

NOIP 数论基础

吴一岐

2018 年 7 月 5 日

目录

1 素数筛法	3
1.1 朴素版筛法	3
1.2 优化版筛法	3
1.3 线性筛法	4
2 欧几里得	5
2.1 辗转相减	5
2.2 辗转相除	5
3 扩展欧几里得	7
3.1 解的存在性问题	7
3.2 特解与通解	7
3.3 关于特解的绝对值大小	9
4 欧拉函数	10
4.1 定义	10
4.2 欧拉函数值	10
4.3 欧拉函数常用性质总结	10
5 欧拉定理	13
6 费马小定理	14

7 乘法逆元	14
7.1 线性递推求逆元	14
8 中国剩余定理	15

1 素数筛法

1.1 朴素版筛法

将每一个数的所有的倍数都筛去，由于素数只能被 1 和它本身整除，因此素数不会被筛到，计算量为 $n/2+n/3+n/4+\dots$ 这个是调和级数，复杂度 $O(n\log n)$

朴素版筛法

```
const int N = 100010;
bool flag[N];
int p[N], tot;
void init(int n) {
    for (int i = 2; i <= n; i++) {
        for (int j = i + i; j <= n; j += i) {
            flag[j] = true;
        }
    }
    for (int i = 2; i <= n; i++) {
        if (!flag[i]) {
            p[tot++] = i;
        }
    }
}
```

1.2 优化版筛法

我们注意到朴素版筛法里面 36 会被 2,3,4,6,9,12,18 这 6 个数筛到，相当于被 $2*18$, $3*12$, $4*9$, $6*6$, $9*4$, $12*3$, $18*2$ 筛到，实际上 $a*b$ 跟 $b*a$ 是等价的，于是，我们可以通过枚举小的那个因子，来优化一半的计算量，相当于 $i*(i+1)$, $i*(i+2)$ 这样子去筛，复杂度不变，常数上优化了

```
void init(int n) {
    int up = (int)sqrt(1.0*n) + 1;
    for (int i = 2; i <= up; i++) {
        for (int j = i * i; j <= n; j += i) {
            flag[j] = true;
        }
    }
    for (int i = 2; i <= n; i++) {
        if (!flag[i]) {
            p[tot++] = i;
        }
    }
}
```

1.3 线性筛法

那么存在 $O(n)$ 的筛法，使得每个数只被筛到一次么？答案是肯定的。
下面这个代码短小精悍，可以做到每一个数只被其最小的素因子筛选到

```
void init(int n) {
    for (int i = 2; i <= n; i++) {
        if (!flag[i]) {
            p[tot++] = i;
        }
        for (int j = 0; j < tot; j++) {
            if (i * p[j] > n) {
                break;
            }
            flag[i * p[j]] = true; //p[j]是i * p[j]的最小素因子
            if (i % p[j] == 0) { // 再往后p[j]就不是i * p[j]的最小素因子了
                break;
            }
        }
    }
}
```

2 欧几里得

2.1 辗转相减

利用辗转相减法求最大公约数, 即 $\gcd(a, b)$ 。假设 $a > b$, 则 $\gcd(a, b) = \gcd(a - b, b)$, 不断的利用大的数减去小的数, 就能得到最大公约数。

证明:

$$a = k_1 * g$$

$$b = k_2 * g$$

假设 g 为最大公约数, 那么 $\gcd(k_1, k_2) = 1$, 设 $a > b$, 令 $a = a - b$,

$$a = (k_1 - k_2) * g$$

$$b = k_2 * g$$

我们发现 a, b 的变化为两个互质的系数在不断相减 (始终还是互质), g 始终是它们的最大公约数, 所以当有一个数变成 0 的时候, 另一个数就是最大公约数

辗转相减

```
while (a != b) {  
    if(a > b) {  
        a -= b;  
    } else {  
        b -= a;  
    }  
}
```

