RSA Encryption Tool

An Investigatory Project Report

Submitted by

Nebhrajani, Aditya V.

The Orchid School
Baner Pune 411045

2020-2021

The Orchid School

Certificate

This is to certify that Aditya V. Nebhrajani of Class XII I1 has successfully completed the Informatics Practices Investigatory Project in the partial fulfillment of curriculum of Central Board of Secondary Education (CBSE) leading to award of Annual Examination of the year 2020-2021.

External Examiner

Teacher In-Charge
(Jayashree Samudra)

Unit Head
(Netre Kulkarni)

Principal
(Namrata Majhail)

Acknowledgements

No project can ever be completed without the graceful contribution of many, many people.

I would, first and foremost, like to express my gratitude to our IP teacher, Ms. Jayashree Samudra for her valuable advice and inputs which enhanced the quality of my work. This project could not have been completed successfully without her help. I would also like to thank Netre Di for her encouragement and support, and our respected principal Namrata Di for giving me an opportunity to work on such an intriguing topic.

This entire project was written on open source software, using open source programming and markup languages. I am hence deeply indebted to the open source community at large. I would like to separately acknowledge the GNU Emacs[1], LaTeX[2], Python[3] and the StackExchange[4] communities for their powerful – and open source – tools.

Lastly, I would like to thank my parents for their love and guidance without which nothing can be accomplished.

[1] The Emacs text editor/eLISP REPL: https://www.gnu.org/software/emacs/
[2] LaTeX document creation system: https://www.latex-project.org/
[3] Python programming language: https://www.python.org/
[4] StackExchange forums: https://stackexchange.com/

Contents

List of Figures

1 Introduction

Democratic governments the world over have a similar system of guaranteeing rights to people. However, one right which is still not guaranteed by most is the right to privacy. Prior to the advent of computer technology, privacy was easy to ensure: one had but to close the door or seal a letter. Things have changed since then: humans communicate more and more over the Internet. This makes our communications less private, since the medium of communication is not be secure.

In any case, there are tremendous vested interests in invading people's privacy, from showing more personalised ads to societal control and conditioning. After whistleblower Edward Snowden leaked the United States' National Security Agency's highly classified information, it is public domain knowledge that not only companies but also governments invade our privacy to their own ends.

It is difficult to overstate how morally incorrect and dangerous such a practice is. Snowden put the situation excellently:

> "Arguing that you don't care about privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say."

The irony of the situation is that most people are aware that companies such as Google, Microsoft and Facebook steal their data[5], but they do nothing about it. This is best explained through consumers preferring convenience over privacy. Google is easier to use than DuckDuckGo, and is better developed. To expect anybody other than a few highly privacy-conscious individuals to stop using Instagram and Windows 10 is not an idea solution. Instead, the convenience of encryption needs to be available to the general public in a user-friendly interface. Ideas like PGP have tried and fallen into obscurity, PAKE didn't gain traction[5], and people are trusting, or trying to trust, WhatsApp's 'end-to-end-encryption'. There is an opening for a system that lets people secure their privacy while also being accessible and easy to use.[4][6]

This project is an attempt to remedy this situation by building a framework for quick and easy to use encryption. It's written in Python, which makes it easy to port or include in other applications. The interactive CLI and Pandas database are merely for demonstration purposes. Ideally, this project should be used as a module as part of a larger communication application.

[5]Reference: Ask anyone.

2 Cryptography

2.1 In History

Since time immemorial, kings, queens, and armies have relied on the privacy of their communications to communicate battle plans to units spread over a large area. Other people reliant on privacy were secret lovers, those inciting rebellion, and even Mary Queen of Scots.

