1.

```
#!/usr/bin/python3
# why the shebang here, when it's imported?  Can't really be used stand alone, right?  And fermat.py didn't have one...
# this is 4-5 seconds slower on 1000000 points than Ryan's desktop...  Why?


from which_pyqt import PYQT_VER

if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF, QThread, pyqtSignal
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF, QThread, pyqtSignal
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import time
import math


# Using the left hull find its highest x value
# Time Complexity is O(n) where n is the size of the leftside_hull list of points
# Space Complexity is constant O(1) because we aren't storing anything new here
def find_rightmost(leftside_hull):
    assert(type(leftside_hull) == list and type(leftside_hull[0] == QPointF))
    for i in range(len(leftside_hull)):
        if ((i + 1) >= len(leftside_hull)):  # If adding one to the index makes it bigger or equal to the length we already
found the highest x value
            return i  # Also ensures that we don't go out of range

        if (leftside_hull[i + 1].x() < leftside_hull[i].x()):  # If the next x value is lower than the current x value then we
found the max
            return i

# Find the slope of the two points
# Time Complexity is O(1) because it's constant time for finding the slope cause it is just one coordinate minus another
    # and it has nothing to do with the number of points total
# Space Complexity is O(1) because we aren't storing anything new here just returning a value
def find_slope(point1, point2):
    assert(type(point1) == QPointF and type(point2) == QPointF)

    return ((point2.y() - point1.y()) / (point2.x() - point1.x()))

# Finds upper or lower tangent and returns the indices of the upper or lower tangents
# Time complexity is O(n) because the worst case is that the index that is needed is the last one in the list that is
checked
    # n is the size of the hull being passed in (number of points)
# The space complexity is O(1) because it is a constant stack being used,
    # there are no recursive calls increasing the memory being used
def find_tangent(leftside_hull, rightside_hull, lefthull_index, righthull_index):
    slope = find_slope(leftside_hull[lefthull_index], rightside_hull[righthull_index])
    update = True

    while (update):
        update = False

        left_decreasing = True
        while (left_decreasing):
            newleftpt_index = lefthull_index - 1
            if (newleftpt_index < 0):  # If the index becomes negative wrap around to end of list
                newleftpt_index = len(leftside_hull) - 1

            new_slope = find_slope(leftside_hull[newleftpt_index], rightside_hull[righthull_index])
```

```
        if (new_slope > slope):  # Slope needs to decrease in the left hull
            left_decreasing = False
        else:
            update = True
            lefthull_index = newleftpt_index
            slope = new_slope

    right_increasing = True
    while (right_increasing):
        newrightpt_index = righthull_index + 1
        if (newrightpt_index == len(rightside_hull)):  # Wrap to beginning of list if index is equal to length
            newrightpt_index = 0

        new_slope = find_slope(leftside_hull[lefthull_index], rightside_hull[newrightpt_index])

        if (new_slope < slope):  # Slope needs to increase in the right hull
            right_increasing = False
        else:
            update = True
            righthull_index = newrightpt_index
            slope = new_slope

    return lefthull_index, righthull_index


# Combines the two arrays of QPointF points into one hull. The array returned is still in clockwise order
# Time complexity here is O(n) because it calls find_tangent and find_rightmost which both of them are O(n)
# Space complexity here is O(n) where n is the size of the merged hull from the left and right hulls
# n in both of these cases have to deal with the number of points passed in from each side of the hull
def combine_hulls(leftside_hull, rightside_hull):

    lefthull_index = find_rightmost(leftside_hull)  # The right most can be anywhere in the array but it can be found by finding the highest x valued point
    righthull_index = 0  # The right's left most point will always be the first index because of the clockwise order of how it was made

    #  Finds the upper and lower tangents of hulls left and right based off of the index of the left most and right most points
    leftpointidx_uppertangent, rightpointidx_uppertangent = find_tangent(leftside_hull, rightside_hull, lefthull_index, righthull_index)
    rightpointidx_lowertangent, leftpointidx_lowertangent = find_tangent(rightside_hull, leftside_hull, righthull_index, lefthull_index)

    merged_hull = leftside_hull[0:leftpointidx_uppertangent + 1] # Grab the first to the leftpoint
    merged_hull += (rightside_hull[rightpointidx_uppertangent:rightpointidx_lowertangent + 1]) # Grab the points from the right hull of the upper tangent to the lower tangent and add to the left hull array

    if (rightpointidx_lowertangent == 0):
        merged_hull += rightside_hull[rightpointidx_uppertangent:]  # Grab all of the points since 0 is the start of the right array that needs to be merged
        merged_hull.append(rightside_hull[0])

