The approach I chose for this problem is to choose two points opposite to each other (on the same diagonal of the rectangle) and check to see if the other two points corresponding to the other diagonal, exist in the input list of points.
I chose to write the code in Java: I created a class representing a point object and the Main class in which the operations required are done. The class Main contains the main method and two other methods, one for processing the input file and another for counting the rectangles.

- The input is given as a text file, with the format given in the requirements of the problem: (x1, y1), (x2, y2);
- To process the input file, I used a Buffer Reader and read each line one by one from the input file, removed the spaces, parentheses, and commas, parsed them to Float numbers and stored them into an Array List.
- The input file may contain multiple lines, as long as the format is like the required one.
- In the method "main", I call the method for reading the input and check if the processing was done successfully.
- The method that counts the rectangles works as described above. Since I need to check if a point exists in the input list, I created a hash map containing all the points, for a faster access time.
- The second point taken will always be the position after the first one, since I order the list of points and the "for" clause for the second point always starts at the next position.
- In order to take the diagonal points I check if the points are not on the same axis (Ox or Oy) by comparing their x and y coordinates.
- Compute the other two points and check if they exist in the input list, if so, increase the number of rectangles.
- The point class implements Comparable method and overrides the compareTo method, along with equal and hashCode method in order to use the "contains" method, the hashsSet and the sorting of the list.

- I assumed that squares are also counted as rectangles and that the input is not always ordered ascendingly (even though the complexity increases because of the sorting).

- The solution with HashSet (since it has a faster access time) reduces the overall complexity of the problem, even though it adds additional space.