Jason Paul Darivemula (002198774)

Ebenezer Ajay Williams(002166250)

INFO 6205 Fall 2021 Project

Abstract

We present a report that summarises the findings of sorting English text and natural Chinese text using MSD radix sort, LSD radix sort, Dual Pivot Quick sort, Husky sort and Tim Sort algorithms. We present a comparative study on the various sorts in this report.

For these experiments we use the doubling technique where the size of the data is doubled in each trial for benchmarking the algorithms. We took the readings for array sizes starting from 15625 till 4 million which gave us 9 data points per sorting algorithm.

Background and overview

For the purpose of these experiments, we use 2 different text files consisting of 4 million strings of english and chinese. Unlike the case of English characters where the ASCII representation of the alphabets are ordered in ascending order, the conventional order for Chinese is according to the English order of the Pinyin which is not necessarily in monotonically increasing order of the unicode. For this purpose we utilize the java default Collator (java.text.Collator) and the IBM Collator (com.ibm.icu.text.Collator). Both these collators have different ordering of the pinyin and so we perform the experiments using both the collators.

For MSD and LSD Radix sort, since we are using the pinyin order for Chinese strings, we had to come up with a custom implementation of the MSD and LSD sort which will be described below. However, for sorting English strings, we use the default implementation found in the repository and we set the radix to 256 which is nothing but the ASCII range.

For husky sort we have two versions which are the quick husky sort and the pure husky sort for which we have implemented a new Chinese encoder.

Tasks performed

- Implemented various sorting algorithms in java.
- Implemented classes required for timing and benchmarking the algorithms.
- Implemented unit tests.
- Did the benchmarking for the sorting algorithms by setting the number of trial runs to be 100.
- Performed literature survey to understand the background of our work.

Implementation

General Implementation Strategy

- We leveraged the code found in the class repository and added methods to perform similar operations using the *java.text.Collator* and *com.ibm.icu.text.Collator*.

- We design a **FileUtil** class that we use for reading and writing data to text files located in the RandomString and SortedString directories. The read method in the class can be passed a parameter specifying the number of strings to be returned in the array.
- We implemented a simple Enum called **Language** that we used in the **Driver** class to benchmark the sorting algorithms.
- The **Driver** class contains the following static methods
    - *benchMarkAnySort()* - This method takes a Consumer as a parameter named *sortFunction* which is the sorting method that needs to be benchmarked. It takes in another parameter named lang which specifies what language to use as input for the sorting method.
    - *getDataToSort()* - This method returns an array of strings which serves as the input to the sorting method being benchmarked.

Implementation Strategy For MSD and LSD Radix Sorts

- For MSD Sort and LSD Sort, we used a TreeMap to store the keys and passed the Collator to determine the ordering of the keys in the map. These implementations are used while sorting Chinese strings. Here we explain the concept implemented for MSD Radix Sort and the same one has been used for LSD Radix Sort.
    - Consider we initiated MSD sort on a given input of strings. If we freeze the running of the algorithm precisely at the start of a recursive call on a sub-array of the original array with the value of **d** set to *x*. The sub-array consists of strings that have the same first *x* characters.
    - **Count** - The first step is to count the number of occurrences of different characters at position **d**. There are 2 possibilities when we encounter a character-
        - Encounter a new character: We add it to the TreeMap and store the count as 1.
        - Encounter an examined character: We increment the count by 1 for the character in the TreeMap.

      The value of a key(character) in the tree map now represents the count of the strings having that character at position d.
    - **Transform -** The second step is to transform the counts to useful indices. We leverage the fact that keys (i.e. characters in our case) are stored in an orderly way in our TreeMap. We iteratively compute the sum of counts (calculated earlier) of all keys that come before a key '**k**' and store it as the value for the key. The values in the tree map now represent the starting index of the character in the auxiliary array.
    - **Distribute -** Now that we have the starting indices, distributing the strings in an auxiliary array works similar to how normal MSD works with a count array. We keep incrementing the value whenever the key is encountered while distributing. The values in the tree map now represent the starting index of the succeeding character in the auxiliary array.
    - A simple tabular representation of what does the value of a key represent in the TreeMap

Step	**Value for a given key C**
Count	The **count** of strings in the array that have character **C** at the position d.

