Atkinson, Jenna- Convex Hull

1. **Correct functioning code to solve the Convex Hull problem using the divide and conquer scheme discussed above. Include your documented source code.**

hull.py (defining classes)

```
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))

# Space: O(1) (both Point obj are pointers)
class Point:
  # QPointF pt
  # Point next #clockwise
  # Point prev #counterclockwise
  def __init__(self, point:QPointF, next=None, prev=None):
    self.pt = point
    self.next = next
    self.prev = prev
    if next is None:
      self.next = {}
    if prev is None:
      self.prev = {}

  def x(self):
    return self.pt.x()

  def y(self):
    return self.pt.y()

  # Next point
  def setNext(self, next):
    if isinstance(next, Point):
      self.next = next

  def clockwise(self):
    return self.next

  # Prev Point
```

```
  def setPrev(self, prev):
    if isinstance(prev, Point):
      self.prev = prev

  def counterclockwise(self):
    return self.prev

  def __str__(self):
    return f"Point (X: {self.pt.x()} Y: {self.pt.y()})"

# Space: O(1) (both Point obj are pointers)
class Hull:
  # Point leftmostPt
  # Point rightmostPt
  # int hullLen
  def __init__(self, left:Point, right:Point, hullLen:int):
    self.leftmostPt = left
    self.rightmostPt = right
    self.hullLen = hullLen

  def setLeftmost(self, left:Point):
    self.leftmostPt = left

  def setRightmost(self, right:Point):
    self.rightmostPt = right

  def __str__(self):
    return f"Hull (EdgeLen: {self.hullLen})\nLeftmost:
{self.leftmostPt}\nRightmost: {self.rightmostPt}"
```

