QUICKSORT WITHOUT A STACK

Branislav Ďurian
VÚVT Žilina, k.ú.o, Nerudová 33, 01001 Žilina, Czechoslovakia

ABSTRACT: The standard Quicksort algorithm requires a stack of size $O(\log_2 n)$ to sort a set of n elements. We introduce a simple nonrecursive version of Quicksort, which requires only a constant, O(1) additional space because the unsorted subsets are searched instead of stacking their boundaries as in the standard Quicksort. Our O(1)-space Quicksort is probably the most efficient of all the sorting algorithms which need a constant workspace only.

KEYWORDS: Algorithm, O(1)-space, Quicksort, Searching, Sorting, Stack.

1. Introduction

The Quicksort [1,2] is the most favourite sorting algorithm for its simplicity, elegancy, efficiency and universal character. In this paper, we suggest a version of Quicksort, which does not make use of a stack. In our nonrecursive version of the Quicksort (from now IQSORT) which needs only a few variables for the control of sorting (thus: O(1)-space algorithm), the operations on stack are replaced by the repeated search for the bounds of the next unsorted subset. The additional costs for this space complexity reduction are surprisingly low, the modification is simple (see programs in the Section 3) and our IQSORT with an efficient search for the bounds (see the Section 4) is approximately only by some 4-8% slower than the standard Quicksort with a stack (from now QSORT). Our O(1)-space Quicksort is probably the most efficient of all the sorting algorithms which need a constant workspace only.

2. Method description

The Quicksort is one of the most successful applications of the divide and conquer method. In the decomposition step one can choose an element in the set S as a pivot p, by which S is rearranged into subsets S_1, S_2 of elements < and > than p, respectively. The elements equal to p are assumed to appear as in [2] (see procedures in Section 3, too).

For the time and space efficiency the recursion from the Quicksort has been eliminated and replaced by the explicite stack manipulations. However, is it necessary to use a stack? Our answer is not! Briefly; we can search for the bounds of the next subproblem instead of storing them in the stack in a certain previous step.

In the QSORT after the decomposition step left and right bounds of set S_2 are pushed on the stack and S_1 is dealt with further. Instead, our suggestion is to interchange the first element of S_2 with the pivot p´, splitting thereby set S and next set S´ as in Fig. 1.

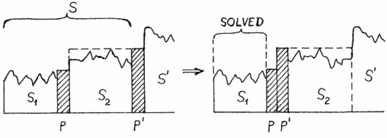

Fig. 1.

Now, the ´borrowed´ pivot p´ is the first element in the set S_2 and it is greater than or equal to all the other elements of set S_2. It will be used for the search of the right-hand side bound of the set S_2 later. The decomposition of the set S_1 follows.

When the decomposition of the set S_1 is over, we suggest - instead of the usual popping of the bounds from the stack - the following:

A) Choose the left-hand side bound of the set S_2, which is situated behind the pivot p, to separate S_1 and S_2. The first element not equal to p´ (of the set S_2) is searched. This is the ´borrowed´ pivot p´.

B) The first element greater than p´ at the right of p´ is searched, i.e an element which definitely doesn´t belong to the set S_2, but which belongs to the next set S´ (Fig. 2a.).

C) The ´borrowed´ pivot p´ will be returned into the original place between the sets S_2 and S´ as in Fig. 2b. (pivot p´ is interchanged with the last element of the set S_2).

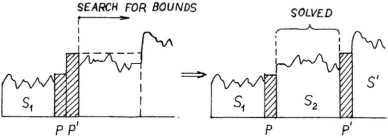

Fig. 2a. Fig. 2b.

When the right-hand side bound of the set S_2 is found, one can continue in the decomposition of the set S_2. The algorithm will finish when the chosen left-hand side bound is ´behind´ (i.e. right to) the original set S.

3. Program description

In this Section, we will show how the O(1)-space version can be easy developed from the stack version of Quicksort. The both procedures are written in a PASCAL. Well-known codes for the interchanging of two elements (swap) and for sorting in the final phase (insertsort) are not given here. Let the set of sorted elements be represented as the array A[1..n].

