**Implementation Description**

For this project, I decided to implement most parts of the RSA and Elgamal algorithm myself so that I can clearly understand their process. External libraries and modules were used for accurate time measurement and mathematical calculation.

Both RSA and Elgamal are implemented as classes with a *run* function that prompt for input for each run which return the ciphertext and deciphered text, and a *run\_n* function that run the algorithm for all the given inputs (List of string) which return time consumed. For this project, all strings are randomly generated from letters.

Note that input starting a string with ‘0b’ represent it as binary integers, ‘int’ as literal integer, and others as literal string. (The encoding process will not be counted in time measurement)

**Steps:**

1. Run the RSA and Elgamal for 100 times with a string size of 5, 10, 15, 20, 25, and 30 with p = 7, q = 5, a = 2.
2. Run RSA and Elgamal for 100 times with a string size of 5, 10, 15, 20, 25, and 30 with p = 41, q = 43, a = 37.
3. Run the RSA and Elgamal for 100 times with a string size of 5, 10, 15, 20, 25, and 30 with p = 241, q = 251, a = 211.
4. Run the RSA for 100 times with a string size of 20 and 30 with p = 41, q = 43 and record the result of each run.
5. Run the RSA for 100 times with a string size of 5, 10, 15, 20, 25, and 30 with p = 41, q = 43, e = 167, d = 503.

**Results**

**Discussion**

**Data Inconsistency:**

As the data above shown, the benchmark results for RSA algorithm in figures 1, 2, and 3 are inconsistent. This phenomenon might be caused by other processes’ interference, CPU clock cycle, caching, or other unknown reason. I have tried multiple benchmarks while making sure that no other process other than Window’s default processes are running, but the above data is the most consistent result that shown relatively less interference. In fact, this is the first time I encounter major inconsistency for benchmarking. Figure 6 verified that the inconsistent is not caused by the RSA algorithm itself as it appears that the time consumption decreased when all variables are constant.

**Time Comparison:**

1. Value of p, q, and a:

* From figures 1, 2, 3, and 4 we can see that the bigger the value of p, q, and a, the more time it takes for both RSA and Elgamal .
* The value of p and q greatly increase the time consumption of RSA.
* The effect of the value of p and a on Elgamal’s time consumption is minor.

1. Value of e and d:

* As figure 5 shown,
* The bigger the value of p and q, the more chance that the random e chosen does not meet the requirement of gcd(e, z) = 1
* The bigger the value of p and q, the more complex it is to calculate the inverse modular of e to find d.
* Giving a fixed e and d in RSA will significantly reduce the complexity of the process due to the naïve brute force calculation of inverse modular of e which has a complexity of O(n).
* There is more effective method to calculate the inverse modulo with a O(Log m) for which I did not figured out how to implement in this project, but it would still lower than O(1) of Elgamal scheme.

1. **I**nput Size:

* As figures 1, 2, 3, and 4 shown, Elgamal scheme is significantly faster than RSA with a consistent time consumption regardless of the input size.
* RSA require more time as input size increase.
* Elgamal’s time consumption is not affected by the input size.

**Security Comparison:**

1. Variation of output:

* Elgamal generate different output every time it encrypts a message because a different i is chosen.
* RSA generate the same output every time it encrypts a message because no variation in process was made in the algorithm
* Some implementations of RSA overcome this weakness by using a padding the message before encrypting it.

1. Attack on deciphering the message:

* Elgamal is significantly harder to be decipher than RSA if the keys were not made available.
* Elgamal implement the Discrete Logarithm Problem which make it significantly more computationally complex to decipher the message without knowing the key.

**Conclusion**

RSA algorithm is clearly inferior to the Elgamal scheme in this implementation. RSA require a significant overhead to encrypt the message while not being as secure as the Elgamal scheme.

