Hamming Codes

Points:
Due date:

The project description borrows in part from Wikipedia's Hamming Code page. You are encouraged to read their description of Hamming Codes for a more in depth understanding.

Background

Most students understand parity bits. A parity bit is an extra 0 or 1 attached to a byte (or larger block of data) to help detect if an error has occurred. For example, with even parity, each byte together with its parity bit will contain an even number of 1s. If the byte itself contains an odd number of 1s, then the parity bit is set to 1, to make the total number even. Otherwise the parity bit is set to 0. Obviously, if any one of the bits gets flipped, the number of 1s will be odd and we can determine that the byte contains an error.

There are a couple of problems with parity bits. First, if multiple errors occur within the same byte, our parity check might not return an error. Also, all the parity bits can do is check if an error has occurred. It has no way to determine which bit is incorrect, so it is impossible to correct the error.

By contrast, Hamming codes are error-correcting codes. They are named after their inventor, Richard Hamming. Hamming was working for Bell labs in the 1940s. He grew frustrated with how often he had to restart his programs because a read error occurred. He developed Hamming codes as a way of detecting and correcting errors. Hamming codes can detect and correct single-bit errors or can detect (but not correct) up to two simultaneous bit errors.

Hamming codes require $O(lg(n))$ parity bits for n data bits. Each parity bit checks some (but not all) of the data bits. If an error occurs in a data bit, all the parity bits that checked that data bit will show the error, allowing us to uniquely determine where the error occurred.

The following general algorithm uses a one based numbering scheme for the bit positions. The parity bits are at powers of two to ease calculation. Without too many problems, however, the scheme may be applied with the data and parity bits located at any position.

All bit positions that are powers of two are used as parity bits. (positions 1, 2, 4, etc.).
All other bit positions are for the data to be encoded. (positions 3, 5, 6, 7, etc.).

Each parity bit calculates the even parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips.

- Position 1 *(n=1)*: is applied to every other bit: e.g. Check one bit; skip one bit; check one bit; skip one bit.
  Bits checked = 1,3,5,7,...
- Position 2 *(n=2)*: skip 1 bit, then check two, skip two, check two, etc.
  Bits checked = 2,3,6,7,...
- Position 4 *(n=4)*: skip 3 bits, then check 4 bits, skip 4 bits, check 4 bits, etc.
  Bits checked = 4,5,6,7,...

The general rule for position n: skip $n-1$ bits, check n bits, skip n bits, check n bits..., and so on.

In other words, the parity bit at position 2^{k-1} checks bits in positions having bit k set in their binary representation.

This general rule can be shown visually:

Bit position		1	2	3	4	5	6	7	
Encoded		p1	p2	d1	p3	d2	d3	d4	
Parity bit coverage	p1	X		X		X		X	···
	p2		X	X			X	X	
	p3				X	X	X	X	

Shown are 7 encoded bits (3 parity, 4 data) but the pattern continues indefinitely. The key thing about Hamming Codes that can easily be seen from visual inspection is that any given bit has a unique parity bit coverage. For example, the only data bit covered by p1 and p3 (and no other bits) is bit 5 (d2). It is this unique bit coverage that lets a Hamming Code correct any single bit error.

It also shows how a two-bit error can be detected but not corrected. For example, if bits 1 (p1) and 2 (p2) were flipped then this would be confused with bit 3 (d1) being flipped since the parity bit coverage of bit 3 is bits 1 and 2.

Hamming codes can be described in terms of the total number of bits per block and the number of data bits. Hamming(7,4) encodes 4 data bits into 7 bits by adding three parity bits. Basically, it implements the table above.

Let's work through an example:
Suppose we want to use Hamming (7,4) to encode the byte 1011 0001.
The first thing we will do is split the byte into two Hamming code data blocks, 1011 and 0001.
We expand the first block on the left to 7 bits: _ _ 1 _ 0 1 1.
The first missing bit (bit 1) is 0, because adding bits 3, 5 and 7 gives an even number (2).
The second missing bit (bit 2) is 1, because adding bits 3, 6 and 7 gives an odd number (3).
The last missing bit (bit 4) is 0, because adding bits 5, 6 and 7 gives an even number (2).
This means our 7 bit block is: 0 1 1 0 0 1 1

We expand the second block to 7 bits using similar logic, giving: 1 1 0 1 0 0 1

7 bits do not a byte make, so we add a leading 0 to each code block.

For those with a mathematical bent, expanding the data block is simply doing a matrix-vector multiplication modulo 2. The code generator matrix pre-mulitplies the 4 bit data block modulo 2 to give the 7 bit code block. (Note that the data block will be written as a column vector to do the multiplication.)

The code generator matrix for Hamming(7,4) is:

$$\mathbf{G} := \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Similarly, we can also use matrix multiplication to check and see if there is an error. Pre-multiply mod 2 the code block by H. If there are no errors, it should give a result of all zeroes.

$$\mathbf{H} := \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Let's work through our example using matrices.
We pre-multiply (mod 2) our first data block 1011 by G

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{X} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

If we pre-multiply our code block by H mod 2 we get: 0 0 0. This means that there are no errors.

We can recover our original data block by simply pulling out the data bits from the code block. The data bits are in locations 3, 5, 6, and 7, giving 1011. This can also be done using matrix multiplication, if you wish. What would the matrix look like?

Now suppose we send our code block to a file, or out across the Internet, and accidentally bit 5 gets flipped, giving: 0 1 1 0 **1** 1 1.

When we pre-multiply (mod 2) using H we get: 1 0 1. That means parity bits 1 and 4 are incorrect. (Note that bit 1 is from the first row and bit 4 is from the last row.) We add the 1 and 4, which tells us bit 5 is the incorrect bit. Thus, we can flip bit 5 back and recover our correct data.

Note that after multiplying by H, if only one bit is 1, then it is a parity bit that is incorrect, not a data bit.

Your Assignment
You are to implement a Hamming (7,4) code in Java. Your class should be named Hamming. It should contain the following public methods:

```
public static void encode (String inFileName, String outFileName)
```

This method should read in the input file one byte at a time, create two code blocks (with a leading 0 in each) and write them out to the output file.

```
public static void decode (String inFileName, String outFileName)
```

This method should take as an input as file that contains bytes of Hamming(7,4) code words. It should decode these bytes, pulling out the data bits and correcting errors as it goes. As discussed above, it is possible to have multiple errors in a single code block, in which case the program should print an error message but not try to correct the data.

What to Submit
Only electronic submission of your source code is required. Your instructor will provide a test driver to thoroughly test your class, so be certain to use the correct class name and method signatures.

Grading Considerations
Your program must compile and run to receive any points.
Good software engineering is expected. Use lots of comments and follow appropriate indentation, capitalization and variable naming standards.