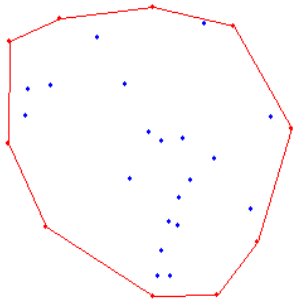
CS255
Program2: ConvexHulls
A Divide-and-Conquer Algorithm
15 points
Due: Fri, March 30 – 11:59pm

Convex Hulls

There is a complex mathematical definition of a convex hull, but the informal definition is sufficient for our purposes. The convex hull of a set of points in the plane is the shape taken by a rubber band stretched around nails pounded into the plane at each point:



There are many different convex hull finding algorithms. All of them take in a set of points and return a counter-clockwise (ccw) ordered subset of those points that form the convex hull. For this assignment you will be implementing 2 of these convex hull finders: QuickHull and MergeHull. As the names suggest, they are similar in structure to QuickSort and MergeSort, respectively. That is, they are both Divide-and-Conquer algorithms. Additionally, they have similar time complexities as those sorts, making them some of the more efficient convex hull finding algorithms.

ConvexHullFinder

Since you will be constructing 2 different implementations that both find the convex hull of a set of points, you should construct an interface for them both to implement. That is, client code that is using either QuickHull or MergeHull shouldn't really care which implementation it is using. All it should care about is that it has a convex hull finder that can find convex hulls by calling a particular method. This will allow for easy switching of the algorithm without major rewriting of the client code. This is the functionally that the ConvexHullFinder Interface provides.

The only method in the ConvexHullFinder Interface is:
public List<Point2D> computeHull(List<Point2D> points)

The computeHull function should take in a List of Point2Ds (representing the points) and return a List of Point2Ds (representing the convex hull). Note that the returned List of Points2Ds should be a subset of the full points List, **and the hull points should be stored in ccw order**.

You will then be implementing the QuickHull and MergeHull classes, both of which implement the ConvexHullFinder Inderface:

QuickHull

QuickHull should implement ConvexHullFinder. That implies that it needs a computeHull method that matches the one in the interface. Additionally, you need an internal method that will call itself recursively. That is, there is some initial setup work to do when starting the hull finding process. After that one time initial work, each additional step is exactly the same – that is where the recursive method comes into play.

The initial setup work is to:

- Find the leftmost and rightmost points
- Construct a line connecting the leftmost to rightmost points
- Divide the points into 2 sets, those above the line and those below the line
- Call the recursive method, giving the line and the top set of points to produce the top half of the hull
- Call the recursive method, giving the reversed line and the bottom set of points to produce the bottom half of the hull
- Once both halves are obtained, glue them together to form a ccw set of hull points – note any starting location is acceptable, but the points need to go in ccw order around the hull.

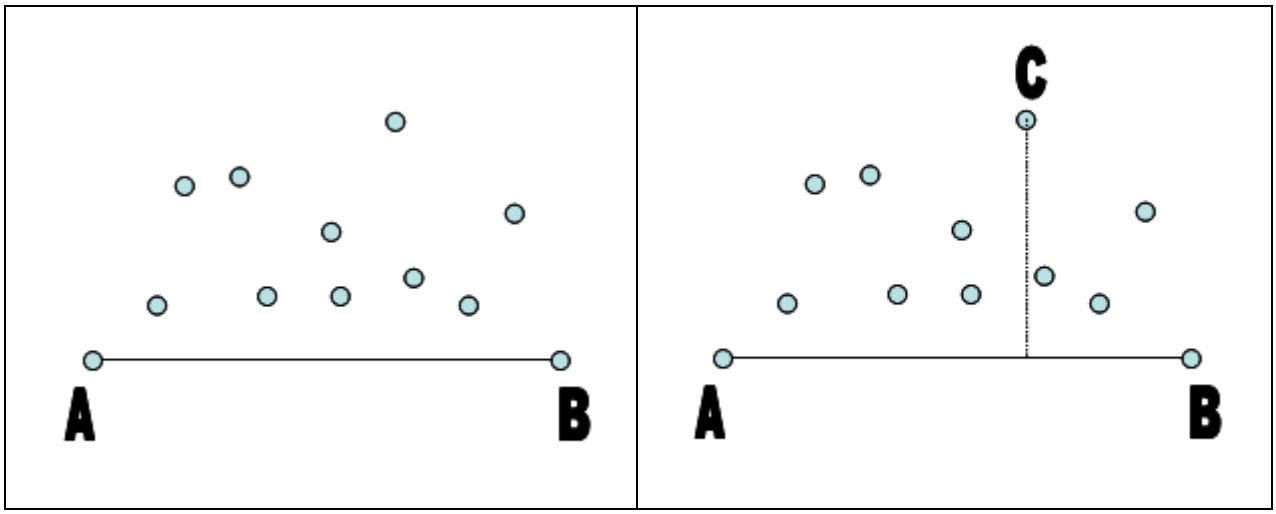
That is, the recursive method only works to find a half a hull at a time. You always need to give it a baseline and the points to the side of the line you are interested in. The recursive method's signature should be:
private List<Point2D> recursiveQuickHull(Line2D lineAB, List<Point2D> pointsAB)

The recursive method will go through the 3 major steps of Divide-and-Conquer:

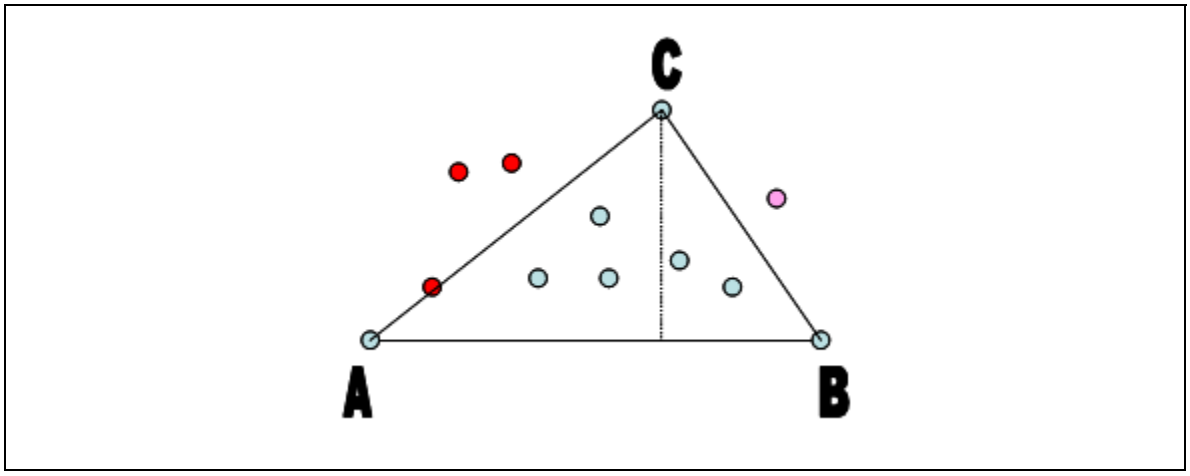
- Divide the large problem into smaller subproblems
- Recursively Conquer the smaller subproblems
- Combine the results of the subproblems into a solution for the large problem

Divide:

- Given a baseline from point A to point B (lineAB) and a set of points above this line (pointsAB), find the point C that is farthest from the line.



- Form a line from A to C (lineAC) and select all the points to the left of this line (pointsAC)
- Form a line from B to C (lineBC) and select all the points to the right of this line (pointsBC)

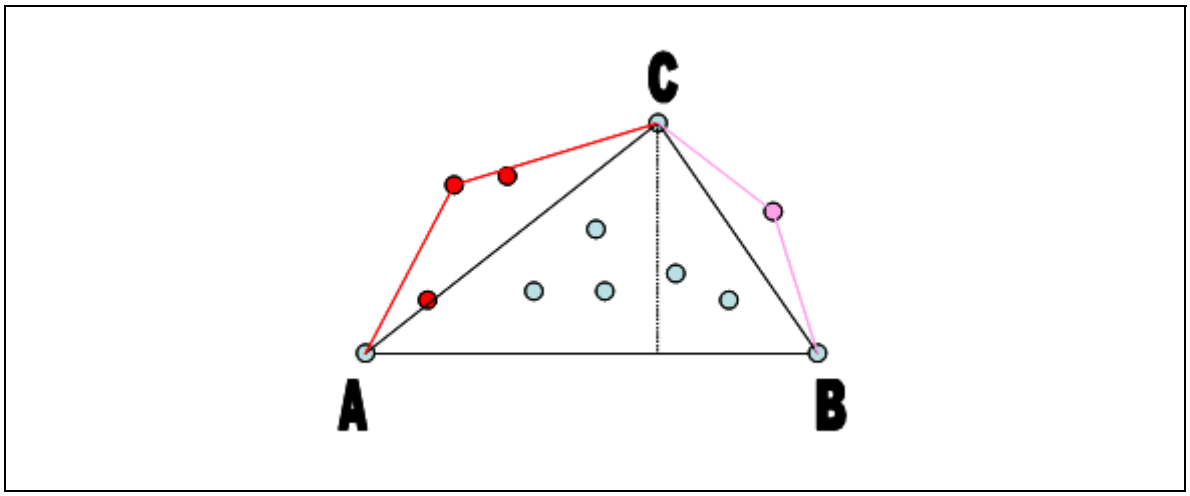


The above divides the large problem of finding the half-hull of the pointAB into 2 smaller problems of finding the half-hulls of the pointsAC and pointsBC.

Conquer:

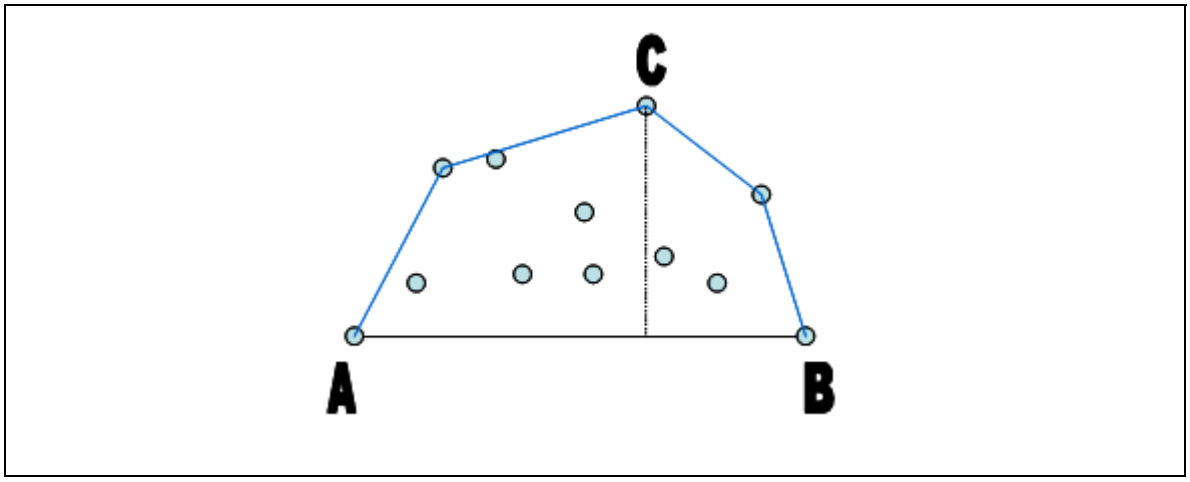
- Call the recursive method on both subproblems (lineAB with pointsAB and lineBC with pointsBC)

When this returns it will provide you the half-hulls from A to C and from C to B



Combine:

- Simply append the two half-hulls together to obtain a half-hull from A to B. Make sure your final result is in ccw order.



Note that no work takes place in the combine, only in the divide. This is the exactly the same as the QuickSort algorithm. Also note that, just as with QuickSort, the worst case complexity is $O(n^2)$. However, in practice, the average case is much faster.

Feel free to add other private helper methods to this class if it helps make the code more readable. However, you are **not** to use instance variables to gather the hull results – this will cause you all sorts of trouble – trust me on that one. Plus, you aren't allowed :-P

You should also take a close look at what you pick for your base case – you don't want duplicate points in your final convex hull list, nor do you really want to have to remove

the duplicates when putting it together. You would rather pick your base case in such as way as to have the duplicates automatically not be included.

MergeHull

The same information about the interface and recursive method that was discussed above for QuickHull also applies to MergeHull.

One of the caveats about MergeHull is that we want to keep its running time to $O(n \log_2 n)$. That is, there are certain implementation decision one can make when implementing sub-steps of this algorithm that will cause it to go up to $O(n^2)$. We want to avoid making those mistakes, so that we have a truly $O(n \log_2 n)$ algorithm (everycase).

The initial setup work is to:

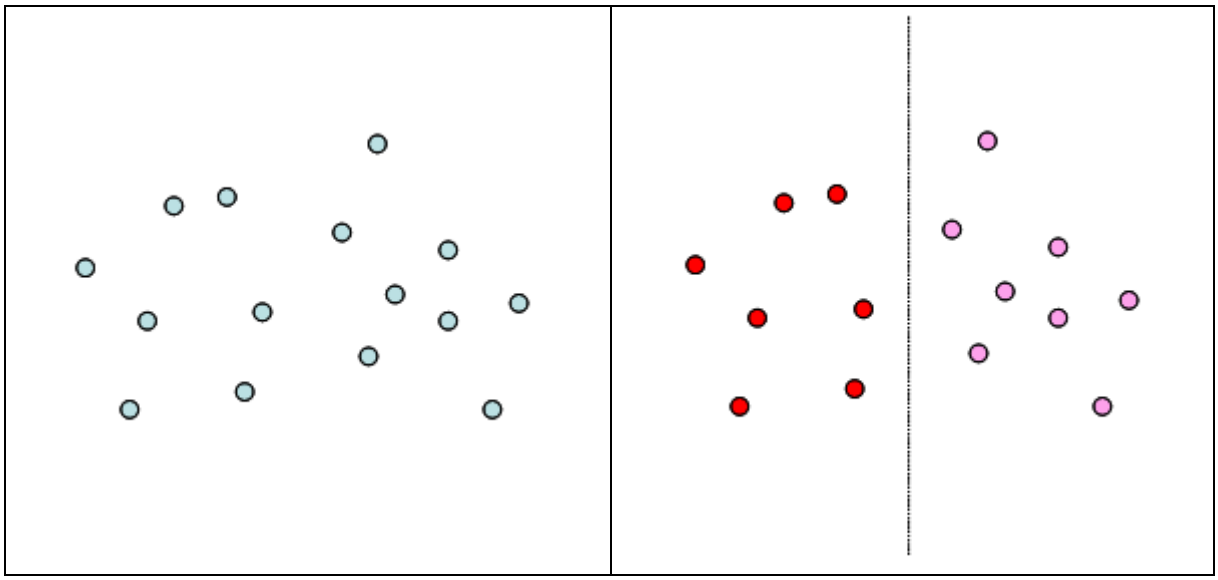
- Sort the points into order from leftmost to rightmost. Note that we want our final running time to be $O(n \log_2 n)$. This step happens before anything else, so our final running time will be O(this sorting step) + O(rest of the mergehull). If we pick a sort that is $O(n^2)$ then the entire algorithm can't be better than $O(n^2)$. If we pick a sort that is $O(n \log_2 n)$, then as long as the rest of the mergehull algorithm is $O(n \log_2 n)$ we can obtain the $O(n \log_2 n)$ that we want. Thus you will need a sort that is $O(n \log_2 n)$. Note that Collections.sort is that fast, so you should use that as your sorting algorithm. You should not code your own sorting algorithm.
- Call the recursive method with the sorted set of points

That is, the recursive method assumes that the points are in sorted order. The signature of the recursive method is:
private List<Point2D> recursiveMergeHull(List<Point2D> points)

The recursive method will go though the 3 major steps of Divide-and-Conquer outlined in the QuickHull discussion. The only difference is where the majority of the work takes place (i.e. for MergeHull it is in the Combine, for QuickHull it is in the Divide).

Divide:

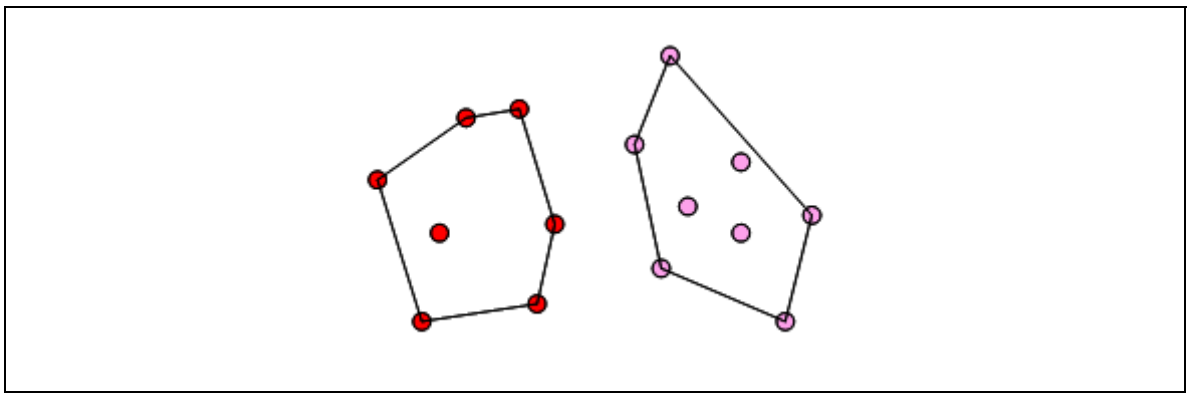
- Start with a set of points that are ordered from left to right and divide them into 2 equal sets



Note that since the incoming list of points is already sorted, this requires no work. Just as in MergeSort, all the work will be in the combine (merge) step.

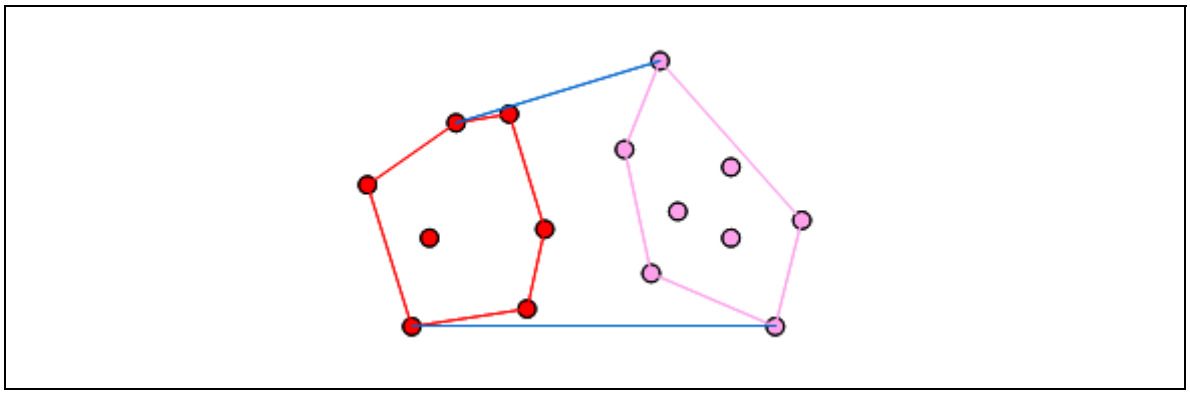
Conquer:

- Recursively compute the hull of each subset of points. Note that you should be careful about picking your base case.

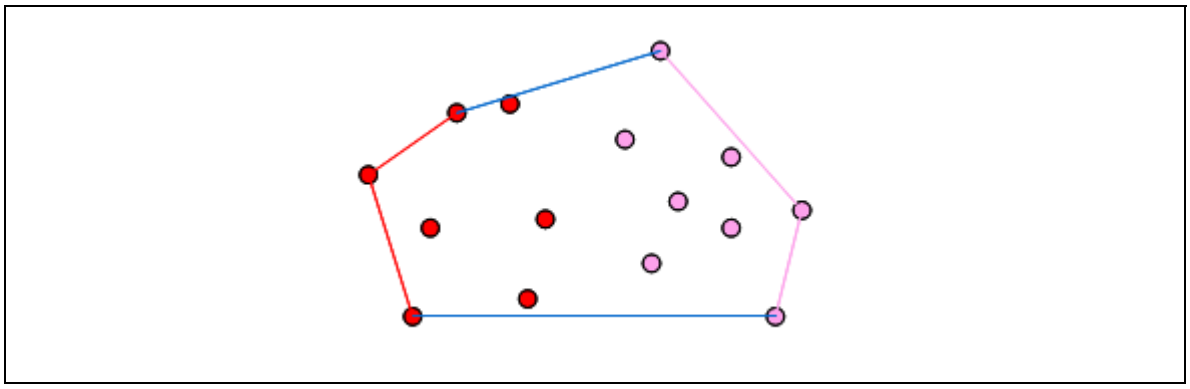


Combine:

This is the step where all the work takes place. It is the goal of this step to combine the two smaller hulls in to a larger hull. In order to do this we must find the lower tangent line and the upper tangent lines:



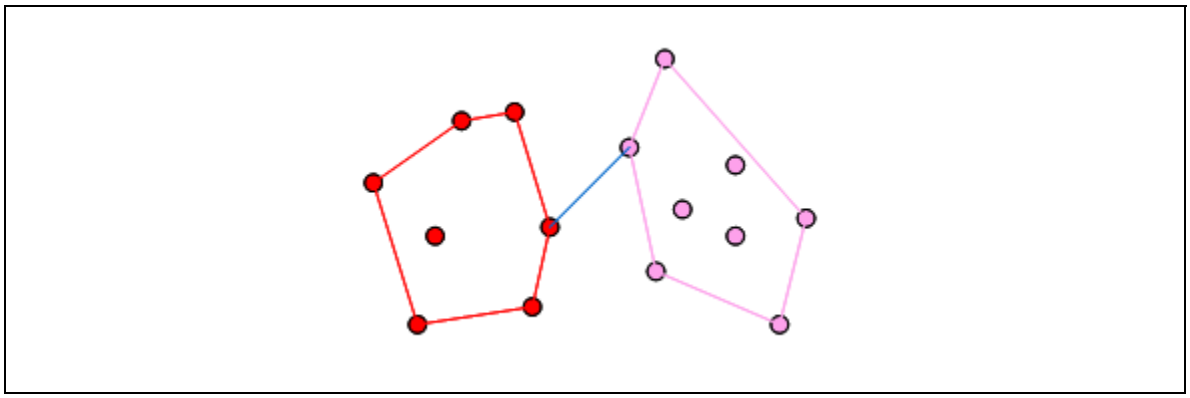
Once we have these lines it is easy to combine the two hulls:



However, the naive algorithm for finding these tangent lines by comparing all possible pairs of points is $O(n^2)$. This won't do, because we want the final running time to be $O(n \log_2 n)$. Since there are going to be $O(\log_2 n)$ levels in our tree, we can only do n amount of work at each level. Thus, we need to find a linear time algorithm for finding the tangent lines.

To find the tangent lines in linear time, do the following for both the lower and upper lines independently:

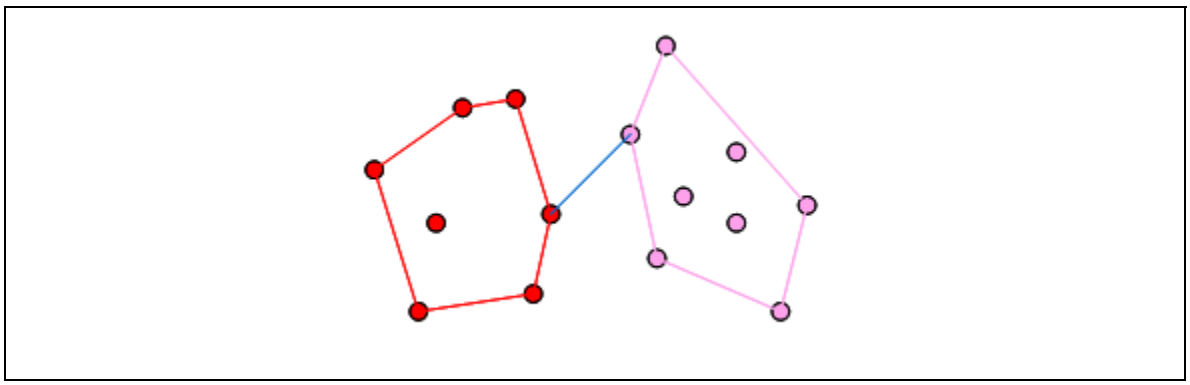
- Find the rightmost point in the left hull and the leftmost point in the right hull. Connect these lines together to form a starting tangent. Note that you are finding the rightmost and leftmost points in the returned hull, not in the incoming points. The point here is that the returned hull is in ccw order but not in sorted order (as the incoming points were). So you must actually search to find these points.

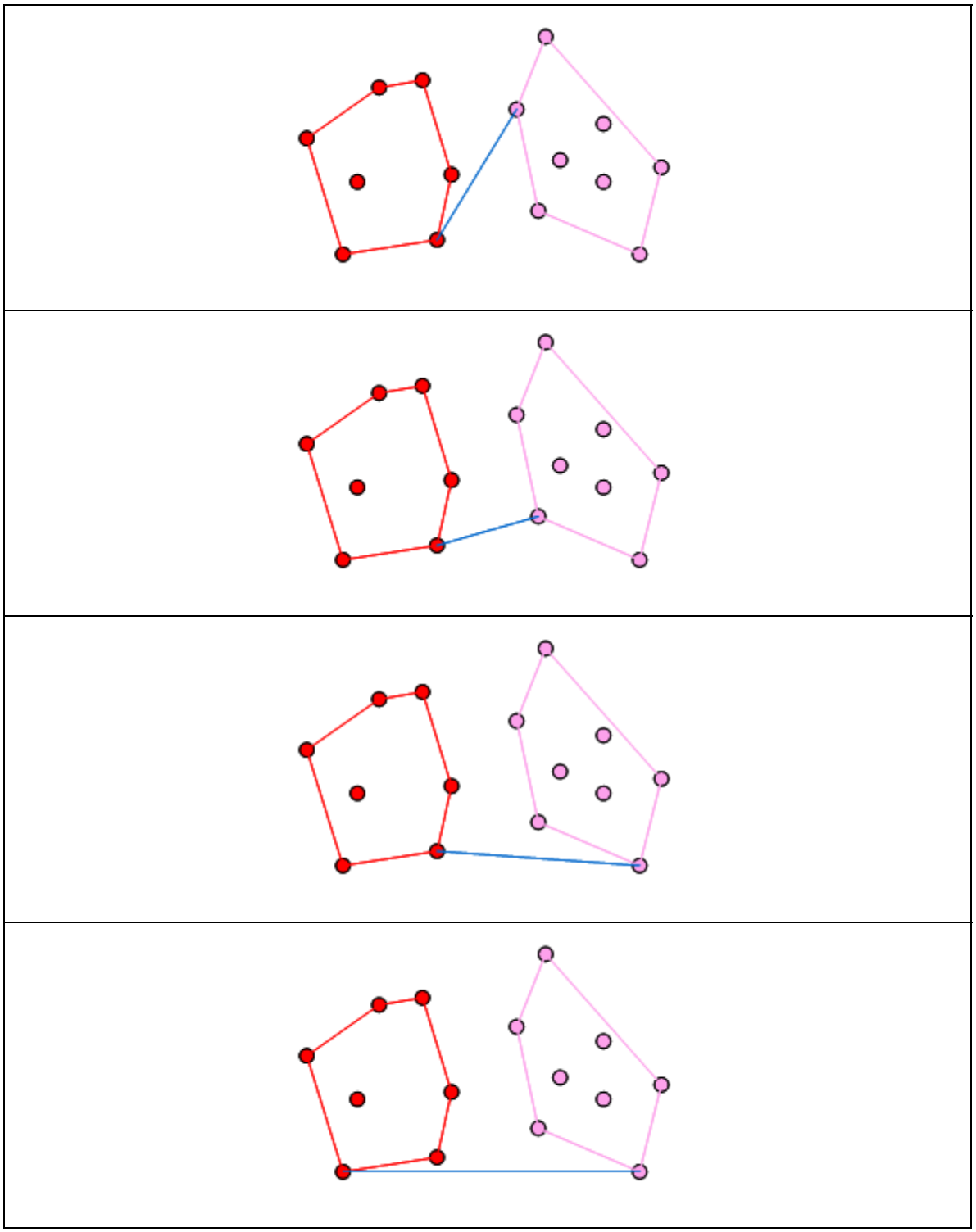


- Now "walk" the tangent line downward until it is the lower tangent line. The pseudo-code for walking looks like:

```
A ← rightmost point of left hull
B ← leftmost point of right hull
While (T = AB is not the lower tangent to both left and right hulls) {
    While (T not lower tangent to left hull) {
        A ← A – 1  (assumption is that hulls are defined in ccw order)
     }
    While (T not lower tangent to right hull) {
        B ← B + 1  (assumption is that hulls are defined in ccw order)
     }
}
```

The above "walking" algorithm produces the following intermediate steps:





Note that to determine if some line, T = AB, is a lower tangent to the left hull you only need to check the 2 points on either side of point A (A-1 and A+1). If both these points are above the line T then all the points in the left hull are above the line T. That result comes from the fact that the hulls are convex. Note that you

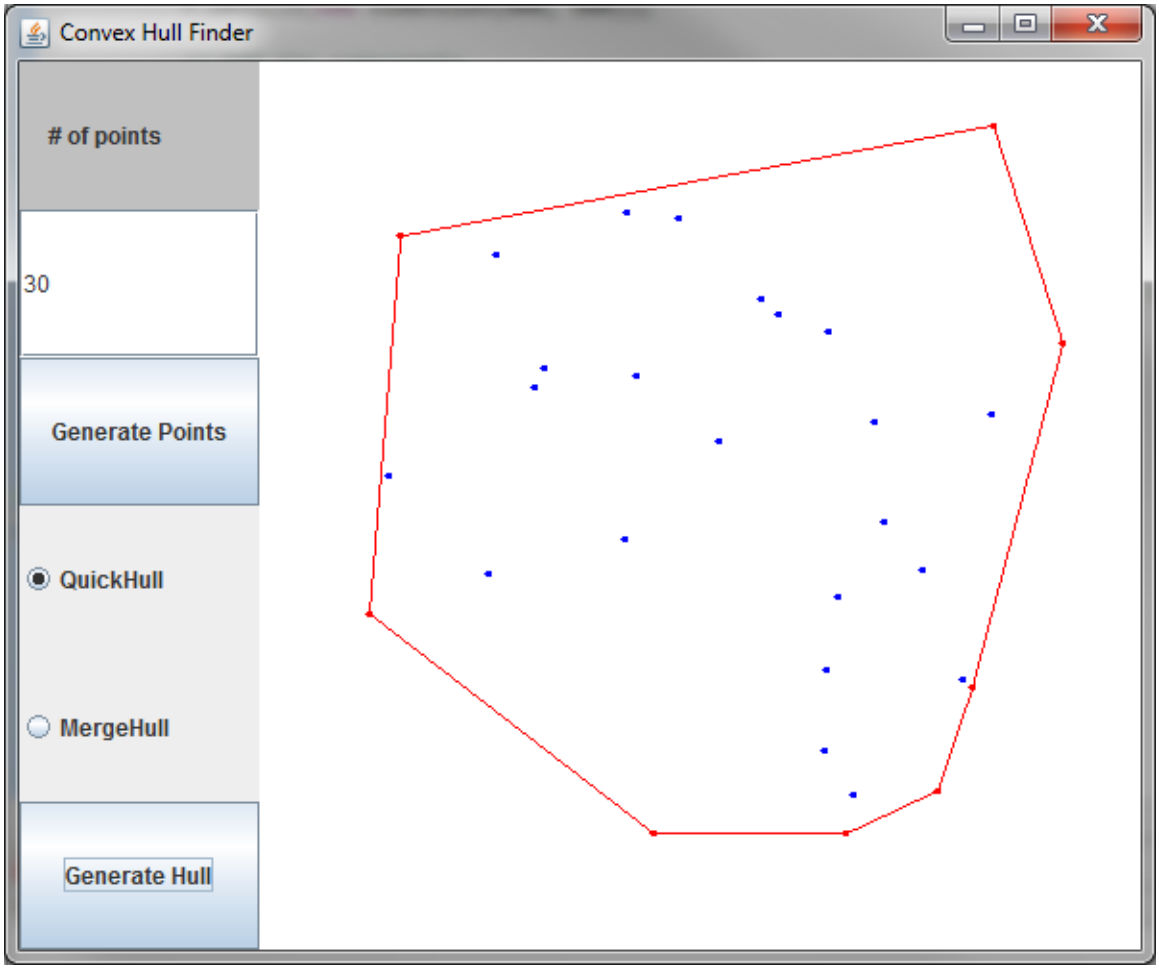
have to be careful about accessing the A-1 and A+1 elements so you don't go out of the array bounds.

Feel free to add other private helper methods to this class if it helps make the code more readable. However, you are not to use instance variables to gather the hull results – this will cause you all sorts of trouble.

In addition, you should think carefully about your tangent walking code. You certainly don't want to write code to walk the tangent up and code to walk the tangent down if the same code can be made to do both.

The GUI

In addition to constructing the basic convex hull finding algorithms, you also need a GUI to display the results. The GUI is given to you.



Submission

You are to turn in:

- A printed picture of your GUI results for a sample QuickHull run
- A printed picture of your GUI results for a sample MergeHull run
- A printed single page (max) document describing any problems that your code might still have if it is not complete. Or if it is complete, possible places for improvement in your code.
- A jar file of your entire package of source code so that I can run and grade your program on my machine – this should be placed in the class W: drive.

You may work with a partner.