Part One

Pseudo Code for my implementation:

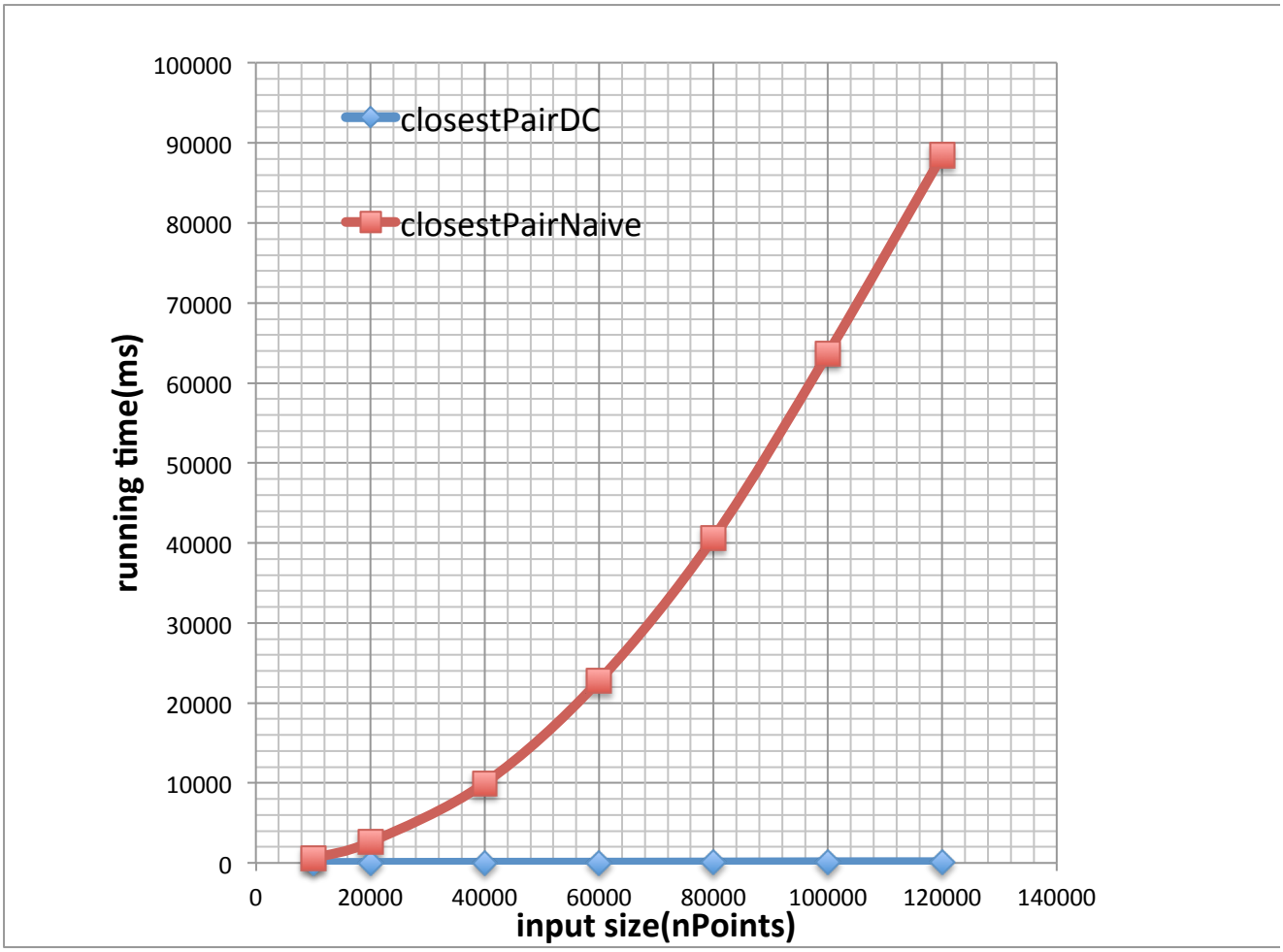
```
ClosestPair(ptsByX, ptsByY, n)
if (n = 1) return ∞
if (n = 2) return distance(ptsByX[0], ptsByX[1])

// Divide into two subproblems
mid ← ⌈n/2⌉ − 1
copy ptsByX[0... mid] into new array XL in x order.
copy ptsByX[mid+1 . . . n − 1] into new array X R in x order.
copy ptsByY into arrays Y L and Y R in y order, s.t.
XL and YL refer to same points, as do XR and YR

// Conquer
distL ← ClosestPair(XL, Y L, ⌈n/2⌉)
distR ← ClosestPair(XR, Y R, ⌊n/2⌋)

// Combine (was a separate procedure in class) midPoint ← ptsByX[mid]
lrDist ← min(distL, distR)
Construct array yStrip, in increasing y order,
of all points p in ptsByY s.t. |p.x − midPoint.x| < lrDist
minDist ← lrDist
for (j ← 0; j ≤ yStrip.length − 2; j++)
    k←j+1
    while (k ≤ yStrip.length − 1 and yStrip[k].y − yStrip[j].y < lrDist)
        d ← distance(yStrip[j], yStrip[k])
        minDist ← min(minDist, d)
        k++
return minDist
```

Conclusion:
For sufficiently small input size, naïve algorithm could be faster; for larger input size, divide-and-conquer algorithm is definitely much faster.

Part Two

1. Naive VS Divide-and-conquer comparison

nPoints	DC(ms)	Naive(ms)
10000	7.767	584.237
20000	15.789	2613.35
40000	34.813	9974.17
60000	45.952	22798.5
80000	66.551	40645.4
100000	96.881	63714.8
120000	102.747	88462.7



Conclusions:

For large input sizes, divide-and-conquer is a powerful tool for solving difficult problems. It is much faster than naive algorithm because divide-and-conquer works by splitting up the original problem into smaller subproblems and then solves the smaller subproblems, these solutions reduce the total amount of work to do with respect to solving the original problem so yields Algorithm efficiency and Parallelism.

2. Variation in time of divide-and-conquer algorithm

1) Lab1 was modified to time 100 randomly generated inputs of size 50000 and report the min, max, average running times

45.577, 47.76, 54.979, 49.319, 42.733, 43.163, 42.529, 43.735, 42.455, 43.278,
43.518, 43.768, 44.875, 40.519, 46.486, 43.816, 41.347, 51.814, 41.896, 44.07,
52.34, 58.412, 69.399, 66.959, 66.743, 68.335, 64.307, 67.636, 65.798, 64.027,
71.539, 60.126, 45.637, 44.695, 48.083, 48.135, 44.33, 44.21, 48.919, 43.525,
44.903, 44.91, 48.553, 41.09, 43.15, 41.476, 43.938, 44.563, 49.922, 41.307, 44.761,
40.684, 47.701, 44.911, 44.299, 46.077, 40.417, 44.872, 41.413, 56.1, 44.841,
43.254, 44.776, 41.988, 47.325, 50.16, 42.955, 44.751, 44.481, 47.623, 44.757,
40.833, 44.01, 46.614, 43.388, 41.672, 40.619, 44.415, 48.762, 44.845, 44.581,
50.02, 47.212, 43.585, 45.193, 47.212, 44.371, 40.86, 41.059, 40.591, 40.875,
44.643, 41.374, 44.297, 41.775, 49.155, 48.312, 46.663, 46.558, 46.777,
min: 40.417
max: 71.539
average: 47.2302
standard deviation: 7.27404
95% confidence interval: (47.2302-1.42571,47.2302+1.42571)

2) do 100 runs with the same input size of 50000 and report the min, max, average running times

46.607, 45.216, 45.59, 43.416, 44.961, 43.916, 46.191, 46.723, 45.733, 45.16,
43.329, 46.297, 48.183, 46.94, 44.023, 47.848, 44.35, 51.661, 49.389, 44.678,
44.497, 45.095, 48.005, 50.864, 50.222, 52.627, 44.905, 43.636, 45.628, 51.094,
45.165, 43.476, 44.028, 50.647, 46.724, 50.328, 45.811, 47.094, 43.345, 45.876,
44.917, 46.359, 44.2, 51.227, 44.035, 44.05, 45.09, 51.93, 45.917, 45.49, 44.73,
45.458, 46.518, 55.762, 44.287, 43.2, 46.597, 55.086, 46.402, 60.092, 50.41, 44.478,
52.744, 46.17, 47.576, 46.675, 57.989, 61.068, 46.555, 44.65, 50.025, 48.22, 44.597,
45.732, 48.343, 54.597, 49.688, 52.69, 44.514, 44.664, 45.291, 43.596, 50.825,
45.778, 46.163, 52.151, 53.874, 48.159, 52.368, 46.911, 45.417,
47.802, 48.472, 50.627, 44.657, 59.324, 44.893, 55.699, 51.704, 47.973
min: 43.2
max: 61.068
average: 47.7369
standard deviation: 3.94713
95% confidence interval:(47.7369-0.773637,47.7369+0.773637)

Conclusions:
From my tests, the variation from one input to the next(first analysis) is a bit larger than the variation on the same input size(second analysis). Variations in time to run on the same input can be attributed to external factors.

Part Three

If the input size is small, the naïve algorithm might be a little faster because the code is simpler, thus is easier to implement, and there is less overhead.

I have run each algorithm in a loop for 100000 trials.

For n = 108,
Total running time for DC: 9145.57
Total running time for Naive: 8862.91
Average running time for DC: 0.0914557
Average running time for Naive: 0.0886291

For n = 109,
Total running time for DC: 9392.98
Total running time for Naive: 9322.87
Average running time for DC: 0.0939298
Average running time for Naive: 0.0932287

For n = 110,
Total running time for DC: 9331.17
Total running time for Naive: 9609.68
Average running time for DC: 0.0933117
Average running time for Naive: 0.0960968

For n = 111,
Total running time for DC: 8850.42
Total running time for Naive: 9114.22
Average running time for DC: 0.0885042
Average running time for Naive: 0.0911422

For n = 112,
Total running time for DC: 8800.38
Total running time for Naive: 9379.01
Average running time for DC: 0.0880038
Average running time for Naive: 0.0937901

Therefore, the “crossover” input size is 110!

Summary:
For sufficiently small input size, naïve algorithm could be faster due to its simplicity.
And from my tests, the crossover input size is 110.