A generalized, one-way, stackless quicksort

Article *in* BIT. Numerical mathematics · March 1987

DOI: 10.1007/BF01937353

CITATIONS
7

READS
420

1 author:

Lutz Wegner
Universität Kassel
81 PUBLICATIONS **476** CITATIONS

BIT 27 (1987), 44–48

A GENERALIZED, ONE-WAY, STACKLESS QUICKSORT

LUTZ M. WEGNER*

IBM Heidelberg Scientific Center, Tiergartenstrasse 15, D-6900 Heidelberg, Germany

Abstract.

This note generalizes the one-way, stackless quicksort of Huang and Knuth to work for any type of sort key. It thus proves that quicksort can run with minimal space in $O(N \log N)$ average time.

AMS 68.E.05. C.R. F.2.2.

1. Introduction.

Lately, several variants of Hoare's quicksort [1, 2] have appeared. Some of the novel properties which were introduced are stability (equal keys retain their relative order) [3], smoothness (speed-up for presorted or multiset input) [4, 5], and minimal space (quicksort without a stack) [6].

The last property is achieved by a surprisingly simple algorithm, called OSORT, which scans the file in one direction only. This is a common technique mentioned e.g. in [7] as "Lomuto's method". It is extensively used for smooth variants in [5]. To do away with the stack, OSORT negates keys to indicate the end of a subfile yet to be sorted. As a consequence, this technique works only for positive numbers as input. Figure 1 below shows the PASCAL implementation from [6].

In this note we generalize the idea so that the new algorithm, called GOSORT, works for any type of key. Like OSORT, the resulting sort is not stable, i.e. equal keys do not retain their original relative positions and both are not truly competitive with the best quicksort versions [5, 8] in terms of speed. However, OSORT and GOSORT shed some light on an open problem, namely whether there is a stable, in situ, $O(N \log N)$ partition-exchange sort operating on an array (cf. [9] based on merging). The existence of such a method is to be expected if quicksort and mergesort are each representations of dual classes of sorts as claimed in [10].

Received August 1986. Revised December 1986.

* On leave from FH Fulda, D-6400 Fulda (Germany).

```
procedure OSORT (N: integer);
label 3;
var L: 1..M; (* progress indicator *)
    J: 1..M; (* end of current pivoting region *)
    I: 1..M; (* gap in current pivoting region *)
    P: entry; (* the pivot record *)
begin R[N+1].key := -1;
for L := 1 to N do
    begin while R[L].key > 0 do
          begin J := L; I := L; P := R[L];
          3: repeat J := J+1 until P.key > R[J].key;
          if R[J].key > 0 then
            begin R[I] := R[J]; I := I+1;
            R[J] := R[I]; goto 3;
            end;
          P.key := -P.key; R[I] := P;
          end;
    R[L].key := -R[L].key;
    end;
end;
```

Fig. 1. OSORT as PASCAL-Procedure.

2. The method.

We adopt the notation from [6]. Thus we wish to rearrange an array of N records $R_1 \ldots R_N$, where the records include keys $K_1 \ldots K_N$, such that the keys are in nondecreasing order. The basic idea is to do a one-way scan, successively partitioning the leftmost subfile. However, instead of negating a key as in OSORT to indicate the end of a subfile, we use the largest key from the subfile preceding it in the file, i.e. the left neighbour subfile. Let $R_L \ldots R_m$ $(1 \leq L < m \leq N)$ be the subfile currently being partitioned. If, in general, we partition subfiles subject to all keys in the left subfile being less than or equal the pivot, and all keys in the right subfile are greater than the pivot, then any key value less than or equal to the previous pivot, i.e. K_{L-1}, unambiguously indicates the end m of the subfile. We shall call such a record a "stopper" (sentinel). The record in position $m+1$ is then the next pivot and the next subfile starts at position $m+2$.

The partitioning is done as in OSORT with three pointers L, i, j, where L indicates the left end of the subfile, i the current gap for the pivot and j the currently examined record $(L \leq i \leq j)$. The outline of the algorithm is then as follows.

while file not completely sorted **do**
  if next subfile empty (L on stopper)
  then advance L to next subfile ($L := L+2$)
  else partition next subfile starting at L by making R_L the pivot and
    stopping the scan with j when $K_j \leq K_{L-1}$;
    let i be the new pivot position ($L \leq i < j$) and
    let K_{i-1} be the maximum key of the left subfile;
    if left subfile has length ≤ 2
    then move the old stopper R_j to its proper place $L-2$,
      move R_{i-1} as new stopper to j, advance L to $i+1$
    else swap the maximum in $i-1$ with the stopper in j.

Fig. 2. Outline of Stackless, One-way Quicksort.

The finer point is that, in partitioning, we keep the maximum of the left subfile in position $i-1$, i.e. immediately to the left of the gap for the pivot. This is easily achieved by means of one extra comparison for each key less than or equal to the pivot.

The second observation is the exchange of records R_{i-1} and R_j, where the later is the stopper from the subfile which was just partitioned. Because the old stopper was less than any key in the old subfile, it may serve again as stopper in the new left subfile because it is still smaller than any key in it. At the same time, the maximum from the left subfile serves well as stopper for the right subfile. Our choice of the maximum is arbitrary; with the maximum the stopper is restored to the reserved spot immediately to the left of the leftmost pivot. Taking any other key from the left subfile and letting it afterwards sink into its proper position is possible as well.

