Lab 8 (October 4 or October 5)

Instructions: Complete the steps below. Be sure to show your code to one of the lab TAs before you leave, so that you can receive credit for this lab. You must also upload a copy of all your source code (.java) files to the link on Blackboard by 11:59 PM on Thursday, October 5.

In class, we discussed binary (base 2) representation, and a simple algorithm for converting decimal (base 10) numbers into binary (or any other base). Our discussion was limited to positive integer values, however, which prevents us from representing negative integers (or floating-point values, but that's a story for another time). We also didn't go into detail regarding how arithmetic (for example, addition) is performed inside the computer on binary values. In this lab assignment, we will implement three Java methods to add binary numbers and to represent negative integers in binary.

1. Start by implementing a helper method named `pad()`. This method takes two arguments: a `String` representing a binary value (i.e., it only contains the characters '1' and '0') and a non-negative integer representing the desired length. If the length of the `String` argument is less than the integer argument, the method prepends additional '0' characters to the beginning of the `String` until it reaches the desired length, and then returns the modified string. If the `String` argument is already at (or longer than) the desired length, the method just returns the original `String`.

   For example, `pad("110", 5)` will return "00110", while `pad("101101", 3)` will return "101101" (because "101101" already has a length greater than or equal to 3).

2. Binary addition is performed much like standard (decimal) addition. We line up the two values vertically and aligned to the right, and add one column of digits (bits) at a time, working from right to left. If the two operands are not of equal length, then we "pad" the shorter one with leading zeros on the left side to match the length of the other operand.

   There are three basic scenarios to consider (four once we include the possibility of a "carry bit"):

   a. 0 + 0 = 0 (with no value carried to the next column)

   b. 0 + 1 = 1 (with no value carried to the next column)

   c. 1 + 1 = 0 with 1 carried to the next column (1 + 1 is 2, which is represented in binary as 10)

d. 1 + 1, plus a carry bit of 1, is represented as 1 with another 1 carried to the next column (1 + 1 + 1 is 3, which is represented in binary as 11)

Once we have processed the final (leftmost) column, if the carry bit is still 1, then we add it to the beginning (left side) of the result.

Example: 1101 + 111. The second operand has fewer bits, so we pad it with leading zeros to get 0111. Then we add:

```
  1101
+ 0111
------
```

The rightmost column contains two 1s, so we place a 0 in that column and carry 1 over to the next column:

```
    1
  1101
+ 0111
------
     0
```

The next-to-last column contains two 1s and a 0. 1 + 1 in binary is still 2 (or 10), so we place a 0 in that column and carry 1 over to the next column:

```
   1
  1101
+ 0111
------
    00
```

The next column contains three 1s. 1 + 1 + 1 in binary is 3 (or 11), so we place a 1 in that column and carry 1 over to the next column:

```
  1
  1101
+ 0111
------
   100
```

The last column contains two 1s and a 0. 1 + 1 + 0 in binary is 2 (or 10), so we place a 0 in that column and carry 1 over to the next (empty) column:

```
 1
  1101
+ 0111
------
  0100
```

We have completed adding the columns in the original numbers, but we have a leftover carry bit of 1. That bit just "drops down" to the front of our answer to get our final result: 10100 (our two operands translated to 13 and 7 in base 10; our sum translates to 20).

Implement a Java method named `addBinary()` that takes two `String` arguments (each representing a binary value) and returns a new `String` corresponding to the result of performing binary addition on those arguments. Before you begin, if one of the arguments is shorter than the other, call your `pad()` method from the previous step to extend it to the desired length.

Suggested pseudocode:

```
carry_bit = 0
sum = ""

if (operand_1 is shorter than operand_2):
   pad operand_1 to the length of operand_2
else if (operand_2 is shorter than operand_1):
   pad operand_2 to the length of operand_1

for index from (length of operand_1 – 1) down through 0:
   if (carry_bit + (value of operand_1[index]) + (value of operand_2[index]) == 0):
      sum = "0" + sum
      carry_bit = 0
   else if (carry_bit + (value of operand_1[index]) + (value of operand_2[index]) == 1):
      sum = "1" + sum
      carry_bit = 0
   else if (carry_bit + (value of operand_1[index]) + (value of operand_2[index]) == 2):
      sum = "0" + sum
      carry_bit = 1
   else: // a sum of 3 means write down a 1 and carry the other 1
      sum = "1" + sum
      carry_bit = 1

if (carry_bit == 1): // Deal with any leftover carry bit
   sum = "1" + sum

return sum
```

3. Now it's time to tackle the problem of negative numbers in binary. Clearly, computers must be able to handle negative integers; otherwise, we could never execute an instruction like

```
int x = -5;
```

They do so using a variant of binary known as *two's complement*. In two's complement, the leftmost bit is called the *sign bit*, and serves double duty. In addition to making up part of the number's value, if the sign bit is 1, then the overall value is negative; if the sign bit is 0, then the overall value is positive. For example, the two's complement value 1101 is negative (-3 in this case), while the two's complement value 0011 is positive (3 in this case). Two's complement values are always represented using a fixed number of bits, so we can use leading 0s to represent positive numbers. This means that there are multiple ways to represent the same value in two's complement, based on the fixed length. For example, 1101 represents the value -3 in 4-bit two's complement, while 11101 represents -3 in 5-bit two's complement.

Define a Java method named `toTwosComplement()` that takes two integer arguments: a base 10 value to be translated into two's complement representation, and a second integer representing the length of the two's complement representation. The method converts the first argument into a `String` (containing only 1s and 0s) of the specified length and returns that `String`.

Use the following algorithm to convert an integer value into two's complement form:

a. If the integer value is positive or zero:

    i. Translate the integer value into a binary string using repeated division by 2 (i.e., using the algorithm you discussed in class).

    ii. Use `pad()` to extend the binary string to the required length.

b. Otherwise (meaning that the integer value is negative):

    i. Translate the absolute value of the integer into a binary string (for example, given the integer value -6, translate 6 into a binary string).

    ii. Use `pad()` to extend the binary string to the required length.

    iii. ***Negate*** the binary string by inverting its value and adding 1 (using binary addition):

        1. To invert a binary string, create a new `String` of the same length where all the 1s have been replaced by 0s and all the 0s have been replaced by 1s. For example, inverting "0010101" would produce the string "1101010".

2. Add the binary value 1 to the inverted string from the previous step using your `binaryAdd()` method.

iv. If the negated binary string is greater than the specified length (due to an extra carry bit), remove the leading character (bit).

For example, suppose that we wish to convert -13 to 6-bit two's complement representation (meaning that our method call is `toTwosComplement(-13, 6)`). Positive 13 in binary is "1101"; padding this to a length of 6 gives us "001101". Since the original number was negative, we invert the bits of the padded binary representation to get "110010". Adding 1 with binary addition gives us a final value of "110011" (if we had exceeded 6 bits, we would have returned only the last 6 bits of the result).

4. Finally, write a small Java program that tests your methods via two tasks:

a. Read in two binary strings from the user, add them using binary addition, and print the result.

For example, 110 + 1101 = 10011. Likewise, 1001 + 1111 = 11000.

b. Read in a decimal (base 10) integer value from the user and a desired length in bits. Your program should calculate and display the two's complement representation of the original integer value.

For example, 34 in 8-bit two's complement is 00100010, while -305 in 10-bit two's complement is 1011001111.

Grading Guidelines: This lab is graded on a scale of 0-3 points, assigned as follows:

0 points: Student is absent or does not appear to have completed any work for the lab

1 point: Student has written only one or two methods, but the code does not compile or run at all due to errors.

2 points: Student has written (or attempted to write) all threes, but only one or two compile and run without error.

3 points: Student has written a complete program (with all three methods) that compiles and runs correctly, without any apparent errors.