The comparison between Radix Sort and other sorts

Xuhui Zhu,
Weifeng Lai,
Cheng Yan

Abstraction

Your 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

We implement a MSD radix sort for Chinese, which uses Unicode characters, and compare it to other sorting algorithms including Timsort, Dual-pivot, Quicksort, Huskysort, and LSD radix sort. Our MSD sort performs well and beats most sorting algorithms.

The following sections show how we implement the algorithm, our benchmark results and our analysis based on the results.

2.Approching

Though Chinese characters use Unicode, the natural order of Chinese is sorted by Pinyin. We therefore use the pinyin4j library to convert Chinese characters to Pinyin before sorting, and map them back after sorting. We do the same thing on our LSD sorting, and because the Chinese characters we sort are not in the same length, we padding them with “#”, which is smaller than all English characters, to keep the sorting results correct. The same modification is also done on HuskySort.

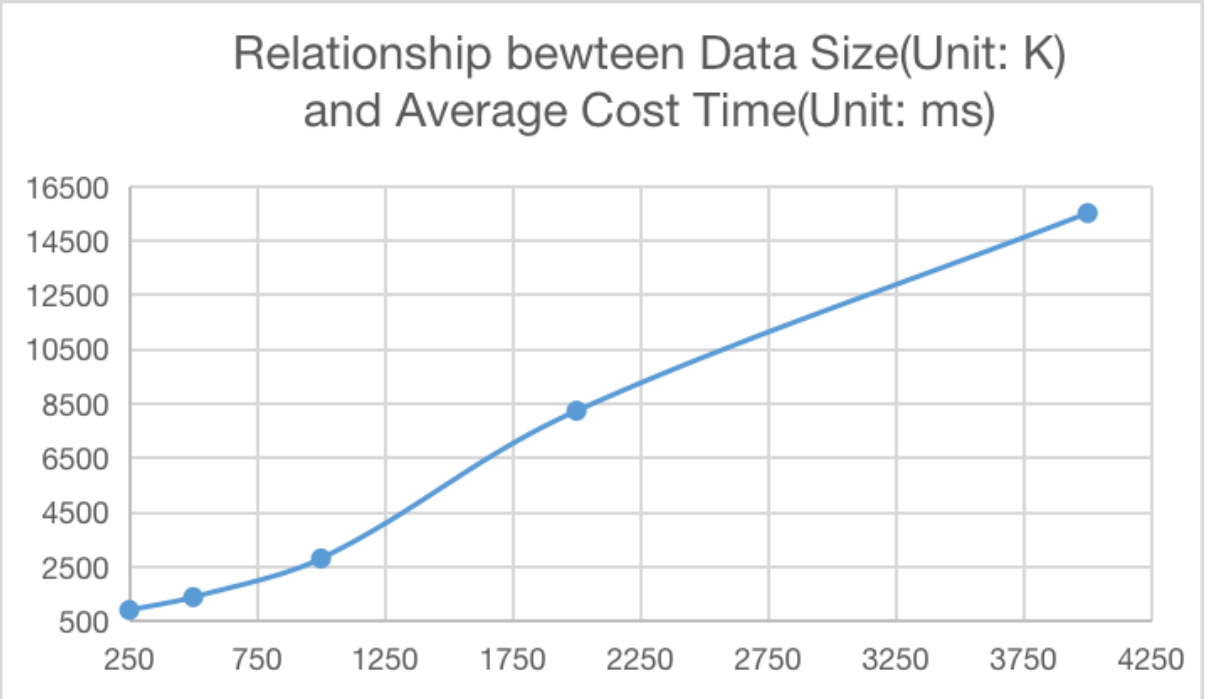
For Timsort and Quicksort, we use the Collator library to compare Chinese characters. The results are slightly different from algorithms that use pinyin4j because the two libraries treat Chinese characters differently(for example, “褚” has two pronunciations, which are “zhu” and “chu”, pinyin4j transfers it to “chu” while collator treats it as “zhu”).

3.Benchmark Results

We do tests on all sorting algorithms for 250k, 500k, 1m, 2m and 4m names 10 times each. We count the average total time from receiving the array of Chinese names to getting the sorted array, including converting Chinese to Pinyin or any other data pre-processing.

Average cost time for MSD Sort in different sizes of data:

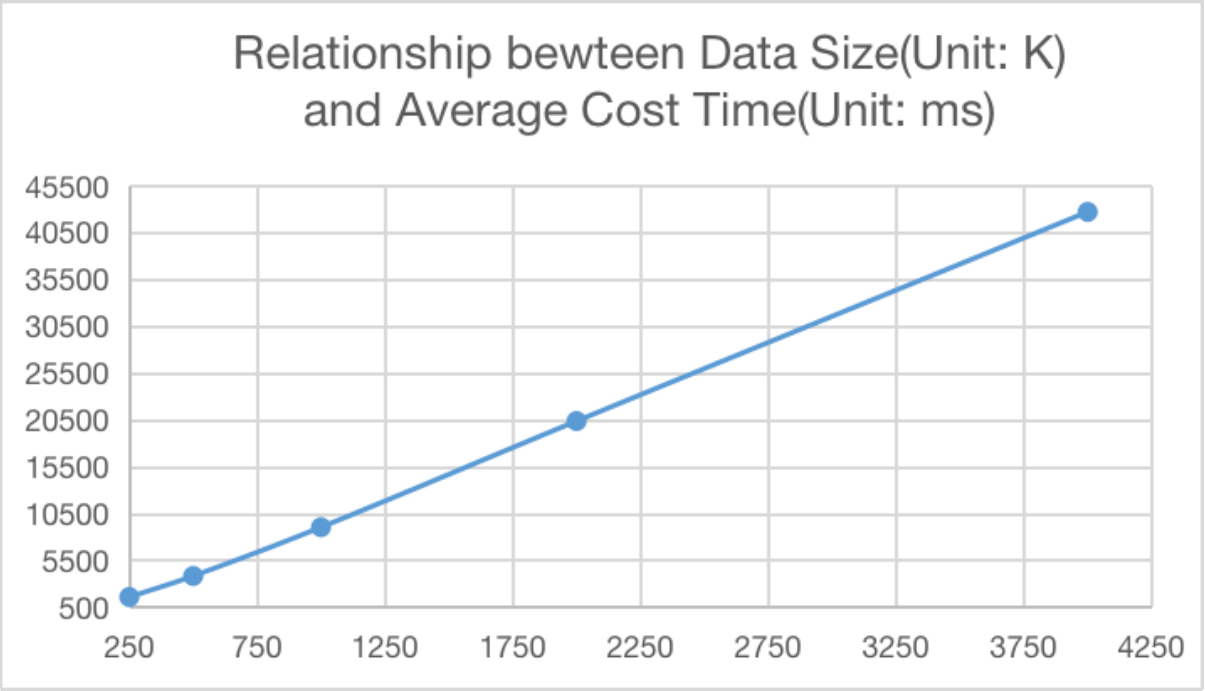
Data Size	Average Cost Time
250k	909 ms
500k	1382 ms
1m	2804 ms
2m	8241 ms
4m	15519 ms



Average cost time for LSD Sort in different sizes of data:

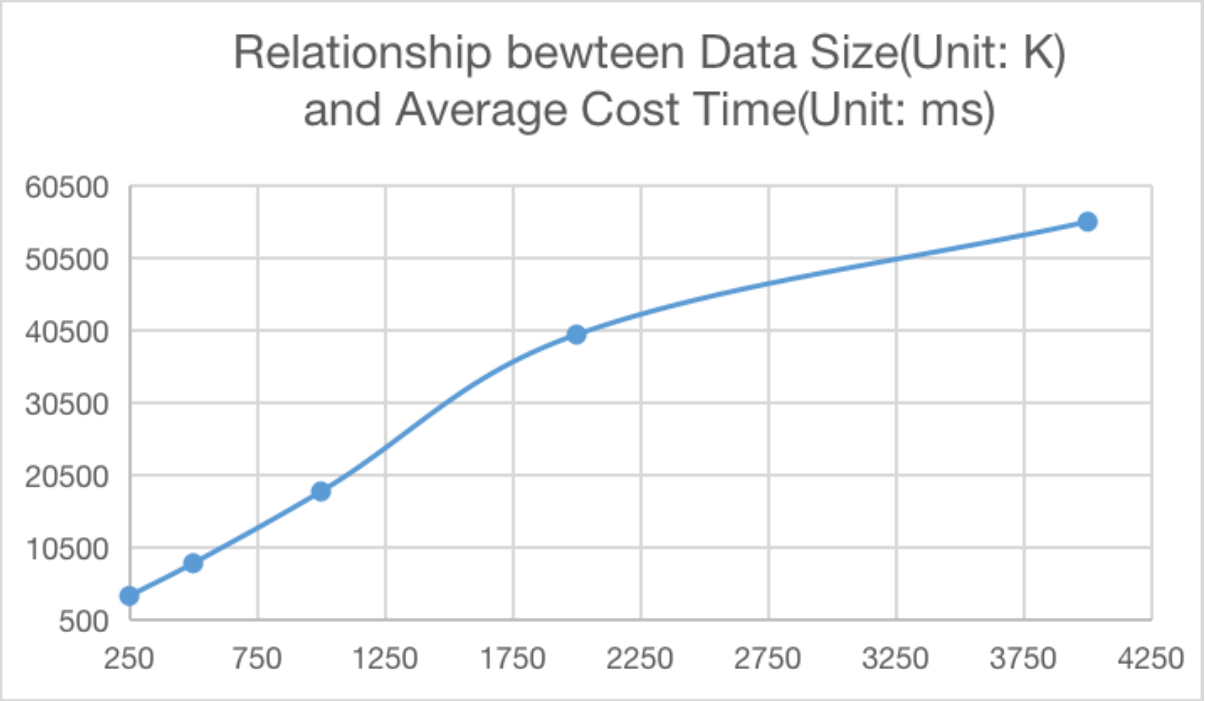
Data Size	Average Cost Time
250k	1656 ms
500k	3903 ms

