Working Code:

divideAndConquer() -

```
#   divideAndConquer()
#   Time: O(nlog(n)), because a = 2, b = 2, and d = 1, which leaves us with O(nlog(n)) according to the master's theorm
#   Space: O(nlog(n)), gets called log(n) times in recursion, each time storing a list with size n, so ends up being a total of O(nlog(n))
def divideAndConquer(sortedPoints):
    if len(sortedPoints) < 3:        # This will mean that when there are only 2 or less points will we start combining
        return sortedPoints
    else:
        leftSide = divideAndConquer(sortedPoints[:len(sortedPoints)//2])
        rightSide = divideAndConquer(sortedPoints[len(sortedPoints)//2:])
        return merge(leftSide, rightSide)
```

merge() -

```
#   merge()
#   Time: O(n), many of the function calls and other parts of this function are all O(n) time
#   Space: O(n), the end of this function creates a new array of at most n elements
def merge(leftSide, rightSide):
    leftStart = getClosestRightPoint(leftSide)
    rightStart = getClosestLeftPoint(rightSide)

    leftTopTangent, rightTopTangent = getUpperTangent(leftStart, rightStart, leftSide, rightSide)
    leftBottomTangent, rightBottomTangent = getLowerTangent(leftStart, rightStart, leftSide, rightSide)

    endList = []                      # Because there are a few different steps of O(n), the total complexity is still O(n) at worst case
    i = rightTopTangent

    # Starts at top right tangent, and loops around shape clockwise, checking to see if tangents on either side are the same
    if rightTopTangent != rightBottomTangent:          # Worst case loops through each point, O(n)
         while i != rightBottomTangent:
            endList.append(rightSide[i])
            i = (i + 1) % len(rightSide)
         endList.append(rightSide[rightBottomTangent])
    else:
         endList.append(rightSide[rightTopTangent])

    if leftTopTangent != leftBottomTangent:            # Worst case loops through each point, O(n)
        i = leftBottomTangent
        while i != leftTopTangent:
            endList.append(leftSide[i])
            i = (i + 1) % len(leftSide)
        endList.append(leftSide[leftTopTangent])
    else:
        endList.append(leftSide[leftBottomTangent])
    return endList
```


```
#   divideAndConquer()
#   Time: O(nlog(n)), because a = 2, b = 2, and d = 1, which leaves us with O(nlog(n)) according to the master's theorm
#   Space: O(nlog(n)), gets called log(n) times in recursion, each time storing a list with size n, so ends up being a total of O(nlog(n))
def divideAndConquer(sortedPoints):
    if len(sortedPoints) < 3:        # This will mean that when there are only 2 or less points will we start combining
        return sortedPoints
    else:
        leftSide = divideAndConquer(sortedPoints[:len(sortedPoints)//2])
        rightSide = divideAndConquer(sortedPoints[len(sortedPoints)//2:])
        return merge(leftSide, rightSide)
```

getUpperTangent()-

```python
#   getUpperTangent()
#   Time: O(n), worst case scenario we loop through each node in the two lists
#   Space: O(1), it is constant because it simply returns two values
def getUpperTangent(leftStart, rightStart, leftList, rightList):
    currentSlope = getSlope(leftList[leftStart], rightList[rightStart])
    rightLength = len(rightList)
    leftLength = len(leftList)
    slopeIncreasing = True
    slopeDecreasing = True
    currentRightTangent = rightStart
    currentLeftTangent = leftStart
    leftPoint = leftStart
    rightPoint = rightStart

    while True:                         # This only loops when a change is found, still O(n) time;
        slopeIncreasing = True          # Two steps both with O(n), which comes to O(n) total time
        slopeDecreasing = True

        while slopeIncreasing:                      # Starts at closest points, follows left point up first
            nextPointSlope = getSlope(leftList[leftPoint], rightList[(rightPoint + 1) % rightLength])

            if nextPointSlope > currentSlope:        # At worst case loops through each point, but is still O(n)
                currentSlope = nextPointSlope
                rightPoint = (rightPoint + 1) % rightLength
                slopeIncreasing = True
            else:
                slopeIncreasing = False

        while slopeDecreasing:                  # Now starts at left point and rotates it up until at the top
            nextPointSlope = getSlope(leftList[(leftPoint - 1) % leftLength], rightList[rightPoint])

            if nextPointSlope < currentSlope:        # At worst case loops through each point, but is still O(n)
                currentSlope = nextPointSlope;
                leftPoint = (leftPoint - 1) % leftLength
                slopeDecreasing = True
            else:
                slopeDecreasing = False

        if (rightPoint == currentRightTangent) and (leftPoint == currentLeftTangent):
            return leftPoint, rightPoint
        else:
            currentRightTangent = rightPoint
            currentLeftTangent = leftPoint
```

