main.py:

```
#!/usr/bin/env python3

import rsa

def readKey(keyFile):
	# read a key from a specified file

	# open, read, and close the file
	file = open(keyFile, 'r')
	key = [int(x) for x in file.read().split(', ')]
	file.close()
	return key

def writeKey(keyFile, key):
	# write a key to a specified file

	# open, write, and close the file
	file = open(keyFile, 'w')
	file.write('{0}, {1}'.format(key[0], key[1]))
	file.close()
	return 0

def readData(encryptedFile):
	# read encrypted data from a specified file

	# open, read, and close the file
	file = open(encryptedFile, 'r')
	encryptedData = [int(x) for x in file.read().split(', ')]
	file.close()
	return encryptedData

def writeData(encryptedFile, encryptedData):
	# write encrypted data to a specified file

	# open, write, and close the file
	file = open(encryptedFile, 'w')
	file.write(', '.join(str(char) for char in encryptedData))
	file.close()
	return 0


if __name__ == "__main__":
	quit = 0 # set to 1 to quit program
```

```
	pubKey, privKey = None, None # set to None to make it easy to check if the keys are set

	while quit != 1:
	print("What would you like to do?")
	# get user input in lowercase
	userInput = input("(G)enerate keys, (W)rite keys to file, (L)oad keys from file, (E)ncrypt,
(D)ecrypt, (Q)uit: ").lower()
	if userInput == "g": # generate keys
	pubKey, privKey = rsa.genKeys(1024, 40)
	print("pubKey: {}\nprivKey: {}".format(pubKey, privKey))
	print("Keys generated!")

	elif userInput == "w": # write current keys to file
	if pubKey == None or privKey == None: # keys are not set
		print("Error: The keys are not defined. Please either generate keys or load them
from files.")
	else: # keys are set
		# get file names from user input
		pubFile = input("Where would you like to store the public key? ['pubKey']: ")
		privFile = input("Where would you like to store the private key? ['privKey']: ")

		# write pubKey to file
		if pubFile == "": # default to 'pubKey'
		print("writing pubKey to: 'pubKey'")
		writeKey('pubKey', pubKey)
		else:
		print("writing pubKey to: '{}'".format(pubFile))
		writeKey(pubFile, pubKey)

		# write privKey to file
		if privFile == "": # default to 'privKey'
		print("writing privKey to: 'privKey'")
		writeKey('privKey', privKey)
		else:
		print("writing privKey to: '{}'".format(privFile))
		writeKey(privFile, privKey)

	elif userInput == "l": # load keys from file
	# get file names from user input
	pubFile = input("Where is the public key located? ['pubKey']: ")
	privFile = input("Where is the private key located? ['privKey']: ")

	# load pubKey from file
	if pubFile == "": # default to 'pubKey'
```

```
                print("loading pubKey from: 'pubKey'")
                pubKey = readKey('pubKey')
        else:
                print("loading pubKey from: '{}'".format(pubFile))
                pubKey = readKey(pubFile)

        # load privKey from file
        if privFile == "": # default to 'privKey'
                print("loading privKey from: 'privKey'")
                privKey = readKey('privKey')
        else:
                print("loading privKey from: '{}'".format(privFile))
                privKey = readKey(privFile)

        elif userInput == "e": # encrypt data (and save to a file)
        if pubKey == None or privKey == None: # keys are not set
                print("Error: The keys are not defined. Please either generate keys or load them
from files.")
                continue
        else: # keys are set
                plaintext = input("Enter a message to encrypt: ")
                ciphertext = rsa.encrypt(plaintext, pubKey)
                print("Encrypted data: {}".format(ciphertext))

                # save encrypted data to a file
                encryptedFile = input("Where do you want to save the encrypted data?
['encryptedData']: ")

                if encryptedFile == "": # default to 'encryptedData'
                print("writing encrypted data to: 'encryptedData'")
                writeData('encryptedData', ciphertext)
                else:
                print("writing encrypted data to: '{}'".format(encryptedFile))
                writeData(encryptedFile, ciphertext)

        elif userInput == "d": # decrypt data
        if pubKey == None or privKey == None: # keys are not set
                print("Error: The keys are not defined. Please either generate keys or load them
from files.")
                continue
        else: # keys are set
                encryptedFile = input("Where is the encrypted data located? ['encryptedData']: ")

                if encryptedFile == "": # default to 'encryptedData'
```

```
                ciphertext = readData('encryptedData')
                else:
                ciphertext = readData(encryptedFile)

                # decrypt and display data
                plaintext = rsa.decrypt(ciphertext, privKey)
                print("Decrypted message: {}".format(plaintext))

        elif userInput == "q": # quit the program
        quit = 1

        else: # the user has entered an unknown option
        print("Error: unknown option:", userInput)
```

rsa.py:

```
#!/usr/bin/env python3

import random

def gcd(a, b):
        # find gcd of a & b
        # uses the euclidean algorithm
        while b: # b != 0
        a, b = b, a % b
        return a

def millerRabin(n, k):
        """
        Based on:
        https://rosettacode.org/wiki/Miller–Rabin_primality_test
        """

        # use the Miller-Rabin Primality Test to return True if n is probably prime, or False if n is
definitely composite
        # uses k rounds

        # find s and d so that: n - 1 = (2^s) * d
        # s must be a positive int
        # d must be an odd positive int
        s = 0
        d = n - 1
        while d % 2 == 0: # while d is even
        d>>=1 # bitwise right shift to divide d by 2
        s+=1 # increment s by 1
```

```
        def testComposite(a):
        # test if n is composite with base a, by testing following conditions:
        # a^d === 1 (mod n)
        # a^((2^r) * d) === -1 (mod n) for some 0<=r<s
        if pow(a, d, n) == 1: # if a^d % n == 1
        return False # n is strong probable prime to base a
        for r in range(s):
        if pow(a, 2**r * d, n) == n - 1: # if a^((2^r) * d) % n == - 1
                return False # n is strong probable prime to base a
        return True # n failed the tests, so n is definitely composite

        for r in range(k): # k being the number of trials
        a = random.randrange(2, n) # in range of 2, n-1
        if testComposite(a):
        return False # n is deemed composite by the tests

        return True # n passed all tests, so it is probably prime

def isPrime(n, k):
        # return true if n is probably prime, and false if n is definitely composite
        # use k rounds in miller rabin test

        # all primes < 1000
        primeList = [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, 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, 677, 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]

        if n in primeList: # if n is in primeList
        return True # n is definitely prime

        for prime in primeList:
        if n % prime == 0:
        return False # n is composite

        if not millerRabin(n, k):
        return False # n is definitely composite
```

```
        return True # n passed tests, it is probably prime

def primeFinder(b, k):
        # find a prime of b bits using the miller rabin primality test with k rounds


        while True:
        n = random.SystemRandom().getrandbits(b) # random num of b bits
        if isPrime(n, k):
        return n # n is probably prime, so return n

def encrypt(plaintext, pubKey):
        # encrypt the given plaintext with the public key
        e, n = pubKey # derive e and n from the public key

        ciphertext = [] # encrypted text stored as a list of integers (representing characters)

        for char in plaintext:
        ciphertext.append(pow(ord(char), e, n)) # calculate the new value of the character as
(char^e) % n

        return ciphertext # return the encrypted message

def decrypt(ciphertext, privKey):
        # decrypt the given ciphertext using the private key
        d, n = privKey # derive d and n from the private key

        plaintext = [] # plaintext stored as a list of integers (representing characters)

        for char in ciphertext:
        plaintext.append(chr(pow(char, d, n))) # calculate the new value of the character as
(char^d) % n

        return ''.join(plaintext)

def genKeys(b, k):
        # generate public and private keys using primes of b bits with k rounds of the miller rabin
primality test
        p = primeFinder(b, k) # find a prime
        q = primeFinder(b, k) # find a prime

        n = p * q # compute n

        phi = (p - 1) * (q - 1) # compute phi
```

```
# find a value of e where 2 < e < phi and gcd(e, phi) = 1
while True:
e = random.randrange(3, phi)
if gcd(e, phi) == 1:
break

d = pow(e, -1, phi) # compute d as the modular multiplicative inverse of e % phi

pubKey = [e, n] # set the public key
privKey = [d, n] # set the private key

return pubKey, privKey
```