At a basic level, one would expect that constructive combinatorics would address the question of how one constructs the fundamental objects in combinatorics. In fact, "constructing" these objects could mean providing an algorithm for listing all of them, or it could mean generating one of them at random. While both questions are of interest, we shall concentrate on the first.

It is frequently useful in combinatorics to have such listing algorithms. The most obvious application is to use the algorithms to produce computer programs which test conjectures and theorems for combinatorial objects. Better yet, conjectures might be discovered in the resulting data. In another direction, the algorithm itself could be of mathematical interest. The algorithm might be a proof of a theorem. For instance, the existence of an algorithm which lists permutations by transposing adjacent objects proves that any permutation can be written as a product of adjacent transpositions.

Any such list of objects gives the objects a linear order. This means that if a and b are objects, then we can say that a < b if a precedes b on the list. This linear order clearly gives a ranking function on the objects. One might expect that the first object has rank one, and so on. However, we shall find it more useful to define the Rank of an object as the number of objects in the list which precede it. So if a is first, Rank(a) = 0. If the list has N objects, the function Rank must map these objects to the set \{0, 1, \ldots, N-1\}.

While the theoretical definition of Rank is obvious, it is often not at all clear how to construct Rank without listing all of the objects. In fact, algorithms to rank objects (find Rank(a)) and unrank integers (find Unrank(i) = Rank^{-1}(i), i \in \{0, 1, \ldots, N-1\}) often give great insight into the algorithm.

In this chapter we give listing algorithms for these combinatorial objects: permutations, subsets of a set, integer partitions, set partitions and product spaces. Each algorithm will be based upon a recursive formula for the number of objects listed. For example, subsets may be listed by using the combinatorial interpretation of Pascal's triangle.

The most important ranking function will use "lexicographic" ordering. It can
be used for virtually any combinatorial object. We shall see in Chapter 2 that it also
has many remarkable and surprising theoretical properties.

§1.1 Permutations

A permutation of \( n \) distinct objects of length \( k \) is an ordered arrangement of
any \( k \) of the objects. For instance, the permutations of \( \{a, b, c, d\} \) of length two
are \( ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db \) and \( dc \). The next proposition is
clear.

**Proposition 1.1** The number of permutations of \( n \) objects of length \( k \) is
\( n(n-1) \cdots (n-k+1) \).

Sometimes we shall write \( (n)_k \) (called the falling factorial) for
\( n(n-1) \cdots (n-k+1) \).

A permutation of \( n \) objects of length \( n \) is frequently called a permutation of \( n \)
objects (or simply a permutation of \( n \)). It is clear that we can take the set \( [n] = \{1, 2, \ldots , n\} \) for the \( n \) objects. We shall frequently use this notation. Proposition
1.1 shows that the number of permutations of \( n \) is \( (n)_n = n! \).

Perhaps the most natural ordering of the permutations of \( n \) is lexicographic
(lex) order. We say that \( \pi \) precedes \( \sigma \) in lex order, if, for some \( i \), the first \( i \)
entries of \( \pi \) and \( \sigma \) are the same, and the \((i+1)\)th entry of \( \pi \) is less than the \((i+1)\)th
entry of \( \sigma \). The lex list of the permutations of 3 is 123, 132, 213, 231, 312 and
321. This ordering is quite simple. You are asked to consider it in Exercises 2 and 3.
We shall return to lex order in §1.2.

We consider instead an algorithm to list all permutations of \( n \) that is due
to Johnson [Joh] and Trotter [T]. It is based on a "combinatorial proof" of \( n! =
n(n-1)! \): for each of the \((n-1)! \) permutations of \([n-1] \), there are \( n \) "positions" into
which \( n \) may be inserted. The algorithm has the property that each permutation
differs from its predecessor by only a transposition of adjacent symbols. The lex list
does not have this property.

How does the algorithm work? Suppose we have the list for permutations of
\([n-1] \): \( \pi^{(0)}, \pi^{(1)}, \ldots \). Then we construct the list for permutations of \([n] \) by
inserting \( n \) into each of the \( n \) possible positions of each \( \pi^{(i)} \). The insertions go
from left to right if \( i \) is odd and right to left if \( i \) is even. The lists for \( n = 1, 2, 3 \)
and 4 are given below with the recursive structure indicated.