**Appendix**

RSA.py

from NextPrime import is\_prime

from math import gcd

#from modular\_inverse import modinv

import random

from GS\_timing import millis

class RSA:

    """

    Calculate the public key and private key given 2 prime number and a message in binary

    Steps:

    1. Find two large primes p and q.

    2. Find n such that n = p(q).

    3. Find z such that z = (p-1)(q-1).

    4. Find e such that e ∈ {1,2,...,z-1} subject to gcd(e,z) = 1

    5. Find d such that d = e^-1 % z, and e != d.

    6. Find ciphertext K+(m) = m^e % n = c , and plaintext K-(c) = c^d % n = m

    Public key = (n, e)

    Private key = (d)

    """

    def \_\_init\_\_(self, pq=None):

        if not pq:

            self.p = int(input("p: "))          # large prime

            assert is\_prime(self.p), "p value is not a prime"

            self.q = int(input("q: "))          # large prime

            assert is\_prime(self.q), "q value is not a prime"

        else:

            self.p, self.q = pq

        self.m = -1                             # plaintext in bits

        self.n = self.find\_n(self.p, self.q)    # public key

        self.z = self.find\_z(self.p, self.q)

        self.e = self.find\_e(self.z)            # public key

        self.d = self.find\_d(self.e, self.z)    # private key

        self.c = -1                             # ciphertext

        self.d\_m = -1                           # deciphered text

    @staticmethod

    def encode\_m(m):

        """

        @param string m: message

        """

        # if input is binary

        if m[0:2] == '0b':

            return int(m[2:], 2)

        # if input is integer

        elif m[0:3] == 'int':

            return int(m[3:])

        # if input is string

        else:

            return int(''.join(format(ord(i), 'b') for i in m), 2)

    @staticmethod

    def find\_n(p, q):

        """

        Find n such that n = p(q)

        """

        return p \* q

    @staticmethod

    def find\_z(p, q):

        """

        Find z such that z = (p-1)(q-1)

        """

        return (p - 1) \* (q - 1)

    @staticmethod

    def find\_e(z):

        """

        This mode used the explanation given from CSC652

        Find e such that e ∈ {1,2,...,z-1} subject to gcd(e,z) = 1

        """

        for \_ in range(1, z):

            e = int(random.uniform(1., z))

            if gcd(e, z) == 1:

                return e

        raise Exception('Unable to find e')

    @staticmethod

    def find\_d(e, z):

        """

        Find d such that d = e^-1 % z, and e != d

        \* Optimize later for debugging purpose

        """

        # Fermat Little Theorem

        if is\_prime(z):

            d = (e\*\*(z-2)) % z

            if d != e:

                return d

        d = 2

        while ((d\*e) % z) != 1 or d == e:

            d += 1

        return d

        """

        while True:

            d = modinv(e, z)

            if d != e:

                return e

        """

    @staticmethod

    def find\_c(m, e, n):

        """

        Find public key K+(m) = m^e % n = c

        return cyphertext

        """

        return (m \*\* e) % n

    @staticmethod

    def find\_d\_m(c, d, n):

        """

        Find decyphertext K-(c) = c^d % n = m

        """

        return (c\*\*d) % n

    def run(self, auto=False):

        """

        This function run all the process

        return (ciphertext, deciphered text)

        """

        if not auto:

            print(

                "Begin with '0b' for binary or 'int' for integer or '' for literal string")

            self.m = self.encode\_m(input("m: "))

            print("-------------------")

        self.c = self.find\_c(self.m, self.e, self.n)

        self.d\_m = self.find\_d\_m(self.c, self.d, self.n)

        return self.c, self.d\_m

    def run\_n(self, m):

        """

        This function run the rsa for all m

        @param list m: list of plaintext

        return float: time consumed in milliseconds

        """

        t = 0.

        for i in range(len(m)):

            self.m = self.encode\_m(m[i])

            # minimize the effect of unrelated operation for time measurement

            t1 = millis()

            self.run(auto=True)

            t2 = millis()

            t += t2 - t1

        return t

    def run\_1(self, m):

        """

        This function run the rsa for m

        @param string m: plaintext

        return float: time consumed in milliseconds

        """

        t = 0.

        self.m = self.encode\_m(m)

        t1 = millis()

        self.run(auto=True)

        t2 = millis()

        return t2 - t1

    def \_\_str\_\_(self):

        s = "\*\*\* Results: \*\*\*\n"

        s += "p: " + str(self.p) + "\n"

        s += "q: " + str(self.q) + "\n"

        s += "m: " + str(self.m) + "\n"

        s += "n: " + str(self.n) + "\n"

        s += "z: " + str(self.z) + "\n"

        s += "e: " + str(self.e) + "\n"

        s += "d: " + str(self.d) + "\n"

        s += "K+(" + str(self.m) + ") = " + str(self.c) + "\n"

        s += "K-(" + str(self.c) + ") = " + str(self.d\_m)

        return s

    def bin\_to\_str(b):

