Yazan Halawa
2/3/16
CS 312 Sec. 1

Lab 2 Report

Lab Code:

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////// Helper Functions ////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
/**
  * Helper function to create a list of 2 or 3 points in counter clock wise
order
  * Time Complexity: O(n log n) where the worst case would be O(n^2) since it
the sort method uses Quick sort
  * Space Complexity: O(log n) as it uses Quick sort
  */
        public void connectPoints(ref List<System.Drawing.PointF> pointList)
        {
            // find the right most point
            PointF rightMostIter = pointList[locateRightMost(pointList)];

            // sort the list in counter clock wise order
            pointList.Sort(delegate (PointF first, PointF second)
            {
                if (first == rightMostIter) return -1;
                else if (second == rightMostIter) return 1;
                else
                {
                    double firstSlope = computeSlope(rightMostIter, first);
                    double secondSlope = computeSlope(rightMostIter, second);
                    return secondSlope.CompareTo(firstSlope);
                }
            });
        }

        /**
        * Helper function to locate the right most point in a list of points
and return its index
        * Time Complexity: O(n) as it iterates through the list one time
        * Space Complexity: O(n) as it creates a copy of the list passed by
value
        */
        int locateRightMost(List<System.Drawing.PointF> pointList)
        {
            int indexOfMax = 0;
            int counter = 0;
            float rightMostVal = -1000;
            foreach (PointF point in pointList)
            {
                if (point.X > rightMostVal)
```

```
                {
                    indexOfMax = counter;
                    rightMostVal = point.X;
                }
                counter++;
            }
            return indexOfMax;
        }

        /**
        * Helper function to locate the left most node in a list of points
and return it
        * Time Complexity: O(n) as it iterates through the list one time
        * Space Complexity: O(n) as it creates a copy of the list passed by
value

        */
        int locateLeftMost(List<System.Drawing.PointF> pointList)
        {
            int indexOfMin = 0;
            int counter = 0;
            float leftMostVal = 1000;
            foreach (PointF point in pointList)
            {
                if (point.X < leftMostVal)
                {
                    indexOfMin = counter;
                    leftMostVal = point.X;
                }
                counter++;
            }
            return indexOfMin;
        }

        /**
        * Helper function to computer slope between two points
        * Time Complexity: O(1) as it just does basic subtraction operation
that does not depend on the size of the input
        * Space Complexity: O(1) as it does not create any additional
variables that depend on the size of the input
        */
        double computeSlope(PointF first, PointF second)
        {
            double YDiff = second.Y - first.Y;
            double XDiff = second.X - first.X;
            return YDiff / XDiff;
        }


////////////////////////////////////////////////////////////////////////////
////////////////// Functions Responsible for Finding Tangents ///////////////
```

```
//////////////////////////////////////////////////////////////////////////////

/**
  * This function is to find the upper common tangent
  * Time Complexity: O(n + m) where n is the length of the first list and m
is the length of the second list, because the worst case is we iterate
through all the points in each list trying to find the upper tangent.
  * Space Complexity: O(1) as it doesn't create variables the depend on the
size of the input
  */
        void findUpperCommonTangent(ref PointF upperLeft, ref PointF
upperRight, List<System.Drawing.PointF> list_first_half,
                                    List<System.Drawing.PointF>
list_second_half)
        {
            // Define Flags and other variables
            bool upperTangentToBoth = false;
            bool upperTangentToLeft;
            bool upperTangentToRight;
            bool iteratedOnLeft;
            bool iteratedOnRight;
            double oldSlope;
            int indexOnLeftHullList = list_first_half.IndexOf(upperLeft);
            int indexOnRightHullList = list_second_half.IndexOf(upperRight);

            // Keep Going until you find the upper tangent
            while (!upperTangentToBoth)
            {
                // Setup the inital state of each iteration
                indexOnLeftHullList = list_first_half.IndexOf(upperLeft);
                indexOnRightHullList = list_second_half.IndexOf(upperRight);
                iteratedOnLeft = false;
                iteratedOnRight = false;
                upperTangentToLeft = false;
                upperTangentToRight = false;