Privacy was ensured by somehow obscuring the original message using methods such as steganography or cryptography. Steganography is of less interest to us, as it involves physically hiding a message, for instance, by engraving it in a hidden place. The other method, cryptography, involved taking the original message, the 'plaintext', and converting it via an algorithm to an obscure mess of letters, the 'ciphertext'. The receiver then performed the reverse algorithm and retrieved the plaintext. Thus, even if the message were intercepted, it could not be read and its information exploited.[6]

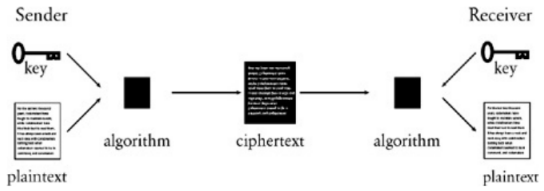

Figure 1: Principle of encryption

In cryptography, the security of the whole system depends not on the security of the communication medium but rather on the algorithm used. An example is the very simple Caesar cipher[2]:

> The action of a Caesar cipher is to replace each plaintext letter with a different one a fixed number of places down the alphabet. The cipher illustrated here uses a left shift of three, so that (for example) each occurrence of E in the plaintext becomes B in the ciphertext.

In this case, the 'key' would be the number of places to shift each letter by, or the 'shift'. The Caesar cipher, like many others of its time, was broken by frequency analysis: given a long enough sample of the ciphertext, a cryptanalyst could replace the most frequently occurring letter by the one used most in the English alphabet ('e'), or the second most used ('t'), and so one. By trail and error, the original message is soon revealed. In fact, the cipher of Mary Queen of Scots was broken by frequency analysis by a cryptanalyst called Thomas Phelippes. Once the cipher was broken, the Babington plot was revealed, and she was executed, with the cracked cipher being the central evidence in the case against Mary Queen of Scots.

Much later, during the Second World War, the Polish and British forces joined hands to crack the German Enigma cipher, which used an Enigma machine to encrypt the plaintext. The Polish designed machines to crack the Enigma code (called 'bombes' for the loud noises they made), which were further refined by Alan Turing, who insisted on spending time designing mechanical machines that could break the cipher rather than spend time cracking the ciphers by hand. This was one of the first instances of machines being used to encrypt plaintext, and with Turing, machines being used to break ciphers.

2.2 Modern Methods and Some Mathematics

The primary problems in cryptography are key distribution and the security of the ciphertext. While the security of the ciphertext can, in theory, be ensured by using complicated algorithms implemented

[6]Image credit: The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography, Simon Singh. OCLC: 59459928

by machines, the problem of secure key distribution remains. If the keys are distributed through an insecure communication medium, this renders the algorithm a mere computational hurdle at best.

2.2.1 Diffie-Hellman Key Exchange

To solve the problem of secure key-distribution, Diffie, Hellman, and Merkle created a method for generating keys even on an insecure communication medium. [3] The method is shown in the figure below.[7]

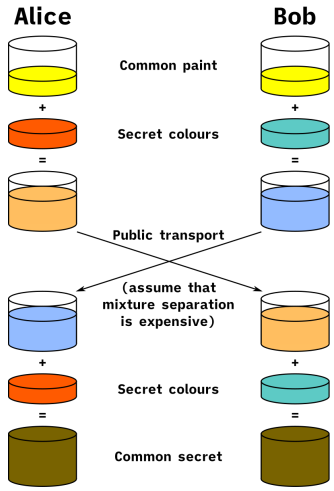

Figure 2: Diffie-Hellman key exchange

As evident from the figure, a man-in-the-middle, despite having knowledge of the starting color Alice and Bob agree on (yellow), and the intermediaries that they exchange (orange-tan and light blue), he cannot find the final key since Alice and Bob do not exchange information about the secret colors they use (blue and orange). In practice, of course, Alice and Bob use large numbers and the modulo function (modulus from clock math) to eventually get the same secret key, which is a large integer. This algorithm, proposed in 1976, is used to generate secret keys throughout the Internet. A simple generalisation of the principle used (using finite cyclic groups) is as follows (a cyclic group is generated by a single element, and any element in the group can be obtained by repeated group operations. A clock with an arithmetic of addition defined is a cyclic group of order 12.):

1. Alice and Bob agree on a finite cyclic group G (multiplicative) of order n and a generating element g in G. (g is assumed to be known by all attackers.)
2. Alice picks a random natural number a $(1 < a < n)$ and sends g^a to Bob.
3. Bob picks a random natural number b $(1 < b < n)$ and sends g^b to Alice.
4. Alice finds $(g^b)^a$.
5. Bob finds $(g^a)^b$.
6. Since $(g^b)^a = (g^a)^b = g^{ab}$, both Alice and Bob have the same group element, which serves as the secret key.