Transform	The **starting index** of the strings having character **C** at the position d.
Distribute	The **starting index** of the strings having succeeding character **(the next key)** at the position d.

- The main differences between this implementation and the general implementation of MSD sort
  - The TreeMap will consist of only characters that are encountered whereas the general implementation would initiate a count array of the size of the alphabet.
  - This implementation allows for a passing of a comparator to obtain custom orderings.

Implementation Strategy For Husky Sort of Chinese Strings

- For Husky Sort, we added a HuskyCoder named **ChineseCoder**.
- Our central idea behind the coder is that for sorting Chinese strings we use a Collator and Collator has a method called *toByteArray()* which returns a byte array representation of the key. Based on the method description - *if two CollationKeys could be legitimately compared, then one could compare the byte arrays for each of those keys to obtain the same result,* we decided to convert the byteArrays to String and encode it.
- We used the **unicodeCoder** to encode the string obtained from the byteArray.
- To verify if this encoding improves the time taken to sort we ran a few tests with variable inputs where we benchmarked the first run of husky sort and the second run of Tim Sort and compared it by sorting the same input using only Tim Sort. Below are our findings.

Number of Chinese Strings	**Time Taken for First Pass of HuskySort - QuickHuskySort**	**Time Taken for Second Pass of HuskySort - Tim Sort**	**Overall Time Taken by HuskySort**	**Overall Time Taken by only Tim Sort**
250000	288	2041	2329	4163
500000	194	3532	3726	8687

Time measured in milli secs.

**This time was obtained by doing multiple individual trials and then taking the average.*

- The coding was not perfect as the strings passed to the unicodeCoder had length greater than 4. Moreover, the byteArrays had negative values and so the conversion to strings resulted in random characters such as '□'. We proceeded with this encoding to obtain the benchmarking which is captured in *Table B*.
- As we were working on figuring out a better encoding we came across Professor Robin's encoding mechanism. We used this encoding and the findings have been stated under Graphs & Findings.

Tabulated Data

Benchmark timings measured for english text in milli secs.

Number of strings	Quick sort dual pivot	Tim sort	Quick Husky Sort	LSD Radix	MSD Radix	Pure Husky Sort
15625	12.99693999	5.77008052	3.25279298	1.6005736	1.59684114	2.77954125
31250	9.59922752	8.81633191	6.78085525	3.81122032	3.05648162	6.18640021
62500	19.73408131	20.63617085	17.88101981	18.76384052	8.33072188	11.63893631
125000	53.84545949	50.78244	43.80316784	68.71808407	23.8026744	29.41506606
250000	123.4015912	118.1596807	115.0248854	342.4934936	56.25233763	89.44368037
500000	308.9290994	263.4864199	270.9646531	903.5052577	118.1403496	182.3211858
1000000	744.9933485	621.8781507	594.8859302	2008.756693	318.8671335	440.1193686
2000000	720.1873312	631.2204251	585.0848651	1873.591962	288.0626413	464.3342607
4000000	747.302311	630.90115	556.2273393	2101.779661	323.7129713	389.6452477

Table A - The average time recorded for sorting random English strings

*Note - For QuickHuskySort and PureHuskySort we used the **englishCoder**. The maximum length of the strings for our input was **10** and so the encoding was **perfect**.*

Benchmark timings measured for Chinese text in milli secs.