                // Move counter clock wise on left hull
                oldSlope = computeSlope(upperLeft, upperRight);
                while (!upperTangentToLeft)
                {
                    indexOnLeftHullList++;
                    if (indexOnLeftHullList > list_first_half.Count - 1) //
Avoid out of range
                        indexOnLeftHullList = 0;
                    double newSlope =
computeSlope(list_first_half[indexOnLeftHullList], upperRight);
                    if (newSlope - oldSlope > 0) // slope is increasing
                    {
                        iteratedOnLeft = true;
                        // Move the iterator to the new point
                        upperLeft = list_first_half[indexOnLeftHullList];
```

```
                    oldSlope = newSlope;
                }
                else // we found the top point on left hull
                {
                    break;
                }
            }
            // Set the status bit of the left hull tangent to true
            upperTangentToLeft = true;

            // Move Clock wise on right hull
            oldSlope = computeSlope(upperRight, upperLeft);
            while (!upperTangentToRight)
            {

                indexOnRightHullList--;
                if (indexOnRightHullList < 0) // Avoid out of range
                    indexOnRightHullList = list_second_half.Count - 1;
                double newSlope =
computeSlope(list_second_half[indexOnRightHullList], upperLeft);
                if (newSlope - oldSlope < 0) // slope is decreasing
                {
                    iteratedOnRight = true;
                    // Move the iterator to the new point
                    upperRight = list_second_half[indexOnRightHullList];
                    oldSlope = newSlope;
                }
                else // we found the top point on right hull
                {
                    break;
                }
            }
            // Set the status bit for the right hull upper tangent to
true
            upperTangentToRight = true;

            // Break when both tangents are found
            if (upperTangentToLeft && upperTangentToRight &&
!iteratedOnLeft && !iteratedOnRight) // We have found the upper tangent
            {
                upperTangentToBoth = true;
            }
        }
    }




 /**
      * This function is to find the lower common tangent
```

```
      * Time Complexity: O(n + m) where n is the length of the first list and
m is the length of the second list, because the worst case is we iterate
through all the points in each list trying to find the upper tangent.
      * Space Complexity: O(1) as it doesn't create variables the depend on
the size of the input

       */
        void findLowerCommonTangent(ref PointF lowerLeft, ref PointF
lowerRight, List<System.Drawing.PointF> list_first_half,
                                    List<System.Drawing.PointF>
list_second_half)
        {
            // Create Flags to monitor the loop
            bool lowerTangentToBoth = false;
            bool lowerTangentToLeft;
            bool lowerTangentToRight;
            bool iteratedOnLeft;
            bool iteratedOnRight;
            double oldSlope;
            int indexOnLeftHullList = list_first_half.IndexOf(lowerLeft);
            int indexOnRightHullList = list_second_half.IndexOf(lowerRight);

            // Keep going until you find the lower tangent
            while (!lowerTangentToBoth)
            {
                // Setup the initial state of each iteration
                indexOnLeftHullList = list_first_half.IndexOf(lowerLeft);
                indexOnRightHullList = list_second_half.IndexOf(lowerRight);
                iteratedOnRight = false;
                iteratedOnLeft = false;
                lowerTangentToLeft = false;
                lowerTangentToRight = false;

                // Move clock wise on left hull
                oldSlope = computeSlope(lowerLeft, lowerRight);
                while (!lowerTangentToLeft)
                {
                    indexOnLeftHullList--;
                    if (indexOnLeftHullList < 0) // Avoid out of range
                        indexOnLeftHullList = list_first_half.Count - 1;
                    double newSlope =
computeSlope(list_first_half[indexOnLeftHullList], lowerRight);
                    if (newSlope - oldSlope < 0) // slope is decreasing
                    {
                        iteratedOnLeft = true;

                        // Move the iterator to the new point
                        lowerLeft = list_first_half[indexOnLeftHullList];
                        oldSlope = newSlope;
                    }
                    else // we found the top point on left hull
```

```
                    {
                        break;
                    }
                }
                // Setup the state bit for the left hull lower tangent to
true
                lowerTangentToLeft = true;

