A Survey on various comparison and non-comparison based sorting algorithms

Mohith Lalita Kumar Parvataneni[1]
Master of Science In Information Systems, Northeastern University, Boston, MA
Email: parvataneni.m@northeastern.edu

Trinadh Phani Ginjupalli[2]
Master of Science In Information Systems, Northeastern University, Boston, MA
Email: ginjupalli.t@northeastern.edu

Anil Kumar Potharaju[3]
Master of Science In Software Engineering Systems, Northeastern University, Boston, MA
Email: @northeastern.edu

Abstract—

Nowadays, sorting plays a major role and is one of the most studied problems in computer science. Sorting is the process of rearranging the data in a particular order which is either numerical order or lexicographical order. Mainly there are two types of sorting algorithms : comparison sorts and non-comparison sorts. One of the main differences between these two types of sorting algorithms is speed. Non comparison sorts are generally faster than comparison sorts. Speed of comparison sorts is Linearithmic(O(NlogN)) whereas for non-comparison sorts it is Linear(O(N)). The best case space complexity for a comparison sort is O(1) whereas for a non-comparison based sorting space complexity is O(N).

In this paper, we are going to present a brief discussion on some of the comparison and non-comparison based sorting algorithms Quick Sort, Dual-Pivot Quick Sort, Tim Sort, Radix Sort and discuss about their pros and cons by comparing their performances in terms of execution time and space utilization. We are also going to discuss techniques of Forward Radix Sort and Radix Exchange Sort (MateSort) which are enhancements of classic radix sort and are faster compared to comparison based sorting algorithms.

I. INTRODUCTION

Data is increasing day-by-day and the efficiency of algorithms plays a crucial role in processing this huge amount of data. Many attempts have been made to analyze the complexity of sorting algorithms and different sorting algorithms have been proposed. Even though sorting is one of the extremely studied problems in computer science, it remains the most general integrative algorithm problem in practice [1].

Every sorting algorithm has its own pros and cons. For instance insertion sort is efficient for sorting a small number of items whereas quick sort sort would be efficient for larger numbers of items. The performance of every sorting algorithm relies upon the data being sorted and the machine used for sorting [1].

Selecting a good sorting algorithm depends upon several factors such as the size of the input data, available main memory, disk size, the extent to which the list is already sorted and the distribution of values [1]. To measure the performance of different sorting algorithms we need to consider various factors such as the number of operations, the execution time and the space required for the algorithm [2].

There are many sorting algorithms [3] such as BubbleSort, InsertionSort and SelectionSort which are simple to develop and to realize, but they have a weak performance (time complexity is O(n2)). Several researchers have used MergeSort, HeapSort and QuickSort with O(nlog(n)) time complexity to resolve the restricted performance of these algorithms [3]. Authors concluded that Quicksort is the fastest sorting algorithm.

Unfortunately, any algorithm which has a time complexity of at least O(NlogN) remains slow, when used with a large number of keys. In such cases, Radix sort can be used. Unlike comparison sorts, radix sort orders keys by iteratively partitioning keys based on successive bytes within keys.[4] But radix sort takes more space compared to other sorting algorithms.

In Java, dual pivot quicksort is used for primitives whereas Tim sort is used for objects which is a derived algorithm of merge sort(bottom-up merge sort). Tim sort is most commonly used for partially sorted data. Quick sort linearithmic time complexity and Tim sort is not effective for unordered data.

II. SURVEY OF SORTING ALGORITHMS

In this section, we are going to describe various sorting algorithms. Each algorithm is explained and their efficiencies are discussed. The sorting algorithms that we are going to study are

- Quick Sort & Dual-Pivot Quick Sort
- Tim Sort
- Radix Sort (LSD and MSD)

Quick Sort:

Quick sort [5] is a comparison based sort and the fastest algorithm on average among other common sorting algorithms. It does not require extra memory like merge sort for sorting an array. It is a widely used sorting algorithm in real time applications with large data sets. It uses a divide and conquer approach to solve the problem. It is an in-place sorting algorithm and it preserves the order of elements after sorting, hence it is considered as a stable sort. The worst case complexity of insertion sort is O(N^2) and the best case and average case complexity is O(NlogN). It is difficult to implement quick sort for smaller arrays, so we implement quick sort often with insertion sort for smaller arrays.