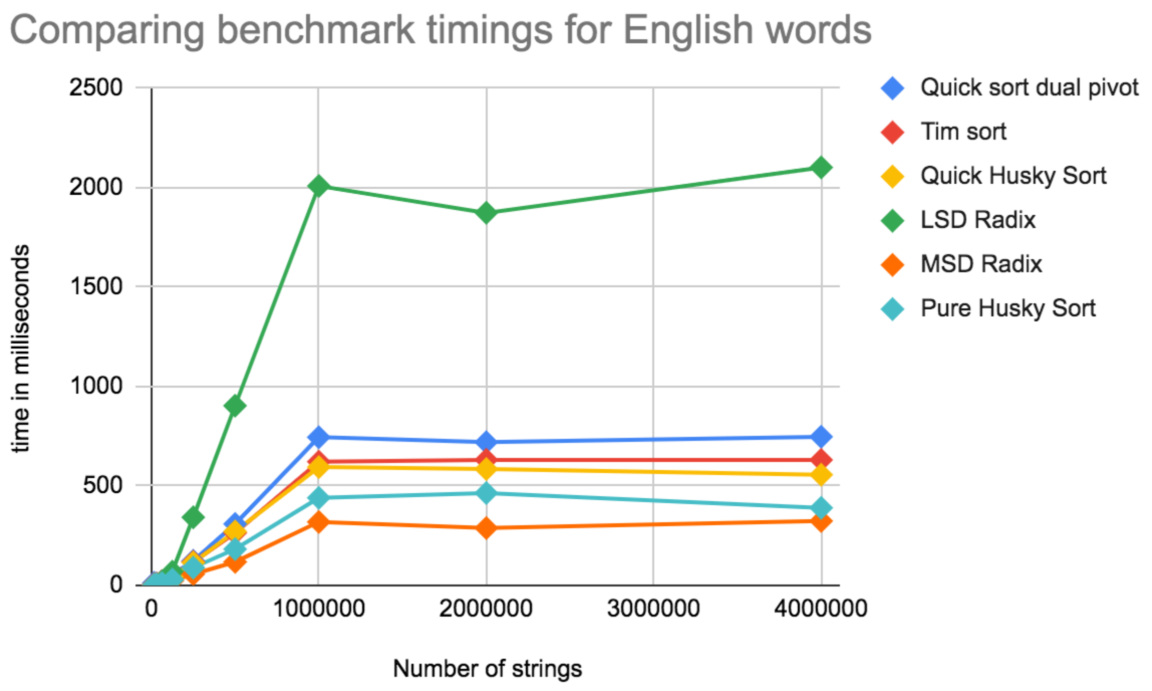
Number of strings	Quick sort dual pivot	Tim sort	Quick Husky Sort	LSD Radix	MSD Radix	Pure Husky Sort
15625	248.9166066	68.80313355	40.31728255	1566.765698	1084.23419	132.4269146
31250	581.9828231	139.8660678	84.569402	3081.122785	2135.153149	169.0497902
62500	1011.474929	306.5400566	180.3907964	6288.437827	4476.656293	216.0441482
125000	1465.196506	674.9790307	385.8999837	12617.64583	3847.852838	441.4615976
250000	1872.608787	1487.091981	861.8213791	10034.45491	8272.206276	892.6188224
500000	4080.678239	3240.293632	2035.332246	20494.38934	18047.10225	1999.334747
1000000	9174.621854	7049.329813	4439.051592	41230.35074	40585.05737	6814.02412
2000000	19987.80019	15867.99131	9641.259589	83919.67399	69643.90669	9553.95698
4000000	43880.81461	33506.47417	21908.81678	169679.0614	135062.7858	21478.69579

Table B - The average time recorded for sorting random Chinese strings

Number of strings	Prof Hillyard's encoding - Quick Husky Sort	Prof Hillyard's encoding - Pure Husky Sort	Custom encoded Quick Husky Sort	Custom encoded Pure Husky Sort
15625	46.1335882	25.4672796	40.31728255	132.4269146
31250	90.2823946	48.196931	84.569402	169.0497902
62500	151.7757744	92.2576956	180.3907964	216.0441482
125000	244.8494176	199.2935768	385.8999837	441.4615976
250000	441.834637	401.024247	861.8213791	892.6188224
500000	892.5900984	860.394988	2035.332246	1999.334747
1000000	1961.292336	1786.3054	4439.051592	6814.02412
2000000	3897.096	3679.096701	9641.259589	9553.95698
4000000	8191.511835	7810.398933	21908.81678	21478.69579

Table C - The average time recorded for sorting random Chinese strings for different encoders of Husky Sort.

