Solutions to exercises

2 The *Mathematica* language

2.1 Expressions

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]]

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

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

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

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

In[3]:= ((x^2 + y) z / w) [[2, 1, 2]]

Out[3]= 2
```

4. There are three terms in the expression, with the term `b x` being the second.

```
In[4]:= expr = a x^2 + b x + c;

In[5]:= FullForm[expr]

Out[5]//FullForm=
Plus[c, Times[b, x], Times[a, Power[x, 2]]]
```

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

```
In[6]:= expr[[2]]

Out[6]= b x

In[7]:= expr[[2, 1]]

Out[7]= b
```

2.2 Definitions

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_] := Table[Random[], {n}]

In[2]:= ? randLis1

        Global`randLis1

        randLis1[n_] := Table[Random[], {n}]

In[3]:= randLis1[3]

Out[3]= {0.0405431, 0.043554, 0.699358}
```

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[4]:= randLis2[n_] := (x = Random[]; Table[x, {n}])

In[5]:= ? randLis2

        Global`randLis2

        randLis2[n_] := (x = Random[]; Table[x, {n}])

In[6]:= randLis2[3]

Out[6]= {0.651026, 0.651026, 0.651026}
```

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

```
In[7]:= randLis3[n_] := (x := Random[]; Table[x, {n}])

In[8]:= ? randLis3

        Global`randLis3

        randLis3[n_] := (x := Random[]; Table[x, {n}])

In[9]:= randLis3[3]

Out[9]= {0.304574, 0.184163, 0.744351}
```

d. Recall that 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[10]:= randLis4[n_] = Table[Random[], {n}]
         Table::iterb : Iterator {n} does not have appropriate bounds. More…
Out[10]= Table[Random[], {n}]

In[11]:= ? randLis4
         Global`randLis4
         randLis4[n_] = Table[Random[], {n}]
```

2.3 Predicates and Boolean operations

1. There are several ways to define this function, using either 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. A number *n* can be considered a natural number if it is an integer and greater than or equal to zero.

```
In[5]:= Positive[0]
Out[5]= False

In[6]:= NaturalQ[n_] := IntegerQ[n] && n ≥ 0

In[7]:= NaturalQ[0]
Out[7]= True

In[8]:= NaturalQ[-4]
Out[8]= False
```

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

```
In[9]:= SubsetQ[{}, lis2_] := True
```

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

```
In[10]:= SubsetQ[lis1_, lis2_] := Intersection[lis1, lis2] == lis1

In[11]:= A = {a, b, c};
         B = {a, b, c, d, e};

In[13]:= SubsetQ[A, B]

Out[13]= 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[14]:= lis1_ ⊂ lis2_ := Intersection[lis1, lis2] == lis1

In[15]:= A ⊂ B

Out[15]= True
```

3 Lists

3.2 Creating and measuring lists

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

```
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. This is probably the simplest way to generate random −1s, 0s, and 1s.

```
In[3]:= Table[Random[Integer, {-1, 1}], {10}]

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

3. Here are three ways to generate the list.

```
In[4]:= Table[2 Random[Integer] - 1, {10}]

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

```
In[5]:= Table[(-1)^Random[Integer], {10}]

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

The following solution will become clearer in the next section after we have discussed the `Part` function in some detail.

```
In[6]:= {1, -1}[[Table[Random[Integer, {1, 2}], {10}]]]

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

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

```
In[7]:= Table[f[i], {i, 5}]

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

```
In[8]:= Table[f[i, j], {i, 3}, {j, 4}]

Out[8]= {{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. From the top level, there are two lists, each consisting of two sublists, each sublist consisting of two elements.

```
In[9]:= Dimensions[{{{1, a}, {4, d}}, {{2, b}, {3, c}}}]

Out[9]= {2, 2, 2}
```

3.3 Manipulating lists

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

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

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

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

```
In[2]:= Transpose[{{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}, {x5, y5}}]

Out[2]= {{x1, x2, x3, x4, x5}, {y1, y2, y3, y4, y5}}
```

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

```
In[3]:= NSEW = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
```

```
In[4]:= Table[NSEW[[Random[Integer, {1, 4}]]], {10}]

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

4. We first drop 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[5]:= Rest[{a, b, c, d, e, f, g}]

Out[5]= {b, c, d, e, f, g}

In[6]:= Partition[%, 1, 2]

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

In[7]:= Flatten[%]

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

5.

```
In[8]:= {a, b, c, d}[[{3, 2, 4, 1}]]

Out[8]= {c, b, d, a}
```

6.

```
In[9]:= Transpose[{{3, 2, 4, 1}, {a, b, c, d}}]

Out[9]= {{3, a}, {2, b}, {4, c}, {1, d}}

In[10]:= Sort[%]

Out[10]= {{1, d}, {2, b}, {3, a}, {4, c}}

In[11]:= Transpose[%]

Out[11]= {{1, 2, 3, 4}, {d, b, a, c}}

In[12]:= %[[2]]

Out[12]= {d, b, a, c}
```

3.4 Working with several lists

1. `Join` expects lists as arguments.

```
In[1]:= Join[{z}, {x, y}]

Out[1]= {z, x, y}
```

2. The trick here is partitioning the joined list so that you get every other element.

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

In[3]:= Rest[expr]
Out[3]= {2, 3, 4, a, b, c, d}

In[4]:= Partition[%, 1, 2]
Out[4]= {{2}, {4}, {b}, {d}}

In[5]:= Flatten[%]
Out[5]= {2, 4, b, d}
```

This can also be done using the `Take` function.

```
In[6]:= Take[expr, {2, Length[expr], 2}]
Out[6]= {2, 4, b, d}
```

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

```
In[7]:= A = {a, b, c, d};
        B = {a, b, e, f};

In[9]:= Complement[A ⋃ B, A ⋂ B]
Out[9]= {c, d, e, f}

In[10]:= Complement[Union[A, B], Intersection[A, B]]
Out[10]= {c, d, e, f}
```

3.5 Strings and characters

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
```

2. We first need to extract the character codes from this string.

```
In[8]:= numstr = "73"
Out[8]= 73

In[9]:= ToCharacterCode[numstr][[1]]
Out[9]= 55

In[10]:= 10 (% - 48)
Out[10]= 70

In[11]:= ToCharacterCode[numstr][[2]]
Out[11]= 51

In[12]:= % - 48
Out[12]= 3
```

```
In[13]:= % + %%%

Out[13]= 73
```

Here it is all put together in one line.

```
In[14]:= 10 (ToCharacterCode[numstr][[1]] - 48) + (ToCharacterCode[numstr][[2]] - 48)

Out[14]= 73
```

There is a built-in function for this task, `ToExpression`. See the next exercise for details.

3.

```
In[15]:= numb = ToCharacterCode["73"]

Out[15]= {55, 51}

In[16]:= numb - 48

Out[16]= {7, 3}

In[17]:= 8 Part[%, 1] + Part[%, 2]

Out[17]= 59
```

Here is another approach that converts the single characters into regular expressions and then operates on those directly.

```
In[18]:= ToExpression[Characters["73"]]

Out[18]= {7, 3}

In[19]:= 8 First[%] + Last[%]

Out[19]= 59
```

4. One approach converts the string to character codes.

```
In[20]:= ToCharacterCode["10495"]

Out[20]= {49, 48, 52, 57, 53}

In[21]:= % - 48

Out[21]= {1, 0, 4, 9, 5}

In[22]:= Reverse[Table[10^j, {j, 0, 4}]]

Out[22]= {10000, 1000, 100, 10, 1}
```

```
In[23]:= %.%%
Out[23]= 10495
```

A direct approach uses `ToExpression`.

```
In[24]:= ToExpression["10495"]
Out[24]= 10495
```

5. First, consider the character code of a string.

```
In[25]:= ToCharacterCode["best"]
Out[25]= {98, 101, 115, 116}
```

Then we need only know if this list of codes is in order.

```
In[26]:= OrderedQ[%]
Out[26]= True
```

So here is our Boolean function `OrderedWordQ`.

```
In[27]:= OrderedWordQ[w_String] := OrderedQ[ToCharacterCode[w]]
```

Now we will find all the palindromic words in the dictionary file that comes with *Mathematica*. First we generate a platform-independent path to the dictionary file.

```
In[28]:= wordfile = ToFileName[{$TopDirectory, "Documentation",
            "English", "Demos", "DataFiles"}, "dictionary.dat"]
Out[28]= C:\Program Files\Wolfram Research\Mathematica\5
           .1\Documentation\English\Demos\DataFiles\dictionary.dat
```

Then we read the file using `ReadList`, specifying the type of data we are reading in as a `Word`.

```
In[29]:= words = ReadList[wordfile, Word];
```

Finally, we select those elements from the list `words` that pass the `OrderedWordQ` test.

```
In[30]:= Select[words, OrderedWordQ] // Shallow
Out[30]//Shallow=
    {a, AAA, AAAS, abbe, abbey, abbot, Abbott, abc, Abe, Abel, <<565>>}
```

6. Here is the function that checks if a string is a palindrome.

```
In[31]:= PalindromeQ[str_String] := StringReverse[str] == str
```

```
In[32]:= PalindromeQ["mood"]

Out[32]= False

In[33]:= PalindromeQ["PoP"]

Out[33]= True

In[34]:= PalindromeQ[num_Integer] := PalindromeQ[ToString[num]]

In[35]:= PalindromeQ[12522521]

Out[35]= True
```

Create a path to the file `dictionary.dat`.

```
In[36]:= dictfile = ToFileName[{$BaseDirectory,
            "Applications", "IPM3", "DataFiles"}, "dictionary.dat"]

Out[36]= C:\Documents and Settings\All Users\Application Data\
           Mathematica\Applications\IPM3\DataFiles\dictionary.dat
```

Import the file.

```
In[37]:= words = Import[dictfile, "Words"];

In[38]:= Select[words, PalindromeQ]

Out[38]= {a, AAA, ABA, ala, AMA, ana, b, bib, bob, bub, c, CDC, civic, d, dad, deed,
          did, DOD, dud, e, eke, ere, eve, ewe, eye, f, g, gag, gig, gog, h, huh, i,
          ii, iii, j, k, l, level, m, madam, minim, mum, n, non, noon, nun, o, p, pap,
          PDP, peep, pep, pip, poop, pop, pup, q, r, radar, refer, rever, rotor, s,
          sis, s's, t, tat, teet, tenet, tit, TNT, toot, tot, u, v, w, wow, x, y, z}
```

4 Functional programming

4.2 Functions for manipulating expressions

1. Here is a sample set of pairs of numbers.

```
In[1]:= data = {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}};
```

The `pairSum` function can be written simply as:

```
In[2]:= addPair[{x_, y_}] := x + y
```

Finally we map `pairSum` across data.

```
In[3]:= Map[addPair, data]
Out[3]= {3, 5, 7, 9, 11}
```

2. Here is a sample set of pairs of numbers.

```
In[4]:= data = {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}};
```

Since `Apply` normally works at level 0, we need to give it a third argument to get it to apply `Plus` at level 1.

```
In[5]:= Apply[Plus, data, {1}]
Out[5]= {3, 5, 7, 9, 11}
```

3. First you need to transpose the matrix and then reverse the pairs.

```
In[6]:= lis = {{1, 2, 3}, {4, 5, 6}}
Out[6]= {{1, 2, 3}, {4, 5, 6}}

In[7]:= Transpose[lis]
Out[7]= {{1, 4}, {2, 5}, {3, 6}}

In[8]:= Map[Reverse, %]
Out[8]= {{4, 1}, {5, 2}, {6, 3}}
```

This can also be accomplished using `Thread`.

```
In[9]:= Map[Reverse, Thread[lis]]
Out[9]= {{4, 1}, {5, 2}, {6, 3}}
```

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

```
In[10]:= Transpose[{{1, 2}, {3, 4}}] {x, y}
Out[10]= {{x, 3 x}, {2 y, 4 y}}

In[11]:= Apply[Plus, %]
Out[11]= {x + 2 y, 3 x + 4 y}

In[12]:= Inner[Times, {{1, 2}, {3, 4}}, {x, y}, Plus]
Out[12]= {x + 2 y, 3 x + 4 y}
```

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

```
In[13]:= facs = FactorInteger[3628800]
Out[13]= {{2, 8}, {3, 4}, {5, 2}, {7, 1}}

In[14]:= Apply[Power, facs, 2]
Out[14]= {256, 81, 25, 7}
```

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

```
In[15]:= Apply[Times, %]
Out[15]= 3628800
```

Here is a function that puts this all together.

```
In[16]:= ExpandFactors[lis_] := Apply[Times, Apply[Power, lis, 2]]

In[17]:= FactorInteger[29523279903960414084761860964352000000]
Out[17]= {{2, 32}, {3, 15}, {5, 7}, {7, 4}, {11, 3},
          {13, 2}, {17, 2}, {19, 1}, {23, 1}, {29, 1}, {31, 1}}

In[18]:= ExpandFactors[%]
Out[18]= 29523279903960414084761860964352000000
```

Another approach would be to use `Transpose` to separate the bases from their exponents, then use `MapThread` to raise each base to the corresponding exponent.

```
In[19]:= Transpose[facs]
Out[19]= {{2, 3, 5, 7}, {8, 4, 2, 1}}

In[20]:= MapThread[Power, %]
Out[20]= {256, 81, 25, 7}
```

Finally, apply `Times` to the list.

```
In[21]:= Apply[Times, %]
Out[21]= 3628800

In[22]:= ExpandFactors2[lis_] := Apply[Times, MapThread[Power, Transpose[lis]]]
```

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

```
In[23]:= facs = FactorInteger[10!]
Out[23]= {{2, 8}, {3, 4}, {5, 2}, {7, 1}}
```

First we extract the prime bases and their exponents.

```
In[24]:= bases = Transpose[facs][[1]]
Out[24]= {2, 3, 5, 7}

In[25]:= exponents = Transpose[facs][[2]]
Out[25]= {8, 4, 2, 1}
```

Here then is the inner product, threading `Power` over the lists and then multiplying the resulting terms with `Times`.

```
In[26]:= Inner[Power, bases, exponents, Times]
Out[26]= 3628800
```

Here is a function that combines these steps.

```
In[27]:= ExpandFactors3[lis_] := Module[{facs = Transpose[lis]},
           Inner[Power, facs[[1]], facs[[2]], Times]]

In[28]:= ExpandFactors3[facs]
Out[28]= 3628800
```

7. 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[29]:= Inner[d, {e1, e2, e3}, {v1, v2, v3}, Plus]
Out[29]= d[e1, v1] + d[e2, v2] + d[e3, v3]
```

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

```
In[30]:= 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 0.

```
In[31]:= div[{x, y, z} / (x^2 + y^2 + z^2)^(3/2), {x, y, z}]
Out[31]= -(3 x^2)/(x^2 + y^2 + z^2)^(5/2) - (3 y^2)/(x^2 + y^2 + z^2)^(5/2) - (3 z^2)/(x^2 + y^2 + z^2)^(5/2) + 3/(x^2 + y^2 + z^2)^(3/2)

In[32]:= Simplify[%]
Out[32]= 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. The

reader should refer to Chapter 6 for a discussion of how to use pattern matching to deal with this issue.

4.3 Iterating functions

1. First we generate the step directions.

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

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

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

```
In[2]:= FoldList[Plus, 0, %]

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

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

```
In[3]:= {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}[[Table[Random[Integer, {1, 4}], {10}]]]

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

In[4]:= FoldList[Plus, {0, 0}, %]

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

3. Starting with 1, we want to fold the `Times` functions across the first *n* integers.

```
In[5]:= fac[n_] := Fold[Times, 1, Range[n]]

In[6]:= fac[10]

Out[6]= 3628800
```

4.4 Programs as functions

1. The obvious way to do this is to take the list and simply pick out elements at random locations. *Note*: the right-most location in the list is given by `Length[`*lis*`]`, using the built-in `Part` and `Random` functions.

```
In[1]:= chooseWithReplacement[lis_, n_] :=
         lis[[Table[Random[Integer, {1, Length[lis]}], {n}]]]

In[2]:= chooseWithReplacement[{a, b, c, d, e, f, g, h}, 3]

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

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

```
In[3]:= stringInsert[str1_, str2_, pos_] :=
         FromCharacterCode[Join[Take[ToCharacterCode[str1], pos - 1],
           ToCharacterCode[str2], Drop[ToCharacterCode[str1], pos - 1]]]

In[4]:= stringInsert["Joy world", "to the ", 5]

Out[4]= Joy to the world

In[5]:= stringDrop[str_, pos_] :=
         FromCharacterCode[Drop[ToCharacterCode[str], pos]]
```

3. There are many ways of defining this function. Here we take advantage of the fact that if `p` and `q` are each lists of two numbers, then `p-q` will subtract element-wise.

```
In[6]:= distance[pt1_, pt2_] := √Apply[Plus, (pt1 - pt2)^2]

In[7]:= distance[{2, 5}, {6, 8}]

Out[7]= 5
```

4. 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[8]:= interLeave2[lis1_, lis2_] :=
         Flatten[Join[Transpose[{lis2, Take[lis1, Length[lis2]]}],
           Take[lis1, Length[lis2] - Length[lis1]]]]

In[9]:= interLeave2[{a, b, c, d}, {1, 2, 3}]

Out[9]= {1, a, 2, b, 3, c, d}
```

5. After creating the card deck, we cut it in half and interleave the two halves.

```
In[10]:= cardDeck =
          Flatten[Outer[List, {♣, ♢, ♡, ♠}, Join[Range[2, 10], {J, Q, K, A}]], 1];

In[11]:= Flatten[Transpose[Partition[cardDeck, 26]], 1]

Out[11]= {{♣, 2}, {♡, 2}, {♣, 3}, {♡, 3}, {♣, 4}, {♡, 4}, {♣, 5}, {♡, 5}, {♣, 6},
          {♡, 6}, {♣, 7}, {♡, 7}, {♣, 8}, {♡, 8}, {♣, 9}, {♡, 9}, {♣, 10},
          {♡, 10}, {♣, J}, {♡, J}, {♣, Q}, {♡, Q}, {♣, K}, {♡, K}, {♣, A}, {♡, A},
          {♢, 2}, {♠, 2}, {♢, 3}, {♠, 3}, {♢, 4}, {♠, 4}, {♢, 5}, {♠, 5}, {♢, 6},
          {♠, 6}, {♢, 7}, {♠, 7}, {♢, 8}, {♠, 8}, {♢, 9}, {♠, 9}, {♢, 10},
          {♠, 10}, {♢, J}, {♠, J}, {♢, Q}, {♠, Q}, {♢, K}, {♠, K}, {♢, A}, {♠, A}}
```

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

```
In[12]:= FromCharacterCode[
          Join[ToCharacterCode["To be, "], ToCharacterCode["or not to be"]]]

Out[12]= To be, or not to be
```

And here is a how we might implement a `StringReverse`.

```
In[13]:= FromCharacterCode[Reverse[ToCharacterCode[%]]]

Out[13]= eb ot ton ro ,eb oT
```

4.5 Auxiliary functions

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

```
In[1]:= latticeWalk1[n_] := Module[{steps},
          steps[m_] := {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}[[
            Table[Random[Integer, {1, 4}], {m}]]; FoldList[Plus, {0, 0}, steps[n]]]

In[2]:= latticeWalk1[10]

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

Here we use two auxiliary functions, making the code a bit easier to read.

```
In[3]:= latticeWalk2[n_] :=
         Module[{choices, steps}, choices = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
          steps[m_] := choices[[Table[Random[Integer, {1, 4}], {m}]]];
          FoldList[Plus, {0, 0}, steps[n]]]

In[4]:= latticeWalk2[10]

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

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[5]:= PerfectSearch[n_, m_] := Module[{perfectQ},
          perfectQ[j_] := Apply[Plus, Divisors[j]] == 2 j;
          Select[Range[n, m, 2], perfectQ]]

In[6]:= PerfectSearch[2, 1000]

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

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 numbers below at least 10^{300}, none is reported.

```
In[7]:= PerfectSearch[1, 1000]

Out[7]= {}
```

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[8]:= Clear[PerfectSearch]

In[9]:= PerfectSearch[n_?EvenQ, m_] := Module[{perfectQ},
          perfectQ[j_] := Apply[Plus, Divisors[j]] == 2 j;
          Select[Range[n, m, 2], perfectQ]]
```

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

```
In[10]:= PerfectSearch[2, 1000]

Out[10]= {6, 28, 496}

In[11]:= PerfectSearch[1, 1000]

Out[11]= PerfectSearch[1, 1000]
```

3. This only requires a slight change to the code from the `PerfectSearch` function from the previous exercise.

```
In[12]:= PerfectSearch[n_, m_, 3] := Module[{perfectQ},
           perfectQ[j_] := Apply[Plus, Divisors[j]] == 3 j;
           Select[Range[n, m], perfectQ]]
```

It appears as if there are only three 3-perfect numbers below 10^6.

```
In[13]:= PerfectSearch[1, 10^6, 3]

Out[13]= {120, 672, 523776}
```

4. Again, this function only requires a slight modification from that for the `PerfectSearch` function above.

```
In[14]:= PerfectSearch[n_, m_, 4] := Module[{perfectQ},
           perfectQ[j_] := Apply[Plus, Divisors[j]] == 4 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.

```
In[15]:= PerfectSearch[1, 2200000, 4] // Timing

Out[15]= {54.769 Second, {30240, 32760, 2178540}}
```

5. This function requires a third argument.

```
In[16]:= Clear[PerfectSearch];
         PerfectSearch[n_, m_, k_] := Module[{perfectQ},
           perfectQ[j_] := Apply[Plus, Divisors[j]] == k j;
           Select[Range[n, m], perfectQ]]

In[18]:= PerfectSearch[1, 100, 2]

Out[18]= {6, 28}
```

6. This function will require two auxiliary functions, the function σ and a predicate to determine whether a number is super-perfect.

