Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT
Journal of Algorithms 50 (2004) 96–105
Journal of Algorithms
www.elsevier.com/locate/jalgor

Deterministic sorting in $O(n \log\log n)$ 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, USA

Received 29 May 2002

Abstract

We present a fast deterministic algorithm for integer sorting in linear space. Our algorithm sorts n integers in the range $\{0, 1, 2, \ldots, m-1\}$ in linear space in $O(n \log\log n)$ time. This improves our previous result [Y. Han, Inform. and Comput. 170 (1) (2001) 81–94] which sorts in $O(n \log\log n \log\log\log n)$ time and linear space. This also improves previous best deterministic sorting algorithm [A. Andersson, T. Hagerup, S. Nilsson, R. Raman, in: Proc. 1995 Symposium on Theory of Computing (1995) 427–436; Y. Han, X. Shen, in: Proc. 1995 International Computing and Combinatorics Conference, in: Lecture Notes in Comput. Sci. 959 (1995) 324–333] which sorts in $O(n \log\log n)$ time but uses $O(m^{\varepsilon})$ space. Our results also improves the result of Andersson et al. [A. Andersson, T. Hagerup, S. Nilsson, R. Raman, in: Proc. 1995 Symposium on Theory of Computing (1995) 427–436] which sorts in $O(n \log\log n)$ time and linear space but uses randomization.
© 2003 Elsevier Inc. All rights reserved.

Keywords: Algorithms; Sorting; Integer sorting; Time complexity; Linear space

1. Introduction

Sorting is a classical problem which has been studied by many researchers. Although the complexity for comparison sorting is now well understood, the picture for integer sorting is still not clear. The only known lower bound for integer sorting is the trivial $\Omega(n)$

☆ Preliminary version of this paper has been presented at 2002 ACM Symposium on Theory of Computing (STOC'02). Research supported in part by NSF Grant CCR-03/0245.
E-mail address: hanyij@umkc.edu.
URL: http://welcome.to/yijiehan.

bound. Continuous research efforts have been made by many researchers on integer sorting [1–3,6–8,10,11,13–15]. Recent advances in the design of algorithms for integers sorting have resulted in fast algorithms [3,10,15]. However, these algorithms use randomization or superlinear space. For sorting integers in $\{0, 1, \dots, m-1\}$ $O(m^{\varepsilon})$ space is used in the algorithms reported in [3,10]. When m is large (say $m = \Omega(2^n)$) the space used is excessive. Integer sorting using linear space is therefore extensively studied by researchers. An earlier work by Fredman and Willard [6] shows that n integers can be sorted in $O(n \log n / \log\log n)$ time in linear space. Raman [13] showed that sorting can be done in $O(n\sqrt{\log n \log\log n})$ time in linear space. Later Andersson [2] improved the time bound to $O(n\sqrt{\log n})$. Then Thorup [14] improved the time bound to $O(n(\log\log n)^2)$. Our previous results showed time $O(n \log\log n \log\log\log n)$ [8] for deterministic linear space integer sorting. In this paper we further improve upon previous results. We show that n integers in $\{0, 1, 2, \dots, m-1\}$ can be sorted in $O(n \log\log n)$ time in linear space.

Our result improves on time on the previous best linear space sorting algorithm [8] which uses $O(n \log\log n \log\log\log n)$ time. Our result also improves on space on the previous fastest deterministic sorting algorithm [3,10] which sorts in $O(n \log\log n)$ time and $O(m^{\varepsilon})$ space, where $\{0, 1, \dots, m-1\}$ is the range of the integers. This previous result was obtained independently by Andersson et al. [3] and by Han and Shen [10]. The space used in these previous algorithms is actually $O(m)$. But we may assume that space is reduced to $O(m^{\varepsilon})$ by using radix sorting. Our result also improves the result of Andersson et al. [3] which sorts in $O(n \log\log n)$ time and linear space using randomization.

The techniques used in our algorithm include coordinated pass down of integers on the Andersson's exponential search tree [2] and the linear time multi-dividing of the bits of integers. Although we used multi-dividing technique in our previous design [8], there multi-dividing takes nonlinear time and therefore is too slow. Our new multi-dividing can only be accomplished with coordinated pass down of integers. Instead of inserting integers one at a time into the exponential search tree we pass down all integers one level of the exponential search tree at a time. Such coordinated passing down provides us the chance of performing multi-dividing in linear time and therefore speeding up our algorithm.

We would like to comment on the complexity of $O(n \log\log n)$. This bound was manifested as the best bound even for non-linear space deterministic sorting. Andersson [2] showed several algorithms for sorting, none of them could break the $O(n \log\log n)$ bound. Even for very large integers Andersson showed time $O(n(\log n / \log b + \log\log n))$ where b is the word length (the number of bits in a word). Thus no matter how large the integer is $O(n \log\log n)$ time is needed in Andersson's algorithm. In contrast, for very large integers its large word length can be exploited in a randomized algorithm [3]. Since Andersson's exponential search tree requires $O(n \log\log n)$ time to balance, it would be unlikely that any deterministic algorithm uses exponential search tree approach could undercut the $O(n \log\log n)$ time complexity. As the time of $O(n \log\log n)$ is the converge point for currently the best bound for linear space sorting as demonstrated in this paper, for non-linear space sorting as shown in [3,10], and for a randomized linear space sorting [3,15], it can be viewed as we have reached a milestone.

Although $O(n \log\log n)$ is a natural deterministic bound, recently Han and Thorup find that this complexity can be improved in a randomized setting. In [9] Han and Thorup obtained a randomized integer sorting algorithm which sorts in $O(n\sqrt{\log\log n})$ time and linear space.

2. Preliminary

Our algorithm is built upon the concept of Andersson's exponential search tree [2]. An exponential search tree of n leaves consists of a root r and n^{ε} exponential search subtrees, $0 < \varepsilon < 1$, each having $n^{1-\varepsilon}$ leaves and rooted at a child of r. Thus an exponential search tree has $O(\log\log n)$ levels. Sorting is done by inserting integers into the exponential search tree. When imbalance happens in the tree rebalance needs to be done. In [2] Andersson has shown that rebalance takes $O(n\log\log n)$ time when n integers are inserted into the tree. The dominating time is taken by the insertion. Andersson has shown that insertion can be done in $O(\sqrt{\log n})$ time. He inserts one integer into the exponential tree at a time. Thorup [14] finds that by inserting integers in batches the amortized time for insertion can be reduced to $O(\log\log n)$ for each level of the tree. The size of one batch b at a node is defined by Thorup to be equal to the number of children d of the node. In our previous design [8] we pass down d^2 integers in a batch. We showed [8] that we can speed up computation by such a scheme.

An integer sorting algorithm sorts n integers in $\{0, 1, \ldots, m-1\}$ is called a conservative algorithm [11] if the word length (the number of bits in a word) used in the algorithm is $O(\log(m+n))$. It is called a nonconservative algorithm if the word length used is larger than $O(\log(m+n))$.

One way to speed up sorting is to reduce the number of bits in integers. After the number of bits is reduced we can apply nonconservative sorting. If we are sorting integers in $\{0, 1, \ldots, m-1\}$ with word length $k\log(m+n)$ with $k \geqslant 1$ then we say that we are sorting with nonconservative advantage k.

We use the following notation. For a set S we let $\min(S) = \min\{a \mid a \in S\}$ and $\max(S) = \max\{a \mid a \in S\}$. For two sets S_1, S_2 we denote $S_1 < S_2$ if $\max(S_1) \leqslant \min(S_2)$.

One way to reduce the number of bits in an integer is to use bisection (binary dividing) on the bits of the integer (it is sometimes called exponential range reduction). This idea was first invented by van Emde Boas et al. [4]. In each step, the number of remaining bits is reduced to half. Thus in $\log\log m$ steps $\log m$ bits of the integers are reduced to constant number of bits. This scheme, although very fast, requires a very large amount of memory. It requires $O(m)$ memory cells and therefore cannot be directly executed in linear space ($O(n)$ space). Andersson [2] invented the exponential search tree and he used perfect hashing to reduce the space to linear. He can store only one integer into a word and then applies the hash function. To speed up the algorithm for sorting, we need to pack several integers into one word and then to use constant number of steps to accomplish the hashing for all integers stored in the word. In order to achieve this we relax the demand of perfect hashing. We do not demand n integers to be hashed into a table of size $O(n)$ without any collision. A hash function hashes n integers into a table of size $O(n^2)$ in constant time and without collision suffice for us. Therefore we use the improved version of the hashing function given by Dietzfelbinger et al. [5] and Raman [13] as shown in the following lemma.

Let $b \geqslant 0$ be an integer and let $U = \{0, \ldots, 2^b - 1\}$. The class $\mathcal{H}_{b,s}$ of hash functions from U to $\{0, \ldots, 2^s - 1\}$ is defined as $\mathcal{H}_{b,s} = \{h_a \mid 0 < a < 2^b$, and a is odd$\}$ and for all $x \in U$:

$$h_a(x) = \left(ax \bmod 2^b\right) \operatorname{div} 2^{b-s}.$$

Lemma 1 (Lemma 9 in [13]). *Given integer $b \geqslant s \geqslant 0$ and $T \subseteq \{0, \ldots, 2^b - 1\}$ with $|T| = n$, and $t \geqslant 2^{-s+1}\binom{n}{2}$, a function $h_a \in \mathcal{H}_{b,s}$ can be chosen in $O(n^2 b)$ time such that the number of collisions* $\operatorname{coll}(h_a, T) \leqslant t$.

Take $s = 2\log n$ we obtain a hash function h_a which hashes n integers in U into a table of size $O(n^2)$ without any collision. Obviously $h_a(x)$ can be computed for any given x in constant time. If we pack several integers into one word and have these integers properly separated with several bits of 0's we can safely apply h_a to the whole word and the result is that hashing values for all integers in the word have been computed. Note that this is possible because only the computation of a multiplication, mod 2^b and div 2^{b-s} is involved in computing a hash value.

Andersson et al. [3] used a randomized version of a hash function in $\mathcal{H}$ because they could not afford to construct the function deterministically.

A problem with Raman's hash function is that it takes $O(n^2 b)$ time to find the right hash function. Here b is the number of the bits in an integer. What we needed is a hash function which can be found in $O(n^c)$ time for a constant c because this is needed in the exponential search tree [2,13]. Obviously Raman's hash function does not satisfy this criterion when b is large. However, Andersson's result [2] says that n integers can be sorted in linear space in $O(n(\log n/\log b + \log\log n))$ time. Thus if $b > n$ we simply use Andersson's sorting algorithm to sort in $O(n \log\log n)$ time. Thus the only situation we have to consider is $b \leqslant n$. Fortunately, for this range of b, $O(n^2 b) = O(n^3)$. Therefore we can assume the right hash function can be found in $O(n^3)$ time.

Note that although the hash table has size $O(n^2)$ it does not affect our linear space claim because we do not use hash value to index into a table. Hashing is only used to serve the purpose of reducing the number of bits in an integer.

We will use signature sorting [3] in our algorithm. Signature sorting works as follows. Suppose that n integers have to be sorted and each integer has $\log m$ bits. We view that each integer has h segments with each segment containing $\log m/h$ bits. Now we apply hashing to each and every segment in each integer and we get $2h \log n$ bits of hashed values for each integer. After sorting on hashed values for all integers the original sorting problem (of sorting n integers of $\log m$ bits each) can be transformed to the sorting problem of sorting n integers of $\log m/h$ bits each.

The point in the above hashing is that there are h "distinguishing" segments that should hash uniquely into $2\log n$ bits. A random hash function will do, but deterministically, we need $O(n^3)$ time to construct the hash function. However, what we can do deterministically in $O(p^3)$ time is to find a good hash function for a small set of integers $a_1, \ldots, a_p$. With this fixed hash function and minor modifications, we can apply it to a set S of n integers to gain nonconservative advantage. It requires $O(p^3)$ time to identify an appropriate hash function over the a_i's. A subsequent sorting over n packed hash values of $2h \log p$ bits

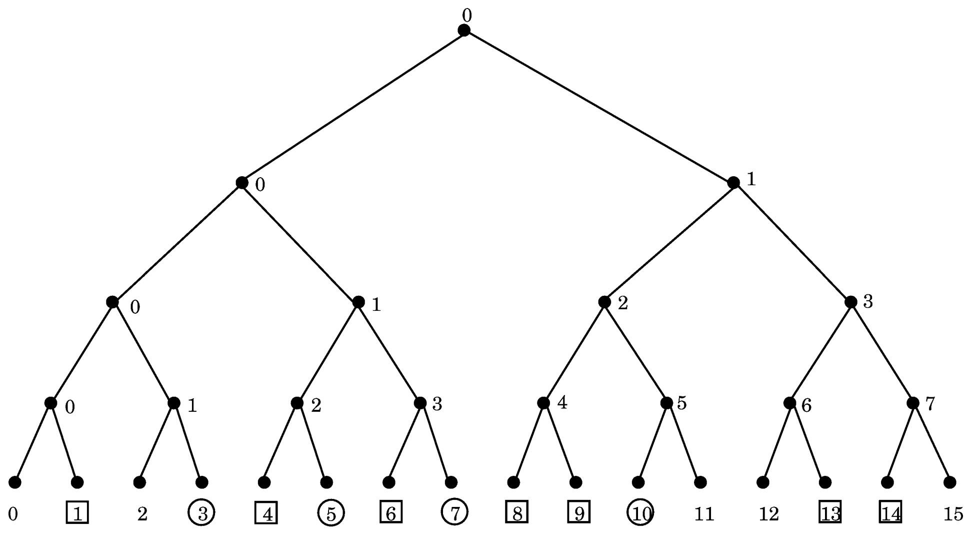

Fig. 1. Set partitioning. The numbers in circles are partitioning integers. The numbers in squares are integers in set S.

will enable us to gain additional h nonconservative advantage because the multi-cutting will cut all h segments except one. We partition the bits in a_i into h segments and take some of these h segments. We will also partition bits for each integer in S into h segment and take one segment and discard other segments. For each a_i we essentially take all the h segments. However, the corresponding segments of a_i and a_j may be identical. In this case we just need one of them. The segment we take for an integer in S is the segment which "branches out" of a_i's. We therefore cut integers to $\log m/h$ bits. In Fig. 1 we show that

$$a_1 = 3, \quad a_2 = 5, \quad a_3 = 7, \quad a_4 = 10, \quad S = \{1, 4, 6, 8, 9, 13, 14\}.$$

We partition each integer into 2 segments. From $a_1 = 3$ we obtain upper segment 0, lower segment 3. From $a_2 = 5$ we obtain upper segment 1 and lower segment 1. From $a_3 = 7$ we obtain upper segment 1 and lower segment 3. From $a_4 = 10$ we obtain upper segment 2 and lower segment 2. For $1 \in S$ we obtain lower segment 1 because it branches out from $a_1 = 3$ in the lower segment. For $4 \in S$ we obtain lower segment 0. For $8 \in S$ we obtain lower segment of 0. For $9 \in S$ we obtain lower segment of 1. For $13 \in S$ we obtain upper segment of 3. For $14 \in S$ we obtain upper segment of 3. Therefore the number of bits in each integer in S is cut to half.

In [8] the following Lemma 2 was given. The details of the proof for the conservative version of the algorithm for Lemma 2 was provided in [8]. The nonconservative version of the algorithm for Lemma 2 can also be readily derived and it was sketched in [8].

Lemma 2. *n integers can be sorted into $\sqrt{n}$ sets $S_1, S_2, \ldots, S_{\sqrt{n}}$ such that each set has $\sqrt{n}$ integers and $S_i < S_j$ if $i < j$, in time $O(n \log\log n/\log k)$ and linear space with nonconservative advantage $k \log\log n$.*

3. Sorting in $O(n \log\log n)$ time and linear space

For sorting n integers in the range $\{0, 1, 2, \ldots, m-1\}$ we assume that the word length used in our conservative algorithm is $O(\log(m+n))$. The same assumption is made in previous designs [2,6,8,13,14]. In integer sorting we often pack several small integers into one word. We always assume that all the integers packed in a word use the same number of bits.

We are going to use the basic ideas of an Andersson's exponential search tree [2]: we construct a balanced search tree where the degree of a node with m descendents is $\Theta(m^{1/5})$, hence where each child has $\Theta(m^{4/5})$ descendents.

In Andersson's exponential search tree [2], integers are inserted (passed down) into the tree one at a time. Thorup [14] suggested to pass down d integers at a time, where d is the number of children of the node in the tree where integers are to be passed down. In our previous design [8] we passed down d^2 integers at a time. Here we will stick with this scheme, namely passing down d^2 integers at a time. What is different from our previous design is that we will not pass down the d^2 integers all the way down the tree. Instead we will pass down one level of the tree d^2 integer at a time until all integers are passed down one level. Thus at the root we pass down $n^{2/5}$ integers at a time to the next level. After we have passed down all integers to the next level we essentially partitioned integers into $t_1 = n^{1/5}$ sets $S_1, S_2, \ldots, S_{t_1}$ with each S_i containing $n^{4/5}$ integers and $S_i < S_j$ if $i < j$. We then take $n^{(4/5)\cdot(2/5)}$ integers from each S_i at a time and coordinate them to be passed down to the next level of the exponential search tree. We repeat this until all integers are passed down to the next level. At this time we have partitioned integers into $t_2 = n^{1/5} \cdot n^{4/25} = n^{9/25}$ sets $T_1, T_2, \ldots, T_{t_2}$ with each set containing $n^{16/25}$ integers and $T_i < T_j$ if $i < j$. Now we are ready to pass integers down to the next level in the exponential search tree.

It should not be difficult to see that the tree balance operation takes $O(n \log\log n)$ time with $O(n)$ time for each level. This is the same as in the original exponential search tree proposed by Andersson [2]. For example, at the root we first take $n^{1/5}$ integers and sort them by comparison sorting. This builds one level of the exponential search tree. We then start to pass integers down the level. If the number of integers at a child exceeds $2n^{4/5}$ we split the node into two nodes. Thus at the end of this passing down we end up with at most $2n^{1/5}$ children for the root. We then regroup them to form exactly $n^{1/5}$ sets $S_1, S_2, \ldots, S_{t_1}$ as mentioned above.

We shall number the levels of the exponential search tree top down so that root is at level 0. Now consider the passing down at level s. Here we have $t = n^{1-(4/5)^s}$ sets $U_1, U_2, \ldots, U_t$ with each set containing $n^{(4/5)^s}$ integers and $U_i < U_j$ if $i < j$. Because each node at this level has $p = n^{(1/5)(4/5)^s}$ children at level $s+1$ we will pass down $q = n^{(2/5)(4/5)^s}$ integers for each set, or a total of $qt \geqslant n^{2/5}$ integers for all sets, at a time.

The pass down can be viewed as sorting q integers in each set together with the p integers $a_1, a_2, \ldots, a_p$ in the exponential search tree so that these q integers are partitioned into $p+1$ sets $S_0, S_1, \ldots, S_p$ such that $S_0 < \{a_1\} < S_1 < \{a_2\} < \cdots < \{a_p\} < S_p$.

Since we do not have to totally sort the q integers and $q = p^2$, a temptation is to use Lemma 2 to sort. For that we need nonconservative advantage which we will derive below. We will use linear timed multi-dividing technique given below to accomplish this.

In [8, Section 7] it is shown that sorting the integers down the exponential search tree takes no more than $O(n\sqrt{\log\log n})$ time per level. Therefore we assume we have already sorted to level $l = 2\log\log\log n$ and we are considering the sorting down the levels greater than $2\log\log\log n$.

We use signature sorting [3] to accomplish multi-dividing. We adapt signature sorting to work for us as follows. Suppose we have a set T of p integers already sorted as $a_1, a_2, \ldots, a_p$ and we wish to use the integers in T to partition a set S of q integers $b_1, b_2, \ldots, b_q$ to $p+1$ sets $S_0, S_1, \ldots, S_p$ such that $S_0 < \{a_1\} < S_1 < \cdots < \{a_p\} < S_p$. We will call this as partitioning q integers by p integers. Let $h = \log n/(c\log p)$ for a constant $c > 1$. $h/\log\log n \log p$-bit integers can be stored in one word such that each word contains only $(\log n)/(c\log\log n)$ bits. We first view the bits in each a_i and each b_i as of $h/\log\log n$ segments of equal length. We view each segment as an integer. To gain nonconservative advantage for sorting we hash the integers in these words (a_i's and b_i's) to get $h/\log\log n$ hashed values in one word. In order to have intermediate values in the computing of hash values do not interfere between adjacent segments we can separate even and odd segments into two words by applying a suitable mask. We then compute hash values for the two words and then combine the hashed values of these two words into one. Let a_i' be the hashed word corresponding to a_i and b_i' be the hashed word corresponding to b_i. Note that the hashed values have a total of $(2\log n)/(c\log\log n)$ bits. However, these hashed values are separated into $h/\log\log n$ segments in each words. There are "null spaces" between two adjacent segments. We can set these "null spaces" to 0's by applying a mask. We first pack all segments into $(2\log n)/(c\log\log n)$ bits (details below in Section 3.2, the $\log\log n$ in the denominator is needed for this purpose). Now we view each hashed word as an integer and sort all these hashed words (this sorting which takes linear time will be described in detail below in Section 3.1). After this sorting the bits in a_i and b_i are cut to $(\log\log n/h)$th. Thus we have additional multiplicative advantage of $h/\log\log n$.

After repeating the above process g times we gain nonconservative advantage of $(h/\log\log n)^g$ while we spend $O(g)$ time per integer (or $O(gqt)$ time for qt integers) because each multi-dividing is done in linear time (constant time per integer or $O(qt)$ time for qt integers).

The hashing function we used for hashing is obtained as follows. Because we will hash segments which are $\log\log n/h$th, $(\log\log n/h)^2$th, $\ldots$ of the whole integer, we will use hash functions for segments which are $\log\log n/h$th, $(\log\log n/h)^2$th, $\ldots$. of the whole integer. The hash function for segments which are $(\log\log n/h)^t$th of the whole integer is obtained by cutting each of the p integers into $(h/\log\log n)^t$ segments. Viewing each segment as an integer we obtain $p(h/\log\log n)^t$ integers. We then obtain one hash function for these $p(h/\log\log n)^t$ integers. Because $t < \log n$ we obtain no more than $\log n$ hash functions.

3.1. Linear timed sorting and time analysis

Now let us take a look at the linear time sorting we mentioned earlier. Assume that we have packed the hashed values for each word into $(2\log n)/(c\log\log n)$ bits. We have t sets with each set containing $q+p$ hashed words of $(2\log n)/(c\log\log n)$ bits each. These integers are to be sorted within each set. If we sort each set individually we cannot achieve linear time. What we do is to combine all hashed words into one pool and sort them as follows.

```
Procedure Linear-Time-Sort.
Input: there are $r \geqslant n^{2/5}$ integer $d_i$'s; $d_i.value$ is the integer value of $d_i$ which has
$(2\log n)/(c\log\log n)$ bits; $d_i.set$ is the set $d_i$ is in. Note that there are only $t$ sets.
begin
(1) Sort all $d_i$'s by $d_i.value$ using bucket sort. Assume that the sorted integers are in $A[1..r]$.
    This step takes linear time because there are at least $n^{2/5}$ integers to be sorted.
(2) for $j = 1$ to $r$ do
        Put $A[j]$ into set $A[j].set$.
end
```

Thus we have all sets sorted in linear time.

As we have said that after g times of reduction of bits we have nonconservative advantage $(h/\log\log n)^g$. We do not carry this bits reduction to the end because after we gained sufficient nonconservative advantage we can switch to Lemma 2 for completion of partitioning q integers by the p integers for each set.

Now we invoke Lemma 2 to do the partition. Because we have $(h/\log\log n)^g$ nonconservative advantage the algorithm in Lemma 2 takes $O(\log\log n/(g(\log h - \log\log\log n) - \log\log\log n))$ time per integer. We know that we spend $O(g)$ time per integer for gaining nonconservative advantage and $O(\log\log n/(g(\log h - \log\log\log n) - \log\log\log n))$ time for applying Lemma 2 to partition. To optimize we let these two quantities equal. That is we let $g = \log\log n/(g(\log h - \log\log\log n) - \log\log\log n)$. We arrive at

$$\begin{aligned} g &= \bigl(\log\log n/(\log h - \log\log\log n - (\log\log\log n)/g)\bigr)^{1/2} \\ &< \bigl(\log\log n/(\log h - 2\log\log\log n)\bigr)^{1/2}. \end{aligned}$$

Since we have assumed that we are working on levels greater than $2\log\log\log n$ we need to sum g for levels from $2\log\log\log n$ to $\log\log n$. Since $\log h$ indicates the level we need to let $\log h$ assume values $2\log\log\log n, 2\log\log\log n + 1, \ldots, \log\log n$ for summing g which is the time spent in one level. Thus we have

$$\begin{aligned} \sum_{\text{level}=2\log\log\log n}^{\log\log n} g &\leqslant \sum_{\log h=2\log\log\log n}^{\log\log n} \bigl(\log\log n/(\log h - 2\log\log\log n)\bigr)^{1/2} \\ &= \sum_{a=2\log\log\log n}^{\log\log n} \bigl(\log\log n/(a - 2\log\log\log n)\bigr)^{1/2} \\ &= O(\log\log n). \end{aligned}$$

That is: each integer incurs $O(\log\log n)$ time.

3.2. *Packing*

Now we are back to the packing problem which we solve as follows. We can assume that the number of bits $\log m$ in a word satisfying $\log m \geqslant \log n \log\log n$, for otherwise we can use radix sort to sort the integers. A word has $h/\log\log n$ hashed values (segments) in it at level $\log h$ of the exponential search tree. The total number of hashed bits in a word is $(2\log n)/(c\log\log n)$ bits. Therefore the hashed bits in a word looks like $0^i t_1 0^i t_2 0^i \ldots t_{h/\log\log n}$, where t_k's are hashed bits and 0^i are the null spaces between hashed bits. We first pack $\log\log n$ words into one word to get

$$w_1 = 0^j t_{11} t_{21} \ldots t_{\log\log n,1} 0^j t_{12} t_{22} \ldots t_{\log\log n,2} 0^j \ldots t_{1,h/\log\log n} t_{2,h/\log\log n} \\ \ldots t_{\log\log n,h/\log\log n},$$

where $t_{i,k}$'s, $k = 1, 2, \ldots, h/\log\log n$, are from the ith word. We then use the packing algorithm in Leighton [12, Section 3.4.3] in $O(n \log\log n)$ steps to pack w_1 to

$$w_2 = 0^{jh/\log\log n} t_{11} t_{21} \ldots t_{\log\log n,1} t_{12} t_{22} \ldots t_{\log\log n,2} \cdots t_{1,h/\log\log n} t_{2,h/\log\log n} \\ \ldots t_{\log\log n,h/\log\log n}.$$

Now the packed hash bits in w_2 has only $2\log n/c$ bits. We use $O(\log\log n)$ time to unpack w_2 to $\log\log n$ words

$$w_{3,k} = 0^{jh/\log\log n} 0^r t_{k1} 0^r t_{k2} 0^r \ldots t_{k,h/\log\log n}, \quad k = 1, 2, \ldots, \log\log n.$$

We then use $O(\log\log n)$ time to pack these $\log\log n$ words into one word

$$w_4 = 0^r t_{11} 0^r t_{12} 0^r t_{13} 0^r \ldots t_{1,h/\log\log n} 0^r t_{21} 0^r t_{22} 0^r \ldots t_{2,h/\log\log n} 0^r \\ \ldots t_{\log\log n,1} 0^r t_{\log\log n,2} 0^r \ldots t_{\log\log n,h/\log\log n}.$$

We then use $O(\log\log n)$ steps to pack w_4 to

$$w_5 = 0^s t_{11} t_{12} t_{13} \ldots t_{1,h/\log\log n} t_{21} t_{22} \ldots t_{2,h/\log\log n} \cdots t_{\log\log n,1} t_{\log\log n,2} \\ \ldots t_{\log\log n,h/\log\log n}.$$

We finally use $O(\log\log n)$ steps to unpack w_5 to $\log\log n$ packed words. Overall we spent $O(\log\log n)$ time for packing $\log\log n$ words. Thus for each word the time spent is constant.

Thus we have:

Theorem 1. *n integers can be sorted in $O(n\log\log n)$ time and linear space.*

4. Conclusions

We have finally achieved $O(n\log\log n)$ time and linear space for integer sorting. This can be viewed as a milestone reached by us. The true bound for integer sorting is still unknown at this moment, but we are approaching to it gradually.

Acknowledgment

The author is grateful to the referee who suggested the simplification of the paper and who's comments have significantly improved the presentation of the paper.

References

[1] S. Albers, T. Hagerup, Improved parallel integer sorting without concurrent writing, Inform. and Comput. 136 (1997) 25–51.

[2] A. Andersson, Fast deterministic sorting and searching in linear space, in: Proc. 1996 IEEE Symp. on Foundations of Computer Science, 1996, pp. 135–141.

[3] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, Sorting in linear time?, in: Proc. 1995 Symposium on Theory of Computing, 1995, pp. 427–436.

[4] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10 (1977) 99–127.

[5] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen, A reliable randomized algorithm for the closest-pair problem, J. Algorithms 25 (1997) 19–51.

[6] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. System Sci. 47 (1994) 424–436.

[7] T. Hagerup, H. Shen, Improved nonconservative sequential and parallel integer sorting, Inform. Process. Lett. 36 (1990) 57–63.

[8] Y. Han, Improved fast integer sorting in linear space, Inform. and Comput. 170 (1) (2001) 81–94.

[9] Y. Han, M. Thorup, Sorting integers in $O(n\sqrt{\log\log n})$ expected time and linear space, in: Proc. 2002 IEEE Symposium on Foundations of Computer Science (FOCS'02), 2002, pp. 135–144.

[10] Y. Han, X. Shen, Conservative algorithms for parallel and sequential integer sorting, in: Proc. 1995 International Computing and Combinatorics Conference, in: Lecture Notes in Comput. Sci., Vol. 959, 1995, pp. 324–333.

[11] D. Kirkpatrick, S. Reisch, Upper bounds for sorting integers on random access machines, Theoret. Comput. Sci. 28 (1984) 263–276.

[12] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Kaufmann, San Mateo, CA, 1992.

[13] R. Raman, Priority queues: small, monotone and trans-dichotomous, in: Proc. 1996 European Symp. on Algorithms, in: Lecture Notes in Comput. Sci., Vol. 1136, 1996, pp. 121–137.

[14] M. Thorup, Fast deterministic sorting and priority queues in linear space, in: Proc. 1998 ACM–SIAM Symp. on Discrete Algorithms (SODA'98), 1998, pp. 550–555.

[15] M. Thorup, Randomized sorting in $O(n\log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations, in: Proc. 8th ACM–SIAM Symp. on Discrete Algorithms (SODA'97), 1997, pp. 352–359.