6 Counting

Introduction

This chapter presents a variety of techniques that are available in Maple 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.

Most of the Maple commands relevant to this chapter dwell in the **combinat** package. Since this package will be used extensively in this chapter, we load it now.

```
> with(combinat):
```

Advanced readers may wish to explore the **combstruct** package on their own. This package will not be discussed in this manual as the **combinat** package is sufficient for the material covered here. The **combstruct** package contains commands related to combinatorial structures, and can be thought of as generalizing some of the commands provided by **combinat**.

6.1 The Basics of Counting

In this section, we will see how Maple 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 will see how the principles of counting can be used to count the number of operations used by a Maple procedure. This section concludes by using Maple procedures to solve counting problems by enumerating all the possibilities.

Basic Examples

We begin with two basic examples to demonstrate the use of some useful Maple commands.

Counting One-to-One Functions

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

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

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 Maple. For instance, the statement below computes the number of one-to-one functions from a set of four elements to a set of 20 elements.

```
> 20 · 19 · 18 · 17
```

$$116\,280 \qquad \textbf{(6.1)}$$

For larger values of m, it is more convenient to use the **mul** command. The **mul** command is used to multiply a sequence of values. Its syntax is similar to the syntax for **seq**. 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, in the most common usage, has the form **i=a..b** with the range **a..b** representing the values the variable is to take.

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

```
> mul(20 - i, i = 0..3)
```

$$116\,280 \tag{6.2}$$

The second argument indicates that the index variable **i** 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.

```
> mul(300 - i, i = 0..11)
```

$$425\,270\,752\,192\,695\,317\,567\,218\,560\,000 \tag{6.3}$$

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 consisting of uppercase letters or digits. In addition, 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

```
> P[6] := 36^6 - 26^6
```

$$P[6] := 1\,867\,866\,560 \tag{6.4}$$

```
> P[7] := 36^7 - 26^7
```

$$P[7] := 70\,332\,353\,920 \tag{6.5}$$

```
> P[8] := 36^8 - 26^8
```

$$P[8] := 2\,612\,282\,842\,880 \tag{6.6}$$

Thus, the total number of possible passwords is

```
> P := P[6] + P[7] + P[8]
```

$$P := 2\,684\,483\,063\,360 \tag{6.7}$$

We can use the **add** command to make this calculation a bit easier. The **add** command has the same syntax as **mul**, but is used to add its arguments rather than multiply them.

```
> add(36^i - 26^i, i = 6..8)
```

$$2\,684\,483\,063\,360 \tag{6.8}$$

This makes it easier to compute the number of valid passwords for larger ranges. For instance, it is common to require passwords to be between 8 and 64 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 statement below.

```
> add(36^i - 26^i, i = 8..64)
```

$$4\,126\,620\,251\,202\,066\,828\,551\,300\,229\,999\,712\,527\,129\,717\,696\,057\,592\,889\,450\,099\,703\backslash$$
$$219\,511\,292\,080\,116\,014\,779\,039\,630\,823\,409\,920 \qquad \textbf{(6.9)}$$

Working with Large Integers

Maple'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 Maple compute this value, we just enter the statement. (Note that since exponentiation is not associative, parentheses are required in this calculation.)

```
> DNAsequences := 4^(10^8)
```

[*Length of output exceeds limit of 1 000 000*] **(6.10)**

Maple reports that the length of the output exceeds a limit. This does not imply that Maple has not computed it, it only means that displaying the integer would require excessive space.

Maple 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.

```
> DNAsequences mod 1000
```

376 **(6.11)**

We can calculate the number of digits in the result by applying the **ilog10** command. This command computes an integer approximation of the base 10 logarithm of its argument. Specifically, as long as the argument x is a real number, the result of **ilog10(x)** is an integer r such that $10^r < |x| < 10^{r+1}$. In particular, the result is one less than the number of digits of x.

Apply **ilog10** to the result for the number of possible sequences of bases in DNA.

```
> ilog10(DNAsequences)
```

60 205 999 **(6.12)**

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

```
> evalf((6.12)/(64 · 45))
```

$$20\,904.86076 \qquad \textbf{(6.13)}$$

pages to print the entire number.

Variable Names in Maple

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. We will calculate the number of possible names in Maple.

In Maple, the basic form of a name is a letter possibly followed by additional letters, digits, or underscores. Maple considers uppercase distinct from lowercase. There are 52 uppercase or lowercase letters that can be used as the first character. Including the underscore and the 10 digits, there are 63 possibilities for each character following the first.

The maximum length of a name depends on the computer. On a 32-bit system, the maximum length is 268 435 439 characters. On a 64-bit machine, the maximum is over 34 billion characters long. It is unlikely that you would ever use such a name, but you could if you wanted to.

How many possible names are there? The total number of possible names on a 32-bit system is $\sum_{i=0}^{268\,435\,438} 52\,63^i$. Attempting to calculate this would be rather time consuming.

We will ask a more reasonable question. How many names have at most 15 characters? To answer this question, we use the **add** command, as we did above. The formula is $52 \cdot 63^i$ where the index variable i represents the number of characters in the name beyond the first and ranges from 0 to 14.

```
> add(52 · 63^i, i = 0..14)
```

$$819\,822\,618\,169\,347\,817\,599\,853\,876 \qquad \textbf{(6.14)}$$

We see that even limiting ourselves to a maximum of 15 characters, there are over 800 septillion distinct Maple names. (The number above is slightly inaccurate because it does not exclude Maple keywords and other protected names that you are not allowed to assign values to. Of course, those are relatively insignificant.)

Counting Operations in a Procedure

Next, we consider an example of counting the number of operations performed by a procedure. 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, consider the **MakePostage** procedure from Section 5.1 of this manual. Recall that the purpose of this procedure was to determine the numbers of stamps of two given denominations that are required to make a given amount of postage. Here is the procedure definition again.

```
1 MakePostage := proc(A::posint,B::posint,postage::posint)
2     local a, b;
3     for a from 0 to floor(postage/A) do
```

```
        for b from 0 to floor(postage/B) do
            if A*a + B*b = postage then
                return [a,b];
            end if;
        end do;
    end do;
    return FAIL;
end proc:
```

We will count the number of multiplications and additions that this procedure requires in the worst case. The **return** command means that once the procedure has found a way to make the desired amount of postage, execution is immediately terminated. Therefore, knowing the number of iterations used for a particular input value is equivalent to knowing the output of the procedure. If there was a formula for that, we would not need the procedure. By considering the worst-case scenario, we can get an idea of the complexity of the algorithm without having to execute the procedure.

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 $\left\lfloor \frac{postage}{A} \right\rfloor$ and the inner loop will range from 0 to $\left\lfloor \frac{postage}{B} \right\rfloor$. Thus, the number of times the if statement is executed is $\left(\left\lfloor \frac{postage}{A} \right\rfloor + 1\right)\left(\left\lfloor \frac{postage}{B} \right\rfloor + 1\right)$. Therefore, in the worst case, the **MakePostage** procedure 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 Maple procedures 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 69 in Section 6.1.)

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

In order to generate all of the possible subsets of $\{1, 2, \ldots 10\}$, we use the **subsets** command, first introduced in Section 2.1 of this manual. The **subsets** command is part of the **combinat** package.

The **subsets** command accepts one argument: the set (or list) whose subsets are to be generated. It returns a table with two entries. The **nextvalue** entry is a procedure that takes no arguments and that, when executed, returns a subset. The **finished** entry is a Boolean value that is initially false but is set to true once the **nextvalue** procedure has returned the final subset.

Here is an example of using the **subsets** command to print all of the subsets of the set $\{1, 2\}$.

```
> subs12 := subsets({1, 2}):

> while not subs12[finished] do
    print(subs12[nextvalue]( ))
  end do
```

$$\emptyset$$
$$\{1\}$$
$$\{2\}$$
$$\{1, 2\}$$

(6.15)

The assignment to the result of **subsets** establishes **subs12** as the name storing the table. Then, **subs12[finished]** accesses the Boolean value of the **finished** entry from the table. As this is false until all subsets have been produced, its logical negation is a useful control for a while loop. To produce the subsets, **subs12[nextvalue]()** calls the **nextvalue** procedure, which returns the "next" subset.

We will use **subsets** to solve the problem of counting the number of subsets of $\{1, 2, 3, \ldots, 10\}$ whose sum is less than 15. Instead of printing all subsets as the simple loop above does, we will test the subset produced by **nextvalue** to see if its sum is less than 15.

To calculate the sum of the elements of a set, we use the **add** command. To add the elements of a list or set, the **add** command will accept the list or set as the sole argument. For example, to add the elements of the set $\{1, 3, 7, 8\}$, we can execute the following expression.

```
> add({1, 3, 7, 8})
```

$$19$$

(6.16)

Since the problem, as stated, was to count the number of subsets with sum less than 15, we use a variable that is initialized to 0 and incremented each time a subset has the desired property. We generalize the problem a bit and write a procedure that accepts a set of positive integers and a target value and counts all the subsets of the given set whose elements sum to a value less than the target.

```
SubsetSumCount := proc(S::set(posint),target::posint)
   uses combinat;
   local count, P, s, x;
   count := 0;
   P := subsets(S);
   while not P[finished] do
      s := P[nextvalue]();
      x := add(s);
      if x < target then
         count := count + 1;
      end if;
   end do;
   return count;
end proc:
```

Applying the procedure to $\{1, 2, 3, \ldots, 10\}$ and 15 will answer the original question.

```
> SubsetSumCount({1,2,3,4,5,6,7,8,9,10},15)
                              99
```
(6.17)

Bit Strings

For the second example, we will consider a problem similar to Example 21. How many bit strings of length 10 do not have three consecutive 1s?

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 22 of Section 6.1, and especially Figure 4, 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. In addition, 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 procedures. The first procedure 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 procedure will apply the first to an entire set of bit strings.

First, we implement the procedure that extends a single bit string. The parameter to this procedure will be a single bit string. The procedure will first create a new bit string by appending a zero to the input.

