1. Exercise 6.1 in lecture notes 3: You can build a Min-heap (which takes $O(n)$). You can then do $k$ times Heap-Extract-Min, which is $O(k \log n)$. Overall the runtime is then $O(n + k \log n)$

2. Problem 7-4 (p. 188)
   
   (a) TAIL-RECURSIVE-QUICKSORT differs from QUICKSORT in only the last line of the loop. They do the same thing otherwise.
   
   (b) The stack depth will be $O(n)$ if the input array is already sorted. There will always be a partition of size 0 and a partition of size $n-1$, so there will be $n-1$ recursive calls.
   
   (c) Always call partition for the smaller sub-array first. It does not change the run time but it guarantees that the size never goes above $n/2$.

3. Let us prove by induction:
   
   - Base case: When a tree has depth 0 it has exactly $2^0 = 1$ node which is a leaf.
   - Inductive hypothesis: Let us assume that it’s true for any $k < n$, in other words:
     - If a binary tree has depth 1, it has at most $2^1 = 2$ leaves
     - If a binary tree has depth 2, it has at most $2^2 = 4$ leaves
     - If a binary tree has depth 3, it has at most $2^3 = 8$ leaves
     - etc.
   - Let us prove for $k = n$. When the depth of the tree becomes $n$, the maximum nodes we can add is two children per leaf (at the $k - 1$ level). By inductive hypothesis that when $k = n - 1$ the tree has at most $2^{k-1}$ leaves. Therefore, at the maximum case we convert each leaf to a parent of two new leaves, so the number of leaves becomes at most $2^n$.

4. 6.5-7: We can play with the insertion time as keys so that a newly inserted element would have a priority that puts it in the appropriate place in the queue or stack. For a stack, the operation Push can be implemented as Heap-Insert($x$) when $x$ is the new element, and its priority is the largest (for example, keep track of the priority $p$ of the current top of the heap and the new element has priority $p + 1$. Pop would be Heap-Extract-Max and top would be Heap-Maximum. For a queue, new elements should have the lowest priority in the heap, so that they will be the last to go. So one can maintain a variable which is the priority of the most recently inserted element and decrease it by 1 whenever we want to insert a new element. This way we implement a first-in-first-out queue.

5. 6.5-8: Say we want to delete node $i$. One can, for example, call $IncreaseKey(A, i, \infty)$ followed by $ExtractMax(A)$. The increase of the key to infinity will put it at the top of the heap, and the ExtractMax will take care of removing that key and arranging the heap. Both operations are $O(\log n)$, so calling them one after the other is still logarithmic.
Comment: Exchanging the $i^{th}$ number with the last, delete the last and do a heapify on the new $A[i]$ is only a partial answer, because heapify only deals with a case where $A[i]$ may be smaller than its subtree but OK with respect to the rest of the heap, which may not be the case. A correct answer is to call heapify if indeed $A[i]$ is smaller than its child(ren), and if not – check whether it is not smaller than its parent. If so - float it up (as you would with increaseKey for example).

6. This is sort of like the binomial theorem. Let us look at the full levels: At the $k$ level, every stage is the product of $k$ stages, where each stage was either 0.1 or 0.9 of the stage above. So it is a binomial distribution with $p = 0.1$ and $q = 0.9$ and the partition cost which sums up to the overhead on the right hand side is: $\sum_{i=0}^{k} cn\left(\binom{k}{i}(0.9)^i(0.1)^{k-i}\right) = cn$ according to the binomial theorem. Obviously for partially full levels this is an upper bound as some of these terms are missing.

7. This can probably be shown in more than one way, here is the one I found easiest (there may be more elegant ways though): Let’s look at the first row $a$ after the sorting of rows and then columns. Let us look at any two indices $i, j$ such that $i < j$. The element at position $i$ at row $a$ after the double sorting was originally the $i^{th}$ smallest element, following row sorting and before column sorting, of some row $l$ – let us call this element $l_i$, and the element at position $j$ was, after row sorting and before column sorting, the $j^{th}$ smallest element at some row $m$, denoted $m_j$. Since this is the first row, then $l_i$ is the smallest element in column $i$, and in particular - $l_i \leq m_i$ where $m_i$ was the element that was originally the $i^{th}$ smallest at row $m$ after row sorting. Since $i < j$, then after row sorting $m_i \leq m_j$ and therefore $m_j \geq l_i$. This is true for any $i, j$ and therefore the top row is sorted. We can remove it from consideration and apply the same logic to the second row which is the smallest remaining etc.

8. Assume that $c \geq 0$. Assume you had some kind of super-hardware that, when given two lists of length $n$ that are sorted, merges them into one sorted list, and takes only $nc$ steps.

(a) The recursive method can be like MergeSort: Split at the midpoint, call recursively on each half and merge the two with the super-hardware procedure. The boundary condition is of course if the lists are empty or have one item.

(b) $T(n) = 2T(n/2) + 2(n/2)^c$ – it should be noted that the two merged list are of size $n/2$ each.

(c) According the the master theorem, any value of $c < 1$ would basically fall in case 1 and would make $T(n) = n$. Therefore, we would be able to sort in significantly less than $O(n \log n)$, which is highly unlikely.