```python
def solve_hull(self, points):  # n log n time and space complexity

    if len(points) > 3:
        m = math.floor(len(points) / 2)
        leftHalf = points[:m]
        rightHalf = points[m:]
        leftHull = self.solve_hull(leftHalf)
        rightHull = self.solve_hull(rightHalf)
        upperTan = find_upper_tangent(leftHull, rightHull)  # n time
        lowerTan = find_lower_tangent(leftHull, rightHull)  # n time
        hull = merge(leftHull, rightHull, upperTan, lowerTan)  # n time
        return hull
    else:
        hull = make_hull(points)
        return hull
```

Divide and Conquer Solve Hull Function:

Time Complexity:

- O(n) from Finding Upper Tangent (see below)
- O(n) from Finding Lower Tangent (see below)
- O(n) from the Merge Function (see below)
- O(log n) from the recursive call that cuts the hull in half each time
- Overall we get O((3n) * log n) => **O(nlog(n))**

Space Complexity:

- O(n) from the array that gets made for the hull in the merge function
- O(n) from the starting array
- O(2n) => **O(n)**

Theoretical Analysis for the Entire Algorithm:

The worst case time complexity for this divide and conquer algorithm is O(nlog(n)). The recursion is related to this time complexity because at each level of recursion you'll split the points in half until you cannot anymore. By splitting into two at each level of recursion, you end up having log(n) levels to deal with. At each level you have a complexity of O(n), which you do O(log(n)) times, which overall makes the entire algorithm O(nlog(n)).

```python
def find_upper_tangent(leftHull, rightHull):
    rightMostIndex = find_right_most(leftHull)
    leftMostIndex = find_left_most(rightHull)
    currentIndexLeft = rightMostIndex
    currentIndexRight = leftMostIndex
    previousLeftHullIndex = rightMostIndex
    previousRightHullIndex = leftMostIndex
    isTopForLeft = False
    isTopForRight = False
    currentSlope = find_slope(leftHull[rightMostIndex], rightHull[leftMostIndex])
    while not isTopForLeft or not isTopForRight:  # n time
        while not isTopForLeft:
            if currentIndexLeft == 0:
                currentIndexLeft = len(leftHull) - 1
            else:
                currentIndexLeft = currentIndexLeft - 1

            tempSlope = find_slope(leftHull[currentIndexLeft], rightHull[currentIndexRight])
            if tempSlope < currentSlope:
                currentSlope = tempSlope
                previousLeftHullIndex = currentIndexLeft
                isTopForLeft = False
                isTopForRight = False
            else:
```

```python
                currentIndexLeft = previousLeftHullIndex
                isTopForLeft = True

        while not isTopForRight:
            if currentIndexRight == len(rightHull) - 1:
                currentIndexRight = 0
            else:
                currentIndexRight = currentIndexRight + 1

            tempSlope = find_slope(leftHull[currentIndexLeft], rightHull[currentIndexRight])

            if tempSlope > currentSlope:
                previousRightHullIndex = currentIndexRight
                currentSlope = tempSlope
                isTopForLeft = False
                isTopForRight = False
            else:
                currentIndexRight = previousRightHullIndex
                isTopForRight = True

    return [leftHull[currentIndexLeft], rightHull[currentIndexRight]]
```

Find Upper Tangent Function:

Time Complexity:

- n is the number of points in the left hull
- m is the number of points in the right hull
- Say n is bigger than m, than worse case complexity is O(2n) =>O(n)
- O(n) from find_right_most(lefthull)
- O(n) from find_left_most(righthull)
- O(n)+O(n)+O(n) => **O(n)**

Space Complexity:

- Nothing is stacked onto each other in memory as you go from iteration to iteration
- **O(1)**

```python
def find_lower_tangent(leftHull, rightHull):
    rightMostIndex = find_right_most(leftHull)
    leftMostIndex = find_left_most(rightHull)
    currentIndexLeft = rightMostIndex
    currentIndexRight = leftMostIndex
    previousLeftHullIndex = rightMostIndex
    previousRightHullIndex = leftMostIndex
    isBotForLeft = False
    isBotForRight = False
    currentSlope = find_slope(leftHull[rightMostIndex], rightHull[leftMostIndex])
    while not isBotForLeft or not isBotForRight:
        while not isBotForLeft:
            if currentIndexLeft == len(leftHull) - 1:
                currentIndexLeft = 0
            else:
                currentIndexLeft = currentIndexLeft + 1

            tempSlope = find_slope(leftHull[currentIndexLeft], rightHull[currentIndexRight])
            if tempSlope > currentSlope:
                currentSlope = tempSlope
                previousLeftHullIndex = currentIndexLeft
                isBotForLeft = False
                isBotForRight = False
            else:
                currentIndexLeft = previousLeftHullIndex
                isBotForLeft = True
```

```python
        while not isBotForRight:
            if currentIndexRight == 0:
                currentIndexRight = len(rightHull) - 1
            else:
                currentIndexRight = currentIndexRight - 1

            tempSlope = find_slope(leftHull[currentIndexLeft], rightHull[currentIndexRight])

            if tempSlope < currentSlope:
                previousRightHullIndex = currentIndexRight
                currentSlope = tempSlope
                isBotForRight = False
                isBotForLeft = False
            else:
                currentIndexRight = previousRightHullIndex
                isBotForRight = True

    return [leftHull[currentIndexLeft], rightHull[currentIndexRight]]
```

