Multilingual Radix Sort

1st Nishay Madhani
College of Engineering
Northeastern University
madhani.n@northeastern.edu

2nd Aravind Polepeddi
College of Engineering
Northeastern University
polepeddi.a@northeastern.edu

3rd Sai Poojitha Konduparti
College of Engineering
Northeastern University
konduparti.s@northeastern.edu

***Abstract*—Utilization of comparison based sorting algorithms has been predominant for ordering elements ever since the start of the century. Ease of programming can be attributed to its prevalence during this time. However, these algorithms only focus on minimizing the number of comparisons or swaps made. For divide and conquer or heap based algorithms we theoretically cannot do better than a O(N log N) time complexity this leads us to find alternative solutions as presented in this paper.Radix sorting algorithms have been eclipsed by Quicksort and Timsort even though it has a superior run time complexity of O(N) ie. linear time. Another disadvantage when using radix sort is the lack of order available when assigning buckets for multilingual schemes. We present an algorithm that uses the Unicode Collation algorithm to generate keys which are in tern used for sorting a given array. In the paper we also discuss some of the improvements made over the years on the algorithm itself.**

***Index Terms*—Sorting, Quicksort, Timsort, Insertion sort, Complexity, Comparison Sorting, Radix Sort, LSD Radix Sort, MSD Radix sort.**

I. Introduction

Sorting algorithms can be often be divided into comparison based and non-comparison based algorithms. Quick-sort, Timsort or any divide and conquer based algorithms all use comparison to be able to sort an array. Radix sort is another approach to sorting data which uses key-indexed counting to sort the elements. Radix sort is efficient algorithm that has been misrepresented by its misconceptions. It is often believed what radix sort compares all characters in order to sort the array, however that has been refuted by several sources [2]. Radix sort when implemented correctly can be faster than strictly comparison-based sorting for average data.

Radix sort algorithms divides a particular element into units that can be compared independently to maintain the sort order. For Strings this would mean comparing characters, for integers it would mean digits and for bytes it would be binary. A natural question that arises then is what would the the order for successive character iteration. In this light, we can have two different algorithms LSD or Least Significant Digit radix sort which sorts characters from right to left and MSD or Most Significant Digit radix sort iterating from right to left. A common analysis for LSD would be that it starts from the digit which has the minimal importance and therefore wold be an inefficient algorithm. For those reasons, we do not discuss it going forward.

The Algorithm is as follows:

- Split the strings into groups according to their First character and arrange the groups in ascending order.
- Apply the algorithm recursively on each group separately, disregarding the First character of

In the upcoming sections we discuss several papers that proved influential to the implementation of MSD radix sort and our approach to using it for the Unicode encoding scheme. Finally we compare our method with comparison based algorithms to prove its efficiency.

II. Literature Survey

In the subliminal paper by Davis [1] an implementation of radix sort is presented which can sort fixed length keys of any size much faster than comparison-based algorithms. Their implementation details the utilization of the key for creation of p partitions as the key is processed from left to right. As this process is recursively done for each successive byte within the key, the list may loose its stability. For the implementation of the algorithm itself C code has been provided which uses a static map of 256 entries to represent a count for each partition. This count is initialized to 0 for each recursive call made to partition the keys starting from the most to the least significant byte. At each step once the size of the partition has been determined a linked list is created to represent the partition. Each node within the linked will point to the next node within the partition. At the end of the algorithm a successive scan is done to the keys to use the partition to sort the keys in-place for the sort order. As an optimization to the algorithm for small sub-arrays i.e size of sub-array is less than or equal to 16 insertion sort has been used.

The paper continues to argue about the validity of the hypothesis it offers. Based on strictly mathematical analysis, time complexity for the algorithm is to be $O(N * M)$ where N is number of keys and M is the average size of each key. To assess this claim a experimental setup is concocted between the best known implementation of Quicksort and the implementation provided. As a control for the algorithm Qsort a variant of the Quicksort algorithm provided by ATT is also included. In all experiments and tests except five, radix sort was the clear winner. In terms of practical usage of the algorithm, it cannot be used for multi-key use cases due to its in-stable nature and would not be useful when there is constraint on number of exchanges or memory available.

Another work published by Andersson & Nilsson [2], proposes two implementations of radix sort and compares its efficacy

in terms of time complexity and applicabilty against Quicksor and other algorithms. As the paper notes LSD Radix sort which sorts keys from left to right has major downfalls: inspection of largest empty buckets for variable length keys and comparison of least significant bytes first. To overcome these issues, the paper presents MSD Radix sort as an alternative. When it comes to any radix based sorting algorithm choosing the length of the alphabet i.e radix is a complicated decision. A large radix will reduce total number of passes but will increase the amount of memory required and the buckets that need to be inspected. However, with an optimization involing storing the minimum and the maximum character encountered and only looking at buckets between those values will offset the time taken.
Adaptive Radix sort involving the use of distributive partitioning is presented. The algorithm sorts N real numbers by distributing them into N intervals of equal width. The process is repeated recursively for each interval containing more than 1 element. To optimize the size of the alphabet, the size is chose depending upon the number of elements remaining.For the purposes of the paper, the author has implemented the adaptive radix sort with 8-bit and 16-bit alphabet sizes. Another implementation provided is Forward Radix sort which combines the strengths of LSD and MSD radix sort. LSD radix sort inspects a complete horizontal strip at a time and MSD radix sort does not use partitions very well. Forward radix sort starts from the most significant digit and performs bucketing only for each horizontal strip. An important invariant for the algorithm is that after the ith pass the first i chracters are sorted. Using this property of MSD Radix sort, we can group strings into buckets and distiguish between the buckets as finisehd or finished if it cotains only one string or if all the strings in the group are equal and not longer than i. Performance of both algorithms was analyzed and radix sort came out on top each time. Adaptive sort was faster than Forward sort, but Forward sort was found to be more suited for large alphabets such as Unicode.
In [3] Mcllroy and Bostic offer three different implementations of radix sort and draw comparisons between them. Their first implementation is called a stable list sort which creates linked lists for each partition and sorts them recrusively. Second, a two arrays are used to in order to provide a stable implementation of radix sort. This implementation uses a count for storing the length of each parition and thus reduces the need of a linked-list. This implemetaion is similar to the most implementation of MSD Radix sort. The last implementation provided tries to solve the dutch national flag problem [5]. It is quite similar to the one presented by Bentley [4].

Finally, the authors provide the results of their benchmarks and prove that for all implementations of Radix sort it is quite faster than comparison based sorting methods for string arrays. From the three algorithms provided two-array implementation was the most efficient.

Betley et al [4] provide an algorithm which combines radix sort with quicksort to provide a three-way quicksort. In the paper they an present efficient algorithms for searching and sorting strings that are faster than previously known algorithms. For the purposes of this paper we focus on their use of radix sort and quicksort to solve the ”Dutch National Flag” partitioning problem as popularized by Dijkstra [5]. The authors make a key association about quicksort algorithm and binary search trees. In both cases for each element, all elements to the left of it is less than it and all elements to the right of the are greater than it. This association can be extend with radix sort algorithm and the tries data structure since they both involve creating ”branches” when the current node is equal for two elements. Ternary search trees wherein an extra middle node is added for all elements equal to the current one can be extended with radix sort to improve quicksort. The results declared in the paper were quite commenable with an efficieny of O(N log N) for average input and O(N^2) for worst-cases. However, this algorithm does not provide any extra benefits when compared with Timsort as it is not stable.

