CSE 101: Introduction to Computational and Algorithmic Thinking

Homework #2-2

Fall 2018

Assignment Due: Monday, December 10, 2018, by 11:59 PM

Getting Started

To help you get started on this assignment, we have given you a skeleton Python file named `32fp.py`. This file contains several *function stubs* and a few tests you can try out to see if your code seems to be correct. Stubs are functions that have no bodies (or have very minimal bodies). You must fill in the bodies of these functions for the assignments. **Do not, under any circumstance, change the names of the functions or their parameter lists.** The automated grading system will be looking for exactly those functions provided in `2-2.py`. **Please note that the test cases we provide you in the bare bones files will not necessarily be the same ones we use during grading!**

Directions

This assignment is worth a total of 24 points. The automated grading system will execute your solution several times, using different input values each time. Each test that produces the correct/expected output will earn 4 points. You must earn at least 18 points (75%) in order to pass (receive credit for) this assignment.

- At the top of the `32fp.py` file, include the following information in comments, with each item on a separate line:
  - your full name ***AS IT APPEARS IN BLACKBOARD***
  - your Stony Brook ID #
  - your Stony Brook NetID (your Blackboard username)
  - the course number (CSE 101)
  - the assignment name and number (Assignment #2-2)

⚠ Any functions that we tell you to complete (i.e., that we provide stubs for in the starter code) **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters, or that use different parameter names) can't be graded by our automated grading system, and may receive a grading penalty.

⚠ Be sure to submit your final work as directed by the indicated due date and time. Late work will not be accepted for grading. Work is late if it is submitted after the due date and time.

⚠ Programs that crash will likely receive a grade of zero, so make sure you thoroughly test your work before submitting it.

Representing Floating-Point Values in Binary

In class, we mentioned the fact that different types of data (integers, strings, etc.) are stored in the computer's memory in different ways. *Floating-point values* (numbers with a decimal point, like 3.14) are normally encoded in either 32 or 64 bits using a special format known as IEEE 754 representation (the "754" refers to the IEEE standards document that defines the format).

Converting a base 10 (decimal) value into 32-bit IEEE 754 floating-point format requires several steps. After translating the original value into binary,[1] we need to *normalize* the binary value by shifting the decimal point[2] so that it appears immediately after the first 1 in the binary representation.[3] The number of positions by which the decimal/binary point moves gives us the *exponent*, in the sense that, for example, 1101.010 is equivalent to 1.101010×2^3. This exponent (an integer in the range -127 through 128) is adjusted by adding a *bias value* of 127, which forces it into the non-negative range 0–255. This allows us to encode it as an unsigned 8-bit binary value between 00000000 and 11111111 (inclusive). The *mantissa* consists of the first 23 bits of the normalized number *after* the leading 1. The final 32-bit representation consists of a single *sign bit* (0 or 1, depending on whether the original number was positive or negative), followed by the 8-bit biased exponent, followed by the 23-bit mantissa (if the mantissa contains fewer than 23 bits, additional 0s are added to the end to fill out the required 23-bit length).

Consider the following two examples:

Example 1: Translate 71.25 into 32-bit floating-point representation.

71.25 is positive, so the sign bit will be 0.

71.25 in binary is 1000111.01

Normalizing this value requires us to shift the binary point 6 positions to the left, giving us an exponent of 6. The normalized value is 1.00011101×2^6.

The biased exponent is (6 + 127) or 133, which is 10000101 in unsigned binary (**NOT** two's complement!).

The mantissa is 00011101. Extending this to 23 bits gives us 00011101000000000000000.

Combining the sign bit, biased exponent, and mantissa gives us 0 + 10000101 + 00011101000000000000000, or 01000010100011101000000000000000, which can be represented in hexadecimal (base 16) as 428E8000.

