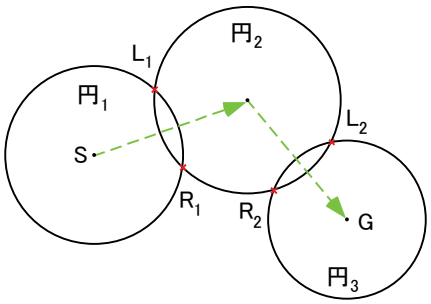
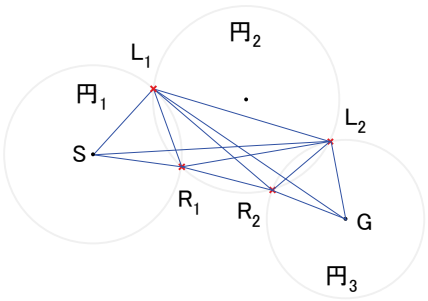
ACM ICPC 国内予選問題 E 「鎖中経路」

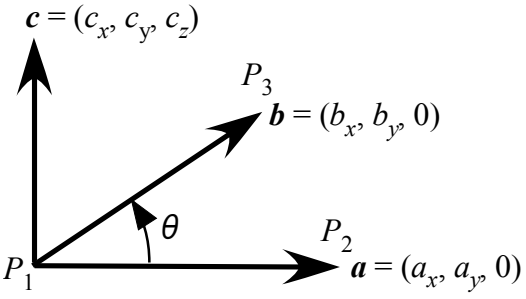
- 最初の円の中心、最後の円の中心、二つの円の交点からなるグラフを考える。グラフ上の任意の 2 点に対して、その 2 点を結ぶ線分が鎖内にあるとき、その 2 点を結ぶ辺を設け、線分の長さをその辺の長さとする。その上で、最初の円の中心を出発点、最後の円の中心を目的地とする最短経路長をダイクストラ法などで求めれば良い。
- 右図に示すように、円 $_i$ と円 $_{i+1}$ の交点のうち進行方向(円 $_i$ から円 $_{i+1}$)右手の点を R_i, 左手の点を L_i とする。また円 $_1$ の中心を出発始点 S, 円 $_n$ の中心を目的地 G とする（図では n=3)。このとき、これらの点の間を直線で結ぶ鎖中経路が存在するための条件は



  - 線分 S → L_i 又は R_i : $1 \leqq \forall j \leqq i\text{-}1$ に対して、点 L_j は線分の左側にあり、点 R_j は線分の右側にある。
  - 線分 L_i 又は R_i → G : $i+1 \leqq \forall j \leqq n\text{-}1$ に対して、点 L_j は線分の左側にあり、点 R_j は線分の右側にある。
  - 線分 L_i 又は R_i → L_j 又は R_j $(i \neq j)$: $i+1 \leqq \forall k \leqq j\text{-}1$ に対して、点 L_j は線分の左側にあり、点 R_j は線分の右側にある。
  - 線分 L_i → R_i 又は R_i → L_i : 常に存在
- これより、右図のようなグラフが得られる。グラフの各辺の距離は、幾何学的距離である。このグラフに対して、S から G への最短経路長を求めれば良い。ダイクストラ法により最短経路長を求めることを考慮し、このグラフを、各節点に対して隣接節点のリストを与える形で表現する。



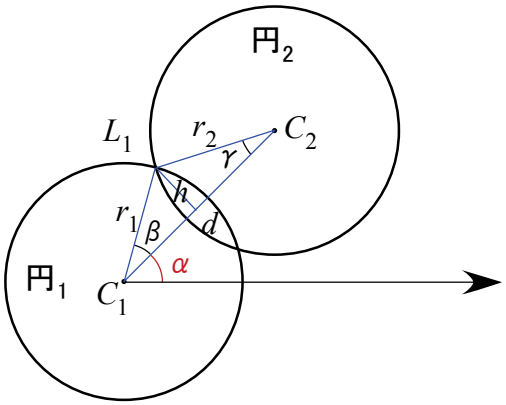
- このアルゴリズムでは、グラフの構成に $O(n^3)$の時間がかかるが、n は 100 以下なので問題無いと考えられる。
- <u>ベクトルに対する点の位置関係(右側または左側)の判定方法：</u> 右図に示すように、ベクトル $\boldsymbol{a}$ に対してベクトル $\boldsymbol{b}$ が左方向(半時計方向)を向いている時は、c_z が正になり、右方向(反時計回り)を向いている場合は、c_z が負になる。ここで、3 点 $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $P_3 = (x_3, y_3)$ に対して、ベクトル $\boldsymbol{a} = (P_1, P_2)$, $\boldsymbol{b} = (P_1, P_3)$ とすると、$a_x = x_2 - x_1$, $a_y = y_2 - y_1$, $b_x = x_3 - x_1$, $b_y = y_3 -$



