## Warm-up

**Problem 1.** Come up with an instance showing that SELECTION-SORT takes $\Omega(n^2)$ time in the worst case.

**Solution 1.** Any array will takes $\Omega(n^2)$: For each of the first $n/2$ iterations of the top level for loop, the inner loop runs for at least $n/2$ iterations, each one taking constant time, so $\Omega(n^2)$ overall.

**Problem 2.** Come up with an instance showing that INSERTION-SORT takes $\Omega(n^2)$ time in the worst case.

**Solution 2.** Consider the array sorted in descending order. In the $i$th iteration of the for loop when we need to insert $A[i]$ we need to move all the entries from 0 to $i-1$ one position to the right. Therefore, the last $n/2$ iterations of the for loop will cause the inner while loop to iterate at least $n/2$ times, so $\Omega(n^2)$ time overall.

## Problem solving

**Problem 3.** Come up with an instance showing that HEAP-SORT takes $\Omega(n \log n)$ time in the worst case.

**Solution 3.** Imagine a heap with keys $1, \ldots, n$, such that $n = 2^h - 1$; that is, the last level (i.e., level $h-1$) is full. Imagine the last level of the heap has the keys $2^h - 1, 2^h - 2, \ldots, 2^{h-1}$ listed from left to right. Now suppose we do $2^{h-1}$ remove-min operations. On each operation, after moving the key of the last node (call it $k$) to the root, we need to perform a down-heap operation to repair the heap-order property. Note that every node in level $h-2$ and up is smaller than $k$, while every node in level $h-1$ has a larger key; thus the down-heap operation brings $k$ down to level $h-2$, which takes $\Omega(h)$ work. Thus, the total time spent on these operations is $\Omega(2^{h-1}h) = \Omega(n \log n)$.

**Problem 4.** Given an array $A$ with $n$ integers, an inversion is a pair of indices $i < j$ such that $A[i] > A[j]$. Show that the in-place version of INSERTION-SORT runs in $O(n + I)$ time where $I$ is the total number of inversions.

**Solution 4.** Let $I_i$ be the number of inversions at the beginning of the iteration of the for loop corresponding to $i \in [1, n)$ and $I_n$ be the number of inversions at the end; thus we have $I_1 = I$ and $I_n = 0$. Note that if the inner while loop runs for $k_i$ iterations for a given value of $i$, it removes $k_i$ inversions and thus $I_i = I_{i+1} + k_i$. Let $W$ be the total number of iteration of the inner while loop. Then,

$$W = \sum_{i=1}^{n-1} k_i = \sum_{i=1}^{n-1} (I_i - I_{i+1}) = I_1 - I_n = I.$$

The total running time is clearly $O(n + W)$, so the total running time in $O(n + I)$.

**Problem 5.** Given an array $A$ with $n$ distinct integers, design an $O(n \log k)$ time algorithm for finding the $k$th value in sorted order.

**Solution 5.** We use a priority queue that supports MAX and REMOVE-MAX operations. We start by inserting the first $k$ elements from $A$ into the priority queue. Then we scan the rest of array comparing the current entry $A[i]$ to the maximum value in the priority queue, if $A[i]$ is greater than the current maximum, we can safely ignore it as it cannot be the $k$th value due to the fact that the priority queue holds $k$ values smaller than $A[i]$. On the other hand, if $A[i]$ is less than the current maximum, we remove the maximum from the queue and insert $A[i]$. After we are done processing all the entries in $A$, we return the maximum value in the priority queue.

To argue the correctness of the algorithm we can use induction to prove that after processing $A[i]$, the $k$ smallest elements in $A[0,i]$ are in the queue. Since we return the largest element in the queue after processing the whole array $A$ (which we just argued holds the $k$ smaller elements in $A$), we are guaranteed to return the $k$th element of $A$.

The time complexity is dominated by $O(n)$ plus the time it takes to perform the REMOVE-MAX and INSERT operations in the priority queue. In the worst case we perform $n$ such operations each one taking $O(\log k)$ time, when we implement the priority queue as a heap. Thus, the overall time complexity is $O(n \log k)$.

**Problem 6.** Given $k$ sorted lists of length $m$, design an algorithm that merges the list into a single sorted lists in $O(mk \log k)$ time.

**Solution 6.** We keep a priority queue holding pointers to positions in the lists where the priority of a pointer is the value it points to. Initially, we add the head of each list to the priority queue. We iteratively remove the minimum priority pointer, add it at the end of the merged list, and then, provided we are not already at the end of that list, insert the next element of the list where that pointer came from.

To argue the correctness of the algorithm, notice that the minimum value in the queue never goes down, as the next pointer we add can only have larger value than the minimum we just removed because the input lists are sorted. Therefore the output is in sorted order.

For the time complexity, note that we always keep at most $k$ items in the priority queue, so each remove and insert operations takes $O(\log k)$, when using a heap. Since there are a total of $mk$ elements, the total time is $O(mk \log k)$.

**Problem 7.** In this question we're going to analyse a response generated by an AI, in this case ChatGPT. It was asked to design an algorithm for the following problem that runs in linear time, argue its correctness, and analyse its running time.

**Problem Description:** We have $n \geq 1$ lighthouses of distinct positive integer height on a line and these lighthouses need to communicate with each other. We are given the heights of these lighthouses in an array $A$, where $A[i]$ stores the height of lighthouse $i$. Unfortunately, the area these lighthouses are in is so remote that it doesn't have phone lines and cell phone coverage is spotty at best. So the lighthouse keepers have found a different way to talk to each other: paper airplanes.

Assume all lighthouse keepers are (former) olympic champions in paper airplane throwing, meaning that they can reach any lighthouse as long as there's no lighthouse higher than the thrower's lighthouse between them (the lighthouses are equipped with paper airplane catchers, so hitting the wall of the lighthouse counts as reaching it). Of course, because of the wind coming from the ocean, lighthouse keepers can only send their paper planes to the left (towards the land). More formally, lighthouse $k$ can send a paper airplane to lighthouse $i$ ($i < k$), if there is no lighthouse $j$ such that $i < j < k$ and $A[k] < A[j]$. Note that lighthouse $i$ itself can be higher than lighthouse $k$ because of the paper airplane catchers. We are interested in determining the leftmost lighthouse that each of the lighthouse keepers can reach. To distinguish between a paper airplane stopping at lighthouse 0 and passing over all lighthouses to the left of its starting point, we record the latter as $-1$ (imagining an infinitely high lighthouse preceding all others where the airplane stops).

Example:
$A : [5, 3, 1, 6, 4]$: return $[-1, 0, 1, -1, 3]$. A paper airplane thrown by the lighthouse keeper of lighthouse 0 passes over all lighthouses to its left and is thus recorded as $-1$. Lighthouse 1 can send their airplanes to lighthouse 0, but not beyond. Lighthouse 2 can communicate with lighthouse 1, but not with lighthouse 0, since lighthouse 1 stops any airplanes throw from lighthouse 2. Lighthouse 3 is higher than any of the previous lighthouses and thus its airplanes pass over them. Finally, lighthouse 4 can send airplanes to lighthouse 3, but no further.

Your task is to design an algorithm that computes for each lighthouse the leftmost lighthouse that its keeper can reach. For full marks, your algorithm should run in $O(n)$ time.

**AI Output:**
Algorithm Description:

1. Initialization: Initialize an array reach of size $n$ to store the results, where reach$[i]$ will eventually hold the index of the leftmost lighthouse that lighthouse $i$ can reach. Initialize a list candidates to store the indices of lighthouses that have been processed and can potentially be reached by future lighthouses.

2. Processing Each Lighthouse:

   - Iterate through each lighthouse $i$ from left to right.
   - For each lighthouse $i$, use binary search (specifically, 'bisect_right') on the candidates list to find the smallest index $j$ such that $A[j] \geq A[i]$.
   - If such $j$ exists and $j$ is less than $i$, update reach$[i]$ to $j$.
   - If $j$ does not exist or $j \geq i$, set reach$[i] = -1$.

3. Maintaining Candidates: After updating reach$[i]$, insert $i$ into the candidates list using binary insertion (inserting into a sorted list using 'bisect.insort').

4. Output: After processing all lighthouses, reach will contain the indices of the leftmost reachable lighthouse for each lighthouse.

Correctness Proof:

- Binary Search and Insertion: Using 'bisect_right' ensures that $j$ is found efficiently in $O(\log n)$ time for each lighthouse $i$. Inserting into candidates with 'bisect.insort' also maintains the sorted order in $O(\log n)$ time per insertion.

- Updating reach: For each lighthouse $i$, reach$[i]$ is correctly updated to the smallest index in candidates where the height $A[j] \geq A[i]$, ensuring it's the leftmost reachable lighthouse.

Time Complexity Analysis:

- Traversal: The main loop iterates through $n$ lighthouses.
- Binary Search and Insertion: Each operation (search and insertion) within the candidates list is $O(\log n)$.
- Therefore, the overall time complexity of the algorithm is $O(n \log n)$.

This algorithm efficiently computes the leftmost reachable lighthouse for each lighthouse in $O(n \log n)$ time, meeting the problem's requirements.

**Your task:** To analyse the above response consider each of the following questions and briefly explain your answer to each.

a) Does the AI response answer the question, i.e., does the algorithm solve the stated problem and does it run in the required time?

b) Is the algorithm described in sufficient detail, i.e., is all information present or are any parts unclear or missing?

c) Is the correctness argument complete or are there leaps or errors in the logic?

d) Is the running time analysis correct for the provided algorithm?

**Solution 7.**

a) This is unclear, but it's highly likely to be incorrect. Especially the part where "bisect.insort" is called to insert a lighthouse into a list is unclear, as this function isn't specified and "binary insertion" isn't a name commonly used, so it's unclear what is meant here. Even assuming that "bisect_right" and "bisect.insort" are correct implementations of binary search and insertion in a sorted list (hopefully this is what is meant), respectively, and run in the claimed times (more on this later), the algorithm would still need to have a way to keep track of which lighthouses are still reachable (i.e., not blocked by a later lighthouse) and as it isn't doing that, this would result in incorrect outputs.

b) No, there are a few parts that require improving. In Step 1, a list of candidates is initialized, but it isn't specified whether this list is empty or contains all lightbouses initially. It also isn't mentioned that this list will contain the lighthouses in sorted order, which is a requirement for the binary search in Step 2 to work correctly.

   In Step 2, the algorithm uses an unspecified function "bisect_right". Since this isn't a standard operation, its function and implementation should be explained. Later in this unit, we'll see binary search and after that's been covered in the lectures "bisect_right" no longer needs to be described, as just saying that binary search is applied would have sufficed.

   In Step 3, another unspecified function "bisect.insort" is used and the description of what this does should've been included. As it wasn't specified that the candidates list is sorted, it's very unclear what this function is supposed to achieve.

c) The first point seems to confuse a correctness proof with a running time analysis and parts of it should thus not be placed here. This is the first place where it's mentioned that candidates is a sorted list, a rather important detail. The second point is correct, since it depends on a binary search. At this point in the unit, this should still be explained, but once binary search is covered in the lectures, this argument would be correct.

d) This is an interesting one, since the algorithm claims that it can insert into a sorted list in $O(\log n)$ time. If this list is an array, this isn't true, since inserting into an array takes linear time. If the list is a singly- or doubly-linked list, insertion would take constant time (i.e., better than claimed), but we can no longer efficiently binary search on it, since we can't access an arbitrary index in constant time. In order to make this implementation run in $O(n \log n)$ time, a balanced binary search tree should be used, as there we can search and insert in $O(\log n)$ time. Hence, the analysis is incorrect.

   We should also note that this problem can actually be solved in linear time using only a stack, but the AI doesn't warn us of this and thus we ended up with an implementation that's less efficient than it could've been.