Gavin Wolf
COT 6405: Analysis of Algorithms
Professor: Dr. Mihaela Cardei
Project
Due Date: 4/21/16

The Closest Pair of Points Problem

1. Introduction

In this paper I will present and evaluate the experimental results of the program I built to compute the real execution times for two different algorithms that solve the *Closest Pair of Points Problem*. The *Closest Pair of Points Problem* is quite straightforward: given a set of points in a plane, find the closest pair of points. A brute force solution to the problem is also quite straightforward: compute the distance between each pair of points and return the minimum distance. The brute force solution has a running time of $O(n^2)$. A more efficient algorithm, however, is not as obvious or straightforward. The first O(n log n) solution, using the divide and conquer technique, was discovered by Shamos and Hoey in the early 1970s. An even more efficient solution using randomization has a running time of O(n), although I do not plan to address that solution as part of this project (KT 226). The program I built includes both the brute force algorithm and an algorithm that uses the divide and conquer technique.

2. Real-World Applications

The *Closest Pair of Points Problem* arises frequently in many domains: any time you want to know the closest two objects in a group. Such objects could correspond to airplanes, post offices, database records, statistical samples or DNA sequences (Levitin 108), as well as parts in a computer chip, stars in a galaxy, or irrigation systems (Manber 278). In controlling air traffic, for instance, it would be useful to know which two planes are closest to one another in order to prevent a collision (CLRS 1039). Other real-world applications include: computer vision, geographic information systems and molecular modeling (KT 226).

3. Pseudocode

I based my program code on the following pseudocode, which I adapted from KT (KT 230-231). I should note that this pseudocode is high-level and does not fully express the minutiae of implementing the algorithm.

```
closestPair(P, n)
  Construct Px and Py //Px is all the points sorted by x-coordinate and
                      //Py is all the points sorted by y-coordinate
  return closestPairRec(Px, Py)

closestPairRec(Px, Py, n)
  if n <= 3
    find the distance between the closest pair of points by measuring all pairwise distances
      //i.e., brute force

  Construct Qy and Ry //Qy is all the points in the left half of P, sorted by y-coordinate
                      //Ry is all the points in the right half of P, sorted by y-coordinate
```

```
dl = closestPairRec(Px, Qy, ceiling(n / 2))
dr = closestPairRec(Px + n / 2, Ry, floor(n / 2))
midpoint = Px[ceiling(n / 2)]

Construct S //S is the set of points in P within d distance of a line passing through the midpoint of the
            //x-coordinates, sorted by y-coordinate

Find the distance between the closest pair of points in S by measuring all pairwise distances
  //although this may seem like an O(n^2) calculation, it is actually O(n) because of the
  // mathematical property that any 2 points in S must be within 15 positions of each other.
Let ds be the minimum of these distances

return min(dl, dr, ds)
```

4. Experiment Methodology

In order to display the performance of the two algorithms, I graphed the execution times of the algorithms (y-axis) as a function of the input size (x-axis). For each input size, I ran each algorithm twenty times and then plotted the average execution time. For each input size, I also computed the hidden constant by dividing the (average) actual execution time by the theoretical running time. Then, I calculated the minimum and maximum values of c, and plotted a line representing each the minimum value and maximum value of the hidden constant multiplied by the theoretical running time for each input size. The "tightness" of the minimum and maximum lines indicate the accuracy of the theoretical complexity estimates when compared to the real execution times.

I calculated the execution times in milliseconds and used twenty different input values, ranging from 1,000 to 20,000 (in increments of 1,000). I implemented the program in C++, using its rand() function to generate the "points" used as inputs.

5. Experiment Results

5.1 Brute Force vs. Divide and Conquer Results

Figure 1 shows the real running times for the Brute Force and Divide and Conquer algorithms. Because the Brute Force algorithm has a theoretical complexity of $O(n^2)$ and the Divide and Conquer algorithm has a theoretical complexity of O(n lg n), the Brute Force algorithm grows at a much faster rate. In order to display the results for both of the algorithms, I displayed each algorithm on a different scale. The Brute Force algorithm is plotted using the left scale, which goes up to 2 million, while the Divide and Conquer algorithm is plotted using the right scale, which goes up to 16 thousand. Based on these scales, for an input size of 20 thousand, the execution time for the Brute Force algorithm is three orders of magnitude greater than the execution time for the Divide and Conquer algorithm – 10^6 vs. 10^3.

