College of Engineering, Software Engineering Department

SE 2231/36 Algorithms

Laboratory 3

Collinear Points

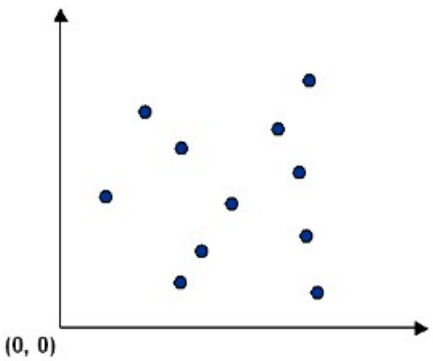
Objectives:

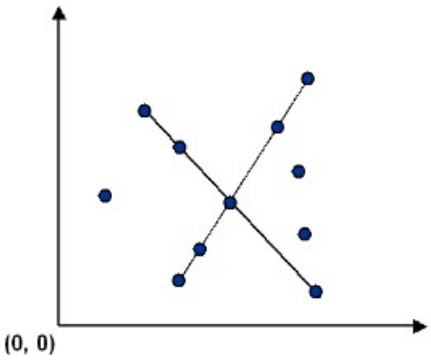
- Apply a sorting algorithm on a mathematical problem

Write a program to recognize line patterns in a given set of points.

Computer vision involves analyzing patterns in visual images and reconstructing the real-world objects that produced them. The process is often broken up into two phases: *feature detection* and *pattern recognition*. Feature detection involves selecting important features of the image; pattern recognition involves discovering patterns in the features. We will investigate a particularly clean pattern recognition problem involving points and line segments. This kind of pattern recognition arises in many other applications such as statistical data analysis.

The problem. Given a set of *n* distinct points in the plane, find every (maximal) line segment that connects a subset of 4 or more of the points.





Point data type. Create a class `Point` that represents a point in the plane by implementing the following API:

```
from __future__ import annotations
from functools import total_ordering

import turtle


@total_ordering
class Point:
    def __init__(self, x: float, y: float):
        # DO NOT MODIFY
        self.x = x
        self.y = y

    def draw(self) -> None:
        # DO NOT MODIFY
        turtle.penup()
        turtle.goto(self.x, self.y)
        turtle.dot(8, "black")

    def draw_to(self, that: Point) -> None:
        # DO NOT MODIFY
        turtle.penup()
        turtle.goto(self.x, self.y)
        turtle.pendown()
        turtle.pensize(2)
        turtle.pencolor("black")
        turtle.goto(that.x, that.y)

    def __str__(self) -> str:
        # DO NOT MODIFY
        return f"({self.x}, {self.y})"

    def _is_valid_operand(self, that: object) -> bool:
        # DO NOT MODIFY
        return isinstance(that, Point)

    def __lt__(self, that: Point) -> bool:
        # use _is_valid_operand above to check if `that` is a Point and raise
an error if it isn't a Point object
        pass

    def __eq__(self, that: Point) -> bool:
        # use _is_valid_operand above to check if `that` is a Point and raise
an error if it isn't a Point object
        pass

    def slope_to(self, that: Point) -> float:
        # calculate the slope from this to that Point
        pass
```

```
    @staticmethod
    def main():
        LENGTH = 800
        WIDTH = 600
        MARGIN = 50

        # Initialize turtle screen
        screen = turtle.Screen()
        screen.setup(LENGTH, WIDTH)

        # Transfer the point of origin to the bottom left
        screen.setworldcoordinates(-MARGIN, -MARGIN, LENGTH - MARGIN, WIDTH -
MARGIN)
        turtle.hideturtle()

        # Create points
        p1 = Point(0, 0)
        p2 = Point(150, 200)

        # Draw points
        p1.draw()
        p2.draw()

        # Draw line between points
        p1.draw_to(p2)

        # Calculate slope (uncomment this if you have implemented slope_to)
        # print("Slope between p1 and p2:", p1.slope_to(p2))

        # Keep the window open
        turtle.done()


# Example usage
if __name__ == "__main__":
    Point.main()
```

To get started, use the class `Point`, which already has some methods implemented. Your job is to implement `slope_to()`, `__lt__()`, and `__eq__()`.

- The `slope_to()` method should return the slope between the invoking point (x_0, y_0) and the argument point (x_1, y_1), which is given by the formula $(y_1 - y_0) / (x_1 - x_0)$. Treat the slope of a horizontal line segment as zero; treat the slope of a vertical line segment as positive infinity; treat the slope of a degenerate line segment (between a point and itself) as negative infinity.

