VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**DISCRETE STRUCTURE ESSAY**

**RSA CRYPTOSYSTEM USING MODULAR ARITHMETIC**

*Supervisor*: **MR TRAN LUONG QUOC DAI**

*Author*: **NGUYEN PHUONG TAI – 521H0480**

*Class* : **21H50201**

*Class of* : **25**

**HO CHI MINH CITY, 2023**

VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**DISCRETE STRUCTURE ESSAY**

**RSA CRYPTOSYSTEM USING MODULAR ARITHMETIC**

*Supervisor*: **MR TRAN LUONG QUOC DAI**

*Author*: **NGUYEN PHUONG TAI – 521H0480**

*Class* : **21H50201**

*Class of* : **25**

**HO CHI MINH CITY, 2023**

ACKNOWLEDGEMENT

I want to express my gratitute towards Mr. Tran Luong Quoc Dai for helping me in answering my many questions as well as making recommendations for valuable learning resources that I can complete this essay.

**ESSAY ACCOMPLISHED**

**AT TON DUC THANG UNIVERSITY**

I ensure that this is my personal essay, accomplished with the support of Mr. Tran Luong Quoc Dai. All of the research content, implementation in this essay are true and have never been published under any circumstances before. All of the statistics in this essay are referenced in the references section and are used for analysing, commenting and documentation purposes.

Also, the essay also contains many comments, analysis and documentations from various authors, organizations and are referenced as well.

**If there are any fraudulence, I am fully responsible for my research content.** Ton Duc Thang University does not have anything to do with any copyright infringement that I am responsible for (if there are any).

*Ho Chi Minh City, April 1, 2023*

*Author*

*( signature and full name )*

*Nguyen Phuong Tai*

SUPERVISOR’S CONFIRMATION AND EVALUATION

**Supervisor’s confirmation**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ho Chi Minh City, April 1, 2023

(signature and full name)

**Supervisor’s evaluation**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ho Chi Minh City, April 1, 2023

(signature and full name)

SUMMARY

The advancement of technology nowadays has brought many conveniences to our daily lives. It makes the world a smaller place, connecting peoples from all over the world and thus, making personal information and data privacy susceptible to many cyber threats such as hacking, injection.

The birth of cryptography has brought many improvements to cybersecurity such as data encryption, authentication which ensures data confidentiality and integrity to keep users’ information safe.

Consequently, the field of cryptography is mostly based around the foundation of discrete strutures concepts such as modular arithmetic.

TABLE OF CONTENTS

ACKNOWLEDGEMENT i

SUPERVISOR’S CONFIRMATION AND EVALUATION iii

SUMMARY iv

TABLE OF CONTENTS 1

LIST OF TABLES, IMAGES 2

CHAPTER 1 – FINDING THE MODULAR INVERSE OF N 3

1.1 Key Concepts 3

1.1.1 Congruence 3

1.1.2 Modular Multiplicative Inverse 3

1.2 Extended Euclidean Algorithm 4

1.2.1 Overview 4

1.2.2 Algorithm 4

1.2.3 Generalization 5

1.3 Finding the modular inverse of a number 6

1.3.1 Theoretical Concept 6

1.3.2 Python Implementation 7

CHAPTER 2 – RSA CRYPTOSYSTEM 9

2.1 Introduction 9

2.2 Mathematical Concepts 10

2.2.1 Prime Number Generation 10

2.2.2 Prime Factorization 17

2.2.3 Euler’s totient function 18

2.3 RSA Cryptography 19

2.3.1 Algorithm 19

2.3.2 Example 20

2.3.3 Implementation 21

2.3.4 Efficiency and security 24

2.3.5 Threats and Limitations 25

2.3.6 Conclusion and Recommendation 26

REFERENCES 28

SELF – EVALUATION 29

LIST OF TABLES, IMAGES

[Table 1.1: Extended Euclidean table for 105 and 75 6](#_Toc132200661)

[Table 1.2: Extended Euclidean table for 11 and 5 7](#_Toc132200662)

[Figure 1.3: Extended Euclidean Function 7](#_Toc132200663)

[Figure 1.4: Finding Modular Inverse Funtion 8](#_Toc132200664)

[Figure 1.5: Printing the function result 9](#_Toc132200665)

[Figure 1.6: Function result 9](#_Toc132200666)

[Figure 2.1: Public – key cryptography demonstration 10](#_Toc132200667)

[Figure 2.2: Generate Random Prime Number Brute Force Method 11](#_Toc132200668)

[Figure 2.3: Table of numbers 12](#_Toc132200669)

[Figure 2.4: Cross out mutiples of 2 12](#_Toc132200670)

[Figure 2.5: Cross out mutiples of 3 12](#_Toc132200671)

[Figure 2.6: Final Table 12](#_Toc132200672)

[Figure 2.7: Generate Random Prime Number Sieve Method 12](#_Toc132200673)

[Figure 2.8: Generating Large Prime Number 15](#_Toc132200674)

[Figure 2.9: isPrimeMillerRabin Function 15](#_Toc132200675)

[Figure 2.10: smallPrime List 16](#_Toc132200676)

[Figure 2.11: millerRabinTest Function 16](#_Toc132200677)

[Figure 2.12: Printing the Big Primes by sizes 16](#_Toc132200678)

[Figure 2.13: Results of Big Prime Generator 17](#_Toc132200679)

[Figure 2.14: ASCII Alphabet 20](#_Toc132200680)

[Figure 2.15: isCoPrime Function 21](#_Toc132200681)

[Figure 2.16: rsa Function 22](#_Toc132200682)

[Figure 2.17: Message Prompt 23](#_Toc132200683)

[Figure 2.18: Demo of ‘Hello World’ 23](#_Toc132200684)

[Figure 2.19: ‘rsa’ Library Implementation 24](#_Toc132200685)

