ACM 算法模板

Otter.put

2025.05.26

目录

1 基础算法

1.1 高精度

1.1.1 高精度加法

C = A + B，满足 A ≥ 0, B ≥ 0；A、B、C 均为逆序 vector（即高位在前，低位在后）

Listing 1: 高精度加法

```
vector<int> add(vector<int> &A, vector<int> &B) {
    if (A.size() < B.size()) return add(B, A);

    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i ++ )
    {
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }

    if (t) C.push_back(t);
    return C;
}
```

1.1.2 高精度减法

C = A - B，满足 A ≥ B, A ≥ 0, B ≥ 0；A、B、C 均为逆序 vector（即高位在前，低位在后）

Listing 2: 高精度减法

```
vector<int> sub(vector<int> &A, vector<int> &B) {
    vector<int> C;
    for (int i = 0, t = 0; i < A.size(); i ++ ) {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        if (t < 0) t = 1;
        else t = 0;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

1.1.3 高精度乘低精度

C = A * B，满足 A $\geq$ 0, B > 0；A、C 均为逆序 vector（即高位在前，低位在后）

Listing 3: 高精度乘低精度

```
vector<int> mul(vector<int> &A, int b) {
    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size() || t; i ++ ) {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t /= 10;
    }

    return C;
}
```

1.1.4 高精度乘高精度

C = A * B，满足 A $\geq$ 0, B > 0；A、B、C 均为逆序 vector（即高位在前，低位在后）

Listing 4: 高精度乘高精度

```
bool is_zero(vector<int>& A) {
    return A.size() == 1 && A[0] == 0;
}

vector<int> mul(vector<int>& A, vector<int>& B) {
    if(is_zero(A) || is_zero(B))
        return {0};

    int m = A.size(), n = B.size();
    vector<int> C(m + n, 0);

```

```
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            C[i + j] += A[i] * B[j];     // 累加到对应位置

    // 统一处理进位
    int carry = 0;
    for (int i = 0; i < m + n; i++) {
        int total = C[i] + carry;
        C[i] = total % 10;
        carry = total / 10;
    }

    if (carry)
        C.push_back(carry);

    while (C.size() > 1 && C.back() == 0)
        C.pop_back();
    return C;
}
```

1.1.5 高精度除低精度

A / b = C ... r，A ≥ 0, b > 0；A、C 均为逆序 vector（即高位在前，低位在后）

Listing 5: 高精度除低精度

```
vector<int> div(vector<int> &A, int b, int &r) {
    vector<int> C;
    r = 0;
    for (int i = A.size() - 1; i >= 0; i -- ) {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r %= b;
    }
    reverse(C.begin(), C.end());
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

1.1.6 封装为类 + 压位高精度

注意，我们封装的高精度类为了实现的便捷没有符号的概念。同时，使用压位高精度的思想来提升运算速度。

Listing 6: 压位高精度类

```
const int BASE = 10000;
const int BASE_DIGITS = 4;

struct Big {
```

```
    vector<int> digits;

    // 当前数 > other，返回 1；小于返回 -1；等于返回 0
    int cmp(const Big& other) const {
        if (digits.size() != other.digits.size())
            return digits.size() > other.digits.size() ? 1 : -1;

        for (int i = digits.size() - 1; i >= 0; i--)
            if (digits[i] != other.digits[i])
                return digits[i] > other.digits[i] ? 1 : -1;

        return 0;
    }

    Big() : digits({0}) {}

    Big(long long num) {
        if (num == 0) {
            digits = {0};
            return;
        }

        while (num > 0) {
            digits.push_back(num % BASE);
            num /= BASE;
        }
    }

    Big(const string& str) {
        for (int i = str.size() - 1; i >= 0; i -= BASE_DIGITS) {
            int digit = 0;
            int end = max(0, i - BASE_DIGITS + 1);
            for (int j = end; j <= i; j++) {
                digit = digit * 10 + (str[j] - '0');
            }
            digits.push_back(digit);
        }

        while (digits.size() > 1 && digits.back() == 0)
            digits.pop_back();
    }

    Big(const Big& other) : digits(other.digits) {}

    Big& operator=(const Big& other) {
        digits = other.digits;
        return *this;
    }

