Introduction

Every day the average person sends information all across the internet, information that contains some of their most sensitive data. Private messages, financial account details, and personally identifiable information all travel along networks that touch thousands of computers, computers that users might not know and might not trust. Through the use of complex encryption methods, this information can all be obfuscated in a way that nefarious actors will have no way to view a person’s most important data. One such method is the RSA public-key cryptosystem. Invented by Ron Rivest, Adi Shamir and Leonard Adleman, RSA relies on the numerical properties of prime numbers and remainders to encrypt data in a way that makes it extremely computationally demanding to ever decrypt without the message recipient’s private key. To demonstrate its functionality, this project walks through the major algorithms required to implement such a system and explains their use within the greater implementation. Starting with Fast Modular Exponentiation, the program leverages squares and modulus to compute the remainder for very large numbers quickly and efficiently. Next it implements a version of the extended Euclidean Algorithm to find the greatest common divisor and modular inverses of two numbers. Finally, it puts everything together to generate keys, encode messages, and decode cyphers. By the end, it will be perfectly clear why this simple technique is so powerful.

Code Demonstration

To Begin, we will first need to generate a public and private key that will allow us to encrypt and decrypt messages respectively. The public key is used to generate encrypted versions of messages and should be given to the sender by the recipient. Since this key only allows an individual to encrypt a message, there is no risk of sending this information over networks. To generate these keys, a recipient must first select two (preferably large) prime numbers. For this example, to match the textbook (Rosen, 300-301), the algorithm will use $43$ and $59$.

Next, the algorithm generates the public key, passing the two values into the “Find\_Public\_Key\_e” function. This function takes two prime numbers as input and returns the public key as a tuple. The public key consists of $(n, e)$ where $n$ is the product of $p$ and $q$, and $e$ is a relatively prime number to the product of $(p - 1)$ and $(q - 1)$

After the algorithm determines the public key, it must generate the private key. The private key is used to generate the plain text versions of a message that was encrypted with the corresponding public key. This key should be kept private and not sent over networks, as the recipient uses this key to decrypt data. The function “Find\_Private\_Key\_d” takes the previously generated value for $e$ as well as $p$ and $q$ and returns the value $d$ such that $d$ is the modular inverse of $e \bmod (p-1)(q-1)$. The private key then consists of $(n, d)$.

To find the inverse, the function leverages the Extended Euclidean Algorithm to find the Bezout coefficients of $e$ and $(p-1)(q-1)$ such that $se + t((p-1)(q-1)) = 1$. In doing so, it can be sure that the value of $s$ is one possible modular inverse of $e$.

Now that both keys have been generated, it is possible to encode a message.

To encode a message, the algorithm calls the “Encode” function. This function takes as input the public key value $n$, the public key value $e$, and the message in the form of a string. It then returns the encrypted message as a python list of integers, where each list item corresponds to a character in the original message. For example, we will use the message “Hello World!” with the previously generated public key.

Under the hood, the function leverages Fast Modular Exponentiation to compute the value of $M^e \bmod n$ where $M$ is the ascii value of each character in the message, and $n$ and $e$ are the public key values. It is important to note that in real world implementations of the RSA algorithm, the original $p$ and $q$ values, and subsequently the public and private keys, are primes with hundreds or thousands of digits. In those cases, Fast Modular Exponentiation is essential because the value of $M^e$ will be larger than normal processors can accommodate, and it would require eons to compute.

Finally, using the private key generated earlier, we can decode the encrypted message to ensure it is valid.

To decode a message, the algorithm calls the “Decode” function. This function takes the private key values $n$ and $d$ and the encoded message as input. Once complete, it returns the decrypted version of the input message.

As demonstrated, the deciphered message is equal to the original plain text message from before. Just as with the “Encode” function, the “Decode” function leverages Fast Modular Exponentiation to compute the value of $C^d \bmod n$ where $C$ is the encrypted message, $d$ is a modular inverse of $e \bmod (p-1)(q-1)$, and $n$ is the product of $p$ and $q$.

Code Exchange Results

Narrative

From this demonstration, it is clear to see just how powerful the RSA algorithm can be. Leveraging Fast Modular Exponentiation and the Extended Euclidean Algorithm, RSA can be used to create complex encryptions that require a great deal of computer power to crack, even at lower values. With the demonstration complete, it is now valuable to discuss some of the techniques and processes used in the project to employ this relatively simple algorithm. I will start by discussing my overall strategy for putting together this project and how the Mastery Workbooks contributed to it’s development. Following that, I will expand on my specific experiences when developing the Fast Modular Exponentiation and Euclidean Algorithm implementations, the core of the project. Finally, I will explain the coding methods I used to complete the remaining functions required to fully implement the project.

For the overall process, I found that the easiest way to tackle this problem was to follow the outline of the Mastery Workbooks. In doing so, I was able to gather a solid understanding of the two important algorithms required to implement RSA. The Mastery Workbooks did an excellent job of demonstrating the fundamental procedures, and core theoretical concepts that are essential for understanding how Fast Modular Exponentiation and the Extended Euclidean Algorithm work. Then, only after building up a strong foundation, the Master Workbooks guide you through developing the implementation of the two algorithms. Had I instead tried to design each algorithm without working through the workbooks, I would not have had as strong of an understanding of how they work, and most certainly would have struggled implementing them. By the time I had completed each Mastery Workbook, I was able to adapt the algorithm for the project with ease, fully understanding each step. The other resource that I leveraged most were the lecture videos on each of the algorithms. It took me a few times watching them to internalize them, but by the time I completed the code in the Mastery Workbooks, I felt very confident in their implementation.

Starting with Fast Modular Exponentiation, I generally followed the pseudocode that was discussed by Sriram in the lecture videos. For the majority of the algorithm, the code did not require much adjustment and I was able to adjust the required variables with relative ease. The only major problem that needed to be solved from the pseudo code was the risk of an infinite loop. Because the loop operates until $n$ is less than $1$, there needs to be a way to update n to reach the exit condition. Additionally, since in each iteration, the algorithm examines the least significant bit of n, there also needs to be a way to either evaluate the next bit or move the next least significant bit into that position to be evaluated. Under normal circumstances, I would have implemented a right shit to accomplish both requirements in one operation. However, because bit shifting was forbidden for the project, I instead used integer division to divide n by 2, effectively accomplishing the same operation. Moving into the code feature portion of the lab, I intend to modify this function by using bit shifting, and bitwise comparisons to slightly optimize operation count and memory space.

For the next major implementation, the Extended Euclidean Algorithm, I again followed the pseudocode by Sriram in the lecture videos. However for this implementation, I had to make a few more adjustments to employ it within the project. First, I implemented an additional if statement at the beginning of the function to ensure that the two integers passed into the function are positive. Next, I considered flipping the two integers to ensure that the larger of the two was first, although this proved to cause more issues than fix. I found that I did not run into any anomalous return values when the first integer was less than the second, but I did have issues with ensuring the Bezout coefficients correspond to the correct integer. Instead of attempting to correct the latter issue, I neglected to implement the original swapping, leaving the code as is. Finally, unlike the pseudocode, I only employed one pair of temporary variables to update $s\_1, t\_1$ and $s\_2, t\_2$.

Ater employing the two core functions above, all that remained were the functions to generate keys, encode messages, and decode messages. Since there was not any direct lecture with pseudo code for these functions, they required more work, but proved to be significantly less complex algorithms than the previous.

Starting with the public key generation, I began the code by calculating $n$ as the product of $p$ and $q$. Then, to find $e$, I first found the product of $(p-1)$ and $(q-1)$, then iterated through all values less than the smaller of $p$ and $q$ until I found an value that was relatively prime. I chose to start with the largest value and decrement, to identify the largest possible value of e, instead of smallest.

For the private key generation, I initially assumed that this would be more difficult but found it to be slightly easier. With the understanding that the Bezout coefficients are modular inverses, the only operation I had to perform was to pass $e$ and the product of $(p-1)$ and $(q-1)$ into the Extended Euclidean Algorithm, and then set $d = s$. To ensure $d \geq 0$ I then implemented a while loop that continued to add the product of $(p-1)$ and $(q-1)$ to $d$ until $d \geq 0$.

To implement the Encode and Decode functions, I relied heavily on example 8 and 9 in the textbook pages 300 and 301. Using that resource, and list comprehension in python, I was easily able to implement a few lines of code to encode and decode messages with Fast Modular Exponentiation.

After completing all code elements, I was eager to test it. Trying my code on the initial encrypted message I ran into an issue. I had at first neglected to adjust the variables of my code implementations to the required ones in the project. After making the appropriate changes, I was pleasantly surprised to get a full English sentence on my first attempt to decode. Following the initial attempt, I only again ran into issues when my code execution was out of order in the Jupyter Lab code blocks.

Overall the most challenging part of this project was understanding the core algorithms. Leveraging the Mastery Workbooks and the video lectures I was able to gain a full in depth understanding and employ them successfully.

* Describe your process. Did you plan ahead? Did you wing it? What have you learned about your own process?
* Describe the results of exchanging keys and codes with classmates. Did it work the first time or did you need to make adjustments? Use the word fortitude
* What was most challenging? Use a cooking metaphor.
* Who helped you with your project? Which resources were most helpful?
* What was your Best Mistake? (funniest? most frustrating? the one you learned the most from?)

Code Breaking

To implement an algorithm that can decrypt a message without using a private key, it must be able to generate a modular inverse of $ e \bmod (p-1)(q-1)$ with only knowing the value of $n$. Since $n$ and $e$ are part of the public key, it is assumed that in this case those two values are known. To originally generate the public key, the algorithm computes the modular inverse of $e \bmod (p-1)(q-1)$ using $e, p, q$. Therefore, to generate a private key using the public key, there needs to be a way to find the value of $p$ and $q$. Since $n = pq$, the most straightforward approach would be to find all the factors of $n$ and repeatedly test them as inputs to the decode function until a readable English sentence is produced.

To test this this process, I will attempt to decode the original message posted by Professor Stade without the public key.

Now that we have produced all the factors of $n$ we need to systematically input them into the Find\_Public\_Key\_d function and attempt to decode with the returned private key. The function below first generates a list of all factors of $n$. Then, the for loop iterates over the half of the list of factors, generating a $d$ for each pair and then using that value as the private key in the decode function.

Since the provided public key has a very low amount of factors, we only need to try two different public keys to break the cypher text. This demonstrates why having large prime values for $p$ and $q$ will provide the most security. When $p$ and $q$ are large, $n$ is large, and when $n$ is large, the list of factors is extensive.

To show further, we can attempt the same decode with a more difficult public key.

To measure the time that this takes, I will import the time module and implement a timer using this guide from Geeks for Geeks.

<https://www.geeksforgeeks.org/python-measure-time-taken-by-program-to-execute/>

One important observation from our initial trials, is that despite $n$ being large, the algorithm only produces a few private keys.

To improve the code breaking function we must look at improving our factorization algorithm. Currently, the function runs at $O(n - 2)$ and since it must check every value from $2$ to $n-1$ it also runs at $\Omega(n - 2)$. However, we can improve this in a few ways. Since p and q must be prime we can immediately rule out $1$, $n$, and all even factors of n. To do this, we can adjust the factorize function below

* Explain in detail how breaking works in this section using markdown blocks. Demo a short example with a small n. What are the complete steps? What tools do you need? Use a sports metaphor.
* Try brute forcing a larger value of n, say 15-18 digits long. What do you notice? (If this is taking a while, just ride it out, or Ctrl+C will KeyboardInterrupt the cell block). Provide a sample large n that will take ~3-5 min to crack.
* Optional Challenge: Consider illustrating this with python math and time modules. matplotlib and time are good examples. Documentation is available.
* How secure is RSA? Why is it difficult to break codes in RSA?
* Is our implementation of RSA secure? What are it's flaws?

Custom Code Feature