Sorting almost ordered arrays

By Caxton C. Foster*

This paper presents an algorithm for sorting a collection of items that are already almost sorted. It is assumed that each item has a sorting key associated with it and that it is desired to arrange the items so that their keys are in ascending order. If duplicate keys are encountered only the item associated with the 'latest' occurrence of the duplicated key is preserved, earlier occurrences being discarded.

The need for such an algorithm is most obvious in the case of on-line record editing in a time-sharing environment where most of the records or sentences will have been entered correctly, but occasionally there will be a replacement of an old record or the insertion of new ones into already entered text.

This algorithm first constructs a table with one entry in the table for each part of the source text that is internally ordered. It then performs an *N*-way merge of the parts.

(First received November 1967)

During the planning stages of UMASS (Unlimited Machine Access from Scattered Sites), a time-sharing system for the CDC-3600 written at the Research Computing Center of the University of Massachusetts, it soon became obvious that one of the most frequently used system routines would be the one which posted the changes to the user's program that occur during the process of debugging. Beginning with a program retrieved from back-up storage or entered over a teletype there would ensue a dialogue between the user and the compiler until an executable version of the program was obtained, and then between the user and the program until the results corresponded to the user's expectations.

Typically each interchange would involve one or more corrections to or insertions into the already present text. Most of the statements or lines of data would already be in proper order. In the UMASS system each line of information (program statement or data) must begin with a sequence number between 1 and $2^{15} - 1$ and the users are encouraged to number their program statements by tens. Insertion is accomplished by interpolation and replacement by entering the old sequence number followed by the new text. Incoming sentences are appended to the already present text as they arrive, and before filing, listing, or compiling a program the routine called 'Sorter' is entered which will post all the changes and insertions before giving control to the appropriate successor routine.

Since Sorter is obviously used with great frequency it is necessary that it be as efficient as possible. Several algorithms were considered and the one presented below seemed to take the maximum advantage of what is known in advance, namely that most of the text will already be in order. Because of the pressure of time and a lack of detailed knowledge about the users' behaviour no attempt was made to run a comparison between this and other alternatives.

The method employed is that of an *N*-way merge. First the text is broken up into chunks that are internally ordered and then these chunks are examined to find the one whose initial record has the smallest sorting key. Since the number of chunks in an almost ordered array will be very much smaller than the number of items in the array this approach offers considerable improvement over any method which searches the original array.

In a conventional *N*-way merge the discovered item would be transmitted to the output area or device, table pointers updated and a new search instituted for the smallest remaining item.

In the CDC-3600 there exists a transmit instruction which copies information from one area in core to another. The overhead for this instruction includes the initialisation of five index registers and therefore it is desirable to transmit as large a record at a time as is possible. Consequently this algorithm is designed to minimise the number of transmits required and is a bit more complicated than a conventional merge.

Construction of the chunk table

A coherent chunk is defined as a group of records (sentences) that lie consecutively in the source area and have the property that the sorting key (sequence number) of the $(n + 1)$th record of the chunk is greater than the sorting key of the nth record for all n. That is: the records within a chunk are already in the desired order. Thus the source area shown in **Table 1** should be divided into chunks at the indicated horizontal lines.

The chunk table has four items of information stored in each entry. These are:

START — the core address of the first word of the chunk
END — the core address plus one of the last word of the chunk
FIRST — the sorting key of the first record
LAST — the sorting key of the last record

* *Research Computing Center, University of Massachusetts, Amherst, Mass.* (Currently temporary Lecturer in Computer Science, *University of Edinburgh*, 8 *Buccleuch Place, Edinburgh.*)

Downloaded from https://academic.oup.com/comjnl/article/11/2/134/378390 by guest on 19 April 2024

Table 1

A typical source area divided into chunks

ADDRESS	KEY	CONTENTS	CHUNK NUMBER
1	10	Ann	I
2	20	Betty	
3	30	Estex	
4	5	Alice	II
5	26	Doris	
6	40	Gwen	
7	30	Esther	III
8	35	Francis	
9	50	Harry	
10	50	Harriet	IV
11	60	Irene	
12	70	June	
13	80	Kathy	

Table 2

The chunk table constructed from Table 1

CHUNK NUMBER	START	END	FIRST	LAST
I	1	4	10	30
II	4	7	5	40
III	7	10	30	50
IV	10	14	50	80

Table 2 shows the chunk table that should be constructed for the source area shown in Table 1.

Fig. 1 displays the flow diagram for constructing such a table.

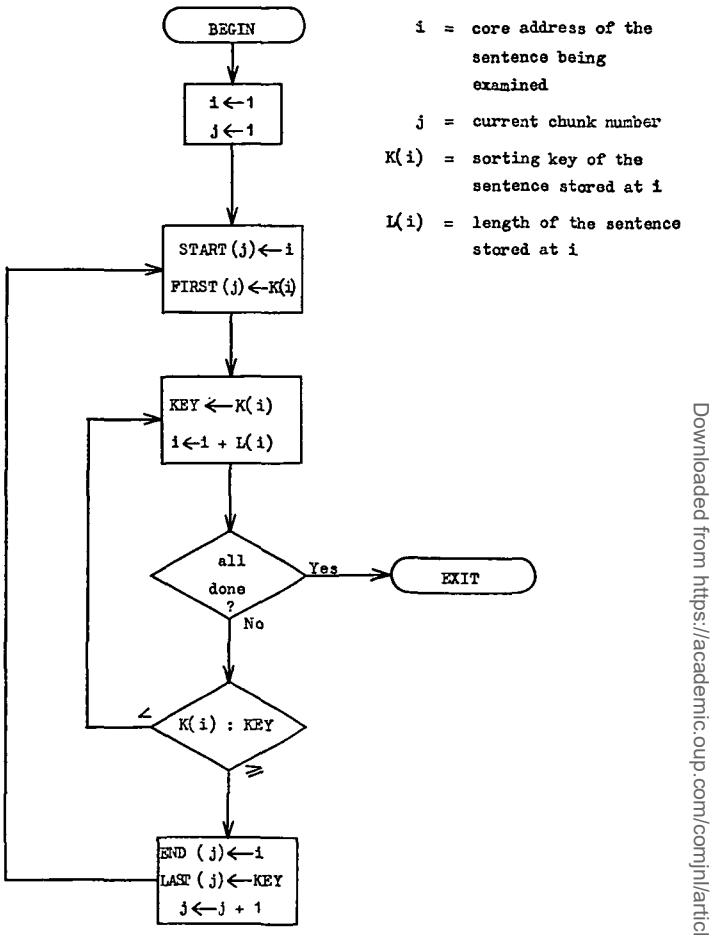

Fig. 1. Construction of the chunk table

The merge

Once the table of coherent chunks has been constructed it is a relatively straightforward matter to merge the N ordered strings of records. **Fig. 2** shows the method used.

To begin with the table is searched for the entry which has the smallest value assigned in the 'FIRST' field. This entry thus points to the chunk whose initial record has a KEY value smaller than any other record. This table entry will be referred to as X. At γ we look for the 'next smallest' key, a table entry Y such that FIRST (Y) > FIRST (X) and for all $Z \neq X \neq Y$:

$$\text{FIRST } (Y) < \text{FIRST } (Z).$$

Should we find a sentence with a key equal to the FIRST (X) we must remove the older (earlier) sentence from its chunk, updating the table entries appropriately. This process is called 'popping up' and will be described below.

It is of course possible that there is no table entry which satisfies the conditions on Y. In order for this to be true, X must be the only remaining entry in the table. Consequently, chunk X can be sent in its entirety to the destination area and the merge is completed.

Assuming that we find a Y we compare the LAST (X) with the FIRST (Y). If the last record in chunk X has a sorting key smaller than the first record of Y then all of chunk X has KEYs less than any other records in the source area, and since the records within a chunk are by definition in proper order, we can send all of chunk X to the destination area at once. Having done this we remove the entry for X from the table, and since Y now points to the smallest key remaining we put Y in for X and go back to γ.

If the LAST (X) is not less than the FIRST (Y) we will be able to send only part of X to the destination area at this time because given this condition we know that Y contains a record which should be placed amid the records of X. To discover how much of X we can

Downloaded from https://academic.oup.com/comjnl/article/11/2/134/378390 by guest on 19 April 2024

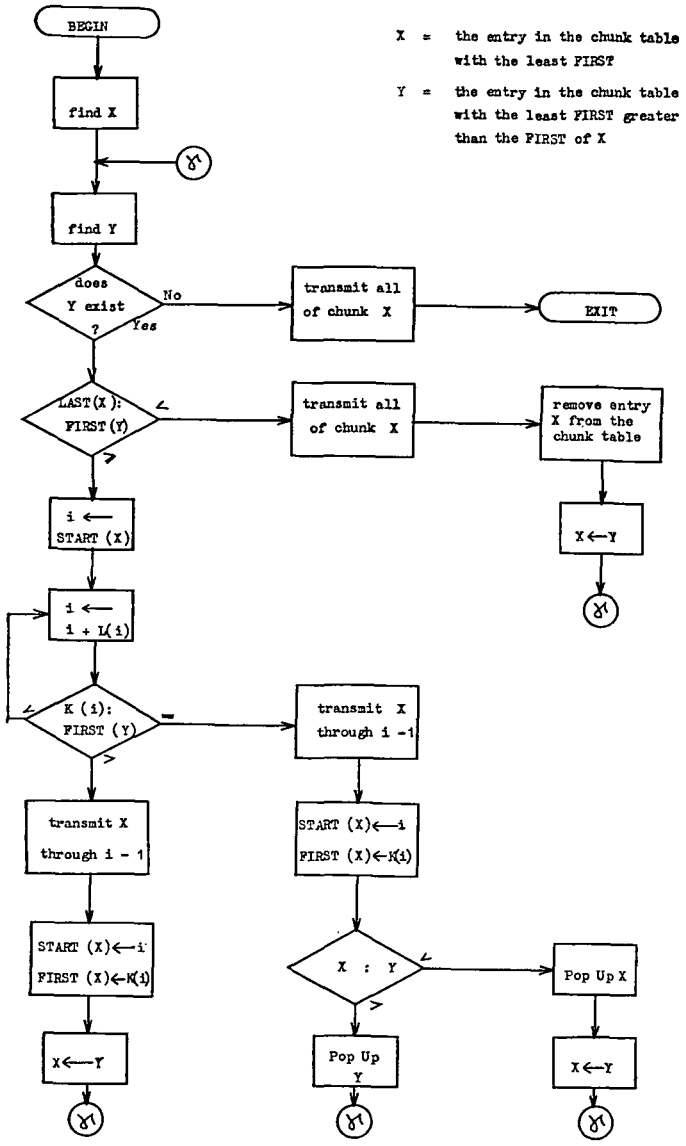

Fig. 2. The merge algorithm

transmit we set a pointer i equal to the START (X), the core location of the first record of chunk X. We increase i by the length of this record and examine the second record of X. We repeat this procedure until i points at a record whose KEY is greater than or equal to the KEY of the first record of chunk Y;

$$\text{KEY}\ (i) \geqslant \text{FIRST}\ (Y).$$

Now we transmit chunk X through location $i - 1$ to the destination area and then update the table entry for X. If the KEY (i) is greater than the FIRST (Y) we put Y in for X and go to γ. If the two are equal we must pop up one chunk or the other. Since the table was constructed from oldest to newest records we can decide which chunk to pop by seeing whether X is earlier than (less than) Y or vice versa. If X is less than Y we pop up chunk X and putting Y in for X go to γ. If the reverse was true we pop up Y. Since the new beginning of chunk Y is now greater than the beginning of chunk X we do not substitute Y for X but go directly to γ.

Popping up

If two records with equal sorting keys are discovered we wish to discard the earlier of the two saving only the latest version of the record. Since this condition can be detected at several points in the algorithm we use a subroutine to accomplish the action. Suppose we wish to pop up chunk Z. We compare the FIRST (Z) with the LAST (Z). If they are equal then the chunk Z consists of only one record (which is to be eliminated) so we merely remove the entry for Z from the table. If they are not equal we must update the table entry to indicate the removal of the first record. Let that record lie at core location P and have length L.

We set

$$q = P + L$$

so that q points to the beginning of the next record and the

$$\text{START}\ (Z) \leftarrow q$$
$$\text{FIRST}\ (Z) \leftarrow \text{KEY}\ (q)$$

Utilisation of core

In order for this algorithm to be at all efficient the entire source text must be present in core. In addition to this space we will need an area to construct the chunk table in. In the UMASS system each entry in the chunk table requires two 48-bit cells. This is because the sorting key is 21 bits long and the addresses are 15 bits in length. Each 'insertion' or 'replacement' will generate at most one chunk so a space of 200 cells, or 100 entries, seems adequate for most purposes. The relatively rare case when there are more than 100 chunks can be handled by sorting the first hundred into one ordered chunk and then combining that one with the next 99 and so forth.

In addition to these two areas we have mentioned a 'destination area'. In UMASS this area is equal in size to the source area and when a chunk, or part of a chunk, is 'transmitted' to the destination area it is actually copied into the next free cells of this area, resulting in one ordered contiguous copy of the text that may then be written out onto the backing store. The CDC-3600 allows the programmer to 'chain' I/O commands so, theoretically, all that would need to be present in the destination area would be a string of I/O control words that would direct the device controller to do successive writes from appropriate disjoint portions of the source area. Unfortunately, the transfer rate of words from core to the drum is so fast that chaining control words causes one to miss the next drum location (while picking up the next control word) and consequently to lose an entire revolution time for each new control word. A slower drum or a more sophisticated drum controller (with control word look-ahead) would thus save the

Downloaded from https://academic.oup.com/comjnl/article/11/2/134/378390 by guest on 19 April 2024

time required to copy each word of text from the source to the destination area and incidentally save most of the core space used for the destination area.

Note that if control word chaining is used (instead of copying) it is imperative to have a large enough chunk table and control word area to hold all the possible items, since the technique of sorting part of the text and then repeating the process will not work in this case.

The sorting key

In UMASS the sorting key for each sentence is stored in a control word at the beginning of the sentence. It consists of a string number (1–63) and a sequence number (0–32767), both in binary. A programmer may have up to 63 separate strings present in his working space. Each string may be a program, a subroutine, a collection of data, or whatever he wishes. This permits him to use the same series of sequence numbers for his sentences in two different strings without confusion arising and without having to worry about whether a subroutine written today can be used in conjunction with one written last week. As long as they have different names there is no problem.

To edit one of the strings in the working space, say one called SMITH, the programmer types NAME SMITH and all sentences entered until the next NAME command is given will have the string number associated with SMITH placed in their control words.

The use of a string number as part of the sorting key requires a few changes to the algorithm as presented above. First, in Fig. 1 the test to see if a new chunk should be started must be changed to read:

> Is the string number of this sentence the same as that stored in the FIRST (j) *and* is the sequence number of this sentence greater than the KEY?

Thus each change of NAME and each insertion or replacement will generate a new chunk.

In UMASS there is a 'string directory' stored at the beginning of the source area which holds the names and numbers of all strings in the source area in the order of their occurrence.

After constructing the chunk table, the first item in this directory is picked up and a pass is made through the merging algorithm (Fig. 2) modified so that the searches for 'least' and 'next least' consider only chunks with the correct string number. Once the first string has been merged the next entry in the directory is obtained and processed, etc.

This scheme has two further advantages. It makes the erasure of a string (removing it from the working space) simple. All that is necessary is to remove the entry from the string directory and on the next 'sort' the string will disappear. Second, if the user desires to rearrange the order in which strings are stored in the work space he need only re-order the entries in the string directory. The next sort will then take care of the re-arrangement.

Conclusions

An algorithm has been presented for sorting and replacing records for a time-sharing environment. It takes advantage of the fact that most items will be in the proper sequence with relatively few insertions or replacements. It further permits the user to have several strings in his work space and to work on whichever one he desires. This algorithm has been implemented on the CDC-3600 for use in UMASS and is considered to be fast enough to permit the source area to be sorted before executing each command, thus simplifying the system control logic. Details of timing and search techniques are so highly machine dependent that they are not presented here.

Downloaded from https://academic.oup.com/comjnl/article/11/2/134/378390 by guest on 19 April 2024