- __eq__() and `__lt__()` should compare points by their *y*-coordinates, breaking ties by their *x*-coordinates. Formally, the invoking point (x_0, y_0) is *less than* the argument point (x_1, y_1) if and only if either $y_0 < y_1$ or if $y_0 = y_1$ and $x_0 < x_1$.

Corner cases. To avoid potential complications with integer overflow or floating-point precision, you may assume that the constructor arguments *x* and *y* are each between 0 and 32,767. (*You may have to change the dimensions of the screen in order to support up to 32,767*)

Line segment data type. To represent line segments in the plane, use the class LineSegment which has the following API:

```
from __future__ import annotations
from point import Point

import turtle


class LineSegment:
    def __init__(self, p: Point, q: Point):
        # DO NOT MODIFY
        self.p = p
        self.q = q

    def draw(self) -> None:
        # DO NOT MODIFY
        turtle.penup()
        turtle.goto(self.p.x, self.p.y)
        turtle.pendown()
        turtle.pensize(2)
        turtle.pencolor("black")
        turtle.goto(self.q.x, self.q.y)

    def __str__(self) -> str:
        # DO NOT MODIFY
        return f"{self.p} -> {self.q}"

    @staticmethod
    def main():
        LENGTH = 800
        WIDTH = 600
        MARGIN = 50

        # Initialize turtle screen
        screen = turtle.Screen()
        screen.setup(LENGTH, WIDTH)

        # Transfer the point of origin to the bottom left
```

```
        screen.setworldcoordinates(-MARGIN, -MARGIN, LENGTH - MARGIN, WIDTH -
MARGIN)
        turtle.hideturtle()

        # Create points
        p1 = Point(50, 100)
        p2 = Point(150, 200)

        # Create a line segment
        line = LineSegment(p1, p2)

        # Draw the line segment
        line.draw()

        # Keep the window open
        turtle.done()


# Example usage
if __name__ == "__main__":
    LineSegment.main()
```

Brute force. Write a class `BruteCollinearPoints` that examines 4 points at a time and checks whether they all lie on the same line segment, returning all such line segments. To check whether the 4 points *p*, *q*, *r*, and *s* are collinear, check whether the three slopes between *p* and *q*, between *p* and *r*, and between *p* and *s* are all equal.

```
from line_segment import LineSegment
from point import Point


class BruteCollinearPoints:
    def __init__(self, points: list[Point]):
        # YOUR CODE HERE
        pass

    def number_of_segments(self) -> int:
        # YOUR CODE HERE
        pass

    def segments(self) -> list[LineSegment]:
        # YOUR CODE HERE
        pass
```

The method `segments()` should include each line segment containing 4 points exactly once. If 4 points appear on a line segment in the order $p \to q \to r \to s$, then you should include either the line segment $p \to s$ or $s \to p$ (but not both) and you should not include *subsegments* such as $p \to r$ or $q \to r$. For simplicity, we will not

supply any input to `BruteCollinearPoints` that has 5 or more collinear points.

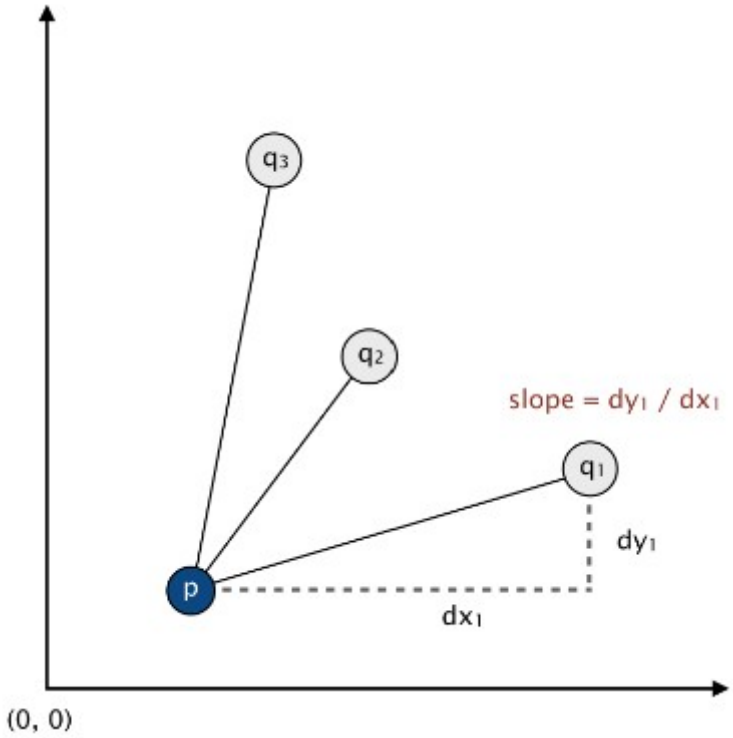
Corner cases. Raise an `Error` if the argument to the constructor is `None`, if any point in the array is `None`, or if the argument to the constructor contains a repeated point.

