**Practical No: 7**

**Practical Name: Study and implement a program for RSA Cipher.**

**Roll No: 22BCP250**

**Name: Anusi Patel**

**Division and batch: Div4 G7**

**Original Approach**

**Introduction:** The RSA algorithm is a public-key cryptography method for secure data transmission. It relies on the difficulty of factoring large prime numbers. RSA uses two keys: a public key for encryption and a private key for decryption. Only the intended recipient can decrypt the message, maintaining data security.

**Example:**

* **Key Generation**:
  + Choose two large prime numbers: p=3 and q=11
  + Compute n=p×q=33
  + Compute ϕ(n)=(p−1)×(q−1)=2×10=20
  + Choose e such that 1<e<ϕ(n) and e is co-prime to ϕ(n), let e=3
  + Compute d such that (e×d)mod  ϕ(n)=1, let d=7
* **Public Key**: (e=3, n=33)  
  **Private Key**: (d=7, n=33)
* **Encryption**:
  + Message m=7
  + Ciphertext c=m^e mod  n=7^3 mod  33=343 mod  33=13
* **Decryption**:
  + Ciphertext c=13
  + Decrypted message m=c^d mod  n=13^7 mod  33=62748517 mod  33=7

**Source Code:**

# Function to calculate GCD

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

# Function to find a suitable public exponent e

def find\_e(phi):

for e in range(2, phi):

if gcd(e, phi) == 1:

return e

return None

# Function to calculate modular inverse

def mod\_inverse(e, phi):

t1, t2 = 0, 1

r1, r2 = phi, e

while r2 > 0:

quotient = r1 // r2

r1, r2 = r2, r1 - quotient \* r2

t1, t2 = t2, t1 - quotient \* t2

if r1 == 1:

return t1 % phi

return None

# Function for modular exponentiation

def mod\_exp(base, exp, mod):

result = 1

while exp > 0:

if exp % 2 == 1:

result = (result \* base) % mod

base = (base \* base) % mod

exp //= 2

return result

# Function to convert numerical message into string of digits

def message\_to\_numeric(message):

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

numeric\_value = ""

for char in message:

numeric\_index = alphabet.index(char)

numeric\_value += f"{numeric\_index:02d}" # Format as two digits

return int(numeric\_value) # Convert the entire string to an integer

# Function to convert numeric value back to message

def numeric\_to\_message(numeric\_value):

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

chars = []

temp\_value = str(numeric\_value)

# Process the numeric value from the start

while len(temp\_value) >= 2:

pair = temp\_value[:2] # Take first two characters

numeric\_index = int(pair)

chars.append(alphabet[numeric\_index]) # Convert index back to character

temp\_value = temp\_value[2:] # Remove the first two characters

# Handle any remaining single digit if necessary

if temp\_value:

numeric\_index = int(temp\_value)

chars.append(alphabet[numeric\_index]) # Convert index back to character

return ''.join(chars)

# RSA encryption and decryption process

def rsa\_encryption(p, q, message):

n = p \* q

phi = (p - 1) \* (q - 1)

# Automatically find a valid e

e = find\_e(phi)

print(f"Automatically selected public exponent e: {e}")

# Convert the message to a numeric value

numeric\_message = message\_to\_numeric(message)

print(f"Numeric equivalent of message: {numeric\_message}")

# Calculate the private key d

d = mod\_inverse(e, phi)

# Encrypt the numeric message

encrypted\_message = mod\_exp(numeric\_message, e, n)

print(f"Encrypted message (numeric): {encrypted\_message}")

# Decrypt the numeric message

decrypted\_message = mod\_exp(encrypted\_message, d, n)

print(f"Decrypted message (numeric): {decrypted\_message}")

# Convert decrypted numeric back to message

original\_message = numeric\_to\_message(decrypted\_message)

print(f"Original message: {original\_message}")

# Example usage

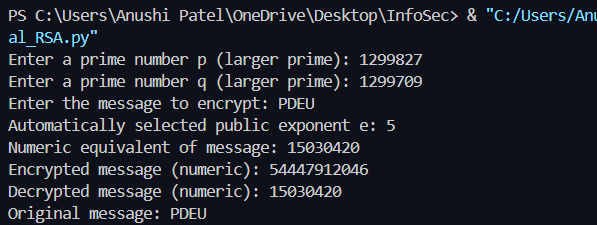
p = int(input("Enter a prime number p (larger prime): "))

q = int(input("Enter a prime number q (larger prime): "))

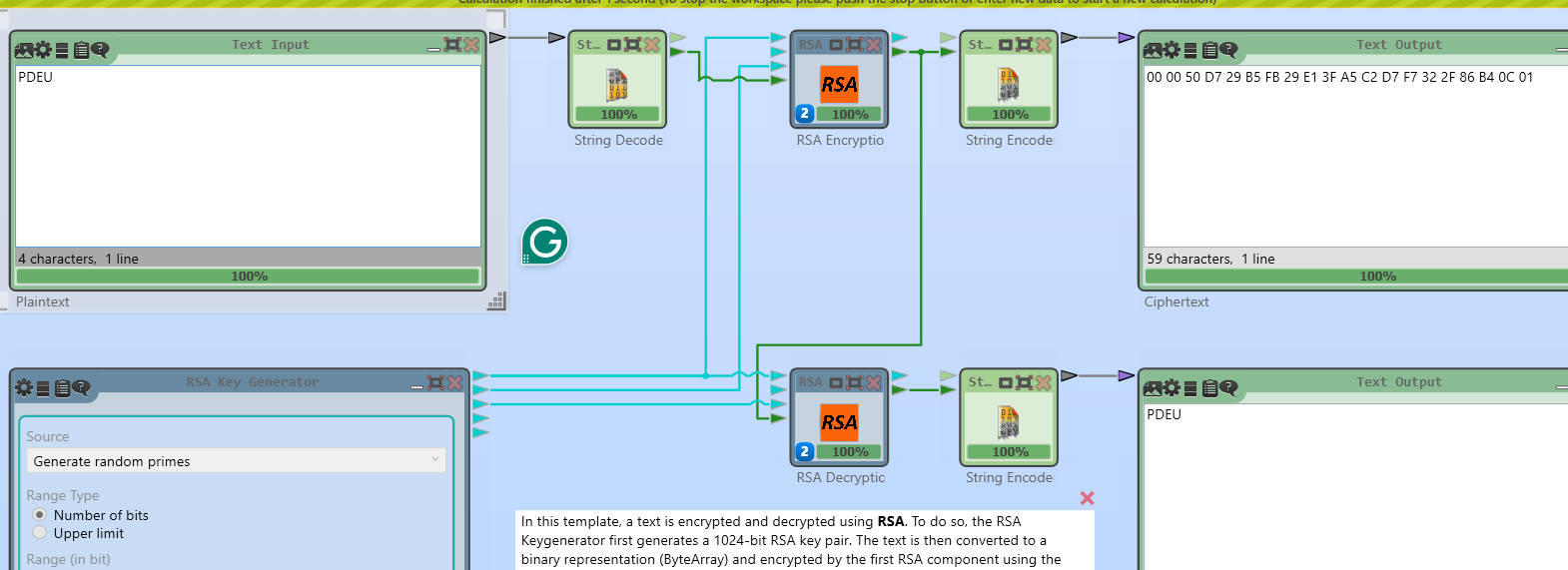
message = input("Enter the message to encrypt: ")

rsa\_encryption(p, q, message)

**Output:**

****

**Cryptool:**

****

**Cryptanalysis** **on RSA:**

1. **Brute-Force Attack:**

* **Description:** A brute-force attack on RSA involves trying every possible private key to decrypt a given ciphertext.
* **Impact:** While theoretically possible, brute-force attacks are impractical for RSA due to the large key size (e.g., 2048-bit keys). The immense number of combinations makes this method infeasible in practice with current computational power.

1. **II. Known-Plaintext Attack:**

* **Description:** Involves having a portion of the plaintext and its corresponding ciphertext to deduce the private key.
* **Impact:** If an attacker has access to a portion of plaintext and the corresponding ciphertext, they may attempt to reverse-engineer the encryption process. However, without access to the private key, this attack is challenging in RSA due to the one-way nature of the encryption function.

1. **III. Chosen-Plaintext Attack (CPA):**

* **Description:** In a CPA, the attacker can choose arbitrary plaintexts and obtain their corresponding ciphertexts to gain information about the private key.
* **Impact:** RSA is vulnerable to CPA if not implemented correctly. However, when proper padding schemes like OAEP (Optimal Asymmetric Encryption Padding) are used, the attack becomes impractical.

**Conclusion:**

The RSA algorithm is a cornerstone of public-key cryptography, enabling secure communication through a pair of keys: a public key for encryption and a private key for decryption. Its security relies on the difficulty of factoring large prime numbers, making it widely used in applications like SSL/TLS and digital signatures. While RSA is robust, it requires proper key sizes (recommended at least 2048 bits) and secure implementation to protect against potential vulnerabilities. As technology evolves, RSA remains vital for ensuring data confidentiality and integrity in the digital landscape.

**References:**

* Zhou, X., & Tang, X. (2011, August). Research and implementation of RSA algorithm for encryption and decryption. In *Proceedings of 2011 6th international forum on strategic technology* (Vol. 2, pp. 1118-1121). IEEE.