```
In[19]:= SuperPerfectSearch[a_, b_] := Module[{sigma, superQ},
           sigma[n_] := Apply[Plus, Divisors[n]];
           superQ[n_] := Nest[sigma, n, 2] == 2 n;
           Select[Range[a, b], superQ]]
```

Here, then, are all super-perfect numbers less than 100,000.

```
In[20]:= SuperPerfectSearch[1, 100000]

Out[20]= {2, 4, 16, 64, 4096, 65536}
```

7. Many implementations are possible for `convertToDate`. We show here a version that uses string manipulation. First we extract the digits from the 8-digit number.

```
In[21]:= d = IntegerDigits[20030515]

Out[21]= {2, 0, 0, 3, 0, 5, 1, 5}
```

The first four digits give us the year.

```
In[22]:= d[[Range[4]]]

Out[22]= {2, 0, 0, 3}
```

Here is a function that takes a list of digits, converts them to strings, concatenates them into one string, and then converts that into a number.

```
In[23]:= convert[str_] := ToExpression[StringJoin[Map[ToString, str]]]

In[24]:= convert[d[[Range[4]]]]

Out[24]= 2003
```

```
In[25]:= Head[%]

Out[25]= Integer
```

Using `convert`, here are the auxiliary functions to extract the year, month, and day as numbers.

```
In[26]:= year[str_] := convert[str[[Range[4]]]]

In[27]:= year[d]

Out[27]= 2003

In[28]:= month[str_] := convert[str[[{5, 6}]]]

In[29]:= month[d]

Out[29]= 5

In[30]:= day[str_] := convert[str[[{7, 8}]]]

In[31]:= day[d]

Out[31]= 15
```

And here are all the pieces put together in the function `convertToDate`.

```
In[32]:= convertToDate[n_] := Module[{d, convert, year, month, day},
           d = IntegerDigits[n];
           convert[st_] := ToExpression[StringJoin[Map[ToString, st]]];
           year[st_] := convert[st[[Range[4]]]];
           month[st_] := convert[st[[{5, 6}]]];
           day[st_] := convert[st[[{7, 8}]]];
           {year[d], month[d], day[d]}]

In[33]:= convertToDate[20030515]

Out[33]= {2003, 5, 15}
```

4.6 Pure functions

1. This function adds the squares of the elements in `lis`.

```
In[1]:= elementsSquared[lis_] := Apply[Plus, lis^2]

In[2]:= elementsSquared[{3, 29, 2, 17}]

Out[2]= 1143
```

Using a pure function, this becomes:

```
In[3]:= Function[lis, Apply[Plus, lis^2]][{3, 29, 2, 17}]
Out[3]= 1143
```

or simply,

```
In[4]:= Apply[Plus, #^2] &[{3, 29, 2, 17}]
Out[4]= 1143
```

2. Here is the function that sums the digits of any integer.

```
In[5]:= sumdigits[x_Integer] := Apply[Plus, IntegerDigits[x]]
In[6]:= sumdigits[629]
Out[6]= 17
```

Using a pure function, this becomes:

```
In[7]:= Function[x, Apply[Plus, IntegerDigits[x]]][629]
Out[7]= 17
In[8]:= Apply[Plus, IntegerDigits[#]] &[629]
Out[8]= 17
```

3. First, here is the distance function.

```
In[9]:= distance[pt1_, pt2_] := √Apply[Plus, (pt1 - pt2)^2]
```

Here are some sample points.

```
In[10]:= points = Table[Random[], {5}, {2}]
Out[10]= {{0.408123, 0.110529}, {0.640705, 0.227085},
          {0.605818, 0.074615}, {0.868053, 0.302804}, {0.381267, 0.66605}}
```

Just as a check, this computes the distance between the first and second points in our list.

```
In[11]:= distance[points[[1]], points[[2]]]
Out[11]= 0.260153
```

Now we need the distance between every pair of points. So we first create the set of pairs.

```
In[12]:= pairs = Distribute[{points, points}, List];
```

Then we apply the `distance` function and take the `Max`.

```
In[13]:= Max[Apply[distance, pairs, {1}]]

Out[13]= 0.632628
```

This puts it all together using a pure function in place of the distance function. Since the `diameter` function operates on lists of pairs of numbers, we need to specify them in our pure function by means of `#1` and `#2`.

```
In[14]:= diameter[lis_] :=
          Max[Apply[Sqrt[Apply[Plus, (#1 - #2)^2]] &,
            Distribute[{lis, lis}, List], {1}]]

In[15]:= diameter[points]

Out[15]= 0.632628
```

As a final note, this function is not as efficient as it could be since it computes the distance from every point to itself, as well as computing both the distance from point a to point b and from point b to point a, for every pair of points a and b. In other words, for n points, we are computing n^2 distances when we only need to compute $\binom{n}{2}$ distances, highly sub-optimal. We leave the optimization of this function as an exercise to the reader.

4. Using pure functions, `removeRand` becomes:

```
In[16]:= Function[lis, Delete[lis, Random[Integer, {1, Length[lis]}]]][
          {a, b, c, d, e}]

Out[16]= {a, b, c, e}

In[17]:= Delete[#1, Random[Integer, {1, Length[#]}]] &[{a, b, c, d, e}]

Out[17]= {a, c, d, e}
```

5. Here is the `deal` function written using a pure function in place of `removeRand`.

```
In[18]:= deal[n_] := Module[{cardDeck}, cardDeck =
             Flatten[Outer[List, {♣, ♢, ♡, ♠}, Join[Range[2, 10], {J, Q, K, A}]], 1];
           Complement[cardDeck,
            Nest[Delete[#1, Random[Integer, {1, Length[#1]}]] &, cardDeck, n]]]

In[19]:= deal[5]

Out[19]= {{♣, A}, {♡, 2}, {♡, 9}, {♠, 2}, {♠, 5}}
```

6. This function is ideally written as an iteration.

```
In[20]:= RepUnit[n_] := Nest[(10 # + 1) &, 1, n - 1]
```

```
In[21]:= RepUnit[7]

Out[21]= 1111111

In[22]:= Map[RepUnit[#] &, Range[12]]

Out[22]= {1, 11, 111, 1111, 11111, 111111, 1111111, 11111111,
          111111111, 1111111111, 11111111111, 111111111111}
```

7. Notice that it is not necessary to use the `Module` function here because the only expressions on the right-hand side of the function definition are pure functions, built-in functions, and the names of the arguments of the function.

```
In[23]:= chooseWithoutReplacement[lis_, n_] := Complement[lis,
           Nest[Delete[#1, Random[Integer, {1, Length[#1]}]] &, lis, n]]

In[24]:= chooseWithoutReplacement[{a, b, c, d, e}, 4]

Out[24]= {a, c, d, e}
```

8. Using the list of the step increments in the north, south, east, and west directions, this ten-step walk starts at the origin.

```
In[25]:= NestList[#1 + {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}[[Random[Integer, {1, 4}]]] &,
          {0, 0}, 10]

Out[25]= {{0, 0}, {1, 0}, {1, 1}, {1, 0}, {0, 0},
          {-1, 0}, {-2, 0}, {-3, 0}, {-4, 0}, {-4, 1}, {-4, 2}}
```

9. Here is the path to the dictionary file.

```
In[26]:= dictfile = ToFileName[{$TopDirectory, "Documentation",
            "English", "Demos", "DataFiles"}, "dictionary.dat"]

Out[26]= C:\Program Files\Wolfram Research\Mathematica\5
           .1\Documentation\English\Demos\DataFiles\dictionary.dat
```

This reads in the file using `ReadList` specifying the type of data we are reading in as a `Word`.

```
In[27]:= words = ReadList[dictfile, Word];
```

Here are three words from the dictionary.

```
In[28]:= words[[{5, 55, 555}]]

Out[28]= {Aaron, abolish, alder}
```

First we need to create a function that takes a string as an argument and returns `True` if its first character is `char`. As a first step, here is a pure function that checks if the first character of the argument being passed to it (`"abolish"`) starts with the letter `"a"`.

```
In[29]:= (StringTake[#, 1] === "a") &["abolish"]

Out[29]= True
```

Now we can use this pure function as the test to select all those words in `lis` that pass this particular test.

```
In[30]:= WordsStartingWith[lis_, char_] :=
          Select[lis, StringTake[#, 1] === char &]
```

Finally we can check all the words in the dictionary file that start with the letter `"z"` say.

```
In[31]:= WordsStartingWith[words, "z"]

Out[31]= {z, zag, zagging, zap, zazen, zeal, zealot, zealous, zebra, zenith, zero,
          zeroes, zeroth, zest, zesty, zeta, zig, zigging, zigzag, zigzagging,
          zilch, zinc, zing, zip, zircon, zirconium, zloty, zodiac, zodiacal,
          zombie, zone, zoo, zoology, zoom, zounds, z's, zucchini, zygote}
```

This can also be accomplished using the new (in Version 5.1) `StringMatchQ` together with a wildcard character.

```
In[32]:= Select[words, StringMatchQ[#, "z*"] &]

Out[32]= {z, zag, zagging, zap, zazen, zeal, zealot, zealous, zebra, zenith, zero,
          zeroes, zeroth, zest, zesty, zeta, zig, zigging, zigzag, zigzagging,
          zilch, zinc, zing, zip, zircon, zirconium, zloty, zodiac, zodiacal,
          zombie, zone, zoo, zoology, zoom, zounds, z's, zucchini, zygote}
```

Or you can get all those words that start with either "z" or "Z" by using the `IgnoreCase` option to `StringMatchQ`.

```
In[33]:= Select[words, StringMatchQ[#, "z*", IgnoreCase → True] &]

Out[33]= {z, Zachary, zag, zagging, Zagreb, Zaire, Zambia, Zan, Zanzibar,
          zap, zazen, zeal, Zealand, zealot, zealous, zebra, Zeiss,
          Zellerbach, Zen, zenith, zero, zeroes, zeroth, zest, zesty,
          zeta, Zeus, Ziegler, zig, zigging, zigzag, zigzagging, zilch,
          Zimmerman, zinc, zing, Zion, zip, zircon, zirconium, zloty, zodiac,
          zodiacal, Zoe, Zomba, zombie, zone, zoo, zoology, zoom, Zorn,
          Zoroaster, Zoroastrian, zounds, z's, zucchini, Zurich, zygote}
```

Or, using the new (in Version 5.1) `Pick` function:

```
In[34]:= Pick[words, StringMatchQ[words, "z" ~~ __]]

Out[34]= {zag, zagging, zap, zazen, zeal, zealot, zealous, zebra, zenith, zero,
          zeroes, zeroth, zest, zesty, zeta, zig, zigging, zigzag, zigzagging,
          zilch, zinc, zing, zip, zircon, zirconium, zloty, zodiac, zodiacal,
          zombie, zone, zoo, zoology, zoom, zounds, z's, zucchini, zygote}
```

10. Several modifications to the solution to Exercise 9 are needed. First, we must choose only those words with string length greater than or equal to the string length of the second argument to `WordsStartingWith`. Secondly, from this modified list, we choose those words whose first several characters match the string we are working with.

```
In[35]:= Clear[WordsStartingWith]

In[36]:= WordsStartingWith[lis_, str_] := Module[{lis2},
           lis2 = Select[lis, StringLength[#] ≥ StringLength[str] &];
           Select[lis2, StringTake[#, StringLength[str]] === str &]]

In[37]:= WordsStartingWith[words, "zoo"]

Out[37]= {zoo, zoology, zoom}
```

Or, using `StringMatchQ` from Version 5.1, you have to join the string with the wildcard character using ~~.

```
In[38]:= Clear[WordsStartingWith]

In[39]:= WordsStartingWith[lis_List, str_String] :=
          Select[lis, StringMatchQ[#, str ~~ "*"] &]

In[40]:= WordsStartingWith[words, "zoo"]

Out[40]= {zoo, zoology, zoom}
```

Or, using the new (in Version 5.1) `Pick` function (note the need for the triple-blank here):

```
In[41]:= Pick[words, StringMatchQ[words, "zoo" ~~ ___]]

Out[41]= {zoo, zoology, zoom}
```

11. Using `Fold`, this pure function requires two arguments. The key is to start with initial value 0.

```
In[42]:= Horner[list_List, base_] := Fold[base #1 + #2 &, 0, list];

In[43]:= Horner[{a, b, c, d, e}, x]

Out[43]= e + x (d + x (c + x (b + a x)))

In[44]:= Expand[%]

Out[44]= e + d x + c x^2 + b x^3 + a x^4
```

4.7 One-liners

1. If we map the `Mod` function with base 2 over a list, it will return 1 for every odd element and 0 for every even element.

```
In[1]:= Map[(Mod[#, 2] &), {1, 1, 0, 2, 1}]

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

   Taking two lists, if we add them element-wise, we then need to select those that pass the mod test above.

```
In[2]:= l1 = {1, 0, 0, 1, 1};
        l2 = {0, 1, 0, 1, 0};

In[4]:= lis = l1 + l2

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

In[5]:= Select[lis, (Mod[#, 2] == 1 &)]

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

   And finally, we need to know how many elements are in this last list.

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

Out[6]= 3

In[7]:= HammingDistance3[lis1_, lis2_] :=
         Length[Select[lis1 + lis2, (Mod[#, 2] == 1 &)]]
```

   Actually this could have been done more cleanly by using the predicate `OddQ`.

```
In[8]:= HammingDistance4[lis1_, lis2_] :=
         Length[Select[lis1 + lis2, OddQ]]
```

2. Using `Total`, which simply gives the sum of the elements in a list, Hamming distance can be computed as follows:

```
In[9]:= HammingDistance5[lis1_, lis2_] := Total[Mod[lis1 + lis2, 2]]

In[10]:= HammingDistance5[l1, l2]

Out[10]= 3
```

   Some timing tests show that the implementation with `Total` is quite a bit more efficient than the previous versions.

```
In[11]:= data1 = Table[Random[Integer], {10^6}];
```

```
In[12]:= data2 = Table[Random[Integer], {10^6}];

In[13]:= Timing[HammingDistance5[data1, data2]]

Out[13]= {0.06 Second, 499016}

In[14]:= Timing[HammingDistance4[data1, data2]]

Out[14]= {0.691 Second, 499016}

In[15]:= Timing[HammingDistance3[data1, data2]]

Out[15]= {2.514 Second, 499016}
```

3. *a.*

```
In[16]:= frequencies[lis_] := Module[{pair},
           pair[x_] := {x, Count[lis, x]};
           Map[pair, Union[lis]]]

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

Out[17]= {{a, 3}, {b, 3}, {c, 2}}
```

b.

```
In[18]:= 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[19]:= split1[Range[10], {2, 5, 0, 3}]

Out[19]= {{1, 2}, {3, 4, 5, 6, 7}, {}, {8, 9, 10}}

In[20]:= split2[lis_, parts_] := Module[{lis1},
           lis1[x_] := Take[lis, x + {1, 0}];
           Map[lis1, Partition[FoldList[Plus, 0, parts], 2, 1]]]

In[21]:= split2[Range[10], {2, 5, 0, 3}]

Out[21]= {{1, 2}, {3, 4, 5, 6, 7}, {}, {8, 9, 10}}
```

c.

```
In[22]:= lotto1[lis_, n_] := Module[{lis1, lis2, lis3}, lis1[x_] :=
            Flatten[Rest[MapThread[Complement, {RotateRight[x], x}, 1]]];
           lis2[y_] := Delete[y, Random[Integer, {1, Length[y]}]];
           lis3[z_] := NestList[lis2, z, n];
           lis1[lis3[lis]]]
```

```
In[23]:= lotto1[Range[10], 5]

Out[23]= {2, 5, 4, 6, 8}

In[24]:= lotto2[lis_, n_] := Take[Transpose[
             Sort[Transpose[{Table[Random[], {Length[lis]}], lis}]]][[2]], n]

In[25]:= lotto2[Range[10], 5]

Out[25]= {2, 5, 1, 8, 7}
```

4.

```
In[26]:= {Timing[lotto1[Range[50000], 3];], Timing[lotto2[Range[50000], 3];]}

Out[26]= {{0.09 Second, Null}, {0.421 Second, Null}}

In[27]:= {Timing[lotto1[Range[50000], 60];], Timing[lotto2[Range[50000], 60];]}

Out[27]= {{1.362 Second, Null}, {0.42 Second, Null}}
```

5. Here are the list of coins.

```
In[28]:= coins = {p, p, q, n, d, d, p, q, q, p}

Out[28]= {p, p, q, n, d, d, p, q, q, p}

In[29]:= pocketChange2[x_] :=
          Dot[Map[(Count[x, #] &), {p, n, d, q}], {1, 5, 10, 25}]

In[30]:= pocketChange2[coins]

Out[30]= 104

In[31]:= pocketChange3[x_] :=
          Inner[Times, Map[(Count[x, #] &), {p, n, d, q}], {1, 5, 10, 25}, Plus]

In[32]:= pocketChange3[coins]

Out[32]= 104
```

6.

```
In[33]:= makeChange[x_] := Module[{coins = {25, 10, 5, 1}},
           Quotient[FoldList[Mod, x, Drop[coins, -1]], coins]]

In[34]:= makeChange[119]

Out[34]= {4, 1, 1, 4}
```

7.

```
In[35]:= offLattice[n_] :=
          Map[({Sin[#], Cos[#]} &), Table[Random[Real, {0, 2 π}], {n}]]

In[36]:= offLattice[n_] := Module[{step},
           step[x_] := {Sin[x], Cos[x]};
           Map[step, Table[Random[Real, {0, 2 π}], {n}]]]

In[37]:= offLattice[3]

Out[37]= {{0.194181, 0.980966}, {0.956556, -0.291548}, {-0.431374, -0.902173}}
```

8. First, notice what `FromDigits` does.

```
In[38]:= ? FromDigits

         FromDigits[list] constructs an integer
            from the list of its decimal digits. FromDigits[
            list, b] takes the digits to be given in base b. More...
```

We use `With` to create a local constant `d`, as this expression never changes throughout the body of the function.

```
In[39]:= convertToDate2[num_] := With[{d = IntegerDigits[num]},
           {FromDigits[Take[d, 4]],
            FromDigits[Take[d, {5, 6}]],
            FromDigits[Take[d, {7, 8}]]}]

In[40]:= convertToDate2[20030515]

Out[40]= {2003, 5, 15}
```

5 Procedural programming

5.2 Loops and iteration

1. Using a compound expression inside the `Do` function, this computes the next approximations of both square roots each time through the loop.

```
In[1]:= next[fun_, x_] := N[x - fun[x] / fun'[x]]
```

```
In[2]:= a = 50;
        b = 60;
        Do[
         a = next[#^2 - 50 &, a];
         b = next[#^2 - 60 &, b],
         {10}]

In[5]:= {a, b}

Out[5]= {7.07107, 7.74597}
```

2. Notice that to compute the square root of a number r, we need to iterate the following expression.

```
In[6]:= fun[x_] := x^2 - r;
        Simplify[x - fun[x]/fun'[x]]

Out[7]= (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.

```
In[8]:= nestSqrt[r_, init_] := Nest[(r + #^2)/(2 #) &, N[init], 10]

In[9]:= nestSqrt[50, 10]

Out[9]= 7.07107
```

3. We need to place the two expressions that were in the body of the `Do` into a list. Try copying the body of the `Do` exactly as above and see what happens.

```
In[10]:= next[fun_, x_] := N[x - fun[x]/fun'[x]]

In[11]:= a = 50;
         b = 60;
         Table[{
           a = next[(#^2 - 50) &, a],
           b = next[(#^2 - 60) &, b]},
          {10}]

Out[13]= {{25.5, 30.5}, {13.7304, 16.2336}, {8.68597, 9.96482}, {7.22119, 7.993},
          {7.07263, 7.74978}, {7.07107, 7.74597}, {7.07107, 7.74597},
          {7.07107, 7.74597}, {7.07107, 7.74597}, {7.07107, 7.74597}}
```

To mimic the solution to this problem obtained with the Do loop, we need to extract the last set of values obtained.

```
In[14]:= Last[%]

Out[14]= {7.07107, 7.74597}
```

4. Note that this version of the Fibonacci function is much more efficient than the simple recursive version, and is closer to the version that uses dynamic programming.

```
In[15]:= fib[n_] := Module[{prev = 0, this = 1, next},
           Do[next = prev + this;
            prev = this;
            this = next,
            {n}];
           prev]

In[16]:= Table[fib[i], {i, 1, 10}]

Out[16]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

Actually, this code can be simplified a bit by using parallel assignments.

```
In[17]:= fib2[n_] := Module[{f1 = 0, f2 = 1},
           Do[{f1, f2} = {f2, f1 + f2},
            {n - 1}];
           f2]

In[18]:= Table[fib2[i], {i, 1, 10}]

Out[18]= {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[19]:= {Timing[fib[100000];], Timing[fib2[100000];]}

Out[19]= {{0.781 Second, Null}, {0.631 Second, Null}}
```

5. We compute the derivative `df` inside the `Module` and then use that throughout the body of the function.

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

In[21]:= findRoot[fun_, init_, ϵ_] :=
          Module[{xi = init, funxi = fun[init], df = fun'}, While[Abs[funxi] > ϵ,
            xi = N[xi - funxi/df[xi]];
            funxi = fun[xi]];
           xi]
```

```
In[22]:= findRoot[f, 50, 0.0001]

Out[22]= 50
```

6. The variable `b` is the current approximation, and the variable `a` is the previous approximation.

```
In[23]:= findRoot[fun_, init_, ϵ_] := Module[{a = init, b = fun[init]},
           While[Abs[b - a] > ϵ,
            a = b;
            b = N[b - fun[b]/fun'[b]]];
           b]

In[24]:= f[x_] := x^2 - 50

In[25]:= findRoot[f, 10, .001]

Out[25]= 7.07107
```

7. This solution is based on the solution to Exercise 5 above.

```
In[26]:= 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 = N[b - fun[b]/fun'[b]];
            solns = Join[solns, {a}]];
           Join[solns, {b}]]

In[27]:= f[x_] := x^2 - 50

In[28]:= findRootList[f, 50, 10^-6]

Out[28]= {50, 25.5, 13.7304, 8.68597, 7.22119, 7.07263, 7.07107, 7.07107}
```

8. We go back to a previous version of `findRoot` and add multiple initial values.

