Base Conversion

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

The Ingredients

Remember that at this point in the course we are still using the functional-programming paradigm. If you happen to know about "for" loops or "while" loops or other constructs that we have not yet discussed, you should NOT use them here. Similarly, you should not use any fancy python features or special-purpose packages. Everything you need for these problems has been covered. However, there are a few built-in functions, described below, that you will want to use:

- `ord(C)`

  is a function that takes a character as input and returns the integer representation (ASCII value) of that character.

- `chr(N)`

  is a function that takes an integer as input and returns the character represented by that integer (in ASCII).

- `str(N)`

  is a function that takes an integer as input and returns the string version of that integer.

- `int(S)`

  is a function that takes a string of digits as input and returns the integer version of that string.

- `map`

  Look this one up. We use it frequently.

- `reduce`

  This is a powerful built-in function that you may find useful.

Another handy thing to keep in mind in this assignment is that the * operator can be used with strings. For example:

```
>>> 3*"spam"

'spamspamspam'
```

```
>>> -1*"spam"

''
```

Computing with base-2 (binary)

The goal of this part of the problem is to motivate a ***right-to-left*** conversion of decimal numbers into binary form.

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

```
>>> numToBinary(5)

'101'

>>> numToBinary(12)

'1100'
```

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

- First, **write a Python function** called `isOdd(n)` that takes as input an integer `n` and returns the value `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`. Here are two examples of `isOdd` in action:

  ```
  >>> isOdd( 42 )

  False

  >>> isOdd( 43 )

  True
  ```

- Next, let's revisit the process of converting from base 10 to base 2. In particular, we'll walk through the process of converting the number 42 in base 10 to a number in base 2. The *left-to-right* way to do this is to observe that the largest power of 2 that occurs in 42 is 32. So, we write down a 1 in the "32's place". Now, we have 42-32=10 remaining. The next lower power of 2 after 32 is 16. But 16 doesn't go into 10, so have a 0 in the "16's place". Now what? **In a comment in your file,** under your `isOdd` function, include the complete base-2 representation of the number 42.
- The method of converting from base 10 to base 2 that we just looked at went from *left to right*. That is, we computed the most-significant bit - the largest power of 2 - first. Then we wrote down

less-significant bits - lower powers of 2 - afterwards. This is **not** the only way to do the conversion!

- In fact, a program to do it from *right to left* is both shorter and more "elegant." This is what we will do next. To get started with this right-to-left conversion from decimal to binary, here's a question for you to answer: If you are given an ***odd*** base-10 number, what will the least-significant bit - the rightmost bit - be in its base-2 representation? Similarly, if you are given an ***even*** base-10 number, what will the least-significant bit - the rightmost bit - be in its base-2 representation? **In 1-2 sentences in a comment in your file, explain your answer to these two questions.**
- Next, suppose we have a base-2 number and we *eliminate* the least-significant bit (the rightmost digit). What does this do to the value of the original number? For example, if we start with`1010` and eliminate the `0` at the end, we get `101`. Similarly, if we start with `1011` and eliminate the `1` at the right, we get `101`. **Briefly explain how the value of the original number is changing** in another comment in your file. Hint: if you're not sure, read the next question for a hint...
- Imagine that we have a number N written in base 10 (decimal). Let Y denote N/2, where we round down if N is odd (this is Python's ordinary *integer* division). If we *already had* the base-2 representation of Y, perhaps from recursion, how would this allow us to easily find the base-2 representation of N? *(You'll need two answers here: one in the case that N is odd and one in the case that N is even.)* **Briefly explain your answer.**
- We *can* get the binary representation of N/2 by *using recursion* to call `numToBinary` on that value! Thus, we have all of the pieces needed to convert a number from base 10 to base 2 using the modulus operator `%`, division, and recursion. (You can use the shift operator `>>` instead of division, if you like.) Next, **write a Python function** `numToBinary(N)`. You should assume that its input `N` will be `>=` 0. `N` will be a normal, base-10 integer. `numToBinary` should return a **string** that represents the number `N` in base 2. If `N` equals zero, however, your `numToBinary` function should return **the empty string**, not the string `'0'`. This is important as a base case!

