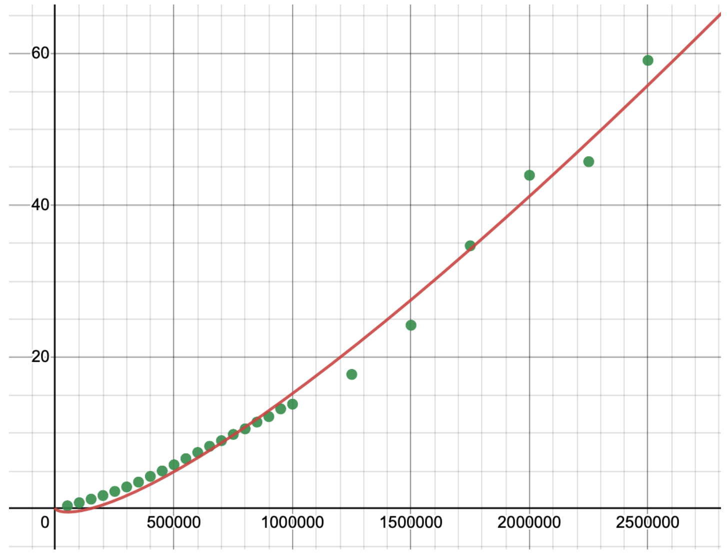
Tommy Buston
Eric Shaw

CSC 440 A2: Convex Hull

compute_hull() - Execution Time (s) vs. Input Size (number of coordinate points)

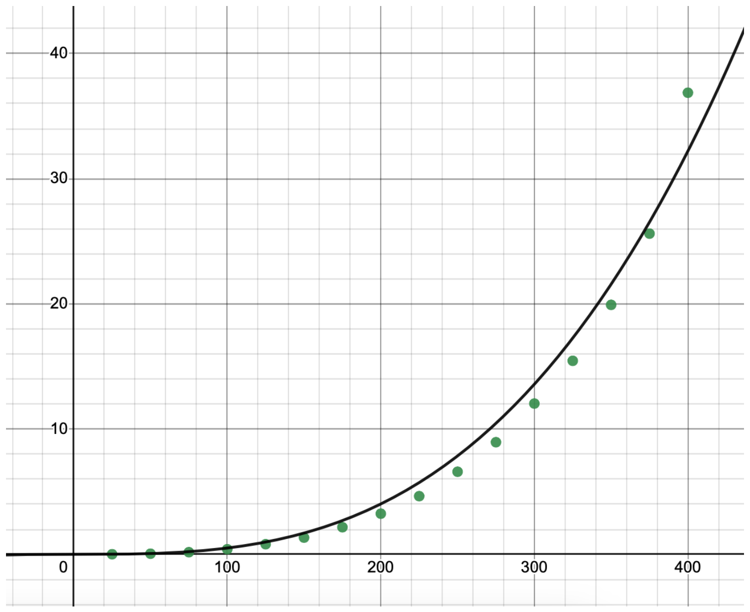


O(n*log(n)), R^2=0.989
https://www.desmos.com/calculator/aamykekffj

This graph clearly resembles the O(n log n) time complexity for the convex hull algorithm. O(n log n) time complexity refers to an algorithmic time complexity that grows logarithmically with respect to the size of the input n, multiplied by the size of the input n. The input data is repeatedly divided into smaller subproblems until the problem becomes small enough to be solved directly. This is where our base case or naive algorithm comes into place. The program keeps dividing into smaller subproblems until there is less than six points to work with, that is when our base case hits and our native algorithm performs its task. Once the convex hulls of each subset are computed, they are merged together to form the convex hull of the original set of points. This process results in a logarithmic factor, as the input data is divided into halves or smaller portions with each recursive step. As you can see, the curve representing the algorithm's running time grows slowly as n increases, indicating that the algorithm scales well with increasing input size. We can say quite confidently that the n*log(n) regression is a good fit for our data because the R^2 value of 0.989 is *very* close to 1. It is also likely that the later test cases suffered from some CPU throttling due to the temperature of the macbook air increasing, so we could have achieved an even better fit if this was avoided. With this in mind we conclude that the theoretical estimation of n*log(n) is accurate in practice and conversely that our implementation is an accurate representation of the divide and conquer Convex Hull algorithm.

Tommy Buston
Eric Shaw

base_case_hull() - Execution Time (s) vs. Input Size (number of coordinate points)



O(n^3), R^2=0.9789
https://www.desmos.com/calculator/yxdnkzbn7k

This graph shows the O(n^3) time complexity for our base_case_hull. O(n^3) time complexity refers to the algorithm's running time growing proportional to the cube of the size of the input. This function checks every point in the input data (n), every pair of points (n), and the helper functions check_line_UNdefined and check_line_defined check that the line formed by the pair of points is greater than all other points or less than all other points (n) making the complexity time O(n^3). Instead of using the divide and conquer technique it checks everypoint to see if there is a "good line". If the line is good the point from the outermost loop is added to the hull. R^2 is not as close to one 1 as our compute_hull, but having a value of 0.9789 is still very close to 1. The R^2 could have suffered once again due to the overheating of the macbook. A better R^2 could have been achieved if we could have avoided that problem. With that, we can conclude that our implementation of base_case_hull() has a time complexity of O(n^3) which is nearly the same as the theoretical O(n^3) that was calculated.

Tommy Buston
Eric Shaw

Invariant 1:

- The compute_hull() and base_case_hull() function returns only the points that are on the convex hull.

Initialization:

- The problem begins by sorting the given points and recursively dividing the points in half by their x-values until the problem becomes small enough that the brute force algorithm is efficient. When this happens hulls of six or less points are calculated by checking each point for a possible line that has all other points to one side of it.

Maintenance:

- As the base case is returned up the call stack, the previous level now has two hulls divided by their x-values. These hulls are then merged by finding the upper and lower tangent and popping points in between off of the hull. This new hull is then returned up the call stack to fill in its place as either the left or right hull of the function that called it.

Termination:

- When the first function call receives its left and right hull from all of the recursive calls that came after it, these hulls are merged and the newly calculated hull is returned and this is the hull for the entire data set.

Invariant 2:

- The points on the hull are always clockwise sorted.

Initialization:

- Once the points are sorted and the problem is divided in half enough times to reach the base case (as detailed above) the hull is calculated with the naive algorithm. These hull points are then sorted clockwise by the clock_wise() sort function.

Maintenance:

- The merging of the hulls that are received up the call stack *maintains* the clockwise order of the points by adding the points to the hull list in clockwise order. This is done by starting with the left bottom tangent point and moving clockwise to the left top tangent point. From there points are added starting with the right top tangent point and ending with the right bottom tangent point. This order inherently maintains the clockwise order of the hulls that were provided already sorted from the base case.

Termination:

- When the first function call receives its left and right hull from all of the recursive calls that came after it these hulls are merged and clockwise order is maintained as explained.