Department
of
APPLIED MATHEMATICS

An algorithm for internal merging of two subsets with small extra storage requirements.

by

Terje O. Espelid

Report No. 50 September 1974

UNIVERSITY OF BERGEN
Bergen, Norway

0084-778X

An algorithm for internal merging of two subsets with small extra storage requirements.

by

Terje O. Espelid

Report No. 50 September 1974

Abstract

An algorithm for internal merging of two disjoint linearly ordered subsets into one set is given and analysed. The two subsets are supposed to be given in one array with interlacing elements. The algorithm is based on an interchanging of elements and requires therefore only a fixed amount of extra storage.

Introduction

Given a 2-ordered array a[1:n], that is

(1) $a[i] \le a[i+2]$, $i = 1(1)n-2$,

see Knuth [3] p.86.

The array consists of two disjoint linearly ordered subsets. Now the problem is to rearrange the elements in a[1:n] such that the resulting array a'[1:n] is 1-ordered.

(2) $a'[i] \le a'[i+1]$, $i = 1(1)n-1$.

This obviously is a merging problem and can be solved in different ways with and without extra storage. In this paper we will only consider methods which work without using a working area. For the purpose of analysis we will assume that $a[i] \neq a[j]$, $\forall i \neq j$. Furthermore the possible permutations of the elements indexes in a' relative to a , $\binom{n}{\lfloor n/2 \rfloor}$*, are supposed to be equiprobable.

Sifting or straight insertion is one wellknown algorithm which can be used. This algorithm is a sorting algorithm, that is, it does not take into account the special form, (1), of the array in question. On the other hand this algorithm is easy to program and analysis in Espelid [2] and [3], shows that the number of comparisons in average will be

$$C_S(n) \approx .1566643\, n\sqrt{n} + n .$$

* $\lfloor x \rfloor$ ($\lceil x \rceil$) means the greatest (smallest) integer not greater (smaller) than x .

Merging algorithms which make use of a working area will need maximum $n-1$ comparisons to merge the subsets in $a[1:n]$. The sifting algorithm therefore makes a lot of superfluous comparisons. It is reasonable however that a method which makes use of a minimum of extra storage has to pay for it in longer running time, compared to usual merging algorithms. Batcher's (parallel) method, see [1], is a sorting algorithm which is based on merging the subsets of 2-ordered arrays taking into account that the arrays really are 2-ordered, and thus reducing the number of comparisons in average compared to sifting. We find that using the main idea in Batcher's method the number of comparisons (independent of a) will be

$$C_B(n) \approx \frac{1}{2} n \lceil \log_2 n \rceil - \frac{1}{2} n .$$

The number of comparisons is reduced compared to sifting but unfortunately the amount of bookkeeping needed to control the sequence of comparisons is rather large. Which of the two methods one should choose is not obvious and needs a more thorough analysis. The main power of Batcher's method lies in the possibility of parallel processing.

Still another method on a related problem is given by Kronrod [4]. Kronrod's algorithm seems complicated but he succeeds in forcing the number of comparisons down to

$$C_K(n) = O(n) .$$

The number of comparisons when using sifting will at minimum be $n-1$ and at maximum be $O(n^2)$. We get the maximum when

a[1] is greater than the element with largest even index, a[2⌊n/2⌋]. The number of exchanges in average will be of the same magnitude as the number of comparisons for all three methods.

We now consider a new method which uses at maximum n-1 comparisons to do the merging. The number of exchanges however turn out to be only slightly better than in sifting.

A merge exchange algorithm

To clarify the idea we will consider an example. Suppose that a[1:9] is given by

i	1	2	3	4	5	6	7	8	9
a[i]	5	1	6	2	7	3	9	4	10

The sifting algorithm will need 17 comparisons and 10 exchanges to sort this array. By comparing a[1] to a[i], i = 2,4,6,8 only 4 comparisons are needed. Now the problem is how to exchange elements without using more than one intermediate record, say w, such that at least one element comes to its final position in each step. We start with

$w \leftarrow a[1]$

then the sequence

$$a[1] \leftarrow a[2\times 1] \ ; \ a[2] \leftarrow a[2\times 2] \ ; \ a[4] \leftarrow a[2\times 4] \ ;$$

Now a[8] is free to use and we move elements in the opposite direction

$$a[8] \leftarrow a[8-2^0] \ ; \ a[7] \leftarrow a[7-2^1] \ ; \ a[5] \leftarrow w \ ;$$

The last move had to be made because the original $a[5-2^2]$ has been moved to w at the beginning. Now the array is changed to

1 2 6 4 5 3 7 9 10

where an arrow shows the direction in which an element has been moved, connecting the positions involved. We have to finish with

$$w \leftarrow a[3] \ ; \ a[3] \leftarrow a[2\times 3] \ ; \ a[6] \leftarrow w \ ;$$

This completes the merging in 8 moves, neglecting the moves from a to w .

