When values p, q, e, and d are injected:

```
$ C:/Python311/python.exe c:/Users/sidth/OneDrive/Documents/UVADocuments/Semester06/CS3710/ProgrammingAssignments/PA1/rsa.py
3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.1934 64 bit (AMD64)]
p = 3011
q = 1009
e = 13
d = 2100517
Note that if a complete numerical string is entered "012345" the resulting encryption will treat the string as a number.  In the case of a string-based input, "Hi my name is Sid", the individual characters will first be converted to their equivalent ASCII values  and then encrypted to numbers in a list. For decryption these list numbers will be treated as individual characters and decrypted.
Enter message: 67
Encrypted message = 108634
Decrypted message = 67
```

Source code:

```
# program must take a message M as input
# generate random p/q values < 10^6
# output p,q,e,d values of RSA
# M --encrypt--> C --decrypt--> M
import random
import math
import sys

lowLevelPrimes = [
    3, 5, 7, 11, 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, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
    271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367,
    373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,
    463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571,
    577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661,
    673, 67, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773,
    787,797,809,811,821,823,827,829,839,853,857,859,863,877,881,883,
    887,907,911,919,929,937,941,947,953,967,971,977,983,991,997

]
```

```
def main():
    print(sys.version)
    (p, q, e, d, n) = RSA()

    print("p = {}".format(p))
    print("q = {}".format(q))
    print("e = {}".format(e))
    print("d = {}".format(d))

    print("Note that if a complete numerical string is entered \"012345\" the
resulting encryption will treat the string as a number. ",
          "In the case of a string-based input, \"Hi my name is Sid\", the
individual characters will first be converted to their equivalent ASCII values ",
          "and then encrypted to numbers in a list. For decryption these list
numbers will be treated as individual characters and decrypted. ")
    userInput = input("Enter message: ")


    if ( userInput.isnumeric ):
        print("INPUT DETECTED: NUMBER")
        cipherNum = encryptNum((e, n), int(userInput))
        print("Encrypted message = {}".format(cipherNum))

        plainNum = decryptNum((d, n), cipherNum)
        print("Decrypted message = {}".format(plainNum))

    else:
        print("INPUT DETECTED: STRING")
        cipher = encryptText((e, n), userInput)
        cipherText = ''.join(str(x) for x in cipher)
        print("Encrypted message = {}".format(cipherText))

        plainText = decryptText((d, n), cipher)
        print("Decrypted message = {}".format(plainText))

# This is our primary driver function
# we can see tested comments of p, q, e, and d values
# But normally we generate p, q, and e randomly
# while d is determined via calculation on p, q, and e.
# we include assertions to show that the e and d calculations hold
def RSA():
    # large prime 1:
    p = generate_prime()
    # p = 3011
    # large prime 2:
```

```
    q = generate_prime()
    # q = 1009

    n = p * q

    # Carmichael's totient function lambda(n)
    lambda_n = math.lcm(p - 1, q - 1)

    # generating e
    # so e has to be between 2 and lambda_n
    # gcd(e, lambda_n) = 1 so the two are coprime
    # for e to be good we want short bit length
    # and short hamming weight
    e = random.randint(2, lambda_n - 1)
    while math.gcd(e, lambda_n) != 1:
        e = random.randint(2, lambda_n -1)
    # e = 13
    assert math.gcd(lambda_n, e) == 1, "Failed assertion for math.gcd(lambda_n,
e), math.gcd(lambda_n, e) != 1"

    # generating d
    d = multiplicative_inverse(e, lambda_n)
    # d = 2100517

    assert ((d * e) % (lambda_n)) == 1, "Failed assertion for d value, (de) mod
lamda_n != 1"

    return (p, q, e, d, n)

def generate_prime():
    # p and q should be random, similar in magnitude, but differ in length
    # and they should also be prime

    candidate = random.randint(1000, 1000000)
    candidate = candidate - (1 - candidate % 2)

    while( True ):
        if ( check_low_level_prime(candidate) ):
            if ( rabin_miller(candidate) ):
                break
            else:
                candidate += 2
        else:
            candidate += 2
```

```
    # at this point candidate is most likely to be prime
    return candidate

# returns true if prime via low level prime division
# note that this is a PROBABILISTIC primality test
# i know how to check if a number is prime via brute force
# but this seems a lot more fun
# because we do 20 iterations declaring the candidate n
# probably prime incorrectly will be at most 1/(4^20) which is pretty good
# from: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
# and: https://www.w3schools.com/python/ref_func_pow.asp
def check_low_level_prime(candidate):
    for prime in lowLevelPrimes:
        if candidate % prime == 0:
            return False

    return True

# returns true if prime via rabinmiller
def rabin_miller(candidate):
    maxPowerofTwo = 0
    evenComponent = candidate - 1

    while evenComponent % 2 == 0:
        evenComponent >>= 1 # effectively dividing by 2
        maxPowerofTwo += 1


    def trialComposite(round_tester):
        if pow(round_tester, evenComponent, candidate) == 1: # this is just
r_t^eC % cand
            return False

        for i in range(maxPowerofTwo):
            if pow(round_tester, 2**i * evenComponent, candidate) == candidate -
1:
                return False

        return True

    numberOfTrials = 20
    for i in range(numberOfTrials):
        round_tester = random.randrange(2, candidate)

        if trialComposite(round_tester):
```

```
            return False

    return True


# this is based off the code found here:
https://gist.github.com/djego/97db0d1bc3d16a9dcb9bab0930d277ff
def multiplicative_inverse(e, lambda_n):
    inv = eea(e, lambda_n)[0]
    if inv < 1:
        inv += lambda_n
    return inv

# This is Euclid's Extended Algorithm for finding the multiplicative inverse of
two numbers
def eea(a, b):
    if b == 0:
        return (1, 0)
    (q, r) = (a//b, a%b)
    (s, t) = eea(b, r)
    return (t, s-(q*t))




# These are our encryption methods for the two types of input

# String based encryption:
# For strings we first convert each character to a number before
encrypting/decrypting.
def encryptText(keys, text):
    e, n = keys
    cipher = [pow(ord(char), e, n) for char in text]
    return cipher

def decryptText(keys, text):
    d, n = keys
    plain = [chr(pow(char, d, n)) for char in text]
    return ''.join(plain)


# Number based encryption
# If we get an input as a number, the encrytpion is pretty simple, just C = M^e %
n
def encryptNum(keys, num):
```

```
    e, n = keys
    cipher = pow(num, e, n)
    return cipher

# same with decryption, M = C^d % n
def decryptNum(keys, num):
    d, n = keys
    plain = pow(num, d, n)
    return plain

if __name__ == "__main__":
    main()
```