, 19, 286, 100, 358, 37, 175, 340, 148, 316, 169, 250,
          322, 259, 76, 379, 352, 328, 349, 283, 55, 64, 199, 319, 214, 163, 160}
```

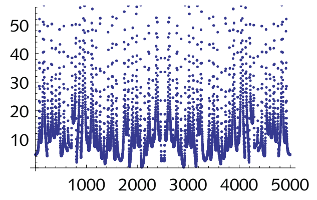
Sometimes it is hard to see if your generator is doing a poor job. Graphical analysis can help by allowing you to see patterns over larger domains. Here is a `ListPlot` of this sequence taken out to 5000 terms.

```
In[29]:= ListPlot[data, PlotStyle → PointSize[.005]]
```

Out[29]=


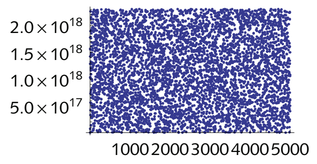
It appears as if certain numbers are repeating. Looking at the plot of the Fourier data shows peaks at certain frequencies, indicating a periodic nature to the data.

```
In[30]:= ListPlot[Abs[Fourier[data]], PlotStyle → PointSize[.005]]
```

Out[30]=


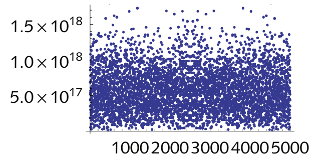
Using a much larger modulus and multiplier and an increment of zero (actually, these are the default values for *Mathematica*'s built-in `"Congruential"` method for `SeedRandom`), you can keep your generator from getting in such short loops.

```
In[31]:= ListPlot[
          data = NestList[linearCongruential[#1, 2 305 843 009 213 693 951,
                 1 283 839 219 676 404 755, 0] &,
            1, 5000], PlotStyle → PointSize[.005]]
```

Out[31]=


```
In[32]:= ListPlot[Abs[Fourier[data]], PlotStyle → PointSize[.005]]
```

Out[32]=



12. Here is a simple implementation of the middle square method. It assumes a ten-digit seed. To work with arbitrary-length seeds, modify the number of middle digits that are extracted with the `Part` function.

```
In[33]:= middleSquareGenerator[n_, seed_: 1 234 567 890] :=
          Module[{tmp = {seed}, s2, len, s = seed},
           Do[
            s2 = IntegerDigits[s^2];
            len = Length[s2];
            s = FromDigits[If[len < 20, PadLeft[s2, 20, 0], s2][[6 ;; 15]]];
            AppendTo[tmp, s],
            {n}];
           tmp
          ]

In[34]:= middleSquareGenerator[3]

Out[34]= {1 234 567 890, 1 578 750 190, 4 521 624 250, 858 581 880}

In[35]:= data = middleSquareGenerator[1000];
         Take[data, 12]

Out[36]= {1 234 567 890, 1 578 750 190, 4 521 624 250, 858 581 880,
          1 628 446 643, 8 384 690 979, 428 133 239, 2 980 703 366,
          5 925 560 837, 2 712 329 881, 7 333 833 654, 1 160 645 429}
```

13. Run 10 000 trials with a range of probabilities from 0 to 1 in increments of .001.

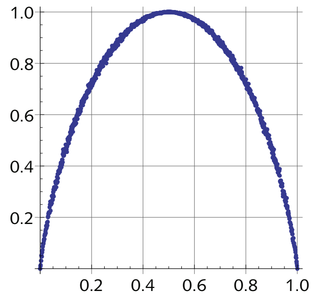
```
In[37]:= incr = 0.001;
         trials = 10 000;
         lis = Table[RandomVariate[
             BernoulliDistribution[p], trials], {p, 0, 1, incr}];
```

Pair up the probabilities with the entropies (in base 2) for each trial.

```
In[40]:= info = Transpose[{Range[0, 1, incr], Map[Entropy[2, #] &, lis]}];
```

Make a plot.

```
In[41]:= ListPlot[info, AspectRatio → 1,
          GridLines → Automatic, PlotStyle → PointSize[Small]]

Out[41]=
```



8.2 *Numerical computation*

1. Explain why *Mathematica* is unable to produce a number with 100 digits of precision in the following example.

```
In[1]:= N[1.23, 100]

Out[1]= 1.23

In[2]:= Precision[%]

Out[2]= MachinePrecision
```

2. Determine what level of precision is necessary when computing `N`$\left[\sqrt{2}\text{ , } prec\right]^{200}$ to produce accuracy in the output of at least 100 digits.

3. Explain why the following computation produces an unexpected result (that is, why the value 0.000000000001 is not returned).

```
In[3]:= 1.0 - 0.999999999999

Out[3]= 9.99978 × 10^-13
```

4. How close is the number $e^{\pi\sqrt{163}}$ to an integer? Use `N`, but mind the precision of your computations.

8.2 *Solutions*

1. The number 1.23 has machine precision.

```
In[1]:= Precision[1.23]

Out[1]= MachinePrecision
```

Asking *Mathematica* to generate 100 digits of precision from a number that only contains about 16 digits of precision would require it to produce 84 digits without any information about where those digits should come from.

2. This generates a table showing the number of digits of precision needed in the input compared with the accuracy of the result.

```
In[2]:= Table[{x, Accuracy[N[√2, x]^200 - (√2)^200]},
          {x, 100, 140, 5}] // TableForm
```

```
                  100    67.596
                  105    72.596
                  110    77.596
                  115    82.596
Out[2]//TableForm= 120    87.596
                  125    92.596
                  130    97.596
                  135    102.596
                  140    107.596
```

8.3 *Arrays of numbers*

1. Create a function `RandomSparseArray[n]` that generates an $n \times n$ sparse array with random numbers along the diagonal.
2. Write a function `TridiagonalMatrix[n, p, q]` that creates an $n \times n$ matrix with the integer p on the diagonal, the integer q on the upper and lower subdiagonals, and 0s everywhere else.
3. Create a vector `vec` consisting of 100 000 random real numbers between 0 and 1. Check that it is indeed a packed array by using ``Developer`PackedArrayQ``. Then replace one element in `vec` with an integer. Check that this new vector is not a packed array. Finally, perform some memory and timing tests on these two vectors, using functions such as `Max`, `Norm`, `RootMeanSquare`.
4. An interesting computation of the Fibonacci numbers can be obtained using the determinant of a certain tri-diagonal matrix: 1s on the diagonal and $i = \sqrt{-1}$ running along each subdiagonal. For example, the following 4×4 matrix has determinant equal to the fifth Fibonacci number.

```
In[1]:= | 1 i 0 0 |
        | i 1 i 0 |
        | 0 i 1 i |
        | 0 0 i 1 |

Out[1]= 5
```

   Create a function that computes the nth Fibonacci number using a sparse array implementation of this tri-diagonal matrix. You will need special rules for $n = 1$ and $n = 2$.

5. An efficient approach to computing large Fibonacci numbers relies upon the observation that a certain matrix has its characteristic polynomial equal to the characteristic equation for the Fibonacci numbers.

```
In[2]:= mat = (1 1
               1 0);
        poly = CharacteristicPolynomial[mat, x]

Out[3]= -1 - x + x^2
```

```
In[4]:= Solve[poly == 0, x]
```

Out[4]= $\left\{\left\{x \to \frac{1}{2}\left(1-\sqrt{5}\right)\right\}, \left\{x \to \frac{1}{2}\left(1+\sqrt{5}\right)\right\}\right\}$

The Fibonacci numbers F_n can be generated from successive powers of this matrix.

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Use these facts to implement an algorithm for computing the Fibonacci numbers using the built-in `MatrixPower` function with sparse arrays.

8.3 *Solutions*

1. Note the need for a delayed rule in this function.

```
In[1]:= RandomSparseArray[n_Integer] :=
          SparseArray[{Band[{1, 1}] :> RandomReal[]}, {n, n}]

In[2]:= Normal[RandomSparseArray[5]] // MatrixForm

Out[2]//MatrixForm=
        0.778393     0          0          0          0
           0      0.685614      0          0          0
           0         0       0.639995      0          0
           0         0          0       0.79101       0
           0         0          0          0       0.427544
```

2. Here is the definition of `TridiagonalMatrix`.

```
In[3]:= TridiagonalMatrix[n_, p_, q_] := SparseArray[
           {Band[{1, 1}] → p, Band[{1, 2}] → q, Band[{2, 1}] → q}, {n, n}]

In[4]:= TridiagonalMatrix[5, α, β]

Out[4]= SparseArray[<13>, {5, 5}]

In[5]:= Normal[%] // MatrixForm

Out[5]//MatrixForm=
        α  β  0  0  0
        β  α  β  0  0
        0  β  α  β  0
        0  0  β  α  β
        0  0  0  β  α
```

3. First we create the packed array vector.

```
In[6]:= vec = RandomVariate[NormalDistribution[1, 3], {10^5}];

In[7]:= Developer`PackedArrayQ[vec]

Out[7]= True
```

Replacing the first element in `vec` with a 1 gives an expression that is not packed.

```
In[8]:=  newvec = ReplacePart[vec, 1, 1];

In[9]:=  Developer`PackedArrayQ[newvec]

Out[9]=  False
```

The size of the unpacked object is about four times larger than the packed array.

```
In[10]:=  Map[ByteCount, {vec, newvec}]

Out[10]=  {800 168, 3 200 040}
```

Sorting the packed object is about three times faster than sorting the unpacked object.

```
In[11]:=  Timing[Do[Sort[vec], {5}]]

Out[11]=  {0.094712, Null}

In[12]:=  Timing[Do[Sort[newvec], {5}]]

Out[12]=  {0.243469, Null}
```

Finding the minimum element is about one order of magnitude faster with the packed array.

```
In[13]:=  Timing[Min[vec];]

Out[13]=  {0.000213, Null}

In[14]:=  Timing[Min[newvec];]

Out[14]=  {0.002076, Null}
```

4. Since the definition involving determinants only makes sense for $n > 2$, we include a condition on the left-hand side of that definition and also specific rules for the cases $n = 1, 2$.

```
In[15]:=  fibMat[n_ /; n > 2] :=
           Det@SparseArray[{Band[{1, 1}] → 1, Band[{2, 1}] → ⅈ,
              Band[{1, 2}] → ⅈ}, {n - 1, n - 1}]

In[16]:=  fibMat[1] = fibMat[2] = 1;

In[17]:=  Table[fibMat[i], {i, 1, 20}]

Out[17]=  {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,
           144, 233, 377, 610, 987, 1597, 2584, 4181, 6765}
```

The computation of determinants using decomposition methods is on the order of $O(n^3)$ computational complexity. So this computation tends to slow down considerably for large n.

```
In[18]:= Timing[fibMat[10^3]]

Out[18]= {6.7663,
          43 466 557 686 937 456 435 688 527 675 040 625 802 564 660 517 371 780 402 ⋱
            481 729 089 536 555 417 949 051 890 403 879 840 079 255 169 295 922 593 080 ⋱
            322 634 775 209 689 623 239 873 322 471 161 642 996 440 906 533 187 938 298 ⋱
            969 649 928 516 003 704 476 137 795 166 849 228 875}

In[19]:= Fibonacci[10^3] == fibMat[10^3]

Out[19]= True
```

5. The sparse array needs only one rule `{2, 2} → 0` together with a third argument that specifies the default values should be set to 1. Then pick off the *n*th Fibonacci number in the first row, second column.

```
In[20]:= fibMat2[n_] := Module[{mat},
           mat = SparseArray[{{2, 2} → 0}, {2, 2}, 1];
           MatrixPower[mat, n][[1, 2]]
          ]
```

Quick check of the first few numbers.

```
In[21]:= Table[fibMat2[n], {n, 1, 10}]

Out[21]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

The time to compute a large number is quite fast.

```
In[22]:= Timing[fibMat2[10^3]]

Out[22]= {0.000216,
          43 466 557 686 937 456 435 688 527 675 040 625 802 564 660 517 371 780 402 ⋱
            481 729 089 536 555 417 949 051 890 403 879 840 079 255 169 295 922 593 080 ⋱
            322 634 775 209 689 623 239 873 322 471 161 642 996 440 906 533 187 938 298 ⋱
            969 649 928 516 003 704 476 137 795 166 849 228 875}
```

Check correctness against the built-in function, using a large random integer *n*.

```
In[23]:= With[{n = RandomInteger[10^6]},
          fibMat2[n] == Fibonacci[n]
         ]

Out[23]= True
```

8.4 *Examples and applications*

1. Write a functional implementation of the secant method. Your function should accept as arguments the name of a function and two initial guesses. It should maintain the precision of the inputs and it should output the root at the precision of the initial guess, and the number of iterations required to compute the root. Consider using the built-in functions `FixedPoint` or `Nest`.

2. The `findRoot` function developed in this section suffers from several inefficiencies. One of them is that if the precision goal is no more than machine precision, all intermediate computations should be done at the more efficient machine precision as well. Modify `findRoot` so that it will operate at machine precision if the precision goal is at most machine precision.

3. In the `findRoot` program, we added `SetPrecision[`*result*`,` *precisionGoal*`]` at the very end to return the final result at the precision goal, but we have done no test to insure that the result meets the required precision. Add a test to the end of the `findRoot` function so that, if this condition is not met, an error message is generated and the current result is output.

4. Some functions tend to cause root-finding methods to converge rather slowly. For example, the function $f(x) = \sin(x) - x$ requires over ten iterations of Newton's method with an initial guess of $x_0 = 0.1$ to get three-place accuracy.

```
In[1]:= FindRoot[Sin[x] - x, {x, 0.1},
          MaxIterations → 12, EvaluationMonitor :→ Sow[x]] // Reap

        FindRoot::cvmit : Failed to converge to the requested accuracy or precision within 12 iterations. ≫

Out[1]= {{x → 0.000770503},
          {{0.1, 0.0666556, 0.0444337, 0.0296215, 0.0197474,
            0.0131648, 0.00877654, 0.00585102, 0.00390068,
            0.00260045, 0.00173363, 0.00115576, 0.000770503}}}
```

Implement the following acceleration of Newton's method and determine how many iterations of the function $f(x) = \sin(x) - x$, starting with $x_0 = 0.1$, are necessary for six-place accuracy.

$$\text{accelNewton}(x) = \frac{f(x)\, f'(x)}{\left[f'(x)\right]^2 - f(x)\, f''(x)}$$

This accelerated method is particularly useful for functions with multiple roots.

5. The *norm* of a matrix gives some measure of the size of that matrix. The norm of a matrix A is indicated by $\|A\|$. There are numerous matrix norms, but all share certain properties. For $n \times n$ matrices A and B:

(i.) $\|A\| \geq 0$;

(ii.) $\|A\| = 0$ if and only if A is the zero matrix;

(iii.) $\|c\, A\| = |c|\; \|A\|$ for any scalar c;

(iv.) $\|A + B\| = \|A\| + \|B\|$;

(v.) $\|A\, B\| \leq \|A\|\; \|B\|$.

One particularly useful norm is the l_∞ norm, sometimes referred to as the *max norm*. For a vector, this is defined as

$$\|\vec{x}\|_\infty = \max_{1 \leq i \leq n} |x_i| \,.$$

The corresponding matrix norm is defined similarly. Hence, for a matrix $A = a_{ij}$, we have

$$\|A\|_\infty = \max_{1\le i\le n} \sum_{j=1}^{n} \left| a_{ij} \right| .$$

This computes the sum of the absolute values of the elements in each row, and then takes the maximum of these sums, that is, the l_∞ matrix norm is the max of the l_∞ norms of the rows.

Write a function `norm[mat, ∞]` that takes a square matrix as an argument and outputs its $\|\cdot\|_\infty$ norm. Compare your function with the built-in `Norm` function. Include rules for the l_2 and l_1 norms.

6. If a matrix A is nonsingular (invertible), then its *condition number* is defined as $\|A\| \cdot \left\|A^{-1}\right\|$. A matrix is called *well-conditioned* if its condition number is close to 1, the condition number of the identity matrix. A matrix is called *ill-conditioned* if its condition number is significantly larger than 1.

   Write a function `conditionNumber[mat]` that uses `norm` defined in the previous exercise or the built-in `Norm` function and outputs the condition number of *mat*. Use `conditionNumber` to compute the condition number of the first ten Hilbert matrices.

7. Create a function `LagPlot[data, lag]` that plots *data* (a one-dimensional vector) against the data lagged by a displacement, *lag*. For example, if *lag* = 1, then `LagPlot` would display values $\{x_{i-1}, x_i\}$. Use NIST's lew.dat which consists of 200 observations of beam deflection data and whose lag plot indicates a lack of randomness in the sequence of numbers. You can import and post-process the data using the following:

```
In[2]:= data = Import[
           "http://itl.nist.gov/div898/education/eda/lew.dat", "Data"];
        Short[lewdata = Cases[data, {x_?NumberQ} :> x]]

Out[3]//Short= {-213, -564, -35, -15, 141, <<190>>, -385, 198, -218, -536, 96}
```

   Or, if you have the files associated with this book, use something like the following:

```
In[4]:= lewdata = Import[FileNameJoin[
            {NotebookDirectory[], "Data", "lew.dat"}], "List"];
```

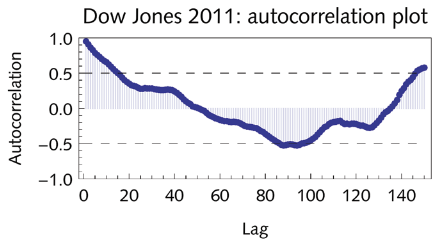
8. Modify the `Correlogram` function developed in this section to provide for an option, `Coefficient`, that sets the range of values for the dashed lines within which the autocorrelation coefficients are hoped to lie. In addition, set things up so `Correlogram` inherits all the options of `ListPlot`.

   Then use your function to look at some time-series data, such as that below; the plot here shows a high degree of autocorrelation for small time lags, but less so for larger lags, suggesting a serial dependence in the data. In finance, autocorrelation analysis (usually referred to as serial correlation) is used to predict how price movements may be affected by each other.

```
In[5]:= data =
          FinancialData["^DJI", {{2011, 1, 1}, {2011, 12, 31}}, "Value"];
```

```
In[6]:= Correlogram[data, {1, 150}, Coefficient → 0.5,
         Filling → Axis, PlotRange → {-1, 1},
         PlotLabel → Style["Dow Jones 2011: autocorrelation plot", 10],
         FrameLabel → {{"Autocorrelation", None}, {"Lag", None}}]
```

Out[6]=



9. Create random walks on the binary digits of π. For a one-dimensional walk, use `RealDigits[num, 2]` to get the base 2 digits and then convert each 0 to −1 so that you have a vector of ±1s for the step directions; then use `Accumulate`. For the two-dimensional walk, use `Partition` to pair up digits and then use an appropriate transformation to have the four pairs, `{0, 0}`, `{0, 1}`, `{1, 0}`, and `{1, 1}` map to the compass directions; then use `Accumulate`. See Bailey et al. (2012) for more on visualizing digits of π.

8.4 Solutions

1. We will overload `findRoot` to invoke the secant method when given a list of two numbers as the second argument.

```
In[1]:= Options[findRoot] = {
           MaxIterations :→ $RecursionLimit,
           PrecisionGoal → Automatic,
           WorkingPrecision → Automatic
          };

In[2]:= findRoot[fun_, {var_, x1_?NumericQ, x2_?NumericQ},
          OptionsPattern[]] := Module[{maxIterations, precisionGoal,
           workingPrecision, initx, df, next, result},
          {maxIterations, precisionGoal, workingPrecision} = OptionValue[
            {MaxIterations, PrecisionGoal, WorkingPrecision}];
          If[precisionGoal === Automatic,
           precisionGoal = Min[{Precision[x1], Precision[x2]}]];
          If[workingPrecision === Automatic,
           workingPrecision = precisionGoal + 10];
          initx = SetPrecision[{x1, x2}, workingPrecision];
          df[a_, b_] := (fun[b] - fun[a]) / (b - a);
          next[{a_, b_}] := {a, b - fun[b]/df[a, b]};
          result = SetPrecision[
            FixedPoint[next, initx, maxIterations][[2]], precisionGoal];
          {var → result}]
In[3]:=  f[x_] := x^2 - 2

In[4]:=  findRoot[f, {x, 1., 2.}]

Out[4]=  {x → 1.41421}

In[5]:=  findRoot[f, {x, 1.0`60, 2.0`50}]

Out[5]=  {x → 1.4142135623730950488016887242096980785696740946953}

In[6]:=  Precision[%]

Out[6]=  50.
```

5. Here is a three-dimensional vector.

```
In[7]:=  vec = {1, -3, 2};
```

This computes the l_∞ norm of the vector.

```
In[8]:=  norm[v_?VectorQ, l_: Infinity] := Max[Abs[v]]

In[9]:=  norm[vec]

Out[9]=  3
```

Compare this with the built-in `Norm` function.

```
In[10]:=  Norm[vec, Infinity]

Out[10]=  3
```

Here is a 3×3 matrix.

```
In[11]:=  mat = {{1, 2, 3}, {1, 0, 2}, {2, -3, 2}};
```

Here, then, is the matrix norm.

```
In[12]:=  norm[m_?MatrixQ, l_: Infinity] :=
            norm[Total[Abs[Transpose[m]]], Infinity]

In[13]:=  norm[mat]

Out[13]=  7
```

Again, here is a comparison with the built-in `Norm` function.

```
In[14]:=  Norm[mat, Infinity]

Out[14]=  7
```

Notice how we *overloaded* the definition of the function `norm` so that it would act differently depending upon what type of argument it was given. This is a particularly powerful feature of *Mathematica*. The expression `_?MatrixQ` on the left-hand side of the definition causes the function `norm` to use the definition on the right-hand side *only if* the argument is in fact a matrix (if it passes the `MatrixQ` test). If that argument is a vector (if it passes the `VectorQ` test), then the previous definition is used.

6. Here is the function to compute the condition number of a matrix (using the l_2 norm).

```
In[15]:= conditionNumber[m_?MatrixQ] := Norm[m, 2] Norm[Inverse[m], 2]

In[16]:= conditionNumber[HilbertMatrix[3]] // N

Out[16]= 524.057
```

Compare this with the condition number of a random matrix.

```
In[17]:= mat = RandomInteger[5, {3, 3}];
         conditionNumber[mat] // N

Out[18]= 2.31709
```

An alternative definition for the condition number of a matrix is the ratio of largest to smallest singular value.

```
In[19]:= N@SingularValueList[mat]

Out[19]= {6.37231, 4.22261, 2.75013}

In[20]:= First[%] / Last[%]

Out[20]= 2.31709

In[21]:= conditionNumber2[mat_?MatrixQ] :=
          Module[{sv = SingularValueList[mat]},
           First[sv] / Last[sv]]

In[22]:= conditionNumber2[mat] // N

Out[22]= 2.31709
```

7. Pairing up values with preceding values is accomplished by transposing the appropriate lists.

```
Transpose[{Drop[data, lag], Drop[data, -lag]}]
```

Here then is the code for `LagPlot`.

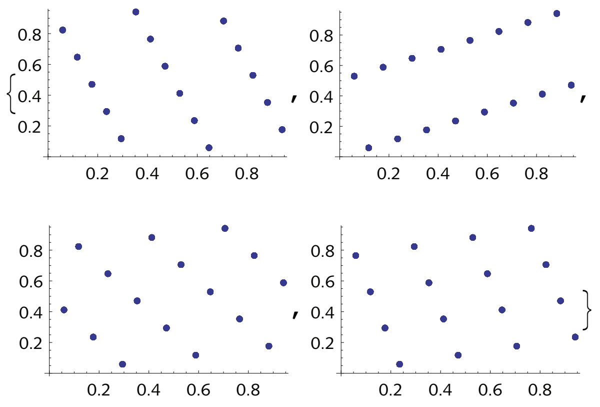
```
In[23]:= LagPlot[data_, lag_ : 1, opts : OptionsPattern[ListPlot]] :=
          ListPlot[Transpose[{Drop[data, lag], Drop[data, -lag]}], opts]
```

Trying it out on a sequence of “random” numbers generated using a linear congruential generator shows patterns that indicate a very low likelihood of randomness in the sequence.

```
In[24]:= data = BlockRandom[SeedRandom[1, Method → {"Congruential",
               "Multiplier" → 11, "Increment" → 0, "Modulus" → 17}];
            RandomReal[1, {1000}]];
```

```
In[25]:= Table[LagPlot[data, i, ImageSize → Small], {i, 1, 4}]
```

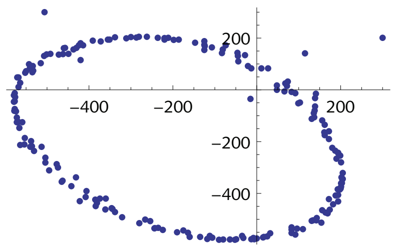
Out[25]=



NIST describes the data in lew.dat as originating "from an underlying single-cycle sinusoidal model."

```
In[26]:= lewdata = Import[FileNameJoin[
             {NotebookDirectory[], "Data", "lew.dat"}], "List"];

In[27]:= LagPlot[lewdata, 1, ImageSize → Small]
```

Out[27]=



8. First, set the options for `Correlogram`, giving a default value for `Coefficient` of 0.05.

```
In[28]:= Options[Correlogram] =
           Join[{Coefficient → 0.05}, Options[ListPlot]];

In[29]:= Correlogram[data_,
           {lagmin_, lagmax_, incr_ : 1}, opts : OptionsPattern[]] :=
          Module[{rh, corrs},
           rh = OptionValue[Coefficient];
           corrs = Table[{lag, AutoCorrelation[data, lag]},
             {lag, lagmin, lagmax, incr}];
           ListPlot[corrs,
            FilterRules[{opts}, Options[ListPlot]], AspectRatio → .4,
            Frame → True, Axes → False, PlotRange → Automatic,
            FrameTicks → {{Automatic, False}, {Automatic, False}},
            Epilog → {Thin, Dashed,
              Line[{{0, rh}, {(lagmax - lagmin + 1) / incr, rh}}],
              Line[{{0, -rh}, {(lagmax - lagmin + 1) / incr, -rh}}]}]
          ]
```

```
In[30]:= AutoCorrelation[data_, lag_: 1] :=
          Correlation[Drop[data, lag], Drop[data, -lag]]
```

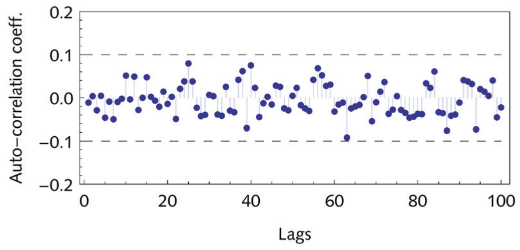
Try out the function on some sinusoidal data with some noise added.

```
In[31]:= data = Table[RandomReal[{-2, 2}]
              Sin[x + RandomReal[{-.25, .25}]], {x, 0, 10 π, .05}];
```

Exercise some of the options.

```
In[32]:= Correlogram[data, {1, 100}, Coefficient → 0.1,
          Filling → Axis, PlotRange → {-0.2, 0.2},
          FrameLabel → {{"Auto-correlation coeff.", None}, {"Lags", None}}]
```

Out[32]=



9. Here are the binary digits of π. `First` is used to get only the digits from `RealDigits`.

```
In[33]:= First[RealDigits[N[Pi, 12], 2]]

Out[33]= {1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0,
          1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0}
```

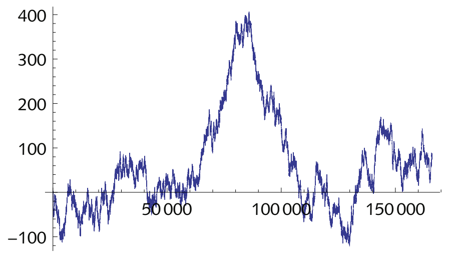
Convert 0s to –1s.

```
In[34]:= 2 % - 1

Out[34]= {1, 1, -1, -1, 1, -1, -1, 1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, -1, 1, 1,
          -1, 1, -1, 1, -1, 1, -1, -1, -1, 1, -1, -1, -1, 1, -1, -1, -1, 1, -1}
```

Here then is a plot for the first fifty thousand digits.

```
In[35]:= ListLinePlot[
          With[{digits = 50 000},
           Accumulate[2 First[RealDigits[N[Pi, digits], 2]] - 1]
          ]]
```

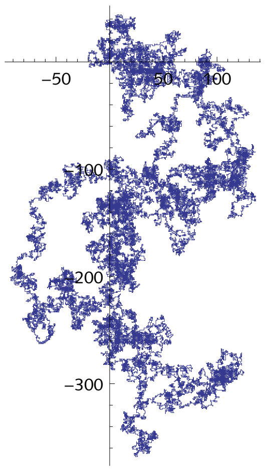
Out[35]=



For the two-dimensional case, use `Partition` to pair up the binary digits, then a transformation rule to convert them to compass directions.

```
In[36]:= With[{digs = First[RealDigits[N[Pi, 50 000], 2]]},
          ListLinePlot[Accumulate[
             Partition[digs, 2, 2] /. {{0, 0} → {-1, 0}, {1, 1} → {0, -1}}],
           AspectRatio → Automatic]]
```

Out[36]=



9 Strings

9.1 *Structure and syntax*

1. Convert the first character in a string (which you may assume to be a lowercase letter) to uppercase.
2. Given a string of digits of arbitrary length, convert it to its integer value. (Hint: you may find that the `Dot` function is helpful.)
3. Create a function `UniqueCharacters[`*str*`]` that takes a string as its argument and returns a list of the unique characters in that string. For example, `UniqueCharacters["Mississippi"]` should return `{M, i, s, p}`.

9.1 *Solutions*

1. Here is a test string we will use for this exercise.

```
In[1]:= str = "this is a test string"
Out[1]= this is a test string
```

This extracts the first character from `str`.

```
In[2]:= StringTake[str, 1]
Out[2]= t
```

Here is its character code.

```
In[3]:= ToCharacterCode[%]
Out[3]= {116}
```

For each lowercase letter of the English alphabet, subtracting 32 gives the corresponding uppercase character.

```
In[4]:= % - 32
Out[4]= {84}
```

Convert back to a character.

```
In[5]:= FromCharacterCode[%]
Out[5]= T
```

Take the original string minus its first character.

```
In[6]:=  StringDrop[str, 1]
Out[6]=  his is a test string
```

Finally, join the previous string with the capital *T*.

```
In[7]:=  StringJoin[%%, %]
Out[7]=  This is a test string
```

You can do this more efficiently using `ToUpperCase` and `StringTake`. This approach is more general in that it does not assume that the first character in your string is lower case.

```
In[8]:=  ToUpperCase[StringTake[str, 1]]
Out[8]=  T

In[9]:=  StringTake[str, 2 ;; -1]
Out[9]=  his is a test string

In[10]:=  ToUpperCase[StringTake[str, 1]] <> StringTake[str, 2 ;; -1]
Out[10]=  This is a test string
```

2. One approach converts the string to character codes.

```
In[11]:=  ToCharacterCode["10495"]

In[12]:=  % - 48

In[13]:=  Table[10^j, {j, 4, 0, -1}]

In[14]:=  %.%%
```

This is a good place to use `Fold`. Using `FoldList`, you can see how the expression is built up.

```
In[15]:=  FoldList[#2 + 10 #1 &, 0, ToCharacterCode["10495"] - 48]
```

Much more directly, use `ToExpression`.

```
In[16]:=  ToExpression["10495"]
```

3. Start by extracting the individual characters in a string.

```
In[11]:=  str = "Mississippi";
          Characters[str]
Out[12]=  {M, i, s, s, i, s, s, i, p, p, i}
```

This gives the set of unique characters in this string.

```
In[13]:=  Union[Characters[str]]
Out[13]=  {i, M, p, s}
```

`Union` sorts the list whereas `DeleteDuplicates` does not.

```
In[14]:= DeleteDuplicates[Characters[str]]

Out[14]= {M, i, s, p}
```

Here then is the function.

```
In[15]:= UniqueCharacters[str_String] := DeleteDuplicates[Characters[str]]
```

Try it out on a more interesting example.

```
In[16]:= protein = ProteinData["PP2672"]

Out[16]= MKSSEELQCLKQMEEELLFLKAGQGSQRARLTPPLPRALQGNFGAPALCGIWFAEHLHPAVGMP⋱
             PNYNSSMLSLSPERTILSGGWSGKQTQQPVPPLRTLLLRSPFSLHKSSQPGSPKASQRIHP⋱
             LFHSIPRSQLHSVLLGLPLLFIQTRPSPPAQYGAQMPLRYICFGPNIFWGSKKPQKE
```

```
In[17]:= UniqueCharacters[protein]

Out[17]= {M, K, S, E, L, Q, C, F, A, G, R, T, P, N, I, W, H, V, Y}
```

It even works in the degenerate case.

```
In[18]:= UniqueCharacters[""]

Out[18]= {}
```

9.2 *Operations on strings*

1. Create a function `PalindromeQ[`*str*`]` that returns a value of `True` if its argument *str* is a palindrome, that is, if the string *str* is the same forward and backward. For example, *refer* is a palindrome.
2. Create a function `StringRotateLeft[`*str*`, `*n*`]` that takes a string *str*, and returns a string with the characters rotated to the left *n* places. For example:

```
In[1]:= StringRotateLeft["a quark for Muster Mark ", 8]

Out[1]= for Muster Mark a quark
```

3. In creating the function `MakeVarList` in this section, we were not careful about the arguments that might be passed. Correct this problem using pattern matching on the arguments to this function to insure that the indices are positive integers only.
4. Create a function `StringPad[`*str*`, {`*n*`}]` that pads the end of a string with *n* whitespace characters. Then create a second rule `StringPad[`*str*`, `*n*`]` that pads the string out to length *n*. If the input string has length greater than *n*, issue a warning message. Finally, mirroring the argument structure for the built-in `PadLeft`, create a third rule `StringPad[`*str*`, `*n*`, `*m*`]` that pads with *n* whitespaces at the front and *m* whitespaces at the end of the string.
5. Modify the Caesar cipher so that it encodes by shifting five places to the right. Include the space character in the alphabet.
6. A mixed-alphabet cipher is created by first writing a keyword followed by the remaining letters of the alphabet and then using this as the substitution (or cipher) text. For example, if the keyword is *django*, the cipher text alphabet would be:

```
djangobcefhiklmpqrstuvwxyz
```

So, *a* is replaced with *d*, *b* is replaced with *j*, *c* is replaced with *a*, and so on. As an example, the piece of text

the sheik of araby

would then be encoded as

tcg scgeh mo drdjy

Implement this cipher and go one step further to output the cipher text in blocks of length five, omitting spaces and punctuation.

7. Modify the alphabet permutation cipher so that instead of being based on single letters, it is instead based on adjacent pairs of letters. The single letter cipher will have $26! = 403\,291\,461\,126\,605\,635\,584\,000\,000$ permutations; the adjacent pairs cipher will have $26^2! = 1.88370768413381 \times 10^{1621}$ permutations.

9.2 *Solutions*

1. Here is the function that checks if a string is a palindrome.

```
In[1]:=  PalindromeQ[str_String] := StringReverse[str] == str

In[2]:=  PalindromeQ["mood"]

Out[2]=  False

In[3]:=  PalindromeQ["PoP"]

Out[3]=  True
```

An argument that is a number is converted to a string and then the previous rule is called.

```
In[4]:=  PalindromeQ[num_Integer] := PalindromeQ[ToString[num]]

In[5]:=  PalindromeQ[12 522 521]

Out[5]=  True
```

Get all words in the dictionary that comes with *Mathematica*.

```
In[6]:=  words = DictionaryLookup[];
```

Select those that pass the `PalindromeQ` test.

```
In[7]:=  Select[words, PalindromeQ]

Out[7]=  {a, aha, aka, bib, bob, boob, bub, CFC, civic, dad, deed, deified, did,
          dud, DVD, eke, ere, eve, ewe, eye, gag, gig, huh, I, kayak, kook,
          level, ma'am, madam, mam, MGM, minim, mom, mum, nan, non, noon,
          nun, oho, pap, peep, pep, pip, poop, pop, pup, radar, redder, refer,
          repaper, reviver, rotor, sagas, sees, seres, sexes, shahs, sis,
          solos, SOS, stats, stets, tat, tenet, TNT, toot, tot, tut, wow, WWW}
```

2. Use the argument structure of `RotateLeft`.

```
In[8]:= StringRotateLeft[str_, n_: 1] :=
          StringJoin[RotateLeft[Characters[str], n]]

In[9]:= StringRotateLeft["squeamish ossifrage", 5]

Out[9]= mish ossifragesquea
```

3. Using a negative index is a problem when the string is converted using `ToExpression`.

```
In[10]:= ToString[x] <> ToString[-2] // FullForm

Out[10]//FullForm= "x-2"

In[11]:= ToExpression[%] // FullForm

Out[11]//FullForm= Plus[-2, x]
```

Argument checking with a different pattern corrects this problem.

```
In[12]:= Clear[MakeVarList]

In[13]:= MakeVarList[x_Symbol, {n_Integer?Positive, m_Integer?Positive}] :=
          ToExpression[Map[ToString[x] <> ToString[#] &, Range[n, m]]]

In[14]:= MakeVarList[tmp, {2, 4}]

Out[14]= {tmp2, tmp3, tmp4}
```

4. First, using `StringJoin`, put *n* spaces at the end of the string.

```
In[15]:= StringPad[str_String, {n_}] := StringJoin[str, Table[" ", {n}]]

In[16]:= StringPad["ciao", {5}] // FullForm

Out[16]//FullForm= "ciao     "
```

For the second rule, first create a message that will be issued if the string is longer than *n*.

```
In[17]:= StringPad::badlen = "Pad length `1` must
             be greater than the length of string `2`.";

In[18]:= StringPad[str_String, n_] :=
          With[{len = StringLength[str]}, If[len > n,
            Message[StringPad::badlen, n, str], StringPad[str, {n - len}]]]

In[19]:= StringPad["ciao", 8] // FullForm

Out[19]//FullForm= "ciao    "

In[20]:= StringPad["ciao", 3]

StringPad::badlen : Pad length 3 must be greater than the length of string ciao.
```

Finally, here is a rule for padding at the beginning and end of the string.

```
In[21]:= StringPad[str_String, n_, m_] :=
          StringJoin[Table[" ", {n}], str, Table[" ", {m}]]

In[22]:= StringPad["ciao", 3, 8] // FullForm

Out[22]//FullForm= "   ciao        "
```

Note, `StringInsert` could also be used.

```
In[23]:= StringInsert["ciao", " ", {1, -1}] // FullForm

Out[23]//FullForm= " ciao "

In[24]:= StringPad2[str_String, n_, m_] :=
          StringInsert[str, " ", Join[Table[1, {n}], Table[-1, {m}]]]

In[25]:= StringPad2["ciao", 3, 8] // FullForm

Out[25]//FullForm= "   ciao        "
```

5. This is a simple modification of the code given in the text. But first we add the space character to the alphabet.

```
In[26]:= ToCharacterCode[" "]

Out[26]= {32}

In[27]:= alphabet = Join[{FromCharacterCode[32]}, CharacterRange["a", "z"]]

Out[27]= {  , a, b, c, d, e, f, g, h, i, j, k,
          l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}

In[28]:= coderules = Thread[alphabet → RotateRight[alphabet, 5]]

Out[28]= {  → v, a → w, b → x, c → y, d → z, e →  , f → a, g → b, h → c,
          i → d, j → e, k → f, l → g, m → h, n → i, o → j, p → k, q → l,
          r → m, s → n, t → o, u → p, v → q, w → r, x → s, y → t, z → u}

In[29]:= decoderules = Map[Reverse, coderules]

Out[29]= {v →  , w → a, x → b, y → c, z → d,   → e, a → f, b → g, c → h,
          d → i, e → j, f → k, g → l, h → m, i → n, j → o, k → p, l → q,
          m → r, n → s, o → t, p → u, q → v, r → w, s → x, t → y, u → z}

In[30]:= code[str_String] := Apply[StringJoin, Characters[str] /. coderules]

In[31]:= decode[str_String] :=
          Apply[StringJoin, Characters[str] /. decoderules]

In[32]:= code["squeamish ossifrage"]

Out[32]= nlp whdncvjnndamwb

In[33]:= decode[%]

Out[33]= squeamish ossifrage
```

6. First, here is the list of characters from the plaintext alphabet.

```
In[34]:= PlainAlphabet = CharacterRange["a", "z"]
Out[34]= {a, b, c, d, e, f, g, h, i, j, k,
          l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
```

Here is our key, *django*:

```
In[35]:= key = "django"
Out[35]= django
```

And here is the cipher text alphabet, prepending the key:

```
In[36]:= StringJoin[Characters@key,
          Complement[PlainAlphabet, Characters@key]]
Out[36]= djangobcefhiklmpqrstuvwxyz
```

Make a reusable function.

```
In[37]:= CipherAlphabet[key_String] := With[{k = Characters[key]},
           StringJoin[k, Complement[CharacterRange["a", "z"], k]]]
```

Generate the coding rules:

```
In[38]:= codeRules =
          Thread[PlainAlphabet → Characters@CipherAlphabet["django"]]
Out[38]= {a → d, b → j, c → a, d → n, e → g, f → o, g → b, h → c,
          i → e, j → f, k → h, l → i, m → k, n → l, o → m, p → p, q → q,
          r → r, s → s, t → t, u → u, v → v, w → w, x → x, y → y, z → z}
```

The encoding function follows that in the text of this section.

```
In[39]:= encode[str_String] := StringJoin[Characters[str] /. codeRules]

In[40]:= encode["the sheik of araby"]
Out[40]= tcg scgeh mo drdjy
```

Omit spaces and punctuation and output in blocks of length 5 (using `StringPartition` from Section 9.5).

```
In[41]:= StringPartition[str_String, seq__] :=
          Map[StringJoin, Partition[Characters[str], seq]]

In[42]:= StringSplit[encode["the sheik of araby"],
          RegularExpression["\\W+"]]
Out[42]= {tcg, scgeh, mo, drdjy}

In[43]:= StringJoin[Riffle[StringPartition[StringJoin[%], 5, 5, 1, ""], " "]]
Out[43]= tcgsc gehmo drdjy
```

Finally, this puts all these pieces together.

```
In[44]:= Clear[encode];
         encode[str_String, key_String, blocksize_: 5] :=
          Module[{CipherAlphabet, codeRules, s1, s2, s3},
           CipherAlphabet[k_] :=
            StringJoin[Characters[k],
             Complement[CharacterRange["a", "z"], Characters[k]]];

           codeRules = Thread[
             CharacterRange["a", "z"] → Characters@CipherAlphabet[key]];

           s1 = StringJoin[Characters[str] /. codeRules];
           s2 = StringSplit[s1, RegularExpression["\\W+"]];
           s3 =
            StringPartition[StringJoin[s2], blocksize, blocksize, 1, ""];
           StringJoin[Riffle[s3, " "]]]

In[46]:= encode["the sheik of araby", "django", 3]

Out[46]= tcg scg ehm odr djy
```

9.3 *String patterns*

1. At the end of Section 9.1 we created a predicate `OrderedWordQ` to find all words in a dictionary whose letters are in alphabetic order. This predicate used character codes and returned incorrect results for words that started with a capital letter. Correct this error by only selecting words from the dictionary that start with a lowercase letter. Consider using a conditional string pattern involving the built-in function `LowerCaseQ`.
2. Given a list of words, some of which start with uppercase characters, convert them all to words in which the first character is lowercase. You can use the words in the dictionary as a good sample set.
3. Create a function `Palindromes[n]` that finds all palindromic words of length *n* in the dictionary. For example, *kayak* is a five-letter palindrome.
4. Find the number of unique words in a body of text such as *Alice in Wonderland*.

```
In[1]:= text = ExampleData[{"Text", "AliceInWonderland"}];
```

After splitting the text into words, convert all uppercase characters to lowercase so that you count words such as *hare* and *Hare* as the same word.

Such computations are important in information retrieval systems, for example, in building term-document incidence matrices used to compare the occurrence of certain terms across a set of documents (Manning, Raghavan, and Schütze 2008).

9.3 *Solutions*

1. First, recall the predicate created in Section 9.1.

```
In[1]:= OrderedWordQ[word_String] := OrderedQ[ToCharacterCode[word]]
```

`DictionaryLookup` can be given a pattern as its argument and it will return only those words that match the pattern. Using `StringJoin`, test the first character with `LowerCaseQ`; the remainder of the word (zero or more characters) has no conditions.

```
In[2]:= words = DictionaryLookup[f_?LowerCaseQ ~~ r___];
        Short[words, 4]

Out[3]//Short= {a, aah, aardvark, aardvarks, abaci, aback, abacus, <<81 804>>,
                zwieback, zydeco, zygote, zygotes, zygotic, zymurgy}

In[4]:= Select[words, OrderedWordQ];
        RandomSample[%, 20]

Out[5]= {now, coop, ills, firs, chops, ass, chin, ells, go, clops,
         lox, sty, beep, alp, dims, befit, any, accost, aims, amp}
```

2. We will work with a small sample of words from the dictionary.

```
In[6]:= words = DictionaryLookup[];
        sample = RandomSample[words, 12]

Out[7]= {stiffest, alchemists, suppresses, deliberateness,
         frangibility, palaeontologists, progressions,
         tithers, retirement, burdening, Parkman, actresses}
```

`StringReplace` operates on any words that match the pattern and leave those that do not match unchanged.

```
In[8]:= StringReplace[sample, f_?UpperCaseQ ~~ r___ :> ToLowerCase[f] ~~ r]

Out[8]= {stiffest, alchemists, suppresses, deliberateness,
         frangibility, palaeontologists, progressions,
         tithers, retirement, burdening, parkman, actresses}
```

3. You can do a dictionary lookup with a pattern that tests whether the word is palindromic. Then find all palindromic words of a given length. Note the need for `BlankSequence` (__) as the simple pattern _ would only find words consisting of one character.

```
In[9]:= Palindromes[len_Integer] := DictionaryLookup[
          w__ /; (w == StringReverse[w] && StringLength[w] == len)]
```

We also add a rule to return all palindromes of any length.

```
In[10]:= Palindromes[] := DictionaryLookup[w__ /; (w == StringReverse[w])]

In[11]:= Palindromes[7]

Out[11]= {deified, repaper, reviver}
```

```
In[12]:=  Palindromes[]

Out[12]=  {a, aha, aka, bib, bob, boob, bub, CFC, civic, dad, deed, deified, did,
           dud, DVD, eke, ere, eve, ewe, eye, gag, gig, huh, I, kayak, kook,
           level, ma'am, madam, mam, MGM, minim, mom, mum, nan, non, noon,
           nun, oho, pap, peep, pep, pip, poop, pop, pup, radar, redder, refer,
           repaper, reviver, rotor, sagas, sees, seres, sexes, shahs, sis,
           solos, SOS, stats, stets, tat, tenet, TNT, toot, tot, tut, wow, WWW}
```

4. First import some sample text.

```
In[13]:=  text = ExampleData[{"Text", "AliceInWonderland"}];
```

To split into words, use a similar construction to that in this section.

```
In[14]:=  words = StringSplit[text, Characters[":;\"',.?/\-` *"] ..];
          Short[words, 4]

Out[15]//Short=  {I, DOWN, THE, RABBIT, HOLE, Alice, was, beginning,
                  ≪9955≫, might, what, a, wonderful, dream, it, had, been}
```

Get the total number of (nonunique) words.

```
In[16]:=  Length[words]

Out[16]=  9971
```

Convert uppercase to lowercase.

```
In[17]:=  lcwords = ToLowerCase[words];
          Short[lcwords, 4]

Out[18]//Short=  {i, down, the, rabbit, hole, alice, was, beginning,
                  ≪9955≫, might, what, a, wonderful, dream, it, had, been}
```

Finally, count the number of unique words.

```
In[19]:=  DeleteDuplicates[lcwords] // Length

Out[19]=  1643
```

In fact, splitting words using a list of characters as we have done here is not terribly robust. A better approach uses regular expressions (introduced in Section 9.4):

```
In[20]:=  words = StringSplit[text, RegularExpression["\\W+"]];
          Length[words]

Out[21]=  9970

In[22]:=  lcwords = StringReplace[words,
             RegularExpression["([A-Z])"] :→ ToLowerCase["$1"]];
          DeleteDuplicates[lcwords] // Length

Out[23]=  1528
```

9.4 *Regular expressions*

1. Rewrite the genomic example in Section 9.3 to use regular expressions instead of string patterns to find all occurrences of the sequence `AA`*anything*`T`. Here is the example using general string patterns.

```
In[1]:=  gene = GenomeData["IGHV357"];

In[2]:=  StringCases[gene, "AA" ~~ _ ~~ "T"]

Out[2]=  {AAGT, AAGT, AAAT, AAGT, AAAT, AAAT}
```

2. Rewrite the web page example in Section 9.3 to use regular expressions to find all phone numbers on the page; that is, expressions of the form *nnn–nnn–nnnn*. Modify accordingly for other web pages and phone numbers formatted for other regions.

3. Create a function `UcLc`[*word*] that takes its argument *word* and returns the word with the first letter uppercase and the rest of the letters lowercase.

4. Use a regular expression to find all words given by `DictionaryLookup` that consist only of the letters *a*, *e*, *i*, *o*, *u*, and *y* in any order with any number of repetitions of the letters.

