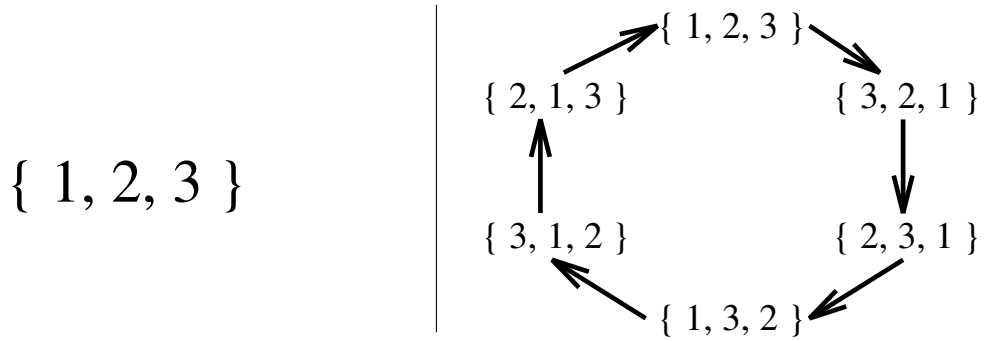

INPUT OUTPUT

14.4 Generating Permutations

Input description: An integer n.

Problem description: Generate (1) all, or (2) a random, or (3) the next permutation of length n.

Discussion: A permutation describes an arrangement or ordering of items. Many algorithmic problems in this catalog seek the best way to order a set of objects, including *traveling salesman* (the least-cost order to visit n cities), *bandwidth* (order the vertices of a graph on a line so as to minimize the length of the longest edge), and *graph isomorphism* (order the vertices of one graph so that it is identical to another). Any algorithm for solving such problems exactly must construct a series of permutations along the way.

There are $n!$ permutations of n items. This grows so quickly that you can't really expect to generate all permutations for $n > 12$, since $12! = 479{,}001{,}600$. Numbers like these should cool the ardor of anyone interested in exhaustive search and help explain the importance of generating random permutations.

Fundamental to any permutation-generation algorithm is a notion of order, the sequence in which the permutations are constructed from first to last. The most natural generation order is *lexicographic*, the sequence they would appear if they were sorted numerically. Lexicographic order for $n = 3$ is $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$, $\{3, 1, 2\}$, and finally $\{3, 2, 1\}$. Although lexicographic order is aesthetically pleasing, there is often no particular advantage to using it. For example, if you are searching through a collection of files, it does not matter whether the filenames are encountered in sorted order, so long as you eventually search through all of them. Indeed, nonlexicographic orders lead to faster and simpler permutation generation algorithms.

There are two different paradigms for constructing permutations: ranking/unranking and incremental change methods. The latter are more efficient, but ranking and unranking can be applied to solve a much wider class of problems. The key is to define the functions *rank* and *unrank* on all permutations p and integers n, m, where $|p| = n$ and $0 \leq m \leq n!$.

- *Rank(p)* – What is the position of p in the given generation order? A typical ranking function is recursive, such as basis case $Rank(\{1\}) = 0$ with

  $$Rank(p) = (p_1 - 1) \cdot (|p| - 1)! + Rank(p_2, \ldots, p_{|p|})$$

  Getting this right means relabeling the elements of the smaller permutation to reflect the deleted first element. Thus

  $$Rank(\{2, 1, 3\}) = 1 \cdot 2! + Rank(\{1, 2\}) = 2 + 0 \cdot 1! + Rank(\{1\}) = 2$$

- *Unrank(m,n)* – Which permutation is in position m of the $n!$ permutations of n items? A typical unranking function finds the number of times $(n-1)!$ goes into m and proceeds recursively. $Unrank(2, 3)$ tells us that the first element of the permutation must be '2', since $(2-1) \cdot (3-1)! \leq 2$ but $(3-1) \cdot (3-1)! > 2$. Deleting $(2-1) \cdot (3-1)!$ from m leaves the smaller problem $Unrank(0, 2)$. The ranking of 0 corresponds to the total order. The total order on the two remaining elements (since 2 has been used) is $\{1, 3\}$, so $Unrank(2, 3) = \{2, 1, 3\}$.

What the actual rank and unrank functions are does not matter as much as the fact that they must be inverses of each other. In other words, $p = Unrank(Rank(p), n)$ for all permutations p. Once you define ranking and unranking functions for permutations, you can solve a host of related problems:

- *Sequencing permutations* – To determine the *next* permutation that occurs in order after p, we can $Rank(p)$, add 1, and then $Unrank(p)$. Similarly, the permutation right before p in order is $Unrank(Rank(p) - 1, |p|)$. Counting through the integers from 0 to $n! - 1$ and unranking them is equivalent to generating all permutations.

- *Generating random permutations* – Selecting a random integer from 0 to $n!-1$ and then unranking it yields a truly random permutation.