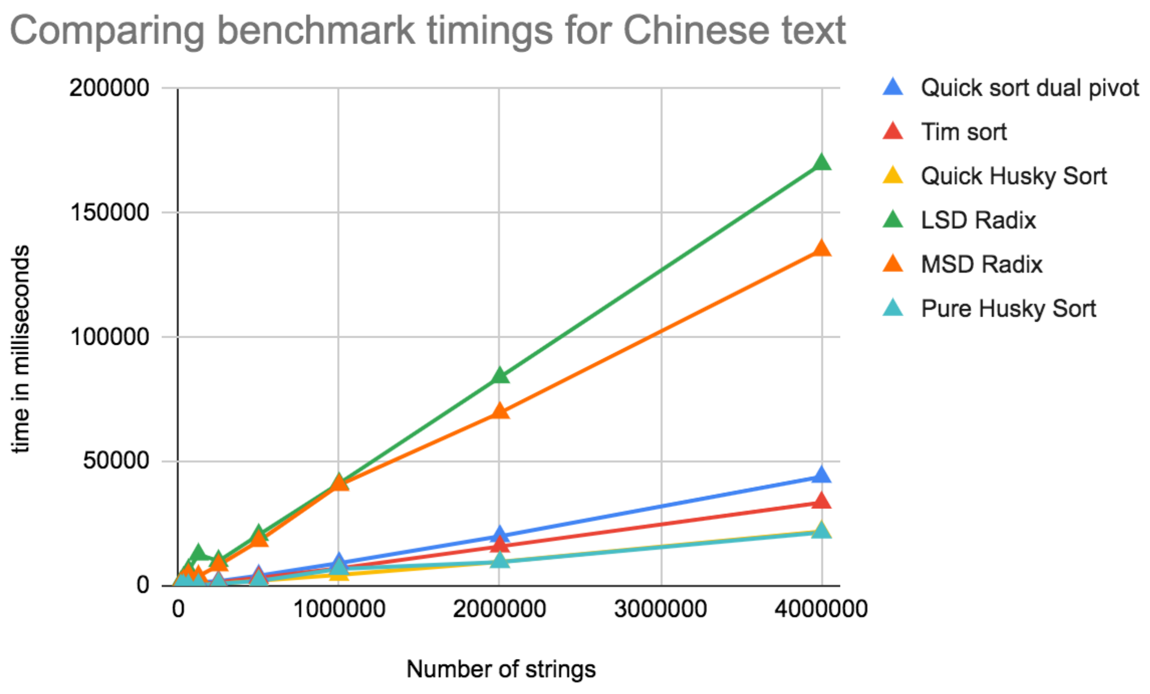
Graphs & findings



Fig(i) Graphical Output of Table A

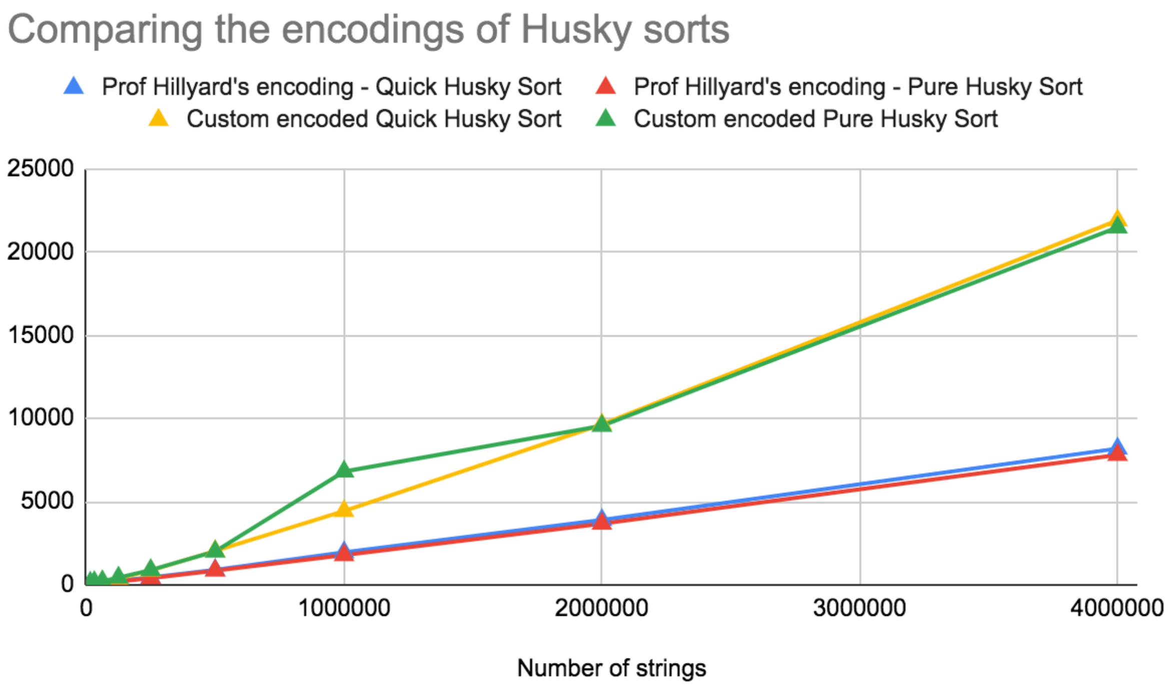
- In the above graph we see that MSD Radix sort performs the fastest.
- The pure Husky sort performs well, however it is still slower than the MSD radix sort.
- The LSD Radix sort is the worst performing sort with all sizes of the data.

