2025/7/7 课程 第一部分

数论基础和线性同余方程

陈景泰

2025-07-06

1. 数论基础

我们从一个简单的问题入手：

最大公因数 问题 1

给出两个正整数 a, b，求出正整数 d，使得它是 a 和 b 的最大公因数（也就是求最大的 d，满足 $d \mid a$ 且 $d \mid b$）。例如，12 和 18 的最大公因数是 6。我们一般记作 $d = \gcd(a, b)$。另外，如果 $\gcd(a, b) = 1$，我们称 a 和 b 互质。

另外的，我们给出最小公倍数的定义：求最小的 d，使其同时是 a 和 b 的倍数。我们一般记作 $d = \operatorname{lcm}(a, b)$。

这个问题是最经典的数学问题之一。接下来，我们会以这个问题为起点，讨论第一部分的所有内容。

1.1.1 从简单开始：相减法

对于最大公因数问题，假设 $a > b$，且 d 是 a 和 b 的最大公因数。接下来不妨尝试论证：d 同时也是 b 和 $a - b$ 的最大公因数。

证明：相减法 证明 2

这个证明分为两步：d 能同时被 b 和 $a - b$ 整除；d 是满足这一条件下的最大正整数。

1. 我们已经知道了 a 和 b 都是 d 的倍数，那么 $a - b$ 自然也是 d 的倍数。第一步的条件就自然满足了。
2. 不妨使用反证法。假设存在一个正整数 $d' > d$ 使得 d' 能同时被 b 和 $a - b$ 整除。此时 b 和 $a - b$ 都是 d' 的倍数，那么 a 自然也是 d' 倍数，那么 d' 就是 a 和 b 的公因数。此时我们知道 d 是 a 和 b 的最大公因数，那么理应有 $d' \leq d$，矛盾！

根据相减法，我们就得到了求最大公因数最基本的流程：

cpp

```cpp
int gcd(int a, int b) {
  if (a == b) return a;   // 这里选用的终止条件是 a = b
  if (a < b)  swap(a, b);
  return gcd(b, a - b);
}
```

这个做法看上去比枚举 d 并判断是否整除 a 和 b 快多了，但是实际上，这个做法的时间复杂度为 $\mathcal{O}(\max(a,b))$。不妨考虑下面的例子：$a=10^9, b=1$，此时我们会进行 10^9-1 次减法，在最后得到 $a=b=1$ 时才可以确认 $\gcd(a,b)=1$！我们显然不能接受和暴力跑的差不多快的算法，那么不妨考虑进行优化。

1.1.2 向前一小步：欧几里得算法

优化的突破口在于：当 a 比 b 大非常多的时候，我们需要耐住性子从 a 中逐次减 b，直到 $a < b$ 为止。

可以注意到，这个工作可以使用取模代劳，每次将 a 赋值为 $a \bmod b$ 就好了！这个操作实际上等价于前面提到的整个流程，不过相较于前面给出的代码，此时的终止条件有一点点不一样：

终止条件 说明 3

可以注意到前面的代码中给出的递归终止条件为：`if (a == b) return a;`

辗转相减法在计算过程中可以保证 $a, b > 0$，所以使用的性质是 $\gcd(a, a) = a$。而这里，$a \bmod b$ 有可能等于 0！此时我们可以使用另一个性质：$\gcd(a, 0) = a$。

这个性质虽然有点跳出了最大公约数的定义，不过在扩充这个定义后，后续的实现会方便很多。

考虑到每次取模后会满足 $a \bmod b < b$，那么从宏观的角度看，整个计算流程就是：将 a 赋值为 $a \bmod b$，赋值后有 $a < b$，所以需要交换一次，再进行下一次取模、交换、取模、交换……这实际上可以看作两个数字互相取模。

我们一般称这个方法为辗转相除法，又叫欧几里得算法。以下是一个粗略的代码实现：

cpp

```cpp
int gcd(int a, int b) {
  if (a > b)  swap(a, b);
  if (b == 0) return a;
  return gcd(a % b, b);
}
```

这个代码其实并不是欧几里得算法的最终形态。首先可以发现，最开始的交换其实是可以通过参数处理的。根据前面的讨论，我们每次取模后都会交换二者，那么这个交换可以直接通过“交换递归时的参数”实现，也就是：

cpp

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

让我们进一步使用三元表达式重写：

cpp

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

这就是大多数人经常使用的欧几里得算法实现。

欧几里得算法的时间复杂度为 $\mathcal{O}(\log \max(a, b))$。

证明：欧几里得算法的时间复杂度 证明 4

在计算 $\gcd(a, b)$ 时，存在两个分支：

- 如果 $a < b$，那么会递归到 $\gcd(b, a \bmod b)$，也就是 $\gcd(b, a)$，这只是交换了两个参数而已；
- 如果 $a \geq b$，那么考虑令 $a' = a \bmod b$，我们可以证明 $a' < \frac{a}{2}$。考虑反证法，如果此时 $a' \geq \frac{a}{2}$，那么 $0 \leq 2a' - a < a'$，而根据取模的性质，我们知道 $(a - a') \bmod b = 0$，那么

$$2a' - a \bmod b = a' + (a' - a) \bmod b = a' \bmod b = a \bmod b$$

我们知道 $2a' - a$ 作为比 a' 还要小的非负整数，对 b 取模的结果居然和 a 相同，那么 a' 自然不会等于 $a \bmod b$，矛盾！

我们在调用一次 $\gcd(a, b)$ 的时候，程序首先有可能会进入第一个分支，随后就会一直留在第二个分支，此时 a 和 b 都在轮流折半，算出解自然就只需要 $\mathcal{O}(\log \max(a, b))$ 步。

在求出 a 和 b 的最大公约数后，我们不妨考虑解决下一个问题：

“扩展最大公因数问题” 问题 5

对于正整数 a, b，求出满足如下条件的任意一组整数解 (x, y)：

$$ax + by = \gcd(a, b)$$

这个问题要是直接思考的话会非常困难，不过其实标题已经剧透了做法——通过修改欧几里得算法的代码，我们可以在算出最大公因数的同时得到任意一组整数解！

考虑欧几里得算法的代码中遇到的两种情况：

1. $a \neq 0, b = 0$，此时我们可以确认最大公约数就是 a，同时也直接得到了一组解：$(x, y) = (1, 0)$，因为

$$a \times 1 + b \times 0 = a + 0 = a = \gcd(a, 0)$$

2. $a \neq 0, b \neq 0$，此时令 $f = \lfloor \frac{a}{b} \rfloor, r = a \bmod b$。我们通过递归法得到了 b 和 r 的最大公约数 d，并且可以顺便得到一组满足 $bx' + ry' = d$ 的解。我们知道

$$a = b \times \left\lfloor \frac{a}{b} \right\rfloor + (a \bmod b) = fb + r$$

代入到 $d = bx' + ry'$ 的式子中可以得到：

$$d = bx' + ry' = bx' + (a - fb)y' = ay' + b(x' - fy')$$

从上面的式子中可以得到一个非常符合直觉的解：

$$\begin{cases} x = y' \\ y = x' - fy' \end{cases}$$

我们就可以通过 $\gcd(b, r)$ 的计算结果得到 $\gcd(a, b)$ 的计算结果。

上面的算法流程就被称为扩展欧几里得算法。让我们把思路整合一下，给出如下代码：

cpp

```cpp
struct Result {
  int d, x, y;
};

Result exgcd(int a, int b) {
  if (b == 0) return {a, 1, 0};
  Result res = exgcd(b, a % b);
  int f = a / b;
  return {res.d, res.y, res.x - f * res.y};
}
```

当然，如果你比较习惯 C++ 的引用特性的话，可以改写为如下形式：

cpp

```cpp
int exgcd(int a, int b, int &x, int &y) {
  if (b == 0) {
    x = 1, y = 0;
    return a;
  }
  // 需要注意，这里 x 和 y 互换了，所以此时 x = y' 和 y = x'
  int d = exgcd(b, a % b, y, x);
  // 解需要满足 x = y', y = x' - (a / b) * y'，所以只需要更改 y
  y -= (a / b) * x;
  return d;
}
```

可以证明 x 和 y 在整个计算过程中都不会超出数据范围，并且在最后有 $-b \leq x \leq b$ 和 $-a \leq y \leq a$。证明依然可以使用递归的方式，这里就不再展开了。

gcd 和 lcm 之间存在一些经典的性质，这里列举一部分：

gcd 和 lcm 的部分性质 定理 6

- 对于正整数 d，只要 a 和 b 都是 d 的倍数（公因数），就有 d 是 $\gcd(a,b)$ 的因数。
- 对于正整数 d，只要 d 同时是 a 和 b 的倍数（公倍数），就有 d 是 $\mathrm{lcm}(a,b)$ 的倍数。
- 如果 $a \times b$ 是 c 的倍数，但是 $\gcd(b,c)=1$，那么 a 就是 c 的倍数。
- $\gcd(a,b) \times \mathrm{lcm}(a,b) = a \times b$。在代码中，我们一般使用 `a / gcd(a, b) * b` 计算两个数的最小公倍数，避免直接计算 `a * b` 带来的整数溢出。
- 对任意整数 r，有 $\gcd(a+rb,b)=\gcd(a,b)$。这个性质对 lcm 不成立。

前四条性质都可以通过质因数分解后对每个质因子的指数分析得到，证明就不展开了。

欧拉函数的定义如下：

互质计数 问题 7

给出一个正整数 n，令欧拉函数 $\varphi(n)$ 等于满足 $1 \le i \le n$ 且 $\gcd(i, n) = 1$ 的整数 i 的个数。

以下是一些常见的欧拉函数值：

n	满足条件的 i	$\varphi(n)$
6	1, 5	2
12	1, 5, 7, 11	4
16	1, 3, 5, 7, 9, 11, 13, 15	8
24	1, 5, 7, 11, 13, 17, 19, 23	8

接下来简单介绍一些欧拉函数的性质。

- 对于质数 p，我们有 $\varphi(p) = p - 1$。

证明 证明 8

所有小于 p 的正整数都和 p 互质，数量自然就是 $p - 1$。

- 对于质数 p 和正整数 k，我们有 $\varphi\left(p^k\right) = p^k - p^{k-1}$。

证明 证明 9

一个正整数 i 和 p^k 不互质实际上等价于 i 是 p 的倍数。考虑 $[1, p^k]$ 内的数字，其中 p 的倍数有 p^{k-1} 个，那么 $\varphi\left(p^k\right)$ 等于区间内的数字个数减去区间内和 p^k 不互质的数字个数，也就是 $p^k - p^{k-1}$。

- 对于两个互质的正整数 a, b，我们有 $\varphi(ab) = \varphi(a)\varphi(b)$。

证明

证明 10

这个性质等价于说明 φ 是一个积性函数。对应的，这个证明将会在第二部分补充。

- 考虑到 n 存在唯一的质因数分解，假设质因数分解的结果为：

$$n = p_1^{a_1} p_2^{a_2} ... p_k^{a^k}$$

那么有

$$\varphi(n) = \varphi(p_1^{a_1})\varphi(p_2^{a_2})...\varphi(p_k^{a_k}) = \left(p_1^{a_1} - p_1^{a_1-1}\right)\left(p_2^{a_2} - p_2^{a_2-1}\right)...\left(p_k^{a_k} - p_k^{a_1-k}\right)$$

$$= \left(p_1^{a_1} \times \frac{p_1 - 1}{p_1}\right)\left(p_2^{a_2} \times \frac{p_2 - 1}{p_2}\right)...\left(p_k^{a_k} \times \frac{p_k - 1}{p_k}\right) = n \times \frac{p_1 - 1}{p_1} \times \frac{p_2 - 1}{p_2} \times ... \times \frac{p_k - 1}{p_k}$$

不妨验证一下：

n	分解结果	$\varphi(n)$
6	$6 = 2^1 \times 3^1$	$6 \times \frac{2-1}{2} \times \frac{3-1}{3} = 2$
12	$12 = 2^2 \times 3^1$	$12 \times \frac{2-1}{2} \times \frac{3-1}{3} = 4$
16	$16 = 2^4$	$16 \times \frac{2-1}{2} = 8$
24	$24 = 2^3 \times 3^1$	$24 \times \frac{2-1}{2} \times \frac{3-1}{3} = 8$

有关欧拉函数，存在一个非常好用的定理：

欧拉定理 定理 11

对于正整数 $a, n > 1$，只要有 $\gcd(a, n) = 1$，就可以得到

$$a^{\varphi(n)} \bmod n = 1$$

证明涉及到代数结构的一些知识，以下提供一个理解上稍微简单一点的证明。

证明：欧拉定理

证明 12

我们知道在 $[1,n]$ 内和 n 互质的数字有 $\varphi(n)$ 个，不妨假设为 $B=\left\{b_1,b_2,...,b_{\varphi(n)}\right\}$。

随后考虑数组 $A=\left\{ab_1 \bmod n, ab_2 \bmod n, ..., ab_{\varphi(n)} \bmod n\right\}$，我们发现：

- 数组 A 的所有元素和 n 都互质。这是因为有 $\gcd(a,n)=1$ 且 $\gcd(b_i,n)=1$，从而得到 $\gcd(ab_i,n)=1$；
- 数组 A 中的元素两两不同。使用反证法，不妨假设存在 $i\neq j$ 使得 $ab_i \bmod n = ab_j \bmod n$。此时我们知道 $a(b_i-b_j)$ 是 n 的倍数。考虑到 a 和 n 互质，可以发现 b_i-b_j 应该是 n 的倍数，但是这是不可能实现的，矛盾！

那么数组 A 就包含了和 n 互质的 $\varphi(n)$ 个互不相同的数字。不难发现，数组 B 的元素在数组 A 中都出现了恰好一次。那么理论上，两个数组的乘积是相同的，可以得到：

$$
\begin{aligned}
& n \mid \left[(ab_1)(ab_2)...\left(ab_{\varphi(n)}\right)-b_1b_2...b_{\varphi(n)}\right] \\
\Rightarrow\ & n \mid \left(a^{\varphi(n)}-1\right)\left(b_1b_2...b_{\varphi(n)}\right) \\
\Rightarrow\ & n \mid \left(a^{\varphi(n)}-1\right) \Rightarrow a^{\varphi(n)} \bmod n = 1
\end{aligned}
$$

