Algorithms for Data Science
Lab 4
Closest Pair DAC

For this lab, you will be solving the following problem: Given a group of points, find the closest pair. However, to make this lab significantly easier to code, you only need to solve the problem of finding the *distance* between the closest pair of points. Additionally, you will work only with 1-D points (i.e., with only one coordinate, like points on a number line) instead of 2-D or 3-D points.

You can solve this problem in several ways. The obvious brute-force algorithm is to order the points and compare adjacent points' distances. Note, however, that this method doesn't work when extended into 2-D. The obvious 2-D method is to compare every pair of points, which is significantly more time consuming. For this lab, you are to solve this problem another way, using a recursive divide-and-conquer algorithm. The algorithm states that the distance between the closest pair of points in a list is minimum of

1. The distance between the closest pair of points in the first half of the list
2. The distance between the closest pair of points in the second half of the list
3. The distance between the closest pair of points where one point of the pair is in the first half of the list and the other point of the pair is in the second half of the list

Thus, we have the following three steps:

- Divide: Split the list into two equal pieces.
- Conquer: Recursively find the closest pair distance in each sublist (obtains distances 1 and 2 from above).
- Combine: Compute the remaining distance (3 from above), and combine them by taking the minimum of the three.

Note that with some minor adjustments, this algorithm does extend to 2-D points, but you don't have to extend it for this lab.

You are to implement this algorithm recursively in the following way. You should have a function called

```
def cPairDist(points):
```

It takes in a list of 1-D points (i.e., integers) and returns the distance between the closest pair of points. Note that the divide-and-conquer algorithm given above is made significantly easier if the incoming list of points is in increasing order (i.e., the divide part is then made trivial). However, we can't assume that it will be passed in sorted. Thus, we will have to sort the

list before we perform the divide. The native approach is to perform this extra sorting work at each level in our recursive call tree. A better approach is to perform this sorting once, before the actual algorithm is started. One often sees situations where this type of initial work needs to be performed to get the original input into a form that is better for the recursive algorithm.

After you have performed your sort, you need to call another function that performs the recursion. The signature you should use follows:

```
def recCPairDist(points):
```

This function takes in a sorted list of points and performs the divide/conquer/combine steps outlined earlier. Note that the recursive calls will be to recCPairDist. This function returns the min distance it finds. Note that the distance between two 1-D points is simply the absolute value of the difference between them. Pay special attention to your base case(s).

You are to demo your working code on the following three lists of 1-D points:

1. [7, 4, 12, 14, 2, 10, 16, 6]
2. [7, 4, 12, 14, 2, 10, 16, 5]
3. [14, 8, 2, 6, 3, 10, 12]