[Figure 2.20: Two implementations 24](#_Toc132200686)

CHAPTER 1 – FINDING THE MODULAR INVERSE OF N

* 1. Key Concepts

1.1.1 Congruence

Congruence is one of the fundamental concepts in modular arithmetic. It is the relation between two integers, which we will call them **a** and **b**, that have the same remainder when divided by a positive integer called the modulus, which we will denote it as **m**. The concept of congruence modulo m can be denoted as:

**Keep in mind that the parentheses means that the mod m applies to both side of the equation.**

For example, the number 40 and 22 have the same remainder when dividing them with 18, which is 4. We can rewrite their relation as follow:

If you notice, we can also rewrite the relation as:

In this situation, k will be equal to 1 and thus we can also write the congruence relation as:

1.1.2 Modular Multiplicative Inverse

In general arithmetic, the inverse of a number is the value that, when we multiply it with the original number, will return a product of 1.

For example, the inverse of 5 is because when we multiply them together, we will get a result of 1. We can simply interpret the general multiplicative inverse of number **a** as:

In modular arithmetic, things are different from general arithmetic. The inverse of a number is the value that, when we multiply it with the original number, will return a product that is congruent to 1 modulo a given modulus. So the value of the modular inverse can be varied based on the modulus. Another difference is that, a number can have a modular inverse on one modulus but not on another modulus.

For example, when we have the number 5 and a modulus of 11, the modular inverse of 5 is 9, because (5\*9) mod 11 = 1. However, when working with modulus 10, we will not be able to find the modular inverse since there are no numbers that multiplied by 5 can give a product congruent to 1 modulo 10.

Therefore, the existence of the modular inverse is based on the original number **a** and the modulus **m**, it can only exists if and only if the GCD(**a, m**) = 1 (or greatest common divisor between **a** and **m** is equal to 1).

The modular multiplicative inverse can be denoted as:

Sometimes, it can also be denoted as:

* 1. Extended Euclidean Algorithm

1.2.1 Overview

The original Euclidean Algorithm is mostly best known for its use in finding the greatest common divisor between two numbers or GCD. The extended version of the Euclidean Algorithm is also used for the same purpose, but it also finds the value **x** and **y** such that:

Where **a** and **b** are the integers we want to find the GCD value between them, **x** and **y** are the coefficients of Bézout’s Identity, which states that for two non-zero integers **a** and **b** with a greatest common divisor **d**, there exist integers **x** and **y** such that:

1.2.2 Algorithm

The idea of the extended algorithm is to find the GCD just like the original Euclidean algorithm first and then the work our way backward to get the coefficients.

Example: Find the GCD of 94 and 32

We can write as follows:

Every row is written under the format of . As **a** and **b** are the input numbers and **r** is the remainder. We can see that the GCD between 94 and 32 is 2 as the row with is the last row before . The number **a** will be equal to previous **b** and **b** will be equal to previous **a mod b** on every new iteration.

Now, we will go backward recursively to get the coefficients **x** and **y** of **a** and **b.**

The row can be rewritten as . (1)

The row can be rewritten as . (2)

We can subtitute (2) into (1) to get the result as:

Therefore, the coefficients for the GCD between 94 and 32 are and .

1.2.3 Generalization

For

We can create a table which has 5 columns including: {i}, {qi}, {ri}, {si}, {ti}. As i is the index of every row, qi as the quotient at row i, ri is the remainder of row i, si is the coefficient of number **a** of row i, ti is the coefficient of number **b** of row i. We can re written the linear combination between **a** and **b** in every row as:

(1)

First, we set , and start from . We are going to go through every row such that (2). Combining (1) and (2) will give us the following equation:

After some simplification we get:

(3)

From (1) and (3), we have and **.** We finally completed the whole formula for every column in our generalization table. **Note that we are going to stop iterating when .**

Let’s take an example of finding the GCD and the coefficients of 105 and 75:

We set **a** = 105 and **b** = 75

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 0 |  | 105 | 1 | 0 |
| 1 |  | 75 | 0 | 1 |
| 2 | 105//75 = 1 | 105-75\*1=30 | 1-0\*1 = 1 | 0-1\*1=-1 |
| 3 | 75//30 = 2 | 75-30\*2 = 15 | 0-1\*2= -2 | 1-(-1)\*2=3 |
| 4 | 30//15 = 2 | 30-15\*2=0 | 1-(-2)\*2= 5 | -1-3\*2 = -7 |

Table 1.1: Extended Euclidean table for 105 and 75

Therefore, we can see that the coefficients for 105 and 75 are -2 and 3 respectively with the GCD of 15.

* 1. Finding the modular inverse of a number

1.3.1 Theoretical Concept

Consider 2 number **a** and **m** which have a GCD of 1can be written under the Extended Euclidean Algorithm as:

And then we, apply modulo **m** (mod **m**) to both side of the equation:

Therefore, we can conclude that **x**, or the coefficient of number **a** in the Extended Euclidean Algorithm, is the modular inverse of **a mod m.** Note that the value of x has to be in the range of **{1,2,…m - 1}.**

As we have discussed before, the condition for the modular inverse to exist is that the GCD between **a** and **m** has to be equal to 1 or GCD(**a, m**) = 1.

Let’s take an example such as finding the modular inverse of 5 mod 11.

We also going to create the generalization table as above, **note that we can arrange 5 or 11 as a and b or vice versa** since it does not affect to the final result.

So, I am going to assign **a = 11** and **b = 5.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 0 |  | 11 | 1 | 0 |
| 1 |  | 5 | 0 | 1 |
| 2 | 11//5 = 2 | 11-5\*2 = 1 | 1-0\*2 = 1 | 0-1\*2=-2 |
| 3 | 5//1 = 5 | 5-1\*5=0 | 0-1\*5 = -5 | 1-(-2)\*5 = 11 |

Table 1.2: Extended Euclidean table for 11 and 5

Therefore, we can see that the modular inverse of 5 mod 11 is -2. But since we want the value to be in the range of **{1, 2, …m - 1},** we just need to add the **m** value to the modular inverse value and that will give us 9.

