Benjamin Duncan
CS 312
Feb 9, 2023

Convex Hull Report

Part 1 - Code

See appendix

Part 2 - Complexity

The majority of my implementation is contained within getConvexHull():

```
   # The main function to get the convex hull - it will run log(n)
times
   # In this function there are two while loops, each with O(n) time
complexity
   # Therefore, we get O(n log n) time for this algorithm. The tiem
complexities of
   # each function can be found above the corresponding funciton.
   def getConvexHull(self, points) -> list:
       if len(points) <= 3:
           if len(points) == 3 and self.getSlope(points[0],
points[1]) < self.getSlope(points[0], points[2]):
               (points[1], points[2]) = (points[2], points[1])
       else:
           # get hulls
           midIndex = math.floor(len(points) / 2)
           hull1 = self.getConvexHull(points[0: midIndex])
           hull2 = self.getConvexHull(points[midIndex:])

           # set up variables to get top and bottom lines
           indexTop1 = self.findLeftMostPointIndex(hull1)
           indexTop2 = 0

           indexBottom1 = indexTop1
           indexBottom2 = 0

           nextIndex1 = (indexTop1 + 1) if indexTop1 + 1 !=
len(hull1) else 0
           nextIndex2 = len(hull2) - 1

           slope = self.getSlope(hull1[indexTop1], hull2[indexTop2])
           slopeNew1 = self.getSlope(hull1[nextIndex1],
hull2[indexTop2])
```

```
            slopeNew2 = self.getSlope(hull1[indexTop1],
hull2[nextIndex2])

            # find top two points - linear time
            while slope < slopeNew1 or slope > slopeNew2:
                if slope < slopeNew1:
                    indexTop1 = nextIndex1
                    nextIndex1 = self.getNextPointIndex(hull1,
nextIndex1, True)
                if slope > slopeNew2:
                    indexTop2 = nextIndex2
                    nextIndex2 = self.getNextPointIndex(hull2,
nextIndex2, False)

                slope = self.getSlope(hull1[indexTop1],
hull2[indexTop2])
                slopeNew1 = self.getSlope(hull1[nextIndex1],
hull2[indexTop2])
                slopeNew2 = self.getSlope(hull1[indexTop1],
hull2[nextIndex2])

            nextIndex1 = indexBottom1 - 1 if indexBottom1 != 0 else
len(hull1) - 1
            nextIndex2 = 1

            slope = self.getSlope(hull1[indexBottom1],
hull2[indexBottom2])
            slopeNew1 = self.getSlope(hull1[nextIndex1],
hull2[indexBottom2])
            slopeNew2 = self.getSlope(hull1[indexBottom1],
hull2[nextIndex2])

            # find bottom two points - linear time
            while slope > slopeNew1 or slope < slopeNew2:
                if slope > slopeNew1:
                    indexBottom1 = nextIndex1
                    nextIndex1 = self.getNextPointIndex(hull1,
nextIndex1, False)
                if slope < slopeNew2:
                    indexBottom2 = nextIndex2
                    nextIndex2 = self.getNextPointIndex(hull2,
nextIndex2, True)
                slope = self.getSlope(hull1[indexBottom1],
hull2[indexBottom2])
```

```
                slopeNew1 = self.getSlope(hull1[nextIndex1],
hull2[indexBottom2])
                slopeNew2 = self.getSlope(hull1[indexBottom1],
hull2[nextIndex2])

            # concatenate the two arrays - linear time
            points = hull1[0:indexBottom1 + 1] +
((hull2[indexBottom2:] + hull2[0:1]) if indexTop2 == 0 else
(hull2[indexBottom2:indexTop2 + 1])) + ([] if indexTop1 == 0 else
hull1[indexTop1:])

        return points
```

This function here is O(n log n) time. As you can see in the comments above the function. This is a recursive call, thus giving us the log (n) part of our time complexity. In other words, it runs log(n) times as it divides the points up into sub-hulls. The rest of the functions that I've used are all constant time. You can reference the appendix to see my comments for more information. Within this function, we have 2 while loops which run in linear time. This gives us the theoretical O(n log n) time complexity that we desired.

The space complexity of this algorithm is pretty high. There was no requirement for space complexity so I didn't worry about that very much. However, for each iteration of the getConvexHull algorithm, we have a points array, two arrays representing the sub-hulls, and various integers and floats to represent the data. This gives us a space complexity of about O(n log n) as well. Although that does seem pretty low, I can think of some ways that it could be better, mainly replacing the points array with the sub-hulls as soon as they're back in order to minimize space.

