Noah Schill

CS 312 Spring 2021

Convex Hull

1. Partially functioning code in Appendix
2. Theoretical Analysis:

The algorithm finding the convex hull given n points consists of two main parts: a *convex_hull* function and a *merge* function. *convex_hull* calls *merge* and itself recursively.

Procedure *convex_hull* splits the list of sorted points in half and then calls itself on each of these halves. Therefore, each recursive call takes half as long as its calling stack, introducing an O(log n) complexity.

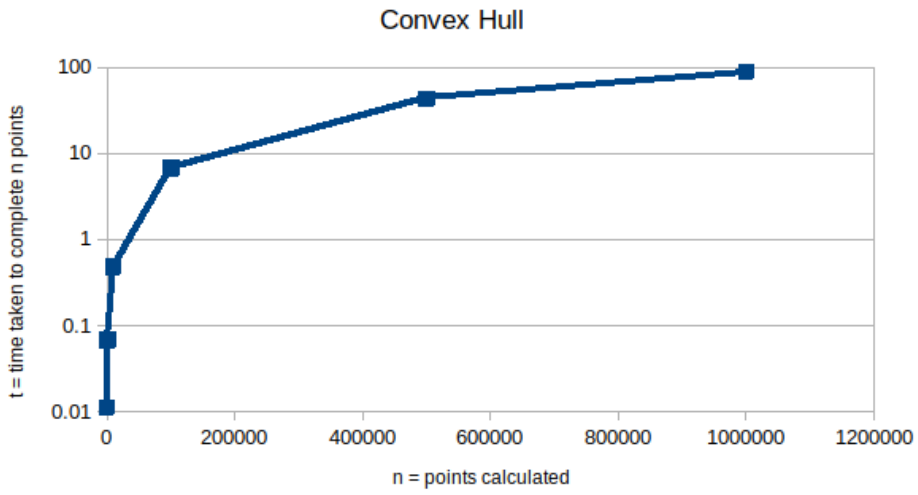
The main task of *merge* is to find an upper and lower tangent between two polygons and to join them together. The upper and lower tangent methods start from the inner points and work their ways outwards until slope decreases. In a worst-case scenario, these will take O(n/2) time or O(1) if the first guess is correct. Running both upper and lower tangent is equivalent to: O(n/2) + O(n/2) = O(n).

Bringing this together with regards to the master theorem, we can say:
a = 2, as we are creating 2 sub-tasks with each recursive call
b = 2, as the time halves with each recursive call
d = 1, as it takes each recursive call is O(n).
T(n) = 2T(n/2)+O(n)
a/b^1 = 2/2 = 1 $\rightarrow$ O(n^d log n) = O(n log n)

This intuitively makes sense, as we are running our O (log n) function n times.

3. Empirical Analysis:

n	time(s)	mean time
10	1E-24	1E+025
100	0.011	9090.909091
1000	0.068	14705.88235
10000	0.481	20790.02079
100000	6.718	14885.38255
500000	43.276	11553.74804
1000000	88.378	11315.03315



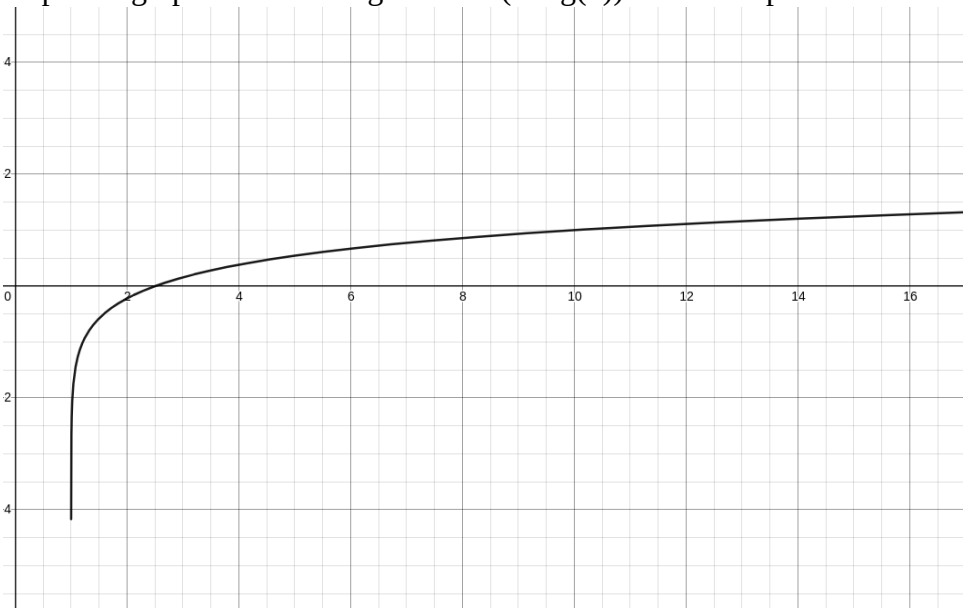
The order of growth which fits best is undoubtedly O (n log n). The reasoning behind the theoretical analysis would point to this being accurate.

By my estimate using empirical data, the constant of proportionality converged at about 8.838E-5.

4. Observations

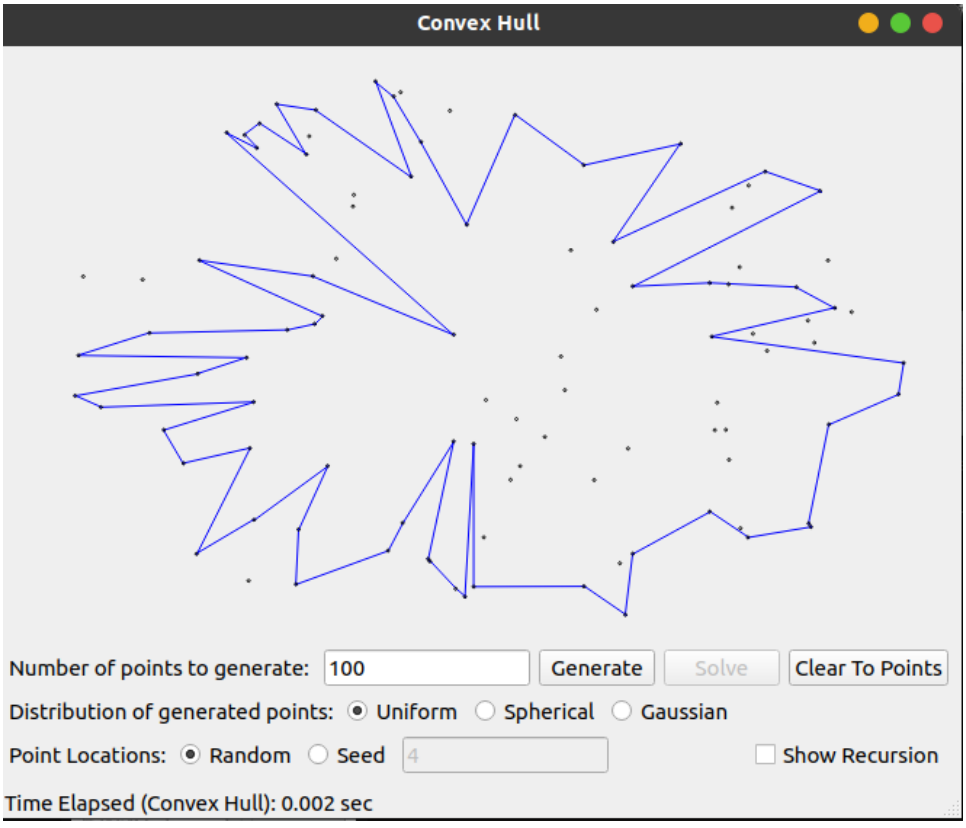
It is fascinating how closely we can predict the behavior and demands of an algorithm using asymptotic notation, the master theorem, among other tools. No major differences. The above

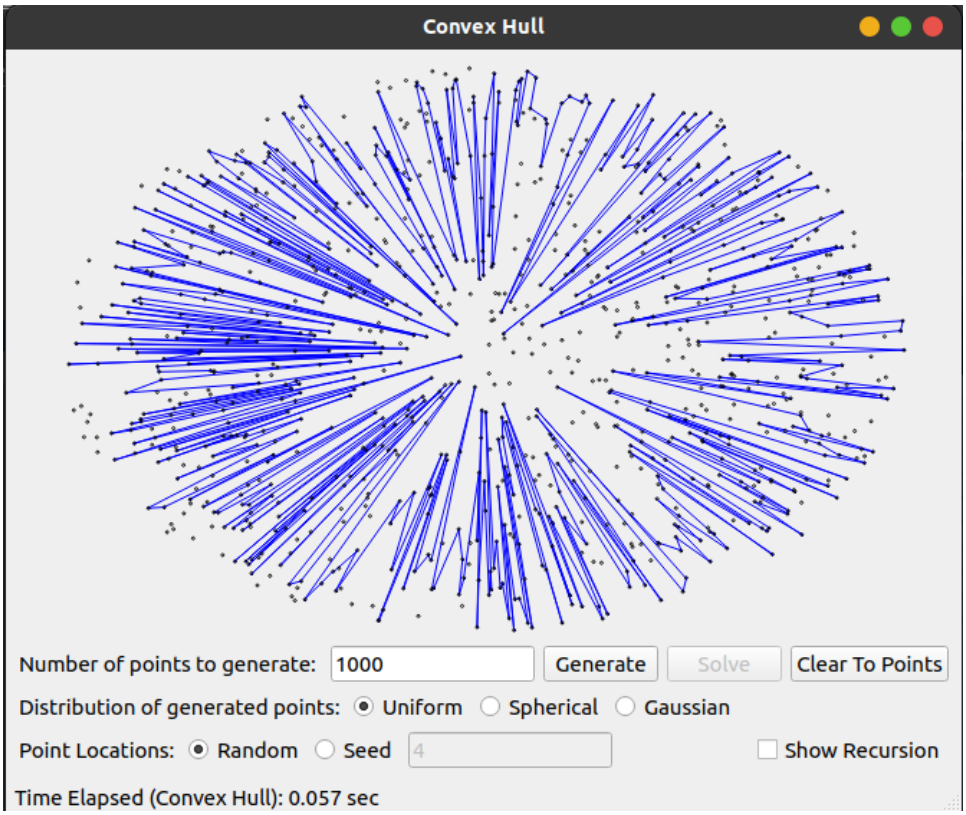
empirical graph looks like a generic O (n log(n)) function I plotted in Desmos below:



5. Screenshots

These are obviously not completely functional, but hopefully the code demonstrates knowledge of divide-and-conquer concepts even if the implementation is poor.





