Gold Problem 2: Conversions and Compressions

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/CleverConversionsGold on 3/17/2017

65 points (individual or pair)   filename: `hw4pr2.py`

These problems are all about the base - of numbers' representation - and converting or computing with them.

To begin, this problem will ask you to convert from base 10 to other bases and vice versa. Because bases higher than base 10 require that new "digits" be introduced, we'll stick to bases that are between 2 and 10.

Function #1: `numToBaseB( N, B )`

Write a Python function called `numToBaseB( N, B )` that takes as input a non-negative integer N and a base B (between 2 and 10 inclusive); it should returns a string representing the number N in base B.

You don't need to check to be sure that `B` is between 2 and 10 -- *we* will ensure those are the only values we test.

Also, remember that *integer-division* (no fractional portions) is what's needed here. In Python, integer division rounds down and uses `//`, e.g.,

```
18//7 == 2
```

Here are some sample runs -- be sure the first two work! Several thoughts and hints are below:

```
In [1]: numToBaseB( 3116, 9)
Out[1]: '4242'

In [2]: numToBaseB( 141474, 8)
Out[2]: '424242'
```

```
In [3]: numToBaseB( 42, 8 )
Out[3]: '52'

In [4]: numToBaseB( 42, 5 )
Out[4]: '132'

In [5]: numToBaseB( 42, 10 )
Out[5]: '42'

In [6]: numToBaseB( 42, 2 )
Out[6]: '101010'

In [7]: numToBaseB(4, 2)
Out[7]: '100'

In [8]: numToBaseB(4, 3)
Out[8]: '11'

In [9]: numToBaseB(4, 4)
Out[9]: '10'

In [10]: numToBaseB(0, 4)
Out[10]: ''                  # notice the empty string for an
input N of 0 !

In [11]: numToBaseB(0, 2)
Out[11]: ''                  # notice the empty string for an
input N of 0 !
```

Thoughts:

- Remember, that your function is returning a **string**, and not a numeric value!
- The `numToBin` example from Thursday's slides is probably the best place to start...
- Ask yourself... what has to change in order to output base `B` instead of base 2

Hints:

- Your code ***must*** output the empty string when the input value of `N` is `0`. (This avoids leading zeros!)
- Here are the Python functions for converting back and forth between strings and numbers:

- `str(x)` returns the string representation of the number (integer) `x`.
- `int(s)` returns the integer value of the string `s`. If `s` doesn't represent an int, Python stops with an error.

Function #2: `baseBToNum( S, B )`

Naturally, we'd like to do the opposite conversion as well!

To that end, write a Python function called `baseBToNum(S, B)` that takes as input a string S and a base B where S represents a number in base B where B is between 2 and 10 inclusive. `baseBToNum` should then return an integer in base 10 representing the same number as `S`.

You don't need to check to be sure that `B` is between 2 and 10 -- we will ensure those are the only values we test.

Here are some sample runs - be sure the first two work (they're the largest examples we'll use):

```
In [1]: baseBToNum( '5733', 9 )
Out[1]: 4242

In [2]: baseBToNum( '1474462', 8 )
Out[2]: 424242

In [3]: baseBToNum( '222', 4 )
Out[3]: 42

In [4]: baseBToNum( "101010", 2 )
Out[4]: 42

In [5]: baseBToNum( "101010", 3 )
Out[5]: 273

In [6]: baseBToNum( "101010", 10 )
Out[6]: 101010

In [7]: baseBToNum("11", 2)
Out[7]: 3

In [8]: baseBToNum("11", 3)
Out[8]: 4
```

```
In [9]: baseBToNum("11", 10)
Out[9]: 11

In [10]: baseBToNum("", 10)
Out[10]: 0                        # the empty string should
return 0
```

Thoughts:

- Remember, that your function is returning a **number**, and taking a string as its input `S`!
- The `binToNum` example from Thursday's slides is probably the best place to start... .
- Again, the key is to ask yourself... what has to change in order to output base `B` instead of base 2?

Hints:

- Your `baseBToNum` function should output `0` when `S` is the empty string.
- As usual, the rightmost character of the string will be the "one's column," the *least* significant digit of the number in base B.
- But this means that the value of the rightmost character is simply `int( S[-1] )`!
- Here are the Python functions for converting back and forth between strings and numbers:
  - `str(x)` returns the string representation of the number (integer) `x`.
  - `int(s)` returns the integer value of the string `s`. If `s` doesn't represent an int, Python stops with an error.