Approach

One common shortcoming on Radix sort has always been it use or extension to other languages apart from English. This is due to the fact that when characters were introduced in the early ages stages of many programming languages they used ASCII or Extended ASCII for denoting them. This lead to implementations of Radix sort for the English alphabet very simple due to the type casting available in many languages. Another reason for this downfall is the lack of availability of linguistic orders for many languages. Although, we do have the Unicode encoding scheme [6] for multiple languages. The actual order within the scheme does not correspond to native ”orders” used for that language. Unicode would later go on to release the Unicode collation algorithm [7] which is designed for these cases. Rules were created for different orders and places within the CLDR.

Our approach is a modification of the two array approach discussed by Mcllroy and Bostic [3]. For the purposes of using the collation rules provided by unicode we used open-source libraries provided by IBM to generate collation keys for each specific input. These collation keys can then be used to create byte arrays which can be sorted to result in a sort order. For small sub-arrays we have use insertion sort as a fallback and the cutoff value has been kept constant at 128 elements. Flowchart depicted in Fig 5.

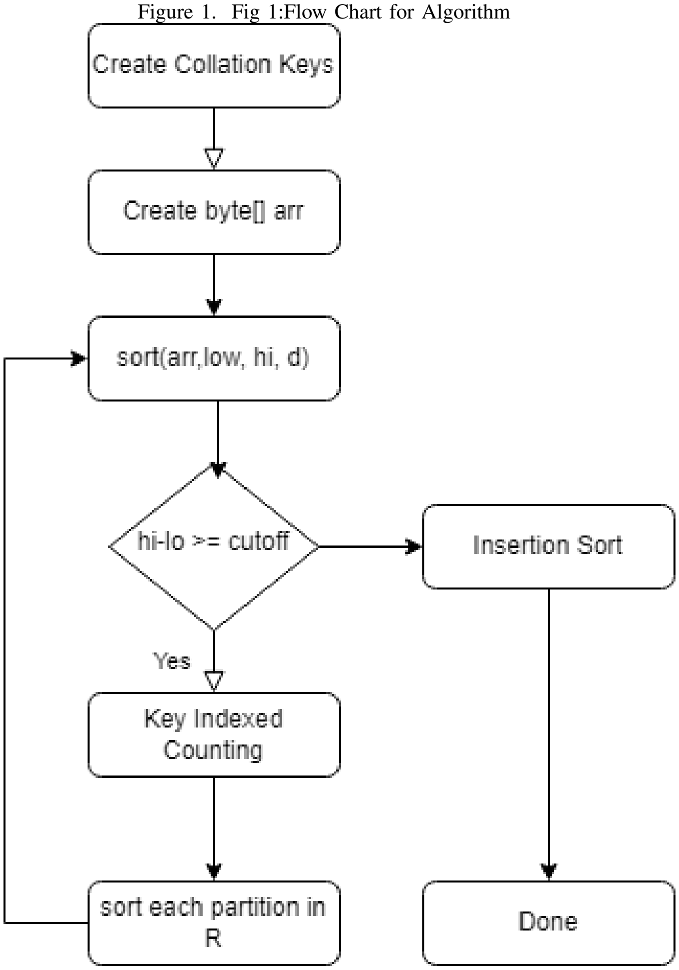
A. *Improvements*

One simple improvement could be made in the use of multiple bytes rather than a single byte for each digit. This would obviously increase the size of r but such a increase is well within its rights when comparing the time taken to sort the algorithm.

B. *Scope*

Unicode currently supports 159 languages all of which can be sorted using this algorithm as long as an adequate sort order is available in the UCA [7].

Figure 1. Fig 1:Flow Chart for Algorithm



C. Time Complexity

Based on analysis performed in ”Engineering of Radix Sort” [3], we can conclude that this algorithm takes $O(N*M)$ where N is the size of the data and M is average size for the byte representation of collation keys.

III. Benchmarking and Testing

A. Testing

Unit Testing for 8 bit MSD Radix Sort, 16 Bit MSD Radix Sort and LSD Radix Sort has been provided within in the code repository. For the purposes of Quicksort, HuskySort and TimSort basic unit tests have been provided to prove its accuracy. Our Algorithm is shown to be able to sort any language within the unicode specification.

B. Benchmarking

Radix sort algorithms are compared again Timsort, LSD Radix Sort, Dual Pivot Quicksort. Timsort implementation provided JDK is used, Dual Pivot Quicksort implementation is borrowed from INFo6205 class repositort. All Radix sort implementations are adapter from algs4 libarary provided by Sedgewick [8]. These algorithms are adapted to be able to sort byte arrays.

Results

Figure 2. MSD Radix Sort graph

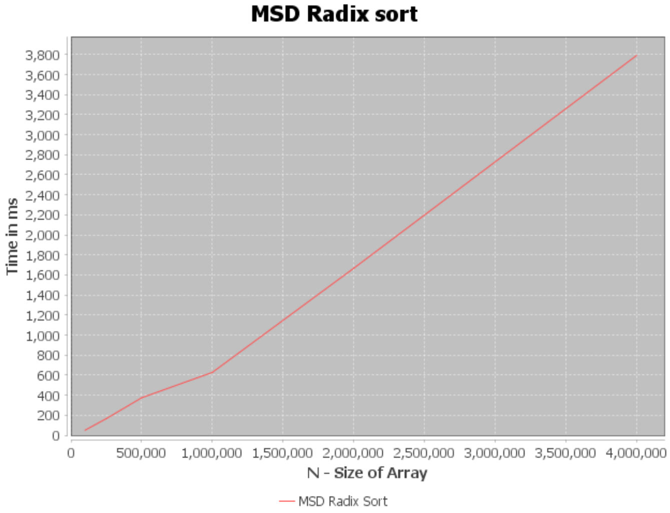


Figure 3. Modified 16 bit MSD Radix Sort

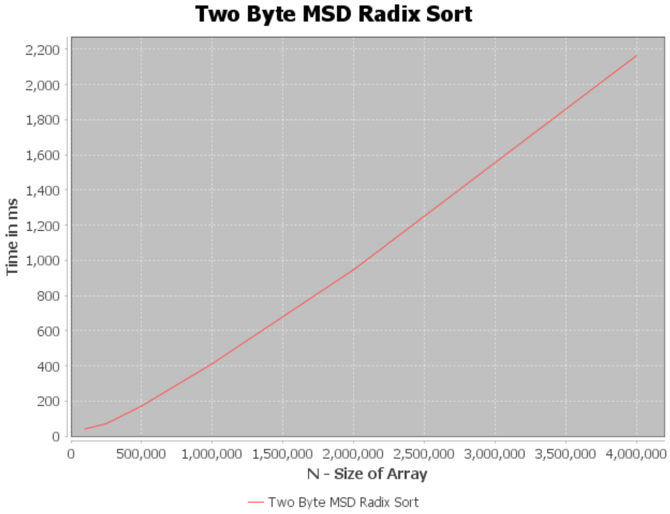


Figure 4. All Sorting Algorithms

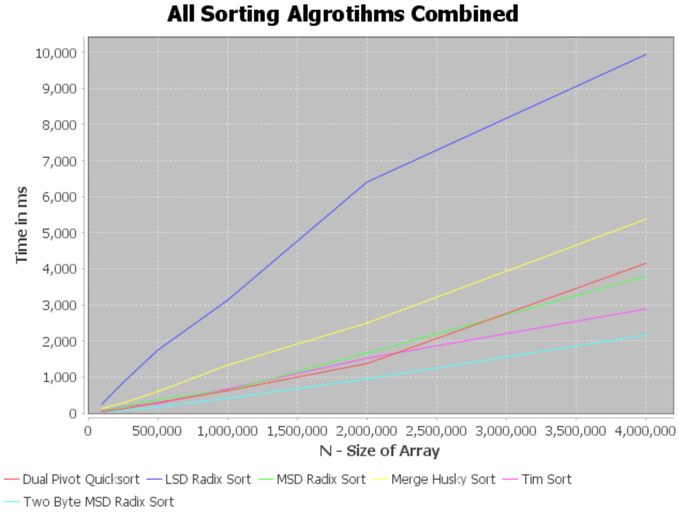
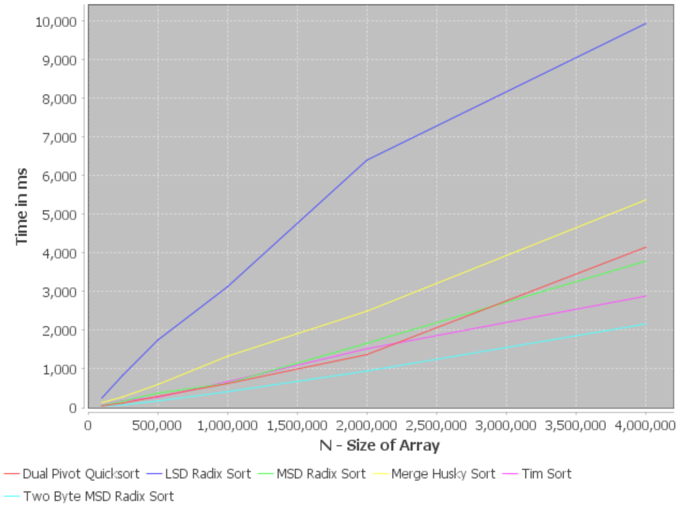


Figure 5. Log Plot for comparison



	250k	500K	1M	2M	4M
16 Bit MSD Radixsort	70.64	171.0	411.92	947.0	2163.92
8 Bit MSD Radixsort	166.0	373.96	627.32	1666.75	3790.48
Timsort	178.84	266.0	669.8	1522.36	2885.36
DualPivot QUicksort	115.76	290.08	621.88	1372.68	4151.68
Merge Huskysort	278.2	594.36	1329.36	2494.92	5374.04
LSD RadixSort	837.68	1743.48	3125.12	6402.72	9938.96

A note on pre-processing of strings, once the strings are created a collator is created for the particular locate. Each sorting algorithm is provided with the collation keys created from this collator.

