Algorithmics II (H)

Tutorial Exercises on Geometric Algorithms

Outline Solutions

1. Here is a suitable definition of the type `Line`:

```
public class Line {
   public Point2D.Double p1,p2;
}
```

The definition `intersect` uses the fact that lines l_1 and l_2 intersect if and only if the gradients of l_1 and l_2 are not equal. Recall that a line through points (x_A, y_A) and (x_B, y_B) has gradient $(y_B - y_A) / (x_B - x_A)$.

```
/** Returns true if the lines l1 and l2 intersect, and
  * returns false otherwise */
public boolean intersect(Line l1, Line l2) {
   return ( (l1.p2.y-l1.p1.y) * (l2.p2.x-l2.p1.x))
       != ( (l1.p2.x-l1.p1.x) * (l2.p2.y-l2.p1.y) );
}
```

2. Here is a suitable definition of the type `LineSegment`:

```
public class LineSegment {
   public Point2D.Double p1,p2;
}
```

The function `boundingBox` may then be defined as follows:

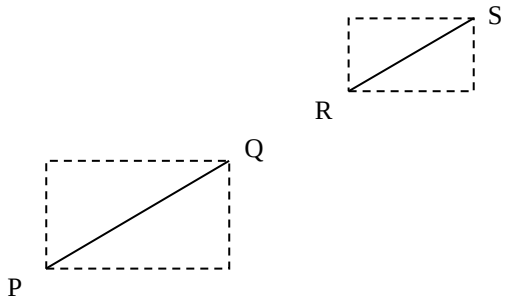
```
/** returns true if the rectangles containing line segments
  * l1 and l2 intersect, and returns false otherwise*/
private boolean boundingBox(LineSegment l1,LineSegment l2) {

   double x1,x2,x3,x4,y1,y2,y3,y4;
   x1 = Math.min(l1.p1.x, l1.p2.x);
   x2 = Math.max(l1.p1.x, l1.p2.x);
   x3 = Math.min(l2.p1.x, l2.p2.x);
   x4 = Math.max(l2.p1.x, l2.p2.x);
   y1 = Math.min(l1.p1.y, l1.p2.y);
   y2 = Math.max(l1.p1.y, l1.p2.y);
   y3 = Math.min(l2.p1.y, l2.p2.y);
   y4 = Math.max(l2.p1.y, l2.p2.y);

  /* Rectangle containing l1 has bottom LH corner (x1,y1)
   * and top RH corner (x2,y2)
   * Rectangle containing l2 has bottom LH corner (x3,y3)
   * and top RH corner (x4,y4) */

   return x4 >= x1 && y4 >= y1 && x2 >= x3 && y2 >= y3;
}
```

3. Two collinear line segments that do not intersect, as shown in the example below, need the `boundingBox` test in addition to the `onOppositeSides` test, in order to correctly conclude that they do not intersect.



Notice that each of the function calls `onOppositeSides(p,q,—r—s—)` and `onOppositeSides(r,s,—p—q—)` returns the value `true`. However the function call `intersect(p—q,r—s)` as defined in lectures correctly returns the value `false`, since `boundingBox(p—q,r—s)` returns the value `false`.

4. Mark each endpoint of the line segments with 'L' or 'R' depending on whether it is a left or right endpoint respectively. Sort the all 2*n* endpoints in non-decreasing order of *x*-coordinate in O(*n*log *n*) time. During sorting we can detect intersections between pairs of line segments that share the same endpoints only. If no endpoints match, we will have a sequence of 2*n* distinct characters 'L' and 'R' when reading the sorted endpoints from left to right. The intervals are disjoint if and only if the characters appear in alternating order in the sequence. After sorting, the check for alternating characters takes O(*n*) time, so overall we need O(*n* log *n*) time to decide the problem.

5. We firstly note that a rectangle may be represented by two points: the bottom left point p_1 and the top right point p_2. This observation leads to the following definition of the type `Rectangle`:

```
public class Rectangle {
   public Point2D.Double p1,p2;
   // assume that p1.X ≤ p2.X and p1.Y ≤ p2.Y
}
```

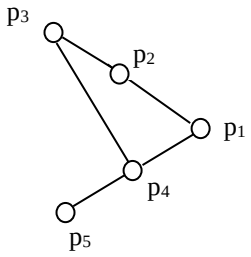
Now let r_1 and r_2 be two rectangles that are known to intersect. We may calculate the co-ordinates of their area of intersection using the function `areaIntersect`, as follows:

```
private Rectangle areaIntersect(Rectangle r1, Rectangle r2) {
   Rectangle r3 = new Rectangle();
   r3.p1.x = Math.max(r1.p1.x, r2.p1.x);
   r3.p1.y = Math.max(r1.p1.y, r2.p1.y);
   r3.p2.x = Math.min(r1.p2.x, r2.p2.x);
   r3.p2.y = Math.min(r1.p2.y, r2.p2.y);
   return r3;
}
```

To solve the problem specified by the question, suppose that we are given a set S of n rectangles. Take any pair of rectangles r_1 and r_2 that belong to S. Using the function `boundingBox` from Question 2, we may check whether r_1 and r_2 intersect. If not, we may halt immediately and report that the locus of intersection of all rectangles is empty. Otherwise the nonempty locus of intersection of r_1 and r_2 may be calculated using the function `areaIntersect` defined above. This defines a new rectangle which may be added to the set. We now have n-1 rectangles and may continue in this way, until we either halt with an empty locus of intersection, or ultimately we end up with a nonempty locus of intersection of all rectangles in S. Clearly this algorithm is of O(n) time complexity.

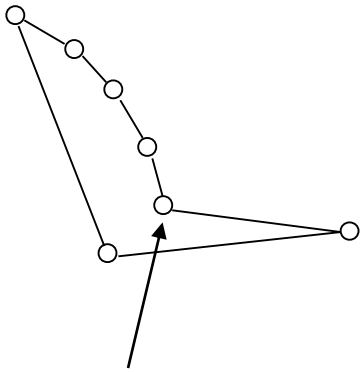
The following is an alternative solution (suggested by a student), with the same time complexity: suppose $S=\{r_1,r_2,\ldots,r_n\}$ is the set of n rectangles. Suppose that r_i has lower left-hand corner coordinates (x_{LL}^i,y_{LL}^i) and upper right-hand corner coordinates (x_{UR}^i,y_{UR}^i) $(1\le i\le n)$. Let $x_{LL}^*=\max\{x_{LL}^i : 1\le i\le n\}$, $y_{LL}^*=\max\{y_{LL}^i : 1\le i\le n\}$, $x_{UR}^*=\min\{x_{UR}^i : 1\le i\le n\}$ and $y_{UR}^*=\min\{y_{UR}^i : 1\le i\le n\}$. If $x_{LL}^* > x_{UR}^*$ or $y_{LL}^* > y_{UR}^*$ then we can report that there is no common area of intersection of all the rectangles in S. Otherwise the rectangle r^* with lower left-hand corner coordinates (x_{LL}^*,y_{LL}^*) and upper right-hand corner coordinates (x_{UR}^*,y_{UR}^*) is the common area of intersection of all the rectangles in S. To see this, we can check that r^* intersects with r_i, for each i $(1\le i\le n)$. This is true because $x_{LL}^* \le x_{UR}^* \le x_{UR}^i$, and $x_{UR}^* \ge x_{LL}^* \ge x_{LL}^i$, and $y_{LL}^* \le y_{UR}^* \le y_{UR}^i$, and $y_{UR}^* \ge y_{LL}^* \ge y_{LL}^i$. (Compare these inequalities with the solution to Question 2, with (x_{LL}^i,y_{LL}^i) and (x_{UR}^i,y_{UR}^i) playing the role of (x_1,y_1) and (x_2,y_2) respectively, and (x_{LL}^*,y_{LL}^*) and (x_{UR}^*,y_{UR}^*) playing the role of (x_3,y_3) and (x_4,y_4) respectively.)

6. Recall that, during the course of the execution of the simple polygon construction algorithm, if collinear points are discovered, the points are indexed in increasing order of distance from the pivot. However if these points are collinear with the final point discovered, then this ordering principle will cause the algorithm to fail to return a simple polygon, as the following example shows (in the example, p_1 is the pivot):



The following modification to the algorithm will take account of this special case: in the case of points that are collinear with the final point discovered, the points should be indexed in decreasing order of distance from the pivot. All other cases of collinear points should be dealt with as before.

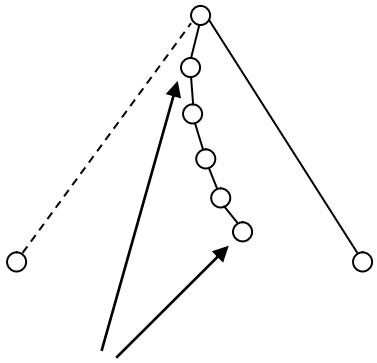
7. The diagram below shows an example that would result in an incorrect convex hull computation. It is clear from this example that arbitrarily many points may be wrongly included in the hull if the algorithm does not start at an extreme point.



starting point

8. The ‘bottleneck’ in the Graham Scan algorithm is actually the sorting phase, which is O(n log n) in comparison with the O(n) behaviour of the scan itself. Hence, strictly speaking, the efficiency of the algorithm will be at its worst for a set of points that causes the sort to do most work. This obviously depends heavily on just which sorting algorithm is used.

   If the question is referring only to the scan phase of the algorithm, then worst-case behaviour will arise when the maximum number of points are provisionally placed on the hull, but then subsequently removed. The diagram shows an example.



these 5 points excluded at next step

9. It is straightforward to amend the algorithm to deal with the case of the furthest pair of points of the same colour. First find the convex hull of the set of red points. The furthest pair of red points are on this hull, and can be found in O(r) time, where r is the number of points on this hull. Similarly find the convex hull of the set of blue points. The furthest pair of blue points are on this hull, and can again be found in O(s) time, where s is the number of points on this hull. One final comparison of the furthest pair of red points and the furthest pair of blue points will reveal the furthest pair of points of the same colour.

   On the other hand, if we find the convex hull of the set of all points, it is perfectly possible that all of the points on it are the same colour. In such a case it is certainly not true that the furthest pair of differently coloured points lie on the convex hull of the set of all points. In this case we need to consider two convex hulls, one for each point set, and use a variant of the rotating calipers algorithm (see https://doi.org/10.1016/0196-6774(83)90040-8 for more details; this is outwith the scope of the course).

10. Statement a) is true, since every time the closest pair distance d is updated within the nested for loops, we can note the pair of points a, b that were responsible for the update.

    Statement b) is false: for example if the point set contained only two points, with coordinates (0,-1) and (0,1), the closest pair distance is clearly 2, but the initial value of d would be 1, which is never improved upon and hence the closest pair distance would be incorrectly reported to be 1.

    Statement c) is true. Run the Closest Pair algorithm once to find the closest pair of points $\{p, q\}$ – these points must be unique since the distances between every pair of points are distinct. A second-closest pair of points $\{r,s\}$ must then satisfy the property that either $p \notin \{r, s\}$ or $q \notin \{r, s\}$, or both. Hence run the Closest Pair algorithm with $P' = P \setminus \{p\}$ and suppose that points $\{w, x\}$ are returned. Next run the Closest Pair algorithm with $P'' = P \setminus \{q\}$ and suppose that points $\{y, z\}$ are returned. If $dist(w, x) < dist(y, z)$ then return $\{w, x\}$ as the second-closest pair, else return $\{y, z\}$. Clearly the process takes $O(n \log n)$ time overall, using three executions of the Closest Pair algorithm.

11. Sort the points by increasing x-coordinate, breaking ties by increasing y-coordinate; e.g. (1,7), (1,14), (1,20), (2,11), (3,4), (3,8), (4,9), (6,2), (7,15), (8,4). Scan the sorted list in reverse, remembering largest y coordinate seen, say y_{max}. The first point is necessarily maximal, and subsequent points will be maximal if and only if they result in an update of y_{max}. The complexity is dominated by the sort, so is $O(n \log n)$, the scan being merely $O(n)$. For this example, the maximal points are (8,4), (7,15), (1,20).

12. First sort the points by increasing x-coordinate, breaking ties by increasing y-coordinate. This can be achieved in $O(n \log n)$ time. In this sorted set, we can search for any particular point in $O(\log n)$ time by binary search. For each pair P, Q of the given set of points, the coordinates of the mid point R of the line segment P—Q can be computed from the coordinates of P and Q in O(1) time, and we can then search the ordered sequence for R in $O(\log n)$ time. There are $n(n-1)/2 = O(n^2)$ possible choices for the pair P,Q. Hence we can try out all possible pairs in $O(n^2 \log n)$ time as required.

13. As a preliminary step, sort the points on their x-coordinate, breaking ties by comparison of the y coordinate. In this sorted sequence it is then possible to use binary search to determine, in $O(\log n)$ time, whether a particular point (x,y) is present. Now take each pair of points (p,q) and (r,s) in turn, where $p<r$ and $q<s$, and check whether they could potentially be on opposite corners of a square, i.e., whether $s-q=r-p$. If so, the other two corners would be (p,s) and (r,q). Thus we can calculate in O(1) time the coordinates of the other points to check, and then use our binary search to establish in $O(\log n)$ time whether these are actually present in the given point set. Since the number of pairs to be checked is $n(n-1)/2 = O(n^2)$, and each check costs $O(\log n)$ time, the overall complexity is $O(n^2 \log n)$.

    Note that this technique will work even if the sides of the square are not required to be horizontal and vertical. In this case if we assume that (p,q) and (r,s) are at opposite corners, the other two points to search for are as follows:

    $$((p+r+s-q)/2, (q+s+p-r)/2) \text{ and } ((p+r+q-s)/2, (q+s+r-p)/2).$$

14. We split up the problem into three sub-problems, involving finding all intersections between the following types of line segments:

    (a) horizontal and vertical line segments;
    (b) vertical and 45° line segments;
    (c) horizontal and 45° line segments.

    Case (a) can be solved using the line-sweep technique as described in lectures. Case (b) may be solved as follows. For every 45° line segment L, we calculate p_L, the x-coordinate of the point of intersection between the x-axis and the line containing L (if (x_1,y_1) is an endpoint of L, then $p_L=x_1-y_1$). We use these values in order to formulate a horizontal line sweep algorithm for case (b) that extends the line-sweep algorithm for case (a), in the following way.

    As before, vertical line segments are represented by one of their endpoints, whilst 45° line segments are represented by both endpoints. All endpoints are then sorted according to x-coordinate (breaking ties so that a 45° line's left endpoint comes before a vertical line endpoint, which comes before a 45° line's right endpoint). We maintain an AVL tree T_D for 45° line segment candidates. The key corresponding to a 45° line segment L to be stored in T_D will be p_L. The endpoints are now processed in increasing order of x-coordinate.

    Firstly, suppose that q is an endpoint belonging to a 45° line segment L. If q is the left endpoint of L, L is added to T_D. Otherwise q is the right endpoint of L, so that L is removed from T_D.

    Secondly, suppose that q is an endpoint belonging to a vertical line segment L. We check for intersections between L and horizontal and 45° line segment candidates as follows. Suppose that (x,y_1) and (x,y_2) are the two endpoints of L, where $y_1 \geq y_2$. In order to find all intersections between L and 45° line segment candidates, we range search T_D with $p_1..p_2$, where p_i is the intersection of the 45° line through (x,y_i) with the x-axis (i=1,2) (in practice $p_1=x-y_1$ and $p_2=x-y_2$). The intuition behind the latter range search is that L intersects with a 45° line segment L' if and only if $p_1 \leq p_{L'} \leq p_2$ (recall from above the definition of $p_{L'}$). Using this procedure, we calculate all intersections between the vertical and 45° line segments.

    It remains to calculate case (c) intersections, i.e. those between horizontal and 45° line segments. This may be done using a similar process to the one just described for case (b), but essentially rotated 90°. That is, a vertical line sweep that considers only horizontal and 45° line segments is now employed. The endpoints of 45° line segments are defined as before, with horizontal line segments being represented by one endpoint. All endpoints are sorted on y-coordinate (breaking ties so that a 45° line's top endpoint comes before a horizontal line endpoint, which comes before a 45° line's bottom endpoint). As before, the AVL tree T_D of 45° line segment candidates is maintained. When processing the endpoints, if a horizontal line segment L with endpoints (x_1,y) and (x_2,y) is encountered, where $x_1 \leq x_2$, then in order to find all intersections between L and 45° line segment candidates, we range search T_D with $p_1..p_2$, where p_i is the intersection of the 45° line through (x_i,y) with the x-axis (i=1,2).