**Program Structures & Algorithms  
Fall 2021  
GROUP ASSIGNMENT**

**Wangzi Wen & Zuyue Xie**

1 INTRODUCTION

Sorting strings can be hard, though there are lots of great sorting algorithms developed throughout the years, there are no optimized algorithm that beats other algorithms in any circumstances. In java, String is a reference type which basically contains strings or list of characters; with different encoding methods like ascii and Unicode, there are huge numbers of characters from different languages that can fit in the String object. Sorting strings in alphabets is a popular and even standard way to demonstrate how good an algorithm is, since alphabets can be transferred into numbers which has a natural way of ordering. Today, we are trying to come up an alternative MSD radix sort to sort the strings which contains Chinese Characters.

2 BACKGROUND

It’s hard to count the numbers of Chinese characters, since they are transformed from the old oracle bone script. There are about fifteen thousand Chinese characters, and people use about thousands of them in daily basis. For comparison, since the Chinese characters have a lot of elements in it, like strokes and PINYIN the official way of sorting is by PINYIN first, then number of strokes if they have the same PINYIN and there is a certain order of strokes if two characters have the same number of Strokes. Other than that, since one single Chinese character may have multiple tones for different cases, it’s extremely hard to have a best order for Chinese characters.

Figure 1 a

A picture containing graphical user interface

Description automatically generated

Figure 2 aa

Logo

Description automatically generated

In this assignment, we are following the Collator from *IBM.ICU* as a correct order in our case. Since the computer would only store zeros and ones, there are lots of encoding method, typically Chinese Characters are encoded as Unicode, and that’s where we start. After examining the input, *shuffledChinese.txt* we believe it’s a one million lines of unique Chinese strings with two to three characters each and that’s also what we want for testing, a random array.

3 DISCOVERING

3.1 UNICODE

After testing, we found the Unicode doesn’t follow the correct order. As shown in Fig. 3, since PINYIN for *“曹玉德”* is *CAOYUDE and PINYIN for “刘持平”*

*is LIUCHIPING* according to the rule; C is supposed to be in front L, but neither UTF8 nor UNICODE demonstrate that. “E6” with “E5” and “u6” with “u5” is contradicted for the order. So, we move on.

Figure aa

Text

Description automatically generated with medium confidence

3.2 SORTING BY PINYIN

Since PINYIN is one crucial part with sorting Chinese characters, we are thinking to convert the Chinese character into PINYIN, sort them and then convert back to Chinese characters afterwards.

Figure aaa

Graphical user interface, application

Description automatically generated

We are using the *PINYIN4J package from BELERWEB* for transform the characters into PINYIN then sort them by Husky sort, Tim sort, Dual-Pivot Quick sort, MSD, and LSD radix sort algorithm. Since these algorithms can compare the integers naturally (each alphabets have its unique ASCII integer), Fig. 5 shows is our workflow.

Figure aaa

Diagram

Description automatically generated

Figure aaa

Graphical user interface

Description automatically generated with medium confidence

As shown in the graph, we have implemented five different methods, and this is the benchmark of them. As the husky sort dominate the sorting result, other sorts give reasonable benchmark result. Since there are radix sort included, here is the distribution numbers of different length of PINYIN from the input. Since we alter the LSD sort to make it can sort alphabets in different length, the formula for LSD and MSD would become and for MSD in best cases. Best case for MSD is nearly never going to happen since with one million inputs, the first digit would be only unique for 26 strings, so we disregard it. According to the formula, we have R = 256 since all alphabets can be represent in ASCII in 256 numbers. W = 17 for the LSD sort and W = 9.309231000000002 for MSD sort, the average number of widths as shown in Fig. 7. Since the W is the only variant for LSD and MSD, we can valid this by observe a nearly double time for LSD compared to MSD. Both Dual pivot quick sort and Tim sort provides just with different constant which won’t affect a lot, that’s explained why they performed about the same in the graph.

Figure aaa

Text, table

Description automatically generated

There is one thing get our attention that MSD is slower than the quick and time sort, by time complexity, it should beat the other two by 9(N+256) () versus N\*19 (); at least for the one million elements case since it have no repetitive elements. Maybe it’s because the ways we implement MSD or the hidden constant factor before every time complexity, and quick usually have the smallest constant factor.

3.3 PROBLEM WITH SORTING BY PINYIN

However, we found out some problem with compare PINYIN in Chinese Characters. Firstly, the stable and not-stable sort like Tim sort and quick sort, they have different output since there are a high chance that Chinese characters have the same PINYIN, so with different stability, the output varies. Secondly, if we combined PINYIN together, it breaks the rules by compare PINYIN one by one instead of characters. Like Fig. 8 shows, by rules, we should compare *“阿”* and *“阿”* and then *“鼎”* and *“迪”*; but with PINYIN, it can’t distinguish between Chinese characters other it compares PINYIN characters instead. Then by PINYN *“N”* is before *“Y”* so does the Chinese characters, but by right rules we know that this comparison shouldn’t even happened since it’s not compared the PINYIN with the Chinese characters with same index. That’s where we come with the next algorithm, the byte array.

Figure aaa

Text

Description automatically generated

3.4 SORTING BY BYTE ARRAY

After wondering a lot, we get inspired by husky sort, after reading the paper for husky sort, we come with the idea that convert the Chinese characters into primitives, with the faster compare time and maybe less times of comparison, we may have better performance for the algorithms. We initially want to learn the idea of convert Chinese characters into long just like husky did, but we just can’t come with the long number that is unique for each character and follow the same order of the original character. Then with the help of the Collator, we found out there is a *toByteArray* function where it can transfer the Chinese characters into byte array with correct locale, and here is where our algorithm starts. Start with the data, since we are sorting the byte array instead of string (character array), they have their unique length distribution as the Fig.9 shows. The W = 10.666943 for MSD and W = 11 for LSD.

Figure aaa

Text

Description automatically generated

We initially come with the idea to combine the byte array and the byte array from Chinese characters in Unicode (UTF-8) and sort them by every byte, and then convert the last part of the byte array, which is the original Chinese character in UTF-8 back to the characters. With MSD husky to have a fixed size byte array after the combination and MSD husk have a variant size byte array after combination. However, this idea didn’t work properly since there are some conversion issues with UTF-8 code back to Chinese characters. Finally, we come up with our final optimization, the MSD byte array, which is the leading algorithm in the chart.

Figure aaaa

Graphical user interface, application, table

Description automatically generated

3.5 SORTING BY OBJECT WITH BYTE ARRAY

Our Byte array method can be examined by Fig.11 which shows the flow.

Figure aaa

Diagram

Description automatically generated

After we construct the byte array object, every time MSD and LSD byte array method ask for the count array position, we return the corresponding int that convert from byte and -1 if it’s a shorter array; we also flip the sign of the byte array since we don’t want negative index for the array.

Figure aaa

Text

Description automatically generated

Figure aaa

Graphical user interface

Description automatically generated with medium confidence

Figure aaaa

Text

Description automatically generated

Figure aaa

Chart, bar chart, histogram

Description automatically generated

As shown in Fig. 15, here’s our result. It’s worth mention that Tim and Quick sort is using the collator.compare(String, String) method for compare and it’s just different than comparing byte array, but since they came with the same result, we also benchmark them for comparison; and just for fun, we put CollationKey as the object instead of ByteNode to see if there are any differences, CollationKey and ByteNode basically have serve the same purpose of compare the byte array field and store the original Chinese characters. As expected, since we reduce the W for LSD, the longest length of the key (17 to 11), the result increase from PINYIN method, although we have increase R by hundred, since byte goes up to 128 where character only goes to 26, R plays a small factor in formula W(N+R), which should not affect a lot. There is something we want to emphasis that we discover from the graph.

1. The comparison between MSD byte array and LSD byte array. (Text below will call MSD and LSD respectively)

MSD generally are better than LSD, when it only takes the average length as W, and it has a perfect N+R run time if the input is all unique in the first index. That’s the reason we believe in 1M comparison, MSD beat LSD. The MSD wins the competition by its average time complexity for random array which is about 3N versus about 11(N+R) for LSD. Then with the number of inputs goes up, the MSD start to lose the huge advantage and sometimes lost to LSD. After lots of thinking, we figured out that’s because the input; we just simply copy the 1M array again to achieve 2M and same thing for 3M and 4M. With critical cases, each byte array is guaranteed to appear **FOUR TIMES** in the array and that’s what slow down the MSD, since MSD must go through all the bytes and building more and more subarray for the recursion.

10.67 11

9.3 17

RECOMMANDATION AND THINKING

There are so many things going on under the JVM and implementation of the algorithm, and we can’t cover all of it in this assignment, so we list them here for future reference.

By efficiency, Is string of PINYIN comparison equal to char array comparison in JAVA? Is access each character of a String equal to access each element of char array in JAVA?

R reduce to 122 since the last letter z is 122 in ASCII table, we may even reduce to 27 if we subtract all the numbers by a constant 96 (since we need 0 to be the least number for the missing character and a is 97 in ASCII table) to increase the sorting performance.