CSCI 534: Homework 03

Nathan Stouffer – Collaborated with Elliott Pryor

Problem 1

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

Answer: First let's present some definitions and assumptions. A polygonal chain is a collection of line segments joined end to end. A polygonal chain is simple if no segments intersection. For general position assumptions, we assume that no two segment end points have the same x coordinate.

To solve this problem, we ended up coming up with a solution very similar to the line segment intersection sweep line algorithm. A quick description of our algorithm would say that we use the line segment intersection algorithm except that we only need to sort $O(k \log k)$ event points and we know the number of segments in the sweep line data structure is $O(k)$. Every vertex will be an event point so there are n events in total.

The algorithm will begin by breaking P into x-monotone chains that are contiguous in P. We claim that there are $O(k)$ x-monotone chains in P. For proof, note that between any two "neighboring" local maxes (the next local max traveling along P), there can be only one turn to the right. If not, the two local maxes would not be neighbors. Since there is only one right turn, we split the chain at the turn and we are left two x-monotone chains to the neighboring local maxes. This produces $O(k)$ chains. There are some edge cases with local maxes near the end of the chain, but they are constant in number and can be wrapped up into the big-O.

With this decomposition $X_1, X_2, ..., X_j$ of P in mind, we can label our three types of event points. Type I: segment beginning an x-montone chain X_i. Type II: vertex ending an x-monotone chain X_i. Type III: other segments in the chain. Note that each event of type I and II has a subsequent event along the x-monotone chain. We will use a heap implementation of a min-priority queue (with the x coordinate as the key) for our event queue. For our sweep line data structure, we will use an ordered dictionary that gives us operations in $O(\log m)$ time (where m is the number of items in the dictionary).

1. Break P into $O(k)$ x-monotone chains (contiguous in P)
2. Insert each Type I into the event queue
3. Pop the event e from the queue.
   If Type I or III: insert segment into sweep line state, test for intersections with the segment's neighbors on the sweep line. If we find an intersection, return NOT SIMPLE. Then insert the subsequent event (along the x-monotone chain) into the event queue.
   If Type II: delete the segment from the sweep line state, test for insersctions of the new neighbors on the sweep line.
4. If the event queue is nonempty, return to line 3.
5. return SIMPLE

We feel that much of the correctness of this algorithm is justified by the correctness of the line segment intersection sweep line algorithm. The only part that needs justification is that not all vertex end points need to be in the event queue at the same time. This is true because each vertex is part of an x-monotone chain. If we are considering some vertex x_i, the next vertex x_{i+1} certainly cannot affect the sweep line at the current state for the segment it corresponds to is entirely to the right of the area we are considering.

Let's justify the run time as $O(n \log k)$. We have already shown that we decompose P into $O(k)$ x-monotone chains. Since the number of Type I events is the same as the number of x-monotone chains, step 2 inserts $O(k)$ events into the queue. A heap with $O(k)$ items provides operations in $O(\log k)$ time, thus step 2 takes $O(k \log k)$ time. We claim step 3 takes $O(\log k)$ time. Editing the event queue takes $O(\log k)$ time and the number of items in the sweep line state is $O(k)$ for we can only have one segment from each monotone chain. Then operations on the sweep line state take $O(\log k)$ time, thus giving us $O(\log k)$ time for step 3. Step 4 has us repeat step 3 as many times as we have events. However we already noted that there are n events (giving $O(n \log k)$ for steps 3 and 4 combined). Then step 5 is constant. This sums to give a total run time of $O(n \log k)$.

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.

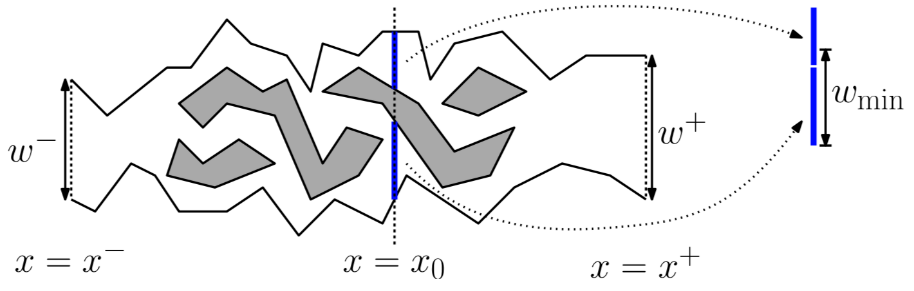

Figure 1: Problem 2: River

Your friend tells you that 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.

Answer: This problem ended up have a very clean solution which we liked a lot! We settled on a sweep line algorithm. All we care about is whether a polygon (island) is on the sweep line or not, so we can ignore its neighbors. This means we do not have to maintain an ordered dictionary as our sweep line data structure, instead we can use an array with constant time access!

Before giving a description of the algorithm, let's discuss some assumptions. First, assume that each polygon is stored in an array with the vertices ordered counter clockwise. As far as general position goes, we assume that no two vertices have the same x-coordinate (with the exception of the left and right points of the river). The event points will be vertices so this means all events will have different x-coordinates.

Now we describe the algorithm. For input, our algorithm will take in the array representing the shore S and the arrays representing each island I_i together with the values w^-, w^+, and w_{min}. As noted before, the vertices become the event points. We have two types of event points: one that contains a vertex of S and another which contains a vertex of some I_i. We sort the events according to their x-coordinates and store the result in a queue. There will not be any more event points so a regular queue will suffice. Now we will compute the river width each event point (we will provide a proof that considering the vertices is sufficient).

One way to compute the width at $x^\star$ is to subtract the width of each island on the sweep line at $x^\star$ from the distance between the shoreline S. The problem with this is that we can construct input with $\Omega(n)$ islands at some $x^\star$. One such input is approximately $n/3$ triangles stacked vertically. This would result in an $O(n^2)$ algorithm.

To bypass this, note that the width of the river is a piecewise-linear continuous function. All we have to do is find the appropriate line segments and perform $O(1)$ operations at each event point. With this model, we can reduce the amount of information we store on the sweep line. All we need is the function representing the current line segment. We don't even need to store what islands the sweep line is intersecting because we can determine the event type just by inspecting which polygon a point comes from!

Now let's figure out the a function for the first line segment. We assumed no two event points have the same x-coordinate, so there exists some interval A for which the left side of the river matches Figure 2. Figure 2 also depicts the function representing the first line segment. The top line has slope s_1 and the bottom line has slope s_2. Certainly we begin with width w^- and then for any x beyond x_0 we can add s_1x and subtract s_2x to obtain the width at x. The function $w_A(x)$ accurately reports the width on the inteval $[x_0, x^\star]$ where $x^\star$ is the x-coordinate of the next event point. Thus we have a linear function to represent the first line segment.

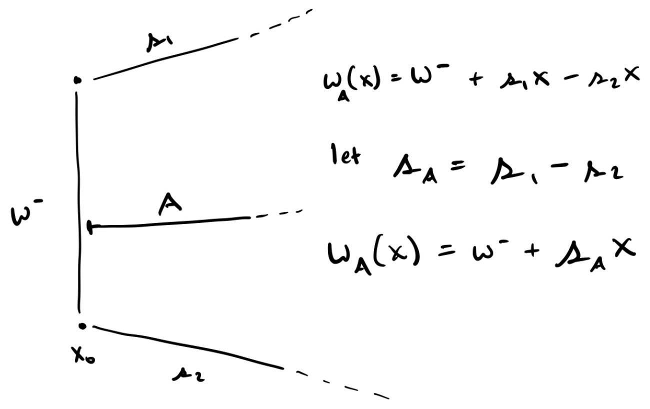

Figure 2: Base case depiction

At $x^\star$, the x-coordinate of the next event point, we can enter one of four cases. In case 1, the event contains a vertex from S the shore line. Case 1 in Figure 3 depicts how $w_A(x)$ transforms into $w_B(x)$ if the vertex is part of the upper shoreline. An analagous operation occurs for the lower shoreline. The remaining 3 cases show what happens if the event contains a vertex from an island I_i. We can determine which case to enter by inspecting the neigboring points of the vertex in I_i. Case 2 covers what to do when an island begins and Case 4 shows what happens when an island ends. Case 3 depicts how $w_B(x)$ is constructed when the vertex is on the upper part of an island and there is analogous operation when the vertex is on the lower part of an island.

