Part 1: See Appendix

Part 2: Time Complexity

divide_and_conquer

```
# O(nlogn)
   def divide_and_conquer(self, points):
       # Base case
       if len(points) <= 3:
           # base case. Trivial convex hull. O(1)
           return self.sort_clockwise(points)

       # Divide the list of points into two halves
       mid = len(points) // 2 # O(1)
       left = points[:mid] # O(n/2), n is the # of points in the hull
       right = points[mid:] #  O(n/2), n is the # of points in the hull

       # Recursive on the left and right halves
       left_hull = self.divide_and_conquer(left)
       right_hull = self.divide_and_conquer(right)
       # Merge the two halves
       return self.merge_hulls(left_hull, right_hull)
```

Explanation:

- The time complexity of this function is O(nlogn), where n is the number of points. The logn part comes from the fact that we are dividing the problem recursively into problems that are half the size and then the merge process takes O(n). The recurrence relation is T(n) = 2T(n/2) + n. Following the master's theorem a/b^d = 1 resulting in O(nlogn).
- The space complexity of this function is also O(nlogn) because the depth of the recursion contributes O(logn) and the merge_hulls requires O(n) space to store to the merged_hulls.

merge_hulls

```
# O(n) for each merge
   def merge_hulls(self, left_hull, right_hull):
       # maintains circularity
       # Find the upper and lower tangents
       upper_tangent = self.find_upper_tangent(left_hull, right_hull) # O(n)
       lower_tangent = self.find_lower_tangent(left_hull, right_hull) # O(n)

       # Merge the two hulls using the upper_tangent and the lower_tangent
       # as a guide
       merged_hull = []

       # Find indices of upper and lower tangent points in left and right hulls
       U1 = left_hull.index(upper_tangent.p1())
       U2 = right_hull.index(upper_tangent.p2())
       L1 = left_hull.index(lower_tangent.p1())
       L2 = right_hull.index(lower_tangent.p2())

       # Traverse the left_hull counter clockwise (i++)
       # O(n), n is the # points apart of the convex hull
```

```
        # on the left side
        i = U1
        while i != L1:
            merged_hull.append(left_hull[i])
            i = (i+1) % len(left_hull)
        merged_hull.append(left_hull[L1])

        # Traverse the right_hull counter clockwise(i++)
        # O(n), n is the # points apart of the convex hull
        # on the right side
        i = L2
        while i != U2:
            merged_hull.append(right_hull[i])
            i = (i+1) % len(right_hull)
        merged_hull.append(right_hull[U2]) # add the end point

        return merged_hull
```

Explanation:

- The time complexity of this function is O(n), where n is the number of points. This is because in the worst case each piece of this function such as upper_tangent, lower_tangent then traversing each side of the hull is at worst case O(n) meaning the worst case scenario is examining all the points. There is no recurrence relation for this function since it is not recursive, but it does contribute to the divide and conquer recurrence relation with +O(n).
- The space complexity of this function is O(n) because in the worst case it requires O(n) space to store the entire hull. This is because in the worst case we store the entire hull, so n represents the number of points in the hull.

<u>Finding upper and lower tangents (same explanation)</u>

```
        # O(n)
    def find_upper_tangent(self, left_hull, right_hull):
        leftmost_point = min(right_hull, key=lambda point: point.x()) # O(n)
        rightmost_point = max(left_hull, key=lambda point: point.x()) # O(n)

        temp = QLineF(rightmost_point, leftmost_point)

        p = left_hull.index(rightmost_point) # O(n), indice of the rightmost_point
        q = right_hull.index(leftmost_point) # O(n), indice of the left_most_point

        done = False
        while not done:
            done = True
            # while temp is not upper tangent to the left hull
            while not self.is_upper_tangent(temp, left_hull): # counter clockwise (++index, % 
for wrap around)
                r = (p+1) % len(left_hull)
```

```
            temp = QLineF(left_hull[r], right_hull[q])
            p = r
            done = False
        while not self.is_upper_tangent(temp, right_hull): # clockwise (--index, % for wrap
around)
            r = (q-1) % len(right_hull)
            temp = QLineF(left_hull[p], right_hull[r])
            q = r
            done = False
    return temp
```

Explanation:

- The time complexity of this function is O(n). This is because finding the extreme points(left and right) takes O(n) time where n represents the number of points in each respective hull. Finding the indices for each of these takes the same amount of time. Finding the upper and lower tangent's worst case could lead to looking through all the points in each respective hull but the average case will be found without having to iterate through each point. Overall the complexity is O(m+N), where m and n are the size of each respective hull, making this just O(n) for the time complexity.
- The space complexity of this function is O(1) because everything that is stored in this function does not scale with the input and uses a fixed amount of space.

