CNS Lab Assignment 6: RSA

111703013 Akshay Deodhar

19th October 2020

Code

1 Check whether given number is prime or not?

```
# Check whether given integer is prime, using the Fermat's test as a heuristic,
# then doing simple division checks
def primalty(n):
    # eliminate possibility of number being even
    if n == 2:
        return True
    elif n % 2 == 0:
        return False

    # Fermat's test
    a = n - 1 # always co prime to n
    test_result = pow(a, n - 1, n) # (a ^ (n - 1)) % n

    if test_result != 1:
        # not prime
        return False

    divisor = 3
    while divisor * divisor <= n:
        if n % divisor == 0:
            return False
        divisor += 1

    return True
```

2 find GCD(Greatest Common Divisor) or given two numbers

```
# Finds the gcd of two given positive integers
# Uses euclid's method
# does not work if both the inputs are 0
def gcd(a, b):
    smaller = min(a, b)
    larger = max(a, b)

    while smaller != 0:
        smaller, larger = larger % smaller, smaller

    return larger
```

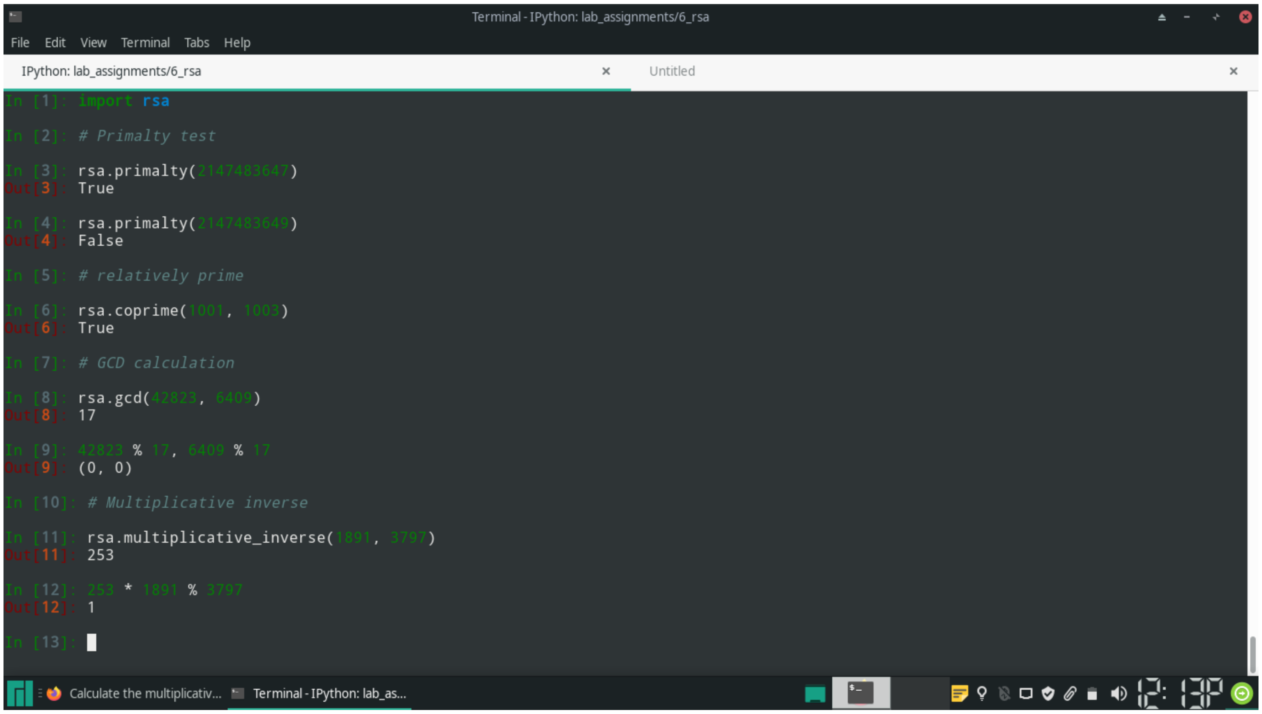
3 check whether given numbers are relatively prime or not?

```
# checks whether a and b are co-prime
def coprime(a, b):
    return gcd(a, b) == 1
```

