Algorithms

6. Algorithms for Sorting and Searching

R K Shyamasundar

R K Shyamasundar is Professor of Computer Science at TIFR, Mumbai and has done extensive research in various foundation areas of computer science.

Previous articles of this series were:
1. Introduction to algorithms, January 1996.
2. The while-construct, March 1996.
3. Procedures and recursion, June 1996.
4. Turtle graphics, September 1996.
5. Data types and their representation in memory, December 1996.

In this article, we describe structured data types such as one dimensional arrays and illustrate the use of data structures in the development of searching and sorting programs.We illustrate an algorithmic design technique referred to as the *Divide–and–Conquer* method and introduce an important and a prominent data structure called *trees*.

Introduction

In the previous article of this series, we looked at simple data types and their representation in computer memory. The notion of a simple data type can be extended to denote a set of elements corresponding to one data item at a higher level. The process of structuring or grouping of the basic data elements is often referred to as *data structures*. Some important data structures commonly used are arrays (one/multi dimensional),lists, sets, trees etc. In this article, we discuss one-dimensional arrays and illustrate their effectiveness in the construction of programs. We use algorithms for searching and sorting for illustration. We also illustrate one of the well-known algorithm design techniques referred to as Divide–and–Conquer method. Further, we introduce the data structure trees and show how the abstraction aids in designing efficient sorting algorithms.

One Dimensional Arrays

Let us look at the procedure of summing N natural numbers discussed in earlier articles of this series. One can notice in the algorithm that it is implicitly understood that we know how to generate the next natural number from the current natural

number. Instead, if we are asked to sum the salaries of N persons, we need N variables corresponding to the salary of each person. In other words, the same algorithmic step has to be performed on different data of the same type. In such circumstances, it is convenient to organize the data as a *one-dimensional array* without reference to a large number of names. Any individual element can be accessed through an index. That is, any element can be accessed without going through all the other elements. For example, let us say that a series of books with the title *Advances in Computers* can be accessed under the same variable name, AC, with an *index*. Thus, the first volume in the series can be accessed by AC[1] and the ith volume can be accessed by AC[i]. The name AC[i] refers to the ith volume in the sequence AC[1], ..., AC[N]. Thus, AC[i+1] refers to the volume after AC[i].Formal declaration of a single dimensional array takes the form:

The process of structuring or grouping of the basic data elements is often referred to as *data structures*.

AC: array [1..100] of some_type

The above statement can be interpreted as: AC is a single dimensional array having 100 elements which are of type 'some_type'; the first element can be accessed by the index 1 and the last element by the index 100. In other words, the data-structure represents a sequence of elements each of which can be accessed by the designated indices.The use of this data structure is illustrated through the *sorting* and *searching* problems.

Sorting

Consider the problem of arranging a list of N distinct natural numbers in ascending order.

A Naive Approach: Let a list be denoted A[1],..., A[N]. Let us assume that the sequence is not already ordered. Then, one can sort the numbers with the following intuitive idea:
Traverse the list in a sequence one element at a time. Whenever

Array data structure represents a sequence of elements each of which can be accessed by the designated indices.

two adjacent elements are found to be in the wrong order (say ith element is larger than $(i+1)$th element) then they are exchanged (swapped). Repeat the process till no exchanges are possible — at this point, the numbers are in the ascending order of their magnitude!

In the sequel, we use the *for-loop* control construct for ease of reading. The construct

for $i:=$ 1 to N *do* stat-body *endfor*

is interpreted as follows:

In this construct, i is referred to as the loop-index, 'stat-body' is any sequence of statements (it may include another for-loop) and N is a constant. Starting with i equal to one, stat-body is executed N times each time incrementing i by 1. It must be noted that the loop-index i cannot be changed in the stat-body.

