Network Security Lab 2

Narendra Gautam Sontu - 002241534

Code: https://github.com/gautamsontu/Network_Security/blob/main/RSA.py

Key Generation Time:

- **Key Size**: 2048, 4096 bits
- **Time Taken for Key Generation**: Approximately 0.8974 secs and 14.1669 secs

Code Explanation:

```
import random
from sympy import randprime
import time



class RSA:
    def __init__(self, keysize):
        self.keysize = keysize
        self.public_key = None
        self.private_key = None
        self.generate_keys()

    def gcd(self, a, b):
        while b:
            a, b = b, a % b
        return a

    def modinv(self, a, m):
        m0, x0, x1 = m, 0, 1
        while a > 1:
            q = a // m
            m, a = a % m, m
            x0, x1 = x1 - q * x0, x0
        return x1 + m0 if x1 < 0 else x1

    def generate_keys(self):
        # Generating two random prime numbers p and q
        p = randprime(2 ** (self.keysize // 2 - 1), 2 ** (self.keysize // 2))
        q = randprime(2 ** (self.keysize // 2 - 1), 2 ** (self.keysize // 2))
        n = p * q
        phi = (p - 1) * (q - 1)
        # Choosing e such that 1 < e < phi and gcd(e, phi) = 1
        e = random.randrange(2, phi)
        while self.gcd(e, phi) != 1:
```

```python
            e = random.randrange(2, phi)
        # Computing d, the modular multiplicative inverse of e (mod phi)
        d = self.modinv(e, phi)
        # Public key (e, n) and private key (d, n)
        self.public_key = (e, n)
        self.private_key = (d, n)

    def encrypt(self, plaintext):
        e, n = self.public_key
        ciphertext = [pow(ord(char), e, n) for char in plaintext]
        return ciphertext

    def decrypt(self, ciphertext):
        d, n = self.private_key
        plaintext = ''.join([chr(pow(char, d, n)) for char in ciphertext])
        return plaintext


def main():
    keysize = int(input("Enter the key size: "))

    start_time = time.time()
    rsa = RSA(keysize)
    key_generation_time = time.time() - start_time

    print(f"Public Key: {rsa.public_key}")
    print(f"Private Key: {rsa.private_key}")
    print(f"Key generation time: {key_generation_time:.4f} secs")

    while True:
        choice = input("Choose an option: (1) Encrypt (2) Decrypt (3) Exit: ")

        if choice == '1':
            plaintext = input("Enter plaintext to encrypt: ")
            start_time = time.time()
            ciphertext = rsa.encrypt(plaintext)
            encryption_time = time.time() - start_time
            print(f"Ciphertext: {ciphertext}")
            print(f"Encryption time: {encryption_time:.4f} secs")
        elif choice == '2':
            ciphertext = input("Enter ciphertext to decrypt: ")
            try:
                ciphertext = list(map(int, ciphertext.strip('[]').split(',')))
                start_time = time.time()
                decrypted_text = rsa.decrypt(ciphertext)
                decryption_time = time.time() - start_time
                print(f"Decrypted Text: {decrypted_text}")
```

```python
                print(f"Decryption time: {decryption_time:.4f} secs")
            except ValueError as e:
                print(f"Error: {e}.Entered wrong ciphertext.")
        elif choice == '3':
            break
        else:
            print("Invalid choice! Please choose again from the given options.")