这个定理有一个非常特殊的形式：

费马小定理 定理 13

只需要将 p 代入到欧拉定理的 n 中，并根据 $\varphi(p) = p - 1$ 就可以得到如下定理：

- 对于正整数 a 和质数 p，有 $a^{p-1} \bmod p = 1$。

乘法逆元的定义如下：

乘法逆元 定义 14

对于正整数 $1 \le a < n$，称符合以下条件的数字 b 为 a 模 n 的逆元，记作 a^{-1}：

$$ab \bmod n = 1$$

首先可以知道，只有在 $\gcd(a, n) = 1$ 的时候，乘法逆元才有可能存在。其次，通过欧拉定理可以知道 $a^{\varphi(n)} \bmod n = 1$。又因为 $\gcd(a, n) = 1$，我们可以得到此时 b 的唯一解：$b = a^{\varphi(n)-1}$。

进一步的，我们把质数 p 代入 n 中，就可以得到一个重要的性质：

质数下的乘法逆元 定理 15

对于质数 p 和整数 $1 \le a < p$，a 模 p 的乘法逆元等于 $a^{\varphi(p)-1} = a^{p-2}$。

这个性质被经常用于乘法逆元的计算中，因为一般题目给出的模数 `mod` 都是质数，那么只需要调用 `qpow(a, mod - 2)` 就可以得到逆元了。

当然，你也可以使用扩展欧几里得算法求解，只需要列出如下方程：

$$\underbrace{a}_{a}\underbrace{b}_{x}+\underbrace{n}_{b}\underbrace{k}_{y}=\gcd(a,n)=1$$

那么只需要求算 `d = exgcd(a, n, b, k)` 后，在 `d == 1` 的前提下，得到的 `b` 就是对应的乘法逆元。

这两种方式都可以在 $\mathcal{O}(\log n)$ 的时间复杂度下得到一个数字的乘法逆元，但是在实际上，存在其他的一些算法，用于对乘法逆元的线性计算。这种做法一般都是在乘法逆元的计算量达到 10^7 量级的时候使用的，在一般的算法竞赛题目上并不常见。具体的做法可以参考 OI Wiki 上的介绍：https://oi-wiki.org/math/number-theory/inverse/

2. 同余方程

一般的同余方程都遵循如下格式：

同余方程 定义 16

假设有自变量 x 和正整数 n，且函数 $f(x) = a_0x^0 + a_1x^1 + ... + a_kx^k$ 是关于自变量 x 的整系数多项式（也就是 a_i 都是整数），那么称如下方程为模 n 的同余方程：

$$f(x) \equiv 0 \pmod{n}$$

这等价于 $f(x)$ 的值是 n 的倍数。一个整数 x_0 是上述同余方程的解，当且仅当 $f(x_0)$ 是 n 的倍数。

求解任意的同余方程还是过于困难，于是我们只需要聚焦于其中一类同余方程：

线性同余方程 定义 17

如果 $f(x)$ 是一个最高一次的函数，写作 $f(x) = ax - b$，那么我们可以得到如下同余方程：

$$ax - b \equiv 0 \pmod{n} \text{ 或者 } ax \equiv b \pmod{n}$$

这就是线性同余方程的形式，其中第二种形式代表的是 ax 和 b 模 n 同余，也就是在对 n 取模后的结果相同。

其实，大家对线性同余方程已经不陌生了。我们实际上已经解决了一个线性同余方程问题——乘法逆元问题。只需要将其表述稍作更改：

乘法逆元 (2) 定义 18

对于正整数 $1 \leq a < n$，称如下线性同余方程的解为 a 模 n 的逆元，记作 a^{-1}：

$$ax \equiv 1 \pmod{n}$$

让我们明确问题：找到所有满足 $ax \equiv b \pmod{n}$ 的整数 x。我们首先证明：这个同余方程有解，当且仅当 b 是 $\gcd(a, n)$ 的倍数。在接下来的证明中，我们可以顺便得到求出一组解的方案。

证明：线性同余方程解的存在性 证明 19

假设 $d = \gcd(a, n)$。首先我们知道，a 和 n 都是 d 的倍数，那么对于任意的整数 x，$ax \bmod n$ 应该都是 d 的倍数。所以在 b 不是 d 的倍数的时候，自然就不可能有解。

接下来我们证明：在 b 是 d 的倍数时，线性同余方程一定有解。事实上，考虑通过扩展欧几里得算法求出如下方程的任意一组解：

$$ax' + ny' = \gcd(a, n) = d$$

然后将左右两侧同时乘以整数 $\frac{b}{d}$，且对 n 取模，就可以得到：

$$a\left(\frac{b}{d}x'\right) \bmod n = \left[a\left(\frac{b}{d}x'\right) + n\left(\frac{b}{d}y'\right)\right] \bmod n = \left(\frac{b}{d} \times d\right) \bmod n = b \bmod n$$

我们就得到了一个解：$x = \frac{b}{d}x'$。

这样，我们就在证明的同时，给出了一个基于扩展欧几里得算法的构造。那么进一步的，我们可以继续计算通解。

线性同余方程的通解 说明 20

假设我们通过前面的方式得到了一个解 x_0，我们可以发现，方程的所有解形如 $x = x_0 + x_c$，其中 x_c 只需要满足 $ax_c \bmod n = 0$。

设 $d = \gcd(a, n)$，那么以上条件等价于 $\left(\frac{a}{d}\right)x_c \equiv 0 \left(\bmod \frac{n}{d}\right)$。此时 $\frac{a}{d}$ 和 $\frac{n}{d}$ 互质，那么方程成立当且仅当 x_c 是 $\frac{n}{d}$ 的倍数。代回后可以得到解的通项式：

$$x = x_0 + \frac{nk}{\gcd(a, n)} \quad \text{其中 } k \text{ 为任意整数}$$

我们据此可以得到最小的非负整数解（令 $t = \frac{n}{\gcd(a,n)}$）：

$$x = (x_0 \bmod t + t) \bmod t$$

3. 中国剩余定理

顾名思义，线性同余方程组就是将一些线性同余方程组合起来。不过为了严谨，我们还是给出一个定义：

线性同余方程组 定义 21

对于一个包含 k 个方程的线性同余方程组，其可以表示为如下形式：

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \cdots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

其中 n_i 都是正整数，而 a_i 是整数。

在给出定义后，我们不妨研究一个简单的案例。

中国剩余的定理的名字起源于如下事实：

物不知数 问题 22

线性同余方程组的求解问题在《孙子算经》中首次出现。这个问题一般通过「物不知数」的方式描述。一个经典的案例如下：

- 有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二。问物几何？

也就是求算如下线性同余方程的解：

$$\begin{cases} x \equiv 2 \pmod 3 \\ x \equiv 3 \pmod 5 \\ x \equiv 2 \pmod 7 \end{cases}$$

上面具体问题的解答口诀由明朝数学家程大位在《算法统宗》中给出：

- 三人同行七十希，五树梅花廿一支，七子团圆正半月，除百零五便得知。

换做数学方式就是：对于如下线性同余方程组：

$$\begin{cases} x \equiv a_1 \pmod 3 \\ x \equiv a_2 \pmod 5 \\ x \equiv a_3 \pmod 7 \end{cases}$$

可以立刻得到一个解：

$$x = (70a_1 + 21a_2 + 15a_3) \bmod 105$$

这里 105 的存在比较直观，毕竟这是三个模数的乘积，但是剩余的三个数字看上去就比较让人捉不着头脑了。我们不妨将这三个数字分别对三个模数取模：

$$\begin{cases} 70 \bmod 3 = 1 \\ 70 \bmod 5 = 0 \\ 70 \bmod 7 = 0 \end{cases} \quad \begin{cases} 21 \bmod 3 = 0 \\ 21 \bmod 5 = 1 \\ 21 \bmod 7 = 0 \end{cases} \quad \begin{cases} 15 \bmod 3 = 0 \\ 15 \bmod 5 = 0 \\ 15 \bmod 7 = 1 \end{cases}$$

我们可以发现：a_i 对应的系数只在对第 i 个模数取模时等于 1，其余情况都等于 0。不妨以 5 举例，我们可以得到

$$x \bmod 5 = (70a_1 + 21a_2 + 15a_3) \bmod 5 = (0a_1 + 1a_2 + 0a_3) \bmod 5 = a_2 \bmod 5$$

因为上面的性质，在对 5 取模的时候，a_1 和 a_3 的系数都变成了 0，自然被消去，而只有 a_2 的系数是 1，这样就完美符合了 $x \equiv a_2 \pmod 5$ 的要求。

那么不妨如法炮制，对于一般的线性同余方程组：

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \dots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

我们对所有 $1 \le i \le k$ 构造 c_i，使得

$$c_i \bmod n_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

那么根据上述的分析，我们就可以得到一个解：

$$x = a_1c_1 + a_2c_2 + \dots + a_kc_k$$

中国剩余定理的核心就在于 c_i 的构造。

中国剩余定理对于模数有一个前提：

前提 说明 23

可以注意到，如果模数不互质的话，c_i 有可能不存在，例如在所有模数都相同的时候。所以在接下来的讨论中，我们需要保证所有的模数两两互质，也就是对任意的 $i \neq j$ 有 $\gcd(n_i, n_j) = 1$。

我们对于 c_i，需要满足

$$c_i \bmod n_j = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

此时求出除了 n_i 外所有模数的乘积，也就是

$$M_i = n_1 \times n_2 \times \ldots \times n_{i-1} \times n_{i+1} \times \ldots \times n_k = \prod_{j \neq i} n_j$$

那么对 c_i 的约束就可以改写为：

$$\begin{cases} c_i \equiv 1 \pmod{n_i} \\ c_i \equiv 0 \pmod{M_i} \end{cases}$$

这样写可能并不直观，我们不妨换一个方式。第二个条件告诉我们 c_i 是 M_i 的倍数，那么假设 $c_i = xM_i$，代入到第一个条件后可以得到

$$xM_i \equiv 1 \pmod{n_i}$$

这样就回到了我们最熟悉的线性同余方程的形式了。进一步的，我们还可以发现：这个线性同余方程满足了乘法逆元的性质！换句话说，由于 $\gcd(M_i, n_i) = 1$，那么 x 就是 M_i 模 n_i 的逆元。

根据这一性质，我们终于可以写出算法全流程：

- 计算所有模数的乘积 M，那么 $M_i = \frac{M}{n_i}$；
- 对于所有 i，计算 x_i 为 M_i 模 n_i 的逆元，和 M_i 相乘后得到 c_i；
- 将 $a_i c_i$ 相加，即可得到方程的一个解。你也可以在中途对 M 取模，因为一个解在加上 M 的倍数后也一定还是原方程组的解。

由于 n_i 可能不是质数，使用欧拉定理时 $\varphi(n_i)$ 的计算会有些棘手，所以这里使用扩展欧几里得算法更优。

据此得到如下代码：

```cpp
int CRT(int k, int a[], int n[]) {
  int ans = 0, M = 1;
  for (int i = 1; i ⩽ k; i ++)
    M *= n[i];
  for (int i = 1; i ⩽ k; i ++) {
    int M_i = M / n[i];
    int x_i, t;
    exgcd(M_i, n_i, x_i, t);
    int c_i = x_i * M_i;
    ans += a[i] * c_i;
    ans = (ans % M + M) % M;
  }
  return ans;
}
```

这份代码的时间复杂度是 $\mathcal{O}(k \log V)$，其中 V 代表的是模数的值域（也可以理解成模数的最大值可能值）。不过，还有一种方法，可以在同样的时间复杂度内解决这个问题，还不需要模数两两互质！这个算法实际上还是来源于出现了很多次的好朋友：扩展欧几里得算法。

假设我们有包含两个线性同余方程的方程组：

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \end{cases}$$

根据这两个线性同余方程，不妨假设有整数 p, q 满足

$$\begin{cases} x = pn_1 + a_1 \\ x = qn_2 + a_2 \end{cases}$$

那么有

$$\begin{aligned} (pn_1 + a_1) - (qn_2 + a_2) &= 0 \\ pn_1 - qn_2 &= a_2 - a_1 \end{aligned}$$

让我们回忆在讨论线性同余方程时使用的做法。考虑设 $d = \gcd(n_1, n_2)$，那么在 $a_2 - a_1$ 不是 d 的倍数时自然是无解的。此时，我们通过扩展欧几里得算法随便找到如下方程的整数解：

$$n_1 p' + n_2 q' = \gcd(n_1, n_2) = d$$

那么我们就可以对应得到 (p, q) 的一组可行解：

$$\begin{cases} p = \frac{a_2 - a_1}{d} p' \\ q = -\frac{a_2 - a_1}{d} q' \end{cases}$$

可以据此算出 x 的一个可能值：$x_0 = pn_1 + a_1$。那么按道理来说，这个线性同余方程组的解可以写作

$$x = x_0 + x_c$$

此时的 x_c 只需要同时是 n_1 和 n_2 的倍数，根据最小公倍数的定义，这等价于 x_c 是 $\operatorname{lcm}(n_1, n_2)$ 的倍数。我们就可以进一步将解写成

$$x = x_0 + k \operatorname{lcm}(n_1, n_2) \quad \text{其中 } k \text{ 为任意整数}$$

或者说——

$$x \equiv x_0 \pmod{\operatorname{lcm}(n_1, n_2)}$$

我们就得到了：

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \end{cases} \Leftrightarrow x \equiv x_0 \pmod{\operatorname{lcm}(n_1, n_2)}$$

这样就能够实现将两个线性同余方程合并为一个线性同余方程。

考虑到我们可以将两个方程合并为一个，那么对于一般的线性同余方程组，我们只需要将第一个方程和第二个方程合并为一个，再将其和第三个方程合并为一个，以此类推，就可以在最后得到一个线性同余方程，而这就是原方程组的解！我们也可以在中途判断无解的情况。

