8 Advanced Counting Techniques

Introduction

In this chapter, we will describe how to apply *Mathematica* to three important topics in counting: recurrence relations, generating functions, and inclusion–exclusion. We begin by describing how *Mathematica* can be used to solve recurrence relations, including the recurrence relations that describe the complexity of divide-and-conquer algorithms. After studying recurrence relations, we show how to use *Mathematica* to manipulate generating functions and how these capabilities can help solve counting problems. We conclude the chapter with a discussion of the principle of inclusion–exclusion.

8.1 Recurrence Relations

A recurrence relation describes a relationship between the members of a sequence and their predecessors. For example, the famous Fibonacci sequence $\{f_n\}$ satisfies the recurrence relation

$$f_n = f_{n-1} + f_{n-2}$$

Together with the initial conditions $f_1 = 1$ and $f_2 = 1$, this relation is sufficient to define the entire sequence $\{f_n\}$.

To understand how to work with recurrence relations in *Mathematica*, we have to remember that a sequence $\{a_n\}$ is a function whose domain is a subset of the integers (usually the positive integers or nonnegative integers, depending on the context) and whose codomain contains the terms of the sequence (which can be numbers, matrices, circles, functions, etc.). (See the definition of sequence given in Section 2.4 of the textbook.)

With this point of view, the sequence $\{a_n\}$ is a function a and the nth term of the sequence is the value of the function evaluated at the integer n, that is, $a_n = a(n)$. This is only a change in notation, but it makes it easier to see that a recurrence relation can be represented with the Wolfram Language as a function taking integer arguments.

We can represent the Fibonacci sequence by the indexed variable and function below, which we use to compute the first 20 terms of the Fibonacci sequence. The initial conditions are given as specific assignments to an indexed variable, and the function that implements the recurrence relation stores computed values by assigning them to the indexed variable.

```
In[1]:=  fibonacci[1]=1;
         fibonacci[2]=1;
         fibonacci[n_Integer]:=
           fibonacci[n]=fibonacci[n-1]+fibonacci[n-2];
```

```
In[4]:=  Table[fibonacci[n],{n,1,20}]

Out[4]=  {1,1,2,3,5,8,13,21,34,55,89,144,
           233,377,610,987,1597,2584,4181,6765}
```

Using indexed variables and functions like the one above is the approach we developed in Section 5.3 of this manual to implement functions with recursive definitions. This is a useful general approach, but the Wolfram Language has more specific, and more efficient, methods for working with recurrence relations.

The Wolfram Language function `RecurrenceTable` is used to generate lists of the values of a sequence defined by a recurrence relation. `RecurrenceTable` requires three arguments. The first argument must be a list of equations specifying the recurrence relation and all initial conditions. Note that these must be given as equations, using `Equal` (==). In addition, the equations should be in terms of indexed variables such as `fibonacci[n]` or `a[k-1]`, or `b[0]`. The second argument is the symbol used to name the sequence in the equations, for example, `fibonacci` or `a`. The final argument is a range specification of the same form as used in a `Table`. For example, `{n,10,20}` would be used to generate the list of the values with $n = 10$ through $n = 20$ and `{k,10}` would generate the values up to $k = 10$.

To illustrate, the following generates the first 20 terms of the Fibonacci sequence.

```
In[5]:=  RecurrenceTable[{a[n]==a[n-1]+a[n-2],
           a[1]==1,a[2]==1},a,{n,20}]

Out[5]=  {1,1,2,3,5,8,13,21,34,55,89,144,
           233,377,610,987,1597,2584,4181,6765}
```

For some common sequences like the Fibonacci sequence, the Wolfram Language includes special functions to compute these values. For example, the 15th Fibonacci number can be computed using the `Fibonacci` function.

```
In[6]:=  Fibonacci[15]

Out[6]=  610
```

The `RecurrenceTable` function is very flexible, able to handle a wide variety of recurrence relations. For example, consider the recurrence relation defined by $a_{n+1} = {a_n}^2 - n \cdot a_{n-1}$, with initial conditions $a_0 = 1$ and $a_1 = 1$. The following expression computes a_{10}. Note the use of the `Last` function to obtain the last element of the sequence. This is equivalent to, but perhaps more expressive than, the `Part` specification `[[-1]]`.

```
In[7]:=  Last[RecurrenceTable[
           {a[n+1]==a[n]^2-n*a[n-1],a[0]==1,a[1]==1},
           a,{n,10}]]

Out[7]=  83095558751833088261963048239982609365670356
```

While `RecurrenceTable` is very general, sometimes a less-flexible function can be useful. The `LinearRecurrence` function produces output similar to that of `RecurrenceTable` but applies only to linear homogeneous recurrence relations (defined in Section 8.2 of the textbook). `LinearRecurrence` applies for recurrence relations of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

with initial conditions $a_1, a_2, \ldots, a_k$. `LinearRecurrence` requires three arguments. The first argument is the list of coefficients in the recurrence relation, i.e., $\{c_1, c_2, \ldots, c_k\}$. The second argument is the list of initial conditions $\{a_1, a_2, \ldots, a_k\}$. Note that the two lists should generally be the same length. The final argument is either a single integer to indicate the number of terms of the sequence to produce or a pair of integers representing a range of indices.

By way of comparison, the following two expressions use `RecurrenceTable` and `Linear-Recurrence` to compute the first 20 terms of the sequence defined by the recurrence relation $a_n = 2a_{n-1} - 3a_{n-2} + a_{n-3}$, with initial values $a_1 = 1, a_2 = 3, a_3 = 5$.

```
In[8]:=  RecurrenceTable[{a[n]==2*a[n-1]-3*a[n-2]+a[n-3],
           a[1]==1,a[2]==3,a[3]==5},a,{n,20}]

Out[8]=  {1,3,5,2,-8,-17,-8,27,61,33,-90,-218,
           -133,298,777,527,-979,-2762,-2060,3187}

In[9]:=  LinearRecurrence[{2,-3,1},{1,3,5},20]

Out[9]=  {1,3,5,2,-8,-17,-8,27,61,33,-90,-218,
           -133,298,777,527,-979,-2762,-2060,3187}
```

Tower of Hanoi Problem

In Example 2 of Section 8.1 of the textbook, the author describes the famous “Tower of Hanoi” puzzle and derives the recurrence relation

$$H_n = 2H_{n-1} + 1, \; H_1 = 1$$

where H_n represents the number of moves required to solve the puzzle for n disks. As discussed in the textbook, this has the solution

$$H_n = 2^n - 1.$$

Later in this manual, we will see how to use *Mathematica* to derive this result.

Rather than just computing the values, we can illustrate the solution to the Tower of Hanoi puzzle by writing functions in the Wolfram Language to compute the moves needed and to describe them. We will write a small program consisting of three functions: the main program `hanoi`, a utility function `printMove`, and `transferDisk`, which does most of the work.

The easiest part to write is the function `printMove`, which merely displays the move to make at a given step.

```
In[10]:= printMove[src_String,dest_String]:=
           Print["Move disk from peg ",src," to peg ",dest,"."]
```

The function `Print` combines all of its arguments and display them on a single line.

Next, we write the recursive function `transferDisk`, which does most of the work. This function models the idea of transferring a stack of `ndisks` disks from the source peg, which is given as the argument `src`, to the destination peg, `dest`, via the intermediate peg, `via`. As is described in the textbook, in order to move a stack of n disks, first move the top $n - 1$ pegs to the intermediate peg (using the destination as the intermediary), then move the bottom disk to the destination, and then move the smaller stack from the intermediate peg to the destination. Unless, of course, there is only one disk, in which you just move that disk to the destination. This is coded as follows:

```
In[11]:=  transferDisk[src_String,via_String,dest_String,
           ndisks_Integer]:=
          If[ndisks==1,
            printMove[src,dest],
            transferDisk[src,dest,via,ndisks-1];
            printMove[src,dest];
            transferDisk[via,src,dest,ndisks-1]
          ]
```

Finally, we package the recursive procedure in a top-level function, `hanoi`, providing an interface to the recursive engine.

```
In[12]:=  hanoi[ndisks_Integer]/;ndisks>1:=
            transferDisk["A","B","C",ndisks]
```

Our `hanoi` program can exhibit a specific solution to the Tower of Hanoi puzzle for any number of disks:

```
In[13]:=  hanoi[2]

          Move disk from peg A to peg B.

          Move disk from peg A to peg C.

          Move disk from peg B to peg C.

In[14]:=  hanoi[3]

          Move disk from peg A to peg C.

          Move disk from peg A to peg B.

          Move disk from peg C to peg B.

          Move disk from peg A to peg C.

          Move disk from peg B to peg A.

          Move disk from peg B to peg C.

          Move disk from peg A to peg C.
```

Try experimenting with different values of `ndisk` to get a feel for how large the problem becomes for even moderately large numbers of disks.

Dynamic Programming

We conclude this section with an implementation of Algorithm 1 from Section 8.1 of the text. Recall that the goal of this algorithm is to find the maximum number of attendees that can be achieved by a schedule of talks.

We will represent each talk as a list of three elements, with the start time being the first element, the end time in the second position, and the weight, or attendance, will be last. Time of day will be represented by a single number with whole part equal to the hour in the 24-hour system and with fractional part equal to the part of an hour that corresponds to the number of minutes. For instance, 2:30 P.M. would be represented as 14.5.

As an example, consider the following eight talks.