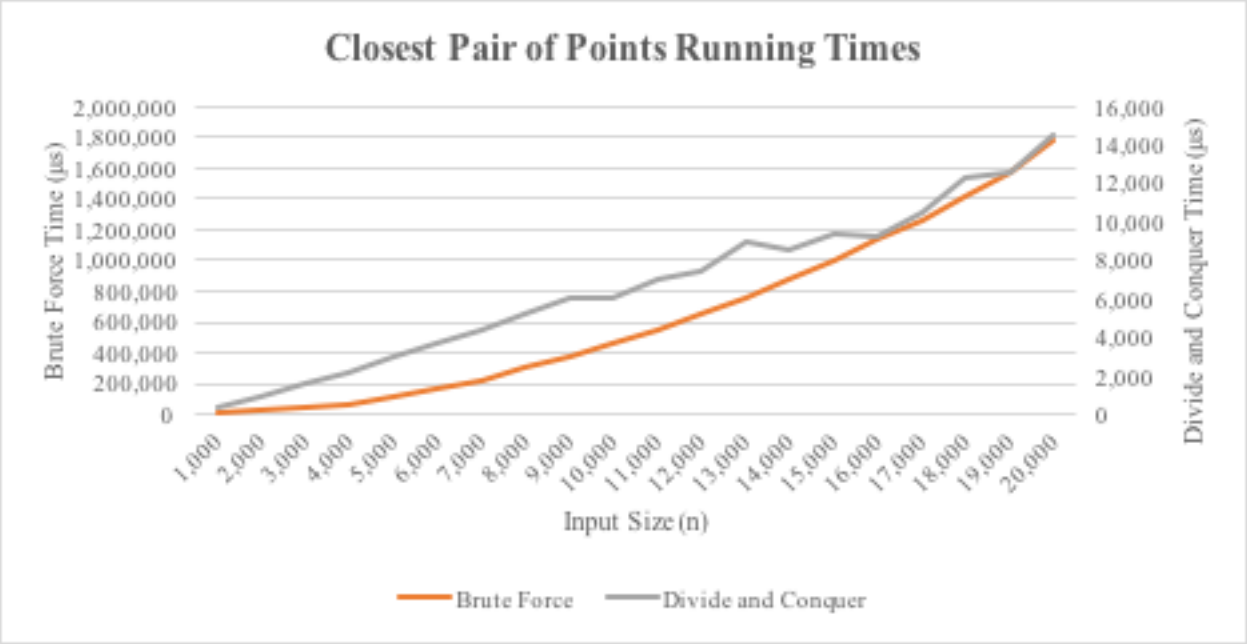

Figure 1

5.2 Brute Force Results

Table 1 shows a comparison of the actual and theoretical running times for the Brute Force algorithm. O-notation gives an upper bound on a function within a constant factor. That constant factor, the hidden constant, can be calculated by dividing the execution running time by the theoretical running time. Note: for the purposes of displaying the results nicely, I scaled the theoretical running time down by a factor of 1000 (this also scales down the hidden constant as well). The calculated hidden constant ranges from a minimum of 4.4 to a maximum of 6.1. Excluding, the first three input sizes, the range is even tighter: 4.4 to 4.8. The hidden constant for the first three input sizes is significantly higher than it is for the other input sizes, which I suspect is due to noise related to the smaller sample sizes.

Brute Force Running Time (RT)

n	Execution RT (µs)	Theoretical RT $O(n^2)$ (÷ 1,000)	Hiddent Constant
1,000	6,055	1,000	6.1
2,000	22,435	4,000	5.6
3,000	45,056	9,000	5.0
4,000	71,963	16,000	4.5
5,000	112,924	25,000	4.5
6,000	162,041	36,000	4.5
7,000	218,081	49,000	4.5
8,000	307,738	64,000	4.8
9,000	374,547	81,000	4.6
10,000	463,725	100,000	4.6
11,000	548,270	121,000	4.5
12,000	646,143	144,000	4.5
13,000	752,939	169,000	4.5
14,000	877,735	196,000	4.5
15,000	994,846	225,000	4.4
16,000	1,135,838	256,000	4.4
17,000	1,262,150	289,000	4.4
18,000	1,411,677	324,000	4.4
19,000	1,574,238	361,000	4.4
20,000	1,775,817	400,000	4.4
Minimum Hidden Constant Value			**4.4**
Maximum Hidden Constant Value			**6.1**

