

QUICKSORT: A TUTORIAL

by
Steve Legenhausen

Technical Report 73006

July, 1973

*MKC
C13924
a*

University Computer Center
University of Minnesota
227 Experimental Engineering
Minneapolis, Minnesota 55455

INTRODUCTION

Several recent studies of computer sorting have acknowledged the superiority of the Quicksort method [7,8,9]. Quicksort was discovered about a dozen years ago by C.A.R. Hoare [4,5]. Although a great deal has been written about Quicksort since that time, very little new information has been added to Hoare's elegant description [5], which the interested reader is urged to examine for himself.

Quicksort has usually been presented as a long, formidable looking program, and for this reason it has acquired a reputation of being a "difficult" sorting method. As a consequence many general purpose internal sorting routines are developed on the basis of a less efficient method.

The purpose of this paper is to demonstrate that Quicksort is in fact an extremely simply sorting method, easy to learn and easy to program. Some relatively minor improvements have tended to add considerable length to the program without changing the basic idea.

A serious attempt was made to develop the programs appearing in this paper according to the principles of structured programming, as treated by Dijkstra [1,2] and Wirth [12,13]. The result is an orderly sequence of programs, hopefully above average in readability, whose correctness can be formally proved [3].

All programs are written in the language Pascal developed by Wirth [11]. The semantics of Pascal are defined in a recent report by Hoare and Wirth [6]. Also a textbook for an introductory course in programming, based on the Pascal notation, has recently appeared [13].

Pascal follows very closely the notation of Algol 60; therefore, the programs in this paper should be understandable to one familiar with Algol 60. Of course the references of the preceding paragraph should be consulted for a more complete understanding.

BASIC PRINCIPLE

The program is to rearrange the values of the elements of a given array segment $A[m..n]$ into ascending order, so that

$$A[m] \leq A[m+1] \leq \dots \leq A[n].$$

There is no better description of the basic principle used in Quicksort than that given by Hoare in his original description [5]:

The Quicksort method is based on the principle of resolving a problem into two simpler subproblems. Each of these subproblems may be resolved to produce yet simpler problems. The process is repeated until all the resulting problems are found to be trivial. These trivial problems may then be solved by known methods, thus obtaining a solution of the original more complex problem.

More specifically, the problem of sorting an array may be reduced to that of sorting two smaller segments of the array, provided that all the elements below a certain dividing line are less than all the elements above this dividing line. In this case the two segments may be sorted separately, and as a result the whole array will be sorted.

Letting d stand for the position of the dividing line, we can write down a program schema embodying the above principle.

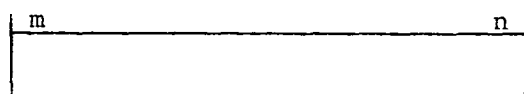
Sort - 1

```
partition(A,m,n,d);
sort(A,m,d);
sort(A,d+1,n);
```

Note: We can apply our program recursively to sort each of the smaller segments remaining after partition, and this is repeated until only segments of length 0 or 1 remain.

PARTITION

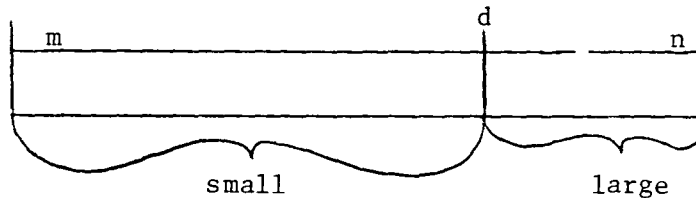
To implement the above schema, it is necessary to have an efficient partitioning procedure. Such a procedure can be developed along the following lines. The method used is based on the principle that the desired effect of partition is to move lower valued elements of the array to one end - the "left-hand" end - and higher valued elements of the array to the other end - the "right-hand" end.



\leftarrow move small values left
large values move right \rightarrow

This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered which is small will remain where it is, but any element which is large stops the scan. A separate scan is

made, starting at the right hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered stops this scan. The two elements pointed at are exchanged. Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be large.



