1. Given set $S$ of 2D points, find the maxima. (Let $n = 15$)

b) Incremental Alg:
First, we sort the points by $x$ coordinate

We then consider each point in turn, pushing $p_i$ to a stack $A$ and popping any points in $A$ that $p_i$ dominates.

We maintain the following invariant on the points in $A$:
- Points in $A$ are in order of increasing $x$.
- Points in $A$ are in order of decreasing $y$.

Each point $p_i$ is added to the stack once, and removed from the stack at most once. Therefore this part of the alg. takes $\Theta(n)$.

c) Divide-and-Conquer Alg:

Given a set $S$, return maxima in order of increasing $x$.

Base case: $n = 1$, the one point is maximal, so return it.

General case: split the $n$ points into 2 disjoint subsets of size $n/2$ and recurse on each subset.

Then merge the resulting sets of maxima $M_1$ and $M_2$ using a stack as in (b). This takes $O(n)$ since the points in $M_1$ and $M_2$ are already sorted.

$T(n) = 2T(n/2) + O(n) = \Theta(n \log n)$

We do not need to sort the points, and our total time is $O(n \log n)$, so we don't gain anything by not sorting. Moreover, our merge takes $O(n)$, so extending this to a dynamic algorithm would be slow (dynamic add/remove in $O(n)$ is trivial, since the points are already sorted).
Looking for a div-and-conq. alg. with $O(\log u)$ merge...

Another Divide-and-Conquer Alg.

First, sort the points in $S$ by $x$.

Then div-and-conq:

Base case: 1 point $\rightarrow$ maximal; return it

General case: split $S$ into two halves $S_L$, $S_R$ by $x$.

Recurse on $S_L$ and $S_R$, which gives us their maximal sets $M_L$ and $M_R$.

Merge $M_L$ and $M_R$ like this:

Let $p$ be the highest point in $M_R$. Binary search for $p,y$ in $M_L$, and discard all points in $M_L$ below that position.

This merge takes $O(\log n)$ assuming a data structure that supports search, split, join in $O(\log n)$.

The div-and-conq. takes $T(n) = 2T(n/2) + O(\log n) = O(n)$.

The entire alg. still takes $\Theta(n \log n)$ because of the sorting, but the $O(\log n)$ merge gives us hope for a dynamic alg.
Suppose that as we divide-and-conquer in (c), we keep the partial results at each step in a balanced BST.

Then when we need to add/remove a point, we modify the tree from leaf to root, re-merging and re-balancing as necessary.

One merge takes $O(\log n)$ and there are $\log n$ levels ⇒ One add/remove takes $O(\log^2 n)$.

The problem here is that we can't really do merges in $O(\log n)$ if we keep copies of the partial results at each node... To fix this, instead of keeping a copy of merge $(M_L, M_R)$ at each node, just store the index $i$ in $M_L$ where $M_R$ is appended.

becomes

becomes
Marriage-before-Conquest Algorithm.

We would like an algorithm in \( O(n \log m) \) where \( m = \# \) of maximal points.

We can modify the divide-and-conquer algorithm like this:

1. Given set \( S \) of \( n \) points (unsorted), find median \( x \)-value and split by it into \( S_L \), \( S_R \).
2. Find point \( p \in S_R \) with maximum \( y \)-coordinate.
3. Discard all points in \( S_L \) that are below \( p \).
4. Compute maximal sets of \( S_L \) and \( S_R \) recursively.
5. Concatenate the two resulting lists, since we now know that no point in \( S_R \) dominates any point in \( S_L \).

Our recurrence is just like for M-B-C convex hull:

\[
T(n, m) = T\left(\frac{n}{2}, m_L\right) + T\left(\frac{n}{2}, m_R\right) + \Theta(n) \quad \text{with} \quad m = m_L + m_R
\]

so the algorithm takes \( O(n \log m) \) time.

This is ensured by the fact that we remove all points in \( S_L \) that are dominated by a point in \( S_R \).
Idea: whenever we encounter a vertex, we want to know which other vertex we can connect it to. Keep a list of segments sorted by y coord, and for each segment store the "goto" vertex for the area above it.

Add an imaginary segment \((x_{min}, y_{min}) - (x_{max} + 1, y_{min} - 1)\) at the bottom.

\[
\frac{4}{4}
\]

[Diagram showing sweep direction]

We sort the vertices by x and sweep from left to right, stopping at every vertex.

We keep a balanced BST of segments that the sweep line intersects, ordered by y. Each segment keeps a goto vertex, s.t. if a vertex is found above this segment, it can be connected to that goto vertex without intersecting anything else.

When we see a vertex \(v\), there are four cases:

1) \(v\) "begins" a triangle (both its edges to the right)
   - Add \(s_1\) to \(T\) with goto \(v\)
   - Add \(s_2\) to \(T\) with goto NULL
   - Connect \(v\) to \(w\), the goto of \(s_{below}\)
   - Update goto of \(s_{below}\) to \(v\)
   (\(s_{below}\) is the segment in \(T\) immediately below \(v\), we find it using binary search)

2) \(v\) "below-continues" a triangle (one edge to the left, one to the right, \(\angle(s_2, s_1) \leq 180^\circ\))
   - Remove \(s_1\) from \(T\)
   - Add \(s_2\) to \(T\) with goto NULL
   - Update goto of \(s_{below}\) to \(v\)
V "top-continues" a triangle (one edge to the left, one to the right, and \( \angle (s_1, s_2) > 180^\circ \))
remove \( s_1 \) from \( T \)
add \( s_2 \) to \( T \) and goto \( V \)
(the goto of \( S_{below} \) will already be null)

V "ends" a triangle (both edges to the left)
remove \( s_1 \) and \( s_2 \) from \( T \)
update goto of \( S_{below} \) to \( V \).

The algo. maintains the following sweep invariant:

Whenever we encounter vertex \( V \) in case 1, it is safe to connect \( V \) to the goto of \( S_{below} \). Moreover, all triangles to the left of the sweep line have already been connected by bridges.

Time: \( \Theta(n \log n) \) to sort + \( \Theta(n \log n) \) to update \( T \)

after seeing each of the \( n \) vertices
\( \Rightarrow \) total \( \Theta(n \log n) \),
a) determine if \( P \) is convex.

For each edge \( V_i \rightarrow V_{i+1} \), translate \( V_i \) to the origin.

These edges must go in counterclockwise order. This can be checked in \( \Theta(n) \), and is equivalent to going around the polygon and making sure we only do left-hand turns.

b) determine if \( P \) is monotone

If there exists a direction of monotonicity, then the origin-truncated vectors will stay on one side of their direction on the upper chain, and another side on the lower chain. In other words, going from \( \overrightarrow{V_{i-1}V_i} \) to \( \overrightarrow{V_iV_{i+1}} \) will cross this direction of monotonicity exactly twice.
Alg. to determine if polygon is monotone:

The idea is to traverse the origin-translated vectors in order, counting how many times we crossed each "wedge" between two vectors. If we traversed the whole polygon and we can find a wedge that was only crossed once, and some wedge opposite it by 180° has also only been crossed once, then the polygon is monotone.

Naively this could take up to $O(n^2)$, but we can bring it down to $O(n)$ by keeping a dynamic circular linked list of wedges labeled with "none", "one", or "multiple" crossings. We start with a single wedge $0-360°$ labeled "none", and update the list as we traverse the polygon. When a wedge becomes labeled "multiple", we can delete it from the list. Each wedge can then pay for its insertion and its removal, making the alg. $O(n)$. Very nice!
Given: \( P \), (simple poly) and \( T \), - its triangulation, including pockets \( P_2 \) (convex poly - \( n \) vertices)

Want: intersection of \( P \) and \( P_2 \) in \( O(m+n) \) time.

First, extract \( C_1 \), the convex hull of \( P \). This can be done in \( O(m) \) by traversing \( T \). Then compute \( C_1 \cap P_2 \). Since these are convex polygons, their intersection can be found in \( O(m+n) \). Then replace \( P_2 \) with \( C_1 \cap P_2 \). This guarantees that any point in \( P_2 \) is within the area that \( T \) describes.

(If \( C_1 \cap P_2 = \emptyset \) then \( P_1 \cap P_2 = \emptyset \) and we quit early.)

Now: pick some vertex \( v \in P_2 \). In \( O(n) \) find which triangle in \( T \) \( v \) belongs to. Traverse \( P_2 \) from \( v \) in CCW order, visiting all triangles in \( T \) that the boundary of \( P_2 \) passes through. \( T \) is a DCEL structure, so this traversal takes \( O(n) \). Every time we cross from a pocket triangle to a triangle in \( P \), we save the intersection as an in-vertex. Similarly, from \( P \) to pocket, save as out-vertex.

Now: traverse \( P \) in CCW order starting at every out-vertex, until we hit an in-vertex. The paths on \( P_2 \) from in-vertices to out-vertices, together with the paths on \( P \), from out-vertices to in-vertices define all areas of \( P \cap P_2 \). Total time \( O(n+m) \). +1
In the solution I submitted, I glossed over how to compute the intersection of two convex polygons, because I only thought about the simple case:

In fact we could get something like, in which case the simple sweep-line algorithm to find the chains of the intersection, which I had in mind, does not work. Instead, as discussed in class today, find the intersection of $C_1$ and $P_2$ in $O(n \log n)$ by taking a point inside $P_2$, sorting all points w.r.t. it by slope, and consider the sectors that they form.

The sorting takes $O(n \log n)$ since $C_1$ and $P_2$ are given in CCW vertex order. We spend $O(1)$ time per sector and there are $O(n + m)$ sectors, so the total time for this step is $O(n + m)$.

Once we have $C_1 \cap P_2$, traverse the DCEL as I described before.