The Growing-Tree Sorting Algorithm

SHERENAZ AL-HAJ BADDAR, SAMI SERHAN, HAMED ABDEL-HAQ
Department of Computer Science
University of Jordan
Queen Rania St., Amman
JORDAN
s.baddar@ju.edu.jo

Abstract: - In this paper, a new sorting algorithm called the Growing-Tree Sorting Algorithm (GTSA) is proposed to sort a vector of items in a linear time. It implements a structured tree that stores the digits of each input integer and retrieves the integers in the correct order. If d is the maximum number of digits per input number and n is the input size, then it can be shown that the GTSA algorithm requires $\theta(dn)$ time to sort both in its best and worst-case scenarios. It can be also shown that the best-case memory complexity of GTSA is $\theta(d)$ and that the worst-case memory complexity is $\theta(dn)$. To evaluate the performance of GTSA, it was compared with the radix and counting sort algorithms in terms of the memory they consume and their corresponding execution times. The experiments showed that GTSA was, in general, more memory-conservative than counting sort and radix sorting algorithms. The experiments also showed that the GTSA was faster than the radix and counting sort algorithms when the input size was relatively large and the input range was relatively small.

Key-Words: - Non-comparative sorting, linear-time sorting, tree, post-order traversal.

1 Introduction

A sorting algorithm receives a sequence A of n numbers such that $A = \langle a_1,a_2,\ldots,a_n\rangle$, and outputs a permutation of the input sequence $\langle a'_1,a'_2,\ldots,a'_n\rangle$ such that $a'_1 \le a'_2 \le \ldots \le a'_n$ [1]. To achieve its goal, a sorting algorithm may use the comparison operator to rearrange the input numbers into the desired order. In this case, the sorting algorithm is said to be comparative. The class of comparative sorting algorithm includes, but is not limited to: quick sort [2], heap sort [3], merge-sort [4], and Shell sort[5] algorithms. Another class of sorting algorithms produces the sorted permutation without using comparisons at all. Instead, these algorithms make certain assumptions about the size and the range of the input numbers to help sort them. This class of sorting algorithms is referred to as non-comparative, and mainly includes: counting sort [4], radix sort [4], and bucket sort [1]. The proposed Growing-Tree Sorting Algorithm (GTSA) is a non-comparative sorting algorithm as well.

To evaluate the performance of a given sorting algorithm, two metrics are usually considered: the execution time, and the memory consumption. A decision-tree model can be used to show that any comparative sorting algorithm must make Ω ($n\log n$) comparisons, in the worst case, to sort n numbers [1]. Non-comparative sorters, on the other hand, produce the sorted permutation of n numbers using O(n) operations. This implies that non-comparative sorters are usually faster than their comparative counterparts.

Some sorting algorithms are referred to as in-place sorters, since they require a constant number of inputs to be stored in extra memory locations, and hence have a memory consumption complexity of O(1). Heap sort, insertion sort and quick sort are in-place sorting algorithms, whereas merge-sort requires O(n) locations to sort n numbers[1]. Counting sort is not an in-place sorter either, since it consumes O(k+n) locations to sort the n numbers that fall in the range 0 to k inclusive[1].

In this paper, a new non-comparative, not-in-place, liner-time sorting algorithm is introduced. The development of GTSA was motivated by the limitations of the widely-used liner-time sorters. Consider counting sort for instance, despite its simplicity it consumes a relatively large amount of memory as the range of input numbers increases. This makes it impractical for applications with relatively large input range. Radix sort also suffers from some drawbacks. It requires all input numbers to have the same number of digits, and if that is not the case, then padding with extra zeros must be

done. Besides, radix sort's execution time significantly grows as the number of digits in input numbers grows. An additional drawback is that neither counting sort nor radix sort can naturally handle signed inputs and/or floating point numbers. This work aims at developing a liner-time sorting algorithm that consumes a relatively smaller amount of memory and can handle not only positive integers, but also negative and floating-point numbers as well.

A sorting algorithm is said to be stable if and only if the input numbers with the same value appear in the sorted output array in the same order as they do in the input array. Sorting stability is necessary to achieve whenever satellite data are associated with each input number[1]. Counting, radix, bucket and are stable sorters. The GTSA sorter described here is stable too.

Section 2 of this paper highlights the most widely-used liner-time sorters. Section 3 describes the GTSA, illustrates it via the means of a numerical example, and establishes its time and memory-complexities. Section 4 describes the GTSA implementation together with the experiments carried out to compare the performances of radix sort, counting sort, and GTSA. It also analyzes the results and emphasizes the improvements achieved by GTSA over radix and counting sorts. Finally, the conclusions drawn together with the future work are depicted in section 5.

