ACM 模板库

Dounm

2014-10-15

BITSS

目录

二维计算几何

```
const int MAXV = 100100 ;
const double eps = 1e-8;
const double inf = 1e8;
const double PI = 2.0*asin(1.0);
struct Point {     //点
	double x,y;
	Point(){}
	Point(double _x,double _y):x(_x),y(_y){}//拷贝构造函数
	//重载运算符不会改变操作符的两个结构体对象
	Point operator+(const Point p){return Point(x+p.x,y+p.y);}
	Point operator-(const Point p){return Point(x-p.x,y-p.y);}
	Point operator*(const double p){return Point(x*p,y*p);}
	Point operator/(const double p){return Point(x/p,y/p);}
	double operator^(const Point p){return x*p.x+y*p.y;}//点积
	double operator*(const Point p){return x*p.y-y*p.x;}//叉积,因为是二维平面,所以返回 double
};
struct Seg{Point a,b;};   //线段
struct Dir{
	double dx,dy;
	double operator^(const Dir p){return dx*p.dx+dy*p.dy;}//点积
	double operator*(const Dir p){return dx*p.dy-dy*p.dx;}//叉积,因为是二维平面,所以返回 double
};   //方向向量
struct Line{Point p; Dir dir;}; //直线
struct Rad{Point Sp; Dir dir;}; //射线
struct Round{Point o; double r;};//圆


int dcmp(double x){return x < -eps ? -1 : x > eps;}//模糊精度
double len(Point p){return sqrt(p.x*p.x+p.y*p.y);}//求向量长度
double len2(Point p){return p.x*p.x+p.y*p.y;}//向量长度的平方
//判断点在直线顺时针侧或逆时针侧,返回叉积，值为正在逆时针，值为负在顺时针
double line_point(Point p1,Line ln){return (p1-ln.p)*Point(ln.dir.dx,ln.dir.dy);}

/*点,直线,线段模板*/
//double p2pdis(Point p1,Point p2){return len(p1-p2);}//求平面上两点之间的距离
//解一元二次方程 Ax^2+Bx+C=0;
//返回-1 无解,1 有两个不同解,0 有一个解且赋值给 x1
int equa(double A,double B,double C,double &x1,double &x2){
	double f=B*B-4*A*C;
	if(dcmp(f)<0) return -1;
	if(dcmp(f)==0){
		x1 = (-B)/(2*A);
		return 0;
	}
	x1 = (-B+sqrt(f))/(2*A);
	x2 = (-B-sqrt(f))/(2*A);
	return 1;
}
//计算直线一般式
void format(Line ln,double &A,double &B,double &C){
	A=ln.dir.dy;
	B=-ln.dir.dx;
	C=ln.p.y*ln.dir.dx-ln.p.x*ln.dir.dy;
}
//>>format();点到直线距离
double p2ldis(Point a,Line ln){
	double A,B,C;
	format(ln,A,B,C);
	return(fabs(A*a.x+B*a.y+C)/sqrt(A*A+B*B));
}
//>>len(),p2pdis();点到线段距离
double p2segdis(Point x,Seg seg){
	double a,b,c,cos1,cos2;
	a = len(x-seg.a);
	b = len(x-seg.b);
	c = len(seg.a-seg.b);
	cos1 = (a*a+c*c-b*b)/(2*a*c);
	cos2 = (b*b+c*c-a*a)/(2*b*c);
	if(cos1 <= 0 || cos2 <= 0)
		return min(a,b);
	Line ln;
	ln.p = seg.a;
	ln.dir.dx = seg.b.x-seg.a.x;
	ln.dir.dy = seg.b.y-seg.a.y;
	return p2ldis(x,ln);
}
//判断点是否在线段上
```

```
bool point_seg(Point p,Seg s){
	return (s.b-s.a)*(p-s.a)==0;
}
//>>line_point();判断直线与线段是否相交,包括端点
bool line_seg(Line ln,Seg seg){

	if(line_point(seg.a,ln)*line_point(seg.b,ln)<=0)
		return true;
	return false;
}
//>>line_point();判断两线段是否相交,包括端点
bool seg_seg(Seg s1,Seg s2){
	Line l1,l2;
	l1.p = s1.a; l1.dir.dx = s1.b.x-s1.a.x; l1.dir.dy = s1.b.y-s1.a.y;
	l2.p = s2.a; l2.dir.dx = s2.b.x-s2.a.x; l2.dir.dy = s2.b.y-s2.a.y;
	if(line_point(s1.a,l2)*line_point(s1.b,l2) > 0)return false;
	if(line_point(s2.a,l1)*line_point(s2.b,l1) > 0)return false;
	return true;
}
//判断两条直线是否相交，0 是相等，-1 平行但不相等，1 相交于一点
int line_line(Line a,Line b){
	if(dcmp(a.dir*b.dir)==0){
		if(dcmp((a.p-b.p)*Point(b.dir.dx,b.dir.dy))==0){
			return 0;
		}
		return -1;
	}
	return 1;
}
//已知两直线相交，求该交点
Point llcross(Line a,Line b){
	double k1,k2,b1,b2;
	Point ans;
	if(dcmp(a.dir.dx) != 0 && dcmp(b.dir.dx) != 0){
		k1 = a.dir.dy/a.dir.dx;
		k2 = b.dir.dy/b.dir.dx;
		b1 = (a.dir.dx*a.p.y-a.p.x*a.dir.dy)/a.dir.dx;
		b2 = (b.dir.dx*b.p.y-b.p.x*b.dir.dy)/b.dir.dx;
		ans.x = (b1-b2)/(k2-k1);
		ans.y = (b1*k2-k1*b2)/(k2-k1);
	}
	else if(dcmp(a.dir.dx) == 0){
		ans.x = a.p.x;
		ans.y = (b.dir.dy*a.p.x+b.dir.dx*b.p.y-b.p.x*b.dir.dy)/b.dir.dx;
	}
	else{
		ans.x = b.p.x;
		ans.y = (a.dir.dy*b.p.x+a.dir.dx*a.p.y-a.p.x*a.dir.dy)/a.dir.dx;
	}
	return ans;
}
//>>format();求 p1 关于直线 ln 的对称点 p2
Point mirror(Point p1,Line ln){
	Point p2;
	double A,B,C;
	format(ln,A,B,C);
	p2.x=((B*B-A*A)*p1.x-2*A*B*p1.y-2*A*C)/(A*A+B*B);
	p2.y=((A*A-B*B)*p1.y-2*A*B*p1.x-2*B*C)/(A*A+B*B);
	return p2;
}



/*圆*/
//>>formant(),equa();求直线与圆的两个交点
//利用解一元二次方程来求解
void lrcross(Round R,Line ln,Point &p1,Point &p2){
	double A,B,C;
	format(ln,A,B,C);
	double x = R.o.x,y = R.o.y,r = R.r;
	if(dcmp(A) == 0){
		p1.y = p2.y = -C/B;
		//令横坐标为 t,(t-x)*(t-x) + (p1.y-y)*(p1.y-y) = r*r
		equa(1.0,-2*x,x*x+(p1.y-y)*(p1.y-y)-r*r,p1.x,p2.x);
	}
	else if(dcmp(B) == 0){
		p1.x = p2.x = -C/A;
		//令纵坐标为 t,(t-y)*(t-y) + (p1.x-x)*(p1.x-x) = r*r
		equa(1.0,-2*y,y*y+(p1.x-x)*(p1.x-x)-r*r,p1.y,p2.y);
	}
	else{
		//令横坐标为 t,(t-x)*(t-x)+((-A*t-C)/B-y)*((-A*t-C)/B-y) = r*r
		equa(A*A+B*B,2*A*C+2*A*B*y-2*x*B*B,B*B*x*x+C*C+2*B*y*C+B*B*y*y-B*B*r*r,p1.x,p2.x);
		p1.y = (-A*p1.x-C)/B;
		p2.y = (-A*p2.x-C)/B;
```

```
	}
}
//>>len();两个圆的两个交点.
bool round_round(Round R1,Round R2, Point& p1, Point& p2){
	Point o1 = R1.o,o2 = R2.o;
	double r1 = R1.r,r2 = R2.r;
	double d = len(o1-o2);
	if( dcmp(d-fabs(r1-r2))<0 || dcmp(d-r1-r2)>0 )
		return false;
	double cosa = (r1*r1 + d*d - r2*r2) / (2 * r1 * d);
	double sina = sqrt(max(0.0, 1.0 - cosa*cosa));
	p1 = p2 = o1;
	p1.x += r1 / d * ((o2.x - o1.x) * cosa + (o2.y - o1.y) * -sina);
	p1.y += r1 / d * ((o2.x - o1.x) * sina + (o2.y - o1.y) * cosa);
	p2.x += r1 / d * ((o2.x - o1.x) * cosa + (o2.y - o1.y) * sina);
	p2.y += r1 / d * ((o2.x - o1.x) * -sina + (o2.y - o1.y) * cosa);
	return true;
}
```

```
/*多边形*/
//对于给定的点集，让他们顺时针或逆时针排序。
//其中 tmp 是该点集中的任意一点作为起始点,排序后 tmp 定然在点集第一个位置
//下面的代码是让其逆时针排序
Point tmp;
bool anticlock_cmp(Point a,Point b){
	if(dcmp(len(a-tmp))==0)
		return true;
	else if(dcmp(len(b-tmp))==0)
		return false;
	if(dcmp(a*b) == 0)
		return a.x<b.x;
	return (a*b > 0);
}
//求多边形面积
//要求 plg 数组里面必须得是顺时针(面积为负)或逆时针(面积为正)方向
double area(int vcount,Point plg[]){
	int i;
	double s = 0;
	if(vcount < 3)return 0;
	for(i = 0;i < vcount-1;i++)
		s += plg[i]*plg[i+1];
	s += plg[vcount-1]*plg[0];
	return s/2;
}
//求多边形重心,需要输入点集为逆时针或顺时针,但是至少 3 个点
Point center(Point ply[],int pcnt){
	Point ret,p0,tri_center;
	ret.x = ret.y = 0;
	p0 = ret;
	double area = 0,tri_area;
	int s,t;
	for(int i = 0;i < pcnt;i ++){
		s = i;
		t = (i==pcnt-1)?0:i+1;
		tri_center.x = (ply[s].x + ply[t].x + p0.x);
		tri_center.y = (ply[s].y + ply[t].y + p0.y);
		tri_area = (ply[s]-p0)*(ply[t]-p0);
		area += tri_area;
		ret.x += tri_center.x*tri_area;
		ret.y += tri_center.y*tri_area;
	}
	//为减少误差,除法最后一起算
	ret.x = ret.x/(3*area);
	ret.y = ret.y/(3*area);
	return ret;
}
//判断点 q 是否在多边形内
//该多边形是任意的凸或凹多边形，但是 Polygon[]必须是多边形顶点逆时针序列
double get_angle(Point a,Point b){return acos(a^b/(len(a)*len(b)));}
bool pinplg(int vcount,Point ply[],Point q){
	double sum_angle = 0;
	for(int i = 0;i < vcount-1;i ++){
		sum_angle += get_angle(ply[i]-q,ply[i+1]-q);
	}
	sum_angle += get_angle(ply[0]-q,ply[vcount-1]-q);
	if(dcmp(sum_angle-2*PI) == 0)
		return true;
	return false;
}
```

```
/*计算几何算法*/
//graham 求凸包，O(n*log(n))
//pnt 为初始点集,res 为凸包点集,逆时针排序好了
bool hull_cmpy(Point a,Point b){
```

```
    if(dcmp(a.y-b.y)==0)return a.x<b.x;
    return a.y<b.y;
}
int graham(Point pnt[],int vcount,Point res[]){
    int i,len,k = 0,top = 1;
    sort(pnt,pnt+vcount,hull_cmpy);
    if(vcount==0)return 0;res[0] = pnt[0];
    if(vcount==1)return 1;res[1] = pnt[1];
    if(vcount==2)return 2;res[2] = pnt[2];
    for(i = 2;i < vcount;i ++){ //增加凸包逆时针上升一侧的点
        while(top && dcmp((pnt[i]-res[top-1])*(res[top]-res[top-1]))>0)
            top--;
        res[++top] = pnt[i];
    }
    len = top; res[++top] = pnt[vcount-2];
    for(i = vcount-3;i >= 0;i --){//增加凸包逆时针下降一侧的点
        while(top!=len && dcmp((pnt[i]-res[top-1])*(res[top]-res[top-1]))>0)
            top--;
        res[++top] = pnt[i];
    }
    return top;//凸包中点的个数为 top:0->(top-1).因为节点 0 算了两次
}
//>>len2();平面最近点对,O(n*log(n))
Point lis[MAXV];
bool minp2p_cmpx(Point a,Point b){return a.x<b.x;}
bool minp2p_cmpy(Point a,Point b){return a.y<b.y;}
double getmindist(Point p[],int l,int r){
    if(r-l==1)return len2(p[r]-p[l]);
    if(r-l==2)return  min(len2(p[r]-p[l+1]),min(len2(p[l+1]-p[l]),len2(p[r]-p[l])));
    int mid = (l+r)>>1,lislen = 0;
    double ans = min(getmindist(p,l,mid),getmindist(p,mid,r));
    int k = mid-1;
    while(k>=l && dcmp(p[mid].x-p[k].x-ans)<=0){
        lis[lislen++] = p[k];
        k--;
    }
    k = mid+1;
    while(k<=r && dcmp(p[mid].x-p[k].x-ans)<=0){
        lis[lislen++] = p[k];
        k++;
    }
    sort(lis,lis+lislen,minp2p_cmpy);
    for(int i = 0;i < lislen;i++){
        for(int j = i+1;j<=i+7&&j<lislen;j ++){
            ans = min(ans,len2(p[i]-p[j]));
        }
    }
    return ans;
}
double mindist_p2p(int vcount,Point p[]){
    sort(p,p+vcount,minp2p_cmpx);
    return sqrt(getmindist(p,0,vcount-1));
}
```

