programming pearls

by Jon Bentley

SELECTION

Suppose you have a list of heights of 101 people. It isn't too hard to find the tallest or the shortest on the list, but how would you identify the most mediocre person (speaking heightwise, of course)? That is, how would you find the person on the list who is taller than the 50 shortest people and shorter than the 50 tallest?

The next section describes the problem around which this column is built: selecting the K^{th}-smallest member in a set of N elements. A program for the task is derived in the following section, and the subsequent section analyzes its (rapid) running time.

The Problem

This excerpt from a table entitled "Density of Population by States" gives the 1980 figures in persons per square mile.

Name	Population Density
West Virginia	80.8
North Carolina	120.4
Virginia	134.7
Pennsylvania	264.3
New York	370.6
Maryland	428.7
Connecticut	637.8
New Jersey	986.2
District of Columbia	10,132.3

If you had to choose a single number to characterize the "typical" density in these nine contiguous areas, what would it be? The average (arithmetic mean) value is 1461.8, but that seems too high: it is greater than eight of the nine values. New York's "middle" value of 370.6 seems more representative; it is the fifth largest of the nine. Statisticians refer to the $M + 1^{st}$-smallest element in a set of $2M + 1$ elements as the *median*. We'll use medians (and other quantiles) later in this column to analyze data on the run time of the selection algorithm.

Computer scientists use medians in many "divide-and-conquer" algorithms. The median partitions a set into two subsets which the algorithm then processes recursively; Problem 8 calls for an algorithm with this structure. Furthermore, the selection problem is a practical introduction to the theoretical field of comparison problems; Problem 9 presents two other representative problems.

Let's turn now from the abstract world of sets to the concrete world of programs. The input to our selection routine will be the positive integer N, the array $X[1..N]$, and the positive integer $K \le N$. The program must permute the array so that $X[1..K-1] \le X[K] \le X[K+1..N]$. At that point, the K^{th}-smallest element in the set resides in its proper position, $X[K]$.

The Program

A simple selection program merely sorts the array X. Unfortunately, this straightforward solution requires $O(N \log N)$ time. In this section we'll study a faster algorithm due to C. A. R. Hoare. His method selects the K^{th}-smallest element in just $O(N)$ average time. Hoare called his program *Find*; I'll refer to the implementation in this column as *Select*.

Hoare's selection algorithm is closely related to his Quicksort program, which was described in the April 1984 column. That divide-and-conquer algorithm can be (roughly) sketched as

```
procedure QSort(set S): sequence
  if size(S) <= 1 then
      return the element in S
    else
      partition S around a random
        element T into subsets A
        and B such that elements
        in A are less than T and
        elements in B are greater
        than T
      return QSort(A) followed by
        T followed by QSort(B)
```

The procedure's input is a set and its output is the sequence of elements in sorted order. Both input and output structures can be efficiently implemented in a single array: the elements in the subvector $X[L..U]$ are represented by the two integers L and U.

The Select algorithm has the same structure as Quicksort. Given $L \le K \le U$, its first step in finding the proper occupant of X[K] is to partition the array around a random element. While Quicksort then recursively operates on both subsequences, Select saves time by recurring only on the side that contains K. Figure 1 shows Select as it finds the median of a 21-element array. Each level in the picture represents a stage of the algorithm, and the array's final configuration is described in the last level. The partitioning element is circled; elements to its left have lesser values, while elements to its right are greater than or equal to the partitioning value.

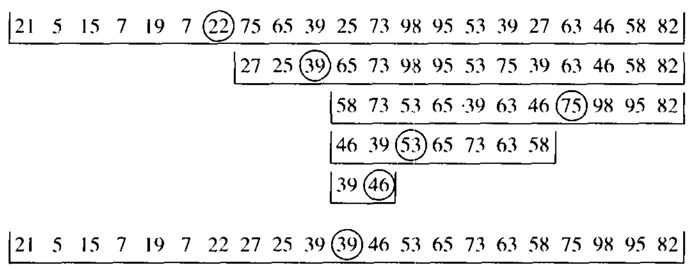


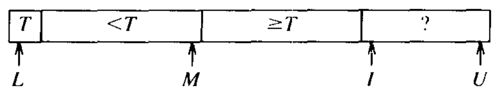
FIGURE 1. Finding the Median of a 21-Element Array

An iterative selection algorithm can be sketched as follows.

