HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Fall 2017

T-301-REIR, Reiknirit

S2: Pattern Recognition

Birgitta Feldís Bjarkadóttir
kt. 180395-3189
Edda Steinunn Rúnarsdóttir
kt. 241095-2909

Group 2

September 15, 2017

TA: Hannes Kristján Hannesson

1 Implementation

Our project determines collinear points and uses two distinct algorithms to do so: a brute force algorithm and a more optimized sorting solution algorithm.

Brute Force

Our brute force algorithm **Brute.java** uses a not-too-elegant algorithm that takes in number of points followed by the points' coordinates x and y and prints all collinear four points. The points are represented by a Point object implemented in **Point.java**. The object has x and y integer variables denoting coordinates. The brute force algorithm completely depends on this data type and it's methods, specifically *compareTo( )* and *slopeTo( )* methods.

compareTo( ) takes in another Point object as a parameter and checks if given point is lexicographically larger or smaller than the parameter point (that is, higher on the y-axis). The function returns -1 if the parameter point is larger and 1 if it is smaller. If the y coordinates of the points the x-coordinates are used as a tie breaker (higher x coordinate is considered lexicographically larger in this case). If the points are exactly the same, the function returns 0.

The *slopeTo ( )* method calculates slope from given point to another parameter point. The function checks edge cases first; whether line is degenerate (returns negative infinity), vertical (returns positive infinity) or horizontal (returns 0). If none of these conditions apply the slope of the two points is returned and calculated.

The first thing necessary to implement the brute force algorithm was reading each of the points from an input file into a Point object array to keep it accessible. This was done by using the code already implemented by teacher in the class Point with a few changes.

Then all possible combinations of four points were extracted via a 4Sum algorithm. After extracting all possible combinations of four points, a function called *areCollinear( )* checked whether their slopes were equal and therefore collinear. The function uses the *SlopeTo( )* function for a Point object to check if the slope between each of the four points is the same, meaning the four points are collinear and if so returns true. If four collinear points are found they are sorted lexicographically and added to the list *allpoints* (via *addToList( )* function). Each adjacent four elements in the list *allpoints* represents an array of size four with four collinear points.

Finally the entire list was printed array-by-array, sorted by their lowest point. This was achieved by creating a function called *printSorted( )* which uses selection sort to sort the arrays before printing. The swapping in the selection sort algorithm is done by a helper function *swaparr( )* which takes in an index i and j, both denoting the first index of arrays to be swapped. Then arrays are swapped, element by element. The *printSorted( )* function takes in an array which includes all points, sorts them and prints them out in increasing order.

Sorting solution

A method called Fast, implemented in **Fast.java** achieves the sorting solution for our project. Fast initially reads points into a Point array from an input file in the exact same way as in the brute force solution.

All points are sorted by their slope in respect to an origin point and stored as such in an array. The sorting is achieved via the *Array.sort( )* method that uses a comparator implemented in **Point.java**. The comparator compares the slope of two points in respect to an origin point; it returns either 1 or -1 for different slopes, indicating which point has larger slope in respect to the origin point and 0 for equal slopes.

After constructing the list this way, the list is iterated and the *slopeTo( )* function is used to check if adjacent points have equal slopes in respect to the origin point (as equal slopes are inevitably adjacent after the sort). We preserve points that have equal slopes in a list. Once a point that does not have an equal slope in respect to the origin point, we print the list out as a set of collinear points, that is, if the list has three or more points (in addition to the origin point) an then re-initiate the list. After this process is over, we check whether the list contains unprinted set of collinear points and print the set out if applicable. This process is repeated using each point as origin point and this way all possible sets of collinear points are extracted.

The sorting solution was then optimized in **Fast2.java** so that the solution does not print out different representations of the same line segment when five or more points are on a line segment. This was done by adding a boolean function check whenever a line segment was to be printed. The boolean function, *alreadyFound( )* takes in already found lines and the current line segment. It checks whether first two indices of the current line are found adjacent in the already found lines list. If so, the current line segment is a subsegment of an already found line. If not, the line is new and can be printed.

2 Empirical Analysis

To observe the differences more clearly between the two algorithms and their efficiency, an experiment was conducted on the run time of the two different types of algorithms implemented and the results were documented. Then a tilde notation was derived from analyzing code structure.

The less optimized brute-force solution had impossible time complexity and the run time could not be measured beyond N = 800 points (an experiment was considered to "time-out" once it had passed 180 seconds). The sorting solution had far better time complexity. See results of run time experiments in table 1.

Table 1: *Running times for the two algorithms*

N	brute	sorting
150	0.42 s	0.02 s
200	1.2 s	0.03 s
300	5.6 s	0.05 s
400	17.3 s	0.2 s
800	*test timeout*	0.3 s
1600	*test timeout*	1.1 s
3200	*test timeout*	2.34 s
6400	*test timeout*	6.41 s
12800	*test timeout*	23.4 s

Running time was determined by applying to both experiment sets the following equation for time complexity: $T(X_i) = aX_i^b$ whereas b was determined as $log(T(X_i)/T(X_{i-1}))$ where X_i is value for variable X in a single experiment $_i$ and $T(X)$ denotes running time for variable X.

Brute: $6.76 * 10^{-10} * N^4$ which yields the tilde notation: $\sim \mathbf{N^4}$.

Sorting: $1.25 * 10^{-7} * N^2$ which yields the tilde notation: $\sim \mathbf{N^2}$

Theoretical For practical reasons the program's order of growth of the worst-case running time was analyzed:

Brute: $\mathbf{O(N^4)}$ is the worst case for the brute force algorithm; we have four loops that are nested that in the worst case all loop N times, so $N * N * N * N = N^4$ - other loops or operations are irrelevant to the order of growth so the order of growth is therefore represented via O notation as $O(N^4)$.

Sorting: $\mathbf{O(N^2 log(N)}$ is the worst case order of growth for the selection sort algorithm; a loop that in the worst case loops N times has a nested loop that in the worst case loops $log(N)$ times, then within that, a function checks whether a line segment is a subsegment of another that in the worst case has time complexity of N. Therefore the order of growth is in the worst case $O(N^2 log(N))$.

3 About This Solution

Have you taken (part of) this course before: **No, neither one of us**
Hours to complete assignment (optional): **15+**

3.1 Known Bugs/Limitations

No known bugs are present in the project.

3.2 Help Received

Any help recieved for the project was in lab class from the TA *(Tómas Ken Magnússon)* who helped us debug minor errors that we had problems with debugging.

3.3 Problem Encountered

No serious problems were encountered when solving the project (apart from misunderstanding the assignment repeatedly, our bad). The assignment went OK as a whole although implementation and understanding concepts was extremely time consuming.