Algorithm Library

st1vdy

2021 年 10 月 26 日

目录

1 多项式

1.1 FFT - tourist

```cpp
/* copy from tourist */
namespace FFT {
    typedef double dbl;

    struct num {
        dbl x, y;
        num() { x = y = 0; }
        num(dbl x, dbl y) : x(x), y(y) { }
    };

    inline num operator+(num a, num b) { return num(a.x + b.x, a.y + b.y); }
    inline num operator-(num a, num b) { return num(a.x - b.x, a.y - b.y); }
    inline num operator*(num a, num b) { return num(a.x * b.x - a.y * b.y, a
        .x * b.y + a.y * b.x); }
    inline num conj(num a) { return num(a.x, -a.y); }

    int base = 1;
    vector<num> roots = { {0, 0}, {1, 0} };
    vector<int> rev = { 0, 1 };

    const dbl PI = acosl(-1.0);

    void ensure_base(int nbase) {
        if (nbase <= base) {
            return;
        }
        rev.resize(1 << nbase);
        for (int i = 0; i < (1 << nbase); i++) {
            rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
        }
        roots.resize(1 << nbase);
        while (base < nbase) {
            dbl angle = 2 * PI / (1 << (base + 1));
            for (int i = 1 << (base - 1); i < (1 << base); i++) {
                roots[i << 1] = roots[i];
                dbl angle_i = angle * (2 * i + 1 - (1 << base));
                roots[(i << 1) + 1] = num(cos(angle_i), sin(angle_i));
            }
            base++;
        }
```

```cpp
    }

    void fft(vector<num>& a, int n = -1) {
        if (n == -1) {
            n = a.size();
        }
        assert((n & (n - 1)) == 0);
        int zeros = __builtin_ctz(n);
        ensure_base(zeros);
        int shift = base - zeros;
        for (int i = 0; i < n; i++) {
            if (i < (rev[i] >> shift)) {
                swap(a[i], a[rev[i] >> shift]);
            }
        }
        for (int k = 1; k < n; k <<= 1) {
            for (int i = 0; i < n; i += 2 * k) {
                for (int j = 0; j < k; j++) {
                    num z = a[i + j + k] * roots[j + k];
                    a[i + j + k] = a[i + j] - z;
                    a[i + j] = a[i + j] + z;
                }
            }
        }
    }

    vector<num> fa, fb;

    vector<long long> multiply(vector<int>& a, vector<int>& b) {
        int need = a.size() + b.size() - 1;
        int nbase = 1;
        while ((1 << nbase) < need) nbase++;
        ensure_base(nbase);
        int sz = 1 << nbase;
        if (sz > (int)fa.size()) {
            fa.resize(sz);
        }
        for (int i = 0; i < sz; i++) {
            int x = (i < (int)a.size() ? a[i] : 0);
            int y = (i < (int)b.size() ? b[i] : 0);
            fa[i] = num(x, y);
        }
        fft(fa, sz);
```

```
    num r(0, -0.25 / (sz >> 1));
    for (int i = 0; i <= (sz >> 1); i++) {
        int j = (sz - i) & (sz - 1);
        num z = (fa[j] * fa[j] - conj(fa[i] * fa[i])) * r;
        if (i != j) {
            fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[j])) * r;
        }
        fa[i] = z;
    }
    for (int i = 0; i < (sz >> 1); i++) {
        num A0 = (fa[i] + fa[i + (sz >> 1)]) * num(0.5, 0);
        num A1 = (fa[i] - fa[i + (sz >> 1)]) * num(0.5, 0) * roots[(sz
            >> 1) + i];
        fa[i] = A0 + A1 * num(0, 1);
    }
    fft(fa, sz >> 1);
    vector<long long> res(need);
    for (int i = 0; i < need; i++) {
        if (i % 2 == 0) {
            res[i] = fa[i >> 1].x + 0.5;
        } else {
            res[i] = fa[i >> 1].y + 0.5;
        }
    }
    return res;
}

vector<long long> square(const vector<int>& a) {
    int need = a.size() + a.size() - 1;
    int nbase = 1;
    while ((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if ((sz >> 1) > (int)fa.size()) {
        fa.resize(sz >> 1);
    }
    for (int i = 0; i < (sz >> 1); i++) {
        int x = (2 * i < (int)a.size() ? a[2 * i] : 0);
        int y = (2 * i + 1 < (int)a.size() ? a[2 * i + 1] : 0);
        fa[i] = num(x, y);
    }
    fft(fa, sz >> 1);
    num r(1.0 / (sz >> 1), 0.0);
```

```
        for (int i = 0; i <= (sz >> 2); i++) {
            int j = ((sz >> 1) - i) & ((sz >> 1) - 1);
            num fe = (fa[i] + conj(fa[j])) * num(0.5, 0);
            num fo = (fa[i] - conj(fa[j])) * num(0, -0.5);
            num aux = fe * fe + fo * fo * roots[(sz >> 1) + i] * roots[(sz
                >> 1) + i];
            num tmp = fe * fo;
            fa[i] = r * (conj(aux) + num(0, 2) * conj(tmp));
            fa[j] = r * (aux + num(0, 2) * tmp);
        }
        fft(fa, sz >> 1);
        vector<long long> res(need);
        for (int i = 0; i < need; i++) {
            if (i % 2 == 0) {
                res[i] = fa[i >> 1].x + 0.5;
            } else {
                res[i] = fa[i >> 1].y + 0.5;
            }
        }
        return res;
    }

    vector<int> multiply_mod(vector<int>& a, vector<int>& b, int m, int eq =
        0) {
        int need = a.size() + b.size() - 1;
        int nbase = 0;
        while ((1 << nbase) < need) nbase++;
        ensure_base(nbase);
        int sz = 1 << nbase;
        if (sz > (int)fa.size()) {
            fa.resize(sz);
        }
        for (int i = 0; i < (int)a.size(); i++) {
            int x = (a[i] % m + m) % m;
            fa[i] = num(x & ((1 << 15) - 1), x >> 15);
        }
        fill(fa.begin() + a.size(), fa.begin() + sz, num{ 0, 0 });
        fft(fa, sz);
        if (sz > (int) fb.size()) {
            fb.resize(sz);
        }
        if (eq) {
            copy(fa.begin(), fa.begin() + sz, fb.begin());
```

```
        } else {
            for (int i = 0; i < (int)b.size(); i++) {
                int x = (b[i] % m + m) % m;
                fb[i] = num(x & ((1 << 15) - 1), x >> 15);
            }
            fill(fb.begin() + b.size(), fb.begin() + sz, num{ 0, 0 });
            fft(fb, sz);
        }
        dbl ratio = 0.25 / sz;
        num r2(0, -1);
        num r3(ratio, 0);
        num r4(0, -ratio);
        num r5(0, 1);
        for (int i = 0; i <= (sz >> 1); i++) {
            int j = (sz - i) & (sz - 1);
            num a1 = (fa[i] + conj(fa[j]));
            num a2 = (fa[i] - conj(fa[j])) * r2;
            num b1 = (fb[i] + conj(fb[j])) * r3;
            num b2 = (fb[i] - conj(fb[j])) * r4;
            if (i != j) {
                num c1 = (fa[j] + conj(fa[i]));
                num c2 = (fa[j] - conj(fa[i])) * r2;
                num d1 = (fb[j] + conj(fb[i])) * r3;
                num d2 = (fb[j] - conj(fb[i])) * r4;
                fa[i] = c1 * d1 + c2 * d2 * r5;
                fb[i] = c1 * d2 + c2 * d1;
            }
            fa[j] = a1 * b1 + a2 * b2 * r5;
            fb[j] = a1 * b2 + a2 * b1;
        }
        fft(fa, sz);
        fft(fb, sz);
        vector<int> res(need);
        for (int i = 0; i < need; i++) {
            long long aa = fa[i].x + 0.5;
            long long bb = fb[i].x + 0.5;
            long long cc = fa[i].y + 0.5;
            res[i] = (aa + ((bb % m) << 15) + ((cc % m) << 30)) % m;
        }
        return res;
    }

    vector<int> square_mod(vector<int>& a, int m) {
```

```
        return multiply_mod(a, a, m, 1);
    }
};
```

1.2 形式幂级数

2 数论

2.1 简单的防爆模板

```
namespace SimpleMod {
    constexpr int MOD = (int)1e9 + 7;
    inline int norm(long long a) { return (a % MOD + MOD) % MOD; }
    inline int add(int a, int b) { return a + b >= MOD ? a + b - MOD : a + b
        ; }
    inline int sub(int a, int b) { return a - b < 0 ? a - b + MOD : a - b; }
    inline int mul(int a, int b) { return (int)((long long)a * b % MOD); }
    inline int powmod(int a, long long b) {
        int res = 1;
        while (b > 0) {
            if (b & 1) res = mul(res, a);
            a = mul(a, a);
            b >>= 1;
        }
        return res;
    }
    inline int inv(int a) {
        a %= MOD;
        if (a < 0) a += MOD;
        int b = MOD, u = 0, v = 1;
        while (a) {
            int t = b / a;
            b -= t * a; swap(a, b);
            u -= t * v; swap(u, v);
        }
        assert(b == 1);
        if (u < 0) u += MOD;
        return u;
    }
}
```

2.2 筛法

2.2.1 线性素数筛

```
vector<bool> isPrime; // true 表示非素数 false 表示是素数
vector<int> prime; // 保存素数
int sieve(int n) {
    isPrime.resize(n + 1, false);
    isPrime[0] = isPrime[1] = true;
    for (int i = 2; i <= n; i++) {
        if (!isPrime[i]) prime.emplace_back(i);
        for (int j = 0; j < (int)prime.size() && prime[j] * i <= n; j++) {
            isPrime[prime[j] * i] = true;
            if (!(i % prime[j])) break;
        }
    }
    return (int)prime.size();
}
```

2.2.2 线性欧拉函数筛

```
bool is_prime[SIZE];
int prime[SIZE], phi[SIZE]; // phi[i] 表示 i 的欧拉函数值
int Phi(int n) { // 线性筛素数的同时线性求欧拉函数
    phi[1] = 1; is_prime[1] = true;
    int p = 0;
    for (int i = 2; i <= n; i++) {
        if (!is_prime[i]) prime[p++] = i, phi[i] = i - 1;
        for (int j = 0; j < p && prime[j] * i <= n; j++) {
            is_prime[prime[j] * i] = true;
            if (!(i % prime[j])) {
                phi[i * prime[j]] = phi[i] * prime[j];
                break;
            }
            phi[i * prime[j]] = phi[i] * (prime[j] - 1);
        }
    }
    return p;
}
```

