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); }

```

```
201     void print() const {
202         cout << digits.back(); // 最高位不需要前导零
203 
204         for (int i = digits.size() - 2; i >= 0; i--)
205             cout << setw(BASE_DIGITS) << setfill('0') << digits[i];
206     }
207 
208     string toString() const {
209         string result;
210         result += to_string(digits.back());
211 
212         for (int i = digits.size() - 2; i >= 0; i--) {
213             string segment = to_string(digits[i]);
214             result += string(BASE_DIGITS - segment.size(), '0') + segment;
215         }
216         return result;
217     }
218 
219     friend ostream& operator<<(ostream& os, const Big& num) {
220         os << num.digits.back();
221         for (int i = num.digits.size() - 2; i >= 0; i--)
222             os << setw(BASE_DIGITS) << setfill('0') << num.digits[i];
223         return os;
224     }
225 };
```

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 整除分块

解决诸如求 $\sum_{i=1}^{n} f(i)\lfloor\frac{n}{i}\rfloor$ 一类的问题；首先对于 $\lfloor\frac{n}{i}\rfloor$ 有两条性质：

性质 1：分块的块数 $\leq 2\lfloor\sqrt{n}\rfloor$

性质 2：i 所在块的右端点为 $\lfloor\frac{n}{\lfloor\frac{n}{i}\rfloor}\rfloor$

所以可以先预处理出 $f(i)$ 的前缀和 $s(i)=\sum_{j=1}^{i} f(j)$，再枚举每一个块 $[l,r]$，累加每块的贡献，时间复杂度 $O(\sqrt{n})$。

Listing 13: 整除分块

```
int div_block(int n) {
    int r = 0, ret = 0;
    for(int l = 1; l <= n; l = r + 1) {
        if(n / l == 0)
            break;
        r = n / (n / l);
        ret += (s[r] - s[l - 1]) * (n / l);
    }
    return ret;
}
```

也可以向二维扩展，求 $\sum_{i=1}^{min(n,m)} f(i)\lfloor\frac{n}{i}\rfloor\lfloor\frac{m}{i}\rfloor$，时间复杂度 $O(\sqrt{n}+\sqrt{m})$。

Listing 14: 整数分块二维扩展

```
int div_block(int n, int m) {
    int r = 0, ret = 0;
    int bound = min(n, m);
    for(int l = 1; l <= bound; l = r + 1) {
        if(n / l == 0)
            break;
        // 求两个块的公共区间
        r = min(n / (n / l), m / (m / l));
        ret += (s[r] - s[l - 1]) * (n / l) * (m / l);
    }
    return ret;
}
```

3.1.8 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 15: 欧几里得算法求 gcd

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

3.1.9 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 16: 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)
                A = A - B;
            else
                B = B - A;
        }
    }

    Big result = Big(A == 0 ? B : A);
    return result * (Big(2) ^ shift);
}
```

3.1.10 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。

3.1.11 扩展欧几里得算法

用于求解 $ax + by = \gcd(a, b)$ 的一组特解。

Listing 17: 扩展欧几里得算法

```
// 返回 a b 的最大公约数 d, 并找到一组特解 x0 y0
int exgcd(int a, int b, int &x, int &y) {
    if(b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int d = exgcd(b, a % b, x, y);
    int z = x;
    x = y;
    y = z - y * (a / b);
    return d;
}
```

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。

3.1.12 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 $x_i = x + i \times (m/d),\ 0 \le i \le d - 1$。

3.1.13 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 18: 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;
}
```

3.1.14 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 $k \times i + j \equiv 0 (\mathrm{mod}\ p)$，有 $k \times j^{-1} + i^{-1} \equiv 0 (\mathrm{mod}\ p)$，即 $i^{-1} \equiv -\lfloor \frac{p}{i} \rfloor \times (p \bmod i)^{-1} (\mathrm{mod}\ p)$。

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

Listing 19: 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;
    }
}
```

3.1.15 欧拉函数

$1 \sim \mathrm{N}$ 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，$\mathrm{N} = \mathrm{p}_1^{\mathrm{c}_1} \mathrm{p}_2^{\mathrm{c}_2} \cdots \mathrm{p}_{\mathrm{m}}^{\mathrm{c}_{\mathrm{m}}}$，则

$$\varphi(\mathrm{N}) = \mathrm{N} \times \frac{\mathrm{p}_1 - 1}{\mathrm{p}_1} \times \frac{\mathrm{p}_2 - 1}{\mathrm{p}_2} \times \cdots \times \frac{\mathrm{p}_{\mathrm{m}} - 1}{\mathrm{p}_{\mathrm{m}}} = \mathrm{N} \times \prod_{\text{prime } \mathrm{p} | \mathrm{N}} (1 - \frac{1}{\mathrm{p}})$$

Listing 20: 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。

3.1.16 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 21: 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]);
        }
    }
}
```

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

3.1.18 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 22: 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);
}
```

3.1.19 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 23: 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;
}
```

3.1.20 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 $m = \text{lcm}(m_1, m_2, \cdots, m_{k-1})$，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 24: 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]);
        x = (x % M + M) % M;
    }
    return x;
}
```

3.1.21 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 25: BSGS 求解高次同余方程

```cpp
// Baby Step, Giant Step 算法
// 注意: a 与 p 必须互质
// 无解时返回 -1
int BSGS(int a, int b, int p) {
    map<int, int> hash;
    hash.clear();
    b %= p;
    int t = (int)sqrt(p) + 1;
    for(int j = 0; j < t; j++) {
        int val = (long long)b * pow(a, j, p) % p;
        hash[val] = j;
    }
    a = pow(a, t, p);
    if(a == 0)
        return b == 0 ? 1 : -1;
    for(int i = 0; i <= t; i++) {
        int val = pow(a, i, p);
        int j = hash.find(val) == hash.end() ? -1 : hash[val];
        if(j >= 0 && i * t >= j)
            return i * t - j;
    }
    return -1;
}
```

3.1.22 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 26: 利用推广的 Wilson 定理求 $(n!)_p$

```cpp
// Calculate (n!)_p mod pa.
// 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 27: 利用 Legendre 公式求 $\nu_p(n!)$

```
// 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 28: 矩阵快速幂

```
// 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 29: 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]);
```

```
18
19          for(int i = 0; i < n; i++) {
20              if(i == nwline)
21                  continue;
22              double mul = p[i][k] / p[nwline][k];
23              for(int j = k; j <= n; j++)
24                  p[i][j] -= p[nwline][j] * mul;
25          }
26          nwline++;
27      }
28
29      // 存在找不到主元的情况
30      // 一定是无解或者无穷多解，优先判断无解
31      if(nwline < n) {
32          while(nwline < n)
33              if(!(fabs(p[nwline++][n]) < eps)) {
34                  printf("-1");
35                  return ;
36              }
37          printf("0");
38          return ;
39      }
40
41      for(int i = 0; i < n; i++)
42          printf("%.2lf\n", fabs(p[i][n] / p[i][i]) < eps ? 0 : p[i][n] / p[i][i]);
43  }
```