4 find multiplicative inverse of given two numbers

```
# finds the coefficients of a, b in expression where they express GCD as their
# linear combination
# requires that a > b
def extended_euclidian(a, b):
    old_r, r = a, b
    old_s, s = 1, 0
    old_t, t = 0, 1

    while r != 0:
        # old_r, r form the current pair

        q = old_r // r
        old_r, r = r, old_r - q * r

        # calculate new Bezout coefficients using the old coefficients
        old_s, s = s, old_s - q * s
        old_t, t = t, old_t - q * t


    return old_r, old_s, old_t

# finds (a ^ (-1)) % n
def multiplicative_inverse(a, n):
    remainder, coeff_n, coeff_a = extended_euclidian(n, a)

    assert remainder == 1

    # 1 = coeff_n * n + coeff_a * a

    return coeff_a if coeff_a > 0 else (coeff_a + n)
```

Output of independent execution of helper functions


Terminal - IPython: lab_assignments/6_rsa
File
Edit
View
Terminal
Tabs
Help
IPython: lab_assignments/6_rsa
Untitled
Calculate the multiplicativ...
Terminal - IPython: lab_as...

```
In [1]: import rsa

In [2]: # Primalty test

In [3]: rsa.primalty(2147483647)
Out[3]: True

In [4]: rsa.primalty(2147483649)
Out[4]: False

In [5]: # relatively prime

In [6]: rsa.coprime(1001, 1003)
Out[6]: True

In [7]: # GCD calculation

In [8]: rsa.gcd(42823, 6409)
Out[8]: 17

In [9]: 42823 % 17, 6409 % 17
Out[9]: (0, 0)

In [10]: # Multiplicative inverse

In [11]: rsa.multiplicative_inverse(1891, 3797)
Out[11]: 253

In [12]: 253 * 1891 % 3797
Out[12]: 1

In [13]: 
```

Figure 1: Each helper function, executed seperately

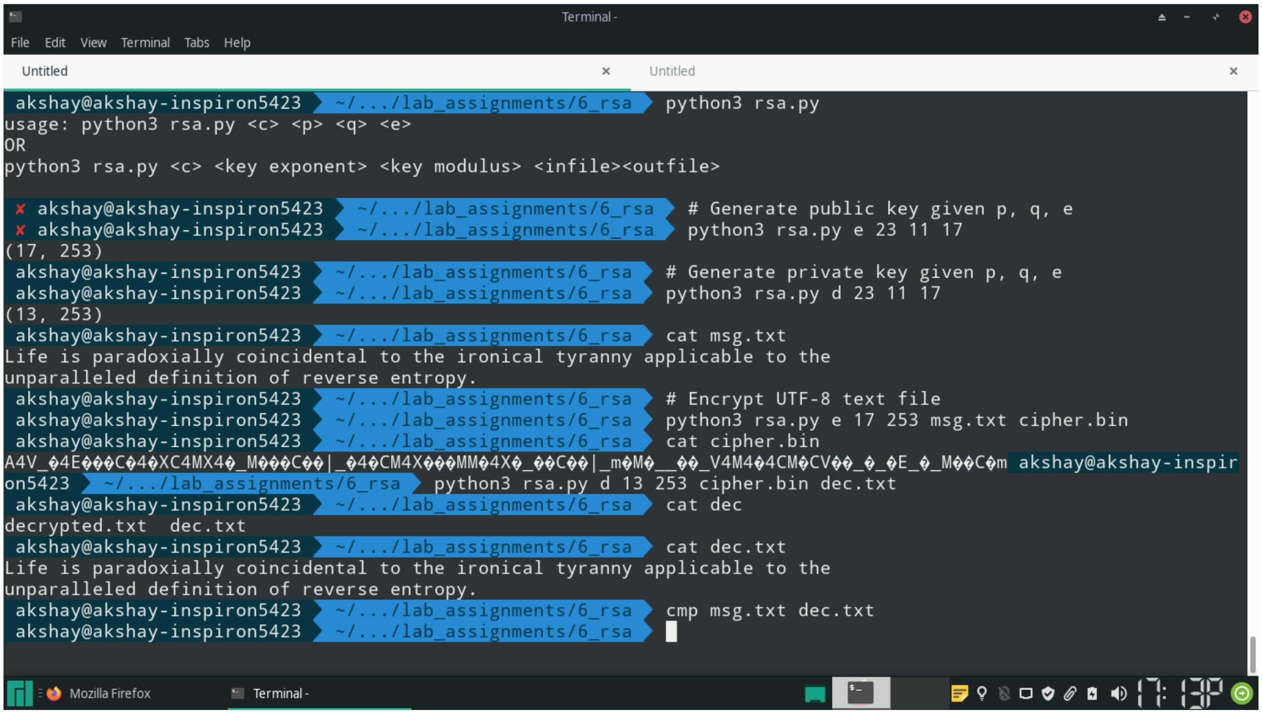
5 implement RSA algorithm

```
# encrypts or decrypts single byte (operation is common)
def rsa_process(msg, key):
    exp, n = key

    # (msg ^ exp) % n
    return pow(msg, exp, n)
```

