Lab 4 From Decimal to Binary and beyond...

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/Lab4 on 3/20/2017

This lab is all about converting numbers to and from base 10 (where *most* humans operate) from and to base 2 (where virtually all computers operate...).

At the end of the lab, you'll extend this to *ternary*, or base-3, where fewer computers and humans, but many more aliens, operate!

Here's a header to start your `hw4pr1.py` file:

```
# CS5 Gold/Black, hw4pr1
# Filename: hw4pr1.py
# Name:
# Problem description: Binary <-> decimal conversions
```

Function #1 `isOdd( N )` warm-up function!

As background, we'll recall how to determine whether values are even or odd in Python:

- First, open up Python and create a new blank file named `hw4pr1.py` - feel free to start the file with the header above.
- Then, just to get started, **write a Python function** called `isOdd(n)` that accepts a single argument, an integer `n`, and returns `True` if `n` is odd and `False` if `n` is even. Be sure to return these values, not strings! The evenness or oddness of a value is considered its *parity*. You should use the `%` operator (the "mod" operator). Remember that in Python `n % d` returns the remainder when `n` is divided by `d` (assuming n >= 0). Here are two examples of `isOdd` in action:

```
In [1]: isOdd(42)
Out[1]: False

In [2]: isOdd(43)
```

```
Out[2]: True
```

Function #2 `numToBinary(N)`

This part of the lab motivates converting decimal numbers into binary form *one bit at a time*, which may seem *"odd"* at first...!

Quick overview: you will end up writing a function `numToBinary(N)` that works as follows:

```
In [1]: numToBinary(5)
Out[1]: '101'

In [2]: numToBinary(12)
Out[2]: '1100'
```

Starter code: If you'd like, we provide one starting point for your `numToBinary` function here. A useful first task is to write a docstring!

```
def numToBinary(N):
    """
    """
    if N == 0:
        return ''
    elif N%2 == 1:
        return   ______________________  + '1'
    else:
        return   ______________________  + '0'
```

Thoughts:

- Notice that this function is, indeed, handling only one "bit" (zero or one) at a time.
- We didn't use `isOdd`—this is OK.   (This way is a bit more flexible for when we switch to base 3!)
- Since we don't want leading zeros, if the input `N` is zero, it returns the empty string.

- *This means that numToBinary(0) will be the empty string.* This is both required and OK!
- If the input `N` is odd, the function adds `1`
- If the input `N` is even (`else`), the function adds `0`
- ***What recursive calls to `numToBinary`—and other computations—are needed in the blank spaces above?***

Hints!:

- You'll want to recurse by calling `numToBinary` on a smaller value.
- What ***value*** of `N` results when one bit (the rightmost bit) of `N` is removed? That's what you'll want!
- Remember that the `//` operator is integer division (with rounding down)
- Stuck? Check this week's class notes for additional details...
- *tfw* binary is *fleek*? Check out the gold notes' *fleek* version (which can easily extend to *any* base...)

More examples!:

```
In [1]: numToBinary(0)
Out[1]: ''

In [2]: numToBinary(1)
Out[2]: '1'

In [3]: numToBinary(4)
Out[3]: '100'

In [4]: numToBinary(10)
Out[4]: '1010'

In [5]: numToBinary(42)
Out[5]: '101010'

In [6]: numToBinary(100)
Out[6]: '1100100'
```

Function #3 `binaryToNum(S)`

Next, you'll tackle the more challenging task of converting from base 2 to base 10, ***again from right to left***. We'll represent a base-2 number as a string of 0's and 1's (bits).

