East Side, West Side . . .

an introduction to combinatorial families–with Maple programming

Herbert S. Wilf

University of Pennsylvania
Philadelphia, PA, USA

August 14, 1999

Contents

1 Introduction

1.1 What this is about

This material is intended for a course that will combine a study of combinatorial structures with introductory recursive programming in Maple.

Maple is a system for doing mathematics on a computer. It is widely available in colleges and universities. Most often Maple is used interactively. That is, you ask it a question, like

```
> 2+2;
```

and it immediately gives you the answer, in this case

```
4
```

But programming is a different kettle of fish altogether. In programming you ask the computer to carry out a sequence of instructions (*program*) that you have written, and the computer then retires to its cave and does so. It will report back to you when it has finished, but in the meantime it might be doing millions of things that you asked it to do, while you will have been eating chocolates and reading a novel.

So programming is a very nontrivial way to relate to a computer. But *recursive* programming raises the stakes once again. A recursive program is one that *calls itself* in order to get the job done. Something like looking up the word *horse* in a dictionary, and finding that it means *horse*, and actually learning something from that exchange. Many high level computer languages are not capable of dealing with recursive programs, and none of them would be suitable for use in a course such as this one. Among languages that do permit recursive programming, one might ask, "Why choose Maple? Why not C^{++}?"

Indeed, Maple is not nearly as elegant a programming language as C^{++}, and a bunch of others that we might mention. In fact Maple is fairly creaky in a number of respects. But there one thing you'll have to admit: Maple is a brilliant mathematician. It not only knows how to answer your questions interactively, and it not only provides a fully equipped, if cranky, computer programming language; it also knows an absolutely startling quantity of higher mathematics. C^{++} doesn't know any of that. For instance, some of the tasks for which we'll write programs below are already built-in to the Maple language.

So for use in a computing environment that will contain a lot of high grade mathematics, a language like Maple is very desirable. There are other languages with similar capabilities, like Mathematica, for instance, but these are not nearly so widely available to students in institutions of learning.

That's why Maple.

But why recursive? Anything that can be programmed recursively can also be programmed nonrecursively. In fact, nonrecursive programs often run much faster, and are often more efficient, than recursive programs. But this is not primarily a course about writing fast and efficient programs. It is about concepts. If a mathematical object has an intrinsically recursive structure, then we will respect that structure by writing a recursive program to build it. In such cases, the recursive programs will often have great elegance and will faithfully mirror the evolution of the structures, without letting it get hidden in a sea of bookkeeping and accounting.

Recursion is very much underrepresented in mathematics curricula these days. One good way to get the subject out of hiding is by coupling it with a mathematical discussion of some structures that are inherently recursive, as are the combinatorial families in this work.

Finally, an excellent book on the nuts and bolts of Maple programming [PG] is now available. We recommend the use of that work in conjunction with this course.

These notes express a point of view that emerged while Albert Nijenhuis and I were writing *Combinatorial Algorithms* [NW], and specifically while we were writing the second edition of same. Indeed, if you are familiar with that volume then you will recognize that these notes cover much of the same ground as that work, adjusted for changes in programming languages and fashions, and reflecting my current predilection for using recursion whenever it is natural to do so. To that collaboration with Albert Nijenhuis I owe most of what I know about this subject.

2 About programming in Maple

These lecture notes are not primarily about Maple programming. They are intended to be used together with a good exposition of the Maple programming language. Since there aren't many good books about that subject (one is [PG]), we will say a few words here about Maple, and the programs will say a few more words.

The heart of writing a Maple program is in writing a Maple *procedure*. A procedure is like a little box with a certain number of input quantities and some output quantities. It is intended to be entirely self-contained so that other programs can freely use it. A Maple procedure begins with lines like

```
findit:=proc(x,y,z,...)
local var1, var2, .. ;
```

Here `findit` is the name of the procedure. The input and output quantities are `x,y,z,...`. The variables `var1, var2, ...` are declared to be `local` variables. Local variables are variables that are used inside your procedure `findit`, and by declaring them to be local you are assuring that their names will not conflict with other names that are used in the programs that use your procedure `findit`. For instance, lots of programs use a variable named `i` as a summation variable. If you declare `i` to be local to your procedure then even though a variable named `i` might also appear in the main program that is using your procedure, their will be no conflict between their names. Any change that is made to the external variable `i` will not affect the one that is internal to procedure `findit` and conversely. Generally speaking, it is good programming practice to declare every variable that occurs inside your procedure to be local, except for the input and output variables `x,y,z,...`, so they will not affect other parts of the program.

Here is a Maple procedure `addthem` whose mission in life is to add up the members of a given list and return the sum.

```
addthem:=proc(x)
local i;
RETURN(add(x[i],i=1..nops(x)));
end:
```

In this procedure, `x` is the input list, and the variable `i` is declared to be local. In that way, the external program that uses this procedure can also name variables as `i`, if it wishes to, and they will not conflict with the ones that are local to this procedure `findit`. To use this routine to add up the numbers in the list `[3,1,8,19]`, just type
`> addthem([3,1,8,19]);`
in a Maple worksheet, for instance, and Maple will reply

```
31
```

The next one is a *recursive* Maple procedure that will calculate $n!$.

```
fact:=proc(n):
# Computes n factorial
if n=0 then RETURN(1) else RETURN(n*fact(n-1)) fi:
end:
```

Recursive program have a certain spare beauty. Notice that this one gets its answer by calling itself with a lower value of n, and then multiplying by n.

Here is a recursive program that will calculate the gcd of two given positive integers m and n. It expresses the mathematical fact that the gcd of m and n is the same as the gcd of n and m mod n. Thus $gcd(18, 14) = gcd(14, 4) = gcd(4, 2) = gcd(2, 0) = 2$. The Maple program is the following.

```
gcdiv:=proc(m,n):
# Finds the gcd of two given integers
if n=0 then RETURN(m) else RETURN(gcdiv(n,m mod n)) fi:
end:
```

A list, in Maple, consists of a left bracket, a number of items separated by commas, and a right bracket. Thus, `yy:=[a,g,379,ww,sam,94]` is a Maple list. If `tt` is a list, then its successive entries are `tt[1], tt[2], ...`, so that `yy[3]` is 379, for instance. The number of items in a list `tt` is `nops(tt)`. For example, `nops(yy)` is 6.

A few more rarefied Maple instructions that will be of much use to us here are the following.

1. The `op` command. If you give to the `op` command a bracketed list, such as `[2,1,5]`, it will strip off the outermost brackets, leaving the bare sequence. Thus `op([9,3,17])` gives `9,3,17`.

2. We can augment a list by using the `op` command. Suppose we have the list `xx:=[1,2,3,4];` and we want to make it into the list `[1,2,3,4,8]`. This can be done by executing the command `y:=[op(xx),8]`, in which the inner `op` strips off the brackets, then we adjoin the `8`, and then we enclose the result in a new pair of brackets to achieve the desired effect.

3. The `map` command is a powerful way to distribute the effect of a function throughout a list. If we have a function, say the function

   `>f:=x->x^2;`
   and we give the command

   `>map(f,[1,2,3,4]);` then the result will be the list

   `[1,4,9,16]`

   in which the function has been applied to every element of the input list.

4. Another command of this kind is `applyop`. This is similar to `map`, except that it applies the mapping only to one chosen part of the object list instead of to all. More precisely, the instruction `applyop(f,j,zz);` will apply the mapping `f` only to the jth element of `zz`. For example,

   ```
   >applyop(x->x^2,2,[1,2,3,4]);
   ```

   ```
   [1,4,3,4]
   ```

   and since `applyop(u->u^2,2,t)` will square the 2nd entry of a given list `t`, we can `map` it too, with results like this:

   ```
   >map(t->applyop(u->u^2,2,t),[[1,2,3],[1,4,5],[4,4,6]]);
   ```

   ```
   [[1,4,3],[1,16,5],[4,16,6]]
   ```

2.1 Exercises

Write the following programs as self contained Maple procedures. In each case debug the program, run it on the sample problem, and hand in the program itself and the printed output.

1. Write procedure `sumcoeff` whose input is a polynomial in x, as an expression, and whose output is the sum of the absolute values of the coefficients of all of the powers of x in that polynomial. Run `sumcoeff(3*x^2+5*x-7)`, for which your program should return the value 15.

2. Write a procedure `sumsqdiv` whose input is a positive integer n and whose output is the sum of the squares of the divisors of n including 1 and n itself. Test your program by running `sumsqdiv(12)` and checking the answer.

3. Write `revlist`, a Maple procedure whose input is a list and whose output is the same list in reverse order. Try your program by calling for `revlist([[a,b],[3,q],[1,h]]);`.

4. Write a procedure `mod3primes` whose input is n, and whose output is the list of just those prime numbers between 1 and n that leave a remainder of 3 when divided by 4. (See the Maple instruction `mod` and see also `isprime` in the package `numtheory`). Run the program with $n = 100$ and hand in that output.

5. Explain the error message that you will get if you run `try(3);` where the procedure `try` is the following:

```
try:=proc(x)
local zz;
x:=x^2+1;
zz:=3;
RETURN(zz);
end:
```

3 Sets and subsets

3.1 What they are

We discuss only finite sets. A set is a collection of objects. The set $S = \{1, 2, 3, 4\}$ is the collection of those four objects. No significance is attached to the order in which the elements of a set are listed. Thus $S = \{1, 2, 3\} = \{1, 3, 2\} = \{3, 1, 2\}$ are all the same set. A set is like a club: all that matters is whether you are a member or not.

The set of no objects at all is the empty set, written $\emptyset$. A subset T of a set S, written $T \subseteq S$, is a set all of whose elements are also members of S, e.g., $\{1, 3\} \subseteq \{1, 2, 3, 4\}$, and $\emptyset \subseteq S$ is true whatever the set S might be.

3.2 How many there are

The empty set has exactly one subset: itself. The set $S = \{1\}$ of 1 object has exactly two subsets, namely $\emptyset$ and $\{1\}$. In general, how many subsets does a set of n objects have?

Let's gather a little more data first. If $f(n)$ is the answer then we have already noted that $f(0) = 1$ and $f(1) = 2$. The reader will quickly check that $f(2) = 4$ and $f(3) = 8$, so our sequence of values of $f(n)$ begins as $1, 2, 4, 8, \ldots$. Well, the next value *might* be 611, but we would bet on 16. In fact, it is starting to look suspiciously as if a set S of n objects has exactly $f(n) = 2^n$ subsets, for each $n = 0, 1, 2, \ldots$.

But why?

Suppose our set S is $\{1, 2, 3, 4\}$. Let's construct a subset T of it. Ready? Pick up the element 1. Shall we put it into the subset T that we are constructing, or not? We can make that decision in two ways, "1 is in" or "1 is out." Having decided the

fate of 1, we move on to 2. Shall we put 2 into the subset that we are building, or not? That decision also can be made in two ways. So *both* of these decisions, the one that affects 1 and the one that affects 2, can be made in four ways. Similarly, we can decide whether 3 shall or shall not be in the subset that we're making in two ways, and two ways for 4, likewise. So this set S of four elements has 16 subsets, one corresponding to each way that we make all four of the decisions that affect the memberships of 1,2,3 and 4.

Theorem 3.1 *For each $n = 0, 1, 2, \ldots$, the set $S_n = \{1, 2, \ldots, n\}$ has exactly 2^n subsets.*

Proof. Clearly true for $n = 0$. If $n > 0$ then S_n has two kinds of subsets, those that do contain n (east-side sets) and those that don't (west-side sets). Evidently there are equal numbers of these two kinds of sets since if we delete n from an east-side set we get a west-side set, and this mapping is 1-1. But by induction, there are 2^{n-1} east-side sets. Thus there are $2 \cdot 2^{n-1} = 2^n$ subsets altogether. □

Example 3.1. Suppose $T \subseteq \{1, 2, \ldots, n\}$. We'll define the *spread* $s(T)$ of T to be the largest element of T minus its smallest element. So $s(\{3, 4, 7\}) = 4$, for instance. OK, for a fixed integers $n > 1$ and $1 \leq s \leq n - 1$, how many subsets of $\{1, 2, \ldots, n\}$ have spread s?

If the smallest element of such a subset is m, then the largest is $m + s$, for $1 \leq m \leq n - s$. What about the other elements of such a subset? They can be any subset at all of the set $\{m + 1, m + 2, \ldots, m + s - 1\}$, so there are 2^{s-1} such subsets for every m, $1 \leq m \leq n - s$, making a total of $(n - s)2^{s-1}$ subsets of $\{1, 2, \ldots, n\}$ that have spread s. □

3.3 Probabilities and averages

Continuing the example above, what is the *probability* that a randomly chosen subset $T \subseteq \{1, 2, \ldots, n\}$ will have spread s? It is the number of subsets of spread s divided by the number of subsets, i.e., $(n - s)2^{s-1}/2^n = (n - s)2^{s-1-n}$. Thus, for each $n > 1$ and $1 \leq s \leq n - 1$, the probability p_s that a subset has spread s is

$$p_s = (n - s)2^{s-1-n}. \qquad (n > 1;\ 1 \leq s \leq n - 1) \tag{3.1}$$

Now let's find the *average* spread of one of these subsets. But first we'd better talk about how to find averages of things in general.

Suppose that students' scores, on a scale of 1-10, from a certain exam are

$$7, 4, 5, 5, 8, 9, 6, 8, 5, 4, 1, 6, 4, 7, 6, 9, 9, 7, 10, 3, 5.$$

How might we calculate the class average? The straightforward way would be to add up all of the scores and divide by the number of scores, to get

$$\frac{7+4+5+5+8+6+8+5+4+1+6+4+7+6+9+9+7+10+3+5}{20}$$
$$= \frac{119}{20} = 5.95.$$

After doing this for a few times you would quickly learn to *group* the data before calculating. You would say that one student got a 1, none got a 2, one score was 3, there were three 4's, four 5's, three 6's, three 7's, two 8's, two 9's and one 10. Therefore the average is

$$\frac{1\cdot 1 + 0\cdot 2 + 1\cdot 3 + 3\cdot 4 + 4\cdot 5 + 3\cdot 6 + 3\cdot 7 + 2\cdot 8 + 2\cdot 9 + 1\cdot 10}{20} = 5.95,$$

as before.