So, we finally come to the conclusion that **x = 9.**

We can also manually calculate this equation as 5\*9 = 45 and 45 mod 11 is 1, 1 mod 11 is also 1 so this is the correct result.

1.3.2 Python Implementation

def extendedEuclidGCD(a, b):  
 last\_x, last\_y = 1, 0  
 x, y = 0, 1   
  
 while b != 0:  
 quotient = a//b  
 remainder = a % b  
  
 a, b = b, remainder  
  
 x, last\_x = last\_x - x \* quotient, x  
 y, last\_y = last\_y - y \* quotient, y  
  
 return a, last\_x, last\_y

Figure 1.3: Extended Euclidean Function

This is the implementation of the Extended Euclidean Algorithm, **last\_x** and **last\_y** are the coefficients and of number **a** and **b,** while **x** and **y** are the current coefficients and **.** Before the while block, we can imagine that the index i is currently at 1.

When the while block begin, the index i is going to be {2, 3, 4,…}. In the while block, we are going to find the **quotient** of each iteration and assign **b** as the **remainder** and **a** as **b** like the normal Euclidean Algorithm, we then update the **x, y, last\_x, last\_y** coefficients for every iteration.

We assign since **.**

We also assign since **.**

Alongside that, we also update the **last\_x** and **last\_y** variable to **x** and **y** because **x** and **y** after every iteration become the old coefficients.

The while block stops when it reaches **.** Which means that the remainder value of the last iteration is 0, to get the coefficients of **a** and **b**, we go back 1 row and get the **last\_x** and **last\_y** value. Also, we will return **a** when **b** is 0 as it is the condition of the Euclidean Algorithm. The function will return the GCD value and the coefficients of **a** and **b.**

Finally, the function of finding the modular inverse of number **a** mod **m** is written as:

def inverseModulo(a, mod):  
 g, x, y = extendedEuclidGCD(a, mod)  
 if g != 1:  
 return "Modular inverse does not exist"  
 while x < 0:  
 x = x + mod  
 return x

Figure 1.4: Finding Modular Inverse Funtion

This function is going to calculate the modular inverse of a mod m, it going to take the GCD, coefficients of a and b from the above Extended Euclidean function and check whether if the GCD is equal to 1.

* If the GCD is not 1, the message of “Modular inverse does not exist” will be returned.
* Else, it going to return the modular inverse as the cofficient of **a** as **x** or **x + mod** until **x** is positive if **x** is negative.

print(inverseModulo(67, 105))  
print(inverseModulo(53, 62))  
print(inverseModulo(10, 5))

Figure 1.5: Printing the function result

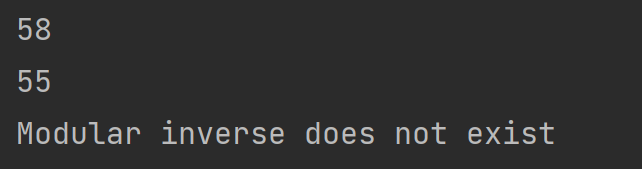


Figure 1.6: Function result

We can verify our results by taking taking the **a** value, multiplied it by the result and modulo the **m** value, check whether the remainder is equal to 1 or not.

CHAPTER 2 – RSA CRYPTOSYSTEM

2.1 Introduction

In our daily lives, we are used to sending messages and sensitive information, these data are usually susceptible to malicious attacks by hackers if not properly protected, so the need for a system that can provide layers of protection and security for users and organization information is undoubtedly one of the fundamental aspects for modern computers to function.

Therefore, computer scientists had created the cryptographic system or cryptosystem for short, it is based on the foundation of modular arithmetic. Modular arithmetic is proven to be helpful in dealing with large integers in many cryptographic systems such as the RSA cryptosystem.

A cryptosystem is the implementation of various cryptographic algorithm to perform various data confidential services. A typical cryptosystem has three main algorithms: key generation, encryption and decryption. Encryption is the process of turning plain data or message into unintelligible information called ciphertext, while decryption is the process of translating or reverting the ciphertext back into plain data or text that is understandable for human. Key generation is the process of creating keys that are used to encrypt and decrypt data.

In particular, the RSA (or Rivest – Shamir – Adleman) cryptosystem is a public – key cryptography, which is a system that uses a pair of public and corresponding private key. The public key is used for encrypting data and it can only be decrypted by a corresponding private key. While the public key can be publically shared to everyone, the private key demands to be kept secret. For example, person A wanted to send a message to person B, person B will be the only person to hold a private key to decrypt the message that A sent over while the public key is published for anyone to encrypt messages.

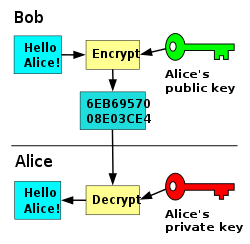


Figure 2.1: Public – key cryptography demonstration

The RSA cryptosystem consists of 4 main methods: creating keys, distributing keys, encryption and decryption. The basic property of RSA is to find three positive integers **e**, **d** and **n** such that:

In the formula, **n** is the modulus, **e** is the public encryption exponent and **d** is the private decryption exponent. The roles of these variables will be stated clearer in the algorithm section of this chapter.

2.2 Mathematical Concepts

2.2.1 Prime Number Generation

In the RSA cryptosystem algorithm, the security of the system is based on the difficulty of breaking (or factoring) the product of two large prime numbers. Therefore, the RSA algorithm is sometimes called a trapdoor function since we can know the product of two prime numbers but not the other way around.

