naiveRSA

RSA encryption and decryption tool accelerated by CRT.

Overview

Encryption Level

In the tool, we provide four different encryption levels: `RSA-768`, `RSA-1024`, `RSA-2048` and `RSA-4096`, which means you can generate keys with `N` of these length(bits length).

Efficiency

In this part, we will show the results of the time consumed by our tool for generating keys, encrypting, decrypting, and decrypting accelerated by CRT.

Test Enviroment

- CPU: 1.4 GHz Four Core Intel Core i5
- OS: macOS Monerey
- `Python 3.8`

Generate Key

For each encryption level, we conducted about 8 tests here. Although the uneven distribution of prime numbers will lead to a large difference in generation time, it also has a certain reference value.

Level	1	2	3	4	5	6	7	8
`RSA-768`	0.0557	0.134	0.08	0.1671	0.067	0.1143	0.0509	0.0604
`RSA-1024`	0.2103	0.2498	0.1179	0.3339	0.2007	0.7944	0.1952	0.3148
`RSA-2048`	1.3014	2.7838	1.6395	5.2259	1.7065	2.2202	2.2947	0.8495
`RSA-4096`	28.493	13.7593	27.1998	7.4041	15.4106	53.4131	8.0431	22.8291

Encrypt and Decrypt

In order to compare the two decryption speeds more clearly and reduce the interference of other factors, we used 8 different keys that complies with RSA-2048 for testing. The information used for encryption contains a total of 300 Chinese and English characters of various types

Operation	1	2	3	4	5	6	7	8
En	0.546	0.5311	0.5231	0.5476	0.548	0.5517	0.5545	0.5439
De	0.5891	0.5483	0.5347	0.5566	0.5622	0.5535	0.5383	0.5738
De_CRT	0.1923	0.1887	0.1956	0.1923	0.192	0.1828	0.1843	0.1906

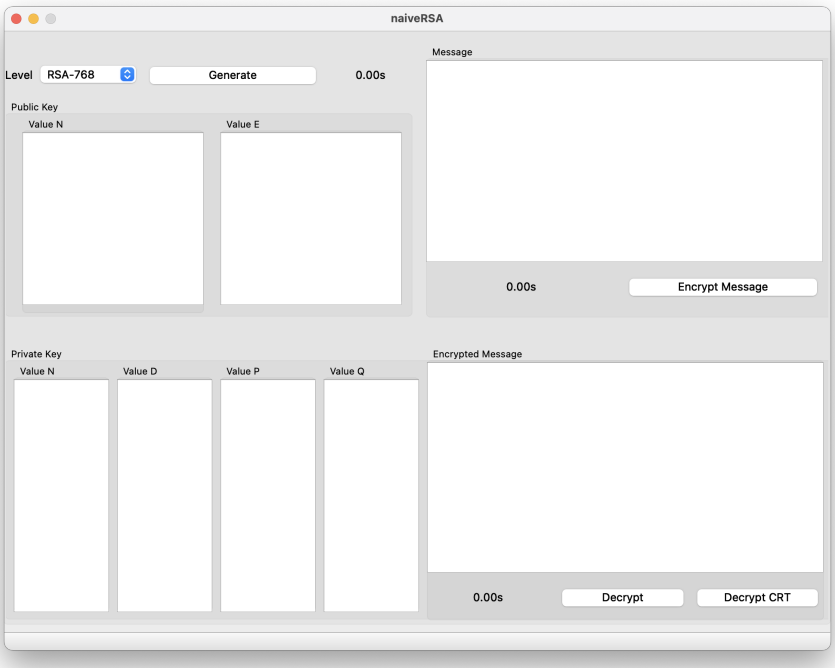
Usage

Quick Start

Execute following command at `./src` to start the tool:

```
python -m pip install -r ./requirements.txt
python main.py
```

Then you can get interface as following:



1. The left part is the key generation part. You can select the encryption level at left-top corner. And if you click the button `Generate`, the process generating key

begins. When the process finish, the key(public key: `<N, E>`, private key: `<N, D>` and `<P, Q>` for crt acceleration) will display on the interface, including `N`, `E`, `D`, `P`, `Q`.
2. The right part is the encryption and decryption part. Input your message and click button `Encrypt Message` to encrypt the message. The ciphertext and encryption time will appeare.
3. When you click button `Decrypt` or `Decrypt CRT`, the ciphertxt will be decrypted and show the result at `Message`.