                // Move Clock wise on right hull
                oldSlope = computeSlope(lowerRight, lowerLeft);
                while (!lowerTangentToRight)
                {
                    indexOnRightHullList++;
                    if (indexOnRightHullList > list_second_half.Count - 1) //
Avoid out of range
                        indexOnRightHullList = 0;
                    double newSlope =
computeSlope(list_second_half[indexOnRightHullList], lowerLeft);
                    if (newSlope - oldSlope > 0) // slope is increasing
                    {
                        iteratedOnRight = true;
                        // Move the iterator to the new point
                        lowerRight = list_second_half[indexOnRightHullList];
                        oldSlope = newSlope;
                    }
                    else // we found the top point on right hull
                    {
                        break;
                    }
                }
                // Setup the state bit for the right hull lower tangent to
true
                lowerTangentToRight = true;

                // Break when both tangents are found
                if (lowerTangentToLeft && lowerTangentToRight &&
!iteratedOnLeft && !iteratedOnRight) // We have found the upper tangent
                {
                    lowerTangentToBoth = true;
                }
            }
        }
```

```
//////////////////////////////////////////////////////////////////////////////
//////////////////Main Divide and Conquer Algorithm ////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

```
        /**
        * Recursive function that uses divide and conquer approach to draw a
convex hull around a specific list of points
        * Time Complexity: Using the Master Theorem, since this algorithm has
two recursive calls each time and each recursive call takes half the input.
So a = 2, b = 2, and the combining work at the end is O(n) since the Tangent
algorithms take O(n+m) where n is the size of the first list and m is the
size of the second list = O(n/2 + n/2) = O(n) and creating the new list takes
O(n) as it just iterates through the points counter clock wise. So d = 1.
a/(b^d) = 2/(2^1) = 2/2 = 1 -> by the theorem, T(n) = O(n^d log_b n) = O(n^1 log_2
n) = O(n log_2 n)
        * Space Complexity: O(n) by the master theorem
        */
        public List<System.Drawing.PointF>
drawPoly(List<System.Drawing.PointF> pointList)
        {
            /////////////////////////// Base Case ////////////////////////
            if (pointList.Count < 4)
            {
                // Connect all the points to form the simplest convex hull
                connectPoints(ref pointList);
                return pointList;
            }
            else
            {
                ////////////////////// Recursive Steps /////////////////

                List<System.Drawing.PointF> list_first_half,
list_second_half;
                // find the middle index of the list
                int halfOfElements = pointList.Count / 2;
                int middleIndex = halfOfElements;

                list_first_half = drawPoly(pointList.GetRange(0,
halfOfElements)); // left half of points
                list_second_half = drawPoly(pointList.GetRange(middleIndex,
pointList.Count - halfOfElements)); // right half of points

                ////////////////// Connecting the two Shapes ///////////

                // first locate the right most point in the left list
                int indexOfRightMost = locateRightMost(list_first_half);

                // Next locate the left most point in the right list
                int indexOfLeftMost = locateLeftMost(list_second_half);


                // Set up some Variables
                PointF rightMostIter = list_first_half[indexOfRightMost];
                PointF leftMostIter = list_second_half[indexOfLeftMost];
                PointF upperLeft = rightMostIter;
```

```
                PointF upperRight = leftMostIter;
                PointF lowerLeft = rightMostIter;
                PointF lowerRight = leftMostIter;

                // find the upper common tangent and draw that line
                findUpperCommonTangent(ref upperLeft, ref upperRight,
list_first_half, list_second_half);

                // find the lower common tangent and draw that line
                findLowerCommonTangent(ref lowerLeft, ref lowerRight,
list_first_half, list_second_half);

                // Construct the new list
                List<System.Drawing.PointF> newList = new
List<System.Drawing.PointF>();

                // Move Counter clock wise from upper left to lower left
                int i = list_first_half.IndexOf(upperLeft);
                while (i != list_first_half.IndexOf(lowerLeft))
                {
                    newList.Add(list_first_half[i]);
                    i++;
                    if (i > list_first_half.Count - 1)
                        i = 0;
                }
                newList.Add(list_first_half[i]);

                // Move Counter clock wise from lower right to upper right
                i = list_second_half.IndexOf(lowerRight);
                while (i != list_second_half.IndexOf(upperRight))
                {
                    newList.Add(list_second_half[i]);
                    i++;
                    if (i > list_second_half.Count - 1)
                        i = 0;
                }
                newList.Add(list_second_half[i]);

                return newList;
            }
        }

/**
* This is the solve function that runs the whole code on the generated
pointList.
* Time Complexity: O(n log₂ n) as the dominant factor is the time of the
divide and conquer algorithm
* Space Complexity: O(n) As the dominant factor is the space of the divide
and conquer algorithm.
*/
        public void Solve(List<System.Drawing.PointF> pointList)
```

```
        {
            // First we sort the list of points in increasing x-values
            // Time Complexity: O(n log n)
            pointList.Sort(delegate (PointF first, PointF second)
            {
                return first.X.CompareTo(second.X);
            });

            // Acquire the list of points that form the convex hull
            List<System.Drawing.PointF> finalList = new
List<System.Drawing.PointF>();
            finalList = drawPoly(pointList);

            // Draw the lines connecting the points of the convex hull
            Pen redPen = new Pen(Color.Red, 3);
            for (int i = 0; i < finalList.Count; i++)
            {
                if (i != finalList.Count - 1)
                    g.DrawLine(redPen, finalList[i], finalList[i + 1]);
                else
                    g.DrawLine(redPen, finalList[i], finalList[0]);
            }
        }
```