Quick overview: you will end up writing a function `binaryToNum(S)` that works as follows:

```
In [1]: binaryToNum('101')
Out[1]: 5

In [2]: binaryToNum('101010')
Out[2]: 42
```

Starter code: If you'd like, we provide one starting point for your `binaryToNum` function here. A useful first task is to write a docstring!

```
def binaryToNum(S):
    """
    """
    if S == '':
        return 0

    # if the last digit is a '1'...
    elif S[-1] ==  '1':
        return   _____________________ + 1

    else: # last digit must be '0'
        return   _____________________ + 0
```

Thoughts:

- Remember that the input is a ***string*** named `S`.
- Notice that this function is, again, handling only one "bit" (zero or one) at a time, right to left.
- Reversing the action of the prior function, if the argument is an empty string, the function returns `0`. This is both required and OK!
- If the *last digit* of `S` is `'1'`, the function adds the value `1` to the result.
- If the *last digit* of `S` is `'0'`, the function adds the value `0` to the result. (Not strictly required, but OK.)
- ***What recursive calls to `binaryToNum`—and other computations—are needed in the blank spaces above?***

Hints!:

- You'll want to recurse by calling `binaryToNum` on a smaller string.
- How do you get the string of everything ***except*** the last digit?! (Use slicing!)
- When you recurse, the recursive call will return the **value** of the smaller string—***This will be too small a value***, but…
- What computation can and should you perform to make the value correct?
- Remember that the recursion is returning the value of a binary string *one bit shorter* (shifted to the right by one spot)—remember how that right-shift changes the overall value. ***Then undo that effect!***
    - You can either multiply or shift, but *be sure to do so* ***outside + after*** *the recursive call to binaryToNum*
- That's all you'll need (it's just one operation after the recursive call)!

More examples!:

```
In [1]: binaryToNum("100")
Out[1]: 4

In [2]: binaryToNum("1011")
Out[1]: 11

In [3]: binaryToNum("00001011")
Out[1]: 11

In [4]: binaryToNum("")
Out[1]: 0

In [5]: binaryToNum("0")
Out[1]: 0

In [6]: binaryToNum("1100100")
Out[1]: 100

In [7]: binaryToNum("101010")
Out[1]: 42
```

Functions #4 and #5 `increment(S)` and `count(S, n)`

Binary Counting!

In this problem we'll write several functions to do count in binary—***use the two functions you wrote above for this!***

Quick overview: you'll write `increment(S)`, which accepts an 8-character string `S` of 0's and 1's and returns the *next largest* number in base 2. Here are some sample calls and their results:

```
In [1]: increment('00000000')
Out[1]: '00000001'

In [2]: increment('00000001')
Out[2]: '00000010'

In [3]: increment('00000111')
Out[3]: '00001000'

In [4]: increment('11111111')
Out[4]: '00000000'
```

Thoughts:

- Notice that `increment('11111111')` should wrap around to the all-zeros string. This can be a special case (`if`).
- You don't need recursion here!
- Instead, use both of the conversion functions you wrote earlier in the lab! Here is pseudocode:
  - Let `n` = the numeric value of the input string S
  - Let `x = n + 1` (this is the increment!)
  - Convert `x` *back* into a binary string with your other converter!
  - Give a name, say `y`, to that newly created binary string...
  - At this point, you're almost finished!

Hints!:

- The tricky part is ensuring you have enough leading zeros (in front of `y`, if you used that name…)

- You could add 42 zeros in front of `y` with `'0'*42 + y`
- Now, consider the *correct* number of zeros to add…it will involve the `len` function!

Next function: here, you'll use the above function to write `count(S, n)` that accepts an 8-character binary input string and then begins counts `n` times upward from `S`, printing as it goes. Here are some examples:

```
In [1]: count("00000000", 4)
00000000
00000001
00000010
00000011
00000100

In [2]: count("11111110", 5)
11111110
11111111
00000000
00000001
00000010
00000011
```

Thoughts:

- This means your function will print a total of `n+1` binary strings.
- You should use the Python `print` command, since nothing is being returned. We're only printing to the screen.
- You ***do*** need recursion here. What are the base case and the recursive case? See below for hints:

Hints!:

- Use the `increment` function!
- Your base case involves `n` (what's the "simplest" value of `n`?)
- Your recursive case will involve `n-1`.