三维计算几何

```
/*
此模板中:
线段用两个端点 a,b 表示
直线用直线上任意两点 a,b 表示
平面用平面上任意一点 p0 以及该平面的法向量的单位向量 n 表示
*/
/**********基础*************/
const double eps = 1e-6;
 int dcmp(double x){
    return x < -eps ? -1 : x > eps;
 }
struct Point3 {
  double x, y, z;
  Point3(double x=0, double y=0, double z=0):x(x),y(y),z(z) { }
};

typedef Point3 Vector3;

Vector3 operator + (const Vector3& A, const Vector3& B) { return Vector3(A.x+B.x, A.y+B.y, A.z+B.z); }
Vector3 operator - (const Point3& A, const Point3& B) { return Vector3(A.x-B.x, A.y-B.y, A.z-B.z); }
Vector3 operator * (const Vector3& A, double p) { return Vector3(A.x*p, A.y*p, A.z*p); }
Vector3 operator / (const Vector3& A, double p) { return Vector3(A.x/p, A.y/p, A.z/p); }

double Dot(const Vector3& A, const Vector3& B) { return A.x*B.x + A.y*B.y + A.z*B.z; }
double Length(const Vector3& A) { return sqrt(Dot(A, A)); }
double Angle(const Vector3& A, const Vector3& B) { return
```

```
acos(Dot(A, B) / Length(A) / Length(B)); }
Vector3 Cross(const Vector3& A, const Vector3& B) { return Vector3(A.y*B.z - A.z*B.y, A.z*B.x - A.x*B.z, A.x*B.y - A.y*B.x); }
double Area2(const Point3& A, const Point3& B, const Point3& C) { return Length(Cross(B-A, C-A)); }
double Volume6(const Point3& A, const Point3& B, const Point3& C, const Point3& D) { return Dot(D-A, Cross(B-A, C-A)); }
// 四面体的重心
Point3 Centroid(const Point3& A, const Point3& B, const Point3& C, const Point3& D) { return (A + B + C + D)/4.0; }

/************点线面*************/
// 点 p 到平面 p0-n 的距离。n 必须为单位向量
double DistanceToPlane(const Point3& p, const Point3& p0, const Vector3& n) {
   return fabs(Dot(p-p0, n)); // 如果不取绝对值，得到的是有向距离
}

// 点 p 在平面 p0-n 上的投影。n 必须为单位向量
Point3 GetPlaneProjection(const Point3& p, const Point3& p0, const Vector3& n) {
   return p-n*Dot(p-p0, n);
}

//直线 p1-p2 或线段 p1-p2 与平面 p0-n 的交点
Point3 LinePlaneIntersection(Point3 p1, Point3 p2, Point3 p0, Vector3 n)
{
      vector3 v = p2-p1;
      if(dcmp(Dot(n,p2-p1)) == 0)//此时直线与平面平行或在平面上
            return Point3(0.0,0.0);
      double t = (Dot(n, p0-p1) / Dot(n, p2-p1));//分母为 0，直线与平面平行或在平面上
      return p1 + v*t; //如果是线段 判断 t 是否在 0~1 之间
}

// 点 P 到直线 AB 的距离
double DistanceToLine(const Point3& P, const Point3& A, const Point3& B) {
   Vector3 v1 = B - A, v2 = P - A;
   return Length(Cross(v1, v2)) / Length(v1);
}

//点到线段的距离
double DistanceToSeg(Point3 p, Point3 a, Point3 b)
{
      if(a == b) return Length(p-a);
      Vector3 v1 = b-a, v2 = p-a, v3 = p-b;
      if(dcmp(Dot(v1, v2)) < 0) return Length(v2);
      else if(dcmp(Dot(v1, v3)) > 0) return Length(v3);
      else return Length(Cross(v1, v2)) / Length(v1);
}

//求异面直线 p1+s*u 与 p2+t*v 的公垂线对应的 s 如果平行|重合，返回 false
bool LineDistance3D(Point3 p1, Vector3 u, Point3 p2, Vector3 v, double& s)
{
      double b = Dot(u, u) * Dot(v, v) - Dot(u, v) * Dot(u, v);
      if(dcmp(b) == 0) return false;
      double a = Dot(u, v) * Dot(v, p1-p2) - Dot(v, v) * Dot(u, p1-p2);
      s = a/b;
      return true;
}

// p1 和 p2 是否在线段 a-b 的同侧
// 前提条件是确定了 p1,p2 和线段 a-b 是在同一平面内
bool SameSide(const Point3& p1, const Point3& p2, const Point3& a, const Point3& b) {
   return dcmp(Dot(Cross(b-a, p1-a), Cross(b-a, p2-a))) >= 0;
}

// 点 P 在三角形 P0, P1, P2 中
// 前提条件:确定 p 和三角形共面
bool PointInTri(const Point3& P, const Point3& P0, const Point3& P1, const Point3& P2) {
   return SameSide(P, P0, P1, P2) && SameSide(P, P1, P0, P2) && SameSide(P, P2, P0, P1);
}

// 三角形 P0P1P2 是否和线段 AB 相交
bool TriSegIntersection(const Point3& P0, const Point3& P1, const Point3& P2, const Point3& A, const Point3& B, Point3& P) {
   Vector3 n = Cross(P1-P0, P2-P0);
   if(dcmp(Dot(n, B-A)) == 0) return false; // 线段 A-B 和平面 P0P1P2 平行或共面
   else { // 平面 A 和直线 P1-P2 有惟一交点
      double t = Dot(n, P0-A) / Dot(n, B-A);
      if(dcmp(t) < 0 || dcmp(t-1) > 0) return false; // 不在线段 AB 上
      P = A + (B-A)*t; // 交点
```

```
    return PointInTri(P, P0, P1, P2);
  }
}

//空间两三角形是否相交
bool TriTriIntersection(Point3* T1, Point3* T2) {
  Point3 P;
  for(int i = 0; i < 3; i++) {
    if(TriSegIntersection(T1[0], T1[1], T1[2], T2[i], T2[(i+1)%3], P)) return true;
    if(TriSegIntersection(T2[0], T2[1], T2[2], T1[i], T1[(i+1)%3], P)) return true;
  }
  return false;
}

//空间两直线上最近点对 返回最近距离 点对保存在 ans1 ans2 中
double SegSegDistance(Point3 a1, Point3 b1, Point3 a2, Point b2)
{
    Vector v1 = (a1-b1), v2 = (a2-b2);
    Vector N = Cross(v1, v2);
    Vector ab = (a1-a2);
    double ans = Dot(N, ab) / Length(N);
    Point p1 = a1, p2 = a2;
    Vector d1 = b1-a1, d2 = b2-a2;
    double t1, t2;
    t1 = Dot((Cross(p2-p1, d2)), Cross(d1, d2));
    t2 = Dot((Cross(p2-p1, d1)), Cross(d1, d2));
    double dd = Length((Cross(d1, d2)));
    t1 /= dd*dd;
    t2 /= dd*dd;
    ans1 = (a1 + (b1-a1)*t1);
    ans2 = (a2 + (b2-a2)*t2);
    return fabs(ans);
}

// 判断 P 是否在三角形 A, B, C 中，并且到三条边的距离都至少为 mindist。保证 P, A, B, C 共面
bool InsideWithMinDistance(const Point3& P, const Point3& A, const Point3& B, const Point3& C, double mindist) {
  if(!PointInTri(P, A, B, C)) return false;
  if(DistanceToLine(P, A, B) < mindist) return false;
  if(DistanceToLine(P, B, C) < mindist) return false;
  if(DistanceToLine(P, C, A) < mindist) return false;
  return true;
}

// 判断 P 是否在凸四边形 ABCD（顺时针或逆时针）中，并且到四条边的距离都至少为 mindist。保证 P, A, B, C, D 共面
bool InsideWithMinDistance(const Point3& P, const Point3& A, const Point3& B, const Point3& C, const Point3& D, double mindist) {
  if(!PointInTri(P, A, B, C)) return false;
  if(!PointInTri(P, C, D, A)) return false;
  if(DistanceToLine(P, A, B) < mindist) return false;
  if(DistanceToLine(P, B, C) < mindist) return false;
  if(DistanceToLine(P, C, D) < mindist) return false;
  if(DistanceToLine(P, D, A) < mindist) return false;
  return true;
}
```