2.2 辗转相除

我们发现当 a 一直大于 b 的时候, 就会一直减去 b , 其实就是变成 $a \% b$, 于是我们可以利用辗转相除来优化,

辗转相除

```
while (a && b) {  
    if (a > b) {  
        a = a % b;  
    } else {  
        b = b % a;  
    }  
}  
//循环结束后 max(a,b) 就是答案  
int gcd(int a, int b) {return !b ? a : gcd(b, a % b);}
```

3 扩展欧几里得

扩展欧几里得的典型应用是解决形如 $a * x + b * y = c$ 的二元一次方程的解的存在性问题以及求出特解和通解。

3.1 解的存在性问题

其实我们通过辗转相减就可以观察出来, a, b 辗转相减的时候相当于 $a * x + b * y$ 中的 x, y 在不断变化的过程, 比如 $a - b, a - 2 * b, 3 * b - a$, 我们通过前面知道这个过程一定能得到最大公约数, 所以 $a * x + b * y = gcd(a, b)$ 一定有解, 那么是否 $gcd(a, b) \neq c$ 的时候就无解呢? 首先我们知道 c 是 $gcd(a, b)$ 的倍数才可能有解, 因为左边一定含有因子 $gcd(a, b)$, 左边等于右边, 那么右边也一定含有 $gcd(a, b)$, 因此我们可以通过两边同时除以最大公约数得到一个新的式子

$$k_1 * x + k_2 * y = c/g, \quad gcd(k_1, k_2) = 1$$

由辗转相减可得: $k_1 * x + k_2 * y = 1$ 一定有解, 所以 $k_1 * x + k_2 * y = c/g$ 一定有解, 所以当 c 是 $gcd(a, b)$ 的倍数的时候, 方程一定有解

3.2 特解与通解

所以我们只要求解形如

$$a * x + b * y = gcd(a, b)$$

然后将解扩大 $c/gcd(a, b)$ 倍就可以解出原方程的解了

观察到这个式子本质其实就是 a 和 b 辗转相减的过程, 假设已经求出了

$$(a - b) * x_1 + b * y_1 = gcd(a, b)$$

的解 x_1, y_1 , 那么变换一下得到 $a * x_1 + b * (y_1 - x_1) = gcd(a, b)$, 我可以得出

$$x = x_1, y = y_1 - x_1$$

得到了原方程的解, 所以我们可以将原问题变成一个规模更小的子问题, 然后根据子问题的答案推出原问题的答案, 考虑到这个辗转相减可以用辗转相除来替代, 我们可以将原问题变成求解

$$b * x_1 + a \% b * y_1 = gcd(a, b)$$

由

$$a \% b = a - a / b * b$$

代入得

$$b * x_1 + (a - a / b * b) * y_1 = \gcd(a, b)$$

等价于

$$a * y_1 + b * (x_1 - a / b * y_1) = \gcd(a, b)$$

得到

$$x = y_1, y = x_1 - a / b * y_1$$

因此我们只需一直缩小问题规模，直到变成 $\gcd(a, b) * x + 0 * y = \gcd(a, b)$ ，然后得到一组特解 $(1, 0)$ ，再将这组特解反推回去，得到一开始的方程的一组特解。这个过程可以用一个递归函数来实现。

exgcd

```
int extgcd(int a, int b, int &x, int &y) {  
    int d = a;  
    if(b != 0) {  
        d = extgcd(b, a % b, x, y);  
        x -= (a / b) * y;  
        std::swap(x, y);  
    } else {  
        x = 1; y = 0;  
    }  
    return d;  
}
```