From the data presented above and it can be inferred that 16bit MSD Radix Sort is more efficient that all algorithms and 8bit MSD Radix Sort is on par with comparison based algorithms such as dual pivot quicksort and timsort. LSD Radix sort is slowest algorithm followed by Merge HuskySort.

Conclusion

In conclusion, we have presented a sorting algorithm for sorting strings specified by the Unicode Collation Algorithm. Our implementation of radixsort is faster than comparison based algorithms and maintains its claim of being a $O(N*M)$ algorithm.

References

[1] Davis, I. J. (1992). A fast radix sort. The computer journal, 35(6), 636-642.

[2] Andersson, A., Nilsson, S. (1998). Implementing radixsort. Journal of Experimental Algorithmics (JEA), 3, 7-es.

[3] McIlroy, P. M., Bostic, K., McIlroy, M. D. (1993). Engineering radix sort. Computing systems, 6(1), 5-27.

[4] Bentley, J. L., Sedgewick, R. (1997, January). Fast algorithms for sorting and searching strings. In Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms (pp. 360-369).

[5] Dijkstra, E. W. (1976). A discipline of programming (Vol. 613924118). Englewood Cliffs: prentice-hall.

[6] Allen, J. D., Anderson, D., Becker, J., Cook, R., Davis, M., Edberg, P., ... Whistler, K. (2012). The unicode standard. Mountain view, CA.

[7] Davis, M., Whistler, K., Scherer, M. (2001). Unicode collation algorithm. Unicode technical standard, 10, 95170-0519.

[8] Sedgewick, R., Wayne, K. (2011). Algorithms. Addison-wesley professional.