Solution 1.

a. 
\[
\begin{array}{cccccccc}
80 & 60 & 70 & 50 & 30 & 10 & 40 & 20 \\
\end{array}
\]
\[1 + 2 + 3 + 4 = 10\] comparisons

b. 
\[
\begin{array}{cccccccc}
70 & 60 & 40 & 50 & 30 & 10 & 20 & 80 \\
60 & 50 & 40 & 20 & 30 & 10 & 70 & 80 \\
50 & 30 & 40 & 20 & 10 & 60 & 70 & 80 \\
40 & 30 & 10 & 20 & 50 & 60 & 70 & 80 \\
30 & 20 & 10 & 40 & 50 & 60 & 70 & 80 \\
20 & 10 & 30 & 40 & 50 & 60 & 70 & 80 \\
10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\
\end{array}
\]
\[4\] comparisons

Total of \[4 + 4 + 4 + 2 + 2 + 1 + 0 = 17\] comparisons, after building the heap.

c. \[1 + 2 + 2 + 3 = 8\] comparisons

d. Total of \[3 + 3 + 3 + 2 + 2 + 1 + 0 = 14\] comparisons, after building the heap.

Solution 2.

(a) We generalize the representation of a 2-ary (binary) heap to a \(d\)-ary heap. Level 0 is stored in array element 1. Level 1 is stored in array elements 2 to \(d + 1\). Level 2 is stored in array elements \(d + 2\) to \(d^2 + d + 1\). In general, level \(j\) has \(d^j\) elements which are stored in locations \(1 + \sum_{i=0}^{j-1} d^i\) to \(\sum_{i=0}^{j} d^i\).

(b) A full tree of height \(k\) has \(\sum_{i=0}^{k-1} d^i = \frac{d^k - 1}{d - 1}\) elements. So a \(d\)-array heap with \(n\) elements must have \(\frac{d^k - 1}{d - 1}\) \(\geq n\). Solving for \(k\) we have \(k \geq \lceil \log_d((d - 1)n + 1) \rceil = \log_d n + O(1)\).

(c) Extract-Max removes the value in the root of the heap (which is the maximum value), places the last element of the heap to the side, and then sifts it down. Sift finds the child of the root with the maximum value, and compares this value to the value being sifted. If the sifted value is larger then it is placed in the root, and the heap has been restored. Otherwise, the maximum child value moves up to the root, and the algorithm is applied recursively using this child as the new root.

Finding the maximum child takes \(d - 1\) comparisons and comparing to the root takes one more, so the number of comparisons at every level is \(d\). Multiplying by the number of levels in the heap (i.e. its height) gives \(d \log_d n + O(1)\) comparisons for Extract-Max (assuming \(d\) is a constant).

(d) The algorithm does \(n - 1\) sifts, the tree always has at most \(n\) nodes, each sift takes at most \(d \log_d n + O(1)\) comparisons, so the total number of comparisons is at most \(nd \log_d n + O(n)\).
More precisely, the sorts are executed on trees with the number of nodes starting with \( n - 1 \), then \( n - 2 \), ..., 1. So the number of comparisons is

\[
\sum_{i=1}^{n-1} (d \log_d i + O(1)) = nd \log_d n + O(n) .
\]

This can be verified by bounding the summation by integrals.

(e) We have \( nd \log_d n = nd \lg n / \lg d \). In order find the optimal value for \( d \), we treat \( d \) as a variable and \( n \) as a constant. For small values of \( d \) we can compute \( d / \lg d \); for \( d = 2 \) (standard heapsort) we get 2, and for \( d = 3 \) we get \( \approx 1.89 \). For \( d > 3 \) the derivative is positive so all values will be greater.

Thus, \( d = 3 \) is the optimum value. This yields an algorithm for Heapsort that uses only \( 1.89n \lg n + O(n) \) comparisons.

**Solution 3.**

Set a counter to 0. Start at the root of the heap and start visiting children of nodes, and children of children, etc. in any order. If a node has value greater than \( x \) do do not visit its children; otherwise, increment the counter (and visit its children). Stop if the counter reaches \( k \), in which case the answer is YES (the \( k \)th smallest is less than or equal to \( x \)); stop if you run out of eligible children to visit, in which case the answer is NO.