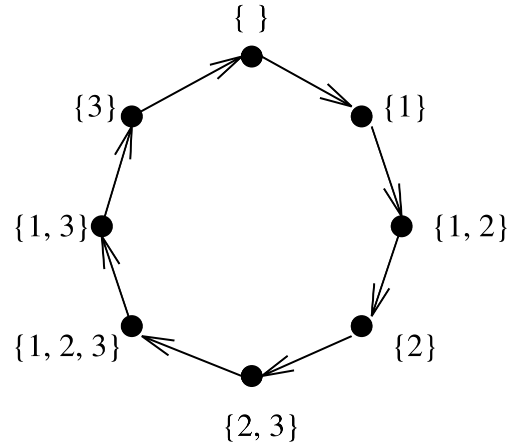
$\{ 1, 2, 3 \}$



INPUT OUTPUT

14.5 Generating Subsets

Input description: An integer n.

Problem description: Generate (1) all, or (2) a random, or (3) the next subset of the integers $\{1, \ldots, n\}$.

Discussion: A subset describes a selection of objects, where the order among them does not matter. Many important algorithmic problems seek the best subset of a group of things: *vertex cover* seeks the smallest subset of vertices to touch each edge in a graph; *knapsack* seeks the most profitable subset of items of bounded total size; while *set packing* seeks the smallest subset of subsets that together cover each item exactly once.

There are 2^n distinct subsets of an n-element set, including the empty set as well as the set itself. This grows exponentially, but at a considerably slower rate than the $n!$ permutations of n items. Indeed, since $2^{20} = 1{,}048{,}576$, a brute-force search through all subsets of 20 elements is easily manageable. Since $2^{30} = 1{,}073{,}741{,}824$, you will certainly hit limits for slightly larger values of n.

By definition, the relative order among the elements does not distinguish different subsets. Thus, $\{1, 2, 5\}$ is the same as $\{2, 1, 5\}$. However, it is a very good idea to maintain your subsets in a sorted or *canonical* order to speed up such operations as testing whether or not two subsets are identical.

As with permutations (see Section 14.4 (page 448)), the key to subset generation problems is establishing a numerical sequence among all 2^n subsets. There are three primary alternatives:

- *Lexicographic order* – Lexicographic order means sorted order, and is often the most natural way to generate combinatorial objects. The eight subsets of $\{1,2,3\}$ in lexicographic order are $\{\}$, $\{1\}$, $\{1,2\}$, $\{1,2,3\}$, $\{1,3\}$, $\{2\}$, $\{2,3\}$, and $\{3\}$. But it is surprisingly difficult to generate subsets in lexicographic order. Unless you have a compelling reason to do so, don't bother.

- *Gray Code* – A particularly interesting and useful subset sequence is the minimum change order, wherein adjacent subsets differ by the insertion or deletion of exactly one element. Such an ordering, called a *Gray code*, appears in the output picture above.

  Generating subsets in Gray code order can be very fast, because there is a nice recursive construction. Construct a Gray code of $n-1$ elements G_{n-1}. Reverse a second copy of G_{n-1} and add n to each subset in this copy. Then concatenate them together to create G_n. Study the output example for clarification.

  Further, since only one element changes between subsets, exhaustive search algorithms built on Gray codes can be quite efficient. A set cover program would only have to update the change in coverage by the addition or deletion of one subset. See the implementation section below for Gray code subset-generation programs.

- *Binary counting* – The simplest approach to subset-generation problems is based on the observation that any subset S' is defined by the items of that S are in S'. We can represent S' by a binary string of n bits, where bit i is 1 iff the ith element of S is in S'. This defines a bijection between the 2^n binary strings of length n, and the 2^n subsets of n items. For $n = 3$, binary counting generates subsets in the following order: {}, {3}, {2}, {2,3}, {1}, {1,3}, {1,2}, {1,2,3}.

  This binary representation is the key to solving all subset generation problems. To generate all subsets in order, simply count from 0 to $2^n - 1$. For each integer, successively mask off each of the bits and compose a subset of exactly the items corresponding to 1 bits. To generate the *next* or *previous* subset, increment or decrement the integer by one. *Unranking* a subset is exactly the masking procedure, while *ranking* constructs a binary number with 1's corresponding to items in S and then converts this binary number to an integer.

  To generate a random subset, you might generate a random integer from 0 to $2^n - 1$ and unrank, although how your random number generator rounds things off might mean that certain subsets can never occur. Much better is to flip a coin n times, with the ith flip deciding whether to include element i in the subset. A coin flip can be robustly simulated by generating a random real or large integer and testing whether it is bigger or smaller than half its range. A Boolean array of n items can thus be used to represent subsets as a sort of premasked integer.

Generation problems for two closely related problems arise often in practice:

- *K-subsets* – Instead of constructing all subsets, we may only be interested in the subsets containing exactly k elements. There are $\binom{n}{k}$ such subsets, which is substantially less than 2^n, particularly for small values of k.

  The best way to construct all k-subsets is in lexicographic order. The ranking function is based on the observation that there are $\binom{n-f}{k-1}$ k-subsets whose smallest element is f. Using this, it is possible to determine the smallest element in the mth k-subset of n items. We then proceed recursively for subsequent elements of the subset. See the implementations below for details.

- *Strings* – Generating all subsets is equivalent to generating all 2^n strings of true and false. The same basic techniques apply to generate all or random strings on alphabets of size α, except there will be α^n strings in total.

Implementations: Kreher and Stinson [KS99] provide generators for both subsets and k-subsets, including lexicographic and Gray code orders. These implementations in C are available 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 subsets and k-subsets (combinations), are available in the combinatorics package 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 subsets and to sequence subsets in Gray code and lexicographic order. They also provide routines to construct random k-subsets and sequence them in lexicographic order. See Section 19.1.10 (page 661) for details on ftp-ing these programs. Algorithm 515 [BL77] of the *Collected Algorithms of the ACM* is another Fortran implementation of lexicographic k-subsets, available from Netlib (see Section 19.1.5 (page 659)).

Combinatorica [PS03] provides Mathematica implementations of algorithms to construct random subsets and sequence subsets in Gray code, binary, and lexicographic order. They also provide routines to construct random k-subsets and strings, and sequence them lexicographically. See Section 19.1.9 (page 661) for further information on Combinatorica.

Notes: The best reference on subset generation is Knuth [Knu05b]. Good expositions include [KS99, NW78, Rus03]. Wilf [Wil89] provides an update of [NW78], including a thorough discussion of modern Gray code generation problems.

Gray codes were first developed [Gra53] to transmit digital information in a robust manner over an analog channel. By assigning the code words in Gray code order, the ith word differs only slightly from the $(i + 1)$st, so minor fluctuations in analog signal strength corrupts only a few bits. Gray codes have a particularly nice correspondence to Hamiltonian cycles on the hypercube. Savage [Sav97] gives an excellent survey of Gray codes (minimum change orderings) for a large class of combinatorial objects, including subsets.

The popular puzzle *Spinout*, manufactured by ThinkFun (formerly Binary Arts Corporation), is solved using ideas from Gray codes.

Related Problems: Generating permutations (see page 448), generating partitions (see page 456).