最大公约数

定义

最大公约数即为 Greatest Common Divisor，常缩写为 gcd。

一组整数的公约数，是指同时是这组数中每一个数的约数的数。± 1 是任意一组整数的公约数。

一组整数的最大公约数，是指所有公约数里面最大的一个。

对不全为 0 的整数 a, b，将其最大公约数记为 $\gcd(a, b)$，不引起歧义时可简写为 (a, b)。

对不全为 0 的整数 $a_1, \dots, a_n$，将其最大公约数记为 $\gcd(a_1, \dots, a_n)$，不引起歧义时可简写为 $(a_1, \dots, a_n)$。

最大公约数与最小公倍数的性质见 数论基础。

那么如何求最大公约数呢？我们先考虑两个数的情况。

欧几里得算法

过程

如果我们已知两个数 a 和 b，如何求出二者的最大公约数呢？

不妨设 $a > b$。

我们发现如果 b 是 a 的约数，那么 b 就是二者的最大公约数。 下面讨论不能整除的情况，即 $a = b \times q + r$，其中 $r < b$。

我们通过证明可以得到 $\gcd(a, b) = \gcd(b, a \bmod b)$，过程如下：

证明

设 $a = bk + c$，显然有 $c = a \bmod b$。设 $d \mid a$，$d \mid b$，则 $c = a - bk, \frac{c}{d} = \frac{a}{d} - \frac{b}{d}k$。

由右边的式子可知 $\frac{c}{d}$ 为整数，即 $d \mid c$，所以对于 a, b 的公约数，它也会是 $b, a \bmod b$ 的公约数。

反过来也需要证明：

设 $d \mid b$，$d \mid (a \bmod b)$，我们还是可以像之前一样得到以下式子 $\frac{a \bmod b}{d} = \frac{a}{d} - \frac{b}{d}k, \frac{a \bmod b}{d} + \frac{b}{d}k = \frac{a}{d}$。

因为左边式子显然为整数，所以 $\frac{a}{d}$ 也为整数，即 $d \mid a$，所以 $b, a \bmod b$ 的公约数也是 a, b 的公约数。

既然两式公约数都是相同的，那么最大公约数也会相同。

所以得到式子 $\gcd(a, b) = \gcd(b, a \bmod b)$

既然得到了 $\gcd(a, b) = \gcd(b, r)$，这里两个数的大小是不会增大的，那么我们也就得到了关于两个数的最大公约数的一个递归求法。

实现

C++

```cpp
// Version 1
int gcd(int a, int b) {
  if (b == 0) return a;
  return gcd(b, a % b);
}

// Version 2
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
```

Java

```java
// Version 1
public int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

// Version 2
public int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

Python

```python
def gcd(a, b):
    if b == 0:
```

```
        return a
    return gcd(b, a % b)
```

递归至 `b == 0`（即上一步的 `a % b == 0`）的情况再返回值即可。

根据上述递归求法，我们也可以写出一个迭代求法：

C++

```cpp
int gcd(int a, int b) {
  while (b != 0) {
    int tmp = a;
    a = b;
    b = tmp % b;
  }
  return a;
}
```

Java

```java
public int gcd(int a, int b) {
    while(b != 0) {
        int tmp = a;
        a = b;
        b = tmp % b;
    }
    return a;
}
```

Python

```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

上述算法都可被称作欧几里得算法（Euclidean algorithm）。

另外，对于 C++17，我们可以使用 `<numeric>` 头中的 `std::gcd` 与 `std::lcm` 来求最大公约数和最小公倍数。

注意

在部分编译器中，C++14 中可以用 `std::__gcd(a,b)` 函数来求最大公约数，但是其仅作为 `std::rotate` 的私有辅助函数。[1]使用该函数可能会导致预期之外的问题，故一般情况下不推荐使用。

如果两个数 a 和 b 满足 $\gcd(a, b) = 1$，我们称 a 和 b 互质。

性质

欧几里得算法的时间效率如何呢？下面我们证明，在输入为两个长为 n 的二进制整数时，欧几里得算法的时间复杂度为 $O(n)$。（换句话说，在默认 a, b 同阶的情况下，时间复杂度为 $O(\log\max(a,b))$。）

证明

当我们求 $\gcd(a,b)$ 的时候，会遇到两种情况：

- $a < b$，这时候 $\gcd(a,b)=\gcd(b,a)$；
- $a \geq b$，这时候 $\gcd(a,b)=\gcd(b, a \bmod b)$，而对 a 取模会让 a 至少折半。这意味着这一过程最多发生 $O(\log a)=O(n)$ 次。