StartTime	EndTime	Attendance
9 : 00AM	11 : 00AM	17
9 : 00AM	10 : 30AM	15
10 : 00AM	11 : 30AM	22
10 : 30AM	12 : 00PM	11
11 : 30AM	1 : 30PM	18
12 : 00PM	1 : 00PM	12
1 : 30PM	3 : 00PM	21
2 : 00PM	4 : 00PM	17

We create the following list of lists to represent the talks.

```
In[15]:=  talks={{9,11,17},{9,10.5,15},{10,11.5,22},
            {10.5,12,11},{11.5,13.5,18},{12,13,12},
            {13.5,15,21},{14,16,17}}

Out[15]=  {{9,11,17},{9,10.5,15},{10,11.5,22},{10.5,12,11},
            {11.5,13.5,18},{12,13,12},{13.5,15,21},{14,16,17}}
```

Recall the description of Algorithm 1 from the text. We summarize the general outline of the algorithm below.

1. Sort the talks in order of increasing end time.
2. For each index j, compute $p(j)$—the maximum index i such that talk i is compatible with talk j.
3. For each index j, compute $T(j)$, which is computed by the recurrence relation $T(j) = \max\left(w_j + T(p(j)), T(j-1)\right)$ and with initial condition $T(0) = 0$.
4. The maximum total number of attendees is $T(n)$, where n is the number of talks.

For step 1, we will make use of `Sort` with a custom ordering function. `Sort` can accept an optional argument in the form of a pure function of two arguments. This function should return true if the first argument precedes the second and false otherwise. Since we must sort the talks in increasing order of end time (which is stored in position 2 in the lists representing the talks), we will use the following function as the second argument to `Sort`. Recall that `#1` and `#2` indicate the inputs to the function and `&` is used to terminate a pure `Function`.

```
#1[[2]]<#2[[2]]&
```

For step 2, we must compute $p(j)$. To do this, we will create a function that accepts the sorted list of talks and returns an indexed variable that represents the function p. Recall that the value of $p(j)$ is the largest index among talks compatible with the talk with index j, so we name the function `compatible`.

After declaring local variables, the function `compatible` will loop through all the indices, j, from 1 to the number of talks in the list. We use a local variable, `jstart`, to store the start time of the current talk being analyzed. We then consider all the talks earlier in the list beginning with the talk with index $j-1$

and working backwards to talk 1. For each talk, we check to see if it ends before talk j starts. When we find such a talk, we set its index to the value of $p(j)$ (since we are working backward, the first one found is the talk with the largest index). If no compatible talk is found, then $p(j)$ is set to 0. Here is the function.

```
In[16]:=  compatible[talkList_]:=Module[{p,j,jstart,i},
            For[j=1,j≤Length[talkList],j++,
              jstart=talkList[[j]][[1]];
              p[j]=Catch[
                For[i=j-1,i≥1,i--,
                  If[talkList[[i]][[2]]≤jstart,
                    Throw[i]
                  ]
                ];
                Throw[0]
              ]
            ];
            p
          ]
```

For step 3, we must compute $T(j)$. To do this, we create a function that accepts as input the sorted list of talks and the indexed variable representing the function p. Initialize `T` by setting its value at 0 to 0. Then, consider each integer j from 1 to the number of talks and apply the formula: $T(j) = \max\left(w_j + T(p(j)), T(j-1)\right)$.

```
In[17]:=  totalAttendance[talkList_,p_]:=Module[{j,T},
            T[0]=0;
            For[j=1,j≤Length[talkList],j++,
              T[j]=Max[talkList[[j]][[3]]+T[p[j]],T[j-1]]
            ];
            T
          ]
```

We can now put the pieces together as outlined at the start of this subsection.

```
In[18]:=  maximumAttendance[talkList_List]:=Module[{L,p,T},
            L=Sort[talkList,#1[[2]]<#2[[2]]&];
            p=compatible[L];
            T=totalAttendance[L,p];
            T[Length[talkList]]
          ]
```

And thus, the maximum attendance for the talks described above is:

```
In[19]:=  maximumAttendance[talks]

Out[19]=  61
```

8.2 Solving Linear Recurrence Relations

The Wolfram Language has a very powerful recurrence solver, `RSolve`. Its use, however, can obscure some of the important ideas that are involved. Therefore, we will first use some of the Wolfram Language's more fundamental facilities to solve certain kinds of recurrence relations one step at a time.

Given a recursively defined sequence $\{a_n\}$, we would like to find a formula, involving only the index n (and, perhaps, other fixed constants and known functions) that does not depend on knowing the value of any prior elements of the sequence.

Linear Homogeneous Recurrence Relations with Constant Coefficients

We will begin by considering recurrence relations that are linear, homogeneous, and which have constant coefficients; that is, they have the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

where $c_1, c_2, \ldots, c_k$ are real constants and c_k is nonzero. Recall that the integer k is called the degree of this recurrence relation. To have a unique solution, at least k initial conditions must be specified.

The general method for solving such a recurrence relation involves finding the roots of its characteristic polynomial

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \cdots - c_{k-1} r - c_k$$

When this polynomial has distinct roots, all solutions are linear combinations of the nth powers of these roots. When there are repeated roots, the situation is a little more complicated, as we will see.

A First Example

Consider the linear homogeneous recurrence relation with constant coefficients of degree two

$$a_n = 2a_{n-1} + 3a_{n-2}$$

subject to the initial conditions $a_1 = 4$ and $a_2 = 2$. Its characteristic equation is

$$r^2 - 2r - 3 = 0$$

To solve the recurrence relation, we find the roots of the equation by applying the `Solve` function.

```
In[20]:=  Solve[r^2-2r-3==0,r]

Out[20]=  {{r→-1},{r→3}}
```

The `Solve` function computes the values of the variable r, given as the second argument, that satisfy the equation in the first argument. Note that the output is a list of lists, within which are rules specifying the solutions.

Now that *Mathematica* has determined that the solutions are $r = -1$ and $r = 3$, we can write down the form of the solution to the recurrence as

$$a_n = \alpha \cdot (-1)^n + \beta \cdot 3^n$$

where α and β are constants that we have yet to determine.

Since the initial conditions are $a_1 = 4$ and $a_2 = 2$, our recurrence relation must satisfy the following pair of equations.

$$\begin{cases} -\alpha + 3\beta = 4 \\ \alpha + 9\beta = 2 \end{cases}$$

To find the solution to this system of linear equations, we again use `Solve`:

```
In[21]:=  Solve[{-alpha+3*beta==4,alpha+9 * beta==2},{alpha,beta}]

Out[21]=  {{alpha→-5/2,beta→1/2}}
```

That the first argument is a list of equations indicates that they represent a system of equations to be solved simultaneously. Likewise, the variables to be solved for form a list.

Now that we have the values for α and β, we see that the complete solution to the recurrence relation is

$$a_n = \frac{-5}{2}(-1)^n + \frac{1}{2}3^n$$

This formula allows us to write a function for finding the terms of the sequence $\{a_n\}$, which can be more efficient than a recursive approach.

```
In[22]:=  aFormula[n_Integer]:=(-5/2)*(-1)^n+(1/2)*3^n

In[23]:=  Table[aFormula[n],{n,1,10}]

Out[23]=  {4,2,16,38,124,362,1096,3278,9844,29522}
```

A Second Example

We consider another example. We will solve the recurrence relation

$$a_n = \frac{-5}{3}a_{n-1} + \frac{2}{3}a_{n-2}$$

with initial conditions $a_1 = \frac{1}{2}$ and $a_2 = 4$.

To do this, we have *Mathematica* solve the characteristic equation of the recurrence relation, and then solve the system of linear equations obtained from the roots of the characteristic equation and the initial conditions. Note that this method works because this recurrence relation is linear, homogeneous, and has constant coefficients.

```
In[24]:=  charEqnRoots=Solve[r^2+(5/3)*r-(2/3)==0,r]

Out[24]=  {{r→-2},{r→1/3}}
```

Note that we can transform the output from a list of lists of rules into a simple list of the solutions using the `ReplaceAll` (`/.`) operator as shown below.

```
In[25]:=  charRootsL=r/.charEqnRoots

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

When ReplaceAll is given a list of lists of rules as its right operand, its output is the list obtained by applying the rules in each sublist in turn.

This allows us to solve for α and β by referencing the solutions to the characteristic equation, rather than typing them.

```
In[26]:=  Solve[
            {alpha*charRootsL[[1]]+beta*charRootsL[[2]]==1/2,
             alpha*charRootsL[[1]]^2+beta*charRootsL[[2]]^2==4},
            {alpha,beta}]
```

Out[26]= $\{\{\text{alpha}\to\frac{23}{28},\text{beta}\to\frac{45}{7}\}\}$

Thus, we see that the solution to the recurrence relation is

$$a_n = \frac{23}{28}(-2)^n + \frac{45}{7}\left(\frac{1}{3}\right)^n$$

The Fibonacci Sequence

We can derive an explicit formula for the Fibonacci sequence this way as well. The characteristic polynomial for the Fibonacci sequence is

$$r^2 - r - 1$$

We find the roots of the characteristic equation.

```
In[27]:=  cEqnRoots=r/.Solve[r^2-r-1==0,r]
```

Out[27]= $\{\frac{1}{2}\ (1-\sqrt{5}),\frac{1}{2}\ (1+\sqrt{5})\}$

Therefore, the formula for the nth Fibonacci number is of the form

```
In[28]:=  Fn=alpha*cEqnRoots[[1]]^n+beta*cEqnRoots[[2]]^n
```

Out[28]= $(\frac{1}{2}\ (1-\sqrt{5}))^n\ \text{alpha}+(\frac{1}{2}\ (1+\sqrt{5}))^n\ \text{beta}$

We find the coefficients α and β in the formula by using the initial conditions.

