6.1-3
Let $i$ denote the index of the element at the root of a subtree of a heap. Then its left child is $A[2i]$ and its right child is $A[2i + 1]$, if available. Recall that the definition of the max-heap property is: For every node $i$ other than root, $A[\text{Parent}(i)] \geq A[i]$ where $\text{Parent}(i) = \lfloor i/2 \rfloor$. Therefore,

For the left child $2i$, $A[\text{Parent}(2i)] = A[\lfloor 2i/2 \rfloor] = A[i] \geq A[2i]$

For the right child $2i+1$, $A[\text{Parent}(2i+1)] = A[\lfloor (2i+1)/2 \rfloor] = A[i] \geq A[2i + 1]$

Since $i$ is arbitrary, the largest element in a subtree of a heap is at the root of the subtree.

6.2-1
The array index begins at 1:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>27</td>
<td>17</td>
<td>3</td>
<td>16</td>
<td>13</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>12</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

1)

2)

3)
6.3-2

Notice that the conversion of an array into a heap is in a bottom-up manner. When processing node \( i \), we want the subtrees of node \( i \) already be heaps. In the other words, we want the node processing order to guarantee that the subtrees rooted at children of a node \( i \) are heaps before HEAPIFY is executed at that node. Because the elements at the bottom have larger index number than their parents’, we want the loop index \( i \) in line 2 of BUILD-MAX_HEAP to decrease to 1.

[We may not obtain a heap if the process starts from the root to the bottom. Try apply the algorithm on \(<4,1,3,8,6>\).]
7.1-1

(This solution is done using the algorithm from the lecture, which also appears in the first edition of the textbook. It now appears as Q7-1 on page 159.)

\[
\begin{array}{c}
\text{pivot} = A[1] = 13 \\
1) \\
\begin{array}{cccccccccccc}
13 & 19 & 9 & 5 & 12 & 8 & 7 & 4 & 11 & 2 & 6 & 21 \\
\end{array} \\
2) \\
\begin{array}{cccccccccccc}
13 & 19 & 9 & 5 & 12 & 8 & 7 & 4 & 11 & 2 & 6 & 21 \\
\end{array}
\text{(swap i,j)} \\
3) \\
\begin{array}{cccccccccccc}
6 & 19 & 9 & 5 & 12 & 8 & 7 & 4 & 11 & 2 & 13 & 21 \\
\end{array} \\
4) \\
\begin{array}{cccccccccccc}
6 & 19 & 9 & 5 & 12 & 8 & 7 & 4 & 11 & 2 & 13 & 21 \\
\end{array}
\text{(swap i,j)} \\
5) \\
\begin{array}{cccccccccccc}
6 & 2 & 9 & 5 & 12 & 8 & 7 & 4 & 11 & 19 & 13 & 21 \\
\end{array} \\
6) \\
\begin{array}{cccccccccccc}
6 & 2 & 9 & 5 & 12 & 8 & 7 & 4 & 11 & 19 & 13 & 21 \\
\end{array}
\text{Return 9 (index begins at 1)}
\end{array}
\]

7.2-4

The performance of quicksort is at best \(\Theta(n \log n)\). However, the performance of insertion-sort is at best \(\Theta(n)\). Now consider the algorithm of insertion-sort on page 24 of the textbook. For an almost-sorted input, the while-loop will be executed a few times more than in the best case. Since the extra steps are still constant, the performance of insertion-sort is still \(\Theta(n)\). Indeed, using the analysis similar to the one on page 24, \(T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1+k_1) + c_6k_2 + c_7k_2 + c_8(n-1) = \Theta(n)\), where \(k_1, k_2\) are some extra steps.

On the other hand, no matter if the input is already sorted or not, the best-case of quicksort is \(\Theta(n \log n)\). Thus, when the input is almost-sorted, insertion-sort beats quicksort.

8.3-3

Base case:
Let \(d = 1\). We only sort on the least significant digit. Obviously, the radix sort works on the least sig. digit.

Hypothesis:
Radix sort works on numbers with arbitrary digits, using an intermediate stable-sort.

We need to assume that radix sort uses a stable sort to sort array \(A\) on digit \(i\) so the intermediate sort is stable. Assume also that, for \(d = k\), the least \(k\)-th sig. digits of the numbers are sorted properly.
Inductive case:
For \( d = k + 1 \), use a stable sort to sort array \( A \) on digit \( k+1 \). Since the least \( k \)-th digits are sorted properly by a stable sort (by assumption), after a stable sort is used on digit \( k+1 \), the order of the least \( k \)-th digits is preserved. Thus, the least \( k+1 \) th digits of \( n \) numbers are in order.

[Not all numbers in array need to have the same number \( d \). The empty digit can be filled with '0']

By induction, radix sort works.

8.6

c)
Suppose there are \( 2n \) elements: \( a_1, a_2, a_3, \ldots, a_{2n} \) such that \( a_1 \leq a_2 \leq \ldots \leq a_{2n} \). Moreover, we have two sorted arrays \( A_1 \) and \( A_2 \), where \( a_i \) in \( A_1 \) and \( a_{i+1} \) in \( A_2 \). Assume that \( a_i \) will not be compared with \( a_{i+1} \). Then,

Case 1: \( a_i \) is compared with an element \( a \neq a_{i+1} \) in \( A_2 \). However, \( a \) must be less than \( a_{i+1} \) because \( a_i \) and \( a_{i+1} \) are consecutive. Thus, all such elements in \( A_2 \) will be merged, and \( a_i \) will finally be compared with \( a_{i+1} \).

Case 2: \( a_{i+1} \) is compared with an element \( b \neq a_i \) in \( A_1 \). However, \( b \) must be less than \( a_i \) because \( a_i \) and \( a_{i+1} \) are consecutive. Thus, all such elements in \( A_1 \) will be merged, and \( a_{i+1} \) will finally be compared with \( a_i \).

By contradiction, if two elements are consecutive in the sorted order and from opposite lists, then they must be compared.

d)
Suppose there are \( 2n \) elements: \( a_1, a_2, a_3, \ldots, a_{2n} \) such that \( a_1 \leq a_2 \leq a_3 \leq \ldots \leq a_{2n} \). The first sorted list \( A_1 = \langle a_1, a_3, \ldots, a_{2n-1} \rangle \) and the second \( A_2 = \langle a_2, a_4, \ldots, a_{2n} \rangle \). By part c, we must have \( a_1 \leq? a_2 \leq? a_3 \leq? \ldots \leq? a_{2n} \) where \( \leq? \) denotes comparison. Therefore, there are \( 2n-1 \) comparisons for \( n \) elements in each list.