Project #2 - Convex Hull

Code

```
class ConvexHullSolver(QObject):

    # Class constructor
    def __init__(self):
        super().__init__()
        self.pause = False


# Some helper methods that make calls to the GUI, allowing us to send updates
# to be displayed.

    def showTangent(self, line, color):
        self.view.addLines(line, color)
        if self.pause:
            time.sleep(PAUSE)

    def eraseTangent(self, line):
        self.view.clearLines(line)

    def blinkTangent(self, line, color):
        self.showTangent(line, color)
        self.eraseTangent(line)

    def showHull(self, polygon, color):
        self.view.addLines(polygon, color)
        if self.pause:
            time.sleep(PAUSE)

    def eraseHull(self, polygon):
        self.view.clearLines(polygon)

    def showText(self, text):
        self.view.displayStatusText(text)


# This is the method that gets called by the GUI and actually executes
# the finding of the hull
```

```
    def compute_hull(self, points, pause, view):
        self.pause = pause
        self.view = view
        assert(type(points) == list and type(points[0]) == QPointF)

        t1 = time.time()
        # TODO: SORT THE POINTS BY INCREASING X-VALUE
        sortedPoints = sorted(points, key=lambda point: point.x())

        t2 = time.time()

        t3 = time.time()
        # this is a dummy polygon of the first 3 unsorted points
        # polygon = [QLineF(points[i], points[(i+1) % 3]) for i in range(3)]
        polygon = self.convex_hull(
            self.divide_and_conquer(sortedPoints))
        # TODO: REPLACE THE LINE ABOVE WITH A CALL TO YOUR DIVIDE-AND-CONQUER CONVEX
HULL SOLVER
        t4 = time.time()

        # when passing lines to the display, pass a list of QLineF objects.  Each
QLineF
        # object can be created with two QPointF objects corresponding to the
endpoints
        self.showHull(polygon, GREEN)
        self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4-t3))

    # Transforms the list of points into a list of QlineF objects
    # This runs in O(n) for both time and space complexity as I'm generating a new
list
    # that will be used to create the lines
    def convex_hull(self, polygon):
        newPolygon = [QLineF(polygon[i], polygon[(i+1) % len(polygon)])
                      for i in range(len(polygon))]

        return newPolygon

    # This is the main helped function that splits the points list into its
corresponding left and right
    # subsets, which then are computed recursively
```

```
    # Time Complexity: O(nlogn) for the recursive call and then for merging them via
merge function
    # Space Complexity: O(n) as I'm generating new lists and later on merging them
    def divide_and_conquer(self, points):

        if len(points) == 1:
            return points

        L = points[:len(points)//2]
        R = points[len(points)//2:]

        leftHull = self.divide_and_conquer(L)  # O(nlogn)
        rightHull = self.divide_and_conquer(R)  # O(nlogn)

        # Base case just in case there is only 3 items, in which I'm just ordering
them manually
        if len(points) == 3:
            mergedHull = [0, 0, 0]
            mergedHull[0] = points[0]
            slope1 = (points[1].y() - points[0].y())/(points[1].x()-points[0].x())
            slope2 = ((points[2].y() - points[0].y())/(points[2].x()-points[0].x()))
            if slope2 > slope1:
                mergedHull[1] = points[2]
                mergedHull[2] = points[1]
            else:
                mergedHull[1] = points[1]
                mergedHull[2] = points[2]
            return mergedHull

        return self.merge(leftHull, rightHull)

    # This is the main function that takes care of correctly find the necessary
tangents
    # in order to merge and connect the hulls
    # Time Complexity: O(n)
    # Space Complexity: O(n)

    def merge(self, leftHull, rightHull):
        # This find the leftmost and rightmost points respectively, O(n)
        leftMostPoint = leftHull.index(max(leftHull, key=lambda leftPoint:
leftPoint.x()))
```

```
        rightMostPoint = rightHull.index(min(rightHull, key=lambda rightPoint:
rightPoint.x()))
        # The upper common tangent can be found by scanning around the left hull in a
        # counter-clockwise direction and around the right hull in a clockwise
direction.
        # Locate upper tangent, O(n)
        upperTangent = self.locate_upper_tangent(
            leftMostPoint, rightMostPoint, leftHull, rightHull)

        # Locate lower tangent, O(n)
        lowerTangent = self.locate_lower_tangent(
            leftMostPoint, rightMostPoint, leftHull, rightHull)

        # After finding the tangents, it's time to
        # This next section time complexity is O(n) and space O(n)
        mergedHull = []
        upperTangentLeft = upperTangent[0]
        upperTangentRight = upperTangent[1]
        lowerTangetLeft = lowerTangent[0]
        lowerTangetRight = lowerTangent[1]

        mergedHull.append(leftHull[lowerTangetLeft])
        # Move counter clockwise from lower left to upper left
        while lowerTangetLeft != upperTangentLeft:
            lowerTangetLeft = (lowerTangetLeft+1) % len(leftHull)
            mergedHull.append(leftHull[lowerTangetLeft])
        mergedHull.append(rightHull[upperTangentRight])
        # Move counter clockwise from upper right to lower right
        while upperTangentRight != lowerTangetRight:
            upperTangentRight = (upperTangentRight+1) % len(rightHull)
            mergedHull.append(rightHull[upperTangentRight])
        return mergedHull

    # Helper function to locate upper tangent - helps with debugging if the tangets
were wrong
    # Creates a slope and loops over slowly decreasing or increasing slope depending
    # which point I'm currently looking for
    # Time Complexity: O(nlogn)
    # Space Complexity: O(1)
    def locate_upper_tangent(self,  leftMostPoint, rightMostPoint, leftHull,
rightHull):
        l, r = leftMostPoint, rightMostPoint
```

```
        slope = (rightHull[r].y() - leftHull[l].y()) / (rightHull[r].x() -
leftHull[l].x())
        lturned = True
        while lturned:
            lturned = False
            while True:  # keeps it moving until the slope is no longer smaller
                tempSlope = (rightHull[r].y() - leftHull[(l-1) % len(leftHull)].y())/(
                    rightHull[r].x() - leftHull[(l-1) % len(leftHull)].x())
                if tempSlope < slope:  # If the slope is decreasing, it means that I
am moving to the left
                    slope = tempSlope
                    l = (l-1) % len(leftHull)
                    lturned = True
                else:
                    break
            while True:  # keeps it moving until the slope is no longer bigger
                tempSlope = (rightHull[(r+1) % len(rightHull)].y() -
leftHull[l].y())/(
                    rightHull[(r+1) % len(rightHull)].x() - leftHull[l].x())
                if tempSlope > slope:  # If the slope is increasing, it means that I
am moving to the right
                    slope = tempSlope
                    r = (r+1) % len(rightHull)
                    lturned = True
                else:
                    break

        return (l, r)

    # Helper function to locate lower tangent - helps with debugging if the tangets
were wrong
    # Creates a slope and loops over slowly decreasing or increasing slope depending
on
    # which point I'm currently looking for
    # Time Complexity: O(nlogn)
    # Space Complexity: O(1)
    def locate_lower_tangent(self, leftMostPoint, rightMostPoint, leftHull,
rightHull):
        l, r = leftMostPoint, rightMostPoint
        slope = (rightHull[r].y() - leftHull[l].y()) /
            (rightHull[r].x() - leftHull[l].x())
        lturned = True
```

```
        while rturned:
            rturned = False
            while True:  # keeps it moving until the slope is no longer smaller
                tempSlope = (rightHull[(r-1) % len(rightHull)].y() -
leftHull[l].y())/(
                    rightHull[(r-1) % len(rightHull)].x() - leftHull[l].x())
                if tempSlope < slope:  # If the slope is decreasing, it means that I
am moving to the right
                    slope = tempSlope
                    r = (r-1) % len(rightHull)
                    rturned = True
                else:
                    break
            while True:  # keeps it moving until the slope is no longer bigger
                tempSlope = (rightHull[r].y() - leftHull[(l+1) % len(leftHull)].y())/(
                    rightHull[r].x() - leftHull[(l+1) % len(leftHull)].x())
                if tempSlope > slope:  # If the slope is increasing, it means that I
am moving to the left
                    slope = tempSlope
                    l = (l+1) % len(leftHull)
                    rturned = True
                else:
                    break
        return (l, r)
```

Time and Space Complexity

`def convex_hull(self, polygon):` This helper method generates a new list of QlineF objects that can be passed in and then drawn. It has a Time Complexity: O(n) because we're iterating over the list of points that were merged, and Space Complexity: O(n) because it's generating a new list.

`def divide_and_conquer(self, points):` This is the main helper function that takes care of splitting the points into two hulls, and then computing them recursively while generating two lists - a leftHull and a rightHull. Later on, it returns the main function that merges both after locating the tangent points. This function has a Time Complexity: O(nlogn) because of the recursive call, and a Space Complexity: O(n) because of the new lists that are being created.

`def merge(self, leftHull, rightHull):` The merge function takes care of correctly identifying all the tangent points that need to be connected later on. It does this by initially locating the leftmost and rightmost points, and then locating all the upper and lower tangents in a clockwise fashion. After locating all the values, they are then stored in a brand new list with all the QpointF needed to draw the convex hull correctly. The function itself has multiple loop calls and helper method calls, but still runs on Time Complexity: O(n) and Space complexity O(n).

```
def locate_upper_tangent(self,  leftMostPoint, rightMostPoint, leftHull, rightHull):
def locate_lower_tangent(self,  leftMostPoint, rightMostPoint, leftHull, rightHull):
```

Both functions have the same functionality of locating the upper leftmost and rightmost tangent, and the lower leftmost and rightmost tangents respectively. It just simply moves until it locates both tangents and returns them, which will then be called in the main merge function. This operation takes in a Time complexity: O(nlogn) and Space complexity: O(1)

The total time complexity for the program is O(nlogn + n + n + n), which is simply O(nlogn).

Experiments and Discussion

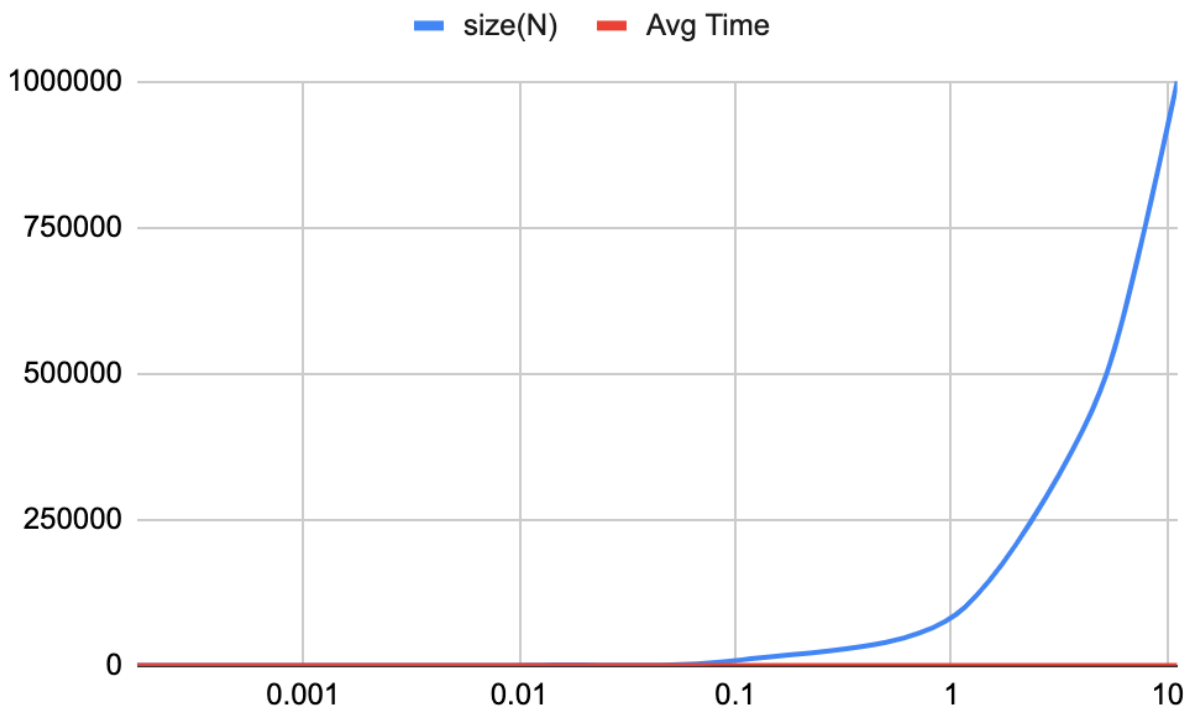
Experimental Outcome:

size(N)	Set 1 Time	Set 2 Time	Set 3 Time	Set 4 Time	Set 5 Time	Avg Time
10	0.000204	0.000160	0.000156	0.000152	0.000181	0.0001706
100	0.001519	0.001678	0.001617	0.001606	0.001531	0.0015902
1000	0.012887	0.012719	0.013522	0.012630	0.012502	0.012852
10,000	0.104205	0.104767	0.105981	0.111203	0.114946	0.1082204
100,000	1.114295	1.274708	1.131742	1.133951	1.120766	1.1550924
500,000	5.259169	5.289241	5.201114	5.229820	5.215472	5.2389632
1,000,000	11.782669	10.636783	10.924412	11.184227	10.902191	11.0860564

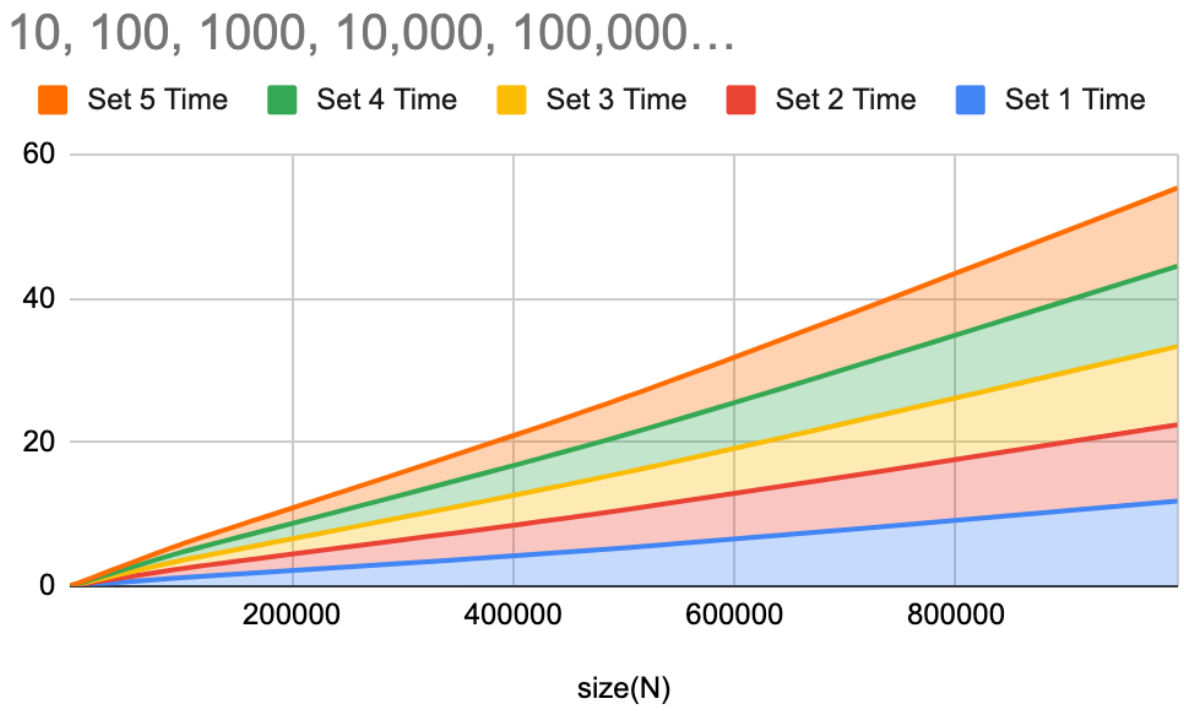
`

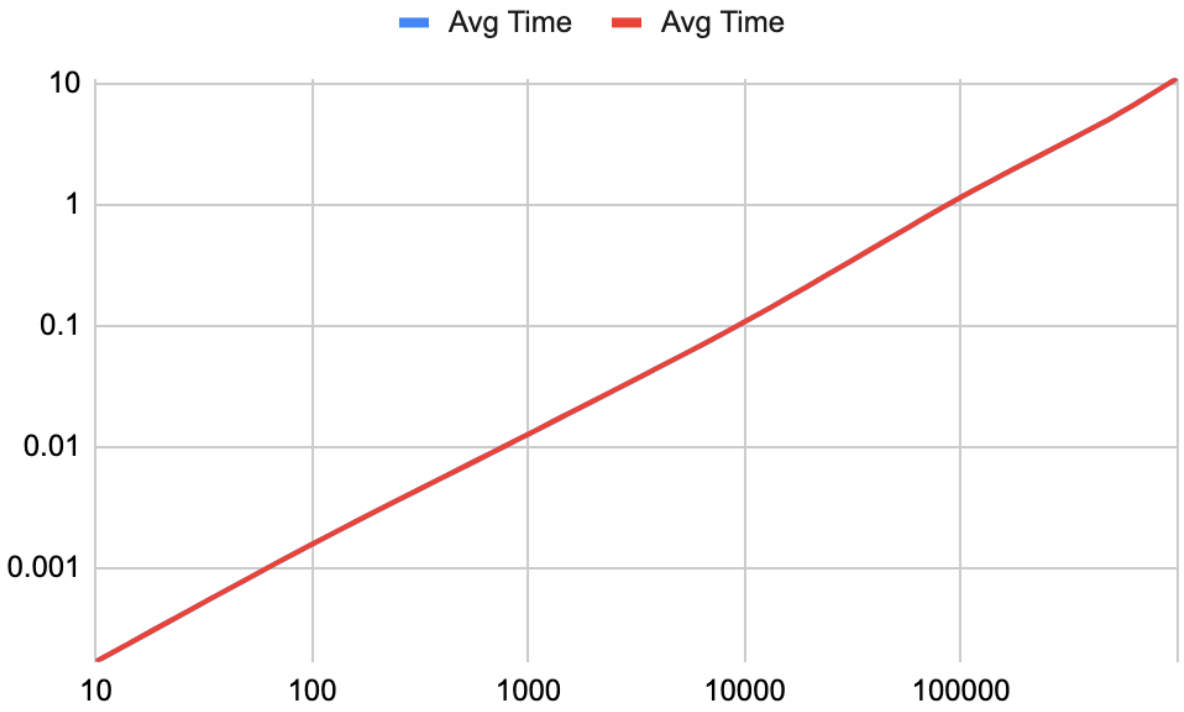
size(N)	Expected O(nlogn) time	Avg Time
10	33.22	0.0001706
100	66.44	0.0015902
1000	9966	0.012852
10,000	132877.12	0.1082204
100,000	1660964.04	1.1550924
500,000	9465784.28	5.2389632
1,000,000	19931568.56	11.0860564

Plots:



This is just a normal plot of seeing that as the size input increases, so does time





This is the plotted graph with seeing the time increasing as the size increases

Observations

From the results obtained after testing the different experiments, it seems that the program does a great job at somewhat matching a logarithmic growth. Initially there were some inefficiencies in my code that made the program take a lot longer than expected to run - mainly caused by an unnecessary sorting algorithm - which then I removed and ran the program at expected times with the varying sizes of data.

The first graph plotted with the time on the x-axis portrays what my analysis expected to see from this application. However, on the third graph, it looks more like a linear operation once plotted on a logarithmic scale for the vertical axis, which was a bit surprising. I think there might be a slight curve to it, which then makes it resemble more O(nlogn), which was the expected outcome of my theoretical analysis.

One thing that is very noticeable is the growth in time for each input, as it looks almost like for every 100,000 points added, it will add one extra second to our operations runtime. Maybe because the range of data is quite large, it kinda shows this linear progression, but I'm confident that if I plotted more intervals it would show more of the logarithmic scaling I was expecting to see.

Now, comparing the expected time with my average time, there are some obvious discrepancies. So, I needed to find the constant of proportionality. To find the constants, I divided the expected time over my average time values and the following table shows the result:

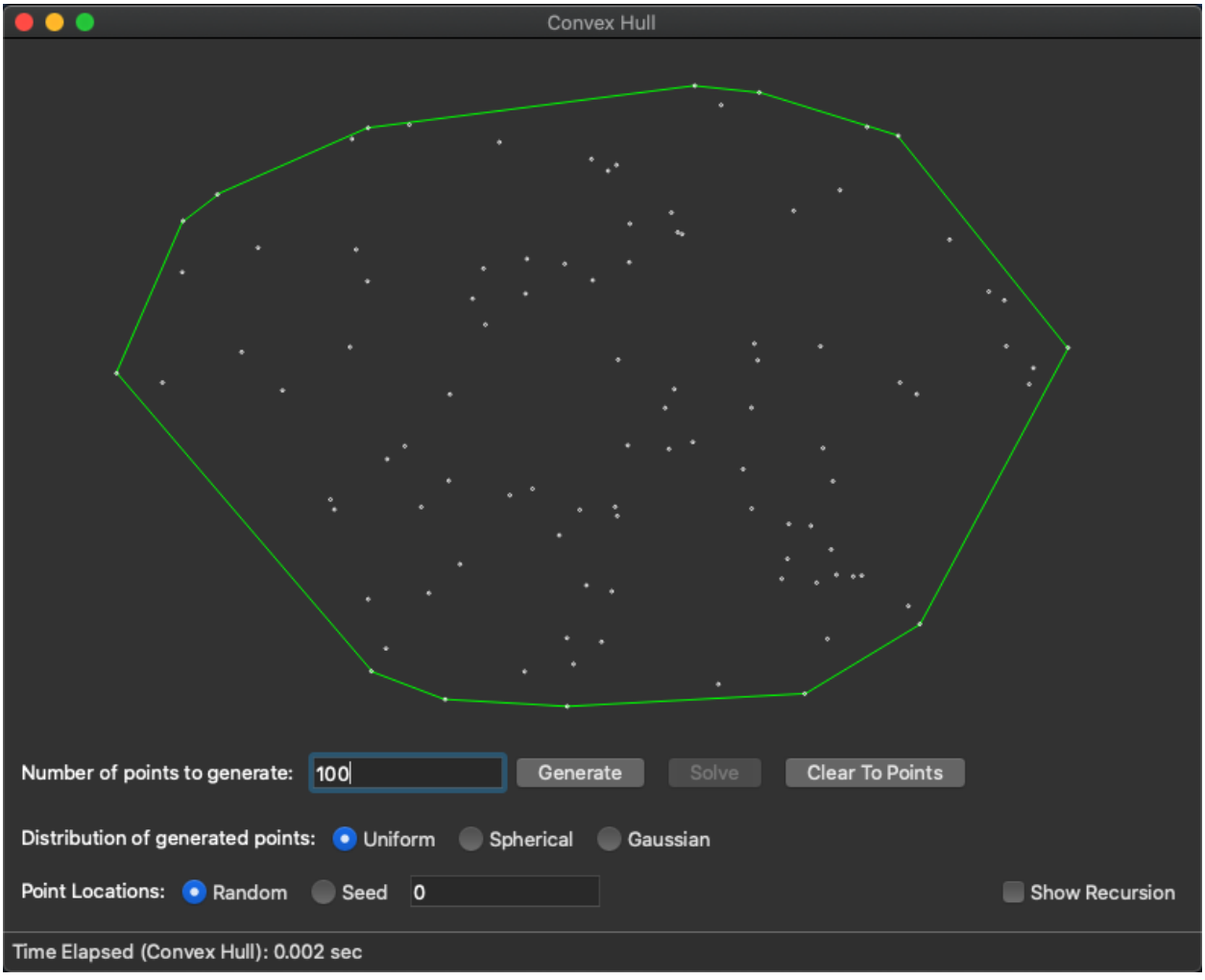
size(N)	Expected O(nlogn) time	Avg Time	Avg Constants
10	33.22	0.0001706	194724.5018
100	66.44	0.0015902	41780.90806
1000	9966	0.012852	775443.5107
10,000	132877.12	0.1082204	1227838.005
100,000	1660964.04	1.1550924	1437949.068
500,000	9465784.28	5.2389632	1806804.881
1,000,000	19931568.56	11.0860564	1797895.288

The average of the constants is 1040348.023. After inputting the value into a calculator, my official constant of proportionality k value estimated is around 9.61 * 10^-7.

Conclusion: CH(Q) = $(9.61 * 10^{-7}) * g(n)$, where g(n) represents O(nlogn).

Screenshots

Example with 100 points:



Example with 1000 points: