# Literature Survey on Sorting words in Natural Language

## Abstract

This literature survey aimed to evaluate the performance of various sorting algorithms used for natural language. Through implementations of Huskysort, Timsort, Dual-pivot quicksort, MSD radix sort, and three-way radix quicksort, we sorted data types of any type including strings. To conclude the above sorting implementations, we benchmarked them with 1 million Chinese words according to pinyin. We also benchmarked them with other natural languages such as Telugu and English. Several unit tests were added to cover each implementation of the algorithm. Huskysort has been observed to outperform all other algorithms when sorting a million words. We had gone through a few relevant technical papers on the subject of sorting.

## Algorithm Overview

This section discusses the work of two technical papers related to Huskysort and Radixsort.

**The first paper** that we analyzed is '**Huskysort'** by R C HILLYARD, YUNLU LIAOZHENG, and SAI VINEETH K R.

In the research paper, the author mentioned the impact of array accesses on the performance of sorting algorithms. It states that an algorithm that moves work from the linearithmic phase to the linear phase may be able to reduce the total number of array accesses and, indirectly, processing time.

This paper describes an approach to sorting which reduces the number of expensive comparisons in the linearithmic phase as much as possible by substituting inexpensive comparisons. It does that by comparing the objects using primitive values in the form of their hashcode.

The ideal use case of Huskysort is unordered data consisting of objects which are relatively expensive to compare.

Huskysort Flow -

- Encode the elements (Objects) into long
- Sort the array of longs using Introsort such a way that when elements $i, j$ of longs are exchanged, then elements $i, j$ of $xs$ are also exchanged
- After this step, if there are any inversions, Timsort is performed on the array of objects to make sure that their elements are truly sorted

The paper also draws the analogy between the Huskysort and Shellsort. Shellsort single swap fixes h-inversions (where we are h-sorting the array). So, its primary mechanism is to try to fix as many inversions as possible before taking the subsequent step. In the same way, Huskysort fixes as many inversions as possible before doing the final step.

The paper describes how compare is a costly operation compared to swapping in the case of objects. The Huskysort performs two swap operations - one for longs(hashcode) and the other for object references. But swapping and comparison of longs are much faster compared to object comparisons.

Given the importance of Hashing in the algorithm, this should be Hashsort, but the name is already taken for a different algorithm.

HuskySort Space Complexity is O(N) as it requires an extra array for storing Hashcode values. Huskysort is not stable unless using the merge sort variation.

Array accesses for Tim/merge sort: 11.5 N ln N + 2 N

Array accesses for Husky sort: 6.4 N ln N + 9 N

As N increases, we expect the performance of HuskySort to get better compared to Mergesort. For the final phase, both Timsort and Insertionsort work well till N = 50000, after that Timsort wins.

The paper also Benchmarked the relative sort times for Huskysort, dual-pivot quicksort, system sort. It also compares Husky with Timsort and Husky with Insertionsort. From the analysis, it seems that HuskySort is the winner among other algorithms. Also, Husky with Timsort performs much better for input size above 1 million.

The author also describes the best use cases and non-use-cases for Huskysort. HuskySort works very well for sorting randomly ordered arrays of objects on the JVM. HuskySort does not compete well with Timsort when an array is partially sorted. HuskySort does not compete well with Dual Pivot Quicksort when sorting primitives.

The author concludes HuskySort in the end. It concludes that, when sorting objects rather than primitives, Huskysort is always faster than dual-pivot quicksort. It is faster than Timsort for arrays that are not partially ordered. It is especially fast for Unicode character strings.

**The second paper** that we analyzed is **'Engineering Radix Sort'** by Peter M. Mcllroy and Keith Bostic. This paper compares the performance and characteristics of MSD Radix sort with other general-purpose sorting algorithms.