$$\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$$
$$\|\boldsymbol{c}\| = \|\boldsymbol{a}\| \cdot \|\boldsymbol{b}\| \cdot \sin\theta$$
$$c_x = c_y = 0,\ c_z = a_x \cdot b_y - a_y \cdot b_x$$

y_1 となる。この値を用いて c_z を求めることにより、$c_z > 0$ なら点 P_3 はベクトル(P_1, P_2)に対して左側に存在し、$c_z < 0$ なら点 P_3 はベクトル(P_1, P_2)に対して右側に存在することがわかる。

- 円と円の交点を求める方法：　円 1 の中心を $C_1 = (x_1, y_1)$, 円 2 の中心を$C_2 = (x_2, y_2)$とする。線分 (C_1, C_2) の長さを d, x軸となす角を α [1]とすると、



$$d = \mathrm{sqrt}\left((x_2 - x_1)^2 + (y_2 - y_1)^2 \right)$$

$$\cos\alpha = (x_2 - x_1) / d$$

となる。

また、2 円の左側の交点を L_1 とし、線分(C_1, L_1)が線分(C_1, C_2)となす角度を β、線分(C_2, C_1)が線分(C_2, L_1)となす角度を γ、点 L_1 から線分(C_1, C_2)に下した垂線の長さを h とすると、$h = r_1 \sin\beta = r_2 \sin\gamma$,　$d = r_1 \cos\beta + r_2 \cos\gamma$　となる。これより、γ を消去して $\cos\beta$ を求めると、$\cos\beta = (d_2 + r_1^2 - r_2^2)/(2\, d\, r_1)$ を得る。したがって点 L_1 の座標(x, y)は $x = r_1 \cos(\alpha + \beta)$,　$y = r_1 \sin(\alpha + \beta)$　となる。

【プログラム例】

```
// ACM ICPC 2012 国内予選問題 E 鎖中経路
// http://www.cs.titech.ac.jp/icpc2012/icpc2012-mondai/icpc2012/contest/E_ja.html
// Filename = pe.c
// Compile:    cc pe.c -lm
// Execution: ./a.out < E0 > E0.result
// Check:       ./check E0.ans E0.result
//
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define MAX_N 100
#define INF 1000000000

//  2 点間の距離を求めるマクロ
#define DISTANCE(p, q) sqrt((p.x - q.x)*(p.x - q.x) + (p.y - q.y)*(p.y - q.y))

typedef struct {                    // 点を表す構造体
   double x;                        // 点の x 座標
   double y;                        // 点の y 座標
} point_t;

typedef struct node_list {          // 隣接節点のリスト
   int node_id;                     // 節点番号
   double dist;                     // 基準節点からの距離
```

