Università di Pisa
Dipartimento di Informatica

Technical Report: TR-03-06

An In-Place Sorting Algorithm Performing $O(n \log n)$ Comparisons and $O(n)$ Data Moves

Gianni Franceschini
francesc@di.unipi.it

March 18, 2003

ADDRESS: via Buonarroti 2, 56127 Pisa, Italy.TEL: +39 050 2212700FAX: +39 050 2212726

An In-Place Sorting Algorithm Performing $O(n \log n)$ Comparisons and $O(n)$ Data Moves

Gianni Franceschini *
francesc@di.unipi.it

March 18, 2003

Abstract

In this paper we give a positive answer to the long-standing problem of finding an in-place sorting algorithm performing $O(n \log n)$ comparisons and $O(n)$ data moves in the worst case. So far, the better in-place sorting algorithm with $O(n)$ moves was proposed by Munro and V. Raman. Their algorithm requires $O(n^{1+\epsilon})$ comparisons in the worst case, for any $\epsilon > 0$. Later, Katajainen and Pasanen discovered the first in-place algorithm with $O(n \log n)$ comparisons and $o(n \log n)$ moves, namely $O(n \log n / \log \log n)$ moves. Our algorithm achieves the same number of comparisons but with $O(n)$ moves, which is asymptotically optimal.

1 Introduction

Sorting is one of the fundamental problems in computation. Knuth [Knu73] gave a long list of motivations, both theoretical and practical, about the importance of this problem. Maybe the simplest of these motivations is also the most meaningful: a sorted sequence of elements drawn from an totally ordered universe is much better searchable than an unsorted one, and thus searching is one of the most natural tasks.

In comparison-based algorithms the only operations allowed in the totally ordered universe are the comparison of two elements and the transfer of a element from a cell to another. In this model, the efficiency of a sorting algorithm (as for almost any other algorithm) can be measured with three metrics: the number of comparisons it requires, the number of element moves it performs and the number of auxiliary memory cells it uses, besides the ones strictly necessary for the input elements.

*Dipartimento di Informatica, Università di Pisa, via Filippo Buonarroti 2, 56127 Pisa, Italy.

In the comparison model, in order to sort and array of n elements, an algorithm must perform at least $n \log n - n \log e$ comparisons in the worst case. For the number of moves the lower bound in the worst case is $\rfloor 3/2n \lfloor$ [MR96]. An *in-place* algorithm uses $O(1)$ auxiliary memory cells.

The well-known Heapsort [Wil64] was the first in-place sorting algorithm performing $O(n \log n)$ comparisons in the worst case. The number of element moves is $O(n \log n)$ and so it is not optimal. On the other hand, the classical selection sort and the algorithm of Munro and Raman [MR92] are in-place and perform $O(n)$ data moves but with a number of comparisons that is, respectively, $O(n^2)$ and $O(n^{1+\epsilon})$. The address-table sort [Knu73] performs an asymptotically optimal number of comparisons and moves but requires $O(n)$ auxiliary cells of memory. Finally, Katajainen and Pasanen [KP99] discovered the first (and only, so far) algorithm requiring $o(n \log n)$ moves in the worst case while guaranteeing in-place and $O(n \log n)$ comparisons. That algorithm performs $O(n \log n / \log \log n)$ data moves in the worst case.

In this paper we give a positive answer to the long-standing open problem of whether there exists an in-place sorting algorithm that performs $O(n \log n)$ comparisons and $O(n)$ data moves. Our algorithm make use of some techniques commonly employed in the study of implicit data structures [Mun86, FGMP02, FG03a, FG03c, FG03b]. A description that do not take care of the constant factors involved in analysis is fairly simple and we believe that this new algorithm can be implemented to be efficient even in practice. This paper contains a preliminary description of the algorithm.

Our algorithm. The core of our algorithm is a data structure that supports an arbitrary sequence of m insertions followed by m deletions in ascending order, with a total cost of $O(m \log m)$ comparisons and $O(m)$ element moves. Searching for an element in such data structure costs $O(\log m)$ comparisons. This data structure needs an auxiliary space of $O(m)$ cells and make use of $O(m/logm)$ pointers $O(\log m)$ bits each. The fundamental characteristic of this data structure is that the above bounds are achieved even when reading and updating a pointer requires $O(\log m)$ comparisons and data movements instead of the usual $O(1)$ cost. Since we want an in-place algorithm, we need to encode all the pointers of the structure. The above characteristic allows us to use a simple permutation of two distinct keys for encode a single bit (technique commonly used in implicit data structures, see [Mun86, FGMP02, FG03a, FG03c, FG03b]. In Section 2 we show how to reduce the problem of sorting an array of n elements using $O(1)$ auxiliary cells and performing $O(n \log n)$ comparisons and $O(n)$ element moves to the problem of sorting a subarray of m element using $\geq \alpha m$ auxiliary data cells (α integer constant) and performing $O(m \log m)$ comparisons and $O(m)$ element moves using our data structure. In section 3 we give a description of this data structure.

