CECS 229: Programming Assignment #1

Due Date:

Sunday, 2/5 @ 11:59 PM

EXTENDED UNTIL Tues 2/7 @ 11:59 PM

Submission Instructions:

To receive credit for this assignment, you must submit **to CodePost** this file converted to a Python script named `pa1.py`

Objectives:

1. Compute the quotient and remainder of two numbers.
2. Apply numerical algorithms for computing the sum of two numbers in binary representation.
3. Apply numerical algorithms for computing the modular exponentiation of a positive integer.

Problem 1:

Program a function `div_alg(a, d)` that computes the quotient and remainder of

$$a \div d$$

according to the Division Algorithm (THM 4.1.2).

The function should satisfy the following:

1. INPUT:
   - `a` - an integer representing the dividend
   - `d` - positive integer representing the divisor

1. OUTPUT:
   - a dictionary of the form `{'quotient' : q, 'remainder' : r}` where `q` and `r` are the quotient and remainder values, respectively. The remainder should satisfy, $0 \leq r < d$.

EXAMPLE:

`>> div_alg( 101 , 11 )`

`{'quotient' : 9, 'remainder' : 2}`

In [48]:
```
def div_alg(a, d): # works with arguments a and d inside, but doesn't without i
    """
    this function finds the quotient and remainder of two integers.
    param@ int a: dividend
    param@ int d: divisor
    returns a dictionary of quotient and remainder back.
    """
    q = a // d # find q by using integer division. this gets us the "highest" q
    r = a - d*q # find r by using 'a' minus divisor*quotient
    return {'quotient' : q, 'remainder' : r}
```

In [45]:
```
# Test case: div_alg(-11,3)
# Test case: div_alg(101, 11)
# BOTH WORK
```

Problem 2:

Program a function `binary_add(a, b)` that computes the sum 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 the algorithm discussed in lecture. 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. The string *may* contain spaces.
    - `b` - a string of the 0's and 1's that make up the first binary number. The string *may* contain spaces.

1. OUTPUT:
    - the string of 0's and 1's that is the result of computing $a + b$. The string must be separated by spaces into blocks of 4 characters or less, beginning at the end of the string.

EXAMPLE:

`>> binary_add( '10 1011' , '11011')`

`'100 0110'`

In [288…

```
def binary_add(a, b):
    c = 0 # carry
    s = '' # sum

    # remove spaces
    space = ' '
    if space in a:
        a = a.replace(' ', '')
    if space in b:
        b = b.replace(' ', '')

    # make a and b len same using zfill(). '.zfill()' puts 0s in front of the s
    a = a.zfill(max(len(a), len(b)))
    b = b.zfill(max(len(a), len(b)))

    # addition loop
    for i in range(len(a)-1, -1, -1):
        # resulting binary digit
        s = str( (int(a[i]) + int(b[i]) + c) % 2 ) + s
        # carry is determined by quotient
        c = (int(a[i]) + int(b[i]) + c) // 2

    # add any remaining carry to the result
    if c  != 0:
        s = str(c) + s

    # adds a space for every 4 bits
    s = s[::-1]
    s = ' '.join([s[i:i+4] for i in range(0, len(s), 4)])
    s = s[::-1]

    return s
```

In [287…

```
# Test cases
# binary_add('101011' , '11011') # expecting: '100 0110' --> works!
```

Problem 3:

Program a function `mod_exp(b, n, m)` that computes

$$b^n \mod m$$

using the algorithm discussed in lecture. No credit will be given to functions that employ any other implementation. For example, if the function implementation simply consists of `b ** n % m`, no credit will be given.

The function should satisfy the following:

1. INPUT:
   - `b` - positive integer representing the base
   - `n` - positive integer representing the exponent
   - `m` - positive integer representing the modulo

1. OUTPUT:

- the computation of $b^n \mod m$ if b, n, m are positive integers, 0 otherwise.

EXAMPLE:

`>> mod_exp( 3 , 644, 645 )`

`36`

In [285…

```
def mod_exp(b, n, m):
    """
    this function computes b^n mod m
    param@ base (b), exponent (n), modulo (m)
    returns an int as result
    """
    # initial check of any negative inputs
    if b < 0 or n < 0 or m < 0:
        return 0
    # convert n to binary. since bin(#) always returns "0b....", we use [2:] to
    nBinary = bin(n)[2:]
    # returning later, so this is the result of the big mod
    x = 1
    # used to keep track of intermediate results during calculations
    p = b % m
    # iterate through the binary number (i.e., length of bin(n)[2:]), from righ
    for i in range(len(nBinary)-1, -1, -1):
        # check if 2^i has a coefficient of 1 in the expansion
        if int(nBinary[i]) == 1:
            x = (x * p) % m # only computed if b^(2^i) mod m is part of expansi
        p = (p * p) % m # see above p
    return x
```

In [286…

```
# TEST CASES
# mod_exp(3, 2003, 99) works!
# mod_exp(3, 644, 645) works!
# mod_exp(7, 644 ,645) works!
# mod_exp(3, 11 ,5) works!
# mod_exp(-14, 1191, 806) works!
```