This idea is represented in schema form below:

Partition - 1

- (1) Choose an arbitrary element from the array segment (call it r) to establish our "small-large" criterion. {Our aim is to impose a partition so that a dividing line d is created, where $A[m], a[m+1], \dots, A[d] \leq r \leq A[d+1], \dots, A[n].$ }
- (2) Introduce pointers (i and j) whose initial values are m and n respectively.
- (3) The lower (leftmost) pointer starts first, passing over all elements $\leq r$, and stopping only when it comes to an element $> r$.
- (4) The upper (rightmost) pointer starts its scan, passing over elements $\geq r$, stopping when it discovers an element $< r$.
- (5) $A[i]$ and $A[j]$ are exchanged, i is increased, j is decreased.
- (6) The previous three steps are repeated until the pointers cross (i.e. until $i > j$). When this happens, suppress the exchange, and the partition process has come to an end. $\{A[m..j]$ and $A[i..n]$ are the two segments.}
- (7) A problem arises if either
 - $r = \max(A[m..n]),$ or
 - $r = \min(A[m..n]).$

The schema translated into Pascal:

Partition - 2

{Partition $A[m..n]$ into $A[m..j]$ and $A[i..n]$ so that

$A[k] \leq r$ for $m \leq k \leq j$,

$A[k] \geq r$ for $i \leq k \leq n$,

$A[k] = r$ for $j < k < i$.

Assume $A[m-1] = -\infty$ and $A[n+1] = +\infty$ so that

$A[m-1] < A[k] < A[n+1]$ for $m \leq k \leq n$.)

procedure partition(var A; vector; m,n: integer;

var i,j: integer);

var r: real;

f: integer;

begin

f := (m + n) div 2;

r := A[f]; i := m; j := n;

while i \leq j do

begin

while $A[i] \leq r$ do i := i + 1;

while $A[j] \geq r$ do j := j - 1;

if i \leq j then

begin

exchange A[i],A[j]);

i := i + 1; j := j - 1

end

end {increase i and decrease j};

if i > n then begin exchange(A[f],A[n]); j := n - 1 end

else

if j < m then begin exchange(A[m],A[f]); i := m + 1 end

end {partition};

Note: partition - 2 is a very messy program in several respects, namely

- (1) the last 3 lines are very ugly and cater only to rare cases;
- (2) the assumption that $A[m-1] = -\infty$ and $A[n+1] = +\infty$ is restrictive (this can be eliminated by always checking $(i \leq n)$ and $(j \geq m)$ in the innermost while loops, at a serious loss of efficiency);
- (3) the exchange procedure is used in several places.

The way out of this dilemma is so obvious that it was overlooked for several years. In the innermost while loops, we can stop when $A[i] = r$ or $A[j] = r$ instead of continuing, i.e. we can exchange elements equal to r. This solves all three objections noted above.

The credit for this ingenious modification goes to Singleton [10]. As a matter of historical interest, this is the only improvement to Quicksort discovered to date which was not foreseen by Hoare in his original description of Quicksort [5].

The modified procedure is

Partition - 3

{Partition A[m..n] into A[m..j] and A[i..n] so that

$A[k] \leq r$ for $m \leq k \leq j$,
 $A[k] \geq r$ for $i \leq k \leq n$,
 $A[k] = r$ for $j < k < i$.}

procedure partition (var A: vector; m,n: integer;

var i,j: integer);

var r,w: real;

f: integer;

begin

f := (m + n) div 2;

r := A[f]; i := m; j := n;

while i < j do

begin

while A[i] < r do i := i + 1;

while A[j] > r do j := j - 1;

if i < j then

begin

w := A[i]; A[i] := A[j], A[j] := w;

i := i + 1; j := j - 1

end

end {increase i and decrease j}

end {partition};

We can now write a complete Quicksort procedure using partition:

Sort - 2

{Sort A[m..n], $m \leq n$.}

procedure sort(var A: vector; m,n: integer);

var i,j: integer;

```

begin
  partition(A,m,n,i,j);
  if m < j then sort(A,m,j);
  if i < n then sort(A,i,n)
end {sort};

```

Note: Conditions for termination of recursive calls:

- (1) A partition containing less than 2 elements must be recognized as already sorted.
- (2) No partition must ever be as large as the original segment to be sorted.