5. The basic rules for pluralizing words in the English language are roughly, as follows: if a noun ends in *ch*, *s*, *sh*, *j*, *x*, or *z*, it is made plural by adding *es* to the end. If the noun ends in *y* and is preceded by a consonant, replace the *y* with *ies*. If the word ends in *ium*, replace with *ia* (Chicago Manual of Style 2010). Of course, there are many more rules and even more exceptions, but you can implement a basic set of rules to convert singular words to plural based on these rules and then try them out on the following list of words.

```
In[3]:=  words = {"building", "finch", "fix", "ratio",
            "envy", "boy", "baby", "faculty", "honorarium"};
```

6. A common task in transcribing audio is cleaning up text, removing certain phrases such as *um*, *er*, and so on, and other tags that are used to make a note of some sort. For example, the following transcription of a lecture from the University of Warwick, Centre for Applied Linguistics (BASE Corpus), contains quite a few fragments that should be removed, including newline characters, parenthetical remarks, and nonwords. Use `StringReplace` with the appropriate rules to “clean” this text and then apply your code to a larger corpus.

```
In[4]:=  text =
           "okay well er today we're er going to be carrying on with the
             er French \nRevolution you may have noticed i was sort
             of getting rather er enthusiastic \nand carried away at
             the end of the last one i was sort of almost er like
             i sort \nof started at the beginning about someone
             standing on a coffee table and s-, \nshouting to arms
             citizens as if i was going to sort of leap up on the
             desk and \nsay to arms let's storm the Rootes Social
             Building [laughter] or er let's go \nout arm in arm
             singing the Marseillaise or something er like that";
```

7. Find the distribution of sentence lengths for any given piece of text. `ExampleData["Text"]` contains several well-known books and documents that you can use. You will need to think about and identify sentence delimiters carefully. Take care to deal properly with words such as *Mr.*, *Dr.*, and so on that might incorrectly trigger a sentence-ending detector.

8. In web searches and certain problems in natural language processing (NLP), it is often useful to filter out certain words prior to performing the search or processing of the text to help with the performance of the algorithms. Words such as *the*, *and*, *is*, and so on are commonly referred to as *stop words* for this purpose. Lists of stop words are almost always created manually based on the constraints of a particular application. We will assume you can import a list of stop words as they are commonly available across the internet. For our purposes here, we will use one such list that comes with the materials for this book.

```
In[5]:= stopwords = Rest@Import["StopWords.dat", "List"];
        RandomSample[stopwords, 12]

Out[6]= {what, look, taken, specify, wants, thorough,
         they, hello, whose, them, mightn't, particular}
```

Using the above list of stop words, or any other that you are interested in, first filter some sample "search phrases" and then remove all stop words from a larger piece of text.

```
In[7]:= searchPhrases = {"Find my favorite phone",
           "How deep is the ocean?", "What is the meaning of life?"};
```

9. Modify the previous exercise so that the user can supply a list of punctuation in addition to the list of stop words to be used to filter the text.

9.4 Solutions

1. The pattern used earlier in the chapter was `"AA" ~~ _ ~~ "T"`. In a regular expression, we want the character *A* repeated exactly once. Use the expression `"A{2,2}"` for this. The regular expression `"."` stands for any character.

```
In[1]:= gene = GenomeData["IGHV357"];

In[2]:= StringCases[gene, RegularExpression["A{2,2}.T"]]

Out[2]= {AAGT, AAGT, AAAT, AAGT, AAAT, AAAT}
```

2. First, read in the web page.

```
In[3]:= webpage =
          Import["http://www.wolfram.com/company/contact.cgi", "HTML"];
```

In the original example in Section 9.3, we used the pattern `NumberString`, to represent arbitrary strings of numbers. The regular expression `"\\d+"` accomplishes a similar thing but it will also match strings of numbers that may not be in a phone number format (try it!). Instead, use `"\\d{3}"` to match a list of exactly three digits, and so on.

```
In[4]:= StringCases[webpage,
          RegularExpression["\\d{3}.\\d{3}.\\d{4}"]] // DeleteDuplicates

Out[4]= {217-398-0700, 217-398-0747, 617-764-0094}
```

3. First, here is the function using regular expressions. `(.)` will be matched by any single character; the parentheses are used to refer to this expression on the right-hand side of the rule as `"$1"`. Similarly, parentheses surround `[a - z] +` which is matched by any sequence of lowercase characters; this expression is referred to on the right as `"$2"`.

```
In[5]:= UcLc[word_String] := StringReplace[word,
          RegularExpression["(.)([a-z]+)"] :→ ToUpperCase["$1"] ~~ "$2"]

In[6]:= UcLc["hello"]

Out[6]= Hello
```

You can also do this with string patterns.

```
In[7]:= UcLc[word_String] := StringReplace[word,
          WordBoundary ~~ x_ ~~ y__ :→ ToUpperCase[x] ~~ ToLowerCase[y]]

In[8]:= UcLc["ciao"]

Out[8]= Ciao
```

4. The first solution uses regular expressions. The second uses string patterns and alternatives.

```
In[9]:= DictionaryLookup[
         RegularExpression["[aeiouy]+"], IgnoreCase → True]

Out[9]= {a, aye, eye, I, IOU, oi, ya, ye, yea, yo, you}

In[10]:= DictionaryLookup[
          ("a" | "e" | "i" | "o" | "u" | "y") .., IgnoreCase → True]

Out[10]= {a, aye, eye, I, IOU, oi, ya, ye, yea, yo, you}
```

5. Here is the short list of words with which we will work.

```
In[11]:= words = {"building", "finch", "fix", "ratio",
            "envy", "boy", "baby", "faculty", "honorarium"};
```

Using regular expressions, these rules encapsulate those given in the exercise.

```
In[12]:= rules = {
            (RegularExpression["(\\w+)x"] :→ "$1" ~~ "x" ~~ "es"),
            (RegularExpression["(\\w+)(ch)"] :→ "$1" ~~ "$2" ~~ "es"),
            (RegularExpression["(\\w+)([aeiou])(y)"] :→
              "$1" ~~ "$2" ~~ "$3" ~~ "s"),
            (RegularExpression["(\\w+)(y)"] :→ "$1" ~~ "ies"),
            (RegularExpression["(\\w+)(i)um"] :→ "$1" ~~ "$2" ~~ "a"),
            (RegularExpression["(\\w+)(.)"] :→ "$1" ~~ "$2" ~~ "s")
          };

In[13]:= StringReplace[words, rules]

Out[13]= {buildings, finches, fixes, ratios,
          envies, boys, babies, faculties, honoraria}
```

Of course, lots of exceptions exist:

```
In[14]:= StringReplace[{"man", "cattle"}, rules]

Out[14]= {mans, cattles}
```

6. We use a combination of string patterns and regular expressions to remove the various fragments. The regular expression `"\[.+\] "` matches strings that start with `[`, followed by an arbitrary number of characters, followed by `]`, followed by a space. Because brackets are used in regular expressions to denote sequences of characters, you need to escape them to refer to the explicit characters `[` or `]`.

```
In[15]:= text =
           "okay well er today we're er going to be carrying on with the
             er French \nRevolution you may have noticed i was sort
             of getting rather er enthusiastic \nand carried away at
             the end of the last one i was sort of almost er like
             i sort \nof started at the beginning about someone
             standing on a coffee table and s-, \nshouting to arms
             citizens as if i was going to sort of leap up on the
             desk and \nsay to arms let's storm the Rootes Social
             Building [laughter] or er let's go \nout arm in arm
             singing the Marseillaise or something er like that";

In[16]:= StringReplace[text, {"\n" → "", " er" → "",
           " s-" → "", RegularExpression["\[.+\] "] → ""}]

Out[16]= okay well today we're going to be carrying on with the French
           Revolution you may have noticed i was sort of getting
           rather enthusiastic and carried away at the end of the
           last one i was sort of almost like i sort of started
           at the beginning about someone standing on a coffee
           table and, shouting to arms citizens as if i was going
           to sort of leap up on the desk and say to arms let's
           storm the Rootes Social Building or let's go out arm
           in arm singing the Marseillaise or something like that
```

7. Start by importing a somewhat lengthy text, Charles Darwin's *On the Origin of Species*.

```
In[17]:= text = ExampleData[{"Text", "OriginOfSpecies"}];
```

There are numerous instances of "Mr." and "Dr.", words that end in a period that would trigger a sentence-ending detector such as `StringSplit`.

```
In[18]:= StringCount[text, "Mr." | "Dr."]

Out[18]= 119
```

To keep our sentence count accurate, we will replace such words (and a few others in this particular text) with words that will not cause errors in our sentence count. This step of cleaning text based on identified issues is a common one in textual analysis.

```
In[19]:=  cleanText = StringReplace[text,
             {"Mr." → "Mr", "Dr." → "Dr", "H.M.S." → "HMS"}];

In[20]:=  t = StringTake[cleanText, 200]

Out[20]=  INTRODUCTION. When on board HMS 'Beagle,' as
            naturalist, I was much struck with certain facts in
            the distribution of the inhabitants of South America,
            and in the geological relations of the present to
```

Now split on a small set of delimiters.

```
In[21]:=  s = StringSplit[cleanText, Characters[".!?"] ..];
          Short[s, 5]

Out[22]//Short=  {INTRODUCTION, ≪4225≫,
                   There is grandeur in this view of life, with
                     its several powers, hav …
                   s most beautiful and most wonderful have been,
                     and are being, evolved}
```

The same thing can be accomplished with a regular expression.

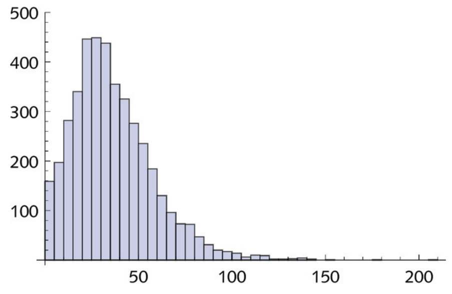
```
In[23]:=  s = StringSplit[cleanText, RegularExpression["[.!?]+"]];
```

Using a regular expression, this counts the number of words in each sentence.

```
In[24]:=  sentenceLens = StringCount[s, RegularExpression["\\w+"]];
```

Finally, here is a histogram displaying the distribution of sentence lengths.

```
In[25]:=  Histogram[sentenceLens]

Out[25]=
```



It looks like there are some very long sentences!

```
In[26]:= Select[s, StringCount[#, RegularExpression["\\w+"]] > 200 &]

Out[26]= { I have attempted to show that the geological record is
            extremely imperfect; that only a small portion of the
            globe has been geologically explored with care; that
            only certain classes of organic beings have been largely
            preserved in a fossil state; that the number both of
            specimens and of species, preserved in our museums, is
            absolutely as nothing compared with the incalculable
            number of generations which must have passed away even
            during a single formation; that, owing to subsidence
            being necessary for the accumulation of fossiliferous
            deposits thick enough to resist future degradation,
            enormous intervals of time have elapsed between the
            successive formations; that there has probably been more
            extinction during the periods of subsidence, and more
            variation during the periods of elevation, and during the
            latter the record will have been least perfectly kept;
            that each single formation has not been continuously
            deposited; that the duration of each formation is, perhaps,
            short compared with the average duration of specific
            forms; that migration has played an important part in
            the first appearance of new forms in any one area and
            formation; that widely ranging species are those which
            have varied most, and have oftenest given rise to new
            species; and that varieties have at first often been local}
```

8. First read in some sample phrases.

```
In[27]:= searchPhrases = {"Find my favorite phone",
            "How deep is the ocean?", "What is the meaning of life?"};
```

There are several ways to approach this problem. We will break it up into two steps: first eliminating punctuation, then a sample set of stop words.

```
In[28]:= tmp =
          StringSplit["How deep is the ocean?", Characters[":,;.!? "] ..]

Out[28]= {How, deep, is, the, ocean}

In[29]:= stopwords = {"how", "the", "is", "an"};

In[30]:= Apply[Alternatives, stopwords]

Out[30]= how | the | is | an
```

Note the need for `WordBoundary` in what follows; otherwise, *ocean* would be split leaving *oce* because *an* is a stop word.

```
In[31]:= StringSplit[tmp, WordBoundary ~~ Apply[Alternatives, stopwords] ~~
            WordBoundary, IgnoreCase → True] // Flatten

Out[31]= {deep, ocean}

In[32]:= FilterText[str_String, stopwords_List] := Module[{tmp},
           tmp = StringSplit[str, Characters[":,;.!? "] ..];
           Flatten@StringSplit[tmp,
             WordBoundary ~~ Apply[Alternatives, stopwords] ~~ WordBoundary,
             IgnoreCase → True]
          ]

In[33]:= SetDirectory@
          FileNameJoin[{$BaseDirectory, "Applications", "PwM", "Data"}]

Out[33]= /Library/Mathematica/Applications/PWM/Data

In[34]:= stopwords = Rest@Import["StopWords.dat", "List"];

In[35]:= FilterText["What is the meaning of life?", stopwords]

Out[35]= {meaning, life}
```

9. A slight modification is needed to accept a list of punctuation.

```
In[36]:= Characters@StringJoin[{".", "?"}] // FullForm

Out[36]//FullForm= List[".", "?"]
```

First remove the punctuation.

```
In[37]:= tmp = StringSplit["What is the meaning of life?",
           Characters@StringJoin[{".", "?"}] ..]

Out[37]= {What is the meaning of life}
```

Split into words.

```
In[38]:= First@StringCases[tmp, RegularExpression["\\w+"]]

Out[38]= {What, is, the, meaning, of, life}
```

Remove stop words.

```
In[39]:= StringSplit[%,
          WordBoundary ~~ Apply[Alternatives, stopwords] ~~ WordBoundary]

Out[39]= {{What}, {}, {}, {meaning}, {}, {life}}
```

Put these pieces together in a reusable function.

```
In[40]:= FilterText[str_String, stopwords_List, punctuation_List] :=
          Module[{tmp},
           tmp = StringSplit[str, Characters@StringJoin[punctuation] ..];
           Flatten@
            StringSplit[First@StringCases[tmp, RegularExpression["\\w+"]],
             WordBoundary ~~ Apply[Alternatives, stopwords] ~~ WordBoundary,
             IgnoreCase → True]
          ]

In[41]:= FilterText["What is the meaning of life?", stopwords, {".", "?"}]

Out[41]= {meaning, life}
```

Try it out on a list of phrases.

```
In[42]:= Map[FilterText[#, stopwords, {".", "?"}] &, searchPhrases]

Out[42]= {{Find, favorite, phone}, {deep, ocean}, {meaning, life}}
```

9.5 *Examples and applications*

1. Generalize the `RandomString` function to allow for a `Weights` option so that you can provide a weight for each character in the generated string. Include a rule to generate a message if the number of weights does not match the number of characters. For example:

```
In[1]:= RandomString[{"A", "T", "C", "G"}, 30, Weights → {.1, .2, .3, .4}]

Out[1]= GCGTCGTCGGGTCAGGTCCTCGTGTGGGCG

In[2]:= RandomString[{"A", "T", "C", "G"},
         {5, 10}, Weights → {.1, .4, .4, .1}]

Out[2]= {TTCACTTCCC, ACAACTGGCC, GATTCTTTTC, TGTCCTTTGA, TTCCTGCTGT}

In[3]:= RandomString[{"A", "T", "C", "G"}, {5, 10}, Weights → {.1, .4}]

        RandomString::badwt : The length of the list of weights must be the same as the length of the list of characters.
```

2. Write the function `Anagrams` developed in Section 9.2 without resorting to the use of `Permutations`. Consider using the `Sort` function to sort the characters. Note the difference in speed of the two approaches: one involving string functions and the other list functions that operate on lists of characters. Increase the efficiency of your search by only searching for words of the same length as your source word.

3. Rewrite the function `FindWordsContaining` using regular expressions instead of the patterns used in this section.

4. Using the text from several different sources, compute and then compare the number of punctuation characters per 1000 characters of text. `ExampleData["Text"]` gives a listing of many different texts that you can use.

5. The function `StringPartition` was developed specifically to deal with genomic data where one often needs uniformly-sized blocks to work with. Generalize `StringPartition` to fully accept the same argument structure as the built-in `Partition`.

Rewrite the text encoding example from Section 9.2 using `StringReplace` and regular expressions. First create an auxiliary function to encode a single character based on a key list of the form $\{\{pt_1, ct_1\}, \ldots\}$ where pt_i is a plaintext character and ct_i is its ciphertext encoding. For example, the pair $\{z, a\}$ would indicate the character *z* in the plaintext will be encoded as an *a* in the ciphertext. Then create an encoding function `encode[`*str*`,` *key*`]` using regular expressions to encode any string *str* using the *key* consisting of the plaintext/ciphertext character pairs.

9.5 *Solutions*

1. One rule is needed for one-dimensional output and another for multi-dimensional output.

```
In[1]:= ClearAll[RandomString]

In[2]:= Options[RandomString] = {Weights → {}};

In[3]:= RandomString::badwt =
          "The length of the list of weights must be the
            same as the length of the list of characters.";

In[4]:= RandomString[{c__String}, n_Integer: 1, OptionsPattern[]] :=
         Module[{wts = OptionValue[Weights]},
          Which[
           Length[wts] == 0, StringJoin[RandomChoice[{c}, n]],
           Length[wts] == Length[{c}],
           StringJoin[RandomChoice[wts → {c}, n]],
           True, Message[RandomString::badwt]
          ]]

In[5]:= RandomString[{c__String}, {n_Integer, len_Integer},
          OptionsPattern[]] := Module[{wts = OptionValue[Weights]},
          Which[
           Length[wts] == 0, Map[StringJoin, RandomChoice[{c}, {n, len}]],
           Length[wts] == Length[{c}],
           Map[StringJoin, RandomChoice[wts → {c}, {n, len}]],
           True, Message[RandomString::badwt]
          ]]

In[6]:= RandomString[{"A", "C", "T"}]

Out[6]= A

In[7]:= RandomString[{"A", "C", "T"}, 10]

Out[7]= TCCTCACCCC

In[8]:= RandomString[{"A", "C", "T"}, {4, 10}]

Out[8]= {ACATCTCATC, TCCCACTATC, AAACCCTCTC, CAATATAATC}

In[9]:= RandomString[{"A", "C", "T"}, {4, 10}, Weights → {.2, .7, .1}]

Out[9]= {CAAAACCCCC, CCCCACCCTC, CACCCCCACC, CAACCCCCCT}
```

```
In[10]:= RandomString[{"A", "C", "T"}, {4, 10}, Weights → {.2, .7}]

  RandomString::badwt : The length of the list of weights must be the same as the length of the list of characters.
```

2. Two words are anagrams if they contain the same letters but in a different order. This function is fairly slow as it sorts and compares every word in the dictionary with the sorted characters of the input word.

```
In[11]:= Anagrams2[word_String] := Module[{chars = Sort[Characters[word]]},
           DictionaryLookup[x__ /; Sort[Characters[x]] == chars]]

In[12]:= Anagrams2["parsley"] // Timing

Out[12]= {2.1535, {parleys, parsley, players, replays, sparely}}
```

You can speed things up a bit by only working with those words in the dictionary of the same length as the source word.

```
In[13]:= Anagrams3[word_String] :=
          Module[{len = StringLength[word], words},
           words = DictionaryLookup[w__ /; StringLength[w] == len];
           Select[words, Sort[Characters[#]] == Sort[Characters[word]] &]
          ]

In[14]:= Anagrams3["parsley"] // Timing

Out[14]= {0.890161, {parleys, parsley, players, replays, sparely}}
```

In fact, you can speed this up a bit further by using regular expressions even though the construction of the regular expression in this case is a bit clumsy looking. The lesson here is that conditional string patterns tend to be slower.

```
In[15]:= Anagrams4[word_String] :=
          Module[{len = StringLength[word], words},
           words = DictionaryLookup[
             RegularExpression["\\w{" <> ToString[len] <> "}"]];
           Select[words, Sort[Characters[#]] == Sort[Characters[word]] &]
          ]

In[16]:= Anagrams4["parsley"] // Timing

Out[16]= {0.098408, {parleys, parsley, players, replays, sparely}}
```

3. The pattern `"\\bcite.*\\b"` matches any string starting with a word boundary followed by the string *cite*, followed by characters repeated one or more times, followed by a word boundary.

```
In[17]:= DictionaryLookup[RegularExpression["\\bcite.*\\b"]]

Out[17]= {cite, cited, cites}
```

With suitable modifications to the above for the target string occurring in the middle, end, or anywhere, here is the rewritten function. Note the need for `StringJoin` here to properly pass the argument `str`, as a string, into the body of the regular expression.

```
In[18]:= Options[FindWordsContaining] = {WordPosition → "Start"};

In[19]:= FindWordsContaining[str_String, OptionsPattern[]] :=
          Module[{wp = OptionValue[WordPosition]},
           Which[
            wp == "Start", DictionaryLookup[
             RegularExpression[StringJoin["\\b", str, ".*\\b"]]],
            wp == "Middle", DictionaryLookup[
             RegularExpression[StringJoin["\\b.+", str, ".+\\b"]]],
            wp == "End", DictionaryLookup[
             RegularExpression[StringJoin["\\b.*", str, "\\b"]]],
            wp == "Anywhere", DictionaryLookup[
             RegularExpression[StringJoin["\\b.*", str, ".*\\b"]]]
           ]]

In[20]:= FindWordsContaining["cite"]

Out[20]= {cite, cited, cites}

In[21]:= FindWordsContaining["cite", WordPosition → "End"]

Out[21]= {anthracite, calcite, cite, excite,
          incite, Lucite, overexcite, plebiscite, recite}

In[22]:= FindWordsContaining["cite", WordPosition → "Middle"]

Out[22]= {elicited, excited, excitedly, excitement, excitements,
          exciter, exciters, excites, incited, incitement,
          incitements, inciter, inciters, incites, Lucites,
          overexcited, overexcites, plebiscites, recited, reciter,
          reciters, recites, solicited, unexcited, unsolicited}

In[23]:= FindWordsContaining["cite", WordPosition → "Anywhere"]

Out[23]= {anthracite, calcite, cite, cited, cites, elicited, excite,
          excited, excitedly, excitement, excitements, exciter, exciters,
          excites, incite, incited, incitement, incitements, inciter,
          inciters, incites, Lucite, Lucites, overexcite, overexcited,
          overexcites, plebiscite, plebiscites, recite, recited,
          reciter, reciters, recites, solicited, unexcited, unsolicited}
```

4. First read in a sample piece of text.

```
In[24]:= text = ExampleData[{"Text", "PrideAndPrejudice"}];
```

Check the length. Then partition into blocks consisting of 1000 characters each.

```
In[25]:= StringLength[text]

Out[25]= 682 262
```

```
In[26]:= blocks = StringPartition[text, 1000];
```

Using regular expressions, we extract all characters from the first block that are not amongst *A* through *z* or 0 through 9 or whitespace.

```
In[27]:= StringCases[blocks[[1]],
          RegularExpression["[^A-z|0-9|\\s]"]
         ]

Out[27]= {,, ,, ., ,, ,, ., ", ., ,, ", ,, ", ?, ", ., ., ", ,, ", ;, ",
          ., ,, ., ", ., ., ", ?, ", ., ", ,, ., ", ., ", ,, ,, ,, ., ;}

In[28]:= Tally[%] // InputForm

Out[28]//InputForm=
         {{",", 12}, {".", 13}, {"\"", 13},
          {"?", 2}, {";", 2}}
```

Now perform the same computation over all blocks and then compute the mean.

```
In[29]:= counts = Map[
            StringCount[#, RegularExpression["[^A-z|0-9|\\s]"]] &, blocks];

In[30]:= N[Mean[counts]]

Out[30]= 34.3021
```

Finally, perform the same computations on a different text.

```
In[31]:= text = ExampleData[{"Text", "OriginOfSpecies"}];
         blocks = StringPartition[text, 1000];
         counts = Map[
            StringCount[#, RegularExpression["[^A-z|0-9|\\s]"]] &, blocks];
         N[Mean[counts]]

Out[34]= 21.9877
```

5. Here is the function as developed in the text.

```
In[35]:= StringPartition[str_String, blocksize_] := Map[StringJoin,
           Partition[Characters[str], blocksize, blocksize, 1, {}]]
```

This passes the argument structure directly to `Partition`.

```
In[36]:= Clear[StringPartition]

In[37]:= StringPartition[str_String, seq__] :=
          Map[StringJoin, Partition[Characters[str], seq]]

In[38]:= str = RandomString[{"A", "C", "G", "T"}, 20]

Out[38]= AGCCGCTGATGCGAAAAATG
```

Try out some of the argument structures commonly used with `Partition`. For example, this partitions the string into blocks of length 3 with offset 1, with no padding

```
In[39]:= StringPartition[str, 3, 3, 1, {}]

Out[39]= {AGC, CGC, TGA, TGC, GAA, AAA, TG}
```

6. Start by creating a substitution cipher by simply shifting the alphabet three characters to the left.

```
In[40]:= keyRL3 = Transpose[{CharacterRange["a", "z"],
            RotateLeft[CharacterRange["a", "z"], 3]}]

Out[40]= {{a, d}, {b, e}, {c, f}, {d, g}, {e, h}, {f, i}, {g, j}, {h, k}, {i, l},
          {j, m}, {k, n}, {l, o}, {m, p}, {n, q}, {o, r}, {p, s}, {q, t}, {r, u},
          {s, v}, {t, w}, {u, x}, {v, y}, {w, z}, {x, a}, {y, b}, {z, c}}
```

Next, encode a single character using a designated key.

```
In[41]:= encodeChar[char_String, key_List] :=
          First@Cases[key, {char, next_} :> next]

In[42]:= encodeChar["z", keyRL3]

Out[42]= c
```

Finally, here is the encoding function. Recall the `"$1"` on the right-hand side of the rule refers to the first (and only in this case) regular expression on the left that is enclosed in parentheses.

```
In[43]:= encode[str_String, key_List] := StringReplace[str,
           RegularExpression["([a-z])"] :> encodeChar["$1", key]]
```

The decoding uses the same key, but reverses the pairs.

```
In[44]:= decode[str_String, key_List] := encode[str, Map[Reverse, key]]

In[45]:= encode["squeamish ossifrage", keyRL3]

Out[45]= vtxhdplvk rvvliudjh

In[46]:= decode[%, keyRL3]

Out[46]= squeamish ossifrage
```

You might want to modify the encoding rule to deal with uppercase letters. One solution is simply to convert them to lowercase.

```
In[47]:= encode[str_String, key_List] := StringReplace[ToLowerCase[str],
           RegularExpression["([a-z])"] :> encodeChar["$1", key]]

In[48]:= encode["Squeamish Ossifrage", keyRL3]

Out[48]= vtxhdplvk rvvliudjh
```

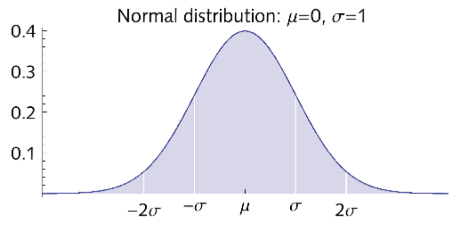
10 Graphics and visualization

10.1 *Structure of graphics*

1. Create a primitive color wheel by coloring successive sectors of a disk according to the `Hue` directive.
2. Create a graphic that contains a circle, a triangle, and a rectangle. Your graphic should include an identifying label for each object.
3. Create a three-dimensional graphic containing six `Cuboid` graphics primitives, randomly placed in the unit cube. Add an opacity directive to make them transparent.
4. Create a graphic consisting of a unit cube together with a rotation of 45° about the vertical axis through the center of that cube. Then create a dynamically rotating cube using `Manipulate`.
5. Create a graphic that consists of 500 points randomly distributed about the origin with standard deviation 1. Then, set the points to have random-size radii between 0.01 and 0.1 units and are colored randomly according to a `Hue` function.
6. Create a graphic that represents the solution to the following algebraic problem that appeared in the Calculus&*Mathematica* courseware (Porta, Davis, and Uhl 1994). Find the positive numbers *r* such that the following system has exactly one solution in *x* and *y*.

$$(x-1)^2+(y-1)^2=2$$
$$(x+3)^2+(y-4)^2=r^2$$

   Once you have found the right number *r*, then plot the resulting circles in true scale on the same axes, plotting the first circle with solid lines and the two solutions with dashed lines together in one graphic.
7. Create a graphic of the sine function over the interval $(0, 2\pi)$ that displays vertical lines at each point calculated by the `Plot` function to produce its plot.
8. Using options to the `Plot` function, create a plot showing the probability density function (pdf) of a normal distribution together with vertical lines at the first and second standard deviations. Your plot should look something like the following for a normal distribution with $\mu = 0$ and $\sigma = 1$:



9. Modify `ProteinDotPlot` from the introduction to this chapter to accept options from `ArrayPlot`.
10. Modify the `Hypocycloid` code to create *epicycloids*, which are like hypocycloids except the smaller circle rotates on the *outside* of the larger circle. Then create an animation showing the epicycloid being sketched out as the smaller circle rotates around the larger circle. If your animation includes a way to select different radii for the circles, you will need to deal with the plot range as the size of the smaller circle changes.

10.1 *Solutions*

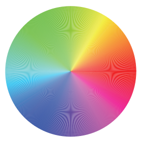
1. The color wheel can be generated by mapping the `Hue` directive over successive sectors of a disk. Note that the argument to `Hue` must be scaled so that it falls within the range 0 to 1.

```
In[1]:= colorWheel[n_] := Graphics[
          ({Hue[Rescale[#, {0, 2 π}]], Disk[{0, 0}, 1, {#, # + n}]} &) /@
           Range[0, 2 π, n]]
```

Here is a color wheel created from 256 separate sectors (hues).

```
In[2]:= colorWheel[π/256]
```

Out[2]=



2. Here is the circle graphic primitive together with a text label.

```
In[3]:= circ = Circle[{0, 0}, 1];

In[4]:= ctext = Text[Style["Circle", FontFamily → "Times",
            FontSlant → "Italic", FontSize → 12], {Cos[3 π/2], Sin[5 π/4]}];
```

This generates the graphics primitive for the triangle and its text label.

```
In[5]:= tri = Line[{{-1, 0}, {0, 1}, {1, 0}, {-1, 0}}];

In[6]:= ttext = Text[Style["Triangle", FontFamily → "Times",
            FontSlant → "Italic", FontSize → 12], {0, 0 + .15}];
```

Here is the rectangle and label.

```
In[7]:= rect = Line[{{-1, -1}, {-1, 1}, {2, 1}, {2, -1}, {-1, -1}}];

In[8]:= rtext = Text[Style["Rectangle", FontFamily → "Times",
            FontSlant → "Italic", FontSize → 12], {1.35, -1 + .15}];
```

Finally, this displays each of these graphics elements all together.

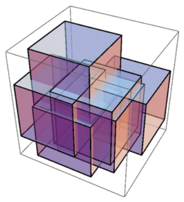
```
In[9]:=  Graphics[{circ, tri, rect, ctext, ttext, rtext}]
```

3. `Cuboid` takes a list of three numbers as the coordinates of its lower-left corner. This maps the object across two such lists.

```
In[10]:=  Map[Cuboid, RandomReal[1, {2, 3}]]

Out[10]=  {Cuboid[{0.603456, 0.627422, 0.210121}],
           Cuboid[{0.157798, 0.810453, 0.96354}]}
```

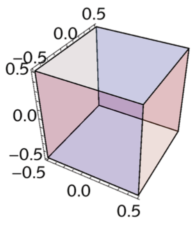
Below is a list of six cuboids and the resulting graphic. Notice the large amount of overlap of the cubes. You can reduce the large overlap by specifying minimum *and* maximum values of the cuboid.

```
In[11]:=  cubes = Map[Cuboid, RandomReal[1, {6, 3}]];

In[12]:=  Graphics3D[{Opacity[.5], cubes}]
```

Out[12]=


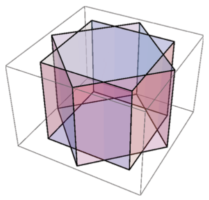
4. Start by creating a unit cube centered on the origin. An opacity directive adds transparency.

```
In[13]:=  Graphics3D[{Opacity[.25], Cuboid[{-0.5, -0.5, -0.5}]},
           Boxed → False, Axes → Automatic]
```

Out[13]=


Next rotate 45°. Note the third argument of `Rotate` used to specify the axis about which the rotation should occur.

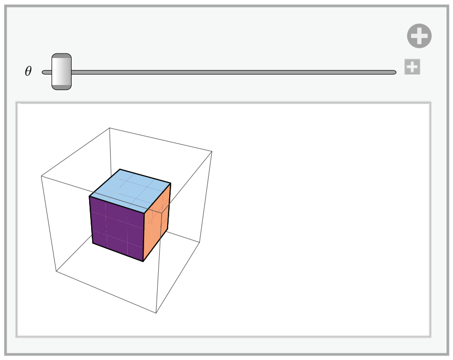
```
In[14]:=  Graphics3D[{Opacity[.25], Cuboid[{-.5, -.5, -.5}],
            Rotate[Cuboid[{-.5, -.5, -.5}], 45°, {0, 0, 1}]}]
```

Out[14]=


Here is the dynamic version. The angle θ is the parameter that is manipulated here.

```
In[15]:= Manipulate[
          Graphics3D[
           Rotate[Cuboid[{-.5, -.5, -.5}], θ, {0, 0, 1}], PlotRange → 1],
          {θ, 0, 2 π}]
```

Out[15]=



5. First we create the `Point` graphics primitives using a normal distribution with mean 0 and standard deviation 1.

```
In[16]:= randomcoords :=
          Point[RandomVariate[NormalDistribution[0, 1], {1, 2}]]
```

This creates the point sizes according to the specification given in the statement of the problem.

```
In[17]:= randomsize := PointSize[RandomReal[{.01, .1}]]
```

This will assign a random color to each primitive. The four-argument form of `Hue` specifies hue, saturation, brightness, opacity.

```
In[18]:= randomcolor := Hue[RandomReal[], 1, 1, .4]
```

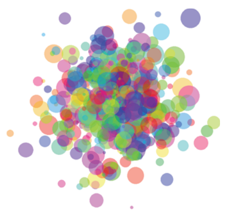
Here then are 500 points. (You may find it instructive to look at just one of these points.)

```
In[19]:= pts = Table[{randomcolor, randomsize, randomcoords}, {500}];
```

And here is the graphic.

```
In[20]:= Graphics[pts]
```

Out[20]=



6. The algebraic solution is given by the following steps. First solve the equations for *x* and *y*.

```
In[21]:= Clear[x, y, r]
```

```
In[22]:= soln = Solve[{(x - 1)^2 + (y - 1)^2 == 2, (x + 3)^2 + (y - 4)^2 == r^2}, {x, y}]
```

Out[22]=
$$\left\{\left\{x\to\frac{1}{50}\left(-58+4\,r^2-3\sqrt{-529+54\,r^2-r^4}\right),\ y\to\frac{1}{50}\left(131-3\,r^2-4\sqrt{-529+54\,r^2-r^4}\right)\right\},\left\{x\to\frac{1}{50}\left(-58+4\,r^2+3\sqrt{-529+54\,r^2-r^4}\right),\ y\to\frac{1}{50}\left(131-3\,r^2+4\sqrt{-529+54\,r^2-r^4}\right)\right\}\right\}$$

Then find those values of *r* for which the *x* and *y* coordinates are identical.

```
In[23]:= Solve[{
            (x /. soln[[1]]) == (x /. soln[[2]]),
            (y /. soln[[1]]) == (y /. soln[[2]])},
          r]
```

Out[23]= $\left\{\left\{r\to-5-\sqrt{2}\right\},\left\{r\to5-\sqrt{2}\right\},\left\{r\to-5+\sqrt{2}\right\},\left\{r\to5+\sqrt{2}\right\}\right\}$

Here then are those values of *r* that are positive.

```
In[24]:= Cases[%, {r → _?Positive}]
```

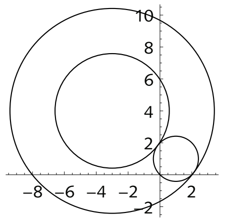
Out[24]= $\left\{\left\{r\to5-\sqrt{2}\right\},\left\{r\to5+\sqrt{2}\right\}\right\}$

To display the solution, we will plot the first circle with solid lines and the two solutions with dashed lines together in one graphic. Here is the first circle centered at (1, 1).

```
In[25]:= circ = Circle[{1, 1}, √2];
```

Here are the circles that represent the solution to the problem.

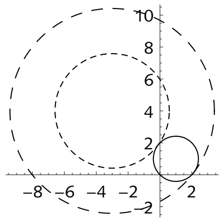
```
In[26]:= r1 = 5 - √2;
         r2 = 5 + √2;

In[28]:= Graphics[{circ, Circle[{-3, 4}, r1], Circle[{-3, 4}, r2]},
          Axes → Automatic]
```

Out[28]=


We wanted to display the solutions (two circles) using dashed lines. The graphics directive `Dashing[{x, y}]` directs all subsequent lines to be plotted as dashed, alternating the dash *x* units and the space *y* units. We use it as a graphics directive on the two circles `c1` and `c2`. The circles inherit only those directives in whose scope they appear.

```
In[29]:= dashc1 = {Dashing[{.025, .025}], Circle[{-3, 4}, r1]};
         dashc2 = {Dashing[{.05, .05}], Circle[{-3, 4}, r2]};

In[31]:= Graphics[{circ, dashc1, dashc2}, Axes → Automatic]
```

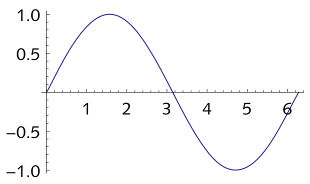
Out[31]=



7. Here is a plot of the sine function.

```
In[32]:= sinplot = Plot[Sin[x], {x, 0, 2 π}]
```

Out[32]=



Using pattern matching, here are the coordinates.

```
In[33]:= Short[coords = Cases[sinplot, Line[{x__}] :→ x, Infinity], 2]

Out[33]//Short= {{1.28228 × 10^-7, 1.28228 × 10^-7}, <<429>>, {<<1>>}}
```

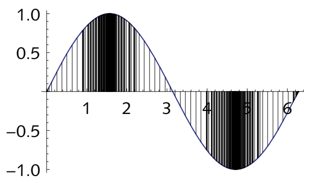
Create vertical lines from each coordinate.

```
In[34]:= Short[lines = Map[Line[{{#[[1]], 0}, #}] &, coords], 2]

Out[34]//Short= {Line[{{1.28228 × 10^-7, 0}, {<<23>>, <<23>>}}], <<430>>}
```

Here then is the final graphic.

```
In[35]:= Show[sinplot, Graphics[{Thickness[.001], lines}]]
```

Out[35]=

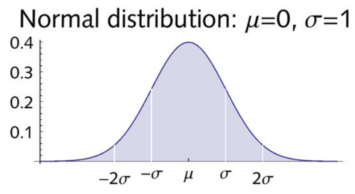


8. First set the distribution and compute the mean and standard deviation.

```
In[36]:= 𝒟 = NormalDistribution[0, 1];
         σ = StandardDeviation[𝒟];
         μ = Mean[𝒟];
```

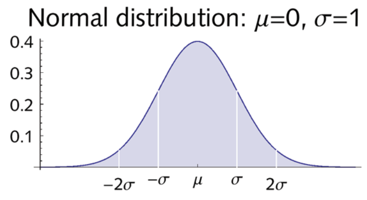
Next we manually construct four vertical lines at the standard deviations going from the horizontal axis to the pdf curve.

```
In[39]:= Plot[PDF[𝒟, x], {x, -4, 4}, Filling → Axis,
          Epilog → {White, Line[{{{μ + σ, 0}, {μ + σ, PDF[𝒟, μ + σ]}}, {{μ - σ, 0},
               {μ - σ, PDF[𝒟, μ - σ]}}, {{μ + 2 σ, 0}, {μ + 2 σ, PDF[𝒟, μ + 2 σ]}},
             {{μ - 2 σ, 0}, {μ - 2 σ, PDF[𝒟, μ - 2 σ]}}}]}, AxesOrigin → {-4, 0},
          Ticks → {{{-2 σ, "-2σ"}, {-σ, "-σ"}, {μ, "μ"}, {σ, "σ"}, {2 σ, "2σ"}},
            Automatic}, AspectRatio → 0.4, PlotLabel →
           StringForm["Normal distribution: μ=`1`, σ=`2` ", μ, σ]]
```

Out[39]=


And here is a little utility function to make the code a bit more readable and easier to use.

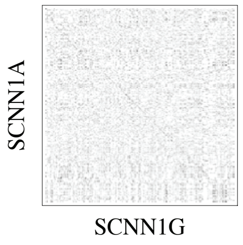
```
In[40]:= sdLine[𝒟_, μ_, σ_] := Line[{{{μ + σ, 0}, {σ + μ, PDF[𝒟, μ + σ]}},
            {{μ - σ, 0}, {-σ + μ, PDF[𝒟, μ - σ]}}}]

In[41]:= Plot[PDF[𝒟, x], {x, -4, 4}, Filling → Axis,
          Epilog → {White, Thickness[.0035], sdLine[𝒟, μ, σ],
            sdLine[𝒟, μ, 2 σ]}, AxesOrigin → {-4, 0},
          Ticks → {{{-2 σ, "-2σ"}, {-σ, "-σ"}, {μ, "μ"}, {σ, "σ"}, {2 σ, "2σ"}},
            Automatic}, AspectRatio → 0.4, PlotLabel →
           StringForm["Normal distribution: μ=`1`, σ=`2` ", μ, σ]]
```

Out[41]=


9. Following the discussion of options in Section 5.7, we use `OptionsPattern` to inherit options from `ArrayPlot`.

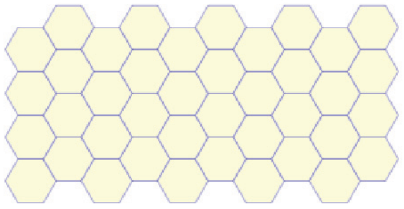
```
In[42]:= ProteinDotPlot[p1_, p2_, opts : OptionsPattern[ArrayPlot]] :=
          ArrayPlot[
           Outer[Boole[#1 == #2] &, Characters[p1], Characters[p2]],
           opts, Frame → True]

In[43]:= seq1 = ProteinData["SCNN1A"];
         seq2 = ProteinData["SCNN1G"];
```

```
In[45]:= ProteinDotPlot[seq1, seq2,
          FrameLabel → {"SCNN1A", "SCNN1G"},
          LabelStyle → {FontFamily → "Times", 11}]
```

Out[45]=


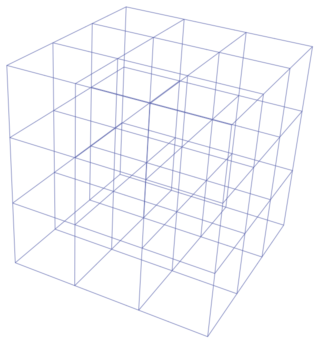
10.2 *Efficient structures*

1. Create a hexagonal grid of polygons like the one below.



First create the grid by performing appropriate translations using either `Translate` or the geometric transformation `TranslationTransform`. Compare this approach with a multi-polygon approach.

2. Create a graphic consisting of a three-dimensional lattice, that is, lines on the integer coordinates in 3-space. Compare approaches that use multi-lines as opposed to those that do not.

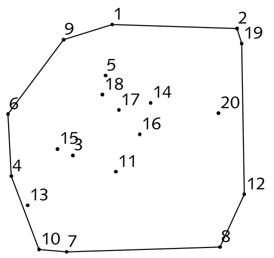


3. A common problem in computational geometry is finding the boundary of a given set of points. One way to think about this is to imagine the points as nails in a board and then to stretch a rubber band around all the nails. The stretched rubber band lies on a convex polygon commonly called the *convex hull* of the point set. The problem of determining the convex hull of a set of points has application in computer vision, pattern recognition, image processing, and many other areas. Using

the `ConvexHull` function defined in the Computational Geometry package, create a function `ConvexHullPlot` for visualizing the convex hull together with its point set. The resulting graphic should include the points labeled with text as well as the convex polygon drawn as a line around the point set.

```
In[1]:=  pts = RandomReal[1, {20, 2}];

In[2]:=  Needs["ComputationalGeometry`"]

In[3]:=  ConvexHull[pts]

Out[3]=  {12, 19, 2, 1, 9, 6, 4, 10, 7, 8}

In[4]:=  ConvexHullPlot[pts]
```

Out[4]=



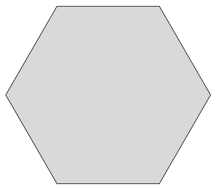
4. Extend Exercise 9 from Section 8.4 to random walks on the base n digits of π. For example, in base 3, a 1 corresponds to an angle of 120° from the current position, 2 corresponds to 240°, and 0 to 360°. In base 4 the step angles will be multiples of 90° and in general, for base n, the step angles will be multiples of 360 °/n. Use `GraphicsComplex` to visualize the walks. Include a color function that depends upon the length of the walk. For more on random walks on digits of π in various bases, see Bailey et al. (2012).

10.2 *Solutions*

1. Here is the implementation using `TranslationTransform`.

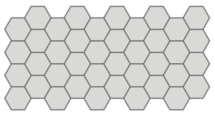
```
In[1]:=  vertices[n_] := Table[{Cos[2 π α / n], Sin[2 π α / n]}, {α, 0, n}]

In[2]:=  hexagon = Polygon[vertices[6]];
         Graphics[{EdgeForm[Gray], LightGray, hexagon}]
```

Out[2]=



```
In[3]:= Graphics[{
          EdgeForm[Gray], LightGray,
          Table[GeometricTransformation[hexagon,
            TranslationTransform[{3 i + 3/4 ((-1)^j + 1), (√3 j)/2}]
          ], {i, 5}, {j, 8}]
        }]
```

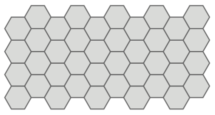
Out[3]=



Or use `Translate` directly.

```
In[4]:= gr1 = Graphics[{
            EdgeForm[Gray], LightGray,
            Table[
             Translate[hexagon, {3 i + 3/4 ((-1)^j + 1), (√3 j)/2}], {i, 5}, {j, 8}]
          }]
```

Out[4]=



This implementation contains one `Polygon` per hexagon.

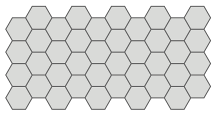
```
In[5]:= Count[gr1, _Polygon, Infinity]

Out[5]= 40
```

Now use multi-polygons. The following version of `hexagon` is defined so that it can take a pair of translation coordinates. Note also the need to flatten the table of vertices so that `Polygon` can be applied to the correct list structure.

```
In[6]:= Clear[hexagon];
        hexagon[{x_, y_}] :=
         Table[{Cos[(2 π i)/6] + x, Sin[(2 π i)/6] + y}, {i, 1, 6}]
```

```
In[8]:= gr2 = Graphics[{EdgeForm[Gray], LightGray, Polygon[Flatten[
            Table[hexagon[{3 i + 3/4 ((-1)^j + 1), (√3 j)/2}], {i, 5}, {j, 8}], 1]]
         }]
```

Out[8]=



```
In[9]:= Count[gr2, _Polygon, Infinity]

Out[9]= 1
```

2. One approach to creating the lattice is to manually specify the coordinates for the lines and then map the `Line` primitive across these coordinates. We will work with a small lattice.

```
In[10]:= xmin = 0; xmax = 3;
         ymin = 0; ymax = 3;
         zmin = 0; zmax = 3;
         Table[{{x, ymin, zmin}, {x, ymax, zmin}}, {x, xmin, xmax}]

Out[13]= {{{0, 0, 0}, {0, 3, 0}}, {{1, 0, 0}, {1, 3, 0}},
          {{2, 0, 0}, {2, 3, 0}}, {{3, 0, 0}, {3, 3, 0}}}
```

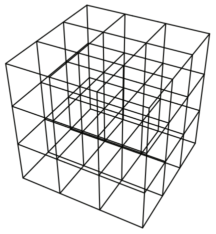
Here are the three grids.

```
In[14]:= gridX = Table[{{xmin, y, z}, {xmax, y, z}},
             {y, ymin, ymax}, {z, zmin, zmax}];
         gridY = Table[{{x, ymin, z}, {x, ymax, z}},
             {x, xmin, xmax}, {z, zmin, zmax}];
         gridZ = Table[{{x, y, zmin}, {x, y, zmax}},
             {x, xmin, xmax}, {y, ymin, ymax}];
```

Finally, map `Line` across these grids and display as a `Graphics3D` object.

```
In[17]:= gr1 = Graphics3D[{
             Map[Line, gridX, {2}],
             Map[Line, gridY, {2}],
             Map[Line, gridZ, {2}]
           }, Boxed → False]
```

Out[17]=

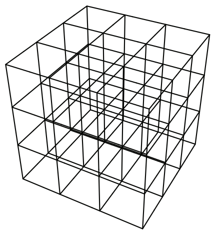


```
In[18]:= Count[gr1, _Line, Infinity]

Out[18]= 48
```

Using multi-lines reduces the number of `Line` objects substantially.

```
In[19]:= gr2 = Graphics3D[{
             Map[Line, gridX],
             Map[Line, gridY],
             Map[Line, gridZ]
           }, Boxed → False]
```

Out[19]=



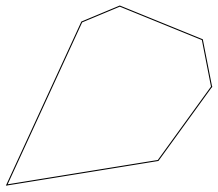
```
In[20]:= Count[gr2, _Line, Infinity]

Out[20]= 12
```

3. The Computational Geometry package contains a function for computing the convex hull. `ConvexHull[`*pts*`]` returns a list of the indices of the points on the convex hull.

```
In[21]:= Needs["ComputationalGeometry`"]

In[22]:= pts = RandomReal[1, {12, 2}];
         ch = ConvexHull[pts]

Out[23]= {10, 1, 2, 9, 11, 7, 8}
```

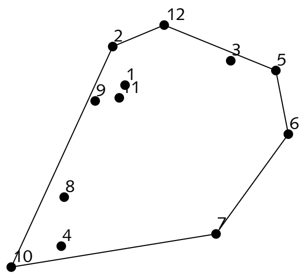
Use those indices as the positions in `pts` through which we wish to pass a line. Note the need to close up the polygon, connecting the last point with the first.

```
In[24]:= Graphics[GraphicsComplex[pts, Line[ch /. {a_, b__} :→ {a, b, a}]]]
```

Out[24]=


Now add the text.