1m	9100 ms
2m	20419 ms
4m	42724 ms



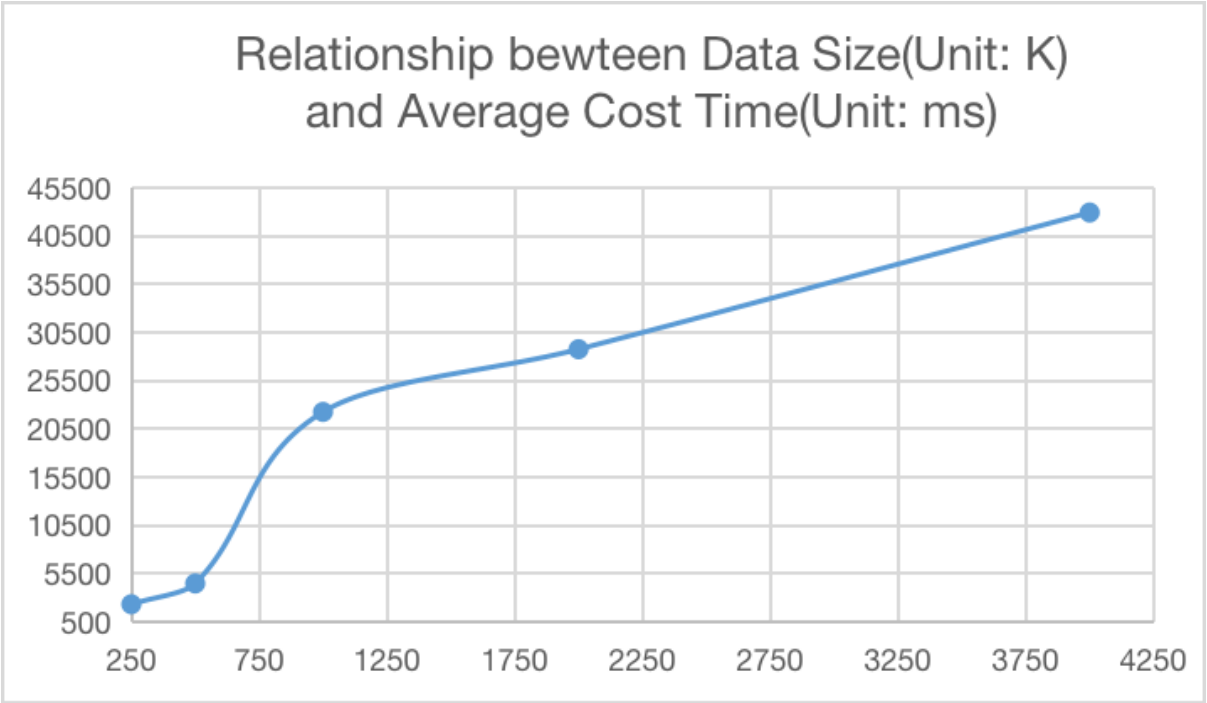
Average cost time for Dual Pivot QuickSort in different sizes of data:

Data Size	**Average Cost Time**
250k	3835 ms
500k	8348 ms
1m	18256 ms
2m	39909 ms
4m	55491 ms