```

```
54      Big operator+(const Big& other) const {
55          Big result;
56          result.digits.clear();
57
58          int carry = 0;
59          int maxSize = max(digits.size(), other.digits.size());
60
61          for (int i = 0; i < maxSize || carry; i++) {
62              int sum = carry;
63              if (i < digits.size()) sum += digits[i];
64              if (i < other.digits.size()) sum += other.digits[i];
65
66              carry = sum >= BASE;
67              if (carry) sum -= BASE;
68
69              result.digits.push_back(sum);
70          }
71
72          return result;
73      }
74
75      // 假设当前数 >= other
76      Big operator-(const Big& other) const {
77          Big result;
78          result.digits.clear();
79
80          int borrow = 0;
81          for (int i = 0; i < digits.size(); i++) {
82              int diff = digits[i] - borrow;
83              if (i < other.digits.size()) diff -= other.digits[i];
84
85              borrow = 0;
86              if (diff < 0) {
87                  diff += BASE;
88                  borrow = 1;
89              }
90
91              result.digits.push_back(diff);
92          }
93
94          // 去除前导零
95          while (result.digits.size() > 1 && result.digits.back() == 0)
96              result.digits.pop_back();
97
98          return result;
99      }
100
101     Big operator*(const int& other) const {
102         if (other == 0) return Big(0);
```

```
103
104         Big result;
105         result.digits.clear();
106
107         long long carry = 0;
108         for (int i = 0; i < digits.size() || carry; i++) {
109             if (i < digits.size()) carry += (long long)digits[i] * other;
110
111             result.digits.push_back(carry % BASE);
112             carry /= BASE;
113         }
114
115         while (result.digits.size() > 1 && result.digits.back() == 0)
116             result.digits.pop_back();
117
118         return result;
119     }
120
121     Big operator*(const Big& other) const {
122         if (other == 0) return Big(0);
123
124         Big result;
125         result.digits.resize(digits.size() + other.digits.size(), 0);
126
127         for (int i = 0; i < digits.size(); i++) {
128             long long carry = 0;
129             for (int j = 0; j < other.digits.size() || carry; j++) {
130                 long long product = result.digits[i + j] + carry;
131                 if (j < other.digits.size())
132                     product += (long long)digits[i] * other.digits[j];
133
134                 result.digits[i + j] = product % BASE;
135                 carry = product / BASE;
136             }
137         }
138
139         // 去除前导零
140         while (result.digits.size() > 1 && result.digits.back() == 0)
141             result.digits.pop_back();
142
143         return result;
144     }
145
146     Big operator/(const int& divisor) const {
147         Big result;
148         result.digits.resize(digits.size());
149
150         long long remainder = 0;
151         for (int i = digits.size() - 1; i >= 0; i--) {
```

```
            long long current = remainder * BASE + digits[i];
            result.digits[i] = current / divisor;
            remainder = current % divisor;
        }

        // 去除前导零
        while (result.digits.size() > 1 && result.digits.back() == 0)
            result.digits.pop_back();

        return result;
    }

    int operator%(const int& divisor) const {
        long long remain = 0;
        for (int i = digits.size() - 1; i >= 0; i--) {
            remain = (remain * BASE + digits[i]) % divisor;
        }
        return remain;
    }

    Big operator^(long long exponent) const {
        Big base(*this);
        Big result(1);

        while (exponent > 0) {
            if (exponent & 1)
                result = result * base;

            base = base * base;
            exponent >>= 1;
        }

        return result;
    }

    bool operator<(const Big& other) const { return cmp(other) == -1; }
    bool operator<=(const Big& other) const { return cmp(other) <= 0; }
    bool operator>(const Big& other) const { return cmp(other) == 1; }
    bool operator>=(const Big& other) const { return cmp(other) >= 0; }
    bool operator==(const Big& other) const { return cmp(other) == 0; }
    bool operator!=(const Big& other) const { return cmp(other) != 0; }

    bool operator<(long long num) const { return *this < Big(num); }
    bool operator>(long long num) const { return *this > Big(num); }
    bool operator<=(long long num) const { return *this <= Big(num); }
    bool operator>=(long long num) const { return *this >= Big(num); }
    bool operator==(long long num) const { return *this == Big(num); }
    bool operator!=(long long num) const { return *this != Big(num); }

```

```
    void print() const {
        cout << digits.back(); // 最高位不需要前导零

        for (int i = digits.size() - 2; i >= 0; i--)
            cout << setw(BASE_DIGITS) << setfill('0') << digits[i];
    }

    string toString() const {
        string result;
        result += to_string(digits.back());

        for (int i = digits.size() - 2; i >= 0; i--) {
            string segment = to_string(digits[i]);
            result += string(BASE_DIGITS - segment.size(), '0') + segment;
        }
        return result;
    }

    friend ostream& operator<<(ostream& os, const Big& num) {
        os << num.digits.back();
        for (int i = num.digits.size() - 2; i >= 0; i--)
            os << setw(BASE_DIGITS) << setfill('0') << num.digits[i];
        return os;
    }
};
```

2 搜索

3 数学

3.1 数论

3.1.1 判定质数

试除法判定某个数是不是质数，时间复杂度 O($\sqrt{n}$)。

Listing 7: 判定素数

```
bool is_prime(int n) {
    if(n < 2)
        return false;
    for(int i = 2; i <= sqrt(n); i++)
        if(n % i == 0)
            return false;
    return true;
}
```

3.1.2 埃氏筛

时间复杂度 O(nloglogn)。

Listing 8: 埃氏筛

```
const int N = 1e6 + 5;
int v[N], prime[N], m;
void primes(int n) {
    memset(v, 0, sizeof(v));
    for(int i = 2; i <= n; i++) {
        if(v[i])
            continue;
        prime[++m] = i;
        for(int j = i; j <= n / i; j++)
            v[i * j] = 1;
    }
}
```

3.1.3 线性筛

线性素数筛，最后 prime[1 ∼ m] 中保存着 1 ∼ m 范围内的所有素数，v[i] 保存着 i 的最小质因子。时间复杂度 O(n)。

Listing 9: 线性筛

```
const int N = 1e6 + 5;
int v[N], prime[N], m;
void primes(int n) {
    memset(v, 0, sizeof(v));
    m = 0;
    for(int i = 2; i <= n; i++) {
        if(v[i] == 0)
```

```
            v[i] = i, prime[++m] = i;

        for(int j = 1; j <= m; j++) {
            if(prime[j] > v[i] || prime[j] > n / i)
                break;
            v[i * prime[j]] = prime[j];
        }
    }
}
```

3.1.4 分解质因数

时间复杂度 $O(\sqrt{n})$。

Listing 10: 分解质因数

```
const int N = 1e5 + 5;
int p[N], c[N], m;

void divide(int n) {
    m = 0;
    for(int i = 2; i <= sqrt(n); i++) {
        if(n % i == 0) {
            p[++m] = i, c[m] = 0;
            while(n % i == 0)
                n /= i, c[m]++;
        }
    }

    if(n > 1)
        p[++m] = n, c[m] = 1;
}
```

3.1.5 求 n 的正约数集合

试除法，时间复杂度 $O(\sqrt{n})$，约数个数上界 $2\sqrt{n}$。

Listing 11: 试除法求 n 的正约数集合

```
int factor[1600], m = 0;
void divide(int n) {
    for(int i = 1; i * i <= n; i++)
        if(n % i == 0) {
            factor[++m] = i;
            if(i != n / i)
                factor[++m] = n / i;
        }
}
```

3.1.6 求 1 ∼ n 的正约数集合

倍数法，时间复杂度 NlogN。

Listing 12: 倍数法求 1 ∼ n 的正约数集合

```
vector<int> factor[500010];
void times(int n) {
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n / i; j++)
            factor[i * j].push_back(i);
}
```

