**Part 1: Closest Pairs**

**README:**

**Python version**: 3.6.2

Input File: closestPairsInput.txt

Input File Format: The input file for the closest pairs problem consists of the list of points to be analyzed. Each point is on its own line, with the x and y coordinate comma-delimited, with no (), [], or {}. One example of a correct input for 5 points would be the following:

1, 2  
3, 5  
5, 6  
10, 1  
1, 9

**Input File Alternate Format**: There is an alternate format for the input file which creates a random point cloud. The format for this file requires that the first line be the string “Randomize Points”. The second line of the file indicates the number of points as an integer. The third and fourth lines of the file represent the maximum x and maximum y values of each of the points, respectively. One example of a correct input for 5 points with a maximum value of 10 for the x value and 10 for the y value would be the following:

Randomize Points  
5  
10  
10

A file is included in the submitted package with the designated formatting for convenience. The input file is the same for both implementations of the closestPairs.

**Execution**: There are two .py files in the directory. ClosestPairsAMain.py will run a brute force algorithm which checks each point in the point cloud and return the points with the closest distance. This program corresponds with part a of the programming assignment. ClosestPairsBMain.py will return the closest points as well, but utilizes a slightly faster algorithm. Each program has its own \_\_main\_\_ function, so you can run each separately from the command line or in your favorite python IDE. The code will read the values in the input file and write the result to an output file corresponding to the file ran, and if the output is less than 30 lines it will output to the console as well.

**Output (A):** After running the code results will appear in closetsPairsAOutput.txt. The output here will first determine how many points are in the set and echo the points to the output. Next there will be some debugging level messages to show each time that the algorithm discovers a pair of points with distance between them less than the previous closest pair. Finally it will output the minimum distance between two points in the point cloud. Example output files are included in the submitted package for convenience.

**Output (B):** After running the code results will appear in closetsPairsBOutput.txt. The output here will first determine how many points are in the set and echo the points to the output. Next there will be some debugging level messages. For each recursive iteration of the algorithm, it will show what the input points are and show which points are in the left sector and right sector of the cloud. As the algorithm recourses down to a base case of 2-3 points, it will use the algorithm in part a to calculate the closest distance between the 2-3 points and display that as output. When the algorithm has to compute the distance in the center “strip”, it will display the points in the strip, the minimum distance outside the strip, and the minimum distance within the strip. For more details on the algorithm used see the Analysis section. Finally it will output the minimum distance between two points in the point cloud. Example output files are included in the submitted package for convenience.

**ANALYSIS:**

**Psuedo-Code (A):** Brute force algorithm

brute\_closest(points): // get the points  
 min\_dist = infinity // set the initial minimum distance really high  
 for pointa in points: // iterate over each point  
 for pointb in points: // iterate over each point again  
 if pointa != pointb: // ensure the points are not the same point  
 dist = distance\_between(pointa, pointb)  
 if dist <= min\_dist: // check if the measured distance is the new smallest  
 min\_dist = dist // set the new smallest if that is the case  
 min\_point\_a = pointa // store the value of pointa  
 min\_point\_b = pointb // store the value of pointb  
 return min\_dist, min\_point\_a, min\_point\_b

**Runtime (A):** For a problem with N points, this will take O(n^2) to solve. Everything within the for loops can be done in O(1) time, and each for loop takes O(n) time. Nesting one for loop in the other produces an algorithm that runs in O(n2) time. Since each point is checked against each other point and the minimum distance is calculated for each, the returned pair must be the closest pair. This implementation will do (n-1)2 comparisons.

**Psuedo-Code (B):** An algorithm that runs faster than brute force. This approach is divided into a few steps shown below for readability, and the “real” pseudo code is shown below that:

1. Sort points according to x and y coordinates (O(nlgn) with merge sort).
2. Divide the points into left and right half and recursively solve for min dist. Maintain the sort for each half in x and y.
3. Take the minimum of the left min dist and right min dist.
4. Create a “strip” at the midpoint of the cloud and include all points that are less than min dist from the midpoint by x coordinate. The reason for this strip is that the closest points may be in the middle of the point cloud e.g. one of the closest set get put in the left half of the cloud, and one gets put on the right.
5. Find the minimum distance in the strip. This can be done in O(n) if the strip is sorted in y.
6. Return the minimum of the min dist and the strip min dist.

