Suppose we base our max-heaps on ternary instead of binary trees. Each node now has a maximum of three children. We still require the tree to be filled by levels: the root, then the three left-to-right children of the root, then the nine left-to-right children at level 2, and so forth. A tree with \( n \) nodes will always have all levels completely full, except possibly the bottom level. For the questions below, assume the tree has \( n \) nodes and levels 0, 1, 2, \ldots, \( k \). For working out the time complexities required, you can assume that all levels are full. We modify the max-heap property to read “A tree satisfies the max-heap property if the value at each node is no smaller than the values at its children.”

1. Prove that \( k = \Theta(\log_3 n) \).

2. Explain how to map the tree into an array with indices running from 1 to \( n \). What formulas calculate the indices of the left, middle, and right children as a function of the parent index? What formula calculates a parent index from the index of a child?

3. Modify the MaxHeapify algorithm for binary heaps to work for ternary heaps. That is, the new routine MaxHeapify\((A, i)\) rearranges the contents of the nodes in the subtree rooted at index \( i \) to satisfy the max-heap property, assuming that the subtrees rooted at the children of index \( i \) already satisfy the property. Show the the worst-case time complexity of the algorithm is \( \Theta(\log_3 n') \), where \( n' \) is the number of nodes in the subtree rooted at \( i \).

4. Modify the BuildMaxHeap algorithm to work for the ternary case. The new algorithm accepts an array with no constraints on the size relationships among the entries. It considers the array to be a ternary tree and by judiciously applying the new MaxHeapify algorithm, it converts the array into a heap. That is, the final array, interpreted as a ternary tree, satisfies the max-heap property at each node. Show the the new algorithm is \( \Theta(n) \), worst case.

5. Finally, modify the Heapsort algorithm to work in the ternary case. Show that the worst-case time complexity is \( \Theta(n \log_3 n) \).
Solutions.

1. Assuming levels 0, 1, . . . , \(k\) are full, the tree contains \(n\) nodes and

\[
\begin{align*}
    n &= \sum_{i=0}^{k} 3^i = \frac{3^{k+1} - 1}{3 - 1} = \frac{3^{k+1} - 1}{2} \\
    2n + 1 &= 3^{k+1} \\
    k &= \log_3(2n + 1) - 1 \\
    k &\leq \log_3(3n) - 1 = \log_3(n) + 1 - 1 = \log_3 n \Rightarrow k = O(\log_3 n) \\
    k &\geq \log_3(n) - 1 \geq \log_3(n) - \frac{1}{3} \log_3(n) = \frac{2}{3} \log_3 n \Rightarrow k = \Omega(\log_3 n),
\end{align*}
\]

the last inequality holding when

\[
\begin{align*}
    \frac{1}{3} \log_3 n &\geq 1 \\
    \log_3 n &\geq 3 \\
    n &\geq 27.
\end{align*}
\]

Consequently, \(k = \Theta \log_3 n\).

2. Map the root to position 1 of the array, as in the binary case, then pack the children as tightly as possible while maintaining a breadth-first traversal of the tree. We get the following arrangement of array indices.

<table>
<thead>
<tr>
<th>parent</th>
<th>children</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>2</td>
<td>5, 6, 7</td>
</tr>
<tr>
<td>3</td>
<td>8, 9, 10</td>
</tr>
<tr>
<td>4</td>
<td>11, 12, 13</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

If \(p\) is the index of a node, the left, middle, and right children of \(p\) are at indices \(3p - 1, 3p, 3p + 1\), respectively, provided these values do not exceed the array length. If \(c\) is the index of a child, then its parent is at \([c+1]/3\], provided this expression is greater than zero.

3. The MaxHeapify routine changes to allow a small root node to swap with its largest of three children, instead of two. The recursion follows into the subtree receiving the root in the same manner as before. Specifically,

\[
\begin{align*}
    \text{MaxHeapify}(A, i) \\
    \quad L = \text{left}(i); \ // L = 3i - 1 \\
    \quad M = \text{middle}(i); \ // M = 3i \\
    \quad R = \text{right}(i); \ // R = 3i + 1 \\
    \quad \text{if } (L \leq \text{heapsize}(A)) \text{ and } (A[L] > A[i]) \\
    \quad \quad \text{largest} = L \\
    \quad \text{else} \\
    \quad \quad \text{largest} = i \\
    \quad \text{if } (M \leq \text{heapsize}(A)) \text{ and } (A[M] > A[\text{largest}]) \\
    \quad \quad \text{largest} = M \\
    \quad \text{if } (R \leq \text{heapsize}(A)) \text{ and } (A[R] > A[\text{largest}]) \\
    \quad \quad \text{largest} = R \\
    \quad \text{if } (\text{largest} \neq i) \\
    \quad \quad \text{exchange } A[i] \leftrightarrow A[\text{largest}] \\
    \quad \text{MaxHeapify}(A, \text{largest})
\end{align*}
\]
In the worst case, node \( i \) will be smaller than any nodes in its three subtrees, and will therefore swap down to the bottom level of one of the subtrees. If the tree rooted at \( i \) contains \( n' \) nodes, and assuming the subtrees are full, this process will involve \( k \) swaps, where \( k \) can be obtained from the formula in Problem 1 by replacing \( n \) with \( n' \). That is, \( k = \log_3(2n' + 1) - 1 \). As in Problem 1, we have \( k = \Theta(\log_3 n) \) and therefore the worst-time complexity of the new MaxHeapify is \( \Theta(\log_3 n') \).

