

Mathematics Extended Essay

How do the mathematics of the RSA encryption help us determine the time required to decrypt an encryption?

In this essay, I will be using the mathematics of RSA encryption to determine a formula used to approximate the amount of time taken to decrypt a code using the principles of RSA depending on the amount of digits of a public key (N).

Session: May 2023

Candidate Number: jgd623

Word Count: 3981

Table of Contents

Introduction to Encryption	3
RSA Mathematics	
Fermat's Little Theorem	4
Euler's Totient Theorem	5
Euler's Theorem	5
Steps to Encrypt RSA	6
RSA Algorithm using Python IDLE	7
Decryption of RSA	8
Modelling the data	11
Evaluation of the model	13
Conclusion	13
Bibliography	14
Appendices	15
Appendix A	15
Appendix B	17
Appendix C	18

Introduction to Encryption

Can secrecy be attained by codemakers? Or will codebreakers be able to produce a computer capable of deciphering all messages? For thousands of years, generals, kings and queens have counted on safe communication systems in order to send messages abroad. At the same time, they have all been aware of the consequences of their messages falling into the wrong hands, revealing precious secrets to rival nations. It was this threat that motivated the creation of coding mechanisms to hide their messages from interceptors.

Cryptography is the art of secret communication. The aim of cryptography is to hide a message's meaning whilst being public. This process is known as encryption. Powerful encryption is required in order to guarantee safe transaction of information within the evolving world of the internet. Encryption methods are constantly evolving, since codes are always under attack. Contrastingly, codebreakers are forced to develop their decryption techniques in order to decipher the new, stronger codes.

In order to render a message unintelligible, it must be translated to symbols or words using a protocol that has been agreed upon by the sender and the receiver. As a result, the recipient can use the protocol's inverse to make the message readable once they have received it. Hence, even if a third party intercepts the message, they won't be able to understand it unless they are aware of the protocol.

There are two types of encryption. Transposition and substitution. Substitution is the coding technique used in RSA and Caesar Cipher (Julius Caesar most frequent use of secret writing). For example, the next letter in the alphabet would be used to replace each letter in a word. A would be replaced by B, B by C, C by D and so on. Thus, "Attack at dawn" would become "Buubdl bu ebxo". Each letter in an alphabet is substituted with a letter that is x amount places further along the alphabet. To code or decode a message, a key is used. A key is similar to the principle of inverse functions in algebra. For example, if x is a number in the alphabet, where $1 = a$, $2 = b$, $3 = c$ and so on, the key to the code the message is a function $f(x)$, and y is the coded output.

If $f(x) = x + 3 \bmod(26)$, then $f^{-1}(x) = x - 3 \bmod(26)$. The function $\bmod(26)$ returns the remainder of a division operation, so it loops through the whole alphabet (because no value can be bigger than 26, since the alphabet has 26 characters). The receiver **knows** the key, which is the inverse of the function, and subsequently only has to input each numerical value of each letter in the alphabet to decipher their real value. This was the stepping stone to encryption, and is important to understand where it comes from to then apply these concepts in modern day encryption.

In the past, users wanted their data to be encrypted to prevent data theft by interceptors. The same scenario still applies, wherein the two users communicating with one another must agree on a key to encrypt and decrypt their messages. But how was the key decided? in person? What if they were hundreds of miles apart? This problem is known as the key distribution problem: Before two people can share data, they must already share a key, meaning that key compromises everything.

Diffie and Hellman, two American computer scientists, managed to solve the problem by coming up with a simple concept. To illustrate this in a realistic simulation, we can have Alice and Bob. Alice wants to send an encrypted message to Bob. Alice does this by locating Bob's public keys (n and e), two values that are accessible to everybody online, and encoding her message using a formula that uses those two values. The formula used to encode the message is called an asymmetric key, meaning that a value can be entered, but to find the inverse in order to obtain the original value of x is near impossible. So, how is Bob supposed to understand the message if you code something and can't decode it? The simple answer is a private key. Bob has a special piece of information (d) that helps him decode the encrypted message with the asymmetric key in easy steps. Nevertheless, Diffie and Hellman only developed a theoretical model. The asymmetric key was later mathematically developed by computer scientists Rivest, Shamir and Adleman, known as RSA encryption.¹

RSA consists of two prime numbers (p and q) that, when multiplied, make the number n . Using the example above, let's say Alice wants to send Bob a message. Alice finds Bob's public encryption, (n and e) that are variables in the general one-way function (the public encryption formula). So, she inserts Bob's public keys n and e into the general one-way encryption formula and sends it to him. Thus, since d consists of values p and q , she is able to decode the one-way function using her private key (d).

After researching, I came to the conclusion that symmetric keys work well when people can meet in advance to exchange keys, but on the internet, people can't do that since they may be far away. So, asymmetric keys are the solution. Knowing the background of RSA encryption helps readers understand such a complex method of securing data in modern-day. Below I will explain the theory of the mathematics behind the RSA encryption and decryption system, and later on do a step-by-step explanation of how it works.

RSA Mathematics

Fermat's Little Theorem:

In simple terms, Fermat's little theorem states that if N is a prime, and a is any integer, then $a^n - a$ is divisible without remainder by n .