Due to the temporary dislocation of records to serve as stoppers, the invariant for the new algorithm is less obvious than in the case of OSORT.

1. records $R_1 \ldots R_{L-3}$ are sorted and $K_1 \ldots K_{L-3} \leq K_{L-1}$.
2. $K_L \ldots K_{i-1} \leq$ the current pivot key K.
3. $K_{i+1} \ldots K_{j-1} > K$.
4. Furthermore, $K_1 \ldots K_{L-3} K_{L-1} \ldots K_{i-1} K_{i+1} \ldots K_{N+1} K$ is a permutation of the original keys, where K_{N+1} is the stopper for the rightmost subfile and K is pivot for the gap at i.

Note how the rightmost key in each subfile has been shifted one subfile to the right leaving another gap at $L-2$.

3. A PASCAL-implementation.

To avoid additional index comparisons, we assume that we have stopper values which are less than the minimal key of the input, e.g. $-MAXINT$ for keytype

integer, in the positions 0 and $N+1$ of the sorting array, which we declared as R: **array**$[-1 \ldots N+1]$ **of** ENTRY. The innermost repeat-loop has been taken from [6] with "$>$" changed to "$> =$".

```
procedure GOSORT (N: integer; MINVALUE: keytype);
label 3;
var I : 1..NMAX;      (* current pivot location *)
    J : 1..NMAX+1;    (* end of current subfile *)
    L : 1..NMAX+2;    (* left end of subfile    *)
    P : ENTRY;        (* pivot record           *)

begin (* GOSORT *)
   L := 1; R[N+1].key := MINVALUE; R[0].key := MINVALUE;
  while L <= N+1 do
    if R[L].key <= R[L-1].key
    then (* R[L] is stopper; an empty subfile *)
      begin R[L-2] := R[L]; L := L+2 end
    else
      begin J := L; I := L; P := R[L];
        3: repeat J := J+1 until P.key >= R[J].key;
        if R[J].key > R[L-1].key
        then (* J not on stopper *)
          begin
            if R[J].key >= R[I-1].key
            then R[I] := R[J] (*R[I] must be maximum of left subfile *)
            else begin R[I] := R[I-1]; R[I-1] := R[J] end;
            I := I+1; R[J] := R[I]; goto 3
          end
        else (* J on stopper *)
          begin R[I] := P;
            if L+2 >= I
            then begin R[L-2] := R[J] := R[I-1]; L := I+1 end
            else begin P := R[I-1]; R[I-1] := R[J]; R[J] := P end
        end
      end (* partition a non-empty subfile, end of while *)
end (* GOSORT *)
```

Fig. 3. GOSORT as PASCAL-Procedure.

4. Conclusion.

GOSORT is not one of the faster quicksort versions and will even be slower than OSORT because it has to maintain the maximum for the left subfile.

OSORT in turn was shown in [6] to be about 60% slower than the most refined versions. The precise performance of GOSORT cannot be analyzed using the common recurrence relations because the subfiles are certainly not in random order. However, it is clear that GOSORT is an $O(N \log N)$ average time minimum space quicksort and that GOSORT degenerates to $O(N^2)$ for sorted or multiset input. Modifications along the line of [5] to achieve smoothness are possible but are omitted here. The main result of this note is then that the common text book phrase "quicksort's run-time stack can be limited to logarithmic size by sorting the shorter subfile first" should be replaced by the phrase "quicksort's run-time stack can be eliminated if a stopper technique for unsorted subfiles is used".

Note added in proof.

Following the submission of this note it was learned that a similar stackless Quicksort has independently been discoverd by B. Durian and was presented at MFCS 1986 in Bratislava, Czechoslovakia. The Durian algorithm is not one-way and thus uses a second pass to find the stopper. Furthermore, the selection of the stopper is different.

Acknowledgements.

I like to thank H.-W. Six for a helpful hint and the colleagues at the IBM scientific center for the friendly introduction to their systems.

REFERENCES

1. C. A. R. Hoare, *Quicksort*, *The Computer Journal*, 5 (1962), 10–15.
2. Donald E. Knuth, *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Reading, Mass.: Addison-Wesley, 1975.
3. D. Motzkin, *A stable quicksort*, *Softw. Pract. Exper.* 11, 6 (June 1981), 607–611.
4. L. M. Wegner, *Sorting a linked list with equal keys*, *Inf. Processing Lett.*, 15, 5 (Dec. 1982), 205–208.
5. L. M. Wegner, *Quicksort for equal keys*, *IEEE TC*, 34, 4 (April 1985), 362–367.
6. Huang Bing-Chao and Donald E. Knuth, *A one-way, stackless quicksort algorithm*, *BIT*, 26 (1986), 127–130.
7. J. Bentley, *Programming pearls: How to sort*, *Comm. ACM*, 27, 4 (April 1984), 287–291.
8. R. Sedgewick, *Implementing quicksort programs*, *Comm. ACM*, 21, 10 (Oct. 1978), 847–857 and 22, 6 (June 1979), 368.
9. L. Trabb Pardo, *Stable sorting and merging with optimal space and time bounds*, *SIAM J. Comput.*, 6, 2 (June 1977), 351–372.
10. S. M. Merritt, *An inverted taxonomy of sorting algorithms*, *Comm. ACM*, 28, 1 (Jan. 1985), 96–99.