增量法求三维凸包

```
#define eps 1e-8
const int N = 310;
struct TPoint
{
    double x,y,z;
    TPoint(){}
    TPoint(double _x,double _y,double _z):x(_x),y(_y),z(_z){} //拷贝构造函数
    TPoint operator+(const TPoint p) {return TPoint(x+p.x,y+p.y,z+p.z);}
    TPoint operator-(const TPoint p) {return TPoint(x-p.x,y-p.y,z-p.z);}
    TPoint operator*(const double p) {return TPoint(x*p,y*p,z*p);}
    TPoint operator/(const double p) {return TPoint(x/p,y/p,z/p);}
    TPoint operator*(const TPoint p) {return TPoint(y*p.z-z*p.y,z*p.x-x*p.z,x*p.y-y*p.x);}//叉积
    double operator^(const TPoint p) {return x*p.x+y*p.y+z*p.z;}//点积
};
struct fac//
{
    int a,b,c;//凸包一个面上的三个点的编号
    //a,b,c 必须是按照逆时针来存储,因为只有这样的话,vec{a->b} x vec{a->c}得到的法向量才指向外面
    bool ok;//该面是否是最终凸包中的面
};
struct T3dhull
```

```
{
    int n;//初始点数
    TPoint ply[N];//初始点
    int trianglecnt;//凸包上三角形数
    fac tri[N];//凸包三角形
    int vis[N][N];//点 i 到点 j 是属于哪个面
    double dist(TPoint a){return sqrt(a.x*a.x+a.y*a.y+a.z*a.z);}//两点长度
    double area2(TPoint a,TPoint b,TPoint c){return dist((b-a)*(c-a));}//三角形面积*2
    double volume6(TPoint a,TPoint b,TPoint c,TPoint d){return (b-a)*(c-a)^(d-a);}//四面体有向体积*6
    double ptoplane(TPoint &p,fac &f)//正：点在面同向
    {
        TPoint m=ply[f.b]-ply[f.a],n=ply[f.c]-ply[f.a],t=p-ply[f.a];
        return (m*n)^t;
    }
    void deal(int p,int a,int b)
    {
        int f=vis[a][b];//与当前面(cnt)共边(ab)的那个面
        fac add;
        if(tri[f].ok)
        {
            if((ptoplane(ply[p],tri[f]))>eps)  dfs(p,f);//如果 p 点能看到该面 f，则继续深度探索 f 的 3 条边，以便更新新的凸包面
            else//否则因为 p 点只看到 cnt 面，看不到 f 面，则 p 点和 a、b 点组成一个三角形。
            {
                add.a=b,add.b=a,add.c=p,add.ok=1;
                vis[p][b]=vis[a][p]=vis[b][a]=trianglecnt;
                tri[trianglecnt++]=add;
            }
        }
    }
    void dfs(int p,int cnt)//维护凸包，如果点 p 在凸包外更新凸包
    {
        tri[cnt].ok=0;//当前面需要删除，因为它在更大的凸包里面

//下面把边反过来(先 b,后 a)，以便在 deal()中判断与当前面(cnt)共边(ab)的那个面。即判断与当头面(cnt)相邻的 3 个面(它们与当前面的共边是反向的，如下图中(1)的法线朝外(即逆时针)的面 130 和 312,它们共边 13，但一个方向是 13,另一个方向是 31)

        deal(p,tri[cnt].b,tri[cnt].a);
        deal(p,tri[cnt].c,tri[cnt].b);
        deal(p,tri[cnt].a,tri[cnt].c);
    }
    bool same(int s,int e)//判断两个面是否为同一面
    {
        TPoint a=ply[tri[s].a],b=ply[tri[s].b],c=ply[tri[s].c];
        return fabs(volume6(a,b,c,ply[tri[e].a]))<eps
            &&fabs(volume6(a,b,c,ply[tri[e].b]))<eps
            &&fabs(volume6(a,b,c,ply[tri[e].c]))<eps;
    }
    void construct()//构建凸包
    {
        int i,j;
        trianglecnt=0;
        if(n<4) return ;
        bool tmp=true;
        for(i=1;i<n;i++)//前两点不共点
        {
            if((dist(ply[0]-ply[i]))>eps)
            {
                swap(ply[1],ply[i]); tmp=false; break;
            }
        }
        if(tmp) return;
        tmp=true;
        for(i=2;i<n;i++)//前三点不共线
        {
            if((dist((ply[0]-ply[1])*(ply[1]-ply[i])))>eps)
            {
                swap(ply[2],ply[i]); tmp=false; break;
            }
        }
        if(tmp) return ;
        tmp=true;
        for(i=3;i<n;i++)//前四点不共面
        {
            if(fabs((ply[0]-ply[1])*(ply[1]-ply[2])^(ply[0]-ply[i]))>eps)
            {
                swap(ply[3],ply[i]); tmp=false; break;
            }
        }
        if(tmp) return ;
        fac add;
        for(i=0;i<4;i++)//构建初始四面体(4 个点为 ply[0],ply[1],ply[2],ply[3])
        {
            add.a=(i+1)%4,add.b=(i+2)%4,add.c=(i+3)%4,add.ok=1;
```

```
                if((ptoplane(ply[i],add))>0)
swap(add.b,add.c);//保证逆时针，即法向量朝外，这样新点才可看到。

vis[add.a][add.b]=vis[add.b][add.c]=vis[add.c][add.a]=trianglecnt;//逆向的有向边保存
                tri[trianglecnt++]=add;
            }
            for(i=4;i<n;i++)//构建更新凸包
            {
                for(j=0;j<trianglecnt;j++)//对每个点判断是否在当前 3 维凸包内或外(i 表示当前点,j 表示当前面)
                {

if(tri[j].ok&&(ptoplane(ply[i],tri[j]))>eps)//对当前凸包面进行判断，看是否点能否看到这个面
                    {
                        dfs(i,j); break;//点能看到当前面，更新凸包的面(递归，可能不止更新一个面)。当前点更新完成后 break 跳出循环

                    }
                }
            }
            int cnt=trianglecnt;//这些面中有一些 tri[i].ok=0，它们属于开始建立但后来因为在更大凸包内故需删除的，所以下面几行代码的作用是只保存最外层的凸包
            trianglecnt=0;
            for(i=0;i<cnt;i++)
            {
                if(tri[i].ok)
                    tri[trianglecnt++]=tri[i];
            }
        }
        double area()//表面积
        {
            double ret=0;
            for(int i=0;i<trianglecnt;i++)

ret+=area2(ply[tri[i].a],ply[tri[i].b],ply[tri[i].c]);
            return ret/2;
        }
        double volume()//体积
        {
            TPoint p(0,0,0);
            double ret=0;
            for(int i=0;i<trianglecnt;i++)

ret+=volume6(p,ply[tri[i].a],ply[tri[i].b],ply[tri[i].c]);
            return fabs(ret/6);
        }
        int facetri() {return trianglecnt;}//表面三角形数
        int facepolygon()//表面多边形数
        {
            int ans=0,i,j,k;
            for(i=0;i<trianglecnt;i++)
            {
                for(j=0,k=1;j<i;j++)
                {
                    if(same(i,j)) {k=0;break;}
                }
                ans+=k;
            }
            return ans;
        }
        //求重心
        TPoint gravity(){
            TPoint ret = TPoint(0.0,0.0,0.0);
            double retv = 0;
            TPoint p0 = ply[0];
            for(int i = 0;i < trianglecnt;i ++){
                double                tmp                =
volume6(ply[tri[i].a],ply[tri[i].b],ply[tri[i].c],p0);
                retv += tmp;
                ret          =          ret          +
(ply[tri[i].a]+ply[tri[i].b]+ply[tri[i].c]+p0)/4.0*tmp;
            }
            return ret/retv;
        }
        //重心到凸包表面最近距离
        double mindis_gra2pla(){
            double ret = inf;
            TPoint center = gravity();
            for(int i = 0;i < trianglecnt;i ++)
                ret          =          min(ret,-
volume6(ply[tri[i].a],ply[tri[i].b],ply[tri[i].c],center)/area2(ply[tri[i].a],ply[tri[i].b],ply[tri[i].c]));
                //凸包的重心必然在凸包内部,所以volume6()求出的有向体积是负数
            return ret;
        }
}hull;
```

判断 8 个点是否为立方体

//原理：立方体中任意一点到两个对角点的距离平方之

```
和为 3*边长平方
struct point{
    ll x;
    ll y;
    ll z;
};

point pt[8];
ll dst[8];
ll dst2[8];
ll dst3[8];

inline double get_dist(int i,int j)
{
    ll ans = (pt[j].x - pt[i].x) * (pt[j].x - pt[i].x);
    ans += (pt[j].y - pt[i].y) * (pt[j].y - pt[i].y);
    ans += (pt[j].z - pt[i].z) * (pt[j].z - pt[i].z);
    return ans;
}

inline bool check()
{
    for(int i = 0; i < 8; ++i)
        dst2[i] = dst[i] = get_dist(i, 7);
    sort(dst, dst + 8);
    ll a2 = dst[1];
    if(a2 != dst[2] || a2 != dst[3])
        return false;
    if(2*a2 != dst[4] || 2*a2 != dst[5] || 2*a2 != dst[6])
        return false;
    if(3*a2 != dst[7])
        return false;

    int ind = 0;
    for(int i = 0; i < 8; ++i)
        if(dst2[i] == 3*a2)
        {
            ind = i;
            break;
        }

        for(int i = 0; i < 8; ++i){
            dst3[i] = get_dist(i, ind);
            if(dst3[i] + dst2[i] != 3 * a2)
                return false;
        }

        return true;
}
```