Table 1

Figure 2 plots the execution time for the Brute Force algorithm, as well as a line representing each the minimum value and maximum value of the hidden constant multiplied by the theoretical running time. The line corresponding to the minimum hidden constant value fits tightly with the line for the actual execution time. As intimated above, the maximum constant value, which resulted from the smallest input size, may be skewed due to the small input size. Given the overall tightness of the range of hidden constants, especially for larger input sizes, the theoretical complexity estimates for the Brute Force algorithm are very accurate. Intuitively, this makes sense because the behavior of the Brute Force algorithm is highly predictable and stable because it compares every point in the set every time it is executed.

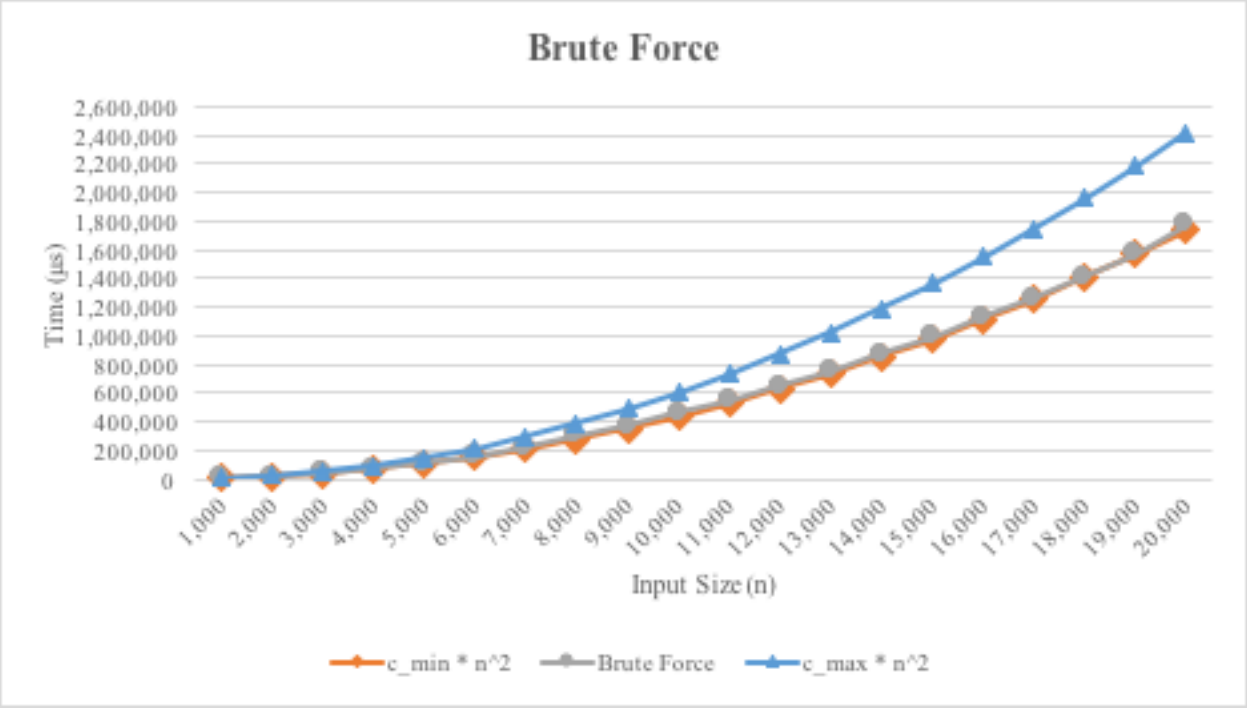

Figure 2

5.3 Divide and Conquer Results

Table 2 shows a comparison of the actual and theoretical running times for the Divide and Conquer algorithm. The calculated hidden constant ranges from a minimum of 41.1 to a maximum of 51.1. This range is noticeably larger than the range for the larger input values of the Brute Force algorithm. And unlike with the Brute Force algorithm, the variability in the hidden constant cannot be attributed to merely the sample sizes – for example, the minimum value, 41.1 occurs both with an input size of 1,000 and with an input size of 16,000. The variability in the hidden constant is likely due to the nature of the algorithm, particularly the step where the distance between the points in S is found ("Find the distance between the closest pair of points in S by measuring all pairwise distances"). In this step, the number of points in S will depend on the distribution of points in the set.

