Generic Algorithm for 0/1-Sorting

Gianni Franceschini[1] Jyrki Katajainen[2,*]

[1] *Max-Planck-Institut für Informatik*
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
gfrances@mpi-inf.mpg.de
[2] *Department of Computing, University of Copenhagen*
Universitetsparken 1, 2100 Copenhagen East, Denmark
jyrki@diku.dk

Abstract. In the 0/1-*sorting problem*, given a sequence S of elements drawn from a universe $\mathcal{E}$ and a characteristic function $f : \mathcal{E} \to \{0, 1\}$, the task is to rearrange the elements in S so that every element x, for which $f(x) = 0$, is placed before any element y, for which $f(y) = 1$. Moreover, this reordering should be done *stably* without altering the relative order of elements having the same f-value, and *space efficiently* using only $O(1)$ words of extra space. In this paper we present a *generic* algorithm for solving the 0/1-sorting problem which works optimally for many different kinds of sequences and characteristic functions. The model of computation used is a word RAM with a two-level memory hierarchy consisting of an ideal cache and an arbitrarily large main memory. The performance of our algorithm can be summarized as follows:

1. Let n denote the length of S. The algorithm performs at most $O(n)$ element exchanges, invocations of f, and word operations.
2. Let $w_f(u)$ denote the amount of work done when applying f to element u. When the cost of the evaluation of the characteristic function is not uniform, but depends on the element f is applied to, the algorithm uses $O\left(\sum_{u \in S} w_f(u)\right)$ work under the assumption that element exchanges involve $O(1)$ work.
3. Let B denote the size of cache blocks measured in elements. The algorithm incurs $O(n/B)$ cache misses in the ideal-cache model under the tall-cache assumption (excluding the cache misses possibly generated by the invocations of f).

Interestingly, our algorithm is neither aware of the characteristics of the cache (hardware obliviousness) nor the implementation of f (software obliviousness).

1. Introduction

We are given a set of n elements in the form of a sequence S. The elements in S are drawn from an arbitrary universe $\mathcal{E}$. Also, we are given a *characteristic*

*Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project Generic programming—algorithms and tools).

function f which is a mapping from set $\mathcal{E}$ to set $\{0, 1\}$. Based on the f-values, we call the elements 0s or 1s. Besides the application of f, the only operation allowed on the elements in S is the exchange of the positions of two of them. In the 0/1-*sorting problem*, the objective is to separate the 0s from the 1s using $O(\log n)$ extra bits of information and retaining the relative order of elements of the same type; that is, the order of all 0s and 1s should be the same in the final sequence as the order in the original sequence.

In the literature, the following special case of the 0/1-sorting problem is called the *stable in-place partitioning problem*: a sequence S and a pivot element v are given and the task is to rearrange the elements in S in-place such that all elements smaller than v precede the elements larger than or equal to v. Munro, Raman, and Salowe [8] gave a solution which has worst-case cost of $O(|S| \log^* |S|)$. Subsequently, Katajainen and Pasanen [6] proposed an improved algorithm which has the optimal worst-case cost of $O(|S|)$. Both of these solutions could be generalized to 0/1-sorting as well.

In this paper we are interested in solving the 0/1-sorting problem in a generic environment. In particular, our target is to develop an algorithm that can be used for the implementation of the C++ standard library function `stable_partition` [2, §25.2.12]. In an abstract form, this function takes two *type parameters*, a sequence type $\mathcal{T}$ and a function type $\mathcal{F}$, and two *data parameters*, an element sequence of type $\mathcal{T}$ and a characteristic function of type $\mathcal{F}$, and as a side-effect the function rearranges the sequence such that all 0s are placed before all 1s and that the relative order of the elements in both groups is preserved. The `stable_partition` function, as any other generic function in the library, takes type parameters, and the function is expected to work well for all potential type parameters, which are unknown at development time.

To illustrate the diverse goals when developing a generic 0/1-sorting algorithm, let us consider two scenarios. First, assume that the input sequence is an array of integers and that the characteristic function performs a single comparison. In this case, the overall performance of the algorithm is determined by its cache behaviour. Second, assume that the input sequence is an array of strings of varying length such that each array entry stores a reference to the corresponding string. In this case, the work required to evaluate the characteristic function f *is not constant*, but depends on the particular element f is applied to. In both of these scenarios the earlier 0/1-sorting algorithms do not perform optimally.

The standard approach used in the development of component libraries is to provide a fan of alternative implementations for different combinations of type parameters. Moreover, since the types of all data parameters are known at compile time, the best suited component can be selected from the fan of alternatives at compile time (see, for example, [3, Chapter 10]). The fundamental problem with this approach is that there are an infinite number of data types so it is impossible to provide all potential specializations in

a program library. Making the library extensible is a possibility, but this solution lacks elegance.

A specification of a data type has two parts: interface and implementation. When developing a generic library component only the interfaces of the data types used are known, but the implementations of the functions called are unknown. We advocate that generic library components should be made—if at all possible—oblivious of the costs caused by these externally defined functions. Additionally, it is important to develop library components that use little extra space so that application programs do not encounter memory allocation problems when using the components. For the space-efficient variant of `stable_partition`, the basic complexity requirement stated in the C++ standard is that at most $n \log_2 n$ element exchanges are performed, n being the number of input elements. We aim at achieving a stronger guarantee of at most $O(n)$ element exchanges. To achieve this and to be space efficient at the same time, we have to rely on random access. (If only sequential access is allowed, the problem can be solved by copying the elements to an array, but this solution is no more space efficient.) We want to make it explicit that our algorithm for 0/1-sorting is generic and space efficient, but unfortunately not practical.

