Extra Practice Problems for Quiz 1 – Solutions

This set of problems is to help you practice for Quiz 1 and is not to be turned in.

Problem 2-1. Lower Bounds in the Comparison Model

Do problems 8.1-1 and 8.1-3 from CLRS.

Solution: Problem 8.1-1. First to clarify, we distinguish 2 types of leaves: reachable and unreachable. Reachable are those for which there exists at least one permutation for which the algorithm reaches the leaf. Otherwise, the leaf is unreachable. For unreachable leaves, the depth could be as small as constant. But we are concerned with reachable leaves.

The answer is \(\Omega(n)\). We will prove in fact a lower bound of \((n - 1)/2\). Intuitively, once we decide on an answer (a permutation), we must have “looked” at all elements, but in one comparison we can look only at 2 elements.

More formally, consider any leaf that is at depth \(d < (n - 1)/2\). The path from the root to the leaf contains \(d\) comparisons, and thus there are at most \(2d < n - 1\) elements involved in these comparisons. Thus there are two elements that have not been involved in any comparison. Since the leaf is reachable, there is at least one permutation for which the algorithm reaches the leaf. But then there must be at least two permutations reaching to this leaf (since we can, for example, exchange the order two unmatched elements and obtain a different permutation that is consistent with all the comparisons that have been made). But this is impossible in a correct algorithm.

Problem 8.1-3. In all three cases the answer is still \(\Omega(n \log n)\).

In the class, we proved that any decision tree for sorting must be able to generate all possible permutations, and thus the worst-case lower bound for sorting is at least \(\log(\text{# permutations})\), which, in general, are \(n!\).

Consider a decision tree that works for at least a half of the permutations. Then we can ignore all the other permutations (and the corresponding part of the decision tree). Then we obtain a decision tree that has \(n!/2\) leaves. The new lower bound will be \(\log(\text{# leaves})\). Now, if the number of possible permutations is \(n!/2\), then the lower bound is \(\log(n!/2) = \log(n!) - 1 = \Omega(n \log n)\). Also, if the number of possible permutations is \(n!/n\), then the lower bound is \(\log(n!/n) = \log(n!) - \log n = \Omega(n \log n)\). Finally, if the number of possible permutations is \(n!/2^n\), then the lower bound is \(\log(n!/2^n) = \log n! - n = \Omega(n \log n)\).

Problem 2-2. Bounded Integers

Do problem 8.2-4 from CLRS.

Solution: In the preprocessing stage, we compute an array \(A_0, \ldots, A_k\) such that \(A_i\) is equal to the number of the integers at less than or equal to \(i\). Once we do such a preprocessing, answering the
query is easy: to find out the number of integers in the range \([a, b]\), compute \(A_b - A_{a-1}\) (assume that \(A_{-1} = 0\)).

The preprocessing is done as follows. First compute the array \(C_0, \ldots, C_k\), where \(C_i\) is the number of integers equal to \(i\) (as in counting sort). Then, scan through \(C\) from 0 to \(k\) to compute \(A_i\):

\[
A_i = A_{i-1} + C_i.
\]

**Problem 2-3. Stable Sorting**

Do problem 8.3-2 from CLRS.

**Solution:** INSERTION-SORT is stable since the **while** condition is \(A[i] > key\). Using \(\geq\) instead of \(>\) would result in a correct but unstable sort.

The stability of MERGE-SORT depends on how we write the procedure MERGE. The pseudo-code for this procedure is not given in the text. Remember how MERGE-SORT works: it divides the array into two sub-arrays, the left (or front) and the right (or back) halves, which it recursively sorts and then merges by comparing the head of the two (sorted) sub-arrays and putting in place the smaller one. If the two elements are equal, MERGE-SORT will be stable if the element of the left sub-array is chosen. It would be correct but unstable if the element of the right sub-array is chosen.

HEAP-SORT is not stable. An example is as follows. Imagine the array is already a heap, and there are two equal elements, \(a\) and \(b\), but \(a\) is in the left subtree (of the root) and \(b\) is in the right subtree. Then, during the sorting, just before the moment when either of the two become the root, both elements are children of the root, and the algorithm chooses the right element, \(b\). But it’s easy to arrange that \(a\) is before \(b\) in the original array.

QUICK-SORT is not stable either (at least the versions from the book). For example, let’s consider the deterministic version from Chapter 7.1 from CLRS. The problem is in the PARTITION algorithm. Specifically, the 6th step breaks the stability. For example, imagine we partition using element \(x = 3\), and we have the array 1, 1, 1, 5, 5, 2. Then the two 5’s will change their positions.

To obtain a stable sorting, take any sorting algorithm and its input array \(A\). We will augment each array element \(A[i]\) by storing with it the value of its initial index in the array, \(i\). Let’s denote this extra field as \(Index[i]\). Whenever the algorithm performs a comparison on two equal array elements \(A[i] = A[j]\), we will have it compare instead the values of \(Index[i]\) and \(Index[j]\). Therefore, if two elements have equal values, the one which started out closer to the beginning of the array will be deemed “less” than the other element. This approach requires \(\Theta(n)\) extra storage space and \(\Theta(n)\) extra running time, but since any sorting algorithm must spend \(\Omega(n)\) time looking at all of its input elements, the extra running time is absorbed and has no impact on the overall asymptotic running time of the algorithm.

**Problem 2-4. Priority Queue Implementation**
In this problem, we will implement some of the other operations on heaps: \textsc{Insert}, \textsc{Delete}, and \textsc{Increase}. (Together with the operations from the class, they will make up a priority queue implementation.)

(a) Implement the procedure \textsc{Insert}(Q, x) that inserts an element \(x\) into the heap \(Q\).

\textbf{Solution:} See CLRS, Section 6.5.

(b) Implement the procedure \textsc{Delete}(Q, i) that deletes the element at position \(i\) from the heap \(Q\). (Note that the procedure takes as a parameter the pointer \(i\) to the deleted element; otherwise, it would take \(\Omega(n)\) time just to find the element in the heap.)

\textbf{Solution:} One possible solution is to set the element to \(-\infty\) and run \textsc{Max-Heapify}(A, i). Then we just need to set the size of the array smaller by one (thus ignoring the \(-\infty\)).

(c) Implement the procedure \textsc{Increase}(Q, i, k) that increases the value of the element at position \(i\) from the heap \(Q\) to a new value, \(k\), which is assumed to have a higher value.

\textbf{Solution:} See CLRS, Section 6.5.