3.1.7 欧几里得算法求 gcd

时间复杂度 $\log(a+b)$。当需要做高精度运算时，建议使用更相减损术代替。

Listing 13: 欧几里得算法求 gcd

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

3.1.8 Stein 算法求大整数 gcd

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。利用 Stein 算法，时间复杂度 O(log(n))。为了优化常数，可以使用压位高精度。

若 $a=b$，则 $\gcd(a,b)=a$，否则：

- 若 a,b 均为偶数，则 $\gcd(a,b)=2\gcd(\frac{a}{2},\frac{b}{2})$
- 若 a 为偶数，b 为奇数，则 $\gcd(a,b)=\gcd(\frac{a}{2},b)$
- 若 a,b 同为奇数，则 $\gcd(a,b)=\gcd(a-b,b)$

Listing 14: Stein 算法

```
// 基于压位高精度类实现
Big Stein(Big A, Big B) {
    int shift = 0;
    while(A != 0 && B != 0) {
        bool f1 = (A.digits[0] % 2) == 0, f2 = (B.digits[0] % 2) == 0;

        if(f1 && f2)
            A = A / 2, B = B / 2, shift++;
        else if(f1)
            A = A / 2;
        else if(f2)
            B = B / 2;
        else {
            if(A >= B)
```

```
15                  A = A - B;
16              else
17                  B = B - A;
18          }
19      }
20
21      Big result = Big(A == 0 ? B : A);
22      return result * (Big(2) ^ shift);
23  }
```

3.1.9 裴蜀定理

对任意整数 a, b，存在一对整数 x, y，s.t. $ax + by = \gcd(a, b)$

逆定理：

设 a, b 是不全为 0 的整数，若 $d > 0$ 是 a, b 的公因数，且存在整数 x, y，s.t. $ax + by = d$，则 $d = \gcd(a, b)$。

设 a, b 是不全为 0 的整数，且存在整数 x, y，s.t. $ax + by = 1$，则 a, b 互质。

3.1.10 扩展欧几里得算法

用于求解 $ax + by = \gcd(a, b)$ 的一组特解。

Listing 15: 扩展欧几里得算法

```
1   // 返回 a b 的最大公约数 d, 并找到一组特解 x0 y0
2   int exgcd(int a, int b, int &x, int &y) {
3       if(b == 0) {
4           x = 1;
5           y = 0;
6           return a;
7       }
8       int d = exgcd(b, a % b, x, y);
9       int z = x;
10      x = y;
11      y = z - y * (a / b);
12      return d;
13  }
```

更一般地，对于方程 $ax + by = c$，它有解当且仅当 $d \mid c$。我们可以先求出 $ax + by = d$ 的一组特解 x_0, y_0，然后就得到了 $ax + by = c$ 的一组特解 $(c/d)x_0, (c/d)y_0$。则 $ax + by = c$ 的通解可表示为：

$$x = \frac{c}{d}x_0 + k\frac{b}{d},\ y = \frac{c}{d}y_0 - k\frac{a}{d}$$

其中 k 取遍整数集合。

3.1.11 线性同余方程

求一个整数 x 满足 $ax \equiv b \pmod{m}$，或者给出无解。

方程可改写为 $ax + my = b$，设 $d = \gcd(a, m)$ 故线性同余方程有解当且仅当 $d \mid b$，利用丢番图方程相关知识，知道 $x = x_0 \times b/d$ 是原线性同余方程的一个特解；方程恰有 d 个模 m 不同余的解，为 $x_i = x + i \times (m/d),\ 0 \le i \le d - 1$。

3.1.12 乘法逆元

若 $ax \equiv 1 \pmod m$，则称 x 为 a 模 m 的逆元，记作 $a^{-1} \pmod m$。

根据费马小定理，当 m 为质数时，b^{m-2} 即为 b 的乘法逆元。

只有 $\gcd(a, m) = 1$ 时才存在乘法逆元，可用扩展欧几里得算法求逆元（本质上还是解线性同余方程）。

Listing 16: 乘法逆元

```
// 扩展欧几里得算法求最小正整数逆元
void exgcd(int a,int b, int &x, int &y) {
    if(!b) {
        x = 1;
        y = 0;
        return ;
    }
    exgcd(b, a % b, y, x);
    y -= a / b * x;
}

// 求 n mod p 的逆元
// 前提是 n 与 p 互质
int inv(int n, int p) {
    int x, y;
    exgcd(n, p, x, y);
    return (x + p) % p;
}
```

3.1.13 线性求 1 ~ N 的逆元

重要前提：p 是质数。

显然，1 对 p 的逆元是 1。

对于某个数 i，利用带余除法，有 $p = k \times i + j$，有 $k \times i + j \equiv 0 (\bmod\ p)$，有 $k \times j^{-1} + i^{-1} \equiv 0 (\bmod\ p)$，即 $i^{-1} \equiv -\lfloor \frac{p}{i} \rfloor \times (p \bmod\ i)^{-1} (\bmod\ p)$。

$$i^{-1} \equiv \begin{cases} 1, & \text{if } i = 1, \\ -\left\lfloor \frac{p}{i} \right\rfloor (p \bmod i)^{-1}, & \text{otherwise.} \end{cases} \pmod p$$

Listing 17: 线性求逆元

```
// 线性求 1 ~ n mod p 的逆元
// 前提是 p 是质数
int get_inv(int n, int p) {
    inv[1] = 1;
    for (int i = 2; i <= n; ++i) {
        inv[i] = (long long)(p - p / i) * inv[p % i] % p;
    }
}
```

3.1.14 欧拉函数

$1 \sim N$ 中与 N 互质的数的个数被称为欧拉函数。

若在算数基本定理中，$N = p_1^{c_1} p_2^{c_2} \cdots p_m^{c_m}$，则

$$\varphi(N) = N \times \frac{p_1 - 1}{p_1} \times \frac{p_2 - 1}{p_2} \times \cdots \times \frac{p_m - 1}{p_m} = N \times \prod_{\text{prime } p|N} (1 - \frac{1}{p})$$

