MSD Radix sort for Chinese characters

Namrata Ruchandani, Varun Venkatesh Gowda, and
Venkteshprasad Maya Rao

College of Engineering, Northeastern University, Boston

Abstract. The task is to implement MSD Radix sort for a natural language which 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.

1 Introduction

Radix sorting is a simple and proficient sorting method, and it is commonly thought that it needs to inspect all characters of the input or trade that off for a high time and space complexity. But with some clever methods and implementations, radix sort can be made to run faster than strict comparison-based sorting algorithms. One such methodology that we have used in this paper is MSD radix sort which sorts by grouping letters. It is hard to sort Chinese characters in their native form because they have phonetic similarities which can make sorting a very messy affair. To make this task easier it would make sense to convert the Chinese characters to Pinyin, which is the romanization for Mandarin Chinese. So we take a String list which we convert to Pinyin. Then we deploy several sorting algorithms to sort the Pinyin string-list. While doing this we are also storing the index in an auxiliary list in order to sort the Chinese string-list. So in theory this method helps us to sort the Chinese String list without going about the complexities of understanding the actual phonetic nature of the words. We are simply sorting the Pinyin list and using their indexes also sorting the Chinese list which gives us a sorted Chinese list.

2 Background

2.1 LSD Radix Sort

The core concept behind LSD Radix sort is that it begins comparison and sorting from the last place of the string i.e least significant digit. But since we are using strings, they can also vary in length, so we pad them using 0 at the start to make all of them of equal length. Then we start sorting from the least significant digit to the most significant digit, i.e., from right to left. The performance of the sort here is 2W(N+R), where W is the maximum length of string present in the list, N is the number of words in the list, and R is a constant 65536, i.e. the number of Unicode characters.

2.2 MSD Radix Sort

LSD radix sort, unfortunately, ends up costing a lot of extra work if the String has varied lengths. To prevent this extra work and optimize further instead of starting from the least significant digit, we can start with the most significant digit and work our way to the bottom. So in this way, we can avoid padding for different length words. Here we partition the array into R pieces according to the first character and then recursively sort all strings that start with each character. Then lends us a performance of 2W(N+R) for MSD Radix Sort.

3 Related work

3.1 Radix Exchange

For binary alphabets, we use a special method called radix exchange. Here we split the strings into 3 piles: empty strings, the ones that begin with 0, and those that begin with 1. Here let's assume all strings are of the same length. Then the piles for empty strings become empty. Splitting can now be done as in quick sort but with a bit of test in place of the comparison. But if the strings are of different lengths, a full three-way split is required among the 3 bands no bit, 0, and 1. The three-way (radix) quicksort uses 2Nln characters compared on average for random strings. It's also cache-friendly, is in place, and has a short inner loop.

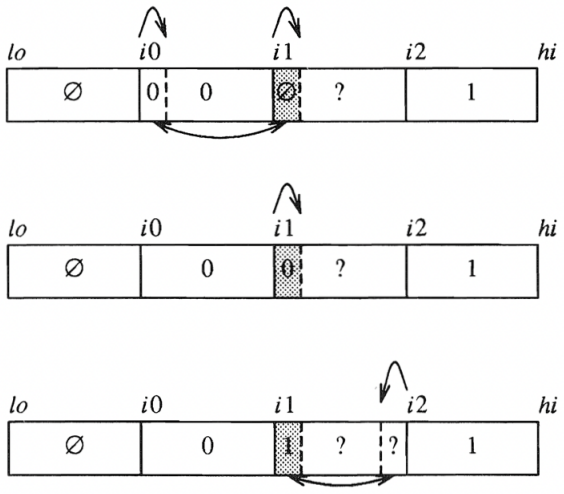

Fig. 1: Three-way splitting

3.2 In-Place MSD Radix Sort

We used buckets in MSD Radix sort, but if we want to reduce the amount of extra memory required from O(n) to O(1) we would need to swap elements in

place. For this purpose, we build a histogram to count the number of elements that belong to each bucket which only requires a single pass through the data. We use head and tail pointers to keep track representing the start and end of each bucket. So while we swap elements, the head and tail keep track of which elements are already swapped in the correct bucket and are incremented or decremented accordingly. The time complexity for the work is O(N log N), as the recurrence for the work is still the same. As for the space, only a constant amount of memory is required here. The depth, however, is now back to O(n) as the process here is sequential and it requires O(n) work and depth.

