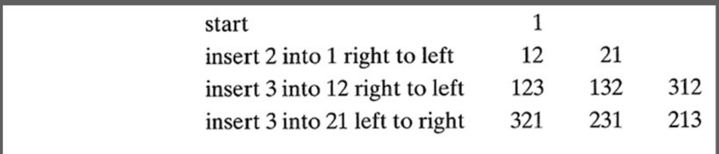
4.3 Algorithms for Generating Combinatorial Objects

Generating Permutations

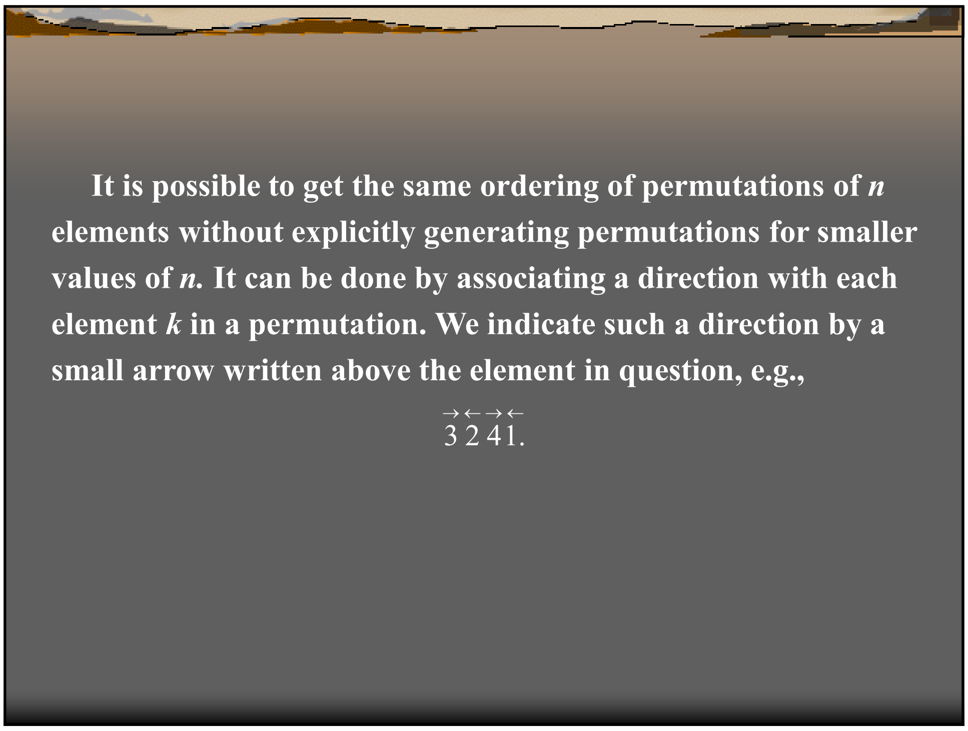
The advantage of this order of generating permutations stems from the fact that it satisfies the *minimal-change requirement*: each permutation can be obtained from its immediate predecessor by exchanging just two elements in it. (For the method being discussed, these two elements are always adjacent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



