(a) Truthfully write the phrase “I have read and understand the course policies.”

Solution: I have read and understand the course policies.

(b) Describe and analyze an $O(n \log n)$ time algorithm to compute/output Pareto($P$).

Solution: We’ll begin by sorting the points of $P$ from right-to-left. Then, we add the rightmost point to Pareto($P$). We’ll then iterate over the rest in right-to-left order. Every time we encounter a point with a higher $y$-coordinate than one we’ve seen before, we’ll add it to Pareto($P$). Indeed, every point $p_i$ we fail to include is dominated by a point further right, which we’ve passed before, and higher up, since $p_i$ does not have the highest $y$-coordinate we’ve seen so far. Every point we include has nothing higher up $\text{and}$ further to the right. The bottleneck step is sorting which takes $O(n \log n)$ time.

(Alternatively, we could sort the points left-to-right as $p_1, \ldots, p_n$ and do an incremental construction. We maintain a stack $S$ containing the current Pareto set in left-to-right order. For each point $p_i$, we repeatedly pop until the top of the stack has a higher $y$-coordinate than $p_i$ and then add $p_i$ to the stack.)

(c) Let $h = |\text{Pareto}(P)|$. Describe and analyze an $O(nh)$ time algorithm to compute/output Pareto($P$).

Solution: We’ll search for the rightmost point and add it to Pareto($P$). We then iteratively find the remaining points. Suppose we added point $p_i$ in the previous iteration. In the current iteration, we search for the rightmost point $p_j$ for which $x_j < x_i$ and $y_j > y_i$ and add $p_j$ to Pareto($P$). The algorithm terminates when it fails to find any suitable point in some iteration. This algorithm successfully adds Pareto points in right-to-left order, because each added point $p_j$ is inductively higher than points to its right, and each skipped point is dominated by the point found in the previous iteration.

Each of the $h$ iterations takes $O(n)$ time, so the running time is $O(nh)$ as desired.

(d) Describe and analyze an $O(n \log h)$ time algorithm to compute/output Pareto($P$). For simplicity, you may assume that the value $h$ is known in advance.

Solution: As in Chan’s algorithm, we will arbitrarily partition $P$ into $\lceil n/h \rceil$ subsets $P_1, \ldots, P_s$, each of size at most $h$. We then compute the Pareto set Pareto($P_i$) for each subset $P_i$ in $\lceil n/h \rceil \cdot O(h \log h) = O(n \log h)$ time total by running the algorithm from part (a) on each subset separately. We will assume each set Pareto($P$) is stored in right-to-left order in an array, which is easy to guarantee since the algorithm sorts them in that order anyway. As a consequence, the points are stored in bottom-up order as well. Finally, observe that any point $p \in P_i \setminus \text{Pareto}(P_i)$ is dominated by a member of Pareto($P_i$) and is therefore not in Pareto($P_i$) either.
Next, we run a variant of the algorithm from part (b). We search for the rightmost point of $P$ and add it to $\text{Pareto}(P)$. We then iteratively find the remaining points. Suppose we added point $p_i$ in the previous iteration. Instead of performing a full $O(n)$ time search for the next point to add, we search each of the subsets $\text{Pareto}(P_\ell)$ for their rightmost point $p_j$ for which $x_j < x_i$ and $y_j > y_i$, and return the rightmost of each of these points. Each $\text{Pareto}(P_\ell)$ can be searched in $O(\log h)$ time by a simple binary search since the points are stored in both right-to-left and bottom-up order. The total time per iteration is $[n/h] \cdot O(\log h)$. There are $h$ iterations, so the total running time is $O(n \log h)$ as desired.

\[\blacksquare\]
Let \( \mathcal{C} = \{C_1, \ldots, C_n\} \) be a set of \( n \) circles in \( \mathbb{R}^2 \) where each circle \( C_i \) is given as its center point \( q_i = (x_i, y_i) \) and radius \( r_i > 0 \). Describe and analyze an \( O(n \log n) \) time algorithm to determine whether or not any pair of circles intersect.

**Solution:** As suggested by the hint, we will run a plane sweep algorithm where we move a vertical line from left to right. Observe that a vertical line intersects a circle in two positions: once on its \( x \)-monotone top half and once on its bottom half. Therefore, we will maintain for the sweep line status the set of circle \textit{halves} intersecting the sweep line in top to bottom order. Both halves of a circle intersect the sweep line at the same point when the sweep line touches the circle's leftmost or rightmost point, so we'll make sure our sweep line data structure always breaks ties so the upper half of a circle lies above the bottom half of the same circle.

Define the leftmost and rightmost points on a circle \( C_i \) as its \textit{endpoints}. Suppose there is an intersection, and consider the leftmost intersection. From immediately left of the intersection to immediately right of the rightmost endpoint left of the intersection, no halves on the sweep line status exchange places in top to bottom order. Therefore, we can take a similar approach to the line segment intersection problem from lecture. We will use the circle endpoints as the event points. At every event point, we update the sweep line status by removing a circle's halves if we're about to move right of the circle or adding a circle's halves if we're about to start sweeping over it. We will check the new pairs of adjacent circle halves for intersections. If an intersection is detected, the algorithm reports an intersection exists and terminates. Otherwise, we move on to the next event point.

We can use an ordered dictionary to update the sweep line status and check for intersections between newly adjacent circle halves in \( O(\log n) \) time per event. All the event points are known in advance, so instead of a dynamic event queue like we saw in class, we'll just sort the circle endpoints left-to-right in \( O(n \log n) \) time and loop over them in that order. The algorithm sees at most \( 2n \) events, so the total running time is \( O(n \log n) \).
Given the polygon $P$ and the value $w_{\text{min}}$, 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 $w_{\text{min}}$.

**Solution:** We will again use a plane sweep algorithm where we move a vertical line from left to right. Observe that as the sweep line moves between two consecutive vertices in left-to-right order (these vertices are not necessarily on the same edge or polygon) the width of the sweep line’s cut monotonically increases or decreases. In particular, the width of the cut is a linear function of $x$, because its rate of change is equal to the sum of some polygon segment slopes and their opposites. Therefore, we’ll let the event points for our plane sweep be the endpoints of the input polygons and we’ll only check cut widths at the event points.

Our algorithm will maintain a changing linear function $w(x) = mx + b$ such that at each event point, evaluating $w(x_0)$ gives the width of the cut at $x = x_0$. Initially, $w(x) = w^\text{in}$. Now, consider when we reach event point $v = (v_x, v_y)$. Set $w_v \leftarrow w(v_x)$. If $w_v < w_{\text{min}}$, then we report that a flood will occur and terminate the algorithm. Otherwise, we update $w(x) = mx + b$ as follows. If there is an incident edge $e$ of $v$ going left with the river above it, we increase $m$ by the slope of $e$. If there is an incident edge $e$ of $v$ going left with the river below it, we decrease $m$ by the slope of $e$. If there is an incident edge $e$ of $v$ going right with the river above it, we decrease $m$ by the slope of $e$. If there is an incident edge $e$ of $v$ going right with the river below it, we increase $m$ by the slope of $e$. Finally, we set $b \leftarrow w_v - mv_x$ so that $w_v = mv_x + b$ is still true.

All the event points are known in advance, so instead of a dynamic event queue like we saw in class, we’ll just sort the edge endpoints left-to-right in $O(n \log n)$ time and loop over them in that order. The algorithm sees $n$ events and takes constant time per event, so the total running time is $O(n \log n)$. ■