



# UNIVERSITY OF SCIENCE AND TECHNOLOGY AT ZEWAIL CITY

COMMUNICATION AND INFORMATION ENGINEERING

DISCRETE MATHEMATICS

**MATH 308**

RSA ENCRYPTION

---

Mohamed Mostafa      201600236  
Phone Number:      01090111040

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Implementation</b>	<b>3</b>
2.1	Auxiliary Functions . . . . .	3
2.1.1	Greatest Common Divisor . . . . .	3
2.1.2	Extended Euclidean Algorithm . . . . .	4
2.1.3	Square and multiply . . . . .	5
2.1.4	Primality test . . . . .	5
2.1.5	Random prime generation . . . . .	6
2.2	Key Generation . . . . .	6
2.3	Encryption . . . . .	7
2.4	Decryption . . . . .	7
<b>3</b>	<b>Result</b>	<b>8</b>
<b>4</b>	<b>Conclusion</b>	<b>8</b>
<b>5</b>	<b>References</b>	<b>9</b>
<b>6</b>	<b>Appendix</b>	<b>10</b>

---

## List of Figures

1	Test output of the implementation . . . . .	8
---	---	---

---

# 1 Introduction

With the dawn of the digital age, previous privacy insurance methods were no longer applicable. New attacks emerged which all can result in the wrongful acquisition of data. Thus motivated by the old ways of secrets exchange, mostly used in military such as the enigma machine or the oldest known cipher named the scytale, efficient encryption methods were introduced into the digital age. Discussed in this paper is an implementation of these encryption methods called RSA.

## 2 Implementation

The encryption method is mainly divided into 3 main sections namely Key Generation, encryption, and decryption with the usage of some auxiliary also implemented in the python script.

### 2.1 Auxiliary Functions

#### 2.1.1 Greatest Common Divisor

Code:

---

```
def gcd(a, b):  
    while b != 0:  
        a, b = b, a % b  
    return a
```

---

#### Proof of correctness:

Assumption to prove: if  $a=bq+r$  where  $a,b,q,r$  are integers then  $\gcd(a,b)=\gcd(b,r)$

Proof:

Suppose an integer  $d$  which divides both  $a$  and  $b$ , then  $a-bq=r$  is also divides  $d$  since  $a$  has been subtracted by a multiple of  $b$ . Since  $a$ ,  $b$ , and  $r$  has the same divisor then  $\gcd(a,b)=\gcd(b,r)$

---

### 2.1.2 Extended Euclidean Algorithm

Code:

---

```
def eea(R1, modulo):
    if(gcd(R1, modulo) != 1):
        raise Exception(f"{R1} has no inverse in modulo {modulo}")
    newR = R1
    oldR = modulo
    newT = 1
    oldT = 0
    while(newR % oldR != 1):
        q = int(oldR / newR)
        oldT, newT = newT, oldT - q * newT
        oldR, newR = newR, oldR - q * newR

    #This while loop is only implemented to
    #return an inverse within the modulo range
    while(newT < 0):
        newT = newT + modulo
    return newT
```

---

#### Proof of correctness:

Assumption: The  $\gcd(m, x)$ , where  $m$  and  $x$  are integers, is only equivalent to 1 meaning that they are coprime when  $x$  has a distinct multiplicative inverse in modulo  $m$ .

Proof:

Suppose integers  $a$  and  $b$  where  $ax \bmod m = bx \bmod m$  where  $a$  and  $b$  are distinct values within modulo  $m$ . Thus  $(a-b)x = 0 \bmod m$  or  $(a-b)x$  is a multiple of  $m$ . Since  $x$  and  $m$  are coprime, then  $a-b$  must be the integer multiple of  $m$ . The previous conclusion contradicts the fact that  $a$  and  $b$  are distinct integer values within modulo  $m$ , thus  $a$  must equal  $b$ . Since  $(a-b)x = 0 \bmod m$ , then  $ax = 1 \bmod m$  where  $a$  is the inverse of  $x$ .

---

### 2.1.3 Square and multiply

Code:

---

```
def sam(expBase,expPower,module):
    binPrime=bin(expPower)[3:]
    result=expBase%module
    for i in binPrime:
        result=(result*result)%module
        if i=="1":
            result=(result*expBase)%module
    return result
```

---

#### Discussion:

The above algorithm does not need a proof of correctness, as this is only a formalized method to divide the exponent into simple steps to find the minimal number of steps to complete the modular exponentiation. The algorithm works by converting the exponent into a binary number. At the first 1 bit on the right the number is set by the modulus of the exponent base. The algorithm loops over the rest of the bits. If the current bit is 0 then the current number is squared and the modulus is calculated and the current number is updated by that number. If the current bit is 1 then the same operation as 0 occurs but with the extension of multiplying the current number with the original exponent base then updating the current number by the modulus of the number calculated.

### 2.1.4 Primality test

Code:

---

```
def isPrime(x):
    if sam(2,x-1,x)==1:
        return True
    else: return False
```

---

The above code implements the Fermat's little theorem to check whether a number is prime or not. The theorem states that if  $p$  is a prime and  $p \nmid a$  then  $a^{p-1} = 1 \pmod p$ .

---

### 2.1.5 Random prime generation

Code:

---

```
def primeGen(bitLength):  
    #the number is ORed with 1 to generate an odd number  
    #to divide the search space by 2  
    p=random.getrandbits(bitLength) | 1  
    while(not isPrime(p)):  
        p=random.getrandbits(bitLength) | 1  
    return p
```

---

Discussion:

A random number odd number is generated with a given bit length and checked if the number is prime or not.

## 2.2 Key Generation

Code:

---

```
def keyGeneration(bitLength):  
    #2 prime numbers with a given bitlength  
    #are generated  
    p=primeGen(bitLength)  
    q=primeGen(bitLength)  
    n=p*q  
    phi=(p-1)*(q-1)  
    e=random.randint(1, phi-1)  
    while(gcd(e, phi)!=1):  
        e=random.randint(1, phi-1)  
  
    #the inverse of e in modulo phi is  
    #calculated and used along with n  
    # as the private key  
    privateKey=(n, eea(e, phi))  
  
    #n and e are used as the public key  
    publicKey=(n, e)  
    return publicKey, privateKey
```

---

---

## 2.3 Encryption

Code:

---

```
def encryption(message, publicKey):  
    return sam(message, publicKey[1], publicKey[0])
```

---

This function calculates cipher by calculating  $m^e \bmod n$ .

## 2.4 Decryption

Code:

---

```
def decryption(cipher, privateKey):  
    return sam(cipher, privateKey[1], privateKey[0])
```

---

This function calculates message by calculating  $c^d \bmod n$ .

**Proof of correctness:**

Required to prove:  $c^d \bmod n = m^{ed} \bmod n = m \bmod n$

Proof:

First we analyze the power of m to see what is it equivalent to.

d is the inverse of e in modulo phi, thus  $ed = 1 \bmod \phi$ . This proves that ed is a multiple of phi which is equivalent to  $(p-1)(q-1)$ , in other words,  $ed-1 = h(p-1)(q-1)$ .

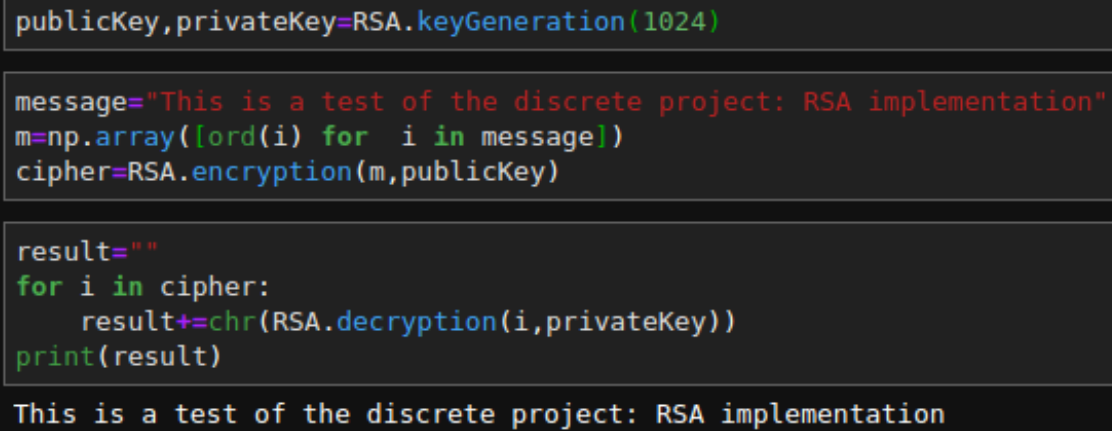
Second we analyze the modulo and split it into 2.

$n = p \cdot q$ . we first analyze  $m^{ed} \bmod p$  as if we found that the result is the same in both mod p and q then it would yield the same result in mod pq as it would only be a multiple from it.  $m \cdot m^{h(p-1)(q-1)} \bmod p$ . Since p is a prime and using Fermat's little theorem the equation simplifies to  $m \bmod p$  and the same can be applied to q. This results in  $m^{ed} \bmod n = m \bmod n$ .



---

### 3 Result

A screenshot of a code editor with a dark background. The code is written in Python and implements RSA encryption and decryption. The code is as follows:

```
publicKey,privateKey=RSA.keyGeneration(1024)

message="This is a test of the discrete project: RSA implementation"
m=np.array([ord(i) for i in message])
cipher=RSA.encryption(m,publicKey)

result=""
for i in cipher:
    result+=chr(RSA.decryption(i,privateKey))
print(result)
```

The output of the code is displayed at the bottom of the editor:

```
This is a test of the discrete project: RSA implementation
```

Figure 1: Test output of the implementation

The project implements encryption of character basis. This could be vulnerable to statistical attacks as the RSA implementation is deterministic meaning that for the same input the algorithm would output the same cipher. This could be solved by implementing a block usage of the encryption method instead of this stream method. Another result that was found was that most of the computational time was at generating a random prime number. The generator implemented in this code uses trial and error which could be improved by employing any other method.

#### Important

Anaconda needs to be installed to view the Demo notebook and run the test.

### 4 Conclusion

In the end the RSA method implemented in this code is referred to as the schoolbook implementation of RSA which is vulnerable to various attacks. Nevertheless the more sophisticated method employs the schoolbook RSA as its core which contains all the discrete computations needed. Also a complexity analysis cannot be done on the implementation as it relies on a random number turning out to be a prime. This is solved in the codes that are available to the public by employing faster primality tests and prime generators.

---

## 5 References

[1]<https://github.com/pycrypto/pycrypto>

[2]<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>

[3]Discrete and CIE 581 lectures

---

## 6 Appendix

---

```
import random
import Crypto.Util.number
import Crypto.Random

# crypto library is imported to only test
# its speed compared to my implementation

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def eea(R1, modulo):
    if(gcd(R1, modulo) != 1):
        raise Exception(f"{R1} has no inverse in modulo {modulo}")
    newR = R1
    oldR = modulo
    newT = 1
    oldT = 0
    while(newR % oldR != 1):
        q = int(oldR / newR)
        oldT, newT = newT, oldT - q * newT
        oldR, newR = newR, oldR - q * newR
    # This while loop is only implemented to
    # return an inverse within the modulo range
    while(newT < 0):
        newT = newT + modulo
    return newT

def sam(expBase, expPower, modulo):
    binPrime = bin(expPower)[3:]
    result = expBase % modulo
    for i in binPrime:
        result = (result * result) % modulo
        if i == "1":
            result = (result * expBase) % modulo
    return result

def isPrime(x):
```

---

```

    if sam(2, x-1, x) == 1:
        return True
    else: return False

def primeGen(bitLength):
    #the number is ORed with 1 to generate an odd number
    #and increase the step by 2 to divide the search space by 2
    p = random.getrandbits(bitLength) | 1
    while(not isPrime(p)):
        p = random.getrandbits(bitLength) | 1
    return p
# return Crypto.Util.number.getPrime(bitLength, randfunc=Crypto.Ran

def keyGeneration(bitLength):
    p = primeGen(bitLength)
    q = primeGen(bitLength)
    n = p * q
    phi = (p-1) * (q-1)
    e = random.randint(1, phi-1)
    while(gcd(e, phi) != 1):
        e = random.randint(1, phi-1)
    privateKey = (n, eea(e, phi))
    publicKey = (n, e)
    return publicKey, privateKey

def encryption(message, publicKey):
    return sam(message, publicKey[1], publicKey[0])

def decryption(cipher, privateKey):
    return sam(cipher, privateKey[1], privateKey[0])

```

---