```
In[29]:=  alphas=
            Solve[{alpha*cEqnRoots[[1]]+beta*cEqnRoots[[2]]==1,
                alpha*cEqnRoots[[1]]^2+beta*cEqnRoots[[2]]^2==1},
              {alpha,beta}]
```

Out[29]= $\{\{\text{alpha}\to-\frac{1}{\sqrt{5}},\text{beta}\to-\frac{-5-\sqrt{5}}{5\ (1+\sqrt{5})}\}\}$

We can use ReplaceAll (/.) to substitute the values for α and β into the formula Fn. Note that we access the "first" element of alphas in order to not have a nested list.

```
In[30]:= Fn/.alphas[[1]]
```

Out[30]= $$-\frac{\left(\frac{1}{2}\left(1-\sqrt{5}\right)\right)^{n}}{\sqrt{5}}-\frac{1}{5}\,2^{-n}\left(-5-\sqrt{5}\right)\left(1+\sqrt{5}\right)^{-1+n}$$

```
In[31]:= Simplify[Fn/.alphas[[1]]]
```

Out[31]= $$-\frac{2^{-n}\left(5+\sqrt{5}\right)\left(\left(1-\sqrt{5}\right)^{n}-\left(1+\sqrt{5}\right)^{n}\right)}{5\left(1+\sqrt{5}\right)}$$

If we are to use such a formula to repeatedly compute values, then we should define a function. You can do this by retyping the formula or copy and paste it into a function definition. Alternatively, you can use the expression as the function definition as shown below.

```
In[32]:= fibonacci2[n_]=Fn/.alphas[[1]]
```

Out[32]= $$-\frac{\left(\frac{1}{2}\left(1-\sqrt{5}\right)\right)^{n}}{\sqrt{5}}-\frac{1}{5}\,2^{-n}\left(-5-\sqrt{5}\right)\left(1+\sqrt{5}\right)^{-1+n}$$

Note that in this situation, it is important to use Set (=), not the usual SetDelayed (:=). This is so that the right-hand side resolves into the expression before the function is defined.

Observe that *Mathematica* will not necessarily simplify values automatically.

```
In[33]:= fibonacci2[1]
```

Out[33]= $$-\frac{1-\sqrt{5}}{2\sqrt{5}}+\frac{1}{10}\left(5+\sqrt{5}\right)$$

However, applying the Simplify function will reduce the expression to the expected value.

```
In[34]:= fibonacci2[1]//Simplify

Out[34]= 1
```

A Solver

Now, we generalize what we have been doing and write a function in the Wolfram Language to solve a degree two linear, homogeneous recurrence relation with constant coefficients, provided that the roots of the characteristic polynomial are distinct. We will write a function recSol2Distinct that solves the recurrence

$$a_n = c \cdot a_{n-1} + d \cdot a_{n-2}$$

subject to the initial conditions $a_1 = u$ and $a_2 = v$, and then returns a Function that can be used to compute terms of the sequence.

For the moment, we assume that the characteristic polynomial $r^2 - c \cdot r - d$ has two distinct roots. Later, we will modify the function to relax that restriction. With the assumption that the roots of the characteristic polynomial are distinct, all our function needs to do is to repeat the steps we did manually in the examples above.

```
In[35]:=  recSol2Distinct[c_,d_,u_,v_]:=
            Module[{CERoots,alphas,alpha,beta,f,r},
              (* first solve the characteristic equation *)
              CERoots=r/.Solve[r^2-c*r-d==0,r];
              (* next solve using the initial conditions *)
              alphas=
                Solve[{alpha*CERoots[[1]]+beta*CERoots[[2]]==u,
                   alpha*CERoots[[1]]^2+beta*CERoots[[2]]^2==v},
                 {alpha,beta}][[1]];
              (* finally substitute the results into the general form *)
              Function[n,
                alpha*CERoots[[1]]^n+beta*CERoots[[2]]^n/.alphas]
            ]
```

Observe that the final expression in the body of the `Module` is an application of `Function`. This means that the result in `recSol2Distinct` will be a pure function. The first argument to `Function` is the variable `n`, which is the variable representing the input to the function as opposed to the symbols used internally to the `Module`.

To construct a function for computing the Fibonacci sequence, invoke the new function with $c = d = 1$ and $u = v = 1$. We assign the resulting `Function` to the symbol `f` and use it to compute the first 10 Fibonacci numbers.

```
In[36]:=  f=recSol2Distinct[1,1,1,1];
          Table[Simplify[f[n]],{n,1,10}]

Out[37]=  {1,1,2,3,5,8,13,21,34,55}
```

We can apply `f` to a variable and simplify to see a closed-form formula for the Fibonacci numbers.

```
In[38]:=  f[n]//Simplify
```

Out[38]=

$$-\frac{2^{-n}\,(5+\sqrt{5})\,\left((1-\sqrt{5})^{n}-(1+\sqrt{5})^{n}\right)}{5\,(1+\sqrt{5})}$$

Evaluating the symbol `f` without an argument tells us very little about the recurrence relation, but provides some information about how `Function` and `Module` work.

```
In[39]:=  f

Out[39]=  Function[n$,alpha$2158 CERoots$2158[[1]]^n$+
            beta$2158 CERoots$2158[[2]]^n$/. alphas$2158]
```

The dollar signs indicate that the variables in the expression are localized to the `Function` and `Module`. The numbers after the dollar signs ensure that the symbols being used are unique. If you use the variable `alphas` in a different module, or even redefine `f`, the values will change. But those unique symbols store the appropriate values, so that when you apply `f` to a value or variable, the substitution represented by `ReplaceAll` (`/.`) and the computations are performed.

A Recurrence with Repeated Roots

We will next create a function that can handle the case of repeated roots. First, though, we look at an example of a recurrence relation whose characteristic polynomial has a double root. The recurrence relation

$$a_n = 4a_{n-1} - 4a_{n-2}$$

has characteristic equation

```
In[40]:=    charEqn=r^2-4*r+4==0

Out[40]=    4-4 r+r^2==0
```

Its roots are:

```
In[41]:=    CERoots=r/.Solve[charEqn,r]

Out[41]=    {2,2}
```

We can clearly see that in this case the root is repeated, but for *Mathematica* to recognize it, we need to use the following test.

```
In[42]:=    CERoots[[1]]==CERoots[[2]]

Out[42]=    True
```

If we call the double root (2 in this case) r_0, then the recurrence relation has the explicit solution

$$a_n = \alpha \cdot r_0^n + n \cdot \beta \cdot r_0^n$$

for all positive integers n, and for some constants α and β. The initial conditions of $a_1 = 1$ and $a_2 = 4$ produce the system of equations

$$\begin{cases} \alpha \cdot 2^1 + 1 \cdot \beta \cdot 2^1 = 1 \\ \alpha \cdot 2^2 + 2 \cdot \beta \cdot 2^2 = 4 \end{cases}$$

As before, we solve this system for α and β.

```
In[43]:=    alphas2=
              Solve[{alpha*CERoots[[1]]+beta*CERoots[[1]]==1,
                 alpha*CERoots[[1]]^2+2*beta*CERoots[[1]]^2==4},
                {alpha,beta}][[1]]

Out[43]=    {alpha→0,beta→1/2}
```

And finally, substitute these values into the general form $a_n = \alpha \cdot r_0^n + n \cdot \beta \cdot r_0^n$.

```
In[44]:=    alpha*CERoots[[1]]^n+n*beta*CERoots[[1]]^n/.alphas2

Out[44]=    2^(-1+n) n
```

A More General Recurrence Solver

The steps carried out above are quite general and we can write a function, `recSolver2`, which solves a recurrence relation (degree two, linear, homogeneous, with constant coefficients) regardless of whether the characteristic polynomial has distinct roots or not. The following function solves the recurrence

$$a_n = c \cdot a_{n-1} + d \cdot a_{n-2}$$

with initial conditions $a_1 = u$ and $a_2 = v$.

```
In[45]:=  recSolver2[c_,d_,u_,v_]:=
            Module[{CERoots,alphas,alpha,beta,r},
              (* first solve the characteristic equation *)
              CERoots=r/.Solve[r^2-c*r-d==0,r];
              (* then test if the roots are the same *)
              If[CERoots[[1]]==CERoots[[2]],
                (* the roots are the same, follow the last example *)
                alphas=
                  Solve[{alpha*CERoots[[1]]+beta*CERoots[[1]]==u,
                     alpha*CERoots[[1]]^2+2*beta*CERoots[[1]]^2==v},
                    {alpha,beta}][[1]];
                Return[Function[n,
                  alpha*CERoots[[1]]^n+n*beta*CERoots[[1]]^n/.
                    alphas]],
                (* otherwise, use the recSol2Distinct method *)
                alphas=
                  Solve[{alpha*CERoots[[1]]+beta*CERoots[[2]]==u,
                     alpha*CERoots[[1]]^2+beta*CERoots[[2]]^2==v},
                    {alpha,beta}][[1]];
                Return[Function[n,
                  alpha*CERoots[[1]]^n+beta*CERoots[[2]]^n/.
                    alphas]]
              ]
            ]
```

The `recSolver2` function first tests for a repeated root and then does the appropriate computation. We test this function on the examples we did by hand, such as the Fibonacci sequence:

```
In[46]:=  recSolver2[1,1,1,1][n]
```

Out[46]= $-\frac{\left(\frac{1}{2}\left(1-\sqrt{5}\right)\right)^n}{\sqrt{5}}-\frac{1}{5}\times 2^{-n}\left(-5-\sqrt{5}\right)\left(1+\sqrt{5}\right)^{-1+n}$

For the example with a double root:

```
In[47]:=  recSolver2[4,-4,1,4][n]
```

Out[47]= $2^{-1+n}\ n$

In both of those examples, the result is consistent with what we had obtained before. We will now use the solver to find the first 10 terms of the sequence defined by the following recurrence relation and initial conditions.

$$a_n = 4a_{n-1} - 3a_{n-2},\ a_1 = 1,\ \text{and } a_2 = 2$$

We define `g` to be the solution, but suppress the output with a semicolon, since we know the output of the assignment will just show the form of the pure function definition in terms of the local variables.