For example, we can factoring 21 into 3 and 7 since the number is small but since the real implementation involves bigger prime numbers, up to the millions and billions. So it will get harder for human to factor a number into two primes that are very since it can be very time consuming.

To generate a random prime, the simplest methods that we use is to brute force every number starting from 1 and checking whether it is prime or not by a supportive. If the number is prime then we add it into a list and randomly select a number to return. Here is a implementation of the brute force method.

def generatePrimeNumberBrute(n):  
 listOfPrime = []  
 for i in range(1, n + 1):  
 if isPrime(i):  
 listOfPrime.append(i)  
 return random.choice(listOfPrime)  
  
  
def isPrime(n):  
 if n < 2:  
 return False  
 for i in range(2, int(math.sqrt(n)) +1):  
 if n % i == 0:  
 return False  
 return True

Figure 2.2: Generate Random Prime Number Brute Force Method

As we can guess, this function is easy to understand but extremly slow when working with larger numbers, and as cryptographic system like RSA, we do not want our system to be easily broken down so another method is need.

Sieve of Eratosthenes is one of the more efficient methods to generate prime numbers as it does not need to check every single number is prime or not by a supportive function but rather by using simple mathematic and logic.

For example, I want to find all of the prime numbers up to n. First, I would need to draw table of numbers starting from 2 and each row maximum value would be multiples of 10 such as 10, 20, 30, 40, … for simplicity, I would choose the number **n** as 10.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Figure 2.3: Table of numbers

And then I will marked out all of the multiples of 2 except for 2 like 4, 6, 8, 10. The red numbers are marked out.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Figure 2.4: Cross out mutiples of 2

Then I will marked out all of the multiples of 3 like 9.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Figure 2.5: Cross out mutiples of 3

We will repeat the process for all of the unmarked numbers like 5, 7 until eventually, we will have a table of unmarked numbers, those will be our prime numbers.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Figure 2.6: Final Table

So our prime numbers that are smaller than 10 are 2, 3, 5 and 7.

def sieveOfEratosthenes(n):  
 primeList = []  
 isPrimeList = [True for i in range(n + 1)]  
 num = 2  
 while num \* num <= n:  
 if isPrimeList[num]:  
 for i in range(num\*num, n+1, num):  
 isPrimeList[i] = False  
 num = num + 1  
 for i in range(2, n+1):  
 if isPrimeList[i]:  
 primeList.append(i)  
 return random.choice(primeList)

Figure 2.7: Generate Random Prime Number Sieve Method

Although the performance has been improved over the brute force method, it is still nowhere near our needed performance for the RSA algorithm, the usual size for the prime numbers must be around 1024 – bit to 2048 – bit long. Therefore, we need to have a better approach, an approach that may not be based on the deterministic of a prime number but rather based on the probability of a number becoming a prime number. Since many types of primality test that based on the deterministic of prime numbers usually cost a lot of resources and performance of the program.

For this approach to work, we need to know a little bit about Miller – Rabin primality test. A primality test function will take a number and tells us whether that number is a prime number or a composite number, which is a number that has more than 2 factors. For example, 49 is a composite number since it can be divisible by 1, 7 and itself.

The Miller – Rabin Test is going to check specific properties, which are known for prime values. For example, all of the prime numbers, except for 2, are odd value. So if a number is odd, it could hold the chance of becoming a prime number.

**Step one**, we need to generate an n – bit number, which is a number that range in decimal terms. For example, consider an 8 – bit integer number, it could range from to **.** But since the RSA algorithm require large prime numbers to make the prime factorization process difficult to obtain, we would atleast need the first bit to be 1 and the rest are either 0s or 1s as before. To do this, we change the range into in decimal terms. Again let’s consider an 8 – bit integer range, , we can see that the starting point number still have 8 – bit and the starting bit is 1. Therefore, our new range is in an 8 – bit scenario.

**Step two**, we do not consider any even number since 2 is the only even prime number, so let’s consider the input odd number **n.** We can write the following equation.

Where **s** is a positive integer and **d** is a positive odd integer. Now let’s choose an **a** integer, which is referred to as the **base.** The number is selected in the range of .

**Step three,** we calculate **.** If or then **n** is probably prime, the reason for this based on theoretical research of the algorithm, which stated that if one of these two congruence relations holds then **n** is a strong probable prime.

Since in step two, therefore we have.

So those are the condition for **n** to probably be a prime number. If the **x** value is not either of those that I have mentioned before, we go to step four.

**Step four,** we initiaze a while loop with the condition of **.** For each iteration, we calculate , if then n is probably a prime, or else if then n is not a prime number, if both conditions are not met then we assign and then repeat the whole loop process. We are essentially looping the value back to and by that logic, the condition still holds true just like the **x** value in step three, but the value of does not hold up anymore. If the whole looping process completed without result, we shall determine thatis probably not a prime number.

For example, in step one, we generate a number of 179 in 8 – bit just for examing purpose.

Step 2, we can prime factorization **,** so and and choose a base **.**

Step 3, we calculate **,** which matches the condition for a possible prime number of .

