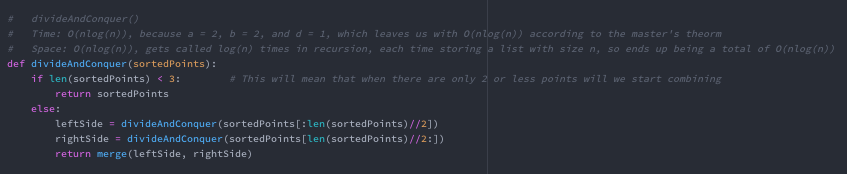
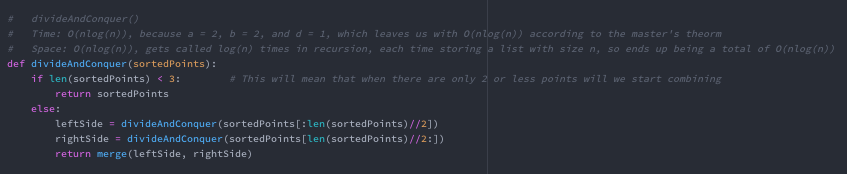
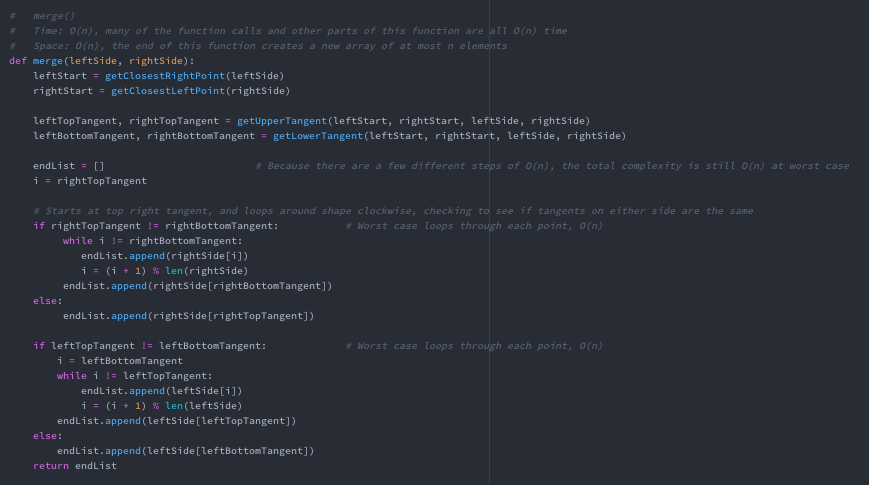
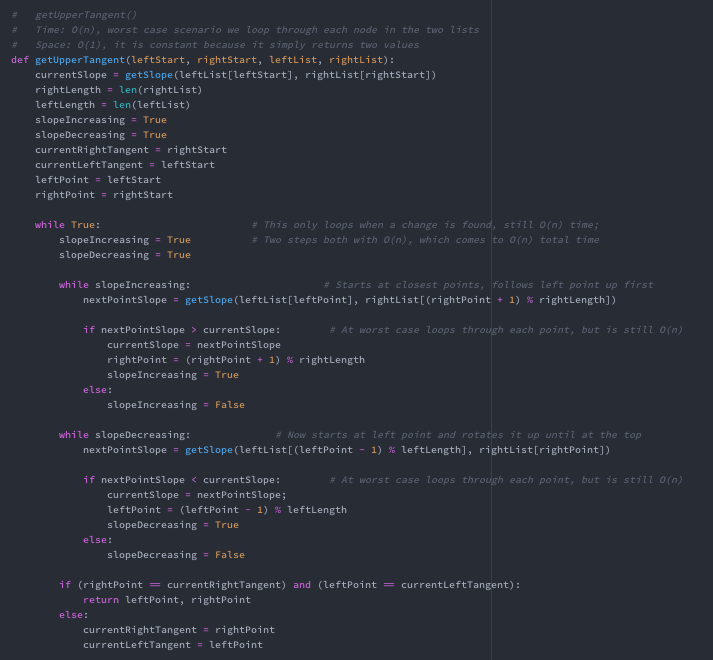
Working Code:

divideAndConquer() -

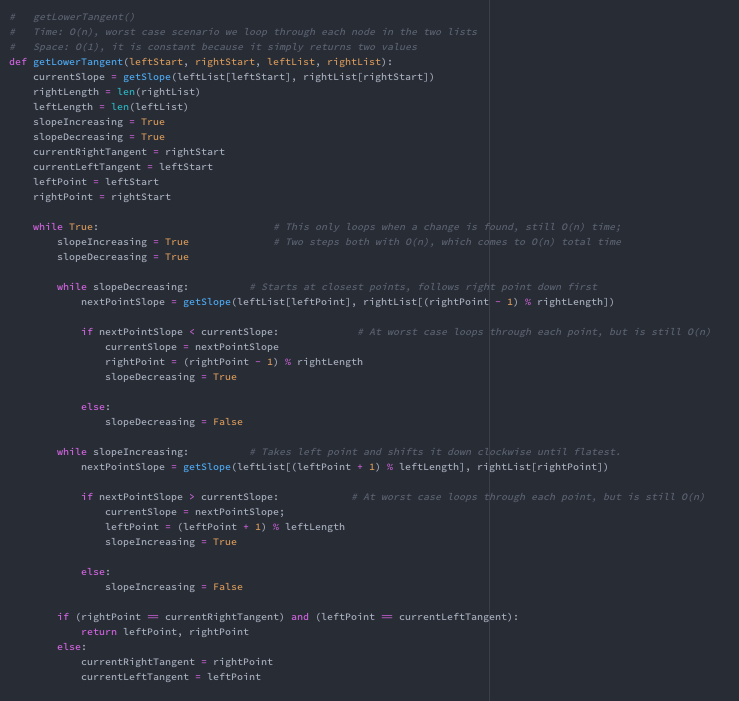


merge() -



getUpperTangent()- 

getLowerTangent() -



Other Helper functions –



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:

