This project has several goals:

- Ensure that you understand the divide-and-conquer closest-pair algorithm presented in class.
- Get a hands-on understanding of the practical benefits of designing more sophisticated algorithms, in particular, the divide-and-conquer design technique.
- Design your own algorithm to solve the problem and compare its performance to the divide-and-conquer algorithms.

We have provided Java code that implement everything needed except for the algorithms themselves including the pre-processing for the divide-and-conquer algorithm.

**Part One: Implement the Brute Force Algorithm (10 points)**
You are to implement the brute force algorithm that computes the distance between each pair of points (just once) and then returns a closest pair of points. You are welcome to modify the `PointPair` class implementation to provide any further functionality that you would like.

**Part Two: Implement the Dividing Portion (25 points)**
You are to implementing the portion of the divide-and-conquer algorithm prior to when the recursive calls are made. This must run in linear time. To check that your implementation is correct, for \( n = 10 \) you should check that the four arrays of points that will be sent to the recursive call (the left and right halves sorted by both x- and y-coordinates) are correct.

**Very Important**: The `pointsByX` array is sorted so that `pointsByX[i].leftOf(pointsByX[i+1])` even if these two points have the same x-coordinate. If you are not using `leftOf` to do this part, then come talk to me or one of the TAs and discuss your implementation since there is most likely a problem when points share the same x-coordinate.

**Part Three: Implement the Divide-and-Conquer Algorithm (35 points)**
You are to complete the implementation of the divide-and-conquer algorithm. You should compare the results with that of the brute force algorithm. Note that there is a unique distance between the closest pair of points but there could be more than one set of pairs with that distance. Determine roughly how many points you can include in your input before the divide-and-conquer algorithm runs for more than a minute. You can increase the memory available to the Java runtime system using the command line arguments `-Xms150m -Xmx150m`. This should work up until about 2,500,000 points. If you want you can use a larger value (such as 200m, 300m, ...) up until the memory you have on your computer. If you are using Eclipse, under the Run menu pick “run...” and then put this under the VM arguments under the arguments tab.

**Part Four: Compare the Algorithms’ Performances (10 points)**
Compare the brute force and divide-and-conquer closest-pair algorithms on inputs of different sizes. Include at least 10 different choices for \( n \) (the number of points) between \( n = 100 \) and a value for \( n \) for which the brute force algorithm takes around 30 seconds. You should run your experiments on a relatively unloaded machine to get consistent results, since the Java timer class measures wall-clock rather than CPU time. Be sure to turn off any printing from within your algorithms before doing timings, as printing is expensive. Create a single graph that plots the running time of both algorithms as a function of the number of points.

**Part Five: Try Something New (20 points)**
Here is your chance to implement an algorithm of your choice and see how it compares to the divide-and-conquer algorithm. Pick something you think might run fast in practice and try it out. If you want to modify the method used to generate the random points you may, but also include test results using the provided methods for generating the points. As in part four, create a single graph that plots the running time of your new algorithm and the divide-and-conquer algorithm as a function of the number of points.