- The Tim sort has the same performance as the Quick Husky sort when the size of the data to be sorted is small, however the Quick husky sort is clearly the winner when the size of data to be sorted increases.
- The dual pivot quick sort is the slowest among all sorts except LSD Radix sort.



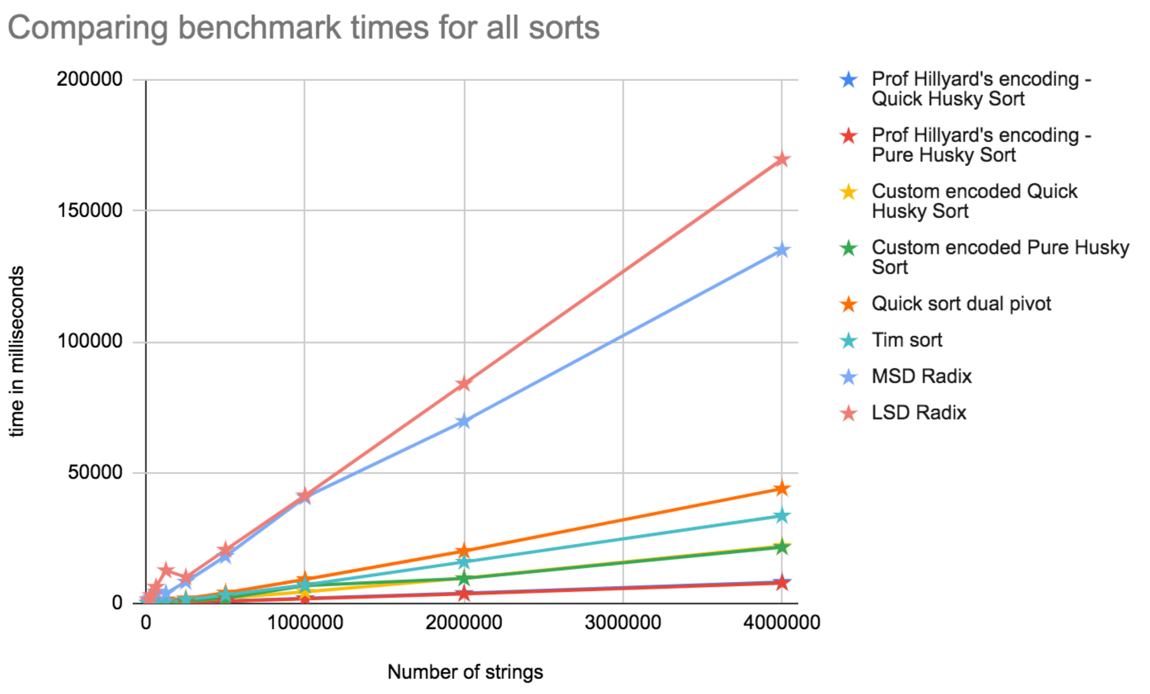
Fig(ii) Graphical Output of Table B

- For Chinese text, our implementation of MSD Radix sort does not perform well. The delay is assumed to be with the TreeMap which uses the Collator to store the text in a binary tree.
- The LSD Radix sort performs the worst just as it did with English text.
- The Quick Husky sort and the pure husky sort seem to have more or less similar performances and are the fastest when comparing Chinese text.
- The Quick Husky sort performs better than the pure husky sort when the size of the data to be sorted is small. However as the size increases, the Pure husky sort has a slight edge.
- Tim sort performs better than the Dual pivot quick sort, but the Husky sorts still outperform all sorts.



Fig(iii) Graphical Output comparing our encoding with Professor Hillyard's

