6.2-6 (easy) Show that the worst-case running time of MAX-HEAPIFY on a heap of size $n$ is $\Omega(\log n)$.  
(Hint: For a heap with $n$ nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a path from the root down to a leaf.)

Solution Consider a heap of $n$ nodes where the root node has been changed to contain the smallest value of all the nodes. Now when we call Max-Heapify on the root, the value will have to be exchanged down with its child at every level, until it reaches the lowest level. This is because, after every swapping, the value will still be smaller than both its children (since it is the minimum), until it reaches the lowest level where it has no more children. In such a heap, the number of exchanges to max-heapify the root will be equal to the height of the tree, which is $\log n$. So the worst case running time is $\Omega(\log n)$.

6.4-4 (easy) Show that the worst-case running time of heapsort is $\Omega(n \log n)$.

Solution Note that heapsort is a comparison based sort. Then the lower bound of $\Omega(n \log n)$ follows from the lower bound on comparison based sorting.

Another way to see this is to use the solution from 6.2-6.

6.4-5 (difficult) Show that when all elements are distinct, the best-case running time of heapsort is $\Omega(n \log n)$.

Solution Note that this question was not graded.

For the solution, look at:

6.5-7 (easy) The operation \textsc{Heap-Delete}(A, i) deletes the item in node \(i\) from heap \(A\). Give an implementation of \textsc{Heap-Delete} that runs in \(O(\log n)\) time for an \(n\)-element max-heap.

\textbf{Solution}

\begin{algorithm}
\textbf{Algorithm 5} \textsc{Heap-Delete}(A, i)
\begin{algorithmic}
\State \textbf{Input:} A max-heap \(A\) and integers \(i\).
\State \textbf{Output:} The heap \(A\) with the element at position \(i\) deleted.
\State \(A[i] \leftarrow A[\text{heap-size}[A]]\)
\State \text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1
\State \text{key} \leftarrow A[i]
\If {\text{key} \leq A[\text{Parent}[i]]}
\State \text{MAX-HEAPIFY}(A, i)
\Else
\State \textbf{while} \(i > 1 \text{ and } A[\text{Parent}(i)] < \text{key} \textbf{ do} \)
\State \(A[i] \leftarrow A[\text{Parent}(i)]\)
\State \(i \leftarrow \text{Parent}(i)\)
\EndWhile
\EndIf
\end{algorithmic}
\end{algorithm}

6.5-8 (medium) Give an \(O(n \log k)\)-time algorithm to merge \(k\) sorted lists into one sorted list, where \(n\) is the total number of elements in all the input lists. (Hint: Use a min-heap for \(k\)-way merging.)

\textbf{Solution} We assume that the lists are sorted in ascending order.

Insert all \(k\) elements at position 1 from each list into a heap. Use \textsc{Extract-Min} to obtain the smallest element (say \(x\)) of the heap. Say then \(x\) came from list \(i\), then take the next element from list \(i\) and insert it into the heap. Continuing in this fashion yields the merged list. Clearly the running time is \(O(n \log k)\).

7.2-6 (medium) Argue that for any constant \(0 < \alpha \leq 1/2\), the probability is approximately \(1 - 2\alpha\) that on a random input array, \textsc{Partition} produces a split more balanced than \(1 - \alpha\) to \(\alpha\).

\textbf{Solution} For a random input, the pivot element will be equally likely be in any position of the array after a partition. A split is more balanced than \(1 - \alpha\) to \(\alpha\) if the pivot is settled within the range of positions \((\alpha n, (1 - \alpha)n)\). The probability for that should be \(((1 - \alpha)n - \alpha n)/n\), which is equal to \(1 - 2\alpha\).

7.4-6 (difficult) Consider modifying the \textsc{Partition} procedure by randomly picking three elements from array \(A\) and partitioning about their median (the middle value of the three elements). Approximate the probability of getting at worst an \(\alpha\)-to-\((1 - \alpha)\) split, as a function of \(\alpha\) in the range \(0 < \alpha < 1\).

\textbf{Solution} Let \(A\) (resp. \(B, C\)) denote the subset of elements in the positions range \([0, \alpha n]\) (resp. \([\alpha n, n - \alpha n]\), \((n - \alpha n, n)\),
We analyze the possible cases in which the median gives a valid (i.e. $\alpha$-to-$\lambda(1 - \alpha)$) partition and calculate their respective probabilities.

- $Pr[\text{One element from each of } A, B \text{ and } C] = 3! \cdot \alpha^2(1 - 2\alpha)$
- $Pr[\text{One element from } A, \text{ and two elements from } B] = \binom{3}{2} \alpha(1 - 2\alpha)^2$
- $Pr[\text{One element from } C \text{ and two elements from } B] = \binom{3}{2} \alpha(1 - 2\alpha)^2$
- $Pr[\text{All three elements from } B] = (1 - 2\alpha)^3$

The probability of a valid partition is just the sum of the above terms.

**9.3-4 (difficult)** Suppose that an algorithm uses only comparisons to find the $i^{th}$ smallest element in a set of $n$ elements. Show that it can also find the $i - 1$ smaller elements and the $n - i$ larger elements without performing any additional comparisons.

**Solution** Suppose that the algorithm uses $m$ comparisons to find the $i^{th}$ smallest element $x$ in a set of $n$ elements. If we trace these $m$ comparisons in the log, we can easily find the $i - 1$ smaller elements by transitivity. If $x > a$ and $a > b$, then $x > b$ can be deduced without actually performing a comparison between $x$ and $b$. Similarly, the $n - i$ larger elements can be found, too.

Say we can not decide that some element $e$ is smaller or greater than $x$. Then we claim that the algorithm does not work properly, since an adversary could design an input in which $e$ is the $i^{th}$ largest element instead of $x$. 