Divide and Conquer Algorithm Implementation

Group names: Glen Anderson, Justin Chang, Ronald Salinas

Pseudocode

Brute Force:

```
M = Array of inputs (Contains X and Y)
C = distance of M[0] and M[1]                          O(c)
For i=0 in M                                           O(n)
     For j=i+1 in M                                    O(n^2)
          D = the distance between i and j              O(c)
          If C > D
               C = D
```

Naive:

```
Divide_and_Conquer(Point)
     if (count <= 3)
          then calculate min distance using brute force    (O(n))
          return min distance
     else
          Calculate median of the graph                          O(c)
          Split graph into a leftHalf and rightHalf              O(c)
          deltaL = Divide_and_Conquer(leftHalf)       2O(log n)
          deltaR = Divide_and_Conquer(rightHalf)
          deltaBest = min(deltaL, deltaR)

          Scan graph and place into a strip My all points
          within deltaBest of median                      O(n)
          Sort array by Y value                        O(nlogn)
          return Closest_Cross_Pair(My, deltaBest)

Closest_Cross_Pair(My, deltaBest)
     Save deltaBest into Dmin
     For i  from 0 to the size of My - 1
          j = i + 1
```

```
                while the Y-distance from point i to point j is <=
                deltaBest and j <= My
                      Find distance between point i to point j and
                      save into delta2

                      Dmin = min(deltaBest, delta2)
                      j = j + 1
          return Dmin
```

Enhanced:

```
Divide_and_Conquer(Point X, Point Y)
     if (count <= 3)
          then calculate min distance using brute force    (O(n))
          return min distance
     else
          Calculate median of the graph                    O(c)
          Split graph to leftHalf and rightHalf for X sorted array
                                                 O(c)
          Split graph to leftHalf and rightHalf for Y sorted array
          O(c)
          deltaL = Divide_and_Conquer(leftHalfX,leftHalfY)
          O(logn)

          deltaR = Divide_and_Conquer(rightHalfX, rightHalfY)
          O(logn)

          deltaBest = min(deltaL, deltaR)

          Scan graph and place into a strip My all points within
          deltaBest of median                              O(n)
          Sort array by Y value                      O(nlogn)

          return Closest_Cross_Pair(My, deltaBest)

Closest_Cross_Pair(My, deltaBest)
     Save deltaBest into Dmin

     for i  from 0 to the size of My - 1
          j = i + 1
          while the Y-distance from point i to point j is <=
          deltaBest and j <= My
                Find distance between point i to point j and save
                into delta2
```

```
            Dmin = min(deltaBest, delta2)
            j = j + 1
return Dmin
```

Asymptotic Analysis of Run Time

Brute Force:

T(n) = $T(n^2)+T(n)+2c$
= **O(n^2)**

Naive:

T(n) = 2T(n/2) + cn*log(n)

Relation:

T(k)	= $2^kT(n / 2^k)$ + kcn*log(n)
T(k = log(n/2))	= $2^{log(n/2)}$*T(n / $2^{log(n/2)}$) + log(n/2)*cn*log(n)
	= (n/2)*T(n/(n/2)) + log(n/2)*cnlog(n)
	= (n/2)*T(2) + log(n/2)*cnlog(n)
	= **O(nlog^2(n))**

Enhanced:

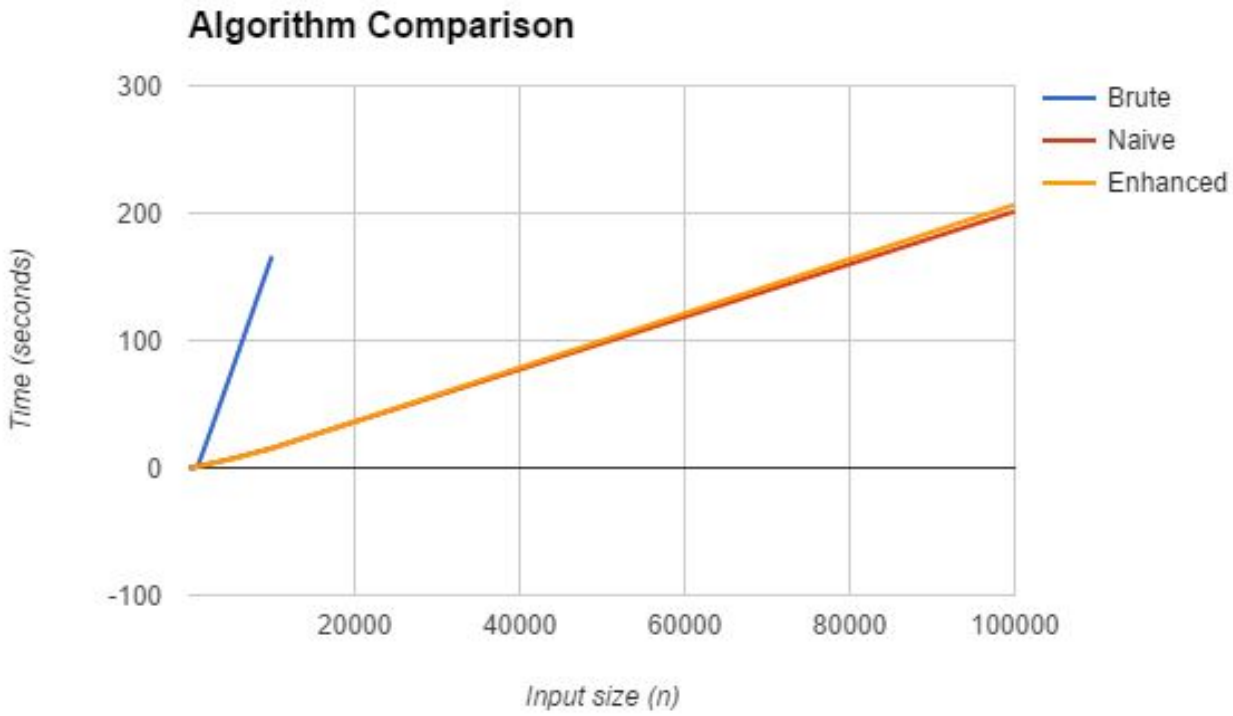
T(n) = 2T(n/2) + cn

Relation:

T(k)	= $2^kT(n / 2^k)$ + kcn
T(k = log(n/2))	= $2^{log(n/2)}$T(n / $2^{log(n/2)}$) + log(n/2)*cn
	= (n/2)*(T(n/(n/2)) + log(n/2)*cn
	= (n/2)*T(2) + log(n/2)*cn
	= **O(nlog(n))**

Plotting Run Time

Input size (n):	Brute-force	Naive divide and conquer	Enhanced divide and conquer
10	.0158	.0527	.03854
100	.0347	.2051	.1856
1000	1.62	1.145	1.075
10000	166.3	15.48	15.12
100000	>20 minutes, could not run on OSU's server as it ran too long	201.3	206.5



Interpretation and Discussion

Discussion of plot:

For the smallest input sizes (into the hundreds), the brute force algorithm is faster at finding the closest pair of points. However, as the size of the input gets larger, it becomes clear that the divide and conquer algorithms are significantly faster. While the enhanced version should be faster than the naive version, there are a number of factors that could have influenced their testing to cause the unexpected difference. Still, the data does not entirely match our expectations based on theoretical bounds.

Possible discrepancies between experimental and asymptotic runtime:

Since Python is an interpreted language, there could be tasks such as reading files and sorting arrays that potentially take longer than O(n), even if this is how we would expect them to run. This is likely what happened in the case of our naive and enhanced versions, where they seem to run at roughly the same speed even for very large values of n. Since we have to do an extra sort and we are using Python (slower than C), the algorithm could be slowed down substantially by this. Where file operations are concerned, they should impact all of the data equally regardless of the algorithm, since file operations are performed on each. Furthermore, change in the speed of the tester's system or internet connection could cause discrepancies between tests.

In addition to these outside factors, the theoretical analysis of the run time assumes a value of n that approaches infinity whereas our value of n cannot, as it would be impossible to test. We try to replicate this by using very high values for n, but it could still cause a discrepancy, particularly with smaller data sets. For small values of n, the brute force algorithm was actually more efficient. This makes sense, as it doesn't need to perform as many operations on the input size and does not bother pre-sorting arrays, which adds up to n additional operations each time it is done.

Another factor that can affect the runtime is copying operations, such as function calls and variable assignments. While variable assignments are roughly equal between programs, there are more function calls in the enhanced version as it needs to recurse over both the x and y sorted arrays. This additional copying could cause a slower run time, particularly with much larger data sets.