Solution — Germs

1 The Problem in a Nutshell

We are given a set of circular germs in a rectangular dish, all of which grow concentrically at the same quadratic rate. A germ dies (but keeps growing) as soon as it touches either another germ or the boundary of the dish. How long does it take until the first germ dies, until strictly more than half of the germs die, and until all germs die?

2 Modeling

In the following, we identify a germ with its center point in the Euclidean plane. Define G to be the set of size n that contains all germs. As all germs grow at the exact same rate, the time until a particular germ dies is determined either by the nearest other germ or by the nearest boundary of the dish.

Let $d(g, g')$ denote the distance between the center points of two germs g and g' and let $t_{g,g'}$ be the time (in hours) until g and g' touch for the first time. Using some simple algebra, we see that

$$d(g, g') \stackrel{!}{=} 2\rho(t_{g,g'}) = 1 + 2t_{g,g'}^2 \quad \Longrightarrow \quad t_{g,g'} = \sqrt{\frac{d(g, g') - 1}{2}}.$$

Similarly, if $d(g, b)$ denotes the distance between the center point of a germ g and a boundary b, and if $t_{g,b}$ is the time (in hours) until g touches b, then

$$d(g, b) \stackrel{!}{=} \rho(t_{g,b}) = 1/2 + t_{g,b}^2 \quad \Longrightarrow \quad t_{g,b} = \sqrt{d(g, b) - 1/2} = \sqrt{\frac{2d(g, b) - 1}{2}}.$$

Consequently, in order to determine the lifetime of a particular germ g, it suffices to find the overall minimum of all values that the following two expressions can attain (where g' ranges over all other germs and b ranges over the four boundaries).

$$d(g, g') \qquad\qquad 2d(g, b)$$

Equivalently from a mathematical viewpoint, but much more conveniently from a practical viewpoint, we can also minimize over the squares of these two expressions, i.e.,

$$S(g, g') := d(g, g')^2 \qquad\qquad S(g, b) := (2d(g, b))^2 = 4d(g, b)^2.$$

Once the time until death has been decided for each germ, the remainder of the problem amounts to a simple determination of the minimum, the maximum, and the median in a totally ordered set. We will ignore this simple part of the problem in the next section, but we will come back to it in the final section.

3 Algorithm Design

From the problem description we learn that the number n of germs is at most 10^5. This means that in an extreme case, the number of pairs (g_i, g_j) of germs could be in the vicinity of 10^{10}. Hence, any algorithm that looks at all pairs of germs is probably not going to be fast enough. Still, for the sake of including partial solutions, let us quickly explore this possibility.

$O(n^2)$ solution (40 points) In essence, there are two things we have to do: Find (i) the minimum $S(g, b)$ for every germ g and find (ii) the minimum $S(g, g')$ for every germ g.

Task (i) can be solved in time $O(n)$. Simply go over all germs g and compute the minimum $S(g, b)$ in constant time, which is possible since there is only a constant number of boundaries b and the distance between a point and a line can be computed in constant time.

For task (ii) we iterate over all pairs (g_i, g_j) of germs in time $O(n^2)$. For each pair we compute $S(g_i, g_j)$ and then check for both g_i and g_j whether the computed value is smaller than any previously observed value. Individual such steps can be implemented in constant time provided that we store the current minimum $S(g, g')$ for every germ g in some data structure, such as an array.

$O(n \log n)$ solution (100 points) It is clear that task (i), as defined above, cannot be solved faster than in time $O(n)$. Task (ii) however can be done much faster if we make use of the following fact which was discussed in the tutorial: The Delaunay triangulation of the point set G contains for each germ g an edge to a germ g' which is closest to g, and it contains at most $O(n)$ edges in total. Therefore, instead of iterating over all pairs (g_i, g_j) as before, we first compute the Delaunay triangulation of G in time $O(n \log n)$, and then we only iterate over the $O(n)$ pairs (g_i, g_j) which correspond to an edge in this triangulation. By the above fact, for each germ g we are still guaranteed to iterate over a pair (g, g') which witnesses the smallest possible value for $S(g, g')$.