IMPROVEMENTS

Can we improve on the above procedure? A good way to look for an answer to this question is to examine the worst case. Consider the case that $r = \max(A[m..n])$ continually. This happens if, for example, we have

2	4	6	8	10	12	14	16	1	9	5	11	3	13	7	15
---	---	---	---	----	----	----	----	---	---	---	----	---	----	---	----

After the first partition we have the two segments $A[m..n-1]$ and $A[n..n]$. Then working on the first of these we get after the next partition $A[m..n-2]$ and $A[n-1..n-1]$. This sorting of one element at a time continues until the bitter end.

Let $N = n - m + 1 =$ length of initial segment. An easy analysis verifies that:

- (1) the auxiliary memory required is proportional to N , since recursive calls continue to a dynamic depth of $N - 1$,
- (2) the running time is proportional to N^2 .

We can guarantee that sort will not generate a dynamic depth exceeding $\log_2 N$, if following a partition, it will only call on itself for sorting the shorter of the two remaining segments. Applying sort recursively to the shorter segment will leave the other segment unsorted, but this can be remedied by repeatedly applying this half-effective technique to the still unsorted segment.

The procedure incorporating this suggestion follows:

Sort - 3

```

{Sort A[m..n].}
procedure sort(var A: vector; m,n: integer);
    var i,j,l,u: integer;
begin
    l := m; u := n;
    while l < u do
        begin
            partition(A,l,u,i,j);
            if j - l < u - i then
                begin if l < j then sort(A,l,j); l := i end
            else
                begin if i < u then sort(A,i,u); u := j end
            end
        end {sort};

```

The preceding program answers one of the two objections raised, namely that now the amount of auxiliary memory space is never more than $\log_2 N$. But the running time for the worst case is still proportional to N^2 . Can this be corrected also?

Suggestions:

- (1) choose r from $A[m..n]$ at random, or
- (2) choose r as the median of a small sample from $A[m..n]$.

The worst case is unchanged, but now it will be very unlikely to occur in practice. Other suggested improvements are:

- (3) define an internal procedure to be called recursively, cutting down on the number of internal variables which have to be manipulated.
- (4) segments of fewer than M elements can be sorted by another method more suited to the simpler task.

A program incorporating the third suggestion appears below.

Sort - 4

```

{Sort A[mm.nn].}
procedure sort(var A: vector; mm,nn: integer);
    var m,n,i,j: integer;
    procedure qsort;
        {sort A[m..n]}
        var l,u: integer;

```



```

begin
  l := m; u := n;
  while l < u do
    begin
      partition(A,l,u,i,j);
      if j - l < u - i then
        begin m := l; n := j; l := i end
      else
        begin m := i; n := u; u := j end;
      if m < n then qsort
    end
  end {qsort};
begin
  m := mm; n := nn;
  if m < n then qsort
end {sort};

```

A Quicksort algorithm incorporating suggestions (2) and (4) above, as well as the other improvements described in this paper, has been published by Singleton [10]. Singleton's algorithm is given both in Algol and in Fortran. For comparison a translation into Pascal appears on the next page.

In this program the explicit recursion has been removed by using the well-known technique of maintaining local variables on a push down list. It should be understood however that the algorithm is still recursive. It is simply a matter of efficiency whether the recursion is explicit and implemented in the language, or whether the recursion is implicit and implemented by means of a push down list.

Preliminary experiments with the CDC 6600 implementation of Pascal show that it makes little difference, either in the running time or the amount of memory space used, if the explicit recursion is left in, thus contradicting the common misconception about the "high-overhead" of recursive programs. Experiments also show that Pascal compares favorably with Fortran from the point of view of efficiency on the CDC 6600.

```

PROCEDURE QSORT (VAR A: VECTOR; II, JJ: INTEGER);
  {QUICKSORT OF A[II..JJ]}
  {SINGLETON, RICHARD C. ALGORITHM 347. COMM. ACM 12, 3
  [MARCH 1969], 185 - 187.}

```

9

```

VAR T, TT: REAL;
    I, J, IJ, K, L, M: INTEGER;
    IL, IU: ARRAY [0..15] OF INTEGER;

