# Selection and Sorting in the "Restore" Model

Timothy M. Chan* J. Ian Munro* Venkatesh Raman†

**Abstract**

We consider the classical selection and sorting problems in a model where the initial permutation of the input has to be *restored* after completing the computation. While the requirement of the restoration is stringent compared to the classical versions of the problems, this model is more relaxed than a read-only memory where the input elements are not allowed to be moved within the input array.

We first show that for a sequence of $n$ integers, selection (finding the median or more generally the $k$-th smallest element for a given $k$) can be done in $O(n)$ time using $O(\lg n)$ words[1] of extra space in this model. In contrast, no linear-time selection algorithm is known which uses polylogarithmic space in the read-only memory model.

For sorting $n$ integers in this model, we first present an $O(n \lg n)$-time algorithm using $O(\lg n)$ words of extra space. When the universe size $U$ is polynomial in $n$, we give a faster $O(n)$-time algorithm (analogous to radix sort) which uses $O(n^{\varepsilon})$ words of extra space for an arbitrarily small constant $\varepsilon > 0$. More generally, we show how to match the time bound of any word-RAM integer-sorting algorithms using $O(n^{\varepsilon})$ words of extra space. In sharp contrast, there is an $\Omega(n^2/S)$-time lower bound for integer sorting using $O(S)$ bits of space in the read-only memory model. Extension of our results to arbitrary input types beyond integers is not possible: for "indivisible" input elements, we can prove the same $\Omega(n^2/S)$ lower bound for sorting in our model.

En route, we develop linear-time in-place algorithms to extract leading bits of the input array and to compress and decompress strings with low entropy; these techniques may be of independent interest.

## 1 Introduction

This paper is concerned with *space-efficient* algorithms that require little (sublinear) extra space besides the input array. Two classes of such algorithms have received considerable attention in the past:

- *In-place* algorithms may use the input array as working space and may modify the array during computation. The output may be put in the same array by the end of the computation, or sent to an output stream. The prototypical example of an in-place algorithm is the classic heapsort, which requires just $O(1)$ words of extra space.[2] More sophisticated in-place sorting algorithms have been explored in the literature. For example, there is an in-place version of radix sort which can sort integers in the range $[U] = \{0, \ldots, U-1\}$ for $U = n^{O(1)}$ in linear time with $O(1)$ words of extra space [18]; and there is an in-place version of any word-RAM integer-sorting algorithm [1, 21, 22] which uses $O(1)$ words of extra space [18]. The standard linear-time algorithms for the classical selection problem, i.e., finding the median or the $k$-th smallest element for a given $k$, can also be made in-place with $O(1)$ words of extra space [24].

  Many in-place algorithms have been devised in different areas such as stringology (e.g., see [17]) and computational geometry (e.g., see [7, 10]). Often low (polylogarithmic) space usage can be guaranteed without much increase in the running time. However, one main disadvantage is that typically at the end of the computation, the original input permutation is lost—this is problematic in certain applications.

- In the *read-only memory* model, algorithms are not allowed to modify the input array at all and may make changes only in the extra storage area. (Single- or multiple-pass *streaming* algorithms fit in this model, although we allow more generally for random access to the read-only input array.) The selection problem has been studied in this model since the early work of Munro and Paterson [25] and Frederickson [19]; many further results were found (see Table 1), but none of the existing deterministic or randomized selection algorithms achieves linear time when polylogarithmic space is desired.

  Read-only-memory algorithms have recently gained more attention in areas such as computational geometry (e.g., see [3, 9]). Historically, read-only memory is in fact one of the more common settings studied from the perspective of time–space tradeoff lower

*Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {tmchan,imunro}@uwaterloo.ca

†The Institute of Mathematical Sciences, Chennai 600 113, India, vraman@imsc.res.in

[1] We use lg to denote logarithm; the base is 2 unless specified otherwise.

[2] A stricter definition of in-place algorithms explicitly requires that the amount of extra space is $O(1)$ words (and that the array can only store a permutation of the input elements at any time), but we will work with a looser definition (e.g., allowing for possibly polylogarithmic extra space) in this paper.

| | | |
|---|---|---|
| $O(n \lg_s n + n \lg^* s)$ | for $s \geq \lg^2 n$ | [25, 19] |
| $O(n)$ | for $s \geq n/\lg n$ | [15] |
| $O(sn^{1+1/s} \lg n)$ | for $s \leq \lg n$ | [29] |
| $O(n \lg \lg_s n)$ randomized | for any $s$ | [26, 8] |
| $O(n \lg_s U)$ | for any $s$ | [11] |
| $O(n \lg n \lceil \lg_s \lg U \rceil)$ | for any $s$ | [11] |

Table 1: Time bounds of selection algorithms in read-only memory using $O(s)$ words of space. All bounds are deterministic unless indicated otherwise; the last two are for integer input.

bounds. Work of Borodin et al. in the early 1980s [5, 6], and a subsequent improvement by Beame [4], investigated the sorting problem in read-only memory; the latter proved that any sorting algorithm using $O(S)$ bits of extra space requires $\Omega(n^2/S)$ time, even when the input consists of integers in a universe of size $U = O(n)$. (This bound was matched by a comparison-based sorting algorithm of Pagter and Rauhe [28] for all $\lg n \leq S \leq n/\lg n$; the range was further extended by the integer sorting algorithm of Pagh and Pagter [27].) Unfortunately, this lower bound indicates that the read-only memory model may incur a far greater loss of efficiency—for example, with polylogarithmic or $n^\varepsilon$ space, sorting requires near-quadratic time!

Still, efficient read-only-memory algorithms in general are desirable for at least two reasons: first, certain applications may require the input to not be destroyed; second, the input may actually be stored in a medium that is physically read-only.

In this paper, we investigate a natural relaxation (and, we feel, a fundamental variant) of the read-only memory model. In this new model, we allow algorithms to modify the input array during the computation, but require that the original input permutation be restored by the end of the computation (thus retaining at least one of the advantages of the read-only memory model, namely, that the input is not destroyed). We call this the RESTORE model. Note that the naive solution of copying the entire array is inadequate, as we are interested in algorithms with sublinear extra space.

