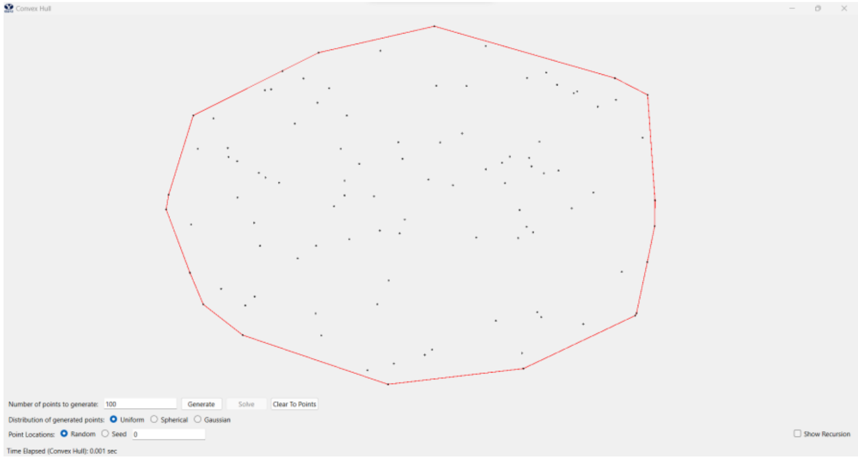
CS 312 Project 2 – Convex Hull

Braden Webb

1. I implemented the algorithm in O(nlogn) time, and have attached all of my code to this document. I thought it may be easier to read those lines of code in a landscape orientation. I've provided my analysis of the time- and space-complexity in the comments of that code below. As demonstration of the working algorithm, here are some screenshots of the GUI running. I noticed that by default, the times provided by the GUI don't include the process of sorting the points; however, I've included that portion of the time in my empirical report.

   Here, I have screenshots of the GUI with 100 and 1000 points.



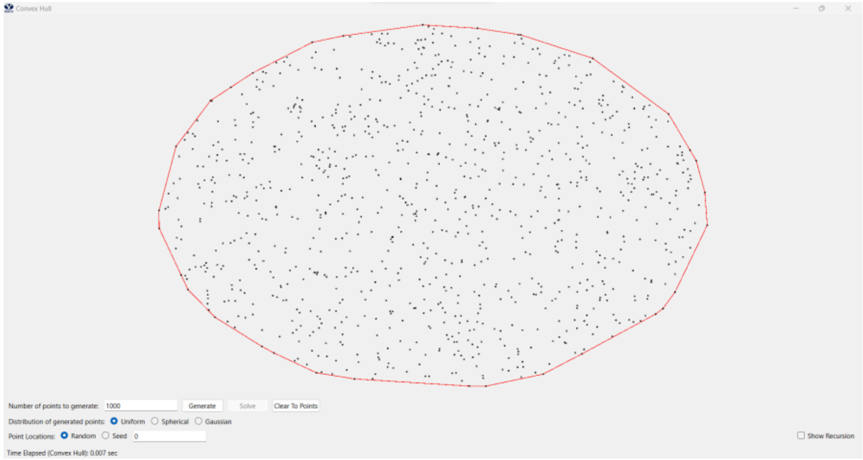
Number of points to generate: 100

Distribution of generated points: ● Uniform ○ Spherical ○ Gaussian

Point Locations: ● Random ○ Seed 0

☐ Show Recursion

Time Elapsed (Convex Hull): 0.001 sec



Number of points to generate: 1000

Distribution of generated points: ● Uniform ○ Spherical ○ Gaussian

Point Locations: ● Random ○ Seed 0

☐ Show Recursion

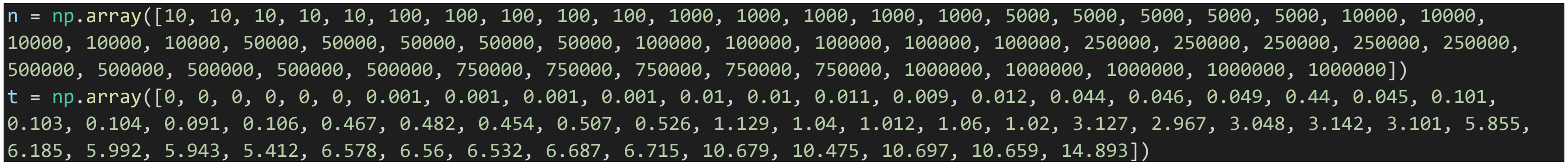
Time Elapsed (Convex Hull): 0.007 sec

2. For each value of n ∈ {10, 100, 1000, 5000, 10000, 100000, 25000, 500000, 750000, 1000000}, I ran my algorithm 5 times and got the following average times:

-	n = 10:	0s
-	n = 100:	0.0008s
-	n = 1000:	0.0104s
-	n = 5000:	0.0456s
-	n = 10000:	0.1011s
-	n = 50000:	0.4871s
-	n = 100000:	1.0522s

- n = 250000:	3.0769s
- n = 500000:	5.8774s
- n = 750000:	6.6146s
- n = 1000000:	11.479s

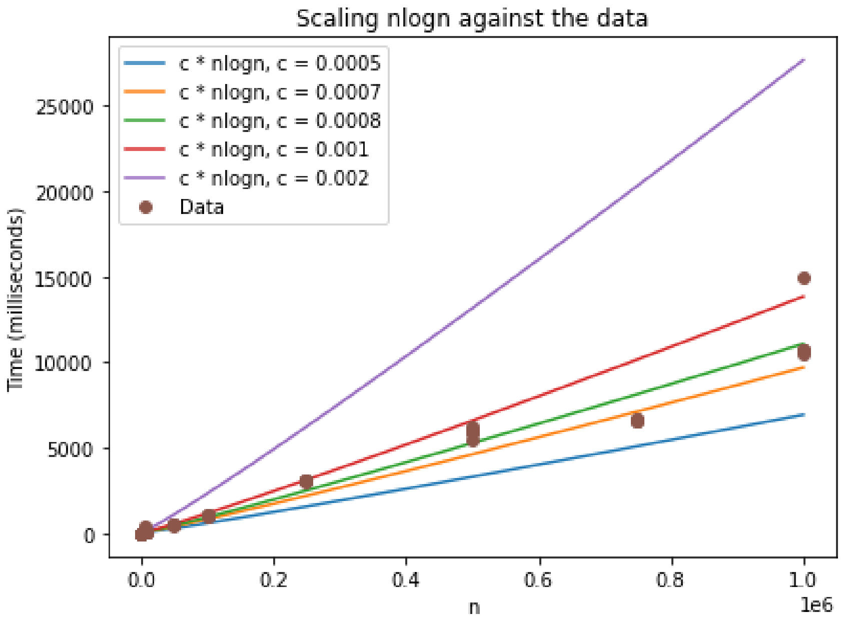
Raw values:

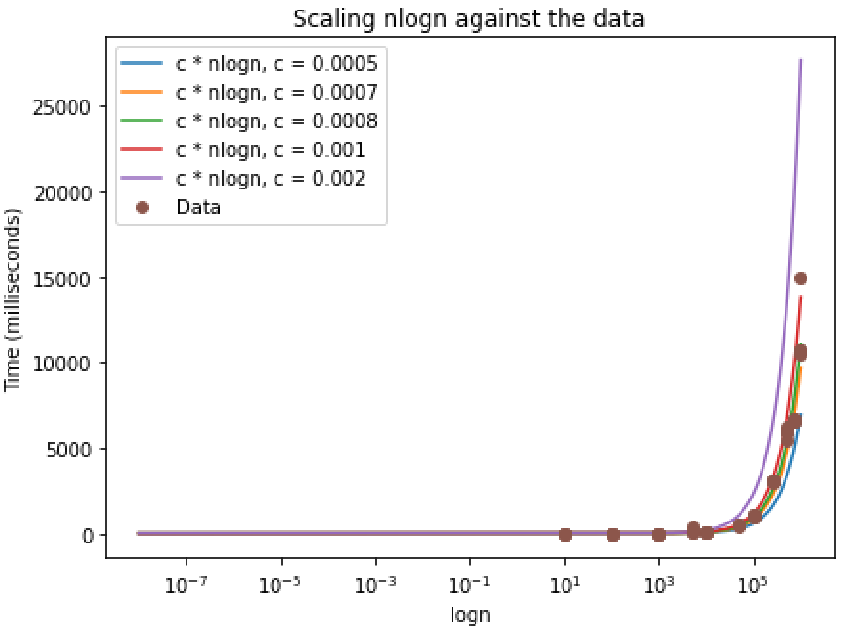


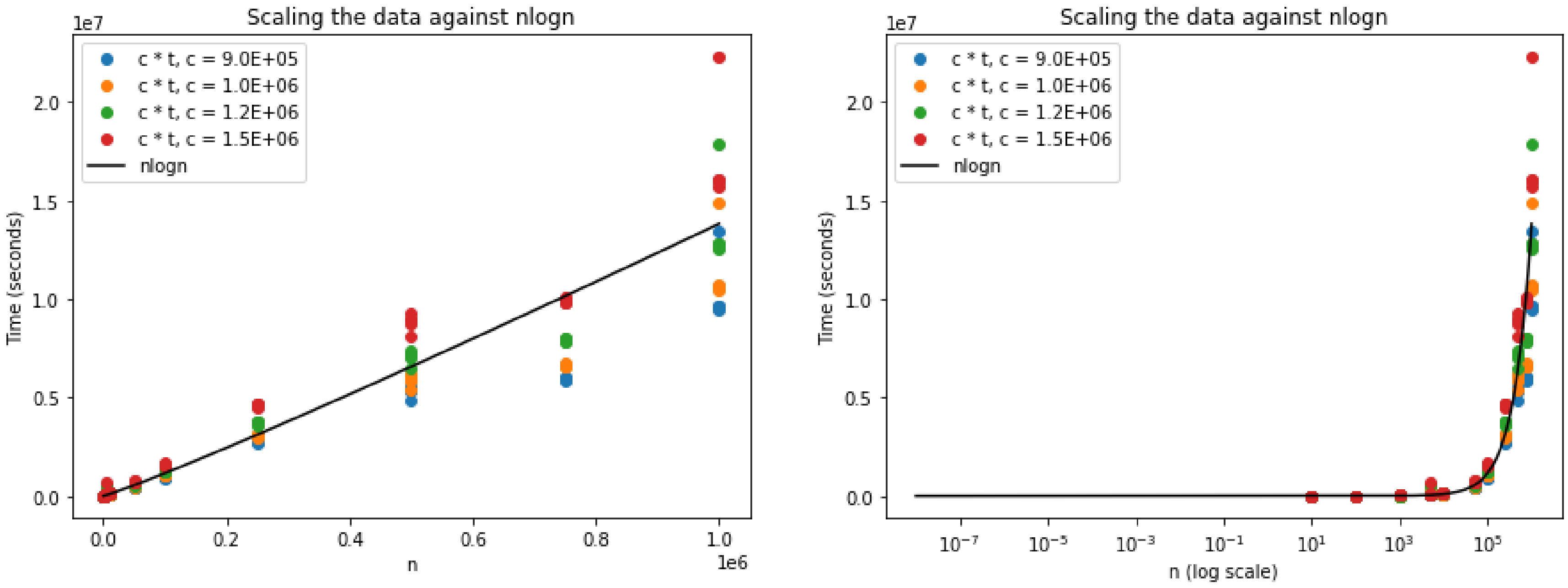
```
n = np.array([10, 10, 10, 10, 10, 100, 100, 100, 100, 100, 1000, 1000, 1000, 1000, 1000, 5000, 5000, 5000, 5000, 5000, 10000, 10000,
10000, 10000, 10000, 50000, 50000, 50000, 50000, 50000, 100000, 100000, 100000, 100000, 100000, 250000, 250000, 250000, 250000, 250000,
500000, 500000, 500000, 500000, 500000, 750000, 750000, 750000, 750000, 750000, 1000000, 1000000, 1000000, 1000000, 1000000])
t = np.array([0, 0, 0, 0, 0, 0, 0.001, 0.001, 0.001, 0.001, 0.01, 0.01, 0.011, 0.009, 0.012, 0.044, 0.046, 0.049, 0.44, 0.045, 0.101,
0.103, 0.104, 0.091, 0.106, 0.467, 0.482, 0.454, 0.507, 0.526, 1.129, 1.04, 1.012, 1.06, 1.02, 3.127, 2.967, 3.048, 3.142, 3.101, 5.855,
6.185, 5.992, 5.943, 5.412, 6.578, 6.56, 6.532, 6.687, 6.715, 10.679, 10.475, 10.697, 10.659, 14.893])
```

3. The empirical data does match the expected nlogn complexity. Rather than multiplying nlogn by c = 8.3e-07 (roughly the value of the proportionality constant), I found that it was more numerically stable to multiply my time values by the reciprocal 1/c (equivalent to measuring time in microseconds rather than seconds). This resulted in the following graphs, for various values of c.

