Solution — The Empire Strikes Back

1 The Problem in a Nutshell

We are given a set A of asteroid particles, a set B of bounty hunters, a set S of shooting points and the total available energy e. Each $a \in A$ is represented by a triplet $a = (x, y, d)$, where (x, y) is its position and d its density. Each $b \in B$ and $s \in S$ is just a point in space.

The question is to decide if it is possible to destroy all asteroid particles, using at most e energy in total. For this, the quadlasers of the Falcon can shoot an amount of energy at each shooting point. This creates a circular explosion around the shooting point, the radius of which is as big as possible such that it does not include any bounty hunters. The energy an asteroid particle receives depends on the distance to the shooting point; to be destroyed it needs to receive an amount of energy that is at least as large as its density. Consider the example in the following figure.

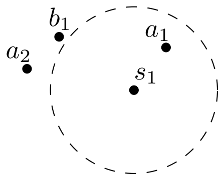

Figure 1: An example: The explosion at shooting point s_1 can reach the asteroid particle a_1 and, thus, contribute to the energy that a_1 receives. On the other hand, it cannot reach a_2 because if it did then it would be discovered by bounty hunter b_1.

2 Modeling

This problem has a geometric flavor and CGAL seems like the first idea that comes to mind. In addition, it has an optimization flavor; the latter comes from trying to minimize the total energy used (equivalently keep the total energy used below the given upper bound e). Moreover, for an asteroid particle to be destroyed the energy it receives has to be as large as its density. The energy an asteroid particle receives is described on the problem sheet by a simple summation; this gives rise to a linear constraint. There is, already, enough hints that this problem should be solved by a Linear Program (LP for short).

But what exactly are the *variables* in this LP? First of all, we have to compute the radius for each shooting point (or explosion). Is this part of the LP or is it a pre-computation step? It is not so hard to see that this is a pre-computation step. The example in Figure 1 already illustrates how we can compute the radius for each shooting point: It is as large as possible so that no bounty hunter is reached. That means that only the *nearest* bounty hunter, to the specific shooting point, is interesting for the computation of the radius.

After we have computed the maximum allowed radius r_s, for each shooting point $s \in S$, we know which shooting points contribute energy to which particles. Our "goal" is to minimize the total energy used and so we have one variable e_s per shooting point s that represents how much energy is used at the specific shooting point. Moreover, there must be one constraint per asteroid particle, which ensures that it receives enough energy to be destroyed.

Complexity of the CGAL LP/QP Solver. The LP we are trying to formulate has one variable per shooting point. From the problem statement we know that this number is at most 10^4. What is exactly the complexity of the Solver and how come 10^4 is not too much? There is a simple explanation: the number of constraints is small, since we have one constraint per asteroid particle (at most 30). That is enough such that the Solver solves this LP efficiently.

Let n be the number of variables and m be the number of constraints of a LP. The time that the CGAL LP/QP Solver would take is proportional to $\min(n, m)$. This is because for each LP there is a so-called *dual* LP which has one variable per constraint. The Solver knows this and can take advantage automatically. Therefore, one of n and m has to be small enough for the Solver to be able to treat it efficiently. To sum up, the Solver, in most cases, will take time that is linear in $\max(n, m)$ if you assume that $\min(n, m)$ is constant. For more details you can look up the CGAL LP/QP Solver documentation

For QP also the rank of the matrix D has to be small but, for Algolab purposes, we will guarantee that this is the case (like we guarantee that the matrix will be positive semidefinite). Other than that, the situation for the solver is the same for QP as it is for LP.

3 Algorithm Design

Firstly, observe that for the first group of test sets there are no bounty hunters. It is enough to formulate the LP (see the next section) to get the first 20 points.

For all other groups of test sets there is a positive number of bounty hunters. As discussed above, we have to find the nearest bounty hunter for each shooting point to define its radius. Of course, this can be computed by checking the distances to all bounty hunters, for each shooting point. This computation will take $O(|S| \cdot |B|)$ time. Writing up a solution like this would not give you full points, though.

For full points, another approach must be taken to find the nearest bounty hunter to each shooting point. In general, every time we want to perform a *nearest neighbor* query, we use Delaunay Triangulations (DT for short). Computing a DT for n points is done in $O(n \log n)$ time. Firstly, let us stress once again (refer to the Proximity Structures tutorial slides) that if we are given a vertex of the DT, the nearest vertex to it will be its neighbor in the DT graph.

In our case, we compute a DT of the bounty hunters in $O(|B| \log |B|)$ time. Then, for each shooting point s we can ask which is the nearest vertex and CGAL will give us an answer in expected logarithmic time, $\log |B|$. Here is a code snippet for this operation:

```
Delaunay::Vertex_handle va = d.nearest_vertex(s);
```

In the above, `s` is a shooting point and `d` is the variable that holds the DT. To turn this vertex handle to a point one can do the following:

```
va->point();
```

In this case, the above function will return an element of type `Point_2`. There is no need to use square roots here, so we will use squared distances instead. For the shooting point s, we find the squared distance to the closest bounty hunter this way:

```
sqdist = CGAL::squared_distance(s, va->point());
```

Then, we decide if shooting point s contributes to the energy an asteroid particle receives if their squared distance is strictly smaller than the value `sqdist` (at squared distance exactly `sqdist` is the nearest bounty hunter to s). We have, thus, figured out which explosions contribute to which asteroid particles and we are ready to formulate the LP.

