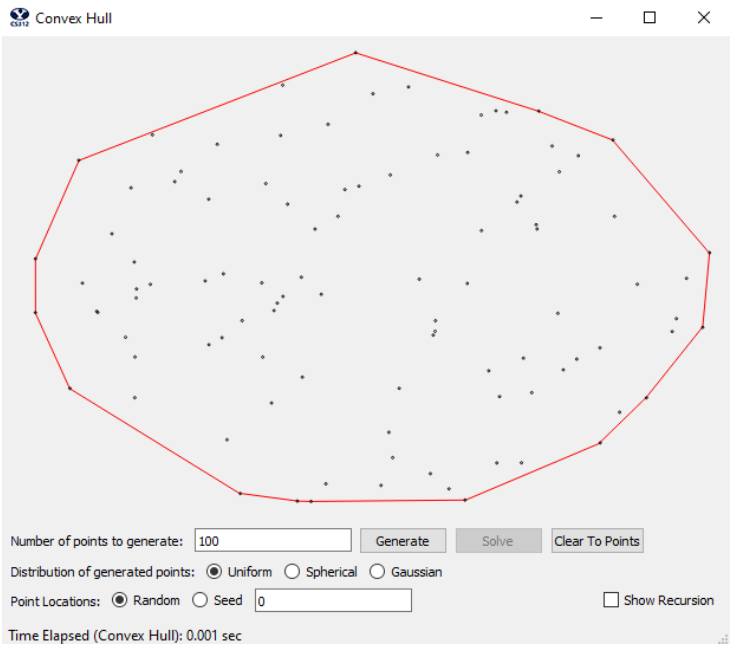
Matthew R. Christensen

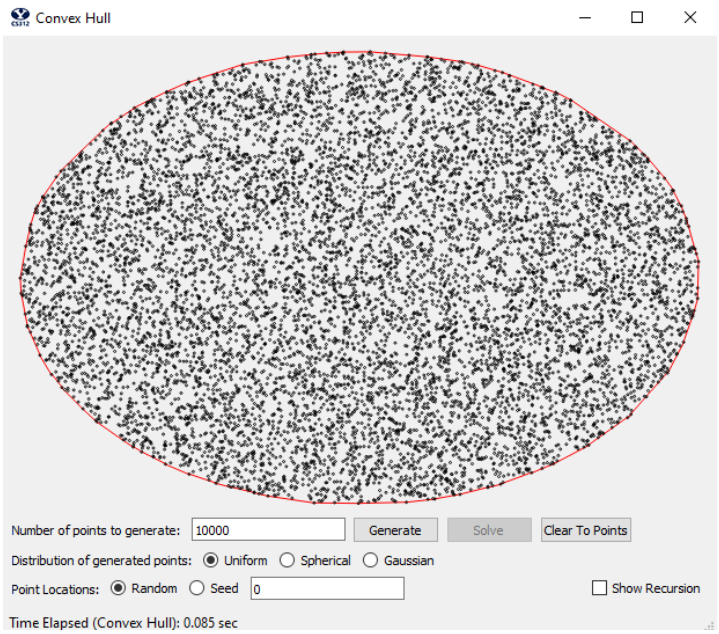
CS 312 (001) – Dr Martinez, Tony R

February 11, 2021

Convex Hull Algorithm Report

Screenshots





Time and Space Complexity

We first will talk about the time and space complexity of the program through the lens of a manual walkthrough of the complexity of the code (using pseudo code), followed by a theoretical approach to the complexity of the code (using the master theorem). If the level of detail is not granular enough in this section (from time and space complexity) please reference the code in the appendix for more detail and specifics with big O.

Manual Analysis

First, we will analyze the complexity of the algorithm through a pseudo code manual assessment. To do so we will use pseudo code to demonstrate the structure and functionality of the algorithm. For a more detailed view of the code, and for more detail and granularity of the big O analysis, see the appendix on this document. Said appendix contains the code, which includes more details of the conclusion of the big O complexity analysis (as well as steps).

Our convex hull solution takes the form of a divide and conquer algorithm structure. Because of the recursive nature of our divide and conquer solution we will analyze the space and time complexity of our solution from the bottom up.

First is our function for finding upper and lower common tangents. A common tangent is a line segment of two convex polygons that intersects each polygon at a single vertex, through the exterior that, if extended forward, will not intersect into the interior of neither polygon. Our code for finding an upper common tangent is as follows:

While edge is not upper_tangent_left and upper_tangent_right:

    While edge is not upper_tangent_left:

Move to next counterclockwise point in left hull

While edge is not upper_tangent_right:

Move to next counterclockwise point in right hull

Because the only number of edges we have to test is at max the number of edges in the previous hull, and because we maintain the ordering of the hull in a counter clockwise position, we are able to find the upper tangents in big O time complexity of n (in the worst case scenario) (three loops of n = O(3n) = O(n)). The space complexity is O(2n) = O(n), which n is the size of either the left or right hull (for completeness, we always assume that n represents the size of the bigger of the two hulls).

The function for finding the lower common tangent is of the same structure, time complexity, and space complexity.

Next are a few helper functions. First we have a function to find the slope of two points. This function uses the normal rise over run equation and consists of just addition and division. Because the numbers are not of arbitrary size than we can assume that both the time and space complexity is constant.

Additionally, we have a function to find the left and right most points in a given hull:

```
index = 0
record = hull[0].x()

# Either direction will have a big o of n because we have to iterate through everything
if direction is "right":
```

```
    for i in range(0, len(hull) - 1):
        if hull[i].x() > record:
            index = i
            record = hull[i].x()
elif direction is "left":
    for i in range(0, len(hull) - 1):
        if hull[i].x() < record:
            index = i
            record = hull[i].x()

return index
```

Because we just loop through the given hull while doing a simple comparison the time and space complexity are both *n | n = num of points in a given hull*.

Next, we have a function to turn an array of points into a line. We use this following function to turn our hulls, which we store as an array of ordered point (counterclockwise), into an array of lines. This function is called at the end of our algorithm to provide the GUI code with lines.

```
lines = []
for i in range(0, len(points)):
    if i == len(points) - 1:
        lines.append(QLineF(points[i], points[0]))
    else:
```

lines.append(QLineF(points[i], points[i + 1]))

return lines

Because we know that in the worst case scenario we have n edges for a given hull, we know that the space and time complexity are of order O(n).

We know look at our function for combining convex hulls togethers:

find_upper_tangent(left_hull, right_hull)

find_lower_tangent(left_hull, right_hull)

combined_hull = []

index = left_hull_start_index

while left_hull[index] != left_hull[left_hull_ending_index]:

index = (index - 1) % len(left_hull)

combined_hull.append(left_hull[index])

index = right_hull_starting_index

while right_hull[index] != right_hull[right_hull_end_index]:

combined_hull.append(right_hull[index])

index = (index - 1) % len(right_hull)

return combined_hull

To combing the hulls we simply iterate through the hulls and add all the edges between the two tangent points. This gets rid of the unnecessary points and is relatively cheap because there is not sorting involved. Furthermore, we avoid the need more future sorting by adding the points to the new "*combined hull*" in the correct order to maintain that relation. We know that the space complexity is O(2n) because at most was have 2 times *n* points (where n is the number of points in either the left or right hull – whichever is greater). And the time complexity is simply O(4n) = O(n) because we have 2 loops of n complexity, and we know from previously that finding the tangents is of n complexity (O(n + n + n + n) = O(4n)= O(n)).

Next, we have our interesting recursive function for solving a hull:

```
if num_points <= 3:  # base case
    if len(points) is 3:
        hull[0] = points[0]
        slope1 = self.get_slope(points[0], points[1])
        slope2 = self.get_slope(points[0], points[2])
        if slope2 > slope1:
            hull[1] = points[2]
            hull[2] = points[1]
        else:
            hull[1] = points[1]
            hull[2] = points[2]
```

```
        return hull
    else:  # if 2 or 1 points
        return points
left_hull = self.solve_hull(points[:num_points//2])
right_hull = self.solve_hull(points[num_points//2:num_points + 1])
return self.combine_hull(left_hull, right_hull)
```

We know that we will call this recursive function a max of log(n) times because each time the size of n is halved. Each recursive call will then have a O(n) because we know that the only significant operation performed by the function is a call to combine the hull, which we found earlier to be O(n). The other functions like *get_slope* are constant time factors and get replaced by the combing function's order of operations as it is bigger. That means that the total time complexity to be O(n) * O(log n) = O(n log n). The space complexity is O(2n) because in the worst case scenario use all of the points to create the convex hull, effectively duplicating the number of points we're storing, *n*.

Finally, we bring it all together in the last function, compute hull:

```
sort(points)
polygon = points_to_line(solve_hull(points))
show(polygon)
```

This function is super simple and acts as the driver for the whole algorithm. First we sort using python's internal sort function. Python uses the timsort algorithm, which has an average performance of O(n log n). We then call solve hull, which we just found to be of O(n log n) complexity, and then points to line, which is of O(n). Thus, we have a time complexity of O(n log n) + O(n log n) + O(n) = O(3n 2 log n) = O(n log n) after simplification. Similaryly, we inherit the space complexity from the solve hull function of a space complexity of O(2n).

Theoretical Analysis

Now we will analyze the algorithm theoretically. We can use the master theorem to get a theoretical hypothesis of our complexity. The master theorem is given as:

$$t(n) = at\left(\frac{n}{b}\right) + O\left(n^{d}\right)$$

Where *a* is the number of sub-tasks that must be solved, *n* is the original task size (or the variable), *n/b* is the size of the sub-instances, and *d* is the polynomial order of work at each node (or the left/partitioning/recombing). This makes up our recurrence relation.

Because, as previously discussed, we split the problem in half, we get that *a* is equal to 2. Next, *b* is 2 because the size of the sub-instances is halved each time as we divide the points into two different groups. Finally, *d* is 1 in our recurrence relation of the divide and conquer approach in our algorithm because our work combining each hull is of the order of n^1 because we only test edges that are part of the previous hull. By doing so, we only have *n* number of edges against *n* number of points and those edges and points form the new hull. Furthermore, we keep our edges ordered counterclockwise, which allows us to merge by finding an upper and lower

tangent for a total complexity on merge of n^1 (the specifics of the tangent algorithm has already been discussing in our manual analysis, for further reference please see the section above). Because we only have to add two new edges, we can add those in constant time, which maintains our complexity of n^2.

We can now take our variables and calculate the big O complexity for our divide and conquer algorithm.

$$t(n) = O\left(n^{d} \log n\right) if \frac{a}{b^{d}} = 1 \Leftrightarrow d = logb(a)$$

Using the equation of above we get that the ratio of our number of sub-taks that must be solved over the factor of size reduction of the sub-instances raised to the power of the order of work done at each node is equal to 1.

$$\frac{a}{b^{d}} = \frac{2}{2^{1}} = 1$$

This implies that our algorithm should be:

$$O\left(n^{d} \log n\right) = O(n \log n)$$

Which correctly coincides with our manual analysis. It is important to note that big O notation allows for the actual algorithm to vary by come constant factor. This will be extensively covered in the coming section.

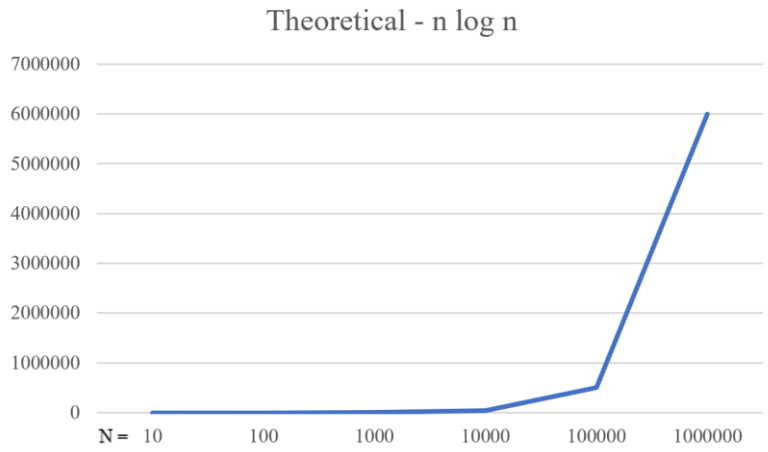
Empirical Result Comparison

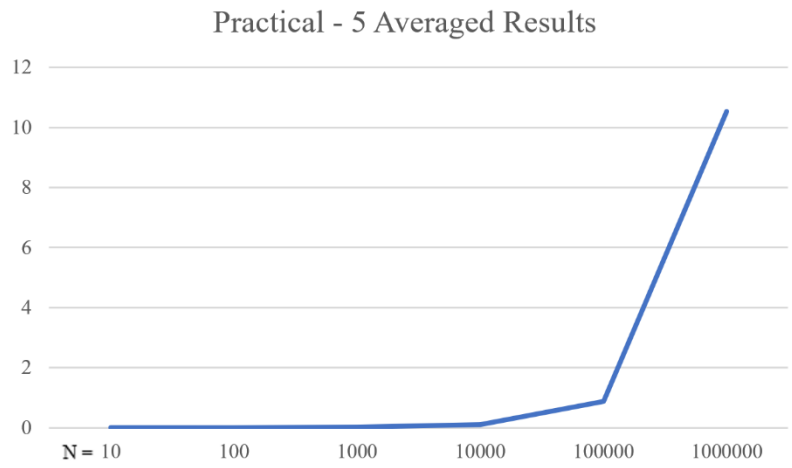
Now that we have defined the complexity of the algorithm both by manual inspection and the theoretical complexity via the master theorem, we can compare what we expect to see against real world results. We let n = {10, 100, 1,000, 10,000, 100,000, 1000,000} and the set *Theoretical* be the list of n log n, i.e. {10, 200, … , 6000000}. We then run our algorithm for every value of n to create a list *Practical*. To create each entry in *Practical* we run our algorithm on a randomly generated list of n nodes five times, and then record the average seconds of completion. The data collected is as follows:

n	Practical	Theoretical	Pratical with K
10	0	10	0
100	0.001	200	570.3655177
1000	0.0112	3000	6388.093798
10000	0.095666667	40000	54564.96786
100000	0.875571429	500000	499395.7511
1000000	10.53233333	6000000	6007279.754

(The induvial test numbers, before averaging, are located in the appendix.)

We then plot the *Theoretical* and *Practical* results to compare the growth with respect to *n*.





What we find is that our algorithm follows the general shape of what we expected: an n log n graph. We know can proceed to finding the constant factor k that our algorithm varies by. We know that our practical results are equal to our theoretical results times some constant, which we define as *k*.

$$Practical = k \times Theoretical$$

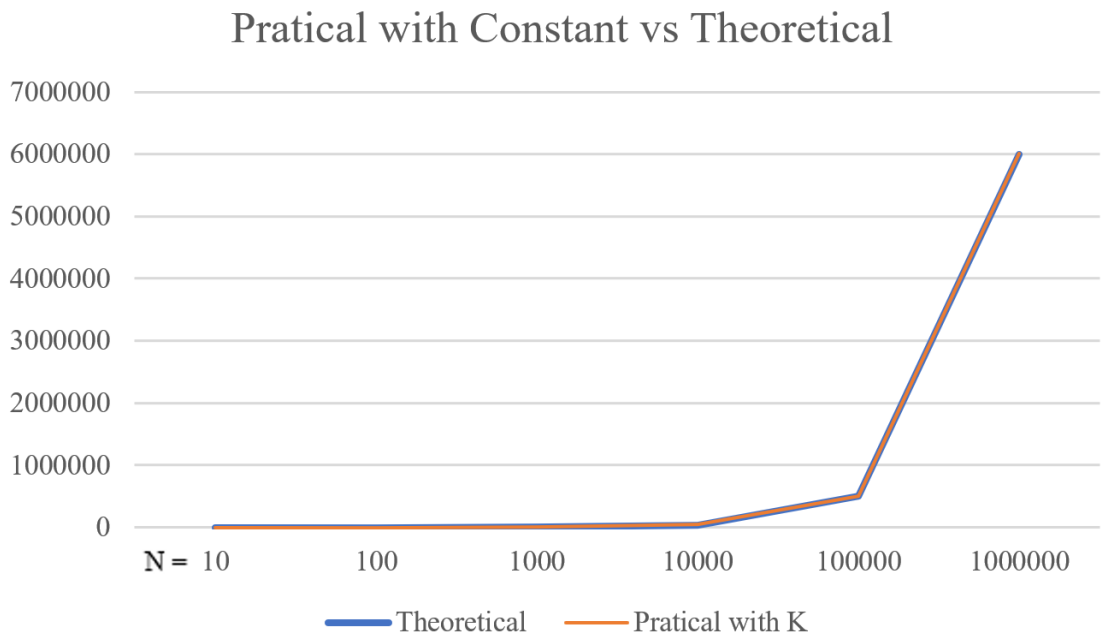
We can simply solve for *k* by finding what our real practical results are divided by the theoretical result of n log n. To find an accurate k we kind two *k* values for relatively high *n* values and take the average (as shown below).

$$k_1 = \frac{\left(Practical\ of\ 10^6 \right)}{\left(10^6 \log 10^6 \right)} = \frac{10.5323}{6000000} = 1.75538333 \times 10^{-6}$$

$$k2 = \frac{\left(Practical\ of\ 10^{5} \right)}{\left(10^{5} \log 10^{5} \right)} = \frac{.87557}{500000} = 1.75114 \times 10^{-6}$$

$$k = \frac{(k1 + k2)}{2} = \frac{\left(1.75538333 \times 10^{-6} + 1.75114\ \square 10 \square 6 \times 10^{-6} \right)}{2} = 1.75326167 \times 10^{-6}$$

This process gives us a *k* of 1.75326167E-06. By multiplying our practical results by the inverse of k we can compare the plots of our practical vs theoretical results.



What we find is that out practical results follow exactly our theoretical prediction of n log n, within a very small range of error. This tells us that that we really do follow the theoretical analysis of a time complexity of bigO(n log n) withing a constant factor. Additionally, we know that we have found a very good constant factor *k* for our analysis by comparing the adjusted practical against the theoretical. Finally, we can reason that because the factor so dominate in

the equation, we know that out complexity is much much smaller smaller than n log n, we run much faster.

Conclusion

We can thus conclude that our algorithm is correctly running at an approximate time complexity of n log n and space complexity of %%%. Furthermore, we know that the constant k is circa 1.75326167E-06, in which our algorithm differs from the theoretical n log n.

Appendix

Practical trial numbers:

n	Trail 1	Trail 2	Trail 3	Trail 4	Trail 5	
10	0	0	0	0	0	(in seconds)
100	0.001	0.001	0.001	0.001	0.001	
1000	0.011	0.01	0.012	0.011	0.012	
10000	0.097	0.096	0.097	0.097	0.093	
100000	0.893	0.888	0.879	0.876	0.859	
1000000	10.971	10.36	10.266	10.125	10.466	

Code:

```
from which_pyqt import PYQT_VER


if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF, QObject
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF, QObject
```

```
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


import time
from functools import cmp_to_key


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

    # Function Time Complexity: We will call this recursive function a max of log(n) times
because each time the size of n is halved.  Each recursive call will then have a O(n + n + n) due
```

*to the function calls and opportations. This combines to be O(3n) * O(log n) = O(3n log n) = O(n log n)*

 # Function Space Complexity: O(2n) because in the worst case scenario we have n edges and points

```
def solve_hull(self, points):

    num_points = len(points)



    if num_points <= 3:  # base case

        if len(points) is 3:

            hull = [None] * 3

            hull[0] = points[0]



            slope1 = self.get_slope(points[0], points[1])

            slope2 = self.get_slope(points[0], points[2])



            if slope2 > slope1:

                hull[1] = points[2]

                hull[2] = points[1]

            else:
```

```
                hull[1] = points[1]

                hull[2] = points[2]


            return hull


        else:  # if 2 or 1 points

            return points


    left_hull = self.solve_hull(points[:num_points//2])

    right_hull = self.solve_hull(points[num_points//2:num_points + 1])


    return self.combine_hull(left_hull, right_hull)


# Function Time Complexity: 4n = n

# Function Space Complexity: 2n = n

def combine_hull(self, left_hull, right_hull):

    upper_tan, left_hull_start_index, right_hull_end_index = self.find_upper_tangent(left_hull,
right_hull)
```

```
        lower_tan, left_hull_ending_index, right_hull_starting_index =
self.find_lower_tangent(left_hull, right_hull)

        # self.showTangent(self.points_to_lines(left_hull), BLUE)
        # self.showTangent(self.points_to_lines(right_hull), BLUE)
        # self.showTangent(self.points_to_lines(upper_tan), YELLOW)
        # self.showTangent(self.points_to_lines(lower_tan), YELLOW)

        combined_hull = []

        found_lower_tangent_point = False

        j = right_hull_end_index

        while found_lower_tangent_point is False:  # bigO(n) bc the furthest distance is the whole
hull
            combined_hull.append(right_hull[j])
            if right_hull[j] == lower_tan[1]:
                found_lower_tangent_point = True
```

```
        j = (j + 1) % len(right_hull)


    i = left_hull_ending_index
    while i is not len(left_hull):  # bigO(n) bc the furthest distance is the whole hull
        combined_hull.append(left_hull[i])
        i = (i + 1) % len(left_hull)


    return combined_hull


# Function Time Complexity: 3n = n (The loops)
# Function Space Complexity: 2n = n (the two halves)
def find_upper_tangent(self, left_hull, right_hull):
    left_tan, right_tan = False, False


    left_hull_index = self.get_point_index(left_hull, "right")
    right_hull_index = self.get_point_index(right_hull, "left")


    current_slope = self.get_slope(left_hull[left_hull_index], right_hull[right_hull_index])
```

```
        # self.showTangent(self.points_to_lines(left_hull), RED)
        # self.showTangent(self.points_to_lines(right_hull), BLUE)

        while not left_tan and not right_tan:
            while not left_tan:  # Total iteration can be all of the left and right hull, thus 2n
                # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])], GREEN)
                left_hull_index = (left_hull_index - 1) % len(left_hull)
                # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])], YELLOW)
                new_slope = self.get_slope(left_hull[left_hull_index], right_hull[right_hull_index])
                if new_slope < current_slope:
                    left_tan, right_tan = False, False
                    current_slope = new_slope
                else:
                    left_tan = True
                    left_hull_index = (left_hull_index + 1) % len(left_hull)  # Since we can't move up anymore we use the last good index
```

```
        while not right_tan: # Total iteration can be n
            # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])],
RED)
            right_hull_index = (right_hull_index + 1) % len(right_hull)
            # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])],
BLUE)
            new_slope = self.get_slope(left_hull[left_hull_index], right_hull[right_hull_index])
            if new_slope > current_slope:
                left_tan, right_tan = False, False
                current_slope = new_slope
            else:
                right_tan = True
                right_hull_index = (right_hull_index - 1) % len(right_hull)  # Since we can't move
up anymore we use the last good index


        # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])],
BLUE)
        return [left_hull[left_hull_index], right_hull[right_hull_index]], left_hull_index,
right_hull_index
```

```
    # Function Time and Space Complexity are the same as the upper tan func (see above)
    def find_lower_tangent(self, left_hull, right_hull):
        left_tan, right_tan = False, False

        left_hull_index = self.get_point_index(left_hull, "right")
        right_hull_index = self.get_point_index(right_hull, "left")

        current_slope = self.get_slope(left_hull[left_hull_index], right_hull[right_hull_index])
        # self.showTangent(self.points_to_lines(left_hull), RED)
        # self.showTangent(self.points_to_lines(right_hull), BLUE)

        while not left_tan and not right_tan:
            while not left_tan:
                # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])],
GREEN)
                left_hull_index = (left_hull_index + 1) % len(left_hull)
                # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])],
YELLOW)
```

```
            new_slope = self.get_slope(left_hull[left_hull_index], right_hull[right_hull_index])
            if new_slope > current_slope:
                left_tan, right_tan = False, False
                current_slope = new_slope
            else:
                left_tan = True
                left_hull_index = (left_hull_index - 1) % len(left_hull)  # Since we can't move up
anymore we use the last good index

        while not right_tan:
            # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])],
RED)
            right_hull_index = (right_hull_index - 1) % len(right_hull)
            # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])],
BLUE)
            new_slope = self.get_slope(left_hull[left_hull_index], right_hull[right_hull_index])
            if new_slope < current_slope:
                left_tan, right_tan = False, False
                current_slope = new_slope
```

```
            else:
                right_tan = True
                right_hull_index = (right_hull_index + 1) % len(right_hull)  # Since we can't move up anymore we use the last good index

        # self.showTangent([QLineF(left_hull[left_hull_index], right_hull[right_hull_index])], BLUE)
        return [left_hull[left_hull_index], right_hull[right_hull_index]], left_hull_index, right_hull_index

    # Function Time Complexity: c
    # Function Space Complexity: c
    def get_slope(self, point1, point2):
        return (point2.y() - point1.y()) / (point2.x() - point1.x())  # Slope = change in y / change in x

    # Function Time Complexity: n
    # Function Space Complexity: n
    def get_point_index(self, hull, direction):
        index = 0
```

```
    record = hull[0].x()

    # Either direction will have a big o of n because we have to iterate through everything
    if direction is "right":
        for i in range(0, len(hull) - 1):
            if hull[i].x() > record:
                index = i
                record = hull[i].x()
    elif direction is "left":
        for i in range(0, len(hull) - 1):
            if hull[i].x() < record:
                index = i
                record = hull[i].x()

    return index

# Function Time Complexity: n
# Function Space Complexity: 2n = n
```

```
    def points_to_lines(self, points):
        lines = []
        for i in range(0, len(points)):
            if i == len(points) - 1:
                lines.append(QLineF(points[i], points[0]))
            else:
                lines.append(QLineF(points[i], points[i + 1]))
        return lines


    # 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)


        def compare(item1,
                    item2):  # https://stackoverflow.com/questions/5213033/sort-a-list-of-lists-with-a-
custom-compare-function
```

if item1.x() < item2.x():

return -1 # return a negative value (< 0) when the left item should be sorted before the right item

elif item2.x() < item1.x():

return 1 # return a positive value (> 0) when the left item should be sorted after the right item

else:

return 0 # return 0 when both the left and the right item have the same weight and should be ordered "equally" without precedence

t1 = time.time()

 # SORT THE POINTS BY INCREASING X-VALUE

sorted_points = sorted(points, key=cmp_to_key(compare)) # Uses timsort algor, which is on average O(n log n)

t2 = time.time()

t3 = time.time()

 # this is a dummy polygon of the first 3 unsorted points

 # polygon = [QLineF(sortedPoints[i], sortedPoints[(i + 1) % 3]) for i in range(3)]

```
    # REPLACE THE LINE ABOVE WITH A CALL TO YOUR DIVIDE-AND-CONQUER
CONVEX HULL SOLVER

    polygon = self.points_to_lines(self.solve_hull(sorted_points))  # n + n +

    t4 = time.time()



    # when passing lines to the display, pass a list of QLineF objects.  Each QLineF

    # object can be created with two QPointF objects corresponding to the endpoints

    self.showHull(polygon, RED)

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