2 Literature review

Counting sort is a stable sort that was developed by H. H. Seward back in 1954 to sort positive integers [4]. This algorithm restricts the range of the input to some value, say k, which implies that all input numbers will fall in the range$[0,k]$. To sort an input element x, counting sort counts the number of input elements less than x. This helps place x in its correct location in the output array. The process is repeated for all input elements to complete the sorting of the input array. Figure 1 depicts the pseudocode of the counting sort algorithm.

Based on Fig. 1, the counting sort time complexity is $\theta(k + n)$, where n is the input size. If k is $O(n)$, then counting sort will sort in $\theta(n)$ time. The memory complexity of this algorithm is also $\theta(k+n)$, which means that memory consumption of count sort increases significantly as the input range increases.

```
Counting sort(A,B,k)
Input: unsorted integer-list
A: ∀a ∈ A,0 ≤ a ≤ k
Output: sorted integer-array
B
For i = 0 to k do
   C[i] = 0
For j= 1 to length(A) do
   C[A[j]] = C[A[j]] + 1
For  I = 1 to k do
   C[i] = C[i] + C[i-1]
For  j = length(A) downto 1
do
{
   B[C[A[j]]] = A[j]
   C[A[j]] = C[A[j]] - 1
}
```

Fig. 1 Counting Sort pseudocode[1].

Radix sort was published in paper by L. J. Comrie in 1929 [4]. Originally, radix sort was used in card-sorting machines. The radix sorting machines organized the n d-digit numbers in d columns and sorted all numbers based on the ith digit in each iteration, starting from the least significant digit. Radix sort, as described in Fig. 2, uses a stable sort like counting sort in the ith iteration. The time it takes radix sort to accomplish its task is $\theta(d(n+k))$ and its memory complexity is $\theta(k+n)$, where each digit is in the range$[0,k]$.

Bucket sorting is another stable non-comparative sorting algorithm that was developed

```
Radix sort(A,d)
Input: unsorted integer-list A
Output: sorted integer-array A
For i = 1 to d   do
      Use a stable sort to sort
array A on digit i
```

Fig. 2 Radix Sort pseudocode[1].

by E. J. Isaac and R. C. Singletonin 1956[1]. Bucket sort assumes that the input is drawn from a uniform distribution and that it is generated by a random processes that produces numbers in the interval [0, 1) uniformly. The input would be split into n equally-sized buckets, and each of the inputs would be placed in its corresponding bucket. To produce the sorted output, the buckets would be traversed in order after sorting the numbers within each bucket. Bucket sorting runs in linear expected-time and requires $\theta(n)$ memory to sort n real

```
Bucket sort(A)
Input: unsorted integer-array A
Output: sorted integer-array A
n = length(A)
For i = 1 to n do
     insert A[i] into list B[⌊nA[i]⌋]
For i = 0 to n-1 do
     sort list B[i] with insertion sort
concatenate the lists B[0], B[1], …, B[n-1]together in order and copy the answer to array A
```

Fig. 3 Bucket Sort pseudocode[1].

numbers in the range [0, 1). Figure 3 describes the bucket sort algorithm.

Bentley et al describe a sorting algorithm that blends quick sort and radix sort in [6]. Wang proposed a non-comparative sorting algorithm, called Self-Indexed Sorting (SIS) in [7]. The SIS algorithm carries out the sorting task by directly mapping each input element into a relative offset based on its value. Some effort was invested in developing algorithms that sort in linear space. Consider, for example, the work in [8] which describes a linear-space sorting algorithm that takes O(n log log n log log log n) time to accomplish its task. Yan et al improved this algorithm even further and came up with a linear-space integer sorter than runs in O($n\sqrt{\log \log n}$) expected time[9]. A comprehensive survey on sorting algorithms is depicted is [10].

3 The GTSA algorithm

Here we develop an integer variant of the GTSA that can be compared with the other non-comparative linear-time integer sorters. The growing-tree sorting algorithm proceeds in two phases. The first phase is the growing-tree construction phase, in which all the input numbers get inserted in their proper locations in the tree. The second phase is the growing-tree traversal phase, in which all the input numbers get retrieved and inserted in their corresponding locations in the sorted output array. In this section we describe these two phases.

