Brandon Lee

Homework 2

CSC 537

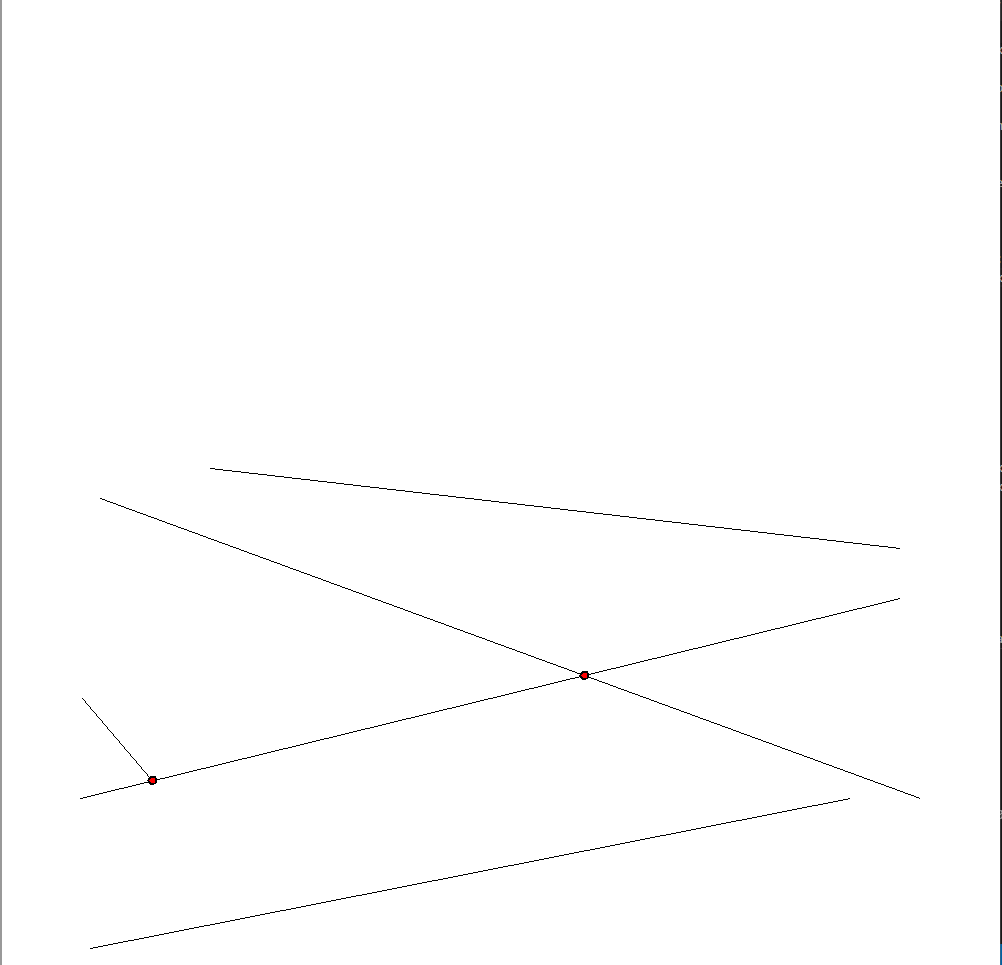
Part 1 Problem 1:

For this problem, I took the red black tree provided and modified the functions searchx, Above, swap, insert\_segment, and insert\_helperx. I also took the seg\_inter.py file and added modified find\_intersections to take into account left, right and intersection events, I also added a function that uses the line equation for two segments and sees where the intersection point is, if it exists.

The insert\_segment works the same way as the insert function, however it passes the segment into the tree instead of just a node value. The insert\_helperx and the searchx work the same way as the insert\_helper and the search functions work in the red black tree. Above, checks if the segment is above another segment. And Swap switches the two nodes that is fed into the function.

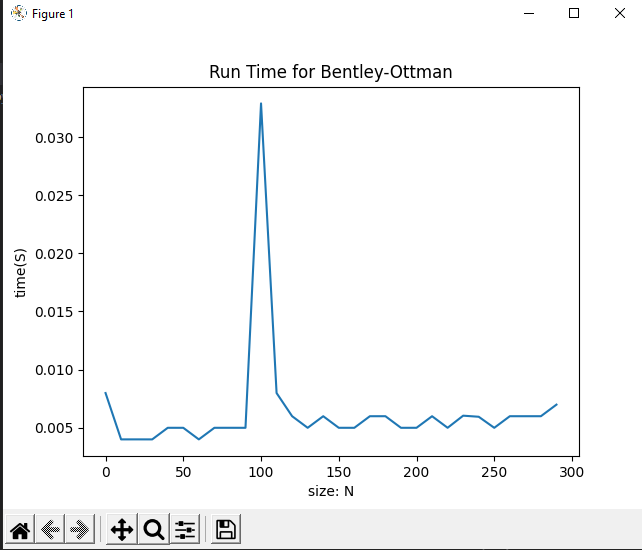
The most important and challenging changes were the intersect function and implementing the right, left, and intersection events. For the intersect function, using 4 points and assuming that point 1 and 2 are the same segment, and that 3 and 4 are another segment. I used the math found here, <https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection>, to construct my function. Since we know the 4 points, we can construct two line segments using the formula y=mx+b. Of course, we need to find the slope and the y intersect to construct these, so I added two simple functions. These functions are slopeOf, and find\_B, which return the slope and y intercept using 2 points. After collecting the Y-intercepts of the segments and the slopes, I just needed to apply the formulas into the x and y values, and return those values.

Figure 1: Simple Line Intersect Function



Here is an example of my line-sweep Bentley-Ottman algorithm with a simple set of line segments.

Figure 2: Runtime of Bentley-Ottman



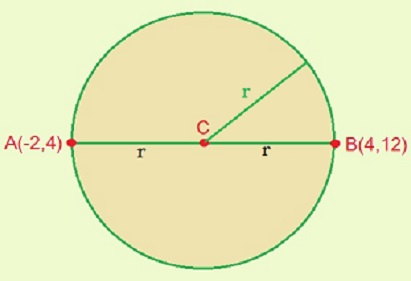
I ran the function 30 times, using 0 to 300 random generated line segments. The runtime increases consistently except for two outliers. At N=0 and at N=100. I think the N=0 discrepancy is due to the logic of the algorithm being designed for N>0 inputs. However, I don’t know why there is this huge discrepancy for N=100.

Part 1 Problem 2

We can create a plane sweep algorithm like the Bentley-ottman algorithm for finding the intersection points to n circles, instead of line segments, in a similar time as the segments. This would require a few modifications, but we can keep the same idea.

One thing we should consider is the radius of the circle, we can find this by finding one end of the circle to the other. So, if we start sweeping through the plane and we see one end of the circle, then go through until we see the other end of the circle. Half of that distance will be the radius. We have left and right end points with Bentley-Ottman, well we have a left and right end points in a circle. So, we can keep the same events, i.e. event on right, left and intersection.

Figure 3: Endpoint Circle Example



Here is what I mean by two endpoints, it should be easy to find the radius too because we can keep the same y values, and just measure the difference between the x values.

We need to make sure that we have a way to find the radius of each circle since we will use these to compare to other circles to see if they intersect. We will also need the midpoint of these circles since that will also play a roll. So, let’s say that we have a circle 0 and circle 1 with r0 and r1 being their radii respectfully. And let’s say that d is the distance between their midpoints. There are several cases, if d > r0 + r1 then they don’t intersect. If d < |r0-r1| then one circle is contained in another, so there are no solutions, d = 0 and r0=r1, then the circles are the same and have infinite solutions. Then if d<r0+r1 and d>|r0-r1| then the circle will intersect.

So, now that we have a way to determine if there are intersection points, we need to know what circle to compare to each other.

We will need to take break all the circles down to half circles, each with a right and left endpoint. The semi circles would be of the upper and lower halves.

**Algorithm.**

Start by breaking down every circle into two semi-circles.

Then we have a priority queue with the leftmost and rightmost endpoints of each circle sorted by their x coordinate.

When we encounter the start event of a circle, we insert the top and bottom halves of that circle, into a balanced binary search tree. Every time we add a new circle to the tree, we test for intersections. When we reach the endpoint of a circle, we remove both half circles.

We test for the intersection in the same way I described before, with comparing the radii and the distance of the midpoints of the circles.

We can then find the intersections using the midpoints, and the distances between the two circles, so we do this by comparing the circle equations. This being (x-x0)^2+(y-y0)^2=r^2, where x0 and y0 being the midpoints of the circles. We need to run through a system of equations to find the intersection points, if they exist, of two circles. So, let’s take two equations for two circles respectively.

(x-x0)^2+(y-y0)^2=r0^2

(x-x1)^2+(y-y1)^2=r1^2

|

|

-2x(x0-x1)-2y(y0-y1) = (r0^2-r1^2)-(x0^2-x1^2)-(y0^2-y1^2)

|

|

Y = ((r0^2-r1^2)-(x0^2-x1^2)-(y0^2-y1^2)+ 2x(x0-x1))/(-2(y0-y1)

With that equation, we can find the y values where the intersections occur, and we can use those y values to find the x values associated with them. The expression for y should also be put back into one of the original circle equations so that we can get a quadratic equation. Then, we can find the answers using the quadratic formula.

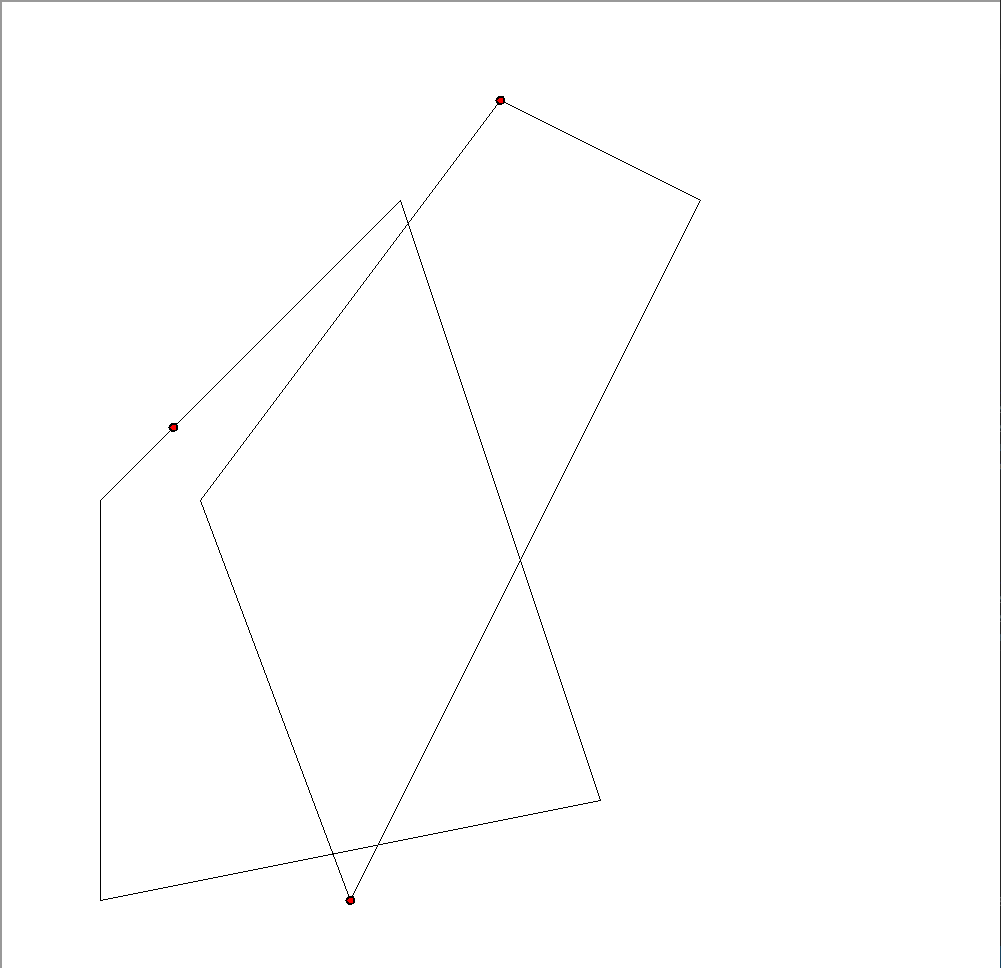
Similar to the Bentley-Ottman line segment algorithm, whenever we add, remove, or swap a semicircle, we test its new neighbors for intersection.

Part 2. Problem 1

For this problem, I used the same red-black tree file and the intersection algorithm from problem 1. However, I had to make a couple of changes. The first change that was made was two added values to the intersect function. I added an x low and an x high value that checks to make sure that the x value is within the range of the segments. This is an insight that I missed on in part 1. The line intersect function needs to find the intersection point if and only if the are within range of each other. Because any continuous line that isn’t parallel, will have an intersection point. This isn’t the case when we are comparing segments because they have a fixed length. So, I needed to add this consideration to the intersection algorithm.

Unfortunately, implementing this change with the same code from Q1, into this code resulted in interesting intersection points.

Figure 4: Line-Sweep DCEL Algorithm Results

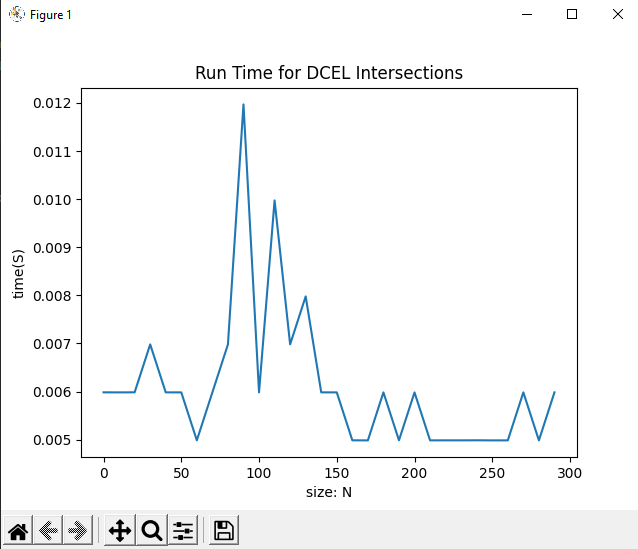




This is what I ended up with after much time trying to debug. This might mean that the way I am picking the segments to compare is problematic. In essence, I am not constructing the line-sweep algorithm correctly. This would explain the reason for the placement of the top left point. The algorithm is confused because it thinks that we should be comparing S1 to S2. And since the intersection point is between both segments, this means that I would need to change how I am constructing the x range.

Here is the runtime of that algorithm.

Figure 5: Runtime of Sweep-Line Algorithm for DCEL:



I ran this again 30 times with size N between 0 and 300. With increments of 10. Again, we get interesting results around the N=100 mark.

I then tried to solve this question using a brute-force algorithm. This resulted in better luck, however still not the results we want. This is likely due to the way I have my intersect function structured, since it can find the intersect points, but it needs a smaller constraint for the x and y ranges that it would accept. Working with this problem again, I think I would change the x constraints to take the smallest range of x values that it could take, that being the maximum left endpoint of the two segments and the minimum right endpoint of the two segments. Then I would add the y ranges being the second and third smallest y values. This is because logically we can’t have an intersection point between the two values if the y value is bigger than the third smallest y or smaller than the second. The reason for this is because for there to exist an endpoint between two segments, if it is smaller than the second smallest y value, then it can only exist on the segment with the smallest y value for one of its endpoints. But it needs to exist on both segments simultaneously. The same thing is true for if it is bigger than the third smallest y value. There is one case that could fail this situation, so we would need to implement another check. That case being if one segment is completely under another segment, the algorithm could find a line intersection point between the two because the rectangle it would use to compare is between S1 and S2. So, I would need to implement a check to make sure that the bigger y value of S1 is bigger than the smaller y value of S2.

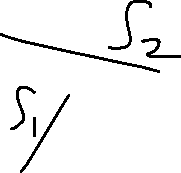


Figure 6: Brute-Force Approach:

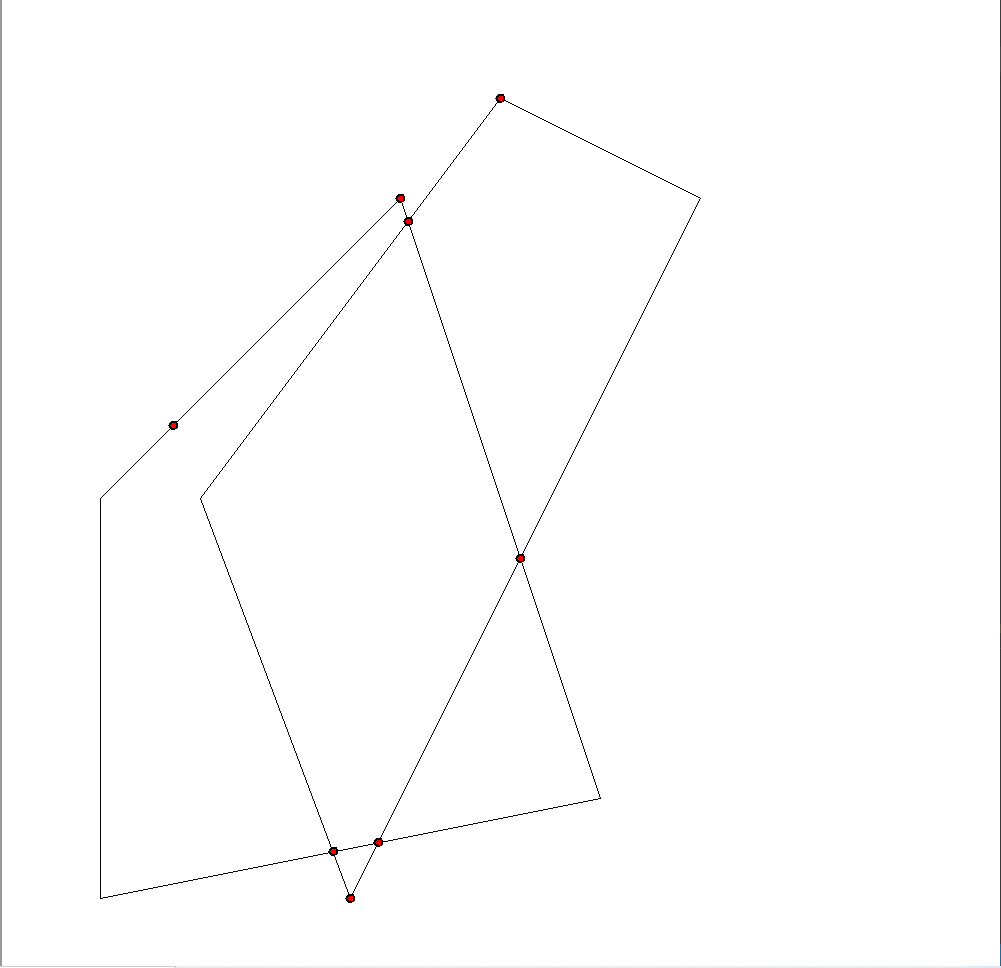
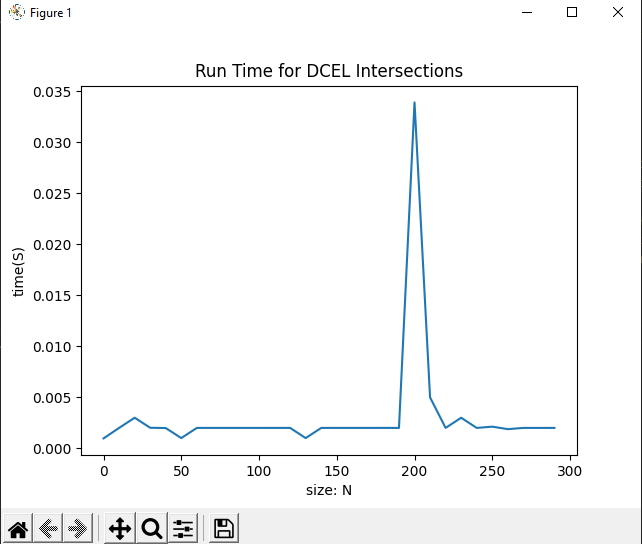


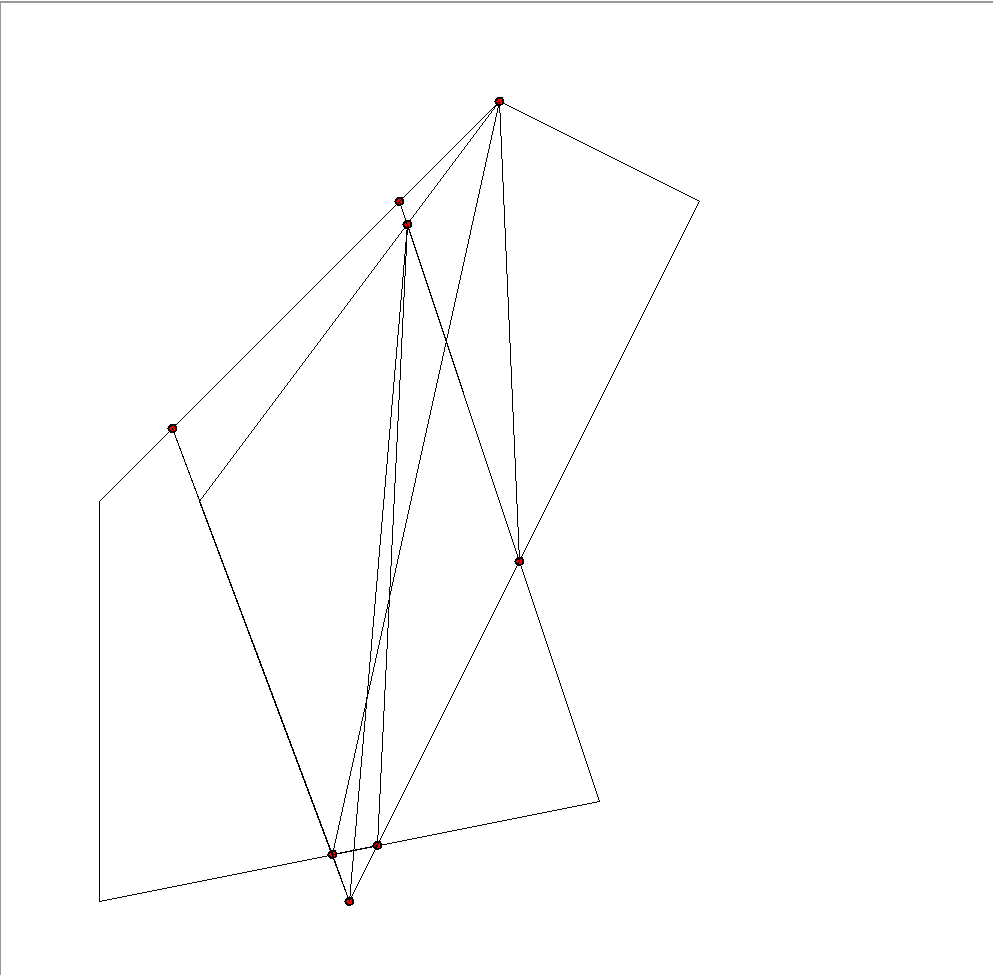
Figure 7: Brute-Force runtime:



Part 2 Question 2:

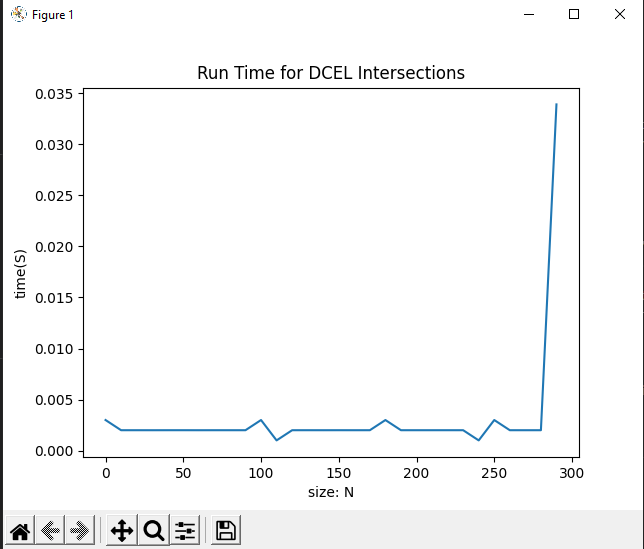
This was the toughest one for me. For this approach, I took the code from part 2 question 1 and connected the intersection points to create the in-between for the map overlay. The approach I took from there was to use the drawFace() function to color in the area where the two DCELs overlap. Although, my algorithm couldn’t get the right intersection points and the segments would cause some issues. So, I was only able to get this as my result.

Figure 8: MapOverlay Result



Another thing I would need to change if I were to work with this algorithm again, is that I need to make sure that the line segments created were in the right order. I would likely use the same approach used in HW1, with cross product and listing the points in clockwise or counter-clockwise order.

Figure 9: runtime for MapOverlay



This is the runtime of the problem, this has the same phenomena that the other runtimes have, where it peaks at one point that is unexpected. I am not sure what is causing this discrepancy.