6.4-3
What is the running time of heapsort on an array $A$ of length $n$ that is already sorted in increasing order? What about decreasing order?

6-2 *Analysis of d-ary heaps*

A *d-ary heap* is like a binary heap but (with one possible exception) non-leaf nodes have $d$ instead of 2 children.

1. How would you represent a $d$-ary heap in an array?

2. What is the height of a $d$-ary heap of $n$ elements in terms of $n$ and $d$?

3. Give an efficient implementation of EXTRACT-MAX in a $d$-ary max-heap. Analyze its running time in terms of $n$ and $d$.

4. Give an efficient implementation of INSERT in a $d$-ary max-heap. Analyze its running time in terms of $n$ and $d$.

5. Give an efficient implementation of INCREASE-KEY($A$, $i$, $k$), which first sets $A[i] \leftarrow \max(A[i], k)$ and then updates the $d$-ary max-heap structure appropriately. Analyze its running time in terms of $d$ and $n$.

7.2-3
Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array $A$ contains distinct elements and is sorted in decreasing order.

7.4-2
Show that quicksort’s best-case running time is $\Omega(n\log n)$. 
7-4 Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After the call to PARTITION, the left subarray is recursively sorted and then the right subarray is recursively sorted. The second recursive call in QUICKSORT is not really necessary; it can be avoided by using an iterative control structure. This technique, called tail recursion, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion.

QUICKSORT\textsc{'}(A, p, r)
1 \textbf{while } p < r
2 \textbf{do } Partition and sort left subarray.
3 \hspace{1em} q \leftarrow \text{PARTITION}(A, p, r)
4 \hspace{1em} QUICKSORT\textsc{'}(A, p, q - 1)
5 \hspace{1em} p \leftarrow q + 1

\textbf{a.} Argue that QUICKSORT\textsc{'}(A, 1, \text{length}[A]) correctly sorts the array $A$.

Compiler usually execute recursive procedures by using a stack that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is invoked, its information is \textit{pushed} onto the stack; when it terminates, its information is \textit{popped}. Since we assume that array parameters, are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The \textit{stack depth} is the maximum amount of stack space used at any time during a computation.

\textbf{b.} Describe a scenario in which the stack depth of QUICKSORT\textsc{'} is $\Theta(n)$ on an $n$-element input array.

\textbf{c.} Modify the code for QUICKSORT\textsc{'} so that the worst-case stack depth is $\theta(lgn)$. Maintain the $O(n\log n)$ expected running time of the algorithm.