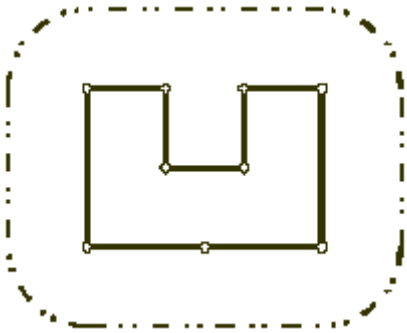
Lab08

E

Description

Once upon a time there was a greedy King who ordered his chief Architect to build a wall around the King's castle. The King was so greedy, that he would not listen to his Architect's proposals to build a beautiful brick wall with a perfect shape and nice tall towers. Instead, he ordered to build the wall around the whole castle using the least amount of stone and labor, but demanded that the wall should not come closer to the castle than a certain distance. If the King finds that the Architect has used more resources to build the wall than it was absolutely necessary to satisfy those requirements, then the Architect will loose his head. Moreover, he demanded Architect to introduce at once a plan of the wall listing the exact amount of resources that are needed to build the wall.



Your task is to help poor Architect to save his head, by writing a program that will find the minimum possible length of the wall that he could build around the castle to satisfy King's requirements.

The task is somewhat simplified by the fact, that the King's castle has a polygonal shape and is situated on a flat ground. The Architect has already established a Cartesian coordinate system and has precisely measured the coordinates of all castle's vertices in feet.

输入格式

Input contains several test cases. The first line of each case contains two integer numbers N and L separated by a space. N ($3 \le N \le 1000$) is the number of vertices in the King's castle, and L ($1 \le L \le 1000$) is the minimal number of feet that King allows for the wall to come close to the castle.

Next N lines describe coordinates of castle's vertices in a clockwise order. Each line contains two integer numbers Xi and Yi separated by a space ($-10000 \le Xi, Yi \le 10000$) that represent the coordinates of ith vertex. All vertices are different and the sides of the castle do not intersect anywhere except for vertices.

Process to end of file.

输出格式

For each case in the input, write to the output file the single number that represents the minimal possible length of the wall in feet that could be built around the castle to satisfy King's requirements. You must present the integer number of feet to the King, because the floating numbers are not invented yet. However, you must round the result in such a way, that it is accurate to 8 inches (1 foot is equal to 12 inches), since the King will not tolerate larger error in the estimates.

样例

input

```
9 100
200 400
300 400
300 300
400 300
400 400
500 400
500 200
350 200
200 200
```

output

```
1628
```

Solution

题目大致意思是 国王有好几个城堡，现在计划绕城堡造一圈城墙，且城墙距离城墙至少得有**r**的距离

理解题意后稍微一分析就会发现是求凸多边形周长，同时由于最近距离的限制，在凸多边形拐角处，是一段圆弧，所有拐角处的圆弧加起来正好是一圈2pi

具体策略即为

先用graham扫描法确定凸点， 放入栈中

再依次取栈，算凸点之间的距离，并求和，

最后再加入圆弧的距离即为所求结果

Code

main.cpp

```cpp
#include <iostream>
#include <math.h>
#include <stack>
#include <algorithm>
using namespace std;
const int MAXN = 1e3 + 10;
const double PI = 3.1415926;

struct point{
```

```
    int x, y;
};

point p0;

bool findp0(point a, point b)
{
    if (a.y != b.y)
        return a.y < b.y;
    else
        return a.x < b.x;
}

int cross(point v1, point v2)
{
    return v1.x * v2.y - v1.y * v2.x;
}

double dist(point a,point b)
{
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}

bool rad(point a, point b)
{
    point v1, v2;
    v1.x = a.x - p0.x;
    v1.y = a.y - p0.y;
    v2.x = b.x - p0.x;
    v2.y = b.y - p0.y;
    int s = cross(v1, v2);
    if (s > 0 || (s==0 && dist(a, p0) < dist(b, p0)))
        return true;
    return false;
}

bool isLeftRotate(point a, point b, point c)
{
    point v1, v2;
    v1.x = b.x - a.x;
    v1.y = b.y - a.y;
    v2.x = c.x - b.x;
    v2.y = c.y - b.y;

    // cout << endl;
    int cha = v1.x * v2.y - v1.y * v2.x;
    if (cha > 0)
        return true;
    else
        return false;
}

point ps[MAXN];
stack<point> st;

void GrahamScan(point *ps, int N)
{
    int i = 0;
```

```
    st.push(ps[i++]);
    st.push(ps[i++]);
    st.push(ps[i++]);

    point pi, pj, pk;
    while (i <= N)
    {
        if (i < N)
            pk = ps[i++];
        else
        {
            pk = p0;
            i++;
        }
        // cout << pk.x << " " << pk.y << endl;
        pj = st.top();
        st.pop();
        pi = st.top();
        st.pop();

        // cout << endl;
        while (!isLeftRotate(pi, pj, pk) && !st.empty())
        {
            pj = pi;
            pi = st.top();
            st.pop();
        }
        // cout << endl;
        st.push(pi);
        st.push(pj);
        st.push(pk);
    }

}

int N, L;

int main()
{
    cin >> N >> L;
    for (int i = 0; i < N; i++)
    {
        cin >> ps[i].x >> ps[i].y;
    }

    sort(ps, ps + N, findp0);
    p0 = ps[0];
    // cout << endl;
    sort(ps + 1, ps + N, rad);
    // cout << endl;

    // for (int i = 0; i < N; i++)
    // {
    //     cout << ps[i].x << " " << ps[i].y << endl;
    // }

    GrahamScan(ps, N);
```

```
    double ans = 0.0;
    ans += PI * 2 * L;

    point p = st.top();
    st.pop();
    point q;
    while (!st.empty())
    {
        q = st.top();
        st.pop();

        ans += sqrt(pow(p.x - q.x, 2) + pow(p.y - q.y, 2));
        p = q;
    }

    cout << round(ans) << endl;
}
```

F

F. Connected Gheeves

单点时限: 2.0 sec

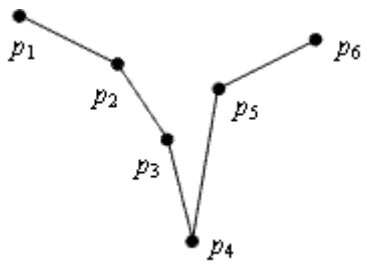
内存限制: 256 MB

Description

Gheeves (plural of gheef) are some objects similar to funnels. We define a gheef as a two dimensional object specified by a sequence of points (p1,p2,...,pn) with the following conditions:

1. 3≤n≤1000
2. If a point pi is specified by the coordinates (xi,yi), there is an index 1<cy2>⋯>yc and yc<yc+1<yc+2<⋯<yn. pc is called the cusp of the gheef.
3. For all 1≤i<c, xi<xc and for all c<i≤n, xi>xc.
4. For 1<i<c, the amount of rotation required to rotate pi−1 around pi in clockwise direction to become co-linear with pi and pi+1, is greater than 180 degrees. Likewise, for c<i<n, the amount of rotation required to rotate pi−1 around pi in clockwise rotation to become co-linear with pi and pi+1, is greater than 180 degrees.
5. The set of segments joining two consecutive points of the sequence intersect only in their endpoints.

For example, the following figure shows a gheef of six points with c=4:



We call the sequence of segments (p1p2,p2p3,...,pn−1pn), the body of the gheef. In this problem, we are given two gheeves P=(p1,p2,...,pn) and Q=(q1,q2,...,qm), such that all x coordinates of pi are negative integers and all x coordinates of qi are positive integers. Assuming the cusps of the two gheeves are connected with a narrow pipe, we pour a certain amount of water inside the gheeves. As we pour water, the gheeves are filled upwards according to known physical laws (**the level of**

water in two gheeves remains the same). Note that in the gheef P, if the level of water reaches miny1,yn, the water pours out of the gheef (the same is true for the gheef Q). Your program must determine the level of water in the two gheeves after pouring a certain amount of water. Since we have defined our problem in two dimensions, the amount of water is measured in terms of area it fills. Note that the volume of pipe connecting cusps is considered as zero.

输入格式

The first number in the input line, t is the number of test cases. Each test case is specified on three lines of input. The first line contains a single integer a(1≤a≤100000) which specifies the amount of water poured into two gheeves. The next two lines specify the two gheeves P and Q respectively, each of the form k x1 y1 x2 y2 ... xk yk where k is the number of points in the gheef (n for P and m for Q), and the xiyi sequence specify the coordinates of the points in the sequences.

输出格式

The output contains t lines, each corresponding to an input test case in that order. The output line contains a single integer L indicating the final level of water, expressed in terms of y coordinates rounded to three digits after decimal points.

样例

input

```
2
25
3 -30 10 -20 0 -10 10
3 10 10 20 0 30 10
25
3 -30 -10 -20 -20 -10 -10
3 10 10 20 0 30 10
```

output

```
3.536
-15.000
```

Solution

题目意思大致是给 两个漏斗一样的东西，在给定水的面积的情况下，漏斗中水的高度是多少（两个漏斗中液面的高度应当一样）

总体策略为：

用二分法枚举水面高度，直到计算得到的水的面积与题给面积之差小于某一精度为止

具体如何计算两个漏斗中水的面积在代码注释中说明

Code

main.cpp

```cpp
#include <iostream>
#include <cmath>
#include <algorithm>
using namespace std;
const int MAXN = 1e3 + 10;
const double eps = 1e-8;
const double pi = acos(-1);
const double inf = ~0u >> 2;

inline int
sign(double x)
{
    if (fabs(x) < eps)
        return 0;
    return x < 0 ? -1 : 1;
}
struct point{
    double x, y;
    point(double x = 0, double y = 0) : x(x), y(y){}
    // struct point operator -(const point &a, const point &b)
    // {
    //     return point(a.x - b.x, a.y - b.y);
    // }
};

//typedef point pointt;
point operator -(point a,point b)
{
    return point(a.x-b.x,a.y-b.y);
}

double dis(point a)
{
    return sqrt(a.x * a.x + a.y * a.y);
}

double cross(point a, point b)
{
    return a.x * b.y - a.y * b.x;
}

double getarea(int n, point *p) // 计算一定序列的点组成的多边形的面积
{
    int i;
    double s = 0;
    for (i = 1; i < n - 1; i++)
    {
        s += cross(p[i] - p[0], p[i + 1] - p[0]);
    }
    return s / 2;
}
```

```
point intersect(point p1, point p2, point p3,point p4) // 返回直线p1->p2 与 p3->p4
的交点
{
    point p;
    double a1, b1, a2, b2, c1, c2, d;
    a1 = p1.y - p2.y;
    b1 = p2.x - p1.x;
    c1 = p1.x * p2.y - p2.x * p1.y;
    a2 = p3.y - p4.y;
    b2 = p4.x - p3.x;
    c2 = p3.x * p4.y - p4.x * p3.y;
    d = a1 * b2 - a2 * b1;
    p.x = (-c1 * b2 + c2 * b1) / d;
    p.y = (-a1 * c2 + a2 * c1) / d;
    return p;
}

bool judge(point a, point b, double y)
{
    return sign(a.y - y) * sign(b.y - y) <= 0;
}

point tmp[MAXN];
double cal(double y, point p[], int n) // 计算容器p中 液面高度为y的 水的面积
{
    int i, j;
    int tot = 0;
    point a = point(0, y), b = point(1, y);  // ab为液面高度所在的直线
    for (i = 1; i < n; i++) // 跳过高于y的点
    {
        if (judge(p[i],p[i+1], y)) // 直到某一段与液面相交
        {
            tmp[tot++] = intersect(p[i], p[i + 1], a, b); // 加入ab与容器的交点
            break;
        }
    }
    for (j = i + 1; j < n; j++)
    { // 低于ab的容器的点全部加入
        tmp[tot++] = p[j];
        if (judge(p[j], p[j+1], y)) // 直到某一段与液面相交
        {
            tmp[tot++] = intersect(p[j], p[j + 1], a, b);
            break;
        }
    }
    double area = getarea(tot, tmp); // 计算水的面积
    return area;
}

point p[MAXN];
point q[MAXN];

int main()
{
    int T;
    cin >> T;
    while(T--)
    {
```

```
        int vol, n, m;
        cin >> vol >> n;
        double minz = inf; // 以两个容器中的最低点为下界
        for (int i = 1; i <= n; i++)
        {
            cin >> p[i].x >> p[i].y;
            minz = min(minz, p[i].y);
        }
        cin >> m;
        for (int i = 1; i <= m;i++)
        {
            cin >> q[i].x >> q[i].y;
            minz = min(minz, q[i].y);
        }
        // cerr << "wrong" << endl;
        double maxz = min(min(p[1].y, p[n].y), min(q[1].y, q[m].y)); // 以两个容器
边界点中的最低点作为上界
        double lef = minz, rig = maxz, mid;
        // cout << lef << " " << rig << endl;
        while (fabs(rig - lef) > eps)
        {
            mid = (rig + lef) / 2.0;
            double s1 = cal(mid, p, n), s2 = cal(mid, q, m);
            if (sign(s1 + s2 - vol) > 0)
                rig = mid;
            else
                lef = mid;
        }
        printf("%.3f\n", lef);
    }
}
```

G

G. 游戏棒

单点时限: 2.0 sec

内存限制: 512 MB

Description

有一个古老的游戏叫做游戏棒，当把一大把细棍从一定的高度一起抛下，使得细棍之间互相碰撞后互相叠在一起，而你的任务则是逐根地将这些细棒向上拾起移出游戏，然而这并不总能做到，如果最后出现了“矛盾”那么之前的努力也会有白费的感觉。

可以拾起的充要条件可以认为是没有任何其他细棒上的点位于当前细棒的正上方。

那么问题来了，现在 kblack 扔下了一把细棒，但他不想玩一个无法完成的游戏，所以判断游戏能否进行完毕的任务就交给你了，因为 kblack 现在去打雀魂了。

输入格式

第一行包含一个整数 n (1≤n≤2 000) 表示细棍的数量。

接下来 n 行每行包含六个整数 ax,ay,az,bx,by,bz 表示三维空间中细棍的两个端点，所有坐标绝对值均小于 100 000 000。

数据保证没有棒相交，没有棒与地面垂直的操作。

输出格式

若所有细棍都能被拾取尽，输出 `Yes`，否则输出 `No`。

样例

input

```
2
0 0 0 1 1 0
0 1 1 1 0 1
```

output

```
Yes
```

input

```
3
0 0 0 1 0 1
1 0 0 0 1 1
0 1 0 0 0 1
```

output

```
No
```

Solution

解法是判断当前是否可以拿掉一个棍子

可以用邻接矩阵 gx[i] [j]表示向量**i**与向量**j**之间的关系 （可以用邻接表简化，减少占用空间）

若**i**在**j**上方，用1表示，在下方，则用-1表示，不相交则用0表示

如何判断哪个向量在上方：

1、先判断两个向量是否相交

2、求出相交向量在xy平面上的交点

3、求解向量在交点上的z值

4、交点上z值较高的向量在上方

总体策略：

在当前向量集中能否找出一个极大值

1、若是能找到一个极大值，便删去该极大值进入下一个循环

2、若是在找极大值的过程中进入了一个循环，便判断为不能取尽游戏棒

Code

main.cpp

```cpp
#include <iostream>
#include <cmath>
#include <algorithm>
#include <vector>
using namespace std;

const int MAXN = 2e3 + 10;
const double eps = 1e-8;
const double pi = acos(-1);

inline int sign(double x)
{
    if (fabs(x) < eps)
        return 0;
    return x < 0 ? -1 : 1;
}

struct point {
    double x, y, z;
};

double xycross(point a, point b)
{
    return a.x * b.y - a.y * b.x;
}

bool xyjudge(point p1, point p2, point p3, point p4) // 计算p1->p2 p3->p4是否相交
{
    point v1, v2, v3, v4;
    v1.x = p3.x - p1.x;
    v1.y = p3.y - p1.y;
    v2.x = p4.x - p1.x;
    v2.y = p4.y - p1.y;
    v3.x = p3.x - p2.x;
    v3.y = p3.y - p2.y;
    v4.x = p4.x - p2.x;
    v4.y = p4.y - p2.y;

    v1.z = v2.z = v3.z = v4.z = 0;
    if (sign(xycross(v1, v2) * xycross(v3, v4)) <= 0 && )
        return true;
    return false;
}
```

```
point intersect(point p1, point p2, point p3, point p4) // 计算向量 p1->p2 p3->p4
的交点
{
    point p;
    double a1, b1, a2, b2, c1, c2, d;
    a1 = p1.y - p2.y;
    b1 = p2.x - p1.x;
    c1 = p1.x * p2.y - p2.x * p1.y;
    a2 = p3.y - p4.y;
    b2 = p4.x - p3.x;
    c2 = p3.x * p4.y - p4.x * p3.y;
    d = a1 * b2 - a2 * b1;
    if (d == 0)
    {
        p.x = p1.x;
        p.y = p1.y;
        p.z = 0;
        return p;
    }
    p.x = (-c1 * b2 + c2 * b1) / d;
    p.y = (-a1 * c2 + a2 * c1) / d;

    p.z = 0;
    return p;
}

int zjudge(point p1, point p2, point p3, point p4)
{
    if (!xyjudge(p1, p2, p3, p4) || !xyjudge(p3, p4, p1, p2))
        return 0;

    point pz = intersect(p1, p2, p3, p4);
    // cout << pz.x << pz.y << pz.z << endl;
    double z1 = p1.z + (p2.z - p1.z) * sqrt(pow(pz.x - p1.x, 2) + pow(pz.y -
p1.y, 2)) / sqrt(pow(p2.x - p1.x, 2)+ pow(p2.y - p1.y, 2));
    double z3 = p3.z + (p4.z - p3.z) * sqrt(pow(pz.x - p3.x, 2)+ pow(pz.y -
p3.y, 2)) / sqrt(pow(p4.x - p3.x, 2)+ pow(p4.y - p3.y, 2));

    // cout << z1 << " " << z3 << endl;
    if (sign(z1 - z3) == 1)
        return 1;
    return -1;
}

point vec[MAXN][2];
vector<int> vect;
int gx[MAXN][MAXN];
vector<int> more[MAXN];

bool judge(vector<int> vect, int N)
{
    if (N == 2)
        return true;

    bool used[MAXN];

    while (N > 2)
    {
```

```
        for (int i = 0; i < N; i++)
            used[i] = false;

        int k = 0;
        used[0] = true;
        bool flag = true;
        while (flag)
        {

            flag = false;
            for (int i = 0; i < N; i++)
            {
