4.3 Algorithms for Generating Combinatorial Objects

In this section, we keep our promise to discuss algorithms for generating combinatorial objects. The most important types of combinatorial objects are permutations, combinations, and subsets of a given set. They typically arise in problems that require a consideration of different choices. We already encountered them in Chapter 3 when we discussed exhaustive search. Combinatorial objects are studied in a branch of discrete mathematics called combinatorics. Mathematicians, of course, are primarily interested in different counting formulas; we should be grateful for such formulas because they tell us how many items need to be generated. In particular, they warn us that the number of combinatorial objects typically grows exponentially or even faster as a function of the problem size. But our primary interest here lies in algorithms for generating combinatorial objects, not just in counting them.

Generating Permutations

We start with permutations. For simplicity, we assume that the underlying set whose elements need to be permuted is simply the set of integers from 1 to n; more generally, they can be interpreted as indices of elements in an n-element set $\{a_1, \ldots, a_n\}$. What would the decrease-by-one technique suggest for the problem of generating all $n!$ permutations of $\{1, \ldots, n\}$? The smaller-by-one problem is to generate all $(n-1)!$ permutations. Assuming that the smaller problem is solved, we can get a solution to the larger one by inserting n in each of the n possible positions among elements of every permutation of $n-1$ elements. All the permutations obtained in this fashion will be distinct (why?), and their total number will be $n(n-1)! = n!$. Hence, we will obtain all the permutations of $\{1, \ldots, n\}$.

We can insert n in the previously generated permutations either left to right or right to left. It turns out that it is beneficial to start with inserting n into $12 \ldots (n-1)$ by moving right to left and then switch direction every time a new permutation of $\{1, \ldots, n-1\}$ needs to be processed. An example of applying this approach bottom up for $n = 3$ is given in Figure 4.9.

The advantage of this order of generating permutations stems from the fact that it satisfies the ***minimal-change requirement***: each permutation can be obtained from its immediate predecessor by exchanging just two elements in it. (For the method being discussed, these two elements are always adjacent to each other.

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

FIGURE 4.9 Generating permutations bottom up.

Check this for the permutations generated in Figure 4.9.) The minimal-change requirement is beneficial both for the algorithm's speed and for applications using the permutations. For example, in Section 3.4, we needed permutations of cities to solve the traveling salesman problem by exhaustive search. If such permutations are generated by a minimal-change algorithm, we can compute the length of a new tour from the length of its predecessor in constant rather than linear time (how?).

It is possible to get the same ordering of permutations of n elements without explicitly generating permutations for smaller values of n. It can be done by associating a direction with each element k in a permutation. We indicate such a direction by a small arrow written above the element in question, e.g.,

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

The element k is said to be ***mobile*** in such an arrow-marked permutation if its arrow points to a smaller number adjacent to it. For example, for the permutation $\overrightarrow{3}\overleftarrow{2}\overrightarrow{4}\overleftarrow{1}$, 3 and 4 are mobile while 2 and 1 are not. Using the notion of a mobile element, we can give the following description of the ***Johnson-Trotter algorithm*** for generating permutations.

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\,\overleftarrow{2}\ldots\overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

Here is an application of this algorithm for $n = 3$, with the largest mobile element shown in bold:

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}} \quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2} \quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{2}} \quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}.$$

This algorithm is one of the most efficient for generating permutations; it can be implemented to run in time proportional to the number of permutations, i.e., in $\Theta(n!)$. Of course, it is horribly slow for all but very small values of n; however, this is not the algorithm's "fault" but rather the fault of the problem: it simply asks to generate too many items.

One can argue that the permutation ordering generated by the Johnson-Trotter algorithm is not quite natural; for example, the natural place for permutation $n(n-1)\ldots 1$ seems to be the last one on the list. This would be the case if permutations were listed in increasing order—also called the ***lexicographic or-***

der—which is the order in which they would be listed in a dictionary if the numbers were interpreted as letters of an alphabet. For example, for $n = 3$,

$$123 \quad 132 \quad 213 \quad 231 \quad 312 \quad 321.$$

So how can we generate the permutation following $a_1 a_2 \ldots a_{n-1} a_n$ in lexicographic order? If $a_{n-1} < a_n$, which is the case for exactly one half of all the permutations, we can simply transpose these last two elements. For example, 123 is followed by 132. If $a_{n-1} > a_n$, we find the permutation's longest decreasing suffix $a_{i+1} > a_{i+2} > \cdots > a_n$ (but $a_i < a_{i+1}$); increase a_i by exchanging it with the smallest element of the suffix that is greater than a_i; and reverse the new suffix to put it in increasing order. For example, 362541 is followed by 364125. Here is pseudocode of this simple algorithm whose origins go as far back as 14th-century India.

ALGORITHM *LexicographicPermute*(n)

```
//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of {1, ..., n} in lexicographic order
initialize the first permutation with 12 ... n
while last permutation has two consecutive elements in increasing order do
    let i be its largest index such that a_i < a_{i+1}   //a_{i+1} > a_{i+2} > ... > a_n
    find the largest index j such that a_i < a_j   //j ≥ i + 1 since a_i < a_{i+1}
    swap a_i with a_j   //a_{i+1} a_{i+2} ... a_n will remain in decreasing order
    reverse the order of the elements from a_{i+1} to a_n inclusive
    add the new permutation to the list
```

Generating Subsets

Recall that in Section 3.4 we examined the knapsack problem, which asks to find the most valuable subset of items that fits a knapsack of a given capacity. The exhaustive-search approach to solving this problem discussed there was based on generating all subsets of a given set of items. In this section, we discuss algorithms for generating all 2^n subsets of an abstract set $A = \{a_1, \ldots, a_n\}$. (Mathematicians call the set of all subsets of a set its ***power set***.)

The decrease-by-one idea is immediately applicable to this problem, too. All subsets of $A = \{a_1, \ldots, a_n\}$ can be divided into two groups: those that do not contain a_n and those that do. The former group is nothing but all the subsets of $\{a_1, \ldots, a_{n-1}\}$, while each and every element of the latter can be obtained by adding a_n to a subset of $\{a_1, \ldots, a_{n-1}\}$. Thus, once we have a list of all subsets of $\{a_1, \ldots, a_{n-1}\}$, we can get all the subsets of $\{a_1, \ldots, a_n\}$ by adding to the list all its elements with a_n put into each of them. An application of this algorithm to generate all subsets of $\{a_1, a_2, a_3\}$ is illustrated in Figure 4.10.

Similarly to generating permutations, we do not have to generate power sets of smaller sets. A convenient way of solving the problem directly is based on a one-to-one correspondence between all 2^n subsets of an n element set $A = \{a_1, \ldots, a_n\}$