BRICS

Basic Research in Computer Science

BRICS RS-01-2 Pagh & Pagter: Optimal Time-Space Trade-Offs for Non-Comparison-Based Sorting

Optimal Time-Space Trade-Offs for Non-Comparison-Based Sorting

Rasmus Pagh
Jakob Pagter

BRICS Report Series **RS-01-2**

ISSN 0909-0878 **January 2001**

Optimal Time-Space Trade-Offs for Non-Comparison-Based Sorting*

Rasmus Pagh and Jakob Pagter

BRICS†
University of Aarhus
DK-8000 Aarhus C
Denmark

January, 2001

*Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

†**B**asic **R**esearch **i**n **C**omputer **S**cience. Centre of the Danish National Research Foundation.

Abstract

We study the fundamental problem of sorting n integers of w bits on a unit-cost RAM with word size w, and in particular consider the time-space trade-off (product of time and space in bits) for this problem. For comparison-based algorithms, the time-space complexity is known to be $\Theta(n^2)$. A result of Beame shows that the lower bound also holds for non-comparison-based algorithms, but no algorithm has met this for time below the comparison-based $\Omega(n \lg n)$ lower bound.

We show that if sorting within some time bound $\tilde{T}$ is possible, then time $T = O(\tilde{T} + n \lg^* n)$ can be achieved with high probability using space $S = O(n^2/T + w)$, which is optimal. Given a deterministic priority queue using amortized time $t(n)$ per operation and space $n^{O(1)}$, we provide a deterministic algorithm sorting in time $T = O(n\,(t(n) + \lg^* n))$ with $S = O(n^2/T + w)$. Both results require that $w \leq n^{1-\Omega(1)}$. Using existing priority queues and sorting algorithms, this implies that we can deterministically sort time-space optimally in time $\Theta(T)$ for $T \geq n\,(\lg \lg n)^2$, and with high probability for $T \geq n \lg \lg n$.

Our results imply that recent lower bounds for deciding element distinctness in $o(n \lg n)$ time are nearly tight.

1 Introduction

1.1 Motivation and results

The study of time-space trade-offs, i.e., formulae that relate the most fundamental complexity measures, time and space, was initiated by Cobham [15], who studied problems like recognizing the set of palindromes on Turing machines. There are two main lines of motivation for such studies: one is the lower bound perspective, where restricting space allows you to prove general lower bounds for decision problems [3, 4, 9, 10, 23]; the other line is the upper bound perspective where one attempts to find time efficient algorithms that are also space efficient (or vice versa). Also, upper bounds are interesting for more "academic" reasons, namely to establish, in conjunction with lower bounds, the computational complexity of fundamental problems such as sorting.

The complexity of sorting is a classical and well-studied problem in computer science. We consider the time-space trade-off of perhaps the most basic form of this problem, namely sequential sorting of a list of w-bit integers, where w refers to the number of bits that can be processed in one time step by the computational model. Our model of computation is a unit-cost RAM with word size w and a standard instruction set.

As space complexity in the setting of time-space trade-offs is often sublinear, the models used have read-only access to the input, and write-only access to the output. Thus, when we speak of space complexity we refer to the space usage in *working memory*, that is, the amount of space used for the data structure employed. A natural question is whether this measure of space is realistic. As an example, consider the task of sorting a large database by secondary key: In such an example it can be important not to overwrite the original database sorted by primary key. A typical example is the customer database of a bank, which will normally be sorted by account numbers. Occasionally the bank might want a phone book over its customers, requiring the database to be sorted by customer names, but rarely will it be interested in erasing the original database used for all standard business transactions. Other examples occur when the input is stored on a medium which is physically read-only, for example on a CD-ROM. Moreover, as argued above, the question is interesting for mere "academic" reasons. To be consistent with the literature in the area, we will measure space in terms of the number of bits, and *not* words, in working memory.

For comparison-based sorting the time-space trade-off is settled, as Borodin et al. proved that any comparison-based algorithm must have $TS = \Omega(n^2)$, and Pagter and Rauhe exhibited an algorithm realizing $TS = O(n^2)$ for all S down to the $\Omega(w)$ space lower bound. Beame extended the lower bound of Borodin et al. to a model encompassing any reasonable sequential model of computation, completely revealing the asymptotic time-space complexity of sorting for time $\Omega(n \lg n)$. This lower bound also holds for the product of the expected time and space for any randomized Las Vegas algorithm.

Below the comparison-based $\Omega(n \lg n)$ time lower bound, however, the exact time-space complexity of sorting is still unknown. The lower bound of Beame holds down to time n, implying for instance that any linear time sorting algorithm must use space $O(n)$. To our knowledge, all algorithms sorting in time $o(n \lg n)$ use space at least $\Omega(nw)$, which is at best a factor $\Omega(\lg n)$ from optimal.

In this paper we provide time-space optimal upper bounds for time in $o(n \lg n)$. Proving a general reduction and applying it to the best known deterministic and randomized priority queues we obtain the following time-space trade-offs, optimal for $w \leq n^{1-\Omega(1)}$:

Theorem 1 *Let $\epsilon > 0$ be constant. For $T \geq n\,(\lg\lg n)^2$, sorting of n words can be done deterministically in time $O(T)$, using $O(n^2/T + n^\epsilon w)$ bits of memory. For $T \geq n \lg\lg n$, sorting of n words can be done in time $O(T)$ w.h.p.*[1]*, using $O(n^2/T + n^\epsilon w)$ bits of memory.*

Note that, since we can always resort to an $O(n^2)$ time sorting algorithm if the time bound is exceeded, the high probability bound can also be made to hold in the expected sense. Theorem 1 improves the space usage of all existing $o(n \lg n)$ sorting algorithms by a factor of $\Omega(w)$. Meeting the lower bound without a condition like $w \leq n^{1-\Omega(1)}$ is probably too much to hope for—indeed, when $w \geq n$ one can only keep a constant number of elements in space $O(n + w)$, which appears to make it hard to benefit from non-comparison-based techniques.

Our main technical contribution is the following lemma:

[1] "With high probability", refers to probability greater than $1 - n^{-c}$, where c is any constant of the user's choice.

Lemma 2 *Let $t : \mathbf{N} \to \mathbf{R}_+$ be non-decreasing, and $\epsilon > 0$ a constant. Suppose there is a (monotone[2]) priority queue supporting* `Insert`*,* `FindMin`*, and* `DeleteMin` *in amortized time $O(t(n))$, using space $n^{O(1)}$. Then any list of n words can, for $T \geq n\,t(n)$, be sorted in time $O(T)$, using $O(n^2 \lg^{(T/n)}(n)/T + n^\epsilon w)$ bits of memory[3]. If the time bound for the priority queue only holds w.h.p. or in the expected sense, the same is true for the sorting time bound.*

Note that $\lg^{(T/n)} n = 1$ for $T \geq n \lg^* n$. Our reduction uses multiplication, but for $t(n) = 2^{\Omega(\lg^* n)}$ we can avoid multiplication if not employed by the priority queue itself (in fact, only AC^0 instructions are introduced).

Using Lemma 2 and a conversion from sorting algorithms to monotone priority queues due to Thorup [25] (see Theorem 4 below), we provide a reduction showing that any $\Omega(n \lg^* n)$ time sorting algorithm can be converted into one using minimal space for $n \geq w^{1+\Omega(1)}$, at no asymptotic cost in time (with high probability for any input). For time $o(n \lg^* n)$ there is a tiny gap to the time-space lower bound (for any constant k, a factor of $O(\lg^{(k)} n)$).

Theorem 3 *Let $\tilde{T} : \mathbf{N} \to \mathbf{R}_+$ be non-decreasing, and $\epsilon > 0$ a constant. Suppose there is an algorithm sorting n words in time $O(\tilde{T}(n))$ w.h.p. Then a list of n words can, for $T \geq \tilde{T}(n)$, be sorted in time $O(T)$ w.h.p., using $O(n^2 \lg^{(T/n)}(n)/T + n^\epsilon w)$ bits of memory.*

The theorem extends to the case where the $O(\tilde{T}(n))$ time bound only holds when $n \leq f(w)$, for some real function f. In this case the resulting space-efficient sorting algorithm also requires that $n \leq f(w)$. If the $O(\tilde{T}(n))$ time bound holds in the expected sense, the same is true for our space-efficient sorting algorithm.

Our algorithms are able to output any information attached to an input element, in the sense that when outputting an element, the location in the input is known.

As an interesting aside, our unconditional upper bounds yield the same upper bounds on deciding element distinctness. This should be compared to the lower bounds of Ajtai [4] and Beame et al. [9]. Ajtai showed that for time $O(n)$, space $\Omega(n)$ is required, and this was later generalized by Beame et al.,

---

[2]A monotone priority queue does not allow insertion of elements less than the current minimum.

[3]$\lg^{(k)} n$ denotes the logarithm of n iterated $\lfloor k \rfloor$ times, e.g. $\lg^{(2)} n = \lg \lg n$. For $k > \lg^* n$ we define $\lg^{(k)} n = 1$.

reference	**T**	**TS**
Andersson et al. [6]	$n \lg\lg n$	$n^{2+o(1)}w$ w.h.p.
Han [21]	$n \lg\lg n \lg\lg\lg n$	$n^{2+o(1)}w$
Pagter and Rauhe [24]	$n \lg n$ **to** n^2/w	n^2
new	$n\,(\lg\lg n)^2$ **to** n^2/w	n^2
new	$n \lg\lg n$ **to** n^2/w	n^2 w.h.p.

Table 1: Time-space upper bounds for sorting, $w \leq n^{1-\Omega(1)}$.

who showed that for space $O(S)$ one must use time $\Omega(n\sqrt{\lg(n/S)/\lg\lg(n/S)})$, even if the algorithm is allowed to use randomization and have two-sided error. This implies that to solve element distinctness in time, say $O(n \lg\lg n)$, in the Las Vegas fashion (which we can do in space $O(n/\lg\lg n)$), space $n^{1-o(1)}$ is required.

We get stronger upper bounds for element distinctness (by sorting) in several special settings. For $(\lg n)^{2+\Omega(1)} \leq w \leq n^{1-\Omega(1)}$ we obtain an algorithm using time $O(n \lg^* n)$ with high probability and space $O(n/\lg^* n)$, using the linear time sorting algorithm from [6][4]. From the element distinctness lower bound, which holds for all $w \geq 2\lg n$, we know that space $\Omega(n/(\lg^* n)^{(\lg^* n)^2})$ is required. For $w > n^{\Omega(1)}$—i.e., really large word sizes—one can match the above performance deterministically, employing a deterministic linear space priority queue using a *non-standard* AC^0 instruction set to provide constant time operations [20].

1.2 History

1.2.1 Lower bounds

Sorting Borodin et al. [13] founded the area of time-space trade-offs for sorting, by proving that any comparison-based sorting algorithm running in time T has $TS = \Omega(n^2)$. To permit more general bounds, Borodin and Cook [11] introduced the R-way branching program model which, for $R = 2^w$, is at least as strong as a unit-cost RAM with word size w, read-only input, write-only output, and *any* instruction set. In this model they proved a lower bound of $TS = \Omega(n^2/\lg n)$ for any T and any $w \geq \lg n$. This was later

[4]The high probability bound for this algorithm is not stated in the paper, but can be achieved by increasing the range of the hash functions employed.

improved by Beame [8], who showed that any sorting algorithm running in time T has $TS = \Omega(n^2)$. The proof counts only accesses to the input, and hence holds for any kind of instruction set as long as we can only read $O(1)$ input words at a time. As noted by Beame, his result also holds for average time and space by using ideas of Abrahamson [1]. Yao's minimax principle [28] then provides the same lower bound for the expected time and space usage of any randomized Las Vegas algorithm.

Element distinctness For the related problem of element distinctness, time-space trade-offs are also well-studied. Borodin et al. [12] proved a lower bound of $TS = \Omega(n^{3/2})$ for comparison-based algorithms, which was later improved by Yao [29] to $TS = \Omega(n^{2-\epsilon(n)})$, where $\epsilon(n) = 5/\sqrt{\lg n}$. In the general setting of R-way branching programs Ajtai [4] showed that for time $O(n)$ one must use space $\Omega(n)$. This result was later generalized by Beame et al. [9], who show that for space $O(S)$ one must use time $\Omega(n\sqrt{\lg(n/S)/\lg\lg(n/S)})$, or conversely, if time is restricted to kn, then space $n/k^{O(k^2)}$ is required.

1.2.2 Upper bounds

Sorting on the RAM The classical comparison-based upper bounds on time for sorting also provide upper bounds on the time-space trade-off, albeit in a limited fashion: A typical algorithm like mergesort uses space at least $n \lg n$, as it stores $\Theta(n)$ (pointers to) elements, hence giving $TS = \Omega(n^2 \lg^2 n)$, for $T = n \lg n$. Note that in-place sorting algorithms, such as quicksort, cannot be implemented directly in our model, as we are not allowed to change the input. Instead, in-place algorithms can be simulated by first copying the entire input to working memory, implying that $S = \Omega(nw)$ for in-place sorting algorithms in our model.

The first attempt to provide more *scalable* solutions was made by Munro and Paterson [22], providing tight bounds when input access is sequential. In the random access model, an optimal $TS = O(n^2)$ upper bound for $n \lg n \leq T \leq n^2/w$ was given by Pagter and Rauhe [24], using ideas of Frederickson [17].

It has long been known that sorting in linear time is possible when $w = O(\lg n)$, by using the indirect addressing features of the RAM (radix sort). A flurry of research on non-comparison-based algorithms was initiated by the seminal paper of Fredman and Willard [18], who exhibited the first truly $o(n \lg n)$ sorting algorithm, with *no restriction* on w. More precisely, they showed how to sort deterministically in time $O(n \lg n / \lg \lg n)$ and space $O(nw)$. They also present an $O(n\sqrt{\lg n})$ time sorting algorithm, but it uses either exponential space or randomization to implement a sparse table in space $O(nw)$.

Another breakthrough was made by Andersson et al. [6], who achieved time $O(n \lg \lg n)$ by a quite simple algorithm. Again, this algorithm uses either exponential space or randomization to implement a sparse table. In the special case $w \geq (\lg n)^{2+\Omega(1)}$, Andersson et al. in fact exhibit a randomized *linear* time sorting algorithm using space $O(nw)$. For deterministic sorting in space $O(nw)$, the current "record holder" is the algorithm of Han [21], who achieves time $O(n \lg \lg n \lg \lg \lg n)$[5]. To our knowledge, this is the fastest deterministic sorting algorithm whose space usage is not exponential in w.

Bounds similar to the above can be achieved via linear space priority queues due to Thorup, achieving $O(\lg \lg n)$ expected amortized time per operation [25], and $O((\lg \lg n)^2)$ deterministically [26]. In fact, Thorup [25] has given a general way of transforming any efficient sorting algorithm into an efficient monotone priority queue. This is described in more detail in section 2.2.1.

For further information on sorting on the RAM, we refer the reader to the surveys of Andersson [5] and Hagerup [20].

Sorting in other models Time-space upper bounds become relevant from a practical point of view when the number of input elements is huge, a line of reasoning that has lead to the study of so-called I/O-space trade-offs. Recently both upper and lower bounds on the I/O-space trade-off for sorting and element distinctness were proved in the I/O-model [2] by Arge and Pagter [7], building on the abovementioned upper and lower bound techniques.

For completeness, we mention that Beame actually provides a tight time-space upper bound *in the branching program model* for elements in the range $1, \ldots, n$. So within this model the lower bound cannot be improved independently of the word size.

Element distinctness In the comparison based setting, the sorting algorithm of Pagter and Rauhe [24] nearly closes the gap to Yao's lower bound [29]. For non-comparison-based algorithms however, much less is known. Of

[5]We have not had access to the paper describing this result.

course, all the abovementioned fast sorting algorithms show how to decide element distinctness, but none of them in a very time-space efficient manner (more than a factor of w from the lower bound). Using universal hash functions [14] one can decide element distinctness in expected linear time, using $O(n \lg n + w)$ bits of space. By another approach based on hashing, Ajtai [4] provides a two-sided error randomized algorithm with time and space $O(n + w)$, which is tight by the lower bound of Beame et al. [9]. The algorithm does not appear to generalize to time $\omega(n)$.

1.3 Intuition

We illustrate our approach by sketching a direct proof of Theorem 1 in the special case where $w \leq (\lg n)^c$ for some constant c. We sketch how to transform a space $O(nw)$ and time $t \geq \lg \lg n$ priority queue into a time-space optimal sorting algorithm for $T = nt$, i.e., using space n/t. Our main ingredient is the algorithm of Pagter and Rauhe [24] which sorts $n \leq 2^t$ numbers in time $O(t)$ using $O(n/t + \lg n)$ bits of space; in fact, their construction allows us to repeatedly retrieve the smallest remaining element (a `DeleteMin` operation) in time $O(t)$. This handles the case $t \geq \lg n$. Otherwise, we start by splitting the input into intervals of length $(\lg n)^{c+1}$, for each of which we construct the data structure of Pagter and Rauhe, allowing us to report the current minimum of each interval in time $O(t)$ using space $O((\lg n)^{c+1}/t + \lg n)$, which, summing over all intervals, yields $O(n/t)$ bits. "On top" of this we now use the t time priority queue to repeatedly report the minimum of the elements not yet reported in the manner of Frederickson [17], the idea being that each interval has exactly one element in the priority queue, namely the current minimum of that interval. Calling `DeleteMin` on the priority queue will then give us the current global minimum. Each time we report an element from some interval, we use the data structure of Pagter and Rauhe to time-space efficiently find the "next minimum" of that interval. Repeating this n times will sort the input. As the priority queue contains one element per interval and each element takes up $w \leq (\lg n)^c$ bits the priority queue uses $O(n/\lg n)$ bits in total, and the total time consumption per element is $O(t)$. Using the above with Thorup's priority queues [25, 26] we arrive at Theorem 1 in the special case $w \leq (\lg n)^c$.

We can extend our approach to time-space optimal sorting for any $T \geq n \lg^* n$ given a monotone priority queue with amortized time complexity $t = T/n$. The general idea is to apply the scheme described above recursively to the intervals. However, we cannot afford a permanent fast priority for each interval (more precisely, the total number of elements in all priority queues used must at any time be $O(n/tw)$ to use $O(n/t)$ bits). The way around this is to compute the smallest elements of each interval in "bursts". During a burst, a fast priority queue will be generating the smallest elements of the interval, in the manner described above. After a burst there will be an array of pointers to the smallest elements in the interval that can be used for subsequent `DeleteMin` operations. A pointer to an element within a small interval takes up much less space than the element itself. In fact, the length of intervals and pointers will decrease exponentially at each step of the recursion. We use the priority queue given only at the topmost $O(1)$ layers, below which the sets become so small that constant time priority queues are possible. At the bottom of the recursion we use the data structure of Pagter and Rauhe. For time below $n \lg^* n$ we use fewer than $\lg^* n$ levels of recursion, giving a slight increase in space.

2 Preliminaries

2.1 Model of computation

Our model of computation is a unit-cost RAM with word size w and a standard instruction set, including multiplication. It is assumed that $w \geq \lg n$, such that pointers to the input is possible. The memory of the RAM is split into three parts: input which is read-only, output which is write-only, and working memory in which we can both read and write. At each time step we may: 1) read one word from input and write it to working memory, 2) write one word to output, or 3) make one binary operation based on at most two words from working memory and write the result to one word in working memory. We will not use random access on the output, but simply output the elements "left to right" in sorted order. Time is measured as the number of operations 1), 2), and 3)—hence the name unit-cost. Space is the maximal number of bits used in working memory during computation.

The comparison-based model used for earlier time-space upper bounds [17, 24] differs from ours in that elements are never explicitly stored in working memory. Indeed, the word size is $O(\lg n)$ and one works with *pointers* to elements. It is possible to compare elements and copy elements to output without using working memory depending on w, allowing space usage down to $O(\lg n)$ rather than $O(w)$. Algorithms in this model can be simulated in our model with a space overhead of $O(w)$. We feel that the present model is more realistic. Further, since our focus is on very fast sorting algorithms, the $\Omega(w)$ space lower bound is insignificant. Finally this is consistent with the R-way branching program model used for the lower bound.

2.2 Some results on priority queues

We now survey two results on priority queues that are used to prove our results. We assume that it is possible to associate w bits of information with each element in a priority queue. This is clearly true for any priority queue we are aware of, and can in fact be assumed without loss of generality (see Appendix B).

2.2.1 From sorting to priority queues

Our reduction of time-space optimal sorting to sorting with no space restriction relies on a transformation result of Thorup [25]. It is a black-box transformation that, given any polynomial space sorting algorithm running in time $O(n\,t(n))$, provides you with a polynomial space monotone priority queue with time $O(t(n))$ per operation in the amortized sense, w.h.p. We may, in fact, remove the assumption that the sorting algorithm uses polynomial space: Since we can assume that it is fast, i.e. accesses $O(n \lg n)$ words in memory, the entire memory of size 2^w can be simulated by a dictionary using $O(n \lg n)$ words of space with constant factor overhead, w.h.p. (the perhaps simplest implementation of this being a hash table with chaining, using the reliable hash functions of Dietzfelbinger et al. [16]).

Theorem 4 (Thorup [25]) *Let $s, t : \mathbf{N} \to \mathbf{R}_+$ be non-decreasing. Suppose there is an algorithm sorting n words in time $O(n\,t(n))$, using $s(n)w$ bits of space. Then there exists a monotone priority queue supporting* `Insert`*,* `FindMin`*, and* `DeleteMin` *in amortized time $O(t(n))$ per operation using $s(n)w + O(nw)$ bits of space. The time bound holds w.h.p., or in the expected sense if the time bound of the sorting algorithm is expected.*

The space bound and the high probability time bound are not explicit in [25], but they are easy to derive. Thorup also describes a variant of the above transformation that yields a deterministic priority queue if the sorting algorithm is deterministic, but that priority queue uses space $2^{\Omega(w)}$ and so is only space-efficient enough for our purposes when $w = O(\lg n)$.

2.2.2 Small constant-time priority queues

We will make use of the *Q*-heaps* of Fredman and Willard (contained in [19], explicitly described in [27]).

Theorem 5 (Fredman and Willard) *Let $M \leq w^{O(1)}$ and $\delta \in \mathbf{R}_+$. There is a priority queue storing any set of $n \leq M$ elements using $O(nw)$ bits of space, supporting all operations in amortized constant time. The priority queue relies on a fixed table (depending only on M) of 2^{M^δ} words. The table can be computed in time $2^{O(M^\delta)}$.*

The bound on the table size stated in [27] is $o(2^M)$ words, but the stronger bound stated here can be easily derived. The crux of the above is that all Q*-heaps can use the same table. In our construction, we will need Q*-heaps of size at most $M = O(\lg^2 n)$. Thus, choosing $\delta = 1/4$ renders both the time and space used for the table negligible.

3 Proofs

For simplicity we will give the proofs under the assumption that n is a power of two. It is of course simple to reduce the general case to this one with a constant factor loss in time and space.

Our basic building block is a "decremental" priority queue, called an *interval sorter*, defined over an interval of input elements. It supports `FindMin` and `DeleteMin`, and initially contains all elements in the interval. We will consider interval sorters over very small intervals, making even $\lg n$ bits of space usage unacceptable. We thus assume that the program performing the priority queue operations has access to the following information (i.e., the information does not reside in the interval sorter itself):

i) Pointers to the leftmost element of the interval and the length of the interval (given as the integer logarithm of this number).

ii) Pointer to an element smaller than the current minimum of the priority queue (if any), but larger than or equal to the last element deleted from the priority queue.

iii) The space used by the interval sorter.

We will distinguish between space permanently used by an interval sorter and space used only during operations (i.e., during initialization and when `FindMin` or `DeleteMin` is being carried out), by referring to the former simply as *space*, and to the latter as *run-time memory*. For purposes of the analysis we require that the time to carry out `FindMin` is constant. We refer to the total time for initialization and all `DeleteMin` operations divided by the interval length as the *operation time*.

The sorting algorithm of Pagter and Rauhe [24] is in fact an interval sorter for any length 2^l and operation time $t \geq cl$, for some constant c, using space $O(2^l/t + l)$ and run-time memory $O(w)$. A self-contained proof of this can be found in Appendix A.

We now show three lemmas on composition of interval sorters. We will refer to the following assumptions:

Assumption 1 *We are given a (monotone) priority queue using at most $t(n)$ time and $s(n)w$ bits of space, where $s, t : \mathbf{N} \to \mathbf{R}_+$ are non-decreasing functions.*

Assumption 2 *We are given $z, l \in \mathbf{N}$, $2 < l < z \leq w$, and an interval sorter for length 2^l using b bits of space, run-time memory at most m and operation time at most a.*

Our first lemma describes the basic way in which we compose interval sorters to cover larger intervals:

Lemma 6 *Under Assumptions 1 and 2 we can construct an interval sorter for length 2^z using $2^{z-l}(b + 2z)$ bits of space, run-time memory at most $m + O(s(2^{z-l})w)$ and operation time at most $a + O(t(2^{z-l}))$.*

Proof. The interval sorter's data structure consists of:

- The number l, easily stored using z bits.
- A counter $p \in \{0, \ldots, 2^{z-l} - 1\}$, using (at most) z bits.
- An array of 2^{z-l} pointers to elements in the interval such that element number p to $2^{z-l} - 1$ are in sorted order, and are the smallest undeleted elements in the interval sorter. If all elements have been deleted, p points to an arbitrary array entry. In total this uses $2^{z-l}z$ bits.

- A length 2^{z-l} array of b-bit sub-interval sorters, the ith one covering elements $2^l i, \dots, 2^l(i+1)-1$, for $i = 0, \dots, 2^{z-l}-1$. This uses a total of $2^{z-l}b$ bits.

The overall space usage is $2^{z-l}(b+z)+2z \leq 2^{z-l}(b+2z)$ bits. Note that we can compute the size of the sub-interval sorters in constant time from the above information.

By invariant we can determine if the interval sorter is empty using ii). If it is not, the interval sorter operations are trivial to perform in constant time as long as there are undeleted elements in the pointer array. To initialize the interval sorter, and whenever the last element in the array is deleted, we perform the following:

1. Insert the minimum element (if any) of each sub-interval sorter in the priority queue, associating with it a pointer to its position in the interval.
2. Repeat 2^{z-l} times:
   (a) Perform `DeleteMin` from the priority queue, getting a pointer to element x.
   (b) Insert the pointer to x in the pointer array.
   (c) Remove x from its sub-interval sorter.
   (d) Put the new minimum from the sub-interval sorter into the priority queue.
3. Set $p = 0$.

This gives the 2^{z-l} smallest undeleted elements, using $O(2^{z-l})$ operations on the priority queue, 2^{z-l} deletions from the sub-interval sorters, and time $O(2^{z-l})$ for everything else. The bound on operation time and run-time memory immediately follows. Since the minimum of the priority queue never decreases, a monotone priority queue suffices by definition. Note that ii) can be provided to the sub-interval sorters by sending a reference to element $2^{z-l}-1$ in the array. □

The second lemma essentially shows how interval sorters using run-time memory $n^{O(1)}w$ can be composed such that the run-time memory in terms of the combined length n is $O(n^{\epsilon}w)$.

Lemma 7 *Under Assumptions 1 and 2 and for any constants $c, p \in \mathbf{N}$, if $s(n) = O(n^c)$ we can construct an interval sorter for length 2^z using $O(2^{z-l}(b+z))$ bits of space, run-time memory $O(m + 2^{z/p}w)$ and operation time $O(a + t(2^z))$.*

Proof. The space usage of applying the priority queue directly is $s(2^z)w = O(2^{zc}w)$. To reduce this we will allow only application of the priority queue to at most $2^{z/pc}$ elements, as this yields the desired run-time memory $s(2^{z/pc})w = O((2^{z/pc})^c w) = O(2^{p/c}w)$. More specifically, we will build an interval sorter for length 2^z by repeatedly applying Lemma 6 (fewer than pc times), such that the ratio between interval sizes is at most $2^{z/pc}$.

For $z < pc$ the lemma is trivial as z is constant and we can apply the priority queue directly. Otherwise we repeatedly compose interval sorters according to Lemma 6. The first composition uses the interval sorter for length 2^l, and all following compositions use the result of the previous composition. Let k be the largest integer such that $z - k\lfloor z/pc \rfloor > l$. We use the ith composition to obtain length 2^{z_i}, where $z_i = z + (i - k - 1)\lfloor z/pc \rfloor$, for $i = 1, \ldots, k + 1$.

The interval sorter obtained by the first composition has size $2^{z_1 - l}(b+2z_1)$. More generally, we show by induction for $i = 1, \ldots, k+1$ that the interval sorter obtained at composition i uses at most $b_i = 2^{z_i - l}(b + 2i\, z_i)$ bits of space. This means that the interval sorter for length 2^z has size at most $b_{k+1} = 2^{z-l}(b + 2(k+1)z)$. For the inductive step, Lemma 6 bounds the size of interval sorter $i > 1$ by:

$$2^{z_i - z_{i-1}}(b_{i-1} + 2z_i) = 2^{z_i - z_{i-1}}(2^{z_{i-1} - l}(b + 2(i-1)\, z_{i-1}) + 2z_i) \leq b_i \ .$$

The bound on operation time follows as we compose a constant number of times, each time adding the stated operation time. Similarly, for each composition the run-time memory usage is increased by at most $s(2^{\lfloor z/pc \rfloor})w = O(2^{z/p}w)$. $\square$

The final lemma gives us an efficient way of composing tiny interval sorters to form an interval sorter for length around $\lg^2 n$. It might be helpful to read the lemma with the assignment $z = \lg\lg n$ in mind. Note that the conditions on k then imply that k is smaller than $\lg^* n$.

Lemma 8 *Under Assumption 2, if $z = O(\lg w)$ and there is $k \in \mathbf{N}$ such that $l = \lfloor 2 \lg^{(k)} z \rfloor$, we can construct an interval sorter for length 4^z using $O(4^z b/2^l)$ bits of space, run-time memory $O(m + 4^z w \lg^* z)$ and operation* *time $O(a+k)$. The interval sorter relies on an external table (depending only on z) of $O(2^{2^{\delta z}} w)$ bits that can be computed in time $2^{O(2^{\delta z})}$, where $\delta > 0$ is a constant of our choice.*

Proof. We again repeatedly use Lemma 6 starting with composition of the interval sorter for length 2^l. The length used for the ith composition is 2^{z_i}, where $z_i = \lfloor 2 \lg^{(k-i)} z \rfloor$, for $i = 1, \ldots, k$. (Note that for $z = \lg\lg n$ the top interval sorter has length roughly $\lg^2 n$, the second length $(\lg\lg n)^2$, the third length $(\lg\lg\lg n)^2$, etc.) We use the *Q^*-heap* priority queue described in Section 2.2.2 with constant time operations.

Now, we show by induction that the interval sorter obtained by the ith composition uses at most $b_i = 2^{z_i - l}\, b\, (1 + \sum_{j=1}^{i} 2^{l+2}/z_j)$ bits of space. Note that indeed $b_k = O(2^{2z-l} b)$. For $i = 0$ we have the interval sorter of Assumption 2 which indeed has size $b_0 = b$. For the inductive step, Lemma 6 bounds the size of interval sorter $i \geq 1$ by:

$$2^{z_i - z_{i-1}}(b_{i-1} + 2z_i) < 2^{z_i}(2^{-l}\, b\, (1 + \sum_{j=1}^{i-1} 2^{l+2}/z_j) + 4/z_i) \leq b_i \enspace .$$

Since we perform $k \leq \lg^* z$ compositions, each increasing run-time memory by $O(4^z w)$ and operation time by $O(1)$, the bounds on run-time memory and operation time are immediate. □

Using the interval sorter of Pagter and Rauhe we can now prove our main lemma:

Proof of Lemma 2. Let $c \in \mathbf{N}$ be a constant such that the priority queue uses space $O(n^c w)$. If $T \geq n \lg n$ we simply use the algorithm of [24]. Otherwise, assume without loss of generality that $T \geq 4n$, and let $r = \lg n$, $t = \lfloor T/n \rfloor$ and $l_1 = 2\lfloor \lg r \rfloor$. Our task can be reduced to that of building an interval sorter for length 2^{l_1} (roughly $\lg^2 n$) with operation time $O(t)$, using space $O(2^{l_1} \lg^{(t)}(n)/t)$ and run-time memory $O(2^{l_1 c})$: Applying Lemma 7 with $p = \lceil 1/\epsilon \rceil$ and $z = r$ to such an interval sorter and the priority queue of Assumption 1, we get an interval sorter for length n with operation time $O(t)$, run-time memory $O(n^\epsilon w)$ and space $O(2^{r-l_1}(2^{l_1} \lg^{(t)}(n)/t + r))$, as desired.

If $t \geq \lg r$ (i.e., $\lg\lg n$) we immediately have the required length 2^{l_1} interval sorter by [24]. Otherwise let $q \geq 1$ be the largest integer for which $\lfloor 2 \lg^{(q)} \lfloor \lg r \rfloor \rfloor \geq \lfloor 2 \lg t \rfloor$. We distinguish two cases:

1. $t \geq q$. Let $l_2 = \lfloor 2\lg^{(q)}\lfloor \lg r \rfloor \rfloor$. By definition of q we have $l_2 \leq 2t + O(1)$, so [24] provides an interval sorter for length 2^{l_2} with operation time $O(t)$, using $O(2^{l_2}/t)$ bits of space and run-time memory $O(w)$. We can thus use Lemma 8 with this and parameters $k = q$, $z = l_1/2$, $l = l_2$ and $\delta = 1/4$ to get the required interval sorter.

2. $t < q$. Let $l'_2 = \lfloor 2\lg^{(t)}\lfloor \lg r \rfloor \rfloor$. We first use Lemma 6 with the Q*-heap and the interval sorter of [24] with length and operation time $2^{\lfloor \lg t \rfloor}$ to get an interval sorter for length $2^{l'_2}$ using space $O(2^{l'_2}\lg^{(t)}(n)/t)$ and with the required operation time and run-time space bounds. Now Lemma 8 with parameters $k = t$, $z = l_1/2$, $l = l'_2$ and $\delta = 1/4$ gives the desired.

Recall that the table used for the Q*-heap takes up $O(2^{2^{\lg\lg(n)/2}}w)$ bits of space and can be precomputed in $2^{O(2^{\lg\lg(n)/2})}$ time, both of which are negligible.

Any expected time bound for the priority queue of Assumption 1 is preserved by linearity of expectation. If the priority queue's time bound holds with high probability, the same holds for our interval sorter: We use $o(n)$ priority queues on sets of size at least n^ϵ for some constant $\epsilon > 0$, so to get error probability n^{-c} it suffices to choose the exponent in the error bounds of the priority queues to be $(c+1)/\epsilon$. $\square$

The Q*-heap introduces a rather large constant factor on the time. If the fast priority queue given has a smaller multiplicative constant on the time, it might be more practical to use it instead of the Q*-heap. However, this only yields an asymptotically optimal solution for time $n\,2^{\Omega(\lg^* n)}$.

4 Open problems

There remains a small gap in our bounds for time $o(n\lg^* n)$. It is plausible that sorting this fast is not possible in general, but it would be nice at least to have tight bounds for cases where a linear time upper bound is known. The role of randomization in sorting presents many open questions—in our case it would be interesting to see if a general deterministic transformation to optimal space is possible. A particular instance of both of the above is whether there exists an algorithm deciding element distinctness deterministically in $O(n)$ time using $O(n)$ bits for $w = O(\lg n)$.

Another problem is to provide support for the intuition that sorting in $o(n \lg n)$ time is not possible in $O(w)$ bits of space (for w large relative to n).

An interesting question is whether our upper bound can be used in showing a super-linear time lower bound for sorting. All known lower bound techniques require some upper bound on space to give a lower bound for time. If it is not possible in general to sort in time, say, $n \lg\lg\lg n$, then it might be easier to show a lower bound knowing that such an algorithm can be assumed to use $O(n/\lg\lg\lg n)$ bits of space.

References

[1] Karl Abrahamson, *Time-Space Tradeoffs for Algebraic Problems on General Sequential Machines*, Journal of Computer and System Sciences **43** (1991), 269–289.

[2] Alok Aggarwal and Jeffrey Scott Vitter, *The Input/Output Complexity of Sorting and Related Problems*, Communications of the ACM **31** (1988), no. 9, 1116–1127.

[3] Miklós Ajtai, *A Non-linear Time Lower Bound for Boolean Branching Programs*, Proceedings of the 40th Annual Symposium on Foundations of Computer Science, IEEE, 1999.

[4] ———, *Determinism versus Non-Determinism for Linear Time RAMs with Memory Restrictions*, Proceedings of the 31st Annual ACM Symposium on Theory of Computing, ACM, 1999.

[5] Arne Andersson, *Sorting and Searching Revisited*, Algorithm Theory - SWAT'96 (Rolf Karlsson and Andrzej Lingas, eds.), Springer-Verlag, 1996, pp. 185–197.

[6] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman, *Sorting in linear time?*, Proceedings of the 27th Annual ACM Symposium on Theory of Computing, ACM, 1995.

[7] Lars Arge and Jakob Pagter, *I/O-Space Trade-Offs*, 7th Scandinavian Workshop on Algorithm Theory (SWAT'00), Lecture Notes in Computer Science, vol. 1851, Springer-Verlag, 2000, pp. 448–461.

[8] Paul Beame, *A General Sequential Time-Space Tradeoff for Finding Unique Elements*, SIAM Journal on Computing **20** (1991), 270–277.

[9] Paul Beame, Michael Saks, Xiaodong Sun, and Erik Vee, *Super-linear time-space tradeoff lower bounds for randomized computation*, Proceedings of the 41st Annual Symposium on Foundations of Computer Science, IEEE, 2000, pp. 169–179.

[10] Paul Beame, Michael Saks, and Jayram S. Thathachar, *Time-Space Tradeoffs for Branching Programs*, Proceedings of the 41st Annual Symposium on Foundations of Computer Science, IEEE, 2000, pp. 254–263.

[11] Allan Borodin and Stephen Cook, *A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation*, SIAM Journal on Computing **11** (1982), no. 2, 287–297.

[12] Allan Borodin, Faith E. Fich, Friedhelm Meyer auf der Heide, Eli Upfal, and Avi Wigderson, *A Time-Space Tradeoff for Element Distinctness*, SIAM Journal on Computing **16** (1987), 97–99.

[13] Allan Borodin, Michael J. Fischer, David G. Kirkpatrick, Nancy A. Lynch, and Martin Tompa, *A Time-Space Tradeoff for Sorting on Non-Oblivious Machines*, Journal of Computer and System Sciences **22** (1981), 351–364.

[14] J. Lawrence Carter and Mark N. Wegman, *Universal classes of hash functions (extended abstract)*, Proceedings of the 9th Annual ACM Symposium on Theory of Computing, ACM, 1977, pp. 106–112.

[15] Alan Cobham, *The Recognition Problem for the Set of Perfect Squares*, Conference Record of 1966 7th Annual Symposium on Switching and Automata Theory ("FOCS 7"), IEEE, 1966, pp. 78–87.

[16] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger, *Polynomial hash functions are reliable (extended abstract)*, Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92) (Berlin), Lecture Notes in Computer Science, vol. 623, Springer-Verlag, 1992, pp. 235–246.

[17] Greg N. Frederickson, *Upper Bounds for Time-Space Trade-offs in Sorting and Selection*, Journal of Computer and Systems Sciences **34** (1987), 19–26.

[18] Michael L. Fredman and Dan E. Willard, *Surpassing the Information Theoretic Bound with Fusion Trees*, Journal of Computer and System Sciences **47** (1993), 424–436.

[19] Michael L. Fredman and Dan E. Willard, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, Journal of Computer and System Sciences **48** (1994), no. 3, 533–551.

[20] Torben Hagerup, *Sorting and Searching on the Word RAM*, Conference Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science, Lexture Notes in Computer Science, vol. 1373, Springer-Verlag, 1998, pp. 366–398.

[21] Yijie Han, *Improved Fast Integer Sorting in Linear Space*, Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM-SIAM, 2001, pp. ??–??

[22] J. Ian Munro and Mike S. Paterson, *Selection and Sorting with Limited Storage*, Theoretical Computer Science **12** (1980), 315–323.

[23] Jakob Pagter, *On Ajtai's Lower Bound Technique for R-way Branching Programs and the Hamming Distance Problem*, Tech. Report RS-00-11, BRICS, Department of Computer Science, University of Aarhus, 2000.

[24] Jakob Pagter and Theis Rauhe, *Optimal Time-Space Trade-Offs for Sorting*, Proceedings of the 39th Annual Symposium on Foundations of Computer Science, IEEE, 1998, pp. 264–268.

[25] Mikkel Thorup, *On RAM priority queues*, Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM-SIAM, 1996, pp. 59–67.

[26] ______, *Faster deterministic sorting and priority queues in linear space*, Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM-SIAM, 1998.

[27] Dan E. Willard, *Examining Computational Geometry, van Emde Boas Trees, and Hashing from the Perspective of the Fusion Tree*, SIAM Journal on Computing **29** (2000), no. 3, 1030–1049.

[28] Andrew Chi-Chih Yao, *Probabilistic computations: toward a unified measure of complexity (extended abstract).*, Proceedings of the 18th Annual Symposium on Foundations of Computer Science, IEEE, 1977, pp. 222–227.

[29] ______, *Near-optimal Time-Space Tradeoff for Element Distinctness*, SIAM Journal on Computing **23** (1994), 966–975.

A The interval sorter of Pagter and Rauhe

In this section we outline the ideas of Pagter and Rauhe [24].

Lemma 9 *For any $z, l \in \mathbf{N}$, $l \leq z$, there is an interval sorter for length 2^z with operation time $O(2^{z-l} + l)$, using $O(2^l + z)$ bits of space and run-time memory $O(w)$.*

Proof sketch. In a traditional binary heap, each leaf node is associated with an input element, and each internal node is associated with the minimum of its two children. In a heap based on intervals, a leaf node will be associated with the minimum of some interval of the input, but an internal node is still associated with the minimum of its two children.