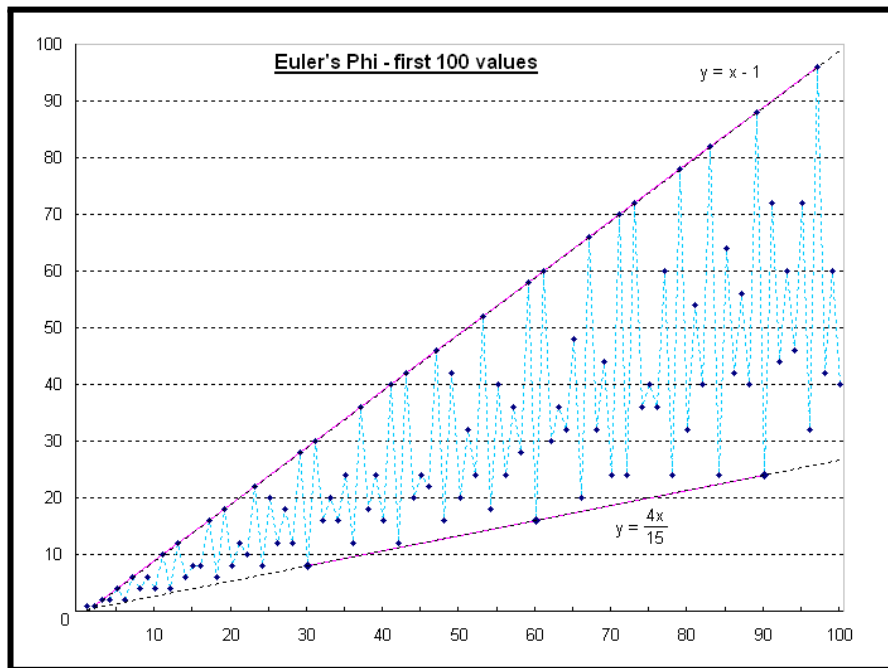
For example, if $n = 7$ and $a = 10$, then $10^7 - 10 = 9999990$ is divisible without remainder by n , so it can be divided by 7. Hence, $\frac{9999990}{7} = 1428570$, which is a whole number. Consequently, we can express this as $a^n - a = 0(\text{mod } n)$, where mod stands for modular which is a mathematical way of representing the remainder of an operation, and in this case would be 0. Following the modular arithmetic definition, if $x = a(\text{mod } n)$, $x = a$ is divisible by n , then $a^{n-1} - 1 = 0 (\text{mod } n)$, where $a \neq n$.

¹ Singh, S. (2000). *The code book: The secret history of codes and code-breaking cryptography*. Fourth Estate.

Euler's Totient theorem:

The phi function (ϕ). Given a number n , $\phi(n)$ is equal to the number of numbers that are less than or equal to n and do not share a common factor with n . In this case, if we want to find $\phi(8)$, we can identify that out of 1, 2, 3, 4, 5, 6, and 7, only 1, 3, 5 and 7 do not share a common factor, so $\phi(8)$ would be 4.

One important feature of Phi is that, as you can see in *graph 1*, it has a pattern and forms the line $y = x - 1$. These values are all the prime numbers. This is because prime numbers don't have a factor greater than 1, so the Phi of any prime number is simply $p - 1$. Therefore, it forms a straight line equation $y = x - 1$.



(Graph 1)

Moreover, ϕ has a special feature that follows the following pattern: $\phi(a \times b) = \phi(a) \times \phi(b)$ (only if a, b have no common factors). Thus, if (as explained before in the RSA section) $n = p_1 \times p_2$, then $\phi(n) = \phi(p_1) \times \phi(p_2)$, and if they are prime numbers, we know that $\phi(p_1) = p_1 - 1$ and $\phi(p_2) = p_2 - 1$, so $\phi(n) = (p_1 - 1) \times (p_2 - 1)$.

Euler's Theorem

Euler's totient theorem states that $m^{\phi(n)} \equiv 1 \pmod{n}$. This theorem only uses two integers, m and n , where n is the product of two prime numbers and m is the encrypted message. If we think about this m being a message, we want to find $m^{ed} \equiv m \pmod{n}$.

We know that someone can decrypt a message using the general formula $m^e \pmod{n} = c$, where c is the decrypted message. The decrypted message is $c^d \pmod{n} = m$, where d is the private key. So, combining both formulas, we get that $(m^e \pmod{n})^d \pmod{n} = m$ to decrypt the message.

By using modular arithmetic properties, we can raise both sides of $m^{\phi(n)} \equiv 1 \pmod{n}$ by k to get $m^{k\phi(n)} \equiv 1^k \pmod{n} \equiv 1$, since $1^k = 1$ for all values of k .

Now we multiply both sides by m to have the equation equal to m , so $m^{\phi(n) \times k} m = m \pmod{n}$, which equals $m^{(\phi(n) \times k) + 1} = m \pmod{n}$.

Thus, by comparing the congruency between $m^{(\phi(n)k+1)} \equiv m \pmod{n}$ and $m^{ed} \equiv m \pmod{n}$, we know that $m^{(\phi(n)k+1)} \equiv m^{ed}$, so $ed = k \times \phi(n) + 1$, meaning that d (the private decryption key) = $d = \frac{k \times \phi(n) + 1}{e}$.

These two theorems help me understand the mathematics of the formulas used in RSA encryption, as they closely flow together to work out the private key encryption. Moreover, it will make it easier for me to program the algorithm, as I will have a deeper understanding of why the private key is the way it is.

Subsequently, I will show the steps for RSA encryption, where I used the following equation: Encryption : $m^e \pmod{n} = c$ and Decryption : $c^d \pmod{n} = m$, where m is the message.