getLowerTangent() -

```python
#   getLowerTangent()
#   Time: O(n), worst case scenario we loop through each node in the two lists
#   Space: O(1), it is constant because it simply returns two values
def getLowerTangent(leftStart, rightStart, leftList, rightList):
    currentSlope = getSlope(leftList[leftStart], rightList[rightStart])
    rightLength = len(rightList)
    leftLength = len(leftList)
    slopeIncreasing = True
    slopeDecreasing = True
    currentRightTangent = rightStart
    currentLeftTangent = leftStart
    leftPoint = leftStart
    rightPoint = rightStart

    while True:                             # This only loops when a change is found, still O(n) time;
        slopeIncreasing = True              # Two steps both with O(n), which comes to O(n) total time
        slopeDecreasing = True

        while slopeDecreasing:          # Starts at closest points, follows right point down first
            nextPointSlope = getSlope(leftList[leftPoint], rightList[(rightPoint - 1) % rightLength])

            if nextPointSlope < currentSlope:             # At worst case loops through each point, but is still O(n)
                currentSlope = nextPointSlope
                rightPoint = (rightPoint - 1) % rightLength
                slopeDecreasing = True

            else:
                slopeDecreasing = False

        while slopeIncreasing:          # Takes left point and shifts it down clockwise until flatest.
            nextPointSlope = getSlope(leftList[(leftPoint + 1) % leftLength], rightList[rightPoint])

            if nextPointSlope > currentSlope:             # At worst case loops through each point, but is still O(n)
                currentSlope = nextPointSlope;
                leftPoint = (leftPoint + 1) % leftLength
                slopeIncreasing = True

            else:
                slopeIncreasing = False

        if (rightPoint == currentRightTangent) and (leftPoint == currentLeftTangent):
            return leftPoint, rightPoint
        else:
            currentRightTangent = rightPoint
            currentLeftTangent = leftPoint
```

Other Helper functions –

```
#   getSlope()
#   Time: O(1)
#   Space: O(1)
def getSlope(A, B):
    return (A.y() - B.y())/(A.x() - B.x())  #returns the slope between A and B

#   getClosestRightPoint()
#   Time: O(n), loops through each element in points
#   Space: O(1), No recursion, stored variables are small in comparison
def getClosestRightPoint(points):
    index = 0;
    xPoint = points[0].x();
    for i in range(len(points)):        #Loops through each element to find leftMostX
        if (points[i].x() > xPoint):
            xPoint = points[i].x()
            index = i
    return index

#   getClosestLeftPoint()
#   Time: O(n), loops through each element in points
#   Space: O(1), No recursion, stored variables are small in comparison
def getClosestLeftPoint(points):
    index = 0;
    xPoint = points[0].x();
    for i in range(len(points)):         #Loops through each element to find leftMostX
        if (points[i].x() < xPoint):
            xPoint = points[i].x()
            index = i
    return index
```

Time and Space complexity:

The whole program runs in O(nlog(n)) time, with space complexity of O(nlog(n)).

Sorting the initial list runs in O(nlog(n)) time and O(n) space using the default python sorter.

The only space complexity worth noting is that in the divideAndConquer() method, as other than this function the space complexity is O(1). As mentioned in the code comments, this function will recurse to log(n) levels, with each level simply allocating half of the previous array, with a total space of n for each level. This leaves us with a total space complexity of O(nlog(n)), which is the cap of the program.

The merge() function runs in O(n) time, with a space complexity of O(1). This function runs in O(n) because although it traverses through each array maybe multiple times, each O(n) function is a different step, so they are never compounded.

Finding both upper and lower tangents end with a time complexity of O(n) as at worst case they traverse through each point in either list. Actually merging the two lists together also is only O(n) because at worst case it iterates through each point of both lists, but because both sections are O(n) and do not contain inner loops, our time complexity stays put with a list of O(n) time steps, simplifying to a total time complexity of O(n).

Other minor functions (such as getting the right most point) have a time complexity of O(n) because they iterate through each point. This time does not add to the time complexity because, as previously explained, it is simply one of the O(n) steps needed during merging, and is not compounded with previous steps.