4 Implementation

Most of the computations that we do will operate on S-expressions (as defined in Section 2) so that we do not have to deal with square roots too often. Still, at the very end when computing the output, we will not get around computing at least some square roots. Hence, we use the following CGAL kernels.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel IK;
typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt EK;
```

We further assume that the input of one particular testcase has already been read into the following variables and data structures.

```
std::size_t n;
IK::FT l, b, r, t;
std::vector<IK::Point_2> germs;
```

The outputs (unrounded, in hours) for this testcase will be stored in the following variables.

```
EK::FT first, median, last;
```

$O(n^2)$ **solution (40 points)** Remember that for each germ g we want to compute and minimize over all possible S-values $S(g, g')$ and $S(g, b)$, as defined in Section 2. While doing that we store the current minimum for each germ in the following data structure, which uses the same indices as `germs`.

```
std::vector<IK::FT> min_S(n);
```

Let us start by computing for each germ g the minimum $S(g, b)$, where b can be any one of the four boundaries.

```
for (std::size_t i = 0; i < n; ++i)
{
  IK::Point_2 const & g = germs[i];
  IK::FT min_horizontal = std::min(g.x() - l, r - g.x());
  IK::FT min_vertical   = std::min(g.y() - b, t - g.y());
  IK::FT min_overall    = std::min(min_horizontal, min_vertical);
  min_S[i] = 4 * min_overall * min_overall;
}
```

In order to minimize over the expressions of the form $S(g, g')$, we iterate over all pairs (g_i, g_j) of germs as follows.

```
for (std::size_t i = 0; i < n; ++i)
{
  for (std::size_t j = i+1; j < n; ++j)
  {
    IK::Point_2 const & gi = germs[i];
    IK::Point_2 const & gj = germs[j];
    IK::FT squared_distance = CGAL::squared_distance(gi, gj);
    min_S[i] = std::min(min_S[i], squared_distance);
    min_S[j] = std::min(min_S[j], squared_distance);
  }
}
```

Given that we now know the minimum S-expression for every germ, we can simply sort all these values and then pick out the desired ones (note that this particular step could be implemented faster, in expected time $O(n)$, for example by using the function `std::nth_element`).

```
std::sort(min_S.begin(), min_S.end());
first  = S_to_hrs(EK::FT(min_S[0]));
median = S_to_hrs(EK::FT(min_S[n/2]));
last   = S_to_hrs(EK::FT(min_S[n-1]));
```

The function `S_to_hrs` does the algebraic conversion from an S-expression to the corresponding time until death in hours (see Section 2 for the mathematical formulae). Note that this is the only place where we compute square roots.

```
EK::FT S_to_hrs(EK::FT const & S)
{
  return CGAL::sqrt((CGAL::sqrt(S) - 1) / 2);
}
```

$O(n \log n)$ **solution (100 points)** Instead of reusing the vector `min_S` we store the current minimum S-expressions in the `info`-field of the corresponding vertex of the Delaunay triangulation. To be able to do that later, we now write the following.

```
typedef CGAL::Triangulation_vertex_base_with_info_2<IK::FT,IK> vertex_t;
typedef CGAL::Triangulation_face_base_2<IK> face_t;
typedef CGAL::Triangulation_data_structure_2<vertex_t,face_t> triangulation_t;
typedef CGAL::Delaunay_triangulation_2<IK,triangulation_t> delaunay_t;

delaunay_t trg;
trg.insert(germs.begin(), germs.end());
```

Apart from the way we do iteration, the computation of the expressions of the form $S(g, b)$ stays the same.

The big difference is in the way we compute the expressions $S(g, g')$. While before we had to iterate over all pairs (g_i, g_j) of germs, we now only iterate over those pairs which correspond to an edge in the Delaunay triangulation.

