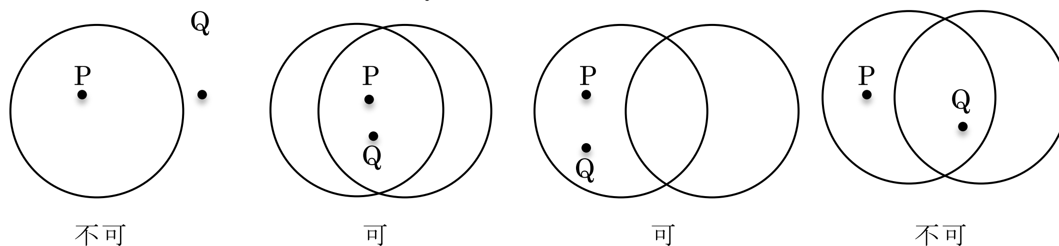
ACM ICPC2014 国内予選　問題 G 円を横切るな

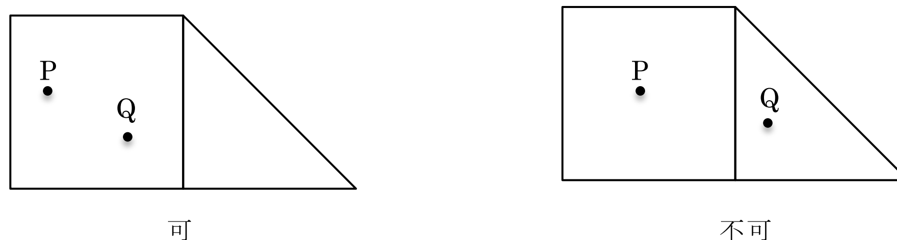
【考察】

- 点 P と点 Q がある円 C に対して、一方が円の内部で他方が円の外部の場合、円を横切ること無く両者を結ぶことはできない。
- 点 P と点 Q が共にある円の内部の場合、他の円に対する包含関係が全て一致していれば円を横切ること無く P,Q を結ぶことができる。




不可　　可　　可　　不可

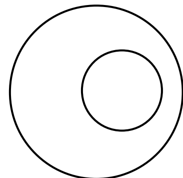
- 点 P と点 Q がすべての円の外にある場合、３つ以上の円が共通点を持たないことから、共通点を持つ２つの円の中心を結ぶ線分で構築される領域を考え、
    - 点 P と点 Q がある領域 F に対して、一方が領域の内部で他方が領域の外部の場合、円を横切ること無く両者を結ぶことはできない。
    - 全ての領域に対して、点 P と点 Q の包含関係が一致していれば、円を横切ること無く両者を結ぶことができる。



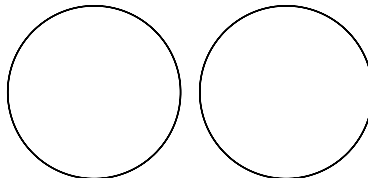

可　　不可

- 点 P が円 C の内部かどうかは、円の中心から点 P までの距離と円の半径を比較すれば判定できる。

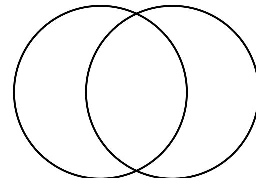
- 二つの円の交差判定は、二つの円の中心間の距離を、二つの円の半径を r1, r2 とすると、
  - $|r1 - r2| > d$： 一方の円が他方の円に含まれるので交差しない
  - $|r1 + r2| < d$： 二つの円は完全に離れているので交差しない
  - 上記以外： 交差する



