Coursework 1 for Inf2B: Finding the Distance of Closest Pairs of Points

Issued: 9 February 2017

Submission Deadlines: The coursework consists of two parts (of a different nature) relating to one problem. As shown below these have separate deadlines, you *cannot* swap the parts round or submit either part later than the stated deadline (unless you have permission to do so from the year organiser).

- Part 1 consists of Exercise 1 on page 3, §2.1 and Exercise 2 on page 9, §2.2; these involve fairly straightforward analysis.

  Submit Part 1 by: 4.00PM, 27 February 2017.

- Part 2 consists of the tasks in §3; these are all software related.

  Submit Part 2 by: 4.00PM, 10 March 2017.

See §4 at the end of this document for instructions on how to submit.

§1. Introduction

In this practical we will consider two methods of finding the distance of a closest pair of points from a given set in the plane (i.e., we restrict to 2 dimensions). For reasons discussed below, we find the square of the distance. It would be quite easy to change the algorithms so that they return a closest pair of points as well without affecting their asymptotic runtime.

The practical has several aims:

1. Practice at asymptotic analysis (with guidance).
2. Careful implementation of one algorithm (an implementation of the other algorithm is supplied).
3. Carry out timing experiments in order to:
   (a) Compare the algorithm that you implement with one that is supplied and decide the point from which one is more efficient than the other (i.e., the overheads are overweighed by the advantages).
   (b) Determine the constant for the asymptotic analysis as it applies to your particular implementation.

You can concentrate on Part 1 (i.e., things before §3) first and then on the rest. However it would be a good idea to read the entire document through initially (without going into great detail) so that you know what is expected for the complete coursework. Note that although the document is fairly long the tasks you are required to carry out are quite modest. The emphasis here is on understanding the problem well and so plenty of discussion has been included; the modest nature of the tasks is deliberate so that you have time to follow up various aspects of the problem to whatever depth you like.

The practical requires you to submit various things. Each of these has some appropriate marks assigned. It would however be a serious error to assume that by carrying out only these tasks you have obtained the most (or indeed the main) benefit from the practical. Throughout the text you are prompted to think about various matters that are not part of the submission. You should consider these carefully along with possibly other issues as they occur to you. In other words avoid the pitfall of just going through the motions in a mechanical fashion. The practical is issued to help you develop your

understanding of the material, the marks are a by-product (of course an important one) and far from being its only rationale.

Notes

Good Scholarly Practice: Please remember the University requirement as regards all assessed work for credit. Details and advoce about this can be found at:

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

and links from there. Note that, in particular, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

Following instructions: The various parts of this practical place certain requirements with a penalty if these are not followed. *No negotiation of any kind will be entered into where the requirements are not followed.* The point of the requirements is to ensure that in doing the various parts you practice and demonstrate skills relating to the appropriate areas of the course. What you submit must be considered work; imagine that your work was being given to you and others as a sample answer. If you, or others, would find it unclear and unhelpful then your work needs further revision.

Regrettably the various requirements and penalties can appear somewhat censorious and limiting. This is not the intention, indeed much of what is stated can be seen in the much more positive light of reassurance that the various tasks have quite simple answers. Please be assured that plenty of variation is possible even when following the requirements, i.e., there isn't necessarily a unique way to do and present things clearly, concisely and correctly.

§2. The problem

We will follow normal practice and specify points in the plane by means of Cartesian coordinates, i.e., as a pair of numbers (x, y). We are given $n \geq 2$ distinct points $(x_1, y_1), \ldots, (x_n, y_n)$. Recall that the distance between two points (x_i, y_i) and (x_j, y_j) is given by

$$\delta_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

More formally this is the *Euclidean distance* but as we will use only this we will just say "distance". The problem is to find the least δ_{ij} over all the given points (considering only distinct pairs, i.e., $i \neq j$). So for example if we are given the set of points $\{(0,0), (-2,1), (3,4), (1,1)\}$ it is clear that $(0,0)$ and $(1,1)$ are the closest pair with a distance of $\sqrt{2}$. If our input included the point $(-3,2)$ then this together with $(-2,1)$ would also be a closest pair.

Note that when comparing distances we can avoid square root calculations by working with the squared distance[1], i.e., we use

$$\delta_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2.$$

This is not only more efficient but it also means that if our inputs are integers then we avoid floating point calculations altogether. It would be a simple matter to extend our arithmetic to fractions represented exactly (as a pair consisting of the numerator and denominator). This is useful because there is a limit to the reliability of floating point whereas calculations with fractions can be kept exact. We will not address this further but it is an important issue to bear in mind for writing reliable software.

[1] Here we are using the fact that distances are never negative.

Input: A set of n points $(x_1, y_1), \ldots, (x_n, y_n)$ with $n \geq 2$.
Output: The squared distance of a closest pair of points.

```
Naive-Closest-Pair
1.  d ← (x_1 − x_2)^2 + (y_1 − y_2)^2
2.  for i ← 1 to n − 1 do
3.      for j ← i + 1 to n do
4.          t ← (x_i − x_j)^2 + (y_i − y_j)^2
5.          if t < d then d ← t
6.  return d
```

Figure 1: Naïve algorithm for finding the squared distance of a closest pair of points.

§2.1 Naïve algorithm

There is an obvious algorithm for our problem which is given in Figure 1. This simply computes the distance for each pair of points and keeps the smallest distance so far. The only "sophistication" is that if the pair of points A, B has been considered then there is no need to consider the pair B, A. By the end we are guaranteed to have the least distance over all pairs. The pseudocode does not check that we are given at least 2 points. Naturally any actual implementation should do this and take appropriate action, if you want to see how it can be done in practice check the naiveClosestPair source code supplied as part of the practical (instructions are given in §3.1).

Exercise 1. Let $T_{NCP}(n)$ denote the worst case runtime of the algorithm Naive-Closest-Pair on inputs of n points. We assume that arithmetic calculations and comparisons take constant time; this will be done throughout the document.

1. Prove that $T_{NCP}(n) = O(n^2)$. For this part you *must* use asymptotic notation throughout together with any standard properties (which you may assume without comment); in particular you may *not* use named constants as part of the analysis. Note also that for full marks you must justify each expression with reference to the algorithm. [15%]

2. Is it true to say that runtime of the algorithm is the same for *all* inputs of n points? (You need only state the answer to the previous question, no justification is required.)

   Prove that $T_{NCP}(n) = \Omega(n^2)$. [10%]

3. Is it true to say that $T_{NCP}(n) = \Theta(n^2)$? Justify your answer very briefly. [5%]

Note: Your answers here and for the next exercise must be written out in full, i.e., you must state your reasoning clearly with appropriate connecting words. Remember the slogan "clear, concise and correct". If an answer is so badly expressed that it cannot be understood within a few minutes it will be deemed to be incorrect; it is important that you allow enough time to produce fluently written answers. A mere sequence of expressions without any explanation will be awarded 0; the exercises test your ability to express the simple arguments involved in an appropriate way, this is a significant part of the exercises

§2.2 The Shamos algorithm

We discuss an algorithm due to M.I. Shamos[2] that runs in time $O(n \lg n)$ which is a very big improvement on the runtime of the naïve algorithm (see the graph of n, $n \lg n$, n^2 and n^3 in Lecture Note 2). We will discuss the algorithm as originally presented in terms of the actual distance between points but remember that in your implementation you will work with the square of the distance.

Before looking at the algorithm itself, it is worth noting that many algorithms in computational geometry are quite intuitive because they appeal to our well developed visual system. However this very feature is also the cause of many incorrect proposals because special cases are easily overlooked. In the case of our problem we would normally think of the n points are being fairly randomly distributed. In general this is true but there is nothing to preclude them all lying on a line (horizontal or vertical) or in any other pattern. Any algorithm must work for all possibilities.

Shamos' algorithm uses a divide and conquer approach. It is based on the following intuitive idea: we find a vertical dividing line so that there are as many points to the left of as to the right (as equal as possible if the number of points is odd). The closest points will either lie to the left or to the right or consist of a point on the left and one on the right of the dividing line (the "cross points case"). We recurse on the left and the right which gives us the closest distances for those two disjoint sets (the base case can be taken to be any number of points but we use 3). We have to deal with the cross points case carefully: our analysis will show that we need only consider points within a narrow vertical strip centred around the vertical dividing line. This is not quite enough, in principal the strip could contain all points at least in some cases, but we will see that armed with the information gathered from the recursion we can find the closest points in the strip in linear time.

We use P to denote the set of input points. The main steps of the algorithm are:

1. If $n \leq 3$ find the closest points by the naïve method and stop.

2. Find a vertical line V such that it divides the input set into two disjoint subsets P_L and P_R of size as equal as possible (i.e., of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ respectively). Points to the left or on the line belong to P_L and points to the right or on the line belong to P_R. (Points on the line can belong to one or other of the sets, more details later.) No point belongs to both since the sets are disjoint. See Figure 2.

3. Recursively find the distance δ_L of a closest pair of points in P_L and the distance δ_R of a closest pair in P_R.

4. Let $\delta = \min(\delta_L, \delta_R)$. The distance of a pair of closest points in the input set P is either that of the points found in the recursive step (i.e., δ) or consists of the distance between a point in P_L and a point in P_R.

   (a) The only candidate points one from P_L and one from P_R must be in a vertical strip consisting of a line at distance δ to the left of the line V and a line at distance δ to the right of V.

   (b) Let Y_V be an array of the points within the strip sorted by non-decreasing y coordinate (i.e., if $i \leq j$ then $Y_V[i].y \leq Y_V[j].y$).

[2] The algorithm is discussed in the book: F.P. Preparata and M.I. Shamos *Computational Geometry: An Introduction*, Springer Verlag, 1985. The algorithm is also discussed in Shamos' Thesis, 1978, p. 163 - 171 available for free online at http://euro.ecom.cmu.edu/people/faculty/mshamos/1978ShamosThesis.pdf. Another relevant paper is http://arxiv.org/pdf/1010.5908.pdf.

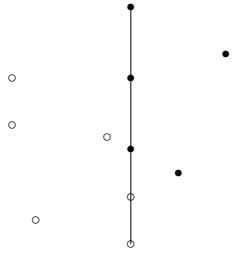

Figure 2: An example input and dividing line (circles show points in P_L and discs show points in P_R).

(c) Starting with the first point in Y_V and stepping through all except the last, check the distance of this point with the next 5 points (or any that are left if there are not as many as 5). If a pair is found with distance strictly less than δ then assign this distance to δ.

5. Return δ.

As with many divide and conquer algorithms the idea is quite straightforward with the trickiest part being how to use the solutions of the two sub-problems to obtain a solution for the overall problem in an efficient way (this is step 4 of the description above). We will discuss the details in the next sub-section but take time to understand the overall idea first, just treat step 4 as saying: "find the closest distance δ_{LR} between pairs of points one from P_L and one from P_R then return $\min(\delta, \delta_{LR})$" and assume we can do this efficiently. This approach is pretty much what is done in trying to design such an algorithm; formulate the overall strategy and address details once that is well established (in reality it might be necessary to modify the overall strategy if some details just cannot be filled in suitably).

At this stage you are encouraged to simulate the algorithm on some example inputs of your own[3] (Figure 2 shows the first stage on a small sample input). For this simulation you could use graph paper and plot the points accurately for each part of the algorithm, this is likely to be more meaningful to you than pairs of coordinates! The description of the algorithm has been stated in terms of actual distances for simplicity of phrasing; recall that in an implementation we use the square of the distance.

§2.3 Filling in the details of the Shamos algorithm

This conceptual description gives the overall plan but we have some issues to address:

1. How do we find a dividing line and assign points to P_L, P_R efficiently?
2. How do we find the array Y_V efficiently?
3. We must justify the claim that in looking at points within the strip around V (i.e., points from the array Y_V) we need only compute the distance of the current point with at most the next 5 points in Y_V.

[3] For consistency with the full algorithm do the following when assigning points on the dividing line. Assign as many as are needed to P_L with lowest y-coordinate and the rest to P_R, this is what has been done in Figure 2.

The first issue is easy to address: sort the input P by non-decreasing value of x coordinates and hold the result in an array X. However, for reasons that will become clear later, we use a slightly more sophisticated order when sorting for X. Define the order $\preceq$ on points as follows:

$$(x_1, y_1) \preceq (x_2, y_2) \text{ if and only if } x_1 < x_2 \text{ or } x_1 = x_2 \text{ and } y_1 \leq y_2. \quad (1)$$

So we compare x coordinates and break ties by looking at the y coordinate. Note that this is a total order on points, i.e., either $(x_1, y_1) \preceq (x_2, y_2)$ or $(x_2, y_2) \preceq (x_1, y_1)$ for all pairs of points [why?].

The set P_L at the top level of recursion can then be taken to be the entries $X[0..\lceil n/2 \rceil - 1]$ and the set P_R is $X[\lceil n/2 \rceil..n-1]$. Thus the top level recursive call uses parameters 0 and $\lceil n/2 \rceil - 1$ to delimit the array of values for P_L while $\lceil n/2 \rceil$ and $n-1$ are the parameters for P_R. In general we are at some recursive level where the sorted array of input values for this level is $X[r..s]$ (which we will refer to as an "array portion[4]"). If $s - r \leq 2$ (i.e., there are no more than 3 values, the "2" on the right hand side is correct [why?]) we just use brute force. Otherwise we make a recursive call with the array for P_L at *this* level being $X[r..\lceil (r+s+1)/2 \rceil - 1]$ and the array for P_R at this level being $X[\lceil (r+s+1)/2 \rceil..s]$. In other words we sort *only once* at the top level and then the data at all levels are identified by upper and lower indices into the sorted array X (we never change X itself apart from the initial sorting). You should think carefully about why we cannot afford to sort at each level of recursion. Note that if the sorted array of input values for a level of recursion is $X[r..s]$ then the vertical dividing line is given by the x-coordinate of the point $X[\lceil (r+s+1)/2 \rceil - 1]$. We will use X_L and X_R to denote the two halves of the relevant section of the array X at a given level of recursion (these contain exactly the points in P_L and P_R respectively at that level of recursion[5]).

The second issue is a little subtler, suppose first that we have an array Y of the points P being considered at the level of recursion, which is sorted by y coordinate. Then finding the array Y_V is simple. Suppose we are at a level of recursion where the input points are $X[r..s]$. Let v be the x-coordinate of the point $X[\lceil (r+s+1)/2 \rceil - 1]$; thus the dividing line at this level is given by $x = v$. This means that the shortest distance of a point (a, b) from this line is $|v - a|$ but as we are working with squared distance we use the quantity $(v-a)^2$. We can now traverse Y; for each point considered we first test if $(v-a)^2 \leq \delta^2$ and only put this into Y_V if the test is passed.

Since we know the points being considered at any level of recursion (as a portion of the array X) we could form Y by copying the points into a new array and sorting them by their y-coordinates. Unfortunately the sorting would cost too much overall. We have already seen that as far as considering points sorted by x-coordinate is concerned we just sort the input points in the global array X before initiating the recursive calls and then just focus on the relevant portion of the array X. However we cannot do this quite so simply with regards to forming the array Y, let's see this with an example. Suppose our set of points at a level of recursion is

$$P = \{ (-2,5), (-1,2), (1,6), (1,-3), (1,-10), (5,-20), (10,0), (12,13) \}.$$

The points P, P_L and P_R are thus given by the following portions of the array X:

$\cdots$	$(-2,5)$	$(-1,2)$	$(1,-10)$	$(1,-3)$	$(1,6)$	$(5,-20)$	$(10,0)$	$(12,13)$	$\cdots$
	P_L				P_R				

[4] Note that an array portion is just an array but with the first element starting at an index that might be different from the standard one (0 in the case of Java).

[5] The difference is that conceptually P_L, P_R are sets while X_L, X_R are arrays consisting of the same points sorted according to the order $\preceq$. Of course in an implementation we might use an array based data structure to hold the members of the set and have them sorted but this is imposing extra structure that cannot be assumed if all we say is that we have a set. Indeed in our enhanced representation we might hold the set/array together with indices r, s that identify the portion of it of interest as one compound data structure (record or class according to implementation language).

Input: A set P of points (split into P_L and P_R) and an array Y of the same input points sorted in increasing order by y-coordinate.
Output: The arrays Y_L, Y_R as described in the text.

splitY
1. new arrays Y_L, Y_R of lengths $P_L.length$ and $P_R.length$ respectively.
2. $l \leftarrow 0$
3. $r \leftarrow 0$
4. **for** $i \leftarrow 0$ **to** $Y.length - 1$ **do**
5. **if** $Y[i] \in P_L$ **then**
6. $Y_L[l] \leftarrow Y[i]$
7. $l \leftarrow l + 1$
8. **else**
9. $Y_R[r] \leftarrow Y[i]$
10. $r \leftarrow r + 1$

Figure 3: Splitting the set Y.

So the next level of recursion would be called with P_L and P_R respectively.

Now consider the array Y of the points contained in P, which is sorted by y coordinate only; the array Y is passed by the call that initiated the current level of recursion. This is:

0	1	2	3	4	5	6	7
$(5,-20)$	$(1,-10)$	$(1,-3)$	$(10,0)$	$(-1,2)$	$(-2,5)$	$(1,6)$	$(12,13)$

In the two recursive calls we want to pass arrays Y_L, Y_R sorted by y-coordinate and consisting of those points of Y that belong to P_L and P_R respectively. Here is the array Y again but below each entry we have indicated to which of the two arrays Y_L or Y_R it *should* belong (after their creation).

0	1	2	3	4	5	6	7
$(5,-20)$	$(1,-10)$	$(1,-3)$	$(10,0)$	$(-1,2)$	$(-2,5)$	$(1,6)$	$(12,13)$
Y_R	Y_L	Y_L	Y_R	Y_L	Y_L	Y_R	Y_R

So the difference is now clear. In order to obtain the arrays for P_L, P_R sorted by x-coordinate (array portions X_L, X_R) we just divide the current portion of the array X into two parts that are as equal as possible. However this will *not* work as regards forming the arrays Y_L and Y_R. The reason is simple: we are using a vertical line to subdivide and the y-coordinates of points in P_L, P_R can be arbitrarily high or low in either set. If we used a horizontal line then the situation just swaps over. To sum up: it is a simple matter to create the array portions X_L, X_R given the section of the array X at some level of the recursion. In order to create the arrays Y_L, Y_R we need to do a little work using the algorithm shown in Figure 3.

There are two points to discuss in connection with the algorithm of Figure 3. First, note that the array Y is given to us from the previous level of the recursive call so it is either the array Y_R or Y_L of the previous level according to the branch of the recursion. The critical point is that it is provided in sorted order (we ensure this is so at the very start by sorting an initial array Y consisting of all the input points and passing it to the method that implements the recursion). Since we examine its points in order and put them in the newly constructed arrays in order it follows that these arrays are also sorted. To sum up: we need only sort the starting array Y before initiating the recursion and thereafter all our arrays formed by splitY stay sorted.

The second, more subtle, point concerns the membership test $Y[i] \in P_L$ of line 5. Since P_L is the sorted array portion X_L we could use binary search, but even though this is very efficient we can do much better[6]. Let n denote the size of P (at the level of the call), this is also the size of Y of course. In order to meet our performance criterion we must ensure that splitY runs in time $O(n)$. Well the loop on line 4 is executed $\Theta(n)$ times so its body has to run in constant time. Everything in the loop clearly meets this requirement except for the membership test. How do we ensure this test also runs in constant time? Let (x_L, y_L) be the last point of the array portion X_L and let (x, y) be the query point (i.e., the point in $Y[i]$ of the membership test). Now if $x < x_L$ then necessarily (x, y) occurs in X_L hence in P_L. Similarly if $x > x_L$ then (x, y) is not in P_L, i.e., it is in P_R. But what happens if $x = x_L$? Recall that we sorted X first by x coordinate and, if necessary, then by y coordinate. It now follows that the case when $x = x_L$ can be resolved by comparing the y coordinates: if $y \leq y_L$ then (x, y) occurs in P_L otherwise in P_R. In the example given above we have three points with equal x coordinates: $(1,-10)$, $(1,-3)$ and $(1,6)$ and $(x_L, y_L) = (1,-3)$. Go through the procedure described in this paragraph to see how the split of Y into Y_L and Y_R works. It follows that the membership test can also be carried out in constant time (at most two comparisons). Finally, note that the algorithm splitY can now be modified to take as parameters the set Y and the deciding point (x_L, y_L), there is no need to pass P as a parameter. The arrays Y_L, Y_R can also be created before the call and passed as parameters.

For the final issue in our list of three on page 5, it is clear that a point in P_L and a point in P_R cannot be of distance less than δ unless they are both in the strip of width 2δ centred around the dividing line. Furthermore if there are two such points then we will find them by comparing points with lower y coordinate to ones with higher (or equal) y coordinate. What remains is to prove that given a first point (x_1, y_1) then we need only consider at most the next 5 points, i.e., any others would be too far away. If we have a point (x_2, y_2) with $y_2 > y_1 + \delta$ then it is clearly too far away (the squared distance is strictly greater than $(y_1 + \delta - y_1)^2 = \delta^2$). Thus the only possible candidates are within the box of width 2δ and height δ that has the point (x_1, y_1) on its lower edge and is centred around the dividing line. This box is made up of 8 squares of size $\delta/2 \times \delta/2$. Now the difference between the x coordinates of any two points within such a square is at most $\delta/2$ and likewise for the y coordinates. Thus the squared distance is at most $(\delta/2)^2 + (\delta/2)^2 = \delta^2/2 < \delta^2$. Since we already know that points from P_L are at distance at least δ (and likewise for points in P_R) it follows that each box has at most one point to be considered, unless a box has a point in P_L and a point in P_R. This exceptional case can only happen if the two points are on the dividing line. So suppose a square S_L shares a common edge with a square S_R on the dividing line. Moreover on this common edge there is a point (x_L, y_L) from P_L and a point (x_R, y_R) from P_R. The preceding discussion shows that S_L cannot contain any other points from P_L (we include the edges of the square here) and similarly for S_R. Assigning (x_L, y_L) to S_L and (x_R, y_R) to S_R, we see that whatever is the case each of the boxes has at most one point to be considered, however there can only be at most two points on the dividing line. This gives us a maximum of 6 points[7]. Since (x_1, y_1), the point with which we start, is one of the points within the boxes we are left with at most 5 more to compare with (x_1, y_1).

An alternative way to see the previous claim is the following. If we pick any point P_L in the left hand box (width and height δ) and draw a disk with centre P_L and radius δ then the box cannot contain any other point that is in the interior of the disc (it would have distance strictly less than δ from P_L). Now if we have any three points within this box which are at distance at least δ from each other then

[6] A question worth asking yourself is: would we achieve the required asymptotic runtime if we used binary search?
[7] You should think carefully about the fact that we *do* need to ensure that points considered are within distance δ of the vertical line, i.e., we cannot omit the test $(v - a)^2 \leq \delta^2$.

their three discs will cover the entire left hand box. The same argument applies to the right hand box, giving us a total of 6 points at most. If you have time you could try to prove the claim about the three discs, it is intuitively plausible but intuition can be faulty.

Exercise 2.

1. We want the sorting of the two arrays X, Y (carried out before intitiating recursion) to be done in $O(n \lg\, n)$ time in the worst case. Suggest one standard algorithm that could be used. [5%]

2. Let $T(n)$ be the runtime of the recursive part of the algorithm. Prove that

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 3; \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n), & \text{if } n > 3. \end{cases}$$

   You may assume without further proof that the runtime of splitY is $O(n)$.

   Deduce that $T(n) = O(n \lg\, n)$ by using an appropriate result from the notes. You should name the result but need not state it in full, however you *must* show how you used it to deduce the $O(\cdot)$ expression. [15%]

3. Use an appropriate property of the big-O notation to deduce that the overall running time of the algorithm is $O(n \lg\, n)$. [5%]

You should think about what would go wrong if the base case of the algorithm tested $n \leq 2$ instead of $n \leq 3$ and how you could fix things while still keeping $n \leq 2$ as the base case. However do not include any discussion of this in your answer.

§3. Software tasks

We first give a guide to some of the supplied classes and methods (some more are mentioned later). If, after studying the software very carefully, you have any queries regarding it then contact the Teaching Assistant for this thread using NB to which you should receive an invitation via your university e-mail around the time when the coursework is released.

§3.1 Supplied software

The files that you will need for this coursework can be found on-line at the coursework webpage:

`http://www.inf.ed.ac.uk/teaching/courses/inf2b/coursework/cwk1.html`

This page contains a file called `inf2bcw1.tar` which you should download. The file is tarred so you need to extract it, e.g. by `tar -xf inf2bcw1.tar`, this will create a subdirectory `src` of your current one that contains the software. The `java` package for this coursework is `package closestPair`. The only class that should be changed is the nearly empty class you will find in the file `StudentCode.java` which is where you will implement your part of the project. In fact the amount of code you need to write is fairly modest.

Prachya Boonkwan and Naums Mogers have kindly written a `README` file giving you some tips about what to do for such things as writing test scripts. Just as importantly there are warnings about what not to do with the code. Make sure you follow their advice. The file is included in the package that you download.

We represent a point in the plane by an object of the class `Point`. This does the obvious thing; it holds the x and y coordinates as numbers so that if `P` is an object of type `Point` then `P.x` and `P.y`

give the coordinates. *In this implementation we insist that the coordinates are integers.* This choice makes sense because under it we never need to use floating point artihmetic (remember that we work with the squared distance) and so we can test two numbers for equality with complete reliability.

The set of input points is represented by an object of the class `PointSet`. Note that as this is a set there can be no repetition of a point and you cannot assume any ordering on the elements. The following methods are supplied:

- `public static PointSet getPoints(String f).`
  This opens a file called `f` and reads points from it. Points are presented in the format

$$\begin{matrix} x_1 & y_1 \\ x_2 & y_2 \\ & \vdots \\ x_n & y_n \end{matrix}$$

  Each x and y value is an integer. Each point is on a line by itself and the end of file indicates the end of input. If any unexpected characters are met then an exception `UnreadablePointSetException` is thrown.

- `public Point nextPoint(PointSet P).`
  This is used to iterate over the points in `P`. That is, the first call returns one point and each subsequent call returns a diferent point until all are returned. Once all points have been returned the next call returns `null`. Subsequent calls to the method will iterate over the set afresh. Note that there is no guarantee that different iterations will return the points in the same order.

The algorithm Naive-Closest-Pair is implemented by the method

- `public static Point naiveClosestPair(PointSet P).`

If there are less than two points in `P` then an exception `TrivialClosestPairException` is thrown.

The next method is used to obtain test samples of points.

- `public static PointSet generatePoints(int n).`
  This returns a set of `n` points, whose coordinates are integers. If $n \leq 0$ then the empty set is returned. The points are generated at random but it is ensured that they are indeed separate.

You are also supplied with an efficient sorting method, i.e., one that works in $O(n \lg\, n)$ time in the worst case.

- `public Point[] sort(char c).`
  This returns the points in the point set in an array sorted in non-decreasing order by coordinate as indicated by the parameter `c`. This parameter takes either the value `'x'` or the value `'y'`, an exception `UnknownSortOptionException` is raised otherwise. If the parameter is `'x'` then the array is sorted in $\preceq$ order as defined in (1) on page 6 (first by x, and then breaks ties by y).

§3.2 Tasks

Note: You are asked to write some code that is in fact very simple thanks to the supplied methods. *Any code that is incorrect will be awarded 0.* Correct code will be marked for style as well. This means that it should have useful comments and helpful indentation. Note that excessive or pointless comments (e.g., stating the obvious such as "now we add the two variables together") or excessive indentation are almost as bad as their complete absence and will incur a penalty.

The method `generatePoints` will supply most sets of points with which you will be working. Of course you can also create sets of points for your own testing if you like.

Your programing tasks are as follows.

1. In the `StudentCode` class, provide implementations of the methods

   - `public static int closestPair(pointSet P).` [10%]
     This does the preparatory work for the recursive part of the algorithm and calls the method `closestPairAux` for the recursive part.
   - `public static int closestPairAux(Point [] X, Point [] Y).` [15%]
     This carries out the recursive part of the algorithm; that is, the bulk of the work.
   - `public static void splitY(Point testPoint, Point [] Y, Point [] YL, Point [] YR)` [5%]
     This implements the algorithm of Figure 3. So the supplied `Point` array `Y` is split and the two resulting `Point` arrays are returned in `YL`, `YR` respectively. The parameter `testPoint` is the deciding point (x_L, y_L).

   Keep your code straightforward, follow the algorithm as outlined in §2.2.

   Note that your code must work on any number of points. If less than two points are supplied then the exception `TrivialClosestPairException` must be raised.

   Test your implementation by using the following method in the `ClosestPair` class:

   - `public static void closestPairCheck(int t, int n)`
     Generates `t` sets of points of size `n` and obtains the squared distance of a claimed closest pair using your implementation of the method `closestPair` for each set. If all results are correct then a message reporting this is returned, else failure is reported. No further information is given. Note that `t` must be strictly positive (an exception `InvalidNumberOfTestsException` is raised otherwise). The point set generated is saved in the file `points.txt` for debugging purposes.

   Naturally the test procedure does not guarantee correctness. All we can say is that if the code fails the tests then there is something wrong. If it passes all tests then is is probably correct.

2. The aim of this part is to compare the runtimes for `naiveClosestPair` with those for `closestPair`. We would expect that as the number of points gets larger the differences would be more significant. Note however that there could be small sets of points for which `naiveClosestPair` runs faster (you should think about why this can be the case).

   The supplied `ClosestPairToolkit` class method

   - `public static void getRuntimes(int p, int t, String f)`

   does the following:

   (a) Generates `p` sets of points for various sizes starting from 10 (the sets are *not* random so that experiments are repeatable).

   (b) Finds a closest pair of points using `naiveClosestPair` and `closestPair`, and takes the cpu times for these.

   (c) Records the worst case times for each algorithm implementation taken over the `p` sets (note that it is pefectly possible that the worst case runtimes for the two algorithm implementations occur for different sets).

   (d) Repeats the above with sets of size $20, 30, \ldots, 10t$.

   (e) Outputs the result for each iteration on the file named `f` in the format

   input-set-size `naiveClosestPair-`*worst-case-runtime*
   `ClosestPair-`*worst-case-runtime*

   For `f` you can give a path name but the simplest thing is to run code from your working directory and use just the name for `f`; the file will be placed in your working directory.

   (Note that the data above are given on a single line of the file, the text is too long to fit on one line of this document.) Carry out experiments with increasing values of `t` until you find a clear difference between the two algorithms. In order to avoid excessive waiting choose the fairly moderate value for `p`. Naturally it makes sense to continue the experiment a little beyond the point of the first clear difference (indeed due to various factors there might be a temporary turn around, although this is unlikely for this experiment).

   Finally, run the experiment with `p=10`, `t=250` and keep the results in a file with the name `closestPairTimes`. Study the output to see what happens in terms of one algorithm outperforming the other. [10%]

   Use this file to find the crossover point at which `closestPair` is consistently faster than `naiveClosestPair`. This is a rough and ready way of obtaining an estimate for the settling in period before the overheads outweigh the benefits. This approach is open to criticism in general (think about why this is the case) but is not misleading in this case.

   From now on we will focus on `closestPair`.

3. We know that the worst case runtime $T_{CP}(n)$ of `closestPair` satisfies $T_{CP}(n) \leq cn \lg n$ for some constant c. Of course asymptotic notation allows for a settling in period, i.e., from some value n_0 of n onwards. In fact this period is not really necessary at least from $n = 2$ onwards because we can just take a larger constant if needed (with the obvious downside that this gives a pessimistic performance guarantee for all large enough values of n). It should be fairly clear from the algorithm that the settling in period is not long; essentially we just need n to be big enough so that $cn \lg n$ dominates the cost of initializing variables and setting up data structures.

   The supplied `ClosestPair` method

   - `public static double[] getRatios(int p, int t, int threshold, String f)`

   is similar to `getRuntimes` except that for each experiment it only uses `closestPair`. Once it has found the worst case runtime for an input of size n it records the value of that runtime divided by $n \lg n$. The results are output to the file `f` in the format

integer-size *ratio*

one per line as before. At the end of this output it also appends three extra lines:

```
Sorted ratios are: r1,r2,...
Truncated maximum ratio is: max-ratio
Truncated average ratio is: ave-ratio
Ratio index theshold is: threshold
```

Thus the final three lines give us the maximum and the average of all ratios except those with the index greater than the threshold index in the sorted ratios array. The threshold index is specified in the *threshold* parameter. The case for taking the average is that it smooths out to some extent the effects of any garbage collection. In a more detailed study we need to be more sophisticated but here we will keep things simple and rely on the plot (see below) to give us an idea of how useful the average is. You can also use the sorted ratios to get an idea of the effects of garbage collection (we would expect ratios affected by this to be significantly bigger than the "real" ones). The function returns the maximum and average ratios so they can be passed easily to `ClosestPairToolkit.plotRuntimes()`.

Use the following values for the parameters:

- `p` set to 10,
- `t` set to 250.

Finally you will plot the data from the file `closestPairTimes` of the previous part and the worst case runtime as determined by the theoretical analysis together with the two estimates for the constant found by the preceding experiment (the maximum and average). Use the supplied `ClosestPair` method

- `public static void plotRuntimes(float c, float a, file f)`

to produce your plot. For `c` use the maximum constant found above, for `a` use the average and for the file `f` pass the data in the file `closestPairTimes` from the preceding part. This will bring up a plot which you should save in a file called `plot`. To be precise this plot will show graphs of the recorded times for `naiveClosestPair`, `closestPair` as well as the curves $cn \lg n$ and $an \lg n$. [5%]

Compiling and Running

From the command line, type `cd` *path-to-src-folder*. Use `javac closestPair/*.java` to compile. Use the command `java closestPair/`*`ClassName`* to run the program (where *ClassName* is the `.java` file where your `main()` target is).

As stated above your code must be well laid out showing its logical structure. Just like good prose or good mathematical writing it must be set out to aid understanding. This way you and others can maintain it as well as spot any errors more easily.

§4. Submitting the Coursework

You must submit the two parts of your work by the deadlines given at the head of this document. If you complete the coursework in full, you should be submitting:

Part A: Analysis of algorithms (both of them in *hardcopy only*).

1. Your answer to Exercise 1 of §2.1.
2. Your answer to Exercise 2 of §2.2.

Part B: Software, for this the submissions are *electronic only*.

1. Your code in the class file `StudentClass.java`.
2. `closestPairTimes`.
3. `plot.jpg`.

Note: Your answers to the two exercises of Part 1 are best handwritten (unless you have a medical condition that prevents this) and neatly presented following the guidelines of Lecture Note 2 (and the notes in general). Your hardcopy submission should be made at the appropriate time to:

- ITO, Room FH-1.B15, Forrest Hill, 5 Forrest Hill, Edinburgh, EH1 2QL.

Important: Put your *matriculation number* very clearly at the top of your submission. You do not have to put your name; marks will be returned to the ITO by matriculation number (this is how the `submit` command (see below) organises things. If you need to submit more than one piece of paper for a part please staple the sheets together and put your matriculation number on the top of each sheet (in case they get separated). Do not use folders to hold several sheets as this holds up the marking process significantly.

For electronic submission follow these instructions.

- First put your submission files (and *nothing else*) into one directory called `inf2b-cw1`.
- Submit this directory using the following command (when logged into DICE):

```
submit inf2b 1 inf2b-cw1
```

Warning: The rule for courseworks is "We mark what is submitted." Before you submit your coursework, make sure that you are submitting the correct files. In some previous years students submitted the wrong files and lost marks that way.

Kyriakos Kalorkoti

Software by Prachya Boonkwan and Naums Mogers

Tuesday 14th February, 2017