Theoretical Analysis:

Knowing that merge() completes in O(n) time, we can now use the master theorem to calculate the total big-O time. Because each time we call divideAndConquer(), we split the problem into two sets both of n/2. This gives us variables a = 2, b = 2, and d = 1 (from merge() function complexity), which equals T(n) = 2T(n/2) + n. Because the result of a/b^d is < 1, according to the master theorem, we can conclude that the total time complexity of the convex_hull program is O(nlog(n)).

Raw Data:

n = 10
0.000
0.000
0.000
0.000
0.000
Average = less than 1 milisecond (.0003)

n = 100
0.002
0.003
0.003
0.003
0.003
Average: 0.0028 or just 0.003

n = 1000
0.019
0.018
0.017
0.018
0.018
Average: 0.018

n = 10000
0.123
0.122
0.122
0.134
0.123
Average: 0.1248 or 0.125

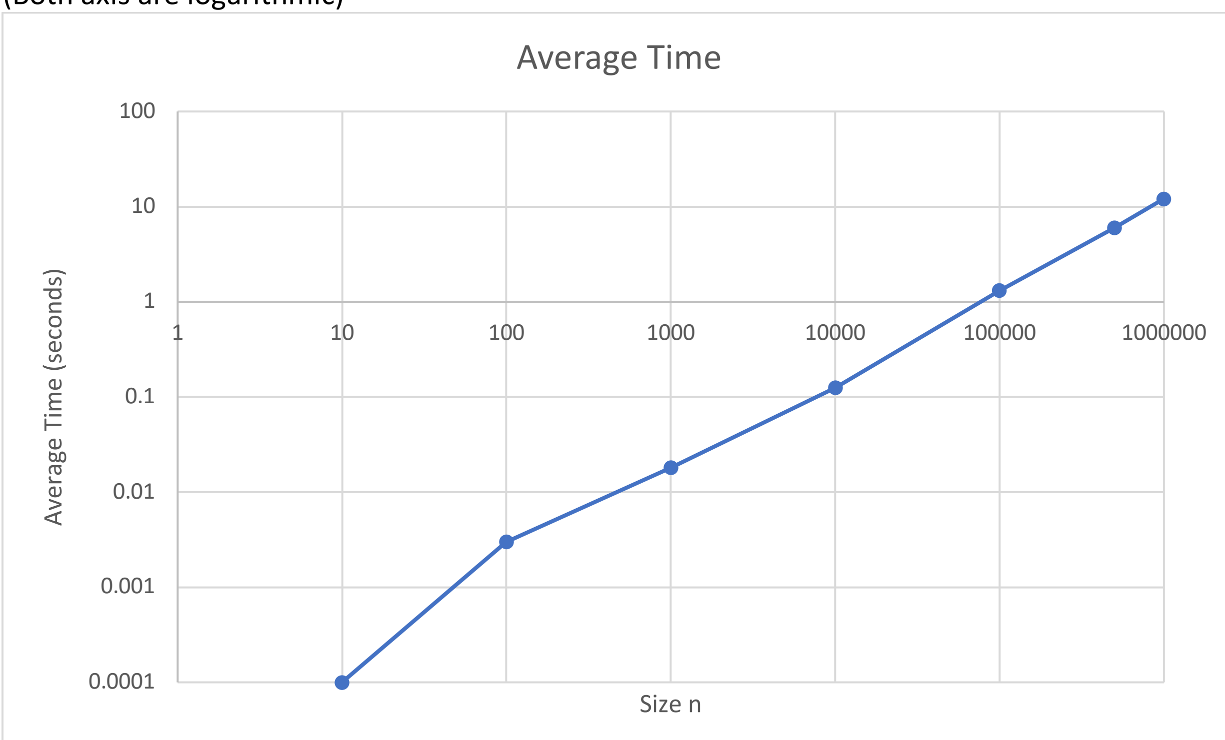
n = 100000
1.282
1.294
1.295
1.389
1.283
Average: 1.3086

n = 500000
6.186
5.929
5.971
5.845
6.037
Average: 5.9936

n = 1000000
12.434
12.003
12.144
12.132
11.816
Average: 12.105

Plot:

(Both axis are logarithmic)



As shown in my plot (being logarithmic axis) we see that my algorithm is roughly linear, which shows that it is roughly nlog(n) on a linear graph. The higher the values for n, the more linear the points fit on the graph, and the greater it resembles the line of nlog(n).

