1. Say we want to delete node $i$. One can, for example, call $\text{IncreaseKey}(A, i, \infty)$ followed by $\text{ExtractMax}(A)$. 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:** I saw that some people exchange the $i^{th}$ number with the last, delete the last and do a heapify on the new $A[i]$. This 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).

2. You can maintain a Min-heap $A$ of size $k$, that at every given moment keeps the minimum element in each of the $k$ heaps. Each time we extract an element from the min-Heap we extract the minimum out of all these minima, and add it to the sorted list of merged items. Each time we put the minimum element in a heap we remove it from its list. We then have to trace back the list it came from and insert the next smallest element into the heap $A$. We perform overall up to $n$ HeapInsert and $n$ HeapExtractMin, but the size of the heap is always at most $k$, so the total runtime is $O(n \log k)$

3. We have to combine several results we’ve shown before: That a completely full heap has $2^H - 1$ nodes, and that the bottom level has between $1$ and $2^{H-1}$ leaves (the latter if completely full). If a heap is completely full, then each subtree has exactly $2^{H-1} - 2$ nodes (that’s the number of nodes in a full heap divided by 2, minus the root). The worse case of imbalance is if the last level is half full, so the entire set of leaves in the $H$ level is $\frac{2^H}{2}$

4. 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 $O(n + k \log n)$

5. 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: $\sum_{i=0}^{k} cn(\binom{k}{i})(0.9)^{i}(0.1)^{k-i} = cn$ according to the binomial theorem. Obviously for partially full levels this is an upper bound as some of these terms are missing.

6. The answer to all the questions is "no". The reason is that for every tree with $L$ leaves, the minimum depth of the tree is $\Omega(\log L)$. So, if we only look at the half of the $n!$, their subtree has $\frac{n!}{2}$ leaves, in which case the depth is $\Omega(\log \frac{n!}{2})$ which is still $\Omega(n \log n)$... The same logic
applies to \( \frac{n!}{n} \) (the depth of this tree is \( \Omega(\log n! - \log n) = \Omega(n \log n) \) and \( \frac{n!}{2^n n} \), where the depth of the tree is \( \Omega(\log n! - n) = \Omega(n \log n) \).

7. Problem 8-5 (a-d only) (page 207).

(a) It means the array is sorted. If we substitute \( k = 1 \) in the formula above we get \( \sum_{j=i}^{i+1} A[j] \rightarrow A[i] \leq A[i + 1] \) for each \( i \), which is the definition of sorting.

(b) For example, \( \{2, 1, 3, 4, 5, 6, 7, 8, 9, 10\} \). For the first pair, \( \sum_{j=1}^{2} A[j] = 1.5 \), and \( \sum_{j=2}^{3} A[j] = 2 \), so the condition holds for the first three indices. For the other consecutive pairs it’s obviously true because the rest of the array is sorted.

(c) We can show that the definition of \( k \)-sorting is equivalent to the condition in the question. By definition, \( \sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k} A[j] \), which means \( \sum_{j=i}^{i+k-1} A[j] - \sum_{j=i+1}^{i+k} A[j] \leq 0 \) or equivalently, \( \sum_{j=i}^{i+k-1} A[j] - \sum_{j=i+1}^{i+k} A[j] \leq 0 \) (after multiplying both sides by \( k \)). These two sums have most terms in common except the first of the first and the last of the last, so when subtracting one from the other most terms cancel out and we get \( A[i] - A[i+k] \leq 0 \) or \( A[i] \leq A[i+k] \)

(d) According to (c) above, an array is \( k \)-sorted iff \( A[i] \leq A[i+k] \) for every \( i \). Therefore, to \( k \) sort an array we only need to make sure that every element \( A[i] \) is only sorted with respect to its \( k \) neighbors. Therefore, we divide the array into \( \frac{n}{k} \) subsets of \( A[1 + ik], A[2 + ik], ... \) and sort them with respect to one another using merge-sort or heap-sort. Each sorting is \( O\left(\frac{n}{k} \log \left(\frac{n}{k}\right)\right) \), and we do \( k \) such sorting operations, one per subset. Overall the runtime is \( O(k \frac{n}{k} \log (\frac{n}{k})) = O(n \log (\frac{n}{k})) \).

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 \( n^c \) 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) \( T(n) = 2T\left(\frac{n}{2}\right) + 2\left(\frac{n}{2}\right)^c \) – it should be noted that the two merged list are of size \( \frac{n}{2} \) 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.