We note, however, that it is likely organisations with large budgets can crack Diffie-Hellman, especially ones that use keys of 1024-bit lengths or less. Moreover, other vulnerabilities have been found in Diffie-Hellman, as described in the paper *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*. [1] According to Adrian et al. around two-thirds of popular HTTPS sites allow

[7]Image credit: Diffie-Hellman key exchange - Wikipedia/Wikimedia Commons

TLS using Diffie-Hellman and 1024-bit-keys. Some even allow legacy 512-bit keys. Two-thirds of the HTTPS servers analyzed also use common groups for generation, meaning some primes are more 'popular' than others. These vulnerabilities were exploited by the paper and the Logjam attack was used to cryptanalyze Diffie-Hellman.

Diffie-Hellman inspired a much more powerful algorithm, the RSA asymmetric key encryption.

2.2.2 RSA Encryption

RSA encryption[7], named after Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman, uses the concept of each user having two keys: one public key and one private key. Assume these keys are pre-generated. Say Bob wants to send a message M to Alice. Alice's public key is a tuple of integers (n, e) and her private key is an integer (d). The public key is available to everyone, maybe Alice even keeps it in her Instagram bio.

1. Bob converts his message M into an integer m using a pre-decided scheme (specified by a standard such as PKCS#1).
2. Bob gets Alice's public key, (n, e) and computes

$$m^e \equiv c \pmod{n}$$

   He then sends Alice c over the communication medium.
3. Alice recovers m from c by using her private key d, thus:

$$c^d \equiv (m^e)^d \equiv m \pmod{n}$$

4. Alice then reverses the message-to-integer scheme and gets M from m.

This method results in only two pieces of information passing through the communication channel: Alice's public key, and the encrypted message, as shown in figure.[8]

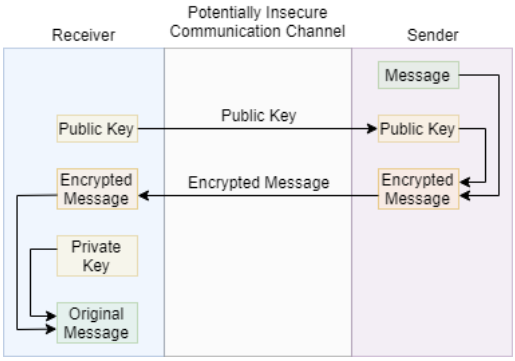

Figure 3: RSA encryption

Key Generation Keys are generated using two large primes, and this is what offers RSA its high level of security once encrypted. The exact procedure is as follows:

1. Choose two distinct primes p and q, at random. These are kept secret.
2. Compute $n = pq$. n is used as the modulus, and its length expressed in bits is defined as the "key length". n is part of the public key tuple.

[8]Image credit: Jin Kyu Lim, Medium https://medium.com/@jinkyulim96/algorithms-explained-rsa-encryption-9a37083aaa62

3. Compute $\lambda(n)$, where λ is Carmichael's totient function. Since $n = pq$,

$$\lambda(n) = \text{LCM}(p-1, q-1)$$

   $\lambda(n)$ is kept secret.

4. Choose integer e such that e and $\lambda(n)$ are coprimes. e is usually set to $2^{16}+1 = 65537$, since it has a low Hamming weight (its binary representation has a lot of zeros). e is called the public exponent.

5. Find d by solving

$$d \cdot e \equiv 1 \pmod{\lambda(n)}$$

   that is, taking the modular multiplicative inverse of $e \pmod{\lambda(n)}$. d is the private key and is kept secret.

We hence have the public key tuple (n, e) and the private key d. The above operations are usually carried out using various optimized algorithms, which will be discussed while implementing the Python code.

2.2.3 RSA Proof and Example

Fermat's Little Theorem Fermat's Little Theorem states that $a^{p-1} \equiv 1 \pmod{p}$ for any integer a and prime p not dividing a.

RSA Proof We are to show,

$$(m^e)^d \equiv m \pmod{pq} \forall m$$

where p and q are distinct primes, and e and d are positive integers satisfying

$$ed \equiv 1 \pmod{\lambda(pq)}$$

Since $\lambda(n) = \text{LCM}(p-1, q-1)$, by construction, divisible by both $p-1$ and $q-1$:

$$ed - 1 = h(p-1) = k(q-1)$$

for some natural numbers h and k.

From the Chinese Remainder Theorem,

> To check whether two numbers, like m^{ed} and m, are congruent mod pq, it suffices (and in fact is equivalent) to check that they are congruent mod p and mod q separately.

To show $m^{ed} \equiv m \pmod{p}$, we consider two cases:

1. If $m \equiv 0 \pmod{p}$, m is a multiple of p (multiplicative cyclic group). Thus, m^{ed} is a multiple of p. Thus, $m^{ed} \equiv 0 \equiv m \pmod{p}$.

2. If $m \not\equiv 0 \pmod{p}$,

$$m^{ed} = m^{ed-1}m = m^{h(p-1)}m \equiv 1^h m \equiv m \pmod{p}$$

   where $m^{h(p-1)} \equiv 1^h m \pmod{p}$ by Fermat's Little Theorem.

We proceed similarly for q, which completes the proof.

RSA Example The following is a simple proof of concept of RSA.

1. Let $p = 11$, $q = 3$.
2. $n = 11 \times 3 = 33$
3. $\phi = (p-1)(q-1) = 20$. Note that we're not taking the LCM. [9]
4. Choose $e = 3$.
5. Find d such that $ed \equiv 1 \pmod{\phi}$. By trial and error, $d = 7$.
6. Hence, the public key is $(33, 3)$ and the private key is 7.
7. Let us encrypt $m = 7$.
$$c \equiv m^e \equiv 7^3 \equiv 343 \pmod{33} = 13$$
The ciphertext is hence 13.
8. Now, decrypting,
$$m \equiv c^d \equiv 13^7 \pmod{33} = 7$$
We hence retrieve the original message.

[9] We are using Euler's rather than Carmichael's totient function. Any d satisfying the former satisfies the latter, however, sometimes, Euler's yields a d larger than required. In this case, the distinction is irrelevant. This doesn't matter for implements that use the Chinese Remainder Theorem, as d is not used directly.

3 Python Implementation

We now focus on building a complete RSA system in Python (3.6) that is capable of generating keys, maintaining a database of users and their public/private keys, and encrypting and decrypting ASCII text files of arbitrary length in reasonable time.

Wherever further mathematics is required for implementing a certain primitive or algorithm, it is discussed prior its implementation in code.

3.1 Installation

The source code for this project is available on GitHub. You should either download it from `https://github.com/nebhrajani-a/rsa-py` or clone it into a new directory.

```
$ git clone https://github.com/nebhrajani-a/rsa-py ./rsa-py
```

3.1.1 Dependencies

The major non-standard dependancy is `progress`, which draws the progress bars during encryptions and decryptions.

```
$ pip install progress
```

Other dependancies are Python 3 (note that this program will **not** run on Python 2), math (for the ceiling function), Secrets (secure random number generation), getpass (secure password entry) and Pandas (database management).

3.1.2 Compatibility

This program is written to be platform independent, using Python implementations everywhere rather than (possibly simpler) alternatives like command line calls. It's been tested on a machine running Linux Mint 19.3 (XFCE 64-bit) and in two Microsoft Windows virtual machines (7 and 10), and behaves as expected. If you think you are experiencing bugs, please write an email to aditya.v.nebhrajani@gmail.com.