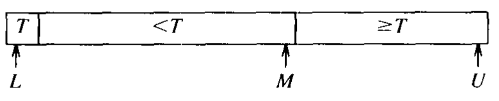
```
set range to entire array
while range is large do
    partition range
    repeat on proper subrange
```

We'll first review the partitioning code described in the April 1984 column on Quicksort, and then turn to the complete algorithm.

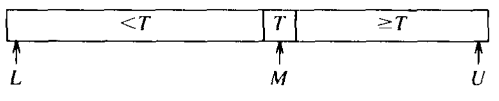
The routine partitions the array $X[L..U]$ around the value $T = X[L]$. After the $I - 1$st step of the iteration, the loop invariant is depicted as



The iterative step inspects the Ith element. If $X[I] \geq T$ then the invariant remains true. When $X[I] < T$, we regain the invariant by incrementing M to index the new location of the small element, and then swapping $X[M]$ with $X[I]$. The loop terminates with $I = U + 1$, leaving



We then swap X[L] with X[M] to give



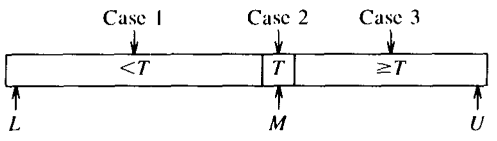
That final swap ensures that we can operate next on $L..M - 1$ or $M + 1..U$. In both cases, we exclude $X[M]$, and thereby avoid an infinite loop.

Partitioning around the first element in the array can require excessive time for some common inputs—for instance, arrays that are already sorted. We do better to choose a partitioning element at random. We'll accomplish this by swapping $X[L]$ with a random entry in $X[L..U]$.[1] The complete partitioning code is

```
Swap(X[L], X[RandInt(L,U)])
M := L
for I := L+1 to U do
    if X[I] < X[L] then
        M := M+1
        Swap(X[M], X[I])
Swap(X[L], X[M])
```

Upon termination, we know that $X[L..M - 1] < X[M] \leq X[M + 1..U]$.

With this partitioning code in hand, we can turn our attention to the complete selection subroutine. Our first version is recursive: *Select*(L, U, K) partitions the array $X[L..U]$ so that $X[L..K - 1] \leq X[K] \leq X[K + 1..U]$. If $L \geq U$ then the subarray contains at most one element, so we can halt. Otherwise, we partition the array around the element T, which is placed in $X[M]$. The position of K relative to M gives three cases:



Case 2 is the easiest: when $K = M$, the K^{th}-smallest element is in its final place and the program is finished. When $K < M$ we have Case 1: the K^{th}-smallest element can't be in $X[M..U]$, so we exclude that range by recursively operating on the range $L..M - 1$. Case 3 is similar, and the recursive routine is sketched as

```
procedure Select(L, U, K)
      pre L <= K <= U
      post X[L..K-1] <= X[K] <= X[K+1..U]
  if L < U then
      /* Partition X[L..U] so that
         X[L..M-1] <= X[M] <= X[M+1..U] */
      if      K < M then Select(L, M-1, K)
      else if K > M then Select(M+1, U, K)
      /* else K = M so finished */
```

Since $X[M]$ is excluded by each recursive call, the program can't have an infinite loop.

[1] If you don't have a *RandInt* function, you can make one using a function *Rand* that returns a random real distributed uniformly in [0, 1) by the expression $L + int(Rand \times (U + 1 - L))$. In the unlikely event that your system doesn't have that routine, consult Knuth's *Seminumerical Algorithms*. But whether you use a system routine or make your own, be careful that *RandInt* returns a value in the range $L..U$—a value out of range is an insidious bug.

The recursive calls in the above procedure are of a special form called *tail recursion*: the call is always the last action in a procedure. A tail-recursive procedure can always be transformed into an equivalent procedure with a `while` loop; Program 1 is an iterative selection subroutine. It uses L and U as local variables, maintaining the relation that $L \le K \le U$ until the final step. After partitioning around X[M], the code adjusts L or U (and sometimes both) to narrow the range $L\,..\,U$.

```
procedure Select(K)
      pre:  1 <= K <= N
      post: X[1..K-1] <= X[K] <= X[K+1..N]
  L := 1; U := N
  while L < U do
      /* Invariant: X[1..L-1] <= X[L..U]
                                <= X[U+1..N] */
      Swap(X[L], X[RandInt(L,U)])
      M := L
      for I := L+1 to U do
          /* Invariant: X[L+1..M] < X[L]
             and X[M+1..I-1] >= X[L] */
          if X[I] < X[L] then
             M := M+1
             Swap(X[M], X[I])
      Swap(X[L], X[M])
      /* X[1..L-1] <= X[L..U] <= X[U+1..N]
         and X[L..M-1] < X[M] <= X[M+1..U] */
      if K <= M then U := M-1
      if K >= M then L := M+1
```

PROGRAM 1. Hoare's Selection Algorithm

Program 1 is the Select algorithm we'll study in the rest of this column, and it is fine for typical day-to-day use. There are, however, several improvements one should incorporate into an industrial-strength selection routine. Speedups to the partitioning code are described in the April 1984 column (see especially Problems 3 and 4), and Problem 1 discusses superior methods for choosing the partitioning element.

Analysis of Run Time

In the previous section we derived a selection routine and informally analyzed its correctness: it halts on all inputs, and always computes the correct answer. We'll turn now to its (allegedly linear) run time. The intuitive idea behind the $O(N)$ average time is that typical iterations remove a substantial fraction of the range $L\,..\,U$. If each step were to remove half the elements, then an identity like

$$N + N/2 + N/4 + N/8 + \cdots \le 2N$$

would describe the total run time.

This section supports our intuition with observations of the algorithm at work. In addition to insight about Select, this exercise illustrates general techniques for the empirical analysis of algorithms. (Problem 6 introduces the mathematical analysis of selection algorithms.)

Figure 1 illustrates the algorithm's behavior on an array of 21 elements. That figure is useful as one first studies the algorithm, but it is too detailed to give much insight into the algorithm's performance. Figure 2 therefore presents a similar picture of an array, and a "stick diagram" representation of the same computation. The horizontal lines represent the subrange $L\,..\,U$ at each iteration, the bullets represent the partitioning elements, and the vertical line represents K. The stick diagram contains less information than the array (we don't know the values being permuted), but it shows the key element of performance: the size of the subarrays throughout the computation.

I generated Figure 2 by adding print statements at key positions in a selection routine. The resulting output was processed by a program written in a language for describing graphical displays of data. The array portion of the figure requires the complete information. The stick diagram, on the other hand, can be constructed by this program that stores only the values of L and U, and does away entirely with the array X:

```
L := 1; U := N
while L < U do
    decrement Y
    M := RandInt(L,U)
    draw a line from L,Y to U,Y
    plot a bullet at M,Y
    if K <= M then U := M-1
    if K >= M then L := M+1
```

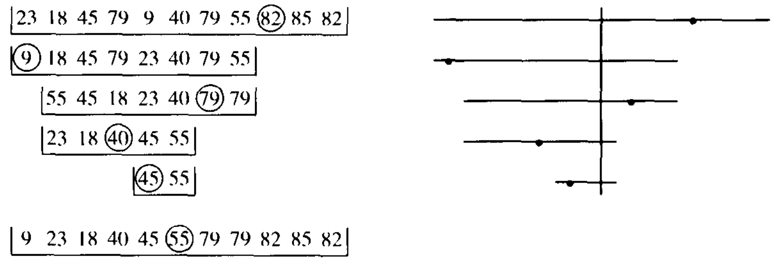


FIGURE 2. Array and Diagram Representations

If the array contains no duplicate elements, then randomly choosing the partition element makes it equally likely to wind up in every position between L and U. For that reason, the above code sets M to a random integer in that range. The statistical nature of the algorithm's performance makes no assumption about the probability distribution of the inputs; the variation is a function of the randomizing *Swap* statement. Figure 3 displays five runs of the program to select the median of 101 elements; the integer plotted at the right of each run is the total number of comparisons used by the process.

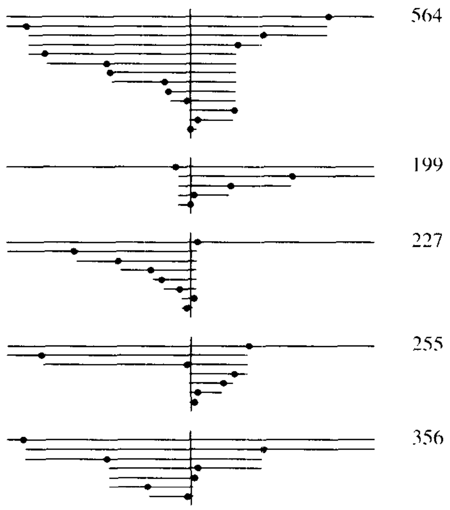


FIGURE 3. Five Runs of Selection

The model of each step halving the range implies that selecting the median of 101 elements requires roughly

$$100 + 50 + 25 + \cdots = 200$$

comparisons. Figure 3 shows that the model is imperfect yet still useful. The second computation was quite close to the model: each guess came close to halving the existing interval. The first computation was particularly unlucky; it chose several partitioning elements near the end of the range. The three other computations fall between those two extremes. The halving model suggests that the algorithm uses $2N$ comparisons; these experiments suggest that the program finds the median in $C_{median} \times N$ comparisons for some value of $C_{median} > 2$.

To estimate the constant C_{median}, we'll gather data on the number of comparisons used by the algorithm. Instead of running the complete algorithm on real data, we'll use this "skeleton" program to count the comparisons the algorithm would use.

```
CCount := 0
L := 1; U := N
while L < U do
    CCount := CCount + U-L
    M := RandInt(L,U)
    if K <= M then U := M-1
    if K >= M then L := M+1
```

Program 1 uses $U - L$ comparisons to partition the $U - L + 1$ elements in the range $L\,..\,U$. The above program can simulate the computation on a set of size $N = 10^6$ in a few dozen steps rather than a few million steps.

Figure 4 plots the results of selecting the median 101 times at five different values of N, ranging from 101 to 1,000,001. The top graph presents the complete data: each mark records the number of comparisons in one experiment divided by N, which estimates the constant C_{median}. It appears that C_{median} is somewhere between 2 and 6, but the sheer bulk of the data obscures the information it contains.

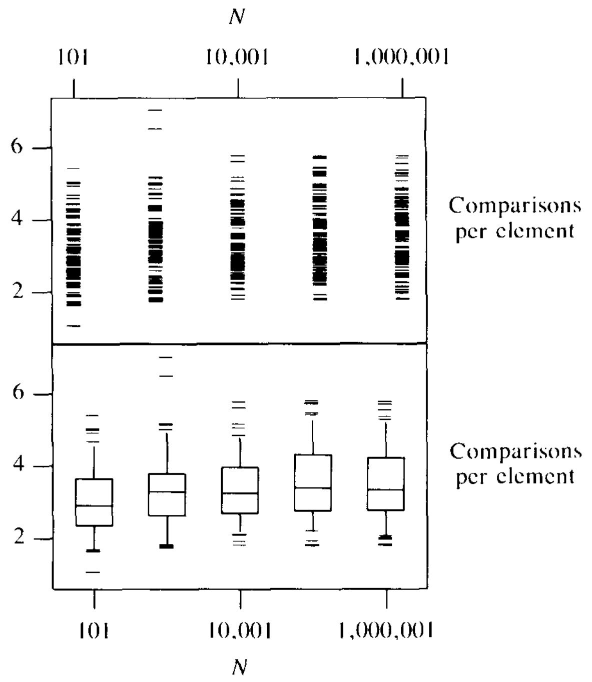


FIGURE 4. Performance of Select in Selecting Medians

The bottom graph in Figure 4 summarizes the top graph using J. W. Tukey's "box plots." The middle horizontal line in the box denotes the median of the samples, and the top and bottom lines denote the upper and lower quartiles (in this case, the 26th- and 76th-smallest elements in the set of 101 real numbers). The lines out of the box show the spread to the 5th and 95th percentiles, and the extreme points beyond those percentiles are plotted explicitly. By highlighting the important quantiles, the box plot shows that C_{median} tends to be between 3 and 4. In 1971, Knuth showed mathematically that its average value tends to 3.39 as N grows large; the five medians in the bottom graph are, in increasing order, 2.90, 3.28, 3.24, 3.37, and 3.32.

So far we have concentrated on computing the median. Figure 5 presents data on selecting the K^{th} value, for $K = 1, 100{,}001, 200{,}001, \ldots, 1{,}000{,}001$; N is fixed at 1,000,001. The top graph indicates that the median is the most expensive to compute, while other values tend to require fewer computations.

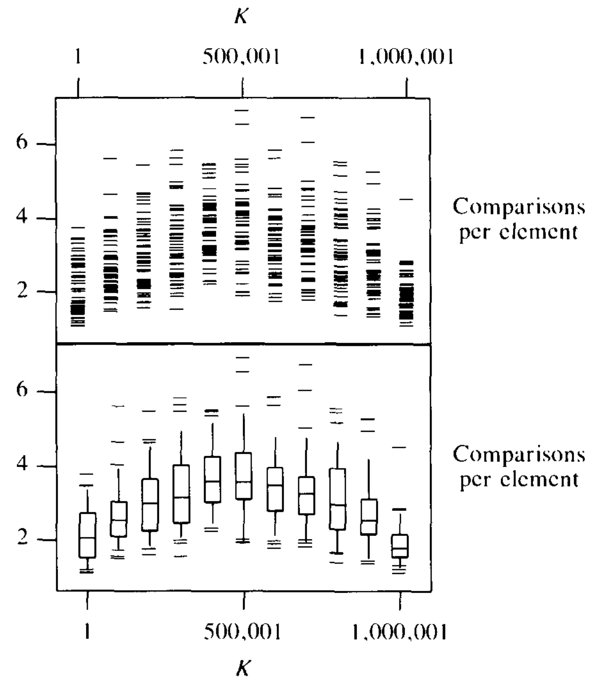


FIGURE 5. Performance of Select When N = 1,000,000

The box plot in the bottom graph in Figure 5 presents the information more clearly. We already knew that the median requires about $3.4N$ comparisons. This graph suggests that the minimum and maximum require about $2N$ comparisons. It also suggests that the cost is symmetric around the median (which makes sense—selecting the K^{th}-smallest is just selecting the $(N - K)^{th}$-largest with the comparisons reversed).

Our analysis of Program 1 concentrated on the fact that it uses $O(N)$ comparisons. Because it does only some constant number of other operations along with each comparison, its total running time is also linear. To gain further insight I implemented Program 1 in C and compared it to the library subroutine `qsort`. The system sort required about $100N \log_2 N$ microseconds to sort an array of N elements, while Program 1 selected the median in about $100N$ microseconds. For N = 100,000, this translates into 10 seconds for Program 1 versus almost three minutes for the sort.

Principles

We have analyzed two aspects of Hoare's selection algorithm: its answer is *correct*, and it computes that answer *efficiently*. This exercise illustrates two important points about the analysis of programs.

A Spectrum of Analyses. There are several reasons why I believe that Program 1 is correct. This column presented both an informal correctness argument and pictures showing the algorithm at work (generated by the program itself). The July 1985 column discussed scaffolding for viewing the program at work and for testing the program. Each of these analyses supports the others: watching the program at work gives insight into its invariant, which in turn is useful for testing.

I am also convinced that Program 1 runs in $O(N)$ time on arrays with few duplicated elements. This column supports that premise with an informal mathematical argument (the "halving model") and a series of experiments observing the program at work. The experiments progressed from detailed pictures of the array to "stick diagrams" illustrating the size of the subrange to graphs counting the number of comparisons. Each experiment in the series described more computations but gave less information about each one. Problem 6 continues this trend, and shows how abstraction of the program can eventually lead to a mathematical analysis.

Skeleton Programs. We saw several programs that provide information about Program 1 without performing all the work of the complete program; Problem 6 describes several additional programs with this flavor. While Program 1 would use several billion steps on a set of size one billion, these programs can gather information on the same computation in just a few dozen operations. These programs are important midpoints on the spectrum of analyses sketched above.

Graphical Methods in Analysis. Graphical output is now available to many programmers; we should use it to understand our programs. All pictures in this column were drawn by simple programs (between 10 and 30 input lines). We understood the correctness of the algorithm with detailed pictures showing the history of the computations and "array boxes" that illustrated the loop invariants. Graphical displays allow us to analyze a large volume of experimental data. The bottom graph in Figure 5, for instance, uses about 150 horizontal and vertical lines to represent 550 computations that together represent over a billion comparisons. Mathematical analysis of most algorithms is downright hard, but simulations and pictures are well within the grasp of most programmers.

Problems

1. Program 1 partitions about a random element in the subrange. Study the effectiveness of using other partitioning elements (such as the median of the first, middle, and last elements in the array or

an appropriate representative of a larger sample).

2. The Select algorithm and its derivatives aren't always the best ways to implement selection. How would you select the second-smallest element in a three-element array? What if $K = 6$ and $N = 11$? What if $K = 1000$ and the $N = 1{,}000{,}000$ input values were stored on a reel of magnetic tape?
3. How would you find the median of one million values stored on magnetic tape if your computer had only about a dozen words of main memory?
4. Although Program 1 runs in $O(N)$ average time, it requires $O(N^2)$ time in the worst case. Describe a selection algorithm with $O(N)$ worst-case time.
5. Perform experiments and display data for the following problems.
   a. The discussion of run time concentrated on the number of comparisons used; that is a good but sometimes imperfect indicator of cost on a real machine. Implement a selection algorithm and measure its run time; any surprises?
   b. Delete the randomizing *Swap* statement from Program 1. How does the average run time change? Describe an input that achieves the worst possible run time.
   c. The analysis in Figure 4 held K fixed at $(N + 1)/2$ and varied N; the analysis in Figure 5 held N fixed at 1,000,001 and varied K. Describe the function of two variables that tells the average number of comparisons needed to find the K^{th}-smallest element in a set of N distinct elements. In particular, what is the shape of the curve induced by varying K when N is fixed? When K is fixed at a constant fraction of N, how does that curve behave?
   d. Our analyses assumed that the input array contained no duplicated elements; how does Program 1 perform if some array elements appear many times? How can that performance be improved?
6. This problem mathematically investigates the performance of Program 1 when it is called with $K = 1$ (that is, when it selects the least element in the array). The skeleton program that counts comparisons (without actually selecting the least element) simplifies to

```
U := N
while U > 1 do
    CCount := CCount + U-1
    U := RandInt(1,U) - 1
```

Show that this recursive program computes the same function

```
function CCount(N)
  if N <= 1 then
      return 0
    else
      return N-1 +
        CCount(RandInt(0,N-1))
```

If C_N denotes the average value of *CCount(N)* after the code is executed, show that it satisfies the recurrence relation

$$C_0 = C_1 = 0$$
$$C_N = N - 1 + 1/N \sum_{0 \le i \le N-1} C_i$$

Write a program that computes $C_0, C_1, \ldots, C_M$. (Hint: first use a table $C[0\,..\,M]$ and $O(M^2)$ time, then make your algorithm run in $O(M)$ time, and finally remove the table.) Use that program to characterize the behavior of C_N; one possible use is to run the program to gather data, while another approach studies its structure to see how to "telescope" the recurrence analytically.

7. [J. M. Chambers] The Select algorithm ensures that $X[1\,..\,K-1] \le X[K] \le X[K+1\,..\,N]$ for a single value of K, while Quicksort establishes that condition for all values of K. The problem of "Partial Sorting" calls for establishing the condition for a set of integers in the range $1\,..\,N$. For instance, in drawing box plots of 101 values we were interested in the set {6, 26, 51, 76, 96}. Show how to modify the Quicksort/Select idea to compute partial orders. Given the input arrays $X[1\,..\,N]$ and $1 \le K[1] \le K[2] \le \cdots \le K[M] \le N$, the program should establish