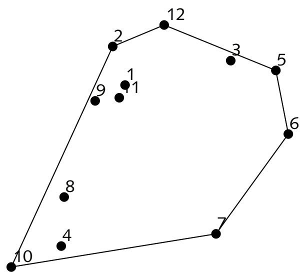
```
In[25]:= ran = Range[Length[pts]];
         Graphics[GraphicsComplex[pts, {Line[ch /. {a_, b__} :→ {a, b, a}],
             PointSize[.015], Point[ran], Map[
              Text[StringForm["`1`", #], pts[[#]], {-1.25, -1.25}] &, ran]}]]
```

Out[26]=


Putting everything together, note that because `Module` is a scoping construct, you need to give full context names for any function that is defined in a package loaded inside `Module`.

```
In[27]:= Clear[ConvexHullPlot]

In[28]:= ConvexHullPlot[pts_, opts : OptionsPattern[Graphics]] :=
          Module[{ch, ran = Range[Length[pts]]},
           Needs["ComputationalGeometry`"];
           ch = ComputationalGeometry`ConvexHull[pts];
           Graphics[{GraphicsComplex[
               pts,
               {Line[ch /. {a_, b__} :→ {a, b, a}],
                PointSize[.015], Point[ran],
                Map[Text[StringForm["`1`", #],
                    pts[[#]], {-1.25, -1.25}] &, ran]
               }
             ]}, opts]]
```

```
In[29]:= ConvexHullPlot[pts]
```

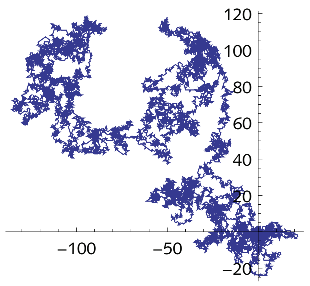
Out[29]=


4. Here is the random walk on the digits of π in bases given by the second argument.

```
In[30]:= RandomWalkPi[d_, base_ /; base > 2] :=
          Module[{digits, angles, rules},
           digits = First[RealDigits[N[π, d], base]];
           angles = Rest@Range[0., 2 π, 2 π / (base)];
           rules = MapThread[#1 → #2 &, {Range[0, base - 1], angles}];
           Accumulate[Map[{Cos[#], Sin[#]} &, digits /. rules]]
          ]
```

Using `ListPlot`, here is a quick visualization on base 5 digits:

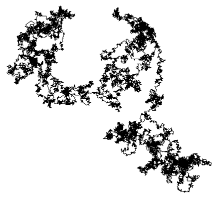
```
In[31]:= ListLinePlot[RandomWalkPi[10 000, 5], AspectRatio → Automatic]
```

Out[31]=


Here is the `GraphicsComplex`.

```
In[32]:= walk = RandomWalkPi[10 000, 5];
         len = Length[walk];
```

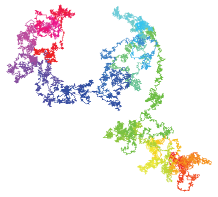
```
In[34]:= Graphics[
          GraphicsComplex[walk, {AbsoluteThickness[.2], Line[Range[len]]}],
          AspectRatio → Automatic]
```

Out[34]=


And here it is with color mapped to the distance from the origin.

```
In[35]:= Graphics[GraphicsComplex[walk,
            Map[{Hue[#[[1]]/len], AbsoluteThickness[.25], Line[#]} &,
             Partition[Range[2, len], 2, 1]]], AspectRatio → Automatic]
```

Out[35]=

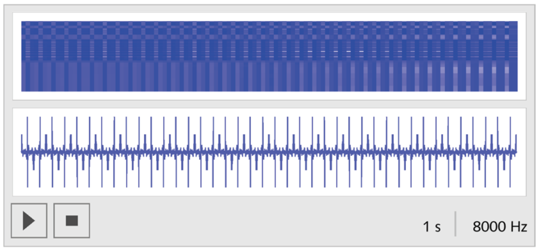


10.3 *Sound*

1. Evaluate `Play[Sin[1000 / x], {x, -2, 2}]`. Explain the dynamics of the sound generated from this function.
2. Experiment with the `Play` function by creating arithmetic combinations of sine functions. For example, you might try the following.

```
In[1]:= Play[Sin[440 × 2 π t]/Sin[660 × 2 π t], {t, 0, 1}]
```

Out[1]=



3. Create a tone that doubles in frequency each second.
4. Create a "composition" using the digits of π as representing notes on the C scale where a digit n is interpreted as a note n semitones from middle C. For example, the first few digits, 1, 4, 1, 5 would give the notes one, four, one, and five semitones from middle C.
5. A *square wave* consists of the addition of sine waves, each an odd multiple of a fundamental frequency, that is, it consists of the sum of sine waves having frequencies $f_0, 3 f_0, 5 f_0, 7 f_0$, etc. Create a square wave with a fundamental frequency f_0 of 440 hertz. The more overtones you include, the "squarer" the wave.
6. Create a square wave consisting of the sum of sine waves with frequencies $f_0, 3 f_0, 5 f_0, 7 f_0$, etc., and amplitudes 1, 1/3, 1/5, 1/7, respectively. This is actually a truer square wave than that produced in the previous exercise.

7. Create a square wave consisting of overtones that are randomly out of phase. How does this wave differ from the previous two?
8. A *sawtooth wave* consists of the sum of both odd- and even-numbered overtones: f_0, $2 f_0$, $3 f_0$, $4 f_0$, etc. with amplitudes in the ratios 1, 1/2, 1/3, 1/4, etc. Create a sawtooth wave and compare its tonal qualities with the square wave.
9. A wide variety of sounds can be generated using *frequency modulation (FM) synthesis*. The basic idea of FM synthesis is to use functions of the form

$$a \ \sin(2\,\pi\,F_c,\ t + \text{mod}\ \ \sin(2\,\pi\,F_m\,t)).$$

where a is the peak amplitude, F_c is the carrier frequency in hertz, mod is the modulation index, and F_m is the modulating frequency in hertz.

Determine what effect varying the parameters has on the resulting tones by creating a series of FM synthesized tones. First, create a function `FM[Amp, Fc, mod, Fm, time]` that implements the above formula and generates a tone using the `Play` function. Then you should try several examples to see what effect varying the parameters has on the resulting tones. For example, you can generate a tone with strong vibrato at a carrier frequency at middle A for one second by evaluating `FM[1, 440, 45, 5, 1]`.

10.3 Solutions

1. When x is close to -2, the frequency is quite low. As x increases, the fraction $1000/x$ increases, making the frequency of the sine function bigger. This in turn makes the tone much higher in pitch. As x approaches 0, the function is oscillating more and more, and at 0, the function can be thought of as oscillating infinitely often. In fact, it is oscillating so much that the sampling routine is not able to compute amplitudes effectively and, hence, we hear noise near $x = 0$.

```
In[1]:=  Play[Sin[1000/x], {x, -2, 2}]
```

3. To generate a tone whose rate increases one octave per second, you need the sine of a function whose derivative doubles each second (frequency is a rate). That function is 2^t. You need to carefully choose values for t that generates tones in a reasonable range.

```
In[2]:=  Play[Sin[2^t], {t, 10, 14}] // EmitSound
```

4. First generate 100 digits for a 100-note “composition”.

```
In[3]:=  digs = First[RealDigits[N[π, 100]]];
```

Fix note duration at 0.5 seconds.

```
In[4]:=  Sound[SoundNote[#, 0.5] & /@ digs] // EmitSound
```

Change the duration to be dependent upon the digit. Also change the MIDI instrument.

```
In[5]:=  Sound[SoundNote[#, 1 / (# + 1), "Vibraphone"] & /@ digs] // EmitSound
```

Go a bit further, expanding the range of notes that will be played.

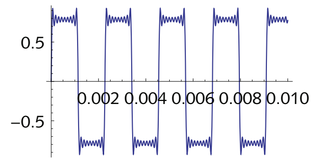
```
In[6]:=  Sound[SoundNote[1 + 2 #, 1 / (# + 1), "Vibraphone"] & /@ digs] //
          EmitSound
```

5. Here is a function that creates a square wave with decreasing amplitudes for higher overtones.

```
In[7]:=  squareWave[freq_, n_] := Sum[Sin[freq i 2 π t]/i, {i, 1, n, 2}]
```

```
In[8]:=  Plot[squareWave[440, 17], {t, 0, .01}]
```

Out[8]=



Here then, is an example of playing a square wave.

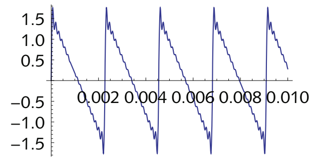
```
In[9]:=  Play[squareWave[440, 17], {t, 0, .5}] // EmitSound
```

8. This function creates a sawtooth wave. The user specifies the fundamental frequency and the number of terms in the approximation.

```
In[10]:=  sawtoothWave[freq_, n_] := Sum[Sin[freq i 2 π t]/i, {i, 1, n}]
```

```
In[11]:=  Plot[sawtoothWave[440, 17], {t, 0, .01}]
```

Out[11]=



This plays the wave for a half-second duration.

```
In[12]:=  Play[sawtoothWave[440, 17], {t, 0, .5}] // EmitSound
```

10.4 *Examples and applications*

1. Create a function `ComplexListPlot` that plots a list of complex numbers in the complex plane using `ListPlot`. Set initial options so that the `PlotStyle` is red, the `PointSize` is a little larger than the default, and the horizontal and vertical axes are labeled “Re” and “Im,” respectively. Set it up so that options to `ComplexListPlot` are inherited from `ListPlot`.

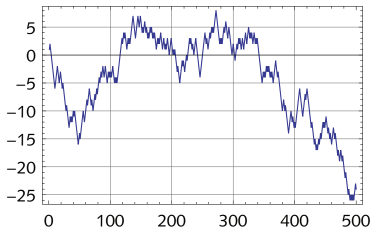
2. Create a function `ComplexRootPlot` that plots the complex solutions to a polynomial in the plane. Use your implementation of `ComplexListPlot` that you developed in the previous exercise.

3. Modify `PathPlot` so that it inherits options from `Graphics` as well as having its own option, `PathClosed`, that can take on values of `True` or `False` and closes the path accordingly by appending the first point to the end of the list of coordinate points.

4. Extend the code for `ListLinePlot3D` so that the rule for multiple datasets incorporates the options that were used for the single dataset rule in the text.
5. Although the program `SimpleClosedPath` works well, there are conditions under which it will occasionally fail. Experiment by repeatedly computing `SimpleClosedPath` for a set of ten points until you see the failure. Determine the conditions that must be imposed on the selection of the base point for the program to work consistently.
6. Modify `SimpleClosedPath` so that the point with the smallest *x*-coordinate of the list of data is chosen as the `base` point; repeat but with the largest *y*-coordinate.
7. Another way of finding a simple closed path is to start with any closed path and progressively make it simpler by finding intersections and changing the path to avoid them. Prove that this process ends, and that it ends with a closed path. Write a program to implement this procedure and then compare the paths given by your function with those of `SimpleClosedPath` given in the text.
8. Following on the framework of the `RootPlot` example in this section, create a function `ShowWalk`[*walk*] that takes the coordinates of a random walk and plots them in one, two, or three dimensions, depending upon the structure of the argument *walk*. For example:

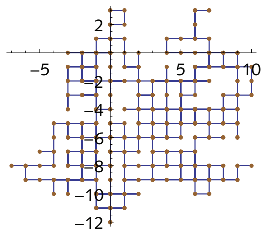
```
In[1]:=  << PwM`RandomWalks`

In[2]:=  ShowWalk[RandomWalk[500, Dimension → 1],
          Frame → True, GridLines → Automatic]
```

Out[2]=



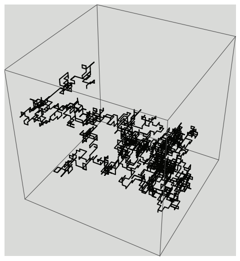
```
In[3]:=  ShowWalk[RandomWalk[500, Dimension → 2],
          Mesh → All, MeshStyle → Directive[Brown, PointSize[Small]]]
```

Out[3]=



```
In[4]:= ShowWalk[RandomWalk[2500, Dimension → 3],
         Background → LightGray, BoxRatios → {1, 1, 1}]
```

Out[4]=



9. Use `Mesh` in a manner similar to its use in the `RootPlot` function to highlight the *intersection* of two surfaces, say $\sin(2x - \cos(y))$ and $\sin(x - \cos(2y))$. You may need to increase the value of `MaxRecursion` to get the sampling just right.
10. Rewrite `TrendPlot` to compute a more robust plot range, one based on the minimum and maximum values of the data together with the minimum and maximum user-specified rates.
11. Modify the graphics code at the end of the `PointInPolygonQ` example so that `GatherBy` always orders the two lists so that the list of points that pass occurs before the list of points that fail the test.
12. Write a function `pentatonic` that generates $1/f^2$ music choosing notes from a five-tone scale. A pentatonic scale can be played on a piano by beginning with C♯, and then playing only the black keys: C♯, E♭, F♯, A♭, C♯. The pentatonic scale is common to Chinese, Celtic, and Native American music.
13. Modify the routine for generating $1/f^0$ music so that frequencies are chosen according to a specified probability distribution. For example, you might use the following distribution that indicates a note and its probability of being chosen: C – 5%, C♯ – 5%, D – 5%, E♭ – 10%, E – 10%, F – 10%, F♯ – 10%, G – 10%, A♭ – 10%, A – 10%, B♭ – 5%, B – 5%, C – 5%.
14. Modify the routine for generating $1/f^0$ music so that the *durations* of the notes obey $1/f^0$ scaling.
15. If you read musical notation, take a musical composition such as one of Bach's *Brandenburg Concertos* and write down a list of the frequency intervals x between successive notes. Then find a function that interpolates the power spectrum of these frequency intervals and determine if this function is of the form $f(x) = c/x$ for some constant c. (*Hint*: To get the power spectrum, you will need to square the magnitude of the Fourier transform: take `Abs[Fourier[…]]`^{2} of your data.) Compute the power spectra of different types of music using this procedure.

10.4 *Solutions*

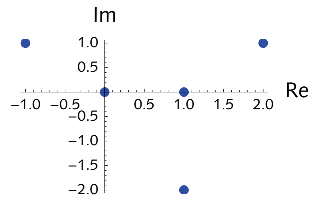
1. The function `ComplexListPlot` plots a list of numbers in the complex plane – the real part is identified with the horizontal axis and the imaginary part is identified with the vertical axis. Start by setting the options for `ComplexListPlot` to inherit those for `ListPlot`.

```
In[1]:= Options[ComplexListPlot] = Options[ListPlot];
```

```
In[2]:= ComplexListPlot[points_, opts : OptionsPattern[]] :=
         ListPlot[Map[{Re[#], Im[#]} &, points],
          opts, PlotStyle → {Red, PointSize[.025]},
          AxesLabel → {Style["Re", 10], Style["Im", 10]},
          LabelStyle → Directive["Menu", 7]]
```

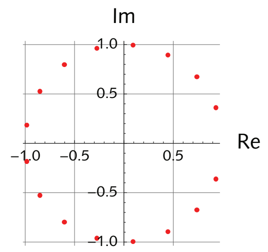
This plots four complex numbers in the plane and uses some options, inherited from `ListPlot`.

```
In[3]:= ComplexListPlot[{-1 + I, 2 + I, 1 - 2 I, 0, 1},
         PlotStyle → {Blue, PointSize[Medium]}]
```

Out[3]=


2. The function `ComplexRootPlot` takes a polynomial, solves for its roots, and then uses `ComplexListPlot` from the previous exercise to plot these roots in the complex plane.

```
In[4]:= ComplexRootPlot[poly_, z_, opts : OptionsPattern[]] :=
         ComplexListPlot[z /. NSolve[poly == 0, z],
          opts, AspectRatio → Automatic]

In[5]:= ComplexRootPlot[Cyclotomic[17, z], z, GridLines → Automatic]
```

Out[5]=


3. First, set up the options structure.

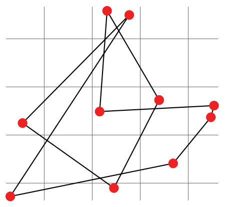
```
In[6]:= Options[PathPlot] = Join[{ClosedPath → True}, Options[Graphics]];
```

Make two changes to the original `PathPlot`: add an `If` statement that checks the value of `ClosedPath` and if `True`, appends the first point to the end of the list; if `False`, it leaves the coordinate list as is. The second change is to filter those options that are specific to `Graphics` and insert them in the appropriate place.

```
In[7]:= PathPlot[lis_List, opts : OptionsPattern[]] :=
         Module[{coords = lis}, If[OptionValue[ClosedPath],
            coords = coords /. {a_, b__} :→ {a, b, a}];
          Graphics[{Line[coords], PointSize[Medium], Red, Point[coords]},
           FilterRules[{opts}, Options[Graphics]]]]

In[8]:= SeedRandom[424];
        coords = RandomReal[1, {10, 2}];
```

```
In[10]:= PathPlot[coords, ClosedPath → True, GridLines → Automatic]
```

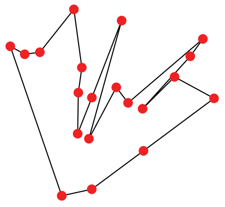
Out[10]=



5. Choosing a base point randomly and then sorting according to the arc tangent could cause a number of things to go wrong with the algorithm. The default branch cut for `ArcTan` gives values between $-\pi/2$ and $\pi/2$. (You are encouraged to think about why this could occasionally cause the algorithm in the text to fail.) By choosing the base point so that it lies at some extreme of the diameter of the set of points, the polar angle algorithm given in the text will work consistently. If you choose the base point so that it is lowest and left-most, then all the angles will be in the range $(0, \pi]$.

```
In[11]:= SimpleClosedPath1[lis_List] := Module[{base, angle, sorted},
           base = First[SortBy[lis, Last]];
           angle[a_, b_] := ArcTan @@ (b - a);
           sorted = Sort[Complement[lis, {base}],
             angle[base, #1] ≤ angle[base, #2] &];
           Join[{base}, sorted, {base}]]

In[12]:= pts = RandomReal[1, {20, 2}];

In[13]:= PathPlot[coords_List] :=
          Show[Graphics[{Line[coords], PointSize[Medium],
             RGBColor[1, 0, 0], Point /@ coords}]]

In[14]:= PathPlot[SimpleClosedPath1[pts]]
```

Out[14]=



6. A simple change to the program `SimpleClosedPath` chooses the base point with the largest y-coordinate.

```
In[15]:= SimpleClosedPath3[lis_] := Module[{base, angle, sorted},
           base = Last[SortBy[lis, Last]];
           angle[a_, b_] := ArcTan @@ (b - a); sorted = Sort[
             Complement[lis, {base}], angle[base, #1] ≤ angle[base, #2] &];
           Join[{base}, sorted, {base}]]

In[16]:= pts = RandomReal[1, {20, 2}];

In[17]:= PathPlot[SimpleClosedPath3[pts]]
```

8. Create three rules, one for each of the three dimensions of random walk that will be passed to `ShowWalk`. Some pattern matching will help to identify the rule to use for the one-, two-, and three-dimensional cases.

```
In[15]:= MatchQ[{1, 2, 3}, _?VectorQ]

Out[15]= True

In[16]:= MatchQ[{{1, 1}, {1, 2}, {0, 2}}, {{_, _} ..}]

Out[16]= True

In[17]:= MatchQ[{{1, 1, 0}, {1, 2, 0}, {0, 2, 0}}, {{_, _, _} ..}]

Out[17]= True
```

The first rule uses a pattern that will be matched by a one-dimensional vector.

```
In[18]:= ShowWalk[coords_?VectorQ, opts : OptionsPattern[]] :=
             ListLinePlot[coords,
           FilterRules[{opts}, Options[ListLinePlot]]]
```

The second rule uses a pattern that will be matched by a list of one or more pairs of numbers.

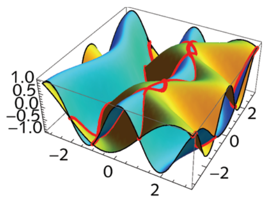
```
In[19]:= ShowWalk[coords : {{_?NumberQ, _?NumberQ} ..},
           opts : OptionsPattern[]] :=
             ListLinePlot[coords, Append[FilterRules[{opts},
            Options[ListLinePlot]], AspectRatio → Automatic]]
```

The third rule uses a pattern that will be matched by one or more triples of numbers.

```
In[20]:= ShowWalk[coords : {{_?NumberQ, _?NumberQ, _?NumberQ} ..},
           opts : OptionsPattern[]] :=
             Graphics3D[Line[coords],
           FilterRules[{opts}, Options[Graphics3D]]]
```

9. Use `PlotStyle` to highlight the two different surfaces and `MeshStyle` and `Mesh` to highlight their intersection.

```
In[21]:= f[x_, y_] := Sin[2 x - Cos[y]];
         g[x_, y_] := Sin[x - Cos[2 y]];

In[23]:= Plot3D[{f[x, y], g[x, y]}, {x, -π, π}, {y, -π, π}, Mesh → {{0.}},
          MaxRecursion → 4, MeshFunctions → (f[#1, #2] - g[#1, #2] &),
          MeshStyle → {Thick, Red}, PlotStyle → {Cyan, Yellow}]
```

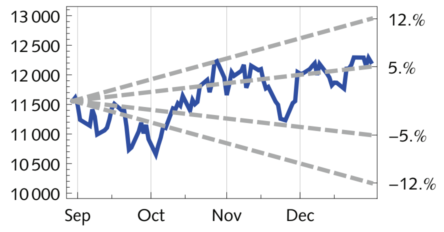
Out[23]=


10. One approach is to compute a plot range using something like the following, where `minR` is the minimum rate and `maxR` is the maximum rate specified by the user.

```
{(1.05 + minR) min, (.95 + maxR) max}
```

```
In[24]:= ClearAll[TrendPlot]

In[25]:= TrendPlot::usage =
           "TrendPlot[data,{r1,r2,…}] plots data with trend
             lines showing growth rates over time.";

In[26]:= TrendlineStyle::usage =
           "TrendlineStyle is an option for TrendPlot
             that specifies the style of the trend lines.";

In[27]:= Options[TrendPlot] =
           Join[{TrendlineStyle → Automatic}, Options[DateListPlot]];

In[28]:= TrendPlot[data_, rates_List, opts : OptionsPattern[]] :=
          Module[{min, max, tlStyle, tLine,
            rtTicks, init = data[[1, 2]], minR, maxR},

           {min, max} = {Min[data[[All, 2]]], Max[data[[All, 2]]]};
           {minR, maxR} = {Min[rates], Max[rates]};

           tlStyle = If[OptionValue[TrendlineStyle] === Automatic,
             {Dashed, Gray}, OptionValue[TrendlineStyle]];

           tLine[r_] := Flatten@{tlStyle,
               Line[{First[data], Last[data] /.
                  {d_List, val_?NumberQ} :→ {d, (1 + r) init}}]};

           rtTicks = MapThread[
             {(1 + #) init, StringForm[" `1`%", #2]} &, {rates, 100 rates}];

           DateListPlot[data, Joined → True,
            FilterRules[{opts}, Options[DateListPlot]],
            Epilog :→ Map[tLine, rates],
            PlotRange → {(1.05 + minR) min, (.95 + maxR) max},
            FrameTicks → {{Automatic, rtTicks}, {Automatic, None}}]
          ]

In[29]:= data =
           FinancialData["^DJI", {"August 30 2011", "December 30 2011"}];
```

```
In[30]:= rates = {0.05, 0.12, -0.05, -0.12};
         TrendPlot[data, rates,
          PlotStyle → {Thick, Blue},
          TrendlineStyle → {Thick, Dashed, Lighter@Gray}]

Out[31]=
```