以下是对应的实现：

cpp

```cpp
int exCRT(int k, int a[], int n[]) {
  int a1 = a[1], n1 = n[1];
  for (int i = 2; i ⩽ k; i ++) {
    int a2 = a[i], n2 = n[i];
    int p_, q_;
    int d = exgcd(n1, n2, p_, q_);
    if (abs(a1 - a2) % d ≠ 0) {
      // 无解，返回 -1
      return -1;
    }
    int p = (a2 - a1) / d * p_;
    a1 = p * n1 + a1;
    n1 = n1 / gcd(n1, n2) * n2;
    a1 = (a1 % n1 + n1) % n1;
  }
  return a1;
}
```

可以发现，这份代码的时间复杂度同样是 $\mathcal{O}(k \log V)$，而且比原版中国剩余定理更通用。尽管如此，中国剩余定理还是具有其优势的，例如：

- 代码相对而言更短；
- 乘除法使用次数更少，对高精度算法更友好

直到现在，中国剩余定理在密码学中经常会用到，可用于破解 DHKE 和 RSA 算法。同时，它也可以和其他算法结合，通过计算简单的余数问题合并出复杂的余数答案。具体的形式会在第三部分提到。

洛谷 P3868 [TJOI2009] 猜数字

习题 24

现有两组数字，每组 k 个。

第一组中的数字分别用 $a_1, a_2, ..., a_k$ 表示，第二组中的数字分别用 $b_1, b_2, ..., b_k$ 表示。

其中第二组中的数字是两两互素的。求最小的非负整数 n，满足对于 $\forall i \in [1, k]$，有 $b_i \mid (n - a_i)$。

数据满足 b_i 的乘积不超过 10^{18}。

这一问题其实就是中国剩余定理的模板题，不过需要注意的是，即使模数乘积在 `long long` 范围内，中国剩余定理的中间结果完全有可能超出 `long long` 的范围。你需要使用倍增加法或者 `int128` 处理中间结果。

4. 扩展内容

对于正整数 a 和质数 p，我们用记号 $\nu_p(a)$ 表示 a 的质因数分解中 p 的指数。或者说，a 是 $p^{\nu_p(a)}$ 的倍数，而不是 $p^{\nu_p(a)+1}$ 的倍数。例如，$\nu_2(12)=2, \nu_3(12)=1, \nu_5(12)=\nu_7(12)=0$。

对于两个正整数 a, b，a 是 b 的因子等价于：对于任意质数 p，都有 $\nu_p(a) \le \nu_p(b)$，也就是 a 对应的指数总是不超过 b 对应的指数。

对于两个正整数 a, b，假设 $g=\gcd(a,b), l=\operatorname{lcm}(a,b)$，那么对于任意的质数 p，都有

$$\nu_p(g)=\min\big(\nu_p(a),\nu_p(b)\big), \quad \nu_p(l)=\max\big(\nu_p(a),\nu_p(b)\big)$$

这就是在质因数分解层面，gcd 和 lcm 的本质：对每个质数，gcd 将对应的两个指数取较小值，而 lcm 则是取较大值。

前面的四个定理就可以和一些数学常识联系上（原因请自行思考）：

- 对于正整数 d，只要 a 和 b 都是 d 的倍数（公因数），就有 d 是 $\gcd(a,b)$ 的因数。
  - 如果 $x \le y, x \le z$，那么 $x \le \min(y,z)$。
- 对于正整数 d，只要 d 同时是 a 和 b 的倍数（公倍数），就有 d 是 $\operatorname{lcm}(a,b)$ 的倍数。
  - 如果 $x \ge y, x \ge z$，那么 $x \ge \max(y,z)$。
- 如果 $a \times b$ 是 c 的倍数，但是 $\gcd(b,c) = 1$，那么 a 就是 c 的倍数。
  - 对于非负整数 x, y, z，如果 $x + y \ge z$，并且 $\min(y,z) = 0$，那么 $x \ge z$。（分类讨论谁等于 0 即可）
- $\gcd(a,b) \times \operatorname{lcm}(a,b) = a \times b$。
  - $\min(x,y) + \max(x,y) = x + y$。

2025/7/7 课程 第二部分

筛法和积性函数

陈景泰

2025-07-06

1. 筛法

让我们明确筛法希望解决的问题：

筛质数问题 问题 1

给定正整数 n，求出所有在 $[1, n]$ 范围内的质数。例如，在 $n = 30$ 时，就需要得到

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29$$

最简单的方法就是枚举每个数字，并判断这个数字是不是质数。具体的代码如下：

cpp

```cpp
bool isPrime(int x) {
  for (int i = 2; i * i ⩽ x; i ++) {
    if (x % i == 0) return false;
  }
  return true;
}

vector<int> naive(int n) {
  vector<int> res;
  for (int i = 2; i ⩽ n; i ++)
    if (isPrime(i)) res.push_back(i);
  return res;
}
```

这个方法显然是 $\mathcal{O}(n\sqrt{n})$ 的，在实际运行下并不够优秀。接下来，让我们通过三个步骤，将这一时间复杂度从 $\mathcal{O}(n\sqrt{n})$ 减小到 $\mathcal{O}(n\log n)$，随后减小到 $\mathcal{O}(n\log\log n)$，最后减小到 $\mathcal{O}(n)$，也就是线性。

我们从最简单的筛法说起。筛法顾名思义，就是在过程中筛掉 1 和合数，剩下就只有质数了。一个简单的方式如下：

简单筛法 定义 2

考虑到合数一定能够写成两个整数 $1 < p, q \leq n$ 的乘积，而质数不可以。

我们对于每个 $1 < p \leq n$，定义“p-筛子”将一堆数字中形如 $2p, 3p, ..., kp, ...$ 的数字筛出。最初的一堆数字是在 $[1, n]$ 内的所有整数，在按顺序经过 2-筛子、3-筛子、4-筛子、……、n-筛子之后，再将 1 取出，剩下的就是所有的质数了。

具体的代码如下：

cpp

```cpp
bool alive[N + 10];

vector<int> simpleSieve(int n) {
  vector<int> res;
  for (int i = 2; i ⩽ n; i ++)
    alive[i] = true;
  for (int p = 2; p ⩽ n; p ++) {
    // p-筛子
    for (int i = 2 * p; i ⩽ n; i += p)
      alive[i] = false;
  }
  for (int i = 2; i ⩽ n; i ++)
    if (alive[i]) res.push_back(i);
  return res;
}
```

让我们讨论一下这个算法的复杂度。在通过 p-筛子的时候，我们可以得到其中 $\left\lfloor \frac{n}{p} \right\rfloor - 1$ 个数字是合数。那么总的计算量可以看作

$$\sum_{p=2}^{n} \left\lfloor \frac{n}{p} \right\rfloor$$

我们可以证明这个式子是 $\mathcal{O}(n \log n)$ 的。事实上，这个级数和经典的调和级数相关。

我们称调和级数为满足如下形式的式子：

$$f(n)=\sum_{i=1}^{n}\frac{1}{i}$$

不难发现，为了证明计算量是 $\mathcal{O}(n\log n)$ 的，只需证明 $f(n)=O(\log n)$。

我们知道

$$\begin{aligned}
&1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8}+\ldots+\frac{1}{2^k-1}\\
\le&\underbrace{1}_{\llcorner}+\underbrace{\frac{1}{2}+\frac{1}{2}}+\underbrace{\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}}+\underbrace{\frac{1}{8}+\ldots+\frac{1}{2^{k-1}}}\\
\le&\underbrace{1+1+1+\ldots+1}_{k}=k
\end{aligned}$$

这告诉我们 $f\left(2^k-1\right)\le k$，例如 $f(7)\le 3, f(15)\le 4$。那么对于任意的正整数 n 都有

$$f(n)\le f\left(2^{\lfloor\log n\rfloor+1}-1\right)\le \lfloor\log n\rfloor+1$$

第一步放缩的形式如下：

n	1	2	3	4	5	6	7	8	9	10
$2^{\lfloor\log n\rfloor+1}-1$	1	3	3	7	7	7	7	15	15	15

我们就证明了 $f(n)\le\lfloor\log n\rfloor+1$，根据增长率的定义，我们就有 $f(n)=\mathcal{O}(\log n)$。当然，除了得到 $f(n)$ 的上界外，使用类似的放缩形式，我们也可以得到 $f(n)$ 的下界 $\frac{1}{2}\log n$，从而得到 $f(n)=\Omega(\log n)$，不过这些都是后话了。

我们就走出了筛法的第一步，将时间复杂度降低到了 $\mathcal{O}(n\log n)$。让我们进入第二步——埃拉托斯特尼筛法。

其实只需要注意到一个很小的细节，就可以实现进一步的优化。

埃拉托斯特尼筛法的原理 说明 3

我们注意到，一个合数总是可以写成 $p \times q$ 的形式，其中 p 是一个质数，而 $q > 1$。当然，质数是无法写成这个形式的。可以发现，我们此时没有必要使用所有的筛子，而只需要使用素数-筛子即可，也就是对每个质数 $2 \leq p \leq n$，让数字经过一次 p-筛子。

这个想法存在一个很小的问题：我们现在需要筛质数，但是现在又需要得到所有质数才能知道要经过哪些筛子，这不是形成了一个死局吗？实则不然。

我们考虑到 p-筛子只能筛出比 p 还要大的数字，那么我们从小到枚举数字的时候，只要这个数字还没有被筛出，那么后续的筛子也不能将它筛出，这个数就一定是质数。这样我们就能够从小到达确定质数并进行筛法了。

根据这个原理，我们可以写出如下代码：

cpp

```cpp
bool alive[N + 10];

vector<int> EratosthenesSieve(int n) {
  vector<int> res;
  for (int i = 2; i ⩽ n; i ++)
    alive[i] = true;
  for (int p = 2; p ⩽ n; p ++) if (alive[p]) {
    // 此时可以确定 p 是质数
    res.push_back(p);
    // p-筛子
    for (int i = 2 * p; i ⩽ n; i += p)
      alive[i] = false;
  }
  return res;
}
```

可以证明，这个筛法的时间复杂度是 $\mathcal{O}(n \log \log n)$。这一证明实在有些复杂，在此略去。作为提示，这个证明需要利用一个数论中很重要的结论：在 $[1, n]$ 范围内的质数个数是 $O\left(\frac{n}{\log n}\right)$ 的。

当然，埃拉托斯特尼筛法还存在一些优化，考虑到我们一般不会使用埃拉托斯特尼筛法，而是选用线性筛法，而且加上这些优化后并没有改变时间复杂度，这里就不展开论述了。具体可以参考 OI Wiki 上的描述：https://oi-wiki.org/math/number-theory/sieve/

让我们更进一步，推导出一个时间复杂度为 $\mathcal{O}(n)$ 的筛法，也就是线性筛法。我们需要观察到一个事实：

合数的“过筛”次数 说明 4

考虑到目前的设计是对每个质数 p 过一次 p-筛子，那么对于合数 $x = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$，就会被 p_1、p_2、…、p_k 这 k 个筛子枚举到（尽管在第一次枚举到的时候就知道这个数是合数了）。

例如，对于 $60 = 2^2 \times 3 \times 5$，就会被 2-筛子、3-筛子和 5-筛子经过。

线性筛法的优化核心在于将合数的“过筛”次数降低到恰好 1 次，也就是只要一个数字被确定是合数，那么在之后的过筛流程中就不会被枚举到。

线性筛法的核心在于灵活运用“最小质因数”的概念。线性筛法希望每个合数只会被其最小的质因数筛出，例如 60 只会被 2 筛出。下面，我们记 $\mathrm{lpf}(k)$ 为 k 的最小质因数。

我们需要注意到如下事实：

线性筛法的核心 说明 5

对于两个整数 $n, m > 1$ 有

$$\mathrm{lpf}(nm) = \min(\mathrm{lpf}(n), \mathrm{lpf}(m))$$

那么，对于整数 $k > 1$ 和质数 p，如果 $p > \mathrm{lpf}(k)$，就会有

$$\mathrm{lpf}(kp) = \min(\mathrm{lpf}(k), \mathrm{lpf}(p)) = \min(\mathrm{lpf}(k), p) = \mathrm{lpf}(k) < p$$

那么此时就必然有 $\mathrm{lpf}(kp) \neq p$，使用 p 筛出 kp 就不符合我们的最初想法：每个合数只会被其最小的质因数筛出。

我们注意到上面的表达形式：固定 k，在 $p > \mathrm{lpf}(k)$ 时，筛选就是不符合最初想法的。考虑到我们每次过筛都是固定 p 枚举 k，这就需要我们稍微更换一下枚举思路：

线性筛法 定义 6

枚举 $1 < k \le n$，随后从小到大枚举所有质数 p，沿途标记 pk 为合数。只要在某一时刻 $p > \mathrm{lpf}(k)$，或者 $pk > n$，我们就立刻停止枚举。这样，对于每一个合数 c，就只有在 $k = \frac{c}{\mathrm{lpf}(c)}, k = \mathrm{lpf}(c)$ 的时候被标记。线性筛法的整体时间复杂度是 $\mathcal{O}(n)$。

这个定义仍然有一些不完备的地方，以下是一些补充说明：

线性筛法的补充说明 说明 7

Q1：如何确定在某一时刻有 $p > \mathrm{lpf}(k)$?

A1：我们知道 $\mathrm{lpf}(kp) = \min(\mathrm{lpf}(k), p)$，那么只要在某一时刻，我们枚举到了最小质因数 $p = \mathrm{lpf}(k)$，那么从下一个质数开始就会有 $p > \mathrm{lpf}(k)$。根据最小质因数的定义，由于我们此时是从小到大枚举质数 p，那么这个时刻就是第一次满足 $k \bmod p = 0$ 的时刻。

Q2：线性筛法本身就是要求所有的质数，这里又要求枚举所有质数，会不会出现问题?