Time and Space Theoretical Analysis:

The times and space consumptions are stated in the comment section above each function in my lab code. The explanation and the correlation to the master theorem is declared in the comment section above the main Divide and Conquer algorithm function drawpoly(List). Please refer to the lab code comments above.

Empirical Analysis:

Data Outcomes:

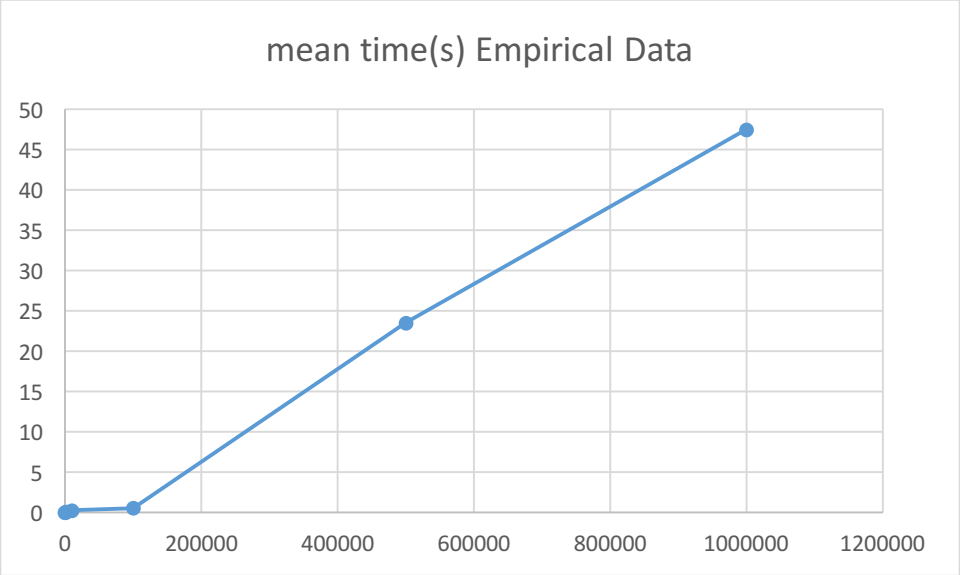
size(n)	Set 1 time(s)	Set 2 time(s)	Set 3 time(s)	Set 4 time(s)	Set 5 time(s)	mean time(s)
10	0.00289	0.0032	0.003	0.003	0.0031	0.00304
100	0.00392	0.00399	0.00467	0.00433	0.00391	0.00416
1000	0.0566	0.0565	0.0595	0.0602	0.0628	0.0591
10,000	0.268	0.263	0.2597	0.272	0.255	0.263
100,000	0.538	0.538	0.555	0.536	0.539	0.541
500,000	23.67	23.32	23.356	23.477	23.85	23.53
1,000,000	47.74	51.24	46.38	46	46.17	47.506

We notice that for small input the time is very small and efficient. However, for large inputs the time is bumped up significantly. Using our divide and conquer algorithm, this should not happen

