4. As you know from **Chapter 5**, integers are represented inside the computer as a sequence of bits, each of which is a single digit in the binary number system and can therefore have only the value 0 or 1. With *N* bits, you can represent 2^N distinct integers. For example, three bits are sufficient to represent the eight (2^3) integers between 0 and 7, as follows:

`000`	$\rightarrow$	0
`001`	$\rightarrow$	1
`010`	$\rightarrow$	2
`011`	$\rightarrow$	3
`100`	$\rightarrow$	4
`101`	$\rightarrow$	5
`110`	$\rightarrow$	6
`111`	$\rightarrow$	7

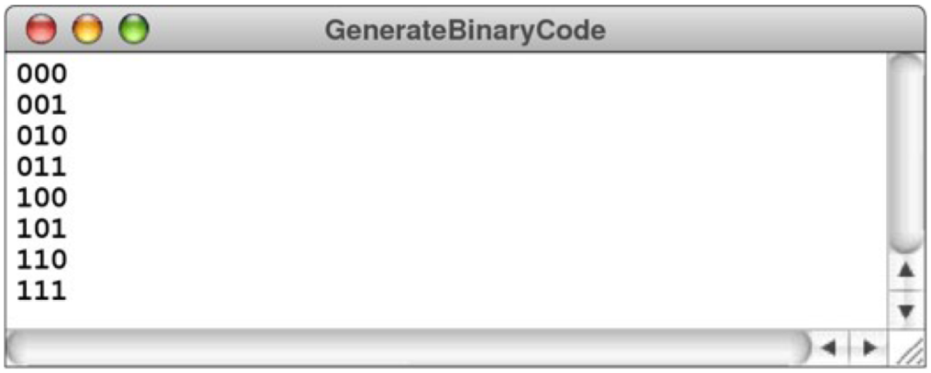
The bit patterns for these integers follow a recursive pattern. The binary numbers with *N* bits consist of the following two sets in order:

- All binary numbers with $N - 1$ bits preceded by a `0`
- All binary numbers with $N - 1$ bits preceded by a `1`

Write a recursive function

```
void generateBinaryCode(int nBits)
```

that generates the bit patterns for the binary representation of all integers that can be represented using the specified number of bits. For example, calling `generateBinaryCode(3)` should produce the following output:



```
000
001
010
011
100
101
110
111
```

5. Although the binary coding used in exercise 4 is good for most applications, it has certain drawbacks. As you count in standard binary notation, there are some points in the sequence at which several bits change at the same time. For example, in the three-bit binary code, the value of every bit changes as you move from 3 (`011`) to 4 (`100`).

   In some applications, this instability in the bit patterns used to represent adjacent numbers can lead to problems. Suppose, for example, that you are using a hardware measurement device containing a three-bit value that varies between 3 and 4. Sometimes, the device will register `011` to indicate the value 3; at other times, it will register `100` to indicate 4. For this device to work correctly, the bit transitions must occur simultaneously. If the first bit changes more quickly than the others, for example, there may be an intermediate state in which the device reads `111`, which would be a highly inaccurate reading.

   You can avoid this problem simply by assigning three-bit values to the numbers 0 through 7 so that only one bit changes in the representation when you move between adjacent integers. Such an encoding is called a ***Gray code*** (after its inventor, the mathematician Frank Gray) and looks like this:

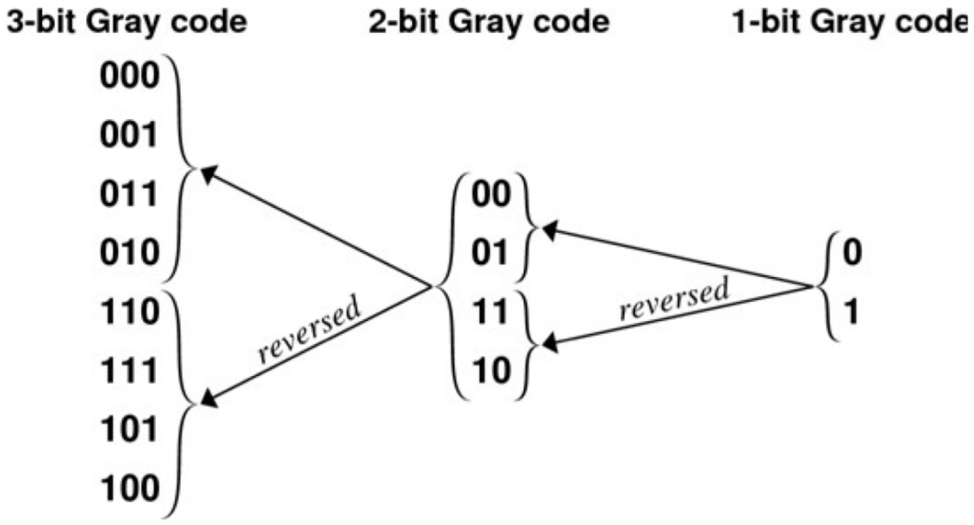
`000`	$\to$	0
`001`	$\to$	1
`011`	$\to$	2
`010`	$\to$	3
`110`	$\to$	4
`111`	$\to$	5
`101`	$\to$	6
`100`	$\to$	7

   The recursive insight that you need to create a Gray code of *N* bits is summarized in the following informal procedure:

   1. Write down the Gray code for $N - 1$ bits.
   2. Copy that same list *in reverse order* below the original one.
   3. Add a `0` bit in front of the encodings in the original half of the list and a `1` bit in front of those in the reversed copy.

   This procedure is illustrated in the following derivation of the Gray code for three bits:

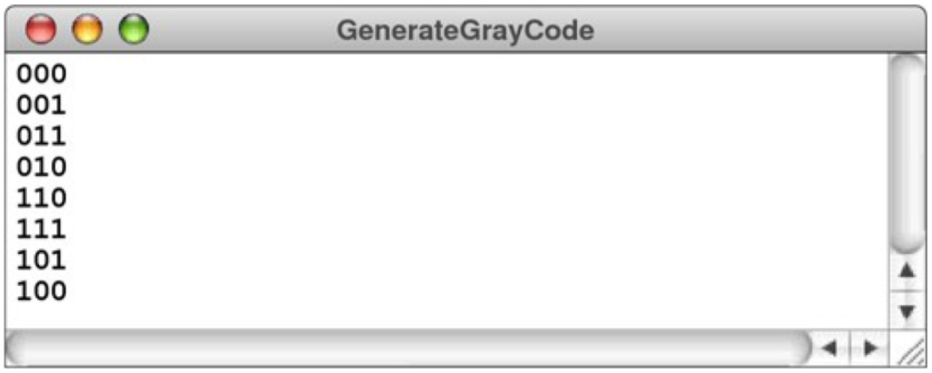
This procedure is illustrated in the following derivation of the Gray code for three bits:



Write a recursive function `generateGrayCode(nBits)` that generates the Gray code patterns for the specified number of bits. For example, if you call the function

```
generateGrayCode(3)
```

the program should produce the following output:



```
000
001
011
010
110
111
101
100
```