2.2.3 线性约数个数函数筛

```
bool is_prime[SIZE];
```

```cpp
int prime[SIZE], d[SIZE], num[SIZE]; // d[i] 表示 i 的因子数 num[i] 表示 i 的最小质因子出现次数
int getFactors(int n) { // 线性筛因子数
    d[1] = 1; is_prime[1] = true;
    int p = 0;
    for (int i = 2; i <= n; i++) {
        if (!is_prime[i]) prime[p++] = i, d[i] = 2, num[i] = 1;
        for (int j = 0; j < p && prime[j] * i <= n; j++) {
            is_prime[prime[j] * i] = true;
            if (!(i % prime[j])) {
                num[i * prime[j]] = num[i] + 1;
                d[i * prime[j]] = d[i] / num[i * prime[j]] * (num[i * prime[j]] + 1);
                break;
            }
            num[i * prime[j]] = 1;
            d[i * prime[j]] = d[i] + d[i];
        }
    }
    return p;
}
```

2.2.4 线性素因子个数函数筛

```cpp
bool is_prime[SIZE];
int prime[SIZE], num[SIZE]; // num[i] 表示 i 的质因子数
int getPrimeFactors(int n) { // 线性筛质因子数
    is_prime[1] = true;
    int p = 0;
    for (int i = 2; i <= n; i++) {
        if (!is_prime[i]) prime[p++] = i, num[i] = 1;
        for (int j = 0; j < p && prime[j] * i <= n; j++) {
            is_prime[prime[j] * i] = true;
            if (!(i % prime[j])) {
                num[i * prime[j]] = num[i];
                break;
            }
            num[i * prime[j]] = num[i] + 1;
        }
    }
    return p;
}
```

2.2.5 线性约数和函数筛

```
bool is_prime[SIZE];
int prime[SIZE], f[SIZE], g[SIZE]; // f[i] 表示 i 的约数和
int getSigma(int n) {
    g[1] = f[1] = 1; is_prime[1] = true;
    int p = 0;
    for (int i = 2; i <= n; i++) {
        if (!is_prime[i]) prime[p++] = i, f[i] = g[i] = i + 1;
        for (int j = 0; j < p && prime[j] * i <= n; j++) {
            is_prime[prime[j] * i] = true;
            if (!(i % prime[j])) {
                g[i * prime[j]] = g[i] * prime[j] + 1;
                f[i * prime[j]] = f[i] / g[i] * g[i * prime[j]];
                break;
            }
            f[i * prime[j]] = f[i] * f[prime[j]];
            g[i * prime[j]] = 1 + prime[j];
        }
    }
    return p;
}
```

2.2.6 线性莫比乌斯函数筛

```
bool is_prime[SIZE];
int prime[SIZE], mu[SIZE]; // mu[i] 表示 i 的莫比乌斯函数值
int getMu(int n) { // 线性筛莫比乌斯函数
    mu[1] = 1; is_prime[1] = true;
    int p = 0;
    for (int i = 2; i <= n; i++) {
        if (!is_prime[i]) prime[p++] = i, mu[i] = -1;
        for (int j = 0; j < p && prime[j] * i <= n; j++) {
            is_prime[prime[j] * i] = true;
            if (!(i % prime[j])) {
                mu[i * prime[j]] = 0;
                break;
            }
            mu[i * prime[j]] = -mu[i];
        }
    }
    return p;
}
```

2.3 扩展欧几里得

2.3.1 线性同余方程最小非负整数解

exgcd 求 $ax+by=c$ 的最小非负整数解详解：

1. 求出 a,b 的最大公约数 $g=\gcd(a,b)$ ，根据裴蜀定理检查是否满足 $c\%g=0$ ，不满足则无解；
2. 调整系数 a,b,c 为 $a'=\frac{a}{g},b'=\frac{b}{g},c'=\frac{c}{g}$ ，这是因为 $ax+by=c$ 和 $a'x+b'y=c'$ 是完全等价的；
3. 实际上 exgcd 求解的方程是 $a'x+b'y=1$ ，求解前需要注意让系数 $a',b'\geq 0$ （举个例子，如果系数 b' 原本 <0 ，我们可以翻转 b' 的符号然后令解 (x,y) 为 $(x,-y)$ ，但是求解的时候要把 y 翻回来）；
4. 我们通过 exgcd 求出一组解 (x_0,y_0) ，这组解满足 $a'x_0+b'y_0=1$ ，为了使解合法我们需要令 $x_0=c'x_0,y_0=c'y_0$ ，于是有 $a'(c'x_0)+b'(c'y_0)=c''$ ；
5. 考虑到 $a'x_0+b'y_0=1$ 等价于同余方程 $a'x_0\equiv 1 \pmod{b'}$ ，因此为了求出最小非负整数解，我们最后还需要对 b' 取模；
6. 最后注意特判 $c'=0$ 的情况，如果要求解 y 且系数 b 发生了翻转，将其翻转回来。

```cpp
long long exgcd(long long a, long long b, long long& x, long long& y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    long long g = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return g;
}

ll x, y; // 最小非负整数解
bool solve(ll a, ll b, ll c) { // ax+by=c
    ll g = gcd(a, b);
    if (c % g) return false;
    a /= g, b /= g, c /= g;
    bool flag = false;
    if (b < 0) b = -b, flag = true;
    exgcd(a, b, x, y);
    x = (x * c % b + b) % b;
    if (flag) b = -b;
    y = (c - a * x) / b;
    if (!c) x = y = 0; // ax+by=0
    return true;
}
```

2.4 欧拉定理

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

2.5 欧拉函数

2.6 中国剩余定理

2.6.1 CRT

```cpp
// 求解形如 x = ci (mod mi) 的线性方程组 (mi, mj)必须两两互质
long long CRT(vector<long long>& c, vector<long long>& m) {
    long long M = m[0], ans = 0;
    for (int i = 1; i < (int)m.size(); ++i) M *= m[i];
    for (int i = 0; i < (int)m.size(); ++i) { // Mi * ti * ci
        long long mi = M / m[i];
        long long ti = inv(mi, m[i]); // mi 模 m[i] 的逆元
        ans = (ans + mi * ti % M * c[i] % M) % M;
    }
    ans = (ans + M) % M; // 返回模 M 意义下的唯一解
    return ans;
}
```

2.6.2 EXCRT

```cpp
long long exgcd(long long a, long long b, long long& x, long long& y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    long long g = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return g;
}

long long mulmod(long long x, long long y, const long long z) { // x * y % z 防爆
    return (x * y - (long long)(((long double)x * y + 0.5) / (long double)z) * z + z) % z;
}

// 求解形如 x = ci (mod mi) 的线性方程组
```

```
long long EXCRT(vector<long long>& c, vector<long long>& m) {
    long long M = m[0], ans = c[0];
    for (int i = 1; i < (int)m.size(); ++i) { // M * x - mi * y = ci - C
        long long x, y, C = ((c[i] - ans) % m[i] + m[i]) % m[i]; // ci - C
        long long G = exgcd(M, m[i], x, y);
        if (C % G) return -1; // 无解
        long long P = m[i] / G;
        x = mulmod(C / G, x, P); // 防爆求最小正整数解 x
        ans += x * M;
       M *= P; // LCM(M, mi)
        ans = (ans % M + M) % M;
    }
    return ans;
}
```

2.7 BSGS

2.8 迪利克雷卷积

$$g(1)S(n) = \sum_{i=1}^{n}(f*g)(i) - \sum_{i=2}^{n} g(i)S(\lfloor \frac{n}{i} \rfloor)$$

2.9 杜教筛

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

3 线性代数

3.1 高斯-约旦消元法

```
/*
 * 高斯-约旦消元法
 * 可以修改为解异或方程组 修改策略为
 * a+b -> a^b
 * a-b -> a^b
 * a*b -> a&b
 * a/b -> a*(b==1)
 * */
constexpr double eps = 1e-7;
double a[SIZE][SIZE], ans[SIZE];
void gauss(int n) {
    vector<bool> vis(n, false);
    for (int i = 0; i < n; i++) {
```

```
        for (int j = 0; j < n; j++) {
            if (vis[j]) continue;
            if (fabs(a[j][i]) > eps) {
                vis[i] = true;
                for (int k = 0; k <= n; k++) swap(a[i][k], a[j][k]);
                break;
            }
        }
        if (fabs(a[i][i]) < eps) continue;
        for (int j = 0; j <= n; j++) {
            if (i != j && fabs(a[j][i]) > eps) {
                double res = a[j][i] / a[i][i];
                for (int k = 0; k <= n; k++) a[j][k] -= a[i][k] * res;
            }
        }
    }
}

int check(int n) { // 解系检测
    int status = 1;
    for (int i = n - 1; i >= 0; i--) {
        if (fabs(a[i][i]) < eps && fabs(a[i][n]) > eps) return -1; // 无解
        if (fabs(a[i][i]) < eps && fabs(a[i][n]) < eps) status = 0; // 无穷解
        ans[i] = a[i][n] / a[i][i];
        if (fabs(ans[i]) < eps) ans[i] = 0;
    }
    return status; // 唯一解或无穷解
}
```

3.2 高斯消元法-bitset

```
constexpr int SIZE = 1001;
bitset<SIZE> a[SIZE];
int ans[SIZE];
void gauss(int n) { // bitset版高斯消元 用于求解异或线性方程组
    bitset<SIZE> vis;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (vis[j]) continue;
            if (a[j][i]) {
                vis.set(i);
                swap(a[i], a[j]);
```

```
                break;
            }
        }
        if (!a[i][i]) continue;
        for (int j = 0; j <= n; j++) {
            if (i != j && (a[j][i] & a[i][i])) {
                a[j] ^= a[i];
            }
        }
    }
}
```

3.3 线性基

```
struct linearBasis {
    /* 线性基性质:
     * 1.若a[i]!=0 (即主元i存在)
     *   则线性基中只有a[i]的第i位是1 (只存在一个主元)
     *   且此时a[i]的最高位就是第i位
     * 2.将数组a插入线性基 假设有|B|个元素成功插入
     *   则数组a中每个不同的子集异或和都出现 2^(n-|B|) 次
     * */
    static const int MAXL = 60;
    long long a[MAXL + 1];
    int id[MAXL + 1];
    int zero;
    /* 0的标志位 =1则表示0可以被线性基表示出来
     * 求第k大元素时 需要注意题意中线性基为空时是否可以表示0
     * 默认不可以表示 有必要时进行修改即可
     * */
    linearBasis() {
        zero = 0;
        fill(a, a + MAXL + 1, 0);
    }
    long long& operator[] (int k) { return a[k]; }
    bool insert(long long x) {
        for (int j = MAXL; ~j; j--) {
            if (!(x & (1LL << j))) { // 如果 x 的第 j 位为 0, 则跳过
                continue;
            }
            if (a[j]) { // 如果 a[j] != 0, 则用 a[j] 消去 x 的第 j 位上的 1
                x ^= a[j];
            } else { // 找到插入位置
```

```
                for (int k = 0; k < j; k++) {
                    if (x & (1LL << k)) { // 如果x存在某个低位线性基的主元k则消去
                        x ^= a[k];
                    }
                }
                for (int k = j + 1; k <= MAXL; k++) {
                    if (a[k] & (1LL << j)) { // 如果某个高位线性基存在主元j则消去
                        a[k] ^= x;
                    }
                }
                a[j] = x;
                return true;
            }
        }
        zero = 1;
        return false;
    }
    long long query_max() { // 最大值
        long long res = 0;
        for (int i = MAXL; ~i; i--) {
            res ^= a[i];
        }
        return res;
    }
    long long query_max(long long x) { // 线性基异或x的最大值
        for (int i = MAXL; ~i; i--) {
            if ((x ^ a[i]) > x) {
                x ^= a[i];
            }
        }
        return x;
    }
    long long query_min() { // 最小值
        for (int i = 0; i < MAXL; i++) {
            if (a[i]) {
                return a[i];
            }
        }
        return -1; // 线性基为空
    }
    long long query_min(long long x) { // 线性基异或x的最小值
```

```cpp
        for (int i = MAXL; ~i; i--) {
            if ((x ^ a[i]) < x) {
                x ^= a[i];
            }
        }
        return x;
    }
    int count(long long x) { // 元素 x 能否被线性基内元素表示
        int res = 0;
        vector<long long> b(MAXL + 1);
        for (int i = 0; i <= MAXL; i++) {
            b[i] = a[i];
        }
        res = this->insert(x);
        for (int i = 0; i <= MAXL; i++) {
            a[i] = b[i];
        }
        return !res; // 成功插入则无法表示 失败则可以表示
    }
    int size() { // 线性基有效元素数量
        int res = 0;
        for (int i = 0; i <= MAXL; i++) {
            if (a[i]) {
                res++;
            }
        }
        return res;
    }
    long long kth_element(long long k) { // 第k大元素
        vector<long long> b;
        for (int i = 0; i <= MAXL; i++) {
            if (a[i]) {
                b.push_back(a[i]);
            }
        }
        if (zero) {
            if (--k == 0) {
                return 0;
            }
        }
        if (k >= (1LL << this->size())) { // k超过了线性基可以表示的最大数量
            return -1;
        }
```

```cpp
        long long res = 0;
        for (int i = 0; i <= MAXL; i++) {
            if (k & (1LL << i)) {
                res ^= b[i];
            }
        }
        return res;
    }
    long long rank(long long x) { // 元素x在线性基内的排名（默认不考虑0）
        vector<long long> b;
        for (int i = 0; i <= MAXL; i++) {
            if (a[i]) {
                b.push_back(1LL << i);
            }
        }
        long long res = 0;
        for (int i = 0; i < (int)b.size(); i++) {
            if (x & b[i]) {
                res |= (1LL << i);
            }
        }
        return res;
    }
    void clear() {
        zero = 0;
        fill(a, a + MAXL + 1, 0);
    }
};
```

3.4 矩阵树定理

```cpp
/*
 * 矩阵树定理
 * 有向图：若 u->v 有一条权值为 w 的边 基尔霍夫矩阵 a[v][v] += w, a[v][u] -= w
 * 生成树数量为除去 根所在行和列 后的n-1阶行列式的值
 * 无向图：若 u->v 有一条权值为 w 的边 基尔霍夫矩阵 a[v][v] += w, a[v][u] -= w, a[u][u] += w, a[u][v] -= w
 * 生成树数量为除去 任意一行和列 后的n-1阶行列式的值
 * 无权图则边权默认为1
 * */
typedef long long ll;
typedef unsigned long long u64;
```

```
int a[SIZE][SIZE];
int gauss(int a[][SIZE], int n) { // 任意模数求行列式 O(n^2(n + log(mod)))
    int ans = 1;
    for (int i = 1; i <= n; i++) {
        int* x = 0, * y = 0;
        for (int j = i; j <= n; j++) {
            if (a[j][i] && (x == NULL || a[j][i] < x[i])) {
                x = a[j];
            }
        }
        if (x == 0) {
            return 0;
        }
        for (int j = i; j <= n; j++) {
            if (a[j] != x && a[j][i]) {
                y = a[j];
                for (;;) {
                    int v = md - y[i] / x[i], k = i;
                    for (; k + 3 <= n; k += 4) {
                        y[k + 0] = (y[k + 0] + u64(x[k + 0]) * v) % md;
                        y[k + 1] = (y[k + 1] + u64(x[k + 1]) * v) % md;
                        y[k + 2] = (y[k + 2] + u64(x[k + 2]) * v) % md;
                        y[k + 3] = (y[k + 3] + u64(x[k + 3]) * v) % md;
                    }
                    for (; k <= n; ++k) {
                        y[k] = (y[k] + u64(x[k]) * v) % md;
                    }
                    if (!y[i]) break;
                    swap(x, y);
                }
            }
        }
        if (x != a[i]) {
            for (int k = i; k <= n; k++) {
                swap(x[k], a[i][k]);
            }
            ans = md - ans;
        }
        ans = 1LL * ans * a[i][i] % md;
    }
    return ans;
}
```

3.5 LGV 引理

4 组合数学

4.1 组合数预处理

```
namespace BinomialCoefficient {
    vector<int> fac, ifac, iv;
    // 组合数预处理 option=1则还会预处理线性逆元
    void prepareFactorials(int maximum = 1000000, int option = 0) {
        fac.assign(maximum + 1, 0);
        ifac.assign(maximum + 1, 0);
        fac[0] = ifac[0] = 1;
        if (option) { // O(3n)
            iv.assign(maximum + 1, 1);
            for (int p = 2; p * p <= MOD; p++)
                assert(MOD % p != 0);
            for (int i = 2; i <= maximum; i++)
                iv[i] = mul(iv[MOD % i], (MOD - MOD / i));
            for (int i = 1; i <= maximum; i++) {
                fac[i] = mul(i, fac[i - 1]);
                ifac[i] = mul(iv[i], ifac[i - 1]);
            }
        } else { // O(2n + log(MOD))
            for (int i = 1; i <= maximum; i++)
                fac[i] = mul(fac[i - 1], i);
            ifac[maximum] = inv(fac[maximum]);
            for (int i = maximum; i; i--)
                ifac[i - 1] = mul(ifac[i], i);
        }
    }
    inline int binom(int n, int m) {
        if (n < m || n < 0 || m < 0) return 0;
        return mul(fac[n], mul(ifac[m], ifac[n - m]));
    }
}
```

4.2 卢卡斯定理

4.3 小球盒子模型

4.4 斯特林数

4.4.1 第一类斯特林数

第一类斯特林数 $\begin{bmatrix} n \\ k \end{bmatrix}$ 表示将 n 个不同元素划分入 k 个非空圆排列的方案数。

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix}$$

边界是 $\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1$ 。

第一类斯特林数三角形，从 s(1, 1) 开始：

1									
1	1								
2	3	1							
6	11	6	1						
24	50	35	10	1					
120	274	225	85	15	1				
720	1764	1624	735	175	21	1			
5040	13068	13132	6769	1960	322	28	1		
40320	109584	118124	67284	22449	4536	546	36	1	
362880	1026576	1172700	723680	269325	63273	9450	870	45	1

4.4.2 第二类斯特林数

第二类斯特林数 $\begin{Bmatrix} n \\ k \end{Bmatrix}$ 表示将 n 个不同元素划分为 k 个非空子集的方案数。

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix} + k \begin{Bmatrix} n-1 \\ k \end{Bmatrix}$$

边界是 $\begin{Bmatrix} 0 \\ 0 \end{Bmatrix} = 1$ 。

基于容斥原理的递推方法：

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$$

第二类斯特林数三角形，从 S(1, 1) 开始：

1									
1	1								
1	3	1							
1	7	6	1						
1	15	25	10	1					
1	31	90	65	15	1				
1	63	301	350	140	21	1			
1	127	966	1701	1050	266	28	1		
1	255	3025	7770	6951	2646	462	36	1	
1	511	9330	34105	42525	22827	5880	750	45	1

5 博弈论

6 图论

6.1 并查集

```
struct dsu {
private:
    // number of nodes
    int n;
    // root node: -1 * component size
    // otherwise: parent
    std::vector<int> pa;
public:
    dsu(int n_ = 0) : n(n_), pa(n_, -1) {}
    // find node x's parent
    int find(int x) {
        return pa[x] < 0 ? x : pa[x] = find(pa[x]);
    }
    // merge node x and node y
    // if x and y had already in the same component, return false, otherwise return true
    // Implement (union by size) + (path compression)
    bool unite(int x, int y) {
        int xr = find(x), yr = find(y);
        if (xr != yr) {
            if (-pa[xr] < -pa[yr]) std::swap(xr, yr);
            pa[xr] += pa[yr];
            pa[yr] = xr; // y -> x
            return true;
        }
        return false;
```

```
    }
    // size of the connected component that contains the vertex x
    int size(int x) {
        return -pa[find(x)];
    }
};
```

6.2 最小树形图

```
namespace ZL {
    // a 尽量开大，之后的边都塞在这个里面
    const int N = 100010, M = 100010, inf = 1e9;
    struct edge {
        int u, v, w, use, id;
        edge(int u_ = 0, int v_ = 0, int w_ = 0, int use_ = 0, int id_ = 0)
            : u(u_), v(v_), w(w_), use(use_), id(id_) {}
    }b[M], a[2000100];
    int n, m, ans, pre[N], id[N], vis[N], root, In[N], h[N], len, way[M];
    // 从root 出发能到达每一个点的最小树形图
    // 先调用init 然后把边add 进去，需要方案就getway, way[i] 为1 表示使用
    // 最小值保存在ans
    void init(int _n, int _root) { // 点数 根节点
        n = _n; m = 0; b[0].w = inf; root = _root;
    }
    void add(int u, int v, int w) {
        m++;
        b[m] = edge(u, v, w, 0, m);
        a[m] = b[m];
    }
    int work() {
        len = m;
        for (;;) {
            for (int i = 1; i <= n; i++) { pre[i] = 0; In[i] = inf; id[i] = 0; vis[i] = 0; h[i] = 0; }
            for (int i = 1; i <= m; i++) {
                if (b[i].u != b[i].v && b[i].w < In[b[i].v]) {
                    pre[b[i].v] = b[i].u; In[b[i].v] = b[i].w; h[b[i].v] = b[i].id;
                }
            }
            for (int i = 1; i <= n; i++) if (pre[i] == 0 && i != root) return 0;
            int cnt = 0; In[root] = 0;
```

```cpp
            for (int i = 1; i <= n; i++) {
                if (i != root) a[h[i]].use++; int now = i; ans += In[i];
                while (vis[now] == 0 && now != root) { vis[now] = i; now = pre[now]; }
                if (now != root && vis[now] == i) {
                    cnt++; int kk = now;
                    while (1) {
                        id[now] = cnt; now = pre[now];
                        if (now == kk) break;
                    }
                }
            }
            if (cnt == 0) return 1;
            for (int i = 1; i <= n; i++) if (id[i] == 0) id[i] = ++cnt;
            // 缩环，每一条接入的边都会茶包原来接入的那条边，所以要调整边权
            // 新加的边是u，茶包的边是v
            for (int i = 1; i <= m; i++) {
                int k1 = In[b[i].v], k2 = b[i].v;
                b[i].u = id[b[i].u];
                b[i].v = id[b[i].v];
                if (b[i].u != b[i].v) {
                    b[i].w -= k1; a[++len].u = b[i].id; a[len].v = h[k2]; b[i].id = len;
                }
            }
            n = cnt; root = id[root];
        }
        return 1;
    }
    void getway() {
        for (int i = 1; i <= m; i++) way[i] = 0;
        for (int i = len; i > m; i--) { a[a[i].u].use += a[i].use; a[a[i].v].use -= a[i].use; }
        for (int i = 1; i <= m; i++) way[i] = a[i].use;
    }
}
```

6.3 最近公共祖先

```cpp
constexpr int SIZE = 200010;
constexpr int DEPTH = 21; // 最大深度 2^DEPTH - 1
int pa[SIZE][DEPTH], dep[SIZE];
vector<int> g[SIZE]; //邻接表
```

```cpp
void dfs(int rt, int fin) { //预处理深度和祖先
    pa[rt][0] = fin;
    dep[rt] = dep[pa[rt][0]] + 1; //深度
    for (int i = 1; i < DEPTH; i++) { // rt 的 2^i 祖先等价于 rt 的 2^(i-1) 祖先 的 2^(i-1) 祖先
        pa[rt][i] = pa[pa[rt][i - 1]][i - 1];
    }
    int sz = g[rt].size();
    for (int i = 0; i < sz; ++i) {
        if (g[rt][i] == fin) continue;
        dfs(g[rt][i], rt);
    }
}

int LCA(int x, int y) {
    if (dep[x] > dep[y]) swap(x, y);
    int dif = dep[y] - dep[x];
    for (int j = 0; dif; ++j, dif >>= 1) {
        if (dif & 1) {
            y = pa[y][j]; //先跳到同一高度
        }
    }
    if (y == x) return x;
    for (int j = DEPTH - 1; j >= 0 && y != x; j--) { //从底往上跳
        if (pa[x][j] != pa[y][j]) { //如果当前祖先不相等 我们就需要更新
            x = pa[x][j];
            y = pa[y][j];
        }
    }
    return pa[x][0];
}
```

6.4 强连通分量

```cpp
namespace SCC {
    // Compressed Sparse Row
    template <class E> struct csr {
        std::vector<int> start;
        std::vector<E> elist;
        explicit csr(int n, const std::vector<std::pair<int, E>>& edges)
            : start(n + 1), elist(edges.size()) {
            for (auto e : edges) {
                start[e.first + 1]++;
```

```
            }
            for (int i = 1; i <= n; i++) {
                start[i] += start[i - 1];
            }
            auto counter = start;
            for (auto e : edges) {
                elist[counter[e.first]++] = e.second;
            }
        }
    };

    struct scc_graph {
    public:
        explicit scc_graph(int n) : _n(n) {}

        int num_vertices() { return _n; }

        void add_edge(int from, int to) { edges.push_back({ from, {to} }); }

        // @return pair of (# of scc, scc id)
        std::pair<int, std::vector<int>> scc_ids() {
            auto g = csr<edge>(_n, edges);
            int now_ord = 0, group_num = 0;
            std::vector<int> visited, low(_n), ord(_n, -1), ids(_n);
            visited.reserve(_n);
            auto dfs = [&](auto self, int v) -> void {
                low[v] = ord[v] = now_ord++;
                visited.push_back(v);
                for (int i = g.start[v]; i < g.start[v + 1]; i++) {
                    auto to = g.elist[i].to;
                    if (ord[to] == -1) {
                        self(self, to);
                        low[v] = std::min(low[v], low[to]);
                    } else {
                        low[v] = std::min(low[v], ord[to]);
                    }
                }
                if (low[v] == ord[v]) {
                    while (true) {
                        int u = visited.back();
                        visited.pop_back();
                        ord[u] = _n;
                        ids[u] = group_num;
```

```cpp
                        if (u == v) break;
                    }
                    group_num++;
                }
            };
            for (int i = 0; i < _n; i++) {
                if (ord[i] == -1) dfs(dfs, i);
            }
            for (auto& x : ids) {
                x = group_num - 1 - x;
            }
            return { group_num, ids };
        }

        // O(N + M)
        // It returns the list of the SCC in topological order.
        std::vector<std::vector<int>> scc() {
            auto ids = scc_ids();
            int group_num = ids.first;
            std::vector<int> counts(group_num);
            for (auto x : ids.second) counts[x]++;
            std::vector<std::vector<int>> groups(ids.first);
            for (int i = 0; i < group_num; i++) {
                groups[i].reserve(counts[i]);
            }
            for (int i = 0; i < _n; i++) {
                groups[ids.second[i]].push_back(i);
            }
            return groups;
        }

    private:
        int _n;
        struct edge {
            int to;
        };
        std::vector<std::pair<int, edge>> edges;
    };
}
```

6.5 最大流

```cpp
template <class T> struct simple_queue {
```

```
    std::vector<T> payload;
    int pos = 0;
    void reserve(int n) { payload.reserve(n); }
    int size() const { return int(payload.size()) - pos; }
    bool empty() const { return pos == int(payload.size()); }
    void push(const T& t) { payload.push_back(t); }
    T& front() { return payload[pos]; }
    void clear() {
        payload.clear();
        pos = 0;
    }
    void pop() { pos++; }
};

template <class Cap> struct mf_graph {
public:
    mf_graph() : _n(0) {}
    mf_graph(int n) : _n(n), g(n) {}

    // returns an integer k such that this is the k-th edge that is added.
    int add_edge(int from, int to, Cap cap) {
        assert(0 <= from && from < _n);
        assert(0 <= to && to < _n);
        assert(0 <= cap);
        int m = int(pos.size());
        pos.push_back({ from, int(g[from].size()) });
        int from_id = int(g[from].size());
        int to_id = int(g[to].size());
        if (from == to) to_id++;
        g[from].push_back(_edge{ to, to_id, cap });
        g[to].push_back(_edge{ from, from_id, 0 });
        return m;
    }

    struct edge {
        int from, to;
        Cap cap, flow;
    };

    edge get_edge(int i) {
        int m = int(pos.size());
        assert(0 <= i && i < m);
        auto _e = g[pos[i].first][pos[i].second];
```

```cpp
        auto _re = g[_e.to][_e.rev];
        return edge{ pos[i].first, _e.to, _e.cap + _re.cap, _re.cap };
    }
    std::vector<edge> edges() {
        int m = int(pos.size());
        std::vector<edge> result;
        for (int i = 0; i < m; i++) {
            result.push_back(get_edge(i));
        }
        return result;
    }
    void change_edge(int i, Cap new_cap, Cap new_flow) {
        int m = int(pos.size());
        assert(0 <= i && i < m);
        assert(0 <= new_flow && new_flow <= new_cap);
        auto& _e = g[pos[i].first][pos[i].second];
        auto& _re = g[_e.to][_e.rev];
        _e.cap = new_cap - new_flow;
        _re.cap = new_flow;
    }

    // max flow from s to t
    // O(M*N^2) general
    // O(min(M*N^2/3, M^3/2)) if capacities of edges are 1
    Cap flow(int s, int t) {
        return flow(s, t, std::numeric_limits<Cap>::max());
    }
    Cap flow(int s, int t, Cap flow_limit) {
        assert(0 <= s && s < _n);
        assert(0 <= t && t < _n);
        assert(s != t);

        std::vector<int> level(_n), iter(_n);
        simple_queue<int> que;

        auto bfs = [&]() {
            std::fill(level.begin(), level.end(), -1);
            level[s] = 0;
            que.clear();
            que.push(s);
            while (!que.empty()) {
                int v = que.front();
                que.pop();
```

```
                for (auto e : g[v]) {
                    if (e.cap == 0 || level[e.to] >= 0) continue;
                    level[e.to] = level[v] + 1;
                    if (e.to == t) return;
                    que.push(e.to);
                }
            }
        };
        auto dfs = [&](auto self, int v, Cap up) {
            if (v == s) return up;
            Cap res = 0;
            int level_v = level[v];
            for (int& i = iter[v]; i < int(g[v].size()); i++) {
                _edge& e = g[v][i];
                if (level_v <= level[e.to] || g[e.to][e.rev].cap == 0) continue;
                Cap d =
                    self(self, e.to, std::min(up - res, g[e.to][e.rev].cap));
                if (d <= 0) continue;
                g[v][i].cap += d;
                g[e.to][e.rev].cap -= d;
                res += d;
                if (res == up) break;
            }
            return res;
        };

        Cap flow = 0;
        while (flow < flow_limit) {
            bfs();
            if (level[t] == -1) break;
            std::fill(iter.begin(), iter.end(), 0);
            while (flow < flow_limit) {
                Cap f = dfs(dfs, t, flow_limit - flow);
                if (!f) break;
                flow += f;
            }
        }
        return flow;
    }

    std::vector<bool> min_cut(int s) {
```

```
        std::vector<bool> visited(_n);
        simple_queue<int> que;
        que.push(s);
        while (!que.empty()) {
            int p = que.front();
            que.pop();
            visited[p] = true;
            for (auto e : g[p]) {
                if (e.cap && !visited[e.to]) {
                    visited[e.to] = true;
                    que.push(e.to);
                }
            }
        }
        return visited;
    }

private:
    int _n;
    struct _edge {
        int to, rev;
        Cap cap;
    };
    std::vector<std::pair<int, int>> pos;
    std::vector<std::vector<_edge>> g;
};
```

6.6 最小费用最大流

```
namespace MCMF {
    template <class T> struct simple_queue {
        std::vector<T> payload;
        int pos = 0;
        void reserve(int n) { payload.reserve(n); }
        int size() const { return int(payload.size()) - pos; }
        bool empty() const { return pos == int(payload.size()); }
        void push(const T& t) { payload.push_back(t); }
        T& front() { return payload[pos]; }
        void clear() {
            payload.clear();
            pos = 0;
        }
        void pop() { pos++; }
```

```
    };

    template <class E> struct csr {
        std::vector<int> start;
        std::vector<E> elist;
        explicit csr(int n, const std::vector<std::pair<int, E>>& edges)
            : start(n + 1), elist(edges.size()) {
            for (auto e : edges) {
                start[e.first + 1]++;
            }
            for (int i = 1; i <= n; i++) {
                start[i] += start[i - 1];
            }
            auto counter = start;
            for (auto e : edges) {
                elist[counter[e.first]++] = e.second;
            }
        }
    };

    template <class Cap, class Cost> struct mcf_graph {
    public:
        mcf_graph() {}
        explicit mcf_graph(int n) : _n(n) {}

        int add_edge(int from, int to, Cap cap, Cost cost) {
            assert(0 <= from && from < _n);
            assert(0 <= to && to < _n);
            assert(0 <= cap);
            assert(0 <= cost);
            int m = int(_edges.size());
            _edges.push_back({ from, to, cap, 0, cost });
            return m;
        }

        struct edge {
            int from, to;
            Cap cap, flow;
            Cost cost;
        };

        edge get_edge(int i) {
            int m = int(_edges.size());
```

```
        assert(0 <= i && i < m);
        return _edges[i];
    }
    std::vector<edge> edges() { return _edges; }

    std::pair<Cap, Cost> flow(int s, int t) {
        return flow(s, t, std::numeric_limits<Cap>::max());
    }
    std::pair<Cap, Cost> flow(int s, int t, Cap flow_limit) {
        return slope(s, t, flow_limit).back();
    }
    std::vector<std::pair<Cap, Cost>> slope(int s, int t) {
        return slope(s, t, std::numeric_limits<Cap>::max());
    }
    std::vector<std::pair<Cap, Cost>> slope(int s, int t, Cap flow_limit) {
        assert(0 <= s && s < _n);
        assert(0 <= t && t < _n);
        assert(s != t);

        int m = int(_edges.size());
        std::vector<int> edge_idx(m);

        auto g = [&]() {
            std::vector<int> degree(_n), redge_idx(m);
            std::vector<std::pair<int, _edge>> elist;
            elist.reserve(2 * m);
            for (int i = 0; i < m; i++) {
                auto e = _edges[i];
                edge_idx[i] = degree[e.from]++;
                redge_idx[i] = degree[e.to]++;
                elist.push_back({ e.from, {e.to, -1, e.cap - e.flow, e.cost} });
                elist.push_back({ e.to, {e.from, -1, e.flow, -e.cost} });
            }
            auto _g = csr<_edge>(_n, elist);
            for (int i = 0; i < m; i++) {
                auto e = _edges[i];
                edge_idx[i] += _g.start[e.from];
                redge_idx[i] += _g.start[e.to];
                _g.elist[edge_idx[i]].rev = redge_idx[i];
                _g.elist[redge_idx[i]].rev = edge_idx[i];
```

```
                }
                return _g;
            }();

            auto result = slope(g, s, t, flow_limit);

            for (int i = 0; i < m; i++) {
                auto e = g.elist[edge_idx[i]];
                _edges[i].flow = _edges[i].cap - e.cap;
            }

            return result;
        }

    private:
        int _n;
        std::vector<edge> _edges;

        // inside edge
        struct _edge {
            int to, rev;
            Cap cap;
            Cost cost;
        };

        std::vector<std::pair<Cap, Cost>> slope(csr<_edge>& g,
            int s,
            int t,
            Cap flow_limit) {
            // variants (C = maxcost):
            // -(n-1)C <= dual[s] <= dual[i] <= dual[t] = 0
            // reduced cost (= e.cost + dual[e.from] - dual[e.to]) >= 0 for
               all edge

            // dual_dist[i] = (dual[i], dist[i])
            std::vector<std::pair<Cost, Cost>> dual_dist(_n);
            std::vector<int> prev_e(_n);
            std::vector<bool> vis(_n);
            struct Q {
                Cost key;
                int to;
                bool operator<(Q r) const { return key > r.key; }
            };
```

```
        std::vector<int> que_min;
        std::vector<Q> que;
        auto dual_ref = [&]() {
            for (int i = 0; i < _n; i++) {
                dual_dist[i].second = std::numeric_limits<Cost>::max();
            }
            std::fill(vis.begin(), vis.end(), false);
            que_min.clear();
            que.clear();

            // que[0..heap_r) was heapified
            size_t heap_r = 0;

            dual_dist[s].second = 0;
            que_min.push_back(s);
            while (!que_min.empty() || !que.empty()) {
                int v;
                if (!que_min.empty()) {
                    v = que_min.back();
                    que_min.pop_back();
                } else {
                    while (heap_r < que.size()) {
                        heap_r++;
                        std::push_heap(que.begin(), que.begin() + heap_r
                            );
                    }
                    v = que.front().to;
                    std::pop_heap(que.begin(), que.end());
                    que.pop_back();
                    heap_r--;
                }
                if (vis[v]) continue;
                vis[v] = true;
                if (v == t) break;
                // dist[v] = shortest(s, v) + dual[s] - dual[v]
                // dist[v] >= 0 (all reduced cost are positive)
                // dist[v] <= (n-1)C
                Cost dual_v = dual_dist[v].first, dist_v = dual_dist[v].
                    second;
                for (int i = g.start[v]; i < g.start[v + 1]; i++) {
                    auto e = g.elist[i];
                    if (!e.cap) continue;
                    // |-dual[e.to] + dual[v]| <= (n-1)C
```

```cpp
                    // cost <= C - -(n-1)C + 0 = nC
                    Cost cost = e.cost - dual_dist[e.to].first + dual_v;
                    if (dual_dist[e.to].second - dist_v > cost) {
                        Cost dist_to = dist_v + cost;
                        dual_dist[e.to].second = dist_to;
                        prev_e[e.to] = e.rev;
                        if (dist_to == dist_v) {
                            que_min.push_back(e.to);
                        } else {
                            que.push_back(Q{ dist_to, e.to });
                        }
                    }
                }
            }
            if (!vis[t]) {
                return false;
            }

            for (int v = 0; v < _n; v++) {
                if (!vis[v]) continue;
                // dual[v] = dual[v] - dist[t] + dist[v]
                //         = dual[v] - (shortest(s, t) + dual[s] - dual[t]) +
                //           (shortest(s, v) + dual[s] - dual[v]) = -shortest(s,
                //           t) + dual[t] + shortest(s, v) = shortest(s, v) -
                //           shortest(s, t) >= 0 - (n-1)C
                dual_dist[v].first -= dual_dist[t].second - dual_dist[v].second;
            }
            return true;
        };
        Cap flow = 0;
        Cost cost = 0, prev_cost_per_flow = -1;
        std::vector<std::pair<Cap, Cost>> result = { {Cap(0), Cost(0)} };
        while (flow < flow_limit) {
            if (!dual_ref()) break;
            Cap c = flow_limit - flow;
            for (int v = t; v != s; v = g.elist[prev_e[v]].to) {
                c = std::min(c, g.elist[g.elist[prev_e[v]].rev].cap);
            }
```

```
                for (int v = t; v != s; v = g.elist[prev_e[v]].to) {
                    auto& e = g.elist[prev_e[v]];
                    e.cap += c;
                    g.elist[e.rev].cap -= c;
                }
                Cost d = -dual_dist[s].first;
                flow += c;
                cost += c * d;
                if (prev_cost_per_flow == d) {
                    result.pop_back();
                }
                result.push_back({ flow, cost });
                prev_cost_per_flow = d;
            }
            return result;
        }
    };
}
```

6.7 全局最小割

6.8 二分图最大权匹配

```
namespace KM {
    typedef long long ll;
    const int maxn = 510;
    const int inf = 1e9;
    int vx[maxn], vy[maxn], lx[maxn], ly[maxn], slack[maxn];
    int w[maxn][maxn]; // 以上为权值类型
    int pre[maxn], left[maxn], right[maxn], NL, NR, N;
    void match(int& u) {
        for (; u; std::swap(u, right[pre[u]]))
            left[u] = pre[u];
    }
    void bfs(int u) {
        static int q[maxn], front, rear;
        front = 0; vx[q[rear = 1] = u] = true;
        while (true) {
            while (front < rear) {
                int u = q[++front];
                for (int v = 1; v <= N; ++v) {
                    int tmp;
                    if (vy[v] || (tmp = lx[u] + ly[v] - w[u][v]) > slack[v])
                        continue;
```

```
                    pre[v] = u;
                    if (!tmp) {
                        if (!left[v]) return match(v);
                        vy[v] = vx[q[++rear]] = left[v]] = true;
                    } else slack[v] = tmp;
                }
            }
            int a = inf;
            for (int i = 1; i <= N; ++i)
                if (!vy[i] && a > slack[i]) a = slack[u = i];
            for (int i = 1; i <= N; ++i) {
                if (vx[i]) lx[i] -= a;
                if (vy[i]) ly[i] += a;
                else slack[i] -= a;
            }
            if (!left[u]) return match(u);
            vy[u] = vx[q[++rear]] = left[u]] = true;

        }

    }
    void exec() {
        for (int i = 1; i <= N; ++i) {
            for (int j = 1; j <= N; ++j) {
                slack[j] = inf;
                vx[j] = vy[j] = false;
            }
            bfs(i);
        }
    }
    ll work(int nl, int nr) { // NL , NR 为左右点数, 返回最大权匹配的权值和
        NL = nl; NR = nr;
        N = std::max(NL, NR);
        for (int u = 1; u <= N; ++u)
            for (int v = 1; v <= N; ++v)
                lx[u] = std::max(lx[u], w[u][v]);
        exec();
        ll ans = 0;
        for (int i = 1; i <= N; ++i)
            ans += lx[i] + ly[i];
        return ans;
    }
    void output() { // 输出左边点与右边哪个点匹配, 没有匹配输出0
```

```
        for (int i = 1; i <= NL; ++i)
            printf("%d ", (w[i][right[i]] ? right[i] : 0));
        printf("\n");
    }
}
```

6.9 一般图最大匹配

6.10 2-sat

6.11 最大团

```
/*
 * 最大团 Bron-Kerbosch algorithm
 * 最劣复杂度 O(3^(n/3))
 * 采用位运算形式实现
 * */
namespace Max_clique {
#define ll long long
#define TWOL(x) (1ll<<(x))
    const int N = 60;
    int n, m;        // 点数 边数
    int r = 0;       // 最大团大小
    ll G[N];         // 以二进制形式存图
    ll clique = 0; // 最大团 以二进制形式存储
    void BronK(int S, ll P, ll X, ll R) { // 调用时参数这样设置: 0, TWOL(n)
        -1, 0, 0
        if (P == 0 && X == 0) {
            if (r < S) {
                r = S;
                clique = R;
            }
        }
        if (P == 0) return;
        int u = __builtin_ctzll(P | X);
        ll c = P & ~G[u];
        while (c) {
            int v = __builtin_ctzll(c);
            ll pv = TWOL(v);
            BronK(S + 1, P & G[v], X & G[v], R | pv);
            P ^= pv; X |= pv; c ^= pv;
        }
    }
    void init() {
```

```
        cin >> n >> m;
        for (int i = 0; i < m; i++) {
            int u, v;
            cin >> u >> v;
            --u, --v;
            G[u] |= TWOL(v);
            G[v] |= TWOL(u);
        }
        BronK(0, TWOL(n)-1, 0, 0);
        cout << r << ' ' << clique << '\n';
    }
}
```

7 数据结构

7.1 树状数组

8 字符串

8.1 KMP

```
namespace KMP {
    vector<int> getPrefixTable(string s) { // 求前缀表
        int n = s.length();
        vector<int> nxt(n, 0);
        for (int i = 1; i < n; i++) {
            int j = nxt[i - 1];
            while (j > 0 && s[i] != s[j]) {
                j = nxt[j - 1];
            }
            if (s[i] == s[j]) j++;
            nxt[i] = j;
        }
        return nxt;
    }

    vector<int> kmp(string s, string t) { // 返回所有匹配位置的集合
        int n = s.length(), m = t.length();
        vector<int> res;
        vector<int> nxt = getPrefixTable(t);
        for (int i = 0, j = 0; i < n; i++) {
            while (j > 0 && j < m && s[i] != t[j]) {
                j = nxt[j - 1];
```

```
            }
            if (s[i] == t[j]) j++;
            if (j == m) {
                res.push_back(i + 1 - m);
                j = nxt[m - 1];
            }
        }
        return res;
    }
}
```

8.2 Z-Function

8.3 Manacher

```
namespace Manacher {
    static constexpr int SIZE = 1e5 + 5; // 预设为原串长度
    int len = 1; // manacher 预处理后字符串的长度
    char stk[SIZE << 1]; // manacher预处理字符串 需要2倍空间+1
    void init(string s) { // 初始化stk
        stk[0] = '*'; len = 1;
        for (int i = 0; i < s.length(); ++i) {
            stk[len++] = s[i];
            stk[len++] = '*';
        }
    }
    int manacher() { // 返回最长回文子串长度
        vector<int> rad(len << 1); // 存储每个点作为对称中心可拓展的最大半径
        int md = 0; // 最远回文串对称中心下标
        for (int i = 1; i < len; ++i) {
            int& r = rad[i] = 0;
            if (i <= md + rad[md]) {
                r = min(rad[2 * md - i], md + rad[md] - i);
            }
            while (i - r - 1 >= 0 && i + r + 1 < len &&
                stk[i - r - 1] == stk[i + r + 1]) ++r;
            if (i + r >= md + rad[md]) md = i;
        }
        int res = 0;
        for (int i = 0; i < len; ++i) {
            if (rad[i] > res) {
                res = rad[i];
            }
        }
```

```
        return res;
    }
}
```

8.4 Trie

```
struct trie {
    int cnt;
    vector<vector<int>> nxt;
    vector<bool> vis;
    /* 初始化的时候size需要设置为字符串总长之和 26是字符集大小 */
    trie(int size_ = 0) :cnt(0), vis(size_, false), nxt(size_, vector<int
        >(26, 0)) {}
    void insert(string s) {  // 插入字符串
        int p = 0;
        for (int i = 0; i < (int)s.length(); i++) {
            int c = s[i] - 'a';
            if (!nxt[p][c]) nxt[p][c] = ++cnt;
            p = nxt[p][c];
        }
        vis[p] = true;
    }
    bool find(string s) {  // 查找字符串
        int p = 0;
        for (int i = 0; i < (int)s.length(); i++) {
            int c = s[i] - 'a';
            if (!nxt[p][c]) return false;
            p = nxt[p][c];
        }
        return vis[p];
    }
};
```

8.5 01-Trie

```
template<typename T> struct xorTrie {
    int HIGHBIT, cnt;
    vector<vector<int>> nxt;
    vector<bool> vis;
    xorTrie(int n_ = 0, int highbit_ = 30) : HIGHBIT(highbit_), cnt(0) {
        int size_ = upperBoundEstimate(n_);
        nxt.resize(size_, vector<int>(2, 0));
        vis.resize(size_, false);
```

```
    }
    int upperBoundEstimate(int n) { // 求内存上界
        int hbit = log2(n);
        return n * (HIGHBIT - hbit + 1) + (1 << (hbit + 1)) - 1;
    }
    void insert(T x) { // 插入
        int p = 0;
        for (int i = HIGHBIT; ~i; i--) {
            int s = ((x >> i) & 1);
            if (!nxt[p][s]) nxt[p][s] = ++cnt;
            p = nxt[p][s];
        }
        vis[p] = true;
    }
    bool find(T x) { // 查询
        int p = 0;
        for (int i = HIGHBIT; ~i; i--) {
            int s = ((x >> i) & 1);
            if (!nxt[p][s]) return false;
            p = nxt[p][s];
        }
        return vis[p];
    }
};
```

9 计算几何

```
namespace Geometry {
#define db long double
#define pi acos(-1.0)
    constexpr db eps = 1e-7;
    int sign(db k) {
        if (k > eps) return 1;
        else if (k < -eps) return -1;
        return 0;
    }
    int cmp(db k1, db k2) { // k1 < k2 : -1, k1 == k2 : 0, k1 > k2 : 1
        return sign(k1 - k2);
    }
    int inmid(db k1, db k2, db k3) { // k3 在 [k1, k2] 内
        return sign(k1 - k3) * sign(k2 - k3) <= 0;
    }

