Honors Project Report: Finding Convex Hull using Divide and Conquer[*]

Caleb Manicke

Abstract

The purpose of this report is to explore the intuition and analyze the reasoning that went into developing my program which when given a set of points, finds the convex hull. Unlike the more widely used Graham's Scan, my algorithm use of a Divide and Conquer approach which, in general, combines the convex hulls of two subgraphs at each recursive level. This report will also document my program's benchmarking results.

1 Intuition and Pseudocode Outline

First, we must establish a strong ground of working intuition. Imagine a 2D graph of randomly scattered coordinates. Even though there may be no pattern to where these coordinates lie on our plane, there must be some points that are farther out than the other points. Imagine then, if we were to connect these "outlying" points such that all other points were located *inside* this shape. This is our **convex hull**.

Definition 1 (Convex Hull)**.** *Given a set of points, the convex hull is the* ***minimum*** *subset of points that form the boundary of our entire set of points.*

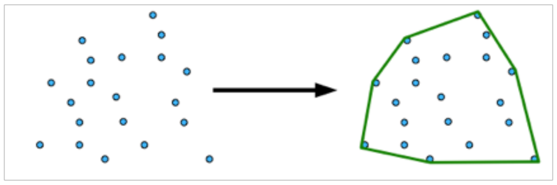


Figure 1: All points not in our convex hull are *contained* inside our convex hull. Image from `https://axon.cs.byu.edu/~dan/312/projects/ConvexHull.php`

[*]Source: `https://www.overleaf.com/project/63d07d57e3c07006b484dc9d`

Although convex hull must contain every other point in a set, there is not necessarily a fixed number of points that can be in our convex hull relative to how many points are in our set. However, there are always trivial cases: consider a set of only one, two, or three points. These sets of points correspond to a single point, a line, and a triangle respectively, and since they don't have any points to contain, they are their own convex hulls.

These trivial cases would act as our base cases in a Divide Conquer algorithm. But how would we divide a graph efficiently to be able to find its convex hull? Suppose we have two subsets of points whose convex hulls we have found. We want to *combine* these hulls to form the convex hull for *both* subsets. If these sets of points were overlapping, or one subset of points is completely encapsulated by the convex hull of our other subset, then we wouldn't be able to make use of both convex hulls to form our larger hull.

In order to be efficient, we need to partition our graph such that no two subgraphs would have an overlapping or encapsulated convex hull. If each convex hull represented some region of our graph, then we would have to use a piece of each region's boundary to form the convex hull for the entire graph. Because we're working on a 2D plane, we can easily partition our x-axis to divide our graph into disjoint subgraphs. Then once we find the find the convex hull for each, we can piece back our x-axis to get our complete convex hull.

With this idea, we can write the pseudocode for our general Divide and Conquer algorithm for finding the convex hull below.

Algorithm 1 Find-Convex-Hull (Graph G):

```
1. Base Case: If our graph has three or less points, it's its own convex hull, so we simply return it
2. Sort each point in our graph by their $x$-coordinates
3. Split our $x$-axis in half, giving us sets of points"left" and "right" for each half
4. left-convex-hull = Find-Convex-Hull(left)
5. right-convex-hull = Find-Convex-Hull(right)
6. Combine left-convex-hull and right-convex-hull, return this
```

Our Divide and Conquer algorithm will work its way down to the base case (subset of three or less points), then call another algorithm to combine our left and right convex hulls. To explain our combine algorithm, we need more intuition: recall that because we sort and split our points based on their x-coordinates, our convex hulls should be disjoint. This will give us two disjoint perimeters, which we want to combine to form a perimeter for both split sets of points.

In abstract, we want to combine two shapes into one. As our subgraphs get larger, we will be working with progressively more "circular" perimeters, so we want a method that can combine any two n-sided polygons into one. When we are dealing with two shapes on a plane, we can form tangent lines to connect them: consider a line that connects one shape from another, but is above all of our points. That is, it only intersects at one of our highest points (in terms of the y-axis) in each subgraph. This is our **upper tangent**. Likewise, a line that intersects one of the lowest points in each set and is “below” every other point would be our **lower tangent**.

