<u>Author: David Stilz</u>

Initial Notes:

- Disclaimer: I did implement Extended Euclid's algorithm using a relatively copied web resource (noted in the python file)

Key Setup:

- To generate a public and private key, I first needed to generate p and q prime numbers
- p and q had to be > 100 digits and have a difference of 95 digits between each other
- To accomplish this, I used python's random library to generate random numbers and used Fermat's algorithm to check if those numbers were prime
- Then I would need to choose a number e such that (p-1)(q-1) was relatively prime to e
- I checked this relatively prime status by using Extended Euclid's algorithm to retrieve the gcd of e and (p-1)(q-1)
    - If the result was 1, this meant that they were relatively prime
- At this point, I had a public key (n,e)
- Now, I needed to generate a private key d
    - This was accomplished using Extended Euclid's algorithm on e and (p-1)(q-1) to get the multiplicative inverse
        - I also applied a modulus operator to the result of Extended Euclid by (p-1)(q-1) to ensure a positive number as the inverse
- At this point, I had a private key (n,d)
- Keys were then outputted to private_key.txt and public_key.txt as well as outputted to the console

Key Generation Example Console Output (keys will also be outputted to private_key.txt and public_key.txt):

- public key:

n=
347797596559411047130324226771813905885560066457629075046493418065356282020277005960827549427410804484214297800909022813705019653435510613053422178497312448107164073702276979982723669436048912928014718670421407759293613633520350008228069756409415454384781099297018979686993904995970480976014971

e= 65537

- private key:

n=
347797596559411047130324226771813905885560066457629075046493418065356282020277005960827549427410804484214297800909022813705019653435510613053422178497312448107164073702276979982723669436048912928014718670421407759293613633520350008228069756409415454384781099297018979686993904995970480976014971

d=
24629272710563905423376638180691644982450894432608397352499747520352144664176046884419498251256748745363175119759155094000798559408601038968764592825794235137040856344481357268794162258349620735739376589648863145779843040518405735769130542365720266201587637731114409273124119006937624213464329Algorithms:

- I manually tested my modular inverse function, extended Euclid function, and Fermat's algorithm with custom global print functions to ensure they worked as I moved through the implementation process

Encryption:

- I used the modular exponentiation function and generated public key information (n,e) to get c = m^e mod n
- Message for program is in message.txt and ciphertext outputted goes to ciphertext.txt

Encryption Example Console Output (also outputted to ciphertext.txt):

- original message:

11111111111111100000000002222222222222223333333333333334444444444444445555555555555556666666666666667777777777777778888888888888889999999999999990000000000000000

- generated ciphertext:

268657241946855837777173940438091896864316696959336480297747964405044358928161105419795018850534268190176663777529764151531359115719918122334296721056433580731803785109697025318809665446049446399015576797208050619164099829540182309848156291877144981236412912677171492818293626429052698645204263

Decryption

- I used the modular exponentiation function and generated private key information (n,d) to get m = c^d mod n
- Ciphertext to decrypt is in ciphertext.txt and decrypted message goes to decrypted_message.txt

Decryption Example Console Output (also outputted to decrypted_message.txt)

- ciphertext:

268657241946855837777173940438091896864316696959336480297747964405044358928161105419795018850534268190176663777529764151531359115719918122334296721056433580731803785109697025318809665446049446399015576797208050619164099829540182309848156291877144981236412912677171492818293626429052698645204263

- decrypted:

11111111111111100000000002222222222222223333333333333334444444444444445555555555555556666666666666667777777777777778888888888888889999999999999990000000000000000

Text Run (console outputs referenced above)

- Input (message.txt):
  - 111111111111111000000000022222222222222222222333333333333333444444444444444
    555555555555555666666666666666777777777777777777778888888888888889999999999
    99999000000000000000
- Public key (public_key.txt)
  - Format: n, e
  - 3477975965594110471303242267718139058855600664576290750464934180653562
    8202027700596082754942741080448421429780090902281370501965343551061305
    3422178497312448107164073702276979982723669436048912928014718670421407
    7592936136335203500082280697564094154543847810992970189796869939049959
    70480976014971,65537
- Private key (private_key.txt)
  - Format: n, d
  - 3477975965594110471303242267718139058855600664576290750464934180653562
    8202027700596082754942741080448421429780090902281370501965343551061305
    3422178497312448107164073702276979982723669436048912928014718670421407
    7592936136335203500082280697564094154543847810992970189796869939049959
    70480976014971,24629272710563905423376638180691644982450894432608397352
    4997475203521446641760468844194982512567487453631751197591550940007985
    5940860103896876459282579423513704085634448135726879416225834962073573
    9376589648863145779843040518405735769130542365720266201587637731114409
    273124119006937624213464329
- Encryption (ciphertext.txt)
  - 2686572419468558377771739404380918968643166969593364802977479644050443
    5892816110541979501885053426819017666377752976415153135911571991812233
    4296721056433580731803785109697025318809665446049446399015576797208050
    6191640998295401823098481562918771449812364129126771714928182936264290
    52698645204263
- Decryption (decrypted_message.txt)
  - 111111111111111000000000022222222222222222222333333333333333444444444444444
    555555555555555666666666666666777777777777777777778888888888888889999999999
    99999000000000000000

Running the RSA Program Modules

- Setting up the keys: run .\setup.py
- Encrypt the default message: run .\encrypt.py
- Decrypt the ciphertext: run .\decrypt.py