$|r1 - r2| > d$

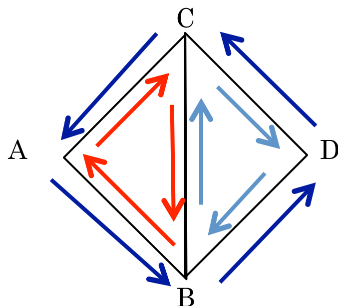


$|r1 + r2| < d$

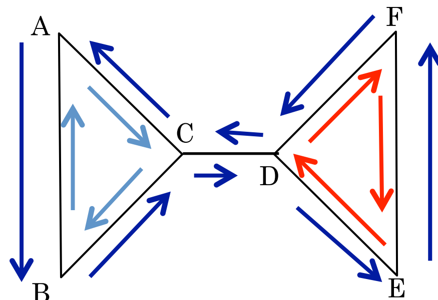


$|r1 + r2| > d$

- 強連結成分の求め方
  1. 各円の中心点は、全て異なる強連結成分（集合）に属するものとしておく。
  2. 全ての二つの円の交差関係を調べ、交差しているならば、これらの円の中心点が属している集合（強連結成分）どうしを併合する。この操作を繰り返すことで、強連結成分が得られる。
- 強連結成分が生成する領域（face）の求め方
  1. 線分を一本選び、その線分に戻ってくるまで時計回りにトレースすると一つの領域が得られる。この時、各線分のトレース方向の向きを使用済みとする。
  2. 未使用の向きの線分を選び、上記と同様に繰り返す。各線分の両方の向きが使用済みになれば、全ての領域が求まっている。
  - 強連結成分の外側の領域も求まる。
  - 各線分は、２つの領域の境界線なので、各々の向きで１回ずつ領域の外周として用いられる。（カットエッジの場合は、外部領域を囲むトレースの中に各々の向きでカットエッジ現れる。）

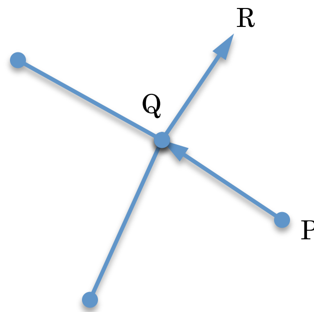



ACBA, BCDB, ABDC

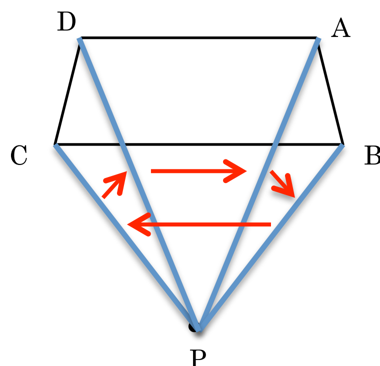


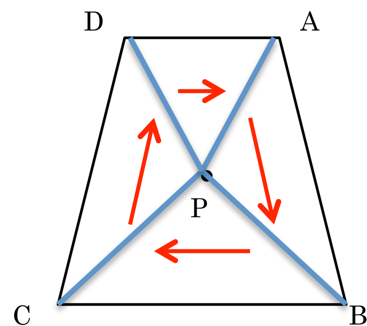

ACBA, DFED, ABCDEFDCA

- 各点の隣接点を角度に基づきソートしておく。ある点Pから現在の点Qに来た場合、ソートされた隣接点でPの次（Pが最後の隣接点の場合は、先頭）の点Rを選ぶことで、時計回りにトレースすることができる。




- 点と線分で囲まれた領域との包含関係のチェック
  1. 点と領域の一つの頂点を結ぶ線分を考える。
  2. 頂点を次の頂点に移動させながら移動時の回転角度を順次加算していく。
  3. 領域の外周を一周した時点で、回転角度の総和がほぼ0ならば点は領域の外に有り、ほぼ±2π ならば点は領域の中にある。




- ２つのベクトル **A** = (Ax, Ay), **B** = (Bx, By) が成す角度θの求め方
  - 内積 **A**・**B** = |**A**||**B**| cos θ =Ax Bx + Ay By なので、

    $$\theta = \mathrm{acos}\left(\frac{AxBx+AyBy}{|A||B|}\right) \quad (0 \leqq \theta \leqq \pi)$$

  - 外積 A×B の大きさ = |A||B| sin θ = Ax By – Ay Bx なので、これが負の時はθを −θとする。

【プログラム例】

```
// ACM-ICPC2014 Domestic Problem G 円を横切るな！
//   http://icpc.iisf.or.jp/past-icpc/domestic2014/#section_G
//   Filename =         pg.c
//   Compile:           cc pg.c
//   Execution:         ./a.out < G0 > G0.out
//   Check: diff G0.ans G0.out

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define MAX_N 100
#define MAX_M 10
#define MAX_XY 10000
#define MAX_R 1000

// #define OPT_PRINT_FACE

int n;                  // 円の個数
int m;                  // 点の組の個数

// 隣接リスト（配列）のセルの構造体
typedef struct {
  int cid;              // 円の ID。中心座標を利用
  int used;             // face 作成に利用済み
  double angle;         // 隣接点への角度
} adj_t;

// 円の構造体
typedef struct {
  int cx, cy;                   // 円の中心座標
  int r;                        // 円の半径
  int nadj;                     // 隣接点の個数
  adj_t adj[MAX_N];             // 隣接点のリスト
} circle;

// face(線分で囲まれた領域)を表す構造体
typedef struct face_type {
  int nnode;                    // face を構成する節点数
  int *nodes;                   // face を構成する節点のリスト。
                                // 要素数は強連結成分の枝数×２＋１必要。
  struct face_type *next;       // 次の face へのポインタ
} face_t;
face_t *face = NULL;            // face の連結リスト
int nface;                      // face の個数
circle c[MAX_N];                // 円のデータを格納
int set[MAX_N];                 // 点（円の中心）が属する集合（強連結成分 scc）
int size[MAX_N];                // 点集合(scc)の大きさ
int first[MAX_N];               // 集合に属する最初の点
int next[MAX_N];                // 次の点
int ls[MAX_N];                  // 集合（scc）に含まれる線分数
int nscc;                       // 強連結成分の個数
```

```
int nls;                        // 線分の総本数

// ２つの円が共通部分を持てば１、さもなくば０を返す関数
int is_connected_circle(int cid1, int cid2)
{
  int dx, dy, sr, dr;
  // (dx, dy) = 円 cid1 の中心から 円 cid2 の中心へのベクトル
  dx = c[cid2].cx - c[cid1].cx;
  dy = c[cid2].cy - c[cid1].cy;
  sr = c[cid2].r + c[cid1].r;                 // 二つの円の半径の和
  dr = c[cid2].r - c[cid1].r;                 // 二つの円の半径の差
  if(dr*dr > dx*dx + dy*dy) return 0;         // 一方の円が他方の円の中に含まれる
  if(sr*sr < dx*dx + dy*dy) return 0;         // 互いの円は他方の円の外
  return 1;                                   // 重なる
}

// 点 (x, y) が円 cid の中なら１、外なら０を返す関数
int is_inside_circle(int cid, int x, int y)
{
  int dx, dy;
  dx = x - c[cid].cx;           // 円の中心からの相対的 x 座標
  dy = y - c[cid].cy;           // 円の中心からの相対的 y 座標
  if(c[cid].r*c[cid].r >= dx*dx + dy*dy) return 1;      // 円の中
  else return 0;                                        // 円の外
}

// 点 (x, y) が pt でさされる領域(face)内の点なら１、さもなくば０を返す関数
// (x, y) から領域の外周の頂点に線分を引き、頂点を順にトレースするときの
// 回転角度を加算していく。一周した時点で、加算した角度が０なら(x,y)は
// 領域の外にあり、加算した角度が±２πなら、領域内のてんである。
int in_face(face_t *pt, int x, int y)           // pt は face へのポインタ
{
  int i;
  double x1, y1, x2, y2;
  double dot, det, d;
  double angsum = 0;                                    // 積算回転角
  double ang;
  for(i=0; i < pt->nnode-1; i++){                       // 領域を構成する各点に対して
    y1 = (double)(c[pt->nodes[i]].cy - y);              // (x1, y1) = (x, y) から
    x1 = (double)(c[pt->nodes[i]].cx - x);              // その点へのベクトル
    y2 = (double)(c[pt->nodes[i+1]].cy - y);            // (x2, y2) = (x, y) から
    x2 = (double)(c[pt->nodes[i+1]].cx - x);            // 次の点へのベクトル
    dot = x1*x2 + y1*y2;                                // 両ベクトルの内積
    det = x1*y2 - y1*x2;                                // 両ベクトルの外積
    d = sqrt((x1*x1+y1*y1)*(x2*x2+y2*y2));              // 両ベクトルの長さの積
    ang = acos(dot/d);                          // (x1, y1) から (x2, y2) への回転角の大きさ
    if(det<0) ang = -ang;         // 外積(sin)が負なら、回転角も負(時計周りの回転)
    angsum += ang;                // 積算回転角に加算
  }
  if(fabs(angsum)<1) return 0;  // 領域外なら、積算回転角 ≒ 0
  else return 1;                // 領域内なら、積算回転角 ≒ ±2π
}
```

```
// 点 P=(px,py)、点 Q=(qx,qy) の間を円に遮られること無く移動可能なら 1
// そうでなければ 0 を返す関数
int check(int px, int py, int qx, int qy)
{
   int flag;              // どれかの円に含まれていることを表すフラグ
   int i, retp, retq;
   face_t *pt;
   // 最初に点 P,Q と円の包含関係をチェックする
   // P と Q の包含関係が一致しなければ、円を横切ってしまう
   // 一致する場合、どれかの円に含まれている場合は、円を横切らない
   // 一致する場合で全ての円の外の場合は、重なる円の中心を結ぶ線分で
   // 構成される領域を求め同じ領域に属しているかのチェックが必要
   flag = 0;
   for(i=0; i<n; i++){                        // 各円に対して
      retp = is_inside_circle(i, px, py);                 // 点 P を含めば 1
      retq = is_inside_circle(i, qx, qy);                 // 点 Q を含めば 1
      if(retp != retq) return 0;    // 一方は円に含まれ他方は含まれないので移動不可
      if(retp == 1) flag = 1;       // 少なくとも一つの円の中
   }
   if(flag == 1) return 1;          // 同一円にのみ含まれるので移動可能
   // どの円にも含まれない
   // ここで重なる円の中心を結ぶ線分からなる領域チェック
   pt = face;                                 // pt = 最初の領域へのポインタ
   while(pt){                                 // 各領域に対して
      retp = in_face(pt, px, py);             // 点 P を含めば 1
      retq = in_face(pt, qx, qy);             // 点 Q を含めば 1
      //      printf("<%d %d>¥n", retp, retq);
      if(retp != retq) return 0;              // 一方の点しか含まれないので移動不可
      pt = pt->next;                          // pt = 次の領域へのポインタ
   }
   return 1;              // 全ての領域への包含関係が一致するので、移動可能
}

void merge(int c1, int c2)         // 2点(円の中心)が属する集合を併合
{
   int cc;
   int s1, s2;
   s1 = set[c1];                              // s1 = 点 c1 が属する集合
   s2 = set[c2];                              // s2 = 点 c2 が属する集合
   if(s1==s2){                                // 両者が等しい場合（このケースは発生しない？）
      ls[s1]++;                               // その集合に線分を追加
      return;
   }
   nscc--;                // 2つの集合の併合により強連結成分数は1減少
   // s1, s2 に対し、小さい方の集合を大きい方の集合に併合する
   if(size[s1] < size[s2]){         // s1 の方が小さいので s1 を s2 に merge
      cc = first[s1];               // cc = 集合 s1 の最初の要素
      while(1){                     // 集合 s1 の各要素に対して
         set[cc] = s2;              // cc が属する集合を s2 に変更
         if(next[cc] == -1) break;
```

```
            cc = next[cc];                  // cc = 集合 s1 の次の要素
        }
        next[cc] = first[s2];           // 旧 s1 の後ろに集合 s2 のリストを接続
        first[s2] = first[s1];          // 旧 s1 の先頭が、新集合 s2 の先頭
        size[s2] += size[s1];           // s1 の要素数を s2 の要素数に加算
        ls[s2] += ls[s1] + 1;           // s1 の線分数を s2 の線分数に加算し 1 増やす
        size[s1] = 0;                   // s1 を空集合にする(要素数=0)
        first[s1] = -1;                 // s1 の最初の要素は無い
        ls[s1] = 0;                     // s1 に属する線分も無い
    }
    else {                              // s2 の大きさは s1 以下なので s2 を s1 に merge
        cc = first[s2];                 // cc = 集合 s2 の最初の要素
        while(1){                       // 集合 s2 の各要素に対して
            set[cc] = s1;               // cc が属する集合を s1 に変更
            if(next[cc] == -1) break;
            cc = next[cc];              // cc = 集合 s2 の次の要素
        }
        next[cc] = first[s1];           // 旧 s2 の後ろに集合 s1 のリストを接続
        first[s1] = first[s2];          // 旧 s2 の先頭が、新集合 s1 の先頭
        size[s1] += size[s2];           // s2 の要素数を s1 の要素数に加算
        ls[s1] += ls[s2] + 1;           // s2 の線分数を s1 の線分数に加算し 1 増やす
        size[s2] = 0;                   // s2 を空集合にする(要素数=0)
        first[s2] = -1;                 // s2 の最初の要素は無い
        ls[s2] = 0;                     // s2 に属する線分も無い
    }
}

// 隣接点を角度の昇順でソートする為の比較関数
int cmp_adj(const void *p1, const void *p2)
{
    adj_t *ap1 = (adj_t *)p1;                          // 隣接点 1 へのポインタ
    adj_t *ap2 = (adj_t *)p2;                          // 隣接点 2 へのポインタ
    if(ap1->angle < ap2->angle) return -1;             // 隣接点 1 の角度<隣接点 2 の角度
    else if(ap1->angle > ap2->angle) return +1;        // 隣接点 1 の角度>隣接点 2 の角度
    else return 0;                                     // 隣接点 1 の角度=隣接点 2 の角度
}

// 指定された強連結成分で構成される領域(face)を求める関数
void find_face(int setid)           // setid = 強連結成分の ID(点集合の ID)
{
    int cc;
    int i,j,k;
    int se1, se2;                            // 領域探索開始時の最初の線分(s1-s2)
    int c1, c2, c3;
    face_t *new;
    nface = 0;                               // 領域数を 0 に初期化
    cc = first[setid];                       // cc = 強連結成分の最初の点
    while(cc != -1){
        for(i=0; i<c[cc].nadj; i++){
            if(c[cc].adj[i].used == 0){ // cc の未使用な隣接点に対して
                // 最初の線分として s1,s2 にセットし、
                // かつ、処理中の線分として c1,c2 にセット
```

```
                c1 = se1 = cc; c2 = se2 = c[cc].adj[i].cid;
                c[cc].adj[i].used = 1;                  // この隣接点を使用済みにする
#ifdef OPT_PRINT_FACE
                printf("face: %d %d", c1, c2);
#endif
                new = (face_t *)malloc(sizeof(face_t));       // 領域用データ構造のメモリ確保
                // 領域の外周点を格納する配列を確保。
                // 強連結成分に含まれる線分数が e=ls[setid]の場合、
                // 配列の大きさは最悪で 2e+1 必要
                new->nodes = (int *)malloc(sizeof(int)*(2*ls[setid]+1));
                new->nnode = 2;                         // まず最初の 2 点を登録
                new->nodes[0] = c1; new->nodes[1] = c2;
                new->next = face; face = new;  // この領域を領域リストに追加
                while(1){                  // この領域の外周を時計周りに構築していく
                  for(j=0; j<c[c2].nadj; j++){      // 点 c2 の隣接点で
                    if(c[c2].adj[j].cid == c1){     // c1 を探す
                      // c2 の隣接点の中で c1 の次の隣接点が構築中の領域の次の外周点となる
                      c3 = c[c2].adj[(j+1)%c[c2].nadj].cid;
                      c[c2].adj[(j+1)%c[c2].nadj].used = 1;              // その点を使用済みに
                      break;
                    }
                  }
                  // 最初の線分に戻ってきたら終了
                  if(c2 == se1 && c3 == se2) break;
                  new->nodes[new->nnode++] = c3;         // 点 c3 を外周点に追加し
                  c1 = c2; c2 = c3;                      // 線分 c2-c3 を線分 c1-c2 とし
#ifdef OPT_PRINT_FACE
                  printf(" %d", c2);
#endif
                  // トレースを継続
                }
#ifdef OPT_PRINT_FACE
                printf("¥n");
#endif
                nface++;  // 新たな領域が見つかったので領域数を 1 増やす
            }
        }
        cc = next[cc];     // 強連結成分に含まれる次の点から領域探索続行
    }
}

void init()
{
    int i, j, c1, c2, cc;
    // 重なっている円の中心同士は線分で結ばれていると考えて強連結成分を求める