        """

        \* In Progress

        Decode ascii to string

        @param string b: a collection of ascii (ex: '0b1011010010001010)

        return string

        """

        if b[:2] == '0b':

            b = b[2:]

        out = ''

        while len(b) > 0:

            out += chr(int(b[:8], 2))

            b = b[8:]

        print(out)

if \_\_name\_\_ == "\_\_main\_\_":

    print('\*\*\* USING RSA \*\*\*')

    rsa = RSA()

    opt = input('Multiple runs? (y/n)\t: ')

    if opt == 'y':

        opt = input('Auto generate strings? (y/n)\t:')

        if opt == 'y':

            l = int(input('Length of input string\t: '))

            n = int(input('Number of runs\t: '))

            import string

            x = [''.join(random.choice(string.ascii\_letters)

                         for \_ in range(l)) for \_ in range(n)]

        elif opt == 'n':

            x = input('Please enter list of strings as shown : [I,am,legend]\n').strip(

                '[]').split(',')

        t = rsa.run\_n(x)

        print('Time consumed for ', n, ' runs with ', l, 'string length\t:', t)

    elif opt == 'n':

        ciphertext, deciphertext = rsa.run()

        print(rsa)

Elgamal.py

from NextPrime import is\_prime

import random

from GS\_timing import millis

class Elgamal:

    """

    Steps:

    1.  Find large prime p.

    2.  Find α such that α ∈ Z^\*\_p

    3.  Find d such that d ∈ {2,...,p-2}

    4.  Find β such that β = α^d % p

    5.  Find i such that i ∈ {2,...,p-2}

    6.  Find k\_E such that k\_E = α^i % p

    7.  Find k\_M such that k\_M =  β^i % p

    8.  Find ciphertext such that y = x\*k\_M % p

    9.  Find k\_M such that k\_M =  k\_E^d % p

    10. Find plaintext such that x = y\*k^(-1)\_M % p

    plaintext = x

    ciphertext = y

    initiator sends:  (p, α, β)

    receiver sends: (k\_E, y)

    """

    def \_\_init\_\_(self, pa=None):

        if not pa:

            self.p = int(input("p: "))                # large prime

            assert is\_prime(self.p), "p value is not a prime"

            self.a = int(input("a: "))                # α ∈ Z^\*\_p

            assert self.a >= 1 and self.a <= self.p-1, "a value is not subset of Z^\*\_p"

        else:

            self.p, self.a = pa

        self.x = -1                                   # plaintext in bits

        self.d = self.find\_d(self.p)                  # private key

        self.b = self.find\_b(self.a, self.d, self.p)  # public key (β)

        self.i = -1                                   # choose new i on each run

        self.k\_E = -1                                 # ephemeral key

        self.k\_M = -1                                 # masking key (sender)

        self.y = -1                                   # ciphertext

        # masking key (receiver)

        self.inv\_d\_k\_M = -1

        self.d\_x = -1                                 # deciphered text

    @staticmethod

    def encode\_x(x):

        """

        @param string x: literal string

        return binary of string ascii

        """

        # if input is binary

        if x[0:2] == '0b':

            return int(x[2:], 2)

        # if input is integer

        elif x[0:3] == 'int':

            return int(x[3:])

        # if input is string

        else:

            return int(''.join(format(ord(i), 'b') for i in x), 2)

    @staticmethod

    def find\_d(p):

        """

        Find d such that d ∈ {2,...,p-2}

        """

        return random.randint(2, p-2)

    @staticmethod

    def find\_b(a, d, p):

        """

        Find β such that β = α^d % p

        """

        return (a\*\*d) % p

    @staticmethod

    def find\_i(p):

        """

        Find i such that i ∈ {2,...,p-2}

        """

        return random.randint(2, p-2)

    @staticmethod

    def find\_k\_E(a, i, p):

        """

        Find k\_E such that k\_E = α^i % p

        """

        return (a\*\*i) % p

    @staticmethod

    def find\_k\_M(b, i, p):

