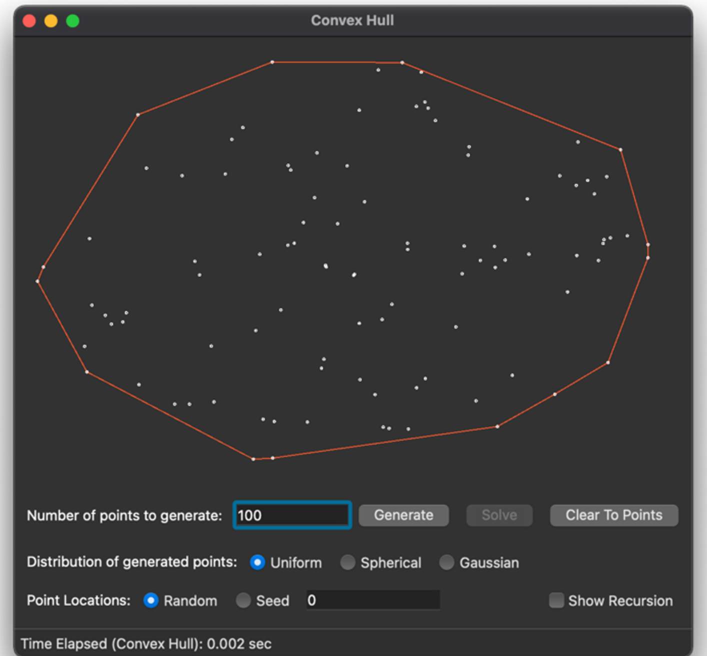
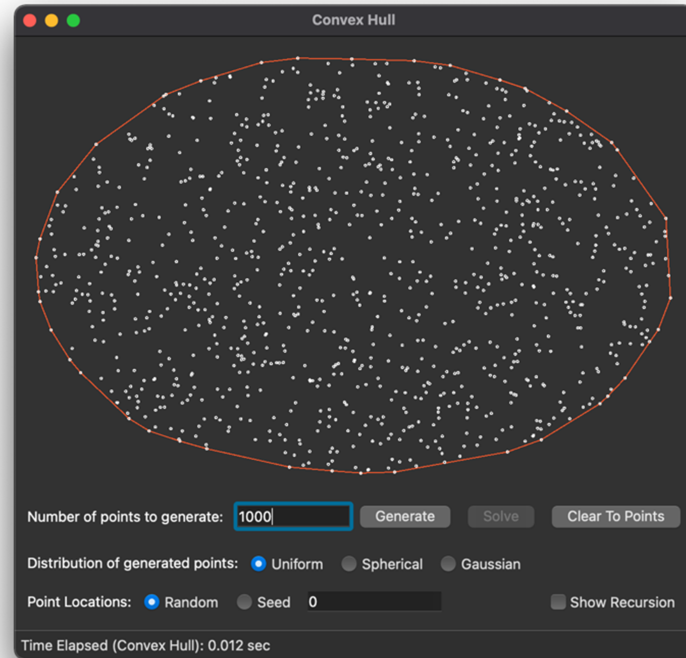
CONVEX HULL – DIVIDE AND CONQUER





Pseudocode:

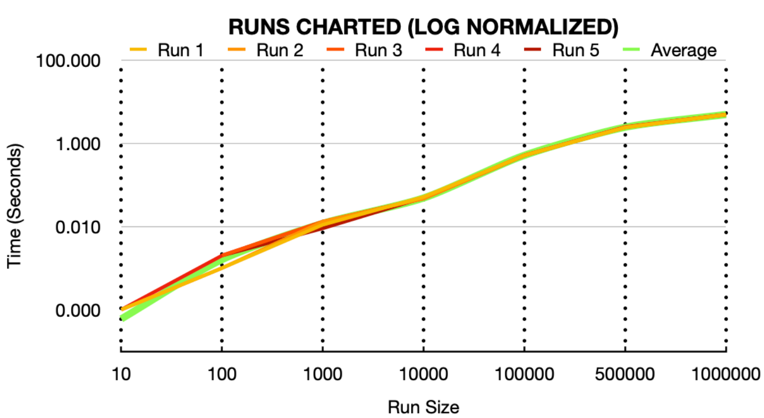
CONVEX HULL
<u>Sort all *points* by their x-values</u>
(Selection sort) O(nlog(n))
<u>Divide and Conquer</u>
Call convex hull for each sub-array until there's only 1-2 points in the data O(nlog(n)
Connect the edges in the lowest case to form a proto hull
<u>Create new hull</u>
<u>Upper Tangent:</u> O(n)
Get the rightmost point of the left hull and leftmost point of the right hull
Draw an edge between the points
Going counterclockwise on the left, check to find the most negative slope from left to right
Going clockwise on the right, check to find the most positive slope from right to left
If a slope changes, keep going back and forth between the left and right hull until nothing changes
We should have the upper tangent at this stage
<u>Lower Tangent:</u> O(n)
Get the rightmost point of the left hull and leftmost point of the right hull
Draw an edge between the points
Going clockwise on the left, check to find the most positive slope from left to right
Going counterclockwise on the right, check to find the most negative slope from right to left
If a point changes, keep going back and forth between the left and right hull until nothing changes
We should have the lower tangent at this stage
Delete all inside points and edges
Hull should be completed O(n)
Continue through the recursion
Return Final Hullre

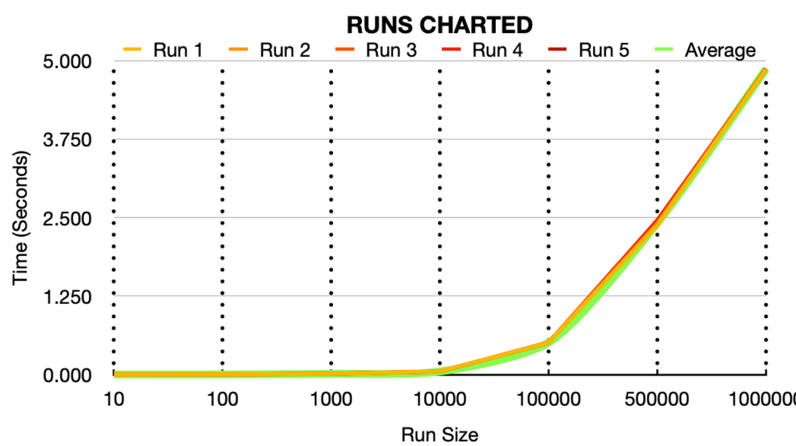
Theoretical Analysis:

Theoretically this program should have a total time complexity of O(nlog(n)). Technically both the python sort we are using and the divide and conquer method we are utilizing are the same complexity, but we can simplify this by using the maximum time and using that as the overall maximum of the program. The merging of the left and right hulls is O(n) time and because we are dividing and conquering, we can use the recurrence relations and the master theorem to confirm that it ends up having a final time complexity of O(nlog(n)).

Empirical Analysis:

RUN	10	100	1000	10000	100000	500000	1000000
Run 1	0.000	0.001	0.011	0.051	0.507	2.373	4.860
Run 2	0.000	0.001	0.013	0.047	0.510	2.402	4.861
Run 3	0.000	0.002	0.013	0.047	0.512	2.405	4.852
Run 4	0.000	0.002	0.013	0.047	0.509	2.458	4.860
Run 5	0.000	0.002	0.009	0.049	0.507	2.407	4.874
AVG	0.0000	0.0016	0.0118	0.0482	0.5090	2.4090	4.8614





Discussion

Overall, I think it is safe to say that the algorithm that I implemented follows the O(nlog(n)) complexity that was established in our theoretical analysis. When we normalize the graph of runs in our empirical analysis, we note that it follows a linear pattern meaning that when we remove log(n) we are left with O(n) meaning that the overall complexity of my current implementation is O(nlog(n)).

With regards to space complexity, it should fall under O(n) because although it divides and conquers the array of points it never creates any new arrays, just splits, and moves so the overall memory pressure is somewhat low. The inefficiencies come from the overall overhead of the repeated stack calls and processing that comes from deep recursion.