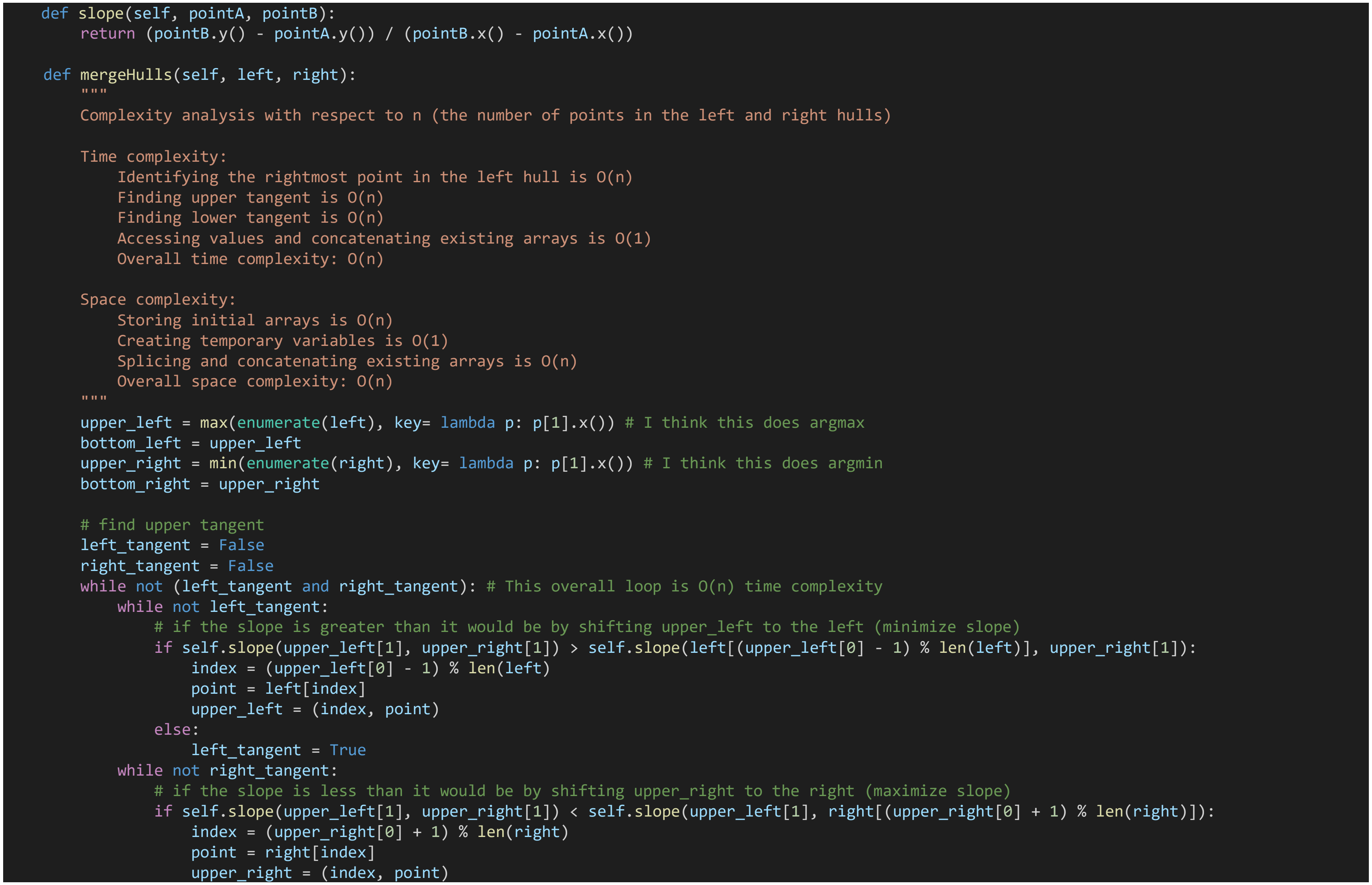
   Inevitably, I think it’s clear that there is a fair amount of variability in time due to how the CPU chooses to prioritize processes, when it chooses to deallocate memory, and so on. Additionally, there is always some fixed overhead of time, so the fit becomes more accurate at the limit. As such, I think that with sufficient trials of sufficient size, we could get our data to fit one of these curves much more nicely.


