Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space

Yijie Han*
School of Interdisciplinary Computing and Engineering
University of Missouri at Kansas City
5100 Rockhill Road
Kansas City, MO 64110
hanyij@umkc.edu
http://welcome.to/yijiehan

Mikkel Thorup
AT&T Labs— Research
Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932
mthorup@research.att.com

Abstract

We present a randomized algorithm sorting n integers in $O(n\sqrt{\log\log n})$ expected time and linear space. This improves the previous $O(n\log\log n)$ bound by Anderson et al. from STOC'95.

As an immediate consequence, if the integers are bounded by U, we can sort them in $O(n\sqrt{\log\log U})$ expected time. This is the first improvement over the $O(n\log\log U)$ bound obtained with van Emde Boas' data structure from FOCS'75.

At the heart of our construction, is a technical deterministic lemma of independent interest; namely, that we split n integers into subsets of size at most $\sqrt{n}$ in linear time and space. This also implies improved bounds for deterministic string sorting and integer sorting without multiplication.

1 Introduction

Integer sorting has always been an important task in connection with the digital computer. A classic example is the folklore algorithm radix sort, which according to Knuth [28] is referenced as far back as in 1929 by Comrie in a document describing punched-card equipment [14].

Whereas radix sort works in linear time for $O(\log n)$-bit integers, it was not until 1990 that Fredman and Willard [17] beat the $\Omega(n\log n)$ comparison-based sorting lower-bound for the case of arbitrary single word integers. The word-size W is determined by the processor. We assume $W \geq \log n$ so that we can address the different integers, but in principle, W can be arbitrarily large compared with n. An equivalent formulation of our assumptions is that we only assume constant time operations on integers polynomial in the sum of the input integers. The assumption that each integer fits in a machine word implies that integers can be operated on with single instructions. A similar assumption is made for comparison based sorting in that an $O(n\log n)$ time bound requires constant time comparisons. However, for integer sorting, besides comparisons, we can use all the other instructions available on integers in standard imperative programming languages such as C [26]. It is important that the word-size W is finite, for otherwise, we can sort in linear time by a clever coding of a huge vector-processor dealing with n input integers at the time [30, 27].

Concretely, Fredman and Willard [17] showed that we can sort deterministically in $O(n\log n/\log\log n)$ time and linear space and randomized in $O(n\sqrt{\log n})$ expected time and linear space. The randomized bound can also be achieved deterministically using space unbounded in terms of n (of the form $2^{\varepsilon W}$ for constant $\varepsilon > 0$), but here we focus on bounds with space bounded in terms of n.

In 1995, Andersson et al. [5] improved Fredman and Willard's $O(n\sqrt{\log n})$ expected time for integer sorting to $O(n\log\log n)$ expected time. Both of these bounds use linear space. A similar result was found independently by Han and Shen [23].

The above mentioned bounds are the best *unrestricted bounds* in the sense that no better bounds are known even if besides the randomization, we have unlimited space and are free to define our own operations on words.

The above results have provided great inspiration for many researchers, trying to improve them in various ways. For example, there has been work on avoiding randomization [4, 21, 22, 31, 33] and there has been work on avoiding multiplication [3, 6, 36]. Also there has been lots of work on more dynamic versions of the problem such as priority queues [13, 18, 31, 33, 34] and searching [3, 4, 8, 9].

The $O(n\log\log n)$ algorithm of Andersson et al. from 1995 [5] is very simple, and the fact that it has sustained so

*Supported in part by University of Missouri Faculty Research Grant K211942

much interest has lead many researchers to think that this could be the complexity for integer sorting, like $O(n \log n)$ is the complexity of comparison based sorting.

However, in this paper, we improve the $O(n \log\log n)$ expected time to $O(n\sqrt{\log\log n})$ expected time:

Theorem 1 *There is a randomized algorithm sorting n integers, each stored in one word, in $O(n\sqrt{\log\log n})$ expected time and linear space.*

We leave open the problems of getting a corresponding deterministic algorithm and avoid the use of multiplication. We note that the dynamic aspects are already pretty settled, for the complexity of dynamic searching is known to be $\Theta(\sqrt{\log n/\log\log n})$ [8, 9], and priority queues have once and for all been reduced to sorting [35].

Since integers of $O(\log n)$ bits can be sorted in linear time using radix sort, we get the following immediate consequence of Theorem 1:

Corollary 2 *We can sort n integers of size at most U in $O(n\sqrt{\log\log U})$ expected time.*

This is the first improvement over the $O(n \log\log U)$ bound obtained with van Emde Boas' data structure from 1975 [37]. Indeed, the $O(n \log\log n)$ sorting algorithm of Andersson et al. [5] combines van Emde Boas' data structure with the packed merging of Albers and Hagerup [2] so as to match the $O(n \log\log U)$ bound for large values of U.

We note here that the general $O(\log\log U)$ bound of van Emde Boas [37] has been improved in the context of static search structures. More precisely, Beame and Fich [9] have shown that one can preprocess a set of n integers in polynomial time and space so that given any x, one can search the largest stored integer below x in $O(\log\log U/\log\log\log U)$ time. However, due to the polynomial construction time, this improvement does not help with sorting. Beame and Fich [9] combine their result with Andersson's exponential search trees from [4], giving a dynamic search structure with an amortized update time of $O(\log\log U \log\log n/\log\log\log U)$. This dynamic search structure could be used for sorting in $O(n \log\log U \log\log n/\log\log\log U)$ time, but this is never better than the best of $O(n \log n)$ time and $O(n \log\log U)$ time. Thus, from the perspective of sorting, our $O(n\sqrt{\log\log U})$ bound constitutes the first improvement over the $O(n \log\log U)$ bound derived from van Emde Boas' data structure from 1975 [37].

We will, in fact, prove the following refinement of Corollary 2:

Theorem 3 *There is a randomized algorithm sorting n word integers, each of size at most $U < 2^W$ in $O(n\sqrt{\log \frac{\log U}{\log n}})$ expected time and linear space.*

Theorem 3 improves a corresponding refinement of Kirkpatrick and Reich [27] of van Emde Boas' bound of $O(n \log \frac{\log U}{\log n})$ expected time.

The following simple lemma states that it suffices for us to prove Theorem 3:

Lemma 4 *Theorem 3 implies Theorem 1 and Corollary 2.*

Proof: Trivially Theorem 3 implies the weaker Corollary 2. However, Andersson et al. [5] have shown that we can sort in linear expected time if $W \geq (\log n)^3$, and otherwise, $\log\log U \leq \log W \leq 3 \log\log n$. ∎

1.1 Other domains

In this paper, we generally assume that our integers to be sorted each fit in one word, where a word is the maximal unit that we can operate on with a single instruction. In this subsection, we briefly discuss implications of this case to many other domains.

First, consider the case of lexicographically ordered strings of words. Andersson and Nilsson [7] have presented an optimal randomized reduction from this case to that of integers fitting in single words. Applying their reduction to Theorem 1, we get

Corollary 5 *We can sort n variable-length strings distributed over N words in $O(n\sqrt{\log\log n} + N)$ expected time. In fact, we can get down to $O(n\sqrt{\log\log n} + L)$ expected time where $L = \sum_i \ell_i$ and ℓ_i is the length of the distinguishing prefix of string i, that is, the smallest prefix of string i distinguishing it from all the other strings, or the length of string i if it is a duplicate.*

We note here that any algorithm sorting the strings will have to read the distinguishing prefixes, and since it takes an instruction to read each word, it follows that an additive $O(L)$ is necessary.

One may instead be interested in variable length multiple-word integers where integers with more words are considered bigger. However, by prefixing each integer with its length, we reduce this case to lexicographic string sorting.

In our presentation, we think of all our integers as unsigned non-negative integers. However, the standard representation of signed integers is such that if we flip the sign-bit, we can sort them as unsigned integers, and then flip the sign-bit back again. Floating points numbers are even easier, for the IEEE 754 floating-point standard [25] is designed so that the ordering of floating point numbers can be deduced by perceiving their representations as multiple word integers. Also, if we are working with fractions where both enumerator and denominator are single word integers,

we get the right ordering if for each fraction, we make the division in floating point numbers with double precision. Now we get the correct ordering of the original integer fractions by perceiving the corresponding floating point numbers as integers.

1.2 Machine model

Recall that our machine model is a normal computer with an instruction set corresponding to what we program in a standard programming language such as C [26] or C++ [32]. We have a processor determined word-size W, limiting how big integers we can operate on in constant time. We assume that each input integer fits in a single word. We note that for generic code, the type of a full word integer, e.g. `long long int`, should be a macro parameter in C or template parameter in C++. We adopt the unit-cost time measure where each operation takes constant time.

Interestingly, the traditional theoretical RAM model of Cook and Reckhow [15] allows infinite words. A disturbing consequence of infinite words is that with normal operations such as shifts or multiplication, we can simulate an exponentially big parallel processor solving all problems in NP in polynomial time. Hence such operations have to be banned from the above unit-cost theory RAM, making it even more contrived from a practical view-point.

However, by adopting the real-world limitation of a limited word-size, we both resolve the above theoretical issue, and we get algorithms that can be implemented in the real world. Hagerup [19] has named this model the *word RAM*. The word RAM has a fairly long tradition within integer sorting, being advocated and used in the 1984 paper [27] by Kirkpatrick and Reisch, and in the seminal 1993 paper of Fredman and Willard [17].

We note that our unit-cost multiplication may be considered somewhat questionable in that multiplication is not in AC^0 [10], that is, there is no multiplication circuit of constant-depth and of size polynomial in W. We leave it as an open problem to improve Thorup's randomized $O(n \log\log n)$ expected time and linear space sorting without multiplication [36].

We will now discuss some of the (delightfully) dirty tricks that we can use on the word RAM, and which are not allowed in the comparison based model or on a pointer machine.

A first nice feature of the word RAM over the comparison based model is that we can add and subtract integers. For example, we can use this to code multiple comparisons of short integers packed in single words. The idea of multiple comparisons was first introduced by Paul and Simon [30] in 1980. It should be noted that this use of uniprocessors as vector processors is a standard trick in practice, not in connection with sorting, but in connection with graphics, where a single word operation is used to manipulate the information on several pixels, each represented by one byte of the word.

The word RAM model distinguishes itself both from the comparison based model and from the pointer machine in that we can use integers, and segments of integers, as addresses. This trick goes back to radix sort where an integer is viewed as a vector of characters, and these characters are used as addresses. Another word RAM trick in this direction is that we can hash integers into smaller ranges. Here radix sort goes back at least to 1929 [14] and hashing goes back at least to 1956 [16], both being developed for efficient problem solving in the real world.

Fredman and Willard [17] further use the RAM for advanced tabulation of complicated functions over small domains. Their tabulation is too complicated to be of practical relevance, but tabulation of functions is in itself commonly used to tune code. As a simple example, Bentley [11, pp. 83–84] suggests that an efficient method for computing the number of set bits in 32-bit integers is to have a preprocessing where we first tabulate the number of set bits in all the 256 different 8-bit integers. Now, given a 32-bit integer x, we view it as the concatenation of four 8-bit integers, and for each of these, we look up the number of set bits in our table. Finally we just add up these four numbers to get the number of set bits in x.

Summing up, we have argued that the "dirty tricks" facilitated by the word RAM are well established in the practice of writing fast code. Hence, if we disallow these tricks, we are not discussing the time complexity of running imperative programs on real world computers. At the same note, it should be admitted that this is a theory paper. The algorithms presented are too complicated and have too large constants hidden in the O-notation to be of any immediate practical use. This does not preclude that some of the ideas may find use in practice. For example, Nilsson [29] has demonstrated that the $O(n \log\log n)$ algorithm of Andersson et al. [5] can be implemented so as to be competitive with the best practical sorting algorithms.

1.3 Deterministic splitting and sorting

The heart of our construction is a deterministic splitting results of independent interest.

Definition 6 *A* splitting *of an ordered set X is a partitioning into sets $X_0 < X_1 < \cdots < X_k$. Here, $A < B$ denotes that $a < b$ for all $(a, b) \in A \times B$.*

We are generally thinking of sets as multisets. If all elements of a set are identical, we call it a *duplicate set*; otherwise, we call it a *diverse set*.

Theorem 7 *We can split a set of n word integers in linear time and space so that each diverse subset has at most $\sqrt{n}$ integers.*

Applying Theorem 7 recursively, we immediately get that we can sort deterministically in $O(n \log\log n)$ time and linear space, but this has already been proved by Han in [22]. However, for deterministic string sorting, we get the following new result which does not follow from [22] (the general reduction of Andersson and Nilsson [7] from word sorting to string sorting is randomized):

Corollary 8 *We can sort n variable-length strings distributed over N words in $O(n \log\log n + N)$ time and linear space. In fact, we can get down to $O(n \log\log n + L)$ time where L is the sum of the lengths of the distinguishing prefixes.*

Proof: To get the corollary, we simply apply Theorem 7 recursively but only to the first unmatched word of each string. That is, our recursive input is a subset of the integers with some common matched prefix. In the root call, the subset is the complete set with nothing matched so far. Integers ending up in a duplicate set match in one more word, and the other integers end up in sets of size reduced to the square root. Since the splitting takes constant time per integer, we pay constant time per word matched and $O(\log\log n)$ time for reductions into smaller sets. ∎

We also present variants of the splitting using only standard AC^0 operations, that is, AC^0 operations available via a standard programming language such as C or C++.

Theorem 9 *For any positive ε, using standard AC^0 operations only, we can split a set of n $(W/(\log\log n)^{1+\varepsilon})$-bit integers in linear time so that diverse subsets have size at most $\sqrt{n}$.*

Corollary 10 *For any positive ε, using standard AC^0 operations only, we can sort n words in $O(n(\log\log n)^{1+\varepsilon})$ time and linear space.*

Proof: We use the same proof as for Corollary 8, but viewing each word as a string of $(W/(\log\log n)^{1+\varepsilon})$-bit characters, to which we can apply Theorem 9. ∎

Corollary 10 improves the previous $O(n(\log\log n)^2/(\log\log\log n))$ bound of Han from [20], and it gets very close to the best multiplication based deterministic bound of $O(n \log\log n)$ [22].

1.4 Contents

The rest of the paper is divided into two sections. Section 2 presents our randomized sorting assuming deterministic splitting, and Section 3 presents our deterministic splitting.

2 Fast randomized sorting

For our fast randomized sorting, we need a slight variant of the signature sorting of Andersson et al. [5] (cf. Appendix A.1):

Lemma 11 *With an expected linear-time additive overhead, signature sorting with parameter r reduces the problem of sorting n integers of ℓ bits to*

(i) the problem of sorting n reduced *integers of $4r \log n$ bits each and*

(ii) the problem of sorting n integer fields *of ℓ/r bits each.*

Here (i) has to be solved before (ii).

We are now going to present our randomized algorithm for sorting n word integers in linear space and $O(n\sqrt{\log p})$ expected time where $p = \log U/\log n$.

Repeated splitting First we apply our splitting from Theorem 7 to recursively split the diverse set of integers $\lceil\sqrt{\log p}\rceil$ times, thus getting a splitting with diverse subsets of size at most $n' = n^{1/2^{\sqrt{\log p}}}$. Each integer is involved in at most $\lceil\sqrt{\log p}\rceil$ linear time splittings, so the total time is $O(n\sqrt{\log p})$.

Repeated signature sort To each of the diverse sets S_i of size at most n', we apply the signature sort from Lemma 11 with $r = 2^{\sqrt{\log p}}$. Then the reduced integers from (i) have $4r \log n' = O(\log n)$ bits.

We are going to sort these reduced integers from all the subsets S_i together, but prefixing reduced integers from S_i with i. The prefixed reduced integers still have $O(\log n)$ bits, so we can radix sort them in linear time. From the resulting sorted list we can trivially extract the sorted sublist of reduced integers for each S_i, thus completing task (i) from Lemma 11.

We have now spent linear expected time on reducing the problem to dealing with the fields from Lemma 11 (ii) and the field length is only a fraction $1/2^{\sqrt{\log p}}$ of original length.

We repeat this signature sorting $\lceil\sqrt{\log p}\rceil$ times, at a total expected cost of $O(n\sqrt{\log p})$. We started with integers of length $\log U$, and each round reduces the integer length by a factor $2^{\sqrt{\log p}}$, so we end up with integers of length at most

$$\log U/(2^{\sqrt{\log p}})^{\sqrt{\log p}} \leq \log U/2^{\log p} = \log n$$

Since the lengths are now at most $\log n$, we can trivially finish with a linear time bucket sort.

Summing up The total expected time spent in the above algorithm is $O(n\sqrt{\log p})$, and we have only used linear space, as desired.

Thus, our only remaining task is to provide the splitting from Theorem 7, that is,

Lemma 12 *Theorem 7 implies Theorem 3.* ∎

3 Deterministic splitting

To prove Theorem 7, first in Section 3.1, we reformulate it in terms of splitting over a given set of splitters.

3.1 Splitting over splitters

We will actually prove our splitting result in terms of an equivalent formulation. A splitting of a set X into sets $X_0 < X_1 < \cdots < X_k$ is a *splitting over k splitters* $y_1 < y_2 < \cdots < y_k$ if $X_0 < \{y_1\} \leq X_1 < \{y_2\} \leq X_3 \cdots < \{y_k\} \leq X_k$.

Lemma 13 *The following statements are equivalent:*

(a) We can split a set of n integers so that diverse subsets are of size at most $n^{1-\varepsilon_a}$ in linear time and space for some positive constant ε_a.

(b) We can split a set of n integers so that diverse sets are of size at most $n^{1-\varepsilon_b}$ in linear time and space for any positive constant ε_b.

(c) We can split a set of n words over n^{ε_c} splitters in linear time and space for any positive constant ε_c.

(d) We can split a set of n words over n^{ε_d} splitters in linear time and space for some positive constant ε_d.

Proof: **(a)⇒(b)** To get diverse subsets of size $n^{1-\varepsilon_b}$ we apply (a) recursively on diverse sets. If (a) is applied i times to subsets containing x, the set containing x ends up non-diverse, or of size at most $n^{(1-\varepsilon_a)^i}$. Thus x gets involved at most $\log_{1-\varepsilon_a}(1-\varepsilon_b) = O(1)$ times.

(b)⇒(c) To prove (c) with any given value of ε_c, we apply (b) with $\varepsilon_b = 2\varepsilon_c$. Now, the diverse subsets have at most $n^{1-2\varepsilon_c}$ integers. Out of these subsets, at most n^{ε_c} contain a splitter. We split each subset with splitters using traditional comparison based sorting. The total number of integers involved in this is at most $n^{1-2\varepsilon_c} n^{\varepsilon_c} = n^{1-\varepsilon_c}$, so the total time for sorting is $O(n^{1-\varepsilon_c} \log n) = O(n)$. Having sorted all diverse subsets with splitters over these splitters, we immediately derive the desired splitting of the original set.

(c)⇒(d) is trivial.

(d)⇒(a) If $n = O(1)$, (a) is trivial, so assume $n = \omega(1)$. We divide our input integers into batches of size $a\sqrt{n}$. Using (d), we can split such a batch over $n^{\varepsilon_d/2}$ splitters in linear time.

We will develop the an appropriate set Y of splitters as we go along, starting with $Y = \emptyset$. Each time we come with a batch of integers, we split them according to the current splitters $y_1 < y_1 < \cdots < y_{k-1}$, adding them to the splitting $X_0 < \{y_1\} \leq X_1 < \{y_2\} \cdots < \{y_k\} \leq X_k$ done so far. If one of the diverse subsets X_i gets $4n^{1-\varepsilon_d/2}$ or more elements, we split it according to its median z, and make z and $z+1$ two new splitters. Obviously, we end with at most $4n^{1-\varepsilon_d/4}$ splitters in each diverse set, and since $n = \omega(1)$, $4n^{1-\varepsilon_d/4} \leq n^{1-\varepsilon_a}$ for some positive constant ε_a.

We will charge each splitting over a median to $2n^{1-\varepsilon_d/2}$ elements. This implies that the median finding and subsequent splitting is done in linear time, and that the total number of splitters is at most $2n/(2n^{1-\varepsilon_d/2}) = n^{\varepsilon_d/2}$, as needed for applying (d) with a batch of $\sqrt{n}$ integers.

The charging is simple: every time a diverse subset is started, it has at most $2n^{1-\varepsilon_d/2}$ charged elements. We only split the set if it gets more than $4n^{1-\varepsilon_d/2}$ elements, which means that we have at $2n^{1-\varepsilon_d/2}$ non-charged elements that we can charge, and we can easily distribute the charging so that each of the resulting divers sets get at most Moreover, any diverse subset starts with at most $2n^{1-\varepsilon_d/2}$ charged elements. ∎

The rest of this paper is devoted to prove Lemma 13 (d) with $\varepsilon_d = \sqrt[3]{n}$, which by Lemma 13 (b) implies Theorem 7. That is, our remaining task is to split n integers over $\sqrt[3]{n}$ splitters in linear time and space.

3.2 Deterministic signature sorting

We shall use a deterministic version of signature sort, essentially done by Han [21] (cf. Appendix A.2).

Lemma 14 *With an $O(n+\frac{\ell}{r}s^2)$ time additive overhead, deterministic signature sorting with parameter r reduces the problem of splitting n ℓ-bit integers over s splitters to*

(i) the problem of splitting n reduced integers of $4r\log n$ bits over s reduced splitters, and

(ii) the problem of splitting n fields of ℓ/r bits over s field splitters.

Here (i) has to be solved before (ii).

Han [21] actually paid an additive $O(\ell s^2)$. The fact that we only pay $O(\frac{\ell}{r}s^2)$ is useful if ℓ is arbitrarily large compared with n as in Lemma 15 below.

Lemma 15 *If $W = (\log n)^c$, $c \geq 5$, we can split n word integers over at most $\sqrt[3]{n}$ splitters in linear time.*

Proof: We apply the signature sorting of Lemma 14 with $r = (\log n)^{c-3}$. Then the additive overhead is

$$O\left(n + \frac{(\log n)^c}{(\log n)^{c-3}}\sqrt[3]{n^2}\right) = O(n).$$

Now, the reduced integers from (i) have length

$$4r(\log n) = 4(\log n)^{c-2} = O(W/(\log n)^2)$$

and that implies that we can sort and hence split them in linear time using the packed sorting of Albers and Hagerup [2]. Similarly, the fields have length $(\log n)^2 = O(W/(\log n)^3)$, so they can also be sorted in linear time. ■

Above, we could let c go down from 5 to 3 if, instead of using the packed sorting of Albers and Hagerup, we used the one by Han and Shen [24] which takes linear time for $O(W/\log n)$-bit integers. However, Han and Shen's result employs the sorting network of Ajtai et al. [1], and the improvement does not affect the overall results of this paper.

Lemma 16 *If $W \leq (\log n)^5$, with a linear time additive overhead, we can reduce the problem of splitting n word integers over $s \leq \sqrt[3]{n}$ splitters into at most four problems of splitting $q(\log n)$-bit integers over s splitters where $q = O(\log n)$ and $W = \Omega(q^4(\log n))$.*

Proof: Let $p = W/\log n \leq (\log n)^4$. First we apply the deterministic signature sort of Lemma 14 with $r = \sqrt{p}$. Now both subproblems have integer length $\sqrt{p}(\log n)$. We then apply Lemma 14 with $r = \sqrt[4]{p}$ to each subproblem, getting four subproblems, each with integer length $O(\sqrt[4]{p}(\log n))$. ■

By the two preceeding lemmas, we can assume that the integers are of length $q(\log n)$ where $q = O(\log n)$, $W = \Omega(q^4(\log n))$, and $W \leq (\log n)^5$.

3.3 String sorting

We are going to show:

Lemma 17 *Consider $n \geq W^6$ integers packed with at most $k(\log k)$ integers in each word. We can sort the integers according to the value of a given segment of at most $(\log n)/2$ consecutive bits so that the time spent on an integer with segment value c is*

$$O\left(\frac{1 + \log n - \log n_c}{\log n} + 1/k\right)$$

where n_c is the number of integers with segment value c.

The above lemma may look somewhat strange, but as demonstrated below, it actually implies our main result.

Lemma 18 *Lemma 17 implies Lemma 13 (d) with $\varepsilon_d = \sqrt[3]{n}$.*

Proof: We need to split n integers over $\sqrt[3]{n}$ splitters. From Lemmas 15 and 16, we know that it suffices to consider $q(\log n)$-bit integers where $q = O(\log n)$, $W = \Omega(q^4(\log n))$, and $W \leq (\log n)^5$. For $n = \omega(1)$, the latter implies $W^6 \leq n$, as required for Lemma 17.

We view each integer as consisting of $4q$ characters, each of $(\log n)/4$ bits. Also, we have plenty of room to pack $q(\log q)$ integers in each word.

The algorithm works recursively, taking a subset of $n' \geq \sqrt{n}$ integers, all with a common prefix of length i. We then use Lemma 17 with $k = q$ to sort the integers according to character $i+1$. We note that this character has $(\log n)/4 \leq (\log n')/2$ bits, as required of the segment in Lemma 17. Also, we note that this is, in fact, a splitting since all the integers agree on the preceding characters. Starting with all n integers, we recurse on diverse subsets until they all have size at most $\sqrt{n}$.

To see that this implies a linear time splitting, let $n_0, n_1, ..., n_t$ be the sizes of the sets that a given integer x is involved in as. That is, we start with $n_0 = n$. For round i, we have $n_{i-1} \geq \sqrt{n}$ integers, and we match $(\log n)/4 \leq (\log n_{i-1})/2$ bits of x, finding agreement with n_i other integers. Hence, the cost for x is

$$O\left(\frac{1 + \log n_{i-1} - \log n_i}{\log \sqrt{n}} + 1/q\right)$$

Summing this for $i = 1, ..., t$, we get a total cost for x of

$$O\left(\frac{\log n_0 - \log n_t}{\log \sqrt{n}} + t/\log \sqrt{n} + t/q\right).$$

Here $n_0 = n$ so the first term is constant. Moreover, by definition, there are $4q = O(\log n)$ characters, and using this upper-bound on t, we see that the last two terms are constant. ■

3.4 Sorting over a segment

The goal of this section is to prove Lemma 17. We will use the lemma below on packed bucketing, essentially proved by Han in [21] (cf. Appendix A.3).

Lemma 19 *Consider n integers packed with $k(\log k)$ integers in each word, and that an ℓ-bit label for each integer is packed in a parallel set of words. We can then sort the integers according to their labels in $O(\ell/\log n + 1/k)$ time per integer.*

Lemma 20 *Consider $n \geq W^6$ integers packed with at least $k(\log k)$ integers in each word. We can group all integers with respect to matches with $t \leq \sqrt[3]{n}$ target integers within a given segment in*

$$O\left(\log(t+1)/\log n + 1/k\right)$$

time per integer. Integers not matching any target integer end in one group.

Proof: Our goal is to construct a parallel set of labels so that we can apply Lemma 19. First we assume that $t \geq 2$.

In $O(t^2W)$ time, we construct a perfect hash function from the segments of the targets into $\lceil 2 \log t \rceil$ bits. Since $k = O(W)$, the time spent is $O(n/k)$.

To apply the hash function to all our integers, we first make copies of the words containing them into a parallel set of words, then we mask out the segments, shifting them to the least significant part of each integer. Finally, we apply the hash function so that we now have a parallel set of $\lceil 2 \log t \rceil$ bit labels, each aligned with the least significant part of its original integer. We generally refer the reader to [5, pp. 77–79] for details on making such word operations on multiple integers in a word, including the hashing. We only spend constant time per word, hence $O(1/(k \log k))$ time per integer.

We now apply Lemma 19, getting the integers sorted according to their labels in $O(\log t/\log n + 1/k)$ time per integer. We have $\Theta(t^2)$ labels, and exactly one label for each target. Fix a_0 to be a label which is not a label of a target. For each target y, we pack $k(\log k)$ copies of y in a word y^*. This takes $O(tk(\log k)) = O(n/k)$ total time.

The integers in the words are sorted according to their labels, and comparing this with the sorted list of target labels, we can easily identify words of integers where all or some of the parallel labels match some target label.

For each word all of whose labels match the label of a target y, we compare the parallel word of integers with y^*. For each non-match, the corresponding label is replaced with a_0. All this takes constant time per word. For words having no target labels, all labels are replaced by a_0. There can be at most $2t$ words with only some labels being target labels. These have at most $2tk \log k = O(n/k)$ parallel integers, and for each such integer, we can trivially, in constant time, replace its label with a_0 if it doesn't match a target.

We now re-apply Lemma 19, getting the integers sorted according to their revised labels. However, this time the sorting gives the desired grouping. The segment of integers with label a_0 are exactly those that do not match any target.

Finally, we have two special cases of $t = 0, 1$. The case $t = 0$ is trivial in that all integers belong in the non-matching group, and hence nothing has to be done, agreeing with $\log(t+1) = 0$ in the time bound. For $t = 1$, we copy the unique target so that all integers in a word can be matched in constant time. We use a 1-bit label with 1 for match and 0 for non-match, and finally apply Lemma 19. Since $\log(t+1) = 1$, this again gives us the desired time bound. ∎

The lemma below essentially shows that if we had guessed the frequencies, we would be done.

Lemma 21 *Consider $n \geq W^6$ integers packed with $k(\log k)$ integers in each word. We are focusing on a specific segment of the integers. Suppose that for different possible segment values c, we are given a suggested frequency $f_c \geq 1/\sqrt[3]{n}$ for integers with that segment value. We further require $\sum_{f_c} \leq 1$. We can then group the integers spending*

$$O\left((1 - \log f_c)/\log n + 1/k\right)$$

time per integer with segment value c.

Proof: The result is achieved by a simple iterative algorithm. First we sort the frequencies in descending order. We start with $t = \lceil n^{1/k} \rceil \geq 2$ and repeat squaring t until it reaches or passes $n^{1/3}$. For a given value of t, we take the t remaining segment values of highest frequency, and apply Lemma 20 to the remaining integers. This groups the integers matching the t segment values, leaving the remaining integers for the remaining rounds.

The time spent per integer in a round is $O(\log(t+1)/\log n + 1/k) = O(\log t/\log n)$. Since $\log t/\log n$ doubles in each round, it is the last round that an integer x participates in that dominates the time spent on that x. However, if the integer has segment value c with frequency f_c, there can be no more than $1/f_c$ earlier frequencies, so the value of t when x is picked is at most $1/f_c^2$, or $\lceil n^{1/k} \rceil$ if $1/f_c \leq n^{1/k}$. Consequently, the total time spent on x is

$$\begin{aligned} &O(\log \max\{1/f_c^2, \lceil n^{1/k} \rceil\}/\log n + 1/k) \\ &= O((1 - \log f_c)/\log n + 1/k). \end{aligned}$$

∎

Finally, we have

Proof of Lemma 17 We want to prove

> *Consider $n \geq W^6$ integers packed with at most $k(\log k)$ integers in each word. We can sort the integers according to the value of a given segment of at most $(\log n)/2$ consecutive bits so that the time spent on an integer with segment value c is*
>
> $$O\left(\frac{1 + \log n - \log n_c}{\log n} + 1/k\right) \qquad (1)$$
>
> *where n_c is the number of integers with segment value c.*

Since there are only $n^{1/2} = O(n/\log n)$ possible segment values, we can initiate arrays using these values as entries. Thus, with each possible segment value c, we can store a counter $\hat{n}_c$ for the number of integers found with configuration c. Initially $\hat{n}_c = 0$. We also have a list of frequencies $\hat{f}_c = \hat{n}_c/n$ for segment values c that are common in the

sense that $\hat{n}_c \geq n^{3/4}$, hence with $\hat{f}_c \geq 1/n^{1/4}$. Initially, this list is empty.

We divide the integers into batches of $n^{3/4}$ integers. First we group the integers in the batch with respect to the common segment values using their current frequencies in Lemma 21. We note here that $\hat{f}_c = \hat{n}_c/n \geq 1/n^{1/4} = 1/\sqrt[3]{n^{3/4}}$, so the conditions of Lemma 21 are satisfied. The cost for an integer with a common segment value c is

$$\begin{aligned} &O((1 - \log f_c)/\log(n^{3/4}) + 1/k) \\ &= O((1 + \log(n/\hat{n}_c))/\log n + 1/k) \end{aligned} \quad (2)$$

All remaining integers in the batch are bucketed using standard bucketing in constant time per integer.

We will now argue that the time spent on inserting the ith integer in one of our buckets is

$$O\left(\frac{1 + \log n - \log i}{\log n} + 1/k\right). \quad (3)$$

If $i \leq 2n^{3/4}$, (3) is a constant, covering the cost of standard bucketing. Otherwise, since each batch adds at most $n^{3/4}$ integers, the batch was bucketed with $\hat{n}_c \geq i - n^{3/4} \geq i/2$. By (2), the cost is as in (3) but with $i/2$ instead of i, but this does not affect the asymptotic value. Thus, the cost of the ith integer is always bounded by (3).

Now, the cost of adding all n_c integers to the bucket for segment value c is

$$\begin{aligned} &\sum_{i=1}^{n_c} \left(\frac{1 + \log n - \log i}{\log n} + 1/k\right) \\ &= O\left(\frac{n_c(1 + \log n) - (n_c \log n_c - n_c(\log e))}{\log n} \right. \\ &\qquad \left. + n_c/k\right) \\ &= O\left(n_c \frac{1 + \log n - \log n_c}{\log n} + n_c/k\right), \end{aligned}$$

which divided by n_c gives the desired time per integer from (1). ∎

4 Summing up

Proof of Theorem 1, Corollary 2, and Theorem 3 The results follow directly from the statements of Lemma 4, 12, 13, 17, and 18. ∎

Proof Sketch for Theorem 9 and Corollary 10 We want to show

> *For any positive ε, using standard AC^0 operations only, we can split a set of n $(W/(\log\log n)^{1+\varepsilon})$-bit integers in linear time so that diverse subsets have size at most $\sqrt{n}$.*

We are can essentially reuse the splitting algorithm for Theorem 7. The only non-AC^0 operation used is multiplication, used for hashing. Brodnik et al. in [12] have shown (see their remark on BlockMult above Theorem 13) that if words are packed with ℓ bit integers, we can multiply them coordinatewise in $O((\log \ell)^{1+\varepsilon})$ time using only standard AC^0 operations. We do such packed multiplication on fields when we use signature sort in Lemma 15 and in Lemma 16, with field lengths of at most $(\log n)^2$ and $(\log n)^3$, respectively. Also, the rest of the algorithm only considers integers of size $O((\log n)^2)$. Thus the packed simulated multiplication takes $O((\log\log n)^{1+\varepsilon})$ time. However, since our input integers are of length $(W/(\log\log n)^{1+\varepsilon})$, the packed multiplications will only be over that many bits. Hence we can pack $\Omega((\log\log n)^{1+\varepsilon})$ original packed multiplications, thus performing $\Omega((\log\log n)^{1+\varepsilon})$ original packed multiplications at the time, in $O(1)$ time per original packed multiplications. ∎

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[2] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Comput.*, 136:25–51, 1997.

[3] A. Andersson. Sublogarithmic searching without multiplications. In *Proc. 36^{th} FOCS*, pages 655–663, 1995.

[4] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37^{th} FOCS*, pages 135–141, 1996.

[5] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comp. Syst. Sc.*, 57:74–93, 1998. Announced at STOC'95.

[6] A. Andersson, P. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theor. Comput. Sc.*, 215(1-2):337–344, 1999.

[7] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. 35^{th} FOCS*, pages 714–721, 1994.

[8] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 32nd STOC*, pages 335–342, 2000.

[9] P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proc. 31^{st} STOC*, pages 295–304, 1999.

[10] P. Beame and J. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM*, 36(3):643–670, 1989.

[11] J. Bentley. *Programming Pearls*. Addison-Wesley, Reading, Massachusetts, 1986.

[12] A. Brodnik, P. B. Miltersen, and I. Munro. Trans-dichotomous algorithms without multiplication - some upper and lower bounds. In *Proc. 5^{th} WADS, LNCS 1272*, pages 426–439, 1997.

[13] B. Cherkassky, A. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. *SIAM J. Comp.*, 28(4):1326–1346, 1999.

[14] L. J. Comrie. The hollerith and powers tabulating machines. *Trans. Office Machinary Users' Assoc., Ltd*, pages 25–37, 1929-30.

[15] S. Cook and R. Reckhow. Time-bounded random access machines. *J. Comp. Syst. Sc.*, 10(2):217–255, 1973.

[16] A. I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

[17] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1993. Announced at STOC'90.

[18] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48:533–551, 1994.

[19] T. Hagerup. Sorting and searching on the word RAM. In *Proc. 15^{th} STACS, LNCS 1373*, pages 366–398, 1998.

[20] Y. Han. Fast integer sorting in linear space. In *Proc. STACS'00, LNCS 1170*, pages 242–253, 2000.

[21] Y. Han. Improved fast integer sorting in linear space. *Inf. Comput.*, 170(8):81–94, 2001. Announced at STACS'00 and SODA'01.

[22] Y. Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. In *Proc. 34^{th} STOC*, 2002.

[23] Y. Han and X. Shen. Conservative algorithms for parallel and sequential integer sorting. In *Proc. 1^{st} COCOON, LNCS 959*, pages 324–333, 1995.

[24] Y. Han and X. Shen. Parallel integer sorting is more efficient than parallel comparison sorting onm exclusive write PRAMs. In *Proc. 10^{th} SODA*, pages 419–428, 1999.

[25] IEEE. Standard for binary floating-point arithmetic. *ACM Sigplan Notices*, 22:9–25, 1985.

[26] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[27] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theor. Comp. Sc.*, 28:263–276, 1984.

[28] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[29] S. Nilsson. *Radix Sorting & Searching*. Ph.D. Thesis, Lund University, Sweden, 1996.

[30] W. Paul and J. Simon. Decision trees and random access machines. In *Proc. Symp. über Logik and Algoritmik*, pages 331–340, 1980.

[31] R. Raman. Priority queues: small, monotone and trans-dichotomous. In *Proc. 4^{th} ESA, LNCS 1136*, pages 121–137, 1996.

[32] B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, Reading, MA, 2000.

[33] M. Thorup. Faster deterministic sorting and priority queues in linear space. In *Proc. 9^{th} SODA*, pages 550–555, 1998.

[34] M. Thorup. On RAM priority queues. *SIAM J. Comp.*, 30(1):86–109, 2000.

[35] M. Thorup. Equivalence between priority queues and sorting, 2002. Also submitted to FOCS'02.

[36] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Algor.*, 42(2):205–230, 2002. Announced at SODA'97.

[37] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. 16^{th} FOCS*, pages 75–84, 1975.

A Variants of results from other papers

In this appendix we justify some simple variants of results stemming from other papers.

A.1 Signature sorting

To describe our slight variant of signature sorting for Lemma 11, we first review the original signature sorting of Andersson et al. [5] is a set S of n integers of length ℓ. Signature sorting reduces the problem of sorting S to that of two sorting problems, each with n integers, but with shorter integers. In the reduction, for some parameter r, we interpret the integers in S as vectors of r equal sized fields. The reduction goes in several steps:

1. For each integer, each field is hashed into $4 \log n$ bits. The hashed fields of an integer are packed together giving us a reduced integer of length $4r \log n$ bits. All the reduced integers are produced in linear total time.

2. The reduced integers are now sorted (our first new sorting problem).

3. In linear time, we identify n fields from the integers in S.

4. The n fields of ℓ/r bits are now sorted (our second new sorting problem).

5. Based on the sorting done above, we sort S in linear time.

The above high level reduction is carefully implemented in [5], to which the reader is referred for details. There is a probability of at most $1/n^2$ that something goes wrong in the hashing. In [5] they just check the final sorting in the end. However, here we will apply signature sorting to many small subproblems, a small fraction of which are likely to fail. Instead of aiming at no errors at all, we introduce the following convenient step between step 2 and step 3.

$2\frac{1}{2}$. In expected linear time, redo and resort the reduced integers if they are not OK.

To implement step $2\frac{1}{2}$, we need to check that the reduced integers are OK. Referring the reader to [5] for details, this is easily done in connection of their linear time construction of a certain compressed unordered trie T_D, needed for steps 3-4. If there is a failure, we return to step 1. However, when we iterate, we can just use bubble-sort to sort the reduced integers in $O(n^2)$. The point is that the probability of iterating is $1/n^2$, so the expected cost of all the iterations is bounded by $O(n^2) \sum_{i=1}^{\infty} n^{-2i} = O(1)$. The most expensive part of step $2\frac{1}{2}$ is therefore the first check which is always executed in linear time.

Summing up, with an expected linear-time additive overhead, signature sorting with parameter r reduces the problem of sorting n integers of ℓ bits to

(i) the problem in step 2 of sorting n reduced integers of $4r \log n$ bits and

(ii) the problem in step 4 of sorting n fields of ℓ/r bits.

Here (i) has to be solved before (ii). This establishes our variant of signature sorting from Lemma 11.

A.2 Deterministic signature sorting

We want to show the statement of Lemma 14:

> *With an $O(n + \frac{\ell}{r}s^2)$ time additive overhead, deterministic signature sorting with parameter r reduces the problem of splitting n ℓ-bit integers over s splitters to*
>
> *(i) the problem of splitting n reduced integers of $4r \log n$ bits over s reduced splitters, and*
>
> *(ii) the problem of splitting n fields of ℓ/r bits over s field splitters.*
>
> *Here (i) has to be solved before (ii).*

Lemma 14 is essentially shown by Han in the beginning of Section 8 in [20]. A small difference is in the additive $O(\frac{\ell}{r}s^2)$ term, where Han has $O(\ell s^2)$. This term is the time it takes to compute a perfect hash function. We just realize that all we need is a hash function on fields such that for any pair of splitters, the hash function should give different values on their first distinguishing field. Since fields have length ℓ/r, the hash function is computed in $O(\frac{\ell}{r}s^2)$ time using simple derandomization as described by Raman [31]. Moreover, Han's version could lead to much more than s splitters in (ii). We need an analog of a simple trick from the original signature sort [5, p. 79]; namely, at each branching point in the trie T_D, to isolate integer fields smaller than the smallest splitter field in linear time. This completes the proof of Lemma 14.

A.3 Packed bucketing

We want to show the statement of Lemma 19:

> *Consider n integers packed with $k(\log k)$ integers in each word, and that an ℓ-bit label for each integer is packed in a parallel set of words. We can then sort the integers according to their labels in $O(\ell/\log n + 1/k)$ time per integer.*

The lemma is essentially just a reformulation of Han's Lemma 5 in [21], and is proved using the same proof. The $O(\ell/\log n)$ term is the inherent cost of packed bucketing using that the labels are small. The $O(1/k)$ term is cost of a matrix transposition of Thorup [36, Lemma 9] with $k \log k$ integers in each word. Also, Han has replaced $\log k$ by $\log \log n$ using $k = O(\log n)$. However, this change is not necessary for his proof. Thus Lemma 19 follows.