Flash-Sort: Sorting by *in situ* Permutation

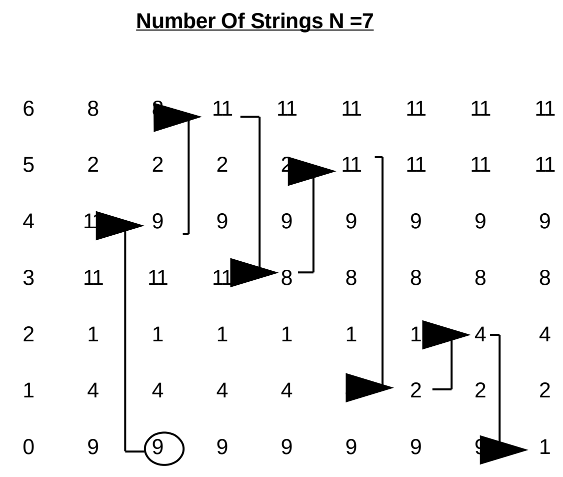
Karl-Dietrich Neubert

Abstract

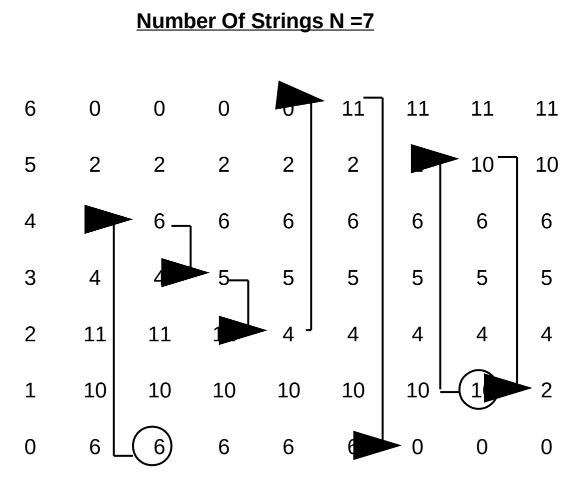
With Flash-Sort we introduce a new sorting algorithm of time complexity O(N).The algorithm is based on *in situ* permutation and requires, in addition to the array holding the elements to be sorted, as an auxiliary memory only a vector of a length equal to the number of different keys. An essential feature of Flash-Sort is the condition for identifying new cycle leaders as given by a word LEADER. This condition is stated here for the first time.
The algorithm and its runtime behaviour will be discussed in detail for the example of a simple byte array. For the general case of arbitrary string and key length, typical results will be presented. Also, some Forth specific aspects will be discussed.

1 Introduction

Great attention has been paid in the past to sorting algorithms based on the comparison of elements.[1,2,3] In theory, these algorithms require $0(N^2)$ time if simple and 0(N logN) time if more complex. In contrast, sorting algorithms based on the classification of elements have found only limited attention.[1,2,3] These algorithms perform ordering in 0(N) time and thus achieve the absolute lowest time complexity for sorting N elements. [4] However, since sorting by classification is believed to require considerable auxiliary memory space, it has not found wide acceptance despite its favorable time behaviour.
Sorting may be viewed to be that permutation which is the inverse to the permutation producing the unsorted array of elements from the sorted one. In the following we understand by permutation this inverse permutation. It is an inherent property of permutations, that, in general, they do not consist of only one cycle (Fig.1), but of many.[5]



Cycle Leader Position	Key - Value	Cycle Length
0	9	7



Cycle Leader Position	Key - Value	Cycle Length
0	6	5
1	10	2

Fig.1 Permutation consisting of

a) one cycle b) two cycles .

Thus, two problems arise with *in situ* permutation:

- finding the position into which a given element has to be placed in the course of permutation cyle.
- finding a genuine new cycle leader after a given cycle has been completed.

As discussed in Sect. 2, the first problem is solved by constructing a vector of class pointers[6,7,8] which point to the current "empty" position of a class. After each movement of an element into its class, the corresponding pointer is updated. Regarding the second problem, Macleod[9] has shown, in some other context, that a genuine new cycle leader may always be identified by the method of cycle tracing which is, however, a rather time consuming method. In a recent paper Pinkus and Schwarz [10] discuss the feasabilty of only *partial sorting* in order to circumvent for certain applications the problem of finding all cycle leaders within an acceptable time span. As we shall show in the next Section, by optimal use of the structural information provided by the elements already sorted , a genuine new cycle leader may be identified by just one test on each candidate for cycle leader and the problem of finding cycle leaders turns out to be actually non existent anymore.

2 Design of Flash-Sort

In this paper we discuss the essence of the algorithm, Fig.2, by assuming the elements to be sorted to be strings of length 1 byt e, stored in an array A(i), i = 0, 1, 2, 3, ..., N -1. We take the view that the array is arranged vertically and that in the ordered state, small numbers reside in the lower and large numbers in the upper part of the array, i.e. the large numbers tend to sift up during ordering.

We introduce a vector L of length M. Because of ist functional role we call this vector the class pointer vector. In CLASSIFY the elements of the array A are counted according to their key for each of the M classes. After completion of the L-VECTOR, each L(k) is equal to the cumulative number of elements A(i) in all the classes 0 through k. The final component L(M-1) is equal to N-1, independent of the distribution of the A(i) into the classes.

Then the words LEADER and PERMUTE are executed in turn until the sorting is completed. In order to facilitate the discussion, we call the position A(i) “empty" if A(i) has not yet been replaced by some other element. At the beginning of the permutation, all positions A(i) are empty, since no element has been moved. If during the permutation an element A(i) has been replaced by some other element we call its position "occupied."

Each cycle starts with a cycle leader. If during the permutation cycle as descibed by the word PERMUTE, the position of an element A(i) becomes occupied by some element FLASH, the corresponding class pointer L(KEY(FLASH)) will be decremented and then will point to the next empty position of the class KEY(FLASH). If the last empty position of that class which provides the cycle leader, becomes occupied, the current permutation cycle is complete.The completion of a cycle is flagged by the fact that the pointer L(KEY(A(j))) of the class providing the cycle leader points to one position below the lowest position of this class. Thus, if A(j) is a cycle leader, the completion of the corresponding cycle is given by the condition

$$L(KEY(A(j))) < j \qquad (1)$$

as shown in Fig.2 with the notation j == JJ .

```
\ ----- Flash-Sort: sorting by in situ Permutation ,  Copyright (c) 1997 Karl-Dietrich Neubert -----------

      VARIABLE   NA        1000000 NA !     NA @ ARRAY A
      VARIABLE   N
      VARIABLE   JJ
      VARIABLE   NMOVE
      VARIABLE   M                256  M !      M @ ARRAY L
      VARIABLE   K

: KEY-VALUE
      ( COLUMN @ + COLLATION-TABLE ) C@ ;

: CLASSIFY
      0 L M @ WSIZE * 0 FILL
      N @ 0 DO
           1 I A KEY-VALUE L+!
      LOOP ;

: L-VECTOR                                              : LEAP
   -1 M @ 0 DO                                           /   JJ @  -1  > IF
        I L DUP >R @ + DUP R> !                         /        -1 K  +!
     LOOP DROP ;                                       /        BEGIN
                                                      /            1 K  +!
: LEADER                                             /              K @ 1+ L @  DUP A KEY-VALUE L @
     LEAP                       <--------------------|              SWAP >=
     BEGIN                                           \          UNTIL
         1 JJ +!                                      \         K @ L @   JJ  !
         JJ @ A KEY-VALUE L @ JJ @ >=                  \   THEN ;
     UNTIL
         JJ @ A KEY-VALUE K ! ;

: PERMUTE
     JJ @ A        DUP @
     SWAP KEY-VALUE
     BEGIN
     K @ L @ JJ @ >= WHILE
         K !
         K @ L @ A DUP KEY-VALUE >R
                                DUP @ >R
         !
         R>
         R>
         -1    K @ L +!
         -1 NMOVE +!
    REPEAT
    DROP DROP ;

: FLASH-SORT ( # of elements ---- )
     N !
     CLASSIFY
     L-VECTOR
     -1           JJ !
     N @ NMOVE !
     M @ 1-       K !
     BEGIN
     NMOVE @ WHILE
         LEADER
         PERMUTE
     REPEAT ;
```

Fig.2 The algorithm Flash-Sort

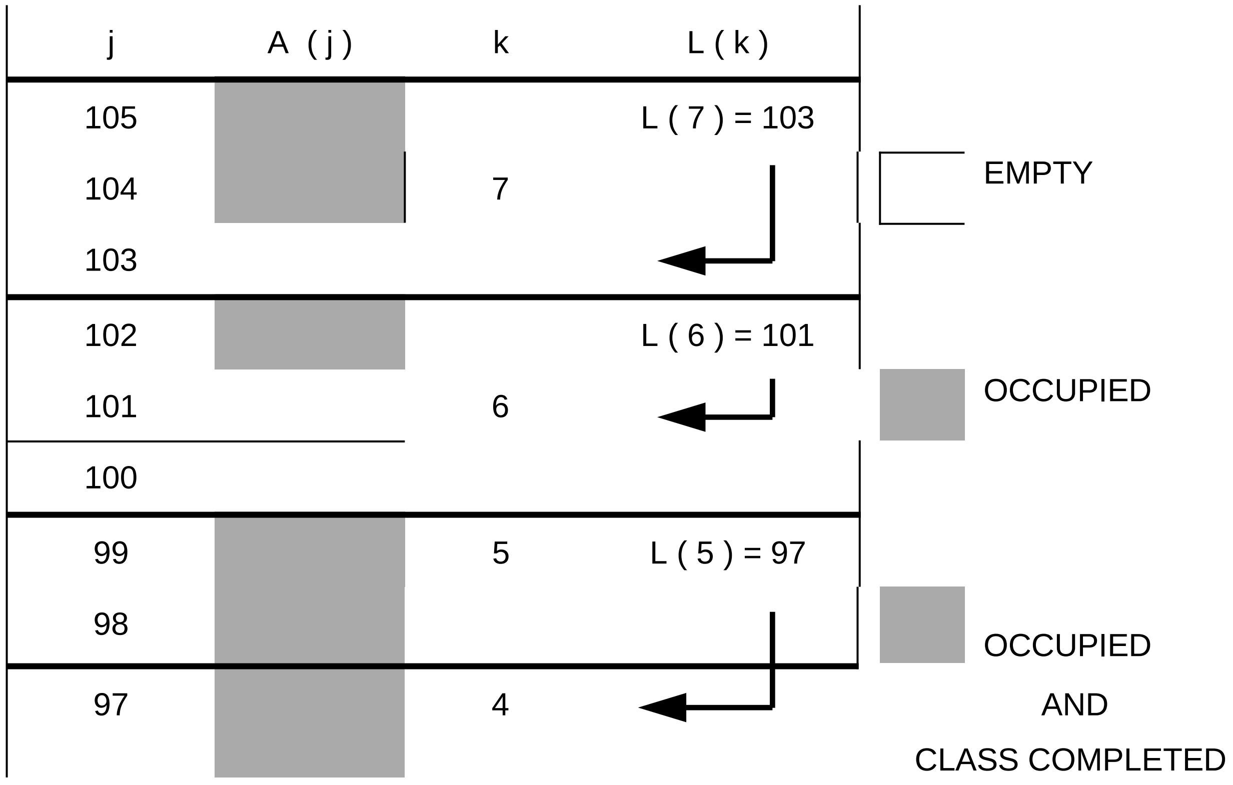


Fig. 3 Various typical possibilities of pointer constellation. j = 0 ... 105 ... N.

Typical situations are illustrated by Fig.3. Up to class k = 5, all elements are sorted. The present cycle leader stems from the position j = 100, class k = 6.

By construction, the pointer L(k) of any class will not be called into action any further once it points to an element not belonging to this class. Whenever the current permutation cycle is completed, a new cycle leader has to be found. A new cycle leader is identified by the rule:

The cycle leader is the element situated in the lowest empty position.

or:

The element A(j) is cycle leader if it is the lowest element satisfying the condition

$$L(KEY(A(j))) >= j \qquad (2)$$

Accordingly, the first cycle leader is the element A(j) = A(0) and subsequent cycle leaders are found by increasing j until an element A(j) is found which satisfies equation (2), as expressed by the word LEADER (Fig.2).

In Fig.3, at the present constellation, the present cycle ends, if the last element of class k = 6 is found in order to occupy position j = 100. If at that instant position j= 103 should still be empty, it would provide the new cycle leader, otherwise it will be an element at some higher position, but at least higher than j = 105.

No search for a new cycle is necessary if the counter for movements, NMOVE, has been decreased from N at the beginning of the process to 0. This is true since we know that the number of movements and replacements necessary to sort N elements by permutation is exactly N. Thus, the search for a new cycle leader ends, if

$$\text{NMOVE} = 0. \qquad (3)$$

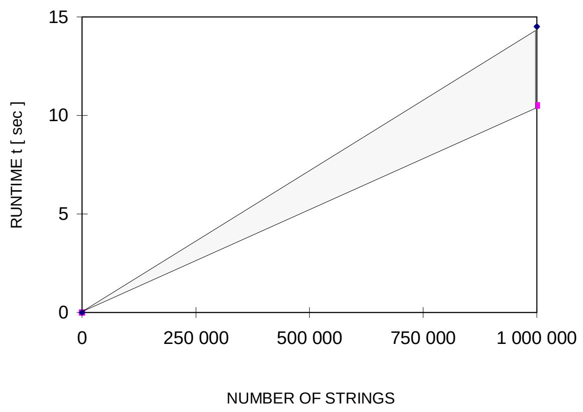
Since the sorting cannot end within a cycle, this condition needs to be checked only between cycles.

The word LEAP in the code, Fig.2, is optional. If this word is ignored, a cycle leader is found by considering up to the last cycle leader every element to be a candidate for cycle leader. Even though suggestive, it is not possible, to test only those elements which are designated by the L-vector, because this vector points to the highest elements of a class and not to the lowest. Fortunately, a hybrid method may be used. The word LEAP first finds, by increasing not j but k, a class just below the class which contains the cycle leader. A member of this class may then be used as starting point for finding a new cycle leader by increasing j. Thus, If $n_{\text{number of cycles}}$ is the number of cycles, the number of steps to find a new cycle leader is reduced in the average from N/2 to $n_{\text{number of cycles}} * N/2M$. With M=256 and the conservative estimate $n_{\text{number of cycles}} = 8$, the reduction factor is equal to about 30.

The reader may have noticed that we do not mention the well-recognized problem connected with elements already in place before the permutation, elements, which cannot be distinguished from those put into place during the permutation. Here this problem does not arise, since every element independent of its class number and its final position is moved exactly once. In the special case that there is exactly one empty position left in a class and the corresponding element is cycle leader, the move degenerate into taking the element out of place and putting it back into the same place. However, since the class pointer and move counter are at still updated correctly, no discrepancies arise from these cycles of length 1.

3 Runtimes

Fig. 4 shows the runtime for sorting N random strings of 1 byte length as a function of N, using a PC with a 166 MHz Pentium Processor and the 80386 UR/FORTH Vers.1.21 of LMI. These runtimes are measured disregarding the word LEAP. The measured runtimes do not ly on a straight line, but within a cone as marked by the shaded area. The reason for this spread becomes evident from Fig.5, where the runtimes for N = 10 ^{6} strings are shown as a function of the position of the last cycle leader. The runtimes exhibit towards the minimal runtime an offset, which is proportional to the position of the last cycle leader. This fact reflects the time needed for finding the last cycle leader. Clearly, the cone in Fig.4 is a consequence of this effect. The number of cycles, on the other hand, is of no influence on the runtime. This number usually is a small number, in this example typically between 5 and 8. Obviously, the smallest runtime occurs, if the last cycle leader is the element A(0), which implies, that there is only one cycle, which is a very rare event, indeed. For the other extreme, that the last cycle leader is nearly the last element of the array - which is also extremely rare - about 30% of the total runtime would be absorbed in finding cycle leaders.



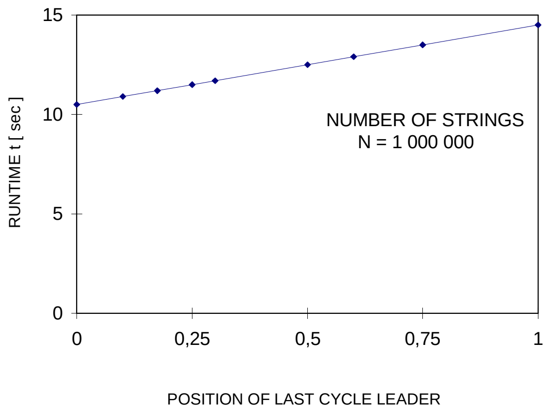


Fig. 4 Runtime for sorting N strings of length 1 byte.

Fig. 5 Runtime as a function of position of the last cycle leader for fixed N. The position is given as a fraction of N.

The average time for finding cycle leaders are roughly halve as large the maximal ones. i.e. only about 15% of the total runtime are required in the average. A cycle of

length 1, which is the case rather frequently for the last cycle, refers to an element already in place and needs not be sorted. Hence, the condition of equat.(3) could be relaxed to NMOVE = 1, and the average time for finding cycle leaders would accordingly be reduced.
With a little more effort, by including the word LEAP, we get a remarcable decrease in the time for finding cycle leaders. Now this time is, in accordance with the estimate given above, barely measurable, and the total time for sorting 1 000 000 bytes is - with the given hardware - 11.92 sec $\pm$ 0.05 sec, i.e. the time for finding cycle leaders is less than 1% of the total run time.

4 Generalization

Sorting on one byte is of limited use for large numbers and was treated here in order to study basic properties. We also have implemented a rather general, recursive version of Flash-Sort. With that version, within the limits of available memory, any number of strings of any length and any number of keys with independent selectable collation sequences for any sort order of columns may be sorted. The overhead due to the more complicated access to the data requires a factor of about 3.5 in runtime, compared to the basic version presented here. This is amply compensated by taking advantage of the Native Code Compiler (NCC) provided by LMI, which results in a speedup by about a factor 5. As an example, sorting 100 000 strings of 50 bytes length with a 50 byte key takes about 4.6 sec, sorting 10^6 strings of the same length and same number of keys requires 46 sec.

5 Discussion

It appears to be a natural phenomenon, that there exist a reciprocal relationship between runtime and memory space. Very often, faster runtime can be achieved by use of additional memory, e. g. in the form of a look up table, or, vice versa, if less memory space is used, the runtime increases. We consider under this aspect the cost in runtime of the *in situ* permutation compared to the algorithm Counting-Sort which is discussed in the book of Cormen, Leiserson and Rivest [11] on standard algorithms under the heading “Sorting in Linear Time“.

Counting-Sort works such, that it sorts by use of a class pointer vector L from a source array A not into A itself, but sequentially into a target array B, i.e. it does not sort in place. The array B will then be copied back into array A. The runtime difference between Flash-Sort and Counting-Sort is evidently the time required to find the cycle leaders in Flash-Sort and it is just this additional time, which is the cost in runtime to save the memory space of array B. For a given hardware configuration, by not needing the memory space of array B, the maximal number of elements which may be sorted with Flash-Sort, is almost doubled. The runtime cost for finding cycle leaders for large N, N >> M, is less than 1% of the total runtime, which is very low, indeed.

(paper presented at euroFORTH'97, Sept:26-28,1997, Oxford,England)

E-mail. karl-dietrich.neubert@usa.net

References

1. D. E. Knuth, The Art of Computer Programming, Vol. 3:
   Sorting and Searching, Addison Wesley Publ. Co., 1973
2. N. Wirth, Algorithm und Datenstrukturen, B. G. Teubner, 1983
3. R. Sedgewick, The Analysis of Quicksort Programs,
   Acta Informatica 7, (1977), 327-355
4. W. Dobosiewicz, Sorting by Distributive Partitioning,
   Information Processing Letters 7, (1978), 1-5
5. D.E. Knuth, Mathematical Analysis of Algorithms in :
   Information Processing 71, North Holland Publ. Co., 1972. Pp. 19-27
6. W. Feurzeig, Algorithm 23: Math Sort,
   Communications of the ACM 3, (1960), 601
7. E. Gamzon, C. -F. Picard, Algorithme de tri par addressage direct,
   C. R. Acad. Sc. Paris 269, (1969), 38-41
8. F. Duccin, Tri par addressage direct,
   R.A.I.R.O. Informatique/Computer Science 13, (1979), 225-259
9. I. D. G. Macleod, An Algorithm for In Situ Permutation,

The Australian Computer Journal 2, (1970), 16-19

10. J.Pinkus, J.Schwarz, Topological Sorting,
Dr.Dobb's Journal 22 (1997), Issue 8, p.113-116

11. T.H. Cormen, C. E. Leiserson and R.L. Rivest
Introduction to Algorithms , p.172 ff
MIT Press, McGraw-Hill Book Company1991