It can be easily observed that the above for-loop is equivalent to

$i:=1;$
while $i \leq$ N *do*
    stat–body; $i:= i+1;$
endwhile

The algorithm for sorting the numbers is described in *Table 1* and the algorithmic steps on a list of 4 numbers shown in *Figure 1*. One can convince oneself that the program given in

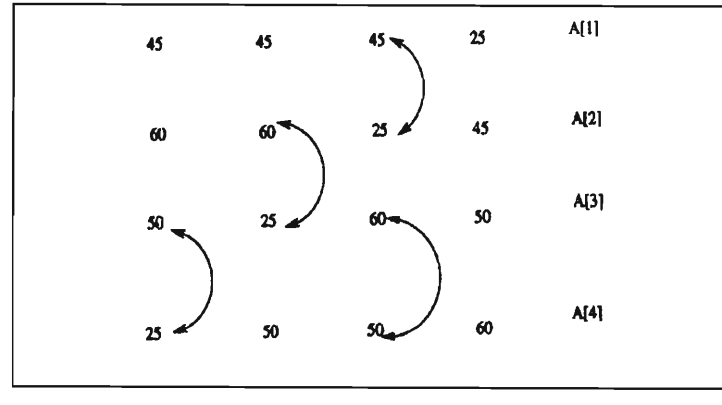

***Figure 1** Algorithmic Steps for the Sorting Process.*

Table 1 Algorithm for sorting numbers.

```
A: array [1..N] of integer
i, j, x: integer (* auxiliary variables of type integer *)
    for i := 1 to N–1 do
        for j := 1 to N–1 do
            if A[j] > A[j+1] then
                (* Beginning of swap action *)
                (* swap elements A[j] and A[j+1] *)
                    x := A[j]; (* x is a temporary variable *)
                    A[j] := A[j+1];
                    A[j+1] := x;
                (* end of swap action*)
            endif
        endfor
    endfor
```

Table 1, sorts the given numbers by the following hint:the maximum number is sinking to the bottom while the smaller number bubbles up. Hence, this algorithm is referred to as *bubble sort*.

Note

- The outermost loop is executed N–1 times; this is sufficient since, at the end of N–1 iterations, N–1 elements would have occupied their final positions.
- In the body of code that swaps (i.e. interchanges) two elements,the order of instructions is important. Further, it can be seen that we can write a separate procedure for swapping two elements of an array as shown in *Table 2*.

```
procedure swap (i, j: integer) (*A is assumed to be global for
simplicity *)
    var x: integer (* local variable *)
    begin
        x := A[ i ]; (* x is a temporary variable *)
        A[ i] := A[ j ];
        A[ j] := x;
    end
```

Table 2 A procedure for swapping.

The maximum number sinks to the bottom while the smaller one bubbles up. Hence the name 'bubble sort.'

The actual code used for swapping the two values A[j] and A[$j+1$] can be replaced by

```
k:= j+1; (* k is a new local variable *)
call swap (j,k)
```

Further, the reader should note that in the replaced code (i) a new local variable k has been introduced, and (ii) indices have been used as formal parameters for the swap procedure rather than the array variables, to avoid the intricacies in understanding the substitution of formal parameters with actual parameters.

Time Complexity

Let us analyze the execution time for the previous program. It can be seen that for each value of i and j, the program either executes a swap procedure and updates the loop-indices or does only the updating of loop-indices $i$and j (in the context of for-loops, these operations are implicit). For the sake of simplicity, let us assume that the time required for either of these operations is bounded by some constant (say *constant*). For each value of i, j takes the values 1 to $N-1$. Thus, execution time taken by the program for each value of i is $(N-1) \times$ *constant*. Since i can take values 1 to $N-1$, the total execution time for the program will be $(N-1) \times (N-1) \times$ *constant*. As $N-1$ is close to N, for large values of N, we can say that the execution time of the program is proportional to N^2.

Search

In this section, we discuss algorithms for sequential and binary searches.

Sequential Search

Consider an array of n distinct numbers stored as a single dimensional array. The problem is to design an algorithm that

will accept as its input the number *key* and will yield index *j* where *key* appears. In other words, the output will be the index of the array if the given element *key* appears in the array and will be 0 otherwise. The simplest algorithm is to compare the elements of the array with the given key starting from the last element till either a match is found or all the elements are exhausted. The reason for starting from the last element lies in the problem description that the output for failure to find the element is 0 (which we obtain naturally if we start from the last element). Using the notations given in the earlier articles,the program is given in *Table 3*. In the program,we have used the boolean variable *continue* for checking the need to do further search as in the given array.

In the program shown, the reader should take care to see that at no point is one checking the contents of the array A whenever the index i is outside the lowest and highest indices of the given array. It may be noted that the same program can in fact be used over an array of strings instead of an array of integers; in that case, the program would correspond to finding an entry in a database or bibliography.

Table 3 Algorithm for sequential search.

```
A: array [1.. n] of integer; (* Given array assumed to be sorted*);
   i, key: integer ; (* local variables of type integer*)
(* key contains the value to be checked*)
i:= n; (* index initialized to last element*)
continue:=true; (* indicates search is not complete*)
while (i ≠ 0) and ( continue:=true) do
                    if A[i] = key then continue:=false
else i := i–1
endif;
endwhile;
print ( i);
```

Sequential search is not an efficient method of search unless the number of entries to be searched are few.

It must be evident to the reader that the algorithm takes at the worst n steps and this situation occurs when the search fails (i.e., there is no entry corresponding to the given key). The sequential search is not an efficient method of search unless the number of entries to be searched are few. If we assume that the given array is already sorted in ascending order, then one can arrive at an efficient method of search referred to as the binary search which is discussed next.

Binary Search

The idea underlying this scheme is quite simple.The method starts by looking at the first entry in the middle of the array. If this entry matches with the *key*, then the search is over. If the *key* is numerically or lexicographically (or alphabetically) less than this entry, then the second half of the array can be omitted completely; if on the other hand, *key* is numerically or alphabetically larger than the middle entry, then the first half of the array can be omitted. The same procedure can be repeatedly applied and each time either the search is complete or the search space is reduced by half. We use the following notation in the binary search program shown in *Table 4*.

Notation: *int_near* $(l + m)/2$ denotes the value of $(l + m)/2$ rounded down to the nearest integer. That is, if *int_near* $(l + m)/2$ is even then the expression denotes $(l + m)/2$ and denotes *int_near* $(l + m - 1)/2$ otherwise.

Let us assume that the array is not vacuous, i.e.,there is at least one element. Initially, l and m have been set to 0 and $n+1$ which are outside the range of array indices of the given array. Thus, for a match the index should be greater than l and less than m. From the program text one can see that the while-loop exits when $j = l$ or when a match has been found (i.e.,*continue* is false).In case $j = l$, it follows that $m = l + 1$ and hence, match could not have been found and hence, a zero is printed. However, if the loop exits because of the match,then the index j

Table 4 Algorithm for binary search.

```
A: array [1.. n] of integer; (* array in sorted order *)
(* A[1] < A[2] < ... A[n] — given *)
(* key is the given number *)
j, l, m, key : integer (* local variables of type integer *)
continue:= true;
l := 0; (* lower index initialized to zero — one less than the given lowest*)
m := n + 1; (* highest index — one higher than the highest*)
j:= int_near(l + m)/2 ;
while (j > l) and (continue = true) do
        if  key = A[ j ] then  continue:=false
        else
                begin
                        if  key  < A[j]  then  m:=j
                        else l:=j
                        endif;
                        j:= int_near(l + m)/2
                end;
        endif;
endwhile;
if  j = l   then print (0)   (* key not found *)
 else print ( j) (* element in jth position has matched *)
endif
```

indicates the position where a match has been found.