数据结构

KD 树

```
/*
解决问题类型:
    1.多维空间询问最近邻
    2.多维空间(常见 2,3 维)对于有关空间的问询,例如三维空间中某立方体内有几个点,
      二维空间某一平面有几个点
*/

#include<cstdio>
#include<algorithm>
#include<cmath>
using namespace std;
typedef long long LL;
typedef pair<int, LL> PII;
typedef pair<int, int> pii;
const int maxn = 111222;    //最大节点数
const int maxD = 2;         //最大维度
const int maxM = 12;        //最大问询第几近的点
const LL INF = 4611686018427387903LL;
int now;
struct TPoint {
    int x[maxD];
    void read(int k) {
        for (int i = 0; i < k; ++i)
            scanf("%d", x + i);
    }
} p[maxn];
bool cmp(const TPoint& a, const TPoint& b) {
    return a.x[now] < b.x[now];
}
template<typename T> T sqr(T n) {
    return n * n;
}
struct KDtree {
    int K, n, top;
    int split[maxn];     //split[i]=j:节点 i 以第 j 维度来分割空间
    LL dis2[maxM];       //dis2[i]:离目标点 mp 第 i+1 近的点与之的距离平方
    TPoint stk[maxn];    //stk[i]:离目标点 mp 第 i+1 近的点
    TPoint kp[maxn];     //存储 KD 树中所有的点
```

```
    TPoint mp;                    //目标点
    void build(int l, int r) {                          //建树节点范围是 l->r-1
        if (l >= r)
            return;
        int i, j, mid = (l + r) >> 1;
        LL dif[maxD], mx;
        for (i = 0; i < K; ++i) {
            mx = dif[i] = 0;
            for (j = l; j < r; ++j)
                mx += kp[j].x[i];
            mx /= r - l;
            for (j = l; j < r; ++j)
                dif[i] += sqr(kp[j].x[i] - mx);
        }
        now = 0;
        for (i = 1; i < K; ++i)
            if (dif[now] < dif[i])
                now = i;

        split[mid] = now;
        nth_element(kp + l, kp + mid, kp + r, cmp);
        //stl,可以在 O(n)内将第 mid 大的数放在处理后数组的第 mid 位,mid 位前面的数都小于
        //该数,后面的数都大于该数,但是不保证是有序数列
        build(l, mid);
        build(mid + 1, r);
    }
    void update(const TPoint& p, int M) {           // 更 新 dis2[]
        int i, j;
        LL tmp = dist(p, mp);
        for (i = 0; i < M; ++i)
            if (dis2[i] > tmp) {
                for (j = M - 1; j > i; --j) {
                    stk[j] = stk[j - 1];
                    dis2[j] = dis2[j - 1];
                }
                stk[i] = p;
                dis2[i] = tmp;
                break;
            }
    }
    void nearest_search(int l, int r, int M) {  //在 l->r-1 的范围内搜索
        if (l >= r)
            return;
        int mid = (l + r) >> 1;
        update(kp[mid], M);
        if (l + 1 == r)
            return;
        LL d = mp.x[split[mid]] - kp[mid].x[split[mid]];
        if (d <= 0) {
            nearest_search(l, mid, M);
            if (sqr(d) < dis2[M - 1])
                nearest_search(mid + 1, r, M);
        } else {
            nearest_search(mid + 1, r, M);
            if (sqr(d) < dis2[M - 1])
                nearest_search(l, mid, M);
        }
    }
    void find_nearest(TPoint p, int M) {      //在 kd 树中找离 p 点第 M 近的点
        for (int i = 0; i < M; ++i) {
            dis2[i] = INF;
        }
        mp = p;
        nearest_search(0, n, M);
    }
    LL dist(const TPoint& a, const TPoint& b) {    //a 和 b 两个点的欧几里得距离
        LL res = 0;
        for (int i = 0; i < K; ++i)
            res += sqr<LL>(a.x[i] - b.x[i]);
        return res;
    }
} KD;
//调用时需要赋值:
//    KD.n 赋值为 KD 树初始总共有多少个点;
//    KD.K 代表 KD 树有几维
//    KD.kp[]就是输入的点,p[i]是对原有点的备份(读入时直接可利用 p[i].read()读入)
//    并且更改 maxn,maxD,maxM
//注意,调用里面函数时范围是:l->r-1
```

Splay 树

```
/*指针版本*/
//pnode l = rt->lchild,r = rt->rchild;
//注意我们如果对 rt->lchild 和 rt->rchild 改变的话记得更新 l,r 的值
#define root top->lchild
#define rnode root->rchild
#define keytree rnode->lchild
//rnode is the right child of node root
```

```
int num[maxn];
//无论是线段树还是 splay，所有需要问询的标记都需要在与
//其相关的懒标记更改时也更改
//如 Min 和 add，sum 和 add
typedef struct Node{
    int sz;//sz is the number of nodes in the subtree
    int val;
    bool rev;//mark if this node is reversed
    //注意有的标记是否会影响到旋转方向的判断
    struct Node *lchild,*rchild,*parent;
}node;
typedef node *pnode;//node pointer
pnode top;
//debug 部分************************************
void Treavel(pnode x)
{
    if(x != NULL)
    {
        printf("结点 val：%2d   父结点 val %2d size %d rever %d",x->val,x->parent->val,x->sz,x->rev);
        pnode l = x->lchild,r = x->rchild;
        if(l != NULL){
            printf("  左儿子 val %2d",l->val);
        }
        if(r != NULL){
            printf("  右儿子 val %2d",r->val);
        }
        printf("\n");
        Treavel(x->lchild);
        Treavel(x->rchild);
    }
}
void debug()
{
    printf("root val:%d\n",root->val);
    Treavel(root);
}
// 以 上 是 debug 部 分 *************************************

void free_pointer(pnode rt){//用完就要释放指针空间，否则可能 MLE
    pnode l = rt->lchild,r = rt->rchild;
    if(l != NULL)
        free_pointer(l);
    if(r != NULL)
        free_pointer(r);
    free(rt);
}
pnode NewNode(pnode rt,int val){
    pnode newnode;
    newnode = (pnode)malloc(sizeof(node));
    //when i don't allocate memory for pointer, it may still works,
    //but it overlays some memory randomly.
    newnode->val = val;
    newnode->rev = false;
    newnode->sz = 1;
    newnode->lchild = newnode->rchild = NULL;
    newnode->parent = rt;
    return newnode;
}
void put_up(pnode x){//将孩子状态更新上来
    x->sz = 1;
    pnode l = x->lchild,r = x->rchild;
    if(l != NULL){
        x->sz += l->sz;
    }
    if(r != NULL){
        x->sz += r->sz;
    }
}
void put_down(pnode x){
    if(x->rev){
        //先旋转自己的左右子树，再传递标记
        pnode tmp;
        tmp = x->lchild;
        x->lchild = x->rchild;
        x->rchild = tmp;
        pnode l = x->lchild,r = x->rchild;
        if(l != NULL)
            l->rev ^= x->rev;
        if(r != NULL)
            r->rev ^= x->rev;
        x->rev = false;
    }
}
void build_tree(pnode rt,int l,int r,int kind){ //建树
    if(l > r)return;
    int m = (l+r)/2;
    pnode newnode;
    newnode = NewNode(rt,num[m]);
    if(kind == 0)
        rt->lchild = newnode;
    else
        rt->rchild = newnode;
    build_tree(newnode,l,m-1,0);
```

```
    build_tree(newnode,m+1,r,1);
    put_up(newnode);
}
void init(int n){
    //为了方便处理边界，加上两个边界顶点 left 和 right
    pnode left,right;
    top = NewNode(NULL,0);
    left = NewNode(top,0);
    right = NewNode(left,0);

    top->lchild = left;
    left->rchild = right;
    left->sz = 2;

    build_tree(right,1,n,0);
    put_up(right);
    put_up(left);
}
void Rotate(pnode k,int kind){
    pnode parent,grandpa;
    parent = k->parent;
    grandpa = parent->parent;
    put_down(parent);//put_down(parent) first
    put_down(k);
    if(kind == 1){//rotate clockwise
        parent->lchild = k->rchild;
        if(k->rchild != NULL)
            k->rchild->parent = parent;
        k->parent = grandpa;
        if(grandpa != NULL){
            if(grandpa->lchild == parent)
                grandpa->lchild = k;
            else
                grandpa->rchild = k;
        }
        k->rchild = parent;
        parent->parent = k;
    }
    else{
        parent->rchild = k->lchild;
        if(k->lchild != NULL)
            k->lchild->parent = parent;
        k->parent = grandpa;
        if(grandpa != NULL){
            if(grandpa->lchild == parent)
                grandpa->lchild = k;
            else
                grandpa->rchild = k;
        }
        k->lchild = parent;
        parent->parent = k;
    }
    put_up(parent);
}
void Splay(pnode k,pnode goal){//将 k 节点旋转到 goal 节点的子节点处
    pnode parent,grandpa;
    parent = k->parent;
    grandpa = parent->parent;
    while(parent != goal){
        int t1 = (grandpa->lchild == parent)?0:1;
        int t2 = (parent->lchild == k)?0:1;
        if(grandpa == goal){
            Rotate(k,!t2);
            break;
        }
        if(t1 == t2)//如果想等，先旋转 parent,在旋转 k
            Rotate(parent,!t1),Rotate(k,!t2);
        else
            Rotate(k,!t2),Rotate(k,!t1);
        parent = k->parent;
        grandpa = parent->parent;
    }
    put_up(k);
}
pnode get_kth(pnode rt,int k){
    pnode l = rt->lchild,r = rt->rchild;
    if(l == NULL){
        if(k==0+1)
            return rt;
        return get_kth(r,k-1);
    }
    else if(r == NULL){
        if(k==l->sz+1)
            return rt;
        return get_kth(l,k);
    }
    else{
        if(l->sz+1 == k)
            return rt;
        else if(k <= l->sz)
            return get_kth(l,k);
        else
            return get_kth(r,k-(l->sz+1));
    }
}
pnode del(int l,int r){ //return the pointer of deleted tree
```

```
    Splay(get_kth(root,l),top);
    Splay(get_kth(root,r+2),root);
    pnode rt = keytree;
    rnode->lchild = NULL;
    rt->parent = NULL;// 不可以直接 keytree->parent=NULL,因为此时 keytree==NULL
    put_up(rnode);
    put_up(root);
    return rt;
}
void inser(int l,pnode rt){//insert an interval after l
    Splay(get_kth(root,l+1),top);
    Splay(get_kth(root,l+2),root);
    keytree = rt;
    rt->parent = rnode;
    put_up(rnode);
    put_up(root);
}
pnode minbigger(int p){//求 val 大于 p 的最小的节点
    pnode t = root;
    pnode ans = NULL;
    while(t != NULL){
        if(t->val >= p){//这等于，下面求小于 p 的最大值就不能等于了
            ans = t;
            t = t->lchild;
        }
        else
            t = t->rchild;
    }
    return ans;
}
pnode maxless(int p){//求 val 小于 p 的最大值
    pnode t = root;
    pnode ans = NULL;
    while(t != NULL){
        if(t->val < p){
            ans = t;
            t = t->rchild;
        }
        else
            t = t->lchild;
    }
    return ans;
}
void update(int l,int r,int c){
    //本来在仅有数组的子树中要找的两个节点是第 l-1 个和第 r+1 个节点
    //但是因为我们加了一个边界节点 root 且 root 在数组子树的左边
    //所以我们在 root 为跟的数组中找第 l 和第 r+2 个节点
    //所以 l 和 r+2 节点的左子树的节点个数就是 l-1 和 r+1
    Splay(get_kth(root,l),top);
    Splay(get_kth(root,r+2),root);
    keytree->add += c;
    keytree->sum += (ll)c*keytree->sz;
}
ll query(int l,int r){
    Splay(get_kth(root,l),top);
    Splay(get_kth(root,r+2),root);
    return keytree->sum;
}
//用的时候在读入区间初始值之后先 init()
//对于一颗树的操作完成之后一定要 free_pointer()
```