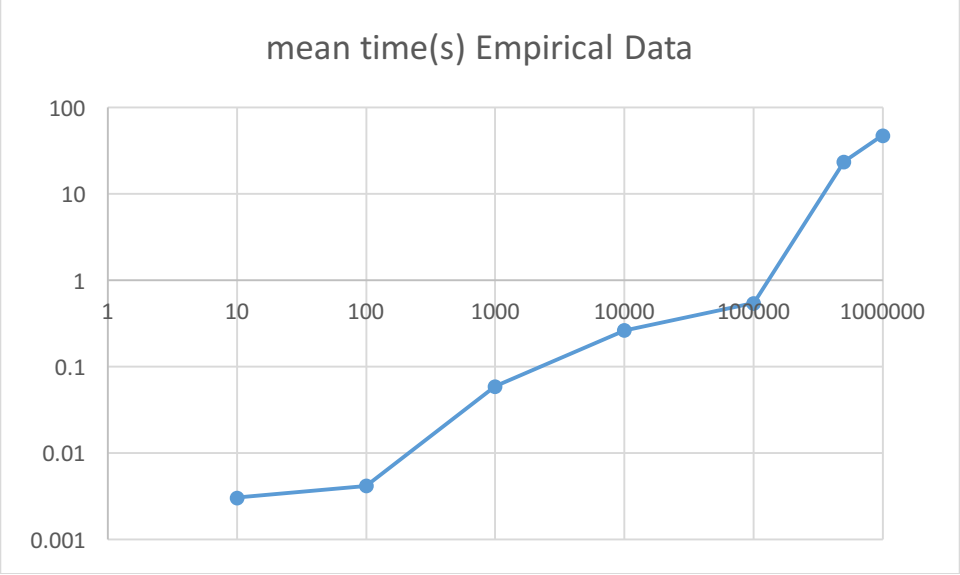
in that severity. To try and find the problem I looked through my code and checked whether I was doing something inefficient like sorting the points at each iteration, but I was not. I checked the functions and they all seem to be as efficient as they should be. Then I figured that probably the time was high because I was running the code on a virtual machine and so the computer would need some time to run through the code.

Plots:

Without any logarithmic Axis:



With Both Logarithmic Axis:



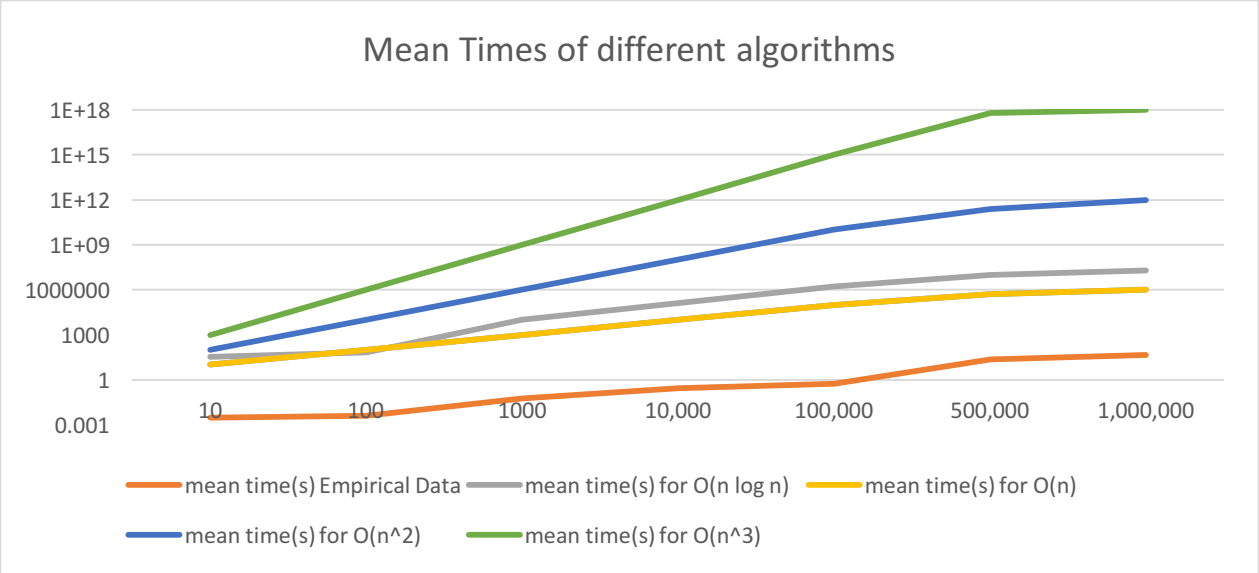
Explanation:

A Logarithmic scale is a nonlinear scale used for large ranges of data. As our size of input ranges from very small (10) to very large (1,000,000), it is hard to represent the data accurately in a linear graph. Using Logarithmic scaling makes sure the scale is based on the order of magnitude which does not have large variations. This makes it easier to analyze the graph and monitor the changes in all ranges of input.