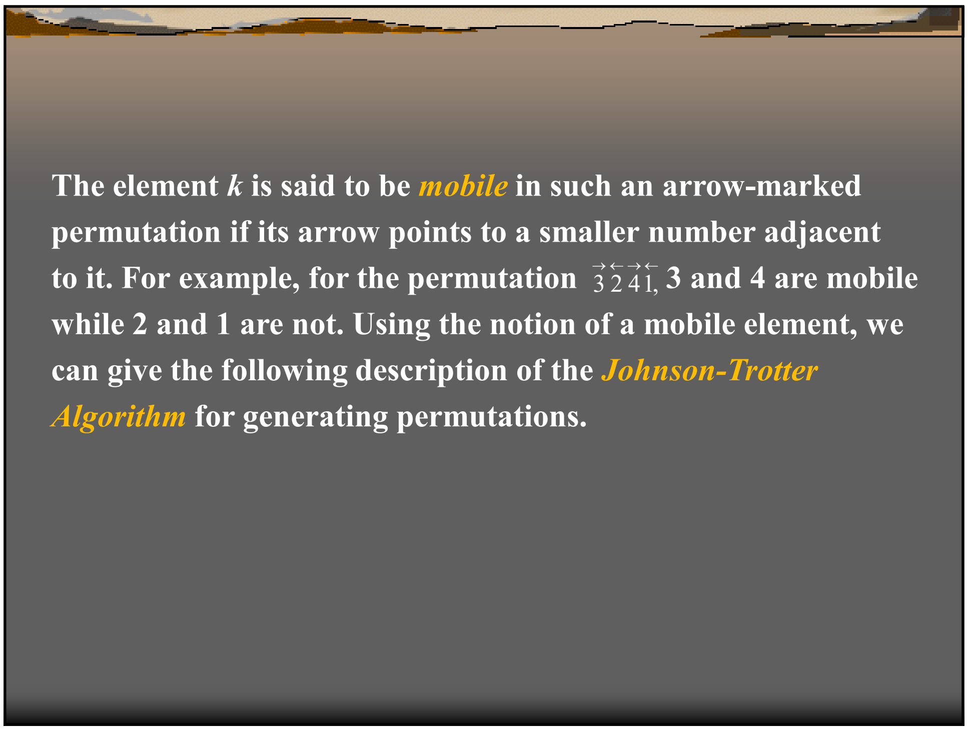
start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

FIGURE 4.9 Generating permutations bottom up.

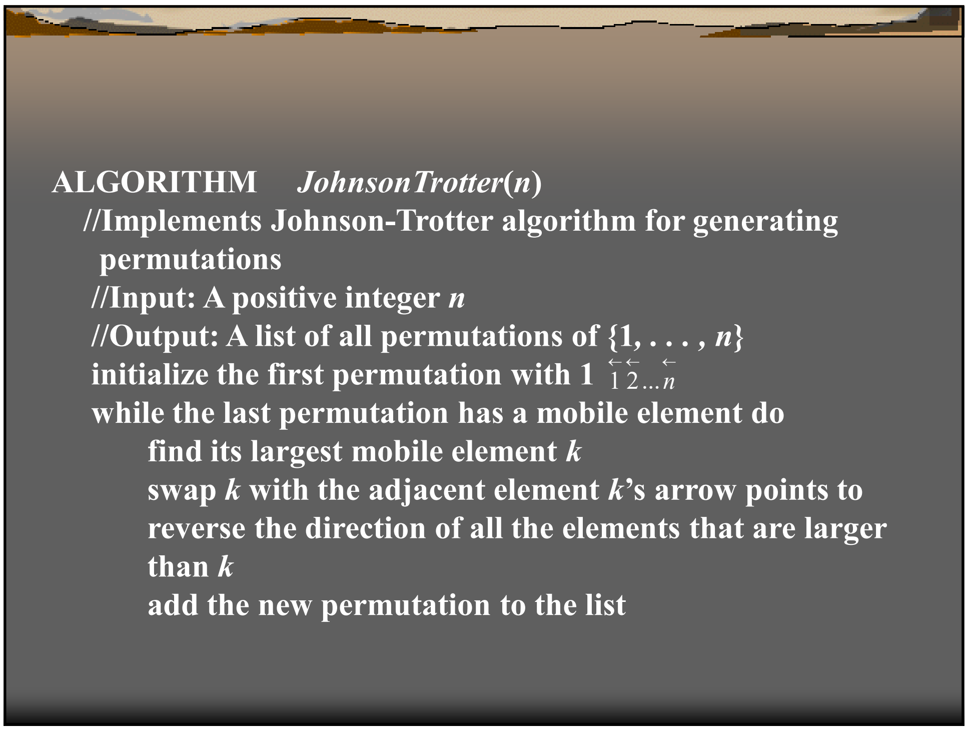


It is possible to get the same ordering of permutations of *n* elements without explicitly generating permutations for smaller values of *n*. It can be done by associating a direction with each element *k* in a permutation. We indicate such a direction by a small arrow written above the element in question,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