```

```

PROCEDURE SPLIT;
  {REQUIRES I < J; SPLIT A INTO TWO SEGMENTS A[I..L] ^ A[K..J] SUCH THAT
  ALL VALUES IN THE FIRST SEGMENT ARE LESS THAN OR EQUAL TO ALL VALUES
  IN THE SECOND SEGMENT}

```

```

BEGIN

```

```

  IJ := (I + J) DIV 2; T := A[IJ]; K := I; L := J;

```

```

  IF A[I] > T THEN

```

```

    BEGIN A[IJ] := A[I]; A[I] := T; T := A[IJ] END;

```

```

  IF A[J] < T THEN

```

```

    BEGIN

```

```

      A[IJ] := A[J]; A[J] := T; T := A[IJ];

```

```

      IF A[I] > T THEN

```

```

        BEGIN A[IJ] := A[I]; A[I] := T; T := A[IJ] END

```

```

    END;

```

```

  REPEAT

```

```

    REPEAT L := L - 1

```

```

      UNTIL A[L] < T;

```

```

    REPEAT K := K + 1

```

```

      UNTIL A[K] > T;

```

```

    IF K <= L THEN

```

```

      BEGIN TT := A[L]; A[L] := A[K]; A[K] := TT END

```

```

    UNTIL K > L

```

```

  END {SPLIT} ;

```

```

PROCEDURE ISORT;

```

```

  {REQUIRES A[I-1] LESS THAN OR EQUAL TO ALL VALUES IN A[I..J];

```

```

  STRAIGHT INSERTION SORT OF A[I..J]}

```

```

BEGIN

```

```

  I := I + 1; WHILE I <= J DO

```

```

    BEGIN

```

```

      T := A[I] K := I - 1;

```

```

      WHILE A[K] > T DO

```

```

        BEGIN A[K+1] := A[K]; K := K - 1 END;

```

```

        A[K+1] := T; I := I + 1

```

```

    END

```

```

  END {ISORT} ;

```

```

BEGIN

```

```

  M := 0; I := II; J := JJ;

```

```

  IF I < J THEN

```

```

    BEGIN

```

```

1:   SPLIT;

```

```

      IF L - I > J - K THEN

```

```

        BEGIN IL[M] := I; IU[M] := L; I := K END

```

```

      ELSE

```

```

        BEGIN IL[M] := K; IU[M] := J; J := L END;

```

```

      M := M + 1;

```

```

2:   IF J - I > 10 THEN GOTO 1;

```

```

      IF I = II THEN

```

```

        BEGIN IF I < J THEN GOTO 1 END;

```

```

        ISORT;

```

```

        M := M - 1; IF M >= 0 THEN

```

```

          BEGIN I := IL[M]; J := IU[M]; GOTO 2 END

```

```

    END

```

```

  END {QSORT} ;

```

CONCLUSION

Singleton's algorithm is believed to be the most efficient sorting algorithm presently known. On the surface it appears to be a very complex program. The intent of this paper is to present Singleton's algorithm as the natural evolution of a simple idea. The degree to which this effort is successful can only be judged by the reader.

REFERENCES

1. Dijkstra, Edsger W. A Short Introduction to the Art of Programming. EWD 316, Technische Hogeschool Eindhoven, Eindhoven, The Netherlands (August 1971), 98 pp.
2. Dijkstra, Edsger W. Notes on Structured Programming. Structured Programming. Academic Press, London, 1972, 1-82.
3. Foley, M. and Hoare, C. A. R. Proof of a recursive program: Quicksort. Computer J.14, 4 (Nov. 1971), 391-395.
4. Hoare, C. A. R. Algorithms 63, 64, 65. Comm. ACM 4, 7 (July 1961), 321-322.
5. Hoare, C. A. R. Quicksort. Computer J.5, 1 (April 1962), 10-15.
6. Hoare, C. A. R. and Wirth, N. An Axiomatic Definition of the Programming Language Pascal. Berichte der Fachgruppe Computer-Wissenschaften, ETH, No. 6 (December 1972), 29 pp. [To be published.]
7. Knuth, Donald E. The Art of Computer Programming, Volume 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973, 114-123.
8. Martin, William A. Sorting. Computing Surveys 3, 4 (December 1971), 147-174.
9. Rich, Robert P. Internal Sorting Methods Illustrated with PL/1 Programs. Prentice-Hall, Englewood Cliffs, N. J., 1972.
10. Singleton, Richard C. Algorithm 347. Comm. ACM 12, 3 (March 1969), 185-187.
11. Wirth, Niklaus. The programming language Pascal. Acta Informatica 1 (1971), 35-63. [Revised Report, Berichte der Fachgruppe Computer-Wissenschaften, ETH, No. 5, (November 1972), 49 pp.]
12. Wirth, Niklaus. Program development by stepwise refinement. Comm. ACM 14, 4 (April 1971), 221-227.
13. Wirth, Niklaus. Systematic Programming: An Introduction. Prentice-Hall, Englewood Cliffs, N. J., 1973.