Then, the procedure 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 sublist consisting of the last two entries with the selection operator: **L[-2..-1]**. 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 procedure 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.

```
AddBit := proc(L::list)
   local L0, L1;
   L0 := [op(L),0];
   if L[-2..-1]=[1,1] then
      return L0;
   else
      L1 := [op(L),1];
      return L0,L1;
   end if;
end proc:
```

We test this procedure on two examples: [1, 0, 1, 1] should produce only [1, 0, 1, 1, 0], while applying the procedure to that result should produce [1, 0, 1, 1, 0, 0] and [1, 0, 1, 1, 0, 1].

```
> AddBit([1, 0, 1, 1])
```

$$[1, 0, 1, 1, 0] \tag{6.18}$$

```
> AddBit(%)
```

$$[1, 0, 1, 1, 0, 0],\ [1, 0, 1, 1, 0, 1] \tag{6.19}$$

Now, we write the main procedure. 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, it will return 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.

```
FindBitStrings := proc(n::nonnegint)
   local S, s, T;
   if n = 2 then
      return {[0,0],[0,1],[1,0],[1,1]};
   else
      S := FindBitStrings(n-1);
      T := {};
      for s in S do
         T := T union {AddBit(s)};
      end do;
      return T;
   end if;
end proc:
```

Applying the procedure to 10 and using **nops** will give us the number of bit strings of length 10 that do not include three successive 1s.

```
> nops(FindBitStrings(10))
```

$$504 \tag{6.20}$$

6.2 The Pigeonhole Principle

In this section, we will see how Maple 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 **ceil** command. This command 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 Maple, this can be computed by the following statement.

> $\text{ceil}\left(\frac{25000000}{8000000}\right)$

4 **(6.21)**

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. Then, there 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 procedure 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 described in the solution of Example 10.

First, we will calculate the numbers $a_1, a_2, \ldots, a_d$ with each a_j equal to the sum of the first j 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 **member** command to check to see if this value is in **A**. The **member** command requires two arguments: the first is the element being sought and the second is the list (or set) to be searched. The command returns true if the element is found and false otherwise. The **member** command also accepts an unevaluated (i.e., enclosed in right single quotes) name as a third, optional argument. If the element is found in the list, then this name is assigned to the position of the first occurrence. For example,

> *member*("*d*", ["*a*", "*b*", "*c*", "*d*", "*e*", "*f*", "*g*"], *'p'*)

true **(6.22)**

> *p*

4 **(6.23)**

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 $i + 1$ through j are the positions of the consecutive subsequence with the desired sum. The procedure returns the starting and ending positions as well as the subsequence.

Here is the procedure.

```
FindSubSum := proc(L::list(posint),T::posint)
   local A, i, j;
   A := [L[1]];
   for i from 2 to nops(L) do
      A := [op(A),A[i-1]+L[i]];
   end do;
   for i from 1 to nops(L) do
      if member(A[i]+T,A,'j') then
         return i+1,j,L[i+1..j];
```

```
10        end if;
11      end do;
12      return FAIL;
13  end proc:
```

We can apply our procedure 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 find the consecutive days during which the team played 14 games.

```
> 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)
    6, 13, [3, 1, 1, 1, 1, 3, 1, 3]                                          (6.24)
```

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 procedure 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 12	11 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 12	11 18	16	
Step 6: Play 3. Checking the first pile, 3 < 7, so 3 is played on the first pile.	3 7 12	 11 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 12	 11 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 12	 11 18	 16	 17 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** procedure 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

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

First, we need to know which "step" we are 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 selection-assignment syntax **piles[i] := x;**. When a new pile is added to the list, we extend the list via the usual syntax **[op(piles),x];**.

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 table, **pointers**, whose indices will be the members of the sequence and whose entries will be set to the previous pile's top card. (We use the name **pointers** for this table because of the similarity to the "linked list" structure used in some programming languages.) For those numbers played in the first pile, the value in the table will be set to **NULL**.

The algorithm consists of two stages. The first stage will be the game stage. After initializing **piles** to the empty list and **pointers** to the empty table, we begin 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. Initialize a variable **whichpile** to 0.
2. Use a for loop from 1 to the current number of piles. Within the for loop, compare the current card to the top of each pile. If the current card is smaller than the value in **piles**, set **whichpile** equal to that pile index.
3. Once the loop exits, check the value of **whichpile**. If it is 0 following termination of the loop, 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 not 0 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** table. 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 0, 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** table associated to that value. Since we are 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 the expression **[pointers[iList[1]],op(iList)]**. 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 procedure returns **iList**.

Here, finally, is the procedure.

```
FindIncreasing := proc(S::list)
   local piles, pointers, step, whichpile, p, iList;
   piles := [];
   pointers := table();
   # game stage
   for step from 1 to nops(S) do
      # playing the card
      whichpile := 0;
      for p from 1 to nops(piles) do
         if S[step] < piles[p] then
            whichpile := p;
```

```
            break;
        end if;
    end do;
    if whichpile = 0 then
        piles := [op(piles),S[step]];
    else
        piles[whichpile] := S[step];
    end if;
    # update pointers
    if whichpile = 1 or nops(piles) = 1 then
        pointers[S[step]] := NULL;
    elif whichpile = 0 then
        pointers[S[step]] := piles[-2];
    else
        pointers[S[step]] := piles[whichpile-1];
    end if;
  end do;
  # backtracking stage
  iList := [piles[-1]];
  while pointers[iList[1]] <> NULL do
    iList := [pointers[iList[1]],op(iList)];
  end do;
  return iList;
end proc:
```

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

```
> FindIncreasing([8,11,9,1,4,6,12,10,5,7])
```

$$[1,4,5,7] \tag{6.25}$$

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, by the generalized pigeonhole principle, 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. Hence, some stack has $n + 1$ values. However, 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 subsequence of length $n + 1$ or there is a stack of size $n + 1$ and hence a decreasing subsequence of length $n + 1$.

6.3 Permutations and Combinations

The **combinat** package contains many functions pertaining to counting and generating combinatorial structures. We will be using **combinat** commands extensively in this section.

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.

```
> 6!
                              720                                    (6.26)
```

The **factorial** command can be used instead of the exclamation mark if you prefer. Otherwise they are equivalent. (Factorial does not depend on the **combinat** package.)

Counting Permutations

To compute the number of permutations, Maple provides the **numbperm** command. This command can be used in a few different ways.

You can give only one argument, an integer.

```
> numbperm(5)
                              120                                    (6.27)
```

In this case, Maple computes the number of permutations, that is, the number of 5-permutations of a set with five elements. This, of course, is the same as computing 5!.

Instead of an integer as the only argument, you can instead provide a set or list of objects.

```
> numbperm({"a", "b", "c", "d", "e"})
                              120                                    (6.28)
```

With this argument, **numbperm** computes the number of permutations of the given set (or list). Since the set had five elements, this result is the same as the previous value.

To compute the number of r-permutations of a set with n objects, use **numbperm** with a second argument indicating the value of r. The number of 4-permutations of a set with seven distinct objects, that is, $P(7, 4)$, is computed by the following command.

```
> numbperm(7, 4)
                              840                                    (6.29)
```

Once again, the first argument could be given as a set or a list of objects instead of the number of objects.

```
> numbperm(["a","b","c","d","e","f","g"],4)
                                840                                (6.30)
```

You may wonder what the purpose is of allowing the first argument to **numbperm** to be a set or list rather than requiring it to always be a positive integer. We will explore this more in Section 6.5, but briefly, the reason is that the objects in a list do not have to be different. By giving the first element as a list of not-necessarily distinct objects, **numbperm** will compute the number of arrangements of those objects, taking into account any duplication of elements.

Listing Permutations

To obtain a list of all permutations, Maple provides the **permute** command. The syntax is the same as **numbperm**, but instead of reporting the number of permutations, a list of the permutations is provided.

Once again, the first argument can be an integer or a set or a list. If the first argument is given as an integer n, Maple will display the permutations of the set $\{1, 2, \ldots, n\}$. For example, the following command lists the permutations of the first four integers.

```
> permute(4)
    [[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]]           (6.31)
```

Providing a specific list or set as the first argument will produce the permutations of the objects in the given set or list. For example, the permutations of the letters a, b, and c are shown below.

```
> permute({"a","b","c"})
    [["a","b","c"],["a","c","b"],["b","a","c"],["b","c","a"],
      ["c","a","b"],["c","b","a"]]                                 (6.32)
```

To obtain the r-permutations instead, you must provide r as the second argument. The following lists the 2-permutations of a set with four distinct objects.

```
> permute(4,2)
    [[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],[3,1],[3,2],[3,4],
      [4,1],[4,2],[4,3]]                                           (6.33)
```

Or you can provide your own objects to be r-permuted.

```
> permute({"a","b","c","d","e"},2)
    [["a","b"],["a","c"],["a","d"],["a","e"],["b","a"],["b","c"],
      ["b","d"],["b","e"],["c","a"],["c","b"],["c","d"],["c","e"],
      ["d","a"],["d","b"],["d","c"],["d","e"],["e","a"],["e","b"],
      ["e","c"],["e","d"]]                                         (6.34)
```

Random Permutations

Maple also provides a command, **randperm**, that will produce a randomly chosen permutation.

Once again, the argument to **randperm** can be an integer or a list or set of objects. If an integer n is given, then **randperm** returns a randomly selected permutation of the set $\{1, 2, \ldots, n\}$.

```
> randperm(10)
```

$$[3, 8, 9, 5, 1, 7, 2, 10, 4, 6] \qquad \textbf{(6.35)}$$

If a list or set is provided as the first argument, then the random permutation is produced using the elements of the given set or list.

```
> randperm(["a", "b", "c", "d"])
```

$$[\text{“}d\text{”}, \text{“}a\text{”}, \text{“}c\text{”}, \text{“}b\text{”}] \qquad \textbf{(6.36)}$$

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

Unlike the **numbperm** and **permute** commands, **randperm** does not accept a second argument. (If one is given, it is ignored.) In order to produce a random r-permutation, you must use one of the following approaches.