                // cout << vect[i] << " " <<  vect[k] << " " << gx[vect[i]]
[vect[k]] << endl;
                //  int *pos = find(&more[i][1], &more[i][more[i][0]], k);
                // cout << find(more[vect[i]].begin(), more[vect[i]].end(),
vect[k]) << endl;
                // auto it = find(more[vect[i]].begin(), more[vect[i]].end(),
vect[k]);
                // if (it != more[vect[i]].end())
                //     cout << *it << endl;
                // cout << more[vect[i]].size() << endl;
                if (i != k && find(more[vect[i]].begin(), more[vect[i]].end(),
vect[k]) != more[vect[i]].end())
                {
                    // cout << endl;
                    if (used[i])
                        return false;
                    else
                    {
                        used[i] = true;
                        k = i;
                        flag = true;
                        // cout << endl;
                        break;
                    }

                }
            }
        }
        swap(vect[N - 1], vect[k]);
        N--;
    }

    // swap(vect[N - 1], vect[k]);
    // cout << endl;
    return true;
}

int test[] = {1, 2, 6};

int main()
{
    int N;
    cin >> N;
    for (int i = 0; i < N; i++)
    {
        cin >> vec[i][0].x >> vec[i][0].y >> vec[i][0].z;
```

```
        cin >> vec[i][1].x >> vec[i][1].y >> vec[i][1].z;
        vect.push_back(i);
    }

    for (int i = 0; i < N; i++)
    {
        for (int j = i + 1; j < N; j++)
        {
            // gx[i][j] = zjudge(vec[i][0], vec[i][1], vec[j][0], vec[j][1]);
            int zj = zjudge(vec[i][0], vec[i][1], vec[j][0], vec[j][1]);
            if (zj == 1)
            {
                more[i].push_back(j);
            } else if (zj == -1)
            {
                more[j].push_back(i);
            }
            // gx[j][i] = gx[i][j] * (-1);
        }
    }
    // cout << gx[0][1] << gx[1][2] << gx[2][0];
    // for (int i = 0; i < 3; i++)
    // {
    //     cout << vec[test[i]][0].x <<" " <<  vec[test[i]][0].y << " "
<<vec[test[i]][0].z << " " <<vec[test[i]][1].x <<" " << vec[test[i]][1].y << " "
<< vec[test[i]][1].z<< endl;
    // }

        bool ans = judge(vect, N);
    if (ans)
        cout << "Yes" << endl;
    else
        cout << "No" << endl;
        // printf();
    // for (int i = 0; i < N; i++)
    // {
    //     for (int j = 0; j < N; j++)
    //     {
    //         if (i == j)
    //             continue;
    //         if (gx[i][j] == -2)
    //         {
    //             gx[i][j] = zjudge(vec[i][0], vec[i][1], vec[j][0], vec[j]
[1]);
    //             gx[j][i] = -gx[i][j];
    //         }
    //         if (gx[i][j] == 1)
    //         {
    //             for (int k = 0; k < N; k++)
    //             {
    //                 if (k == i || k == j)
    //                     continue;

    //                 if (gx[j][k] == -2)
    //                 {
    //                     gx[j][k] = zjudge(vec[j][0], vec[j][1], vec[k][0],
vec[k][1]);
    //                     gx[k][j] = -gx[j][k];
```

```
    //                 }
    //
    //                 if (gx[j][k] == -2)
    //                 {
    //                     gx[j][k] = zjudge(vec[j][0], vec[j][1], vec[k][0],
vec[k][1]);
    //                     gx[k][j] = -gx[j][k];
    //                 }
    //             }
    //         }
    //     }
    // }
}
```