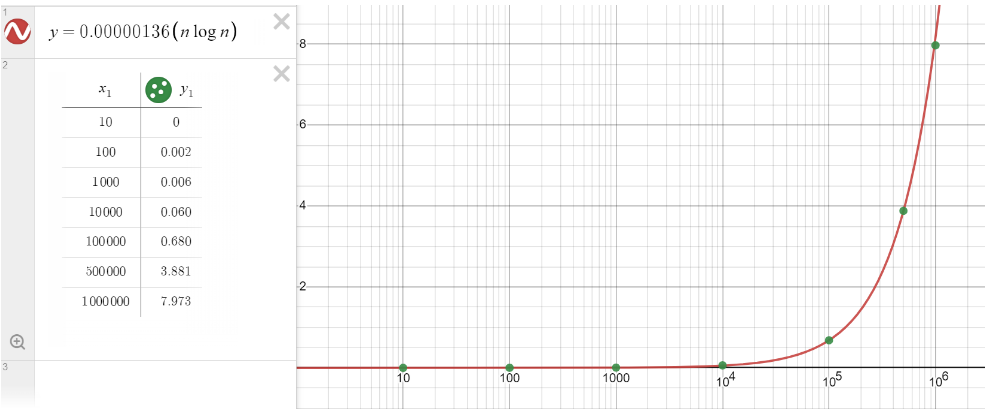
Gideon Hale 2024-02-05

Project 2: Convex Hull

1. Code: see Appendix
2. Time and Space Complexity
    i. Sort — line 71
        a. Python's built-in sorting algorithm sorts in place using Timsort. This yields a space complexity of O(n) and a time complexity of O(nlogn) in the worst case.
    ii. findHull() — line 89
        a. Each leaf node takes constant time, since they don't depend on n. (line 93–95)
        b. The data is split into two section, so a = 2 (line 98,99)
        c. The task is half as large each time, so b = 2 (line 98,99)
        d. Finding each tangent line takes O(n), as is explained in line 104
        e. Each list concatenation in lines 115–120) takes O(n) time
        f. Therefore, d = 1, since each step is O(n) time in total.
        g. By the Master Theorem, a / (b^d) = 1, so the complexity is O((n^d)logn) = O(nlogn), which is what we want.
        h. Since this is a depth-first approach, we use a stack to accomplish everything, so the space complexity is O(n).
3. Empirical Analysis
    i.

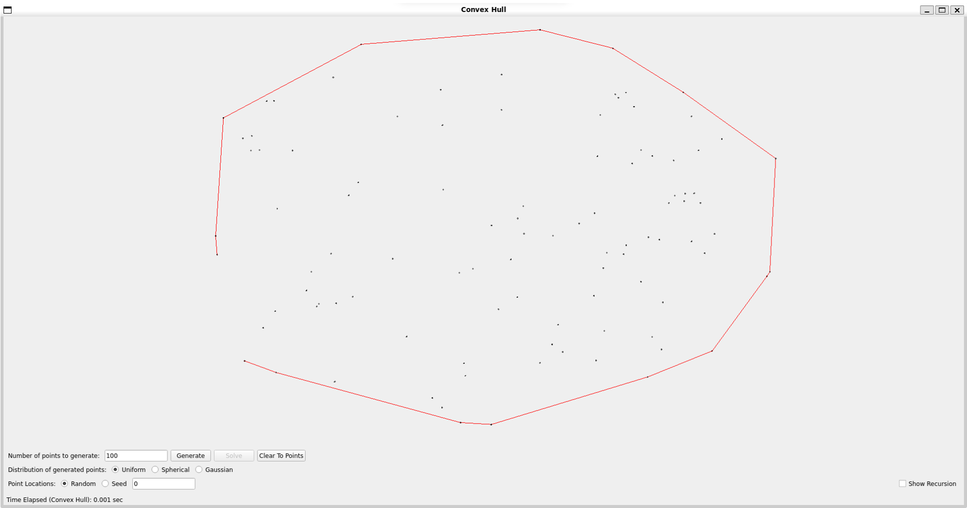
A	B	C	D	E	F	G	H
Data size	10	100	1000	10000	100000	500000	1000000
Test 1	0.000	0.001	0.006	0.068	0.675	3.924	7.910
Test 2	0.000	0.003	0.007	0.052	0.656	3.931	8.169
Test 3	0.000	0.003	0.006	0.058	0.788	3.778	8.213
Test 4	0.000	0.002	0.006	0.054	0.647	3.947	7.794
Test 5	0.000	0.002	0.006	0.067	0.632	3.825	7.781
Mean Wall Time	0.000	0.002	0.006	0.060	0.680	3.881	7.973