Suppose you need a random 4-permutation of the set {“a”, “b”, “c”, “d”, “e”, “f”}. Your first choice is to apply the **randperm** command to the set, and then use the selection operator to select the first four elements.

```
> randperm({"a", "b", "c", "d", "e", "f"})[1..4]
```

$$[\text{“}f\text{”}, \text{“}b\text{”}, \text{“}a\text{”}, \text{“}e\text{”}] \qquad \textbf{(6.37)}$$

The second option is to first use **randcomb** (described below) to obtain a random subset of size 4 and then apply **randperm** to the randomly chosen subset. This approach will be demonstrated below, after describing the commands relevant to combinations. Both approaches will produce a random r-permutation of the given objects. The second approach is faster to execute, however.

Combinations

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

Counting Combinations

The number of combinations is obtained with the **numbcomb** command. Like **numbperm**, this command accepts one or two arguments and the first argument can be a number or a set or a list.

If you provide only one argument to **numbcomb**, it returns the total number of combinations. For instance,

```
> numbcomb(5)
```

$$32 \qquad \textbf{(6.38)}$$

indicates that there are 32 ways to choose some (including all or none) of five objects. In other words, there are 32 subsets of a set of 5 elements.

The first argument can also be a set or a list.

```
> numbcomb({"a","b","c","d","e"})
```

32 **(6.39)**

To compute the number of r-combinations, you provide r as the second argument. Once again, the first argument can be a set or list or a positive integer indicating the size of the set the objects are to be drawn from. The following compute $C(52, 5)$ and the number of ways to choose 3 from the set of vowels.

```
> numbcomb(52, 5)
```

2 598 960 **(6.40)**

```
> numbcomb({"a","e","i","o","u"}, 3)
```

10 **(6.41)**

The text mentions that $C(n, r)$ is also referred to as a binomial coefficient. In Maple, the command **binomial** can also be used to compute $C(n, r)$. For example, $C(52, 5)$ can be computed by:

```
> binomial(52, 5)
```

2 598 960 **(6.42)**

Note that **binomial** and **numbcomb** agree when given two nonnegative integers as arguments. However, they are not identical commands. For one, **binomial** requires two arguments which must evaluate to algebraic expressions and cannot accept a list or set as the first argument. On the other hand, **binomial** uses a more general formula that can compute with rational and floating-point arguments. In addition, **binomial** is not part of the **combinat** package and is thus available without needing to load the package.

Listing Combinations

The **choose** command is the combination analog of **permute**.

Given only one argument, **choose** produces all possible combinations of every size. If the argument is a set or list, it will list the subsets or sublists of the argument.

```
> choose({"a","b","c","d"})
```

{∅, {"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"}} **(6.43)**

```
> choose(["a","b","c","d"])
```

[[ ], ["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"]] **(6.44)**

Note that if the argument is a set, the result is a set of sets. On the other hand, if the argument is a list, the result is a list of lists. Many Maple commands accept either sets or lists as arguments, and they typically return the same kind of object they were given. This was not the case for **permute**, because in that case order is relevant, so it must return lists.

If the argument is a nonnegative integer n, it will use $[1, 2, \ldots, n]$ by default.

```
> choose(3)
    [[ ], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]                    (6.45)
```

Note that with only one argument, **choose** behaves very similarly to **powerset**, which was discussed in Section 2.1 of this manual. The **powerset** command accepts only one argument, which can be a set, list, or nonnegative integer. It returns the set of all subsets (or the list of all sublists) just as **choose** does.

```
> powerset({"a", "b", "c", "d"})
    {∅, {"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"}}                  (6.46)
```

With a second argument given to **choose**, you can specify the number of objects to be selected.

```
> choose({"a", "b", "c", "d"}, 2)
    {{"a", "b"}, {"a", "c"}, {"a", "d"}, {"b", "c"}, {"b", "d"},
      {"c", "d"}}                                                              (6.47)

> choose(5, 2)
    [[1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4], [2, 5], [3, 4], [3, 5], [4, 5]]   (6.48)
```

The **powerset** command does not allow a second argument.

Random Combinations

The **randcomb** command is used to produce a random combination.

As opposed to **randperm**, **randcomb** requires two arguments. The first can be a set or list of objects or a positive integer, and the second must be a nonnegative integer.

```
> randcomb({"a", "b", "c", "d", "e"}, 3)
    {"a", "c", "e"}                                                            (6.49)

> randcomb(5, 3)
    {2, 4, 5}                                                                  (6.50)
```

As mentioned earlier, **randcomb** can be combined with **randperm** to produce a random permutation of a specified size. To obtain a random 4-permutation of {"a", "b", "c", "d", "e", "f"}, for example, first use **randcomb** to pick a random 4-combination. Then, apply **randperm** to the selected combination.

```
> randcomb({"a", "b", "c", "d", "e", "f"}, 4)
    {"a", "c", "d", "f"}                                                       (6.51)

> randperm(%)
    ["f", "c", "a", "d"]                                                       (6.52)
```

You can combine them into one statement, as follows.

```
> randperm(randcomb({"a","b","c","d","e","f"},4))
```

["d","e","a","f"] **(6.53)**

Circular Permutations

The preamble to Exercise 42 in 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 will write a procedure to list all of the circular r-permutations of n people. Having such a procedure 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).

Here is a procedure that, given a list representing a circular permutation, will rotate the permutation by one position.

```
RotatePerm := proc(P::list)
   return [op(2..-1,P),P[1]];
end proc:
```

For example, the seating Abe, Carol, Barbara, is the same as Carol, Barbara, Abe.

```
> RotatePerm(["Abe","Carol","Barbara"])
```

["Carol","Barbara","Abe"] **(6.54)**

Note that rotations start to repeat.

```
> RotatePerm(%)
```

["Barbara","Abe","Carol"] **(6.55)**

```
> RotatePerm(%)
```

["Abe","Carol","Barbara"] **(6.56)**

In fact, for an r-permutation, after r rotations, the list will return to its original state.

Equality of Circular Permutations

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 procedure 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.

```
CPEquals := proc(L1::list,L2::list)
   local i, Lr;
   if L1 = L2 then
      return true;
   end if;
   Lr := L2;
   for i from 1 to nops(L2)-1 do
      Lr := RotatePerm(Lr);
      if L1 = Lr then
         return true;
      end if;
   end do;
   return false;
end proc:
```

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

```
> CPEquals(["Charles","Helen","Dean"],["Helen","Dean","Charles"])
```

true **(6.57)**

Listing all Circular Permutations

Now we are prepared to write a procedure that lists all circular permutations. First, we will use the **permute** command to generate all r-permutations of n people. We will initialize the set 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.

Within a while loop, consider the first element in the list of all permutations. Use **CPEquals** 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 procedure will accept a set as the first argument and r, the number that can be seated at the table, as the second argument.

```
AllCP := proc(S::set,r::posint)
   local allP, allCP, isnew, p;
   allP := combinat[permute](S,r);
   allCP := {allP[1]};
   allP := allP[2..-1];
   while allP <> [] do
      isnew := true;
      for p in allCP do
         if CPEquals(allP[1],p) then
            isnew := false;
            break;
         end if;
      end do;
      if isnew then
```

```
15          allCP := allCP union {allP[1]};
16        end if;
17        allP := allP[2..-1];
18    end do;
19    return allCP;
20 end proc:
```

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”}.

```
> AllCP({"Abe","Barbara","Carol","Dean","Eve"}, 3)
```

{[“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”]} **(6.58)**

```
> nops(%)
```

20 **(6.59)**

It is left to the reader to experiment with other starting sets and values of r to determine a formula. Note that the procedures 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 Maple to compute binomial coefficients, to generate Pascal’s triangle, and to verify identities.

The Binomial Theorem

Recall from the previous section that the Maple command **binomial** can be used to compute $\binom{n}{r}$, which is another notation for $C(n, r)$. With n and r positive integers, this command produces the same result as **numbcomb**. The **binomial** command is designed to be more general, in that it will compute coefficients that appear in Newton’s generalized binomial theorem. The generalization is beyond the scope of this manual.

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 Maple, this can be done easily with the **expand** command. The **expand** command requires one argument, an algebraic expression. It returns the result of "expanding" the expression, that is, of distributing products over sums.

```
> expand((x + y)^5)
```

$$x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5 \qquad \textbf{(6.60)}$$

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** command with first argument 30 and second 12 will produce this value.

```
> binomial(30, 12)
```

$$86\,493\,225 \qquad \textbf{(6.61)}$$

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 $(2x-3y)^{25}$ 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

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

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

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

This is

```
> binomial(25, 13) · 2^12 · (−3)^13
```

$$-33\,959\,763\,545\,702\,400 \qquad \textbf{(6.62)}$$

Pascal's Triangle

As we have seen, it is very easy to compute binomial coefficients with Maple. To compute row n of Pascal's triangle, we apply the **binomial** command with the second argument ranging from 0 to n. This can be done with the **seq** command. For example, the 25th row of Pascal's triangle is shown below.

```
> seq(binomial(25, k), k = 0 ..25)
```

$$1,\ 25,\ 300,\ 2300,\ 12\,650,\ 53\,130,\ 177\,100,\ 480\,700,\ 1\,081\,575,\ 2\,042\,975,$$
$$3\,268\,760,\ 4\,457\,400,\ 5\,200\,300,\ 5\,200\,300,\ 4\,457\,400,\ 3\,268\,760,$$
$$2\,042\,975,\ 1\,081\,575,\ 480\,700,\ 177\,100,\ 53\,130,\ 12\,650,\ 2300,$$
$$300,\ 25,\ 1 \qquad \textbf{(6.63)}$$

When calculating a single binomial coefficient or an isolated row, applying the formula may be the most efficient approach. However, 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 will write two procedures for computing binomial coefficients. The first will simply apply the formula $\frac{n!}{r!\,(n-r)!}$. The second will be a recursive procedure making use of Pascal's identity and symmetry. Then, we will compare the performance of the two procedures in building Pascal's triangle.