        """

        Find k\_M such that k\_M =  β^i % p

        """

        return (b\*\*i) % p

    @staticmethod

    def find\_y(x, k\_M, p):

        """

        Find ciphertext such that y = x\*k\_M % p

        """

        return (x\*k\_M) % p

    @staticmethod

    def find\_inv\_d\_k\_M(k\_E, d, p):

        """

        Find d\_k\_M^-1

        Fermat little theorem:

            k\_E^(p-1) % p = 1

        Then d\_k\_M^-1

            = (k\_E^d)^-1 ∙ 1 mod p

            = (k\_E^d)^-1 ∙ k\_E^(p-1) mod p

            = k\_E^(p-d-1) mod p

        """

        return (k\_E\*\*(p-d-1)) % p

    @staticmethod

    def find\_d\_x(y, inv\_d\_k\_M, p):

        """

        Find plaintext such that x = y\*(k\_M^-1) % p

        """

        return (y\*inv\_d\_k\_M) % p

    def run(self, auto=False):

        """

        This function run the elgamal

        @param bool auto: prompt input

        return (ciphertext, plaintext)

        """

        if not auto:

            print(

                "Begin with '0b' for binary or 'int' for integer or '' for literal string")

            self.x = self.encode\_x(input("x: "))

            print("-------------------")

        self.i = self.find\_i(self.p)

        self.k\_E = self.find\_k\_E(self.a, self.i, self.p)

        self.k\_M = self.find\_k\_M(self.b, self.i, self.p)

        self.y = self.find\_y(self.x, self.k\_M, self.p)

        self.inv\_d\_k\_M = self.find\_inv\_d\_k\_M(self.k\_E, self.d, self.p)

        self.d\_x = self.find\_d\_x(self.y, self.inv\_d\_k\_M, self.p)

        return self.y, self.x

    def run\_n(self, x):

        """

        This function run elgamal for all x, each time using a new i

        @param list x: list of plaintext

        return: time consumed in milliseconds

        """

        t = 0.

        for i in range(len(x)):

            self.x = self.encode\_x(x[i])

            # minimize the effect of unrelated operation for time measurement

            t1 = millis()

            self.run(auto=True)

            t2 = millis()

            t += t2 - t1

        return t

    def \_\_str\_\_(self) -> str:

        s = "\*\*\* Results: \*\*\*\n"

        s += "p\t: " + str(self.p) + "\n"

        s += "a\t:" + str(self.a) + "\n"

        s += "x\t: " + str(self.x) + "\n"

        s += "d\t: " + str(self.d) + "\n"

        s += "b\t: " + str(self.b) + "\n"

        s += "i\t: " + str(self.i) + "\n"

        s += "k\_M\t: " + str(self.k\_M) + "\n"

        s += "k\_E\t: " + str(self.k\_E) + "\n"

        s += "y\t: " + str(self.y) + "\n"

        s += "k\_M^-1\t: " + str(self.inv\_d\_k\_M) + "\n"

        s += "d\_x\t: " + str(self.d\_x)

        return s

if \_\_name\_\_ == "\_\_main\_\_":

    print('\*\*\* USING ELGAMAL ENCRYPTION PROTOCOL \*\*\*')

    elgamal = Elgamal()

    opt = input('Multiple runs? (y/n)\t: ')

    if opt == 'y':

        opt = input('Auto generate strings? (y/n)\t:')

        if opt == 'y':

            l = int(input('Length of input string\t: '))

            n = int(input('Number of runs\t: '))

            import string

            x = [''.join(random.choice(string.ascii\_letters)

                         for \_ in range(l)) for \_ in range(n)]

        elif opt == 'n':

            x = input('Please enter list of strings as shown : [I,am,legend]\n').strip(

                '[]').split(',')

        t = elgamal.run\_n(x)

        print('Time consumed for ', n, ' runs with ', l, 'string length\t:', t)

    elif opt == 'n':

        ciphertext, deciphertext = elgamal.run()

        print(elgamal)

test\_script\_1.py

"""

Run the algorithms for 100 times with a string size of [5, 10, 15, 20, 25, and 30]

with p = [7, 41, 241], q = [5, 43, 251], a = [2, 37, 211] and record the results.

"""

from RSA import RSA

from Elgamal import Elgamal

import random

import string

import xlwt

# All variables

p = [7, 41, 241]  # large prime

q = [5, 43, 251]  # large prime

a = [2, 37, 211]  # ∈ Z^\*\_p

l = [5, 10, 15, 20, 25, 30]   # string length

n = 100  # number of runs

assert len(p) == len(q) == len(a), 'check length of p, q, a'

wb = xlwt.Workbook()

sheet1 = wb.add\_sheet('Sheet 1')

row = 0

for j in range(len(p)):

    sheet1.write(row, 0, 'Variables')

    sheet1.write(row, 1, 'p={}'.format(p[j]))

    sheet1.write(row, 2, 'q={}'.format(q[j]))

    sheet1.write(row, 3, 'a={}'.format(a[j]))

    row += 1

    sheet1.write(row, 0, 'String length')

    sheet1.write(row, 1, 'RSA')

    sheet1.write(row, 2, 'Elgamal')

    row += 1

    for i in range(len(l)):

        x = [''.join(random.choice(string.ascii\_letters)

                     for \_ in range(l[i])) for \_ in range(n)]

        print('Time consumed for ', n, ' runs with ',

              l[i], 'string length with p=', p[j], ' q=', q[j], ' a=', a[j])

        sheet1.write(row, 0, l[i])

        rsa = RSA((p[j], q[j]))

        t = rsa.run\_n(x)

        print('RSA\t: {:.4f} milliseconds'.format(t))

        sheet1.write(row, 1, round(t, 4))

        elgamal = Elgamal((p[j], a[j]))

        t = elgamal.run\_n(x)

        print('Elgamal\t: {:.4f} milliseconds'.format(t))

        sheet1.write(row, 2, round(t, 4))

        row += 1

    row += 1

wb.save('run\_n\_result.xls')

test\_script\_2.py

"""

Run the RSA for 100 times with a string size of [20, 30]

with p = 41, q = 43 and record the result of each run.

"""

from RSA import RSA

import random

import string

import xlwt

# All variables

p = 41  # large prime

q = 43  # large prime

l = [20, 30]   # string length

n = 100  # number of runs

wb = xlwt.Workbook()

sheet1 = wb.add\_sheet('Sheet 1')

row = 0

sheet1.write(row, 0, 'RSA')

row += 1

sheet1.write(row, 0, 'Variables')

sheet1.write(row, 1, 'p={}'.format(p))

sheet1.write(row, 2, 'q={}'.format(q))

row += 1

sheet1.write(row, 0, 'string')

sheet1.write(row, 1, 'x1')

rsa = RSA((p, q))

for i in range(n):

    row += 1

    x1 = ''.join(random.choice(string.ascii\_letters) for \_ in range(l[0]))

    t = rsa.run\_1(x1)

    sheet1.write(row, 0, x1)

    sheet1.write(row, 1, round(t, 4))

row = 2

sheet1.write(row, 3, 'string')

sheet1.write(row, 4, 'x2')

rsa = RSA((p, q))

for i in range(n):

    row += 1

    x2 = ''.join(random.choice(string.ascii\_letters) for \_ in range(l[1]))

    t = rsa.run\_1(x2)

    sheet1.write(row, 3, x2)

    sheet1.write(row, 4, round(t, 4))

wb.save('./Results/run\_1\_result\_1.xls')

test\_script\_3.py

"""

Run the RSA for 100 times with a string size of 30 with

p = 41, q = 43, e = 167, d = 503 and record the result of each run.

"""

import RSA

import RSA\_1

from Elgamal import Elgamal

import random

import string

import xlwt

# All variables

p = 41  # large prime

q = 43  # large prime

a = 37

l = [5, 10, 15, 20, 25, 30]   # string length

n = 100  # number of runs

wb = xlwt.Workbook()

sheet1 = wb.add\_sheet('Sheet 1')

row = 0

sheet1.write(row, 0, 'RSA')

row += 1

sheet1.write(row, 0, 'Variables')

sheet1.write(row, 1, 'p={}'.format(p))

sheet1.write(row, 2, 'q={}'.format(q))

sheet1.write(row, 3, 'a={}'.format(a))

