Lab 13: Hashing & Bitwise Operations

In this lab you will use bitwise operations to explore an older method to calculate hash codes. It is designed to give you a more intuitive idea of the way binary math works. The class should be called `BitwiseHash`.

In the old days, division was too expensive to use frequently, but bitwise operations have always been very efficient. These days, since division and modulus are cheap, a hash code has its value reduced via the mod operation. For example, you might have a hash table with a size of 47—which is a prime number. To take the int hash code and reduce it to a value between 0 and 47, you first mod the hash code by 47, and then take the result's absolute value.

However, it used to be that hash tables usually had a length that was a power of 2, rather than a prime number. For example, a hash table might have a length of 256. In this case, you wouldn't just mod by 256, because this is the same thing as throwing away 3 bytes of a 4-byte number. (For example, the number 1234567890_{dec} is equivalent to $499602\text{D}2_{\text{hex}}$. Modding by 256 (or 100 in hexadecimal) is the equivalent of throwing the top part of the number away, and leaving just D2, or 210 in decimal. This loss of information could result in unbalanced hash tables.) Instead, you would separate the value into 4 8-bit components, and XOR them together. So in this case, you would XOR the numbers 49, 96, 02, and D2 together; the result is 0C.

For this lab, you will immediately enter a loop that receives inputs from the user. Quit if the user enters an empty string. Otherwise, calculate that string's hash code, and split up the number into 4 8-bit sections. Show the results of a bitwise AND, OR, and XOR for four these values. The catch is that all your outputs should be in binary, as shown in this sample output:

```
Give me a string, and I will calculate its hash and the bitwise and, or, and
xor of its 4 bytes.  Or hit enter to quit.

> cat
hash:00000000000000010111111111010110 and:00000000 or:11111111 xor:10101000
> dog
hash:00000000000000011000010100111100 and:00000000 or:10111101 xor:10111000
> hippopotamus
hash:11100011011000010100001000010001 and:00000000 or:11110011 xor:11010001
>
Goodbye!
```

Let's look at the math in more detail. `"cat"` yields a hashcode of 98262, which you can calculate easily using the hashCode() method. This number, in binary, is:

$$00000000\ 00000001\ 01111111\ 11010110_{\text{bin}}.$$

If we divide that into 4 pieces, we get 00000000 00000001 01111111 11010110. If we calculate 00000000 AND 00000001 AND 01111111 AND 11010110, the result is 00000000. The bitwise

OR result is 11111111, and the bitwise XOR result is 10101000.

A good first step to this lab would be to make a function called `makeBinaryString()`. This can take two arguments: a number, and the number of bits to display. So if you gave it 98262 and 32, it would return the string `"00000000000000010111111111010110"`. To do this, you'll need to know if any individual bit is a 1 or a 0. You can figure this out by left-shifting 1 by the correct number of places, and doing an AND operation on the number. For example, to know if the fifth digit is a 1 or a 0, just left-shift 1 by 5, to obtain 32, and AND 32 with the original number. The result will be either 0 or 32, depending on if that bit is 0 or 1, respectively.

Also, remember the difference between the >>> and >> operations. The former shifts in 0s, while the latter preserves sign. You will probably want to use >>> for this assignment.