Scaling nlogn against the data
c * nlogn, c = 0.0005
c * nlogn, c = 0.0007
c * nlogn, c = 0.0008
c * nlogn, c = 0.001
c * nlogn, c = 0.002
Data
Time (milliseconds)
25000
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
n


Scaling nlogn against the data
c * nlogn, c = 0.0005
c * nlogn, c = 0.0007
c * nlogn, c = 0.0008
c * nlogn, c = 0.001
c * nlogn, c = 0.002
Data
Time (milliseconds)
25000
20000
15000
10000
5000
0
10^{-7}
10^{-5}
10^{-3}
10^{-1}
10^{1}
10^{3}
10^{5}
logn


Scaling the data against nlogn
1e7
c * t, c = 9.0E+05
c * t, c = 1.0E+06
c * t, c = 1.2E+06
c * t, c = 1.5E+06
nlogn
Time (seconds)
0.0
0.5
1.0
1.5
2.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
n
Scaling the data against nlogn
1e7
c * t, c = 9.0E+05
c * t, c = 1.0E+06
c * t, c = 1.2E+06
c * t, c = 1.5E+06
nlogn
Time (seconds)
0.0
0.5
1.0
1.5
2.0
10^{-7}
10^{-5}
10^{-3}
10^{-1}
10^{1}
10^{3}
10^{5}
n (log scale)

Appendix



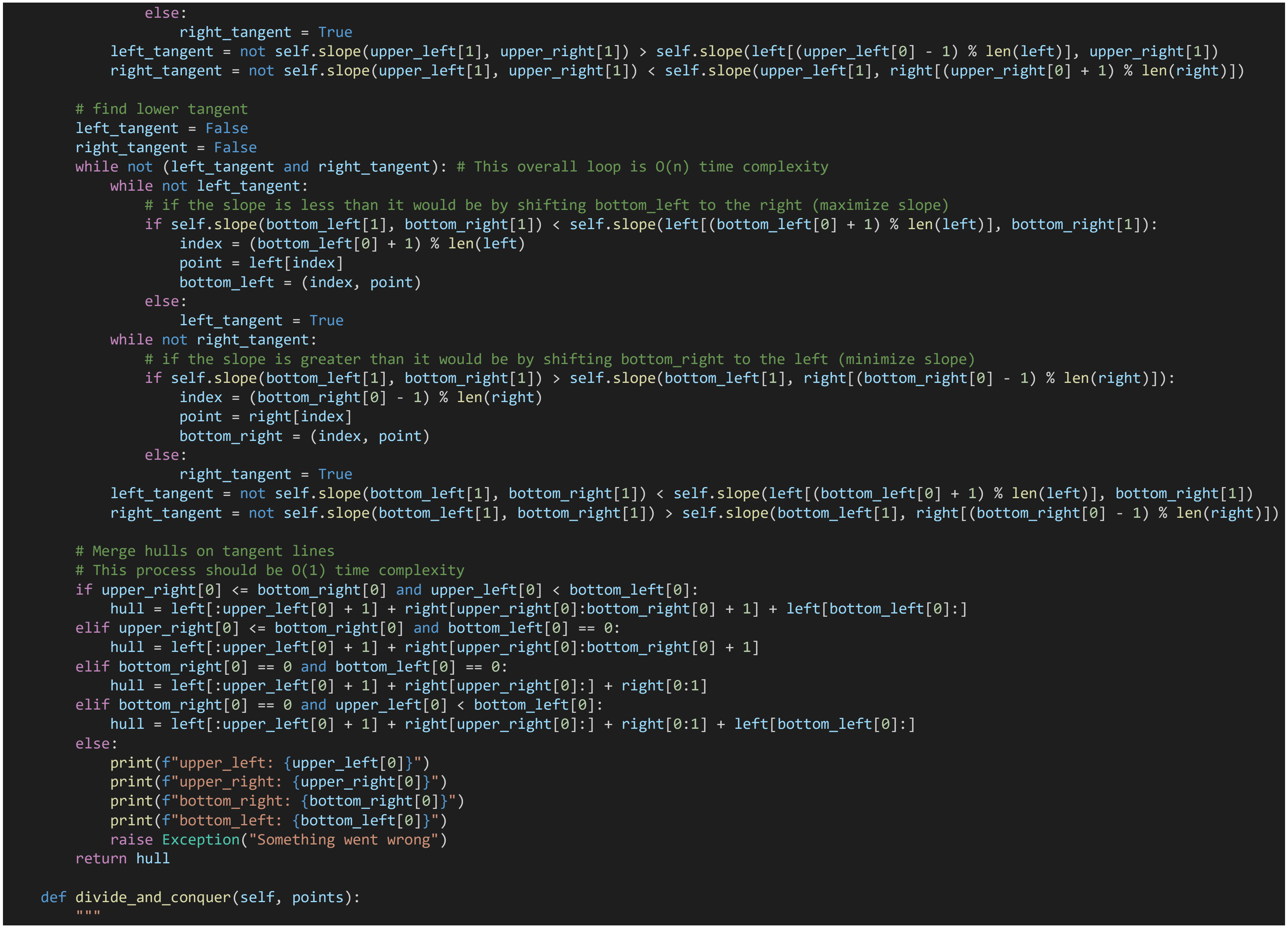
```python
    def slope(self, pointA, pointB):
        return (pointB.y() - pointA.y()) / (pointB.x() - pointA.x())

    def mergeHulls(self, left, right):
        """
        Complexity analysis with respect to n (the number of points in the left and right hulls)

        Time complexity:
            Identifying the rightmost point in the left hull is O(n)
            Finding upper tangent is O(n)
            Finding lower tangent is O(n)
            Accessing values and concatenating existing arrays is O(1)
            Overall time complexity: O(n)

        Space complexity:
            Storing initial arrays is O(n)
            Creating temporary variables is O(1)
            Splicing and concatenating existing arrays is O(n)
            Overall space complexity: O(n)
        """
        upper_left = max(enumerate(left), key= lambda p: p[1].x()) # I think this does argmax
        bottom_left = upper_left
        upper_right = min(enumerate(right), key= lambda p: p[1].x()) # I think this does argmin
        bottom_right = upper_right

        # find upper tangent
        left_tangent = False
        right_tangent = False
        while not (left_tangent and right_tangent): # This overall loop is O(n) time complexity
            while not left_tangent:
                # if the slope is greater than it would be by shifting upper_left to the left (minimize slope)
                if self.slope(upper_left[1], upper_right[1]) > self.slope(left[(upper_left[0] - 1) % len(left)], upper_right[1]):
                    index = (upper_left[0] - 1) % len(left)
                    point = left[index]
                    upper_left = (index, point)
                else:
                    left_tangent = True
            while not right_tangent:
                # if the slope is less than it would be by shifting upper_right to the right (maximize slope)
                if self.slope(upper_left[1], upper_right[1]) < self.slope(upper_left[1], right[(upper_right[0] + 1) % len(right)]):
                    index = (upper_right[0] + 1) % len(right)
                    point = right[index]
                    upper_right = (index, point)
```