A2：实际上并不会。根据筛法性质，我们在枚举 k 时，实际上就已经得到了 $[1, k]$ 范围内的所有质数。根据 A1，我们只要枚举到质数 p 满足 $k \bmod p = 0$，就会立刻停止枚举，那么在过程中枚举到的质数也只会在 $[1, k]$ 范围内。或者说，我们只需要利用当前已经得到的质数。

以下是参考实现：

cpp

```cpp
bool alive[N + 10];
vector<int> linearSieve(int n) {
  vector<int> primes;
  for (int k = 2; k ⩽ n; k ++)
```

```
5       alive[k] = true;
6     for (int k = 2; k ⩽ n; k ++) {
7       if (alive[k]) {
8         // 根据筛法性质，我们知道 k 是质数
9         primes.push_back(k);
10      }
11      // 这句话实际上在按顺序枚举 primes 的元素，也就是从小到大枚举所有质数
12      for (auto p : primes) {
13        // 超出范围则停止
14        if (k * p > n)
15          break;
16        alive[k * p] = false;
17        if (k % p == 0) {
18          // 此时 p = lpf(k)，那么从下一个质数开始就没有枚举的必要了
19          break;
20        }
21      }
22    }
23    return primes;
24  }
```

线性筛法也被称为欧拉筛法。同时需要注意到，在筛出所有质数的同时，我们也得到了每个合数的最小质因数。因为每个合数只会被最小质因数筛出，所以在第 16 行时，可以得到合数 `k * p` 的最小质因数就是 `p`。

我们先从埃拉托斯特尼筛法说起。

埃拉托斯特尼筛法虽然在时间复杂度上不敌线性筛法，但是它也存在自己的优势：埃拉托斯特尼筛法可以做到 $O(\sqrt{n}+B)$ 的辅助空间复杂度（毕竟储存所有质数本身就需要一定的空间）。

第一部分的 $\sqrt{n}$ 其实很好理解。前面提到，一个合数可以写成一个质数乘以另一个数的形式，但在这里，我们可以将这个条件进一步增强：这个质数的大小不超过 $\sqrt{n}$。证明如下：

证明：合数 x 一定存在不超过 $\sqrt{x}$ 的质因数 证明 8

不妨假设 x 是 k 个质数的乘积：$x=p_1p_2...p_k$。因为 x 是合数，所以 $k\geq 2$。

使用反证法，如果所有质数都大于 $\sqrt{x}$，则

$$x=p_1p_2...p_k\geq p_1p_2>\sqrt{x}\sqrt{x}=x$$

矛盾！故 x 必然存在一个不超过 $\sqrt{x}$ 的质因数。

这样，我们就可以先通过线性筛法筛出 $[1,\sqrt{n}]$ 内的质数，然后就可以只使用这些质数筛出 $[1,n]$ 内的所有质数。第一部分对应的空间复杂度就是 $\mathcal{O}(\sqrt{n})$，而第二部分依赖于另一个技巧：分块。

因为我们需要尽可能减少空间的使用量，那么可以考虑将 $[1,n]$ 的区间分为每连续 B 个数字一组，然后再使用埃拉托斯特尼筛法筛出这个区间内的所有质数。每次筛质数时，我们就只需要利用一个长度为 B 的数组，那么第二部分对应的空间复杂度就是 $\mathcal{O}(B)$。

那么，对于一个长度为 B 的区间 $[l,r]$（这里，$1 \le l \le r \le n, r-l+1=B$），我们需要如何进行筛法呢？这就涉及到这一部分的主题：区间筛法。

我们仍然需要通过 $[1,\sqrt{n}]$ 内部的所有质数进行筛法，而对于质数 p，我们需要快速确定区间 $[l,r]$ 内 p 的倍数（筛法还要求这个数不小于 $2p$）。一种简单的方式是：计算这个区间内最小的 p 的倍数，然后不断累加 p，直到超出区间范围。

通过 $kp \ge l$ 可以得到 $k \ge \frac{l}{p}$，故区间内第一个 p 的倍数就是 $\left\lceil \frac{l}{p} \right\rceil p$，据此就可以得到如下代码：

cpp

```cpp
// primes 需要包含 [1, sqrt(r)] 内的所有质数
vector<int> intervalSieve(vector<int> primes, int l, int r) {
  vector<bool> alive(r - l + 1, true);
  for (auto p : primes) {
    // 区间内第一个 p 的倍数
    int cur = (l + p - 1) / p * p;
    // 至少是 2p
    cur = max(cur, 2 * p);
    // p-筛子
    while (cur <= r) {
      alive[cur - l] = false;
      cur += p;
    }
  }
  vector<int> res;
  for (int i = l; i <= r; i ++)
    if (alive[i - l] && i > 1)
      res.push_back(i);
  return res;
}
```

使用这段代码分别对 $[1,B]$，$[B+1,2B]$，……，$[\lfloor\frac{n}{B}\rfloor B+1,n]$ 筛出质数即可。我们一般称这种方法为块筛。接下来是和区间筛法相关的习题：

UVA10140 Prime Distance 问题 9

给定两个正整数 l,r，求 $[l,r]$ 间相邻的两个差最大的质数和相邻 的两个差最小的质数。$1\le l\le r<2^{31},r-l\le 10^6$，有多组测试数据。

根据前面提到的区间筛法，我们首先求出在 $[1,\sqrt{r}]$ 范围内的所有质数，然后在 $[l,r]$ 上进行筛选即可得到 $[l,r]$ 范围内的所有质数。最后只需要计算相邻质数的差即可回答问题。

当然，我们没必要在每组测试数据中都筛一次质数，而只需要在最开始筛出 $\left[1,\left\lfloor\sqrt{2^{31}-1}\right\rfloor\right]$ 范围内的质数即可。

最后，如果你感兴趣的话，区间筛法的时间复杂度可以做到 $O\left(\sqrt{r}+(r-l)\left(\frac{\sqrt{r}}{\log r}+\log\log r\right)\right)$，而块筛的复杂度可以做到 $O\left(\frac{n}{B}\cdot\frac{\sqrt{n}}{\log n}+n\log\log n\right)$。可以得到，块筛的块越小，时间复杂度越高。

2. 积性函数

在这一小节和下一小节中，讨论的核心聚焦在满足下面条件的函数：

数论函数 定义 10

数论函数是一类定义域为正整数、陪域为复数的函数，也就是 $f:\mathbb{Z}^+\to\mathbb{C}$。当然，这个定义看上去太复杂了，让我们换一种描述。

数论函数一般表示为 $f(n)$，它的参数 n 是任意正整数，而 $f(n)$ 的值一般都是整数。目前来说，只需要了解这点就足够了。

接下来是数论函数的一个性质：

积性函数和完全积性函数 定义 11

这里的积性指的是：如果数论函数 f 是积性函数，那么对于正整数 a,b，只要有 $\gcd(a,b)=1$，就必然有 $f(ab)=f(a)f(b)$。

进一步的，完全积性函数的要求严格：对于正整数 a,b，必然有 $f(ab)=f(a)f(b)$，甚至不需要满足 $\gcd(a,b)=1$。

之前已经提到，欧拉函数 $\varphi(n)$ 就符合数论函数的定义，同时也是一个积性函数。在第一小节中提到我们会证明它是一个积性函数，不过为了更好的说明这个结论，我们还需要另一个工具。

狄利克雷卷积 定义 12

对于两个数论函数 f, g，定义其狄利克雷卷积 $(f * g)$ 满足：

$$(f * g)(n) = \sum_{xy=n} f(x)g(y) = \sum_{d|n} f(d)g\left(\frac{n}{d}\right)$$

上面提供了两种写法：

- 第一种写法比较直观地体现了卷积的思想：将乘积等于 n 的两个下标对应的函数值乘起来相加。有一个不那么正式的直观：卷积实际上和整数乘法有一定的关联：按位乘法也是将数量级乘积相同的两个数位对应的值乘起来并相加。
- 第二种写法通过 n 的因数定义，更常用于算术推导中，因为其引入的变量比较少。

狄利克雷卷积满足如下性质：

狄利克雷卷积的性质 定理 13

- 狄利克雷卷积满足交换律，也就是 $f * g = g * f$。

证明：考虑到因数满足一定的对称性，通过第一种写法看其实就易证了。

- 狄利克雷卷积满足结合率，也就是 $(f * g) * h = f * (g * h)$。

证明：

$$[(f * g) * h](n) = \sum_{tc=n} (f * g)(t)h(c) = \sum_{tc=n} h(c) \sum_{ab=t} f(a)g(b) = \sum_{abc=n} f(a)g(b)h(c)$$

左右两边实际上都可以整理成上面的形式，结果相同，结合律得证。

- 对于算式 $f * g = h$，只要其中两个数论函数是积性函数，那么剩余的一个函数也是积性函数。这个结论对完全积性函数不适用，后面将会提到一个例子。

证明：这个证明过于复杂了，在此略去。

接下来，我们将会分别讨论一些常见的积性函数。

2.2.1 1. 卷积的 **“1”** 函数：单位函数 ε（完全积性）

和乘法中的 1 一样（1 乘以任何数都等于被乘数），狄利克雷卷积也存在一个幺元，其对应如下数论函数：

$$\varepsilon(n) = [n = 1] = \begin{cases} 1 & n = 1 \\ 0 & n \neq 1 \end{cases}$$

可以发现，对于任意数论函数 f 都有

$$(f * \varepsilon)(n) = \sum_{xy=n} f(x)\varepsilon(y) = f(n)\varepsilon(1) = f(n)$$

这就可以得到 $\varepsilon * f = f$，根据交换律也可以得到 $f * \varepsilon = f$。

2.2.2 2. 常值函数的“逆元”：莫比乌斯函数 μ（完全积性）

我们首先定义常值函数 1 满足 $1(n)=1$ 恒成立，那么对应的，1 的逆元 μ 需要满足 $\mu*1=\varepsilon$。

首先，$\mu(1)\times 1(1)=\varepsilon(1)$，那么 $\mu(1)=1$。随后，对于任意 $n>1$：

$$\varepsilon(n)=0=\sum_{d|n}\mu(d)1\left(\frac{n}{d}\right)=\sum_{d|n}\mu(d)$$

让我们利用积性函数和狄利克雷卷积的性质：只需要研究 $\mu\left(p^k\right)$ 的值，就可以通过 n 的质因数分解得到最终的函数值。

- $k=0$ 时，$\mu\left(p^k\right)=\mu(1)=1$；
- $k=1$ 时，我们知道 $0=\sum_{d\mid p}\mu(d)=\mu(p)+\mu(1)$，那么 $\mu\left(p^k\right)=\mu(p)=-1$；
- $k=2$ 时，$\mu(p^2)+\mu(p)+\mu(1)=0$，那么 $\mu(p^2)=0$。事实上，在 $k\geq 2$ 时，都有 $\mu\left(p^k\right)=0$。

根据上面的结果，我们将 n 进行质因数分解：$n = p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$，就可以得到：

$$\mu(n) = \mu(p_1^{a_1})\mu(p_2^{a_2})...\mu(p_k^{a_k}) = \begin{cases} 1 & n = 1 \\ (-1)^k & \text{任意 } i, a_i = 1 \\ 0 & \text{存在 } i, a_i > 1 \end{cases}$$

也就是说，如果 n 存在平方因子，那么 $\varphi(n) = 0$；否则，$\mu(n)$ 在 n 的质因数有奇数个时为 -1，否则为 1。例如 $\mu(18) = 0, \mu(30) = -1, \mu(99) = 1$。

2.2.3 3. 恒等函数：id_k（完全积性）

我们定义 $\mathrm{id}_k(n) = n^k$。这个数论函数簇中的每个数论函数显然都是完全积性函数，同时可以发现 $\mathrm{id}_0 = 1$。我们一般称 $\mathrm{id}_1 = \mathrm{id}$。

2.2.4 4. 因数函数：σ_k

定义 $\sigma_k(n)$ 等于 n 的因数的 k 次方和。我们一般称 $d(n) = \sigma_0(n)$ 等于 n 的因数个数，而 $\sigma(n) = \sigma_1(n)$ 等于 n 的因数和。我们写出其定义：

$$\sigma_k(n) = \sum_{d|n} d^k = \sum_{d|n} \mathrm{id}_k(d) 1\left(\frac{n}{d}\right)$$

可以发现 $\sigma_k = \mathrm{id}_k * 1$，那么 σ_k 就是积性函数。不过需要注意的是，虽然 id_k 和 1 都是完全积性函数，但是 σ_k 并不是完全积性函数（例如 $\sigma(4) = 7 \neq 3 \times 3 = \sigma(2) \times \sigma(2)$）。

2.2.5 5. 欧拉函数：φ

我们已经知道欧拉函数 $\varphi(n)$ 代表在 $[1,n]$ 范内和 n 互质的数个数。让我们给出和欧拉函数相关的狄利克雷卷积式，从而证明欧拉函数的确是一个积性函数。

我们定义一个辅助函数 $f(n,x)$ 代表 $[1,n]$ 内和 n 的最大公因数等于 x 的数字个数，那么

- $n=\sum_{i=1}^{n} f(n,i)$，因为 $[1,n]$ 内每个数 x 都被恰好统计一次（在 $i=\gcd(n,x)$ 时）。进一步的，在 n 不是 i 的倍数时 $f(n,i)=0$，所以公式可以进一步写成 $n=\sum_{i|n} f(n,i)$；
- $f(a,b)=f(ca,cb)$，根据最大公因数性质即可得到。

考虑计算 $1*\varphi$：

$$(1*\varphi)(n)=\sum_{d|n}\varphi(d)=\sum_{d|n}\sum_{i=1}^{d}[\gcd(i,d)=1]=\sum_{d|n}f(d,1)=\sum_{d|n}f\left(n,\frac{n}{d}\right)=n$$

我们就得到了 $1*\varphi=\mathrm{id}$，至此就终于证明了 φ 的确是一个积性函数。

我们在最后整理一下目前已经得到的狄利克雷卷积公式，这些公式会在后面经常使用：

- $\varepsilon * f = f$
- $\mu * 1 = \varepsilon$
- $d = 1 * 1$，或者 $1 = d * \mu$
- $\sigma = \mathrm{id} * 1$，或者 $\mathrm{id} = \sigma * \mu$
- $\mathrm{id} = 1 * \varphi$，或者 $\varphi = \mathrm{id} * \mu$

