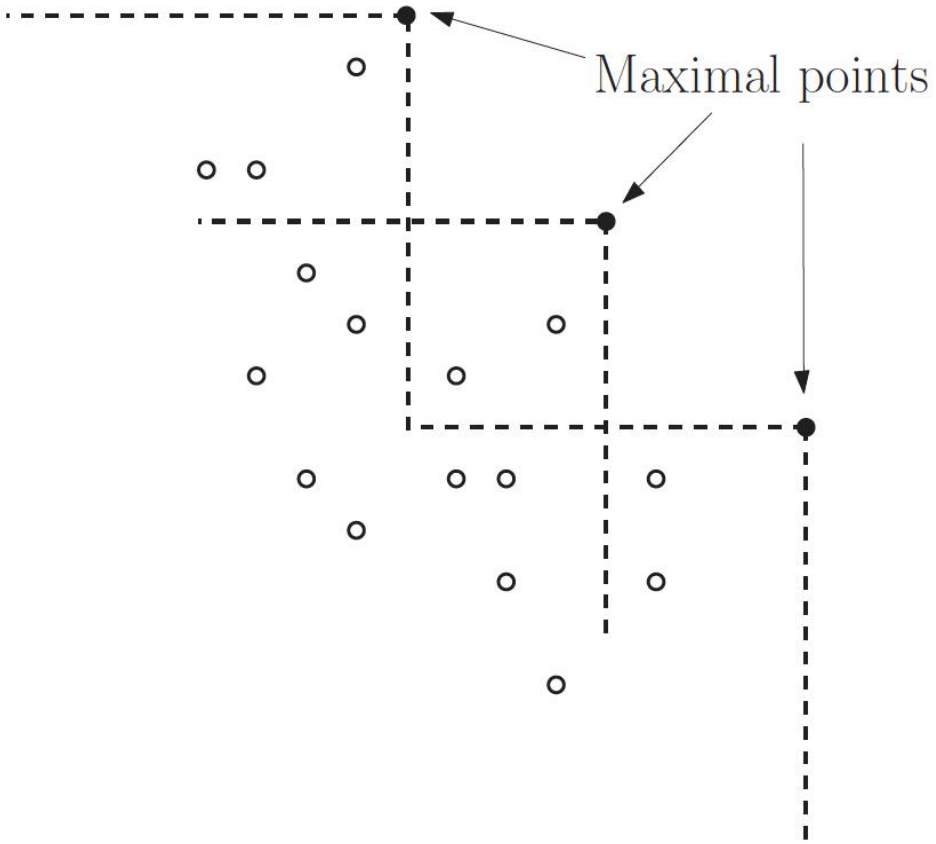
More Instance-Sensitive Algorithms

Michael T. Goodrich
University of California, Irvine

Some slides adapted from material from Jérémy Barbay, Christos Levcopoulos, Ola Petersson, and Wikipedia.

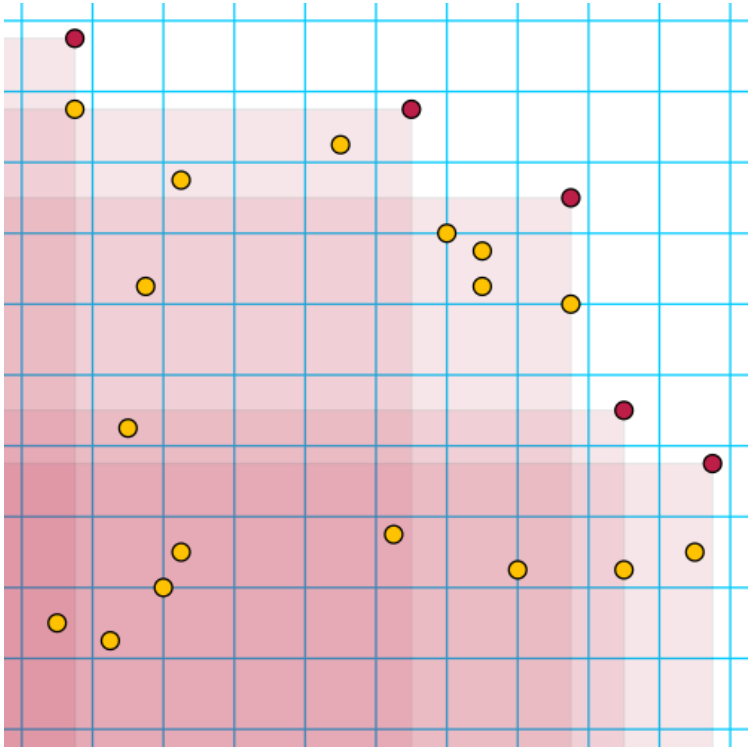
The Maximal Points Problem

- Given a set of n points in the plane, determine the **maximal points** that are not dominated by any other points.
- A point (x,y) is **maximal** if there is no other point with x-coordinate greater than x and y-coordinate greater than y.



A Simple Plane-Sweep Algorithm

1. Sort the points by decreasing x-coordinates.
2. For each point in this order, test whether its y-coordinate is greater than the maximum y-coordinate of any previously processed point.
3. If it is, output the point as a maximal point, and remember its y-coordinate as the greatest seen so far.



This algorithms runs in $O(n)$ time after sorting, so its running time depends on the time to sort.

Combine-and-Conquer

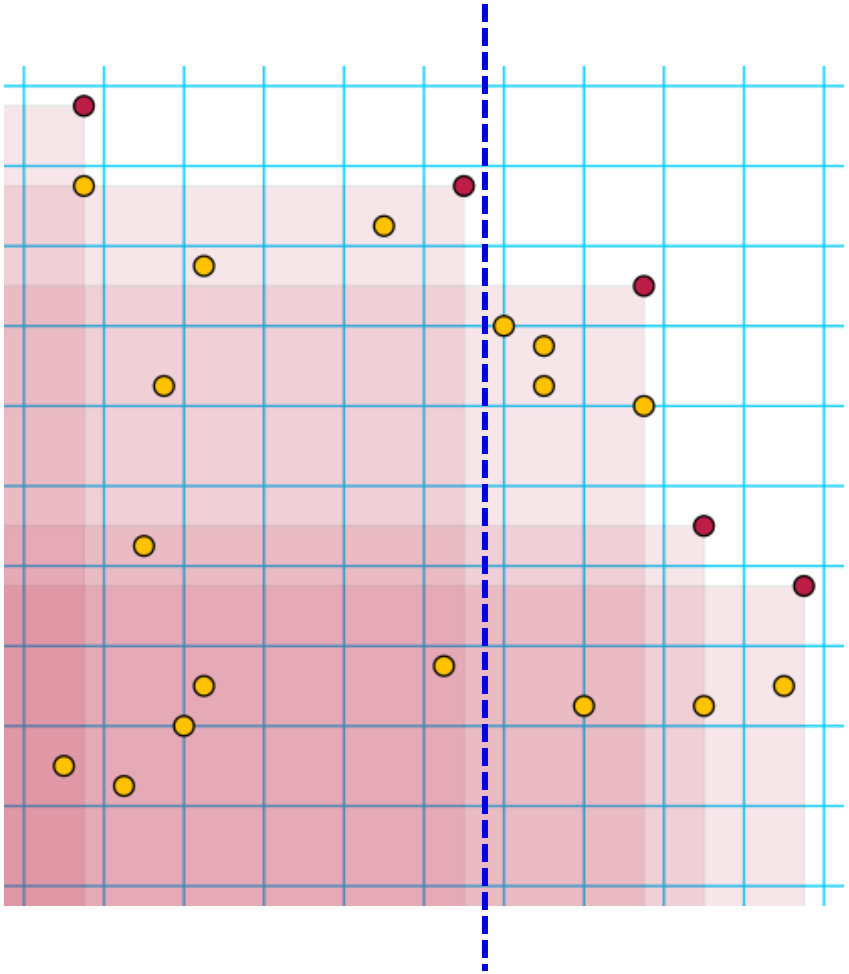
- Apply divide-and-conquer, but do the conquer step first.

MaximalPoints(S):

1. If $|S| = 1$ then return S.
2. Divide S into the left and right halves S_L and S_R by the median x-coordinate. (Can do this in $O(n)$ time.)
3. Find the point q with the maximum y-coordinate in S_R.
4. Prune all points in S_L and S_R that are dominated by q.
5. Return the concatenation of MaximalPoints(S_L) and MaximalPoints(S_R).

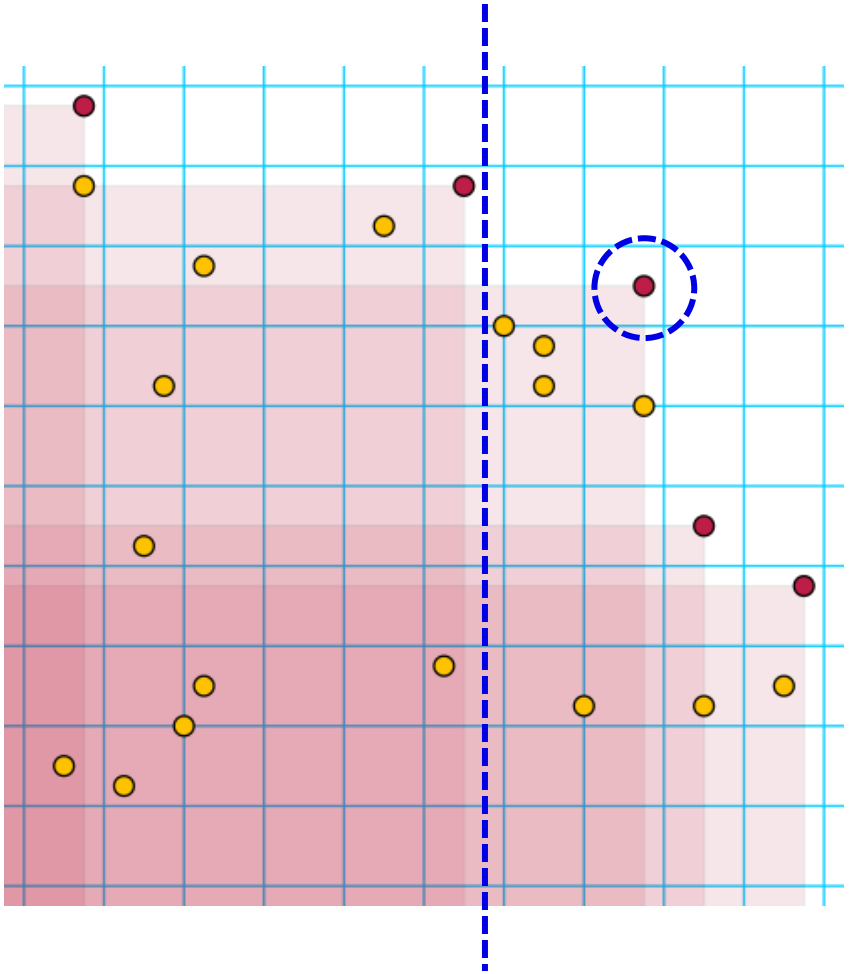
Split Step

- Divide at the median.



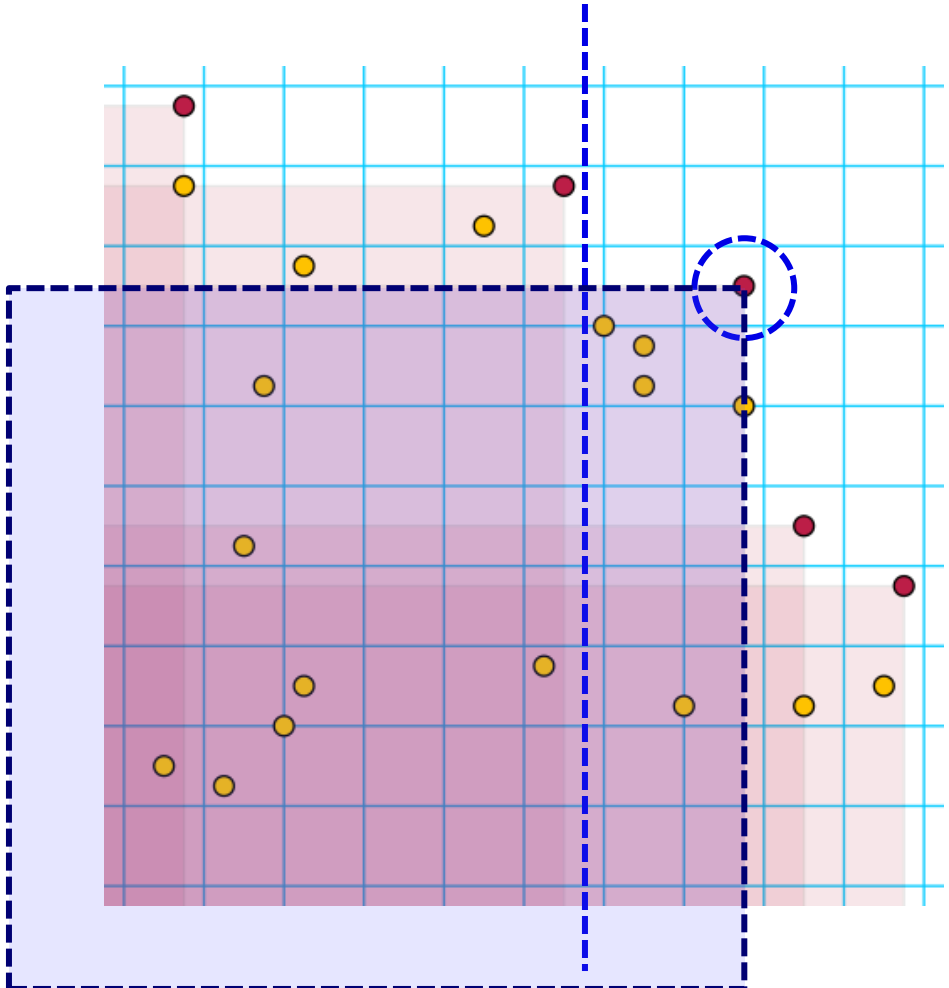
Conquer Step

- Find the point on the right with maximum y-coordinate.



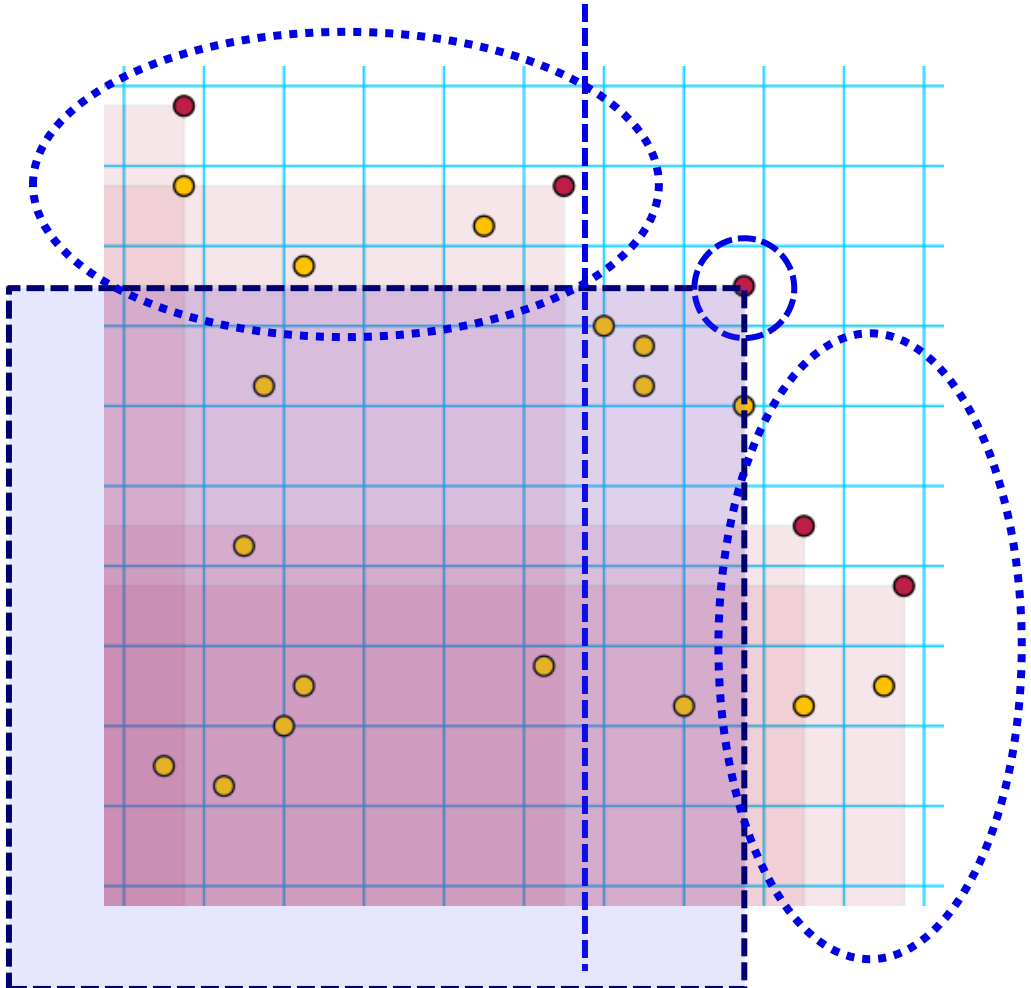
Prune Step

- Prune away points dominated by the maximal point in S_R.



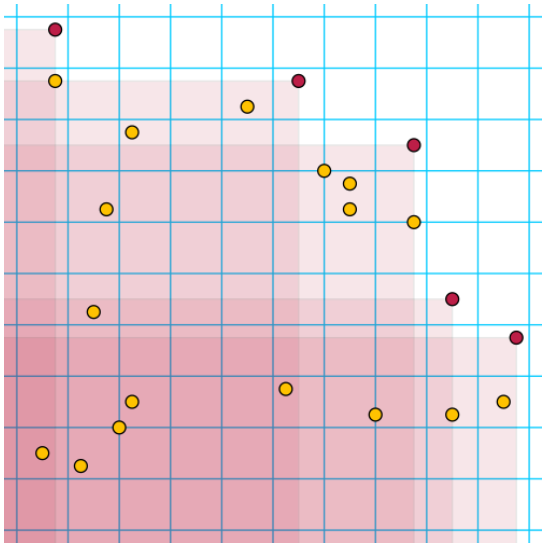
Divide Step

- Recursively solve the remaining points in S_L and S_R.



A First Analysis

- Each recursive call takes $O(n)$ time plus the times for the recursive steps.
- Each recursive call is guaranteed to find a maximal point.
- These facts imply that the time is $O(n \log h)$, where h is the size of the output.
- Why?

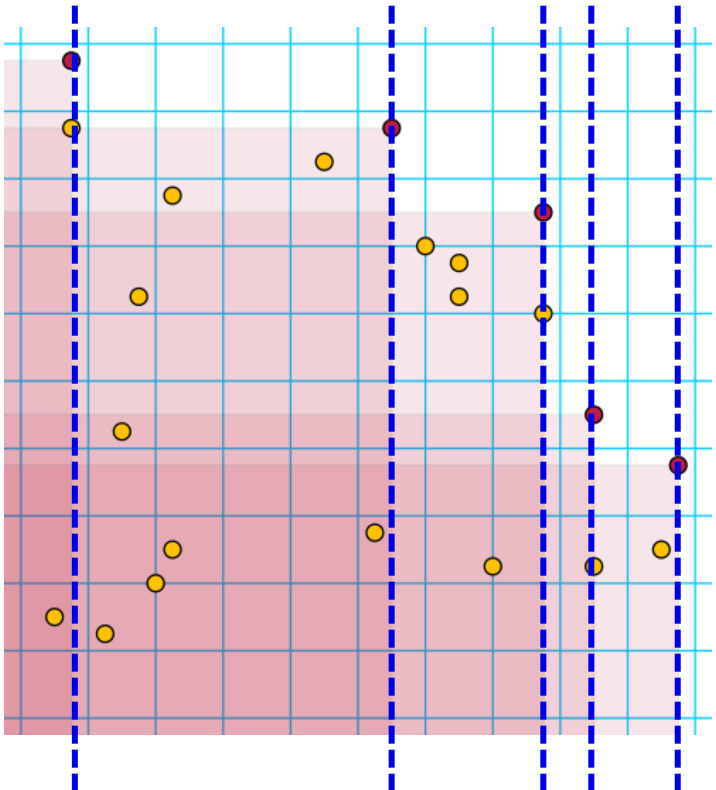


Vertical Entropy

- Let n_i be the number of points dominated by the i-th maximal point but not the (i-1)-set maximal point.

$$H_v = \sum_{i=1}^{h} \frac{n_i}{n} \log_2 \left(\frac{n}{n_i} \right)$$

- Note that H_v is at most $\log_2 h$.