convex_hull.py

```
from hull import Hull, Point
# 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
```

```
# 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

#
# Class to solve the hull
#
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 clearAllLines(self):
    self.view.clearLines()

  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)

  # 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
  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)

  def generatePolygonFromHull(self, hull:Hull):
    return self.generatePolygon(hull.rightmostPt)

  def generatePolygon(self, root:Point):
    polygon = []
    if root == {} or root.pt == {}:
      return polygon
    point = root.next
    if point != {} and point.pt != {}:
      polygon.append(QLineF(root.pt, point.pt))
    while (point != {} and point.pt != {} and point != root):
      polygon.append(QLineF(point.pt, point.next.pt))
      point = point.next
    return polygon

# This is the method that gets called by the GUI and actually executes the
finding of the hull
  # Time: O(nlogn) Space: O(n**2)
  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 points by increasing x value
    points = sorted(points, key=lambda point: point.x()) # Time: O(nlogn) Space:
O(n)

    # Creates a Hull from each point, and builds a list
    hullList = []
    for i in points: # Time: O(n) Space: O(n)
      point = Point(i)
      hullList.append(Hull(point, point, 1))
    t2 = time.time()

    t3 = time.time()
    # Solves the hulls by combining them
    finalHull = self.hull_solver(hullList) # Time: O(nlogn) Space: O(n**2)
```

```
    t4 = time.time()

    self.clearAllLines()
    #self.showHull(self.generatePolygonFromHull(finalHull),RED)
    self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4-t3))

  # Prints all the Hull values starting from the leftmost pt
  def printHullValues(self, hull:Hull):
    print("Hull Values:")
    print(hull.leftmostPt)
    pt = hull.leftmostPt.next
    while (pt != {} and pt != hull.leftmostPt):
      print(pt)
      pt = pt.next

  # Returns a Hull object combined from a list of hulls
  # Time: O(nlogn) Space: O(n**2)
  def hull_solver(self, hullList):
    if len(hullList) == 1:
      return hullList[0]
    else:
      halfLen = int(len(hullList)/2)
      leftHull = self.hull_solver(hullList[:halfLen])
      rightHull = self.hull_solver(hullList[halfLen:])
      return self.combine_hull(leftHull, rightHull) # Time: O(n) Space: O(1)

  # Combines two hulls together into a single hull
  # Time: O(n) Space: O(1)
  def combine_hull(self, leftHull:Hull, rightHull:Hull):
    # Find the top tangent
    # Time: O(n) Space: O(1)
    topLeftTanPt, topRightTanPt = self.findTopTangent(leftHull, rightHull)

    # Find the bottom tangent
    # Time: O(n) Space: O(1)
    bottomLeftTanPt, bottomRightTanPt = self.findBottomTangent(leftHull,
rightHull)

    # Reset points to exclude inside points from old hull (maintaining clockwise
as 'next' ordering)
    topLeftTanPt.setNext(topRightTanPt)
    topRightTanPt.setPrev(topLeftTanPt)

    bottomRightTanPt.setNext(bottomLeftTanPt)
    bottomLeftTanPt.setPrev(bottomRightTanPt)
```

```
    # Find new left and rightmost of the new hull
    # Time: O(n) Space: O(1)
    leftmost, rightmost, hullLen = self.findExtremePts(topLeftTanPt) # choosing
topLeftTanPt is arbitrary

    # Return the new hull
    newHull = Hull(leftmost, rightmost, hullLen) # Time: O(1) Space: O(1)
    return newHull

  # Returns the leftmost, rightmost and number of points in hull's edge after
going around the linked list
  # Time: O(n) Space: O(1)
  def findExtremePts(self, initialPt:Point):
    hullLen = 1
    leftmost = initialPt
    rightmost = initialPt
    pt = initialPt.next
    while(pt != {} and pt != initialPt): # Time: O(n) Space: O(1)
      hullLen += 1
      if pt.x() < leftmost.x():
        leftmost = pt
      if pt.x() > rightmost.x():
        rightmost = pt
      pt = pt.next
    return leftmost, rightmost, hullLen

  # Returns true if testSlope is more negative
  isMoreNegativeSlope = lambda self, testSlope, ogSlope: testSlope < ogSlope

  # Returns true if testSlope is more negative
  isMorePositiveSlope = lambda self, testSlope, ogSlope: testSlope > ogSlope

  # NOTE: clockwise is always trying to find a more positive sloped tangent,
counterclockwise is always trying to find negative
  # Time: O(n) Space: O(1)
  def findBottomTangent(self, leftHull:Hull, rightHull:Hull):
    return self.findTangent(leftHull, Point.clockwise, self.isMorePositiveSlope,
    rightHull,  Point.counterclockwise, self.isMoreNegativeSlope)

  # Time: O(n) Space: O(1)
  def findTopTangent(self, leftHull:Hull, rightHull:Hull):
    return self.findTangent(leftHull, Point.counterclockwise,
self.isMoreNegativeSlope,
    rightHull, Point.clockwise, self.isMorePositiveSlope)
```

```
  # Returns top or bottom tangent based on the directions given
  # Left/right direction is clockwise/counterclockwise
  # Time: O(n) Space: O(1)
  def findTangent(self, leftHull:Hull, leftDirection, leftCompare,
rightHull:Hull, rightDirection, rightCompare):
    leftTangentPt = leftHull.rightmostPt
    rightTangentPt = rightHull.leftmostPt

    # Test tangent slopes by changing points on left # Time: O(n/2) Space: O(1)
    leftTangentPt = self.findBestPtWithSlope(leftTangentPt, rightTangentPt,
leftCompare, False, leftDirection)

    # Test tangent slopes by changing points on right # Time: O(n/2) Space: O(1)
    rightTangentPt = self.findBestPtWithSlope(rightTangentPt, leftTangentPt,
rightCompare, False, rightDirection)

    oldLeftPt = None
    oldRightPt = None
    # Continue testing right or left tangents until neither change
    while (oldLeftPt != leftTangentPt or oldRightPt != rightTangentPt):  # Time:
O(n/2) Space: O(1) because we will run this loop at most 4 times
      oldLeftPt = leftTangentPt
      oldRightPt = rightTangentPt
      # Test tangent slopes on the left again # Time: O(n/2) Space: O(1)
      leftTangentPt = self.findBestPtWithSlope(leftTangentPt, rightTangentPt,
leftCompare, True, leftDirection)

      # Test tangent slopes on the right again # Time: O(n/2) Space: O(1)
      rightTangentPt = self.findBestPtWithSlope(rightTangentPt, leftTangentPt,
rightCompare, True, rightDirection)

    # Return best points on the left and the right
    return leftTangentPt, rightTangentPt

  # Traverse the linked list in search of a more positive or negative slope
  # Time: O(n/2) Space: O(1)
  def findBestPtWithSlope(self, initialPt:Point, otherHullPt:Point, compare,
stopTesting:bool, getNext):
    pt = getNext(initialPt) # Time: O(1)
    tangentSlope = self.slope(initialPt, otherHullPt)
    bestPt = initialPt
    # Test each point until we get back to the beginning
    while (pt != {} and pt != initialPt): # Time: O(n/2) Space: O(1)
      testSlope = self.slope(pt, otherHullPt) # Time: O(1) Space: O(1)
```

```
      # Try to find a more negative/positive slope
      if compare(testSlope, tangentSlope):  # Time: O(1) Space: O(1)
        tangentSlope = testSlope
        bestPt = pt
      # If we don't need to keep testing other slopes, then stop
      elif stopTesting:
        break
      # Go clockwise/counterclockwise to test again
      pt = getNext(pt) # Time: O(1)
    return bestPt

  # Returns slope of two points
  # Time: O(1) Space: O(1)
  def slope(self, pt1:Point, pt2:Point):
    return (pt2.y() - pt1.y()) / (pt2.x() - pt1.x())
```

