# Keep dividing the points by half and call merge function recursively  
# Time complexity of this divide and conquer algorithm would be O(nlogn) by the Master Theorem  
# Space complexity would be O(n)  
def compute\_hull(self, points\_set):  
 # Base case n == 2 or n == 3  
 if len(points\_set) == 2:  
 return points\_set  
 # When n == 3, reorder the points in the list if needed.  
 if len(points\_set) == 3:  
 return self.rearrange(points\_set)  
  
 # Dividing part of the algorithm  
 left = self.compute\_hull(points\_set[:len(points\_set) // 2])  
 right = self.compute\_hull(points\_set[len(points\_set) // 2:])  
  
 # Merge  
 return self.make\_hull(left, right)  
  
# Compare the two slopes from point[0], and change the order if the second slope is bigger than the first slope  
# in order to keep the order same throughout the merging part.  
# Takes O(1) time and space complexity.  
def rearrange(self, points):  
 slope1 = self.get\_slope(points[0], points[1])  
 slope2 = self.get\_slope(points[0], points[2])  
  
 # Swap point[1] and point[2]  
 if slope1 < slope2:  
 temp = points[1]  
 points[1] = points[2]  
 points[2] = temp  
  
 # Returning the reordered point set  
 return points  
  
# This part will take O(n) time / space complexity  
def make\_hull(self, left, right):  
  
 # Find the rightmost index from left hull  
 closest = 0  
 rightmost\_point = -300  
 for i in range(len(left)):  
 if left[i].x() > rightmost\_point:  
 rightmost\_point = left[i].x()  
 closest = i  
 rightmost\_index = closest  
  
 # Find the leftmost index from right hull  
 closest = 0  
 leftmost\_point = 300  
 for i in range(len(right)):  
 if right[i].x() < leftmost\_point:  
 leftmost\_point = right[i].x()  
 closest = i  
 leftmost\_index = closest  
  
 # Get upper / lower tangent (4 points)  
 up\_left\_tangent, up\_right\_tangent = self.get\_upper\_tangent(left, right, rightmost\_index, leftmost\_index)  
 low\_left\_tangent, low\_right\_tangent = self.get\_lower\_tangent(left, right, rightmost\_index, leftmost\_index)  
  
 # Make temp list to return

# should take O(n) time using cut and paste method  
 temp\_list = []  
 for index1 in range(len(left)):  
 temp\_list.append(left[index1])  
  
 if up\_left\_tangent == index1 % len(left):  
  
 for index2 in range(up\_right\_tangent, up\_right\_tangent + len(right)):  
 temp\_list.append(right[index2 % len(right)])  
  
 if low\_right\_tangent == index2 % len(right):  
  
 for index3 in range(low\_left\_tangent, low\_left\_tangent + len(left)):  
  
 if (index3 % len(left)) != 0:  
 temp\_list.append(left[index3 % len(left)])  
 else:  
 break  
 break  
 break  
 return temp\_list  
  
# Calculate slope from given two points  
# Takes O(1) time and space  
def get\_slope(self, left, right):  
 return (right.y() - left.y()) / (right.x() - left.x())  
  
# Get upper tangent indices by calling get\_low\_right\_tangent / get\_low\_left\_tangent function  
# Keep comparing new left/right index and return when they are the same  
# O(n) time / space complexity  
def get\_lower\_tangent(self, left, right, rightmost\_index, leftmost\_index):  
  
 index\_left = rightmost\_index  
 index\_right = leftmost\_index  
 while True:  
 current\_index\_right = self.get\_low\_right\_tangent(left, right, index\_left, index\_right)  
 current\_index\_left = self.get\_low\_left\_tangent(left, right, index\_left, index\_right)  
  
 # if new indices are not changed from the previous indices, return. Otherwise, update the current indices.  
 if index\_left == current\_index\_left and index\_right == current\_index\_right:  
 break  
 else:  
 index\_left = current\_index\_left  
 index\_right = current\_index\_right  
 return index\_left, index\_right  
  
