hust-yc-template

yincircle

2018 年 11 月 14 日

目录

1 数学

1.1 快速幂

```cpp
LL qpow(LL a,LL b,LL Mod){
    LL ret=1;
    while(b){
        if(b&1)ret=(ret*a)%Mod;
        a=(a*a)%Mod;
        b>>=1;
    }
    return ret;
}
```

1.2 筛

1.2.1 线性筛素数

```cpp
const LL p_max=1e6;
LL prime[p_max+100],p_sz;//用vector存素数能优化时间
void get_prime(){
    bool vis[p_max+100];
    for(int i=2;i<=p_max;i++){
        if(!vis[i])prime[p_sz++]=i;
        for(int j=0;j<p_sz&&prime[j]*i<=p_max;j++){
            vis[prime[j]*i]=1;
            if(i%prime[j]==0)break;
        }
    }
}
```

1.2.2 线性筛欧拉函数

```cpp
const LL p_max=1e6;
LL phi[p_max+100],prime[p_max+100],p_sz=0;
bool vis[p_max+100];
void get_phitable(){
    phi[1]=1;
    for(int i=2;i<=p_max;i++){
        if(!vis[i]){
        prime[p_sz++]=i;
        phi[i]=i-1;
        }
        LL d;
        for(int j=0;j<p_sz&&(d=prime[j]*i)<=p_max;j++){
            vis[d]=1;
            if(i%prime[j]==0){
                phi[d]=phi[i]*prime[j];
                break;
            }
            else phi[d]=phi[i]*(prime[j]-1);
        }
    }
}
```

1.2.3 线性筛莫比乌斯函数

```cpp
const LL p_max=1e6;
LL mu[p_max+100],prime[p_max+100],p_sz;
bool vis[p_max+100];
void get_mutable(){
    mu[1]=1;
    for(int i=2;i<=p_max;i++){
        if(!vis[i]){
            prime[p_sz++]=i;
            mu[i]=-1;
        }
        LL d;
        for(int j=0;j<p_sz&&(d=prime[j]*i)<=p_max;j++){
            vis[d]=1;
            if(i%prime[j]==0){
                mu[d]=0;
                break;
            }
            else mu[d]=-mu[i];
        }
    }
}
```

1.2.4 杜教筛

求 $S(n)=\sum_{i=1}^{n} f(i)$，其中 f 是一个积性函数。

构造一个积性函数 g，那么由 $(f*g)(n)=\sum_{d|n} f(d)g(\frac{n}{d})$，得到 $f(n)=(f*g)(n)-\sum_{d|n,d<n} f(d)g(\frac{n}{d})$。

$$
\begin{aligned}
g(1)S(n) &= \sum_{i=1}^{n}(f*g)(i)-\sum_{i=1}^{n}\sum_{d|i,d<i} f(d)g(\frac{n}{d}) && (1)\\
&\overset{t=\frac{i}{d}}{=} \sum_{i=1}^{n}(f*g)(i)-\sum_{t=2}^{n} g(t)S(\lfloor\frac{n}{t}\rfloor) && (2)
\end{aligned}
$$

当然，要能够由此计算 $S(n)$，会对 f,g 提出一些要求：

$f*g$ 要能够快速求前缀和。

g 要能够快速求分段和（前缀和）。

对于正常的积性函数 $g(1)=1$，所以不会有什么问题。

在预处理 $S(n)$ 前 $n^{\frac{2}{3}}$ 项的情况下复杂度是 $O(n^{\frac{2}{3}})$。

杜教筛筛欧拉函数

```cpp
#include<iostream>
#include<unordered_map>
#include<vector>
using namespace std;
typedef long long ll;
const int maxn=5e6;
const int mod=1e9+7;
int Madd(int a,int b){
    return a+b<mod?a+b:a+b-mod;
}
int Msub(int a,int b){
    return a<b?mod+a-b:a-b;
}
int Mmul(int a,int b){
    return (ll)a*b%mod;
}
vector<int>p;
int vis[maxn+10];
int phi[maxn+10];
unordered_map<ll,ll>map;
ll qphi(ll n){
    if(n<maxn)return phi[n];
    auto it=map.find(n);
    if(it!=map.end())
        return it->second;
    ll res=n&1?Mmul((n+1)/2%mod,n%mod):Mmul(n/2%mod,(n+1)%mod);
    for(ll i=2,last;i<=n;i=last+1){
        last=n/(n/i);
        res=Msub(res,Mmul((last-i+1)%mod,qphi(n/i)));
    }
    map.emplace(n,res);
    return res;
}
void init(){
    phi[1]=1;
    for(int i=2;i<maxn;i++){
        if(!vis[i]){
        p.push_back(i);
        phi[i]=i-1;
    }
    for(int j=0;j<(int)p.size()&&i*p[j]<maxn;j++){
            vis[i*p[j]]=1;
            if(i%p[j])
                phi[i*p[j]]=phi[i]*(p[j]-1);
            else{
                phi[i*p[j]]=phi[i]*p[j];
                break;
            }
        }
    }
    for(int i=2;i<maxn;i++)
        phi[i]=(phi[i]+phi[i-1])%mod;
}
int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    init();
    long long a;
    cin>>a;
    cout<<qphi(a)<<endl;
    return 0;
}
```

