**Foundations of computer science coursework**

**RSA algorithm**

Youssef Abdelmaguid Afifi

202200167

# Table of contents

1. Introduction to RSA algorithm
2. Program 1: factoring the modulus using extended GCD
   1. Problem decomposition
   2. Code function
   3. Pseudocode + complexity
   4. N runs + Runtime chart
3. Program 2: Brute force method
   1. Problem decomposition
   2. Code function
   3. Pseudocode + complexity
   4. N runs + Runtime chart
4. Test case generation: values for N and e, for C and M
5. Conclusion

## Introduction

The RSA (Rivest–Shamir–Adleman) algorithm is a widely used asymmetric cryptosystem designed for secure data transmission. An asymmetric encryption method consists of a public key used to encrypt the original message, while a private key is used to decrypt the message back to its original form. A common use of RSA in real life is the RSA SecureID. It works when a client has the SecureID token, they receive a private key on it which allows only them to access information or do certain tasks. That way information is safe as it is being secured by 2 factor authentication.

The goal of python programs completed is to crack an encrypted message while being provided with a public key and a number to decrypt. This report shows the difference between a mathematical approach to the problem as opposed to a brute force method of decrypting the message, respectively showing the difference in the testing results.

## Program 1

### Problem decomposition

In this program, the objective was to factor the modulus N (product of two prime number) and calculating the private exponent. The most efficient way to do this task was to use the extended Euclidean algorithm. The Extended Euclidean Algorithm is an extension of the Euclidean Algorithm that computes the greatest common divisor (GCD) of integers a and b (in this case e and phi\_n) . In addition to computing GCD, Extended Euclidean Algorithm also finds integers s and t such that ax+by=gcd(a,b) . The first function of the code is extended\_gcd. The extended Euclidean algorithm is crucial in finding the private key. it helps find the modular multiplicative inverse of (e). The function is later called and is applied on the public exponent and the phi\_n “(p-1) \*(q-1)”. The coefficients from the extended Euclidean algorithm allow us to express (1 = e \c.x + \phi(n) \c.y), where (x) and (y) are integers.

### Code function

1. extended\_gcd function

* x,y(coefficients) are set to 1 and 0
* while loop that stops when b is equal to 0 starts
* Update the coefficients x and y according to the formula for the extended GCD
* After the loop if x is negative, it is adjusted to ensure it is positive
* Finally, the function returns the GCD (a) and the coefficients x and y.

1. mod\_inverse function

* the extended\_gcd function is called to compute the greatest common divisor (GCD) of e and n, as well as the coefficients x and y such that ex + ny = gcd(e, n). The returned values are stored in variables gcd, x, and y.
* line to return value error if gcd does not equal to 1
* modular inverse is returned

1. factorization function

* line that starts a loop that iterates through integers from 2 to the square root of n
* range(2, int(math.sqrt(n)) + 1) generates integers from 2 to the square root of n plus 1. This range ensures efficient factorization
* line that checks if the current integer i evenly divides n. If i divides n evenly, then i is a prime factor of n. n is then divided by p to calculate q.
* p and q are returned if the prime factors are found.

Outside of the functions:

* factorization is performed on n to get p and q
* Euler’s totient is calculated using (q - 1) \* (p - 1)
* private key (d) is calculated using the public key and phi\_n
* results are printed

### pseudocode and complexity

Initialize factorization\_count to 0

Initialize extended\_gcd\_count to 0

Define extended\_gcd function taking parameters a and b

Increment extended\_gcd\_count by 1

Set x to 1

Set y to 0

While b is not 0

Set q to the integer division of a by b

Set a to b

Set b to the remainder of the division of a by b

Set x to y

Set y to x minus (q times y)

While x is less than 0

Add phi\_n to x

If y is less than 0

Add a to y

Return gcd, x mod n, y

Define mod\_inverse function taking parameters e and n

Find gcd, x, y using extended\_gcd function with parameters e and n

If gcd is not 1

Raise ValueError with message "Modular inverse does not exist"