According to the paper, radix sort is the best way to sort strings based on theory. It also explains the working principle which is as follows - Take your strings and separate them by their first letter. One pile gets the empty strings. The next pile gets the strings that begin with *A*-. The next pile gets the strings that begin with *B*- and so on. In the case of two or more recursive splits, continue splitting the piles until the strings run out. When no more piles remain, collect them all in order. The strings are now sorted. For classical radix exchange, assume that all strings have the same length. As a result, there would be no pile for empty strings, and splitting could be made as with Quicksort, with a bit test instead of Quicksort's comparison to determine which pile the string belongs to.

In the next part, the author compares the MSD and LSD radix sort. It is possible to sort keys in lexicographic order using MSD radix sort. Unlike LSD radix sort, MSD radix sort may not preserve the order of duplicate keys. Unlike LSD, MSD radix sorts process keys from the most significant digit to the least significant digit. When MSD radix sort reaches the unique prefix of the key it stops rearranging the keys.

It also highlights disadvantages of MSD radix sort such as –

- Extra space for aux []
- Extra space for count []
- The inner loop has a lot of instructions
- Accesses memory "randomly" (cache inefficient)

The authors also compare Radixsort and Quicksort. Radix exchange was briefly popular, but Quicksort quickly surpassed it. Not only was radix exchange slower than Quicksort, but its bits were hidden by Fortran or Algol code, so Quicksort was easy to use. Despite this, the Radix exchange is impossible to beat in terms of data examined. Quicksort cannot come close. Quicksort compares Θ(log n) bits per comparison and inspects Ω (log n) bits per comparison. By this measure, the expected running time for Quicksort is Ω (n log^2 n), for radix exchange only Ω (n log n). Worse, Quicksort can "go quadratic" and take time Ω (n log^2 n) on unfortunate input such as date, timestamps.

Major differences between Quicksort and Radixsort –

- Rather than comparing bits, Quicksort can use machine instructions
- Adaptive splitting is the key to achieving minimal expected times in quicksort or radix exchange. Since Quicksort picks splitting values from the data, it should result in roughly 50-50 splits, regardless of skewed data
- Because radix sort relies on data on a finite alphabet, it requires the data to be made to fit the routine rather than vice versa. Quicksort can sort anything. Change the comparison routine, and it is ready to sort anything

The two sorting operations are similar in many other ways. They both sort in place, using little extra space. Both require a recursion stack, which is expected to grow to O (log N). If the strings are long or have varying lengths, either method is helpful, but if the strings are long or have different sizes, it is better to address the strings through uniform descriptors and sort by arranging minor descriptors instead of long ones.

Finally, the paper compares 3-way radix quicksort and standard quicksort

standard Quicksort -

- uses 2N ln N string comparisons on average
- uses costly compares for long keys that differ only at the end, which is a typical case!

3-way radix quicksort -
- avoids re-comparing initial parts of the string
- adapts to data uses just “enough” characters to resolve order
- uses 2 N ln N character comparisons on average for random strings
- is sub-linear when strings are long

It also compares MSD radix sort and 3-way radix quicksort

MSD radix sort -
- has an extended inner loop
- is cache-inefficient
- repeatedly initializes counters for long stretches of equal chars, and this is a typical case

3-way radix quicksort
- uses one compare for equal chars
- is cache-friendly
- adapts to data uses just “enough” characters to resolve order

## Conclusion

We benchmarked the above sorting algorithms for randomized input of 1 million Chinese words. We sorted them as per the pinyin order. The results are as follows -

| **Sort Algorithm** | **Time (in sec)** |
|---|---|
| Huskysort | 0.7 |
| Timsort | 5 |
| Dual Pivot Quicksort | 7 |
| LSD | 21 |
| MSD | 71 |
| Three-way Radix Sort | 11 |

It shows that Huskysort is the best sorting algorithm to sort strings in random order.

## References

https://arxiv.org/pdf/2012.00866.pdf

https://www.cs.princeton.edu/~rs/AlgsDS07/18RadixSort.pdf