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

```
BinomialF := proc(n::nonnegint, k::nonnegint)
   return n!/(k!*(n-k)!);
end proc:
```

A Recursive Procedure

The second procedure 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 procedure will work as follows. We will declare the **remember** option so that Maple will automatically create a remember table. The body of the procedure will consist of an if-elif-else statement. In case the second argument, is 0, the procedure will return 1. That forms the basis case of the recursion. The elif clause will test whether $2k > n$. In this case, we make use of symmetry and call **BinomialR** on n and $n-k$. Finally, in the else clause, we apply Pascal's identity, making recursive calls to **BinomialR**. Here is the implementation.

```
BinomialR := proc(n::nonnegint, k::nonnegint)
   option remember;
   if k=0 then
      return 1;
   elif 2*k > n then
      return BinomialR(n,n-k);
   else
```

```
8         return BinomialR(n-1,k-1) + BinomialR(n-1,k);
9     end if;
10 end proc:
```

We use our two procedures to build Pascal's triangle using a loop to control the value of **n** and the **seq** command.

> **for** n **from** 0 **to** 5 **do**
> $seq(BinomialF(n,k), k = 0..n)$
> **end do**

$$1$$
$$1,\ 1$$
$$1,\ 2,\ 1$$
$$1,\ 3,\ 3,\ 1$$
$$1,\ 4,\ 6,\ 4,\ 1$$
$$1,\ 5,\ 10,\ 10,\ 5,\ 1 \qquad \textbf{(6.64)}$$

> **for** n **from** 0 **to** 5 **do**
> $seq(BinomialR(n,k), k = 0..n)$
> **end do**

$$1$$
$$1,\ 1$$
$$1,\ 2,\ 1$$
$$1,\ 3,\ 3,\ 1$$
$$1,\ 4,\ 6,\ 4,\ 1$$
$$1,\ 5,\ 10,\ 10,\ 5,\ 1 \qquad \textbf{(6.65)}$$

Comparing Performance
Now, we will compare the performance of the two procedures.

First, we use each of them to compute the first 1000 rows of Pascal's triangle and compare the time it takes. We must use **forget** to clear the remember table for **BinomialR**.

> $forget(BinomialR)$

> $st := time():$
> **for** n **from** 0 **to** 1000 **do**
> $seq(BinomialF(n,k), k = 0..n)$
> **end do**:
> $time() - st$

$$9.524 \qquad \textbf{(6.66)}$$

> $st := time():$
> **for** n **from** 0 **to** 1000 **do**
> $seq(BinomialR(n,k), k = 0..n)$
> **end do**:
> $time() - st$

$$1.900 \qquad \textbf{(6.67)}$$

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

However, if we compute some isolated values, the situation is reversed. We will generate some random values of n between 100 and 1000 and random values of k between 0 and 100.

Recall that the **rand** command applied to a range of integers returns a procedure that generates random numbers in that range. For instance, the statement below assigns to the name **randN** a procedure. Calling **randN()** produces a random number.

```
> randN := rand(100..1000):
> randN()
                              286                                    (6.68)
```

Likewise for **randK**. Note that calling **rand** with a single integer as the argument is the same as the range from 0 to that value.

```
> randK := rand(100):
> randK()
                               15                                    (6.69)
```

Next, we generate a list of 100 randomly generated (n, k) pairs. We suppress the output since the list will be long, but note that the elements of this list are 2-element lists.

```
> NKlist := [seq([randN(), randK()], i = 1..100)]:
```

Now, we apply the two procedures to the elements of the randomly generated list. Note that we pass the arguments to both procedures with the syntax **BinomialF(op(NKpair))**. The **op** command applied to **NKpair**, an element of the **NKlist**, takes the 2-element list **NKpair** and returns the underlying expression sequence. The procedures were written to accept two integer arguments, so passing them the 2-element list (which is only one argument) would raise an error.

```
> st := time():
  for NKpair in NKlist do
    BinomialF(op(NKpair))
  end do:
  time() - st
                              0.019                                  (6.70)
> forget(BinomialR)
> st := time():
  for NKpair in NKlist do
    BinomialR(op(NKpair))
  end do:
  time() - st
                              0.210                                  (6.71)
```

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

certain isolated values, the recursive procedure still had to calculate all of the results for lower values of n and k, which the nonrecursive procedure did not need to compute.

Verifying Identities

Maple can help verify identities regarding the binomial coefficients. If you enter an expression using the binomial command that includes symbolic arguments, Maple will echo the statement. First, we ensure that **n** and **k** are unassigned.

```
> n := 'n';
  k := 'k'
```

$$n := n$$
$$k := k \qquad \textbf{(6.72)}$$

```
> binomial(n, k)
```

$$\binom{n}{k} \qquad \textbf{(6.73)}$$

To have Maple produce an expression using factorials, use the **convert** command with the binomial expression as the first argument and the keyword **factorial** as the second argument.

```
> convert(binomial(n, k), factorial)
```

$$\frac{n!}{k!\,(n-k)!} \qquad \textbf{(6.74)}$$

Symmetry

As a first example, we will 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.

```
> left := binomial(n, k):
```

```
> right := binomial(n, n - k):
```

Remember that we need to use the **evalb** command to evaluate Boolean expressions, such as the equality **left=right**.

```
> evalb(left = right)
```

$$false \qquad \textbf{(6.75)}$$

The negative result illustrates that a great deal of care is needed when using Maple to verify identities. Maple does not recognize the above as a true statement and therefore reports false. That does not mean that the identity is not in fact true. It only indicates that Maple was not able to confirm it.

In order to verify the symmetry identity, we need to apply two other commands. First, we use the **convert** command to replace the definitions of **left** and **right**.

```
> left := convert(left, factorial)
```

$$left := \frac{n!}{k!\,(n-k)!} \qquad \textbf{(6.76)}$$

```
> right := convert(right, factorial)
```

$$right := \frac{n!}{k!\ (n-k)!} \tag{6.77}$$

Second, we use **simplify** to help ensure that Maple will perform necessary algebraic manipulations in order for it to recognize the equality. It is often simplest to embed the call to **simplify** within the **evalb** statement. (In this example, **simplify** is not necessary, but it is typically needed.)

```
> evalb(simplify(left = right))
```

$$true \tag{6.78}$$

The Hockeystick Identity
Exercise 31 in Section 6.4 asks you to prove the hockeystick identity:

$$\sum_{k=0}^{r} \binom{n+k}{k} = \binom{n+r+1}{r}.$$

The exercise asks you to prove it using a combinatorial argument and using Pascal's identity. Maple will verify this identity for us using algebra based on the formula for the binomial coefficient.

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

```
> rightHockey := binomial(n + r + 1, r):
```

The left-hand side is a summation. Since it is a symbolic sum, we must use **sum**, rather than **add** to represent it.

```
> leftHockey := sum(binomial(n + k, k), k = 0 .. r):
```

To verify the identity, we need to go a step further and **convert** the expressions to factorials.

```
> leftHockey := convert(leftHockey, factorial)
```

$$leftHockey := \frac{(n+r+1)!}{r!\ (n+1)!} \tag{6.79}$$

```
> rightHockey := convert(rightHockey, factorial)
```

$$rightHockey := \frac{(n+r+1)!}{r!\ (n+1)!} \tag{6.80}$$

Now, **evalb** combined with **simplify** will confirm the identity.

```
> evalb(simplify(leftHockey = rightHockey))
```

$$true \tag{6.81}$$

Keep in mind when using Maple to check identities that it will only report true when the two sides of the expression are identical. 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.

If Maple reports false, however, even after using **convert** and **simplify**, you cannot be certain whether the identity is false or if it is true but more manipulation is needed to get Maple to recognize it. To use Maple to demonstrate that a purported identity is false, you would 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 Maple commands 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

```
> 26^5
```
$$11\,881\,376 \qquad \textbf{(6.82)}$$

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.

```
> 3^4
```
$$81 \qquad \textbf{(6.83)}$$

Recall from the previous section that the **numbperm** and **permute** commands can accept either a number or a list or a set as the first argument. In case you provide a list as the first argument, that list may have repeated elements. In case the list does contain repeated elements, those elements are treated as identical, but are allowed to repeat in the permutations.

For example, consider the statement below.

```
> permute([1, 1, 2])
```
$$[[1, 1, 2], [1, 2, 1], [2, 1, 1]] \qquad \textbf{(6.84)}$$

Given a list of n not necessarily distinct objects and no second argument, the **permute** command 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.

```
> permute([1, 1, 2], 2)
```
$$[[1, 1], [1, 2], [2, 1]] \qquad \textbf{(6.85)}$$

Since 1 appeared twice in the input list, it was allowed to appear twice in the results, 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 **permute** command, 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 **permute** command as shown below. Recall the use of the dollar sign operator with syntax **o $ n** to produce a sequence of the object **o** repeated **n** times.

```
> abcRepeated := permute(["a" $ 4, "b" $ 4, "c" $ 4], 4)
```

$$\begin{aligned} abcRepeated := [&["a","a","a","a"],["a","a","a","b"],\\ &["a","a","a","c"],["a","a","b","a"],["a","a","b","b"],\\ &["a","a","b","c"],["a","a","c","a"],["a","a","c","b"],\\ &["a","a","c","c"],["a","b","a","a"],["a","b","a","b"],\\ &["a","b","a","c"],["a","b","b","a"],["a","b","b","b"],\\ &["a","b","b","c"],["a","b","c","a"],["a","b","c","b"],\\ &["a","b","c","c"],["a","c","a","a"],["a","c","a","b"],\\ &["a","c","a","c"],["a","c","b","a"],["a","c","b","b"],\\ &["a","c","b","c"],["a","c","c","a"],["a","c","c","b"],\\ &["a","c","c","c"],["b","a","a","a"],["b","a","a","b"],\\ &["b","a","a","c"],["b","a","b","a"],["b","a","b","b"],\\ &["b","a","b","c"],["b","a","c","a"],["b","a","c","b"],\\ &["b","a","c","c"],["b","b","a","a"],["b","b","a","b"],\\ &["b","b","a","c"],["b","b","b","a"],["b","b","b","b"],\\ &["b","b","b","c"],["b","b","c","a"],["b","b","c","b"],\\ &["b","b","c","c"],["b","c","a","a"],["b","c","a","b"],\\ &["b","c","a","c"],["b","c","b","a"],["b","c","b","b"],\\ &["b","c","b","c"],["b","c","c","a"],["b","c","c","b"],\\ &["b","c","c","c"],["c","a","a","a"],["c","a","a","b"],\\ &["c","a","a","c"],["c","a","b","a"],["c","a","b","b"],\\ &["c","a","b","c"],["c","a","c","a"],["c","a","c","b"],\\ &["c","a","c","c"],["c","b","a","a"],["c","b","a","b"],\\ &["c","b","a","c"],["c","b","b","a"],["c","b","b","b"],\\ &["c","b","b","c"],["c","b","c","a"],["c","b","c","b"],\\ &["c","b","c","c"],["c","c","a","a"],["c","c","a","b"],\\ &["c","c","a","c"],["c","c","b","a"],["c","c","b","b"],\\ &["c","c","b","c"],["c","c","c","a"],["c","c","c","b"],\\ &["c","c","c","c"]] \end{aligned} \tag{6.86}$$

```
> nops(abcRepeated)
```

$$81 \tag{6.87}$$

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

The **numbperm** command has the same syntax as the **permute** command and will return the number of permutations without listing them all.

```
> numbperm(["a" $ 4, "b" $ 4, "c" $ 4], 4)
```

$$81 \tag{6.88}$$

Combinations with Repetition

Combinations with repetition can be handled in Maple 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 functional operator.

```
> numbcombrep := (n, r) → numbcomb(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.