One could now extend the problem putting a[10] = 8 . 5 comparisons are needed to state that a[8] < a[1] < a[10] . The same procedure now gives in the first step

i	1	2	3	4	5	6	7	8	9	10	. . .
a"[i]	1	2	3	4	5	6	7	9	10	8	. . .

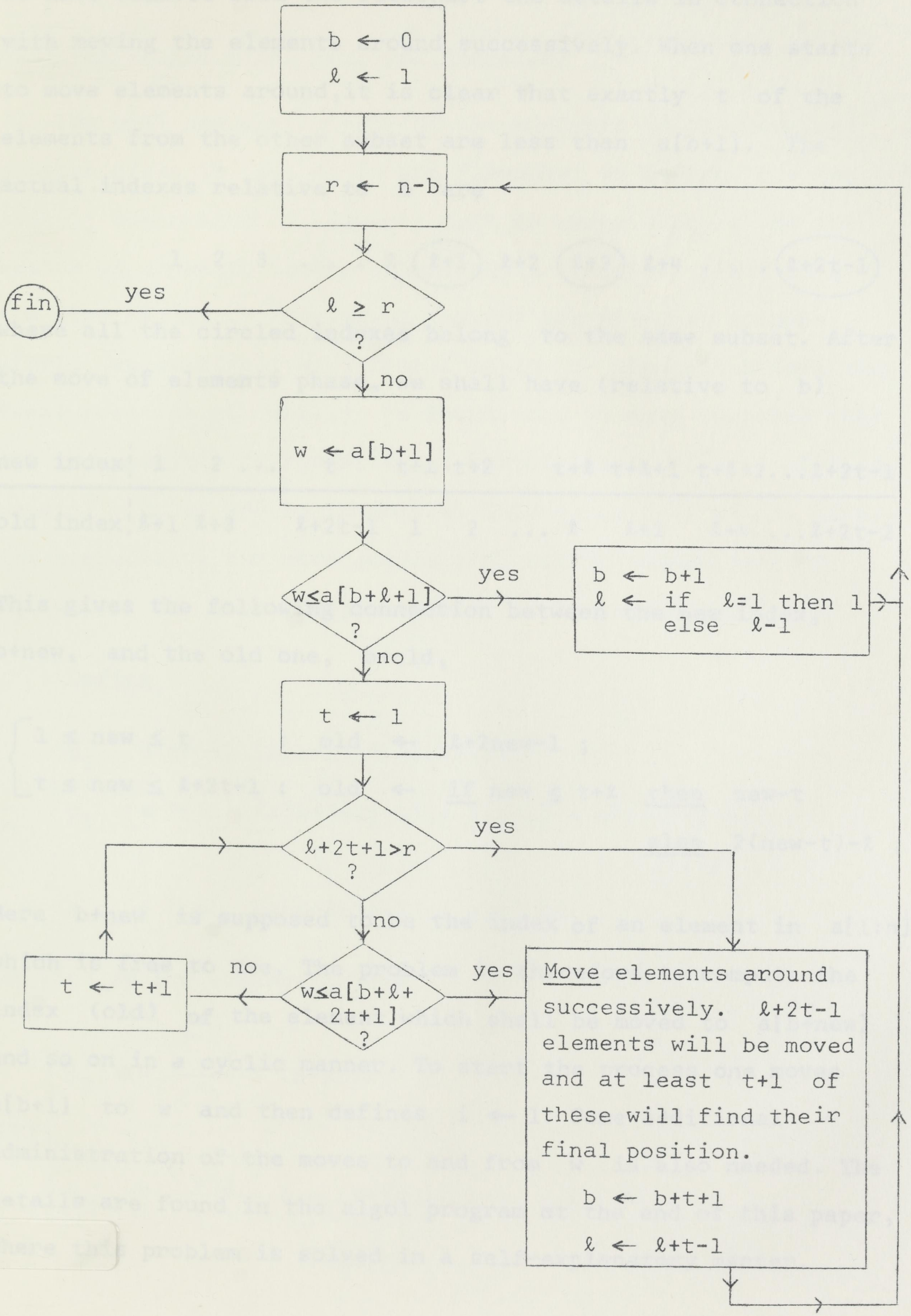
We know that the elements a"[1:5] are correctly sorted and are less than the other elements. By two comparisons we find that a"[6] and a"[7] both are less than a"[10] (= a[10]) such that the first 7 elements have found the final position.

