Homework 3

Elliott Pryor
Collaborated with: Nathan Stouffer

09 March 2021

Problem 1

Given an polygonal chain P of n vertices, we define an vertex v as a *local max* if v if all edges adjacent to v are to the left of v. Show that can determine if a polygonal chain with k local maxes is simple in $O(n \log k)$ time.

We use a sweepline algorithm. But to do that, we first need to do some setup. We observe, that between two maxes, there is at most 2 x-monotone polygonal chains. (Proof later). So this means, we can decompose P into $2k$ x-monotone chains. Since they are x-monotone, they are in sorted order (either strictly increasing or decreasing).

So the algorithm: decompose P into x-monotone chains. Sort the chains by their left endpoints. Then sweepline events are the verticies. On the sweepline, we store pointers to each of the x-monotone chains sorted by y-coordinate. The sweepline (or chain object) has a pointer to the current edge of the chain that intersects the sweepline. Then to check intersection it is just line segment intersection of the edges on sweepline. When we reach a node in the chain we increment the edge to point to the next edge in the chain. We also remove old event points after we pass them, and insert a new event point for the next vertex in the chain.

Algorithm 1 Simple Chain

```
1: function CHAIN(P)
2:     Break up P into k x-monotone chains, C
3:     sort C by leftmost x-coordinate
4:     events ←leftmost x-coordinates
5:     for e ∈ events do
6:         remove e from events
7:         if e is Type 1 then
8:             insert e.chain onto sweepline
9:             add e.chain.next_node to events
10:        else if e is Type 2 then
11:            Compare this chain to neighboring chains.
12:            IF, they switch positions on sweepline, then they intersect. return not simple
13:            otherwise, update sweepline for this node. This updates sweepline edge pointer and ordering
14:            add e.chain.next_node to events
15:        else
16:            Remove e.chain from sweepline.
17:        end if
18:    end for
19:    return Simple!
20: end function
```

Runtime: So the runtime analysis is fairly straightforward. Breaking up P we can do with a single linear pass through P ($O(n)$). We can find the leftmost x-coordinate of any chain in $O(1)$ since it is in either increasing or decreasing order, so we check first or last elements. Then

sorting takes $O(k \log k)$ since there are $2k$ chains.

Then for each event: we establish that there are at most $O(k)$ things on the sweepline (all the chains), and also at most $O(k)$ events on the queue since we remove old points and add new points as we find them. So this means that any operation on either datastructure takes $O(\log k)$ time (assuming heap and balanced tree structures). Then we have a constant number of lookups/operations that take $O(\log k)$ time. We also never elaborated how to update edge pointer, but if the chains are in an array or DLL we can just increment the pointer in constant time. So each event takes $O(\log k)$. Then over the duration of the algorithm, every vertex could be explored $= n$. So $O(n \log k)$ for event processing. Since $k < n$ we have $O(n \log k)$ overall.

We also prove that the number of x-monotone chains is $2k = O(k)$. PROOF. Suppose not, suppose there are more than 2 x-monotone chains in-between two maxes. So assume there are 3 chains. Then the chain bends back upon itself 2 different times (the ends of the chains). Thus it makes an s or ƨ (inverted x). Then clearly there is another max, a contradiction.

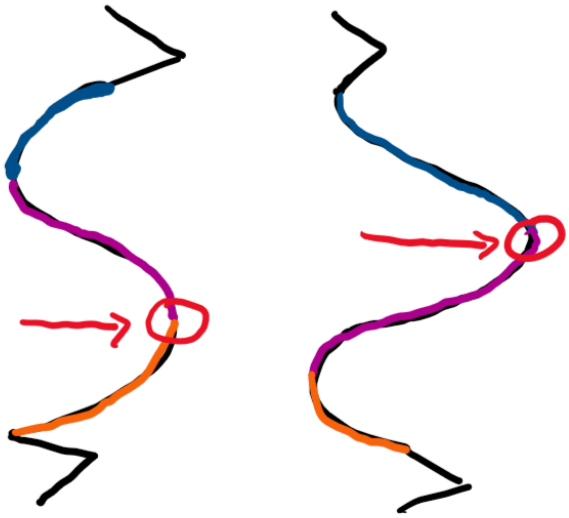


Figure 1: Two possible cases for 3 x-monotone curves between two maxes

□

Correctness:

PROOF. Consider breaking up P into entirely separate line segments. The line segment intersection algorithm will determine if there are any intersections. We show that at any event point, the state of our algorithm is equivalent to line segment intersection.

At event e the line segments on the sweepline are only those who intersect the sweepline. So, consider our algorithm at event e. We update the sweepline, during which we increment what edge the sweepline considers for comparing intersection. Then our algorithm has all the edges

of each chain that intersects the sweepline in the sweepline state. If it doesn't, then it didn't increment (a contradiction) or a chain was not on the sweepline (also a contradiction). These edges have a one-to-one correspondence with line segments in the line segment algorithm. Thus the sweepline of our algorithm has the same status at any event as the line segment algorithm.

Then we need to show that our algorithm considers all the events (up until a segment intersection). Suppose not, suppose our algorithm misses an event. Then there is an isolated edge that didn't get added (contradiction). Or it does not cover all the vertices in a chain. If we sweep the line along the chain, we see that it increments to get the next vertex. So in a sweep, we will cover every vertex. By our decomposition, of P every vertex is part of a x-monotone chain. This means every vertex is covered by the algorithm, so we consider all the same events that segment intersection does.

Then our algorithm is correct since it is equivalent to line segment intersection so it finds any intersection (if there is one). □

Thus our algorithm will correctly determine if there is an intersection. Then there are 2 cases. First case is it finds an intersection $\rightarrow$ it returns not simple. Or it doesn't find an intersection, so it returns simple. Thus our algorithm is correct.

Problem 2

A friend of yours from the civil engineering department wants to analyze whether a dangerous portion of a river will flood. He presents you with the following (admittedly rather unrealistic) model of the river. The portion of the river of interest is modeled as an x-monotone polygon P that is bounded between two vertical lines at $x = x^-$ and $x = x^+$ (see Figure). The river is bounded on its left and right ends by two vertical line segments of lengths w^- and w^+, respectively. Inside the polygon are some number of disjoint x-monotone polygons that represent islands in the river. Let n denote the total number of vertices, including both the outer banks of the river and the islands.

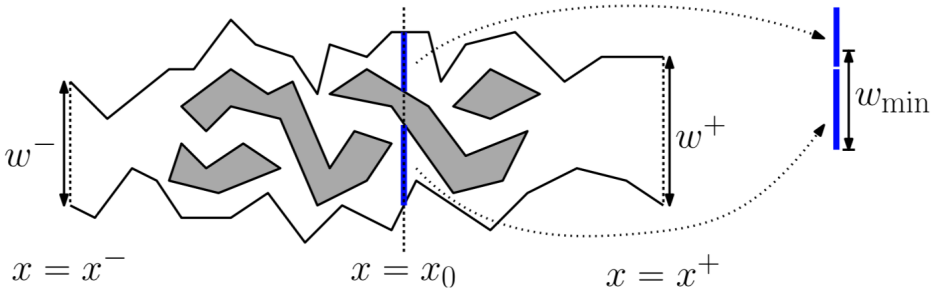


Figure 2: Problem 2: River

Your friend tells you that the in order to avoid a flood, the width of the river (not counting islands) at every vertical cut must be at least some minimum value w_{min}. For example, in the figure, the sum of the two blue vertical segments at $x = x_0$ must be at least w_{min} in order to avoid a flood. Given the polygon P and the value w_{min}, present an $O(n \log n)$ time algorithm that determines whether the river will flood, that is, whether there is a vertical cut whose total width is smaller than w_{min}. If it will flood, your algorithm should output the value x_0 of the bottleneck, that is, the location where the sum of vertical lengths (excluding islands) is the smallest.

1. Hint 1: There is an (uncountably) infinite number of possible vertical cuts to consider. Prove that it suffices to check the width at a discrete set of locations, whose number is $O(n)$.
2. Hint 2: There is a bit of a trick to updating the vertical widths (excluding the islands). For partial credit, explain how to do it under the assumption that the sweep line can only intersect a constant number of islands at any time. (For example, in the figure, the sweep line never hits more than two islands at a time.) For full credit, explain how to do it even if the number of islands hit by the sweep line at any time could be as high as $\Omega(n)$.

As a note, this solution is one of my favorites. I really struggled with it, but at the end the

width update function is one of the coolest methods I have used, I thought that this was quite fun!

So we first create a list of the event points. An event point stores an x location, and a pointer to a vertex. We create one event point per vertex. Each event point will also have the slope of the 'left' edge and the slope of the 'right' edge. One of these are 0 if the vertex is leftmost or rightmost in polygon (e.left = 0 if e is leftmost point). Then we have a sign value (s) which is -1 or 1 if the vertex is on upper hull or lower hull respectively. (One small technicality that the river bank is labeled with opposite sign (ie e.s of a vertex on upper bank of river is $+1$ instead of -1))

We then sort the event points by x coordinate.

On the sweepline we will store 2 variables: a, b that will define our width function.

We iterate over the events. For each event we set $a = f(e)$, and $b = b + e.s(e.right - e.left)$. Our function f is evaluated as $f(e) = a + b(e.x - e_{-1}.x)$ where $(e.x - e_{-1}.x)$ denotes the distance between this point and the previous event point.

We then have a global variable storing the minimum value of a.

Algorithm 2 Flood!!!

```
1: function FLOOD(B, P[])
2:     initialize events
3:     sort events by x
4:     min ← +∞
5:     a ← w⁻, b ← 0
6:     for e ∈ events do
7:         a ← f(e)
8:         min ← a if a < min
9:         b ← b + e.s(e.right − e.left)
10:    end for
11:    return min
12: end function
```

Runtime: Initialize events takes $O(n)$ since it is done with a single linear pass through events and constant time operations. Sorting takes $O(n \log n)$. The internal operations of the for loop are $O(1)$. It runs over each event so there are $O(n)$ operations in the for loop.

Thus in total: $O(n \log n)$

Correctness: This hinges on the correctness that our function f correctly evaluates the width at a point.

Proof. By Induction.

Base case is at the leftmost point. We initialize $a = w^-$ and $b = 0$, so $f = w^-$ at the leftmost point. We then show that it correctly computes the width at the next point. We assume that there are no islands at the beginning, so it just needs to compute the width of the channel. The rate of change in channel width is the slope of upper bank - slope of lower bank Thus we have in our algorithm: $b \leftarrow 0 + (1)(u.right - 0) + (-1)(l.right - 0) = u.right - l.right$ where u is the event of upper bank, and l is lower bank. This is the same as what we described intuitively above. Therefore the base case holds and the function describes the width at the next event point.

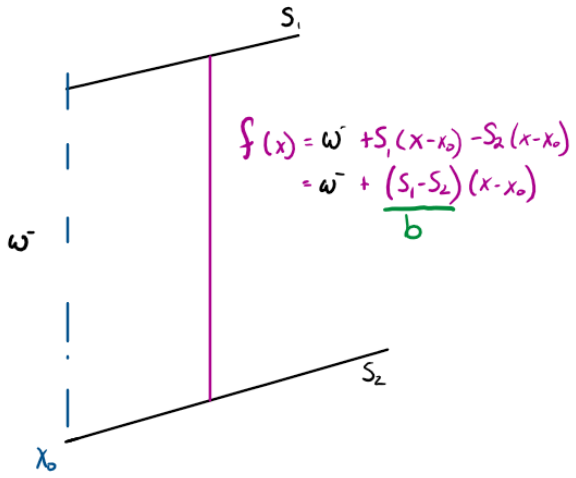


Figure 3: Initialization

Inductive step: assume that the function correctly computes the width up to this event point. Show that it correctly computes at the next event point.

We have two cases: the first case is that this event point is not the start of an island. The second case is the event is the start of an island.

Case 1: So we examine how the rate of change changes. It is not changing anywhere else except at the event point. So we need to analyze the change at the point. It is no longer changing by the rate of the left side so we subtract $e.s * e.left$, and now if it is on the top of a polygon (island), it is changing by $(-1) * e.right$ since a positive slope means the channel is contracting. And on the bottom it is now changing by $(1) * e.right$. This is just the sign $e.s$, so we add $e.s * e.right$, or $e.s * e.right - e.s * e.left = e.s(e.right - e.left)$. Thus our rate of change update is correct and it correctly measures the width at the next event point.

Case 2: It is the beginning of an island. Then we just need to subtract off the wedge that is formed by the start of the island. We split it up into 2 events u, l where u is the upper slope and l is the lower slope. So $u.right > 0$ and $l.right < 0$. So we want to add $l.right$ and

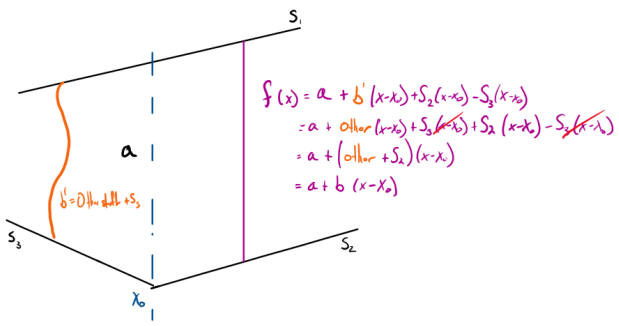


Figure 4: Case 1 changing the slope

subtract $u.right$. So $b \leftarrow b - u.right + l.right = b + (-1)(u.right - 0) + (1)(l.right - 0) = b + u.s(u.right - u.left) + l.s(l.right - l.left)$. Since addition is associative we can do these events separately. Thus, our rate of change of the width is still correct and we correctly evaluate the width at the next point.

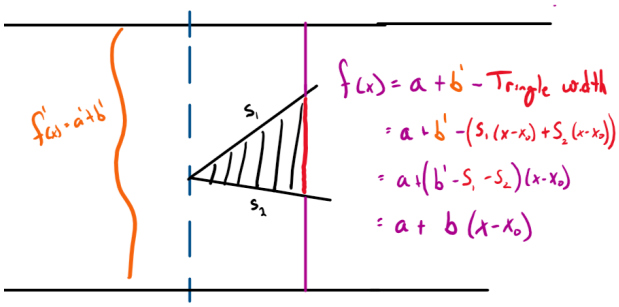


Figure 5: Case 2 changing the slope

So in both cases the rate of change is correct. Then the width at the next point is as follows. width at this point + rate of change * change in x. Well that is $f(next\,e) = a + b * (next\,e_x - e_x)$. Which must be the width at the next point, otherwise the change in width is not described by $b * (next\,e_x - e_x)$, a contradiction since b correctly describes the rate of change of width. □

So now that we have shown we can use our linear function to compute the width. This function is a piecewise continuous function. Then showing that we only need to evaluate the width at $O(n)$ points is fairly trivial.

Proof.

The max or minimum of a function occur at points where the derivative changes (either 0 or DNE). Our function f is piecewise linear, where each piecewise component is linear. If $b = 0$ then it is constant along the interval $[e.x, e_{+1}.x]$ and we can check the value at one of the endpoints. If $b \neq 0$, then there is no max or min in $(e.x, e_{+1}.x)$, and we check the point where

the derivative changes, which is at one of the endpoints.

Thus there cannot be a max or min within $(e.x, e_{+1}.x)$, and we only need to check endpoints. □

Therefore, we only need to check the width at $O(n)$ points, and our function f correctly evaluates the width at each point. So our algorithm will find the minimum width.

Problem 3

1. (7 points) Describe and analyze an algorithm that computes the convex hull of a set of n points in the plane using randomized incremental construction in expected $O(n \log n)$ time. For this problem you are welcome to find an algorithm and its analysis on the web, but please cite where you found it, describe it concisely in your own words, and make the analysis very concise. Where does the log-factor come from?
2. (3 points) Give an example of a set of points in the plane, and a particular input order, that causes the convex hull algorithm to run in $O(n^2)$ when the points are added in this particular order. Make sure it is clear how your example generalizes to arbitrary values of n.

1.

Algorithm `RandomIncrementalCH`

1. Construct the convex hull CH of P_1, P_2, P_3 in clockwise order, stored in a doubly-linked list.
2. Compute a point C inside the convex hull (e.g., the centroid $(P_1 + P_2 + P_3)/3$).
3. Randomly permute the remaining points, and call the new order $P_4, P_5, \ldots, P_n$.
4. For each $P_4, \ldots, P_n$, compute the edge of CH intersected by ray $\overrightarrow{CP_i}$, and associate this edge with P_i.
5. For $i = 4, \ldots, n$:
   (a) Retrieve the associated edge e of P_i, which is visible from P_i.
   (b) Compute the intersection of $\overrightarrow{CP}$ and e; call this point Q. If length CQ is greater than length CP, then P is inside CH, so do nothing and `continue` onto the next iteration.
   (c) Run `BuildTent`(CH, P, e).
   (d) For each deleted edge, reassign the future points associated with that edge to whichever of $\overline{LP}$ and $\overline{PR}$ that intersects ray $\overrightarrow{CP_i}$.

(a) Randomized Convex Hull algorithm

Subroutine `BuildTent`(CH, P, e)

1. Starting from e, go counterclockwise along CH until the next edge to be visited is no longer visible from P. Let L be the left endpoint of the last visible edge.
2. Do the same in the clockwise direction starting from e. Let R be the right endpoint of the last visible edge.
3. Remove all visited edges in both directions, and add the edges $\overline{LP}$ and $\overline{PR}$ in their place in the doubly-linked list.

(b) Subroutine 'build tent' that detects what edges to replace

Figure 6: Randomized convex hull algorithm taken from Lecture 15 of 15-750 Graduate Algorithms at Carnegie Mellon University

Above is the description copied from Carnegie Mellon University lecture notes.

Alg We start by initializing the convex hull to be 3 random points from the point set. They then find some point in the interior of the convex hull: the centroid $C = \frac{P_1+P_2+P_3}{3}$. (the middle of the convex hull).

Then considering a random permutation of the points, they compute an edge that is 'visible' to P_i. An edge is associated with P_i if ray C, P_i intersects the edge. Then, we build the convex hull:

For each point, get the edge that is visible to it. Then check if the point is within the convex hull. Let Q be the point where the ray C, P_i intersects the visible edge. The point is inside iff $dist(C, P_i) < dist(C, Q)$. If P_i is inside, then we don't need to do anything: move on.

If P_i is outside, then we run BuildTent, to compute the left and right vertices. These are the vertices that are tangent to P_i. They then delete the edges between L, R and replace them with L, P_i and P_i, R. Then the points that were associated with deleted edges get re-assigned to either L, P_i or P_i, R based on the ray intersection from the initialization.

Runtime: Step 1, takes $O(1)$ since we make a convex hull of fixed size 3. Step 2, takes $O(1)$ since we compute the centroid. Step 3, takes $O(n)$ since we shuffle all the points. Step 4, takes $O(n)$, this is a little bit less trivial. But we know that we can check line intersections in $O(1)$. We then consider an edge in the convex hull, we can iterate through the entire list of points and check if the ray intersects the line segment in constant time. So this takes $O(n)$. We then only have 3 edges to consider, so $O(n)$ in total. So the entire initialization routine goes $O(n)$

Then we use backwards analysis. We remove a random point P_i from the convex hull, and analyze the cost of doing so. When we remove a point from the CH we must pay a cost of 'undoing' the BuildTent operation. So we pay a cost related to the number of points whose ray intersects these two edges. For each point q outside the CH, ray C, q intersects one edge, which is defined by 2 vertices. So the probability that we pay the cost of q by removing P_i is the probability P_i is one of these 2 vertices: or $2/|CH|$. There are at most $n - i$ vertices on the CH (every vertex that we haven't removed) and i points outside the CH. So summing over all i we get: $\sum_{i=0}^{n} i \cdot 2/(n-i)$. We do change of variables $j = n - i$ (intuitively j is number of vertices on the CH)

$$\sum_{i=0}^{n} i \cdot 2/(n-i) = \sum_{j=0}^{n} (n-j) \cdot 2/j \leq \sum_{j=0}^{n} (n) \cdot 2/j = 2n \cdot \sum_{j=0}^{n} 1/j = 2n\Theta(\ln n) = O(n \log n)$$

The log factor comes from the harmonic series. But physically it comes from the probability that we have to update what edge a point is associated with.

2.

So we get $O(n^2)$ if we have to update what edge a point is associated with every single time for every point.

Consider the points $(0,0),(-1,-1)$ then $p_n = (n, 1/n)$ for $n \geq 1$. If our initial 3 points are $(0,0),(-1,-1),(1,1)$ then every point is associated with edge $[(1,1),(-1,1)]$. Then when we update every point p_j is then associated with $[(n,1/n),(-1,1)]$ for $j > n$, and all the points $j \leq n$ are inside the convex hull.

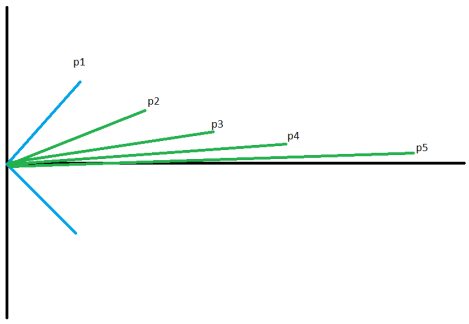


Figure 7: Showing the points and rays resulting in $O(n^2)$

Problem 4

Consider the following instance of the trapezoidal map point location data structure. The left side shows the map, and the right side shows the corresponding DAG. Describe the resulting trapezoidal map and DAG after segment xy has been added.

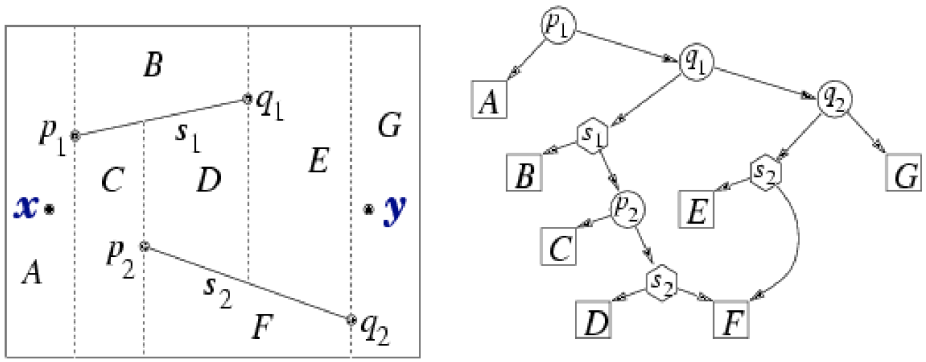
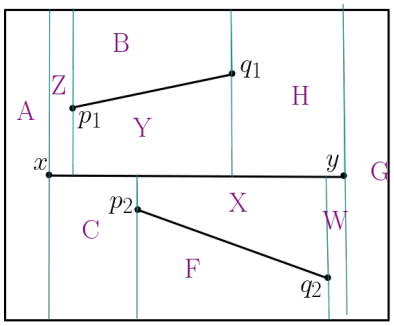
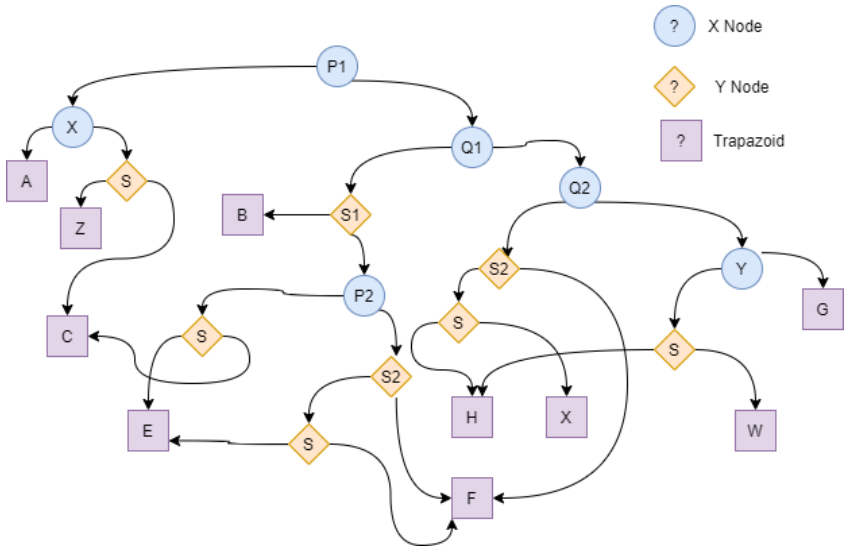


Figure 8: Problem 4: Trapezoid Map



(a) Trap map after adding segment xy



(b) DAG after adding line segment xy

Figure 9: DAG and Tree resulting from adding segment xy

So the DAG became a bit of a tangled mess. I apologize.

Essentially we just add vertical lines from x and y. Then we trim the lines that we intersect with the horizontal line segment. Then in order to build the tree. We take the original map, and apply the 3 cases to it, although we only need cases 1,3. We place these in where the trapazoids A, C, D, E, and G were. Because we intersect them they get modified in some way.

Problem 5

Consider the following algorithm:

```
FindMax(A,n){
    // Finds maximum in set A of n numbers
    if(n==1) return the single number in A
    else {
        x = extract random element from A // in constant time; x is removed from
        A
        y = FindMax(A,n-1)
        if(x<=y) return y;
        else
        Compare x with all remaining elements in A and return the maximum
    }
}
```

1. (4 points) Argue that this algorithm is correct, and give its worst-case runtime. (The runtime is proportional to the number of comparisons made.)
2. (6 points) Compute the expected runtime of this algorithm. (Hint: Introduce an indicator random variable for executing the else branch in the i-th step, and use backwards analysis to simplify the analysis.)

1.

Correctness:

Proof. By Induction.

Base case $|A| = 1$, therefore the only element in A is the max and it is trivially true.

Inductive step: assume we have a max for the $k-1$ elements, show we find max for k elements. Let $A_k = A$ when $|A| = k$, and $A_{k-1} = A_k \setminus \{x\}$. Clearly A_{k-1} is what is covered in the next recursive layer down. So by the inductive assumption y is the max of A_{k-1}.

If $y \geq x$ then y is the max of A_k. We partition A_k into A_{k-1} and x. By inductive assumption y is max of A_{k-1}, and we have that $x \leq y$, so y is the max of both partitions, therefore is the max of A_k.

Otherwise, we search through each element of A_k and find the max. This clearly finds the maximum, because if it didn't one element would not have been considered.

Thus the inductive step holds. □

At worst case, we always choose $x = \max A_k$. Then we always enter the else statement. Since we remove elements one at a time until $|A| = 1$, our recursion depth is $n - 1$. If we enter the else statement we check all the elements of k.

So our recurrence relation is $T(k) = T(k-1) + k$. So therefore at worst case it is $O(n^2)$

2.

Consider iteration i of the algorithm. So $|A| = i$. Let R_i be the random event that the else branch is executed Let P_i be the probability taken over all permutations of A of R_i.

There is only one number $x \in A$ such that $x > y$. Since $y = \max(A \setminus \{x\})$. Thus the probability P_i is the probability that we select this number. So $P_i = 1/i$. When we enter the else statement we check every element which takes $O(i)$ time.

This means that the expected amount of work from the else statement over all iterations is: the probability of entering the else statement * amount of work if we enter it. $E = \sum_{i=1}^{n} O(i) \cdot P_i = \sum_{i=1}^{n} O(i)/i = \sum_{i=1}^{n} c = cn = O(n)$ So the expected amount of work from the else statement is $O(n)$

Thus the running time is $n + O(n)$ where the first n comes from the recursion depth. We consider the recursion depth separately since the second n factor got 'summed out'. So the overall running time is $O(n)$