```
> numbcombrep(7, 5)
```

462 **(6.89)**

We can also make use of the **numbcomb** and **choose** commands in the same way as **numbperm** and **permute** were 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 three ways to solve this problem with Maple.

First, we can use the **numbcombrep** functional operator that we created.

```
> numbcombrep(3, 4)
```

15 **(6.90)**

The second approach is to use **numbcomb**. The first argument is the list of "apple", "orange", and "pear" each repeated four times, and the second argument is 4 indicating that four objects are to be selected.

```
> numbcomb(["apple" $ 4, "orange" $ 4, "pear" $ 4], 4)
```

15 **(6.91)**

The third option is to use **choose** to list all the options. The arguments are the same as they were for **numbcomb**.

```
> choose(["apple" $ 4, "orange" $ 4, "pear" $ 4], 4)
```

[["apple", "apple", "apple", "apple"], ["apple", "apple", "apple", "orange"], ["apple", "apple", "apple", "pear"], ["apple", "apple", "orange", "orange"], ["apple", "apple", "orange", "pear"], ["apple", "apple", "pear", "pear"], ["apple", "orange", "orange", "orange"], ["apple", "orange", "orange", "pear"], ["apple", "orange", "pear", "pear"], ["apple", "pear", "pear", "pear"], ["orange", "orange", "orange", "orange"], ["orange", "orange", "orange", "pear"], ["orange", "orange", "pear", "pear"], ["orange", "pear", "pear", "pear"], ["pear", "pear", "pear", "pear"]] **(6.92)**

```
> nops(%)
```

15 **(6.93)**

Note that when repetition is allowed, all of the commands **numbperm**, **permute**, **numbcomb**, and **choose** must be given a list as the first argument. If you were to use a set instead of a list, the repeated elements in the set would be automatically removed by Maple.

Permutations with Indistinguishable Objects

Maple handles permutations with indistinguishable objects in the same way as when repetition is allowed. The **numbperm** and **permute** commands both accept lists of objects as their first argument. If objects in the list are repeated, Maple considers them as indistinguishable and counts 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 **numbperm**.

```
> numbperm(["S","U","C","C","E","S","S"])
                              420
```
(6.94)

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

Maple 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 number of strings of length 3 that can be made from the letters of the word *SUCCESS* can be calculated as follows.

```
> numbperm(["S","U","C","C","E","S","S"], 3)
                               43
```
(6.95)

Listing these words can be accomplished by use of the **permute** command with the same arguments.

```
> permute(["S","U","C","C","E","S","S"], 3)
   [["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"]]
```
(6.96)

A related question is the number of strings with three or more letters that can be made from the letters of the word *SUCCESS*. To do this, we compute the number of r-permutations with r equal to 3, 4, 5, 6, and 7 and sum these values. Using the **add** command, this can be done in one statement.

Remember that the **add** command accepts a first argument in terms of a variable such as **i**. With the second argument specifying the range of values that **i** can take, the command returns the sum of the numbers obtained by evaluating the first argument at each value for **i**.

```
> add(numbperm(["S","U","C","C","E","S","S"], i), i = 3..7)
```

$$1247 \qquad \textbf{(6.97)}$$

Distinguishable Objects and Distinguishable Boxes

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

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

```
> numbcomb(52, 5) · numbcomb(47, 5) · numbcomb(42, 5) · numbcomb(37, 5)
```

$$1\,478\,262\,843\,475\,644\,020\,034\,240 \qquad \textbf{(6.98)}$$

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

```
> 52!/(5!*5!*5!*5!*32!)
```

$$1\,478\,262\,843\,475\,644\,020\,034\,240 \qquad \textbf{(6.99)}$$

Finally, this same value can be computed using the **multinomial** command. This command requires at least two positive integers as arguments, but can accept any number of positive integers. Regardless of the number of arguments, the first must be equal to the sum of the rest. If n is the first argument and $k_1,\ k_2,\ \ldots,\ k_m$ are the rest of the arguments, then assuming $k_1 + k_2 + \cdots + k_m = n$, the command returns

$$\frac{n!}{k_1!\,k_2!\cdots k_m!}\,.$$

That is, **multinomial** implements the formula of Theorem 4. (Exercise 65 of Section 6.5 explains the reason for the term multinomial.)

Computing the number of ways to deal cards, as described above, can be computed as follows.

```
> multinomial(52, 5, 5, 5, 5, 32)
```

$$1\,478\,262\,843\,475\,644\,020\,034\,240 \qquad \textbf{(6.100)}$$

Revising the multinomial Command

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 distributed. The **multinomial** command 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." We will write a Maple command that will use the formula from Theorem 4 but will not require that we provide the size of the discard box.

In order to make our procedure work as much like **multinomial** as possible, we need it to be able to accept any number of arguments.

To do this, we apply the **seq** modifier to the parameter. As a simple example of how this works, consider the example below.

```
printInts := proc(x::seq(integer))
   print(x);
end proc:
```

The parameter declaration **x::seq(integer)** indicates that the parameter is to accept a sequence of integers. When the procedure is called with a sequence of integers as arguments, the name **x** is assigned to the sequence.

```
> printInts(5, 2, 9, 11)
```

$$5,\ 2,\ 9,\ 11 \qquad \textbf{(6.101)}$$

If nonintegers are included in the call to the procedure, the parameter will only be assigned to the sequence of integers up until the first noninteger.

```
> printInts(−5, 8, 2, "hello", 7, 11)
```

$$-5,\ 8,\ 2 \qquad \textbf{(6.102)}$$

The type in parentheses after the **seq** keyword can be any Maple type. Note that such a parameter can match the empty sequence in which case it is assigned **NULL**.

Our procedure will have two parameters. The first will be a positive integer **n**. The second will be the sequence of positive integers **k**. First, we ensure that **k** is not empty. (If **k = NULL**, the procedure returns 1, as there is 1 way to discard all the objects.) Next, we calculate the difference of the parameter **n** and the sum of the rest of the arguments. Note that **k** must be turned into a list to be used with the **add** (and most other) commands. The difference is the number of objects that are discarded. If this is negative, the procedure will raise an error. Otherwise, it becomes the final k_m in the Theorem 4 formula.

Here is the implementation. Note that we use the **map** command to compute the factorials of each of the integers in the denominator of the formula.

```
distinguishable := proc(n::posint, k::seq(posint))
   local discard, denomList, i;
   if k = NULL then
      return 1;
   end if;
   discard := n − add(i, i=[k]);
   if discard < 0 then
      error " first argument must be at least the sum of the others."
   end if;
   denomList := [k, discard];
```

```
    denomList := map(factorial,denomList);
    return n!/mul(i,i=denomList);
end proc:
```

With this procedure, we can compute the number of ways to deal five cards to each of four players from a deck of 52 cards as follows.

> *distinguishable* (52, 5, 5, 5, 5)

1 478 262 843 475 644 020 034 240 **(6.103)**

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 **numbcombrep** function written earlier.

> *numbcombrep* (8, 10)

19 448 **(6.104)**

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 **composition** and **numbcomp** commands 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. Therefore,

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

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 of n 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.

Counting Weak Compositions

We now return to **numbcomp** and **composition**. The **numbcomp** command accepts two arguments, n and r. It returns the number of r-compositions of n. For example, as we saw, there are three 2-compositions of 4.

> $numbcomp(4, 2)$

3 **(6.105)**

We can create a functional operator to count the number of weak r-compositions of n using the fact that this is the same as the number of r-compositions of $n + r$.

> $numbweakcomp := (n, r) \rightarrow combinat[numbcomp](n + r, r)$:

We saw there were five weak 2-compositions of 4.

> $numbweakcomp(4, 2)$

5 **(6.106)**

The number of ways that ten indistinguishable balls can be placed in eight bins is:

> $numbweakcomp(10, 8)$

$19\,448$ **(6.107)**

Note that weak r-compositions were the subject of Example 5, which asked how many solutions there are to $x_1 + x_2 + x_3 = 11$ with nonnegative integers.

> $numbweakcomp(11, 3)$

78 **(6.108)**

Listing Weak Compositions

The **composition** command is used to create a list of all compositions. The arguments are the same as **numbcomp**. For example, to list the 2-compositions of 4, enter the following.

> $composition(4, 2)$

$\{[1, 3], [2, 2], [3, 1]\}$ **(6.109)**

We can use **composition** to create a **weakcomposition** procedure. Given arguments n and r, we apply **composition** to $n + r$ and r. For each list in the result, we subtract 1 in each position.

The subtraction of 1 in each position can be done with the **map** command using the functional operator **x ->x - 1** as the first argument. For example,

> $map(x \rightarrow x - 1, [1, 2, 3, 4, 5, 6])$

$[0, 1, 2, 3, 4, 5]$ **(6.110)**

The functional operator is applied to each element in the list given as the second argument.

We need to apply the above to each composition in the set produced by the **composition** function. To do this, we use a for loop over the result of **composition** and build a set of weak compositions. Here is the complete procedure.