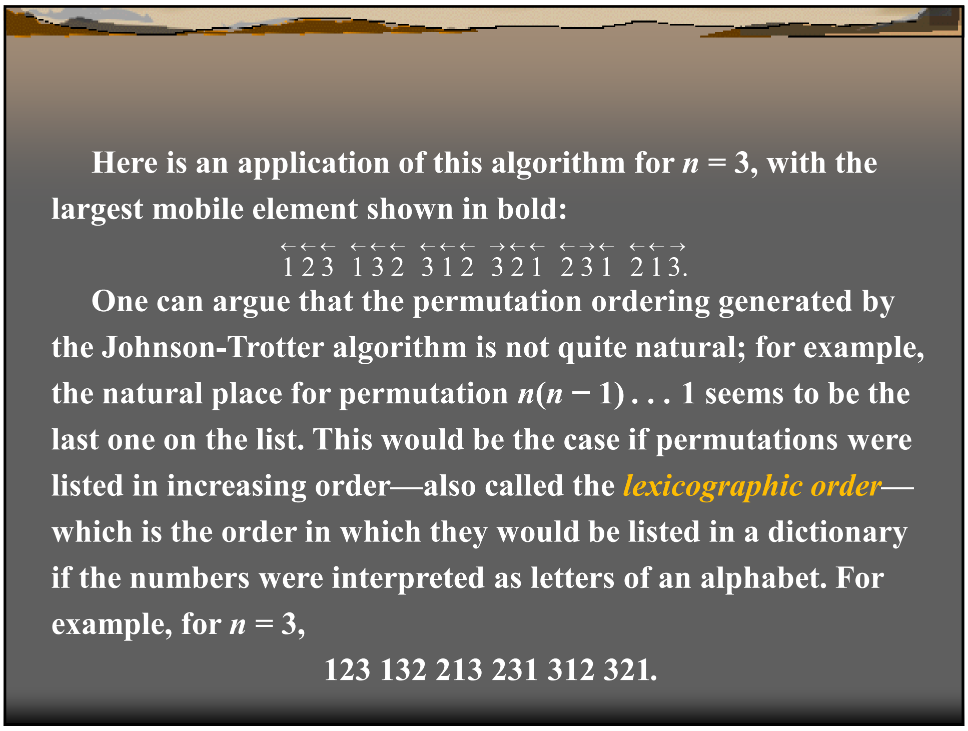
$$\overset{\rightarrow}{3}\overset{\leftarrow}{2}\overset{\rightarrow}{4}\overset{\leftarrow}{1}.$$



