Student Name: ______________________________
Student Name: ______________________________

Question #1

There are 26 characters in the alphabet we use for writing English.

1.1) What is the smallest number of bits needed to give each character a unique bit pattern?

We would need ceiling($\log_2 26$) = **5 bits** to represent all 26 characters, since 5 bits can represent up to 32 different values.

1.2) How many bits would we need to distinguish between upper- and lowercase versions of all 26 characters?

If we had upper- and lowercase versions, we'd have 52 characters, so we'd need ceiling($\log_2 52$) = **6 bits**. Another way to think of this is that if we had twice as many things to represent, we would need one additional bit.

Question #2

A collection of 8 bits is known as a "byte". Java has a "byte" datatype that is stored as an 8-bit two's complement binary number. What is the largest value that can be stored in a Java "byte"?

The largest value would be 01111111, which is **2^7 - 1, or 127**.

Question #3
Convert the following 2's complement binary numbers to decimal.

3.1) 00100001

This is positive since it starts with a 0, so we just need to calculate the magnitude:
$2^5 + 2^0 = 32 + 1 =$ **33**

3.2) 01011010

As above, we just need to calculate its magnitude, which would be
$2^6 + 2^4 + 2^3 + 2^1 = 64 + 16 + 8 + 2 =$ **90**

3.3) 10011001

This is negative since it starts with a 1, so we need to do the following to determine the magnitude:

- Subtract 1: 10011001 - 1 = 10011000
- Flip the bits: 10011000 → 01100111
- Now treat this as an unsigned int to get the magnitude: 01100111 → $2^6 + 2^5 + 2^2 + 2^1 + 2^0 = 64 + 32 + 4 + 2 + 1 = 103$
- So this is **-103**

3.4) 11111111

This is also negative (starts with 1) so to determine the magnitude we need to:

- Subtract 1: 11111111 - 1 = 11111110
- Flip the bits: 11111110 → 00000001
- So this is **-1**

Question #4

Convert the following decimal numbers to 8-bit 2's complement binary numbers.

4.1) 77

This is positive so we only need to get the magnitude as an 8-bit number.

77 % 2 = 1 → _ _ _ _ _ _ _ 1
77 / 2 = 38
38 % 2 = 0 → _ _ _ _ _ _ 0 1
38 / 2 = 19
19 % 2 = 1 → _ _ _ _ _ 1 0 1
19 / 2 = 9
9 % 2 = 1 → _ _ _ _ 1 1 0 1
9 / 2 = 4
4 % 2 = 0 → _ _ _ 0 1 1 0 1
4 / 2 = 2
2 % 2 = 0 → _ _ 0 0 1 1 0 1
2 / 2 = 1
1 % 2 = 1 → _ 1 0 0 1 1 0 1
1 / 2 = 0
0 % 2 = 0 → 0 1 0 0 1 1 0 1

4.2) 102

As above, since this is positive, we can just get the magnitude as an 8-bit number. Here, we're skipping the divide-by-2 part in the description but you still need it for the algorithm.

102 % 2 = 0 → _ _ _ _ _ _ _ 0
51 % 2 = 1 → _ _ _ _ _ _ 1 0
25 % 2 = 1 → _ _ _ _ _ 1 1 0
12 % 2 = 0 → _ _ _ _ 0 1 1 0
6 % 2 = 0 → _ _ _ 0 0 1 1 0
3 % 2 = 1 → _ _ 1 0 0 1 1 0
1 % 2 = 1 → _ 1 1 0 0 1 1 0
0 % 2 = 0 → **0 1 1 0 0 1 1 0**

4.3) -94

Since -94 is negative, we first need to get its magnitude, still using 8 bits:

94 % 2 = 0 → _ _ _ _ _ _ _ 0
47 % 2 = 1 → _ _ _ _ _ _ 1 0
23 % 2 = 1 → _ _ _ _ _ 1 1 0
11 % 2 = 1 → _ _ _ _ 1 1 1 0
5 % 2 = 1 → _ _ _ 1 1 1 1 0
2 % 2 = 0 → _ _ 0 1 1 1 1 0
1 % 2 = 1 → _ 1 0 1 1 1 1 0
0 % 2 = 0 → 0 1 0 1 1 1 1 0
Then flip the bits: 01011110 → 10100001
Then add 1: 10100001 + 1 = **10100010**

4.4) -18

Since -18 is negative, we first need to get its magnitude, still using 8 bits:
18 % 2 = 0 → _ _ _ _ _ _ _ 0
9 % 2 = 1 → _ _ _ _ _ _ 1 0
4 % 2 = 0 → _ _ _ _ _ 0 1 0
2 % 2 = 0 → _ _ _ _ 0 0 1 0
1 % 2 = 1 → _ _ _ 1 0 0 1 0
0 % 2 = 0 → _ _ 0 1 0 0 1 0
At this point, we can put 0s in front of the number to get it to 8 bits, so 00010010.
Then flip the bits: 00010010 → 11101101
Then add 1: 11101101 + 1 = **11101110**

4.5) -129

Here we can use the same approach as above:
129 % 2 = 1 → _ _ _ _ _ _ _ 1
64 % 2 = 0 → _ _ _ _ _ _ 0 1
32 % 2 = 0 → _ _ _ _ _ 0 0 1
16 % 2 = 0 → _ _ _ _ 0 0 0 1
8 % 2 = 0 → _ _ _ 0 0 0 0 1
4 % 2 = 0 → _ _ 0 0 0 0 0 1
2 % 2 = 0 → _ 0 0 0 0 0 0 1
1 % 2 = 1 → 1 0 0 0 0 0 0 1
0 % 2 = 0, so we're done.
Then flip the bits: 10000001 → 01111110
Then add 1: 01111110 + 1 = 01111111
But wait….! The value we're trying to represent is -129, but this binary number starts with a 0, indicating that it's positive!

It turns out that we *can't* represent -129 as an 8-bit two's-complement number. The smallest (most negative) number we can represent is -(2^7) = -128.

4.6) 0

This is a little tricky because 0 is neither positive nor negative, so it's hard to know whether we need to do the "flip the bits and add 1" thing.

Obviously, we can represent the magnitude as 00000000. Note, though, that if we do the "flip the bits and add 1" thing, we'd get:
00000000 → 11111111
11111111 + 1 = 100000000
Because we only want an 8-bit number, we'd just use the last 8 bits and get…. 00000000! Which is the same as before!

<u>Question #5</u>
Your friend tried to convert the decimal number 41 to unsigned binary and got 101101, and wants you to help determine whether it's correct.

5.1) Aside from redoing the conversion yourself, how else can you tell whether this is correct?
We can quickly check the work by trying to convert 101101 to decimal and seeing if we get 41. However, 101101 = 2^5 + 2^3 + 2^2 + 2^0 = 32 + 8 + 4 + 1 = 45, so this is incorrect.

5.2) If it's not correct, where do you think they might have gone wrong?
You may notice that the value (45) is 4 greater than it should be, so it's clear that there's an extra 4, and thus the fourth bit (counting from the left) should be 0, not 1.

If we look at the algorithm they may have used, it should go like this:
41 % 2 = 1 → _ _ _ _ _ _ 1
20 % 2 = 0 → _ _ _ _ _ 0 1
10 % 2 = 0 → _ _ _ _ 0 0 1
However, at this point, the bit in this place in your friend's solution is 1, not 0, so they may have made a mistake here.

<u>Question #6</u>
If the last digit of a 2's complement binary number is 0, then the number is even. If the last two digits of a 2's complement binary number are 00 (e.g., the binary number 01100), what does that tell you about the number?

This means that it's a multiple of four.

<u>Question #7</u>
The decimal number 9 is represented in unsigned binary as 1001. Given this, how could you quickly determine the representation of 72?

The trick here is realizing that 72 = 9 * 8.

Since 9 is 1001, then to get 72 = 9 * 8, which is 9 * 2^3, we just add three 0s to the end, so we get **1001000**.

This is analogous to multiplying a decimal number by 10^3 = 1,000. If we had 35 * 1,000, we just add three 0s to the end of 35 to get 35,000.

Question #8

Another binary representation of signed numbers is called **signed magnitude**, which is a simple representation in which the first bit of the number represents the sign (0 = positive and 1 = negative), and the remaining bits represent the magnitude.

For instance, for a six-bit signed magnitude number:

- 001100 would represent +12, just as it would in two's-complement
- 101100 would represent -12, since the first 1 indicates that it's negative, and the remaining bits (01100) represent the magnitude, which is 12

8.1) First, convince yourself that signed magnitude is different from two's-complement. We said that 101100 represents -12 as a signed magnitude number; what does it represent as a six-bit two's-complement number?

Since 101100 starts with a 1, we know it's negative as a two's-complement number. To get the magnitude, we need to:

- Subtract 1: 101100 - 1 = 101011
- Flip the bits: 101011 → 010100
- This means the magnitude is $2^4 + 2^2 = 20$
- So this is -20 as a two's complement number

8.2) What is an advantage of two's-complement over signed magnitude in terms of the number of distinct values that can be represented?

In signed magnitude, there are two representations of zero: 000000 and 100000.

However, in two's-complement, there is only one representation of 0, which is 000000. Note that 100000 represents -32 as a 6-bit two's-complement number.

Since we can only represent $2^6 = 64$ values with a 6-bit number, this means that signed magnitude can only represent 63 values because there are two representations of zero, whereas two's-complement can represent all 64.

Question #9

As we have also learned, it is possible to add binary numbers in the same way that we can add decimal numbers.

Calculate the sum of the following binary numbers:

1) Unsigned numbers 000101 + 001110 = __________

We can add these numbers in the same way that we'd add decimal numbers by lining them up and carrying values to the left as needed:

```
carry →    1 1
         0 0 0 1 0 1
       + 0 0 1 1 1 0
       _____________
         0 1 0 0 1 1
```

To check our work, we can convert these to decimal and see if we got the right answer:

1000101 → 5
2) 001110 → 14
3) 010011 → 19, which not coincidentally is 5 + 14. Yeah!

2) Signed 2-'2 Complement numbers: 0000 1000 (8) + 0110 0001 (97) = 0110 1001 (105)

3) Signed 2-'2 Complement numbers: 0000 1011 (11) + 1111 0101 (-11) = 0000 0000 (0)

For each of the above, think about how you can check if your answers above are correct. Do so to confirm.

See above.