3.1 The Growing-Tree Construction Phase

The root of the growing tree is a sign node that has two child-nodes as described in Fig4.(a). The left child-node points to the negative sub-tree, i.e. the sub-tree of negative input numbers, whereas the right child-node points to the positive sub-tree, i.e. the sub-tree of positive integers. The root of the

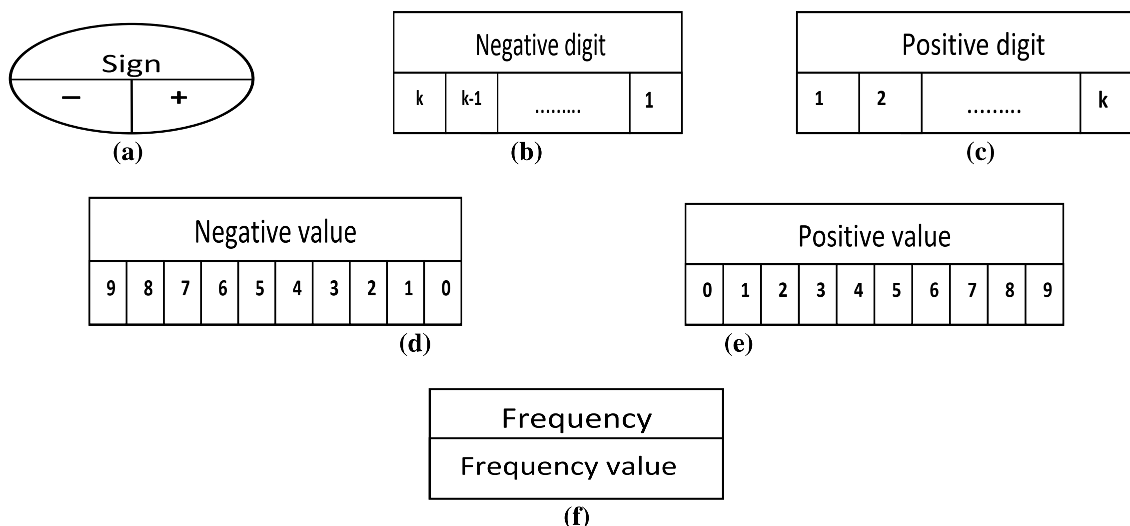

Fig. 4 The nodes that compose a growing tree

```
Growing_Tree_Construction (A,D,T)
Input: unsorted integer-array (A),the maximum number of digits per
input number (D)
Output: Growing-tree of the input numbers T
Begin
Create a Sign Node and denote it by SN
Create a negative digit-node with D pointers, denote it by ND
Create a positive digit-node with D pointers, denote it by PD
Let prt be a pointer to a value node
SN.left_child = ND
SN.right_child = PD
For i = 1 to length(A) do
d = number of digits in A[i]
if( A[i]> 0)
if(PD[d] == null)
create a Positive-value node, PV
PD[d]= PV
ptr = PD[d]
Insert (A[i], T, ptr)
else
if(ND[d] == null)
create a Negative-value-node, NV
ND[d]= NV
ptr = ND[d]
Insert (A[i], T, ptr)
End
```

Fig. 5(a) The Growing-Tree Construction Phase

negative sub-tree is the negative digit-node, which is described in Fig. 4(b). This node is an array of pointers, where the ith pointer is the root of the sub-tree of all negative input numbers with *i* digits. The pointers of the negative digit-node are ordered in a way that would help retrieve the negative numbers in the correct order during the growing-tree traversal phase. Similarly, the positive digit-node is the root of the sub-tree of all positive input numbers. This node, as depicted in Fig.4(c) is an array of pointers where the ith pointer is the root of the sub-tree of all positive input numbers with *i* digits. The negative value-node, as described in Fig.4(d), is an array of 10 pointers, each of which designates a digit between 0 and 9. Again, the order of the pointers in the negative-value node helps retrieve the negative numbers in the correct order during the second phase of this algorithm. Similarly, a positive-value node, as depicted in Fig. 4(e), is an array of 10 pointers. The *i*th pointer designates a digit with value *i*, where *i* in the range 0 to 9 inclusive. Each path in the growing tree designates an input number and is terminated by a leaf node, called the frequency node. This node records the occurrences of that number in the input and is described in Fig.4(f). Now, assume that you want to insert an *i*-digit positive integer, called *x*, into the growing tree for the first time. The integer would be inserted into the positive subtree rooted at the *i*th pointer in the positive-digit node. To do so, *x* would be split into the digits composing it, and each digit would get inserted in its corresponding positive value node starting with the most significant digit. This would create a path in the growing tree that designates *x*. The frequency node is appended to this path as a leaf node and assigned the value of one. For all the subsequent occurrences of the same integer, if any exist, the path would be traversed all the way down to the frequency node which would have its value incremented by one to account for each occurrence of *x*. Figures 5(a) and 5(b) describe the growing-tree construction phase.