```
In[29]:= findRootList[fun_, inits_, 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[30]:= findRootList[(#^2 - 50) &, {25, 50, 75, 100}, .001]

Out[30]= {7.07107}
```

9.

```
In[31]:= bisect[f_, {a_, b_, ϵ_}] := Module[
           {low = Min[a, b], high = Max[a, b], mid = N[(a + b)/2], fofMid = N[f[(a + b)/2]]},
           While[Abs[fofMid] > ϵ,
            If[fofMid < 0, low = mid, high = mid];
            mid = N[(low + high)/2];
            fofMid = N[f[mid]]];
           mid]

In[32]:= f[x_] := x^2 - 2
         bisect[f, {0, 2, .001}]

Out[33]= 1.41406
```

10. Here is a direct implementation of the Euclidean algorithm.

```
In[34]:= gcd[m_, n_] := Module[{a = m, b = n, tmpa},
           While[b > 0,
            tmpa = a;
            a = b;
            b = Mod[tmpa, b]];
           a]

In[35]:= m = 12782;
         n = 5531207;
         gcd[m, n]

Out[37]= 11
```

We can avoid the need for the temporary variable `tmpa` by performing a parallel assignment as in the following function. This results in a much cleaner implementation.

```
In[38]:= gcd[m_, n_] := Module[{a = m, b = n},
           While[b > 0, {a, b} = {b, Mod[a, b]}];
           a]

In[39]:= m = 12782;
         n = 5531207;
         gcd[m, n]

Out[41]= 11
```

11.

a. Create a list `rvec` of 0s, then use a `Do` loop to set `rvec[[i]]` to `vec[[n - i]]`, where *n* is the length of `vec`.

```
In[42]:= Clear[reverse, a, b, c, d, e]

In[43]:= reverse[vec_] := Module[{vecA = Table[0, {Length[vec]}]},
           Do[vecA[[i]] = vec[[Length[vec] - i + 1]],
            {i, 1, Length[vec]}];
           vecA]

In[44]:= reverse[{a, b, c, d, e}]

Out[44]= {e, d, c, b, a}

In[45]:= reverseStruc[vec_] := Module[{vecA = Table[0, {len = Length[vec]}]},
           Table[vecA[[i]] = vec[[len - i + 1]], {i, len}]
          ]

In[46]:= reverseStruc[{a, b, c, d, e}]

Out[46]= {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, …, `Length[`*vec*`]` − 1, whereas vectors are indexed by values 1, …, `Length[`*vec*`]`. This causes a little trickiness in this problem.

```
In[47]:= rotateRight[vec_, n_] := Module[{vecA = Table[0, {Length[vec]}]},
           Do[vecA[[1 + Mod[n + i - 1, Length[vec]]]] = vec[[i]], {i, 1, Length[vec]}];
           vecA]

In[48]:= rotateRight[{a, b, c, d, e}, 2]

Out[48]= {d, e, a, b, c}

In[49]:= rotateRightStruc[vec_, n_] :=
          Module[{vecA = Table[0, {len = Length[vec]}]},
           Table[vecA[[1 + Mod[n + i - 1, len]]] = vec[[i]], {i, len}];
           vecA
          ]

In[50]:= rotateRightStruc[{a, b, c, d, e}, 3]

Out[50]= {c, d, e, a, b}
```

c. Iterate over the rows of `mat`, setting row *i* to the result of calling `rotateRight`.

```
In[51]:= rotateRows[mat_] := Module[{matA = Table[0, {len = Length[mat]}]},
           Do[matA[[i]] = rotateRight[mat[[i]], i],
            {i, 1, len}];
           matA]
```

```
In[52]:= rotateRows[{{a, b, c}, {d, e, f}, {g, h, k}}]

Out[52]= {{c, a, b}, {e, f, d}, {g, h, k}}
```

d.

```
In[53]:= rotateRowsByS[mat_, S_] := Module[{matA = Table[0, {Length[mat]}]},
           Do[matA[[i]] = rotateRight[mat[[i]], S[[i]]],
            {i, 1, Length[mat]}];
           matA]

In[54]:= rotateRowsByS[{{a, b, c}, {d, e, f}, {g, h, k}}, {1, 2, 3}]

Out[54]= {{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[55]:= compress[lisA_, lisB_] :=
          Module[{lisC = Table[0, {Count[lisB, True]}], last = 1},
           Do[If[lisB[[i]], lisC[[last]] = lisA[[i]];
              last = last + 1,
              Null],
            {i, 1, Length[lisB]}];
           lisC]

In[56]:= compress[{a, b, c, d, e}, {True, True, False, False, True}]

Out[56]= {a, b, e}
```

5.3 Flow control

1. Here are the conditional definitions.

```
In[1]:= signum1[x_ /; x < 0] := -1
        signum1[x_ /; x > 0] := 1
        signum1[0] := 0

In[4]:= Map[signum1, {-2, 0, 1}]

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

Here is the signum function defined using `If`.

```
In[5]:= signum2[x_] := If[x < 0, -1, If[x == 0, 0, 1]]
```

```
In[6]:= Map[signum2, {-2, 0, 1}]

Out[6]= {-1, 0, 1}
```

Here is the signum function defined using `Which`.

```
In[7]:= signum3[x_] := Which[x < 0, -1, x == 0, 0, True, 1]

In[8]:= Map[signum3, {-2, 0, 1}]

Out[8]= {-1, 0, 1}
```

Finally, here is the signum function defined using `Piecewise`.

```
In[9]:= Piecewise[{{-1, x < 0}, {1, x > 0}, {0, x == 0}}]

Out[9]= { -1  x < 0
        { 1   x > 0
```

2.

```
In[10]:= signum1[x_ /; x < 0] := -1
         signum1[x_ /; x > 0] := 1
         signum1[0] := 0
         signum1[0.0] := 0

In[14]:= Map[signum1, {-2, 0, 2}]

Out[14]= {-1, 0, 1}

In[15]:= signum2[x_] := If[x < 0, -1, If[x > 0, 1, 0]]

In[16]:= Map[signum2, {-2, 0, 2}]

Out[16]= {-1, 0, 1}

In[17]:= signum3[x_] := Which[x < 0, -1, x > 0, 1, True, 0]

In[18]:= Map[signum3, {-2, 0, 2}]

Out[18]= {-1, 0, 1}
```

3.

```
In[19]:= applyChar[{"+", nums__}] := Apply[Plus, {nums}]
         applyChar[{"-", nums__}] := Apply[Minus, {nums}]
         applyChar[{"*", nums__}] := Apply[Times, {nums}]
         applyChar[{"/", nums__}] := Apply[Divide, {nums}]
         applyChar[_] := Print["Bad argument to applyChar"];
```

4.

a.

```
In[24]:= doublePos[lis_] := Map[If[# > 0, 2 #, #] &, lis]
```

b.

```
In[25]:= remove3Repetitions[lis_] := Fold[
           If[Length[#1] > 2 && #2 == #1[[-1]] == #1[[-2]], #1, Join[#1, {#2}]] &, {}, lis]
```

c.

```
In[26]:= positiveSum[L_] := Fold[If[#1 + #2 < 0, 0, #1 + #2] &, 0, L]
```

5. First we define the auxiliary function using conditional statements.

```
In[27]:= collatz[n_] := n/2 /; EvenQ[n]
```

```
In[28]:= collatz[n_] := 3 n + 1 /; OddQ[n]
```

Then iterate `Collatz`, starting with `n`, and continue while `n` is not equal to 1.

```
In[29]:= CollatzSequence[n_] := NestWhileList[collatz, n, ! (# == 1) &]
```

```
In[30]:= CollatzSequence[13]
Out[30]= {13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

5.4 Examples

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]:= m = 12782;
        n = 5531207;
        gcd[m, n]
Out[5]= 11
```

2. This is a direct implementation using `Piecewise`.

```
In[6]:= 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]

Out[6]= { 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   True
```

```
In[7]:= pointLocPW[{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[8]:= Map[pointLocPW, {{0, 0}, {4, 0}, {0, 1.3},
          {2, 4}, {-2, 4}, {-2, -4}, {2, -4}, {2, 0}, {3, -4}}]

Out[8]= {0, -1, -2, 1, 2, 3, 4, -1, 4}
```

3.

```
In[9]:= pointLoc[{0, 0}] := 0
        pointLoc[{x_, 0}] := -1
        pointLoc[{0, y_}] := -2
        pointLoc[{x_, y_}] := If[x < 0, 2, 1] /; y > 0
        pointLoc[{x_, y_}] := If[x < 0, 3, 4]
        pointLoc[{x_, y_, z_}] := If[x < 0, 2, 1] /; y ≥ 0 && z ≥ 0
        pointLoc[{x_, y_, z_}] := If[x < 0, 3, 4] /; y < 0 && z ≥ 0
        pointLoc[{x_, y_, z_}] := If[x < 0, 6, 5] /; y ≥ 0 && z < 0
        pointLoc[{x_, y_, z_}] := If[x < 0, 7, 8] /; y < 0 && z < 0
```

```
In[18]:= Map[pointLoc, {{2, 0}, {3, -4}}]

Out[18]= {-1, 4}
```

6 Rule-based programming

6.2 Patterns

1. Using the `FullForm` of the expression, we can find many pattern matches.

```
In[1]:= FullForm[x^3 + y z]

Out[1]//FullForm=
Plus[Power[x, 3], Times[y, z]]
```

```
In[2]:= MatchQ[x^3 + y z, _Plus]

Out[2]= True

In[3]:= MatchQ[x^3 + y z, _Power + _Times]

Out[3]= True
```

There are many more possible matches, including the trivial one.

```
In[4]:= MatchQ[x^3 + y z, _]

Out[4]= True
```

2. First look at the `FullForm` of this expression.

```
In[5]:= FullForm[{5, erina, "give me a break"}]

Out[5]//FullForm=
List[5, erina, "give me a break"]

In[6]:= MatchQ[{5, erina, "give me a break"}, _List]

Out[6]= True

In[7]:= MatchQ[{5, erina, "give me a break"}, {_Integer, _Symbol, _String}]

Out[7]= True
```

3. Again, the `FullForm` should help to guide you.

```
In[8]:= FullForm[{4, {a, b}, "g"}]

Out[8]//FullForm=
List[4, List[a, b], "g"]

In[9]:= MatchQ[{4, {a, b}, "g"}, x_List /; Length[x] == 3]

Out[9]= True

In[10]:= MatchQ[{4, {a, b}, "g"}, _List?(Length[#1] == 3 &)]

Out[10]= True

In[11]:= MatchQ[{4, {a, b}, "g"}, {_, y_, _} /; y[[0]] == List]

Out[11]= True

In[12]:= MatchQ[{4, {a, b}, "g"}, {x_, y_, z_} /; AtomQ[z]]

Out[12]= True
```

```
In[13]:= MatchQ[{4, {a, b}, "g"}, {x_, _, _} /; EvenQ[x]]

Out[13]= True
```

4. Here is the original solution as from Chapter 5, but, in this case, we check that both `m` and `n` have head `Integer`.

```
In[14]:= gcd[m_Integer, n_Integer] := Module[{a = m, b = n},
           While[b > 0, {a, b} = {b, Mod[a, b]}];
           a]

In[15]:= gcd[39874, 2868878]

Out[15]= 2
```

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

```
In[16]:= FindSubsequence[lis_List, subseq_List] := Module[{p},
           p = Partition[lis, Length[subseq], 1];
           Position[p, Flatten[{___, subseq, ___}]]
         ]
```

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

```
In[17]:= FindSubsequence[n_Integer, subseq_Integer] :=
          Module[{nlist = IntegerDigits[n], sublist = IntegerDigits[subseq]},
           FindSubsequence[nlist, sublist]
          ]
```

We create the list of the first 100,000 digits of π.

```
In[18]:= pi = FromDigits[RealDigits[N[Pi, 10^5] - 3][[1]]];
```

This show that the subsequence 1415 occurs seven times at the following locations in the digit expansion of π.

```
In[19]:= FindSubsequence[pi, 1415]

Out[19]= {{1}, {6955}, {29136}, {45234}, {79687}, {85880}, {88009}}
```

6. The Collatz function has a direct implementation based on its definition.

```
In[20]:= Collatz[n_?OddQ] := 3 n + 1

In[21]:= Collatz[n_?EvenQ] := n/2

In[22]:= Collatz[4.3]

Out[22]= Collatz[4.3]
```

Here we iterate the Collatz function 111 times starting with an initial value of 27.

```
In[23]:= NestList[Collatz, 27, 111]

Out[23]= {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}
```

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

```
In[24]:= Clear[Collatz]

In[25]:= Collatz[n_] := 3 n + 1 /; OddQ[n] && Positive[n]

In[26]:= Collatz[n_] := n/2 /; EvenQ[n] && Positive[n]

In[27]:= Collatz[4.3]

Out[27]= Collatz[4.3]

In[28]:= Collatz[-3]

Out[28]= Collatz[-3]

In[29]:= NestList[Collatz, 27, 111]

Out[29]= {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}
```

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

```
In[30]:= abs[x_Real | x_Integer | x_Rational] := If[x ≥ 0, x, -x]
```

Here is the definition for complex arguments.

```
In[31]:= abs[x_Complex] := √(Re[x]^2 + Im[x]^2)
```

It is probably a good idea to also add a definition for symbolic arguments.

```
In[32]:= abs[x_Symbol] := Abs[x]
```

```
In[33]:= Map[abs, {-3, 3 + 4 I, -4/5, a}]

Out[33]= {3, 5, 4/5, Abs[a]}
```

9. 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[34]:= swapTwo[{}] := {}
         swapTwo[{x_}] := {x}
```

Now, we use the triple-blank to indicate that `r` could be a sequence of 0 or more elements.

```
In[36]:= swapTwo[{x_, y_, r___}] := {y, x, r}

In[37]:= Map[swapTwo, {{}, {a}, {a, b, c, d}}]

Out[37]= {{}, {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[38]:= swapTwo2[{x_, y_, r___}] := {y, x, r}
         swapTwo2[x_] := x

In[40]:= Map[swapTwo2, {{}, {a}, {a, b, c, d}}]

Out[40]= {{}, {a}, {b, a, c, d}}
```

10. This one requires the triple blank.

```
In[41]:= f[{x1_Integer, ___}, 1] := x1 + 1
         f[x_Integer, y_] := x - y
```

11. Here are two sample lists.

```
In[43]:= l1 = {1, 0, 0, 1, 1};
         l2 = {0, 1, 0, 1, 0};
```

First we pair them up.

```
In[45]:= ll = Transpose[{l1, l2}]

Out[45]= {{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.

```
In[46]:= Cases[ll, {p_, q_} /; p ≠ q]

Out[46]= {{1, 0}, {0, 1}, {1, 0}}
```

The Hamming distance is the number of such non-identical pairs.

```
In[47]:= Length[%]

Out[47]= 3
```

Finally, here is a function that puts this all together.

```
In[48]:= HammingDistance[lis1_List, lis2_List] :=
          Length[Cases[Transpose[{lis1, lis2}], {p_, q_} /; p ≠ q]]

In[49]:= HammingDistance[l1, l2]

Out[49]= 3
```

The running times of this version of `HammingDistance` are comparable with those from Chapter 4, where we used bit operators.

```
In[50]:= HammingDistance2[lis1_, lis2_] := Apply[Plus,
           Apply[BitXor, Transpose[{lis1, lis2}], {1}]
         ]

In[51]:= data1 = Table[Random[Integer], {10^6}];

In[52]:= data2 = Table[Random[Integer], {10^6}];

In[53]:= Timing[HammingDistance[data1, data2]]

Out[53]= {2.905 Second, 500168}

In[54]:= Timing[HammingDistance2[data1, data2]]

Out[54]= {1.592 Second, 500168}
```

6.3 Transformation rules

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

```
In[1]:= maxima[x_] := Union[Rest[FoldList[Max, -∞, x]]]

In[2]:= maximaR[x_List] := x //. {a___, b_, c___, d_, e___} /; d ≤ b :→ {a, b, c, e}
```

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

Out[3]= {maxima[{3, 5, 2, 6, 1, 8, 4, 9, 7}],
         Union[Rest[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}},
          Rest[{-∞, 3, 5, 5, 6, 6, 8, 8, 9, 9}], {3, 5, 5, 6, 6, 8, 8, 9, 9}},
         Union[{3, 5, 5, 6, 6, 8, 8, 9, 9}], {3, 5, 6, 8, 9}}

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

Out[4]= {maximaR[{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}, {6 ≤ 5, False},
         {1 ≤ 3, True}, {5 ≤ 3, False}, {6 ≤ 3, False}, {6 ≤ 5, False},
         {8 ≤ 3, False}, {8 ≤ 5, False}, {8 ≤ 6, False}, {4 ≤ 3, False}, {4 ≤ 5, True},
         {5 ≤ 3, False}, {6 ≤ 3, False}, {6 ≤ 5, False}, {8 ≤ 3, False},
         {8 ≤ 5, False}, {8 ≤ 6, False}, {9 ≤ 3, False}, {9 ≤ 5, False},
         {9 ≤ 6, False}, {9 ≤ 8, False}, {7 ≤ 3, False}, {7 ≤ 5, False},
         {7 ≤ 6, False}, {7 ≤ 8, True}, {5 ≤ 3, False}, {6 ≤ 3, False}, {6 ≤ 5, False},
         {8 ≤ 3, False}, {8 ≤ 5, False}, {8 ≤ 6, False}, {9 ≤ 3, False},
         {9 ≤ 5, False}, {9 ≤ 6, False}, {9 ≤ 8, False}, {3, 5, 6, 8, 9}}
```

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

```
In[5]:= Trace[y = 11; a = 9; y + 3 /. y → a]

Out[5]= {y = 11; a = 9; y + 3 /. y → a, {y = 11, 11}, {a = 9, 9}, {{{y, 11}, 11 + 3, 14},
           {{y, 11}, {a, 9}, 11 → 9, 11 → 9}, 14 /. 11 → 9, 14}, 14}
```

3. First make sure that `a` and `y` have no values associated with them.

```
In[6]:= Clear[a, y]

In[7]:= Hold[y = 11];
        a = 9;
        y + 3 /. y → a

Out[9]= 12
```

4. You need to maintain the left-hand side of the transformation rule unevaluated for purposes of pattern matching and the right-hand side of the rule unevaluated until the rule is used.

```
In[10]:= Trace[g[x_] = x /. +z___ → Times[z]]

Out[10]= {{{{+z___, z___}, {Times[z], z}, z___ → z, z___ → z}, x /. z___ → z, x},
          g[x_] = x, x}

In[11]:= Clear[a, g]

In[12]:= g[x_] := x /. Literal[+z___] :> Times[z]

In[13]:= g[a + b + c]

Out[13]= a b c
```

5. The transformation rule unnests lists within a list.

```
In[14]:= unNest[lis_] := Map[(# //. {x__List} → x &), lis]

In[15]:= unNest[{{a, a, a}, {a}, {{b, b, b}, {b, b}}, {a, a}}]

Out[15]= {{a, a, a}, {a}, {b, b, b}, {b, b}, {a, a}}
```

6.

```
In[16]:= sumList[lis_] := First[lis //. {x_, y___} → x + {y}]

In[17]:= sumList[{1, 5, 8, 3, 9, 3}]

Out[17]= 29
```

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

```
In[18]:= cartesianProduct[lis1_, lis2_] :=
          ReplaceList[{lis1, lis2}, {{___, x_, ___}, {___, y_, ___}} → {x, y}]
```

We should also have a rule for the base case.

```
In[19]:= cartesianProduct[{}] := {}

In[20]:= Clear[x, y, z, a, b, c]

In[21]:= cartesianProduct[{a, b, c}, {x, y, z}]

Out[21]= {{a, x}, {a, y}, {a, z}, {b, x}, {b, y}, {b, z}, {c, x}, {c, y}, {c, z}}

In[22]:= cartesianProduct[{}]

Out[22]= {}
```

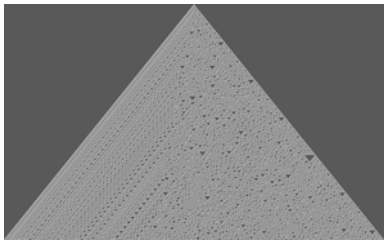
8. Note that `RasterArray` and `Raster` both display an array of values from bottom to top, hence the need to reverse the argument `lis`.

```
In[23]:= CAGraphics[lis_List] := Module[{colors},
           colors = {1 → Hue[.2], 0 → Hue[.8]};
           Graphics[RasterArray[Reverse[lis] /. colors]]
         ]
```

Here is a larger example of rule 30.