```
1 weakcomposition := proc(n::posint,r::posint)
2    local minus1, strong, weak, C;
3    minus1 := x -> x-1;
4    strong := combinat[composition](n+r,r);
```

```
    weak := {};
    for C in strong do
       weak := weak union {map(minus1,C)};
    end do;
    return weak;
end proc:
```

The weak 2-compositions of 4 can now be produced by this procedure.

> *weakcomposition* (4, 2)
>
> {[0, 4], [1, 3], [2, 2], [3, 1], [4, 0]} **(6.111)**

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 command **Stirling2** computes the Stirling number of the second kind. This command requires two arguments, the number of objects and the number of boxes. For example, the statement below computes the number of ways to put seven different employees in four offices when each office must not be empty.

> *Stirling2* (7, 4)
>
> 350 **(6.112)**

In order to compute the number of ways to assign the seven employees to the four offices and allow empty offices, we must add the number of ways to assign all seven employees to one office, to two offices, to three offices, and to four offices.

> *add* (*Stirling2* (7, k), $k = 1..4$)
>
> 715 **(6.113)**

Generating Assignments of Employees to Offices

The **Stirling2** command tells us how many ways there are to place distinguishable objects in indistinguishable boxes, but Maple does not have a command to produce the assignments.

To make such a command, 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, $\{\{D\}, \{A, C\}, \{B, E\}\}$ 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:

$$\{B_1 \cup \{n\}, B_2, \ldots, B_k\},\ \{B_1, B_2 \cup \{n\}, B_3, \ldots, B_k\},\ \ldots,\ \{B_1, B_2, \ldots, B_k \cup \{n\}\}.$$

The assignments produced by the two methods above produce all assignments. The following procedure implements this algorithm.

```
makeStirling2 := proc(n::posint, k::posint)
   local A, k1boxes, kboxes, B, new, i;
   if k = 1 then
      return {{{$1..n}}};
   end if;
   if k > n then
      return {};
   end if;
   A := {};
   # n-1 objects in k-1 boxes
   k1boxes := makeStirling2(n-1,k-1);
   for B in k1boxes do
      new := B union {{n}};
      A := A union {new};
   end do;
   # n-1 objects in k boxes
   kboxes := makeStirling2(n-1,k);
   for B in kboxes do
      for i from 1 to k do
         new := subsop(i=B[i] union {n},B);
         A := A union {new};
      end do;
   end do;
   return A;
end proc:
```

Let us analyze the procedure. It accepts n and k as parameters and returns the set consisting of all possible assignments of distinguishable objects to indistinguishable boxes.

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 procedure returns the set consisting of this single assignment when $k = 1$. Recall the **$** operator with no left argument and a range as its right argument produces the sequence defined by the range.

If $k > n$, then there are no valid assignments and the procedure returns the empty set.

Otherwise, we initialize **A**, which will store the set of assignments that is returned, 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 name **k1boxes** to the set of assignments of $n-1$ objects to $k-1$ boxes. For each such assignment, for example, $\{\{2\},\{4,6\},\{1,3,5\}\}$, we add n in its own box:

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

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, 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:

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

Recall that the **subsop** command is used to replace (substitute) elements in sets and lists. The syntax **subsop(i=x,S)** where **i** is an index of the set or list **S**, causes the expression **x** to replace whatever had been at index **i** in **S**.

Compare the result of our procedure to the solution of Example 10 in Section 6.5 of the text.

```
> makeStirling2(4, 3)
    {{{1},{2},{3,4}},{{1},{3},{2,4}},{{1},{4},{2,3}},
      {{2},{3},{1,4}},{{2},{4},{1,3}},{{3},{4},{1,2}}}                (6.114)
```

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 three offices.

To produce all 14 ways to assign the four employees to three offices with each office containing any number of employees, we need to loop over the different values of k.

```
> offices := { }:
  for k from 1 to 3 do
    offices := offices union makeStirling2(4, k):
  end do:
```

We print them out one at a time.

```
> for o in offices do
    print(o)
  end do
    {{1,2,3,4}}
    {{1},{2,3,4}}
    {{2},{1,3,4}}
```

$$\{\{3\},\{1,2,4\}\}$$
$$\{\{4\},\{1,2,3\}\}$$
$$\{\{1,2\},\{3,4\}\}$$
$$\{\{1,3\},\{2,4\}\}$$
$$\{\{1,4\},\{2,3\}\}$$
$$\{\{1\},\{2\},\{3,4\}\}$$
$$\{\{1\},\{3\},\{2,4\}\}$$
$$\{\{1\},\{4\},\{2,3\}\}$$
$$\{\{2\},\{3\},\{1,4\}\}$$
$$\{\{2\},\{4\},\{1,3\}\}$$
$$\{\{3\},\{4\},\{1,2\}\} \quad \textbf{(6.115)}$$

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 j positive integers is a sum $n = a_1 + a_2 + \cdots + a_j$ with $0 < a_1 \le a_2 \le \cdots \le a_j$. (Note that this is the reverse order from the definition in the text. Maple's commands for producing partitions list them in nondecreasing order, rather than in nonincreasing order as is done in the text.)

The Maple commands **numbpart** and **partition** are used to count and form partitions of integers.

With one argument, a nonnegative integer n, **numbpart** returns the total number of partitions of n into as many as n boxes. Likewise, **partition** applied to one argument returns a list containing lists representing partitions of the argument.

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

```
> numbpart(7)
```
$$15 \quad \textbf{(6.116)}$$

```
> partition(7)
```
$$[[1,1,1,1,1,1,1],[1,1,1,1,1,2],[1,1,1,2,2],[1,2,2,2],[1,1,1,1,3],$$
$$[1,1,2,3],[2,2,3],[1,3,3],[1,1,1,4],[1,2,4],[3,4],[1,1,5],$$
$$[2,5],[1,6],[7]] \quad \textbf{(6.117)}$$

Both commands also accept a second argument, which specifies the maximum integer allowed to appear in a partition. For example, to answer the question: how many ways are there to distribute 7 indistinguishable balls in up to 7 identical boxes when each box can hold at most 4 objects, we would give 4 as the second argument to **numbpart**.

```
> numbpart(7,4)
```
$$11 \quad \textbf{(6.118)}$$

To list them, we give the same arguments with **partition**.

```
> partition(7,4)
```
$$[[1,1,1,1,1,1,1],[1,1,1,1,1,2],[1,1,1,2,2],[1,2,2,2],[1,1,1,1,3],$$
$$[1,1,2,3],[2,2,3],[1,3,3],[1,1,1,4],[1,2,4],[3,4]] \quad \textbf{(6.119)}$$

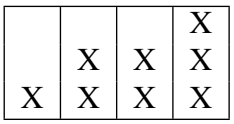
We will now see how to determine the number of ways to partition n into at most k boxes.

Limiting the Number of Boxes

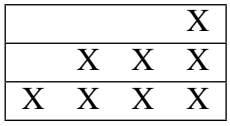
Example 11 asks how many ways there are to pack six copies of the same book into four boxes. The way that **numbpart** and **partition** are described, with the second argument controlling the maximum value in the partition rather than the number of entries in the list, suggests that they do not answer this question. However, as it turns out, they do.

In fact, 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 $[1, 2, 2, 3]$.

Think about stacking the 8 objects in boxes according to the partition $[1, 2, 2, 3]$. (The diagram shown below is called a Ferrers diagram.)



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



Now, the 8 objects are contained in three boxes. One box (the top row) has 1 object, another (the middle row) has 3 objects, and the last box (the bottom row) has 4 objects. In other words, we have partitioned 8 as $[1, 3, 4]$. These two partitions are said to be conjugate. (Note that you can also think about forming the conjugate by rotating the original diagram by 90 degrees or reflecting it across its diagonal while still interpreting the boxes as columns.)

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

The consequence of this discussion is that the number of ways to pack 6 copies of the same book into at most 4 identical boxes that can each hold all the books is the same as the number of ways to pack 6 copies of the same book into boxes that can hold at most 4 books each. That is, the solution to Example 11 is given by the following computation.

```
> numbpart(6, 4)
                    9                                    (6.120)
```

To produce these 9 partitions, we will use the **conjpart** command. This command accepts a partition as its only argument. It returns the conjugate of that partition. Using the example $[1, 2, 2, 3]$ from above,

```
> conjpart([1, 2, 2, 3])
                 [1, 3, 4]                               (6.121)
```

The statement **partition(6,4)** will produce the list of all partitions of 6 with maximum value 4. Applying **conjpart** to each of those partitions produces the list of all partitions of 6 into at most 4 boxes. We use **map** with **conjpart** as the first argument and **partition(6,4)** as the second argument to apply **conjpart** to each partition.

> *map* (*conjpart*, *partition* (6, 4))

$$[[6], [1, 5], [2, 4], [3, 3], [1, 1, 4], [1, 2, 3], [2, 2, 2], [1, 1, 1, 3], [1, 1, 2, 2]] \quad \textbf{(6.122)}$$

Generating Partitions
Here, we will describe how to generate partitions.

The first observation to make is that the partitions of n are identical to the partitions of n when each box can hold at most n objects. In terms of the **partition** command, the following are identical.

> *partition* (7)

$$[[1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 2], [1, 1, 1, 2, 2], [1, 2, 2, 2], [1, 1, 1, 1, 3], [1, 1, 2, 3], [2, 2, 3], [1, 3, 3], [1, 1, 1, 4], [1, 2, 4], [3, 4], [1, 1, 5], [2, 5], [1, 6], [7]] \quad \textbf{(6.123)}$$

> *partition* (7, 7)

$$[[1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 2], [1, 1, 1, 2, 2], [1, 2, 2, 2], [1, 1, 1, 1, 3], [1, 1, 2, 3], [2, 2, 3], [1, 3, 3], [1, 1, 1, 4], [1, 2, 4], [3, 4], [1, 1, 5], [2, 5], [1, 6], [7]] \quad \textbf{(6.124)}$$

We will describe how to form the partitions of n objects when each box can hold at most k objects recursively. The basis cases are: when $k = 1$, there is only one partition of n, the partition consisting of n 1s; when $n \leq 0$, there are no partitions.

To determine the partitions of n when each box can hold at most k objects, 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.

Second, provided that $n - k > 0$, determine all partitions of $n - k$ when each box can hold at most k objects, and append 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:

> *partition* (4, 3)

$$[[1, 1, 1, 1], [1, 1, 2], [2, 2], [1, 3]] \quad \textbf{(6.125)}$$

Therefore, $[1, 1, 1, 1, 3]$, $[1, 1, 2, 3]$, $[2, 2, 3]$, $[1, 3, 3]$ are 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.

Maple Commands for Generating Partitions Sequentially

We have seen that the **partition** command will produce all partitions of a given integer. Maple also has commands for producing partitions sequentially. Like **subsets** and **cartprod** for sequentially computing the subsets of a set and the Cartesian product of sets, producing partitions sequentially rather than all at once can save time and memory.

Maple's procedures are based on a canonical ordering of partitions. This ordering begins with the partition of all 1s and ends with the partition $[n]$. In this ordering, a partition $[x_k, x_{k-1}, .., x_2, x_1]$ precedes the partition $[y_j, y_{j-1}, .., y_2, y_1]$ if, for an index i,

$$x_1 = y_1\ , x_2 = y_2, ..., x_{i-1} = y_{i-1}, \text{and } x_i < y_i.$$

In other words, partition x precedes partition y when the first (from the right) position at which they differ contains a smaller value in x than in y.

For example, the partition $[1, 1, 1, 3, 5]$ precedes the partition $[1, 2, 3, 5]$ because they agree in the two right-most positions, but in the third position from the right, the entry in $[1, 1, 1, 3, 5]$ is smaller than the third value from the right in $[1, 2, 3, 5]$. This is called reverse lexicographic order.

The commands **firstpart** and **lastpart** accept a positive integer n as their only argument. They return the partition consisting of n 1s and the partition $[n]$, respectively.