杜教筛筛莫比乌斯函数

```cpp
#include<iostream>
#include<unordered_map>
#include<vector>
using namespace std;
typedef long long ll;
const int maxn=5e6;

vector<int>p;
int vis[maxn+10];
int mu[maxn+10];
unordered_map<ll,ll>map;
ll qmu(ll n){
    if(n<maxn)return mu[n];
    auto it=map.find(n);
    if(it!=map.end())
        return it->second;
    ll res=1;
    for(ll i=2,last;i<=n;i=last+1){
        last=n/(n/i);
        res-=(ll)(last-i+1)*qmu(n/i);
    }
    map.emplace(n,res);
    return res;
}
void init(){
    mu[1]=1;
```

```
        for(int i=2;i<maxn;i++){
            if(!vis[i]){
            p.push_back(i);
            mu[i]=-1;
        }
        for(int j=0;j<(int)p.size()&&i*p[j]<maxn;j++){
            vis[i*p[j]]=1;
            if(i%p[j])
                mu[i*p[j]]=-mu[i];
            else
                break;
            }
        }
        for(int i=2;i<maxn;i++)
            mu[i]+=mu[i-1];
    }
    int main(){
        ios_base::sync_with_stdio(false);
        cin.tie(0);
        init();
        long long a,b;
        cin>>a>>b;
        cout<<qmu(b)-qmu(a-1)<<endl;
        return 0;
    }
```

1.3 扩展欧几里德

求 $ax+by=gcd(a,b)$ 的一组解

如果 a 和 b 互素，那么 x 是 a 在模 b 下的逆元

注意 x 和 y 可能是负数

```
LL ex_gcd(LL a, LL b, LL &x, LL &y) {
    if (b == 0) { x = 1; y = 0; return a; }
    LL ret = ex_gcd(b, a % b, y, x);
    y -= a / b * x;
    return ret;
}
```

1.4 类欧几里德

$m=\lfloor \frac{an+b}{c} \rfloor$.

$f(a,b,c,n)=\sum_{i=0}^{n}\lfloor \frac{ai+b}{c} \rfloor$: 当 $a \geq c$ or $b \geq c$ 时，$f(a,b,c,n)=(\frac{a}{c})n(n+1)/2+(\frac{b}{c})(n+1)+f(a \bmod c, b \bmod c, c, n)$；否则 $f(a,b,c,n)=nm-f(c,c-b-1,a,m-1)$。

$g(a,b,c,n)=\sum_{i=0}^{n}i\lfloor \frac{ai+b}{c} \rfloor$: 当 $a \geq c$ or $b \geq c$ 时，$g(a,b,c,n)=(\frac{a}{c})n(n+1)(2n+1)/6+(\frac{b}{c})n(n+1)/2+g(a \bmod c, b \bmod c, c, n)$；否则 $g(a,b,c,n)=\frac{1}{2}(n(n+1)m-f(c,c-b-1,a,m-1)-h(c,c-b-1,a,m-1))$。

$h(a,b,c,n)=\sum_{i=0}^{n}\lfloor \frac{ai+b}{c} \rfloor^2$: 当 $a \geq c$ or $b \geq c$ 时，$h(a,b,c,n)=(\frac{a}{c})^2n(n+1)(2n+1)/6+(\frac{b}{c})^2(n+1)+(\frac{a}{c})(\frac{b}{c})n(n+1)+h(a \bmod c, b \bmod c, c, n)+2(\frac{a}{c})g(a \bmod c, b \bmod c, c, n)+2(\frac{b}{c})f(a \bmod c, b \bmod c, c, n)$；否则 $h(a,b,c,n)=nm(m+1)-2g(c,c-b-1,a,m-1)-2f(c,c-b-1,a,m-1)-f(a,b,c,n)$。

1.5 逆元

预处理 1 n 的逆元

```
LL inv[N] = {-1, 1};
void inv_init(LL n, LL p) {
    inv[1] = 1;
    for(int i=2;i<=n;i++)
    inv[i] = (p - p / i) * inv[p % i] % p;
}
```

预处理阶乘及其逆元

```
LL invf[M], fac[M] = {1};
void fac_inv_init(LL n, LL p) {
    for(int i = 2;i <= n ; i++)
        fac[i] = i * fac[i - 1] % p;
    invf[n - 1] = qpow(fac[n - 1], p - 2, p);
    for(int i = n-1;i >= 0 ; i--)
        invf[i] = invf[i + 1] * (i + 1) % p;
}
```

1.6 组合数

如果数较小，模较大时使用逆元前置模板：逆元-预处理阶乘及其逆元

```
inline LL C(LL n, LL m) { // n >= m >= 0
    return n < m || m < 0 ? 0 : fac[n] * invf[m] %
        MOD * invf[n - m] % MOD;
}
```

如果模数较小，数字较大，使用 Lucas 定理

前置模板可选 1：求组合数（如果使用阶乘逆元，需

```
fac_inv_init(MOD, MOD)
```

前置模板可选 2：模数不固定下使用，无法单独使用。

```
LL C(LL n, LL m) { // m >= n >= 0
    if (m - n < n) n = m - n;
    if (n < 0) return 0;
    LL ret = 1;
```

```
        for(int i=1;i<=n;i++)
            ret = ret * (m - n + i) % MOD * qpow(i, MOD -
                 2, MOD) % MOD;
        return ret;
    }

    LL Lucas(LL n, LL m) { // m >= n >= 0
        return m ? C(n % MOD, m % MOD) * Lucas(n / MOD,
            m / MOD) % MOD : 1;
    }
```

1.7 公式

1.7.1 数论公式

当 $x \geq \phi(p)$ 时有 $a^x \equiv a^{x \ mod \ \phi(p)+\phi(p)} \pmod p$

$\mu^2(n) = \sum_{d^2|n} \mu(d)$

$\sum_{d|n} \varphi(d) = n$

$\sum_{d|n} 2^{\omega(d)} = \sigma_0(n^2)$，其中 ω 是不同素因子个数

$\sum_{d|n} \mu^2(d) = 2^{\omega(d)}$

1.7.2 一些数论函数求和的例子

$\sum_{i=1}^n i[gcd(i,n)=1] = \frac{n\varphi(n)+[n=1]}{2}$

$\sum_{i=1}^n \sum_{j=1}^m [gcd(i,j)=x] = \sum_d \mu(d) \lfloor \frac{n}{dx} \rfloor \lfloor \frac{m}{dx} \rfloor$

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

$S(n) = \sum_{i=1}^n \mu(i) = 1 - \sum_{i=1}^n \sum_{d|i,d<i} \mu(d) \overset{t=\frac{i}{d}}{=} 1 - \sum_{t=2}^n S(\lfloor \frac{n}{t} \rfloor)$

利用 $[n=1] = \sum_{d|n} \mu(d)$

$S(n) = \sum_{i=1}^n \varphi(i) = \sum_{i=1}^n i - \sum_{i=1}^n \sum_{d|i,d<i} \varphi(i) \overset{t=\frac{i}{d}}{=} \frac{i(i+1)}{2} - \sum_{t=2}^n S(\frac{n}{t})$

利用 $n = \sum_{d|n} \varphi(d)$

$\sum_{i=1}^n \mu^2(i) = \sum_{i=1}^n \sum_{d^2|n} \mu(d) = \sum_{d=1}^{\lfloor \sqrt{n} \rfloor} \mu(d) \lfloor \frac{n}{d^2} \rfloor$

$\sum_{i=1}^n \sum_{j=1}^n gcd^2(i,j) = \sum_d d^2 \sum_t \mu(t) \lfloor \frac{n}{dt} \rfloor^2 \overset{x=dt}{=} \sum_x \lfloor \frac{n}{x} \rfloor^2 \sum_{d|x} d^2 \mu(\frac{t}{x})$

$\sum_{i=1}^n \varphi(i) = \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n [i \perp j] - 1 = \frac{1}{2} \sum_{i=1}^n \mu(i) \cdot \lfloor \frac{n}{i} \rfloor^2 - 1$

1.7.3 莫比乌斯反演

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

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

1.7.4 低阶等幂求和

$\sum_{i=1}^n i^1 = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$

$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$

$\sum_{i=1}^n i^3 = \left[\frac{n(n+1)}{2}\right]^2 = \frac{1}{4}n^4 + \frac{1}{2}n^3 + \frac{1}{4}n^2$

$\sum_{i=1}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{1}{5}n^5 + \frac{1}{2}n^4 + \frac{1}{3}n^3 - \frac{1}{30}n$

$\sum_{i=1}^n i^5 = \frac{n^2(n+1)^2(2n^2+2n-1)}{12} = \frac{1}{6}n^6 + \frac{1}{2}n^5 + \frac{5}{12}n^4 - \frac{1}{12}n^2$

2 图论

2.1 k 短路

POJ 2449

```
#include<bits/stdc++.h>
using namespace std;
const int maxN=10000;
const int INF=0x3f3f3f3f;
typedef pair<int,int>P;
int n,m,s,t,k;
int dist[maxN],tdist[maxN],cnt[maxN];
bool f[maxN];
vector<P>Adj[maxN];
vector<P>Rev[maxN];
struct edge{
    int to,len;
    edge(){}
    edge(int t,int l):to(t),len(l){}
};
priority_queue<edge> q;
bool operator<(const edge &a,const edge &b){
    return (a.len+dist[a.to])>(b.len+dist[b.to]);
}
void dijkstra(){
    memset(dist,0,sizeof(dist));
    fill(tdist,tdist+maxN,INF);
    tdist[t]=0;
    while(!q.empty())q.pop();
    q.push(edge(t,0));
    while(!q.empty()){
        int x=q.top().to;
        int d=q.top().len;
        q.pop();
        if(tdist[x]<d)continue;
        for(int i=0;i<(int)Rev[x].size();i++){
            int y=Rev[x][i].first;
            int len=Rev[x][i].second;
            if(d+len<tdist[y]){
                tdist[y]=d+len;
                q.push(edge(y,tdist[y]));
            }
        }
    }
    for(int i=1;i<=n;i++)
        dist[i]=tdist[i];
```

```
}
int aStar(){
    if(dist[s]==INF)return -1;
    while(!q.empty())q.pop();
    q.push(edge(s,0));
    memset(cnt,0,sizeof(cnt));
    while(!q.empty()){
        int x=q.top().to;
        int d=q.top().len;
        q.pop();
        cnt[x]++;
        if(cnt[t]==k)return d;
        if(cnt[x]>k)continue;
        for(int i=0;i<(int)Adj[x].size();i++){
            int y=Adj[x][i].first;
            int len=Adj[x][i].second;
            q.push(edge(y,d+len));
            }
        }
    return -1;
}
int main(){
    scanf("%d%d",&n,&m);
    for(int i=0;i<m;i++){
        int st,ed,l;
        scanf("%d%d%d",&st,&ed,&l);
        Adj[st].push_back(make_pair(ed,l));
        Rev[ed].push_back(make_pair(st,l));
    }
    scanf("%d%d%d",&s,&t,&k);
    if(s==t)k++;
    dijkstra();
    printf("%d\n",aStar());
    return 0;
}
```