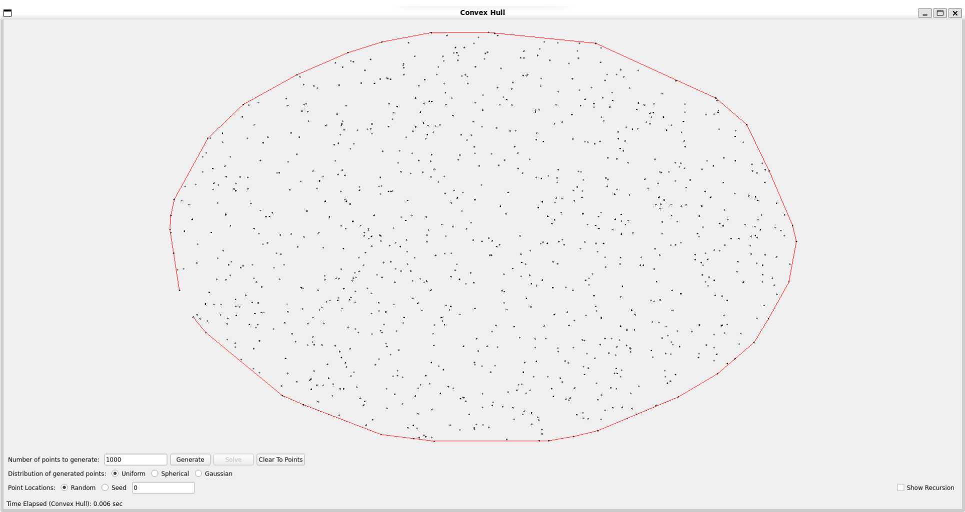
ii. 

iii. The line y = 0.00000136nlogn fits pretty well. I found this simply by trial and error.

4. Part 3. indicates that my algorithm truly runs in O(nlogn) time, with a constant factor of around k = 0.00000136, based on my experimental data. This supports my theoretical analysis in Part 2.

5. Examples

i. 

ii. 

