Review of Radix Sort for Comparison with other Sorting Methods.

Harshal Jaiswal*, Mohit Negi*, Sahil Sonawane*

*Northeastern University, College of Engineering, Program Structure and Algorithms (INFO 6205)

Abstract: - We have studied and evaluated Radixsort by benchmarking it with other sorting algorithms. We focus on radixsort and its variants, as well as their benefits and drawbacks. According to our benchmarking results, MSD radix sort outperforms any comparison-based sorting algorithm. We conduct a survey on in-place MSD radix sort, LSD radix sort, and forward radix sort. We compare MSD radix sort to previous radix sort work in a brief review.

Introduction: -

Sorting is a computational building block of fundamental importance and is the most widely studied algorithmic problem. An important observation is that in order to efficiently implement a string sorting algorithm, we should not move the strings themselves, but only pointers to them. As a result, each string movement is guaranteed to take the same amount of time. This is not a major limitation. In fact, it is common to store the characters of each string in consecutive memory locations and represent each string by a pointer to the string's first character. The string's length can be explicitly stored, or the end of the string can be indicated by a specially designated end-of-string character. For small subproblems, switching to a simple comparison-based method (typically Insertion sort) is a well-known technique for improving the speed of sorting algorithms. This technique proves to be very important in practice, and it is used in all the algorithms discussed. MSD radix sort with 8-bit characters the character's minimum and maximum values are recorded. As a unit, the algorithm moves sub lists of elements with a common character. To keep track of the flow of computation, an explicit stack is used. This way, we only need one bucket array. The algorithm switches to Insertion sort for small groups.

I. Previous Research about Radix Sort: -

LSD Sorting by Radix The second class of radix-sorting methods examines the digits in the keys from right to left, beginning with the least significant digits. Radix sort works on the radix of elements, and the number of passes is determined by the maximum length of elements observed.

• Radix Sort is extremely efficient for data sets of uniform length.

• For data sets with unequal length elements, the number of passes increases as the maximum length of elements in the list increases, increasing Space & Time Complexity.

• In the case of strings, the strings are sorted, but the data is corrupted.

As a result, we cannot expect optimal performance from this algorithm. Another, perhaps more natural, option is to begin scanning the input with the most significant digit. MSD radixsort is the name given to this algorithm. The algorithm functions as follows. Divide the strings into groups based on their first character and arrange them in ascending order. Recursively run the algorithm on each group separately, ignoring the first character of each string. Groups with only one string require no further processing. This algorithm has the advantage of inspecting only the distinguish prefixes, which is the smallest number of characters that must be inspected to ensure that the strings are, in fact, sorted.

II. Forward Radix Sort: -

In this section, we present Forward radixsort, a simple MSD radixsort algorithm that solves the problem. The algorithm combines the benefits of the LSD and MSD radixsort algorithms. LSD radixsort's main strength is that it inspects a complete horizontal strip at once; its main weakness is that it inspects all characters in the input. MSD radixsort only inspects the strings' distinguishing prefixes and does not make efficient use of the buckets. Forward radixsort begins with the most significant digit, buckets only once per horizontal strip, and inspects only the significant characters.

Forward radix sort has good worst-case behavior. Experiment results show that radix sorting is significantly faster (often more than twice as fast) than comparison-based sorting. It is possible to implement a radix sort that has a good worst-case running time without sacrificing average-case performance. The implementations compete with the best previously published string sorting programs.

III. MSD Radix Sort In Place: -

Increasing amount of data available, memory demands have been increased, thus an in-place version of radix sort was developed. We can no longer rely on a separate buckets data structure to reduce the amount of extra memory required from O(n) to O(1). Rather, we must swap elements in place. We modify our MSD radix sort algorithm to successfully permute the elements into their correct positions. Rather than explicitly bucketizing each element, we create a histogram that counts the number of elements that belong to each bucket – this will require only one pass through the data. We can use parallelization once more by dividing the counting between processors. Assuming p processors, each process will only have to count O n p elements; at the end, we simply add the corresponding counts to get the number of elements in the data that belong to each bucket of three (which can also be done in parallel via parallel summation)