Time Complexity

In the linear search case we found that in the worst case, n comparisons are required. Now let us analyze the worst case time complexity (which is proportional to the number of steps executed for the binary search). In the while-loop, the search space is halved for every execution of the body of the loop. Thus, we can say that the number of times the loop can execute is bounded by r where r is the smallest integer such that 2^r is

Divide–and–conquer often dramatically enhances the performance of algorithms.

greater than n; that is, worst time complexity is of the order $log(n)$ (because $log_2(n) < r$). For instance, if we search an array (or a table) having 2^{10} entries, the comparison is done at most 10 times; this is indeed remarkable as compared to the number of comparisons for the previous algorithm. Of course, we should remember that the array is ordered (which is done only once for that matter).

Natural questions that arise in the development of algorithms are:

- Can we arrive at a better (more *efficient* in some sense) program for the sorting and searching problems?
- Are there general algorithmic techniques that lead to the development of *efficient* algorithms?

In the following, we first discuss a widely used general algorithm design technique called *Divide–and–Conquer* that often dramatically enhances the performance of the algorithm and then discuss efficient sorting algorithms.

Divide–and–Conquer

Quite often a problem can be solved by

- reducing it to smaller problems of the same kind,
- solving the sub-problems, and
- integrating the partial solutions to obtain the solution to the original problem.

If the sub–problems are precisely the same problem on hand applied to 'smaller' inputs, then recursion can be effectively used in the algorithmic description. Such a method is referred to as the *Divide–and–Conquer* (in fact, the attentive reader may recognize such a strategy in the solution of the towers-of-Hanoi problem discussed earlier in this series of articles). We shall illustrate the method through the following example.

Example: Finding Maximum and Minimum. Let us consider

an array of N elements. Let N be a power of 2 (i.e., $N = 2^m$ for some m). Our task is to find the maximum and minimum value in the given array.

The complexity of the search algorithm is proportional to the number of comparisons.

Solution 1: A naive way is to find the maximum and minimum separately. The algorithm underlying this idea is described in *Table 5*.

Let us analyze the number of comparisons made for finding the maximum. It can be seen that the complexity will be proportional to the number of comparisons. To find the maximum, we see from the for-loop that N–1 comparisons will be made.The program for finding the minimum follows in a similar way. If we find the minimum after finding the maximum, it is enough to find the minimum among the remaining N–1 numbers (assuming there are at least 2 numbers; otherwise, maximum and minimum will be the same). So to find the maximum and minimum of N numbers, we need $(N–1) + (N–2) = 2N - 3$ comparisons. Now, let us see whether we can develop an algorithm whose complexity is better than that of solution 1 using the divide–and–conquer method.

Solution 2: *Basic Idea*

We see that the problem of finding the maximum or minimum is decomposable in the following sense:

Table 5 Naive way to find maximum.

```
A: Array [1.. N] of integer;
i, mx, mn: integer (* local variables *);
      mx:= A[1]; (* mx denotes maximum and initially
                    any element can be chosen; here, the
                    first element has been chosen *)
      for i:= 2 to N do
            if A[ i ] > mx then mx:= A[ i ] endif
      endfor
```

Maximum of a set of numbers say S, is the same as the maximum of the maximum of two subsets S_1 and S_2 such that $S_1 \cup S_2 = S$. That is,

$$maximum\ (S) = maximum\ (maximum\ (S_1), maximum\ (S_2))$$

This will be the basic principle of the algorithm to find the maximum and minimum of an array of numbers. The main idea of using the divide–and–conquer strategy is described below:

- Divide array A into two sub-parts A_1 and A_2 such that A_1 and A_2 together form A.
- Find the maximum and minimum of each by recursive application of the algorithm.
- The maximum and minimum of A can be computed from the maximum and minimum of A_1 and A_2 by making two comparisons.

The complete algorithm abstracted as procedure MINMAX is described in *Table 6*.

Note: For the sake of simplicity we have not given the actual assignments for the indices in the step of splitting the array. Since we have assumed the number of elements in the array to be a power of two, we have $U - L + 1 = 2^m$ for some m. The values of the indices of the two arrays to which A is split are

$l_1 := L;\ U_1 = 2^{m-1}$
$l_2 := 2^{m-1} + 1;\ U_2 = 2^m$

Let us analyze the number of comparisons made by the above algorithm. Explicit comparisons are made in line (1) where maximum and minimum is computed between two array elements, and in line (5) where maximum and minimum of two elements are computed. The recursive algorithms can be effectively analyzed through recurrence equations (see

Table 6 Procedure for finding maximum and minimum.

```
procedure MINMAX (A, L, U, mx, mn);
        (* A is an array ranging between L and U; mx and mn will have the maximum
        and minimum value respectively when the procedure terminates *)
    var l1, l2, U1, U2: integer ; (* Local Variables *)
        mx1, mx2, mn1, mn2: integer ; (* Local Variables *)
    if (U - L + 1) = 2 then
1.      if A[L] > A[U] then
            mx := A[L]; (* maximum *)
            mn := A[U]; (* minimum *)
        else
            mx := A[U]; (* maximum *)
            mn := A[L]; (* minimum *)
        endif
    else
      Split A into two halves A1 and A2 with lower and upper indices to be
      l1, U1 and l2, U2 respectively;
2.    call MINMAX (A, l1, U1, mx1, mn1 );
3.    call MINMAX (A, l2, U2, mx2, mn2);
4.    mx := maximum (mx1, mx2 ) ; (* maximum returns the maximum of the two values *)
5.    mn := minimum (mn1, mn2); (* minimum returns the minimum of the two values *)
    endprocedure{MINMAX}
```

D E Knuth in Suggested Reading). Let $T(n)$ be the total number of comparisons made by the procedure MINMAX when $U-L+1 = n$. Obviously, $T(2) = 1$. One comparison each is needed for the operations of line 4 and line 5. If $n > 2$, the size of the arrays in lines 2 and 3 will be $n/2$ each. Thus,

$$T(n) = \begin{cases} 1, & \text{if } n = 2, \\ 2 \times T(n/2) + 2, & \text{if } n > 2. \end{cases}$$

It can be shown that the function $T(n) = 3/2n - 2$ is the solution to the above recurrence equation. Further, we can show that $3/2n - 2$ comparisons are necessary for finding the maximum and minimum from n elements. In a sense, the above algorithm is optimal when n is a power of 2.

A *tree* is a hierarchical arrangement of data. If the edges are directed then the tree is referred to as a *directed tree*; otherwise it is referred to as a *tree* or *undirected tree*.

In the previous sections, we have described a sorting algorithm whose worst case time complexity is proportional to n^2. As this problem is encountered in practice widely, one tries to search for better algorithms. Instead of just going for yet another efficient algorithm, let us see whether we can arrive at a lower bound on the time complexity of sorting algorithms in general. Before going further, we introduce another widely used data–structure called *trees*.

Trees

Trees are one of the most important data structures in use. In fact, one uses such an abstraction to describe the family tree, the ancestors, descendents etc. A *tree* is a hierarchical arrangement of data. A typical tree is shown in *Figure 2*. In the figure shown the edges are directed. However, it is not necessary that edges should be directed. If the edges are directed then the tree is referred to as a *directed tree*; otherwise it is referred to as a *tree* or *undirected tree*. One item of the tree is treated in a special way and is referred to as the *root* and the others are organized as its descendants. The descendants are often referred to as *nodes* (note that root is also a node). For instance, node with the label '1' is the root; in computer science, we usually visualize trees with roots at the top. In *Figure 2*, we have shown nodes either

