Data Structures and Objects
CSIS 3700

Spring Semester 2019 — CRN 21212

Project 1 — Cops and Robbers
Due date: Friday, February 8, 2019

Goal

Given the locations of several police officers and several robbers, determine which of a group of people are safe and which will be robbed. Use the **`Fraction`** class you developed in lab and a new **`Point`** class to store the locations of all people in the problem.

Details

▹*The `Fraction` class*

For this project, you should use the **`Fraction`** class you created in lab. Make sure that all methods have been implemented *and tested*.

▹*The `Point` class*

Write a **`Point`** class that implements the following:

- Input and output of a point in the form (x, y) where x and y are two **`Fraction`** objects
- Add and subtract two points using **`operator+`** and **`operator-`**
- Multiply two points, computing the cross product
- Multiply a point by a **`Fraction`**, which multiplies both of the point's coordinates by the given fraction.

Note that both forms of multiply should use **`operator*`**; for cross product the parameter should be a **`Point`** and for scaling it should be a **`Fraction`**.

▹*Source code layout*

In this project you should use a conventional text editor and the **`g++`** and **`make`** tools. Since I want you to get some additional experience with these tools, you should not use an IDE for this project.

The project is also an exercise in *separate compilation*, where the source code is divided into multiple parts. The project consists of six files:

- A **`fraction.h`** header file that contains the class definition for the **`Fraction`** class;
- A **`fraction.cc`** file containing the **`Fraction`** class methods (functions);
- A **`point.h`** header file that contains the class definition for the **`Point`** class;
- A **`point.cc`** file containing the **`Point`** class methods (functions);
- A file containing a **`main`** function that performs the intersection calculations
- A **`Makefile`** that directs the process of creating the executable file

▹Input

The input begins with a single line consisting of three integers *c r o* indicating the number of police officers, the number of robbers and the number of people whose safety is to be determined. The numbers will be in the ranges $3 \le c \le 100$, $3 \le r \le 100$ and $o \ge 1$.

This first line will be followed by c lines, each containing one point for each officer. These are followed by r lines, each with one point for each robber. Finally, there are o lines, each with a point for the remaining people.

▹Processing

For each of the o people, apply these rules in order to determine their safety:

1. If the person is inside or on the edge of a triangle formed by any three officers, then that person is *safe*.
2. Otherwise, if the person is inside or on the edge of a triangle formed by three robbers, then that person is *robbed*.
3. Otherwise, the person is not inside or on the edge of any such triangle and that person is *in danger*.

Note that each person is in exactly one category. If, for example, a person is inside both a triangle of officers and a triangle of robbers, the person is safe.

▹Output

Suppose that there are s safe people, t people who are robbed and d people who are in danger. The output should look like this:

```
Safe:  s
Robbed:  t
In danger:  d
```

▹Arithmetic on points

Suppose you have two points, $\mathbf{a} = (\mathbf{a}_x, \mathbf{a}_y)$ and $\mathbf{b} = (\mathbf{b}_x, \mathbf{b}_y)$. Their sum and difference are defined by taking the sum or difference of their coordinates individually:

$$\mathbf{a} + \mathbf{b} = (\mathbf{a}_x + \mathbf{b}_x, \mathbf{a}_y + \mathbf{b}_y)$$
$$\mathbf{a} - \mathbf{b} = (\mathbf{a}_x - \mathbf{b}_x, \mathbf{a}_y - \mathbf{b}_y)$$

If $r \in \mathbb{R}$ is a real number, then the product of r and $\mathbf{a}$ just multiplies each coordinate by r:

$$r\mathbf{a} = (r\mathbf{a}_x, r\mathbf{a}_y)$$

Note that each of the preceding three operations yields a point as an answer.

It is also possible to "multiply" two points; this is called a *cross product*. The cross product of $\mathbf{a}$ and $\mathbf{b}$ is defined as follows:

$$\mathbf{a} \times \mathbf{b} = \mathbf{a}_x\mathbf{b}_y - \mathbf{a}_y\mathbf{b}_x$$

Note that the cross product yields a single real number as an answer.

▸*Determining safety*

To determine if a person is safe (or robbed), they must be in or on some triangle of three officers (or robbers) from a set of officers. One way to do this is to check all possible triangles. However, there are $\binom{c}{3} = \frac{c^3-3c^2+2c}{6}$ triangles to check for each person. We need a better approach.

The technique we need involves something known as the *convex hull* of the set of officers (or robbers). The convex hull of a set of points is the smallest convex polygon that contains all of the points. Some of the points will be vertices of the hull, some will be on an edge and the remaining points are all inside the hull. It can be shown that a point is inside one of the triangles formed by three points in the set if and only if the given point is inside the convex hull.

The project then reduces to two subproblems:

1. Find the convex hull of officers (and the hull of robbers)
2. For each person, determine if they are inside (or on) the hull.

Both of these are well-known problems in *computational geometry*, a branch of computing dealing with various geometry problems in two, three or more dimensions. As is the case with many classic computational geometry problems, neither of these subproblems requires overly complicated computation or data structures.

The algorithms for each subproblem are given below.