Pros and cons of quick sort:

Quick sort is fast and efficient for large data sets whereas it is inefficient for small data sets , if the elements of the array are already sorted and if every element of that array is equal. Quick is space expensive for large data sets i.e, space complexity will be O(NlogN) for recursive function calls. Also more cache misses in quick sort.

Dual Pivot Quicksort:

It is also called Yaroslavskiy's algorithm. Dual pivot quick sort is faster than traditional quick sort and offers O(NlogN) complexity even in the worst case. It reduces the number of cache misses. It is the combination of insertion sort and quicksort. It uses 5 % less key comparisons, but 80 % more swaps than classic Quicksort. Unless comparisons are very expensive, classic Quicksort is more efficient[6]. It is more complicated than classic quick sort, comparisons and swaps are required for both the pivots.

Tim Sort:

It is also a comparison based sorting technique. It is defined as a hybrid sorting technique as it combines Merge sort and Insertion sort. The main idea of TimSort is to design a merge sort that can exploit the possible “non randomness” of the data, without detecting it beforehand and without damaging the performances on random-looking data. It is not an in-place algorithm as it needs extra memory for merging. The best case complexity of Tim Sort is O(N) whereas the worst case and average case complexity is O(NlogN). If the size of the array is greater or equal than 64 elements, then MergeSort will be considered; otherwise, InsertionSort is selected in the sorting step.

Pros and cons of Tim sort:

It is a stable sorting technique and is faster than merge sort. It’s worst case complexity is also linearithmic.

Radix Sort:

Radix sort is a non-comparison based sorting technique. It is done by sorting the positions of digits/letters. As theory says, Radix sort is the most efficient sorting algorithm to sort strings. The two main implementations of Radix sort are Least Significant Digit (LSD) and Most Significant Digit based radix sorts. The LSD method sorts the array based on the least significant value and continues until the most significant value is reached, whereas the MSD method starts with most significant value until least significant value. Radix sort does not depend on the size and type of the input elements. It performs multi-pass bucket sort. The best, average and worst case time complexity of Radix Sort is O(N*K) where N is the number of elements and K is the size of the largest element. The space complexity is O(N+K)

LSD Sort:

LSD distributes the elements into different groups – commonly known as ‘buckets’ and treated as queues (first-in-first-out data structure) – according to the value of the least significant (rightmost) digit. Then the elements are re-collected from the buckets and the process continues with the next digit[8].

MSD Sort:

MSD radix sort first distributes the elements according to their leftmost digit and then calls the algorithm recursively on each group. MSD needs only to scan distinguishing prefixes, while all digits are scanned in LSD[8].

Pros and cons of Radix Sort:

Radix sort is a stable and linear algorithm. It is not an in-place algorithm. It requires extra spaces to perform count sort. It is less adaptable. It only works for positive data.

III. ANALYSIS OF VARIOUS ENHANCEMENTS OF RADIX SORT

1. Arne Andersson and Stefan Nelson in their paper **“New Efficient Radix Sort”** implemented a new Radix sort technique “Forward Radix Sort” which has a good worst case behaviour and combines the advantages of MSD and LSD sorts.

Forward Radix Sort:

Traditional radix sort algorithms are divided into two main categories: forwards scanning algorithms - radix exchange sort, MSD radix sort and backward scanning algorithms - LSD radix sort or simple radix sort. Forward scanning algorithms split the strings into groups based on their first character and arrange them in sorted order. Backward scanning algorithms split the strings into groups based on their last character and arrange them in sorted order. Forward radix sort combines the advantages of forward and backward scanning algorithms (MSD and LSD).

Invariant: Forward radix sort works on the basis of the invariant that after the i th pass, strings are sorted according to the first i characters[10].

The sorting is performed by separating the strings into groups. Initially all strings are in the same group. This group is then splitted into smaller groups and after the i th pass all strings with the same first i characters will be in the same group. The groups are sorted according to the prefixes seen so far. Each group is associated with a number which indicates the rank in the sorted set of smallest strings in the group. A group will be finished if it contains only one string or if all the strings in the group are equal and not longer than i[10].