[1] ここでは簡単のため $\alpha > 0$ の場合についてのみ説明する。

```
   struct node_list *next;    // 次の隣接節点
} node_list_t;

typedef struct {              // 節点情報を表す構造体
   point_t pos;               // 節点の座標
   double cost;               // この節点までの現時点での最小距離
   int heap_pos;              // この節点のヒープ上での場所
   node_list_t *adj;          // 隣接節点のリスト
} node_t;

int n;                        // 円の個数 3 <= n <= 100
int nh;                       // ヒープ内の節点数
int start;                    // スタート節点番号
int goal;                     // ゴール節点番号
node_t node[MAX_N * 2];       // 節点情報
int heap[MAX_N * 2];          // 最短距離を求めるためのヒープ

// 節点 nid の値が小さくなったときにヒープを再構築する
void update_heap(int nid)
{
   int p,c,t;
   c = node[nid].heap_pos;    // 節点 nid のヒープ内の位置
   while(c>0){                // c が根接点で無い間、以下を繰り返す
      p = (c-1)/2;            // p = 親節点
      // 親節点のコストの方が小さければヒープの更新終了
      if(node[heap[p]].cost <= node[heap[c]].cost) break;
      // 親節点と交換
      node[heap[p]].heap_pos = c;      // 節点 heap[p]のヒープ上の新たな位置は c
      node[heap[c]].heap_pos = p;      // 節点 heap[c]のヒープ上の新たな位置は p
      // heap[p] <==> heap[c]
      t = heap[p];
      heap[p] = heap[c];
      heap[c] = t;
      c = p;           // 親節点が変更されたので、親節点を現在の節点として繰り返す
   }
}

// ヒープに節点 nid を挿入する
void ins_heap(int nid)
{
   if(node[nid].heap_pos == -1){      // ヒープ内にその節点が未登録なら
      heap[nh] = nid;                 // ヒープの最後の位置に nid を挿入
      node[nid].heap_pos = nh++;      // 節点 nid のヒープ上の位置を設定し、
   }                                  // ヒープ内データ数を 1 増やす
   update_heap(nid);                  // 節点 nid からヒープを再構築
}

// ヒープの現在の根節点の id を返す。その根節点は削除する。
int del_heap()
{
   int ret;
   int p, lc, rc, c, t;
```

```
  ret = heap[0];                // 現在の根節点
  node[ret].heap_pos = -1; // 根節点はヒープから削除
  nh--;                         // ヒープ内データ数を 1 減らす
  if(nh){                       // ヒープが空でなければ
    heap[0] = heap[nh];                 // ヒープの最後の節点を根節点へ移動
    node[heap[0]].heap_pos = 0;     // その節点のヒープ内位置を 0(根節点)に設定
    p = 0;          // 節点 p より down heap でヒープを再構築する。最初 p は根節点
    while(1){
      lc = 2*p+1; rc = 2*p+2;          // lc は p の左の子（の位置）、rc は右の子（の位置）
      if(lc >= nh) break;   // 子供が無い場合は処理終了
      c = lc;                   // c は左の子
      if(rc < nh){              // 右の子も存在する場合、小さい方を c とする
        if(node[heap[rc]].cost < node[heap[lc]].cost)
          c = rc;
      }
      // p の値の方が、子供の値よりも小さければ処理終了
      if(node[heap[p]].cost <= node[heap[c]].cost)
        break;
      // 節点 heap[p] と節点 heap[c] のヒープ内位置を交換
      node[heap[p]].heap_pos = c;
      node[heap[c]].heap_pos = p;
      t = heap[p];
      heap[p] = heap[c];
      heap[c] = t;
      p = c;                    // 交換された子の位置を p として down heap を繰り返す。
    }
  }
  return ret;       // 削除した根接点の id を返す
}

// ベクトル (p1 -> p2) から見た 点 p3 の方向を求める
// 1 -> 左側、0 -> ベクトル上、-1 -> 右側
int dir_judge(point_t p1, point_t p2, point_t p3)
{
  double vx2, vy2, vx3, vy3;
  double ep;
  vx2 = p2.x - p1.x; vy2 = p2.y - p1.y;
  vx3 = p3.x - p1.x; vy3 = p3.y - p1.y;
  ep = vx2*vy3 - vx3*vy2;
  if(ep < 0) return -1;
  if(ep > 0) return 1;
  return 0;
}

// 節点 s から節点 t へ円内を通って直線で移動可能かを判定。1 -> 可能、 0 -> 不可能
int is_possible(int s, int g)
{
  int sl, gl;           // 節点 s, g のレベル
  int level, rid, lid;
  sl = (s+1)/2;                     // 節点 s のレベル
  gl = (g+1)/2;                     // 節点 g のレベル
  if(sl == gl) return 0;            // 同一レベル内の移動は無意味
```

```
  if(gl - sl == 1) return 1;              // 隣接レベル間は常に移動可能
  for(level = sl+1; level <= gl-1; level++){      // s と g の間の交点の各レベルに対し
    lid = 2*level; rid = lid - 1;         // lid = そのレベルの左側交点
                                          // rid = そのレベルの右側交点