Steps to encrypt RSA

(1) Bob wants to send Alice a message. Alice picks two numbers, p and q . She uses $p = 11$ and $q = 13$. These two numbers must be kept secret.

(2) Alice gets a number n (that is made public), which is the product of p and q . $n = 143$. e is the other public key, which has to be a value between 2 and n , and it has to share no common factor with n and $\phi(n)$. She picks $e = 7$ (where $(p - 1) \times (q - 1)$ and e are relatively prime, so they only have 1 common factor).

(3) For Bob to encrypt his message, he must convert his message into a number. For example, he can translate the word to ASCII (binary), and use the binary digits as decimal numbers. He uses m , and is encrypted using the formula $c = m^e \pmod{n}$, where e and n are public. In our example, $m = 64$, and we know $n = 143$ and $e = 7$, so $c = 64^7 \pmod{143} = 103$. This value is then sent to Alice.

(4) As explained before, exponentials and modular arithmetic is a one-way function, meaning it is hard to recover M by knowing c , e and n , so any interceptor would not be able to decrypt the message.

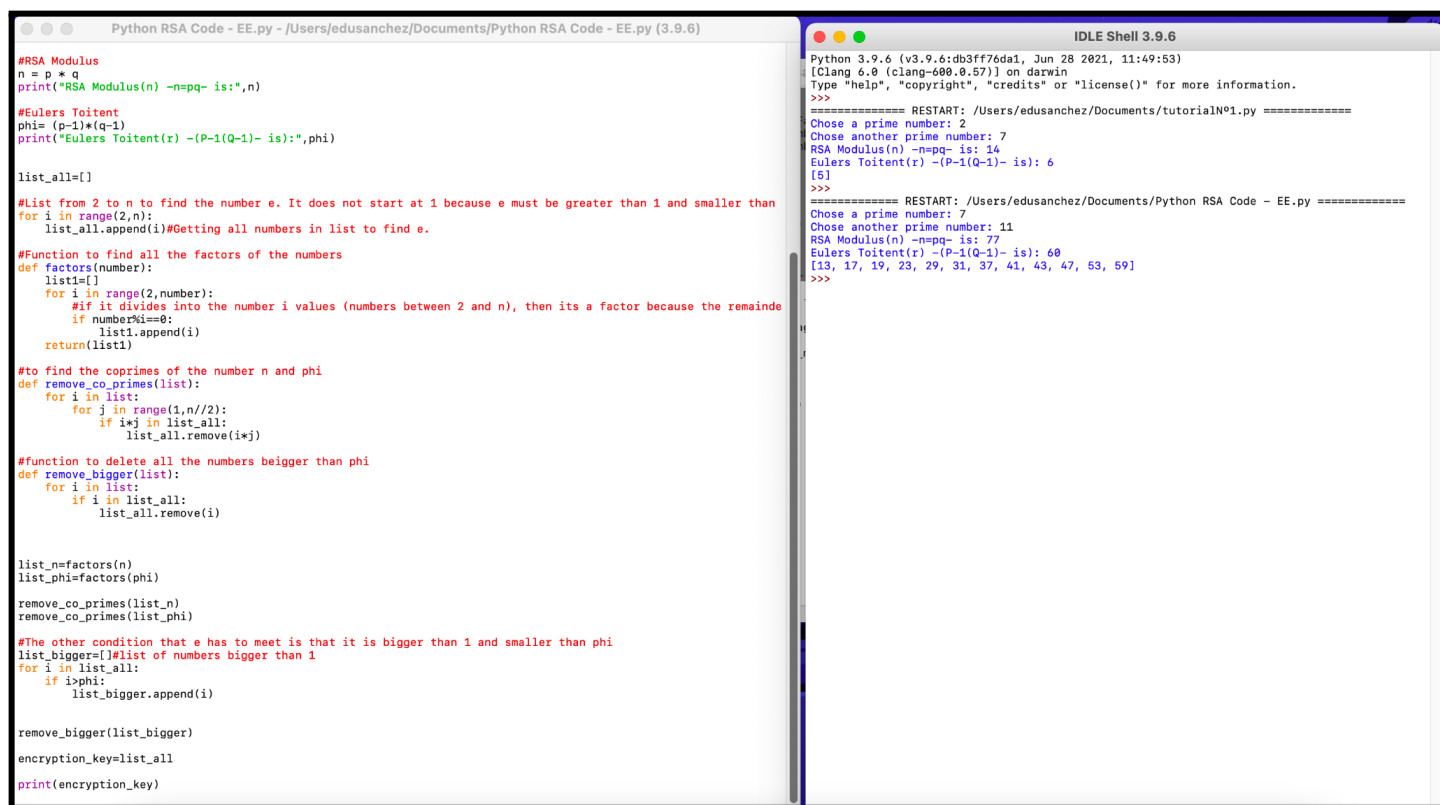
(5) However, Alice knows a special piece of data (p and q). Thus, by formulating a number d , the decryption key, Alice can decrypt Bob's message. Number d is known as her private key. To calculate d , she does $d = \frac{k \times (p-1) \times (q-1) + 1}{e}$, where k is a number that is the multiple of $(p - 1) \times (q - 1) = 120$ (plus 1). So, 121, 241, 361, 481, 601, 721. I found that $(721 \div 7) = 103$, so k is 6, because it is the 6th multiple of 120 + 1 that is divisible by 7. Now, we can calculate $d = \frac{6(10 \times 12) + 1}{7} = 103$.

