Sorting Multisets Stably in Minimum Space[⋆]

Jyrki Katajainen[1,2] and Tomi Pasanen[2]

[1] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark

[2] Department of Computer Science, University of Turku, Lemminkäisenkatu 14 A, SF-20520 Turku, Finland

Abstract. In a decision tree model, $\Omega(n \log_2 n - \sum_{i=1}^{m} n_i \log_2 n_i + n)$ is known to be a lower bound for sorting a multiset of size n containing m distinct elements, where the ith distinct element appears n_i times. We present a *minimum space* algorithm that sorts *stably* a multiset in asymptotically *optimal worst-case time.* A Quicksort type approach is used, where at each recursive step the median is chosen as the partitioning element. To obtain a stable minimum space implemention, we develop linear-time in-place algorithms for the following problems, which have interest of their own:

Stable unpartitioning: Assume that an n-element array A is stably partitioned into two subarrays A_0 and A_1. The problem is to recover A from its constituents A_0 and A_1. The information available is the partitioning element used and a bit array of size n indicating whether an element of A_0 or A_1 was originally in the corresponding position of A.

Stable selection: The task is to find the kth smallest element in a multiset of n elements such that the relative order of identical elements is retained.

1 Introduction

The sorting problem is known to be easier for multisets, containing identical elements, than for sets, in which all elements are distinct. The complexity of an input instance depends on the multiplicities of the elements. When only three-way comparisons are allowed, $\Omega(n \log_2 n - \sum_{i=1}^{m} n_i \log_2 n_i + n)$ is known to be a lower bound for sorting a multiset with multiplicities $n_1, n_2, \ldots, n_m$ (where $n = \sum_{i=1}^{m} n_i$) [19]. Mergesort and Heapsort can be adapted to sort multisets in $O(n \log_2 n - \sum_{i=1}^{m} n_i \log_2 n_i + n)$ time [21] (without knowing the multiplicities beforehand). An optimal in-place implementation based on Heapsort also exists [19], but due to the nature of Heapsort this algorithm is not *stable*, i.e., the relative order of identical elements is not necessarily retained. The main concern of this paper is, how to ensure time-optimality, space-optimality, and stability at the same time, i.e., the problem left open by Munro and Raman [19].

In our accompaning paper [11] we proved that randomized Quicksort can be adapted to sort a multiset stably and in-place such that the running time

[⋆] Presented at the *3rd Scandinavian Workshop on Algorithm Theory*, Helsinki, July 1992.

will be optimal up to a constant factor with high probability. In the present paper we improve this result by showing that multisets can be sorted in optimal time also in the worst case. To adapt Quicksort for sorting multisets, one should perform a three-way partition at each recursive step [24]. For this purpose, we use the linear-time, in-place algorithm for stable partitioning presented in [11]. The standard way to make Quicksort worst-case optimal is to use the median as the partitioning element.

The basic problem encountered is how to select the kth smallest element in a multiset of n elements such that the relative order of elements with equal values is the same before and after the computation. This is called the *stable selection problem*. Actually, we shall also study the following variant of the selection problem, called the *restoring selection problem*: find the kth smallest of n elements such that after the computation the elements are in their original order. The latter problem has applications in other areas, e.g., in adaptive sorting (cf. [17]). An in-place solution for both of these problems is immediately obtained, if we scan through the elements and calculate for each the number of smaller elements. This will, however, require $O(n^2)$ time. On the other hand, if we allow $O(n)$ extra space, the linear-time selection algorithm [1] (or its in-place variant, see [14]) can be used to solve the problems simply by coupling with each element its original position. After the selection, the elements are easily permuted to their original positions (cf. [12, Section 5.2, Exercise 10]).

To solve the restoring selection problem, we implement the prune-and-search algorithm of Blum et al. [1] more carefully. The algorithm is based on repeated partitioning. Therefore the fast, in-place algorithm for stable partitioning is used here. In order to reverse the computation we need a space-efficient solution for the *unpartitioning problem* defined as follows. Assume that an array A of size n undergoes a stable partition. Let the resulting subarrays be A_0 and A_1 with respective sizes n_0 and n_1. The problem is to recover A from its constituents A_0 and A_1. The information available is the partitioning element used and a bit array containing n_0 zeros and n_1 ones. The interpretation of the ith b-bit in position j is that the ith element of A_b is the jth element of A ($b \in \{0, 1\}$). In Section 3 we introduce an algorithm for stable unpartitioning that runs in linear time and requires only a constant amount of additional space.

In Section 4 we show how the restoring selection problem is solved in linear time using $O(n)$ extra bits. By means of this, we are able to develop an algorithm for stable selection that requires linear time and only $O(1)$ extra space. This algorithm presented in Section 5 is then used in the final sorting algorithm which we describe and analyse in Section 6.

Before proceeding we define precisely what we mean by a *minimum space* or *in-place algorithm*. In addition to the array containing the n elements of a multiset, we allow one storage location for storing an array element. This is needed, for example, when swapping two data elements. The elements are regarded to be atomic. They can only be moved and compared with the operations $\{<, =, >\}$ in constant time. Moreover, we assume that a constant number of extra storage locations, each capable for storing a word of $O(\log_2 n)$ bits, is available and that operations $\{<, =, >, +, -, \text{shift}\}$ take constant time for these words. An *unrestricted shift operation* takes two integer operands v and i and produces $\lfloor v \cdot 2^i \rfloor$.