Appendix A -- Python Code:

```
def order_clockwise(self, p):
    points = []
    for point in p:
        points.append((point.x(), point.y()))
    sorted_points = sorted(points, key=lambda point: atan2(point[1], point[0]),
reverse=True)
    out = []
    for point in sorted_points:
        q = QPointF();
        q.setX(point[0])
        q.setY(point[1])
        out.append(q)
    return out


def get_leftmost_index(self, points):
    leftmost_point = points[0]
    for i in range(1, len(points)):
        if points[i].x() < leftmost_point.x():
            leftmost_point = points[i]

    return points.index(leftmost_point)


def get_rightmost_index(self, points):
    rightmost_point = points[0]
    for i in range(1, len(points)):
        if points[i].x() > rightmost_point.x():
            rightmost_point = points[i]

    return points.index(rightmost_point)


def convex_hull(self, sorted_points):
    if len(sorted_points) <= 3:
        return self.order_clockwise(sorted_points)  # return this in clockwise
ordering

    l = self.convex_hull(sorted_points[0:len(sorted_points) // 2])
    r = self.convex_hull(sorted_points[(len(sorted_points) // 2):])
    hull_points = self.merge(l, r)
    hull = self.order_clockwise(hull_points)
    return hull


def merge(self, l, r):
    upper_tangent = self.find_upper_tangent(l, r)
    lower_tangent = self.find_lower_tangent(l, r)
    # hull = [upper_tangent[0], upper_tangent[1], lower_tangent[0],
lower_tangent[1]]
    # hull = [lower_tangent[0], lower_tangent[1]]
    hull = l[l.index(lower_tangent[0]):]
    hull = hull + l[0:l.index(upper_tangent[0])]
    hull.append(upper_tangent[0])
    hull = hull + r[r.index(upper_tangent[1]):]
```

```
    hull = hull + r[0:r.index(lower_tangent[1]) - 1]

    return hull


def find_upper_tangent(self, l, r):
    l_idx = self.get_rightmost_index(l)
    r_idx = self.get_leftmost_index(r)

    prev_l = []
    prev_r = []
    is_left_tangent, is_right_tangent = False, False

    slope = float('inf')
    while (not is_left_tangent) and (not is_right_tangent):
        while not is_left_tangent:
            new_slope = self.get_slope(l[l_idx], r[r_idx])
            if new_slope > slope:
                l_idx = prev_l.pop()
                is_left_tangent = True
                is_right_tangent = False
            else:
                prev_l.append(l_idx)
                l_idx -= 1
                if l_idx < 0:
                    l_idx = len(l) - 1
                slope = new_slope

        slope = float('-inf')
        r_idx = self.get_leftmost_index(r)

        while not is_right_tangent:
            new_slope = self.get_slope(l[l_idx], r[r_idx])
            if new_slope < slope:
                r_idx = prev_r.pop()
                is_right_tangent = True
                is_left_tangent = False
            else:
                prev_r.append(r_idx)
                r_idx += 1
                if r_idx > len(r) - 1:
                    r_idx = 0
                slope = new_slope

    return [l[l_idx], r[r_idx]]


def find_lower_tangent(self, l, r):
    l_idx = self.get_rightmost_index(l)
    r_idx = self.get_leftmost_index(r)
    prev_l = []
    prev_r = []
    is_left_tangent, is_right_tangent = False, False

    slope = float('-inf')

    while (not is_left_tangent) and (not is_right_tangent):
        while not is_left_tangent:
```

```
            new_slope = self.get_slope(l[l_idx], r[r_idx])
            if new_slope < slope:  # If slope decreases, we've found it.
                l_idx = prev_l.pop()
                is_left_tangent = True
            else:
                prev_l.append(l_idx)
                l_idx -= 1
                if l_idx > len(l) - 1:
                    l_idx = 0
                slope = new_slope
                is_right_tangent = False

        slope = float('inf')
        r_idx = self.get_leftmost_index(r)

        while not is_right_tangent:
            new_slope = self.get_slope(l[l_idx], r[r_idx])
            if new_slope > slope:
                r_idx = prev_r.pop()
                is_right_tangent = True
            else:
                prev_r.append(r_idx)
                r_idx -= 1
                if r_idx < 0:
                    r_idx = len(r) - 1
                slope = new_slope
                is_left_tangent = False

    return [l[l_idx], r[r_idx]]


def get_slope(self, p1, p2):
    if p1 == p2:
        return float('inf')
    return (p2.y() - p1.y()) / (p2.x() - p1.x())
```