第一种情况发生后一定会发生第二种情况，因此第一种情况的发生次数一定 **不多于** 第二种情况的发生次数。

从而我们最多递归 $O(n)$ 次就可以得出结果。

事实上，假如我们试着用欧几里得算法去求 斐波那契数列 相邻两项的最大公约数，会让该算法达到最坏复杂度。

更相减损术

大整数取模的时间复杂度较高，而加减法时间复杂度较低。针对大整数，我们可以用加减代替乘除求出最大公约数。

过程

已知两数 a 和 b，求 $\gcd(a,b)$。

不妨设 $a \geq b$，若 $a=b$，则 $\gcd(a,b)=a=b$。 否则，$\forall d \mid a, d \mid b$，可以证明 $d \mid a-b$。

因此，a 和 b 的 **所有** 公因数都是 $a-b$ 和 b 的公因数，$\gcd(a,b)=\gcd(a-b,b)$。

Stein 算法的优化

如果 $a \gg b$，更相减损术的 $O(n)$ 复杂度将会达到最坏情况。

考虑一个优化，若 $2 \mid a, 2 \mid b$，$\gcd(a,b)=2\gcd\left(\frac{a}{2},\frac{b}{2}\right)$。

否则，若 $2 \mid a$（$2 \mid b$ 同理），因为 $2 \mid b$ 的情况已经讨论过了，所以 $2 \nmid b$。因此 $\gcd(a,b)=\gcd\left(\frac{a}{2},b\right)$。

优化后的算法（即 Stein 算法）时间复杂度是 $O(\log n)$。

证明

若 $2 \mid a$ 或 $2 \mid b$，每次递归至少会将 a, b 之一减半。

否则，$2 \mid a - b$，回到了上一种情况。

算法最多递归 $O(\log n)$ 次。

实现

高精度模板见 高精度计算。

高精度运算需实现：减法、大小比较、左移、右移（可用低精乘除代替）、二进制末位 0 的个数（可以通过判断奇偶暴力计算）。

C++

```cpp
Big gcd(Big a, Big b) {
  if (a == 0) return b;
  if (b == 0) return a;
  // 记录a和b的公因数2出现次数，countr_zero表示二进制末位0的个数
  int atimes = countr_zero(a);
  int btimes = countr_zero(b);
  int mintimes = min(atimes, btimes);
  a >>= atimes;
  for (;;) {
    // a和b公因数中的2已经计算过了，后面不可能出现a为偶数的情况
    b >>= btimes;
    // 确保 a<=b
    if (a > b) swap(a, b);
    b -= a;
    if (b == 0) break;
    btimes = countr_zero(b);
  }
  return a << mintimes;
}
```

上述代码参考了 libstdc++ 和 MSVC 对 C++17 `std::gcd` 的实现。在 `unsigned int` 和 `unsigned long long` 的数据范围下，如果可以以极快的速度计算 `countr_zero`，则 Stein 算法比欧几里得算法来得快，但反之则可能比欧几里得算法慢。

关于 countr_zero

1. gcc 有 内建函数 `__builtin_ctz`（32 位）或 `__builtin_ctzll`（64 位）可替换上述代码的 `countr_zero`；
2. 从 C++20 开始，头文件 `<bit>` 包含了 `std::countr_zero`；
3. 如果不使用不在标准库的函数，又无法使用 C++20 标准，下面的代码是一种在 Word-RAM with multiplication 模型下经过预处理后 $O(1)$ 的实现：

```
constexpr int loghash[64] = {0,  32, 48, 56, 60, 62, 63, 31,
47, 55, 59, 61, 30,
                             15, 39, 51, 57, 28, 46, 23, 43,
53, 58, 29, 14, 7,
                             35, 49, 24, 44, 54, 27, 45, 22,
11, 37, 50, 25, 12,
                             38, 19, 41, 52, 26, 13, 6,  3,
33, 16, 40, 20, 42,
                             21, 10, 5,  34, 17, 8,  36, 18, 9,
4,  2,  1};

int countr_zero(unsigned long long x) {
  return loghash[(x & -x) * 0x9150D32D8EB9EFC0Ui64 >> 58];
}
```

而对于高精度运算，如果实现方法类似 `bitset`，则搭配上述对 `countr_zero` 的实现可以在 `O(n / w)` 的时间复杂度下完成。但如果不便按二进制位拆分，则只能暴力判断最大的 2 的幂因子，时间复杂度取决于实现。比如：

```
// 以小端序实现的二进制 Big，要求能枚举每一个元素
int countr_zero(Big a) {
  int ans = 0;
  for (auto x : a) {
    if (x != 0) {
      ans += 32;  // 每一位数据类型的位长
    } else {
      return ans + countr_zero(x);
    }
  }
  return ans;
}

// 暴力计算，如需使用建议直接写进 gcd 加快常数
int countr_zero(Big a) {
  int ans = 0;
  while ((a & 1) == 0) {
    a >>= 1;
    ++ans;
  }
```

```
    return ans;
}
```