当然，光是定义这些积性函数还不够，我们需要实用高效的方式求出积性函数的某些值。为了求出一个积性函数在 $[1, n]$ 上的值，我们可以使用线性筛法。

具体的，考虑线性筛法的关键点：

cpp

```cpp
bool alive[N + 10];
vector<int> linearSieve(int n) {
  vector<int> primes;
  for (int k = 2; k ⩽ n; k ++)
    alive[k] = true;
  for (int k = 2; k ⩽ n; k ++) {
    if (alive[k]) {
      // 关键点 1
      primes.push_back(k);
    }
    for (auto p : primes) {
      if (k * p > n)
        break;
      alive[k * p] = false;
      if (k % p == 0) {
        // 关键点 2
        break;
```

```
18        } else {
19          // 关键点 3
20        }
21      }
22    }
23    return primes;
24  }
```

我们知道，线性筛时对于每个质数都会经过一次关键点 1，而对于每个合数，则会恰好经过关键点 2 或者关键点 3 中的一个。根据前面的讨论，我们知道：在合数 `k * p` 的质因数分解中 `p` 的指数等于 1 的时候经过关键点 3，否则经过关键点 2。

根据这些性质，我们就可以将线性筛改写成计算积性函数的函数。

让我们首先考虑计算莫比乌斯函数 $\mu(n)$。对于质数而言，我们知道 $\mu(p) = -1$；对于合数而言，考虑当前的合数为 kp，其最小质因数就是 p，而我们需要通过 $\mu(k)$ 得到 $\mu(kp)$。

- 如果 k 是 p 的倍数，那么 kp 就是 p^2 的倍数，我们知道此时 $\mu(kp) = 0$；
- 否则，k 和 p 互质，我们有 $\mu(kp) = \mu(k)\mu(p) = -\mu(k)$。

将这些信息放在关键点上就可以得到：

cpp

```cpp
bool alive[N + 10];
int mu[N + 10];
void getMu(int n) {
  mu[1] = 1;
  vector<int> primes;
  for (int k = 2; k ⩽ n; k ++)
    alive[k] = true;
  for (int k = 2; k ⩽ n; k ++) {
    if (alive[k]) {
      mu[k] = -1;
      primes.push_back(k);
    }
    for (auto p : primes) {
```

```
14        if (k * p > n)
15          break;
16        alive[k * p] = false;
17        if (k % p == 0) {
18          mu[k * p] = 0;
19          break;
20        } else {
21          mu[k * p] = - mu[k];
22        }
23      }
24    }
25  }
```

随后考虑计算欧拉函数 $\varphi(n)$。我们回顾一下 $\varphi(p^k)$ 的值：

$$\varphi(p^k) = p^k - p^{k-1}$$

那么对于质数 p 有 $\varphi(p) = p - 1$，而对于合数 kp：

- 如果 k 是 p 的倍数，不妨假设 k 被 p^a 整数而不被 p^{a+1} 整除，且 $t = \frac{k}{p^a}$。可以知道 t 和 p 互质，并且：

$$\varphi(k) = \varphi(p^a t) = (p^a - p^{a-1})\varphi(t)$$
$$\varphi(kp) = \varphi(p^{a+1} t) = (p^{a+1} - p^a)\varphi(t)$$

  可以得到 $\varphi(kp) = p\varphi(k)$；
- 否则，k 和 p 互质，我们有 $\varphi(kp) = \varphi(k)\varphi(p) = (p-1)\varphi(k)$。

对应的代码如下：

cpp

```cpp
bool alive[N + 10];
int phi[N + 10];
void getPhi(int n) {
  phi[1] = 1;
  vector<int> primes;
```

```
 6   for (int k = 2; k ⩽ n; k ++)
 7     alive[k] = true;
 8   for (int k = 2; k ⩽ n; k ++) {
 9     if (alive[k]) {
10       phi[k] = k - 1;
11       primes.push_back(k);
12     }
13     for (auto p : primes) {
14       if (k * p > n)
15         break;
16       alive[k * p] = false;
17       if (k % p == 0) {
18         phi[k * p] = phi[k] * p;
19         break;
20       } else {
21         phi[k * p] = phi[k] * (p - 1);
22       }
23     }
24   }
25 }
```

当然，上述两个案例都属于积性函数计算的特例，因为我们总是能够轻松的从 $f(k)$ 和 p 得到 $f(kp)$。但是对于其他的积性函数则不总是这样。下面将会讨论一些案例。

我们首先考虑计算因子函数 $d(n)$。我们知道 $d(p^a) = a + 1$，这个形式并不像之前一样，可以简单的通过 $d(p)$ 算出 $d(pk)$。为此，我们考虑对每个数 k 维护两个值：

- k 的最小质因子对应的指数 $\text{alpha}(k)$。除非有特殊需要，我们都不需要维护最小质因子本身，因为 k 会在其最小质因子对应的筛子筛出；
- k 在完全除去最小质因子之后的剩余部分 $\text{rest}(k)$，或者说 $k = \text{lpf}(k)^{\text{alpha}(k)}\ \text{rest}(k)$。

在这个定义下，我们就有 $f(k) = f\left(\text{lpf}(k)^{\text{alpha}(k)}\right) f(\text{rest}(k))$，可以直接通过查询得到想要的值。

接下来考虑如何在代码中维护这两个值。

为了更加容易地兼容所有情况，我们定义函数 `int f(int p, int a)`，代表 $f(p^a)$ 的值。

cpp

```cpp
int f(int p, int a) {
  return a + 1;
}
```

首先在代码中定义数组：

cpp

```cpp
int d[N + 10], alpha[N + 10], rest[N + 10];
```

对于第一个关键点：

cpp

```cpp
    if (alive[k]) {
      alpha[k] = 1;
      rest[k] = 1;
      d[k] = f(k, 1); // = 2
      primes.push_back(k);
    }
```

此时 `k` 是一个质数，所以 `alpha[k] = 1`，`rest[k] = 1`。这是很自然的。

随后是剩余二个关键点。此时我们得到了合数 `k * p`，并且它的最小质因数为 `p`。

cpp

```cpp
      if (k % p == 0) {
        alpha[k * p] = alpha[k] + 1;
        rest[k * p] = rest[k];
        d[k * p] = f(p, alpha[k] + 1) * d[rest[k]]; // = (alpha[k] + 2) * d[rest[k]]
        break;
      } else {
        alpha[k * p] = 1;
        rest[k * p] = k;
        d[k * p] = f(p, 1) * d[k]; // = 2 d[k]
      }
```

在第二个关键点处，`k` 本身就包含了质因子 `p`，所以 `alpha` 只需要在 `k` 的基础上加 1，而 `rest` 不变。

在第三个关键点处，`k` 本身并不包含质因子 `p`，所以此时 `alpha` 就是 1，而对应的 `rest` 就是 `k`。

在这样处理之后，我们就可以处理几乎所有的积性函数了。不妨试着求一个积性函数 u，满足 $u(p^a) = p + a$。

接下来是一道例题：

HDU2136 Largest prime factor 问题 14

求出 $[1, 10^6]$ 区间内每个数字求出其最大质因子，记作 $\mathrm{LPF}(k)$。

这道题目并不需要我们求出 `alpha` 或者 `rest`。实际上，对于任意的整数 $a, b > 1$，一定有

$$\mathrm{LPF}(ab) = \max(\mathrm{LPF}(a), \mathrm{LPF}(b))$$

所以我们只需要得到每个数的任意一种分解，就足以递推得到每个数字的最大质因子。在线性筛的过程中，质数的 `LPF` 显然是本身，而对于合数，在知道最小质因子的情况下，我们就得到了这个合数的一种分解方案，从而就能通过上面的性质得到最终的 `LPF`。

3. 莫比乌斯反演

我们在前面已经得到了如下结论：

$$\mu * 1 = \varepsilon$$

对于两个数论函数 f, g，如果 $f = g * 1$，那么在左右两边同时卷积上 μ 后，就可以得到 $f * \mu = g * 1 * \mu = g$。

按照狄利克雷卷积的定义展开即可得到莫比乌斯反演：

莫比乌斯反演 定义 15

对于两个数论函数 f, g，有

$$f(n) = \sum_{d|n} g(d) \Leftrightarrow g(n) = \sum_{d|n} \mu(d) f\left(\frac{n}{d}\right)$$

我们称 f 是 g 的莫比乌斯变换，而 g 是 f 的莫比乌斯反演。

另外，根据 $\mu * 1 = \varepsilon$，可以得到如下推论，其在数学推导中很常用：

$$[n = 1] = \varepsilon(n) = (\mu * 1)(n) = \sum_{d|n} \mu(d)$$

我们进一步代入 $n = \gcd(i, j)$，就可以得到：

$$[\gcd(i, j) = 1] = \sum_{d|\gcd(i,j)} \mu(d) = \sum_{d|i, d|j} \mu(d)$$

让我们利用刚才得到的结论计算一些题目：

洛谷 P2522 [HAOI2011] Problem b 问题 16

对于给出的 n 个询问，每次求有多少个数对 (x,y)，满足 $a \le x \le b$，$c \le y \le d$，且 $\gcd(x,y) = k$。

在简单容斥之后，这道题等价于求解：

$$\sum_{i=1}^{n}\sum_{j=1}^{m}[\gcd(i,j)=k]$$

首先，考虑到此时 i 和 j 都要是 k 的倍数，故同时除以 k：

$$=\sum_{i=1}^{\lfloor \frac{n}{k} \rfloor}\sum_{j=1}^{\lfloor \frac{m}{k} \rfloor}[\gcd(i,j)=1]$$

随后利用前面的表达式：

$$= \sum_{i=1}^{\lfloor \frac{n}{k} \rfloor} \sum_{j=1}^{\lfloor \frac{m}{k} \rfloor} \sum_{d|i,d|j} \mu(d)$$

我们在这里要求 d 是 i 和 j 的公共因子，那么反过来，要求 i 和 j 都是 d 的倍数，从而交换求和顺序：

$$= \sum_{d=1}^{\min(\lfloor \frac{n}{k} \rfloor, \lfloor \frac{m}{k} \rfloor)} \sum_{i=1}^{\lfloor \frac{n}{kd} \rfloor} \sum_{j=1}^{\lfloor \frac{m}{kd} \rfloor} \mu(d)$$

把 $\mu(d)$ 放在前面，将后面对 1 的求和化简，即可得到：

$$= \sum_{d=1}^{\min(\lfloor \frac{n}{k} \rfloor, \lfloor \frac{m}{k} \rfloor)} \mu(d) \left\lfloor \frac{n}{kd} \right\rfloor \left\lfloor \frac{m}{kd} \right\rfloor$$

这个式子在预处理 $\mu(d)$ 的前缀和后即可在 $\mathcal{O}(\sqrt{n})$ 的复杂度求解。为了达到根号的复杂度，我们还需要一个重要的算法策略：

整除分块 定理 17

我们考虑一个在 $[1,n]$ 上的数组 a_i，满足 $a_i = \left\lfloor \frac{n}{i} \right\rfloor$。此时：

- 在 $i \le \sqrt{n}$ 部分中，这些下标对应的 a_i 值在最坏情况下互不相同；
- 在 $i > \sqrt{n}$ 部分中，$a_i = \left\lfloor \frac{n}{i} \right\rfloor \le \sqrt{n}$，所以这部分的数组值满足 $1 \le a_i \le \sqrt{n}$，最多只会有 $\sqrt{n}$ 个互不相同的值。

因此，a_i 只会出现最多 $2\sqrt{n}$ 种互不相同的值，而且根据单调性，每一种值都对应着数组上的一个区间。我们可以通过如下代码得到这些区间：

cpp

```cpp
for (int l = 1, r; l ⩽ n; l = r + 1) {
  r = n / (n / l);
  printf("在 [%d, %d] 区间上，floor(n / i) 的值固定为 %d\n", l, r, n / l);
}
```

对应到前面得到的式子：

$$= \sum_{d=1}^{\min(\lfloor \frac{n}{k} \rfloor, \lfloor \frac{m}{k} \rfloor)} \mu(d) \left\lfloor \frac{n}{kd} \right\rfloor \left\lfloor \frac{m}{kd} \right\rfloor$$

就可以写出如下代码：

```cpp
for (int l = 1, r; l ≤ min(n / k, m / k); l = r + 1) {
  // 此时 floor(n / i) 和 floor(m / i) 是独立的数组，需要找到最短的同值区间
  r = min(n / (n / l), m / (m / l));
  // 在 l ≤ d ≤ r 时，n / d 和 m / d 的值不变，mu(d) 写成前缀和形式：
  ans += (preMu[r] - preMu[l - 1]) * (n / (k * l)) * (m / (k * l));
}
```

可以看到，只要推导出来的式子出现了形如 $\sum_{i=1}^{n} f(i)g(\frac{n}{i})$ 的形式，那么就可以预处理 $f(i)$ 的前缀和，然后通过整除分块的形式确定 $\frac{n}{i}$ 的同值区间。

2025/7/7 课程 第二部分

数论进阶、容斥原理和莫比乌斯反演

陈景泰

2025-07-06

1. 数论进阶

阶乘取模需要解决如下问题：

阶乘取模 问题 1

给定两个正整数 n, m，求出 $n! \bmod m$ 的值。

另外，考虑将 m 的质因数分解写出：$m = p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$，根据中国剩余定理，我们只需要计算出 $n! \bmod p_i^{a_i}$ 的值，就可以合并出最终 $n! \bmod m$ 的值。在接下来的讨论中，我们仅处理 $m = p^a$ 的情况，其中 p 是质数，a 是正整数。

问题的核心在于求出 $n!$ 的质因数分解中 p 的幂次以及除此之外的部分。我们不妨使用 $\nu_p(n!)$ 表示 $n!$ 的质因数分解中 p 的指数，并且 $\mathrm{ex}(n!,p)=\left(n!/p^{\nu_p(n!)}\right)$，那么 $n!=p^{\nu_p(n!)}\times\mathrm{ex}(n!,p)$。

$\nu_p(n!)$ 的计算比较简单，考虑到只有 p 的倍数才会产生贡献，并且 $\nu_p(pi)=1+\nu_p(i)$，我们有：

$$\nu_p(n!)=\sum_{i=1}^{n}\nu_p(i)=\sum_{i=1}^{\lfloor\frac{n}{p}\rfloor}\nu_p(pi)=\left\lfloor\frac{n}{p}\right\rfloor+\sum_{i=1}^{\lfloor\frac{n}{p}\rfloor}\nu_p(i)=\left\lfloor\frac{n}{p}\right\rfloor+\nu_p\left(\left\lfloor\frac{n}{p}\right\rfloor!\right)$$