As one motivation, algorithms in the RESTORE model may potentially be useful in the design of in-place algorithms, when sometimes one encounters subproblems which have to be solved by subroutines. It is important that these subroutines leave the array in its original state by the time they finish, so that computation can be properly resumed. Such subroutines may even be run in succession. (For one concrete example, Crochemore et al. [14] recently designed an in-place algorithm for the inverse Burrows-Wheeler transform which required selection as a subroutine; they invoked a selection algorithm in read-only memory, but a selection algorithm in the RESTORE model would be good enough for such purposes.)

Despite the naturalness of the model, it is unclear if one can actually get better results in the RESTORE model for standard comparison-based problems. Intuitively, at any moment during the computation, the array has retained the same amount of information, i.e., the same entropy, as the original input, in order for restoration to be possible; but this constraint appears very restrictive. Formally, we prove in Section 6 that for the sorting problem, the same $\Omega(n^2/S)$-time lower bound in read-only memory carries over in the RESTORE model, if the input elements are "indivisible" or "unsplittable", i.e., each array entry can only store an original element of the input set. Our proof is based on a simple encoding or Kolmogorov complexity argument.

However, for integer input in $[U]$ without the indivisibility assumption, we show that significantly better algorithms are possible in the RESTORE model: specifically, in Sections 2–5, we give

1. a selection algorithm that runs in $O(n)$ time and uses $O(\lg n)$ words of extra space;
2. a sorting algorithm that runs in $O(n \lg n)$ time and uses $O(\lg n)$ words of extra space;
3. a sorting algorithm that runs in $O(n\lceil \lg U/\lg n \rceil)$ time and uses $O(n^\varepsilon)$ words of extra space for an arbitrarily small constant $\varepsilon > 0$—the running time thus matches that of radix sort;
4. a sorting algorithm that runs in $O(\text{RAM-SORT}(n))$ time and uses $O(n^\varepsilon)$ words of extra space, where $\text{RAM-SORT}(n)$ denotes the time bound for integer sorting on the standard word RAM—current best results have
   - $\text{RAM-SORT}(n) = O(n\sqrt{\lg \lg n})$ with randomization, by Han and Thorup [22];
   - $\text{RAM-SORT}(n) = O(n \lg \lg n)$ without randomization, by Han [21];
   - $\text{RAM-SORT}(n) = O(n)$ with randomization in the case when $\lg U \geq (\lg n)^{2+\Omega(1)}$, by Andersson et al. [1].

What matters is not so much that the elements are integers but that they have bounded precision (a realistic assumption), since for comparison-based problems, we can map floating point numbers to integers by concatenating the exponents and mantissa. We assume a standard word RAM where the word size $w$ is equal to $\lg U$

(so that an input element fits in a word), with $U \geq n$, and standard (arithmetic, shift, and bitwise-logical) operations on words take constant time.

Our sorting algorithms send the elements in sorted order to an output stream, as do previous sorting algorithms in the read-only memory model. They immediately imply similar results for the element distinctness problem, i.e., deciding whether all elements are distinct, if one prefers a problem where the output interface is not a concern. In fact, we can do slightly better for this particular problem: in the RESTORE model, we give

5. an algorithm for element distinctness that runs in $O(n)$ time with randomization and uses $O(n^\varepsilon)$ words of extra space.

**Prior related work.** Our work is inspired by an open question from a recent talk by Roberto Grossi [20], who asked why linear-time selection has to be "destructive" to the input. We have since come across at least two papers that specifically looked at problems in the RESTORE model:

- Prior to Grossi's talk, Claude, Nicholson and Seco [13] had already defined the term *non-destructive* to refer to algorithms in the RESTORE model. They studied the wavelet tree construction problem and gave a non-destructive algorithm; however, the amount of extra space used is close to linear, on top of the space to store the output wavelet tree. Nonetheless, some of the issues encountered in computing wavelet trees turn out be relevant to our algorithms, as we will see later. As motivation for non-destructiveness, Claude et al. also mentioned an application to "a library for succinct data structures such as LIBCDS[3] where the user might want to further process the sequence used to build the wavelet tree". A predecessor to Claude et al.'s paper is Tischler's [30]; he mentioned a similar concept of "reversibility", and observed that with his method it is possible to transform a wavelet tree back to the original input string space-efficiently.

- Much earlier, in 1994, Katajainen and Pasanen [23] had already proposed the selection problem in the RESTORE model (which they explicitly called the *restoring selection problem*) en route to their in-place, stable, adaptive, multiset sorting algorithm. They gave an $O(n)$-time algorithm using $O(n)$ bits for the selection problem in this model, although an $O(n)$-time algorithm using $O(n)$ bits is now known in the less powerful read-only memory model [15]. Their paper left "as an open problem whether there exists a minimum space algorithm for restoring selection".

[3] http://libcds.recoded.cl

**Outline of our approach.** The techniques behind our algorithms are not complicated but we believe are interesting. The first idea is to use the leading bit of each input number to partition the array and then apply recursion. We observe that the standard partitioning algorithm from quicksort [12] is in fact *reversible*, which allows us to restore the input after recursion. The partitioning based on leading bits may not be balanced, however. Our key insight is that in the unbalanced case, the input would not be uniformly distributed and would thus have less entropy and be *compressible*. Compression can save a significant number of bits of space in the input array—enough for us to switch to a direct read-only-memory algorithm. We can then decompress to restore the input.[4]

To carry out this plan, we need the following subroutines, which may be of independent interest:

- a linear-time in-place algorithm for extracting leading bits of the input array (naively invoking a known in-place permutation result [16] would require $O(n \lg n)$ time);

- a linear-time in-place algorithm for compressing and decompressing a string (the trick is simple, but we are unaware of work on this subproblem in the in-place algorithms literature).

In addition, in order to achieve our best results for the sorting problem, we need to perform a multi-way rather than binary partitioning, which creates more challenges. We hope that our ideas will find further applications.