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

Here,

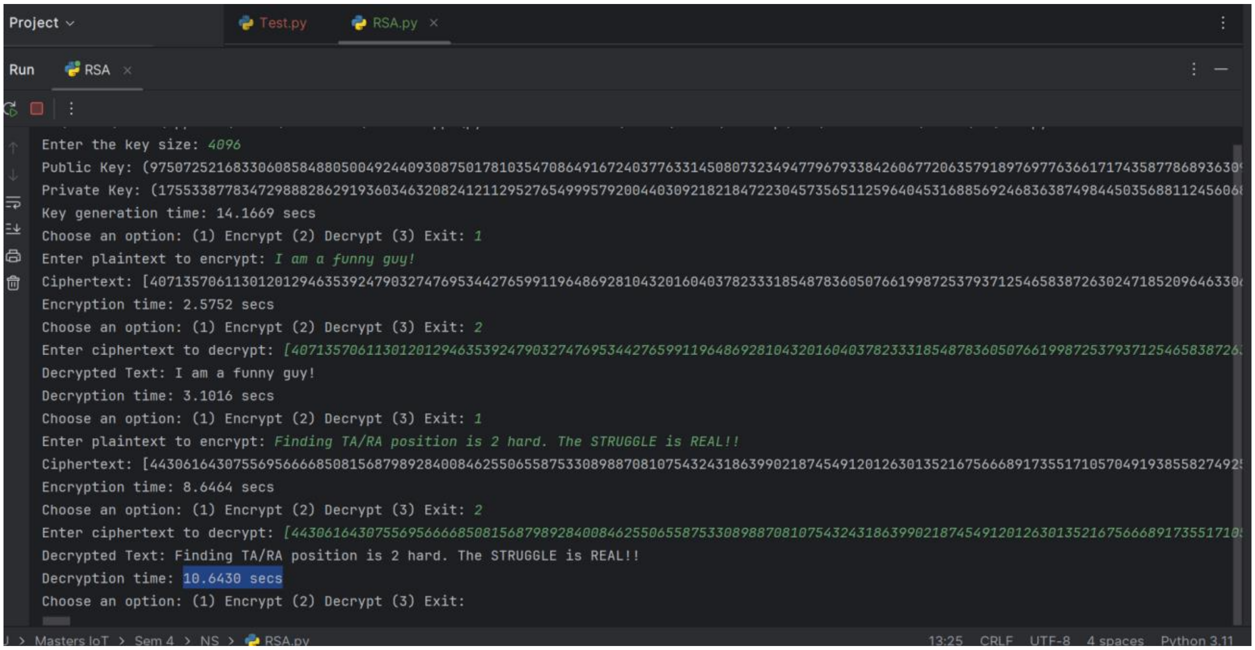
- In the RSA Class,
  - **`__init__`**: Initializes the class with a given key size and calls the `generate_keys` method to create the public and private keys.
  - **`gcd`**: Calculates the greatest common divisor of two numbers `a` and `b` using the Euclidean algorithm.
  - **`modinv`**: Computes the modular inverse of `a` modulo `m` using the extended Euclidean algorithm.
  - **generate_keys**: Generates two random prime numbers p and q, calculates n (the product of p and q), and phi (the totient of n). Then, it finds an e that is coprime with phi and calculates d, the modular inverse of e. Finally, it sets the public key (e, n) and the private key (d, n).
  - **encrypt:** Encrypts the plaintext by converting each character to its ASCII value, raising it to the power of e modulo n, and returns the list of encrypted values.
  - **decrypt:** Decrypts the ciphertext by raising each encrypted value to the power of d modulo n, converting it back to a character, and returning the plaintext.

- In the main function (handles user interaction).

  - **Key Size Input**: Prompts the user to enter the desired key size.
  - **Key Generation**: Measures the time taken to generate the keys and prints the public and private keys along with the key generation time.
  - **User Options**: Provides options to encrypt, decrypt, or exit.
  - **Encrypt**: Prompts the user to enter plaintext, encrypts it, and prints the ciphertext and encryption time.
  - **Decrypt**: Prompts the user to enter ciphertext, decrypts it, and prints the decrypted text and decryption time.
  - **Exit**: Exits the program.