- As we see in the above graph, the encoding provided by Professor Hillyard outperforms our custom encoding.
- We can also see that Pure Husky Sort outperforms Quick Husky Sort.



Fig(iv) Graphical Output of All Sorting Algorithms

- The above graph shows that Professor hillyard's encoding of Pure Husky sort outperforms all other sorts.

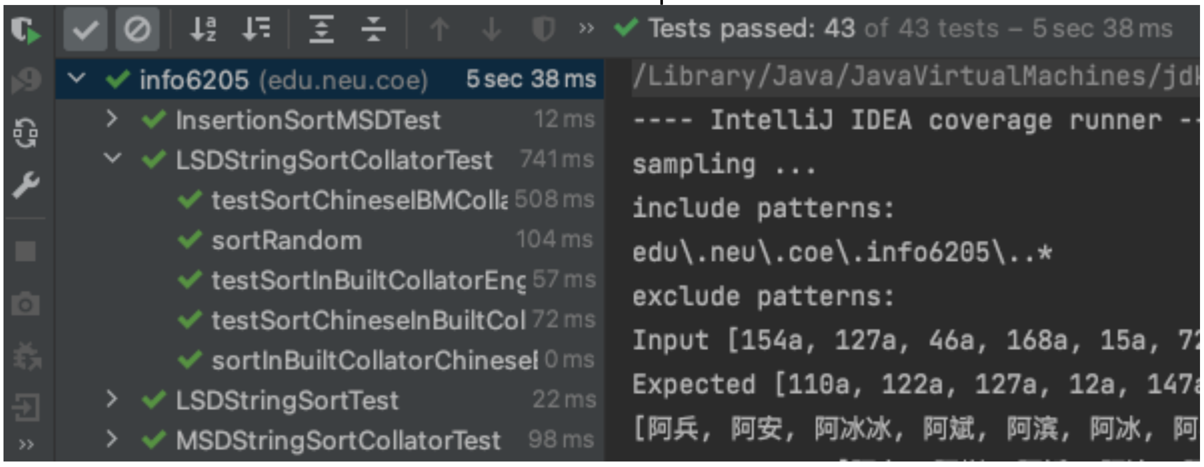
Unit tests

- We made our unit tests extensive to have maximum coverage across all the methods that we used in our experiments.

Testing Strategy for Sorting

- Our strategy for testing the sorting algorithms was to make use of the *Arrays.sort()* method available in java. We obtained our testing **input** (i.e. random strings) to sort random strings from text files that we added in the directory named RandomString.
- We then obtained the **expected** outcome by sorting the copy of the **input** using *Arrays.sort()* method.
- We passed the **input** to the sort method with the makeCopy option set to false so that the **input** array was the one that got sorted.
- We asserted that the **expected** outcome is the same as the sorted output from our sorting algorithm.