Part 3 - Empirical Data

Table 1 shows the empirical data received from multiple runs of my convex hull implementation. I ran 5 iterations of varying points, *n*, where *n = 10, 100, 1,000, 10,000, 100,000, 500,000, 1,000,000*. The mean is below the five iterations followed by the calculated constant of proportionality, *c*. The order of growth in big-O notation is O(n log n) (I work through this in part 3). Using that order of growth, we can calculate a constant of proportionality:

c = mean / (n log n)

In higher numbers, the constant begins to settle at around 1.8e^-6 whereas the number is higher at lower numbers. Therefore, the constant equates to 1.8e^-6 for our big-theta functions.

However, taking a look at this now and trying to find a time complexity from this data, I would claim that it is O(n) given the fact that it grows pretty linearly with a constant of proportionality of *c = 1 / 100,000*. For each *n*, the mean of the iterations is very close to *n/100,000*. For instance, 10.9284 is fairly close to *n / 100,000* where *n = 1,000,000*. I discuss the differences between the theoretical and empirical order of growth in part 4.

i	10	100	1,000	10,000	100,000	500,000	1,000,000
1	0.000	0.001	0.011	0.084	0.923	5.438	10.356
2	0.000	0.001	0.011	0.093	0.861	4.813	10.421
3	0.000	0.001	0.011	0.084	0.902	5.838	11.928
4	0.000	0.001	0.011	0.084	0.897	5.477	10.819
5	0.000	0.001	0.011	0.086	0.871	5.296	11.118
Mean	0.000	0.001	0.011	0.0862	0.8708	5.3724	10.9284
c	*?*	5	3.66e^-6	2.155e^-6	1.741e^-6	1.885e^-6	1.821e^-6

Table 1

Chart 1 shows the values from table 1 in visual form. As you can see, the growth is generally *n log n* as n increases. The individual colors represent the 5 iterations in order.

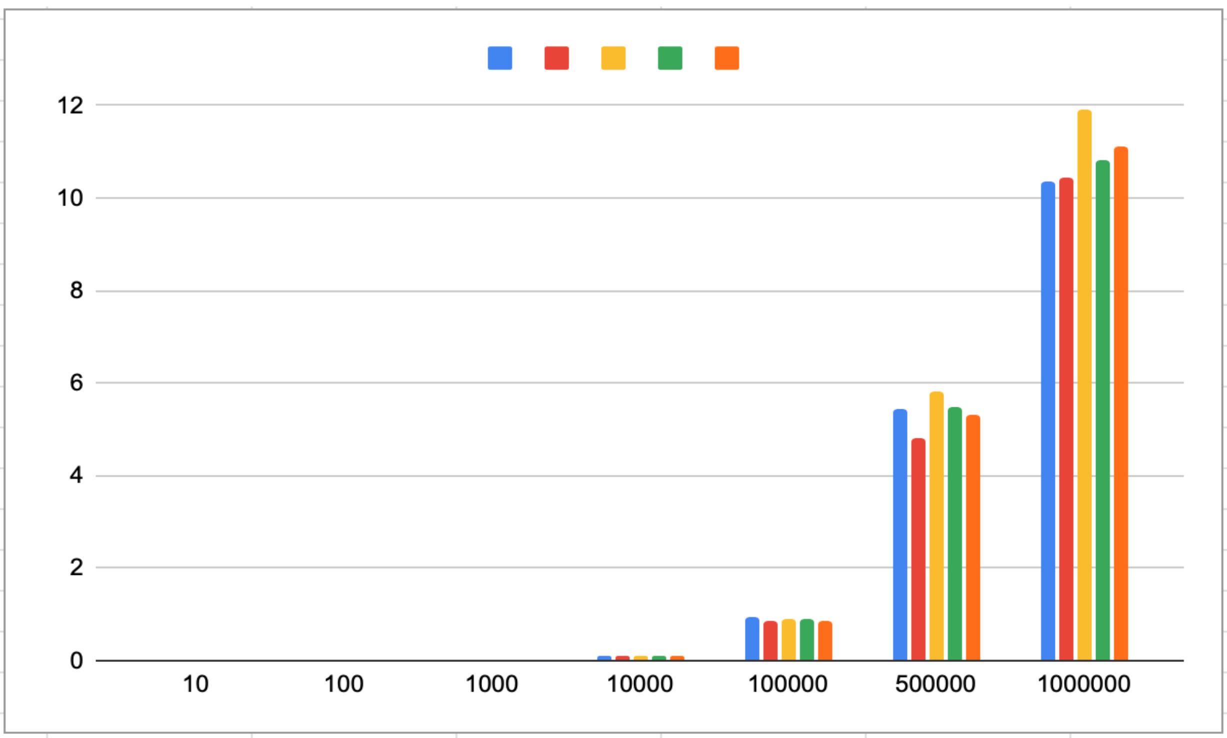

Chart 1

Part 4

The disparity between the theoretical and empirical time complexities confuses me. Even in going through my code, I came to the conclusion that the algorithm should run O(n log n). However, the data never lies. It seems to run at about O(n) time with a constant of proportionality of 1/100,000. I went through the math in section 3. I honestly don't have any ideas as to why this would happen, excepting hardware differences. I'm perfectly happy accepting a lower time complexity.

