Meta-rubric. Grading in this class will use the following crude rubric, for problems worth 5 points:

- 5 points = perfectly correct and perfectly clear
- 4 points = correct, but with minor flaws or omissions
- 3 points = mostly correct
- 2 points = good progress, but significant details are missing or incorrect
- 1 point = good-faith effort but incorrect
- 0 points = no submission (or no good-faith effort)

Partial credit problems or subproblems worth more or less than 5 points will be scaled and rounded to half-integers. I reserve the right to deviate from this rubric for specific issues.

1. A *circular array* is a standard array that is being used to store a circular sequence of values. Every element $A[i]$ in a circular array $A[0..n-1]$ has a successor $A[(i+1) \bmod n]$ and a predecessor $A[(i-1) \bmod n]$.

   A *local minimum* in a circular array is any element that is strictly smaller than both its predecessor and its successor. Similarly, a *local maximum* is any element that is strictly greater than both its predecessor and its successor.

   Finally, a circular array A is *strictly bitonic* if no element of A is equal to its successor, and the smallest element of A is the only local minimum in A.

   (a) Prove that the largest element of a strictly bitonic array is also its only local maximum.

   Solution: Fix a circular array $A[0..n-1]$ where no element is equal to its successor. Define a new circular array $\Delta[0..n-1]$ by setting $\Delta[i] = A[i+1] - A[i]$ for every index i. (All index arithmetic is modulo n.) The following properties are immediate:

   - $\Delta[i] \neq 0$ for every index i.
   - $\Delta[i-1] < 0$ and $\Delta[i] > 0$ if and only if $A[i]$ is a local minimum of A.
   - $\Delta[i-1] > 0$ and $\Delta[i] < 0$ if and only if $A[i]$ is a local maximum of A.

   Now suppose A is bitonic. Then there is exactly one index i such that $\Delta[i-1] < 0$ and $\Delta[i] > 0$. In other words, as we increase i, the sign of $\Delta[i]$ switches from negative to positive once. So there is also exactly one switch from positive to negative; that is, there is a unique index j such that $\Delta[j-1] > 0$ and $\Delta[j] < 0$. The corresponding entry $A[j]$ is the unique local maximum of A. ■

   Rubric: 2 points

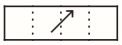
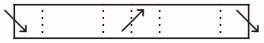
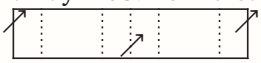
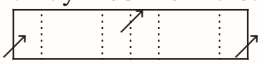
   (b) Describe an algorithm to find the smallest element in a strictly bitonic array $A[0..n-1]$ in $O(\log n)$ time.

   Solution: We use a variant of binary search. The key observation is that any subsequence of a bitonic sequence is also bitonic. Thus, if we determine that the minimum element is in (say) the left half of the input array, we can find it by recursively searching the left half.

(The following algorithm implicitly assumes that $A[i] \neq A[j]$ for all $i \neq j$; if necessary, we can enforce this assumption by breaking ties by index—replace every comparison $A[i] < A[j]$ with $(A[i] < A[j]$ or $(A[i] = A[j]$ and $i < j))$.)

```
BitonicMinimum(A[0..n-1]):
  if n ≤ 10
      find min(A) by brute force
  else
      m ← ⌊n/2⌋
      if A[m-1] < A[m]
          if A[n-1] > A[0] or A[n-1] > A[m-1]
              return BitonicMinimum(A[0..m-1])
          else
              return BitonicMinimum(A[m..n-1])
      else
          if A[n-1] < A[0] or A[n-1] < A[m-1]
              return BitonicMinimum(A[m..n-1])
          else
              return BitonicMinimum(A[0..m-1])
```

The inductive correctness proof follows the various cases of the algorithm.

- The algorithm is trivially correct when $n \leq 10$.
- Otherwise, if $A[m-1] < A[m]$, there are two subcases to consider. 
  - If $A[n-1] > A[0]$, then the minimum element of A must be in the left half of the array (and the maximum must be in the right half). 
  - If $A[n-1] < A[0]$, there are two subsubcases to consider.
    - If the minimum is in the left half of the array, then the right half of the array must be increasing; in particular, $A[m-1] < A[m] < A[n-1]$. 
    - If the minimum is in the right half of the array, then the left half of the array must be increasing. In particular, $A[m-1] > A[0] > A[n-1]$. 

      In both subcases, the induction hypothesis implies that our algorithm returns the minimum element of A.
- The subcases for $A[m-1] > A[m]$ are mirror images of the previous subcases.

Finally, our algorithm runs in $T(n) = O(1) + T(n/2) = \boldsymbol{O(\log n)}$ ***time***. Specifically, the algorithm performs at most $3\log_2 n + O(1)$ comparisons. ■

Solution: We use a variant of binary search. Intuitively, we sample three evenly-spaced entries of the input array, discard the interval between the two highest samples, and recursively search the remaining interval.

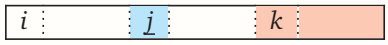
(The following algorithm implicitly assumes that $A[i] \neq A[j]$ for all $i \neq j$; if necessary, we can enforce this assumption by breaking ties by index—replace every comparison $A[i] < A[j]$ with $(A[i] < A[j]$ or $(A[i] = A[j]$ and $i < j))$.)

```
BitonicMinimum(A[0..n-1]):
  i ← 0          ⟨⟨starting index of current subarray⟩⟩
  ℓ ← n          ⟨⟨length of current subarray⟩⟩
  while ℓ > 100
      j ← i + ⌊ℓ/3⌋ mod n
      k ← j + ⌊ℓ/3⌋ mod n
      if min{A[i],A[j],A[k]} = A[k]
          ⟨⟨discard first third⟩⟩
          ℓ ← (i + ℓ − j) mod n
          i ← j
      else if min{A[i],A[j],A[k]} = A[i]
          ⟨⟨discard middle third — only in first iteration!⟩⟩
          ℓ ← (j − k) mod n
          i ← k
      else if min{A[i],A[j],A[k]} = A[j]
          ⟨⟨discard last third⟩⟩
          ℓ ← (k − i) mod n
  find min(A[i..(i + ℓ − 1) mod n]) by brute force
```

Pictorially, the three cases of the main loop look like this. The three sample values are indicated by indices i, j, k; the smallest of these three samples is underlined and shaded blue. In each case, we discard the subarray shaded red; in particular, we discard exactly one of the samples.

- 

- 

- 

In the first and third cases, the remaining elements clearly define a contiguous subarray, but the second case *appears* to be more problematic. Why can't we discard the middle third, and then discard the middle third again, leaving us with a disconnected set of elements to search?

The key insight is that the second case can only happen in the first iteration of the main loop. After the first iteration, the remaining elements are contiguous in all three cases. Consider any later iteration that begins with a contiguous subarray $A[i..i+\ell-1]$. Let $m \approx (i + \lfloor \ell/2 \rfloor) \bmod n$ be the index of the smallest sample from the previous iteration. Then $A[i]$ is the largest element of the subarray $A[i..m]$; in particular, $A[i] > A[j]$. Thus, $\min\{A[i],A[j],A[k]\}$ cannot be equal to $A[i]$; only the first and third cases are possible.

Our algorithm runs in $T(n) = O(1) + T(2n/3) = \boldsymbol{O(\log n)}$ ***time***. Specifically, because every iteration after the first requires only one comparison, this algorithm performs at most $\log_{3/2} n + O(1) < 1.7096 \log_2 n + O(1)$ comparisons, which is almost a factor of 2 faster than the previous algorithm.

If we modify this algorithm to split the current subarray unevenly, into intervals whose lengths have the ratio $\phi : 1 : \phi$, where $\phi = (\sqrt{5}+1)/2 \approx 1.618$, the resulting algorithm uses at most $\log_\phi n + O(1) < 1.4405 \log_2 + O(1)$ comparisons. Ignoring the $O(1)$ term, this is actually optimal! ■

Rubric: 4 points. These are neither the only correct algorithms, nor the only correct description of these algorithms.

(c) Suppose we are given a convex polygon P (represented as usual as a circular array of vertex coordinates) and a line ℓ. Describe an algorithm that determines in $O(\log n)$ time whether ℓ intersects P. For full credit:

- If P and ℓ intersect, your algorithm should return the intersection points.
- Otherwise, your algorithm should return the closest point in P to ℓ.

Solution: Let $P[i] = (x[i], y[i])$ denote the coordinates of the ith vertex of P. Suppose the line ℓ is represented as the coefficients (a, b) of its equation $y = ax - b$.

Implicitly define (but do not actually construct!) an array $A[0..n-1]$ by setting $A[i] = y[i] - (a \cdot x[i] - b)$ for every index i. Because P is convex, this array is bitonic. Thus, we can find the indices i^+ and i^- of the minimum and maximum elements of A in $O(\log n)$ time using our algorithm in part (b). Let ℓ^+ be the line parallel to ℓ that contains the vertex $P[i^+]$, and let ℓ^- be the line parallel to ℓ that contains the vertex $P[i^-]$. Then the entire polygon P lies between ℓ^+ and ℓ^-.

Now there are three cases to consider:

- If $y[i^+] < a \cdot x[i^+] - b$, then P lies entirely below ℓ, and $P[i^+]$ is the closest point in P to ℓ.
- If $y[i^-] > a \cdot x[i^-] - b$, then P lies entirely above ℓ, and $P[i^-]$ is the closest point in P to ℓ.
- Otherwise, we can conclude that ℓ intersects P. The points $P[i^+]$ and $P[i^-]$ split P into two polygonal chains that are monotone with respect to ℓ. (That is, the subarray $A[i^-..i^+]$ is monotonically increasing, and the subarray $A[i^+..i^-]$ is monotonically decreasing, where as usual, indices wrap around.) In each chain, we can find the edge that intersects ℓ using a standard binary search in $O(\log n)$ time. Finally, we can compute the intersection points of ℓ with these two edges in $O(1)$ time.

In all cases, the algorithm runs in $O(\log n)$ time. ■

Rubric: 4 points

2. Let P be a set of points in the plane. A point $p \in P$ is *Pareto-optimal* if no other point in P is both above and to the right of p. The Pareto-optimal points can be connected by horizontal and vertical lines into the *staircase* of P, with a Pareto-optimal point at the top right corner of every step.

   Now suppose you are given a set P of n points in the plane, with distinct x- and y-coordinates.

   (a) Suppose the points in P are given in order from left to right, that is, sorted by increasing x-coordinate. Describe an algorithm to compute the the staircase of P in $O(n)$ time.

Solution (⟵): The solutions for both parts describe algorithms to output the indices of the Pareto-optimal points in P, in order from left to right. It's easy to construct an explicit description of the staircase (as a polygonal line) in linear time from this representation, if necessary.

Scan the points from right to left, and record any point that has the largest y-coordinate seen so far into an array. Finally, reverse the output array (so that the output points are sorted from left to right).

```
ParetoSubset(P[1..n]):
  maxy ← −∞
  h ← 0
  for i ← n down to 1
      if P[i].y > maxy
          h ← h + 1
          Stair[h] ← i
          maxy ← P[i].y
  return Reverse(Stair[1..h])
```

■

Solution (⟶): Scan the points from left to right, maintaining a stack containing the Pareto-optima among the points scanned so far. Finally, transcribe the stack into the output array in reverse order (so that the output points are sorted from left to right).

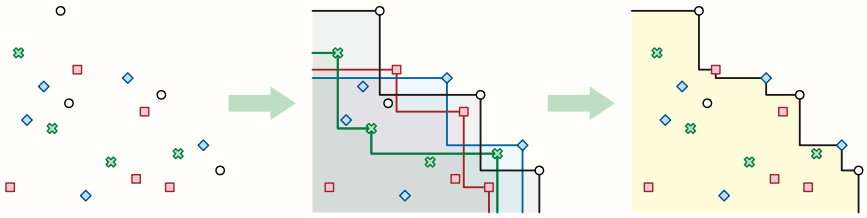
```
ParetoSubset(P[1..n]):
  S ← new empty stack
  for i ← 1 to n
      while S is non-empty and P[i].y > P[S.Top()]
          S.Pop()
      S.Push(i)
  h ← S.Size()
  for i ← h down to 1
      Stair[i] ← S.Pop()
  return Stair[1..h]
```

■

Rubric: 3 points. These are not the only correct solutions!

(b) Part (a) implies that when the the input array P is *not* sorted, we can still compute the staircase of P in $O(n \log n)$ time. Describe an algorithm to compute the the staircase of P in $O(n \log h)$ time, where h is the number of Pareto-optimal points in P. (For partial credit, describe an algorithm that runs in $O(nh)$ time.)

Solution: Assume that we know (a polynomial estimate of) the output size h; we can enforce this assumption using a doubly-exponential search, just like Chan's algorithm. Following Chan, we proceed in two phases.



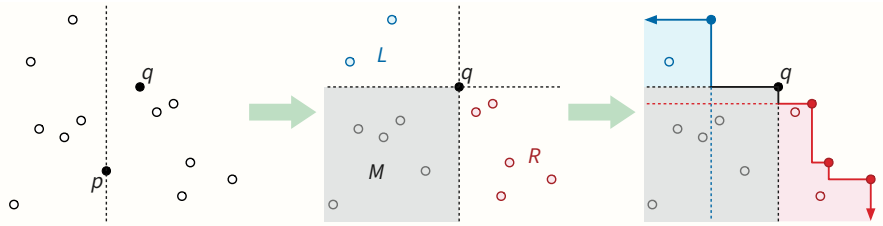
In the first phase, we arbitrarily partition P into $O(n/h)$ subsets of size $O(h)$, and we compute the staircases of those subsets in $O(h \log h)$ time, by sorting and then invoking the algorithm from part (b). The overall running time of this phase is $O(n/h) \cdot O(h \log h) = O(n \log h)$.

In the second phase, we identify the Pareto-optimal points in order from left to right, starting with the point with highest y-coordinate (which is also a point on one of the mini-staircases). Suppose p is a Pareto-optimal point in P. For each index i, let q_i be the highest point to the right of p in the ith mini-staircase. Assuming each mini-staircase is represented as an array S_i of points sorted by x-coordinates, we can identify q_i using a simple binary search in S_i in $O(\log h)$ time. The successor of p on the staircase of P is the highest of the q_i's; we can identify this point by brute force in $O(n/h)$ time after $O(n/h)$ binary searches. The total time spent in each iteration is $O((n/h) \log h)$, and by definition there are h iterations, so the overall running time of this phase is also $O(n \log h)$. ■

Solution: I'll describe a second divide-and-conquer algorithm that loosely mirrors the earliest $O(n \log h)$-time convex hull algorithm, described by Kirkpatrick and Seidel in 1986.

Assume $n \geq 3$, since otherwise the problem can be trivially solved in $O(1)$ time. Let p be point in P with median x-coordinate, and let q be the highest point to the right of p. This point q is Pareto-optimal. Moreover, we can identify q in linear time, using a linear-time selection algorithm to identify p, followed by a brute-force scan.

Now we split P into three subsets: L contains all points above and left of q; R contains all points below and right of q; and M contains all points below and left of q. (Again, there are no points above and right of q!) No point in M is Pareto-optimal, so we can ignore M entirely. The Pareto-optimal points in P consist of the Pareto-optimal points in L, followed by q, followed by the Pareto-optimal points in R. Thus, we can compute the staircase of P by recursively computing the staircases of L and R and sewing them together at q.



Every point in R lies to the right of p, so we have $|R| \le n/2$. The definition of q implies that every point in L lies to the left of p, so we also have $|L| \le n/2$.

Let $T(n,h)$ denote the running time of this algorithm as a function of the input size n and the output size h. Ignoring irrelevant floors and ceilings, this function satisfies the recursive upper bound

$$T(n,h) \le T(n/2,h_L) + T(n/2,h_R) + O(n) \qquad \text{where } h_L + h_R = h-1$$

To simplify the analysis, we can upper-bound the recurrence as follows, for some constant α:

$$T(n,h) \le \max_x \big(T(n/2,x) + T(n/2,h-x)\big) + \alpha n$$

I will prove by induction that $T(n,h) \le \alpha n \lg h$. The inductive hypothesis implies

$$\begin{aligned}
T(n,h) &\le \max_x \big((\alpha n/2)\lg x + (\alpha n/2)\lg(h-x)\big) + \alpha n \\
&= (\alpha n/2) \cdot \max_x \lg(x(h-x)) + \alpha n \\
&= (\alpha n/2) \cdot \lg(h^2/4) + \alpha n \qquad (\star) \\
&= (\alpha n/2) \cdot (2\lg h - 2) + \alpha n \\
&= \alpha n \lg h - \alpha n + \alpha n = \alpha n \lg h
\end{aligned}$$

In the third line $(\star)$, I'm using the fact that the function $x(h-x)$ is maximized when $x = h/2$.

The behavior of this two-parameter recurrence is somewhat counterintuitive (at least to me). Every recursive call reduces the *input* size by at least a factor of two, but it does *not* necessarily split the output evenly. The *worst*-case running time happens when the recursion tree perfectly balanced! If the recursion tree is completely unbalanced, the level sums form a decreasing geometric series, and the running time is only $O(n)$. ■

Rubric: 7 points. First solution: 3 for first phase + 4 for second phase. Second solution: 4 for algorithm + 3 for analysis. These are not the only correct solutions!

3. Suppose we are given a set H of horizontal line segments and a set V of vertical line segments. A *crossing point* is any point that lies on both a horizontal segment in H and a vertical segment in V.

   (a) Describe an algorithm that computes the number of crossing points. For full credit, your algorithm should run in $O(n \log n)$ time, regardless of the number of intersections. Assume that the segments H and V are in general position. *[Hint: Use a sweep-line algorithm. What operations does your sweep data structure need to support? Design a data structure that supports those operations!]*

Solution: We use a standard sweep-line algorithm. The sweep dictionary maintains the y-coordinates of intersection between the the current vertical sweep line and the horizontal segments, subject to the following operations:

- INSERT(y): Insert y into the dictionary (at the left endpoint of a horizontal segment).
- DELETE(y): Delete y from the dictionary (at the right endpoint of a horizontal segment).
- COUNTBETWEEN(b, t): Return the number of values y in the dictionary such that $b \le y \le t$ (at a vertical segment with bottom coordinate b and top y-coordinate t).

We can support each of these operations in $O(\log n)$ time using a modified balanced binary search tree. In addition to a search key, each vertex v of the binary search tree maintains the number of nodes in its subtree in a new field $v.size$. We can maintain subtree sizes in $O(1)$ additional time per rotation, so the time for INSERT and DELETE is still $O(\log n)$.

We can implement COUNTBETWEEN in $O(\log n)$ time with two calls to the following function; specifically, COUNTBETWEEN(t, b) = RANK(t) − RANK(b).

```
⟨⟨Return the number of keys less than t⟩⟩
RANK(t):
  v ← root of BST
  count ← 0
  while v ≠ NULL
      if v.key < t
          if v.left = NULL
              count ← count + 1
          else
              count ← count + 1 + v.left.size
          v ← v.right
      else ⟨⟨general position implies v.key > t⟩⟩
          v ← v.left
  return count
```

Finally, the main algorithm performs $O(n)$ dictionary operations—two for each horizontal segment and one for each vertical segment—and therefore runs in $O(n \log n)$ time.

```
CountIntersections(H, V):
  sort all |V| + 2|H| endpoint x-coordinates into an array X      (★)
  count ← 0
  for i ← 1 to |V| + 2|H|
      if X[i] = H[j].left
          Insert(H[j].y)
      else if X[i] = H[j].right
          Delete(H[j].y)
      else if X[i] = V[j].x
          count ← count + CountBetween(V[j].bottom, V[j].top)
  return count
```

■

Rubric: 6 points

(b) Modify your algorithm from part (a) to handle ***arbitrary*** horizontal and vertical segments. Each crossing point should only be counted exactly once. Your modified algorithm should still run in $O(n \log n)$ time.

Solution: The most significant change to the algorithm is a preprocessing phase, where we *merge* intersecting collinear segments. We preprocess the horizontal segments H as follows; the vertical segments are handled similarly.

First, we sort the segments H by y-coordinate, so that all segments on the same horizontal line are clustered together. Then for each horizontal line that contains a segment in H, we compute the union of all segments on that line as described in the next paragraph. The total time to identify and merge *all* overlapping collinear segments in $O(n \log n)$.

Suppose the line $y = a$ contains m segments; these must be contiguous in the sorted array H. First we sort the $2m$ endpoints of these segments from left to right in $O(m \log m)$ time; to break ties at the same x-coordinate, we sort all left endpoints before all right endpoints. Then we set a *depth* counter to 0 and scan the endpoints in sorted order. Whenever we visit a left endpoint, we increment *depth*; if *depth* $= 0$ before the increment, we've found a left endpoint of an output segment. Whenever we visit a right endpoint, we decrement *depth*; if *depth* $= 0$ after the decrement, we've found a right endpoint of an output segment.

We can now run the earlier sweep algorithm on the merged segments with only two modifications, which intuitively treat all segments as through they were slightly longer in both directions. First, in the sorting step (★), we break ties at the same x-coordinate by sorting all left endpoints first, then all vertical segments, then all right endpoints. Second, in our implementation of CountBetween(b, t), we check whether b and t are in the sweep dictionary.

Merging overlapping segments ensures that each intersection point is counted at most once. Careful tie-breaking during the sweep ensures that each intersection point is counted at least once. The the algorithm still tuns in $O(n \log n)$ time. ■

Rubric: 4 points

4. Suppose we are given a set $S = \{s_1, s_2, \ldots, s_n\}$ of disjoint line segments in the plane. Another set $R = \{r_1, r_2, \ldots, r_n\}$ of *horizontal* line segments is called a *rectification* of S if the following properties hold for all indices i and j:

   - The endpoints of s_i and r_i have the same x-coordinates. Thus, if a vertical line ℓ intersects s_i, then ℓ also intersects r_i.
   - Suppose vertical line ℓ intersects both s_i and s_j. Then $\ell \cap s_i$ is above $\ell \cap s_j$ if and only if $\ell \cap r_i$ is above $\ell \cap r_j$.

   Describe and analyze an algorithm to compute a rectification of S. *[Hint: First prove that S must contain at least one segment with no other segments in S directly above it.]*

Solution: Let's start by following the hint. Call a segment $s \in S$ a *candidate* if no other segment is directly above the right endpoint of s. Let s_1 denote the candidate with the leftmost right endpoint. (The segment with the rightmost right endpoint is a candidate, so s_1 must exist!) Let A denote the set of all segments directly above s_1. The right endpoint of every segment in A must be above s_1, because s_1 is a candidate. If A is non-empty, the segment in A with the rightmost right endpoint is another candidate, contradicting the definition of s_1. We conclude that A is empty; no other segment lies directly above s_1.

We can identify s_1 in $O(n \log n)$ time using a standard sweep algorithm. By repeatedly sweeping and deleting one segment, we can index the segments in S as $s_1, s_2, \ldots, s_n$ in $O(n^2 \log n)$ time, so that for all indices $i < j$, segment s_j is *not* directly above segment s_i. We can then rectify S by changing the y-coordinates of each segment s_i to $n - i$.

We can speed up the algorithm by processing all the segments at once, instead of one at a time. Our earlier argument implies by induction that the "directly above" relation is a partial order, but we can also prove this directly. For the sake of argument, suppose $s_0, s_1, \ldots, s_{\ell-1}$ is a *minimal* cycle of segments in S such that s_i is directly above s_{i-1} for every index i. (All index arithmetic is mod ℓ.) Let s_i be the segment in this cycle with the leftmost right endpoint. The right endpoint of s_i is directly above s_{i-1} and directly below s_{i+1}. It follows that s_{i+1} is directly above s_{i-1}, contradicting the minimality of our cycle.

We can construct a trapezoidal decomposition of S using a standard sweep algorithm in $O(n \log n)$ time. Write $s \uparrow s'$ if there is a trapezoid in the decomposition whose ceiling is part of s and whose floor is part of s'. (We can read $s \uparrow s'$ as "s is *immediately* above s'".) If $s \uparrow s'$, then s is directly above s'; it follows that $\uparrow$ is also a partial order. On the other hand, if s is directly above s', then there is a sequence $s \uparrow s_1 \uparrow \cdots \uparrow s_k \uparrow s'$ of segments, each *immediately* above the next, starting with s and ending with s'. It follows that $\uparrow$ has the same transitive closure as "directly above".

We can construct a directed graph G representing the partial order $\uparrow$ in $O(n)$ time. (Vertices correspond to the n segments, and directed edges correspond to the $O(n)$ trapezoids.) Topologically sort G in $O(n)$ time. The resulting linear order is a linear extension of "directly above". Finally, to rectify G, replace the y-coordinates of each segment with its rank in the topological order. The overall algorithm runs in $\boldsymbol{O(n \log n)}$ ***time***. ■

Solution (Palazzi and Snoeyink[ab]): We can define an *aboveness tree* T for S as follows. The vertices of T are the segments of S, plus an additional sentinel segment s_0 above every segment in S. The root of T is s_0. The parent of each segment is the segment in $S \cup \{s_0\}$ immediately above the right endpoint of s. The children of any segment are naturally ordered by the x-coordinates of their right endpoints. We can construct T in $O(n \log n)$ time using a standard sweep algorithm.

Now index the segments in $S \cup \{s_0\}$ as $s_0, s_1, s_2, \ldots, s_n$ by performing a left-to-right preorder traversal of T. (When the traversal reaches each segment s, we first index s and then recursively traverse each subtree of s, in order from left to right.)

Define the *trace* of a segment s to be the directed path that starts at the left endpoint of s and then alternately moves along the current segment to its right endpoint and then directly upward to the next segment. The traces of two segments s_i and s_k cannot cross, but they do merge together at the lowest common ancestor of s_i and s_j in T. Every trace ends at the right endpoint of the sentinel s_0.

Now suppose some segment s_i is directly above another segment s_j. Let s_k be the lowest common ancestor of s_i and s_j. Because traces do not cross, $s_k \neq s_j$. If $s_k = s_i$, then $i < j$ by definition of preorder. Otherwise, s_i and s_j must be in different subtrees of s_k, and the subtree containing s_i must be to the left of the subtree containing s_j, which also implies $i < j$.

We conclude that for all indices $i < j$, segment s_j is *not* directly above segment s_i. Thus, we can rectify S by changing the y-coordinates of each segment s_i to $n - i$. ■

[a] Larry Palazzi and Jack Snoeyink. Counting and reporting red/blue segment intersections. *CVGIP: Graph. Models Image Proces.* 56(4):304–311, 1994.

[b] See also: Sergio Cabello, Yuanxin Liu, Andrea Mantler, and Jack Snoeyink. Testing homotopy for paths in the plane. *Discr. Comput. Geom.* 31:61–81, 2004.

Solution (variants submitted by several students): I'll describe an algorithm to assign a "rectified y-coordinate" $s.Ry$ to each segment $s \in S$. Unlike the previous algorithms, more than one segment may be assigned the same rectified y-coordinate.

We modify the standard Bentley-Ottmann sweep algorithm. To simplify the algorithm, we keep two sentinel segments s^+ and s^- in the sweep dictionary at all times, with $s^+.Ry = 1$ and $s^-.Ry = 0$. Segment s^+ is above every segment in S, and s^- is below every segment in S. Whenever the sweep line reaches the left endpoint of some segment s, we insert s into the sweep dictionary as usual, and then assign $s.Ry$ to ***any*** value strictly between $\textsc{Pred}(s).Ry$ and $\textsc{Succ}(s).Ry$, where $\textsc{Pred}(s)$ and $\textsc{Succ}(s)$ are the predecessor and successor of s in the sweep dictionary. The modified sweep algorithm still runs in $O(n \log n)$ time.

Alternatively, instead of assigning explicit coordinates during the sweep, we can maintain a doubly-linked list RL of segments, initially containing only the lower sentinel segment s^-. Whenever the sweep encounters the left endpoint of some segment s, we insert s into RL immediately after $\textsc{Pred}(s)$. When the sweep is done, we set each rectified y-coordinate $s.Ry$ to the position of s in the list RL; we can assign all n rectified y-coordinates in $O(n)$ time by traversing RL once. Again, the overall algorithm runs in $O(n \log n)$ time.

Now let R be the segments obtained by changing the y-coordinates of each segment $s \in S$ to $s.Ry$. To prove that R is a rectification of S, it suffices to observe that for any position of the vertical sweep line ℓ, the rectified y-coordinates of segments intersecting ℓ are consistent with the order of intersection along ℓ. Thus, segments in R intersect ℓ in the same order as the corresponding segments in S. ■

Rubric: 10 points = 5 for proving that "above" is a partial order + 5 for the algorithm. This is more detail than necessary for full credit.