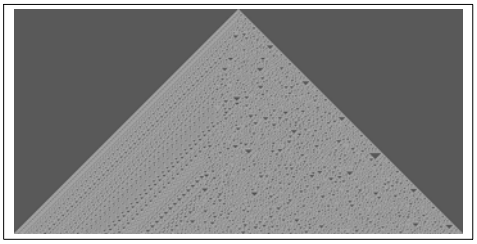
```
In[1]:= ca30 = CellularAutomaton[30, {{1}, 0}, 400];
```

```
In[25]:= Show[CAGraphics[ca30]];
```



Note that this can also be accomplished much more cleanly using `ArrayPlot` (new in Version 5.1).

```
In[26]:= ArrayPlot[ca30, ColorRules → {1 → Hue[.2], 0 → Hue[.8]}];
```



6.4 Examples

1. This is a simple modification of the code given in the text.

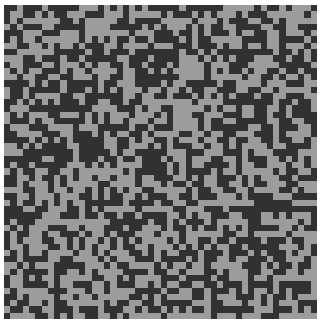
```
In[1]:= alphabet = Map[FromCharacterCode, Range[97, 122]]

Out[1]= {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[2]:= coderules = Thread[alphabet → RotateRight[alphabet, 5]]

Out[2]= {a → v, b → w, c → x, d → y, e → z, 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[3]:= decoderules = Thread[alphabet → RotateLeft[alphabet, 5]]

Out[3]= {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, v → a, w → b, x → c, y → d, z → e}

In[4]:= code[str_String] := Apply[StringJoin, Characters[str] /. coderules]

In[5]:= decode[str_String] := Apply[StringJoin, Characters[str] /. decoderules]

In[6]:= code["squeamish ossifrage"]

Out[6]= nlpzvhdnc jnndamvbz

In[7]:= decode[%]

Out[7]= squeamish ossifrage
```

3. This version of `matrixPlot` requires a list of rules as the second argument.

```
In[8]:= matrixPlot[mat_List, rules_] :=
         Show[Graphics[RasterArray[Reverse[mat] /. rules],
           AspectRatio → Automatic]]

In[9]:= dat = Table[Random[Integer], {50}, {50}];

In[10]:= matrixPlot[dat, {0 → GrayLevel[.2], 1 → GrayLevel[.6]}]
```

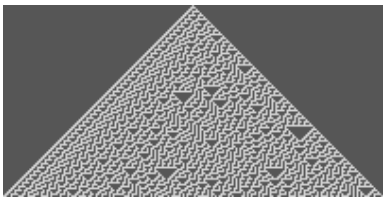


```
Out[10]= - Graphics -
```

You can plot any rectangular array of values with `matrixPlot` so long as you specify the rules for coloring the various elements. For example, the following example generates 100 steps in the evolution of the rule 30 cellular automaton, starting with a single 1 cell and surrounded by 0s.

```
In[11]:= ca30 = CellularAutomaton[30, {{1}, 0}, 100];
```

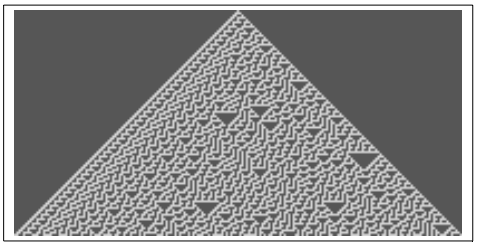
```
In[12]:= matrixPlot[ca30, {1 → Hue[.2], 0 → Hue[.6]}];
```



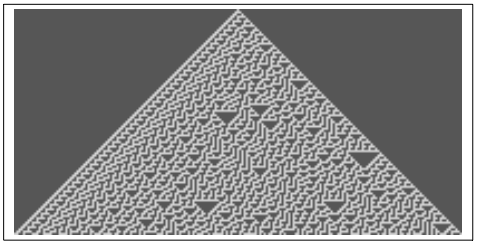
To get `matrixPlot` to produce similar output to the new `ArrayPlot`, you need to make a few changes to the `Frame` and `FrameTicks` options.

```
In[13]:= matrixPlot[mat_List, rules_] :=
          Show[Graphics[RasterArray[Reverse[mat] /. rules],
            AspectRatio → Automatic, Frame → True, FrameTicks → False]]
```

```
In[14]:= matrixPlot[ca30, {1 → Hue[.2], 0 → Hue[.6]}];
```

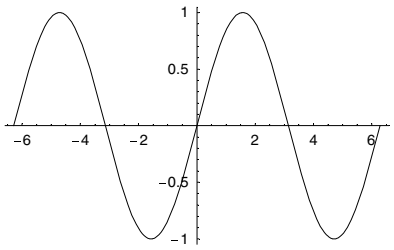


```
In[15]:= ArrayPlot[ca30, ColorRules → {1 → Hue[.2], 0 → Hue[.6]}];
```



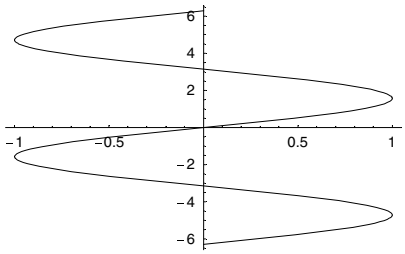
4. Here is the plot of the sine function.

```
In[13]:= splot = Plot[Sin[x], {x, -2 π, 2 π}];
```



This replacement rule interchanges each ordered pair of numbers.

In[14]:=
```
Show[splot /. {x_?NumberQ, y_?NumberQ} → {y, x}];
```

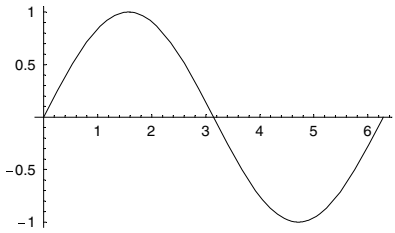


Although this particular example may have worked without the argument checking (`_?NumberQ`), it is a good idea to include it so that pairs of arbitrary expressions are not pattern matched here. We only want to interchange pairs of numbers, not pairs of options or other expressions that might be present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

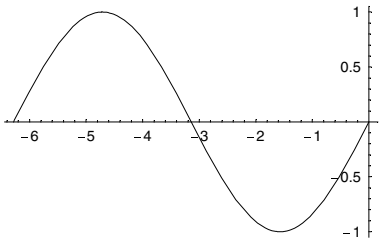
6. Using the standard rotation matrix, each point is taken to its image under the rotation transformation. Notice that this function first checks that its first argument is in fact a graphics object via pattern matching.

In[15]:=
```
rotatePlot[p_Graphics, theta_] := Show[p /. {x_?NumberQ, y_?NumberQ} →
    {x, y}.{{Cos[theta], Sin[theta]}, {-Sin[theta], Cos[theta]}}]
```

In[16]:=
```
plot1 = Plot[Sin[x], {x, 0, 2 π}];
```



In[17]:=
```
rotatePlot[plot1, Pi];
```



7 Recursion

7.1 Fibonacci numbers

1.

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, 209952, 408146688}
```

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[8]]

Out[17]= {0, 0, 1, 2, 4, 7, 12, 20}
```

7.2 List functions

1.

```
In[1]:= sumOddElements[{}] := 0
        sumOddElements[{x_, y___}] :=
         x + sumOddElements[{y}] /; IntegerQ[x] && OddQ[x]
        sumOddElements[{x_, y___}] := sumOddElements[{y}]

In[4]:= sumOddElements[{2, 3, 5, 6, 7, 9, 12, 13}]

Out[4]= 37
```

2.

```
In[5]:= sumEveryOtherElement[{}] := 0
        sumEveryOtherElement[{x_}] := x
        sumEveryOtherElement[{x_, y_, r___}] := x + sumEveryOtherElement[{r}]

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

Out[8]= 25
```

3.

```
In[9]:= addTriples[{}, {}, {}] := {}
        addTriples[{x1_, y1___}, {x2_, y2___}, {x3_, y3___}] :=
         Join[{x1 + x2 + x3}, addTriples[{y1}, {y2}, {y3}]]

In[11]:= addTriples[{w1, x1, y1, z1}, {w2, x2, y2, z2}, {w3, x3, y3, z3}]

Out[11]= {w1 + w2 + w3, x1 + x2 + x3, y1 + y2 + y3, z1 + z2 + z3}
```

4.

```
In[12]:= multAllPairs[{}] := {}
         multAllPairs[{_}] := {}
         multAllPairs[{x_, y_, r___}] := Join[{x y}, multAllPairs[{y, r}]]

In[15]:= multAllPairs[{3, 9, 17, 2, 6, 60}]

Out[15]= {27, 153, 34, 12, 360}
```

5.

```
In[16]:= maxPairs[{}, {}] := {}
         maxPairs[{x_, r___}, {y_, s___}] := Join[{Max[x, y]}, maxPairs[{r}, {s}]]

In[18]:= maxPairs[{1, 2, 4}, {2, 7, 2}]

Out[18]= {2, 7, 4}
```

6.

```
In[19]:= interleave[{}, {}] := {}
         interleave[{x_, r___}, {y_, s___}] := Join[{x, y}, interleave[{r}, {s}]]

In[21]:= interleave[{a, b, c}, {x, y, z}]

Out[21]= {a, x, b, y, c, z}
```

7.3 Thinking recursively: examples

1.

```
In[1]:= prefixMatch[L_, {}] := {}
        prefixMatch[{}, M_] := {}
        prefixMatch[{x_, r___}, {x_, s___}] := Join[{x}, prefixMatch[{r}, {s}]]
        prefixMatch[{x_, r___}, {y_, s___}] := {}
```

2.

```
In[5]:= runEncode2[{}] := {}
        runEncode2[{x_}] := {x}
        runEncode2[{x_, r__}] := runEncode2[{r}] /.
          {{{y_, k_}, s___} → If[x == y, {{x, k + 1}, s}, {x, {y, k}, s}],
           {y_, s___} → If[x == y, {{x, 2}, s}, {x, y, s}]}
```

3. 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[8]:= runEncode[{}] := {}
        runEncode[{x_}] := {{x, 1}}

In[10]:= 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` (named to mimic the built-in `Split`) simply operates on the output of `runEncode`.

```
In[11]:= sp[lis_] := Map[Table[#[[1]], {#[[2]]}] &, lis]

In[12]:= sp[{{3, 2}, {4, 1}, {2, 5}}]

Out[12]= {{3, 3}, {4}, {2, 2, 2, 2, 2}}

In[13]:= split[lis_] := sp[runEncode[lis]]
```

```
In[14]:= split[{9, 9, 9, 9, 9, 4, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5}]

Out[14]= {{9, 9, 9, 9, 9}, {4}, {3, 3, 3, 3}, {5, 5, 5, 5, 5, 5}}
```

4.

```
In[15]:= 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}]]
```

5.

```
In[20]:= maxima[{}] := {}
         maxima[{x_, r___}] := maxima[x, {r}]

In[22]:= maxima[x_, {}] := {x}
         maxima[x_, {y_, r___}] := maxima[x, {r}] /; x ≥ y
         maxima[x_, {y_, r___}] := Join[{x}, maxima[y, {r}]]
```

6.

```
In[25]:= runDecode[{}] := {}
         runDecode[{{x_, k_}, r___}] := Join[Table[x, {k}], runDecode[{r}]]
```

7. We will need two sets of rules for the `subsets` function.

```
In[27]:= Clear[subsets];
         subsets[lis_, {0}] := {{}}
         subsets[{}, {k_}] := {}

In[30]:= subsets[lis_, {k_}] := Module[{ksubs = subsets[Rest[lis], {k - 1}]},
           Join[Map[(Join[{First[lis]}, #] &), ksubs], subsets[Rest[lis], {k}]]]

In[31]:= subsets[Range[5], {2}]

Out[31]= {{1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 4}, {2, 5}, {3, 4}, {3, 5}, {4, 5}}
```

The second form simply calls the first.

```
In[32]:= subsets[lis_, k_] := Flatten[Map[subsets[lis, {#}] &, Range[0, k]], 1]
```

The second form simply calls the first. This gives all subsets up to length 3.

```
In[33]:= subsets[Range[5], 3]

Out[33]= {{}, {1}, {2}, {3}, {4}, {5}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3},
          {2, 4}, {2, 5}, {3, 4}, {3, 5}, {4, 5}, {1, 2, 3}, {1, 2, 4}, {1, 2, 5},
          {1, 3, 4}, {1, 3, 5}, {1, 4, 5}, {2, 3, 4}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5}}
```

A comparison with the built-in `Subsets` functions.

```
In[34]:= Subsets[Range[5], 3]

Out[34]= {{}, {1}, {2}, {3}, {4}, {5}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3},
          {2, 4}, {2, 5}, {3, 4}, {3, 5}, {4, 5}, {1, 2, 3}, {1, 2, 4}, {1, 2, 5},
          {1, 3, 4}, {1, 3, 5}, {1, 4, 5}, {2, 3, 4}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5}}
```

The recursion in this definition of `subsets` can get quite deep.

```
In[35]:= Timing[subsets[Range[1000], 2];]

          $RecursionLimit::reclim : Recursion depth of 256 exceeded. More…

          General::stop : Further output of $RecursionLimit::reclim will
             be suppressed during this calculation. More…
```

You can temporarily increase the value of `$RecursionLimit` to let this computation run to the end.

```
In[36]:= Timing[
          Block[{$RecursionLimit = ∞},
           subsets[Range[1000], 2];]]

Out[36]= {44.855 Second, Null}
```

But we can see pretty clearly just how inefficient our recursive approach to this problem is for large computations by comparing with the built-in `Subsets` function which is more than two orders of magnitude faster for sets of this size.

```
In[37]:= Timing[Subsets[Range[1000], 2];]

Out[37]= {0.14 Second, Null}
```

7.4 Recursion and symbolic computations

1.

```
In[1]:= ddx[c_?NumericQ] := 0
        ddx[x] := 1
        ddx[u_ + v_] := ddx[u] + ddx[v]
        ddx[u_ - v_] := ddx[u] - ddx[v]
        ddx[u_ v_] := u ddx[v] + v ddx[u]
        ddx[u_/v_] := (v ddx[u] - u ddx[v])/v^2
        ddx[u_^c_?NumericQ] := c u^(c-1) ddx[u]
```

```
In[8]:= ddx[Sin[u_]] := Cos[u] ddx[u]
        ddx[ Cos[u_]] := -Sin[u] ddx[u]
        ddx[Tan[u_]] := 1/Cos[u]^2 ddx[u]

In[11]:= ddx[Sin[2 x] + Cos[3 x]]

Out[11]= 2 Cos[2 x] - 3 Sin[3 x]

In[12]:= ddx[Tan[3 x^5]]

Out[12]= 15 x^4 Sec[3 x^5]^2
```

2.

```
In[13]:= Clear[ddx]

In[14]:= ddx[c_?NumericQ] := 0
         ddx[x] := 1
         ddx[u_ + v_] := ddx[u] + ddx[v]
         ddx[u_ - v_] := ddx[u] - ddx[v]
         ddx[u_ v_] := u ddx[v] + v ddx[u]
         ddx[u_/v_] := (v ddx[u] - u ddx[v])/v^2
         ddx[u_^c_?NumericQ] := c u^(c-1) ddx[u]

In[21]:= ddx[u_] := 0 /; nox[u]

In[22]:= nox[c_?NumericQ] := True
         nox[x] := False
         nox[y_] := True /; Head[y] == Symbol && y =!= x
         nox[u_ + v_] := nox[u] && nox[v]
         nox[u_ - v_] := nox[u] && nox[v]
         nox[u_ v_] := u nox[v] && v nox[u]
         nox[u_/v_] := nox[u] && nox[v]
         nox[u_^c_?NumericQ] := nox[u]
```

3.

```
In[30]:= Clear[ddx];
         ddx[c_?NumericQ, y_] := 0
         ddx[x_, x_] := 1
         ddx[y_, x_] := 0 /; FreeQ[y, x]
         ddx[u_ + v_, x_] := ddx[u, x] + ddx[v, x]
         ddx[u_ - v_, x_] := ddx[u, x] - ddx[v, x]
         ddx[u_ v_, x_] := u ddx[v, x] + v ddx[u, x]
         ddx[u_/v_, x_] := (v ddx[u, x] - u ddx[v, x])/v^2
         ddx[u_^c_?NumericQ, x_] := c u^(c-1) ddx[u, x]
```

```
In[39]:= ddx[α ξ^3 + β ξ^2 + γ, ξ]
Out[39]= 2 β ξ + 3 α ξ^2

In[40]:= ddx[θ/(1 + θ^3), θ]
Out[40]= (1 - 2 θ^3)/(1 + θ^3)^2
```

7.5 Classical examples

1. The solution to this problem also appears in Section 8.5. We will call our new function `solvep` (for pivoting).

```
In[1]:= Clear[solve]

In[2]:= solvep[S_] := Module[
          {S1 = pivot[S], E1, a12toa1n, x2toxn}, x2toxn = solvep[elimx1[S1]];
          E1 = First[S1];
          a12toa1n = Drop[Rest[E1], -1];
          Join[{(Last[E1] - a12toa1n.x2toxn)/First[E1]}, x2toxn]];

In[3]:= solvep[{{a11_, b1_}}] := {b1/a11}

In[4]:= elimx1[S_] := Map[subtractE1[S[[1]], #] &, Rest[S]]

In[5]:= subtractE1[E1_, Ei_] := Rest[Ei] - Ei[[1]]/E1[[1]] Rest[E1]

In[6]:= pivot[Q_] := Module[{p, ST1, pivotrow}, ST1 = Transpose[Q][[1]];
          p = Position[ST1, x_ /; x ≠ 0];
          If[p == {},
           Print["Matrix is singular"]; Q,
           pivotrow = p[[1]][[1]]; Join[{Q[[pivotrow]]}, Delete[Q, pivotrow]]]]

In[7]:= solve[A_, B_] := solvep[Transpose[Join[Transpose[A], {B}]]]
```

Here are some test examples.

```
In[8]:= mat = Table[Random[], {4}, {4}]
Out[8]= {{0.554127, 0.426593, 0.861278, 0.492521},
         {0.572684, 0.477244, 0.690375, 0.88366},
         {0.401935, 0.648486, 0.818292, 0.516009},
         {0.129603, 0.562562, 0.116779, 0.699194}}
```

```
In[9]:= b = Table[Random[], {4}]

Out[9]= {0.564681, 0.489887, 0.542515, 0.264061}

In[10]:= x = solve[mat, b]

Out[10]= {1.59998, 0.96497, -0.502052, -0.611457}

In[11]:= mat.x - b

Out[11]= {-1.11022×10^-16, -1.66533×10^-16, -1.11022×10^-16, -4.44089×10^-16}

In[12]:= Chop[%]

Out[12]= {0, 0, 0, 0}
```

2. To compute `solveUpper[`A`,`B`]`, first recursively compute `solveUpper[`A'`,`B'`]`, where A' is the lower-right square submatrix of A, and B' is the `Rest` of `B`. This solution gives the values of x_2, …, x_n. `B[[1]]` is equal to the dot product of the top row of A (*i.e.*, `A[[1]]`) and the vector x_1, …, x_n (that is, `B[[1]]`) is equal to `A[[1]]*`x_1 `+ ... + A[[n]]*`x_n. It is easy to compute x_1 from this formula.

```
In[13]:= solveUpper[{{ann_}}, {bn_}] := {bn/ann}

In[14]:= solveUpper[{A1_, rA__}, {b1_, rB__}] :=
          Module[{subsoln = solveUpper[Rest /@ {rA}, {rB}]},
           Join[{(b1 - Rest[A1].subsoln)/First[A1]}, subsoln]]
```

It is easy to show that if you rotate a matrix by 90 degrees, and turn the vector B upside down, the solution to the resulting system is the same as the solution to the original system, but turned upside down.

```
In[15]:= rotateMatrix[A_] := Reverse[Map[Reverse, A]]

In[16]:= solveLower[A_, B_] := Reverse[solveUpper[rotateMatrix[A], Reverse[B]]]
```

3.

```
In[17]:= LUdecomp1[S_] := Module[{mults = multipliers[S[[1, 1]], Rest[S]]}, Module[
             {Sprime = elimx1[mults, Rest /@ S]}, Module[{LU = LUdecomp1[Sprime]},
              {expandL[mults, LU[[1]]], expandU[First[S], LU[[2]]]}]]]

In[18]:= LUdecomp1[{{a11_}}] := {{{1}}, {{a11}}}

In[19]:= expandU[S1_, U_] := Join[{S1}, (Join[{0}, #1] &) /@ U]

In[20]:= expandL[mults_, L_] := Transpose[expandU[Join[{1}, mults], Transpose[L]]]
```

```
In[21]:= elimx1[mults_, subS_] :=
          Table[subS[[i + 1]] - mults[[i]] subS[[1]], {i, 1, Length[mults]}]

In[22]:= multipliers[S11_, restS_] := Map[ # / S11 &, Transpose[restS][[1]]]

In[23]:= LUdecomp2[S_] := Module[{soln = LUdecomp1[S]},
           soln[[1]] - IdentityMatrix[Length[S]] + soln[[2]]]
```

4.

```
In[24]:= sumNodes[{lab_}] := lab
         sumNodes[{lab_, lc_, rc_}] := lab + sumNodes[lc] + sumNodes[rc]
```

5.

```
In[26]:= catNodes[{lab_}] := lab
         catNodes[{lab_, lc_, rc_}] :=
          StringJoin[lab, catNodes[lc], catNodes[rc]]
```

6.

```
In[28]:= balanced[t_] := balancedHeight[t][[2]]
         balancedHeight[{lab_}] := {0, True}

In[30]:= balancedHeight[{lab_, lc_, rc_}] :=
          Module[{lbh, rbh}, lbh = balancedHeight[lc];
           If[lbh[[2]], rbh = balancedHeight[rc]; If[rbh[[2]] && Abs[lbh[[1]] - rbh[[1]]] ≤ 1,
             {Max[lbh[[1]], rbh[[1]]] + 1, True}, {0, False}], {0, False}]]
```

7.

```
In[31]:= listLevel[0, t_] := {t[[1]]}
         listLevel[{lab_}, n_] := {}

In[33]:= listLevel[{lab_, lc_, rc_}, n_] :=
          Join[listLevel[lc, n - 1], listLevel[rc, n - 1]]
```

8.

```
In[34]:= minInTree[{lab_}] := lab
         minInTree[{lab_, subtrees__}] :=
          Sort[Join[{lab}, Map[minInTree, {subtrees}]]][[1]]

In[36]:= height[{lab_}] := 0
         height[{lab_, subtrees__}] := 1 + Apply[Max, Map[height, {subtrees}]]

In[38]:= printTree[t_] := printTree[t, 0]

In[39]:= printTree[{lab_}, k_] := printIndented[lab, 3 k]
         printTree[{lab_, subtrees__}, k_] :=
          (printIndented[lab, 3 k]; Map[printTree[#, k + 1] &, {subtrees}];)
```

```
In[41]:= printIndented[x_, spaces_] :=
          Print[Apply[StringJoin, Table[" ", {spaces}]], x]
```

9. We have used a slightly different representation for the list of trees than the one shown in the chapter. Instead of a node's label containing a list of characters and a number, it contains a string and a number. The only reason for this is that it makes the result come out looking like the tree called `Htree` (shown in Figure 7.1). Note that the algorithm may give different results depending upon how it is programmed, since there are arbitrary choices made at each step. The result of applying our function `constructHTree` to the initial list of trees shown at the end of the last section (which we have included here as `testlist`) is different from `Htree`.

