THE UN-MERGING PROBLEM

Nicola Santoro
School of Computer Science
Carleton University
Ottawa, K1S 5B6
CANADA

The purpose of this note is to share an intriguing problem which seems to open several unexpected questions. Many of the readers might already have heard of it (in one of its many formulations) since I have described the problem on several occasions to many people in the "algorithm" community; since no definite solution has been found so far, I have decided to make the problem "public".

Following are three formulations of the problem (it is up to you to choose the one you are most comfortable with).

F1. Given an array containing two types of elements, say **red** and **green**, we want to rearrange the array so that all the **red** elements appear before the **green** ones while maintaining the relative ordering between the **reds** as well as between the **greens** (i.e., if a and b are both **red** (**green**) and a was originally stored before b, then a must still be stored before b after the rearrangment). This can be trivially accomplished in time linear in the size n of the array if O(n) workspace is available (counting both comparisons and data movements). **What is the complexity if only O(1) workspace is available?**

F2. In the process of sorting an array, you are told that all elements smaller than 354 are already sorted, and so are the elements greater than 354. **Having only O(1) workspace, how long does it takes you now to sort the array using this information?**

F3. You are given a sorted array C which was obtained by merging two sorted arrays A and B. You now want to **undo** the merging in situ; i.e., move all the elements which came from A before the elements which came from B maintaining both of them sorted. Assume that you can test in O(1) time from which array an element in C originally came. **What is the cost of unmerging in situ?**

Since in-situ merging can be done in linear time, the immediate answer (at least, mine and of those who were posed the question) is: **linear**. However, when trying to produce a solution algorithm, the best bound achieved was (and still is) O(n log n).

This opens a set of questions. The first is obviously

Q1. Is it possible to unmerge in-situ in linear time?

The other questions are meaningful only if the answer to Q1 is negative:

Q2. Why ? Why can't we just apply an 'in-situ' merging algorithm backward? What makes in-situ un-merging intrinsically more difficult than in-situ merging?

Q3. Is O(**n** log **n**) the best we can do? Such a lower bound would imply that even if you know (for free) that all the elements greater than k are already sorted and so are those smaller, you still have to spend O(n log n) to complete the sort in-situ; or, in other words, size of the workspace is more important than this knowledge about pre-sortness.

Q4. If it cannot be done using constant workspace, then what is the smallest amount of extra storage needed to obtain a linear bound?

An observation (made by Shmuel Zaks [Technion]), which supports an affirmative answer to Q2 is that (by taking **red**=0 and **green**=1) any network solving this problem must have complexity O(**n** log **n**).

To obtain an O(**n** log **n**) algorithm is not difficult: any in-situ sorting algorithm can be modified to solve the problem using O(1) workspace.

An elegant O(**n** log **n**) solution (suggested by Mike Atkinson [Carleton U.] upon recalling one of Bentley's "pearls") is the following. First observe that (using Formulation 1), if the array is composed of a sequence of **greens** followed by a sequence of **reds**, then we can solve the problem (call it a "switch") in linear time (easy to prove). In general, the array will be an alternation of sequences of consecutive **reds** and of consecutive **greens** , as shown below

$$R\ G\ R\ G\ R\ G\ R\ G\ R\ G\ \ldots..$$

If we now "switch" the neighbouring sequences underlined below

$$R\ \underline{G\ R}\ G\ \underline{R\ G}\ R\ \underline{G\ R}\ G\ \ldots.$$

we obtain

$$R\ R\ G\ G\ G\ R\ R\ R\ G\ G\ \ldots.$$

thus reducing the number of sequences of consecutive **reds** and consecutive **greens** by one third in one "pass"; we can now repeat the same process until the desired result is achieved. Since each "pass" can be performed in linear time with O(1) workspace, the O(**n** log **n**) worst case bound follows.

Is there any other 'neat' algorithm for this problem?

As for Q4, I have observed that we can unmerge in linear time with O($n^{1/2}$) workspace: just partition the array into $n^{1/2}$ groups of consecutive $n^{1/2}$ elements each, and use the O($n^{1/2}$) workspace to solve the problem in each group (in total linear time); the resulting number of alternating sequences is now at most $2n^{1/2}$ which allows for a linear solution using O($n^{1/2}$) workspace.