(8) To decrypt the message, Alice uses the formula $c^d \pmod{n} = m$, so $103^{103} \pmod{143} = 64$, which is the original number m that Bob had encrypted.

The interceptor can only decrypt the message if they know d , which only has $\phi(n)$ missing, and in order to do so, they have to go through an immensely hard factorization problem if the numbers are large enough, as RSA encryption uses prime numbers with about 617 digits. The interceptor will know the value of n , but they have to find two prime numbers that, when multiplied, create the number n , to find d (private key). This is the beauty of RSA, because it is near impossible and would take computers centuries to find the two prime numbers that make up n . However, it is important to state that there are multiple ways to find the two prime numbers, which will be discussed later.

Now, I will continue my investigation by coding the encryption and decryption process using Python, and identify the amount of time it takes to decrypt different encrypted messages with different prime number sizes, which will then be my data to approximate the biggest n value (in digits) that can't be solved yet.

RSA algorithm using Python IDLE

The image shows a screenshot of the Python IDLE environment. On the left, a window titled 'Python RSA Code - EE.py' displays the RSA algorithm code. The code includes functions for finding factors, removing coprimes, and deleting numbers bigger than phi. It then calculates the encryption key. On the right, the 'IDLE Shell 3.9.6' window shows the execution of the code. It prompts the user to choose a prime number (2), then another (7), and displays the resulting RSA Modulus (14) and Euler's Totient (6).

The first part of the algorithm serves to check if p and q inputs are prime. It gets the number (A), and checks if it equals 2, which is the smallest even prime number, it is a prime. Then, if the value is greater than 2 and any number between 2 and n , has no remainders, it is not prime since prime numbers can't be divided by whole integers. So, if n can't be divided by any numbers between 2 and n , they are prime numbers.

The next part of the code is designed to find the value of e (the encryption key), where e can be any number $1 < e < \phi(n)$ that is coprime to $\phi(n)$, meaning e has to share no common factor with $\phi(n)$.

To do this, I first created a list from 2 to n to find the number e . It does not start at 1 because e must be greater than 1 and smaller than $\phi(n)$. Then, I divide the number n by all numbers in the list, since they can't share any common factors, and delete them from the list. Then, to find the coprimes of e and $\phi(n)$, if $(e \times x)$, where x is any number on the list, equals n or $\phi(n)$, then the value will be deleted from the list. Nevertheless, since we can sometimes have multiple values, I only got the first value of the list.

Next, in order to find k , (the multiplier that gives d a whole number when divided by e), I created a code that runs the code with $k = 1$. The loop finds the remainder of the formula for decryption ($d = \frac{k \times \phi(n) + 1}{e}$), using $\text{mod}(e)$ and if the remainder is not 0, then the value of k is not correct since d has to be a whole number. Then, 1 is added to k , and it iterates until the value of decryption is 0.

Finally, with the values of d and e sorted, I can use the formulas described above to calculate the encrypted code for an input, and then use the encrypted value to decrypt the code. All the numbers are outputted in the process, and this was the final result when inputting "3" and "7" as p and q , and "a" as the message. [Here](#) (Appendix A) you can see the whole code with the comments to guide you through the process.

```
===== RESTART: 7:Users/edusanchez2/Documents/Python RSA Code - EE.py =====
Chose a prime number: 3
Chose another prime number: 7
RSA Modulus(n) n=p*q is: 21
Eulers Toitent(r) (P-1(Q-1) is): 12
The encryption key is (e): 5
The decryption key is (d): 5.0
Enter the message you want to encrypt/decrypt: a
The encrypted message is 13
The decrypted message is  a

>>>
```

This code helps me identify the key features of RSA and thus guides me to further my research on my research question, as I will now be calculating the time it takes to decrypt this algorithm. However, the function to find k is very inefficient, and hence makes the algorithm inefficient. This is because, to find k , the algorithm checks all the number possible numbers of k that make d a whole number, which simply takes a lot of processing and unnecessary calculations. In the future, I could adapt the code to the Euclidean Extended Method, which would make the code more effective when finding the value of k . Though I do not do this, it would be faster to find the value of k , and therefore speed up the process. Subsequently, it would enable me to test my algorithm using larger values of n .

Decryption of RSA algorithm

Now that I have performed RSA encryption with python, I can compute a code in order to find the two prime factors of n , and essentially decrypt the RSA encryption. If an interceptor knows the prime numbers p and q that make up n , they can solve d , meaning that they could decipher the encryption.

To find p and q , I created a list that will have the possible prime factors between 2 and n (because 2 is the smallest prime number). Then, the algorithm divides a number, which we can call P (that starts at 2, as it is the lowest prime number), by the number n , and so on until the number P divides into n , which would be the lowest common factor of n . Finally, it will perform a division (P by n) to get the other prime factor (Q). This process itself is not efficient, as it iterates through all the values between 2 and n . So, if n is 1446464876699971, it would be extremely inefficient.

