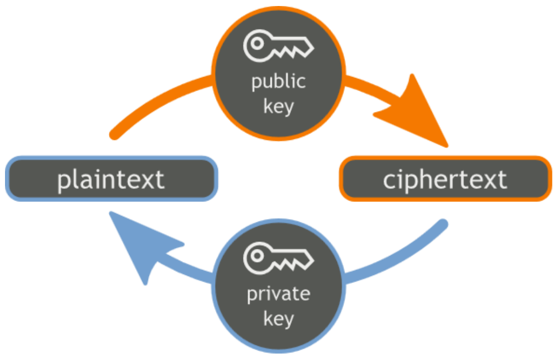
RSA SecureX: Extended-Precision Cryptography Toolkit

Problem Definition

The goal of RSA SecureX is to implement the RSA public-key cryptosystem using an extended-precision arithmetic data type capable of handling huge integers (hundreds of digits). Since native C types (16/32-bit int) are insufficient for the required security, you will first design, implement, and analyze a custom “big integer” library that supports addition, subtraction, multiplication, division, and modular operations. This library is then used to develop a client program that encrypts plaintext messages and decrypts ciphertext using RSA. RSA is an asymmetric cryptosystem in which two mathematically linked keys are generated: a public key for encrypting messages and a private key for decrypting them. Only the owner of the private key can reverse the encryption, ensuring the confidentiality of the messages.

Cryptosystem mean encoding and decoding confidential messages.

RSA is a public key cryptosystem algorithm. Public key cryptosystem requires two separate keys, one of which is secret (or private) and one of which is public. Although different, the two parts of this key pair are mathematically linked. The public key is used to encrypt plaintext; whereas the private key is used to decrypt cipher text. Only the one having the private key can encrypt the cipher text and get the original message.



How RSA Works?

Alice wants to send message M to Bob using RSA. How will this happen?

RSA involves three integer parameters **d, e,** and **n** that satisfy certain mathematical properties. The private key (**d, n**) is known only by Bob, while the public key (**e, n**) is published on the Internet. If Alice wants to send Bob a message (e.g., her credit card number) she encodes her integer **M** that is between 0 and **n**-1. Then she computes:

$$\mathbf{E(M) = M^{e} \bmod n}$$

and sends the integer **E(M)** to Bob. When Bob receives the encrypted communication **E(M)**, he decrypts it by computing:

$$\mathbf{M = E(M)^d \bmod n.}$$

Example:
e = 7, d = 2563, n = 3713 and the message M is 2003

Alice computes:

> $E(M) = 2003^7 \bmod 3713 = 129{,}350{,}063{,}142{,}700{,}422{,}208{,}187 \bmod 3713 = 746.$

Bon receives 746 so he can compute the original message as follows:

> $M = 746^{2563} \bmod 3713 = 2003.$

Goal and Suggestions

Your goal is to implement the BigInteger data type and use it in RSA public-key cryptosystem so you can be able to encrypt and decrypt large integer **M**.

The RSA cryptosystem is easily broken if the private key **d** or the modulus **n** are too small (e.g., 32 bit integers). The built-in C types **`int`** and **`long`** can typically handle only 16, 32 or 64 bit integers. Your main challenge is to design, implement, and analyze a BIG INTEGER that can manipulate large (nonnegative) integers. Your data type will support the following operations: addition, subtraction, multiplication, division, and odd/even checking.

1- The first and most critical step is to choose an appropriate data structure to represent N-digit big integers. A natural approach is to store each digit as an element in an array.

2- You need to implement this BigInteger class along with its basic functions: addition, subtraction, multiplication, division, and odd/even checking. Remember that you're not working with **`int`** or **`long`**

   Efficient Multiplication. Implement the efficient integer multiplication (less than N^2). It should take two big integers as input and return a third big integer that is the product of the two. Observe that if **a** and **b** are N-digit integers, the product will have at most 2N digits.

   Division. Integer division is the most complicated of the arithmetic functions. Here is one of the simplest algorithms to compute the quotient **q = a / b** and the remainder **r = a mod b**, where **a** and **b** are big integers, with **b** not equal to 0.

This algorithm requires the least amount of code, but its correctness may not be immediately apparent.

```
div(a, b)
{
    if (a < b)
        return (0, a)
    (q, r) = div(a, 2b)
    q = 2q
    if (r < b)
        return (q, r)
    else
        return (q + 1, r - b)
}
```

1. After that you'll design and implement an **efficient** algorithm for the RSA encryption and decryption functions as explained before.
2. Analyze your code

Project Requirements

1. Implement the BigInteger Class
    1. Add
    2. Sub
    3. Mul, in **efficient** way
    4. Div
    5. Odd/even checking
2. Implement RSA Encryption algorithm in **efficient** way.
3. Implement RSA Description algorithm.

Input

File containing **number of test cases** in the first line followed by each test case that is represented in 4 lines as follows:

1. N
2. e or d
3. M or E(M)
4. 0 or 1 (0-> encrypt, 1-> decrypt)

Output

1. Save the result of each encryption or decryption process in a single file each result on a separate line.
2. Calculate the execution time encryption/decryption

How to calculate execution time?

To calculate time of certain piece of code:

1. Get the system time before the code
2. Get the system time after the code
3. Subtract both of them to get the time of your code

To get system time in milliseconds using C#, you can use `System.Environment.TickCount`

Test Cases

- Will be announced soon with two versions (sample and complete).

Deliverables

Implementation (60%)

1. `BigInteger` Class with its functions
2. RSA Encryption
3. RSA Decryption

Document (40%)

1. Source code of the `BigInteger` and the RSA algorithm
2. Detailed analysis of it
3. Execution time of "Full Test" Cases only

BONUSES

Level 1:

Adjust RSA to work with strings rather than integers. Note that string size in unknown.

Level 2:

Implement a function that generates the public key (n,e). You need to:

1. Compute two large pseudo-random **prime** numbers ***p*** and ***q*** of a specified number of digits.
2. Compute $\boldsymbol{\varphi}$ = (p - 1) (q - 1), and
3. Select a small integer ***e*** that is relatively prime with $\boldsymbol{\varphi}$. Use these to generate a public key (e, n)