The implementation of Miller – Rabin alongside with the generating large number is as follows.

def generateBigPrimeNumber(bit):  
 while True:  
 num = random.randint(2\*\*(bit-1), 2\*\*bit - 1)  
 if isPrimeMillerRabin(num):  
 return num

Figure 2.8: Generating Large Prime Number

We use these codes to generate a large prime number in step 1, the implementation of the function isPrimeMillerRabin are shown here.

def isPrimeMillerRabin(n):  
 if n == 2 or n == 3:  
 return True  
 if n % 2 == 0:  
 return False  
  
 if n in smallPrime:  
 return True  
  
 for i in smallPrime:  
 if n % i == 0:  
 return False  
  
 d = n-1  
 while d % 2 == 0:  
 d = d//2  
  
 for i in range(128):  
 if not millerRabinTest(n, d):  
 return False  
  
 return True

Figure 2.9: isPrimeMillerRabin Function

The function is used to check for many basic properties of prime numbers such as whether it is odd or even, is the number in the smallPrime category already. We split into like the algorithm in step 2, because we want our answer to be as precise as possible, the for loop is initialized 128 times to perform the Miller – Rabin primality test. The smallPrime list is a list of prime numbers that are smaller than 1000, we check whether our number is a multiples of a smallPrime element or not, and we also check for the appearance of our number inside the list. The full list is written manually in Python just to keep some amount of time when dealing with small prime numbers.

smallPrime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]

Figure 2.10: smallPrime List

Finally, we going to implement the millerRabinTest function to complete the process of randomly choosing a prime large number for the RSA algorithm.

def millerRabinTest(n, d):  
 a = random.randint(2, n-2)  
 x = pow(a, d, n)  
 if x == 1 or x == n-1:  
 return True  
  
 while d != n-1:  
 y = pow(x, 2, n)  
 if y == n-1:  
 return True  
 elif y == 1:  
 return False  
  
 x = y  
 d = d\*2  
 return False

Figure 2.11: millerRabinTest Function

The function has 2 input numbers of **n,** **d** and perform all of the step 3 and 4 algorithm in our previous explanation. To try out this generator, you only need to make a call to the generateBigPrimeNumber function and passing in the length, or bit of your desire and print out the number.

print(generateBigPrimeNumber(256))  
print(generateBigPrimeNumber(32))  
print(generateBigPrimeNumber(64))

Figure 2.12: Printing the Big Primes by sizes

Here are the result.

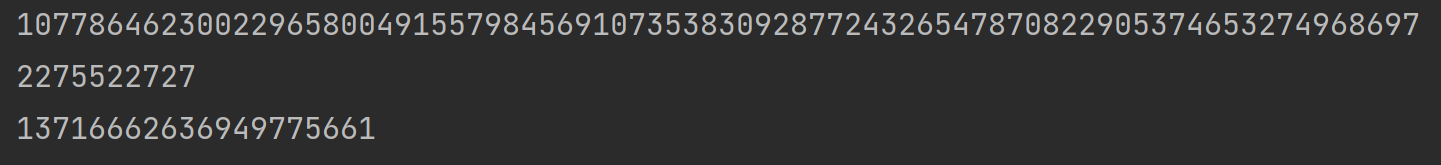


Figure 2.13: Results of Big Prime Generator

Bear in mind that these results do have a chance of being wrong, but since we run 128 tests on each number so the actual percentage is quite low.

2.2.2 Prime Factorization

Prime factorization is a process of analysing a number into a product of prime numbers. The algorithm for the process is described as follows:

**Step 1,** we start with the smallest prime number of 2, we will repeatedly diving our number with 2 if the operation has a remainder of 0.

**Step 2,** if the operation does not result in a remainder of 0, we will go to the next prime number of 3 and repeatedly dividing our number with 3. When the operation continue to not have a remainder 0, the next prime number of 5 is put into use and the process remain the same until our number is equal to 1.

Therefore, the stop condition for the algorithm is the input number has to be reduced to 1. For example, if we want to factorize 60 then we would have the following:

The input number of , the prime number starting from 2 as .

Since , we have to increase by 1 as

Therefore, we can write the prime factorization of 60 as:

If we notice carefully, since the prime factorization process factor a number into multiple primes, the prime factorization of a prime number is itself. Therefore, in the RSA cryptosystem, we want to make the factorization process of the product of 2 primes as hard as possible to keep our private key safe.

2.2.3 Euler’s totient function

Euler’s totient function is a function that return the number of positive integers from 1 to a given number **n** that are **co-prime (or relatively prime)** to **n**. For example, the function when given will give a value of 6, since the GCD between 1 and 9, 2 and 9, 4 and 9, 5 and 9, 7 and 9, 8 and 9 are all 1. The function has the following formula with a given **n**:

The value of in the formula are the prime factors of the number **n.** We will take the previous example of 60, we already know that the prime factors of 60 are 2, 3 and 5. We can subtitute these values into the function to calculate the number of co – prime numbers to 60.

We know that there are 16 co – prime values to 60. We can also test out with 9 since 9 can be rewritten into prime factors as .

Therefore, we can confirm that this formula is correct. Another special property that I want to talk about is the multiplicative characteristic of the function. For example, given 2 numbers **a** and **b**, if then we will have:

We could find the number of co – prime values of the product between **a** and **b**,if they themselves are co – primes.

Now, let’s get into the algorithm of the RSA algorithm, the basic concepts of modular arithmetic have been explained in the previous chapter.

2.3 RSA Cryptography

2.3.1 Algorithm

After obtaining the concepts of basic modular arithmetic, Euler’s totient function, prime factorization and large prime generator, we can go into the many steps of the RSA algorithm.

**Step 1**. We need to generate 2 prime numbers of **p** and **q** and calculate their product as . The number **n** will be the keys modulus. The preferred size for **p** and **q** are around 1024 and 2048 – bit long.

**Step 2**. We calculate , and as we have known before that , and since they are both prime numbers. Therefore we have the following function:

And then we pick an **e** number in the range of and **e** has to be co – prime with , this **e** value is going to be our public key exponent.

**Step 3**. We going to find our private key exponent by finding the modular inverse of **e** modulo . We are going to find

The steps for finding modular inverse have been discussed in the previous chapter.

After those step, we finally obtain our pair of public and private keys, the public key is and the private key is . The following steps are going to be about encrypting and decrypting messages.

**Step 4**. We are going to convert the message into a number **m**, we commonly use the ASCII alphabet. To simply put, the alphabet is a collection between a letter and its corresponding value. For example, the decimal ASCII value of the letter ‘A’ is 65, for ‘B’ is 66 and so on.