更多关于 `gcd` 实现上快慢的讨论可阅读 Fastest way to compute the greatest common divisor。

多个数的最大公约数

那怎么求多个数的最大公约数呢？显然答案一定是每个数的约数，那么也一定是每相邻两个数的约数。我们采用归纳法，可以证明，每次取出两个数求出答案后再放回去，不会对所需要的答案造成影响。

最小公倍数

接下来我们介绍如何求解最小公倍数（Least Common Multiple, LCM）。

定义

一组整数的公倍数，是指同时是这组数中每一个数的倍数的数。0 是任意一组整数的公倍数。

一组整数的最小公倍数，是指所有正的公倍数里面，最小的一个数。

对整数 a,b，将其最小公倍数记为 $\operatorname{lcm}(a,b)$，不引起歧义时可简写为 $[a,b]$。

对整数 $a_1,\dots,a_n$，将其最小公倍数记为 $\operatorname{lcm}(a_1,\dots,a_n)$，不引起歧义时可简写为 $[a_1,\dots,a_n]$。

两个数

设 $a=p_1^{k_{a_1}}p_2^{k_{a_2}}\cdots p_s^{k_{a_s}}$，$b=p_1^{k_{b_1}}p_2^{k_{b_2}}\cdots p_s^{k_{b_s}}$

我们发现，对于 a 和 b 的情况，二者的最大公约数等于

$$p_1^{\min(k_{a_1},k_{b_1})}p_2^{\min(k_{a_2},k_{b_2})}\cdots p_s^{\min(k_{a_s},k_{b_s})}$$

最小公倍数等于

$$p_1^{\max(k_{a_1},k_{b_1})}p_2^{\max(k_{a_2},k_{b_2})}\cdots p_s^{\max(k_{a_s},k_{b_s})}$$

由于 $k_a+k_b=\max(k_a,k_b)+\min(k_a,k_b)$

所以得到结论是 $\gcd(a,b)\times\operatorname{lcm}(a,b)=a\times b$

要求两个数的最小公倍数，先求出最大公约数即可。

多个数

可以发现，当我们求出两个数的 $\gcd$ 时，求最小公倍数是 $O(1)$ 的复杂度。那么对于多个数，我们其实没有必要求一个共同的最大公约数再去处理，最直接的方法就是，当我们算出两个数的 $\gcd$，或许在求多个数的 $\gcd$ 时候，我们将它放入序列对后面的数继续求解，那么，我们转换一下，直接将最小公倍数放入序列即可。

扩展欧几里得算法

扩展欧几里得算法（Extended Euclidean algorithm, EXGCD），常用于求 $ax+by=\gcd(a,b)$ 的一组可行解。

过程

设

$ax_1+by_1=\gcd(a,b)$

$bx_2+(a \bmod b)y_2=\gcd(b,a \bmod b)$

由欧几里得定理可知：$\gcd(a,b)=\gcd(b,a \bmod b)$

所以 $ax_1+by_1=bx_2+(a \bmod b)y_2$

又因为 $a \bmod b=a-(\lfloor \frac{a}{b} \rfloor \times b)$

所以 $ax_1+by_1=bx_2+(a-(\lfloor \frac{a}{b} \rfloor \times b))y_2$

$ax_1+by_1=ay_2+bx_2-\lfloor \frac{a}{b} \rfloor \times by_2=ay_2+b(x_2-\lfloor \frac{a}{b} \rfloor y_2)$

因为 $a=a,b=b$，所以 $x_1=y_2,y_1=x_2-\lfloor \frac{a}{b} \rfloor y_2$

将 x_2,y_2 不断代入递归求解直至 $\gcd$（最大公约数，下同）为 0 递归 $x=1,y=0$ 回去求解。

实现

C++

```cpp
int Exgcd(int a, int b, int &x, int &y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;
  }
  int d = Exgcd(b, a % b, x, y);
  int t = x;
  x = y;
  y = t - (a / b) * y;
  return d;
}
```

Python

```python
def Exgcd(a, b):
    if b == 0:
        return a, 1, 0
    d, x, y = Exgcd(b, a % b)
    return d, y, x - (a // b) * y
```

函数返回的值为 $\gcd$，在这个过程中计算 x, y 即可。

值域分析

$ax + by = \gcd(a, b)$ 的解有无数个，显然其中有的解会爆 long long。
万幸的是，若 $b \neq 0$，扩展欧几里得算法求出的可行解必有 $|x| \leq b, |y| \leq a$。
下面给出这一性质的证明。

