Trevor Ashby

Convex Hull

1. Correct Functioning Code

```python
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
import math

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

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

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

######################
# Time Complexity O(1)
# Space Complexity O(1)
######################
def findSlope(leftMostPoint, point):
   rise = point.y() - leftMostPoint.y()
   run = point.x() - leftMostPoint.x()
   slope = rise / run
   return slope



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)


   ################################
   # Time Complexity O(nlogn)
   # Space Complexity O(nlogn)
   ################################
   def solver(self, points):
      # do recursion until only 3 or less points are left
      if len(points) == 3:
         rightmostPoint = points[len(points)-1]
         #########################
         # Time Complexity O(n)
         # Space Complexity O(n)
         #########################
         sortedPoints = [points[0]] + sorted(points[1:], reverse=True, key=lambda p: findSlope(points[0], p))
         return sortedPoints, rightmostPoint
      elif len(points) == 2:
         rightmostPoint = points[len(points) - 1]
         return points, rightmostPoint

      # split points into two halves, left and right
      ########################
      # Time Complexity O(n)
      # Space Complexity O(n)
      ########################
      left = [left for left in points[0:math.floor(len(points)/2)]]
      right = [right for right in points[math.floor(len(points)/2):len(points)]]

      # recursively run both halves
      left_half, rightmostPoint_left = self.solver(left)
      right_half, rightmostPoint_right = self.solver(right)

      #########################
      # Time Complexity O(n)
      # Space Complexity O(n)
      #########################
      if (len(left_half) <= 3):
         left_half = [left_half[0]] + sorted(left_half[1:], reverse=True, key=lambda p: findSlope(left_half[0], p))

      if (len(right_half) <= 3):
         right_half = [right_half[0]] + sorted(right_half[1:], reverse=True, key=lambda p: findSlope(right_half[0], p))


      # set the left/right most points
      leftMost_left = left_half[0]
      leftMost_right = right_half[0]

      # begin merging the Hulls
      # top
      slope = findSlope(rightmostPoint_left, leftMost_right)
      topLine_right = leftMost_right
      topLine_left = rightmostPoint_left

      leftPos = left_half.index(rightmostPoint_left)
      rightPos = right_half.index(leftMost_right)
      numChanges = 3
      ########################
      # Time Complexity O(n)
      # Space Complexity O(n)
      ########################
      while(numChanges > 0):
         leftChange = True
         while(leftChange):
            if (findSlope(topLine_left, topLine_right) > findSlope(left_half[leftPos - 1], topLine_right)):
               topLine_left = left_half[leftPos - 1]
               leftPos -= 1
               numChanges += 1
            else:
               leftChange = False
               numChanges -= 1

         rightChange = True
         while(rightChange):
            if (findSlope(topLine_left, topLine_right) < findSlope(topLine_left, right_half[(rightPos + 1) %
len(right_half)])):
               topLine_right = right_half[(rightPos + 1) % len(right_half)]
               rightPos += 1
               numChanges += 1
            else:
               rightChange = False
               numChanges -= 1

      # bot
      leftPos = left_half.index(rightmostPoint_left)
      rightPos = right_half.index(leftMost_right)
      botLine_right = leftMost_right
      botLine_left = rightmostPoint_left
      numChanges = 3
      ########################
      # Time Complexity O(n)
      # Space Complexity O(n)
      ########################
      while(numChanges > 0):
         leftChange = True
         while(leftChange):
            if (findSlope(botLine_left, botLine_right) < findSlope(left_half[(leftPos + 1) % len(left_half)], botLine_right)):
               botLine_left = left_half[(leftPos + 1) % len(left_half)]
               leftPos += 1
               numChanges += 1
            else:
               leftChange = False
               numChanges -= 1

         rightChange = True
         while(rightChange):
            if (findSlope(botLine_left, botLine_right) > findSlope(botLine_left, right_half[rightPos - 1])):
               botLine_right = right_half[rightPos - 1]
               rightPos -= 1
               numChanges += 1
            else:
               rightChange = False
               numChanges -= 1

      points_to_return = []
      ########################
      # Time Complexity O(n)
      # Space Complexity O(n)
      ########################
      for i in range(len(left_half)):
         if (left_half[i % len(left_half)] == topLine_left):
            points_to_return.append(left_half[i % len(left_half)])
            break
         else:
            points_to_return.append(left_half[i % len(left_half)])
      ########################
      # Time Complexity O(n)
      # Space Complexity O(n)
      ########################
      for i in range(len(right_half)):
         j = i + right_half.index(topLine_right)
         if (right_half[j % len(right_half)] == botLine_right):
            points_to_return.append(right_half[j % len(right_half)])
            break
         else:
            points_to_return.append(right_half[j % len(right_half)])
      ########################
      # Time Complexity O(n)
      # Space Complexity O(n)
      ########################
      for i in range(len(left_half)):
         j = i + left_half.index(botLine_left)
         if (j > len(left_half)):
            break
         if (left_half[j % len(left_half)] == left_half[0]):
            break
         else:
            points_to_return.append(left_half[j % len(left_half)])

      return points_to_return, rightmostPoint_right



# 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()
      # TODO: SORT THE POINTS BY INCREASING X-VALUE

      ##########################
      # Time Complexity O(nlogn)
      # Space Complexity O(n)
      ##########################
      points = sorted(points, key=lambda k: k.x())
      t2 = time.time()

      t3 = time.time()
      # this is a dummy polygon of the first 3 unsorted points
      # polygon = [QLineF(points[i],points[(i+1)%3]) for i in range(3)]
      returnedPoints, rightMostPoint = self.solver(points)
      polygon = [QLineF(returnedPoints[i], returnedPoints[(i + 1) % len(returnedPoints)]) for i in range(len(returnedPoints))]
      # TODO: REPLACE THE LINE ABOVE WITH A CALL TO YOUR DIVIDE-AND-CONQUER CONVEX HULL SOLVER
      t4 = time.time()
      self.showHull(polygon, RED)
      # 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.showText('Time Elapsed (QuickSort): {:3.14f}'.format(t2-t1) + '  Time Elapsed (Convex Hull): {:3.14f}
sec'.format(t4-t3))
```