```
In[48]:=   g=recSolver2[4,-3,1,2];

In[49]:=   Table[Simplify[g[n]],{n,1,10}]

Out[49]=   {1,2,5,14,41,122,365,1094,3281,9842}
```

As another example, consider the following recurrence relation

$$a_n = -a_{n-1} - a_{n-2}$$

with initial conditions $a_1 = 1$ and $a_2 = 2$.

```
In[50]:=   h=recSolver2[-1,-1,1,2];
           h[n]
```

Out[51]= $$\frac{(-1)^{1+\frac{2n}{3}}\ (2+(-1)^{1/3})}{1+(-1)^{1/3}} - \frac{(-(-1)^{1/3})^{n}\ (1+2\ (-1)^{1/3})}{1+(-1)^{1/3}}$$

Notice that the solution to this recurrence is very complicated and requires the use of cube roots of -1. However, if we compute the first 10 terms, we notice a very simple pattern emerges.

```
In[52]:=   Table[Simplify[h[n]],{n,1,10}]

Out[52]=   {1,2,-3,1,2,-3,1,2,-3,1}
```

Mathematica can make this pattern more explicit if we replace the numerical initial conditions with symbolic constants.

```
In[53]:=   k=recSolver2[-1,-1,λ,μ];
           k[n]
```

Out[54]= $$\frac{(-1)^{1+\frac{2n}{3}}\ ((-1)^{1/3}\ \lambda+\mu)}{1+(-1)^{1/3}} - \frac{(-(-1)^{1/3})^{n}\ (\lambda+(-1)^{1/3}\ \mu)}{1+(-1)^{1/3}}$$

```
In[55]:=   Table[Simplify[k[n]],{n,1,10}]

Out[55]=   {λ,μ,-λ-μ,λ,μ,-λ-μ,λ,μ,-λ-μ,λ}
```

Note that the Greek letters are entered by pressing $\boxed{\text{ESC}}$, then typing the name of the letter, and pressing $\boxed{\text{ESC}}$ again, for example, $\boxed{\text{ESC}}$lambda$\boxed{\text{ESC}}$ produces λ.

Nonhomogeneous Recurrence Relations

So far, we have restricted our attention to homogeneous linear recurrence relations with constant coefficients. However, the techniques used in solving them may be extended to provide solutions to nonhomogeneous linear recurrence relations with constant coefficients. That is, recurrence relations of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + F(n)$$

with $c_1, c_2, \ldots, c_k$ real numbers and $F(n)$ a function depending only on n. To solve this more general form of a recurrence relation, we do two things: (1) find the solutions of the associated homogeneous recurrence relation (the relation obtained by removing $F(n)$) and (2) find a particular solution for the nonhomogeneous equation.

Consider the following example:

$$a_n = 6a_{n-1} - 9a_{n-2} + n \cdot 3^n$$

from Example 12 of Section 8.2 in the text.

The first step is to find the solutions to the associated homogeneous recurrence relation

$$a_n = 6a_{n-1} - 9a_{n-2}$$

To do this, we can use our `recSolver2`. We will use α and β for the initial conditions so that we get all the solutions.

```
In[56]:=  hSolution=recSolver2[6,-9,α,β];
          hSolution[n]

Out[57]=  3^(-2+n) (6 α-β)+3^(-2+n) n (-3 α+β)
```

The second step is to find a particular solution. Theorem 6 in Section 8.2 of the textbook tells us how to find the form of the particular solution. Note that $F(n) = n \cdot 3^n$ and 3 is a root of the characteristic polynomial with multiplicity 2 (you can verify this by solving the characteristic equation of the associated homogeneous relation; it is also made apparent by the form of `hSolution`). Thus, the theorem tells us that there is a particular solution of the form

$$n^2(p \cdot n + q)3^n$$

We will define a function for the form of this particular solution.

```
In[58]:=  pForm[n_]=n^2*(p*n+q)*3^n

Out[58]=  3^n n^2 (n p+q)
```

To find a particular solution, we need to find the values of p and q. To find these values, we substitute the terms of `pForm` into the recurrence relation. This gives us an equation in terms of p and q (and n).

```
In[59]:=  pEqn=pForm[n]==6*pForm[n-1]-9*pForm[n-2]+n*3^n

Out[59]=  3^n n^2 (n p+q)==
            3^n n-3^n (-2+n)^2 ((-2+n) p+q)+2 3^n (-1+n)^2 ((-1+n) p+q)
```

We simplify the equation with `Simplify`.

```
In[60]:=  pEqn=Simplify[pEqn]

Out[60]=  3^n (-6 p+n (-1+6 p)+2 q)==0
```

Next, we will have *Mathematica* solve that equation for p and q. In order to indicate that we want *Mathematica* to find values of p and q that satisfy the equation for all values of n, we will ask *Mathematica* to solve

$$(\forall n)3^n(-6p + n(-1 + 6p) + 2q) = 0$$

We encode this using the `ForAll` expression, with first argument `n`, indicating that `n` is the variable being quantified, and with second argument the equation. This expression will be the first argument to `Solve`. The second argument will be `{p,q}`, indicating that those are the variables we wish to solve for.

```
In[61]:=  pVals=Solve[ForAll[n,pEqn],{p,q}]
```

```
                1    1
Out[61]=  {{p→—,q→—}}
                6    2
```

Thus, the particular solution is:

```
In[62]:=  pForm[n]/.pVals[[1]]
```

```
           n  1 n   2
Out[62]=  3  (—+—) n
              2 6
```

Putting it all together, we see that all solutions to the recurrence relation $a_n = 6a_{n-1} - 9a_{n-2} + n \cdot 3^n$ are of the form:

```
In[63]:=  hSolution[n]+(pForm[n]/.pVals[[1]])
```

```
           n  1 n   2  -2+n              -2+n
Out[63]=  3  (—+—) n +3     (6 α-β)+3     n (-3 α+β)
              2 6
```

where α and β are the initial conditions.

Mathematica's Recurrence Solver

Now that we have seen how to use *Mathematica* to implement an algorithm to solve simple recurrence relations, it is time to introduce the Wolfram Language built-in function for solving recurrence relations.

We have already seen the Wolfram Language function `Solve` for working with polynomial equations and systems of equations. Similarly, there is a function `RSolve`, which is engineered for dealing with recurrence relations. It is a much more sophisticated version of our `recSolver2` function and can deal with recurrence relations of arbitrary degree, repeated roots, and nonlinear recurrence relations. The syntax of `RSolve` is very similar to that of `RecurrenceTable`. The first argument is an equation or list of equations representing the recurrence relation and any initial conditions in terms of symbols of the form `a[n+1]`, `a[n]`, `a[n-1]`, etc. The second argument is of one of two forms: `a` or `a[n]`, where `a` is the symbol used in the first argument to name the sequence. The form of the output depends on which of these is used. The final argument is the symbol used in the first argument to represent the index of the sequence, for example, `n`.

For example, to solve the recurrence relation $a_n = a_{n-1} + 2 \cdot a_{n-2}$ with initial conditions $a_1 = a_2 = 1$, you can enter the following.

```
In[64]:=  solveOutput=
            RSolve[{a[n]==a[n-1]+2*a[n-2],a[1]==a[2]==1},
              a[n],n]
```

```
                  1       1+n  n
Out[64]=  {{a[n]→— ((-1)   +2 )}}
                  3
```

Note that the structure of the solution is similar to that given by `Solve`: a list of lists of rules. If there were multiple solutions, each solution would appear as a separate sublist.

In the example above, we gave the second argument in the form `a[n]`. This resulted in a rule for `a[n]`. The expression on the right side of the rule can be extracted by applying `ReplaceAll` (`/.`) to the expression `a[n]`.

```
In[65]:=  a[n]/.First[solveOutput]

Out[65]=  1/3 ((-1)^(1+n)+2^n)
```

Let's break that down a bit. The `First` function simply takes the first element of the `solveOutput` list, which is the list containing the rule. That is, the above is the same as

```
          a[n]/.{a[n]→1/3(-(-1^n+2^n)}
```

This uses `ReplaceAll` (`/.`) on the expression `a[n]` by applying the rule that substitutes the expression for the symbol `a[n]`. In other words, the above evaluates to the expression $\frac{1}{3}(-(-1)^n + 2^n)$.

If we give the second argument in the form `a` instead of `a[n]`, the output will have the same structure, a list of lists of rules, but the rules will relate the symbol `a` to a pure `Function` (`&`) rather than an expression.