Algorithm 1 In-Place MSD Radix Sort

```
1: function inPlace(data, l, k, processors)
2:     if number of items in data = 1 then
3:         return data
4:     end if
5:     histogram ← buildHistogram(data, l, k, processors)
6:     heads ← [0]
7:     tails ← [histogram[0]]
8:     for i from 1 to 9 do
9:         heads[i] ← heads[i-1]+histogram[i-1]
10:        tails[i] ← tails[i-1]+histogram[i]
11:    end for
12:    for i from 0 to 9 do
13:        while heads[i]<tails[i] do
14:            item ← data[heads[i]]
15:            while item is not in the correct bucket do
16:                temp ← correct bucket of item
17:                swap values of item and data[heads[temp]]
18:                heads[temp]++
19:            end while
20:            data[heads[i]] ← item
21:            heads[i]++
22:        end while
23:    end for
24:    if l<k-1 then
25:        for i from 1 to 9 do
26:            value ← value from bucket[i]
27:            inplace(value,l+1,k,p)
28:        end for
29:    end if
30:    return value
31: end function
```

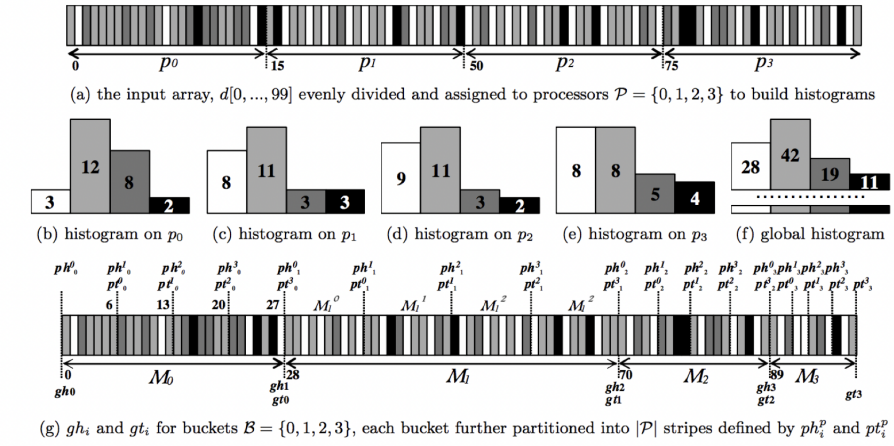

Fig. 2: Parallel histogram building

3.3 Adaptive Radix Sort

In adaptive radix sort, we choose the size of the alphabet adaptively so that it is a function of the number of elements still remaining. And so the number of buckets that need to be traversed will be proportional to the number of scanned characters and hence the cost of visiting buckets matches the cost of inspecting bit patterns. Because we might end up reading bits beyond the prefix, the number of bits that we read here might be greater than for the MSD radix sort. However, this will be linear extra cost at most times and will be dominated by the profit we get from not scanning as many empty buckets. To avoid looking at too many empty buckets, we have used two 8-bit buckets for bucketing which give an alphabet size of 65536. The idea here is to keep track of first and second-place characters, so if x1 and x2 different characters have been found in these two places we only need to inspect x1 and x2 buckets, which are typically smaller than 65536 buckets.

3.4 Cut-off to Insertion Sort

Due to the formation of large number of small subarrays caused by recursion, the time taken to sort these small subarrays individually through MSD Radix sort becomes expensive. Hence, cutting off to Insertion sort for smaller subarrays ensures that the subarrays are sorted more quickly and the larger subarrays are being taken care by original MSD Radix sort.

4 Experimental study

4.1 Benchmarking Results:

We measured the time by benchmarking the results of all methods for 250k, 500k and 1M, 2M, 4M size of list.

