Let \( P = \{ p_1, \ldots, p_n \} \) be a set of \( n \) points in the plane \((\mathbb{R}^2)\), and let \( p_i = (x_i, y_i) \) for each \( i \). The
Pareto set \( \text{Pareto}(P) \subseteq P \) is the subset of \( P \) containing each point \( p_i \) such that there exists no point \( p_j \in P \) with \( i \neq j \) such that both \( x_j \geq x_i \) and \( y_j \geq y_i \).

(In each solution, we assume no pair of points share \( x \)- or \( y \)-coordinates.)

(a) Describe and analyze an \( O(n \log n) \) time algorithm to compute/output \( \text{Pareto}(P) \).

Solution: We’ll begin by sorting the points of \( P \) from right-to-left. Then, we add the rightmost point to \( \text{Pareto}(P) \). 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 \( \text{Pareto}(P) \). 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.

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

Solution: We’ll search for the rightmost point and add it to \( \text{Pareto}(P) \). 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 \( \text{Pareto}(P) \). 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.

(c) Describe and analyze an \( O(n \log h) \) time algorithm to compute/output \( \text{Pareto}(P) \). 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 \( \text{Pareto}(P_i) \) 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 \( \text{Pareto}(P) \) 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 \( \text{Pareto}(P_i) \) and is therefore not in \( \text{Pareto}(P_i) \) 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 \( \text{Pareto}(P_j) \) 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 \( \text{Pareto}(P_j) \) 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 \( \lceil n/h \rceil \cdot O(\log h) \). There are \( h \) iterations, so the total running time is \( O(n \log h) \) as desired.
Let $C = \{c_1, \ldots, c_n\}$ be a set of $n$ circles in $\mathbb{R}^2$ where each circle $c_i$ is given as its center point $a_i = (x_i, y_i)$ and radius $r_i > 0$. Note that some circles may be completely nested in one-another without intersecting. 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 halves intersecting the sweep line in top to bottom order.

Let the leftmost and rightmost points on a circle $c_i$ be its 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 $O(n \log n)$. ■
Let $P$ be an $x$-monotone polygon that is bounded between two vertical lines $x = x^-$ and $x = x^+$. Inside $P$ are some number of disjoint $x$-monotone polygons. Let $n$ denote the total number of vertices on $P$ and the other polygons. Describe and analyze an $O(n \log n)$ time algorithm that returns the length of the shortest vertical line segment intersecting either two non-adjacent edges of $P$ or both an edge of $P$ and an edge of one of the internal polygons.

**Solution:** We will again use a plane sweep algorithm where we move a vertical line from left to right. Observe that given two line segments, the shortest vertical segment between them occurs at one of their endpoints. Therefore, we'll let the event points for our plane sweep be the endpoints of the input polygons. Second, at any moment, the shortest vertical segment as described in the question but also lying on the sweep line must occur between either two non-adjacent edges of $P$ or between the lowest or highest edge outside $P$ and the lower or higher edge of $P$, respectively. To keep track of these lowest and highest edges, we'll again store the segments intersected by the sweep line in top to bottom order.

Our algorithm will maintain a running minimum observed distance as the sweep line moves. At each event point, we add or remove segments exactly like we did for the line segment intersection problem. Then, we check the distance between the non-adjacent edges of $P$ intersecting the sweep line and the other pairs as described above, updating our minimum if we find a shorter vertical segment than we found before.

We can use an ordered dictionary again to update the sweep line status and find the most extreme non-$P$ edges 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 edge endpoints left-to-right in $O(n \log n)$ time and loop over them in that order. The algorithm sees $n$ events, so the total running time is $O(n \log n)$. ■