3.1.3 Executing

To use the program, `cd` to `/src` and execute

```
$ python main.py
```

Note: If you use Windows, Python may be aliased to `py` instead. Make sure `python` runs Python 3 and not Python 2. On some Linux systems Python 3 needs to be called using `python3`.

3.2 Key Generation

To generate keys, we must have: a large prime generator for p and q, and math functions for taking the LCM and modular multiplicative inverse of two numbers.

3.2.1 Large Prime Generation

Large prime generation is a mathematics problem as there is no evident pattern to finding primes. $\pi(n)$ is the prime counting function for primes $\leq n$. According to the prime number theorem[10]

$$\pi(n) \sim \frac{n}{\ln n}$$

or the probability of a random n being prime is $1/\ln n$. For instance, the probability of a random 2048 bit number being prime is $1/\ln 2^{2048} \approx 1/1419$.

[10] Which describes the asymptotic distribution of primes in the natural numbers. In fact, it is more explicitly stated as $\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1$.

Immediate Improvements We can improve the previously calculated probability by noting that no prime other than two is a multiple of two, meaning we have to check only half of the otherwise required 1419 numbers, so ~ 710 numbers must be checked.

Now, to check if a number is prime, we need to divide it by every divisor d such that $1 < d < n$. There's another improvement here: we only have to check till $\sqrt{n}$ since if n is composite and equal to pq, either $p \leq \sqrt{n}$ or $q \leq \sqrt{n}$.

Or, in Python:

```
def prime_generator(n):
    if n % 2 == 0:
        return False
    for i in range(3, sqrt(n), 2):
        if n % d == 0:
            return False
    return True
```

This function certainly works, but has a time complexity $\mathcal{O}(\sqrt{n})$. This will need to be faster for generating primes as large as 2048 bits. We hence turn to the probability based Miller-Rabin primality test.

Non-Trivial Square Roots We first define trivial and non-trivial square roots. The trivial square roots of $1 \mod p$ are 1 and -1, as they always return 1 on being squared modulo p, where p is a prime satisfying $p > 2$.

We define a non-trivial square root of $1 \mod p$ as any root other than 1 and -1. We prove now that if such a root exists, p cannot be prime (a special case of the result that, in a field, a polynomial has as many zeros as its degree).

$$x^2 \equiv 1 \pmod{p}$$
$$(x-1)(x+1) \equiv 0 \pmod{p}$$

Which means $x \equiv 1$ or $-1 \pmod{p}$.

For example, consider $3^2 = 9 \equiv 1 \pmod{8}$, meaning 3 is a non-trivial square root of $1 \mod 8$. Clearly, 8 is composite.

Miller-Rabin Primality Test The Miller-Rabin test relies on finding nontrivial square roots of $1 \mod n$.

Let n be a prime and $n > 2$. $n-1$ must be even, and we write it as $2^s \times d$, where s and d are positive integers and d is odd. For each a in $(Z/nZ)\times$ (in range $[2, n-2]$), either

$$a^d \equiv 1 \pmod{n}$$

or

$$a^{2^r \cdot d} \equiv -1 \pmod{n}$$

for some $0 \leq r \leq s-1$. To prove the claim, check that by Fermat's Little Theorem, $a^{n-1} \equiv 1 \pmod{n}$. Hence, if we repeatedly take the square roots of a^{n-1}, we will get either 1 or -1.

The Miller-Rabin test is the contrapositive of the above, that if we can find an a such that both of the above hold false, n must be composite. a is called a witness to the fact that n is non-prime. If some a does not satisfy the negation of both the above, a is called a strong liar, and n is a strong *probable* prime for base a (that is, it might yet be composite and still hold the contrapositive).