3 计算几何

3.1 二维几何

3.1.1 点与向量

```
#define y1 yy1
#define nxt(i) ((i + 1) % s.size())
typedef double LD;
const LD PI = 3.14159265358979323846;
const LD eps = 1E-10;
int sgn(LD x) { return fabs(x) < eps ? 0 : (x > 0 ? 1 : -1); }
struct L;
struct P;
typedef P V;
struct P {
    LD x, y;
    explicit P(LD x = 0, LD y = 0): x(x), y(y) {}
    explicit P(const L& l);
};
struct L {
    P s, t;
    L() {}
    L(P s, P t): s(s), t(t) {}
};

P operator + (const P& a, const P& b) { return P(a.x + b.x, a.y + b.y); }
P operator - (const P& a, const P& b) { return P(a.x - b.x, a.y - b.y); }
P operator * (const P& a, LD k) { return P(a.x * k, a.y * k); }
P operator / (const P& a, LD k) { return P(a.x / k, a.y / k); }
inline bool operator < (const P& a, const P& b) {
    return sgn(a.x - b.x) < 0 || (sgn(a.x - b.x) == 0 && sgn(a.y - b.y) < 0);
}
bool operator == (const P& a, const P& b) { return !sgn(a.x - b.x) && !sgn(a.y - b.y); }
P::P(const L& l) { *this = l.t - l.s; }
ostream &operator << (ostream &os, const P &p) {
    return (os << "(" << p.x << "," << p.y << ")");
}
istream &operator >> (istream &is, P &p) {
    return (is >> p.x >> p.y);
}

LD dist(const P& p) { return sqrt(p.x * p.x + p.y * p.y); }
LD dot(const V& a, const V& b) { return a.x * b.x + a.y * b.y; }
LD det(const V& a, const V& b) { return a.x * b.y - a.y * b.x; }
LD cross(const P& s, const P& t, const P& o = P()) { return det(s - o, t - o); }
// --------------------------------------------
```

3.1.2 象限

```
// 象限
int quad(P p) {
    int x = sgn(p.x), y = sgn(p.y);
    if (x > 0 && y >= 0) return 1;
    if (x <= 0 && y > 0) return 2;
    if (x < 0 && y <= 0) return 3;
    if (x >= 0 && y < 0) return 4;
    assert(0);
}

// 仅适用于参照点在所有点一侧的情况
struct cmp_angle {
    P p;
    bool operator () (const P& a, const P& b) {
        // int qa = quad(a), qb = quad(b);
        // if (qa != qb) return qa < qb;
        int d = sgn(cross(a, b, p));
        if (d) return d > 0;
        return dist(a - p) < dist(b - p);
    }
};
```

3.1.3 线

```
// 是否平行
bool parallel(const L& a, const L& b) {
    return !sgn(det(P(a), P(b)));
}
// 直线是否相等
bool l_eq(const L& a, const L& b) {
    return parallel(a, b) && parallel(L(a.s, b.t), L
        (b.s, a.t));
}
// 逆时针旋转 r 弧度
P rotation(const P& p, const LD& r) { return P(p.x *
     cos(r) - p.y * sin(r), p.x * sin(r) + p.y *
    cos(r)); }
P RotateCCW90(const P& p) { return P(-p.y, p.x); }
P RotateCW90(const P& p) { return P(p.y, -p.x); }
// 单位法向量
V normal(const V& v) { return V(-v.y, v.x) / dist(v)
    ; }
```

3.1.4 点与线

```
// 点在线段上 <= 0包含端点 < 0 则不包含
bool p_on_seg(const P& p, const L& seg) {
    P a = seg.s, b = seg.t;
    return !sgn(det(p - a, b - a)) && sgn(dot(p - a,
        p - b)) <= 0;
}
// 点到直线距离
LD dist_to_line(const P& p, const L& l) {
    return fabs(cross(l.s, l.t, p)) / dist(l);
}
// 点到线段距离
LD dist_to_seg(const P& p, const L& l) {
    if (l.s == l.t) return dist(p - l);
    V vs = p - l.s, vt = p - l.t;
    if (sgn(dot(l, vs)) < 0) return dist(vs);
    else if (sgn(dot(l, vt)) > 0) return dist(vt);
    else return dist_to_line(p, l);
}
```