The sorting problem left is not exactly of the original form. We still have two subsets but the interlacing character starts with the second element. We therefore need to generalize the problem slightly. Let us now leave the example.

Given an array a[1:n] where the b first elements have found their final position, the elements a[b+1], ... ,a[n] represent the merging problem remaining. These elements consist of two disjoint linearly-ordered subsets. The first ℓ elements, $1 \le \ell \le n-b$, all belong to one of the subsets, element $\ell+1$ belongs to the other subset and from this point the array is 2-ordered, that is

$$
\begin{array}{ll}
 & a[b+1] \le a[b+2] \le \ldots \le a[b+\ell] \le a[b+\ell+2] \le a[b+\ell+4] \le \ldots \\
(6) & \\
 & a[b+\ell+1] \le a[b+\ell+3] \le \ldots
\end{array}
$$

The following flowchart gives the main points in the algorithm:


b ← 0
ℓ ← 1
r ← n-b
fin
yes
ℓ ≥ r ?
no
w ← a[b+1]
w≤a[b+ℓ+1] ?
yes
b ← b+1
ℓ ← if ℓ=1 then 1 else ℓ-1
no
t ← 1
ℓ+2t+1>r ?
yes
no
t ← t+1
no
w≤a[b+ℓ+2t+1] ?
yes
Move elements around successively. ℓ+2t-1 elements will be moved and at least t+1 of these will find their final position.
b ← b+t+1
ℓ ← ℓ+t-1

We have removed from the flowchart the details in connection with moving the elements around successively. When one starts to move elements around, it is clear that exactly t of the elements from the other subset are less than a[b+1]. The actual indexes relative to b are

$$1 \quad 2 \quad 3 \quad . \; . \; . \; \ell \quad \ell+1 \quad \ell+2 \quad \ell+3 \quad \ell+4 \quad . \; . \; . \quad \ell+2t-1$$

where all the circled indexes belong to the same subset. After the move of elements phase, we shall have (relative to b)

new index	1	2 ...	t	t+1	t+2		t+ℓ	t+ℓ+1	t+ℓ+2...ℓ+2t-1
old index	ℓ+1	ℓ+3	ℓ+2t-1	1	2	...	ℓ	ℓ+2	ℓ+4 ...ℓ+2t-2

This gives the following connection between the new index, b+new, and the old one, b+old,

$$\begin{cases} 1 \le \text{new} \le t & : \ \text{old} \leftarrow \ell+2\text{new}-1 \ ; \\ t \le \text{new} \le \ell+2t-1 & : \ \text{old} \leftarrow \underline{\text{if}}\ \text{new} \le t+\ell \ \underline{\text{then}}\ \text{new}-t \\ & \qquad\qquad \underline{\text{else}}\ 2(\text{new}-t)-\ell \ ; \end{cases}$$

Here b+new is supposed to be the index of an element in a[1:n] which is free to use. The problem is therefore to compute the index (old) of the element which shall be moved to a[b+new] and so on in a cyclic manner. To start the process one moves a[b+1] to w and then defines $i \leftarrow 1$. Some additional administration of the moves to and from w is also needed. The details are found in the algol program at the end of this paper, where this problem is solved in a self-explanatory manner.

Analysis of the merge exchange algorithm

We will now concentrate on the analysis of the merging problem given in (6). As in the flowchart we define $r \equiv n-b$, that is the number of elements left to merge. Let s be the number of elements in the subset which contains $a[b+\ell+1]$, $s \equiv \lfloor (r+1-\ell)/2 \rfloor$. This means that we are going to compare $a[b+1]$ successively with from 1 up to s elements before the final position of $a[b+1]$ is found. Now we have supposed that all the possible different final permutations, $\binom{r}{s}$, of this merging problem are equiprobable. This means that we can give the probability for each possible number of comparisons or equivalent for each possible number of exchanges (or moves).

Define

$$(7)\quad \begin{cases} p_1 & \equiv \text{Prob } \{a[b+1] < a[b+\ell+1]\} \\ p_j & \equiv \text{Prob } \{a[b+\ell+2j-3] < a[b+1] < a[b+\ell+2j-1]\}, \\ & \qquad\qquad j = 2(1)s . \\ p_{s+1} & \equiv \text{Prob } \{a[b+\ell+2s-1] < a[b+1]\} \end{cases}$$

We find by our assumption that

$$(8)\quad \begin{cases} p_1 & = \dfrac{\binom{r-1}{s}}{\binom{r}{s}} = (r-s)/r \\ \\ p_{j+1} & = \dfrac{\binom{r-j-1}{s-j}}{\binom{r}{s}} = p_j(s-j+1)/(r-j) \;, \; j = 1(1)s . \end{cases}$$

Now let $c_{r,\ell}$ and $e_{r,\ell}$ mean the average number of comparisons and exchanges (or moves) needed to merge a[b+1:n] by the algorithm. Note that $c_{r,r} = e_{r,r} = 0$. We find the following recurrence relations

(9) $$c_{r,\ell} = p_1(c_{r-1,\max(1,\ell-1)} + 1) + p_{s+1}\cdot s + \sum_{j=2}^{s} p_j(c_{r-j,\ell+j-2} + j)\ ,\quad \ell = 1(1)r-1$$
$$\text{and}\quad s \equiv \lfloor (r+\ell-1)/2 \rfloor$$

(10) $$e_{r,\ell} = p_1(e_{r-1,\max(1,\ell-1)} + p_{s+1}\cdot(\ell+2s-1) + \sum_{j=2}^{s} p_j(e_{r-j,\ell+j-2} + \ell+2j-3)\ ,\quad \ell = 1(1)r-1$$
$$\text{and}\quad s \equiv \lfloor (r+\ell-1)/2 \rfloor$$

We note that the comparisons in our algorithm behave just like an ordinary merge algorithm on two disjoint linearly ordered subsets. This means that $c_{r,s}$ is given by

(11) $$c_{r,\ell} = s(r-s)/(s+1) + s(r-s)/(r-s+1)\ ,$$

see for example [3].

This gives

(12) $$c_{r,1} = \Big(1/(1+\lfloor r/2 \rfloor) + 1/(1+\lceil r/2 \rceil)\Big)\cdot\lfloor r/2 \rfloor\cdot\lceil r/2 \rceil\ .$$

To solve (10) seems considerably more difficult. Using (10) we have computed the first values of $e_{r,\ell}$, $\ell = 1(1)r-1$ in table 1.

r						
2	1.0000					
3	1.3333	1.6667				
4	2.3333	1.7500	2.2500			
5	2.8000	3.3000	2.2000	2.8000		
6	3.8500	4.4667	4.2667	2.6667	3.3333	
7	4.4000	5.0286	4.2381	5.2381	3.1428	3.8571

$e_{r,\ell}$, $\ell = 1(1)r-1$

Table 1.

We are now interested in finding an approximate expression for $e_{r,1}$. Therefore we tabulate, using (10), $e_{r,1}$ for $r = 2^j$, $j = 1(1)8$ in table 2.

r	$e_{r,1}$
2	1.00000 00000
4	2.33333 33333
8	5.50000 00000
16	13.04607 61461
32	31.21811 86596
64	75.73013 95663
128	187.05514 13464
256	471.62559 05589

Table 2.

Using the results in [2,3] we make the guess that

(13) $$g_r \equiv e_{r,1}/r = \alpha r^{\frac{1}{2}} + \beta + \gamma r^{-\frac{1}{2}} + \delta r^{-1} + \ldots$$

Using extrapolations on the values in table 2 and taking into account the form (13) we get the results in table 3 defining

$$h_r = (g_{2r} - g_r)(\sqrt{2} + 1)/\sqrt{r}$$

This extrapolation turns out to be rather successful and gives

$$\alpha \approx .07833215$$

Using the same method to estimate β and γ one finds

$$\begin{cases} \beta \approx .6250 \\ \gamma \approx -.607 \end{cases}$$

Note that

$$2\alpha \approx .1566643$$

which seems to be exactly the same constant as given in [2] for the sifting algorithm used on the same problem. Knuth shows in his analysis [3] that this constant is

$$2\alpha \approx \sqrt{\pi/128}\ .$$

It is remarkable however that the numerical procedure used in [2] comes out with 7 significant digits.

Table 3

r	g_r	h_r	2-extra-polation	$2\sqrt{2}$-extra-polation	4-extra-polation	$4\sqrt{2}$-extra-polation	8-extra-polation	$8\sqrt{2}$-extra-polation
2	.5							
		.14225890						
4	.58333333		.10922168					
		.12574029		.08345379				
8	.6875		.09256411		.07836994			
		.10915220		.07964090		.07831123		
16	.81537976		.08420995		.07832161		.07833413	
		.09668107		.07865143		.07833127		.07833215
32	.97556621		.08061666		.07832956		.07833233	
		.08864887		.07841003		.07833219		
64	1.18328343		.07919019		.07833173			
		.08391953		.07835130				
128	1.46136829		.07864789					
		.08128371						
256	1.84228746							

The computations have been done with 10 decimals and the numbers in this table are given correctly rounded to 8 decimals.

Conclusions

We have found that

$$C_M(n) = (1/(\lfloor n/2\rfloor+1) + 1/(\lceil n/2\rceil+1))\lceil n/2\rceil\lfloor n/2\rfloor \approx n - 2$$

and

$$E_M(n) \approx .078332\ n^{3/2} + .6250\ n - .607\ n^{1/2}$$

for this new merge exchange algorithm. This makes the algorithm considerably better than sifting when only comparisons and moves are taken into account. Asymptotically we will have $C_M(n) + E_M(n) \sim 1/4(C_S(n) + E_S(n))$. In table 4 these expressions are compared for

n	$C_S(n) + E_S(n)$	$C_M(n) + E_M(n)$	
16	35	27	.77
32	88	61	.69
64	224	138	.62
128	581	313	.54
256	1539	726	.471

Table 4

smaller values of n.

Unfortunately the amount of book-keeping in this new method, just as for Batcher's parallel method, is rather large. One way to improve the method might be to increase the working area. The author has not studied how this might influence the book-keeping problem, and the question how much this would reduce the amount of work therefore remains open.

The Algol program

The algol program given has been written only to show how this merge exchange algorithm works. As is seen by a first glance, the program is not optimal. One has tried to use the same notation in this program as in the text. Introducing too many new helpvariables has been avoided although this would have speeded up the program. The author hopes that this fact combined with reading the text makes the program selfexplanatory without too many comments.

```
procedure   Merge_exchange (a,n);
            value n; integer n; integer array a;

comment     This procedure transforms the 2-ordered array
            a[1:n]  to a 1-ordered array  a[1:n]  with small
            extra storage requirements;

integer     b, j, ℓ, r, t, w, new, old, count, windex;
            b ← 0; ℓ ← 1;  comment  b: basis pointer,
            ℓ: see text;
start :     r ← n-b; comment  r: number of elements left;
            if  ℓ ≥ r  then go to  fin;
            w ← a[b+1]; j ← b+ℓ+1;
            if  w ≤ a[j]  then
                begin  b ← b+1; ℓ ←  if  ℓ=1  then  1  else  ℓ-1;
                    go to  start;  end  w  has found its final
                                                         place;
            t ← 1;
            for  j ← j+2  while  j ≤ n  do
                if  w ≤ a[j]  then go to  move  else  t ← t+1;
```

```
move :      count ← 0; windex ← b+1; new ← b+1;
            comment We shall move  ℓ+2t-1  elements.
                    Control : count;

right :     old ← 2×new-b+ℓ-1; count ← count+1;
            a[new] ← a[old]; new ← old;
            if  new ≤ b+t  then go to  right;
left :      old ← if  new ≤ b+ℓ+t  then  new-t  else  2×(new-t)
            -ℓ-b;
            comment  Now there is the chance that old is equal
            to windex. In this case a cycle is finished and we
            have to check if there is more work to be done;
            if  old=windex  then
            begin  a[new] ← w;  count ← count+1;
                if  count=ℓ+2×t-1  then go to  continue;
                for  new ← windex+1  step  1  until  b+ℓ,
                b+ℓ+2  step  2  until  b+2×t-1  do begin
                if  a[new] ≥ w  then begin  w ← a[new];
                windex ← new; go to  right  end; end;
            end  The case old=windex is finished;
            a[new] ← a[old];  new ← old;  count ← count+1
            go   if new<b+t  then  right  else  left;
continue :  b ← b+t-1; ℓ ← ℓ+t-1; go to start
fin :       end  Merge—exchange;
```

References

[1] Batcher, K.E.: "Sorting networks and their application" Proc. AFIPS Spring Joint Comp. Conf. 32(1968) 307-314.

[2] Espelid, T.O.: "Analysis of a Shellsort Algorithm" BIT 13,4 (1973) 394-400.

[3] Knuth, D.E.: "The Art of Computer Programming", Vol. 3, Addison Wesley, Reading, Mass. 1973.

[4] Kronrod, M.A.: "Optimal ordering algorithm without operational field" Soviet Math. Dokl. 10 (1969) 744-746 (Russian).