Recall that procedure *partition* in Algorithm 2.7, which is used in Quicksort (Algorithm 2.6), partitions an array so that all keys smaller than some pivot item come before it in the array and all keys larger than that pivot item come after it. The slot at which the pivot item is located is called the *pivotpoint*. We can solve the Selection problem by partitioning until the pivot item is at the kth slot. We do this by recursively partitioning the left subarray (the keys smaller than the pivot item) if k is less than pivotpoint, and by recursively partitioning the right subarray if k is greater than *pivotpoint*. When $k = pivotpoint$, we're done. This divide-and-conquer algorithm solves the problem by this method.

Algorithm 8.5 **Selection**

Problem: Find the kth-smallest key in array S of n distinct keys.

Inputs: positive integers n and k where $k \leq n$, array of distinct keys S indexed from 1 to n.

Outputs: the kth-smallest key in S. It is returned as the value of function *selection*.

```
keytype selection (index low, index high, index k)
{
  index pivotpoint;

  if (low == high)
     return S[low];
  else{
     partition (low, high, pivotpoint);
     if (k == pivotpoint)
        return S[pivotpoint];
     else if (k < pivotpoint)
        return selection(low, pivotpoint - 1, k);
     else
        return selection(pivotpoint + 1, high, k);
  }
}

void partition (index low, index high,  // This is the same
                index& pivotpoint)      // routine that appears
                                        // in Algorithm 2.7.
{
  index i, j;
  keytype pivotitem;

  pivotitem = S[low];                   // Choose first item for
  j = low;                              // pivotitem.
```

```
    for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem){
            j++;
            exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];    // Put pivotitem at
                                          // pivotpoint.
}
```

As with our recursive functions in previous chapters, n and S are not inputs to function *selection*. The top-level call to that function would be

$$kthsmallest = selection\,(1, n, k)\,.$$

As in Quicksort (Algorithm 2.6), the worst case occurs when the input to each recursive call contains one less item. This happens, for example, when the array is sorted in increasing order and $k = n$. Algorithm 8.5 therefore has the same worst-case time complexity as Algorithm 2.6, which means that the *worst-case time complexity of the number of comparisons of keys done by Algorithm 8.5* is given by

$$W\,(n) = \frac{n\,(n-1)}{2}.$$

Although the worst case is the same as that of Quicksort, we show next that Algorithm 8.5 performs much better on the average.

Analysis of Algorithm 8.5

▶ **Average-Case Time Complexity (Selection)**

Basic operation: the comparison of $S\,[i]$ with *pivotitem* in *partition*.

Input size: n, the number of items in the array.

We assume that all inputs are equally likely. This means that we assume that all values of k are entered with equal frequency and all values of *pivotpoint* are returned with equal frequency. Let p stand for *pivotpoint*. There are n outcomes for which there is no recursive call (that is, if $p = k$ for $k = 1, 2, \ldots, n$). There are two outcomes for which 1 is the input size in the first recursive call (that is, if $p = 2$ with $k = 1$, or $p = n - 1$ with $k = n$). There are $2(2) = 4$ outcomes for which 2 is the input size in the first recursive call (that is, if $p = 3$ with $k = 1$ or 2, or $p = n - 2$