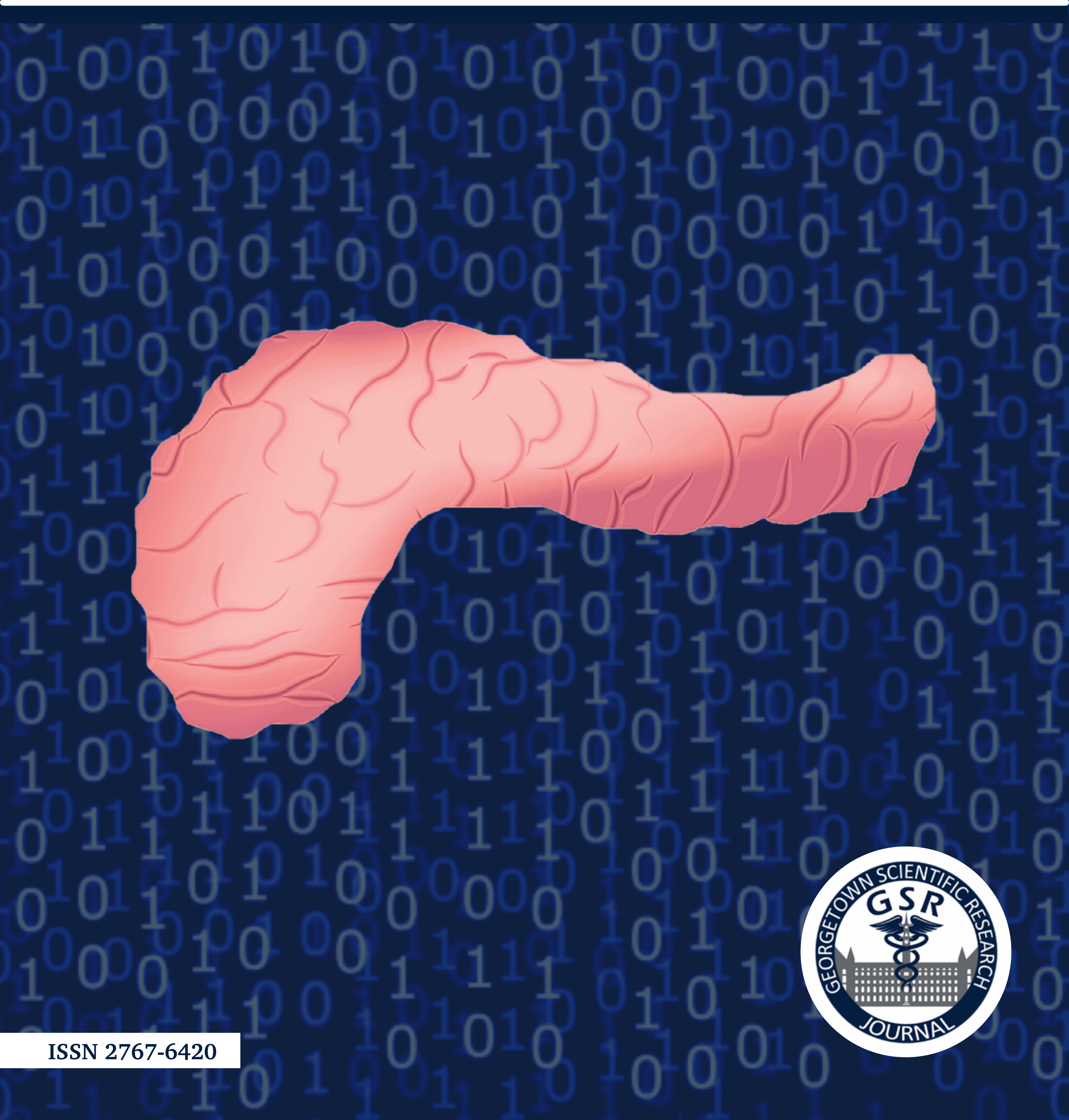
GSR Journal

Georgetown Scientific Research Journal

Volume 1 | Edition 2
July 31, 2021



ISSN 2767-6420

GEORGETOWN SCIENTIFIC RESEARCH JOURNAL

Volume One
Edition Two
Spring 2021

An Investigation Into the Mathematics of Decryption Techniques in RSA Encryption, With an Implementation in Python

Sofia Flynn

An Investigation Into the Mathematics of Decryption Techniques in RSA Encryption, With an Implementation in Python

Sofia Flynn

Department of Mathematics, Georgetown University, Washington D.C.
E-mail: smf133@georgetown.edu

https://doi.org/10.48091/gsr.v1i2.18

Abstract

This study explores the mathematics of two different techniques that can be used to access the decryption key in RSA encryption including semi-prime factorization and a logarithmic method. The study then presents a Python program, written by the author, that automates the calculations for either of the decryption techniques and also calculates the number of iterations required to determine the decryption key in either circumstance. Most importantly, the program utilizes only values of the RSA encryption algorithm that would be made publicly available in actual circumstances to calculate the decryption key so as to mimic real-life occurrences with as much integrity and accuracy as possible.

Keywords: RSA encryption, semi-prime factorization, decryption, Python

1. Introduction

RSA encryption was created in the 1970s as an asymmetric (public-key) form of cryptography that would replace the then-commonly utilized and weakening symmetric cryptography.[1] In symmetric cryptography, in order to exchange secret messages, two individuals would either have to meet beforehand to exchange identical keys or run the risk of their keys being intercepted.[2] In a simplified example of symmetric cryptography, suppose "Bob" wanted to send a secret message to "Alice." Alice and Bob would first have to meet each other to share identical keys that would be used for encryption and decryption, ensuring that each person ultimately possesses the same key. Using one of the identical keys, Bob could then encrypt his secret message and send it to Alice, who would then decrypt the message using the other identical key that she and Bob had exchanged beforehand. If Bob and Alice cannot meet to share keys, then Bob would have to send the key that decrypts his encrypted message along with his encrypted message to Alice, running the risk that a third party could easily intercept both the encrypted message and key before it comes into Alice's possession.

In 1970, James Ellis proposed a hypothetical idea in which secret messages could be exchanged between individuals without them having to meet beforehand to share identical keys, thereby eliminating dependence on the ever-weakening symmetric cryptography.[3] The premise was simple: if Bob wished to send a secret message to Alice, Alice could create a lock and a key that decrypted, or "unlocked', that lock. Alice could then send the open lock to Bob, who would proceed to encrypt, or "lock", his secret message using the open lock. Alice would retain possession of the key that decrypted the lock. Bob would then send his encrypted message to Alice, who would decrypt it using the key that remained in her possession throughout the entire process. Such is the conceptual idea behind asymmetric cryptography; by preventing an initial exchange of keys, one avoids the risk of a third party intercepting the key used to decrypt the encrypted message.

In 1973, Clifford Cocks invented a mathematical implementation of Ellis' idea, which had remained classified for twenty-four years.[4] Later on in 1977, Ron Rivest, Adi Shamir, and Leonard Adleman created the equivalent of Cocks' implementation of Ellis' idea completely independently.[5] The algorithm they created based on Ellis' conceptual idea about asymmetric cryptography was henceforth known as RSA (each of the founders' initials) encryption and is widely used today. RSA has formidable strength; as of yet, the only device that is considered to be capable of breaking RSA is a quantum computer.[6]

Rivest, Shamir, and Adleman's algorithm consists of six values that correspond to either a public or private key. The public key is released publicly and can be used by anyone to encrypt a message; the private key, on the other hand, is kept hidden by the "key generator" (the person who creates the public and private key values; the equivalent of "Alice" in the latter example above) and is used by the key generator to decrypt any encrypted message sent to him or her. The six values generated by the key generator are denoted: P, Q, N, φN, E, and D. P, Q, φN, and D belong to the private key and are kept secret by the key generator. N and E belong to the public key and are released publicly.

The key generator must generate values for P, Q, N, φN, E, and D. The key generator must first choose two large prime numbers P and Q. In real world utilizations of RSA encryption, these prime numbers are typically large, usually hundreds of digits long. The key generator must then multiply these two prime numbers together to attain the value of N, that is, P x Q = N. N is a semi-prime number, or the product of two prime numbers. Next, the key generator calculates the φ or "phi" of N. The phi of a number outputs how many positive integers are less than that number that do not share any common factors greater than one with the number. For example, to find the phi of nine, a key generator would list all of the integers less than nine but greater than or equal to one (eight, seven, six, five, four, three, two, one). From this list, they would then evaluate which integers share common factors with nine that are greater than one. Six and three both share common factors greater than one with nine while eight, seven, five, four, two, and one do not share any common factors greater than one with nine. The phi of nine is six because there are six integers less than nine that do not share any common factors greater than one with nine. Calculating the phi of N is straightforward for the key generator due to the fact that the phi function is multiplicative, meaning that:

$$\varphi N = \varphi P \times \varphi Q \text{ (equation 1)}$$

or the phi of N is equivalent to the phi of each of N's factors multiplied together. This property can be proved using the Chinese Remainder Theorem but such a proof is omitted in this paper.[7] Thus, the key generator can calculate the phi of P and the phi of Q and multiply these two values together to attain the phi of N. The phi of any prime number is simply one less than the original prime number because the only factors that prime numbers have are one and itself.[8] For example, the phi of five (a prime number) is four because there are four positive integers less than five that do not share any common factors greater than one with five. Thus, the phi of the prime number P is (P – 1) and the phi of the prime number Q is (Q – 1). Thus, the phi of the semi-prime N is (P – 1)*(Q – 1). To generate E, the key generator must pick an integer that satisfies the requirement: $1 < E < \varphi N$; E shares no common factors with φN. Finally, to generate D, the key generator must pick an integer that satisfies the requirement:

$$(E^*D) \bmod \varphi N = 1 \text{ (equation 2)}$$

The operation "mod" outputs the remainder of a division. For example, 8 mod 6 = 2 because the remainder of the division eight divided by six is two.

Once the key generator has generated values for P, Q, N, φN, E, and D, they must release the values of N and E (the public key values) to the public but retain the values of P, Q, φN, and D (the private key values). If a person (the "sender"; the equivalent of "Bob" in the latter example above) wishes to send a secret message to the key generator, they can use the values of N and E to encrypt their message using the expression: M^Emod N, where M is the secret message to be encrypted. The sender would then send this encrypted message M^Emod N (the result of which is known as "C", for ciphertext message) to the key generator. The key generator can then decrypt the encrypted message C by using the private key value of D through the expression: C^Dmod N. The result of C^Dmod N yields the original message M. Thus, in summary:

M^Emod N = C (equation 3)

and

C^Dmod N = M (equation 4)

with proof of correctness derived by Rivest, Shamir, and Adleman using Fermat's Little Theorem in their publication.

As can be seen, RSA encryption functions similarly to the simplified example of asymmetric encryption given earlier (Introduction, paragraph two). Alice, the key generator, creates a public and private key containing the values of P, Q, N, φN, E, and D. N and E function as the values that Bob uses to lock the open lock that Alice created and sent to him, while the values of P, Q, φN, and D function as the key that opens the lock once Bob locks it with his message and sends it to Alice.

Although publicly releasing the values of N and E does not detract from the strength of asymmetric RSA encryption, it is integral that the values of P, Q, φN, and D remain hidden and can only be accessed by the key generator: this is because RSA encryption was built around two main assumptions that make it safe and acceptable to release the values of N and E publicly. First, it is exceedingly time-consuming to calculate the decryption key (D) by factoring large semi-prime numbers. Second, it is exceedingly time-consuming to calculate the decryption key (D) using properties of logarithms because of the discrete logarithm problem. If the decryption key D was accessed by a person other than the key generator, then this person would be able to intercept secret encrypted messages meant for the key generator and decrypt them using the intercepted value of D. In short, RSA is a secure cryptosystem because accessing the decryption key by semi-prime factorization or logarithmic properties both prove to be too time-consuming to be pursued realistically, at least until the advent of quantum computing.

If a "message-interceptor" (the equivalent of a "hacker") wants to calculate the value of the decryption key using only the publicly-available values of N and E, they have two options, with the first being semi-prime factorization.[9] When semi-prime factorization is used by a message-interceptor in an attempt to gain access to the decryption key, the magnitude of the semi-prime number proves to be of paramount importance in preventing the message-interceptor from calculating the decryption key. In the case of RSA encryption, the semi-prime number is N (the product of the two prime numbers P and Q), whose value is publicly available to the message-interceptor. The message-interceptor must factor the semi-prime number N into its constituent factors P and Q. From there, the message-interceptor can plug in the values of P and Q into the expression (P – 1)*(Q – 1) to find φN. After having attained the value of φN, the message-interceptor, knowing that (E*D) must equal 1 more than φN in order to achieve a modular relationship with 1 as a remainder, can plug this value along with the public key value of E into equation 2 and solve for D. For example, if E is 7 and φN is 3120, then the formula would be: 7D mod 3120 = 1, and thus, 7D must equal 3121 for the relationship to work. With this in mind, the message-interceptor can easily solve for D:

7D = 3121

D = 445.857143 . . .

D, however, must be an integer value. In order to satisfy this rule, the message-interceptor can look for other modular relationships that yield a remainder of 1. For example, 6241 mod 6240 =1.

Similarly, 9361 mod 9360 = 1. The message-interceptor can keep on multiplying the value of 3120 by increasing integers to achieve different modular relationships that will still yield 1 as a remainder until D is an integer value.
Example:

7D mod (3120*1) = 1; 7D = 3120*1 + 1; D = (3120*1 + 1)/7; D = decimal

7D mod (3120*2) = 1; 7D = 3120*2 + 1; D = (3120*2 + 1)/7; D = decimal

7D mod (3120*3) = 1; 7D = 3120*3 + 1; D = (3120*3 + 1)/7; D = decimal

7D mod (3120*4) = 1; 7D = 3120*4 + 1; D = (3120*4 + 1)/7; D = integer

Here, solving for D only requires 4 iterations. The formula for solving for D simplifies to:

D = (φN* φNMultiplier + 1)/E (equation 5)

where "φNMultiplier" is a specific integer value that the message-interceptor multiplies φN by to achieve different modular relationships that yield 1. With knowledge of the publicly-available value of E as well as the value of φN, the message-interceptor can easily calculate the value of the decryption key, D, having created one equation with only one unknown value. However, one problem exists: factoring N into its constituent factors P and Q becomes increasingly time-consuming as N grows. This is why large prime numbers had to be chosen by the key generator as values for P and Q. The larger N is (the product of the already large values of P and Q), the more time it will take to factor it. In RSA, the types of semi-prime numbers utilized are typically around 600 digits long. Semi-prime numbers of such magnitudes take a huge amount of time to factor using current technologies. In fact, in 1978, Rivest, Shamir, and Adleman predicted that factoring a 500-digit semi-prime number would take approximately 4.2 x 10^{25} years to factor using the Schroeppel factoring algorithm.[10] Therefore, due to the immense amount of time needed to factor N into its constituent factors P and Q, it is impractical for a message-interceptor to use semi-prime factorization to calculate the decryption key, D.[11]

The second option that the message-interceptor has if he wishes to calculate the value of the decryption key using only the publicly available values of N and E is to use the properties of logarithms. Recall that encrypting a message requires plugging a secret message M into the expression M^Emod N, and thus decrypting that message requires plugging in the result of M^Emod N (which is C) into the expression C^Dmod N to attain the original message M. The message-interceptor, like all others, has access to the public key values of N and E. He can generate a secret integer message M and plug it into the expression M^Emod N to attain the ciphertext/encrypted message C. Then, they can plug the value of C along with the public key value of N into the expression C^Dmod N. The message-interceptor knows that the result of C^Dmod N must equal his original message M (see equation 4). The message-interceptor has knowledge of the value of N (a public value) as well as C and M, because he generated these values. Therefore, the message-interceptor has created an equation in which only one unknown, D, the decryption key, exists. For example, if N is 143 and E is 7, and the message-interceptor picks his secret message M to be 24, then the enciphered message would be 24^7mod 143, or 106. The message-interceptor can plug in the values of 143 (N), 24 (M) and 106 (C) into equation 4, yielding 106^Dmod 143 = 24, and subsequently solve for D. Knowing that 106^D must equal to 143 + 24, or 167, for the above equation to work due to modular arithmetic (167 mod 143 = 24), the message-interceptor can set 106^D to 167. Knowing that 106^D = 167, the message-interceptor can easily solve for D using the properties of logarithms:

$$106^D = 167$$

$$\log 106^D = \log 167$$

$$D (\log 106) = \log 167$$

$$D = \log 167/\log 106$$

$$D = 1.09747\ldots$$

However, D has the restriction that it must be an integer value, which the above value is not. Thus, the message-interceptor needs to come up with another way to calculate D using this method. The message-interceptor knows that 167 mod 143 = 24, but there are other values that, when divided by each other, will yield a remainder of 24. 143 multiplied by 2 is 286. 286 plus 24 is 310. Thus, 310 mod 286 = 24. Similarly, 143 multiplied by 3 is 429. 429 plus 24 is 453. Thus, 453 mod 429 = 24. The message-interceptor can keep on multiplying the value of 143 by increasing integers to achieve different modular relationships that will still yield 24 as a remainder until D is an integer value.

Example:

106^D mod (143*1) = 24; 106^D = (143*1) + 24; log(106^D) = log(143*1 +24); D = log(143*1 + 24)/log(106); D = decimal

106^D mod (143*2) = 24; 106^D = (143*2) + 24; log(106^D) = log(143*2 +24); D = log(143*2 + 24)/log(106); D = decimal

106^D mod (143*3) = 24; 106^D = (143*3) + 24; log(106^D) = log(143*3 +24); D = log(143*3 + 24)/log(106); D = decimal

The formula for solving for D in this way can be simplified to:

D = log(N*NMultiplier + M)/log(C) (equation 6)

where "NMultiplier" is the increasing integer by which the message-interceptor multiplies N to achieve different modular relationships that will still yield the secret message, M. Even though the message-interceptor can keep on running this type of sequence to achieve an integer value for D, the amount of iterations it would take to finally attain an integer value for D is astronomical, making the logarithmic decryption key calculations just as impractical and time-consuming as semi-prime factorization. The difficulty of calculating the exponent D in this way is known as the discrete logarithm problem, or the "RSA problem."

Decryption through semi-prime factorization and decryption through logarithmic properties are both processes that are referred to as "trapdoor one-way functions".[12] Trapdoor one-way functions are easy to compute in one direction but difficult to compute in the other direction unless special "trapdoor" information is known. In the case of decryption through semi-prime factorization, Rivest, Shamir, and Aldeman realized that it was trivially easy to multiply two large prime numbers P and Q together yet infinitely more difficult and time-consuming to factor the resulting semi-prime number, N. Similarly, they realized that it was easy to calculate M in equation 4 given the values of C, N, and most importantly, D, but far more arduous to calculate the value of D in equation 4 given the values of C, N, and M. These two utilizations of trapdoor one-way functions in RSA encryption are what make it so formidably strong and currently unbreakable without a quantum computer.[13]

In the case of decryption through either semi-prime factorization or logarithmic properties, an immense number of iterations are required to finally attain the decryption key, D. In the case of semi-prime factorization, if every integer less than the semi-prime number, N, and greater than or equal to two was tested to see if it divided evenly into N, the number of iterations required would be N – 2. If N is a huge semi-prime number that is hundreds of digits long, the number of iterations required to factor N would be two subtracted from the huge semi-prime number. While it is true that not all the numbers less than N would have to be assessed if it was certain that N was semi-prime (because this would narrow the numbers that needed to be tested down to only prime numbers), if access to a large bank of prime numbers is not unattainable (as in this project), all integers greater than or equal to two and less than N would still have to be tested to see if they divide evenly into N, making the expression for the amount of iterations needed to factor N in this project N – 2. In the algorithm presented for factoring N in this manner, time complexity is O(N). In the case of decryption through logarithmic properties, $\frac{10^{(logC)*D} - M}{N}$ iterations are required to acquire D. Because the formula for solving for D through logarithmic techniques can be simplified to equation 6, where "NMultiplier" is the increasing integers the message-interceptor multiplies N by to achieve different modular relationships that will still yield the secret message, M, "NMultiplier" can be isolated and solved for to attain the number of iterations it would take to solve for D. The following demonstrates the process of isolating "NMultiplier" from the equation D = log(N*NMultiplier + M)/log(C):

$$D = \log(N*NMultiplier + M)/\log(C)$$

$$D*[\log(C)] = \log(N*NMultiplier + M)$$

$$10^{\{D*[\log(C)]\}} = N*NMultiplier + M$$

$$10^{\{D*[\log(C)]\}} - M = N*NMultiplier$$

$$NMultiplier = \frac{10^{\{D*[\log(C)]\}} - M}{N} \text{ (equation 7)}$$

By solving for "NMultiplier", the number of iterations needed to calculate D through logarithmic techniques has been calculated to be $\frac{10^{\{D*[\log(C)]\}} - M}{N}$. Because this program allows a maximum of one million iterations to compute D in this manner, the associated time complexity is less than O(1000000).

The purpose of this research study is to discuss the function of a program built using Python that calculates the decryption key using semi-prime factorization as well as the logarithmic method. Further, the program calculates the number of iterations required to calculate D using either of these methods, indicating the efficacy and advantageousness of using one method over the other in different circumstances. This research study will delve into explaining the code written in the program and how it contributes to achieving the end result of decryption as well as the number of iterations derived necessary to calculate D using either of the aforementioned methods, being that of semi-prime factorization or logarithms.

2. Materials

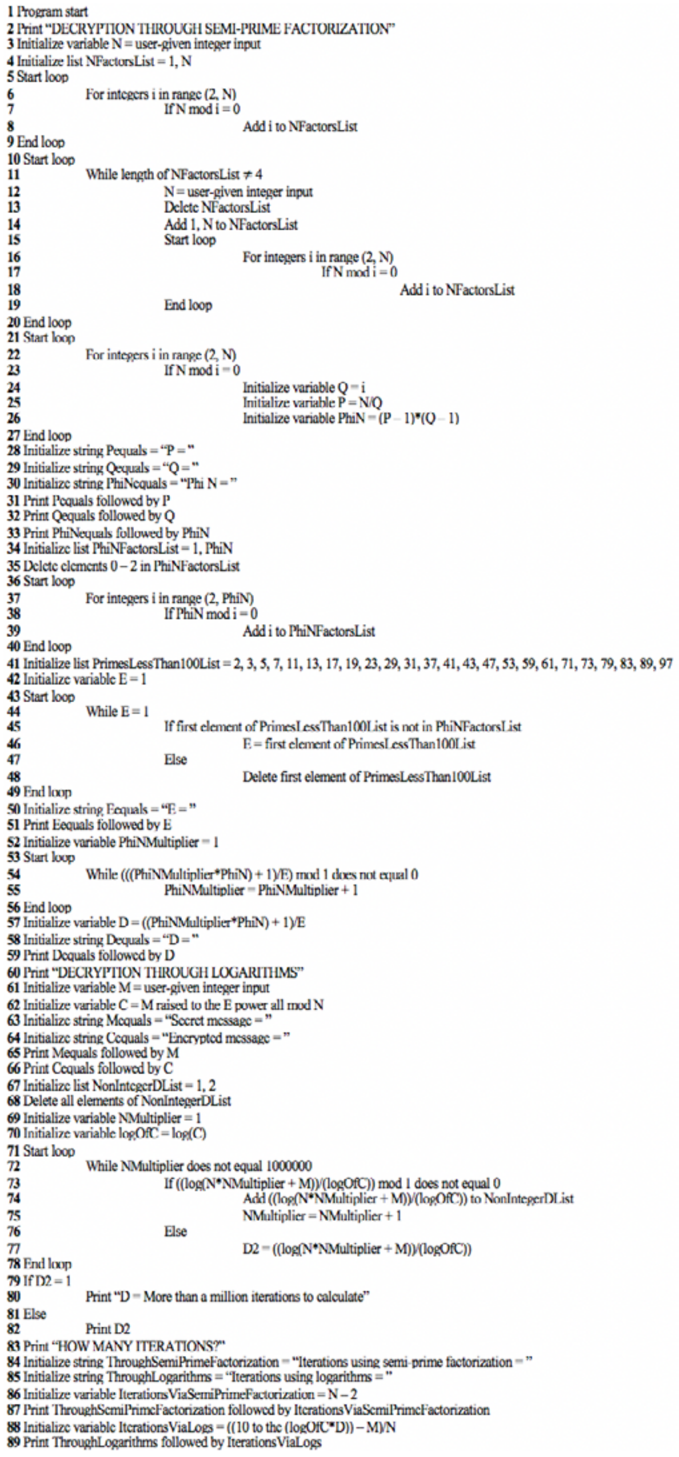
2.1 Required Materials:

Python programming system installed on MacBook laptop or other personal computer, complete with IDLE and running module.
Access to Python's "math" module located in Python's standard library. The module includes mathematical functions necessary to the operation of the program.

2.1 Recommended Materials:

Access to a computational knowledge/answer engine like Wolfram Alpha used to initially verify the accuracy of the results obtained from the program.[14]

2. Methods/Procedures



```
1 Program start
2 Print "DECRYPTION THROUGH SEMI-PRIME FACTORIZATION"
3 Initialize variable N = user-given integer input
4 Initialize list NFactorsList = 1, N
5 Start loop
6               For integers i in range (2, N)
7                               If N mod i = 0
8                                               Add i to NFactorsList
9 End loop
10 Start loop
11              While length of NFactorsList ≠ 4
12                              N = user-given integer input
13                              Delete NFactorsList
14                              Add 1, N to NFactorsList
15                              Start loop
16                                              For integers i in range (2, N)
17                                                              If N mod i = 0
18                                                                              Add i to NFactorsList
19                              End loop
20 End loop
21 Start loop
22              For integers i in range (2, N)
23                              If N mod i = 0
24                                              Initialize variable Q = i
25                                              Initialize variable P = N/Q
26                                              Initialize variable PhiN = (P – 1)*(Q – 1)
27 End loop
28 Initialize string Pequals = "P = "
29 Initialize string Qequals = "Q = "
30 Initialize string PhiNequals = "Phi N = "
31 Print Pequals followed by P
32 Print Qequals followed by Q
33 Print PhiNequals followed by PhiN
34 Initialize list PhiNFactorsList = 1, PhiN
35 Delete elements 0 – 2 in PhiNFactorsList
36 Start loop
37              For integers i in range (2, PhiN)
38                              If PhiN mod i = 0
39                                              Add i to PhiNFactorsList
40 End loop
41 Initialize list PrimesLessThan100List = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 71, 73, 79, 83, 89, 97
42 Initialize variable E = 1
43 Start loop
44              While E = 1
45                              If first element of PrimesLessThan100List is not in PhiNFactorsList
46                                              E = first element of PrimesLessThan100List
47                              Else
48                                              Delete first element of PrimesLessThan100List
49 End loop
50 Initialize string Eequals = "E = "
51 Print Eequals followed by E
52 Initialize variable PhiNMultiplier = 1
53 Start loop
54              While (((PhiNMultiplier*PhiN) + 1)/E) mod 1 does not equal 0
55                              PhiNMultiplier = PhiNMultiplier + 1
56 End loop
57 Initialize variable D = ((PhiNMultiplier*PhiN) + 1)/E
58 Initialize string Dequals = "D = "
59 Print Dequals followed by D
60 Print "DECRYPTION THROUGH LOGARITHMS"
61 Initialize variable M = user-given integer input
62 Initialize variable C = M raised to the E power all mod N
63 Initialize string Mequals = "Secret message = "
64 Initialize string Cequals = "Encrypted message = "
65 Print Mequals followed by M
66 Print Cequals followed by C
67 Initialize list NonIntegerDList = 1, 2
68 Delete all elements of NonIntegerDList
69 Initialize variable NMultiplier = 1
70 Initialize variable logOfC = log(C)
71 Start loop
72              While NMultiplier does not equal 1000000
73                              If ((log(N*NMultiplier + M))/(logOfC)) mod 1 does not equal 0
74                                              Add ((log(N*NMultiplier + M))/(logOfC)) to NonIntegerDList
75                                              NMultiplier = NMultiplier + 1
76                              Else
77                                              D2 = ((log(N*NMultiplier + M))/(logOfC))
78 End loop
79 If D2 = 1
80              Print "D = More than a million iterations to calculate"
81 Else
82              Print D2
83 Print "HOW MANY ITERATIONS?"
84 Initialize string ThroughSemiPrimeFactorization = "Iterations using semi-prime factorization = "
85 Initialize string ThroughLogarithms = "Iterations using logarithms = "
86 Initialize variable IterationsViaSemiPrimeFactorization = N – 2
87 Print ThroughSemiPrimeFactorization followed by IterationsViaSemiPrimeFactorization
88 Initialize variable IterationsViaLogs = ((10 to the (logOfC*D)) – M)/N
89 Print ThroughLogarithms followed by IterationsViaLogs
```

Figure 1. Code script

3.1 The Code Behind Semi-Prime Factorization to Calculate D (Lines 1-47)

3.1.1 Factoring N

In lines 3 – 9, the program asks for a semi-prime input which it then sets to the variable N. The program is designed to test all integers less than N but greater than or equal to two to determine whether they divide evenly into N. An integer is a factor of N if the remainder of N divided by said integer equals zero. The integers that do divide evenly into N are deemed to be factors of N and are appended to a list called "NFactorsList".

In lines 10 – 20, the program forces another input to be given for N if the first input is not a semi-prime number. If the length of NFactorsList is only four, this indicates that N is a semi-prime number because semi-prime numbers only have four factors: one, the semi-prime itself, and the two prime numbers multiplied together to attain the semi-prime number.

In lines 21 – 25, after ensuring a semi-prime input for N, the program sets the variables P and Q to the two prime numbers that N is divisible by.

3.1.2 Factoring φN

In lines 26 – 27, the program sets the variable PhiN to (P – 1)*(Q – 1) based on the mathematics described in the introduction for the calculation of φN.

3.1.3 Generating a public key value for E

In lines 34 – 40, the program initializes a list called "PhiNFactorsList" that lists the factors of the value of φN by testing which integers are greater than two but less than φN and yield a remainder of zero when divided into φN.

In lines 41 – 42, the program initializes a list called "PrimesLessThan100List" which lists all of the prime numbers less than 100 and also initializes that variable E by setting it to a placeholder value of 1.

In lines 43 – 49, the program generates a value for E, which must be greater than 1, less than φN, and share no common factors with φN. The program cross-checks values in the PrimesLessThan100List against values in the PhiNFactorsList until it finds a value in the PrimesLessThan100List that is not in the PhiNFactorsList, which it then sets it to the variable E. By checking primes as possible values for E against the factors of φN, the program ensures that whatever value it picks for E shares no common factors with φN.

3.1.4 Generating D

In lines 52 – 56, the program sets a variable called "PhiNMultiplier" to 1, which will act as φNMultiplier in equation 5. While D does not equal an integer solution, the program continuously adds 1 to the value of PhiNMultiplier. When PhiNMultiplier iterates enough that it yields an integer value for D, the program sets D to (φN* φNMultiplier + 1)/E, where φNMultiplier represents the number of iterations it took to yield an integer solution for D.

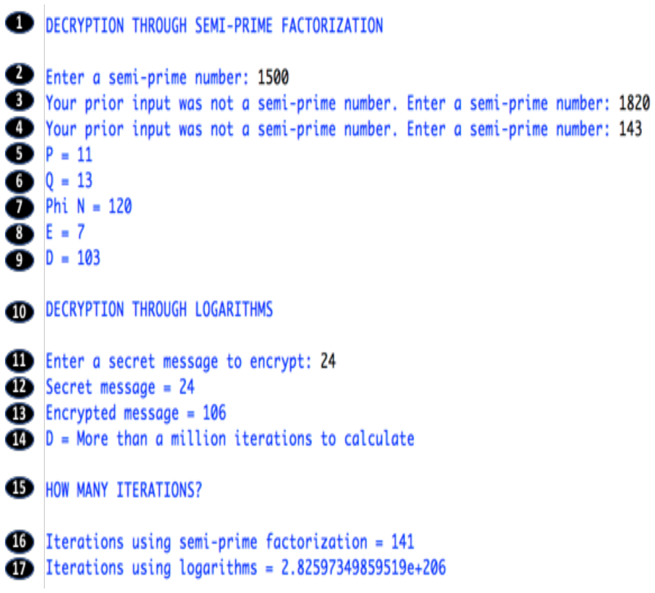
3.2 The Code Behind Logarithmic Techniques to Calculate D (Lines 48 – 71)

In lines 61 – 62, the program asks for an integer input and sets it to the variable M. It then encrypts M by raising it to the value of E, dividing the entire expression by N, and taking the modulus of the division, which is C, the ciphertext message. In lines 67 – 82, the program initializes a variable called "NMultiplier" to 1 which acts as NMultiplier in equation 6. Because Python has a limited capacity for running large numbers of iterations, the program continuously adds 1 to NMultiplier until NMultiplier = 1000000 (the first million iterations) instead of until D is an integer. When NMultiplier = 1000000, although D might not be an integer, the program still cuts off at this point and prints that D takes more than one million iterations to calculate. All non-integer numbers resulting from log(N*NMultiplier + M)/log(C) are appended to the list "NonIntegerDList" and stored.

3.3 The Code Behind Iterations for Semi-Prime Factorization to Calculate D and Logarithmic Techniques to Calculate D (Lines 72 – 80)

In lines 86 – 89, the iterations for decryption using semi-prime factorization and for decryption using logarithmic techniques are both calculated using the expressions N − 2 and $\frac{10^{(\log C)*D} - \mathrm{M}}{N}$, respectively (see introduction of derivation of iterations). The program then prints the number of iterations needed in either case, allowing for quick comparison of either method's effectiveness in different circumstances.

4. Results



```
1  DECRYPTION THROUGH SEMI-PRIME FACTORIZATION

2  Enter a semi-prime number: 1500
3  Your prior input was not a semi-prime number. Enter a semi-prime number: 1820
4  Your prior input was not a semi-prime number. Enter a semi-prime number: 143
5  P = 11
6  Q = 13
7  Phi N = 120
8  E = 7
9  D = 103

10 DECRYPTION THROUGH LOGARITHMS

11 Enter a secret message to encrypt: 24
12 Secret message = 24
13 Encrypted message = 106
14 D = More than a million iterations to calculate

15 HOW MANY ITERATIONS?

16 Iterations using semi-prime factorization = 141
17 Iterations using logarithms = 2.82597349859519e+206
```

Figure 2. Program yield

The creation of the Python code described on the previous pages (Figure 1) resulted in a program that, when run, yielded the above result (Figure 2). Lines 1 – 9 print each of the values necessary to calculate the decryption key, D, through semi-prime factorization. In line 2, the program asks for a semi-prime number input. If a number that is not semi-prime is inputted, as in Figure 2 when 1500 was inputted, the program asks for another semi-prime value instead, displaying the message "Your prior input was not a semi-prime number.

Enter a semi-prime number:" until a semi-prime value has been given. When a semi-prime value is given, this number is factored into its constituent factors, P and Q, which are then printed. φN is then calculated by plugging P and Q into the expression (P – 1)*(Q – 1) and subsequently printed. Once a public key value E is generated and printed, D is solved for in equation 2 and printed. Lines 10 – 14 print the necessary information to calculate D through logarithms. In line 11, the program asks for an integer message M, prints M in line 12, prints C (the value of M, encrypted) in line 13, and prints D in line 14 by solving for it in equation 4. However, in all the times that the program has been run, it has never been able to calculate D through logarithms due to the fact that it would take an incredible amount of iterations to do this. Instead, the program prints, "D = more than a million iterations to calculate" if it cannot calculate D through logarithms in under a million iterations. Finally, in lines 15 – 17, the program prints the number of iterations that the program needs to calculate D in either instance. On occasion, the program will experience an overflow error when it tries to calculate the iterations for decryption through logarithms because Python does not have the capacity for such huge calculations.

This project would be economically feasible to implement as it would not cost anything to download an app version of this program onto a mobile device once a graphical user interface has been created for all of the code that has been written.

5. Discussion and Conclusions

The research study undertaken was largely successful because it met all of the objectives defined in the introduction: to describe the mathematics of decryption techniques, to create a program capable of calculating the decryption key in RSA using two different methods and calculating the number of iterations required to accomplish this in either instance, and to explain the code written in the program and its roles in ultimately achieving the end result of decryption. The program has the capacity to calculate the decryption key through both semi-prime factorization and through the logarithmic method. That being said, the logarithmic method almost always takes too many iterations to feasibly calculate the decryption key because of the exponentiation operations involved. However, the program is still theoretically capable of calculating the decryption key through the logarithmic method. The program is also capable of calculating the number of iterations needed to calculate D through either method but will at times experience an overflow error if the number of iterations needed to calculate D through logarithmic techniques exceeds Python's programming capacity. If the code in this study can be replicated using more powerful and faster programs, it would be surmised that D would be calculated with more rapidity through the logarithmic technique and the program would not experience overflow errors. When such errors occur, exception handling can be used to catch the errors using Python's OverflowError. Such an implementation is provided in the appendix.

Additionally, there are many other techniques in existence to factor large semi-prime numbers, though this program only makes use of one of the most rudimentary such techniques. The quadratic sieve and the general number field sieve are two such integer factorization algorithms that far outstrip the technique presented in this research.[15]

Further, this program could have possible implementations in the future. Because it is capable of calculating the number of iterations needed to obtain the decryption key through either semi-prime factorization or logarithms, the program could potentially be used as a starting point to create an entirely new study in which the efficacy of either decryption method is compared when certain variables like the length of the semi-prime number N or the public exponent E are varied.

Acknowledgements

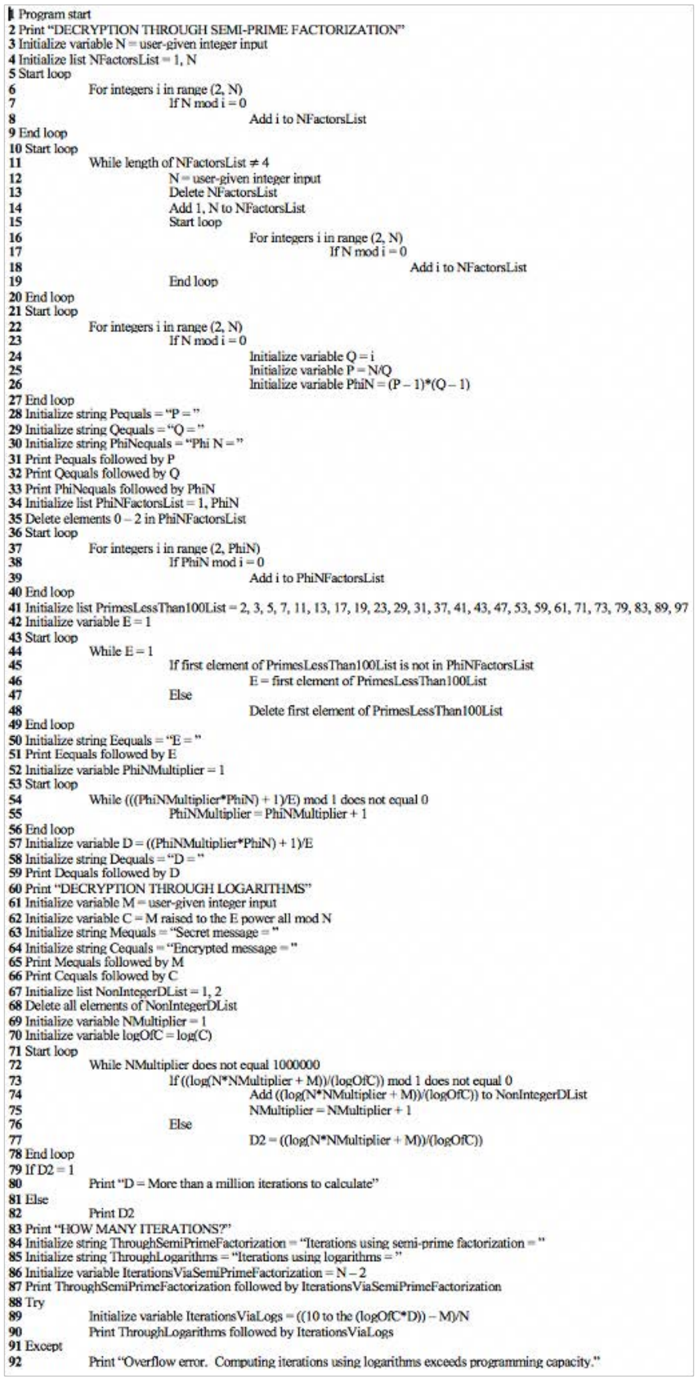
I would like to offer my thanks to Dr. Michal Kolpak and Mr. Quillian Haralson for inspiring this project, as well as to the myriad other friends, educators, and family who have given their encouragement and support in the creation of this and many other projects.

References

1. Luciano, D., Prichett, G. (1987). Cryptology: from Caesar ciphers to public-key cryptosystems. *The College Mathematics Journal, 18*(1), 2-17. https://doi.org/10.2307/2686311.
2. Deffs, H., Helmut, K. *Introduction to Cryptography, Principles and Applications 2nd ed.* (2007). Springer. 11-31.
3. Ellis, J. H. (1970). The possibility of secure non-secret digital encryption. UK Communications Electronics Security Group. www.cesg.gov.uk/site/publications/media/possnse.pdf. _________. (1999). The history of non-secret encryption. *Cryptologia* 23(3) 267–73. http://www.informaworld.com/10.1080/0161-119991887919.
4. Cocks, C. C. (1973). A Note on non-secret encryption. UK Communications Electronics Security Group. http://www.cesg.gov.uk/publications/media /notense.pdf.
5. Rivest, R. L., Shamir, A., Adleman, L. (1977). A method for obtaining digital signatures and public-key cryptosystems. Technical Memo Number MIT-LCS-TM-082. MIT. http://publications .csail.mit.edu/lcs/specpub.php?id=81.
6. Holden, J. (2017). *The Mathematics of Secrets: Cryptography from Caesar Ciphers to Digital Encryption*. Princeton University Press. 216.
7. Dence, J. B., Dence, T. P. (1999*). Elements of the Theory of Numbers*. Harcourt Academic Press. 156.
8. https://mathworld.wolfram.com/PrimeNumber.html. Retrieved March 1, 2021.
9. Wagstaff, S. (2013). *The Joy of Factoring*. American Mathematical Society. 195–202.
10. Rivest, R. L., Shamir, A., Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2).
11. Gardner, Martin. (1977). Mathematical games: A new kind of cipher that would take millions of years to break. *Scientific American* 237(2) 120–24. https://simson.net/ref/1977/Gardner_RSA.pdf.
12. Diffie, W., Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22 (6): 644–654. https://doi.org/10.1109/TIT.1976.1055638
13. Gidney, C., Ekerå, M. (2021). How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum 5*. (433). arXvix: 1905.09749v3. https://doi.org/10.22331/q-2021-04-15-433.
14. https://www.wolframalpha.com/
15. Pomerance, C. (1996). A tale of two sieves. *Notices of the American Mathematical Society*. 43 (12). 1473–1485.

Appendix

An implementation of the program in pseudocode is provided below in which exception handling is used to handle overflow errors.



```
Program start
Print "DECRYPTION THROUGH SEMI-PRIME FACTORIZATION"
Initialize variable N = user-given integer input
Initialize list NFactorsList = 1, N
Start loop
    For integers i in range (2, N)
        If N mod i = 0
            Add i to NFactorsList
End loop
Start loop
    While length of NFactorsList ≠ 4
        N = user-given integer input
        Delete NFactorsList
        Add 1, N to NFactorsList
        Start loop
            For integers i in range (2, N)
                If N mod i = 0
                    Add i to NFactorsList
        End loop
End loop
Start loop
    For integers i in range (2, N)
        If N mod i = 0
            Initialize variable Q = i
            Initialize variable P = N/Q
            Initialize variable PhiN = (P – 1)*(Q – 1)
End loop
Initialize string Pequals = "P = "
Initialize string Qequals = "Q = "
Initialize string PhiNequals = "Phi N = "
Print Pequals followed by P
Print Qequals followed by Q
Print PhiNequals followed by PhiN
Initialize list PhiNFactorsList = 1, PhiN
Delete elements 0 – 2 in PhiNFactorsList
Start loop
    For integers i in range (2, PhiN)
        If PhiN mod i = 0
            Add i to PhiNFactorsList
End loop
Initialize list PrimesLessThan100List = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 71, 73, 79, 83, 89, 97
Initialize variable E = 1
Start loop
    While E = 1
        If first element of PrimesLessThan100List is not in PhiNFactorsList
            E = first element of PrimesLessThan100List
        Else
            Delete first element of PrimesLessThan100List
End loop
Initialize string Eequals = "E = "
Print Eequals followed by E
Initialize variable PhiNMultiplier = 1
Start loop
    While (((PhiNMultiplier*PhiN) + 1)/E) mod 1 does not equal 0
        PhiNMultiplier = PhiNMultiplier + 1
End loop
Initialize variable D = ((PhiNMultiplier*PhiN) + 1)/E
Initialize string Dequals = "D = "
Print Dequals followed by D
Print "DECRYPTION THROUGH LOGARITHMS"
Initialize variable M = user-given integer input
Initialize variable C = M raised to the E power all mod N
Initialize string Mequals = "Secret message = "
Initialize string Cequals = "Encrypted message = "
Print Mequals followed by M
Print Cequals followed by C
Initialize list NonIntegerDList = 1, 2
Delete all elements of NonIntegerDList
Initialize variable NMultiplier = 1
Initialize variable logOfC = log(C)
Start loop
    While NMultiplier does not equal 1000000
        If ((log(N*NMultiplier + M))/(logOfC)) mod 1 does not equal 0
            Add ((log(N*NMultiplier + M))/(logOfC)) to NonIntegerDList
            NMultiplier = NMultiplier + 1
        Else
            D2 = ((log(N*NMultiplier + M))/(logOfC))
End loop
If D2 = 1
    Print "D = More than a million iterations to calculate"
Else
    Print D2
Print "HOW MANY ITERATIONS?"
Initialize string ThroughSemiPrimeFactorization = "Iterations using semi-prime factorization = "
Initialize string ThroughLogarithms = "Iterations using logarithms = "
Initialize variable IterationsViaSemiPrimeFactorization = N – 2
Print ThroughSemiPrimeFactorization followed by IterationsViaSemiPrimeFactorization
Try
    Initialize variable IterationsViaLogs = ((10 to the (logOfC*D)) – M)/N
    Print ThroughLogarithms followed by IterationsViaLogs
Except
    Print "Overflow error.  Computing iterations using logarithms exceeds programming capacity."
```