函数执行完毕后代码里面的 (x, y) 就是 $a * x + b * y = \gcd(a, b)$ 的一组特解，通解的变化规律就是 x 和 y 的值往相反的方向变化，比如 x 变大一些， y 变小一些，使得答案不变，设这个变化的最小单位值为 d_1, d_2

$$a * (x + d_1) + b * (y - d_2) = \gcd(a, b)$$

$$a * d_1 = b * d_2$$

两边同除 $\gcd(a, b)$ 令 $k_1 = a / \gcd(a, b)$, $k_2 = b / \gcd(a, b)$

$$k_1 * d_1 = k_2 * d_2$$

这个时候 k_1, k_2 互质，显然 $d_1 = k_2, d_2 = k_1$ ，可得通解为

$$(x + k * d_1, y - k * d_2)$$

即

$$(x + k * b / \gcd(a, b), y - k * a / \gcd(a, b))$$

3.3 关于特解的绝对值大小

我们通过递归函数的实现可以观察到，解的绝对值是跟 a, b 的绝对值同一个级别的

4 欧拉函数

4.1 定义

欧拉函数 $\varphi(n)$ 表示小于或者等于 n 的正整数中与 n 互质的数的数目, 例如 $\varphi(8) = 4$, 因为 1, 3, 5, 7 与 8 互质。

前 20 个欧拉函数为 1 1 2 2 4 2 6 4 6 4 10 4 12 6 8 8 16 6 18 8

4.2 欧拉函数值

特殊的 $\varphi(1) = 1$

若 n 是质数 p 的 k 次幂,

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

相当于总数减去 p 的倍数

利用类似的想法, 我们发现 $\varphi(p_1^{k_1} * p_2^{k_2}) = \varphi(p_1^{k_1}) * \varphi(p_2^{k_2})$, 等价于总数减去 p_1 的倍数, 减去 p_2 的倍数, 再加上 p_1, p_2 的倍数, 这个其实就是欧拉函数的积性性质,

积性函数

若 m, n 互质, 有 $f(mn) = f(m) * f(n)$, 那么称函数 f 为积性函数

因此若质因数分解后

$$n = p_1^{k_1} p_2^{k_2} \dots p_r^{k_r}$$
$$\varphi(n) = \prod_{i=1}^r p_i^{k_i-1} (p_i - 1) = n * \prod_{i=1}^r \left(\frac{p_i - 1}{p_i} \right)$$

4.3 欧拉函数常用性质总结

1. 欧拉函数是积性函数, 当正整数 m, n 互质时, $\varphi(m * n) = \varphi(m) * \varphi(n)$
2. 当 n 为奇数时, $\varphi(2 * n) = \varphi(n)$
3. $\varphi(n) = \varphi(n/p) * p$, (p 能整除 n/p), 可以利用上一小节最后一个式子来推
4. $\varphi(n) = \varphi(n/p) * (p-1)$, (p 与 n/p 互质, 且 p 是质数)
5. $\sum_{d|n} \varphi(d) = n$

性质 3 证明:

性质 4 证明: 利用欧拉函数的积性性质即可

性质 5 证明: 设 $f(n) = \sum_{d|n} \varphi(d)$. 若 n, m 互质,

$$f(nm) = \sum_{d|nm} \varphi(d) = \left(\sum_{d|n} \varphi(d) \right) * \left(\sum_{d|m} \varphi(d) \right) = f(n) * f(m)$$

, 所以 $f(n)$ 是积性函数.

$$f(p^m) = \sum_{d|p^m} \varphi(d) = \varphi(1) + \varphi(p) + \varphi(p^2) + \dots + \varphi(p^m) = p^m$$

因此

$$f(n) = \prod_{i=1}^m f(p_i^{c_i}) = \prod_{i=1}^m p_i^{c_i} = n$$

代码实现:

法一: 利用普通筛法实现, 直接根据定义来计算

普通筛法实现欧拉函数

```
void init(int n) {
    phi[1]=1;
    for(int i=2;i<=n;i++)
        phi[i]=i;
    for(int i=2;i<=n;i++)
        if(phi[i]==i)
            for(int j=i;j<=n;j+=i)
                phi[j]=phi[j]/i*(i-1); //先进行除法是为了防止中间数据的溢出
}
```