As we consider each event point, we can keep track of the smallest width found so far and the x-coordinate that is occurs at. In keeping with our piecewise-linear continuous model for the width of the river let's store them in variables called min_x for the x-coordinate and min_y for the width

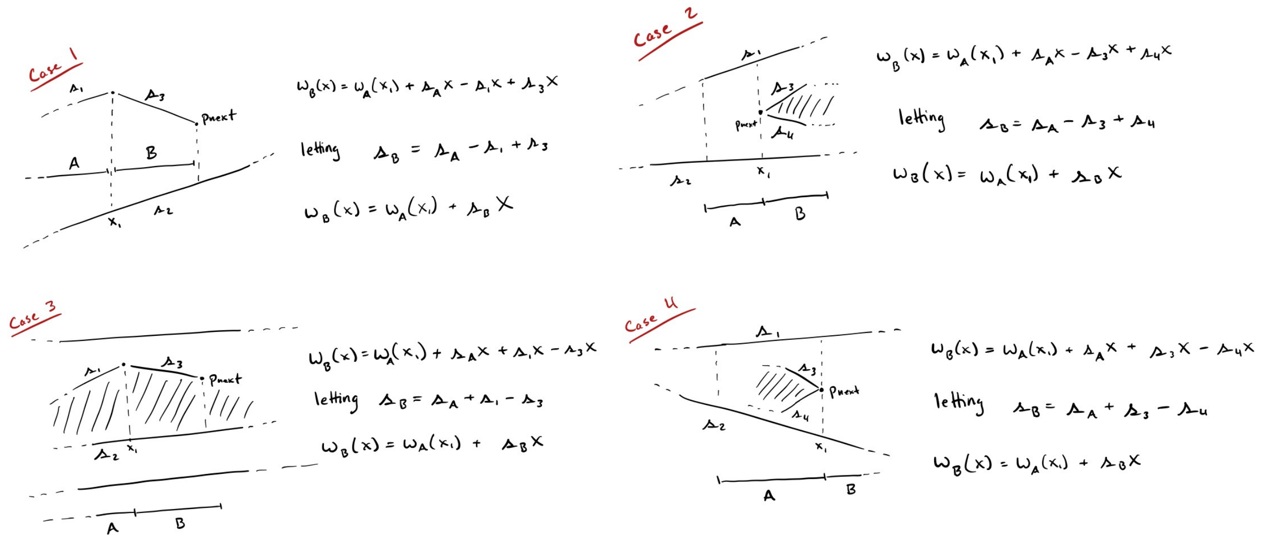
Case 1
s1
s3
pnext
A
B
x1
s2
ω_B(x) = ω_A(x1) + s_A x − s1 x + s3 x
letting s_B = s_A − s1 + s3
ω_B(x) = ω_A(x1) + s_B x
Case 2
s1
s3
pnext
s4
s2
x1
A
B
ω_B(x) = ω_A(x1) + s_A x − s3 x + s4 x
letting s_B = s_A − s3 + s4
ω_B(x) = ω_A(x1) + s_B x
Case 3
s1
s3
pnext
s2
x1
A
B
ω_B(x) = ω_A(x1) + s_A x + s1 x − s3 x
letting s_B = s_A + s1 − s3
ω_B(x) = ω_A(x1) + s_B x
Case 4
s1
s3
pnext
s4
s2
x1
A
B
ω_B(x) = ω_A(x1) + s_A x + s3 x − s4 x
letting s_B = s_A + s3 − s4
ω_B(x) = ω_A(x1) + s_B x

Figure 3: Cases for updating the sweep line

of the river at min_x. When the sweep terminates, if $min_y < w_{min}$ return min_x, otherwise return NO FLOOD.

Before discussing correctness, let's look at run time. There are two bigs steps in this algorithm. The first is sorting n points which takes $O(n \log n)$ time. The next big step is to process the n events. In every case, we perform a constant number of operations so the run time for this step is $n * O(1) = O(n)$. Therefore the entire run time is $O(n \log n) + O(n) = O(n \log n)$ as desired.

The correctness of the updates to the slope is proved in Figure 3. However, we do need to show that the minimum must lie at one of the vertices. Let's continue with the piecewise-linear continuous representation of the width. Then vertices are the only places where the slope changes. Suppose the minimum occurs at $x^\star$ a value that is not a vertex. Then the slope does not change at $x^\star$. But if the slope is not changing, we can travel in a direction (either positive or negative) that decreases the value of the width, a contradiction. The previous statement excludles segments with slope 0. If this is the case then the slope is 0 for the whole segment and we will detect it at the vertex. Thus we only need to concsider vertices as event points.

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.

Answer:

1. The paper that I looked at can be found at `https://dl.acm.org/doi/pdf/10.1145/355759.355766?casa_token=rLrtzRrif4kAAAAA%3AU9g3Xil8GMLkRECw4Whv-ZvLYPSrHypLUS_VL75ycErIk3aTX-4m` It was originally published in 1977 by William F. Eddy of Carnegie-Mellon University. It works by finding the farthest left and right points of the data set and drawing a line between them. Then it processes each point to find the furthest point on each side of the line. WLOG, select a side. The three points (the line endpoints and the furthest point from the line) form a triangle. Any points inside the triangle can be eliminated and then the algorithm is called recursively on the remaining points on the right or left of the furthest point. Additionally, the points form an incremental hull for the data seen so far. The log factor arrives because a random distribution of points will remove half of the points in expectation.

2. This algorithm has a loophole in that if the points are distributed along a circle it will take $O(n^2)$ time for any order of points. This is because no points will lie in any inscribed triangle of the circle. I am just realizing that this algorithm is not really a randomized incremental construction but it is close to the deadline so I'm sticking with what I've got!

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.

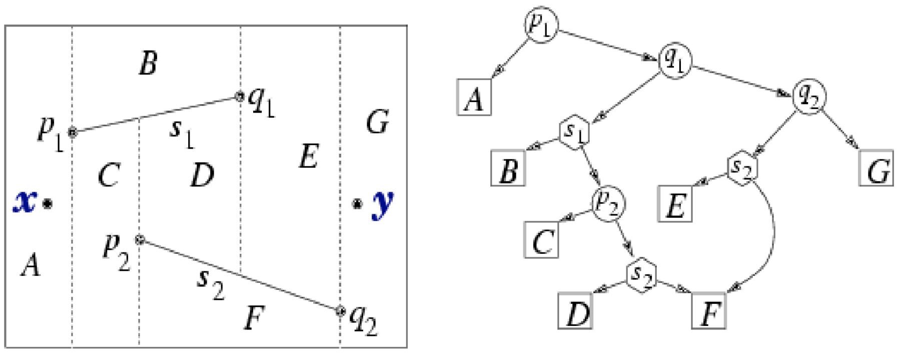

Figure 4: Problem 4: Trapezoid Map

Answer: After we add segment xy, we must add the changes displayed in Figure 5. Traps A and G lie in case 1 of updating the trap map and traps C, D, and E are in case 3 of updating the trap map.

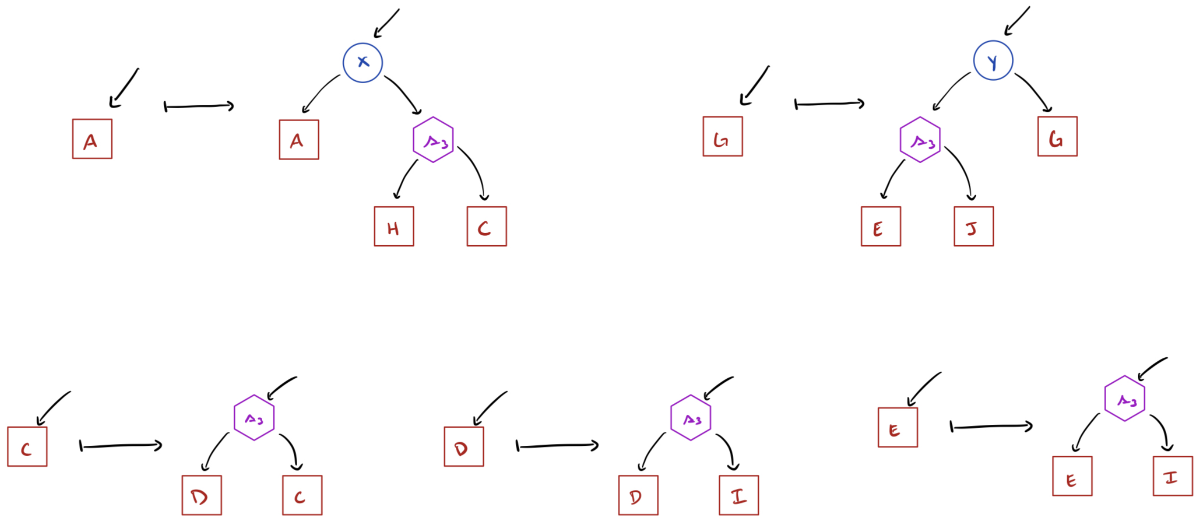

Figure 5: Steps of updating the DAG

Figure 6 shows the updated trap map. Notice that vertical border (originating from a point) extends beyond any line segment. The DAG that represents the trap map must reflect this. The final DAG to represent the trap map after segment xy has been inserted is displayed in Figure 7.

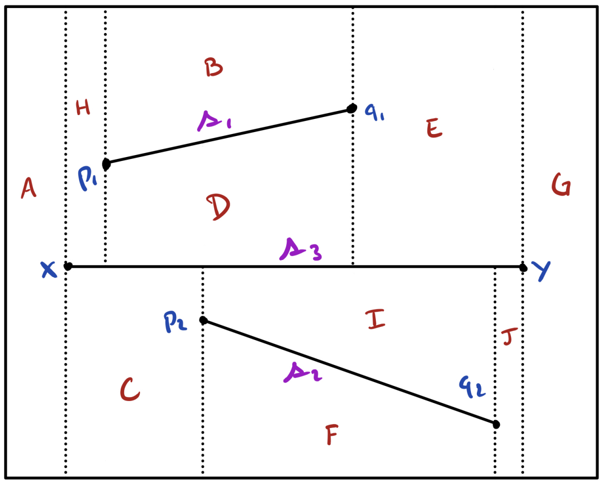

Figure 6: Depicting the updated Trap Map

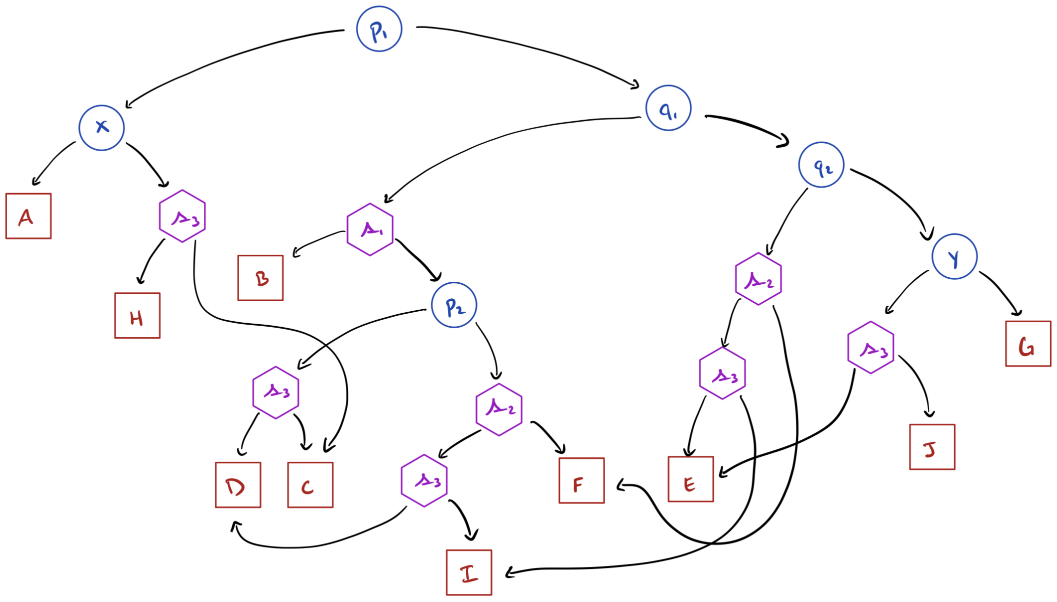

Figure 7: Depicting the updated DAG

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.)

Answer:

1. Let's begin by proving the correctness of this algorithm. We assume that when FindMax(A, n) is called for a set $A \subset \mathbb{R}$ where $|A| = n$. We will proceed by induction on n. For the base case, take $n = 1$. Here, the set A is a singleton and the maximum of a singleton is the element. The algorithm FindMax returns the only element when $n = 1$ so it must return the max. Thus FindMax is correct when $n = 1$.

   Now suppose, for some natural number $k \in \mathbb{N}$, that FindMax(A, k) correctly returns the maximum value of the set A. We will prove that FindMax$(A \cup \{a\}, k + 1)$ correctly returns the maximum of the set $A \cup \{a\}$ for $a \in \mathbb{R}$. Since k is a natural number, $k + 1 > 1$ so we must enter the else statement. Then we extract some x from $A \cup \{a\}$ and store whatever is returned from FindMax$(A \cup \{a\} \setminus \{x\}, k)$ in y. Since $A \cup \{a\} \setminus \{x\}$ has cardinality k, we know (by supposition) that y is the maximum value of $A \cup \{a\} \setminus \{x\}$. Then if $y \geq x$, we have $y = \max(A \cup \{a\})$ which matches the behavior of the algorithm. Otherwise, the algorithm compares each remaining element with x and returns the max. This is certainly correct for if $x > y$ then x is the maximum value of $A \cup \{a\}$ and that line will return x.

   Thus the algorithm is proved correct for any n by induction. Let's now discuss the worst case run time. In the worst case, we have to compare x with the remaining elements in every call to FindMax. We should note that the algorithm consists of making exactly n calls to FindMax. Thus to compute the run time, we don't have to build a recurrence, we can just count the number of comparisons made on an aribitrary call to FindMax and then sum those values for

$n = \{1, 2, ..., n\}$. Let $T(n)$ denote the number of comparisons made in the (worst case) call to FindMax(A, n). If $n = 1$, then $T(1) = 1$. If $n > 1$, then $T(n) = 1 + 1 + (n - 2) = n$. Thus the total number of comparisons is $\sum_{k=1}^{n} k$ which is known to equal $n(n + 1)/2 = O(n^2)$. Since the algorithm runs in time proportional to the number of comparisons, the run time is $O(n^2)$.

2. Now let's compute the expected run time over all permutations of the elements of A. On the call FindMax(A, i), we must compare x with all the remaining elements with probabilty $1/i$. Let k_i denote the number of comparisons made when we call FindMax(A, i). Also let

$$\delta_i = \begin{cases} 1 & \text{if the bad else statement runs on the } i^{th} \text{ call} \\ 0 & \text{otherwise} \end{cases}$$

   Then $k_i = 1 = O(i)$ when $i = 1$ and $k_i = 1 + 1 + \delta_i(i - 2) = 2 + \delta_i(i - 2) \leq 2 + i - 2 = i = O(i)$ otherwise. The expected value is $E[k_i] = 1/i * O(i) = O(1)$ so we expect to have a constant number of comparisons for an arbitrary value of i. Since we make n calls to FindMax, this gives an expected run time of $n * O(1) = O(n)$.