In the numerator of the grouped method of finding the average, we see the sum of each integer times the frequency with which it occurred, and in the denominator is the total number of students. So if for each $j = 0, 1, 2, \ldots, 10$, the number of students who got grade j was n_j, then this formula says that the average is

$$\frac{\sum_{j=0}^{10} j n_j}{N} = \sum_{j=0}^{10} j \left(\frac{n_j}{N}\right)$$

where $N = \sum_j n_j$ is the total number of students. But in this formula we recognize the quantity in parentheses, n_j/N, as the *probability* that a student will get a score of j. So the grouped formula for finding the average score is

$$\text{average} = \sum_j j p_j. \tag{3.2}$$

So what is the average spread of a subset of $\{1, 2, ..., n\}$? The probability of spread s is given by (3.1), so by (3.2) the average spread is

$$\bar{s}(n) = \sum_{s=1}^{n-1} s(n-s)2^{s-1-n}. \tag{3.3}$$

This sum can actually be expressed in a simple, closed form. To do this without[1] having to think about it, you can open a Maple worksheet and type

$$\texttt{simplify(sum(s * (n - s) * 2\^{}(s - 1 - n), s = 1..n - 1));}$$

Maple will respond with $n - 3 + (n + 3)2^{-n}$ as the value of the sum. The average spread of a subset of $\{1, 2, \ldots, n\}$ is therefore very close to $n - 3$.

3.4 k-subsets

A k-subset of a set S is a subset whose cardinality is k. The 2-subsets of $\{1, 2, 3, 4\}$ are $\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}$. There are six of them. In general, a set of n objects has exactly $\binom{n}{k} = n!/(k!(n-k)!)$ k-subsets. The numbers $\binom{n}{k}$ ("n choose k") are the famous binomial coefficients. If n is a nonnegative integer, then the binomial coefficient $\binom{n}{k}$ vanishes if $k < 0$ or $k > n$.

If n is any real number then the binomial coefficient is well-defined as long as k is a nonnegative integer. Indeed

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{k!},$$

and that makes perfect sense even if, say, $n = -3/4$. For example,

$$\binom{-\frac{3}{4}}{2} = \frac{(-\frac{3}{4})(-\frac{7}{4})}{2} = \frac{21}{32}.$$

The binomial coefficients can be nicely displayed in a triangular array called Pascal's triangle. The rows are indexed by $n = 0, 1, 2, 3, \ldots$, and in the nth row of Pascal's triangle there are the numbers $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots, \binom{n}{n}$. The triangle begins as in Fig. 3.4.

3.5 East side, west side,... (I)

A quick glance at Pascal's triangle suggests that each entry in it is the sum of the two entries that are northeast and northwest of it. In symbols, what we see is that

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{3.4}$$

[1]This sum can be done in many ways. But computers can now do all sums of this general kind, completely automatically, giving you a simple formula for the sum, or else proving that no simple formula exists. So why work? Be lazy, and just hit `Enter`. See [PWZ] for details.

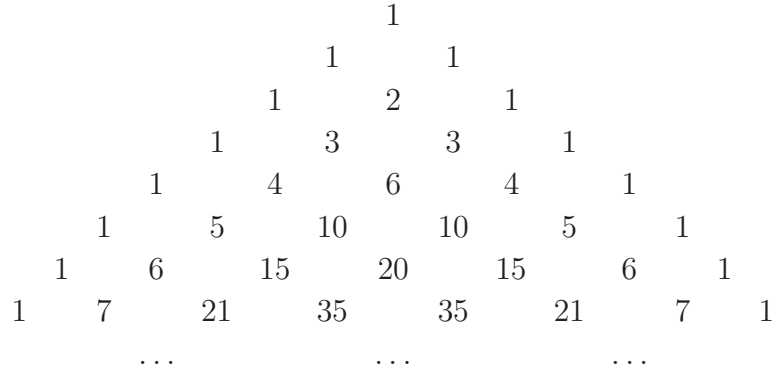

Figure 3.1: The Pascal triangle of binomial coefficients

There are many ways to prove this, but the one that we will use is the east-side-west-side paradigm, because the same method will work on a variety of problems that we will encounter later.

Let's do it. The number $\binom{n}{k}$ counts the k-subsets of the set $\{1, 2, 3, \ldots, n\}$. Imagine that all of these k-subsets have been strewn on the sidewalks and boulevards, all around the town. Now take a walking tour and inspect them as they lie on the ground. If a certain subset does not contain the letter n, then move it to the east side of town, and if it does contain n then move it to the west side.

Now instead of one large collection of subsets all over the place, we have two nice neat piles, one on the east side, and the other on the west side of town.

How many subsets are on the east side? These are the ones that don't have n in them. Since they don't have n then they must all be subsets of $\{1, 2, 3 \ldots, n-1\}$. Since they are k-subsets, they must be k-subsets of $\{1, 2, \ldots, n-1\}$, and there are $\binom{n-1}{k}$ of these.

So the total number of k-subsets of $\{1, 2, \ldots, n\}$ that are on the east side of town is $\binom{n-1}{k}$.

How many subsets are on the west side? These are the ones that do have n in them. If such a k-subset does contain n, what must the rest of the elements of the subset be? Evidently they must be some $(k-1)$-subset of $\{1, 2, \ldots, n-1\}$, and there are $\binom{n-1}{k-1}$ of these.

So the total number of k-subsets of $\{1, 2, \ldots, n\}$ that are on the west side of town is $\binom{n-1}{k-1}$.

Therefore the number of all k-subsets, which is $\binom{n}{k}$, must be equal to $\binom{n-1}{k}+\binom{n-1}{k-1}$ since every k-subset landed on one side of town or the other. □

That proof was perhaps a bit more complicated than this particular situation requires, but it has the advantage that it will apply in numerous similar situations.

3.6 Making lists and random choices of sets and subsets

The counting arguments are directly wired to algorithms for making lists and for choosing objects at random.

Consider the question of making a list of all subsets of $\{1, 2, \ldots, n\}$. We discovered that there are 2^n of these by observing that a subset can be described by giving the status of each possible element $\{1, 2, \ldots, n\}$ as "in" or "out". This can conveniently be done with a vector of 0's and 1's, where a 1 means that the element is in, and a 0 means it is out, of the subset under consideration. Thus 0001 is the subset $\{4\}$ of $\{1, 2, 3, 4\}$, while 1010 is the subset $\{1, 3\}$ and 0000 is the empty subset.

We can make a list $L(n)$ of all subsets of $\{1, 2, \ldots, n\}$ recursively, as follows. Take the list $L(n-1)$ and augment every string of 0's and 1's on the list by adjoining a 0 to the beginning of the string. Then take the same list $L(n-1)$ and augment every string of 0's and 1's on the list by adjoining a 1 to the end of the string. The union of the two lists just produced is the desired list $L(n)$.

For instance, the list $L(1)$ is `[[0],[1]]`. So, for $n = 2$, we first glue a 0 onto the beginning of each member of $L(1)$, getting `[[0,0],[0,1]]`, and next we glue a 1 onto the beginning of each member of $L(1)$, which gives `[[1,0],[1,1]]`. Now $L(2)$ is the union of these two lists, namely $L(2) =$`[[0,0],[0,1],[1,0],[1,1]]`. A Maple program[2] that performs this task is as follows.

```
GlueIt:=(z,d)->[d,op(z)]:
Subsets:=proc(n)
 local east, west ; options remember;
 if n=0 then RETURN([[]])
        else
    east:=map(GlueIt,Subsets(n-1),0);
    west:=map(GlueIt,Subsets(n-1),1);
    RETURN([op(east),op(west)])
 fi;
 end:
```

[2]See the program notes on page 59.

If we call this program with
`>Subsets(3)`
for instance, then Maple will reply with

$$[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]] \tag{3.5}$$

Next suppose we want to choose a subset of $\{1,2,\ldots,n\}$ uniformly at random (uar), i.e., in such a way that each of the 2^n possible subsets will have an equal chance of being selected. That's easy too. Just choose a random integer, using your random number generator, in the range $0,1,\ldots,2^n-1$, and express it as a binary string of 0's and 1's, and you're all done. Equivalently, toss a fair coin n times. After the ith toss, put the letter i into the random subset that is under construction if the coin came up as Heads, and leave i out of the subset if it landed as Tails. Note that in Maple, a call to `rand` returns a *function* for generating random numbers, rather than the random numbers themselves. A Maple program[3] for choosing a random subset of $\{1,2,\ldots,n\}$ is shown below.

```
RandSub:=proc(n)
local rn, set, i;
rn:=rand(0..1):
set:=[]:
for i from 1 to n do set:=[op(set),rn()] od:
RETURN(set):
end:
```

So much for *all* subsets of n things. Let's turn now to subsets of given cardinality k ("k-subsets"). With k-subsets we can also use the east side-west side method. To make a list $L(n,k)$ of all k-subsets of an n-set we recursively make the lists $L(n-1,k-1)$ and $L(n-1,k)$. The desired list $L(n,k)$ is obtained by beginning with the list $L(n-1,k)$ and following that by the list that is obtained by adjoining the element n to each member of $L(n-1,k-1)$. Let's write that symbolically as

$$L(n,k) = L(n-1,k), L(n-1,k-1) \otimes n. \tag{3.6}$$

To implement that in Maple, we will represent each subset by a list of its members. Here is a procedure[4] that will list all of the k-subsets of $\{1,2,\ldots,n\}$.

[3] See the program notes on page 59.
[4] See the program notes on page 59.

```
westop:=(y,m)->[op(y),m]:
ListKSubsets:=proc(n,k)
local east,west ; options remember;
#Returns a list of the k-subsets of {1,2,...,n}
 if n<0 or k<0 or k>n then RETURN([])
                      elif n=0 or k=0 then RETURN([[]])
                      else
   east:=ListKSubsets(n-1,k);west:=ListKSubsets(n-1,k-1);
   west:=map(westop,west,n);
   RETURN([op(east),op(west)]);
 fi;
end:
```

For example, a call to `ListKSubsets(5,3)` produces the response

$$[[1,2,3],[1,2,4],[1,3,4],[2,3,4],[1,2,5],[1,3,5],[2,3,5],[1,4,5],[2,4,5],[3,4,5]]. \tag{3.7}$$

To choose, uniformly at random, a k-subset of $\{1,\ldots,n\}$, here's what to do: With probability $\binom{n-1}{k-1}/\binom{n}{k} = k/n$ choose uar a $(k-1)$-subset of $\{1,\ldots,n-1\}$ and adjoin n to it, or with probability $1-k/n$, simply output a randomly chosen k-subset of $\{1,\ldots,n-1\}$. The Maple program[5] follows.

```
RandomKSubsets:=proc(n,k)
local rno,east,west ;
if n<0 or k<0 or k>n then RETURN()
                      elif n=0 and k=0 then RETURN([])
                      else
rno:=10^(-12)*rand();
if rno<k/n then
  east:=RandomKSubsets(n-1,k-1);
  RETURN([op(east),n])
           else
  west:=RandomKSubsets(n-1,k);
  RETURN(west)
fi;fi;
end:
```

[5]See the program notes on page 60.

3.7 Ranking sets and subsets

Suppose we have a list: {horse, turtle, cow, pig}. The *rank* of an element of the list is simply its position in the list, with however one technical ingredient added: the ranks start at 0. Thus, in the list above, the rank of "horse" is 0, the rank of "turtle" is 1, the rank of cow is 2 and the rank of "pig" is 3.

For a somewhat more realistic example, in the list (3.7), the rank of `[1,2,3]` is 0, and the rank of `[2,3,4]` is 3.

The *ranking problem* is this: given an element of a list, find its rank in that list. Of course if the list itself has actually been computed already then we can find the rank of an element just by searching for it in the list. In the ranking problems that we consider here, one should think of the lists as *not* having been already computed, and we are being asked to find the rank of a given element in a list even though the list is not available.

Example: What's the rank of the subset `[3,4,7,9]` in the list of all of the 2^{10} subsets of $\{1, 2, 3, \ldots, 10\}$?

To answer this, first observe that we are representing ("encoding") subsets by strings of 0's and 1's. This particular subset is therefore the string {0011001010}.

Next, the way we listed the subsets was that we first listed all of the strings that begin with a 0 and then we listed all that begin with a 1. Now here's the key point: this string `ss` begins with a 0. Therefore it occurs in the first half of the list, and its position in that first half is the same as the rank of the truncated string `ss`':={011001010} in the shorter list of all subsets of $\{2, 3, \ldots, 10\}$.

But now suppose that the given string `ss` begins with a 1, say

$$\texttt{ss} := \{1011001010\}.$$

Then it lies in the second half of the full list $L(10)$, and what is its rank in that full list? Well, all of the 2^9 strings that begin with a 0 precede it in the full list. So its rank is 2^9 plus the rank of the truncated string $\texttt{ss}' := \{011001010\}$.

The beauty of the recursive view of the world is that we don't have to say "contiuing in this way," or "and so on," at this point. Instead we can just leave it at that and write out a formal scheme for ranking a given string of n 0's and 1's in the list $L(n)$.

Let `ss` be the given string, let $\texttt{ss}'$ denote the truncation of `ss` that is obtained by deleting its leading 0 or 1, and let $\text{rank}(\texttt{ss}, L(n))$ denote the rank of `ss` in the list $L(n)$. If the first bit of `ss` is a 0, then

$$\text{rank}(\texttt{ss}, L(n)) = \text{rank}(\texttt{ss}', L(n-1)),$$

while if the first bit is a 1 then

$$\text{rank}(\texttt{ss}, L(n)) = 2^{n-1} + \text{rank}(\texttt{ss}', L(n-1)).$$

Thus we have the following Maple program for computing the rank of a subset. The subset is input as an array of 0's and 1's.

```
truncate:=ss->[op(2..nops(ss),ss)]:
  RankSubset:=proc(ss)
  #Finds the rank of subset ss in the list L(nn), where
  #nn is the length of the array ss.
  local nn;
  nn:=nops(ss);
  if nn=0 then RETURN(0)
     elif ss[1]=0 then RETURN(RankSubset(truncate(ss)))
     else RETURN(RankSubset(truncate(ss))+2^(nn-1))
  fi;
  end:
```

Now we can answer the question that was posed in the example above. To find the rank of the subset `[3,4,7,9]` in the list of all of the 2^{10} subsets of $\{1, 2, 3, \ldots, 10\}$, we call

$$\texttt{RankSubset}([0, 0, 1, 1, 0, 0, 1, 0, 1, 0]).$$

Maple returns 202, so this is the subset of rank 202 in the list, which is to say that `[3,4,7,9]` is the $203rd$ subset in the list of all 1024 subsets of $\{1, 2, 3, \ldots, 10\}$.

Now we turn to ranking k-subsets. Given some k-subset S, what is the rank of S in the list of all $\binom{n}{k}$ k-subsets of $\{1, 2, 3, \ldots, n\}$? But, in exactly which list of those k-subsets? The list that we'll have in mind is the list $L(n, k)$ that is produced by `ListKSubsets` above.