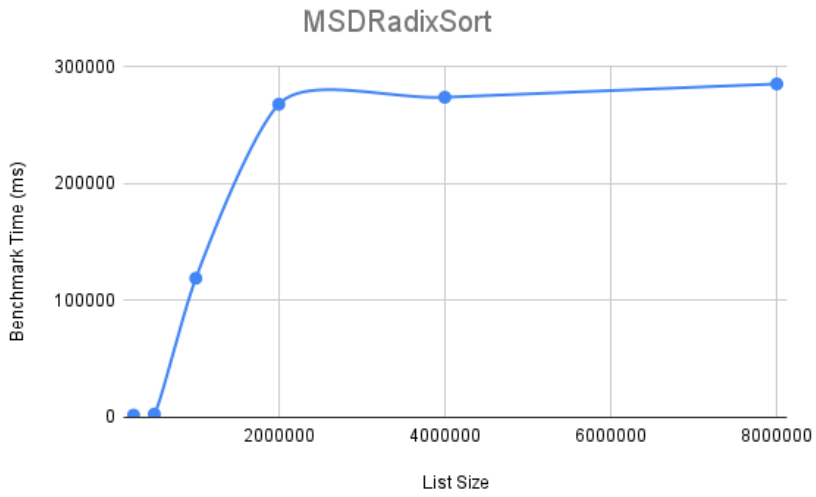

Fig. 3: MSD Radix Sort

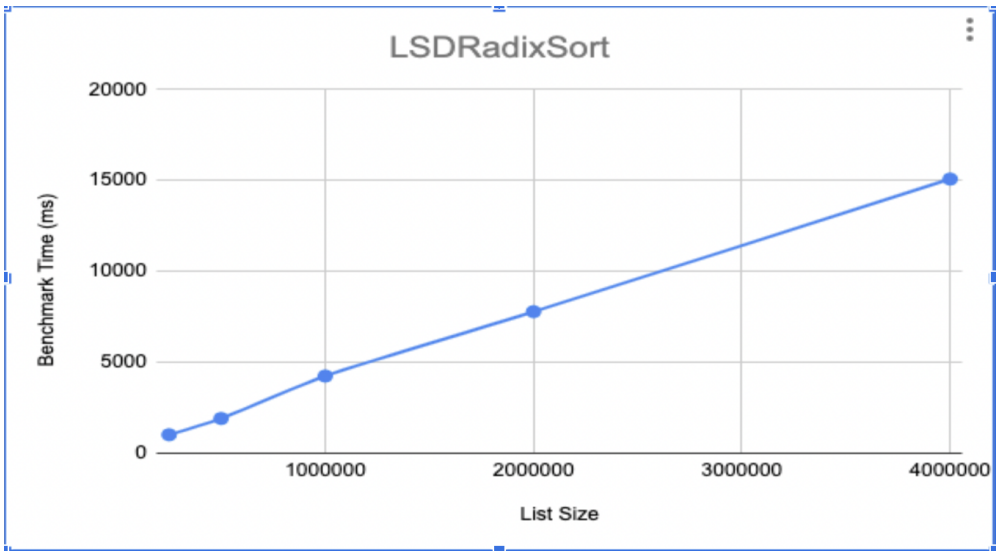

Fig. 4: LSD Radix Sort

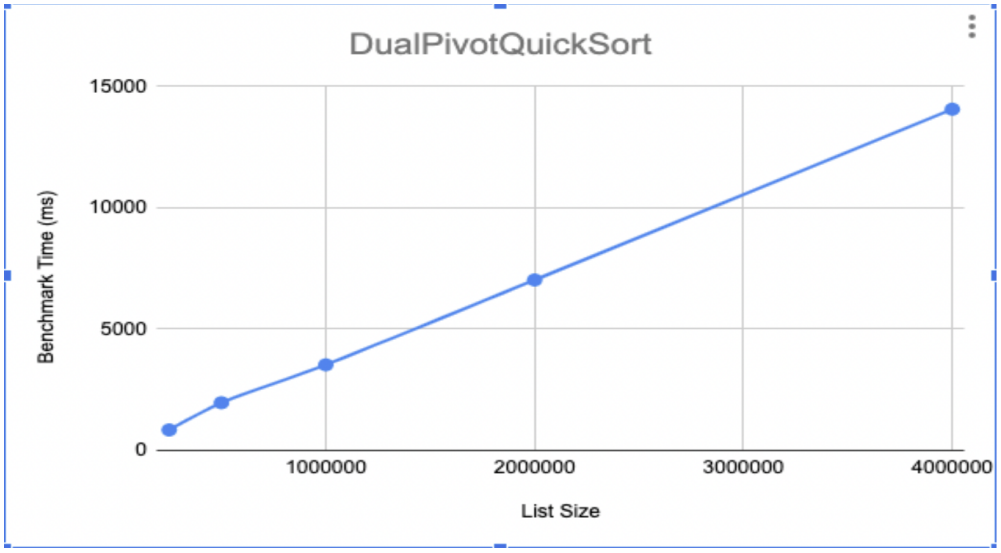

Fig. 5: Dual Pivot Quick Sort

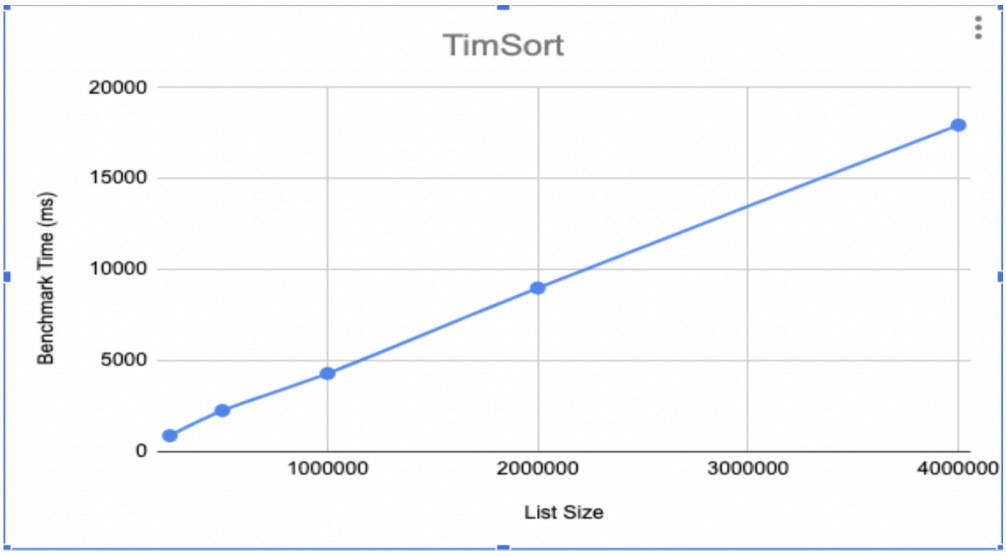

