Convex Hull Report

Time and Space Complexity Analysis

From lowest functions up (helper functions, merge, findConvexHull, and Sort), the complexity analysis will be given:

```python
def merge(self, hL, hR):
    assert (type(hL) == list and \
            type(hR) == list and \
            type(hL[0]) == QPointF and \
            type(hR[0]) == QPointF))

    # Find the rightmost point of the left hull (as an index)
    right_point_hL = hL[0]  # Start with the leftmost point for
    rightmost_hL_index = 0  # Hold the index of the rightmost po
    for i in range(len(hL)):
        if QPointF.x(hL[i]) > QPointF.x(right_point_hL):
            right_point_hL = hL[i]
            rightmost_hL_index += 1
            continue
        elif i != 0:  # We've found the rightmost point in the c                    nt time)
            break
```

This loop runs (in the worst case) n times where n is the number of points in the left hull.

Time Complexity: O(n)

Space Complexity: O(n)

```python
    # Find the leftmost point of the right hull (as an index)
    # By default that will always be the point at index 0
    leftmost_hR_index = 0

    # Make two copies of each index for finding upper and lower
    upp_tan_hL_index = rightmost_hL_index
    lower_tan_hL_index = rightmost_hL_index
    upp_tan_hR_index = leftmost_hR_index
    lower_tan_hR_index = leftmost_hR_index
```

Constant time

```python
# FIND UPPER TANGENT INDEXES FOR hL and hR
# Keep track of the previous lines slope, starting with the left/right-most points of either hull
prev_slope = self.slope_points(hL[rightmost_hL_index], hR[leftmost_hR_index])

# See-saw back and forth until the upper tangent is found
```

Slope function is constant time

```python
while True:
    left_side_stopped = False
    right_side_stopped = False

    if self.slope_points(hL[(upp_tan_hL_index - 1) % len(hL)], hR[upp_tan_hR_index]) < prev_slope:
        upp_tan_hL_index = (upp_tan_hL_index - 1) % len(hL)
        prev_slope = self.slope_points(hL[upp_tan_hL_index % len(hL)], hR[upp_tan_hR_index])
    else:
        left_side_stopped = True

    if self.slope_points(hL[upp_tan_hL_index], hR[(upp_tan_hR_index + 1) % len(hR)]) > prev_slope:
        upp_tan_hR_index = (upp_tan_hR_index + 1) % len(hR)
        prev_slope = self.slope_points(hL[upp_tan_hL_index], hR[upp_tan_hR_index % len(hR)])
    else:
        right_side_stopped = True

    if left_side_stopped and right_side_stopped:
        break
```

If we assume hull 1 and hull 2 have appx. the same num of points, this loop runs n/2 + n/2 = n times

Time Complexity: O(n)

Space Complexity: O(n)

```python
# FIND LOWER TANGENT INDEXES FOR hL and hR
# Keep track of the previous lines slope, starting with the left/right-most points of either hull
prev_slope = self.slope_points(hL[rightmost_hL_index], hR[leftmost_hR_index])

# See-saw back and forth until the lower tangent is found
while True:
    left_side_stopped = False
    right_side_stopped = False

    if self.slope_points(hL[(lower_tan_hL_index + 1) % len(hL)], hR[lower_tan_hR_index]) > prev_slope:
        lower_tan_hL_index = (lower_tan_hL_index + 1) % len(hL)
        prev_slope = self.slope_points(hL[lower_tan_hL_index % len(hL)], hR[lower_tan_hR_index])
    else:
        left_side_stopped = True

    if self.slope_points(hL[lower_tan_hL_index], hR[(lower_tan_hR_index - 1) % len(hR)]) < prev_slope:
        lower_tan_hR_index = (lower_tan_hR_index - 1) % len(hR)
        prev_slope = self.slope_points(hL[lower_tan_hL_index], hR[lower_tan_hR_index % len(hR)])
    else:
        right_side_stopped = True

    if left_side_stopped and right_side_stopped:
        break
```

This loop has the same space/time complexity except it runs for the lower tangent

Time Complexity: O(n)

Space Complexity: O(n)

```python
# MAKE LIST OF NEW HULL POINTS
# Move in quadrant-like fashion moving in a clockwise direction
new_hull = []
i = 0
new_hull.append(hL[i])
while i % len(hL) != upp_tan_hL_index:
    if i % len(hL) != 0:
        new_hull.append(hL[i % len(hL)])
    i += 1

if i != 0 and upp_tan_hL_index != lower_tan_hL_index:
    new_hull.append(hL[i % len(hL)])

i = upp_tan_hR_index
new_hull.append(hR[i])
while i % len(hR) != lower_tan_hR_index:
    if i % len(hR) != upp_tan_hR_index:
        new_hull.append(hR[i % len(hR)])
    i += 1

if i % len(hR) != upp_tan_hR_index and upp_tan_hR_index != lower_tan_hR_index:
    new_hull.append(hR[i % len(hR)])

i = lower_tan_hL_index
if lower_tan_hL_index != 0:
    while i % len(hL) != 0:
        new_hull.append(hL[i % len(hL)])
        i += 1

return new_hull
```

In worst case, all points from both hulls will need to be added, therefore each point is visited once, or 2n visits. Though a new array is created, it cannot be bigger than either hull combined (2n).

Time Complexity: O(n)

Space Complexity: O(n)

Total time/space complexity for merge is O(*n*) where *n* is the number of points in each hull.

```python
# RECURSIVE CONVEX HULL ALGORITHM SOLVER
def findConvexHull(self, points):
    assert (type(points) == list and type(points[0]) == QPointF)

    # Base Case when n < 4
    if len(points) < 4:

        # If there's only one point listed, then that is the hul
        # If there's two points, then only one line can be drawn
        # thus it's already in order of decreasing slope and ord
        # leftmost x-value at index 0.
        # No sorting needed
        if len(points) == 1 or len(points) == 2:
            return points

        # Sort points by decreasing slope
        # Index 0 is always going to be the "anchor" point for comparing slopes

        # Make a list of available lines
        lines = []
        for i in range(1, len(points)):
            lines.append(QLineF(points[0], points[i]))

        # Since there's only going to be a maximum of two lines, we can just swap the
        # two if out of order
        if self.slope(lines[0]) < self.slope(lines[1]):
            lines[0], lines[1] = lines[1], lines[0]

        # Recreate the array of points ordered by decreasing slope
        new_points = [QLineF.p1(lines[0]), QLineF.p2(lines[0]), QLineF.p2(lines[1])]

        return new_points

    subset_len = math.floor(len(points) / 2)
    hull1 = self.findConvexHull(points[: subset_len])
    hull2 = self.findConvexHull(points[subset_len:])
    return self.merge(hull1, hull2)
```

All operations in the base case run in constant time. New memory is not needed in the recursive process

Time Complexity: O(1)

Space Complexity: O(n)

findConvexHull() is called log(*n*) times, where *n* is the number of points in the graph. For each call, merge() is called with a time complexity of O(*n*) where *n* is the number of points in each hull.

Time Complexity: O(n*log(n))

Space Complexity: O(n)

The total time complexity for findConvexHull is O(*n* * log(*n*)) where *n* is the number of points in the graph. Total space complexity is O(*n*) because no new memory aside from the initial array of points and that of the hulls at each recursive step is needed.

```python
# QUICKSORT FUNCTIONS
def partition(self, points, low, high):
    i = (low - 1)
    pivot = points[high]

    for j in range(low, high):  # Only sort by x value
        if QPointF.x(points[j]) < QPointF.x(pivot):
            i = i + 1
            points[i], points[j] = points[j], points[i]

    points[i + 1], points[high] = points[high], points[i + 1]
    return i + 1

def quickSort(self, points, low, high):
    if low < high:
        pi = self.partition(points, low, high)
        self.quickSort(points, low, pi - 1)
        self.quickSort(points, pi + 1, high)

def sort(self, points):
    assert (type(points) == list and type(points[0]) == QPointF)

    n = len(points)
    self.quickSort(points, 0, n - 1)
```

This is a traditional implementation of a quicksort algorithm with time/space complexity:

Time Complexity: O(n*log(n))

Space Complexity: O(n)

The total time/space complexity for the sorting of points by x-value is O(*n* * log(*n*)) and O(*n*), respectively.

Therefore, to first sort the points according to x-value and then compute the convex hull, the time complexity is O(*n* * log(*n*) + *n* * log(*n*)) = O(2 * *n* * log(*n*)) = O (*n* * log(*n*)), where *n* is the number of points in the graph. The space complexity is O(*n*), where *n* is the number of points in the graph.

The analysis of the algorithm in terms of pseudocode and worst-case time efficiency is given:

findConvexHull(points) :

if the number of points is less than 4

if there are only 1 or 2 points return the list of points as is

else

sort the 3 points by decreasing slope

// Since the number of points to sort does not change depending on the size of the graph, this counts as constant time. The above section runs in constant time.

Else

Find the left hull by calling findConvexHull on the left half of the points list

Find the right hull by calling findConvexHull on the right half of the points list

Return the two hulls merged (by calling merge(hull1, hull2))

// findConvexHull is recursively called log(*n*) times, and after each call, merge is performed, taking log(*n*) time. Thus, finding the convex hull of a graph of points takes O(*n**log(*n*)) time.

The size of the problem is cut in half at each level and the work to be done at each level is equal to *n*, or the length of the list. Therefore, using the Master Theorem, the Big-Oh runtime of the algorithm is O(*n**log(*n*)).

Empirical Analysis

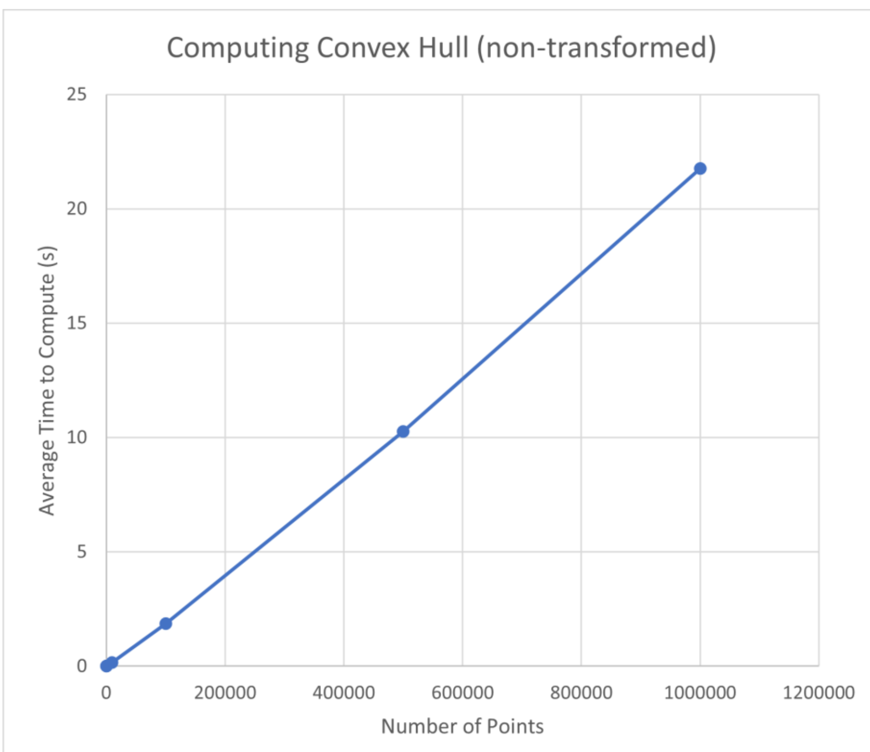
Trials were automated, calculated, and exported with this code:

```
iteration = 0
for i in [10, 100, 10000, 100000, 500000, 1000000]:
    for j in range(5):
        points = [QPointF(random.uniform(-1, 1), random.uniform(-1, 1)) for k in range(i)]
        trial = time_trial
        time1, time2, time3, time4 = trial.compute_hull_time_trial(trial, points)
        dataframe = pd.DataFrame({'num_points': i, 'algorithm_time': (time4 - time1)},
                                 index=[iteration])
        # df = df.append({'num_points': i, 'sort_time': (time2 - time1), 'convex_time': (time4 - time3)}, ignore_index=True)
        with open('C:\Cameron USB Restore\BYU\Winter 2021\CS 312\Convex Hull\empirical-analysis.csv', 'a') as f:
            dataframe.to_csv(f, header=False)
        iteration += 1
```

This produced the following data:

Run	Points	Time		Points	Average Time
0	10	0.000502		10	0.0001
1	10	0		100	0.001399
2	10	0		10000	0.1721
3	10	0		100000	1.862199
4	10	0		500000	10.2642
5	100	0.0015		1000000	21.772
6	100	0.000999			
7	100	0.001499			
8	100	0.0015			