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 $\overset{\rightarrow}{3}\overset{\leftarrow}{2}\overset{\rightarrow}{4}\overset{\leftarrow}{1}$, 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 $\{1, . . . , n\}$
  initialize the first permutation with 1 $\overleftarrow{1}\,\overleftarrow{2}\ldots\overleftarrow{n}$
  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
```

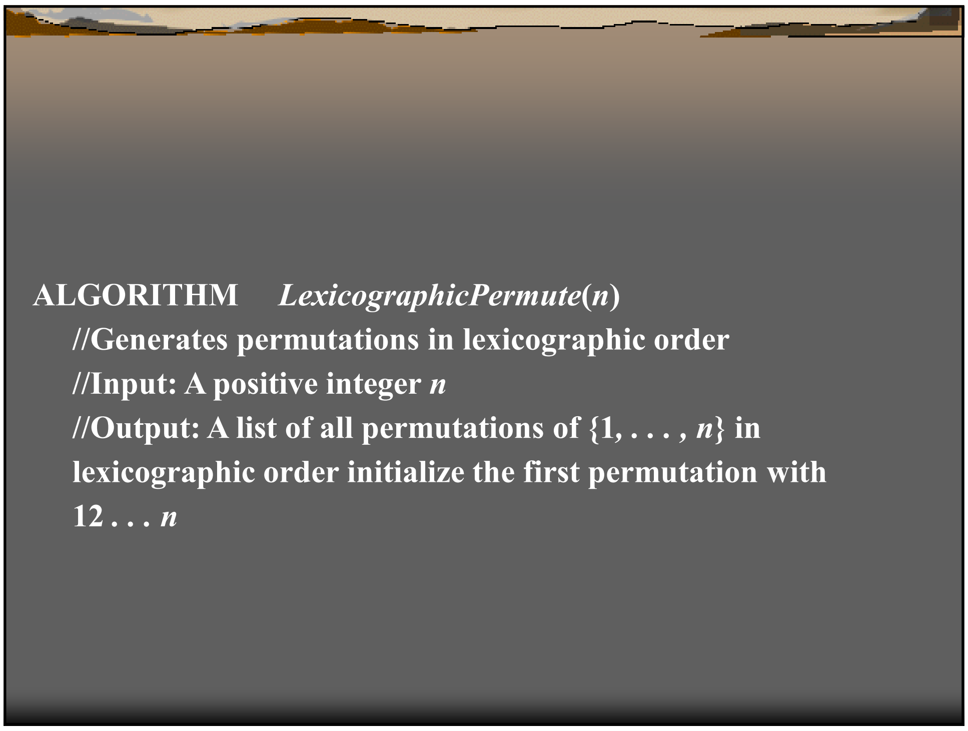


Here is an application of this algorithm for $n = 3$, with the largest mobile element shown in bold:

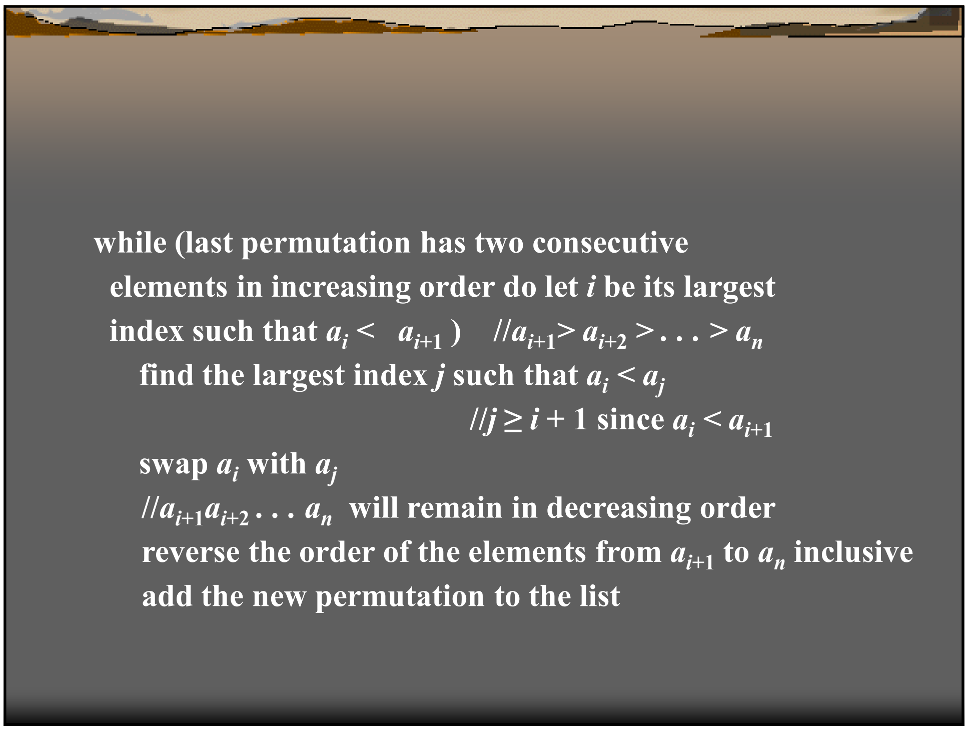
$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3}\quad \overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{2}\quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2}\quad \overrightarrow{3}\,\overleftarrow{2}\,\overleftarrow{1}\quad \overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1}\quad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

One can argue that the permutation ordering generated by the Johnson-Trotter algorithm is not quite natural; for example, the natural place for permutation $n(n - 1) . . . 1$ 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 order*—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 $n = 3$,

123 132 213 231 312 321.



```
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
    $12 . . . n$
```

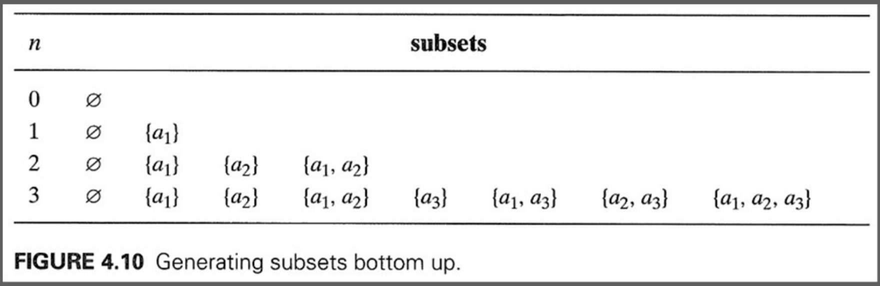


