7 A brute-force algorithm for the closest-pair problem

The closest pair problem is defined as follows: Given a set of points, determine the two points that are closest to each other in terms of distance. Furthermore, if there are more than one pair of points with the closest distance, all such pairs should be identified. So the input is a point set with size n and the output is a point pair set. The problem has applications in, for example, traffic control systems. A system for controlling air (3-D) and sea (2-D) traffic might need to know which are the two closest vehicles in order to detect potential collisions.

Given the coordinates of two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. The distance between the points $|\overline{p_1 p_2}|$ is given by

$$|\overline{p_1 p_2}| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

A brute-force closest pair algorithm simply looks at all the $\binom{n}{2} = \frac{1}{2}n(n-1)$ point pairs and picks the ones with the smallest distance. The corresponding function in the point set class is `PointPairSet closestPointBF (void)`. Note that this algorithm works for both 2-D and 3-D cases, although you are only required to consider the 2-D case in this project. Obviously, the time complexity of this algorithm is $O(n^2)$. Although this is polynomial, when you test the algorithm on a large point set, you will see it takes quite a while before the algorithm finds the closest pairs.

8 Sorting a point set by a certain coordinate

In the divide-and-conquer algorithm for the closest pair problem, you will first be asked to sort the point set by a certain coordinate (e.g., x or y). We have learned in class that Quick Sort is an easy-to-implement sorting algorithm with an $O(n \log n)$-time average-case time complexity. Although its worst-case time is $O(n^2)$, Quick Sort performs extremely well in practice and it is often favored over Heap Sort and Merge Sort.

You are asked to implement the protected function `void qsort (int type = SORT_X)`. The algorithm will call the recursive protected function `static void quickSort(PointSet&, points, int type, int left, int right)`, which sorts a portion of a set defined by indices `left` and `right`. The constants `SORT_X` and `SORT_Y` are used to indicate whether the point set will be sorted by the x-coordinates or the y-coordinates. An additional requirement for `qsort`: To sort a point set by x-coordinates, if the point set contains points with the same x, they should be ordered by increasing y. For example, $(0,3),(2,1),(2,4),(2,5),(4,3),(6,3)$. The similar requirement applies to point set to be sorted by y-coordinates. For example, $(2,1),(0,3),(4,3),(6,3),(2,4),(2,5)$.

9 A divide-and-conquer algorithm for 2-D closest pair problem

Let Q refer to the set of n 2-D points. Each recursive invocation of the algorithm takes as input a subset $P \subseteq Q$ and arrays X and Y, each of which contains all the points of the input subset P. The points in array X are sorted so that their x-coordinates are monotonically increasing. Similarly, array Y is sorted by monotonically increasing y-coordinates. Note that in order to attain the $O(n \log n)$ time bound, we cannot afford to sort in each recursive call. So sorting has to be done only twice in a preprocessing stage before the divide-and-conquer algorithm starts.

A given recursive invocation with inputs P, X, and Y first checks whether $|P| = 2$. If so, the invocation solves the problem directly. Further, when $|P| = 3$, divide-and-conquer does not work since you will end up with a point set with just one point. Therefore, the case of $|P| = 3$ should also be solved directly without recursion. You may call `closestPointBF`. If $|P| > 3$, the recursive invocation carries out the divide-and-conquer paradigm as follows:

- Divide: The algorithm first finds (implicitly) a vertical line l that bisects the point set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, all points in P_L are on or to the left of line l, and all points in P_R are on or to the right of l. The array X is then divided into X_L and X_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing x-coordinates. Similarly, the array Y is divided into arrays Y_L and Y_R, which contain points of P_L and P_R respectively, sorted by monotonically increasing y-coordinates. Note that no sorting is required to obtain X_L, X_R, Y_L, and Y_R. How? Make sure that only $O(|P|)$ is spent at this stage.

- Conquer: Having divided P into P_L and P_R, the algorithm makes two recursive calls, one to find the closest pairs of points in P_L and the other to find the closest pairs of points in P_R. The input to the first call are subset P_L and arrays X_L and Y_L; the second call receives the input P_R, X_R and Y_R. Let the closest-pair distances returned for P_L and P_R be δ_L and δ_R, respectively, and let $\delta = \min\{\delta_L, \delta_R\}$. Clearly, the time spent at this stage is $2T(|P|/2)$, assuming that $T(|P|)$ is the time spent by the algorithm to find the closest pair in point set P.

- Combine/Merge: The closest pairs for P are either the pairs with distance δ found by the recursive calls, or pairs of points with one point in P_L and the other in P_R. The algorithm then determines in $O(|P|)$ if there are such pairs whose distance is less than δ. How? Pay close attention to the related lectures and read MAW.

10 Test

There are two purposes for testing your implementations thoroughly. One is to make sure that your two member functions (`closestPointBF()` and `closestPointDC()`) work correctly even for special cases such as that

- there are points with the same x-coordinates in the input; and
- there are points with the same y-coordinates in the input.

You may come up with other special cases that might happen in practice. The second purpose for testing is to make sure that you see the advantage of the $O(n \log n)$-time divide-and-conquer algorithm over the $O(n^2)$-time brute-force algorithm for point sets with large sizes. When the point set sizes are in hundreds, the straightforward brute-force algorithm may be faster. However, when the point set sizes increase to thousands or more, the divide-and-conquer algorithm outperforms the brute-force algorithm with a significant margin.

To complete this part of the project, you need to implement the functions `qsort`, `closestPointBF`, `closest-PointDC`, `quickSort`, and `closestPoints`, and conduct a thorough test of your implementation. To receive credit for this part of the project, submit your `Point.cpp, PointPair.cpp, PointPairSet.cpp, PointSet.cpp` by the midnight of the due date.

The end.