- *Keep track of a set of permutations* – Suppose we want to construct random permutations and act only when we encounter one we have not seen before. We can set up a bit vector (see Section 12.5 (page 385)) with $n!$ bits, and set bit i to 1 if permutation *Unrank(i,n)* has been seen. A similar technique was employed with k-subsets in the Lotto application of Section 1.6 (page 23).

This rank/unrank method is best suited for small values of n, since $n!$ quickly exceeds the capacity of machine integers unless arbitrary-precision arithmetic is available (see Section 13.9 (page 423)). The incremental change methods work by defining the *next* and *previous* operations to transform one permutation into another, typically by swapping two elements. The tricky part is to schedule the swaps so that permutations do not repeat until all of them have been generated. The output picture above gives an ordering of the six permutations of $\{1, 2, 3\}$ using a single swap between successive permutations.

Incremental change algorithms for sequencing permutations are tricky, but they are so concise that they can be expressed in a dozen-line program. See the implementation section for pointers to code. Because the incremental change is only a single swap, these algorithms can be extremely fast—on average, constant time—which is independent of the size of the permutation! The secret is to represent the permutation using an n-element array to facilitate the swap. In certain applications, only the change between permutations is important. For example, in a brute-force program to search for the optimal TSP tour, the cost of the tour associated with the new permutation will be that of the previous permutation, with the addition and deletion of four edges.

Throughout this discussion, we have assumed that the items we are permuting are all distinguishable. However, if there are duplicates (meaning our set is a *multi-set*), you can save considerable time and effort by avoiding identical permutations. For example, there are only ten distinct permutations of $\{1, 1, 2, 2, 2\}$, instead of 120. To avoid duplicates, use backtracking and generate the permutations in lexicographic order.

Generating random permutations is an important little problem that people often stumble across, and often botch up. The right way is to use the following two-line, linear-time algorithm. We assume that *Random[i,n]* generates a random integer between i and n, inclusive.

for $i = 1$ to n do $a[i] = i$;
for $i = 1$ to $n - 1$ do $swap[a[i], a[Random[i, n]]$;

That this algorithm generates all permutations uniformly at random is not obvious. If you think so, convincingly explain why the following algorithm *does not* generate permutations uniformly:

for $i = 1$ to n do $a[i] = i$;
for $i = 1$ to $n - 1$ do $swap[a[i], a[Random[1, n]]$;

Such subtleties demonstrate why you must be very careful with random generation algorithms. Indeed, we recommend that you try some reasonably extensive experiments with *any* random generator before really believing it. For example, generate 10,000 random permutations of length 4 and see whether all 24 distinct permutations occur approximately the same number of times. If you know how to measure statistical significance, you are in even better shape.

Implementations: The C++ *Standard Template Library* (STL) provides two functions (`next_permutation` and `prev_permutation`) for sequencing permutations in lexicographic order. Kreher and Stinson [KS99] provide implementations of minimum change and lexicographic permutation generation in C at *http://www.math.mtu.edu/~kreher/cages/Src.html*.

The *Combinatorial Object Server* (*http://theory.cs.uvic.ca/*) developed by Frank Ruskey of the University of Victoria is a unique resource for generating permutations, subsets, partitions, graphs, and other objects. An interactive interface enables you to specify which objects you would like returned to you. Implementations in C, Pascal, and Java are available for certain types of objects.

C++ routines for generating an astonishing variety of combinatorial objects, including permutations and cyclic permutations, are available at *http://www.jjj.de/fxt/*.

Nijenhuis and Wilf [NW78] is a venerable but still excellent source on generating combinatorial objects. They provide efficient Fortran implementations of algorithms to construct random permutations and to sequence permutations in minimum-change order. Also included are routines to extract the cycle structure of a permutation. See Section 19.1.10 (page 661) for details.

Combinatorica [PS03] provides Mathematica implementations of algorithms that construct random permutations and sequence permutations in minimum change and lexicographic orders. It also provides a backtracking routine to construct all distinct permutations of a multiset, and it supports various permutation group operations. See Section 19.1.9 (page 661).

Notes: The best recent reference on permutation generation is Knuth [Knu05a]. Sedgewick's excellent survey on the topic is older [Sed77], but this is not a fast moving area. Good expositions include [KS99, NW78, Rus03].

Fast permutation generation methods make only a single swap between successive permutations. The Johnson-Trotter algorithm [Joh63, Tro62] satisfies an even stronger condition, namely that the two elements being swapped are always adjacent. Simple linear-time ranking and unranking functions for permutations are given by Myrvold and Ruskey [MR01].

In the days before ready access to computers, books with tables of random permutations [MO63] were used instead of algorithms. The swap-based random permutation algorithm presented above was first described in [MO63].

Related Problems: Random-number generation (see page 415), generating subsets (see page 452), generating partitions (see page 456).