Complete Code

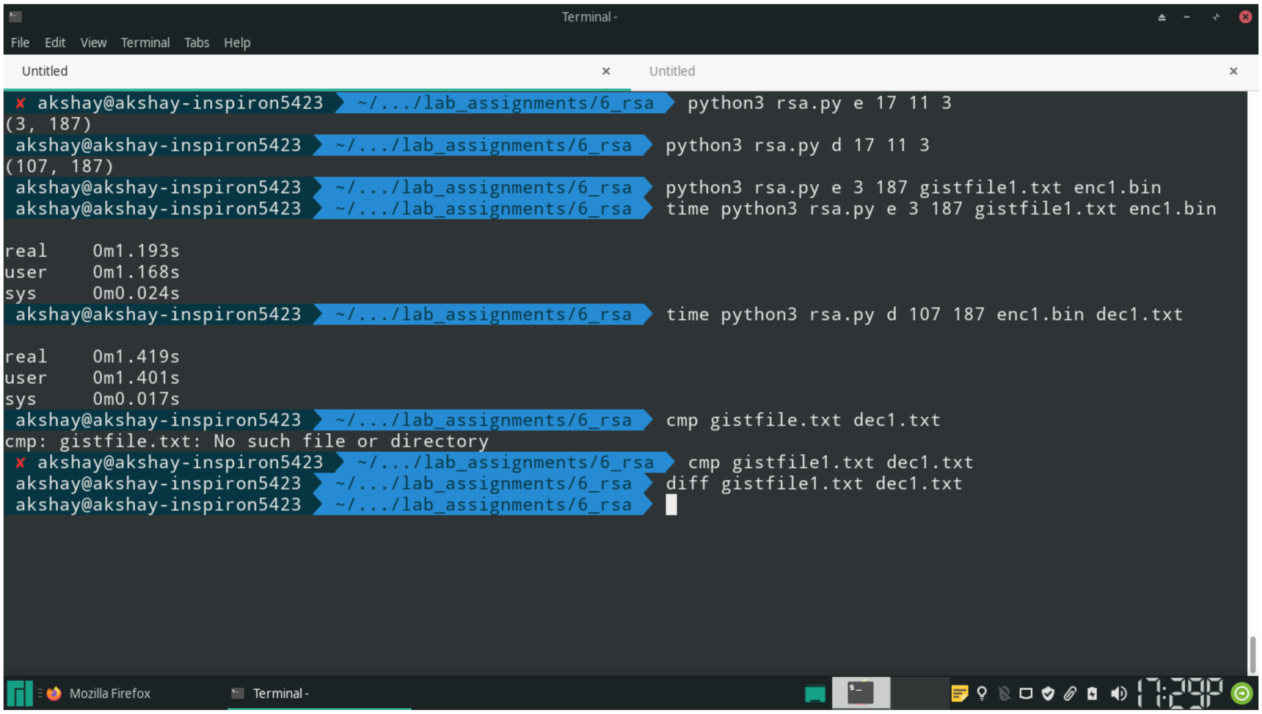
```
# Write a program to
# 1)check whether given number is prime or not?
# 2)find GCD (Greatest Common Divisor) of given two numbers.
# 3)check whether given numbers are relatively prime or not?
# 4)find multiplicative inverse of given two numbers
# 5)implement RSA algorithm
# Compute the time required for encryption and decryption.


# Check whether given integer is prime, using the Fermat's test as a heuristic,
# then doing simple division checks
def primalty(n):
    # eliminate possibility of number being even
    if n == 2:
        return True
    elif n % 2 == 0:
        return False

    # Fermat's test
    a = n - 1 # always co prime to n
    test_result = pow(a, n - 1, n) # (a ^ (n - 1)) % n

    if test_result != 1:
        # not prime
        return False

    divisor = 3
    while divisor * divisor <= n:
        if n % divisor == 0:
            return False
        divisor += 1

    return True

# Finds the gcd of two given positive integers
# Uses euclid's method
# does not work if both the inputs are 0
def gcd(a, b):
    smaller = min(a, b)
    larger = max(a, b)

    while smaller != 0:
        smaller, larger = larger % smaller, smaller

    return larger

# checks whether a and b are co-prime
def coprime(a, b):
    return gcd(a, b) == 1


# finds the coefficients of a, b in expression where they express GCD as their
# linear combination
# requires that a > b
def extended_euclidian(a, b):
    old_r, r = a, b
    old_s, s = 1, 0
    old_t, t = 0, 1

    while r != 0:
        # old_r, r form the current pair

        q = old_r // r
        old_r, r = r, old_r - q * r

        # calculate new Bezout coefficients using the old coefficients
        old_s, s = s, old_s - q * s
        old_t, t = t, old_t - q * t


    return old_r, old_s, old_t

# finds (a ^ (-1)) % n
def multiplicative_inverse(a, n):
    remainder, coeff_n, coeff_a = extended_euclidian(n, a)

    assert remainder == 1
    # 1 = coeff_n * n + coeff_a * a

    return coeff_a if coeff_a > 0 else (coeff_a + n)


# encrypts or decrypts single byte (operation is common)
def rsa_process(msg, key):
    exp, n = key

    # (msg ^ exp) % n