The model of computation used throughout this paper is a word RAM with a two-level memory hierarchy consisting of an ideal cache and an arbitrary large backup memory [5, 10]. We assume that the cache can store M elements and $O(\log n)$ extra bits at any given time. The information transfer between the cache and backup memory is done in blocks of B elements. As usual in the ideal-cache model, we make the standard *tall-cache assumption*, i.e. that $M = \Omega\left(B^2\right)$. When expressing the efficiency of 0/1-sorting algorithms, we use the term *cost* to denote the sum of the number of element exchanges, f-invocations, and word operations performed. By *work*, we mean the total number of word operations performed in the worst case to handle an input instance of particular size. In order to be able to talk about the actual running time—which we consciously avoid—one should further know the duration of individual word operations which may vary depending on the context they are executed. For example, an element access can be expensive if it causes a cache miss.

Let $w_f(u)$ denote the amount of work done when applying f to element u. We present an optimal 0/1-sorting algorithm that uses $O\left(\sum_{u \in S} w_f(u)\right)$ work under the assumption that element exchanges involve $O(1)$ work. Moreover, in the ideal-cache model under the tall-cache assumption, our algorithm incurs $O(n/B)$ cache misses while rearranging the elements (excluding the cache misses possibly generated by the invocations of f). An interesting characteristic of our algorithm is that of being not only oblivious of the parameters of the memory hierarchy, but also *oblivious of the costs of* f. In order to achieve the $O\left(\sum_{u \in S} w_f(u)\right)$ work complexity, the algorithm satisfies the following condition: for any element u in S, f is applied to u a constant number of times throughout the entire computation. As we will see, this condition is particularly difficult to be maintained when space optimality

is required, mainly because of the implicit bit-encoding techniques widely used in space-efficient algorithms.

Summing up, in this paper we will prove the following theorem:

Theorem 1. *When solving the* 0/1-sorting problem *for a sequence S and a characteristic function f, our generic algorithm gives the following performance guarantees:*

(i) It uses $O(\log |S|)$ extra bits and performs $O(|S|)$ element exchanges.

(ii) In the ideal-cache model, assuming $M = \Omega(B^2)$, it incurs $O(|S|/B)$ cache misses while performing the exchanges (excluding the cache misses possibly caused by the invocations of f).

(iii) For any u in S, the characteristic function f is applied to u $O(1)$ times.

We will present our algorithm in Section 3 and the complexity analysis in the appendix. Directly from Theorem 1, we get the following corollary:

Corollary 1. *Under the assumption that element exchanges involve $O(1)$ work, the* 0/1-sorting problem *can be optimally solved using $O(\sum_{u \in S} w_f(u))$ work.*

2. Toolbox

The techniques used in our algorithm are basic and are summarized in this section. The crux of our algorithm is in the way that these techniques are combined.

2.1 Interchanging two consecutive sequences

From a sequence $X = \langle x_1 \dots x_t \rangle$ of t consecutive elements, we can obtain the reverse $X^R = \langle x_t \dots x_1 \rangle$ by exchanging x_1 with x_t, x_2 with x_{t-1}, and so forth. Then the order of two consecutive sequences X and Y can be interchanged stably and in-place with three sequence reversals, since $YX = (X^R Y^R)^R$. Clearly, the cost of the whole interchanging process is $O(|X| + |Y|)$, and since the sequences are processed sequentially, at most $O((|X| + |Y|)/B)$ cache misses are incurred. For a detailed discussion of the cache efficiency of this and other interchanging algorithms, we refer to [1].

2.2 Inverting a permutation

Given a permutation sequence $\Pi = \langle \pi_1, \pi_2 \dots \pi_t \rangle$ and an element sequence $X = \langle x_1, x_2 \dots x_t \rangle$, inverting the given permutation is an operation where element x_i is moved to the place earlier occupied by x_{π_i}, and this is to be done in parallel for all $i \in \{1, 2 \dots t\}$. One way of computing the inverse is to use the cycle-leader algorithm (for other alternatives, see, e.g. [7, Section 1.3.3]). We start by exchanging x_1 and x_{π_1}, simultaneously exchanging π_1 and π_{π_1}, and setting π_{π_1} negative, which indicates that x_1 is already in its final location. In a similar fashion, using the modified permutation sequence we exchange x_1 and x_{π_1}. We go forward in this fashion until a cycle of the permutation is completed (i.e. when $\pi_1 = 1$). Then we continue with the

next unprocessed cycle; the leader of that cycle can be recognized by finding the first non-negative index in the modified permutation sequence. The main drawback of the cycle-leader algorithm is its poor cache behaviour. Therefore, we use it sparingly to only solve small subproblems.

2.3 Packing small integers

If the size of the 0/1-sorting problem is n, we assume that the word size λ of the word RAM is larger than or equal to $\log_2 n$ and that words of that size can be manipulated at unit cost. Under this assumption, the extra bits used by our algorithm can be packed in $O(1)$ words.

Let k and b be integers and assume that $k \times b = O(\lambda)$. When manipulating a sequence of k integers, each consisting of b or fewer bits, the integers can also be packed in $O(1)$ words and the value of an integer can be read and written using $O(1)$ work by applying standard bit-manipulation techniques. To make this possible the word RAM should provide left and right shifts, and bitwise boolean operations. The same set of primitive operations was used in [6].

2.4 Implicit bit encoding

The value of one bit can be encoded in the relative order of two elements with a simple rule: the pair is left in order if the value to be encoded is a zero; otherwise, the order of the elements is reversed. This implicit bit-encoding technique was introduced by Munro [9] in 1986, and thereafter extensively used in many in-place algorithms in various forms (see [4] and the references mentioned therein).

