Brandon Dring & Alec Zitzelberger

1.) Pseudo-code. Please provide the Pseudo-code for each of the three algorithms.

```
BruteForce
Read in points

Bruteforce(points)
    Minimum = distance between point1 and point 2
    MinArray = [] // empty

    For i in points
        j= i + 1;
        For j in points
            Check = i and j distance
            If Check < Minimum
                Clear MinArray and put in the new points
                Minimum = Check
            If Check = Minimum
                Add the points to MinArray
    Output results
```

-

```
NaiveDnC
Main
    Read in points
    minpoints= []
    Array = buildArray(points)
    Sort Array by x
    results=getSmallestDistance(Array, Array size, minpoints)
    Print results

getSmallestDistance(array, n, minpoints)
    If n <= 3
        Return minpoints using bruteforce // essentially like above

    midpoint= int(n/2)
    middleValue= array(midpoint)

    leftSmallest = getSmallestDistance(array up to midpoint, midpoint, minpoints)
    rightSmallest= getSmallestDistance(array after midpoint, n - midpoint, minpoints)

    d= min(leftSmallest, rightSmallest)
    pointStrip= []
    For i in array
        If the difference in x of a point to the middle is less than d
            pointStrip.append(i)

    return(min(d, prunemid(pointStrip, d, minpoints)))

prunemid(shortPoints, min, minpoints)
    Sort shortPoints by y

    For i in shortPoints -1
        j= i + 1
        While the y distance between i and j < min and j < length of shortPoints
            If the distance between i and j is smaller than the current points in
minpoints distance
                Mininum is the new distance
                Minpoints is cleared and the new points are put in

            Else if they are the same
                Add the points to the minpoints

            J++
    Return minimum
```

-

```
EnhancedDnC
Main
    Read in points
    minpoints= []
    Array = buildArray(points)
    Sort Array by x
    arrayY = deepcopy(Array)
    Sort arrayY by y
    results=getSmallestDistance(Array, arrayY, Array size, minpoints)
    Print results

getSmallestDistance(array, arrayY, n, minpoints)
    If n <= 3
        Return minpoints using bruteforce // essentially like above

    midpoint= int(n/2)
    middleValue= array(midpoint)

    Pyl = []
    Pyr = []

    For i in arrayY // sort based on the middle into two sub arrays
        If i[0] <= middleVlaue
            Pyl.append(i)
        Else
            Pyr.append(i)

    leftSmallest = getSmallestDistance(array up to midpoint, Pyl, midpoint, minpoints)
    rightSmallest= getSmallestDistance(array after midpoint, Pyr, n - midpoint, minpoints)

    d= min(leftSmallest, rightSmallest)

    pointStrip= []
    For i in arrayY
        If the difference in x of a point to the middle is less than d
            pointStrip.append(i)

    return(min(d, prunemid(pointStrip, d, minpoints)))

prunemid(shortPoints, min, minpoints)

    For i in shortPoints -1
        j= i + 1
        While the y distance between i and j < min and j < length of shortPoints
            If the distance between i and j is smaller than the current points in
minpoints distance
                Mininum is the new distance
                Minpoints is cleared and the new points are put in

            Else if they are the same
                Add the points to the minpoints

            J++
    Return minimum
```

2.) Asymptotic Analysis of run time. Please analyze the runtime for the three algorithms. In particular, please provide the recursive relation of the runtime for algorithm 2 and 3 and solve them.
The Brute force method is simply O(n^2) because it requires checking every single combination of two points iteratively.

NaiveDnC has a recurrence relation T(n) = 2T(n/2) + cnlogn because at each step the problem is split in two halfs equally and each half must sort its own subarray of points by the y value. Sorting is O(nlogn) so each level does O(nlogn) work. When we solve this recurrence relation we see that the naive implementation is O($n\log^2 n$) which is better than Bruteforce but it can be improved.