# O(n) time / space complexity  
def get\_low\_left\_tangent(self, left, right, rightmost\_index, leftmost\_index):  
  
 slope = self.get\_slope(left[rightmost\_index], right[leftmost\_index])  
 temp = 0  
  
 while True:  
 # Calculate slope between leftmost index and next clockwise point from left hull  
 next\_slope = self.get\_slope(left[(rightmost\_index + 1 + temp) % len(left)], right[leftmost\_index])  
  
 # if current slope is bigger than the next slope, we're done. Otherwise, update the current slope.  
 if slope > next\_slope:  
 break  
 else:  
 temp = temp + 1  
 slope = next\_slope  
 return (rightmost\_index + temp) % len(left)  
  
# O(n) time / space complexity  
def get\_low\_right\_tangent(self, left, right, rightmost\_index, leftmost\_index):  
 slope = self.get\_slope(left[rightmost\_index], right[leftmost\_index])  
 temp = 0  
  
 while True:  
 # Calculate slope between rightmost index and next counterclockwise point from right hull  
 next\_slope = self.get\_slope(left[rightmost\_index], right[(leftmost\_index - temp - 1) % len(right)])  
  
 # if current slope is smaller than the next slope, we're done. Otherwise, update the current slope.  
 if slope < next\_slope:  
 break  
 else:  
 temp = temp + 1  
 slope = next\_slope  
 return (leftmost\_index - temp) % len(right)  
  
# Get upper tangent indices by calling get\_up\_right\_tangent / get\_up\_left\_tangent function  
# Continue until finding the tangent line from both left and right hull  
# O(n) time / space complexity  
def get\_upper\_tangent(self, left, right, rightmost\_index, leftmost\_index):  
 index\_left = rightmost\_index  
 index\_right = leftmost\_index  
 while True:  
 current\_index\_right = self.get\_up\_right\_tangent(left, right, index\_left, index\_right)  
 current\_index\_left = self.get\_up\_left\_tangent(left, right, index\_left, index\_right)  
  
 # if new indices are not changed from the previous indices, return. Otherwise, update the current indices  
 if index\_left == current\_index\_left and index\_right == current\_index\_right:  
 break  
 else:  
 index\_right = current\_index\_right  
 index\_left = current\_index\_left  
 return index\_left, index\_right  
  
# O(n) time / space complexity  
def get\_up\_right\_tangent(self, left, right, rightmost\_index, leftmost\_index):  
 slope = self.get\_slope(left[rightmost\_index], right[leftmost\_index])  
 temp = 0  
 while True:  
 # Calculate the slope between rightmost index and next clockwise point from right hull  
 next\_slope = self.get\_slope(left[rightmost\_index], right[(leftmost\_index + 1 + temp) % len(right)])  
 # if current slope is bigger than the next slope, we're done. Otherwise, update the current slope.  
 if next\_slope < slope:  
 break  
 else:  
 temp = temp + 1  
 slope = next\_slope  
 return leftmost\_index + temp  
  
# O(n) time / space complexity  
def get\_up\_left\_tangent(self, left, right, rightmost\_index, leftmost\_index):  
 slope = self.get\_slope(left[rightmost\_index], right[leftmost\_index])  
 temp = 0  
 while True:  
 # Calculate the slope between leftmost index and next counterclockwise point from left hull  
 next\_slope = self.get\_slope(left[(rightmost\_index - temp - 1) % len(left)], right[leftmost\_index])  
  
 # if current slope is smaller than the next slope, we're done. Otherwise, update the current slope.  
 if next\_slope > slope:  
 break  
 else:  
 temp = temp + 1  
 slope = next\_slope  
 return (rightmost\_index - temp) % len(left)

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()  
 # *TODO: SORT THE POINTS BY INCREASING X-VALUE* # Sorting the given set of points by using x value as key  
 sorted\_points = sorted(self.points, key=lambda p: p.x())  
  
 t2 = time.time()  
 print('Time Elapsed (Sorting): {:3.3f} sec'.format(t2-t1))  
  
 t3 = time.time()  
 # *TODO: COMPUTE THE CONVEX HULL USING DIVIDE AND CONQUER* # Calling the divide & conquer algorithm  
 points = self.compute\_hull(sorted\_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,(0,255,0))  
  
 else:  
 # *TODO: PASS THE CONVEX HULL LINES BACK TO THE GUI FOR DISPLAY* polygon = [QLineF(points[i], points[(i+1)%len(points)]) for i in range(len(points))]  
 assert( type(polygon) == list and type(polygon[0]) == QLineF)  
 self.show\_hull.emit(polygon, (0,255,0))  
  
 # 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. Analysis of Algorithm

- The whole algorithm will take O(nlogn) by the Master Theorem. We divide into 2 subproblems, so a = 2, and size is n / 2, so b = 2. Since the merging part would take linear time, d = 1. Therefore, the Master Theorem gives O(nlogn) time complexity. For space complexity, the space needed grows linearly in the algorithm, mostly the lists to store the sub hulls. Therefore, it has O(n) space complexity.

3. Empirical analysis

1) n = 10 (0.000s, 0.000s, 0.001s, 0.001s, 0.001s) mean time required = 0.0006s

2) n = 100 (0.009s, 0.001s, 0.005s, 0.006s, 0.005) mean time required = 0.0052s

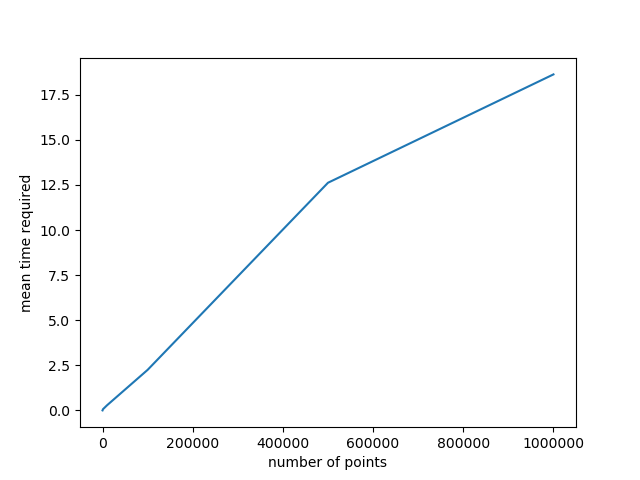
3) n = 1,000 (0.044s, 0.072s, 0.068s, 0.084s, 0.066s) mean time required = 0.0668s

4) n = 10,000 (0.272s, 0.262s, 0.267s, 0.337s, 0.289s) mean time required = 0.2854s

5) n = 100,000 (1.995s, 2.089s, 2.403s, 2.591s, 2.165s) mean time required = 2.2486s

6) n = 500,000 (13.177s, 12.533s, 14.023s, 14.375s, 8.996s) mean time required = 12.621s

7) n = 1,000,000 (18.397s, 18.208s, 19.678s, 18.631s, 18.22s) mean time required = 18.627s



* The graph is showing it has O(nlogn) time complexity. Since the number of inputs’ scale is logarithmic and the graph increases linearly, it is O(nlogn).

1) n = 10, 0.0006 = k \* 10 log 10, k = 0.00006

2) n = 100, 0.0052 = k \* 100 log 100, k = 0.000026

3) n = 1000, 0.0668 = k \* 1000 log 1000, k = 0.000022

4) n = 10000, 0.2854 = k \* 10000 log 10000, k = 0.0000071

5) n = 100000, 2.2486 = k \* 100000 log 100000, k = 0.0000045

6) n = 500000, 12.621 = k \* 500000 log 500000, k = 0.0000044

7) n = 1000000, 18.627 = k \* 1000000 log 1000000, k = 0.0000031

The values of constant proportionality k are tiny that it doesn’t really affect the time. So, it confirms that nlogn best fits for g(n) in this case.