The encoded bits can be grouped into words of length κ that can be used as normal words. However, the encoded memory has two major limitations: (i) it is *slow*, since decoding the value of a word requires $O(\kappa)$ applications of f and encoding a new value requires $O(\kappa)$ applications of f and $O(\kappa)$ element exchanges in the worst case; (ii) each word of the encoded memory can be used, read or written, *only* $O(1)$ *times* (that is a consequence of the fact that we want to apply f only $O(1)$ times for each input element during the entire computation).

2.5 Recursion

Often recursion is avoided in in-place algorithms because a recursion stack may become the main obstacle to achieving the desired space bound. We use recursion, but we take the following measures to avoid the growth of the recursion stack. First, we maintain $O(\log |P_I|)$ bits for each recursive invocation I on a subproblem P_I of size $|P_I|$. We use a variable-length bit encoding when storing integers in the recursion stack. Additionally, we ensure that each integer can be encoded and decoded using $O(\log |P_I|)$ work (e.g. universal coding could be used). Second, we make sure that the sizes

of nested subproblems scale at least quadratically. Third, during the computation we handle the recursive subproblems in a depth-first fashion. So, globally, we have to maintain $O(\log |P_I|)$ bits for any nested invocation I, for a total of $O(\log n)$ bits assuming that the size of the whole problem is n, and another $O(\log n)$ bits for the global index that stores the starting position of the subsequence considered in the *active* (deepest) nested invocation. Fourth, when the active invocation, say I, is suspended to execute a new nested invocation J within I, we add the $O(\log |P_I|)$-bit integer, representing the offset within P_I of the recursive subproblem associated to J, to the global index of $O(\log n)$ bits. When J returns, we subtract the saved offset from the global index of $O(\log n)$ bits (that, by induction, is now set again to the value it had before the execution of J) and we go on the next nested invocation within I we have to perform.

3. Algorithm

The computation begins with the execution of CREATE_UNITS(S, f) (see Section 3.1) where the original sequence S is transformed into a sequence U defined as follows. U is divided into *units* $U_1, U_2 \dots U_t$ of $q = \log^\gamma |S|$ contiguous elements each, where $\gamma \geq 3$ is an integer constant. For each unit U_i of U, we have that: (i) the elements of U_i are of the *same type*; (ii) U_i is logically divided into *packets* $U_{i,1}, U_{i,2} \dots$ of $\log |S|$ contiguous elements each. The case where the number of 0s and the number of 1s are not multiples of the unit size q can be treated easily. After U is computed, the execution of SORT_UNITS$(U, f, 1)$ (see Section 3.2) delivers the final sequence.

3.1 CREATE_UNITS(S, f)

The invocation of CREATE_UNITS(S, f) performs the following seven main steps.

Step 1. Let us divide S into $S_1 S_2 \dots S_p$ contiguous subsequences of $\sqrt{\log |S|}$ elements each. We sort S_1 using the elements in S_p as placeholders in the following way. We first scan S_1 from left to right looking for 0s and each time we find one, we exchange it with the leftmost original element of S_p. Then, we scan S_1 from right to left looking for 1s and each time we find one, we exchange it with the rightmost original element of S_p. After that, the elements originally in S_1 are now in sorted order in S_p and the elements originally in S_p are now arbitrarily permuted in S_1. Finally, we exchange S_1 and S_p so that the elements originally in S_1 are brought back in sorted order. In this process the initial order of the elements in S_p is *lost*. However, it is easy to see that we need only $o(\log |S|)$ extra bits to store that initial order ($\log \log |S|$ bits for any of the $\sqrt{\log |S|}$ elements of S_p). Therefore, the stability is temporarily lost in S_p, but we can maintain the original order by storing the position of each element in a packed form. Each time an element is moved its position is moved accordingly. After S_1, we repeat the same process for each S_i, for $2 \leq i \leq p-1$, each time changing the packed bits to

reflect the new permutation of the elements of S_p. It is important to point out that during the entire first step, *we do not need to apply f to any of the placeholder elements in S_p.*

Step 2. After the first step, $S_1, S_2 \dots S_{p-1}$ are sorted and we can recover the original order of S_p by permuting its elements according to the information stored in the $o(\log |S|)$ extra bits. Here the cycle-leader algorithm can be applied. Finally, we sort S_p in the following way. With $O(1)$ scans of S_p we can easily compute the rank of any element of S_p (that is, its position in S_p if S_p was sorted). Using the encoded ranks, S_p can be sorted with the cycle-leader algorithm.

Step 3. We pass from sequence $S = S_1 S_2 \dots S_p$ where each S_i is sorted to $H = H_1 H_2 \dots H_p$, where, for each H_i, the $\sqrt{\log |S|}$ elements of H_i are of the same type (actually H_p could contain both 0s and 1s but that is a simple special case). Since each S_i is sorted, the task is quite simple. Let $S_i = Z_i O_i$, where Z_i (O_i) contains the 0s (1s) of S_i. We start considering S_1 and S_2. If $|Z_1| + |Z_2| \leq |O_1| + |O_2|$ ($|Z_1| + |Z_2| > |O_1| + |O_2|$), we pass from $S_1 S_2 = Z_1 O_1 Z_2 O_2$ to $H_1 T_1 = O_1 O_2 Z_1 Z_2$ ($H_1 T_1 = Z_1 Z_2 O_1 O_2$) with $O(1)$ sequence exchanges. We do the same with T_1 and S_3, obtaining H_2 and T_2, then with T_2 and S_4 obtaining H_3 and T_3, and so forth.

Step 4. After the third step we have $H = H_1 H_2 \dots H_p$ where $|H_i| = \sqrt{\log |S|}$ and H_i is homogeneous, for any i, $i \in \{1, 2 \dots p\}$. Let us divide H into $p' = p/\sqrt{\log |S|}$ groups $S'_1, S'_2 \dots S'_{p'}$ such that $S'_1 = H_1 H_2 \dots H_{\sqrt{\log |S|}}$, $S'_2 = H_{\sqrt{\log |S|}+1} H_{\sqrt{\log |S|}+2} \dots H_{2\sqrt{\log |S|}}$, and so forth. It is easy to see that subsequences $S'_1, S'_2 \dots S'_{p'-1}$ can be sorted using the homogeneous subsequences composing $S'_{p'}$ as placeholders as we did in the first step. The process is the same but now the basic objects we have to handle are not single elements but single subsequences H_i.

Step 5. The original order of the subsequences composing $S'_{p'}$ can be recovered the same way we did in the second step. Again, the process is the same but now the basic objects we have to permute back in their original order are not single elements but single subsequences H_i. $S'_{p'}$ can be finally brought in sorted order with the same cycle-leader permuting technique used in the second step.

Step 6. Finally, we pass from sequence $S' = S'_1 S'_2 \dots S'_{p'}$ where each S'_i is sorted to $H' = H'_1 H'_2 \dots H'_{p'}$, where, for each H'_i, the $\sqrt{\log |S|}$ *homogeneous subsequences* of H'_i are of the same type. Once again, we use the same techniques used in the third step just scaled up.

Step 7. We repeat the fourth, fifth, and sixth steps until we end up with *units* $U_1, U_2 \dots U_t$ of q contiguous elements each. Since each iteration of the fourth, fifth, and sixth steps enlarges the sizes of the homogeneous contiguous subsequences by a factor $\sqrt{\log |S|}$, we need only a constant number of iterations.

Lemma 1. *For any sequence S, the execution of* CREATE_UNITS(S, f) *has the following properties:*

(i) It uses $O(\log |S|)$ extra bits and performs $O(|S|)$ element exchanges.

(ii) In the ideal-cache model, assuming $M = \Omega(B^2)$, it incurs $O(|S|/B)$ cache misses while performing the exchanges.
(iii) It applies f to v $O(1)$ times, for any v in S.

3.2 SORT_UNITS(U, f, ℓ)

The sequence U received as input is composed of *units* $U_1, U_2 \dots U_t$ of q contiguous elements each. The elements of each unit U_i are of the *same type* and U_i is logically divided into packets of $\log |S|$ contiguous elements each and denoted by $U_{i,1}, U_{i,2} \dots$ As the name suggests, the units of U will be the main objects on which SORT_UNITS will operate. We will call a unit entirely composed of 0s (1s) a 0-*unit* (1-*unit*). During the computation, we will often have to classify a unit by applying f to one of its elements. As we will see, some units may be involved in recursive calls of SORT_UNITS and for that reason we have to pay attention not to accumulate applications of f on the same element of a unit involved in nested recursive calls. The input parameter ℓ will give us a little help in order to circumvent this problem.

In the following, for any sequence of units W, we will denote with $W[i]$ and $W[i \dots j]$ the ith unit of W and the sequence of units from $W[i]$ to $W[j]$, respectively. Occasionally, we will use the same notation to denote *single elements* of a sequence. When we use this notation, its meaning will be clear from the context. Unlike the previous notation, for any sequence W, $|W|$ will always denote the number of elements in it.

We have five main phases.

3.3 First phase: is U very unbalanced?

In the first phase, we handle the particular case where either the number of 0s or the number of 1s in U is less than $\sqrt{|U|}$. The first phase has the following two steps.

Step 1. We establish how many 0s and 1s occur in U. In order to do so, we have to classify only the units. Each time we classify a unit U_i, we apply f to the first element of the ℓth packet of U_i. Let z and o be the number of 0s and 1s, respectively. We store the values of z and o into $O(\log |U|)$ extra bits for the rest of the algorithm. If $z \geq \sqrt{|U|}$ and $o \geq \sqrt{|U|}$, we proceed directly to the next phase. Otherwise, we execute the next step of this phase and then the computation ends.

Step 2. Let us suppose that the number of 0s is less than $\sqrt{|U|}$ (the other case is symmetric). We start scanning U from the right end until we find the two rightmost 0-units; let them be in positions i_2 and i_1, respectively. Each time we have to classify a unit U_i we apply f to the first element of the ℓth packet of U_i. We exchange $U[i_1]$ with $U[i_2 + 1 \dots i_1 - 1]$ obtaining a new sequence $U^1 Z_2 O^1$ where $Z_2 = U[i_2]U[i_1]$. We continue scanning U^1 from the right end, until we find the next 0-unit; let it be in position i_3 in U^1. We exchange Z_2 with $U^1[i_3 + 1 \dots i_2 - 1]$, thus obtaining a new sequence $U^2 Z_3 O^2$ where $Z_3 = U[i_3]U[i_2]U[i_1]$. We continue in this fashion until there

are no more 0-units, ending up with a sequence $O'ZO''$ where all the 0-units are in Z. Finally, we exchange Z with O' and we are done.

Lemma 2. *The first phase of the execution of* Sort_Units(U, f, ℓ) *has the following properties:*

(i) It uses $O(\log|U|)$ extra bits and performs $O(|U|)$ element exchanges.
(ii) In the ideal-cache model, assuming $M = \Omega(B^2)$, it incurs $O(|U|/B)$ cache misses while performing the exchanges.
(iii) For any unit U_i in U, it applies f exclusively to the first element of $U_{i,\ell}$ (the ℓth packet of U_i) and only $O(1)$ times.

3.4 Second phase: finding some 0*s and* 1*s*

In this phase we extract as many 0s and 1s we can. We will use them for encoding purposes in the subsequent phases. The second phase has the following two steps.

Step 1. Let us logically divide U into $\log|U|$ *zones* $Q_1Q_2\ldots$ of $|U|/(q\log|U|)$ contiguous units each (and $|U|/\log|U|$ contiguous elements each). Let Q^z and Q^o be the zones with the largest number of 0s and 1s, respectively. We sort Q^z and Q^o with the following slight variation of binary mergesort. As usual, the process has $\log(|U|/(q\log|U|))$ passes where pairs of sorted subsequences of increasing sizes are merged. Let us consider an arbitrary kth pass where we have to merge pairs of subsequences of 2^k units each. Let R' and R'' be two contiguous subsequences to be merged. Scanning them, we find the subsequences Z', Z'', O', O'' such that $R' = Z'O'$, $R'' = Z''O''$ and where Z' and Z'' (O' and O'') contain the 0-units (1-units) of R' and R'', respectively. During the scans the classification of any unit U_i considered is done *by applying f to the kth element of the ℓth packet of U_i*. When Z', Z'', O', O'' and their boundaries have been found, we do the actual merging step passing from $R'R'' = Z'O'Z''O''$ to $R''' = Z'Z''O'O''$ with a simple sequence exchange.

Step 2. Once they are sorted, we move zones Q^z and Q^o to the beginning of the sequence, passing from $U = Q_1\ldots Q_{i_z-1}Q^zQ_{i_z+1}\ldots Q_{i_o-1}Q^oQ_{i_o+1}\cdots$ to Q^zQ^oU', where U' contains the remaining unsorted zones (the case where Q^z follows Q^o is analogous). Clearly with this step we lose the stability but we can store enough information to recover it later, when also U' is sorted. It is sufficient to store the following information into $O(\log|U|)$ extra bits: (i) the ranks in U of the leftmost 0 and the leftmost 1 (if any) of Q^z; (ii) the ranks in U of the leftmost 0 (if any) and the leftmost 1 of Q^o.

Lemma 3. *The second phase of the execution of* Sort_Units(U, f, ℓ) *has the following properties:*

(i) It uses $O(\log|U|)$ extra bits and performs $O(|U|)$ element exchanges.
(ii) In the ideal-cache model, assuming $M = \Omega(B^2)$, it incurs $O(|U|/B)$ cache misses while performing the exchanges.
(iii) For any unit $U_i \in Q^z \cup Q^o$, it applies f exclusively on the elements of $U_{i,\ell}$ (the ℓth packet of U_i) and only $O(1)$ times for each of these elements.

(iv) For any unit $U_i \notin Q^z \cup Q^o$, it applies f exclusively to the first element of $U_{i,\ell}$ (the ℓth packet of U_i) and only $O(1)$ times.

It is easy to see that the number of 0s in Q^z and the number of 1s in Q^o are greater than or equal to $min(z,o)/\log|U|$. Let $Q^z = \hat{Z}\hat{Q}^z$ and $Q^o = \hat{Q}^o\hat{O}$, where $\hat{Z}$ ($\hat{O}$) contains the $min(z,o)/\log|U|$ leftmost (rightmost) 0s (1s) of Q^z (Q^o). In the subsequent phases we will use the $min(z,o)/\log|U|$ pairs of elements $(\hat{Z}[1],\hat{O}[1]),(\hat{Z}[2],\hat{O}[2])\ldots$ to implicitly encode the values of $min(z,o)/\log|U|$ bits of information.

3.5 Third phase: is recursion needed?

After the second phase we have the sequence $\hat{Z}\hat{Q}^z\hat{Q}^o\hat{O}U'$ and the main task left now is to sort U'. We have at our disposal an *implicitly encoded memory* of $min(z,o)/\log^2|U|$ words of $\log|U|$ bits each but with the two limitations mentioned in Section 2 under implicit bit encoding. The third phase has the following three steps.

Step 1. We divide U' into $m = min(z,o)/\log^2|U|$ *segments* $V_1V_2\ldots V_m$ of $w = max(1,|U|/(qm))$ contiguous units each (and wq contiguous elements each). If $min(z,o) \geq \frac{|U|}{q}\log^2|U| = \Theta\left(|U|/\log^{\gamma-2}|S|\right)$, then each segment is composed of only one unit and we have at our disposal one encoded word for any segment. Since the units are homogeneous, we can proceed directly to the next phase.

Step 2. Otherwise, each segment contains more than one unit and we need to enlarge the size of the basic homogeneous objects in order to cope with the scarcity of encoded words. We consider each segment V_i starting from the leftmost one and do the following. We scan V_i to establish if it is homogeneous. During the scan, the classification of any unit U_j examined is done *by applying f to the first element of the ℓth packet of U_j*. If V_i is homogeneous, we continue with the next segment. If V_i is not homogeneous, we sort it recursively by invoking Sort_Units$(V_i, f, \ell+1)$.

Step 3. After the second step each segment is sorted. In this step we proceed to make them homogeneous (as we will see, many of them are already homogeneous and did not need to be recursively sorted in the second step). We start with V_1 and V_2. Scanning them, we find the subsequences $V_1^z, V_2^z, V_1^o, V_2^o$ such that $V_1 = V_1^zV_1^o$, $V_2 = V_2^zV_2^o$ and where V_1^z and V_2^z (V_1^o and V_2^o) contain the 0-units (1-units) of V_1 and V_2, respectively. During the scans the classification of any unit U_i considered is done *by applying f to the first element of the ℓth packet of U_i*. If $|V_1^z|+|V_2^z| \leq |V_1^o|+|V_2^o|$ ($|V_1^z|+|V_2^z| > |V_1^o|+|V_2^o|$), we perform $O(1)$ sequence exchanges to pass from $V_1V_2 = V_1^zV_1^oV_2^zV_2^o$ to $V_1'T_1 = V_1^oV_2^oV_1^zV_2^z$ ($V_1'T_1 = V_1^zV_2^zV_1^oV_2^o$). We do the same with T_1 and V_3, obtaining V_2' and T_2, then with T_2 and V_4 obtaining V_3' and T_3, and so forth.

