# Solution — Graypes

## 1 The Problem in a Nutshell

Given a set of graypes in the plane, all of which are running at unit speed to the nearest fellow graype (in case of a tie the graype with the lexicographically smallest position is preferred), how long does it take for the first pair of graypes to reach each other?

## 2 Modeling

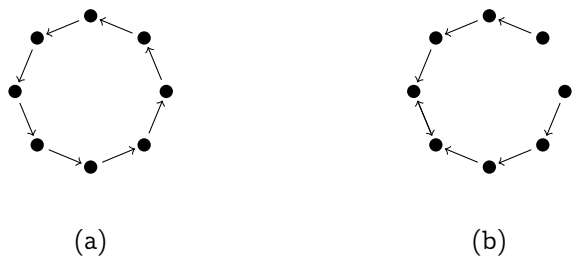
First and foremost, it is crucial to understand the importance of the way that ties are broken. If it were not for that assumption, the problem would be much more difficult because of the existence of directed cycles of graypes running towards one another, as exemplified in Figure 1(a). Because of the additional assumption, however, the situation will look as depicted in Figure 1(b).

In particular, it is not hard to see that there always will be a pair $(g_1, g_2)$ of graypes which are at minimum distance and such that $g_1$ runs towards $g_2$ while $g_2$ runs towards $g_1$ (see the bidirectional edge in the figure). Indeed, out of all graypes which have at least one fellow graype at overall minumum distance, simply pick $g_1$ as the graype with lexicographically smallest position and pick $g_2$ as the graype that $g_1$ is running towards. This also guarantees that $g_1$ and $g_2$ run towards each other on a straight line segment.

Consequently, the posed problem can be solved as follows. We model the set of graypes as a set $G$ of $n$ points in the Euclidean plane. Then we find a (not necessarily unique) pair of points in $G$ at minimum distance. Finally, we compute the time needed for one graype to run half that distance.

## 3 Algorithm Design

From the problem description we learn that the number $n$ of points (a.k.a. graypes) is at most $100,000$. This means that the number of pairs of points, which is of order $\Theta(n^2)$, is already in



(a) (b)

Figure 1: (a) Confused graypes running in circles. (b) Clever graypes with a sense of order.

the billions. While this implies that a $O(n^2)$ algorithm might not be fast enough, it might still be a feasible solution for the easier test cases.

**$O(n^2)$ solution (20 points)** The simplest way to solve the problem is to iterate over all pairs $(g_1, g_2)$ of graypes, to compute the corresponding distances between $g_1$ and $g_2$ and then to pick the overall minimum distance. One way to make this approach a bit faster is to observe that the pair $(g_1, g_2)$ with minimum distance is the same as the pair $(g_1, g_2)$ with minimum *squared* distance. Dealing with squared distances is much easier since they do not require the computation of square roots. Therefore, it makes more sense to find the pair $(g_1, g_2)$ with minimum squared distance, and then to compute the actual distance (using the computation of one single square root) at the very end.

**$O(n \log n)$ solution (100 points)** A more efficient approach is to first compute the Delaunay triangulation of the point set G corresponding to the graypes, and to compute a solution based on the length of the shortest edge in this triangulation. This is a sound strategy as long as (one of) the overall shortest edges is contained in the Delaunay triangulation. Luckily for us, this is precisely one of the many convenient properties of the Delaunay triangulation. Moreover, since CGAL is able to compute the Delaunay triangulation of $n$ points in expected time $O(n \log n)$ and the number of edges in any triangulation is bounded by $O(n)$, the overall running time of this second approach will be only $O(n \log n)$.

### 4 Implementation

For the final output we need to compute actual distances, that is, square roots. But everywhere else we can work with squared distances and avoid square roots. Moreover, all input numbers are specified to be integers of absolute value at most $2^{25}$. Therefore, the squared distance between two input points $p = (x_1, y_1)$ and $q = (x_2, y_2)$ is bounded by

$$\|p - q\|^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 \leq (2^{26})^2 + (2^{26})^2 = 2^{53},$$

which is exactly the mantissa width of `double`. These observations suggest the use of the following two CGAL kernels.

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

Let us assume that the input of one particular testcase has already been read into the following data stucture.

```
1 std::vector<IK::Point_2> graypes;
```

The output (unrounded, in hundredths of a second) for this testcase will be stored in the following variable.

```
1 EK::FT time_cs;
```

$O(n^2)$ **solution (20 points)** Iteration over all pairs of elements in a `std::vector` can be implemented as shown below. As mentioned earlier, we use the function `CGAL::squared_distance` in order to avoid unnecessary square root computations and using `IK::FT` (a.k.a. `double`) is fine here. Finally, take note of the cast to `EK::FT` right before calling the function `CGAL::sqrt`, which ensures that also the square root is computed without any loss of precision.

```
IK::FT min_squared_distance = std::numeric_limits<IK::FT>::max();
for (auto g1 = graypes.begin(); g1 != graypes.end(); ++g1)
{
  for (auto g2 = std::next(g1); g2 != graypes.end(); ++g2)
  {
    min_squared_distance = std::min(
      min_squared_distance,
      CGAL::squared_distance(*g1, *g2));
  }
}
time_cs = 50 * CGAL::sqrt(EK::FT(min_squared_distance));
```

$O(n \log n)$ **solution (100 points)** Computation of the Delaunay triangulation and iteration over all its edges can be implemented as shown below. Since computing the Delaunay triangulation does not require any non-trivial constructions, it is still safe to use the inexact kernel `IK` here.

```
typedef CGAL::Delaunay_triangulation_2<IK> delaunay_t;
delaunay_t trg;
trg.insert(graypes.begin(), graypes.end());

IK::FT min_squared_distance = std::numeric_limits<IK::FT>::max();
for (auto e = trg.finite_edges_begin(); e != trg.finite_edges_end(); ++e)
{
  min_squared_distance = std::min(
    min_squared_distance,
    trg.segment(*e).squared_length());
}
time_cs = 50 * CGAL::sqrt(EK::FT(min_squared_distance));
```

## 5 A Complete Solution

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <vector>
#include <iostream>

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

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;
}

void test_case(int n)
```

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

  typedef CGAL::Delaunay_triangulation_2<IK> delaunay_t;
  delaunay_t trg;
  trg.insert(graypes.begin(), graypes.end());

  IK::FT min_squared_distance = std::numeric_limits<IK::FT>::max();
  for (auto e = trg.finite_edges_begin(); e != trg.finite_edges_end(); ++e)
  {
    min_squared_distance = std::min(
      min_squared_distance,
      trg.segment(*e).squared_length());
  }
  EK::FT time_cs = 50 * CGAL::sqrt(EK::FT(min_squared_distance));

  std::cout << ceil_to_double(time_cs) << "\n";
}

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