3.1.5 线与线

```
// 求直线交 需要事先保证有界
P l_intersection(const L& a, const L& b) {
    LD s1 = det(P(a), b.s - a.s), s2 = det(P(a), b.t
        - a.s);
    return (b.s * s2 - b.t * s1) / (s2 - s1);
}
// 向量夹角的弧度
LD angle(const V& a, const V& b) {
    LD r = asin(fabs(det(a, b)) / dist(a) / dist(b))
        ;
    if (sgn(dot(a, b)) < 0) r = PI - r;
    return r;
}
// 线段和直线是否有交 1 = 规范, 2 = 不规范
int s_l_cross(const L& seg, const L& line) {
    int d1 = sgn(cross(line.s, line.t, seg.s));
    int d2 = sgn(cross(line.s, line.t, seg.t));
    if ((d1 ^ d2) == -2) return 1; // proper
    if (d1 == 0 || d2 == 0) return 2;
    return 0;
}
// 线段的交 1 = 规范, 2 = 不规范
int s_cross(const L& a, const L& b, P& p) {
```

```cpp
    int d1 = sgn(cross(a.t, b.s, a.s)), d2 = sgn(
        cross(a.t, b.t, a.s));
    int d3 = sgn(cross(b.t, a.s, b.s)), d4 = sgn(
        cross(b.t, a.t, b.s));
    if ((d1 ^ d2) == -2 && (d3 ^ d4) == -2) { p =
        l_intersection(a, b); return 1; }
    if (!d1 && p_on_seg(b.s, a)) { p = b.s; return
        2; }
    if (!d2 && p_on_seg(b.t, a)) { p = b.t; return
        2; }
    if (!d3 && p_on_seg(a.s, b)) { p = a.s; return
        2; }
    if (!d4 && p_on_seg(a.t, b)) { p = a.t; return
        2; }
    return 0;
}
```

3.2 多边形

3.2.1 面积、凸包

```cpp
typedef vector<P> S;

// 点是否在多边形中 0 = 在外部 1 = 在内部 -1 = 在边界
    上
int inside(const S& s, const P& p) {
    int cnt = 0;
    FOR (i, 0, s.size()) {
        P a = s[i], b = s[nxt(i)];
        if (p_on_seg(p, L(a, b))) return -1;
        if (sgn(a.y - b.y) <= 0) swap(a, b);
        if (sgn(p.y - a.y) > 0) continue;
        if (sgn(p.y - b.y) <= 0) continue;
        cnt += sgn(cross(b, a, p)) > 0;
    }
    return bool(cnt & 1);
}
// 多边形面积，有向面积可能为负
LD polygon_area(const S& s) {
    LD ret = 0;
    FOR (i, 1, (LL)s.size() - 1)
    ret += cross(s[i], s[i + 1], s[0]);
    return ret / 2;
}
// 构建凸包 点不可以重复 < 0 边上可以有点， <= 0 则不
    能
// 会改变输入点的顺序
const int MAX_N = 1000;
S convex_hull(S& s) {
    // assert(s.size() >= 3);
    sort(s.begin(), s.end());
    S ret(MAX_N * 2);
    int sz = 0;
    FOR (i, 0, s.size()) {
        while (sz > 1 && sgn(cross(ret[sz - 1], s[i],
            ret[sz - 2])) < 0) --sz;
        ret[sz++] = s[i];
    }
    int k = sz;
    FORD (i, (LL)s.size() - 2, -1) {
        while (sz > k && sgn(cross(ret[sz - 1], s[i],
            ret[sz - 2])) < 0) --sz;
        ret[sz++] = s[i];
    }
    ret.resize(sz - (s.size() > 1));
    return ret;
}

P ComputeCentroid(const vector<P> &p) {
    P c(0, 0);
    LD scale = 6.0 * polygon_area(p);
    for (unsigned i = 0; i < p.size(); i++) {
        unsigned j = (i + 1) % p.size();
        c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[
            j].x * p[i].y);
    }
    return c / scale;
}
```

3.2.2 旋转卡壳

```cpp
LD rotatingCalipers(vector<P>& qs) {
    int n = qs.size();
    if (n == 2)
    return dist(qs[0] - qs[1]);
    int i = 0, j = 0;
    FOR (k, 0, n) {
        if (!(qs[i] < qs[k])) i = k;
        if (qs[j] < qs[k]) j = k;
    }
    LD res = 0;
    int si = i, sj = j;
    while (i != sj || j != si) {
        res = max(res, dist(qs[i] - qs[j]));
```

```
            if (sgn(cross(qs[(i+1)%n] - qs[i], qs[(j+1)%n] - qs[j])) < 0)
            i = (i + 1) % n;
            else j = (j + 1) % n;
        }
        return res;
    }

    int main() {
        int n;
        while (cin >> n) {
            S v(n);
            FOR (i, 0, n) cin >> v[i].x >> v[i].y;
            convex_hull(v);
            printf("%.0f\n", rotatingCalipers(v));
        }
    }
```