Convex-Hull-DivAndConq – convex_hull.py

```python
def div_and_conq(self, points, pause, view):

    num_points = len(points)
    if num_points == 1:
        return points

    # CALL CONVEX HULL FOR EACH SUB-ARRAY UNTIL THERE'S ONLY 1-2 POINTS IN THE DATA
    left_hull = self.div_and_conq(points[:num_points // 2], pause, view)
    right_hull = self.div_and_conq(points[num_points // 2:], pause, view)

    # CONNECT THE EDGES IN THE LOWEST CASE
    if len(left_hull) == 1 and len(right_hull) == 1:
        left_hull.extend(right_hull)
        return left_hull

    # FIND THE RIGHT MOST POINT OF THE LEFT HULL AND THE LEFT MOST POINT OF THE RIGHT HULL
    # WORST CASE O(n)
    left_initial = left_hull.index(
        max(left_hull, key=lambda left_point: left_point.x()))
    right_initial = right_hull.index(
        min(right_hull, key=lambda right_point: right_point.x()))

    # FIND THE UPPER TANGENT -----------------------------------------------------------------------
    # WORST CASE O(n)
    i = left_initial
    j = right_initial
    left = True
    right = True
    slope = (right_hull[j].y() - left_hull[i].y()) / (right_hull[j].x() - left_hull[i].x())

    # CLIMB UP ALL THE POINTS IN THE HULL UNTIL YOU REACH THE TOP AS DECIDED BY THE SLOPE
    while left or right:
        left = False
        right = False
        while True:
            temp_slope = (right_hull[j].y() - left_hull[(i - 1) % len(left_hull)].y()) / (
                    right_hull[j].x() - left_hull[(i - 1) % len(left_hull)].x())

            if temp_slope < slope:
                left = True
                slope = temp_slope
                i = (i - 1) % len(left_hull)
            else:
                break

        while True:
            temp_slope = (right_hull[(j + 1) % len(right_hull)].y() - left_hull[i].y()) / (
                    right_hull[(j + 1) % len(right_hull)].x() - left_hull[i].x())

            if temp_slope > slope:
                right = True
                slope = temp_slope
                j = (j + 1) % len(right_hull)
            else:
                break

    upper_tangent = (i, j)
```

Convex-Hull-DivAndConq – convex_hull.py

```python
    # FIND THE LOWER TANGENT -----------------------------------------------------------------------
    # WORST CASE O(n)
    i = left_initial
    j = right_initial
    left = True
    right = True
    slope = (right_hull[j].y() - left_hull[i].y()) / (right_hull[j].x() - left_hull[i].x())

    # CLIMB UP ALL THE POINTS IN THE HULL UNTIL YOU REACH THE TOP AS DECIDED BY THE SLOPE
    while left or right:
        left = False
        right = False
        while True:
            temp_slope = (right_hull[j].y() - left_hull[(i + 1) % len(left_hull)].y()) / (
                    right_hull[j].x() - left_hull[(i + 1) % len(left_hull)].x())
            if temp_slope > slope:
                left = True
                slope = temp_slope
                i = (i + 1) % len(left_hull)
            else:
                break

        while True:
            temp_slope = (right_hull[(j - 1) % len(right_hull)].y() - left_hull[i].y()) / (
                    right_hull[(j - 1) % len(right_hull)].x() - left_hull[i].x())
            if temp_slope < slope:
                right = True
                slope = temp_slope
                j = (j - 1) % len(right_hull)
            else:
                break

    lower_tangent = (i, j)

    # SHOW RECURSION IF SELECTED
    # IGNORED FROM O(n) CALCULATION
    if pause:
        self.show_recursive_hull(left_hull, right_hull, upper_tangent, lower_tangent)

    # COMBINE THE TWO HULLS WITH UPPER AND LOWER TANGENT -------------------------------------------
    # WORST CASE O(n)
    final_hull = []
    k = lower_tangent[0]
    final_hull.append(left_hull[k])

    while k != upper_tangent[0]:
        k = (k + 1) % len(left_hull)
        final_hull.append(left_hull[k])
    k = upper_tangent[1]
    final_hull.append(right_hull[k])

    while k != lower_tangent[1]:
        k = (k + 1) % len(right_hull)
        final_hull.append(right_hull[k])
    return final_hull
    # BUMP UP A LEVEL IN RECURSION HERE
```