    // 右側交点が左側にあったり、左側交点が右側にある場合、直線移動は不可
    if(dir_judge(node[s].pos, node[g].pos, node[rid].pos) > 0) return 0;
    if(dir_judge(node[s].pos, node[g].pos, node[lid].pos) < 0) return 0;
  }
  return 1;          // s と g の間の全てのレベルの右交点は右側、左交点は左側にあるので
                     // s から g へ円の内部のみを通って直線移動可能
}

void get_adj_list()              // 隣接節点リストを求める
{
  int s, g;
  node_list_t *pnl;

  for(s=0; s<goal; s++){      // 各節点 s と
    for(g=s+1; g<=goal; g++){          // s 以降の各節点 g に対して
      if(is_possible(s,g)){                    // s から g へ円の内部で直線移動可能ならば
                                               // g を s の隣接節点として登録する
        pnl = (node_list_t *)malloc(sizeof(node_list_t));
        pnl->node_id = g;
        pnl->dist = DISTANCE(node[s].pos, node[g].pos);
        pnl->next = node[s].adj;
        node[s].adj = pnl;
      }
    }
  }
}

void solve()                     // 問題を解く
{
  int i, nid;
  point_t p1, p2;
  double r1, r2;
  nh = 0;
  nid = 0;
  for(i=0; i<n; i++){
    scanf("%lf %lf %lf", &p2.x, &p2.y, &r2);
    if(i==0){                              // 最初の円の中心がスタート地点
      node[nid].pos = p2;
      node[nid].cost = 0;
      node[nid].heap_pos = -1;
      node[nid++].adj = NULL;
      ins_heap(0);
    }
    else {           // 中心 p1 半径 r1 の円と 中心 p2 半径 r2 の円の交点を求める
      point_t lp, rp;
      double alpha, beta,d;
      d = DISTANCE(p1, p2);        // ２つの円の中心の距離
      alpha = acos((p2.x - p1.x)/d);  // ベクトル(p1->p2)と x 軸のなす角度を求める
      if(p2.y - p1.y < 0) alpha = -alpha;
```

```
        // ベクトル(p1->p2)とベクトル(p1->交点)のなす角度を求める
        beta = acos((d*d + r1*r1 - r2*r2)/(2*d*r1));
        // 進行方向右側の交点 rp を求める
        rp.x = p1.x + r1*cos(alpha-beta);
        rp.y = p1.y + r1*sin(alpha-beta);
        // 進行方向左側の交点 lp を求める
        lp.x = p1.x + r1*cos(alpha+beta);
        lp.y = p1.y + r1*sin(alpha+beta);
        // rp, lp の順に節点登録
        node[nid].pos = rp;
        node[nid].cost = INF;
        node[nid].heap_pos = -1;
        node[nid++].adj = NULL;
        node[nid].pos = lp;
        node[nid].cost = INF;
        node[nid].heap_pos = -1;
        node[nid++].adj = NULL;
      }
      p1 = p2;
      r1 = r2;
    }
    node[nid].pos = p1;                    // 最後の円の中心はゴール
    node[nid].cost = INF;
    node[nid].heap_pos = -1;
    node[nid].adj = NULL;
    start = 0; goal = nid;
    get_adj_list();                        // 隣接節点リスト、初期コストの設定
    while(1){                              // ダイクストラ法により最短経路長を求める
      int dn;
      node_list_t *pt;
      dn = del_heap();                     // ヒープより最小コストの節点を取ってくる
      if(dn == goal){                      // それがゴールなら最短経路長が求まった
        printf("%lf¥n", node[dn].cost);
        return;
      }
      pt = node[dn].adj;                   // pt->隣接節点
      while(pt){                 // 節点 dn 経由の方がコストが低ければ隣接節点のコストを更新
        if(node[pt->node_id].cost > node[dn].cost + pt->dist){
          node[pt->node_id].cost = node[dn].cost + pt->dist;
          ins_heap(pt->node_id);
        }
        pt = pt->next;
      }
    }
}

void free_all()      // 問題を解くために使用したグラフの隣接節点リスト領域を解放
{
    int i;
    node_list_t *pnl1, *pnl2;
    for(i=0; i<= goal; i++){    // 各節点に対して
      pnl1 = node[i].adj;                  // pnl1 -> 隣接節点リスト
      while(pnl1){                         // リストが空になるまでリストの要素を順に開放
```

```
            pnl2 = pnl1->next;
            free(pnl1);
            pnl1 = pnl2;
        }
    }
}

int main()
{
    int i;
    while(1){
        scanf("%d", &n);          // 円の個数を読み込む
        if(n==0) break;           // 個数が 0 なら終了
        solve();                  // 問題を解く
        free_all();               // 問題を解くために使用した領域を開放
    }
}
```