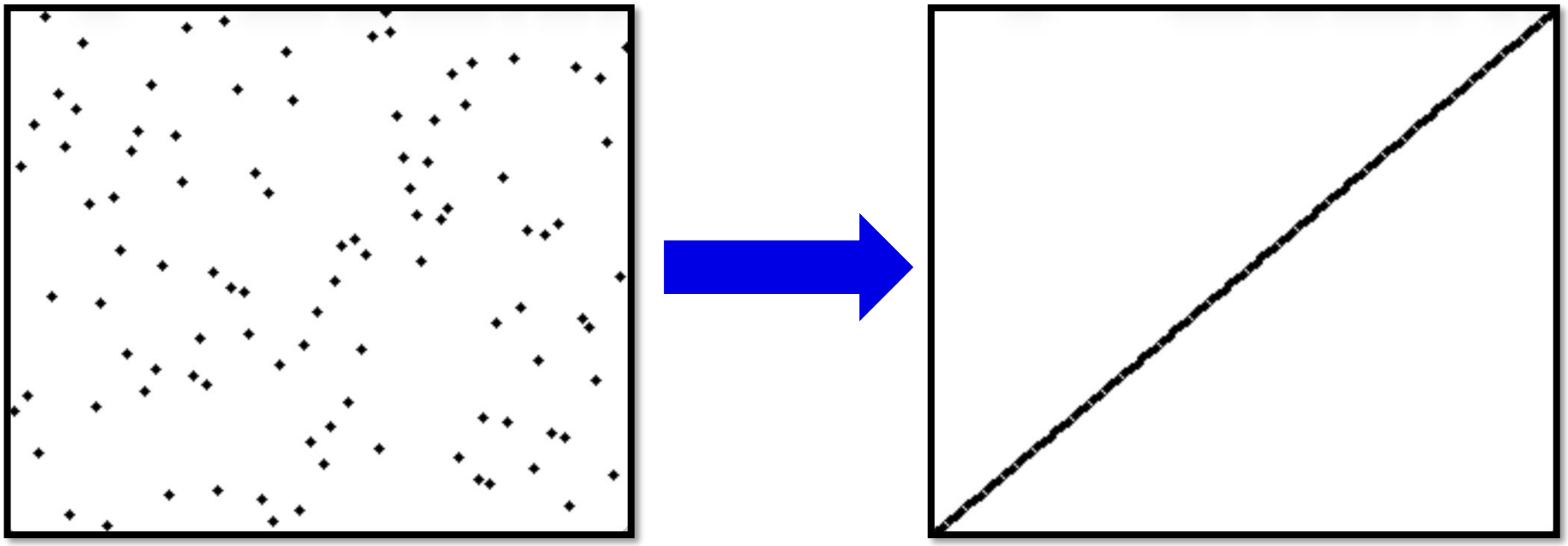


An Entropy-Sensitive Analysis

- The combine-and-conquer algorithm runs in $O(nH_v)$ time.
- **Proof sketch**:
- If a maximal point dominates at least n_i input points, then it will be identified by the MaximalPoints algorithm after at most $\log_2(n/n_i)$ rounds.
- For example, if $n_i \geq n/2$, it will be identified immediately; if $n_i \geq n/4$ it will be identified after at most two levels of recursion; and so on.
- Thus, the n_i input points that it dominates contribute at most n_i comparisons to at most $\log_2(n/n_i)$ partition phases.

Revisiting the Sorting Problem

- Given an array, X, of n elements, reorder the elements to be in nondecreasing order.

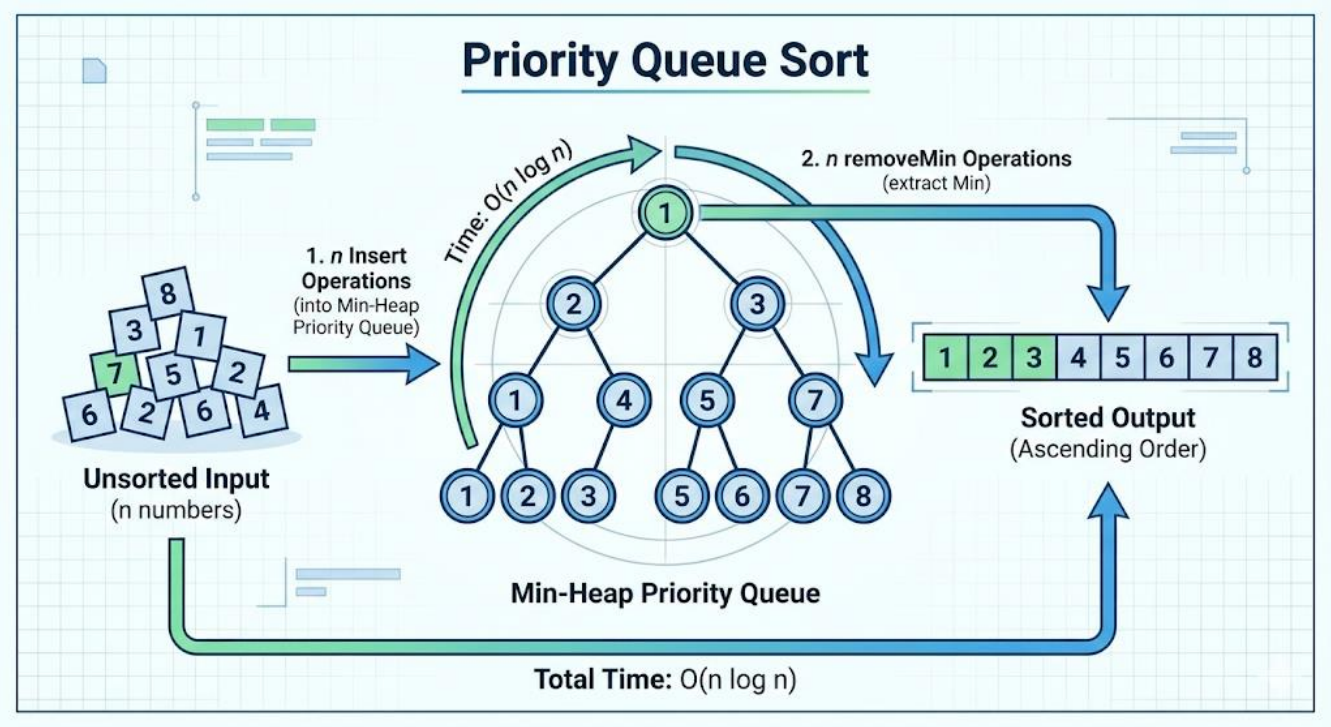


Recalling Priority Queue Sort (Heapsort)

- Let Q be a priority queue, e.g., using a heap that supports insert and extractMin operations in $O(\log n)$ time:

1. Insert the elements into Q using n insert operations.
2. Perform n extractMin operations on Q to output the elements in sorted order.

Worst case: $O(n \log n)$ time

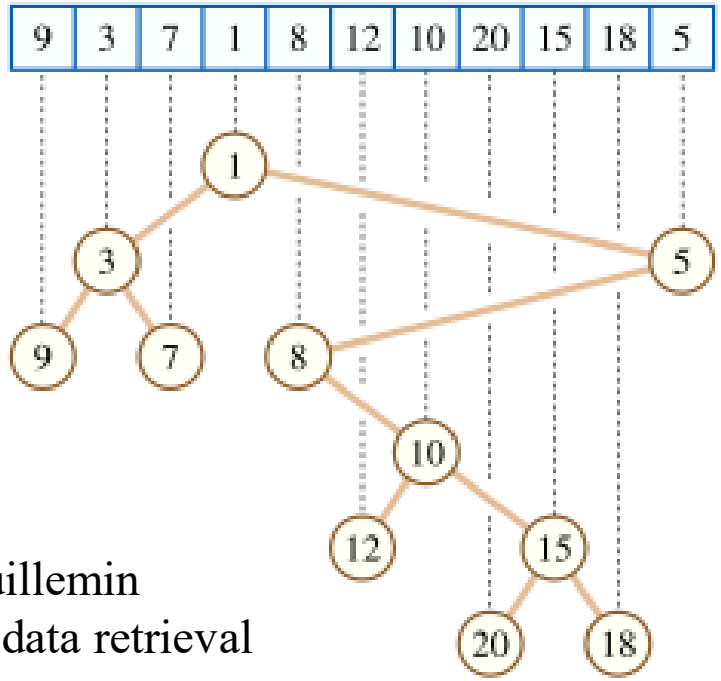


Best case: $O(n \log n)$ time

Image generated using Nano Banana Pro.

Cartesian Tree

- A Cartesian tree is a binary tree derived from a sequence of numbers.
- To construct the Cartesian tree, set its root to be the minimum number in the sequence, and recursively construct its left and right subtrees from the subsequences before and after this number.

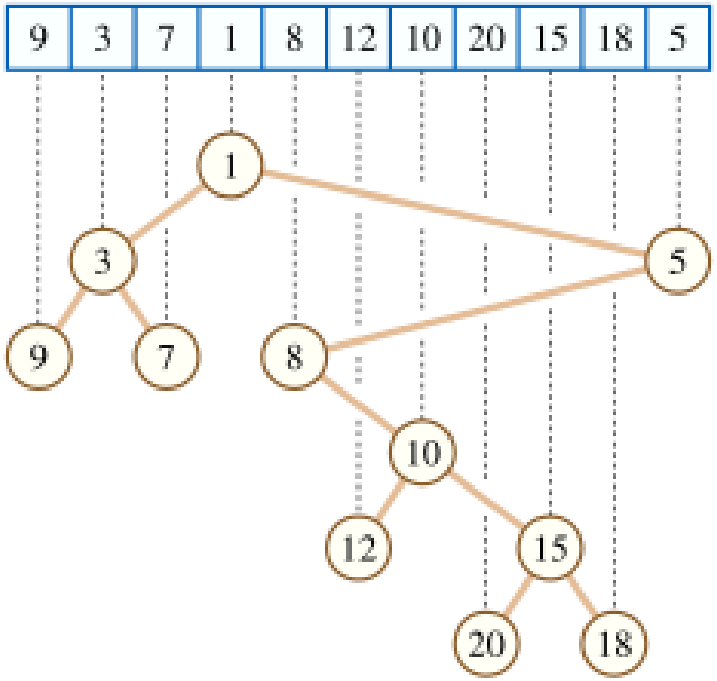


Introduced by Jean Vuillemin
in 1980 for geometric data retrieval

Hybrid Nature of Cartesian Trees

- A Cartesian tree elegantly combines the properties of two distinct data structures:

1. A Heap (specifically, a min-heap or max-heap).
2. A Binary Search Tree (in terms of its structural traversal).



Two Properties for Cartesian Trees

1. **Heap Property**: The value of any node is strictly less than (or strictly greater than) the values of all its descendants. Result: The minimum (or maximum) value of the sequence is at the root.
2. **In-Order Traversal Property**: If you perform an in-order traversal (Left Child -> Node -> Right Child), you recreate the original sequence.

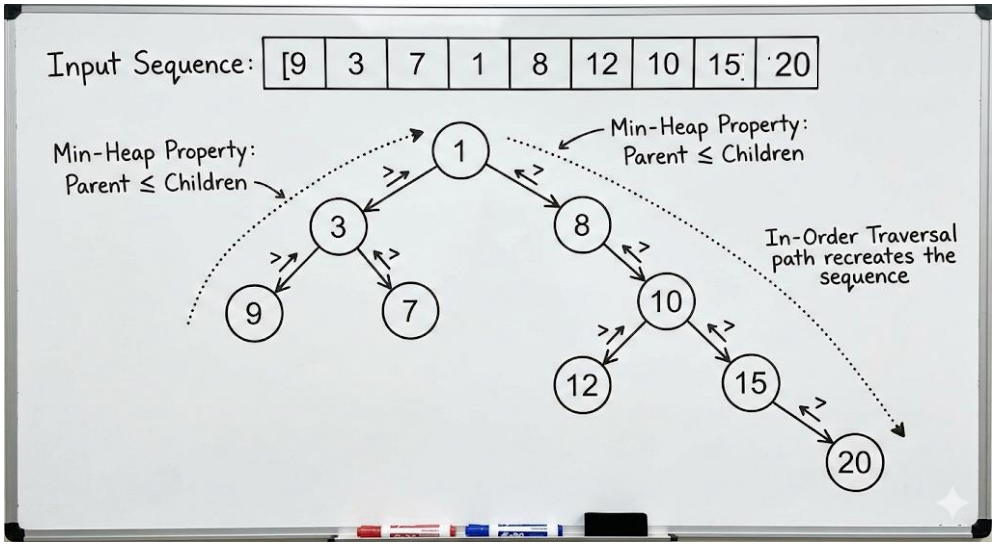


Image generated using Nano Banana Pro.

Algorithm for Constructing a Cartesian Tree

- The recursive method can degrade to an algorithm running in $O(n^2)$ time, but we can build it in $O(n)$ time using a stack:
- Process the array left to right, maintaining a stack of nodes for the "right spine," i.e., the nodes from the last inserted elements up to the root.
- When inserting a new number, pop nodes off the stack as long as they are larger than the new number. The last node popped becomes the left child of the new number. The new number becomes the right child of the new top of the stack. Push the new number onto the stack.

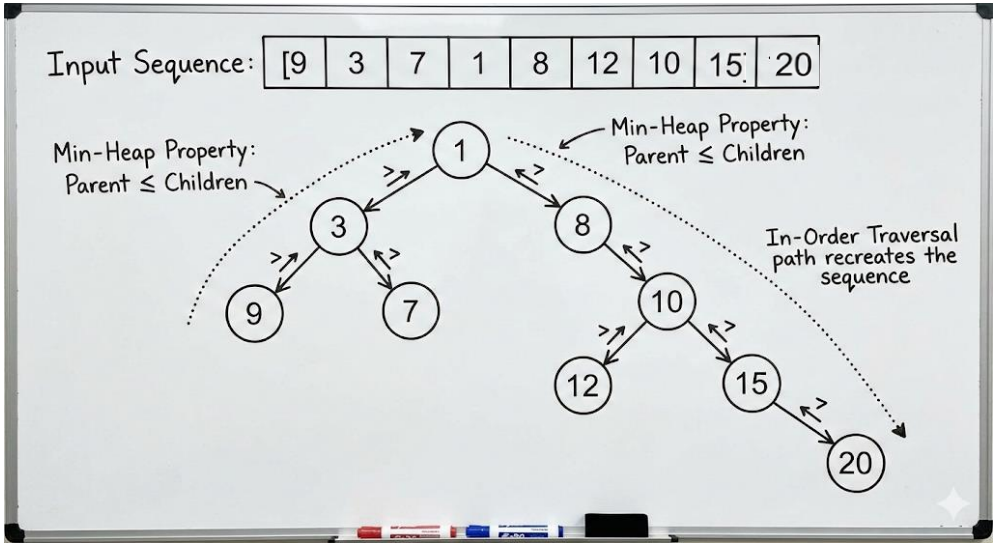


Image generated using Nano Banana Pro.

Cartesian-Tree Priority Queue Sort

1. Construct a cartesian tree for the input sequence.
2. Initialize a priority queue, Q, initially containing only the tree root of the cartesian tree.
3. While Q is non-empty:
    1. Find and remove the minimum value in Q
    2. Add this value to the output sequence
    3. Add the Cartesian tree children* of the removed value to Q

- This algorithm is correct because of the heap property of the cartesian tree. [Levocopoulos & Petersson, 1989]

*We assume each element is stored along with a pointer to its node in the cartesian tree.

Step 1: Construct the Cartesian Tree

- For the sequence [9, 3, 7, 1, 8, 12, 10, 15, 20], the smallest element 1 is the root and the tree is as follows:

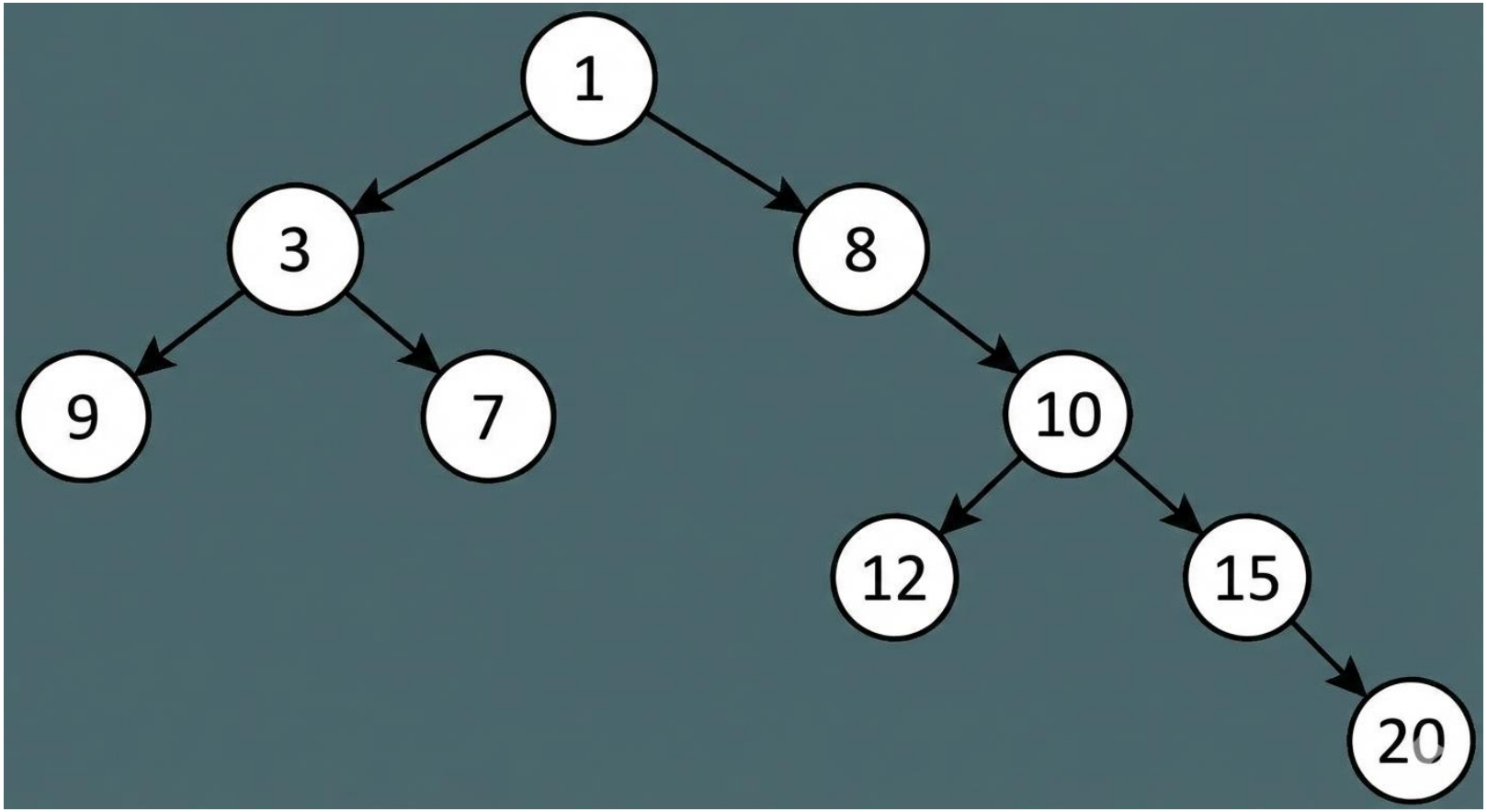


Image generated using Nano Banana Pro.

Step 2: Initialization

- Initialize the priority queue, Q.

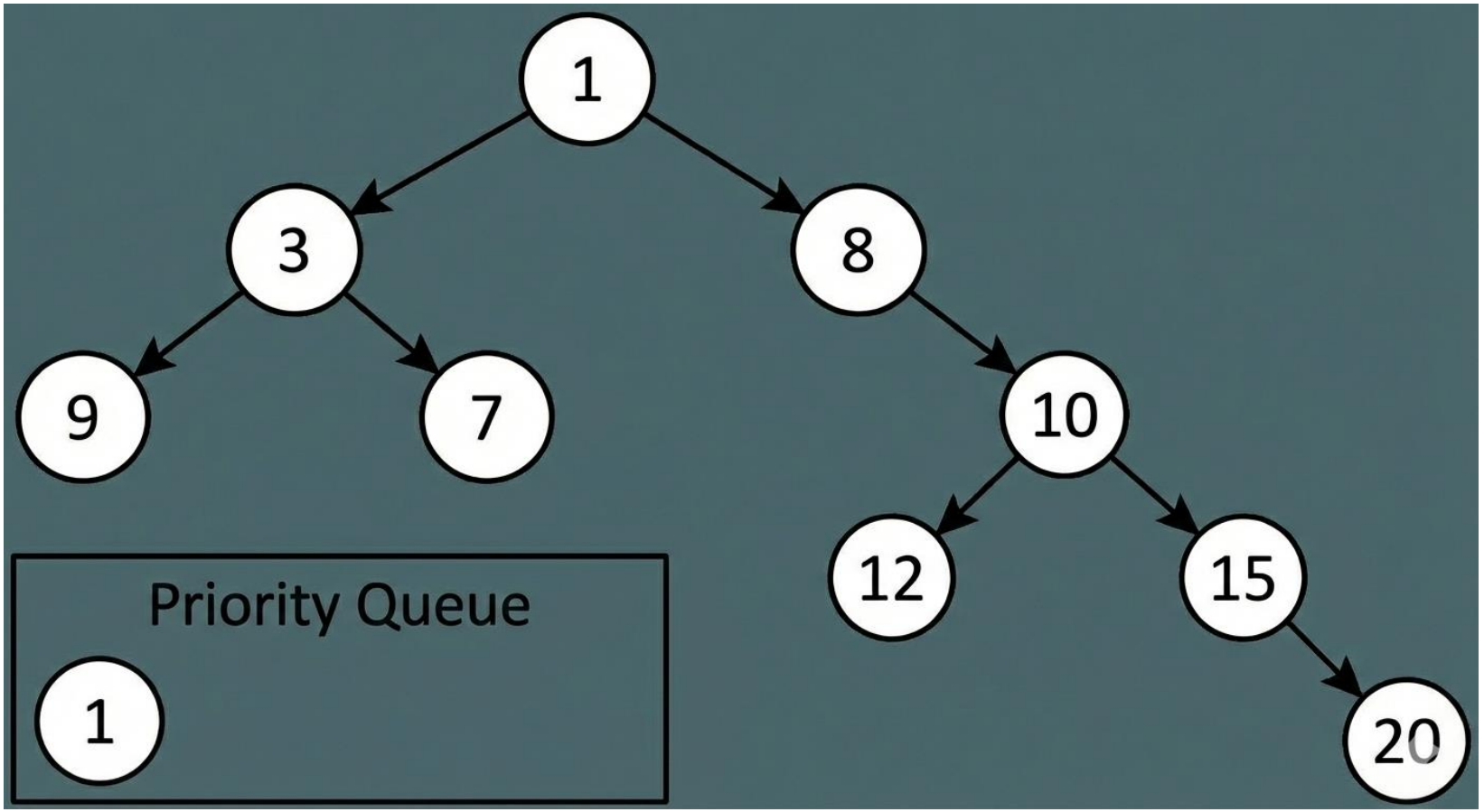


Image generated using Nano Banana Pro.

The Beginning of the Loop

- We find and remove the minimum value (1) from Q. We add 1 to our output sequence. We add the children of 1 from the Cartesian tree, which are 3 and 8, to Q.

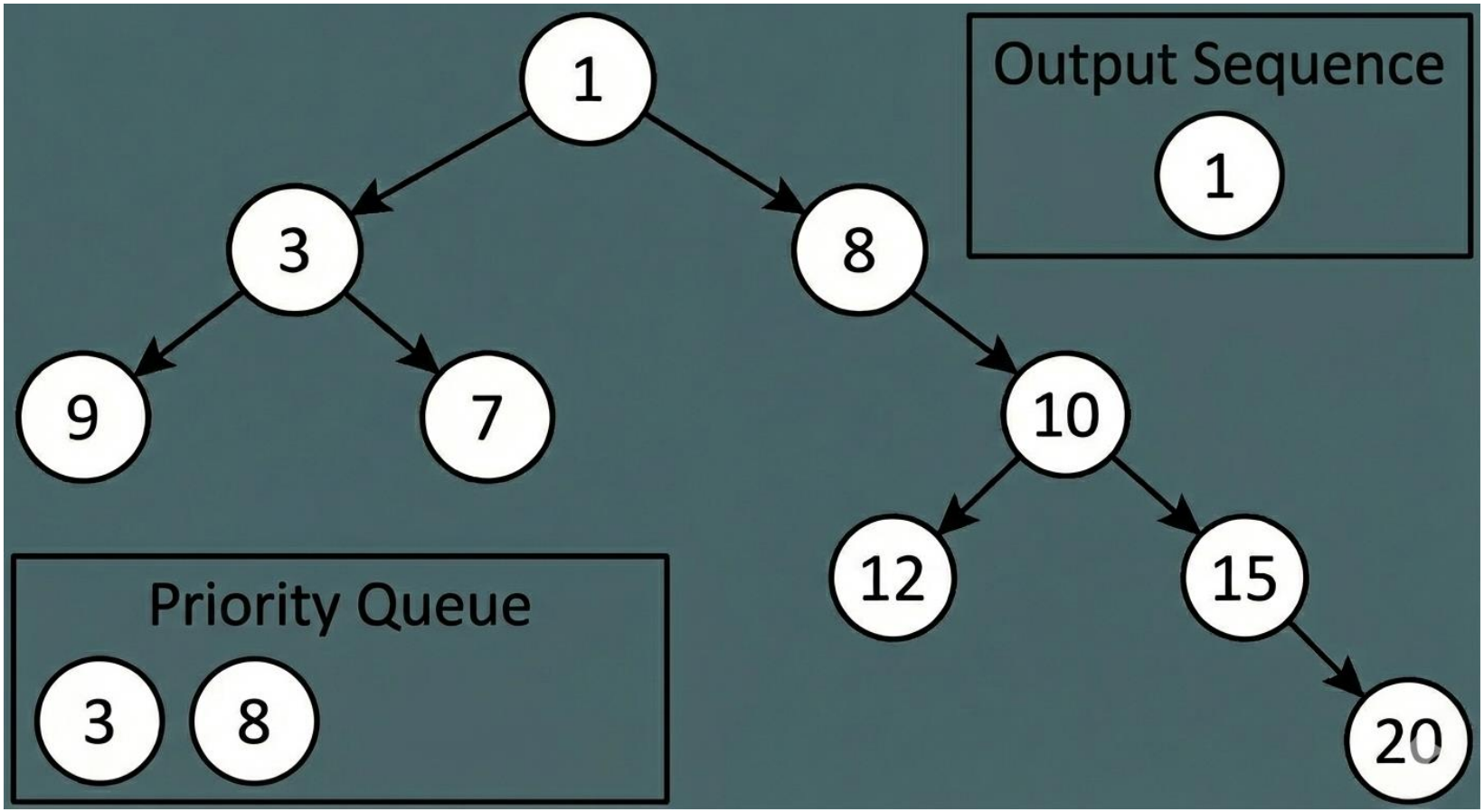


Image generated using Nano Banana Pro.

Analysis

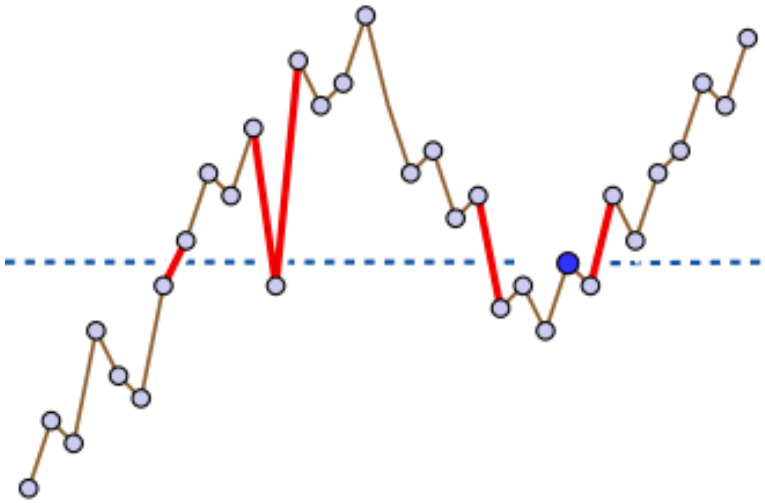
- The running time for Cartesian-tree Priority Queue sort is $O(n \log n)$ in the worst case, just like heapsort.
- We can do a finer analysis by considering an **oscillation measure**, Osc(X), for the input sequence, X.

Definition 2.1 Let $X = \langle x_1, \ldots, x_n \rangle$ be a sequence. For each $i \in \{1, \ldots, n\}$, let $Cross(x_i) = \{j \mid 1 \leq j < n \text{ and } \min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\}\}$.

We define the following measure of presortedness.

Definition 2.2 For a sequence X of length n, let $Osc(X) = \sum_{i=1}^{n} |Cross(x_i)|$.

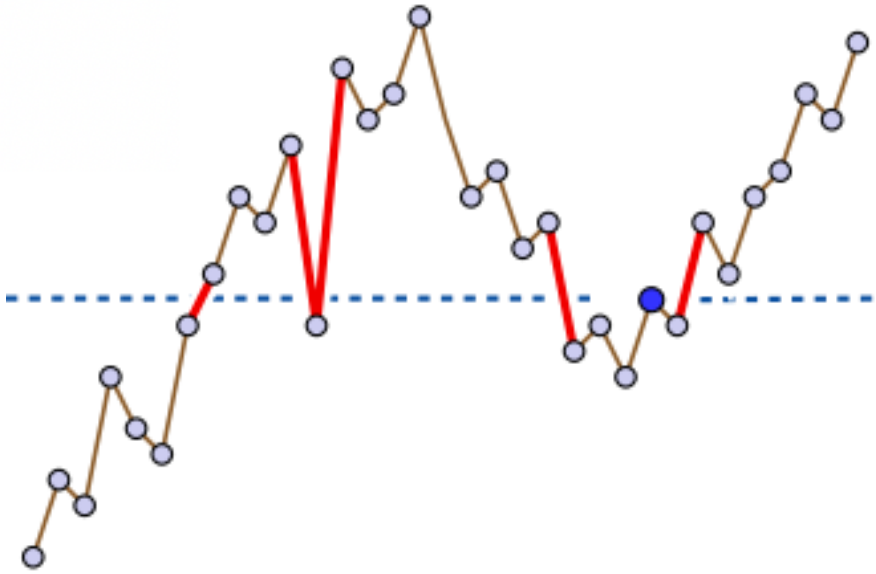
Pairs of consecutive sequence values (shown as the thick red edges) that bracket a sequence value (the darker blue point). The cost of including this value in the sorted order produced by the Levcopoulos–Petersson algorithm is proportional to the logarithm of this number of bracketing pairs.



Properties of Osc(X)

- This oscillation measure has the following properties:

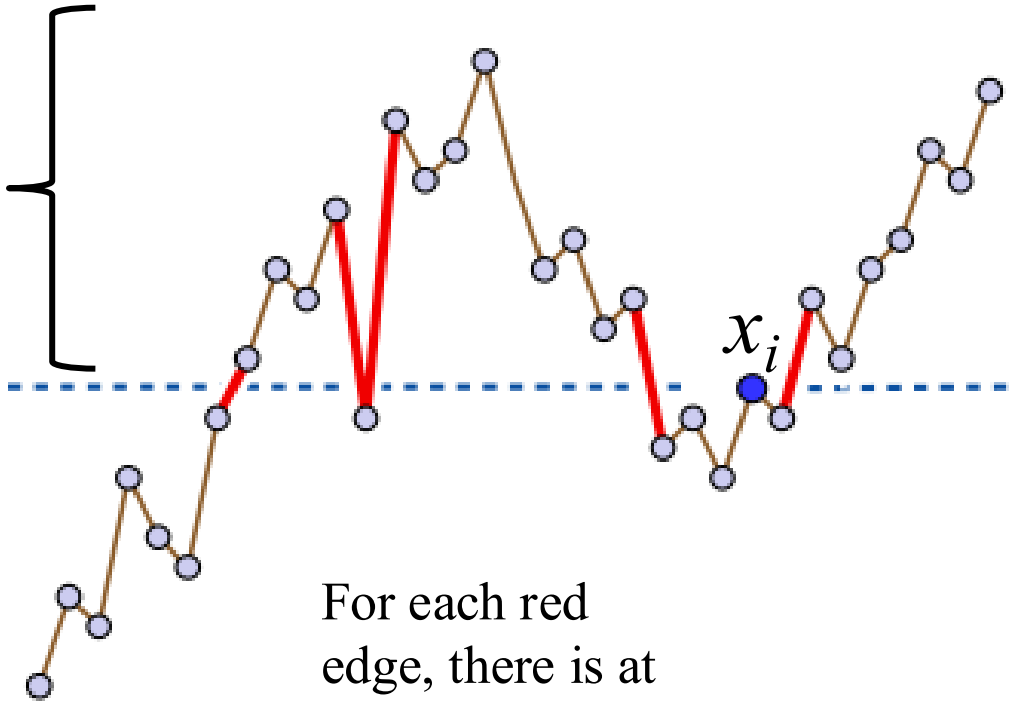
1. $Osc(X) \geq 0$, with equality if and only if X is in ascending or descending order.
2. If $X = \langle x_1, \ldots, x_n \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ and $x_i < x_j$ if and only if $y_i < y_j$, for all $i, j \in \{1, \ldots, n\}$, then $Osc(X) = Osc(Y)$.
3. If Y is a subsequence of X then $Osc(Y) \leq Osc(X)$.
4. For all $x \in N$, $Osc(\langle x \rangle X) \leq |X| + Osc(X)$.
5. $Osc(X) \leq n^2/2$.
6. $Osc(\langle x_n, x_{n-1}, \ldots, x_1 \rangle) = Osc(X)$.



ExtractMin Lemma

- When an element x_i is removed from the priority queue, Q, there are $O(|\text{Cross}(x_i)|)$ elements in Q.

These elements have already been removed when we remove x_i.



For each red edge, there is at most one element currently in Q.

The Running Time Lemma

- The running time for Cartesian-tree Priority Queue sort is

$$O(n + \sum_{x_i} \log(|Cross(x_i)|)).$$

- **Proof**:
- By the ExtractMin Lemma, the size of the priority queue is $O(|\text{Cross}(x_i)|)$ when x_i is removed.
- When x_i is removed, we add two more elements to Q.
- Thus, processing each x_i takes $O(\log |\text{Cross}(x_i)|)$ time.

Total Running Time

- The total running time for Cartesian-tree Priority Queue sort is

$$O(n + n \log(\frac{Osc(X)}{n}))$$

- **Proof**:

$$\begin{aligned}
T(n) &= O(n + \sum_{x_i} \log(|Cross(x_i)|)) \\
&= O(n + \log(\prod_{i=1}^{n} |Cross(x_i)|)) \\
&= O(n + n \cdot \log(\prod_{i=1}^{n} |Cross(x_i)|)^{1/n}) \\
&\leq O(n + n \cdot \log(\frac{1}{n} \cdot \sum_{i=1}^{n} |Cross(x_i)|)) \\
&= O(n + n \log(\frac{Osc(X)}{n}))
\end{aligned}$$

because of the inequality for geometric and arithmetic averages.