Lemma 4. *If we exclude the recursive invocations possibly happening in the second step, the third phase of the execution of* Sort_Units(U, f, ℓ) *has the following properties:*

(i) It uses $O(\log |U|)$ extra bits and performs $O(|U|)$ element exchanges.
(ii) In the ideal-cache model, assuming $M = \Omega(B^2)$, it incurs $O(|U|/B)$ cache misses while performing the exchanges.
(iii) For any unit $U_i \in U'$, it applies f exclusively to the first element of $U_{i,\ell}$ (the ℓth packet of U_i) and only $O(1)$ times.

3.6 Fourth phase: permuting the homogeneous segments

After the third phase we have the sequence $\hat{Z}\hat{Q}^z\hat{Q}^o\hat{O}V'$. $\hat{Z}\hat{Q}^z\hat{Q}^o\hat{O}$ are still unvaried since the end of the second phase. V' is composed of $m \leq min(z, o)/\log^2 |U|$ homogeneous *segments* $V_1, V_2 \dots V_m$ of $w = max(1, |U|/(qm))$ contiguous units each (and wq contiguous elements each). Moreover, we have at our disposal one encoded word of length $\log |U|$ for any segment. The fourth phase has the following five steps.

Step 1. We logically divide V' into h *groups* $G_1 \dots G_h$ of $g = \sqrt{|V'|}/(wq)$ contiguous segments each (as always, let us suppose for simplicity that the last group has g segments too). We want to sort each group G_i, for $i = 1 \dots h-1$ using the segments in G_h as placeholders. We start with G_1. To record the original order of the segments of G_h, we "allocate" in the encoded memory an array P_1 of g encoded words. Then, we start scanning G_1 from left to right looking for 0-segments. As always, during the scan the classification of any unit U_t considered is done *by applying f to the first element of the ℓth packet of U_t*. We maintain two counters j' and j'' initially equal to 1. If the jth segment of G_1 is a 0-segment, we exchange it with the j''th segment of G_h, we set $P_1[j'] = j''$ and we increase both j' and j'' by one. Otherwise, we just increase j'. When we reach the right end of G_1, we do the symmetrical process for the 1-segments of G_1 (that is, scanning from right to left). Finally, we exchange the contents of G_1 and G_h. In the end, we have that (i) the segments originally in G_1 are back in it but in sorted order and (ii) the segments originally in G_h are back in it with the original order lost but encoded into P_1 (that is, the value of the ith encoded word of P_1 is the original position of the ith segment of G_h). After G_1, we sort G_2 in the following way. We "allocate" in the encoded memory a *new* array P_2 of g encoded words, without "deallocating" P_1. Then, we start scanning G_2 from left to right looking for 0-segments, analogously to what we did for G_1. Again, we use the two counters j' and j'' initially equal to 1. If the j'th segment of G_2 is a 0-segment, we exchange it with the j''th segment of G_h, we set $P_2[j'] = P_1[j'']$ and we increase both j' and j'' by one. Otherwise, we just increase j'. Then, we do the symmetrical process for the 1-segments of G_2 and we exchange the contents of G_2 and G_h. In the end, we have that (i) G_2 sorted and (ii) the segments originally in G_h are back in it furtherly scrambled but with the original order now encoded into P_2. We continue this process for $G_3, G_4 \dots$ and so forth, each time "allocating" new encoded arrays $P_3, P_4 \dots$ to preserve the memory of the original order of the segments in G_h. (It is easy to understand that if we used the same encoded array for any step of the process, we could not guarantee $O(1)$ applications

of f for any input element.) It is important to point out that during the entire process *we do not need to apply f to any of the elements of G_h*.

Step 2. At the end of the first step $G_1, G_2 \dots G_{h-1}$ are sorted and the original order of the segments of G_h is stored in the last encoded array produced in the process, that is P_{h-1}. We recover the original order of G_h in the following way. We decode the value encoded in the first entry of P_{h-1}, let it be i_1. Then, we exchange the first segment of G_h with the i_1th one. After that we decode the value in the i_1th entry of P_{h-1}, let it be i_2, and we encode a zero in the i_1th entry. Then, we exchange the first segment of G_h (that was the i_1th one before the first exchange) with the i_2th one. We continue in that way, until we eventually find the segment whose original position was the first one in G_h. After that, we scan G_h until we find a segment whose corresponding entry in P_{h-1} is not zero. At that point, we apply the same cyclic process we used before. When the scan reaches the right end of G_h, the original order of the segments is restored. Let us point out that during this entire process *we do not need to apply f to any of the elements of G_h* but only to the elements of the pairs that encode the array P_{h-1}.

Step 3. At the end of the second step the original order of G_h is restored and now we are ready to sort it. We "allocate" an encoded array R of g words. We compute and encode in R the ranks of the segments in G_h (that is, the positions they would have in G_h, if their group was sorted) in the following way. We first compute the number of 0-segments in G_h, let it be s_0, by a usual scan where the classification of any unit U_i considered is done *by applying f to the first element of the ℓth packet of U_i*. Then we find the ranks in the normal way. We maintain two indices i_0, i_1 initially set to 1 and $s_0 + 1$, respectively. Scanning again G_h from left to right (once again, the classification of any unit U_i considered is done *by applying f to the first element of the ℓth packet of U_i*), if the ith segment encountered is a 0-segment (1-segment) we encode i_0 (i_1) into the ith word of R and we increase i_0 (i_1). Finally, having the ranks encoded in R, we can now sort G_h with the same approach used in the second step to recover the original order of the segments.

Step 4. Now that all the groups are sorted they can be made homogeneous very easily with the same approach we used in the third step of the third phase to make the segments homogeneous.

