Lab 3: part 2

Course goals

- To implement and use data structures and algorithms in application programs.
- To analyze and evaluate various data structures and algorithms in solving computational problems with respect to efficiency and appropriateness.

Getting started

To get started with the exercise, perform the steps indicated below.

- Create a folder for this lab part named e.g. `Lab3-part2`.
- Download the zipped folder with the files for this exercise from the course website and unzip it into the lab folder.
- Navigate to the folder `detectionSystem` and execute the instructions in the file `README.md` (can be opened with Notepad). Your IDE should then show two projects, `LinesDiscovery` and `Rendering`.
- Select `LinesDiscovery` as startup project. Compile, link, and execute the program (the `main` is in `find-patterns.cpp`). When requested for a points file, enter e.g. `points200.txt`. As the system for detecting lines in a set of points is not implemented yet, the program produces no visible output. This implementation is assigned as part of an exercise.
- Select `Rendering` as startup project. Compile, link, and execute the program (the `main` is in `lab3-part2.cpp`). When requested for a points file, enter e.g. `points200.txt`. All points in the input file are then plotted. Note that this program only provides plotting functionality, for both input points and line segments[1]. You don't need to modify this program — it's ready to be used.
- Read the remaining lab description.
- Review lecture 8.

If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "`TND004: ...`".

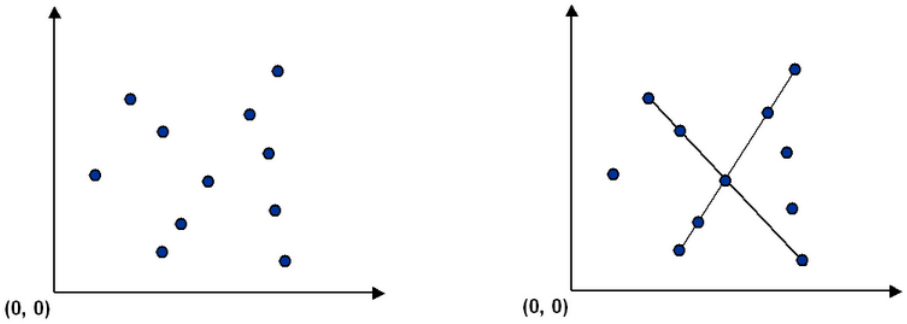
Exercise 1: To find patterns in large datasets of points

In many real-world applications, data is captured as collections of points — whether from sensors, images, or spatial measurements. A fundamental challenge is to infer the underlying structures that generated those points. **Identifying collinear points** is a key step in this process, as they often reveal the presence of **linear features or alignments** in the data. For example, in geographic information systems (GIS), groups of collinear points may represent roads, pipelines, rivers, or power lines — features critical for mapping and spatial analysis. In interactive environments like computer games, detecting collinear arrangements can enable strategic behaviors, such as targeting multiple enemies along a straight path.

[1] At the start of this exercise, there are no line segments yet discovered from the points files. Thus, no line segments are plotted, yet.

This exercise focuses on precisely that problem:

1. Given a dataset of $n > 0$ distinct $2D$ points, find every line segment that connects four or more of them, as illustrated below.



The brute-force approach would be to check all combinations of 4 points, which requires $O(n^4)$ time, and is clearly inefficient for large n.

```
for (i1 = 0; i1 < size(points) - 3; ++i1) {
    for (i2 = i1 + 1; i2 < size(points) - 2; ++i2) {
        for (i3 = i2 + 1; i3 < size(points) - 1; ++i3) {
            for (i4 = i3 + 1; i4 < size(points); ++i4) {
                if (onSameLine(points[i1], points[i2], points[i3], points[i4])) {
                    // …
                }
            }
        }
    }
}
```

It is possible to solve the problem much faster than the brute force solution. An algorithm is presented in the next section.

A sorting-based algorithm

You should implement the following algorithm which involves sorting. For each point p, the following method determines whether p belongs to a set of 4 or more collinear points.

1. For each other point q, consider the slope of the line segment that connects point p with q[2].
2. Sort the points according to the slope with point p.
3. Check if any 3 (or more) adjacent points in the sorted order have the same slopes with p. If so, these points, together with p, are collinear and form a line segment that connects 4 or more distinct points.

   Note that the sorting places together the points that have the same slope with respect to point p.

Implement the above-described algorithm and insert your code into the file named `find-patterns.cpp` within the `linesDiscovery` project.

[2] The line defined by the points $p = (x_1, y_1)$ and $q = (x_2, y_2)$ has slope $\frac{(y_2 - y_1)}{(x_2 - x_1)}$.