For simplicity we will begin with the simple stack version of the Quicksort (procedure SQS). The stack is created in the array STACK.

```
procedure SQS(var n: integer);
{ the simple stack-version of QUICKSORT to sort elements in
  A[1..n], sentinel: A[n+1]:=maxval, maxval=∞.
  Another improvements from [2] can be applied, too.}
var i,j,l,r,m,s: integer;
var p: integer;  {keytype: integer or real or array of characters}
begin
  l:=1;r:=n+1;m:=9;s:=0;
  repeat begin
    while r-l>m do begin
      i:=l;j:=r;
      p:=A[l];  {or choosing the pivot as the median of 3 elements}
      repeat
        repeat i:=i+1; until A[i]>=p;
        repeat j:=j-1; until A[j]<=p;
        if i<j then swap(A[i],A[j]);
      until i>=j;
      A[l]:=A[j];A[j]:=p;
      STACK[s]:=r;s:=s+1;     {the stack operations in [2] are more
                                complex }
      r:=j
    end;
    l:=r+1;
    if l<=n then begin
       s:=s-1;r:=STACK[s]
    end;
  end;
  until l>n;
  insertsort(n)
end.
```

The stack operations in [2] are more complicated to guarantee the stack size $O(\log_2 n)$. For the comparison with the procedure SQS we present the procedure IQS for IQSORT immediately. Our solution is simpler and needs only a constant workspace. In the Section 4 we are interested in the question in what degree the search for bounds can affects the total sorting time.

```
procedure IQS(var n: integer);
 {the O(1)-space version of QUICKSORT to sort elements in
  A[1..n], sentinels: A[n+1]:=maxval-1, A[n+2]:=maxval,  maxval=∞
  Another improvements from [2] can be applied, too.}
var i,j,l,r,m: integer;
var p: integer;
begin
  l:=1;r:=n+1;m:=9;
  repeat begin
    while r-l>m do begin
      i:=l;j:=r;
      p:=A[l];  {or choosing the pivot as the median of 3 elements}
      repeat
        repeat i:=i+1; until A[i]>=p;
        repeat j:=j-1; until A[j]<=p;
        if i<j then swap(A[i],A[j]);
      until i>=j;
      A[l]:=A[j];A[j]:=p;
      swap(A[i],A[r]);   {instead of pushing on the stack}
      r:=j
    end;
    l:=r;repeat l:=l+1; until A[l]<>p;
    if l<=n then begin   {instead of popping from the stack
                          sequential search for bounds follows}
      p:=A[l];r:=l;
      repeat r:=r+1; until A[r]>p;
      r:=r-1;A[l]:=A[r];A[r]:=p
    end;
  end;
  until l>n;
  insertsort(n)
end.
```

4. The search for bounds

The sequential (linear) search implemented in the program IQS of the Section 3 is the simplest, but also the least efficient one. It is easy to analyze it, since the search for the right bound (line 15) is exactly opposite to the changes of the variable j (line 6) in the inner loop. Therefore, it consumes a half of the time which is needed for the indexes i and j to cross in the inner loop (lines 5-6). Now, (empirically, for n=1000) some 60% of total running time is spent in partitioning of QSORT. Thus the equivalent O(1)-space version should be slower by some 30%. For the model in [2] (assembly language similar to Knuth's MIX [1]) the total expected running time of procedure IQS (SQS) is 15.6667 n ln n - 5.253 n (11.6667 n ln n - 0.555 n) [3]. This is better performance than that of the Heapsort [1] which is O(1)-space, too, and runs approximately (in the expected case) twice slower than the QSORT.

These results can be considerably improved by making use of searching methods superior to the sequential search [1]. For instance, we can proceed as follows:

A) The redundant search for the right-hand side bound of the upper subarray (after the last decomposition) can be removed if the lines 10-11 are changed to

```
10    if j-l<=m then l:=i else begin swap(A[i],A[r]);
11                                   r:=j
                              end;
```

B) For the search in the subarray A[l..n] we suggest to use the sequential search with the step s, s>1 (the best of all is s=10 for n=1000, s=12 for n=10000 [3]). Between lines 14,15 we can insert

```
14a   if A[n-s]> p then begin
14b     repeat r:=r+s; until A[r]>=p;
14c     r:=r-s
      end
14d   else r:=n-s;
```

Improving the search in this way one can show (by implementations in PASCAL, C-language, for n=1000-10000) that the IQSORT is slower only by 4-8% than QSORT . This is verified by using model in [2], where the total expected running time is 11.8333 n ln n + 2.265 n, for m=9, s=12 [3]. Therefore, the cost paid for the excluded workspace ($O(\log_2 n)$) is low, indeed.

Acknowledgment:

The author would like to thank dr. S. Dvořák, dr. D. Pokorná and dr. J. Vyskoč for helpful suggestions.

REFERENCES

[1] D. E. Knuth: The Art of Computer Programming, Vol. III: Sorting and Searching. Addison-Wesley Reading, MA, 2nd ed. 1975.

[2] R. Sedgewick: Implementing QUICKSORT Programs. Comm. ACM, Vol 21, No. 10, 1978, 847-856.

[3] B. Ďurian: Quicksort without a stack: Design and analysis. Unpublished, 1986.