sheet1.write(row, 4, 'Fixed e & d:')

sheet1.write(row, 5, 'e=167')

sheet1.write(row, 6, 'd=503')

row += 1

sheet1.write(row, 0, 'string length')

sheet1.write(row, 1, 'RSA random e & d')

sheet1.write(row, 2, 'RSA fixed e & d')

sheet1.write(row, 3, 'Elgamal')

rsa = RSA.RSA((p, q))

rsa\_1 = RSA\_1.RSA((p, q))

for i in range(len(l)):

    x = [''.join(random.choice(string.ascii\_letters)

                 for \_ in range(l[i])) for \_ in range(n)]

    row += 1

    t = rsa.run\_n(x)

    sheet1.write(row, 0, l[i])

    sheet1.write(row, 1, round(t, 4))

    t = rsa\_1.run\_n(x)

    sheet1.write(row, 2, round(t, 4))

    elgamal = Elgamal((p, a))

    t = elgamal.run\_n(x)

    sheet1.write(row, 3, round(t, 4))

wb.save('./Results/fixed\_RSA\_result\_2.xls')

NextPrime.py

"""

By primo

https://codegolf.stackexchange.com/questions/10701/fastest-code-to-find-the-next-prime

"""

# legendre symbol (a|m)

# note: returns m-1 if a is a non-residue, instead of -1

def legendre(a, m):

    return pow(a, (m-1) >> 1, m)

# strong probable prime

def is\_sprp(n, b=2):

    d = n-1

    s = 0

    while d & 1 == 0:

        s += 1

        d >>= 1

    x = pow(b, d, n)

    if x == 1 or x == n-1:

        return True

    for r in range(1, s):

        x = (x \* x) % n

        if x == 1:

            return False

        elif x == n-1:

            return True

    return False

# lucas probable prime

# assumes D = 1 (mod 4), (D|n) = -1

def is\_lucas\_prp(n, D):

    P = 1

    Q = (1-D) >> 2

    # n+1 = 2\*\*r\*s where s is odd

    s = n+1

    r = 0

    while s & 1 == 0:

        r += 1

        s >>= 1

    # calculate the bit reversal of (odd) s

    # e.g. 19 (10011) <=> 25 (11001)

    t = 0

    while s > 0:

        if s & 1:

            t += 1

            s -= 1

        else:

            t <<= 1

            s >>= 1

    # use the same bit reversal process to calculate the sth Lucas number

    # keep track of q = Q\*\*n as we go

    U = 0

    V = 2

    q = 1

    # mod\_inv(2, n)

    inv\_2 = (n+1) >> 1

    while t > 0:

        if t & 1 == 1:

            # U, V of n+1

            U, V = ((U + V) \* inv\_2) % n, ((D\*U + V) \* inv\_2) % n

            q = (q \* Q) % n

            t -= 1

        else:

            # U, V of n\*2

            U, V = (U \* V) % n, (V \* V - 2 \* q) % n

            q = (q \* q) % n

            t >>= 1

    # double s until we have the 2\*\*r\*sth Lucas number

    while r > 0:

        U, V = (U \* V) % n, (V \* V - 2 \* q) % n

        q = (q \* q) % n

        r -= 1

    # primality check

    # if n is prime, n divides the n+1st Lucas number, given the assumptions

    return U == 0

# primes less than 212

small\_primes = set([

    2,  3,  5,  7, 11, 13, 17, 19, 23, 29,

    31, 37, 41, 43, 47, 53, 59, 61, 67, 71,

    73, 79, 83, 89, 97, 101, 103, 107, 109, 113,

    127, 131, 137, 139, 149, 151, 157, 163, 167, 173,

    179, 181, 191, 193, 197, 199, 211])

# pre-calced sieve of eratosthenes for n = 2, 3, 5, 7

indices = [

    1, 11, 13, 17, 19, 23, 29, 31, 37, 41,

    43, 47, 53, 59, 61, 67, 71, 73, 79, 83,

    89, 97, 101, 103, 107, 109, 113, 121, 127, 131,

    137, 139, 143, 149, 151, 157, 163, 167, 169, 173,

    179, 181, 187, 191, 193, 197, 199, 209]