[1]To convert a decimal fraction into binary, subtract successively smaller powers of 2, starting with 0.5 (which is 2^{-1}), until you reduce the fractional part to 0 (note that many decimal fractions can't be represented in a fixed number of bits). If the current power of 2 can be subtracted from the remaining value, subtract it and append 1 to your result; otherwise, append 0. For example, suppose that you have the fractional value 0.6875:

1. We subtract 0.5 (or 2^{-1}) from this value and place a 1 in the appropriate column. We have 0.1875 left to account for.
2. The next smaller power of 2 is 0.25 (or 2^{-2}). We can't subtract this from what's left, so we place a 0 in the next column.
3. The next smaller power of 2 is 0.125 (or 2^{-3}). Subtracting this from 0.1875 gives us 0.0625, and we place a 1 in the next column.
4. The next smaller power of 2 is 0.0625 (or 2^{-4}). Subtracting this from what's left gives us 0, so we place a 1 in the next column and stop the conversion process with 1011 after the decimal point.

[2]This is probably more appropriately called the *binary point* in this case

[3]Normalization is very similar to scientific notation, which represents very large (or very small) numbers as a set of significant digits multiplied by a power of 10. For example, we can represent the mass of a proton (in kilograms) as 1.672621×10^{-27}, rather than 0.000000000000000000000000001672621.

Example 2: Translate -2.3125 into 32-bit floating-point representation.

-2.3125 is negative, so the sign bit will be 1. We continue the conversion using the absolute value of the original number, or 2.3125.

2.3125 in binary is 10.0101

The normalized version of this value is 1.00101×2^1. The mantissa is 00101000000000000000000, and the exponent is 1.

The biased exponent is (1 + 127) or 128, which is 10000000 in unsigned binary.

The final result is 1 + 10000000 + 00101000000000000000000, or 11000000000101000000000000000000 (in hexadecimal: C0140000).

For this assignment, you will develop a Python program that translates floating-point values (in base 10) into 32-bit IEEE 754 binary form, and then into hexadecimal (assume that the input will never be exactly 0). The `32fp.py` starter code includes a potentially useful helper function for you. Follow the instructions below to complete the Python functions that will execute the main program logic.

Part I: Converting a Fraction Into Binary (4 points)

Start by completing the `fractionToBinary()` function, which takes a single string argument representing a fraction in base 10 (e.g., `".75"`). This function uses the algorithm below to create and return a string corresponding to the binary equivalent of this fraction (or the Python value `None` if the input does not contain exactly one period). You may assume that the argument always consists of a single period followed by one or more digits.

```
1. If the parameter (value) has no periods, or has more than one period, return None
2. Convert value from a string to a float
3. result ← the empty string
4. power ← 0.5 (the first negative power of 2)
5. While value is greater than 0:
     • If power ≥ value, subtract power from value and append "1" to result
     • Otherwise, append "0" to result
   Then divide power by 2 (using regular division, not floor division)
6. When the loop ends, return result
```

Examples:

Function Call	**Return Value**
`fractionToBinary("0.5")`	`1`
`fractionToBinary("0..25")`	`None`
`fractionToBinary("0.75")`	`11`
`fractionToBinary("0.0625")`	`0001`
`fractionToBinary("0.328125")`	`010101`

Part II: Shifting the Binary Point (6 points)

Next, complete the `shiftBinaryPoint()` function, which takes a string consisting of 1s and 0s. If the input string is malformed (it contains more than one period), the function returns the Python value `None` to indicate an error. Otherwise, the function returns a 2-element list (or tuple) containing the exponent value and the mantissa (in that order).

To shift the binary point, use a process similar to the following:

1. If the parameter contains more than one period, return `None`.
2. If the parameter does not contain any periods, add `".0"` to the end.
3. Split the parameter based on the period character. This will produce two strings.
4. If the first string contains a 1, locate the position of the first 1. Set the exponent to the number of characters in the first string after the first 1. Set the mantissa to all of the bits in the first string after the first 1, plus the entire second string. If the mantissa has more than 23 bits, set the mantissa to just the first 23 bits.
5. Otherwise, locate the position of the first 1 in the second string (call it `pos` for this description). Set the exponent to $-1 \times (\texttt{pos} + 1)$. Set the mantissa to all of the bits in the second string after `pos`. If the mantissa has more than 23 bits, set the mantissa to just the first 23 bits.
6. Finally, return a new list (or tuple) whose first element is the exponent (an integer) and whose second element is the mantissa (a string).