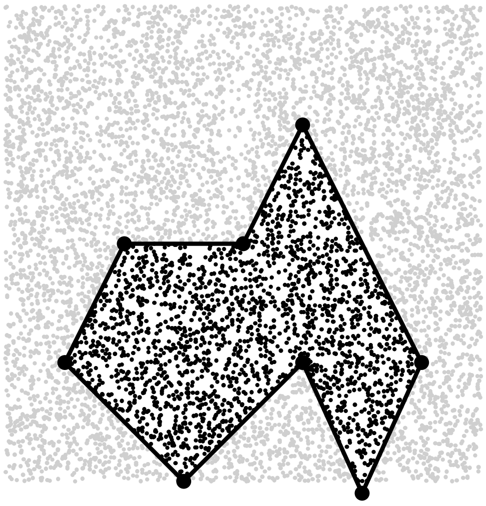


11. If the first point returned by `GatherBy` fails the `PointInPolygonQ` test, then reverse the two lists (`out` and `in`), otherwise, leave it alone.

```
In[32]:= poly = {{-0.5, 0}, {0.5, -1}, {1.5, 0},
            {2., -1.1}, {2.5, 0}, {1.5, 2}, {1., 1}, {0., 1}};
         pts = RandomReal[{-1, 3}, {7500, 2}];

In[34]:= TriangleArea[tri : {v1_, v2_, v3_}] :=
          Det[Map[PadRight[#, 3, 1] &, tri]] / 2

In[35]:= PointInPolygonQ[poly : {{_, _} ..}, pt : {x_, y_}] :=
          Module[{edges, e2, e3, e4},
           edges = Partition[poly /. {a_, b__} :→ {a, b, a}, 2, 1];
           e2 = DeleteCases[edges, {{x1_, y1_}, {x2_, y2_}} /; y1 == y2];
           e3 = DeleteCases[e2,
             {{x1_, y1_}, {x2_, y2_}} /; (Min[y1, y2] ≥ y || Max[y1, y2] < y)];
           e4 = Map[Reverse@SortBy[#, Last] &, e3];
           OddQ[Count[TriangleArea[Join[#, {pt}]] & /@ e4, _?Positive]]
          ]
```

```
In[36]:= gbPts = GatherBy[pts, PointInPolygonQ[poly, #] &];
         Graphics[{
           {PointSize[Small], If[PointInPolygonQ[poly, gbPts[[1, 1]]],
              gbPts, Reverse[gbPts]] /. {in_List, out_List} :>
              {{Black, Point@in}, {LightGray, Point@out}}},
           Thick, Line[poly /. {a_, b__} :> {a, b, a}],
           PointSize[Large], Point[poly]
          }]
```

Out[37]=



12. First create the pentatonic scale using symbolic notes.

```
In[38]:= pscale = {"C♯", "Eb", "F♯4", "Ab", "C♯2"};
```

```
In[39]:= With[{steps = 12, instr = "Vibraphone"},
           notes = RandomChoice[pscale, {steps}];
           durs = RandomChoice[Range[1 / 16, 1, 1 / 16], {steps}];
           Sound@MapThread[SoundNote[#1, #2, instr] &, {notes, durs}]
          ] // EmitSound
```

14. First set up the options structure.

```
In[40]:= Options[BrownianCompose] = {Weights → Automatic};
```

```
In[41]:= BrownianCompose[steps_Integer, instr_: "Vibraphone",
           OptionsPattern[]] := Module[{walk, durs, weights},
           weights = If[OptionValue[Weights] === Automatic,
             Table[1 / 9, {9}], OptionValue[Weights]];
           walk[n_] := Accumulate[RandomChoice[weights → Range[-4, 4], n]];
           durs = RandomChoice[Range[1 / 16, 1, 1 / 16], {steps}];
           Sound@
            MapThread[SoundNote[#1, #2, instr] &, {walk[steps], durs}]
          ]
```

```
In[42]:= BrownianCompose[18, "Marimba"] // EmitSound

In[43]:= BrownianCompose[18, "Marimba", Weights →
            Abs@RandomVariate[NormalDistribution[0, 4], 9]] // EmitSound
```

11 Dynamic expressions

11.1 *Manipulating expressions*

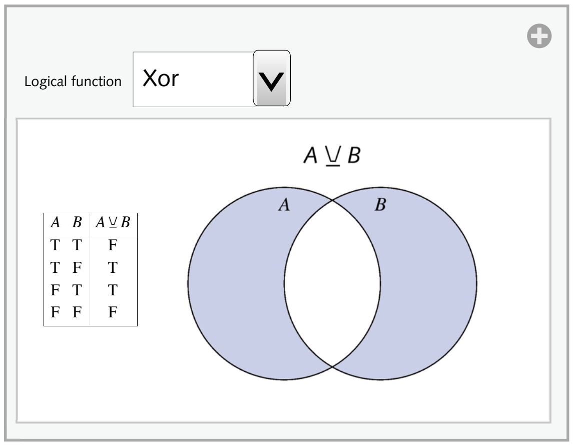
1. Create a dynamic interface that displays various diagrams and plots of the amino acids. A list of the amino acids is given by:

```
In[1]:=  ChemicalData["AminoAcids"]

Out[1]=  {Glycine, LAlanine, LSerine, LProline, LValine, LThreonine,
          LCysteine, LIsoleucine, LLeucine, LAsparagine, LAsparticAcid,
          LGlutamine, LLysine, LGlutamicAcid, LMethionine, LHistidine,
          LPhenylalanine, LArginine, LTyrosine, LTryptophan}
```

   The diagrams and plots that should be included are built into `ChemicalData`:

```
In[2]:=  StringCases[ChemicalData["Properties"],
           __ ~~ "Diagram" | (__ ~~ "Plot")] // Flatten

Out[2]=  {CHColorStructureDiagram, CHStructureDiagram,
          ColorStructureDiagram, MoleculePlot,
          SpaceFillingMoleculePlot, StructureDiagram}
```

2. Create a dynamic interface that applies several built-in effects to an image. The effects are given by `ImageEffect` and include `"Charcoal"`, `"Solarization"`, `"GaussianNoise"` and many others. See the documentation for `ImageEffect` for a complete list.
3. Modify the dynamic Venn diagram created in this section to display a truth table like that developed in Exercise 9 from Section 5.8. Include the truth table side-by-side with the Venn diagram, like in the following:



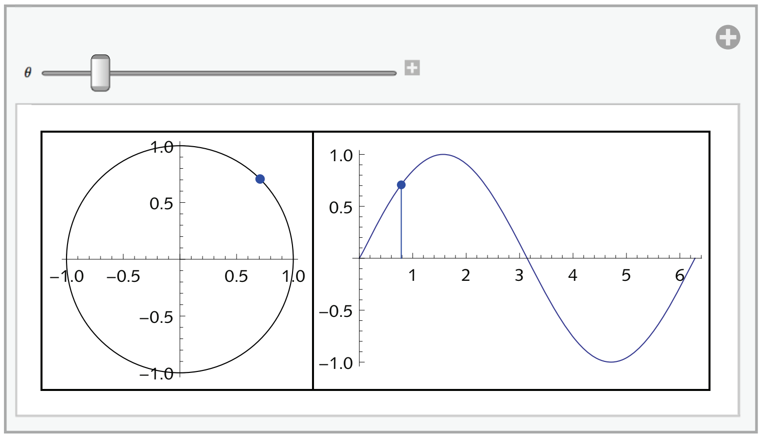
A	B	$A \veebar B$
T	T	F
T	F	T
F	T	T
F	F	F

4. Create a dynamic interface that displays some sample text using two different fonts from your system's list of fonts. Set it up so that you can select which two fonts to compare by using a pull-down menu. The list of fonts on your system is given by the following:

```
In[3]:=  fonts = FE`Evaluate[FEPrivate`GetPopupList["MenuListFonts"]];

In[4]:=  RandomSample[fonts, 3]

Out[4]= {Gurmukhi Sangam MN → Gurmukhi Sangam MN, Impact → Impact,
         DTL Albertina TOT Italic → DTL Albertina TOT Italic}
```

5. Take one of the two-dimensional random walk programs developed elsewhere in this book (for example, Sections 8.1 and 13.1) and create an animation that displays successive steps of the random walk.

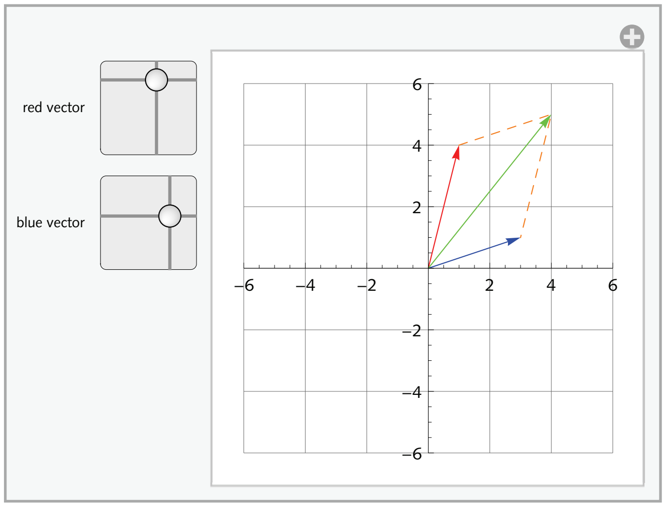
6. Create a plot of sin(θ) side-by-side with a circle and a dynamic point that moves along the curve and the circle as θ varies from 0 to 2π.



7. Modify the `Manipulate` expression that animates the hypocycloid so that the plot range deals with the situation when the radius of the inner circle is larger than the radius of the outer circle.
8. An *epicycloid* is a curve that can be generated by tracing out a fixed point on a circle that rolls around the outside of a second circle. The formula for an epicycloid is quite similar to that for the hypocycloid. The epicycloid is given parametrically by the following:

$$x = (a + b)\cos(\theta) - b\cos\left(\frac{a+b}{b}\,\theta\right),$$
$$y = (a + b)\sin(\theta) - b\sin\left(\frac{a+b}{b}\,\theta\right).$$

   Create a dynamic interface to animate the epicycloid similar to that for the hypocycloid in this section.
9. In the 1920s and 1930s the artist Marcel Duchamp created what he termed *rotoreliefs*, spinning concentric circles (and variants thereof) giving a three-dimensional illusion of depth (Duchamp 1926). Create you own rotoreliefs by starting with several concentric circles of different radii, then varying their centers around a path given by another circle, and animating.
10. Create a dynamic table that displays the temperature of several cities around the world. Include a control (pulldown menu or setter bar) to switch the display between Celsius and Fahrenheit.
11. Looking forward to Chapter 13 where we develop a full application for computing and visualizing random walks, create a dynamic interface that displays random walks, adding controls to select the number of steps from a pulldown menu, the dimension from a setter bar, and a checkbox to turn on and off lattice walks.
12. Create a visualization of two-dimensional vector addition. The interface should include either a 2D slider for each of two vectors in the plane or locators to change the position of each vector; the display should show the two vectors as well as their vector sum. Extend the solution to three dimensions. (The solution of this vector arithmetic interface is due to Harry Calkins of Wolfram Research.)



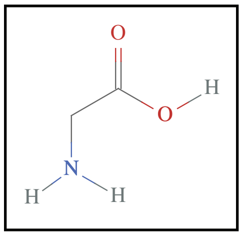
13. Create a dynamic interface to display information about a word drawn from `WordData`. The interface should include an input field for the word and use tabs to display either a definition, the Porter stem, or synonyms (try other word properties in `WordData`).

11.1 *Solutions*

1. We will put this together in two parts: first create a function to display any amino acid using one of the various diagrams; then pour it into a `Manipulate`. Note, this function is dependent upon `ChemicalData` to create the displays. You could modify it to use you own visualizations, such as the space-filling plots in Section 10.4.

```
In[1]:= AminoAcidPlot[aa_String, diagram_: "ColorStructureDiagram"] :=
          Labeled[Framed[ChemicalData[aa, diagram], ImageSize → All],
           ChemicalData[aa, "Name"], LabelStyle → Directive["Menu", 9]]

In[2]:= AminoAcidPlot["Glycine"]
```

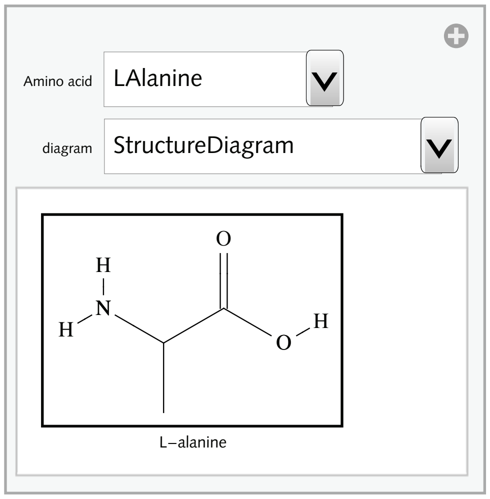
Out[2]=



glycine

```
In[3]:= Manipulate[
         AminoAcidPlot[aminoacid, diagram],
         {{aminoacid, "LAlanine", "Amino acid"}, aa},
         {diagram, {"StructureDiagram", "CHColorStructureDiagram",
           "CHStructureDiagram", "ColorStructureDiagram",
           "MoleculePlot", "SpaceFillingMoleculePlot"}},
         Initialization :> {aa = ChemicalData["AminoAcids"]}]
```

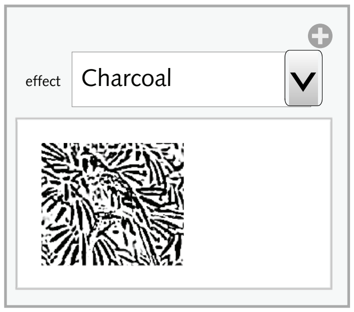
Out[3]=



2. This is a straightforward use of `Manipulate`. The lengthy parameter list forces a pulldown menu to be used as the control.

```
In[4]:= Manipulate[
          ImageEffect[
```

```
                      , effect],
          {effect, {"Charcoal", "Embossing", "OilPainting",
            "Posterization", "Solarization", "MotionBlur", "Noise",
            "GaussianNoise", "SaltPepperNoise", "PoissonNoise"}}]
```

Out[4]=


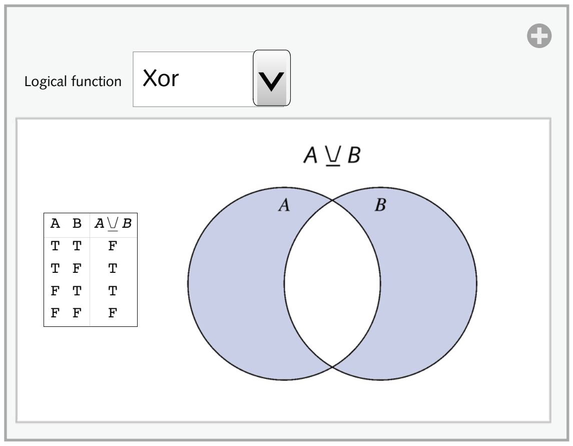
3. Here is the code for the `TruthTable` function from Exercise 9 in Section 5.8:

```
In[5]:= TruthTable[expr_, vars_] :=
         Module[{len = Length[vars], tuples, rules, table, head},
          tuples = Tuples[{True, False}, len];
          rules = (Thread[vars → #1] &) /@ tuples;
          table = Transpose[Join[Transpose[tuples], {expr /. rules}]];
          head = Append[vars, TraditionalForm[expr]];
          Grid[Prepend[table /. {True → "T", False → "F"}, head],
           Dividers → {{1 → {Thin, Black}, -1 → {Thin, Black},
               -2 → {Thin, LightGray}}, {1 → {Thin, Black},
               2 → {Thin, LightGray}, -1 → {Thin, Black}}}]]
```

This puts the truth table together with the Venn diagram using `Row`.

```
In[6]:= Manipulate[
          Row[{TruthTable[f[A, B], {A, B}], Show[RegionPlot[f @@ eqns,
              {x, -2, 2}, {y, -2, 2}, Frame → None, PlotLabel → f[A, B],
              PlotRange → {{-2, 2}, {-1.2, 1.2}}, AspectRatio → Automatic,
              MaxRecursion → 5], Graphics[{Circle[c1], Circle[c2],
               Text[Style["A", FontSlant → "Italic"], {-.5, .8}],
               Text[Style["B", FontSlant → "Italic"], {.5, .8}]}],
             ImageSize → Small]}], {{f, Xor, "Logical function"},
           {And, Or, Xor, Implies, Nand, Nor}},
          Initialization :→ {c1 = {-1/2, 0}; c2 = {1/2, 0};
            eqns = Apply[(#1 + x)^2 + (#2 + y)^2 < 1 &, {c1, c2}, {1}]},
          SaveDefinitions → True]
```

Out[6]=



Logical function: Xor

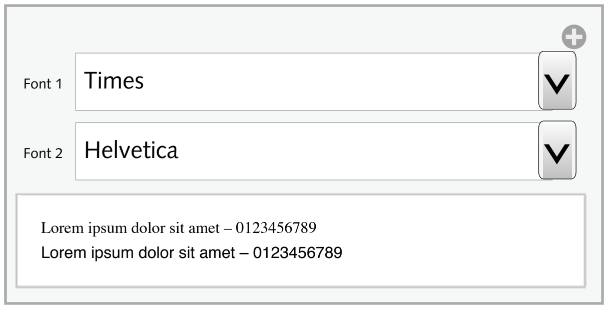
A	B	A ⊻ B
T	T	F
T	F	T
F	T	T
F	F	F

4. You might want to add some additional options to each of the `Style` expressions; for example, `FontSize` or `FontWeight`.

```
In[7]:= fonts = FE`Evaluate[FEPrivate`GetPopupList["MenuListFonts"]];
```

```
In[8]:= Manipulate[
         Column[{
           Style["Lorem ipsum dolor sit amet – 0123456789",
            FontFamily → font1],
           Style["Lorem ipsum dolor sit amet – 0123456789",
            FontFamily → font2]}],
         {{font1, "Times", "Font 1"}, fonts},
         {{font2, "Helvetica", "Font 2"}, fonts}, SaveDefinitions → True]
```

Out[8]=



```
Font 1  Times
Font 2  Helvetica

Lorem ipsum dolor sit amet – 0123456789
Lorem ipsum dolor sit amet – 0123456789
```

5. First load the package that contains the random walk code. You could use you own implementation as well.

```
In[9]:= << PwM`RandomWalks`
```

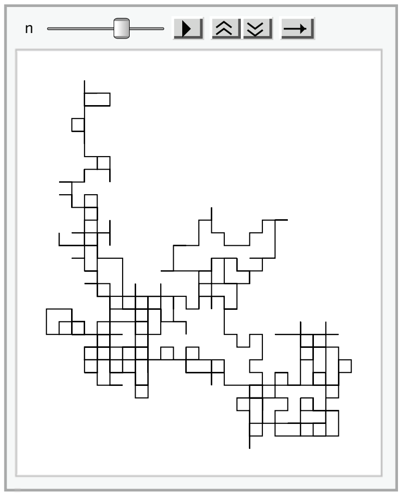
Create a 1000-step, two-dimensional, lattice walk.

```
In[10]:= rw = RandomWalk[1000, Dimension → 2, LatticeWalk → True];
```

This is a basic start. `Take` is used to display successive increments. Note the need for the 1 in the parameter list to insure that steps only take on integer values.

```
In[11]:= Animate[
          Graphics[Line[Take[rw, n]]],
          {n, 2, Length[rw], 1}]
```

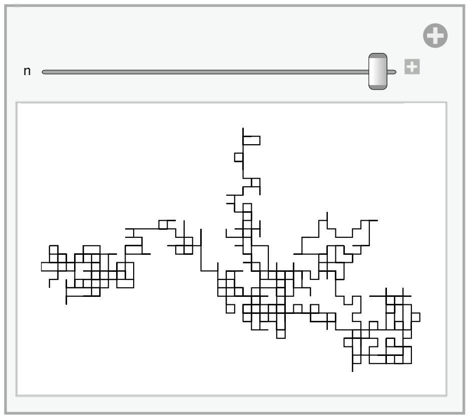
Out[11]=



The output above suffers from the fact that the display jumps around a lot as *Mathematica* tries to figure out a sensible plot range for *each* frame. Instead, we should fix the plot range for *all* frames to avoid this jumpiness. This is done in the definitions for `xran` and `yran` in the `Initialization` below.

```
In[12]:= Manipulate[
          Graphics[Line[Take[rw, n]], PlotRange → {xran, yran}],
          {n, 2, Length[rw], 1},
          Initialization :→ {
            rw = RandomWalk[1000, Dimension → 2, LatticeWalk → True];
            {xran, yran} = Map[{Min[#1], Max[#1]} &, Transpose[rw]]}]
```

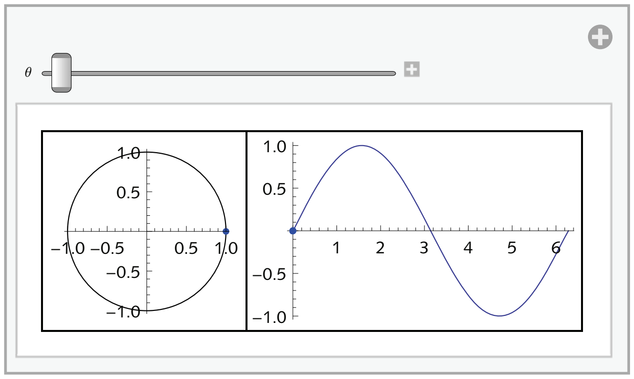
Out[12]=



6. Putting the two graphics pieces (`Graphics[…]` and `Plot[…]`) in a grid gives you finer control over their placement and formatting.

```
In[13]:= Manipulate[Grid[{{Graphics[
             {Circle[], Blue, PointSize[.04], Point[{Cos[θ], Sin[θ]}]},
             Axes → True], Plot[Sin[x], {x, 0, 2 π}, ImageSize → 300,
             Epilog → {Blue, Line[{{θ, 0}, {θ, Sin[θ]}}], PointSize[.025],
               Point[{θ, Sin[θ]}]}]}}, Frame → All], {θ, 0, 2 π}]
```

Out[13]=

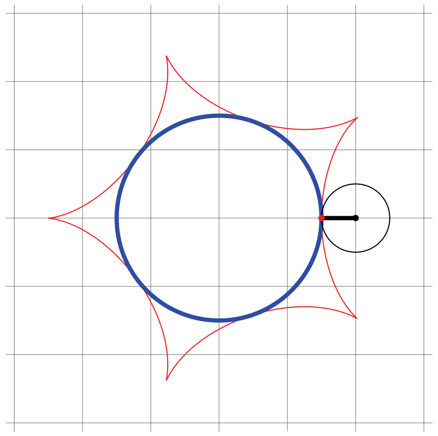


8. Just a few modifications to the code for the hypocycloid are needed: use the formula for the epicycloid; change the center of the rotating circle so that its radius is $R + r$, not $R - r$; and modify the plot range.

```
In[14]:= EpicycloidPlot[R_, r_, θ_] := Module[{epicycloid, center},
           epicycloid[{a_, b_}, t_] :=
            {(a + b) Cos[t] - b Cos[t (a + b)/b], (a + b) Sin[t] + b Sin[t (a + b)/b]};
           center[th_, R1_, r2_] := (R1 + r2) {Cos[th], Sin[th]};
           Show[{
             ParametricPlot[epicycloid[{R, r}, t],
              {t, 0, θ}, PlotStyle → Red, Axes → None],
             Graphics[{
               {Blue, Thick, Circle[{0, 0}, R]},
               {Circle[center[θ, R, r], r]},
               {PointSize[.015], Point[center[θ, R, r]]},
               {Thick, Line[{center[θ, R, r], epicycloid[{R, r}, θ]}]},
               {Red, PointSize[.015], Point[epicycloid[{R, r}, θ]]}
            }]}, PlotRange → 1.5 (R + r), GridLines → Automatic]]
```

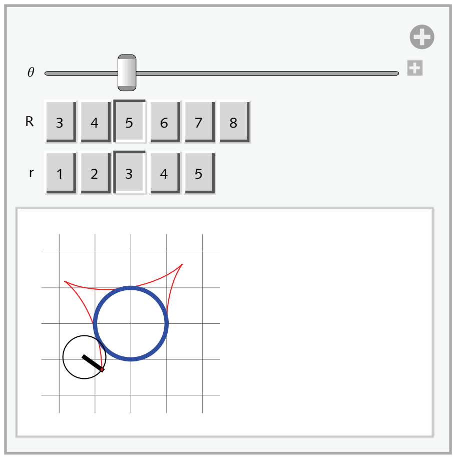
First, create a static image.

```
In[15]:= EpicycloidPlot[3, 1, 2 π]
```

Out[15]=


And here is the dynamic version.