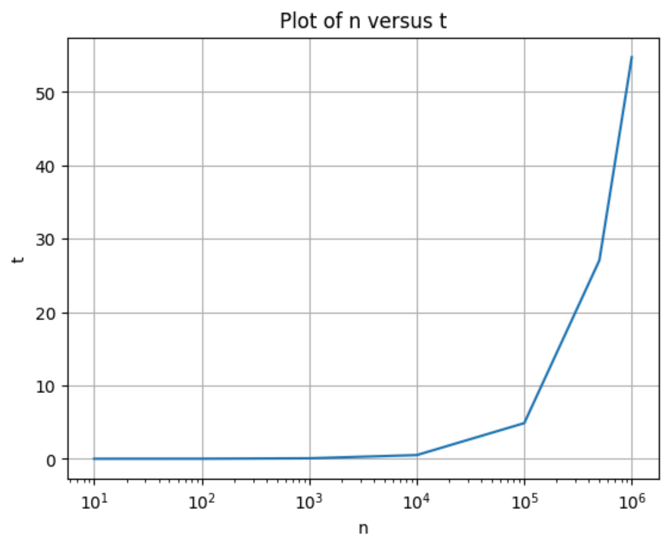
<u>Is upper and lower tangent (same explanation)</u>

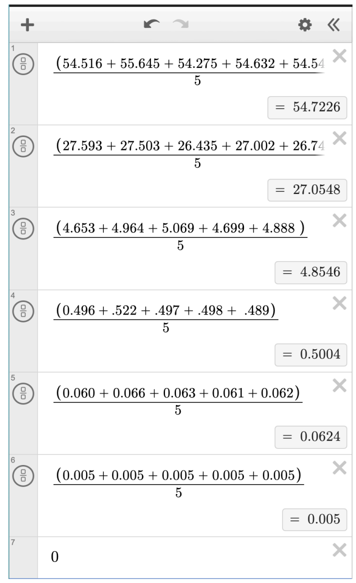
```
# O(n) where n is the number of point in the hull given
   def is_upper_tangent(self, line, hull):
       A = line.p1()
       B = line.p2()

       for P in hull:
           # Calculate cross product (AB x AP)
           cross_product = ((B.x() - A.x()) * (P.y() - A.y())) - ((B.y() - A.y()) * (P.x() -
A.x()))

           if cross_product > 0:
               return False  # Not an upper tangent

       return True # All points are below or on the line
```

Explanation:

- The time complexity of this function is O(n) where n is the # of points in the respective hull. In the worst case this function will iterate over every point in the hull when the line is indeed a tangent of the hull. The cross product calculation is O(1) so the overall complexity is O(1 + n) making it just O(n).
- The space complexity of this function is O(1) because all the things stored in this function take up a fixed amount of space and do not grow with the size of the input.

Part 3: Experimentation





Discussion:

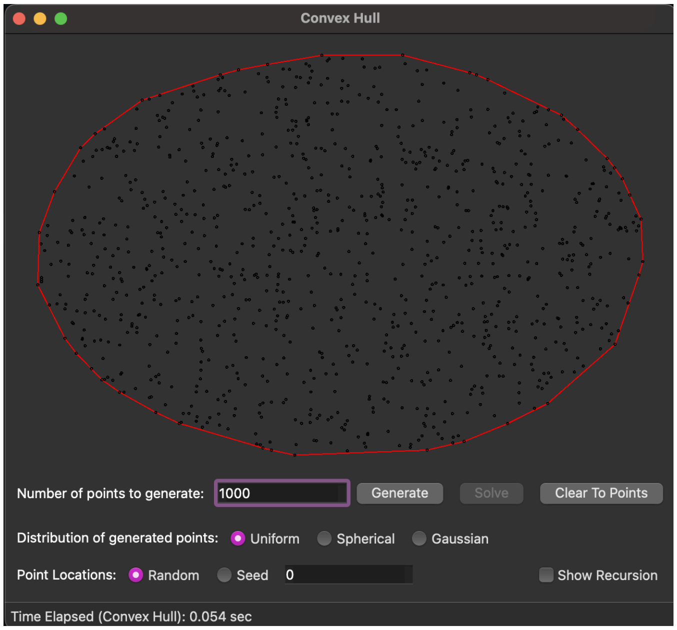
- Explanation:
  - The reason the distribution is shaped this way with the curve is because we are using logarithmic scale which gives us more of an insight into the behavior or the program as the # points rise. When not using a logarithmic scale the shape is linear which deceives the true behavior of the program as the input increases. This shows how for smaller values of n, the constant factors and lower-order terms in the algorithm's runtime are more significant. As n increases, the nlogn behavior becomes dominant, leading to a steeper increase in runtime.
- Which order of growth fits best?
  - Based on the explanation above we can say the order of the growth fits with nlogn.
- Estimate of constant of proportionality:
  - Looking at the curve my estimate for the constant of proportionality is 45 because of measuring the slope at the steepest part of the graph which is from 10^5 to 10^6. This means that for every one-unit increase in log(n) base 10, the dependent variable T increases by 45 units.

