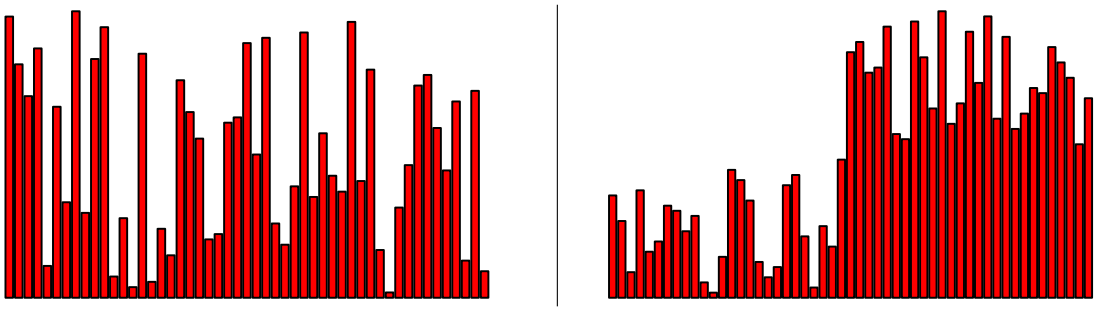

INPUT OUTPUT

14.3 Median and Selection

Input description: A set of n numbers or keys, and an integer k.

Problem description: Find the key smaller than exactly k of the n keys.

Discussion: Median finding is an essential problem in statistics, where it provides a more robust notion of average than the *mean*. The mean wealth of people who have published research papers on sorting is significantly affected by the presence of one William Gates [GP79], although his effect on the *median* wealth is merely to cancel out one starving graduate student.

Median finding is a special case of the more general *selection* problem, which asks for the kth element in sorted order. Selection arises in several applications:

- *Filtering outlying elements* – In dealing with noisy data, it is usually a good idea to throw out (say) the 10% largest and smallest values. Selection can be used to identify the items defining the 10^{th} and 90^{th} percentiles, and the outliers then filtered out by comparing each item to the two selected bounds.
- *Identifying the most promising candidates* – In a computer chess program, we might quickly evaluate all possible next moves, and then decide to study the top 25% more carefully. Selection followed by filtering is the way to go.
- *Deciles and related divisions* – A useful way to present income distribution in a population is to chart the salary of the people ranked at regular intervals, say exactly at the 10th percentile, 20th percentile, etc. Computing these values is simply selection on the appropriate position ranks.
- *Order statistics* – Particularly interesting special cases of selection include finding the smallest element ($k = 1$), the largest element ($k = n$), and the median element ($k = n/2$).

The mean of n numbers can be computed in linear time by summing the elements and dividing by n. However, finding the median is a more difficult problem. Algorithms that compute the median can readily be generalized to arbitrary selection. Issues in median finding and selection include:

- *How fast does it have to be?* – The most elementary median-finding algorithm sorts the items in $O(n \lg n)$ time and then returns the item occupying the $(n/2)$nd position. The good thing is that this gives much more information than just the median, enabling you to select the kth element (for any $1 \leq k \leq n$) in constant time after the sort. However, there are faster algorithms if all you want is the median.

  In particular, there is an $O(n)$ *expected*-time algorithm based on quicksort. Select a random element in the data set as a pivot, and use it to partition the data into sets of elements less than and greater than the pivot. From the sizes of these sets, we know the position of the pivot in the total order, and hence whether the median lies to the left or right of this point. Now we recur on the appropriate subset until it converges on the median. This takes (on average) $O(\lg n)$ iterations, with the cost of each iteration being roughly half that of the previous one. This defines a geometric series that converges to a linear-time algorithm, although if you are very unlucky it takes the same time as quicksort, $O(n^2)$.

  More complicated algorithms can find the median in worst-case linear time. However, the expected-time algorithm will likely win in practice. Just make sure to select random pivots to avoid the worst case.

- *What if you only get to see each element once?* – Selection and median finding become expensive on large datasets because they typically require several passes through external memory. In data-streaming applications, the volume of data is often too large to store, making repeated consideration (and thus exact median finding) impossible. Much better is computing a small summary of the data for future analysis, say approximate deciles or frequency moments (where the kth moment of stream x is defined as $F_k = \sum_i x_i^k$).

  One solution to such a problem is random sampling. Flip a coin for each value to decide whether to store it, with the probability of heads set low enough that you won't overflow your buffer. Likely the median of your samples will be close to that of the underlying data set. Alternately, you can devote some fraction of memory to retaining (say) decile values of large blocks, and then combine these decile distributions to yield more refined decile bounds.

- *How fast can you find the mode?* – Beyond mean and median lies a third notion of average. The *mode* is defined to be the element that occurs the greatest number of times in the data set. The best way to compute the mode sorts the set in $O(n \lg n)$ time, which places all identical elements next to each other. By doing a linear sweep from left to right on this sorted set, we can

count the length of the longest run of identical elements and hence compute the mode in a total of $O(n \lg n)$ time.

In fact, there is no faster worst-case algorithm possible to compute the mode, since the problem of testing whether there exist two identical elements in a set (called element uniqueness) can be shown to have an $\Omega(n \log n)$ lower bound. Element uniqueness is equivalent to asking whether the mode occurs more than once. Possibilities exist, at least theoretically, for improvements when the mode is large by using fast median computations.

Implementations: The C++ *Standard Template Library* (STL) provides a general selection method (`nth_element`) implemented using the linear expected-time algorithm. It is available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

Notes: The linear expected-time algorithm for median and selection is due to Hoare [Hoa61]. Floyd and Rivest [FR75] provide an algorithm that uses fewer comparisons on average. Good expositions on linear-time selection include [AHU74, BvG99, CLRS01, Raw92], with [Raw92] being particularly enlightening.

Streaming algorithms have extensive applications to large data sets, and are well surveyed by Muthukrishnan [Mut05].

A sport of considerable theoretical interest is determining *exactly* how many comparisons are sufficient to find the median of n items. The linear-time algorithm of Blum et al. [BFP+72] proves that $c \cdot n$ suffice, but we want to know what c is. Dor and Zwick [DZ99] proved that $2.95n$ comparisons suffice to find the median. These algorithms attempt to minimize the number of element comparisons but not the total number of operations, and hence do not lead to faster algorithms in practice. They also hold the current best lower bower bound of $(2 + \epsilon)$ comparisons for median finding [DZ01].

Tight combinatorial bounds for selection problems are presented in [Aig88]. An optimal algorithm for computing the mode is given by [DM80].

Related Problems: Priority queues (see page 373), sorting (see page 436).