In our solution, we solve the problem of finding the two trees of smallest weight by keeping the list of trees sorted by weight; then we simply always pick the first two.

```
In[42]:= HTreeSort[trees_] := Sort[trees, #1[[1, 2]] < #2[[1, 2]] &]
```

```
In[43]:= joinHTrees[{{{cl_, wt_}, kids___}}] := {cl, kids}
         joinHTrees[{{{cl1_, wt1_}, kids1___},
            {{cl2_, wt2_}, kids2___}, trees___}] := joinHTrees[
           HTreeSort[{{{cl1 <> cl2, wt1 + wt2}, {cl1, kids1}, {cl2, kids2}}, trees}]]
```

```
In[45]:= constructHTree[t_] := joinHTrees[HTreeSort[t]]
```

```
In[46]:= htnode[a_, b_] := {{{a, b}}}
```

```
In[47]:= testlist = Join[htnode[" ", 6], htnode["A", 3],
           htnode["B", 1], htnode["E", 5], htnode["H", 2],
           htnode["N", 2], htnode["O", 2], htnode["S", 3], htnode["T", 3]]
```

```
Out[47]= {{{ , 6}}, {{A, 3}}, {{B, 1}}, {{E, 5}},
          {{H, 2}}, {{N, 2}}, {{O, 2}}, {{S, 3}}, {{T, 3}}}
```

10. To make the results here comparable to those in the book, we will use `Htree` from the book as our sample tree. `makeTreeTable[tree]` produces a list of rules as described in the problem. `encodeString[`*str*`, `*rules*`]` decodes the string according to those rules.

```
In[48]:= Htree = {" ABEHONST", {" AT", {" "}, {"AT", {"T"}, {"A"}}},
            {"BEHONS", {"EON", {"E"}, {"ON", {"O"}, {"N"}}},
             {"BHS", {"BH", {"H"}, {"B"}}, {"S"}}}};
```

```
In[49]:= makeTreeTable[prefix_, {ch_}] = {ch → prefix};
```

```
In[50]:= makeTreeTable[prefix_, {_, left_, right_}] :=
          Join[makeTreeTable[Join[prefix, {0}], left],
           makeTreeTable[Join[prefix, {1}], right]]
```

```
In[51]:= makeTreeTable[tree_] := makeTreeTable[{}, tree]
```

```
In[52]:= HtreeRules = makeTreeTable[Htree]

Out[52]= {  → {0, 0}, T → {0, 1, 0}, A → {0, 1, 1}, E → {1, 0, 0}, O → {1, 0, 1, 0},
          N → {1, 0, 1, 1}, H → {1, 1, 0, 0}, B → {1, 1, 0, 1}, S → {1, 1, 1}}

In[53]:= encodeString[str_, rules_] := Flatten[Characters[str] /. rules]

In[54]:= encodeString[str_] := encodeString[str, HtreeRules]
```

7.6 Dynamic programming

1. This implementation uses the identities given in the exercise together with some pattern matching

```
In[1]:= F[1] := 1
        F[2] := 1

In[3]:= F[n_ /; EvenQ[n]] := 2 F[n/2 - 1] F[n/2] + F[n/2]^2

        F[n_ /; OddQ[n]] := F[(n - 1)/2 + 1]^2 + F[(n - 1)/2]^2

In[5]:= Map[F, Range[10]]

Out[5]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

2.

```
In[6]:= FF[1] := 1
        FF[2] := 1

In[8]:= 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[10]:= Map[FF, Range[12]]

Out[10]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144}
```

3.

```
In[11]:= Clear[collatz]

In[12]:= collatz[n_, 0] := n

In[13]:= collatz[n_, i_] :=
           (collatz[n, i] = collatz[n, i - 1]/2) /; EvenQ[collatz[n, i - 1]]
```

```
In[14]:= 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[15]:= collatz[27, 5]

Out[15]= 31
```

Here is the Collatz sequence for 27. You can see that it takes a long time for this sequence to settle down to the cycle 4, 2, 1.

```
In[16]:= Table[collatz[27, i], {i, 0, 116}]

Out[16]= {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, 4, 2}
```

7.7 Higher-order functions and recursion

1. First, here is the definition for our user-defined `fold`.

```
In[1]:= fold[f_, x_, {}] := x
        fold[f_, x_, {a_, r___}] := fold[f, f[x, a], {r}]

In[3]:= fold[Plus, 0, {a, b, c, d, e}]

Out[3]= a + b + c + d + e

In[4]:= foldList[f_, x_, {}] := {x}
        foldList[f_, x_, {a_, r___}] := Join[{x}, foldList[f, f[x, a], {r}]]

In[6]:= foldList[Times, 1, Range[6]]

Out[6]= {1, 1, 2, 6, 24, 120, 720}
```

And here is `nestList`.

```
In[7]:= nestList[f_, x_, 0] := {x}
        nestList[f_, x_, n_] := Join[{x}, nestList[f, f[x], n - 1]]

In[9]:= nestList[Sin, θ, 3]

Out[9]= {θ, Sin[θ], Sin[Sin[θ]], Sin[Sin[Sin[θ]]]}
```

2. First, here is the recursive `repeat` function from this section.

```
In[10]:= repeat[f_, lis_, pred_] := lis /; pred[Drop[lis, -1], Last[lis]]

In[11]:= repeat[f_, lis_, pred_] := repeat[f, Append[lis, f[Last[lis]]], pred]
```

Then the `MemberQ` function is used to test whether the currently computed point is a member of the existing list of points.

```
In[12]:= landMineWalk[] :=
          repeat[#1 + {{0, 1}, {0, -1}, {1, 0}, {-1, 0}}[[Random[Integer, {1, 4}]]] &,
           {{0, 0}}, MemberQ[#1, #2] &]
```

Here is a test. On average, these walks will not be terribly long.

```
In[13]:= landMineWalk[]

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

8 Numerics

8.2 Numbers

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]}
```

Here are the computations for the examples in the text.

```
In[2]:= complexToPolar[3 + 3 i]

Out[2]= {3 Sqrt[2], Pi/4}

In[3]:= complexToPolar[E^(Pi i/3)]

Out[3]= {1, Pi/3}
```

2. This function uses a default value of 2 for the base. (Try replacing `Fold` with `FoldList` to more clearly see 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
```

This does the same for the number 129 in base 16.

```
In[7]:= IntegerDigits[129, 16]
Out[7]= {8, 1}

In[8]:= convert[%, 16]
Out[8]= 129
```

This function is essentially an implementation of Horner's method for fast polynomial multiplication.

```
In[9]:= Clear[a, b, c, d, e, x]

In[10]:= convert[{a, b, c, d, e}, x]
Out[10]= e + x (d + x (c + x (b + a x)))

In[11]:= Expand[%]
Out[11]= e + d x + c x^2 + b x^3 + a x^4
```

4. Here is the `sumsOfCubes` function.

```
In[12]:= sumsOfCubes[n_Integer] := Apply[Plus, IntegerDigits[n]^3]
```

Here is the function that performs the iteration.

```
In[13]:= sumsOfSums[n_Integer, iter_] := NestList[sumsOfCubes, n, iter]
```

We see that the number 4 enters into a cycle.

```
In[14]:= sumsOfSums[4, 12]
Out[14]= {4, 64, 280, 520, 133, 55, 250, 133, 55, 250, 133, 55, 250}
```

In fact, it appears as if many initial values enter cycles.

```
In[15]:= sumsOfSums[32, 12]
Out[15]= {32, 35, 152, 134, 92, 737, 713, 371, 371, 371, 371, 371, 371}

In[16]:= sumsOfSums[7, 12]
Out[16]= {7, 343, 118, 514, 190, 730, 370, 370, 370, 370, 370, 370, 370}
```

```
In[17]:= sumsOfSums[372, 12]

Out[17]= {372, 378, 882, 1032, 36, 243, 99, 1458, 702, 351, 153, 153, 153}
```

6. The function `sumsOfPowers` is a straightforward generalization of the previous cases.

```
In[18]:= sumsOfPowers[n_, p_] := Apply[Plus, IntegerDigits[n]^p]

In[19]:= sumsOfPowers[123, 5]

Out[19]= 276
```

8. Using the number 100 as an example, let us first put it in base 2.

```
In[20]:= BaseForm[100, 2]

Out[20]//BaseForm=
1100100_2
```

Here is the list of its digits.

```
In[21]:= IntegerDigits[100, 2]

Out[21]= {1, 1, 0, 0, 1, 0, 0}
```

This performs 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[22]:= l = RotateLeft[IntegerDigits[100, 2], 1]

Out[22]= {1, 0, 0, 1, 0, 0, 1}
```

This converts back from base 2 to base 10 (using the `convert` function from Exercise 2).

```
In[23]:= convert[l, 2]

Out[23]= 73
```

Now we can put all of this code together to make the `survivor` function.

```
In[24]:= survivor[n_] :=
          Module[{p}, p = RotateLeft[IntegerDigits[n, 2]]; Fold[(2 #1 + #2) &, 0, p]]

In[25]:= survivor[100]

Out[25]= 73
```

You could of course do the same thing without the symbol `p`, but it is just a bit less readable.

```
In[26]:= survivor2[n_Integer] :=
          Fold[(2 #1 + #2) &, 0, RotateLeft[IntegerDigits[n, 2]]]
```

```
In[27]:= survivor2[100]

Out[27]= 73
```

9. This function has a straightforward implementation. Each die can be viewed as a random integer between 1 and 6.

```
In[28]:= rollEm := {Random[Integer, {1, 6}], Random[Integer, {1, 6}]}

In[29]:= rollEm

Out[29]= {3, 2}
```

Here are five rolls in a row.

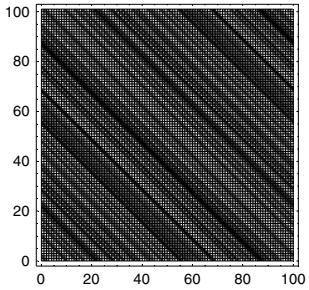
```
In[30]:= Table[rollEm, {5}]

Out[30]= {{6, 4}, {2, 3}, {5, 3}, {4, 3}, {4, 5}}
```

10. First generate a vector of 100 random real numbers on the interval 0 to 1.

```
In[31]:= data = Table[Random[], {100}];
```

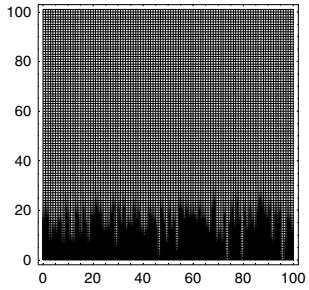
You could rotate once to the left for each successive row.

```
In[32]:= ListDensityPlot[NestList[RotateLeft, data, Length[data]]];
```

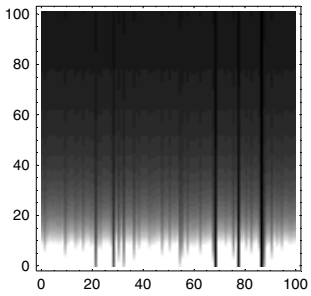


Here are a few other things you can try.

```
In[33]:= ListDensityPlot[NestList[#^.75 &, data, Length[data]]];
```



```
In[34]:= ListDensityPlot[NestList[#1/(1 + #1) &, data, Length[data]], Mesh → False];
```



11. Here is the linear congruential generator.

```
In[35]:= linearCongruential[x_,m_,b_] := Mod[b x + 1, m]
```

With modulus 100 and multiplier 15, this generator quickly gets into a cycle.

```
In[36]:= NestList[linearCongruential[#, 100, 15] &, 5, 10]

Out[36]= {5, 76, 41, 16, 41, 16, 41, 16, 41, 16, 41}
```

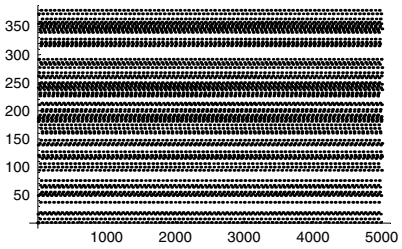
With a larger modulus and multiplier, it appears as if this generator is doing better.

Here are the first 60 terms starting with a seed of 0.

```
In[37]:= data = NestList[linearCongruential[#, 381, 15] &, 0, 60]

Out[37]= {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, 115}
```

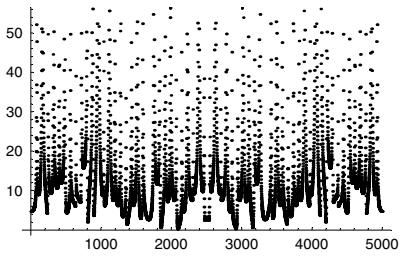
Sometimes it is hard to see if your generator is getting into a rut. Graphical analysis can help by allowing you to see patterns over larger domains. Here is a `ListPlot` of this sequence taken out to 5,000 terms.

```
In[38]:= ListPlot[NestList[linearCongruential[#, 381, 15] &, 0.0, 5000]];
```



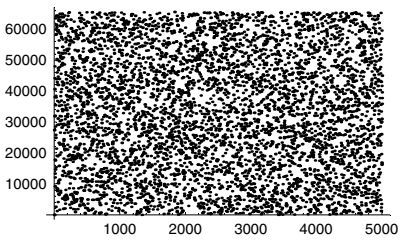
It appears as if certain numbers are repeating. Looking at the plot of the Fourier data shows peaks at certain frequencies, indicating a periodic nature to the data.

```
In[39]:= ListPlot[Abs[Fourier[
         NestList[linearCongruential[#,381,15]&,0.0,5000]]]];
```

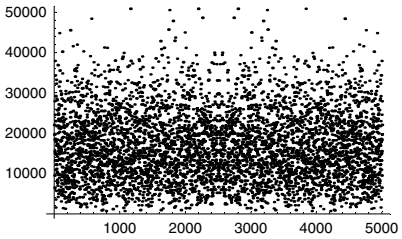


Using a much larger modulus and multiplier (chosen carefully), you can keep your generator from getting in such short loops.

```
In[40]:= ListPlot[
           data = NestList[linearCongruential[#, 2^16, 27421] &, 0.0, 5000]];
```




```
In[41]:= ListPlot[Abs[Fourier[data]]];
```



13. First we implement the chi-square test and then use it to run tests on some data in the next exercise.

```
In[42]:= chiSquare[lis_List] := Module[{m = Length[lis], n = Max[lis]},
          (Sum[(Count[lis, i] - m/n)^2, {i, 1, n}])/(m/n)]
```

14. Here are some data generated using the linear congruential generator with small modulus and multiplier.

```
In[43]:= data = NestList[linearCongruential[#, 381, 15] &, 0, 1000];

In[44]:= chiSquare[data]

Out[44]= 5018521/1001

In[45]:= N[%]

Out[45]= 5013.51
```

Notice that the statistic is quite far from $2\sqrt{n}$ of n. This is a particularly pathological sequence. You can see a cycle of length 63 within the first 100 iterates.

```
In[46]:= NestList[linearCongruential[#1, 381, 15] &, 0, 100]

Out[46]= {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, 115, 202, 364, 127, 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}
```

Here are those positions that contain the number 1.

```
In[47]:= Position[%, 1]

Out[47]= {{2}, {65}}
```

8.3 Working with numbers

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. You could simply produce 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[Sqrt[2], x]^200 - (Sqrt[2])^200]}, {x, 100, 140, 5}] // TableForm
```

```
Out[2]//TableForm=
100  67.596
105  72.596
110  77.596
115  82.596
120  87.596
125  92.596
130  97.596
135  102.596
140  107.596
```

8.4 Working with arrays of numbers

1. Note the need for a delayed rule in this function.

```
In[1]:= RandomSparseArray[n_Integer] := SparseArray[{{i_, i_} :> Random[]}, {n, n}]
```

```
In[2]:= Normal[RandomSparseArray[5]] // MatrixForm
```

```
Out[2]//MatrixForm=
⎛0.197227  0         0         0         0        ⎞
⎜0         0.509405  0         0         0        ⎟
⎜0         0         0.965962  0         0        ⎟
⎜0         0         0         0.873469  0        ⎟
⎝0         0         0         0         0.959528 ⎠
```

2. Here is the definition of `tridiagonalMatrix`.

```
In[3]:= tridiagonalMatrix[n_, p_, q_] :=
         SparseArray[{{i_, i_} -> p, ({i_, j_} /; Abs[i - j] == 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 = Table[Random[], {10^6}];

In[7]:= Developer`PackedArrayQ[vec]

Out[7]= True
```

Replacing the first element in `vec` with a 1 gives us an expression which 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 two and a half times larger than the packed array.

```
In[10]:= Map[ByteCount, {vec, newvec}]

Out[10]= {8000056, 20000032}

In[11]:= %[[2]] / %[[1]] // N

Out[11]= 2.49999
```

Sorting the packed object is about four or five times faster than sorting the unpacked object.

```
In[12]:= Timing[Do[Sort[vec], {5}]]

Out[12]= {4.406 Second, Null}

In[13]:= Timing[Do[Sort[newvec], {5}]]

Out[13]= {18.777 Second, Null}
```

Finding the minimum element is about one order of magnitude faster with the packed array.

```
In[14]:= Timing[Min[vec];]

Out[14]= {0.01 Second, Null}

In[15]:= Timing[Min[newvec];]

Out[15]= {0.131 Second, Null}
```

8.5 Numerical computations

1. We will overload `newton` to invoke the secant method when given a list of two numbers as the second argument.

```
In[1]:= Options[newton] = {
          MaxIterations :> $RecursionLimit,
          PrecisionGoal → Automatic,
          WorkingPrecision → Automatic
         };

In[2]:= newton[fun_, {x1_?NumericQ, x2_?NumericQ}, opts___?OptionQ] :=
         Module[{maxIterations, precisionGoal,
           workingPrecision, initx, df, next, result},
          {maxIterations, precisionGoal, workingPrecision} =
           {MaxIterations, PrecisionGoal, WorkingPrecision} /. Flatten[{opts}] /.
            Options[newton];
          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 = FixedPoint[next, initx, maxIterations][[2]];
          SetPrecision[result, precisionGoal]
         ]

In[3]:= f[x_] := x^2 - 2

In[4]:= newton[f, {1., 2.}]

Out[4]= 1.41421

In[5]:= newton[f, {1.0`60, 2.0`50}]

Out[5]= 1.41421356237309504880168872420969807856967409469 53

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
```

You can 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}}

Out[11]= {{1, 2, 3}, {1, 0, 2}, {2, -3, 2}}
```

Here, then, is the matrix norm.

```
In[12]:= norm[m_?MatrixQ, l_: Infinity] :=
          norm[Apply[Plus, Abs[Transpose[m]]], Infinity]

In[13]:= norm[mat]

Out[13]= 7
```

Again, 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_∞ norm).

```
In[15]:= conditionNumber[m_?MatrixQ] :=
          norm[m, Infinity] norm[Inverse[m], Infinity]

In[16]:= HilbertMatrix[n_] := Table[1/(i + j - 1), {i, n}, {j, n}]

In[17]:= conditionNumber[HilbertMatrix[3]]

Out[17]= 748
```

Compare this with the condition number of a random matrix.

```
In[18]:= conditionNumber[Table[Random[], {3}, {3}]]

Out[18]= 18.7428
```

Here are the condition numbers of the first ten Hilbert matrices.

```
In[19]:= Map[conditionNumber[HilbertMatrix[#]] &, Range[10]]

Out[19]= {1, 27, 748, 28375, 943656, 29070279, 1970389773/2,
          33872791095, 2199309082685/2, 35357439251992}

In[20]:= N[%]

Out[20]= {1., 27., 748., 28375., 943656., 2.90703×10^7,
          9.85195×10^8, 3.38728×10^10, 1.09965×10^12, 3.53574×10^13}
```

9 Graphics programming

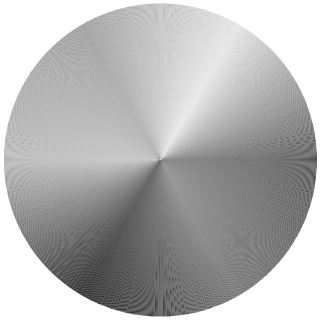
9.1 Structure of graphics

1. The color wheel can be obtained 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_] :=
          Show[Graphics[Map[{Hue[#/(2 π - n)], Disk[{0, 0}, 1, {#, # + n}]} &,
              Range[0, 2 π - n, n]], AspectRatio → Automatic]]
```

Here is a color wheel created from 256 separate sectors (hues).

```
In[2]:= colorWheel[π/256];
```



2. Here is the circle graphic primitive together with a text label.

```
In[3]:= circ = Circle[{0, 0}, 1];

In[4]:= ctext = Text[StyleForm["Circle",
            FontFamily → "Times", FontSlant -> "Italic", FontSize → 12],
          {Cos[5 π / 4] + .25, Sin[5 π / 4]}];
```

This generates the graphics primitive for the triangle and its text label.