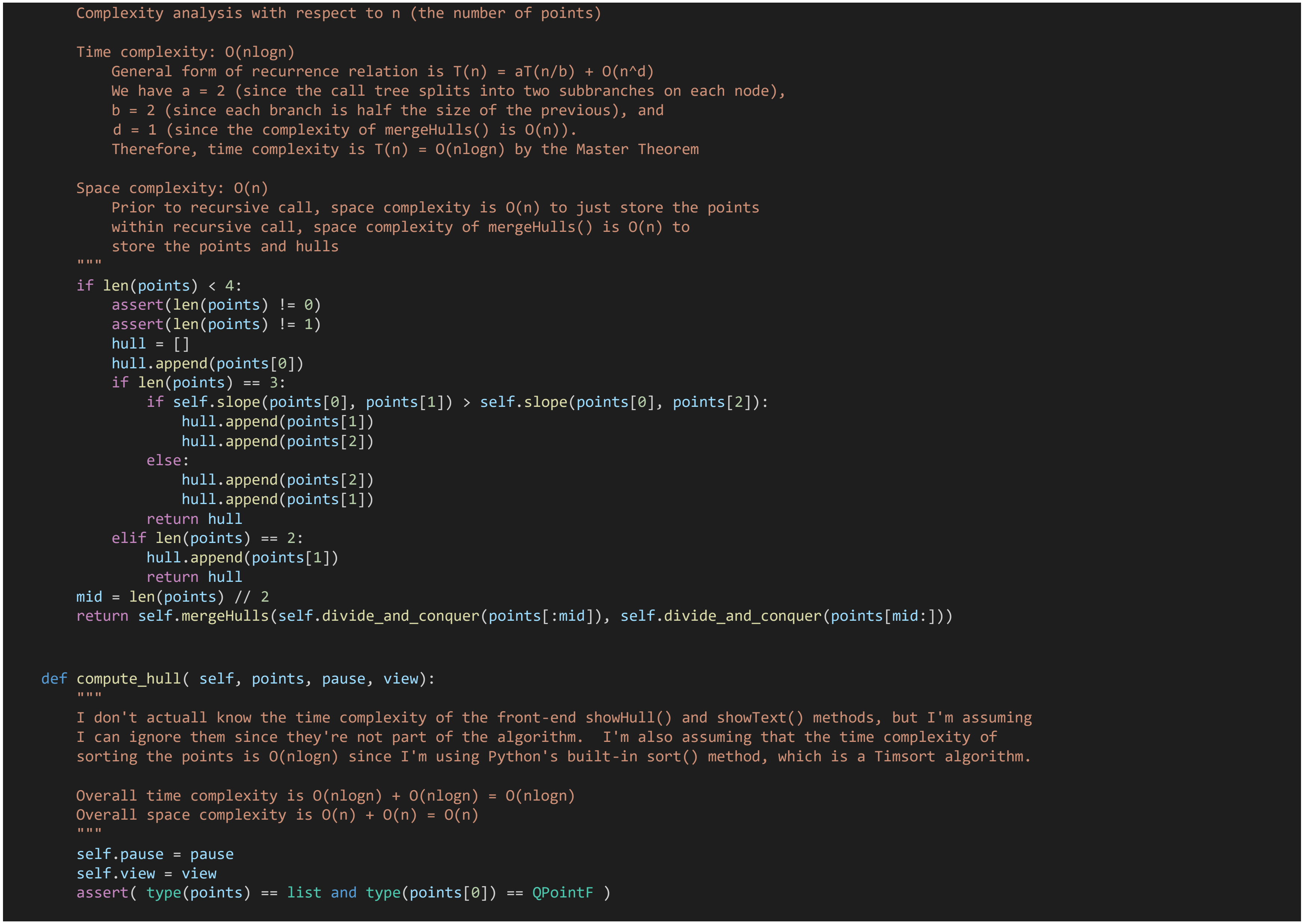


```
                else:
                    right_tangent = True
            left_tangent = not self.slope(upper_left[1], upper_right[1]) > self.slope(left[(upper_left[0] - 1) % len(left)], upper_right[1])
            right_tangent = not self.slope(upper_left[1], upper_right[1]) < self.slope(upper_left[1], right[(upper_right[0] + 1) % len(right)])

        # find lower tangent
        left_tangent = False
        right_tangent = False
        while not (left_tangent and right_tangent): # This overall loop is O(n) time complexity
            while not left_tangent:
                # if the slope is less than it would be by shifting bottom_left to the right (maximize slope)
                if self.slope(bottom_left[1], bottom_right[1]) < self.slope(left[(bottom_left[0] + 1) % len(left)], bottom_right[1]):
                    index = (bottom_left[0] + 1) % len(left)
                    point = left[index]
                    bottom_left = (index, point)
                else:
                    left_tangent = True
            while not right_tangent:
                # if the slope is greater than it would be by shifting bottom_right to the left (minimize slope)
                if self.slope(bottom_left[1], bottom_right[1]) > self.slope(bottom_left[1], right[(bottom_right[0] - 1) % len(right)]):
                    index = (bottom_right[0] - 1) % len(right)
                    point = right[index]
                    bottom_right = (index, point)
                else:
                    right_tangent = True
            left_tangent = not self.slope(bottom_left[1], bottom_right[1]) < self.slope(left[(bottom_left[0] + 1) % len(left)], bottom_right[1])
            right_tangent = not self.slope(bottom_left[1], bottom_right[1]) > self.slope(bottom_left[1], right[(bottom_right[0] - 1) % len(right)])

        # Merge hulls on tangent lines
        # This process should be O(1) time complexity
        if upper_right[0] <= bottom_right[0] and upper_left[0] < bottom_left[0]:
            hull = left[:upper_left[0] + 1] + right[upper_right[0]:bottom_right[0] + 1] + left[bottom_left[0]:]
        elif upper_right[0] <= bottom_right[0] and bottom_left[0] == 0:
            hull = left[:upper_left[0] + 1] + right[upper_right[0]:bottom_right[0] + 1]
        elif bottom_right[0] == 0 and bottom_left[0] == 0:
            hull = left[:upper_left[0] + 1] + right[upper_right[0]:] + right[0:1]
        elif bottom_right[0] == 0 and upper_left[0] < bottom_left[0]:
            hull = left[:upper_left[0] + 1] + right[upper_right[0]:] + right[0:1] + left[bottom_left[0]:]
        else:
            print(f"upper_left: {upper_left[0]}")
            print(f"upper_right: {upper_right[0]}")
            print(f"bottom_right: {bottom_right[0]}")
            print(f"bottom_left: {bottom_left[0]}")
            raise Exception("Something went wrong")
        return hull

    def divide_and_conquer(self, points):
        """
```