    if not (leftpointidx_lowertangent == 0):  # Grab all of the rest of the points in the left hull
        merged_hull += (leftside_hull[leftpointidx_lowertangent:])

    return merged_hull


# Cuts the sorted_points in half until a list of 3 points is made (base case)
# It orders them in a list in clockwise order
# Time complexity is O(nlogn) where n is the number of sorted_points
```

```
# Space complexity is O(n) because the call is recursive and makes a new stack with
    # each recursive call to make_convex which stores n points in the left or right hull
def make_convex(sorted_points):
    if (len(sorted_points) <= 3):  # Base case
        # If the length is 1 or 2 the array will already be in clockwise order since it doesn't matter which
        # point you start with it will be the same order clockwise no matter what
        # So need to ensure that three points are in clockwise order
        if (len(sorted_points) == 3):  # Big O is constant time here to look up and switch
            slope = find_slope(sorted_points[0], sorted_points[1])
            next_slope = find_slope(sorted_points[0], sorted_points[2])
            if (next_slope > slope):  # To go in clockwise order the slopes should be decreasing so need to swap points here
                point_holder = sorted_points[2]
                sorted_points[2] = sorted_points[1]
                sorted_points[1] = point_holder
            elif (sorted_points[1].y() == sorted_points[2].y()):
                # Checks to see if the y values are equal cause if they are we need to check if the first point is above the other
two
                # If it is above its y value will be greater than the other two's y values and thus points at 1 and 2 need to be
                # Swapped to maintain clockwise order
                if (sorted_points[0].y() > sorted_points[1].y()):
                    point_holder = sorted_points[2]
                    sorted_points[2] = sorted_points[1]
                    sorted_points[1] = point_holder
        return sorted_points
    # Divide and conquer part that makes it O(nlogn) recurse till base case above is reached
    middle = math.floor(len(sorted_points) / 2)
    leftside_hull = make_convex(sorted_points[:middle])  # Takes points from 0 to middle - 1
    rightside_hull = make_convex(sorted_points[middle:])  # Takes points from middle to end

    combined_hull = combine_hulls(leftside_hull, rightside_hull)

    return combined_hull


# The overall time complexity here is O(nlogn) this is from the sorted function and also the recursive make_convex
function
    # n is the number of points passed in
# There is also a complexity of O(n) from iterating over the array to draw the lines but this is less than O(nlogn) so it
doesn't matter
# The space complexity is O(n) where n is the number of points passed in. This is because the points are stored in
    # sorted_points by their x value and there are n of them
class ConvexHullSolverThread(QThread):
    def __init__(self, unsorted_points, demo):
        self.points = unsorted_points
        self.pause = demo
        QThread.__init__(self)

    def __del__(self):
        self.wait()

    show_hull = pyqtSignal(list, tuple)
    display_text = pyqtSignal(str)

    # some additional thread signals you can implement and use for debugging, if you like
    show_tangent = pyqtSignal(list, tuple)
    erase_hull = pyqtSignal(list)
    erase_tangent = pyqtSignal(list)

    def run(self):
        assert (type(self.points) == list and type(self.points[0]) == QPointF)

        n = len(self.points)
```

```
    print('Computing Hull for set of {} points'.format(n))

    t1 = time.time()
    sorted_points = sorted(self.points, key=lambda p: p.x())  # Sorts the points by x value
    t2 = time.time()
    print('Time Elapsed (Sorting): {:3.3f} sec'.format(t2 - t1))

    t3 = time.time()
    convex_hull = make_convex(sorted_points)  # Makes an array of the convex hull points
    t4 = time.time()

    USE_DUMMY = False
    if USE_DUMMY:
        # this is a dummy polygon of the first 3 unsorted points
        polygon = [QLineF(self.points[i], self.points[(i + 1) % 3]) for i in range(3)]

        # 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
        assert (type(polygon) == list and type(polygon[0]) == QLineF)
        # send a signal to the GUI thread with the hull and its color
        self.show_hull.emit(polygon, (255, 0, 0))

    else:
        # Store the lines of the hull in convex_lines array to be sent to the GUI to draw them
        # Iterate over the array hull is complexity O(n) where n is the size of the hull Space is O(n) too
        # The % makes it wrap back around to the beginning of the array list so that the last point can be connected to
the first point
        convex_lines = [QLineF(convex_hull[i], convex_hull[(i + 1) % len(convex_hull)]) for i in
range(len(convex_hull))]
        assert (type(convex_lines) == list and type(convex_lines[0]) == QLineF)
        self.show_hull.emit(convex_lines, (255, 0, 0))

    # send a signal to the GUI thread with the time used to compute the hull
    self.display_text.emit('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 - t3))
    print('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 - t3))
```