Divide and Conquer Running Time (RT)

n	Execution RT (µs)	Theoretical RT O(n lg n) (÷ 1,000)	Hiddent Constant
1,000	410	10	41.1
2,000	960	22	43.8
3,000	1,661	35	47.9
4,000	2,172	48	45.4
5,000	3,022	61	49.2
6,000	3,680	75	48.9
7,000	4,323	89	48.3
8,000	5,188	104	50.0
9,000	6,045	118	51.1
10,000	6,071	133	45.7
11,000	7,071	148	47.9
12,000	7,492	163	46.1
13,000	9,027	178	50.8
14,000	8,585	193	44.5
15,000	9,425	208	45.3
16,000	9,183	223	41.1
17,000	10,485	239	43.9
18,000	12,302	254	48.3
19,000	12,581	270	46.6
20,000	14,542	286	50.9
Minimum Hidden Constant Value			**41.1**
Maximum Hidden Constant Value			**51.1**

Table 2

Figure 3 plots the execution time for the Divide and Conquer algorithm, as well as a line representing each the minimum value and maximum value of the hidden constant multiplied by the theoretical running time. In this graph we see that the execution time jumps up and down within the band of minimum and maximum theoretical times. This volatility indicates that the theoretical running time is a less reliable predictor of the actual execution time for the Divide and Conquer algorithm than it is for the Brute Force algorithm. This is likely the case because the Divide and Conquer algorithm is more complex, with more room for variation. Even with the variability in the hidden constant values, the theoretical complexity estimates for the Divide and Conquer algorithm are still quite accurate.

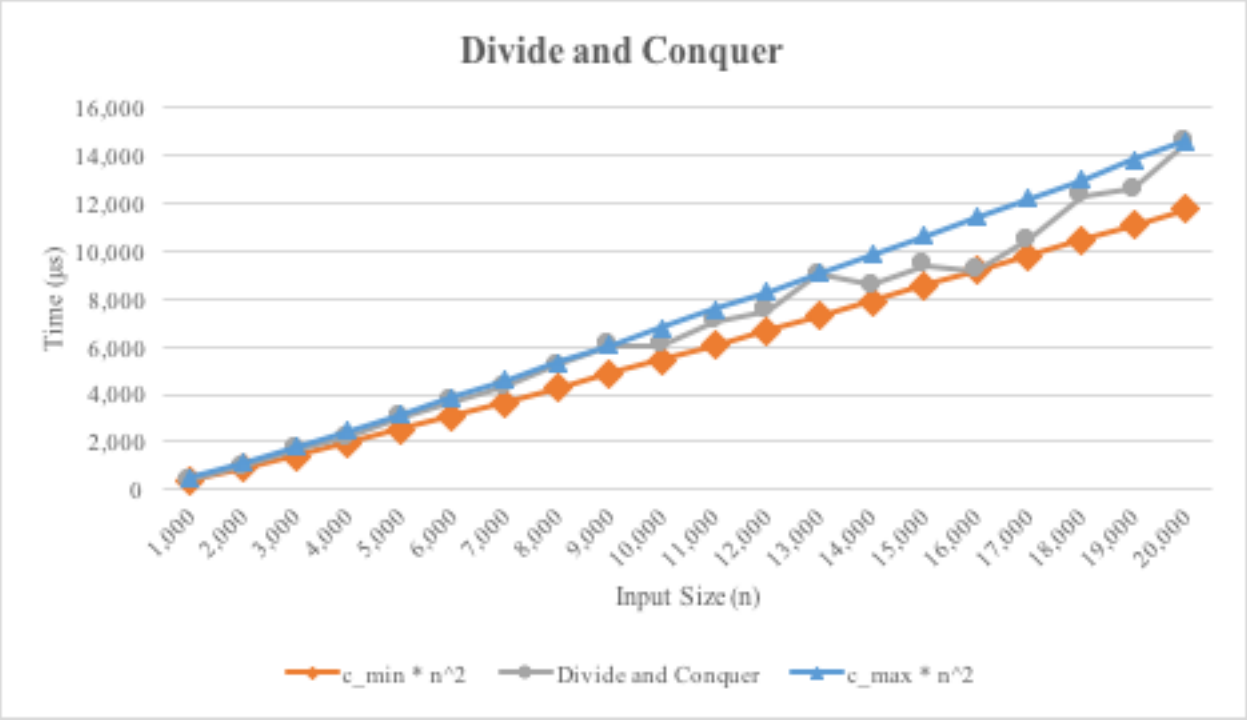

Figure 3

6. Notes on the Program

The algorithm requires sorting of arrays in O(n lg n) time. To ensure that this requirement is met, I implemented my own MergeSort algorithm. The sort() function in the C++ Standard Library can use either QuickSort (which has an average case running time of O(n lg n), but a worst case running time of $O(n^2)$) or IntroSort (which is a hybrid sorting algorithm that provides fast average performance and asymptotically optimal worst-case performance) (Musser 983-984). Due to the idiosyncratic natures of the running times for these algorithms, I used MergeSort instead, which likely has more consistent performance.

While I had hoped to come up with a fully automated program that looped through the input sizes from 1 thousand to 20 thousand and then output all of the results, this proved too taxing for my computer's memory. Instead, I manually ran the program for each input size. This seemed better than the alternative option of simply using smaller input values.

See Appendix A for the actual program code. See Appendix B for the results from a modified version of the program code, which shows the details of a computation with an input size of 10 – this is intended to demonstrate with actual values that the algorithms actually work.