```
for (auto e = trg.finite_edges_begin();
     e != trg.finite_edges_end(); ++e)
{
  auto vi = e->first->vertex(trg.cw(e->second));
  IK::Point_2 const & gi = vi->point();
  IK::FT & info_gi = vi->info();

  auto vj = e->first->vertex(trg.ccw(e->second));
  IK::Point_2 const & gj = vj->point();
  IK::FT & info_gj = vj->info();

  IK::FT squared_distance = CGAL::squared_distance(gi, gj);
  info_gi = std::min(info_gi, squared_distance);
  info_gj = std::min(info_gj, squared_distance);
}
```

Once the minimum S-expressions have been extracted from the `info`-fields of the triangulation, the remainder of the code can be reused.

5 A complete solution

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <CGAL/Triangulation_data_structure_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <algorithm>
#include <iostream>
#include <limits>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel IK;
typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt EK;
typedef CGAL::Triangulation_vertex_base_with_info_2<IK::FT,IK> vertex_t;
typedef CGAL::Triangulation_face_base_2<IK> face_t;
typedef CGAL::Triangulation_data_structure_2<vertex_t,face_t> triangulation_t;
typedef CGAL::Delaunay_triangulation_2<IK,triangulation_t> delaunay_t;

double ceil_to_double(EK::FT const & x)
{
  double a = std::ceil(CGAL::to_double(x));
```

```
  while (a < x)    a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

EK::FT S_to_hrs(EK::FT const & S)
{
  return CGAL::sqrt((CGAL::sqrt(S) - 1) / 2);
}

void test_case(int n)
{
  IK::FT l, b, r, t;
  std::cin >> l >> b >> r >> t;

  std::vector<IK::Point_2> germs(n);
  for (std::size_t i = 0; i < n; ++i)
  {
    std::cin >> germs[i];
  }

  delaunay_t trg;
  trg.insert(germs.begin(), germs.end());

  for (auto v = trg.finite_vertices_begin(); v != trg.finite_vertices_end(); ++v)
  {
    IK::Point_2 const & g = v->point();
    IK::FT & info_g = v->info();
    IK::FT min_horizontal = std::min(g.x() - l, r - g.x());
    IK::FT min_vertical   = std::min(g.y() - b, t - g.y());
    IK::FT min_overall    = std::min(min_horizontal, min_vertical);
    info_g = 4 * min_overall * min_overall;
  }

  for (auto e = trg.finite_edges_begin(); e != trg.finite_edges_end(); ++e)
  {
    auto vi = e->first->vertex(trg.cw(e->second));
    IK::Point_2 const & gi = vi->point();
    IK::FT & info_gi = vi->info();

    auto vj = e->first->vertex(trg.ccw(e->second));
    IK::Point_2 const & gj = vj->point();
    IK::FT & info_gj = vj->info();

    IK::FT squared_distance = CGAL::squared_distance(gi, gj);
    info_gi = std::min(info_gi, squared_distance);
    info_gj = std::min(info_gj, squared_distance);
  }

  std::vector<IK::FT> min_S;
  min_S.reserve(n);
  for (auto v = trg.finite_vertices_begin(); v != trg.finite_vertices_end(); ++v)
  {
    min_S.push_back(v->info());
  }

  std::sort(min_S.begin(), min_S.end());
  EK::FT first  = S_to_hrs(EK::FT(min_S[0]));
  EK::FT median = S_to_hrs(EK::FT(min_S[n/2]));
```

```
  EK::FT last   = S_to_hrs(EK::FT(min_S[n-1]));

  std::cout
    << ceil_to_double(first) << " "
    << ceil_to_double(median) << " "
    << ceil_to_double(last) << "\n";
}

int main()
{
  std::ios_base::sync_with_stdio(false);
  int n;
  while (std::cin >> n && n != 0) test_case(n);
}
```