```

```
struct point { // 点类
    db x, y;
    point() {}
    point(db x_, db y_) :x(x_), y(y_) {}
    point operator + (const point& k) const { return point(k.x + x, k.y
        + y); }
    point operator - (const point& k) const { return point(x - k.x, y -
        k.y); }
    point operator * (db k) const { return point(x * k, y * k); }
    point operator / (db k1) const { return point(x / k1, y / k1); }
    point turn(db k1) { return point(x * cos(k1) - y * sin(k1), x * sin(
        k1) + y * cos(k1)); } // 逆时针旋转
    point turn90() { return point(-y, x); } // 逆时针方向旋转 90 度
    db len() { return sqrt(x * x + y * y); } // 向量长度
    db len2() { return x * x + y * y; } // 向量长度的平方
    db getPolarAngle() { return atan2(y, x); } // 向量极角
    db dis(point k) { return ((*this) - k).len(); } // 到点k的距离
    point unit() { db d = len(); return point(x / d, y / d); } // 单位向
        量
    point getdel() { // 将向量的方向调整为指向第一/四象限 包括y轴正方向
        if (sign(x) == -1 || (sign(x) == 0 && sign(y) == -1))
            return (*this) * (-1);
        else return (*this);
    }
    bool operator < (const point& k) const { // 水平序排序 x坐标为第一关
        键字,y坐标第二关键字
        return x == k.x ? y < k.y : x < k.x;
    }
    bool operator == (const point& k) const { return cmp(x, k.x) == 0 &&
        cmp(y, k.y) == 0; }
    bool getP() const { // 判断点是否在上半平面 含x负半轴 不含x正半轴及
        零点
        return sign(y) == 1 || (sign(y) == 0 && sign(x) == -1);
    }
    void input() { cin >> x >> y; }
};
db cross(point k1, point k2) { return k1.x * k2.y - k1.y * k2.x; } // 向
    量 k1,k2 的叉积
db dot(point k1, point k2) { return k1.x * k2.x + k1.y * k2.y; }   // 向
    量 k1,k2 的点积
db rad(point k1, point k2) { // 向量 k1,k2 之间的有向夹角
    return atan2(cross(k1, k2), dot(k1, k2));
}
```

```
    int inmid(point k1, point k2, point k3) { // k1 k2 k3共线时 判断点 k3 是否在线段 k1k2 上
        return inmid(k1.x, k2.x, k3.x) && inmid(k1.y, k2.y, k3.y);
    }
    int compareAngle(point k1, point k2) { // 比较向量 k1,k2 的角度大小 角度按照atan2()函数定义
        // k1 < k2 返回 1, k1 >= k2 返回 0
        return k1.getP() < k2.getP() || (k1.getP() == k2.getP() && sign(cross(k1, k2)) > 0);
    }
    point proj(point k1, point k2, point q) { // q 到直线 k1,k2 的投影
        point k = k2 - k1; return k1 + k * (dot(q - k1, k) / k.len2());
    }
    point reflect(point k1, point k2, point q) { return proj(k1, k2, q) * 2 - q; } // q 关于直线 k1,k2 的对称点
    int counterclockwise(point k1, point k2, point k3) { // k1 k2 k3 逆时针1 顺时针-1 否则0
        return sign(cross(k2 - k1, k3 - k1));
    }
    int checkLL(point k1, point k2, point k3, point k4) { // 判断直线 k1k2 和直线k3k4 是否相交
        // 即判断直线 k1k2 和 k3k4 是否平行 平行返回0 不平行返回1
        return sign(cross(k2 - k1, k4 - k3)) != 0;
    }
    point getLL(point k1, point k2, point k3, point k4) { // 求 k1k2 k3k4 两直线交点
        db w1 = cross(k1 - k3, k4 - k3), w2 = cross(k4 - k3, k2 - k3);
        return (k1 * w2 + k2 * w1) / (w1 + w2);
    }
    int intersect(db l1, db r1, db l2, db r2) { // 判断 [l1,r1] 和 [l2, r2] 是否相交
        if (l1 > r1) swap(l1, r1);
        if (l2 > r2) swap(l2, r2);
        return cmp(r1, l2) != -1 && cmp(r2, l1) != -1;
    }
    int checkSS(point k1, point k2, point k3, point k4) { // 判断线段 k1k2 和线段 k3k4 是否相交
        return intersect(k1.x, k2.x, k3.x, k4.x) && intersect(k1.y, k2.y, k3.y, k4.y) &&
            sign(cross(k3 - k1, k4 - k1)) * sign(cross(k3 - k2, k4 - k2)) <= 0 &&
            sign(cross(k1 - k3, k2 - k3)) * sign(cross(k1 - k4, k2 - k4)) <= 0;
```

```
    }
    db disSP(point k1, point k2, point q) { // 点 q 到线段 k1k2 的最短距离
        point k3 = proj(k1, k2, q);
        if (inmid(k1, k2, k3)) return q.dis(k3);
        else return min(q.dis(k1), q.dis(k2));
    }
    db disLP(point k1, point k2, point q) { // 点 q 到直线 k1k2 的最短距离
        point k3 = proj(k1, k2, q);
        return q.dis(k3);
    }
    db disSS(point k1, point k2, point k3, point k4) { // 线段 k1k2 和线段 k3k4 的最短距离
        if (checkSS(k1, k2, k3, k4)) return 0;
        else return min(min(disSP(k1, k2, k3), disSP(k1, k2, k4)),
            min(disSP(k3, k4, k1), disSP(k3, k4, k2)));
    }
    bool onLine(point k1, point k2, point q) { // 判断点 q 是否在直线 k1k2 上
        return sign(cross(k1 - q, k2 - q)) == 0;
    }
    bool onSegment(point k1, point k2, point q) { // 判断点 q 是否在线段 k1k2 上
        if (!onLine(k1, k2, q)) return false; // 如果确定共线 要删除这个特判
        return inmid(k1, k2, q);
    }
    void polarAngleSort(vector<point>& p, point t) { // p为待排序点集 t为极角排序中心
        sort(p.begin(), p.end(), [&](const point& k1, const point& k2) {
            return compareAngle(k1 - t, k2 - t);
        });
    }

    struct line { // 直线 / 线段类
        point p[2];
        line() {}
        line(point k1, point k2) { p[0] = k1, p[1] = k2; }
        point& operator [] (int k) { return p[k]; }
        point dir() { return p[1] - p[0]; } // 向量 p[0] -> p[1]
        bool include(point k) { // 判断点是否在直线上
            return sign(cross(p[1] - p[0], k - p[0])) > 0;
        }
        bool includeS(point k) { // 判断点是否在线段上
            return onSegment(p[0], p[1], k);
```

```
        }
        line push(db len) { // 向外（左手边）平移 len 个单位
            point delta = (p[1] - p[0]).turn90().unit() * len;
            return line(p[0] - delta, p[1] - delta);
        }
    };
    bool parallel(line k1, line k2) { // 判断是否平行
        return sign(cross(k1.dir(), k2.dir())) == 0;
    }
    bool sameLine(line k1, line k2) { // 判断是否共线
        return parallel(k1, k2) && parallel(k1, line(k2.p[0], k1.p[0]));
    }
    bool sameDir(line k1, line k2) { // 判断向量 k1 k2 是否同向
        return parallel(k1, k2) && sign(dot(k1.dir(), k2.dir())) == 1;
    }
    bool operator < (line k1, line k2) {
        if (sameDir(k1, k2)) return k2.include(k1[0]);
        return compareAngle(k1.dir(), k2.dir());
    }
    bool checkLL(line k1, line k2) {
        return checkLL(k1[0], k1[1], k2[0], k2[1]);
    }
    point getLL(line k1, line k2) {  // 求 k1 k2 两直线交点 不要忘了判平行!
        return getLL(k1[0], k1[1], k2[0], k2[1]);
    }
    bool checkpos(line k1, line k2, line k3) {  // 判断是否三线共点
        return k3.include(getLL(k1, k2));
    }

    struct circle { // 圆类
        point o;
        double r;
        circle() {}
        circle(point o_, double r_) : o(o_), r(r_) {}
        int inside(point k) {  // 判断点 k 和圆的位置关系
            return cmp(r, o.dis(k)); // 圆外:-1, 圆上:0, 圆内:1
        }
    };
    int checkposCC(circle k1, circle k2) { // 返回两个圆的公切线数量
        if (cmp(k1.r, k2.r) == -1) swap(k1, k2);
        db dis = k1.o.dis(k2.o);
        int w1 = cmp(dis, k1.r + k2.r), w2 = cmp(dis, k1.r - k2.r);
        if (w1 > 0) return 4; // 外离
```

```
        else if (w1 == 0) return 3; // 外切
        else if (w2 > 0) return 2;  // 相交
        else if (w2 == 0) return 1; // 内切
        else return 0; // 内离(包含)
    }
    vector<point> getCL(circle k1, point k2, point k3) { // 求直线 k2k3 和圆 k1 的交点
        // 沿着 k2->k3 方向给出 相切给出两个
        point k = proj(k2, k3, k1.o);
        db d = k1.r * k1.r - (k - k1.o).len2();
        if (sign(d) == -1) return {};
        point del = (k3 - k2).unit() * sqrt(max((db)0.0, d));
        return { k - del,k + del };
    }
    vector<point> getCC(circle k1, circle k2) { // 求圆 k1 和圆 k2 的交点
        // 沿圆 k1 逆时针给出, 相切给出两个
        int pd = checkposCC(k1, k2); if (pd == 0 || pd == 4) return {};
        db a = (k2.o - k1.o).len2(), cosA = (k1.r * k1.r + a -
            k2.r * k2.r) / (2 * k1.r * sqrt(max(a, (db)0.0)));
        db b = k1.r * cosA, c = sqrt(max((db)0.0, k1.r * k1.r - b * b));
        point k = (k2.o - k1.o).unit(), m = k1.o + k * b, del = k.turn90() *
            c;
        return { m - del,m + del };
    }
    vector<point> tangentCP(circle k1, point k2) { // 点 k2 到圆 k1 的切点 沿圆 k1 逆时针给出
        db a = (k2 - k1.o).len(), b = k1.r * k1.r / a, c = sqrt(max((db)0.0,
            k1.r * k1.r - b * b));
        point k = (k2 - k1.o).unit(), m = k1.o + k * b, del = k.turn90() * c
            ;
        return { m - del,m + del };
    }
    vector<line> tangentOutCC(circle k1, circle k2) {
        int pd = checkposCC(k1, k2);
        if (pd == 0) return {};
        if (pd == 1) {
            point k = getCC(k1, k2)[0];
            return { line(k,k) };
        }
        if (cmp(k1.r, k2.r) == 0) {
            point del = (k2.o - k1.o).unit().turn90().getdel();
            return { line(k1.o - del * k1.r,k2.o - del * k2.r),
                line(k1.o + del * k1.r,k2.o + del * k2.r) };
```

```
        } else {
            point p = (k2.o * k1.r - k1.o * k2.r) / (k1.r - k2.r);
            vector<point> A = tangentCP(k1, p), B = tangentCP(k2, p);
            vector<line> ans; for (int i = 0; i < A.size(); i++)
                ans.push_back(line(A[i], B[i]));
            return ans;
        }
    }
    vector<line> tangentInCC(circle k1, circle k2) {
        int pd = checkposCC(k1, k2);
        if (pd <= 2) return {};
        if (pd == 3) {
            point k = getCC(k1, k2)[0];
            return { line(k, k) };
        }
        point p = (k2.o * k1.r + k1.o * k2.r) / (k1.r + k2.r);
        vector<point> A = tangentCP(k1, p), B = tangentCP(k2, p);
        vector<line> ans;
        for (int i = 0; i < (int)A.size(); i++) ans.push_back(line(A[i], B[i
            ]));
        return ans;
    }
    vector<line> tangentCC(circle k1, circle k2) { // 求两圆公切线
        int flag = 0;
        if (k1.r < k2.r) swap(k1, k2), flag = 1;
        vector<line> A = tangentOutCC(k1, k2), B = tangentInCC(k1, k2);
        for (line k : B) A.push_back(k);
        if (flag) for (line& k : A) swap(k[0], k[1]);
        return A;
    }
    db getAreaCT(circle k1, point k2, point k3) { // 圆 k1 与三角形 k2k3k1.o
        的有向面积交
        point k = k1.o; k1.o = k1.o - k; k2 = k2 - k; k3 = k3 - k;
        int pd1 = k1.inside(k2), pd2 = k1.inside(k3);
        vector<point> A = getCL(k1, k2, k3);
        if (pd1 >= 0) {
            if (pd2 >= 0) return cross(k2, k3) / 2;
            return k1.r * k1.r * rad(A[1], k3) / 2 + cross(k2, A[1]) / 2;
        } else if (pd2 >= 0) {
            return k1.r * k1.r * rad(k2, A[0]) / 2 + cross(A[0], k3) / 2;
        } else {
            int pd = cmp(k1.r, disSP(k2, k3, k1.o));
            if (pd <= 0) return k1.r * k1.r * rad(k2, k3) / 2;
```

```
            return cross(A[0], A[1]) / 2 + k1.r * k1.r * (rad(k2, A[0]) +
                rad(A[1], k3)) / 2;
        }
    }
    db getAreaCC(circle k1, circle k2) { // 两圆面积交
        db d = k1.o.dis(k2.o);
        if (cmp(d, k1.r + k2.r) >= 0) return 0; // 两圆相离
        if (cmp(k1.r, k2.r) == -1) swap(k1, k2);
        if (cmp(k1.r - k2.r, d) >= 0) return pi * k2.r * k2.r; // 圆k1包含k2
        db g1 = acos((k1.r * k1.r + d * d - k2.r * k2.r) / (2 * k1.r * d));
        db g2 = acos((k2.r * k2.r + d * d - k1.r * k1.r) / (2 * k2.r * d));
        return g1 * k1.r * k1.r + g2 * k2.r * k2.r - k1.r * d * sin(g1);
    }
    circle getCircleOut(point k1, point k2, point k3) { // 三角形外接圆
        db a1 = k2.x - k1.x, b1 = k2.y - k1.y, c1 = (a1 * a1 + b1 * b1) / 2;
        db a2 = k3.x - k1.x, b2 = k3.y - k1.y, c2 = (a2 * a2 + b2 * b2) / 2;
        db d = a1 * b2 - a2 * b1;
        point o(k1.x + (c1 * b2 - c2 * b1) / d, k1.y + (a1 * c2 - a2 * c1) /
            d);
        return circle(o, k1.dis(o));
    }
    circle getCircleIn(point k1, point k2, point k3) {  // 三角形内切圆
        db a = k1.dis(k2), b = k2.dis(k3), c = k3.dis(k1);
        db len = a + b + c;
        db r = abs(cross(k1 - k2, k1 - k3)) / len;
        point o((k1.x * b + k2.x * c + k3.x * a) / len, (k1.y * b + k2.y * c
            + k3.y * a) / len);
        return circle(o, r);
    }
    circle minCircleCovering(vector<point> A) { // 最小圆覆盖 O(n)随机增量法
        // random_shuffle(A.begin(), A.end()); // <= C++14
        auto seed = chrono::steady_clock::now().time_since_epoch().count();
        default_random_engine e(seed);
        shuffle(A.begin(), A.end(), e); // >= C++11
        circle ans = circle(A[0], 0);
        for (int i = 1; i < A.size(); i++) {
            if (ans.inside(A[i]) == -1) {
                ans = circle(A[i], 0);
                for (int j = 0; j < i; j++) {
                    if (ans.inside(A[j]) == -1) {
                        ans.o = (A[i] + A[j]) / 2;
                        ans.r = ans.o.dis(A[i]);
                        for (int k = 0; k < j; k++) {
```

```
                        if (ans.inside(A[k]) == -1)
                            ans = getCircleOut(A[i], A[j], A[k]);
                    }
                }
            }
        }
    }
    return ans;
}

typedef vector<point> polygon;
db area(polygon p) { // 多边形有向面积
    if (p.size() < 3) return 0;
    db ans = 0;
    for (int i = 1; i < p.size() - 1; i++)
        ans += cross(p[i] - p[0], p[i + 1] - p[0]);
    return 0.5L * ans;
}

int checkConvexP(polygon p, point a) { // O(logn)判断点是否在凸包内 2内部 1边界 0外部
    // 必须保证凸多边形是一个水平序凸包且不能退化
    // 退化情况 比如凸包退化成线段 可使用 onSegment() 函数特判
    auto check = [&](int x) {
        int ccw1 = counterclockwise(p[0], a, p[x]),
            ccw2 = counterclockwise(p[0], a, p[x + 1]);
        if (ccw1 == -1 && ccw2 == -1) return 2;
        else if (ccw1 == 1 && ccw2 == 1) return 0;
        else if (ccw1 == -1 && ccw2 == 1) return 1;
        else return 1;
    };
    if (counterclockwise(p[0], a, p[1]) <= 0 && counterclockwise(p[0], a, p.back()) >= 0) {
        int l = 1, r = p.size() - 2, mid;
        while (l <= r) {
            mid = (l + r) >> 1;
            int chk = check(mid);
            if (chk == 1) l = mid + 1;
            else if (chk == -1) r = mid;
            else break;
        }
        int res = counterclockwise(p[mid], a, p[mid + 1]);
        if (res < 0) return 2;
```

```
            else if (res == 0) return 1;
            else return 0;
        } else {
            return 0;
        }
    }
    int checkPolyP(vector<point> p, point q) { // O(n)判断点是否在一般多边形内
        // 必须保证简单多边形的点按逆时针给出 返回 2 内部 1 边界 0 外部
        int pd = 0, n = p.size();
        for (int i = 0; i < n; i++) {
            point u = p[i], v = p[(i + 1) % n];
            if (onSegment(u, v, q)) return 1;
            if (cmp(u.y, v.y) > 0) swap(u, v);
            if (cmp(u.y, q.y) >= 0 || cmp(v.y, q.y) < 0) continue;
            if (sign(cross(u - v, q - v)) < 0) pd ^= 1;
        }
        return pd << 1;
    }
    db convexDiameter(polygon p) { // 0(n)旋转卡壳求凸包直径 / 平面最远点对的平方
        int n = p.size(); // 请保证多边形是凸包
        db ans = 0;
        for (int i = 0, j = n < 2 ? 0 : 1; i < j; i++) {
            for (;; j = (j + 1) % n) {
                ans = max(ans, (p[i] - p[j]).len2());
                if (sign(cross(p[i + 1] - p[i], p[(j + 1) % n] - p[j])) <= 0) break;
            }
        }
        return ans;
    }
    polygon convexHull(polygon A, int flag = 1) { // 凸包 flag=0 不严格 flag=1 严格
        int n = A.size(); polygon ans(n + n);
        sort(A.begin(), A.end()); int now = -1;
        for (int i = 0; i < A.size(); i++) {
            while (now > 0 && sign(cross(ans[now] - ans[now - 1], A[i] - ans[now - 1])) < flag)
                now--;
            ans[++now] = A[i];
        }
        int pre = now;
```

```
        for (int i = n - 2; i >= 0; i--) {
            while (now > pre && sign(cross(ans[now] - ans[now - 1], A[i] -
                ans[now - 1])) < flag)
                now--;
            ans[++now] = A[i];
        }
        ans.resize(now);
        return ans;
    }
    bool checkConvexHull(polygon p) { // 检测多边形是否是凸包（可以有三点共
        线）
        int sgn, n = p.size(), i = 0; // 如果三点共线不算凸包 去掉ccw=0的情
            况
        for (;; i++) { // 这一步是为了防止第一步遇到共线的三个点
            sgn = counterclockwise(p[i], p[(i + 1) % n], p[(i + 2) % n]);
            if (sgn) break;
        }
        for (; i < n; i++) {
            int ccw = counterclockwise(p[i], p[(i + 1) % n], p[(i + 2) % n])
                ;
            if (ccw && ccw != sgn) {
                return false;
            }
        }
        return true;
    }
    polygon convexCut(polygon A, point k1, point k2) { // 半平面 k1k2 切凸包
        A
        int n = A.size();  // 保留所有满足 k1 -> p -> k2 为逆时针方向的点
        A.push_back(A[0]); // 保留的点可能有重点
        polygon ans;
        line cut(k1, k2);
        for (int i = 0; i < n; i++) {
            int ccw1 = counterclockwise(k1, k2, A[i]);
            int ccw2 = counterclockwise(k1, k2, A[i + 1]);
            if (ccw1 >= 0) ans.push_back(A[i]);
            if (ccw1 * ccw2 <= 0) {
                if (sameLine(cut, line(A[i], A[i + 1]))) { // 半平面恰好切到
                    凸包上某条边
                    ans.push_back(A[i]);
                    ans.push_back(A[i + 1]);
                } else {
                    ans.push_back(getLL(k1, k2, A[i], A[i + 1]));
```

```
                }
            }
        }
        return ans;
    }

    vector<line> getHL(vector<line>& L) { // 求半平面交 逆时针方向存储
        sort(L.begin(), L.end());
        deque<line> q;
        for (int i = 0; i < (int)L.size(); ++i) {
            if (i && sameDir(L[i], L[i - 1])) continue;
            while (q.size() > 1 && !checkpos(q[q.size() - 2], q[q.size() - 1], L[i])) q.pop_back();
            while (q.size() > 1 && !checkpos(q[1], q[0], L[i])) q.pop_front();
            q.push_back(L[i]);
        }
        while (q.size() > 2 && !checkpos(q[q.size() - 2], q[q.size() - 1], q[0])) q.pop_back();
        while (q.size() > 2 && !checkpos(q[1], q[0], q[q.size() - 1])) q.pop_front();
        vector<line> ans;
        for (int i = 0; i < q.size(); ++i) ans.push_back(q[i]);
        return ans;
    }

    db closestPoint(vector<point>& A, int l, int r) { // 最近点对, 先要按照 x 坐标排序
        if (r - l <= 5) {
            db ans = 1e20;
            for (int i = l; i <= r; ++i)
                for (int j = i + 1; j <= r; j++)
                    ans = min(ans, A[i].dis(A[j]));
            return ans;
        }
        int mid = l + r >> 1;
        db ans = min(closestPoint(A, l, mid), closestPoint(A, mid + 1, r));
        vector<point> B;
        for (int i = l; i <= r; i++)
            if (abs(A[i].x - A[mid].x) <= ans)
                B.push_back(A[i]);
        sort(B.begin(), B.end(), [&](const point& k1, const point& k2) {
            return k1.y < k2.y;
```

```
        });
        for (int i = 0; i < B.size(); i++)
            for (int j = i + 1; j < B.size() && B[j].y - B[i].y < ans; j++)
                ans = min(ans, B[i].dis(B[j]));
        return ans;
    }
}
using namespace Geometry;
```

10 杂项

10.1 蔡勒公式