In that list $L(n, k)$, all of the k-subsets that do not contain the letter n precede all of the subsets that do contain n. But that says it all. So to find the rank of a given k-subset S in $L(n, k)$,we ask if $n \in S$ or not. If $n \notin S$, then the rank of our set is the same as its rank in the smaller list $L(n-1, k)$ of all k-subsets of $\{1, 2, \ldots, n-1\}$. If, on the other hand, $n \in S$, then let $S' = S \backslash \{n\}$. The rank of S in $L(n, k)$ is $\binom{n-1}{k}$ plus the rank of S' in $L(n-1, k-1)$. The Maple program is as follows.

```
 cutn:=w->[op(1..nops(w)-1,w)];
 RankKSubset:=proc(ss,n,k);
#Finds the rank of the k-subset ss in the list of all
#k-subsets of 1,...,n. ss is given as a list of members.
 if k=0 then RETURN(0)
        elif ss[k]=n
    then RETURN(RankKSubset(cutn(ss),n-1,k-1)+binomial(n-1,k))
    else RETURN(RankKSubset(ss,n-1,k))
 fi;
 end:
```

Now if we call

```
> RankKSubset([1,2,5],5,3);
```

Maple responds with 4, in agreement with eq. (3.7).

3.8 Unranking sets and subsets

The unranking problem is the inverse of the ranking problem. In ranking, we are given a combinatorial object, such as a set, and we are asked to find its rank on a certain list. In unranking we are given an integer r and a list, and we are asked to find the object whose rank is r on that list.

For instance, which is the subset of rank 784 in the list of all subsets of 13 objects? To answer that, observe that there are $2^13 = 8192$ subsets on the full list, so the one we seek lies in the first half of the list. But every subset in the first half of the list does not contain the first object. That is, the first entry in the array of 0's and 1's that describes the subset will be a 0. The rest of that array will be the set of rank 784 on the list of all subsets of 12 objects.

Suppose we had asked for the subset of rank 5000 in the same list. Then the first entry of the output array would be 1. The rest of the output array would be the set of rank $2^{12} - 5000 = 3192$ in the list of all subsets of 12 objects.

This reasoning leads to the following Maple program.

```
UnrankSubsets:=proc(r,n)
#Finds the subset of rank r in the list of all subsets of n things
#Subsets are presented as bit arrays
 if n=0 then RETURN([])
        elif r<2^(n-1)
   then RETURN([0,op(UnrankSubsets(r,n-1))])
   else RETURN([1,op(UnrankSubsets(r-2^(n-1),n-1))])
 fi;
 end:
```

So if we call

```
> UnrankSubsets(784,13);
```

then Maple would quickly inform us that the subset of rank 784 in the list of all subsets of 13 things is

$$[0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0].$$

3.9 Exercises

1. How many subsets of n things have even cardinality? Your answer should not contain any summations.

2. How many subsets of $\{1, 2, \ldots, n\}$ contain no two consecutive elements? Hint: Let $f(n)$ be the number, and find a recurrence formula for $f(n)$ by the east side-west side way of thinking.

3. How many k-subsets of $\{1, 2, \ldots, n\}$ contain no two consecutive elements?

4. (a) How many ordered pairs (A, B) of subsets of $\{1, 2, \ldots, n\}$ are there such that A, B are disjoint? Your answer should be a simple function of n, with no summation signs involved.

   (b) How many ordered pairs (A, B) of subsets of $\{1, 2, \ldots, n\}$ are there such that $B \subseteq A$? Your answer should be a simple function of n, with no summation signs involved.

   (c) Explain the striking similarity of the answers to 4a and 4b above. That is, show that those two answers must be the same without finding either answer explicitly.

5. Find a simple formula for the average of the squares of the first n whole numbers. For the average of the squares of the first n odd numbers.

6. Write, debug, and run Maple programs that will do each of the following:

   (a) Calculate $n!$, recursively.

   (b) Make a list of all subsets of $\{1, 2, \ldots, n\}$ that contain no two consecutive elements, recursively.

   (c) Choose, uar, a pair of disjoint subsets of $\{1, 2, \ldots, n\}$.

   (d) Calculate the average size of the largest gap between two consecutive elements of a subset of $\{1, 2, \ldots, n\}$, and plot a graph of this for $n = 5(5)100$.

   (e) Find the first four perfect numbers. A number n is perfect if it is equal to the sum of all of its divisors except for itself. The first two perfect numbers are 6 and 28. Use the Maple package `numtheory`.

7. Similarly to the recursive construction (3.6), write a Maple procedure that will, for n, k given, output a list of all of the k-subsets of $\{1, 2, \ldots, n\}$, subject to the following restriction. The successor of each set S must be a set that can be obtained from S by deleting one element and adjoining one element. For example, here is the beginning of such a list when k=4 and n=7:

   ```
   [1,2,3,4],[1,2,4,5],[2,3,4,5],[1,3,4,5],[1,2,3,5],[1,2,5,6],..
   ```

   Hint: Modify the recurrence (3.6) by doing a bit of list-reversal.

8. The purpose of this exercise is to run a few tests of Maple's random number generator, to see "how random" the numbers that it produces really are.

   (a) Generate 1000 random real numbers x in the range $0 < x < 1$. Tabulate the number of them, $n_1, \ldots, n_{10}$, that lie in each of the 10 subintervals $(0, .1), \ldots, (.9, 1.0)$. Roughly 100 of them should lie in each of these subintervals.

   (b) Expand your program for problem 8a above, so it computes the χ^2 (*chi-squared*) statistic. To do this, suppose that $n_1, n_2, \ldots, n_{10}$ are the 10 occupancy numbers that your program of exercise 8a produces. Then calculate

$$\chi^2 := \sum_{i=1}^{10} \frac{(n_i - 100)^2}{100}.$$

If Maple's random numbers are "truly random" then 95 percent of the time this χ^2 statistic should lie between 3.3 and 17. Does it?

(c) *The coupon-collector's test.* Generate random integers m, $1 \leq m \leq 10$, only until each of the integers $1, 2, \ldots, 10$ has been obtained at least once. Tabulate the number of random integers that you generated until that complete collection was obtained. Call all of that one "experiment." Do 500 such experiments, and print out the average number of random numbers that you used per experiment. If Maple's random numbers are good ones, then this average should be rather close to $10(1 + 1/2 + 1/3 + \ldots + 1/10)$, which is about 29.3. That is, if you want to see each integer $1, 2, \ldots, 10$ at least once, then you had better generate about 29 random integers independently. Is your observed average near 29?

9. (Gray codes) A Gray code is a list of all 2^n possible strings of n 0's and 1's, arranged so that successive strings on the list differ only in a single bit position. For example,

$$000, 001, 011, 010, 110, 111, 101, 100$$

is such a list, when $n = 3$.

(a) Suppose $\mathcal{L}(n)$ is such a list, for a certain value of n. Then show that we can get a list for $n+1$ if we (a) concatenate a new 0 bit onto the beginning of each member of $\mathcal{L}(n)$, and (b) follow that with the result of concatenating a new 1 bit onto the beginning of each member of the *reverse* of the list $\mathcal{L}(n)$.

(b) Write a fully recursive Maple program which, given n, will output the list $\mathcal{L}(n)$ described above.

(c) Let's number the 2^n strings on the list $\mathcal{L}(n)$ with the integers $0, 1, \ldots, 2^n - 1$. The number that each string receives will be called its rank. Suppose we are given a particular string σ, of n 0's and 1's. Show that the following procedure will produce its rank in the list $\mathcal{L}(n)$: Read the string σ from left to right. Each time you encounter a bit that has an even number of 1's to its left, enter that bit unchanged into the output string σ'. Each time you meet a bit that has an odd number of 1's to its left in σ, enter the complement of that bit into the output string σ'. When finished, σ' will hold the binary digits of the rank of the input string σ.

4 Permutations and their cycles

4.1 What permutations are

A permutation of a set S is a 1-1 mapping of S to itself. The mapping

$$\{1 \to 3, 2 \to 2, 3 \to 5, 4 \to 6, 5 \to 4, 6 \to 1\}$$

is a permutation of the set $\{1, 2, 3, 4, 5, 6\}$. This particular permutation would usually be written in *the two line form*

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 2 & 5 & 6 & 4 & 1 \end{pmatrix}$$

in which we see $f(x)$ directly underneath x, for each $x = 1, \ldots, 6$.

4.2 What cycles are

Suppose f is a permutation of some finite set S. We will define the *cycles* of f. If x and y are two elements of S, say that x and y are equivalent if $y = f^j(x)$, for some integer $j <=> 0$. Note here that $f^j(x)$ means the result of applying f j times to x, if $j \geq 0$, or of applying f^{-1} $|j|$ times, if $j < 0$.

This is an equivalence relation on the set S. Indeed, $x = f^0(x)$, so the relation is reflexive. Further, if $x = f^j(y)$ then $y = f^{-j}(x)$, so the relation is symmetric. Finally, if $x = f^j(y)$ and $y = f^k(z)$ then $x = f^{j+k}(z)$, which shows the transitivity.

The cycles of f are the equivalence classes of this relation. But they aren't just sets. The elements within a cycle are arranged in an ordered list. To exhibit a cycle explicitly, we choose an element x, follow it with $f(x)$, then with $f(f(x))$, etc., until x reappears. The cycle then terminates, and we begin the next cycle with some element that is not in a cycle that has already been constructed.

In the example permutation above, there is a cycle $1 \to 3 \to 5 \to 4 \to 6$ and another one that contains only 2. Hence we can write this same permutation f *in cycle form* by exhibiting the cycles explicitly, as (13546)(2). In the cycle form, the parentheses delimit the cycles. Each element is the image, under the permutation, of the element that is written immediately to its left, except that the first element of each cycle is the image of the last element of the same cycle.

A larger example is provided by the permutation of 12 letters which in two line form is

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 5 & 12 & 1 & 9 & 7 & 6 & 3 & 10 & 4 & 8 & 2 & 11 \end{pmatrix}$$

					1							
				1		1						
			2		3		1					
		6		11		6		1				
	24		50		35		10		1			
120		274		225		85		15		1		
720	1764		1624		735		175		21		1	
	...				...				...			

Figure 4.2: The triangle of Stirling cycle numbers

This same permutation in cycle form is (1 5 7 3)(2 12 11)(4 9)(6)(8 10), and it has five cycles altogether.

4.3 Counting permutations by cycles

There are $n!$ permutations of n letters. These permutations have various numbers of cycles, from a minimum of one cycle to a maximum of n cycles. For each n, k we introduce the binomial-coefficient-like symbol $\begin{bmatrix} n \\ k \end{bmatrix}$ for the number of permutations of n letters that have exactly k cycles. These are the *Stirling cycle numbers*. In the literature they are also called the absolute values of the *Stirling numbers of the first kind*. Thus, for each n we will have $\sum_k \begin{bmatrix} n \\ k \end{bmatrix} = n!$.

Evidently $\begin{bmatrix} 1 \\ 1 \end{bmatrix} = 1$, $\begin{bmatrix} 2 \\ 1 \end{bmatrix} = 1$, $\begin{bmatrix} 2 \\ 2 \end{bmatrix} = 1$. Of the six permutations of three letters, written in cycle form, there are two that have one cycle, namely (123) and (132) (why not (312)?), and three that have two cycles, namely (1)(23) and (2)(13) and (3)(12), and one that has three cycles, namely (1)(2)(3). We can create a Pascal-like triangle that has in it the Stirling cycle numbers $\begin{bmatrix} n \\ k \end{bmatrix}$, instead of the binomial coefficients. This triangle begins as in Fig. 4.2. The sum of the entries in the nth row is $n!$, for each $n = 1, 2, 3, \ldots$. We notice that it is no longer true that each entry in the triangle is the sum of the two that are diagonally above it. But there is a very similar recurrence that does hold in this triangle, and which greatly facilitates the computation of these numbers.

4.4 East side, west side ... (II)

Fix positive integers n, k. Imagine that all of the permutations of n letters that have exactly k cycles have been strewn on the sidewalks and streets, all around the town.

Now take a walking tour and inspect them as they lie on the ground. If, in a certain permutation, the letter n lives in a cycle all by itself, then move that permutation to the east side of town, and if, on the contrary, n lives in a cycle with at least one other letter, then move it to the west side.

Now instead of one large collection of permutations all over the place, we have two nice neat piles, one on the east side, and the other on the west side of town.

How many permutations are on the east side? These are the ones in which n lives all by itself in a cycle of length 1. How many such permutations are there? Well, since n lives alone, the rest of the permutation must be comprised of $k-1$ cycles involving the other $n-1$ letters. That means that altogether, there are $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$ permutations that are stacked up on the east side.

How many are on the west side? These are the ones in which n lives in a cycle with other letters. If we remove the letter n from the cycle that it lives in, we will be looking at some permutation of $n-1$ letters into the same number, k, of cycles.

However, the converse is a bit sticky. If, conversely, we take some permutation of $n-1$ letters into k cycles, there are many ways in which we might insert the letter n into one of the cycles. In fact, since there are $n-1$ letters in the permutation, and we can insert the letter n in between any two consecutive letters on any cycle, there are $n-1$ ways to insert n.

That means that the number of permutations on the west side is $n-1$ times the number of permutations of $n-1$ letters into k cycles, i.e., it is $(n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$.

Therefore the number of all permutations of n letters that have k cycles must be equal to $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$ since every such permutation landed on one side of town or the other.

Theorem 4.1 *The Stirling cycle numbers* $\begin{bmatrix} n \\ k \end{bmatrix}$ *satisfy the Pascal-triangle-like recurrence*

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}. \tag{4.1}$$

Here is a recursive Maple procedure to calculate the Stirling cycle numbers.

```
StirCyc:=proc(n,k)
 options remember;
#Returns the no. of perms of n letters with k cycles
if n<1 then RETURN(0)
       elif n=1 then
         if k<>1 then RETURN(0)
                 else RETURN(1) fi
                else
RETURN(StirCyc(n-1,k-1)+(n-1)*StirCyc(n-1,k))
fi:
end:
```

4.5 The generating function

From the recurrence (4.1) we can easily find a nice formula for the generating function

$$f_n(x) \stackrel{def}{=} \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k,$$

and that formula will come in handy when we want to find the average number of cycles that permutations of n letters have.

To find it, multiply both sides of (4.1) by x^k and sum over all integers k. On the left side we will obtain $f_n(x)$, the unknown generating function. The second term on the right will give us $(n-1)f_{n-1}(x)$, so that part is easy.

What about the first term on the right? It is

$$\sum_k \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} x^k = x \sum_k \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} x^{k-1} = x f_{n-1}(x).$$

(Be sure to absorb that footwork before proceeding.) We have found that

$$f_n(x) = x f_{n-1}(x) + (n-1) f_{n-1}(x) = (x+n-1) f_{n-1}(x).$$

Since $f_1(x) = x$ (why?), we find successively that $f_2(x) = x(x+1)$, $f_3(x) = x(x+1)(x+2)$, and so forth. By induction, it is trivial now to establish the following result.