2.
Time and Space Complexity

<u>Find_rightmost function</u>
This function takes in the left hull that is a list of n points. It iterates over the list looking for which point has the highest x-value. Therefore its time complexity is ***O(n) where n is the size of the leftside_hull list of points***. It is just iterating over the given list and doesn't store anything new. Thus, its ***space complexity is constant O(1).***

<u>Find_slope function</u>
This function takes in two points to figure out what the slope is. Time complexity here is constant ***O(1) this is because the slope is found by subtracting two coordinates which has nothing to do with the number of points taken in by the program.*** The number of points is what causes this program to take a certain amount of time not subtracting bits. The ***space complexity is O(1) as well because we aren't storing anything new here just returning a value.***

<u>Find_tangent function</u>

This function takes in two hulls, the left and right that are both a list of n points, and the indices of the rightmost point in the left hull list and the leftmost point in right hull list. Its time complexity is ***O(n) because the worst case is that it has to move through all the list of points till the correct tangent is found. N is the size of the hull list.*** The space complexity is ***O(1) since it is a constant stack being used. In other words there are no recursive calls that are increasing the memory being used.***

Combine_hulls function

This function takes in two hulls, the left and right which are both a list of n points. The time complexity is ***O(n) because it calls find_tangent and find_rightmost which are both O(n)*** (see above). The space complexity is ***O(n) where n is the size of the new hull created from merging the left and right hulls together.***

Make_convex function

This function takes in the list of sorted_points. It recursively calls itself to keep dividing them by the midpoint of the list. Since it does this the time complexity is ***O(nlogn) where n is the size of the sorted_points list***. The space complexity is ***O(n) where n is the number of points being stored in the left or right hull with each recursive call.***

ConvexHullSolverThread

This is the main function that calls everything else. It takes in a QThread that contains all of the points n. Thus the overall time complexity is ***O(nlogn) this comes from the sorted function that sorts the points as well as the call to make_convex where n is the number of points passed in.*** The overall space complexity is ***O(n) where n is the number of points passed in. This is because the points are stored in sorted_points by their x value and there are n of them.***

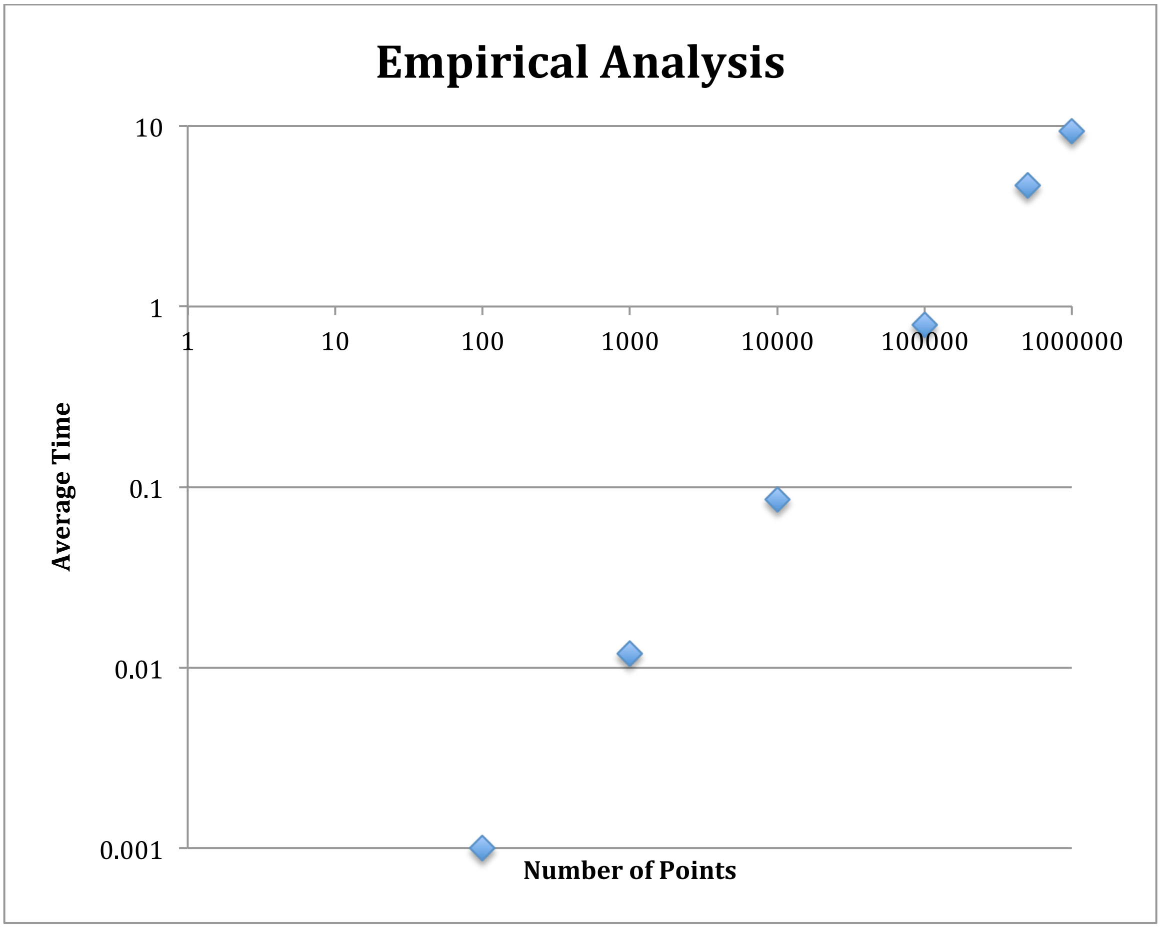
Theoretical Analysis with Recurrence Relation

Worst-case time efficiency is O(nlogn) this is cause of the divide and conquer algorithm used. This theoretical time efficiency comes from the Master Theorem/recurrence relation which is: T(n) = aT(n / b) + O(n^d). The a stands for the number of subproblems. The n is the size of the original problem. The b is how splits are made. And the d is the work at each level. From my algorithm I will get a = 2 (a left and right hull is made at each level) b = 2 (splitting the n in half by cutting the list in half each time) and d = 1 (putting the hulls together will be O(n) because it will require iterating through both n sized hulls). Therefore my master theorem is T(n) = 2T(n / 2) + O(n). log2(2) = 1 since logb(a) = d by the master theorem we can tell that the time complexity will be O(nlogn)

3.

Empirical Analysis

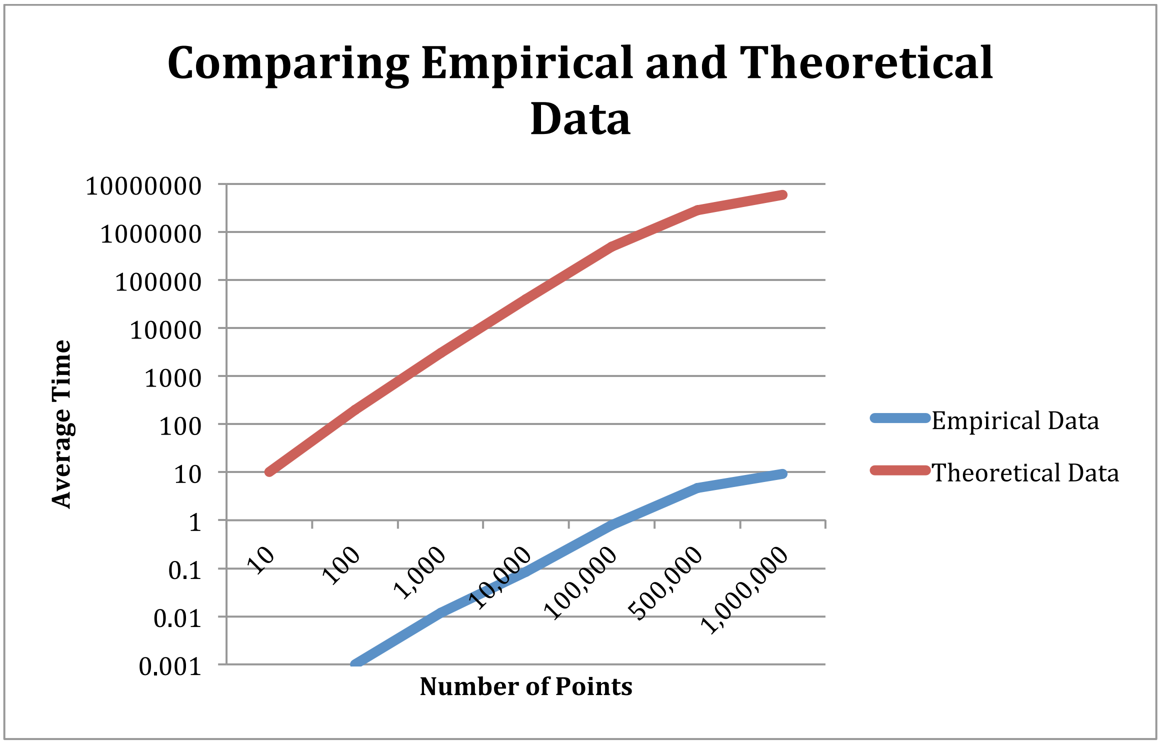
| N points | 10 | 100 | 1,000 | 10,000 | 100,000 | 500,000 | 1,000,000 |
|---|---|---|---|---|---|---|---|
| Test 1 | 0 | 0.001 | 0.012 | 0.085 | 0.786 | 4.7 | 9.333 |
| Test 2 | 0 | 0.001 | 0.012 | 0.089 | 0.806 | 4.644 | 9.345 |
| Test 3 | 0 | 0.001 | 0.012 | 0.084 | 0.786 | 4.653 | 9.401 |
| Test 4 | 0 | 0.001 | 0.012 | 0.084 | 0.782 | 4.666 | 9.392 |
| Test 5 | 0 | 0.001 | 0.012 | 0.084 | 0.782 | 4.738 | 9.332 |
| Average | 0 | 0.001 | 0.012 | 0.0852 | 0.7884 | 4.6802 | 9.3606 |



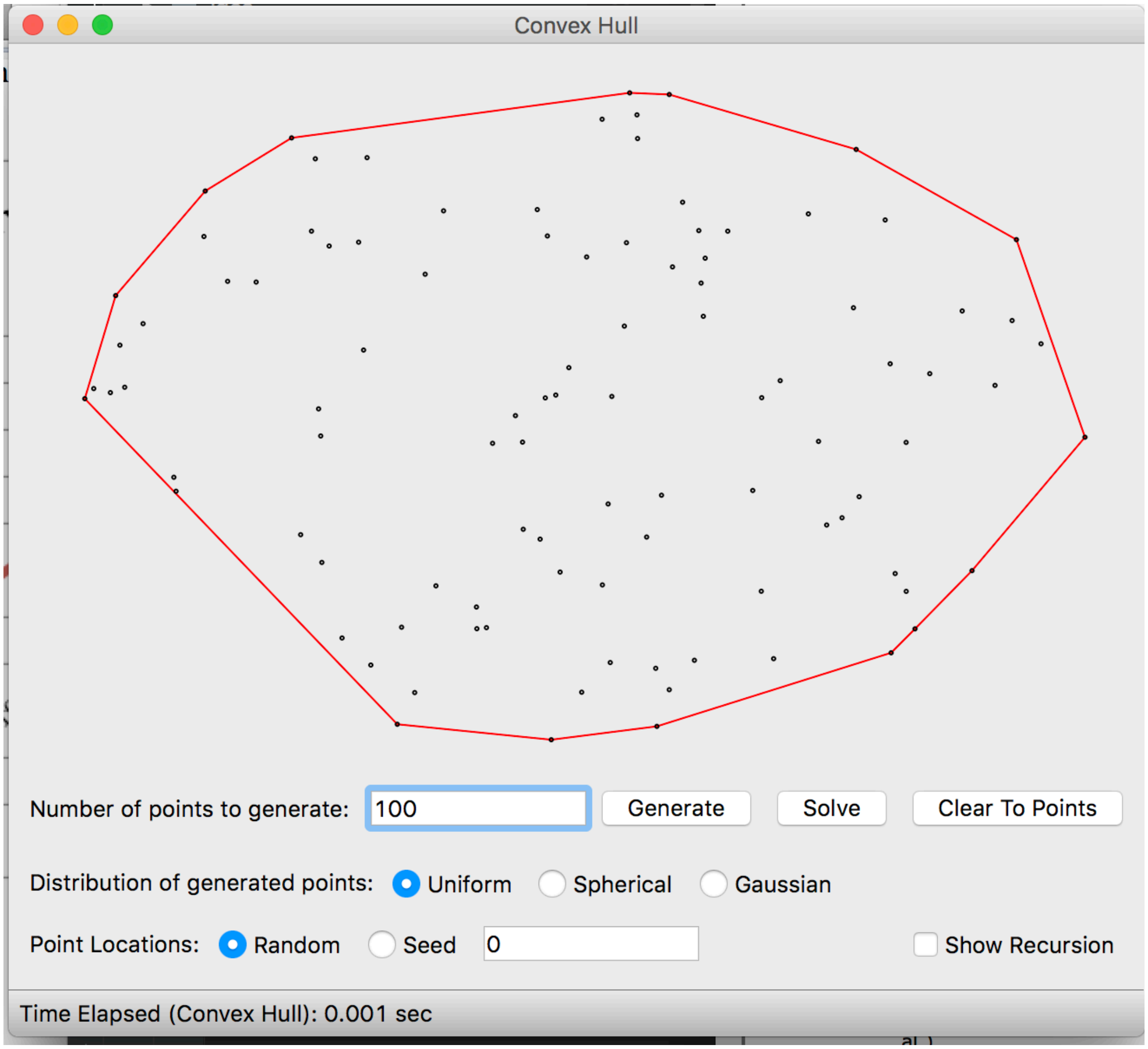

I used a logarithmic scale in the y axis which changed my line to look more linear. This happens because I am normalizing the data for a polynomial space. Normalizing the data helps see the points at 100, and 1000 because if I kept it as a normal graph these points would all look like they don't move at all and stay at zero.

My plot looks very close to a n log n plot as can be seen when graphing both my numbers and what n log n gives (see below). Thus the order of grow that fits best is n log n. As can be seen the graphs differ by a constant. This constant can be found as so.
When n is 10 n log n equals 10. The time it took for 10 points was 0. So, 10x = 0 the constant here would be zero. For 100 n log n is 200x = 0.001. The constant here is 5x10^-6. For 1000 n log n is 3000x = 0.012. The constant here is 4x10^-6. For 10000 n log n is 40000x = 0.0852. The constant here is 2.13x10^-6. For 100000 n log n is 500000x = 0.7884. The constant here is 1.58x10^-6. For 500000 n log n is 2849485x = 4.6802. The constant here is 1.64x10^-6. For 1000000 n log n is 6000000x = 9.3606. The constant here is 1.56x10^-6. Thus, as more points are calculated for the convex hull the proportionality constant rarely changes. In fact for all of the points it hardly changes at all since all of them are 10^-6 which is a very small number. Thus I will take the average of them all to calculate my proportionality constant. This comes out to be 2.27x10^-6.

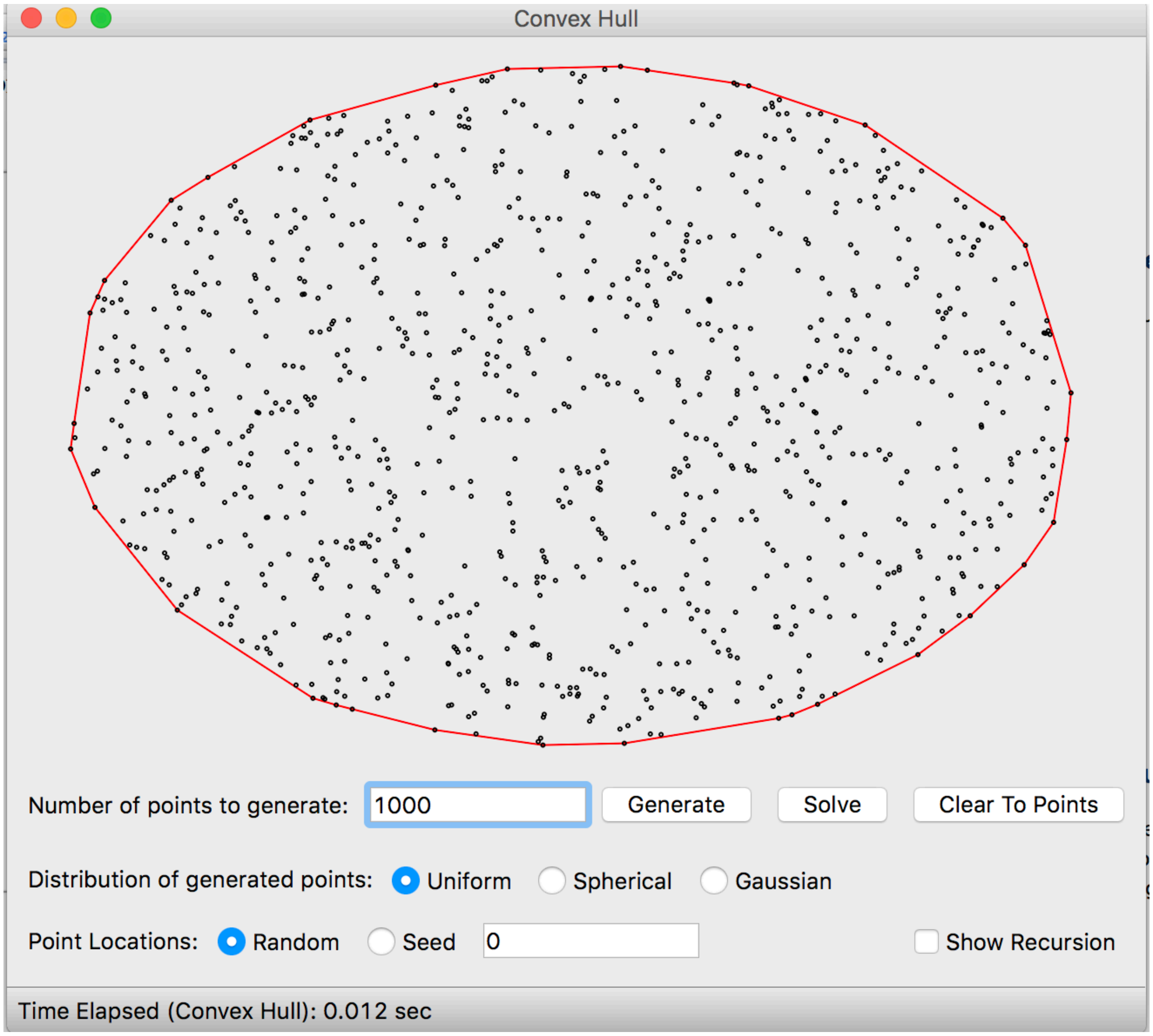
4.
There were no differences seen from my theoretical and empirical analyses. It makes sense that the plot would look n log n because through the master theorem I deduced that in theory it should be. Then through my empirical analysis I was able to plot and compare my data to what an n log n graph would look like. They matched up perfectly and only differed slightly by a very small constant between the two lines. Therefore through my observations of both my theoretical and empirical analyses I can firmly say that my convex hull algorithm runs in O(n log n) time complexity.




5.
Screenshot of 100 points




Screenshot of 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): 0.012 sec