Part 5

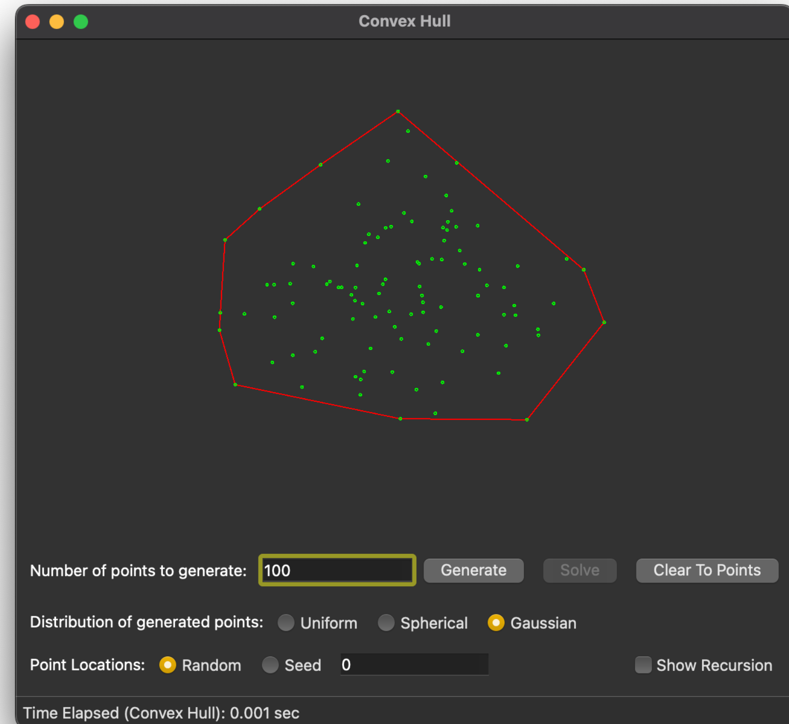

Fig 1: Example of a successful convex hull when n = 100

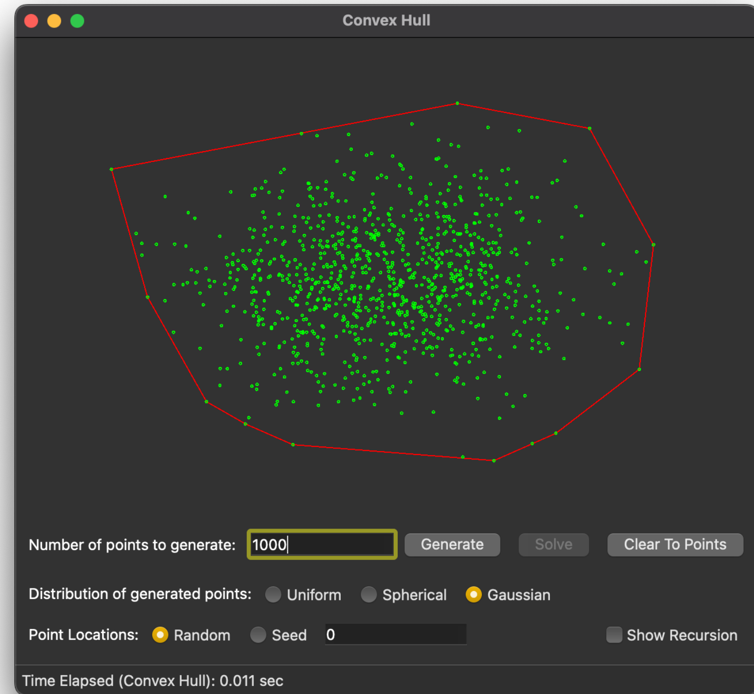

Fig 2: Example of a successful convex hull when n = 1,000

Appendix - Code

convex_hull.py

```
from typing import List

from which_pyqt import PYQT_VER
import math

if PYQT_VER == 'PYQT5':
   from PyQt5.QtCore import QLineF, QPointF, QObject
elif PYQT_VER == 'PYQT4':
   from PyQt4.QtCore import QLineF, QPointF, QObject
elif PYQT_VER == 'PYQT6':
   from PyQt6.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

#
# 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)

    # MY FUNCTIONS
    # Used to help the quick sort algorithm below, sortPointsByX()
    #
    def partition(self, array, low, high):
        pivot = array[high]
        i = low - 1
        for j in range(low, high):
            if array[j].x() > pivot.x():
                i = i + 1
                (array[i], array[j]) = (array[j], array[i])

        (array[i + 1], array[high]) = (array[high], array[i + 1])
        return i + 1

    # A quick sort algorithm to sort the inital unsorted points by x
value.
    # This is a O(n log n) tiem complexity (given from wikipedia).
    def sortPointsByX(self, array, low, high):
        if low < high:
            pi = self.partition(array, low, high)
            self.sortPointsByX(array, low, pi - 1)
            self.sortPointsByX(array, pi + 1, high)

    # This is used to get the slope - constant time
    def getSlope(self, p1, p2) -> int:
        return (p1.y() - p2.y())/(p1.x() - p2.x())

    # Get the left-most point - n time
    def findLeftMostPointIndex(self, points) -> int:
        leftMostIndex = 0
        while points[leftMostIndex + 1].x() <
points[leftMostIndex].x():
            leftMostIndex += 1
            if leftMostIndex + 1 >= len(points):
                break
        return leftMostIndex

    # Get the next point - constant time
    def getNextPointIndex(self, points, index, clockwise: bool):
        if clockwise:
            if len(points) == index + 1:
                return 0
            else:
```

```
                return index + 1
        else:
            if index == 0:
                return len(points) - 1
            else:
                return index - 1

    # The main function to get the convex hull - it will run log(n)
times
    # In this function there are two while loops, each with O(n) time
complexity
    # Therefore, we get O(n log n) time for this algorithm. The tiem
complexities of
    # each function can be found above the corresponding funciton.
    def getConvexHull(self, points) -> list:
        if len(points) <= 3:
            if len(points) == 3 and self.getSlope(points[0],
points[1]) < self.getSlope(points[0], points[2]):
                (points[1], points[2]) = (points[2], points[1])
        else:
            # get hulls
            midIndex = math.floor(len(points) / 2)
            hull1 = self.getConvexHull(points[0: midIndex])