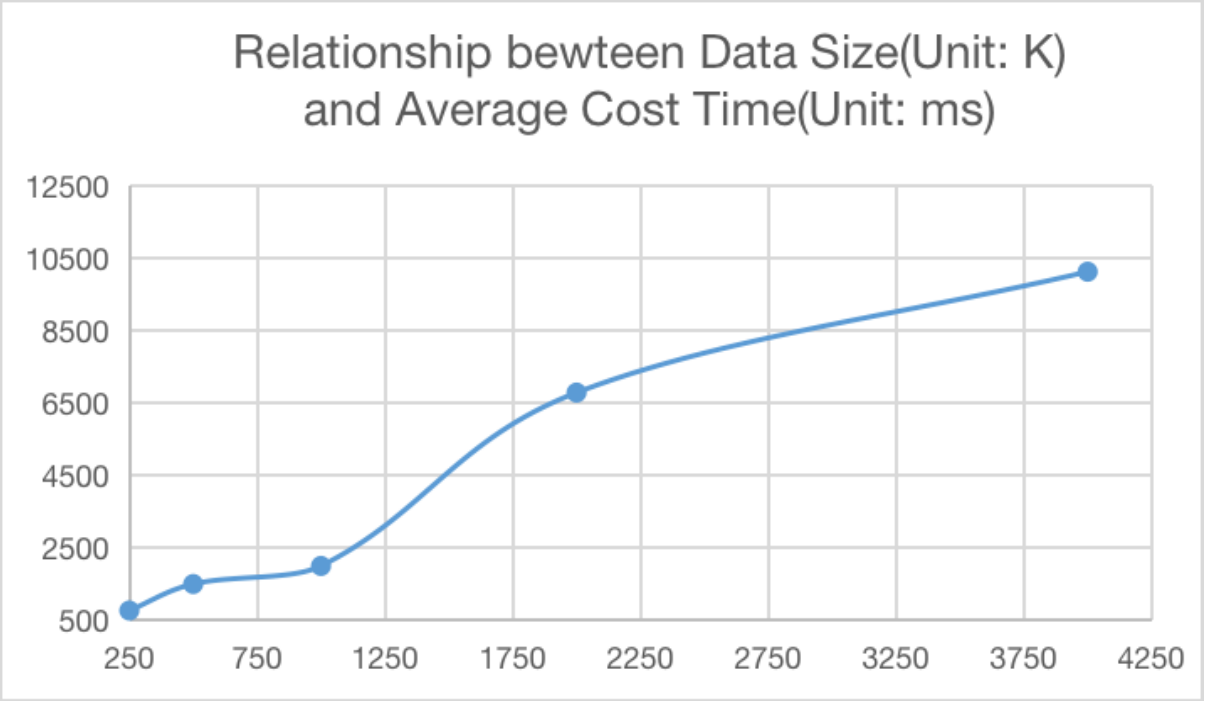
Average cost time for TimSort in different sizes of data:

Data Size	Average Cost Time
250k	2349 ms
500k	4503 ms
1m	22258 ms
2m	28750 ms
4m	42900 ms



Average cost time for Husky Sort in different sizes of data:

Data Size	Average Cost Time
250k	764 ms
500k	1496 ms
1m	1995 ms
2m	6780 ms
4m	10118 ms



4.Analysis

when we compare the average cost time for each sort algorithm in the same size of data and draw a sheet as follow:

Cost Time of / Data Size	MSD Sort	LSD Sort	Dual Pivot QuickSort	TimSort	Husky Sort
250K	909 ms	1656 ms	3835 ms	2349 ms	764 ms
500K	1382 ms	3903 ms	8348 ms	4503 ms	1496 ms
1M	2804 ms	9100 ms	18256 ms	22258 ms	1995 ms
2M	8241 ms	20419 ms	39909 ms	28750 ms	6780 ms
4M	15519 ms	42724 ms	55491 ms	42900 ms	10118 ms

We can find out that:
when data size is 250K, the ranking of time spent in ascending order is:

Husky Sort < MSD Sort < LSD Sort < TimSort < Dual Pivot QuickSort

when data size is 500K, the ranking of time spent in ascending order is:

MSD Sort < Husky Sort < LSD Sort < TimSort < Dual Pivot QuickSort

when data size is 1M, the ranking of time spent in ascending order is:

Husky Sort < MSD Sort < LSD Sort < Dual Pivot QuickSort < TimSort

when data size is 2M, the ranking of time spent in ascending order is:

Husky Sort < MSD Sort < LSD Sort < TimSort < Dual Pivot QuickSort

when data size is 4M, the ranking of time spent in ascending order is:

Husky Sort < MSD Sort < LSD Sort < TimSort < Dual Pivot QuickSort

We can conclude that, except HuskySort, sorts based on no comparison take less time than sorts based on comparison, and among sorts based on comparison, those that take multiple sort strategies have better performance than those with only one sort strategy.

Why does HuskySort perform better than sorts based on no comparison, for example, MSD Sort or LSD Sort? Our guess is these:

1. The time complexity of RadixSort is O(d*(n+r)), where d is the maximum number of digits in the string after the name is converted into pinyin, which is equal to 21 in our program, and r is the base, which is equal to 256 in our program. The time complexity of HuskySort depends on whether InsertionSort or HeapSort is used. It can be simply considered that the average time complexity is O(n * logn). When the size of the input data set is less than 4M, the coefficient d = 21, otherwise the coefficient logn <= 22, so RadixSort takes more time.
2. The encoding method of the two is different. For RadixSort, a piece of data will be encoded into a string consisting of pinyin and vowels, and for HuskySort, a piece of data will be encoded into a number. The comparison of strings is more time-consuming than the comparison of numbers.
3. HuskySort will shuffle the data in advance. Since there is some duplicate data in the input data set, after the data is shuffled, the performance of comparison and sorting will be better.