Examples:

Function Call	**Return Value**
`shiftBinaryPoint("01101.10100")`	`[3, "10110100"]`
`shiftBinaryPoint("1.001")`	`[0, "001"]`
`shiftBinaryPoint("11")`	`[1, "10"]`
`shiftBinaryPoint("1..1")`	`None`
`shiftBinaryPoint("0.011010101")`	`[-2, "1010101"]`

Part III: Generating a Biased Exponent in Binary (4 points)

Third, complete the `getBiasedExponent()` function, which takes a single integer argument. If the argument is less than -127 or greater than 128, this function returns the Python value `None` to indicate an error. Otherwise, the function adds 127 to its argument, translates it into an 8-bit binary value, and returns the result as a string.

Hint: Use Python's built-in `bin()` function, which transforms an integer into its binary equivalent (as a string beginning with "0b"). Just remove the leading "0b" with string slicing and add leading 0s until the string has exactly 8 characters.

Examples:

Function Call	**Return Value**
`getBiasedExponent(25)`	`10011000`
`getBiasedExponent(130)`	`None`
`getBiasedExponent(0)`	`01111111`
`getBiasedExponent(-3)`	`01111100`
`getBiasedExponent(-203)`	`None`

Part IV: Assembling the Binary Representation (4 points)

Fourth, complete the `assembleValue()` function, which takes three arguments (in order): a Boolean value indicating the original number's sign (`True` means negative, while `False` means positive), a string of bits representing the exponent, and a string of bits representing the mantissa.

If the original value was negative, the function creates a string consisting of the bit `"1"`, followed by the exponent, followed by the mantissa. Otherwise, the function returns a string consisting of the bit `"0"`, followed by the exponent, followed by the mantissa. The function should then pad this string with trailing 0s so that its length is exactly 32 characters before returning it.

Examples:

Function Call	Return Value
`assembleValue(False, "10000010", "011100")`	`01000001001110000000000000000000`
`assembleValue(False, "10000101", "00110101100")`	`01000010100110101100000000000000`
`assembleValue(True, "10000100", "000000110")`	`11000010000000011000000000000000`
`assembleValue(True, "10000101", "101011100")`	`11000010110101110000000000000000`

Part V: Putting the Pieces Together (6 points)

Finally, complete the `encode()` function, which takes a floating-point (decimal) number as its only argument. This function converts its argument into 32-bit floating-point representation and returns the answer as an 8-character hexadecimal (base 16) string (or `None` in case an error occurs).

Your solution strategy should resemble the following:

1. If the parameter is negative, set a variable to `True` and multiply the parameter by -1. Otherwise, set that variable to `False`.
2. Convert the parameter to a string. If it does not have a period, add `".0"` to the end (so 45 becomes 45.0).
3. Locate the position of the first period (the decimal point). Use Python's `bin()` function to convert the "whole" part of the number (everything prior to this position) into a string of bits (remember that `bin()` adds `"0b"` to the beginning of its result, so you need to remove that prefix). Save this result.
4. Call your `fractionToBinary()` function with a string consisting of a leading 0 followed by every character in the parameter from the first period onward (e.g., if the original value was 27.193, you would call `fractionToBinary()` with the value 0.193). Save this result as well.
5. If Step 4 results in the value `None`, return `None` to indicate an error. This should never happen in practice, but we will include it as an example of "defensive programming".
6. Call `shiftBinaryPoint()` with a string consisting of your variables fro Steps 3 and 4, separated by a period character (for example, `whole + "." + fractional`). If `shiftBinaryPoint()` returns `None`, return `None` as well to indicate an error.
7. Call `assembleValue()` with your sign variable from Step 1 and the values returned by `shiftBinaryPoint()` in the previous step. Pass the resulting 32-bit value to the `binToHex()` helper function and return the final hexadecimal string.

Examples:

Function Call	**Return Value**
`encode(77.375)`	`429AC000`
`encode(-32.375)`	`C2018000`
`encode(11.5)`	`41380000`
`encode(-18.25)`	`C1920000`
`encode(0.101)`	`3DCED916`
`encode(-21)`	`C1A80000`

Examples:

Function Call	Return Value