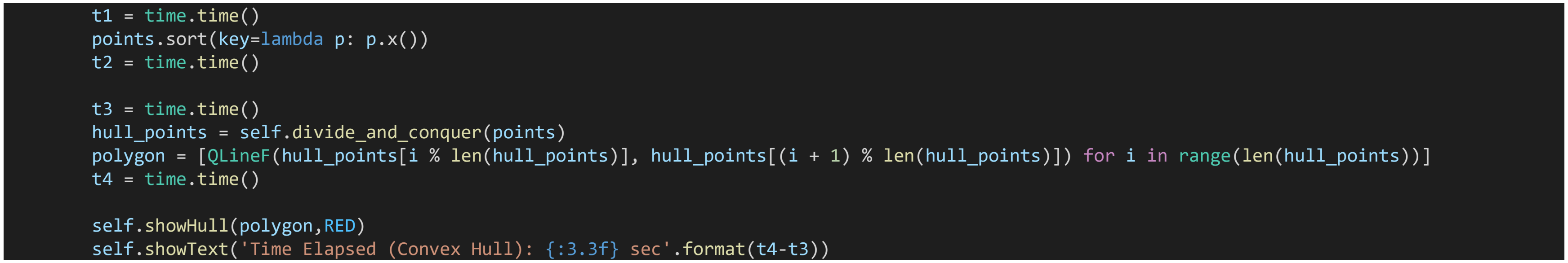
```
        Complexity analysis with respect to n (the number of points)

        Time complexity: O(nlogn)
            General form of recurrence relation is T(n) = aT(n/b) + O(n^d)
            We have a = 2 (since the call tree splits into two subbranches on each node),
            b = 2 (since each branch is half the size of the previous), and
            d = 1 (since the complexity of mergeHulls() is O(n)).
            Therefore, time complexity is T(n) = O(nlogn) by the Master Theorem

        Space complexity: O(n)
            Prior to recursive call, space complexity is O(n) to just store the points
            within recursive call, space complexity of mergeHulls() is O(n) to
            store the points and hulls
        """
        if len(points) < 4:
            assert(len(points) != 0)
            assert(len(points) != 1)
            hull = []
            hull.append(points[0])
            if len(points) == 3:
                if self.slope(points[0], points[1]) > self.slope(points[0], points[2]):
                    hull.append(points[1])
                    hull.append(points[2])
                else:
                    hull.append(points[2])
                    hull.append(points[1])
                return hull
            elif len(points) == 2:
                hull.append(points[1])
                return hull
        mid = len(points) // 2
        return self.mergeHulls(self.divide_and_conquer(points[:mid]), self.divide_and_conquer(points[mid:]))


    def compute_hull( self, points, pause, view):
        """
        I don't actuall know the time complexity of the front-end showHull() and showText() methods, but I'm assuming
        I can ignore them since they're not part of the algorithm.  I'm also assuming that the time complexity of
        sorting the points is O(nlogn) since I'm using Python's built-in sort() method, which is a Timsort algorithm.

        Overall time complexity is O(nlogn) + O(nlogn) = O(nlogn)
        Overall space complexity is O(n) + O(n) = O(n)
        """
        self.pause = pause
        self.view = view
        assert( type(points) == list and type(points[0]) == QPointF )
```



```
        t1 = time.time()
        points.sort(key=lambda p: p.x())
        t2 = time.time()

        t3 = time.time()
        hull_points = self.divide_and_conquer(points)
        polygon = [QLineF(hull_points[i % len(hull_points)], hull_points[(i + 1) % len(hull_points)]) for i in range(len(hull_points))]
        t4 = time.time()

        self.showHull(polygon,RED)
        self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4-t3))
```