Theorem 4.2 *The Stirling cycle number* $\begin{bmatrix} n \\ k \end{bmatrix}$ *is the coefficient of* x^k *in the polynomial* $x(x+1)\ldots(x+n-1)$. *That is, we have*

$$\sum_{k=1}^{n} \begin{bmatrix} n \\ k \end{bmatrix} x^k = x(x+1)(x+2)\ldots(x+n-1). \tag{4.2}$$

For example,

$$x(x+1)(x+2)(x+3) = 6x + 11x^2 + 6x^3 + x^4,$$

and the row of coefficients is identical with the fourth row in the triangle of Fig. 4.2 (do have a look, and check this).

4.6 The average number of cycles

On average, how many cycles does a permutation of n letters have? We use equation (3.2) for the average expressed in terms of the probabilities of each value.

First, what is the probability that a permutation of n letters has exactly k cycles? Evidently it is $p_k = \begin{bmatrix} n \\ k \end{bmatrix}/n!$. Now from equation (3.2) we have that the average number of cycles among permutations of n letters, let's call it $\bar{k}(n)$, is

$$\begin{aligned} \bar{k}(n) &= \sum_k k p_k \\ &= \sum_k k \frac{\begin{bmatrix} n \\ k \end{bmatrix}}{n!} \\ &= \frac{1}{n!}\left[\frac{d}{dx}\left(\sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k\right)\right]_{x=1} \\ &= \frac{1}{n!}\left[\frac{d}{dx}\left(x(x+1)(x+2)\ldots(x+n-1)\right)\right]_{x=1}. \end{aligned} \tag{4.3}$$

If you ever have to differentiate a function $g(x)$ which is a product of lots of things, as is the case here, it's usually better to use the fact that

$$g'(1) = \frac{g'(1)}{g(1)} g(1) = g(1)\left(\frac{d}{dx}\log\left(g(x)\right)\right)_{x=1}.$$

With $g(x) = x(x+1)\ldots(x+n-1)$, this reads as

$$\begin{aligned} g'(1) &= n!\left[\frac{d}{dx}\left(\log x + \log x + 1 + \ldots + \log(x+n-1)\right)\right]_{x=1} \\ &= n!\left[\frac{1}{x} + \frac{1}{x+1} + \ldots + \frac{1}{x+n-1}\right]_{x=1} \\ &= n!\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right). \end{aligned}$$

If we substitute this in (4.3) we find that

$$\bar{k}(n) = \frac{1}{n!}n!(1 + \frac{1}{2} + \ldots + \frac{1}{n}) = 1 + \frac{1}{2} + \ldots + \frac{1}{n}.$$

Let's define the nth *harmonic number* H_n to be

$$H_n \stackrel{def}{=} 1 + \frac{1}{2} + \ldots + \frac{1}{n}.$$

Theorem 4.3 *The average number of cycles that permutations of n letters have is the harmonic number H_n.*

When n is large, the harmonic number H_n is well approximated by $\log n$, as can be seen by comparing it with the integral $\int_1^n \frac{dt}{t}$. So permutations of n letters have an average of about $\log n$ cycles.

4.7 An application

We consider an application of the theory of permutations and their cycles to a simple algorithm for finding the largest member of a set or list.

Suppose $\{a_1, a_2, \ldots, a_n\}$ is a given list of n numbers, and we want to find the largest number in the list. We'll use the obvious algorithm, namely we will test each a_i, in turn, against the current contents of a certain quantity, `winner`, say. If $a_i >$ *winner*, then a_i will replace `winner`. Otherwise we just go to the next value of i. More formally, we'll do (using Maple) -

```
bigone:=proc(a,n)
local winner,i;
# Returns the largest element of a given
# array a of length n.
winner:=a[1];
for i from 2 to n do
   if a[i]>winner then winner:=a[i] fi;
                       od;
RETURN(winner);
end:
```

We want to know about the complexity of this routine, that is, about the amount of labor that is involved in using it. At first glance this seems totally trivial since we

do one pairwise comparison ("`if a[i]>winner then ..`") for each i, the thing does about n pairwise comparisons, and that all there is to say, right?

No, there's more to say. Aside from the pairwise comparisons of size, this algorithm does another kind of work: it moves data from one place to another. Every time that `a[i]` is bigger than `winner`, the program moves `a[i]` to `winner`. How much work is that? It depends. What it depends on is how many times that data movement takes place. In other words, it depends on the number of times that we find an array entry `a[i]` that is larger than the current occupant of the `winner`'s circle.

Let's assume that the array entries are all distinct. Then all that matter are the *relative* sizes of the array entries, so we might as well assume that the input array is (the 2nd line of the 2-line form of) a permutation of n letters.

As we read ('parse') the values of the permutation from left to right, the number of times the algorithm will have to replace the contents of `winner` is equal to the number of values of i for which $a[i]$ is larger than all $a[j]$ for $j < i$. Another way to think about it is that we are interested in the number of times a new world's record is set, as we read the values of the input permutation. For example, if the input array is

$$\{6, 2, 4, 8, 1, 9, 5, 3, 10, 7\}$$

then 4 world's records are set, since each of the 4 entries 6,8,9,10 is larger than every entry to its left.

An entry that is larger than everything to its left will be called an *outstanding element* of the permutation. The number of times our algorithm will have to move data is equal to the number of outstanding elements that the input array has. The array displayed above has four outstanding elements. Every permutation has at least one of them.

So our question is, *what is the average number of outstanding elements that permutations of n letters have?* Whatever that answer is, it's also the average amount of work that our algorithm will have to do by moving pieces of data around.

Theorem 4.4 *The number of permutations of n letters that have k outstanding elements is equal to the number that have k cycles.*

To see why this is so, we will exhibit an explicit 1-1 mapping between these two flavors of permutations.

First, given a permutation of n letters with k cycles, say σ, do the following.

1. Standardize σ by presenting the cycles in such a way that the largest element of each cycle appears first in that cycle, and the sequence of cycles of σ is arranged so that, from left to right, the sequence of largest elements within each cycle increases.

2. Then drop all of the parentheses that appear in the cycle presentation of σ.

3. Interpret the result as the second line of the two line form of a certain permutation $f(\sigma)$. This latter permutation has exactly k outstanding elements.

Conversely, if we are given the second line of the two line form of $f(\sigma)$, we can recover σ uniquely as follows.

1. Place a left parenthesis at the extreme left of the given line.

2. Read the line from left to right, and each time an element x is encountered that is larger than every element to its left, put a right parenthesis and an opening left parenthesis immediately to the left of x.

3. Place a right parenthesis at the extreme right of the given line.

4. Interpret the result as the cycle form of some permutation σ. That permutation will have exactly k cycles. □

For example, if we are given, in cycle form, $\sigma = (362)(4)(517)$, which has three cycles, we first standardize its presentation as $(4)(623)(751)$. Then we drop the parentheses to obtain $f(\sigma) = 4623751$, which is the second line of the 2-line form of a permutation that has three outstanding elements.

Conversely, if we are given $f(\sigma) = 4623751$, we insert parentheses as instructed to get $(4)(623)(751)$, which is the cycle form of a permutation σ with three cycles.

It follows from the theorem that the average number of outstanding elements of permutations of n letters is the same as the average number of cycles, namely, exactly H_n, and approximately $\log n$.

Therefore, in the course of finding the largest element of a given array, it will happen about $\log n$ times, on the average, that we will move some data from the array to `winner`.

4.8 Making lists and random choices of permutations and their cycles

Again, the counting arguments in this section will immediately suggest ways of making lists of permutations with given numbers of cycles and of making random choices of them.

Suppose n, k are given. To make a list of all permutations of n letters that have k cycles, let's list those in which n lives in a cycle by itself first, and then all of the others second. Suppose $L(n, k)$ denotes this list of permutations.

To list those in which n lives alone, we take the full list $L(n-1, k-1)$, and we glue onto each permutation on that list a new cycle that contains only the letter n. To list those in which n lives in a cycle with other letters, we take the full list $L(n-1, k)$, and for each permutation on that list we construct $n-1$ new permutations by inserting n succesively into the "cracks" between consecutive elements around each of its cycles.

Let's make the list $L(4, 2)$. First take $L(3, 1)$, which consists of the two permutations (123) and (132), in cycle form. We glue onto these two a new cycle that contains only the letter 4, to obtain the two east side permutations (123)(4) and (132)(4). Now take the list $L(3, 2)$, which consists of the three permutations (1)(23), (2)(13), and (3)(12), and insert the new letter 4 into each position between two consecutive letters on each cycle. This yields the nine west side permutations (41)(23), (1)(423), (1)(243) and (42)(13), (2)(413), (2)(143) and (43)(12), (3)(412), (3)(142). The desired list $L(4, 2)$ consists of the two east side followed by the nine west side permutations that we have just constructed.

Here is a Maple procedure[6] to make a list of all n-permutations with k cycles. Remarkably, it turns out to be nicer to implement this using the 2-line form of the permutation rather than the cycle form.

[6] See the program notes on page 60.

```
eastop:=(y,j)->[op(y),j]:
westop:=(y,j)->[op(subsop(j=1+nops(y),y)),y[j]]:
ListKPerms:=proc(n,k)
local east,west,out,j ; options remember;
#Lists, in 2-line form, the perms of n letters, k cycles
if n<1 or k<1 or k>n then RETURN([])
                     else
if n=1 then RETURN([[1]])
       else
east:=ListKPerms(n-1,k-1);west:=ListKPerms(n-1,k);
out:=map(eastop,east,n);
for j from 1 to n-1 do
   out:=[op(out),op(map(westop,west,j))] od;
RETURN(out);fi;fi;
end:
```

To run this program, we might call `ListKPerms(4,2);` and the output would then be as follows:

$$[[3,1,2,4],[2,3,1,4],[4,1,3,2],[4,2,1,3],[4,3,2,1],[2,4,3,1],\\ [3,4,1,2],[1,4,2,3],[2,1,4,3],[3,2,4,1],[1,3,4,2]] \tag{4.4}$$

Thus the output shows the eleven permutations of four letters that have two cycles. Notice that each permutation is displayed with the second line of its 2-line form, and not in its cycle form. Thus, the first output permutation, which is shown as `[3, 1, 2, 4]`, is the permutation f for which $f(1) = 3, f(2) = 1, f(3) = 2, f(4) = 4$, which indeed has two cycles.

How do we choose a permutation of n letters and k cycles uniformly at random? First we decide whether we are going to choose an east side or a west side permutation. How to do that? Well, the probability of choosing an east side permutation is obviously $p = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} / \begin{bmatrix} n \\ k \end{bmatrix}$. So calculate the number p, and then choose a random real number ξ in the range $(0, 1)$. If $\xi < p$ then the output will be an east side permutation, and otherwise it will be a west side permutation.

If the output is to be from the east side, then here's what to do. Recursively, choose uar a permutation of $n-1$ letters with $k-1$ cycles, and then glue onto it a new cycle that contains only the letter n. Finished. Otherwise, if you want a west side permutation, then recursively choose uar a permutation of $n-1$ letters with

k cycles. In this permutation there are $n-1$ cracks between consecutive letters on cycles. Choose one of these cracks uar and insert the new letter n into the chosen crack. All finished.

Here is a Maple procedure[7] that will do the random selection. Again, the procedure presents the output as the sequence of values of the permutation chosen, rather than in cycle form.

```
RandKPerms:=proc(n,k)
local rno,r,east,west,rn ;
#Returns a random perm of n letters, k cycles, in 2-line form
if n=1 then
   if k<>1 then RETURN([]) else RETURN([1]) fi;
       else
  rno:=10^(-12)*rand();
  if rno<StirCyc(n-1,k-1)/StirCyc(n,k) then
    east:=RandKPerms(n-1,k-1);
    RETURN([op(east),n])
                                        else
    rn:=rand(1..n-1);r:=rn();
    west:=RandKPerms(n-1,k);
    RETURN(subsop(r=n,[op(west),west[r]]))
  fi;fi;
end:
```

4.9 Ranking permutations by cycles

What is the rank of the permutation $\tau := (16)(382)(45)(7)$ (cycle form) of 8 letters with 4 cycles in the list of all such permutations? Recall our scheme for listing these permutations. First in the list are those in which the highest letter, in this case 8, is a fixed point of the permutation. In τ, 8 is not a fixed point. Thus all of the $\begin{bmatrix}7\\3\end{bmatrix}$ permutations in which 8 is a fixed point have lower rank than τ.

After those in the list come the $\begin{bmatrix}7\\4\end{bmatrix}$ permutations σ for which $\sigma(1) = 8$, then the $\begin{bmatrix}7\\4\end{bmatrix}$ of them in which $\sigma(2) = 8$, etc. In our sample permutation τ we have $\tau(3) = 8$. Hence the rank of τ is equal to $\begin{bmatrix}7\\3\end{bmatrix} + 2\begin{bmatrix}7\\4\end{bmatrix}$ plus the rank of the deleted permutation $\tau' := (16)(32)(45)(7)$ in the list of all permutations of 7 letters and 4 cycles. This leads to the following Maple program for ranking permutations by cycles. It uses the program `StirCyc`, of page 26 above.

[7] See the program notes on page 61.

```
RankPermsCycles:=proc(ss,n,k)
 local jj,tt;
 if k=n or k=0 then RETURN(0)
        elif ss[n]=n
          then RETURN(RankPermsCycles(subsop(n=NULL,ss),n-1,k-1))
          else
  member(n,ss,jj);
  tt:=ss;
  tt[jj]:=tt[n];
  tt:=[op(1..n-1,tt)];
  RETURN(RankPermsCycles(tt,n-1,k)+(jj-1)*StirCyc(n-1,k)+StirCyc(n-1,k-1))
 fi;
 end:
```

4.10 Exercises

1. Exhibit a permutation of n letters, other than the identity permutation, whose "square" is the identity. That is, find a permutation f such that $f \circ f$ is the identity. Such a permutation is called an *involution*. What do the cycles of an involution look like?

2. Suppose that a certain permutation f, of n letters, has ν_i cycles of length i, for each $i = 1, 2, \ldots$. Then how many cycles of each length does $f \circ f$ have?

3. The *order* of a permutation f in the group S_n of all permutations of n letters is the least positive integer k such that f^k is the identity permutation. Express the order of a permutation f in terms of the lengths of its cycles.

4. Describe the cycles of f^{-1} in terms of those of f.

5. How many permutations of n letters have exactly two cycles? That is, find a simple formula for $\begin{bmatrix} n \\ 2 \end{bmatrix}$.