Algorithm 1 Finding the convex hull of a set of points $p_0, p_1, \cdots, p_{k-1}$

```
1:  procedure FindHull(Point P[ ], int k)
2:      Sort P by x-coordinate                ▸ Break ties in favor of smaller y-coordinate

3:      lower[0] ← p_0
4:      t ← 0
5:      for i ← 1 to k − 1 do
6:          while t > 0 and lower[t − 1] → lower[t] → p_i does not turn left do
7:              t ← t − 1
8:          end while

9:          t ← t + 1
10:         lower[t] ← p_i
11:     end for

12:     upper[0] ← p_{k−1}
13:     u ← 0
14:     for i ← k − 2 downto 0 do
15:         while u > 0 and upper[u − 1] → upper[u] → p_i does not turn left do
16:             u ← u − 1
17:         end while

18:         u ← u + 1
19:         upper[u] ← p_i
20:     end for

21:     return lower[0], lower[1], ··· , lower[t − 1], upper[0], upper[1], ··· , upper[u − 1]
22: end procedure
```

Algorithm 2 Determining if point P is inside or on a convex hull $h_0, h_1, \cdots, h_{k-1}$

```
1: procedure IsInsideHull(Point P, Point H[ ], int k)
2:     for i ← 0 to k − 1 do
3:         j ← (i + 1) mod k
4:         if P → h_i → h_j turns right then
5:             return false
6:         end if
7:     end for

8:     return true
9: end procedure
```

Note that this algorithm determines if the point is inside or on the convex hull. If you wanted to determine if the point is stictly inside the hull, replace "turns right" with "does not turn left."

There is a simple, clever method for determining a left-hand turn that relies on the cross-product. To see if $a \to b \to c$ turns left, calculate $d = (b - a) \times (c - a)$, where $\times$ denotes cross product. Then, $a \to b \to c$ turns left if $d > 0$, turns right if $d < 0$ and the three points are collinear if $d = 0$.

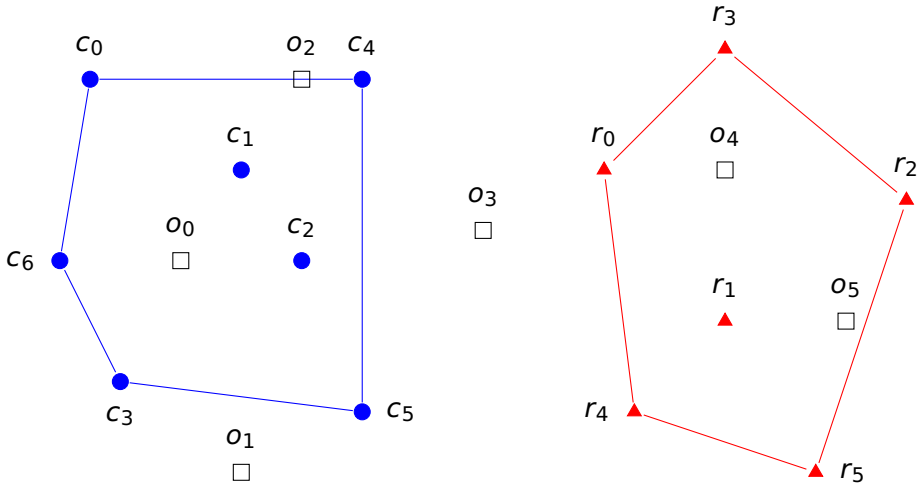
What to turn in

Turn in your source code and `Makefile`.

Examples

▸Example 1

Diagram



Input

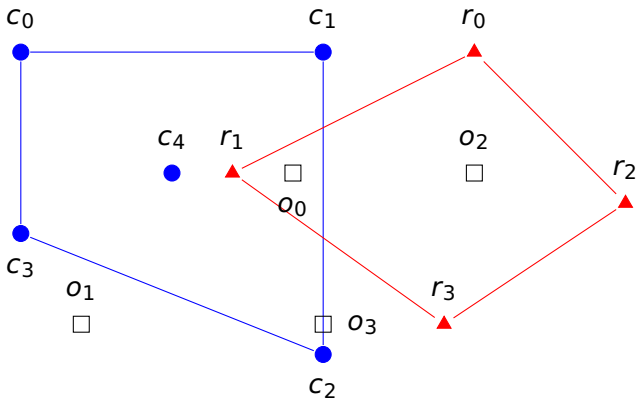
```
7 6 6
(3/5,18/5)
(8/5,3/1)
(2/1,12/5)
(4/5,8/5)
(12/5,18/5)
(12/5,7/5)
(2/5,12/5)
(4/1,3/1)
(24/5,2/1)
(6/1,14/5)
(24/5,19/5)
(21/5,8/5)
(27/5,1/1)
(6/5,12/5)
(8/5,1/1)
(2/1,18/5)
(16/5,12/5)
(24/5,3/1)
(28/5,2/1)
```

Output

```
Safe:  2
Robbed:  2
In danger:  2
```

▸*Example 2*

Diagram



Input

```
5 4 4
(1/1,3/1)
(3/1,3/1)
(3/1,1/1)
(1/1,9/5)
(2/1,11/5)
(4/1,3/1)
(12/5,11/5)
(5/1,2/1)
(19/5,6/5)
(14/5,11/5)
(7/5,6/5)
(4/1,11/5)
(3/1,6/5)
```

Output

```
Safe:  2
Robbed:  1
In danger:  1
```