4. The new BuildMaxHeap will follow the same process as the binary algorithm. That is, it will start its right-to-left scan of the tree levels at the level just above the bottom. If \( n \) is the length of the array, then the node at \( i \) is not a leaf if and only if it has at least one child. That is, \( i \) is an internal node if and only if \( 3i - 1 > n \), or equivalently \( i > (n + 1)/3 \). So, the largest index belonging to an internal node is at \([(n + 1)/3]\), which marks the right end of the level just above the bottom. Since the nodes on this level have only singleton subtrees, they trivially satisfy the requirement of MaxHeapify. We can therefore apply MaxHeapify to each internal node as we scan right-to-left on each level, starting on the second-to-bottom level and proceeding to the root. Specifically,

\[
\text{BuildMaxHeap}(A)
\begin{align*}
\text{heapsize}(A) &= \text{length}(A); \\
\text{for } i &= \lceil \text{(length}(A) + 1)/3 \rceil \text{ downto } 1 \\
\text{MaxHeapify}(A, i)
\end{align*}
\]

For a worst-case analysis, assume the original array contains distinct integers in ascending order. Then as the algorithm encounters smaller nodes in its journey toward the root, successive MaxHeapify calls must swap these nodes to the bottom level. For nodes on the second-to-bottom level, this process involves just one swap. But, for nodes near the root, it requires nearly \( k \) swaps, where \( k \) is the lowest level. Assuming a full tree with levels 0, 1, 2, \ldots, \( k \), Problem 1 reveals the total number of nodes to be \( n = (3^{k+1} - 1)/2 \). We make a table of the number of nodes on each level and the distance they must travel to reach the bottom.

<table>
<thead>
<tr>
<th>level</th>
<th># of nodes</th>
<th># swaps to reach bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k - 1 )</td>
<td>( 3^{k-1} )</td>
<td>1</td>
</tr>
<tr>
<td>( k - 2 )</td>
<td>( 3^{k-2} )</td>
<td>2</td>
</tr>
<tr>
<td>( k - 3 )</td>
<td>( 3^{k-3} )</td>
<td>3</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>2</td>
<td>( 3^2 )</td>
<td>( k - 2 )</td>
</tr>
<tr>
<td>1</td>
<td>( 3^1 )</td>
<td>( k - 1 )</td>
</tr>
<tr>
<td>0</td>
<td>( 3^0 )</td>
<td>( k )</td>
</tr>
</tbody>
</table>

Because BuildMaxHeap is dominated by the number of calls to MaxHeapify, its asymptotic class will be the same to that of the total work done by these calls, say \( T(n) \). Therefore,

\[
T(n) = \sum_{i=0}^{k-1} (k - i)3^i = k \sum_{i=1}^{k} i \cdot 3^i = 3^k \sum_{i=1}^{k} \left( \frac{1}{3} \right)^i.
\]
To evaluate the sum, we argue as follows for $|t| < 1$.

$$f(t) = \sum_{i=0}^{s} t^i = \frac{1 - t^{s+1}}{1 - t}$$

$$f'(t) = \sum_{i=1}^{s} it^{i-1} = \frac{(1 - t)[-t(s + 1)t] - (1 - t^{s+1})(-1)}{(1 - t)^2}$$

$$= \frac{(1 - t^{s+1}) - (s + 1)t^s(1 - t)}{(1 - t)^2}$$

$$\sum_{i=1}^{s} i \cdot t^i = \frac{t(1 - t^{s+1}) - (s + 1)t^s(1 - t)}{(1 - t)^2}.$$ 

Letting $t = 1/3$ and $s = k$, we have

$$T(n) = 3^k \left( \frac{1}{3} \right)^{2(k+1)} - (k + 1) \left( \frac{2}{3} \right)^{k+1} \frac{2}{(2/3)^2} = 3^k \left( \frac{1}{3} \right)^{2(k+1)} \left[ 1 - \left( \frac{1}{3} \right)^{k+1} - 2(k + 1) \left( \frac{1}{3} \right)^{k+1} \right]$$

$$= \frac{1}{4} \left[ 3^{k+1} - 1 - 2(k + 1) \right] = \frac{1}{2} \left[ \frac{3^{k+1} - 1}{2} - (k + 1) \right].$$

Recalling, again from Problem 1, that $n = (3^{k+1} - 1)/2$ and $k + 1 = \log_3(2n + 1)$, we obtain

$$T(n) = \frac{1}{2} \left[ n - \log_3(2n + 1) \right] = \frac{n}{2} \left[ 1 - \frac{\log_3(2n + 1)}{n} \right].$$

Since $[\log_3(2n + 1)]/n \to 0$ as $n \to \infty$, we have $T(n) = \Theta(n)$.

5. No changes are needed in the HeapSort algorithm, other than substituting the ternary versions of BuildMaxHeap and MaxHeapify. The analysis remains unchanged because BuildMaxHeap remains $\Theta(n)$ and MaxHeapify remains $\Theta(\log_3 n') = \Theta(\log_2 n')$ in the worst case. Therefore, the worst case run time of HeapSort remains $\Theta(n \log n)$. 