Data Structures and Objects
CSIS 3700

Fall Semester 2022 — CRN 41416

Project 1 — Polygon Darts

Due date: Monday, September 26, 2022

Goal

Use the **`Fraction`** class developed in lab 2 and develop a **`Point`** class and use them to determine whether or not a set of darts hit a set of polygon shapes.

Details

▹*The* `Fraction` *class*

For this project, you should use the **`Fraction`** class you created in lab 2. 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.
- Getter methods that return the x-coordinate and y-coordinate.
- Comparison to see if two **`Point`** objects are the same or not the same; this should use **`operator==`** and **`operator!=`**.

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 should consist 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 **project1.cc** file containing other functions necessary for the program.
- A **Makefile** that directs the process of creating the executable file

▹Input

The input begins with a Point **b**; the dart board is a rectangle whose opposite corners are (0, 0) and **b**.

Following the point there is a positive integer $n \leq 100$; this is followed by n lines. Each of these lines has the form

$$k \quad \mathbf{p}_1 \quad \mathbf{p}_2 \quad \cdots \quad \mathbf{p}_k$$

where $3 \leq k \leq 20$ is the number of vertices in one polygon and the $\mathbf{p}_i$ are the vertices of the polygon, moving counterclockwise around the perimeter. Each polygon is simple (no lines cross), but may not be convex (which affects our choice of algorithm). No two polygons touch.

Next, there is a positive integer $m \leq 10$ representing the number of darts thrown. This is followed by m Points $\mathbf{d}_i$ indicating where each dart strikes the dartboard.

▹Scoring

Let A be the area of the dartboard. Let a_i be the area of polygon i. The game score is given by the following:

$$S = \sum_{p \text{ is hit}} \frac{A}{a_p}$$

where p is a polygon. Note that a polygon can only count once toward the score even if it is hit by multiple darts.

▹Output

Output the game score as a fraction.

▹*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.

▹*Calculating the area of a polygon*

Calculating polygon area is a classic problem in *computational geometry* and is rather simple to compute using the cross product. If $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k$ are the vertices of a polygon in counterclockwise order around the polygon, the area is

$$A = \frac{1}{2}\sum_{i=1}^{k} \mathbf{p}_i \times \mathbf{p}_{i+1}$$

where $\mathbf{p}_{k+1} = \mathbf{p}_1$.

▹*Determining if two line segments intersect*

Line segment intersection is another classic computational geometry problem. It is more involved than computing polygon area, but requires nothing more than vector (point) addition and subtraction, cross product and basic arithmetic.

The following algorithm determines if two segments intersect. It is modified from the general algorithm so that if either segment touches the other at the first endpoint, it counts as an intersection; however, touching at the other endpoint does not count. This prevents an edge case from messing up the algorithm for determining if a point is inside the polygon.

Algorithm 1 Determine two line segments intersect

```
1:  procedure INTERSECT(p1, p2, q1, q2, &x)          ▷ x is a reference parameter
2:      r = p2 − p1                                  ▷ Distance from p1 to p2
3:      s = q2 − q1                                  ▷ Distance from q1 to q2

4:      v = q1 − p1
5:      d = r × s

6:      if d ≠ 0 then
7:          t = (v×s)/d                              ▷ Intersection point along P in units of r
8:          u = (v×r)/d                              ▷ Intersection point along Q in units of s

9:          if 0 < t < 1and 0 < u < 1 then
10:             x ← p + r · t                        ▷ Calculate intersection point
11:             return true
12:         end if
13:     end if

14:     return false
15: end procedure
```

▹*Determining if a point is inside a polygon*

This is yet another classic computational geometry problem. The idea is to take a line segment that starts outside of the polygon and goes to the point in question, and count the number of times that segment intersects the polygon. In traveling from the known-to-be-outside point, after the first intersection, we are inside the polygon. After the second intersection, we are outside. Third intersection puts us inside, fourth one takes us outside, and this pattern repeats until we reach the point in question. If the number of intersections is odd, the point is inside. Otherwise, the point is outside.

Algorithm 2 on the next page describes this process.

▹*Tying the pieces together*

Write separate functions for the two algorithms and one to compute polygon area. Write a main function that does the input, high-level processing and score output.

What to turn in

Turn in your source code and **`Makefile`**.

Algorithm 2 Determining if a point $\mathbf{p}$ is inside polygon Q

```
1:  procedure IsInsidePolygon(p, c, q_1, q_2, ..., q_k)      ▷ c is corner of board opposite (0, 0)
2:      w ← 0                                                 ▷ Winding number

3:      v_1 ← (p.x, 0)                                        ▷ v_1 is point on board boundary directly below p
4:      v_2 ← (p.x, c.y)                                      ▷ v_2 is point on board boundary directly above p

5:      h_1 ← (0, p.y)                                        ▷ h_1 is point on board boundary directly left of p
6:      h_2 ← (c.x, p.y)                                      ▷ h_2 is point on board boundary directly right of p

7:      for i ← 1 to k do                                     ▷ q_{k+1} = q_1
8:          if Intersect(v_1, v_2, q_i, q_{i+1}, x) then      ▷ Passing directly above or below p
9:              z_1 ← x − p
10:             z_2 ← q_{i+1} − q_i

11:             if z_1.y · z_2.x < 0 then
12:                 w ← w + 1                                 ▷ Add if we are moving counterclockwise around p
13:             else
14:                 w ← w − 1                                 ▷ Subtract if we are moving clockwise around p
15:             end if
16:         end if

17:         if Intersect(h_1, h_2, q_i, q_{i+1}, x) then      ▷ Passing directly left or right of p
18:             z_1 ← x − p
19:             z_2 ← q_{i+1} − q_i

20:             if z_1.x · z_2.y > 0 then
21:                 w ← w + 1
22:             else
23:                 w ← w − 1
24:             end if
25:         end if
26:     end for

27:     if w = 0 then
28:         return false
29:     else
30:         return true
31:     end if
32: end procedure
```

Examples

▹Example 1

Input

```
(10/1,10/1)
2
3 (0/1,0/1) (3/1,0/1) (2/1,1/1)
4 (5/1,1/1) (8/1,4/1) (5/1,7/1) (2/1,4/1)
1
(4/1,9/2)
```

Output

```
Score:  50 / 9
```

▹Example 2

Input

```
(10/1,10/1)
2
3 (0/1,0/1) (3/1,0/1) (2/1,1/1)
4 (5/1,1/1) (8/1,4/1) (5/1,7/1) (2/1,4/1)
1
(4/1,1/3)
```

Output

```
Score:  0 / 1
```

▹Example 3

Input

```
(10/1,10/1)
2
3 (0/1,0/1) (3/1,0/1) (2/1,1/1)
4 (5/1,1/1) (8/1,4/1) (5/1,7/1) (2/1,4/1)
1
(2/1,1/3)
```

Output

```
Score:  200 / 3
```

▸*Example 4*

Input

```
(10/1,10/1)
2
3 (0/1,0/1) (3/1,0/1) (2/1,1/1)
4 (5/1,1/1) (8/1,4/1) (5/1,7/1) (2/1,4/1)
3
(2/1,1/3)
(5/1,4/1)
(5/1,5/1)
```

Output

```
Score:  650 / 9
```