COL100 Assignment 2

Due date: Monday, 3 January 2021

All functions must be documented with comments giving a precise specification of what the function is supposed to do. For example:

```
def hasDivisor(n, a, b):
    # Returns True if and only if there exists a number i
    # between a and b which is a divisor of n.
    if a > b:
        # range is empty
        return False
    elif isDivisor(a, n):
        # n is divisible by a
        return True
    else:
        # recurse on the smaller range a+1 to b
        return hasDivisor(n, a+1, b)
```

The first comment in the above example, giving the specification of the function, is required. The other comments explaining the parts of the code are optional, but it is a good habit to add them.

You will need to submit two files for this assignment:

1. a document `<entryNumber>_assignment_2.pdf` document containing the mathematical definitions of the algorithms and their proofs of correctness, and
2. a Python file `<entryNumber>_assignment_2.py` in which the algorithm is implemented.

For example, if your entry number is 2021CS12345, your files should be called `2021CS12345_assignment_2.pdf` and `2021CS12345_assignment_2.py`.

Part 1

In the last question of Assignment 1, we asked you to multiply two 4-bit numerals by repeated addition. This is not the way computers actually perform integer multiplication, nor is it a particularly efficient algorithm. Instead consider the long multiplication method we have all learned to do by hand in decimal notation, e.g.

$$\begin{array}{cccc} & 1 & 2 & 4 \\ \times & & 2 & 6 \\ \hline & 7 & 4 & 4 \\ 2 & 4 & 8 & \\ \hline 3 & 2 & 2 & 4 \end{array}$$

The same approach can be applied to binary arithmetic as well, for example to calculate ten (1010 in binary) multiplied by eleven (1011 in binary):

$$\begin{array}{ccccccc} & & 1 & 0 & 1 & 0 & \\ & \times & 1 & 0 & 1 & 1 & \\ \hline & & 1 & 0 & 1 & 0 & \\ & 1 & 0 & 1 & 0 & & \\ 0 & 0 & 0 & 0 & & & \\ 1 & 0 & 1 & 0 & & & \\ \hline 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{array}$$

The answer, 1101110, is indeed one hundred and ten (sixty-four + thirty-two + eight + four + two).

1. Mathematically design an algorithm which, given two binary numerals $a_m a_{m-1} \ldots a_1 a_0$ and $b_n b_{n-1} \ldots b_1 b_0$, computes the binary numeral $c_p c_{p-1} \ldots c_1 c_0$ representing their product. Note that we have not specified a maximum size of these numerals. Your algorithm must follow the same digit-by-digit approach as long multiplication.

   Right now, do not worry about how the algorithm might be implemented in a programming language! Define it as a mathematical function that can

be evaluated manually by a human being. You should express it in terms of basic operations that are possible on sequences of bits, such as single-bit operations, and rearranging, removing, or adding bits. You can also assume that an addition operation is available, so you can take two binary numerals of arbitrary size and get their sum as a binary numeral $c_p c_{p-1} \ldots c_1 c_0 = a_m a_{m-1} \ldots a_1 a_0 + b_n b_{n-1} \ldots b_1 b_0$.

To develop this algorithm, you may need to introduce multiple helper functions to do simpler tasks; that is part of your job in this question. For each such helper functions you define, you must also provide a mathematically precise specification.

2. Prove the correctness of your algorithm and all its helper functions. For the helper functions, this just means proving that for all valid inputs, the output matches your specification.

   For the multiplication algorithm, correctness means proving that the algorithm actually does perform multiplication. Recall that a binary numeral $a_m a_{m-1} \ldots a_1 a_0$ represents the natural number $\sum_{i=1}^{m} 2^m a_m$. So, you have to prove that if your algorithm applied to two binary numerals $a_m a_{m-1} \ldots a_1 a_0$ and $b_n b_{n-1} \ldots b_1 b_0$ computes the binary numeral $c_p c_{p-1} \ldots c_1 c_0$, then $\sum_{i=1}^{p} 2^i c_i = \left(\sum_{i=1}^{m} 2^i a_i\right) \left(\sum_{i=1}^{n} 2^i b_i\right)$.

3. Implement your algorithm for 4-bit binary numerals as a Python function `mul4(a,b)`, which produces an 8-bit binary numeral. Just like in Assignment 1, both the input and output should be tuples of Boolean values in reverse order, e.g. 1011 will be represented as `(True,True,False,True)`.

   For example, the product shown earlier, $1010 \times 1011 = 1101110$, should be computed by your program as

   ```
   >>> mul4((False,True,False,True), (True,True,False,True))
   (False,True,True,True,False,True,True,False)
   ```

   In your program, you are allowed to use logical Boolean operators (`and`, `or`, `not`), equality testing (`==`, `!=`), conditional statements (`if`, `elif`, `else`), tuples, functions, and recursion. You can also reuse your functions from

Assignment 1. Anything else which has not been covered in the lectures, like loops and lists, should not be used.

Part 2

Now we that we have designed the algorithm, we should also analyze its computational complexity.

4. Determine the time and space complexity of the algorithm you designed in Question 1. Your answer should be in big O notation, and should depend on m and n, the lengths of the input numerals $a_m a_{m-1} \ldots a_1 a_0$ and $b_n b_{n-1} \ldots b_1 b_0$.

   For the time complexity, consider only the time required for single-bit operations, such as addition of three bits, multiplication of two bits, checking whether a bit is 0 or 1, etc. You may assume that each such operation takes one unit of time, and that rearranging, adding, or removing bits from a numeral does not require additional time.

   For the space complexity, assume that each bit requires one unit of space, so storing an m-bit numeral $a_m a_{m-1} \ldots a_1 a_0$ takes m space.

   Note that to complete the analysis, you may also need to analyze the time and space complexity analysis of binary addition. You should clearly state what addition algorithm you are considering here. You do not need to prove its correctness.

5. Traditionally when performing long multiplication, we first compute all the intermediate products (e.g. the numbers 744 and 248 in the decimal example on page 2) and then add them up afterwards. Instead, we could iteratively add up each product as soon as we compute it.

   (a) Redesign your algorithm using this iterative approach, so that it requires no deferred additions.

   (b) Implement this iterative algorithm as a Python function `mul4i(a,b)`.