Listing 18: 欧拉函数

```
// 求单个数的 Euler 函数，时间复杂度 O(sqrt(n))
int phi(int n) {
    int ans = n;
    for(int i = 2; i <= sqrt(n); i++)
        if(n % i == 0) {
            ans = ans / i * (i - 1);
            while(n % i == 0)
                n /= i;
        }
    if(n > 1)
        ans = ans / n * (n - 1);
    return ans;
}
```

欧拉函数的性质

1. φ 是积性函数
2. 若 f 是积性函数，且在算数基本定理中有 $n = p_1^{c_1} p_2^{c_2} \cdots p_m^{c_m}$，则 $f(n) = \prod_{i=1}^{m} f(p_i^{c_i})$
3. $\forall\, n > 1$，$1 \sim n$ 中与 n 互质的数和为 $n \times \varphi(n)/2$
4. 设 p 为质数，若 $p \mid n$ 且 $p^2 \mid n$，则 $\varphi(n) = \varphi(n/p) \times p$
5. 设 p 为质数，若 $p \mid n$ 且 $p^2 \nmid n$，则 $\varphi(n) = \varphi(n/p) \times (p - 1)$
6. $\sum_{d|n} \varphi(d) = n$
7. 若 $n = p^k$，其中 p 是质数，那么 $\varphi(n) = p^k - p^{k-1}$
8. 若 n 是质数，显然有 $\varphi(n) = n - 1$

第二条对所有积性函数都适用，后面几条仅对欧拉函数适用。

3.1.15 求 $2 \sim N$ 中每个数的欧拉函数

利用线性筛法中，每个数字都会被自己的最小质因子筛掉的性质，以及上面的性质 4、5，进行优化。时间复杂度 O(n)。

Listing 19: 求 2 ∼ N 中每个数的欧拉函数

```
int v[maxn], prime[maxn], phi[maxn];
void euler(int n) {
    // v 用来记录最小质因子
```

```
    memset(v, 0, sizeof(v));
    m = 0;
    for(int i = 2; i <= n; i++) {
        if(v[i] == 0) {
            v[i] = i, prime[++m] = i;
            phi[i] = i - 1;
        }
        for(int j = 1; j <= m; j++) {
            if(prime[j] > v[i] || prime[j] > n / i)
                break;
            v[i * prime[j]] = prime[j];
            phi[i * prime[j]] = phi[i] * (i % prime[j] ? prime[j] - 1 : prime[j]);
        }
    }
}
```

3.1.16 欧拉定理

费马小定理

若 p 是质数，则 $a^p \equiv a \pmod{p}$，实际上是欧拉定理的特殊情况。

欧拉定理

若正整数 a,n 互质，则 $a^{\varphi(n)} \equiv 1 \pmod{n}$。

3.1.17 扩展欧拉定理

用于降幂：

$$a^b \equiv \begin{cases} a^{b \bmod \varphi(n)}, & \gcd(a,n)=1, \\ a^b, & \gcd(a,n)\neq 1, b<\varphi(n), \\ a^{(b \bmod \varphi(n))+\varphi(n)}, & \gcd(a,n)\neq 1, b\geq \varphi(n). \end{cases} \pmod{n}$$

Listing 20: 扩展欧拉定理

```
// 其中 a 和 n 都是 int; b 是大整数
int ExEuler(int a, Big b, int n) {
    int phi_n = phi(n);
    if(gcd(a, n) == 1)
        return quickPow(a, b % phi_n, n);
    else if(b < phi_n)
        return quickPow(a, b, n);
    return quickPow(a, b % phi_n + phi_n, n);
}
```

3.1.18 中国剩余定理求解线性同余方程组

设 $m_1, m_2, \cdots, m_n$ 是两两互质的整数，$m = \prod_{i=1}^{n} m_i$，$M_i = m/m_i$，t_i 是线性同余方程 $M_i t_i \equiv 1 \pmod{m_i}$ 的一个解。对于任意的 n 个整数 $a_1, a_2, \cdots, a_n$，方程组

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

有整数解，解为 $x = \sum_{i=1}^{n} a_i M_i t_i$，在模 m 意义下有唯一解。

Listing 21: 中国剩余定理

```
// 求同余方程组最小非负整数解
int intChina(int r) {
    int Mi, x0, y0, d, ans = 0;
    int M = 1;
    for(int i = 1; i <= r; i++)
        M *= m[i];
    for(int i = 1; i <= r; i++) {
        Mi = M / m[i];
        // 求出每个 ti 的逆元
        exgcd(Mi, m[i], x0, y0);
        ans = (ans + Mi * x0 * a[i]) % M;
    }

    return (ans + M) % M;
}
```

3.1.19 扩展中国剩余定理

中国剩余定理要求 N 个模数两两互质，其实这个条件表苛刻。当 N 个模数不满足两两互质时，中国剩余定理不再适用。可以考虑用数学归纳法，假设已经求出了前 $k-1$ 个方程构成的方程组的一个解 x。记 $m = \mathrm{lcm}(m_1, m_2, \cdots, m_{k-1})$，则 $x + i \times m$ $(i \in Z)$ 是前 $k-1$ 个方程的通解。

考虑第 k 个方程,求出一个整数 t,s.t. $x + t \times m \equiv a_k \pmod{m_k}$,该方程等价于 $t \times m \equiv a_k - x \pmod{m_k}$，其中 t 是未知量，这就是一个线性同余方程，可以使用 exgcd 求解。若有解，则 $x' = x + t \times m$ 就是前 k 个方程构成的方程组的一个解。

简单来说，就是使用 n 次扩展欧几里得算法。

Listing 22: 扩展中国剩余定理

```
// EXCRT, 用来解决模数不一定两两互质的情况
// 容易溢出时将数据类型都换为 __int128_t
ll excrt(int n) {
    // x 保存前 k - 1 个方程的解, M 保存前 k - 1 个数的 lcm
    ll x = a[1] % m[1], M = m[1];
    for(int i = 2; i <= n ;i++) {
        ll t, y;
        ll d = exgcd(M, m[i], t, y);
        t = (a[i] - x) / d * t;
        x = x + t * M;
        M = lcm(M, m[i]);
```

```
12          x = (x % M + M) % M;
13      }
14      return x;
15  }
```