3.2.3 半平面交

```
struct LV {
    P p, v; LD ang;
    LV() {}
    LV(P s, P t): p(s), v(t - s) { ang = atan2(v.y, v.x); }
}; // 另一种向量表示

bool operator < (const LV &a, const LV& b) { return a.ang < b.ang; }
bool on_left(const LV& l, const P& p) { return sgn(cross(l.v, p - l.p)) >= 0; }
P l_intersection(const LV& a, const LV& b) {
    P u = a.p - b.p; LD t = cross(b.v, u) / cross(a.v, b.v);
    return a.p + a.v * t;
}

S half_plane_intersection(vector<LV>& L) {
    int n = L.size(), fi, la;
    sort(L.begin(), L.end());
    vector<P> p(n); vector<LV> q(n);
    q[fi = la = 0] = L[0];
    FOR (i, 1, n) {
        while (fi < la && !on_left(L[i], p[la - 1])) la--;
        while (fi < la && !on_left(L[i], p[fi])) fi++;
        q[++la] = L[i];
        if (sgn(cross(q[la].v, q[la - 1].v)) == 0) {
            la--;
            if (on_left(q[la], L[i].p)) q[la] = L[i];
        }
        if (fi < la) p[la - 1] = l_intersection(q[la - 1], q[la]);
    }
    while (fi < la && !on_left(q[fi], p[la - 1])) la--;
    if (la - fi <= 1) return vector<P>();
    p[la] = l_intersection(q[la], q[fi]);
    return vector<P>(p.begin() + fi, p.begin() + la + 1);
}

S convex_intersection(const vector<P> &v1, const vector<P> &v2) {
    vector<LV> h; int n = v1.size(), m = v2.size();
    FOR (i, 0, n) h.push_back(LV(v1[i], v1[(i + 1) % n]));
    FOR (i, 0, m) h.push_back(LV(v2[i], v2[(i + 1) % m]));
    return half_plane_intersection(h);
}
```

3.3 圆

```
struct C {
    P p; LD r;
    C(LD x = 0, LD y = 0, LD r = 0): p(x, y), r(r) {}
    C(P p, LD r): p(p), r(r) {}
};
```

3.3.1 三点求圆心

```
P compute_circle_center(P a, P b, P c) {
    b = (a + b) / 2;
    c = (a + c) / 2;
    return l_intersection({b, b + RotateCW90(a - b)}, {c , c + RotateCW90(a - c)});
}
```

3.3.2 圆线交点、圆圆交点

圆和线的交点关于圆心是顺时针的

```
vector<P> c_l_intersection(const L& l, const C& c) {
    vector<P> ret;
    P b(l), a = l.s - c.p;
    LD x = dot(b, b), y = dot(a, b), z = dot(a, a) -
        c.r * c.r;
    LD D = y * y - x * z;
    if (sgn(D) < 0) return ret;
    ret.push_back(c.p + a + b * (-y + sqrt(D + eps))
        / x);
    if (sgn(D) > 0) ret.push_back(c.p + a + b * (-y
        - sqrt(D)) / x);
    return ret;
}

vector<P> c_c_intersection(C a, C b) {
    vector<P> ret;
    LD d = dist(a.p - b.p);
    if (sgn(d) == 0 || sgn(d - (a.r + b.r)) > 0 ||
        sgn(d + min(a.r, b.r) - max(a.r, b.r)) < 0)
    return ret;
    LD x = (d * d - b.r * b.r + a.r * a.r) / (2 * d)
        ;
    LD y = sqrt(a.r * a.r - x * x);
    P v = (b.p - a.p) / d;
    ret.push_back(a.p + v * x + RotateCCW90(v) * y);
    if (sgn(y) > 0) ret.push_back(a.p + v * x -
        RotateCCW90(v) * y);
    return ret;
}
```

3.3.3 圆圆位置关系

```
// 1:内含 2:内切 3:相交 4:外切 5:相离
int c_c_relation(const C& a, const C& v) {
    LD d = dist(a.p - v.p);
    if (sgn(d - a.r - v.r) > 0) return 5;
    if (sgn(d - a.r - v.r) == 0) return 4;
    LD l = fabs(a.r - v.r);
    if (sgn(d - l) > 0) return 3;
    if (sgn(d - l) == 0) return 2;
    if (sgn(d - l) < 0) return 1;
}
```

3.3.4 圆与多边形交

HDU 5130 注意顺时针逆时针（可能要取绝对值）

```
LD sector_area(const P& a, const P& b, LD r) {
    LD th = atan2(a.y, a.x) - atan2(b.y, b.x);
    while (th <= 0) th += 2 * PI;
    while (th > 2 * PI) th -= 2 * PI;
    th = min(th, 2 * PI - th);
    return r * r * th / 2;
}

LD c_tri_area(P a, P b, P center, LD r) {
    a = a - center; b = b - center;
    int ina = sgn(dist(a) - r) < 0, inb = sgn(dist(b
        ) - r) < 0;
    // dbg(a, b, ina, inb);
    if (ina && inb) {
        return fabs(cross(a, b)) / 2;
    } else {
        auto p = c_l_intersection(L(a, b), C(0, 0, r)
            );
        if (ina ^ inb) {
            auto cr = p_on_seg(p[0], L(a, b)) ? p[0]
                : p[1];
            if (ina) return sector_area(b, cr, r) +
                fabs(cross(a, cr)) / 2;
            else return sector_area(a, cr, r) + fabs(
                cross(b, cr)) / 2;
        } else {
            if ((int) p.size() == 2 && p_on_seg(p[0],
                L(a, b))) {
                if (dist(p[0] - a) > dist(p[1] - a))
                    swap(p[0], p[1]);
                return sector_area(a, p[0], r) +
                    sector_area(p[1], b, r)
                    + fabs(cross(p[0], p[1])) / 2;
            } else return sector_area(a, b, r);
        }
    }
}

typedef vector<P> S;
LD c_poly_area(S poly, const C& c) {
    LD ret = 0; int n = poly.size();
    FOR (i, 0, n) {
        int t = sgn(cross(poly[i] - c.p, poly[(i + 1)
            % n] - c.p));
```

```
        if (t) ret += t * c_tri_area(poly[i], poly[(i
            + 1) % n], c.p, c.r);
    }
    return ret;
}
```