Finding Lower Tangent Function:

Time Complexity:

- n is the number of points in the left hull
- m is the number of points in the right hull
- Say n is bigger than m, than worse case complexity is O(2n) =>O(n)
- O(n) from find_right_most(lefthull)
- O(n) from find_left_most(righthull)
- O(n)+O(n)+O(n) => **O(n)**

Space Complexity:

- Nothing is stacked onto each other in memory as you go from iteration to iteration
- **O(1)**

```python
def merge(leftHull, rightHull, upperTan, lowerTan):
    mergedHull = []
    foundTopRight = False
    TopRightIndex = 0
    while not foundTopRight:
        if rightHull[TopRightIndex] == upperTan[1]:
            foundTopRight = True
        else:
            TopRightIndex = TopRightIndex + 1

    foundBottomRight = False
    BotRightIndex = 0
    while not foundBottomRight:
        if rightHull[BotRightIndex] == lowerTan[1]:
            foundBottomRight = True
        else:
            BotRightIndex = BotRightIndex + 1
```

```python
    foundBottomLeft = False
    BotLeftIndex = 0
    while not foundBottomLeft:
        if leftHull[BotLeftIndex] == lowerTan[0]:
            foundBottomLeft = True
        else:
            BotLeftIndex = BotLeftIndex + 1

    while TopRightIndex != BotRightIndex:
        mergedHull.append(rightHull[TopRightIndex])
        if TopRightIndex == len(rightHull) - 1:
            TopRightIndex = 0
        else:
            TopRightIndex = TopRightIndex + 1

    mergedHull.append(rightHull[BotRightIndex])

    while BotLeftIndex != TopLeftIndex:
        mergedHull.append(leftHull[BotLeftIndex])
        if BotLeftIndex == len(leftHull) - 1:
            BotLeftIndex = 0
        else:
            BotLeftIndex = BotLeftIndex + 1

    mergedHull.append(leftHull[TopLeftIndex])
    return mergedHull
```

Merge Function:

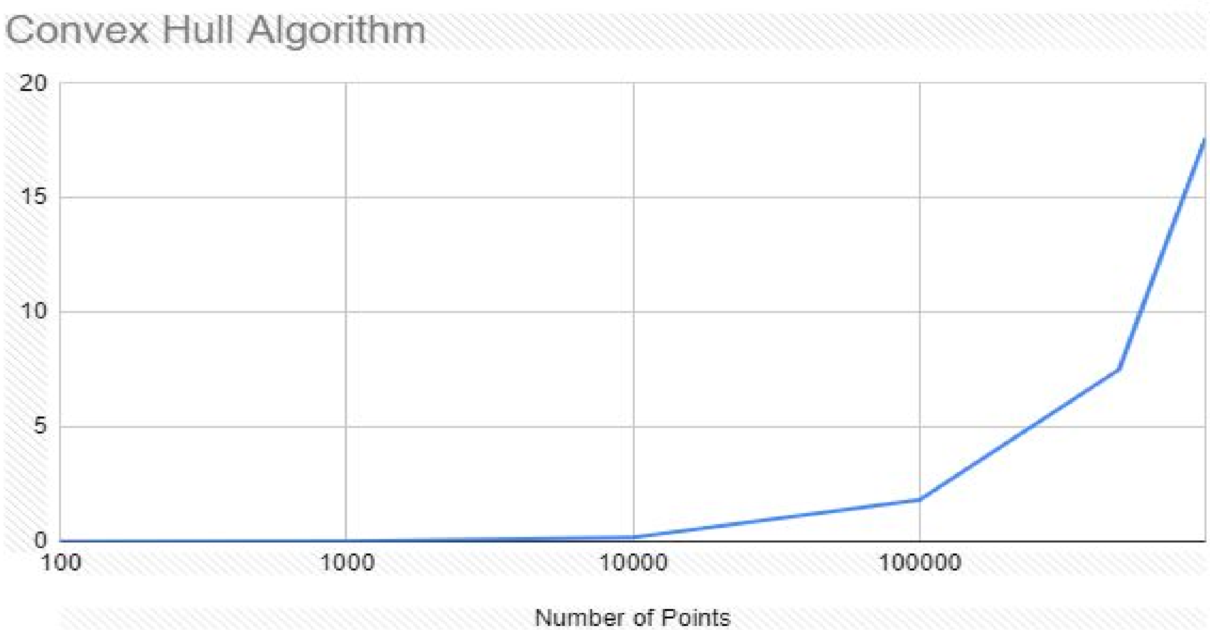
Time Complexity:

- n is the number of points in the left and right hull
- O(n) is worst case as we travel around each while loop in case we have to hit every point before the while loop stops
- O(5n) from all the while loops => **O(n)**

Space Complexity:

- n is the number of points in the hulls
- **O(n)** is the worst case because we are storing an array of all the points

	10	100	1000	10000	100000	500000	1000000
1	0.0001	0.001	0.018	0.204	1.667	7.642	15.051
2	0.0001	0.002	0.026	0.211	1.702	8.501	13.759
3	0	0.001	0.015	0.151	1.48	8.107	25.766
4	0	0.001	0.014	0.122	2.085	6.425	19.561
5	0	0.001	0.014	0.196	2.12	6.906	13.936

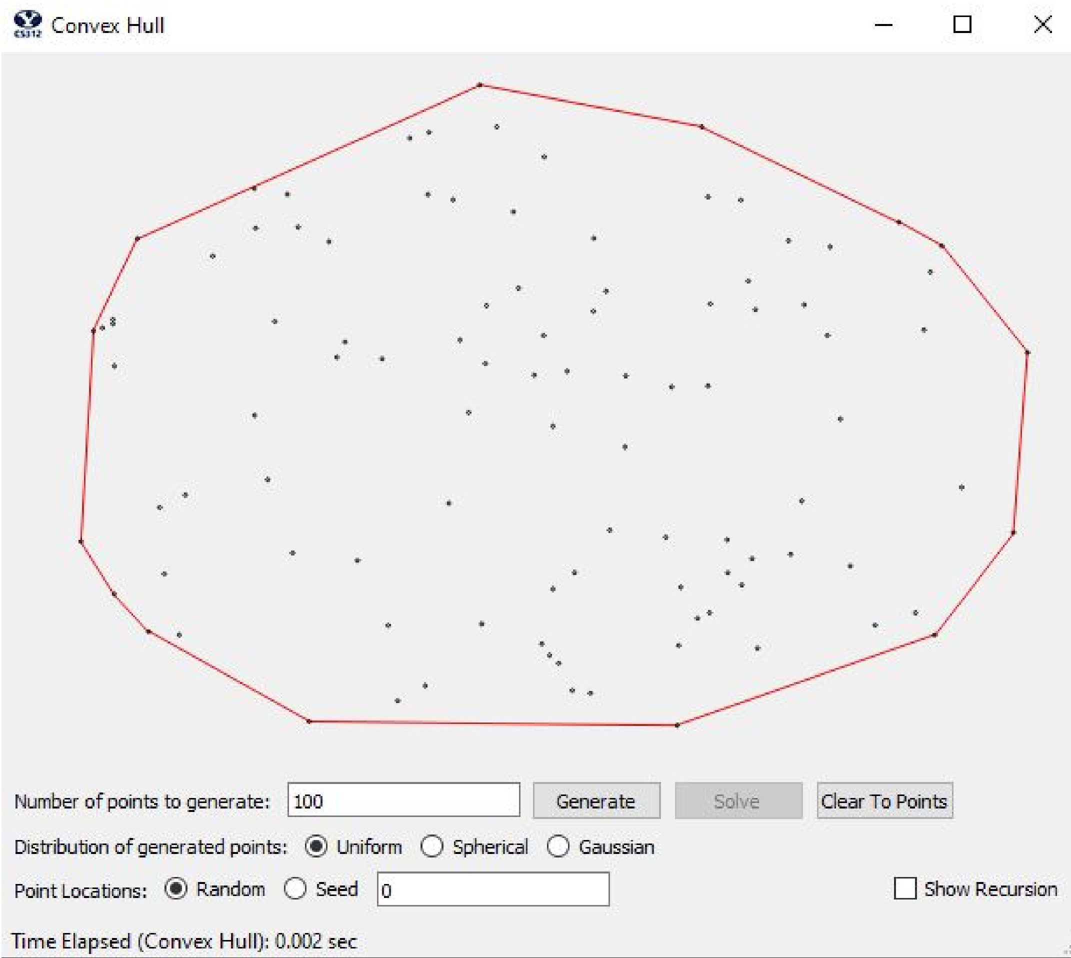


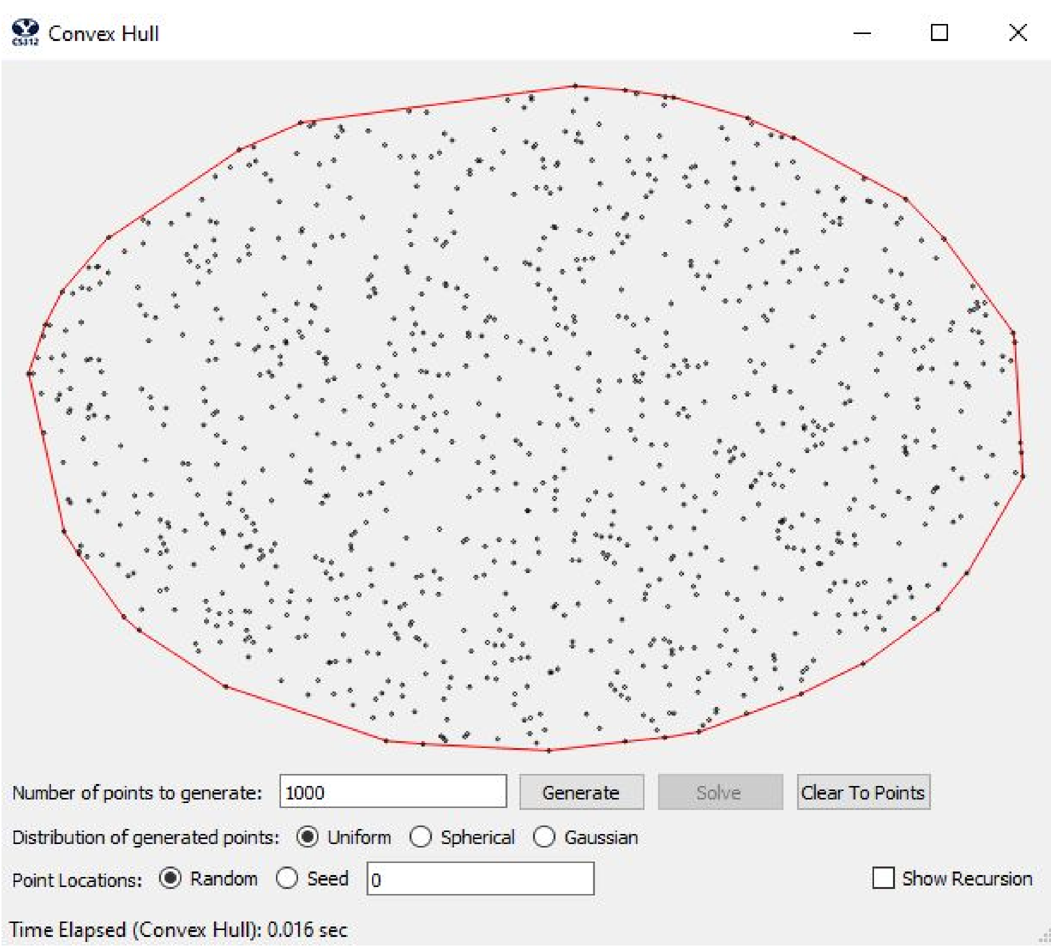
Discussion on the Pattern of the Plot:

I believe that the order of growth that fits best is something more linear than the expression O(nlog(n)). My graph does not relate very well proportionately to the graph of nlog(n). In fact, as the number of points increases, the proportion between what theoretically is happening with the nlogn time and the actual time increases as well.

For example, the ratio between nlogn and the empirical time at 100 points was around 500,000. But at 1,000,000 points the ratio was over a million. This means that the empirical results performed much better than the theoretical results (which makes sense, because nlogn is worst case scenario. If I had to make an estimate of what the proportion constant would be, it about 1/800,000.

The biggest difference between what is empirically happening and what we theorized would happen is that the order of growth is a lot smaller in our algorithm. This is because we theorized the Big O to be what our worst-case scenarios would be. This is a safe claim, but is also unlikely because it would be very rare to have the worst-case scenarios happen, especially with a larger number of points (which is also why the constant of proportionality gets bigger as the amount of points gets larger).


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): 0.002 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): 0.016 sec