Solution to Exercise 7-1

a. The array and auxiliary values for all iterations of the while loop:

$x = 13$

$i = p - 1$

$j = r + 1$

Iteration 1 of while loop:

p,i										j	r
13	19	9	5	12	8	7	4	11	2	6	21

p,i										j	r
6	19	9	5	12	8	7	4	11	2	13	21

Iteration 2 of while loop:

p	i								j		r
6	19	9	5	12	8	7	4	11	2	13	21

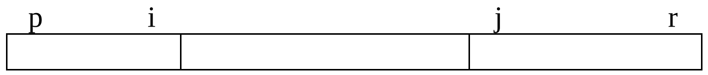
p	i								j		r
6	2	9	5	12	8	7	4	11	19	13	21

Iteration 3 of while loop:

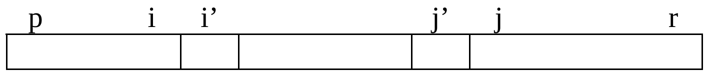
p								j	i		r
6	2	9	5	12	8	7	4	11	19	13	21

So the while loop terminates and return the value of *j*.

b & c. The index *i* and *j* have initial values as p-1 and r+1. At the first iteration (k=1) of while loop, $p \le j \le r$ and $i = p$.
For k>1, at the beginning of the kth iteration, the following holds: $p \le i < j < r$, and every element in $A[p,\dots,i] \le x$ and $A[j,\dots,r] \ge x$.
At the beginning of the kth iteration,



Assuming that the algorithm does not terminate by the end of this iteration, we have at the beginning of the (k+1)th iteration,



where *i'* and *j'* are the new values of *i* and *j* and $i' < j'$. Every element in $A[i+1,\dots,i'] \le x$ and every element in $A[j',\dots,j-1] \ge x$, and $p \le i \le i' < j' \le j \le r$.

If the algorithm terminates at the end of kth iteration, then $i \le j' \le i' \le j \Rightarrow p \le i \le j' \le i' \le j < r$. Hence the algorithm will never access elements outside $A[p,\dots,r]$.

d. The while loop exits with $p \le j \le i < r$, and every element in $A[p,\dots,j] \le x$ and $A[j+1,\dots,r] \ge x$. Therefore, when HOARE-PARTITION terminates, every element in $A[p..j]$ is less than or equal to every element in $A[j+1..r]$.

e. The following procedure implements quicksort using the Hoare partitioning strategy.

```
QUICKSORT(A, p, r)
if p<r
   then q ← HOARE-PARTITION(A, p, r)
        QUICKSORT(A, p, q)
        QUICKSORT(A, q+1, r)
```

Solution to Exercise 8.3-3

Basis: If $d = 1$, there's only one digit, so sorting on that digit sorts the array.

Inductive step: Assuming that radix sort works for $d - 1$ digits, we'll show that it works for d digits.
Radix sort sorts separately on each digit, starting from digit 1. Thus, radix sort of d digits, which sorts on digits $1, \dots, d$ is equivalent to radix sort of the low-order $d - 1$ digits followed by a sort on digit d. By our induction hypothesis, the sort of the low-order $d - 1$ digits works, so just before the sort on digit d, the elements are in order according to their low-order $d - 1$ digits.
The sort on digit d will order the elements by their dth digit. Consider two elements, a and b, with dth digits a_d and b_d respectively.

- If $a_d < b_d$, the sort will put a before b, which is correct, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will put a after b, which is correct, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order they were in, because it is stable. But that order is already correct, since the correct order of a and b is determined by the low-order $d - 1$ digits when their dth digits are equal, and the elements are already sorted by their low-order $d - 1$ digits. If the intermediate sort were not stable, it might rearrange elements whose dth digits were equal—elements that *were* in the right order after the sort on their lower-order digits.

Solution to Exercise 8.4-2

The worst-case running time for the bucket-sort algorithm occurs when the assumption of uniformly distributed input does not hold. If, for example, all the input ends up in the first bucket, then in the insertion sort phase it needs to sort all the input, which takes $O(n^2)$ time.
A simple change that will preserve the linear expected running time and make the worst-case running time $O(n \lg n)$ is to use a worst-case $O(n \lg n)$-time algorithm like merge sort instead of insertion sort when sorting the buckets.

