CECS 229: Programming Assignment #1

Due Date:

Sunday, 2/11 @ 11:59 PM

Submission Instructions:

Complete the programming problems in the file named `pa1.py`. You may test your implementation on your Repl.it workspace by running `main.py`. When you are satisfied with your implementation, download `pa1.py` and submit it to the appropriate CodePost auto-grader folder.

Objectives:

1. Find all integers in a given range that are congruent to an integer a under some modulo m.
2. Find the b-representation of a given integer.
3. Apply numerical algorithms for computing the sum of two numbers in binary representation.
4. Apply numerical algorithms for computing the product of two numbers in binary representation.

Problem 1:

Complete the function `equiv_to(a, m, low, high)` that returns a list of all the integers x in the range $[\text{low}, \text{high}]$ such that $x \equiv a \pmod{m}$.

EXAMPLES:

Finding all integers $-10 \leq x \leq 15$ such that $x \equiv 3 \pmod{5}$:

`IN: equiv_to(3, 5, -10, 15)`

`OUT: [-7, -2, 3, 8, 13]`

Finding all integers $-29 \leq x \leq -11$ such that $x \equiv 3 \pmod{5}$:

`IN: equiv_to(12, 15, -29, -11)`

`OUT: [-18]`

Finding all integers $3 \leq x \leq 21$ such that $x \equiv -20 \pmod{4}$:

`IN: equiv_to(-20, 4, 3, 21)`

`OUT: [4, 8, 12, 16, 20]`

HINT:

By definition, all integers x that are equivalent to a under modulo m must satisfy that

$$x - a = m \cdot k \quad \text{for some integer } k$$

Hence,

$$x = mk + a$$

Notice that if all the x values must to be in the range $[\text{low}, \text{high}]$, then

$$\text{low} \leq mk + a \leq \text{high}$$

What lower- and upper-bound does this place on k? How do these k-values allow us to find the goal x-values?

In [ ]:
```
def equiv_to(a, m, low, high):
    k_low = # FIXME: update k_low
    k_high = # FIXME: update k_high
    k_vals = list(range(k_low, k_high +1))
    x_vals = # FIXME: update x_vals
    return x_vals
```

Problem 2:

Complete the function `b_rep(n, b)` that computes the base b-representation of an integer `n` given in decimal representation (i.e. typical base 10 representation). Your implementation must use ALGORITHM 1.2.1 of the "Integer Representations & Algorithms" lecture notes.

No credit will be given to functions that employ any other implementation. The function can not use built-in functions that already perform some kind of base b-representation. For example, the function implementation can **not** use the functions `bin()` or `int(a, base=2)`.

The function should satisfy the following:

1. INPUT:
   - `n` - a positive integer representing a number in decimal representation
   - `b` - an integer representing the desired base

1. OUTPUT:
   - a string containing the b-expansion of integer `a`.

EXAMPLES:

`IN: b_rep(10, 2)`

`OUT: 1010`

`IN: b_rep(10, 8)`

`OUT: 12`

`IN: b_rep(10, 16)`

`OUT: A`

In [22]:
```
def b_rep(n, b):
    digits = [] # stores the digits of the b-expansion
    q = n
    while q != 0:
        digit = # FIXME: Update digit
        if b == 16 and digit > 9:
            hex_dict = {10: 'A', 11 : 'B', 12: 'C', 13: 'D', 14: 'E', 15 : 'F'}
            digit = # FIXME: Update digit
        digits.append(digit)
        q = # FIXME: Update q
    return # FIXME: Return the string of digits
```

Problem 3:

Complete the function `binary_add(a, b)` that computes the sum of the binary numbers

$$a = (a_{i-1}, a_{i-2}, \ldots, a_0)_2$$

and

$$b = (b_{j-1}, b_{j-2}, \ldots, b_0)_2$$

using ALGORITHM 1.2.3 of the "Integer Representations & Algorithms" lecture notes.

No credit will be given to functions that employ any other implementation. The function can not use built-in functions that already perform some kind of binary representation or addition of binary numbers. For example, the function implementation can **not** use the functions `bin()` or `int(a, base=2)`.

The function should satisfy the following:

1. INPUT:
   - `a` - a string of the 0's and 1's that make up the first binary number. Assume the string contains no spaces.
   - `b` - a string of the 0's and 1's that make up the second binary number. Assume the string contains no spaces.

1. OUTPUT:
   - the string of 0's and 1's that is the result of computing $a + b$.

EXAMPLE:

`IN: binary_add( '101011' , '11011')`

OUT: '1000110'

In [42]:
```
def binary_add(a, b):
    # removing all whitespace from the strings
    a = a.replace(' ', '')
    b = b.replace(' ', '')

    # padding the strings with 0's so they are the same length
    if len(a) < len(b):
        diff = len(b) - len(a)
        a = "0" *diff + a
    elif len(a) > len(b):
        diff = len(a) - len(b)
        b = "0" *diff + b

    # addition algorithm
    result = ""
    carry = 0
    for i in reversed(range(len(a))):
        a_i = int(a[i])
        b_i = int(b[i])

        result += # FIXME: Update result
        carry = # FIXME: Update carry
    if carry == 1:
        result += # FIXME: Update result
    return # FIXME return the appropriate string
```

Problem 4:

Complete function `binary_mul(a, b)` that computes the product of the binary numbers

$$a = (a_{i-1}, a_{i-2}, \dots, a_0)_2$$

and

$$b = (b_{j-1}, b_{j-2}, \dots, b_0)_2$$

using ALGORITHM 1.2.4 of the "Integer Representations & Algorithms" lecture notes. No credit will be given to functions that employ any other implementation. The function can not use built-in functions that already perform some kind of binary representation or addition of binary numbers. For example, the function implementation can **not** use the functions `bin()` or `int(a, base=2)`.

The function should satisfy the following:

1. INPUT:
   - `a` - a string of the 0's and 1's that make up the first binary number. Assume the string contains no spaces.
   - `b` - a string of the 0's and 1's that make up the second binary number. Assume the string contains no spaces.

1. OUTPUT:
    - the string of 0's and 1's that is the result of computing $a \times b$.

EXAMPLE:

`IN: binary_mul( '101011' , '11011')`

`OUT: '10010001001'`

In [75]:
```
def binary_mul(a, b):
    # removing all whitespace from the strings
    a = a.replace(' ', '')
    b = b.replace(' ', '')

    # multiplication algorithm
    partial_products = []
    i = 0 # tracks the index of the current binary bit of string 'a' beginning at 0, r
    for bit in reversed(a):
        if bit == '1':
            partial_products.append("""FIXME: Append the correct object to partial pro
        i += 1

    result = '0'
    while len(partial_products) > 0:
        result = binary_add("FIXME: Input the correct arguments")
        del partial_products[0]
    return # FIXME: Return the appropriate result
```