3.1.20 高次同余方程

给定整数 a, b, p，其中 a, p 互质，求一个非负整数 x，s.t. $a^x \equiv b \pmod p$。

Listing 23: BSGS 求解高次同余方程

```
1  // Baby Step, Giant Step 算法
2  // 注意: a 与 p 必须互质
3  // 无解时返回 -1
4  int BSGS(int a, int b, int p) {
5      map<int, int> hash;
6      hash.clear();
7      b %= p;
8      int t = (int)sqrt(p) + 1;
9      for(int j = 0; j < t; j++) {
10         int val = (long long)b * pow(a, j, p) % p;
11         hash[val] = j;
12     }
13     a = pow(a, t, p);
14     if(a == 0)
15         return b == 0 ? 1 : -1;
16     for(int i = 0; i <= t; i++) {
17         int val = pow(a, i, p);
18         int j = hash.find(val) == hash.end() ? -1 : hash[val];
19         if(j >= 0 && i * t >= j)
20             return i * t - j;
21     }
22     return -1;
23 }
```

3.1.21 阶乘取模问题

由中国剩余定理，阶乘取模问题可以转化为模数为素数幂 p^α 的情形，可以对 $n!$ 做如下分解：

$$n! = p^{\nu_p(n!)}(n!)_p$$

其中，$\nu_p(n!)$ 表示阶乘 $n!$ 的素因数分解中 p 的幂次，$(n!)_p$ 表示在阶乘 $n!$ 的结果中去除所有 p 的幂次得到的整数。下面将讨论 $(n!)_p$ 在素数幂模下的余数以及幂次 $\nu_p(n!)$ 的具体计算方法。

Wilson 定理

对于自然数 $n > 1$，当且仅当 n 是素数时，$(n-1)! \equiv -1 \pmod n$。

推广之后，可以求出 $(n!)_p$。预处理的时间复杂度为 $O(p^\alpha)$，单次询问的时间复杂度为 $O(\log_p n)$。

Listing 24: 利用推广的 Wilson 定理求 $(n!)_p$

```
1  // Calculate (n!)_p mod pa.
```

```cpp
// pa 是某个素数的幂次
int factmod(int n, int p, int pa) {
    std::vector<int> f(pa);
    f[0] = 1;
    for (int i = 1; i < pa; ++i)
        f[i] = i % p ? (long long)f[i - 1] * i % pa : f[i - 1];
    bool neg = p != 2 || pa <= 4;
    int res = 1;
    while (n > 1) {
        if ((n / pa) & neg)
            res = pa - res;
        res = (long long)res * f[n % pa] % pa;
        n /= p;
    }
    return res;
}
```

Legendre 公式

Legendre 公式可以求出 p 的幂次 $\nu_p(n!)$。它的时间复杂度为 $O(\log n)$。

Listing 25: 利用 Legendre 公式求 $\nu_p(n!)$

```cpp
// Obtain multiplicity of p in n!.
int multiplicity_factorial(int n, int p) {
    int count = 0;
    do {
        n /= p;
        count += n;
    } while (n);
    return count;
}
```

利用上面两个公式可以在 $O(logn)$ 时间内求出阶乘的模（只能求出模数为质数的幂次的情景，当模数不是质数幂次时可以用中国剩余定理进行求解）。

3.2 线性代数

3.2.1 矩阵乘法

矩阵乘法满足结合律、分配率，但不满足交换律。

矩阵相当于状态转移，一维空间状态转移矩阵中 $A_{i,j}$ 表示状态 i 如何递推，得到下一时刻的状态 j。同理可扩展到高维空间（但是建议在代码实现层面，将高维空间压缩为一维空间）。

简单来说，状态转移矩阵中每个位置的值由状态 i 如何转移到状态 j 决定。

Listing 26: 矩阵快速幂

```cpp
// 矩阵乘法加速线性递推：矩阵快速幂
// 时间复杂度 O(n^3logT)，其中 T 是递推总轮数
struct matrix {
    ll a[sz][sz];

```

```
    matrix() {memset(a, 0, sizeof(a));}

    matrix operator*(const matrix& T) const {
    matrix res;
    int r;
    for (int i = 0; i < sz; ++i)
      for (int k = 0; k < sz; ++k) {
        r = a[i][k];
        for (int j = 0; j < sz; ++j)
          res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= mod;
      }
    return res;
  }
}

matrix matQuickPow(matrix a, int b) {
    matrix ret;
    for(int i = 0; i < sz; i++)
        ret.a[i][i] = 1;
    while(b) {
        if(b & 1)
            ret = ret * a;
        a = a * a;
        b >>= 1;
    }
    return ret;
}
```

3.2.2 高斯消元法求解线性方程组

高斯消元用来求解线性方程组。

线性方程组的所有系数可以写成一个 $m \times n$ 的系数矩阵，再加上每个方程等号右侧的常数，可以写成一个 $m \times (n+1)$ 的增广矩阵，然后对增广矩阵进行初等行变换：

- 用一个非零的数乘某一行 - 把其中一行的若干倍加到另一行上 - 交换两行的位置

直到将增广矩阵化为一个上三角矩阵，依次回带，得到一个简化阶梯型矩阵，给出了方程组的解。

高斯消元完成后，若存在系数全为零，常数不为零的行，则方程组无解；若主元恰好有 n 个，方程组有唯一解；若主元有 $k < n$ 个，方程组有无穷多个解。

高斯消元法就是通过初等行变换把增广矩阵变为简化阶梯型矩阵的线性方程组求解算法。

Listing 27: 高斯消元法

```
// 时间复杂度 O(n^3)
// 无解时输出 -1, 无穷多解时输出 0
void gaussElimination(int n) {
    int nwline = 0;
    for(int k = 0; k < n; k++) {
        // 用于行主元选取
        int maxRow = nwline;
```

```
        for(int i = nwline + 1; i < n; i++)
            if(fabs(p[i][k]) > fabs(p[maxRow][k]))
                maxRow = i;

        if(fabs(p[maxRow][k]) < eps)
            continue;

        // 交换当前行和最大元素所在行
        for(int i = 0; i <= n; i++)
            swap(p[nwline][i], p[maxRow][i]);

        for(int i = 0; i < n; i++) {
            if(i == nwline)
                continue;
            double mul = p[i][k] / p[nwline][k];
            for(int j = k; j <= n; j++)
                p[i][j] -= p[nwline][j] * mul;
        }
        nwline++;
    }

    // 存在找不到主元的情况
    // 一定是无解或者无穷多解，优先判断无解
    if(nwline < n) {
        while(nwline < n)
            if(!(fabs(p[nwline++][n]) < eps)) {
                printf("-1");
                return ;
            }
        printf("0");
        return ;
    }

    for(int i = 0; i < n; i++)
        printf("%.2lf\n", fabs(p[i][n] / p[i][i]) < eps ? 0 : p[i][n] / p[i][i]);
}
```