2. Time and Space Complexity

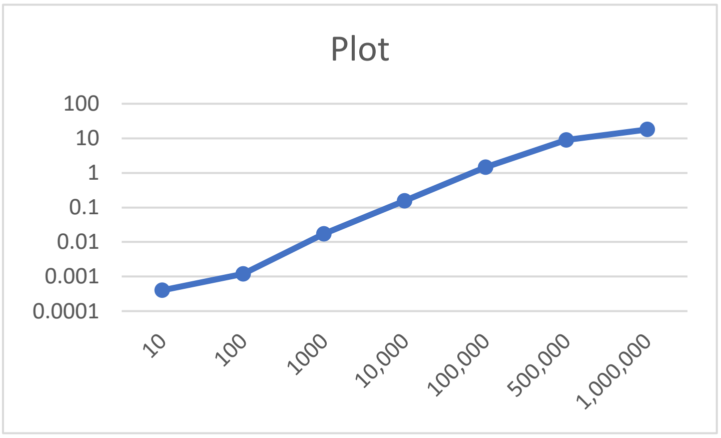
Time Complexity:
The Time Complexity of the Solver Algorithm is O(nlogn). The reason for this is because with each recursion of the divide and conquer, it becomes two sub-tasks. The size of each sub-instance is n/2 and the order of work at each node is constant. Therefore, based on the Master Theorem Solver must be O(nlogn). Sorting the points based on x value will also take O(nlogn) time.

Space Complexity:
The Space Complexity is like the Time Complexity. Due to the divide and conquer algorithm and the two sub-tasks that are created, with each level of recursion the space complexity will decrease. Therefore the Space Complexity will also be O(nlogn).

3. Raw and mean experimental outcomes

Number of Points	**TIM Sort**	**Convex Hull**	**Total Run Time**	**Constant of Proportionality:**
10	0.0005	0.0005	0.001	k = time / nlogn
10	0	0.0005	0.0005	
10	0	0	0	
10	0	0.0005	0.0005	
10	0	0	0	
Average Run Time:			0.0004	0.00004
100	0	0.001	0.001	
100	0	0.001	0.001	
100	0	0.0015	0.0015	
100	0.0005	0.001	0.0015	
100	0	0.001	0.001	
Average Run Time:			0.0012	0.000006
1000	0.0005	0.017	0.0175	
1000	0	0.0165	0.0165	
1000	0	0.017	0.017	
1000	0.00049	0.017	0.01749	
1000	0.0005	0.0165	0.017	
Average Run Time:			0.017098	5.69933E-06
10,000	0.003	0.1585	0.1615	
10,000	0.0025	0.1495	0.152	
10,000	0.0025	0.155	0.1575	
10,000	0.0025	0.1565	0.159	
10,000	0.00249	0.15	0.15249	
Average Run Time:			0.156498	3.91245E-06
100,000	0.033	1.41	1.443	
100,000	0.038	1.432	1.47	

100,000	0.04	1.454	1.494	
100,000	0.034	1.453	1.487	
100,000	0.035	1.424	1.459	
Average Run Time:			1.4706	2.9412E-06
500,000	0.253	8.769	9.022	
500,000	0.237	8.573	8.81	
500,000	0.229	8.903	9.132	
500,000	0.228	8.65	8.878	
500,000	0.247	8.752	8.999	
Average Run Time:			8.9682	3.14731E-06
1,000,000	0.508	17.413	17.921	
1,000,000	0.506	17.638	18.144	
1,000,000	0.534	17.335	17.869	
1,000,000	0.492	17.37	17.862	
1,000,000	0.516	17.443	17.959	
Average Run Time:			17.951	2.99183E-06



The order of growth that best fits this plot is O(n). I think that O(n) fits the plot better than O(nlogn) because there are more cases of O(n) occurring rather than O(logn). This means for example, if 100 cases of O(n) complexity happen for each O(logn) case, that O(logn) will be minor when compared to the O(n) complexity. Because as ‘n’ grows, ‘logn’ does not get that much bigger in comparison. This causes the plot to look more linear rather than O(nlogn) even though the complexity is O(nlogn).

Estimate of Constant of Proportionality: 9.24173E-06

4. Theoretical and empirical analysis

Theoretical:

My analysis is that because my Solver Algorithm is of O(nlogn) complexity, that given a specific ‘n’ value, I should be able to calculate the amount of time that a particular data set will require to be solved. Below I ran similar ‘n’ values through the equation ‘nlogn’ and recorded the values.

n	time
10	10
100	200
1,000	3,000
10,000	40,000
100,000	500,000
500,000	2,849,485
1,000,000	6,000,000

As can be seen, the time values to not represent that of the Empirical Data. I believe that the reason for this is due to the fact that a “conversion factor” so to speak is missing from the equation. It isn’t necessarily that the values are wrong, just that they are in the wrong “units”. Using the calculated Constant of Proportionality, we can turn whatever these “units” are, into seconds. So my theory is when ‘k’ is multiplied to ‘nlogn’ that the values will be much closer.

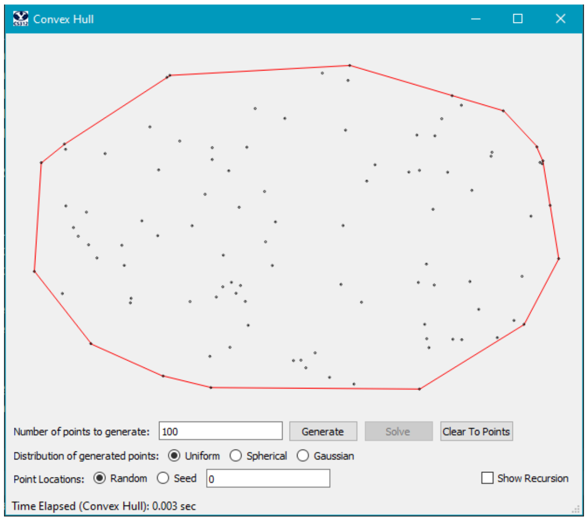
Empirical:

n	time
10	0.0004
100	0.0012
1,000	0.0171
10,000	0.1565
100,000	1.4706
500,000	8.9682
1,000,000	17.951

For example, 3,000 is the Theoretical time value for 1,000 data points. If we multiply 3,000 by the Constant of Proportionality it will give us 0.0277. If we compare this to the value in the Empirical table above for 1,000 data points we see that they are much closer. (0.0277 & 0.0171). If we use this constant, it will match our theoretically calculated values much closer to the actual graph representation of our data. Thus, proving that the Algorithm is indeed O(nlogn).

5. Screenshot Example

100 Points:



Number of points to generate: 100

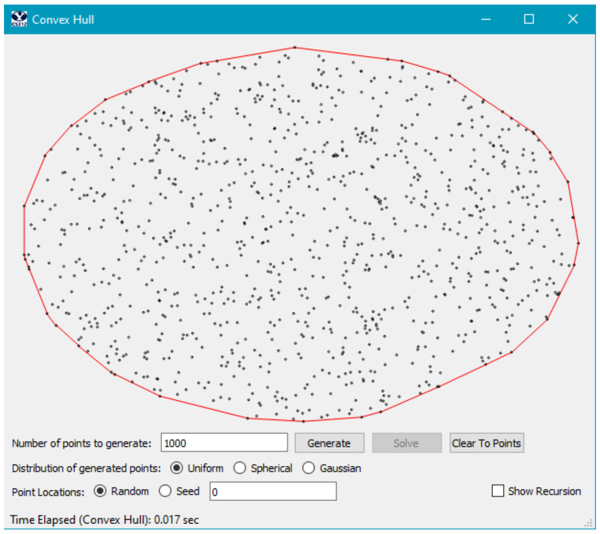
Distribution of generated points: [✓ Uniform] Spherical Gaussian

Point Locations: [✓ Random] Seed 0

Show Recursion

Time Elapsed (Convex Hull): 0.003 sec

1000 Points:



Number of points to generate: 1000

Distribution of generated points: [✓ Uniform] Spherical Gaussian

Point Locations: [✓ Random] Seed 0

Show Recursion

Time Elapsed (Convex Hull): 0.017 sec