3.3.5 最小圆覆盖

随机增量。期望复杂度 $O(n)$。

```
P compute_circle_center(P a, P b) { return (a + b) /
    2; }
bool p_in_circle(const P& p, const C& c) {
    return sgn(dist(p - c.p) - c.r) <= 0;
}
C min_circle_cover(const vector<P> &in) {
    vector<P> a(in.begin(), in.end());
    dbg(a.size());
    random_shuffle(a.begin(), a.end());
    P c = a[0]; LD r = 0; int n = a.size();
    FOR (i, 1, n) if (!p_in_circle(a[i], {c, r})) {
        c = a[i]; r = 0;
        FOR (j, 0, i) if (!p_in_circle(a[j], {c, r}))
            {
            c = compute_circle_center(a[i], a[j]);
            r = dist(a[j] - c);
            FOR (k, 0, j) if (!p_in_circle(a[k], {c,
                r})) {
                c = compute_circle_center(a[i], a[j],
                    a[k]);
                r = dist(a[k] - c);
            }
        }
    }
    return {c, r};
}
```

3.3.6 圆的反演

```
C inv(C c, const P& o) {
    LD d = dist(c.p - o);
    assert(sgn(d) != 0);
    LD a = 1 / (d - c.r);
    LD b = 1 / (d + c.r);
    c.r = (a - b) / 2 * R2;
    c.p = o + (c.p - o) * ((a + b) * R2 / 2 / d);
    return c;
}
```

3.4 三维计算几何

```
struct P;
struct L;
typedef P V;

struct P {
    LD x, y, z;
    explicit P(LD x = 0, LD y = 0, LD z = 0): x(x),
        y(y), z(z) {}
    explicit P(const L& l);
};

struct L {
    P s, t;
    L() {}
    L(P s, P t): s(s), t(t) {}
};

struct F {
    P a, b, c;
    F() {}
    F(P a, P b, P c): a(a), b(b), c(c) {}
};

P operator + (const P& a, const P& b) { return P(a.x
    + b.x, a.y + b.y, a.z + b.z); }
P operator - (const P& a, const P& b) { return P(a.x
    - b.x, a.y - b.y, a.z - b.z); }
P operator * (const P& a, LD k) { return P(a.x * k,
    a.y * k, a.z * k); }
P operator / (const P& a, LD k) { return P(a.x / k,
    a.y / k, a.z / k); }
inline int operator < (const P& a, const P& b) {
    return sgn(a.x - b.x) < 0 || (sgn(a.x - b.x) ==
        0 && (sgn(a.y - b.y) < 0 ||
    (sgn(a.y - b.y) == 0 && sgn(a.z - b.z) < 0)));
}
bool operator == (const P& a, const P& b) { return !
    sgn(a.x - b.x) && !sgn(a.y - b.y) && !sgn(a.z -
    b.z); }
P::P(const L& l) { *this = l.t - l.s; }
ostream &operator << (ostream &os, const P &p) {
    return (os << "(" << p.x << "," << p.y << "," <<
        p.z << ")");
```

```cpp
}
istream &operator >> (istream &is, P &p) {
    return (is >> p.x >> p.y >> p.z);
}

// --------------------------------------------
LD dist2(const P& p) { return p.x * p.x + p.y * p.y + p.z * p.z; }
LD dist(const P& p) { return sqrt(dist2(p)); }
LD dot(const V& a, const V& b) { return a.x * b.x + a.y * b.y + a.z * b.z; }
P cross(const P& v, const P& w) {
    return P(v.y * w.z - v.z * w.y, v.z * w.x - v.x * w.z, v.x * w.y - v.y * w.x);
}
LD mix(const V& a, const V& b, const V& c) { return dot(a, cross(b, c)); }
```

3.4.1 旋转

```cpp
// 逆时针旋转 r 弧度
// axis = 0 绕 x 轴
// axis = 1 绕 y 轴
// axis = 2 绕 z 轴
P rotation(const P& p, const LD& r, int axis = 0) {
    if (axis == 0)
    return P(p.x, p.y * cos(r) - p.z * sin(r), p.y * sin(r) + p.z * cos(r));
    else if (axis == 1)
    return P(p.z * cos(r) - p.x * sin(r), p.y, p.z * sin(r) + p.x * cos(r));
    else if (axis == 2)
    return P(p.x * cos(r) - p.y * sin(r), p.x * sin(r) + p.y * cos(r), p.z);
}
// n 是单位向量 表示旋转轴
// 模板是顺时针的
P rotation(const P& p, const LD& r, const P& n) {
    LD c = cos(r), s = sin(r), x = n.x, y = n.y, z = n.z;
    // dbg(c, s);
    return P((x * x * (1 - c) + c) * p.x + (x * y * (1 - c) + z * s) * p.y + (x * z * (1 - c) - y * s) * p.z,
    (x * y * (1 - c) - z * s) * p.x + (y * y * (1 - c) + c) * p.y + (y * z * (1 - c) + x * s) * p.z,
        (x * z * (1 - c) + y * s) * p.x + (y * z * (1 - c) - x * s) * p.y + (z * z * (1 - c) + c) * p.z);
}
```