The screenshots below show the algorithm (which can be found in Appendix B) that I created. Also, I put comments on the coding to understand the process, which can be seen in Appendix C.

```
primefactoridentification.py - /Users/edusanchez/Documents/primefactoridentification.py (3.9.6)
#In order to do this, I will first create an input for the user to insert a number that is the product of two primes (N)
N = int(input("What is your value of N?: "))
def prime_fact(N):
    prime_factors = [] #Second, I created a list that will have the possible prime factors between 2 and N (because 2 is the smallest prime number)
    divider = 2
    while divider <= N:
        if N%divider == 0: #If divisor is bigger than x, then there is no way it can divide into x. Each time we run the loop,
            prime_factors.append(divider) #if X is divided by the divisor and the remainder is 0, then it is a factor. I do this by using
            N = N/divider #With this .append function, the divider will be added to the list, aswell as
        else: #Now, if the value of x is divided by the divisor, I will get the other factor, which will be the
            divider += 1 #If the divisor (2) is not a factor of N, the code must still be running to the next possible prime factor
    return prime_factors
print("Your Prime Factors for", N, "are: ", prime_fact(N))
```

```
IDLE Shell 3.9.6
Python 3.9.6 (v3.9.6:db3ff76da1, Jun 28 2021, 11:49:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/edusanchez/Documents/primefactoridentification.py =====
What is your value of N?: 1446464876699971
Your Prime Factors for 1446464876699971 are: [20090893, 71996047]
>>> |
```

Similar to the previous code of RSA, this code is also inefficient, since it iterates through all the values between n and 2, meaning that it will check multiples of 2 that are even numbers and will not be prime. This will be discussed in the evaluation of the code, as it is quite a critical fact about the algorithm that needs to be improved in the future.

As you can see in the screenshot above, I worked out the product of 20090893×71996047 , that gave me 1446464876699971, which I inserted as my value of n , and the program managed to figure out the two prime factors. Now, to determine the time taken to figure out the two prime numbers, I used a python library that calculated the time from before and after the program, so all I had to do was subtract the two values.

Finally, using a python IDLE library named TraceMalloc, I found the amount of memory used by the program to solve the algorithm. The memory used at peak is measured in mebibyte (MiB), which is equal to 2^{20} bytes. In other words, we can identify 1 MiB to be equal to about 1.05 MegaBytes.

```

IDLE Shell 3.9.6
Python 3.9.6 (v3.9.6:db3ff76da1, Jun 28 2021, 11:49:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/edusanchez/Documents/primefactoridentification.py =====
What is your value of N?: 1446464876699971
Your Prime Factors for 1446464876699971 are: [20090893, 71996047]
81.91615796089172 Seconds
(4083, 14089)
>>> |
  
```

Annotations in the image:

- An arrow points from the text "Seconds taken to complete the code" to the value "81.91615796089172 Seconds".
- An arrow points from the text "Memory used at its peak" to the value "(4083, 14089)".

In the sample data collected, 14889 Mib of RAM memory was used by a MacBookPro 2019 in order to factor a number with 16 digits ($1.446464876699971 \times 10^{15}$), and took about 82 seconds to complete.

After multiple attempts of finding data, I managed to find the amount of time taken to find the factors of n for up to 19 digits of n . I recorded it on the table in order to later on use Excel spreadsheets to make a model to predict the amount of time it takes to decode number n that has an x amount of digits.

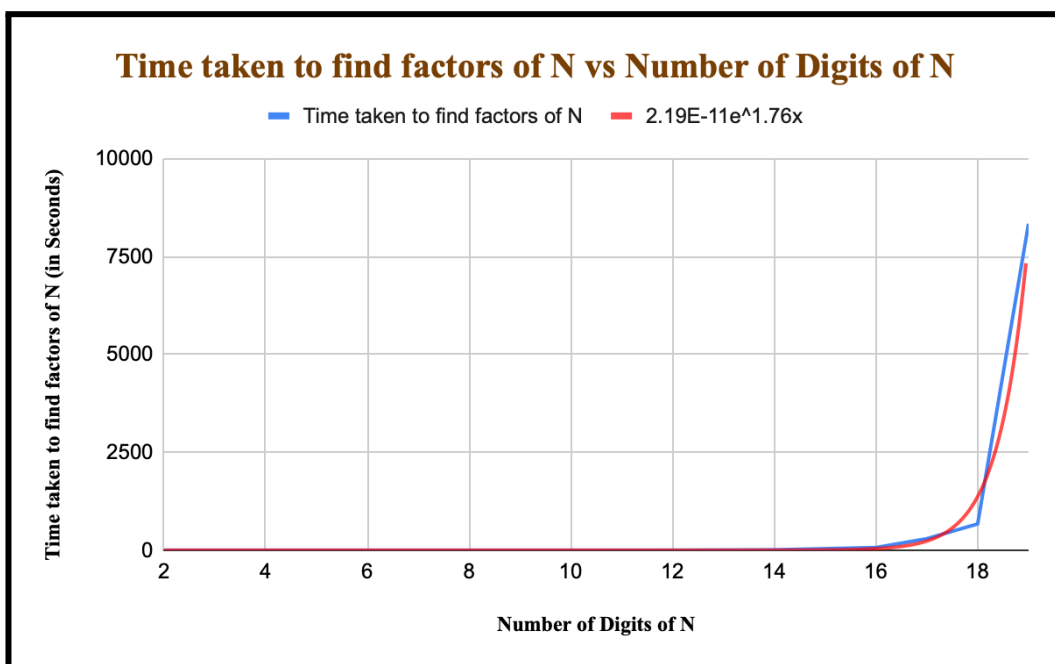
Value of N	Number of Digits of N	Time taken to find factors of n (Seconds)	Memory used to find factor of n (MiB)
21	2	0.02328	33056
403	3	0.03972	33057
9481	4	0.04363	33062
55007	5	0.04257	33062
275141	6	0.05919	33092
3813947	7	0.04549	33096
67858621	8	0.06308	33096
634197919	9	0.09408	33100
2296663169	10	0.15294	33100
47155851707	11	0.75551	33102

405109778273	12	1.06723	33104
42818940967339	14	10.2803	33108
1904120393108471	16	69.1315	34556
28659301043279113	17	294.625	34558
105551548565437487	18	671.078	34560
6618208057895591429	19	8336.01	34570

Although this data has the amount of digits that it is supposed to have, the increase in value is not consistent, which affects the data. For example, 105551548565437487 is a lot smaller than the same 18-digit prime number 890113675671199363, meaning that the time taken may not be entirely accurate for all 18-digit n values, and thus making the model less accurate.