```
In[66]:=  solveOutput2=
            RSolve[{a[n]==a[n-1]+2*a[n-2],a[1]==a[2]==1},a,n]

Out[66]=  {{a→Function[{n},1/3 ((-1)^(1+n)+2^n)]}}
```

This output is more difficult to read, but it makes it easier to transform the output of `RSolve` into a function that we can use to compute values. To assign the symbol `F` to the function that calculates values of this sequence, we just have to assign `F` to the pure function, which we can extract just as we did above.

```
In[67]:=  F=a/.First[solveOutput2]

Out[67]=  Function[{n},1/3 ((-1)^(1+n)+2^n)]
```

Now, `F` can be used to compute values.

```
In[68]:=  F[42]

Out[68]=  1466015503701

In[69]:=  Table[F[n],{n,1,10}]

Out[69]=  {1,1,3,5,11,21,43,85,171,341}
```

Observe that these agree with the values obtained with `RecurrenceTable`.

```
In[70]:=  RecurrenceTable[{a[n]==a[n-1]+2*a[n-2],
            a[1]==a[2]==1},a,{n,1,10}]

Out[70]=  {1,1,3,5,11,21,43,85,171,341}
```

The RSolve function will let us solve nonhomogeneous recurrence relations like the Tower of Hanoi problem very easily. Recall that the Tower of Hanoi problem has the recurrence relation

$$H_n = 2 \cdot H_{n-1} + 1$$

with initial condition $H_1 = 1$.

```
In[71]:=  Clear[H];
          H=H/.First[RSolve[{H[n]==2*H[n-1]+1,H[1]==1},H,n]]

Out[72]=  Function[{n},-1+2^n]
```

Observe that in the above we used H for both the name of the sequence and the symbol representing the resulting Function. While this is natural, it carries the caveat that it can only be executed once, because H now has a value and thus cannot be used within RSolve. Hence, it is a good idea to Clear the symbol before making the assignment.

It is not necessary to specify the initial conditions for a recurrence relation when applying RSolve. If they are not present, *Mathematica* will still solve the equation, inserting symbolic constants (e.g., C[1] and C[2]) in place of numeric values, as the following example illustrates.

```
In[73]:=  Clear[G];
          G=G/.First[RSolve[G[n]==2*G[n-1]-6*G[n-2],G,n]]

Out[74]=  Function[{n},(1-i √5)^n C[1]+(1+i √5)^n C[2]]
```

The function G is still able to compute values, but they will be in terms of the constants.

```
In[75]:=  G[5]

Out[75]=  (1-i √5)^5 C[1]+(1+i √5)^5 C[2]
```

The capabilities of RSolve, like other Wolfram Language functions, are constantly being enhanced and extended. However, RSolve is not a panacea—you can easily find recurrence relations that it is incapable of solving. When RSolve is unable to solve a recurrence relation, it simply returns unevaluated, as below.

```
In[76]:=  RSolve[u[n]==u[n-1]^2-Exp[2*e[n-2]],u,n]

Out[76]=  RSolve[u[n]==-e^(2 e[-2+n])+u[-1+n]^2,u,n]
```

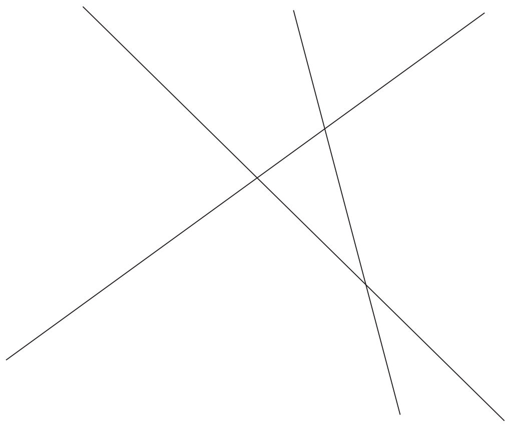
Problem Solving with *Mathematica* and Recurrence Relations

It is often the case that a problem, as presented, gives no clue that a solution may be found using recurrence. Let us explore how we can use *Mathematica* to solve a problem that is not explicitly expressed as one requiring the use of recurrence for its solution.

Here is our problem: into how many regions is the plane divided by 1000 lines, assuming that no two of the lines are parallel, and no three are coincident? Such a situation may arise in an attempt to model fissures in the ocean floor.

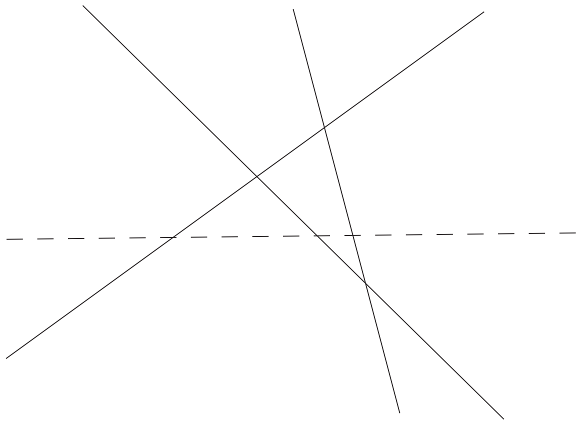
To start, we might try to discover the answer for smaller numbers of lines. To generalize the problem, we may ask for the number of regions produced by n lines, where n is some positive integer. It is fairly obvious that a single line (corresponding to the case $n = 1$) divides the plane into two regions. Two lines, if they are not parallel, can easily be seen to divide the plane into four regions. (Two parallel lines produce only three regions.) If we call the number of regions produced by n lines, no two of which are parallel and no three of which are coincident, R_n, then $R_1 = 2$ and $R_2 = 4$.

Figure 8.1: Three lines dividing the plane



What does the situation look like when $n = 3$? Figure 8.1 is representative of this situation. In this case, the number of regions is 7, so $R_3 = 7$. To find R_4, we must add a fourth line to the diagram. This suggests trying to compute R_4 in terms of R_3 so that we begin to think of R_n as a recurrence relation. Figure 8.2 shows what the situation looks like when a fourth line is added to the three existing lines. From the assumptions that no two of the lines are parallel and no three pass through a single point, it follows that the new line must intersect each of the existing three lines in exactly one point. This means that the new line passes through exactly four of the regions formed by the original three lines. Each region that it passes through is divided into two regions, so the total number of new regions added by the fourth point is 4. Thus, $R_4 = R_3 + 4$. Similar arguments for a general configuration of lines reveals that R_n satisfies the recurrence relation $R_n = R_{n-1} + n$.

Figure 8.2: Four lines dividing the plane



Furthermore, we have already computed the initial condition $R_1 = 2$. This is enough to solve the recurrence.

```
In[77]:=  Clear[R];
          R=R/.First[RSolve[{R[n]==R[n-1]+n,R[1]==2},R,n]]

Out[78]=  Function[{n},1/2 (2+n+n^2)]
```

To answer the question: how many regions is the plane divided by 1000 lines with no two parallel and no three coincident?

```
In[79]:=  R[1000]

Out[79]=  500501
```

8.3 Divide-and-Conquer Algorithms and Recurrence Relations

A very good example of divide and conquer relations is the one provided by the binary search algorithm. Here, we consider a practical application of this algorithm in an implementation of a binary search on a sorted list of integers. This is the implementation of the algorithm described in Algorithm 3 in Section 3.1 of the textbook and first presented in Section 3.1 of this manual.

```
In[80]:=  binarySearch[x_Integer,A:{__Integer}]/;Less@@A:=
            Module[{n,i,j,m,location},
              n=Length[A];
              i=1;
              j=n;
              While[i<j,
                m=Floor[(i+j)/2];
                If[x>A[[m]],
                  i=m+1,
                  j=m
                ]
              ];
              If[x==A[[i]],
                location=i,
                location=0
              ];
              location
            ]
```

The variable `A` is the list of integers to search, which is assumed to be sorted in increasing order, and `x` is the integer to search for. The local variables `j` and `i` are initialized to the number of elements in the list and 1, respectively. The `While` loop continues as long as `i` and `j` are different from each other. Each step of the loop serves to narrow the difference between them by calculating the middle of the list, represented by `m`, and determining which half `x` is in. Eventually, the search will focus on one location in the list, which is either `x` or, if not, the search has failed and the function returns 0.

Let us do the analysis of the algorithm to see how divide and conquer recurrence relations are generated. In general, a divide and conquer type recurrence relation has the form

$$f(n) = a \cdot f(n/b) + g(n)$$

Each iteration of the `While` loop of `binarySearch` produces a single list half the size of the original. Thus, $a = 1$ and $b = 2$. The function $g(n)$, which measures the number of comparisons added in performing the reduction, is identically 2. This is because one comparison is added to see which half of the list the key is on, and one is added to see if the `While` loop needs to continue. Therefore, for the `binarySearch` algorithm, the recurrence relation is

$$f(n) = f(n/2) + 2$$

Additionally, we can see that $f(1) = 2$, because if the list is of length 1, then the algorithm will do one comparison to determine that the `While` loop is unnecessary and one comparison to make sure that the element being searched for is the one element in the list. We can now use `RSolve` to solve this recurrence.