把递归展开后就是：

$$\nu_p(n!)=\sum_{i=1}^{+\infty}\left\lfloor\frac{n}{p^i}\right\rfloor$$

为了计算 $\mathrm{ex}(n!,p)$，我们先着手解决 $n!\bmod p$ 的问题，此时需要计算 $\mathrm{ex}(n!,p)\bmod p$。首先给出一个定理：

Wilson 定理 定理 2

对于任意质数 p，都有 $(p-1)!\equiv-1\pmod p$，反之也成立。

然后考虑计算 $\operatorname{ex}(n!,p)$。已知有如下公式成立：

$$\operatorname{ex}(n!,p)\equiv\prod_{i=1}^{n}\operatorname{ex}(i,p)\equiv\prod_{1\le i\le n,p\nmid i}i\times\prod_{1\le i\le\left\lfloor\frac{n}{p}\right\rfloor}\operatorname{ex}(pi,p)\equiv\prod_{1\le i\le n,p\nmid i}(i\bmod p)\times\underbrace{\prod_{1\le i\le\left\lfloor\frac{n}{p}\right\rfloor}\operatorname{ex}(i,p)}_{\operatorname{ex}\left(\left\lfloor\frac{n}{p}\right\rfloor!,p\right)}\pmod p$$

这个公式将 ex 操作分配给 $[1,n]$ 内的每个整数，而这个操作对于非 p 倍数的整数没有任何影响，而对于 p 的倍数，我们根据定义可以知道 $\operatorname{ex}(pi,p)=\operatorname{ex}(i,p)$，那么后半部分就是一个子问题。

而对于前半部分，考虑到我们是每次取出 p 的倍数，这会将 $[1,n]$ 区间分成若干段：

- 除了最后一段之外的段都形如 $[np+1,np+p-1]\left(0\le n<\left\lfloor\frac{n}{p}\right\rfloor\right)$ 的形式。根据 Wilson 定理，这一段数字的乘积对 p 取模后可以写成 $[1,p-1]$ 内所有数字的乘积，结果就是 -1；
- 最后一段可以表示为 $\left[\left\lfloor\frac{n}{p}\right\rfloor p+1,\left\lfloor\frac{n}{p}\right\rfloor p+(n\bmod p)\right]$，这部分的乘积对 p 取模后可以写成 $[1,n\bmod p]$ 内所有数字的乘积，也就是 $(n\bmod p)!$。

我们就得到了 $\operatorname{ex}(n!,p)$ 的递归式：

$$\begin{aligned}\operatorname{ex}(n!,p) &\equiv \prod_{1\le i\le n,p\nmid i}(i \bmod p)\times\prod_{1\le i\le\left\lfloor\frac{n}{p}\right\rfloor}\operatorname{ex}(i,p)\\ &\equiv(-1)^{\left\lfloor\frac{n}{p}\right\rfloor}\times(n \bmod p)!\times\operatorname{ex}\left(\left\lfloor\frac{n}{p}\right\rfloor!,p\right)\pmod p\end{aligned}$$

最后，对于 $n!\bmod p=p^{\nu_p(n!)}\operatorname{ex}(n!,p)\bmod p$，直接通过公式或者递归算出 $\nu_p(n!)$ 和 $\operatorname{ex}(n!,p)$ 的值就可以了。

接下来需要推导更一般的形式，也就是 $n! \bmod p^a$ 的情况。此时 $\nu_p(n!)$ 的求解不受影响，但是 $\mathrm{ex}(n!, p) \bmod p^a$ 的计算就变得复杂了。

为了解决这个问题，我们给出 Wilson 定理的一个推广：

Wilson 定理的推广 定理 3

对于任意质数 p 和正整数 a，都有

$$\prod_{1 \le i < p^a, p \nmid i} i \equiv \begin{cases} 1 & p = 2 \text{ 且 } a \ge 3 \\ -1 & \text{其余} \end{cases} \pmod{p^a}$$

需要注意，式子的左侧是所有 $[1, p^a]$ 内的非 p 倍数的乘积，而不是阶乘。考虑到右侧讨论形式比较简单，我们今后使用 ± 1 表示这个结果，代表实际值需要具体讨论。

接下来依葫芦画瓢进行推导：

$$\operatorname{ex}(n!,p)\equiv\prod_{i=1}^{n}\operatorname{ex}(i,p)\equiv\prod_{1\le i\le n,p\nmid i}i\times\prod_{1\le i\le\left\lfloor\frac{n}{p}\right\rfloor}\operatorname{ex}(pi,p)\equiv\prod_{1\le i\le n,p\nmid i}(i\bmod p^a)\times\underbrace{\prod_{1\le i\le\left\lfloor\frac{n}{p}\right\rfloor}\operatorname{ex}(i,p)}_{\operatorname{ex}\left(\left\lfloor\frac{n}{p}\right\rfloor!,p\right)}\pmod{p^a}$$

后面依然是子问题，但是前面的拆段就需要以 p^a 的长度作为整体考虑。我们最后可以拆出 $\left\lfloor\frac{n}{p^a}\right\rfloor$ 个完整的段和一个不完整的段，具体细节就不展开了。最后可以得到：

$$\operatorname{ex}(n!,p)\equiv(\pm1)^{\left\lfloor\frac{n}{p^a}\right\rfloor}\times\underbrace{\prod_{1\le i\le n\bmod p^a,p\nmid i}i}_{F_{n\bmod p^a}}\times\operatorname{ex}\left(\left\lfloor\frac{n}{p}\right\rfloor!,p\right)\pmod{p^a}$$

我们令 $F_n=\prod_{1\le i\le n,p\nmid i}i$，不难发现这个数组可以通过线性递推得到，我们就可以在 $\mathcal{O}(p^a)$ 的时间复杂度内预处理 F_n，然后用对数的时间复杂度计算递归式。

我们定义组合数 $\binom{n}{m} = \frac{n!}{m!(n-m)!}$，并且在参数不满足 $0 \le m \le n$ 时令 $\binom{n}{m} = 0$。

卢卡斯定理希望解决大组合数的取模问题。这里直接给出其定义：

Lucas 定理 问题 4

对于质数 p，总是有

$$\binom{n}{k} \equiv \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor} \binom{n \bmod p}{k \bmod p} \pmod{p}$$

至于证明，则可以灵活运用前面的阶乘取模结果，在此就不作展开了。

进一步，我们考虑将 n 和 k 写成 p 进制：

$$n = a_0 p^0 + a_1 p^1 + a_2 p^2 + \ldots \ (0 \le a_i < p)$$
$$k = b_0 p^0 + b_1 p^1 + b_2 p^2 + \ldots \ (0 \le b_i < p)$$

那么 Lucas 定理就变成：

$$\binom{n}{k} \equiv \binom{a_0}{b_0}\binom{a_1}{b_1}\binom{a_2}{b_2}\ldots \pmod{p}$$

这里的乘法并不是永无止境的。可以发现，在 i 到达对数量级之后，$a_i = b_i = 0$，那么 $\binom{a_i}{b_i} = 1$。这样就可以将连乘压缩到一个比较小的范围内。

exLucas 算法是基于 Lucas 定理的组合数取模算法，可以对任意模数 m 计算 $\binom{n}{k} \bmod m$ 的值。实际上，exLucas 的形式可以直接通过前面的所有讨论推导，比 Lucas 定理的证明更加容易一些。

exLucas 算法 问题 5

给定正整数 n, k 和模数 m，求出 $\binom{n}{k} \bmod m$ 的值。

另外，考虑将 m 的质因数分解写出：$m = p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$，根据中国剩余定理，我们只需要计算出 $\binom{n}{k} \bmod p_i^{a_i}$ 的值，就可以合并出最终 $\binom{n}{k} \bmod m$ 的值。在接下来的讨论中，我们仅处理

而对于 $\binom{n}{k} \bmod p^a$ 的问题，我们知道 $\binom{n}{k} = \frac{n!}{k!(n-k)!}$，那么

$$\binom{n}{k} \equiv p^{\nu_p(n!)-\nu_p(k!)-\nu_p((n-k)!)} \times \frac{\mathrm{ex}(n!, p)}{\mathrm{ex}(k!, p)\ \mathrm{ex}((n-k)!, p)} \pmod{p^a}$$

这里使用的所有参数都可以通过前面的阶乘取模结果计算出来，我们就完成了 exLucas 算法的讨论。

以下是一道相关练习题：

洛谷 P2480 [SDOI2010] 古代猪文 问题 6

给定 n, g，计算：

$$g^{\sum_{d|n}\binom{n}{d}} \bmod 999911659$$

首先可以发现，我们只需要计算 $\sum_{d|n}\binom{n}{d} \bmod \varphi(999911659)$ 的结果，就可以将其作为幂次计算快速幂得到答案。考虑到 999911659 恰好是一个质数，这等价于计算：

$$\sum_{d|n}\binom{n}{d} \bmod 999911658$$

我们枚举 n 的所有因数 d 并计算 $\binom{n}{d}$。考虑到 $999911658 = 2 \times 3 \times 4697 \times 35617$，并没有质数幂取模，故可以直接使用 Lucas 定理解决。最后使用中国剩余定理合并所有答案即可。

群是代数结构中极为重要的一个概念，其内涵比较复杂，我们在此仅利用其性质，而不深入讨论。

我们讨论 $[1, n-1]$ 内和 n 互质的所有数字对 n 取模对应的群。不难发现这个群的大小，或者说阶，等于 $\varphi(n)$。根据群论知识，我们作出如下定义：

阶和原根 定义 7

对于这个群中的一个元素 a，需要满足 $\gcd(a, n) = 1$。我们从 1 开始，不断乘以 a 并对 n 取模，可以证明最后一定会在有限步重新得到 1。我们称这个步数为 a 的阶，记作 $\mathrm{ord}_n(a)$。换句话说，$\mathrm{ord}_n(a)$ 是满足 $a^k \bmod n = 1$ 的最小正整数 k。

如果一个元素 a 在用作乘法时遍历了群内的所有元素，也就是 $\mathrm{ord}_n(a) = \varphi(n)$，那么我们成这个群是一个循环群，称 a 为这个循环群的生成元，也称其为 n 的原根。

我们不妨以 $m = 18$ 举例，其所有原根为 $5, 11$，取原根 $g = 5$ 可以得到如下循环：

$$1 \xrightarrow{\times g} 5 \xrightarrow{\times g} 7 \xrightarrow{\times g} 17 \xrightarrow{\times g} 13 \xrightarrow{\times g} 11 \xrightarrow{\times g} 1$$

我们知道 $\varphi(18) = 18 \times \frac{2-1}{2} \times \frac{3-1}{3} = 6$，那么这个原根自然就便利了群内所有的元素。另外，群中每个元素都可以表示为 g^i 的形式，乘上这个数字就等于在循环上沿着箭头前进 i 步。

接下来考虑如何判断一个数字 g 是不是 n 的原根。我们可以以如下性质为基础：

- 如果对于正整数 k 有 $g^k \bmod n = 1$，那么对于 g 生成的循环，这相当于从 1 开始前进 k 步最终回到 1。这说明 k 是环的大小，也就是 $\mathrm{ord}_n(g)$ 的倍数。因此，我们可以给定一个正整数 k 并验证 $\mathrm{ord}_n(g)$ 是不是 k 的因子。

数论中的拉格朗日定理指出，群内元素 a 的阶必然是群的阶的因子，也就是 $\mathrm{ord}_n(a) \mid \varphi(n)$。为了验证 $\mathrm{ord}_n(a) = \varphi(n)$，我们考虑处理这个条件的另一写法：$\mathrm{ord}_n(a)$ 不是 $\varphi(n)$ 的真因子。

真因子的判断并不复杂。我们在求出 $\varphi(n)$ 后进行质因数分解，得到 $\varphi(n)=p_1^{a_1}p_2^{a_2}...p_k^{a_k}$。那么对于 n 的任意一个真因子 c，考虑随便取 $\frac{n}{c}$ 中的一个质因数 p_i，那么 c 就一定是 $\frac{n}{p_i}$ 的因子。换句话说：

原根判定 说明 8

$\mathrm{ord}_n(a)=\varphi(n)$ 的充分必要条件为：对所有 $1\le i\le k$，$\mathrm{ord}_n(a)$ 都不是 $\frac{\varphi(n)}{p_i}$ 的因子，也即

$$a^{\frac{\varphi(n)}{p_i}}\not\equiv 1\pmod n$$

我们不加证明的给出如下事实：

原根的性质 说明 9

一个整数 n 存在原根，当且仅当 n 可以写作 $2,4,p^a,2p^a$ 中的一种。其中 p 是奇质数，而 a 是正整数。另外，一个质数 p 的最小原根被证明是 $\mathcal{O}(p^{0.25+\varepsilon})$ 的，因此我们通过从小到大的顺序枚举得到质数的最小原根的时间复杂度是可以接受的。

接下来简单讨论一下原根的数量。

原根个数 说明 10

假设 n 存在至少一个原根，那么其原根的个数为 $\varphi(\varphi(n))$。

证明：任意取一个原根 g，并考虑通过 g 生成的长度为 $\varphi(n)$ 的环。对于所有元素，都可以表示为 g^i 的形式。容易发现在长度为 $\varphi(n)$ 的环，每次走 i 步的情况下将会在 $\frac{\varphi(n)}{\gcd(\varphi(n),i)}$ 步回到起点，而 g^i 是原根等价于必须要走满 $\varphi(n)$ 步后才能回到起点。

因此，g^i 是 n 的原根，当且仅当 $\gcd(\varphi(n),i)=1$。这就证明了原根的个数是 $\varphi(\varphi(n))$ 个。另外，我们也可以通过这一性质得到一个数的所有原根。

照着这个思路就可以完成如下例题：

洛谷 P6091 【模板】原根 问题 11

给定整数 n，求它的所有原根。

为了减小你的输出量，给出输出参数 d，只需要从小到大，每隔 d 个输出一个原根。

首先通过前面提到的原根判别法找到 n 的最小原根 g，然后计算 $\varphi(n)$ 并得到所有和 $\varphi(n)$ 互质的数字，最后计算 g 的幂次并排序即可。