树状数组

```
/*
6 的原码是 00000110
6 的反码是 11111001
反码+1 以后表示负数
11111010
这就是-6
*/
#include<stdio.h>
#include<string.h>
#include<algorithm>
using namespace std;

#define maxn    5010
long long c[maxn];
init(){
     memset(c,0,sizeof(c));
}
int lowbit(int x)//求得是 c[x]这个数组包括了(a[x]+a[x-1]+...+a[x-lowbit(x)+1]）共 lowbit(x)个 a[]数组的和
{
    return x&(-x);
}
void update(int x,int add)
{
    int i = x;
    while(i <= maxn)//因为要更新的话就需要把结点上层的包含此节点的 c[i]全部更新
                        //c[i]的父节点就是 c[i+lowbit(i)]
```

```
	{
		c[i] += add; //此处因为求逆序数，所以更新为加 1
		i += lowbit(i);
	}
}
int sum(int x) //得到 a[1]->a[n]之和
{
	int ans = 0;
	for(int i = x ; i > 0 ; i -= lowbit(i))//c[i]包含 lowbit(i)个a[]数组，从 a[i]->a[i-lowbit(i)+1],
	                                                    //继续更新 i，然后求 c[i]即求出 sum
		ans += c[i];
	return ans;
}
```

线段树+扫描线求矩形覆盖面积

```
//离散化+扫描线
#include<stdio.h>
#include<iostream>
#include<string.h>
#include<algorithm>
#include<math.h>
#include<queue>
using namespace std;
#define ll long long
const ll Mod=(1e9+7);
const int inf = 0x3f3f3f3f;
const int maxn = 2010;
const int maxm = 4*maxn;

int n;
double Y[maxn],qc[maxn];
struct Line{
	double x;
	double y1,y2;
	int flag;//flag 为 1 代表矩形左边，flag 为-1 代表矩形的右边
}line[maxn];
struct Node{
	int l,r;
	double dl,dr;	//l,r 离散化前的值
	double one,two; //one:覆盖一次的总长度，two:覆盖两次的总长度
	int cover; //该节点被覆盖了几次
}node[maxm];
void init(){
	memset(node,0,sizeof(node));
	memset(line,0,sizeof(line));
	memset(Y,0,sizeof(Y));
}
void build(int id,int l,int r){
	node[id].l = l;
	node[id].r = r;
	node[id].dl = qc[l];
	node[id].dr = qc[r];//此处即离散化了
	if(l+1 == r)return;
	//此处注意，扫描线线段树中的节点指的仍然是坐标系中 y 的值，而非一段 y
	//所以建树的时候是 l->mid;mid->r;而叶子结点长度为 2，即 l+1==r
	int mid = (l+r)/2;
	build(id*2,l,mid);
	build(id*2+1,mid,r);
}
void put_up(int id){
//更新每个节点的 one 和 two 的值
//保证每个节点的 two 的值必然小于等于 one 的值
	if(node[id].cover >= 2)
		node[id].two = node[id].one = node[id].dr - node[id].dl;
	else if(node[id].cover == 1){
		node[id].one = node[id].dr - node[id].dl;
		if(node[id].l+1 == node[id].r) //注意是否为叶子结点
			node[id].two = 0;
		else
			node[id].two = node[id*2].one + node[id*2+1].one;
	}
	else{
		if(node[id].l+1 == node[id].r) //注意是否为叶子结点
			node[id].two = node[id].one = 0;
		else{
			node[id].two = node[id*2].two + node[id*2+1].two;
			node[id].one = node[id*2].one + node[id*2+1].one;
		}
	}
}
void update(int id,Line ln){
	if(node[id].dl == ln.y1 && node[id].dr == ln.y2){
		node[id].cover += ln.flag;
```

```
		put_up(id);		//此处也得更新 id 的 one 和
two
		return;
	}
	int mid = (node[id].l + node[id].r)/2;
	if(ln.y2 <= node[id*2].dr)
		update(id*2,ln);
	else if(ln.y1 >= node[id*2+1].dl)
		update(id*2+1,ln);
	else{
		Line e = ln;
		e.y2 = node[id*2].dr;
		update(id*2,e);
		e.y1 = node[id*2+1].dl;
		e.y2 = ln.y2;
		update(id*2+1,e);
	}
	put_up(id);
}
bool cmpx(Line a,Line b){
	if(a.x == b.x) return a.flag > b.flag;//重边时先入后出
	return a.x < b.x;
}
int main(){
	int T;
	int Case = 1;
	scanf("%d",&T);
	while(T--){
		scanf("%d",&n);
		init();
		int t = 1;
		for(int i = 1;i <= n;i ++){
			double x1,y1,x2,y2;
			scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);
			line[t].x = x1; line[t].y1 = y1; line[t].y2 = y2;
line[t].flag = 1;
			Y[t] = y1;
			t++;
			line[t].x = x2; line[t].y1 = y1; line[t].y2 = y2;
line[t].flag = -1;
			Y[t] = y2;
			t++;
		}
		sort(Y+1,Y+t);
		 //接下来的是去重的步骤
		int cnt = 1;
		double mmax = -1000000;
		for(int i = 1;i < t;i ++){
			if(Y[i] > mmax){
				qc[cnt++] = Y[i];
				mmax = Y[i];
			}
		}
		sort(line+1,line+t,cmpx);
		build(1,1,cnt-1);
		update(1,line[1]);
		double ans = 0;
		for(int i = 2;i < t;i ++){
			ans += node[1].two*(line[i].x-line[i-1].x);
			update(1,line[i]);
		}
		printf("%.2lf\n",ans);
	}
	return 0;
}
```

线段树+扫描线求重叠矩形周长

```
//离散化+扫描线
#include<stdio.h>
#include<iostream>
#include<string.h>
#include<algorithm>
#include<math.h>
#include<queue>
using namespace std;
#define ll long long
const ll Mod=(1e9+7);
const int inf = 0x3f3f3f3f;
const int maxn = 10010;
const int maxm = 4*maxn;

int n;
double Y[maxn],qc[maxn];//qc 存储去重后的 y 值
struct Line{
	double x;
	double y1,y2;
	int flag;
}line[maxn];
struct Node{
	int l,r;
	int dl,dr;//离散化前的坐标
	int lp,rp;//表示这个节点左右两个端点没有被覆盖，
有为 1，无为 0
	double len;//区间被覆盖的总长度
	int cover;//区间被重复覆盖几次
	int num;//区间被多少个线段覆盖，如果两个线段连
```

```
在一起，算一个线段
}node[maxm];
void init(){
	memset(node,0,sizeof(node));
	memset(line,0,sizeof(line));
	memset(Y,0,sizeof(Y));
	memset(qc,0,sizeof(qc));
}
void build(int id,int l,int r){
	node[id].l = l;
	node[id].r = r;
	node[id].dl = qc[l];
	node[id].dr = qc[r];
	if(l+1 == r)return;
	int mid = (l+r)/2;
	build(id*2,l,mid);
	build(id*2+1,mid,r);
}
void put_up(int id){
	if(node[id].cover >= 1){
		node[id].len = node[id].dr-node[id].dl;
		node[id].rp = node[id].lp = 1;
		node[id].num = 1;
	}
	else{
		if(node[id].l+1 == node[id].r){
			node[id].rp = node[id].lp = 0;
			node[id].len = 0;
			node[id].num = 0;
		}
		else{
			node[id].len = node[id*2].len + node[id*2+1].len;
			node[id].lp = node[id*2].lp;
			node[id].rp = node[id*2+1].rp;
			node[id].num = node[id*2].num + node[id*2+1].num;
			if(node[id*2].rp == 1 && node[id*2+1].lp == 1)
				node[id].num--;
		}
	}
}
void update(int id,Line ln){
	if(ln.y1 == node[id].dl && ln.y2 == node[id].dr){
		node[id].cover += ln.flag;
		put_up(id);
		return;
	}
	if(ln.y2 <= node[id*2].dr)
		update(id*2,ln);
	else if(ln.y1 >= node[id*2+1].dl)
		update(id*2+1,ln);
	else{
		Line tmp;
		tmp = ln;
		tmp.y2 = node[id*2].dr;
		update(id*2,tmp);
		tmp.y1 = node[id*2+1].dl;
		tmp.y2 = ln.y2;
		update(id*2+1,tmp);
	}
	put_up(id);
}
bool cmpx(Line a,Line b){
	if(a.x == b.x)
		return a.flag > b.flag;//边重合时，应该先插入，再删除
	return a.x < b.x;
}
int main(){
	int Case = 1;
	while(~scanf("%d",&n)){
		init();
		int t = 1;
		for(int i = 1;i <= n;i ++){
			double x1,y1,x2,y2;
			scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);
			line[t].x = x1; line[t].y1 = y1; line[t].y2 = y2; line[t].flag = 1;
			Y[t] = y1;
			t++;
			line[t].x = x2; line[t].y1 = y1; line[t].y2 = y2; line[t].flag = -1;
			Y[t] = y2;
			t++;
		}
		sort(Y+1,Y+t);
		//接下来的是去重的步骤
		int cnt = 1;
		double mmax = -1000000;
		for(int i = 1;i < t;i ++){
			if(Y[i] > mmax){
				qc[cnt++] = Y[i];
				mmax = Y[i];
			}
		}
		sort(line+1,line+t,cmpx);
```

```
        build(1,1,cnt);
        double ans = 0;
        update(1,line[1]);
        ans += line[1].y2-line[1].y1;
        for(int i = 2;i < t;i ++){
            ans += 2*node[1].num*(line[i].x-line[i-1].x);
            double tlen = node[1].len;
            update(1,line[i]);
            ans += fabs(tlen-node[1].len);
        }
        printf("%d\n",(int)ans);
    }
    return 0;
}
```

数论

GCD

```
//最快速 gcd
int gcd(int a, int b)
{
  while ( b ^= a ^= b ^= a %= b );
  return a;
}
/*
Stein 算法是一种计算两个数最大公约数的算法，
它是针对欧几里德算法在对大整数进行运算时，
需要试商导致增加运算时间的缺陷而提出的改进算法。
*/
typedef long long LL;
LL gcd(LL u,LL v)
{
    if (u == 0) return v;
    if (v == 0) return u;
    // look for factors of 2
    if (~u & 1) // u is even
    {
        if (v & 1) // v is odd
            return gcd(u >> 1, v);
        else // both u and v are even
            return gcd(u >> 1, v >> 1) << 1;
    }

    if (~v & 1) // u is odd, v is even
        return gcd(u, v >> 1);
    // reduce larger argument
    if (u > v)
        return gcd((u - v) >> 1, v);

    return gcd((v - u) >> 1, u);
}
```