  Here is some sample input and output:

```
>>> numToBinary(0)
''
>>> numToBinary(1)
'1'
```

```
>>> numToBinary(4)
'100'
>>> numToBinary(10)
'1010'
>>> numToBinary(42)
'101010'
>>> numToBinary(100)
'1100100'
```

Hints: Recursion will be helpful here, as the above analysis suggests. If you're stuck, consider the following skeleton for your approach:

```
if N==0:
   return ''
elif isOdd(N):
   return numToBinary( ...
else:
   return numToBinary( ...
```

Here, you will need to carefully choose two things: what inputs to pass to the recursive calls *and* the additional work needed after the recursive calls.

- Next, let's convert 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. **Write a Python function** called `binaryToNum(S)` that takes as input a string `S` of 0's and 1's representing a base-2 number and returns the corresponding integer in base 10. Again, the exception is that an input of the empty string should return the numeric value `0`. In this case, input strings of zeros should also output the integer `0`. Here are some examples:

```
>>> binaryToNum("100")
4
```

```
>>> binaryToNum("1011")
11
>>> binaryToNum("00001011")
11
>>> binaryToNum("")
0
>>> binaryToNum("0")
0
>>> binaryToNum("1100100")
100
>>> binaryToNum("101010")
42
```

Hint: you will *still* want to start at the **right side** (the least-significant bit) of the binary number that is input as a string. Then, work your way to the left using recursion. Note that you can convert any string or substring, such as `S[-1]`, to its numeric, integer value with the function `int()`. For example, `int('7') = 7` (note that one value is a string and the other is not!), and `int(S[-1])` gives the numeric value of the last character of `S`. **Warning!** The input to `binaryToNum` is a **string**! Be sure any comparisons of it with other values are also with strings, for example,

```
if S[-1] == '0':
```

It won't work to compare `S[-1]` with the integer `0`!

Binary Counting!

In this problem we'll write several functions to do counting in binary using increasingly fancy techniques.

- To get started, **write a Python function** called `increment(S)` that takes as input an 8-bit string `S` of 0's and 1's and returns the *next largest* number in base 2. Here is some sample input and output:

```
>>> increment('00000000')
'00000001'
>>> increment('00000001')
'00000010'
>>> increment('00000111')
'00001000'
>>> increment('11111111')
'00000000'
```

Notice that `increment('11111111')` should wrap around to the all-zeros string. **Hint**: use both of the conversion functions you wrote earlier in the lab! Consider how could you use the function `len()` and string multiplication with `*` to make sure that the output has enough leading zeros?

- Now use your `increment` function to write a function called `count(S, n)` that takes an 8-bit binary string as input and begins counting up by one from the input `S` for `n` increments, printing each number as it goes. This means it will print a total of `n+1` binary strings. You should use the python `print` command, since nothing is being returned in this case—just output to the screen. Here are some examples:

```
>>> count("00000000", 4)
00000000
00000001
00000010
00000011
00000100
>>> count("11111110", 5)
11111110
11111111
```

```
00000000

00000001

00000010

00000011
```

Computing in 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.

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, when evaluated from right to left, evaluates as `0` ones, `2` threes, `1` nine, and `1` twenty-seven. Or, to summarize, it is `0*1 + 2*3 + 1*9 + 1*27 == 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 output a ternary string representing the value of the input `N` (just as `numToBinary` does)
- `ternaryToNum(S)`, which should output the value equivalent to the input string `S`, when `S` is interpreted in ternary.

Here are a pair of examples to check for each:

```
>>> numToTernary(42)

'1120'

>>> numToTernary(4242)

'12211010'

>>> ternaryToNum('1120')
```

```
42

>>> ternaryToNum('12211010')

4242
```