3.2.3 高斯消元法求解异或方程组

$$\begin{cases} a_{1,1}x_1 \oplus a_{1,2}x_2 \oplus \cdots \oplus a_{1,n}x_n & = b_1 \\ a_{2,1}x_1 \oplus a_{2,2}x_2 \oplus \cdots \oplus a_{2,n}x_n & = b_2 \\ \cdots & \cdots \\ a_{m,1}x_1 \oplus a_{m,2}x_2 \oplus \cdots \oplus a_{m,n}x_n & = b_1 \end{cases}$$

由于异或满足交换律和结合律，故可以用高斯消元法解决异或方程组。当异或方程组多解时，解的数量就是 2^{cnt}，其中 cnt 是自由变元的个数，因为自由变元任取 0 或 1。

Listing 28: 高斯消元法

```
// 时间复杂度 O(n^2m / w), w 是利用压位进行优化

void gaussElimination(int n) {
    int ans = 1;
    for(int i = 1; i <= n; i++) {
        for(int j = i + 1; j <= n; j++)
            if(a[j] > a[i])
                swap(a[i], a[j]);

        // 消元完毕, 有 i - 1 个主元, n - i + 1 个自由元
        if(a[i] == 0) {
            ans = 1 << (n - i + 1);
            break;
        }

        // 出现 0 = 1, 无解
        if(a[i] == 1) {
            ans = 0;
            break;
        }

        for(int k = n; k; k--)
            if(a[i] >> k & 1) {
                for(int j = 1; j <= n; j++)
                    if(i != j && (a[j] >> k & 1))
                        a[j] ^= a[i];

                break;
            }
    }

    if(ans == 0)
        printf("No Solution!");
    else if(ans != 1)
        printf("%d", ans);
    else {
        for(int i = 1; i <= n; i++)
            printf("%d : %d\n", i, a[i] & 1);
    }
}
```

3.2.4 线性基

线性空间是一个关于向量加法和数乘运算封闭的向量集合。任意选出线性空间中的若干个向量，若其中存在一个向量能被其它向量表出，则称这些向量线性相关，否则称这些向量线性无关。线性空间的极大线性无关子集称为基，基包含的向量个数称为线性空间的维数。

通过原集合 S 的某一最小子集 S_1 内元素相互异或得到的值域与原集合 S 相互异或所得到的值

域相同。也就是说在 mod 2 意义下，有 n 个长度为 m 的向量，这 n 个向量的线性基为其所组成的线性空间的基。

1. 原序列里面的任意一个数都可以由线性基里面的一些数异或得到
2. 线性基里面的任意一些数异或起来都不能得到 0
3. 线性基里面的数的个数唯一，并且在保持性质一的前提下，数的个数是最少的

增量法构造线性基

Listing 29: 增量法构造线性基

```
void addnum(ll x) {
    for(int i = 60; i >= 0; i--)
        if((x >> i) & 1) {
            if(d[i])
                x ^= d[i];
            else {
                d[i] = x;
                break;
            }
        }
}
```

线性基求异或最大值

Listing 30: 线性基求异或最大值

```
ll getmax() {
    ll res = 0;
    for(int i = 60; i >= 0; i--)
        if((res ^ d[i]) > res)
            res ^= d[i];
    return res;
}
```

线性基求异或最小值

Listing 31: 线性基求异或最小值

```
ll getmin() {
    ll res = 0, cnt = 0;
    for(int i = 60; i >= 0; i--)
        if(d[i])
            cnt++, res = d[i];
    return cnt < n ? 0 : res;
}
```

线性基求异或第 k 大值（结果去重）

可以将线性基消成对角矩阵，这样就得到了一组各个维度上互不重合的线性基，显然他们有明确的大小顺序。

如果原数能异或出 0 的话（即原数集线性相关），那么 0 作为最小值，我们就要找线性基能组成的第 $k-1$ 大的值。

第 j 大的值就是对 j 进行二进制拆分后取走对应的线性基组合。

Listing 32: 线性基求异或第 k 大值（结果去重）

```
// 线性基对角化
void change() {
    for(int i = 60; i >= 0; i--)
        for(int j = i - 1; j >= 0; j--)
            if((d[i] >> j) & 1)
                d[i] ^= d[j];
    for(int i = 0; i <= 60; i++)
        if(d[i])
            d2[cnt++] = d[i];
}

ll query(ll k) {
    if(n > cnt)
        k--;
    if(k >= (1ll << cnt))
        return -1;
    ll ret = 0;
    for(int i = 0; i < cnt; i++)
        if((k >> i) & 1)
            ret ^= d2[i];
    return ret;
}
```

线性基求异或第 k 大值（结果不去重）

求出一组基后，不在线性基中的整数还有 $n-t$ 个，从其中任选若干个，显然有 2^{n-t} 种选法，每种选法与基底结合，由于结果不重复（异或的性质），且所有结果都能被基底表示出来，所以显然恰好遍历去重异或集合一次。

