January 2006 Problem 'roping' Analysis

by Bruce Merry

The problem consists of two parts: determining which ropes do not intersect the circles, and choosing the maximum number of these ropes.

We first consider how to test whether a line (as opposed to a line segment) intersects a circle. Consider a triangle with vertices at the centre of the circle and at the posts defining the line. Its area can be computed in two ways: as the length of the line segment times the distance from the centre of the tree to the line, over 2, or using the cross product. The latter can be computed directly, so equating it to the former gives the perpendicular distance. If this is less than or equal to R, then the line intersects the circle. To avoid dealing with floating point errors, it is a good idea to square out and cross-multiply to work entirely in integers; one catch is that this could push the values over 64 bits, so it is necessary to create a big integer type.

It is also possible that the line meets the circle but the line segment does not. Clearly if either end-point is inside the circle that the line cuts the circle. Otherwise, orient the diagram so that the centre of the circle is at the origin and point A is on the positive X axis. If point B lies to the right of point A, then the line clearly misses the circle. If point B lies to the left of point A, and A also lies to the left of point B when point B is placed on the X axis, then the line segment will hit the circle if the line does.

This procedure allows us to identify valid ropes in $O(N^2T)$ time, but we still need to pick a subset that is free of intersections. We use dynamic programming, solving for the largest set between circles whose indices range from A to B (for every pair A < B). Suppose that there is at least one rope emanating from A, and consider the one that ends closest to B (i.e. the endpoint C has the highest index). The interior of triangle ABC must be completely empty, so the set of ropes is just the union of those ropes in the smaller cases (A, C) and (C, B), as well as the rope AB if it is legal. If A has no ropes emanating from it then we can set C = A + 1 and the above statements remain true. By iterating over all possible values of C, we can compute the optimal number of ropes for (A, B) in terms of the optimal numbers for smaller cases. The entire DP takes $O(N^3)$ time, making the overall algorithm $O(N^2T + N^3)$.

```
#define __STDC_LIMIT_MACROS

#include <iostream>
#include <fstream>
#include <complex>
#include <cassert>
#include <climits>

using namespace std;

#define MAXN 150
#define MAXT 100
#define MAXR 100000
#define MAXXY 1000000
typedef complex<long long> point;
```

```
static int N, T, R;
static point posts[MAXN];
static point trees[MAXT];
static bool valid[MAXN][MAXN];
static int dp[MAXN][MAXN];

static void readin() {
    long long x, y;
    ifstream in("roping.in");
    in >> N >> T >> R;
    assert(2 <= N && N <= MAXN);
    assert(0 <= T && T <= MAXT);
    assert(1 <= R && R <= MAXR);
    for (int i = 0; i < N; i++) {
        in >> x >> y;
        posts[i] = point(x, y);
    }
    for (int i = 0; i < T; i++) {
        in >> x >> y;
        trees[i] = point(x, y);
    }
}

static inline long long dot(const point &a, const point &b) {
    return real(conj(a) * b);
}

static inline long long cross(const point &a, const point &b) {
    return imag(conj(a) * b);
}

static inline long long area(const point &a, const point &b, const point &c)
{
    return cross(b - a, c - a);
}

static void validate() {
    for (int i = 0; i < N; i++) {
        assert(0 <= real(posts[i]) && real(posts[i]) <= MAXXY);
        assert(0 <= imag(posts[i]) && imag(posts[i]) <= MAXXY);
    }

    long long l = LONG_LONG_MAX, h = LONG_LONG_MIN;
    if (N >= 3) {
        for (int i = 1; i < N; i++)
        {
            int j = (i + 1) % N;
            int k = (i + 2) % N;
            assert(posts[i] != posts[j]);
            long long s = area(posts[i], posts[j], posts[k]);
            if (s == 0)
                assert(norm(posts[j] - posts[i]) < norm(posts[k] - posts[i])
                   && norm(posts[k] - posts[j]) < norm(posts[k] - posts[i]));
            l <?= s;
            h >?= s;
        }
        assert(l >= 0 || h <= 0);
    }

    for (int i = 0; i < T; i++) {
        long long l2 = LONG_LONG_MIN;
        long long h2 = 0;

        assert(0 <= real(trees[i]) && real(trees[i]) <= MAXXY);
        assert(0 <= imag(trees[i]) && imag(trees[i]) <= MAXXY);
        for (int j = 0; j < N; j++) {
            int k = (j + 1) % N;
            long long s = area(posts[j], posts[k], trees[i]);
            if (s == 0)
                assert(norm(trees[i] - posts[j]) < norm(posts[k] - posts[j])
                   && norm(trees[i] - posts[k]) < norm(posts[k] - posts[j]));
            l2 <?= s;
            h2 >?= s;
            // assert(norm(trees[i] - posts[j]) >= (long long) R * R);
        }
        assert(l < 0 || l2 >= 0);
        assert(h > 0 || h2 <= 0);
    }
}

struct huge {
    /* Base 2^64 */
    unsigned long long high;
    unsigned long long low;
};

static inline void huge_accum(huge &h, unsigned long long m) {
    unsigned long long m1 = m >> 32;
    unsigned long long m0 = (m & UINT_MAX) << 32;
    h.high += m1;
    if (h.low >= UINT64_MAX - m0) h.high++;
    h.low += m0;
}

static huge huge_mult(unsigned long long x, unsigned long long y) {
    unsigned long long x1 = x >> 32;
    unsigned long long x0 = x & UINT_MAX;
    unsigned long long y1 = y >> 32;
    unsigned long long y0 = y & UINT_MAX;
    huge ans = {x1 * y1, x0 * y0};
    huge_accum(ans, x1 * y0);
    huge_accum(ans, x0 * y1);
    return ans;
}

static bool intersects(point A, point B, point O, long long R) {
    A -= O;
    B -= O;
    if (norm(A) <= R * R) return true;
    if (norm(B) <= R * R) return true;
    /* We want to check if cross(A, B)^2 > norm(A - B) * R^2, which would
     * tell us if the line itself misses the circle. However, both sides
     * may overflow.
     */
    unsigned long long c = ::llabs(cross(A, B));
    unsigned long long n = norm(A - B);
    unsigned long long r = R * R;
    huge left = huge_mult(c, c);
    huge right = huge_mult(n, r);
    if (left.high > right.high
        || (left.high == right.high && left.low > right.low)) return false;
    if (cross(A, B) > sqrtl(n) * R) return false;
    if (dot(A, B) > norm(A)) return false;
    if (dot(A, B) > norm(B)) return false;
    return true;
}

static void make_valid() {
    memset(valid, 0, sizeof(valid));
    for (int i = 0; i < N; i++)
        for (int j = i + 2; j < N; j++)
        {
            valid[i][j] = true;
            for (int k = 0; k < T; k++)
                if (intersects(posts[i], posts[j], trees[k], R))
                {
                    valid[i][j] = false;
                    break;
                }
            valid[j][i] = valid[i][j];
        }
    valid[0][N - 1] = valid[N - 1][0] = false;
}

static void solve() {
    make_valid();
    memset(dp, 0, sizeof(dp));
    for (int d = 2; d < N; d++)
        for (int i = 0; i < N - d; i++)
        {
            int j = i + d;
            for (int k = i + 1; k < j; k++)
                dp[i][j] >?= dp[i][k] + dp[k][j];
            dp[i][j] += valid[i][j] ? 1 : 0;
        }
}

static void writeout() {
    ofstream out("roping.out");
    out << dp[0][N - 1] << "\n";
}

int main()
{
    readin();
    validate();
    solve();
    writeout();
    return 0;
}
```