Below are Screen shots of unit tests written and passed

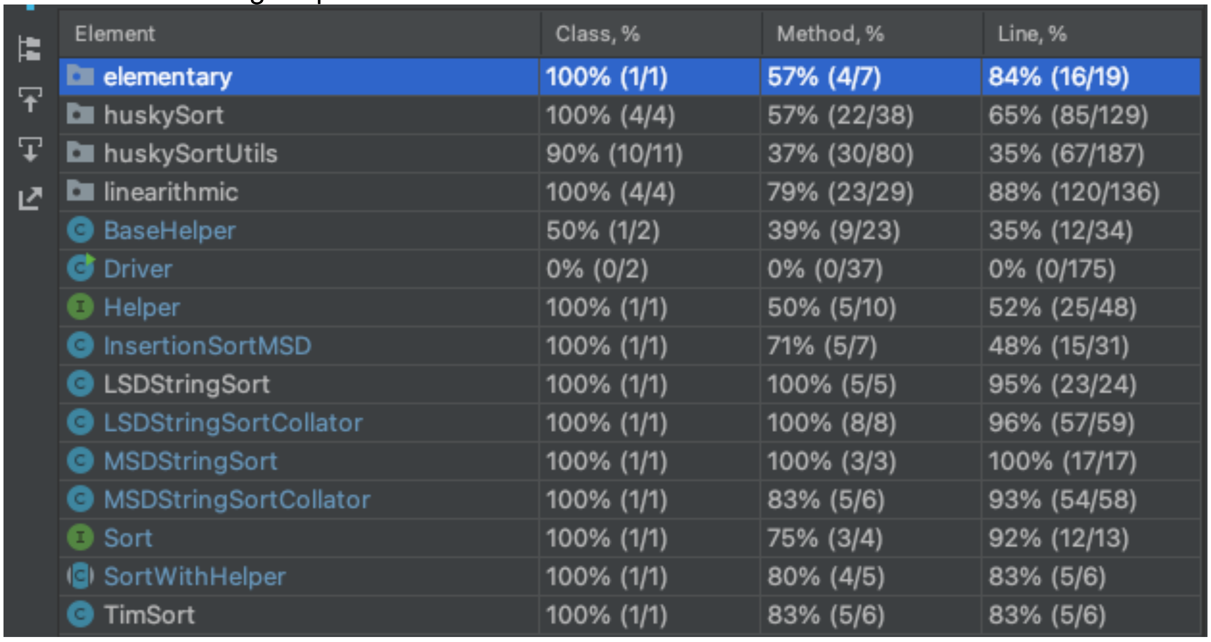


```
✓ Tests passed: 43 of 43 tests – 5 sec 38 ms

✓ info6205 (edu.neu.coe)                5 sec 38 ms
  ✓ InsertionSortMSDTest                12 ms
  ✓ LSDStringSortCollatorTest           741 ms
    ✓ testSortChineseIBMColl            508 ms
    ✓ sortRandom                        104 ms
    ✓ testSortInBuiltCollatorEng        57 ms
    ✓ testSortChineseInBuiltCol         72 ms
    ✓ sortInBuiltCollatorChinesel       0 ms
  ✓ LSDStringSortTest                   22 ms
  ✓ MSDStringSortCollatorTest           98 ms
```

```
/Library/Java/JavaVirtualMachines/jd
---- IntelliJ IDEA coverage runner --
sampling ...
include patterns:
edu\.neu\.coe\.info6205\..*
exclude patterns:
Input [154a, 127a, 46a, 168a, 15a, 7
Expected [110a, 122a, 127a, 12a, 147
[阿兵, 阿安, 阿冰冰, 阿斌, 阿滨, 阿冰, 阿
```

The code coverage report is shown below.



Element	Class, %	Method, %	Line, %
elementary	100% (1/1)	57% (4/7)	84% (16/19)
huskySort	100% (4/4)	57% (22/38)	65% (85/129)
huskySortUtils	90% (10/11)	37% (30/80)	35% (67/187)
linearithmic	100% (4/4)	79% (23/29)	88% (120/136)
BaseHelper	50% (1/2)	39% (9/23)	35% (12/34)
Driver	0% (0/2)	0% (0/37)	0% (0/175)
Helper	100% (1/1)	50% (5/10)	52% (25/48)
InsertionSortMSD	100% (1/1)	71% (5/7)	48% (15/31)
LSDStringSort	100% (1/1)	100% (5/5)	95% (23/24)
LSDStringSortCollator	100% (1/1)	100% (8/8)	96% (57/59)
MSDStringSort	100% (1/1)	100% (3/3)	100% (17/17)
MSDStringSortCollator	100% (1/1)	83% (5/6)	93% (54/58)
Sort	100% (1/1)	75% (3/4)	92% (12/13)
SortWithHelper	100% (1/1)	80% (4/5)	83% (5/6)
TimSort	100% (1/1)	83% (5/6)	83% (5/6)