6. Fix positive integers n and j, with $j \leq n$. Choose a random permutation of n letters. What is the probability that the element n lives in a cycle of length j? (Note that the answer depends remarkably little on j.)

4.11 Maple Programming Exercises

1. Write a Maple procedure `PermProd`, whose input is a pair of permutations f, g, of the same number of letters, given by their lists of values, and whose output

is the list of values of $f \circ g$.

2. Write a Maple procedure `InvertIt(f)` whose input is the list of values of a permutation f, and whose output is the list for f^{-1}.

3. Write a Maple procedure `MaxOrder(n)` that will use the result of exercise 3 above to find the largest possible order that a permutation of n letters can have. Run this program for $n = 2, 3, \ldots$ up to as large a value of n as you can manage, and tabulate the largest possible orders as a function of n.

4. Write a Maple program which, for given n, will print a list of the Stirling cycle numbers of order n, by using the following method. Take the generating polynomial (4.2), expand it, and print out its coefficient list. Use the Maple instructions `expand`, `product`, and `coeffs`.

5. Same as the previous problem, except instead of using the `expand` instruction, use the Maple `series` instruction. For a technical matter, you will need the instruction `convert(...,polynom)` in order to convert the output of the `series` command to the form of a polynomial, so that its `coeffs` can be extracted.

6. Write a Maple program `RandTest` that will generate 1000 random real numbers x in the range 0¡x¡1 (do not print these individual numbers), and which will output a table showing how many of those 1000 lay in the interval (0,.1), how many were in (.1,.2), ..., how many were in (.9, 1.0), Use the Maple instruction `rand()`. Do the results look random?

7. Write a Maple procedure `RandPerm` which, given a positive integer n will output a randomly chosen permutation of n letters in the form of the second line of the 2-line form of the output permutation.

8. Write a procedure `CountCycles(f)` that will take as input a list of length n that is the second line of the 2-line form of a permutation and will output the number of cycles that the input permutation has.

9. Use the above two procedures to do the following computation: Generate 1000 random permutations of $\{1, 2, 3, ..., 10\}$. For each one compute the number of cycles that it has. Print only the average number of cycles of these 1000 random permutations as well as the average value that you would expect based on our work in class.

5 Set partitions

5.1 What set partitions are

A partition of a set S is a collection $A_1, \ldots, A_k$ of nonempty subsets of S for which

(i) For all $1 \le i < j \le k$, $A_i \cap A_j = \emptyset$, and

(ii) $S = A_1 \cup A_2 \cup \ldots \cup A_k$.

The sets A_i are the *classes* of the partition.

We can write a particular set partition by using braces, each pair of which contains the elements of one of the classes A_i. Thus

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

is one way to partition the set $\{1, 2, 3, 4, 5, 6\}$ into three classes. For computer work it is often better to represent a set partition by means of its *class vector*. For a partition of $\{1, 2, \ldots, n\}$ into k classes, the class vector is of length n, and for each $i = 1, \ldots, n$, its ith entry is the class to which i belongs. The class vector of the partition displayed above, for instance, is $(1, 2, 3, 1, 1, 3)$.

5.2 Counting set partitions by classes

The number of partitions of a set of n elements into k classes is called the *Stirling set number*, and is denoted by $\left\{ {n \atop k} \right\}$. It is also called the Stirling number of the second kind. If we make a triangle which contains, in its nth row, the numbers $\left\{ {n \atop 1} \right\}, \left\{ {n \atop 2} \right\}, \ldots, \left\{ {n \atop n} \right\}$, for each $n = 1, 2, 3, \ldots$, then we will have the Pascal-like triangle that these set numbers occupy. It begins as shown in Fig. 5.3.

Look at the sums of the rows in this Stirling triangle. These are the numbers $1, 2, 5, 15, 52, \ldots$, and they are called the Bell numbers. The Bell number $b(n)$ is the number of partitions of a set of n elements. So $b(4) = 15$ means that there are 15 ways to partition a set of 4 elements. Can you write them all out?

Let's take some particular number in this array and try to visualize what it counts. Like the 6 that appears in the $(n, k) = (4, 3)$ position in the array. It says that a set of 4 things can be partitioned into 3 classes in exactly 6 ways. These 6 ways are

$$\{1, 2\}\{3\}\{4\}, \{1, 3\}\{2\}\{4\}, \{1, 4\}\{2\}\{3\}, \{2, 3\}\{1\}\{4\}, \{2, 4\}\{1\}\{3\}, \{3, 4\}\{1\}\{2\}.$$

					1							
				1		1						
			1		3		1					
		1		7		6		1				
	1		15		25		10		1			
1		31		90		65		15		1		
1	63		301		350		140		21		1	
	...				...				...			

Figure 5.3: The triangle of Stirling set numbers

5.3 East side, west side ... (III)

Fix positive integers n, k. Imagine that all of the partitions of n letters into exactly k classes have been strewn on the sidewalks and streets, all around the town.

Now take a walking tour and inspect them as they lie on the ground. If, in a certain set partition, the letter n lives in a class all by itself, then move that set partition to the east side of town, and if, on the contrary, n lives in a class with at least one other letter, then move it to the west side.

Now instead of one large collection of set partitions all over the place, we have two nice neat piles, one on the east side, and the other on the west side of town.

How many set partitions are on the east side? These are the ones in which n lives all by itself in a class of size 1. How many such set partitions are there? Well, since n lives alone, the rest of the partition must be comprised of $k-1$ classes involving the other $n-1$ letters. That means that altogether, there are $\left\{ {n-1 \atop k-1} \right\}$ set partitions that are stacked up on the east side.

How many are on the west side? These are the ones in which n lives in a class with other letters. If we remove the letter n from the class that it lives in, we will be looking at some partition of $n-1$ letters into the same number, k, of classes.

However, the converse is a bit sticky. If, conversely, we take some partition of $n-1$ letters into k classes, there are many ways in which we might insert the letter n into one of the classes. In fact, since there are k classes in the partition, and we can insert the letter n into any class, there are k ways to insert n.

That means that the number of set partitions on the west side is k times the number of set partitions of $n-1$ letters into k classes, i.e., it is $k\left\{ {n-1 \atop k} \right\}$.

Therefore the number of all set partitions of n letters that have k classes must be equal to $\left\{{n-1 \atop k-1}\right\} + k\left\{{n-1 \atop k}\right\}$ since every such partition landed on one side of town or the other.

Theorem 5.1 *The Stirling set numbers* $\left\{{n \atop k}\right\}$ *satisfy the Pascal-triangle-like recurrence*

$$\left\{{n \atop k}\right\} = \left\{{n-1 \atop k-1}\right\} + k\left\{{n-1 \atop k}\right\}. \tag{5.1}$$

It is important to observe that numbers that are defined recursively can be calculated recursively, i.e., with a computer program that calls itself. This can be done only in a recursion-capable computer language. In Maple, a procedure that will calculate $\left\{{n \atop k}\right\}$ recursively is as follows.

```
StirSet:=proc(n,k)
  options remember:
#Returns the Stirling number of the 2nd kind
if n<1 then RETURN(0)
       else
         if n=1 then
                    if k<>1 then RETURN(0)
                        else RETURN(1)
                    fi
                else
                    RETURN(StirSet(n-1,k-1)+k*StirSet(n-1,k))
         fi:
fi:
end:
```

5.4 The generating function

If we look for the generating function of the Stirling set numbers $\left\{{n \atop k}\right\}$, we might be tempted to multiply both sides of (5.1) by x^k and sum over all integer k, as we did in the case of the cycle numbers. Here, though, we would be doomed to failure, because of the appearance of the "k" on the right side of (5.1), which would interfere with that operation mightily.

Instead, let's define the generating functions $f_k(x) = \sum_n \left\{{n \atop k}\right\} x^n$, and we *will* succeed in finding them explicitly. To do that, multiply (5.1) on both sides by x^n and sum on n. The result is that $f_k(x) = x f_{k-1}(x) + kx f_k(x)$, i.e., $f_k(x) = (x/(1-kx)) f_{k-1}(x)$. Since $f_1(x) = x/(1-x)$ (why?), we have proved the following result.

Theorem 5.2 *The Stirling set number* $\left\{ {n \atop k} \right\}$ *is the coefficient of* x^n *in the generating function* $x^k/((1-x)(1-2x)(1-3x)\dots(1-kx))$. *That is, we have*

$$\sum_{n\geq 0} \left\{ {n \atop k} \right\} x^n = \frac{x^k}{(1-x)(1-2x)(1-3x)\dots(1-kx)} \qquad (k = 1, 2, 3, \dots) \tag{5.2}$$

5.5 An application

Suppose that there are d different kinds of coupons, and that we want to collect at least one specimen of each kind. We embark on a series of trials, in each of which we choose, uniformly at random, one of the coupons and add it to our collection. What is the probability that at the nth trial we find, for the first time, that we have at least one specimen of all of the d kinds of coupons?

Let $p(n, d)$ denote the required answer.

If at the nth trial we have a complete collection for the first time, then suppose we happened to have chosen coupon i at that nth trial. Then i was the only missing coupon from the results of the first $n-1$ trials. That means that the first $n-1$ trials had at least one specimen of each of the coupons $1, 2, \dots, i-1, i+1, \dots d$, and that the missing type i was then chosen.

For each $j = 1, \dots, n$, let $a[j]$ denote the number of the coupon that was drawn on the jth trial. For $n = 10$, $d = 4$, $i = 3$, for instance, the array $(a[1], a[2], \dots, a[10])$ might look like

$$(1, 4, 4, 1, 2, 4, 4, 2, 1, 3).$$

We observe that this array a describes ('encodes') a certain set partition of the set $\{1, 2, \dots, n-1\}$ into $d-1$ classes. Indeed, the class to which the letter j belongs is $a[j]$. The example array above describes a partition of the set $\{1, 2, \dots, 9\}$ into 3 classes, namely the partition $\{1, 4, 9\}, \{5, 8\}, \{2, 3, 6, 7\}$.

But there's a little difference that's important. The classes of the partition are *ordered*. That is, the partition $\{1, 4, 9\}, \{5, 8\}, \{2, 3, 6, 7\}$ is counted as a different partition from the partition $\{5, 8\}, \{2, 3, 6, 7\}, \{1, 4, 9\}$, for example. That's because the first of these two partitions comes from the array $(1, 4, 4, 1, 2, 4, 4, 2, 1, 3)$, as we saw, while the second comes from the array $(4, 2, 2, 4, 1, 2, 2, 1, 4, 3)$. The number of partitions with ordered classes that correspond to a single partition into k classes is $k!$, since there are that many ways to rearrange the classes.

Thus we can describe the sequence of trials of a coupon collector who succeeded on the nth trial in getting a complete collection of coupons, by giving the vector $\mathbf{a}$ for the sequence of coupon numbers that were drawn on successive trials, and that

can be done in $(d-1)!\left\{{n-1 \atop d-1}\right\}$ ways for each choice of the index i that completes the sequence, for a total of $d!\left\{{n-1 \atop d-1}\right\}$ ways altogether.

To return to the *probability* that n trials will be needed, that is the number of possible vectors $\mathbf{a}$ that correspond to n trials, divided by the total number of possible outcomes on n trials, which is d^n, that is, the probability that exactly n trials are needed is

$$p_n = \frac{d!}{d^n}\left\{{n-1 \atop d-1}\right\}.$$

The probability generating function is

$$f(x) = \sum_n p_n x^n = \sum_n \frac{d!}{d^n}\left\{{n-1 \atop d-1}\right\}x^n.$$

If we refer to (5.2) the above can be rewritten as

$$\begin{aligned} f(x) &= \sum_n \frac{d!}{d^n}\left\{{n-1 \atop d-1}\right\}x^n = d!\frac{x}{d}\sum_n \left\{{n-1 \atop d-1}\right\}\left(\frac{x}{d}\right)^{n-1} \\ &= (d-1)!x\frac{\left(\frac{x}{d}\right)^{d-1}}{(1-\frac{x}{d})(1-2\frac{x}{d})\ldots(1-(d-1)\frac{x}{d})} \\ &= \frac{(d-1)!x^d}{(d-x)(d-2x)(d-3x)\ldots(d-(d-1)x)} \end{aligned}$$

We note in passing that $f(1) = 1$, as must be true for probability generating functions.

Theorem 5.3 *The probability that exactly n trials will be needed to complete a collection of d different coupons is the coefficient of x^n in the power series for the function*

$$f(x) = \frac{(d-1)!x^d}{(d-x)(d-2x)(d-3x)\ldots(d-(d-1)x)}.$$

The *average* number of trials will be $f'(1)$, but this is $\left(\frac{d}{dx}\log f(x)\right)_{x=1}$, or

$$\begin{aligned} &\left(\frac{d}{dx}\left(\log(d-1)! + d\log x - \log(d-x) - \ldots - \log(d-(d-1)x)\right)\right)_{x=1} \\ = \;& d + \frac{1}{d-1} + \frac{2}{d-2} + \frac{3}{d-3} + \ldots + \frac{d-1}{1} \end{aligned}$$

$$= \quad d + \sum_{j=1}^{d-1} \frac{j}{d-j} = d + \sum_{j=1}^{d-1} \frac{(j-d)+d}{d-j}$$

$$= \quad d - (d-1) + d \sum_{j=1}^{d-1} \frac{1}{d-j} = 1 + d(H_d - \frac{1}{d}) = dH_d$$

Theorem 5.4 *The average number of trials that are needed to complete a collection of d kinds of coupons is dH_d, where H_d is the harmonic number.*

5.6 Making lists and random choices of set partitions

To make a list $L(n,k)$ of all partitions of $\{1,2,\ldots,n\}$ into k classes, we do the following. Take the list $L(n-1,k-1)$, and to each partition on that list adjoin a new class consisting solely of the letter n. That will list the portion of $L(n,k)$ that comes from the east side. Next, take the list $L(n-1,k)$, and for each partition Π on that list create k new partitions, namely those that are obtained by inserting the letter n as a new member of each of the k classes, in turn, of Π.

A good data structure for representing a set partition in a computer is the *class vector* of the partition. The class vector of a partition of the set $\{1,2,\ldots,n\}$ into k classes is the n-vector $(a_1,a_2,\ldots,a_n)$ in which for each i, a_i is the class to which i belongs. Thus the partition $(1,7)(2,3,5)(4,6)$ of $\{1,2,\ldots,7\}$ into three classes is represented by the class vector $(1,2,2,3,2,3,1)$.

To adjoin a new class consisting solely of the letter n, we take the class vector and adjoin one new entry to it, namely the entry $a_n = k$. To insert n into one of the already existing k classes, we take the class vector and adjoin one new entry to it, namely the entry $a_n = i$, if i is the index of the class into which we are inserting the letter n.

A Maple procedure[8] for making a list of all partitions of $\{1,2,\ldots,n\}$ into k classes is as follows.