            hull2 = self.getConvexHull(points[midIndex:])

            # set up variables to get top and bottom lines
            indexTop1 = self.findLeftMostPointIndex(hull1)
            indexTop2 = 0

            indexBottom1 = indexTop1
            indexBottom2 = 0

            nextIndex1 = (indexTop1 + 1) if indexTop1 + 1 !=
len(hull1) else 0
            nextIndex2 = len(hull2) - 1

            slope = self.getSlope(hull1[indexTop1], hull2[indexTop2])
            slopeNew1 = self.getSlope(hull1[nextIndex1],
hull2[indexTop2])
            slopeNew2 = self.getSlope(hull1[indexTop1],
hull2[nextIndex2])

            # find top two points - linear time
            while slope < slopeNew1 or slope > slopeNew2:
                if slope < slopeNew1:
```

```
                    indexTop1 = nextIndex1
                    nextIndex1 = self.getNextPointIndex(hull1,
nextIndex1, True)
                if slope > slopeNew2:
                    indexTop2 = nextIndex2
                    nextIndex2 = self.getNextPointIndex(hull2,
nextIndex2, False)

                slope = self.getSlope(hull1[indexTop1],
hull2[indexTop2])
                slopeNew1 = self.getSlope(hull1[nextIndex1],
hull2[indexTop2])
                slopeNew2 = self.getSlope(hull1[indexTop1],
hull2[nextIndex2])

            nextIndex1 = indexBottom1 - 1 if indexBottom1 != 0 else
len(hull1) - 1
            nextIndex2 = 1

            slope = self.getSlope(hull1[indexBottom1],
hull2[indexBottom2])
            slopeNew1 = self.getSlope(hull1[nextIndex1],
hull2[indexBottom2])
            slopeNew2 = self.getSlope(hull1[indexBottom1],
hull2[nextIndex2])

            # find bottom two points - linear time
            while slope > slopeNew1 or slope < slopeNew2:
                if slope > slopeNew1:
                    indexBottom1 = nextIndex1
                    nextIndex1 = self.getNextPointIndex(hull1,
nextIndex1, False)
                if slope < slopeNew2:
                    indexBottom2 = nextIndex2
                    nextIndex2 = self.getNextPointIndex(hull2,
nextIndex2, True)
                slope = self.getSlope(hull1[indexBottom1],
hull2[indexBottom2])
                slopeNew1 = self.getSlope(hull1[nextIndex1],
hull2[indexBottom2])
                slopeNew2 = self.getSlope(hull1[indexBottom1],
hull2[nextIndex2])

            # concatenate the two arrays - linear time
```

```
           points = hull1[0:indexBottom1 + 1] +
((hull2[indexBottom2:] + hull2[0:1]) if indexTop2 == 0 else
(hull2[indexBottom2:indexTop2 + 1])) + ([] if indexTop1 == 0 else
hull1[indexTop1:])

       return points

   # 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()
       self.sortPointsByX(points, 0, len(points) - 1)
       t2 = time.time()

       t3 = time.time()
       points = self.getConvexHull(points)
       polygon = [QLineF(points[i], points[(i + 1) % len(points)])
for i in range(len(points))]
       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))
```