Appendix A – The Program Code

```
/*****************************************************************************************
Name: Gavin Wolf      Z#: 15289719
Course: COT 6405: Analysis of Algorithms
Professor: Dr. Mihaela Cardei
Due Date: 4/21/2016
Programming Project

Description: This program implements two algorithms to compute the closest pair of points
 in a set, and measures their actual run-times over a number of trials.
*****************************************************************************************/

#include <iostream>
#include <float.h>
#include <random>
using namespace std;

/*****************************************************************************************
User inputs.
*****************************************************************************************/

//Enter input size up to 20000 to run:
int g = 20000;

//Select algorithm:
// 1 for bruteForce
// 2 for closestPair (divide and conquer)
int select = 2;

//Enter number of trials
int trials = 20;

/*****************************************************************************************
Struct, function and array declarations.
*****************************************************************************************/

//A structure to represent a point in a plane
struct Point
{
    unsigned int x, y;
};

//Function declarations
double dist(Point p1, Point p2);
double bruteForce(Point P[], int n);
double SClosest(Point *S, int size, double d);
double closestPairRec(Point *Px, Point *Py, int n);
double closestPair(Point *P, int n);
void Merge(Point *M, int p, int q, int r, int compareOption);
void MergeSort(Point *M, int p, int r, int compareOption);

//Array declarations
Point P[20000];

// for MergeSort
Point L[10001]; //+1 to account for sentinel value
Point R[10001]; //+1 to account for sentinel value

/*****************************************************************************************
Simulation and running time measurements.
*****************************************************************************************/
int main()
{
    //Initialize seed for random number generator
```

```
    srand(time(NULL));

    //Heading for output
    cout << endl << "Results:" << endl << endl << "# Elements\t\tRun Time\t\tTrials" << endl;

    int sumTime = 0; //Will be used to calculate the average

    //Loop to run multiple trials
    for (int i = 0; i < trials; i++)
    {
        //Fill array P with random values.
        // Note: rand() + rand() used instead of just rand() to increase the likelihood that
        //  the arrays will have distinct values.
        // Note: It is important to generate a new set of random numbers for each trial in
        //  order to test the algorithms' run times on different inputs.
        for (int j = 0; j < g; j++)
        {
            P[j].x = (unsigned int)(rand() + rand() - 1);
            P[j].y = (unsigned int)(rand() + rand() - 1);
        }

        //Store startTime of algorithm execution
        auto startTime = std::chrono::high_resolution_clock::now();

        //Execute algorithm
        if (select == 1)
        {
            bruteForce(P, g);
        }
        else
        {
            closestPair(P, g);
        }

        //Store finishTime of algorithm execution and calculate runTime
        auto finishTime = std::chrono::high_resolution_clock::now();
        auto runTime = finishTime - startTime; //Execution time
        long long microseconds =
std::chrono::duration_cast<std::chrono::microseconds>(runTime).count();

        sumTime += microseconds; //When loop exits, this will be the sum of all the trials
    }

    //Calculate average run time over all trials
    int averageTime = sumTime / trials;

    //Output results
    cout << g << "\t\t\t" << averageTime << "\t\t\t\t" << trials << endl;

    return 0;
}

/*****************************************************************************************
closestPair function - Divide and conquer algorithm
*****************************************************************************************/
//Constructs Px (all points sorted by x-coordinate in increasing order)
//Constructs Py (all points sorted by y-coordinate in increasing order)
//Calls closestPairRec (recursive function) on Px/Py
//Returns the closest pair of points
double closestPair(Point *P, int n)
{
    Point Px[n];
    Point Py[n];
    for (int i = 0; i < n; i++)
    {
```

```
        Px[i] = P[i];
        Py[i] = P[i];
    }

    //Use MergeSort to sort arrays
    MergeSort(Px, 0, n - 1, 1); //1 for compare x
    MergeSort(Py, 0, n - 1, 2); //2 for compare y

    //Call closestPairRec to find the smallest distance
    return closestPairRec(Px, Py, n);
}

/*****************************************************************************************
closestPairRec function - Recursive function
*****************************************************************************************/
//Recursive function to find the smallest distance
double closestPairRec(Point *Px, Point *Py, int n)
{
    //If there are 2 or 3 points, just use brute force
    if (n <= 3)
    {
        return bruteForce(Px, n);
    }

    //Find the middle point of the x-coordinates
    int mid = ((n - 1) / 2);
    Point midPoint = Px[mid];

    //Divide points in y sorted array around the vertical line.
    Point Qy[mid + 1]; //The left-half of P, sorted by y-coordinate
    Point Ry[n - mid - 1]; //The right-half of P, sorted by y-coordinate

    //Indexes of left and right subarrays
    int leftIndex = 0;
    int rightIndex = 0;

    //Construct Qy and Ry by traversing through Py
    for (int i = 0; i < n; i++)
    {
        if (Py[i].x <= midPoint.x)
        {
            Qy[leftIndex] = Py[i];
            leftIndex++;
        }
        else
        {
            Ry[rightIndex] = Py[i];
            rightIndex++;
        }
    }

    //Note: the pseudocode in Kleinberg & Tardos indicates to construct the following arrays:
    // Qx - The left-half of P, sorted by x-coordinate
    // Rx - The right-half of P, sorted by x-coordinate
    //Instead of constructing these arrays explicitly, the code below uses the arrays
    // implicitly by using "mid" to delimit the left and right halves of P.

    //Recursively find the closest pair among the points in Q and R
    double dl = closestPairRec(Px, Qy, mid);
    double dr = closestPairRec(Px + mid, Ry, n - mid - 1);

    //Take the minimum of the two distances
    double d = (dl < dr) ? dl : dr;

    //S is is the set of points in P within d distance of a line passing through the
```

```
    // midpoint of the x-coordinates, sorted by y-coordinate.