Every odd composite number n has many witnesses a, however, if we're unlucky, a will be a strong liar for n and we'll categorize it as prime. There's no known way of generating an a which isn't a liar. We instead make the test probability based: we choose a random non-zero a and test it. Since it might be a strong liar, we repeat the test for a different value of a. The greater the number of iterations we check, the higher is the probability of generating a prime n.

The time complexity of the Miller-Rabin test is $\mathcal{O}(k \log^3 n)$, which means it is a polynomial time algorithm, a vast improvement over the simple method discussed earlier.

Final Implementation We first generate a random number n using Python's `secrets` module. [11] We first implement a low level checker: we divide the randomly generated n by every element in a static list of the first 2000 primes. This eliminates obvious composite numbers early on, speeding up our work. Also, this prevents some strong liars in Miller-Rabin. This is implemented as follows:

```
import first_primes as fp
import secrets

def gen_random(l):
    '''Generate a random number l bits long.'''
    randgenerator = secrets.SystemRandom()
    return randgenerator.randrange(2**(l-1)+1, 2**l - 1)
```

Note that the random number generated is in the range $[2^{l-1}+1, 2^l-1]$ to ensure the correct bit-length and to ensure that no extra even numbers are checked. Also note that `first_primes` is just a file with one list declaration. Files can be imported as modules in Python 3 without an `__init.py__` file.

```
def low_level_checker(l):
    '''Check that the random number isn't divisible by the first few primes.'''
    while True:
        x = gen_random(l)
        for divisor in first_primes:
            if x % divisor == 0 and divisor**2 <= x:
                break
        else: return x
```

n is then divided by every number in the first 2000 primes which is less than $\sqrt{n}$. Miller Rabin is then implemented exactly as described mathematically:

```
def miller_rabin_checker(mrc):
    '''Run 40 iterations of the Miller-Rabin Primality Test.'''
    randgenerator = secrets.SystemRandom()
    max_divisions_by_two = 0
    y = mrc-1
    while y % 2 == 0:
        y >>= 1
        max_divisions_by_two += 1
    assert(2**max_divisions_by_two * y == mrc-1)

    def trial_composite(round_tester):
        if pow(round_tester, y, mrc) == 1:
            return False
        for i in range(max_divisions_by_two):
            if pow(round_tester, 2**i * y, mrc) == mrc-1:
                return False
        return True
    number_of_rabin_trials = 40
    for i in range(number_of_rabin_trials):
        round_tester = randgenerator.randrange(2, mrc)
        if trial_composite(round_tester):
            return False
    return True
```

[11] The `random` module is not suitable for cryptographic applications. Secrets instead calls the system's most secure source of randomness, such as /dev/random on Linux.

Note that the third argument of Python's `pow()` function takes an optional argument for the modulus.

Now that there is an implementation for both a low level and a high level checker (Miller-Rabin), the last requirement is a driver function to actually generate the prime.

```
def driver(l):
    while True:
        prime_candidate = low_level_checker(l)
        if not miller_rabin_checker(prime_candidate):
            continue
        else:
            return prime_candidate
```

This driver function keeps generating random numbers until a number passes both the low level checker and 40 iterations of Miller-Rabin[12], which means it has an acceptably high probability of being prime.

3.2.2 Mathematics Functions

Now that there is a framework for generating the random primes p and q, a framework for taking the LCM (lowest common multiple) and the modular multiplicative inverse is required.

LCM For any integers a and b, $ab = \text{LCM}(a,b) \times \text{GCD}(a,b)$. Calculating the GCD (greatest common divisor or highest common factor) of two numbers is a simple recursive algorithm called the (Standard) Euclidean Algorithm. It involves a succession of Euclidean divisions (division with remainder).