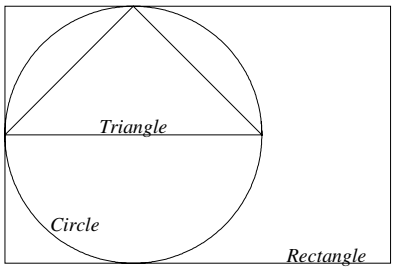
```
In[5]:= tri = Line[{{-1, 0}, {0, 1}, {1, 0}, {-1, 0}}];

In[6]:= ttext = Text[StyleForm["Triangle", FontFamily → "Times",
            FontSlant → "Italic", FontSize → 12], {0, 0 + .05}];
```

Here is the rectangle and label.

```
In[7]:= rect = Line[{{-1, -1}, {-1, 1}, {2, 1}, {2, -1}, {-1, -1}}];

In[8]:= rtext = Text[StyleForm["Rectangle", FontFamily → "Times",
            FontSlant → "Italic", FontSize → 12], {1.5, -1 + .05}];
```

Finally, this displays each of these graphics elements all together.

```
In[9]:= Show[Graphics[{circ, tri, rect, ctext, ttext, rtext}],
          AspectRatio → Automatic];
```



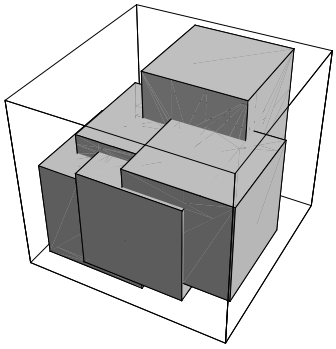
3. First, we need to create the cuboid graphic object. `Cuboid` takes a list of three numbers as the coordinates of its lower-left corner. This maps the object across two such lists.

```
In[10]:= Map[Cuboid[#] &, Table[Random[], {2}, {3}]]

Out[10]= {Cuboid[{0.177395, 0.551966, 0.857107}],
          Cuboid[{0.545712, 0.76829, 0.48344}]}
```

Here 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[#1] &, Table[Random[], {6}, {3}]];
```

```
In[12]:= Show[Graphics3D[cubes]];
```



4. First we create the `Point` graphics primitives randomly placed in the unit square.

```
In[13]:= randomcoords := Point[{Random[], Random[]}];
```

This creates the point sizes according to the specification given in the statement of the problem.

```
In[14]:= randomsize := PointSize[Random[Real, {.01, .1}]]
```

This will assign a random color to each primitive.

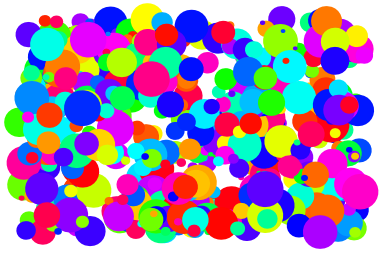
```
In[15]:= randomcolor := Hue[Random[]]
```

Here then are 500 points. (You may find it instructive to look at just one of these points.)

```
In[16]:= pts = Table[{randomcolor, randomsize, randomcoords}, {500}];
```

And here is the graphic.

```
In[17]:= Show[Graphics[pts, PlotRange → All]]
```



```
Out[17]= - Graphics -
```

5. The algebraic solution is given by the following steps. First solve the equations for x and y.

```
In[18]:= Clear[x, y, r]
```

```
In[19]:= soln = Solve[{(x - 1)^2 + (y - 1)^2 == 2, (x + 3)^2 + (y - 4)^2 == r^2}, {x, y}]
```

$$\text{Out[19]=}\ \left\{\left\{x \to \frac{1}{50}\left(-58 + 4\,r^2 - 3\sqrt{-529 + 54\,r^2 - r^4}\right),\right.\right.$$
$$\left.y \to \frac{1}{50}\left(131 - 3\,r^2 - 4\sqrt{-529 + 54\,r^2 - r^4}\right)\right\},$$
$$\left\{x \to \frac{1}{50}\left(-58 + 4\,r^2 + 3\sqrt{-529 + 54\,r^2 - r^4}\right),\right.$$
$$\left.\left.y \to \frac{1}{50}\left(131 - 3\,r^2 + 4\sqrt{-529 + 54\,r^2 - r^4}\right)\right\}\right\}$$

Then find those values of r for which the x and y coordinates are identical.

```
In[20]:= Solve[{(x /. soln[[1]]) == (x /. soln[[2]]),
           (y /. soln[[1]]) == (y /. soln[[2]])}, r]
```

$$\text{Out[20]=}\ \left\{\left\{r \to -5 - \sqrt{2}\right\}, \left\{r \to 5 - \sqrt{2}\right\}, \left\{r \to -5 + \sqrt{2}\right\}, \left\{r \to 5 + \sqrt{2}\right\}\right\}$$

Here then are those values of r that are positive.

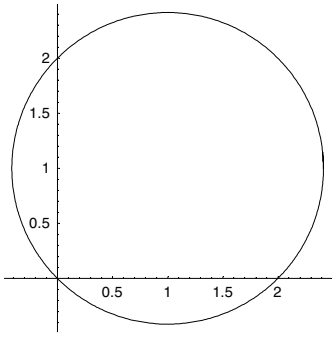
```
In[21]:= Cases[%, {r → _?Positive}]
```

$$\text{Out[21]=}\ \left\{\left\{r \to 5 - \sqrt{2}\right\}, \left\{r \to 5 + \sqrt{2}\right\}\right\}$$

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[22]:= circ = Circle[{1, 1}, Sqrt[2]];
```

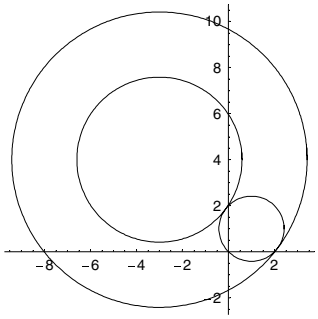
```
In[23]:= Show[Graphics[circ, Axes → Automatic, AspectRatio → Automatic]];
```



Notice that we have used the `Axes` and `AspectRatio` options because we want these commands to apply to the entire graphic.

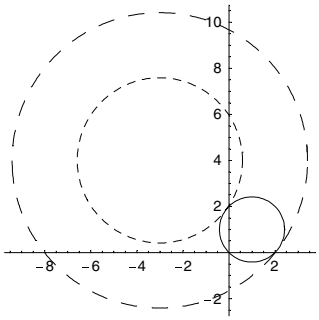
Here are the circles that represent the solution to the problem.

```
In[24]:= r1 = 5 - √2 ;
         r2 = 5 + √2 ;

In[26]:= Show[Graphics[{circ, Circle[{-3, 4}, r1], Circle[{-3, 4}, r2]},
            Axes → Automatic, AspectRatio → Automatic]];
```



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 important point to note here is that each of the circles inherits only those directives in whose scope they appear.

```
In[27]:= dashc1 = {Dashing[{.025, .025}], Circle[{-3, 4}, r1]};
         dashc2 = {Dashing[{.05, .05}], Circle[{-3, 4}, r2]};

In[29]:= Show[Graphics[{circ, dashc1, dashc2},
            Axes → Automatic, AspectRatio → Automatic]];
```



6. This loads the package containing the definitions for the polyhedra.

```
In[30]:= Needs["Graphics`Polyhedra`"]
```

It is often helpful to get a list of the functions defined in a recently loaded package.

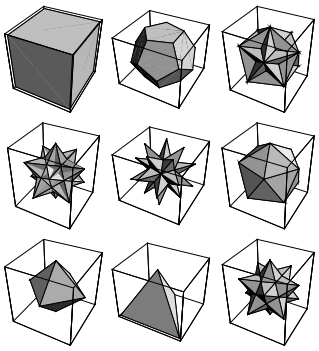
```
In[31]:= Names["Graphics`Polyhedra`*"]

Out[31]= {Geodesate, GreatDodecahedron, GreatIcosahedron,
          GreatStellatedDodecahedron, OpenTruncate, Polyhedra,
          Polyhedron, SmallStellatedDodecahedron, Stellate, Truncate}
```

First the polyhedra are turned into `Graphics3D` objects.

```
In[32]:= solids = Map[Graphics3D, {Cube[], Dodecahedron[], GreatDodecahedron[],
             GreatIcosahedron[], GreatStellatedDodecahedron[], Icosahedron[],
             Octahedron[], Tetrahedron[], SmallStellatedDodecahedron[]}];
```

We then use `Partition` to split the list of nine solids into three sublists and then display the nine polyhedra with `GraphicsArray` and `Show`.

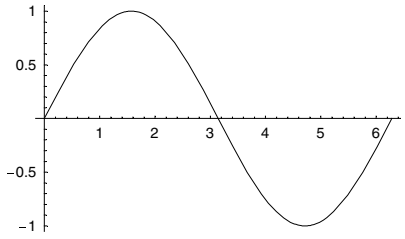
```
In[33]:= Show[GraphicsArray[Partition[solids, 3]]]
```



```
Out[33]= - GraphicsArray -
```

7. Here is a plot of the sine function.

```
In[34]:= sinplot = Plot[Sin[x], {x, 0, 2 π}]
```

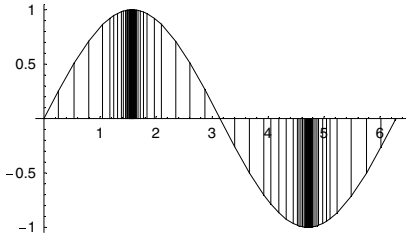



```
Out[34]= - Graphics -
```

This solution is essentially that given in *Exploring Mathematics with Mathematica* (Gray and Glynn 1991). Extracting the points from which *Mathematica* constructs the plot is accomplished by the `Nest` statement. The `Line` primitive is then mapped across those points in

such a way as to create lines from the points on the x-axis with the same x-coordinate.

```
In[35]:= Show[sinplot,
           Graphics[
               {Thickness[.001],
               Map[Line[{{#[[1]], 0}, #}]&,
                 Nest[First, sinplot, 4]]}]]
```




```
Out[35]= - Graphics -
```

You could also construct this using pattern matching. Here are the coordinates.

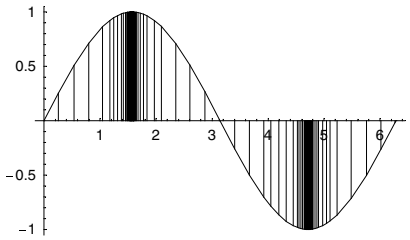
```
In[36]:= (coords = Cases[sinplot, {p_?NumericQ, q_?NumericQ}, Infinity]) // Short

Out[36]//Short=
{{2.61799×10^-7, 2.61799×10^-7}, <<80>>, {6.28319, -<<23>>}}
```

Here is what we use to create vertical lines from each coordinate.

```
In[37]:= (lines = Map[Line[{{#[[1]], 0}, #}] &, coords]) // Short

Out[37]//Short=
{Line[{{2.61799×10^-7, 0}, {2.61799×10^-7, <<23>>}}], <<80>>, Line[<<1>>]}
```

Here then is the final graphic.

```
In[38]:= Show[sinplot, Graphics[
           Map[Line[{{#[[1]], 0}, #}] &,
            Cases[sinplot, {p_?NumericQ, q_?NumericQ}, Infinity]]
          ]];
```



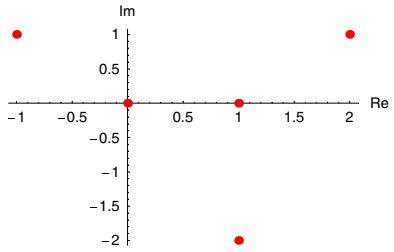
9.2 Graphics programming

1. The function `ComplexListPlot` plots a list of complex numbers in the complex plane, with the real part identified with the horizontal axis and the imaginary part identified with the vertical axis. The appropriate options are extracted from `ComplexListPlot` using `FilterOptions`, given the name `complexOpts`, and then passed to `ListPlot`.

```
In[1]:= << Utilities`FilterOptions`

In[2]:= ComplexListPlot[points_, opts___] :=
         Module[{complexOpts = FilterOptions[ListPlot, opts]},
          ListPlot[Map[{Re[#1], Im[#1]} &, points], complexOpts, PlotStyle →
            {RGBColor[1, 0, 0], PointSize[.025]}, AxesLabel → {"Re", "Im"}]]
```

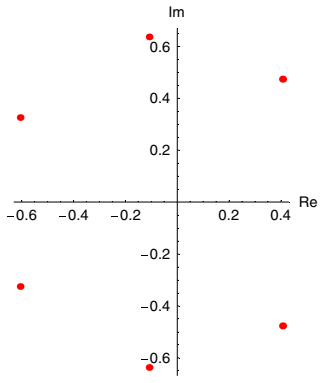
This plots four complex numbers in the plane.

```
In[3]:= ComplexListPlot[{-1 + I, 2 + I, 1 - 2 I, 0, 1}];
```



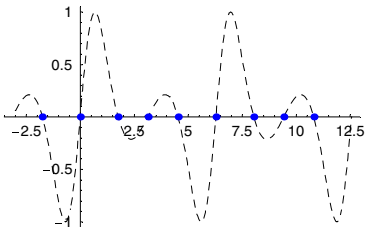
2. The function `RootPlot`, takes a polynomial, solves for its roots, and then uses `ComplexListPlot` from Exercise 1 to plot these roots in the complex plane.

```
In[4]:= RootPlot[poly_, z_, opts___] :=
         ComplexListPlot[z /. NSolve[poly == 0, z], opts]

In[5]:= RootPlot[1 + z + 2 z^2 + 3 z^3 + 5 z^4 + 8 z^5 + 13 z^6, z, AspectRatio → Automatic];
```



3.

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

In[7]:= << Utilities`FilterOptions`

In[8]:= RootPlot[fun_, {x_, xmin_, xmax_}, opts___] := Module[
          {z, fplot, pts, spts, roots, points, f = Function[x, Evaluate[fun]]},
          fplot = Plot[f[x], {x, xmin, xmax}, DisplayFunction → Identity,
            Evaluate[FilterOptions[Plot, opts]]];
          pts = Cases[fplot, Line[{z__}] → z, ∞];
          spts = Map[First, Select[Split[pts, Sign[Last[#2]] == -Sign[Last[#1]] &],
             Length[#1] == 2 &], {2}];
          roots = Map[FindRoot[f[x] == 0, {x, #[[1]], #[[2]]}] &, spts];
          points = Map[Point[{#, 0}] &, x /. roots];
          Show[fplot, DisplayFunction → $DisplayFunction,
           Epilog → {RGBColor[0, 0, 1], PointSize[.02], points}];
          roots]

In[9]:= RootPlot[Sin[x + √2 Sin[x]], {x, -π, 4 π}, PlotStyle → Dashing[{.02, .02}]];
```



4. Here is the new code for `DataPlot`.

```
In[10]:= Clear[DataPlot]

In[11]:= Needs["Utilities`FilterOptions`"]

In[12]:= Options[DataPlot] = Options[ListPlot];

In[13]:= DataPlot::baddim = "The data used by DataPlot must
             be in the form of a one- or two-dimensional list.";

In[14]:= DataPlot[data_, opts___] := Module[{pjQ, pts, lines},
           pjQ = PlotJoined /. Flatten[{opts, Options[DataPlot]}];
           pts = Which[
             VectorQ[data], MapIndexed[{#2[[1]], #1} &, data],
             Dimensions[data][[2]] == 2, data,
             True, Message[DataPlot::baddim]; $Failed];
           If[pts =!= $Failed,
            Show[Graphics[{PointSize[.02],
               Point /@ pts, If[pjQ, lines = Line[pts], lines = {}]}],
             FilterOptions[Graphics, opts], Axes → Automatic]]]
```

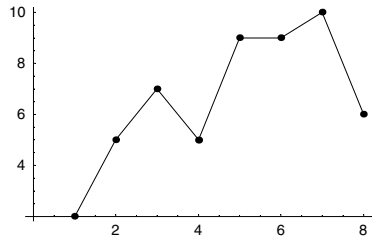
Here is some sample two-dimensional data.

```
In[15]:= data2D = {{0.043, 0.575}, {0.151, 0.120},
            {0.234, 0.001}, {0.283, 0.930}, {0.343, 0.569}, {0.416, 0.768},
            {0.465, 0.675}, {0.539, 0.528}, {0.786, 0.856}, {0.914, 0.794}};
```

And here is some sample one-dimensional data.

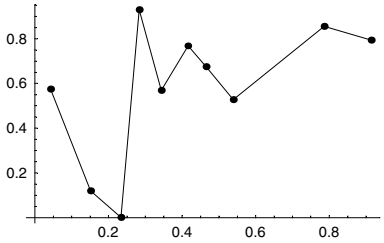
```
In[16]:= data1D = Table[Random[Integer, {1, 10}], {8}]

Out[16]= {2, 5, 7, 5, 9, 9, 10, 6}
```

```
In[17]:= DataPlot[data1D, PlotJoined → True]
```




```
Out[17]= - Graphics -
```

```
In[18]:= DataPlot[data2D, PlotJoined → True]
```



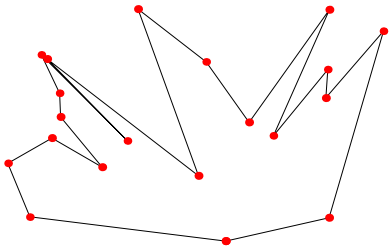

```
Out[18]= - Graphics -
```

Here is some data that `DataPlot` is not designed to deal with.

```
In[19]:= DataPlot[{{1, 2, 3}, {2, 3, 4}, {4, 5, 6}}]

DataPlot::baddim : The data used by DataPlot
   must be in the form of a one- or two-dimensional list.
```

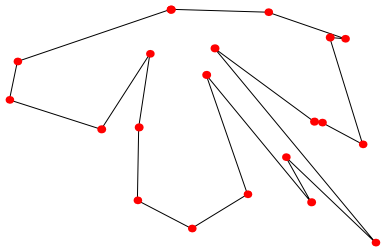
5. There are a number of things that could go wrong with the algorithm by just choosing a base point randomly and then sorting according to the arctangent. The default branch cut for `ArcTan` gives values between $-\pi/2$ and $\pi/2$. (The reader is encouraged to think about why this could occasionally 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[20]:= simpleClosedPath1[lis_List] := Module[{base, angle, sorted},
           base = Last[Sort[lis, (#2[[2]] < #1[[2]]) &]];
           angle[a_, b_] := Apply[ArcTan, b - a];
           sorted =
            Sort[Complement[lis, {base}], (angle[base, #1] ≤ angle[base, #2]) &];
           Join[{base}, sorted, {base}]
          ]

In[21]:= pts = Table[Random[], {20}, {2}];

In[22]:= PointPlot[coords_List] :=
          Show[Graphics[{
             Line[coords],
             PointSize[.02], RGBColor[1, 0, 0], Map[Point, coords]
            }]]

In[23]:= PointPlot[simpleClosedPath1[pts]];
```



7. A simple change to the program `simpleClosedPath` given in Exercise 5 chooses the base point with the largest y-coordinate.

```
In[24]:= simpleClosedPath3[lis_] :=
          Module[{base, angle, sorted}, base = Last[Sort[lis, #2[[2]] > #1[[2]] &]];
           angle[a_, b_] := ArcTan@@ (b - a); sorted = Sort[Complement[lis, {base}],
             angle[base, #1] ≤ angle[base, #2] &]; Join[{base}, sorted, {base}]]

In[25]:= pts = Table[Random[], {20}, {2}];

In[26]:= PointPlot[simpleClosedPath3[pts]];
```



8. 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. Since the vectors we are working with are in two-dimensional space, we embed them in three-dimensional space in the plane $z = 0$ so that we can compute the cross product which, for the purposes of this problem, only makes sense in three dimensions.

```
In[27]:= << "Calculus`VectorAnalysis`"

In[28]:= CrossProduct[{x2, y2} - {x1, y1}, {x3, y3} - {x1, y1}] /. {x_, y_} → {x, y, 0}

Out[28]= {0, 0, -x2 y1 + x3 y1 + x1 y2 - x3 y2 - x1 y3 + x2 y3}
```

Here are the coordinates for a triangle.

```
In[29]:= a = {0, 0};
         b = {5, 0};
         c = {3, 2};
```

And here is the computation for the cross product.

```
In[32]:= CrossProduct[b - a, c - a] /. {x_, y_} → {x, y, 0}

Out[32]= {0, 0, 10}
```

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

```
In[33]:= Apply[Plus, %]
         ──────────────
                2

Out[33]= 5
```

Here is a function that computes the area of any triangle using the cross product.

```
In[34]:= triangleArea[v_List] := 1/2 Apply[Plus,
            (CrossProduct[v[[2]] - v[[1]], v[[3]] - v[[1]]] /. {x_, y_} → {x, y, 0})]
```

This is done more simply using determinants and this method generalizes more easily to higher dimensions.

```
In[35]:= triangleArea[{v1_, v2_, v3_}] := 1/2 Det[{v1, v2, v3} /. {x_, y_} → {x, y, 1}]

In[36]:= triangleArea[{a, b, c}]

Out[36]= 5
```

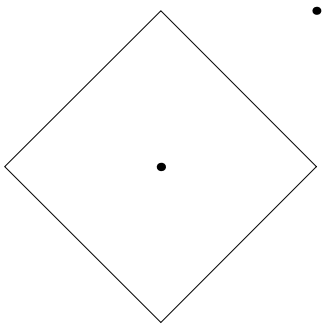
9. The key observation is that in computing the area of a triangle using the determinant formulation as in Exercise 9, the area will be a positive quantity if the points are given in counter-clockwise order, and will be negative if in clockwise order. So, for a given point p not on a line ab, the area of $\triangle abp$ will be positive (computed using determinants), if p is to the left of ab. Similarly, for each of the lines in a polygon, relative to the given point p. So, to perform the computation, we first partition the polygon into pairs of points, and then map the triangle area

function with the given point across each pair. If all such areas are greater than or equal to zero, then a value of `True` is returned.

```
In[37]:= pointInPolygonQ[poly_, p_] := Module[{area},
           area[{v1_, v2_, v3_}] := 1/2 Det[{v1, v2, v3} /. {x_, y_} → {x, y, 1}] ≥ 0;
           Apply[And, Map[area[Join[{p}, #]] &,
             Partition[poly /. {a_, b__} :→ {a, b, a}, 2, 1]]]]
```

Here are the coordinates for a quadrilateral and two distinct points.

```
In[38]:= quad = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

In[39]:= p1 = {0, 0};
         p2 = {1, 1};

In[41]:= Show[Graphics[{Line[quad /. {a_, b__} :→ {a, b, a}],
             PointSize[.025], Point[p1], Point[p2]}, AspectRatio → Automatic]];
```