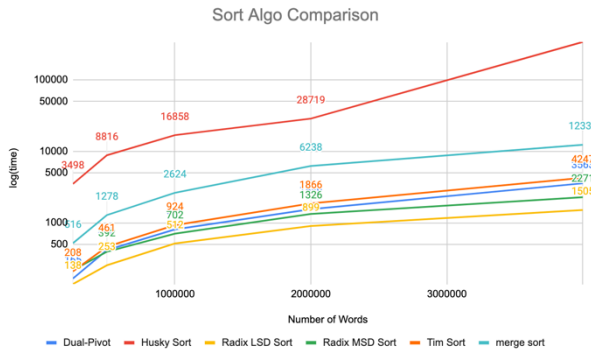
We want to use head and tail pointers to represent the beginning and end of each bucket, and then iterate through the data, swapping elements into their correct bucket as we go. The head and tail pointers will keep track of which elements have already been swapped into the appropriate bucket and will increment/decrement accordingly. Finally, once all elements have been swapped to their proper buckets, we can resume sorting on the next digit as before Because the recurrence for work remains constant, it is easy to verify that the work is still O(n log n). Furthermore, it is clear that this algorithm requires only a constant amount of 4 memory. However, depth has returned to O(n) because the permutation process is sequential, requiring total O(n) work and depth. Radix sort and its in-place variant and many parallelization ideas that can be applied to radix sort to improve its runtime efficiency. However, due to the inherently sequential nature of standard in-place radix sorting, combining these two ideas of parallelization and in-place sort proves difficult. Unfortunately, the LSD radix sort lends itself to a sequential nature. MSD radix sort can be used to introduce some parallelization. Instead of beginning with the least significant digit, we begin with the most significant digit. Using buckets to sort and recursively sorting the buckets Because buckets are independent of one another, recursive calls can be parallelized.

Comparison of non-Comparison based Sorting Algorithms:-

Name Case n<	Average Case	Worst Case	n<<2K	Advantage/disadvantage
Bucket Sort	O(n.k)	O(n2.k)	NO	1. Stable & fast. 2. Used in special cases when the key can be used to Calculate the address of Buckets
Counting Sort	O(n+2k)	O(n+2k)	Yes	1. Stable, used for repeated Value & often used as a subroutine in radix sort. 2. Valid for integer only
Radix Sort	O(n.ks)	O(n.ks)	NO	1. Stable, straight forward 2. Applicable to data set with multiple fields.
MSD Radix Sort	O(n.ks)	O(n.ks)	NO	1. Highly efficient for sorting large data sets. 2. Bad worst-case performance due to data fragmentation.

LSD Radix Sort	O(n.ks)	O(n.ks.2s)	NO	1. Stable & fast sorting method.

Observation:-



When plotting graph for 800 runs for 25K, 50K, 1M, 2M & 4M words, it can be observed that Radix LSD was the fastest followed by the Radix MSD after that Dual-Pivot, Tim Sort, Merge Sort and Husky Sort.

Justification: Radix LSD came out to be fastest as we restrict the string length to three which is the length of the shortest name in the array list. This was done because of the restrictions post by the LSD sorting algoritms.

Conclusion: -

Radix sort is a fast non-comparative integer sorting algorithm that can also be applied to floating point numbers and character strings. It can achieve a theoretical linear runtime complexity; additionally, in-place radix sort can do this while only using constant memory. Its appeal stems from its simplicity as well as its theoretical worst-case linear runtime complexity of O(kn), where k represents the maximum number of digits for any given number in the data.

As a result, radix sort can outperform any comparison-based sorting method. Unfortunately, in practice, the constant k factor is frequently non-negligible and can frequently exceed log n. As a result, in many applications, standard radix sort falls short. The appeal of radix sorts stems from their simplicity as well as their theoretical worst-case linear runtime complexity of O (kn), where k is a constant representing the maximum number of digits in the data for any given number. As a result, radix sort can outperform any comparison-based sorting method. Unfortunately, in practice, the constant k factor is frequently non-negligible and can frequently exceed log n.

References:

[1] Avinash Shukla, 'Review of Radix Sort & Proposed Modified Radix Sort for Heterogeneous Data Set in Distributed Computing Environment', (IJERA) ISSN: 2248-9622 www.ijera.com Vol. 2, Issue 5, September- October 2012, pp.555-560

[2] Minsik Cho, 'PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort', IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

[3] Nasir Al-Darwish, 'Formulation and analysis of in-place MSD radix sort algorithms', Journal of Information Science 2005; 31; 467, DOI: 10.1177/0165551505057007

[4] Arne Andersson, ' A new Efficient Radix Sort', DOI: 10.1109/SFCS.1994.365721 · Source: IEEE Xplore

Authors:

Harshal Jaiswal, *Graduate Student, Northeastern University (MS in Information Systems),* *Jaiswal.ha@northeastern.edu*

Mohit Negi, *Graduate Student, Northeastern University (MS in Information Systems),* *negi.m@northeastern.edu*

Sahil Sonawane, *Graduate Student, Northeastern University (MS in Information Systems),* *sonawane.sa@northeastern.edu*