Modelling the data

Using the data above, I created a model relating the number of digits of n , and the time needed to find the two prime factors that form n . With this model, I can approximate the time needed to complete the process of finding the factors of n . Instead of using a linear equation, I noticed that the time taken increases exponentially as the number of digits increases. Consequently, I came to the conclusion that it would be an exponential function, so I used exponential correlation on google spreadsheets to get the equation $y = (2.19 \times 10^{-11})e^{1.76(x)}$. This means, that approximately, an n value with 25 digits would take $(2.19 \times 10^{-11})e^{1.76(25)} = 281450042$ seconds to find the prime factors, which equals to about 3257 days.



RSA encryption uses about 167 digits. Therefore, if I plug $x = 167$ in the equation, we get that 167 digits of n would result in $(2.19 \times 10^{-11})e^{1.76(169)} = 9.734 \times 10^{116}$ seconds in order to find the prime factors. If I divide $\frac{9.734 \times 10^{116}}{60} = 3.09 \times 10^{109}$ years, to complete. Theoretically, the age of the universe is about 13.9×10^9 years old. Thus, if we divide $3.09 \times 10^{109} \div 13.9 \times 10^9 = 6.68 \times 10^{107}$, meaning that it would take

[illegible]

times the age of the universe to complete the calculation.

The memory usage does not change much. This is because it comes to a point where the computer can't use more RAM memory to calculate the factors of n , due to the hardest calculation of the code being *mod* in order to see whether the divisor is divisible by n , so it does not store any data itself. Thus, the highest memory peak does not necessarily change. However, the peak computer power used may decrease if the algorithm was more efficient, as it would run through less iterations per second and hence use less memory, which would enable me to find more data if the time taken for the algorithm to run takes less time. By taking about the even numbers, it could make the processing speed twice as fast, meaning that I could make the prediction model more accurate. Additionally, this will help me completely answer my research question, because by inserting the amount of digits of n in the equation, I will be able to predict how much time it will take to decrypt an encryption. However, after finding d I would have to work backwards and decrypt all the message, which, with a computer program, may still take a long time.

Evaluation of model

Overall, the $y = (2.19 \times 10^{-11})e^{1.76(x)}$ is an approximation to measure time taken to factor n , where y is the time taken for the algorithm to decrypt RSA, and x is the amount of digits the number n has. Moreover, even though this model may not be exactly accurate, it gives us the idea that the graph should be exponential, and that every digit increased will vastly increase the amount of time the program takes to decrypt the number n . Furthermore, as stated before, my code is not entirely efficient, since it runs through all the values between 2 and n (including all even numbers) even though n can't be divided by 2, increasing the running time since it has to go through multiple loops to check if multiples of 2 are factors of n . Thus, in the future, it would make the code more efficient to not check if any multiples of p are multiples of 2. However, it does the job to create an expectation of the graph, as we can expect the graph to have an exponential curve, and further on develop similar and more accurate models, whilst knowing that the correlation is exponential. Additionally, we can deduce that, with a certain amount of digits, the amount of time needed to decode an encryption may take multiple times the age of the universe.

Overall, I doubt this model would work on different computers like Windows, since each computer has a different speed of processing data, and, as time passes, processing speeds will get faster. Similarly, quantum computers will most certainly take over the world of encryption. Hackers will be able to decipher any code in a matter of seconds, as quantum computers have the ability to try several numbers simultaneously, significantly decreasing the amount of iterations needed to decipher RSA. Thus, the internet security and data integrity will be chaos.

Conclusion

The investigation is complete. I managed to use mathematics to solve a very important question that will, for most coders, dawn upon them for the rest of their time using the internet. Will my message ever be decrypted? I found the answer. Probably not. At least not with a modern-day computer with an Intel Core i5 8GB processor like mine (or any other computer). By the time you find the two prime factors of n , the sun may have exploded and the galaxy might not even exist. However, in the future, quantum computers might destroy RSA. After all, at some point, there will be a way to crack it in a matter of seconds. Moreover, I discovered that even though it is theoretically possible to deduce d from e and n in RSA, the time required to do so when n is large makes it an effective solution in practice. However, it is possible to deduce d in a more reasonable amount of time if you have ‘special’ information about n (i.e. p and q). Thus, alluding back to my research question, I can put forth that this research will help readers understand how mathematics is involved in RSA, and based on that, learn how to decrypt it, whilst recognizing that the exponential correlation between the digits of n and the time taken to decrypt it.