```
In[81]:=  RSolve[{b[n]==b[n/2]+2,b[1]==2},b[n],n]

                          Log[n]
Out[81]=  {{b[n]→2  (1+ ──────── )}}
                          Log[2]
```

8.4 Generating Functions

Generating functions are a powerful tool for manipulating sequences of numbers and for solving a variety of counting problems. In this section, we will see how *Mathematica* can be used to represent and manipulate generating functions.

The generating function $G(x)$ for a sequence $\{a_k\}$ is the formal power series

$$\sum_{k=0}^{\infty} a_k x^k = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_k x^k + \cdots$$

It is called formal because we are not interested in evaluating it as a function of x. Our focus is on finding a formula for its coefficients. In particular, this means that there are no convergence issues to be considered.

Generating Functions Tools

The Wolfram Language provides extensive facilities for manipulating generating functions.

The first thing we need to do is to learn how to create a power series in the Wolfram Language, which is done with the function `GeneratingFunction`. This function requires three arguments. The last argument is the variable, such as x, that the generating function is to be written in terms of. The second argument is a variable, such as n, representing the index of summation. The first argument is an expression in terms of the index of summation that computes the coefficients of the series.

For example, to create the generating function for the sequence $\{3^k\}$, we use the following code.

```
In[82]:=  GeneratingFunction[3^k,k,x]

            1
Out[82]=  ─────
          1-3 x
```

Observe that *Mathematica* has automatically found a rational expression for the generating function. While this is a useful and exact representation of the generating function, you often want to be able to view the generating function as a power series.

To obtain a series representation of the generating function, we use the `Series` function. `Series` is used to find the Taylor series of a function about a point. If you have taken Calculus, you may remember Taylor series. If not, it is enough to know that the Taylor series of a generating function around $x = 0$ is the correct power series for our purposes.

The `Series` function requires two arguments. The first is the function, for example, $\frac{1}{1-3x}$. The second is a list of length three: the variable x, the number 0, and the largest order, or exponent, to be displayed. For example, to display the first ten terms of the power series for $\frac{1}{1-3x}$, we enter the following.

```
In[83]:=  seriesEx=Series[1/(1-3x),{x,0,10}]

Out[83]=  1+3 x+9 x^2+27 x^3+81 x^4+243 x^5+729 x^6+
            2187 x^7+6561 x^8+19683 x^9+59049
        x^10+O[x]^11
```

The output ends with $\boldsymbol{O[x]^{11}}$, which indicates that the series continues with terms of degree 11 and higher.

If you wish to view the coefficients, without the generating function, you can apply `CoefficientList`, as shown below. The first argument is the series and the second is the variable.

```
In[84]:=  CoefficientList[seriesEx,x]

Out[84]=  {1,3,9,27,81,243,729,2187,6561,19683,59049}
```

To create a power series from a recurrence relation, we need to combine the techniques of the last section for using `RSolve` to solve a recurrence relation with the `GeneratingFunction` command. For example, consider the recurrence relation given by $a_n = 2a_{n-1} - a_{n-2} + 1$ with initial conditions $a_0 = 1$ and $a_1 = 2$.

First, we apply the `RSolve` function and extract an expression for the general term of the sequence. Note that because we are interested in an expression that we can use in `GeneratingFunction`, it makes sense to solve for `a[n]` rather than solving for `a` and obtaining a pure `Function`. Either approach would work, however.

```
In[85]:=  recurrenceFormula=a[n]/.First[
            RSolve[{a[n]==2*a[n-1]-a[n-2]+1,a[0]==1,a[1]==2},
              a[n],n]
          ]

Out[85]=  1/2 (2+n+n^2)
```

Now, we apply `GeneratingFunction` to this formula.

```
In[86]:=  recurrenceGFunction=
            GeneratingFunction[recurrenceFormula,n,x]

Out[86]=  (-1+x-x^2)/(-1+x)^3
```

As above, we can use `Series` to view this as a formal power series.

```
In[87]:=  recurrenceSeries=Series[recurrenceGFunction,{x,0,5}]

Out[87]=  1+2 x+4 x^2+7 x^3+11 x^4+16 x^5+O[x]^6
```

We can also use the function `SeriesCoefficient` to obtain specific coefficients, rather than the entire series. This function has two distinct forms that are useful. First, you can give the output of `Series` as the first argument and an integer as the second, and `SeriesCoefficient` will return the coefficient of that term. Note that for this to work, you must have already computed the series to the desired term. That is, the second argument of `SeriesCoefficient` must be no greater than the order of the output from `Series`.

```
In[88]:=  SeriesCoefficient[recurrenceSeries,3]

Out[88]=  7
```

The second, and even more useful, application of `SeriesCoefficient` has the same arguments as `Series`. Specifically, a generating function and a list consisting of the variable used in the function, the number 0, and a positive integer n. However, where `Series` outputs the entire series out to the term of degree n, `SeriesCoefficient` just returns the coefficient of x^n.

```
In[89]:=  SeriesCoefficient[recurrenceGFunction,{x,0,3}]

Out[89]=  7
```

Solving Problems with Generating Functions

Generating functions are more than just a convenient way to represent numerical sequences. They are a powerful tool for solving recurrence relations, as well as other kinds of counting problems. This power stems from our ability to manipulate them like ordinary power series from Calculus and to interpret those manipulations. To illustrate *Mathematica*'s facilities for manipulating generating functions, consider Example 12 from Section 8.4 of the text:

> Use generating functions to determine the number of ways to insert tokens worth \$1, \$2, and \$5 into a vending machine to pay for an item that costs 7 dollars in both the cases when the order in which the tokens are inserted does not matter and when the order does matter.

Following the text, the solution to the problem when order does matter is the coefficient of x^r in the generating function

$$\left(1+x+x^2+x^3+\cdots\right)\left(1+x^2+x^4+x^6+\cdots\right)\left(1+x^5+x^{10}+x^{15}+\cdots\right)$$

To solve the problem, we need to create the three power series and multiply them together. To create the first series, we can use the `GeneratingFunction` command demonstrated above. For example, the first series has every coefficient equal to 1, so its generating function can be found as shown below.

```
In[90]:=  token1D=GeneratingFunction[1,n,x]

           1
Out[90]=  ───
          1-x
```

For the \$2 tokens, we need to represent $1+x^2+x^4+\cdots$. Remember that `GeneratingFunction` requires an expression for the general coefficient as its first argument. In this case, it is easier to describe the series if we can include the entire term, not just the coefficients. That is, it is easier to say that the series is $\sum_{k=0}^{\infty} x^{2k}$ than to write a formula for the coefficients.

Fortunately, we can do exactly that by applying the `Sum` function. Recall that the first argument to `Sum` is an expression for the generic term in terms of an index of summation, and the second argument defines the bounds of the summation. In this case, those bounds are 0 and `Infinity`.

```
In[91]:=    token2D=Sum[x^(2*k),{k,0,Infinity}]
```

```
Out[91]=    1/(1-x^2)
```

$$\frac{1}{1-x^2}$$

Observe that *Mathematica* has automatically calculated the rational expression for this sum.

We could also use `Sum` for the \$5 tokens. Alternatively, we could just refer to Table 1 in Section 8.4 of the text and enter the rational form $\frac{1}{1-x^5}$ directly.

```
In[92]:=    token5D=1/(1-x^5)
```

```
Out[92]=
```

$$\frac{1}{1-x^5}$$

We can use `Series` to confirm that *Mathematica* does in fact recognize this as the generating function $1+x^5+x^{10}+x^{15}+\cdots$.

```
In[93]:=    Series[token5D,{x,0,20}]
```

```
Out[93]=
```

$$1+x^5+x^{10}+x^{15}+x^{20}+O[x]^{21}$$

We algebraically combine the series with the usual multiplication and `Series` will show us the power series expansion of the result.

```
In[94]:=    tokens=token1D*token2D*token5D
```

```
Out[94]=
```

$$\frac{1}{(1-x)\ (1-x^2)\ (1-x^5)}$$

```
In[95]:=    Series[tokens,{x,0,7}]
```

```
Out[95]=
```

$$1+x+2\ x^2+2\ x^3+3\ x^4+4\ x^5+5\ x^6+6\ x^7+O[x]^8$$

We can see from the above that there are 6 ways to pay for a \$7 item (since the coefficient of x^7 is 6), just as was computed in the text. If we wanted to know the number of ways to pay for an item costing \$234, all we would need to do is find the coefficient of x^{234}.

```
In[96]:=    SeriesCoefficient[tokens,{x,0,234}]

Out[96]=    2832
```

For the second part of the question, the case where the order does matter, the text explains that the generating function we need is

$$1+\left(x+x^2+x^5\right)+\left(x+x^2+x^5\right)^2+\cdots=\frac{1}{1-(x+x^2+x^5)}$$

Again, if we did not already know the rational expression for the generating function, *Mathematica* could find it using `Sum`.

```
In[97]:=    tokens2=Sum[(x+x^2+x^5)^n,{n,0,Infinity}]
```

```
Out[97]=
```

$$\frac{1}{1-x-x^2-x^5}$$

```
In[98]:=  SeriesCoefficient[tokens2,{x,0,7}]

Out[98]=  26
```

Thus, the coefficient of x^7 is 26, so there are 26 ways to pay for a $7 item when order does matter.

8.5 Inclusion–Exclusion

In this section, we will apply the principle of inclusion and exclusion. At the heart of the principle of inclusion and exclusion is the formula

$$|A \cup B| = |A| + |B| - |A \cap B|$$

which says that, for two finite sets A and B, the number of elements in the union $A \cup B$ of the two sets may be found by adding the sizes of A and B and then subtracting the number of elements common to both A and B, which would otherwise be counted twice. This formula can be generalized to count the number of elements in the union of any finite number of finite sets.

Recall that in the Wolfram Language, sets are represented as lists, but that the set operations put elements in a canonical order and remove duplicates.

```
In[99]:=  A={1,2,3}

Out[99]=  {1,2,3}
```

To find the cardinality of a set, you can use `Length`.

```
In[100]:=  Length[A]

Out[100]=  3
```

The set operations `Union` and `Intersection` can be applied in functional form, with each set an argument. You can also access operator forms with the aliases $\boxed{\text{ESC}}$un$\boxed{\text{ESC}}$ and $\boxed{\text{ESC}}$inter$\boxed{\text{ESC}}$.

```
In[101]:=  X={1,2,3,4,5};
           Y={4,5,6,7,8};

In[103]:=  Union[X,Y]

Out[103]=  {1,2,3,4,5,6,7,8}

In[104]:=  X⋃Y

Out[104]=  {1,2,3,4,5,6,7,8}

In[105]:=  Intersection[X,Y]

Out[105]=  {4,5}
```

The set theoretic difference is computed by the *Mathematica* function `Complement`. The following computes $X - Y$.

```
In[106]:=  Complement[X,Y]

Out[106]=  {1,2,3}
```

Let us use the operations to illustrate the principle of inclusion and exclusion with a particular example.

```
In[107]:=  flintstones={"Fred","Wilma","Pebbles"};
           rubbles={"Barney","Betty","Bam Bam"};
           husbands={"Fred","Barney"};
           wives={"Wilma","Betty"};
           kids={"Pebbles","Bam Bam"};
```

If this were a complete census, then the number of children living in Bedrock would be

```
In[112]:=  Length[kids]

Out[112]=  2
```

The number of Bedrock inhabitants who are either Flintstones or children is

```
In[113]:=  Length[Union[flintstones,kids]]

Out[113]=  4
```

According to the principle of inclusion and exclusion, this number should be the same as

```
In[114]:=  Length[flintstones]+Length[kids]-
             Length[Intersection[flintstones,kids]]

Out[114]=  4
```

which, of course, it is.

As another example, consider the problem of determining the number of positive integers less than or equal to 100 that are not divisible by either 2 or 11. First, we generate the set of positive integers less than or equal to 100.

```
In[115]:=  hundred=Range[100]

Out[115]=  {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
             17,18,19,20,21,22,23,24,25,26,27,28,29,30,
             31,32,33,34,35,36,37,38,39,40,41,42,43,44,
             45,46,47,48,49,50,51,52,53,54,55,56,57,58,
             59,60,61,62,63,64,65,66,67,68,69,70,71,72,
             73,74,75,76,77,78,79,80,81,82,83,84,85,86,
             87,88,89,90,91,92,93,94,95,96,97,98,99,100}
```

Next, we remove those elements that are divisible by 2:

```
In[116]:=  divBy2=Complement[hundred,Table[2*i,{i,50}]]

Out[116]=  {1,3,5,7,9,11,13,15,17,19,21,23,25,
             27,29,31,33,35,37,39,41,43,45,47,49,
             51,53,55,57,59,61,63,65,67,69,71,73,
             75,77,79,81,83,85,87,89,91,93,95,97,99}
```

and those that are divisible by 11:

```
In[117]:=  divBy11=Complement[hundred,Table[11*i,{i,9}]]

Out[117]=  {1,2,3,4,5,6,7,8,9,10,12,13,14,15,16,17,18,19,
             20,21,23,24,25,26,27,28,29,30,31,32,34,35,36,
             37,38,39,40,41,42,43,45,46,47,48,49,50,51,52,
             53,54,56,57,58,59,60,61,62,63,64,65,67,68,69,
             70,71,72,73,74,75,76,78,79,80,81,82,83,84,
             85,86,87,89,90,91,92,93,94,95,96,97,98,100}
```

We are looking for integers that belong to either or both of those sets, that is, to their union, so we calculate:

```
In[118]:=  Length[Union[divBy2,divBy11]]

Out[118]=  96
```

According to the principle of inclusion and exclusion, this value could also be computed as

```
In[119]:=  Length[divBy2]+Length[divBy11]-
             Length[Intersection[divBy2,divBy11]]

Out[119]=  96
```

8.6 Applications of Inclusion–Exclusion

In this section, we will explore the following problem: Three sets of twins, Ashley and Amanda Abel, Brandon and Benjamin Bernoulli, and Christopher and Courtney Cartan (none of whom bear any relation to the mathematicians with the same surname), are to be seated in a row. List the ways in which they can be seated so that no person sits next to their twin.

The principle of inclusion–exclusion gives us insight into how we might accomplish this task. Rather than attempting to generate only those seating arrangements in which no person sits next to their twin, it will be easier to consider all the possible arrangements of the twins and then exclude those that do not satisfy the condition. To begin, we define lists to store the names of the twins.

```
In[120]:=  abels={"Ashley","Amanda"};
           bernoullis={"Brandon","Benjamin"};
           cartans={"Christopher","Courtney"};

In[123]:=  twins={abels,bernoullis,cartans}

Out[123]=  {{Ashley,Amanda},
             {Brandon,Benjamin},{Christopher,Courtney}}
```

Next, we create a function to test whether an arrangement satisfies the condition of having no twins seated next to each other. We will use this function to determine which seatings to accept.

To test whether a given arrangement has a pair of twins seated next to one another, we will consider the five pairs of seats, 1 and 2, 2 and 3, ..., 5 and 6, and check to see if the people seated in those positions are twins.

```
In[124]:=  testSeating[seating_List]:=Module[{i,twinpair},
             Catch[
               For[i=1,i≤5,i++,
                 Do[If[MemberQ[twinpair,seating[[i]]]&&
                    MemberQ[twinpair,seating[[i+1]]],Throw[False]]
                   ,{twinpair,twins}]
               ];
               Throw[True]
             ]
           ]
```

The function is passed a list of the six people's names, representing a seating, so that the person in the third seat is `seating[[3]]`. The `For` loop indexed by `i` goes through the five pairs of seats. The inner `Do` loop avoids having to duplicate the `If` statement. We could have written one `If` statement checking to see if the people in seats i and $i+1$ are both Abels, and then a second `If` statement to see if they are both Bernoullis, and then a third to see if they are both Cartans. Instead, the loop sets the `twinpair` variable to each of the lists in `twins` in turn. That is, `twinpair` represents, at each step in the loop, one of the families. Then, the `If` statement checks to see whether the people in the seats i and $i+1$ are members of that family, using the `MemberQ` test. Recall that `MemberQ` returns true if the second argument is a member of the first. If any of these `If` statements are true, that the people in the pair of consecutive seats are from the same family, then false is thrown to the enclosing `Catch`, indicating that the seating is not acceptable. If the seating survives all of the `If` statements, then the function returns true.

To check the `testSeating` function, consider the following potential seatings.

```
In[125]:=  seating1={"Ashley","Amanda","Brandon","Benjamin",
             "Christopher","Courtnet"};
           seating2={"Ashley","Brandon","Christopher","Amanda",
             "Benjamin","Courtney"};
```

We see that the first seating fails but the second passes, as they should.

```
In[127]:=  testSeating[seating1]

Out[127]=  False

In[128]:=  testSeating[seating2]

Out[128]=  True
```

We now have a function to test a potential seating for the condition of not having twins seated next to each other. To generate a list of all of such seatings, we will use the Wolfram Language's `Permutations` function to generate all the possible permutations of the people and then test to see which are valid and which should be discarded. The `Permutations` command takes a list and returns all the possible permutations of the objects. (Refer to Chapter 6 of this manual for more information about this function.)

Note that the `Permutations` function expects a list of objects, so we will need a list of all the people. We can use the `Flatten` function to turn our `twins` list into a list of all the names. `Flatten` takes a list and removes any nesting of lists so that the result is a list of the objects.

```
In[129]:=  Flatten[twins]

Out[129]=  {Ashley,Amanda,Brandon,Benjamin,Christopher,Courtney}
```

To create the list of only the valid seatings, we will apply the `Select` function. `Select` requires two arguments. The first is a list of objects, and the second is a function of one argument that returns true for those elements of the list that satisfy the desired criterion. The result is the list of the elements that passed the test.

We apply `Select` to all of the permutations of the flattened list of twins with criterion function `testSeating` and store its output in the variable `twinSeatings` but suppress the output. Then, we use `Length` to check how many possible seatings there are. (It is generally a good idea to suppress the output of a function that is listing what may be a very large number of possibilities until you know how many there are, as the output may take some time to display.)

```
In[130]:=  twinSeatings=Select[Permutations[Flatten[twins]],
             testSeating];

In[131]:=  Length[twinSeatings]

Out[131]=  240

In[132]:=  twinSeatings[[123]]

Out[132]=  {Benjamin,Ashley,Christopher,Amanda,Brandon,Courtney}
```

We see that there are 240 possible seatings and have displayed the 123rd seating.

Solutions to Computer Projects and Computations and Explorations

Computer Projects 12

Given positive integers m and n, find the number of onto functions from a set with m elements to a set with n elements.

Solution: We have a very convenient formula:

$$\sum_{k=0}^{n-1}(-1)^k C(n,k)(n-k)^m$$

This is the number of onto functions from a set of m elements to a set of n elements, assuming $m \geq n$. This formula is derived in the textbook from the principle of inclusion–exclusion (see Theorem 1 of Section 8.6). The only input required in this formula are the integer parameters m and n, which represent the sizes of the domain and codomain, respectively. The Wolfram Language's `Sum` function will compute summations such as the one above. The first argument is an expression in terms of an index of summation (e.g., k), and the second is of the form `{k,a,b}` indicating the bounds of the summation. For example, to compute

$$\sum_{k=3}^{8}\frac{1}{k}$$

you would enter the following expressions.