    return pow(msg, exp, n)

# returns nothing, encrypts bytes array consisting of ASCII characters
def process_wrapped(char_array, key):
    n = len(char_array)
    for i in range(n):
        char_array[i] = rsa_process(int(char_array[i]), key)


if __name__ == "__main__":

    from sys import argv, stderr
    n = len(argv)

    if n == 5:
        # rsa.py <c> <p> <q> <e>
        __, c, p, q, e = argv

        p = int(p)
        q = int(q)
        e = int(e)

        n = p * q
        if n < 128 or n > 256:
            print("128 <= n < 256 required", file = stderr)
            exit(1)

        m = (p - 1) * (q - 1)

        if e > m:
            print("e < (p - 1) * (q - 1) required", file = stderr)
            exit(2)

        if c == "e":
            # present the encryption key
            print("({}, {})".format(e, n))
        elif c == "d":
            # present the decryption key
            d = multiplicative_inverse(e, m)
            print("({}, {})".format(d, n))

        exit()

    if n != 6:
        print("usage: python3 rsa.py <c> <p> <q> <e>\n"\
              "OR\n"\
              "python3 rsa.py <c> <key exponent> <key modulus> <infile>"\
              "<outfile>\n", file = stderr)
        exit(3)


    __, c, k, n, infile, outfile = argv

    k = int(k)
    n = int(n)

    # NOTE:
    # The plaintext should always be UTF-8- the MSB is reserved
    # this is because the ciphertext is a number in range (0, n).
    # If 0-255 were allowed in plaintext, then the ciphertext might
    # have a value greater than 255
    # If n > 255, then the size of the ciphertext will be more than 1 byte!
    #
    # tl;dr
    # plaintext should be UTF-8 1 byte
    # p * q should be in range (128, 256]

    if c == 'e'or c == 'd':
        with open(infile, 'rb') as fi:
            with open(outfile, 'wb') as fo:
                # convert the file to a mutable array
                data_array = bytearray(fi.read())

                # encrypt / decrypt each byte in mutable array independently using the key
                # given
                process_wrapped(data_array, (k, n))
                fo.write(data_array)
    else:
        print("<c> has to be 'e' or 'd'", file = stderr)
```

Output



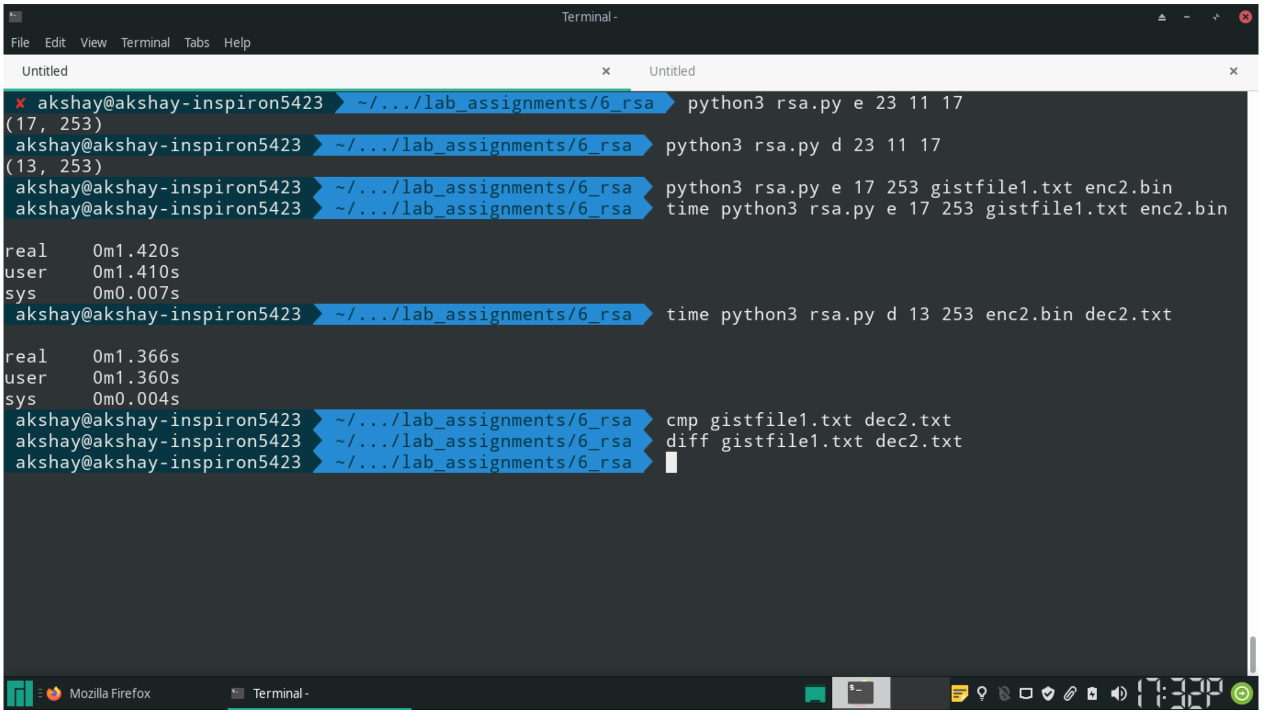
```
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py
usage: python3 rsa.py <c> <p> <q> <e>
OR
python3 rsa.py <c> <key exponent> <key modulus> <infile><outfile>

 ✘ akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  # Generate public key given p, q, e
 ✘ akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py e 23 11 17
(17, 253)
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  # Generate private key given p, q, e
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py d 23 11 17
(13, 253)
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cat msg.txt
Life is paradoxially coincidental to the ironical tyranny applicable to the
unparalleled definition of reverse entropy.
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  # Encrypt UTF-8 text file
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py e 17 253 msg.txt cipher.bin
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cat cipher.bin
A4V_�4E���C�4�XC4MX4�_M���C��|_�4�CM4X���MM�4X�_��C��|_m�M�__��_V4M4�4CM�CV��_�_�E_�_M��C�m akshay@akshay-inspir
on5423  ~/.../lab_assignments/6_rsa  python3 rsa.py d 13 253 cipher.bin dec.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cat dec
decrypted.txt  dec.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cat dec.txt
Life is paradoxially coincidental to the ironical tyranny applicable to the
unparalleled definition of reverse entropy.
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cmp msg.txt dec.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  █
```

