Arithmetic and Number Theory

Reference:
Chapters 5 and 7 of
Programming Challenges – The programming contest training manual

Number Systems

- Natural Numbers: 1, 2, 3, …
- Integers: … -2, -1, 0 ,1 ,2, …
- Rational Numbers: A/B where A and B are integers (e.g. 6/7, 234/4567)
- Real Numbers: all numbers on the number line (e.g. sqrt(2)=1.41421…, e=2.71828…, π=3.1415926…)
- Complex Numbers: including imaginary numbers (e.g. 2i, 1+3i)

Notes:

- There is always a rational number between any two rational numbers x and y e.g. $(x+y)/2$

Manipulating rational numbers

Each rational number *x*/*y* can be represented by two integers *x* and *y*, where *x* is the numerator and *y* is the denominator.

Basic arithmetic operations on $c = x_1/y_1$, $d = x_2/y_2$:

- Addition: $c + d = \frac{x_1 y_2 + x_2 y_1}{y_1 y_2}$
- Subtraction: $c - d = \frac{x_1 y_2 - x_2 y_1}{y_1 y_2}$
- Multiplication: $c \times d = \frac{x_1 x_2}{y_1 y_2}$
- Division: $c/d = \frac{x_1}{y_1} \times \frac{y_2}{x_2} = \frac{x_1 y_2}{x_2 y_1}$

A decimal number can be represented in two parts: integer part and fractional part using three integers:
e.g. 3.1415 = 3 + 1415/10000

Manipulating rational numbers

It is important to reduce fractions to their simplest representation.
e.g. repalce 2/4 by 1/2. How?

Solution: cancel out the greatest common divisor (GCD) of the numerator and the denominator.

Representing an infinite decimal number using fractions:

infinite decimal representation, i.e., $1/3 = 0.33333...$, $1/7 = 0.142857142857...$.

- usually a decimal representation with the first ten or so significant digits suffices
- exact representation, i.e., $1/30 = 0.0\overline{3}$, or $1/7 = 0.\overline{142857}$.

How to find such a fraction for representing an infinite decimal number?

E.g. Find a/b = 0.0123123...
Solution:
Set R = 123 (i.e. the repeated part) and set L = 3 (i.e. the length of the repeated part)
Then consider 10^Lx(a/b)-(a/b) = 12.3123... - 0.0123123... = R/10.
Hence a/b = R/10/(10^3 – 1) = 123/9990.

Useful Mathematical Functions

C/C++

```
#include <math.h>               /* include the math library */

double floor(double x);    /* chop off fractional part of x */
double ceil (double x);    /* raise x to next largest integer */
double fabs(double x);     /* compute the absolute value of x */

double sqrt(double x);     /* compute square roots */
double exp(double x);      /* compute e^x */
double log(double x);      /* compute the base-e logarithm */
double log10(double x);    /* compute the base-10 logarithm */
double pow(double x, double y);   /* compute x^y */
```

Modular Arithmetic

Useful when we want to find the remainders of integers module another integer.

$\boxed{\text{Modular Arithmetic}}$: The number we are dividing by is called the modulus, and the remainder left over is called residue.
E.g. $76 \bmod 23 = 7$
where 23 is the modulus and 7 is the residue.

Addition/Subtraction: $(x \pm y) \bmod n = ((x \bmod n) \pm (y \bmod n)) \bmod n$.
Suppose x is a negative number in $[-n+1, -1]$, then
$x \bmod n = x + n \in \{0, 1, \cdots, n-1\}$
E.g. $-3 \bmod 5 = 2$

Suppose your birthday this year is on Wednesday. What day of the week will it fall on next year?

Solution: suppose there are 365 days a year. Consider 0 as Sunday, 1 as Monday and so on. Then next year, the day of the week will be (3+365) mod 7 = 4. Hence it will be Thursday.

Greatest Common Divisor (GCD)

GCD: the largest divisor shared by a given pair of integers

Two integers are relatively prime if their greatest common divisor is 1.