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 30: 高斯消元法

```
1   // 时间复杂度 O(n^2m / w)，w 是利用压位进行优化
2
3   void gaussElimination(int n) {
4       int ans = 1;
5       for(int i = 1; i <= n; i++) {
6           for(int j = i + 1; j <= n; j++)
7               if(a[j] > a[i])
8                   swap(a[i], a[j]);
9
```

```
10          // 消元完毕，有 i - 1 个主元，n - i + 1 个自由元
11          if(a[i] == 0) {
12              ans = 1 << (n - i + 1);
13              break;
14          }
15
16          // 出现 0 = 1，无解
17          if(a[i] == 1) {
18              ans = 0;
19              break;
20          }
21
22          for(int k = n; k; k--)
23              if(a[i] >> k & 1) {
24                  for(int j = 1; j <= n; j++)
25                      if(i != j && (a[j] >> k & 1))
26                          a[j] ^= a[i];
27
28                  break;
29              }
30      }
31
32      if(ans == 0)
33          printf("No Solution!");
34      else if(ans != 1)
35          printf("%d", ans);
36      else {
37          for(int i = 1; i <= n; i++)
38              printf("%d : %d\n", i, a[i] & 1);
39      }
40  }
```

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 31: 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 32: 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 33: 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 34: 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 35: 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 $S = \{n_1 a_1, n_2 a_2, \cdots, n_k a_k\}$，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 $S = \{n_1 a_1, n_2 a_2, \cdots, n_k a_k\}$，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 $S = \{n_1 a_1, n_2 a_2, \cdots, n_k a_k\}$，设 $r \le n$。则从 S 中取出一个 r 个元素的多重集的组合数（证明需要容斥原理）

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

注意当 $n = 1$ 时需要特判，上面的公式并不适用。多重集组合数的代码可以看容斥原理一节。

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 36: Lucas 定理

```
// p 一定要是质数
// 为了求解组合数进行预处理
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 37: 扩展 Lucas 定理

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

```
5
6  ll read() {
7      ll tem;
8      scanf("%lld",&tem);
9      return tem;
10 }
11
12 int p[N], pa[N], c[N], prime_cnt;
13 void divide(int n) {
14     prime_cnt = 0;
15     for(int i = 2; i <= sqrt(n); i++) {
16         if(n % i == 0) {
17             p[++prime_cnt] = i, pa[prime_cnt] = 1, c[prime_cnt] = 0;
18             while(n % i == 0)
19                 n /= i, pa[prime_cnt] *= i, c[prime_cnt]++;
20         }
21     }
22
23     if(n > 1)
24         p[++prime_cnt] = n, pa[prime_cnt] = n, c[prime_cnt] = 1;
25 }
26
27 // a 代表余数, pa 代表模数
28 int a[N];
29 // 求解 (n!)_p % pa
30 int factmod(ll n, int p, int pa) {
31     vector<int> f(pa);
32     f[0] = 1;
33     for(int i = 1; i < pa; i++)
34         f[i] = i % p ? (ll)f[i - 1] * i % pa : f[i - 1];
35     bool neg = p != 2 || pa <= 4;
36     int res = 1;
37     while(n > 1) {
38         if((n / pa) & neg)
39             res = pa - res;
40         res = (ll)res * f[n % pa] % pa;
41         n /= p;
42     }
43     return res;
44 }
45
46 // 求乘法逆元
47 void exgcd(ll a, ll b, int &x, int &y) {
48     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
98          a[i] = quickPow(p[i], tem, pa[i]) * num1 * inv2 * inv3 % pa[i];
99  }
100
101 int intChina(int r) {
102     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。以求解多重集的组合数为例，给出模板题的代码：

Listing 38: 容斥原理示例 - 多重集组合数

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 1e5 + 5;
const int MOD = 1e9 + 7;

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

ll f[25];

int popcount(int i) {
    int ret = 0;
    while(i) {
        if(i & 1)
            ret++;
```

```
        i >>= 1;
    }
    return ret;
}

ll inv[25], fac_inv[25];
void init(int n) {
    inv[0] = fac_inv[0] = 1;
    inv[1] = fac_inv[1] = 1;
    int p = MOD;
    for(int i = 2; i <= n; i++) {
        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;
    ll tem = 1;
    for(int i = 0; i < m; i++)
        tem = tem * (n - i) % MOD;
    tem = tem * fac_inv[m] % MOD;
    return tem;
}

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

int main() {
    ll n = read(), s = read();
    init(n);
    for(int i = 1; i <= n; i++)
        f[i] = read();
    if(n == 1)
        return printf("%lld", f[1] >= s ? 1 : 0), 0;
    ll ans = 0;
    for(int i = 0; i < (1 << n); i++) {
        int tem = popcount(i);
        int mi = tem % 2 ? -1 : 1;
        int r = i, cnt = 0;
        ll top = n + s - 1 - tem;
        while(r) {
            cnt++;
            if(r & 1)
```

```
                top -= f[cnt];
            r >>= 1;
        }
        ans = (ans + mi * lucas(top, n - 1)) % MOD;
    }
    ans = (ans + MOD) % MOD;
    printf("%lld", ans);
    return 0;
}
```

3.3.7 Mobius 函数

$$\mu(N) = \begin{cases} 0 & \exists i \in [1,m], c_i > 1 \\ 1 & m \equiv 0(\mathrm{mod}2), \forall i \in [1,m], c_i = 1 \\ -1 & m \equiv 1(\mathrm{mod}2), \forall i \in [1,m], c_i = 1 \end{cases}$$

若只求一项的 Mobius 函数，可以利用分解质因数计算；如果想要求 $1 \sim N$ 每一项的 Mobius 函数，可以利用埃氏筛计算：

Listing 39: 利用埃氏筛计算 $1 \sim N$ 的 Mobius 函数

```
int miu[N], v[N];
void Mobius_func(int n) {
    for(int i = 1; i <= n; i++)
        miu[i] = 1, v[i] = 0;
    for(int i = 2; i <= n; i++) {
        if(v[i])
            continue;
        miu[i] = -1;
        for(int j = 2 * i; j <= n; j += i) {
            v[j] = 1;
            if((j / i) % i == 0)
                miu[j] = 0;
            else
                miu[j] *= -1;
        }
    }
}
```

Mobius 函数的一个简单应用是可以用来求满足 $1 \le \mathrm{x} \le \mathrm{a}$、$1 \le \mathrm{y} \le \mathrm{b}$ 的且互质的 (x,y) 的对数。假设用 $\mathrm{F}[\mathrm{a},\mathrm{b}]$ 表示满足 $1 \le \mathrm{x} \le \mathrm{a}$、$1 \le \mathrm{y} \le \mathrm{b}$ 的且互质的 (x,y) 的对数；用 $\mathrm{D}[\mathrm{a},\mathrm{b},\mathrm{k}]$ 表示满足 $1 \le \mathrm{x} \le \mathrm{a}$、$1 \le \mathrm{y} \le \mathrm{b}$ 且 $k \mid gcd(x,y)$ 的 (x,y) 的对数。那么有：

$$F[a,b] = \sum_{i=1}^{min(a,b)} \mu(i) \times D[a,b,i]$$

3.3.8 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 40: 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 ;
}
```

3.4 博弈论

3.4.1 Nim 游戏

地上有 n 堆石子，每人每次可从任意一堆石子里取出任意多枚石子扔掉，可以取完，不能不取。每次只能从一堆里取。最后没石子可取的人就输了。问是否存在先手必胜的策略。

Nim 博弈先手必胜，当且仅当 $\mathrm{A_1}$ xor $\mathrm{A_2}$ xor$\cdots$ xor $\mathrm{A_n} \neq 0$。

3.4.2 公平组合游戏 ICG

若一个游戏满足：

- 由两名玩家交替行动
- 在游戏进程的任意时刻，可以执行的合法行动与轮到哪名玩家无关
- 不能行动的玩家判负

则称该游戏是一个公平组合游戏。Nim 游戏是公平组合游戏，但常见的棋类游戏，比如围棋，就不是公平组合游戏，因为不满足条件 2 和 3。

3.4.3 有向图游戏

给定一个 DAG，图中有一个唯一的起点，在起点上放有一个棋子。两名玩家交替把棋子沿有向边进行移动，每次可以移动一步，无法移动者判负。该游戏被称为有向图游戏。任何一个 ICG 都能转化为有向图游戏。

3.4.4 SG 函数

在有向图游戏中，对于每个节点 x，设从 x 出发共有 k 条有向边，分别到达结点 $y_1, y_2, \cdots, y_k$，定义 SG(x) 为 x 的后继结点的 SG 函数值构成的集合再执行 mex 运算，即

$$SG(x) = mex(SG(y_1), SG(y_2), \cdots, SG(y_k))$$

特别地，整个有向图游戏 G 的 SG 函数值被定义为起点 s 的 SG 函数值，即 SG(G) = SG(s)。

3.4.5 有向图游戏的和

设 $G_1, G_2, \cdots, G_m$ 是 m 个有向图游戏。定义有向图游戏 G，它的行动规则是任选某个有向图游戏 G_i，并在 G_i 上行动一步，则 G 称为有向图游戏 $G_1, G_2, \cdots, G_m$ 的和，它的 SG 函数为：

$$SG(G) = mex(SG(G_1), SG(G_2), \cdots, SG(G_m))$$

有向图游戏的某个局面必胜，当且仅当该局面对应结点的 SG 函数值大于 0。
有向图游戏的某个局面必败，当且仅当该局面对应结点的 SG 函数值等于 0。

3.4.6 操作出某个状态取胜的博弈游戏

如果题目中给出获胜条件是操作出某个状态，那么我们就不能直接应用有向图游戏，因为有向图游戏要求“不能操作者失败”。我们可以进行一个转换，往前推直到推出必败态作为有向图游戏的终点。然后就可以继续利用 SG 理论了。同时，这个必败态一般还不会直接是获胜条件，而是沿着这个状态走下去，一定能导致对方获胜的状态；也就是说，走下去一定失败，但是实际上并没有走下去。

4 字符串

4.1 KMP

给定一个模式串 P 和主串 S，求模式串在主串中出现的位置（字符串下标均从 1 开始）：

1. 用双指针表示 $S[i-j+1\cdots i]$ 与 $P[1\cdots j]$ 完全相等。i 是不断增加的，j 相应地变化，且始终满足前面的关系
2. 当 $S[i+1]=P[j+1]$ 时，i 与 j 各加 1
3. 当 $j=m$ 时，发现 P 是 S 的子串，并输出相应位置
4. 当 $S[i+1]\neq P[j+1]$ 时，减小 j 值，使得新的 j 值仍使关系得到满足。调整 j 值到多少呢？不难发现可以维护一个数组 next[i]，用来存储模式串 $P[1\cdots i]$ 中相等真前后缀的最长长度

Listing 41: KMP

```
// KMP（双指针）

// nxt 数组，以及字符串都从 1 开始
// 求 next 数组
void pre() {
    nxt[1] = 0;
    int j = 0;
    for(int i = 1; i < m ;i++) {
        while(j > 0 && P[j + 1] != P[i + 1])
            j = nxt[j];
        if(P[j + 1] == P[i + 1])
            j++;
        nxt[i + 1] = j;
    }
}

// S 是主串，P 是模式串
void kmp() {
    int j = 0;
    for(int i = 0; i < n ;i++) {
        while(j > 0 && P[j + 1] != S[i + 1])
            j = nxt[j];
        if(P[j + 1] == S[i + 1])
            j++;
        if(j == m)
            printf("%d␣", i + 1 - m + 1), j = nxt[j];
    }
}
```

4.1.1 Power Strings 问题

求一个字符串由多少个重复的子串连接而成。

如果 $n \bmod (n-\text{next}[n])=0$，那么答案就是 $n/(n-\text{next}[n])$，否则答案就是 1。

4.1.2 非严格 Power Strings

比如 abcabcab 的最小循环串是 abc，长度为 3
最小循环串长度：n − next[n]

4.2 字符串哈希

对字符串 $C = c_1c_2\cdots c_m$，定义 Hash 函数

$$H(c) = (c_1b^{m-1} + c_2b^{m-2} + \cdots + c_mb^0) \bmod h$$

设 H(c, k) 为对前 k 个字符串求 Hash 值，则有

$$H(c, k+1) = b \times H(c, k) + c_{k+1}$$

求子串的 Hash 值,对于字符串 $C = c_1c_2\cdots c_m$ 从位置 k + 1 开始的长度为 n 的子串 $C' = c_{k+1}c_{k+2}\cdots c_{k+n}$，有

$$H(c') = H(c, k+n) - H(c, k) \times b^n$$

哈希冲突的问题

1. 一般可以取 $h = 2^{32}$ 或 $h = 2^{64}$，利用 unsigned int 或者 unsigned long long 自然溢出计算 Hash 值
2. 为了保险起见，可以采取双哈希策略
3. b 常定义为 131 或 13331

Listing 42: 字符串哈希

```
typedef unsigned long long ull;
const int P = 131;
// 字符串和 h 数组都从 1 开始
ull p[maxn], h[maxn];

// 预处理 hash 函数的前缀和
void init() {
    p[0] = 1, h[0] = 0;
    for(int i = 1; i <= n; i++) {
        p[i] = p[i - 1] * P;
        h[i] = h[i - 1] * P + s[i];
    }
}

// 计算 s[l ~ r] 的 hash 值
ull get(int l, int r) {
    return h[r] - h[l - 1] * p[r - l + 1];
}
```

4.3 Trie

树的每条边恰好对应一个字符，每条根到顶点的路径对应了一个字符串。利用了串的公共前缀，节约了存储空间。

基本操作：初始化、插入、删除

Listing 43: Trie

```
// 初始化
int trie[maxn][26], tot = 1;

// 插入字符串
void insert(char *str) {
    int len = strlen(str), p = 1;
    for(int k = 0; k < len; k++) {
        int ch = str[k] - 'a';
        if(trie[p][ch] == 0)
            trie[p][ch] = ++tot;
        p = trie[p][ch];
    }
    end[p] = true;
}

// 检索字符串
bool search(char *str) {
    int len = strlen(str), p = 1;
    for(int k = 0; k < len; k++) {
        p = trie[p][str[k] - 'a'];
        if(p == 0)
            return false;
    }
    return end[p];
}
```

4.3.1 最大异或对问题

从 n 个数中挑两个，让他们的异或和最大。

每一步都尝试沿着与当前位相反的字符指针向下访问（如果为空节点就只能访问相同的）。

4.4 AC 自动机

与 KMP 算法类似，AC 自动机也用来处理字符串匹配问题。不过 KMP 用来处理单模式串的问题，而 AC 自动机用来处理多模式串的问题。AC 自动机建立在 KMP 算法和 Trie 树的基础上。

构建一个 AC 自动机用于模式匹配需要三步：

1. 用所有模式串构建 Trie 树
2. 在 Trie 上建立 **回跳边**和 **转移边**两类边（Trie 树中的树边算一种特殊的转移边，不需要经过回跳转移）

3. 扫描主串进行匹配

建 Trie 没什么好说的，主要看第二步里的建立回跳边和转移边。

回跳边的作用是，在匹配失败时的跳转路径，回跳边的所指结点就是当前结点的最长后缀。回跳边的构建方式是：**回跳边指向父节点的回跳边所指结点的儿子**（根节点、第一层结点的回跳边都指向根节点）。

转移边表示匹配过程中的跳转路径。转移边所指结点就是当前结点的最短路（即距离当前结点最近的还能继续匹配新字符串的地方）。转移边的构建方式是：**转移边指向当前结点的回跳边所指结点的儿子**。

综上，利用 BFS 建立 AC 自动机的过程就是：

1. 初始化，把根节点的儿子们入队
2. 只要队不空，节点 u 出队
3. 枚举 u 的 26 个儿子，若儿子存在（有树边），则爹帮儿子建立回跳边，并把儿子入队；若儿子不存在，则爹自建转移边。

利用 AC 自动机查找单词的出现次数，需要扫描主串，依次取出字符 s[k]。一边走串，一边把当前串的所有后缀串搜索出来。

1. i 指针走主串对应的结点，沿着树边或转移边走，保证不回退。
2. j 指针沿着回跳边搜索模式串，每次从当前结点走到根节点，把当前结点中的所有后缀模式串一网打尽，保证不遗漏。
3. 扫描完主串，返回答案。

Listing 44: AC 自动机

```
// 求有多少个模式串在主串里出现过
int trie[maxn][26], tot = 0, cnt[maxn], nxt[maxn];

// 插入字符串，用来建立初始字典树
void insert(char *str) {
    int len = strlen(str), p = 0;
    for(int k = 0; k < len; k++) {
        int ch = str[k] - 'a';
        if(trie[p][ch] == 0)
            trie[p][ch] = ++tot;
        p = trie[p][ch];
    }
    cnt[p]++;
}

// 建立 AC 自动机
// 时间复杂度 O(26 |S|)
void build() {
    queue<int> q;
    for(int i = 0; i < 26; i++)
```

```
        if(trie[0][i])
            q.push(trie[0][i]);

    while(q.size()) {
        int u = q.front();
        q.pop();
        for(int i = 0; i < 26; i++) {
            int v = trie[u][i];
            if(v)
                nxt[v] = trie[nxt[u]][i], q.push(v);
            else
                trie[u][i] = trie[nxt[u]][i];
        }
    }
}

// 查找有多少模式串在主串里出现过
// 时间复杂度 O(n)
int query(char *s) {
    int ans = 0;
    for(int k = 0, i = 0; s[k]; k++) {
        i = trie[i][s[k] - 'a'];
        for(int j = i; j && ~cnt[j]; j = nxt[j])
            ans += cnt[j], cnt[j] = -1;
    }
    return ans;
}
```

4.4.1 AC 自动机的拓扑排序优化

在我们上面的实现里，每次匹配，会一直向回跳边走来找到所有的匹配，但是这样的效率其实很低。

我们可以利用回跳边的一个性质：在一个 AC 自动机中，如果只保留前向边，那么剩下的图一定是一棵树（每个节点有唯一前驱，也就是前向边指向的结点）。这样 AC 自动机的匹配就可以转化为前向树上的链求和问题，只需要优化这一部分就可以了。

时间主要浪费在每次都要跳前向边，如果我们可以预先记录，最后一并求和，那么效率就会优化。于是我们可以在前向树上，做一次拓扑排序，一次性求出每个模式串的出现次数。具体地说，在建立 AC 自动机的过程中，求每个节点在前向树中的入度（注意前向树中的边由孩子指向父亲）；然后在查询的时候，就可以只为被找到的结点的 vis 加上 1；最后再利用拓扑排序，结合 vis 数组求出答案。

Listing 45: AC 自动机的拓扑排序优化

```
// idx 记录了每个树上点对应的字符串编号
// 这种出现几次的问题不需要 cnt
// 因为某一串可能在另一串的中间出现（如 aa 和 baaaa）
int trie[maxn][26], tot = 0, idx[maxn], nxt[maxn];

// 插入字符串
```

```
7  void insert(char *str, int i) {
8      int len = strlen(str), p = 0;
9      for(int k = 0; k < len; k++) {
10         int ch = str[k] - 'a';
11         if(trie[p][ch] == 0)
12             trie[p][ch] = ++tot;
13         p = trie[p][ch];
14     }
15     idx[p] = i;
16 }
17
18 // 维护每个树上结点的入度
19 int degree[maxn];
20 // 建立 AC 自动机
21 void build() {
22     queue<int> q;
23     for(int i = 0; i < 26; i++)
24         if(trie[0][i])
25             q.push(trie[0][i]);
26
27     while(q.size()) {
28         int u = q.front();
29         q.pop();
30         for(int i = 0; i < 26; i++) {
31             int v = trie[u][i];
32             if(v) {
33                 nxt[v] = trie[nxt[u]][i];
34                 degree[trie[nxt[u]][i]]++;
35                 q.push(v);
36             }
37             else
38                 trie[u][i] = trie[nxt[u]][i];
39         }
40     }
41 }
42
43 // 统计每个结点的直接出现次数
44 int vis[maxn];
45 void query(char *s) {
46     for(int k = 0, i = 0; s[k]; k++) {
47         i = trie[i][s[k] - 'a'];
48         vis[i]++;
49     }
50 }
51
52 // 统计最后每个单词的出现次数
53 int ans[maxn];
54 void topo() {
55     queue<int> q;
```

```
    for(int i = 0; i <= tot; i++)
        if(degree[i] == 0)
            q.push(i);

    while(!q.empty()) {
        int u = q.front();
        q.pop();
        ans[idx[u]] = vis[u];
        int v = nxt[u];
        vis[v] += vis[u];
        degree[v]--;
        if(degree[v] == 0)
            q.push(v);
    }
}
```

4.4.2 最短母串问题

找一个字符串，包含所有模式串，同时字典序最小，长度最短。

在 AC 自动机的 Trie 树里，每个节点表示一个字符串，但是这个字符串与根如何走到它无关，仅仅与它在树中的位置有关。如果我们考虑根经过何种路径最后走到某节点，那么每往下扩展一层，就相当于向后面增加一个字符。就相当于利用广搜，直到遇到第一个包含了所有模式串的字符串为止（在往下扩展时，按照字典序枚举字符，保证了最后字符串是字典序最小的）。那么如何判断某个字符串是否包含所有模式串呢？可以利用状压的思想，给每个节点增加一个状态位，表示仅仅考虑位置而言这个结点包含了哪几个字符串，bfs 时每扩展一个新的节点就或上这个状态值。