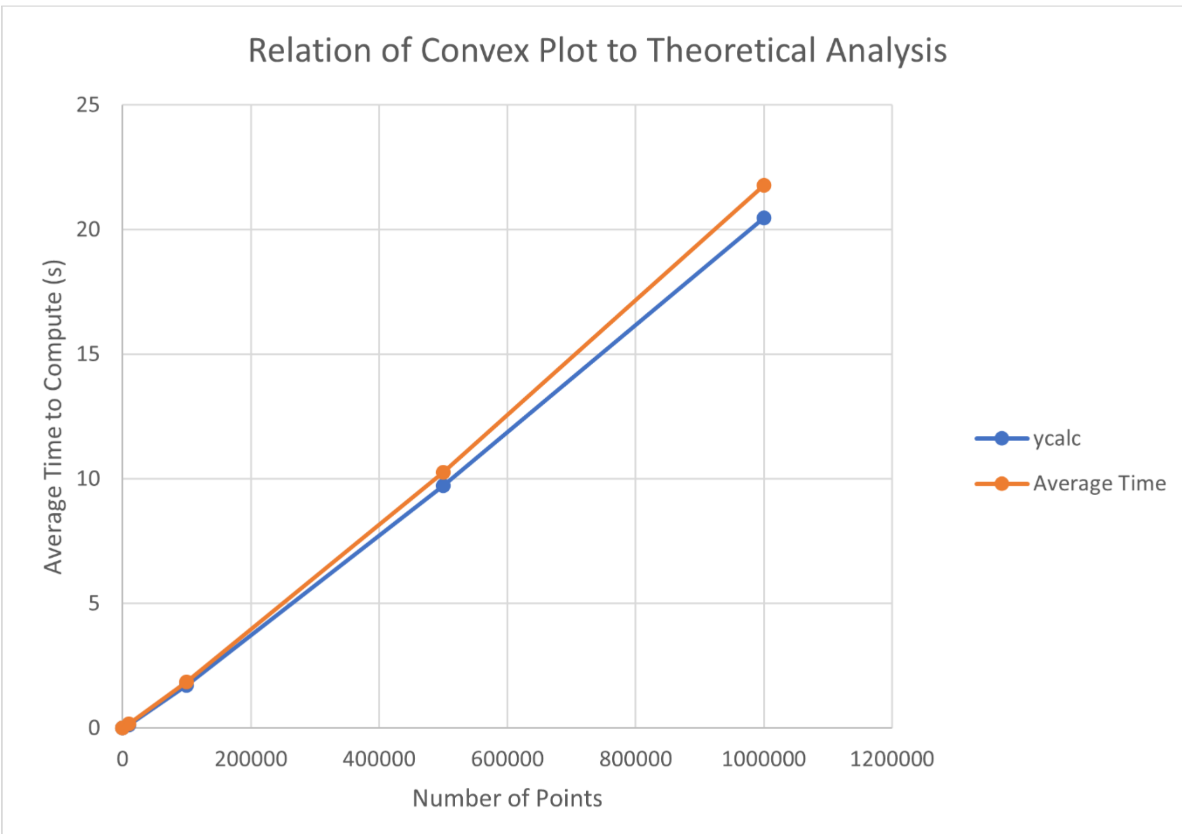
9	100	0.001498			
10	10000	0.182001			
11	10000	0.168			
12	10000	0.170501			
13	10000	0.163999			
14	10000	0.176			
15	100000	1.861501			
16	100000	1.841499			
17	100000	1.847497			
18	100000	1.848999			
19	100000	1.911499			
20	500000	10.1915			
21	500000	10.47			
22	500000	10.2845			
23	500000	10.2665			
24	500000	10.1085			
25	1000000	21.268			
26	1000000	22.36			
27	1000000	21.688			
28	1000000	22.145			
29	1000000	21.399			



The shape of the graph fits an *n* * log(*n*) shape best, so fitting a curve of that order of growth seems appropriate.

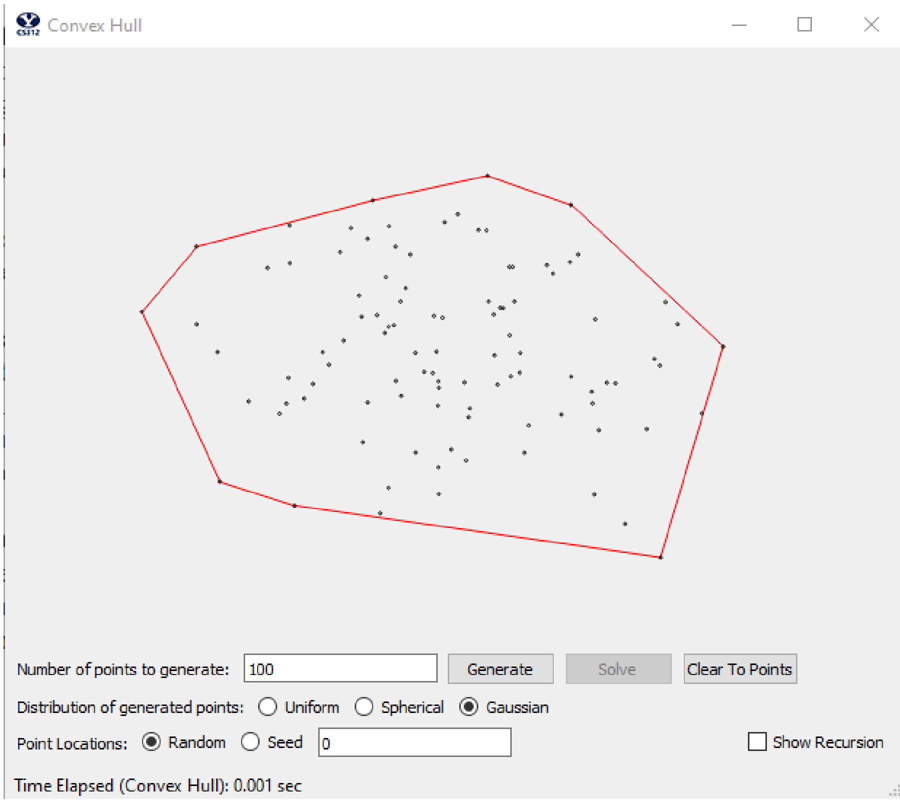
By using the equation form CH(Q) = k * n * log(n) and minimizing sum of squared residuals, the best-approximation values were found:

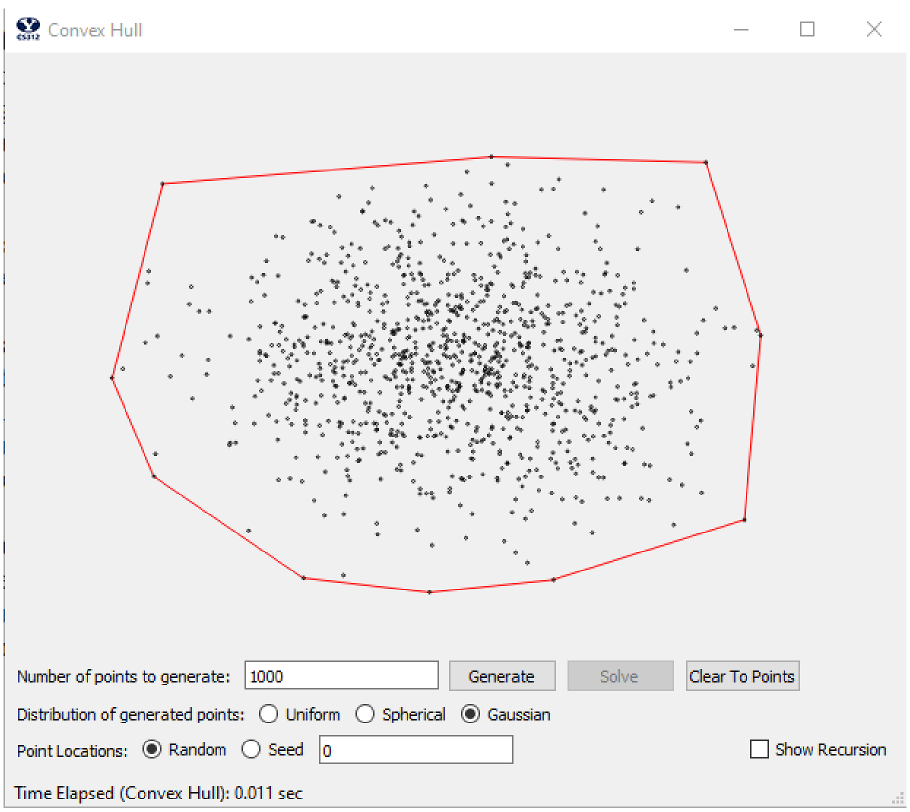
equation: k * n * log(n)			
k	1.03E-06		Sum Sq:
			1.994939696



The constant of proportionality k = 1.03E-06. The approximated curve seems to match the recorded data well. Although, there seems to be a widening gap as the number of points gets larger. This probably has less to do with the algorithm not being nlog(n) but rather more to do with hardware limitations as data sets become increasingly larger.

Examples


Convex Hull
Number of points to generate:
100
Generate
Solve
Clear To Points
Distribution of generated points:
Uniform
Spherical
Gaussian
Point Locations:
Random
Seed
0
Show Recursion
Time Elapsed (Convex Hull): 0.001 sec


Convex Hull
Number of points to generate:
1000
Generate
Solve
Clear To Points
Distribution of generated points:
Uniform
Spherical
Gaussian
Point Locations:
Random
Seed
0
Show Recursion
Time Elapsed (Convex Hull): 0.011 sec