6 Counting

Introduction

This chapter presents a variety of techniques that are available in *Mathematica* for counting a diverse collection of discrete objects, including combinations and permutations of finite sets. Objects can be counted using formulae or by using algorithms to list the objects and then directly counting the size of the list.

6.1 The Basics of Counting

In this section we will see how *Mathematica* can be used to perform the computations needed to solve basic counting problems. We will begin by looking at some examples. We will discuss computations involving large integers. Then we'll see how the principles of counting can be used to count the number of operations used by a *Mathematica* function. This section concludes by using *Mathematica* functions to solve counting problems by enumerating all the possibilities.

Basic Examples

We begin with two basic examples to demonstrate the use of some useful *Mathematica* functions.

Counting One-to-one Functions

Recall Example 7 from Section 6.1 of the text. That example calculated that the number of one-to-one functions from a set with m elements to a set with n elements is

$$n(n-1)(n-2)\cdots(n-m+1)$$

Note that we can rewrite this using product notation as

$$\prod_{i=0}^{m-1} n-i$$

For small values of m, it is easy to enter this product in *Mathematica*. For instance, the expression below computes the number of one-to-one functions from a set of 4 elements to a set of 20 elements.

```
In[1]:= 20 * 19 * 18 * 17
Out[1]= 116 280
```

For larger values of m, it is more convenient to use the `Product` function. The `Product` function is used to multiply a sequence of values. Its syntax is identical to that of `Sum`. The first argument is an expression in terms of a variable (e.g., i) that evaluates to the values that are to be multiplied together.

The second argument can take a variety of forms. One usage has the form **{*i*, *a*, *b*}** which indicates that the variable ***i*** is to range from a minimum of ***a*** to a maximum of ***b***. If the minimum value is to be 1, then ***a*** can be omitted.

For example, we can recompute the number of one-to-one functions from a set of 20 elements to a set of 4 elements as follows.

```
In[2]:= Product[20 - i, {i, 0, 3}]
Out[2]= 116 280
```

The second argument indicates that the index variable will be assigned the integers 0, 1, 2 and 3. These are then substituted into the first argument, **20 - i**, producing the values 20, 19, 18, and 17, which are multiplied together.

We can easily compute the number of one-to-one functions from a set of 12 elements to a set of 300 elements.

```
In[3]:= Product[300 - i, {i, 0, 11}]
Out[3]= 425 270 752 192 695 317 567 218 560 000
```

Computer Passwords

Example 16 from Section 6.1 describes a computer system in which each user has a password that must be between six and eight characters long and consisting of uppercase letters or digits. Also, each password must contain at least one digit.

The solution to the example describes how to calculate this. For each password length, 6, 7, or 8, the number of passwords are

```
In[4]:= P6 = 36^6 - 26^6
Out[4]= 1 867 866 560
In[5]:= P7 = 36^7 - 26^7
Out[5]= 70 332 353 920
In[6]:= P8 = 36^8 - 26^8
Out[6]= 2 612 282 842 880
```

Thus the total number of possible passwords is

```
In[7]:= P = P6 + P7 + P8
Out[7]= 2 684 483 063 360
```

We can use the Sum function to perform this calculation in one step.

```
In[8]:= Sum[36^i - 26^i, {i, 6, 8}]
Out[8]= 2 684 483 063 360
```

This makes it easy to compute the number of valid passwords for larger ranges. For instance, it is common to require passwords to be between 8 and 16 characters. If we retain the rules that the password be uppercase letters or numbers and include at least one number, then the total number of possible passwords is calculated with the expression below.

```
In[9]:= Sum[36^i - 26^i, {i, 8, 16}]
Out[9]= 8 140 698 334 757 962 630 580 480
```

Working with Large Integers

Mathematica's computational engine is able to work with arbitrarily large integers, subject only to the limitations imposed by the computer's memory and speed.

DNA

In Example 11, the text provides a brief description of DNA and concludes that there are at least 4^{10^8} different possible sequences of bases in the DNA for complex organisms.

To have *Mathematica* compute this value, we just enter the expression and wait for it to complete the computation.

```
In[10]:= DNAsequences = 4^10^8
```

Out[10]=

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

```
13 576 763 067 144 082 109 859 764 943 426 215 110 923 900 940 995 100 ⁖
  861 638 392 126 957 291 410 917 608 911 489 552 935 725 380 527 448 ⁖
  262 970 047 404 815 772 410 095 615 122 011 337 654 446
 <<60 205 707>>
 434 758 009 705 913 813 453 625 377 772 169 404 221 062 381 750 849 ⁖
  126 730 846 432 378 801 419 793 910 906 892 709 552 426 541 101 314 ⁖
  285 829 832 336 140 162 991 461 016 516 356 273 787 109 376
```

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

Mathematica reports that the result is very large. The output shows the first several digits and the last several digits, with the number in between indicating the number of digits that were omitted. This does not imply that *Mathematica* has not computed the entire value, it only means that displaying the integer would require excessive space.

Mathematica has computed and stored the exact value and we can use this value in further computations. For example, we can find the last three digits of the number by computing the result modulo 1000.

```
In[11]:= Mod[DNAsequences, 1000]
Out[11]= 376
```

We can determine the number of digits in the result by applying the `IntegerLength` function. This function is slightly more efficient than computing the base 10 logarithm of the integer.

```
In[12]:= IntegerLength[DNAsequences]
Out[12]= 60 206 000
```

Remember that the approximation 4^{10^8} was a lower bound. In other words, the minimum number of

possible sequences of bases in the DNA of a complex organism has over 60 million digits.

Suppose you wanted to print this number. Using a typical fixed-width 12-point font and 1-inch margins, you can fit about 64 digits in each line and 45 lines on a page. With these parameters, it would require

```
In[13]:= N[IntegerLength[DNAsequences] / (64 * 45)]
Out[13]= 20 904.9
```

pages to print the entire number.

Symbol Names in Mathematica

Example 15 in Section 6.1 of the text calculated that in one version of the programming language BASIC, there were 957 different names for variables.

Mathematica is extremely flexible with regards to symbol names. For simplicity, we will say that a symbol consists of a letter possibly followed by additional letters or numbers. *Mathematica* considers uppercase distinct from lowercase. There are 52 uppercase or lowercase letters that can be used as the first character. With the ten digits included, there are 62 possibilities for each character following the first.

Mathematica does not impose a maximum length on variable names. For this example, however, we will consider only variable names up to a reasonable length of 15 characters.

How many possible symbols are there? We need to compute the sum $\sum_{i=0}^{14} 52 \cdot 62^i$. Apply the `Sum` function to calculate this value.

```
In[14]:= Sum[52 * 62^i, {i, 0, 14}]
Out[14]= 655 464 010 775 997 815 815 360 444
```

We see that even limiting ourselves to a maximum of 15 characters, there are over 650 septillion distinct *Mathematica* symbols. (The number above is slightly inaccurate because it does not exclude *Mathematica* keywords and other protected symbols that you are not allowed to assign values to. Of course, those are relatively insignificant.)

Counting Operations in a Function

Next we'll consider an example of counting the number of operations performed by a function. In Example 9 in Section 6.1 of the textbook, it is shown that the number of times that the innermost statement in a nested `For` loop is executed is the product of the number of iterations of each loop.

As an example of this, we'll consider the **`makePostage`** function from Section 5.1 of this manual. Recall that the purpose of this function was to determine the numbers of stamps of two given denominations that are required to make a given amount of postage. Here is the definition again.

```
In[15]:= makePostage[stampA_Integer, stampB_Integer, postage_Integer] :=
          Module[{a, b},
           Catch[
            For[a = 0, a ≤ Floor[postage / stampA], a++,
             For[b = 0, b ≤ Floor[postage / stampB], b++,
              If[stampA * a + stampB * b == postage, Throw[{a, b}]]
             ]
            ]
           ]
          ]
```

We will count the number of multiplications and additions that this function requires in the worst case. The Catch/Throw pair means that once the function has found a way to make the desired amount of postage, execution is immediately terminated. This means that knowing the number of iterations used for a particular input value is equivalent to knowing the output of the function. If there was a formula for that, we wouldn't need the function. By considering the worst-case scenario, we can get an idea of the complexity of the algorithm without having to execute the function.

The worst-case scenario, that is, the situation that requires the most number of iterations of the loop, occurs when the desired postage cannot be made. In this case, the outer loop variable will range from 0 to $\left\lfloor \frac{\text{postage}}{\text{stampA}} \right\rfloor$ and the inner loop will range from 0 to $\left\lfloor \frac{\text{postage}}{\text{stampB}} \right\rfloor$. Thus the number of times the if statement is executed is $\left(\left\lfloor \frac{\text{postage}}{\text{stampA}} \right\rfloor + 1\right)\left(\left\lfloor \frac{\text{postage}}{\text{stampB}} \right\rfloor + 1\right)$. Therefore, in the worst case, the **makePostage** function requires that number of additions and twice as many multiplications.

Counting by Listing All Possibilities

At the end of Section 6.1, the text discusses using tree diagrams to solve counting problems. Tree diagrams provide a visual way to organize information so you can be sure that you arrive at all possible results. We will not, in this section, implement trees, as that is the focus of Chapter 11. The goal of a tree diagram is to list all of the possibilities. In this subsection we will consider two problems that can be solved by using *Mathematica* functions to list all possibilities.

Subsets

For the first example, we consider the following question: how many subsets of the set of the integers 1 through 10 have sums less than 15? (This is similar to Exercise 67 in Section 6.1.)

To solve this problem, we'll consider all of the possible subsets and count those that satisfy the condition.

In order to generate all of the possible subsets of {1, 2, …, 10}, we'll use the Subsets function, first introduced in Section 2.1 of this manual.

The Subsets function accepts a list (representing a set) as an argument and produces the list of all subsets. Here is an example of Subsets applied to the set {1, 2}.

```
In[16]:= Subsets[{1, 2}]
Out[16]= {{}, {1}, {2}, {1, 2}}
```

We will use Subsets to solve the problem of counting the number of subsets of {1, 2, …, 10} whose

sum is less than 15. Instead of displaying all subsets, we'll instead test their sum using the Select function.

Select, when applied to a list and a Boolean-valued function, returns the sublist consisting of all elements of the original list that cause the function to return True. In our case, the first argument will be the output from Subsets. The second argument will be the pure function that adds the elements in the subset and compares the result to the target of 15.

To add the elements of a list, we will Apply (@@) the addition operator Plus (+) to the set, as illustrated below.

```
In[17]:= Apply[Plus, {1, 2, 3, 4}]

Out[17]= 10
```

The Apply function has the effect of replacing the head of the second argument with the first argument. In the example above, the expression {1, 2, 3, 4} has head List. Apply replaces List with Plus, transforming the expression into Plus[1, 2, 3, 4], which is then evaluated and outputs the sum. Apply can be used any time you need to use the elements of a list as the arguments to a function.

To compare the sum with 15, we just need to add the inequality to the expression.

```
In[18]:= Apply[Plus, {1, 2, 3, 4}] < 15

Out[18]= True
```

And to turn this example into a pure function, we replace the specific set with a Slot (#) and put an ampersand at the end to indicate its status as a Function (&).

```
Apply[Plus, #] < 15 &
```

With this as the second argument to Select, and an application of Subsets as the first argument, we obtain the list of all subsets with the specified sum. As an example, we list the subsets of {1, 2, 3} whose sum is less than 5.

```
In[19]:= Select[Subsets[{1, 2, 3}], Apply[Plus, #] < 5 &]

Out[19]= {{}, {1}, {2}, {3}, {1, 2}, {1, 3}}
```

The original question was to count the number of subsets of {1, 2, 3, …, 10} whose sum is at most 15. So all that remains is to apply Length in order to get a count of the number of subsets. We can generalize a bit and create a function that accepts a set of integers and a target value and counts the number of subsets of the given set whose elements sum to a value less than the target.

```
In[20]:= subsetSumCount[S : {__Integer}, target_Integer] :=
          Length[Select[Subsets[S], Apply[Plus, #] < target &]]
```

Applying this function to {1, 2, …, 10} and 15 will answer the original question.

```
In[21]:= subsetSumCount[Range[10], 15]

Out[21]= 99
```

Bit Strings

For the second example, we'll consider a problem similar to Example 21. How many bit strings of length 10 do not have three consecutive ones?

We could use an approach similar to the previous example and produce all bit strings of length 10 and then count the number that do not contain three consecutive 1s. However, the solution to Example 21 of Section 6.1, and especially Figure 2, suggests a more efficient solution. Instead of creating all the possible bit strings, we can build them in such a way as to only create those that satisfy the limitation on the number of consecutive 1s.

To solve this problem, we will use a recursive algorithm. The basis step will be the set consisting of all bit strings of length 2 (these cannot have three consecutive 1s). In the recursive step, the new version of the set will be constructed as follows. For each bit string in the previous set, we will append a 0. And we will append a 1 to all of the bit strings whose last two bits are not both 1.

For this problem, we will model bit strings as lists of 0s and 1s. The algorithm described above will be implemented as two functions. The first function will be responsible for extending a particular bit string. That is, given a bit string (a list of 0s and 1s), it will return either the two bit strings obtained by appending a 0 and by appending a 1, or it will return the single bit string formed by appending a 0 if appending a 1 would result in three consecutive 1s. The second function will apply the first to build the entire set of admissable bit strings.

First we implement the function that extends a single bit string. The parameter to this function will be a bit string, that is a list of 0s and 1s. The function will first create a new bit string by appending a zero to the input.

Then the function will test the last two elements of the original bit string to determine if they are both ones. We will accomplish this test by extracting the last two entries with the Part (**[[…]]**) operator. We use the fact that negative integers indicate the position from the end of the list: **L[[-1]]** is the last element, **L[[-2]]** is the next to last element, and so on. We use a Span (**;;**), i.e., **L[[-2 ;; -1]]**, so that we obtain the sublist of the last two elements. We can then compare the result against the list **{1, 1}**. If those lists are equal, that is, the last two bits are both 1, then the function only returns the list obtained by adding 0. Otherwise, it will create the list with 1 added and return both extended bit strings as a sequence.

Here is the implementation.

```
In[22]:= addBit[L_List] := Module[{L0, L1},
           L0 = Append[L, 0];
           If[L[[-2 ;; -1]] == {1, 1}, Return[{L0}],
            L1 = Append[L, 1];
            Return[{L0, L1}]
           ]
          ]
```

Let's test this function: {1, 0, 1, 1} should produce only {1, 0, 1, 1, 0}, while applying the function to that result should produce {1, 0, 1, 1, 0, 0} and {1, 0, 1, 1, 0, 1}.

```
In[23]:= addBit[{1, 0, 1, 1}]
Out[23]= {{1, 0, 1, 1, 0}}
In[24]:= addBit[{1, 0, 1, 1, 0}]
Out[24]= {{1, 0, 1, 1, 0, 0}, {1, 0, 1, 1, 0, 1}}
```

Now we write the main function. It will accept as input a positive integer *n* representing the length of the bit strings to be output. In case this value is 2, the output is the four bit strings of length 2. For

values of n larger than 2, it will recursively call itself on $n-1$ and store the result of the recursion as **S**. It then initializes a new list **T** to the empty set. Finally, it loops through the set **S**, applying **addBit** to each member and adding the result to **T**.

Here is the implementation.

```
In[25]:= findBitStrings[n_Integer] := Module[{S, s, T = {}},
           If[n == 2,
            {{0, 0}, {0, 1}, {1, 0}, {1, 1}},
            S = findBitStrings[n - 1];
            Do[T = Join[T, addBit[s]], {s, S}];
            T
           ]
          ]
```

Applying the function to 10 and using Length on the output will give us the number of bit strings of length 10 that do not include three successive 1s.

```
In[26]:= Length[findBitStrings[10]]

Out[26]= 504
```

6.2 The Pigeonhole Principle

In this section we will see how *Mathematica* can be used to help explore two problems related to the pigeonhole principle: finding consecutive entries in a sequence with a given sum, and finding increasing and decreasing subsequences.

Before considering those two problems, however, recall the Ceiling function. This function calculates the ceiling of an expression. For example, the solution to Example 8 in the text indicates that the minimum number of area codes needed to assign different phone numbers to 25 million phones is $\left\lceil \frac{25\,000\,000}{8\,000\,000} \right\rceil$. In *Mathematica*, this is computed by the following expression.

```
In[27]:= Ceiling[25 000 000 / 8 000 000]

Out[27]= 4
```

Consecutive Entries with a Given Sum

Example 10 in Section 6.2 describes the solution to the following problem. In a month with 30 days, a baseball team plays at least one game per day but at most 45 games during the month. There then must be a period of consecutive days during which the team plays exactly 14 games.

The problem can be generalized. Given a sequence of d positive integers whose sum is at most S, there must be a consecutive subsequence with sum T for any $T < S - d$. We leave it to the reader to prove this assertion.

We will write a function that, given a sequence and target sum T, will find the consecutive terms whose sum is T. Our solution will be based on the approach used in Example 10.

First, we will calculate the numbers $a_1, a_2, \ldots, a_d$ with each a_i equal to the sum of the first i terms in

the sequence. These values will be stored as a list, **A**. We will calculate these sums using the observation that each one is equal to the previous sum plus the next entry in the sequence.

Then we will calculate $a_i + T$ for each i and use the MemberQ and Position functions to check to see if this value is in **A**. Both functions requires two arguments: the first is the list to be searched and the second is the element being sought. MemberQ returns True or False depending on whether the object is or is not a member of the list. The Position function returns a list of position specifications. Consider the example below.

```
In[28]:= Position[{"a", "b", "c", "d", "b", "a", "a", "c"}, "a"]
Out[28]= {{1}, {6}, {7}}
```

Note that the result is a list of three sublists. Each element of the main list indicates the position at which an "a" occurs, namely positions 1, 6, and 7. The reason that the location specifications are enclosed in their own lists is for the case when the desired element is found within sublists of the original list. For example, in the example below, the character "c" is found both in the third element of the sublist at position 2 as well as in position 4.

```
In[29]:= Position[{"a", {"b", "a", "c", "d"}, "d", "c", "a"}, "c"]
Out[29]= {{2, 3}, {4}}
```

Our list **A** will be flat, that is, with no sublists. Also, we are only interested in finding the first match, since that will tell us the location of a sublist with the desired sum. To find the position of the first match, we use the Part (**[[…]]**) specification **[[1,1]]**.

Finally, if $a_i + T$ is found in the list $a_1, a_2, \ldots, a_d$, say at position j, then we know that i through j are the positions of the consecutive subsequence with the desired sum. The function will output the starting and ending positions as well as the subsequence.

Here is the function. Note that the first line uses the Table to update the value of **a**, which serves as the sum of the list elements, at the same time it populates the list **A** with those values.

```
In[30]:= findSubSum[L : {__Integer}, T_Integer] :=
          Module[{A, i, j, a = 0, p},
           A = Table[a = a + L[[i]], {i, Length[L]}];
           Catch[
            For[i = 1, i ≤ Length[L], i++,
             If[MemberQ[A, A[[i]] + T],
              j = Position[A, A[[i]] + T][[1, 1]];
              Throw[{i, j, L[[i ;; j]]}]
             ]
            ]
           ]
          ]
```

We apply the function to the following sequence, representing the number of games a baseball team played on each day of a 30-day month:

2, 1, 3, 1, 1, 3, 1, 1, 1, 1, 3, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 3, 1, 3, 1, 1

As in Example 10, we'll find the consecutive days during which the team played 14 games.

```
In[31]:= findSubSum[{2, 1, 3, 1, 1, 3, 1, 1, 1, 1, 3, 1, 3,
           1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 3, 1, 3, 1, 1}, 14]
Out[31]= {5, 13, {1, 3, 1, 1, 1, 1, 3, 1, 3}}
```

Strictly Increasing Subsequences

Theorem 3 of Section 6.2 asserts that every sequence of $n^2 + 1$ distinct real numbers contains a subsequence of length $n + 1$ that is either strictly increasing or strictly decreasing. We will develop a function that will find a longest strictly increasing subsequence.

The Patience Algorithm

To find the longest increasing subsequence, we will use a greedy strategy based on "Patience sorting" (the name refers to the solitaire card game also called Klondike). The idea is as follows. Imagine that the numbers in the sequence are written on cards. The cards are placed in a "deck" in the order they appear in the sequence and with the first element of the sequence on top. Now play a "game" using the deck of cards based on the following rules.

The cards are "dealt" one at a time onto a series of piles on the table. Initially there are no piles. The top card (the first element in the sequence) is the first card dealt and forms the first pile. To play the next card, check to see if it is less than or greater than the first card. If the second card (the second element of the sequence) is less than the first, then it is placed on the first pile, on top of the first card. If the second card is greater than the first, then it starts a new pile to the right of the first.

The "game" continues in this way. At each step, the table has on it a series of piles. To play the next card, you compare the value on the card to the card on top of the first pile. If the card to be played has a value smaller than the number showing on the first pile, then the new card is placed on top of the first pile. Otherwise, you look at the second pile. If the card being played is smaller than the value on the second pile, it is placed on top of the second pile. Continue in this fashion until either the card has been played or, if it is larger than the top card on every existing pile, then it begins a new pile to the right of all the others.

An illustration is in order. Consider the sequence 12, 18, 7, 11, 16, 3, 20, 17.

Step 1: Play the first entry, 12, as the first pile	12
Step 2: Play the second entry, 18. Since 18 > 12, 18 starts a new pile.	12 18
Step 3: Play 7. Checking the first pile, note that 7 < 12, so 7 is played on the first pile.	7 12 18
Step 4: Play 11. Checking the first pile, 11 > 7, so do not play 11 on first pile. Checking the second pile, 11 < 18, so play 11 on the second pile.	7 11 12 18
Step 5: Play 16. Checking the first pile, 16 > 7 so do not play 16 on the first pile. Checking the second pile, 16 > 11, so 16 begins a third pile.	7 11 12 18 16
Step 6: Play 3. Checking the first pile, 3 < 7, so 3 is played on the first pile.	3 7 11 12 18 16
Step 7: Play 20. Checking the first pile, 20 > 3. Checking the second pile, 20 > 11. Checking the third pile, 20 > 16, so 20 starts a new pile.	3 7 11 12 18 16 20
Step 8: Play 17. Checking the first pile, 17 > 3. Checking the second, 17 > 11. Checking the third, 17 > 16, but 17 < 20, so 17 is played on the fourth pile.	3 7 11 17 12 18 16 20

Once the "game" is complete, the length of the longest strictly increasing subsequence is equal to the number of piles.

Now that all of the cards are played, we obtain a strictly increasing subsequence by backtracking. The top card on the final pile is 17, so 17 will be the last entry in the subsequence. When 17 was placed on the pile, the top card on the pile before it was 16 (step 8), so 16 precedes 17 in the subsequence. When 16 was placed on the third pile, the top card on the second pile was 11 (step 5), so 11 precedes 16. And when 11 was placed on the second pile, the top card on the first pile was 7, so 7 is first in the subsequence. Thus, 7, 11, 16, 17 is a strictly increasing subsequence of maximal length.

You are encouraged to "play" through this approach a few times with your own sequences to ensure that you understand the process before continuing on to the implementation of the procedure below. You can apply the **`findIncreasing`** function defined below to make sure that you are arriving at the same result. Be sure to keep track of what was on top of the previous pile when each number is

played, as you need that information in the backtracking stage.

Implementing the Algorithm

Now we will implement the Patience algorithm. The given sequence will be input to the function as a list. Within the function, we need to track three kinds of information.

First, we need to know which "step" we're in, that is, which card is being played. This will be represented by the variable to a `For` loop ranging from 1 to the size of the list.

Second, we need to know what cards are on the piles. Specifically, we need to know the top card of each pile. This will be represented as a list, **`piles`**. When a card is placed on top of an existing pile, we can replace the current value in that position with the syntax **`piles[[i]] = x`**. When a new pile is added to the list, we extend the list via `AppendTo`.

Third, in order to backtrack and recover the longest increasing sequence, we need to store, for each member of the sequence, the value that was on the top of the pile to the left of the entry's pile. For this, we will use a indexed variable, **`pointers`**, whose indices will be the members of the sequence and whose values will be set to the previous pile's top card. (We use the name **`pointers`** for this variable because of the similarity to the "linked list" structure used in many programming languages.) For those numbers played in the first pile, the value in the variable will be set to `Null`. Note that since pointers will be an indexed variable, rather than a list, we use a single pair of brackets around its indices, not the double brackets used with `Part` (**`[[…]]`**) and lists.

The algorithm will consist of two stages. The first stage will be the game stage. We begin with a `For` loop with loop variable **`step`** running from 1 to the length of the input sequence **`S`**. Within this main loop, two tasks are performed.

The first thing that happens within the **`step`** loop is determining on which pile to play the current card. Note that the value of the current card is accessed by **`S[[step]]`**. To determine the proper location for the current card, we do the following.

1. Open a `Catch` block. A variable **`whichpile`** is set to the output of this `Catch`.
2. Use a `For` loop from 1 to the current number of piles. Within the loop, compare the current card to the top of each pile. If the current card is smaller than the value in a pile, `Throw` that pile index, causing the loop to be short-circuited and assigning that value to **`whichpile`**. If the `For` loop terminates, `Throw` `Null` to indicate that the card cannot be placed on any existing pile.
3. Next, check the value of **`whichpile`**. If it is `Null`, that means a pile was not found for the current card, and thus the **`piles`** list must be extended to create a new pile for this card. Otherwise, the value of **`whichpile`** is the appropriate pile and we update the corresponding entry in **`piles`** to indicate that the latest card is placed on top in that position.

The second task within the **`step`** loop is to update the **`pointers`** variable. This also depends on the value of **`whichpile`**.

- If **`whichpile`** is 1 or if **`piles`** only contains 1 entry, then the latest card was played on the first pile and the associated value should be **`Null`**.
- If **`whichpile`** is `Null`, then the value associated with the latest card is **`piles[[-2]]`**, the top card on what had been the last pile but is now the next to last pile. (Note that the previous condition, that **`whichpile`** is 1 or **`piles`** has only 1 entry will ensure that this second condition is only tested if **`piles`** has at least 2 entries and thus **`piles[[-2]]`** is a valid selection.)
- Otherwise, the value is **`piles[[ whichpile - 1 ]]`**.

That concludes the game stage. The second stage is the backtracking stage, which is much simpler.

First, access the top card of the last pile with **piles[[-1]]**, and initialize the maximal increasing list, **iList**, to the list consisting of this value.

Then we extend **iList** on the left with the entry in the **pointers** variable associated to that value. Since we're building the list from right to left, **iList[[1]]** always contains the most recently added number. So **pointers[iList[[1]]]** is the new value, which is added via PrependTo. Since the cards played in the first pile were associated with Null, we can use a While loop with condition **pointers[iList[[1]] =!= Null** to fill the **iList**. At the conclusion of the loop, the function returns **iList**.

Note throughout that whenever making a comparison which may be a comparison between an integer and the symbol Null, you must use SameQ (**===**) or UnsameQ (**=!=**), not Equal (**==**) and Unequal (**!=**).

Here, finally, is the function.

In[32]:=
```
findIncreasing[S : {__Integer}] :=
 Module[{piles = {}, pointers, step, whichpile, p, iList},
  (*game stage*)
  For[step = 1, step ≤ Length[S], step++,
   (*play the card*)
   whichpile = Catch[
     For[p = 1, p ≤ Length[piles], p++,
      If[S[[step]] < piles[[p]], Throw[p]]
     ];
     Throw[Null]
    ];
   If[whichpile === Null,
    AppendTo[piles, S[[step]]],
    piles[[whichpile]] = S[[step]]
   ];
   (*update pointers*)
   Which[whichpile === 1 || Length[piles] == 1,
    pointers[S[[step]]] = Null,
    whichpile === Null, pointers[S[[step]]] = piles[[-2]],
    True, pointers[S[[step]]] = piles[[whichpile - 1]]
   ]
  ]; (*ends main For loop*)
  (*backtracking stage*)
  iList = {piles[[-1]]};
  While[pointers[iList[[1]]] =!= Null,
   PrependTo[iList, pointers[iList[[1]]]]
  ];
  iList
 ]
```

Example 12 from the text involved the sequence 8, 11, 9, 1, 4, 6, 12, 10, 5, 7.

```
In[33]:= findIncreasing[{8, 11, 9, 1, 4, 6, 12, 10, 5, 7}]
Out[33]= {1, 4, 5, 7}
```

This is one of the four sequences given in the text.

Connection to the Pigeonhole Principle

Recall that Theorem 3 asserted that every sequence of $n^2 + 1$ distinct real numbers contains a subsequence of length $n + 1$ that is either strictly increasing or strictly decreasing. It may appear that the Patience algorithm has no connection to this theorem or to the pigeonhole principle.

However, the Patience algorithm does in fact suggest a proof of Theorem 3 via the pigeonhole principle. When the Patience algorithm is executed on a list of $n^2 + 1$ distinct real numbers, either there are at least $n + 1$ stacks or there are at most n stacks. If there are $n + 1$ stacks, then there is a strictly increasing subsequence of length $n + 1$.

On the other hand, assume that there are at most n stacks. Take the $n^2 + 1$ values to be the pigeons and the stacks the pigeonholes. When $n^2 + 1$ objects are placed in n boxes, then there is a box containing at least $\left\lceil \frac{n^2+1}{n} \right\rceil = \left\lceil \frac{n^2}{n} + \frac{1}{n} \right\rceil = n + \left\lceil \frac{1}{n} \right\rceil = n + 1$ objects (Theorem 2). Hence some stack has $n + 1$ values. But the rules of the game ensure that each stack is a strictly decreasing subsequence, since one value is placed on top of another in a stack only when the second value is lesser than, and appears in the sequence later than, the lower value.

In short, either there are $n + 1$ stacks and hence an increasing sequence of length $n + 1$ or there is a stack of size $n + 1$ and hence a decreasing sequence of length $n + 1$.

6.3 Permutations and Combinations

Mathematica includes many functions pertaining to counting and generating combinatorial structures.

Permutations

We begin by looking at commands related to permutations of objects.

We have seen in previous chapters the use of the exclamation mark for factorial.

```
In[34]:= 6!
Out[34]= 720
```

The `Factorial` (`!`) function can be used instead of the exclamation mark if you prefer. Otherwise they are equivalent.

To compute the number of r-permutations of n distinct objects, you can use the formula $P(n, r) = \frac{n!}{(n-r)!}$, given as Corollary 1 in Section 6.3 of the textbook. We can define a function in *Mathematica*, which we call **`numPerm`**, based on this formula. Recall that the formula requires that $0 \le r \le n$. We will include that as a `Condition` (`/;`) in the definition of the function.

```
In[35]:= numPerm[n_Integer, r_Integer] /; 0 ≤ r ≤ n := n! / (n - r)!
```

Then the number of 4-permutations of a set with 7 distinct objects, that is, $P(7, 4)$ is computed with the

following expression.

```
In[36]:= numPerm[7, 4]

Out[36]= 840
```

Listing Permutations

To obtain a list of all permutations, *Mathematica* provides the Permutations function.

The first argument is a list containing the objects to be permuted. If this is the only argument given, then the function returns the list of all permutations of those objects. For example, to list the permutations of {"a", "b", "c"}, you enter the following.

```
In[37]:= Permutations[{"a", "b", "c"}]

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

If you wish to list the permutations of the integers from 1 to a specified maximum, you can combine the Permutations function with Range applied to the maximum value. For example, the following produces all permutations of the first 4 positive integers.

```
In[38]:= Permutations[Range[4]]

Out[38]= {{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}, {1, 3, 4, 2},
          {1, 4, 2, 3}, {1, 4, 3, 2}, {2, 1, 3, 4}, {2, 1, 4, 3},
          {2, 3, 1, 4}, {2, 3, 4, 1}, {2, 4, 1, 3}, {2, 4, 3, 1},
          {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 2, 1, 4}, {3, 2, 4, 1},
          {3, 4, 1, 2}, {3, 4, 2, 1}, {4, 1, 2, 3}, {4, 1, 3, 2},
          {4, 2, 1, 3}, {4, 2, 3, 1}, {4, 3, 1, 2}, {4, 3, 2, 1}}
```

To produce the r-permutations of a list of objects, you only have to provide a second argument to Permutations. If you give an integer, n, as the second argument, the function will return the list of all permutations with at most n elements. That is, it produces all of the r-permutations for $r \leq n$. For example, the following produces all permutations of at most two elements of the set of the first five positive integers.

```
In[39]:= Permutations[Range[5], 2]

Out[39]= {{}, {1}, {2}, {3}, {4}, {5}, {1, 2}, {1, 3}, {1, 4}, {1, 5},
          {2, 1}, {2, 3}, {2, 4}, {2, 5}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
          {4, 1}, {4, 2}, {4, 3}, {4, 5}, {5, 1}, {5, 2}, {5, 3}, {5, 4}}
```

To obtain only the r-permutations for a specific r, enter the second argument as {r}. For example, to list all of the 3-permutations of {1, 2, 3, 4, 5}, you would enter the following.

```
In[40]:= Permutations[Range[5], {3}]

Out[40]= {{1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 2}, {1, 3, 4}, {1, 3, 5},
          {1, 4, 2}, {1, 4, 3}, {1, 4, 5}, {1, 5, 2}, {1, 5, 3}, {1, 5, 4},
          {2, 1, 3}, {2, 1, 4}, {2, 1, 5}, {2, 3, 1}, {2, 3, 4}, {2, 3, 5},
          {2, 4, 1}, {2, 4, 3}, {2, 4, 5}, {2, 5, 1}, {2, 5, 3}, {2, 5, 4},
          {3, 1, 2}, {3, 1, 4}, {3, 1, 5}, {3, 2, 1}, {3, 2, 4}, {3, 2, 5},
          {3, 4, 1}, {3, 4, 2}, {3, 4, 5}, {3, 5, 1}, {3, 5, 2}, {3, 5, 4},
          {4, 1, 2}, {4, 1, 3}, {4, 1, 5}, {4, 2, 1}, {4, 2, 3}, {4, 2, 5},
```

```
{4, 3, 1}, {4, 3, 2}, {4, 3, 5}, {4, 5, 1}, {4, 5, 2}, {4, 5, 3},
{5, 1, 2}, {5, 1, 3}, {5, 1, 4}, {5, 2, 1}, {5, 2, 3}, {5, 2, 4},
{5, 3, 1}, {5, 3, 2}, {5, 3, 4}, {5, 4, 1}, {5, 4, 2}, {5, 4, 3}}
```

You can also provide a range of values for *r* by entering the second argument as **{*min*, *max*}**.

Random Permutations

Mathematica also provides a function, RandomSample, that will produce a randomly chosen permutation.

Once again, the first argument is a list of the objects to be permuted. If no second argument is given, RandomSample will output a randomly chosen permutation of all of the elements. The following produces a random permutation of the letters "a" through "e".

```
In[41]:= RandomSample[{"a", "b", "c", "d", "e"}]
Out[41]= {b, e, d, c, a}
```

You can also provide a positive integer as the second argument. In this case, RandomSample will produce a random permutation of that size. Note that the second argument must be less than the size of the list. The following outputs a random 3-permutation of the first ten positive integers.

```
In[42]:= RandomSample[Range[10], 3]
Out[42]= {2, 3, 8}
```

Note that the permutation is selected so that each permutation has the same probability of being chosen.

Combinations

The functions related to combinations are very similar to those for permutations.

To compute the total number of combinations of a set of a specific size, use the formula 2^n, where *n* is the number of objects being selected from. For example, the following shows that there are 32 subsets of a set of 5 elements.

```
In[43]:= 2^5
Out[43]= 32
```

To compute the number of *r*-combinations of a set with *n* elements, *Mathematica* provides the Binomial function. The following computes *C*(52, 5).

```
In[44]:= Binomial[52, 5]
Out[44]= 2 598 960
```

Listing Combinations

The Subsets function is the combination analog of Permutations.

Given a list as the only argument, Subsets outputs all possible subsets of every size.

```
In[45]:= Subsets[{"a", "b", "c", "d"}]
Out[45]= {{}, {a}, {b}, {c}, {d}, {a, b}, {a, c}, {a, d}, {b, c}, {b, d},
          {c, d}, {a, b, c}, {a, b, d}, {a, c, d}, {b, c, d}, {a, b, c, d}}
```

To produce the subsets of the set {1, 2, …, *n*}, give the argument as **Range[n]**.

```
In[46]:= Subsets[Range[3]]

Out[46]= {{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}
```

Subsets accepts a second argument with the same syntax as Permutations. If an integer is given as a second argument, the output will be all subsets whose cardinality is at most that integer. For example, the following finds all of the *r*-combinations of {"a", "b", "c", "d"}, for $r \leq 2$.

```
In[47]:= Subsets[{"a", "b", "c", "d"}, 2]

Out[47]= {{}, {a}, {b}, {c}, {d}, {a, b},
          {a, c}, {a, d}, {b, c}, {b, d}, {c, d}}
```

If the second argument is provided as a list containing a single integer, then the output will be the subsets whose cardinality is equal to that integer. For example, you obtain the 3-combinations of {1, 2, 3, 4, 5} as shown below.

```
In[48]:= Subsets[Range[5], {3}]

Out[48]= {{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}}
```

With a list of two integers as the second argument, you obtain the subsets whose cardinalities are between the two values. The integers must be given with the smaller value first, or the output will be the empty list. The following calculates the proper, non-empty subsets of {1, 2, 3, 4, 5}.

```
In[49]:= Subsets[Range[5], {1, 4}]

Out[49]= {{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}, {1, 2, 3, 4},
          {1, 2, 3, 5}, {1, 2, 4, 5}, {1, 3, 4, 5}, {2, 3, 4, 5}}
```

With a third argument, you can limit the number of results that are included. A positive integer given as the third argument will cause Subsets to output at most that number of subsets. Note that the second argument must be present to be able to use this option. If you wish to use the third argument, but provide no restriction on the size of the subsets considered, you can give All as the second argument. The following outputs the first 10 subsets of {1, 2, 3, 4, 5}.

```
In[50]:= Subsets[Range[5], All, 10]

Out[50]= {{}, {1}, {2}, {3}, {4}, {5}, {1, 2}, {1, 3}, {1, 4}, {1, 5}}
```

You can obtain a single subset by entering **{*n*}** as the third argument. This will output the list containing the *n*th subset in the canonical order used by *Mathematica* when listing subsets. You can also give a pair **{*n*, *m*}** to list the subsets from the *n*th through the *m*th.

```
In[51]:= Subsets[Range[5], All, {7}]

Out[51]= {{1, 2}}
```

```
In[52]:= Subsets[Range[5], All, {3, 7}]

Out[52]= {{2}, {3}, {4}, {5}, {1, 2}}
```

Random Combinations

Mathematica does not explicitly provide a function for selecting random combinations. To randomly select an *r*-combination for a particular *r*, one approach is to apply Sort to a random permutation obtained with RandomSample. For example, the following produces a random 3-combination of {1, 2, …, 10}.

```
In[53]:= Sort[RandomSample[Range[10], 3]]

Out[53]= {1, 2, 4}
```

The application of Sort is not strictly necessary, but helps conceptually. The difference between an *r*-permutation and an *r*-combination is that order "matters" for the permutation and does not for the combination. For example, {1, 2, 3} and {2, 3, 1} are different as permutations but the same as combinations or sets. By applying Sort, we impose the same order on them, which makes the order they originally had irrelevant.

A second approach to is to randomly select from the output of Subsets. The function RandomChoice, applied to a list, will return one randomly chosen element of the list. Since Subsets outputs a list of subsets, it can be used as the argument to RandomChoice. The following demonstrates an alternate approach to choosing a random 3-combination of {1, 2, …, 10}. Recall that the second argument of {3} limits the output to the subsets of size 3.

```
In[54]:= RandomChoice[Subsets[Range[10], {3}]]

Out[54]= {1, 7, 9}
```

A third approach is to first select a random integer using the RandomInteger function, and then use that randomly selected integer in the third argument of Subsets. You must give the RandomInteger function the single argument of the form {1, *n*} where *n* is the number of combinations being selected from. This causes it to produce a random integer from 1 to that maximum. In the below, we first select a random integer between 1 and $C(10, 3)$, the number of 3-combinations of {1, 2, …, 10}. This value is stored as **whichone**. Then we apply Subsets with first argument **Range[10]**, which produces the set {1, 2, …, 10}; second argument **{3}**, indicating that only 3-combinations are allowed; and third argument **{whichone}**, which causes the function to return only the single subset at that position in the list of all the 3-combinations. Note that since Subsets always outputs a list of sets, the result will be nested.

```
In[55]:= whichone = RandomInteger[{1, Binomial[10, 3]}];
         Subsets[Range[10], {3}, {whichone}]

Out[56]= {{2, 5, 8}}
```

The first approach is the most efficient, but can only be applied when selecting a random *r*-combination for a specific value of *r*. If you wish to select a random subset whose cardinality can vary, the first method will select longer combinations more frequently than short ones. The second approach, using RandomChoice, is fairly straightforward and can be easily used to select a random combination without the restriction that they all be the same size. This approach can also be useful if you want to randomly select a combination from a list of combinations that meet some criteria. However, it is

inefficient and restricts the maximum size of the original set. The third option, using RandomInteger, is reasonably efficient, although it is not quite as fast as the first option. It is also fairly flexible, as you are not restricted to a single cardinality, provided that you accurately count the number of possible combinations when applying RandomInteger.

Circular Permutations

The prelude to Exercises 40 and 41 of Section 6.3 describes circular permutations. A circular *r*-permutation of *n* people is a seating of *r* of those *n* people at a circular table. Moreover, two seatings are considered the same if one can be obtained from the other by rotation.

The exercises ask you to compute the number of circular 3-permutations of 5 people and to arrive at a formula for that number. In this subsection, we'll write a function to list all of the circular *r*-permutations of *n* people. Having such a function can help you more easily explore the concept and test your formula.

Rotating a Permutation

The key to listing all circular permutations is to devise a way to test whether two circular permutations are equal. According to the definition, two circular permutations are considered to be equal if one can be obtained from the other by a rotation. If we use a list to represent a permutation, a rotation will consist of moving the first element to the end of the list (or the last to the front).

Mathematica includes the functions RotateLeft and RotateRight that manipulate lists in exactly this way. RotateLeft moves the first element to the end of the list and RotateRight moves the last element to the beginning, as illustrated below.

```
In[57]:= RotateLeft[{1, 2, 3, 4, 5}]
Out[57]= {2, 3, 4, 5, 1}
In[58]:= RotateRight[{1, 2, 3, 4, 5}]
Out[58]= {5, 1, 2, 3, 4}
```

Both of these functions can also accept an integer as a second optional argument to move more than one element at a time. For example, the following moves the first two elements of the list to the end.

```
In[59]:= RotateLeft[{1, 2, 3, 4, 5}, 2]
Out[59]= {3, 4, 5, 1, 2}
```

Equality of Circular Permutations

Note that repeatedly rotating a permutation will eventually result in the original. Specifically, for an *r*-permutation, after *r* rotations, the list will return to its original state. This is illustrated below with an example "seating."

```
In[60]:= RotateLeft[{"Abe", "Carol", "Barbara"}]
Out[60]= {Carol, Barbara, Abe}
In[61]:= RotateLeft[%]
Out[61]= {Barbara, Abe, Carol}
```

```
In[62]:= RotateLeft[%]
Out[62]= {Abe, Carol, Barbara}
```

This observation indicates that, given two r-permutations, we can test to see if they are the same by rotating one of them $r-1$ times. The function below returns true if the two input lists represent the same circular permutation and false otherwise. It first checks equality without performing rotation. Then, using a For loop, it rotates the second list, checking for equality after each rotation.

```
In[63]:= CPEqualQ[L1_List, L2_List] := Module[{i, Ltest = L2},
           If[L1 == Ltest, Return[True]];
           For[i = 1, i ≤ Length[Ltest] - 1, i++,
            Ltest = RotateLeft[Ltest];
            If[L1 == Ltest, Return[True]]
           ];
           Return[False]
          ]
```

We can use this function to confirm equality of circular permutations. For example,

```
In[64]:= CPEqualQ[{"Charles", "Helen", "Dean"},
          {"Helen", "Dean", "Charles"}]
Out[64]= True
```

Listing All Circular Permutations

Now we are prepared to write a function that lists all circular permutations. First, we'll use Permutations to generate all r-permutations of n people. We will initialize the list of all distinct circular permutations to the first element of the list of all permutations. That first permutation is then removed from the list of all permutations. Recall that Delete applied to a list and an index returns the list obtained by removing the element at the given location from the list.

Within a While loop conditioned on the list of permutations being nonempty, consider the first element in the list of all permutations. Use **CPEqualQ** and a loop to see if the first element is identical to any of the members of the set of circular permutations. If not, add it to the set of circular permutations. In either case, it is deleted from the list of all permutations. This continues until the list of all permutations has been emptied.

Here is the implementation. Our function will accept a list of "people" to be seated as the first argument and r, the number that can be seated at the table, as the second argument.

```
In[65]:= circularPermutations[S_List, r_Integer] :=
          Module[{allP, allCP, isnew, p},
           allP = Permutations[S, {r}];
           allCP = {allP[[1]]};
           allP = Delete[allP, 1];
           While[allP ≠ {},
            isnew = Catch[
              Do[If[CPEqualQ[allP[[1]], p], Throw[False]], {p, allCP}];
              Throw[True]
             ];
            If[isnew, AppendTo[allCP, allP[[1]]]];
            allP = Delete[allP, 1]
           ];
           allCP
          ]
```

Note that the **isnew** Boolean is used to track whether or not the current first member of **allP** is new or not.

The following computes the possible circular 3-permutations of the set {"Abe", "Barbara", "Carol", "Dean", "Eve"}.

```
In[66]:= circularPermutations[
          {"Abe", "Barbara", "Carol", "Dean", "Eve"}, 3]
```

```
Out[66]= {{Abe, Barbara, Carol}, {Abe, Barbara, Dean},
          {Abe, Barbara, Eve}, {Abe, Carol, Barbara},
          {Abe, Carol, Dean}, {Abe, Carol, Eve}, {Abe, Dean, Barbara},
          {Abe, Dean, Carol}, {Abe, Dean, Eve}, {Abe, Eve, Barbara},
          {Abe, Eve, Carol}, {Abe, Eve, Dean}, {Barbara, Carol, Dean},
          {Barbara, Carol, Eve}, {Barbara, Dean, Carol},
          {Barbara, Dean, Eve}, {Barbara, Eve, Carol},
          {Barbara, Eve, Dean}, {Carol, Dean, Eve}, {Carol, Eve, Dean}}
```

```
In[67]:= Length[circularPermutations[
           {"Abe", "Barbara", "Carol", "Dean", "Eve"}, 3]]
```

```
Out[67]= 20
```

It is left to the reader to experiment with other starting sets and values of *r* to determine a formula. Note that the functions in this subsection were written using a very naive approach. There are simpler and more efficient approaches, but those would give away the key idea used to create the formula.

6.4 Binomial Coefficients and Identities

In this section we will use *Mathematica* to compute binomial coefficients, to generate Pascal's triangle, and to verify identities.

The Binomial Theorem

Recall from the previous section that the *Mathematica* function `Binomial` can be used to compute $\binom{n}{k}$, which is another notation for $C(n, k)$. The `Binomial` function is in fact more general than the binomial coefficients described in the textbook, as it will compute coefficients that appear in Newton's generalized binomial theorem. The generalization is beyond the scope of this manual, but be aware that the function may return values even for inputs that you might expect should cause an error.

Here, we will consider questions such as Examples 2 through 4 from section 6.4 of the text.

First consider the problem of expanding $(x+y)^5$. In *Mathematica*, this can be done easily with `Expand`. The `Expand` function requires one argument, an algebraic expression. It returns the result of expanding the expression, that is, of distributing products over sums.

```
In[68]:= Expand[(x + y)^5]
Out[68]= x^5 + 5 x^4 y + 10 x^3 y^2 + 10 x^2 y^3 + 5 x y^4 + y^5
```

Now consider the question of finding the coefficient of $x^{18}\,y^{12}$ in the expansion of $(x+y)^{30}$. The binomial theorem tells us that this coefficient is $\binom{30}{12}$. The `Binomial` function with first argument 30 and second 12 will produce this value.

```
In[69]:= Binomial[30, 12]
Out[69]= 86 493 225
```

Thus, the expansion of $(x+y)^{30}$ contains the term $86\,493\,225\,x^{18}\,y^{12}$.

Finding the coefficient of $x^{12}\,y^{13}$ in the expansion of $(2\,x-3\,y)^{25}$ requires that we include the coefficients of x and y in the computation. As explained in the solution to Example 4 of the text, the expansion is

$$(2\,x+(-3\,y))^{25}=\sum_{j=0}^{25}\binom{25}{j}(2\,x)^{25-j}\,(-3\,y)^{j}$$

The coefficient of $x^{12}\,y^{13}$ is found by taking $j = 13$:

$$\binom{25}{13}2^{12}\,(-3)^{13}$$

This is

```
In[70]:= Binomial[25, 13] * 2^12 * (-3)^13
Out[70]= -33 959 763 545 702 400
```

Pascal's Triangle

As we've seen, it is very easy to compute binomial coefficients with *Mathematica*. To compute row n of Pascal's triangle, we apply the `Binomial` function with the second argument ranging from 0 to n. This can be done with the `Table` function. For example, the 25th row of Pascal's triangle is shown

below.

```
In[71]:= Table[Binomial[25, n], {n, 0, 25}]

Out[71]= {1, 25, 300, 2300, 12 650, 53 130, 177 100, 480 700,
  1 081 575, 2 042 975, 3 268 760, 4 457 400, 5 200 300,
  5 200 300, 4 457 400, 3 268 760, 2 042 975, 1 081 575,
  480 700, 177 100, 53 130, 12 650, 2300, 300, 25, 1}
```

When calculating a single binomial coefficient or an isolated row, applying the formula may be the most efficient approach. But if you wish to build a sizable portion of Pascal's triangle, making use of Pascal's identity (Theorem 2 of Section 6.4) and the symmetry property (Corollary 2 of Section 6.3) can be more effective.

In this subsection, we'll write two functions for computing binomial coefficients. The first will simply apply the formula $\frac{n!}{k!\,(n-k)!}$. The second will be a recursive function making use of Pascal's identity and symmetry. Then we'll compare the performance of the two functions in building Pascal's triangle.

The first function will be a straightforward application of the formula. We name it **binomialF** (for formula).

```
In[72]:= binomialF[n_Integer, k_Integer] /; 0 ≤ k ≤ n := n! / (k! * (n - k)!)
```

A Recursive Function

The second function will be called **binomialR** (for recursive). Recall Pascal's identity:

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$$

Rewriting this in terms of n and $n-1$, we have

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Also recall that the binomial coefficients are symmetric, that is,

$$\binom{n}{k} = \binom{n}{n-k}$$

With these facts in mind, our recursive function will work as follows. It will be designed to store values in an indexed variable with the same name as the function, **binomialR**. The body of the function will consist of a Which statement. In case the second argument is 0, the result is 1. That forms the basis case of the recursion. The second possibility is that $2k > n$. In this case, we make use of symmetry and call **binomialR** on n and $n-k$. Finally, if neither of those conditions are met, we apply Pascal's identity, making recursive calls to **binomialR**. Here is the implementation.

```
In[73]:= binomialR[n_Integer, k_Integer] /; 0 ≤ k ≤ n := Module[{},
           Which[k == 0, binomialR[n, k] = 1,
            2 * k > n, binomialR[n, k] = binomialR[n, n - k],
            True,
            binomialR[n, k] = binomialR[n - 1, k - 1] + binomialR[n - 1, k]
           ]
          ]
```

We use our two functions to build Pascal's triangle using a `Table` with two loop variables. We use the `Column` function with `Center` as second argument to give the table its usual triangular form.

```
In[74]:= Column[Table[binomialF[n, k], {n, 0, 5}, {k, 0, n}], Center]
```

```
Out[74]=
                {1}
              {1, 1}
            {1, 2, 1}
          {1, 3, 3, 1}
        {1, 4, 6, 4, 1}
      {1, 5, 10, 10, 5, 1}
```

```
In[75]:= Column[Table[binomialR[n, k], {n, 0, 5}, {k, 0, n}], Center]
```

```
Out[75]=
                {1}
              {1, 1}
            {1, 2, 1}
          {1, 3, 3, 1}
        {1, 4, 6, 4, 1}
      {1, 5, 10, 10, 5, 1}
```

Comparing Performance

Now we'll compare the performance of the two functions.

First, we'll use each of them to compute the first 1000 rows of Pascal's triangle and compare the time it takes. We use `Clear` and redefine **`binomialR`** in order to remove any values already stored so that the comparison is fair.

```
In[76]:= Timing[
          Table[binomialF[n, k], {n, 0, 1000}, {k, 0, n}];
         ]
```

```
Out[76]= {9.268064, Null}
```

In[77]:=
```
Clear[binomialR];
binomialR[n_Integer, k_Integer] /; 0 ≤ k ≤ n := Module[{},
   Which[k == 0, binomialR[n, k] = 1,
    2 * k > n, binomialR[n, k] = binomialR[n, n - k],
    True,
    binomialR[n, k] = binomialR[n - 1, k - 1] + binomialR[n - 1, k]
   ]
  ];
Timing[
 Table[binomialR[n, k], {n, 0, 1000}, {k, 0, n}];
]
```

Out[79]=
```
{4.557143, Null}
```

You see that the difference between the two performance of the two functions is substantial.

However, if we compute some isolated values, the situation is reversed. We will generate some random values of n between 100 and 200 and random values of k between 0 and the corresponding value of n.

The RandomInteger function applied to an integer outputs a random integer between 0 and that value, and given a pair $\{a, b\}$, it returns a randomly selected integer between a and b. With a positive integer as a second argument, the function will produce that many random integers in the selected range.

To produce 100 random values of n, we simply call RandomInteger on the range {100, 200} and the number 100 and assign the resulting list of n values to the symbol **randomNs**.

In[80]:=
```
randomNs = RandomInteger[{100, 200}, 100];
```

To produce the list of values of k, we use Table. Each value of k will be found by applying RandomInteger to the value found in **randomNs**. That way, the values of k will be between 0 and the corresponding value of n.

In[81]:=
```
randomKs = Table[RandomInteger[randomNs[[i]]], {i, 100}];
```

We can time the performance of **binomialF** simply by applying the function to the pairs of values.

In[82]:=
```
Timing[
 Table[binomialF[randomNs[[i]], randomKs[[i]]], {i, 100}];
]
```

Out[82]=
```
{0.001035, Null}
```

To check the performance of the recursive function, we must again Clear and redefine the function, as it currently stores the entire table up to $n = 1000$. We also must override *Mathematica*'s recursion limit, $RecursionLimit, to avoid errors based on the depth of the recursion we are attempting. This was not necessary when computing the table through row 1000 above, because each row was computed in turn and thus each individual computation had to look only one row down to find a known value. We set the recursion limit inside of a Block to localize the override.

In[83]:=
```
Clear[binomialR];
binomialR[n_Integer, k_Integer] /; 0 ≤ k ≤ n := Module[{},
   Which[k == 0, binomialR[n, k] = 1,
    2 * k > n, binomialR[n, k] = binomialR[n, n - k],
    True,
    binomialR[n, k] = binomialR[n - 1, k - 1] + binomialR[n - 1, k]
   ]
  ];
Block[{$RecursionLimit = Infinity},
 Timing[
  Table[binomialR[randomNs[[i]], randomKs[[i]]], {i, 100}];
 ]
]
```

Out[85]= `{0.075153, Null}`

The reason for the difference in relative performance is that when generating Pascal's triangle, all of the values beginning with $n = 0$, $k = 0$ needed to be calculated. On the other hand when calculating isolated values, the recursive function still had to calculate all of the results for lower values of n and k, which the non-recursive function did not need to compute.

Verifying Identities

Mathematica can help verify identities involving the binomial coefficients. If you enter an expression using the Binomial function and include symbolic arguments, *Mathematica* will try to simplify the expression algebraically. First, we ensure that **n** and **k** are unassigned.

In[86]:=
```
Clear[n, k]
```

For example, *Mathematica* can transform $C(n, 3)$ into an algebraic expression.

In[87]:=
```
Binomial[n, 3]
```

Out[87]=
$$\frac{1}{6}\,(-2+n)\,(-1+n)\,n$$

More general expressions, however, will be left unsimplified.

In[88]:=
```
Binomial[n, k]
```

Out[88]= `Binomial[n, k]`

However, we can force *Mathematica* to unwind this general expression into its formula with the FunctionExpand function. FunctionExpand is a way to insist that certain complex functions, such as Binomial, are expanded when doing so might allow further simplification. Observe what happens:

In[89]:=
```
FunctionExpand[Binomial[n, k]]
```

Out[89]=
$$\frac{\text{Gamma}[1+n]}{\text{Gamma}[1+k]\,\text{Gamma}[1-k+n]}$$

A complete description of the gamma function is beyond the scope of this manual. Suffice it to say that for positive integers, $\Gamma(n) = (n-1)!$. Because we are concerned only with the domain of positive

integers, we can transform the expression above into its more typical form by applying ReplaceAll (/.) and the rule Γ(n)=(n-1)!.

```
In[90]:= FunctionExpand[Binomial[n, k]] /. Gamma[n_] → (n - 1) !
```

$$\text{Out[90]}= \frac{n!}{k!\,(-k+n)!}$$

Verifying Symmetry

As a first example, we'll verify the identity $C(n, k) = C(n, n-k)$, the symmetry identity.

Assign names to the left and right hand sides of the identity.

```
In[91]:= left = Binomial[n, k]
Out[91]= Binomial[n, k]

In[92]:= right = Binomial[n, n - k]
Out[92]= Binomial[n, -k + n]
```

Note that if you simply try to apply Equal (==), *Mathematica* will simply echo the expressions, indicating that *Mathematica* is not able to confirm that the expressions are equal or not.

```
In[93]:= left == right
Out[93]= Binomial[n, k] == Binomial[n, -k + n]
```

Also, SameQ (===) will return False, since the expressions are not identical.

```
In[94]:= left === right
Out[94]= False
```

The above illustrate that a great deal of care is required when using *Mathematica* to verify identities. A negative result from SameQ (===) indicates that the expressions are not identical, while they still may be algebraically equivalent. And a failure of Equal (==) generally means that *Mathematica* needs more direction about how to proceed.

In particular, *Mathematica* needs to be told to expand the Binomial function. We saw one way to this above, using FunctionExpand. FullSimplify is another function that will tell *Mathematica* to delve into the definition of a function and attempt to perform algebraic simplification. Applying FullSimplify to the Equal (==) test yields the correct result.

```
In[95]:= FullSimplify[left == right]
Out[95]= True
```

A Second Identity

Exercise 26 in Section 6.4 asks you to prove the identity:

$$\sum_{k=1}^{n}\binom{n}{k}\binom{n}{k-1} = \binom{2n+2}{n+1}\Big/2 - \binom{2n}{n}$$

Mathematica will verify this identity for us.

First, we will give the right hand side of the identity a name.

```
In[96]:= right2 = Binomial[2 n + 2, n + 1] / 2 - Binomial[2 n, n]
```

$$\text{Out[96]= } -\text{Binomial}[2\,n,\,n] + \frac{1}{2}\,\text{Binomial}[2 + 2\,n,\,1 + n]$$

As you may have expected, *Mathematica* simply echoed the expression

The left hand side is a summation, so we apply the Sum function.

```
In[97]:= left2 = Sum[Binomial[n, k] * Binomial[n, k - 1], {k, 1, n}]
```

$$\text{Out[97]= } \frac{4^n\, n\, \left(\frac{1}{2}\,(-1 + 2\,n)\right)!}{\sqrt{\pi}\;(1 + n)!}$$

Mathematica automatically simplified the summation and returned a closed formula.

To verify the identity, apply FullSimplify to the expression identifying the two sides of the formula. Note that FullSimplify is required, using Equal (==) alone will not produce a truth value.

```
In[98]:= FullSimplify[left2 == right2]
Out[98]= True
```

Keep in mind when using *Mathematica* to check identities that it will only report true when the two sides of the expression are algebraically equivalent. If it does report true, you can be fairly confident that the identity does hold, though a truly convincing proof requires that you explicitly show the algebraic manipulations or provide a combinatorial argument justifying the equivalence.

If *Mathematica* reports false or just echoes the expression, however, even after using FullSimplify, you can not be certain whether the identity is false or if it is true but more manipulation is needed to get *Mathematica* to recognize it. To use *Mathematica* to demonstrate that a purported identity is false, you would need to find a counterexample by computing the values of both expressions and finding inputs that result in different values. (Refer to Section 1.7 for examples of finding counterexamples.)

6.5 Generalized Permutations and Combinations

In this section we will introduce a variety of *Mathematica* functions related to permutations and combinations with repetition allowed and related to distributing objects in boxes where the objects and the boxes may or may not be distinguishable.

Permutations with Repetition

Recall from Theorem 1 that the number of r-permutations of n objects is n^r if repetition is allowed.

For example, the number of strings of length 5 that can be formed from the 26 uppercase letters of the English alphabet is

```
In[99]:= 26^5
Out[99]= 11 881 376
```

As a second example, we compute the number of ways that four elements can be selected in order from

a set with three elements when repetition is allowed.

```
In[100]:= 3^4
Out[100]= 81
```

Recall from the previous section that the Permutations function accepts a list as its first argument. In case the list contains repeated elements, those elements are treated as identical, but are allowed to repeat in the permutations.

For example, consider the expression below.

```
In[101]:= Permutations[{1, 1, 2}]
Out[101]= {{1, 1, 2}, {1, 2, 1}, {2, 1, 1}}
```

Given a list of n not necessarily distinct objects and no second argument, the Permutations function produces all of the n-permutations of the n objects. Note that 1 appeared twice in the input and thus appears twice in the results, while 2 appeared once in the input and so appears once in the output.

With a second argument, you can specify the length of the permutations, just as was described in the previous section.

```
In[102]:= Permutations[{1, 1, 2}, {2}]
Out[102]= {{1, 1}, {1, 2}, {2, 1}}
```

Since 1 appeared twice in the input list, it was allowed to appear twice in the permutations, but 2 appeared only one time in the input and thus was not allowed to repeat.

This means that if you want to list the ways that four elements can be selected in order from a set with three elements when repetition is allowed, you can use the Permutations function, so long as you repeat the elements in the input. In order to generate all r-permutations of n objects with repetition allowed, you must use as input the list consisting of the n objects each repeated r times. If an object is repeated fewer than r times in the input list, then that will limit the number of times it is allowed to repeat in the results.

To form 4-permutations with repetition allowed of {"a", "b", "c"}, we apply the Permutations function after using ConstantArray and Join to build its argument.

The ConstantArray function takes two arguments. The first is the object to be repeated and the second is the number of times the object is to be repeated. The output is the list consisting of that number of copies of the element. For example, the following creates the list consisting of four copies of the character “a”.

```
In[103]:= ConstantArray["a", 4]
Out[103]= {a, a, a, a}
```

The Join function accepts a number of lists and outputs the list obtained by merging them. The following produces the list of four “a”s and four “b”s and four “c”s.

```
In[104]:= Join[ConstantArray["a", 4],
           ConstantArray["b", 4], ConstantArray["c", 4]]
Out[104]= {a, a, a, a, b, b, b, b, c, c, c, c}
```

You can make this more compact, especially for large numbers of symbols, by applying Table with a table variable ranging over the desired elements. Remember that if the second argument to Table is

of the form {*variable*, *list*}, then the table variable will be assigned each element of the list in turn. Using ConstantArray as the first argument to Table, with the table variable as the array element, we can obtain the list of "constant arrays."

```
In[105]:= Table[ConstantArray[i, 4], {i, {"a", "b", "c"}}]

Out[105]= {{a, a, a, a}, {b, b, b, b}, {c, c, c, c}}
```

Since Join expects several lists as arguments, not a single list of lists, we need to use the Apply (@@) operator. Apply (@@) has the effect of taking the elements of a list and giving them as the arguments to a function.

```
In[106]:= Join @@ Table[ConstantArray[i, 4], {i, {"a", "b", "c"}}]

Out[106]= {a, a, a, a, b, b, b, b, c, c, c, c}
```

This is the list we give to the Permutations function, along with the size specification {4}, to obtain permutations of length 4.

```
In[107]:= abcRepeated = Permutations[
            Join @@ Table[ConstantArray[i, 4], {i, {"a", "b", "c"}}],
            {4}
          ]

Out[107]= {{a, a, a, a}, {a, a, a, b}, {a, a, a, c}, {a, a, b, a}, {a, a, b, b},
           {a, a, b, c}, {a, a, c, a}, {a, a, c, b}, {a, a, c, c}, {a, b, a, a},
           {a, b, a, b}, {a, b, a, c}, {a, b, b, a}, {a, b, b, b}, {a, b, b, c},
           {a, b, c, a}, {a, b, c, b}, {a, b, c, c}, {a, c, a, a},
           {a, c, a, b}, {a, c, a, c}, {a, c, b, a}, {a, c, b, b},
           {a, c, b, c}, {a, c, c, a}, {a, c, c, b}, {a, c, c, c},
           {b, a, a, a}, {b, a, a, b}, {b, a, a, c}, {b, a, b, a},
           {b, a, b, b}, {b, a, b, c}, {b, a, c, a}, {b, a, c, b},
           {b, a, c, c}, {b, b, a, a}, {b, b, a, b}, {b, b, a, c},
           {b, b, b, a}, {b, b, b, b}, {b, b, b, c}, {b, b, c, a},
           {b, b, c, b}, {b, b, c, c}, {b, c, a, a}, {b, c, a, b},
           {b, c, a, c}, {b, c, b, a}, {b, c, b, b}, {b, c, b, c}, {b, c, c, a},
           {b, c, c, b}, {b, c, c, c}, {c, a, a, a}, {c, a, a, b},
           {c, a, a, c}, {c, a, b, a}, {c, a, b, b}, {c, a, b, c},
           {c, a, c, a}, {c, a, c, b}, {c, a, c, c}, {c, b, a, a},
           {c, b, a, b}, {c, b, a, c}, {c, b, b, a}, {c, b, b, b},
           {c, b, b, c}, {c, b, c, a}, {c, b, c, b}, {c, b, c, c}, {c, c, a, a},
           {c, c, a, b}, {c, c, a, c}, {c, c, b, a}, {c, c, b, b},
           {c, c, b, c}, {c, c, c, a}, {c, c, c, b}, {c, c, c, c}}
```

```
In[108]:= Length[abcRepeated]

Out[108]= 81
```

Note that the size of the list produced by Permutations agrees with the answer given by the formula n^r.

Combinations with Repetition

Combinations with repetition can be handled in *Mathematica* in much the same way as permutations with repetition are.

Theorem 2 of Section 6.5, asserts that the number of r-combinations of a set of n objects when repetition of elements is allowed is $C(n + r - 1, r)$. This suggests the following useful function.

```
In[109]:= CRep[n_Integer, r_Integer] /; n > 0 && r ≥ 0 := Binomial[n + r - 1, r]
```

Thus we can compute, for example, the number of ways to select five bills from a cash box with seven types of bills (Example 3) as follows.

```
In[110]:= CRep[7, 5]

Out[110]= 462
```

We can also make use of the Subsets function similar to how Permutations was used above.

Consider Example 2 from Section 6.5. In this example, we are given a bowl of apples, oranges, and pears and are to select four pieces of fruit from the bowl provided that it contains at least four pieces of each kind of fruit. We have two ways to solve this problem with *Mathematica*.

First, we can use the **CRep** function we created.

```
In[111]:= CRep[3, 4]

Out[111]= 15
```

The other approach is to use Subsets to list all the options. We form the argument to Subsets in the same way as in the last subsection. However, the difference between Subsets and Permutations is that Subsets treats repeated objects as distinct. This means that combinations will be repeated. For example, consider the following.

```
In[112]:= Subsets[{"a", "b", "b"}]

Out[112]= {{}, {a}, {b}, {b}, {a, b}, {a, b}, {b, b}, {a, b, b}}
```

The subsets {*b*} and {*a*, *b*} appear twice because *Mathematica* considered the two "b"s in the input to be different from each other. So one {*b*} in the output is from the first "b" in the input, and the other {*b*} is the second "b" in the input. To eliminate these redundancies, we can apply DeleteDuplicates, a function that removes duplicate elements from a list.

```
In[113]:= DeleteDuplicates[Subsets[{"a", "b", "b"}]]

Out[113]= {{}, {a}, {b}, {a, b}, {b, b}, {a, b, b}}
```

To answer the question about the bowl of fruit, we enter the following.

```
In[114]:= DeleteDuplicates[Subsets[Join @@ Table[
            ConstantArray[i, 4], {i, {"apple", "pear", "orange"}}],
           {4}
          ]
         ]

Out[114]= {{apple, apple, apple, apple}, {apple, apple, apple, pear},
           {apple, apple, apple, orange}, {apple, apple, pear, pear},
           {apple, apple, pear, orange}, {apple, apple, orange, orange},
```

```
{apple, pear, pear, pear}, {apple, pear, pear, orange},
{apple, pear, orange, orange}, {apple, orange, orange, orange},
{pear, pear, pear, pear}, {pear, pear, pear, orange},
{pear, pear, orange, orange}, {pear, orange, orange, orange},
{orange, orange, orange, orange}}
```

In[115]:= **Length[%]**

Out[115]= 15

Pay careful attention to the difference between Permutations and Subsets. Permutations considered repeated elements to be identical, while Subsets treats them as distinct.

Permutations with Indistinguishable Objects

Mathematica handles permutations with indistinguishable objects in the same way as when repetition is allowed. The Permutations function accepts a list of objects as its first argument. If objects in the list are repeated, *Mathematica* considers them as indistinguishable and produces the permutations appropriately.

For example, to solve Example 7, finding the number of different strings that can be made from the letters of the word SUCCESS, we use the list **{"S", "U", "C", "C", "E", "S", "S"}** as the argument to Permutations. Since we are only interested in the number of permutations, we apply Length as well.

In[116]:= **Length[Permutations[{"S", "U", "C", "C", "E", "S", "S"}]]**

Out[116]= 420

Observe that this gives the same result as the formula given in Theorem 3.

Mathematica makes it easy to go a bit further than Theorem 3, which is restricted to the situation when you are permuting all n objects. To find the number of r-permutations of n objects where some objects are indistinguishable, you give **{r}** as the second argument.

For example, the strings of length 3 that can be made from the letters of the word SUCCESS can be found as follows.

In[117]:= **Permutations[{"S", "U", "C", "C", "E", "S", "S"}, {3}]**

```
Out[117]= {{S, U, C}, {S, U, E}, {S, U, S}, {S, C, U}, {S, C, C}, {S, C, E},
           {S, C, S}, {S, E, U}, {S, E, C}, {S, E, S}, {S, S, U}, {S, S, C},
           {S, S, E}, {S, S, S}, {U, S, C}, {U, S, E}, {U, S, S}, {U, C, S},
           {U, C, C}, {U, C, E}, {U, E, S}, {U, E, C}, {C, S, U},
           {C, S, C}, {C, S, E}, {C, S, S}, {C, U, S}, {C, U, C},
           {C, U, E}, {C, C, S}, {C, C, U}, {C, C, E}, {C, E, S},
           {C, E, U}, {C, E, C}, {E, S, U}, {E, S, C}, {E, S, S},
           {E, U, S}, {E, U, C}, {E, C, S}, {E, C, U}, {E, C, C}}
```

Applying Length returns the number of such strings.

In[118]:= **Length[%]**

Out[118]= 43

A related question is the number of strings with 3 or more letters that can be made from the letters of the word SUCCESS. To find this, we only have to change the second argument of Permutations to {3,7}, indicating that the length of the permutations should be allowed to range from 3 to 7 (the number of letters in SUCCESS).

```
In[119]:= Length[Permutations[{"S", "U", "C", "C", "E", "S", "S"}, {3, 7}]]
Out[119]= 1247
```

Distinguishable Objects and Distinguishable Boxes

Example 8 asks how many ways there are to distribute hands of 5 cards to each of four players from a deck of 52 cards. There are several ways to compute this value in *Mathematica*.

First, we can use the expression in terms of combinations, $C(52, 5)\,C(47, 5)\,C(42, 5)\,C(37, 5)$, by using Binomial.

```
In[120]:= Binomial[52, 5] * Binomial[47, 5] *
           Binomial[42, 5] * Binomial[37, 5]
Out[120]= 1 478 262 843 475 644 020 034 240
```

Second, we can use the formula from Theorem 4: $\frac{52!}{5!\cdot 5!\cdot 5!\cdot 5!\cdot 32!}$.

```
In[121]:= 52! / (5! * 5! * 5! * 5! * 32!)
Out[121]= 1 478 262 843 475 644 020 034 240
```

Finally, this same value can be computed using the Multinomial function. Recall Theorem 4 in the text asserts that the number of ways to place n distinguishable objects into boxes so that n_1 objects are placed in box 1, n_2 objects are put in box 2, etc., is $\frac{n!}{n_1!\cdot n_2!\cdots n_k!}$. The Multinomial function takes $n_1, n_2, \ldots, n_k$ as arguments and applies the formula with n computed as $n_1 + n_2 + \cdots + n_k$. So to compute the answer to Example 8, you enter the following.

```
In[122]:= Multinomial[5, 5, 5, 5, 32]
Out[122]= 1 478 262 843 475 644 020 034 240
```

Revising the Multinomial Function

It is common in questions about distributing distinguishable objects into distinguishable boxes that you want to distribute only some of the objects. In Example 8, for instance, not all of the cards are dealt to players. The Multinomial function requires that you include the remainder of the cards as an argument. Conceptually, you can think of making one more box to hold the objects that are not placed in any of the other boxes.

This is such a common occurrence, however, that it seems more natural to forget about this "discard box" and instead include the total number of objects. We will write a *Mathematica* function that will use the formula from Theorem 4 but will require the total number of objects as the first argument and calculate the size of the discard box.

We would like our function to, like Multinomial, accept any number of arguments, rather than requiring the n_i to be collected in a list. That is, we would like to be able to compute the answer to Example 8 as follows.

```
myMultinomial[52, 5, 5, 5, 5]
```

The first argument will be the total number of objects, and it should be followed by at least one integer indicating the sizes of the boxes.

To allow the function to accept arbitrary numbers of argument, we use the BlankSequence (__) pattern, entered with two underscores. In the past, we've used this inside braces as a way to specify that an argument must be a list of a certain kind, as in **a:{__Integer}**. This will be similar, but without the braces.

The function definition will begin as shown below.

```
myMultinomial[n_Integer, L__Integer] :=
```

The difficulty arises when working with the sequence of integers matched by **L**. In particular, **L** does not represent a list, it represents a sequence of arguments. So functions that accept a list argument, such as Length, will raise an error, as illustrated below.

```
In[123]:= numberArgs[L__] := Length[L]

In[124]:= numberArgs[1, 2, 3, 4]

          Length::argx : Length called with 4 arguments; 1 argument is expected. >>

Out[124]= Length[1, 2, 3, 4]
```

The way to deal with this is to wrap **L** in braces before applying the function.

```
In[125]:= numberArgs2[L__] := Length[{L}]

In[126]:= numberArgs2[1, 2, 3, 4]

Out[126]= 4
```

On the other hand, functions that normally accepts arbitrary numbers of arguments and must usually be used in conjunction with Apply to operate on a list, can be given the sequence directly.

```
In[127]:= sumArgs[L__] := Plus[L]

In[128]:= sumArgs[1, 2, 3, 4]

Out[128]= 10
```

Now that we know how to work with the BlankSequence, writing the **myMultinomial** function is straightforward. We compute the size of the discard box by subtracting the first argument from the sum of the rest. The Factorial function automatically threads over lists, so we can apply it to the list formed from **L** and the discard box size. We need to use Apply (**@@**) with Times in order to multiply the factorials, and then divide $n!$ by that result.

```
In[129]:= myMultinomial[n_Integer, L__Integer] :=
           Module[{discard, denomList, denom},
            discard = n - Plus[L];
            denomList = Factorial[{L, discard}];
            denom = Times @@ denomList;
            n! / denom
           ]

In[130]:= myMultinomial[52, 5, 5, 5, 5]

Out[130]= 1 478 262 843 475 644 020 034 240
```

The reader is encouraged to eliminate the local variables and write a "one line" version of this function.

Indistinguishable Objects and Distinguishable Boxes

The text describes the correspondence between questions about placing indistinguishable objects into distinguishable boxes and about combination with repetition questions.

Example 9 asks how many ways 10 indistinguishable balls can be placed in 8 bins. We can use the **CRep** function written earlier.

```
In[131]:= CRep[8, 10]

Out[131]= 19 448
```

It may seem that the arguments were reversed. Keep in mind that the connection to combinations with repetition is that you are selecting 10 bins from the 8 available bins, with repetition allowed.

Compositions and Weak Compositions

We can also use the Compositions and NumberOfCompositions functions to answer questions of this kind. A k-composition of a positive integer n is a way of writing n as the sum of k positive integers where the order of the summands matters. For example, 4 has three distinct 2-compositions: $3+1$, $2+2$, and $1+3$.

A weak composition is similar, but the terms in the sum are allowed to be 0. Thus 4 has five distinct weak 2-compositions: $4+0$ and $0+4$ in addition to the three listed before.

Note that the weak r-compositions of n correspond to the r-compositions of $n+r$. For suppose that $x_1 + x_2 + \cdots + x_r = n$ is a weak r-composition. Then each x_i is nonnegative. So

$$(x_1 + 1) + (x_2 + 1) + \cdots + (x_r + 1) = n + r,$$

and each $x_i + 1$ is positive, and hence this is a composition of $n+r$. Likewise, any r-composition of $n+r$ can be transformed into a weak r-composition by subtracting 1 from each term.

Also note that weak r-compositions of n correspond to placing n indistinguishable balls into r distinguishable bins. Suppose $x_1 + x_2 + \cdots + x_r = n$ is a weak r-composition of n. This can be identified with placing x_1 of the objects into the first bin, x_2 objects in the second bin, etc.

We now return to Compositions and NumberOfCompositions. These are functions contained in the *Combinatorica* package, and are not available without loading that package by executing a Needs statement.

In[132]:=
```
Needs["Combinatorica`"]
```

General::compat :
Combinatorica Graph and Permutations functionality has been superseded by preloaded functionality. The package now being loaded may conflict with this. Please see the Compatibility Guide for details.

You may safely ignore the compatibility warning in this context. Some of the functionality of the *Combinatorica* package has been superseded by new functionality in the main kernel in *Mathematica*. However, this does not affect the functions we will be using.

The `Compositions` and `NumberOfCompositions` functions accept the same arguments: the integers n and r. `NumberOfCompositions` outputs the number of *weak* r-compositions of n, while `Compositions` returns a list of them. Observe that the names *Combinatorica* uses for these functions is slightly at odds with the usual terminology.

In[133]:=
```
NumberOfCompositions[4, 2]
```

Out[133]=
```
5
```

In[134]:=
```
Compositions[4, 2]
```

Out[134]=
```
{{0, 4}, {1, 3}, {2, 2}, {3, 1}, {4, 0}}
```

The number of ways to place n indistinguishable objects in r distinguishable boxes is the same as the number of weak r-compositions of n. Consequently, we can determine the number of ways 10 balls can be placed in 8 bins, the same question as before, as follows.

In[135]:=
```
NumberOfCompositions[10, 8]
```

Out[135]=
```
19 448
```

Distinguishable Objects and Indistinguishable Boxes

As described in the text, the number of ways to place n distinguishable objects in k indistinguishable boxes is given by the Stirling numbers of the second kind, $S(n, k)$.

The function `StirlingS2` computes the Stirling number of the second kind. This function requires two arguments, the number of objects and the number of boxes. For example, the expression below computes the number of ways to put seven different employees in 4 different offices when each office must not be empty.

In[136]:=
```
StirlingS2[7, 4]
```

Out[136]=
```
350
```

In order to compute the number of ways to assign the 7 employees to the 4 offices and allow empty offices, we must add the number of ways to assign all 7 employees to 1 office, to 2 offices, to 3 offices, and to 4 offices.

In[137]:=
```
Sum[StirlingS2[7, i], {i, 1, 4}]
```

Out[137]=
```
715
```

Generating Assignments of Employees to Offices

The StirlingS2 function tells us how many ways there are to place distinguishable objects in indistinguishable boxes. In this subsection, we will create a function to list these. However, note that the *Combinatorica* package has a function, similar to the one we will create by hand, called SetPartitions.

To create a function that will list the possible assignments of distinguishable objects to boxes, we rely on the following observations. First, as indicated in the text, a choice of distinguishable objects to indistinguishable boxes can be modeled as a set of subsets. For instance, $\{\{A, C\}, \{D\}, \{B, E\}\}$ represents the assignment of A and C to one box, D to a box of its own, and B and E to another box. The set of subsets must not contain the empty set and must be such that the union of the subsets be the entire collection of objects.

We can produce such assignments recursively. The basis step is that there is only one way to assign n objects to 1 box and there are no ways to assign n objects to k boxes for $k > n$ (under the requirement that no box be empty). To assign n objects to k boxes with $k \leq n$, proceed as follows.

First, find all assignments of $n-1$ objects to $k-1$ boxes and update each assignment by placing object n in a box by itself. In terms of the set representation, given $\{B_1, B_2, \ldots, B_{k-1}\}$, we produce $\{B_1, B_2, \ldots, B_{k-1}, \{n\}\}$.

Second, find all assignments of $n-1$ objects to k boxes. For each such assignment $\{B_1, B_2, \ldots, B_k\}$, produce the following k assignments of n objects to the k boxes:

$$\{B_1 \bigcup \{n\}, B_2, \ldots, B_k\}, \{B_1, B_2 \bigcup \{n\}, B_3, \ldots, B_k\}, \ldots \{B_1, B_2, \ldots, B_k \bigcup \{n\}\}$$

The assignments of objects to boxes produced by the two methods above produce all assignments. The following function implements this algorithm.

```
In[138]:= makeStirling2[n_Integer, k_Integer] /; n > 0 && k > 0 :=
 Module[{A, k1boxes, kboxes, B, new, i},
  Which[k == 1,
   A = {{Range[n]}},
   k > n,
   A = {},
   True,
   (* the recursion: *)
   A = {};
   (* n-1 objects in k-1 boxes *)
   k1boxes = makeStirling2[n - 1, k - 1];
   Do[new = Union[B, {{n}}]; AppendTo[A, new]
    , {B, k1boxes}];
   (* n-1 objects in k boxes *)
   kboxes = makeStirling2[n - 1, k];
   Do[For[i = 1, i ≤ k, i++,
     new = ReplacePart[B, i → Append[B[[i]], n]];
     AppendTo[A, new]
    ], {B, kboxes}]
  ];
  A
 ]
```

Let us analyze the function. It accepts n and k as parameters and returns the list of all possible assignments of distinguishable objects to indistinguishable boxes. The symbol **A** stores the list that will ultimately be output.

In the case that $k = 1$, there is only one possible assignment, all objects are assigned to the single box. This assignment is represented by $\{\{1, 2, \ldots, n\}\}$, since an assignment corresponds to a set of subsets. The function sets the output symbol **A** to the list consisting of this single assignment when $k = 1$. Recall the Range function applied to a positive integer produces the list of integers from 1 to that value.

If $k > n$, then there are no valid assignments and the function sets the output symbol **A** to the empty set.

Otherwise, the symbol **A** is initialized to the empty set. Recall that there are two recursive steps. First expanding on the assignments of $n - 1$ objects to $k - 1$ boxes. And then expanding on the assignments of $n - 1$ objects to k boxes.

For the first part, we assign the symbol **k1boxes** to the set of assignments of $n - 1$ objects to $k - 1$ boxes. For each such assignment, that is each **B** in **k1boxes**, we add n in its own box, e.g., for $\{\{1, 3, 5\}, \{2\}, \{4, 6\}\}$, we would add 7:

$$\{\{1, 3, 5\}, \{2\}, \{4, 6\}\} \bigcup \{\{7\}\} = \{\{1, 3, 5\}, \{2\}, \{4, 6\}, \{7\}\}.$$

This **new** assignment is then added to **A**.

In the second part, we assign **kboxes** to the set of assignments of $n - 1$ objects to k boxes. For each such assignment **B**, we consider each of the k boxes in turn and add n to that box. For instance, the

assignment {{2, 3}, {1}, {5}, {4, 6}} would generate the four assignments:

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

To create these four **new** assignments from the initial assignment **B**, we use the ReplacePart function. In its simplest form, as we use it here, ReplacePart takes two arguments. The first argument is an expression to be manipulated, in this case **B**, the original list representing an assignment of objects to boxes. The second argument is a rule of the form ***index*** **->** ***replacement***. The *index* is the location within **B** that is to be substituted with the *replacement* expression. In **makeStirling2**, we use a For loop variable as the index and the replacement expression is obtained from Append to add the value *n* to the box at that index.

Compare the result of our function to the solution of Example 10 in Section 6.5 of the text. We use Column to put each assignment on its own line.

In[139]:=
```
makeStirling2[4, 3] // Column
```

Out[139]=
```
{{3}, {4}, {1, 2}}
{{2}, {4}, {1, 3}}
{{1}, {4}, {2, 3}}
{{1, 4}, {2}, {3}}
{{1}, {2, 4}, {3}}
{{1}, {2}, {3, 4}}
```

Except for using the integers 1 through 4 instead of the letters A through D, the output above is the same as the six ways listed in the text for placing the four employees in 3 offices.

To produce all 14 ways to assign the 4 employees to 3 offices with each office containing any number of employees, we need to loop over the different values of *k*. Using Table to vary the number of offices results in a list, each element of which is a list of results. Using Flatten with second argument 1 combines this list of lists into a single list of results. The 1 as the second argument indicates that only the top level of lists are to be flattened. The result can be displayed with Column.

In[140]:=
```
Flatten[Table[makeStirling2[4, k], {k, 1, 3}], 1] // Column
```

Out[140]=
```
{{1, 2, 3, 4}}
{{4}, {1, 2, 3}}
{{3, 4}, {1, 2}}
{{3}, {1, 2, 4}}
{{1, 3, 4}, {2}}
{{1, 3}, {2, 4}}
{{1, 4}, {2, 3}}
{{1}, {2, 3, 4}}
{{3}, {4}, {1, 2}}
{{2}, {4}, {1, 3}}
{{1}, {4}, {2, 3}}
```

```
{{1, 4}, {2}, {3}}
{{1}, {2, 4}, {3}}
{{1}, {2}, {3, 4}}
```

Indistinguishable Objects and Indistinguishable Boxes

As described in the text, distributing n indistinguishable objects into k indistinguishable boxes is identical to forming a partition of n into k positive integers. A partition of n into k positive integers is a sum $n = a_1 + a_2 + \cdots + a_k$ with $a_1 \geq a_2 \geq \cdots \geq a_k > 0$.

The *Mathematica* functions IntegerPartitions and PartitionsP are used to form and count partitions of integers. With one argument, a nonnegative integer n, PartitionsP returns the total number of partitions of n into as many as n boxes. Likewise, IntegerPartitions applied to one argument returns a list containing lists representing partitions of its argument.

For example, the expressions below compute the number of partitions of 7 and lists all the partitions of 7.

```
In[141]:= PartitionsP[7]
Out[141]= 15
In[142]:= IntegerPartitions[7]
Out[142]= {{7}, {6, 1}, {5, 2}, {5, 1, 1}, {4, 3}, {4, 2, 1}, {4, 1, 1, 1},
           {3, 3, 1}, {3, 2, 2}, {3, 2, 1, 1}, {3, 1, 1, 1, 1}, {2, 2, 2, 1},
           {2, 2, 1, 1, 1}, {2, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1}}
```

The IntegerPartitions function also accepts a second argument, which specifies the number of elements allowed to appear in a partition. For example, to answer the question: how many can you distribute 7 indistinguishable balls in up to 3 identical boxes, we would give 3 as the second argument to IntegerPartitions.

```
In[143]:= IntegerPartitions[7, 3]
Out[143]= {{7}, {6, 1}, {5, 2}, {5, 1, 1},
           {4, 3}, {4, 2, 1}, {3, 3, 1}, {3, 2, 2}}
```

This second argument, which specifies the length of the partitions, can take on many of the usual forms: n, as we saw, indicates the maximum length; {n} limits the output to those partitions of length exactly n, and {n, m} produces the partitions from length n to m. Also, the symbol All can be used to list all of the partitions, just as if no second argument were given.

For example, the following produces the partitions of 7 whose lengths are between 3 and 5.

```
In[144]:= IntegerPartitions[7, {3, 5}]
Out[144]= {{5, 1, 1}, {4, 2, 1}, {4, 1, 1, 1}, {3, 3, 1}, {3, 2, 2},
           {3, 2, 1, 1}, {3, 1, 1, 1, 1}, {2, 2, 2, 1}, {2, 2, 1, 1, 1}}
```

IntegerPartitions also accepts a third optional argument to control what values may appear in the partitions. Note that in order to use the this option, the second argument, specifying the length, must be given. To allow unrestricted lengths, you should give All as the second argument. The third argument has only one form: a list of the allowable values.

For example, to partition 15 using only 2, 3, and 4, you would enter the following.

```
In[145]:= IntegerPartitions[15, All, {2, 3, 4}]

Out[145]= {{4, 4, 4, 3}, {4, 4, 3, 2, 2}, {4, 3, 3, 3, 2}, {4, 3, 2, 2, 2, 2},
           {3, 3, 3, 3, 3}, {3, 3, 3, 2, 2, 2}, {3, 2, 2, 2, 2, 2, 2}}
```

By using Range in the third argument, you can easily specify a maximum allowable value, or another range of values. For example, to answer the question: how many ways are there to distribute 7 indistinguishable balls in up to 5 identical boxes when each box can hold at most 4 objects, you would given **Range[4]** as the third argument.

```
In[146]:= IntegerPartitions[7, 5, Range[4]]

Out[146]= {{4, 3}, {4, 2, 1}, {4, 1, 1, 1}, {3, 3, 1}, {3, 2, 2},
           {3, 2, 1, 1}, {3, 1, 1, 1, 1}, {2, 2, 2, 1}, {2, 2, 1, 1, 1}}
```

To determine the number, without outputting the list, simply apply Length.

```
In[147]:= Length[IntegerPartitions[7, 5, Range[4]]]

Out[147]= 9
```

Note that these optional arguments are not available for PartitionsP.

Equivalence of Maximum Length and Maximum Value

The partitions of n with at most k objects in a box are in one-to-one correspondence with the partitions of n into at most k boxes. To understand why, consider the partition {3, 2, 2, 1}.

Think about placing the 8 objects in boxes according to the partition {3, 2, 2, 1}. (The diagram shown below is a Ferrers diagram. A version of this diagram can be created with the function FerrersDiagram in the *Combinatorica* package.)

X	X	X	X
X	X	X	
X			

Instead of thinking about the columns as the boxes, we can instead consider the rows as boxes.

X	X	X	X
X	X	X	
X			

Now the 8 objects are contained in three boxes. One box (the top row) has 4 objects, another (the middle row) has 3 objects, and the last box (the bottom row) has one object. In other words, we've partitioned 8 as {4, 3, 1}. This partition is said to be the transpose of the first. (Note that you can also think about forming the transpose by reflecting it across its diagonal.)

The *Combinatorica* function TransposePartition will compute the transpose of a given partition.

```
In[148]:= TransposePartition[{3, 2, 2, 1}]

Out[148]= {4, 3, 1}
```

We began with a partition whose maximum entry was 3 and found that its transpose was a partition into 3 boxes. It is always the case that the transpose of a partition with maximum n is a partition into n

boxes. Moreover, this correspondence is one-to-one. We leave it to the reader to prove these facts.

Generating Partitions

Here we will describe how to generate partitions and use this description to generalize the *Mathematica* function PartitionsP with a second argument limiting the maximum number in the partition.

We will describe how to recursively form the partitions of n objects when each box can hold at most k objects. The basis cases are: when $k = 1$, there is only one partition of n, the partition consisting of n 1s; when $n = 0$, there is only one partition, the empty partition {}.

To determine the partitions of n when each box can hold at most k objects, with $n > 0$ and $k > 1$, proceed as follows. First, determine all partitions of n when each box can hold at most $k - 1$ objects. These partitions are also partitions of n satisfying the requirement that each box holds at most k objects.

Second, provided that $n - k \geq 0$, determine all partitions of $n - k$ when each box can hold at most k objects, and prepend k to each partition. For example, with $n = 7$ and $k = 3$, we have $n - k = 4$ and the partitions of 4 with each box holding at most 3 are:

```
In[149]:= IntegerPartitions[4, All, Range[3]]
Out[149]= {{3, 1}, {2, 2}, {2, 1, 1}, {1, 1, 1, 1}}
```

By prepending 3 to each of these partitions, we obtain {3, 3, 1}, {3, 2, 2}, {3, 2, 1, 1}, {3, 1, 1, 1, 1}, which are each partitions of 7 with each box having at most 3 objects.

Combining the partitions of n with each box holding at most $k - 1$ objects with the partitions of n formed by appending k to the partitions of $n - k$ with each box holding at most k objects produces all partitions of n with each box holding at most k objects. Setting $k = n$ produces all partitions of n.

It is an exercise to implement this algorithm to generate partitions.

We can use the above to generalize PartitionsP. We will define a function **myPartitionsP**. This function will be defined recursively. The basis cases are when $k = 1$ or $n = 0$, there is one partition.

```
In[150]:= myPartitionsP[_, 1] := 1;
          myPartitionsP[0, _] := 1
```

For $n < 0$ or $k < 1$, there are no partitions.

```
In[152]:= myPartitionsP[n_, k_] /; n < 0 || k < 1 := 0
```

Provided that $n > 0$ and $k > 1$, the discussion above leads us to the following recursive definition.

```
In[153]:= myPartitionsP[n_Integer, k_Integer] /; n > 0 && k > 1 :=
           myPartitionsP[n, k - 1] + myPartitionsP[n - k, k]
```

We see that the value of this function coincides with the number of partitions produced by IntegerPartitions.

```
In[154]:= Length[IntegerPartitions[20, 5]]
Out[154]= 192
In[155]:= myPartitionsP[20, 5]
Out[155]= 192
```

We can easily extend our function to compute all partitions of an integer n, as follows.

```
In[156]:= myPartitionsP[n_] := myPartitionsP[n, n]
```

Note that the results agree with the built-in function.

```
In[157]:= myPartitionsP[15]
Out[157]= 176
In[158]:= PartitionsP[15]
Out[158]= 176
```

6.6 Generating Permutations and Combinations

In this section, we will implement Algorithm 1 from Section 6.6 of the text for generating the next permutation in lexicographic order. Implementing Algorithms 2 and 3 will be left as exercises for the reader.

interchange

Before implementing Algorithm 1, we will first write a function to interchange two elements in a list. This will be called by the function for generating permutations.

The **interchange** function will require three parameters: the list and two integers representing the indices to be swapped.

The function operates as follows. A copy of the list is made. The element in the list at the first position to be swapped is assigned to a local variable. Then the first position is assigned to the value in the second position. Finally, the second position is assigned to the value stored in the temporary name and the list is returned.

Here is the implementation and an example of applying it.

```
In[159]:= interchange[L_List, i_Integer, j_Integer] /;
            0 < i ≤ Length[L] && 0 < j ≤ Length[L] := Module[{l = L, temp},
            temp = l[[i]];
            l[[i]] = l[[j]];
            l[[j]] = temp;
            l
           ]
In[160]:= interchange[{"a", "b", "c", "d", "e", "f"}, 2, 5]
Out[160]= {a, e, c, d, b, f}
```

nextPermutation

The input to the function will be a permutation $\{a_1, a_2, \ldots, a_n\}$ of the set $\{1, 2, \ldots, n\}$. Algorithm 1 consists of three steps: finding the largest j such that $a_j < a_{j+1}$; finding the smallest a_k to the right of a_j and interchanging a_k and a_j; and putting the elements in positions $j+1$ and beyond in increasing order.

The first step comprises the first four lines of the body of Algorithm 1 in the text, ending with the first comment. The index j is initialized to the next to last index in the permutation. A `While` loop is used

to conduct the search. The body of the loop decreases the value of j by one, and it is controlled by the condition $a_j > a_{j+1}$. When the `While` loop terminates, it will be the case that $a_j < a_{j+1}$ and j is the largest index for which that is true. Consequently, $a_{j+1} > a_{j+2} > \cdots > a_n$.

The second step is to find the smallest a_k to the right of and larger than a_j and interchange the two. Since we are guaranteed that the elements to the right of a_j are in increasing order, a_n is the smallest element to the right of a_j, a_{n-1} is the next smallest, and so on. We are again searching from the right. Initialize k to n. A `While` loop is used to decrease k by one so long as $a_j > a_k$. When the `While` loop terminates, k will be such that $a_j < a_k$. Note that the loop is guaranteed to stop with $k > j$ since $a_j < a_{j+1}$. Once j has been identified, we interchange a_j and a_k using the **`interchange`** function.

The third step is to put the elements of the permutation to the right of position j in increasing order. Note that before the interchange, a_{j+1} through a_n were in decreasing order. After the interchange of a_j with a_k, the tail end of the permutation remains in decreasing order. This is because a_j was smaller than a_k, but a_k was the smallest of the entries bigger than a_j. Thus all of $a_{k+1}, \ldots, a_n$ are smaller than a_j, and all of $a_{j+1}, \ldots, a_{k-1}$ are larger than a_k which is larger than a_j. Therefore,

$$a_{j+1}, \ldots, a_{k-1}, a_j, a_{k+1}, \ldots, a_n$$

is in decreasing order.

To put the tail in increasing order, we follow the instructions in the pseudocode in the textbook that follows the interchange of a_j and a_k. Variables r and s are initialized to n and $j+1$, respectively. Provided that r remains larger than s, we interchange a_r and a_s and then decrease r by 1 and increase s by 1. This has the effect of swapping a_{j+1} with a_n, then a_{j+2} with a_{n-1}, then a_{j+3} with a_{n-2}, etc.

Once the tail is in increasing order, the result is the new permutation and it is returned.

We need to add to the function two tests to ensure that the input is valid. First, as a `Condition` (**`/;`**) to the function definition, we ensure that the input is a permutation of $\{1, 2, \ldots, n\}$. We do this by sorting the input, using the `Sort` function, and comparing it with the result of applying `Range` to the `Length` of the input list. Second, as the first expression in the function, we check to see if the input is the list $\{n, n-1, \ldots, 2, 1\}$, which is the final permutation in the canonical order. The `Reverse` function, applied to a list, simply reverses the order of the list. This is used in conjunction with `Range` to produce the list $\{n, n-1, \ldots, 2, 1\}$, which is compared to the input. If they agree, `Return` is used to terminate the function and cause the output to be `Null`.

Here is the implementation.

```
In[161]:= nextPermutation[A_List] /; Sort[A] == Range[Length[A]] :=
 Module[{a = A, n = Length[A], i, j, k, r, s},
  If[a == Reverse[Range[n]], Return[Null]];
  (* step 1: find j *)
  j = n - 1;
  While[a[[j]] > a[[j + 1]],
   j = j - 1
  ];
  (* step 2: find k and interchange aj and ak *)
  k = n;
  While[a[[j]] > a[[k]],
   k = k - 1
  ];
  a = interchange[a, j, k];
  (* step 3: sort the tail *)
  r = n;
  s = j + 1;
  While[r > s,
   a = interchange[a, r, s];
   r = r - 1;
   s = s + 1
  ];
  a
 ]
```

Example 2 of Section 6.6 finds that the permutation after 362541 is 364125. Let's use that example to confirm that our function is working.

```
In[162]:= nextPermutation[{3, 6, 2, 5, 4, 1}]
Out[162]= {3, 6, 4, 1, 2, 5}
```

To generate all permutations of a set {1, 2, …, *n*} we use a `While` loop.

```
In[163]:= aperm = Range[4];
          While[aperm =!= Null,
           Print[aperm];
           aperm = nextPermutation[aperm]
          ]
          {1, 2, 3, 4}
          {1, 2, 4, 3}
          {1, 3, 2, 4}
          {1, 3, 4, 2}
          {1, 4, 2, 3}
```

```
{1, 4, 3, 2}
{2, 1, 3, 4}
{2, 1, 4, 3}
{2, 3, 1, 4}
{2, 3, 4, 1}
{2, 4, 1, 3}
{2, 4, 3, 1}
{3, 1, 2, 4}
{3, 1, 4, 2}
{3, 2, 1, 4}
{3, 2, 4, 1}
{3, 4, 1, 2}
{3, 4, 2, 1}
{4, 1, 2, 3}
{4, 1, 3, 2}
{4, 2, 1, 3}
{4, 2, 3, 1}
{4, 3, 1, 2}
{4, 3, 2, 1}
```

Finding Permutations of Other Sets

As mentioned in the text, any set with n elements can be put in one-to-one correspondence with $\{1, 2, \ldots, n\}$. Consequently, any permutation of a set with n elements can be obtained from a permutation of $\{1, 2, \ldots, n\}$ and the correspondence.

In *Mathematica*, applying a permutation, represented as an arrangement of $\{1, 2, \ldots, n\}$, to a list is particularly easy. The `Part` (`[[…]]`) operator can be applied to a list of integers and will return the list ordered by the argument. For example, consider the set {"a", "b", "c"} and the permutation {3, 1, 2}. To arrange {"a", "b", "c"} by the permutation, you simply apply the `Part` (`[[…]]`) operator to {"a", "b", "c"} with argument {3, 1, 2}.

```
In[165]:= {"a", "b", "c"}[[{3, 1, 2}]]
Out[165]= {c, a, b}
```

As another example, consider the set {2, 10, 13, 19} and the permutation {4, 2, 3, 1}. Then the following expression applies the permutation to the set.

```
In[166]:= {2, 10, 13, 19}[[{4, 2, 3, 1}]]
Out[166]= {19, 10, 13, 2}
```

Note that the braces inside the Part ([[…]]) operator are required. Without them, *Mathematica* will interpret it as representing nested levels rather than a rearrangement.

A Function to Permute a General List

We can use the **nextPermutation** function and the above information about the Part ([[…]]) operator to write a function that outputs all permutations of any list.

The input to the function will be the list to be permuted. The bulk of the function will be contained in a Reap, with Sow applied to each permutation. Otherwise, the function will mirror the While loop above that was used to list the permutations of {1, 2, 3, 4}. The main difference is that, rather than just printing those permutations, they are used as the argument to the Part ([[…]]) operator to permute the given list.

```
In[167]:= permuteList[L_List] := Module[{perm},
            perm = Range[Length[L]];
            Reap[
              While[perm =!= Null,
               Sow[L[[perm]]];
               perm = nextPermutation[perm]
              ]
             ][[2, 1]]
           ]
```

Below, we list all the permutations of the set {"a", "b", "c"}.

```
In[168]:= permuteList[{"a", "b", "c"}]
Out[168]= {{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a}}
```

Solutions to Computer Projects and Computations and Explorations

Computer Projects 10

> Given positive integers n and r, list all the r-combinations, with repetition allowed, of the set $\{1, 2, 3, \ldots, n\}$.

Solution: In Section 6.5 of this manual, we showed that the Subsets function could be used to generate combinations with repetition by repeating the elements in the list given as the first argument to Subsets.

To generate the 2-combinations of {1, 2, 3}, for example, we apply the Subsets function to the list consisting of {1, 2, 3}, each repeated twice. Note that the number of repetitions must be the same as r in order to choose r all of the same object.

```
In[169]:= DeleteDuplicates[
           Subsets[Join @@ Table[ConstantArray[i, 2], {i, {1, 2, 3}}], {2}]
          ]
Out[169]= {{1, 1}, {1, 2}, {1, 3}, {2, 2}, {2, 3}, {3, 3}}
```

Refer back to Section 6.5, in the subsection titled "Combinations with Repetition" for information about the use of DeleteDuplicates, Join, Table, and ConstantArray in this application.

We now write a function that accepts n and r as input and produces all r-combinations with repetition allowed.

```
In[170]:= subsetsRepetition[n_Integer, r_Integer] /; n > 0 && r > 0 :=
           Module[{L, i},
            L = Range[n];
            DeleteDuplicates[
             Subsets[Join @@ Table[ConstantArray[i, r], {i, L}], {r}]
            ]
           ]
```

We can obtain all of the 3-combinations of {1, 2, 3, 4, 5} by

```
In[171]:= subsetsRepetition[5, 3]
Out[171]= {{1, 1, 1}, {1, 1, 2}, {1, 1, 3}, {1, 1, 4}, {1, 1, 5}, {1, 2, 2},
           {1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 3}, {1, 3, 4}, {1, 3, 5},
           {1, 4, 4}, {1, 4, 5}, {1, 5, 5}, {2, 2, 2}, {2, 2, 3}, {2, 2, 4},
           {2, 2, 5}, {2, 3, 3}, {2, 3, 4}, {2, 3, 5}, {2, 4, 4}, {2, 4, 5},
           {2, 5, 5}, {3, 3, 3}, {3, 3, 4}, {3, 3, 5}, {3, 4, 4}, {3, 4, 5},
           {3, 5, 5}, {4, 4, 4}, {4, 4, 5}, {4, 5, 5}, {5, 5, 5}}
```

Computations and Explorations 1

> Find the number of possible outcomes in a two-team playoff when the winner is the first team to win 5 out of 9, 6 out of 11, 7 out of 13, and 8 out of 15.

Solution: We will designate the two teams as 1 and 2 and model a playoff as a list of 1s and 2s. For example, {1, 2, 2, 1, 1, 1, 2, 1} is a playoff in which team 1 wins the first game, team 2 wins games 2 and 3, team 1 wins games 4, 5, and 6, team 2 wins game 7, and them team 1 wins game 8. If the winner is the first team to win 5 out of 9 games, then team 1 has won the tournament after 8 games.

We will write a function to produce all of the possible outcomes in a playoff where the winner is the first team to win n out of $2n - 1$ games.

First we will create a small function that will determine, given a list of the outcomes of individual games and the number of games needed to win, whether a team has won the playoff or not. The function will return true if one of the teams has won or false if neither team has reached the threshold for winning.

The function counts the 1s and 2s in the list. If either number is equal to n, it returns true. We will use *Mathematica*'s Count function to determine the number of 1s and 2s in the list. Count requires two

arguments, an expression and a pattern. It returns the number of times the pattern appears in the expression. In this case, the expression will be the list representing the playoff and the pattern will be alternately 1 and 2.

```
In[172]:= playoffWonQ[L_List, n_Integer] :=
           Count[L, 1] == n || Count[L, 2] == n
```

For instance, in our example {1, 2, 2, 1, 1, 1, 2, 1}, the function recognizes that the playoff has been won.

```
In[173]:= playoffWonQ[{1, 2, 2, 1, 1, 1, 2, 1}, 5]

Out[173]= True
```

We will construct the possible outcomes as follows. Begin with a list **outcomes** and a list **S**. Initialize **outcomes** to the empty list and **S** to the list {{1}, {2}}.

Consider the first element of **S**, say **p**. Remove **p** from the list. Then construct the two lists formed by adding 1 and 2 respectively to **p**. For each of these, use **playoffWon** to determine whether or not they are outcomes. If so, they are added to the **outcomes** list, and if not, they are added to the end of **S**. When **S** is empty, then **outcomes** consists of all possible outcomes of the playoff.

Here is the function.

```
In[174]:= allPlayoffs[n_Integer] :=
           Module[{outcomes = {}, S = {{1}, {2}}, p, p1, p2},
            While[S ≠ {},
             p = S[[1]];
             S = Delete[S, 1];
             p1 = Append[p, 1];
             p2 = Append[p, 2];
             If[playoffWonQ[p1, n],
              AppendTo[outcomes, p1], AppendTo[S, p1]];
             If[playoffWonQ[p2, n], AppendTo[outcomes, p2],
              AppendTo[S, p2]];
            ];
            outcomes
           ]
```

We now apply this function to playoffs that are best 3 out of 5.

```
In[175]:= best3of5 = allPlayoffs[3]

Out[175]= {{1, 1, 1}, {2, 2, 2}, {1, 1, 2, 1}, {1, 2, 1, 1}, {1, 2, 2, 2},
           {2, 1, 1, 1}, {2, 1, 2, 2}, {2, 2, 1, 2}, {1, 1, 2, 2, 1},
           {1, 1, 2, 2, 2}, {1, 2, 1, 2, 1}, {1, 2, 1, 2, 2},
           {1, 2, 2, 1, 1}, {1, 2, 2, 1, 2}, {2, 1, 1, 2, 1}, {2, 1, 1, 2, 2},
           {2, 1, 2, 1, 1}, {2, 1, 2, 1, 2}, {2, 2, 1, 1, 1}, {2, 2, 1, 1, 2}}

In[176]:= Length[best3of5]

Out[176]= 20
```

The reader is left to apply the function to the cases called for in the problem and to conjecture a general formula.

Computations and Explorations 3

> Verify that $C(2n, n)$ is divisible by the square of a prime, when $n \neq 1, 2,$ or 4, for as many positive integers n as you can. [The theorem that tells that $C(2n, n)$ is divisible by the square of a prime for $n \neq 1, 2,$ or 4 was proved in 1996 by Andrew Granville and Olivier Ramaré. Their proof settled a conjecture made in 1980 by Paul Erdös and Ron Graham.]

Solution: We will first consider one example to see exactly what we need to do. Then we will write a general function. Consider $n = 3$, the smallest n for which the theorem is true.

First, compute $C(2n, n)$ for $n = 3$.

```
In[177]:= c3 = Binomial[6, 3]
Out[177]= 20
```

To determine whether or not $C(2n, n)$ is divisible by the square of a prime, we could look at its prime factorization. If any of the exponents in the prime factorization are 2 or greater, then we know the number is divisible by the square of the corresponding prime.

We can use the function FactorInteger (first discussed in Section 4.3 of this manual). The FactorInteger function requires one argument, an integer. Its output is a list of the form $\{\{p_1, e_1\}, \{p_2, e_2\}, \ldots, \{p_m, e_m\}\}$, where $p_1, p_2, \ldots, p_m$ are the primes in the prime factorization and the $e_1, e_2, \ldots, e_m$ are the corresponding exponents.

Apply FactorInteger to $C(6, 3)$.

```
In[178]:= FactorInteger[c3]
Out[178]= {{2, 2}, {5, 1}}
```

The result tells us that $C(6, 3) = 2^2 \cdot 5^1$.

We are interested in the exponents of the primes. To extract the second element of each pair, we can apply the Part ([[…]]) operator with the part specification **All,2**.

```
In[179]:= FactorInteger[c3][[All, 2]]
Out[179]= {2, 1}
```

Given the list of exponents, determining whether or not $C(6, 3)$ is divisible by the square of a prime just amounts to checking whether the list contains a value greater than 1.

As is so often the case, *Mathematica* provides a shortcut. The function SquareFreeQ, applied to an integer, outputs True if the integer is not divisible by the square of a number and it returns False if the integer is divisible by a square.

For the example $C(6, 3)$, SquareFreeQ will output False since 20 is divisible by 2^2.

```
In[180]:= SquareFreeQ[c3]
Out[180]= False
```

Note that SquareFreeQ returns False when the assertion is verified and True when the assertion

fails, such as for $n = 4$.

```
In[181]:= SquareFreeQ[Binomial[8, 4]]
Out[181]= True
```

The problem was to verify the conjecture for "as many positive integers n as you can." We will use the TimeConstrained function to run the test for a specified amount of time. The second argument of TimeConstrained is an amount of time, in seconds, that bounds the time that *Mathematica* will spend executing the first argument. For the first argument, we will use an infinite loop that will display values of n for which the statement is found to be false. TimeConstrained takes an optional third argument that is evaluated if the allotted time expires. We will use this argument to display the maximum value of n for which the statement was checked. Note that this maximum n must be one less than the index used in the loop.

```
In[182]:= TimeConstrained[
           n = 1;
           While[True,
            If[SquareFreeQ[Binomial[2 * n, n]],
             Print["Found counterexample: ", n]];
            n++
           ],
           1 , (* 1 second limit *)
           Print["Checked through n=", n - 1]
          ]
          Found counterexample: 1
          Found counterexample: 2
          Found counterexample: 4
          Checked through n=8831
```

Exercises

1. Build a recursive version of **subsetSumCount**, using the ideas of **findBitStrings**. Your function should determine all subsets of a given set whose sum is less than a target value. Rather than considering all sets, it should build potential sets recursively using the fact that once a set has sum larger than the target no larger set of positive integers can have smaller sum. Compare the performance of your procedure with **subsetSumCount**.
2. Create a function **findDecreasing** by modifying **findIncreasing** in order to determine a strictly decreasing subsequence of maximal length.
3. Modify the Patience algorithm to find *all* of the strictly increasing subsequences of maximal length.
4. Use *Mathematica* to show that, in Example 11 of Section 6.2, n positive integers not exceeding $2n$ are not sufficient to guarantee that one integer divides one of the others.

5. Use *Mathematica* to determine how many different strings can be made from the word "PAPARAZZI" when all the letters are used, when any number of letters are used, when all the letters are used and the string begins and ends with the letter "Z", and when all the letters are used and the three "A"s are consecutive.

6. Suppose that a certain Mathematics Department has m male faculty and f female faculty. Write a *Mathematica* function to find all committees with $2k$ members in which both sexes are represented equally.

7. Use *Mathematica* to prove the identity

$$\binom{n+1}{k} = \frac{n+1}{k}\binom{n}{k-1}$$

for positive integers n and k with $k \le n$.

8. Use *Mathematica* to prove Pascal's identity:

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$$

for all positive integers n and k with $k \le n$.

9. Use *Mathematica* to generate many rows of Pascal's triangle. See if you can formulate any conjectures involving identities satisfied by the binomial coefficients. Use *Mathematica* to help you verify that your conjecture is true by using the techniques at the end of Section 6.4 of this manual.

10. Write a function that mixes the techniques used in **`binomialF`** and **`binomialR`** to generate the rows of Pascal's triangle from row a to row b for $b > a > 0$.

11. Use *Mathematica* to count and list all solutions to the equation

$$x_1 + x_2 + x_3 + x_4 = 25$$

where x_1, x_2, x_3, and x_4 are non-negative integers. Also count and list all solutions such that $x_1 \ge 1$, $x_2 \ge 2$, $x_3 \ge 3$, and $x_4 \ge 4$.

12. Generate a large triangle of Stirling numbers of the second kind and look for patterns that suggest identities among the Stirling numbers. Also see if you can make any conjectures about the relationship between Stirling numbers and the binomial coefficients.

13. Implement the algorithm described in the "Generating partitions" subsection of Section 6.5 of this manual.

14. Write a *Mathematica* function that takes as input three positive integers n, k, and i, and returns the ith multinomial, in lexicographic order, of the polynomial $(x_1 + x_2 + \cdots + x_k)^n$. Write its inverse; that is, given a multinomial, the inverse should return its index (position) in the sorted polynomial.

15. Implement Algorithm 2 of Section 6.6 for generating the next largest bit string.

16. Implement Algorithm 3 of Section 6.6 for generating the next r-combination.

17. Write a *Mathematica* function to compute the Cantor expansion of an integer. (See the prelude to Exercise 14 of Section 6.6 of the text.)

18. Implement the algorithm for generating the set of all permutations of the first n integers using the bijection from the collection of all permutations of the set $\{1, 2, \ldots, n\}$ to the set $\{1, 2, \ldots, n!\}$ described prior to Exercise 14 of Section 6.6 of the textbook.