Bibliography

- Appendices: Google. (n.d.). Google docs: Sign-in. Retrieved October 26, 2022, from <https://docs.google.com/document/d/1RSkBHIH8276YW6Zs8c0vtATDrKw-vDKZLJoZhAmPusk/edit>
- *Fermat's little theorem*. from Wolfram MathWorld. (n.d.). Retrieved September 19, 2022, from <https://mathworld.wolfram.com/FermatsLittleTheorem.html#:~:text=The%20theorem%20is%20sometimes%20also,not%20sufficient%20test%20for%20primality>
- Google. (n.d.). Google docs: Sign-in. Retrieved September 19, 2022, from <https://docs.google.com/document/d/1J3OIydwAsjNE8cWjhlftlzsAh3VzW5E-sdTIYBKbYdk/edit>
- *Monitoring memory usage of a running python program*. GeeksforGeeks. (2021, November 23). Retrieved September 19, 2022, from <https://www.geeksforgeeks.org/monitoring-memory-usage-of-a-running-python-program/>
- PlanetUnknownPlanetUnknown *What is the relation between RSA & Fermat's little theorem?* Cryptography Stack Exchange. Retrieved September 19, 2022, from <https://crypto.stackexchange.com/questions/388/what-is-the-relation-between-rsa-fermats-little-theorem>
- Singh, S. (2000). *The code book: The secret history of codes and code-breaking cryptography*. Fourth Estate.
- YouTube. (2014). *YouTube*. Retrieved September 19, 2022, from <https://www.youtube.com/watch?v=oOcTVTpUsPQ>.
- YouTube. (2014). *YouTube*. Retrieved September 19, 2022, from <https://www.youtube.com/watch?v=Z8M2BTscoD4>.
- YouTube. (2022). *YouTube*. Retrieved September 19, 2022, from <https://www.youtube.com/watch?v=MScK3qlhYUs>.
- Projects, Contributors to Wikimedia. Wikimedia Commons. Wikimedia Foundation, Inc., April 26, 2022. https://commons.wikimedia.org/wiki/Main_Page.

Appendices Extended Essay

Appendix A: Below is the RSA Algorithm Simulation that I did using python 3.9.

RSA Algorithm Simulator using Python

```
p = int(input("Chose a prime number: "))
q = int(input("Chose another prime number: "))

#The function below gets a number a and checks whether the value inputted is prime. It gets the number,
and if it equals 2, which is the smallest even prime number, then it is a prime. Moreover, if the value
is greater than 2 and, when divided, has no remainders (a%2), the value a is not prime since prime
numbers can't be divided by whole integers. Later, if the value can't be divided by any number between 2
and a, it is a prime number, so q and p are valid numbers.
def prime_check(a):
    if(a==2):
        return True
    elif((a<2) or ((a%2)==0)):
        return False
    elif(a>2):
        for i in range(2,a):
            if not(a%i):
                return False
    return True

check_p = prime_check(p)
check_q = prime_check(q)
while(((check_p==False)or(check_q==False))):
    p = int(input("Enter a prime number for p: "))
    q = int(input("Enter a prime number for q: "))
    check_p = prime_check(p)
    check_q = prime_check(q)

#RSA Modulus
n = p * q
print("RSA Modulus(n) n=p*q is:",n)

#Euler's Totient
phi= (p-1)*(q-1)
print("Euler's Totient(r) (P-1(Q-1) is):",phi)
list_all=[]

#List from 2 to n to find the number e. It does not start at 1 because e must be greater than 1 and
smaller than n
for i in range(2,n):
    list_all.append(i)#Getting all numbers in the list to find e.

#Function to find all the factors of the numbers
def factors(number):
    list1=[]
    for i in range(2,number):
        #if it divides into the number i values (numbers between 2 and n), then its a factor because the
remainder is 0
        if number%i==0:
            list1.append(i)
    return(list1)

#to find the coprimes of the number n and phi
def remove_co_primes(list):
    for i in list:
        for j in range(1,n//2):
            if i*j in list_all:
```

```

list_all.remove(i*j)

#function to delete all the numbers bigger than phi
def remove_bigger(list):
    for i in list:
        if i in list_all:
            list_all.remove(i)

#creating the decryption using the formula  $C=m^e \bmod(n)$ 
def decryption(en):
    K=1
    while i>0:
        formula=(1+(phi*K))%en #the x multiplier is the number that is 1 more than phi, the common
factor with the encryption key
        dec= int(1+(phi*K))/en
        if formula == 0:#if the formula is 0, then the common multiplier will be found.
            return(dec)#decryption is the D value
        K+=K

def encrypt(value):#In this function, I will be encrypting the message inputted by the user
    cypher = (value**en)%n *** means to the power of (in this case the encryption value which is E), then
mod(n)
    return(cypher)

def decrypt(cypher):#In this function, I will be decrypting the message inputted by the user
    decrypted = (cypher**dec)%n *** means to the power of (in this case the encryption value which is E),
then mod(n)
    return(decrypted)

list_n=factors(n)
list_phi=factors(phi)

remove_co_primes(list_n)
remove_co_primes(list_phi)

#The other condition that e has to meet is that it is bigger than 1 and smaller than phi
list_bigger=[]#list of numbers bigger than 1
for i in list_all:
    if i>phi:
        list_bigger.append(i)

remove_bigger(list_bigger)

en=list_all[0]
decryption_key=decryption(en) #I have to use e value in order to decrypt

print("The encryption key is (e): ", en)
print("The decryption key is (d): ", decryption_key)

value = ord(input('Enter the message you want to encrypt/decrypt: '))#ord translate from string code to
ascii code
encrypted_message = encrypt(value)
print("The encrypted message is ", encrypted_message)

decrypted = chr(decrypt(encrypted_message))#chr translates from ascii code to string
print("The decrypted message is ", decrypted)

```

Resulting Code with Different Numbers for p and q :

```
----- RESTART: 7Users/edusanchez/Documents/Python RSA Code - EE.py -----
Chose a prime number: 3
Chose another prime number: 7
RSA Modulus(n) n=p*q is: 21
Eulers Toitent(r) (P-1(Q-1) is): 12
The encryption key is (e): 5
The decryption key is (d): 5.0
Enter the message you want to encrypt/decrypt: a
The encrypted message is 13
The decrypted message is a
>>>
```

Appendix B: Algorithm to find the prime factors of n , that I did using python 3.9.