An Important Theorem related to GCD:

If $a = qb + r$ for some integers q and r, then

$$\gcd(a, b) = \gcd(r, b)$$

In other words,

$$\gcd(a, b) = \gcd(a \bmod b, b)$$

Question: How to find GCD?
Solution: Euclid's Algorithm

Extended Euclid's Algorithm

Euclid's algorithm can give us more than just the $\gcd(a, b)$. It can also find integers x, y such that

$$a \cdot x + b \cdot y = \gcd(a, b)$$

```
/*      Find the gcd(p,q) and x,y such that p*x + q*y = gcd(p,q)      */

long gcd(long p, long q, long *x, long *y)
{
        long x1,y1;                     /* previous coefficients */
        long g;                         /* value of gcd(p,q) */

        if (q > p) return(gcd(q,p,y,x));

        if (q == 0) {
                *x = 1;
                *y = 0;
                return(p);
        }

        g = gcd(q, p%q, &x1, &y1);

        *x = y1;
        *y = (x1 - floor(p/q)*y1);

        return(g);
}
```

Least Common Multiple (LCM)

LCM: the smallest integer which can be divided by both of the given integers.

- $lcm(x, y) \geq \max(x, y)$ for any x, y
- $lcm(x, y) \leq x \cdot y$
- $lcm(x, y) = x \cdot y / \gcd(x, y)$.

Modular Arithmetic

Multiplication: $xy \bmod n = (x \bmod n)(y \bmod n) \bmod n$.
Thus, $x^y \bmod n = (x \bmod n)^y \bmod n$.
E.g. $8 \cdot 5 \bmod 3 = (8 \bmod 3)(5 \bmod 3) \bmod 3$
$= 2 \cdot 2 \bmod 3 = 1$.

Example:

Finding the Last Digit: What is the last digit of 2^{100}?

$$
\begin{aligned}
2^3 \bmod 10 &= 8 \\
2^6 \bmod 10 &= 8 \times 8 \bmod 10 \to 4 \\
2^{12} \bmod 10 &= 4 \times 4 \bmod 10 \to 6 \\
2^{24} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{48} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{96} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{100} \bmod 10 &= 2^{96} \times 2^3 \times 2^1 \bmod 10 \to 6
\end{aligned}
$$

Modular Inverse - Division

- Division: The inverse $b^{-1} = 1/b$ of an integer b is an integer such that $bb^{-1} \equiv 1 \pmod{n}$.
- But this inverse doesn't always exist - try to find a solution to $2x \equiv 1(\mod 4)$.
- An inverse only exists if $\gcd(b, n) = 1$.

e.g.
3^{-1} mod 5 = 2
6^{-1} mod 33 has no solution

How to find a^{-1} mod n?

Solution: Use Euclid's algorithm to find

$$a*s + n*t = \gcd(a, n) = 1$$

Since $a*s + n*t$ (mod n) = 1
We have $a*s$ (mod n) = 1.
Hence s is the modular inverse of a.

Euler's Theorem

- The multiplicative group for $\boldsymbol{Z_n}$, denoted with $\boldsymbol{Z^*_n}$, is the subset of elements of $\boldsymbol{Z_n}$ relatively prime with $\boldsymbol{n}$
- The totient function of $\boldsymbol{n}$, denoted with $\phi(\boldsymbol{n})$, is the size of $\boldsymbol{Z^*_n}$
- Example

  $\boldsymbol{Z^*}_{10} = \{ 1, 3, 7, 9 \}$ $\qquad\qquad \phi(10) = 4$
- If $\boldsymbol{p}$ is prime, we have

  $\boldsymbol{Z^*_p} = \{1, 2, \ldots, (\boldsymbol{p} - 1)\}$ $\qquad \phi(\boldsymbol{p}) = \boldsymbol{p} - 1$

Euler's Theorem

For each element $\boldsymbol{x}$ of $\boldsymbol{Z^*_n}$, we have $\boldsymbol{x}^{\phi(\boldsymbol{n})} \bmod \boldsymbol{n} = 1$

- Example ($\boldsymbol{n} = 10$)

  $3^{\phi(10)} \bmod 10 = 3^4 \bmod 10 = 81 \bmod 10 = 1$

  $7^{\phi(10)} \bmod 10 = 7^4 \bmod 10 = 2401 \text{ mo d } 10 = 1$

  $9^{\phi(10)} \bmod 10 = 9^4 \bmod 10 = 6561 \bmod 10 = 1$

RSA Public-Key Encryption

- By Rivest, Shamir & Adleman of MIT in 1977
- Best known and widely used public-key algorithm
- Uses exponentiation of integers modulo a prime
- encrypt: $C = M^e \bmod n$
- *decrypt:* $M = C^d \bmod n = (M^e)^d \bmod n = M$
- both sender and receiver know values of n and e
- only receiver knows value of d
- public-key encryption algorithm with
  - public key $PU = \{e, n\}$ and private key $PR = \{d, n\}$.

RSA Cryptosystem

- Setup:
  - $n = pq$, with p and q primes
  - e relatively prime to $\phi(n) = (p - 1)(q - 1)$
  - d inverse of e in $Z_{\phi(n)}$
- Keys:
  - Public key: $K_E = (n, e)$
  - Private key: $K_D = d$
- Encryption:
  - Plaintext M in Z_n
  - $C = M^e \bmod n$
- Decryption:
  - $M = C^d \bmod n$

- Example
  - Setup:
    - $p = 7, \ q = 17$
    - $n = 7 * 7 = 119$
    - $\phi(n) = 6*6 = 96$
    - $e = 5$
    - $d = 77$
  - Keys:
    - public key: (119, 5)
    - private key: 77
  - Encryption:
    - $M = 19$
    - $C = 19^5 \bmod 119 = 66$
  - Decryption:
    - $C = 66^{77} \bmod 119 = 19$

Complete RSA Example

- Setup:
  - $p = 5$, $q = 11$
  - $n = 5 * 11 = 55$
  - $\phi(n) = 4 * 10 = 40$
  - $e = 3$
  - $d = 27$ $(3 * 27 = 81 = 2 * 40 + 1)$
- Encryption
  - $C = M^3 \bmod 55$
- Decryption
  - $M = C^{27} \bmod 55$

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

Security

- Security of RSA based on difficulty of factoring
  - Widely believed
  - Best known algorithm takes exponential time
- RSA Security factoring challenge (discontinued)
- In 1999, 512-bit challenge factored in 4 months using 35.7 CPU-years
  - 160 175-400 MHz SGI and Sun
  - 8 250 MHz SGI Origin
  - 120 300-450 MHz Pentium II
  - 4 500 MHz Digital/Compaq
- In 2005, a team of researchers factored the RSA-640 challenge number using 30 2.2GHz CPU years
- In 2004, the prize for factoring RSA-2048 was $200,000
- Current practice is 2,048-bit keys
- Estimated resources needed to factor a number within one year

Length (bits)	PCs	Memory
430	1	128MB
760	215,000	4GB
1,020	342×10^{6}	170GB
1,620	1.6×10^{15}	120TB

Correctness

- We show the correctness of the RSA cryptosystem for the case when the plaintext $\boldsymbol{M}$ does not divide $\boldsymbol{n}$
- Namely, we show that

$$(\boldsymbol{M}^{\boldsymbol{e}})^{\boldsymbol{d}} \bmod \boldsymbol{n} = \boldsymbol{M}$$

- Since $\boldsymbol{ed} \bmod \phi(\boldsymbol{n}) = 1$, there is an integer $\boldsymbol{k}$ such that

$$\boldsymbol{ed} = \boldsymbol{k}\phi(\boldsymbol{n}) + 1$$

- Since $\boldsymbol{M}$ does not divide $\boldsymbol{n}$, by Euler's theorem we have

$$\boldsymbol{M}^{\phi(\boldsymbol{n})} \bmod \boldsymbol{n} = 1$$

- Thus, we obtain

$$\begin{aligned}
&(\boldsymbol{M}^{\boldsymbol{e}})^{\boldsymbol{d}} \bmod \boldsymbol{n} = \\
&\qquad \boldsymbol{M}^{\boldsymbol{ed}} \bmod \boldsymbol{n} = \\
&\qquad \boldsymbol{M}^{\boldsymbol{k}\phi(\boldsymbol{n}) + 1} \bmod \boldsymbol{n} = \\
&\qquad \boldsymbol{M}\boldsymbol{M}^{\boldsymbol{k}\phi(\boldsymbol{n})} \bmod \boldsymbol{n} = \\
&\qquad \boldsymbol{M}\ (\boldsymbol{M}^{\phi(\boldsymbol{n})})^{\boldsymbol{k}} \bmod \boldsymbol{n} = \\
&\qquad \boldsymbol{M}\ (\boldsymbol{M}^{\phi(\boldsymbol{n})} \bmod \boldsymbol{n})^{\boldsymbol{k}} \bmod \boldsymbol{n} = \\
&\qquad \boldsymbol{M}\ (1)^{\boldsymbol{k}} \bmod \boldsymbol{n} = \\
&\qquad \boldsymbol{M} \bmod \boldsymbol{n} = \\
&\qquad \boldsymbol{M}
\end{aligned}$$