Results:

When there are only a few elements to sort, the single most essential optimization for the sorting algorithms covered in this work is to switch to a basic comparison-based approach. According to previous study [Bentley and McIlroy 1993], the Insertion sort is the best option. It isn't necessary to know the exact value of the breaking point. The results are satisfactory when the values are in the range of 10 to 30. This improvement cuts the running time of Forward radix sort by roughly 40%. Because the Insertion sort process consumes a significant portion of the total running time, it is critical to carefully execute this routine. This enhancement reduces the overall operating time by as much as 10% for extremely repetitive data. The enormous space overhead was one of the most difficult issues to solve in the implementation of Forward radix sort. Forward radix Sort is most suited for large alphabets, such as the Unicode 16-bit character set

2. Nasir Al-darwish in his paper **“Formulation and Analysis of in-place MSD radix sort algorithms”** used the idea of in-place partitioning to develop binary mate sort algorithm which is a recast of classical radix exchange sort. This paper presents a comprehensive, unified and readable treatment of in-place MSD radix sort.

Radix Exchange Sort/MateSort:

MSD and LSD radix sorts use extra space and their space complexity is O(N). A notable excep- tion is ‘radix exchange’ sort [12] and further generaliza- tion by

McIlroy et al. [9]. Radix exchange sort was first suggested for binary-alphabet but can be used with strings provided that bit-extraction and testing are done as low-level machine operations. The basic idea of radix exchange sort is to split in-place the data into two groups based on the most significant bit. This is done using two oppositely moving pointers; the left (right) pointer skips elements having 0-bit (1-bit); otherwise, it exchanges the elements pointed to by left and right pointers. Then the process is applied recursively to each group considering the next bit. Radix exchange sort is best thought of as a 'mating' of Radix sort and Quicksort, since in-place partitioning is a characteristic of Quicksort. Therefore, the author suggests that it be called 'Matesort'[8], the evolution from classic radix exchange sort.

Binary MateSort:

The binary Matesort algorithm has some striking resemblance to Quicksort – the difference is in the extra bit location input parameter and the partition method used[8]. Assume that any of the elements in the arrayA[1...n] are encoded with kbits(bk–1 bk–2...b0) and that A[1...n] is partitioned depending on the most significant bit (bk–1), with bk–1 = 0 appearing before bk–1 = 1. Then we have to sort each group.

Proximity map algorithm:

The 'proximity map' sort is a significant attempt to use the element value to determine the sorting address. It uses a hashing function to identify the location of the element in the final sorting order. Such algorithms, on the other hand, are typically sophisticated and are only suitable for particular types of data. Dobosiewicz [13] proposed a distributive partitioning-based sorting method which sorts real numbers by dividing them into n equal-width intervals. His algorithm has an expected time of O(n) and a worst-case time of O(n log n).

General Radix MateSort Algorithm:

The basic Matesort algorithm processes the data one bit at a time, whereas the general radix Matesort processes the data one digit (a group of bits) at a time.

Binary MateSort vs General radix MateSort:

The BitPartition method in binary matesort is replaced with the DigitPartition method, which takes two input parameters: digitloc (digit position) and 'digitval' (digit value). Second, the data must be divided into r groups, one for each radix value, for radix r and a particular digitloc. This procedure can be carried out sequentially, one digit value at a time (sequential partitioning),or using divide-and-conquer (divide-and-conquer). For sequential partitioning, we must apply DigitPartition exactly (r – 1) times to split the data into r parts, and then call GenMatesort Seq r times, one for each part, to process the data for the next digit location (digitloc – 1).

QuickSort vs MateSort:

There are two problems that affect the performance of quick sort. The first problem is Quick sort is slow when applied to small arrays especially for the arrays which are nearly sorted. The second problem is that the recursion depth and execution time increases as the repetition increases. The Quicksort algorithm can be modified easily using 'tail-recursion elimination' to limit the recursion depth[11]. Quicksort and Matesort have comparable speed, Matesort algorithms, when run for 'n' numbers with a precision of k bits, have worst-case running time of O(kn) vs O(n2) for Quicksort[8].

Conclusion:

This universal radix Matesort algorithm employs three partitioning methods: sequential, divide-and-conquer, and permutation-loop. Experiments have revealed that the generic radix Matesort with divide-and-conquer partitioning is the fastest for English text. Furthermore, the divide-and-conquer strategy can be improved to take use of data redundancy. Finally, the findings demonstrate that Matesort (and also Quicksort) are substantially faster (in many cases twice as fast) than the built-in Microsoft.Net Array Sort technique, implying that Microsoft should investigate their implementation.