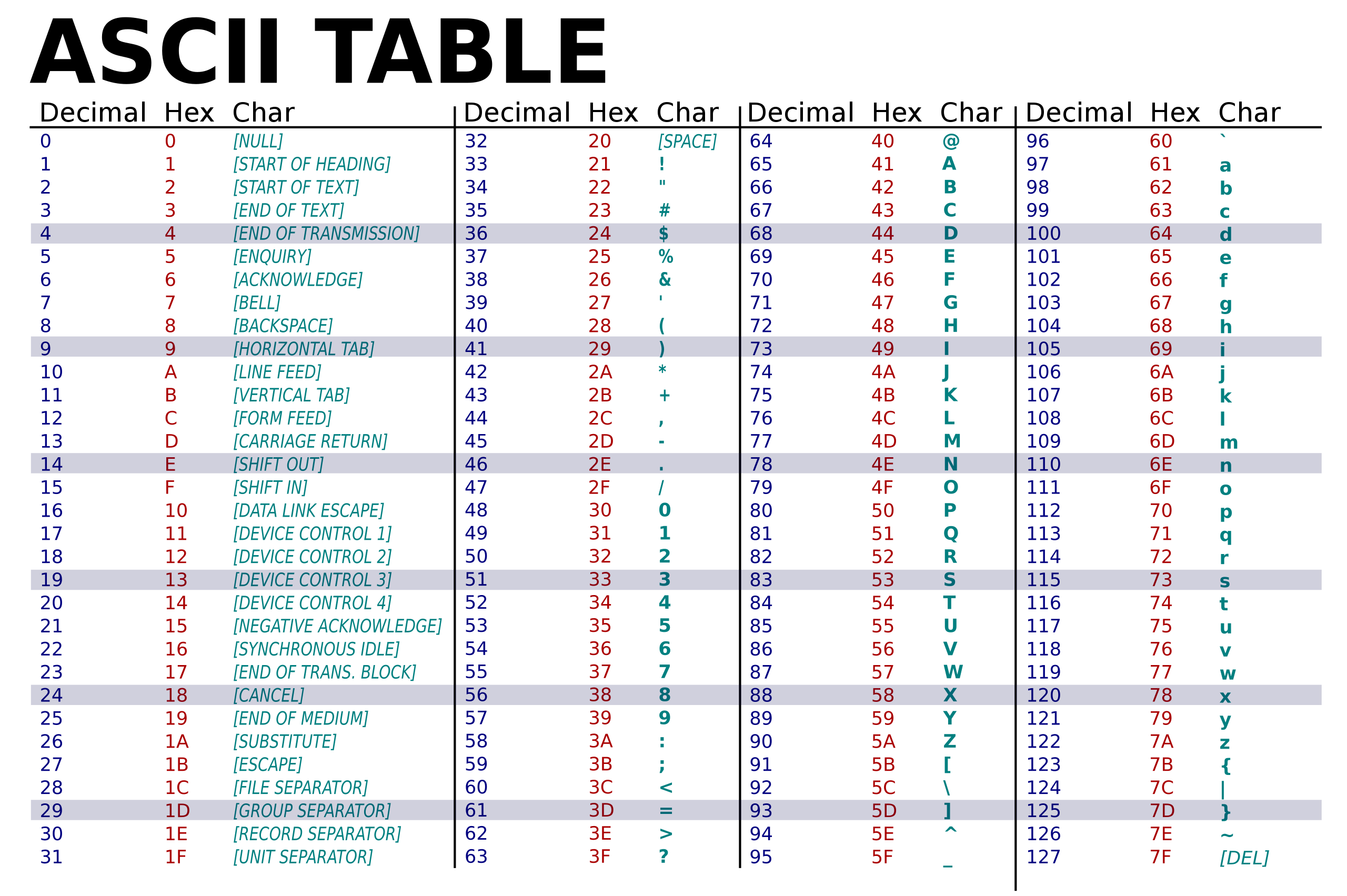


Figure 2.14: ASCII Alphabet

Let’s take an example of a message of ‘ABC’, the ASCII value for it is 656667, this is going to be our **m** value. The encryption process into a cipher can be done by using the below formula.

With **c** as our ciphertext, this is the only information that hackers can get since the private key is kept secret.

**Step 5**. The receiver whom hold the private key will decrypt the ciphertext **c** by the following formula.

After this, we can receive the message value **m** and revert it back to text by using the ASCII table. Please note that since we do not want to lose our data by using modulo operation.

2.3.2 Example

For instance, a person A want to send a message of ‘T’ to person B. The whole process is going to be as follows.

**Step 1**. . We are choosing small number just to keep everything easy to follow. Therefore .

**Step 2**. We then calculate , and randomly choose **.**

**Step 3**. We calculate the decryption exponent **e** as follows:

**Step 4**. The decimal value of message ‘T’ in our ASCII alphabet is 84. Therefore, we calculate the ciphertext **c** as follows:

**Step 5**. The receiver B can decrypt the message from the ciphertext by using the formula above

We can see that we have obtain our message by converting the ASCII decimal value to a message.

2.3.3 Implementation

First, I would like to present the RSA cryptosystem without using library. The function is supported by various supportive function in the previous chapter and in this chapter as well.

def isCoPrime(a, b):  
 g, x, y = extendedEuclidGCD(a, b)  
 if g == 1:  
 return True  
 return False

Figure 2.15: isCoPrime Function

This function is used to check whether 2 numbers **a** and **b** are co – prime or not. We also use the extendedEuclidGCD and the inverseModulo function in Chapter 1. Now, let’s go to the main implementation without using the rsa library.

def rsa(message, length = 1024):  
 # step 1  
 p = generateBigPrimeNumber(length)  
 q = generateBigPrimeNumber(length)  
  
 n = p\*q  
  
 # step 2  
  
 euler = (p-1) \* (q-1)  
 e = random.randint(2, euler-1)  
 while True:  
 if isCoPrime(e, euler):  
 break  
 e = random.randint(2, euler - 1)  
 # step 3  
 d = inverseModulo(e, euler)  
  
 # step 4  
 m = ""  
 width = []  
 for letter in message:  
 value = ord(letter)  
 width.append(len(str(value)))  
 m = m + str(value)  
 print("The original message is", m)  
  
 cipher = pow(int(m), e, n)  
 print("The encrypt message is", cipher)  
  
 # step 5  
  
 decryptM = pow(cipher, d, n)  
 stringDecrypt = str(decryptM)  
 firstPos = 0  
 lastPos = 0  
 returnMsg = ""  
 for i in range(0, len(width)):  
 if lastPos == 0:  
 firstPos = lastPos  
 else:  
 firstPos = lastPos + 1  
 lastPos = firstPos + width[i] - 1  
 asciiCode = ""  
 for j in range(firstPos, lastPos + 1):  
 asciiCode = asciiCode + stringDecrypt[j]  
 asciiCode = int(asciiCode)  
 returnMsg = returnMsg + chr(asciiCode)  
 print("The decrypt message is:", returnMsg)

Figure 2.16: rsa Function

The rsa function does every step in the algorithm section in the RSA Cryptosystem sub – section in Chapter 2. In summary, the function is going to choose 2 prime numbers **p** and **q** and calculate , it then continue to calculate the Euler’s totient function between **p** and **q** and choose a random **e** number as co – prime to the Euler’s function result. After that, it going to calculate the **d** value and calculate the ciphertext **c**, the **width** list is used to store the length of every ASCII character. For example, ‘A’ has its value as 65 and so its width is 2.

Lastly, the function is going to decrypt the message from the ciphertext **c** and store it in a value **mDecrypt**, the **lastPos** and **firstPos** variables is used to determine the first and last index of each ASCII character value in the **mDecrypt** variable, because this variable holds all of the ASCII value. For example, if I have a width list of and the **mDecrypt** variable has its value as 110101, this means that the first ASCII character value is 110 and the second ASCII character value is 101. The function is going to return the message after the decryption process.

You can run the program by entering a message and choosing the key size.

msg = input("Enter you message: ")  
rsa(msg, 1024)

Figure 2.17: Message Prompt

Here is a demo of a message of “Hello World”.

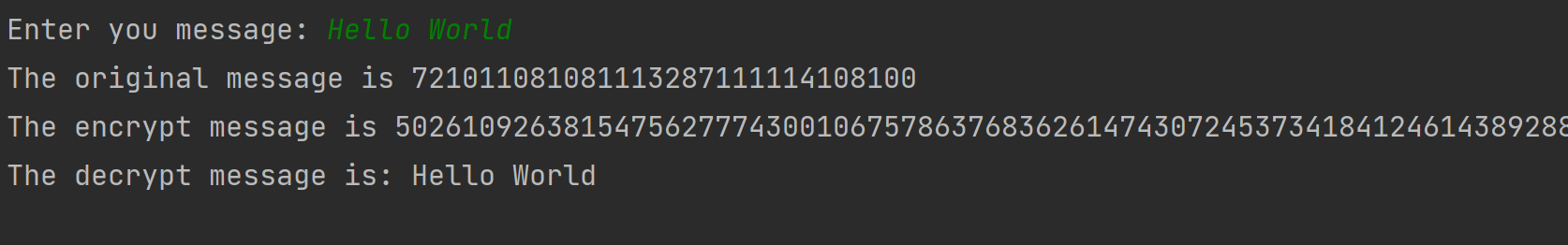


Figure 2.18: Demo of ‘Hello World’

Since the key size is 1024 – bit, the ciphertext has a very big value but I can not capture all of the number since it has gone off screen.

The implementation of the RSA cryptosystem could also be achieved by the use of the ‘rsa’ library. The process of coding is much simplier and easier to understand and implement. Here are the implementation of the RSA cryptosystem using the ‘rsa’ library.

import rsa as RSA  
  
pubKey, priKey = RSA.newkeys(1024)  
msg = bytes(msg, "ASCII")  
print("Original message",msg)  
encryptMsg = RSA.encrypt(msg, pubKey)  
print("Encrypt msg:", encryptMsg)  
decryptMsg = RSA.decrypt(encryptMsg, priKey)  
print("Decrypt msg:", decryptMsg)

Figure 2.19: ‘rsa’ Library Implementation

In this implementation, the rsa will generate its keys from the newkeys function, which has an input of the bit – size. Next, we need to convert the message string in the previous implementation into bytes with the format of ASCII. This is needed because in the encrypt function, the message is required to be a bytes type variable, the function also require a public key to encrypt the message.

Finally, we decrypt the encrypted message by using the decrypt function, provided the function with the encrypted message, private key and it will return the original message.

Here are the demonstration of this implementation along side with the previous non – library approach.

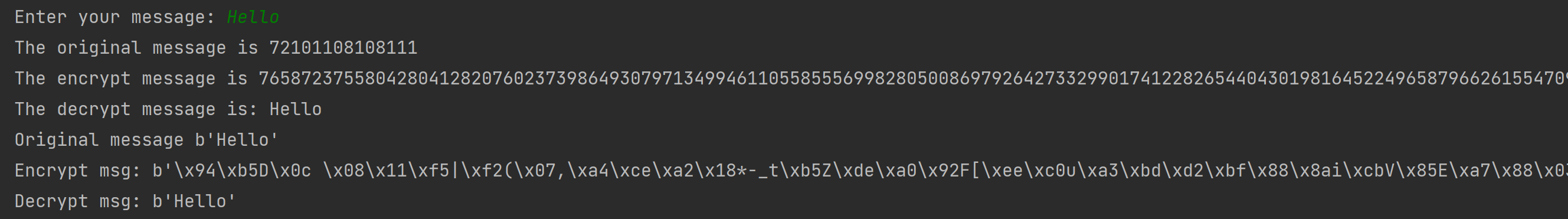


Figure 2.20: Two implementations

As we can see, the encryption format is different since our non – library implementation is just only a demonstrating of the algorithm, the actual implementation would have some additional steps but the general algorithm remains the same.

