Optimizing Integer Sorting in $O(n \log \log n)$ Expected Time in Linear Space

Ajit Singh
M. E. (Computer Science and Engineering)
Department of computer Science and Engineering,
Thapar University, Patiala
thaparian.ajit@gmail.com

Dr. Deepak Garg
Asst. Prof.
Department of Computer Science and Engineering,
Thapar University, Patiala
dgarg@thapar.edu

Abstract-**The traditional algorithms for integer sorting give a bound of $O(n \log n)$ expected time without randomization and $O(n)$ with randomization. Recent researches have optimized lower bound for deterministic algorithms for integer sorting [4, 5, 7]. We present a fast deterministic algorithm for integer sorting in linear space. The algorithm discussed in this paper sorts n integers in the range $\{0, 1, 2 \ldots m-1\}$ in linear space in $O(n \log \log n)$ expected time. This improves the traditional deterministic algorithms. The algorithm can be compared with the result of Andersson, Hagerup, Nilson and Raman which sorts n integers in $O(n \log \log n)$ expected time but uses $O(m^{\varepsilon})$ space [2, 14]. The algorithm can also be compared with the result of Andersson which sort n integers in $O(n \log \log n)$ expected time and linear space but uses randomization [2, 5]. The result of this paper can also be compared with the result of Yijie Han which sort n integers in $O(n \log \log n)$ expected time and linear space but passes integers in a batch i.e. all integers at a time [8]. This paper also gives an implementation view of integer sorting in $O(n \log \log n)$ in linear space.**

Keywords-Deterministic Algorithms; Sorting; Integer Sorting; Complexity; Space Requirement.

1. INTRODUCTION

In modern computer world most of the problem is being solved by sorting. It is a classical problem which has been studied by many researchers. The traditional algorithms for sorting give a clear picture for complexity. 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 $\Omega(n)$ bound. Many continuous research efforts have been made by many researchers on integer sorting [2, 3, 5-16]. Recent advances in the design of algorithms for integers sorting have resulted in fast algorithms [5-12, 15, 16]. However, many of these algorithms use randomization or super-linear space. The table-1 shows the comparative analysis of various traditional algorithms:

Table-1: Complexity Comparison

Name	Best	Average	Worst	Memory
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell Sort	$O(n)$	$Depends$	$O(n(logn)^2)$	$O(n)$
Binary tree sort	$O(n)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Heap sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	Depends
Quick sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(\log n)$
Randomized Quick sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(\log n)$
Signature Sort	$O(n)$	$O(n)$	$O(n)$	Linear

For sorting integers in [0, m – 1] range $O(m^{\varepsilon})$ space is used in many algorithms. When m is large, the space used is excessive. Thus integer sorting using linear space is more important and therefore extensively studied by researchers.

Fredman and Willard showed that n integers can be sorted in $O(n \log n / \log \log n)$ time in linear space [11]. Raman showed that sorting can be done in $O\left(n\sqrt{\log n \log \log n}\right)$ time in linear space [15]. Later Andersson improved the time bound to $O\left(n\sqrt{\log n}\right)$ [3]. Then Thorup improved the time bound to $O(n(\log \log n)^2)$ [16]. Later Yijei Han showed $O(n \log \log n \log \log \log n)$ time for deterministic linear space integer sorting [6]. Yijei Han again showed improved result with $O(n \log \log n)$ time and linear space [8].

In these algorithms expected time is achieved by using Andersson's exponential tree [3]. The height of such a tree is $O(\log \log n)$. Also the balancing of the tree will take only $O(\log \log n)$ time. Balancing does not take much time. The major time is taken by the insertion of integers as proved by the Andresson [3]. Thus we only need to reduce the insertion time of integers.

Andersson has shown that if we pass down integers in exponential tree one by one than the insertion takes $O(\sqrt{\log n})$ for each integer i.e. total complexity will be $O(n\sqrt{\log n})$ [3]. This is improvement over the result of Raman which takes $O(n\sqrt{\log n \log\log n})$ expected time [15].

Yijie Han has given an idea which reduces the complexity to $O(n\, log\, log\, n)$ expected time in linear space [8]. The technique used by him is coordinated pass down of integers on the Andersson's exponential search tree [3] and the linear time multi-dividing of the bits of integers. Instead of inserting integer one at a time into the exponential search tree he passed down all integers one level of the exponential search tree at a time. Such coordinated passing down provides the chance of performing multi-dividing in linear time and therefore speeding up the algorithm.

This paper will present a different idea to achieve the complexity of deterministic algorithm for integer sorting in $O(n\, log\, log\, n)$ expected time and linear time which is easy to implement and very simple enough. We are also using Andersson exponential tree [3] to perform the sorting. The exponential tree can't be used as it is for our idea, so we need to modify the exponential tree. We will pass down integers one at a time but limit the comparison one per level. Thus total number of comparison for any integer will be $O(log\, log\, n)$ i.e. total time taken for all integers insertion will be $O(\,n\, log\, log\, n)$ as the height of the tree will be $O(log\ \ log\ \ n)$. In order to perform this we introduce a new concept of Node word. Each node will be containing one extra word which will represent all the keys at that node. We will perform comparison of integer with this word.

2. EXPONENTIAL TREE

The exponential tree was first introduced by Andersson in his research for fast deterministic algorithms for integer sorting [3]. In such a tree the number of children increases exponentially.

An exponential tree is almost identical to a binary search tree, with the exception that the dimension of the tree is not the same at all levels. In a normal binary search tree, each node has a dimension (d) of 1, and has 2^d children. In an exponential tree, the dimension equals the depth of the node, with the root node having a $d = 1$. So the second level can hold two nodes, the third can hold eight nodes, the fourth 64 nodes, and so on. This shows that number of children at each level increased by a multiplicative factor of 2 i.e. exponential increases in number of children at each level.

The tree itself is very complex to handle as number of integer increases. We are using modified concept of exponential tree for our convenience for integer sorting. Here we will say that a tree with properties of binary search tree will be the exponential tree if it has following properties:

1. Each node at level k will hold k number of keys (or integers in our case) i.e. at depth k the number of key in any node will be k keeping root at level 1.
2. Each node at level k will be having $k+1$ children i.e. at depth k the number of children will be $k+1$.
3. All the keys in any node must be sorted.
4. An integer in child i must be greater than key $i-1$ and less than key i.

Thus the node at root will look like as depicted in figure 1.

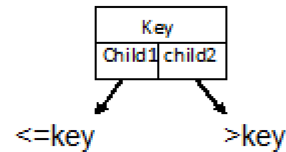

Figure 1: Root Node of Exponential Tree

And the node at ith level or depth will look like as shown in figure 2.

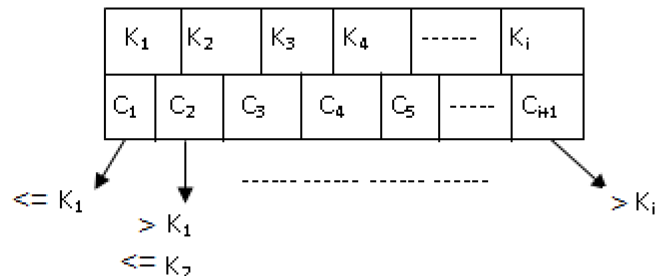

Figure 2: Node at ith level in Exponential Tree

The height of the tree will remain $O(log\, log\, n)$ which can be proved by using induction. The modification will not only reduce the complexity of exponential tree involved in implementing it but also improves the balancing method as well as sorting technique. Integer sorting will be more convenient and fast with this modification.

2.1 Balancing

The balancing of exponential tree is one of the most important and necessary task in order to achieve the $O(n\, log\, log\, n)$ expected time in linear space [3]. If the tree is not balance then the worst case will happen and the expected running time will increase to undesired level. The balancing will

guarantee that the depth of tree will remain $O(log\ log\ n)$ which will lead us to our target. We will say that tree is balanced if the difference between depths (or heights) of both sided children of a key will not exceed by 2. The balancing will happen on all the children of the node. Each key will play a role of a node like in binary search tree in balancing. To balance the tree we will keep track of number of keys passed through a particular node. This track will help us when to call the balancing procedure. We will find the difference between depths of two children by checking how many keys we have passed to those children. If this difference will increase the maximum number of keys the node can hold we will need the balancing.

We are introducing strictly balanced tree concept. The tree will be called strictly balanced if the difference between heights (or depths) of both sided children of a key will not exceed by 1. This balancing will help us to achieve a true bound of $O(n\ log\ log\ n)$ expected time. We can use the self balancing technique which is already in existence in self balancing binary tree.

3. INSERTION IN EXPONENTIAL TREE

The sorting of integers will happens with inserting them into exponential tree. After inserting the integers into tree, we will trace the tree in in-order to get the sorted list as output. As proved by Andersson, insertion into exponential tree takes most of the time [3], thus our focus is to achieve the insertion in lesser expected time.

As discussed above the exponential tree is a binary search tree itself, thus the insertion requires finding the appropriate place of the integer. This will be happening by comparing integer with keys presented in any node. There are at most k number of keys in any node at depth or level k, which means there will be at most k number of comparisons for passing the integer through that level.

Andersson has proved that if we pass down integer to exponential tree one by one then the insertion will take $O\left(n\sqrt{\log n}\right)$ expected time, which does not seem well enough optimized [3]. But in the best case scenario the insertion will take $O(n\ log\ log\ n)$ time as only one comparison will required at each level. The best case happens rarely, thus we can't rely on the best case scenario. Hence we need to further modify the concept to achieve better result.

As discussed passing down integers one at a time does not give well enough expected time. Yijie Han came up with the idea of passing down the integers in group [8]. He used multi-dividing technique to split the integers into smaller lists. He suggested that instead of inserting integer one at a time into the exponential search tree we can pass down all integers to one level of the exponential search tree at a time. Such coordinated passing down provides the chance of performing multi-dividing in linear time and therefore speeding up the algorithm. This technique gives an expected time of $O(\ n\ log\ log\ n)$ in linear space. But it is very difficult to pass all integers at once if the number of integers is very large. Thus the insertion of integers in groups does not seem very good in terms of implementation and complexity associated in passing down all integers at once.

We will pass down integers one by one in our design but will reduce the number of comparison required at each level.

4. SORTING IN $O(\ n\ log\ log\ n\ log\ log\ log\ n)$

While passing down integer one by one in exponential tree, we need to perform comparison at each level in a particular node with all the keys presented in that node as the tree posses the property of binary search tree. But in our modified design of exponential tree, the keys in a particular node are sorted itself. We can use this property to enhance our algorithm. The use of this property can reduce the insertion time of integers. This property allows us to use the binary search method which is already in literature and well used.

As we know that the binary search takes $O(log\ n)$ expected time and uses only $O(1)$ extra memory space. Thus if we find the position of integer to be inserted in the node or pass the integer through a level using binary search it will take $O(log\ k)$ number of comparisons at each level where k is the depth of that node or level. This shows that in average case passing down all integers will take $O(n\ log\ log\ n\ log\ k)$ expected time. In the best case when only one comparison will be required to perform the passing the expected reduces to $O(n\ log\ log\ n)$. The worst case will happen when the number of integers is highest at any node, which is $O(log\ log\ n)$ as there are at most $O(log\ log\ n)$ levels in the tree. Hence in worst case the expected time with binary search will be $O(n\ log\ log\ n\ log\ log\ log\ n)$ which is better than $O\left(\ n\sqrt{\log n}\right)$. It also uses no extra memory. The steps associated in this sorting will be:

$Step1$: $Create\ Exponential\ tree.$
$Step2$: $Repeat\ step\ 3\ to\ 7\ while\ there\ is\ more\ integers.$
// Step 2 Will be repeated n times.
$Step3$: $Set\ l = 1.$
$Step4$: $Repeat\ step\ 5\ and\ 6\ while\ integer\ is\ not\ inserted.$
// Step 4 Repeated $log\ log\ n$ times.
$Step5$: $Search\ the\ position\ where\ integer\ to\ be\ inserted$

$or\ to\ be\ passed\ to\ at\ level\ l\ using\ binary\ search.$
// Step 5 Takes $log\ l$ comparison to find the index.
$Step6: if\ the\ node\ is\ full\ then\ l = l +$
$1\ else\ insert\ the\ integer\ in\ that\ node.$
$Step7: Call\ the\ balancing\ method.$
$Step8: Trace\ the\ tree\ in -$
$order\ to\ get\ the\ sorted\ list.$
$Step9: stop.$

Let us discuss the algorithm given above step by step. $Step1$ just creates the exponential tree in which integers are supposed to be inserted. In $step2$ the insertion begins. $Step3\ to\ 7$ find the position of integer using binary search where it is to be inserted.

Now the $step2$ will be called n times. $Step4$ will be repeated $log\ log\ n$ times and $step5$ will require $log\ l$ number of comparisons where l is the level. Thus total number of comparisons will be $O(n\ log\ log\ n\ log\ l)$. Hence when the level is highest which will be $log\ log\ n$ then the worst case occurs. So the algorithm takes $O(n\ log\ log\ n\ log\ log\ log\ n)$ expected time in linear space as discussed above.

5. NODE WORD

Now we have reached up to $O(\ n \log\log n \log\log\log n)$ expected time in linear space. Here we are introducing the concept of node word. As we know that in modern era of computers, the word length i.e. bits that can be processed per instruction by a computer is very high (64 bits in general PCs). Such a high word length can be used to speed up the processing. Most of the bits of any instruction go useless as these are not used for any data.

The concept is to exploit the inherent parallelism present in uniprocessor system. Instead of comparing on integer at a time we can compare multiple integers at once. This is a simple trick which is already in use in graphics. Here we will use it for integer comparison. We will achieve a time bound of $O(n\ log\ log\ n)$ in linear space using this concept. In order to do so we have to limit the number of comparisons to $O(1)$ at per node or level which means only one comparison must tell us that where to insert the integer or pass it down.

To limit the number of comparisons to $O(1)$ we will merge the all keys presented in any node to one long word. We will call it node word. When the integer needs to be passing down we will create a long word of that integer by duplicating its value. We must duplicate the value of integer by the number of keys presented in that node times. Thus only one comparison will be enough to tell us the accurate position of the integer to be passed down. Hence the node at ith level or depth will look like as shown in figure 3.

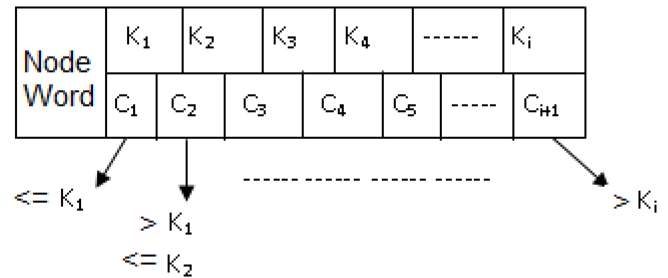

Figure 3: ith level Node in Exponential Tree with Node Word.

The creation of node word can be done by using a hash function that will convert the integers into smaller values which can be represented in lesser bits than actual integer. The lesser the bits to be merge more convenient it will be to make a node word. Now the function will also ensure that as the integer proceeds from one level to another down the tree, the node words are having all keys merged in it. This can be done by choosing the hash function in such a way that it is a function of number of keys in node, the range of key's value and the bit length of the node word.

The process of creating the node word may take well enough processing time. We can reduce it by modifying the node of the tree. Now the tree node will also carry a variable to store node word. As we know that the node word once created will only needs to change when new key is inserted in that node which will happen only with leaf nodes or in balancing the tree. Hence if we store the node word in node itself we only need to create long word for the integer to be inserted. This will save a lot of processing time.

Here we need to consider the time taken by the process of creation of long word for the integer to be inserted. As we know that the height of the tree will be $O(n\ log\ log\ n)$, thus only $O(n\ log\ log\ n)$ words will be created, one at each level. Thus the expected time for this will not exceed $O(n\ log\ log\ n)$ which means we have achieved our targeted complexity.

Hence by using node word we can achieve a true time bound of $O(n\ log\ log\ n)$ in linear space.

6. CONCLUSION

We have achieved $O(n\ log\ log\ n\ log\ log\ log\ n)$ expected time in linear space using binary search in exponential tree. This result is very important in view of implementation. And also, we have achieved $O(n\ log\ log\ n)$ expected time in linear

space for integer sorting using exponential tree. We achieved this by passing one integer at a time. This has reduced the complexity associated with passing all integers at once.

REFERENCES

[1] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10, 1977.

[2] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, Sorting in linear time?, Symposium on Theory of Computing, 1995.

[3] A. Andersson, Fast deterministic sorting and searching in linear space, IEEE Symposium on Foundations of Computer Science, 1996.

[4] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen, A reliable randomized algorithm for the closest-pair problem, J. Algorithms 25, 1997.

[5] 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 Symposium on Discrete Algorithms (SODA'97), 1997.

[6] Y. Han, Improved fast integer sorting in linear space, Inform. and Comput., 2001.

[7] Y. Han, M. Thorup, Sorting integers in $O\left(n\sqrt{\log \log n}\right)$ expected time and linear space, IEEE Symposium on Foundations of Computer Science (FOCS'02), 2002.

[8] Y. Han, Deterministic sorting in $O(n \log \log n)$ time and linear space, 34th STOC, 2002.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms, Second Edition,The MIT Press and McGraw-Hill Book Company, 2001.

[10] S. Albers, T. Hagerup, Improved parallel integer sorting without concurrent writing, Inform. and Comput., 1997.

[11] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. System Sci., 1994.

[12] T. Hagerup, H. Shen, Improved nonconservative sequential and parallel integer sorting, Inform. Process. Lett., 1990.

[13] D. Kirkpatrick, S. Reisch, Upper bounds for sorting integers on random access machines, Theoret. Comput. Sci., 1984.

[14] Y. Han, X. Shen, Conservative algorithms for parallel and sequential integer sorting, International Computing and Combinatorics Conference, 1995.

[15] R. Raman, Priority queues: small, monotone and trans-dichotomous, European Symp. on Algorithms, 1996.

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