我们最后考虑通过原根定义离散对数。

离散对数 定义 12

对于模数 n，如果其存在原根 g，则对于 $[1,n]$ 区间与 n 互质的数字 a，存在唯一的 $0 \le i < \varphi(n)$，使得 $g^k \bmod n = a$。我们称 k 为以 g 为底时模 n 的离散对数，写作 $\gamma(a) = k$。

不难发现，$\gamma(ab) = (\gamma(a) + \gamma(b)) \bmod \varphi(n)$，也就是乘积的函数等于函数的和。这和 log 的性质十分类似。

离散对数相关的题目一般涉及到 BSGS 等内容，因此在这里仅作为扩展内容。

2. 容斥原理

容斥原理是通过集合论得到的数学推导公式，因此我们先简单介绍集合。

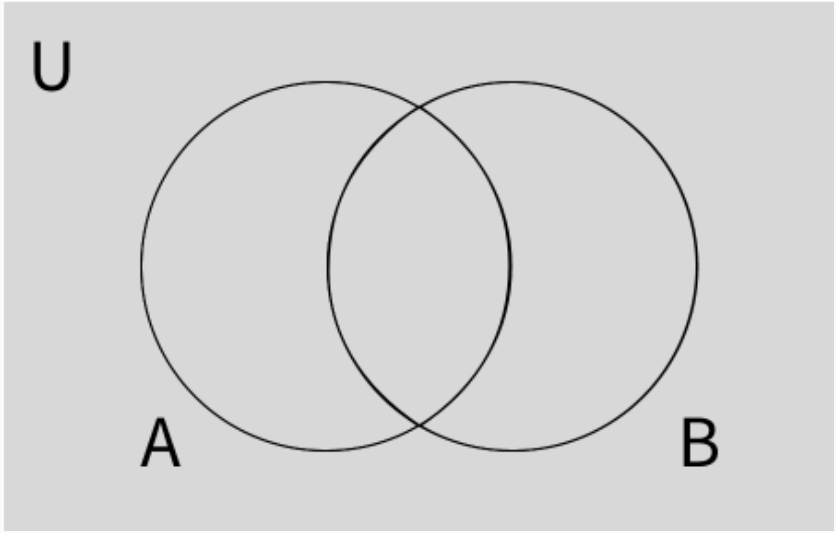
集合本质上就是一堆互不相同的数字形成的结构，在 C++ 中和 `set<T>` 类似。在数学上，一般使用 A, B, C 等大写字母表示集合，而使用 U 表示当前考察的元素类型的全集（例如，在考察自然数时，$U = \mathbb{N}$，而在考察实数时，$U = \mathbb{R}$）。如果 o 是 A 的元素，则使用 $o \in A$ 表示，否则使用 $o \notin A$ 表示。

一般的集合表示法包括：

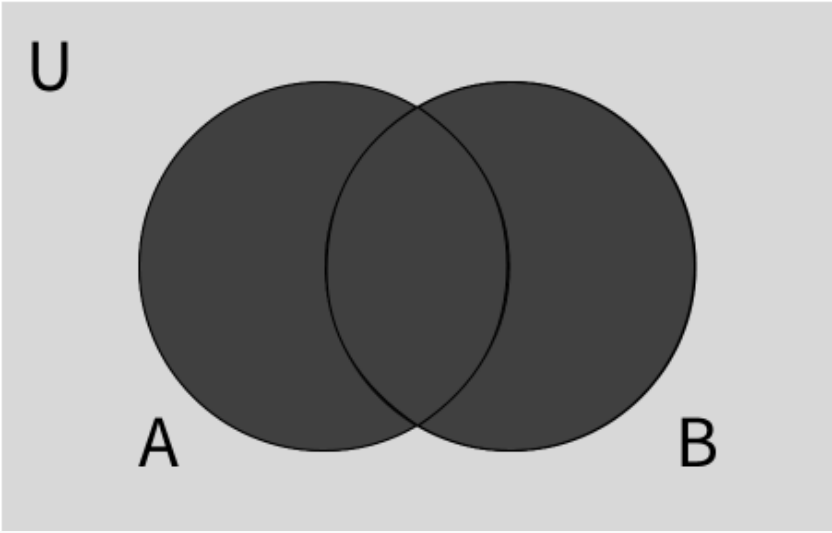
- 列举法：直接将集合的所有元素写出，例如 $A = \{2, 4, 5, 7, 11\}$；
- 归纳法：给出归纳基和归纳步，从而定义集合，例如：
  - 归纳基：$4 \in A, 7 \in A$；
  - 归纳步：如果 $a, b \in A$，则 $a + b \in A$。
- 描述法：通过一段描述定义集合，例如 $A = \{x \mid 2 \mid x \text{ 且 } 3 \mid x\}$。

除了元素和集合的包含关系之外，集合和集合之间也存在包含关系。对于两个集合 A, B，如果 A 中的所有元素 a 都在集合 B 中出现，则称集合 A 是集合 B 的子集，写作 $A \subseteq B$。子集符号的横杠代表集合 A 和集合 B 可能相等，而如果集合 A 和集合 B 不相等，则称集合 A 是集合 B 的真子集，写作 $A \subset B$。另外，我们也会说 B 是 A 的超集。

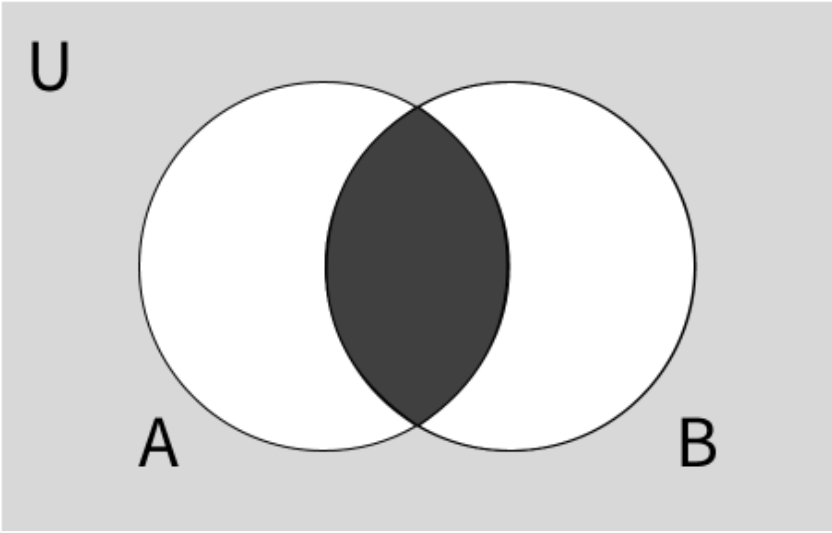
另外，集合之间也存在一些计算。我们一般会使用文氏图直观体现集合之间的关系，以下是接下来会使用的文氏图表示：



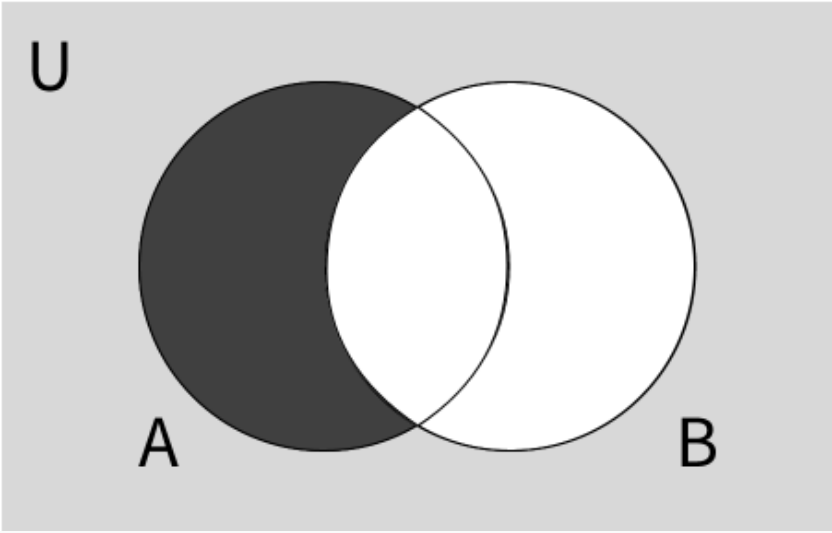
集合的并：两个集合 A 和 B 的并定义为 $A \cup B = \{x \mid x \in A \text{ 或 } x \in B\}$，类似于将两个集合的元素放在一起并去重。



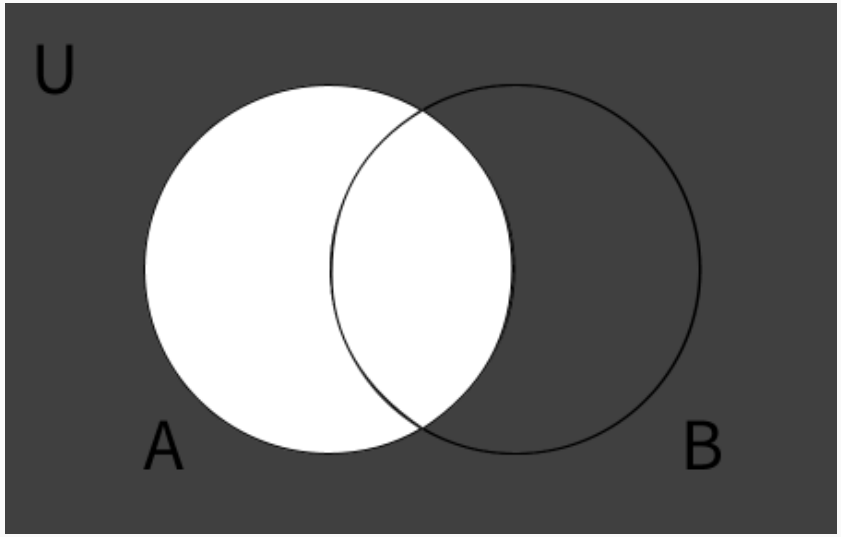
集合的交：两个集合 A 和 B 的交定义为 $A \cap B = \{x \mid x \in A \text{ 且 } x \in B\}$，类似于将两个集合都有的元素拿出来形成的集合。



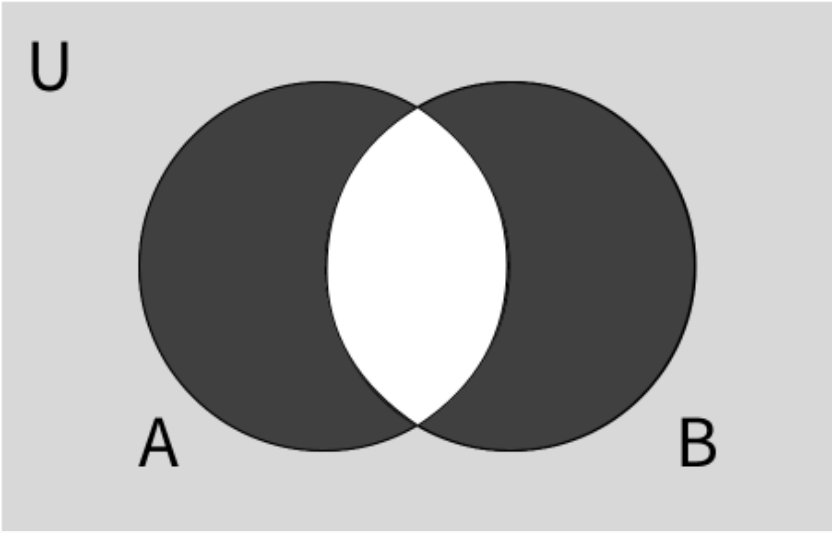
集合的差集：两个集合 A 和 B 的差集定义为 $A - B = \{x \mid x \in A \text{ 且 } x \notin B\}$，也就将 B 中所有元素都从 A 中移除。



集合的补集：一个集合 A 在全集 U 中的补集定义为 $\complement_U A = U - A$。差集也可以写作 $A - B = A \cap \complement_U B$。

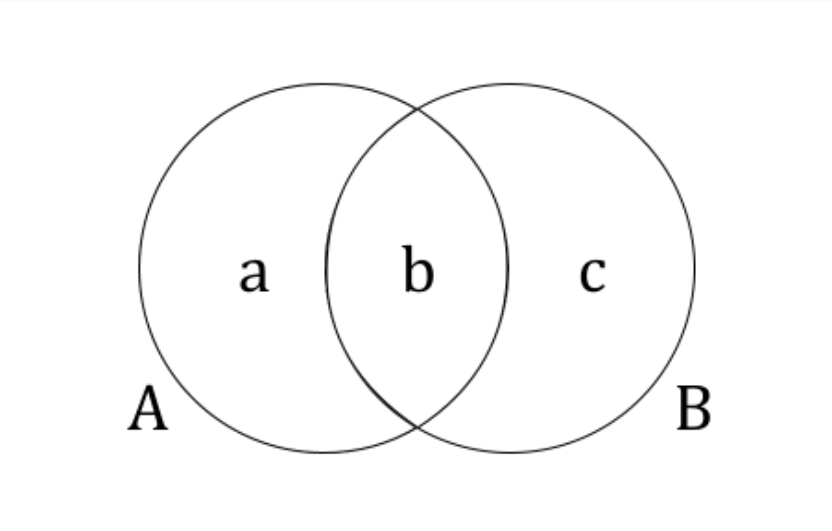


集合的对称差：两个集合 A 和 B 的对称差定义为 $A \oplus B = \{x \mid x \in A \text{ 异或 } x \in B\}$。代表将在两个集合中共出现恰好一次的元素放在一起。



