Exercise 8.1-1

We can construct the graph whose vertex set is the indices, and we place an edge between any two indices that are compared on the shortest path. We need this graph to be connected, because otherwise we could run the algorithm twice, once with everything in one component less than the other component, and a second time with the everything in the second component larger. As long as we maintain the same relative ordering of the elements in each component, the algorithm will take exactly the same path, and so produce the same result. This means that there will be no difference in the output, even though there should be. For a graph on n vertices, it is a well known that at least $n-1$ edges are necessary for it to be connected, as the addition of an edge can reduce the number of connected components by at least one, and the graph with no edges has n connected components.

So, it will have depth at least $n-1$.

Exercise 8.2-1

We have that $C = \langle 2, 4, 6, 8, 9, 9, 11 \rangle$. Then, after successive iterations of the loop on lines 10-12, we have $B = \langle \, , \, , \, , \, , 2, \, , \, , \, , \, \rangle$,$B = \langle \, , \, , \, , \, , 2, \, , 3, \, , \, , \, \rangle$,$B = \langle \, , \, , \, , 1, \, , 2, \, , 3, \, , \, , \, \rangle$, and at the end, $B = \langle 0, 0, 1, 1, 2, 2, 3, 3, 4, 6, 6 \rangle$

Exercise 8.3-1

Starting with the unsorted words on the left, and stable sorting by progressively more important positions.

COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	RUG	SEA	DIG
MOB	DOG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

Solution to Exercise 9.1-1

The smallest of n numbers can be found with $n-1$ comparisons by conducting a tournament as follows: Compare all the numbers in pairs. Only the smaller of each pair could possibly be the smallest of all n, so the problem has been reduced to that of finding the smallest of $\lceil n/2 \rceil$ numbers. Compare those numbers in pairs, and so on, until there's just one number left, which is the answer.

To see that this algorithm does exactly $n-1$ comparisons, notice that each number except the smallest loses exactly once. To show this more formally, draw a binary tree of the comparisons the algorithm does. The n numbers are the leaves, and each number that came out smaller in a comparison is the parent of the two numbers that were compared. Each non-leaf node of the tree represents a comparison, and there are $n-1$ internal nodes in an n-leaf full binary tree (see Exercise (B.5-3)), so exactly $n-1$ comparisons are made.

In the search for the smallest number, the second smallest number must have come out smallest in every comparison made with it until it was eventually compared with the smallest. So the second smallest is among the elements that were compared with the smallest during the tournament. To find it, conduct another tournament (as above) to find the smallest of these numbers. At most $\lceil \lg n \rceil$ (the height of the tree of comparisons) elements were compared with the smallest, so finding the smallest of these takes $\lceil \lg n \rceil - 1$ comparisons in the worst case.

The total number of comparisons made in the two tournaments was

$$n - 1 + \lceil \lg n \rceil - 1 = n + \lceil \lg n \rceil - 2$$

in the worst case.

Solution to Exercise 9.2-3

```
ITERATIVE-RANDOMIZED-SELECT(A, p, r, i)
  while p < r
      q = RANDOMIZED-PARTITION(A, p, r)
      k = q − p + 1
      if i == k
          return A[q]
      elseif i < k
          r = q − 1
      else p = q + 1
          i = i − k
  return A[p]
```

Exercise 9.3-7

Find the $n/2 - k/2$ largest element in linear time. Partition on that element. Then, find the k largest element in the bigger subarray formed from the partition. Then, the elements in the smaller subarray from partitioning on this element are the desired k numbers.

Solution to Exercise 9.3-8

Let's start out by supposing that the median (the lower median, since we know we have an even number of elements) is in X. Let's call the median value m, and let's suppose that it's in $X[k]$. Then k elements of X are less than or equal to m and $n-k$ elements of X are greater than or equal to m. We know that in the two arrays combined, there must be n elements less than or equal to m and n elements greater than or equal to m, and so there must be $n-k$ elements of Y that are less than or equal to m and $n-(n-k)=k$ elements of Y that are greater than or equal to m.

Thus, we can check that $X[k]$ is the lower median by checking whether $Y[n-k] \le X[k] \le Y[n-k+1]$. A boundary case occurs for $k=n$. Then $n-k=0$, and there is no array entry $Y[0]$; we only need to check that $X[n] \le Y[1]$.

Now, if the median is in X but is not in $X[k]$, then the above condition will not hold. If the median is in $X[k']$, where $k' < k$, then $X[k]$ is above the median, and $Y[n-k+1] < X[k]$. Conversely, if the median is in $X[k'']$, where $k'' > k$, then $X[k]$ is below the median, and $X[k] < Y[n-k]$.

Thus, we can use a binary search to determine whether there is an $X[k]$ such that either $k < n$ and $Y[n-k] \le X[k] \le Y[n-k+1]$ or $k=n$ and $X[k] \le Y[n-k+1]$; if we find such an $X[k]$, then it is the median. Otherwise, we know that the median is in Y, and we use a binary search to find a $Y[k]$ such that either $k < n$ and $X[n-k] \le Y[k] \le X[n-k+1]$ or $k=n$ and $Y[k] \le X[n-k+1]$; such a $Y[k]$ is the median. Since each binary search takes $O(\lg n)$ time, we spend a total of $O(\lg n)$ time.

Here's how we write the algorithm in pseudocode:

```
TWO-ARRAY-MEDIAN(X, Y)
  n = X.length                    // n also equals Y.length
  median = FIND-MEDIAN(X, Y, n, 1, n)
  if median == NOT-FOUND
      median = FIND-MEDIAN(Y, X, n, 1, n)
  return median

FIND-MEDIAN(A, B, n, low, high)
  if low > high
      return NOT-FOUND
  else k = ⌊(low + high)/2⌋
      if k == n and A[n] ≤ B[1]
          return A[n]
      elseif k < n and B[n − k] ≤ A[k] ≤ B[n − k + 1]
          return A[k]
      elseif A[k] > B[n − k + 1]
          return FIND-MEDIAN(A, B, n, low, k − 1)
      else return FIND-MEDIAN(A, B, n, k + 1, high)
```