Lucas 求组合数取模

```
#include <cstdio>
#include <iostream>
#include <cmath>
#include <cstring>
#include <algorithm>
using namespace std;
#define maxn 100010
typedef long long LL;
LL m,n,p;
LL Pow(LL a,LL b,LL mod)    //快速幂求 a^b(%mod)
{
    LL ans=1;
    while(b)
    {
        if(b&1)
        {
            b--;
            ans=(ans*a)%mod;
        }
        else
        {
            b/=2;
            a=(a*a)%mod;
        }
    }
    return ans;
}
LL C(LL n,LL m)
{
    if(n<m)
        return 0;
    LL ans=1;
    for(int i=1;i<=m;i++)
    {
        ans=ans*(((n-m+i)%p)*Pow(i,p-2,p)%p)%p;
    }
    return ans;
}
```

```
LL Lucas(LL n,LL m)
{
	if(m==0)
		return 1;
	return (Lucas(n/p,m/p)*C(n%p,m%p))%p;
}
int main()
{
	int t;
	scanf("%d",&t);
	while(t--)
	{
		scanf("%lld%lld%lld",&n,&m,&p);
		printf("%lld\n",Lucas(n,m));
	}
	return 0;
}
```

错排公式

```
/*
组合数学。考虑一个有 n 个元素的排列，若一个排列中所有的元素都不在自己原来的位置上，
那么这样的排列就称为原排列的一个错排。
*/
void init()
{
	s[0]=0; s[1]=0; s[2]=1;
	int i;
	for (i=3;i<=100;i++)
		s[i]=(i-1)*(s[i-1]+s[i-2])%Mod;
}
```

高斯消元

```
#include<stdio.h>
#include<algorithm>
#include<iostream>
#include<string.h>
#include<math.h>
using namespace std;

const int MAXN=50;
int A[MAXN][MAXN];//增广矩阵
int x[MAXN];//解集
bool free_x[MAXN];//标记是否是不确定的变元,true 代表不确定
int Free[MAXN],num;//储存自由变元
inline int gcd(int a,int b)
{
	int t;
	while(b!=0)
	{
		t=b;
		b=a%b;
		a=t;
	}
	return a;
}
inline int lcm(int a,int b)
{
	return a/gcd(a,b)*b;//先除后乘防溢出
}
void enumerate(int a[MAXN][MAXN],int k,int var){	//枚举所有的自由变元的情况
//模版中是翻牌的枚举
	int tmp = var-k;
	bool tmp_free[MAXN];
	int tx[MAXN];
	//tmp_free[]和 tx[]分别用来暂存 free_x[]和 x[]
	for(int i = 0;i < (1<<tmp); i++){
		int cnt = 0;//在 i 情况下一共翻了几张牌
		for(int j = 0;j < var;j ++){
			if(free_x[j]==false && x[j])cnt++;//确定取值的变量后面没有记录
			tmp_free[j] = free_x[j];
			tx[j] = x[j];
		}
		int t = i;
		for(int j = 0;j < tmp;j ++){
			tmp_free[Free[j]] = 0;//暂时将自由变元标记为有解
			tx[Free[j]] = (t&1);
			if(tx[Free[j]]){
				cnt++;
			}
			t = t>>1;
		}
		for(int j = k-1;j >= 0;j --){
			int index = 0;
			for(int p = 0;p < var;p ++){
				if(a[j][p] && tmp_free[p])
				//在自由变元被赋值之后，每一行至多有一个未知变量
				//第 j 行左起第 1 个仍不知道解得变量必然就是该未知变量
```

```
                    index = p;
                }
                int tmp = a[j][var];
                for(int p = 0;p < var;p ++)
                    if(a[j][p]) tmp ^= tx[p];
                tx[index] = tmp;
                tmp_free[index] = 0;//tx[index]也暂时有确定解了
                if(tx[index])
                    cnt++;
            }
            result = min(result,cnt);//result 是最小翻牌数
        }
}
// 高斯消元法解方程组(Gauss-Jordan elimination).(-2 表示有浮点数解，但无整数解，
//-1 表示无解，0 表示唯一解，大于 0 表示无穷解，并返回自由变元的个数)
//有 equ 个方程，var 个变元。增广矩阵行数为 equ,分别为 0 到 equ-1,列数为 var+1,分别为 0 到 var.
int Gauss(int a[][],int equ,int var)
{
    int i,j,k;
    int max_r;// 当前这列绝对值最大的行.
    int col;//当前处理的列
    int ta,tb;
    int LCM;
    int temp;
    int free_x_num;
    int free_index;

    num = 0;
    for(int i=0;i<=var;i++)
    {
        x[i]=0;
        free_x[i]=true;
    }

    //转换为阶梯阵.
    col=0; // 当前处理的列
    for(k = 0;k < equ && col < var;k++,col++)
    {// 枚举当前处理的行.
// 找到该 col 列元素绝对值最大的那行与第 k 行交换.(为了在除法时减小误差)
        max_r=k;
        for(i=k+1;i<equ;i++)
        {
            if(abs(a[i][col])>abs(a[max_r][col])) max_r=i;
        }
        if(max_r!=k)
        {// 与第 k 行交换.
            for(j=k;j<var+1;j++) swap(a[k][j],a[max_r][j]);
        }
        if(a[k][col]==0)
        {// 说明该 col 列第 k 行以下全是 0 了，则处理当前行的下一列.
        //k 先--后来,k++,col++
            Free[num++] = col;
            k--;
            continue;
        }
        for(i=k+1;i<equ;i++)
        {// 枚举要删去的行.
            if(a[i][col]!=0)
            {
                LCM = lcm(abs(a[i][col]),abs(a[k][col]));
                ta = LCM/abs(a[i][col]);
                tb = LCM/abs(a[k][col]);
                if(a[i][col]*a[k][col]<0)tb=-tb;//异号的情况是相加
                for(j=col;j<var+1;j++)
                {
                    a[i][j] = a[i][j]*ta-a[k][j]*tb;
                }
            }
        }
    }

    // 1. 无解的情况: 化简的增广阵中存在(0, 0, ..., a)这样的行(a != 0).
    for (i = k; i < equ; i++)
    {   //如果 col 先结束，那么此时第 k~i 行必然是系数部分全部为 0，
        if (a[i][col] != 0) return -1;
    }
    // 2. 无穷解的情况: 在 var * (var + 1)的增广阵中出现(0, 0, ..., 0)这样的行，即说明没有形成严格的上三角阵.
    // 且出现的行数即为自由变元的个数.
    if (k < var)
    {
        // 首先，自由变元有 var - k 个，即不确定的变元至少有 var - k 个.
        for (i = k - 1; i >= 0; i--)
        {
            // 第 i 行一定不会是(0, 0, ..., 0)的情况，因为这样的行是在第 k 行到第 equ 行.
            // 同样，第 i 行一定不会是(0, 0, ..., a), a != 0 的情况，这样的无解的.
```

```
            free_x_num = 0; // 用于判断该行中的不确定的变元的个数，如果超过 1 个，则无法求解，它们仍然为不确定的变元.
            for (j = 0; j < var; j++)
            {
                if (a[i][j] != 0 && free_x[j]) free_x_num++, free_index = j;
            }
            if (free_x_num > 1) continue; // 无法求解出确定的变元.
            // 说明就只有一个不确定的变元 free_index，那么可以求解出该变元，且该变元是确定的.
            temp = a[i][var];
            for (j = 0; j < var; j++)
            {
                if (a[i][j] != 0 && j != free_index) temp -= a[i][j] * x[j];
            }
            x[free_index] = temp / a[i][free_index]; // 求出该变元.
            free_x[free_index] = 0; // 该变元是确定的.
        }
        enumerate();
        return var - k; // 自由变元有 var - k 个.
    }
    // 3. 唯一解的情况: 在 var * (var + 1)的增广阵中形成严格的上三角阵.
    // 计算出 Xn-1, Xn-2 ... X0.
    for (i = var - 1; i >= 0; i--)
    {
        temp = a[i][var];
        for (j = i + 1; j < var; j++)
        {
            if (a[i][j] != 0) temp -= a[i][j] * x[j];
        }
        if (temp % a[i][i] != 0) return -2; // 说明有浮点数解，但无整数解.
        x[i] = temp / a[i][i];
    }
    return 0;
}
```

计算二进制中 1 的个数

```
/*
0x1100010101000 在这个二进制数， 我们可以看到 0，1 交替出现，当我们用这个数减去一个 1 的时候，
 我们发现最这个数最大的影响就是这个数的最后一一个 1 变为了 0， 且这个 1 后面全部的 0 变为了 1，
 于是我们可以想到对数进行“与”操作， 去掉受影响的部分，这样下去直到整个目标数变为 0.
 算法复杂度为 1 的个数
*/
int count(long v)
{
    int number = 0;

    while(v)
    {
        v &= (v-1);
        number++;
    }
    return number;
}
```

矩阵快速幂

```
#define ll long long
const ll Mod = 1e9+7;
struct Matrix{
    ll a[2][2];
}M,ini;
void init(){
    M.a[0][0] = 1;
    M.a[0][1] = 1;
    M.a[1][0] = -1;
    M.a[1][1] = 0;
    ini.a[0][0] = Y;
    ini.a[0][1] = X;
    ini.a[1][0] = ini.a[1][1] = 0;
}
Matrix Multi(Matrix a,Matrix b){
    Matrix c;
    int i,j,k;
    for (i = 0; i < 2; i++){
        for (j = 0; j < 2; j++){
            c.a[i][j] = 0;
            for(k = 0 ; k < 2 ; k++)
                c.a[i][j] = (c.a[i][j] + a.a[i][k] * b.a[k][j])%Mod;
            while(c.a[i][j] < 0)   //对于可能出现的负数
                c.a[i][j] += Mod;
        }
    }
    return c;
}
```

```
Matrix Matrix_pow(int x){
	if(x == 1)return M;
	Matrix tmp = Matrix_pow(x/2);
	if(x % 2 == 0)
		return Multi(tmp,tmp);
	else
		return Multi(Multi(tmp,tmp),M);
}
ll get(int x){
	if(x == 0)return 0;
	if(x == 1)return X;
	if(x == 2)return Y;
	Matrix tmp = Matrix_pow(x-2);
	//上面几行由递推公式决定
	tmp = Multi(ini,tmp); //此处注意，一定是 ini 在左，tmp 在右
	return tmp.a[0][0];
}
//每次先运行 init();
```