- Proof of correctness can be extended to the case when the plaintext $\boldsymbol{M}$ divides $\boldsymbol{n}$

Algorithmic Issues

- The implementation of the RSA cryptosystem requires various algorithms
- Overall
  - –Representation of integers of arbitrarily large size and arithmetic operations on them
- Encryption
  - –Modular power
- Decryption
  - –Modular power
- Setup
  - –Generation of random numbers with a given number of bits (to generate candidates p and q)
  - –Primality testing (to check that candidates p and q are prime)
  - –Computation of the GCD (to verify that e and $\phi(n)$ are relatively prime)
  - –Computation of the multiplicative inverse (to compute d from e)

Big number

Sometimes, we need to write our own data structure and handle arithmetic operations on integers which are tens or even hundreds of digits long.

- factorial： 500!
- exponential： 103^{900}
- recursion formula： $f(n) = f(n-1) + f(n-2)$
- m – ary base： $a_0 + a_1 m^1 + a_2 m^2 + \ldots + a_n m^n$
- …
- operation: most operation in big number are addition(+), multiplication(*)

array & number

Below is an example on how the data structure is defined and how the arithmetic operations are implemented.

- Use array bit[] to represent a big number n
- Use bit[0] as the # of digits of the number.

➢ r = bit[0]

➢ $n = bit[1] \times 1 + bit[2] \times 10^1 + \ldots + bit[r] \times 10^{r-1}$

- $n = 720 = 0 \times 1 + 2 \times 10 + 7 \times 100$
- bit[] = {3,0,2,7}

number to array

- ❑ $n = bit[1] \times 1 + bit[2] \times 10^1 + \ldots + bit[r] \times 10^{r-1}$

```
Initially, bit[0] = 0;
//bit[0] contain # of digits of number n
void numberToArray(int n,int bit[]){
while(n!=0){
     bit[ ++bit[0] ] = n%10;
     n = n/10;
}}
```

- n = 720
- bit[] = {3,0,2,7}

big number: addition

1) Do operation.

c[] = a[]+b[]

2) handle carry-bit.

what if c[] ≥ 10?

c[i]’= (c[i]+flag)%10

flag’= (c[i]+flag)/10

3) handle carry flag.

what if flag > 0?

```
//addition: c[] = a[] + b[].
//Initially, c[]={0}
void add(int a[], int b[], int c[]){
//[1] add digits: c = a + b
    FOR(i, 1, a[0]) c[i] += a[i];
    FOR(i, 1, b[0]) c[i] += b[i];
    c[0] = max(a[0],b[0]);
//[2] handle carry-bit,
    int flag = 0; //carry flag
    FOR(i,1,c[0]){
        c[i] += flag;
        flag = c[i]/10;
        c[i] = c[i]%10;
    }
//[3] handle carry-flag, flag <10
    if(flag) c[++c[0]] = flag;
}
```

big number: multiplication

1) Do multiplication.

c[] = a[] × b[]

2) handle carry-bit.

what if c[] ≥ 10?

c[i]' = (c[i]+flag)%10

flag' = (c[i]+flag)/10

3) handle carry flag.

what if flag > 10?

```
//multiplication: c[] = a[] * b[]
void multiply(int a[],int b[],int c[]){
//[1] multiply digits: c = a * b
    FOR(i, 1, a[0])
        FOR(j, 1, b[0])
                c[i+j-1] += a[i] * b[j];
    c[0] = a[0] + b[0] - 1;
//[2] handle carry bit,
    same with addtion
//[3] handle carry flag
    while(flag){
        c[++c[0]] = flag%10;
        flag = flag/10;
    }
}
```