## 2 Selection and Sorting in $O(n \lg U)$ Time

In this and the next three sections, we will assume that the input is a sequence of $n$ integers in $[U]$ where $U$ is a power of 2. We begin this section by giving a simple, easy-to-implement $O(n \lg U)$-time algorithm for selection and sorting in the RESTORE model using $O(\lg n)$ words of space. For selection, the result is not new, since an $O(n \lg U)$-time algorithm using $O(1)$ words is already known in read-only memory [11], but we show how to refine our approach to obtain better results later.

[4] At least one prior paper by Franceschini et al. [18] on in-place integer sorting has also used the idea of compression and decompression, by exploiting the fact that the entropy is decreased after the array is sorted; however, our situation is very different from the in-place setting, since we need the array to preserve entropy at all times, to enable restoration.

THEOREM 2.1. *Given a sequence of $n$ integers in $[U]$ and given $k$, we can find the $k$-th smallest element of the sequence in $O(n \lg U)$ time using $O(\lg n)$ words of extra space in the* RESTORE *model. Furthermore, we can output the elements in sorted order in $O(n \lg U)$ time using $O(\lg n)$ words of extra space in the* RESTORE *model.*

*Proof.* The following selection algorithm achieves the claimed bounds.

**Step 1** Our first step is to partition the given array into two by pivoting on $U/2$, i.e., we move the elements with the most significant bit 0 to the first part and those with the most significant bit 1 to the second part. For this, we adopt the textbook partitioning algorithm used by quicksort: Keep two pointers at either end of the array. Scan with each pointer until we find an element that should go to the other half. Swap the two elements pointed to by the two pointers, until we come to the middle of the array. When we swap, we move all but the most significant bits of each element. The most significant bits of the elements remain in their original positions to help with the restoration process. (Note that the partitioning is not required to be "stable".)

**Step 2** Depending on the size of each part, recursively find the appropriately ranked element in the portion that contains the $k$-th smallest element and restore the part to its permutation (before the recursive step).

**Step 3** By using the most significant bits, reverse the moves made in Step 1, to restore to the original array. The reversal can be done by re-running the same partitioning algorithm.

For the sorting problem, we modify Step 2 of the algorithm to use recursion on both parts of the array instead.

Of course the position of the most significant bit advances as we descend during the recursion. When outputting an element, we should replace the bits to the left of the most significant bit position with the correct bits, which can be determined from the current branch of the recursion tree.

It is clear that the algorithm takes $O(n \lg U)$ time as the universe size $U$ comes down by a factor of two in each recursive call. The recursion stack needs $O(\lg U \lg n)$ bits to remember the positions of the subarrays. □

## 3 Selection in Linear Time and Sorting in $O(n \lg n)$ Time

Note that in the algorithm of Section 2, if the pivot at every step is close to the median of the sequence, which is what we would expect in a random instance, then the number of elements would decrease by a constant factor in each recursive call, and we could then improve the time bound for selection to $O(n)$ and sorting to $O(n \lg n)$. Unfortunately, we do not know how to pivot around a value other than $U/2$ efficiently in the RESTORE model.

However, when our pivot $U/2$ is far from the median element, we observe that the size of the parts are unbalanced, i.e., we have a lot more elements with leading bit 0 than with leading bit 1 or vice versa. In this case, the leading bits have less entropy, hence can be compressed to release some extra (close to $O(n)$) bits of space which can be used to terminate the recursion by switching to a known read-only memory algorithm.

Before compression, we first need to extract the most significant bits of the elements into a prefix of the array. This is handled by the following lemma (for now, we only need the special case of $\ell = 1$).

LEMMA 3.1. **(In-place extraction/un-extraction of the leading bits)** *Given a sequence of $n$ integers in $[2^w]$ and a number $\ell \leq w$, we can extract the $\ell$ most significant bits of each of them, put them in a prefix of the array, and put the remaining bits in a suffix of the array, while preserving the given order, in $O(n)$ time using $O(\ell \lg n)$ words of space. The extraction can be undone within the same bounds.*

*Proof.* Divide the sequence into blocks of $aw$ elements each for some integer parameter $a$. Do the following for each block: Extract the $\ell$ most significant bits of each element in the block to the extra storage area, which requires $aw\ell$ bits, i.e., $a\ell$ words. Now, move these extracted most significant bits in that order to the beginning of the block and shift the remaining bits of each element to the end of the block; this can easily be done in linear time.

At the end of this step, we have a sequence of the form $A_1, B_1, A_2, B_2, \ldots$ where each $A_i$ occupies $a\ell$ words and each $B_i$ occupies $a(w - \ell)$ words. We want to permute the array to get $A_1, A_2, \ldots, B_1, B_2, \ldots$ We can apply a general in-place permutation algorithm by Fich et al. [16] to solve this subproblem. This algorithm assumes oracles for both the permutation and its inverse, which are easily implementable in our case. It requires just $O(1)$ words of space; however, its running time is $O(n \lg n)$.

To speed up the process, the trick is to pick a nonconstant $a$ and view the sequence as a sequence of segments of $a$ words each. The permutation we seek is a permutation of the segments. We apply Fich et al.'s algorithm to the $n/a$ segments, which requires $O((n/a) \lg(n/a))$ time, which is $O(n)$ by choosing $a =$

$\lceil \lg n \rceil$. A move in this situation costs $O(a)$ as a move involves moving $a$ words. But as the number of moves in Fich et al.'s algorithm is linear in the number of segments $n/a$ (their algorithm uses the minimum number of moves since it decomposes the permutation into disjoint cycles), the overall cost remains $O(n)$.

Un-extraction can be done similarly, by reversing all the steps.

We remark that this bit extraction problem also arose in a previous work on space-efficient construction of wavelet trees by Claude et al. [13]; they also used Fich et al.'s algorithm but did not obtain a purely in-place, linear-time solution to our subproblem. □

Next, we describe how to do the compression step in a space-efficient manner:

LEMMA 3.2. **(In-place compression/decompression)** *Given a string $s = s_1 s_2 \cdots s_n$ with $s_i \in [2^\ell]$ and a suffix code[5] for the alphabet $[2^\ell]$, where the maximum character code length is $L$ with $\ell \le L \le w$, we can compute the encoding of $s$ in $O(n + 2^L)$ time using $O(2^L)$ words of extra space, provided that the code compresses $s$ (i.e., the encoding of $s$ is not longer than $s$). Decoding can also be done within the same bounds.*

*Proof.* Note that a linear-time encoding algorithm is straightforward, since we can scan the string from left to right and generate the code of each character by table lookup. However, this naive algorithm does not guarantee constant space, because the code is assumed to compress $s$, but not necessarily compress every prefix of $s$.

If we make an extra assumption that the encoding of $s$ saves a linear number of bits, one possible approach would be to use recursion (divide into two halves, recursively generate the code of the more compressible half, then directly compute the code of the remaining half using the linear number of bits saved). We describe a different, simple encoding algorithm that does not require any extra assumption:

**Step (i)** Let $c_i$ be $\ell$ minus the code length of $s_i$. Let $C_i = c_1 + \cdots + c_i$. First compute the index $i$ that minimizes $C_i$; this can be done by an obvious linear scan with $O(1)$ space.

**Step (ii)** Next run the naive algorithm to generate the code of the cyclically shifted string $s_{i+1} \cdots s_n s_1 \cdots s_i$, starting at position $i + 1$ of the array.

**Step (iii)** Finally cyclically shift the encoded string to lie in a prefix of the array.

To see why the naive algorithm is applicable to the cyclically shifted string in Step (ii), we note that for every $j > i$, the code compresses $s_{i+1} \cdots s_j$ since the number of bits saved is $C_j - C_i \ge 0$, and moreover for every $j \le i$, the code compresses $s_{i+1} \cdots s_n s_1 \cdots s_j$ since the number of bits saved is $C_n - C_i + C_j \ge C_n \ge 0$.

For the decoding algorithm, we assume that we are given the index $i$ and a pointer to the end of the encoded string, which take $O(1)$ words of space. We just reverse all the steps of the encoding algorithm. To reverse the execution of the naive algorithm in Step (ii), note that from the last $L$ bits of the encoded string, we can determine which character in $[2^\ell]$ maps to a suffix of the encoded string by table lookup (the precomputation of the table takes $O(2^L)$ time); we can then update the end pointer of the encoded string, and repeat. □

With these two lemmas, we can now obtain an $O(n)$-time selection algorithm and an $O(n \lg n)$-time sorting algorithm in the RESTORE model:

THEOREM 3.1. *Given a sequence of $n$ integers in $[U]$ and given $k$, we can find the $k$-th smallest element of the sequence in $O(n)$ time using $O(\lg n)$ words of extra space in the* RESTORE *model. Furthermore, we can output the elements in sorted order in $O(n \lg n)$ time using $O(\lg n)$ words of extra space in the* RESTORE *model.*

*Proof.* Let $\delta > 0$ be a sufficiently small constant. We modify the algorithm in Theorem 2.1. In Step 2, if the size of one of the two parts is more than $(1 - \delta)n$, we switch to the following method instead of recursion: We first extract and move the most significant bits of the elements to a prefix of the array in $O(n)$ time using $O(\lg n)$ words of space by Lemma 3.1. (Note that at an intermediate step of the recursion, the "most significant bit" may actually refer to some intermediate bit position; we can rearrange the bits of each element to make this position the most significant before invoking the lemma.) Say the number of 0s is more than $(1-\delta)n$ and the number of 1s is at most $\delta n$. In this case, we compress the string of the most significant bits in $O(n)$ time by Lemma 3.2, for example, by dividing into blocks of 2 and using a suffix code with $\ell = 2$ and $L = 3$ which maps 00 to 0 (shortening by one bit) and maps $y$ to $y1$ (lengthening by one bit) for each $y \in \{01, 10, 11\}$. Let $n_{00}$ be the number of occurrences of 00; then $n_{00} \ge (1/2-\delta)n$. With this suffix code, the net decrease

[5] Suffix code is similar to prefix code. The code of a string is the concatenation of the code of its characters. To guarantee a unique decoding, we require that no character's code is a suffix of another character's code.

in the number of bits is at least $n_{00} - \delta n \geq \alpha n$ for $\alpha = 1/2 - 2\delta$.

With the extra space saved by the compression, we can now solve the problem directly by invoking a known selection algorithm in read-only memory with $O(n)$ time and $\alpha n$ bits of space [15], or a known sorting algorithm in read-only memory with $O(n \lg n)$ time and $\alpha n$ bits of space [19]. (Pagter and Rauhe [28] (see also [2]) have given a more complicated sorting algorithm which improves space by a logarithmic factor to $O(n/\lg n)$ bits, but we do not need this improvement here.) Note that after bit extraction, accessing an array element may require some extra overhead in address calculations but still takes constant time. (Also note that if the "most significant bit" is actually an intermediate bit position, an access to an element by the read-only memory algorithm should retrieve only the bits to the right of that position.) Finally, we decompress the string by Lemma 3.2, and undo the bit extraction by Lemma 3.1.

Since recursion is done only when both parts are of size at most $(1-\delta)n$, the number of elements drops by a constant fraction of $n$, resulting in a recursion depth of $O(\lg n)$. The linear time bound for selection follows from a geometric series. □

## 4 Matching Radix Sort

In the case of integers, there are sorting algorithms with running time better than $O(n \lg n)$. For example, radix sort runs in linear time for $U = n^{O(1)}$. We will show that similar algorithms are possible in the RESTORE model.

We begin with a faster version of the $O(n \lg U)$-time algorithm in Section 2. The idea is to generalize binary partitioning to $b$-way partitioning.

