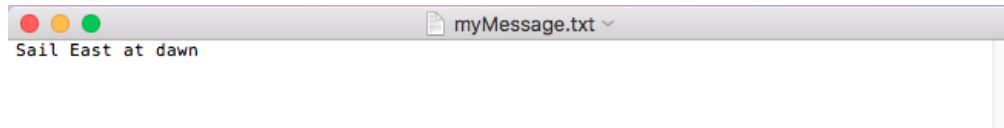


messageEncrypt

I. Encrypt

The program prompts the user to select encrypt or decrypt. If the user selects encrypt, the program will prompt the user to input the name of the text file containing the message to be encrypted. It will then output a text file containing an encrypted message, a text file containing a public key, and a text file containing a private key.

If we have a .txt file myMessage:



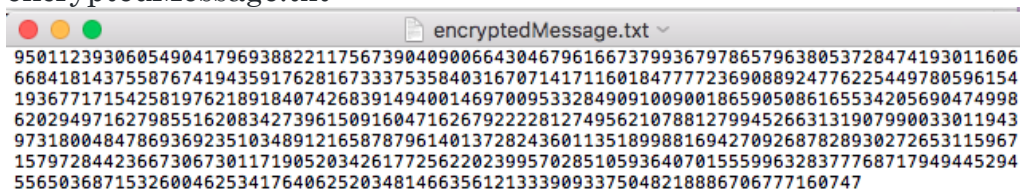
saved in the current directory, and we run:

```
Ryans-MacBook-Pro:message-encrypt ryantine$ python messageEncrypt.py
```

we will receive following prompts and answer them so to encrypt myMessage.txt:

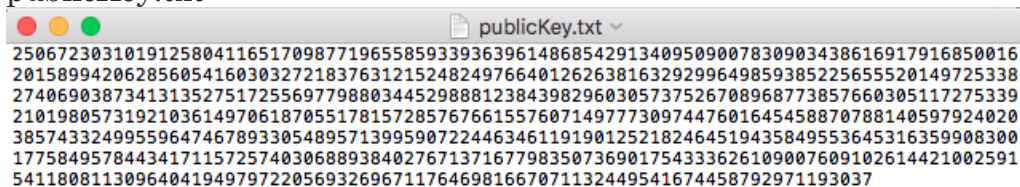
```
Would you like to encrypt or decrypt a message? encrypt
Enter the name of the text file containing your message to be encrypted or decrypted: myMessage.txt
Please find a file named encryptedMessage.txt containing your encrypted message, a file named publicKey.txt
containing your public key which may be stored anywhere and a file named privateKey.txt which must be stored
safely
```

The following files will then appear in the current directory:
encryptedMessage.txt:

A screenshot of a macOS-style text editor window. The title bar shows three colored window control buttons on the left and a document icon followed by the filename 'encryptedMessage.txt' on the right. The main text area contains a single line of very long, alphanumeric encrypted text.

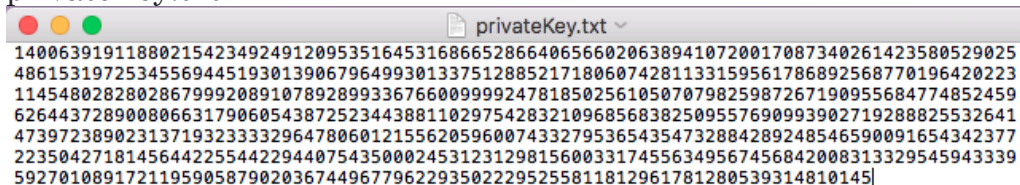
```
950112393060549041796938822117567390409006643046796166737993679786579638053728474193011606
66841814375587674194359176281673337535840316707141711601847772369088924776225449780596154
193677171542581976218918407426839149400146970095332849091009001865905086165534205690474998
620294971627985516208342739615091604716267922228127495621078812799452663131907990033011943
973180048478693692351034891216587879614013728243601135189988169427092687828930272653115967
157972844236673067301171905203426177256220239957028510593640701555996328377768717949445294
5565036871532600462534176406252034814663561213339093375048218886706777160747
```

publicKey.txt:

A screenshot of a macOS-style text editor window. The title bar shows three colored window control buttons on the left and a document icon followed by the filename 'publicKey.txt' on the right. The main text area contains a single line of very long, alphanumeric public key data.

```
250672303101912580411651709877196558593393639614868542913409509007830903438616917916850016
201589942062856054160303272183763121524824976640126263816329299649859385225655520149725338
274069038734131352751725569779880344529888123843982960305737526708968773857660305117275339
210198057319210361497061870551781572857676615576071497773097447601645458870788140597924020
385743324995596474678933054895713995907224463461191901252182464519435849553645316359908300
177584957844341711572574030688938402767137167798350736901754333626109007609102614421002591
54118081130964041949797220569326967117646981667071132449541674458792971193037
```

privateKey.txt:

A screenshot of a macOS-style text editor window. The title bar shows three colored window control buttons on the left and a document icon followed by the filename 'privateKey.txt' on the right. The main text area contains a single line of very long, alphanumeric private key data.

```
140063919118802154234924912095351645316866528664065660206389410720017087340261423580529025
486153197253455694451930139067964993013375128852171806074281133159561786892568770196420223
114548028280286799920891078928993367660099992478185025610507079825987267190955684774852459
626443728900806631790605438725234438811029754283210968568382509557690993902719288825532641
47397238902313719323332964780601215562059600743327953654354732884289248546590091654342377
223504271814564422554422944075435000245312312981560033174556349567456842008313329545943339
59270108917211959058790203674496779622935022295255811812961781280539314810145|
```

II. Decrypt

If the user selects decrypt, the program will then prompt the user to input the name of a text file containing an encrypted message, the name a text file containing the public key associated with that message, and the name of the text file containing the private key associated with that message. The program will output a text file containing the decrypted message.

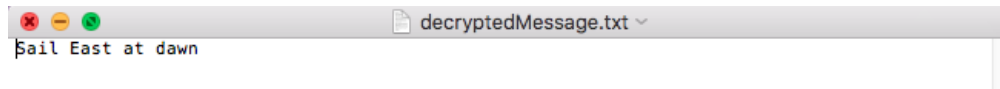
If we have the text files containing an encrypted message, and the public and private keys associated with that message saved in the current directory, and we run:

```
Ryans-MacBook-Pro:message-encrypt ryantine$ python messageEncrypt.py
```

we will receive the following prompts, and answer so to decrypt our message (we will continue from our example above and use the files encryptedMessage.txt, publicKey.txt, and privateKey.txt; these could be saved under any names when executed by the user).

```
Would you like to encrypt or decrypt a message? decrypt
Enter the name of the text file containing your message to be encrypted or decrypted: encryptedMessage.txt
Enter the name of a the text file containing the public key: publicKey.txt
Enter the name of the text file containing private key: privateKey.txt
Please find a file named decryptedMessage.txt containing your decrypted mesaage
```

The following file will appear in the current directory:



Which contains our decrypted message!

III. Implementation

This process is completed using optimal asymmetric encryption padding and the RSA algorithm. We imported SHA.256 from Crypto.Hash for hash functions when padding. In the RSA algorithm, the Rabin Miller primality test is used to determine if very large random numbers are prime:

```
def isPrime(num):
    """Return True if the number is prime, and False otherwise."""
    # Two basic cases where number can be quickly seen as prime or not prime
    if (num<2):
        return False
    elif num%2==0:
        return num == 2 # Two is the only even prime number

    # Start of Rabin Miller Primality Test
    s = num - 1
    counter1 = 0
    # if s is even keep halving it
    while s%2 == 0:
        s = s/2
        # increment counter1
        counter1 += 1
    trial = 8 # Number of times we will check num's primality. Accuracy is improved with increased trials
    while trial>0:
        rando = random.randrange(2,num)
        modNum = pow(rando,s,num) # modNum is equal to rando^r mod num
        if modNum != 1: # Rabin miller test does not apply if v = 1
            counter2 = 0
            while modNum != (num - 1):
                # case that would mean that num is not prime
                if counter2 == counter1-1:
                    return False
                else:
                    # increment counter2
                    counter2 += 1
                    # update v
                    modNum = (modNum**2)%num
            trial -= 1
        # if none of these conditions have been met, num is likely true
    return True
```

Function generateLargePrime() calls isPrime() which implements the rabin miller test:

```
def generateLargePrime():
    """Returns a prime number in the range from 2^1023 to (2^1024)-1"""
    while True:
        num = random.randrange(2**(1023), 2**(1024)-1)
        if isPrime(num):
            return num
```

The encryption algorithm also features a recursive implementation of the extended GCD algorithm, which is used to calculate modular inverse:

```
def extendedGCD(a, b):
    """Returns gcd, x and y so that a*x+b*y = gcd(x,y)"""
    # Base case (when a = 0)
    if a == 0:
        return (b,0,1)
    # Recursive case
    else:
        gcd, x, y = extendedGCD(b%a,a)
        return (gcd, y-(b/a)*x, x)

def modularInverse(a,mod):
    """Returns the value whose product with (a % mod) is equal to 1"""
    gcd,x,y = extendedGCD(a,mod)
    return x%mod
```