Solution to Problem 8-3

a. The usual, unadorned radix sort algorithm will not solve this problem in the required time bound. The number of passes, d, would have to be the number of digits in the largest integer. Suppose that there are m integers; we always have $m \le n$. In the worst case, we would have one integer with $n/2$ digits and $n/2$ integers with one digit each. We assume that the range of a single digit is constant. Therefore, we would have $d = n/2$ and $m = n/2 + 1$, and so the running time would be $\Theta(dm) = \Theta(n^2)$.
Let us assume without loss of generality that all the integers are positive and have no leading zeros. (If there are negative integers or 0, deal with the positive numbers, negative numbers, and 0 separately.) Under this assumption, we can observe that integers with more digits are always greater than integers with fewer digits. Thus, we can first sort the integers by number of digits (using counting sort), and then use radix sort to sort each group of integers with the same length. Noting that each integer has between 1 and n digits, let m_i be the number of integers with i digits, for $i = 1, 2, \dots, n$. Since there are n digits altogether, we have $\sum_{i=1}^{n} i \cdot m_i = n$

It takes $O(n)$ time to compute how many digits all the integers have and, once the numbers of digits have been computed, it takes $O(m + n) = O(n)$ time to group the integers by number of digits. To sort the group with m_i digits by radix sort takes $\Theta(i \cdot m_i)$ time. The time to sort all groups, therefore, is

$$\begin{aligned}\sum_{i=1}^{n} \Theta(i \cdot m_i) &= \Theta(\sum_{i=1}^{n} i \cdot m_i) \\ &= \Theta(n).\end{aligned}$$

b. One way to solve this problem is by a radix sort from right to left. Since the strings have varying lengths, however, we have to pad out all strings that are shorter than the longest string. The padding is on the right end of the string, and it's with a special character that is lexicographically less than any other character (e.g., in C, the character `'\0'` with ASCII value 0). Of course, we don't have to actually change any string; if we want to know the j th character of a string whose length is k, then if $j > k$, the j th character is the pad character.
Unfortunately, this scheme does not always run in the required time bound. Suppose that there are m strings and that the longest string has d characters. In the worst case, one string has $n/2$ characters and, before padding, $n/2$ strings have one character each. As in part (a), we would have $d = n/2$ and $m = n/2 + 1$. We still have to examine the pad characters in each pass of radix sort, even if we don't actually create them in the strings. Assuming that the range of a single character is constant, the running time of radix sort would be $\Theta(dm) = \Theta(n^2)$.

To solve the problem in $O(n)$ time, we use the property that, if the first letter of string x is lexicographically less that the first letter of string y, then x is lexicographically less than y, regardless of the lengths of the two strings. We take advantage of this property by sorting the strings on the first letter, using counting sort. We take an empty string as a special case and put it first. We gather together all strings with the same first letter as a group. Then we recurse, *within each group*, based on each string with the first letter removed.

The correctness of this algorithm is straightforward. Analyzing the running time is a bit trickier. Let us count the number of times that each string is sorted by a call of counting sort. Suppose that the ith string, s_i, has length l_i. Then s_i is sorted by at most $l_i + 1$ counting sorts. (The "+1" is because it may have to be sorted as an empty string at some point; for example, `ab` and `a` end up in the same group in the first pass and are then ordered based on `b` and the empty string in the second pass. The string `a` is sorted its length, 1, time plus one more time.) A call of counting sort on t strings takes $\Theta(t)$ time (remembering that the number of different characters on which we are sorting is a constant.) Thus, the total time for all calls of counting sort is

$$\begin{aligned} O\left(\sum_{i=1}^{m}(l_i+1)\right) &= O\left(\sum_{i=1}^{m} l_i + m\right) \\ &= O(n+m) \\ &= O(n), \end{aligned}$$

where the second line follows from $\sum_{i=1}^{m} l_i = n$, and the last line is because $m \leq n$.