Home Work 6
Yacob Alemneh
RSA Encryption/Decryption Program

The program I wrote has three major functions, namely: keySetup(), encryption(), decryption(). These are the functions that are called in the main functions. But to implement them I have written six other functions to help implement the above functions. They are: euclidsGcd(a,b), randomPrime(lim), modularExp(a,p,n), extendedEuclids(a,b), modularInverse(a, m), fermatTest(n). I will first explain these functions work and then explain how the first three functions utilize them.

euclidsGcd(a,b): Takes two integer arguments, calculates their GCD recursively and returns it.

randomPrime(lim): This functions takes an integer to be used as a limit to generate random large prime integers. It enters a loop and it raises the given limit to the power of 100 for the first integer its going to output, namely q, it then generates another large random number q, but it makes sure that its at least 10^{95} larger than p. It then uses the fermatTest(n) function to check if both numbers are prime. If not, the while loop iterates again, until the numbers generated are checked by the fermatTest(n) and seen to be prime. It then returns the generated prime integers p and q.

modularExp(a, p, n): This funcition implements the modular exponentiation algorithm recursively. It takes a integers a, p and n and solves for $a^p \pmod n$. If p is 0 it returns 1. If p is one it returns a. If neither is true, it finds the remainder (r) and quotient (q) using the inbuilt python function *divmod* and checks if r is 1, if it is it returns a call on it self, *modularExp(*$a^2 \bmod n$, q, n) * a mod n. If r isn't 1 it returns a call on it-self *modularExp*($a^2 \bmod n$, q, n).

extendedEuclids(a, b): This function also calculates the GCD like **euclidsGCD(a,b)** with the added functionality that we can use this to calculate the modular inverse of a and b because it gives us x and y which we are going to use in the future. It returns g, x and y, where gcd(a,b) = ax + by.

modularInverse(a, b): This function takes two arguments, call on **extendedEuclids(a, b)** and gets g, x and y. Makes sure that g = 1, if it is it returns the modular inverse, which is x % b.

fermatTest(n): This function uses fermatTest to check if the given n is prime or not. It is the function called in **randomPrime(lim)**. It uses the formula $a^{p-1} \equiv 1 \pmod p$. It checks whether the given congruence is valid for 5 arbitrary values of a < n – 2. Makes sure its valid all five times, by solving the congruence using the **modularExp(a, p, n)** function. If the number is tested to be prime, it returns True.

All these functions are tied together by the keySetup(), encryption(), decryption() which are called in the main function.

The first function is keySetup(). When keySetup() starts it initializes two empty lists, one to store the private key, and one to store the public key. It generates two random primes using the **randomPrime(lim)** function. Calculates *n = p* q, phi = (p – 1) * (q – 1)* and sets up as *e = 65537*. It checks if *phi* and *e* are co-prime, if not it generates another p and q until they are. It then opens two files *public_key.txt* and *private_key.txt* to write the public key and private keys in. After writing the public key in *public_key.txt*, it calls on **modularInverse(e, phi)** to calculate the private key, *d*, and it writes it to *private_key.txt*.

The second function is encryption(). When it starts it initializes two empty lists to store the public key retrieved from *public_key.txt* and the other one to store message to be encrypted which is read from *message.txt.* After reading the keys from *public_key.txt* it and storing them in a list, it assigns the first element in the list to *n (which is equal to p * q)* and the second element to *e.* After retrieving the message from the .txt file and storing it in a list as an int, it uses the modular exponentiation algorithm **modularExp(mesToEncrypt, e, n)** with the shown inputs, there by encrypting the message. After its done its stores the encrypted message into *ciphertext.txt*.

The third function is decryption(). When it starts it initializes three lists to store the private key, public key and cipher message. After reading *ciphertext.txt, public_key.txt, private_key.txt* and it retrieves the values *n, e, d* from them, it uses the modular exponentiation algorithm **modularExp(mesToDecrypt, d, n)**, with the shown values to calculate the value of the decrypted message and store it in *decrypted_message.txt*.

We call these functions from a main function. One problem I had made me struggle when writing this program was trying to process the large integers that were generated. For example, using an iterative test to check for primality made the program crash on my computer, so I had to solve it recursively. The other thing was finding the modular inverse, the program works fine for smaller primes (15 decimals) and small messages (around 15 decimals), but for some reason it doesn't decrypt the message correctly for the large numbers that were required. I've thoroughly checked trying to find out what the problem is, added some lines of code redundantly to check whether number are prime or co-prime, but I couldn't find the problem. What I think the problem is has to do with how I calculate the key, when I find the modular inverse for *e and phi*, I believe it might be giving me a wrong value for *d*. But I am still not sure if that's the case since it works fine for small values. Everything else works fine.

I wrote the program in Python3 so to run program I use *python3 rsa.py.*