综上，不去重异或集合就是去重异或集合中的 2^t 个整数各重复 2^{n-t} 次形成的。

3.3 组合数学

3.3.1 组合数

$C_m^n = C_{n-m}^n$

$C_m^n = C_{m-1}^n + C_{m-1}^{n-1}$

$C_0^n + C_1^n + \cdots + C_n^n = 2^n$

根据性质 2，可以在时间复杂度 $O(n^2)$ 内求出 $0 \le y \le x \le n$ 的所有组合数 C_y^x：

Listing 33: 求组合数

```
// 预处理出组合数数组
// 时间复杂度 O(n^2)
for(int i = 1; i <= k; i++) {
    c[i][0] = c[i][i] = 1;
    for(int j = 1; j <= i - 1; j++)
        c[i][j] = c[i - 1][j - 1] + c[i - 1][j];
}
```

组合数的结果一般较大，若题目要求出 C_m^n 对一个数 p 取模后的结果，并且 $1 \sim n$ 都存在模 p 乘法逆元，则可以先计算分子 n! mod p，再计算分母 m!(n − m)! mod p 的逆元，乘起来得到 C_m^n mod p，时间复杂度 O(n)。

若在计算阶乘过程中，把 $0 \le k \le n$ 的每个 k! mod p 及其逆元分别保存在两个数组 jc 和 jc_inv 中，则可以在 O(nlogn) 的预处理后，以 O(1) 的时间计算出

$$C_y^x \bmod p = jc[x] \times jc\_inv[y] \times jc\_inv[x-y] \bmod p$$

若题目要求对 C_m^n 进行高精度运算，为了避免除法，可以使用阶乘分解的做法，将分子分母快速分解质因数，在数组中保存各项质因子的指数。然后将分子分母各质因子的指数对应相减，最后把剩余质因子乘起来，时间复杂度 O(nlogn)。

3.3.2 二项式定理

$$(a+b)^n = \sum_{k=0}^{n} C_k^n a^k b^{n-k}$$

3.3.3 多重集的排列数与组合数

多重集的排列数

设 $S = \{n_1a_1, n_2a_2, \cdots, n_ka_k\}$，则 S 的全排列数为

$$\frac{n!}{n_1!n_2!\cdots n_k!}$$

多重集的组合数（$r \le n_i$）

设 $S = \{n_1a_1, n_2a_2, \cdots, n_ka_k\}$，设 $r \le n_i$ $(\forall\, i \in [1,k])$。则从 S 中取出一个 r 个元素的多重集的组合数

$$C_{k-1}^{k+r-1}$$

多重集的组合数

设 $S = \{n_1a_1, n_2a_2, \cdots, n_ka_k\}$，设 $r \le n$。则从 S 中取出一个 r 个元素的多重集的组合数

$$\sum_{p=0}^{k}(-1)^p \sum_{A} \binom{k+r-1-\sum_A n_{A_i}-p}{k-1}$$

3.3.4 Lucas 定理

用于对大组合数取模。

当 p 是素数时，有：

$$C_m^n \equiv C_{m \bmod p}^{n \bmod p} * C_{m/p}^{n/p} (\bmod\ p)$$

即把 n 和 m 当成 p 进制数，对 p 进制下的每一位分别计算组合数，最后再乘起来。

Listing 34: Lucas 定理

```
1 // p 一定要是质数
2 // 为了求解组合数进行预处理
3 ll fac[N], inv[N], fac_inv[N];
```

```
void init(int n) {
    fac[0] = inv[0] = fac_inv[0] = 1;
    fac[1] = inv[1] = fac_inv[1] = 1;
    for(int i = 2; i <= n; i++)
        fac[i] = fac[i - 1] * i % p,
        inv[i] = (ll)(p - p / i) * inv[p % i] % p,
        fac_inv[i] = fac_inv[i - 1] * inv[i] % p;
}

ll c(ll n, ll m) {
    if(n < m)
        return 0;
    return (fac[n] * fac_inv[m]) % p * fac_inv[n - m] % p;
}

ll lucas(ll n, ll m) {
    if(!m)
        return 1;
    return c(n % p, m % p) * lucas(n / p, m / p) % p;
}
```

3.3.5 扩展 Lucas 定理

根据阶乘取模问题一节的 Legendre 公式，可以得到 Kummer 定理。

Kummer 定理

素数 p 在组合数 $\binom{m}{n}$ 中的幂次，恰好是 p 进制下 m 减掉 n 需要借位的次数，亦即：

$$\nu_p\left(\binom{n}{m}\right)=\frac{S_p(m)+S_p(n-m)-S_p(n)}{p-1}.$$

其中 $S_p(n)$ 为 p 进制下 n 的各个数位的和。

当组合数对某质数的幂次取模时，有

$$\binom{n}{k}=p^K\frac{(n!)_p}{(k!)_p((n-k)!)_p}.$$

可以通过阶乘取模问题一节中的 **Wilson 定理的推广**来求解。

当模数 m 是一般的合数时，先进行质因数分解：

$$m=p_1^{\alpha_1}p_2^{\alpha_2}\cdots p_s^{\alpha_s}$$

然后，分别计算出模 $p_i^{\alpha_i}$ 下组合数 $\binom{n}{k}$ 的余数，就得到 s 个同余方程：

$$\begin{cases}\binom{n}{k}\equiv r_1, & (\mathrm{mod}p_1^{\alpha_1})\\ \binom{n}{k}\equiv r_2, & (\mathrm{mod}p_2^{\alpha_2})\\ \cdots & \\ \binom{n}{k}\equiv r_s, & (\mathrm{mod}p_s^{\alpha_s})\end{cases}$$

最后，利用中国剩余定理求出模 m 的余数。

简单来说，就是：**Wilson 定理的推广** + **Kammer 定理** + **中国剩余定理**。下面给出 P4720 模板题的代码：

Listing 35: 扩展 Lucas 定理

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 1e5 + 5;

ll read() {
    ll tem;
    scanf("%lld",&tem);
    return tem;
}

int p[N], pa[N], c[N], prime_cnt;
void divide(int n) {
    prime_cnt = 0;
    for(int i = 2; i <= sqrt(n); i++) {
        if(n % i == 0) {
            p[++prime_cnt] = i, pa[prime_cnt] = 1, c[prime_cnt] = 0;
            while(n % i == 0)
                n /= i, pa[prime_cnt] *= i, c[prime_cnt]++;
        }
    }

    if(n > 1)
        p[++prime_cnt] = n, pa[prime_cnt] = n, c[prime_cnt] = 1;
}

// a 代表余数, pa 代表模数
int a[N];
// 求解 (n!)_p % pa
int factmod(ll n, int p, int pa) {
    vector<int> f(pa);
    f[0] = 1;
    for(int i = 1; i < pa; i++)
        f[i] = i % p ? (ll)f[i - 1] * i % pa : f[i - 1];
    bool neg = p != 2 || pa <= 4;
    int res = 1;
    while(n > 1) {
        if((n / pa) & neg)
            res = pa - res;
        res = (ll)res * f[n % pa] % pa;
        n /= p;
    }
    return res;
}