Errata

p. 5 line 20: $A[i] > r$ should be $A[i] < r$

p. 7 line 10: $u - 1$ should be $u - i$

p. 8 line 9: $m := 1$ should be $m := i$

p. 9 lines 2, 3, 4, 7, 11, 30, 32, 33, 42, 60:

(should be {
) should be }

line 15: $A[1]$ should be $A[I]$

line -13: $L - 1$ should be $L - I$

line -8: 2. should be 2:

line -1: FND should be END

p. 11 References should be numbered consecutively 1 - 13.

Errata

p. 5 line 20: $A[i] > r$ should be $A[i] < r$

p. 7 line 10: $u - 1$ should be $u - i$

p. 8 line 9: $m := 1$ should be $m := i$

p. 9 lines 2, 3, 4, 7, 11, 30, 32, 33, 42, 60:

(should be {
) should be }

line 15: $A[1]$ should be $A[I]$

line -13: $L - 1$ should be $L - I$

line -8: 2. should be 2:

line -1: FND should be END

p. 11 References should be numbered consecutively 1 - 13.

Errata

p. 5 line 20: $A[i] > r$ should be $A[i] < r$

p. 7 line 10: $u - 1$ should be $u - i$

p. 8 line 9: $m := 1$ should be $m := i$

p. 9 lines 2, 3, 4, 7, 11, 30, 32, 33, 42, 60:

(should be {
) should be }

line 15: $A[1]$ should be $A[I]$

line -13: $L - 1$ should be $L - I$

line -8: 2. should be 2:

line -1: FND should be END

p. 11 References should be numbered consecutively 1 - 13.

Errata

p. 5 line 20: $A[i] \succ r$ should be $A[i] \prec r$

p. 7 line 10: $u - 1$ should be $u - i$

p. 8 line 9: $m := 1$ should be $m := i$

p. 9 lines 2, 3, 4, 7, 11, 30, 32, 33, 42, 60:

(should be {
) should be }

line 15: $A[1]$ should be $A[I]$

line -13: $L - 1$ should be $L - I$

line -8: 2. should be 2:

line -1: FND should be END

p. 11 References should be numbered consecutively 1 - 13.

Errata

- p. 5 line 20: $A[i] > r$ should be $A[i] < r$
- p. 7 line 10: $u - 1$ should be $u - i$
- p. 8 line 9: $m := 1$ should be $m := i$
- p. 9 lines 2, 3, 4, 7, 11, 30, 32, 33, 42, 60:
(should be {
) should be }
- line 15: $A[1]$ should be $A[I]$
- line -13: $L - 1$ should be $L - I$
- line -8: 2. should be 2:
- line -1: FND should be END
- p. 11 References should be numbered consecutively 1 - 13.

Errata

p. 5 line 20: $A[i] \succ r$ should be $A[i] \leftarrow r$

p. 7 line 10: $u - 1$ should be $u - i$

p. 8 line 9: $m := 1$ should be $m := i$

p. 9 lines 2, 3, 4, 7, 11, 30, 32, 33, 42, 60:

(should be {
) should be }

line 15: $A[1]$ should be $A[I]$

line -13: $L - 1$ should be $L - I$

line -8: 2. should be 2:

line -1: FND should be END

p. 11 References should be numbered consecutively 1 - 13.