Input data files

Several input files with 2D-points are provided in the folder `detectionSystem/data`. (e.g. `points1.txt`, `points5.txt`, `largeMystery.txt`). These text files contain points coordinates which are integers in the interval $[0, 32767]$. Each input file starts with the number of points ($n > 0$) followed by the points coordinates, one point per line. You can assume there are no duplicate points in the input files.

Output data files

Given an input points file, your program should produce two output text files.

1. A file with the line segments to be rendered.
   - Name the file based on the input file. For instance, if the input file is named `points6.txt` then the line segments file must be named `segments-points6.txt`. Place the file in the `detectionSystem/data/output` folder so the rendering program can locate it.
   - Each line in this file should contain the coordinates of two points—four integers in total—representing the start and end of a line segment (see example below).

```
10000 0 0 10000
3000 4000 20000 21000
```

2. A file listing the discovered line segments, each accompanied by the data points that lie on it. Save this file in a subfolder that you create inside the `detectionSystem/data/output` directory.

Requirements

Your program must satisfy the following requirements.

- Run in $O(n^2 \log n)$ time.
- Execute in $O(n + m)$ extra space, where $n > 0$ is the number of input points and $m > 0$ is the number of non-redundant line segments.

Rendering system

To render the points and line segments, select `Rendering` as startup project and execute the program.

In the folder `detectionSystem/data/output/line-plots`, you can also find `png` files with the plot of the line segments discovered from the corresponding points files, with exception for the input file `largeMystery.txt`.

Hints

- Start by sorting the input points first by the y-coordinate and then by the x-coordinate (e.g. point $(14080, 12032)$ appears before point $(13056, 13056)$, while point $(13120, 20224)$ appears before $(14080, 20224)$).
- Standard library functions in C++ should be utilized for sorting operations.
- Three points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, and $p_3 = (x_3, y_3)$ are in the same line, if $((y_2 - y_1) \times (x_3 - x_1)) = ((y_3 - y_1) \times (x_2 - x_1))$. Note that these expressions

do not involve floating-point computations and should be used whenever possible.

- The sorting (based on slope) in step 2 of the algorithm involves floating-point comparisons[3]. You don't need to take any special action for this.

Exercise 2: Filtering redundant results

Your program may be generating redundant line segments, which can also slow down the rendering process. Let's work on optimizing the code to address this.

For each input points file, the expected discovered line segments (accompanied by the data points that lie on it) are available in the folder `detectionSystem/data/output/lines-discovered`. The points forming the line segments are sorted first by the y-coordinate and then by the x-coordinate.

Start by comparing the line segments discovered by your program (file in point 2 of "Output data files") with the corresponding given file in the folder mentioned above. Verify whether your program lists redundant line segments.

The output list of line segments is considered non-redundant if the following two criteria are met.

- The list contains only one representation of each line segment (e.g. different permutations of the same points should be excluded).
- Subsegments of a line (or of one of its permutations) should not be part of the output, either.

For example, consider the data points in the input file `points6.txt`. Then, the program should only output the following line.

```
(14000,10000)->(18000,10000)->(19000,10000)->(21000,10000)->(32000,10000)
```

Including any of the following in the list would have made it redundant.

```
(14000,10000)->(18000,10000)->(19000,10000)->(21000,10000)
(18000,10000)->(19000,10000)->(21000,10000)->(32000,10000)
(19000,10000)->(32000,10000)->(21000,10000)->(14000,10000)->(18000,10000)
```

Modify your program to eliminate redundancy. To assist with this, you are encouraged to use an AI tool (e.g. Copilot) to explore strategies for eliminating redundancy. Be sure to document the prompts you used and the responses provided by the chatbot, as you will need to present them during the lab redovisning.

Exercise 3: Time and space complexity analysis

Provide a written analysis of your program's time and space complexity to demonstrate that it meets the specified requirements.

Presenting part2 and deadlines

The exercises in this lab part are compulsory and you should demonstrate your solutions during the lab session *Lab3 RE*. Read the instructions given in the labs webpage and consult the course schedule.

[3] The slope should be a `double`.

Necessary requirements for approving your lab are given below.

- The code must be readable, well-indented, and use good programming practices.
- Present your implementation for the sorting-based algorithm.
- Explain the strategy your program uses to remove redundant line segments from the output. Additionally, include examples of the prompts you used when consulting your AI assistant, along with the responses you received.
- Present the time and space complexity analysis. Remember to bring this part also written. The lab assistant may require you to hand-in the analysis part for further feedback. Do not forget to indicate the name plus LiU-id of each group member. Unreadable answers will be rejected.
- Answer the question: what pattern is hidden in the points file `largeMystery.txt`?

If your solutions for the exercises in lab3/part2 have not been approved in the scheduled lab session *Lab3 RE* then it is considered a late lab. Late labs can be presented provided there is time in a another RE lab session. All groups have the possibility to present one late lab on the extra RE lab session scheduled in the end of the course.