3.2 The Growing-Tree Traversal Phase

In this phase, post-order tree traversal is used to retrieve the integers in the sorted order. In the post-order tree traversal, the left sub-tree is visited first, then the right sub-tree is visited next, and finally the root is visited. Each of the

```
Insert(a,T,ptr)
Input: integer (a), growing-tree(T),
ptr: pointer to the node under which a should be inserted
Output: T after inserting integer a into it
Begin
Let a be equal to a_d a_(d-1)...a_1 ,where d is the number of digits in a
For j = d downto 1 do
If( ptr[a_j]==null and a >= 0)
Create new positive-value-node call it PV
ptr[a_j] = PV
If( ptr[a_j]==null and a < 0)
Create new positive-value-node call it PV
ptr[a_j] = NV
ptr = ptr[a_j]
if (ptr does not point to a frequency node)
append a frequency node
to ptr and initialize its value to one
else
increment the frequency value by 1
End
```

Fig. 5(b) The Growing-Tree Construction Phase(continued)

aforementioned sub-trees is traversed, recursively, in a similar fashion. The pointers in the digit nodes as well as the value nodes are ordered properly. Thus, post-order tree traversal allows us to retrieve the nodes that actually have smaller values before the ones with larger values. When a frequency node is reached, the retrieved integer would get copied to the output array as many times as recorded in the frequency node. Figure 6 describes the growing-tree traversal phase.

3.3 A Numerical Example

In this section, we illustrate the GTSA algorithm described in section 3 via the means of an example. Assume you want to sort the numbers: 5117, -102, 675, 3, -194, and 3 via GTSA. It is quite obvious that the maximum number of digits per input number is 4. Firstly, each number in the input needs to be inserted into the growing tree. Figure 7 depicts the growing tree after inserting 5117, and Fig. 8 describes the growing tree after inserting the remaining numbers.

In order to retrieve the sorted output, the growing tree depicted in Fig.8 would be traversed in a post-order fashion. Thus, the first number that would be retrieved is -194 followed by -102. Next, number 3 would be traversed twice since its frequency is 2 resulting in the sub-array 3, 3. Afterwards, the number 675 would be traversed once, followed by the number 5117. The resulting sorted output array is -194, -102, 3, 3, 675, 5117.

3.4 The GTSA Time and Memory complexities

The time and memory complexities of the GTSA algorithm are depicted in theorems 1 and 2 respectively.

Theorem 1

It takes the GTSA algorithm $\theta(dn)$ time to sort n d-digit numbers in both the best and worst-case scenarios.

Proof

Let $F(n)$ denote the time complexity of the GTSA algorithm. Also, let $f(n)$ denote the time complexity of the growing-tree construction phase, and let $g(n)$ denote the time complexity of the growing-tree traversal. Obviously, $F(n) = f(n) + g(n)$.

Let us first consider the best-case scenario, which happens when the n d-digit numbers are the same number which we will call x. In this case, it would take $\theta(d)$ time to insert the d digits of x. Then, the constructed path would be traversed all the way to the frequency node to increment its value for the remaining occurrences of x in the input. Thus, the construction procedure would take $\theta(dn)$ time to accomplish its job in the best-case.

```
Traverse_Growing_Tree(T,A)
Input: Growing-tree(T)
Output: sorted output array(A)
Begin
Let D be the maximum number of digits per integer
For i = D downto 1 do
  Let ptr = ND[i]
   For each path, p, rooted at ptr
     Apply post-order traversal to p and retrieve the
       corresponding integer, denote it by a
     Copy -a to A c times, where c is the value of the frequency
       node appended to p
For i = 1 to D  do
     Let ptr = PD[i]
     For each path, p, rooted at ptr
       Apply post-order traversal to p and retrieve the
       corresponding integer, denote it by a
       Copy a to A c times, where c is the value of the frequency
       node appended to p
End
```

Fig. 6 The Growing-Tree Traversal Phase

In the worst-case scenario, the n d-digit numbers are distinct. Notice that the amount of time it takes to insert a digit is constant. Thus, it would take $\theta(d)$ time to insert each input number and set its frequency node value to one. This means that the overall time it would take to insert the n d-digit numbers would be $\theta(dn)$. As a consequence, $f(n)$ is $\theta(dn)$ both in the best-case and the worst-case scenarios.

The best-case scenario of the growing-tree traversal phase happens when the same d-digit number, called x, is repeated n times. Then, it would take d operations to retrieve the digits composing x from the growing tree and n operations to copy it into the sorted array n times. Hence, it takes $\theta(d+n)$ time to traverse the growing-tree in the best-case scenario. The worst-case scenario occurs when the input contains n distinct d-digit numbers. There, it would take $\theta(d)$ time to retrieve each input number from the tree and copy it into the sorted output array. Thus, it would take $\theta(dn)$ time to perform the growing tree traversal phase in the worst case scenario. Therefore, $g(n)$ is $\theta(d+n)$ in the best case and $\theta(dn)$ in the worst case.

Based on the arguments above, it can be concluded that $F(n)$, i.e. the time complexity of the GTSA algorithm, is $\theta(dn)$ both in the best-case and worst-case scenarios. ■

Theorem 2

The space complexity of the GTSA in the best case is $\theta(d)$ and is $\theta(dn)$ in the worst case.

Proof

Let $F(n)$ denote the space complexity of the GTSA sorting algorithm. Also, let $f(n)$ denote the space complexity of the growing-tree construction phase and let $g(n)$ denote the space complexity of the growing-tree traversal phase. Obviously, $F(n) = f(n) + g(n)$.

In the best-case scenario of the growing-tree construction phase, the same d-digit number, called

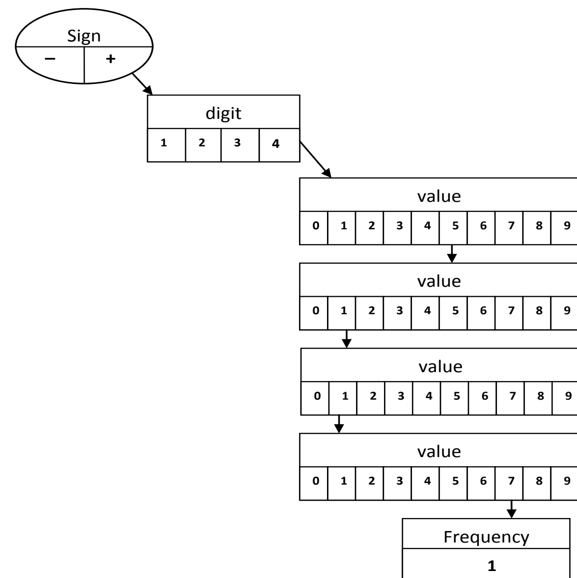

Fig. 7 Inserting 5117 in the growing tree

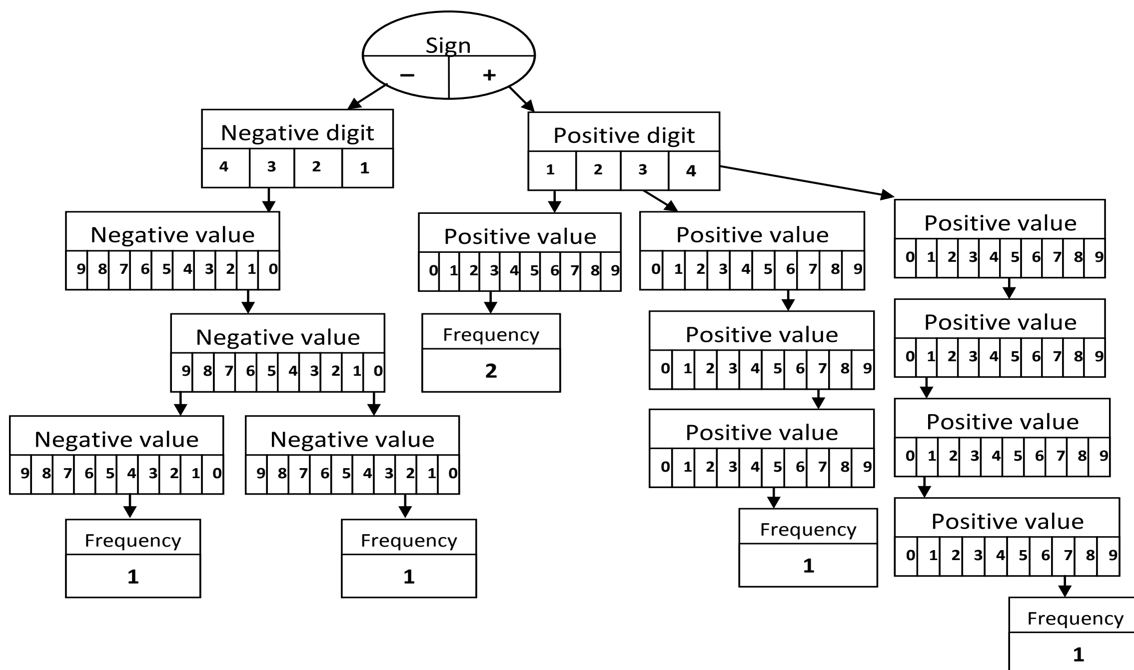

Fig. 8 The growing tree after inserting 5117, -102, 675, 3, -194, and 3

x, appears n times. It takes θ(d) memory to store the digits of x in their proper positions in the growing tree and to append one frequency node that contains the value of n. Thus, the growing tree contains one path in this case and, consequently, $f(n)$ is θ(d) . In the worst case scenario, the input consists of n distinct d-digit numbers. Each number requires θ(d) memory locations to form its distinct path in the tree. Consequently, $f(n)$ is θ(dn) in the worst case.

In the best and worst case scenarios, the traversal phase only copies the retrieved numbers to the output array using a small constant amount of memory. Besides, we can copy the retrieved integers back into the input array. Thus, $g(n)$ is θ(1) for both the best and the worst cases. Consequently, $F(n)$ is θ(d) in the best case and θ(dn) in the worst case. ■

4 Experimentations

In order to assess the performance of the GTSA algorithm, a Java program was developed together with two other enhanced Java implementations for the radix and counting sorts. The enhanced implementations augmented the classical radix and counting sorting algorithms with the instructions necessary for handling negative integers as well. Table 1 summarizes the asymptotic behaviors of the three non-comparative sorting algorithms, where the input range, the number of digits per input number, and the input size are denoted by k, d, and n respectively.

If we assume that k is proportional to n and that d is a constant, then the three algorithms will have comparable memory and execution times complexities. However, this does not necessarily guarantee that the three sorters are actually as good as each other in terms of space and time. Determining which algorithm is the best not only depends on the asymptotic behaviors, but also on the practicality of the implementations. The complexity constants together with the implementation details have profound effect on determining the actual performance of any algorithm. Here we try to figure out the circumstances under which the GTSA would actually excel and outperform both radix sort and counting sort algorithms in terms of space and time.

Three sets of experiments were carried out to help determine the actual performance of the three

Table 1 The asymptotic behavior of counting, radix, and GTSA sorts

Algorithm	Worst-case memory complexity	Worst-case time complexity
Radix sort	θ (k+n)	θ (d(n+k))
Counting sort	θ (k+n)	θ (k+n)
GTSA	θ(dn)	θ(dn)

sorters. The performance metrics considered were the memory consumptions (in Megabytes) and the execution time (in milliseconds). Firstly, Java was used to randomly generate 70 *d*-digit files. A *d*-digit file contains *N* positive and negative integers with no more than *d* digits per integer. The values of *d* varied between 1 and 10 inclusive, and the values of *N* were: 100, 500, 1000, 5000, 10000, 50000, and 100000. The radix sort, count sort, and GTSA algorithms were all used to sort each of the 70 files 10 times. The corresponding execution times and memory consumptions were recorded for each run. Then, the 10 execution times and memory consumption values recorded for sorting a given file were averaged and that process was repeated for every file.

Figure 9 describes the memory consumptions of the three sorters for the input sizes: 100, 1000, 10000, and 100000, considering all the *d*-digit files, where *d* was in the range 2 to 7 inclusive. The reason why *d* was limited to 7 is that the counting sort regularly generated heap-memory-out-of-space exceptions whenever *d* exceeded 7.

Examining Fig.9 shows that the GTSA is, in general, more memory-conservative than its counterparts. Memory consumption of the GTSA is pretty close to the counting sort memory consumption for input sizes less than or equal to 100. For larger input sizes the GTSA algorithm consumes smaller amount of memory than counting sort for all examined input ranges. Similarly, the memory consumption of GTSA and radix sort are pretty comparable when the input sizes are less than or equal to 1000. The same figure shows that GTSA consumes a smaller amount of memory than radix sort for all examined input sizes greater than 1000 as long as *d* is less than 6.

The results depicted in Fig. 9 show that GTSA becomes significantly more memory-conservative as the input size increases for most of the depicted values of *d*. Consider, for instance, the memory consumptions of the three algorithms when *N*=1000 and *d* = 4. In this case, GTSA uses almost 100% of the memory used by the two other algorithms to sort the input. When *N* is increased to 10000 and *d* remains 4, GTSA consumes 50% of the memory used by radix sort and by counting sort to accomplish the sorting task. For *N*=100000 and the same value of *d*, GTSA uses 30% of the memory used by radix sort , and 22% of the memory used by counting sort to accomplish the sorting task.

Figure 10 depicts the execution times of the three sorters for the input sizes: 100, 1000, 10000, and 100000, considering the *d*-digit files where *d* is between 2 and 7 inclusive. Again, *d* was limited to 7 due to the fact that the counting sort regularly generated heap-memory-out-of-space exceptions wherever *d* exceeded the value of 7.

Fig. 10 shows that no specific sorter is faster than the others in all situations. Careful examination of the same figure, though, shows that GTSA outperforms the counting sort execution time when *d* exceeds 6 digits and when *N* is less than 100000. For *N* = 100000, the GTSA is faster than counting sort whenever *d* is greater than or equal to 7. Thus, GTSA is faster than the counting sort when the range of the input values is relatively large, regardless of the input size.

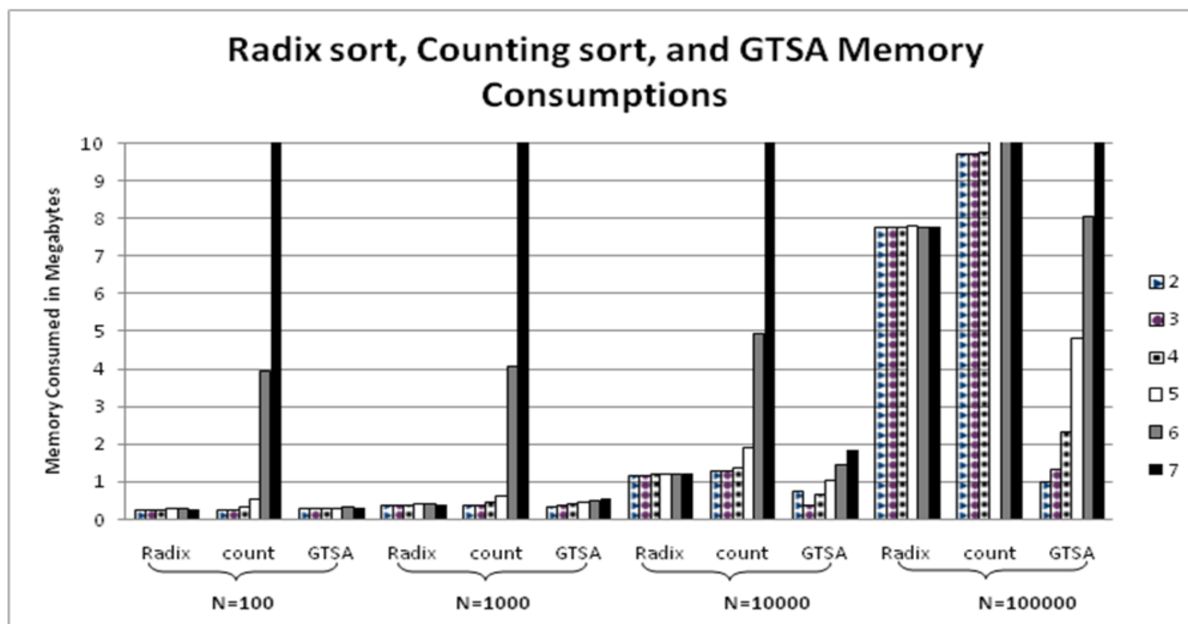

Fig. 9 The memory consumption of the three linear-time sorters.

Examining the same figure shows that GTSA is faster than radix sort only when the input size *N* exceeds 10000 and the number of digits per integer is no more than 5. Thus, GTSA runs faster than the radix sort only when the input size is relatively large and the range of input numbers is relatively small.

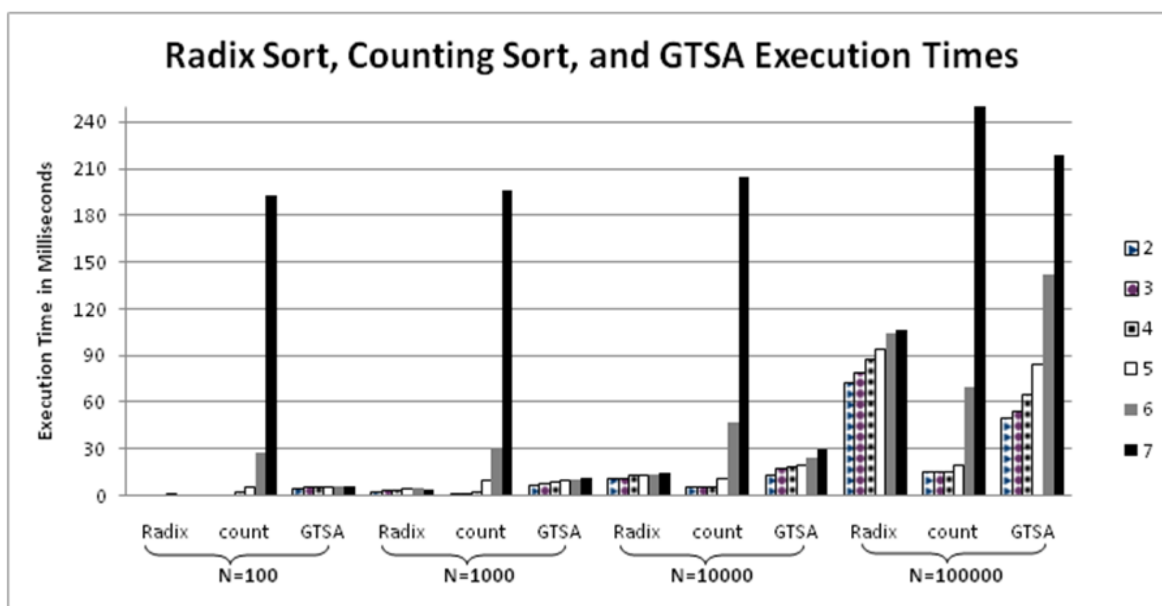

Fig. 10 The execution times of the three linear-time sorters.

4 Conclusion and Future Work

Here we introduce the Growing-Tree Sorting Algorithm(GTSA), a non-comparative linear-time sorter that handles positive as well as negative integers. The GTSA proceeds in two phases: the construction phase and the traversal phase. In the construction phase each distinct integer gets inserted in the tree digit-by-digit forming a path. This path ends with the frequency node that contains the number of occurrences of the designated integer. In the traversal phase the growing tree is traversed in a post-order fashion, and all the integers get retrieved in the correct order and inserted into their corresponding slots in the sorted output array. It can be shown that the GTSA runs in no more than $\theta(dn)$ time and uses no more than $\theta(dn)$ memory.

A set of experiments was carried out to compare the GTSA with two enhanced versions of radix and count sorts that handled negative as well as positive integers. The experimentations showed that GTSA consumed fewer memory locations than radix sort when the maximum number of digits per input was 6 or less. They also showed that GTSA consumed fewer memory locations than counting sort when the maximum number of digits per input was 4 or more. Besides, the experimentations showed that GTSA was faster than radix sort when the maximum number of digits per input varied between 1 and 5 digits for 10000 or more integers. The GTSA was found to be faster than counting sort when the maximum number of digits per integer exceeded 5 for all examined input sizes.

GTSA saves more memory and runs faster when the input numbers exhibit repetition. Hence, additional set of experiments that exploit inputs with this characteristic can be conducted. Also, a signed floating-point version of GTSA can be developed and compared with similar already-existing linear - time sorters. These algorithms execution times and memory consumptions can be studied and contrasted under various input scenarios to determine the conditions under which GTSA would excel. Also, it would be interesting to develop a parallel version of GTSA and compare it with the already existing parallel liner-time sorters.

References:

[1] Cormen, T.H., Leiserson, C.E., Rivest, R. L. ,and Stein, C. *Introduction to Algorithms*. The MIT Press, USA, Third Edition, (2009).

[2] Hoare, C. A. Quicksort, *Computer Journal*. Vo.**5, No.**1, 1962, pp. 10-15.

[3]Williams,J.W.,Algorithm232 (HEAPSORT). *Communications of the ACM*, Vol.**7,** 1964, pp. 347-348.

[4] Knuth, D. E. *The Art of Computer Programming* Vol. 3: *Sorting and Searching*, Addison-Wesley, Reading Mass., USA, Second Edition, 1998.

[5] Shell, D. L.,A high-speed sorting procedure. *Communications of the ACM* , Vol.**2, No.**7, 1959, pp. 30 – 32.

[6] Bentley, J.L. and Sedgwick, R., Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, USA, January 4th -January 7th, 1997, pp. 36-369, SIAM, Philadelphia PA, USA.

[7] Wang, Y., A New Sort Algorithm: Self-Indexed Sort. *Communications of ACM SIGPALN*, Vol. **31, No.** 3, 1996, pp. 28-36.

[8] Han Y., Improved fast Integer sorting in linear space. *In proceedings of the 12th ACM SIAM Symposium on Discrete Algorithms*, Washington D.C., USA, January 7th-January 9th, ,2001, pp. 793-796, SIAM, Philadelphia PA, USA.

[9] Han Y. and Thorup M. , Integer Sorting in O($n\sqrt{\log\log n}$) Expected Time and Linear Space. *Proceedings of the 43rd Symposium on Foundations of Computer Science*, Vancouver, BC, Canada, November 16th –November 19th, 2002, pp. 135-144, IEEE Computer Society, Washington, DC, USA.

[10] Martin, W.A. Sorting, *ACM Computing Surveys*. Vol. **3, No.** 4, 1971, pp. 147-174.