# distances between sieve values

offsets = [

    10, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4, 6,

    6, 2, 6, 4, 2, 6, 4, 6, 8, 4, 2, 4,

    2, 4, 8, 6, 4, 6, 2, 4, 6, 2, 6, 6,

    4, 2, 4, 6, 2, 6, 4, 2, 4, 2, 10, 2]

max\_int = 2147483647

# an 'almost certain' primality check

def is\_prime(n):

    if n < 212:

        return n in small\_primes

    for p in small\_primes:

        if n % p == 0:

            return False

    # if n is a 32-bit integer, perform full trial division

    if n <= max\_int:

        i = 211

        while i\*i < n:

            for o in offsets:

                i += o

                if n % i == 0:

                    return False

        return True

    # Baillie-PSW

    # this is technically a probabalistic test, but there are no known pseudoprimes

    if not is\_sprp(n):

        return False

    a = 5

    s = 2

    while legendre(a, n) != n-1:

        s = -s

        a = s-a

    return is\_lucas\_prp(n, a)

# next prime strictly larger than n

def next\_prime(n):

    if n < 2:

        return 2

    # first odd larger than n

    n = (n + 1) | 1

    if n < 212:

        while True:

            if n in small\_primes:

                return n

            n += 2

    # find our position in the sieve rotation via binary search

    x = int(n % 210)

    s = 0

    e = 47

    m = 24

    while m != e:

        if indices[m] < x:

            s = m

            m = (s + e + 1) >> 1

        else:

            e = m

            m = (s + e) >> 1

    i = int(n + (indices[m] - x))

    # adjust offsets

    offs = offsets[m:]+offsets[:m]

    while True:

        for o in offs:

            if is\_prime(i):

                return i

            i += o

GS\_timing.py

"""

GS\_timing.py: https://stackoverflow.com/questions/38319606/how-can-i-get-millisecond-and-microsecond-resolution-timestamps-in-python

-create some low-level Arduino-like millis() (milliseconds) and micros()

 (microseconds) timing functions for Python

By Gabriel Staples

http://www.ElectricRCAircraftGuy.com

-click "Contact me" at the top of my website to find my email address

Started: 11 July 2016

Updated: 13 Aug 2016

History (newest on top):

20160813 - v0.2.0 created - added Linux compatibility, using ctypes, so that it's compatible with pre-Python 3.3 (for Python 3.3 or later just use the built-in time functions for Linux, shown here: https://docs.python.org/3/library/time.html)

-ex: time.clock\_gettime(time.CLOCK\_MONOTONIC\_RAW)

20160711 - v0.1.0 created - functions work for Windows \*only\* (via the QPC timer)

References:

WINDOWS:

-personal (C++ code): GS\_PCArduino.h

1) Acquiring high-resolution time stamps (Windows)

   -https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408(v=vs.85).aspx

2) QueryPerformanceCounter function (Windows)

   -https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx

3) QueryPerformanceFrequency function (Windows)

   -https://msdn.microsoft.com/en-us/library/windows/desktop/ms644905(v=vs.85).aspx

4) LARGE\_INTEGER union (Windows)

   -https://msdn.microsoft.com/en-us/library/windows/desktop/aa383713(v=vs.85).aspx

-\*\*\*\*\*https://stackoverflow.com/questions/4430227/python-on-win32-how-to-get-

absolute-timing-cpu-cycle-count

LINUX:

-https://stackoverflow.com/questions/1205722/how-do-i-get-monotonic-time-durations-in-python

"""

import ctypes, os

#Constants:

VERSION = '0.2.0'

#-------------------------------------------------------------------

#FUNCTIONS:

#-------------------------------------------------------------------

#OS-specific low-level timing functions:

if (os.name=='nt'): #for Windows:

    def micros():

        "return a timestamp in microseconds (us)"

        tics = ctypes.c\_int64()

        freq = ctypes.c\_int64()

        #get ticks on the internal ~2MHz QPC clock

        ctypes.windll.Kernel32.QueryPerformanceCounter(ctypes.byref(tics))

        #get the actual freq. of the internal ~2MHz QPC clock

        ctypes.windll.Kernel32.QueryPerformanceFrequency(ctypes.byref(freq))

        t\_us = tics.value\*1e6/freq.value

        return t\_us

    def millis():