```
while (last permutation has two consecutive
  elements in increasing order do let $i$ be its largest
  index such that $a_i < a_{i+1}$ )    //$a_{i+1} > a_{i+2} > . . . > a_n$
    find the largest index $j$ such that $a_i < a_j$
                                        //$j \geq i + 1$ since $a_i < a_{i+1}$
    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
```

Example. The first eight element in the list of permutation of {1, 2, 3, 4} in lexicographic order.
1234 1243 1324 1342 1423 1432 2134 2143

Generating Subsets

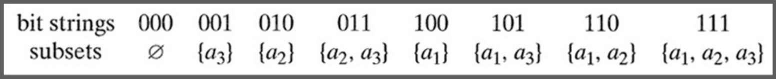
Recall that in Section 3.4 we examined the knapsack problem, which asks to find the most valuable subset of items that fits a knapsack of a given capacity. The exhaustive-search approach to solving this problem discussed there was based on generating all subsets of a given set of items. In this section, 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*.)



n	subsets							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

FIGURE 4.10 Generating subsets bottom up.

For example, for the case of $n = 3$, we obtain



bit strings	000	001	010	011	100	101	110	111
subsets	$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

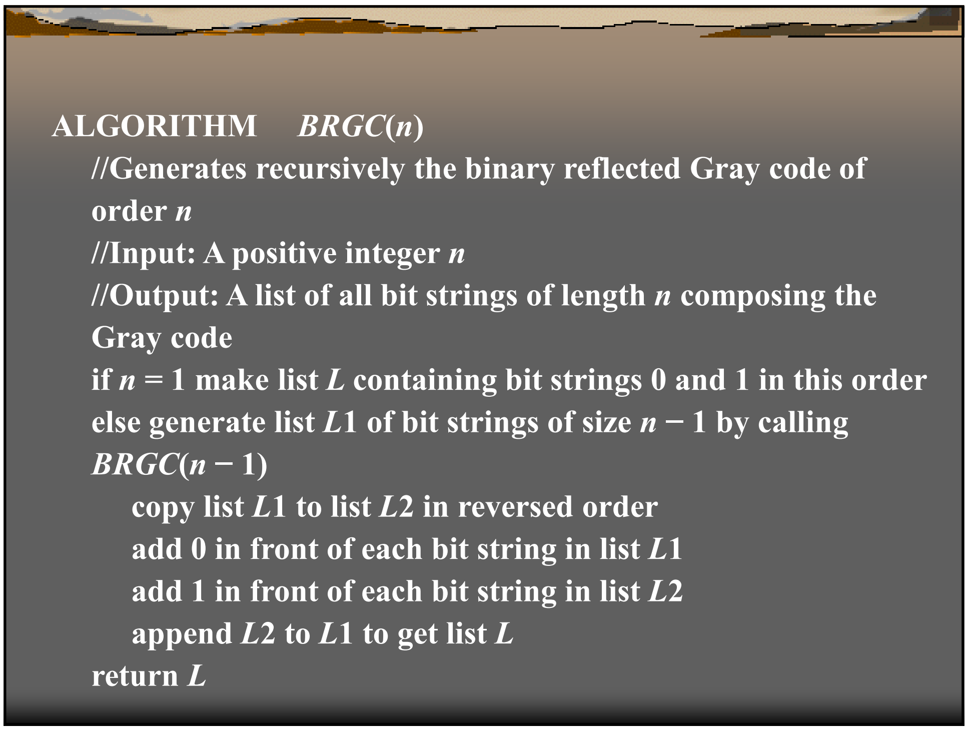
Note that although the bit strings are generated by this algorithm in lexicographic order (in the two-symbol alphabet of 0 and 1), the order of the subsets looks anything but natural. For example, we might want to have the so-called *squashed order*, in which any subset involving a_j can be listed only after all the subsets involving $a_1, \ldots, a_{j-1}$. For example, the list of three elements is

$\emptyset, \{a_1\}, \{a_2\}, \{a_1, a_2\}, \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$.

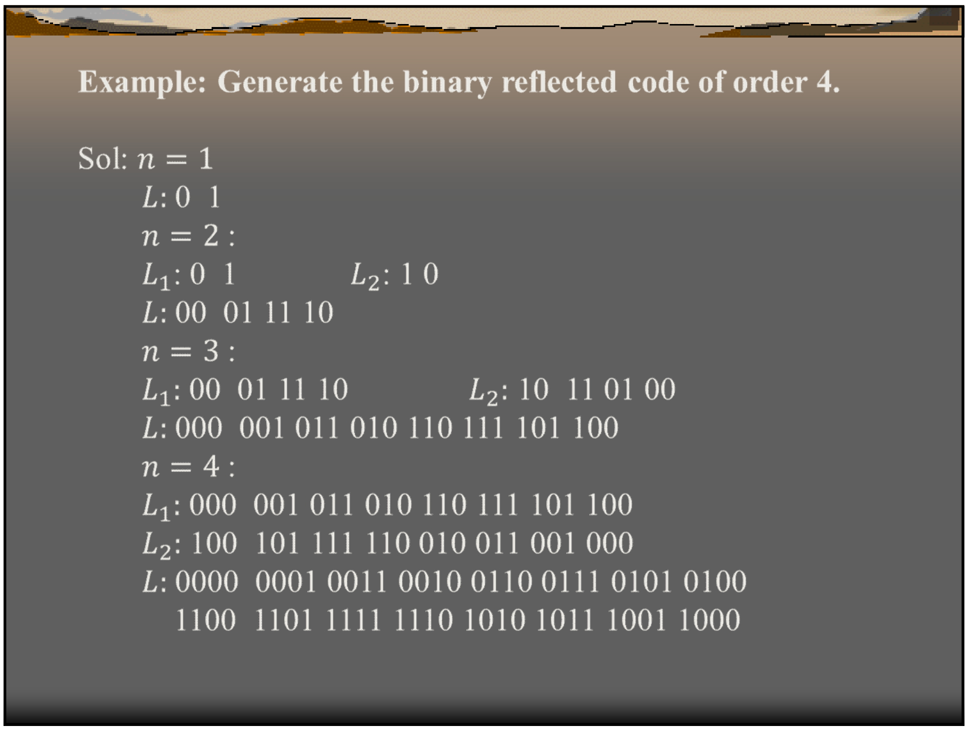