在有了文氏图和一些经典的集合操作定义后，我们就可以开始定义容斥原理了。让我们先考虑两个集合的情况。

定义 $|A|$ 代表集合 A 的大小。为了得到 $|A \cup B|$，我们可以通过文氏图展示两个集合划分的三个区域（小写字母代表区域的元素个数）：



我们知道 $|A \cup B| = a + b + c, |A| = a + b, |B| = b + c, |A \cap B| = b$，那么可以得到：

$$|A \cup B| = |A| + |B| - |A \cap B|$$

接下来考虑三个集合的情况，计算 $|A \cup B \cup C|$。

为了计算 $|A \cup B \cup C| = a + b + c + u + v + w + o$，我们首先计算

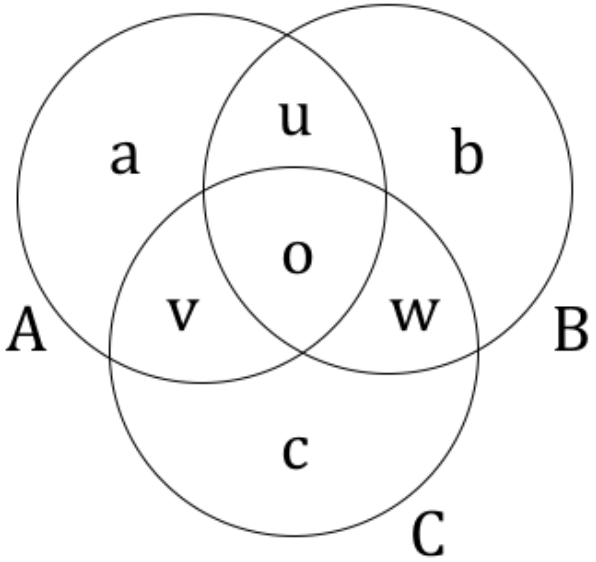
$$|A| + |B| + |C| = (a + b + c) + 2(u + v + w) + 3o$$

发现 u, v, w, o 都算重了，通过 $|A \cap B| + |B \cap C| + |C \cap A|$ 调整：

$$\begin{aligned} &|A| + |B| + |C| - (|A \cap B| + |B \cap C| + |C \cap A|) \\ &= (a + b + c) + (u + v + w) \end{aligned}$$

最后发现少算了 o，再加上 $|A \cap B \cap C|$ 补全，最后得到

$$\begin{aligned} |A \cup B \cup C| = &|A| + |B| + |C| \\ &-(|A \cap B| + |B \cap C| + |C \cap A|) \\ &+ |A \cap B \cap C| \end{aligned}$$



事实上，这种“多了就减、少了就加”的思路就是“容斥原理”名称的由来。通过对前面讨论的适当扩展，我们可以得到容斥原理的完整定义：

容斥原理 定理 13

对于 n 个集合 $A_1, A_2, ..., A_n$，有

$$|A_1 \cup A_2 \cup ... \cup A_n| = \sum_{1 \le a_1 \le n} |A_{a_1}| - \sum_{1 \le a_1 < a_2 \le n} |A_{a_1} \cap A_{a_2}| + \sum_{1 \le a_1 < a_2 < a_3 \le n} |A_{a_1} \cap A_{a_2} \cap A_{a_3}| - ... + (-1)^{n-1} |A_1 \cap A_2 \cap ... \cap A_n|$$

在理解容斥原理后，让我们看一道例题：

洛谷 P1450 [HAOI2008] 硬币购物 问题 14

有 4 种面值的硬币，每种硬币的面值为 C_i。每次给出每种硬币的个数 D_i 和价格 S，求出使用硬币凑出恰好 S 元的方案数。

这道题目本质上等价于如下问题：对每次询问，求出满足如下要求的自然数四元组 (a_1, a_2, a_3, a_4) 的数量：

- $a_i \le D_i$（称为条件 1-i）；
- $a_1C_1 + a_2C_2 + a_3C_3 + a_4C_4 = S$（称为条件 2）。

这一题虽然是经典的背包问题，但是物品的限制是随着询问变动的，每次查询的时候从重新跑一次背包显然无法接受，接下来考虑使用容斥原理将背包性质进行转换。

定义：

- 全集 $U = \{(a_1, a_2, a_3, a_4) \mid (a_1, a_2, a_3, a_4)$ 满足条件 2$\}$（对条件 1 没有要求）；
- 四个集合 $A_i = \{(a_1, a_2, a_3, a_4) \mid (a_1, a_2, a_3, a_4)$ 满足条件 2 且 不满足条件 1-i$\}$（对其余条件 1 没有要求）。

那么 $|A_1 \cup A_2 \cup A_3 \cup A_4|$ 就是所有满足条件 2，但是不满足任意一个条件 1 的四元组数量，只需要将全集大小去掉这些不满足条件 1 的四元组，得到的就是答案了。

注意到 A_i 的并大小难以求解，但是交大小比较容易求解。例如，$|A_1 \cap A_2|$ 就是所有满足条件 2，但不满足条件 1-1 和 1-2 的四元组 (a_1, a_2, a_3, a_4)。此时 $a_1 > D_1, a_2 > D_2$，那么设 $a_1' = a_1 - D_1 - 1, a_2' = a_2 - D_2 - 1$，那么考虑新的四元组 (a_1', a_2', a_3, a_4)，其中只需要保证每个数都是自然数，而且

$$a_1'C_1 + a_2'C_2 + a_3C_3 + a_4C_4 = S - (D_1 + 1)C_1 - (D_2 + 1)C_2$$

可以发现，全集 $|S|$ 和上述问题都符合完全背包的要求，那么可以提前预处理出 f_n 表示使用四种面值的硬币凑出 n 元且硬币不限的方案数，就有

$$|U| = f_S, |A_1 \cap A_2| = f_{S-(D_1+1)C_1-(D_2+1)C_2}$$

在算出 A_i 的所有交集的大小后，套用四个集合的容斥原理即可算出 A_i 的并集的大小，最后算出问题的答案。

容斥原理除了直接使用外，也可以和数论等内容进行结合。以下是一个经典例子：

洛谷 P1447 [NOI2010] 能量采集 问题 15

给定 n, m，计算

$$\sum_{i=1}^{n}\sum_{j=1}^{m}[2\gcd(i,j)-1]$$

不妨令 f_k 表示对所有满足 $1 \le i \le n, 1 \le j \le m$ 的数对 (i,j) 中，满足 $\gcd(i,j)=k$ 的个数，那么要求的答案就是 $\sum_{i=1}^{n} f_i \times (2i-1)$。

考虑根据 f_k 的定义使用容斥原理。我们已经知道，$k \mid \gcd(i,j)$ 等价于 $k \mid i$ 且 $k \mid j$，这两个条件分别代表对 i,j 的限制且互相独立，而满足第一个限制的 i 有 $\lfloor \frac{n}{k} \rfloor$ 个，满足第二个限制的 j 有 $\lfloor \frac{m}{k} \rfloor$ 个，那么满足 $k \mid \gcd(i,j)$ 的数对就有 $\lfloor \frac{n}{k} \rfloor \lfloor \frac{m}{k} \rfloor$ 个。

但是对于 $k \mid \gcd(i,j)$，除了 $\gcd(i,j)=k$ 的情况外，还存在 $\gcd(i,j)=2k, 3k, \ldots$ 等被额外计算的情况，此时根据容斥原理，只需要去掉这些答案，就可以得到 f_k 的真实值。

我们就可以得到 f_k 的递推式：

$$f_k = \left\lfloor \frac{n}{k} \right\rfloor \left\lfloor \frac{m}{k} \right\rfloor - \sum_{i=2}^{\left\lfloor \frac{n}{k} \right\rfloor} f_{ik}$$

代码实现如下：

cpp

```cpp
for (long long k = n; k >= 1; k--) {
  f[k] = (n / k) * (m / k);
  for (long long i = 2 * k; i <= n; i += k)
    f[k] -= f[i];
}
```

事实上，容斥原理并不局限于计算集合的大小。它可以扩展为如下形式：

容斥原理的一般化 定理 16

对于两个在集合上的函数 $f(S), g(S)$，有

$$f(S) = \sum_{T \subseteq S} g(T) \Leftrightarrow g(S) = \sum_{T \subseteq S} (-1)^{|S| - |T|} f(T)$$

这个式子看上去和莫比乌斯反演很像。实际上，我们的确可以通过莫比乌斯反演导出上面的式子，从而绕开对组合数学的需求。

我们假设全集 U 包含 n 个元素：$u_1, u_2, ..., u_n$。我们将 u_i 映射到从小到第 i 个质数 P_i，那么 $u_1 \to 2, u_2 \to 3, u_3 \to 5...$。

我们定义一个辅助的数论函数 $F(n)$，并且令 $f(S) = f\left(\left\{u_{a_1}, u_{a_2}, ..., u_{a_k}\right\}\right) = F\left(P_{a_1}P_{a_2}...P_{a_k}\right)$，同理定义 $G(n)$。在这样的刻画下，f 和 g 的子集关系就变成了 F 和 G 的整除关系，也就是说 $F = G * 1$（不考虑其他没填写的项）。

那么根据莫比乌斯反演，我们有 $G = F * \mu$，也就是

$$G(n) = \sum_{d|n} \mu\left(\frac{n}{d}\right) F(d)$$

在代回到 f 和 g 的时候，$d \mid n$ 就又变回了子集关系 $T \subseteq S$，而此时 $\frac{n}{d}$ 代表的就是 $S - T$，并且根据定义，$\frac{n}{d}$ 不包含平方因子，那么 $\mu\left(\frac{n}{d}\right)$ 就变成了 $(-1)^{|S| - |T|}$。这样就得到了容斥原理一般化的式子。

让我们通过实例阐释这一公式的用途。

洛谷 P10596 BZOJ2839 集合计数 问题 17

一个有 n 个元素的集合有 2^n 个不同子集（包含空集），现在要在这 2^n 个集合中取出若干集合（至少一个），使得它们的交集的元素个数为 K，求取法的方案数，答案模 10^9+7。

我们定义 $f(S)$ 为交集为 S 的超集的方案数，而 $g(S)$ 为交集恰好为 S 的方案数。这个时候我们知道：

$$f(S)=\sum_{S\subseteq T\subseteq U} g(T)$$

此时的求和方向不一样，那么对应的推论公式也会发生翻转（只需要将 $f(S)$ 换成 $f'(\complement_U S)$ 即可证明）：

$$g(S)=\sum_{S\subseteq T\subseteq U} (-1)^{|T|-|S|} f(T)$$

我们可以很容易得算出 $f(S)$。$f(S)$ 实际上只要求选出的集合都是 S 的超集，而这样的集合有 $2^{n-|S|}$ 个，而选出的集合不能为空，就对应有 $2^{2^{n-|S|}}-1$ 种选择。

那么对应有 $g(S)=\sum_{S\subseteq T\subseteq U}(-1)^{|T|-|S|}\left(2^{2^{n-|T|}}-1\right)$。通过观察可以发现，只要 $|S|=|S'|$，那么就必然有 $f(S)=f(S')$ 与 $g(S)=g(S')$，下面不妨直接使用大小 k 作为 f 和 g 的参数，即可得到：

$$g(k)=\sum_{k\le i\le n}(-1)^{i-k}\times\binom{n-k}{i-k}\times\left(2^{2^i}-1\right)$$

最后考虑到大小为 K 的集合共有 $\binom{n}{K}$ 个，答案就是

$$\binom{n}{K}g(K)=\binom{n}{K}\sum_{K\le i\le n}(-1)^{i-K}\times\binom{n-K}{i-K}\times\left(2^{2^i}-1\right)$$

3. 习题回顾

ABC412E LCM Sequence

问题 18

定义 A_n 为 $[1,n]$ 内所有数字的最小公倍数，给定区间 $[l,r]$ 满足 $1 \le l \le r \le 10^{12}$ 且 $r-l \le 10^7$，计算 $[l,r]$ 范围内 A_n 的不同值个数。

只需要考虑何时出现 $A_n \neq A_{n-1}$ 的情况。考虑 A_n 以及一个质数 p，根据最小公倍数的性质，我们知道 A_n 中 p 对应的指数就是 $[1,n]$ 中的整数对应的指数最大值，也就是 $\lfloor \log_p n \rfloor$。那么 $A_n \neq A_{n-1}$ 当且仅当在 A_n 处，某一个指数发生了变化，简单推导可以得到其等价于 n 是某个质数的幂次（或者就是质数）。

进行区间筛法，除了判断区间中哪些数是质数之外，还需要记录下其余数字被筛出时对应的质因数，进而判断这个数字是不是该质因数的幂次。

洛谷 P1829 [国家集训队] Crash 的数字表格 / JZPTAB 问题 19

计算

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\operatorname{lcm}(i,j)$$

这一道题目是十分经典的莫比乌斯反演练习，希望大家可以通过课堂上学习到的工具独立推出以下的公式：

$$
\begin{aligned}
&\sum_{i=1}^{n}\sum_{j=1}^{m}\operatorname{lcm}(i,j)\\
=&\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{ij}{\gcd(i,j)}\\
=&\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{d=1}^{n}[\gcd(i,j)=d]\cdot\frac{ij}{d}\\
=&\sum_{d=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{m}[\gcd(i,j)=d]\cdot\frac{ij}{d}\\
=&\sum_{d=1}^{n}\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{d}\rfloor}[\gcd(i,j)=1]\cdot ijd\\
=&\sum_{d=1}^{n}d\cdot\underbrace{\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{d}\rfloor}[\gcd(i,j)=1]\cdot ij}_{g(\lfloor\frac{n}{d}\rfloor,\lfloor\frac{m}{d}\rfloor)}
\end{aligned}
$$

$$
\begin{aligned}
g(n,m)&=\sum_{i=1}^{n}\sum_{j=1}^{m}[\gcd(i,j)=1]\cdot ij\\
&=\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{d\mid i,d\mid j}\mu(d)\cdot ij\\
&=\sum_{d=1}^{n}\mu(d)d^2\underbrace{\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{d}\rfloor}ij}_{h(\lfloor\frac{n}{d}\rfloor,\lfloor\frac{m}{d}\rfloor)}\\
h(n,m)&=\sum_{i=1}^{n}\sum_{j=1}^{m}ij\\
&=\frac{1}{4}nm(n-1)(m-1)
\end{aligned}
$$

求算答案时利用整除分块并固定 g 的参数，而在计算 g 的时候继续整除分块并固定 h 的参数。时间复杂度 $\mathcal{O}(n+m)$，需要预处理 $\mu(d)d^2$ 的前缀和。