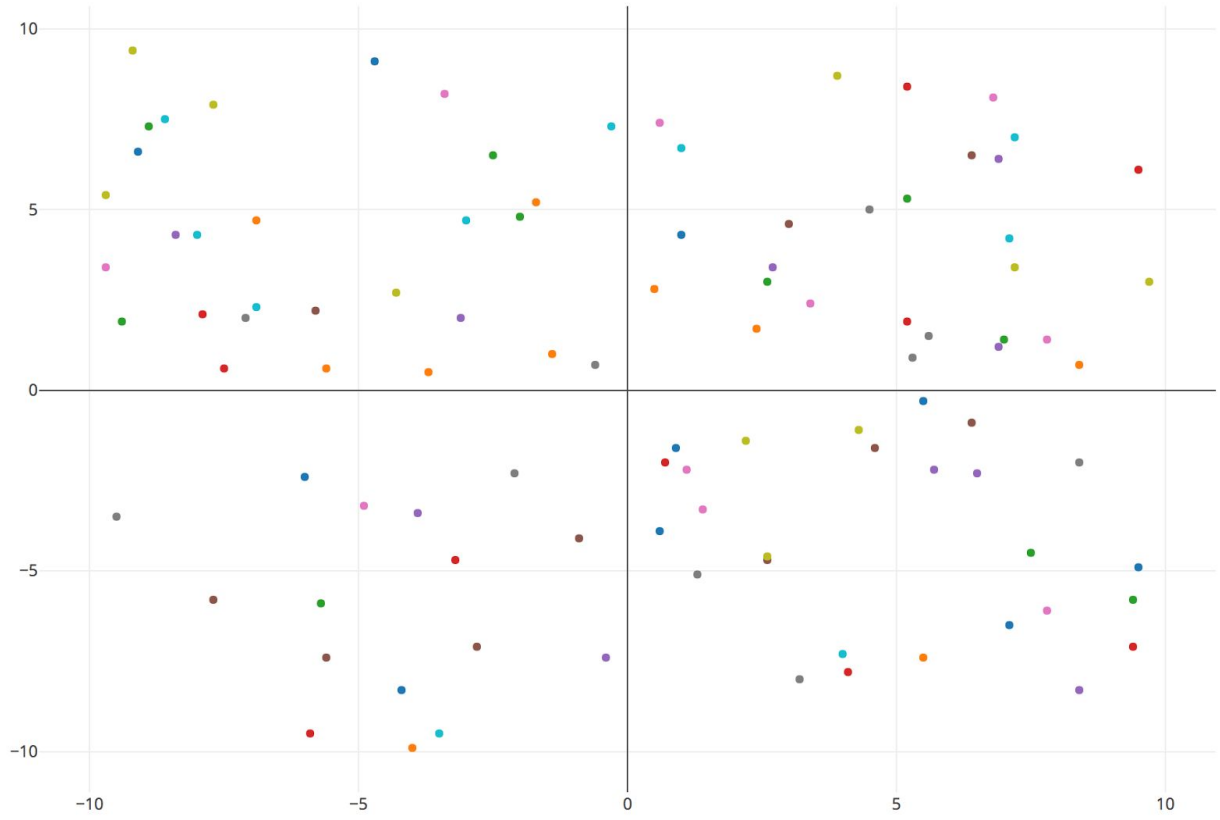
CSE 3353

Assigned: Monday Feb. 12, 2017
Due: Monday Feb 26, 2017 at 11:59pm

Project 1 - Protect the Planes!

In CSE 2341, you were tasked with determining if it was possible to fly between two cities given a data representation of city connections. A hip new company called Algo Airlines has been using your product with massive success, but they have come to you with another problem: they have had to add so many new planes that they need to be sure they won't crash into each other during flight!

Algo Airlines wants to know how close their planes are flying to one another. In particular, they want to know which pair of planes are flying the closest to one another so that they can alert the planes via air traffic control to avoid one another. This is known as the **Closest Pair Problem**. Consider the following graph of points, where each point is a plane in the air:



Note that we are only concerned with a 2D plane of x-y coordinates at the moment (assume that all of Algo Airline's planes fly on the same horizontal… plane). In data form, we can describe each plane as having the following structure:

```
[
    {
        flightNumber: "BC1234",
        x: 2.0,
        y: 3.2
    },
    {
        flightNumber: "XY3456",
        x: 2.3,
        y: 4.6
    },
    ... and so on
];
```

As you can see on the graph, some planes are far apart while others are very close together. The goal of this project is to come up with several algorithms of increasing efficiency to solve the closest pair problem.

You can assume that the data will be well structured, because you will be generating the data yourself as seen below.

You are tasked with the following:

1. Learn more about node.js and 3rd party libraries by completing the following:
    a. Create a folder for this project. In terminal, navigate to this folder and run the command `npm init`. npm stands for Node Package Manager, and is the tool used to set up and run projects, as well as install third party libraries. You will be guided through the process of creating what's called a "package.json" file. At the very least, provide a package name and description. For all other prompts simply press Enter to let npm provide defaults.
    b. Next, run the command `npm install chance`. You'll notice that your directory now has a new directory called node_modules. This is where third party libraries get installed (if you list the contents of node_modules you will see a folder for chance). Observe your package.json file: chance is now listed as a **dependency**.
    c. Review the documentation for chance at the following site: http://chancejs.com. You'll note that it has lots of utility functions for creating random data. Note that it's not truly random, but for our purposes it's random enough.
    d. Write a program that will generate a dataset of 100 random airline points with the structure given above (an array of objects). The x and y coordinates should be floating point numbers bounded by [-10, 10], with 4 decimal points of precision. The flightNumber should be two characters followed by four numbers. The results should be output to a file called `airline_map.json`

2. Design and implement a brute force algorithm to solve the Closest Pairs problem. The program should load your `airline_map.json` file as input. Your program should print the two points that are closest to one another (x, y, and flightNumbers), and how far apart they are.
    a. Describe in plain english how your algorithm works. This can be in a separate text file or as comments in the program.
    b. What is the big-O runtime complexity of your algorithm? Explain by doing a T(n) analysis of your code.

3. Duplicate your program from part 2. Channel your knowledge from Data Structures to implement a sorting algorithm that runs in O(nlgn) time. **Do not use the built in JS Array.sort function**. Sort the airline map in order of increasing y coordinates. Now that the dataset is sorted, modify your brute-force algorithm to leverage that fact and minimize the number of calculations.
    a. What changes were made to your brute-force algorithm?
    b. What is the big-O runtime complexity of this algorithm, taking into account the sorting algorithm as a part of the total runtime? Did it change from part 2?
    c. Use the bash command `time` to do some real runtime analysis of your programs. Which program runs faster on the same dataset, your program from Part 2 or Part 3? Why do you think that is? (Note: you may need to re-run your program from Part 1 and generate many more than 100 data-points to notice any

real difference). In your analysis, include the size of the dataset and how long your program took to process that dataset. Provide multiple examples.

4. In a new file, describe and implement a recursive solution for the Closest Pairs problem. The core idea is similar to the maximum subarray problem: if we divide the dataset in half, we know that the closest pairs will either be on the left (or top) side, right (or bottom) side, or crossing the boundary.
    a. Tip: Your base case will not necessarily be a dataset of size n = 1, since you cannot determine distance between a pair of points… with only 1 point.
    b. Tip 2: Continue sorting the data as your first step before running your recursive algorithm.
    c. Describe in plain English how this algorithm works.
    d. What is the big-O runtime complexity of this algorithm? Do a T(n) analysis of your algorithm to determine this answer.
    e. Using `time` again, how does the actual runtime of this algorithm compare to the algorithms from part 2 and 3? Again, provide multiple examples on datasets of different sizes. Is it exhibiting the expected behavior (according to your big-O analysis)?

You should feed the same dataset for parts 2, 3, and 4 when running comparisons. As expected, all three parts should give the exact same pairs of points as the closest pair. If they give different results, then it is your job to find the source of error and correct it.

Heading and Collision Risk

Suppose each plane also has a property called “heading” that describes the cardinal direction that the plane is traveling. It is a floating point number in the range [0, 360] degrees, where 0 degrees is north, 90 is east, 180 is south, and 270 is west. Expand your data generation function to include a randomly generated heading property to each data point.

Once you run your closest pair algorithms above, add an additional step to answer the following question about the two planes: based on the their position *and* heading, are they actually at risk of crashing? By that I mean that if you drew lines starting at their position to the direction of their heading, will they intersect? Your output should indicate yes or no (include the point data to verify correctness), and in your report you should outline the algorithm you made to answer that question. Provide T(n) and big-O analysis here as well. Visual examples here are expected.

What to Submit:

Upload a single zip file onto Canvas containing all code, documents, and the package.json file. Documents describing T(n), big-O, etc. can be typed out or hand-written and scanned neatly as PDFs. **Do not include the node_modules folder in your submission**. Since your dependencies are listed in your package.json file, I will be able to run `npm install` and

download the dependencies on my local machine. This is the industry standard approach for sharing JavaScript based projects.