A more challenging question is whether there exists a minimal-change algorithm for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit. (In the language of subsets, we want every subset to differ from its immediate predecessor by either an addition or a deletion, but not both, of a single element.) The answer to this question is yes. For example, for *n* = 3, we can get

000 001 011 010 110 111 101 100.

Such a sequence of bit strings is called the *binary reflected Gray code*. Frank Gray, a researcher at AT&T Bell Laboratories, reinvented it in the 1940s to minimize the effect of errors in transmitting digital signals (see, e.g., [Ros07], pp. 642–643). Seventy years earlier, the French engineer E' mile Baudot used such codes in telegraphy. Here is pseudocode that generates the binary reflected Gray code recursively.



```
ALGORITHM    BRGC(n)
    //Generates recursively the binary reflected Gray code of order n
    //Input: A positive integer n
    //Output: A list of all bit strings of length n composing the Gray code
    if n = 1 make list L containing bit strings 0 and 1 in this order
    else generate list L1 of bit strings of size n − 1 by calling BRGC(n − 1)
        copy list L1 to list L2 in reversed order
        add 0 in front of each bit string in list L1
        add 1 in front of each bit string in list L2
        append L2 to L1 to get list L
    return L
```



Example: Generate the binary reflected code of order 4.

Sol: $n = 1$

L: 0 1

$n = 2$:

L_1: 0 1 L_2: 1 0

L: 00 01 11 10

$n = 3$:

L_1: 00 01 11 10 L_2: 10 11 01 00

L: 000 001 011 010 110 111 101 100

$n = 4$:

L_1: 000 001 011 010 110 111 101 100

L_2: 100 101 111 110 010 011 001 000

L: 0000 0001 0011 0010 0110 0111 0101 0100
1100 1101 1111 1110 1010 1011 1001 1000