Fig. 6: Tim Sort

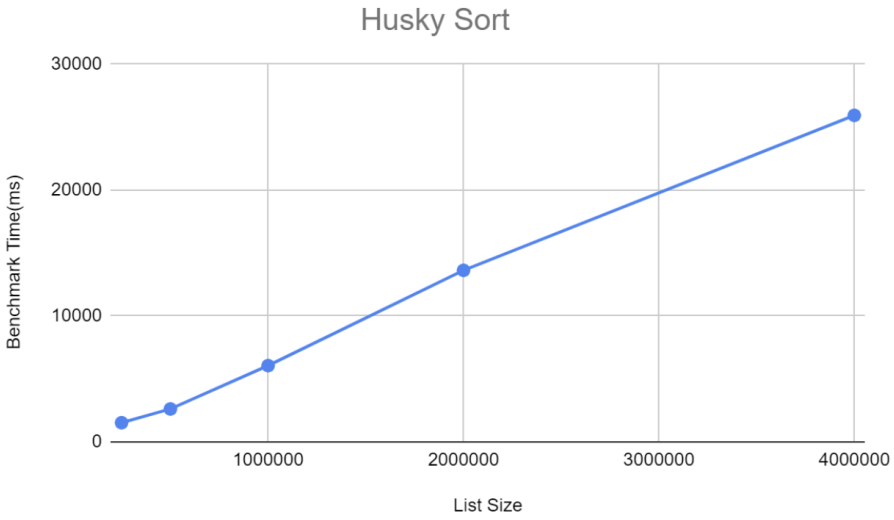

Fig. 7: Husky Sort

Post implementation of MSD Radix sort with cut-off to Insertion sort for a cut-off value of 16. It is observed that the benchmark time is exponentially reduced.