Note: The classes which were custom implemented such as MSDStringSortCollator,LSDStringSortCollator etc have code coverage greater than 90% through unit tests.
The main method (Driver.java) has 0 % because that method is modified for running experiments and tuning parameters.

IBM Collator Error

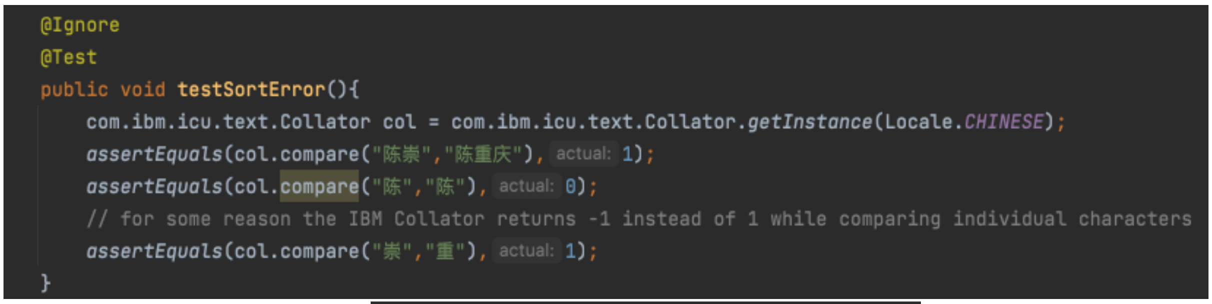
While testing our MSD Radix sort for Chinese characters one of our test cases failed. We were comparing the sorted outcome obtained from Arrays.Sort() with IBM Collator and with the sorted outcome from our MSD Radix Sort.

On analysis we found that the 2 strings, "陈重庆" and "陈崇", were being wrongly sorted by our MSD Radix Sort.

- As per Arrays.Sort() with IBM collator the sorted order is "陈重庆" and "陈崇"
- Our MSD Radix Sort() with IBM collator considers "陈崇" to be lesser than “陈重庆”

On further analysis we realized that it could be because in MSD Radix Sort we are actually comparing characters and not entire strings. So we ran the below simple test to verify our hypothesis.

- First, we asserted that "陈崇" to be greater than “陈重庆” using IBM Collator.
- Next, we compared the first characters of each string and we confirmed that they are the same.
- Based on the above two points, it is logical to conclude that “崇” is greater than “重”. As they are the second characters in both the strings and they are different.
- When we run the assertion that “崇” is greater than “重” it failed.



```
@Ignore
@Test
public void testSortError(){
    com.ibm.icu.text.Collator col = com.ibm.icu.text.Collator.getInstance(Locale.CHINESE);
    assertEquals(col.compare("陈崇","陈重庆"), actual: 1);
    assertEquals(col.compare("陈","陈"), actual: 0);
    // for some reason the IBM Collator returns -1 instead of 1 while comparing individual characters
    assertEquals(col.compare("崇","重"), actual: 1);
}
```

In the test below, the `assertEquals(col.compare("崇","重"),1);` *code fails*

We can safely assume that we found a potential bug in the IBM Collator.

Conclusion

- For sorting of english characters MSD Radix Sort proved to be the best. The implementation used for the MSD Radix Sort is the standard one that employs a count and aux array and the radix was set at 256.
- For sorting of chinese characters Pure Husky sort outperformed all other sorting methods. Our implementation of MSD Radix Sort is able to sort the input correctly but takes a considerable amount of time.

Github

- Repo Link

- Results Folder Contents
    - BenchmarkingResults.txt - file containing the benchmarking time measurements
    - SortedOutput Directory - Contains the sorted output of the first 1000 strings from shuffledChinese.txt file under RandomString