Listing 46: 最短母串问题

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 605;
const int maxm = (1 << 12) + 5;

int read() {
    int tem;
    scanf("%d",&tem);
    return tem;
}

// sta 用于状压
int trie[maxn][26], tot, sta[maxn], nxt[maxn];

// 给每个节点打上初始标记
void insert(char *str, int i) {
    int len = strlen(str), p = 0;
    for(int k = 0; k < len; k++) {
        int ch = str[k] - 'A';
        if(trie[p][ch] == 0)
```

```
22              trie[p][ch] = ++tot;
23          p = trie[p][ch];
24      }
25      sta[p] |= 1 << i;
26  }
27
28  // build 之后，每个节点上的 sta 都被维护好了
29  void build() {
30      queue<int> q;
31      for(int i = 0; i < 26; i++)
32          if(trie[0][i])
33              q.push(trie[0][i]);
34
35      // 因为 bfs 是按层进行遍历的
36      // 所以维护该层的时候前几层的 sta 值一定维护完了
37      while(q.size()) {
38          int u = q.front();
39          q.pop();
40          for(int i = 0; i < 26; i++) {
41              int v = trie[u][i];
42              if(v) {
43                  nxt[v] = trie[nxt[u]][i];
44                  sta[v] |= sta[trie[nxt[u]][i]];
45                  q.push(v);
46              } else
47                  trie[u][i] = trie[nxt[u]][i];
48          }
49      }
50  }
51
52  bool vis[maxn][maxm];
53  int n, pre[maxn * maxm];
54  char str[15][55], w[maxn * maxm];
55  // 这里利用了一个巧妙的思想，同步父亲编号和孩子编号
56  void bfs() {
57      queue<int> qnode, qsta;
58      int cnt = 0, cnt_vis = 0;
59      qnode.push(0), qsta.push(0);
60      vis[0][0] = 1;
61      while(qnode.size()) {
62          int node = qnode.front();
63          int st = qsta.front();
64          qnode.pop(), qsta.pop();
65
66          // 这个就是答案
67          if(st == (1 << n) - 1) {
68              vector<int> ans;
69              while (cnt_vis) {
70                  ans.push_back(w[cnt_vis]);
```

```
                cnt_vis = pre[cnt_vis];
            }
            reverse(ans.begin(), ans.end());
            for(auto it : ans)
                putchar(it);
            return ;
        }

        // 依次遍历孩子，进行转移
        for(int i = 0; i < 26; i++) {
            if(!vis[trie[node][i]][st | sta[trie[node][i]]]) {
                vis[trie[node][i]][st | sta[trie[node][i]]] = true;
                qnode.push(trie[node][i]);
                qsta.push(st | sta[trie[node][i]]);

                cnt++;
                w[cnt] = i + 'A';
                pre[cnt] = cnt_vis;
            }
        }

        cnt_vis++;
    }
}

int main() {
    n = read();
    for(int i = 0; i < n; i++) {
        scanf("%s", str[i]);
        insert(str[i], i);
    }
    build();
    bfs();
    return 0;
}
```

4.4.3 AC 自动机上 dp 问题

在解决这类问题时，一般的思路就是状态有两个维度，分别表示当前的步数（主串的长度）和当前的顶点。价值通过回跳边传递（即标记下传，比如每遇到一个模式串就加一分，或者问有多少个方案能够包含至少一个模式串之类的最优化 dp 与计数 dp 问题）；状态通过树边和转移边进行转移，因为每次部署 + 1（向下再走一层，再往下扩展一个字符），所以可以用 bfs 来实现转移。

在进行最优化 dp 时，别忘了对初始状态进行合理的赋值。

4.5 最小表示法

给定一个字符串 S[1 ∼ n]，可以不断地把最后一个字符放到开头，最终将得到 n 个字符串，其中字典序最小的一个称为 S 的最小表示。

1. 可先将 S 复制一份在它的结尾，得到的字符串为 SS，有 $B[i] = SS[i \sim i+n-1]$。

2. 在比较 B[i] 与 B[j] 的过程中，如果在 $i+k$ 与 $j+k$ 处发现不相等，假设 $SS[i+k] > SS[j+k]$，那么显然 B[i] 不是 S 的最小表示；此外还可以知道 $B[i+1], B[i+2] \cdots B[i+k]$ 也不是 S 的最小表示。

3. 同理，如果 $SS[i+k] < SS[j+k]$，那么 $B[j], B[j+1] \cdots B[j+k]$ 都不是 S 的最小表示。

Listing 47: 最小表示法

```
// 字符串从 1 开始
int n = strlen(s + 1);
for(int i = 1; i <= n; i++)
    s[n + i] = s[i];
int i = 1, j = 2, k;
while(i <= n && j <= n) {
    for(k = 0; k < n && s[i + k] == s[j + k]; k++);
    if(k == n)
        break;
    if(s[i + k] > s[j + k]) {
        i = i + k + 1;
        if(i == j) i++;
    } else {
        j = j + k + 1;
        if(i == j) j++;
    }
}
int ans = min(i, j);
```

4.6 扩展 KMP

对一个长度为 n 的字符串 S，用 z[i] 表示 S 与其后缀 S[i, n] 的最长公共前缀（LCP）的长度。暴力计算时间复杂度 $O(n^2)$。

对于 i，我们称区间 $[i, i+z[i]-1]$ 是 i 的匹配段，叫做 z-box。算法过程中我们维护右端点最靠右的 z-box，记作 [l, r]。那么 S[l, r] 是 S 前缀，在计算 z[i] 时要保证 $l \le i$。利用盒子，借助之前的状态来加速计算新的状态，即由 $z[1], \cdots, z[i-1]$ 快速计算 z[i]。

算法流程

计算完前 $i-1$ 个 z 函数，维护盒子 [l, r]，则 $s[l, r] = s[1, r-l+1]$

1. 如果 $i \le r$（在盒内），则有 $s[i, r] = s[i-l+1, r-l+1]$

   (a) 若 $z[i-l+1] < r-i+1$，则 $z[i] = z[i-l+1]$

   (b) 若 $z[i-l+1] \ge r-i+1$，则令 $z[i] = r-i+1$，从 $r+1$ 往后暴力枚举

2. 如果 $i > r$（在盒外），则从 i 开始暴力枚举

3. 求出 z[i] 后，如果 $i+z[i]-1 > r$，则更新盒子 $l = i, r = i+z[i]-1$

Listing 48: 扩展 KMP

```
// 字符串从 1 开始
void get_z(char *s, int n) {
    z[1] = n;
    for(int i = 2, l, r = 0; i <= n; i++) {
        if(i <= r)
            z[i] = min(z[i - l + 1], r - i + 1);
        while(s[1 + z[i]] == s[i + z[i]])
            z[i]++;
        if(i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
}
```

还有对于求两个不同字符串使用的扩展 KMP：

Listing 49: 扩展 KMP