2 Tools for building minimum space algorithms

In this section we briefly review the basic techniques for minimum space algorithms.

Blocking: The input array is divided into equal sized blocks. Often blocking with blocks of size $\sqrt{n}$ or $\log_2 n$ works well. (This requires that good estimates for the numbers $\sqrt{n}$ and $\log_2 n$ are available but these are easily computed from n in $O(n)$ time.) Most efficient in-place algorithms in the literature are based on the blocking technique (see, e.g., [7–9, 20, 22, 23]).

Internal buffering: Usually some blocks are employed as an *internal buffer* to aid in rearranging or manipulating the other blocks in constant extra space. This idea dates back to Kronrod [13] (see also [22]) and is frequently used in minimum space algorithms. If the goal is a stable algorithm, the internal buffer should be manipulated carefully, since otherwise the stability might be lost.

Block interchanging: A block X can be reversed in-place in linear time by swapping the pair of end elements, then the pair next to the ends, etc. Let X^R be X reversed. The order of two consecutive blocks (not necessarily of the same size) X and Y may be interchanged by performing three block reversals, namely $YX = (X^R Y^R)^R$. This idea seems to be part of computer folklore.

Bit stealing: Let x and y be two elements, which are known to be distinct. Depending on the order, in which the elements are stored in the array, extra information is obtained. The order xy, $x < y$, may denote a 0-bit and the order yx a 1-bit. This technique has been used for example by Munro [18] in his implicit dictionary. With $\lceil \log_2(n+1) \rceil$ stolen bits it is possible to implement a counter taking values from the interval $[0..n]$, but the manipulation of this counter will take $O(\log_2 n)$ time.

Packing small integers: Let us assume that we have t small integers each represented by m bits. That is, the integers are from the domain $[0..2^m - 1]$. Further, assuming that $t \cdot m \leq \log_2 n$, the integers can be packed into one word w of $\lceil \log_2 n \rceil$ bits. Let us number the bits of w from right to left such that the righmost (least significant) bit has number 0 and the leftmost bit has number $\lceil \log_2 n \rceil - 1$. Now the integer i_j $(j = 1, 2, \ldots, t)$ is stored by using the bits $(j-1)m, \ldots, jm - 1$ of w. Each integer is easily recovered from w in constant time if multiplications and divisions by a power of 2 are constant time operations. The value v of i_j is obtained as follows:

$$v = \{w - [(w \text{ shift } -jm) \text{ shift } jm]\} \text{ shift } -(j-1)m.$$

(Observe that in our algorithms m can be chosen to be a power of 2, so we do not need general multiplication.) With a code similar to this the value of i_j can be updated. Previously the packing technique has been used for example in [2, 15].

In some in-place algorithms also the modification of the input data is allowed (see, e.g., [4, 6] or [23, Theorem 3.2]). However, we consider this as an illegal trick.

3 Stable minimum space unpartitioning

The heart of our selection and sorting algorithms will be the linear-time, minimum space algorithm for stable partitioning given in [11]. Another important subroutine is a fast, minimum space algorithm for stable unpartitioning which is the topic of this section. We show that the computation of the partitioning algorithm is reversible, even if the steps executed are not recorded.

In an abstract setting the *stable partitioning problem* can be defined as follows: Given an n-element array A and a function f mapping each element to the set $\{0, 1\}$, the task is to rearrange the elements such that all elements, whose f-value is zero, come before elements, whose f-value is one. Moreover, the relative order of elements with equal f-values should be retained. Let the resulting subarrays be A_0 and A_1. For the sake of simplicity, we call the elements of A_0 *zeros* and elements of A_1 *ones.* The *stable unpartitioning problem* is to recover A from its constituents A_0 and A_1. The information available is the f-function and a *placement array*, a bit array of size n indicating whether an element of A_0 or A_1 was originally in the corresponding position of A. Observe that in our formulation of the problem it is essential that f is known during unpartitioning.

Stable merging can be seen as a special case of stable unpartitioning, since the placement array is easily created by scanning the input of a merging problem with two cursors. By unpartitioning the original merging problem is solved. This indicates that it might be possible to generalize the algorithms for stable merging to solve the stable unpartitioning problem. Generally, this cannot be done since most algorithms for stable merging utilize the fact that the A_b-elements appear in sorted order. In unpartitioning this is not necessarily the case. However, there are great similarities between our unpartitioning algorithm and parallel merging algorithms given in [10].

The stable unpartitioning is easily done in linear time when $O(n)$ extra space is available. Algorithm A to be described next does this by scanning the placement array and storing the site together with each A_b-element. During the scan two cursors C_0 and C_1 are maintained, the former pointing to A_0 and the latter to A_1. Initially, C_b will point to the first element of A_b. If the jth position contain the bit b, then the C_bth element of A_b is coupled with its *site* j and the counter C_b is advanced. After computing the sites, the elements are permuted to their final positions without using excess memory space (this permutation problem was ranked as a one-hour exercise by Knuth [12, Section 5.2, Exercise 10]). Hence we have

Lemma 1. *Algorithm A solves a stable unpartitioning problem of size* n *in* $O(n)$ *time with* $n + O(1)$ *counters, each requiring at most* $\lceil \log_2(n+1) \rceil$ *bits.*

For the time being let us assume that n, the number of elements is a power of 2. Lateron we show how to get rid of this assumption. In our improved algorithms we divide the input into blocks by using a *blocking factor* $\lg n$ or $2^{(\lg n)/2}$ ($\approx \sqrt{n}$), where $\lg n$ denotes the smallest power of 2 greater than or equal to $\log_2 n$. Since n is assumed to be a power of 2, it is divisible by both of these blocking factors.

Before proceeding, we will introduce some terminology. Let us call a block containing only zeros as a *0-block* and a block containing only ones as a *1-block*. If a block is a 0-block or 1-block it is called a *0/1-block*. Further, let *0&1-block* denote a block consisting of two sequences, a sequence of zeros followed by a sequence of ones. The basic idea of our algorithms is simply to transform the original problem to n/t similar subproblems of size t. That is, by using the terminology introduced above, the goal is to transform one 0&1-block of size n to n/t 0&1-blocks of size t such that in each subblock the number of zeros and ones is equal to that of 0- and 1-bits in the corresponding part of the placement array. Hence, after this transformation the subproblems can be solved locally.

When the input array is divided into the blocks of size t, one complication is that one of the blocks might be a 0&1-block while the others are 0/1-blocks. The single 0&1-block is handled as follows. We first interchange the zeros of the block to the end of the input array and then move them gradually into the blocks they belong. This can be done by repeated block interchanges. Each zero of the 0&1-block takes part in at most n/t interchanges, whereas each one of the input takes part in at most two block interchanges. Therefore the total work done here is $O(n)$. The blocks at the end of the array are called *finished* if they got all their zeros. The last *unfinished*, or *half-finished*, 1-block may have obtained only some of the zeros that should be there. The half-finished block is however seen as a 1-block, though the zeros at the end are kept untouched.

Let the *leader* of a 0-block be its *first* element and the *leader* of a 1-block its *last* element. Now the basic steps of the transformation, called *one-to-many transformation*, are the following (see also Fig. 1):

1. Divide the input array A into blocks of size t.
2. If there exists a 0&1-block then move its zeros to their own blocks.
3. Merge the unfinished blocks such that the sites of their leaders are in sorted order. This way the elements will come closer to their final positions.
4. Transform the unfinished 0/1-blocks to 0&1-blocks such that each element is placed in its own block.
5. Move the zeros (if any) at the end of the half-finished block over the ones in the block.

To perform Step 1 only the value t has to be computed but this is easily done in linear time. Step 2 requires linear time as well (cf. the discussion above). Step 5 requires only $O(t)$ time. The most critical parts are the merging of the blocks (Step 3) and the transformation from 0/1-blocks to 0&1-blocks (Step 4). We show first that Step 4 can be executed in linear time. The proof of the next lemma is similar to that given in [10, Section 3.2] or [20, Lemma 2, Step 3].

Lemma 2. *Step 4 of the one-to-many transformation can be done in linear time for any blocking factor t.*

Proof. Let $X_1, X_2, \ldots, X_{n/t}$ be the order of the blocks after the block permutation in Step 3. Consider any boundary between a 1-block and a 0-block in this sequence. Since the leader of a 1-block is its last element and the leader of a

0-block is its first element, no element has to be moved across the boundary. Let us therefore divide the sequence $X_1, X_2, \ldots, X_{n/t}$ into pieces, where each piece consists of two subsequences, a sequence of 0-blocks followed by a sequence of 1-blocks. Let us number the pieces from 1 to p.

Consider an arbitrary piece $X_{i_1}, X_{i_2}, \ldots, X_{i_j}$ and assume that this piece contains ℓ_i 0-blocks and m_i 1-blocks. Now only some of the zeros in the last 0-block should be moved to the left and some of the ones in the first 1-block should be moved to the right (see Fig. 2). The zeros to be moved in the last 0-block are obtained into their correct blocks by performing at most m_i block interchanges. Each one is involved in at most one block interchange. Therefore the work here is proportional to $m_i \cdot t$. In the same way, one can show that the work required when moving the ones (now at the end of the first 0&1-block) to the right is proportional to $\ell_i \cdot t$. Since $\sum_{i=1}^{p}(\ell_i + m_i) = n/t$, the claim follows. $\square$

The question that remains to be answered is how the merging in Step 3 is implemented. First, assume that the blocking factor is $\lg n$. Now one possibility is to store the sites of the leaders explicitely. If these are available, Step 3 can be implemented by using any in-place merging algorithm. Since the blocks are of equal size they can be easily swapped in time proportional to their size. Hence, Step 3 can be done $O(n)$ time.

Algorithm B performs the one-to-many transformation as described above and solves the subproblems of size $\lg n$ by Algorithm A. Now Lemma 1 implies the result of the next lemma.

Lemma 3. *Algorithm B solves a stable unpartitioning problem of size n $(= 2^k)$ in $O(n)$ time with $O(n/\log_2 n)$ counters, each requiring $O(\log_2 n)$ bits.*

In Algorithm C the sites of the leaders are stored in a bit array. This means that we need $O(\log_2 n)$ time when manipulating a site. Hence the total time needed for the one-to-many transformation is $O(n \log_2 n)$. However, the number of element moves is only linear! The resulting 0&1-blocks are unpartitioned by Algorithm B. The critical observation is that we have to store only $O(\log_2 n/\log_2 \log_2 n)$ counters, each of $O(\log_2 \log_2 n)$ bits (and $O(1)$ indices, each of $O(\log_2 n)$ bits). The total number of bits required is only $O(\log_2 n)$. Therefore we can pack the integers into few words and manipulate them effeciently with shift operations. Thus each block is handled in $O(\log_2 n)$ time using $O(1)$ words of $O(\log_2 n)$ bits. The performance of Algorithm C is stated in the following lemma.

Lemma 4. *Algorithm C solves a stable unpartitioning problem of size n $(= 2^k)$ in $O(n \log_2 n)$ time, using an array of $O(n)$ bits and a constant amount of words of $O(\log_2 n)$ bits, but makes only $O(n)$ moves.*

The space requirements can be further reduced by using bit stealing. (Note that in order to use bit stealing the f-function must be known.) Let us now divide the input into blocks of size $\sqrt{n}$ (or more precisely into blocks of size $2^{(\lg n)/2}$). The first $c\sqrt{n}$ zeros and $c\sqrt{n}$ ones are saved in an internal buffer, where c is a suitably chosen constant. Of course, it might happen that we do not have as many zeros or ones as needed. Such an input instance is however easily solved by moving the elements we fall short of to their proper places one-by-one. For example, if we run short of zeros, each zero would be involved in at most $c\sqrt{n}$ block interchanges, whereas each one in at most one interchange. This totals $O(n)$ time. The same can be done if we run short of ones. Hence, assume that we have sufficiently many zeros and ones.

One can view the merging task in the one-to-many transformation as an unpartitioning problem (cf. the discussion in the beginning of this section). Now this unpartitioning is implemented by Algorithm C and the elements of the internal buffer are used to steal the bits needed. The new placement array is computed in linear time by scanning through the original placement array. Since the size of the placement array created is about $\sqrt{n}$ it can be stored as a part of the internal buffer. Step 3 of the transformation requires $O(\sqrt{n}\log_2 n)$ time for comparisons and index calculations, and $O(\sqrt{n})$ block swaps; so $O(n)$ time in total. Hence, the whole transformation requires linear time. The subproblems of size $\sqrt{n}$ are also solved by Algorithm C and the bits required are stolen from the internal buffer. The post-processing step, where the elements of the internal buffer are moved to their proper places, is again done in linear time by repeated block interchanges.

We have thus obtained a new algorithm, call it Algorithm D, which is as fast as Algorithm C but requires only a constant amount of additional space.

Lemma 5. *Algorithm D solves a stable unpartitioning problem of size n ($= 2^k$) in $O(n\log_2 n)$ time and constant extra space, but makes only $O(n)$ moves.*

Our final algorithm, Algorithm E is again based on $\lg n$-blocking. The general structure of Algorithm E is similar to that of the previous algorithms. Now Algorithm D is employed for implementing Step 3 of the one-to-many transformation and Algorithm B for unpartitioning the blocks of size $\lg n$. As in Algorithm D the one-to-many transformation takes $O(n)$ time, but now only a constant amount of additional space is needed. As in Algorithm C, we use the technique of packing small integers to solve the subproblems in $O(\log_2 n)$ time with a constant number of words of $O(\log_2 n)$ bits. The total time for solving the subproblems is linear. Therefore Algorithm E requires $O(n)$ time and $O(1)$ extra space.

Up to now we have assumed that n, the number of elements is a power of 2. If this is not the case, the following method can be used to reduce the original problem to subproblems, whose size is a power of 2. First, compute by repeated doubling the largest 2^k that is smaller than n. Second, scan through the first 2^k positions of the placement array and count the total number of 0-bits n_0 and 1-bits n_1 in there. Third, interchange the block of zeros (if any) lying after the first n_0 zeros with the block of the first n_1 ones. Fourth, unpartition the first 2^k elements with Algorithm E. Finally, use the same method for unpartitioning the last $n - 2^k$ elements. Since Algorithm E runs in linear time, the running time of this method is proportional to $\sum_{k=\lfloor\log_2 n\rfloor}^{0} 2^k$, which totals $O(n)$ time. As compared to Algorithm E, the space requirements are increased only by an additive contant. Hence, we have proved the following theorem.

Theorem 1. *A stable unpartitioning problem of size n can be solved in $O(n)$ time and $O(1)$ extra space.*

4 Restoring selection

In this section we implement the (slow) linear-time selection algorithm of Blum et al. [1] to solve the restoring selection problem space-efficiently. In the next section this algorithm is then used to solve the stable selection problem in minimum space.

Let us recall the essence of the prune-and-search algorithm for selecting the kth smallest element in the multiset S of n elements (cf. the implementation given in [5, Algorithm 3.17] which requires $O(\log_2 n)$ extra space):

1. If n is "small" then determine the median p of S in a brute force manner and return p.
2. Divide S into $\lfloor n/5 \rfloor$ blocks of size 5, ignore excess elements.
3. Let M be the set of medians of these blocks. Compute the median p of M by applying the selection algorithm recursively.
4. Partition S stably into three parts $S_<, S_=$, and $S_>$ such that each element of $S_<$ is less than p, each element of $S_=$ is equal to p, and each element of $S_>$ is greater than p.
5. If $|S_<| < k \leq |S_<| + |S_=|$ then return p. Otherwise call the selection algorithm recursively to find the kth smallest element in $S_<$ if $k \leq |S_<|$, or the $(k - |S_<| - |S_=|)$th smallest element in $S_>$ if $k > |S_<| + |S_=|$.

Next we describe the implementation details that will make it possible to restore the elements into their original positions. In Step 1 the median of small sets is computed by the quadratic algorithm that will not move the elements. (We do not specify, when to switch to the brute force algorithm, but refer to any textbook on algorithms, e.g., [5, Section 3.6].) In Step 3 the medians of the blocks are also found without moving the elements. To access a block median we store an offset indicating the place of the median inside the block. Here we need $3n/5 + O(1)$ bits in total. A convenient place to store the set M is at the front of the input array. In [5, Algorithm 3.17] it is shown how the elements of M are moved in-place. It is easy to reverse this computation.

In Step 4 the multiset S is partitioned stably by using the linear-time minimum space algorithm [11]. Now we use $2n$ bits to indicate whether before partitioning the corresponding position contained an element of $S_<, S_=$, or $S_>$. By using the stable unpartitioning algorithm developed in Section 3, we can reverse the computation done in Step 4. In Step 5 it is again convinient to move the multiset $S_\diamond (\diamond \in \{<, >\})$ that we shall work with to the front of the array. If the multisets are stored in order $S_<, S_>, S_=$ or $S_>, S_<, S_=$, the block interchanges performed can be easily reversed.

The sizes of the manipulated multisets are stored in unary form. At each "recursive call" we have to store also the type of the call telling whether the procedure was called in Step 3 or Step 5. When these sizes and types are available the recursive calls can be handled iteratively. The overall organization of the storage is simply a “stack” of bit sequences. Of course, these sequences are stored in a bit array. From the standard analysis of the prune-and-search algorithm it follows that the total number of extra bits needed is linear.

We summarize the above discussion in the following theorem.

Theorem 2. *The restoring selection problem of size n can be solved in $O(n)$ time using an extra array of $O(n)$ bits and a constant amount of words of $O(\log_2 n)$ bits.*

In adaptive sorting it is extremely important not to destroy the existing order among the input data. Therefore our algorithm for restoring selection could be used to improve the space-efficiency of some adaptive sorting algorithms, e.g., that of Slabsort presented in [16]. We leave it as an open problem whether there exists a minimum space algorithm for restoring selection. Such an algorithm would make it possible to develop new in-place sorting algorithms that are also adaptive.

5 Stable selection

Next we show that stable selection is possible in linear time in minimum space. Our construction is based on a minimum space algorithm for selecting an approximate median. When this is used as a subroutine in the standard prune-and-search algorithm (cf. Section 4), instead of the median-of-medians method, an in-place algorithm for stable selection is obtained.

Let $S = \{x_1, x_2, \ldots, x_n\}$ be a multiset. Further, let the *rank* of an element $x_j \in S$ be the cardinality of the multiset $\{x_i \in S \mid x_i < x_j \text{ or } (x_i = x_j \text{ and } i \leq j)\}$. An element x is said to be an *approximate median* of S, if there exists an element $x_i \in S$ such that $x_i = x$ and that the rank of x_i is in the interval $[\alpha n..(1-\alpha)n]$, for some fixed constant α, $0 < \alpha \leq 1/2$. In the following, we do not try to determine any value for the constant α; the existence of such a constant is enough for our purposes.

To find an approximate median for a multiset S such that the relative order of the identical elements is not changed, we use $\sqrt{n}$-blocking. The median of blocks of size $\sqrt{n}$ is computed by the algorithm of Section 4. After computing the median of a block, the block is partitioned stably and in-place such that the elements equal to the median come to the front of the block. Then the first element of each block is used to find the median of medians. Here we use the trivial quadratic-time algorithm that do not move the elements. It is easy to see that the final output is an approximate median of S (cf. the analysis of the standard selection algorithm). The overall running time is linear and the number of extra bits needed $O(\sqrt{n})$.

This algorithm can be further improved by bit stealing. Next we show how the extra bits can be stolen from an internal buffer which is created as a preprocessing step. Our technique is similar to that used by Lai and Wood [14] in their selection algorithm, or Levcopoulos and Petersson [15] in their adaptive sorting algorithm. The contribution here is that the bits can be stolen without losing stability.

Assume that t bits are needed, $t \in O(\sqrt{n})$. Let S_0 denote the first $2t$ elements of the original input S. Now sort S_0 stably for example by the straight selection sort algorithm. This takes $O(t^2)$ time, that is in our case linear time. First, consider the case where none of the elements appears more than t times in S_0. By pairing the first element with the $(t+1)$st element, the second element with the $(t+2)$nd element, and so on, t pairs of different elements are obtained. These pairs are then used to represent the bits required. An approximate median is then searched for the elements in $S \setminus S_0$. Since the size of the buffer is proportional to $\sqrt{n}$ the result will still be an approximate median (under the assumption that n is large enough, but recall that small multisets are handled separatively).

Second, assume that some element x appears more than t times in S_0. Partition S (including S_0) stably into two parts: S_1 containing the elements equal to x, and S_2 containing the elements not equal to x. If the cardinality of S_2 is less than t the input instance is easy. The block S_2 is sorted by the stable, quadratic-time selection-sort algorithm and then S_1 is embedded into the result of this sort by a single block interchange. In this case, even the actual median can be returned in linear time. If the cardinality of S_2 is greater than t, the first t elements of S_1 (forming S_3) and the first t elements of S_2 (forming S_4) are used to create the internal buffer. To do this the blocks $S_1 \setminus S_3$ and S_4 are interchanged. Finally, an approximate median is searched for the elements belonging to $S \setminus (S_3 \cup S_4)$.

To summarize, an approximate median of n elements can be found in $O(n)$ time, using $O(1)$ extra space, such that the relative order of the identical elements is retained. The routine for finding an approximate median can be applied in the prune-and-search selection algorithm, instead of using the median-of-medians method. Hence, the result of the following theorem follows from the analysis of the prune-and-search algorithm.

Theorem 3. *The stable selection problem of size n can be solved in $O(n)$ time, using only $O(1)$ extra space.*

6 Stable sorting of multisets

In this section we describe and analyse a Quicksort type algorithm that sorts multisets stably in optimal time and minimum space. Let us assume that S, the multiset to be sorted is non-empty. The basic steps of the algorithm are:

1. Find the median p of S.
2. Partition S stably into three parts $S_<, S_=, S_>$ such that each element of $S_<$ is less than p, each element of $S_=$ is equal to p, and each element of $S_>$ is greater than p.
3. Sort the two multisets $S_<$ and $S_>$ recursively if they are not empty.

In Step 1 the median is determined stably and in-place by the algorithm of Section 5. Step 2 is implemented stably and in-place by using the algorithm given in [11]. To avoid the recursion stack in Step 3 we can use the implementation trick — based on stoppers — proposed by Ďurian [3]. His Quicksort implementation performs two-way partitions, but it is easily modified to handle three-way partitions as well. We describe the method here in order to show that stability is not lost when using stoppers.

For the sake of simplicity, we assume that there exist two elements p and q such that, for all $x \in S$, $x < p \geq q$. Further, assume that the multiset S is given in the array $S[1..n]$, and assign $S[n+1] = p$ and $S[n+2] = q$. If these extra elements are not available beforehand, we can find such as follows. Let x be equal to the second largest element of S. We perform now a three-way partition of $S[1..n]$ stably and in-place by using x as a partitioning element. The first element equal to x is chosen as p and the the element right after it as q. The total time required by this preprocessing is clearly linear. The elements smaller than p can then be sorted by the procedure to be described below.

Consider the case when we are solving the subproblem $S[\ell..h]$ followed by the elements p and q as described above. The invariant of the algorithm is that after sorting $S[\ell..h]$ the next subproblem to be solved can be determined by using h only. Let us assume that the median partitions $S[\ell..h]$ into the three parts $S_< = S[\ell..h_<]$, $S_= = S[h_1+1..\ell_> - 1]$, and $S_> = S[\ell_>..h]$. The correctness of the algorithm is established by induction. It follows from the induction hypothesis that the next subproblem after $S_>$ can be determined by using h only. Therefore the main task illustrated in Fig. 3 is to show how $S_>$ is recovered after sorting $S_<$.

Before solving the subproblem $S[\ell..h_1]$ recursively, we swap the elements $S[\ell_3]$ and $S[h+1]$. When sorting $S_<$, we can use the first two elements of $S_=$ in the role of p and q because they are greater than any element of $S_<$. If $S_=$ is a singleton set, we can use the first element of $S_>$ as q. After the recursion terminates for $S_<$, that is, sorting is done and swapped elements are restored to their correct places, we start a scan from $h_1 + 1$ until the first element p larger than $S[h_1+1]$ is found. The index ℓ' of p is the left border of the next subproblem. Then we scan further to find the first element q larger than or equal to p. Let the index of q be h'. We restore the correct order by swaping the elements p and $S[h'-1]$. The right border of the next subproblem is therefore $h' - 2$. After determining the borders, the next subproblem can be processed.

Now we are ready to prove our main result.

Theorem 4. *Quicksort can be adapted to sort stably a multiset of size n with multiplicities $n_1, n_2, \ldots, n_m$ in $O(n \log_2 n - \sum_{i=1}^{m} n_i \log_2 n_i + n)$ time and $O(1)$ extra space.*

Proof. According to the previous discussion our implementation is stably and in-place. So let us concentrate on analysing the running time of the algorithm.