3. **Arne Maus** in his paper **"ARL, a faster in-place, cache friendly sorting algorithm"** introduced a new sorting algorithm ARL which does in-place and non-stable sorting.

ARL Sort:

ARL (Adaptive Left Radix) is a modification of the usual Left Radix, which is often called MSD(most significant digit). The space and time complexity respectively are O(N + logM) and O(N*logM). It is a recursive algorithm that sorts from left to right i.e. the most significant digit is sorted first. It uses a dynamically defined radix for each pass[14].

The goal of this paper was to accomplish two things. The first is to present ARL, a new method for left radix sorting that adds two additional features. These modifications turn ARL into an in-place sorting algorithm that adapts to practically any data distribution. It has been proved that it outperforms Quicksort by a factor of two in most circumstances. ARL has the drawback of not being a stable sorting algorithm, which means that equal elements on the input may be swapped on the output. For the most part, this isn't an issue. As a result, it is suggested that ARL be used instead of Quicksort as a general-purpose sorting algorithm[14].

The impact of caching on our programs is the second focus of this study. An access to main memory that causes a cache miss now takes more than 100 times as long as an access to data is already in the nearest cache. With the introduction of 10 GHz, 64 bit processors in the next decade, this effect increased by a factor of 500 or more. Radix and ARL beat Quicksort by doing significantly fewer (but more complex) data accesses, resulting in fewer cache misses. That might be interpreted as a general message: fewer data accesses and programs that make the most of data fetched by the CPU are better to many quick passes over the same data, which usually result in a binary decision each time.

Effect of Caching:

The CPU searches in its level 1 cache first when trying to access a memory location (for data or instructions). If it isn't located there, a 'cache miss' is generated to the next level cache to see if it can be found there, and so on until it is found in main memory. When a cache miss occurs, a 32-byte cache line is normally transported from where it is located all the way up to the level 1 cache before the CPU uses that memory address. This approach works because programs have a tendency to access memory elements in a sequential order, and programs include loops[14].

For each 11 bit radix we sort on, the standard Radix algorithm performs the scanning for each pass of data – basically two sequential reads and one 'random' write of the entire array (plus one back-copying into the original array if we do an odd number of such passes). Radix similarly employs two N-dimensional arrays, therefore it encounters this cache-incompatible condition before any in-place sorting algorithm.

ARL algorithm first performs an uncached data transfer by splitting the data record into 2048 individual data records. Each such data record can then be sorted individually and fits well into a level 1 (or at least level2) cache. Therefore, the ARL only runs paths that are not suitable for such caches.

Quicksort on the other hand, is basically cache friendly in the sense that it reads sequentially (forwards and backwards) data swapping unsorted data, each time splitting data into two separate data sets that can be sorted individually. The disadvantage with Quicksort, is that it does so many of these passes. When N is large, it takes quite a number of passes before the data sets get small enough to fit into the caches.

Conclusion:

From the above analysis, it can be assumed that ARL should be more cache friendly, and hence relatively faster than Radix for large data sets.

From this paper, We can infer that both Radix and ALR outperform Quicksort, although Radix plainly loses to Quicksort in three of the ten distributions (sorted, inverse sorted, and almost sorted). In nine out of 10 distributions, ALR is superior, with the exception of the inverse sorted example, where ALR is worse for some values of N but superior for extremely high values.

Because Radix consumes twice as much RAM as ARL, we recommend using ARL as a general-purpose sorting algorithm if you can live with the fact that it is not a stable sorting method.

4. Peter M. Mcllroy and Keith Bostic in their paper **“Engineering Radix Sort”** discussed three different methods to sort strings.

When comparison is not a unit time operation, Radix sort performs asymptotically on string inputs. It is useful to use radix sort for memories which are byte addressable. There are three ways to sort strings by bytes left-to right

- Stable List sort
- Stable two array sort
- In-place American Flag

All three are twice as fast as quick sort. It is recommended to use American Flag sort for general purposes

In theory radix sort is very efficient. By creating strings into piles by their first letters. One pile will be empty. The next pile will get all those that start with A, with B and so on and so forth. But practically, it is very difficult to keep the piles in order. Bookkeeping is a major concern which has the potential to break the radix sort,

Why Quicksort needs to be re-examined: -

Quicksort performs (log n) comparisons and inspects (log n) bits per comparison. The expected running time for quick sort is (n log^2 n). Quicksort can also go quadratic when inputs are reasonable too.

Stable List Sort

The 8-bit byte is a natural radix which overcomes the slowness of radix exchange caused by bit picking. A byte radix makes for 256- or 257 -way splitting[9]. Managing space is a huge problem for so many levels of unknown sizes at each stage of recursion. An array of linked lists is a solution to this problem. Picking up the sorted piles and joining them together into a single list is a bit difficult but is possible. Empty piles cause trouble. All piles remain empty except first and second. We can clear the 255 empty piles by certain improvements

Deal into the same array and clear out the occupied piles between deals and stack those piles

Manage the stack directly. Since the number of occupied piles is unpredictable, and probably small except at the first two levels of recursion, lot of space can be saved. The piles will be stacked in first-to-last order, and they will pop off in last-to-first order. Don't try to split singleton piles. Eliminate computations which are redundant. Use pointers instead of subscripts. Avoid empty piles.

Two-Array Sort

Suppose the strings to be sorted are in the form of an array in radix exchange. The two piles are in known positions i.e. bottom and top of the array. If we know the size of the piles, we don’t need linked lists. Radix sort is very useful when using large arrays. When the pile's size is small, we can opt for shell sort. An array interface is more natural than a list interface but by using two arrays this advantage is diluted. By using O(n) working space and its storage is dynamically allocated.

American Flag Sort

It is a nice practice to rearrange an array of n integer values in ascending order in the range 0 to m-1[9]. Here m is 256 and n is arbitrary. Special cases are the partition step of quicksort (m: 2) and the Dutch national flag problem(m: 3). By comparing with the latter, we call the general problem the American Flag sort. American Flag sort differs from the two-array sort in its final phase

Conclusion:

Radix sorts have unbeatable asymptotic performance, but they present few practical problems like managing large piles and handling complex keys.

List-based radix sort is faster than pure array-based radix sorts. American flag sort is the best as an all-round algorithm for sorting strings.

REFERENCES

[1] A.D. Mishra and D. Garg. Selection of Best Sorting Algorithm. International Journal of Intelligent Processing, 2(2):363–368, July-December 2008.

[2] Donald E.Knuth. The Art of Computer Programming Second Edition, volume 3. ADDISON-WESLEY, 1998.

[3] Harkins, J., El-Ghazawi, T., El-Araby, E., & Huang, M. (2005). Performance of sorting algo- rithms on the src 6 reconfigurable computer. *IEEE International Conference on Field-Programmable Technology*, pp. 295–296.

[4] I.J. Davis, A fast radix sort, *The Computer Journal* 35(6) (1991) 636–42.R. Sedgewick, *Algorithms* (Addison-Wesley, Reading, MA, 1988).

[5] A Comparative Study of Sorting Algorithms with FPGA Acceleration by High Level Synthesis - Yomna Ben Jmaa, Rabie Ben Atitallah, David Duvivier, Maher Ben Jemaa

[6]Average_Case_and_Distributional_Analysis_of_Java_7.pdf

[7] Jmaa, Y. B., Ali, K., Duvivier, D., Jemaa, M. B., & Atitallah, R. B. (2017). An efficient hardware implementation of timsort and mergesort algorithms using high level synthesis. *IEEE International Conference on High Performance Computing & Simulation (HPCS)*, pp. 580–587.

[8] Formulation and analysis of in-place MSD radix sort algorithms by Nasir Al-Darwish

[9] P.M. McIlroy, K. Bostic and M.D. McIlroy, Engineering radix sort, *Computing Systems* 6(1) (1993) 5–27.

[10] A. Andersson and S. Nilsson, Implementing radix sort, *ACM Journal of Experimental Algorithmics* 3 (1998) article 7. Available at http://wotan.liu.edu/docis/dbl/ acmexa (accessed 18 July 2005).

[11] C.A.R. Hoare, Quicksort, *Computer Journal* 5(1) (1962) 10–16.

[12]P. Hildebrandt and H. Isbitz, Radix exchange – an internal sorting method for digital computers, *Journal of the ACM* 6(2) (1959) 156–63.

[13] W. Dobosiewicz, Sorting by distributive partition, *Infor- mation Processing Letters* 7(1) (1978) 1–6.

[14] Arne Maus. ARL, a faster in-place, cache friendly sorting algorithm. in NIK'2002, Norwegian Informatics Conf, Kongsberg, Norway, 2002 (ISBN 82-91116-45-8)