```
In[133]:=  Sum[1/k,{k,3,8}]
           341
Out[133]=  ───
           280
```

We now create a function encapsulating the formula above.

```
In[134]:=  ontoFunctions[m_Integer,n_Integer]/;m>0&&n>0:=
             If[m≥n,Sum[(-1)^k * Binomial[n,k]*(n-k)^m,
               {k,0,n-1}],0]
```

The `If` statement captures the fact that there are no onto functions from a set to a set that is larger and ensures that the result is 0 in that case. The `Condition` (`/;`) ensures that the input to the function is positive and will return no output if not, since it is meaningless to ask about functions between sets of nonpositive cardinalities.

```
In[135]:=  ontoFunctions[4,9]

Out[135]=  0

In[136]:=  ontoFunctions[-3,0]

Out[136]=  ontoFunctions[-3,0]
```

As an example, we can use our function to compute the number of onto functions from a set with 100 elements to a set with 20 elements.

```
In[137]:=  ontoFunctions[100,20]

Out[137]=  1123819591031965792853944703814317028551789497509576949629431
           9007413091913959828334936464196298192508890182316163261067934
           269440000
```

Computations and Explorations 2

Find the smallest Fibonacci number greater than 1000000, greater than 1000000000, and greater than 1000000000000.

Solution: We can solve this quite easily with *Mathematica* using a simple `While` loop. In this chapter, we have seen several ways to compute Fibonacci numbers, including the `fibonacci` function we created in Section 8.1 and the formula `fibonacci2` in Section 8.2. For this exercise, we will use the Wolfram Language built-in function `Fibonacci`.

The idea is to compute Fibonacci numbers until the value exceeds the target. The `While` loop is well suited to this sort of problem. We will create a function that takes the target value as input and prints out the desired Fibonacci number and its index.

```
In[138]:=  findFib[target_Integer]:=Module[{n=1},
             While[Fibonacci[n]<target,n++];
             Print["The ",n,"th Fibonacci number is ",
               Fibonacci[n]]
           ]
```

As long as the *n*th Fibonacci number is smaller than the target value, the index *n* is increased. Once the target has been exceeded, the `Print` statement displays the index and the value of the Fibonacci number.

The numbers called for by the question are:

```
In[139]:=   findFib[1000000]

            The 31th Fibonacci number is 1346269

In[140]:=   findFib[1000000000]

            The 45th Fibonacci number is 1134903170

In[141]:=   findFib[1000000000000]

            The 60th Fibonacci number is 1548008755920
```

Computations and Explorations 3

Find as many prime Fibonacci numbers as you can. It is unknown whether there are infinitely many of these.

Solution: Using *Mathematica*, this sort of problem becomes fairly straightforward. We can simply use the Wolfram Language function `Fibonacci` to generate Fibonacci numbers and use the `PrimeQ` function to test each for primality. We will wrap this in a function that takes a number of seconds as an argument and uses `TimeConstrained` to control the length of the evaluation. Note the use of `Sow` within the `TimeConstrained` block and `Reap` surrounding it. This allows the termination of the `TimeConstrained` portion of the function to trigger the `Reap`.

```
In[142]:=   primeFib[time_]:=Module[{i=0,temp,primes={}},
              Reap[
                TimeConstrained[
                  While[True,
                    i++;
                    temp=Fibonacci[i];
                    If[PrimeQ[temp],Sow[temp]]
                ],time]
              ][[2,1]]
            ]
```

We can obtain several examples even in only a hundredth of a second.

```
In[143]:=   primeFib[0.01]

Out[143]=   {2,3,5,13,89,233,1597,28657,514229,
              433494437,2971215073,99194853094755497,
              1066340417491710595814572169,
              19134702400093278081449423917}
```

Computations and Explorations 11

Compute the probability that a permutation of n objects is a derangement for all positive integers not exceeding 20 and determine how quickly these probabilities approach the number $1/e$.

Solution: To solve this problem, we will make use of the formula which gives the number of derangements of n objects, namely,

$$D_n = n!\left[1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \ldots + (-1)^n \frac{1}{n!}\right]$$

The total number of permutations of n objects is, of course, $n!$; so, the probability that one of them is a derangement is the ratio $\frac{D_n}{n!}$, which is given by the expression

$$1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \ldots + (-1)^n \frac{1}{n!}$$

A very simple *Mathematica* function will compute these values for us.

```
In[144]:=  derangementP[n_Integer]/;n>0:=
             Sum[(-1)^k*1/k!,{k,0,n}]
```

The probabilities that a permutation of n objects is a derangement for $n \leq 20$ are:

```
In[145]:=  Table[derangementP[n],{n,20}]

Out[145]=  {0, 1/2, 1/3, 3/8, 11/30, 53/144, 103/280, 2119/5760, 16687/45360, 16481/44800,
             1468457/3991680, 16019531/43545600, 63633137/172972800, 2467007773/6706022400,
             34361893981/93405312000, 15549624751/42268262400, 8178130767479/22230464256000,
             138547156531409/376610217984000, 92079694567171/250298560512000, 4282366656425369/11640679464960000}
```

To see how these probabilities differ from $1/e$, we multiply them by e and subtract 1. The symbol `E` represents the number e in the Wolfram Language. We apply `N` with second argument 10 in order to obtain numerical approximations with 10 digits of precision.

```
In[146]:=  Table[N[E*derangementP[n]-1,10],{n,20}]

Out[146]=  {-1.000000000,0.3591409142,-0.09390605718,
             0.01935568567,-0.003296662898,0.0004787285301,
             -0.00006061310257,6.804601513×10^-6,-6.862544954×10^-7,
             6.283110546×10^-8,-5.267585531×10^-9,4.073053851×10^-10,
             -2.922468532×10^-11,1.956033996×10^-12,
             -1.226806251×10^-13,7.239038692×10^-15,-4.032944742×10^-16,
             2.127959055×10^-17,-1.066412864×10^-18,5.088730674×10^-20}
```

Exercises

1. Implement a function to find the optimal schedule that maximizes total attendance.
2. Implement a dynamic programming algorithm for finding the maximum sum of consecutive terms of a sequence of real numbers. (See Exercise 56 in Section 8.1.)
3. Implement a dynamic programming algorithm for optimally computing matrix-chain multiplication. (See Exercise 57 in Section 8.1.)
4. Use *Mathematica* to solve the following recurrence relations.

   a. $a_n = a_{n-1} - a_{n-2}$, $a_1 = 1$, $a_2 = 1$;
   b. $a_n = 15a_{n-1} + \frac{1}{2}a_{n-2}$, $a_1 = \frac{23}{22}$, $a_2 = \frac{7}{2}$
5. Solve each of the recurrence relations in Exercise 1 in Section 8.2 of the textbook. (Solve even those that are not linear homogeneous recurrence relations with constant coefficients.)
6. Write a general solver in the Wolfram Language for linear homogeneous recurrence relations with constant coefficients of degree 3 with distinct roots. Your solver should check that the roots are in fact distinct and, if they are not, should issue a `Message` and return `$Failed` , which is a standard return value for a Wolfram Language function when it cannot complete a computation for some reason.
7. Use *Mathematica* to investigate the behavior of the limit

$$\lim_{n\to\infty} \frac{\varphi_n}{\psi_n}$$

   where φ_n is defined to be the number of prime Fibonacci numbers less than or equal to n, and ψ_n is defined to be the number of Fermat numbers less than or equal to n.
8. Implement the recursive algorithm described in Example 12 of Section 8.3 of the textbook for solving the closest-pair problem.
9. Use *Mathematica* to find the number of square-free integers less than 100000000.
10. Use *Mathematica* to find the number of onto functions from a set with 1000000 elements to a set with 1000 elements.
11. It is probably obvious that the number of onto functions from one set to another increases with the sizes of either the domain or the range. Using *Mathematica* to experiment, explore whether an increase in the size of the domain or the size of the range has the greater impact on the number of onto functions.
12. To generate the *lucky numbers*, start with the positive integers and delete every second integer in the list, starting the count with 1 (e.g., delete 2, 4, 6, etc., leaving 1, 3, 5, 7, ...). Other than 1, the smallest integer left is 3. Continue by deleting every third integer from those that remain, starting the counting with 1 (since 1, 3, 5, 7, 9, ... remain, 1 is the first number left, 3 is the second one left, 5 is the third left and gets deleted, and so on). Continue the process where at each stage, every kth integer is deleted, where k is the smallest integer left, other than 1 and the previous values of k. The integers that remain are the lucky numbers. Develop a function in the Wolfram Language that generates the lucky numbers up to n.

13. Can you make any conjectures about lucky numbers by looking at a list of the first 1000 of them? For example, what sort of conjectures can you make about twin lucky numbers? What evidence do you have for your conjectures?

14. Generalize the `listSeatings` function to accept one argument, a list of lists (the same structure as the twins list), and determines the arrangements such that no two from the same sublist are seated next to one another.

15. Further generalize the `listSeatings` function so that it takes two arguments: a list of lists as before and a number *n*. The function should determine the arrangements of the people such that no *n* from the same sublist are seated together.