2. **Explain the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code. Also, include your theoretical analysis for the entire algorithm including discussion of the recurrence relation.**

(Note: I detail each function starting from the bottom of my file, since top functions rely on the helpers below)

Both Hull and Point objects need space in memory, however most objects simply have references to each other, and the only information being recorded is the QPointF object (which there are n), and an int, which is negligible. Therefore, I am counting the “extra” space needed for the objects as O(1), and I note the O(n) space for the QPointF objects elsewhere.

```
  # Returns slope of two points
  # Time: O(1) Space: O(1)
  def slope(self, pt1:Point, pt2:Point):
```

slope simply calculates the slope using subtraction and division. Although division is not constant, in the scheme of n for this algorithm, it is negligible, so I’m going to ignore it.

```
  # Traverse the linked list in search of a more positive or
  negative slope
  # Time: O(n/2) Space: O(1)
  def findBestPtWithSlope(self, initialPt:Point, otherHullPt:Point,
  compare, stopTesting:bool, getNext):
```

findBestPtWithSlope loops through all the points of one of the hulls we are trying to merge.

```
    # Test each point until we get back to the beginning
    while (pt != {} and pt != initialPt): # Time: O(n/2) Space: O(1)
```

It will go through each point, and if we determine the points of the left + right hulls we are merging as n, then this loop will look at half of those points, hence **O(n/2) time.**

The **space is constant**, since we are storing some temp values and returning.

```
# Returns top or bottom tangent based on the directions given
# Left/right direction is clockwise/counterclockwise
# Time: O(n) Space: O(1)
def findTangent(self, leftHull:Hull, leftDirection, leftCompare, rightHull:Hull, rightDirection, rightCompare):
```

findTangent calls findBestPtWithSlope two initial times, and then up to 4 times in a while loop to verify that we did find the correct tangents.

```
while (oldLeftPt != leftTangentPt or oldRightPt != rightTangentPt):  # Time: O(n/2) Space: O(1) because we will run this loop at most 4 times
```

That means the **time complexity is O(n),** and the **space complexity is constant** with local variables.

```
# Returns the leftmost, rightmost and number of points in hull's edge after going around the linked list
# Time: O(n) Space: O(1)
def findExtremePts(self, initialPt:Point):
```

findExtremePts loops through the entire combined hull to find the new leftmost and rightmost, which means we visit each node for a **time complexity of O(n). The space is constant.**

```
# Combines two hulls together into a single hull
# Time: O(n) Space: O(1)
def combine_hull(self, leftHull:Hull, rightHull:Hull):
```

combine_hull calls findTangent twice, and then findExtrememPts once, for a **time complexity of O(n) and a constant space.**

```
# Returns a Hull object combined from a list of hulls
# Time: O(nlogn) Space: O(n**2)
def hull_solver(self, hullList):
```

hull_solver is my function which recurses by splitting a list of Hull objects in half until each section is 1, and then combining them on the way up.

```
halfLen = int(len(hullList)/2)
leftHull = self.hull_solver(hullList[:halfLen])
rightHull = self.hull_solver(hullList[halfLen:])

return self.combine_hull(leftHull, rightHull) # Time: O(n) Space: O(1)
```

The recurrence relationship depends on how many subsections we divide, how big those sections are and the work to put them back together. Since I split the array into two subsections, my ‘a’ is 2. Each subsection is n/2, so ‘b’ is 2. Assuming that the combine_hull function which is the work to put them together before returning is O(n), then d is 1.

Theorem (Master Theorem Restated)
If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0, b > 1, d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } 1 > \frac{a}{b^d} \\ O(n^d \log n) & \text{if } 1 = \frac{a}{b^d} \\ O(n^{\log_b a}) & \text{if } 1 < \frac{a}{b^d} \end{cases}$$

According to the Master Theorem, $\frac{a}{b^d} = 1$, which means the Big-O for hull_solver is **O(nlogn).**

Space complexity is O(n2)** because we have O(n) QPoints of memory and then our recursive tree is O(n) deep.

```
# This is the method that gets called by the GUI and actually executes the finding of the hull
  # Time: O(nlogn) Space: O(n**2)
  def compute_hull(self, points, pause, view):
```