[8] See the program notes on page 61.

```
EWop:=(y,m)->[op(y),m]:
ListSetPtns:=proc(n,k)
local east,west,i,out ; options remember;
#Lists all partitions of the set 1,..,n into k classes
#Output array[i] is the class to which i belongs.
if n=1 then
   if k<>1 then RETURN([]) else RETURN([[1]]) fi:
       else
east:=ListSetPtns(n-1,k-1): west:=ListSetPtns(n-1,k):
out:=map(EWop,east,k);
for i from 1 to k do out:=[op(out),op(map(EWop,west,i))] od;
RETURN(out);
fi;
end:
```

A procedure that will select a partition of the set $\{1, 2, \ldots, n\}$ with k classes uar is also easy to do recursively. With probability $\left\{{n-1 \atop k-1}\right\}/\left\{{n \atop k}\right\}$ we will choose uar a partition of $\{1, \ldots, n-1\}$ into $k-1$ classes and then insert the letter n as a new singleton class. With probability $1 - \left\{{n-1 \atop k-1}\right\}/\left\{{n \atop k}\right\}$, we will choose uar a partition of $\{1, \ldots, n-1\}$ into k classes and then insert the letter n into a randomly chosen one of those k classes. The Maple program[9] follows.

```
RandSetPtns:=proc(n,k)
local rno,class ;
#Returns a random partition of 1..n into k classes
if n=1 then
   if k<>1 then RETURN([]) else RETURN([1]) fi;
       else
           rno:=10^(-12)*rand();# Choose a random real number in (0,1)
           if rno<StirSet(n-1,k-1)/StirSet(n,k)
                then #we go to the east side
                   RETURN([op(RandSetPtns(n-1,k-1)),k])
                else #we go to the west side
                   class:=rand(1..k);
                   RETURN([op(RandSetPtns(n-1,k)),class()])
           fi;
   fi;
end:
```

[9] See the program notes on page 62.

5.7 Ranking set partitions

What is the rank of the partition $\Pi := \{1,4\}\{2,7,8\}\{3\}\{5,6\}$ of the set $\{1,2,\ldots,8\}$ into 4 classes in the list of all partitions of that set into 4 classes?

Our list of partitions puts those set partitions in which the highest letter n lives in a singleton class first. Following those are the partitions in which n lives, not alone, in class 1, then those in which n is, not alone, in class 2, etc., where the classes are numbered in order of their smallest element. For this example partition Π, 8 is not in a singleton class, so it is preceded in the list by all of the $\left\{ {7 \atop 3} \right\}$ partitions in which 8 does live in a singleton class. Further, 8 does not live in the first class of Π, so Π is also preceded by all $\left\{ {7 \atop 4} \right\}$ partitions that are obtained by inserting 8 into the *first* class of some partition of 7 letters into 4 classes.

Thus, if Π' denotes the partition obtained from Π by deleting 8, then

$$\mathrm{rank}(\Pi) = \left\{ {7 \atop 3} \right\} + \left\{ {7 \atop 4} \right\} + \mathrm{rank}(\Pi'),$$

which is the idea of the recursive algorithm. The full Maple program for ranking set partitions is as follows.

5.8 Exercises

1. Find a simple formula for the number of partitions of a set of n elements into 2 classes, i.e., for the Stirling set number $\left\{ {n \atop 2} \right\}$.

2. In how many partitions of a set of n elements does the element n live in a class of size j? Express your answer in terms of the Bell numbers.

3. Write a Maple program that will carry out the Coupon Collector's Test on random numbers, as described in section 5.5. Specifically, perform the following experiment 1000 times: *Choose integers at random between 1 and 10, inclusive, until for the first time you have a complete collection of all 10 of them. Record the number of integers that you had to choose in order to get this complete collection.* After doing the 1000 experiments, print out just one number: the average of the 1000 numbers that you recorded. This output number should be around 29, if Maple's random numbers are good ones. Your program should be quite short. Try to be sneaky and clever in the way you write it.

6 Integer partitions

6.1 What they are

A partition of an integer n is a representation of n as a sum of positive integers, the order of the latter being immaterial. The partitions of 5 are 5, $4+1$, $3+2$, $3+1+1$, $2+2+1$, $2+1+1+1$, and $1+1+1+1+1$, so we can partition the integer 5 in 7 ways. If $p(n)$ denotes the number of partitions of the integer n, then the sequence of values of $p(n)$, for $n = 0, 1, 2, \ldots$, begins as

$$\{1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56, 77, 101, 135, 176, \ldots\}.$$

Generically, we will write a partition as $n = a_1 + a_2 + \ldots + a_r$, where $a_1 \geq a_2 \geq \ldots \geq a_r \geq 1$. The numbers $a_i (i = 1, \ldots, r)$ are the *parts* of the partition. So we will standardize by presenting a partition with its parts in nonascending order.

Let $p(n, k)$ denote the number of partitions of the integer n whose largest part is k. For example, $p(4, 2) = 2$ since there are two partitions of 4 whose largest part is 2, viz., $4 = 2 + 2$ and $4 = 2 + 1 + 1$. The table of values of $p(n, k)$, with rows labeled by $n = 1, 2, \ldots$, looks like this.

					1					
				1		1				
			1		1		1			
		1		2		1		1		
	1		2		2		1		1	
1		3		3		2		1		1
1		3		4		3		2		1		1

The suspense is geting unbearable. Could it be that we have yet another east side-west side situation developing here?

6.2 East side, west side ... (IV)

Fix positive integers n, k. Imagine that all of the partitions of the integer n whose largest part is k have been strewn on the sidewalks and streets, all around the town.

Now take a walking tour and inspect them as they lie on the ground. If, in a certain partition of the integer n whose largest part is k, the part k occurs only once in the partition, then move that partition of n to the east side of town, and if, on the contrary, the part k is used more than once, then move the partition to the west side.

Now instead of one large collection of partitions of the integer n all over the place, we have two nice neat piles, one on the east side, and the other on the west side of town.

How many partitions of n are on the east side? These are the ones in which k occurs just once as a part. So the east side partitions look like this-

$$n = k + (\text{a part } \leq k-1) + (\text{a part } \leq k-1) + \ldots + (\text{a part } \leq k-1).$$

How many such integer partitions are there? Subtract 1 from the left side of the above equation, and from the right side subtract 1 from the part k. We now are looking at a partition of the integer $n-1$ whose largest part is $k-1$. Conversely, given such a partition of $n-1$ we can add 1 to a part $k-1$, thereby geting a partition of n whose largest part is k and in which that largest part k occurs just once, i.e., an east side partition of n.

That means that altogether, there are $p(n-1, k-1)$ partitions that are stacked up on the east side.

How many are on the west side? These are the ones in which the part k is repeated, and they look like

$$n = k + k + (\text{a part } \leq k) + (\text{a part } \leq k) + \ldots + (\text{a part } \leq k).$$

If we subtract k from the left side, and delete one of the k's from the right, we will be looking at some partition of $n-k$ whose largest part is k. If, conversely, we take some partition of $n-k$ whose largest part is k, and adjoin a new part k to that partition, we will have a partition of n whose largest part is k, and in which k is a repeated part, i.e., we will have a west side partition of n.

That means that altogether, there are $p(n-k, k)$ partitions that are stacked up on the west side. It follows that $p(n,k) = p(n-1, k-1) + p(n-k, k)$, since every partition of n landed on one side of town or the other.

Theorem 6.1 *The number $p(n,k)$ of partitions of n whose largest part is k, satisfies the "Pascal triangle" recurrence*

$$p(n,k) = p(n-1, k-1) + p(n-k, k) \tag{6.1}$$

with the initial values $p(n,k) = 0$ if $k < 1$ or $k > n$ or $n \leq 0$, and $p(0,0) = 1$.

A Maple program[10] that will list all partitions of the integer n whose largest part is k follows.

[10] See the program notes on page 62.

```
eastop:=y->applyop(t->t+1,1,y):
westop:=(y,j)->[j,op(y)]:
ListKPtns:=proc(n,k)
local east, west ; options remember;
# Lists the partitions of n whose largest part is k
if n<=0 or k<=0 or k>n then RETURN([])
       elif n=1 then RETURN([[1]])
                        else
east:=ListKPtns(n-1,k-1);
west:=ListKPtns(n-k,k);
RETURN([op(map(westop,west,k)),op(map(eastop,east))]);
fi;
end:
```

6.3 Exercises

1. Write a recursive Maple procedure that will compute the numbers $p(n,k)$. Print out a table of these numbers for $n = 1, \ldots, 10$ and $1 \leq k \leq n$.

2. Write a recursive Maple procedure that will make a list of all partitions of a given integer n into k parts. Run your program with $n = 8$ and $k = 4$.

3. Write a recursive Maple procedure that will choose, uniformly at random, a partition of n whose largest part is k. Run your program fifty times with $n = 6$, $k = 3$.

4. Let $\tilde{p}(n,k)$ denote the number of partitions of n into k parts. Show that $\tilde{p}(n,k) = p(n,k)$ as follows:

   (a) Show that $\tilde{p}(n,k)$ satisfies the same recurrence as $p(n,k)$, using the east side-west side method. You will of course have to find the correct criterion for which partitions get thrown onto which of the two piles.

   (b) Then show that $\tilde{p}(n,k)$ and $p(n,k)$ satisfy the same set of initial values, these values being sufficient to determine the two sequences uniquely.

   (c) This quaint method implies a certain direct bijection between the sets of partitions of n that have k parts and those whose largest part is k. Find this bijection.

7 . . . and all around the town

We have now encountered a fairly large number of combinatorial families, and their recursive structures are quite similar in a number of respects. So what we'd like to do now is to abstract from these examples their common characteristics and discuss them at a greater level of generality. We'll see that the view from a higher perch is a lot nicer; the details recede into the distance, while the fundamental structural properties of these families will be etched more clearly.

Let's focus for a moment on the example of k-subsets of an n-set. There are $\binom{n}{k}$ of them, and recall the steps that were involved in counting them. Let's denote by $B(n,k)$ the collection of *all* k-subsets of an n-set.

1. First we designate some of these sets as east side sets and the rest of them as the west side. Specifically, those sets that do contain the highest letter n are put on the west side and those that do not are on the east side. Let $B_e(n,k)$ (resp. $B_w(n,k)$) denote the collection of all east side (resp. west side) k-subsets of an n-set.

2. Next we took the west side sets, the ones that do contain n, and identified them with *all* of the $(k-1)$-subsets of $n-1$ things. What does that word "identified" mean? It means that we created a $1-1$ mapping, f_w, say, between the west side k-subsets of an n-set and all $(k-1)$-subsets of an $n-1$ set. In the usual mathematical notation, we have

   $$f_w : B_w(n,k) \to B(n-1,k-1).$$

   And what, precisely, is that mapping f_w? That is, if we are given a west side set $S \in B_w(n,k)$, which $(k-1)$-subset of an $(n-1)$-set do we use for $f_w(S)$? Easy. Delete the element n from S (why are we so sure that $n \in S$?), and what remains is $f_w(S)$.

3. Onwards to the east side. We took the east side sets, the ones that do not contain n, and identified them with all of the k-subsets of an $(n-1)$-set. What does the word "identified" mean now? It means that we created a $1-1$ mapping, f_e, say, between the east side k-subsets of an n-set and all k-subsets of an $(n-1)$-set. In the usual mathematical notation, we have

   $$f_e : B_e(n,k) \to B(n-1,k).$$

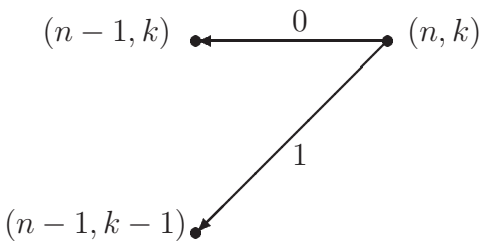

Figure 7.4: The neighborhood of (n,k)

And what, precisely, is that mapping f_e? That is, if we are given a west side set $S \in B_e(n,k)$, which k-subset of an $(n-1)$-set do we use for $f_e(S)$? Easier. Use S itself. It doesn't contain n, therefore it belongs to $B(n-1,k)$.

4. The mappings f_e, f_w are both invertible. So we obtained the recurrence $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. And the recurrence was constructively proved, by means of the pair of mappings.

Let's draw a little picture of all of that, as shown in Fig. 7.4.

In the diagram the directed edge from (n,k) to $(n-1,k)$ represents the map f_e, while that from (n,k) to $(n-1,k-1)$ stands for the map f_w. In this way we can connect each point (n,k) to the two points $(n-1,k-1)$ and $(n,k-1)$, except that points (n,n) can be connected only to $(n-1,n-1)$ and points $(n,0)$ only to $(n-1,0)$. We will label the edges that are outbound from each vertex consecutively. If there are two outbound edges from (n,k) then we label the westbound edge with 0 and the southwestbound edge with 1. If there is only one outbound edge then we label it 0.

If we were to make all of these connections at once we would find ourselves looking at a graph as shown in Fig. 7.5. This directed graph ("digraph"), and its generalizations, has a number of useful properties that will enable us to perform certain standard algorithms on the family of k-subsets, and on all of the east-west families that we have discussed, in a unified way, and quite straightforwardly.

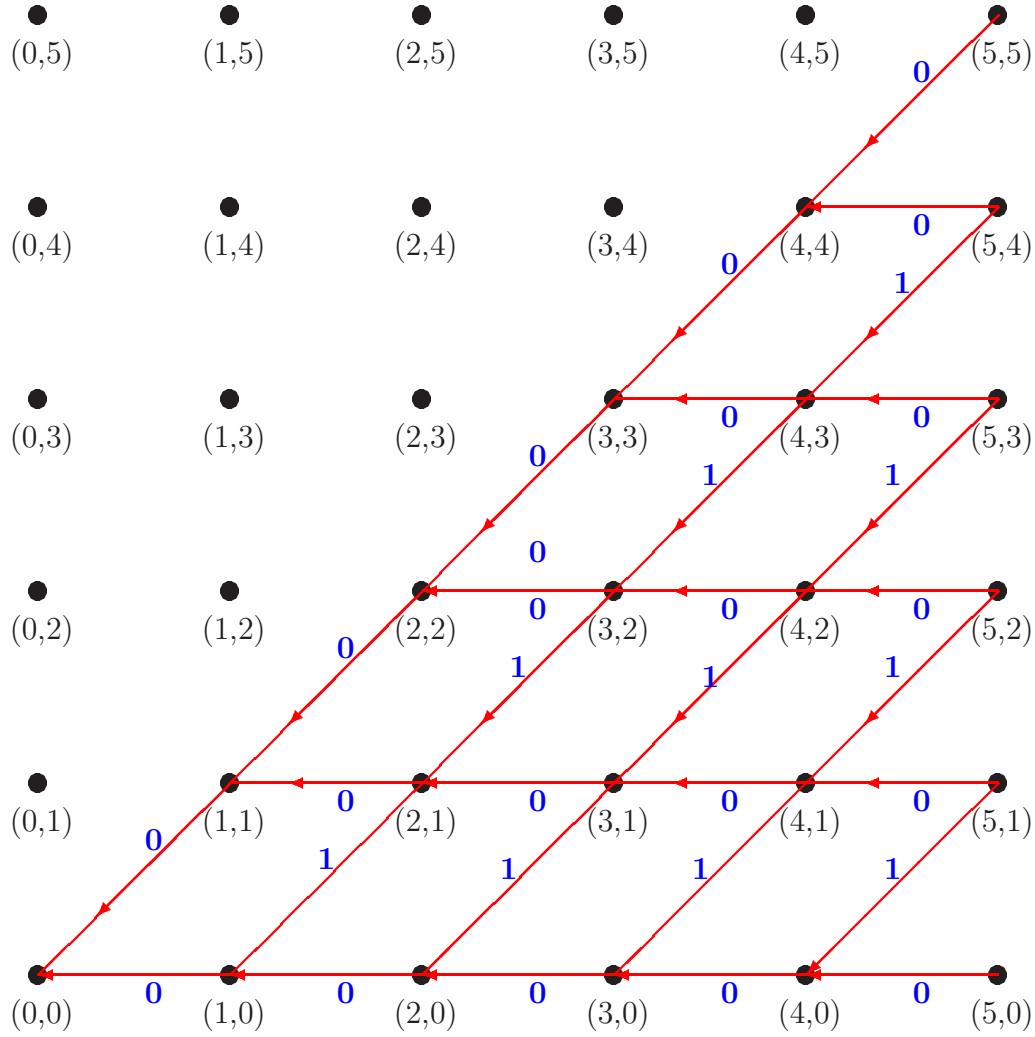

Figure 7.5: The digraph for the k-subsets of an n-set, showing the $0, 1$ labels on the edges

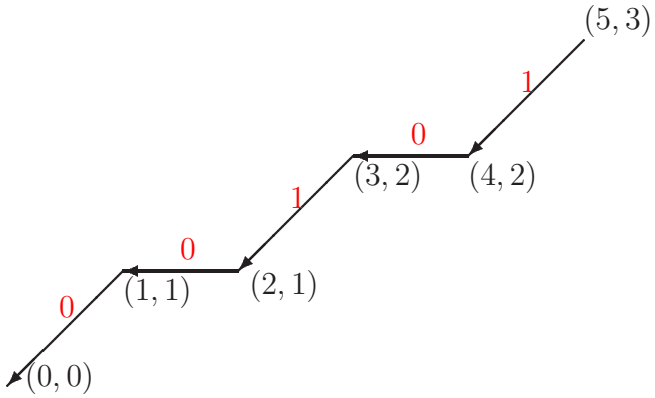

Figure 7.6: Portrait of the subset $\{1, 3, 5\}$

7.1 k-subsets and codewords

The most important single feature of the digraph shown in Fig. 7.5 is the following: *there is a 1-1 correspondence between k subsets of an n-set, on the one hand, and walks in the digraph beginning at the point (n, k) and ending at the origin.* That is, for instance, each 3-subset of $\{1, 2, 3, 4, 5\}$ corresponds to a certain walk from the point $(5, 3)$ to the origin. We might say that the walk *encodes* the subset.

To see what this means, consider the walk from the point $(5, 3)$ to the origin that is shown in Fig. 7.6. We claim that this walk corresponds to (*encodes, represents, is*) the 3-subset $\{1, 3, 5\}$ of the set $\{1, 2, 3, 4, 5\}$.

Indeed, the first step in the walk goes from $(5, 3)$ to $(4, 2)$. Since both the n and the k coordinates changed, we must have chosen the element $n = 5$ to live in our subset. The second step goes from $(4, 2)$ to $(3, 2)$. On that step, the n-coordinate changed, but the k-coordinate did not. So we still are looking for the same number, 2, of elements in a shrunken universe, viz. $\{1, 2, 3\}$. On the next step, to $(2, 1)$, both n and k changed, so we did choose the element 3 to live in our subset. The step $(2, 1) \to (1, 1)$ is again horizontal, so we do not choose 2 to add to our subset, and on the last step we do choose 1. Hence the subset ends up as $\{1, 3, 5\}$.

In general, if we have a walk from a point (n, k) to $(0, 0)$ we can discover the k-subset of $\{1, 2, \ldots, n\}$ that it corresponds to as follows.

- Begin with the empty set S, and follow the given walk to the origin.
- Each time you encounter a diagonal step (as opposed to a horizontal step), say from (n', k') to $(n' - 1, k' - 1)$, adjoin the element n' to the set S.
- When you reach the origin, output the set S.

As we walk from (n, k) to the origin, we can write down the numbers of the edges that we encounter. In the walk of Fig. 7.6, we use first edge 1, then 0, 1, 0 and 0, in that order. Thus the walk generates a *codeword*, in this case 10100.

Conversely, if we are given the codeword and (n, k), then we can reconstruct the walk. If we have 10100, and we begin at $(5, 3)$, for example, the codeword tells us which road we must follow out of each vertex that we encounter.

Thus, the list of all k-subsets of n elements corresponds to a list of $\binom{n}{k}$ codewords of n 0's and 1's. The 10 3-subsets of $\{1, 2, 3, 4, 5\}$ correspond to the list

$$00000, 01000, 01100, 01110, 10000, 10100, 10110, 11000, 11010, 11100$$

of codewords. Note that these are arranged in lexicographic order, considered as 5 letter words over an alphabet of two letters $\{0, 1\}$.

It is easy to design a recursive program that will list, in lexicographic order, all of the codewords that correspond to walks that originate at a fixed vertex (n, k). Let `ListWalks(n,k)` denote a call to such a program. Then, aside from consideration of the base cases, what we need to do is this-

1. Recursively, obtain the list `ListWalks(n-1,k)`, and prepend to each codeword on that list a "0", and then,

2. recursively obtain the list `ListWalks(n-1,k-1)`, and prepend to each codeword on that list a "1".

A Maple program that will do this is as follows.

```
prepend:=(arry,j)->[j,op(arry)]:
ListWalks:=proc(n,k)
 local yy,zz ;
 if k>n or k<0 then RETURN([])
                    elif
        n=k or k=0 then RETURN([[seq(0,j=1..n)]])
                    else
        yy:=map(prepend,ListWalks(n-1,k),0);
        zz:=map(prepend,ListWalks(n-1,k-1),1);
                     fi;
 RETURN([op(yy),op(zz)]);
 end:
```

To test the program, we call `ListWalks(5,3);` and the following output results:

```
[[0,0,0,0,0],[0,1,0,0,0],[0,1,1,0,0],[0,1,1,1,0],[1,0,0,0,0],
 [1,0,1,0,0],[1,0,1,1,0],[1,1,0,0,0],[1,1,0,1,0],[1,1,1,0,0]]
```

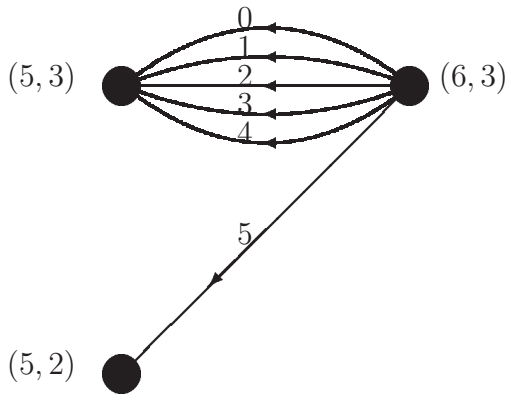

Figure 7.7: The neighborhood of (n, k) $\quad$ ($(6, 3)$ is shown)

7.2 A look at one more family

Here we'll have a brief look at the family of n-permutations with k-cycles. To draw the directed graph that is associated with this family we begin by recalling the recurrence relation (4.1) that these permutations satisfy,

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}. \tag{7.1}$$

This tells us that the grid point (n, k) will connect to points $(n-1, k-1)$ and $(n-1, k)$, but because of the factor of $(n-1)$, there will have to be $(n-1)$ directed edges from (n, k) to $(n-1, k)$. If we number those edges $0, 1, \ldots, n-2$, then the environment of a typical vertex (n, k) is as shown in Fig. 7.7. When we assemble these neighborhoods of all vertices into a single directed graph, we obtain Fig. 7.8.

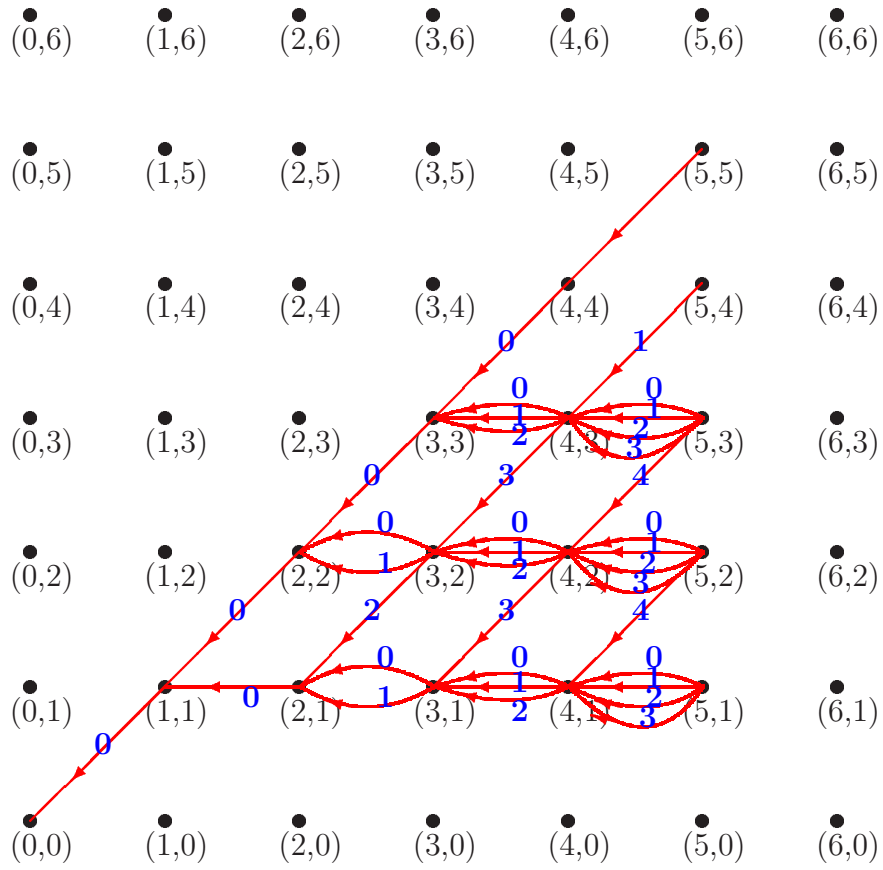

Figure 7.8: Permutations and their cycles (showing edge numbering)

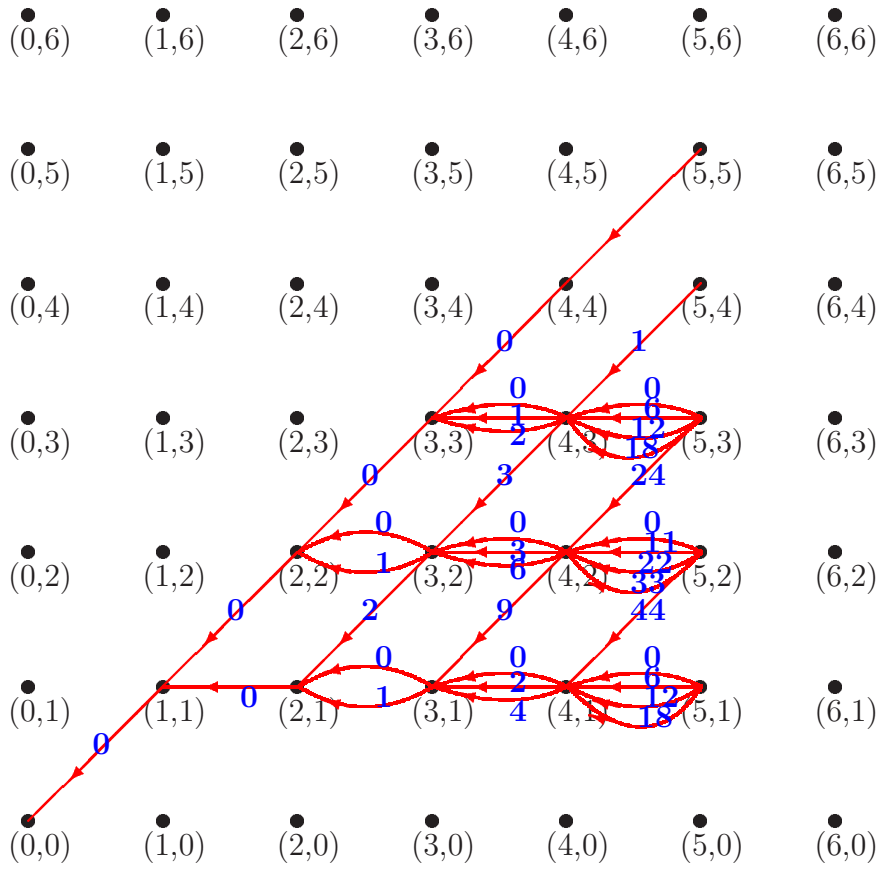

Figure 7.9: Permutations and their cycles (showing toll charges on the roads)