```
> firstpart(11)
```
$$[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] \qquad \textbf{(6.126)}$$

```
> lastpart(11)
```
$$[11] \qquad \textbf{(6.127)}$$

The command **nextpart** accepts as its sole argument a partition, that is, a nondecreasing list of positive integers. It returns the next partition in the ordering.

```
> nextpart([1, 1, 1, 2, 2, 4])
```
$$[1, 2, 2, 2, 4] \qquad \textbf{(6.128)}$$

Likewise, **prevpart** accepts a partition and returns the previous partition in the ordering.

```
> prevpart(%)
```
$$[1, 1, 1, 2, 2, 4] \qquad \textbf{(6.129)}$$

Using these commands, we can easily generate all partitions of a given number one at a time. We initialize a name with the **firstpart** command. Inside a while loop controlled by testing the current partition against the last partition, we update the partition using the **nextpart** command.

```
> p := firstpart(5)
```
$$p := [1, 1, 1, 1, 1] \qquad \textbf{(6.130)}$$

```
> while p ≠ lastpart(5) do
    p := nextpart(p)
  end do
```

$p := [1,1,1,2]$
$p := [1,2,2]$
$p := [1,1,3]$
$p := [2,3]$
$p := [1,4]$
$p := [5]$ **(6.131)**

Maple also includes the **randpart** command for generating a random partition. This command accepts only one argument, the integer to be partitioned.

```
> randpart(5)
```
$[1,4]$ **(6.132)**

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 procedure to interchange two elements in a list. This will be called by the procedure for generating permutations.

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

The procedure has five statements. A copy of the list is made, since parameters cannot be assigned to. 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.

```
Interchange := proc(L::list, i::integer, j::integer)
   local l, temp;
   l := L;
   temp := l[i];
   l[i] := l[j];
   l[j] := temp;
   return l;
end proc:
```

```
> Interchange(["a","b","c","d","e","d"],2,5)
```
$["a","e","c","d","b","d"]$ **(6.133)**

nextpermutation

The input to the procedure 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, through the 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 while loop decreases the value of j by one, and it is controlled by the condition $a_{j+1} < a_j$. 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 $a_k < a_j$. When the while loop terminates, k will be such that $a_j < a_k$. Note that the loop is guaranteed to stop with $j < k$ since $a_j < a_{j+1}$. Once j has been identified, we interchange a_j and a_k using the **Interchange** procedure.

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 that follow 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 procedure two tests to ensure that the input is valid. We will declare the input to be a list of positive integers. Within the body of the procedure, we will check first to ensure that all of the elements in the list are no greater than n, the length of the list. Otherwise, the procedure will generate an error. We will also compare the input to the final permutation $[n, n-1, .., 1]$ and return **FAIL** if they are equal.

Here is the implementation.

```
nextpermutation := proc(A::list(posint))
   local a, n, i, j, k, r, s;
   a := A;
   n := nops(a);
   for i from 1 to n do
      if a[i] > n then
         error "Input must be a permutation of {1,2,...,n}.";
      end if;
```

```
    end do;
    if a = [seq(n-i,i=0..(n-1))] then
       return FAIL;
    end if;
    # step 1: find j
    j := n - 1;
    while a[j] > a[j+1] do
       j := j - 1;
    end do;
    # step 2: find k and interchange aj and ak
    k := n;
    while a[j] > a[k] do
       k := k - 1;
    end do;
    a := Interchange(a,j,k);
    # step 3: sort the tail
    r := n;
    s := j + 1;
    while r > s do
       a := Interchange(a,r,s);
       r := r - 1;
       s := s + 1;
    end do;
    return a;
end proc:
```

Example 2 of Section 6.6 finds that the permutation after 362 541 is 364 125. We use that example to confirm that our procedure is working.

```
> nextpermutation([3,6,2,5,4,1])
```

$$[3,6,4,1,2,5] \tag{6.134}$$

To generate all permutations of a set $\{1,2,\ldots,n\}$, we use a while loop.

```
> aperm := [$1..4]:
  while aperm ≠ FAIL do
    print(aperm);
    aperm := nextpermutation(aperm)
  end do:
```

$$[1,2,3,4]$$
$$[1,2,4,3]$$
$$[1,3,2,4]$$
$$[1,3,4,2]$$
$$[1,4,2,3]$$
$$[1,4,3,2]$$
$$[2,1,3,4]$$
$$[2,1,4,3]$$

$$
\begin{array}{l}
[2,3,1,4]\\
[2,3,4,1]\\
[2,4,1,3]\\
[2,4,3,1]\\
[3,1,2,4]\\
[3,1,4,2]\\
[3,2,1,4]\\
[3,2,4,1]\\
[3,4,1,2]\\
[3,4,2,1]\\
[4,1,2,3]\\
[4,1,3,2]\\
[4,2,1,3]\\
[4,2,3,1]\\
[4,3,1,2]\\
[4,3,2,1]
\end{array}
\qquad \textbf{(6.135)}
$$

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 Maple, we can use the **subs** command to transform a permutation of $\{1, 2, \ldots, n\}$ into a permutation of another set with n elements.

The **subs** command is used to perform substitutions in an expression. It has several forms, but we will use it with two arguments. The second argument will be the permutation of $\{1, 2, \ldots, n\}$. The first argument will be a list of equations of the form $i = x$ where i is an integer in $\{1, 2, \ldots, n\}$ and x is the corresponding element of the set being permuted.

For example, consider the set $\{a, b, c\}$ and the permutation $[3, 1, 2]$ of $\{1, 2, 3\}$. We establish the correspondence that 1, 2, and 3 are identified with a, b, and c, respectively. To produce the corresponding partition, we use the **subs** command with first argument $[1 = a, 2 = b, 3 = c]$ to specify the correspondence. The second argument is the permutation $[3, 1, 2]$.

```
> subs([1 = "a", 2 = "b", 3 = "c"], [3, 1, 2])
```

$$
[\text{“}c\text{”}, \text{“}a\text{”}, \text{“}b\text{”}] \qquad \textbf{(6.136)}
$$

As another example, consider the set $\{2, 10, 13, 19\}$ and the permutation $[4, 2, 3, 1]$. Identify 1 with 2, 2 with 10, 3 with 13, and 4 with 19. Then, the following command produces the desired permutation.

```
> subs([1 = 2, 2 = 10, 3 = 13, 4 = 19], [4, 2, 3, 1])
```

$$
[19, 10, 13, 2] \qquad \textbf{(6.137)}
$$

Note that **subs** will allow you to provide the equations without enclosing them in a list. However, putting them in a list ensures that the substitutions are done simultaneously rather than sequentially. In this example, sequential assignment would not have produced the correct result.

A Procedure to Apply Permutations

We conclude by writing a procedure that, given a set and a permutation of $\{1, 2, \ldots, n\}$ with n equal to the size of the set, will return the corresponding permutation of the set.

The key is automating the creation of the equations defining the correspondence. This can be done with the **zip** command. Recall that **zip** requires three arguments. The second and third arguments are lists while the first argument is a procedure on two parameters. The result is the list formed by applying the procedure to corresponding pairs of elements from the two lists.

We create a functional operator that takes two expressions and forms an equation from them.

```
> makeEqn := (a,b) -> a=b
```

$$makeEqn := (a,b) \mapsto a = b \tag{6.138}$$

```
> makeEqn(5,a)
```

$$5 = a \tag{6.139}$$

Using the list $[1, 2, 3]$ and the list formed from the elements of the set $\{a, b, c\}$ as the second and third arguments, **zip** will produce the list of equations used in **(6.136)**.

```
> zip(makeEqn,[1,2,3],["a","b","c"])
```

$$[1 = \text{“}a\text{”}, 2 = \text{“}b\text{”}, 3 = \text{“}c\text{”}] \tag{6.140}$$

This is all we need to write a procedure that accepts a set and a permutation of $\{1, 2, \ldots, n\}$ and returns the corresponding permutation of the elements of the set.

```
permuteSet := proc(S::set,P::list(posint))
   local L, M, makeeqn, eqns;
   L := [op(S)];
   M := [$1..nops(S)];
   makeeqn := (a,b) -> a=b;
   eqns := zip(makeeqn,M,L);
   return subs(eqns,P);
end proc:
```

Using the procedure, we can compute the permutation of $\{a, b, c, d, e\}$ corresponding to $[3, 5, 2, 1, 4]$.