Appendix

```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
elif PYQT_VER == 'PYQT6':
    from PyQt6.QtCore import QLineF, QPointF, QObject
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))



import time
import copy

# Some global color constants that might be useful
RED = (255,0,0)
ORANGE = (255,165,0)
YELLOW = (255,255,0)
GREEN = (0,255,0)
BLUE = (0,0,255)
PURPLE = (128,0,128)

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

class ConvexHullSolver(QObject):

    count = 0

# 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)
```

```python
    # 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()

        points.sort(key=lambda point: point.x()) # uses Timsort, which worst case O(nlogn) time

        t2 = time.time()

        t3 = time.time()

        hullPoints = self.findHull(points)[0]
        polygon = self.pointsToLines(hullPoints)

        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))

    # input a list of QPointF objects describing all points in the region
    # output a list of QLineF objects describing only the convex hull, as well as the rightmost point
    def findHull(self, points):

        # the size of task at the leaves takes constant time in either case, since sortClock() only accepts
        # a list of size 3 and therefore doesn't depend on n
        if len(points) == 3: return self.sortClockwise(points)
        if len(points) == 2: return points, 1
        if len(points) <  2: raise ValueError("The data set got broken up smaller than 2. This should not have
happened.")

        # splits into two subtasks of half the size of the previous task, so a = b = 2
        lPoints = points[: ((len(points) - 1) // 2) + 1]
        rPoints = points[((len(points) - 1) // 2) + 1 :]

        lPoints, rightmostLPointIndex = self.findHull(lPoints)
        rPoints, rightmostRPointIndex = self.findHull(rPoints)

        # finding the upper and lower tangent lines each take O(n) time, possibly needing
        # to cycle through every node to rech the top or bottom, respectively
        UL, UR = self.upperTangent(lPoints, rPoints, rightmostLPointIndex)
        LL, LR = self.lowerTangent(lPoints, rPoints, rightmostLPointIndex)

        rightmostRPointIndex = (UL - UR + rightmostRPointIndex + 1)

        # this accounts for the case that either of the bottom two connection points are the
        # first element of either subhull;
        # each list concatenation operation takes O(n), since Python implements lists as dynamic arrays;
        # each merge action therefore is O(n), so d = 1
        if LR == 0:
            if LL == 0: return lPoints[:UL + 1] + rPoints[UR:] + [rPoints[LR]], rightmostRPointIndex
            else: return lPoints[:UL + 1] + rPoints[UR:] + [rPoints[LR]] + lPoints[LL:], rightmostRPointIndex
        else:
            if LL == 0: return lPoints[:UL + 1] + rPoints[UR:LR + 1], rightmostRPointIndex
            else: return lPoints[:UL + 1] + rPoints[UR:LR + 1] + lPoints[LL:], rightmostRPointIndex

    # sorts a list of size 3 in a clockwise direction and keeps track of the rightmost index
```

```python
    def sortClockwise(self, points):

        # constant time calculations
        m1 = self.findSlope(points[0], points[1])
        m2 = self.findSlope(points[0], points[2])

        if m1 < m2:
            tmp = points[1]
            points[1] = points[2]
            points[2] = tmp
            return points, 1

        return points, 2

    # finds the upper tangent line between two subhulls, returns the indices of the two endpoints
    def upperTangent(self, lPoints, rPoints, rightmostLPointIndex):

        lIndex = rightmostLPointIndex
        lPoint = lPoints[lIndex] # set to be the rightmost point in lPoints

        rIndex = 0
        rPoint = rPoints[rIndex] # set to be the leftmost point in rPoints

        markedSlope = self.findSlope(lPoint, rPoint)
        isFound = False
        testsPassedStreak = 0 # initialize streak to 0
        numReqTests = 3 # number of sequential passed tests required to insure tangent line is done moving
        while not isFound:

            # decrement around the left points until the slope doesn't decrease
            testIndex = lIndex
            while True:

                # check next point over
                if testIndex > 0: testIndex = testIndex - 1
                else: testIndex = len(lPoints) - 1 # instead of decrementing, loop over to the end of the array
                testPoint = lPoints[testIndex]
                testSlope = self.findSlope(testPoint, rPoint)

                # test if next point will decrease the slope
                if testSlope < markedSlope:
                    lIndex = testIndex
                    lPoint = testPoint
                    markedSlope = self.findSlope(lPoint, rPoint)
                    testsPassedStreak = 0 # reset streak
                else:
                    testsPassedStreak = testsPassedStreak + 1 # passed test 'cause slope did not change
                    break

            # increment around the right points until the slope doesn't increase
            testIndex = rIndex
            while True:

                # check next point over
                if testIndex < len(rPoints) - 1: testIndex = testIndex + 1
                else: testIndex = 0 # instead of incrementing, loop over to the beginning of the array
                testPoint = rPoints[testIndex]
                testSlope = self.findSlope(lPoint, testPoint)

                # test if next point will increase the slope
                if testSlope > markedSlope:
                    rIndex = testIndex
                    rPoint = testPoint
```

```python
                markedSlope = self.findSlope(lPoint, rPoint)
                testsPassedStreak = 0 # reset streak
            else:
                testsPassedStreak = testsPassedStreak + 1 # passed test 'cause slope did not change
                break

        if testsPassedStreak >= numReqTests: isFound = True

    return lIndex, rIndex

# finds the lower tangent line between two subhulls, returns the indices of the two endpoints
def lowerTangent(self, lPoints, rPoints, rightmostLPointIndex):

    lIndex = rightmostLPointIndex
    lPoint = lPoints[lIndex] # set to be the rightmost point in lPoints

    rIndex = 0
    rPoint = rPoints[rIndex] # set to be the leftmost point in rPoints

    markedSlope = self.findSlope(lPoint, rPoint)
    isFound = False
    testsPassedStreak = 0 # initialize streak to 0
    numReqTests = 3 # number of sequential passed tests required to insure tangent line is done moving
    while not isFound:

        # increment around the left points until the slope doesn't increase
        testIndex = lIndex
        while True:

            # check next point over
            if testIndex < len(lPoints) - 1: testIndex = testIndex + 1
            else: testIndex = 0 # instead of incrementing, loop over to the beginning of the array
            testPoint = lPoints[testIndex]
            testSlope = self.findSlope(testPoint, rPoint)

            # test if next point will inrease the slope
            if testSlope > markedSlope:
                lIndex = testIndex
                lPoint = testPoint
                markedSlope = self.findSlope(lPoint, rPoint)
                testsPassedStreak = 0 # reset streak
            else:
                testsPassedStreak = testsPassedStreak + 1 # passed test 'cause slope did not change
                break

        # decrement around the right points until the slope doesn't decrease
        testIndex = rIndex
        while True:

            # check next point over
            if testIndex > 0: testIndex = testIndex - 1
            else: testIndex = len(rPoints) - 1 # instead of decrementing, loop over to the end of the array
            testPoint = rPoints[testIndex]
            testSlope = self.findSlope(lPoint, testPoint)

            # test if next point will decrease the slope
            if testSlope < markedSlope:
                rIndex = testIndex
                rPoint = testPoint
                markedSlope = self.findSlope(lPoint, rPoint)
                testsPassedStreak = 0 # reset streak
            else:
                testsPassedStreak = testsPassedStreak + 1 # passed test 'cause slope did not change
```

```
                break

        if testsPassedStreak >= numReqTests: isFound = True

    return lIndex, rIndex

# calculates the slope between two points
def findSlope(self, p1, p2):
    return (p2.y() - p1.y()) / (p2.x() - p1.x())

# sends the set of points and the message to the GUI
def showPoints(self, points, color, message):
    self.showText(message)
    print(message)
    self.showHull(self.pointsToLines(points), color)
    self.eraseHull(self.pointsToLines(points))

# converts a set of QPointF points to a set of QLineF lines
def pointsToLines(self, points):
    hull = [QLineF(points[i], points[i + 1]) for i in range(len(points) - 2)]
    pointFinal = points[-1]
    pointInitial = points[0]
    lineFinal = QLineF(pointFinal, pointInitial)
    hull.append(lineFinal)
    return hull
```