2.3.4 Efficiency and security

From the implementation and analysis we can make conclusion that the RSA algorithm are relatively slow but the security is what makes still a good option. We say the algorithm “slow” because it involves modular exponentation, which requires a large computer resource when the number grows larger. Specifically, the numbers involves in the RSA algorithm are huge, so any computation and operation becomes very time – consuming. The process of choosing prime numbers is also a consideration since we know that this algorithm requires two very large prime numbers, so the process of choosing good prime numbers is also computationally expensive in terms of time and resources.

The security of the RSA algorithm mainly comes from the difficulty of finding the factor of the product between two prime numbers. The fastest possible factoring algorithm we currently know is the General Number Field Sieve, which has the time complexity of

We can calculate the time it take by subtituting our bit – size into the formula and have an exponential time complexity, and as we have known, exponential is the slowest time complexity. Therefore, if we have a sufficient key size and two good prime numbers, the algorithm is relatively safe.

2.3.5 Threats and Limitations

As we have discussed before, the security of the RSA algorithm mainly comes from the difficulty of factoring of a product of two prime numbers. Therefore, any bad choices of prime number could lead to hackers easily breaking down the private key and use it to their need. The main reason for this threat could trace back to the insufficient choice of key – size, too small of it the system will make bad choices.

Quantum computer is also one major threat to the cryptosystem. In summary, quantum computer is different from the classical computer we use nowadays, instead of using bit to represent state, it uses a unit called qubit. Qubit is can represent the multiple states at once instead of just either 0 or 1 like bit. Therefore, making the computational process massively quicker than the normal bit. Even if we use brute force, quantum computer could find the factors of the product between two large prime easily and efficiently.

The RSA is also vulnerable to channel acttacks and chosen ciphertext attack. These types of attack could retreive the private key and use it to decrypt the message.

2.3.6 Conclusion and Recommendation

In conclusion, we have successfully implemented the RSA cryptography algorithm in Python. Please beware that any implementations in this essay is the simplest form of the RSA cryptography and many other different factors should be taken into account.

We can make one such improvement in the key generation phase. This will enhance the security of the cryptosystem as well as increase the time to attack the system. Instead of just calculating , we could calculate it as . Whereas **n** is still unchanged, with **r** and **s** are 2 prime numbers as well.

The main different is that instead of using **N** as the modulus, we are still going to use **n**. This will make the attacker even have the information about **n**, can not find all the basis for **N.** Here are the algorithm

**Step 1**. Choose 4 large prime numbers **p**, **q**, **r**, and **s**.

**Step 2**. Calculate and . Then, compute .

**Step 3**. Calculate the Euler’s totient function of **n** and **m**. and . Therefore, we have

**Step 4**. Find the **e1** number that and . We also find **e2** such that and .

**Step 5.** Calculate .

**Step 6**. Find **E** such that and . As we can see, **E** is going to be our public exponent.

**Step 7**. Calculate the private exponent **D** such that .

Therefore, the public key is **(E, n)** and the private key is **(D, n)**.

The ciphertext **C** will be encrypted as follows with **M** as the plain text message.

And the decryption process will be similar to the decryption step in the original RSA algorithm with **P** as the decrypted plain text message.

The main purpose of this process is to increase the attack time on the system as we need to find 4 large prime numbers. The drawback to this approach is the time to generate keys, encryption and decryption is going to take longer since we are working with 4 prime numbers.

REFERENCES

**English**

1. Xing Zhou (2017), *Number Theory – Modular Arithmetic: Math for Gifted Students*, CreateSpace Independent Publishing Platform, United States.
2. Steve Burnett, Stephen Paine (2001), *RSA Security's Official Guide to Cryptography 1st Edition*, McGraw-Hill Osborne Media.
3. Dr. Mohamed Omar (2017), *Number Theory Toward RSA Cryptography: in 10 Undergraduate Lectures (Discrete Mathematics) 1st Edition,* CreateSpace Independent Publishing Platform, United States.
4. Engr. Shaheen Saad Al-Kaabi and Dr. Samir Brahim Belhaouari (2019), Methods Toward Enhancing RSA Algorithm: A Survey, College of Science and Engineering, Hamad Bin Khalifa University (HBKU), Doha, Qatar.
5. George E. Andrews (1994), *Number Theory (Dover Books on Mathematics)*, Dover Publications.

SELF – EVALUATION

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | | Scale | Self – evaluation | Reason |
| Task 1 | Theoretical Research | 1 | 1 | Conduct full research with detailed explanation |
| Implementation | 2 | 2 | Complete Python code with explaination for each block of code, without any run – time error |
| Test | 1 | 1 | Enough test for all cases with verification |
| Task 2 | Theoretical Research | 1 | 1 | Conduct full research with detailed explanation |
| Implementation | 2 | 1 | Complete Python code with explaination for each block of code, the code will take longer for bigger messages have about 500 words or above |
| Test | 1 | 1 | Enough test for all cases with verification |
|  | Analysis | 0.5 | 0.5 | Given enough detailed explanation and comments |
|  | Discussion | 0.5 | 0.25 | Not enough details, no comments |
|  | Recommendation | 0.5 | 0.5 | Given enough explaination and algorithm for improvements |
|  | References | 0.5 | 0.5 | References given in the right format |
|  | **Total** | 10 | 8.75 |  |