小数的大数次幂

```
//求小树的大数次幂模某个值。例如 2^(10^100000)
#include"math.h"
#define LL long long
#define nmax 100000    //大数字符串的长度
int flag[nmax], prime[nmax];
int plen;
void mkprime() {
	int i, j;
	memset(flag, -1, sizeof(flag));
	for (i = 2, plen = 0; i < nmax; i++) {
		if (flag[i]) {
			prime[plen++] = i;
		}
		for (j = 0; (j < plen) && (i * prime[j] < nmax); j++) {
			flag[i * prime[j]] = 0;
			if (i % prime[j] == 0) {
				break;
			}
		}
	}
}
int getPhi(int n) {
	int i, te, phi;
	te = (int) sqrt(n * 1.0);
	for (i = 0, phi = n; (i < plen) && (prime[i] <= te); i++) {
		if (n % prime[i] == 0) {
			phi = phi / prime[i] * (prime[i] - 1);
			while (n % prime[i] == 0) {
				n /= prime[i];
			}
		}
	}
	if (n > 1) {
		phi = phi / n * (n - 1);
	}
	return phi;
}
int cmpCphi(int p, char *ch) {
	int i, len;
	LL res;
	len = strlen(ch);
	for (i = 0, res = 0; i < len; i++) {
		res = (res * 10 + (ch[i] - '0'));
		if (res > p) {
			return 1;
		}
	}
	return 0;
}
int getCP(int p, char *ch) {
	int i, len;
	LL res;
	len = strlen(ch);
	for (i = 0, res = 0; i < len; i++) {
		res = (res * 10 + (ch[i] - '0')) % p;
	}
	return (int) res;
}
int modular_exp(int a, int b, int c) {
	LL res, temp;
	res = 1 % c, temp = a % c;
	while (b) {
		if (b & 1) {
			res = res * temp % c;
		}
		temp = temp * temp % c;
		b >>= 1;
	}
	return (int) res;
}
//a 是底数，ch 是次幂，c 是模
//ch 不能有前置 0，例如 099 不行
void solve(int a, int c, char *ch) {
	int phi, res, b;
```

```
    phi = getPhi(c);
    if (cmpCphi(phi, ch)) {
        b = getCP(phi, ch) + phi;
    } else {
        b = atoi(ch);
    }
    res = modular_exp(a, b, c);
    printf("%d\n", res);
}
```

扩展欧几里得及其应用

```
int Extended_Euclid(int a,int b,int& x,int &y)
{
    if(b==0){
        x=1;
        y=0;
        return a;
    }
    int d=Extended_Euclid(b,a%b,x,y);
    int temp=x;x=y;y=temp-a/b*y;
    return d;
}
//用扩展欧几里得算法解线性方程 ax+by=c;
bool linearEquation(int a,int b,int c,int& x,int &y)
{
    int d=Extended_Euclid(a,b,x,y);
    if(c%d) return false;

    int k=c/d;
    x*=k;y*=k;//求的只是其中一个解
    //给一组特解(x, y), 通解为(x - kb', y + ka').
    //a'=a/gcd(a,b),b'=b/gcd(a,b)
    return true;
}
//用扩展欧几里得求模线性方程 ax=b(mod n)
bool linear_mod_equation (int a, int b, int n, int *sol)
{
    int d, x, y, min_po_sol;
    d = Extended_Euclid(a,n,x,y);
    if (b % d) return false;
    else
    {
        sol [0] = x * ( b / d ) % n ;
        min_po_sol = (sol[0]%(n/d)+(n/d))%(n/d);
        for (int i = 1; i < d; ++i)
            sol[i] = (sol[i - 1] + n / d) % n ;
    }
        return true;
}
```

求欧拉函数

```
int phi(int n)
{
    int ans,i,k;
    if(n==1)
        ans=0;
    else{
        ans=n;
        k=1;
        for(i=2;n!=1;i+=k){
            if(n%i==0){
                ans*=(i-1);ans/=i;   //注意可能爆 int
                while(n%i==0)n/=i;
                i=k;
            }
        }
    }
    return ans;
}
```

素数打表

```
//o(n)时间的一种打表方法
#define N 50000   //质数范围
int prime[N];   //prime[0]=2,prime[1]=3,prime[2]=5,……

void init_prime()
{
    int i,j;
    memset(prime,0,sizeof(prime));
    for(i = 2;i <= sqrt(N*1.0); ++i)
    {
        if(!prime[i])
            for(j = i * i; j < N; j += i)//o->i*i 之间的数字已经判断出是否为素数了
                prime[j] = 1;
    }
    //prime[]在上面是作为标记数组，0 代表是质数，1 代表不是

    //prime[]在下面作为存储素数的数组
    j = 0;
    for(i = 2;i <= N; ++i)
```

```
		if(!prime[i])
			prime[j++] = i;
}
```

位运算 O(n)求解全部组合数

```
/*
这个函数用来求 C(n, k)，其中 comb 就是二进制形式表示的子集，
例如 000111 表示由后三个元素构成的组合或子集，
并且该迭代过程可生成字典序递增的组合，这一特性非常有用。
*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main()
{
	int k=2,n=5;
	int comb = (1<<k) - 1;
	while(comb < 1<<(n))
	{
		//printf("%d\n",comb);//先使用
		int x = comb & -comb, y = comb + x;
		comb = ((comb & ~y) / x >> 1) | y;
	}
	system("pause");
	return 0;
}
```

字符串

O(n)求最长回文子串

```
//as for the initial string, p[i*2+1]-1 is the length of
//palindrome substring whose center is s[i]
int p[2*maxn+3];
char s[2*maxn+3];
int palindrome(char ini_s[]){
	memset(p,0,sizeof(p));
	//change the string
	int ini_len = strlen(ini_s);
	for(int i = 0;i < ini_len; i++){
		s[2*i] = '#';
		s[2*i+1] = ini_s[i];
	}
	s[2*ini_len] = '#';
	s[2*ini_len+1] = '\0';
	//cal p[i]
	int id = -1,mx = 0,len = strlen(s);
	for(int i = 0;i < len;i ++){
		if(mx > i)
			p[i] = min(p[2*id-i],mx-i);
		else
			p[i] = 1;
		while(s[i-p[i]] == s[i+p[i]]){
			p[i]++;
			if(i-p[i]<0 || i+p[i]>=len)
				break;
		}
		if(i+p[i] > mx){
			id = i;
			mx = i+p[i];
		}
	}
	//get the longest palindrome substring
	int ans = -1;
	for(int i = 0;i < len;i ++)
		ans = max(ans,p[i]-1);
	return ans;
}
```

其他

Lca

```
//Tarjan 复杂度 O(n)
//Tarjan:我的版本（略去了 ancestor[]数组）
vector<int>Tree[maxn];
vector<pair<int,int> >Q[maxn];//存储问询,first 是所问的节点标号，second 的是 LCA
vector<pair<int,int> >::iterator it;
int parent[maxn];
bool vis[maxn];
void init(){
	memset(vis,false,sizeof(vis));
	for(int i = 0;i < maxn;i ++){
		Tree[i].clear();
		Q[i].clear();
	}
}
```

```
int find_parent(int x){
	if(parent[x] == x)
		return x;
	return parent[x]=find_parent(parent[x]);
}
void union_set(int x,int y){
	x = find_parent(x);
	y = find_parent(y);
	parent[y] = x;//此处是把 parent[y]赋值为 x
}
void LCA(int x){
	parent[x] = x;
	for(int i = 0;i < Tree[x].size();i ++){
		int tmp = Tree[x][i];
		LCA(tmp);
		union_set(x,tmp);//注意顺序
	}
	vis[x] = true;
	for(it=Q[x].begin(); it != Q[x].end();it++){
		int tmp = it->first;
		if(vis[tmp]){
			it->second = find_parent(tmp);
		}
	}
}
```

```
//Tarjan:网上流传版本
vector<int>Tree[maxn];
vector<pair<int,int> >Q[maxn];//存储问询,first 是所问的节点标号，second 的是 LCA
vector<pair<int,int> >::iterator it;
int ancestor[maxn],parent[maxn];
bool vis[maxn];
void init(){
	memset(vis,false,sizeof(vis));
	for(int i = 0;i < maxn;i ++){
		Tree[i].clear();
		Q[i].clear();
	}
}
int find_parent(int x){
	if(parent[x] == x)
		return x;
	return parent[x]=find_parent(parent[x]);
}
void union_set(int x,int y){
	x = find_parent(x);
	y = find_parent(y);
	parent[y] = x;//此处是把 parent[y]赋值为 x
}
void LCA(int x){
	parent[x] = x;
	ancestor[x] = x;
	for(int i = 0;i < Tree[x].size();i ++){
		int tmp = Tree[x][i];
		LCA(tmp);
		union_set(x,tmp);//注意顺序
		ancestor[find_parent(tmp)] = x;
	}
	vis[x] = true;
	for(it=Q[x].begin(); it != Q[x].end();it++){
		int tmp = it->first;
		if(vis[tmp]){
			it->second = ancestor[find_parent(tmp)];
		}
	}
}
/*
use make_pair()方法
Q[a].push_back(make_pair<int,int>(b,-1));
Q[b].push_back(make_pair<int,int>(a,-1));
*/
```

三分法

```
/*
http://hi.baidu.com/vfxupdpaipbcpuq/item/81b21d1910ea729c99ce33db?qq-pf-to=pcqq.group#
二分法求单调函数；三分法求 凸性函数，例如一元二次方程
*/
double Calc(Type a)
{
	/* 根据题目的意思计算 */
}
void Solve(void)
{
	double Left, Right;
	double mid, midmid;
	double mid_area, midmid_area;
	Left = MIN; Right = MAX;
	while (Left + EPS < Right)
	{
		mid = (Left + Right) / 2;
		midmid = (mid + Right) / 2;
```

```
            mid_area = Calc(mid);
            midmid_area = Calc(midmid);
            // 假设求解最大极值.
            if (mid_area >= midmid_area) Right = midmid;
            else Left = mid;
        }
}
```

输入加速

```
/*
getchar()比 scanf()快
仅用于整数
*/
int input()
{
        char ch=' ';
        while(ch<'0'||ch>'9')ch=getchar();
        int x=0;
        while(ch<='9'&&ch>='0')x=x*10+ch-'0',ch=getchar();
        return x;
}
```

四边形不等式优化 DP

```
#include<stdio.h>
#include<iostream>
#include<string.h>
#include<algorithm>
#include<math.h>
#include<queue>
using namespace std;
#define ll __int64
const ll Mod=(1e9+7);
const int inf = 0x3f3f3f3f;
const int maxn = 1010;
const int maxm = 10100;

int n;
int x[maxn],y[maxn];
int dp[maxn][maxn],s[maxn][maxn];
//s[i][j] is the value of k of the best chioce of dp[i][j]
int dis(int i,int k,int k1,int j){
        return abs(x[k1]-x[i])+abs(y[k]-y[j]);
}
int main(){
        while(scanf("%d",&n) != EOF){
                for(int i = 1;i <= n;i++)
                        scanf("%d%d",&x[i],&y[i]);
                memset(dp,0,sizeof(dp));
                for(int i = 1;i <= n;i ++){
                        s[i][i] = i;
                        dp[i][i] = 0;
                }
                for(int len = 2;len <= n;len ++){
                        //outer loop is the number of nodes that we connect
                        //because the number of nodes of both s[i][j-1] and s[i+1][k] are len-1
                        for(int i = 1;i <= n-len+1;i ++){
                                int j = i+len-1;
                                dp[i][j] = inf;
                                for(int k = s[i][j-1];k <= s[i+1][j]; k++){
                                        if(dp[i][j] > dp[i][k]+dp[k+1][j]+dis(i,k,k+1,j)){
                                                dp[i][j] = dp[i][k]+dp[k+1][j]+dis(i,k,k+1,j);
                                                s[i][j] = k;
                                        }
                                }
                        }
                }
                printf("%d\n",dp[1][n]);
        }
        return 0;
}
```