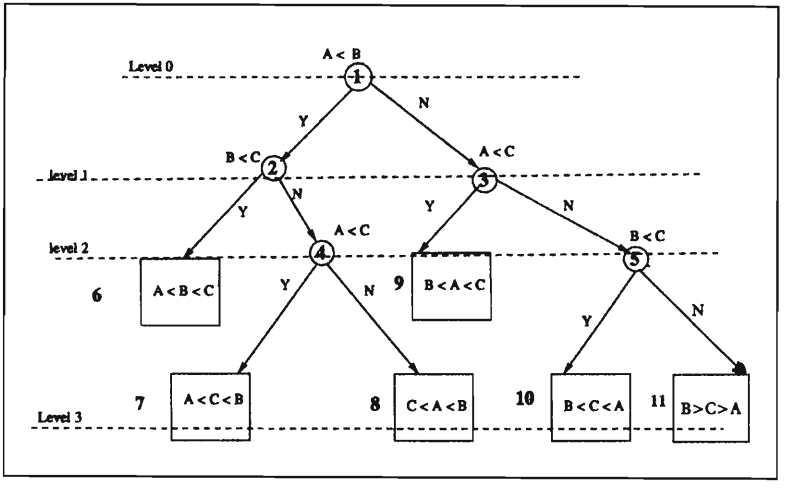

***Figure 2** Algorithmic steps for the sorting process.*

circumscribed by circles or rectangles. Nodes circumscribed by circles have descendants (nodes,1,2,3,4,5) and the nodes circumscribed by rectangles have no descendants (nodes 6,7,8,9,10,11); the latter nodes are referred to as the *leaves*. Edges connecting one node to another are called *branches*. The leaves in the order of their left-to-right occurrence (6, 7, 8, 9, 10, 11) are referred to as the *frontier*. For node 1 (i.e., the root), nodes 2 and 3 are referred to as its *descendants* or *children*. Node 1 is the ancestor of node 2 and the direct ancestors are referred to as *parents*. The descendants of the same node are referred to as *siblings*. For instance, nodes 2 and 3 are siblings of node 1. In the tree shown, nodes which are not leaves have two descendants. Such trees where the descendants are at most two are referred to as *binary trees*. If all nodes in a binary tree except the leaves have at most two descendants, it is referred to as a *complete binary tree*. We say that there is a *path* from node i to node j, if there is a sequence of branches from node i to node j. For example, node 2 is connected to node 8 through 2 branches. The number of branches in the path is called the length of the path. The *depth* of a node, i, in a tree is the length of the path from the root to node i. The *height of a vertex* i in a tree is the length of the longest path from node i to a leaf. For example, node 2 has depth 1 and height 2.

Trees where the descendants are at most two are referred to as *binary trees*. If all nodes in a binary tree except the leaves have at most two descendants, it is referred to as a *complete binary tree*.

We illustrate how the tree abstraction aids in the development of a lower bound on sorting algorithms before illustrating its use in the development of programs directly.

Efficient Sorting Algorithms

Consider the problem of sorting a sequence of n elements drawn from a linearly ordered set S having no known structure. Let $a_1, \ldots, a_n$ be the set of elements to be sorted. Let us assume that the only operation that is available to us is the comparison of elements. For instance, let us consider the process of sorting three elements A, B and C. The possible linear orderings of these elements are depicted in *Figure 2*. The tree shown is

A binary tree of height h has at most 2^h leaves.

essentially a decision tree where comparisons are done at each node. If the answer to the question asked at the node is 'yes' then it branches to the left; otherwise, it branches to the right. The branching terminates if there are no more questions to ask; in this case, termination amounts to arriving at the order among the elements. It can be seen that possible orderings among three distinct elements are 6 (which is nothing but 3!). Assuming the test at each node is binary ('yes' or 'no' answers), we can derive the following results about such decision trees.

Lemma 1: A binary tree of height h has at most 2^h leaves.

Proof: It can be observed that a binary tree is composed of a root and at most two subtrees the height of each subtree can be at most $h-1$. Now using simple induction, it follows that it has at most 2^h leaves.

Theorem 1: Any decision tree that sorts n distinct elements has height at least $\log\, n\,!$.

Proof: As the elements are distinct there are $n!$ distinct permutations. Sorting n elements can map to any one of these permutations, it follows from Lemma 1, the height must be at least $\log n\,!$.