Constant Proportionality:

Time(n)	O(nlog(n))	k (O(nlogn)/time)
0.0003	33.219	110730
0.0028	664.386	237280.71428
0.018	9965.784	553654.666
0.1248	132877.124	1064720.544
1.3086	1660964.047	1269267.955
5.9936	9465784.285	1579315.269
12.1058	19931568.569	1646447.865

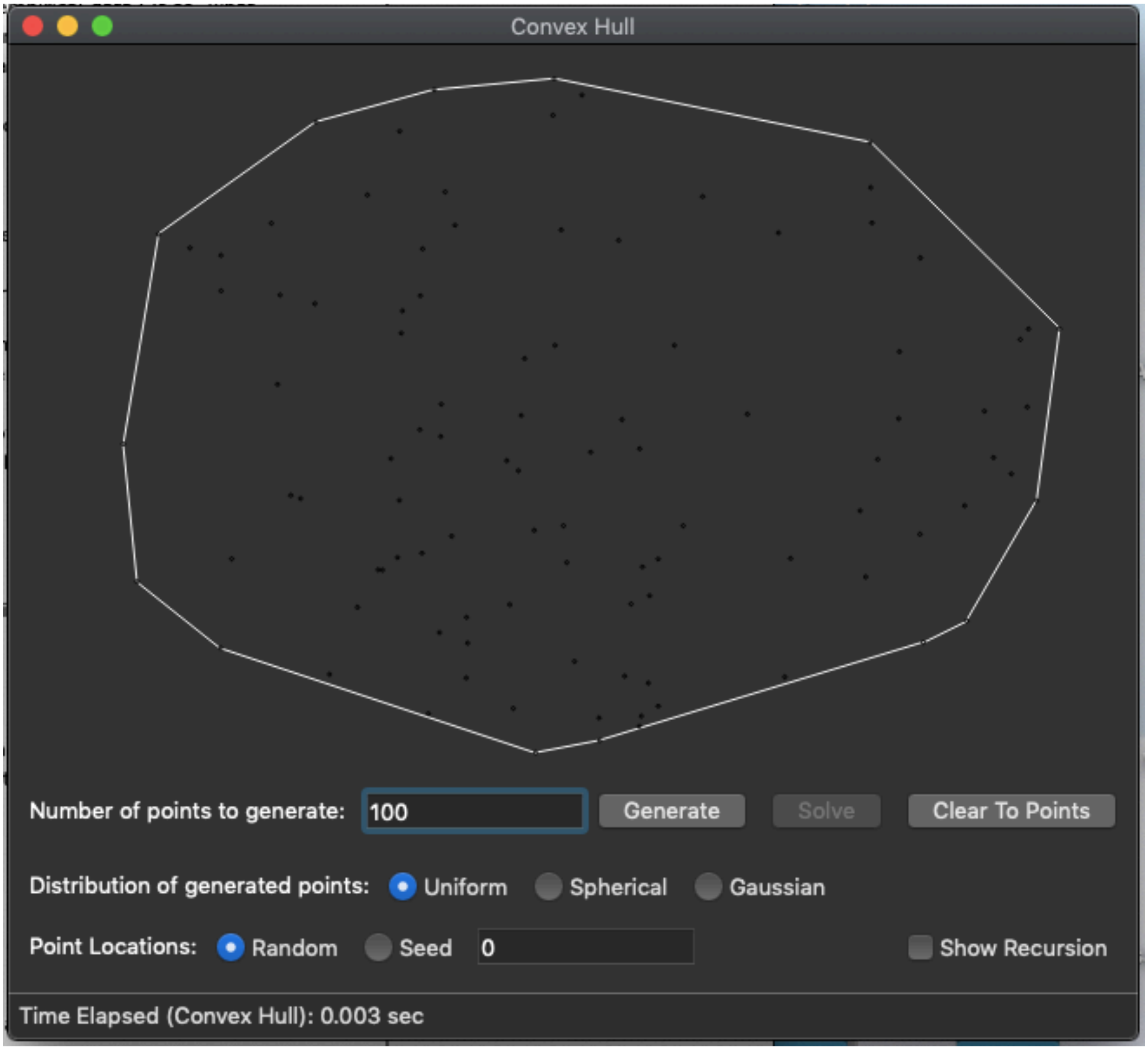
K averages out to be about 9.23 x 10^5, which means that CH(n) = 1.79 x 10^5 g(n)