    for(i=0; i<n; i++){  // まず最初に点を 1 個だけ含む n 個の集合を構築
        set[i] = i;                 // 点 i は集合 i に属している
        size[i] = 1;                // 集合 i の要素数は 1
        first[i] = i;               // 集合 i の最初の点は i
        next[i] = -1;               // 点 i と同じ集合に属する次の要素は無い
        c[i].nadj = 0;              // 点 i の隣接点は無い
        ls[i] = 0;                  // 集合 i に属する線分は無い
```

```
    }
    nls = 0;                        // 最初、線分数は 0
    nscc = n;                       // 最初、強連結成分の個数は n
    for(c1=0; c1<n; c1++){
      for(c2=c1+1; c2<n; c2++){  // 異なる２円 c1, c2 に対して
        if(is_connected_circle(c1, c2)){        // 2 円が重なるなら
            nls++;                                  // ２円の中心を結ぶ線分を追加
            merge(c1, c2);                          // c1, c2 を含む強連結成分をマージ
            c[c1].adj[c[c1].nadj].cid = c2;     // c1 の隣接点に c2 を追加
            c[c1].adj[c[c1].nadj].used = 0;   // 線分 c1-c2 は、最初は未使用
            c[c1].adj[c[c1].nadj++].angle = // 線分 c1-c2 の角度を求める
              atan2((double)(c[c2].cy-c[c1].cy),(double)(c[c2].cx-c[c1].cx));
            c[c2].adj[c[c2].nadj].cid = c1;     // c2 の隣接点に c1 を追加
            c[c2].adj[c[c2].nadj].used = 0;   // 線分 c2-c1 は、最初は未使用
            c[c2].adj[c[c2].nadj++].angle =  // 線分 c2-c1 の角度を求める
              atan2((double)(c[c1].cy-c[c2].cy),(double)(c[c1].cx-c[c2].cx));
        }
      }
    }
    for(i=0; i<n; i++){              // 各点(円の中心)に対して
      // 隣接点を角度の昇順にソート
      qsort(c[i].adj, c[i].nadj, sizeof(adj_t), cmp_adj);
    }
    //   printf("nls = %d, nscc = %d¥n", nls, nscc);
    for(i=0; i<n; i++){
      if(size[i]>2){                 // 要素数が３以上の強連結成分に対して
        //         printf("[%d]¥n", ls[i]);
        find_face(i);                // 領域のデータを構築
      }
    }
    //   printf("nnode=%d, nls=%d, nscc=%d, nface=%d¥n", n, nls, nscc, nface);
}

int main(void)
{
    int i, j, first;
    int px, py, qx, qy, ret;
    face_t *pt, *npt;
    while(1){
      scanf("%d %d", &n, &m);                  // 円の個数 n と点対の個数 m を入力
      if(n==0 && m==0) break;                  // m=n=0 なら終了
      face = NULL;                             // 最初の領域へのポインタを初期化
      for(i=0; i<n; i++)                       // 各円の中心座標と半径を入力
        scanf("%d %d %d", &c[i].cx, &c[i].cy, &c[i].r);
      init();                                  // データ構造の初期化
      first = 1;                               // 最初の点対
      for(j=0; j<m; j++){                      // 各点対に対して
        scanf("%d %d %d %d", &px, &py, &qx, &qy); // それらの座標を入力
        if(first == 0) printf(" ");            // ２番目以降はまず空白を出力
        else first = 0;                        // 次は２番目以降
        ret = check(px, py, qx, qy);           // 点 P,Q 間の移動可能性をチェック
        if(ret == 1) printf("YES");            // 移動可能なら「YES」
```

```
        else printf("NO");                    // 移動不可能なら「NO」
      }
      printf("¥n");                           // 改行
      pt = face;                              // 領域を表すデータ構造で使用したメモリを解放
      while(pt){                              // pt = 領域へのポインタ
        npt = pt->next;                       // npt = 次の領域へのポインタ
        free(pt->nodes);                      // まず、領域を構成する節点リストの配列を解放
        free(pt);                             // 次に、領域自体のデータ構造を解放
        pt = npt;                             // pt = 次の領域へのポインタ
      }
    }
}
```