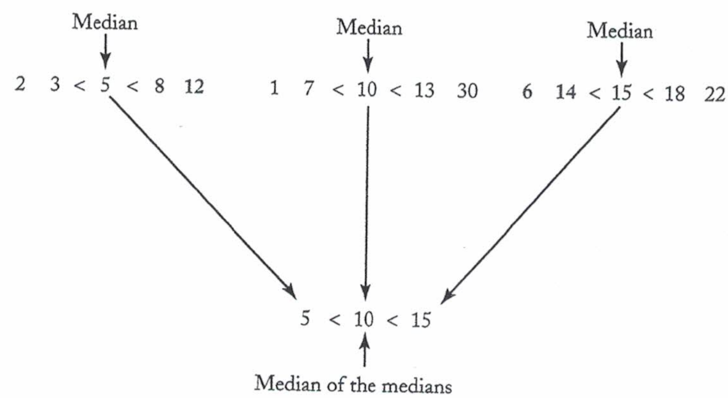

Figure 8.12 Each bar represents a key. We do not know if the boldfaced keys are less than or greater than the median of the medians.

the keys to the left of the largest median (keys 6 and 14) could lie on either side of the median of medians. Notice that there are

$$2\left(\frac{15}{5}-1\right)$$

keys that could be on either side of the median of the medians. It is not hard to see that, whenever n is an odd multiple of 5, there are

$$2\left(\frac{n}{5}-1\right)$$

keys that could lie on either side of the median of the medians. Therefore, there are at most

$$\underbrace{\frac{1}{2}\left[n-1-2\left(\frac{n}{5}-1\right)\right]}_{\text{Number of keys we know are on one side}}+2\left(\frac{n}{5}-1\right)=\frac{7n}{10}-\frac{3}{2}$$

keys on one side of the median of the medians. We return to this result when we analyze the algorithm that uses this strategy. First we present the algorithm.

Algorithm 8.6

Selection Using the Median

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

Inputs: positive integers n and k where $k \le n$, 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 *select*.

```
keytype select (int n,
                keytype S[],
                index k)
```

```
{
  return selection2(S, 1, n, k);
}

keytype selection2 (keytype S[],
                    index low, index high, index k)
{
  if (high == low)
     return S[low];
  else{
     partition2(S, low, high, pivotpoint);
     if (k == pivotpoint)
        return S[pivotpoint];
     else if (k < pivotpoint)
        return selection2(S, low, pivotpoint − 1, k);
     else
        return selection2(S, pivotpoint + 1, high, k);
  }
}

void partition2 (keytype S[],
                 index low, index high,
                 index& pivotpoint)
{
  const arraysize = high − low + 1;
  const r = ⌈arraysize / 5⌉;
  index i, j, mark, first, last;
  keytype pivotitem, T[1..r];

  for (i = 1; i <= r, i++){
     first = low + 5*i − 5;
     last = minimum(low + 5*i − 1, arraysize);
     T[i] = median of S[first] through S[last];
  }
  pivotitem = select(r, T, ⌊(r + 1) / 2⌋ ); // Approximate the median.
  j = low;
  for (i = low; i <= high; i++)
     if (S[i] == pivotitem){
        exchange S[i] and S[j];
        mark = j;                               // Mark where pivotitem placed.
        j++;
     }
     else if (S[i] < pivotitem){
        exchange S[i] and S[j];
        j++;
     }
```

G PROBLEM


```
    pivotpoint = j − 1;
    exchange S[mark] and S[pivotpoint];      // Put pivotitem at pivotpoint.
}
```

In Algorithm 8.6, unlike our other recursive algorithms, we show a simple function that calls our recursive function. The reason is that this simple function needs to be called in two places with different inputs. That is, it is called in procedure *partition2* with T being the input, and globally as follows:

$$kthsmallest = select\,(n, S, k)\,.$$

We also made the array an input to the recursive function *selection2* because the function is called to process both the global array S and the local array T.

Next we analyze the algorithm.

Analysis of Algorithm 8.6

▶ Worst-Case Time Complexity (Selection Using the Median)

Basic operation: the comparison of $S[i]$ with pivotitem in *partition2*.

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

For simplicity, we develop a recurrence assuming that n is an odd multiple of 5. The recurrence approximately holds for n in general. The components in the recurrence are as follows.

- The time in function *selection2* when called from function *selection2*. As already discussed, if n is an odd multiple of 5, at most

$$\frac{7n}{10} - \frac{3}{2} \text{ keys}$$

  end up on one side of *pivotpoint*, which means that this is the worst-case number of keys in the input to this call to *selection2*.
- The time in function *selection2* when called from procedure *partition2*. The number of keys in the input to this call to *selection2* is $n/5$.
- The number of comparisons required to find the medians. As mentioned previously, the median of five numbers can be found by making six comparisons. When n is a multiple of 5, the algorithm finds the median of exactly $n/5$ groups of five numbers. Therefore, the total number of comparisons required to find the medians is $6n/5$.
- The number of comparisons required to partition the array. This number is n (assuming an efficient implementation of the comparison).

median.

item placed.