2 Reduction of the problem

Let $A[0 \dots n-1]$ be the array of n cells containing the n elements to be sorted. We assume that there are not repeated elements, so we can use the pointer and integer encoding to which we refer above directly with the elements of the subarray subject to the sorting process. The general case of repeated elements can be solved as in [MR92] and we will return on this topic at the end of the section.

As in [KP99], the algorithm has two phases: the sorting phase and the merging phase. Also the logical subdivision of array A and the merging phase are the same as in [KP99]. Let 2^w be the largest power of 2 smaller than n. The array A is subdivided in $w+2$ subarrays: we set $A_0 = A[0,0]$ and, for $i = 1 \dots w$, $A_i = A[2^{i-1} \dots 2^i-1]$. Hence the size of A_i is 2^{i-1} if $i \leq w$ and $n-2^w$ if $i = w+1$. In the sorting phase all those subarrays are sorted independently. In the merging phase a standard 2-way in-place merging algorithm is used to sort the whole array A. Given the size constraints on the subarrays A_i, the worst case number of comparisons and moves of the merging phase is $O(\sum_{i=1}^{w} 2^i + \sum_{i=0}^{w+1} 2^{i-1}) = O(n)$.

In order to sort each subarray from A_w to A_0, we can use, for a generic A_i, an auxiliary subarray B_i that is the union of all subarrays A_j preceding A_i (i.e. for $j = 0 \dots i-1$). These subarrays are yet unsorted and so their elements, which we call *ignored elements*, can be swapped with *sorting elements* (the ones of A_i) during the sorting process of A_i. Thus we can simulate a real empty auxiliary array increasing the complexity of each iteration only by a constant factor. Naturally we have to be able to distinguish the ignored elements from the sorting elements as they can be totally reinserted into B_i for the successive iteration.

Hence can reduce the problem to that of sorting an array A' of n' element using an auxiliary array B of size n' that contains ignored elements of which we do not care and which will be swapped with sorting elements during the process. Array A' is subdivided in a constant number α of subarrays of size $m = \lceil n'/\alpha \rceil$ (except for the last one). Therefore the problem is finally reduced to the sorting (in $O(m \log m)$ comparisons and arithmetic operations and $O(m)$ moves) of an array C containing m sorting elements, using an auxiliary array B of size $m' \geq \alpha m$ (full of ignored elements). This last step will be performed using a data structure that supports an arbitrary sequence of m insertions followed by m deletions in ascending order, with a total cost of $O(m \log m)$ comparisons and $O(m)$ element moves. Searching for an element in such data structure costs $O(\log m)$ comparisons. Moreover it uses $O(m)$ auxiliary cells of memory (depending on this we chose the constant α) and does not store explicitly any structural information like pointers or indices.

3 The data structure

The data structure that we use is not fully implicit as it uses more space than that strictly needed by the keys. It uses some simple coding techniques and takes advantage of the $m' > m$ locations of B in order to make few moves. We encode data by the classical pairwise (odd-even) permutation of keys [Mun86, FGMP02, FG03a, FG03c, FG03b]. To encode a pointer or an integer of b bits by using $2b$ distinct keys $x_1, y_1, x_2, y_2, \ldots, x_b, y_b$, we permute them in pairs x_i, y_i by the following rule: if the ith bit is 0, then $\min\{x_i, y_i\}$ precedes $\max\{x_i, y_i\}$; else, the bit is 1 and $\max\{x_i, y_i\}$ precedes $\min\{x_i, y_i\}$.

We also use chunks. A *chunk* is a set of $k = \Theta(\log m')$ elements stored in contiguous locations. The elements in any chunk belong to a certain interval of values, and the chunks are pairwise disjoint when considered as intervals. This allows us to define a total order on the set of chunks, so that we write $c_1 < c_2$ for any two chunks meaning that the elements in chunk c_1 are all smaller than those in chunk c_2.

In general, the data structure groups all the elements (of C) in chunks with some differences for the leaves, as we shall see. The elements of a chunk are not maintained in order. The parameter k is set to be equal to $\beta \log m'$ for a proper constant depending on the set of structural information used in the data structure (see [Mun86, FGMP02, FG03a, FG03c, FG03b]).

The data structure is subdivided in two layers. The top layer is a new variant of a well-known (see [IKR81, Wil82, BDFC00]) data structure used to maintain a set of element in sorted order in a contiguous zone of memory. The bottom layer is populated by balanced trees of constant height.

Top layer. The top layer T is stored in a contiguous zone of memory. T is logically subdivided in *segments* of $s = k\gamma \log m'$ elements (for γ constant). Since all segments have the same length, T can be seen as an array of segments, so with $T[i]$ we mean the i-th segment of T and with $|T|$ the number of segments of T. The same notation could be used when T is seen like an array of elements. What $T[i]$ actually means would be clear from the context.

Like in [BDFC00], we assume for simplicity that the number of segment of T is a power of 2. On T it is implicitly defined a binary tree of height $h = O(\log m') = O(\log m)$. The root of the tree is the entire array T. The left child and the right child of the root are, respectively, the subarray $T[1, \ldots, |T|/2]$ and the subarray $T[|T|/2 + 1, \ldots, |T|]$ and so forth to the leaves that are the single segments of T.

The objects that are contained in segments are the chunks (so a segment can contain at most $\gamma \log m'$ chunks). During the lifetime of the data structure, chunks coming from the bottom layer will be inserted in segments. The set of chunk of a segment is maintained in sorted order (here the "order" is that defined on the chunks; the elements within a chunks are unsorted) by a brute-force shift with a worst-case cost of $O(\log^2 m')$. Hence, if we see a segment s as an array of $\gamma \log m'$ slots containing a variable number $c \leq \gamma \log m'$ of chunks, then the chunks of s occupy the first c slots of it. The number of chunks of a segment is encoded in the first chunk and, as we will see, during the sequence of insertions in the whole data structure we maintain the invariant that each segment has at least one chunk.

With this organization of segments, a search for a element in the top layer can be performed in three steps.

1. A binary search is performed on the set of the first elements of each segment (all this elements are in fixed positions). Since all chunks are disjoint sets of elements, this step can determine at most two segments in which the search can be continued (recall that the elements of a chunk are unsorted).

2. The correct segment is determined examining the elements of the first chunk of one of the two segments. Then a binary search is performed on the set of the first elements of the chunks contained in the segment. For the same reasons above, this step can determine at most two chunks in which the search can be continued.

3. The correct chunk is determined and the search ends with a linear scan of its elements.

So, the search for an element in the top layer can be performed in $O(\log m)$ comparisons. However, there is another possible organization for the segments that is similar to that for the internal nodes of the trees in the bottom layer.

Naturally, during the sequence of insertions in the whole data structure the top layer have to be maintained balanced in order to pay a small amortized cost for insertions of chunks. Intuitively, the segments must not be too full. Moreover, we have to maintain the invariant that any segment must contain at least a chunk. This goal is fairly standard in this type of data structures. The definitions of density, threshold and node overflow are a variation of these in [IKR81, Wil82, BDFC00]. The main difference is that our basic entries are the chunks instead of the individual elements as in [IKR81, Wil82, BDFC00].

A node v has associated a density $d(v)$ defined as the ratio between the number of chunks contained in its subarray and the size of this. Let τ_0 be a constant less than 1 and let δ be $(1-\tau_0)/h$. We define the density threshold for the level l of the tree (the root is at level 0) as $\tau_l = \tau_0 + l\delta$. A leaf overflows if its corresponding segment is full of chunks. An internal node v at level l overflows if $d(v) > \tau_l$ and at least one of its children overflows. When a chunk have to be inserted in a segment that is already full, we climb up in the path of the tree until we find the lowest non-overflowing ancestor v of the leaf associated to the segment. At this point we redistribute the chunks in v evenly among the array associated with v so that, for each segment s of v, s contains at least one and at most $\tau_l \gamma \log m'$ chunks, where l is the level of v.

The analysis for the total cost of a sequence of t insertions of chunks is perfectly analogous to those in [IKR81, Wil82, BDFC00]. The only difference is that we manage object of $k = O(\log m)$ elements instead of single elements. The cost is $O(t \log^d t)$ comparisons and moves, where d is a small constant (3 or 4) depending on how some marginal technicalities are addressed. For the purpose of that paper the precise evaluation of d is unimportant since, as we will see, we are able to manage trees of any constant height in the bottom level, performing $O(\log m)$ amortized comparisons and $O(1)$ amortized moves while inserting an element and $O(\log m)$ comparisons while searching for an element.

Bottom layer Each chunk c of the top layer has associated a small search tree of height d. The pointer to the tree is encoded in c. All the elements of the tree of c are greater than or equal to those of c and less than or equal to those of the successor of c among the chunks of the top layer.

During the sequence of m consecutive insertions in the whole structure the nodes of any small tree will respect an invariant on the number of sorting elements they contain. Each internal node of a tree can contain from $\log m'$ to $2 \log m'$ chunks. Each leaf can contain from k to $2k + k - 1$ elements.

An internal node has a child associated with each chunk (the pointer to this child is encoded in the chunk). A tree is maintained balanced with variations of the split operation defined for the nodes of the classical B-trees. When a leaf is full, that is it has $2k + k - 1$ sorting elements plus the one that have to be inserted, it is split in two leaves of k elements each and the median chunk is inserted in the parent. From a high-level point of view the cases for internal nodes are analogous. We only say that when a root of a tree splits, the median chunk is inserted into the top layer and a rebalancing can be triggered.

The important point is the internal organization of the nodes and the allocation of memory. Each node (leaf or internal) is stored in a contiguous zone of memory of the maximum size the node can have ($2k \log m'$ for internal nodes, $2k + k - 1$ for leaves).

The internal organization of the leaves is simple: a new element is inserted at the end. The number of elements in a leaf is encoded in the first k elements of it (these elements exist because of the invariant on the size of a leaf). The updates of this counter can be performed in $O(1)$ amortized moves, since the sequences of operation on the data structure have all the insertions before the deletions.

Hence the elements of a leaf are unsorted. The splitting of a leaf can be performed with a series of implicit selections (see [LW88]) and distributions. For these reasons, inserting a element in a leaf costs, in the worst case, $O(1)$ moves when it is not full and $O(\log m')$ moves when it has to be split.

The internal organization of internal nodes is more involved, since we have to pay $O(\log m')$ moves in the worst case for the insertion of a chunk and $O(\log^2 m')$ moves in the worst case for splitting in order to obtain the $O(1)$ amortized moves. The space allocated for an internal node is of $2k \log m'$ locations and these location are all occupied by elements of C or B. The first $4 \log m'$ locations are reserved for a *directory* that will facilitate all operations (search included). The other $2 \log m'$ *groups* of $k-2$ locations each, will possibly contain the chunks without the smallest and largest elements (that are stored in the directory). All the groups that contain ignored elements are "free".

The situation of the allocation of groups is represented in the directory: the i-th couple of elements in the directory is in increasing order if and only if the i-th group is occupied by a chunk. Hence, searching in a node costs $O(\log m')$ comparisons. Inserting a chunk in a non-maximal node costs $O(\log m')$ comparisons and moves worst case. Splitting a node costs $O(\log^2 m')$ comparisons and moves in the worst case (later we will deal with the memory allocation/deallocation of nodes).

For all the above invariants, searching for a element in a tree costs $O(\log m')$ comparisons while inserting a element in a tree costs $O(\log m')$ amortized comparisons and $O(1)$ amortized moves.

Memory layout The structure is stored in three zones in an auxiliary array B. At the beginning, all the positions in B are occupied by ignored elements. These elements have to be moved during the sorting process of the elements of C.

The first zone contains the array T of the top layer. The second zone is divided into allocation units of size $2k \log m'$ each, and contains all the internal nodes of trees in the bottom layer. The third zone has allocation units of $2 \log m'$, and contains all the leaves of trees. The free allocation units of the second and third zone are maintained in two linked lists. The structural information of these lists can be easily encoded in the $O(\log m)$ ignored elements of each free allocation unit (a bit can be encoded in any pair of disjoint elements, so also the ignored elements can be used for this purpose).

The structure has to contain a bounded number of elements and we keep the constraint in size of all the parts of it (segment, nodes, leaves). Hence, the maximal sizes of the three zones can be calculated statically. However, even a dynamic management of the size of zones could be amortized. Let us discuss an example of that dynamic management. If the second zone needs a new allocation unit in order to store a new internal node, the first k allocation units of the third zone can be moved at the end of that zone (where there must be space, given the choice of α). That operation requires

- $O(2k \log m')$ moves in order to transfer the elements;
- k searches in order to find the parent of the leaves contained in the moved allocation units;
- $O(k \log m')$ moves in order to update the encoded pointers.

These operations are required when a split of an internal node has to be performed. If an internal node needs to be split then there exist $\Omega(k \log m')$ past insertions in it that can be uniquely associated with this split operation and that can be charged with $O(1)$ moves each, from the $O(k \log m')$ moves total.

Using the whole structure. Sorting the subarray C using the subarray B as auxiliary memory is accomplished in two phases. In the first phase we build up the data structure inserting one sorting element at a time starting from $C[1]$.

When a small tree needs to be split then there exist $\Omega(\log^d m)$ past insertions in it that can be uniquely associated with this event. Let the whole number of this events (i.e. the number of insertions of chunks in the top layer) be t. Then there are $\Omega(t \log^d m)$ distinct operations that can be charged with $O(1)$ comparisons and moves each, from the total $O(t \log^d t) = O(t \log^d m)$ cost of the t insertions of chunks in top layer. ¿From the amortized costs of insertions in a small tree it follows that the total cost of the whole sequence of m insertions of sorting elements is $O(m \log m)$ comparisons and $O(m)$ moves in the worst case.

At the end of the first phase all the sorting elements originally in C are in the structure stored in B and C is full of ignored elements. In the second phase the m sorting keys are extracted in sorted order from the structure in B and are reinserted in C. The sequence of m extractions is doable with a simple in-depth visit of the whole data structure. Since the whole structure has a constant number of levels, during the visit we have to store only a constant number of pointers to remember the path currently visited.

Encoding structural information in the case of repeated elements When multiple elements appear in A, we adopt the solution in [MR92]. The important aspect is that in order to operate in $O(\log m')$ time on an encoded pointer or counter, there must be a correspondence between the position of each parts of the structure and the element in the *encoding subarray* created with the method in [MR92]. That is, the allocation of the elements in the encoding array have to follow that of the space in B. For example, if an internal node begins at i-th location of B then the encoding of the situation of the allocation of its groups have to begins at i-th location of the encoding subarray. Moreover, if a chunk of that node begins at location $i + j$ of B then the encoding of the child associated with it have to begins at location $i + j$ of the encoding subarray. That correspondence is easy to maintain once we know the starting positions of the two subarrays.

References

[BDFC00] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In IEEE, editor, *41st Annual Symposium on Foundations of Computer Science: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 399–409, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. IEEE Computer Society Press.

[FG03a] Gianni Franceschini and Roberto Grossi. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log \log n)$. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 670–678. SIAM, 2003.

[FG03b] Gianni Franceschini and Roberto Grossi. Optimal cache-oblivious implicit dictionaries. In *30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, 2003.

[FG03c] Gianni Franceschini and Roberto Grossi. Optimal space-time dictionaries over an unbounded universe with flat implicit trees. Technical report TR-03-03, January 30, 2003.

[FGMP02] Gianni Franceschini, Roberto Grossi, J. Ian Munro, and Linda Pagli. Implicit B-trees: New results for the dictionary problem. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002.

[IKR81] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming, 8th Colloquium*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Acre (Akko), Israel, 13–17 July 1981. Springer-Verlag.

[Knu73] D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching.* Addison–Wesley, Reading, Massachusetts, 1973.

[KP99] Jyrki Katajainen and Tomi A. Pasanen. In-place sorting with fewer moves. *Information Processing Letters*, 70(1):31–37, April 1999.

[LW88] Tony W. Lai and Derick Wood. Implicit selection. In R. Karlsson and A. Lingas, editors, *SWAT 88: 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 14–23, Halmstad, Sweden, 5–8 July 1988. Springer-Verlag.

[MR92] J. Ian Munro and Venkatesh Raman. Sorting with minimum data movement. *Journal of Algorithms*, 13(3):374–393, September 1992.

[MR96] J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theoretical Computer Science*, 165(2):311–323, 10 October 1996.

[Mun86] J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[Wil82] Dan E. Willard. Maintaining dense sequential files in a dynamic environment. In *ACM Symposium on Theory of Computing (STOC '82)*, pages 114–121, Baltimore, USA, May 1982. ACM Press.