We notice that the y values for small input are close together but then the differ much higher for large input. We first need to figure out which function or bound would fit this graph best. To do that I followed the approach of graphing several functions on excel and finding out which one most closely fits my data.

I constructed the following table with possible functions and plotted them:

Size (n)	Mean time (Empirical)	Mean time (O(n log n))	Mean time (O(n))	Mean time (O(n^2))	Mean time (O(n^3))
10	0.00304	33.22	10	100	1000
100	0.00416	66.44	100	10000	1000000
1000	0.0591	9966	1000	1000000	1000000000
10,000	0.263	132877.1	10,000	100,000,000	1,000,000,000,000
100,000	0.541	1660964	100,000	10,000,000,000	1,000,000,000,000,000
500,000	23.53	9465784.5	500,000	250,000,000,000	625,000,000,000,000,000
1,000,000	47.506	19931569	1,000,000	1,000,000,000,000	1,000,000,000,000,000,000



We notice that our empirical graph is not exactly linear so that takes out O(n), O(n^2) and O(n^3). Our graph mostly resembles O(n log n) which is what we expected from our theoretical analysis. However, there is a proportional constant that we need to find.

To Find this constant, I divided the time values I got from O(n log n) over the time values I got in my empirical data. The Following Were the Results:

size(n)	Results of Division
10	10927.63
100	15971.15
1000	168629.4
10,000	505,236
100,000	3,070,174
500,000	402,286
1,000,000	419,559

Then I averaged these values and got 656,111.883 = 6.56 * 10 ^ 5

So my data, let's call CH(Q), and the theoretical function g(n) = n log n relate to each other in this manner:

CH(Q) = k g(n) ➔ k = CH(Q) / g (n)

But from my results I got g(n) / CH(Q) = 6.56 * 10 ^ 5

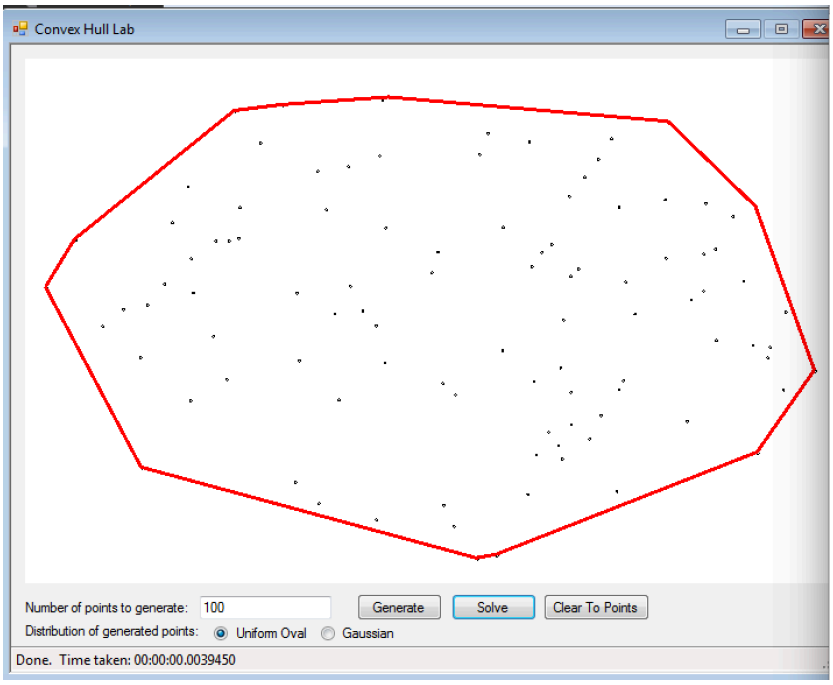
So g(n) / CH(Q) = 1 / (6.56*10^5) = 1.52 * 10^{-6}

So my constant of proportionality **k = 1.52 * 10^{-6}**

So CH(Q) = (1.56*10^{-6})* n log n

Screen Shots:

When n = 100:



When n = 1000: