Gregory Knapp

CS 312

Dr. Grimsman

9/30/2021

Project 2 Convex Hull Solver Report

Time and Space Complexity Analysis of Project 2 – Breakdown by Subsection

The time complexity for each function found in convex_hull.py of the Project 2 code is as follows and pulled from the comments for each function in the code.

DNCHull(self, arr): Lines: 82 - 106

```
# This is the core divide and conquer algorithm of the project. It takes in
the full x-value sorted array from the
# main compute_hull function of size N, divides it by 2 over and over until
sub-arrays of size 5 or smaller are
# made. These 5 item or smaller arrays make up the base case for the function
and each sub-array's convex hull
# is found using the brute force convex hull algorithm below. When the array
does not meet the base case, it is
# split in half and the recursive DNCHull() call is made on the left and
right sub-arrays.
#
# Once the base cases are handled and the recursive calls are returned, the
function then passes the arrays of
# points for the convex hulls of the left and right sub-arrays to
mergeHulls(). mergeHull then processes these
# arrays using the upper and lower tangents to return an array of points that
make up the combined convex hull
# for the left and right sub-hulls. This result is then either returned up
the recursive stack or back to
# compute_hull() if it is the final call. The final result is the array of
points making up the convex hull of the
# entire set of points.
#
# The time complexity of this function comes down to the divide and conquer
nature. A more complete theoretical
# analysis of the Divide and Conquer function is found in the full report.
Since the function is breaking down
# the larger array of points into two sub-problems of N/2 size. This makes
the time complexity of the Divide and
# Conquer aspect of the algorithm O(log n).
#
```

```
# The time complexity is directly affected by the merging function since it
is considered post-work to the recursive
# part of the function. Time complexity for mergeHull() is O(N), which is
detailed below.
#
# This makes our final time complexity for the Divide and Conquer Convex Hull
Algorithm = O(N log N).
#
# Space complexity for this algorithm is ultimately O(N) as it has the one
array containing all of the points that
# is being split up and worked on by the various helper functions
```

mergeHulls(self, left, right) – Lines 129 – 151

```
# This is the function used to merge the left and right convex hulls either
found in the base case or
# returned by the previous recursive iteration of DNCHull(). The function
relies on two helper functions,
# getuppertangent() and getlowertangent(), which both return QLineF objects
containing the two points of either
# tangent line for the combined hulls.
#
# Once the tangent lines are solved, mergeHulls performs one last combining.
Because at least one of the upper or
# lower tangent lines will move from the starting line made from the right-
most point of the left hull and the
# left-most point of the right hull, points need to be excluded from the
combined hull array while still
# maintaining clean clockwise order to make the recursive merging work. This
is accomplished by doing a full
# clockwise traversal of the two hulls, adding points to the result array one
by one. Points that should be dropped
# are identified using p1 and p2 of the upper and lower tangent lines as
reference. These points are not skipped in
# iteration, they are just ignored and not added to the result array. The
clockwise ordered result array is then
# returned and is ready to be recursively merged again, or returned as the
final result.
#
# The time complexity for this function is O(N) because it needs to iterate
over each point during the merging
# process when it calls on the get upper and lower tangent helper functions.
There is iteration over the points
# again at the end to maintain the clockwise ordering of the points, but this
only makes the time complexity for
# the function O(2N) which can just be simplified down to O(N)
#
# Space complexity for this functions is O(N^2), as we have the two sub-hull
arrays, each of size O(N/2) or a
# combined size of O(N) and the result array. While the result array does not
include every point from the original
# arrays, the merge usually only results in a few points being lost between
the sub-arrays and the combined one, so
# I say it is fair to consider result equal in size to the two sub-arrays,
resulting in O(N^2) space being used.
```

BruteForce(self, arr) – Lines 200 – 233

```
# Brute force algorithm used only to calculate the convex hull of the base case arrays. This algorithm iterates
# through x-sorted array of points over a loop and two inner loops. The inner loop checks each possible line made by
# two points in the hull against every point in the hull, skipping cases where the point to check is the same as
# one of the points in the line. If every remaining point in the hull is one the same side of the line, above
# or below, both points in the line are added to the hullpoints array, as long as they do not already exist in
# the array.
#
# Points being above or below the line is evaluated using the relationship ax + by = c, where a = y2 - y1,
# b = x1 - x2, and c = (x1*y2) - (x2*y1). Using a and b calculated from pointone and pointtwo, the point to test
# arr[k] is evaluated. If c > a*arr[k].x()  + b*arr[k].y(), then the point was above the line.
# If c < a*arr[k].x()  + b*arr[k].y(), then the point was below. Points above the line are counted. After all
# iteration of the k inner loop, if count == the number of points tested then all points were above the line,
# and the two points in the line were added to the hullpoints array. If count == 0, then all points were
# below the line and the points from the line were also in the array and were added.
#
# Once the points are found, they are passed to a helper function to order them in clockwise order in order to
# simplify the all subsequent merging. The clockwise order is maintained carefully by the merging function, so the
# helper function is only used in the context of the brute force algorithm.
#
# Simply put, the brute force hull algorithm has a time complexity of O(N^3), since it iterates using three for
# loops over the same array of points to test each possible combination. The function is slightly more optimized
# since it skips any repeat points before doing any calculation. This does not change the overall complexity.
#
# That being said, it is important to consider how this function is being used. It is only used in solving the
# convex hulls of the bases cases which are all of size <= 5, which is very different from the original N points,
# at least once the number of points gets larger. Because of this, it is probably more fair to say that the time
# complexity of the brute force algorithm is more like O(s) where little s is the size of the input array which are
# all of size <= 5. Because it is only being use on the smaller base case arrays, it does not affect the overall
# time complexity of the at-large divide and conquer algorithm.
#
# Space complexity for this function is O(s) (where s is the size <= 5). At most, the hullpoints result array can
# contain all of the 5 or fewer points in the input array, and one at worst
```

```
(though it is more likely to be 2 or 3
# in the smallest cases). Because the difference between 1 and 5 is
insignificant, the space complexity can just be
# called O(s).
```

clockwisesort(self, arr) – Lines 286 – 319

```
# A helper function used to ensure that the points in the base case convex
hulls are ordered in clockwise order to
# use in a circular array for merging. Input to the brute force algorithm
comes in as an array of points sorted by
# x-values, but it does not guarantee that points are in clockwise order.
This function changes the array from
# x-sorted to clockwise ordered.
#
# The further left point in the array is chosen as the de facto starting
point for the clockwise order. From there,
# the coordinates of the center point of the hull array is calculated by
summing the x and y values of
# the array and dividing the the two results by the length of the array. Once
the center point is found, a line
# is drawn between the starting point and the center array.
#
# While looping through each point in the hull arrays, a second line is drawn
between each point in the array and
# the center point. Using the built in QLineF function QLineF.angleTo(line),
the angle in degrees between the
# starting point and the second line is calculated. This angle in degrees is
saved to a separate array as a duple
# paired with its index in the hull point array.
#
# Once the angle has been calculated, the angle-index duple array is sorted
by increasing angle in degrees
# using a lambda function as key for python's list.sort() function, to sort
by increasing angle. This works
# because the lowest angle from the starting line would be the next point in
clockwise order. By iterating one
# more time through the array of angles and indices, the point from the
original array is added using the index
# from the duple array, and finally added to the clockwiseorder point array
and returned
#
# Similar to the base case brute force algorithm above, this function only
deals with arrays of size s (size <= 5).
# Because of this, time complexity calculations are simplified. This function
loops through the values of the array
# three times: Once to get the sum of x and y values to find the center, a
second time to get the angles of each
# line from the center, and a third to add the points in the correct order to
the result array. This would normally
# be a time complexity of O(3N), but using the guaranteed small array size I
have designated as s, this time
# complexity would be O(3s) which is just O(s).
#
# Same as the time complexity, the space complexity only deals with a largest
size of s for the result array.
```

```
# Since this is a type of sorting algorithm, the result array will have the
exact size of the input array, just in
# a different order. We do create a second array of duples that is used to
store angle and index values, which would
# be of the same length, just with a second column of values per row. In
total, we essentially have 3 arrays of size
# s created and used during this function plus the input array of O(s0,
# so our space complexity could be labeled O(4s) or just O(s).
# three times: Once to get the sum of x and y values to find the center, a
second time to get the angles of each
# line from the center, and a third to add the points in the correct order to
the result array. This would normally
# be a time complexity of O(N^3), but using the guaranteed small array size I
have designated as s, this time
# complexity would be O(s^3)
#
# Same as the time complexity, the space complexity only deals with a largest
size of s for the result array.
# Since this is a type of sorting algorithm, the result array will have the
exact size of the input array, just in
# a different order. We do create a second array of duples that is used to
store angle and index values, which would
# be of the same length, just with a second column of values per row. In
total, we essentially have 3 arrays of size
# s created and used during this function plus the input array of O(s0,
# so our space complexity could be labeled O(4s) or just O(s).
```

getRightMostX(self, arr) – Lines 369 – 390

```
# A helper function used in getuppertangent() and getlowertangent() in order
to find the starting value of the
# left side array for merging. Finding the left most x-value in the right
side array is trivial, since it was
# purposefully added first to the clockwise sorted arrays, it can be
extracted directly. However, the rightmost
# x-value which is used to make the starting line for testing for the convex
hull tangent lines is no longer found
# at the final value of the left points array after the clockwise sort of the
base case hulls.
#
# This helper function is just a simple for loop through the left array to
find the max x-value and return the index
# it is found at. The code is separated into the helper function to reduce
duplicate code since it is used by
# both the upper and lower tangent functions.
#
# This is a simple looping function meant to identify the max x value from an
array. The arrays used in this
# can be as small as s to N/2, since this function is only called on the left
sub-hull of the hulls to be merged.
# In the context of this function, the time complexity could be said to be
O(N) since it is a for loop over the
# entire array of points. However, in the larger picture of the overall
function, it acts as more of a O(1) constant
# (more accurately O(N/2) time function, since it never iterates over the
entire array of N points, even on the
```

```
# final step of recursion. Because of this, it does not impact the overall
time complexity of the divide and
# conquer
#
# The space complexity of this function is similarly dependant on the input
size, but has a max size of O(N/2)
# where N/2 is the size of the largest array that will be passed into it.
Similar to time complexity, in the
# context of the function, the space complexity if O(N), however, in the
bigger picture it is more accurately
# O(N/2) which is more similar to O(1).
```

getuppertangent(self, left, right) – Lines 407 – 443

```
# A helper function for mergehulls() using the left and right sub-hull arrays
from each recursive call.
# The implementation is done with a larger while loop that contains two while
loops; the first one iterates until
# the tangent line of the left array is found and the second one iterates
through until the tangent line of the
# right array is found. The outer while loop will only stop when both inner
while loops do not complete a full
# iteration, indicating that the tangent line connecting both hulls is the
tangent line of each individual sub-hull,
# and as a result is the upper tangent of the combined hull.
#
# The implementation uses a stepping pattern to check if the current given
line is a tangent line of the sub-hulls.
# The starting line, tangentline, is drawn from the rightmost point of the
left array and the left most point of the
# right array. The outer while loop start and sets its own conditional
variable to true. In the first while loop,
# the next counter-clockwise point in the circular array is saved. The slope
of the original tangent line and the
# line using the same point from the right array and the next counter-
clockwise point is saved. The loop
# then checks if the slope of the current tangent line < the line using the
next point. If this condition
# is true, the while loop breaks, since the current tangentline is a
tangentline of the left array. If it is false,
# tangentline is updated to the next point, the outer loop conditional
variable is set to false, and the first inner
# loop continues.
#
# The same process happens for the second inner loop on the right sub-hull,
except it uses the next clockwise point
# and checks for the currentslope > slope of line using the next clockwise
point for the loop break condition.
# This conditional checking creates the stepping action for both side. When
the outer while loop traverse both
# inner loops without either of them moving the tangent line, then the line
is a tangent of both sub-hulls and the
# combined hull as a result.
#
# The time complexity of this function changes with each recursive call, as
the input arrays get larger and larger
```

```
# approach size of N between the two arrays. During the final recursive call, the two hulls to combine will be
# of size N points, however, traversal of the upper tangent does not take O(N) time to complete, since the furthest
# the tangent line could be would be about on the other side from the most inside points, and often is much shorter
# than that. As a result, the time complexity is more accurately estimated in a worst case scenario to be O(N/2),
# which is more similar to a constant O(1) time. In conjunction with getlowertangent(), the time complexity for the
# two tangent functions together could be estimated at O(N) (which I have done in the larger time complexity
# analysis of the entire divide and conquer) since it is possible to traverse the almost the entire array in search
# of the tangent lines depending on the spread of points. Individually however, it is more like O(N/2) time.
#
# Space complexity for this function is more straightforward. The final recursive merges will have hulls containing
# essentially the full set of points to work with or O(N) points between the left and right sub-hull arrays.
# Other than the input arrays, this function works in constant space variables, so the O(N) is the total size
# complexity for this function.
```

getlowertangent(self, left, right) – Lines 524 – 543

```
# A helper function for mergeHulls parallel to getuppertangent(). This function takes the same left and right
# sub-hulls as getuppertangent and performs similar calculations on them, this time to get the lower tangent line of
# the combined hull.
#
# The two functions are essentially identical with 4 differences.
# 1) The first inner while loop (traversing the left sub-hull) now moves in a clockwise direction.
# 2) The first inner while loop now checks for decreasing slope to know when to break the loop
# 3) The second inner while loop (traversing the right sub-hull) now moves in a counter-clockwise direction.
# 4) The second inner while loop now checks for increasing slope as its condition to break the loop
#
# In essence, getting the lower tangent is the same functionally, only reversing the movement and loop conditions
# of the upper tangent function. Because of the similarities, the specific workings of the functions won't be
# repeated again here.
#
# The justification for the time and space complexity of this function is the same as for getuppertangent, so it
# will not be duplicated here.
# The time complexity for getlowertangent in isolation is O(N/2), O(N) when considered together
# with getuppertangent.
#
```

```
# The space complexity for getlowertangent is O(N) for the combined size of
the two input arrays.
```

Theoretical Analysis of Project 2 Code

As depicted in the subsection time and space complexity breakdown of the code above, the overall time complexity of the Convex Hull Solver is dictated by two main parts: The divide and conquer function it self and the mergeHulls function called at the end of the divide and conquer.

The mergeHulls function is easy enough to evaluate complexity wise; the function iterates through N values across the two arrays in order to make the merge happen, giving us a time complexity of O(N) time. This value is our post-work time complexity in our divide and conquer algorithm.

Referring to the divide and conquer function, because we know our post-work time complexity, we can take a look the actual time complexity of the function itself. The DNCHull() function takes the array of size N and divides it into two sub problems, each one of size N/2. We now know that in the Master theorem, our a = 2, b = 2 and d = 1 because for O(N^d) = O(N), d must equal 1.

Plugging this into for our recurrence relation, we get the following:

T(n) = 2T(N/2) + O(N)

If we then evaluate our Master theorem relationship for a, b, and d, we get the following:

a/(b^d) = 2/2^1 = 1 -> 1 == 1 Therefore running time should be O(N^d log N).

Know that our d = 1, we get the expected running time of O(N log N), which matches up with the comments in the code found in the first section of this report.

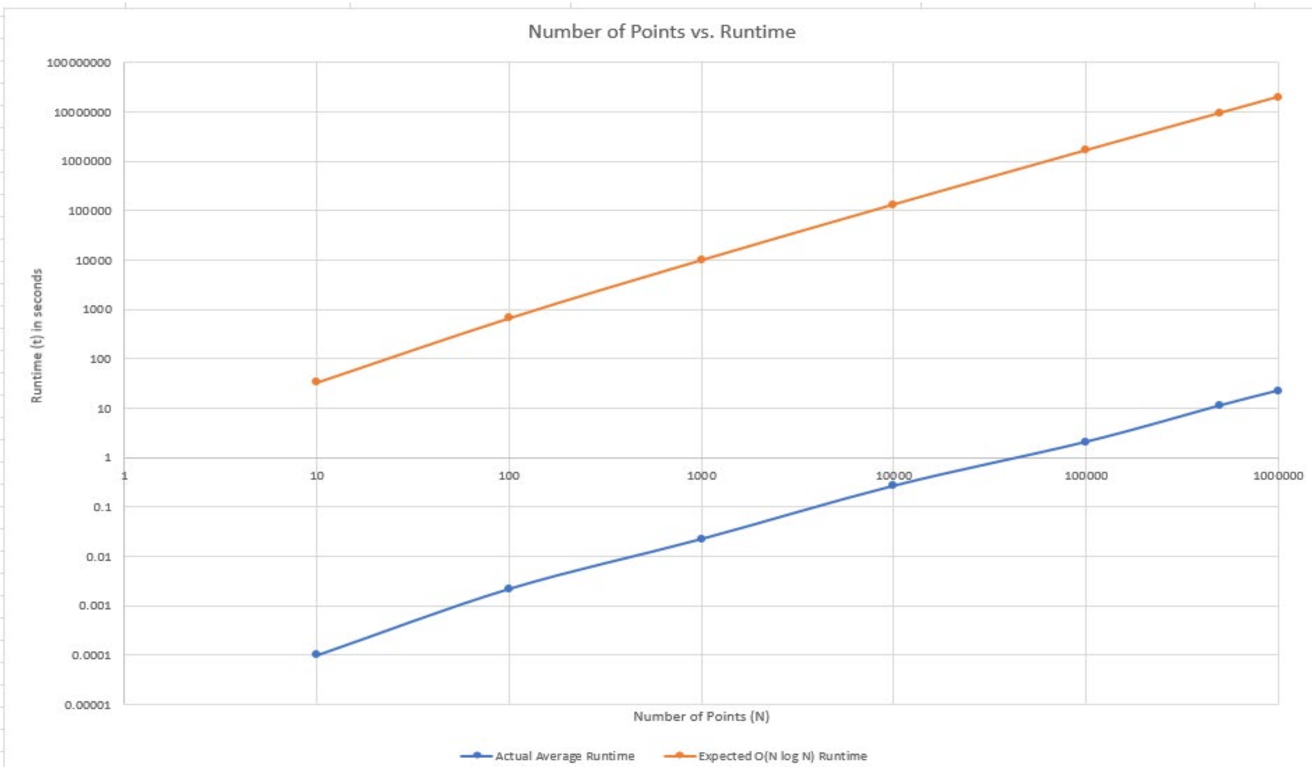
Empirical Analysis of Project 2 Code

The following table is the results of my empirical testing on the Convex Hull Solver

N # of Points	Test 1	Test 2	Test 3
10	0.0001	0.0001	0.0001
100	0.002	0.002	0.003
1000	0.022	0.023	0.023
10000	0.266	0.266	0.266
100000	2.047	2.091	2.075
500000	11.201	11.208	11.305
1000000	22.636	22.378	22.46

Test 4	Test 5	Average Runtime	Expected O(N log N)
0.0001	0.0001	0.0001	33.21928095
0.002	0.002	0.0022	664.385619
0.022	0.022	0.0224	9965.784285
0.267	0.27	0.267	132877.1238
2.061	2.044	2.0636	1660964.047
11.207	11.323	11.2488	9465784.285
22.427	22.505	22.4812	19931568.57

From this table, I created the following graph of points (N) against runtime (t) with logarithmic axis. The blue curve is the actual runtimes from my trial and the orange curve is the expected run times assuming the theoretical analysis runtime of O(N log N).



As is clear in the graph, the two curves, although similar in shape and curvature, are not even close in terms of time for each value of N. Because of the similarities in shape, I did not think that there was a mistake in the run time being g(n) = O(N log N) for the convex hull algorithm, so I decided to look into constants of proportionality k that could make the CH(P) = k * g(n).

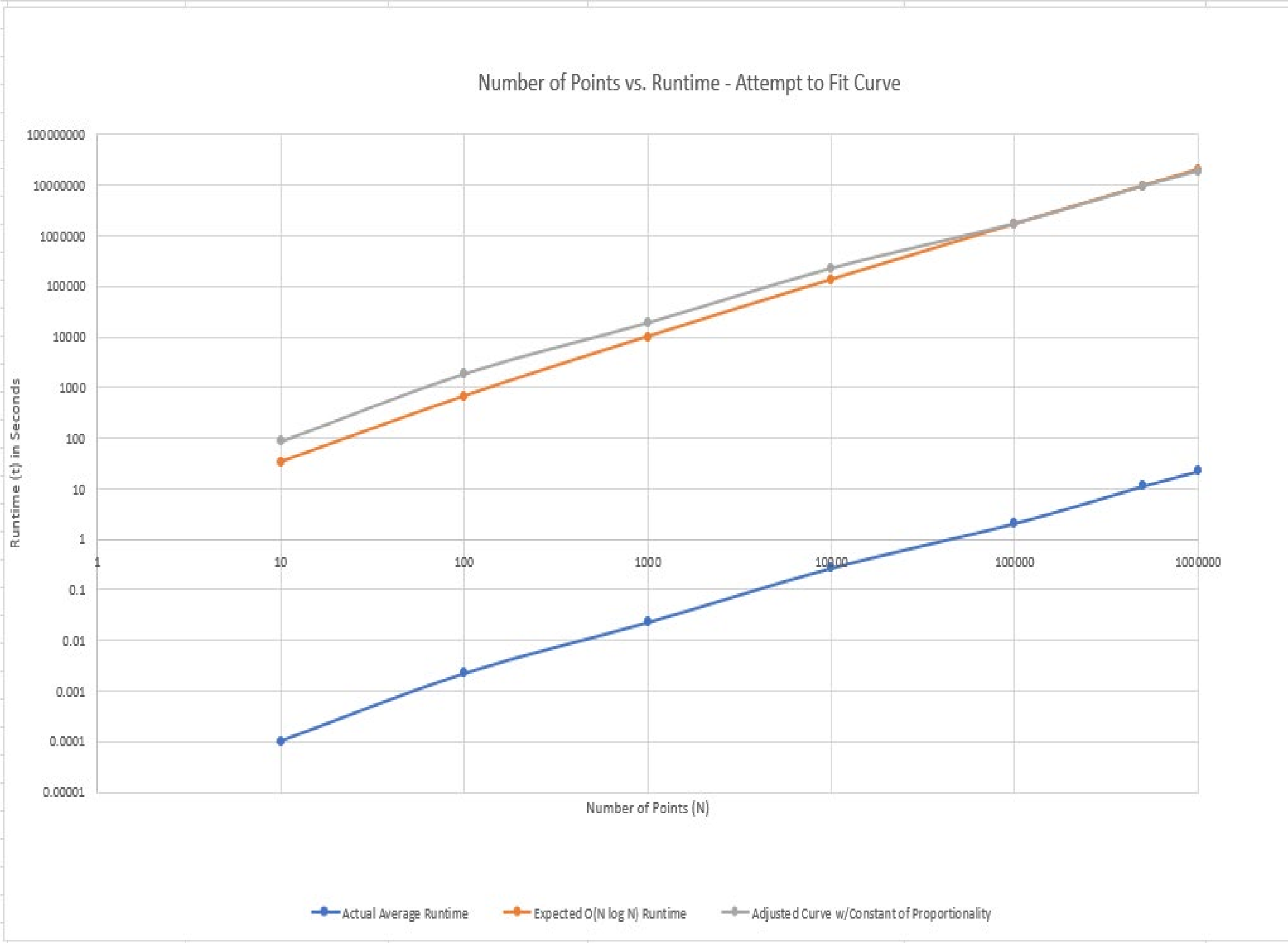
Based on the equation CH(P) = k * g(n), I used my actual and expected values to add a column in my table for k as seen below.

N # of Points	Actual Average Runtime	Expected O(N log N) Runtime
10	0.0001	33.21928095
100	0.0022	664.395619
1000	0.0224	9965.784285
10000	0.267	132877.1238
100000	2.0636	1660964.047
500000	11.2488	9465784.258
1000000	22.4812	19931568.57

Adjusted Curve w/Constant of Proportionality	Constant of proportionality k = CH(Q) / g(N)
84.14928044	3.0103E-06
1851.28417	3.31128E-06
18849.43882	2.24769E-06
224678.5788	2.00938E-06
1736504.551	1.24241E-06
9465784.258	1.18836E-06
18917768.03	1.12792E-06

The final column "Constant of proportionality k = CH(Q) / g(N)" is the result for each row that I calculated k for. The second to final column is the values for each value of N when I applied the constant that I chose for my graph.

Thinking that Big O notation is concerned with the behavior of functions in relation to one another when input numbers get extremely large, I applied the second to last value of k that I got when I divided 11.2488 / 9465784.258 which was 1.18836e^-6, resulting in the values in the second to last column. When I plotted these new values along with the actual and expected curves, I got the following graph.



The gray line is the adjusted curve with the constant of proportionality while the orange and blue curves are the same as before. With 1.18836e^-6 = k applied to the curve, we see that the beginning of the curve does stretch above the expected O(N log N) values for the graph but comes down below it and settles in right below O(N log N) at around N = 1,000,000 and continues that way, so that g(n) does act like an upper bound for our adjusted curve.

Theoretical vs. Empirical Analysis - Conclusions

With the analysis above, I still feel comfortable concluding that although the actual and expected values from the theoretical and empirical analysis were very different, the runtime for the Convex Hull algorithm should be O(N log N).

The similarity in the shape of the two curves over the large data points initially made me think that it was not a different type of function, an O(N) curve would grow at an entirely different rate, and O(log N) curve would be much less steep in terms of growth rate. This made me think that it is more likely to be a case of finding the right constant of proportionality such that g(n) = O(N log N) would be an upper bound for CH(P).

I will admit, I am not familiar with curve fitting software or coding packages for python or any other language, so all of this work was done just with Excel and from observation and based on what was mentioned in the lab specs, so it is in no way the correct constant of proportion. However, the graph behavior with k = 1.18835e^-6 added in does look much more like the relationship we would expect between the two functions, so I feel comfortable with it as base analysis for the results.

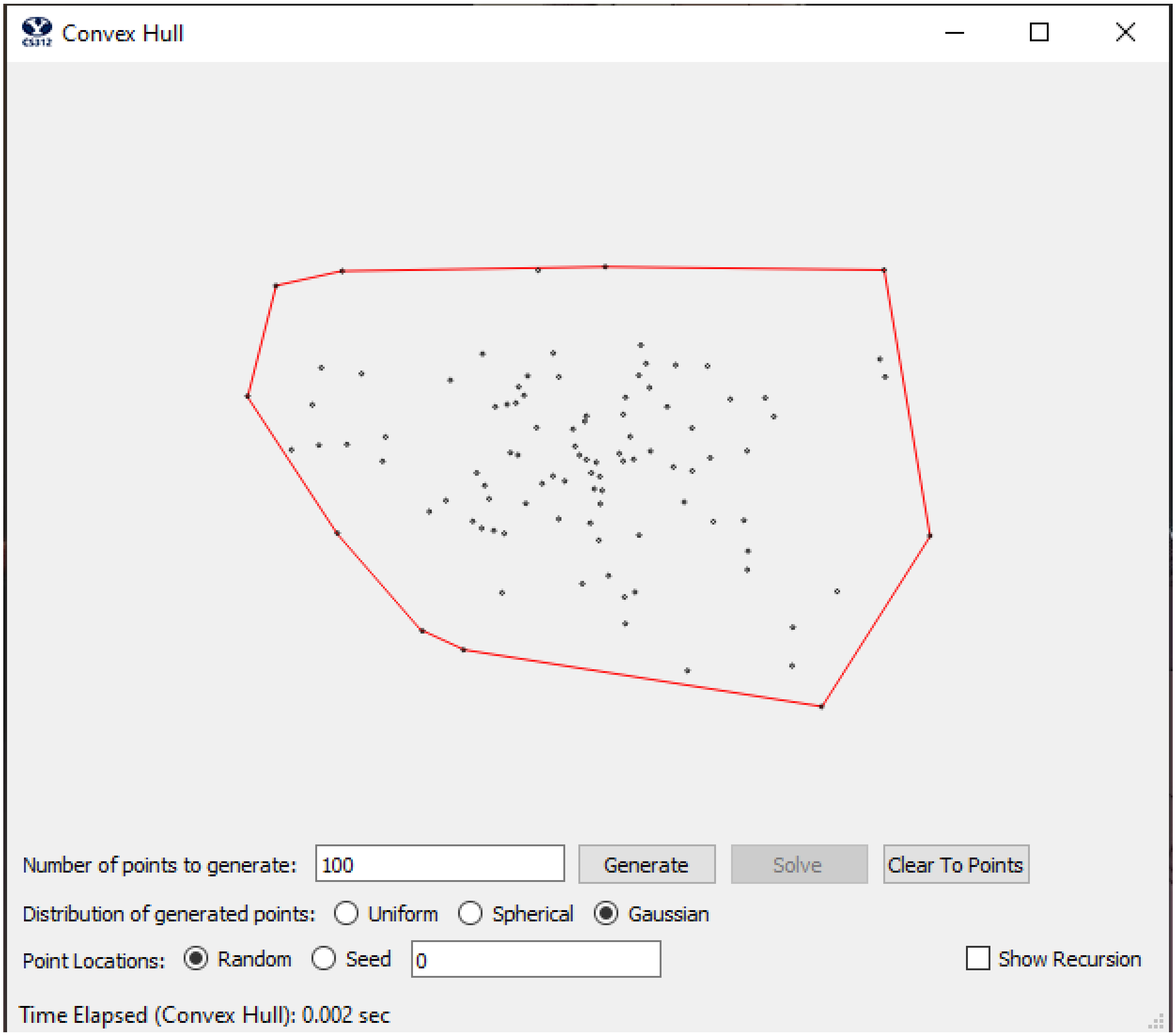
As for why the differences between the theoretical and empirical analysis came to be, it is hard to say. The first thing that comes to my mind is hardware power. It is hard to assume what speed processes are handled at when doing theoretical analysis, and this can only be evaluated by running empirical testing. Just from keeping up with the slack channel, I've seen students run the 1,000,000 test as low as nearly 10 seconds and over 200 seconds. I fall on the more average spectrum of about 22 seconds, which is still a huge difference from 200 seconds, so hardware certainly had an impact on the differences.

The biggest source for the difference in my mind comes from the N part of O(N log N). Assuming our time is in seconds, saying O(N) runtime for an algorithm would mean that for each one piece of input, we are taking 1 seconds to process it. In my algorithm, the O(N) part of O(N log N) stemmed from the mergeHulls() function that was called in the recursive divide and conquer function. Theoretically, I labeled it O(N) because it is iterating two different not nested times over the array of points in the hull. This should be the main time eating part of the algorithm, since O(N) is significantly slower than O(log N), however to the computer, O(N) processes really are trivial things. Until we reach exponential time, computers handle things in constant, linear, and lower polynomial time so easily that the O(N) functions really do seem more like constant function (even if they are not in reality and shouldn't be assumed to be so).

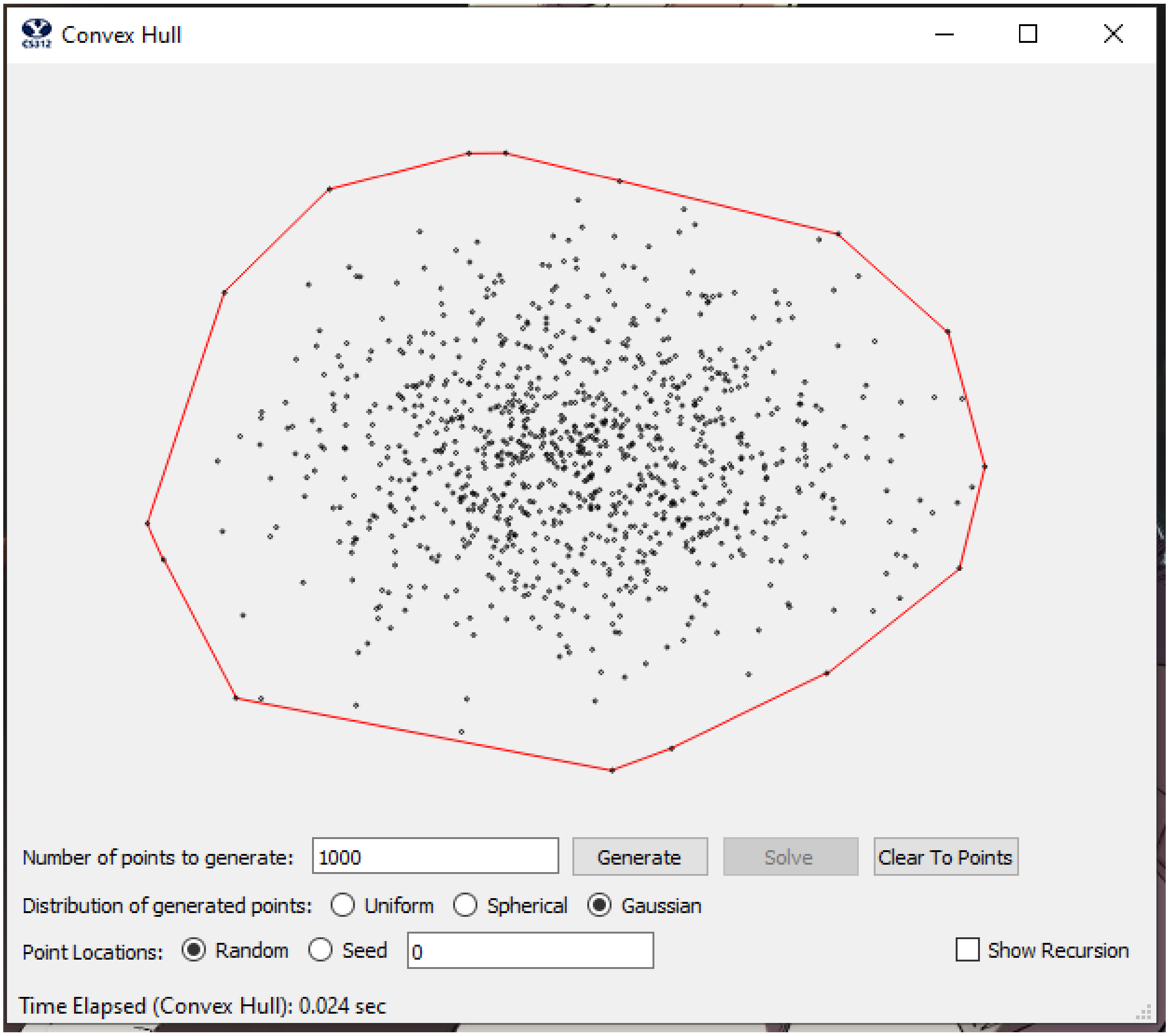
I think the discrepancy between expected time to run and actual time to run in the O(N) part of my algorithm would cause the large gap between expected runtime and actual runtime.

Working Screenshots

Screenshot #1 – Convex Hull Solver ran with 100 Points



Screenshot #2 – Convex Hull Solver with 1000 points


Convex Hull
Number of points to generate:
1000
Generate
Solve
Clear To Points
Distribution of generated points:
Uniform
Spherical
Gaussian
Point Locations:
Random
Seed
0
Show Recursion
Time Elapsed (Convex Hull): 0.024 sec

Source Code Files

convex_hull.py code file

```
from which_pyqt import PYQT_VER

if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF, QObject
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import time

# Some global color constants that might be useful
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
BLACK = (0, 0, 0)

# Global variable that controls the speed of the recursion automation, in
seconds
#
PAUSE = 0.25


#
# This is the class you have to complete.
#
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)

        # Sort points using Python List.sort()
        points.sort(key=QPointF.x)  # Time Complexity for Python's sort is
O(n log n) according to documentation

        t3 = time.time()

        # Call Divide and Conquer
        finalhullpoints = self.DNCHull(points)  # The main Divide and Conquer
function Time Complexity: O(N^2 log N)

        # Turn points from list into array of lines to draw on GUI
        finalhull = [QLineF(finalhullpoints[i], finalhullpoints[(i + 1) %
len(finalhullpoints)]) for i in
                     range(len(finalhullpoints))]

        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(finalhull, RED)
        self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 -
t3))

    # This is the core divide and conquer algorithm of the project. It takes
in the full x-value sorted array from the
    # main compute_hull function of size N, divides it by 2 over and over
until sub-arrays of size 5 or smaller are
    # made. These 5 item or smaller arrays make up the base case for the
function and each sub-array's convex hull
    # is found using the brute force convex hull algorithm below. When the
array does not meet the base case, it is
    # split in half and the recursive DNCHull() call is made on the left and
right sub-arrays.
    #
    # Once the base cases are handled and the recursive calls are returned,
the function then passes the arrays of
    # points for the convex hulls of the left and right sub-arrays to
mergeHulls(). mergeHull then processes these
    # arrays using the upper and lower tangents to return an array of points
that make up the combined convex hull
    # for the left and right sub-hulls. This result is then either returned
up the recursive stack or back to
```

```
    # compute_hull() if it is the final call. The final result is the array
of points making up the convex hull of the
    # entire set of points.
    #
    # The time complexity of this function comes down to the divide and
conquer nature. A more complete theoretical
    # analysis of the Divide and Conquer function is found in the full
report. Since the function is breaking down
    # the larger array of points into two sub-problems of N/2 size. This
makes the time complexity of the Divide and
    # Conquer aspect of the algorithm O(log n).
    #
    # The time complexity is directly affected by the merging function since
it is considered post-work to the recursive
    # part of the function. Time complexity for mergeHull() is O(n^2), which
is detailed below.
    #
    # This makes our final time complexity for the Divide and Conquer Convex
Hull Algorithm = O(n^2 log n).
    #
    # Space complexity for this algorithm is ultimately O(N) as it has the
one array containing all of the points that
    # is being split up and worked on by the various helper functions
    def DNCHull(self, arr):  # Space complexity - O(N)

        # Base Case of Recursion - When divided arrays are 5 points or less,
find convex hull by brute force
        if len(arr) <= 5:  # Check if statement - Time: O(1)
            hull = self.BruteForce(arr)  # Call BruteForce(arr) to get convex
hull of small arrays -
            # Time: O(s^3) - see BruteForce()

            return hull  # Return array of points for small convex hull -
Time: O(1)

        # When array of points is not small enough yet for base case
        else:  # Check else statement - Time: O(1)
            half = len(arr) // 2  # Find midpoint of array of points using
floor division. Time: O(1)
            left = arr[:half]  # Set left subarray - Time: O(arr.size() / 2)
using python slicing
            right = arr[half:]  # Set right subarray - Time: O(arr.size() /
2) using python slicing

            # Recursively call back DNC Hull for left and right sub-arrays
            leftHull = self.DNCHull(left)  # Left sub-array recursive call
Time: O(log N)
            rightHull = self.DNCHull(right)  # Right sub-array recursive call
Time: O(log N)

            # Return the array of points for the merged left and right hulls
            return self.mergeHulls(leftHull, rightHull)  # MergeHulls call -
Time: O(N^2)

    # This is the function used to merge the left and right convex hulls
either found in the base case or
    # returned by the previous recursive iteration of DNCHull(). The function
```

```
relies on two helper functions,
    # getuppertangent() and getlowertangent(), which both return QLineF
objects containing the two points of either
    # tangent line for the combined hulls.
    #
    # Once the tangent lines are solved, mergeHulls performs one last
combining. Because at least one of the upper or
    # lower tangent lines will move from the starting line made from the
right-most point of the left hull and the
    # left-most point of the right hull, points need to be excluded from the
combined hull array while still
    # maintaining clean clockwise order to make the recursive merging work.
This is accomplished by doing a full
    # clockwise traversal of the two hulls, adding points to the result array
one by one. Points that should be dropped
    # are identified using p1 and p2 of the upper and lower tangent lines as
reference. These points are not skipped in
    # iteration, they are just ignored and not added to the result array. The
clockwise ordered result array is then
    # returned and is ready to be recursively merged again, or returned as
the final result.
    #
    # If there was no need to keep the clockwise order of the merged hulls,
the runtime complexity would be about
    # O(N) for this function, since the result of getuppertangent and
getlowertangent end up essentially traversing the
    # all the points across the two hulls in order to find the tangent lines.
However, my implementation of the code to
    # check all the points to remove non-necessary hull points from the
combined array is another loop through all
    # of the points in the two hulls again, which takes O(N) time.
    #
    # Because of this second traversal across the points in order to ensure
clockwise order is maintained for the new
    # hull, my mergeHulls() has a time complexity of O(N^2), losing some time
compared to what is expected. There is
    # likely a simpler, more natural way to remove unnecessary points from
the combined array while still maintaining
    # clockwise order for later merges. I just chose a very safe, if slightly
slower, approach to the solution.
    #
    # Space complexity for this functions is O(N^2), as we have the two sub-
hull arrays, each of size O(N/2) or a
    # combined size of O(N) and the result array. While the result array does
not include every point from the original
    # arrays, the merge usually only results in a few points being lost
between the sub-arrays and the combined one, so
    # I say it is fair to consider result equal in size to the two sub-
arrays, resulting in O(N^2) space being used.
    def mergeHulls(self, left, right):  # Space complexity of input values:
O(N)
        result = []  # Initialize result array containing merged hull values
- Time: O(1), Size: O(1)

        # Get upper and lower tangent lines - **Time Complexity of two
functions together considered to be O(N)
        uppertangent = self.getuppertangent(left, right)  # getuppertangent()
```

```
- Time: O(N/2)
        lowertangent = self.getlowertangent(left, right)  # getlowertangent()
- Time: O(N/2)

        # Initialize index variables to use in removing points not in hull
from result while maintaining
        # clockwise order
        leftindex = 0  # Initialize leftindex - Time: O(1)
        rightindex = 0  # Initialize rightindex - Time: O(1)

        # Add left-most point to result since it will always be in the hull
        result.append(left[leftindex])  # Append value to result - Time: O(1)

        # Add tangent points, remove unnecessary values, and maintain
clockwise order
        # Time: O(N) for entire section, divided into 5 subsection below of
O(N/5)
        # Space: O(N) for result array values added.

        # Append values from the left hull until reaching uppertangent.p1()
        while left[leftindex % len(left)] != uppertangent.p1():  # Time:
about O(N/5)
            leftindex = leftindex + 1  # Iterate variable - Time: O(1)
            result.append(left[leftindex % len(left)])  # Append value to
result - Time: O(1)

        # Iterate points of right hull until reaching uppertangent.p2(), not
appending any points
        while right[rightindex % len(right)] != uppertangent.p2():  # Time:
about O(N/5)
            rightindex = rightindex + 1  # Iterate variable - Time: O(1)

        # Explicitly append uppertangent.p2() to result in case that
uppertangent.p2() == lowertangent.p2()
        result.append(right[rightindex % len(right)])  # Append value to
result - Time: O(1)

        # Continue iterating through right hull until reaching
lowertangent.p2(), appending values on the way
        while right[rightindex % len(right)] != lowertangent.p2():  # Time:
about O(N/5)
            rightindex = rightindex + 1  # Iterate variable - Time: O(1)
            result.append(right[rightindex % len(right)])  # Append value to
result - Time: O(1)

        # Switch back to left hull, iterate through until reaching
lowertangent.p1(), not appending intermediate values.
        while left[leftindex % len(left)] != lowertangent.p1():  # Time:
about O(N/5)
            leftindex = leftindex + 1  # Iterate variable - Time: O(1)

        # Iterate and append points from left hull, until reaching starting
point which should not be added again.
        while left[leftindex % len(left)] != result[0]:  # Time: about O(N/5)
            result.append(left[leftindex % len(left)])  # Append value to
result - Time: O(1)
            leftindex = leftindex + 1  # Iterate variable - Time: O(1)
```

```
        # Return array of clockwise ordered hull points
        return result  # Return result - Time: O(1)

    # Brute force algorithm used only to calculate the convex hull of the
base case arrays. This algorithm iterates
    # through x-sorted array of points over a loop and two inner loops. The
inner loop checks each possible line made by
    # two points in the hull against every point in the hull, skipping cases
where the point to check is the same as
    # one of the points in the line. If every remaining point in the hull is
one the same side of the line, above
    # or below, both points in the line are added to the hullpoints array, as
long as they do not already exist in
    # the array.
    #
    # Points being above or below the line is evaluated using the
relationship ax + by = c, where a = y2 - y1,
    # b = x1 - x2, and c = (x1*y2) - (x2*y1). Using a and b calculated from
pointone and pointtwo, the point to test
    # arr[k] is evaluated. If c > a*arr[k].x()  + b*arr[k].y(), then the
point was above the line.
    # If c < a*arr[k].x()  + b*arr[k].y(), then the point was below. Points
above the line are counted. After all
    # iteration of the k inner loop, if count == the number of points tested
then all points were above the line,
    # and the two points in the line were added to the hullpoints array. If
count == 0, then all points were
    # below the line and the points from the line were also in the array and
were added.
    #
    # Once the points are found, they are passed to a helper function to
order them in clockwise order in order to
    # simplify the all subsequent merging. The clockwise order is maintained
carefully by the merging function, so the
    # helper function is only used in the context of the brute force
algorithm.
    #
    # Simply put, the brute force hull algorithm has a time complexity of
O(N^3), since it iterates using three for
    # loops over the same array of points to test each possible combination.
The function is slightly more optimized
    # since it skips any repeat points before doing any calculation. This
does not change the overall complexity.
    #
    # That being said, it is important to consider how this function is being
used. It is only used in solving the
    # convex hulls of the bases cases which are all of size <= 5, which is
very different from the original N points,
    # at least once the number of points gets larger. Because of this, it is
probably more fair to say that the time
    # complexity of the brute force algorithm is more like O(s) where little
s is the size of the input array which are
    # all of size <= 5. Because it is only being use on the smaller base case
arrays, it does not affect the overall
    # time complexity of the at-large divide and conquer algorithm.
    #
```

```
    # Space complexity for this function is O(s) (where s is the size <= 5).
At most, the hullpoints result array can
    # contain all of the 5 or fewer points in the input array, and one at
worst (though it is more likely to be 2 or 3
    # in the smallest cases). Because the difference between 1 and 5 is
insignificant, the space complexity can just be
    # called O(s).
    def BruteForce(self, arr):  # Space: O(s) for input array where s <= 5
        hullpoints = []  # Initialize result array hullpoints - Time: O(1)

        # Outer for loop for iterating pointone of line
        for i in range(len(arr)):  # First for loop - Time O(s)
            pointone = arr[i]  # Assign arr[i] to pointone for testing -
Time: O(1)

            # First inner for loop for iterating pointtwo of line
            for j in range(len(arr)):  # Second for loop - Time O(s)
                pointtwo = arr[j]  # Assign arr[j] to pointtwo for testing -
Time: O(1)

                # Skip cases where pointtwo is the sameas pointone
                if pointtwo == pointone:  # Check if condition - Time: O(1)
                    continue  # Continue - O(1)

                # Create QLineF object from pointone and pointtwo for
comparison.
                line = QLineF(pointone, pointtwo)  # Create line - O(1)

                # Solve for a, b, and c based on pointone and pointtwo
                a = line.y2() - line.y1()  # Calculate a - O(1)
                b = line.x1() - line.x2()  # Calculate b - O(1)
                c = (line.x1() * line.y2()) - (line.x2() * line.y1())  #
Calculate c - O(1)

                # Initialize count variable used for seeing how many points
were above/below the line
                count = 0  # Initialize counting variable - O(1)

                # Inner loop for iterating testpoint to check against the
line - Time:
                for k in range(len(arr)):  # Third for loop - Time O(s)
                    testpoint = arr[k]  # Assigning testpoint = arr[k] -
Time: O(1)

                    # Skip cases where testpoint is the same as either point
in the line to test
                    if testpoint == pointone or testpoint == pointtwo:  #
Evaluate if statement - Time: O(1)
                        continue  # Continue - O(1)

                    # Check if ax + by > c/testpoint is above line. If so,
iterate count.
                    if (a * testpoint.x()) + (b * testpoint.y()) > c:  #
Evaluate if statement - Time: O(1)
                        count = count + 1  # Iterate count - Time: O(1)

                # After inner for loop k is completed, if all points are
```

```
above or below line, add points to hullarray.
                if count == 0 or count == len(arr) - 2:  # Evaluate if
statement - Time: O(1)

                    # Skip adding points if point already exists in array
                    if pointone not in hullpoints:  # Evaluate if statement -
Time: O(1)
                        hullpoints.append(pointone)  # Append value to
hullpoints - Time: O(1)
                    if pointtwo not in hullpoints:  # Evaluate if statement -
Time: O(1)
                        hullpoints.append(pointtwo)  # Append value to
hullpoints - Time: O(1)

        # Get points sorted in clockwise order using helper function
clockwisesort() - Time:
        sortpoints = self.clockwisesort(hullpoints)  # Function sortpoints -
Time: O(s^3)

        return sortpoints  # Return array of hull points sorted in clockwise
order - Time: O(1)

    # A helper function used to ensure that the points in the base case
convex hulls are ordered in clockwise order to
    # use in a circular array for merging. Input to the brute force algorithm
comes in as an array of points sorted by
    # x-values, but it does not guarantee that points are in clockwise order.
This function changes the array from
    # x-sorted to clockwise ordered.
    #
    # The further left point in the array is chosen as the de facto starting
point for the clockwise order. From there,
    # the coordinates of the center point of the hull array is calculated by
summing the x and y values of
    # the array and dividing the the two results by the length of the array.
Once the center point is found, a line
    # is drawn between the starting point and the center array.
    #
    # While looping through each point in the hull arrays, a second line is
drawn between each point in the array and
    # the center point. Using the built in QLineF function
QLineF.angleTo(line), the angle in degrees between the
    # starting point and the second line is calculated. This angle in degrees
is saved to a separate array as a duple
    # paired with its index in the hull point array.
    #
    # Once the angle has been calculated, the angle-index duple array is
sorted by increasing angle in degrees
    # using a lambda function as key for python's list.sort() function, to
sort by increasing angle. This works
    # because the lowest angle from the starting line would be the next point
in clockwise order. By iterating one
    # more time through the array of angles and indices, the point from the
original array is added using the index
    # from the duple array, and finally added to the clockwiseorder point
array and returned
    #
```

```
    # Similar to the base case brute force algorithm above, this function
only deals with arrays of size s (size <= 5).
    # Because of this, time complexity calculations are simplified. This
function loops through the values of the array
    # three times: Once to get the sum of x and y values to find the center,
a second time to get the angles of each
    # line from the center, and a third to add the points in the correct
order to the result array. This would normally
    # be a time complexity of O(N^3), but using the guaranteed small array
size I have designated as s, this time
    # complexity would be O(s^3)
    #
    # Same as the time complexity, the space complexity only deals with a
largest size of s for the result array.
    # Since this is a type of sorting algorithm, the result array will have
the exact size of the input array, just in
    # a different order. We do create a second array of duples that is used
to store angle and index values, which would
    # be of the same length, just with a second column of values per row. In
total, we essentially have 3 arrays of size
    # s created and used during this function plus the input array of O(s0,
    # so our space complexity could be labeled O(4s) or just O(s).
    def clockwisesort(self, arr):  # Space: O(s) for input
        clockwiseorder = []  # Initialize clockwiseorder array - Time: O(1),
Space: O(1)

        # Initialize index value to get first item from original array and
use later for iterating through hull points
        minindex = 0  # Initialize variable - Time: O(1)

        # Initialize sum values used to calculate center of hull points
        sumx = 0  # Initialize variable - Time: O(1)
        sumy = 0  # Initialize variable - Time: O(1)

        # Loop through original array and calculate the sum of all x and y
values
        for index in range(len(arr)):  # First for loop - Time: O(s)
            sumx += arr[index].x()  # Add value to sum - Time: O(1)
            sumy += arr[index].y()  # Add value to sum - Time: O(1)

        # Calculate x and y coordinates of center point using sum values from
above.
        center = QPointF((sumx / len(arr)), (sumy / len(arr)))  # Calculate
middle - Time: O(1)

        # Add first point from original array to clockwiseorder array to use
as de facto starting point
        clockwiseorder.append(arr[minindex])  # Append value to
clockwiseorder - Time: O(1)

        # Create QLineF using center point and starting point of
clockwiseorder
        startline = QLineF(center, arr[minindex])  # Create line - Time: O(1)

        # Create empty array for angle and indices duple pairs
        angleandindex = []  # Create empty duple array - Time: O(1)
```

```
        # Loop through all points in original array except first point to
calculate clockwise angle from starting point
        for i in range(minindex + 1, len(arr)):  # Second for loop - Time:
O(s), Space O(2s)
            # Save current point from array and make line with center point
to calculate angle
            point = arr[i % len(arr)]  # Get point - Time: O(1)
            testline = QLineF(center, point)  # Make line with center point -
Time: O(1)

            # Calculate angle using angleTo() and save to angleandindex array
            angle = startline.angleTo(testline)  # Calculate angle - Time:
O(1)
            angleandindex.append([i, angle])  # Append duple to array - Time:
O(1)

        # Use list.sorted(key) with lambda function to sort on angle values
as key for function
        angleindexsorted = sorted(angleandindex, key=lambda l: l[1])  #
Python sorted - Time: O(1)

        # Loop through all values in angleindexsorted to extract index from
duple and save value to clockwiseorder
        # array in angle-determined clockwise order
        for j in range(len(angleindexsorted)):  # Third for loop - Time: O(s)
            # Extract index from duple and append point from original array
at index to clockwiseorder
            index = angleindexsorted[j][0]  # Get index from duple array -
Time: O(1)
            clockwiseorder.append(arr[index])  # Append to clockwise order -
Time: O(1), Space: O(s)

        return clockwiseorder  # Return clockwiseorder array back to
BruteForce() function - Time: O(1)

    # A helper function used in getuppertangent() and getlowertangent() in
order to find the starting value of the
    # left side array for merging. Finding the left most x-value in the right
side array is trivial, since it was
    # purposefully added first to the clockwise sorted arrays, it can be
extracted directly. However, the rightmost
    # x-value which is used to make the starting line for testing for the
convex hull tangent lines is no longer found
    # at the final value of the left points array after the clockwise sort of
the base case hulls.
    #
    # This helper function is just a simple for loop through the left array
to find the max x-value and return the index
    # it is found at. The code is separated into the helper function to
reduce duplicate code since it is used by
    # both the upper and lower tangent functions.
    #
    # This is a simple looping function meant to identify the max x value
from an array. The arrays used in this
    # can be as small as s to N/2, since this function is only called on the
left sub-hull of the hulls to be merged.
    # In the context of this function, the time complexity could be said to
```

```
be O(N) since it is a for loop over the
    # entire array of points. However, in the larger picture of the overall
function, it acts as more of a O(1) constant
    # (more accurately O(N/2) time function, since it never iterates over the
entire array of N points, even on the
    # final step of recursion. Because of this, it does not impact the
overall time complexity of the divide and
    # conquer
    #
    # The space complexity of this function is similarly dependant on the
input size, but has a max size of O(N/2)
    # where N/2 is the size of the largest array that will be passed into it.
Similar to time complexity, in the
    # context of the function, the space complexity if O(N), however, in the
bigger picture it is more accurately
    # O(N/2) which is more similar to O(1).
    def getRightMostX(self, arr):  # Space: O(N) in function context, O(N/2)
in big picture context for input array

        # Set initial value of array to be the starting max x value and
starting index of max value to 0
        maxx = arr[0]  # Initialize variable - Time: O(1)
        maxindex = 0  # Initialize variable - Time: O(1)

        # Loop through all values of left side array to look for max x value,
found somewhere in the middle.
        for i in range(len(arr)):  # For loop on arr - Time: O(N) (in
function context)/ O(N/2) (in bigger picture)

            # If the x-value of the current point is greater than previous
max, update maxx and maxindex
            if arr[i].x() > maxx.x():  # Evaluate if statement - Time: O(1)
                maxx = arr[i]  # Assign variable - Time: O(1)
                maxindex = i  # Assign variable - Time: O(1)

        return maxindex  # Return index where max value is found to tangent
functions - Time: O(1)

    # A helper function for mergehulls() using the left and right sub-hull
arrays from each recursive call.
    # The implementation is done with a larger while loop that contains two
while loops; the first one iterates until
    # the tangent line of the left array is found and the second one iterates
through until the tangent line of the
    # right array is found. The outer while loop will only stop when both
inner while loops do not complete a full
    # iteration, indicating that the tangent line connecting both hulls is
the tangent line of each individual sub-hull,
    # and as a result is the upper tangent of the combined hull.
    #
    # The implementation uses a stepping pattern to check if the current
given line is a tangent line of the sub-hulls.
    # The starting line, tangentline, is drawn from the rightmost point of
the left array and the left most point of the
    # right array. The outer while loop start and sets its own conditional
variable to true. In the first while loop,
    # the next counter-clockwise point in the circular array is saved. The
```

```
slope of the original tangent line and the
    # line using the same point from the right array and the next counter-clockwise point is saved. The loop
    # then checks if the slope of the current tangent line < the line using the next point. If this condition
    # is true, the while loop breaks, since the current tangentline is a tangentline of the left array. If it is false,
    # tangentline is updated to the next point, the outer loop conditional variable is set to false, and the first inner
    # loop continues.
    #
    # The same process happens for the second inner loop on the right sub-hull, except it uses the next clockwise point
    # and checks for the currentslope > slope of line using the next clockwise point for the loop break condition.
    # This conditional checking creates the stepping action for both side. When the outer while loop traverse both
    # inner loops without either of them moving the tangent line, then the line is a tangent of both sub-hulls and the
    # combined hull as a result.
    #
    # The time complexity of this function changes with each recursive call, as the input arrays get larger and larger
    # approach size of N between the two arrays. During the final recursive call, the two hulls to combine will be
    # of size N points, however, traversal of the upper tangent does not take O(N) time to complete, since the furthest
    # the tangent line could be would be about on the other side from the most inside points, and often is much shorter
    # than that. As a result, the time complexity is more accurately estimated in a worst case scenario to be O(N/2),
    # which is more similar to a constant O(1) time. In conjunction with getlowertangent(), the time complexity for the
    # two tangent functions together could be estimated at O(N) (which I have done in the larger time complexity
    # analysis of the entire divide and conquer) since it is possible to traverse the almost the entire array in search
    # of the tangent lines depending on the spread of points. Individually however, it is more like O(N/2) time.
    #
    # Space complexity for this function is more straightforward. The final recursive merges will have hulls containing
    # essentially the full set of points to work with or O(N) points between the left and right sub-hull arrays.
    # Other than the input arrays, this function works in constant space variables, so the O(N) is the total size
    # complexity for this function.
    def getuppertangent(self, left, right):  # Space O(N) for two input arrays considered together.
        # Get starting indices of middle points from left and right circular arrays
        leftsidestartindex = self.getRightMostX(left)  # Call get right most x function - Time: O(N/2) or O(1)
        rightsidestartindex = 0  # Initialize variable - Time: O(1)

        # Get starting points using indices for leftmost and rightmost points above.
```

```
        p = left[leftsidestartindex]  # Initialize variable - Time: O(1)
        q = right[0]  # Initialize variable - Time: O(1)

        # Get starting tangentline of right most point of left and left most
point of right.
        tangentline = QLineF(p, q)  # Make line - Time: O(1)

        # Initialize outer loop conditional variable to False to enter loop
for the first time
        foundtangentline = False  # Initialize variable - Time: O(1)

        # Set iterator values used to traverse left and right sub-hulls in
clockwise or counter-clockwise order by steps
        leftiterator = 1  # Initialize variable - Time: O(1)
        rightiterator = 1  # Initialize variable - Time: O(1)

        # Enter outer while loop, should not break until combined upper
tangent line is found.
        while not foundtangentline:  # Run while loop - O(1)

            # Set outer loop conditional variable to true, if tangentline
does not move, this value will not change.
            foundtangentline = True  # Assign variable - Time: O(1)

            # Enter first inner for loop to check if tangentline is tangent
to left sub-hull
            while True:  # Run while loop - Time: O(1)

                # Get slope using current values of tangent line saved to p
and q
                currentslope = (q.y() - p.y()) / (q.x() - p.x())  # Calculate
slope - Time: O(1)

                # Get the point for the next counter-clockwise value in left
sub-hull, save to r.
                r = left[(leftsidestartindex - leftiterator) % len(left)]  #
Get next r - Time: O(1)

                # Get value of slope of the new line using the same point q
on the right side and the new point r.
                nextslope = (q.y() - r.y()) / (q.x() - r.x())  # Calculate
slope - Time: O(1)

                # If slope is increasing with the new point, break the loop.
tangentline for left hull is valid.
                if currentslope < nextslope:  # Evaluate if statement - Time:
O(1)
                    break  # Break - Time: O(1)
                # If slope is decreasing, update tangentline to new value r
and p = r to update for next iteration.
                tangentline = QLineF(r, q)  # Update tangent line - Time:
O(1)
                p = r  # Update p - Time: O(1)

                # Update counter-clockwise traversal iterator value
                leftiterator = leftiterator + 1  # Assign variable - Time:
O(1)
```

```
                # Set outer loop condition to false; Outer loop will have to
run at least one more iteration to find
                # combined upper tangent line.
                foundtangentline = False  # Assign variable - Time: O(1)

            # Enter second inner for loop to check if tangentline is tangent
to right sub-hull
            while True:  # Run while loop - Time: O(1)

                # Get slope using current values of tangent line saved to p
and q after left side iteration.
                currentslope = (q.y() - p.y()) / (q.x() - p.x())  # Calculate
slope - Time: O(1)

                # Get the point for the next clockwise value in right sub-
hull, save to r.
                r = right[(rightsidestartindex + rightiterator) % len(right)]
# Get next r - Time: O(1)

                # Get value of slope of the new line using the same point p
on the left side and the new point r.
                nextslope = (r.y() - p.y()) / (r.x() - p.x())  # Calculate
slope - Time: O(1)

                # If slope is decreasing with the new point, break the loop.
tangentline for right hull is valid.
                if currentslope > nextslope:  # Evaluate if statement - Time:
O(1)
                    break  # Break - Time: O(1)

                # If slope is increasing, update tangentline to new value r
and q = r to update for next iteration.
                tangentline = QLineF(p, r)  # Update tangent line - Time:
O(1)
                q = r  # Update q - Time: O(1)

                # Update clockwise traversal iterator value
                rightiterator = rightiterator + 1  # Assign variable - Time:
O(1)

                # Set outer loop condition to false; Outer loop will have to
run at least one more iteration to find
                # combined upper tangent line.
                foundtangentline = False  # Assign variable - Time: O(1)

        return tangentline  # Once outer while loop breaks, return true
uppertangentline - Time: O(1)

    # A helper function for mergeHulls parallel to getuppertangent(). This
function takes the same left and right
    # sub-hulls as getuppertangent and performs similar calculations on them,
this time to get the lower tangent line of
    # the combined hull.
    #
    # The two functions are essentially identical with 4 differences.
    # 1) The first inner while loop (traversing the left sub-hull) now moves
```

```
in a clockwise direction.
    # 2) The first inner while loop now checks for decreasing slope to know
when to break the loop
    # 3) The second inner while loop (traversing the right sub-hull) now
moves in a counter-clockwise direction.
    # 4) The second inner while loop now checks for increasing slope as its
condition to break the loop
    #
    # In essence, getting the lower tangent is the same functionally, only
reversing the movement and loop conditions
    # of the upper tangent function. Because of the similarities, the
specific workings of the functions won't be
    # repeated again here.
    #
    # The justification for the time and space complexity of this function is
the same as for getuppertangent, so it
    # will not be duplicated here.
    # The time complexity for getlowertangent in isolation is O(N/2), O(N)
when considered together
    # with getuppertangent.
    #
    # The space complexity for getlowertangent is O(N) for the combined size
of the two input arrays.
    def getlowertangent(self, left, right):  # Space O(N) for two input
arrays considered together.
        # Get starting indices of middle points from left and right circular
arrays
        leftsidestartindex = self.getRightMostX(left)  # Call get right most
x function - Time: O(N/2) or O(1)
        rightsidestartindex = 0  # Initialize variable - Time: O(1)

        # Get starting points using indices for leftmost and rightmost points
above.
        p = left[leftsidestartindex]  # Initialize variable - Time: O(1)
        q = right[0]  # Initialize variable - Time: O(1)

        # Get starting tangentline of right most point of left and left most
point of right.
        tangentline = QLineF(p, q)  # Make line - Time: O(1)

        # Initialize outer loop conditional variable to False to enter loop
for the first time
        foundtangentline = False  # Initialize variable - Time: O(1)

        # Set iterator values used to traverse left and right sub-hulls in
clockwise or counter-clockwise order by steps
        leftiterator = 1  # Initialize variable - Time: O(1)
        rightiterator = 1  # Initialize variable - Time: O(1)

        # Enter outer while loop, should not break until combined upper
tangent line is found.
        while not foundtangentline:  # Run while loop - O(1)

            # Set outer loop conditional variable to true, if tangentline
does not move, this value will not change.
            foundtangentline = True  # Assign variable - Time: O(1)
```

```
            # Enter first inner for loop to check if tangentline is tangent
to left sub-hull
            while True:  # Run while loop - Time: O(1)

                # Get slope using current values of tangent line saved to p
and q
                currentslope = (q.y() - p.y()) / (q.x() - p.x())  # Calculate
slope - Time: O(1)

                # Get the point for the next clockwise value in left sub-
hull, save to r.
                r = left[(leftsidestartindex + leftiterator) % len(left)]  #
Get next r - Time: O(1)

                # Get value of slope of the new line using the same point q
on the right side and the new point r.
                nextslope = (q.y() - r.y()) / (q.x() - r.x())  # Calculate
slope - Time: O(1)

                # If slope is decreasing with the new point, break the loop.
tangentline for left hull is valid.
                if currentslope > nextslope:  # Evaluate if statement - Time:
O(1)
                    break  # Break - Time: O(1)
                # If slope is decreasing, update tangentline to new value r
and p = r to update for next iteration.
                tangentline = QLineF(r, q)  # Update tangent line - Time:
O(1)
                p = r  # Update p - Time: O(1)

                # Update counter-clockwise traversal iterator value
                leftiterator = leftiterator + 1  # Assign variable - Time:
O(1)

                # Set outer loop condition to false; Outer loop will have to
run at least one more iteration to find
                # combined upper tangent line.
                foundtangentline = False  # Assign variable - Time: O(1)

            # Enter second inner for loop to check if tangentline is tangent
to right sub-hull
            while True:  # Run while loop - Time: O(1)

                # Get slope using current values of tangent line saved to p
and q after left side iteration.
                currentslope = (q.y() - p.y()) / (q.x() - p.x())  # Calculate
slope - Time: O(1)

                # Get the point for the next counter-clockwise value in right
sub-hull, save to r.
                r = right[(rightsidestartindex - rightiterator) % len(right)]
# Get next r - Time: O(1)

                # Get value of slope of the new line using the same point p
on the left side and the new point r.
                nextslope = (r.y() - p.y()) / (r.x() - p.x())  # Calculate
slope - Time: O(1)
```

```
                # If slope is decreasing with the new point, break the loop.
tangentline for right hull is valid.
                if currentslope < nextslope:  # Evaluate if statement - Time:
O(1)
                    break  # Break - Time: O(1)

                # If slope is increasing, update tangentline to new value r
and q = r to update for next iteration.
                tangentline = QLineF(p, r)  # Update tangent line - Time:
O(1)
                q = r  # Update q - Time: O(1)

                # Update clockwise traversal iterator value
                rightiterator = rightiterator + 1  # Assign variable - Time:
O(1)

                # Set outer loop condition to false; Outer loop will have to
run at least one more iteration to find
                # combined upper tangent line.
                foundtangentline = False  # Assign variable - Time: O(1)

        return tangentline  # Once outer while loop breaks, return true
uppertangentline - Time: O(1)
```

Proj2GUI.py

```
#!/usr/bin/env python3


import math
import random
import signal
import sys
import time


from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
   from PyQt5.QtWidgets import *
   from PyQt5.QtGui import *
   from PyQt5.QtCore import *
elif PYQT_VER == 'PYQT4':
   from PyQt4.QtGui import *
   from PyQt4.QtCore import *
else:
   raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


#TODO: Error checking on txt boxes
#TODO: Color strings


# Import the code with the actual implementation
from convex_hull import *
#from convex_hull_complete_nonthread import *


# This class controls the visual stuff in the GUI.  An instance of it is
passed to the solver
# when it is called so that wrapper functions in the file "convex_hull.py"
can update the GUI
#
class PointLineView( QWidget ):
   def __init__( self, status_bar ):
      super(QWidget,self).__init__()
      self.setMinimumSize(600,400)

      self.pointList  = {}
      self.lineList   = {}
      self.status_bar = status_bar

   def displayStatusText(self, text):
      self.status_bar.showMessage(text)
      self.update()
      app.processEvents               #Why is this necessary????

   def clearPoints(self):
      self.pointList = {}

   def clearLines(self, lines=None):
      if(not lines):
```

```
      self.lineList = {}
   else:
      for color in self.lineList:
         for line in lines:
            try:
               self.lineList[color].remove(line)
            except:
               pass
   self.update()
   app.processEvents()                #Why is this necessary????

def addPoints( self, point_list, color ):
   if color in self.pointList:
      self.pointList[color].extend( point_list )
   else:
      self.pointList[color] = point_list

def addLines( self, line_list, color ):
   if color in self.lineList:
      self.lineList[color].extend( line_list )
   else:
      self.lineList[color] = line_list
   self.update()
   app.processEvents()                #Why is this necessary????

def paintEvent(self, event):
   painter = QPainter(self)
   painter.setRenderHint(QPainter.Antialiasing,True)

   w = self.width() / 2.0
   h = self.height() / 2.0
   w2h_desired_ratio = 1.5
   if w / h < w2h_desired_ratio:
      h = w / w2h_desired_ratio
   else:
      w = h * w2h_desired_ratio

   tform = QTransform()
   tform.translate(self.width()/2.0,self.height()/2.0)
   tform.scale(1.0,-1.0)
   painter.setTransform(tform)

   for color in self.lineList:
      c = QColor(color[0],color[1],color[2])
      painter.setPen( c )
      for line in self.lineList[color]:
         ln = QLineF( w*line.x1(), h*line.y1(), w*line.x2(), h*line.y2() )
         painter.drawLine( ln )

   for color in self.pointList:
      c = QColor(color[0],color[1],color[2])
      painter.setPen( c )
      for point in self.pointList[color]:
         pt = QPointF(w*point.x(), h*point.y())
         painter.drawEllipse( pt, 1.0, 1.0)
```

```
# Main GUI class
#
class Proj2GUI( QMainWindow ):
   def __init__( self ):
      super(Proj2GUI,self).__init__()

# This is where the points for a problem instance are kept
      self.points = None

# Getting an instance of your solver
      self.solver = ConvexHullSolver()

# start the GUI
      self.initUI()

# Generator for new sets of points that represent hull finding problem
instances
   def newPoints(self):

      # TODO - ERROR CHECKING!!!!
      if self.randBySeed.isChecked():
         seed = int(self.randSeed.text())
         random.seed( seed )
      else: # do by time
         random.seed( time.time() )

      ptlist = []
      unique_xvals = {}
      max_r  = 0.98
      WIDTH  = 1.0
      HEIGHT = 1.0
      npoints = int(self.npoints.text())
      if self.distribOval.isChecked():
         while len(ptlist) < npoints:
            x = random.uniform(-1.0,1.0)
            y = random.uniform(-1.0,1.0)
            if x**2+y**2 <= max_r**2:
               xval = WIDTH*x
               yval = HEIGHT*y
               if not xval in unique_xvals:
                  ptlist.append( QPointF(xval,yval) )
                  unique_xvals[xval] = 1      # dict/map with float keys?
      elif self.distribSphere.isChecked():
         while len(ptlist) < npoints:
            x = random.uniform(-1.0,1.0)
            y = random.uniform(-1.0,1.0)
            z = random.uniform(-1.0,1.0)
            if x**2 + y**2 + z**2 <= max_r**2:
               xval = WIDTH*x
               yval = HEIGHT*y
               if not xval in unique_xvals:
                  ptlist.append( QPointF(xval,yval) )
                  unique_xvals[xval] = 1
      elif self.distribGaussian.isChecked():
         while len(ptlist) < npoints:
            x = random.gauss(0.0,0.25)
            y = random.gauss(0.0,0.25)
```

```
            if x**2+y**2 <= max_r**2:
               xval = WIDTH*x
               yval = HEIGHT*y
               if not xval in unique_xvals:
                  ptlist.append( QPointF(xval,yval) )
                  unique_xvals[xval] = 1
      return ptlist

# Methods that handle GUI events
   def clearClicked(self):
      self.view.clearLines()
      self.view.displayStatusText('')
      self.solveButton.setEnabled(True)
      self.view.update()
      app.processEvents()                      #Why is this necessary?????

   def generateClicked(self):
      if self.points:
         self.view.clearPoints()
         self.view.clearLines()
      self.points = self.newPoints()
      self.view.addPoints( self.points, (0,0,0) )
      self.solveButton.setEnabled(True)
      self.view.update()
      app.processEvents()                      #Why is this necessary?????

# This the method that hooks into your solver.  It passes the
# problem instance/solution request (a set of points) to the solver, along
with
# the recursion flag to indicate whether to animate the solution
# and a view object so the GUI can be updated
   def solveClicked(self):
      self.generateButton.setEnabled(False)
      self.clearButton.setEnabled(False)
      self.solveButton.setEnabled(False)
      self.view.update()
      app.processEvents()                      #Why is this necessary?????

self.solver.compute_hull(self.points,self.showRecursion.isChecked(),self.view
)
      self.generateButton.setEnabled(True)
      self.clearButton.setEnabled(True)
      self.view.update()
      app.processEvents()                      #Why is this necessary?????

   def _randbytime(self):
      self.randSeed.setEnabled(False)

   def _randbyseed(self):
      self.randSeed.setEnabled(True)

# Setting up the GUI
   def initUI( self ):
      self.setWindowTitle('Convex Hull')
      self.setWindowIcon( QIcon('icon312.png') )

      self.statusBar = QStatusBar()
```

```
        self.setStatusBar( self.statusBar )

        vbox = QVBoxLayout()
        boxwidget = QWidget()
        boxwidget.setLayout(vbox)
        self.setCentralWidget( boxwidget )

        self.view           = PointLineView( self.statusBar )
        self.npoints        = QLineEdit('10')
        self.generateButton = QPushButton('Generate')
        self.solveButton    = QPushButton('Solve')
        self.clearButton    = QPushButton('Clear To Points')
        self.distribOval    = QRadioButton('Uniform')
        self.distribSphere  = QRadioButton('Spherical')
        self.distribGaussian= QRadioButton('Gaussian')

        self.randByTime     = QRadioButton('Random')
        self.randBySeed     = QRadioButton('Seed')
        self.randSeed       = QLineEdit('0')

        self.showRecursion = QCheckBox('Show Recursion')

        h = QHBoxLayout()
        h.addWidget( self.view )
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addWidget( QLabel( 'Number of points to generate: ' ) )
        h.addWidget( self.npoints )
        h.addWidget( self.generateButton )
        h.addWidget( self.solveButton )
        h.addWidget( self.clearButton )
        h.addStretch(1)
        vbox.addLayout(h)

        h = QHBoxLayout()
        grp = QButtonGroup(self)
        grp.addButton(self.distribOval)
        grp.addButton(self.distribSphere)
        grp.addButton(self.distribGaussian)
        h.addWidget( QLabel( 'Distribution of generated points: ' ) )
        h.addWidget( self.distribOval )
        h.addWidget( self.distribSphere )
        h.addWidget( self.distribGaussian )
        h.addStretch(1)
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addWidget( QLabel( 'Point Locations: ' ) )
        grp = QButtonGroup(self)
        grp.addButton(self.randByTime)
        grp.addButton(self.randBySeed)
        h.addWidget( self.randByTime )
        h.addWidget( self.randBySeed )
        h.addWidget( self.randSeed )
        h.addStretch(1)
        h.addWidget(self.showRecursion)
```

```python
        vbox.addLayout(h)

        self.generateButton.clicked.connect(self.generateClicked)
        self.solveButton.clicked.connect(self.solveClicked)
        self.clearButton.clicked.connect(self.clearClicked)

        self.randByTime.clicked.connect(self._randbytime)
        self.randBySeed.clicked.connect(self._randbyseed)


        self.randByTime.setChecked(True)
        self.distribOval.setChecked(True)
        self.generateClicked()

        self.showRecursion.setChecked(False)

        self.show()




if __name__ == '__main__':
    # This line allows CNTL-C in the terminal to kill the program
    signal.signal(signal.SIGINT, signal.SIG_DFL)

    app = QApplication(sys.argv)
    w = Proj2GUI()
    sys.exit(app.exec())
```

which_pyqt.py

```
PYQT_VER = 'PYQT5'
```