算法入门-基础数学

Junlist

2025 年 11 月 12 日

1 欧几里得算法

1.1 数学原理

X 是整数 A 和 B 的最大公因数（即 $X = \gcd(A, B)$），根据最大公因数的定义，可得：

$$A = K_1 \cdot X, \quad B = K_2 \cdot X$$

其中 K_1 和 K_2 是互质的整数（即 $\gcd(K_1, K_2) = 1$）。

A 除以 B 的余数（记为 $A \mod B$）可表示为：

$$A \mod B = A - n \cdot B$$

其中 n 是商（$n = \left\lfloor \frac{A}{B} \right\rfloor$）。

将 $A = K_1 \cdot X$ 和 $B = K_2 \cdot X$ 代入上式，得：

$$A \mod B = K_1 \cdot X - n \cdot K_2 \cdot X = (K_1 - n \cdot K_2) \cdot X$$

显然，$A \mod B$ 仍是 X 的倍数。根据最大公约数的性质：

$$\gcd(K_2, K_1 - n \cdot K_2) = \gcd(K_2, K_1) = 1$$

这意味着 X 也是 B 和 $A \mod B$ 的最大公因数。因此，求 $\gcd(A, B)$ 的问题可转化为求 $\gcd(B, A \mod B)$。

1.2 算法的结束条件

欧几里得算法的结束条件是当余数为 0 时：

$$\gcd(A, B) = \begin{cases} A & \text{if } B = 0 \\ \gcd(B, A \mod B) & \text{otherwise} \end{cases}$$

数学证明：当 $B = 0$ 时，A 除以 0 无定义，但根据定义 $\gcd(A, 0) = |A|$（$A \neq 0$）由于 $A \mod 0$ 无意义，算法在 $B = 0$ 时终止，此时 A 即为最大公约数。

1.3 为什么不能使用 $\gcd(A, A \mod B)$?

数学证明：假设 $\gcd(A, A \mod B) = \gcd(A, B)$，则：

$$\begin{aligned}\gcd(A, B) &= \gcd(A, A \mod B) \\ &= \gcd(A, A - n \cdot B) \\ &= \gcd(A, -n \cdot B) \\ &= \gcd(A, n \cdot B)\end{aligned}$$

不妨令:

$$A = q \cdot B + r$$

则一定有:

$$A = \left\lfloor \frac{q}{n} \right\rfloor \cdot n \cdot B + \left(\frac{q}{n} - \left\lfloor \frac{q}{n} \right\rfloor \right) \cdot B + r$$

那么易知只有$\frac{q}{n} - \left\lfloor \frac{q}{n} \right\rfloor = 0$时才有$\gcd(A, A \mod B) = \gcd(A, B)$ 而这只是一个特殊的情况而不是普遍结论故而不能得到$\gcd(A, A \mod B) = \gcd(A, B)$ 。

1.4 非递归版本代码

```
int gcd(int A, int B)
{
    while(B != 0){
        int temp = B;
        B = A % B;
        A = temp;
    }
    return A;
}
```

Listing 1: 非递归版本代码

1.5 递归版本代码

```
int gcd(int A, int B)
{
    return ((B == 0) ? A : gcd(B, A % B));
}
```

Listing 2: 递归版本代码

2 循环节

循环节（Cycle Detection）是算法与数论中的经典概念，其计算本身往往简单直接，但关键挑战在于如何从问题描述中识别出潜在的周期性结构。在算法竞赛与实际编程问题中，循环节问题频繁出现，却极少以“请找出循环节”这样直白的形式提出。更常见的是：给定一个递推规则、模运算序列、状态转移过程，或小数展开形式，要求预测第 n 项、判断是否重复、或计算长期行为。此时，若能敏锐察觉其中隐含的周期性，问题便可迎刃而解。

在一些简单的循环节问题中，我们可以通过判断递推式不同组成部分的不同形态来判断递推式一共有几种形式，进而根据**抽屉原理**推断会在几个周期内产生循环节。

3 快速幂

3.1 数学原理

快速幂的核心思想是将指数 b 表示为二进制形式，并将 a^b 分解为若干个幂的乘积。设 b 的二进制表示为 $b = (b_k b_{k-1} \dots b_1 b_0)_2$（其中 $b_i \in \{0, 1\}$），则：

$$a^b = \prod_{i=0}^{k} a^{b_i \cdot 2^i} = a^{b_k \cdot 2^k} \cdot a^{b_{k-1} \cdot 2^{k-1}} \cdot \ldots \cdot a^{b_1 \cdot 2^1} \cdot a^{b_0 \cdot 2^0}$$

关键观察： 当 $b_i = 0$ 时，$a^{b_i \cdot 2^i} = a^0 = 1$，因此实际计算中只需对二进制位为 1 的项进行乘法运算。例如：

- 若 $b = 10_{(10)} = 1010_{(2)}$（即 $b_3 = 1, b_2 = 0, b_1 = 1, b_0 = 0$），

- 则 $a^{10} = a^{1\cdot 2^3} \cdot a^{0\cdot 2^2} \cdot a^{1\cdot 2^1} \cdot a^{0\cdot 2^0} = a^8 \cdot a^2$

计算效率： 通过二进制分解，只需计算 $\lfloor \log_2 b \rfloor$ 次平方运算（得到 a^{2^i}）和 popcount(b) 次乘法（b 的二进制中 1 的个数），将时间复杂度从 $O(b)$ 优化至 $O(\log b)$。

算法应用： 在算法中其主要为了快速计算$A^B \bmod C$由于在算法比赛中B和A的值一般来说非常大，故需要在每次运算mod$C$以及在做预算时候防止整数溢出。

3.2 非递归版本代码

```
long long fast_pow(long long a, long long b, long long c)
{
    long long res = 1;
    while(b != 0){
        if (b & 1){
            res = (res * a) % c;
        }
        a = (a * a) % c;
        b >>= 1;
    }
    return res;
}
```

Listing 3: 非递归版本代码

3.3 递归版本代码

```
long long fast_pow(long long a, long long b, long long c)
{
    if (b == 0){
        return 1;
    }
    long long res = fast_pow(a, b >> 1, c);
    res = (res * res) % c;
    if (b & 1) {
        res = (res * a) % c;
    }
```

```
11      return res;
12  }
```

Listing 4: 递归版本代码