Basically the data structure of Pagter and Rauhe is a heap based on intervals, i.e., for every interval of the form $2^{z-i}j, \ldots, 2^{z-i}(j+1) - 1$, where $0 \leq i \leq l$ and $0 \leq j < 2^i$, we remember the minimum $a_{i,j}$. Instead of remembering $a_{i,j}$ explicitly, we maintain a pointer $p_{i,j}$ to the leaf-layer interval containing $a_{i,j}$, allowing us to recover $a_{i,j}$ from $p_{i,j}$ making a linear search through an interval of $2^z/2^l = 2^{z-l}$ elements. Such a pointer takes up $l - i$ bits, as we need l bits to name one of the 2^l leaf-layer intervals, but the i most significant bits of $a_{i,j}$ will simply encode the integer j, and do not need to be stored. To speed up the recovery process, we augment each pointer, $p_{i,j}$, with an additional $l - i$ bits used to store more bits of $a_{i,j}$, and thus reducing the number of elements to be searched. To recover $a_{i,j}$ from $p_{i,j}$ we then only need to make a linear search through $\lceil 2^{z-2l+i} \rceil$ elements. The total space usage is $\sum_{i=0}^{l} 2^i \, 2(l-i) < 4 \cdot 2^l$. We leave to the reader the details of a memory layout that does not waste space and allows data to be accessed efficiently.

Clearly the interval sorter can be initialized in linear time by the usual bottom up algorithm. To perform `DeleteMin`, we use the constant time `FindMin` to find the next element to be reported, after which we delete it by updating pointers $a_{0,j_0}, a_{1,j_1}, \ldots, a_{m,j_m}$, where $m = l - 1$, for nodes on the path from the root to the element. It costs time $O(2^{z-l})$ to find the new value of a_{m,j_m}, and then one can iteratively find, for $i = m-1, \ldots, 0$, the new value of a_{i,j_i} in time $O(\lceil 2^{z-2l+i} \rceil)$, i.e., the search times constitute an arithmetical progression (rounded up), yielding a total operation time of $O(2^{z-l} + l)$.

To accommodate `FindMin` in constant time, we use z bits to store a pointer to the current minimum (or to any element if the interval sorter is empty). □

This of course implies that for $T \geq n \lg n$ one can sort in time $O(T)$, using $O(n^2/T + w)$ bits of space.

B Associated memory in priority queues

In the randomized setting, the w bits of associated information can be retrieved by a dynamic dictionary that uses linear space and amortized constant time per operation w.h.p. However, we would like not to introduce randomization in our reduction. We will assume only that the priority queue has the three operations `Insert`, `FindMin` and `DeleteMin`. Assume for the moment that w is even. It is simple to extend this to the case of odd w.

We use two levels of priority queues. The level-one priority queue contains keys of the form ap, $a, p \in \{0,1\}^{w/2}$, where a is the first $w/2$ bits of some "original" key and p is a pointer unique to a. Regarding p as an integer, we can use it as a pointer into a length n array of level-two priority queues. The level-two priority queue corresponding to a contains keys of the form $a'p'$, $a', p' \in \{0,1\}^{w/2}$, where a' is the last $w/2$ bits of an original key having a as the first $w/2$ bits and p' is a pointer unique to a'. Now p' can be used as a pointer to an array of up to n words of associated information.

When inserting, pointers need to be allocated, for which purpose a standard free-list is used. If no free space exists, the array is expanded by the usual doubling technique. It should be clear that `FindMin` and `DeleteMin` can be done with two priority queue calls plus a constant overhead. To keep space down to $O(nw)$ bits one needs to shrink arrays if a large constant fraction of the elements are deleted from a priority queue. This is done in a standard way, and we do not go into details.

Recent BRICS Report Series Publications

RS-01-2 **Rasmus Pagh and Jakob Pagter.** ***Optimal Time-Space Trade-Offs for Non-Comparison-Based Sorting.*** **January 2001. 20 pp.**

RS-01-1 **Gerth Stølting Brodal, Anna Östlin, and Christian N. S. Pedersen.** ***The Complexity of Constructing Evolutionary Trees Using Experiments.*** **2001.**

RS-00-52 **Claude Crépeau, Frédéric Légaré, and Louis Salvail.** ***How to Convert a Flavor of Quantum Bit Commitment.*** **December 2000. 24 pp. To appear in** ***Advances in Cryptology: International Conference on the Theory and Application of Cryptographic Techniques*****, EUROCRYPT '01 Proceedings, LNCS, 2001.**

RS-00-51 **Peter D. Mosses.** ***CASL for CafeOBJ Users.*** **December 2000. 25 pp. Appears in Futatsugi, Nakagawa and Tamai, editors,** ***CAFE: An Industrial-Strength Algebraic Formal Method*****, 2000, chapter 6, pages 121–144.**

RS-00-50 **Peter D. Mosses.** ***Modularity in Meta-Languages.*** **December 2000. 19 pp. Appears in** ***2nd Workshop on Logical Frameworks and Meta-Languages*****, LFM '00 Proceedings, 2000.**

RS-00-49 **Ulrich Kohlenbach.** ***Higher Order Reverse Mathematics.*** **December 2000. 18 pp.**

RS-00-48 **Marcin Jurdziński and Jens Vöge.** ***A Discrete Stratety Improvement Algorithm for Solving Parity Games.*** **December 2000.**

RS-00-47 **Lasse R. Nielsen.** ***A Denotational Investigation of Defunctionalization.*** **December 2000. Presented at** ***16th Workshop on the Mathematical Foundations of Programming Semantics*****, MFPS '00 (Hoboken, New Jersey, USA, April 13–16, 2000).**

RS-00-46 **Zhe Yang.** ***Reasoning About Code-Generation in Two-Level Languages.*** **December 2000.**

RS-00-45 **Ivan B. Damgård and Mads J. Jurik.** ***A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System.*** **December 2000. 18 pp. Appears in Kim, editor,** ***Fourth International Workshop on Practice and Theory in Public Key Cryptography*****, PKC '01 Proceedings, LNCS 1992, 2001, pages 119–136. This revised and extended report supersedes the earlier BRICS report RS-00-5.**