        "return a timestamp in milliseconds (ms)"

        tics = ctypes.c\_int64()

        freq = ctypes.c\_int64()

        #get ticks on the internal ~2MHz QPC clock

        ctypes.windll.Kernel32.QueryPerformanceCounter(ctypes.byref(tics))

        #get the actual freq. of the internal ~2MHz QPC clock

        ctypes.windll.Kernel32.QueryPerformanceFrequency(ctypes.byref(freq))

        t\_ms = tics.value\*1e3/freq.value

        return t\_ms

elif (os.name=='posix'): #for Linux:

    #Constants:

    CLOCK\_MONOTONIC\_RAW = 4 # see <linux/time.h> here: https://github.com/torvalds/linux/blob/master/include/uapi/linux/time.h

    #prepare ctype timespec structure of {long, long}

    class timespec(ctypes.Structure):

        \_fields\_ =\

        [

            ('tv\_sec', ctypes.c\_long),

            ('tv\_nsec', ctypes.c\_long)

        ]

    #Configure Python access to the clock\_gettime C library, via ctypes:

    #Documentation:

    #-ctypes.CDLL: https://docs.python.org/3.2/library/ctypes.html

    #-librt.so.1 with clock\_gettime: https://docs.oracle.com/cd/E36784\_01/html/E36873/librt-3lib.html #-

    #-Linux clock\_gettime(): http://linux.die.net/man/3/clock\_gettime

    librt = ctypes.CDLL('librt.so.1', use\_errno=True)

    clock\_gettime = librt.clock\_gettime

    #specify input arguments and types to the C clock\_gettime() function

    # (int clock\_ID, timespec\* t)

    clock\_gettime.argtypes = [ctypes.c\_int, ctypes.POINTER(timespec)]

    def monotonic\_time():

        "return a timestamp in seconds (sec)"

        t = timespec()

        #(Note that clock\_gettime() returns 0 for success, or -1 for failure, in

        # which case errno is set appropriately)

        #-see here: http://linux.die.net/man/3/clock\_gettime

        if clock\_gettime(CLOCK\_MONOTONIC\_RAW , ctypes.pointer(t)) != 0:

            #if clock\_gettime() returns an error

            errno\_ = ctypes.get\_errno()

            raise OSError(errno\_, os.strerror(errno\_))

        return t.tv\_sec + t.tv\_nsec\*1e-9 #sec

    def micros():

        "return a timestamp in microseconds (us)"

        return monotonic\_time()\*1e6 #us

    def millis():

        "return a timestamp in milliseconds (ms)"

        return monotonic\_time()\*1e3 #ms

#Other timing functions:

def delay(delay\_ms):

    "delay for delay\_ms milliseconds (ms)"

    t\_start = millis()

    while (millis() - t\_start < delay\_ms):

      pass #do nothing

    return

def delayMicroseconds(delay\_us):

    "delay for delay\_us microseconds (us)"

    t\_start = micros()

    while (micros() - t\_start < delay\_us):

      pass #do nothing

    return

#-------------------------------------------------------------------

#EXAMPLES:

#-------------------------------------------------------------------

#Only executute this block of code if running this module directly,

#\*not\* if importing it

#-see here: http://effbot.org/pyfaq/tutor-what-is-if-name-main-for.htm

if \_\_name\_\_ == "\_\_main\_\_": #if running this module as a stand-alone program

    #print loop execution time 100 times, using micros()

    tStart = micros() #us

    for x in range(0, 100):

        tNow = micros() #us

        dt = tNow - tStart #us; delta time

        tStart = tNow #us; update

        print("dt(us) = " + str(dt))

    #print loop execution time 100 times, using millis()

    print("\n")

    tStart = millis() #ms

    for x in range(0, 100):

        tNow = millis() #ms

        dt = tNow - tStart #ms; delta time

        tStart = tNow #ms; update

        print("dt(ms) = " + str(dt))

    #print a counter once per second, for 5 seconds, using delay

    print("\nstart")

    for i in range(1,6):

        delay(1000)

        print(i)

    #print a counter once per second, for 5 seconds, using delayMicroseconds

    print("\nstart")

    for i in range(1,6):

        delayMicroseconds(1000000)

        print(i)