// 求乘法逆元
void exgcd(ll a, ll b, int &x, int &y) {
    if(!b) {
```

```
49          x = 1;
50          y = 0;
51          return ;
52      }
53      exgcd(b, a % b, y, x);
54      y -= a / b * x;
55  }
56
57  // 求 n % p 的逆元
58  int inv(int n, int p) {
59      int x, y;
60      exgcd(n, p, x, y);
61      return (x + p) % p;
62  }
63
64  // 计算 p 进制下 n 的数字和
65  int cnt_p(ll n, int p) {
66      int ret = 0;
67      while(n)
68          ret += n % p, n /= p;
69      return ret;
70  }
71
72  // Kummer 定理求幂次
73  int kummer(ll n, ll m, int p) {
74      return cnt_p(m, p) + cnt_p(n - m, p) - cnt_p(n, p);
75  }
76
77  ll quickPow(ll a, int b, int c) {
78      ll ret = 1;
79      while(b) {
80          if(b & 1)
81              ret = ret * a % c;
82          a = a * a % c;
83          b >>= 1;
84      }
85      return ret;
86  }
87
88  // 计算组合数比上第 i 个模数的余数
89  void calc_a(int i, ll n, ll m) {
90      int num1 = factmod(n, p[i], pa[i]);
91      int num2 = factmod(m, p[i], pa[i]);
92      int num3 = factmod(n - m, p[i], pa[i]);
93      int inv2 = inv(num2, pa[i]), inv3 = inv(num3, pa[i]);
94      int tem = kummer(n, m, p[i]) / (p[i] - 1);
95      if(tem >= c[i])
96          a[i] = 0;
97      else
```

```
        a[i] = quickPow(p[i], tem, pa[i]) * num1 * inv2 * inv3 % pa[i];
}

int intChina(int r) {
    ll Mi, d, ans = 0;
    int x0, y0;
    ll M = 1;
    for(int i = 1; i <= r; i++)
        M *= pa[i];
    for(int i = 1; i <= r; i++) {
        Mi = M / pa[i];
        exgcd(Mi, pa[i], x0, y0);
        ans = (ans + Mi * x0 * a[i]) % M;
    }
    return (ans + M) % M;
}

int main() {
    ll n = read(), m = read(), p = read();
    divide(p);
    for(int i = 1; i <= prime_cnt; i++)
        calc_a(i, n, m);
    printf("%d", intChina(prime_cnt));
    return 0;
}
```

3.3.6 容斥原理

集合的并 = 集合交的交错和

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_{m=1}^{n}(-1)^{m-1} \sum_{a_i<a_{i+1}} \left|\bigcap_{i=1}^{m} S_{a_i}\right|$$

在实际编写代码时，通常需要枚举状态。

3.3.7 Catalan 数

Catalan 数应用的特征：一种操作数不能超过另外一种操作数，或者两种操作不能有交集，这些操作的合法方案数，通常是 Catalan 数（1、1、2、5、14、42、132、429、1430、$\cdots$）。

关于 Catalan 数的常见公式：

$$H_n = \frac{\binom{2n}{n}}{n+1} (n \geq 2, n \in \mathbf{N}_+)$$

$$H_n = \begin{cases} \sum_{i=1}^{n} H_{i-1}H_{n-i} & n \geq 2, n \in \mathrm{N}_+ \\ 1 & n = 0, 1 \end{cases}$$

$$H_n = \frac{H_{n-1}(4n-2)}{n+1}$$

$$H_n = \binom{2n}{n} - \binom{2n}{n-1}$$

下面这些问题的解都是 Catalan 数

1. 有一个大小为 $n \times n$ 的方格图左下角为 (0, 0) 右上角为 (n, n)，从左下角开始每次都只能向右或者向上走一单位，不走到对角线 $y = x$ 上方（但可以触碰）的情况下到达右上角有多少可能的路径？
2. 在圆上选择 2n 个点，将这些点成对连接起来使得所得到的 n 条线段不相交的方法数？
3. 由 n 个 +1 和 n 个 -1 组成的 2n 个数 $a_1, a_2, \cdots, a_{2n}$，其部分和满足 $a_1 + a_2 + \cdots + a_k \geq 0$ $(k = 1, 2, 3, \cdots, 2n)$，有多少个满足条件的数列？
4. 包括 n 组括号的合法运算式的个数有多少？
5. n 个结点可构造多少个不同的二叉树？
6. 通过连接顶点而将 n + 2 边的凸多边形分成 n 个三角形的方法数？
7. 一个栈（无穷大）的进栈序列为 $1, 2, 3, \cdots, n$ 有多少个不同的出栈序列？

路径计数问题

1. 从 $(0, 0)$ 到 (m, n) 的非降路径数：$\binom{n+m}{m}$.
2. 从 $(0, 0)$ 到 (n, n) 的除端点外不接触直线 $y = x$ 的非降路径数：$2\binom{2n-2}{n-1} - 2\binom{2n-2}{n}$.
3. 从 $(0, 0)$ 到 (n, n) 的除端点外不穿过直线 $y = x$ 的非降路径数：$\frac{2}{2n+1}\binom{2n}{n}$.（即 $2H_n$）

Listing 36: 卡特兰数

```
ll cat[25];
void catalan_init(int n) {
    cat[0] = 1;
    for (int i = 1; i <= n; i++)
        cat[i] = cat[i - 1] * (4 * i - 2) / (i + 1);
    return ;
}
```