4 Formulating the Linear Program

For the LP, we have a simple minimization objective; that is, to minimize the total used energy. If we denote with e_j the energy used at the explosion of shooting point $j \in S$, then we have to minimize $\sum_{j=1}^{|S|} e_j$.

The constraints of the LP are quite straightforward, we just need one constraint per asteroid particle. Here is our LP:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{|S|} e_j \\ \text{subject to} \quad & \sum_{s \in R(a)} \frac{e_s}{\max\{1, \|a - s\|^2\}} \geq d_a, \quad \forall a \in A \end{aligned}$$

where $R(a) \subseteq S$ is the set of shooting points that can reach a, that is, $\|a - s\| \leq r_s$. For each asteroid $a \in A$, d_a is its density. In words, the above constraint makes sure that the energy every asteroid particle a receives is at least as high as its density. You can see how we set up the LP constraints in the complete solution at the end of this document, in Lines 41-53.

We formulate the above LP and give it to the CGAL solver. After it solves the LP we compare the optimal value of the result with the total available energy e (which is in variable `upperbound` in the code below).

```
Solution sol = CGAL::solve_linear_program(lp, ET());
if (sol.is_infeasible()) cout << "n" << endl;
else {
  if (sol.objective_value() <= upperbound)
    cout << "y" << endl;
  else
    cout << "n" << endl;
}
```

Note that one could, of course, impose the upper bound on the total energy as a constraint of the LP. Then, the y/n output would only depend on if the LP is feasible or not.

5 Implementation

Here are a few remarks about the implementation. A complete solution in code follows in the next section.

- Firstly, let us discuss the number types for this LP. In this exercise, the coefficients of our LP will actually be rational numbers. Therefore, we choose to use Gmpq for the input and exact type:

  ```
  typedef CGAL::Gmpq                                        ET;
  typedef CGAL::Quadratic_program<ET>                       Program;
  typedef CGAL::Quadratic_program_solution<ET>              Solution;
  ```

  So we will use Gmpq to set up the LP and we will also tell the Solver to use Gmpq, internally, as the exact type (the type used for the calculations that the Solver performs). Also, note that for computing the fractions Gmpq should also be used. For example, see Line 47 of the code below.

- For computing the nearest neighbor we use a DT. It has been discussed extensively (e.g. see solution for Goldeneye) that the computations associated with DT only use predicates and thus the EPIC-kernel is good enough.

- We have one vector for each set of asteroids, shooting point, and bounty hunters. For each asteroid we have a pair of a Point_2 and an int. The latter represents the density of the asteroid and is given in the input. For each shooting point we have a pair of a Point_2 and a double. The latter represents the squared distance to the closest bounty hunter, which we compute in Lines 35-36 (of the complete solution below), for each shooting point. Finally, the bounty hunters are represented by points.

6 A Complete Solution

```cpp
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpq.h>
#include <vector>

typedef CGAL::Gmpq                                                  ET;
typedef CGAL::Quadratic_program<ET>                                 Program;
typedef CGAL::Quadratic_program_solution<ET>                        Solution;
typedef CGAL::Exact_predicates_inexact_constructions_kernel   K;
typedef CGAL::Delaunay_triangulation_2<K>                           Delaunay;
typedef K::Point_2                                                  P;
typedef std::vector<P>                                              PV;
typedef std::vector<std::pair<P, int> >                             AV;
typedef std::vector<std::pair<P, double> >                          SV;

void testcase(const AV& ast, SV& shoot, const PV& bh, int upperbound)
{
  int a = ast.size();
  int s = shoot.size();
  int b = bh.size();

  // compute DT
  Delaunay d;
  d.insert(bh.begin(), bh.end());

  Program lp(CGAL::LARGER, true, 0, false, 0);

  // compute max_sqdist for each shooting point
  // and set c for the LP
  for (int i = 0; i < s; ++i) {
    if (b > 0) {
      Delaunay::Vertex_handle va = d.nearest_vertex(shoot[i].first);
      shoot[i].second = CGAL::squared_distance(shoot[i].first, va->point());
    }
    lp.set_c(i, 1);
  }

  // setup the LP constraints - one for each asteroid
  for (int i = 0; i < a; ++i) {
    for (int j = 0; j < s; ++j) {
      K::FT sqdist = CGAL::squared_distance(ast[i].first, shoot[j].first);
      if (sqdist < shoot[j].second) {
        if (sqdist > 0)
          lp.set_a(j, i, 1 / ET(sqdist));
        else
          lp.set_a(j, i, 1);
      }
    }
    lp.set_b(i, ast[i].second);
  }

  Solution sol = CGAL::solve_linear_program(lp, ET());
  if (sol.is_infeasible() || sol.objective_value() > upperbound)
    std::cout << "n\n";
  else
```

```cpp
    std::cout << "y\n";
}

int main()
{
  std::cin.sync_with_stdio(false);
  int t;
  for (std::cin >> t; t > 0; --t) {
    int a, s, b, up;
    std::cin >> a >> s >> b >> up;

    AV ast;
    for (int i = 0; i < a; ++i) {
      P as;
      int intens;
      std::cin >> as >> intens;
      ast.push_back(std::make_pair(as, intens));
    };

    SV shoot;
    for (int i = 0; i < s; ++i) {
      P sh;
      std::cin >> sh;
      shoot.push_back(std::make_pair(sh, 1<<30));
    }

    PV bh;
    for (int i = 0; i < b; ++i) {
      int x, y;
      std::cin >> x >> y;
      bh.push_back(P(x,y));
    }

    testcase(ast, shoot, bh, up);
  }
  return 0;
}
```