Performance requirement. The order of growth of the running time of your program should be n^4 in the worst case.

A faster, sorting-based solution. Remarkably, it is possible to solve the problem much faster than the brute-force solution described above. Given a point p, the following method determines whether p participates in a set of 4 or more collinear points.

- Think of p as the origin.
- For each other point q, determine the slope it makes with p.
- Sort the points according to the slopes they make with p.
- Check if any 3 (or more) adjacent points in the sorted order have equal slopes with respect to p. If so, these points, together with p, are collinear.

Applying this method for each of the n points in turn yields an efficient algorithm to the problem. The algorithm solves the problem because points that have equal slopes with respect to p are collinear, and sorting brings such points together. The algorithm is fast because the bottleneck operation is sorting.



Write a class `FastCollinearPoints` that implements this algorithm.

```
from line_segment import LineSegment
from point import Point


class FastCollinearPoints:
    def __init__(self, points: list[Point]):
        # YOUR CODE HERE
        pass

    def number_of_segments(self) -> int:
        # YOUR CODE HERE
        pass

    def segments(self) -> list[LineSegment]:
        # YOUR CODE HERE
        pass
```

The method `segments()` should include each *maximal* line segment containing 4 (or more) points exactly once. For example, if 5 points appear on a line segment in the order $p \to q \to r \to s \to t$, then do not include the subsegments $p \to s$ or $q \to t$.

Corner cases. Raise an `Error` if the argument to the constructor is `None`, if any point in the array is `None`, or if the argument to the constructor contains a repeated point.

Performance requirement. The order of growth of the running time of your program should be $n^2 \log n$ in the worst case. `FastCollinearPoints` should work properly even if the input has 5 or more collinear points.

Sample client. This client program takes the name of an input file as a command-line argument; read the input file (in the format specified below); prints to standard output the line segments that your program discovers, one per line; and draws to standard draw the line segments.

```
import turtle

from fast_collinear_points import FastCollinearPoints
from point import Point
from sys import argv


# Main program
def main(filename):
    # Read points from file
    with open(filename, 'r') as f:
        n = int(f.readline().strip())
        points = []
        for _ in range(n):
            x, y = map(int, f.readline().strip().split())
            points.append(Point(x, y))

    # Draw the points
    LENGTH = 32678
    WIDTH = 32678

    # Initialize turtle screen
    screen = turtle.Screen()
    screen.setup(LENGTH, WIDTH)

    # Transfer the point of origin to the bottom left
    turtle.setworldcoordinates(0, 0, LENGTH, WIDTH)
    turtle.hideturtle()

    for p in points:
        p.draw()

    # Print and draw the line segments
    collinear = FastCollinearPoints(points)
    for segment in collinear.segments():
```

```
        print(f"Line segment between ({segment.p.x}, {segment.p.y}) and
({segment.q.x}, {segment.q.y})")
        segment.draw()

    turtle.done()


if __name__ == "__main__":
    # Call this file with `python client.py <txt_file_name>`
    # e.g. python client.py input.txt
    main(argv[1])
```

Input format. We supply several sample input files (suitable for use with the test client above) in the following format: An integer *n*, followed by *n* pairs of integers (*x*, *y*), each between 0 and 32,767. Below are two examples.

```
% cat input6.txt
6
19000  10000
18000  10000
32000  10000
21000  10000
 1234   5678
14000  10000
```

```
% cat input8.txt
8
 10000      0
     0  10000
  3000   7000
  7000   3000
 20000  21000
  3000   4000
 14000  15000
  6000   7000
```

We can expect
the output (in Java) to look similar to this:

```
% java-algs4 BruteCollinearPoints input8.txt
(10000, 0) -> (0, 10000)
(3000, 4000) -> (20000, 21000)

% java-algs4 FastCollinearPoints input8.txt
(3000, 4000) -> (20000, 21000)
(0, 10000) -> (10000, 0)

% java-algs4 FastCollinearPoints input6.txt
(14000, 10000) -> (32000, 10000)
```

Web submission. Submit a .zip file containing only `brute_collinear_points.py`, `fast_collinear_points.py`, and

`point.py`. You can easily use `sort()` on the array you'd like to have sorted.
Deadline: March 31, 2025, 11:59pm

Submit to https://www.dropbox.com/request/RKKrzEioh90WrHpobC2M

Notes: You can import `itertools`. Make sure to let me know if you're planning to use other libraries/packages.