Final Team Project - Implementation of MSD Radix Sort for Hindi

James Shah [Section – 6] (002107975)

Gaurav Thorat [Section – 6] (002957343)

Akash Bhimani [Section – 6] (002199939)

Task: Implement MSD radix sort for a natural language that uses Unicode characters. You may choose your own language or (Simplified) Chinese. Additionally, you will complete a literature survey of relevant papers and you will compare your method with Timsort, Dual-pivot Quicksort, Huskysort, and LSD radix sort.

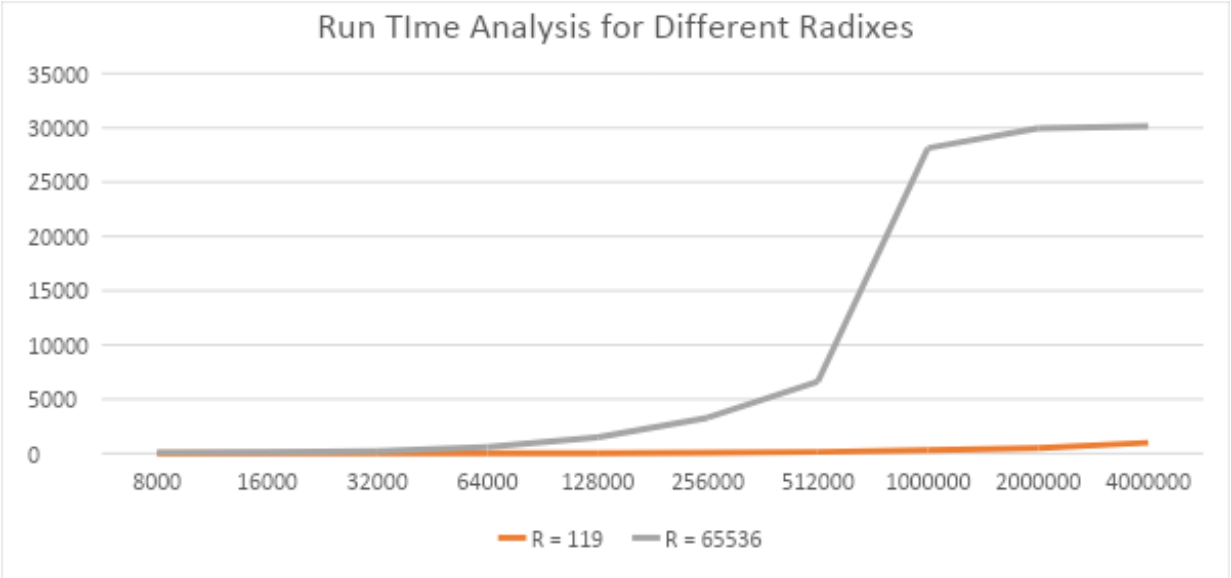
Proposed Solution: Some great optimizations to this MSD Radix algorithm can be found on the internet. Some include maintaining an invariant after I passed, others have suggested that MSD Radix sort can be effectively optimized by parallelizing it, while some optimization proposes optimization via Cutoffs. In our approach, we have tried to optimize this MSD Radix on natural languages via using optimal bucket sizes or radix and cutoff.

In our solution, we found that by figuring out the appropriate radix size needed for a natural language we can reduce the time and space complexities for MSD Radix Sort. Our solution considers the range of characters for a given language that needs to be sorted before running the MSD Radix Sort Algorithm. Hindi is written in the Devanagari script, and we have used Hindi words to test our algorithm. The range of Hindi letters in Unicode Encoding is 119. This range is critical for the optimization of MSD String sort. So, we kept the radix or base as 119 rather than 65536 for Unicode values. We then map these specific Unicode values to the corresponding bucket for the grouping of specific prefixes.

This saves us a lot of unnecessary recursion steps, that is for characters that are not even present in a specific language. As of now, we do not have to visit a large number of empty buckets.

Following are our findings,

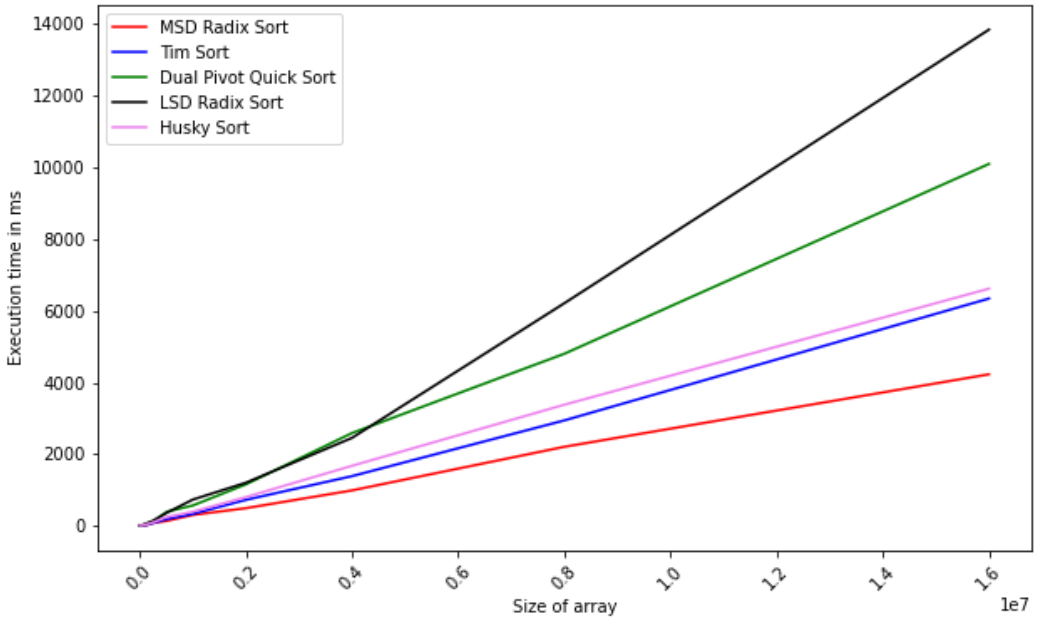
Size of Array	R = 119	R = 65536
8000	6.5	96.1
16000	10.13	119.6
32000	12.8	224.7
64000	14.1	597.8
128000	23.9	1481
256000	83.6	3291
512000	141.3	6601.8
1000000	307.6	28141.3
2000000	496	29945.7
4000000	991.9	30131.7



As we can see in the above figure, manipulating the number of radices can have a huge impact on the running time of the algorithm.

When we compared our algorithm, to other algorithms such as LSD Radix Sort, Husky Sort, Tim Sort, Dual Pivot Quick Sort we found that our proposed optimization to MSD Radix Sort is faster than all the other aforementioned sorts. Comparisons between them are as follows,

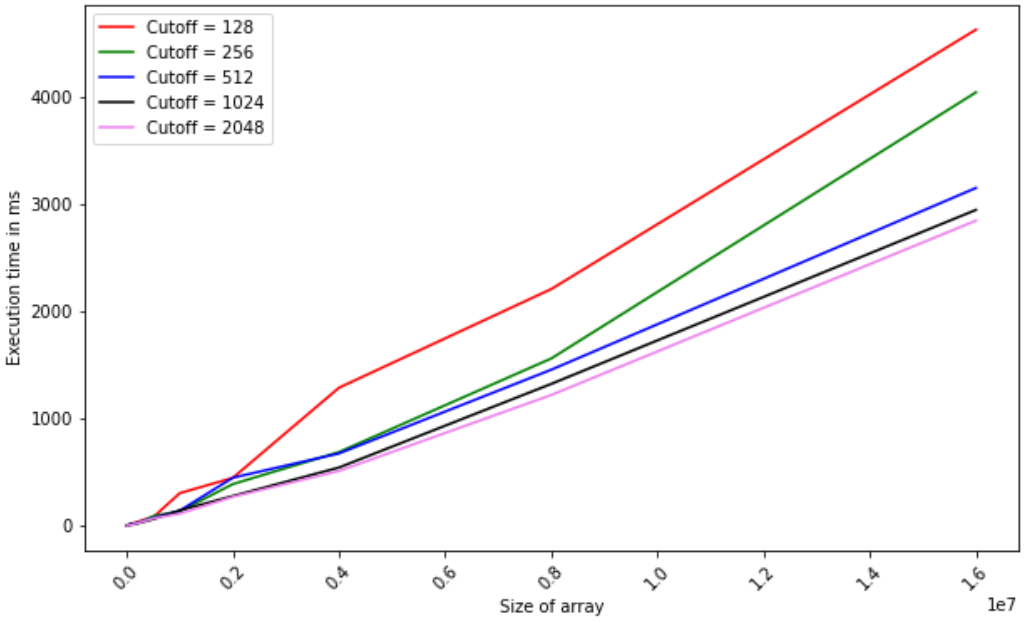
Size of Array	MSD Radix Sort (ms)	Tim Sort (ms)	Dual Pivot Quick Sort (ms)	LSD Radix Sort (ms)	Husky Sort (ms)
8000	6.5	4.6	5	10.4	14.7
16000	10.13	6	7	12.6	8.3
32000	12.8	8.4	14	16.2	13.5
64000	11.1	13.4	27	34	20.6
128000	23.9	28.5	57	69.7	44.5
256000	83.6	84.5	127	148.3	103
512000	141.3	190.7	398	368.2	250.6
1000000	307.6	334.4	571	735.4	398.4
2000000	496	727.9	1160	1205.9	804.6
4000000	991.9	1390.1	2592	2456.8	1678.5
8000000	2208.7	2944.9	4804	6213.5	3382
16000000	4230.6	6344.2	10100	13846.7	6620.1



Further optimization is done to our proposed solution is done via introducing cutoff values. We found out that when we cut off the MSD Radix Sort to Insertion sort, our algorithm performed better. As the performance of MSD Radix sort is bad for small-sized arrays, as there is a huge number of small subarrays created because of recursion.

Observations are as follows,

Size of Array	Cutoff = 128	Cutoff = 256	Cutoff = 512	Cutoff = 1024	Cutoff = 2048
8000	4.9	4	3.3	4.1	4.2
16000	8.3	2.2	3.2	3	2.9
32000	13.5	10.7	8.1	4.7	7.5
64000	13.5	11.1	16.9	15.4	6.3
128000	26.1	14.7	21.9	16.9	15.7
256000	47.3	35.6	36.4	31	34.3
512000	87.5	95	76.9	65.4	72.5
1000000	307.2	136.9	143.1	146.3	114.8
2000000	450	391.8	451.5	280.5	272.5
4000000	1292.3	691.1	678.3	546.7	514.3
8000000	2214.4	1565.9	1460.8	1328.4	1224.5
16000000	4638.4	4052.4	3159.1	2954	2853.9



Conclusions: In conclusion, we can write that the size of radix for Unicode makes a huge difference, for such a large number of radices, the time taken by the algorithm explodes, thus it should be used with a lower number of radices. The introduction of cutoff also greatly affects the performance of the algorithm. The cut-off size for using insertion sort in MSD makes a very big difference when considering the big number of elements in the array to be sorted.

Notes:

1) We have used the Benchmark_Timer method to benchmark MSD, LSD, Tim, Husky Sort.
2) For Dual Pivot Quick Sort, System.currentMilliseconds function was used to benchmark our results.
3) We have not written unit tests for Husky Sort as it was used directly from the repository