Step 5. After the first four steps, V' has been transformed into a sequence of $h = |V'|/g = \Theta\left(\sqrt{|U|}\right)$ homogeneous groups of $g = \sqrt{|V'|}/(wq)$ contiguous segments each (and $\Theta\left(\sqrt{|U|}\right)$ contiguous elements each). Therefore, we can now sort them easily using the same approach used in the third step.

Lemma 5. *The fourth phase of the execution of* SORT_UNITS(U, f, ℓ) *has the following properties:*

(i) It uses $O(\log |U|)$ extra bits and performs $O(|U|)$ element exchanges.

(ii) In the ideal-cache model, assuming $M = \Omega(B^2)$, it incurs $O(|U|/B)$ cache misses while performing the exchanges.

(iii) For any element v in $\hat{Z} \cup \hat{O}$, it applies f to v $O(1)$ times.
(iv) For any unit $U_i \in V'$, it applies f exclusively to the first element of $U_{i,\ell}$ (the ℓth packet of U_i) and only $O(1)$ times.

3.7 Fifth phase: cleaning up

After the previous phase we have the sequence $\hat{Z}\hat{Q}^z\hat{Q}^o\hat{O}V''$. Subsequence V'' is sorted. Subsequences $\hat{Q}^z$ and $\hat{Q}^o$ are unvaried since the end of the second phase. On the other hand, pairs of elements from $\hat{Z}$ and $\hat{O}$ may have been exchanged during the fourth phase. The fifth phase has the following two steps.

Step 1. We recover the original order (holding at the end of the second phase) of the misplaced elements in $\hat{Z}$ and $\hat{O}$. This can be done very easily scanning $\hat{Z}$ and $\hat{O}$.

Step 2. We need to move $\hat{Z}$, $\hat{Q}^z$, $\hat{Q}^o$ and $\hat{O}$ in their right positions in V''. In the second step of the second phase we stored in $O(\log |U|)$ extra bits the following information: (i) the ranks in U of the leftmost 0 and the leftmost 1 (if any) of Q^z $(= \hat{Z}\hat{Q}^z)$; (ii) the ranks in U of the leftmost 0 (if any) and the leftmost 1 of Q^o $(= \hat{Q}^o\hat{O})$. It is easy to see that this stored information allows us to place $\hat{Z}$, $\hat{Q}^z$, $\hat{Q}^o$ and $\hat{O}$ in their right position in V'' with $O(1)$ sequence exchanges. It is worth to point out that in this second step *we do not need to apply the function f.*

Lemma 6. *The fifth phase of the execution of* Sort_Units(U, f, ℓ) *has the following properties:*

(i) It uses $O(\log |U|)$ extra bits and performs $O(|U|)$ element exchanges.
(ii) In the ideal-cache model, assuming $M = \Omega(B^2)$, it incurs $O(|U|/B)$ cache misses while performing the exchanges.
(iii) It applies f exclusively to the elements in $\hat{Z}\hat{Q}^z\hat{Q}^o\hat{O}$ and $O(1)$ times for each of them.

4. The proof of Theorem 1

We are finally able to prove our main statements.

Let us start considering the space complexity of our algorithm. From Lemma 1, we know that Create_Units(S, f) uses $O(\log |S|)$ extra bits. Since it is invoked only once, we can conclude that Create_Units(S, f) contributes to the space complexity of the entire algorithm with $O(\log |S|)$ extra bits. If *we exclude the recursive calls possibly happening in the second step of the third phase of* Sort_Units, from Lemmas 2, 3, 4, 5, and 6 we know that the five phases of Sort_Units(U, f, ℓ) use $O(\log |U|)$ extra bits. The size of any recursive sub-problem (a segment) solved in the second step of the third phase of Sort_Units(U, f, ℓ) is $q \cdot max(1, |U|/(qm))$ where $m = min(z, o)/\log^2 |U|$ and $q = \log^\gamma |S|$. From the first step of the first phase of Sort_Units(U, f, ℓ) we know that the computation can continue

with the phases following the first one only if $min(z, o) \geq \sqrt{|U|}$. Therefore the *size of any recursive sub-problem solved during the execution of* SORT_UNITS(U, f, ℓ) *is* less than or equal to $\sqrt{|U|}$ and the space complexity $space(|U|)$ of SORT_UNITS(U, f, ℓ) satisfies

$$space(|U|) \leq space\left(\sqrt{|U|}\right) + O(\log |U|)$$

and we can conclude that the space complexity of SORT_UNITS(U, f, ℓ) is $O(\log |U|)$ bits.

Let us consider the number of element exchanges performed by our algorithm. From Lemma 1, we know that CREATE_UNITS(S, f), which is invoked only once, performs $O(|S|)$ element exchanges. *Excluding the recursive calls possibly happening in the second step of the third phase of* SORT_UNITS, from Lemmas 2, 3, 4, 5 and 6 we know that the five phases of SORT_UNITS(U, f, ℓ) performs $O(|U|)$ element exchanges. We know that in the second step of the third phase *only the non-homogeneous segments* are sorted recursively. Let $\hat{m}$ be the number of non-homogeneous segments. Hence, the number of element exchanges $exc(|U|)$ performed by SORT_UNITS(U, f, ℓ) satisfies $exc(|U|) \leq \hat{m} \cdot exc(w \cdot q) + O(|U|)$, where w and q are the number of units in a segment and the number of elements in a unit, respectively. We know that $q = \log^{\gamma} |S|$ and $w = max(1, |U|/(qm))$, where $m = min(z, o)/\log^2 |U|$. Moreover, we know that the second step of the third phase is executed only if $min(z, o) < \frac{|U|}{q} \log^2 |U|$. Therefore $w \cdot q = |U|/m$ and, substituting m, we have that $w \cdot q = (|U| \log^2 |U|)/min(z, o)$ and, finally, $min(z, o) = (|U| \log^2 |U|)/(w \cdot q)$. Since the size of the smallest contiguous subsequence of elements of the same type is a multiple of the unit size q, it is clear that $\hat{m}$ is less than or equal to the number of units containing elements of the minority type, that is $\hat{m} \leq min(z, o)/q$. Substituting $min(z, o)$, we have that $\hat{m} \leq (|U| \log^2 |U|)/(w \cdot q^2)$. Therefore, the number of element exchanges $exc(|U|)$ performed by SORT_UNITS(U, f, ℓ) satisfies

$$exc(|U|) \leq \frac{|U| \log^2 |U|}{w \cdot q^2} exc(w \cdot q) + O(|U|).$$

Assuming by induction that $exc(w \cdot q) = O(w \cdot q)$ and knowing that $q = \log^{\gamma} |S|$ with $\gamma \geq 3$, we can conclude that the right side of the above recurrence is dominated by the second term (that is $O(|U|)$).

Essentially, the proof for the cache complexity of our algorithm is a simple variation of the proof for the number of element exchanges performed. We just have to pay attention to the cases requiring the application of the tall-cache assumption (i.e. $M = \Omega(B^2)$). We leave that proof for the full version of this paper.

Finally, let us evaluate the number of applications of the characteristic function f to any generic input element. From Lemma 1, we know that during the execution of CREATE_UNITS(S, f), for any v in S, f is applied to v $O(1)$ times. Let us consider an arbitrary invocation of SORT_UNITS(U, f, ℓ).

During the execution of SORT_UNITS(U, f, ℓ), the units in U can be logically partitioned into two sets, the set of *encoding units* and the set of *canonical units*. If the second step of the first phase is executed (that is, if the elements of the minority type are less than $\sqrt{|U|}$), all the units in U are *canonical units*. Otherwise, if we reach the second phase, the units in the zones Q^z and Q^o found in the first step of the phase (that is, the zones with the largest number of 0s and 1s, respectively) form the set of *encoding units* of SORT_UNITS(U, f, ℓ) (and the remaining units are the canonical ones). From Lemmas 2 (iii), 3 (iv), 4 (iii) and 5 (iv), we know that, during the execution of SORT_UNITS(U, f, ℓ), *excluding any possible recursive call in the third phase*, for any *canonical* unit U_i, f is applied exclusively to the first element of $U_{i,\ell}$ (the ℓth packet of U_i) and $O(1)$ times only. Since for any recursive execution of SORT_UNITS the input parameter ℓ is equal to the depth of the recursion, it is clear that a canonical unit is probed in one different packet for any nested recursive call it goes through. As we noticed before, the size of any recursive sub-problem solved during the execution of SORT_UNITS (U, f, ℓ) is less than or equal to $\sqrt{|U|}$. Therefore a unit can go through at most $\log\log|U|$ nested recursive executions of SORT_UNITS. Since the size of a unit is $\log^\gamma|S|$, with $\gamma \geq 3$, any unit has enough packets to go through the maximum number of nested recursive executions of SORT_UNITS. Let us consider now the encoding units. From Lemmas 3 (iii), 5 (iii) and 6 (iii), we know that *all the elements* of an encoding unit of SORT_UNITS(U, f, ℓ) may be probed with f. However, it is clear that a unit U_i can be an encoding unit only of the deepest one among the nested recursive execution of SORT_UNITS in which U_i is involved.

5. Conclusions

In this paper we improved the earlier linear-cost algorithm by Katajainen and Pasanen [6] for 0/1-sorting in two respects. Our algorithm is neither aware of the cache parameters B and M nor the implementation of the characteristic function f. Hence, the algorithm could be used in a generic environment where good performance is expected for different types of caches, input elements, and characteristic functions. The algorithm by Katajainen and Pasanen is complicated and not competitive in practice when compared to algorithms using more extra space. The new algorithm is even more complicated, so our main contribution is that the 0/1-sorting problem can be solved in an asymptotically optimal manner in a generic setting. In contemporary program libraries (e.g. SGI STL [11]) 0/1-sorting is accomplished by relying on non-optimal algorithms so a simplification of any of the linear-cost algorithms would be of practical relevance.

Up to now, we have assumed that element exchanges are equally expensive irrespective to the elements involved. However, in a generic environment the amount of work involved in element moves may vary, so it would be interesting to know whether there exists a 0/1-sorting algorithm that is

oblivious of the costs of element moves such that each element is only moved $O(1)$ times during the whole computation. We do not know how to give this guarantee if, at the same time, the algorithm is only allowed to perform a linear amount of work.

References

[1] J. Bojesen and J. Katajainen, Interchanging two segments of an array in a hierarchical memory system, *Proceedings of the 4th International Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **1982**, Springer-Verlag (2000), 159–170.

[2] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).

[3] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley (2000).

[4] G. Franceschini, On the computational power of permutations in sorting and searching problems, Ph. D. Thesis, Department of Computer Science, University of Pisa (2005).

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandra, Cache-oblivious algorithms, *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, IEEE (1999), 285– 297.

[6] J. Katajainen and T. Pasanen, Stable minimum space partitioning in linear time, *BIT* **32** (1992), 580–585.

[7] D. E. Knuth, *Fundamental Algorithms*, *The Art of Computer Programming* **1**, 3rd Edition, Addison Wesley Longman (1997).

[8] J. Munro, V. Raman, and J. Salowe, Stable in situ sorting and minimum data movement, *BIT* **30** (1990), 220–234.

[9] 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.

[10] H. Prokop, Cache-oblivious algorithms, M. Sc. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (1999).

[11] Silicon Graphics, Inc., Standard template library programmer's guide, Website accessible at `http://www.sgi.com/tech/stl/` (1993–2004).