斜率优化 DP

/*

我们知道，有些 DP 方程可以转化成 DP[i]=f[j]+x[i]的形式，其中 f[j]中保存了只与 j 相关的量。这样的 DP 方程我们可以用单调队列进行优化，从而使得 O(n^2)的复杂度降到 O(n)。

可是并不是所有的方程都可以转化成上面的形式，举个例子：dp[i]=dp[j]+(x[i]-x[j])*(x[i]-x[j])。如果把右边的乘法化开的话，会得到 x[i]*x[j]的项。这就没办法使得 f[j]里只存在于 j 相关的量了。于是上面的单调队列优化方法就不好使了。

这里学习一种新的优化方法，叫做斜率优化，其实和凸包差不多，下面会解释。

举例子说明是最好的！HDU 3507，很适合的一个入门题。

http://acm.hdu.edu.cn/showproblem.php?pid=3507

大概题意就是要输出 N 个数字 a[N]，输出的时候可以连续连续的输出，每连续输出一串，它的费用是 “这串数字和的平方加上一个常数 M”。
我们设 dp[i]表示输出到 i 的时候最少的花费，sum[i]表示从 a[1]到 a[i]的数字和。于是方程就是：
dp[i]=dp[j]+M+(sum[i]-sum[j])^2；
很显然这个是一个二维的。题目的数字有 500000 个，不用试了，二维铁定超时了。那我们就来试试斜率优化吧，看看是如何做到从 O(n^2)复杂度降到 O(n)的。

分析:
我们假设 k<j<i。如果在 j 的时候决策要比在 k 的时候决策好，那么也是就是 dp[j]+M+(sum[i]-sum[j])^2<dp[k]+M+(sum[i]-sum[k])^2。(因为是最小花费嘛，所以优就是小于)
两边移项一下，得到：(dp[j]+num[j]^2-(dp[k]+num[k]^2))/(2*(num[j]-num[k]))<sum[i]。我们把 dp[j]-num[j]^2 看做是 yj，把 2*num[j]看成是 xj。
那么不就是 yj-yk/xj-xk<sum[i]么？　左边是不是斜率的表示？
那么 yj-yk/xj-xk<sum[i]说明了什么呢？　我们前面是不是假设 j 的决策比 k 的决策要好才得到这个表示的？　如果是的话，那么就说明 g[j,k]=yj-jk/xj-xk<sum[i]代表这 j 的决策比 k 的决策要更优。

关键的来了：现在从左到右，还是设 k<j<i，如果 g[i,j]<g[j,k]，那么 j 点便永远不可能成为最优解，可以直接将它踢出我们的最优解集。为什么呢？
我们假设 g[i,j]<sum[i]，那么就是说 i 点要比 j 点优，排除 j 点。
如果 g[i,j]>=sum[i]，那么 j 点此时是比 i 点要更优，但是同时 g[j,k]>g[i,j]>sum[i]。这说明还有 k 点会比 j 点更优，同样排除 j 点。
排除多余的点，这便是一种优化！

所以这样相当于在维护一个下凸的图形，斜率在逐渐增大。
通过一个单调队列来维护。

```
*/
#include<stdio.h>
#include<iostream>
#include<string.h>
#include<algorithm>
#include<math.h>
#include<queue>
using namespace std;
#define ll long long
const ll Mod=(1e9+7);
const int inf = 0x3f3f3f3f;
const int maxn = 500150;
const int maxm = 10100;

int n,m;
int sum[maxn],dp[maxn],q[maxn];
int get_up(int j,int k){      //求斜率分子
    return (dp[j]+sum[j]*sum[j]) - (dp[k]+sum[k]*sum[k]);
}
int get_down(int j,int k){   //斜率分母
    return 2*(sum[j]-sum[k]);
}
int get_dp(int i,int k){
    int ans = dp[k]+(sum[i]-sum[k])*(sum[i]-sum[k])+m;
    return ans;
}
int main(){
    while (scanf("%d%d", &n, &m) != EOF) {
        sum[0] = 0;
        dp[0] = 0;
        for(int i = 1;i <= n;i ++){
            int a;
            scanf("%d",&a);
            sum[i] = sum[i-1]+a;
        }
        int head=0,tail=0;
        q[tail++] = 0;//0 是有含义的，代表所有字母都在第一行的情况
        for(int i=1;i<=n;i++)
        {
            //把斜率转成相乘，注意顺序，否则不等号方向会改变的
            while(head+1<tail && get_up(q[head+1],q[head])<=sum[i]*get_down(q[head+1],q[head]))
                head++;
            //上面循环令 head 为 dp[i]的最优解
            //因为 sum[i]是递增的，这保证了能用斜率优化
            //所以对于 i 来说 head 前面的点被删去，那么对于 i+1，head 前面的点也会被删去
            dp[i]=get_dp(i,q[head]);
            while(head+1<tail && get_up(i,q[tail-1])*get_down(q[tail-1],q[tail-2])<=get_up(q[tail-1],q[tail-2])*get_down(i,q[tail-1]))
                    tail--;
            q[tail++]=i;
            //下面循环保证队列的单调性
```

```
        }
        printf("%d\n",dp[n]);
    }
    return 0;
}
```

网络流

网络流 DINIC

```
const int maxnode = 1000 + 5;
const int maxedge = 1000 + 5;
const int oo = 1000000000;
int node, src, dest, nedge;
int head[maxnode], point[maxedge], next1[maxedge], flow[maxedge], capa[maxedge];//point[x]==y 表示第 x 条边连接 y，head，next 为邻接表，flow[x]表示 x 边的动态值，capa[x]表示 x 边的初始值
int dist[maxnode], Q[maxnode], work[maxnode];//dist[i]表示 i 点的等级
void init(int _node, int _src, int _dest){//初始化，node 表示点的个数，src 表示起点，dest 表示终点
    node = _node;
    src = _src;
    dest = _dest;
    for (int i = 0; i < node; i++) head[i] = -1;
    nedge = 0;
}
void addedge(int u, int v, int c1, int c2){//增加一条 u 到 v 流量为 c1，v 到 u 流量为 c2 的两条边
    point[nedge] = v, capa[nedge] = c1, flow[nedge] = 0, next1[nedge] = head[u], head[u] = (nedge++);
    point[nedge] = u, capa[nedge] = c2, flow[nedge] = 0, next1[nedge] = head[v], head[v] = (nedge++);
}
bool dinic_bfs(){
    memset(dist, 255, sizeof (dist));
    dist[src] = 0;
    int sizeQ = 0;
    Q[sizeQ++] = src;
    for (int cl = 0; cl < sizeQ; cl++)
        for (int k = Q[cl], i = head[k]; i >= 0; i = next1[i])
            if (flow[i] < capa[i] && dist[point[i]] < 0){
                dist[point[i]] = dist[k] + 1;
                Q[sizeQ++] = point[i];
            }
    return dist[dest] >= 0;
}
int dinic_dfs(int x, int exp){
    if (x == dest) return exp;
    for (int &i = work[x]; i >= 0; i = next1[i]){
        int v = point[i], tmp;
        if (flow[i] < capa[i] && dist[v] == dist[x] + 1 && (tmp = dinic_dfs(v, min(exp, capa[i] - flow[i]))) > 0){
            flow[i] += tmp;
            flow[i^1] -= tmp;
            return tmp;
        }
    }
    return 0;
}
int dinic_flow(){
    int result = 0;
    while (dinic_bfs()){
        for (int i = 0; i < node; i++) work[i] = head[i];
        while (1){
            int delta = dinic_dfs(src, oo);
            if (delta == 0) break;
            result += delta;
        }
    }
    return result;
}
//建图前,运行一遍 init()；
//加边时，运行 addedge(a,b,c,0),表示点 a 到 b 流量为 c 的边建成（注意点序号要从 0 开始）
//求解最大流运行 dinic_flow(),返回值即为答案
```

网络流：最小费用最大流 MCMF

```
const int N = 1010;//点
const int M = 4 * 10010;//边
const int inf = 1000000000;
struct Node{//边，点 f 到点 t，流量为 c，费用为 w
    int f, t, c, w;
}e[M];
int next1[M], point[N], dis[N], q[N], pre[N], ne;//ne 为已添加的边数，next，point 为邻接表,dis 为花费，pre 为父亲节点
bool u[N];
void init(){
    memset(point, -1, sizeof(point));
    ne = 0;
}
```

```
void add_edge(int f, int t, int d1,int w){//f 到 t 的一条边，流量为 d1,花费 w,反向边花费-w（可以反悔）
	e[ne].f = f, e[ne].t = t, e[ne].c = d1, e[ne].w = w;
	next1[ne] = point[f], point[f] = ne++;
	e[ne].f = t, e[ne].t = f, e[ne].c = 0, e[ne].w = -w;
	next1[ne] = point[t], point[t] = ne++;
}
bool spfa(int s, int t, int n){
	int i, tmp, l, r;
	memset(pre, -1, sizeof(pre));
	for(i = 0; i < n; ++i)
		dis[i] = inf;
	dis[s] = 0;
	q[0] = s;
	l = 0, r = 1;
	u[s] = true;
	while(l != r) {
		tmp = q[l];
		l = (l + 1) % (n + 1);
		u[tmp] = false;
		for(i = point[tmp]; i != -1; i = next1[i]) {
			if(e[i].c && dis[e[i].t] > dis[tmp] + e[i].w) {
				dis[e[i].t] = dis[tmp] + e[i].w;
				pre[e[i].t] = i;
				if(!u[e[i].t]) {
					u[e[i].t] = true;
					q[r] = e[i].t;
					r = (r + 1) % (n + 1);
				}
			}
		}
	}
	if(pre[t] == -1)
		return false;
	return true;
}
void MCMF(int s, int t, int n, int &flow, int &cost){//起点 s，终点 t，点数 n，最大流 flow，最小花费 cost
	int tmp, arg;
	flow = cost = 0;
	while(spfa(s, t, n)) {
		arg = inf, tmp = t;
		while(tmp != s) {
			arg = min(arg, e[pre[tmp]].c);
			tmp = e[pre[tmp]].f;
		}
		tmp = t;
		while(tmp != s) {
			e[pre[tmp]].c -= arg;
			e[pre[tmp] ^ 1].c += arg;
			tmp = e[pre[tmp]].f;
		}
		flow += arg;
		cost += arg * dis[t];
	}
}
```

//建图前运行 init()

//节点下标从 0 开始

//加边时运行 add_edge(a,b,c,d)表示加一条 a 到 b 的流量为 c 花费为 d 的边（注意花费为单位流量花费）

// 特别注意双向边，运行 add_edge(a,b,c,d),add_edge(b,a,c,d)较好，不要只运行一次 add_edge(a,b,c,d),费用会不对。

//求解时代入 MCMF(s,t,n,v1,v2)，表示起点为 s，终点为 t，点数为 n 的图中，最大流为 v1，最大花费为 v2