```
// 求 S 与 T 的每一个后缀 LCP 的长度数组 p（对于 T 而言）
void get_p(char *s, int n, char *t, int m) {
    for(int i = 1, l, r = 0; i <= m; i++) {
        if(i <= r)
            p[i] = min(z[i - l + 1], r - i + 1);
        while(1 + p[i] <= n && i + p[i] <= m && s[1 + p[i]] == t[i + p[i]])
            p[i]++;
        if(i + p[i] - 1 > r)
            l = i, r = i + p[i] - 1;
    }
}
```

4.7 Manacher

可以在 O(n) 的时间求出一个字符串中的**最长回文串**。思路与扩展 KMP 类似。

首先需要改造字符串，在字符之间和串两端插入 #，改造后，原来的回文串都变成奇回文串，方便统一处理。同时设置 s[0] = $ 作为哨兵结点。

5 动态规划

5.1 背包

5.1.1 0-1 背包

Listing 50: 0-1 背包滚动优化写法

```
for(int i = 1; i <= n ; i++)
    for(int j = m; j >= v[i]; j--)
        f[j] = max(f[j], f[j - v[i]] + w[i]);
```

5.1.2 完全背包

Listing 51: 完全背包

```
for(int i = 1; i <= n; i++)
    for(int j = v[i]; j <= m; j++)
        f[j] = max(f[j], f[j - v[i]] + w[i]);
```

5.1.3 多重背包

最朴素的想法：都拆成一个物体，转化成 01 背包：

Listing 52: 多重背包朴素写法

```
for(int i = 1; i <= n; i++)
    for(int j = m; j >= 0; j--)
        for(int k = 1; k <= s[i] && k * v[i] <= j; k++)
            f[j] = max(f[j], f[j - k * v[i]] + k * w[i]);
```

二进制优化：

Listing 53: 多重背包二进制优化

```
for(int i = 1; i <= n; i++) {
    for(int k = 1; k <= s[i]; k *= 2)
        s[i] -= k, goods.push_back({v[i] * k, w[i] * k});
    if(s[i] > 0)
        goods.push_back({v * s[i], w * s[i]});
}

// 然后当成 01 背包
for(auto good : goods)
    for(int j = m; j >= good.v ; j--)
        f[j] = max(f[j], f[j - good.v] + good.w);
```

单调队列优化：

f 数组是按照类更新的，可以把 f[1 · · · m] 按体积 v 的余数拆分成 v 个类：

- $f[0], f[v], f[2v], \cdots, f[kv]$

- $f[1], f[1+v], f[1+2v], \cdots, f[1+kv]$
- $f[2], f[2+v], f[2+2v], \cdots, f[2+kv]$
- $\cdots$
- $f[j], f[j+v], f[j+2v], \cdots, f[j+kv]$

f[j] 是由前面不超过 s 个的同类值递推得到的，相当于从前面宽度为 s 的窗口中挑选最大值来更新当前值，所以选择用单调队列维护窗口最大值。

但是这还需要顺序更新 f 值，可以增加一个备份数组 g。

Listing 54: 多重背包单调队列优化

```
for(int i = 1; i <= n; i++) {
    memcpy(g, f, sizeof(f));
    // 拆分为 v 类
    for(int j = 0; j < v[i]; j++) {
        int h = 0, t = -1;
        // 对每一类用单调队列
        for(int k = j; k <= m; k += v[i]) {
            // q[h] 不在窗口 [k - s * v, k - v] 中，队头出队
            if(h <= t && q[h] < k - s * v[i])
                h++;
            // 使用队头最大值更新 f
            if(h <= t)
                f[k] = max(g[k], g[q[h]] + (k - q[h]) / v[i] * w[i]);
            // 当前值比队尾更有价值，队尾出队
            while(h <= t && g[k] >= g[q[t]] + (k - q[t]) / v[i] * w[i])
                t--;
            // 下标入队，便于队头出队
            q[++t] = k;
        }
    }
}
```

5.1.4 混合背包

01 背包 + 完全背包 + 多重背包。先把多重背包，进行二进制优化，转化为 01 背包：

Listing 55: 混合背包

```
for(auto thing : things) {
    if(01 背包)
        for(int j = m; j >= thing.v; j--)
            f[j] = max(f[j], f[j - thing.v] + thing.w);
    else if(完全背包)
        for(int j = thing.v; j <= m; j++)
            f[j] = max(f[j], f[j - thing.v] + thing.w);
}
```

5.1.5 分组背包

一个组里最多选一个物品。

Listing 56: 分组背包

```
for(int i = 1; i <= n; i++)
    for(int j = m; j >= 0; j--)
        for(int k = 0; k < s[i]; k++)
            if(j >= v[i][k])
                f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);
```

5.1.6 二维费用背包

Listing 57: 二维费用背包

```
for(int i = 1; i <= n; i++)
    for(int j = v; j >= a[i]; j--)
        for(int k = m; k >= b[i]; k--)
            f[j][k] = max(f[j][k], f[j - a[i]][k - b[i]] + c[i]);
```

5.1.7 有依赖的背包

Listing 58: 有依赖的背包

```
void dfs(int u) {
    for(int i = h[u]; ~i ; i = ne[i]) {
        int son = e[i];
        dfs(son);
        for(int j = m - v[u]; j >= 0 ;j--)
            for(int k = 0; k <= j; k++)
                f[u][j] = max(f[u][j], f[u][j - k] + f[son][k]);
    }

    for(int i = m; i >= v[u] ;i--)
        f[u][i] = f[u][i - v[u]] + w[u];
    for(int i = 0; i < v[u] ;i++)
        f[u][i] = 0;

    return ;
}
```

5.1.8 背包问题求方案数

g 数组记录达到容量 i 对应的最大价值的方案数，f 数组维护达到容量 i（注意，这里与之前不同，应该是刚好达到）对应的最大价值，f 数组初始化为 $-\infty$。时间复杂度 O(logV)。

Listing 59: 背包问题求方案数

```
g[0] = 1;
for(int i = 1; i <= m; i++)
```

```
    f[i] = -INF;

for(int i = 1; i <= n; i++)
    for(int j = m; j >= v[i]; j--) {
        int t = max(f[j], f[j - v[i]] + w[i]);
        int s = 0;
        // 不选第 i 个物品可以满足
        if(t == f[j])   s += g[j];
        // 选第 i 个物品可以满足
        if(t == f[j - v[i]] + w[i]) s += g[j - v];
        if(s >= mod)    s -= mod;
        f[j] = t;
        g[j] = s;
    }
```

5.1.9 背包问题求具体方案

Listing 60: 背包问题求具体方案

```
for(int i = n; i >= 1 ;i--)
    for(int j = 0; j <= m ;j++) {
        f[i][j] = f[i + 1][j];
        if(j >= v[i])
            f[i][j] = max(f[i][j], f[i + 1][j - v[i]] + w[i]);
    }

int vol = m;
// 这样就保证了字典序最小
for(int i = 1; i <= n ;i++)
    if(f[i][vol] == f[i + 1][vol - v[i]] + w[i]) {
        cout << i << "␣";
        vol -= v[i];
    }
```

5.2 树形 dp

使用二次扫描与换根法，在一类树上问题中，需要我们以每个节点为根统计一些信息。如果我们暴力枚举每个点为根，假设统计复杂度是 O(P) 的，那么总时间复杂度会达到 O(NP) 的级别。这种问题我们一般通过两次扫描、换根的方式来优化复杂度：

- 第一次扫描，任选一个结点为根进行树形 DP，此时我们回溯时自底向上统计信息到根。
- 第二次扫描，从刚才选择的根出发，进行一次 DFS。这时需要推算出把根从当前结点换到儿子结点造成的影响，也就是转移方法。在进行递归前按照方程自顶向下把状态转移到儿子结点。

5.3 环形结构上的 dp

一般的思路是**先思考链式问题**，再看如何从环式问题转换为链式问题。

- 可以枚举分开的位置，选择一个位置把环断开，将环转化为链。一般需要执行两次 DP，第一次在任意位置把环断开成链，按照线性问题求解；第二次通过适当的条件和赋值，保证计算出的状态等价于把断开的位置强制相连。
- 可以将链延长两倍，最后取 $\mathrm{dp}[1,\mathrm{n}],\mathrm{dp}[2,\mathrm{n}+1],\cdots,\mathrm{dp}[\mathrm{n}-1,2\mathrm{n}-2]$ 中的最优解（如果与头尾相关的话再加上 $\mathrm{dp}[\mathrm{n},2\mathrm{n}-1]$）。

5.4 单调队列优化 dp

本质就是借助单调性，即时排除不可能的决策。具有以下形式状态转移方程的 dp 通常可以利用单调队列进行优化：

$$F[i]=\min_{L(i)\le j\le R(i)}\{F[j]+val(i,j)\}$$

其中 $L(i)$ 和 $R(i)$ 是关于变量 i 的一次函数，限制了决策 j 的取值范围。通常将 $val(i,j)$ 拆成两部分，一部分仅与 i 有关，另一部分仅与 j 有关。一般分为以下几步：

- 加入所需元素：向单调队列重复加入元素直到当前元素达到所求区间的右边界，这样就能保证所需元素都在单调队列中。在加入的过程中别忘了维护单调性（即加入新元素时，可以让队尾出队）。
- 弹出越界元素：我们弹出在左边界外的元素，以保证单调队列中的元素都在所求区间中。
- 获取最值：直接取队首作为答案即可。

在上面的模型中，$val(i,j)$ 的每一项仅与 i 和 j 的其中一个有关，是使用单调队列进行优化的基本条件。

5.5 斜率优化 dp

简单来说，就是利用单调队列维护下凸壳。具有以下形式状态转移方程的 dp 通常可以利用斜率优化，其中 $\mathrm{val(i,j)}$ 中可能含有 i 和 j 的乘积项：

$$F[i]=\min_{L(i)\le j\le R(i)}\{F[j]+val(i,j)\}$$

我们以一道例题为例介绍斜率优化 dp，假设状态转移方程是：

$$f_i=min(f_j+(s_i-s_j)^2+m)$$

其中 $\mathrm{s_i}$ 单调递增。先把 min 去掉，把状态转移方程变为

$$f_i=f_j+s_i^2+s_j^2-2s_is_j+m$$

由于每次进行转移时，一个 i 对应多个 j，所以相当于含 i 的项都是一个常量，含 j 的项都是一个变量，按照 $\mathrm{y=kx+b}$ 的形式分解原式，得到：

$$f_j+s_j^2=2s_is_j+f_i-s_i^2-m$$

显然对于一个确定的 i，斜率 $2s_i$ 是确定的，同时还知道经过点 $(s_j, f_j + s_j^2)$（由于 j 的值可以变化，所以这是一组点，也就对应了一组斜率相同的直线），于是通过让截距最小，就能让 f_i 最小。也就是让斜率为 $2s_i$ 的直线从下往上扫描，碰到第一个集合中的点时获得最小的截距。

当 i 的值增加时，斜率也增加，那这条直线先碰到的点也一定是下凸壳上的点，所以我们利用单调队列维护决策点（$(s_j, f_j + s_j^2)$）的下凸壳即可：

- 若新点 $i-1$ 与队尾点的斜率 $\leq$ 队尾临点直线的斜率，则队尾出队，删除无用点，维护决策点的下凸壳。
- 新点 $i-1$ 入队。
- 若队头临点直线的斜率 $\leq k_i$，则队头出队，删除过时点（**这一步是基于随着 i 的增加，斜率一定增加才有的**）。
- 此时队头就是最优决策点，做状态转移。

Listing 61: 斜率优化 dp

```cpp
// 斜率优化：时间复杂度 O(n)
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e5 + 5;

int n, m, q[maxn];
ll s[maxn], f[maxn];

ll delta_y(int i, int j) {
    ll y_i = f[i] + s[i] * s[i], y_j = f[j] + s[j] * s[j];
    return y_i - y_j;
}

ll delta_x(int i, int j) {
    ll x_i = s[i], x_j = s[j];
    return x_i - x_j;
}

ll cmp_slope(int i, int j, int k) {
    return delta_y(i, j) * delta_x(j, k) - delta_y(j, k) * delta_x(i, j);
}

int main() {
    scanf("%d%d", &n, &m);
    for(int i = 1; i <= n; i++)
        scanf("%d", &s[i]), s[i] += s[i - 1];
    int h = 1, t = 0;

    for(int i = 1; i <= n; i++) {
        while(h < t && cmp_slope(i - 1, q[t], q[t - 1]) <= 0)
            t--;
```

```
        q[++t] = i - 1;
        while(h < t && delta_y(q[h + 1], q[h]) <= delta_x(q[h + 1], q[h]) * 2 * s[i
            ])
            h++;
        int j = q[h];
        f[i] = f[j] + (s[i] - s[j]) * (s[i] - s[j]) + m;
    }
    printf("%lld\n", f[n]);
    return 0;
}
```

如果 s_i 不具有单调性的话，此时就必须保留整个凸壳，也就是每次不需要把队头与斜率进行比较。同时，队头此时不一定是最优决策了，这样一来就必须在单调队列中二分查找，找到一个位置 p，使得左侧线段斜率都比目标值小，右侧线段斜率都比目标值大，时间复杂度 O(nlogn)。

Listing 62: 斜率优化 dp + 二分法

```
// 斜率优化 + 二分法: 时间复杂度 O(nlogn)
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 3e5 + 5;

int n, m, q[maxn];
ll s[maxn], f[maxn];

ll delta_y(int i, int j) {
    ll y_i = f[i] + s[i] * s[i], y_j = f[j] + s[j] * s[j];
    return y_i - y_j;
}

ll delta_x(int i, int j) {
    ll x_i = s[i], x_j = s[j];
    return x_i - x_j;
}

ll cmp_slope(int i, int j, int k) {
    return delta_y(i, j) * delta_x(j, k) - delta_y(j, k) * delta_x(i, j);
}

int find(int l, int r, int i) {
    // 目标斜率 k
    // 寻找第一个 >= 斜率的线段, 它的左端点即为所求
    int k_i = 2 * s[i];
    int ret = r;
    while(l <= r) {
        int mid = l + r >> 1;
        if(k_i * delta_x(q[mid + 1], q[mid]) <= delta_y(q[mid + 1], q[mid]))
            ret = mid, r = mid - 1;
        else
            l = mid + 1;
```

```
35      }
36      return ret;
37  }
38
39  int main() {
40      scanf("%d%d", &n, &m);
41      for(int i = 1; i <= n; i++)
42          scanf("%d", &s[i]), s[i] += s[i - 1];
43      int h = 1, t = 0;
44
45      for(int i = 1; i <= n; i++) {
46          while(h < t && cmp_slope(i - 1, q[t], q[t - 1]) <= 0)
47              t--;
48          q[++t] = i - 1;
49          int j = q[find(h, t, i)];
50          f[i] = f[j] + (s[i] - s[j]) * (s[i] - s[j]) + m;
51      }
52      printf("%lld\n", f[n]);
53      return 0;
54  }
```

5.6 四边形不等式优化 dp

设 w(x,y) 是定义在整数集合上的二元函数，若对于定义域上的任意整数 a,b,c,d，其中 $a \leq b \leq c \leq d$，都有 $w(a,d)+w(b,c) \geq w(a,c)+w(b,d)$ 成立，则称函数 w 满足四边形不等式。

5.6.1 一维线性 dp 的四边形不等式优化

对于形如 $F[i]=\min_{0\leq j<i}\{F[j]+val(j,i)\}$ 的状态转移方程，记 p[i] 为令 F[i] 取到最小值的 j 的值，即 p[i] 是 F[i] 的最优决策。若 p 在 $[1,N]$ 上单调不减，则称 F 具有决策单调性。

在状态转移方程 $F[i]=\min_{0\leq j<i}F[j]+val(j,i)$ 中，若 val 满足四边形不等式，则 F 具有决策单调性。

当 F 具有决策单调性时，可以用类似于单调队列的结构，将时间复杂度降到 $O(nlogn)$。我们执行以下操作：

1. 开始时，我们将 [0, 1, n] 插入到队中，分别表示决策和左右边界。
2. 检查队头：设队头为 (j_0,l_0,r_0)，若 $r_0=i-1$，删除队头。否则令 $l_0=i$。
3. 取队头保存的 j 为最优决策，执行状态转移，计算出 $F[i]$。
4. 尝试插入新决策 i（即根据决策单调性，从某一个位置开始，将该位置直到最后的决策都改为 i），步骤如下：
   (a) 取出队尾，记为 (j_t,l_t,r_t)。
   (b) 若对于 $F[l_t]$ 来说，i 是比 j_t 更优的决策，即 $F[i]+val(i,l_t)\leq F[j_t]+val(j_t,l_t)$，记 $pos=l_t$，删除队尾，回到步骤 (a)
   (c) 若对于 $F[r_t]$ 来说，j_t 是比 i 更优的决策，即 $F[j_t]+val(j_t,r_t)\leq F[i]+val(i,r_t)$，去往步骤 (e)

(d) 否则，在 $[l_t, r_t]$ 上二分查找，求出位置 pos，在此之前决策 j_t 更优，在此之后决策 i 更优，去往步骤 (e)

(e) 把三元组 (i, pos, n) 插入队尾

Listing 63: 四边形不等式优化一维 dp

```
// 四边形不等式 + 二分队列：时间复杂度 O(nlogn)
// 下面的 val(j, i) 实际上表示我们上面说的 F[j] + val(j, i)
void quad_dp() {
    h = 0, t = -1;
    q[++t] = 0;
    lt[0] = 1, rt[0] = n;
    for (int j = 1; j <= n; ++j) {
        // 1. 检查队头
        if (h <= t && rt[q[h]] < j)
            h++;
        if (h <= t)
            lt[q[h]] = j;

        // 2. 取队头的 p 作为 i 的最优决策，进行状态转移
        f[j] = val(q[h], j);
        pre[j] = q[h];

        // 3. 尝试插入新决策
        while (h <= t && val(j, lt[q[t]]) < val(q[t], lt[q[t]]))
            t--;

        if (h > t) {
            lt[j] = j;
            rt[j] = n;
            q[++t] = j;
        }
        else if (val(j, rt[q[t]]) >= val(q[t], rt[q[t]])) {
            if (rt[q[t]] < n) {
                lt[j] = rt[q[t]] + 1;
                rt[j] = n;
                q[++t] = j;
            }
        }
        else {
            int ll = lt[q[t]], rr = rt[q[t]], i = rr;
            // 二分
            while (ll <= rr) {
                int mm = (ll + rr) / 2;
                if (val(j, mm) < val(q[t], mm))
                    i = mm, rr = mm - 1;
                else
                    ll = mm + 1;
            }
            rt[q[t]] = i - 1;
```

```
            lt[j] = i;
            rt[j] = n;
            q[++t] = j;
        }
    }
}
```

注意，为了保持决策单调性，一些题目的要求中有类似于如果最后结果大于 100，输出 no solution，这个时候我们不能在转移过程中直接取 min(101, ...)，而是算出最后结果后，将 n 个 f 都转移之后再跟 100 比较（不然的话，由于取了 min，决策单调性可能不再成立，就不能再用上面那样二分队列的做法了）。

5.6.2 二维区间 dp 的四边形不等式优化

在区间 dp 问题中，我们经常遇到下面这样的状态转移方程：

$$F[i,j] = \min_{i \le k < j} \{F[i,k] + F[k+1,j] + w(i,j)\}$$

利用四边形不等式，也能对其进行优化。

定理 1 在状态转移方程 $F[i,j] = \min_{i \le k < j}\{F[i,k] + F[k+1,j] + w(i,j)\}$ 中（特别地，$F[i,i] = w[i,i] = 0$），如果下面两个条件成立：

1. w 满足四边形不等式
2. 对于任意的 $a \le b \le c \le d$，有 $w(a,d) \ge w(b,c)$

那么 F 也满足四边形不等式。

定理 2 在状态转移方程 $F[i,j] = \min_{i \le k < j}\{F[i,k] + F[k+1,j] + w(i,j)\}$ 中（特别地，$F[i,i] = w[i,i] = 0$），记 $P[i,j]$ 为令 $F[i,j]$ 取到最优值的 k。如果 F 满足四边形不等式，那么对于任意 $i < j$，有 $P[i,j-1] \le P[i,j] \le P[i+1,j]$。

5.7 数位 dp

带前置 0 的写法

Listing 64: 数位 dp，允许有前置 0

```
// 求 [0, n] 中满足条件的数的个数
// high 表示这一位前面是不是都是卡着上限填的
// 当前置 0 不影响答案时可以使用
// 使用前，一定要补前置 0 让所有数位数相同
int dfs(int u, int high) {
    if (u == s.size()) return 1;
    if (!high && f[u] != -1) return f[u];
    int l = 0, r = high ? s[u] - '0' : 9;
    int ret = 0;
    for (int i = l; i <= r; i++)
        ret += dfs(u + 1, high && i == r);
    if (!high) f[u] = ret;
    return ret;
}
```

不带前置 0 的写法

Listing 65: 数位 dp，不允许有前置 0

```
// 求 [1, n] 中满足条件的数的个数
// lead 表示前面是否出现过前置 0 以外的其它数字
// 当前置 0 影响答案时可以使用
// 使用前，一定要补前置 0 让所有数位数相同
int dfs(int u, bool high, bool lead) {
    if (u == s.size()) return !lead;
    if (!high && !lead && f[u] != -1) return f[u];
    int l = 0, r = high ? s[u] - '0' : 9;
    int ret = 0;
    if(lead)
        ret = dfs(u + 1, high && s[u] == '0', true);

    for (int i = max(l, (int)lead); i <= r; i++)
        ret += dfs(u + 1, high && i == r, false);
    if (!high && !lead) f[u] = ret;
    return ret;
}
```