Finally, here are the computations for these points and polygon.

```
In[42]:= pointInPolygonQ[quad, p1]

Out[42]= True

In[43]:= pointInPolygonQ[quad, p2]

Out[43]= False
```

12. `RT` (for Reingold–Tilford), replaces the `placeTree` function. In `placeTree`, the result was a separation tree plus two numbers: width of the left side of the tree and width of the right side of the tree. In `RT`, the result is instead a separation tree plus two lists, the first giving the width of the left side of each level of the tree, the second giving the corresponding widths on the right side. `sep` is calculated by adding the right widths of the left subtree to the left widths of the right subtree at each level, and taking the maximum separation. `drawSepTree` is unchanged.

```
In[44]:= << IPM3`Trees`
```

```
In[45]:= RT[{_}] := {{}, {}, {}}

In[46]:= RT[{_, lc_, rc_}] :=
          With[{left = RT[lc], right = RT[rc], minsep = 2.0}, With[{sep =
               1/2 (Max[0, Max @@ (Plus @@ #1 &) /@ zip[left[[3]], right[[2]]]] + minsep)},
            With[{newtree = {sep, left[[1]], right[[1]]},
              leftedge = Join[{sep}, extend[left[[2]], right[[2]], sep]],
              rightedge = Join[{sep}, extend[right[[3]], left[[3]], sep]]},
             {newtree, leftedge, rightedge}]]]

In[47]:= placeTree[{_}] := {{}, 0, 0}
         placeTree[{_, lc_, rc_}] :=
          Module[{left = placeTree[lc], right = placeTree[rc], minsep = 1.0, sep},
           sep = left[[3]] + right[[2]] + minsep;
           {{sep, left[[1]], right[[1]]}, left[[2]] + sep/2, right[[3]] + sep/2}]

In[49]:= drawSepTree[{}, lev_, xaxis_] := {Disk[{xaxis, lev}, 0.1]}
         drawSepTree[{sep_, lc_, rc_}, lev_, xaxis_] :=
          Join[{Disk[{xaxis, lev}, 0.1], Line[{{xaxis, lev}, {xaxis - sep, lev - 1}}],
            Line[{{xaxis, lev}, {xaxis + sep, lev - 1}}]},
           drawSepTree[lc, lev - 1, xaxis - sep], drawSepTree[rc, lev - 1, xaxis + sep]]

In[51]:= drawTree[t_] := drawSepTree[RT[t][[1]], 0, 0]
```

The auxiliary functions are `zip` and `extend`. Given the *left* widths of each level of the *right* subtree, and the *right* widths of each level of the *left* subtree, the separation of the two subtrees is determined by adding those numbers at each level and taking the maximum. `zip` is used to join those two lists into a list of pairs; it facilitates this process.

```
In[52]:= zip[{}, _] := {}
         zip[_, {}] := {}
         zip[{x1_, y1___}, {x2_, y2___}] := Join[{{x1, x2}}, zip[{y1}, {y2}]]
```

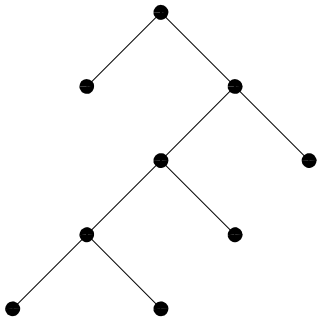
When the separation of a tree's subtrees is determined, the lists of left and right widths of the combined tree are computed from the corresponding lists for the subtrees. This is simple enough for the most part: the left widths of the tree are obtained mainly by taking the left widths of the left subtree and shifting them left; and similarly for the right widths. There is an exception, though: if the right subtree is taller than the left subtree, the left widths of the bottom part of the tree are obtained from the left widths of the bottom part of the right subtree. Combining the left widths of the two subtrees to create the list of left widths of the combined tree is done by `extend`.

```
In[55]:= extend[edges1_, edges2_, sep_] := Join[edges1 + sep,
           Take[edges2, Min[0, Length[edges1] - Length[edges2]]] - sep]
```

The same reasoning applies to computing the right widths, and `extend` is also used for that.

Here is the "tricky" tree drawn in Figure 9.5.

```
In[56]:= Clear[a, b, c, d, e, f, g]

In[57]:= t1 = {a, {b}, {a, {c, {e, {g}, {f}}, {d}}, {b}}};

In[58]:= Show[Graphics[drawTree[t1]], AspectRatio → Automatic];
```



13. `RT` is modified so that the left widths and right widths of each row take into account the width of the labels.

```
In[59]:= RT[{x_}] := {{}, {width[x]/2}, {width[x]/2}}
         RT[{x_, lc_, rc_}] :=
          With[{left = RT[lc], right = RT[rc], minsep = 0.5}, With[{sep =
               1/2 (Max[0, Max @@ (Plus @@ #1 &) /@ zip[left[[3]], right[[2]]] + minsep)},
            With[{newtree = {sep, left[[1]], right[[1]]},
              leftedge = Join[{width[x]/2}, extend[left[[2]], right[[2]], sep]],
              rightedge = Join[{width[x]/2}, extend[right[[3]], left[[3]], sep]]},
             {newtree, leftedge, rightedge}]]]
         width[t_] := StringLength[t]
```

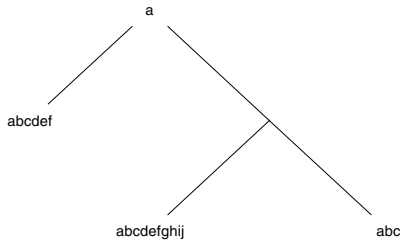
Drawing the following tree using the new `RT` and the old `drawSepTree` will show the difference in the layout of the trees. However, since `drawSepTree` above only prints disks at each node, a new version of it is required.

```
In[62]:= t1 = {"a", {"abcdef"}, {"", {"abcdefghij"}, {"abc"}}};
```

The new version of `drawSepTree` draws the labels at each node instead of a disk. A complicating factor is that we can no longer just draw the lines from the center of the disk, since this would collide with the text. So the lines are now drawn in such a way as to leave a gap between the text and the line.

```
In[63]:= settext[lab_] := FontForm[lab, {"Helvetica", 9}]
```

```
In[64]:= drawSepTree[{lab_}, {}, lev_, xaxis_] := {Text[settext[lab], {xaxis, lev}]}

In[65]:= drawSepTree[{lab_, lc_, rc_}, {sep_, ls_, rs_}, lev_, xaxis_] :=
          With[{h1 = If[lab == "", 0, .3], h2 = If[lc[[1]] == "", 0, .3],
            h3 = If[rc[[1]] == "", 0, .3]}, Join[{Text[settext[lab], {xaxis, lev}],
             Line[{{xaxis - sep h1/2, lev - h1/2}, {xaxis - sep + sep h2/2, lev - 1 + h2/2}}],
             Line[{{xaxis + sep h1/2, lev - h1/2},
               {xaxis + sep - sep h3/2, lev - 1 + h3/2}}]}, drawSepTree[lc, ls,
             lev - 1, xaxis - sep], drawSepTree[rc, rs, lev - 1, xaxis + sep]]]

In[66]:= drawTree[t_] := drawSepTree[t, RT[t][[1]], 0, 0]

In[67]:= Show[Graphics[drawTree[t1]], PlotRange → All];
```



9.3 Sound

1. When x is close to -2, the frequency is quite low. As x increases, the fraction $1000/x$ gets larger, 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 effectively compute amplitudes and, hence, we hear noise in this region.

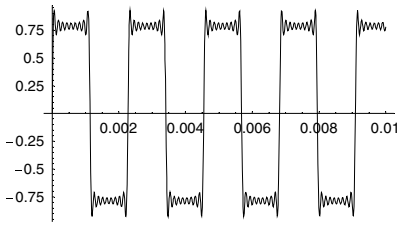
```
In[1]:= Play[Sin[1000/x], {x, -2, 2}]

        Power::infy : Infinite expression 1/0. encountered. More…

Out[1]= - Sound -
```

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, so here is the command to produce the tone. You need to carefully choose a range for t that generates tones in a reasonable range.

```
In[2]:= Play[Sin[2^t], {t, 10, 14}]

Out[2]= - Sound -
```

5. Here is a function that creates a square wave with decreasing amplitudes for higher overtones.

```
In[3]:= SquareWave[freq_, n_] := Sum[Sin[freq i 2 π t]/i, {i, 1, n, 2}]

In[4]:= Plot[SquareWave[440, 17], {t, 0, .01}];
```



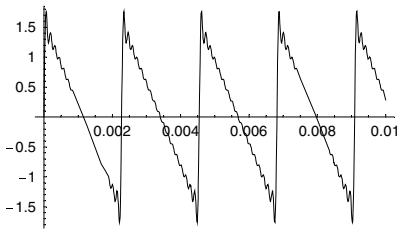

```
In[5]:= squarePlay[freq_, n_, time_] :=
          Play[Sum[Sin[freq i 2 π t]/i, {i, 1, n, 2}], {t, 0, time}]
```

Here then, is an example of playing a square wave.

```
In[6]:= Play[SquareWave[440, 17], {t, 0, .5}]

Out[6]= - Sound -
```

7. This function creates a saw-tooth wave. The user specifies the fundamental frequency and the number of terms in the approximation.

```
In[7]:= SawtoothWave[freq_, n_] := Sum[Sin[freq i 2 π t]/i, {i, 1, n}]

In[8]:= Plot[SawtoothWave[440, 17], {t, 0, .01}];
```



This plays the wave for a half-second duration.

```
In[9]:= Play[SawtoothWave[440, 17], {t, 0, .5}]
```

```
Out[9]= - Sound -
```

Here are definitions for true sawtooth and square waves.

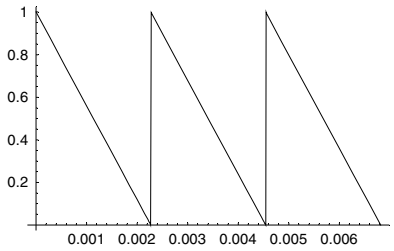
```
In[10]:= Fractional[x_] := x - Floor[x]
```

```
In[11]:= SawtoothWave[x_]:= Fractional[-x]
```

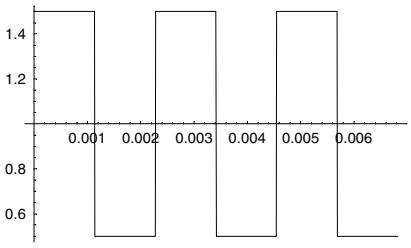
```
In[12]:= SquareWave[x_] := 1/2 Sign[SawtoothWave[x] - 1/2] + 1
```

Here are plots at the fundamental frequency of 440.

```
In[13]:= Plot[SawtoothWave[440 t], {t, 0, 3/440}];
```




```
In[14]:= Plot[SquareWave[440 t], {t, 0, 3/440}];
```



9. Here is a function that picks out frequencies from the pentatonic scale, using essentially brownian motion $1/f^2$ to select notes.

```
In[15]:= pentatonic[n_Integer, r_ : 2] := Module[{pscale, steps},
           pscale = {277.183, 311.13, 369.99, 415.30, 466.16, 554.37};
           steps = Table[Random[Integer, {-r, r}], {n}];
           pscale[[Mod[FoldList[Plus, 3, steps], 4] + 1]]]
```

You could play a pentatonic "melody" as follows:

```
In[16]:= SetAttributes[PlayTones, Listable]
```

```
In[17]:= PlayTones[freq_, time_: 0.5] := Play[Sin[2 π t freq], {t, 0, time}]

In[18]:= PlayTones[pentatonic[24]]

Out[18]= {- Sound -, - Sound -, - Sound -, - Sound -, - Sound -, - Sound -, - Sound -,
           - Sound -, - Sound -, - Sound -, - Sound -, - Sound -, - Sound -,
           - Sound -, - Sound -, - Sound -, - Sound -, - Sound -, - Sound -,
           - Sound -, - Sound -, - Sound -, - Sound -, - Sound -, - Sound -}
```

11. In this function, the notes are randomly chosen from the C major scale $1/f^0$ and the durations are randomly chosen from the list that represents eighth notes, quarter notes, half notes, and whole notes (also $1/f^0$). `PlayTones` accepts two arguments, so `MapThread` threads corresponding notes and durations through `PlayTones`.

```
In[19]:= tonesAndTimes[n_] := Module[{cmajor, notes, durs},
           cmajor = Table[N[261.62558 2^(j/12)], {j, 0, 11}];
           notes := Table[cmajor[[Random[Integer, {1, 12}]]], {n}];
           durs := Table[1/2^Random[Integer, {0,3}], {n}];
           MapThread[PlayTones, {notes, durs}]]
```

13. Following the implementation in the text, we first create ten steps between −2 and 2 (you can alter the range of step movements). These steps will determine how to move up or down the list of tone durations (1/8, 1/4, 1/2, 1).

```
In[20]:= d10 = Table[Random[Integer, {-2, 2}], {10}]

Out[20]= {0, 0, -2, -1, 2, 1, 2, 2, -2, -2}

In[21]:= Mod[FoldList[Plus, 0, d10], 4] + 1

Out[21]= {1, 1, 1, 3, 2, 4, 1, 3, 1, 3, 1}

In[22]:= durs = {1/8, 1/4, 1/2, 1}[[%]]

Out[22]= {1/8, 1/8, 1/8, 1/2, 1/4, 1, 1/8, 1/2, 1/8, 1/2, 1/8}
```

Here are some $1/f^2$ tones.

```
In[23]:= s10 = Table[Random[Integer, {-2, 2}], {10}]

Out[23]= {-2, -2, -1, -2, -1, 1, -2, 2, -1, 1}

In[24]:= pos = Mod[FoldList[Plus, 0, s10], 13] + 1

Out[24]= {1, 12, 10, 9, 7, 6, 7, 5, 7, 6, 7}
```

```
In[25]:= Cmajor = Table[N[261.62558 2^(j/12)], {j, 0, 12}]

Out[25]= {261.626, 277.183, 293.665, 311.127, 329.628, 349.228,
          369.994, 391.995, 415.305, 440., 466.164, 493.883, 523.251}

In[26]:= Length[Cmajor]

Out[26]= 13

In[27]:= tones = Cmajor[[pos]]

Out[27]= {261.626, 493.883, 440., 415.305, 369.994,
          349.228, 369.994, 329.628, 369.994, 349.228, 369.994}

In[28]:= MapThread[PlayTones, {tones, durs}];
```

And finally, here is one function that puts this all together.

```
In[29]:= tonesAndTimes2[n_] := Module[{cmajor, tones, durs, d, t},
           cmajor = Table[N[261.62558 2^(j/12)], {j, 0, 12}];
           d = Table[Random[Integer, {-2, 2}], {n}];
           durs = {1/8, 1/4, 1/2, 1}[[Mod[FoldList[Plus, 0, d], 4] + 1]];
           t = Table[Random[Integer, {-2, 2}], {n}];
           tones = cmajor[[Mod[FoldList[Plus, 0, t], 13] + 1]];
           MapThread[PlayTones, {tones, durs}]]

In[30]:= tonesAndTimes2[12];
```

10 Front end programming

10.2 The structure of cells and notebooks

1. Here is the expression to create the notebook.

```
In[1]:= nb = NotebookPut[
            Notebook[{
              Cell["Demo Notebook", "Title"],
              Cell["Section 1: Sample Cells", "Section"],
              Cell["This is a text cell", "Text"],
              Cell["2(3+5)", "Input"],
              Cell["1+2+3", "Input"]}]
           ]

Out[1]= NotebookObject[ <<Untitled-3>> ]
```

2. First we read the notebook from Exercise 1 into the kernel with `NotebookGet`.

```
In[2]:= nbkernel = NotebookGet[nb]

Out[2]= Notebook[
         {Cell[CellGroupData[{Cell[Demo Notebook, Title], Cell[CellGroupData[
               {Cell[Section 1: Sample Cells, Section], Cell[This is a text cell,
                 Text], Cell[2(3+5), Input], Cell[1+2+3, Input]}, Open]]},
            Open]]}, FrontEndVersion→5.1 for Microsoft Windows,
         ScreenRectangle→{{0., 1024.}, {0., 681.}}]
```

Then we do a substitution on cells that contain `"Section"` as their second argument (their style) and finally use `NotebookPut` to display the resulting notebook in the front end.

```
In[3]:= NotebookPut[nbkernel /. Cell[str_, "Section"] → Cell[str, "Subsection"]]

Out[3]= NotebookObject[ <<Untitled-4>> ]
```

10.3 Cell data types

1. Here is the notebook object with three `ValueBox`es.

```
In[1]:= nb = NotebookPut[
           Notebook[{
             Cell[TextData[
               {"The current version is ", ValueBox["$Version"]}], "Text"],
             Cell[TextData[{"The operating system is ",
                ValueBox["$OperatingSystem"]}], "Text"],
             Cell[TextData[{"Current user is ", ValueBox["$UserName"]}], "Text"]
            }]]

Out[1]= NotebookObject[ <<Untitled-5>> ]
```

10.4 GridBoxes

1. There are several ways of approaching this problem. One way is to create a function that contains the formatting rules for the heading.

```
In[1]:= headstyle[str_] :=
          StyleBox[(MakeBoxes[#, StandardForm] &)[str], FontFamily→"Helvetica",
           FontWeight→"Bold", FontColor→RGBColor[0, 0, 1], FontSize→10];
```

Here are some sample strings for the heading.

```
In[2]:= headings = {"first", "second", "third"};

In[3]:= data = {{"α", "β", "γ"},
          {1.234, 2.3451, 3.4567801}, {SqrtBox["π"], "x/y", "Γ(n)"}};
```

We now need to create a list of the headings together with their styles and prepend it to the original data. This way the headings will be the first row of the new data set.

```
In[4]:= Prepend[data, Map[headstyle, headings]]

Out[4]= {{StyleBox["first", FontFamily → Helvetica,
     FontWeight → Bold, FontColor → RGBColor[0, 0, 1], FontSize → 10],
    StyleBox["second", FontFamily → Helvetica, FontWeight → Bold,
     FontColor → RGBColor[0, 0, 1], FontSize → 10],
    StyleBox["third", FontFamily → Helvetica, FontWeight → Bold,
     FontColor → RGBColor[0, 0, 1], FontSize → 10]},
   {α, β, γ}, {1.234, 2.3451, 3.45678}, {SqrtBox[π], x/y, Γ(n)}}
```

```
In[5]:= ShowTable[data_, headings_List] := DisplayForm[StyleBox[
          GridBox[Prepend[data, Map[headstyle, headings]],
           GridFrame → 2, GridFrameMargins → {{1, 1}, {1, 1}},
           RowLines → 1, ColumnLines → 1],
          FontFamily → "Times",
          Background → GrayLevel[.8], SingleLetterItalics → True
         ]]
```

```
In[6]:= ShowTable[data, {"first", "second", "third"}]
```

Out[6]//DisplayForm=

first	second	third
α	β	γ
1.234	2.3451	3.4567801
$\sqrt{\pi}$	$\frac{x}{y}$	$\Gamma(n)$

A cleaner approach would be to set up the headings as an option to `ShowTable`. In addition, the header formatting should be incorporated into `ShowTable`. Here is one approach.

```
In[7]:= Options[ShowTable] = {Headings → {}};
```

```
In[8]:= ShowTable[data_, opts___?OptionQ] := Module[{headstyle, headings},
          headstyle[str_] :=
           StyleBox[(MakeBoxes[#, StandardForm] &)@str, FontFamily -> "Helvetica",
            FontWeight → "Bold", FontColor → RGBColor[0, 0, 1], FontSize → 10];
          headings = Headings /. Flatten[{opts}] /. Options[ShowTable];
          DisplayForm[StyleBox[
            GridBox[Prepend[data, Map[headstyle, headings]],
             GridFrame → 2, GridFrameMargins → {{1, 1}, {1, 1}},
             RowLines → 1, ColumnLines → 1],
            FontFamily → "Times",
            Background → GrayLevel[.8], SingleLetterItalics → True
           ]]]
```

```
In[9]:= ShowTable[data, Headings → {"premier", "deuxieme", "troisieme"}]
```

Out[9]//DisplayForm=

premier	deuxieme	troisieme
α	β	γ
1.234	2.3451	3.4567801
$\sqrt{\pi}$	$\frac{x}{y}$	$\Gamma(n)$

3. First, here is the table of all possible truth values for three variables. We will generalize this below.

```
In[10]:= ins = Distribute[Table[{True, False}, {3}], List, List, List]

Out[10]= {{True, True, True}, {True, True, False},
          {True, False, True}, {True, False, False}, {False, True, True},
          {False, True, False}, {False, False, True}, {False, False, False}}
```

Here is the logical expression.

```
In[11]:= expr = Implies[Or[A, B], C]

Out[11]= Implies[A || B, C]
```

This creates a set of rules for all possible truth value combinations.

```
In[12]:= vars = {A, B, C};
         Map[Thread[vars → #] &, ins]

Out[13]= {{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}}
```

And here we substitute these rules into the logical expression we are working with.

```
In[14]:= expr /. %

Out[14]= {True, False, True, False, True, False, True, True}
```

Here then is the `TruthTable` function.

```
In[15]:= TruthTable[expr_, vars_] := Module[{len = Length[vars], n},
           ins = Distribute[Table[{True, False}, {len}], List, List, List];
           res = (expr /. Thread[vars → #1] &) /@ ins;
           DisplayForm[GridBox[Prepend[Transpose[Append[Transpose[ins],
                 (If[! MemberQ[{True, False}, #1], "*", #1] &) /@ res]] /.
               {True → "T", False → "F"}, Append[vars,
               TraditionalForm[expr]]], GridFrame → True,
             RowLines → Prepend[Table[0, {Length[res] - 1}], 2],
             ColumnLines → Append[Table[0, {Length[vars] - 1}], 2]]]]
```

In[16]:=
```
TruthTable[Implies[A || B, C], {A, B, C}]
```

Out[16]//DisplayForm=

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

10.5 Buttons

1. Here is the code for the `Plot3D` template button.

In[1]:=
```
ButtonBox["Plot3D[fun,{x,xmin,xmax},{y,ymin,ymax}]", Active → True] //
 DisplayForm
```

Out[1]//DisplayForm=
```
Plot3D[fun, {x, xmin, xmax}, {y, ymin, ymax}]
```

Alternately, you can use placeholders.