Using simple algebraic manipulations we can reduce the above expression as follows:

$$n\,! \geq n\,(n-1)\ldots\,(\lceil n/2 \rceil) \geq (n/2)^n/2.$$

Taking logarithms, we get $\log\, n\,! \geq (n/2)\log\,(n/2) \geq n/4\log\, n$ for $n \geq 4$. A better approximation can be obtained using Stirling's approximation when n is large.

Thus, we can see that sorting requires comparisons of the order of $n\ \log\ n$.

Table 7 Common Sorting Methods

Method	Basic Idea	Complexity
Bubblesort	Bubbling up the smaller number	n^2
Heapsort	Based on rearranging elements in the tree until the element at each node is ≥ its children	$n \log n$
Merge sort	Split into sequences, sort each and merge the two sequences	$n \log n$
Quicksort	Explained already	n^2

The above bound fixes the lower bound. Now, if can obtain an algorithm with the lower bound as its complexity, then it must be clear that the algorithm will be optimal within some constant of proportionality.

There have been a plethora of sorting algorithms that have been designed. Some of the prominent ones are given in *Table 7*. Another very interesting sorting algorithm is the Quick sort algorithm designed by C A R Hoare. We briefly outline the algorithm which is again based on the divide–and–conquer strategy discussed previously. The basic idea in this method lies

Code Tuning

Some programmers worry too much about efficiency and try to optimize even little things and thus create a clever program that is hard to understand and maintain. On the other hand, some programmers pay too little attention to efficiency and performance and create a beautiful structured program that is too inefficient and hence useless. Good programmers keep a good perspective on efficiency; it is just one of the many important problems the software design encounters. *Code Tuning* locates the expensive parts of an existing program and makes little changes to improve its performance. Even though the approach is not glamorous and is not always the right approach, quite often it makes a big difference in program performance. More about these aspects can be found in the book of Jon Bentley.

Premature optimization is the root of all evil

Don Knuth

Suggested Reading

- D E Knuth. *Fundamental Algorithms*. Addison-Wesley. Reading. Mass., 1968.
- N Wirth. *Systematic Programming: An Introduction*. Prentice-Hall. Englewood Cliffs. New Jersey, 1972.
- D E Knuth. *Sorting and Searching*. Addison-Wesley. Reading. Mass., 1973.
- N Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall. Englewood Cliffs. New Jersey, 1976.
- R G Dromey. *How to Solve it by Computer*, Prentice Hall International, 1982.
- J Bentley. *Programming Pearls*. Addison-Wesley. Reading. Mass., 1986.

Table 8 Outline for quicksort.

```
procedure QUICKSORT(A, L, U); (* A is the array to be sorted
                                 L, U are lower and highest indices *)
    if U=L then A is already sorted (* A has only one element *)
    else
        Select an item k in the array and let p be its final position;
        Let A1 and A2 be the subarrays got from the elements of
        A that are less than and greater than k respectively;
        QUICKSORT(A, 1, p-1); (* Defines A1 *)
        QUICKSORT(A, p+1, U); (* Defines A2 *)
    endif
endprocedure
```

in partitioning the array, A, into two parts, say A_1, A_2 with respect to an element of the array, say k, such that the elements in A_1 are less than k and the elements in A_2 are larger than k. Now, the sorting of the original array is obtained by concatenating the sorted A_1 to the left of element k and the sorted A_2 to the right of element k. The procedure is applied recursively. The outline of the algorithm for a given array of distinct elements is given in *Table 8*.

One of the most interesting aspects of this algorithm is that even though its worst case complexity is quadratic (the curious reader can try to sort a list through the procedure), its average case complexity is proportional to $n \log n$ and the constant of proportionality is quite small.

Exercise: Complete the above outline into a complete program. In the next article, we will discuss further common data structures, other algorithm design techniques and the merits of recursion vs iteration.

Address for Correspondence
R K Shyamasundar
Computer Science Group
Tata Institute of Fundamental Research
Homi Bhabha Road
Mumbai 400 005, India
email:shyam@tcs.tifr.res.in
Fax:022-215 2110