    //Note: S has the property that if any two points within it are closer to each other
    // than d, then those two points are within 15 positions of each other in S.
    // Therefore, the closest pair of points in S can be computed in linear time,
    // faster than the quadratic time found in the brute force method of finding the
    // closest pair of points in a set.
    Point S[n];
    int j = 0;
    for (int i = 0; i < n; i++)
    {
        long long int abso = 0;

        if ((Py[i].x - midPoint.x) < 0)
        {
            abso = -(Py[i].x - midPoint.x);
        }
        else
        {
            abso = (Py[i].x - midPoint.x);
        }

        if (abso < d)
        {
            S[j] = Py[i];
            j++;
        }
    }

    //Call helper function SClosest to computer the minimum distance in S
    return SClosest(S, j, d);
}

/*****************************************************************************************
SClosest function
*****************************************************************************************/
//Helper function to find the distance between the closest pair of points in S.
double SClosest(Point *S, int size, double d)
{
    double min = d; // Initialize the minimum distance as d

    //If two points are found whose y coordinates are closer than the minimum distance,
    // then investigate whether those points are closer than the minimum distance, and keep
    // track of the minimum value so far.
    for (int i = 0; i < size; i++)
    {
        for (int j = i + 1; j < size && (S[j].y - S[i].y) < min; j++)
        {
            if (dist(S[i], S[j]) < min)
            {
                min = dist(S[i], S[j]);
            }
        }
    }

    return min;
}

/*****************************************************************************************
bruteForce function
*****************************************************************************************/
//Computes the closest pair of points in P in a brute force manner - by computing the
// distance each pair of points and returning the minimum distance.
double bruteForce(Point *P, int n)
{
    double min = DBL_MAX;
```

```
    for (int i = 0; i < n; ++i)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (dist(P[i], P[j]) < min)
            {
                min = dist(P[i], P[j]);
            }
        }
    }
    return min;
}

/*************************************************************************************************
dist function
*************************************************************************************************/
//Computes the Euclidean distance between two points.
double dist(Point p1, Point p2)
{
    //Casting each int value to long long int so that the values can be safely multiplied,
    // i.e., without the risk of "overflowing" the capacity of int.
    long long int p1x = p1.x;
    long long int p2x = p2.x;
    long long int p1y = p1.y;
    long long int p2y = p2.y;

    return sqrt( (p1x - p2x) * (p1x - p2x) + (p1y - p2y) * (p1y - p2y) );
}

/*************************************************************************************************
MergeSort function
*************************************************************************************************/
//Recursively sorts the array of structs M by either the x-coordinates, or the y-coordinates.
// If compareOption == 1, the array of structs will be sorted based on the x-coordinate;
// otherwise, the array of structs will be sorted based on the y-coordinate.
void MergeSort(Point *M, int p, int r, int compareOption)
{
    if (p < r)
    {
        int q = (p + r) / 2;
        MergeSort(M, p, q, compareOption);
        MergeSort(M, q + 1, r, compareOption);
        Merge(M, p, q, r, compareOption);
    }
}

/*************************************************************************************************
Merge function
*************************************************************************************************/
//A helper function for MergeSort
void Merge(Point *M, int p, int q, int r, int compareOption)
{
    int n1 = q - p; //Last index of left array
    int n2 = r - q - 1; //Last index of right array

    //Construct the left half of array M
    for (int i = 0; i <= n1; i++)
        L[i] = M[p + i];

    //Construct the right half of array M
    for (int j = 0; j <= n2; j++)
        R[j] = M[q + j + 1];

    //Depending on the value of compareOption, combine the left and right halves of array
    // back into M.
```

```
    if (compareOption == 1)
    {
        L[n1 + 1].x = RAND_MAX + RAND_MAX; //The sentinel value
        R[n2 + 1].x = RAND_MAX + RAND_MAX; //The sentinel value

        int i = 0;
        int j = 0;

        for (int k = p; k <= r; k++)
        {
            if (L[i].x <= R[j].x)
            {
                M[k] = L[i];
                i++;
            }
            else
            {
                M[k] = R[j];
                j++;
            }
        }
    }
    else { //sort based on Y-coordinate
        L[n1 + 1].y = RAND_MAX + RAND_MAX; //The sentinel value
        R[n2 + 1].y = RAND_MAX + RAND_MAX; //The sentinel value

        int i = 0;
        int j = 0;

        for (int k = p; k <= r; k++)
        {
            if (L[i].y <= R[j].y)
            {
                M[k] = L[i];
                i++;
            }
            else
            {
                M[k] = R[j];
                j++;
            }
        }
    }
}

/*
***SAMPLE OUTPUT***

Results:

# Elements    Run Time      Trials
20000        14828          20

*/
```

Appendix B – A Small Example with Modified Program

To prove out that the algorithm is working properly, I used the following arbitrary small set of points as an input:

(10, 7), (30, 12), (8, 1), (37, 18), (14, 15), (6, 40), (50, 20), (19, 2), (4, 50), (3, 20)

And the computed shortest distance between two points was: *6.32*, which is indeed the shorted distance – the distance between *(10, 7)* and *(8, 1)*, the closest pair of points in the set.

References

Introduction to Algorithms, 3rd edition, T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, The MIT Press, 2009. 1039-1043.

Algorithm Design, J. Kleinberg and E. Tardos, Addison Wesley, 2006. 225-231, 741-750.

The Design & Analysis of Algorithms, 2nd edition, A. Levitin, Addison Wesley, 2007. 108-109, 192-195.

Introduction to Algorithms: A Creative Approach, U. Manber, Addison Wesley, 1989. 278-281.

"Introspective Sorting and Selection Algorithms," *Software Practice and Experience*, D.R. Musser, 27(8):983-993, 1997.

"Divide and Conquer – Set 2 (Closest Pair of Points)," GeeksforGeeks, Web, http://www.geeksforgeeks.org/closest-pair-of-points, 2 Apr. 2016.