More Basic Conversion

In this problem we'll convert from base 10 to arbitrary bases and vice versa. However, because bases higher than 10 require that new "digits" be introduced, in this problem we'll stick to bases that are between 2 and 10 (*unless* you do the bonus part described at the very end of this page!)

Part 1

Write a Python function called `numToBaseB(N, B)` that takes as input a non-negative (0 or larger) integer N and a base B (between 2 and 10 inclusive) and returns a string representing the number N in base B. (The rightmost digit of the string is the least significant digit of the number in base B.) Your code should output the string '0' when the input value of `N` is `0`.

Remember, that your function is returning a **string**, and not a numeric value. Here are the Python functions for converting back and forth between strings and numbers:

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

Here are some sample runs:

```
>>> numToBaseB(4, 2)
'100'
>>> numToBaseB(4, 3)
'11'
>>> numToBaseB(4, 4)
'10'
>>> numToBaseB(0, 4)
'0'
>>> numToBaseB(0, 2)
'0'
```

Notice that the output of `numToBaseB` never has leading 0's. For example, when converting the base-10 number 4 to base 2, `numToBaseB` returned '100' and not '0100' nor '00000100'. Your implementation of `numToBaseB` should also not have leading 0's in its output.

Hint: A good way to implement `numToBaseB` is to use a helper function that does the same thing, except that for input 0 it returns the empty string. The helper can be recursive, and returning the empty string when the input value of `N` is `0` will help you to make sure other outputs do not have leading zeros. The main function can convert the empty string to '0', in case the input is 0, and for any other input it does not need to do anything.

Part 2

Naturally, we'd like to do the opposite conversion as well! 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. The rightmost character of the string should be the *least* significant digit of the number in base B—this is the familiar representation. `baseBToNum` should return an integer in base 10 representing the same number as `S`. Note that if the input string S is the empty string, the function `baseBToNum` should return `0`.

Here are some sample runs:

```
>>> baseBToNum("11", 2)

3

>>> baseBToNum("11", 3)

4

>>> baseBToNum("11", 10)

11

>>> baseBToNum("", 10)

0                        # the empty string should return 0
```

Part 3

Now, we can assemble what we've written to write a function called called `baseToBase(B1,B2,SinB1)` that takes three inputs: a base `B1`, a base `B2` (both of which are between 2 and 10, inclusive) and `SinB1`, which is a string representing a number in base `B1`. `baseToBase` should return a string representing the same number in base `B2`. Here is some sample input and output:

```
>>> baseToBase(2, 10, "11")    # 11 in base 2 is 3 in base 10...

'3'

>>> baseToBase(10, 2, "3")     # 3 in base 10 is 11 in base 2...
```

```
'11'

>>> baseToBase(3, 5, "11")     # 11 in base 3 is 4 in base 5...

'4'
```

Again, the output of `baseToBase` should have no leading 0's. So, the output '11' in the second example above is OK; however, the output '011' would not be. Note that you don't have to rewrite your `numToBaseB` function—instead, you can simply *use* `numToBaseB` and its twin, `baseBToNum`!

Part 4

Here's a short problem that puts what you've written to use! Write a program called `add(S,T)` that takes two binary strings `S` and `T` as input and returns their sum, also in binary. You can do this by converting the two binary strings to two base-10 numbers, adding the two numbers, and then converting the resulting sum back into base 2! Here is some sample input and output:

```
>>> add("11", "01")

'100'

>>> add("011", "100")

'111'

>>> add("110", "011")

'1001'
```

Part 5

`add` shows one way of adding two binary numbers: first convert them to base 10, add them, and then convert the result back to binary. In this problem, you will implement a different, more direct, method for adding two binary numbers:

```
  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` that takes two strings as input. These strings are the representations of binary numbers. As usual, the rightmost bit of a string is the least significant bit. Your `addB` function should return a new string representing the sum of the two input strings. The sum needs to be computed using the binary addition algorithm, shown above, and not using base conversions.

Your implementation should permit the input strings to have *potentially different* lengths and should output the sum **without** leading 0's. You may wish to define one or more "helper" functions that are used by `addB` (but you won't *have* to). Here is some sample input and output:

```
>>> addB("11", "1")

'100'

>>> addB("011", "100")

'111'
```

Note that `addB` should work **entirely** in binary, and entirely with strings. **Do not** use int() and str() to convert between the two data types.

HINT: Part 5 is challenging because it asks for everything to be done in terms of strings. One way to do this is to use a dictionary to hold the truth table for computing the bits by a full adder:

```
# Each row has (x,y,carry-in) : (sum,carry-out)
FullAdder =
{ ('0','0','0') : ('0','0'),
('0','0','1') : ('1','0'),
('0','1','0') : ('1','0'),
('0','1','1') : ('0','1'),
('1','0','0') : ('1','0'),
('1','0','1') : ('0','1'),
('1','1','0') : ('0','1'),
('1','1','1') : ('1','1') }
```

Notice that when we apply, say `FullAdder[('0','0','1')]`, we get back a tuple, in this case `('1','0')`. To get the two elements out of the tuple you can write this assignment:

```
sumBit, carryBit = FullAdder[('0','0','1')]
```

Now the main function recursively uses the full adder, working from the right (one's place) just like the addition circuit.