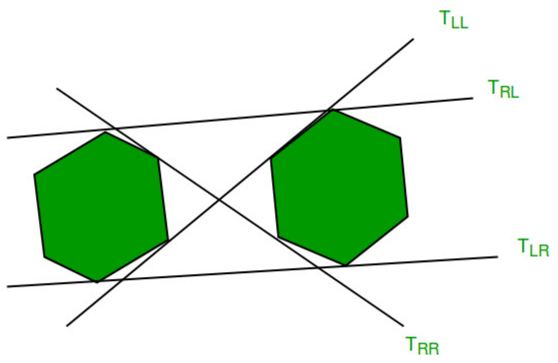


Figure 2: Notice how our shapes are completely between our upper and lower tangents. Our upper and lower tangents should intersect at most one point in both shapes. Image from `https://www.geeksforgeeks.org/tangents-two-convex-polygons/`

If we can find the upper and lower tangents for any two convex hulls, we can combine them into a larger boundary by merging the points where our upper and lower tangents intersect at as well as the points that form the outer perimeter between these intersecting points, disregarding all points “inside” of our new perimeter.

We need to be more precise: how can we more exactly determine which points to keep and which points to throw away in our new convex hull? Our points in each shape are sorted according to their place on the x-axis, but we need a more clear relation in order to find the “outer” boundary connecting the upper and lower tangents in each shape.

Suppose that we found the centroid for a given set of points, a.k.a. the point that would represent the rough middle of our shape, and sorted our points in clockwise order relative to our centroid. Then in order to find the perimeter from our new convex hull, we could start at one of our intersecting points, say on the upper tangent, work our way clockwise towards our intersecting point on the lower tangent, make our way to our other shape and work our way up to the other point on the upper tangent, so our merged Convex Hull will preserve this clockwise order. We outline our merge algorithm below:

Algorithm 2 Merge-Two-Convex-Hulls (Sets P and Q):

```
1. Find the centroid for both $P$ and $Q$
2. Sort each point in $P$ and $Q$ clockwise of their centroids
3. Find the upper and lower tangents of $P$ and $Q$, let $p_1$, $q_1$, $p_2$, $q_2$ denote
their intersections on the upper and lower tangents respectively
4. Merged-Convex-Hull = $[q_1]$
5. From $q_1$, iterate through $Q$ clockwise, adding points to Merged-Convex-
Hull until we reach $q_2$
6. Since $q_2$ is connected to $p_2$ by our lower tangent, add $p_2$, iterate through
$P$ clockwise, adding points to Merged-Convex-Hull until we reach $p_1$
Return Merged-Convex-Hull
```

We have one more critical algorithm to discuss: given two subgraphs, how can we find the upper and lower tangent? Suppose we have two sets of points P and Q, where P is to the left of Q from our x-axis partitioning. We know our points in P and Q are sorted clockwise to their relative centroids, so we will iterate through P and Q to find our intersecting points for each tangent line.

For the upper tangent, we start with the rightmost point in P and the leftmost point in Q, which we will denote as p_1 and q_1 respectively, in order to minimize the distance of our tangent line between our sets. Because our upper tangent will intersect our sets P and Q at a top-most point, we want to move clockwise for Q and counter-clockwise for P. We will first iterate through our points in Q, so we can denote the three points we work with initially as p_1, q_1, and q_2, where q_2 is one step clockwise from q_1.

We will iterate from q_2 to some q_i where the **tangent line** between q_i and p is **above** all of our other points in Q. At each instance, given some p, q_i and its clockwise point q_{i+1}, we want to make sure that the **slope** formed by p and q_{i+1} is greater than the slope between p and q_i. We continue until we reach a point where this is not true: that is, the point one step counterclockwise to q_{i+1} forms a greater slope. This is what we define as the **Orientation of Three Ordered Pairs**.

Definition 2 (Orientation of Three Ordered Pairs)**.** *Suppose we are given three ordered pairs (a, b, c). Consider a directed triangle formed by a, b, c such that a points to b, b to c, and c to a, where a lies farther away horizontally from b and c than b to c do.*

- *We can compute the slope of the tangent line from a to b as $(b[1] - a[1])/(b[0] - a[0])$ and the slope from b to c as $(c[1] - b[1])/(c[0] - b[0])$.*
- *If our slope from b to c is **greater** that our slope from a to b, then $(b[1] - a[1]) \cdot (c[0] - b[0]) > (b[0] - a[0]) \cdot (c[1] - b[1])$, meaning our directed triangle is oriented clockwise from a.*
- *If our slope from b to c is **less** that our slope from a to b, so $(b[1] - a[1]) \cdot$*

$(c[0]-b[0]) < (b[0]-a[0])\cdot(c[1]-b[1])$, our directed triangle is oriented counterclockwise from a.

- *If our slopes are the same, so $(b[1]-a[1])\cdot(c[0]-b[0]) = (b[0]-a[0])\cdot(c[1]-b[1])$, then all of our points form a line rather than a triangle without any clockwise orientation. We call this orientation "co-linear".*

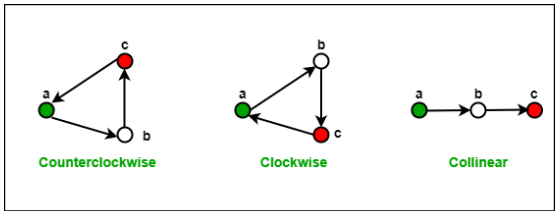


Figure 3: In these graphs, replace a, b, c with p, q_i and q_{i+1} respectively. We will iterate through Q by incrementing i until our triangle is NOT oriented clockwise anymore. Image from `https://www.geeksforgeeks.org/orientation-3-ordered-points/`

Since co-linearity is extremely rare in a randomized set of points, we can disregard it. This is how we find our q_i on our upper tangent: we will iterate through Q, finding the orientation of the triangle formed by p, q_i, and q_{i+1}, taking more clockwise steps while our triangle also has a clockwise orientation.

Once we find a q_i such that its triangle with p and q_{i+1} would form a counter-clockwise oriented triangle, we fix our q and iterate through p_i. Here, the reverse applies: we want to take counter-clockwise steps by considering if the triangle formed by q, p_i and p_{i-1} is oriented counter-clockwise as well. We stop once we find a p_i such that its triangle with q and p_{i-1} is oriented clockwise, so the tangent formed by q and p_{i-1} would be below the tangent formed by q and p_i. We outline our upper-tangent algorithm below:

Algorithm 3 Find-Upper-Tangent (Sets P and Q):

```
1. Find the rightmost point on the $x$-axis in $P$, $p_i$
2. Find the leftmost point on the $x$-axis in $Q$, $q_i$
3. While $(p_i, q_i, q_{i+1})$ form a clockwise oriented triangle, set $q_i = q_{i+1}$ by moving one-step clockwise
4. While $(q_i, p_i, p_{i-1})$ form a counter-clockwise oriented triangle, set $p_i = p_{i-1}$ by moving one-step counterclockwise
5. Return the tangent line formed by $(p_i, q_i)$
```

Our algorithm for finding the lower tangent is identical, with a couple tweaks: first, we find the *leftmost* point in P and the *rightmost* point in Q. We iterate through q_i while (p_i, q_i, q_{i+1}) form a *counter-clockwise* oriented triangle, then through p_i while (q_i, p_i, p_{i-1}) forms a clockwise oriented triangle.

We have fully outlined the intution, design, and function behind the algorithms my program will implement. Therefore, we will analyze my code by diving into implementation tweaks and benchmarking tests.

2 Code Analysis

We will now translate each pseudocode algorithm into a Python function, highlighting the implementational choices made in each line of code. We start with "findOrientation", which returns the orientation a triangle formed by three ordered pairs:

```
# Find orientation of directed triangle formed by (a, b, c) by
    comparing slopes: O(1)
# Note: collinear, cw and ccw helper functions above produce
    runtime errors for certain random inputs
def findOrientation(a, b, c):
    # Compute orientation terms derived from slopes of tangents
    from points a to b and b to c respectively
    a_to_b_orientation = (b[1] - a[1]) * (c[0] - b[0])
    b_to_c_orientation = (b[0] - a[0]) * (c[1] - b[1])
    # If slope from a to b is equal to slope from b to c, it's
    collinear: greater --> triplet has clockwise orientation,
    lesser --> counterclockwise orientation
    if ((a_to_b_orientation - b_to_c_orientation) == 0):  return 0
    elif (a_to_b_orientation > b_to_c_orientation): return 1
    else: return -1
```

Recall our Orientation of Three Ordered Points: given (a, b, c), where each a, b, c is a coordinate on a 2D plane that form some shape where a points to b, b to c, and c to a, we can compute the slopes of our tangent lines formed by a to b and a to c. Rearranging our terms in our slope give us two terms $(b[1] - a[1]) \cdot (c[0] - b[0])$ and $(b[0] - a[0]) \cdot (c[1] - b[1])$ which we denote as *a_to_b_orientation* and *b_to_c_orientation* above.

Once we compute these terms, a simple logic check will return whether our points form a line (collinear), or a triangle oriented clockwise or counterclockwise around a. This function will be our main logic check for whether we should continue iterating through the points in two hulls to find an upper and lower tangent. We give the algorithm for finding our upper and lower tangents below:

```python
def findTangents(left_hull, right_hull):
    # Find the indexes of the rightmost point of the left hull and the leftmost point of the right hull: O(n)
    left_index, right_index = 0, 0
    for i in range(len(left_hull)):
        if left_hull[i][0] >= left_hull[left_index][0]:
            left_index = i
    for i in range(len(right_hull)):
        if right_hull[i][0] <= right_hull[right_index][0]:
            right_index = i

    # Iterate through each point in left_hull and right_hull to find the upper tangent: O(n)
    while True:
        # If triangle (p_i, q_i, q_{i+1}) is oriented clockwise, the tangent from p_i to q_{i+1} will be above tangent connected to q_i: move right_index clockwise
        if findOrientation(left_hull[left_index], right_hull[right_index], right_hull[(right_index + 1) % len(right_hull)]) > 0:
            right_index = (right_index + 1) % len(right_hull)
        # If triangle (q_i, p_i, p_{i-1}) is oriented counterclockwise, the tangent q_i to p_{i-1} is above tangent q_i to p_i: move left_index counterclockwise
        elif findOrientation(right_hull[right_index], left_hull[left_index], left_hull[(left_index + len(left_hull) - 1) % len(left_hull)]) < 0:
            left_index = (left_index + len(left_hull) - 1) % len(left_hull)
        # Once we found maximally clockwise point for left_hull for forming a tangent with any point in right_hull above all points in left_hull, and maximal counterclockwise point in right_hull for same reason, break from loop
        else:
            break
    upper = (left_index, right_index)

    # Find the indexes of the leftmost point of the left hull and the rightmost point of the right hull: O(n)
    left_index, right_index = 0, 0
    for i in range(len(left_hull)):
        if left_hull[i][0] <= left_hull[left_index][0]:
            left_index = i
    for i in range(len(right_hull)):
        if right_hull[i][0] >= right_hull[right_index][0]:
            right_index = i

    # Iterate through left_hull and right_hull to find lower tangent: O(n)
    while True:
        # If triangle (p_i, q_i, q_{i+1}) is oriented counterclockwise, the tangent formed from p_i to q_{i+1} is below tangent from p_i q_i: move right_index clockwise
        if findOrientation(left_hull[left_index], right_hull[right_index], right_hull[(right_index + 1) % len(right_hull)]) < 0:
            right_index = (right_index + 1) % len(right_hull)
        # If triangle (q_i, p_i, p_{i-1}) is oriented clockwise, the tangent q_i to p_{i-1} is below the tangent q_i to p_i: move left_index counterclockwise
        elif findOrientation(right_hull[right_index], left_hull[left_index], left_hull[(left_index + len(left_hull) - 1) % len(left_hull)]) > 0:
            left_index = (left_index + len(left_hull) - 1) % len(left_hull)
        # We found p_i whose tangent with q_i is below all other points in left_hull and q_i whose tangent with p_i is below all other points in right_hull: break from loop
        else:
            break
    lower = (left_index, right_index)

    return upper, lower
```

Our first step when finding the upper tangent is to locate the rightmost point of our left hull and the leftmost point of our right hull. This is done by a simple for-loop, which compares all of the x-coordinates in both hulls and saves our min-max points. Since we sort our hulls clockwise around their centroids, when we run into a tie (both points have the same x-coordinate), our left hull will prioritize the point with the lower y-value whereas right-hull prioritizes a higher y-value (by moving clockwise regardless). This will help us find our intersecting point in our right hull since the slope between left-hull[i], right-hull[j] and right-hull[j+1] will be larger.

To find our upper tangent, we use an infinite loop that will first iterate through each point in our right hull until the next clockwise point forms a counterclockwise triangle, THEN we iterate through each point in our left hull until the next counterclockwise point forms a clockwise triangle. To ensure that we iterate through our right-hull first, we make use of an if...elif...else statement that will prioritize moving our right_index clockwise as long as the next point forms a clockwise-oriented triangle while keeping our left_index fixed. Once we no longer meet this condition, our right_index becomes fixed so we can find a point at left_index that also intersects our upper tangent. Once our indices fail to meet both conditions, we break free from our loop: because our points are bounded by a $(0, n) \times (0, n)$ grid, an upper tangent **must** exist so our loop will always terminate.

Our program for finding the lower tangent follows the exact same format: we first iterate through our left hull to find its leftmost point, then through our right hull to find its rightmost point. From there we loop through each point clockwise of our right hull while our next clockwise point forms a counter-clockwise oriented triangle, and THEN we loop counterclockwise through our left hull while our next point forms a clockwise triangle.

We return the indices of each point in our left and right hulls respectively that intersect our upper and lower tangent. Because we have not made any modifications to the clockwise order of our points in our left and right hull, we can easily work with our upper and lower tangent indices to iterate through our left and right hulls and merge our convex hulls. The code for merging two convex hulls is shown below:

```
    def mergeHulls(left_hull, right_hull):
    # Sort points in left_hull and right_hull in counterclockwise order: O(n log (n))
    clockwiseSort(left_hull)
    clockwiseSort(right_hull)

    # Find the upper tangent and lower tangent of the hulls
    upper, lower = findTangents(left_hull, right_hull)

    # Merge the hulls using the upper and lower tangents
    merged_hull = []

    # Add points in left_hull to merged_hull, going clockwise from upper to lower tangent
    i = upper[0]
    while i != lower[0]:
        merged_hull.append(left_hull[i])
        i = (i + len(left_hull) - 1) % len(left_hull)
    merged_hull.append(left_hull[lower[0]])

    # Add points in right_hull to merged_hull, going clockwise from lower to upper tangent
    j = lower[1]
    while j != upper[1]:
        merged_hull.append(right_hull[j])
        j = (j + len(right_hull) - 1) % len(right_hull)
    merged_hull.append(right_hull[upper[1]])

    return merged_hull
```

Because we are appending each point in our merged hull to a list, we need to iterate through our points counter-clockwise so our points in our merged hull will be ordered clockwise of one another. We work from our left hull first, working our way counter-clockwise down from the upper tangent to the lower tangent. From there, we can transfer over to the point intersecting the lower tangent on our right hull, then work our way up by adding the next counter-clockwise point until we reach the upper tangent. This ensures that the hull we return in our main "computeHull" function maintains a clockwise order.

```
def computeHull(points, initial = True):
    # If we have three points or less, we have a triangle, line or single point which is its own convex hull
    if len(points) <= 3: return points

    # Sort the points by x-coordinate before making a recursive call, set initial = False so we don't sort by x-coordinates again
    if initial == True:
        points = sorted(points, key=lambda x: (x[0], x[1]))
        initial = False

    # Recursively find the convex hulls of the left and right point
     sets
    mid = len(points) // 2
    left_hull = computeHull(points[:mid], initial)
    right_hull = computeHull(points[mid:], initial)

    # Merge left and right convex hulls
    convex_hull = mergeHulls(left_hull, right_hull)

    return convex_hull
```

Our main function follows a simple Divide and Conquer approach, with only one small but necessary tweak: we sort our points by their x-coordinates in our initial pass. This is so that when we perform our divide step, no two subgraphs will have overlapping points or hulls, but it is only necessary in our initial pass so we keep track of an implicit "initial" variable here. Afterwards, we recursively divide our set of points in half, until we hit our base case: a set with three or less points, which is its own convex hull so we simply return it. From there we will work our way back up the recursive tree, finding the hulls for two halved subgraphs, combining these hulls, then returning it for the next recursive level.

3 Tests and Results

To make sure my implementation works, I've implemented several test cases to test key component of my program. To the human eye, evaluating the correctness of a convex hull algorithm should be easy since if we can graph our points as well as the convex hull our program produces, we can pinpoint which points are missing or which points should not be included. Therefore, for every test case in test_convex_hull.py, I used MatPlotLib to produce a graph to evaluate our results. We will go through each test case one-by-one.

Convex Hull Base Case: The trivial case for finding a convex hull is if we have a set of only one, two, or three points. Since a single point, a line, and a triangle don't have any other points to "contain", they are their own convex hulls. Therefore, if our main "computeHull" function receives a list of three or less points, it should simply return that set. This is the only test case we don't provide a graph for.

Merge Defined Triangles: After hitting our base case, the direct level up our recursive tree will require our algorithm to merge two sets of three or less points into one convex hull. Most of the heavy lifting by our program is done through our findTangents and mergeHulls functions; therefore, we need to evaluate if mergeHulls works correctly at this recursive level. We provide two triangles defined by points [(30, 70), (40, 20), (50, 30)] and [(60, 50), (70, 40), (80, 70)], so they are both split along the x-axis. Calling mergeHull on these points should create a 4-sided polygon whose sides consist of the upper and lower tangent as well as a side from each triangle, and we see that it does in our graph below.

Merge Random Triangles: To ensure that our mergeHull function works for any set of points one recursive level above our base case, we will randomly generate two sets of triangles on a $(0, 100) \times (0, 100)$ grid. Our mergeHull should produce a polygon that has at most three points inside its resulting convex hull, since our convex hull can range from a large triangle to a hexagon. We see that the diagram below confirms this.

Merge Defined 5-sided Polygons: Several recursive levels above our base case, we should be merging n-dimensional convex hulls, so we must test our mergeHulls function on larger shapes than triangles. We define 10 points [(11, 30), (23, 42), (34, 11), (54, 54), (55, 37), (76, 87), (87, 99), (89, 31), (90, 34), (99, 45)] that when split in half, create two 5-sided polygons that could represent our left and right convex hulls at any level. Then our resulting merged convex hull should consist of at least 5 points (meaning all other points are contained within this perimeter), and we see that the optimal convex hull is created below.

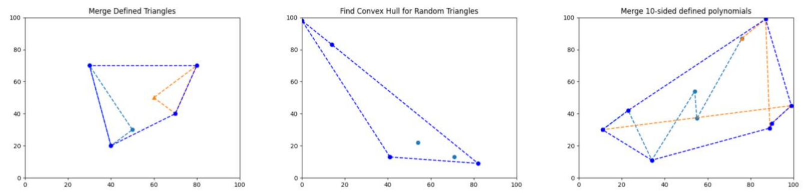
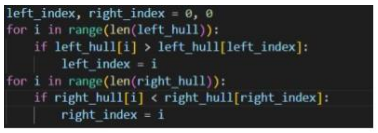


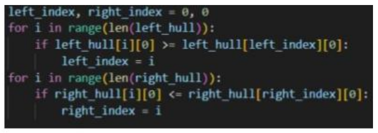
Figure 4: Notice that when we merge two n-sided polygons, the resulting convex hull will consist of at least n points. We see that all other points that are not in our convex hull are contained within them, so our program passes our defined and randomized *trivial* cases.

I will take a detour and highlight an interesting test case for an example our program initially did not perform well on, but became much better through a change in implementation. Notice the figure below on the bottom left - it's the convex hull for an instance of 100 randomly generated points. The hull has an "indent" in the bottom right that our convex hull program should NOT have counted. Because we are trying to find the **minimum** number of points that form the boundary of a set of points, instead of counting the points inside this indent, our program should have connected two points between this indent.

After changing the code for finding the left/right-most points in the left and right hull in findTangents to only consider the x-coordinates and to be inclusive, our program smoothed the hull over this indentation. We have mentioned why this produces better convex hulls: when we have a greater distance between our initial left and right hull elements, our tangent slopes become more drastic, letting us find better upper and lower tangents faster. Better upper and lower tangents means smoother and rounder convex hulls.



```
left_index, right_index = 0, 0
for i in range(len(left_hull)):
    if left_hull[i] > left_hull[left_index]:
        left_index = i
for i in range(len(right_hull)):
    if right_hull[i] < right_hull[right_index]:
        right_index = i
```



```
left_index, right_index = 0, 0
for i in range(len(left_hull)):
    if left_hull[i][0] >= left_hull[left_index][0]:
        left_index = i
for i in range(len(right_hull)):
    if right_hull[i][0] <= right_hull[right_index][0]:
        right_index = i
```

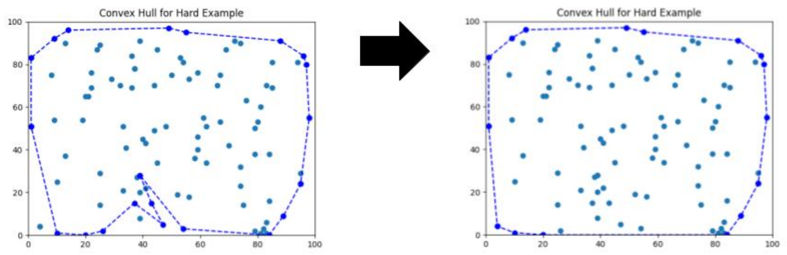


Figure 5: With a simple fix in how we initialize our indices for finding the upper and lower tangents, our convex hulls became noticeably more smooth among harder examples.

n **Random Points:** This is the true benchmark for our computeHull program. Our main test case will generate n random points on a $(0, n) \times (0, n)$ grid, generate the convex hull using computeHull, then display the points and convex hull on a graph. For most examples, we find the optimal minimum convex hull, where there are no downwards indents that could be smoothed over or points outside our hull. However, there are some examples that still produce these sorts of errors. Most notably, our program will disregard points close to the corners of our grid, forgetting to include them inside the convex hull. Outlying points are also a reoccurring issue.

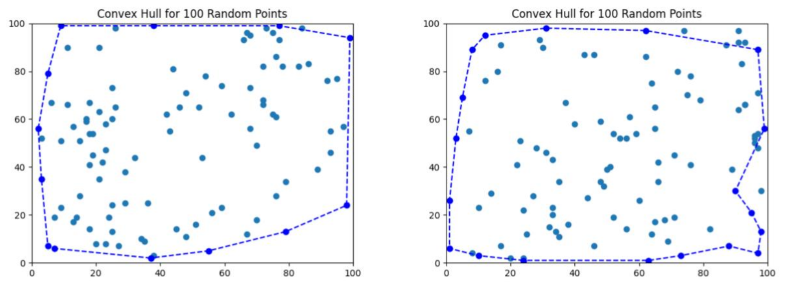


Figure 6: On the left is a common example of a correct convex hull. On the right is an example where our convex hull does not contain several points, most notably on the rightmots upper corner. You will also notice its convex hull contains more points than it should: look at the bottom right corner - it has a jaged edge, which would be smoothed over by connecting the point on the edge to a point farther down the x-axis.

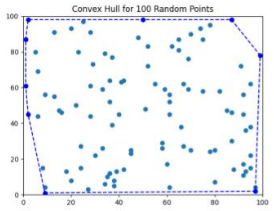
As n increases, our program will most frequently find the correct convex hull, but these issues still occur. Interestingly enough, outside points and edges become much less worse when n becomes *much* bigger, so compare to $n = 100$, the convex hulls for $n = 1000$ tend to be more smooth, simple, and contain less outlying points.

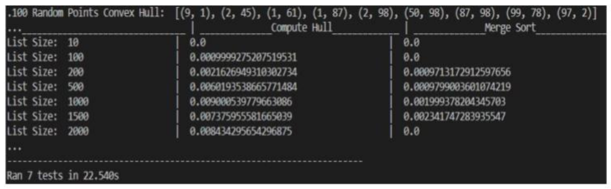
Implementation Note: I suspect this is an issue with co-linearity, since when we have several points bunched-up in a corner, it's possible that our orientation of three ordered points function may return a 0. However, for reasons I have not been able to figure out, computeHull will run into infinite loops if we change findTangents to include co-linearity as a valid condition to move our points clockwise or counterclockwise (Meaning for our "while True" loops, changing > 0 or < 0 to ≥ 0 or ≤ 0 will cause our program to get stuck).

Running Time Analysis: Our final evaluation will be to find the running time of our algorithm. Because we have a Divide and Conquer algorithm, we want to express our running time as a recurrence relation.

First, we consider our main computeHull function: at the initial level, we sort all of our points by their x-coordinates. We are using Python's built-in "sorted" function, which has a running time of $O(n \log(n))$. At every recursive level, our main function divides our algorithm in half, makes two recursive calls, and calls mergeHulls as its merge step. Dividing our problem into two equal halves means our recursive term should be $2 \cdot T\left(\frac{n}{2}\right)$. Our base case simply returns a list of size $n \leq 3$, so putting this all together gives us $T(n) = \begin{cases} O(1) & \text{when } n \leq 3 \\ 2 \cdot T\left(\frac{n}{2}\right) + O(n \log(n)) + M(n) & \text{otherwise} \end{cases}$, where $M(n)$ is the running time for our mergeHulls algorithm.

Our mergeHulls algorithm sorts our points in clockwise order, finds the upper and lower tangents of the given left and right hulls, then iterates through both hulls to finally merge them. findTangents iterates through both hulls a total of four times when finding the leftmost/rightmost elements of each hull and finding the element in each hull that would intersect our tangents, so it has a linear running time. Our merge step in mergeHulls is also linear because it also just iterates through each hull. However, mergeHulls is NOT linear because our our initial clockwise sort, so at best, our combine step is $O(n \log(n)$. Thus, our total recurrence relation is $T(n) = \begin{cases} O(1) & \text{when } n \leq 3 \\ 2 \cdot T\left(\frac{n}{2}\right) + O(n \log(n)) & \text{otherwise} \end{cases}$, which when evaluated gives us a running time of $O(n \log^2(n))$.





```
.100 Random Points Convex Hull:  [(9, 1), (2, 45), (1, 61), (1, 87), (2, 98), (50, 98), (87, 98), (99, 78), (97, 2)]
..._____________________________ | _____________Compute Hull_____________ | _____________Merge Sort_____________
List Size:  10                   | 0.0                                    | 0.0
List Size:  100                  | 0.0009999275207519531                  | 0.0
List Size:  200                  | 0.0021626949310302734                  | 0.0009713172912597656
List Size:  500                  | 0.006019353866577148 4                 | 0.0009799003601074219
List Size:  1000                 | 0.009000539779663086                   | 0.001999378204345703
List Size:  1500                 | 0.007375955581665039                   | 0.002341747283935547
List Size:  2000                 | 0.008434295654296875                   | 0.0
...
----------------------------------------------------------------------
Ran 7 tests in 22.540s
```

Figure 7: For comparison, we plot the time it takes to run computeHull on sets of n points versus the time it takes to run MergeSort on sets of integers of the same size. The running time of computeHull grows by a small factor greater than MergeSort. Above our running time graph is our convex hull for the graph to the left: it takes $O(n \log^2(n))$ time to find a **clockwise** convex hull.

References

[1] An efficient way of merging two convex hulls, Feb 2019. `https://algorithmtutor.com/Computational-Geometry/An-efficient-way-of-merging-two-convex-hull/`.

[2] Rajeev Agrawal. Orientation of 3 ordered points, Apr 2023.

[3] Sanjana Babu. Orientation of three ordered points, Nov 2021. `https://iq.opengenus.org/orientation-of-three-ordered-points/`.

[4] cody, user17762, and Naveen Kumar. How to solve this recurrence $t(n) = 2t(n/2) + n \log n$, Jun 2012. `https://math.stackexchange.com/questions/159720/how-to-solve-this-recurrence-tn-2tn-2-n-log-n`.

[5] Aarti Rathi and Amritya Vagmi. Tangents between two convex polygons, Mar 2023. `https://www.geeksforgeeks.org/tangents-two-convex-polygons/`.