证明

- $\gcd(a, b) = b$ 时，$a \bmod b = 0$，必在下一层终止递归。
  得到 $x_1 = 0, y_1 = 1$，显然 $a, b \geq 1 \geq |x_1|, |y_1|$。
- $\gcd(a, b) \neq b$ 时，设 $|x_2| \leq (a \bmod b), |y_2| \leq b$。
  因为 $x_1 = y_2, y_1 = x_2 - \left\lfloor \frac{a}{b} \right\rfloor y_2$
  所以 $|x_1| = |y_2| \leq b, |y_1| \leq |x_2| + |\left\lfloor \frac{a}{b} \right\rfloor y_2| \leq (a \bmod b) + \left\lfloor \frac{a}{b} \right\rfloor |y_2|$
  $\leq a - \left\lfloor \frac{a}{b} \right\rfloor b + \left\lfloor \frac{a}{b} \right\rfloor |y_2| \leq a - \left\lfloor \frac{a}{b} \right\rfloor (b - |y_2|)$
  $a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b \leq a - \left\lfloor \frac{a}{b} \right\rfloor (b - |y_2|) \leq a$
  因此 $|x_1| \leq b, |y_1| \leq a$ 成立。

迭代法编写扩展欧几里得算法

首先，当 $x = 1$，$y = 0$，$x_1 = 0$，$y_1 = 1$ 时，显然有：

$$\begin{cases} ax + by & = a \\ ax_1 + by_1 & = b \end{cases}$$

成立。

已知 $a \bmod b = a - (\lfloor \frac{a}{b} \rfloor \times b)$，下面令 $q = \lfloor \frac{a}{b} \rfloor$。参考迭代法求 gcd，每一轮的迭代过程可以表示为：

$$(a, b) \rightarrow (b, a - qb)$$

将迭代过程中的 a 替换为 $ax + by = a$，b 替换为 $ax_1 + by_1 = b$，可以得到：

$$
\begin{cases} ax + by & = a \\ ax_1 + by_1 & = b \end{cases}
$$
$$
\to \begin{cases} ax_1 + by_1 & = b \\ a(x - qx_1) + b(y - qy_1) & = a - qb \end{cases}
$$

据此就可以得到迭代法求 exgcd。

因为迭代的方法避免了递归，所以代码运行速度将比递归代码快一点。

```
int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0;
  int x1 = 0, y1 = 1, a1 = a, b1 = b;
  while (b1) {
    int q = a1 / b1;
    tie(x, x1) = make_tuple(x1, x - q * x1);
    tie(y, y1) = make_tuple(y1, y - q * y1);
    tie(a1, b1) = make_tuple(b1, a1 - q * b1);
  }
  return a1;
}
```

如果你仔细观察 a_1 和 b_1，你会发现，他们在迭代版本的欧几里德算法中取值完全相同，并且以下公式无论何时（在 while 循环之前和每次迭代结束时）都是成立的：$x \cdot a + y \cdot b = a_1$ 和 $x_1 \cdot a + y_1 \cdot b = b_1$。因此，该算法肯定能正确计算出 $\gcd$。

最后我们知道 a_1 就是要求的 $\gcd$，有 $x \cdot a + y \cdot b = g$。

矩阵的解释

对于正整数 a 和 b 的一次辗转相除即 $\gcd(a, b) = \gcd(b, a \bmod b)$ 使用矩阵表示如

$$
\begin{bmatrix} b \\ a \bmod b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}
$$

其中向下取整符号 $\lfloor c \rfloor$ 表示不大于 c 的最大整数。我们定义变换 $\begin{bmatrix} a \\ b \end{bmatrix} \mapsto \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$。

易发现欧几里得算法即不停应用该变换，有

$$
\begin{bmatrix} \gcd(a, b) \\ 0 \end{bmatrix} = \left(\cdots \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} a \\ b \end{bmatrix}
$$

令

$$
\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} = \cdots \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
$$

那么

$$
\begin{bmatrix} \gcd(a, b) \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}
$$

满足 $a \cdot x_1 + b \cdot x_2 = \gcd(a, b)$ 即扩展欧几里得算法，注意在最后乘了一个单位矩阵不会影响结果，提示我们可以在开始时维护一个 2×2 的单位矩阵编写更简洁的迭代方法如

```cpp
int exgcd(int a, int b, int &x, int &y) {
  int x1 = 1, x2 = 0, x3 = 0, x4 = 1;
  while (b != 0) {
    int c = a / b;
    std::tie(x1, x2, x3, x4, a, b) =
        std::make_tuple(x3, x4, x1 - x3 * c, x2 - x4 * c, b, a - b * c);
  }
  x = x1, y = x2;
  return a;
}
```

这种表述相较于递归更简单。

应用

- 10104 - Euclid Problem
- GYM - (J) once upon a time
- UVa - 12775 - Gift Dilemma

参考资料与链接

---

1. libstdc++: std Namespace Reference ↩