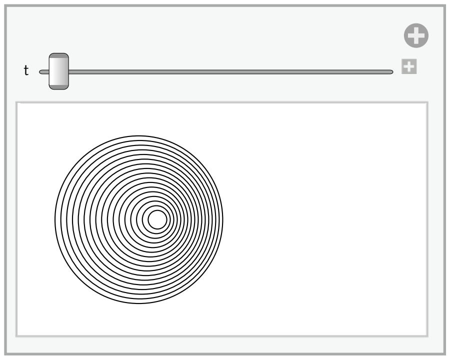
```
In[16]:= Manipulate[EpicycloidPlot[R, r, θ],
          {θ, 0 + 0.01, 2 Denominator[(R - r) / r] π},
          {R, {3, 4, 5, 6, 7, 8}, Setter},
          {r, {1, 2, 3, 4, 5}, Setter}, SaveDefinitions → True]
```

Out[16]=


9. Modify the radii and the centers to get different effects. Try using transparent disks instead of circles.

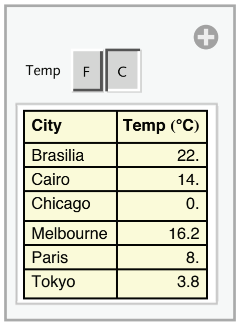
```
In[17]:= Manipulate[
          Graphics[
           Table[Circle[r / 4 {Cos[t], Sin[t]}, 1.1 - r], {r, .2, 1, .05}],
           PlotRange → 1],
          {t, 0, 2 π, .1},
          TrackedSymbols :→ {t}]
```

Out[17]=



10. The `Units` package contains the function `ConvertTemperature` as well as `Centigrade` and `Fahrenheit` scales. We load it as part of the initialization. Since this might take some time, we also set `SyncrhonousUpdating` to `False` to make sure the evaluation does not time out before the default five second limit in `Manipulate` is reached.

```
In[18]:= Manipulate[
          Grid[Join[{{
              Style["City", FontWeight → "Bold"],
              Style[
               StringForm["Temp (°`1`)", sym], FontWeight → "Bold"]}},
            Map[{#,
               If[sym === "C",
                WeatherData[#, "Temperature"],
                Units`ConvertTemperature[WeatherData[#, "Temperature"],
                 Units`Centigrade, Units`Fahrenheit]
               ]} &, {"Brasilia", "Cairo", "Chicago",
              "Melbourne", "Paris", "Tokyo"}]
           ],
           Frame → All, Background → LightYellow,
           Alignment → {{Left, Right, Right}, Automatic},
           BaseStyle → {FontFamily → "Helvetica"}],
          {{sym, "C", "Temp"}, {"F", "C"}},
          Initialization :→ {Needs["Units`"]}, SynchronousUpdating → False,
          FrameMargins → 0, Deployed → True]
```

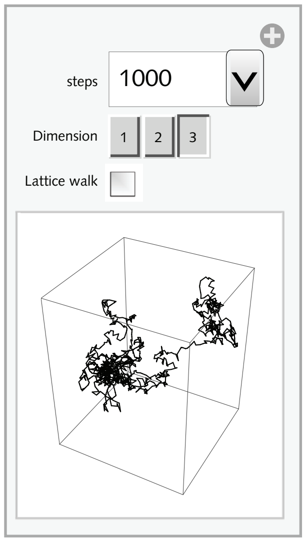
Out[18]=


City	Temp (°C)
Brasilia	22.
Cairo	14.
Chicago	0.
Melbourne	16.2
Paris	8.
Tokyo	3.8

11. Using the programs developed in Section 13.1, here is the code, including a pulldown menu for the steps parameter, a setter bar for the dimension parameter, and a checkbox for the lattice parameter.

```
In[19]:= Manipulate[
           ShowWalk@
            RandomWalk[steps, Dimension → dim, LatticeWalk → latticeQ],
           {steps, {100, 250, 500, 750, 1000, 10 000}},
           {{dim, 1, "Dimension"}, {1, 2, 3}},
           {{latticeQ, True, "Lattice walk"}, {True, False}},
           Initialization :→ Needs["PWM`RandomWalks`"],
           SaveDefinitions → True]
```

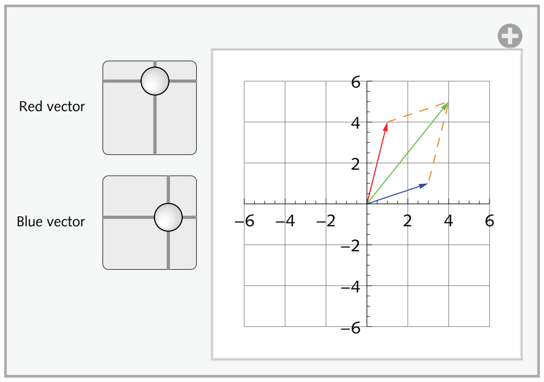
Out[19]=



12. Here is the solution using `Slider2D`. Using `Locator` instead is left for the reader.

```
In[20]:= Manipulate[
          Graphics[{
            Red, Arrow[{{0, 0}, pt1}],
            Blue, Arrow[{{0, 0}, pt2}],
            Green, Arrow[{{0, 0}, pt1 + pt2}],
            Dashed, Orange, Line[{pt1, pt1 + pt2, pt2}]},
           PlotRange → 6, Axes → True, GridLines → Automatic],
          {{pt1, {1, 4}, "Red vector"}, {-5, -5}, {5, 5}},
          {{pt2, {3, 1}, "Blue vector"}, {-5, -5}, {5, 5}},
          ControlPlacement → Left]
```

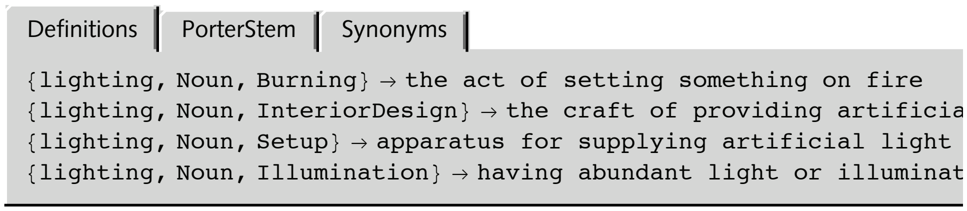
Out[20]=



13. First create a `TabView` for one word.

```
In[21]:= With[{word = "lighting"},
          TabView[{
            "Definitions" → TableForm@WordData[word, "Definitions"],
            "PorterStem" → WordData[word, "PorterStem"],
            "Synonyms" → TableForm@WordData[word, "Synonyms"]
           }]
         ]
```

Out[21]=

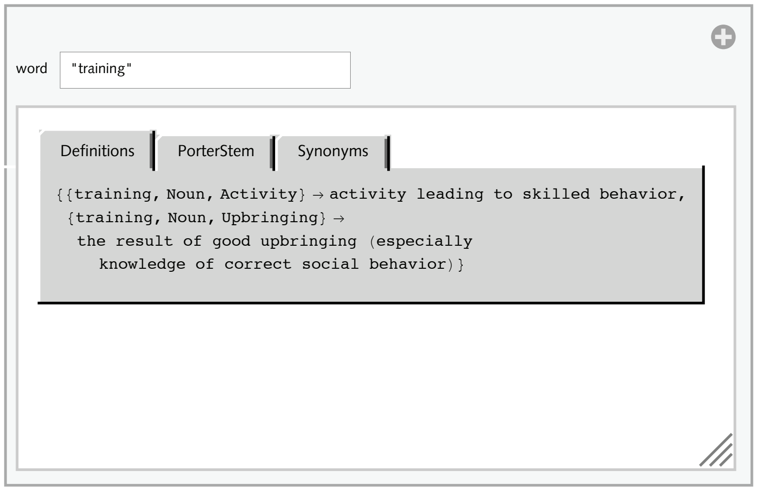


```
{lighting, Noun, Burning} → the act of setting something on fire
{lighting, Noun, InteriorDesign} → the craft of providing artificia
{lighting, Noun, Setup} → apparatus for supplying artificial light
{lighting, Noun, Illumination} → having abundant light or illuminat
```

Then make the word the parameter inside a `Manipulate`, using `InputField` as the `ControlType`. Use `ToString` to insure that the expression passed to `WordData` is a string, regardless of what is typed in the input field.

```
In[22]:= Manipulate[
          TabView[{
            "Definitions" → WordData[ToString@word, "Definitions"],
            "PorterStem" → WordData[ToString@word, "PorterStem"],
            "Synonyms" → WordData[ToString@word, "Synonyms"]
           }],
          {word, "training"}, ControlType → InputField,
          ContentSize → {400, 200}]
```

Out[22]=



```
{{training, Noun, Activity} → activity leading to skilled behavior,
 {training, Noun, Upbringing} →
  the result of good upbringing (especially
    knowledge of correct social behavior)}
```

11.2 *The structure of dynamic expressions*

1. Display a random word from the dictionary (`DictionaryLookup`) that changes every second.
2. Create a dynamic interface consisting of a locator constrained to the unit circle.
3. Create a dynamic interface that controls one sphere rotating about another.

11.2 *Solutions*

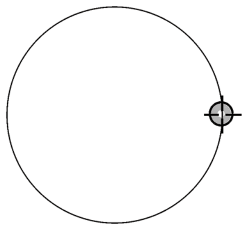
1. Use the `UpdateInterval` option to `Dynamic`.

```
In[23]:= Dynamic[RandomChoice[DictionaryLookup[]], UpdateInterval → 5]

Out[23]= Waksman
```

2. `Normalize` takes a vector as input and returns a unit vector.

```
In[24]:= DynamicModule[{pt = {1, 0}}, Graphics[{
            Circle[],
            Locator[Dynamic[pt, (pt = Normalize[#]) &]]
           }]]
```

Out[24]=



11.3 *Examples and applications*

1. Here are data on Nobel prizes in the fields of chemistry, medicine, and physics, available from the National Bureau of Economic Research.

```
In[1]:= data = Import[
           "http://www.nber.org/nobel/Jones_Weinberg_2011_PNAS.xlsx",
           {"XLSX", "Data", 1}];

In[2]:= Take[data, 4]

Out[2]= {{name, field, year_birth, year_prize, year_research_mid,
          year_death, TheoryOrTheoryAndEmpirical, age_highdegree},
         {Van'T Hoff, Jacobus Henricus, Chemistry,
          1852., 1901., 1885., 1911., 1., 22.},
         {Fischer, Hermann Emil, Chemistry, 1852., 1902.,
          1895., 1919., 0., 22.}, {Arrhenius, Svante August,
          Chemistry, 1859., 1903., 1884., 1927., 1., 25.}}
```

Create a `TabView` visualization showing the age of each prize recipient vs. the year of prize award. Include one tab for each of the three fields given in the data and also include a plot label that displays the mean age at award for each field.

2. Using `FunctionsWithAttribute` developed in Section 5.6, create a paneled interface that displays all built-in functions with a specified attribute. Include an input field control to allow the user to type in an attribute. Do likewise for `FunctionsWithOption` also developed in Section 5.6.

3. Create a dynamic interface that displays twenty random points in the unit square whose locations are randomized each time you click your mouse on the graphic display of these points. Add a checkbox to toggle the display of the shortest path (`FindShortestTour`) through the points.

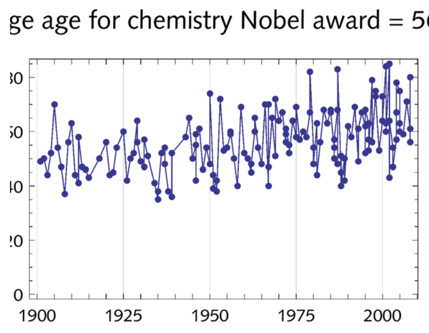
4. Create a similar dynamic interface to that in the industrial production index problem in this section but comparing industrial production with unemployment rates with retail sales data over the last twenty years or some other suitable time period. Annual and historical retail sales data are available at the US Census Bureau (www.census.gov/retail); unemployment data are available at the US Bureau of Labor Statistics (www.bls.gov/cps/cpsatabs.htm); industrial production indices are available at the US Federal Reserve System (www.federalreserve.gov/releases/g17/).

11.3 Solutions

1. Import the data only; the first four columns give name, field, birth year, award year.

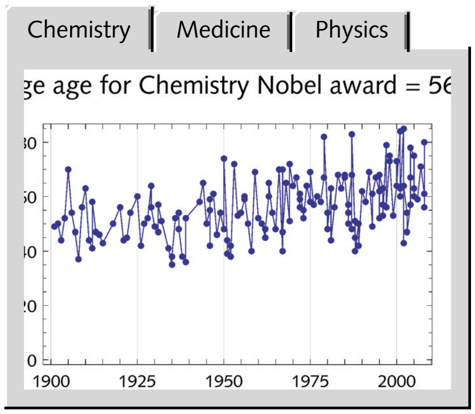
```
In[1]:= data = Import[
           "http://www.nber.org/nobel/Jones_Weinberg_2011_PNAS.xlsx",
           {"XLSX", "Data", 1, All, {1, 2, 3, 4}}];

In[2]:= data[[{1, -1}]]

Out[2]= {{name, field, year_birth, year_prize},
         {Nambu, Yoichiro, Physics, 1921., 2008.}}

In[3]:= data[[-1]] /. {a__String, birth_Real, award_Real} :>
          {a, birth, award, award - birth}

Out[3]= {Nambu, Yoichiro, Physics, 1921., 2008., 87.}

In[4]:= nobelData = data[[2 ;; -1]] /. {a__String, birth_Real, award_Real} :>
            {a, birth, award, award - birth};

In[5]:= chem = Cases[nobelData, {name_String, "Chemistry", rest__}];
        med = Cases[nobelData, {name_String, "Medicine", rest__}];
        physics = Cases[nobelData, {name_String, "Physics", rest__}];

In[8]:= timeChem = chem[[All, {4, 5}]];
        timeMed = med[[All, {4, 5}]];
        timePhysics = physics[[All, {4, 5}]];

In[11]:= DateListPlot[Tooltip[timeChem /. {a_, b_} :> {{Round@a}, b}],
          Joined → True, Mesh → All, PlotLabel →
           StringForm["Average age for chemistry Nobel award = `1`",
            Mean[timeChem[[All, 2]]]]]
```

Out[11]=



```
In[12]:= TabView[
          MapThread[
           #1 → DateListPlot[Tooltip[#2 /. {a_, b_} :→ {{Round@a}, b}],
              Joined → True, Mesh → All, PlotLabel →
               StringForm["Average age for `1` Nobel award = `2`",
                #1, Mean[#2[[All, 2]]]]] &,
           {{"Chemistry", "Medicine", "Physics"},
            {timeChem, timeMed, timePhysics}}]]
```

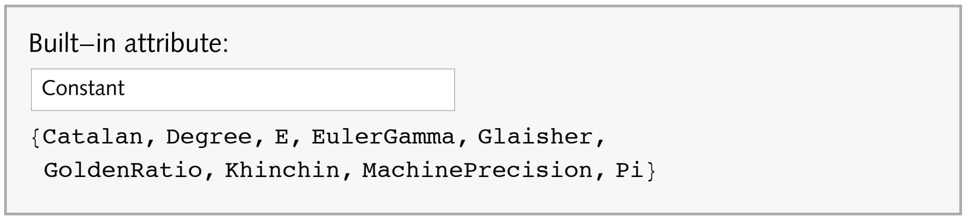
Out[12]=



2. First, here is the interface for listing all functions with a given attribute. We have used the `Initialization` option to `DynamicModule` to define the function `FunctionsWithAttributes`.

```
In[13]:= Panel[DynamicModule[{att = Constant},
           Column[{
             Style["Built-in attribute:", "Menu"],
             InputField[Dynamic[att]],
             Dynamic@TextCell[FunctionsWithAttribute[att]]
            }], Initialization :→ {FunctionsWithAttribute[attrib_Symbol] :=
              Select[Names["System`*"], MemberQ[Attributes[#], attrib] &]}
          ], ImageSize → {360, Automatic}]
```

Out[13]=



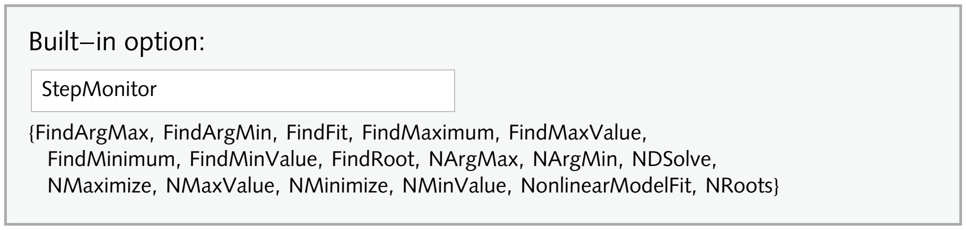
Built–in attribute:

Constant

```
{Catalan, Degree, E, EulerGamma, Glaisher,
 GoldenRatio, Khinchin, MachinePrecision, Pi}
```

Here is the panel for `FunctionsWithOption`.

```
In[14]:= Panel[DynamicModule[{option = StepMonitor},
           Column[{
             Style["Built-in option:", "Menu"],
             InputField[Dynamic[option]],
             Dynamic[FunctionsWithOption[option]]
            }], Initialization :→
            {FunctionsWithOption[opt_Symbol] := Quiet[Select[Names[
                 "System`*"], MemberQ[Options[Symbol[#]], opt, {2}] &]]}
          ], ImageSize → {360, Automatic}]
```



Out[14]=

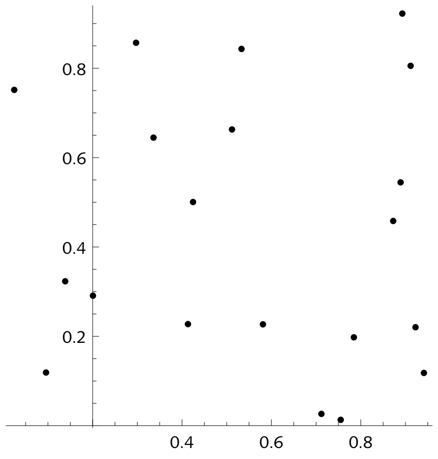
```
Built–in option:
StepMonitor
{FindArgMax, FindArgMin, FindFit, FindMaximum, FindMaxValue,
  FindMinimum, FindMinValue, FindRoot, NArgMax, NArgMin, NDSolve,
  NMaximize, NMaxValue, NMinimize, NMinValue, NonlinearModelFit, NRoots}
```

3. Create a static version of the problem; we use `GraphicsComplex` to display the points and the tour.

```
In[15]:= pts = RandomReal[1, {20, 2}];
```

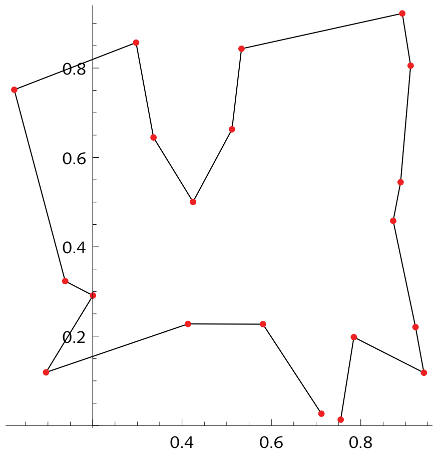
```
In[16]:= Graphics[GraphicsComplex[pts, Point@Range[Length[pts]]],
          Axes → Automatic]
```

Out[16]=



```
In[17]:= tour = Last[FindShortestTour[pts]];
         Graphics[GraphicsComplex[pts, {Line[tour], Red,
            PointSize[.015], Point[tour]}], Axes → Automatic]
```

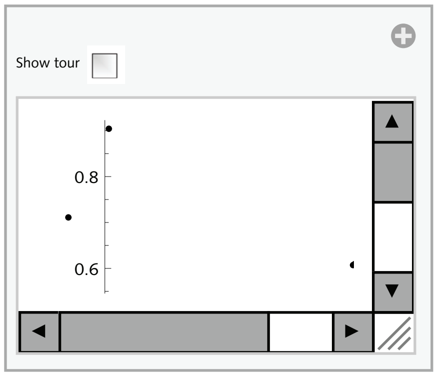
Out[18]=



Here is the dynamic interface using `EventHandler` to choose a new set of random points with each mouse click.

```
In[19]:= Manipulate[
          DynamicModule[{pts = RandomReal[1, {20, 2}], tour},
           tour = Dynamic[Last[FindShortestTour[pts]]];
           EventHandler[
            Dynamic[Graphics[GraphicsComplex[pts, If[Not[showtour],
                Point@Range[Length[pts]], {Line[tour], Red,
                 PointSize[Medium], Point[tour]}]], Axes → Automatic]],
            {"MouseClicked" :→ (pts = RandomReal[1, {20, 2}])}
           ]],
          {{showtour, False, "Show tour"}, {True, False}},
          ContentSize → {220, 140}]
```

Out[19]=



A suggested addition would be to add a control to change the number of points that are used. But be careful: traveling salesman type problems are notoriously hard; in fact they are known to be *NP*-

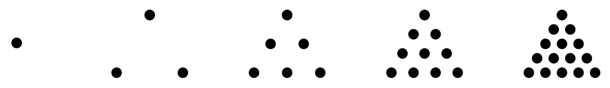
hard, meaning they cannot be computed in polynomial time. See Lawler et al. (1985) for more on traveling salesman problems.

12 Optimizing *Mathematica* programs

12.2 *Efficient programs*

1. Modify `AverageTiming` to return both the average time and the result of evaluating its argument, mirroring the behavior of `Timing` and `AbsoluteTiming`.
2. The *n*th triangular number is defined as the sum of the integers 1 through *n*. They are so named because they can be represented visually by arranging rows of dots in a triangular manner (Figure 12.1). Program several different approaches to computing triangular numbers and compare their efficiency.

FIGURE 12.1. *Pictorial representation of the first five triangular numbers.*



3. Several different implementations of the Hamming distance computation were given in Section 5.8; some run much faster than others. For example, the version with bit operators runs about one-and-a-half orders of magnitude faster than the version using `Count` and `MapThread`. Using some of the concepts from this section, determine what is causing these differences.

```
In[1]:= HammingDistance1[lis1_, lis2_] :=
         Count[MapThread[SameQ, {lis1, lis2}], False]

In[2]:= HammingDistance2[lis1_, lis2_] := Total[BitXor[lis1, lis2]]

In[3]:= sig1 = RandomInteger[1, {10^6}];

In[4]:= sig2 = RandomInteger[1, {10^6}];

In[5]:= Timing[HammingDistance1[sig1, sig2]]

Out[5]= {0.459499, 498 955}

In[6]:= Timing[HammingDistance2[sig1, sig2]]

Out[6]= {0.00906, 498 955}
```

12.2 *Solutions*

1. Collect the results of the `Table` and pull out the parts needed – the timings and the result.

```
In[1]:= SetAttributes[AverageTiming, HoldAll]

In[2]:= AverageTiming[expr_, trials_] := Module[{lis},
          lis = Table[AbsoluteTiming[expr], {trials}];
```

```
           {Mean[lis[[All, 1]]], lis[[1, 2]]}
          ]
In[3]:=  AverageTiming[FactorInteger[50! + 1], 5]
Out[3]=  {1.311202,
           {{149, 1}, {3989, 1}, {74 195 127 103, 1}, {6 854 870 037 011, 1},
            {100 612 041 036 938 568 804 690 996 722 352 077, 1}}}
```

2. The first implementation essentially performs a transpose of the two lists, wrapping `SameQ` around each corresponding pair of numbers. It then does a pattern match (`Count`) to determine which expressions of the form `SameQ`[$expr_1$, $expr_2$] return `False`.

```
In[4]:=  HammingDistance1[lis1_, lis2_] :=
           Count[MapThread[SameQ, {lis1, lis2}], False]
In[5]:=  HammingDistance2[lis1_, lis2_] := Total[BitXor[lis1, lis2]]
In[6]:=  sig1 = RandomInteger[1, {10^6}];
In[7]:=  sig2 = RandomInteger[1, {10^6}];
```

In this case, it is the threading that is expensive rather than the pattern matching with `Count`.

```
In[8]:=  res = MapThread[SameQ, {sig1, sig2}]; // Timing
Out[8]=  {0.469637, Null}
In[9]:=  Count[res, False] // Timing
Out[9]=  {0.049376, 499 582}
```

The reason the threading is expensive can be seen by turning on the packing message as discussed in this section.

```
In[10]:=  SetSystemOptions["PackedArrayOptions" → "UnpackMessage" → True]
Out[10]=  PackedArrayOptions → {ListableAutoPackLength → 250,
            PackedArrayMathLinkRead → True, PackedArrayPatterns → True,
            PackedRange → True, UnpackMessage → True}
In[11]:=  res = MapThread[SameQ, {sig1, sig2}];
```

```
Developer`FromPackedArray::punpack1 : Unpacking array with dimensions {1000000}. ≫
```

The other factors contributing to the significant timing differences have to do with the fact that `BitXor` has the `Listable` attribute. `MapThread` does not. And so, `BitXor` can take advantage of specialized (compiled) codes internally to speed up its computations.

```
In[12]:=  Attributes[BitXor]
Out[12]=  {Flat, Listable, OneIdentity, Orderless, Protected}
In[13]:=  Attributes[MapThread]
Out[13]=  {Protected}
```

```
In[14]:= Timing[temp = BitXor[sig1, sig2];]

Out[14]= {0.00373, Null}
```

And finally, compute the number of 1s using `Total` which is extremely fast at adding lists of numbers.

```
In[15]:= Timing[Total[temp];]

Out[15]= {0.003227, Null}
```

Return the packed array messaging to its default value.

```
In[16]:= SetSystemOptions["PackedArrayOptions" → "UnpackMessage" → False];
```

3. A first attempt, using a brute force approach, is to total the list `{1, 2, …, n}` for each *n*.

```
In[17]:= TriangularNumber[n_] := Total[Range[n]]

In[18]:= Table[TriangularNumber[i], {i, 1, 100}]

Out[18]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120,
          136, 153, 171, 190, 210, 231, 253, 276, 300, 325, 351, 378,
          406, 435, 465, 496, 528, 561, 595, 630, 666, 703, 741, 780,
          820, 861, 903, 946, 990, 1035, 1081, 1128, 1176, 1225, 1275,
          1326, 1378, 1431, 1485, 1540, 1596, 1653, 1711, 1770, 1830,
          1891, 1953, 2016, 2080, 2145, 2211, 2278, 2346, 2415, 2485,
          2556, 2628, 2701, 2775, 2850, 2926, 3003, 3081, 3160, 3240,
          3321, 3403, 3486, 3570, 3655, 3741, 3828, 3916, 4005, 4095,
          4186, 4278, 4371, 4465, 4560, 4656, 4753, 4851, 4950, 5050}

In[19]:= Timing[TriangularNumber[10^7]]

Out[19]= {3.86688, 50 000 005 000 000}
```

A second approach uses iteration. As might be expected, this is the slowest of the approaches here.

```
In[20]:= TriangularNumber2[n_] := Fold[#1 + #2 &, 0, Range[n]]

In[21]:= Timing[TriangularNumber2[10^7]]

Out[21]= {7.34643, 50 000 005 000 000}
```

This is a situation where some mathematical knowledge is useful. The *n*th triangular numbers is just the $(n+1)$th binomial coefficient $\binom{n+1}{2}$.