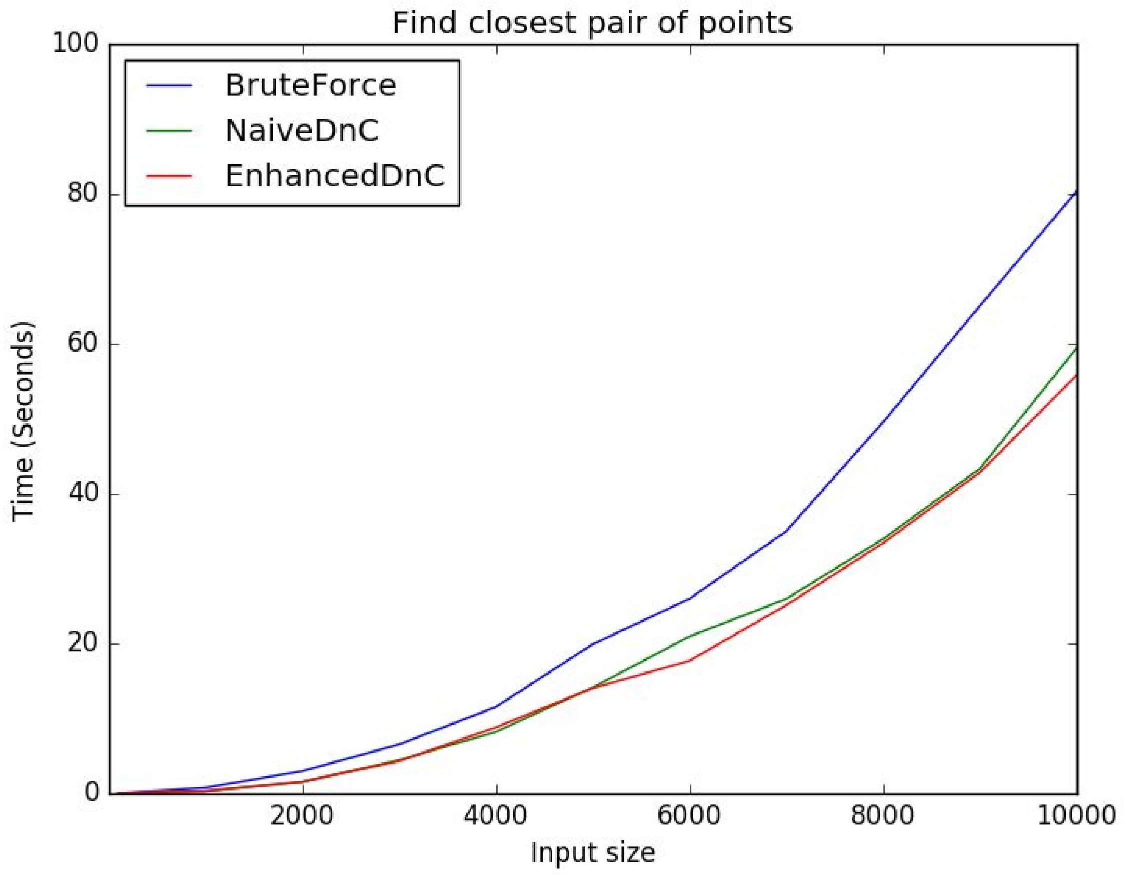
EnhancedDnC has a recurrence relation of T(n) = 2T(n/2) + cn because we do mostly the same as the naive version, but we eliminate the need to sort the array at each level by making a duplicate array that is sorted by y and passing down alongside the array that is sorted by x. At each level we split the y array based on the x split with O(n) work and pass the half down with the matching half of the x array. This way when we make the strip, we use the y array instead of the x, and because it is presorted at the beginning by y, it retains this when its points are placed in the strip. This bypasses the need to sort by y at each level which means that the Big-O at each level is O(n). When we solve the recurrence relation, we end up with T(n) = O(nlogn) which is better than that of NaiveDnC.

3.) Plotting the runtime. Plot the empirically measured runtime of the three algorithms as a function of the input size. Your plot should have clearly labeled axes and legends.

Check the .zip folder that was submitted

4.) Interpretation and discussion Discuss the runtime plot. Do the growth curves match your expectation based on their theoretical bounds? Discuss and provide possible explanations for any discrepancy between the experimental runtime and the asymptotic runtime.

At first they didn't particularly match what the theoretical speed up moving from Naive DnC to the Enhanced version. After some tweaking, and writing a little bash script to handle all tests, I do see about a 10 second difference between the two. Which in a 62 second long period is about an ~18% difference? Which is roughly what is expected when moving from $(n \log n)^2$ to a (n log n) runtime, so everything in the end worked out as it was supposed to. When we were having trouble at first with trying to speed up the Naive DnC, we thought it had something to do with how Python isn't particularly memory efficient or fast. Due to in the Enhanced version one must create the a master copy of the sorted graphs points on each axis. Which as variables are passed by reference in Python, meant that they were deep-copied, which is horridly slow. So slow, that we thought it was slowing the program down to the Naive speed. I've seen the latter of the two algorithms run anywhere from 1 second faster to 12 seconds faster, depending on how the computer is feeling at that moment in time.


Find closest pair of points
BruteForce
NaiveDnC
EnhancedDnC
Time (Seconds)
Input size