1. Stable sorting and in-place sorting.

<table>
<thead>
<tr>
<th></th>
<th>Stable?</th>
<th>In-place?</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge sort</td>
<td>Yes</td>
<td>No</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>Quick sort</td>
<td>No</td>
<td>Yes</td>
<td>$\Theta(n \log n)$ on average (or expected for randomized alg) but $\Theta(n^2)$ worst case.</td>
</tr>
<tr>
<td>Heap sort</td>
<td>No</td>
<td>Yes</td>
<td>$\Theta(n \log n)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>Yes</td>
<td>Yes</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Selection sort</td>
<td>No</td>
<td>Yes</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Counting sort</td>
<td>Yes</td>
<td>No</td>
<td>$\Theta(n + k)$ or $\Theta(n)$ if $k \in O(n)$</td>
</tr>
<tr>
<td>Radix sort</td>
<td>Yes</td>
<td>No</td>
<td>$\Theta(dn/\log n)$</td>
</tr>
</tbody>
</table>

2. Counting sort.

The algorithm will still sort correctly, but it is no longer a stable sorting algorithm. This can be seen using the example provided during lecture.

3. Algorithm design.

This can be achieved using radix sort, using $n$ as the radix. The running time of radix sort is $(n+2^r)d/r$, where $r = \lceil \log_2 n \rceil$ and $d = \lceil \log_2 (n^2) \rceil$. Therefore, $(n+2^r)d/r \approx 4n = \Theta(n)$.

Or equivalently, partition each number into two $\lceil \log_2 n \rceil$ bits numbers; then apply counting sort to the least significant $\lceil \log_2 n \rceil$ bits followed by applying counting sort to the most significant $\lceil \log_2 n \rceil$ bits.

4. Smallest $i$ numbers in sorted order.

Strategy (a) takes $\Theta(n \log n) + \Theta(i) = \Theta(n \log n)$ time with merge sort, or $\Theta(n \log n)$ expected running time with quick sort ($\Theta(n^2)$ in worst case, however).

Strategy (b) takes $\Theta(n) + \Theta(i \log n)$ time.

Strategy (c) takes $\Theta(n) + \Theta(i \log i)$ expected time, but the worst case could be $\Theta(n^2)$.

For large $i$ (e.g., $i = \Theta(n)$), all three strategies have the same asymptotic order in practice (expected running time), $\Theta(n \log n)$. However, strategy (a) with quick sort or (c) could have a worst-case running time of $\Theta(n^2)$ in theory (unlikely to happen).

When $n$ is much larger than $i$, (b) and (c) is in general more efficient than (a) because of their $\Theta(n)$ running time compared to the $\Theta(n \log n)$ running time of (a). Strategy (c) is also expected to be more efficient than (b) in practice, because of the relatively larger constant factor associated with buildHeap.

5. Extra credit.

Here I show an informal analysis where floors and ceilings are ignored. In other words, I assume $n$ is divisible by any number.

When the elements are divided into groups of 7, there will be $n/7$ groups, and there will be at least 4 \[ n/7 \div 2 = 2n/7 \] elements that are no greater than the median of the group medians. (In half of the $n/7$ groups, each group contains four elements that are less than or equal to the corresponding group medians, which are no greater than the median of the medians.) Similarly, there will be at least $2n/7$ elements that are no less than the median of the medians. After partitioning the elements using the median of the medians as the pivot, the larger subarray can have at most $n - 2n/7 = 5n/7$ elements. Therefore, in the worst case, the running time of the SELECT algorithm can be computed using the following recurrence:

\[ T(n) = T(n/7) + T(5n/7) + n, \]

where the first recursive term, $T(n/7)$, is the time needed to find the median
of the n/7 group medians, and the second recursive term, T(5n/7), is the time needed to recursively apply
SELECT to the larger subarray, and n is the time for partition.

Using the recursion-tree method it can be shown that the cost of each level is a decreasing geometric
series, with a decreasing factor 5/7 + 1/7 = 6/7. Therefore, the total sum is Θ(n). This can also be proved
by the substitution method.

If we use groups of 3 instead, there will be n/3 groups, and there will be at least 2 * n/3 / 2 = n/3
elements that are no greater than the median of the medians, and similarly at least n/3 elements that are
no less than the median of the medians. Using the median of the medians as the pivot, we can guarantee
that the larger array can contain at most n - n/3 = 2n/3 elements. Therefore, the worst-case running
time of the algorithm is T(n) = T(n/3) + T(2n/3) + n, where T(n/3) is for finding the median of the
medians and T(2n/3) is for applying SELECT recursively on the larger subarray. This recurrence solves
to T(n) = Θ(n log n).


\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
i = 4 & 6 & 3 & 16 & 11 & 7 & 17 & 14 & 8 \\
\hline
\end{array}
\]

\[k=2, \; i = 4 > k\]
So take the right subarray and new i = 4 - k = 2

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
3 & 6 & 16 & 11 & 7 & 17 & 14 & 8 \\
\hline
\end{array}
\]

\[k=5, \; i = 2 < k\]
So take the left subarray and still have i = 2

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
14 & 11 & 7 & 8 & 16 & 17 \\
\hline
\end{array}
\]

\[k=4, \; i = 2 < k\]
So take the left subarray and still have i = 2

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
11 & 7 & 8 & 14 \\
\hline
\end{array}
\]

\[k=3, \; i = 2 < k\]
So take the left subarray and still have i = 2

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
8 & 7 & 11 \\
\hline
\end{array}
\]

\[k=2, \; i = 2 = k\]
Done! Return 8.