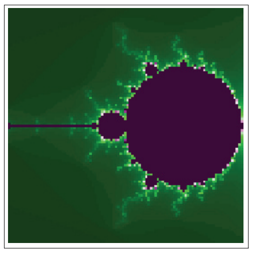
```
In[22]:= TriangularNumber3[n_] := Binomial[n + 1, 2]

In[23]:= Timing[TriangularNumber3[10^7]]

Out[23]= {0.000045, 50 000 005 000 000}
```

12.3 *Parallel processing*

1. In the eighteenth century, Leonhard Euler proved that all even perfect numbers must be of the form $2^{p-1}(2^p - 1)$ for $2^p - 1$ prime. (No one has yet proved that any odd perfect numbers exist.) Use this fact to find all even perfect numbers for $p < 10^4$.

2. The following code can be used to create a plot of the Mandelbrot set. It uses `Table` to compute the value for each point in the complex plane on a small grid. We have deliberately chosen a relatively coarse grid ($n = 100$) as this is an intensive and time-consuming computation. The last argument to `NestWhileList`, 250 here, sets a limit on the number of iterations that can be performed for each input.

```
In[1]:= Mandelbrot[c_] :=
          Length[NestWhileList[#^2 + c &, 0, Abs[#] < 2 &, 1, 250]]

In[2]:= data = With[{n = 100}, Table[Mandelbrot[x + I y],
               {y, -0.5, 0.5, 1/n}, {x, -1.75, -0.75, 1/n}]];

In[3]:= ArrayPlot[data, ColorFunction → "GreenPinkTones"]

Out[3]=
```



Increase the resolution of the graphic by running the computation in parallel.

12.3 *Solutions*

1. First we find those values of p for which $2^p - 1$ is prime. This first step is quite compute-intensive; fortunately, it parallelizes well.

```
In[1]:= LaunchKernels[]

Out[1]= {KernelObject[1, local], KernelObject[2, local],
         KernelObject[3, local], KernelObject[4, local]}

In[2]:= primes = Parallelize[Select[Range[10 000], PrimeQ[2^# - 1] &]]

Out[2]= {2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521,
         607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941}
```

So for each of the above values of the list `primes`, $2^{p-1}(2^p - 1)$ will be perfect (thanks to Euler).

```
In[3]:= perfectLis = Map[2^(#-1) (2^# - 1) &, primes];
```

And finally, a check.

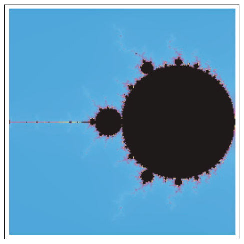
```
In[4]:= perfectQ[j_] := Total[Divisors[j]] == 2 j;

In[5]:= Map[perfectQ, perfectLis]

Out[5]= {True, True, True, True, True, True, True, True, True, True, True,
         True, True, True, True, True, True, True, True, True, True, True}

In[6]:= CloseKernels[];
```

These are very large numbers indeed.

```
In[7]:= 2^(#-1) (2^# - 1) &[9941] // N

Out[7]= 5.988854963873362 × 10^5984
```

2. Only two changes are required to run this in parallel – distribute the definition for `Mandelbrot` and change `Table` to `ParallelTable`. Of course, to increase the resolution, the grid now has many more divisions in each direction ($n = 500$).

```
In[8]:= Mandelbrot[c_] :=
         Length[NestWhileList[#^2 + c &, 0, Abs[#] < 2 &, 1, 250]]

In[9]:= LaunchKernels[]

Out[9]= {KernelObject[5, local], KernelObject[6, local],
         KernelObject[7, local], KernelObject[8, local]}

In[10]:= DistributeDefinitions[Mandelbrot]

Out[10]= {Mandelbrot}

In[11]:= data = With[{n = 500}, ParallelTable[Mandelbrot[x + ⅈ y],
              {y, -0.5, 0.5, 1/n}, {x, -1.75, -0.75, 1/n}]];

In[12]:= ArrayPlot[data, ColorFunction → "CMYKColors"]

Out[12]=
```



12.4 *Compiling*

1. Create a compiled function that computes the distance to the origin of a two-dimensional point. Then compare it to some of the built-in functions such as `Norm` and `EuclideanDistance` for a large set of points. If you have a C compiler installed on your computer, use the `Compile` option, `CompilationTarget → "C"` and compare the results.

2. Modify the previous exercise under the assumption that complex numbers are given as input to your compiled function.
3. Many other iteration functions can be used for the Julia set computation. Experiment with some other functions such as $c \sin(z)$, $c\, e^z$, or Gaston Julia's original function:

   $$z^4 + z^3/(z-1) + z^2/\left(z^3 + 4 z^2 + 5\right) + c.$$

   For these functions, you will have to adjust the test to determine if a point is unbounded upon iteration. Try `(Abs[Im[#]] > 50 &)`.

12.4 *Solutions*

1. First, create a test point with which to work.

```
In[1]:= pt = RandomReal[1, {2}]

Out[1]= {0.333881, 0.135321}
```

The following does not quite work because the default pattern is expected to be a flat expression.

```
In[2]:= distReal = Compile[{{p, _Real}}, Sqrt[First[p]^2 + Last[p]^2],
          RuntimeAttributes → {Listable}, Parallelization → True]

   Compile::part : Part specification p[[1]] cannot be compiled since the argument
       is not a tensor of sufficient rank. Evaluation will use the uncompiled function. ≫

Out[2]= CompiledFunction[{p}, √(First[p]^2 + Last[p]^2), -CompiledCode-]
```

Give a third argument to the pattern specification to deal with this: `{p, _Real, 1}`.

```
In[3]:= ArrayDepth[pt]

Out[3]= 1

In[4]:= distReal = Compile[{{p, _Real, 1}}, Sqrt[First[p]^2 + Last[p]^2],
          RuntimeAttributes → {Listable}, Parallelization → True]

Out[4]= CompiledFunction[{p}, √(First[p]^2 + Last[p]^2), -CompiledCode-]

In[5]:= distReal[pt]

Out[5]= 0.360261
```

Check it against the built-in function:

```
In[6]:= Norm[pt]

Out[6]= 0.360261
```

Check that it threads properly over a list of points.

```
In[7]:=  pts = RandomReal[1, {3, 2}]

Out[7]=  {{0.223743, 0.810299}, {0.873595, 0.72168}, {0.951892, 0.547475}}

In[8]:=  distReal[pts]

Out[8]=  {0.840622, 1.13313, 1.0981}
```

`Norm` does not have the `Listable` attribute so it must be mapped over the list.

```
In[9]:=  Map[Norm, pts]

Out[9]=  {0.840622, 1.13313, 1.0981}

In[10]:= distReal[pts] == Map[Norm, pts]

Out[10]= True
```

Now scale up the size of the list of points and check efficiency.

```
In[11]:= pts = RandomReal[1, {10^6, 2}];

In[12]:= AbsoluteTiming[distReal[pts];]

Out[12]= {0.109824, Null}

In[13]:= AbsoluteTiming[Map[Norm, pts];]

Out[13]= {0.113652, Null}

In[14]:= distReal[pts] == Map[Norm, pts]

Out[14]= True
```

Compiling to C (assuming you have a C compiler installed), speeds things up even more.

```
In[15]:= distReal = Compile[{{p, _Real, 1}},
           Sqrt[First[p]^2 + Last[p]^2], RuntimeAttributes → {Listable},
           Parallelization → True, CompilationTarget → "C"]

Out[15]= CompiledFunction[{p}, Sqrt[First[p]^2 + Last[p]^2], -CompiledCode-]
```

You can squeeze a little more speed out of these functions by using `Part` instead of `First` and `Last`.

```
In[16]:= distReal2 = Compile[{{p, _Real, 1}},
           Sqrt[p[[1]]^2 + p[[2]]^2], RuntimeAttributes → {Listable},
           Parallelization → True, CompilationTarget → "C"]

Out[16]= CompiledFunction[{p}, Sqrt[p⟦1⟧^2 + p⟦2⟧^2], -CompiledCode-]
```

```
In[17]:= AbsoluteTiming[distReal2[pts];]

Out[17]= {0.059632, Null}
```

As an aside, the mean distance to the origin for random points in the unit square approaches the following, asymptotically.

```
In[18]:= NIntegrate[√(x² + y²), {x, 0, 1}, {y, 0, 1}]

Out[18]= 0.765196

In[19]:= Mean@distReal[pts]

Out[19]= 0.765452
```

2. We need to make just three slight modifications to the code from the previous exercise: remove the rank specification; specify `Complex` as the type; extract the real and imaginary parts to do the norm computation.

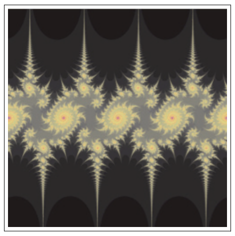
```
In[20]:= Clear[distComplex];
         distComplex = Compile[{{z, _Complex}}, Sqrt[Re[z]² + Im[z]²],
           RuntimeAttributes → {Listable}, Parallelization → True]

Out[21]= CompiledFunction[{z}, √(Re[z]² + Im[z]²), -CompiledCode-]

In[22]:= pts = RandomComplex[1, {3}]

Out[22]= {0.349519 + 0. i, 0.506776 + 0. i, 0.153516 + 0. i}

In[23]:= distComplex[pts]

Out[23]= {0.349519, 0.506776, 0.153516}

In[24]:= distComplex[pts] == Map[Norm, pts]

Out[24]= True
```

3. Here is the computation for the iteration function $c \sin(z)$ using $c = 1 + 0.4\,i$.

```
In[25]:= cJulia2 = Compile[{{z, _Complex}, {c, _Complex}}, Module[{cnt = 1},
            FixedPoint[(cnt++; c Sin[#]) &,
             z, 100, SameTest → (Abs[Im[#2]] > 50 &)]; cnt],
          CompilationTarget → "C", RuntimeAttributes → {Listable},
          Parallelization → True, "RuntimeOptions" → "Speed"]

Out[25]= CompiledFunction[{z, c},
          Module[{cnt = 1}, FixedPoint[(cnt++; c Sin[#1]) &, z, 100,
            SameTest → (Abs[Im[#2]] > 50 &)]; cnt], -CompiledCode-]
```

```
In[26]:= With[{res = 100}, ArrayPlot[
           ParallelTable[-cJulia2[x + y I, 1 + 0.4 I], {y, -2 π, 2 π, 1/res},
            {x, -2 π, 2 π, 1/res}], ColorFunction → ColorData["CMYKColors"]]]
```

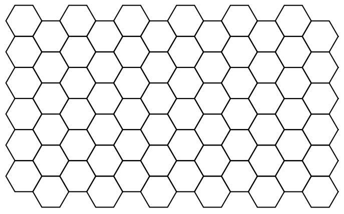
Out[26]=



13
Applications and packages

13.1 *Random walk application*

1. Although all the lattice walks in this chapter were done on the square lattice, we could also implement the walks on lattices with different geometries. For example, the hexagonal lattice in two dimensions can be used as the grid on which our random walkers move.

   Create a two-dimensional random walk that can move in one of six directions each separated by 60 degrees.
2. Generate random walks where the step length t occurs with a probability proportional to $1/t^2$. These walks are sometimes referred to as Lévy flights.
3. Create a version of `ShowWalk` that uses `GraphicsComplex` directly. The first argument to `GraphicsComplex` is the coordinate information as given by `RandomWalk`; the second argument should be graphics primitives (`Line`, `Point`) that indicate how the coordinates should be displayed.
4. Create a visualization of random walks that takes advantage of the efficiency of `Graph` to store and represent large amounts of graphical data. The first argument to `Graph` can be a list of rules that represents the connectivity information. For example, `2` $\leftrightarrow$ `3` indicates that the second vertex is connected to the third vertex with a directed edge. Use the option `VertexCoordinates` to pass the explicit coordinate information from `RandomWalk` to `Graph`. Run some tests to determine the efficiency (in terms of running time and memory) of this approach as compared to the `ShowWalk` function that was developed in this chapter.
5. Create a function that generates random walks with random step length. Advanced: Do the same but allow for arbitrary distributions.
6. Create a random walk that is bounded by a region in the Cartesian plane, for example, a circle centered at the origin of radius 2.
7. Create a one-dimensional random walk over the digits of π – if the digit is even, take a step to the right; if the digit is odd, take a step to the left.

13.1 Solutions

3. Here is the usage message for `GraphicsComplex`,

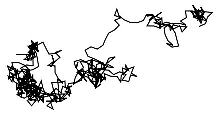
```
In[1]:= ? GraphicsComplex
```

GraphicsComplex[{pt_1, pt_2, …}, *data*] represents a graphics complex in which coordinates given as integers *i* in graphics primitives in *data* are taken to be pt_i. ≫

The first argument to `GraphicsComplex` is a list of coordinate points, such as the output from `RandomWalk`. The second argument is a set of graphics primitives indexed by the positions of the points in the list of coordinates. Here are two examples, one in two dimensions and the other in three.

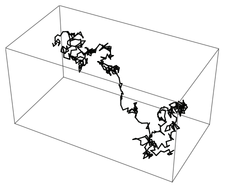
```
In[2]:= Needs["PWM`RandomWalks`"]

In[3]:= Graphics[GraphicsComplex[
          RandomWalk[500, LatticeWalk → False], Line[Range[500]]]]
```

Out[3]=



```
In[4]:= Graphics3D[GraphicsComplex[RandomWalk[500,
            Dimension → 3, LatticeWalk → False], Line[Range[500]]]]
```

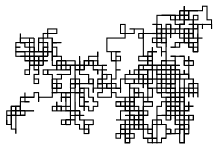
Out[4]=



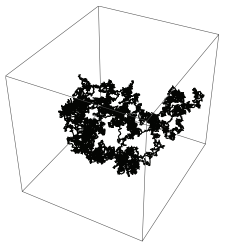
We can quickly modify the code for `ShowWalk` developed in the chapter to use `GraphicsComplex` instead.

```
In[5]:= ShowWalkGC[walk_] :=
         Module[{dim = Dimensions[walk], ran = Range[Length[walk]]},
          If[Length[dim] == 1 || dim[[2]] == 2,
           Graphics[GraphicsComplex[walk, Line[ran]]],
           Graphics3D[GraphicsComplex[walk, Line[ran]]]]]

In[6]:= ShowWalkGC[RandomWalk[2500]]
```

Out[6]=



```
In[7]:= ShowWalkGC[RandomWalk[2500, Dimension → 3, LatticeWalk → False]]
```

Out[7]=


Here are some comparisons of running times for this approach and the `ShowWalk` function developed in the chapter.

```
In[8]:= rw = RandomWalk[1 000 000, Dimension → 3, LatticeWalk → False];

In[9]:= Timing[gc = ShowWalkGC[rw];]

Out[9]= {0.003836, Null}

In[10]:= Timing[sw = ShowWalk[rw];]

Out[10]= {0.12881, Null}
```

4. Start by creating a list of rules that indicate the first point is connected to the second, the second point is connected to the third, and so on. If you have ten points, partition them as follows.

```
In[11]:= Partition[Range[10], 2, 1]

Out[11]= {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {6, 7}, {7, 8}, {8, 9}, {9, 10}}
```

The graph rules are created by applying `DirectedEdge` at level 1.

```
In[12]:= Apply[DirectedEdge, %, {1}]

Out[12]= {1 ↦ 2, 2 ↦ 3, 3 ↦ 4, 4 ↦ 5, 5 ↦ 6, 6 ↦ 7, 7 ↦ 8, 8 ↦ 9, 9 ↦ 10}
```

Here is a little function that puts these pieces together.

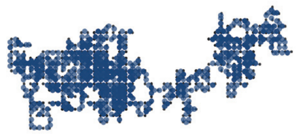
```
In[13]:= Clear[bonds];
         bonds[n_] := Apply[DirectedEdge, Partition[Range[n], 2, 1], {1}]
```

The bond information is the first argument to `Graph`; the coordinates given by `RandomWalk` are the value of the option `VertexCoordinates`.

```
In[15]:= << PWM`RandomWalks`
```

```
In[16]:= With[{steps = 1500}, Graph[bonds[steps],
           VertexCoordinates → RandomWalk[steps, LatticeWalk → True]]]
```

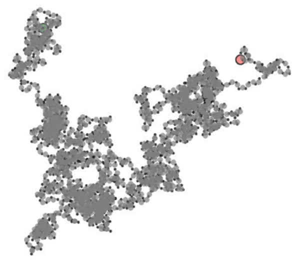
Out[16]=



One of the advantages of representing these random walks as graphs is that you have all the graph formatting and styling functions available to quickly modify your graph.

```
In[17]:= With[{steps = 1500}, Graph[bonds[steps],
           DirectedEdges → False, EdgeStyle → Gray, VertexSize →
            {1 → {"Scaled", .025}, steps → {"Scaled", .025}}, VertexStyle →
            {1 → {Opacity[0.4], Green}, steps → {Opacity[0.4], Red}},
           VertexCoordinates → RandomWalk[steps, LatticeWalk → False]]]
```

Out[17]=



The disadvantage of this approach is that it is limited to two-dimensional walks. `Graph` does not support three-dimensional objects and it does not make much sense in one dimension.

13.4 *Creating packages*

1. The following set of exercises will walk you through the creation of a package `Collatz`, a package of functions for performing various operations related to the Collatz problem that we investigated earlier (Exercises 3 and 4 of Section 4.1, Exercise 6 of Section 6.2, and Exercise 4 of Section 7.3). Recall that the Collatz function, for any integer n, returns $3n + 1$ for odd n, and $n/2$ for even n. The (as yet unproven) Collatz Conjecture is the statement that, for any initial positive integer n, the iterates of the Collatz function always reach the cycle 4, 2, 1,…. Start by creating an auxiliary function `collatz[n]` that returns $3n + 1$ for n odd and $n/2$ for n even.

   a. Create the function `CollatzSequence[n]` that lists the iterates of the auxiliary function `collatz[n]`. Here is some sample output of the `CollatzSequence` function.

```
In[1]:= CollatzSequence[7]

Out[1]= {7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

b. Create a usage message for `CollatzSequence` and warning messages for each of the following situations:

`notint`: the argument to `CollatzSequence` is not a positive integer

`argx`: `CollatzSequence` was called with the wrong number of arguments

c. Modify the definition of `CollatzSequence` that you created in part a. above so that it does some error trapping and issues the appropriate warning message that you created in part b.

d. Finally, put all the pieces together and write a package ``Collatz` `` that includes the appropriate `BeginPackage` and `Begin` statements, usage messages, warning messages, and function definitions. Make `CollatzSequence` a public function and `collatz` a private function. Put your package in a directory where *Mathematica* can find it on its search path and then test it to see that it returns correct output such as in the examples below.

```
In[11]:= Quit[];

In[1]:= << PwM`Collatz`

In[2]:= ? CollatzSequence
```

CollatzSequence[*n*] computes the sequence of Collatz iterates starting with initial value *n*. The sequence terminates as soon as it reaches the value 1.

Here are various cases in which `CollatzSequence` is given bad input.

```
In[3]:= CollatzSequence[-5]

        CollatzSequence::notint : First argument, -5, to CollatzSequence must be a positive integer.

In[4]:= CollatzSequence[4, 6]

        CollatzSequence::argx : CollatzSequence called with 2 arguments; 1 argument is expected. >>

Out[4]= CollatzSequence[4, 6]
```

And this computes the sequence for starting value 27.

```
In[5]:= CollatzSequence[27]

Out[5]= {27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484,
         242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466,
         233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890,
         445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283,
         850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079,
         3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367,
         4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732,
         866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46,
         23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

2. The square end-to-end distance of a two-dimensional walk is defined as $(x_f - x_i)^2 + (y_f - y_i)^2$, where $\{x_i, y_i\}$ and $\{x_f, y_f\}$ are the initial and final locations of the walk, respectively. Assuming the initial

point is the origin, then this simplifies to $x_f^2 + y_f^2$. Write a function `SquareDistance` that takes a two-dimensional walk as an argument and computes the square end-to-end distance. Write a usage message and include this function as a publicly exported function in the RandomWalks package.

13.4 *Solutions*

1. Here are the definitions for the auxiliary `collatz` function.

```
In[1]:= collatz[n_?EvenQ] := n / 2
In[2]:= collatz[n_?OddQ] := 3 n + 1
```

a. This is essentially the definition given in the solution to Exercise 5 from Section 6.2.

```
In[3]:= CollatzSequence[n_] := NestWhileList[collatz, n, # ≠ 1 &]

In[4]:= CollatzSequence[7]

Out[4]= {7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

b. First we write the usage message for `CollatzSequence`, our public function. Notice that we write no usage message for the private `collatz` function.

```
In[5]:= CollatzSequence::usage =
          "CollatzSequence[n] computes the sequence of Collatz
            iterates starting with initial value n. The sequence
            terminates as soon as it reaches the value 1.";
```

Here is the warning message that will be issued whenever `CollatzSequence` is passed an argument that is not a positive integer.

```
In[6]:= CollatzSequence::notint =
          "First argument, `1`, to CollatzSequence
            must be a positive integer.";
```

c. Here is the modified definition which now issues the warning message created above whenever the argument *n* is not a positive integer.

```
In[7]:= CollatzSequence[n_] :=
         If[IntegerQ[n] && n ≥ 0, NestWhileList[collatz, n, # ≠ 1 &],
          Message[CollatzSequence::notint, n]]
```

The following case covers the situation when `CollatzSequence` is passed two or more arguments. Note that it uses the built-in `argx` message, which is issued whenever built-in functions are passed the wrong number of arguments.

```
In[8]:= CollatzSequence[_, args__] /; Message[CollatzSequence::argx,
           CollatzSequence, Length[{args}] + 1] := Null
```

d. The package begins by giving *usage messages* for every exported function. The functions to be exported are *mentioned* here – *before* the subcontext `Private`` is entered – so that the symbol `CollatzSequence` has context `Collatz``. Notice that `collatz` is *not* mentioned here and hence will not be accessible to the user of this package.

```
In[9]:= Quit[]
```

```
In[1]:= BeginPackage["PwM`Collatz`"];
In[2]:= CollatzSequence::usage =
          "CollatzSequence[n] computes the sequence of Collatz
            iterates starting with initial value n. The sequence
            terminates as soon as it reaches the value 1.";
In[3]:= CollatzSequence::notint =
          "First argument, `1`, to CollatzSequence
            must be a positive integer.";
```

A new context PwM`Collatz`Private` is then begun *within* PwM`Collatz. All the definitions of this package are given within this new context. The context PwM`Collatz`CollatzSequence is defined within the System` context. The context of collatz, on the other hand, is PwM`Collatz`Private`.

```
In[4]:= Begin["`Private`"];
In[5]:= collatz[n_?EvenQ] := n / 2
In[6]:= collatz[n_?OddQ] := 3 n + 1
In[7]:= CollatzSequence[n_] :=
         If[IntegerQ[n] && n ≥ 0, NestWhileList[collatz, n, # ≠ 1 &],
          Message[CollatzSequence::notint, n]]
In[8]:= CollatzSequence[_, args__] /; Message[CollatzSequence::argx,
           CollatzSequence, Length[{args}] + 1] := Null
In[9]:= End[];
In[10]:= EndPackage[]
```

After the End[] and EndPackage[] functions are evaluated, $Context and $ContextPath revert to whatever they were before, except that PwM`Collatz` is added to $ContextPath. Users can refer to CollatzSequence using its short name, but they can only refer to the auxiliary function collatz by its full name. The intent is to discourage clients from using collatz at all, and doing so should definitely be avoided, since the author of the package may change or remove auxiliary definitions at a later time.