Base-3: ternary and balanced ternary

There are 10 types of people in the world: those who know ternary, those who don't, and those who think this is a binary joke.

Functions #6 and #7 `numToTernary(N)` and `ternaryToNum(S)`

"Ordinary" Ternary

For this part of the lab, we extend these representational ideas from base 2 (binary) to base 3 (ternary). Just as binary numbers use the two digits 0 and 1, ternary numbers use the digits 0, 1, and 2. Consecutive columns in the ternary numbers represent consecutive powers of *three*. For example, the ternary number

`1120`

when evaluated from right to left, evaluates as `1` twenty-seven, `1` nine, `2` threes, and `0` ones. Or, to summarize, it is `1*27 + 1*9 + 2*3 + 0*1 == 42`.

In a comment or triple-quoted string, explain what the ternary representation is for the value `59`, and why it is so.

Use the thought processes behind the conversion functions you have already written to create the following two functions:

- `numToTernary(N)`, which should return a ternary string representing the value of the argument `N` (just as `numToBinary` does)
- `ternaryToNum(S)`, which should return the value equivalent to the argument string `S`, when `S` is interpreted in ternary.

Hints!:

- We're not providing starter code here, but…
- Base your solution from the corresponding functions `numToBinary` and `binaryToNum`!
- In fact, copying-and-pasting those functions (and then changing as needed) is a great strategy here.

Examples:

```
In [1]: numToTernary(42)
Out[1]: '1120'

In [2]: numToTernary(4242)
Out[2]: '12211010'

In [3]: ternaryToNum('1120')
Out[3]: 42

In [4]: ternaryToNum('12211010')
Out[4]: 4242
```

Finale! f'ns #8 and #9 `balancedTernaryToNum(S)` and `numToBalancedTernary(N)`

Balanced Ternary

It turns out that the use of *positive* digits is common, but not at all necessary. A variation on ternary numbers, called *balanced ternary* uses three digits:

- `+` (the plus sign) represents `+1`
- `0` represents zero, as usual
- `-` (the minus sign) represents `-1`

This leads to an unambiguous representation using the same power-of-three columns as ordinary ternary numbers. For example,

```
+0-+
```

can be evaluated, from right to left, as `+1` in the ones column, `-1` in the threes column, `0` in the nines column, and `+1` in the twenty-sevens column, for a total value of `1*27 + 0*9 -1*3 + 1*1 == 25`.

For this problem, write functions that convert to and from balanced ternary analogous to the base-conversions above:

- `balancedTernaryToNum(S)`, which should return the decimal value equivalent to the balanced ternary string `S`
- `numToBalancedTernary(N)`, which should return a balanced ternary string representing the value of the argument `N`

Again, a good strategy here is to start with copies of your `numToTernary` and `ternaryToNum` functions, and then alter them to handle balanced ternary instead.
Here are some examples with which to check your functions:

```
In [1]: balancedTernaryToNum('+---0')
Out[1]: 42

In [2]: balancedTernaryToNum('++-0+')
Out[2]: 100

In [3]: numToBalancedTernary(42)
Out[3]: '+---0'

In [4]: numToBalancedTernary(100)
Out[4]: '++-0+'
```

As a hint, consider that switching from a digit of value 2 to a digit of value -1 ***actually decreases the value of N by 3***! To avoid changing the overall value of N, you'll have to get that three back—by adding it back in!

Though binary is the representation underlying all of today's digital machines, it was not always so—and who knows how long binary's predominance will continue? Qubits are lurking!

Submit!

When you're finished with the lab (or the time is up!), go ahead and submit your `hw4pr1.py` file to the submission server.

The rest of the homework will involve some additional conversion and compressions (image compression, in particular).

If you're considering working ahead, it's true that Gold's first four functions of `hw4pr2.py` require no additional background ...