3.4.2 线、面

函数相互依赖，所以交织在一起了。

```cpp
// 点在线段上 <= 0包含端点 < 0 则不包含
bool p_on_seg(const P& p, const L& seg) {
    P a = seg.s, b = seg.t;
    return !sgn(dist2(cross(p - a, b - a))) && sgn(dot(p - a, p - b)) <= 0;
}
// 点到直线距离
LD dist_to_line(const P& p, const L& l) {
    return dist(cross(l.s - p, l.t - p)) / dist(l);
}
// 点到线段距离
LD dist_to_seg(const P& p, const L& l) {
    if (l.s == l.t) return dist(p - l.s);
    V vs = p - l.s, vt = p - l.t;
    if (sgn(dot(l, vs)) < 0) return dist(vs);
    else if (sgn(dot(l, vt)) > 0) return dist(vt);
    else return dist_to_line(p, l);
}

P norm(const F& f) { return cross(f.a - f.b, f.b - f.c); }
int p_on_plane(const F& f, const P& p) { return sgn(dot(norm(f), p - f.a)) == 0; }

// 判两点在线段异侧 点在线段上返回 0 不共面无意义
int opposite_side(const P& u, const P& v, const L& l) {
    return sgn(dot(cross(P(l), u - l.s), cross(P(l), v - l.s))) < 0;
}

bool parallel(const L& a, const L& b) { return !sgn(dist2(cross(P(a), P(b)))); }
// 线段相交
int s_intersect(const L& u, const L& v) {
    return p_on_plane(F(u.s, u.t, v.s), v.t) &&
    opposite_side(u.s, u.t, v) &&
    opposite_side(v.s, v.t, u);
}
```

3.4.3 凸包

增量法。先将所有的点打乱顺序，然后选择四个不共面的点组成一个四面体，如果找不到说明凸包不存在。然后遍历剩余的点，不断更新凸包。对遍历到的点做如下处理。

1. 如果点在凸包内，则不更新。2. 如果点在凸包外，那么找到所有原凸包上所有分隔了对于这个点可见面和不可见面的边，以这样的边的两个点和新的点创建新的面加入凸包中。

```
struct FT {
    int a, b, c;
    FT() { }
    FT(int a, int b, int c) : a(a), b(b), c(c) { }
};

bool p_on_line(const P& p, const L& l) {
    return !sgn(dist2(cross(p - l.s, P(l))));
}

vector<F> convex_hull(vector<P> &p) {
    sort(p.begin(), p.end());
    p.erase(unique(p.begin(), p.end()), p.end());
    random_shuffle(p.begin(), p.end());
    vector<FT> face;
    FOR (i, 2, p.size()) {
        if (p_on_line(p[i], L(p[0], p[1]))) continue;
        swap(p[i], p[2]);
        FOR (j, i + 1, p.size())
        if (sgn(mix(p[1] - p[0], p[2] - p[1], p[j] - p[0]))) {
            swap(p[j], p[3]);
            face.emplace_back(0, 1, 2);
            face.emplace_back(0, 2, 1);
            goto found;
        }
    }
    found:
    vector<vector<int>> mk(p.size(), vector<int>(p.size()));
    FOR (v, 3, p.size()) {
        vector<FT> tmp;
        FOR (i, 0, face.size()) {
            int a = face[i].a, b = face[i].b, c = face[i].c;
            if (sgn(mix(p[a] - p[v], p[b] - p[v], p[c] - p[v])) < 0) {
                mk[a][b] = mk[b][a] = v;
                mk[b][c] = mk[c][b] = v;
                mk[c][a] = mk[a][c] = v;
            } else tmp.push_back(face[i]);
        }
        face = tmp;
        FOR (i, 0, tmp.size()) {
            int a = face[i].a, b = face[i].b, c = face[i].c;
            if (mk[a][b] == v) face.emplace_back(b, a, v);
            if (mk[b][c] == v) face.emplace_back(c, b, v);
            if (mk[c][a] == v) face.emplace_back(a, c, v);
        }
    }
    vector<F> out;
    FOR (i, 0, face.size())
    out.emplace_back(p[face[i].a], p[face[i].b], p[face[i].c]);
    return out;
}
```

4 杂项

4.1 数位 dp

```
LL dfs(LL base, LL pos, LL len, LL s, bool limit) {
    if (pos == -1) return s ? base : 1;
    if (!limit && dp[base][pos][len][s] != -1)
        return dp[base][pos][len][s];
    LL ret = 0;
    LL ed = limit ? a[pos] : base - 1;
    for(int i= 0;i<ed + 1;i++) {
        tmp[pos] = i;
        if (len == pos)
            ret += dfs(base, pos - 1, len - (i == 0), s, limit && i == a[pos]);
        else if (s &&pos < (len + 1) / 2)
            ret += dfs(base, pos - 1, len, tmp[len - pos] == i, limit && i == a[pos]);
        else
        ret += dfs(base, pos - 1, len, s, limit && i == a[pos]);
    }
    if (!limit) dp[base][pos][len][s] = ret;
    return ret;
}
```

```
LL solve(LL x, LL base) {
    LL sz = 0;
    while (x) {
        a[sz++] = x % base;
        x /= base;
    }
    return dfs(base, sz - 1, sz - 1, 1, true);
}
```