INFO 6205 Final Project Report

Qinyun Lin , Xinlei Bian , and Yiqing Huang

College of Engineering
Northeastern University
Boston, MA
{*lin.qiny, bian.xin, huang.yiqin*}*@northeastern.edu*

***Abstract*—Radix Sort is a non-comparative sorting algorithm, which avoids comparison by creating and distributing elements into buckets according to the radix. In this project, we enhanced the traditional Radix Sort to support sorting Chinese characters in *Pinyin* order. We introduced two ways of *Pinyin* representation and designed a novel space-efficiency solution to store *Pinyin* characters and the related Chinese character.**

1. Enhanced MSD Radix Sort

In this section, we propose our *enhanced MSD radix sort algorithm*. Firstly, we introduce the different orders that Chinese characters can be sorted in, and how we represent them by index. Then, we detail our novel data structure which stores the Chinese words and its *Pinyin* representation in a significantly small size, and how to encode, decode, and apply this data structure to MSD radix sort.

1.1. Chinese Character Order

Unlike English, there is no default order for Chinese character order. There are several approval orders for Chinese characters, such as *character stroke number order*, *character components order* and *Pinyin order*. The first two order focus on the shape of Chinese characters, while the third one, *Pinyin*, focuses on the pronunciation. Nowadays, *Pinyin order* has become the most popular order for sorting Chinese characters both in China and throughout the world. Therefore, to sort Chinese characters, our first choice is to transfer Chinese into *Pinyin* characters, and then sort them by *English alphabetical order*.

1.2. Pinyin Length And Index Design

After analyzing Chinese pronunciation, we notice that we can reduce the "bucket" number when executing radix sort for Chinese. Considering MSD radix sort for English characters, there will be at most 26 possible letters in each position, and the length of a word varies. While in Chinese *Pinyin*, we represent a Chinese character in at most 6 alphabets plus a number stands for the tone. The number of possible letters for each location starting from the second one is less than 9.

In addition, if we strictly split a Chinese character into *Pinyin* by the pronunciation rule, we represent a Chinese character by **only 4** syllabification characters instead of by 6 alphabets and a number as mentioned above. For example, in naive *Pinyin*, we convert a Chinese character into `["s", "h", "u", "a", "n", "g", "1"]`, while according to syllabification, we represent the same character as `["sh", "u", "ang", "1"]`. In other words, `"sh"` and `"ang"` should be seen as a whole rather than several characters. Parenthetically, in syllabification, we call `"sh"` as *cacuminal*.

When doing radix sort, the performance relates to the maximum length of all the word to be sorted. The more characters those words have, the more layers of recursion the program will have. Thus, by using syllabification characters to represent for Chinese characters can shorten the length of the words from $7n$ to $4n$, which eventually brings a better performance.

Nevertheless, using *syllabification order* to sort Chinese words can sometimes results in a different outcome comparing to using *pinyin natural order*. For instance, considering about sorting the three Chinese characters `["za1", "zi1", "zhi1"]`. After splitting them by alphabet, it is `[["z", "a", "1"], ["z", "i", "1"], ["z", "h", "i", "1"]]`. It is obvious that after comparing the second digit of each character, they are sorted. The outcome is `["za1", "zhi1", "zi1"]`. After splitting them by syllabification, it is `[["z", "a", "1"], ["z", "i", "1"], ["zh", "i", "1"]]`. Since `"zh"` has a lower priority than `"z"`, the result will be `["za1", "zi1", "zhi1"]`, which is different from that of the former method. The good news is that those conflicting cases are rare.

Since representing Chinese characters in *syllabification* brings too much advantages, we would like to reserve or even embrace this way. Thus, we implement the two ways to sort Chinese. Parenthetically, getting inspiration from *HuskySort (Hillyard et al. 2020 [1])*, if we firstly sort a series of Chinese words in *syllabification order* and get an almost sorted result, and then input the result to sort it in *Pinyin natural order*, we can both promise the correctness of the outcome as well as a nice performance.

1.2.1. Chinese Character in Pinyin Letter Representation.

Now, most of the sorting algorithms sort pinyin based on alphabetical order. It is an easy way for those who do not know Chinese to understand this sorting strategy since this sorting strategy sorts pinyin as an English word. For instance, “zi, zhi” is sorted as “zhi , zi” since “i, h” is sorted as “a,h” in alphabetical order. The longest pinyin has a length of 6 and not all alphabets will appear in each position. For instance, there are not any pinyin that begin with “i”. So we create 6 dictionaries to hold the correct order of all possible alphabets in each position. There are 23 alphabets in the first dictionary, 9 in the second dictionary, 7 in the third dictionary, 7 in the fourth dictionary, 5 in the fifth dictionary, and only one in the sixth dictionary.

1.2.2. Chinese Character in Pinyin Syllabification Representation.

The most significant benefit of sort by syllabification is reducing and maintaining the iteration (or recursion) times for each position in MSD radix sort to 4. For every pinyin, the largest syllabification number is 3, so we can transfer pinyin to a String array of size 4 (including a tone which is an integer less than 5), like [sylla_1, sylla_2, sylla_3, tone]. For those syllabification less than 2 or 3, we simply set the following sylla_2 and sylla_3 to an empty string. In Chinese pinyin, “empty” is ranked first.

For each syllabification position, we create a dictionary and set the correct index of all possible syllabification. The largest index for first position is 36, and 23 for the other two (including empty string), both of them are much less than the max number of ASCII and Unicode.

In particular, we considered the influence of “cacuminal” in this case, which makes the pinyin order here slightly different from English order, but makes more sense for Chinese.

1.3. Representation of Pinyin

In this part, we would like to introduce our novel way to store the converted *Pinyin* characters and its related Chinese character.

As mentioned in *Section 1.2.1* and *1.2.2*, we use 7 integers to represent for a Chinese character in *Pinyin letter* and use 4 integers to represent in *Pinyin syllabification*. Since in `Java`, a primitive integer use 4 bytes of memory, the total space needed for storing the *Pinyin* representation of a Chinese character can be $4*7 = 28$ or $4*4 = 16$ bytes. Even when using `java.text.Collator` to set strength of each digit and use 7 or 4 ASCII characters to store them, it can cost 7 or 4 bytes.

Since the index of each of digit is fixed and is far less than 256, we use a byte array to represent for a Chinese character in *Pinyin*. What is more, for the convenience to get the result in the original string, we use an extra 4 bytes to store the original UTF-8 character. There may be some waste of the memory, since most of the Chinese character in UTF-8 uses $2-3$ bytes. What is more, choosing array as the container takes the benefit of low-cost accessing. However, if the number is not fixed, it can be much more complicated to both decoding and to do radix sort. *Figure 1* shows the representation.

1.3.1. Encoding.

Take encoding a syllabification representation as example:

```java
void convertStrArrToByteArr(String[] wordArr) {
    final ChsCharToIdxArrBySylla cctiabs = new
    ChsCharToIdxArrBySylla();

    for (int k = 0; k < wordArr.length; k++) {
      String word = wordArr[k];
      int byteShift = 0;
      int[] pinyinIdxArr = cctiabs.CharAt(word);

      for (int i = 0; i < word.length(); i++) {
        // 1st idx
        compArr[k][byteShift] |= (byte) (
    pinyinIdxArr[i * 4] - 1);
        // 2nd idx
        compArr[k][byteShift + 1] |= (byte)
    pinyinIdxArr[i * 4 + 1];
        // 3rd idx
        compArr[k][byteShift + 2] |= (byte)
    pinyinIdxArr[i * 4 + 2];
        // 4th idx
        compArr[k][byteShift + 3] |= (byte) (
    pinyinIdxArr[i * 4 + 3] - 1);
        // original word
        copyWordToByteArr(word, i, k, byteShift);
        byteShift += 8;
      }
    }
  }
```

1.3.2. Decoding.

Take decoding a syllabification representation as example:

```java
void convertByteArrToStrArr(String[] wordArr) {
    for (int k = 0; k < wordArr.length; k++) {
      byte[] bA = compArr[k];
      StringBuilder sb = new StringBuilder();
      for (int i = 0; i < bA.length; i += 8) {
        int j;
        for (j = 4; j < 8; j++) {
          if (bA[i + j] != 0x00) {
            break;
          }
        }
        // Stop when no remaining character
        if (j == 7) {
          break;
        }
        sb.append(
            new String(Arrays.copyOfRange(bA, i +
    j, i + 8), StandardCharsets.UTF_8));
      }
      wordArr[k] = sb.toString();
    }
  }
```

1.4. Enhanced MSD Radix Sort Algorithm

Now, we have a nice data structure to store both the *Pinyin* representation and the original Chinese character. It’s time to implement Radix Sort based on the data structure. The main difference is on calling `charAt()` when getting

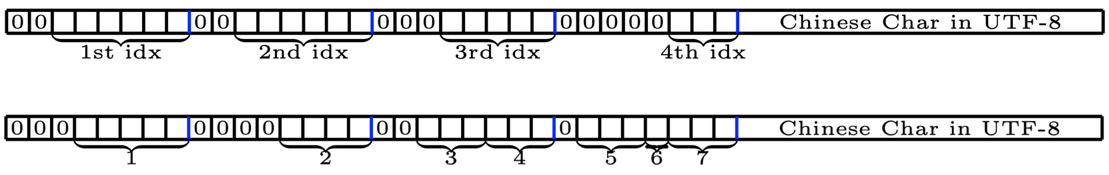


Figure 1. Representation of Pinyin

the index of each digit, and the `step`. For the traditional radix sort, the `step` is always 1, while here our `step` varies periodically according to the current location (`d`).

Still, we take by syllabification as example:

```
void sort(int lo, int hi, int d, int cirIdx) {
  if (hi <= lo) {
  } else if (hi < lo + cutoff && hi > lo + 1) {
   InsertionSortPinyin.sort(compArr, lo, hi, d)
  ;
  } else {
   int[] count = new int[radix[cirIdx] + 2];
   for (int i = lo; i < hi; i++) {
    count[charAt(compArr[i], d, cirIdx) +
  2]++;
   }

   for (int r = 0; r < radix[cirIdx] + 1; r++)
     // Transform counts to indices.
   {
    count[r + 1] += count[r];
   }

   for (int i = lo; i < hi; i++) {
    aux[count[charAt(compArr[i], d, cirIdx) +
  1]++] = compArr[i];
   }

   // Copy back
   System.arraycopy(aux, 0, compArr, lo, hi -
  lo);

   int step = getStep(cirIdx);
   int period = radix.length;

   for (int r = 0; r < radix[cirIdx]; r++) {
    sort(lo + count[r], lo + count[r + 1], d +
   step, (cirIdx + 1) % period);
   }
  }
 }

int getStep(int cirIdx) {
  switch (cirIdx) {
   case 0:
   case 1:
   case 2:
    return 1;
   case 3:
    return 5;
   default:
    return 0;
  }
 }

 int charAt(byte[] bArr, int d, int cirIdx) {
  if (d < bArr.length) {
   int loc = (d / 8) * 8;
   switch (cirIdx) {
    case 0:
     return bArr[loc];
    case 1:
     return bArr[loc + 1];
    case 2:
     return bArr[loc + 2];
    case 3:
     return bArr[loc + 3];
   }
  }
  return -1;
 }
}
```

2. Experiment

2.1. Benchmark Design and Implementation

To measure the performance of our MSD algorithm, we run a benchmark on MSD sort, LSD sort, dual-pivot quicksort, and Huskysort sort algorithm. We use the SorterBenchmark class from the husky repository to implement the benchmark.

2.1.1. Pre-processing And Post-processing. Since we use pinyin to sort Chinese characters, the preprocessing includes transferring Chinese characters to pinyin and index mapping. Moreover, after applying sorting algorithms, there should be some post-processing to make sure sorting algorithms work properly.

2.1.1.1 Pre-processing. For Huskysort, dual-pivot Quicksort, and Timsort, the preprocessing should construct a list that each element contains Chinese words to be sorted, corresponding pinyin, and comparison strategies. Since we have two ways to sort pinyin, by letters or by syllabification. Two classes, NameByLetter and NameBySyllabification, are respectively designed to store the data and comparison strategies fitting two comparison strategies. When instantiating NameByLetter or NameBySyllabification class, the input Chinese words are transferred to pinyin by using the external library `pinyin4j`. Then, referring to the corresponding pinyin index maps, an array of pinyin is converted to an array of indexes. These two classes both implement Comparable Interface. In the comparison function, two objects are compared by comparing the array of indexes from left to right.

In pre-processes, a list containing NameByLetter or NameBySyllabification objects is constructed to hold all elements to be sorted. Each object instantiates with input a word of Chinese characters to be sorted.

For the LSD and MSD sorting algorithm, the input is a string function. In preprocessing, a byte array is constructed that stores Chinese characters and corresponding mapping indexes.

2.1.1.2 Post-processing. After the sorting process, the output of Huskysort, dual-pivot Quicksort, and Timsort will be a sorted list containing NameByLetter or NameBySyllabification objects. To output the sorted words of Chinese characters as a String array in post-processing, the sorted list is iterated in order, and written the corresponding Chinese words into a string array.

For LSD and MSD sorting algorithms, the output is a byte array that contains a word of Chinese characters and mapping indexes. In post-processing, the byte array is converted to a string array with sorted words.

Then, after all sorting processes are finished, the sorted list should be checked to make sure the sorting algorithms work properly.

2.1.2. Implementation. We use the `SorterBenchmark` class from the husky repository to implement the benchmark. To run the benchmark with `SorterBenchMark` class, we import and modify some sorting algorithms provided in the huskysort repository.

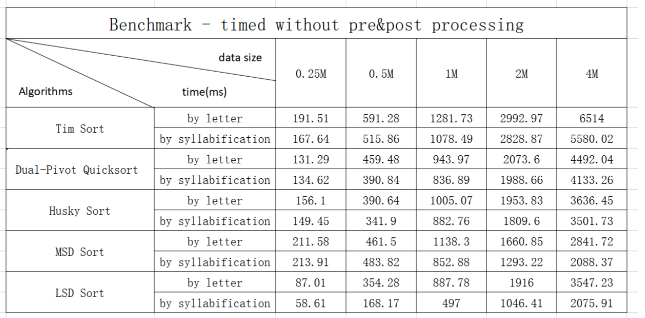
- *Dual-pivot Quicksort:* It is implemented by importing the `QuickSort_DualPivot` class from the `Huskysort` repository.
- *Timsort:* It is implemented by importing the `TimSort` class from the `Huskysort` repository.
- *HuskySort:* `PureHuskySort` class is imported from the `huskysort` repository and is used to implement husky sort. Besides, since `PureHuskySort` is not inherited from `SortWithHelper`, a `PureHuskySortWithHelper` is created and inherited from `SortWithHelper`.

To apply the pre-processing and post-processing, a `HelperWithTesting` class is implemented from the `Helper` interface with implementing `preProcess` and `postProcessing` methods for Quicksort, TimSort, and Huskysort. TMSD and LSD sorting algorithms implement `SortWithHelper` and override `preProcess()` and `postProcess()`.

The input data is a $1M$ Chinese names that is read from an txt file and constructed as a $4M$ list. All the five algorithms are timed with 125000, 250000, 500000, 1000000, 2000000, and 4000000 input Chinese words. To get more accurate results, they run 100 times for each size of the input and use the average time cost as a result.

What is more, we designed an experiment to find out the influence of pre-processing and post-processing in MSD radix sort. In the first experiment, preprocessing and post-processing are not timed while the pre-processing and the post-processing are timed in the second experiment.

2.1.3. Result. Due to the limitation of space, we stored our raw data in our repository.



Benchmark - timed without pre&post processing						
Algorithms	data size / time(ms)	0.25M	0.5M	1M	2M	4M
Tim Sort	by letter	191.51	591.28	1281.73	2992.97	6514
	by syllabification	167.64	515.86	1078.49	2828.87	5580.02
Dual-Pivot Quicksort	by letter	131.29	459.48	943.97	2073.6	4492.04
	by syllabification	134.62	390.84	836.89	1988.66	4133.26
Husky Sort	by letter	156.1	390.64	1005.07	1953.83	3636.45
	by syllabification	149.45	341.9	882.76	1809.6	3501.73
MSD Sort	by letter	211.58	461.5	1138.3	1660.85	2841.72
	by syllabification	213.91	483.82	852.88	1293.22	2088.37
LSD Sort	by letter	87.01	354.28	887.78	1916	3547.23
	by syllabification	58.61	168.17	497	1046.41	2075.91

Figure 2. Raw-data of Algorithms with Maximum 3 Words

3. Analysis

3.1. Compare Among Sort Algorithms

To compare the performance and view the growth of each algorithm, we plot the results on a *log/log graph (Figure. 3)*. The values of *log(Time)* are modified by the *Least Mean Square* method. It is obvious that LSD performed the best, and Huskysort was better than other comparable algorithms. Besides, it is interesting that MSD performed the worst when data size was small, and it turns out to be the best as data size increases.

3.1.1. Huskysort Performance Analysis. We believe that the main reason for Huskysort not performing very well is the weakness of the *Chinese Husky Encoder*, the max length of a permitted Chinese character string is only 2. Therefore, Huskysort can not get the perfect encode and has to do an extra sort. However, the encode still helps it get to partial order very quickly. That is why Huskysort performs better than any other comparable sorts.

Besides, another reason why we did not encode Chinese characters via the `Collator` Collator is that we sort in various pinyin orders as described in *Section 1.2*.

3.1.2. Radix Sort Performance Analysis. We notice that MSD is not the best until the data size is large enough, which is different from our expectation. Based on the differences of the implementation between MSD and LSD, we think the main reason is that we used recursion for each position in MSD while it was iterated in LSD. Due to the limit of time, we can not implement non-recursive MSD and do further experiments to prove our analysis, but we will finish it in the future.

We are not surprised to find that they are the best. Apart from the optimizations we did to radix sort as mentioned in *Section 1.2* and *1.3*), there is another reason. The data to be sorted contains at most 3 Chinese characters, which means the iteration number of radix sort will be lower than other sorting cases since the time complexity is $O(w * n)$ where w (the maximum length of a word) is small. The maximum w value is 21 for sorting by letter, and 12 for sorting by syllabification.

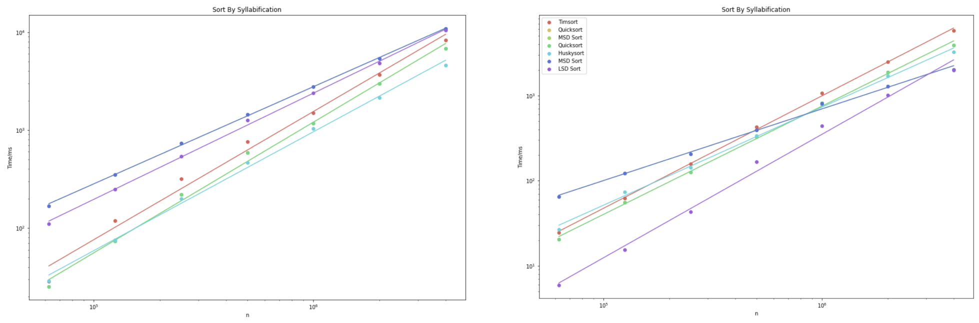


Figure 3. Comparison of Algorithms with Maximum 3 Words *(Log-Log Plot)*

In addition, the $O(w*n)$ also tells why MSD turns out to be the best in large data size cases. We know that iteration is definitely faster than recursive, while this influence becomes little as data is increasing. Because recursion operations number only relative to the w, it will become less and less significant as n comes up. Furthermore, according to $O(w*n)$ and $O(n*log(n))$, we know MSD will only perform better than comparable sorting when $w < log(n)$.

3.2. Compare Among Various Pinyin Orders

As *Figure. 4* shows, we found that the performance of sorting by syllabification will be better than by letters in any algorithms. The reason is that syllable sorting has fewer positions in pinyin to be processed as described in *Section 1.2.2*.

Although the final result of syllable sorting is slightly different from that of pure letter sorting as described in *Section 1.2.2*, only a small fraction of pinyin with `["ch", "sh", "zh"]` will be different. If there is a difference in the result, it will not be widespread. Compared with the information entropy of the original random data, the entropy between the two orders is little, so the influence of the different final orders on the performance should be ignored in the sorting algorithm. Even if we consider the syllable sorting result as not in order, it won't take much work to input the almost-sorted result to sort again by another order.

To prove the analysis above, we did an extra experiment outside the benchmark. (Due to the limitation of time, it is hard for us to add it into the benchmark.) We took Java System sort and Huskysort as examples and did one more sort as letter order after we sort as syllabification, and compared the sorting time to just sort by letter. It is surprising that even if we did 2 times sort, it is still faster than straightly sorting by letter.

3.3. Advanced Analysis for Pre-processing And Post-processing

As *Figure. 3* and *5* shows, when the pre-processing and post-processing are timed, our radix sorts turn out to be the worst. The reason is that we used a different data structure for radix sort as described in *Section 2.3*, which does a good job on saving space and element accessing, but when doing encoding and decoding especially decoding, its bit operations and converting byte arrays to string one character by on character after sorting is time-consuming. While for comparable sorts here, we just need to get an attribute from the object we constructed. Therefore, the time consumed for post-processing in our radix sort is large. This is the drawback of our radix sort.

Although the time consuming for decoding is now far from satisfactory, we believe that it is still valuable to be optimized since do save a bunch of space for both *Pinyin* and Chinese characters representation. This should be helpful for some special cases if the space is strictly limited.

4. Conclusion

We introduced two kinds of Chinese Pinyin representation, which help us to reduce and maintain the word length for radix sort in only 7 and 4 digits. Based on *Pinyin letter* and *Pinyin Syllabification*, we designed related dictionaries for each position to get the correct index effectively. In addition, we came up with a novel way which stores Pinyin and the relative Chinese character only 8 bytes, so that we don't need any extra data structure and space to save them. Although the time for encoding and especially decoding (converting byte arrays into characters) is far from satisfactory, it performs relatively well among a series of sorting algorithms.

In particular, *Pinyin Syllabification* representation has shown the excellent performance among all the sorting algorithms. The most significant drawback is that its order is slightly different from English alphabetical order, which can be solved by an extra sort with little time consuming because of partial order.

In the future, we will try to figure out a better pre-processing and post-processing way and more efficient sort algorithms for the extra sort in *Pinyin Syllabification*.

Appendix

1. Unit Test

We conducted test cases for all codes that we implemented and modified.

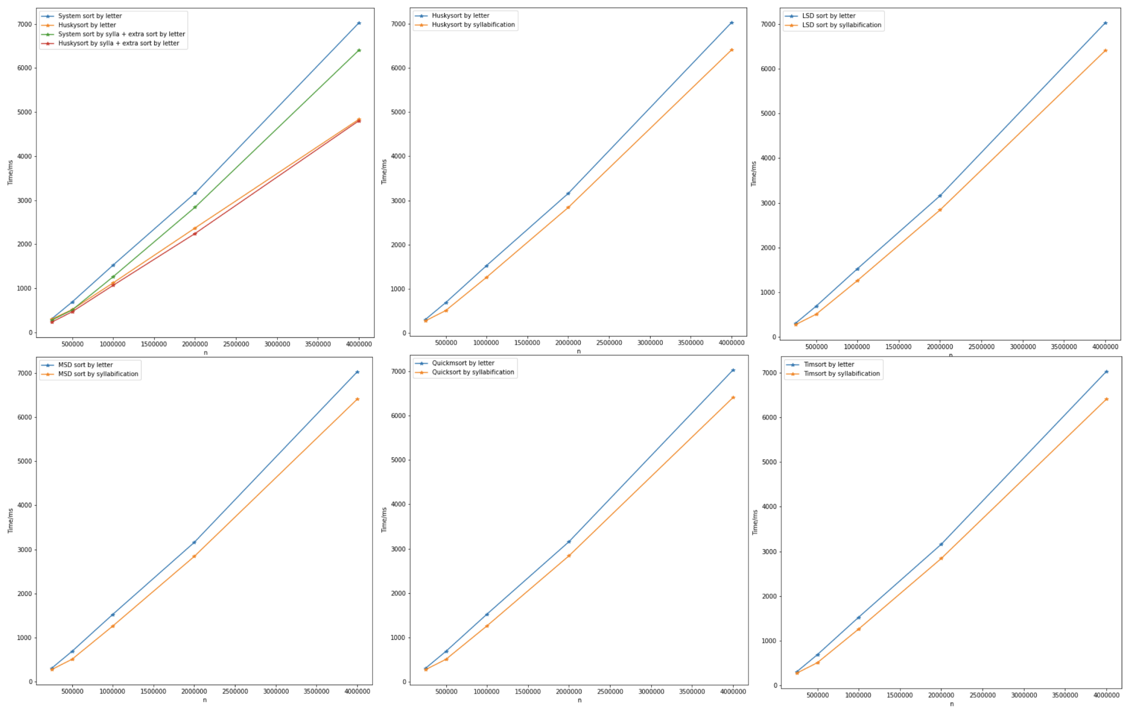


Figure 4. Comparision on The Same Algorithm By Various Orders

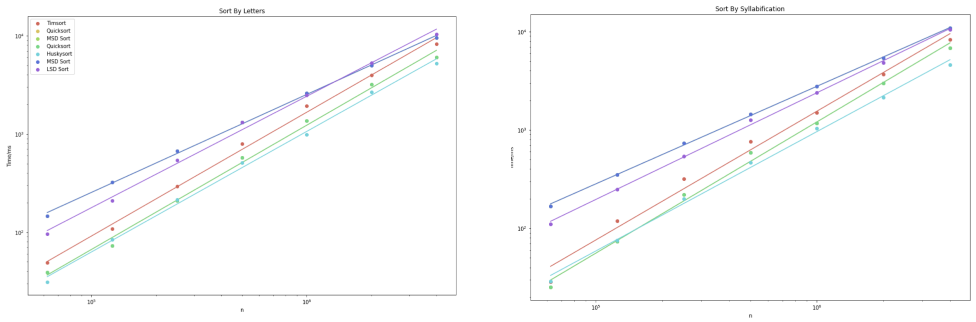
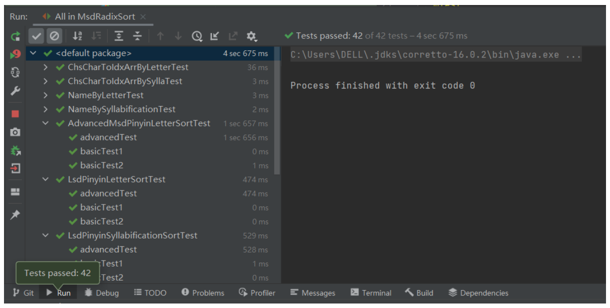


Figure 5. Comparison of Algorithms with Maximum 6 Words *(Log-Log Plot)*



```
Run:  All in MsdRadixSort
Tests passed: 42 of 42 tests – 4 sec 675 ms
<default package>                        4 sec 675 ms
  ChsCharToIdxArrByLetterTest                   36 ms
  ChsCharToIdxArrBySyllaTest                     3 ms
  NameByLetterTest                               3 ms
  NameBySyllabificationTest                      2 ms
  AdvancedMsdPinyinLetterSortTest        1 sec 657 ms
    advancedTest                         1 sec 656 ms
    basicTest1                                   0 ms
    basicTest2                                   1 ms
  LsdPinyinLetterSortTest                      474 ms
    advancedTest                               474 ms
    basicTest1                                   0 ms
    basicTest2                                   0 ms
  LsdPinyinSyllabificationSortTest             529 ms
    advancedTest                               528 ms
    Test1                                        1 ms
    Test2                                        0 ms

C:\Users\DELL\.jdks\corretto-16.0.2\bin\java.exe ...

Process finished with exit code 0

Tests passed: 42
```

Figure 6. Unit Test Result

References

[1] R. Hillyard, Y. Liaozheng *et al.*, “Huskysort,” *arXiv preprint arXiv:2012.00866*, 2020.