1a. If the array is increasingly sorted and we always select the first element as the pivot, then \( p = q \). Therefore, the first subarray \( A[p, q - 1] \) will be empty and the second subarray \( A[q, r] \) will have the same size as the original array \( A[p, r] \). As a result, the recursive call will not terminate because the size of the subproblem is the same as that of the original problem.

2a. False. With the standard quick sort it would run in \( \Theta(n^2) \) time. With a randomized quick sort it would still take \( \Theta(n \log n) \) time expected and \( \Theta(n^2) \) in the worst case.

2b. True.
2c. False. Randomized quick sort, similar to the standard quick sort, has expected running time of $\Theta(n \log n)$ and worst-case running time of $\Theta(n^2)$.

2d. True.

3a. In the best case, the running time can be defined by $T(n) = 2T(n/3) + n$ which solves to $T(n) \in \Theta(n)$. 

3b. In the worst case, the running time can be defined by $T(n) = 2T(n/2) + n$ which solves to $T(n) \in \Theta(n \log n)$.

3c. The expected running time can be expressed as $\bar{T}(n) = 0.5 \cdot 2T(n/2) + 0.5 \cdot 2T(n/3) + n = \bar{T}(n/2) + \bar{T}(n/3) + n$. 

3d. Using recursion tree it can be seen that $\bar{T}(n) \in \Theta(n)$. This can be proved using substitution method easily (details skipped).

3e. The expected running time can be expressed as $\bar{T}(n) = 0.9 \cdot 2T(n/2) + 0.1 \cdot 2T(n/3) + n = 1.8T(n/2) + 0.2T(n/3) + n$. It is easy to see that $\bar{T}(n) \in \Omega(n)$. Using substitution method it is easy to show that $\bar{T}(n) \in O(n)$: assuming $T(n/2) \leq cn/2$ and $T(n/3) \leq cn/3$, we have 

$$\bar{T}(n) = 1.8\bar{T}(n/2) + 0.2\bar{T}(n/3) + n \leq 1.8cn/2 + 0.2cn/3 + n \leq 29cn/30 + n \leq cn$$

for $c \geq 30$ and $n \geq 0$.

4a. No, this is not a heap. As shown below, the two gray nodes violate the heap property.

![Diagram of a non-heap structure]

4b. A heap of height $h$ has the minimum number of elements when the lowest level of the tree only has one node. Therefore, the minimum number of elements for a heap of height $h$ is $1 + 2 + 2^2 + \ldots + 2^{h-1} + 1 = 2^h$. Similarly, a heap of height $h$ has the maximum number of elements when the lowest level of the tree is full. Therefore, the maximum number of elements for a heap of height $h$ is $1 + 2 + 2^2 + \ldots + 2^{h-1} + 2^h = 2^{h+1} - 1$.

4e. To call $\text{Heapify}(A, i)$, we need to make the assumption that the subtrees rooted at the left and right children of $i$ are both heaps. This assumption is invalid if we start building the heap from $A[1]$.

4f. In either case, $\text{buildHeap}$ will take $\Theta(n)$ time: to build a heap using an array that is sorted in increasing order, we have to sift every elements down (worst-case scenario), which is in $\Theta(n)$; to build a heap using an array that is sorted in decreasing order also takes $\Theta(n)$ time because we have to call $\text{Heapify}$ n/2 times, even though each call to $\text{Heapify}$ will return immediately and takes constant time. After the heap is constructed, each call to $\text{Heapify}$ takes $\Theta(\log n)$ time. Therefore the total cost for heap sort is $\Theta(n \log n)$, regardless of the initial order of the array.

5a. No. Consider an array with three elements: [3, 5, 10]. With the procedure $\text{BuildHeap}$, the final heap is [10, 5, 3]. With the procedure $\text{BuildHeap2}$, the final heap is [10, 3, 5].

5b. $\Theta(n \log n)$. In the worst case, each insertion takes $h$, which is the current height of the heap. A heap with $i$ elements have height $\lceil \log_2 i \rceil$. Therefore, the overall complexity is $\sum_{i=2}^{n} \lceil \log_2 i \rceil$, which is in $\Theta(n \log n)$, as shown below.

$$\sum_{i=2}^{n} \lceil \log_2 i \rceil \leq \sum_{i=2}^{n} \log_2 i \leq \log_2(n!) = \Theta(n \log n)$$

$$\sum_{i=2}^{n} \lceil \log_2 i \rceil \geq \sum_{i=2}^{n} (\log_2 i − 1) \geq \log_2(n!) − n = \Theta(n \log n − n) = \Theta(n \log n)$$

5c. $\text{BuildHeap}$ is asymptotically more efficient than $\text{BuildHeap2}$ as the former is in $\Theta(n)$ and the latter is in $\Theta(n \log n)$.
4c.

4d.

Insert 15

ExtractMax

ChangeKey (A, 2, 8)

This task can be achieved by the following algorithm:

I. randomly choose a bolt.

II. Use the bolt chosen in step 1 to partition the nuts into three groups: those that are smaller than the bolt (group 1), those that are larger than the bolt (group 2), and those that fit the bolt exactly (group 3).

III. Take a nut from the group 3 nuts obtained above in step 2, and use the nut to partition the bolts into three groups: those that are smaller than the nut (group 1), those that are larger than the nut (group 2), and those that fit the nut exactly (group 3).

IV. Match each group 3 nut with a group 3 bolt.

V. Recursively apply steps I-IV to match group 1 nuts with group 1 bolts, and group 2 nuts with group 2 bolts.

The analysis of the above algorithm follows exactly the analysis of randomized quick sort. The bolt chosen in step 1 can be considered as the pivot element. Step 2 and Step 3 take $\Theta(n)$ time to partition the bolts and nuts. Running time of Step 4 depends on how many bolts have equal sizes, and is in $O(n)$ nevertheless. Step 5 makes two recursive calls, similar to the two recursive calls in Quick Sort.

Thus, by choosing a random bolt (pivot) in step 1, the algorithm’s expected running time is $\Theta(n \log n)$. 