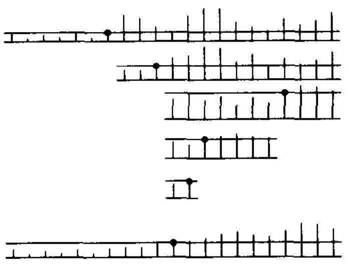
$$X[1..K[1]-1] \le X[K[1]] \le$$
$$X[K[1]+1..K[2]-1] \le X[K[2]] \le$$
$$X[K[2]+1..K[3]-1] \le X[K[3]] \le \cdots$$

8. For this problem, assume that every element of the array X has two fields: $X[I].key$ is the key of the I^{th} element, and $X[I].wt$ is its weight (a positive real number). Let S denote $\sum_{1 \le i \le N} X[i].wt$. The "weighted median" problem calls for computing the integer K and partitioning the array such that these conditions hold:

$$X[1..K-1].key \le X[K].key \le X[K+1..N].key$$
$$\sum_{1 \le i < K} X[i].wt < S/2$$
$$\sum_{K < i \le N} X[i].wt < S/2$$

Modify Program 1 to perform this task in linear expected time. Show how to use a solution to Problem 4 as a subroutine to solve this problem in linear worst-case time. Modify both algorithms to find other "weighted quantiles": given a real $0 < Q < 1$, find a record such that the weights of lesser keys sum to at most QS, while the weights of greater keys sum to at most $(1 - Q)S$.

9. Give algorithms for finding the minimum and maximum elements in a set and for finding the maximum and second-largest elements. Try to use as few comparisons as possible.
10. Experiment with other graphical representations of median computations. This picture, for instance, illustrates the computation depicted in Figure 1; numbers in Figure 1 are represented here as vertical bars.



Try "animating" this or other representations on interactive displays or as simple "movies."

Further Reading

Hoare originally described Quicksort and Find in one page of the July 1961 *Communications.* He illustrated the young field of program verification by arguing the correctness of Find in the January 1971 *Communications.* Knuth analyzed the run time of the algorithm in his "Mathematical Analysis of Algorithms" on pages 19–27 of the proceedings of the 1971 IFIP Congress. In the March 1975 *Communications,* Floyd and Rivest present a selection algorithm that uses just $N + K + o(N)$ comparisons. Their algorithm is close to the theoretical optimum, and runs like the wind when implemented as a program.

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Ave., Murray Hill, NJ 07974.

ACM Forum *(continued from p. 1120)*

are any hardware features that improve programmer productivity—except that they make that software run faster. The object-oriented Smalltalk-80® system is a software system that relies on extravagant hardware to improve the programmer's lot, and it currently needs a Xerox Dorado per programmer, a $120K user-microprogrammable personal minicomputer built from ECL. It is this "cost/performance barrier" that prevents programmers from using Smalltalk-like systems, so this seemed like a good test case for RISCs. Our experiments showed that the simple case was again the frequent case, so Smalltalk On A RISC (SOAR) includes tags to check types in parallel with fast execution and traps to software in the infrequent case of type disagreement [2]. We just received SOAR chips that worked at speed [3], and our simulations show that NMOS VLSI SOAR chip runs Smalltalk-80 as fast as the Dorado [1], thereby demonstrating that RISCs lower the cost/performance barrier for Smalltalk.

Nelson's references suggest that he believes in a future based on high-level, object-oriented computers running Forth and Pascal. I do not. Fortunately such a difference of opinion is not resolved in the pages of a journal. Computer designers and computer users "vote with their feet," so the future will tell whether people build and use non–von Neumann Forth machines, or something considerably RISCier.

Smalltalk-80 is a trademark of the Xerox Corporation.

David A. Patterson
Dept. of Electrical Engineering and Computer Sciences
Computer Science Division
University of California
Berkeley, CA 94720

REFERENCES
1. Pendleton, J.N. A design methodology of VLSI processors. Ph.D. dissertation, Dept. of EECS, University of California, Berkeley, Sept. 1985.
2. Ungar, D., Blau, R., Foley, P., Samples, D., and Patterson, D. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Symposium on Computer Architecture* (Ann Arbor, Mich., June 5–7). ACM, New York, 1984, pp. 188–197.
3. Ungar, D.M. Design and evaluation of a high-performance Smalltalk computer. Computer Science Division, Dept. of EECS, Sept. 1985.