First, take the larger of a and b, the two numbers whose GCD we are to find. Divide the larger (say a), by the smaller (say b). Then, divide b by the remainder of this division. Continue to take successive remainders as divisors for the next step and divisors for the former as dividends for the latter. Continue this process until a remainder of zero is obtained. The divisor for this operation is the GCD. The proof is trivial and well-known, so we skip to the Python implementation.

```
import sys
sys.setrecursionlimit(10**6)

def gcd(a,b):
    if a == 0:
        return b
    return gcd(b % a, a)
```

Note that the system recursion limit is increased to 10^6 to prevent Python's stack overflow inhibitor from kicking in for large numbers. While an iterative GCD function could be used, recursive functions are funner and simpler to write. In addition, Python uses enough system resources that this increase is not too resource-intensive. We do note, however, that Python is not a functional language and an iterative implementation is a better choice for real world applications.

The LCM is now trivial to calculate by simple division:

```
def lcm(a,b):
    return (a*b) // gcd(a,b)
```

'Floor' division is used as opposed to 'true' division to prevent integer to float conversion, which results in rounding and loss of LSB information.

[12] 40 iterations were chosen from the reasoning in `https://stackoverflow.com/a/6330138`

Modular Multiplicative Inverse The modular multiplicative inverse is calculated using the Extended Euclidean Algorithm, which is similar to the Standard Euclidean Algorithm used for calculating the GCD, except that the we use two sequences rather than one. In the Standard Euclidean Algorithm the first divisors are represented by $r_0 = a$ and $r_1 = b$ with the sequence following through $r_{i+1} = r_{i-1} - q_i r_i$ till $r_{k+1} = 0$ and r_k is the GCD.

In the Extended Euclidean Algorithm, the two sequences start with three values each, the first being $r_0 = a$, $s_0 = 1$, $t_0 = 0$ and the other $r_1 = b$, $s_1 = 0$, $t_1 = 1$. The sequence is followed through $r_{i+1} = r_{i-1} - q_i r_i$, $s_{i+1} = s_{i-1} - q_i s_i$, $t_{i+1} = t_{i-1} - q_i t_i$, until we find $r_{k+1} = 0$ and r_k as GCD.

However, s_k and t_k are the Bézout coefficients of a and b[13]. Bézout's identity is defined as:[14]

> Bézout's identity — Let a and b be integers with greatest common divisor d. Then, there exist integers x and y such that $ax + by = d$. More generally, the integers of the form $ax + by$ are exactly the multiples of d.

The Bézout's coefficients can be used to efficiently calculate the modular inverse, thus: $ax + by = \text{GCD}(a, b) = 1$ or, $ax - 1 = (-y)b$ or,

$$ax \equiv 1 \pmod{b}$$

which completes our task. This algorithm runs in $\mathcal{O}(\log b^2)$ time. The Python implementation of the above two algorithms is as follows:

```
def eea(a, b):
    '''Extended Euclidean algorithm'''
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = eea(b % a, a)
        return (g, x - (b // a) * y, y)
```

which is very similar to the recursive GCD algorithm, albeit with different sequences.

```
def modinv(a, m):
    '''Find modular multiplicative inverse using EEA'''
    g, x, y = eea(a, m)
    if g != 1:
        raise Exception('Modular inverse does not exist.')
    else:
        return x % m
```

which simply calls `eea()` and takes the modulus.

3.2.3 Putting It Together

We are now set to generate keys, having all the required functions to do so. This is done as outlined in Section 2.2.2. The code is self-explanatory.

```
import prime_generator as pg
import math_functions as mf

def gen_keys(bits):
    p = pg.driver(bits//2)
    q = pg.driver(bits//2)
    n = p*q
```

[13] The proof for this claim is not within the scope of this paper.

[14] From Wikipedia, `https://en.wikipedia.org/wiki/B%C3%A9zout%27s_identity`

```
    lambda = mf.lcm((p-1), (q-1))
    e = 65537
    d = mf.modinv(e, lambda)
    return [p, q, n, e, d, bits]
```

`gen_keys()` takes an argument for the bit-length of the resulting public key n, and returns a list of the prime factors of n, n, the public and private keys, and the bit-length of the public key. This data will be added to the user database eventually.