Function #3: `baseToBase(B1,B2,s_in_B1)`

Now, we can assemble what we've written to write a function called called `baseToBase(B1,B2,s_in_B1)` that takes three inputs: a base `B1`, a base `B2` (both of which are between 2 and 10, inclusive) and `s_in_B1`, which is a string representing a number in base `B1`.

Then, your `baseToBase` function should return a string representing the same number in base `B2`.

Here is some sample input and output:

```
In [1]: baseToBase(2, 10, "11")    # 11 in base 2 is 3 in base
10...
Out[1]: '3'

In [2]: baseToBase(10, 2, "3")     # 3 in base 10 is 11 in base
2...
Out[2]: '11'

In [3]: baseToBase(3, 5, "11")     # 11 in base 3 is 4 in base
5...
Out[3]: '4'

In [4]: baseToBase( 2, 3, '101010' )
Out[4]: '1120'

In [5]: baseBToNum( '1120', 3 )
Out[5]: 42

In [6]: baseToBase( 2, 4, '101010' )
Out[6]: '222'

In [7]: baseToBase( 2, 10, '101010' )
Out[7]: '42'

In [8]: baseToBase( 5, 2, '4321' )
Out[8]: '1001001010'

In [9]: baseToBase( 2, 5, '1001001010' )
Out[9]: '4321'
```

Thoughts:

- Don't rewrite any conversions at all! Instead, convert to decimal and then back to the desired final base!

Hints:

- First *use* `baseBToNum` to get the ordinary, decimal number for `s_in_B1`. Give it a name.
- Then, use the `numToBaseB` function to convert that value to the appropriate base!

Function #4: `add(S,T)`

Here's a short problem that puts what you've written to use!

Write a function, called `add(S,T)` , that takes two binary strings `S` and `T` as input and returns their sum, ***also in binary***.

We encourage you to do this by converting the two binary strings to two base 10 numbers, add the two numbers, and then convert the resulting sum back into base 2!

Here is some sample input and output:

```
In [1]: add("11", "1")
Out[1]: '100'

In [2]: add("11", "100")
Out[2]: '111'

In [3]: add("110", "11")
Out[3]: '1001'

In [4]: add("11100", "11110")
Out[4]: '111010'

In [5]: add("10101", "10101")
Out[5]: '101010'
```

Function #5: `addB(S,T)`

Above, `add` shows one way of adding two binary numbers: first convert them to base 10, add those two base 10 numbers, and then convert the result back to binary.

In this problem, however, you will implement a different, more direct, method for adding two binary numbers, using the "elementary-school" algorithm we used in class:

```
  101110
  100111
--------
```

which, after the addition would look like this:

```
    111
   101110
   100111
---------
  1010101
```

Here the "carry" bits are in blue.

For this problem, write a Python function called `addB(S,T)` that takes two strings as input. These strings are the representations of binary numbers.

Your `addB` function should return a new string representing the sum of the two input strings.

The sum needs to be computed using the "elementary-school" binary addition algorithm, shown above and in class, and ***not using base conversions***. That is - this is *purely syntactic* addition!

Hints

- You'll need at least two base cases. A base-case question to consider is *what if `S` had no characters, but `T` still did have characters* - what would be the correct string to return?
- The "carry" case is another tricky part of this problem.
- When you need to carry, you will have THREE strings to add: (1) the rest of `S` (`S[:-1]`), (2) the rest of `T`, and the additional carry string, which is simply `'1'`.
- The key is to use `addB` ***twice***: once to add the carry string with one of the two "rests," and a second time to add the first result to the other of the two "rests"!

Here's a starting point:

```
def addB(S,T):
    """ docstring - please include """
    # base cases!

    # There will be four recursive cases - here is the first
one:
    if S[-1] == '0' and T[-1] == '0':
        return addB( S[:-1], T[:-1] ) + '0'   # 0 + 0 == 0

    # three more recursive cases still to specify...
```

Here is some sample input and output:

```
In [1]: addB("11100", "11110")
```

```
Out[1]: '111010'

In [2]: addB("10101", "10101")
Out[2]: '101010'

In [3]: addB("11", "1")
Out[3]: '100'

In [4]: addB("11", "100")
Out[4]: '111'
```

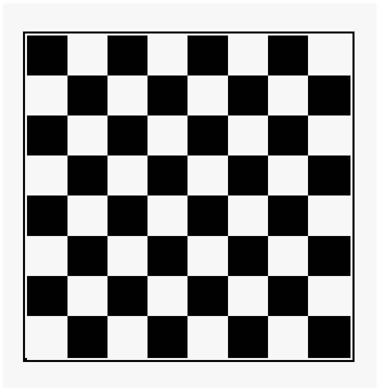
Functions #6 and #7: `compress( I )` and `uncompress( C )`:

Run-length image compression

Up to now we've explored how numbers are symbols are represented in binary, but in this problem we'll explore the representation of images using 0's and 1's.

For this part you'll write two functions, `compress( I )` and `uncompress( C )`, along with one or more helper functions. You should put these functions solutions in ***the same file***, `hw4pr2.py` along with the others above.

Let's begin by considering just 8-by-8 black-and-white images such as the one below:



Each cell in the image is called a "pixel". A white pixel is represented by the digit 0 and a black pixel is represented by the digit 1. The first digit represents the pixel at the top left corner of the image. The next digit represents the pixel in the top row and the second column. The eighth bit represents the pixel at the right end of the top row. The next bit represents the leftmost pixel in the second row and so forth. Therefore, the image above is represented by the following binary string of length 64:

```
'10101010010101011010101001010101101010100101010110101010010101010
1'
```

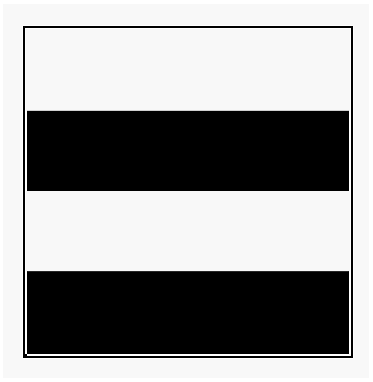
Another way to represent that same string in python is

```
'1010101001010101'*4
```

The background story

So now what? Here's the gratuitous background story: You've been hired by MASA ("Mudd Air and Space Administration"). MASA has a deep space satellite that takes 8-by-8 black-and-white images and sends them back to earth as binary strings of 64 bits as described above. In order to save precious energy required for transmitting data, MASA would like to "compress" the images sent into a format that uses as few bits as possible. One way to do this is to use the **run-length encoding algorithm**.

Imagine that we have an image that looks like this, for example:



Using our standard sequence of 64 bits, this image is represented by a binary string beginning with 16 consecutive 0's (for two rows of white pixels) followed by 16 consecutive 1's (for two rows of black pixels) followed by 16 consecutive 0's followed by 16 consecutive 1's.

Run-length encoding (which, by the way, is used as part of the JPEG image compression algorithm) says: Let's represent that image with the code "16 white, 16 black, 16 white, 16 black". That's a much shorter description than listing out the sequence of 64 pixels "white, white, white, white, ...".

Run-length encoding

In general, our run-length coding represents an image by a sequence (called a "run-length sequence") of 8-bit bytes:

- The first bit of each byte represents the `bit` that will appear next in the image, either `0` or `1`.
- The final seven bits contain the number *`in binary`* of those bits that appear consecutively at the current location in the image.

Notice that this run-length encoding will result in a relatively small number of bits to represent the 4-stripe image above. However, it will do very badly (in terms of the number of bits that it uses) in representing the checkerboard image that we looked at first. In general, run-length encoding does a good job "compressing" images that have large blocks of solid color. Fortunately, this is true of many real-world images, such as the images that MASA gets, which are mostly white with a few black spots representing celestial bodies.

Function #6: `compress(S)`

Whew! So here's your task.

Write a function called `compress(S)` that takes a binary string `S` of length less than or equal to 64 as input and returns another binary string as output. The output binary string should be a run-length encoding of the input string, as described above.

You may need a helper function or two - you may name these whatever you like. Also, you may want to copy a function or two from the previous hw problem and/or from the lab and/or the in-class "Quiz" this week!

Function #7: `uncompress(C)`

Next, write a function called `uncompress(C)` that "inverts" or "undoes" the compressing in your `compress` function.

That is, `uncompress(compress(S))` should give back `S` .

Again, helper functions are OK, as is using this week's previous problems and/or lab code you've written.

Here are a couple of examples of `compress` and `uncompress` in action:

```
In [1]: compress( 64*'0' )
Out[1]: '01000000'

In [2]: uncompress( '10000101' )   # 5 1's
Out[2]: '11111'

In [3]: compress( '11111' )
Out[3]: '10000101'

In [4]: Stripes = '0'*16 + '1'*16 + '0'*16 + '1'*16

In [5]: compress(Stripes)
Out[5]: '00010000100100000001000010010000'

In [6]: uncompress('00010000100100000001000010010000')
Out[6]:
0000000000000000111111111111111100000000000000001111111111111111
```

Submitting

Be sure to submit your working functions as `hw4pr2.py` at the submission site.

Not imaged out?

Try the image-processing extra credit, if you'd like!