RSA Public-Key Cryptosystem

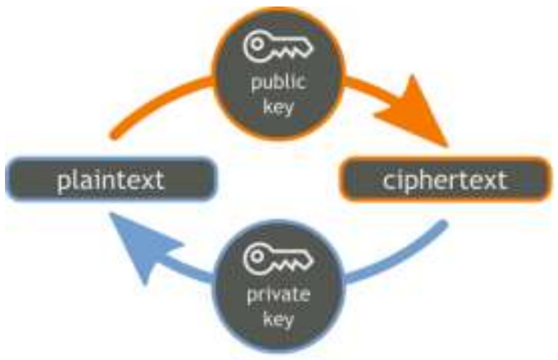
Problem Definition

The goal is to implement the RSA public-key cryptosystem. The RSA (Rivest-Shamir-Adleman) cryptosystem is widely used for secure communication in browsers, bank ATM machines, credit card machines, mobile phones, smart cards, and the Windows operating system. It works by manipulating integers. To prevent listeners, the RSA cryptosystem must manipulate huge integers (hundreds of digits). The built-in C type `int` is only capable of dealing with 16 or 32 bit integers, providing little or no security. You will:

1. Implement an extended precision arithmetic data type (big integer) that is capable of manipulating much larger integers.
2. Use this data type to write a client program that encrypts and decrypts messages using RSA.

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 decrypt 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 remaining functions (three functions) in the BigInteger data type to use it in RSA public-key cryptosystem so you can be able to encrypt and decrypt large integer **M**.

Given

BigInteger class contains many functions. The following functions are the main functions that you can use (**already implemented**):

1. **Add**: adds 2 non-negative big integers (A + B)
2. **Subtract**: subtracts 2 non-negative integers (A – B)
3. **Is_Even**: Check parity (even/odd) of a big integer
4. **PadWithZeros**: Pad a big integer with zeros

Requirements

Complete the implementation of RSA encryption/decryption via implementing the following functions:

1. **Multiply**: Efficient multiplication of two big integers
2. **DivMod**: Efficient div-mod of two big integers
3. **ModOfPower**: Efficient mod-of-power of two big integers

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. The main challenge is to design, implement, and analyze a BIG INTEGER that can manipulate large (nonnegative) integers. the data type will support the following operations: addition, subtraction, multiplication, division, and mod of power.

You need to implement the rest of functions in this BigInteger, the required functions: multiplication, division, and mod of power. Remember that you're not working with `int` or `long`

Efficient Multiplication. Implement the efficient integer multiplication (less than N^2) using the **Karatsuba algorithm** *(for detailed explanation of Karatsuba Algorithm please refer to **Lecture 5 D&C II**)*. 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.

```
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)
}
```

Mod of Power . It is used to calculate:

$$\boldsymbol{R = B^P mod\ M}$$

Direct calculation of $\boldsymbol{B^P\ mod\ M}$ take O(P) for integer values. However, we can make use of the following theory to reduce its complexity.

(A × B × C) mod N = **[**(A mod N) × (B mod N) × (c mod N)**]** mod N

Since we want **B** to the power **P** and take modulus **M**, so it is going to be:

(**(B** mod **M)** × **(B** mod **M)** × **(B** mod **M)** × ...*(**P** times)*) mod **M**

Hint: You can think of it using D&C

Task Requirements

Required Implementation

Requirement	Performance
1. Multiply function for 2 big integers.	**Time: LESS THAN O(N^2)**, N is the number of digits of the big int
2. DivMod function for 2 big integers.	**Time:** should be **bounded by O(N Log(Q))**, N is the number of digits of the big int, Q is the value of the dividend
3. Mod of power function for 2 big integers.	**Time:** should be **LESS THAN O(N^2 Log(P))**, N is the number of digits of the divisor and P is the number of digits of the power

Input

1. Multiplication

File containing **number of test cases** in the first line followed by each test case that is represented in 3 lines as follows:

1. The first operand (A)
2. The second operand (B)
3. Multiplication Result (A * B) *(for validation purpose, **not to use in your implementation**)*

2. Div-Mod

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

1. The dividend (A)
2. The divisor (B)
3. Division result (A/B) *(for validation purpose, **not to use in your implementation**)*
4. Division remainder (A mod B) *(for validation purpose, **not to use in your implementation**)*

3. Mod of Power

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

1. The base (B)
2. The exponent (P)
3. The mod (M)

4. Mod of Power result (B^P) mod M *(for validation purpose, **not to use in your implementation**)*

4. RSA

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)
5. The result of encryption or decryption *(for validation purpose, **not to use in your implementation**)*

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

1. Multiplication

- Sample Case: 10 test cases with values you can trace. Not very large integers. To be able to check your output.
- Complete Test: 100 test cases with big integers.

2. Div Mod

- Sample Case: 10 test cases with values you can trace. Not very large integers. To be able to check your output.
- Complete Test: 100 test cases with big integers.

3. Mod of Power

- Sample Case: 10 test cases with values you can trace. Not very large integers. To be able to check your output.
- Complete Test: 20 test cases with big integers.

4. RSA
    - Sample Case: 10 test cases with values you can trace. Not very large integers. To be able to check your output, consider each 2 test cases together, the first is the encryption and the second is its decryption.

    - Complete Test: 6 test cases with big integers.

Note: Reading the input from files is already handled for you. It's described so that to enable you to try new test cases as you want.

Given

1. A start-up code with test-cases that contains the following ready-made code:
    1. BigInteger class
    2. Add two big integers
    3. Subtract two big integers
    4. Check parity (even/odd) of a big integer
    5. Pad a big integer with a given number of zeros
2. Sample set of RSA public & private keys

Requirements

Complete the implementation of RSA encryption/decryption via implementing the following functions:

4. Efficient multiplication of two big integers
5. Efficient div-mod of two big integers
6. Efficient mod-of-power of two big integers

Make sure that you **test your code** successfully on the given **test cases** before submitting it.

Deliverables

Paste the <u>ENTIRE CONTENT</u> of "BigInteger.cs" file (DON'T paste the function only) in the online form.

Registration rules and deadlines:

Task is individual

Delivery date: **SAT 12 June 23:59**

If you need any support. You can ask during the office hours.

Evaluation

Requirement	**Correctness (based on test cases)**	**Efficiency (based on time limit)** **(MUST be correct)**
Efficient Mul	15%	15%
Efficient Div-Mod	12.5%	12.5%
Efficient Mod-of-Pow	12.5%	12.5%
Final Integration of RSA	-	20%

Allowed Codes [if any]

- Using any other BigInteger class is **NOT ALLOWED**. Using C# BigInteger class is **STRICTLY NOT ALLOWED**.
- External code is **NOT ALLOWED**.