A Concrete Introduction to Number Theory and Algebra–Chapter 1-2

王立斌

School of Computer Science, South China Normal University

September 5, 2022

Table of contents

Motivations.

We focus on these topics:

- Number Theory
- Abstract Algebra
- Algorithms

Motivations.

We focus on these topics:

- Number Theory
- Abstract Algebra
- Algorithms

Notice!

It should be emphasized, we focus on algorithms and heavily rely on programming, which is a manner called "computational(计算性)".

References.

- Number Theory: A Friendly Introduction to Number Theory (FINT)
- Abstract Algebra: Abstract Algebra: Theory and Applications (AATA)
- Algorithms: Introductin to Algorithms (CLRS)

References.

- Number Theory: A Friendly Introduction to Number Theory (FINT)
- Abstract Algebra: Abstract Algebra: Theory and Applications (AATA)
- Algorithms: Introductin to Algorithms (CLRS)

Wanted!

Reading, thinking and programming.

Remark.

本课程最重要的思路!

Finding patterns. （寻找模式.）

Numbers.

Number systems

$$\mathbb{N} = \{0, 1, 2, \cdots\}$$

$$\mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \cdots\}$$

$$\mathbb{Q}$$

$$\mathbb{R}$$

Warm-up-1.

Simple Code:

```
1 unsigned char a = 255, b = 2;
2 a = a + b;
3 printf("a == %d \n", a);
```

Warm-up-2.

Simple Code:

```
1  int n = 32; /*try different values here;*/
2  unsigned int val = 1; /*or try a different type here;*/
3  for(int i = 1; i <= n; i++)
4  {
5          val = val * 2;
6          printf("What you get is %u \n.", val);
7  }
```

Warm-up-3.

CSers 的计算方法.

请计算以下等式:

$$1 + 2 + 2^2 + 2^3 + \cdots + 2^{10}$$

Numbers in Computer Systems.

Observations:

- All computation in computers are fixed-sized(定长计算).
- Addition is natural while multiplication is not.
- Subtraction is the same as addition; the relation between multiplication and division is more complicate.

Simple rules for addition and multiplication.

Well-known and useful rules of arithmetic:

- The last bit of an even(odd) number is $0(1)$;
- To multiply(divide) an unsigned integers by 2 is equivalent to shifting left(right) the number by 1 bit.

诡异的加法.

整数加法

```
1  //输入: 两个整数a和b
2  //输出: a与b的和
3  int add(int a, int b) {
4
5      if (b == 0) return a;
6      return add(a ^ b, (b & a) << 1);
7
8  }
```

理解加法.

要理解这个加法算法只需要地正确回答出以下三个问题:

1. $a \wedge b$ 得到的是什么?
2. $(b\&a) << 1$ 得到的是什么?
3. 该算法为什么会终止?

Naïve Multiplication.

```
1  # Input: two integers a and b, where a >= b.
2  # Output: the product of a and b.
3  def naive_multiply(a, b):
4      if b == 0:
5          return 0
6      return a + naive_multiply(a, b - 1)
```

Three key issues.

In general, three key issues should be considered when we encounter a new algorithm.

- **Correctness(正确性)**. Is the algorithm correctly achieve the goal?
- **Efficiency(效率、复杂度)**. How long does the algorithm take when it terminates?
- **Optimization(优化)**. Can we provide a better method?

Rule of Multiplication.

Rule of Multiplication.

$$a \cdot b = \begin{cases} 2(a \cdot \lfloor b/2 \rfloor) & \text{if } b \text{ is even;} \\ a + 2(a \cdot \lfloor b/2 \rfloor) & \text{if } b \text{ is odd.} \end{cases} \tag{1}$$

Simple multiplication.

```
1  # Input: two integers a and b.
2  # Output: the product of a and b.
3  def multiply(a, b):
4      if b == 0:
5          return 0;
6      if is_even(b): # the last bit of b is 0;
7              return 2*multiply(a, b/2);
8      else: # b is odd
9              return 2*multiply(a, b/2) + a;
```

Simple multiplication.

Intuition.

For two n-bits integers a and b, view b as a binary number in position notation(位置记数法).

$$\begin{aligned} a \cdot b &= a \cdot (b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2 + b_0) \\ &= b_{n-1}(a \cdot 2^{n-1}) + b_{n-2}(a \cdot 2^{n-2}) + \cdots + b_1(a \cdot 2) + b_0 a \end{aligned}$$

Observations: 1. Every term in the last equation has $2^i a$, which means we must continue to do $a* = 2$
2. The bit b_i in every term plays as a flag, when $b_i == 1$, we must have $res+ = a$;

Simple multiplication.

Example.

To compute 3×10.

Simple multiplication.

Example.

To compute 3×10.

10's binary string is 1010.

Simple multiplication.

Example.

To compute 3×10.
10's binary string is 1010.
Repeatly multiply 3 by 2^i and get: 3, 6, 12, 24.

Simple multiplication.

Example.

To compute 3×10.
10's binary string is 1010.
Repeatly multiply 3 by 2^i and get: 3, 6, 12, 24.
Choose the right terms to add, they are 6 and 24.

Simple multiplication.

Example.

To compute 3×10.
10's binary string is 1010.
Repeatly multiply 3 by 2^i and get: 3, 6, 12, 24.
Choose the right terms to add, they are 6 and 24.
Return the result which is 30.

Simple multiplication.

Analysis

- Correctness. Obviously...
- Efficiency. $O(n^2)$ bit operations.
- Optimization. To be or not to be...

Notice!

The multiplication is implemented NOT using multiplication !

Division Algorithm.

Theorem

(Division Algorithm) Given integers a and b, with $b > 0$, there exist unique integers q and r, called quotient and remainder respectively, such that

$$a = qb + r$$

where $0 \leq r < b$.

Geometric interpretation of division.

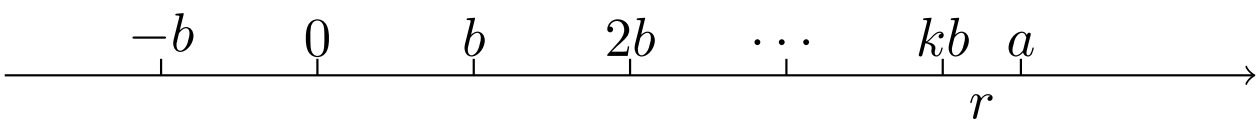

Figure: Geometric interpretation of division.

Axiom.

Principle of Well-Ordering(良序原则).

Every nonempty subset of natural numbers contain a least element.

Proof of division theorem.

Proof.

The proof contains two parts: existence(存在性) and uniqueness(唯一性).

Existence of q and r. Let

$$S = \{a - bk : k \in \mathbb{Z} \text{ and } a - bk \geq 0\}.$$

It is easy to show that S is nonempty. By the Principle of Well-Ordering, there is a smallest member $r \in S$, and $r = a - qb$. Therefore, $a = qb + r$, $r \geq 0$. To show $r < b$ is leaved as an exercise.

Uniqueness of q and r. This is an exercise.

Naive Division.

```
1  def Navie_Divide(a, b):
2      q, r = 0, 0
3      while(a >= b):
4          a, q = a - b, q + 1
5      r = a
6      return q, r
```

Simple Division.

```
1  def Divide(a, b):
2      if(a == 0):
3          return 0, 0
4      q, r = Divide(a / 2, b)
5      q, r = 2 * q, 2 * r
6      if(a & 1): #if a is an odd number
7          r = r + 1
8      if(r >= b):
9          r, q = r - b, q + 1
10     return q, r
```

Important remark.

Important remark.

Multiplication is implemented by using addition and shift, and division is implemented by using subtraction and shift. Moreover, it is not about implementation, but is about thinking. We will see it later.

Some terms.

- **Divisibility.** Let a and b be integers, a is said to be *divisible* by b when $a = qb$ for some integer q, we write $b \mid a$.
- **Common divisor.** An integer d is called a common divisor of a and b if $d \mid a$ and $d \mid b$.
- **Greatest common divisor.** An integer d is the greatest common divisor(gcd) of a and b, if d is a common divisor of a and b, and for any other common divisor of a and b, says d', then $d' \mid d$.
- **Relatively prime.** If $gcd(a, b) == 1$, we say that a and b are relatively prime.

The Euclidean Algorithm.

Euclidean algorithm(gcd algorithm).

Given two positive integers a and b with $a \geq b$:

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

where $a \bmod b$ means to divide a by b to get the remainder r.

The Euclidean Algorithm.

Example

Given $a = 12345$ *and* $b = 678$,

$$gcd(12345, 678) = gcd(678, 141) = gcd(141, 114)$$

$$= gcd(114, 27) = gcd(27, 6) = gcd(6, 3) = gcd(3, 0)$$

Hence, $gcd(12345, 678) = 3$.

The Euclidean Algorithm.

```
1  # Input: positive integers a and b, with a > b .
2  # Output: the greatest common divisor of a and b.
3  def gcd(a, b):
4      if(b == 0):
5          return a
6      else:
7          return gcd(b, a % b)
```

Correctness of The Euclidean Algorithm.

Correctness. It is enough to show a slightly simpler rule:

$$gcd(a, b) = gcd(a - b, b).$$

From this rule, we can repeatedly subtract b from a (recall the Naive division algorithm) and derive the final rule as follows:

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

Correctness of The Euclidean Algorithm.

To show $gcd(a, b) = gcd(a - b, b)$, we have two directions to prove.

- any integer that divides both a and b must also divides $a - b$, thus $gcd(a, b) \leq gcd(a - b, b)$.
- any integer that divdes both $a - b$ and b must also divides a and b, thus $gcd(a - b, b) \leq gcd(a, b)$.

Efficiency of The Euclidean Algorithm.

Efficiency. To understand that the *gcd* algorithm is quite fast, we should be aware of the fact that a mod b is a substantial reduction. It is easy to check:

Lemma

If $a \geq b$, then $(a \bmod b) < a/2$.

Hence, after two consecutive rounds, both a and b are at least halved in value, means the algorithm will terminate after $O(\log a)$ recursive calls.

Optimization of The Euclidean Algorithm.

Can we do better? The *binary gcd algorithm*, also known as Stein's algorithm, has the same function as the gcd algorithm. This algorithm avoids complex operations, such as division and multiplication; instead, it relies only on subtraction, and more efficient bit right shift and bit left shift operations. The analysis of binary gcd algorithm is an exercise.

Bézout's Theorem.

Theorem

(Bézout) Let a and b be nonzero integers. Then there exist integers r and s such that

$$gcd(a, b) = ar + bs$$

Furthermore, the greatest common divisor of a and b is unique.

Bézout's Theorem.

Theorem

(Bézout) Let a and b be nonzero integers. Then there exist integers r and s such that

$$gcd(a, b) = ar + bs$$

Furthermore, the greatest common divisor of a and b is unique.

Proof of Bézout's Theorem.

Proof.

构造集合

$$S = \{am + bn : m, n \in \mathbb{Z} \text{ 且 } am + bn > 0\}.$$

显然，集合 S 非空，根据良序原则，取其中最小值 $d = ar + bs$。然后证明两个属性：d 是 a 和 b 的公因子（提示：除法算法!）；如果存在 a 和 b 的公因子 d'，则 $d' \mid d$。这样就证明了 $d = gcd(a, b)$。请读者完成细节，留作课后练习。 □

The Extended Euclidean Algorithm.

The extended Euclidean algorithm.

The *extended Euclidean algorithm*(*egcd* algorithm) is a simple extension of the Euclidean algorithm. Given two nonzero integers$a$ and b, it efficiently computes the *Bézout coefficients* r and s, and plays an important role in modular arithmetic.

The Extended Euclidean Algorithm.

Example

Given $a = 19$, $b = 13$, *to find* r *and* s, *s.t.* $ar + bs = gcd(19, 13)$.

$$a \cdot 1 + b \cdot 0 = 19 \tag{2}$$

$$a \cdot 0 + b \cdot 1 = 13 \tag{3}$$

$$a \cdot 1 + b \cdot (-1) = 6 \tag{4}$$

$$a \cdot (-2) + b \cdot 3 = 1 \tag{5}$$

Finally, we have $r = -2$, $s = 3$ *and* $d = 1$, *which is the answer.*

The Extended Euclidean Algorithm.

Recapitulate the process.

- Write out two obviously true linear equations.;
- Repeatedly subtract some integer multiple of one equation from the other, until one of the equation's right side achieves $gcd(a, b)$.

The Extended Euclidean Algorithm.

Two Observations.

At least two facts should be noticed

- The iterations of the right side of the aforemention equations do the same job of *gcd* algorithm.
- The a and b are just place holders, they do not involve in the computation, we can safely get rid of them in the equations.

The Extended Euclidean Algorithm.

Example

Given $a = 13$, $b = 9$, *to find* r *and* s, *s.t.* $ar + bs = gcd(13, 9)$. *We write out a matrix:*

$$\begin{pmatrix} 1 & 0 & 13 \\ 0 & 1 & 9 \end{pmatrix}$$

If we subtract row 2 from row 1, we get a new matrix:

$$\begin{pmatrix} 0 & 1 & 9 \\ 1 & -1 & 4 \end{pmatrix}$$

The Extended Euclidean Algorithm.

Example

Then we subtract 2 multiple of row 2 from row 1, we get:

$$\begin{pmatrix} 1 & -1 & 4 \\ -2 & 3 & 1 \end{pmatrix}$$

Finally, we have $r = -2$, $s = 3$ *and* $d = 1$, *which is the answer.*

The Extended Euclidean Algorithm.

From the perspective of matrix, the process starts with a matrix

$$M = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \end{pmatrix}$$

which satisfies

$$M \begin{pmatrix} a \\ b \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

By repeatedly subtracting a multiple of one row of M from the other row or exchanging the two rows, finally we obtain a matrix M'

$$M' = \begin{pmatrix} r & s & d \\ R & S & 0 \end{pmatrix}$$

The Extended Euclidean Algorithm.

$$M' = \begin{pmatrix} r & s & d \\ R & S & 0 \end{pmatrix}$$

satisfies

$$M' \begin{pmatrix} a \\ b \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

That means $ar + bs = d$.

The Extended Euclidean Algorithm.

The extended Euclidean algorithm is essentially the process to calculate Bézout coefficients we described above.

```
1  # Input: integers a and b, with a > b .
2  # Output: d=gcd(a,b), and
3  # r and s s.t. d = a*r + b*s
4  def egcd(a, b):
5      r0, r1, s0, s1 = 1, 0, 0, 1
6      while(b):
7          q, a, b = a/b, b, a%b
8          r0, r1 = r1, r0 - q*r1
9          s0, s1 = s1, s0 - q*s1
10     return a, r0, s0
```

Analysis of The Extended Euclidean Algorithm.

- Correctness: it is easy.
- Efficiency: it is trivial.
- Optimization: binary-egcd.

Concluding Remarks.

- The content is easy, but is not trivial.
- Algorithms should be focused by CSers.
- Useful tools should be mastered.

Happy ending.

- Any questions?
- Feedback will be welcomed : lbwang@gmail.com.
- Thanks!