International Journal of
Emerging Research in Management &Technology
ISSN: 2278-9359 (Volume-2, Issue-12)

Research Article | December 2013

Analysis of Non-Comparison Based Sorting Algorithms: A Review

Rohit Joshi, Govind Singh Panwar, Preeti Pathak
Dept. of CSE, GEHU, Dehradun
India

Abstract –

An algorithm is combination of sequence of finite steps to solve a particular problem. Any algorithm can be analysed in terms of time and space. Sorting algorithms involves rearranging information in a certain order, either in ascending order or descending order with numerical data and alphabetically with character data. Now a days, there are a number of programming applications used in computer science which apply sorting techniques to solve a problem. Non-comparison based sorting algorithms make assumptions about the input. All elements of the input are required to fall within a range of constant length in order to ensure linear time complexity. On the other hand comparison based sorting algorithms make no assumption about the input and are able to address any case. The goal of this paper is too reviewed on different non-comparison based sorting algorithms. In this paper we check the performance and comparison of all non-comparison based sorting algorithm for a given input sequence.

Keywords - bucket sort, counting sort, radix sort, MSD radix sort, LSD radix sort.

I. INTRODUCTION

One of the fundamental problems of computer science is ordering a list of items. There is a plethora of solutions to this problem, known as sorting algorithms. Researchers have attempted in past to develop algorithms efficiently in terms of optimum memory requirement and minimum time requirement i.e., Time or Space Complexities. Algorithmic complexity of sorting algorithm is generally written in a form known as Big-O notation, where the O represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against. For example, O(n) means that an algorithm has a linear complexity [5]. Together with searching, sorting is probably the most used algorithm in Computing, and one in which, statistically, computers spend around half of the time performing All the comparison based sorting algorithms use comparisons to determine the relative order of elements. e.g., insertion sort, merge sort, quick sort, heap sort. The best worst-case running time that we've seen in [1], for comparison sorting is O(nlgn). There are sorting algorithms that run faster than O(nlgn) time but they require special assumptions about the input sequence to be sort. Examples of sorting algorithms that run in linear time are counting sort, radix sort and bucket sort. Counting sort and radix sort assume that the input consists of integers in a small range. Whereas, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval. Since we already know that the comparison-based sorting algorithm can do sorting in Ω(nlgn), it is not difficult to figure out that linear-time sorting algorithms use operations other than comparisons to determine the sorted order.

II. SORTING IN LINEAR TIME

A. Comparison sort:

- Lower bound: Ω(nlgn).

B. Non comparison sort:

- Bucket sort, counting sort, radix sort
- They are possible in linear time (under certain assumption).

III. NON-COMPARISON SORT

A. Bucket sort:

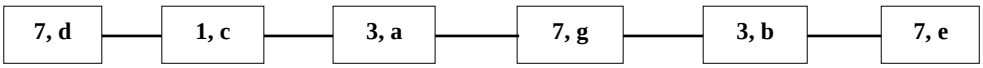
Bucket Sort is a sorting method that subdivides the given data into various buckets depending on certain characteristic order, thus partially sorting them in the first go. Then depending on the number of entities in each bucket, it employs either bucket sort again or some other ad hoc sort. Bucket sort runs in linear time on an average. Bucket sort is stable. It assumes that the input is generated by a random process that distributes elements uniformly over the interval 1 to m.

Analysis of BUCKET-SORT(A):