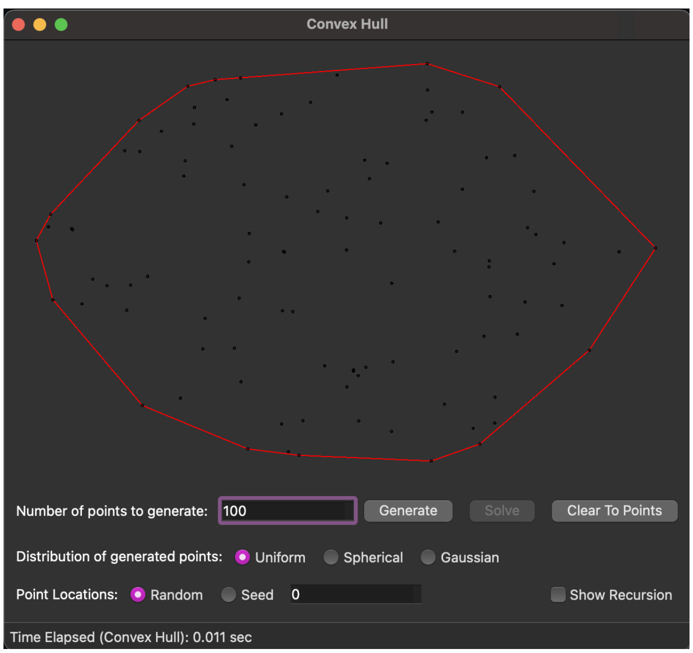
Part 4: Empirical analyses

Discussion:

- The initial flatness of the curve for lower n values in the empirical data does not initially support the theoretical O(nlogn) but as you can see it exponentially grows between 10^4 and 10^6 which is consistent with O(nlogn) complexity.

Part 5: Screenshots of working examples





Appendix

```
class ConvexHullSolver(QObject):

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

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

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

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

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

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

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

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

# This is the method that gets called by the GUI and actually executes
# the finding of the hull
   def compute_hull( self, points, pause, view):
       self.pause = pause
       self.view = view
       assert( type(points) == list and type(points[0]) == QPointF )

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

       t2 = time.time()

       t3 = time.time()
       convex_hull_points = self.divide_and_conquer(points)
```

```python
        t4 = time.time()

        polygon = self.convert_points_to_lines(convex_hull_points)
        # 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))

    # O(n)
    def convert_points_to_lines(self, sorted_points): # O(n)
        lines = []
        for i in range(len(sorted_points)):
            lines.append(QLineF(sorted_points[i], sorted_points[(i+1) % len(sorted_points)]))

        return lines

    # O(nlogn)
    def divide_and_conquer(self, points):
        # Base case
        if len(points) <= 3:
            # base case. Trivial convex hull. O(1)
            return self.sort_clockwise(points)

        # Divide the list of points into two halves
        mid = len(points) // 2 # O(1)
        left = points[:mid] # O(n/2), n is the # of points in the hull
        right = points[mid:] #  O(n/2), n is the # of points in the hull

        # Recursive on the left and right halves
        left_hull = self.divide_and_conquer(left)
        right_hull = self.divide_and_conquer(right)
        # Merge the two halves
        return self.merge_hulls(left_hull, right_hull)

    # O(n)
    def merge_hulls(self, left_hull, right_hull):
        # maintains circularity
        # Find the upper and lower tangents
        upper_tangent = self.find_upper_tangent(left_hull, right_hull) # O(n)
        lower_tangent = self.find_lower_tangent(left_hull, right_hull) # O(n)

        # Merge the two hulls using the upper_tangent and the lower_tangent
        # as a guide
        merged_hull = []

        # Find indices of upper and lower tangent points in left and right hulls
        U1 = left_hull.index(upper_tangent.p1())
        U2 = right_hull.index(upper_tangent.p2())
        L1 = left_hull.index(lower_tangent.p1())
        L2 = right_hull.index(lower_tangent.p2())

        # Traverse the left_hull counter clockwise (i++)
```

```python
        # O(n), n is the # points apart of the convex hull
        # on the left side
        i = U1
        while i != L1:
            merged_hull.append(left_hull[i])
            i = (i+1) % len(left_hull)
        merged_hull.append(left_hull[L1])

        # Traverse the right_hull counter clockwise(i++)
        # O(n), n is the # points apart of the convex hull
        # on the right side
        i = L2
        while i != U2:
            merged_hull.append(right_hull[i])
            i = (i+1) % len(right_hull)
        merged_hull.append(right_hull[U2]) # add the end point

        return merged_hull

    # O(n)
    def find_upper_tangent(self, left_hull, right_hull):
        leftmost_point = min(right_hull, key=lambda point: point.x()) # O(n)
        rightmost_point = max(left_hull, key=lambda point: point.x()) # O(n)

        temp = QLineF(rightmost_point, leftmost_point)

        p = left_hull.index(rightmost_point) # O(n), indice of the rightmost_point
        q = right_hull.index(leftmost_point) # O(n), indice of the left_most_point

        done = False
        while not done:
            done = True
            # while temp is not upper tangent to the left hull
            while not self.is_upper_tangent(temp, left_hull): # counter clockwise (++index, % for wrap around)
                r = (p+1) % len(left_hull)
                temp = QLineF(left_hull[r], right_hull[q])
                p = r
                done = False
            while not self.is_upper_tangent(temp, right_hull): # clockwise (--index, % for wrap around)
                r = (q-1) % len(right_hull)
                temp = QLineF(left_hull[p], right_hull[r])
                q = r
                done = False
        return temp

    # O(n)
    def find_lower_tangent(self, left_hull, right_hull):
        leftmost_point = min(right_hull, key=lambda point: point.x()) # O(n)
        rightmost_point = max(left_hull, key=lambda point: point.x()) # O(n)
```

```
        temp = QLineF(rightmost_point, leftmost_point)

        p = left_hull.index(rightmost_point) # O(n), indice of the rightmost_point
        q = right_hull.index(leftmost_point) # O(n), indice of the left_most_point

        done = False
        while not done:
            done = True
            # while temp is not lower tangent to the left hull
            while not self.is_lower_tangent(temp, left_hull): # clockwise (--index, % for
wrap around)
                r = (p-1) % len(left_hull)
                temp = QLineF(left_hull[r], right_hull[q])
                p = r
                done = False
            while not self.is_lower_tangent(temp, right_hull): # counter clockwise (++index,
% for wrap around)
                r = (q+1) % len(right_hull)
                temp = QLineF(left_hull[p], right_hull[r])
                q = r
                done = False
        return temp

    # O(n) where n is the number of point in the hull given
    def is_upper_tangent(self, line, hull):
        A = line.p1()
        B = line.p2()

        for P in hull:
            # Calculate cross product (AB x AP)
            cross_product = ((B.x() - A.x()) * (P.y() - A.y())) - ((B.y() - A.y()) * (P.x() -
A.x()))

            if cross_product > 0:
                return False  # Not an upper tangent

        return True # All points are below or on the line

    # O(n) where n is the number of points in the hull given
    def is_lower_tangent(self, line, hull):
        A = line.p1()
        B = line.p2()

        for P in hull:
            # Calculate cross product (AB x AP)
            cross_product = ((B.x() - A.x()) * (P.y() - A.y())) - ((B.y() - A.y()) * (P.x() -
A.x()))

            if cross_product < 0:
                return False  # Not an lower tangent

        return True # All points are above or on the line
```

```python
# O (nlogn),
# but only sort once at base case 3 so essentially O(1)
def sort_clockwise(self, points):
    centroid = self.calculate_centroid(points) # (avg position of each point)
    def polar_angle(point):
    # Calculate the polar angle of the point relative to the centroid
        return math.atan2(point.y() - centroid.y(), point.x() - centroid.x())

    # Sort the points based on the polar angle with respect to the centroid
    sorted_points = sorted(points, key=polar_angle) # O(nlogn) (O(1) for 3)

    return sorted_points
# helper method
def calculate_centroid(self, points): # average position of all the points
    x_sum, y_sum = 0, 0
    for p in points:
        x_sum += p.x()
        y_sum += p.y()

    return QPointF(x_sum / len(points), y_sum / len(points))
```