\_\_main\_\_:  
 points = get\_points() // get the points  
 points\_x = merge\_sort(points, x) // merge sort the points by x coordinate O(nlgn)  
 points\_y = merge\_sort(points, y) // merge sort the points by y coordinate O(nlgn)  
 closest\_fast(points\_x, points\_y) // feed both sorted lists to recursive algorithm  
  
closest\_fast(points\_x, points\_y):  
 if len(points\_x) <= 3:  
 return brute\_closest(points\_x) // return brute force from A if less than 3 points O(1)  
 mid\_index = len(points\_x) / 2 // get the index of the middle point  
 midpoint = points\_x[mid\_index] // get the midpoint in x of the point cloud   
 points\_x\_left = points\_x[:mid\_index] // get the left half of point cloud sorted in x (O(n))  
 points\_x\_right = points\_x[mid\_index:] // get the right half of point cloud sorted in x (O(n))  
 for point in points\_y:  
 if point.x < midpoint.x:  
 points\_y\_left.append(point)  
 else:  
 points\_y\_right.append(point) // get the left/right half of point cloud sorted in y (O(n))  
 min\_left = closest\_fast(points\_x\_left, points\_y\_left) // recursively solve left hand side  
 min\_right = closest\_fast(points\_x\_right, points\_y\_right) // recursively solve right hand side  
 min\_dist = min(min\_left, min\_right) // get the minimum between both sides  
 strip\_points = [] // closest maybe between LHS/RHS, construct “strip”  
 for point in points\_y: // append points to strip sorted in y  
 if abs(point.x – midpoint.x) < min\_dist  
 strip\_points.append(point) // create strip with all points within Δx=min\_dist of mid  
 min\_strip = closest\_strip(strip\_points) // calculate the closest points within the strip  
 return min(min\_strip, min\_dist) // return the closest pair in the set  
  
closest\_strip(points, min\_outside\_dist): // note the points are sorted in y and runs in O(n)  
 min\_dist = min\_outside\_dist // initialize the minimum distance   
 for index in range(len(points) // for N points index=0,1,2,3,…,N  
 left\_index = index  
 right\_index = index // index values to track points above and below current  
 left\_index -= 1 // check the point to the left of current first  
 // ensure left\_index is valid and the observed point is not further than min\_dist  
 while left\_index >= 0 and points[index].y – points[left\_index].y < min\_dist:  
 // check if there is a new minimum distance  
 if distance\_between(points[index], points[left\_index]) < min\_dist:  
 min\_dist = distance\_between(points[index], points[left\_index]  
 left\_index -= 1 // go to the next leftmost point  
  
 right\_index += 1 // check the point to the right of current first  
 // ensure right\_index is valid and the observed point is not further than min\_dist  
 while right\_index < len(points) and points[right\_index].y - points[index].y < min\_dist:  
 // check if there is a new minimum distance  
 if distance\_between(points[index], points[right\_index]) < min\_dist:  
 min\_dist = distance\_between(points[index], points[right\_index])  
 right\_index += 1 // go to the next rightmost point  
 return min\_dist // return the closest distance in the strip

**Runtime (B):** For a problem with N points, this will take O(nlgn) to solve. The key to this is being able to solve the minimum distance in the strip in O(n) time. Seeing the while() loops nested in the for() loops may make it seem that this is not the case at first. However, we can take a closer look at the points in the strip.

For each point to the left of the midpoint, we can safely determine that the distance between any pair of points is no less than the size of the strip. This is because the size of the strip is constructed based on the minimum distance between points outside of the strip. We can construct a bounding are within which points on the LHS will satisfy the while() loop (within strip size in y coordinate of the point being analyzed). Within this bounding box, we can fit no more than 4 points with a distance between them that is less than the strip size (see figure below for illustration).

Extrapolating this analysis on the right hand side, we will have to analyze no more than 8 points for each iteration of the for() loop. Thus everything within the for() loop can be done in O(1) time, and finding the minimum distance within the strip can be done in O(n).

We only have to construct the strip O(lgn) times, thus the total time to compute the minimum distance is O(nlgn) time.

Several other tasks happen in parallel to this task. For instance, merge sort takes O(nlgn) time, and happens in parallel, and this is done in x and y. Constructing the strip takes O(n) time and happens O(lgn) times, thus takes O(nlgn) time as well. Dividing the points into two halves in x and y take O(n) time and occurs O(lgn) times. However, since O(nlgn) + O(nlgn) + … = O(nlgn), the algorithmic runtime for this algorithm is O(nlgn).

Splitting the points into two halves and checking the points within those two halves will not always return the closest pair. That is the reason that we have to consider the points in the strip, since the closest pair can be one point just to the left of the dividing line and another point just to the right of the dividing line. Analyzing a buffer area along the dividing line as well as the right and left then, must produce the closest pair, since two points outside of the buffer zone are known to have distance greater than an already measured distance between a pair. Thus this algorithm will always produce the closest pair in a point cloud.

This implementation will perform O(nlgn) comparisons.

**Empirical Measurements:**

While empirical measurements are prone to error and are dependent on a number of factors internal to your machine, they often still give good results for the algorithmic runtime of your program. For example, we can see below that the brute force algorithm matches well with an O(n2) runtime, while as the O(lgn) term is taken over by the O(n) term, we fit nicely to a linear runtime. See the two charts below and corresponding data table for an analysis of the empirical runtime:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| #points: | 10 | 100 | 250 | 500 | 750 | 1000 |
| Brute | 0.000931 | 0.017609 | 0.088419 | 0.363976 | 0.82509 | 1.432521 |
| Fast | 0.004503 | 0.049259 | 0.17262 | 0.40528 | 0.623479 | 0.819405 |

This gives a pretty good indication that my analysis for the runtime of the algorithms matches with the actual runtime of the algorithm.