1a. The Partition algorithm can result in extremely unbalanced partition when all elements are equal. Run time is $\Theta(n^2)$.

1b. 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.

1c. The Partition algorithm can result in extremely unbalanced partition when all elements are equal. Run time is $\Theta(n^2)$.

1d. 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.
1e. 7 times for the array in 1a (as shown above in the circles) and 9 times for the array in 1c. Partition is called once for each subarray of size larger than one.

2a. Let $P(n)$ be the number of times that the procedure Partition is called, for an input of size $n$. $P(n)$ can be computed recursively as follows.

$$P(n) = \begin{cases} 
P(0) + P(n - 1) + 1 & \text{if } 0 : n - 1 \text{ split,} \\
P(1) + P(n - 2) + 1 & \text{if } 1 : n - 2 \text{ split,} \\
\vdots \\
P(n - 1) + P(0) + 1 & \text{if } n - 1 : 0 \text{ split.}
\end{cases}$$

Therefore, the expected value of $P(n)$, denoted by $\bar{P}(n)$, can be computed by

$$\bar{P}(n) = \frac{1}{n} \sum_{k=0}^{n-1} (\bar{P}(k) + \bar{P}(n - k - 1)) + 1$$

$$= \frac{2}{n} \sum_{k=0}^{n-1} \bar{P}(k) + 1$$

2b. 

Proof. To prove that $\bar{P}(n) \in O(n)$, we need to show that $\bar{P}(n) \leq cn$ for some $c$ and sufficiently large $n$. Using the substitution method, assume that this is true for all $k < n$, i.e., $\bar{P}(k) \leq ck$ for some $c$. Substitute $ck$ for $\bar{P}(k)$ in the recurrence, we have

$$\bar{P}(n) = \frac{2}{n} \sum_{k=0}^{n-1} \bar{P}(k) + 1$$

$$\leq \frac{2}{n} \sum_{k=0}^{n-1} ck + 1$$

$$= \frac{2c}{n} \sum_{k=0}^{n-1} k + 1$$

$$= \frac{2cn(n-1)}{2} + 1$$

$$= cn - c + 1$$

$$\leq cn \quad \text{if } c \geq 1.$$ 

Therefore, by definition, $\bar{P}(n) \in O(n)$. 

2c. Each call to Partition will cause at least one element (the pivot) to be sorted into the right position. In addition, for an array of size $> 3$, each call to Partition will cause at most two elements to be sorted into the right positions (the pivot element, plus one more element if it is a 1 : $n - 1$ or $n - 1 : 1$ split. Therefore, the total number of partitions to sort $n$ numbers should be between $n$ and $n/2$. 

3a. 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.

3b. True.

3c. 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 $\Theta(n^2)$. 

3d. True.


Answer: 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 $\Theta(n \log n)$.

Part II

Sorry about the numbering. I messed up when splitting the hw into two parts.

3a. No, this is not a heap. As shown below, the two gray nodes violate the heap property.

```
  23
  /|
 5 6 17
 /|
7 13 14
 /|
  10 12
```

3b. 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 $1 + 2 + 2^2 + \ldots + 2^{h-1} + 2^h = 2^{h+1} - 1$.

3c.
3d.

3e. To call 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].

3f. In either case, 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 Heapify $n/2$ times, even though each call to Heapify will return immediately and takes constant time. After the heap is constructed, each call to 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.

4a. No. Consider an array with three elements: $[3, 5, 10]$. With the procedure BuildHeap, the final heap is $[10, 5, 3]$. With the procedure BuildHeap2, the final heap is $[10, 3, 5]$.

4b. $\Theta(n \log n)$. In the worst case, each insertion takes time $h$, 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.

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

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

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