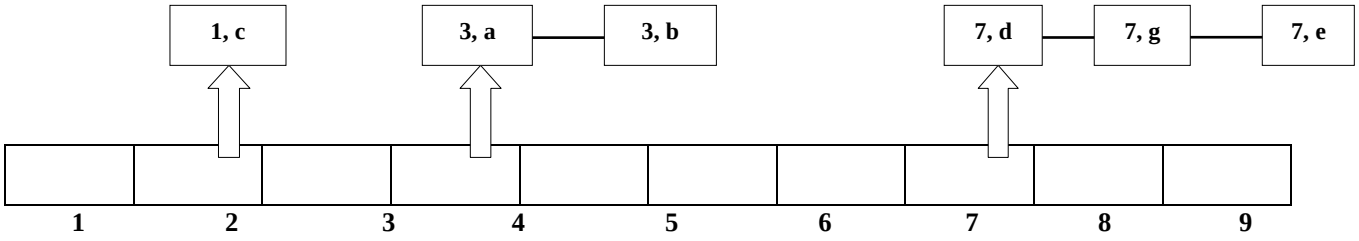
1.	n ← length[A]	Ω(1)
2.	**for** i ← 1 to n	O(n)
3.	**do** insert A[i] into bucket B[⌊nA[i]⌋]	Ω(1) (i.e. total O(n))
4.	**for** i ← 0 to n-1	O(n)
5.	**do** sort bucket B[i] with insertion sort	$O(n_i^2)$ ($\sum_{i=0}^{n-1} O(n_i^2)$)
6.	Concatenate bucket B[0],B[1],…,B[n-1]	O(n)

Where n_i is the size of bucket B[i].
Thus $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$
$= \Theta(n) + nO(2-1/n) = \Theta(n)$. **Beat Ω(nlgn)**

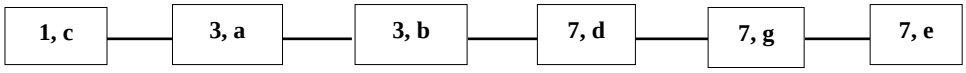
Following are the procedure to sort a given set of element
Key range: [0,9]



Phase 1: Filling the buckets and sorting the bucket with insertion sort



Phase 2: emptying the buckets into the list



- Drawback: For a non-uniform distribution this fails to be so efficient because the empty buckets would unnecessarily cause slowdown.
- Solution: For large data sets with non-uniform distribution, find the ranges with maximum density (again above a particular threshold) and apply bucket sort there. For rest of the parts use some other algorithm.

B. Counting sort:

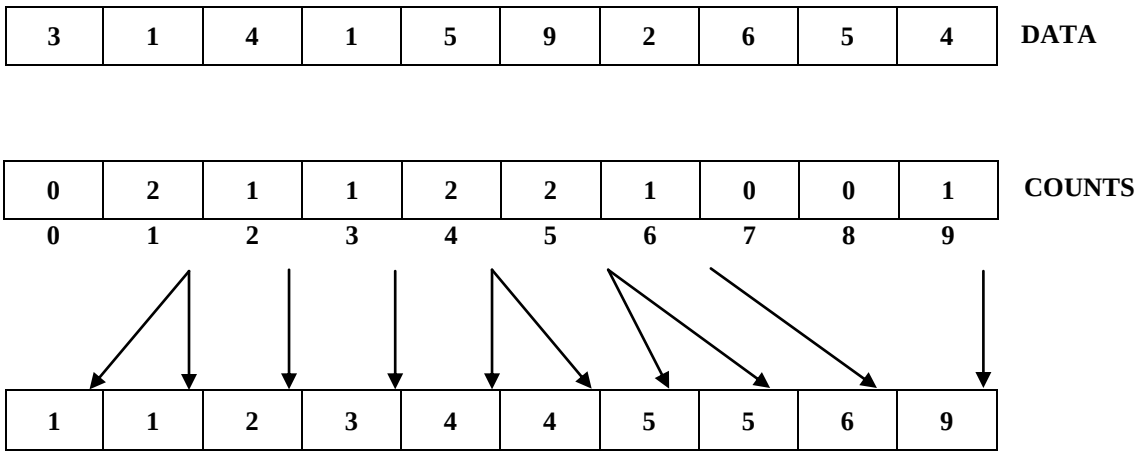
Counting sort is an algorithm used to sort data whose range is pre-specified and multiple occurrences of the data are encountered. It is possibly the simplest sorting algorithm. The essential requirement is that the range of the data set from which the elements to be sorted are drawn, is small compared to the size of the data set. Counting sort works by determining how many integers are behind each integer in the input array A. Using this information, the input integer can be directly placed in the output array B. Counting sort is stable.

Analysis of COUNTING-SORT(A,B,k):

1.	**for** i ← 0 to k	Θ(k)
2.	**do** C[i] ← 0	Θ(1)
3.	**for** j ← 1 to length[A]	Θ(n)
4.	**do** C[A[j]] ← C[A[j]]+1	Θ(1) (Θ(1) Θ(n)= Θ(n))
5.	// C[i] contains number of elements equal to i.	Θ(0)
6.	**for** i ← 1 to k	Θ(k)
7.	**do** C[i]=C[i]+C[i-1]	Θ(1) (Θ(1) Θ(n)= Θ(n))
8.	// C[i] contains number of elements ≤ i.	Θ(0)
9.	**for** j ← length[A] **downto** 1	Θ(n)
10.	**do** B[C[A[j]]] ← A[j]	Θ(1) (Θ(1) Θ(n)= Θ(n))
11.	C[A[j]] ← C[A[j]]-1	Θ(1) (Θ(1) Θ(n)= Θ(n))

Total cost is Θ(k+n), suppose k=O(n), then total cost is Θ(n). **Beat Ω(nlgn).**

Following are the procedure to sort a given set of element:



This sort works optimally in the case when the data is uniformly distributed

Example of efficient use of Counting-sort:

- Ranking 200 students on the basis of their score out of 100.
- Sorting 1500 employees with respect to the occurrence of their birthday in a year.

Drawback:

- If the range m>>n (where n is the number of data while m is the range of the data), the complexity will not be linear in n and thus this sort does not remain useful anymore. This is because chances of introduction of gaps, that is counters for those elements which do not exist in the list, will cause a higher space complexity.

C. Radix sort:

A radix sort is an algorithm that can rearrange integer representations based on the processing of individual digits in such a way that the integer representations are eventually in either ascending or descending order. Integer representations can be used to represent things such as strings of characters (names of people, places, things, the words and characters, dates, etc.) and floating point numbers as well as integers. So, anything which can be represented as an ordered sequence of integer representations can be rearranged to be in order by a radix sort.

Most digital computers internally represent all of their data as electronic representations of binary numbers, so processing the digits of integer representations by groups of binary digit representations is most convenient. Two classifications of radix sorts are:

- Least significant digit (LSD) radix sort.
- Most significant digit (MSD) radix sort.

LSD radix sorts process the integer representations starting from the least significant digit and move the processing towards the most significant digit

MSD radix sorts process the integer representations starting from the most significant digit and move the processing towards the least significant digit.

The integer representations that are processed by sorting algorithms are often called "keys," which can exist all by themselves or be associated with other data. LSD radix sorts typically use the following sorting order: short keys come before longer keys, and keys of the same length are sorted lexicographically. This coincides with the normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

MSD radix sorts use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations. A sequence such as b, c, d, e, f, g, h, i, j, ba would be lexicographically sorted as b, ba, c, d, e, f, g, h, i, j.

If lexicographic ordering is used to sort variable-length integer representations, then the representations of the numbers from 1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9, as if the shorter keys were left-justified and padded on the right with blank characters to make the shorter keys as long as the longest key for the purpose of determining sorted order.

Analysis of RADIX-SORT(A,B,k):

1. for i ← 1 to d
2. do use a stable sort to sort Array A on digit i // counting sort will do the job//

The running time depends on the stable sort used as an intermediate stage of the sorting algorithm. When each digit is in the range 1 to k, and k is not too large, counting sort can be taken. In case of counting sort, each pass over n d-digit numbers takes O(n + k) time. There are d passes, so the total time for radix sort is Θ(n + k) time. There are d passes, so the total time for Radix sort is Θ(d (n+ k)). When d is constant and k = Θ(n), the Radix sort runs in linear time.

Following are the procedure to sort a given set of element using LSD sort:

523	153	088	554	235

Phase 1: Sort the list of numbers according to the ascending order of least significant bit. The sorted list is given by:

523	153	554	235	088

Phase 2: Then sort the list of numbers according to the ascending order of 1st significant bit. The sorted list is given by:

523	235	153	554	088

Phase 3: Then sort the list of numbers according to the ascending order of most significant bit. The sorted list is given by:

088	153	235	523	554

Following are the procedure to sort a given set of element using MSD sort:

170	045	075	090	002	024	802	066

Phase 1: Sort the list by most significant digit. It gives:

045	075	090	002	024	066	170	802

Phase 2: Then sort the list by the next significant digit (10s place):

002	802	024	045	066	075	170	090

Phase 3: Then sort the list by least significant digit (1st place):

002	024	045	066	075	090	170	802

Drawback:

- Speed of radix sort largely depends on the inner basic operations and if operations are not efficient enough radix sort can be slower than some other algorithms such as quick sort or merge sort. These operations include the insert delete function of the sublists and the process of isolating the digit we want.
- Radix Sort can also take up more space than other sorting algorithms, since in addition to the array that will be sorted; we need to have a sub list for each of the possible digits or letters. If pure English words are sorted then at least 26 sublists are needed and if alpha numeric words are sorted then probably more than 40 sublists are required.
- Radix Sort is also less flexible as compared to other sorts.

Table1: Strength and weakness of non-comparison based sorting algorithm

Name	Average Case	Worst Case	$n<<2^k$	Advantage/Disadvantage
Bucket Sort	O(n,k)	$O(n^2.k)$	No	1. Stable, fast. 2. Valid only in range O to some maximum value M. 3. Used in special cases when the key can be used to calculate the address of buckets.
Counting Sort	$O(n+2^k)$	$O(n+2^k)$	Yes	1. Stable, used for repeated value. 2. Often used as a subroutine in radix sort. 3. Valid in the range (O k] Where k is some integer.
Radix Sort	O(n.k÷s)	O(n.k÷s)	No	1. Stable, straight forward code. 2. Used by the card-sorting machines. 3. Used to sort records that are keyed by multiple fields like date (Year, month and day).
MSD Radix Sort	O(n.k÷s)	O(n.k÷s)	No	1. Enhance the radix sorting methods, unstable sorting. 2. To sort large computer files very efficiently without the risk of overflowing allocated storage space. 3. Bad worst-case performance due to fragmentation of the data into many small sub lists. 4. It inspects only the significant characters.
LSD Radix Sort	O(n.k÷s)	$O(n.k÷s.2^s)$	No	1. It inspects a complete horizontal strip at a time. 2. It inspects all characters of the input. 3. Stable sorting method.

IV. CONCLUSION

In this paper, we have studied and analysed about non-comparison based sorting algorithms. We analyse the time complexity of each algorithms with time taken by each step of algorithm. After analysing the time we have seen that non-comparison based sorting algorithms Beat O (nlgn) by O(n).

ACKNOWLEDGMENT

Our thanks to MR. NAVIN GARG (Head of Department) of Computer Science, for providing necessary infrastructure for the research work and helping us to write this paper. We would also like to thanks to our colleague MR.PRASHANT BADONI who gave us a valuable support in writing this paper.

REFERENCES

[1] Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Second Edition (Prentice Hall of India private limited), New Delhi-110001.
[2] Aho, A.V., Hopcroft, J.E., Ullman, J. D., 1974."The Design and Analysis of Computer Algorithms" Addison-Wesley.
[3] A Comprehensive Note on Complexity Issues in Sorting Algorithms Advances in Computational Research, ISSN: 0975–3273, Volume 1, Issue 2, 2009, pp-1-09
[4] Comparison of Sorting Algorithms based on Input Sequences International Journal of Computer Applications (0975 – 8887)Volume 78 – No.14, September 2013.
[5] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Journal of Computer and System Sciences, vol 57, pp 74-93, 1998..
[6] http://stackoverflow.com/questions/17641858/linear-sorting-algorithms.
[7] Parag Bhalchandra, Proliferation of Analysis of Algorithms with application to Sorting Algorithms, M.Phil Dissertation, VMRF, India, July 2009.