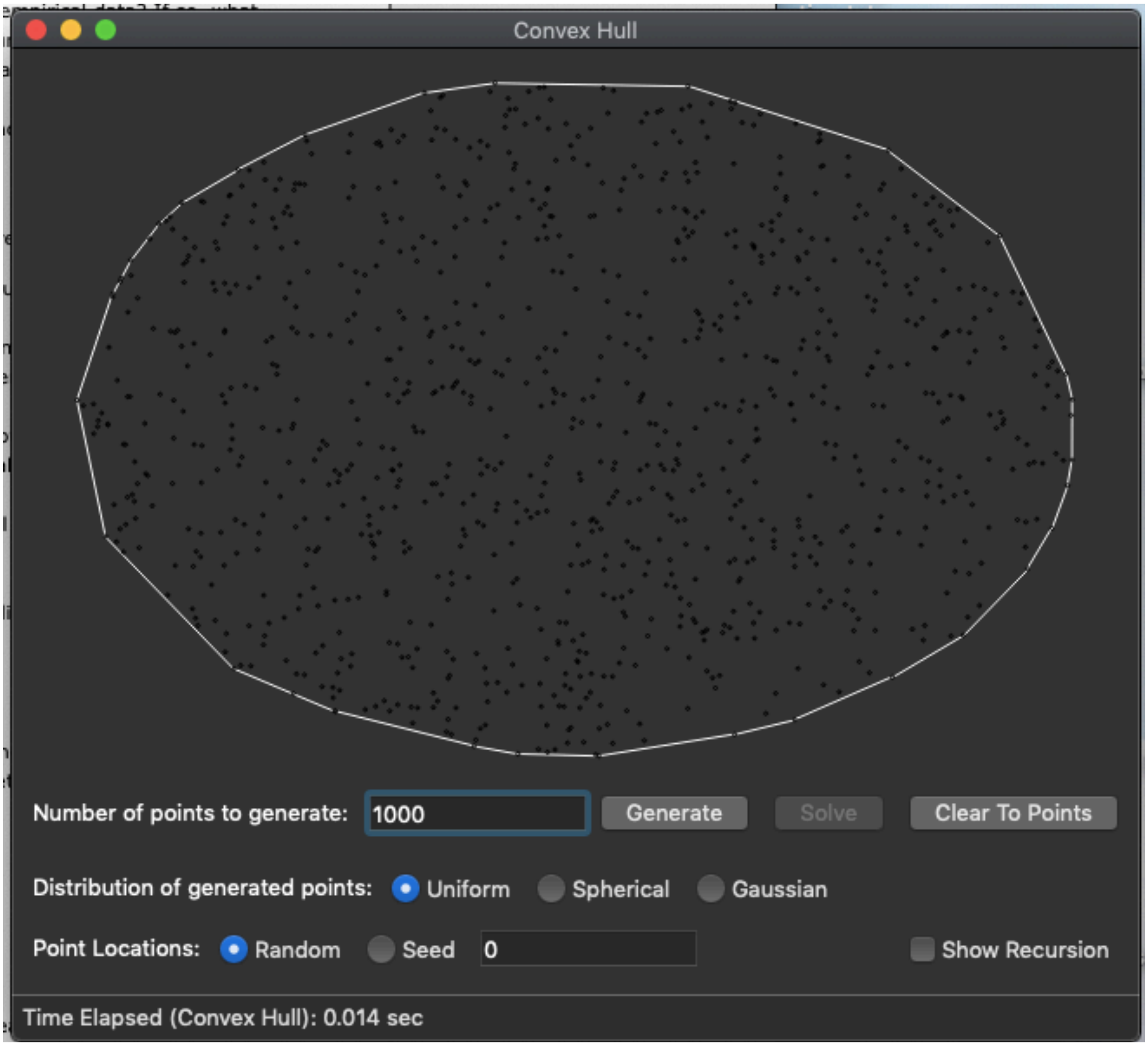
Observations:

According to the plotted points, my theoretical analysis is correct. As we can see with the graph we get a roughly linear time on a logarithmic graph, which plots to a nlog(n) time on a linear graph. Seeing the data plotted out shows that my algorithm runs in nlog(n) time. This shows that g(n) does fit my empirical data, and even my constant k is a fairly good estimate for my graph. My k does not plot exactly the same as the actual data, but that could be due to other factors with smaller n values and the computer which the algorithm runs on.

One thing I observed which I briefly mentioned before is how the greater the values of n, the more similar the graph is to the graph of nlog(n). It seems my algorithm approaches that slope with bigger n values, and with lower n values it seems to have different behaviors, and does not truly emulate the theoretical equation of nlog(n).

Example Screenshots:


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): 0.003 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): 0.014 sec