THEOREM 4.1. *Given a sequence of $n$ integers in $[U]$ and a parameter $b \leq \min\{n, U\}$, we can output the elements in sorted order in $O(n \lg_b U)$ time using $O(b^{O(1)} \lg n)$ words of extra space in the* RESTORE *model.*

*Proof.* We may assume that $b$ is a power of 2. Furthermore, we may assume that $n \geq b^3$, because otherwise we can directly run radix sort [12], which takes $O(n \lg_n U) \leq O(n \lg_b U)$ time (by the $n \geq b$ condition) with $O(b^3)$ words of space.

We generalize Step 1 of the algorithm in Theorem 2.1 to a $b$-way partitioning, based on the $\lg b$ most significant bits of the integers. We want to move elements so that at the end of the partitioning step, the $i$-th part of the array contains all elements with their $\lg b$ most significant bits representing the integer $i \in [b]$. When we do the partitioning, we move only the $\lg U - \lg b$ least significant bits of each element. We initially find the sizes of the $b$ parts by a linear scan and virtually divide the array into $b$ parts. We maintain a table of $b$ pointers $p_1, \ldots, p_b$, where $p_i$ points to an element $A[p_i]$ currently residing in the $i$-th part that needs to be moved to another part (as indicated by the element's $\lg b$ most significant bits). Pointers always advance from left to right and become null when their parts are exhausted. At each round, we begin with the non-null pointer $p_i$ with the smallest $i$ (it is important to keep each step deterministic, so that we know how to reverse the partitioning algorithm later). If $A[p_i]$ needs to be moved to the $j$-th part, we move $A[p_i]$ to $A[p_j]$, advance pointer $p_j$, and repeatedly move $A[p_j]$, until we return to $A[p_i]$ to complete a cycle. We then advance pointer $p_i$ and proceed with the next round. The total cost of the partitioning step is $O(n)$ (setting up the table of pointers requires $O(b)$ time, but $n \geq b$).

In Step 2, we recursively sort the $b$ parts. For a part of size less than $b$, we technically cannot use recursion, but we can directly sort in $O(b \lg b)$ time with $O(b)$ words of space. The total extra cost $O(b^2 \lg b)$ can be absorbed by $O(n)$, since $n \geq b^3$.

In Step 3, by using the $\lg b$ most significant bits, we can re-simulate the $b$-way partitioning algorithm and reverse the moves made, following the same cycle decomposition.

The recursion depth is reduced to $O(\lg_b U)$, so the total running time is $O(n \lg_b U)$ and the total space in bits is $O(b^{O(1)} \lg U + (\lg U/\lg b) \cdot b^{O(1)} \lg n)$. □

By setting $b = n^{\Theta(\varepsilon)}$, this theorem implies an algorithm with $O(n\lceil \lg U/\lg n \rceil)$ time and $O(n^\varepsilon)$ words of space in the RESTORE model. The running time matches that of radix sort (with base $n$), and is linear when $U = n^{O(1)}$. The algorithm is simple enough for implementation (no bit extraction or compression is necessary).

Alternatively, by setting $b = \lg^{\Theta(\varepsilon)} n$, the theorem implies an algorithm for $U = n^{O(1)}$ with $O(n \lg n/\lg\lg n)$ time and $O(\lg^{1+\varepsilon} n)$ words of space in the RESTORE model.

## 5 Matching Any RAM Sorting Algorithm

We now show that $o(n \lg n)$-time sorting algorithms in the RESTORE model are theoretically possible for any universe size $U$. In fact, we can match the time bound of any word-RAM integer sorting algorithm if we allow $O(n^\varepsilon)$ words of space.

The plan is to reduce the $O(n \lg_b U)$ running time in Section 4 to near $O(n \lg_b n)$, in the same manner that we have reduced the $O(n \lg U)$ running time in Section 2 to $O(n \lg n)$ in Section 3, by using extraction and compression of the leading bits, coupled with $b$-way

partitioning.

Below, we assume the function RAM-SORT$(\cdot)$ satisfies $\sum_i \text{RAM-SORT}(n_i) \le \text{RAM-SORT}(\sum_i n_i)$.

THEOREM 5.1. *Given a sequence of $n$ integers in $[U]$, a parameter $b \le n$, and any constant $\varepsilon > 0$, we can output the elements in sorted order in $O(n \lg_b n + \text{RAM-SORT}(n))$ time using $O(b^{O(1)} \lg n + \min\{n^\varepsilon, \lg^{O(\varepsilon)} U\})$ words of extra space in the* RESTORE *model. Furthermore, we can decide whether the elements are distinct in $O(n \lg_b n)$ expected time using $O(b^{O(1)} \lg n)$ words of extra space in the* RESTORE *model.*

*Proof.* We may assume that $b$ and $\lg b$ are powers of 2. We generalize Step 1 and Step 3 of the algorithm in Theorem 2.1 to a $b$-way partitioning, in exactly the same way described in the proof of Theorem 4.1.

In Step 2, we modify the approach in the proof of Theorem 3.1: We first extract the $\lg b$ most significant bits of each integer in $O(n)$ time using $O(\lg b \lg n)$ words of space by Lemma 3.1. We then compress the string of these most significant bits by Lemma 3.2. For each part with size at most $2^t n/b$ for some parameter $t$, we recursively sort the elements. (For a part of size $n_i$ less than $b$, we technically cannot use recursion, but we can directly sort in $O(\text{RAM-SORT}(n_i))$ time with $O(b)$ words of space.) For each part with size more than $2^t n/b$, we solve the problem directly by invoking a known read-only-memory algorithm with the extra space saved by compression. Finally, we decompress the string and undo the bit extraction.

For the compression, we can for example use the following suffix code[6] with $\ell = \lg b$ and $L = \lg b + 1$: Let $n_i$ denote the size of the $i$-th part. First observe that there are at most $b/2^t$ parts of size more than $2^t n/b$, and at least $b/2$ parts of size at most $2n/b$. Thus, it is possible to form disjoint groups of $2^{t-1}$ indices each, such that each *bad* index $i$ with $n_i > 2^t n/b$ is grouped with $2^{t-1} - 1$ *good* indices $j$ with $n_j \le 2n/b$. We permute the alphabet $[b]$ so that for each group, the bad index has binary representation of the form $0^{t-1}z$ and the good indices have binary representations of the form $yz$ with $|y| = t - 1$ and $y \ne 0^{t-1}$ for a common string $z$. We then map $0^{t-1}z$ to $0z$ (shortening by $t-2$ bits) and map $yz$ to $y1z$ (lengthening by one bit). With this suffix code, the net decrease in the number of bits is at least $\sum_{i \text{ bad}}[(t-2)n_i - (2^{t-1}-1)(2n/b)] \ge \sum_{i \text{ bad}}[(t-2)n_i - n_i] \ge \Omega(\lg b) \sum_{i \text{ bad}} n_i$ by setting $t = (\lg b)/2$.

[6] We can also use Huffman coding, which produces an optimal prefix/suffix code, but the argument in this paragraph is still needed to bound the number of bits saved.

For each bad index $i$, the number of bits saved by the compression is thus at least $\Omega(n_i \lg b)$. We invoke a sorting algorithm in read-only memory, as noted in the appendix, with $O(T + \text{RAM-SORT}(n_i))$ time and $O((n_i^2/T)\lg n_i + n_i^\varepsilon \lg U)$ bits of space for any given $T$. (Pagh and Pagter [27] have given a more complicated algorithm which improves space by almost a logarithmic factor to $O((n_i^2/T)\lg^{(T/n_i)} n_i + n_i^\varepsilon \lg U)$ bits,[7] but we do not need this improvement here.) We can set $T = \Theta(n_i \lg_b n_i)$ so that the first term of the space bound matches $O(n_i \lg b)$. The second term $O(n_i^\varepsilon \lg U)$ is absorbed by the first, unless $n_i^{1-\varepsilon} \le \lg U$, in which case the second term is bounded by $O(\min\{n^\varepsilon, \lg^{O(\varepsilon)} U\} \cdot \lg U)$.

For the element distinctness problem, we invoke instead an algorithm in read-only memory based on hashing, as noted in the appendix, with $O(T)$ expected time and $O((n_i^2/T)\lg n_i + \lg U)$ bits of space.

Since recursion is done only for parts of size at most $2^t n/b = n/\sqrt{b}$, the recursion depth is $O(\lg_b n)$. □

By setting $b = n^{\Theta(\varepsilon)}$, this theorem implies a sorting algorithm with $O(\text{RAM-SORT}(n))$ time and $O(n^\varepsilon)$ words of space, and an algorithm for element distinctness with $O(n)$ expected time and $O(n^\varepsilon)$ words of space, in the RESTORE model. For sorting, for a specific bound such as $\text{RAM-SORT}(n) = O(n\sqrt{\lg \lg n})$ by Han and Thorup [22], we can set $b = n^{\Theta(\varepsilon/\sqrt{\lg \lg n})}$ instead and obtain a slightly better expression $O(n^{\varepsilon/\sqrt{\lg \lg n}} + \lg^\varepsilon U)$ for the space usage.

Alternatively, by setting $b = \lg^{\Theta(\varepsilon)} n$, the theorem implies an algorithm for any $U$ with $O(n \lg n / \lg \lg n)$ time and $O(\lg^{1+\varepsilon} n + \lg^\varepsilon U)$ words of space in the RESTORE model.

## 6 Lower Bound for Indivisible Elements

We finally show that if the elements are indivisible, i.e., if the input array must be a permutation of the input sequence at any point of time, then the RESTORE model is not any more powerful than the read-only memory model for the sorting problem, i.e., there is a lower bound of $\Omega(n^2/S)$ time for sorting, matching the best known lower bound in read-only memory [4]. This justifies the need to manipulate bits of the input integers in all our algorithms in the RESTORE model.

Since the RESTORE model essentially forces the array to preserve entropy at all times, it is natural to consider an encoding or Kolmogorov complexity style argument, which we will use in the following lower bound proof.

[7] Here, $\lg^{(j)}$ denotes the slow-growing $j$-th iterated logarithm function.

THEOREM 6.1. *Any algorithm that, given a sequence of $n$ indivisible elements, outputs the elements in sorted order requires $\Omega(n^2/S)$ worst-case time in the* RESTORE *model, where $S \geq \lg n$ is the number of bits of extra space available.*

*Proof.* Suppose there is an algorithm that uses fewer than $n^2/(cS)$ steps on all input, where $c$ is a sufficiently large constant. We describe a way to encode an arbitrary sequence of $n$ integers in $[U]$ (with $U = \Omega(n)$) in less than $n \lg U$ bits, which would lead to a contradiction.

Consider the execution of the algorithm on the given integer sequence. Divide $[U]$ into $\lceil n/(2cS) \rceil$ intervals of length $\Omega(U \cdot cS/n)$. Divide the execution of the algorithm into phases where in the $i$-th phase, the elements ouputted by the algorithm are from the $i$-th interval. Then some phase $i$ must use fewer than $(n^2/(cS))/\lceil n/(2cS) \rceil \leq n/2$ steps.

Let $I$ be the interval for that phase $i$. We encode the given sequence as follows: First record the index $i$ and the state $\sigma$ of the algorithm at the beginning of phase $i$, in $O(S)$ bits. Let $x_1, \ldots, x_n$ be the sequence of elements in the order they are first accessed when starting at state $\sigma$. Since all elements in $I$ are ouputted (and thus accessed) during the first $n/2$ steps, all elements in $I$ are from the first half of this sequence. In other words, $x_{n/2+1}, \ldots, x_n \in [U] - I$. Record $x_1, \ldots, x_{n/2}$ as they are, in $(n/2) \lg U$ bits, and then record $x_{n/2+1}, \ldots, x_n$, in $\lceil \lg(|[U] - I|^{n/2}) \rceil |$ bits. Since the length of $I$ is $|I| = \Omega(U \cdot cS/n)$, the total number of bits in the encoding is

$$\begin{aligned}
&(n/2) \lg U + (n/2) \lg(U - |I|) + O(S) \\
&\quad = \quad n \lg U + (n/2) \lg(1 - |I|/U) + O(S) \\
&\quad \leq \quad n \lg U - \Omega(n|I|/U) + O(S) \\
&\quad \leq \quad n \lg U - \Omega(cS) + O(S) \;<\; n \lg U
\end{aligned}$$

for a sufficiently large constant $c$.

From this encoding, we can recover the original sequence as follows: simulate the algorithm from state $\sigma$ on an array of "unknowns". Whenever the algorithm accesses an element whose value is currently unknown, fill in its value by retrieving the next element from the sequence $x_1, \ldots, x_n$. (Note that the argument holds even if the algorithm is allowed to move elements in the array.) By definition of the RESTORE model, the content of the array at the end of the execution gives us exactly the input in its original permutation. □

Our proof extends to randomized algorithms: by Yao's principle, it suffices to consider a random, uniformly distributed input, but such an input is incompressible with high probability.

Our proof appears simpler than previous lower bound proofs for sorting in read-only memory [4, 5, 6], but the results are not directly comparable. For example, Beame's proof [4] holds without any indivisibility restriction and for the weaker problem of outputting the unique elements in arbitrary order (for which our proof would not work). Our basic observation is that for most input instances, a large number of steps must be made before an algorithm can encounter all elements lying in an interval of a certain length. This type of observation has appeared in lower bound proofs before (e.g., see [8, Lemma 4.1]), and it was typically shown via a counting or probabilistic argument. Our above proof essentially recasts this as an encoding argument.

## 7 Concluding Remarks

We conjecture that for indivisible elements, many existing lower bounds in read-only memory should carry over to the RESTORE model. For example, for the $\Omega(n \lg \lg_s n)$ randomized lower bound for selection [8], this might require rethinking the various probabilistic arguments used in that proof as encoding arguments.

Another open question is whether the $O(n^\varepsilon)$ space bound of our integer sorting algorithms from Sections 4 and 5 could be reduced to polylogarithmic, or even $O(1)$ words (as in-place algorithms are possible in the standard RAM model [18]), or if some lower bound could be proved to rule out this possibility.

We hope that our work will inspire further studies in the RESTORE model. For example, we are able to extend our approach to the problem of counting inversions in a sequence. This requires additional effort to ensure the partitioning step be done stably; details will be forthcoming. We can also investigate applications to problems from other areas, such as computational geometry, in the RESTORE model.

## References


[1] A. Andersson, T. Hagerup, S. Nilsson and R. Raman, 'Sorting in linear time?', *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

[2] T. Asano, A. Elmasry and J. Katajainen, 'Priority queues and sorting for read-only data', in *Proc. 10th International Conference on Theory and Applications of Models of Computation* (TAMC 2013), 32–41.

[3] T. Asano, W. Mulzer, G. Rote and Y. Wang, 'Constant-work-space algorithms for geometric problems', *Journal of Computational Geometry*, 2(1):46–68, 2011.

[4] P. Beame, 'A general sequential time-space tradeoff for finding unique elements', *SIAM Journal on Computing*, 20(2):270–277, 1991.

[5] A. Borodin and S. A. Cook, 'A time-space tradeoff for sorting on a general sequential model of computation', *SIAM Journal on Computing*, 11(2):287–297, 1982.

[6] A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch and M. Tompa, 'A time-space tradeoff for sorting on non-oblivious machines, *Journal of Computer and System Sciences*, 22(3):351–364, 1981.

[7] H. Brönnimann, T. M. Chan and E. Y. Chen, 'Towards in-place geometric algorithms and data structures', in *Proc. 20th ACM Symposium on Computational Geometry* (SoCG 2004), 239–246.

[8] T. M. Chan, 'Comparison-based time-space lower bounds for selection', *ACM Transactions on Algorithms*, 6(2):26, 2010.

[9] T. M. Chan and E. Y. Chen, 'Multi-pass geometric algorithms', *Discrete and Computational Geometry*, 37(1):79–102, 2007.

[10] T. M. Chan and E. Y. Chen, 'Optimal in-place and cache-oblivious algorithms for 3-d convex hulls and 2-d segment intersection', *Computational Geometry: Theory and Applications*, 43(8):636–646, 2010.

[11] T. M. Chan, J. I. Munro and V. Raman, 'Faster, space-efficient selection algorithms in read-only memory for integers', to appear in *Proc. 24th International Symposium on Algorithms and Computation* (ISAAC 2013).

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, 'Introduction to Algorithms', MIT Press, 3rd ed., 2009.

[13] F. Claude, P. K. Nicholson and D. Seco, 'Space efficient wavelet tree construction', in *Proc. 18th International Symposium on String Processing and Information Retrieval* (SPIRE 2011), 185–196.

[14] M. Crochemore, R. Grossi, J. Kärkkäinen and G. M. Landau, 'A constant-space comparison-based algorithm for computing the Burrows-Wheeler transform', in *Proc. 24th Symposium on Combinatorial Pattern Matching* (CPM 2013), 74–82.

[15] A. Elmasry, D. D. Juhl, J. Katajainen and S. Rao Satti, 'Selection from read-only memory with limited work space', in *Proc. 17th International Conference on Computing and Combinatorics* (COCOON 2013), 147–157.

[16] F. E. Fich, J. I. Munro and P. V. Poblete, 'Permuting in place', *SIAM Journal on Computing*, 24(2):266–278, 1995.

[17] G. Franceschini and S. Muthukrishnan, 'In-place suffix sorting', in *Proc. 34th International Colloquium on Automata, Languages, and Programming* (ICALP 2007), 533–545.

[18] G. Franceschini, S. Muthukrishnan and M. Pătraşcu, 'Radix sorting with no extra space', in *Proc. 15th European Symposium on Algorithms* (ESA 2007), 194–205.

[19] G. Frederickson, 'Upper bounds for time-space tradeoffs in sorting and selection', *Journal of Computer and System Sciences*, 34(1):19–26, 1987.

[20] R. Grossi, private communication, 2011; see slide 50 of `https://www.imsc.res.in/~dsmeet/grossi_dsmeet2011_1.pdf`.

[21] Y. Han, 'Deterministic sorting in $O(n \lg \lg n)$ and linear space', *Journal of Algorithms*, 50(1):96–105, 2004.

[22] Y. Han and M. Thorup, 'Integer sorting in $O(n\sqrt{\lg \lg n})$ expected time and linear space', in *Proc. 43rd IEEE Symposium on Foundations of Computer Science* (FOCS 2002), 135–144.

[23] J. Katajainen and T. Pasanen, 'Sorting multisets stably in minimum space', *Acta Informatica*, 31(4):301-313, 1994.

[24] T. W. Lai and D. Wood, 'Implicit selection', in *Proc. 1st Scandinavian Workshop on Algorithm Theory* (SWAT 1988), 14–23.

[25] J. I. Munro and M. Paterson, 'Selection and sorting with limited storage', *Theoretical Computer Science*, 12:315-323, 1980.

[26] J. I. Munro and V. Raman, 'Selection from read-only memory and sorting with optimum data movement', *Theoretical Computer Science*, 165(2):311-323, 1996.

[27] R. Pagh and J. Pagter, 'Optimal time-space trade-offs for non-comparison-based sorting', in *Proc. 13th ACM–SIAM Symposium on Discrete Algorithms* (SODA 2002), 9–18.

[28] J. Pagter and T. Rauhe, 'Optimal time-space tradeoffs for sorting', in *Proc. 39th IEEE Symposium on Foundations of Computer Science* (FOCS 1998), 264–268.

[29] V. Raman and S. Ramnath, 'Improved upper bounds for time-space tradeoffs for selection', *Nordic Journal of Computing*, 6(2):162–180, 1999.

[30] G. Tischler, 'On wavelet tree construction', in *Proc. 22nd Symposium on Combinatorial Pattern Matching* (CPM 2011), 208–218.

## A Appendix

The following lemma was used in the proof of Theorem 5.1. The sorting part was known: Pagh and Pagter [27] have in fact reduced the $\lg n$ factor in the space bound to an iterated logarithm, and completely eliminated the extra factor when $T = \Omega(n \lg^* n)$. However, the weaker bound we need has a much easier proof, which we include for the sake of completeness.

LEMMA A.1. **(Read-only-memory integer sorting and element distinctness)** *Given a sequence of $n$ integers in $[U]$, a parameter $n \leq T \leq n^2$ and any constant $\varepsilon > 0$, we can output the elements in sorted order in $O(T+\text{RAM-SORT}(n))$ time using $O((n^2/T)\lg n + n^\varepsilon \lg U)$ bits of space in the read-only memory model. Furthermore, we can decide whether the elements are distinct in $O(T)$ expected time using $O((n^2/T)\lg n + \lg U)$ bits of space in the read-only memory model.*

*Proof.* The obvious way to translate a word-RAM sorting algorithm into one in read-only memory is to copy

the input first, which would require at least $\Theta(n \lg U)$ bits of space in general. We first show how to convert any RAM-SORT$(n)$-time sorting algorithm into one with $O(\text{RAM-SORT}(n))$ time and $O(n \lg n + n^\varepsilon \lg U)$ bits of space.

One approach is a multi-way quicksort: First select $O(b)$ approximate quantiles to decompose into $O(b)$ intervals each containing at most $n/b$ elements, where $b$ is a fixed parameter to be set later. For this step, we can for example apply a streaming algorithm of Munro and Paterson [25], which uses $O(b^{O(1)} \lg^{O(1)} n \lg U)$ bits of space and $O(n/(b \lg_b n))$ merging/sorting operations for sublists of size $O(b \lg_b n)$; this requires $O((n/(b \lg_b n)) \,\text{RAM-SORT}(O(b \lg_b n))) \le O(\text{RAM-SORT}(n))$ time.

For each element, we locate which interval it is in. For this step, we can take each group of $b$ elements and sort them along with the $O(b)$ quantiles; this requires $O((n/b) \,\text{RAM-SORT}(O(b))) \le O(\text{RAM-SORT}(n))$ time and $O(b^{O(1)} \lg U)$ bits of space. In addition, to keep pointers between elements and intervals, we need $O(n \lg n)$ bits of space.

Finally, we recursively sort the elements inside each interval. If the input size is less than $b$, we can sort directly in RAM-SORT$(n)$ time using $O(b^{O(1)} \lg U)$ space.

Because the recursion depth is $O(\lg_b n)$, the total time is $O(\text{RAM-SORT}(n) \lg_b n)$ and the total space in bits is $O(b^{O(1)} \lg^{O(1)} n \lg U)$ plus $O(n \lg n + (n/b) \lg(n/b) + \cdots) = O(n \lg n)$. We can then set $b = n^{\Theta(\varepsilon)}$ to get $O(\text{RAM-SORT}(n))$ time and $O(n \lg n + n^\varepsilon \lg U)$ bits of space.

Now, to obtain the time–space tradeoff for sorting, assume that $T \le n \lg n$, for otherwise we can use a known comparison-based algorithm in read-only memory [19]. Select $O(T/n)$ quantiles to decompose into $O(T/n)$ intervals each containing at most $n^2/T$ elements. For this step, we can make $O(T/n)$ invocations to a known selection algorithm in read-only memory with $O(n)$ time and $O(n)$ bits of space [15]. Afterwards, for each interval in sorted order, we find all the elements in the interval by a linear scan, store a list of pointers to such elements using $O((n^2/T) \lg n)$ bits of space, then run the preceding algorithm on just these elements. The total time is $O((T/n) \cdot (n + \text{RAM-SORT}(n^2/T))) \le O(T + \text{RAM-SORT}(n))$.

For element distinctness, there is a known algorithm with $O(n)$ expected time by universal hashing [12], and this algorithm already uses just $O(n \lg n + \lg U)$ bits of space. From this, the time–space tradeoff can be obtained in the same way. □