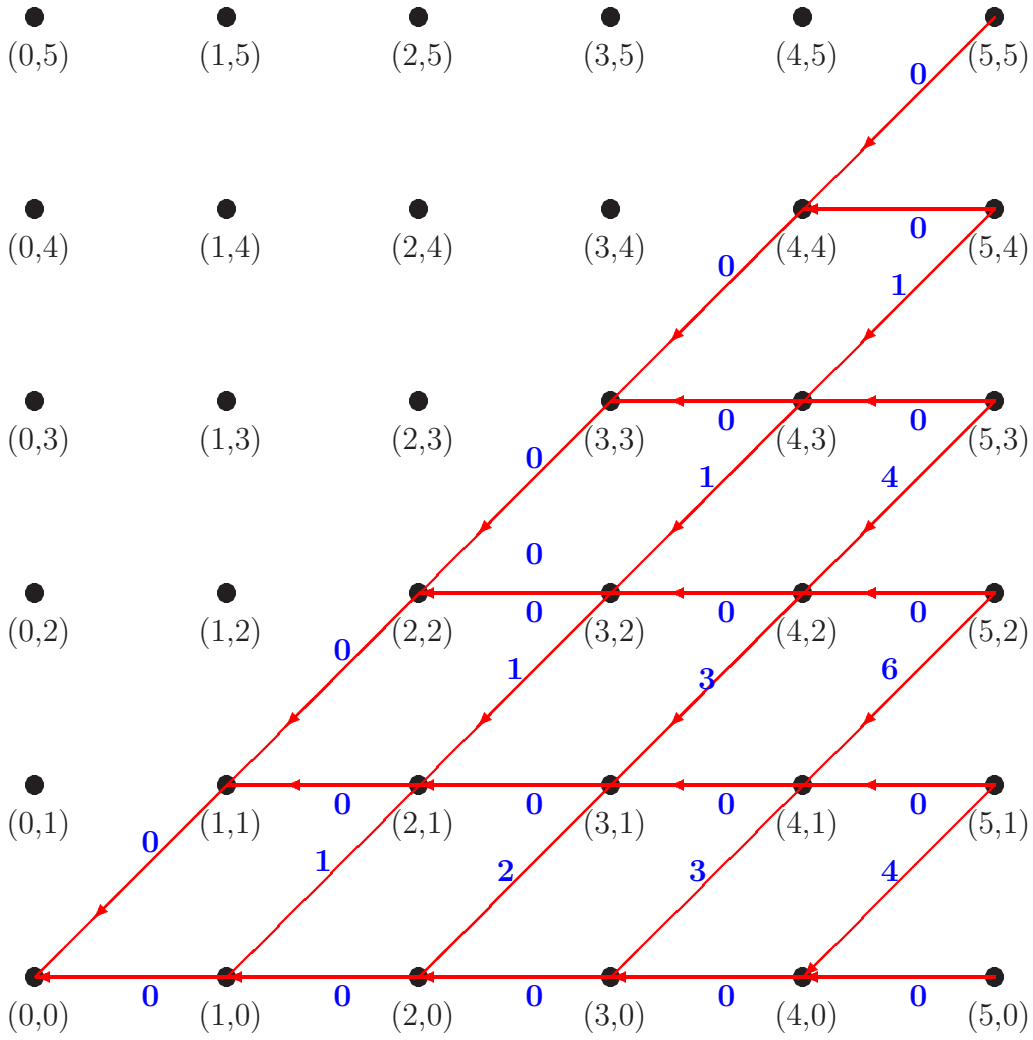

Figure 7.10: The k-subsets of an n-set, showing toll charges on the roads

8 The EastWest Maple package

In this section we describe the package EastWest, which is available from

<http://www.cis.upenn.edu/∼wilf>

in the form of a binary file `EastWest.m`. To use this file, open a new Maple worksheet, and enter

```
>read `[your path]/EastWest.m`;
```

Then enter

```
>hello();
```

if you want to see a printout of operating instructions.

The purpose of this package is to perform any one of six functions on any one of seven combinatorial families. The seven families are as follows.

1. k-subsets of an n-set. This family is encoded by member lists. That is to say, a particular k subset of $\{1, \ldots, n\}$ is represented by a Maple list of the elements of the subset. Thus `[1,3,4]` is the 3-subset whose elements are 1, 3, and 4.

2. permutations of n letters with k cycles. A permutation in this family is represented by its list of values, i.e., by the second line of its two-line form. Thus `[1,4,3,2]` is the permutation σ on 4 letters with 3 cycles for which $\sigma(1) = 1$, $\sigma(2) = 4$, $\sigma(3) = 3$, $\sigma(4) = 2$,

3. Partitions of a set of n elements with k classes. A member of this family is represented by its class vector array, which is an array of length n whose ith entry is the number of the class to which i belongs. The classes are arranged with their smallest elements in increasing order. Thus the partition $(1, 3, 5)(4)(2)$ is encoded by the vector `[1,2,1,3,1]`.

4. Partitions of the integer n into k parts. An integer partition is encoded by its array of parts, listed in nonincreasing order. Thus the partition $6 = 2+2+1+1$ is represented by the array `[2,2,1,1]`.

5. Ordered partitions of the integer n into k nonnegative parts. These are encoded by their arrays of parts. The ordered partition $6 = 2 + 0 + 0 + 1 + 3$ of 6 into 5 parts is represented by the list `[2,0,0,1,3]`.

6. Permutations of n letters with k runs up. A "run up" in a permutation is a maximal sequence of consecutive increasing values of the permutation. The

permutation whose values are `[1,4,3,6,8,5,2,7]` has 4 runs up, namely $\{1, 4\}$, $\{3, 6, 8\}$, $\{5\}$, $\{2, 7\}$. These permutations are also encoded by their arrays of values.

7. Partitions of the integer n whose largest part is k, encoded by their arrays of parts

The six functions are

1. `Count` counts the members of the specified family and size. Call it with

   ```
   >Count(famno,n,k);
   ```
   where `famno` is the number of the family, and `n,k` are the size parameters.

2. `List` returns a list of all objects of the specified size, in the standard encoding for the family. Call with

   ```
   >List(famno, n, k);
   ```

3. `Random` returns a single object, chosen uniformly at random from the members of the family of the given size. Call with

   ```
   >Random(famno, n, k);
   ```

4. `Rank` returns the rank of the input object on the list of all members of the family of the size of the input object. Call with

   ```
   >Rank(famno, object);
   ```

5. `Unrank` returns the member of the family of given rank in the list of all members of the family of given size. It is called as

   ```
   >Unrank(famno,n,k,object);
   ```

6. `Successor` returns the object that immediately follows the given object on the list of all members of the family of the given size. Called by

```
>Successor(famno,n,k,object)
```

Here are some samples of the package in action.

- *How many permutations of 8 letters have exactly 3 runs up?* We're in family # 6, and we want the `Count` function. The Maple dialogue is

  ```
  >Count(6,8,3);
  ```

  ```
  4293
  ```

  so there are 4293 such permutations.

- *How many partitions of the integer 30 have largest part 8?* Now we're in family # 7, so we type
  `>Count(7,30,8);`
  to which Maple replies

  ```
  638
  ```

  .

- *List all partitions of the set* $\{1,2,3,4,5\}$ *into 2 classes.* For this we enter
  `>List(3,5,2);`
  which gets the answer

  ```
  [[1, 1, 1, 1, 2], [1, 1, 1, 2, 1], [1, 1, 2, 1, 1], [1, 2, 1,
  1, 1], [1, 2, 2, 1, 1], [1, 1, 2, 2, 1], [1, 2, 1, 2, 1], [1,
   2, 2, 2, 1], [1, 1, 1, 2, 2], [1, 1, 2, 1, 2], [1, 2, 1, 1,
  2], [1, 2, 2, 1, 2], [1, 1, 2, 2, 2], [1, 2, 1, 2, 2], [1, 2,
                              2, 2, 2]]
  ```

- *In the list of all partitions of the set* $\{1,2,3,4,5\}$ *into 2 classes, what is the successor of the partition* `[1,1,2,2,1]`*?* For this we enter

  ```
  >Successor(3,5,2,[1,1,2,2,1]);
  ```

  and Maple says

  ```
  [1,2,1,2,1]
  ```

9 Program notes

Program `Subsets`, on page 14, lists all of the subsets of an n-set.

line 1: First we define the function `GlueIt`. This enlarges a given array `z` by inserting one new entry, `d`, in as a new first entry.

line 3: In `Subsets` itself, the `options remember;` phrase causes Maple to remember the output of `Subsets` each time it is called. That's very helpful in a recursive routine since on a new call, Maple will not have to go all the way back to the beginning each time. Instead it needs to go back only far enough to meet some quantity that it computed on an earlier call. Thus it gets smarter every time you use it during a single session.

line 6: The instruction `east:=...` takes the array of all subsets of $n-1$, and glues onto each one a new first entry equal to 0.

line 7: The line `west:=...` takes the array of all subsets of $n-1$, and glues onto each one a new first entry equal to 1.

line 8: The `RETURN` line returns as output the single array that one gets by concatenating `east` and `west`.

Program `RandSub`, on page 15, chooses at random a subset of $\{1, 2, \ldots, n\}$.

line 3: The `rand` instruction in Maple returns a program to choose random numbers. It does not choose a random number. Thus `rn` is defined to be a program which when called will return a random integer that will be either 0 or 1 with equal probability.

line 4: We initialize the output set `set` to be the empty set.

line 5: we successively augment `set` by adjoining to it a randomly chosen 0 or 1.

1. A question for you. This is a recursive routine. So why isn't there an `options remember;` phrase in it? If this puzzles you, put such a phrase in, and run the resulting program a few times until you see what the problem is.

Program `ListKSubsets`, on page 16, lists all of the k-subsets of an n-set.

line 1: We define the operation `westop`. It takes two arguments, an array `x` and an integer `m`. It adjoins to the end of the array `x` a new entry equal to `m`.

line 8: After attending to the base cases, the general construction appears. Recursively we call the lists for `(n-1,k)` and for `(n-1,k-1)`, calling them `east` and `west`, respectively.

line 9: We need to adjoin a new element, namely `n` to each set on the list `west`. We do this by mapping the operation `westop`, with arguments `west` and `n`, over all of the elements of the array `west`.

line 10: Finally, we `RETURN` the concatenation of the two lists `east` and `west`.

Program `RandomKSubsets`, on page 16, chooses, uniformly at random, a k-subset of `[1..n]`.

line 6: After doing the base cases, we create an instruction `rno` that will, when called, produce a random number in the range $(0, 1)$.

line 7: We want to do something with probability k/n and something else otherwise. So we ask if a random number is less than k/n.

line 8: If it is then we want to adjoin `n` to a randomly chosen $k-1$ subset of $\{1, \ldots, n-1\}$. So we choose one of those, recursively, and call it `east`. Then we `RETURN` the result of adjoining `n` to the end of the array `east`.

line 11: Otherwise we want to choose, recursively, a random k-subset of $\{1, \ldots, n-1\}$, call it `west`, and `RETURN` it.

Program `ListKPerms`, on page 31, lists all of the permutations of an n-set that have k cycles.

line 1: We first define, externally to the routine itself. two auxiliary operations that will be needed. `eastop` simply adjoins a new entry, `j`, to a given list `y`.

line 2: `westop`, on the other hand, replaces the `j`th entry of the list `y` by `n`, where `n-1` is the length of the input list `y`, and adjoins as a new array entry at the end of `y` the former array entry `y[j]`, that was just replaced by `n`. Note carefully that if the input list is the sequence of values of (i.e., the second line of the two line form of) some permutation of `n-1` letters, then this operation in effect inserts `n` between two consecutive entries of some cycle, since before the operation we had consecutive letters $j \to y[j]$ and after the operation we have $j \to n \to y[j]$ consecutively around a cycle.

line 10: After taking care of the base cases of the recursion, we create, recursively, the two lists `east` and `west`, being, respectively, the lists of permutations of `n-1` letters with `k-1` cycles, and with `k` cycles.

line 11: The list `out` lifts the list `east` by adjoining the element `n` to each permutation on that list as a new singleton cycle.

1. 13 The hard part of the recursive construction takes the list `west`, of permutations of `n-1` letters and `k` cycles, and it inserts the new letter `n` in between two consecutive letters of some cycle in all possible ways, on each cycle. This is done by mapping the operation `westop` to the array `west` with all values of `j`.

Program `RandKPerms`, on page 33, chooses a random permutation of an n-set that has k cycles.

line 7: After doing the base cases of the recursion, we make a program `rno` that will, when called, generate a uniformly random number between 0 and 1.

line 8: We want to decide between east and west with probability $\left\{{n-1 \atop k-1}\right\}/\left\{{n \atop k}\right\}$. So we choose a random number `rno`, and if it is less than $\left\{{n-1 \atop k-1}\right\}/\left\{{n \atop k}\right\}$ we will head east, otherwise west.

line 9: To go east, we create, recursively, a random `k-1` permutation of `n-1` letters, then in the next line we adjoin a new entry `n`, to create a singleton cycle for `n`, and exit.

line 12: To go west, we first choose a random integer `r` in the range $(1, n-1)$.

line 13: A random permutation `west` of `n-1` letters with `k` cycles is chosen recursively.

line 14: Then we adjoin a copy of the `r`th entry of `west` as a new `n`th entry of `west`, and finally we substitute `n` for the `r`th entry of `west`, and exit.

Program `ListSetPtns`, on page 41, lists all of the partitions of an n-set into k classes.

line 1: An externally defined operation `EWop` adjoins `m` to the end of the input list `y`.

line 8: Following the disposal of the recursive base cases in lines 5,6 we recursively call this program with `(n-1,k-1)` and `(n-1,k)`, putting the resulting lists into `east` and `west`, respectively.

line 9: By mapping the operation of adjoining `k` to the end of a list, over the list `east`, we create the list `out` of those set partitions in which `n` lives alone in class `k`.

line 10: Here we successively concatenate the list `out` with `k` other lists, the `i`th one of which is the list of all partitions where `n` lives in class `i`, completing the construction.

Program `RandSetPtns`, on page 42, chooses a random partition of an n-set into k classes.

line 7: To generate random numbers in the range $(0, 1)$ we make the program `rno`.

line 8: Now we choose such a random number, and if it is less than $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix} / \begin{bmatrix} n \\ k \end{bmatrix}$ we will go east, else west.

line 10: To go east, we generate, recursively, a random set partition of an $n-1$-set into $k-1$ classes, and adjoin `n` as the singleton `k`th class.

line 12: Else, we go west, and must choose a random class for `n` to live in. This is done by the routine `class`.

line 13: We append the randomly chosen class of `n` to a randomly chosen set partition of `n-1` things into `k` classes, and exit.

Program `ListKPtns`, on page 46, lists the partitions of an integer n whose largest part is k.

line 1: `eastop` simply adds 1 to the first entry of a given list `y`.

line 2: `westop` extends a given list `y` by inserting `j` as a first entry, keeping all of the previous entries.

line 11: The output list of partitions is obtained by concatenating two lists. The first of these is the recursively generated list `west` of partitions of `n-k` whose largest part is `k` to all of which a new part equal to `k` is inserted as a new first part. The second is the recursively generated list `east` of partitions of `n-1` whose largest part is `k-1`, with the first part of every partition on the list increased by 1.

References

[NW] Albert Nijenhuis and Herbert Wilf, *Combinatorial Algorithms*, Academic Press, New York, 1975 and 1978.

[PG] M. B. Monagan et al, *Maple V Programming Guide*, Springer, 1998.

[PWZ] Marko Petkovšek, Herbert Wilf and Doron Zeilberger, $A=B$, A K Peters, Ltd., Wellesley, MA, 1996.