Algorithm Implementation

Big Integer Operation

In addition to the `Python` version, we also used `C++` to implement it, but in the end it was **discarded** due to the performance problems of **division and modulus**. You can find it here.

We implement a class `BigInteger`, which contains the operations of unsigned big integer, including `+`, `-`, `*`, `/`, `%`, `pow()`, `abs()`, `==`, `!=`, `>`, `>=`, `<`, `<=`.

In order to improve the calculation speed, we used the following methods:

1. 2^{16} base

   When implementing large number operations, a very important basic idea is to reduce the low-efficiency algorithms implemented by yourself as much as possible, and use the language's own operations more. And increasing the base of large numbers is one of the methods. In order to prevent overflow, the intermediate result of all operations should not exceed the maximum value of unsigned long long, which is 2^64-1. So the theoretical maximum base is 2^32, here we use unsigned int to store each bit, so we use 2^16 base。

2. Fast power and fast power modulus

   In the Miller-Rabin primality test, there are many power modulus operations, which can be the key to improve the speed of prime number determination. Using fast exponentiation can reduce the complexity of exponentiation from $O(n)$ to $O(log\ n)$.

3. Misplaced subtraction, try quotient and binary search

   As we mentioned earlier, accelerating the Miller-Rabin primality test is the key to improving the key generation speed. In addition to using fast exponentiation to take the modulus, it is also important to increase the speed of modulo/division operations. According to vertical division, we achieve division through subtraction. In decimal, each round of subtraction can be performed at most ten times, but since

we are using 2^{32} base, each round of subtraction will increase to 2^{32} at most, which will bring great computational burden. . In order to reduce the number of subtraction calculations, we need to quickly find the final number of subtractions to be performed in each round, and use multiplication to greatly reduce the number of subtractions. When searching, we use binary search to reduce the complexity from $O(n)$ to $O(log\ n)$.

Big Prime Generation

The basic idea of large prime number generation is to randomly generate an odd number with a fixed length. If it can pass the Miller-Rabin primality test, it is considered a prime number.

Therefore, improving the test speed is the key to quickly generating large prime numbers. Some important methods are introduced in the section of Big Integer Operations. In the specific implementation of primality test, judging composite numbers instead of prime numbers will reduce the number of loop many times.

```
def is_composite(a, m, k, num):
    if fast_pow(int(a), int(m), int(num)) == 1:
        return False # not True
    for j in range(k):
        if fast_pow(int(a), int(m * (2 ** j)), int(num)) == num - 1:
            return False # not continue and set flag
    return True # not False
```

Encryption and Decryption

In order to support plaintext of any length when encrypting and decrypting, we encrypt the plaintext in segments, and add tags at the beginning and end of each ciphertext to ensure that decryption is feasible.

```
<ciphertext>10ff2895af1c7d5f7e75b00b54b51adfd7b75276bfac5
   3a93b6683ba824035dbd64c5ae920e0e833aa16676493bbd6f6b92
   53c354e739b3e5d84971e0d2af8c6d0fc797f90ad17b</ciphertext>
```

Multi Threads

There are many places in the generation of large prime numbers that can be accelerated by multi-threading:

- Two threads generate prime numbers respectively, and continue to run down when both of them have completed their tasks.

- Multiple threads each perform a round of Miller-Rabin primality test. If you want to go through 5 tests, five threads are used to perform one test. When all threads pass the test or one of the threads fails the test, the test is completed
- When looking for prime numbers, multiple threads can be used to search backwards from different starting positions. Once a thread find one, the search complete and other threads come to the end.

In fact, when we use python to achieve multi-threading, the speed has not been improved, this is because there is a global interpretation lock (GIL) in Cpython. When the interpreter interprets and executes Python code, it must first obtain it. This means that multithreaded code is actually serial.

Some Thoughts and Suggestions

In the specific implementation, there are actually many content that are not introduced in the course. Especially when implementing large integer operations, how to optimize each operation requires a lot of information and many attempts. The final change from `C++` to `Python` is also because the operation speed of division/modulus cannot meet the requirements. This is the most complicated part of the whole experiment. The remaining Miller-Rabin primality test and rsa are very easy to implement. I hope to get some necessary guidance in the almost irrelevant parts of these courses in the future.