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 ith 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 kth 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 kth 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 kth 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 $O(mk \log k)$.