```
> permuteSet({"a","b","c","d","e"},[3,5,2,1,4])
```

$$[\text{“}c\text{”}, \text{“}e\text{”}, \text{“}b\text{”}, \text{“}a\text{”}, \text{“}d\text{”}] \tag{6.141}$$

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 **choose** command could be used to generate combinations with repetition by repeating the elements in the list given as the first argument to **choose**.

To generate the 2-combinations of $\{1, 2, 3\}$, for example, we apply the **choose** command to the list consisting of 1, 2, and 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.

```
> choose([1$2, 2$2, 3$2], 2)
```

$$[[1, 1], [1, 2], [1, 3], [2, 2], [2, 3], [3, 3]] \qquad \textbf{(6.142)}$$

Recall that **x$n** generates the sequence of n x's.

The first argument to **choose** can be generated using **seq** as follows.

```
> [seq(i$2, i = 1..3)]
```

$$[1, 1, 2, 2, 3, 3] \qquad \textbf{(6.143)}$$

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

```
chooseRepetition := proc(n::posint, r::posint)
   local L, i;
   L := [seq(i$r, i=1..n)];
   return combinat[choose](L,r);
end proc:
```

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

```
> chooseRepetition(5, 3)
```

$$[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 1, 4], [1, 1, 5], [1, 2, 2], [1, 2, 3], [1, 2, 4], [1, 2, 5],$$
$$[1, 3, 3], [1, 3, 4], [1, 3, 5], [1, 4, 4], [1, 4, 5], [1, 5, 5], [2, 2, 2], [2, 2, 3], [2, 2, 4],$$
$$[2, 2, 5], [2, 3, 3], [2, 3, 4], [2, 3, 5], [2, 4, 4], [2, 4, 5], [2, 5, 5], [3, 3, 3], [3, 3, 4],$$
$$[3, 3, 5], [3, 4, 4], [3, 4, 5], [3, 5, 5], [4, 4, 4], [4, 4, 5], [4, 5, 5], [5, 5, 5]] \qquad \textbf{(6.144)}$$

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 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 procedure 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 need to be able to 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. To do this, we will use

the **Collect** command from the **ListTools** package. When applied to a list, **Collect** returns a list of two-element lists. Each sublist contains a unique element of the input list followed by the number of times that object appeared in the input. For example,

```
> ListTools[Collect](["a","b","b","c","c","c","d","d","d","d"])
    [["d",4],["a",1],["b",2],["c",3]]                                    (6.145)
```

We will use this function to create a small procedure to determine whether or not the playoff tournament is over. The procedure will return true if one of the teams has won or false if neither team has reached the threshold for winning. Note that since we do not care which team won, we will take the output of **Collect** and immediately extract the count value by using **map** with a function that extracts the second member of each sublist. (An alternative approach to extracting the second elements of a list of sublists is illustrated in the solution to Computations and Explorations 3.)

```
> map(L -> L[2], (6.145))
    [4,1,2,3]                                                            (6.146)
```

We conclude the procedure by using **max** to compare the number of wins of whichever team has won more games with the threshold for winning the playoff.

```
playoffWon := proc(L::list({1,2}),n::posint)
   local collected;
   collected := ListTools[Collect](L);
   evalb(max(map(L->L[2],collected))>=n);
end proc:
```

For instance, in [1, 2, 2, 1, 1, 1, 2, 1], the procedure recognizes that the playoff has been won.

```
> playoffWon([1,2,2,1,1,1,2,1],5)
    true                                                                 (6.147)
```

With this helper procedure in place, we are ready to construct all possible outcomes. Begin with a set **outcomes** and a list **S**. Initialize **outcomes** to the empty set 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** set, 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 procedure.

```
allPlayoffs := proc(n::posint)
   local outcomes, S, p, p1, p2;
   outcomes := {};
   S := [[1],[2]];
   while S <> [] do
      p := S[1];
      S := S[2..-1];
      p1 := [op(p),1];
      p2 := [op(p),2];
```

```
10        if playoffWon(p1,n) then
11          outcomes := outcomes union {p1};
12        else
13          S := [op(S),p1];
14        end if;
15        if playoffWon(p2,n) then
16          outcomes := outcomes union {p2};
17        else
18          S := [op(S),p2];
19        end if;
20      end do;
21      return outcomes;
22  end proc:
```

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

> *best3of5* := *allPlayoffs*(3)

best3of5 := {[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]} **(6.148)**

> *nops*(*best3of5*)

20 **(6.149)**

The reader is left to apply the procedure 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 Erdős and Ron Graham.]

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

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

> c := binomial(6, 3)

$c := 20$ **(6.150)**

To determine whether or not $C(2n, n)$ is divisible by the square of a prime, we 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 use the command **ifactors** (first discussed in Section 4.3 of this manual). The **ifactors** command requires one argument, an integer. Its output is a list of the form $[sign, [[p_1, e_1], [p_2, e_2], \ldots, [p_m, e_m]]]$

where *sign* is positive or negative 1; p_1, p_2, …, p_m are the primes in the prime factorization; and e_1, e_2, …, e_m are the corresponding exponents.

Apply **ifactors** to $C(6, 3)$.

```
> ifactors(c)
    [1, [[2, 2], [5, 1]]]                                    (6.151)
```

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

We are interested in the exponents of the primes. We take the second element of the list to obtain the list of pairs of primes and their exponents without the sign component.

```
> cFacts := ifactors(c)[2]
    cFacts := [[2, 2], [5, 1]]                               (6.152)
```

This gives us a list of lists, where the first element in each internal list is the prime factor and the second element in each pair is the exponent associated to that prime. Since we are only interested in the exponents, we separate them out. (An alternative approach to extracting the second elements of a list of sublists is illustrated in the solution to Computations and Explorations 1.)

```
> cPowers := seq(x[2], x = cFacts)
    cPowers := 2, 1                                          (6.153)
```

Provided that there is an exponent of 2 or larger, we know that the square of a prime divides the number. In this case, the first exponent we obtained is 2, so in fact $C(6, 3)$ is divisible by the square of a prime.

Combining these steps, we write a procedure, **HasPrimeSquare**, that, given n, returns true if $C(2n, n)$ is divisible by the square of a prime and false if not.

```
HasPrimeSquare := proc(n::posint)
   local c, facts, powers, x, e;
   c := binomial(2*n,n);
   facts := ifactors(c)[2];
   powers := seq(x[2],x=facts);
   for e in powers do
      if e >= 2 then
         return true;
      end if;
   end do;
   return false;
end proc:
```

As in the solution of Computations and Explorations 2 in Chapter 5, we will use **timelimit** to test as many values of n as we can in a given amount of time. The **timelimit** command accepts an amount of time as its first argument and a procedure call as its second. If the execution of the command takes longer than the amount of time allowed, a "time expired" error will be raised. Since we will

be using **timelimit** to control an otherwise infinite loop, we are expecting the error and so will use a try...catch block to catch the time limit error.

First, we need to write the procedure that will execute the infinite loop. Note that the name storing the integer being checked is made global so that it can be printed once time has expired.

```
CheckPrimeSquares := proc()
   global n;
   for n from 1 do
      if not HasPrimeSquare(n) then
         print(cat("Found counterexample: ",n));
      end if;
   end do;
end proc:
```

Now we execute this procedure for 1 second and verify that the only small values of n for which $C(2n, n)$ is not divisible by the square of a prime are 1, 2, and 4, as expected.

```
> try
    timelimit(1, CheckPrimeSquares( ))
  catch "time expired" :
    print(cat("Checked through n = ", n − 1));
  end try;
    "Found counterexample: 1"
    "Found counterexample: 2"
    "Found counterexample: 4"
    "Checked through n = 442"                                    (6.154)
```

Exercises

Exercise 1. Build a recursive version of **SubsetSumCount**, using the ideas of **FindBitStrings**. Your procedure 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**.

Exercise 2. Create a procedure **FindDecreasing** by modifying **FindIncreasing** in order to determine a strictly decreasing subsequence of maximal length.

Exercise 3. Modify the Patience algorithm to find *all* of the strictly increasing subsequences of maximal length.

Exercise 4. Use Maple to find an example demonstrating that n positive integers not exceeding $2n$ are not sufficient to guarantee that one integer divides one of the others (see Example 11 of Section 6.2).

Exercise 5. The functions for comparing and generating circular permutations, **CPEquals** and **allCP**, are inefficient. Using the idea that explains the formula for the number of circular r-permutations of n people, write more efficient procedures.

Exercise 6. Use Maple 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.

Exercise 7. Suppose that a certain Department of Mathematics and Statistics has m mathematics faculty and s statistics faculty. Write a Maple procedure to find all committees with $2\,k$ members in which both mathematicians and statisticians are represented equally.

Exercise 8. Use Maple to prove the identity

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

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

Exercise 9. Use Maple to prove Pascal's identity:

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

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

Exercise 10. Use Maple to generate many rows of Pascal's triangle. See if you can formulate any conjectures involving identities satisfied by the binomial coefficients. Use Maple to help you verify that your conjecture is true by using the techniques at the end of Section 6.4 of this manual.

Exercise 11. Write a procedure that mixes the techniques used in **BinomialF** and **BinomialR** to generate the rows of Pascal's triangle from row a to row b for $b > a > 0$.

Exercise 12. Use Maple 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 nonnegative integers. Also count and list all solutions such that $x_1 \geq 1$,$x_2 \geq 2$, $x_3 \geq 3$, and $x_4 \geq 4$.

Exercise 13. Generate a large triangle of Stirling numbers of the second kind and look for patterns that suggest identities among the Stirling numbers. In addition, see if you can make any conjectures about the relationship between Stirling numbers and the binomial coefficients.

Exercise 14. Implement the algorithm described in the "Generating Partitions" subsection of Section 6.5 of this manual.

Exercise 15. Write a procedure that generates all possible schedules for airplane pilots who must fly d days in a month with m days with the restrictions that they cannot work on consecutive days (see Exercise 22 in Section 6.5).

Exercise 16. Write a Maple procedure 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.

Exercise 17. Implement Algorithm 2 of Section 6.6 for generating the next largest bit string.

Exercise 18. Implement Algorithm 3 of Section 6.6 for generating the next r-combination.

Exercise 19. Write a Maple procedure to compute the Cantor expansion of an integer. (See the preamble to Exercise 14 of Section 6.6 of the text.)

Exercise 20. 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.