In[2]:=
```
ButtonBox["Plot3D[□,{□,□,□},{□,□,□}]", Active → True] // DisplayForm
```

Out[2]//DisplayForm=
```
Plot3D[□, {□, □, □}, {□, □, □}]
```

2. Here is the code to create the `Expand` button.

In[3]:=
```
ButtonBox["Expand[■]", Active → True,
  ButtonStyle → "CopyEvaluateCell"] // DisplayForm
```

Out[3]//DisplayForm=
```
Expand[■]
```

Selecting the expression below and then clicking the `Expand` button will cause a new input cell to be created with `Expand` wrapped around the selected expression; then that cell will be evaluated to produce the expanded polynomial below.

In[4]:=
```
(α + β - γ)^5
```

In[5]:=
```
Expand[(α + β - γ)^5]
```

Out[5]=
$$\alpha^5 + 5\,\alpha^4\,\beta + 10\,\alpha^3\,\beta^2 + 10\,\alpha^2\,\beta^3 + 5\,\alpha\,\beta^4 + \beta^5 - 5\,\alpha^4\,\gamma - 20\,\alpha^3\,\beta\,\gamma - 30\,\alpha^2\,\beta^2\,\gamma - 20\,\alpha\,\beta^3\,\gamma - 5\,\beta^4\,\gamma + 10\,\alpha^3\,\gamma^2 + 30\,\alpha^2\,\beta\,\gamma^2 + 30\,\alpha\,\beta^2\,\gamma^2 + 10\,\beta^3\,\gamma^2 - 10\,\alpha^2\,\gamma^3 - 20\,\alpha\,\beta\,\gamma^3 - 10\,\beta^2\,\gamma^3 + 5\,\alpha\,\gamma^4 + 5\,\beta\,\gamma^4 - \gamma^5$$

3. Here is the code for the palette.

```
Cell[BoxData[GridBox[{
        {
          ButtonBox[
            RowBox[{"Expand", "[", "■", "]"}],
            ButtonStyle->"CopyEvaluateCell",
            Active->True, ButtonEvaluator->Automatic]},
        {
          ButtonBox[
            RowBox[{"Factor", "[", "■", "]"}],
            ButtonStyle->"CopyEvaluateCell",
            Active->True, ButtonEvaluator->Automatic]},
        {
          ButtonBox[
            RowBox[{"Apart", "[", "■", "]"}],
            ButtonStyle->"CopyEvaluateCell",
            Active->True, ButtonEvaluator->Automatic]},
        {
          ButtonBox[
            RowBox[{"Together", "[", "■", "]"}],
            ButtonStyle->"CopyEvaluateCell",
            Active->True, ButtonEvaluator->Automatic]}
        },
      RowSpacings->0,
      ColumnSpacings->0]], "Input"]
```

Here is how the palette looks when the above expression is formatted.

In[6]:=

Expand[■]
Factor[■]
Apart[■]
Together[■]

If you wanted to turn this into a free-standing palette, select the above cell and choose Generate Palette from Selection from the File menu.

Notice that in the code for the palette, each of the buttons used the same three options, `ButtonStyle`, `Active`, and `ButtonEvaluator`, with identical values. Using `ButtonBox`'s `Options`, we can set each value once at the `GridBox` level and each of the buttons will inherit the option, This cleans up the code considerably.

```
Cell[BoxData[GridBox[{
        {
          ButtonBox[
            RowBox[{"Expand", "[", "■", "]"}]]},
        {
          ButtonBox[
            RowBox[{"Factor", "[", "■", "]"}]]},
        {
          ButtonBox[
            RowBox[{"Apart", "[", "■", "]"}]]},
        {
          ButtonBox[
            RowBox[{"Together", "[", "■", "]"}]]}
        },
      RowSpacings->0,
      ColumnSpacings->0]], "Input",
      ButtonBoxOptions->{ButtonStyle->"CopyEvaluateCell",
            Active->True, ButtonEvaluator->Automatic}
    ]
```

Finally, here is the formatted palette with an input cell and the result of selecting that input cell and clicking the `Together[■]` button.

In[7]:=

Expand[■]
Factor[■]
Apart[■]
Together[■]

In[8]:=

$$\frac{1}{x} + \frac{1}{y}$$

In[9]:=

$$\text{Together}\left[\frac{1}{x} + \frac{1}{y}\right]$$

Out[9]=

$$\frac{x + y}{x\,y}$$

11 Examples and applications

11.1 Manipulating data files

1. We first borrow the options from `ReadList` that we wish to pass into `ReadSolarData`.

```
In[1]:= Options[ReadSolarData] = {
          WordSeparators → ",",
          RecordLists -> True,
          RecordSeparators → {"\r\n", "\n", "\r"}
         };

In[2]:= ReadSolarData[file_, opts___?OptionQ] := Module[{ws, rl, rs, raw, data},
          {ws, rl, rs} = {WordSeparators, RecordLists, RecordSeparators} /.
             Flatten[{opts}] /. Options[ReadSolarData];
          raw = ReadList[file, Word, RecordLists → rl,
            RecordSeparators → rs, WordSeparators → ws];
          data = Select[raw, StringTake[#[[1]], 1] =!= "\"" &];
        Map[ToExpression, data, {2}]
         ]

In[3]:= datafile = ToFileName[{"IPM3", "DataFiles"}, "23232.txt"]

Out[3]= IPM3\DataFiles\23232.txt

In[4]:= data = ReadSolarData[datafile];
```

Now we can use the `GetData` function developed in Section 11.1 to select those records that fall between certain dates.

```
In[5]:= GetData[dat_, {m1_, y1_}, {m2_, y2_}] :=
         Select[dat, (#[[1]] == y1 && #[[2]] ≥ m1) || (#[[1]] == y2 && #[[2]] ≤ m2) &]
```

For example, here are the records between November 1976 and March 1977.

```
In[6]:= GetData[data, {11, 76}, {3, 77}]

Out[6]= {{76, 11, 2.5, 3.5, 3.9, 4.1, 3.6, 3.5, 4.3,
           4.6, 4.8, 4.8, 2.7, 2.2, 3.1, 3.2}, {76, 12, 2.3, 3.7, 4.4,
           4.7, 4.4, 3.5, 4.6, 5.1, 5.4, 5.5, 3.6, 2.5, 3.8, 4.1},
         {77, 1, 1.9, 2.5, 2.7, 2.8, 2.4, 2.3, 2.8, 3., 3.1, 3.1, 1.5, 1.1, 1.6, 1.7},
         {77, 2, 3.3, 4.5, 4.9, 5.1, 4.3, 4.7,
           5.6, 5.9, 6.1, 6.1, 3.4, 3.1, 4.1, 4.2},
         {77, 3, 4.8, 5.8, 6.1, 6., 4.4, 6.8, 7.6, 7.8, 7.7, 7.8, 4.2, 4.8, 5.6, 5.6}}
```

2. First we need to read the data using the function created in the previous exercise.

```
In[7]:= datafile = ToFileName[{"IPM3", "DataFiles"}, "23232.txt"]

Out[7]= IPM3\DataFiles\23232.txt
```

```
In[8]:= data = ReadSolarData[datafile];
```

Here is the first row of the extracted data.

```
In[9]:= data[[1]]

Out[9]= {61, 1, 1.6, 1.9, 2., 2., 1.6, 1.7, 2., 2., 2.1, 2.1, 0.7, 0.5, 0.8, 0.8}
```

For 1961, we want to extract the elements in the sixth column and add them; then repeat for each of the successive years. For example, this function selects all those rows that start with 61, then pulls off all sixth column elements.

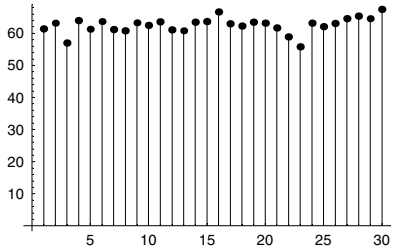
```
In[10]:= Part[Select[data, (#[[1]] == 61) &], All, 6]

Out[10]= {2., 4.4, 5.1, 6.4, 5.7, 6., 6.3, 6.2, 6.8, 5.9, 4.2, 2.3}

In[11]:= Length[%]

Out[11]= 12
```

The total solar radiation for 1961 is the sum of these values.

```
In[12]:= Apply[Plus, Part[Select[data, (#[[1]] == 61) &], All, 6]]

Out[12]= 61.3
```

Here are the minimum and maximum years (from the first column).

```
In[13]:= {ymin, ymax} = Map[{Min[#], Max[#]} &, Part[data, All, 1], {0}]

Out[13]= {61, 90}

In[14]:= yearlydata = Table[
           Apply[Plus, Part[Select[data, (#[[1]] == y) &], All, 6]], {y, ymin, ymax}]

Out[14]= {61.3, 63.1, 56.9, 63.9, 61.2, 63.6, 61.1, 60.7, 63.2,
          62.4, 63.5, 61., 60.7, 63.4, 63.6, 66.6, 62.9, 62.2, 63.4,
          63.1, 61.6, 58.8, 55.7, 63.1, 62., 63., 64.5, 65.3, 64.5, 67.4}

In[15]:= << Graphics`MultipleListPlot`

In[16]:= MultipleListPlot[yearlydata, SymbolShape → Stem];
```



3. We first load the necessary packages.

```
In[17]:= << Graphics`MultipleListPlot`

In[18]:= << Utilities`FilterOptions`

In[19]:= PlotSolarData[dat1_, dat2_, opts___?OptionQ] := Module[{months},
           months = {{1, "Jan"}, {2, "Feb"}, {3, "Mar"},
             {4, "Apr"}, {5, "May"}, {6, "Jun"}, {7, "Jul"}, {8, "Aug"},
             {9, "Sep"}, {10, "Oct"}, {11, "Nov"}, {12, "Dec"}};
           MultipleListPlot[dat1, dat2, FilterOptions[MultipleListPlot, opts],
            PlotJoined → True, AspectRatio → Automatic,
            Ticks → {months, Automatic}, AxesLabel → {None, "kWh/m^2/day"}]];
```

Read in the file.

```
In[20]:= datafile = ToFileName[{"IPM3", "DataFiles"}, "23232.txt"]

Out[20]= IPM3\DataFiles\23232.txt
```

Use `ReadSolarData` from the previous exercise to strip out all lines that start with a quote character and insure each element is a number.

```
In[21]:= data = ReadSolarData[datafile];
```

Using `GetData` developed in Section 11.1, we extract the sixth column for all dates between January 1980 and December 1980.

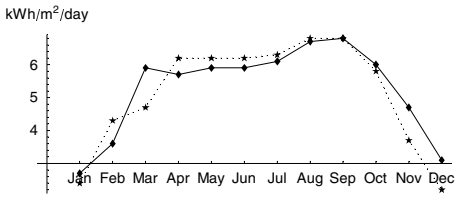
```
In[22]:= d1 = Part[GetData[data, {1, 80}, {12, 80}], All, 6]

Out[22]= {2.7, 3.6, 5.9, 5.7, 5.9, 5.9, 6.1, 6.7, 6.8, 6., 4.7, 3.1}
```

Similarly for data collected in 1981.

```
In[23]:= d2 = Part[GetData[data, {1, 81}, {12, 81}], All, 6]

Out[23]= {2.4, 4.3, 4.7, 6.2, 6.2, 6.2, 6.3, 6.8, 6.8, 5.8, 3.7, 2.2}
```

Finally, here is the plot.

```
In[24]:= PlotSolarData[d1, d2]
```




```
Out[24]= - Graphics -
```

11.2 Random walks

1. First defining `walk1DOffLattice` and then inserting an `If` statement immediately following `dim==1` will do the trick.

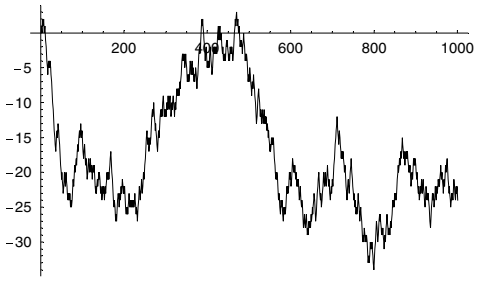
```
In[1]:= walk1DOffLattice[n_] :=
         FoldList[Plus, 0, Table[Random[Real, {-1, 1}], {n}]]

In[2]:= walk1D[n_] := NestList[# + (-1)^Random[Integer] &, 0, n]

In[3]:= walk2D[n_] :=
         Module[{NSEW = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}},
          FoldList[Plus, {0, 0},
           NSEW[[Table[Random[Integer, {1, 4}], {n}]]]]]

In[4]:= walk2DOffLattice[n_] :=
         FoldList[Plus, {0, 0},
          Map[{Cos[#], Sin[#]} &, Table[Random[Real, {0, 2 π}], {n}]]]

In[5]:= walk3D[n_] := Module[{NSEW3 = √2 Vertices[Cube]},
          FoldList[Plus, {0, 0, 0}, NSEW3[[Table[Random[Integer, {1, 8}], {n}]]]]]

In[6]:= walk3DOffLattice[n_] := FoldList[Plus, {0, 0, 0},
          Map[{Cos[#], Sin[#], #/(2 π)} &, Table[Random[Real, {-2 π, 2 π}], {n}]]]

In[7]:= Options[RandomWalk] = {LatticeWalk → True, Dimension → 2}

Out[7]= {LatticeWalk → True, Dimension → 2}

In[8]:= RandomWalk[n_, opts___?OptionQ] := Module[{dim, latticeQ},
          If[Not[IntegerQ[n] && n > 0],
           Message[RandomWalk::rwn, n], {latticeQ, dim} =
            {LatticeWalk, Dimension} /. Flatten[{opts, Options[RandomWalk]}];
           Which[
            dim == 1, If[latticeQ, walk1D[n], walk1DOffLattice[n]],
            dim == 2, If[latticeQ, walk2D[n], walk2DOffLattice[n]],
            dim == 3, If[latticeQ, walk3D[n], walk3DOffLattice[n]]
           ]]]
```

2. The output from `RandomWalk` with the option `Dimension` set to 1 is a one-dimensional list of integers.

```
In[9]:= RandomWalk[10, Dimension → 1]

Out[9]= {0, 1, 2, 1, 2, 3, 2, 1, 0, 1, 0}
```

This list can be passed directly to `ListPlot`.

```
In[10]:= ShowWalk[coords_, opts___] := Which[
           Length[Dimensions[coords]] == 1,
               ListPlot[coords, opts, PlotJoined → True],
           Dimensions[coords][[2]] == 2,
               Show[Graphics[Line[coords], opts, AspectRatio → Automatic]],
           Dimensions[coords][[2]] == 3,
               Show[Graphics3D[Line[coords], opts, AspectRatio → Automatic]]]

In[11]:= ShowWalk[RandomWalk[1000, Dimension → 1]];
```



3. First we write down the formulas for representing a point on the unit sphere.

```
In[12]:= x[ϕ_, θ_] := √(1 - Cos[ϕ]^2) Cos[θ];
         y[ϕ_, θ_] := √(1 - Cos[ϕ]^2) Sin[θ];
         z[ϕ_] := Cos[ϕ];
```

This checks that the formulas are correct:

```
In[15]:= Simplify[√(x[ϕ, θ]^2 + y[ϕ, θ]^2 + z[ϕ]^2)]

Out[15]= 1
```

The next step is to create a pair of angles between 0 and 2π.

```
In[16]:= ran = Table[Random[Real, {0, 2 π}], {2}]

Out[16]= {3.65812, 1.45235}
```

This applies the functions `x`, `y`, and `z` to this pair of angles.

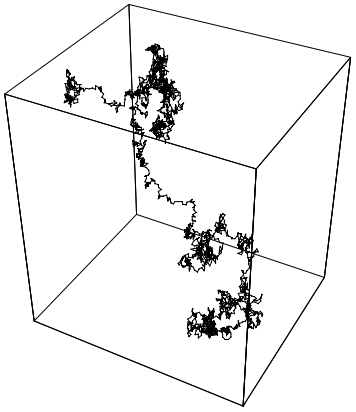
```
In[17]:= Apply[{x[#1, #2], y[#1, #2], z[#1]} &, ran]

Out[17]= {0.0583615, 0.490403, -0.869539}
```

Starting at the origin and folding `Plus` across this function gives the following.

```
In[18]:= n = 3;
         FoldList[Plus, {0, 0, 0}, Apply[{x[#1, #2], y[#1, #2], z[#1]} &,
           Table[Random[Real, {0, 2 π}], {n}, {2}], {1}]]

Out[19]= {{0, 0, 0}, {-0.0583035, 0.166546, 0.984308},
          {0.86393, 0.535037, 1.10136}, {0.64296, 0.540758, 0.126093}}
```

Here then is the rewritten `walk3DOffLattice` with this code inserted.

```
In[20]:= <<Graphics`Polyhedra`

In[21]:= walk3DOffLattice[n_] := Module[{x, y, z},
           x[ϕ_, θ_] := √(1 - Cos[ϕ]^2) Cos[θ];
           y[ϕ_, θ_] := √(1 - Cos[ϕ]^2) Sin[θ];
           z[ϕ_] := Cos[ϕ];
           FoldList[Plus, {0, 0, 0}, Apply[{x[#1, #2], y[#1, #2], z[#1]} &,
             Table[Random[Real, {0, 2 π}], {n}, {2}], {1}]]]

In[22]:= ShowWalk[
           RandomWalk[2500, Dimension → 3, LatticeWalk → False]];
```



5.

```
In[23]:= AnimateWalk[coords_, opts___] := Scan[
           Show[Graphics[{{RGBColor[1, 0, 0], PointSize[.02], Point[coords[[#1]]]},
               Line[Take[coords, #1]]}], opts, AspectRatio → Automatic,
             PlotRange → Map[{Min[#1] - .2, Max[#1] + .2} &, Transpose[coords]]] &,
           Range[2, Length[coords]]]

In[24]:= AnimateWalk[RandomWalk[50, LatticeWalk → False]];
```

11.3 The Game of Life

1.

```
In[1]:= Options[LifeGraphics] =
          Colors → {1 → RGBColor[1, 0, 0], 0 → RGBColor[0, 0, 0]};

In[2]:= LifeGraphics[lis_, opts___?OptionQ] := Module[{colors},
          colors = Colors /. Flatten[{opts, Options[LifeGraphics]}];
          Map[
           Graphics[RasterArray[
              Reverse[# /. colors]],
             AspectRatio → Automatic] &, lis]]

In[3]:= Options[LifeGraphics]

Out[3]= {Colors → {1 → RGBColor[1, 0, 0], 0 → RGBColor[0, 0, 0]}}

In[4]:= LifeGame[n_Integer?Positive, steps_] :=
         Module[{gameboard, liveNeighbors, update},
          gameboard = Table[Random[Integer], {n}, {n}];
          liveNeighbors[mat_] := Apply[Plus, Map[RotateRight[mat, #] &,
             {{-1, -1}, {-1, 0}, {-1, 1},
              {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}}]];
          update[1, 2] := 1;
          update[_, 3] := 1;
          update[_, _] := 0;
          SetAttributes[update, Listable];
          FixedPointList[update[#, liveNeighbors[#]] &, gameboard, steps]
         ]

In[5]:= Show[Last[LifeGraphics[LifeGame[10, 5]]]];
```

```
In[6]:= << Graphics`Colors`
```

```
In[7]:= Show[Last[LifeGraphics[LifeGame[10, 5],
          Colors → {0 → Blue, 1 → Green}]]]
```

```
Out[7]= - Graphics -
```

```
In[8]:= << Utilities`FilterOptions`
```

```
In[9]:= Options[LifeGraphics] =
          Colors → {1 → RGBColor[1, 0, 0], 0 → RGBColor[0, 0, 0]};
```

```
In[10]:= AnimateLife[lis_, opts___?OptionQ] :=
          Scan[Show, LifeGraphics[lis, FilterOptions[LifeGraphics, opts]]]
```

2. We will use essentially the same function as before, but we will "overload" the function by providing a definition for the case when a third argument is provided.

```
In[11]:= LifeGame[n_, steps_, lifeform_List] :=
          Module[{init = Table[0, {n}, {n}], gameboard, liveNeighbors, update},
           gameboard = ReplacePart[init, 1, lifeform];
           liveNeighbors[mat_] := Apply[Plus, Map[RotateRight[mat, #] &, {{-1, -1},
               {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}}]];
           update[1, 2] := 1;
           update[_, 3] := 1;
           update[_, _] := 0;
           Attributes[update] = Listable;
           FixedPointList[update[#, liveNeighbors[#]] &, gameboard, steps]]
```

If `LifeGame` is called with two arguments, then the definition given earlier will be applied (random initial game board). If `LifeGame` is called with three arguments, then this definition above will be matched.

Here is a game played on a 50×50 board, starting with a glider object initially at lattice site (20, 20), and played for ten generations.

```
In[12]:= glider[x_, y_] := {{x, y}, {x + 1, y}, {x + 2, y}, {x + 2, y + 1}, {x + 1, y + 2}}
```

```
In[13]:= lg50 = LifeGame[50, 10, glider[20, 20]];
```

This game could then be animated by evaluating `AnimateLife[lg50]`.

12 Writing packages

12.5 Writing your own packages

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
```

2. This is essentially the definition given in the solution to Exercise 5 from Section 5.3.

```
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}
```

3. 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.";
```

4. Here is the modified definition which now issues the warning message created in Exercise 3 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
```

5. 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 name `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["IPM3`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 `IPM3`Collatz`Private`` is then begun *within* `IPM3`Collatz`. All of the definitions of this package are given within this new context. The context `IPM3`Collatz`` `CollatzSequence` is defined within the `System`` context. The context of `collatz`, on the other hand, is `IPM3`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 `IPM3`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.