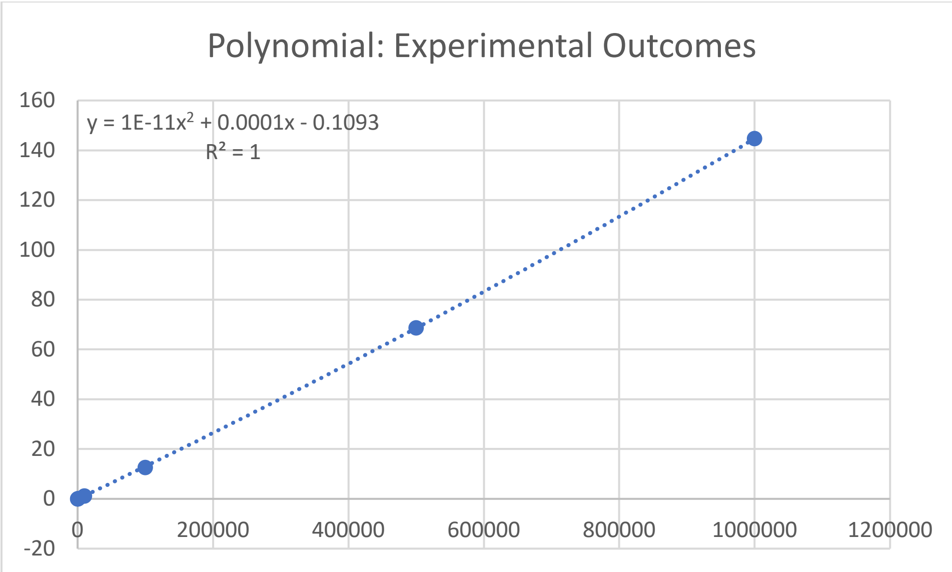
compute_hull is in charge of first sorting the points, which with Python's Timesort is O(nlogn), and then calling hull_solver which is O(nlogn) time, for a total **time complexity of O(nlogn).** We get the **space complexity from hull_solver, for O(n**2).**

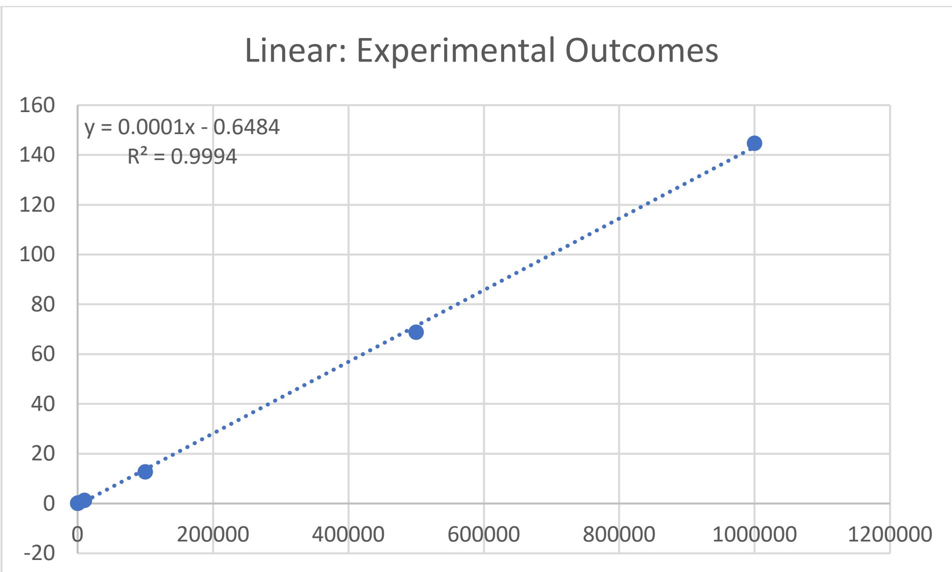
3. **Include your raw and mean experimental outcomes, plot, and your discussion of the pattern in your plot. Which order of growth fits best? Give an estimate of the constant of proportionality. Include all work and explain your assumptions.**

N points	Time1	Time2	Time3	Time4	Time5	Mean
10	0.003	0.004	0.003	0.002	0.002	**0.0028**
100	0.021	0.032	0.026	0.028	0.032	**0.0278**
1000	0.117	0.119	0.131	0.126	0.131	**0.1248**
10000	1.095	1.123	1.227	1.127	1.153	**1.145**
100000	13.201	11.727	11.766	11.846	14.596	**12.6272**
500000	60.896	69.063	68.775	73.661	71.231	**68.7252**
1000000	145.628	147.728	140.587	-	-	**144.647**

The r**2 values for the experimental outcomes were very close between the polynomial and linear, which is due to there being such a small constant of proportionality for the polynomial graph. *(see the next page for charts, r**2 values and line equations)*

It seems that the best order of growth for Big-O would be O(n).





4. Discuss and explain your observations with your theoretical and empirical analyses, including any differences seen.

My theoretical analysis said the Big-O was O(nlogn), however this trendline seems to indicate that it is closer to O(n). I believe this difference is because many of the parts of the algorithm assume worst case, aka somehow magically most points are on the edge and there isn't anyway to remove most of them. This is an unlikely distribution with our random generator, so it appears to be closer to linear time in reality, instead of our O(nlogn) worse case.

5. **Include a correct screenshot of an example with 100 points and a screenshot of an example with 1000 points.**