Let $S_1, S_2, \ldots, S_m$ be the minimum partition of the input into classes of equal elements. Without loss of generality, we can assume that the elements in S_i are smaller than those in S_j, for all $i < j$. Furthermore, let the cardinality of these subsets be $n_1, n_2, \ldots, n_m$, respectively. Now we denote by $T(i..k)$ the time it takes to sort the classes $S_i, S_{i+1}, \ldots, S_k$. Since median finding and partitioning are done in linear time, there exists a constant c such that the running time of the algorithm is bounded by the following recurrence

$$T(i..k) \leq \begin{cases} T(i..j-1) + T(j+1..k) + c(\sum_{h=i}^{k} n_h) & \text{for } i < k \text{ and} \\ cn_i & \text{for } i = k. \end{cases}$$

Let us use the following shorthand notations: $N_1 = \sum_{h=i}^{j-1} n_h$, $N_2 = \sum_{h=j+1}^{k} n_h$, $N = \sum_{h=i}^{k} n_h$. It is easy to establish by induction that $T(i..k) \leq c(N \log_2 N - \sum_{h=i}^{k} n_h \log_2 n_h + N)$ ($0 \log_2 0$ means 0). This is because $N_i \leq N/2$ ($i = 1, 2$) and therefore $N_1 \log_2 N_1 + N_2 \log_2 N_2 + n_j \log_2 n_j + N_1 + N_2 \leq N \log_2 N$. Hence we have proved that $T(1..m) \in O(n \log_2 n - \sum_{i=1}^{m} n_i \log_2 n_i + n)$. $\square$

Acknowledgements

The discussions with Niels Christian Juul and Ola Petersson are gratefully acknowledged.

References

1. M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, R.E. Tarjan: Time bounds for selection. *Journal of Computer and System Sciences* **7** (1973) 448–461.
2. S. Carlsson, J.I. Munro, P.V. Poblete: An implicit binomial queue with constant insertion time. *Proc. of the 1st Scandinavian Workshop on Algorithm Theory.* Lecture Notes in Computer Science **318**. Springer-Verlag, 1988, pp. 1–13.
3. B. Ďurian: Quicksort without a stack. *Proc. of the 12th Symposium on Mathematical Foundations of Computer Science.* Lecture Notes in Computer Science **233**. Springer-Verlag, 1986, pp. 283–289.
4. T.F. Gonzalez, D.B. Johnson: Sorting numbers in linear expected time and optimal extra space. *Information Processing Letters* **15** (1982) 119–124.
5. E. Horowitz, S. Sahni: *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.
6. E.C. Horvath: Stable sorting in asymptotically optimal time and extra space. *Journal of the ACM* **25** (1978) 177–199.
7. B.-C. Huang, M.A. Langston: Practical in-place merging. *Communications of the ACM* **31** (1988) 348–352.
8. B.-C. Huang, M.A. Langston: Fast stable merging and sorting in constant extra space. *Proc. of the 1st International Conference on Computing and Information*, 1989, pp. 71–80.
9. B.-C. Huang, M.A. Langston: Stable dublicate-key extraction with optimal time and space bounds. *Acta Informatica* **26** (1989) 473–484.
10. J. Katajainen, C. Levcopoulos, O. Petersson: Space-efficient parallel merging. Informatique Théorique et Applications, to appear. A preliminary version in *Proc. of the 4th International PARLE Conference (Parallel Architectures and Languages Europe).* Lecture Notes in Computer Science **605**. Springer-Verlag, 1992, pp. 37–49.

11. J. Katajainen, T. Pasanen: Stable minimum space partitioning in linear time. *BIT*, to appear.
12. D. E. Knuth: *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, 1973
13. M.A. Kronrod: Optimal ordering algorithm without operational field. *Soviet Mathematics* **10** (1969) 744–746.
14. T.W. Lai, D. Wood: Implicit selection. *Proc. of the 1st Scandinavian Workshop on Algorithm Theory.* Lecture Notes in Computer Science **318**. Springer-Verlag, 1988, pp. 14–23.
15. C. Levcopoulos, O. Petersson: An optimal adaptive in-place sorting algorithm. *Proc. of the 8th International Conference on Fundamentals of Computation Theory.* Lecture Notes in Computer Science **529**. Springer-Verlag, 1991, pp. 329–338.
16. C. Levcopoulos, O. Petersson: Sorting shuffled monotone sequences. *Information & Computation*, to appear.
17. A. Moffat, O. Petersson: An overview of adaptive sorting. *The Australian Computer Journal* **24** (1992) 70–77.
18. J.I. Munro: An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences* **33** (1986) 66–74.
19. J.I. Munro, V. Raman: Sorting multisets and vectors in-place, *Proc. of the 2nd Workshop on Algorithms and Data Structures.* Lecture Notes in Computer Science **519**. Springer-Verlag, 1991, pp. 473–480.
20. J.I. Munro, V. Raman, J.S. Salowe: Stable in situ sorting and minimum data movement. *BIT* **30** (1990) 220–234.
21. J.I. Munro, P.M. Spira: Sorting and searching in multisets. *SIAM Journal on Computing* **5** (1976) 1–8.
22. J.S. Salowe, W.L. Steiger: Simplified stable merging tasks. *Journal of Algorithms* **8** (1987) 557–571.
23. J.S. Salowe, W.L. Steiger: Stable unmerging in linear time and constant space. *Information Processing Letters* **25** (1987) 285–294.
24. L.M. Wegner: Quicksort for equal keys. *IEEE Transactions on Computers* **C34** (1985) 362–367.