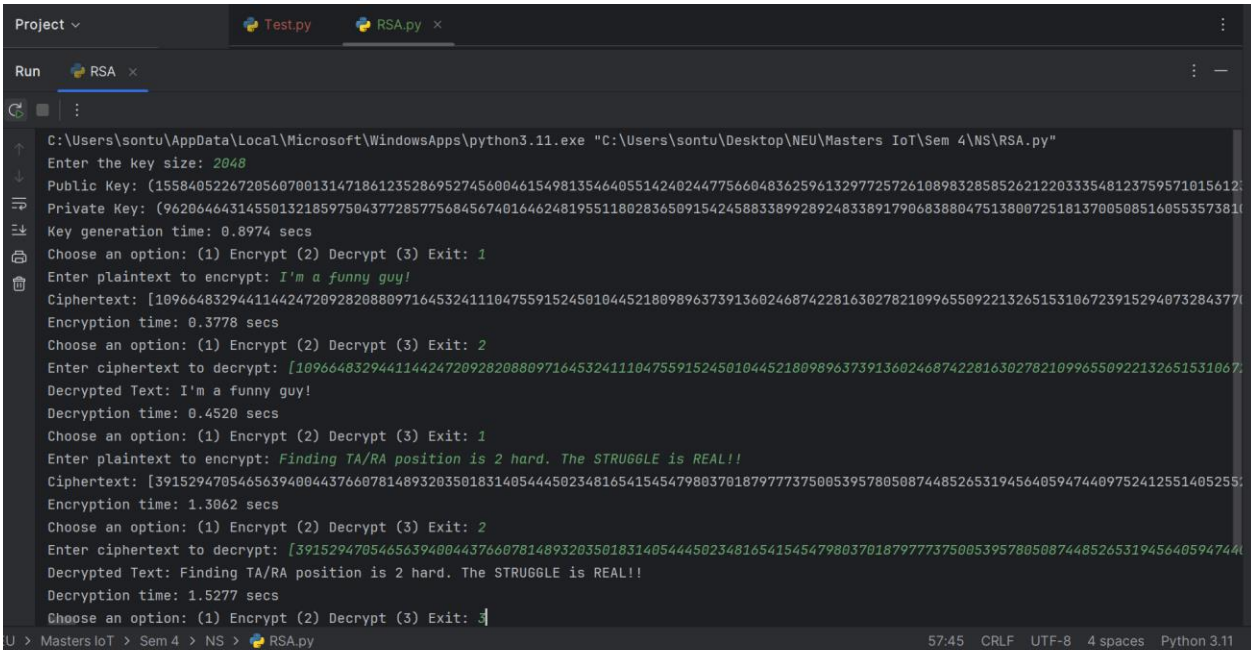
Results of Testing:

1. **Key Size – 4096 bits:**



a. **Plaintext**: *"I am a funny guy!"*
   - **Encryption Time**: 2.5752 secs
   - **Decryption Time**: 3.1016 secs
b. **Plaintext**: *"Finding TA/RA position is 2 hard. The STRUGGLE is REAL!!"*
   - **Encryption Time**: 8.6464 secs
   - **Decryption Time**: 10.6430 secs
c. **Plaintext**: *"1234567890"*
   - **Encryption Time**: 1.7161 secs
   - **Decryption Time**: 2.0156 secs
d. **Plaintext**: *"9246171850"*
   - **Encryption Time**: 1.6479 secs
   - **Decryption Time**: 2.0810 secs
e. **Plaintext**: *"9876543"*
   - **Encryption Time**: 1.0787 secs
   - **Decryption Time**: 1.3154 secs

2. **Key Size – 2048 bits:**



a. **Plaintext**: *"I am a funny guy!"*
   - **Encryption Time**: 0.3778 secs
   - **Decryption Time**: 0.4520 secs

b. **Plaintext**: *"Finding TA/RA position is 2 hard. The STRUGGLE is REAL!!"*
   - **Encryption Time**: 1.3062 secs
   - **Decryption Time**: 1.5277 secs

c. **Plaintext**: *"1234567890"*
   - **Encryption Time**: 0.2922 secs
   - **Decryption Time**: 0.3023 secs

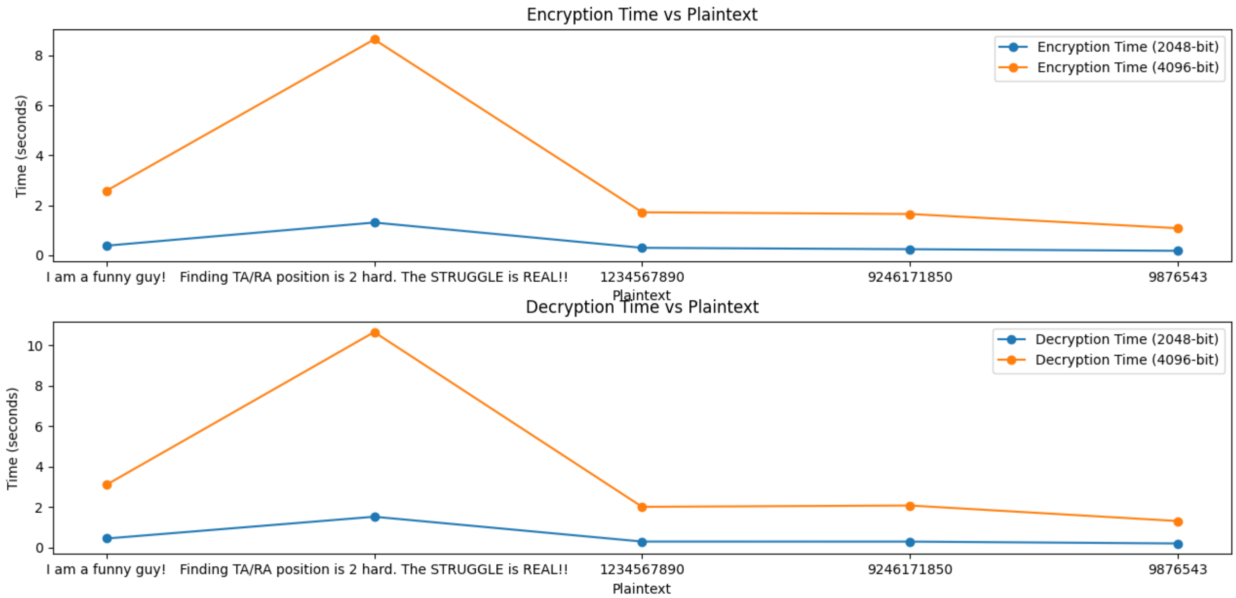
d. **Plaintext**: *"9246171850"*
   - **Encryption Time**: 0.2372 secs
   - **Decryption Time**: 0.2999 secs

e. **Plaintext**: *"9876543"*
   - **Encryption Time**: 0.1724 secs
   - **Decryption Time**: 0.2077 secs

Findings/Discussions

3. **Key Generation:**
   - **Efficient for Smaller Key Sizes**: The 2048-bit key generation is relatively quick and efficient.
   - **Increased Time for Larger Keys**: The 4096-bit key generation takes significantly longer due to the complexity of generating large prime numbers.
4. **Encryption and Decryption:**
   - **Shorter Messages**: Encryption and decryption are relatively quick for shorter messages.

- **Longer Messages**: As the message length increases, the time for both encryption and decryption increases linearly.
- **Decryption Takes Longer**: Decryption generally takes slightly longer than encryption due to larger exponentiation operations (since the decryption exponent `d` is typically larger than the encryption exponent `e`).



Conclusion

The RSA algorithm's performance is a trade-off between security (key size) and computational efficiency. While larger key sizes offer better security, they require more time for key generation, encryption, and decryption. The implementation demonstrates the RSA algorithm's practical usage and highlights the importance of choosing an appropriate key size based on security requirements and computational capabilities.