Algorithm to find prime factors of n using Python

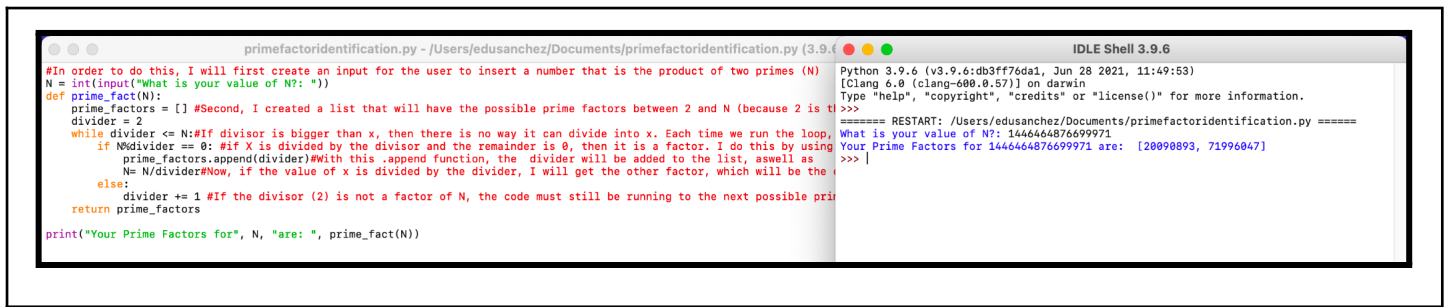
```
import time #library to identify time taken for code to run.

import tracemalloc #library to identify the amount of memory used at the peak of the
program.

startScript = time.time() #getting the time at the start of the script

N = int(input("What is your value of N?: ")) (1)
def prime_numbers(N):
    prime_factors = [] (2)
    divider = 2
    while divider <= N: (3)
        if N%divider == 0: (4)
            prime_factors.append(divider) (5)
            N= N/divider (6)
        else:
            divider += 1 (7)
    return prime_factors

tracemalloc.start() #starting to monitor the time taken
print("Your Prime Factors for", N, "are: ", prime_numbers(N))
print(tracemalloc.get_traced_memory()) #printing the memory used (current,peak) memory
usage.
tracemalloc.stop() #stopping the library
```

```
primefactoridentification.py - /Users/edusanchez/Documents/primefactoridentification.py (3.9.6) IDLE Shell 3.9.6
#In order to do this, I will first create an input for the user to insert a number that is the product of two primes (N)
N = int(input("What is your value of N?: "))
def prime_fact(N):
    prime_factors = [] #Second, I created a list that will have the possible prime factors between 2 and N (because 2 is the smallest prime number)
    divisor = 2
    while divisor <= N: #If divisor is bigger than x, then there is no way it can divide into x. Each time we run the loop,
    if N%divisor == 0: #If X is divided by the divisor and the remainder is 0, then it is a factor. I do this by using %, which represents "mod"
        prime_factors.append(divisor) #With this .append function, the divisor will be added to the list, as well as the n value.
        N = N/divisor #Now, if the value of x is divided by the divisor, I will get the other factor, which will be the other prime if n was the product of two primes.
    else:
        divisor += 1 #If the divisor (2) is not a factor of N, the code must still be running to the next possible prime number. However, I can avoid the complicated code to find the next prime number and simply add 1 to the divisor so that it checks the next value (in this case 3). Therefore, if the code will continue until it finds a factor that divides into the number n, and when it does, it will finish.
    return prime_factors
print("Your Prime Factors for", N, "are: ", prime_fact(N))

Python 3.9.6 (v3.9.6:db3ff76da1, Jun 28 2021, 11:49:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/edusanchez/Documents/primefactoridentification.py =====
What is your value of N?: 1446464876699971
Your Prime Factors for 1446464876699971 are: [28090893, 71996047]
>>> ]
```

Appendix C: Steps and comments for the Algorithm to find the prime factors of N

- In order to do this, I will first create an input for the user to insert a number that is the product of two primes (n).
- Second, I created a list that will have the possible prime factors between 2 and n (because 2 is the smallest prime number). Therefore, the divisor variable will be 2.
- If the divisor is bigger than n , then there is no way it can divide into n . Each time we run the loop, it is important to check whether the divisor is a factor of n .
- If n is divided by the divisor and the remainder is 0, then it is a factor. I do this by using %, which represents "mod"
- With this .append function, the divisor will be added to the list, as well as the n value.
- Now, if the value of n is divided by the divisor, I will get the other factor, which will be the other prime if n was the product of two primes.
- If the divisor (2) is not a factor of n , the code must still be running to the next possible prime number. However, I can avoid the complicated code to find the next prime number and simply add 1 to the divisor so that it checks the next value (in this case 3). Therefore, if the code will continue until it finds a factor that divides into the number n , and when it does, it will finish.