`(a)` $$\underline{0_5\ 0_7\ 0_8\ 0_9}\quad\underline{0_{12}\ 0_{14}\ 1_1\ 1_2}\quad\underline{1_3\ 1_4\ 1_6\ 1_{10}}\quad\underline{1_{11}\ 1_{13}\ 1_{15}\ 1_{16}}$$

`(b)` `1 1 1 1 0 1 0 0 0 1 1 0 1 0 1 1`

`(c)` $$\underline{0_5\ 0_7\ 0_8\ 0_9}\quad\underline{1_1\ 1_2\ 1_3\ 1_4}\quad\underline{1_6\ 1_{10}\ 1_{11}\|0_{12}}\quad\underline{0_{14}\ 1_{13}\ 1_{15}\ 1_{16}}$$

`(d)` $$\underline{1_1\ 1_2\ 1_3\ 1_4}\quad\underline{0_5\ 0_7\ 0_8\ 0_9}\quad\underline{1_6\ 1_{10}\ 1_{11}\|0_{12}}\quad\underline{0_{14}\ 1_{13}\ 1_{15}\ 1_{16}}$$

`(e)` $$\underline{1_1\ 1_2\ 1_3\ 1_4}\quad\underline{0_5\ 0_7\ 0_8\ 1_6}\quad\underline{0_9\ 1_{10}\ 1_{11}\|0_{12}}\quad\underline{0_{14}\ 1_{13}\ 1_{15}\ 1_{16}}$$

`(f)` $$\underline{1_1\ 1_2\ 1_3\ 1_4}\quad\underline{0_5\ 0_7\ 0_8\ 1_6}\quad\underline{0_9\ 0_{12}\ 1_{10}\ 1_{11}}\quad\underline{0_{14}\ 1_{13}\ 1_{15}\ 1_{16}}$$

Fig. 1. One-to-many transformation. (a) Example input with $n = 16$ and $\lg n = 4$. (b) Placement array. (c) Single 0&1-block is handled. (d) Block permutation is performed. (e) 0/1-blocks are transformed to 0&1-blocks. (f) Half-finished block is cleaned up.

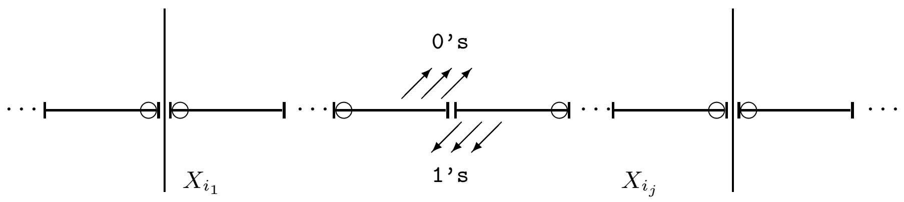
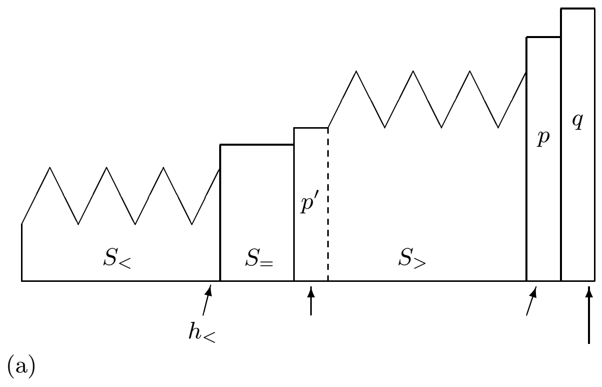


Fig. 2. Block sequence $X_{i_1}, X_{i_2}, \ldots, X_{i_j}$. A leader of each block is marked by a circle.



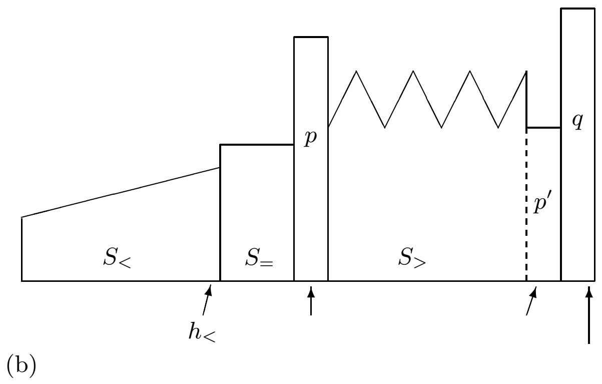


Fig. 3. (a) After a three-way partition of S we swap p' and p. (b) After sorting $S_<$ we search for the borders of $S_>$ by starting a scan from $h_<$.