Figure 2: Demonstration of encoding and decoding for a small file



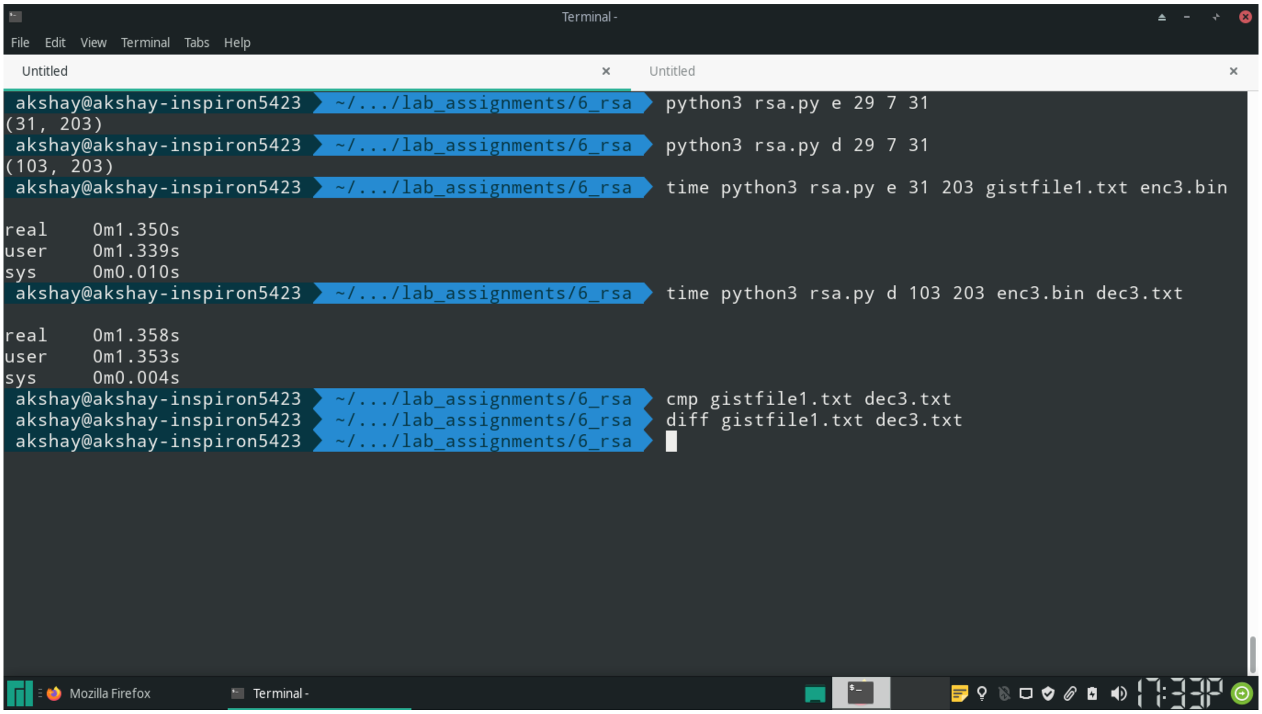
```
 ✘ akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py e 17 11 3
(3, 187)
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py d 17 11 3
(107, 187)
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py e 3 187 gistfile1.txt enc1.bin
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  time python3 rsa.py e 3 187 gistfile1.txt enc1.bin

real    0m1.193s
user    0m1.168s
sys     0m0.024s
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  time python3 rsa.py d 107 187 enc1.bin dec1.txt

real    0m1.419s
user    0m1.401s
sys     0m0.017s
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cmp gistfile.txt dec1.txt
cmp: gistfile.txt: No such file or directory
 ✘ akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cmp gistfile1.txt dec1.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  diff gistfile1.txt dec1.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  █
```

Figure 3: Iteration 1, encryption and decryption of 1 MB file



```
✘ akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py e 23 11 17
(17, 253)
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py d 23 11 17
(13, 253)
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py e 17 253 gistfile1.txt enc2.bin
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  time python3 rsa.py e 17 253 gistfile1.txt enc2.bin

real    0m1.420s
user    0m1.410s
sys     0m0.007s
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  time python3 rsa.py d 13 253 enc2.bin dec2.txt

real    0m1.366s
user    0m1.360s
sys     0m0.004s
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cmp gistfile1.txt dec2.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  diff gistfile1.txt dec2.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  
```

Figure 4: Iteration 2, encryption and decryption of 1 MB file



```
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py e 29 7 31
(31, 203)
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  python3 rsa.py d 29 7 31
(103, 203)
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  time python3 rsa.py e 31 203 gistfile1.txt enc3.bin

real    0m1.350s
user    0m1.339s
sys     0m0.010s
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  time python3 rsa.py d 103 203 enc3.bin dec3.txt

real    0m1.358s
user    0m1.353s
sys     0m0.004s
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  cmp gistfile1.txt dec3.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  diff gistfile1.txt dec3.txt
 akshay@akshay-inspiron5423  ~/.../lab_assignments/6_rsa  
```

Figure 5: Iteration 3, encryption and decryption of 1 MB file

Statistics

The file used for encryption is an ASCII text file having size **1 MB**

The average time needed for encryption (3 repetionts, different keys each time) is **1.30s**

The average time needed for decryption (3 repetitions, different keys each time) is **1.37s**

Notes

- The encryption input is restricted to characters whose binary encoded value is < 128
- p and q must be such that 128 < p * q <= 256
- This is done to avoid the problem of ciphertext size being different from plaintext size.
- The last bit in the byte can be considered as *padding*
- The time required for encryption or decryption depends on the exponent- this is the reason why the decryption time is more- the keys were such that the exponent in the public key was lesser than the one in the private key, or the two were almost equal