Return x mod n

Define factorization function taking parameter n

For each integer i from 2 to the square root of n

Increment factorization\_count by 1

If n is divisible by i

Set p to i

Set q to the integer division of n by i

Return p, q

Return None, 0

Input n from the user

Input e from the user

Set start\_time to the current time in milliseconds

Perform factorization of n and assign results to p and q

If both p and q are not None

Calculate phi\_n as (q - 1) times (p - 1)

Calculate d using mod\_inverse with parameters e and phi\_n

Output p, q

Output d

Output the runtime as the current time minus start\_time in milliseconds

Output the total number of runs as the sum of factorization\_count and extended\_gcd\_count

### Time complexity

1. Factorization Function (factorization(n)):

The function iterates from 2 to the square root of n.

Within each iteration, it performs a modulus operation.

Time Complexity: O(√n)

1. Extended GCD Function (extended\_gcd(a, b)):

The extended Euclidean algorithm typically runs in logarithmic time in the smaller of the two numbers.

Time Complexity: O(log(min(a, b)))

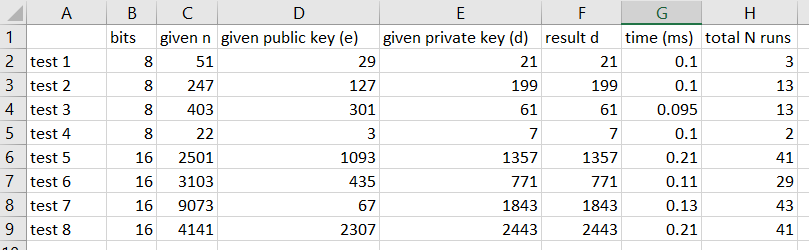
1. Modular Inverse Function (mod\_inverse(e, n)):

Calls the extended GCD function.

Time Complexity: O(√n)

Overall, the most significant factor in determining the time complexity is the factorization function since it runs in O(√n). The extended GCD and modular inverse functions have a logarithmic time complexity.

### N runs and runtime chart



## Program 2

## Problem decomposition

Unlike program 1 which uses an advanced mathematical process to factorize n, this program uses brute force to keep generating prime numbers until p\*q are equal to n. the code features a prime\_check function as well as a generate\_prime function. These functions are responsible for finding p and q before they are implemented into a formula that finds the private key (e \* d) % phi\_n = 1 .

### Code function

1. is\_prime function

* checks if value is prime. It iterates from 2 up to the square root of num checking for any factors. If it finds a factor, it returns False; otherwise, it returns True.

1. generate\_prime function

* generates a prime number less than or equal to N
* It uses the is\_prime() function to check if a randomly generated number is prime
* repeats this process until it finds a prime number
* returns the prime number

1. brute force code

* p and q are both set to 0
* while loop that keeps generating a prime value for p until p\*q is equal to N
* once p is found, N is divided by p to find q

1. find\_private\_key

* This function calculates the private key d given the public key e and Euler's totient function of N, denoted as phi\_n
* It iterates through possible values of d from 2 to phi\_n, checking if (e \* d) % phi\_n == 1
* once it finds such a d, it returns it.

Initialize loop\_counter\_generate\_prime and loop\_counter\_find\_private\_key to 0

Define a function is\_prime(num) to check if a number is prime:

If num is less than 2, return False

Iterate through numbers from 2 to the square root of num:

If num is divisible evenly by any number in the range, return False

If num passes all checks, return True

Define a function generate\_prime(N) to generate a prime number:

Initialize loop\_counter\_generate\_prime to 0

Loop infinitely:

Increment loop\_counter\_generate\_prime by 1

Generate a random prime\_option between 2 and N

If prime\_option is prime, return it

Input the value of N (product of two primes) from the user

Input the value of e (public key) from the user

Record the starting time

Initialize p and q to 0

Repeat until p\*q equals N:

Generate a prime number p using the generate\_prime function

Calculate q = N // p

Calculate φ(n) = (p - 1) \* (q - 1)

Print the values of p and q

Define a function find\_private\_key(e, phi\_n) to find the private key d:

Initialize loop\_counter\_find\_private\_key to 0

Iterate over numbers from 2 to phi\_n:

Increment loop\_counter\_find\_private\_key by 1

If (e \* d) % phi\_n equals 1, return d

Find the private key d using the find\_private\_key function

Record the ending time

Calculate the execution time

Print the execution time in milliseconds

Print the total loop iterations in generate\_prime and find\_private\_key functions

### Time complexity

Generating Primes:

The generate\_prime function uses a probabilistic method to generate prime numbers. The time complexity of checking whether a number is prime within a given range is O(sqrt(n)). However, since this function is called repeatedly until a prime is found, its overall time complexity can be considered as O(n), where n is the upper limit for prime generation.

Finding Private Key:

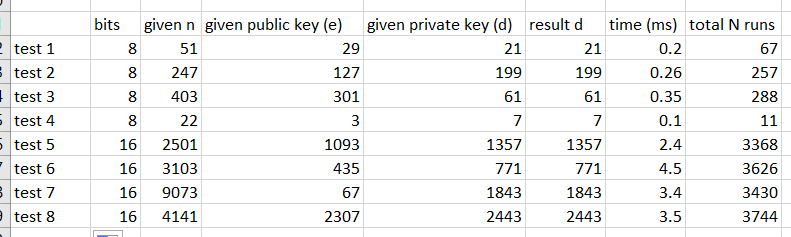
The find\_private\_key function iterates through numbers from 2 to phi\_n, which has a time complexity of O(phi\_n) or O(n), where n is the value of phi\_n.

Overall:

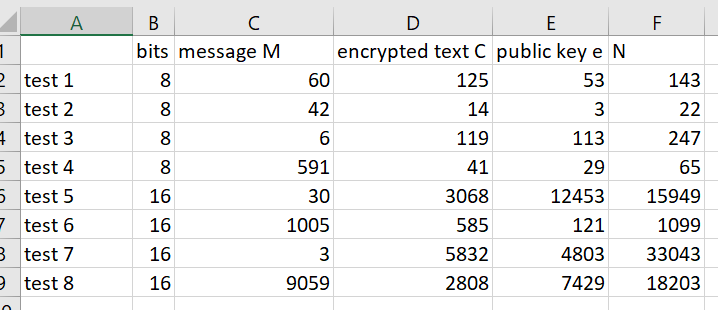
The main loop that generates p and q continues until p\*q equals N. The worst-case scenario is when p and q are close to the square root of N. Hence, the time complexity of this part can be considered O(sqrt(N)).

The calculations for phi\_n, printing values, and other constant time operations do not significantly affect the overall time complexity.

### N runs + runtime



## Test case generation



## Conclusion

When comparing the efficiency of cryptographic algorithms like RSA, particularly in terms of decrypting messages, the choice of algorithm can significantly impact runtime performance. Through the process of creating two RSA decryption programs—one utilizing the extended greatest common divisor (gcd) function and the other relying solely on brute force—it becomes evident that algorithmic efficiency plays a pivotal role, especially with larger input sizes. During testing phases using 8 and 16-bit numbers, the runtime difference between the two approaches may not be readily apparent to most observers. However, as the size of the numbers involved increases—say, to 32, 64, or even 128 bits—the runtime gap becomes more pronounced. This widening gap can be attributed to the increased average number of iterations required by the brute force method compared to the more optimized extended gcd approach. It is crucial to recognize that the efficiency gained from employing optimized algorithms, such as the extended gcd function, extends beyond just shorter execution times. These algorithms offer scalability, allowing them to handle larger input sizes without experiencing a significant increase in execution time. This scalability is particularly vital for cryptographic algorithms like RSA, which are commonly used to secure sensitive communication and data transfers over networks. Therefore, when implementing cryptographic algorithms, developers must prioritize efficiency and carefully select the most appropriate algorithms for the task at hand. By doing so, they can ensure not only robust security but also efficient and scalable performance, even when dealing with larger input sizes. This emphasis on efficiency becomes increasingly crucial as the demands for secure communication and data protection continue to grow in today's interconnected digital landscape.