3.3 Primitive RSA Encryption and Decryption

Now that the keys have been generated, encryption and decryption is merely an exponentiation in modulo n as described in Section 2.2.2. The functions can hence be written as:

```
def enc(m, e, n):
    c = pow(m, e, n)
    return c
```

and,

```
def dec(c, d, n):
    m = pow(c, d, n)
    return m
```

The current implementation can be improved by using the Chinese Remainder Algorithm to speed up the decryption process.

3.3.1 Chinese Remainder Algorithm for Fast Decryption

The Chinese Remainder Algorithm is a generalisation of the method used to solve problems such as:

> There are certain things whose number is unknown. If we count them by threes, we have two left over; by fives, we have three left over; and by sevens, two are left over. How many things are there?

We use the following optimisation to decrypt, based on the general Chinese Remainder Theorem. The following values are precomputed and stored as part of the private key:

1. p and q: the primes from generation.
2. $d_P = d \pmod{p-1}$
3. $d_Q = d \pmod{p-1}$
4. $q_{inv} = q^{-1} \pmod{p}$

We then use these four values to calculate the $c^d \pmod{pq}$ thus:

1. $m_1 = c^{d_P} \pmod{p}$
2. $m_2 = c^{d_Q} \pmod{q}$
3. If $m_1 > m_2$:
$$h = q_{inv}(m_1 - m_2)$$
4. Else:
$$h = q_{inv}[(m_1 + \left\lceil \frac{q}{p} \right\rceil p) - m_2] \pmod{p}$$
5. $m = m_2 + hq \pmod{pq}$

This algorithm derives its efficiency from calculating two small modular exponentiations rather than one large one. The Python implementation is as follows:

```
def cra(c, p, q, d_p, d_q, q_inv):
    '''Chinese Remainder Algorithm'''
    m_1 = pow(c, d_p, p)
    m_2 = pow(c, d_q, q)
    if m_1 >= m_2:
        h = q_inv*(m_1-m_2) % p
    else:
        h = q_inv*((m_1 + (ceil(q/p))*p) - m_2) % p
    return pow((m_2 + h*q), 1, p*q)
```

3.4 ASCII to Integer via Octet Streams and Back

3.4.1 PT2OS

3.4.2 OS2IP

3.4.3 I2OSP

3.4.4 OS2PT

3.4.5 Putting It Together

3.5 File Chunking

3.6 User Data Pandas DataFrame

3.7 Interaction Functions

3.8 Putting It All Together

4 Results

5 Author's Note

5.1 Tools

5.2 This Paper

This paper is written as a submission to the Central Board of Secondary Education (India) as a Informatics Practices Investigatory Project for the year 2020-21.
It was noticed by the author of this paper that generally, investigatory projects, which are an amazing opportunity to showcase research and academia-related talent, typically end up being plagiarised experiments from textbooks or the World Wide Web. In fact, the top ten results of an Internet search for this very project: 'determination of caffeine content in tea samples', are all nearly identical papers, with the same content and errors. I hence perceive that on a general basis, students' participation is lacking in investigatory projects.
This paper is an attempt to fully utilize the opportunity gracefully granted to me by CBSE and The Orchid School, and to raise the bar for all students. It is my hope that this will inspire new and fruitful research at the secondary level of education.
It has been my attempt to write this paper in the same way as a scholarly article insofar as I could. If there are any errors or discrepancies, they remain my own.
I thank you for taking the time to go through this paper, and I hope you enjoyed reading it as much as I enjoyed writing it.

6 Source Code

References

[1] David Adrian et al. “Imperfect forward secrecy: How Diffie-Hellman fails in practice”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 5–17.

[2] *Caesar cipher - Wikipedia*. URL: `https://en.wikipedia.org/wiki/Caesar_cipher`.

[3] W. Diffie and M. Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[4] *HackerNews Discussion on Encryption Tools*. URL: `https://news.ycombinator.com/item?id=19173326`.

[5] *Let's talk about PAKE*. URL: `https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/`.

[6] *Modern Alternatives to PGP*. URL: `https://blog.gtank.cc/modern-alternatives-to-pgp/`.

[7] Ronald L Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.