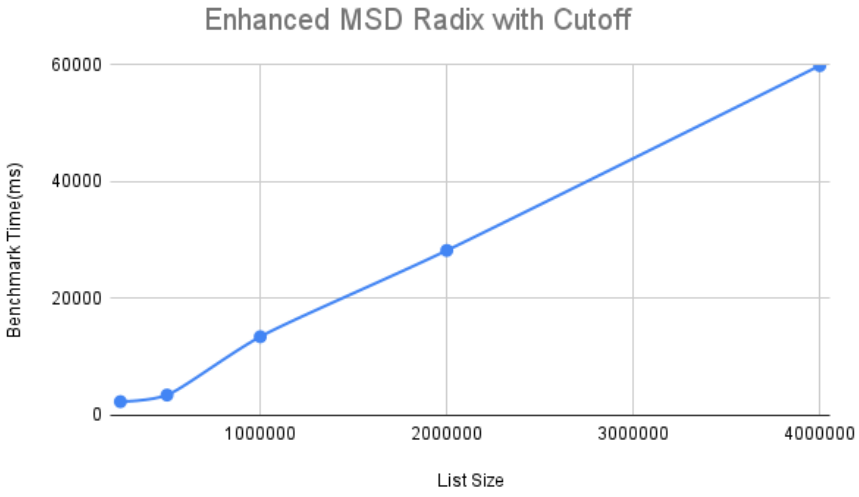

Fig. 8: Enhanced MSD Radix with cutoff to Insertion sort

4.2 Table

size of list	MSDRadix	LSDRadix	DPQuick	TimSort	HuskySort
250k	1986.18	1016.43	844.49	880.28	1524.32
500k	2879.04	1911.3	1965.7	2258.97	2616.29
1M	121316.39	4245.88	3529.24	4294.36	6059.85
2M	273390.71	7785.21	7025.6	8990.69	13627.9
4M	282746.86	15080.06	14062.73	17945.94	25938.19

Table 1: Benchmarking for different size of arrays for different sorts

size of list	Enhanced MSD Radix with Cutoff
250k	2347.19
500k	3475.5
1M	13462.7
2M	28238.28
4M	59883.98

Table 2: Enhanced MSD Radix with Cutoff value of 16

5 Conclusion

The task was to implement MSD Radix Sort for comparing Chinese words and names in pinyin order and to compare it with other sorting algorithms. In order to accomplish this, Dual Pivot Quicksort, Husky Sort, TimSort, MSD Radix Sort, and LSD Radix sort were implemented. There are instances of different Chinese characters having the same pinyin spelling(and hence the same precedence). Hence, sorting using these different algorithms produced different resultant arrays in Chinese, but their pinyin equivalents were the same. Hence, the resultant Chinese arrays were converted to pinyin to perform accuracy checks. As per the results posted above, MSD Radix Sort performs really poorly for smaller arrays but starts to level off as the array size reaches the 2-4 million mark.

Comparing different MSD Radix sorts as mentioned in related work, Radix exchange can be used when the input consists of the same length strings as it performs better. In-Place MSD Radix sort is a good method when we want to reduce the amount of extra memory from $O(n)$ to $O(1)$ by using a histogram to store the value of the count of alphabets. Also, Adaptive Radix sort suggests choosing the size of the alphabet adaptively. This will be linear extra cost at most times and will be dominated by the profit we get from not scanning as many empty buckets.

References

1. Engineering Radix Sort by Peter M. Mcllroy and Keith Bostic University of California at Berkeley; and M. Douglas Mcllroy ATT Bell Laboratories (1992
2. PARADIS: A PARALLEL IN-PLACE RADIX SORT ALGORITHM ROLLAND HE (rhe@stanford.edu)

3. Radix Sort
https://en.wikipedia.org/wiki/Radix sort
4. Implementing Radixsort by Arne Andersonn, Department of Computer Science, Lund University; and Stefan Nilsson, Department of Computer Science, Helsinki University of Technology