法二: 利用线性筛递推实现, 利用到了第 3 和第 4 个性质

线性筛法实现欧拉函数

```
void init(int n) {
    phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!flag[i]) {
            p[tot++] = i;
            phi[i] = i - 1; // 素数的欧拉函数值等于素数-1
        }
        for (int j = 0; j < tot; j++) {
            if (i * p[j] > n) {
                break;
            }
            flag[i * p[j]] = true;
            if (i % p[j] == 0) {
                phi[i * p[j]] = phi[i] * p[j]; //性质3
                break;
            }
            phi[i * p[j]] = phi[i] * (p[j] - 1); // 性质4
        }
    }
}
```

5 欧拉定理

欧拉定理的定义是：如果 a, n 为正整数, 且 $\gcd(a, n) = 1$, 则

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

证明：

1: 令集合

$$S = \{x_1, x_2, \dots, x_{\varphi(n)}\}$$

表示所有的小于 n 并且与 n 互质的数

$$T = \{a * x_1 \% n, a * x_2 \% n, \dots, a * x_{\varphi(n)} \% n\}$$

由于 $\gcd(a, n) = 1, \gcd(x_i, n) = 1$, 所以 $\gcd(a * x_i, n) = 1$, 所以

$$\gcd(a * x_i \% n, n) = 1$$

容易通过反证得出 S 中的元素是两两不相同的, 因此集合 S 跟 T 是相同的, 所以

$$\begin{aligned} & a^{\varphi(n)} * x_1 * x_2 * \dots * x_{\varphi(n)} \% n \\ & \equiv a * x_1 * a * x_2 * \dots * a * x_{\varphi(n)} \% n \\ & \equiv x_1 * x_2 * \dots * x_{\varphi(n)} \% n \end{aligned}$$

所以, 欧拉定理得证

6 费马小定理

费马小定理是欧拉定理的特殊情况，即当 n 是质数的时候

$$a^{n-1} \equiv 1 \pmod{n}$$

7 乘法逆元

当我们需要计算 $a/b \% c$ ，又由于 a, b 在运算过程中会变得非常大，不能用高精度保存的时候，典型的就算组合数取模，这个时候可以利用乘法逆元将除法转换成乘法，并提前进行取模运算。设 $inv * b \equiv 1 \pmod{n}$ 假设

$$a/b = k * c + r$$

两边同乘以 b 可得

$$a = k * b * c + b * r$$

两边同乘以 inv 可得

$$a * inv \equiv k * c + r \pmod{n}$$

到此，除法已经转换成了乘法，所以我们只需要求出 inv 就可以了， inv 的求解本质上就是解一个二元一次方程

$$inv * b + k * n = 1$$

的一个正整数解，利用扩展欧几里得求解即可。

特殊的，当 n 是质数的时候，利用费马小定理， $inv = b^{n-2} \% n$

7.1 线性递推求逆元

P 是质数

$$inv(i) = (P - P/i) * inv(P \% i) \% P$$

令 $t = P/i, k = P \% i$, 则

$$t * i + k \equiv 0 \pmod{P}$$

,

$$-t * i \equiv k(\text{mod } P)$$

两边同除以 $i * k$

$$-t * \text{inv}(k) \equiv \text{inv}(i)(\text{mod } P)$$

将 t, k 代入即可

8 中国剩余定理

设 $m_1, m_2, m_3, \dots, m_n$ 两两互质, $m = \prod_{i=1}^n m_i, M_i = m/m_i, s_i$ 是同余方程 $M_i s_i \equiv 1(\text{mod } m_i)$ 的一个解。对于任意的 n 个整数 a_1, a_2, \dots, a_n , 方程组

$$\left\{ \begin{array}{l} x \equiv a_1(\text{mod } m_1) \\ x \equiv a_2(\text{mod } m_2) \\ \dots \\ x \equiv a_n(\text{mod } m_n) \end{array} \right.$$

的解为 $x = \sum_{i=1}^n a_i * M_i * s_i$