INFO 6205 Program Structures & Algorithms Summer 2022

Team Project, Team 4

Ching-Fong Chen, Kunlun Wang, Yi-Chen Yi

Task: Quicksort

Goal

Do you agree that the number of swaps in "standard" quicksort is $^1/_6$ times the number of comparisons? Is this figure correct? If not, why not. How do you explain it.

Sorting Mechanism

1. Implement the quick sort

To check if the number of swaps in the "standard" quicksort is 1⁄6 times the number of comparisons, we implemented the Quicksort in QuickSort.java.

2. Implemented the counting function

By using "huskyHelper" function, we designed the compare function avoiding having direct access to the long array.

3. Prevent from "long" array

We used the compare and swap functions which are set in AbstractHuskySort to count the number of comparisons and swaps.

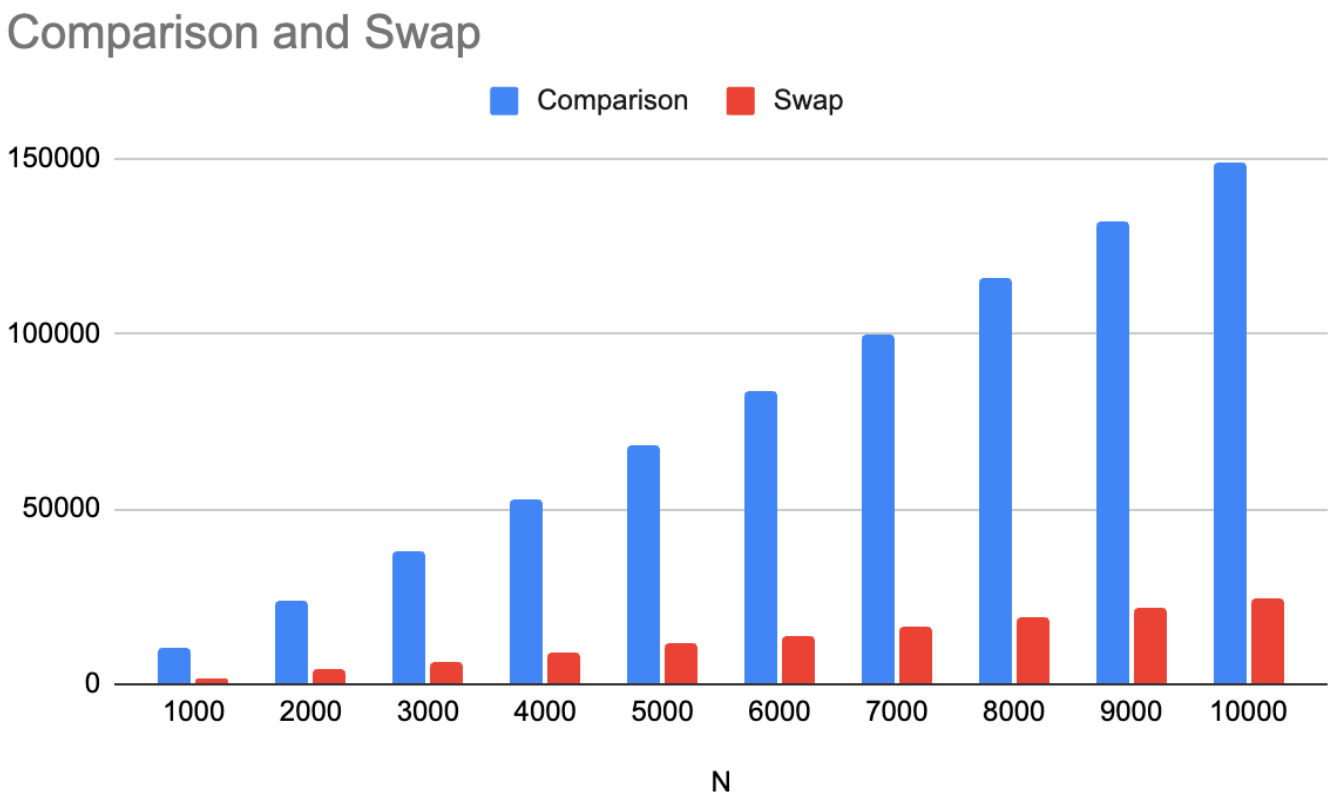
```
    private void quickSort(final X[] objects,  final int from, final int to) {
        final int lo = from;
        final int hi = to;
        if (hi <= lo) return;
        int j = partition(objects, lo, hi);
        quickSort(objects, lo,  to: j-1);
        quickSort(objects,  from: j+1, hi);
    }

    private int partition(final X[] objects,  final int lo, final int hi) {
        int i = lo;
        int j = hi;
        int pivot = lo;
        while (true) {
            //lo id the pivot
            // swap item on lo
            while (compare(objects, i++, pivot) < 0) {
                if (i == hi - 1) break;
            }
            // find item to swap
            while (compare(objects, pivot, --j) < 0) {
                if (j == lo) break;
            }
            // check if i and j cross
            if (i >= j) break;
            swap(objects, i, j);
        }
        // partitioning item lo at j
        swap(objects, lo, j);
        return j;
    }
}
```

Benchmarking

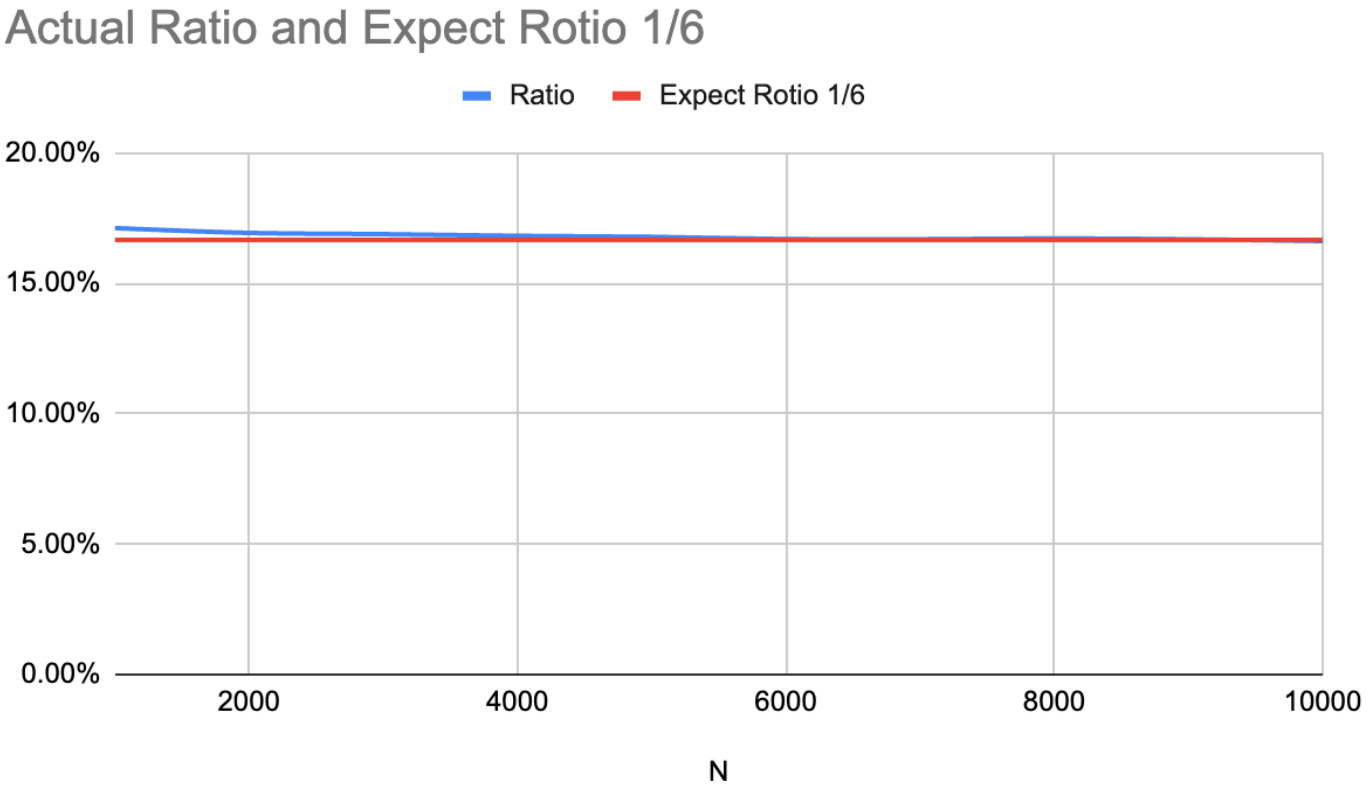
We increased the size of the array by 1000, the array size grows from 1000 to 10000, and we record the number of swaps and comparisons as below.

N	Comparison	Swap	Ratio	Expect Rotio 1/6
1000	10697.992	1831.33	17.12%	16.67%
2000	23946.597	4054.976	16.93%	16.67%
3000	38070.871	6428.64	16.89%	16.67%
4000	52838.089	8887.23	16.82%	16.67%
5000	68105.03	11429.458	16.78%	16.67%
6000	83885.401	14005.753	16.70%	16.67%
7000	99736.771	16646.591	16.69%	16.67%
8000	115650.936	19338.565	16.72%	16.67%
9000	132120.861	22048.208	16.69%	16.67%
10000	149114.727	24784.314	16.62%	16.67%

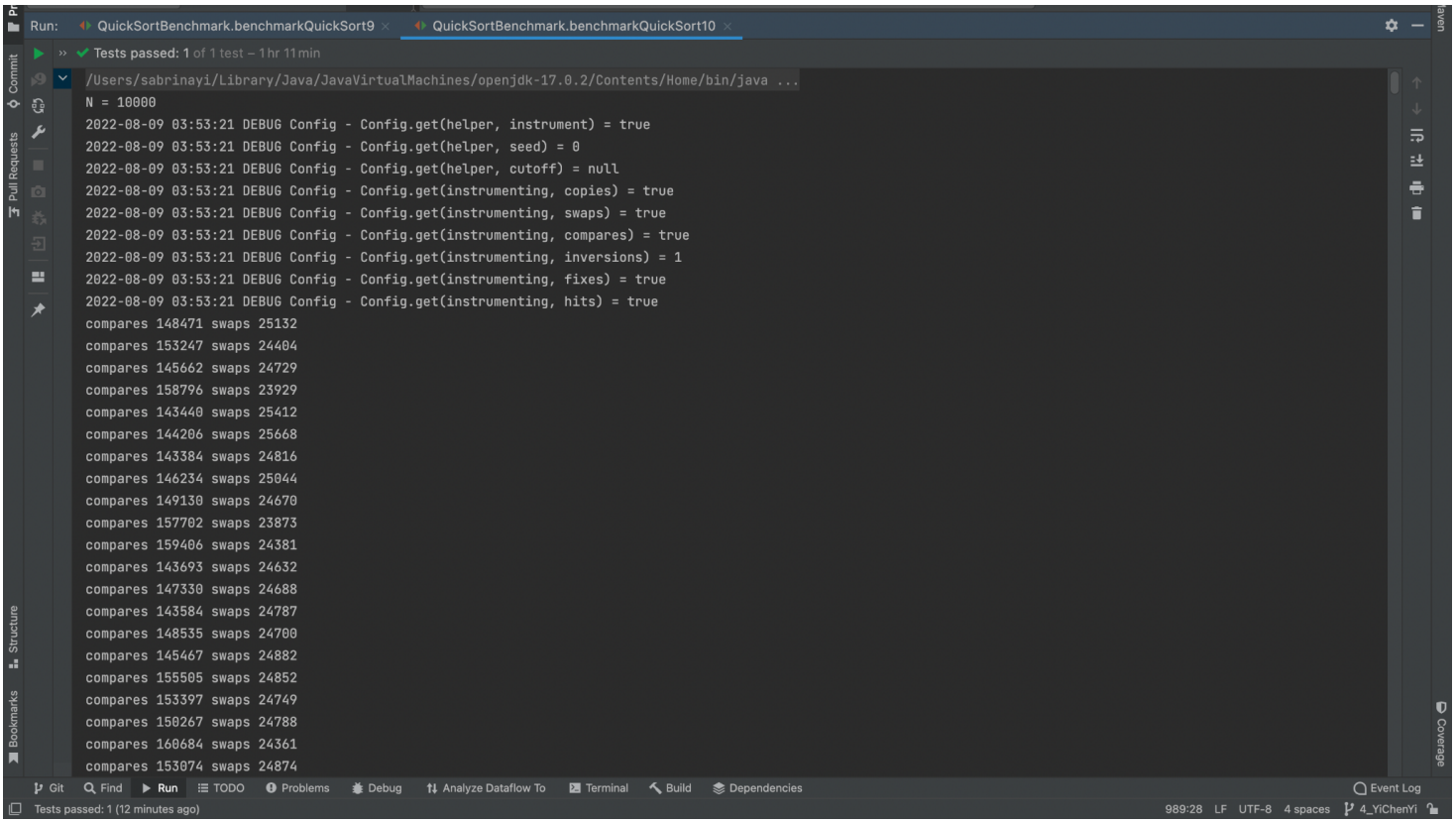
The result of compares and swaps in different array size



The relationship between compares and swaps



The ratio of compares and swaps



The screenshot of the benchmark log

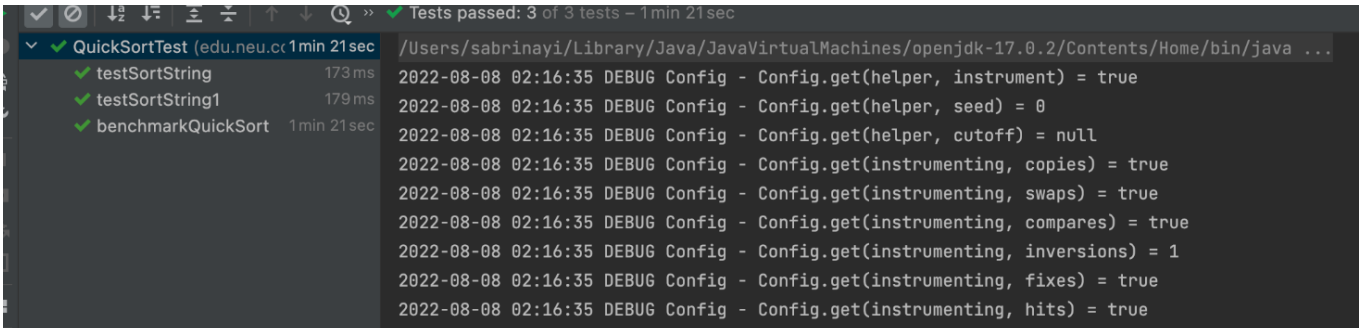
Conclusions

Yes. I agree that the number of swaps in “standard” quicksort is $^{1}/_{6}$ times the number of comparisons.

First, according to the result of the benchmark, the ratio between compare and swap is near $^{1}/_{6}$.

Besides the result of coding, I found a handout from the University of Illinois Chicago[1], which provided the math calculation of why the number of swaps in quicksort is 1/6 times the number of comparisons.

Unit Tests Result



Task: Hibbard deletion of BST

Goal

The main goal is to find the average height of the binary search tree after a number of Hibbard deletion processes. Compare to the original version of deletion, different preferences of deletion could affect the results as well. In short, we want to find an optimized solution following the Arbitrary Substitution Principle. After the sufficient benchmarks and running repeats, we disagree the average height of the tree is $\sqrt{n}$ after a number of deletions have been made.

Mechanism

4. **Insert node into a binary search tree**

First of all, we perform random insertions in which the node is drawn from a relatively huge domain. We select a random integer from 0 to 10,000,000 as node's value and convert the generated number into the String format as node's key. In order to make successfully node comparisons in both insertion and deletion, we align the length of the strings by adding extra leading zeros of possible maximum digits so that it won't cause issues. For example, we generate integer 412,146 as the root node, and 83,144 is drawn afterward. The number latter should be formatted to "00083144", otherwise, "412146" will be smaller than "83144" due to Unicode values ordering. We use a String type ArrayList to record the existing nodes of the binary search tree. We add a node (in a String format) only when the generated integer still does not exist in the tree to prevent uneven-sized binary search trees.

5. **Deletion of the binary search tree**
    a. **Original Hibbard deletion**

Unlike the insertion process, we choose a random index from the ArrayList to delete an existing node from the binary search tree. In addition, a random node will add to the tree as same as the insertion process after a node was deleted. We consider a long sequence of insertion and deletion while the total number of tree nodes is constant.

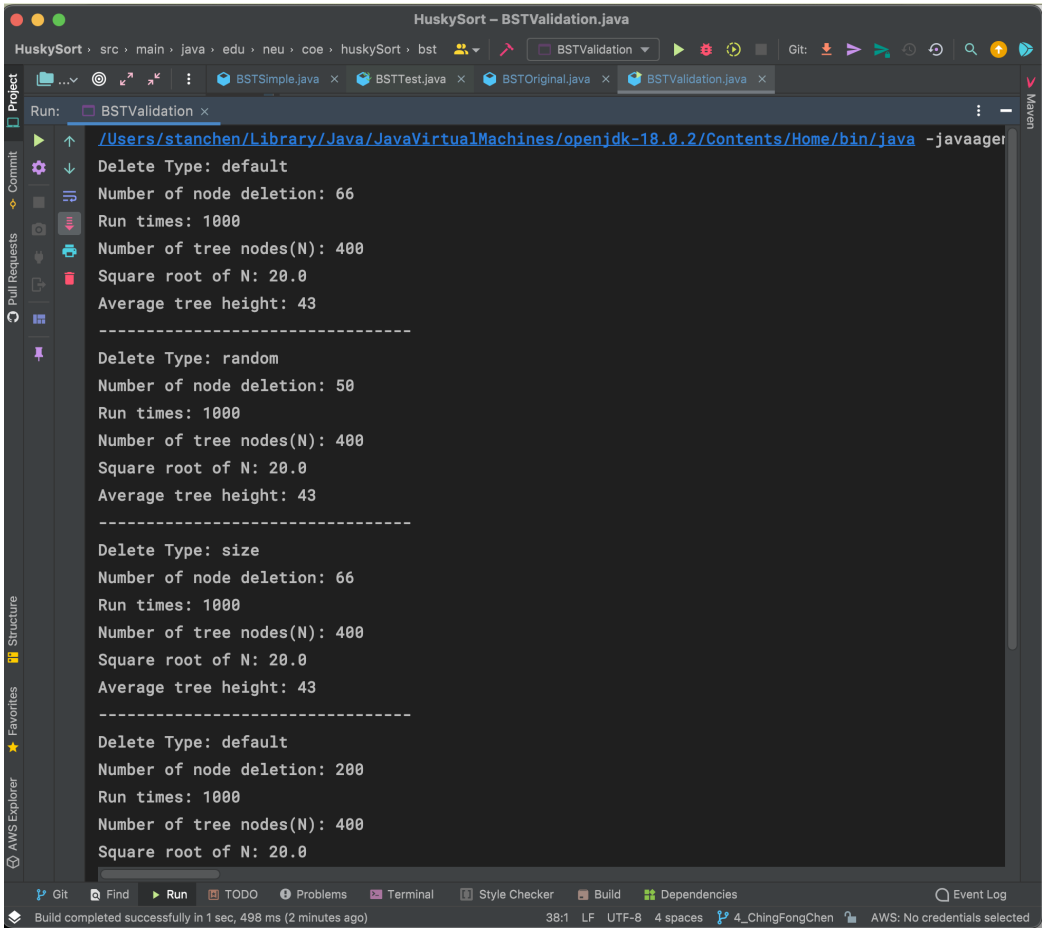
    b. **Randomly choose the direction to look for the node to be deleted**

Instead of traditional deletion, it has an equal chance to look for nodes in either direction. For the smaller or lefthand side, the node deletion logic will be opposite toward the original Hibbard deletion which always starts from the larger or righthand side.

    c. **Choose the direction according to the size of the candidate nodes**

If one subtree's size is larger than the other one, we look up from the former direction to keep the tree more balanced. We choose a random direction if the tree is currently balanced.

Code Snippets



Binary search tree deletion result

Benchmarking

The tree height is different for corresponding number of nodes, number of node deletion ratio, and types of deletion (Hibbard, random direction, and subtree size direction). We set the tree size from 25 to 6,400 and use doubling method for each benchmarks. The following table displays the average tree heights after deletion running 1,000 times for each tree size.

# of Nodes(N)	# of Deletion	Square root of N	Hibbard Deletion Tree Height	Random Deletion Tree Height	Size Deletion Tree Height
25	N/10	5.00	7.4	7.1	7.5
50	N/10	7.07	9.5	9.1	10
100	N/10	10.00	12.3	12.1	12.3
200	N/10	14.14	15.1	14.3	14.7
400	N/10	20.00	17.5	16.2	17
800	N/10	28.28	20.2	20	20.2
1600	N/10	40.00	21.7	20.3	23.4
3200	N/10	56.57	25.3	24.4	25.5
6400	N/10	80.00	28.3	27.3	29.6

Running result of three types of deletion with deletion size ratio = N/10

We wonder how the numbers or ratios of deletion will affect the tree height after deletion thus we examine other ratio of deletion. But the ratio doesn't affect the tree height significantly in the long sequence of deletion and node put back process.

# of Nodes(N)	# of Deletion	Square root of N	Hibbard Deletion Tree Height	Random Deletion Tree Height	Size Deletion Tree Height
25	N/6	5.00	7.7	7.2	7.4
50	N/6	7.07	9.6	9.1	9.7
100	N/6	10.00	11.4	10.6	11.6
200	N/6	14.14	14.3	13.5	14.2
400	N/6	20.00	17.3	16.4	18.1
800	N/6	28.28	19.9	18.7	20.2
1600	N/6	40.00	23	21.3	22.6
3200	N/6	56.57	24.3	22.4	25.6
6400	N/6	80.00	27.5	26.1	27.6

Running result of three types of deletion with deletion size ratio = N/6

# of Nodes(N)	# of Deletion	Square root of N	Hibbard Deletion Tree Height	Random Deletion Tree Height	Size Deletion Tree Height
25	N/2	5.00	7	6.5	7.2
50	N/2	7.07	9.1	9	9.1
100	N/2	10.00	11.9	11.4	12.2
200	N/2	14.14	14.15	13.7	14.2
400	N/2	20.00	17	16.3	17.7
800	N/2	28.28	19.7	19.1	19.6
1600	N/2	40.00	21.65	20.8	22.7
3200	N/2	56.57	25.1	24.2	24.8
6400	N/2	80.00	28.3	25.9	28.4

Running result of three types of deletion with deletion size ratio = N/2

The statistics above indicates the deletion ratio wouldn't affect the tree height significantly. To be more specific, the average tree heights from each type of deletion increared or decreased less than 3% compared to the other ratio of deletion. We choose the N/10 as ratio of deletion and the following tables display the tree heights comparison, the height growth trends, and scales of three deletion approaches.

Tree height distribution of three types of deletion with total tree node size from 25 to 400

We observe that the average heights of the tree are close to the theoretical value $\sqrt{n}$ (as the blue line)when the tree is relatively small. But we didn't ovserve the expected average heights of tree when the tree become bigger. The following diagram shows different scales on the tree heights growth.

Tree height distribution of three types of deletion with total tree node size from 25 to 6,400

Conclusions

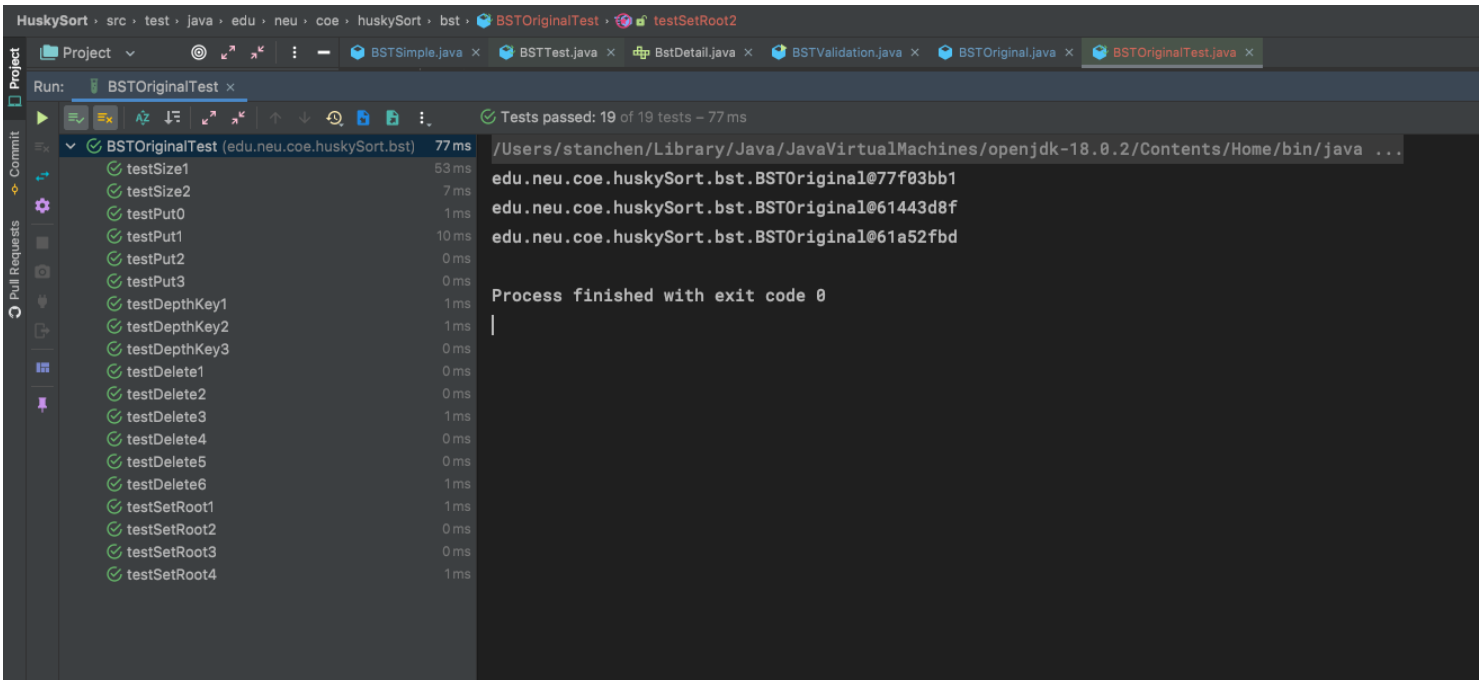
Overall, we disagree the theoretical value that the average height of the tree is $\sqrt{n}$ after a number of deletions have been made. In our observation, we discover that the average tree heights of three types of deletion converge in similar value as the tree size growing. However, these tree heights gradully get further and further away from the ideal value. Even the results aren't as expected, we still observe that the approach to random choose from either direaction has the best effect to minimize the height of tree after deletion. It approximately reduce 3 to 9 percent of average tree height from the original Hibbard deletion. The reason is it won't cause biased behavior in the long sequence of dynamic insertion and deletion process. The Arbitrary Substitution Principle helps us to delete the nodes in unbiased behavior.

The reason that why the results are not expected as the theoretical value may happens in random putting nodes and the add-one-node process after deletion comparing to traditional Hibbard deletion experiments. Additional researches and observations would improve the result.

Unit Tests

We add new functions in BSTOriginal.java as well as the related utility functions from INFO6205 repository and Robert Sedgewick's implementation of binary search tree. These unit tests covers more new features in put, delete, and size methods than BSTTest.java unit test in INFO6205 repository. The unit tests run successfully as the figure

displayed below.



Unit tests running results of binary search tree functions

Task: MSD Radix Sort

1. **Improve Cutoff**
2. **Implement Bopomofo**

Part 1- Improve Cutoff

Goal

Find the best value of cutoff.

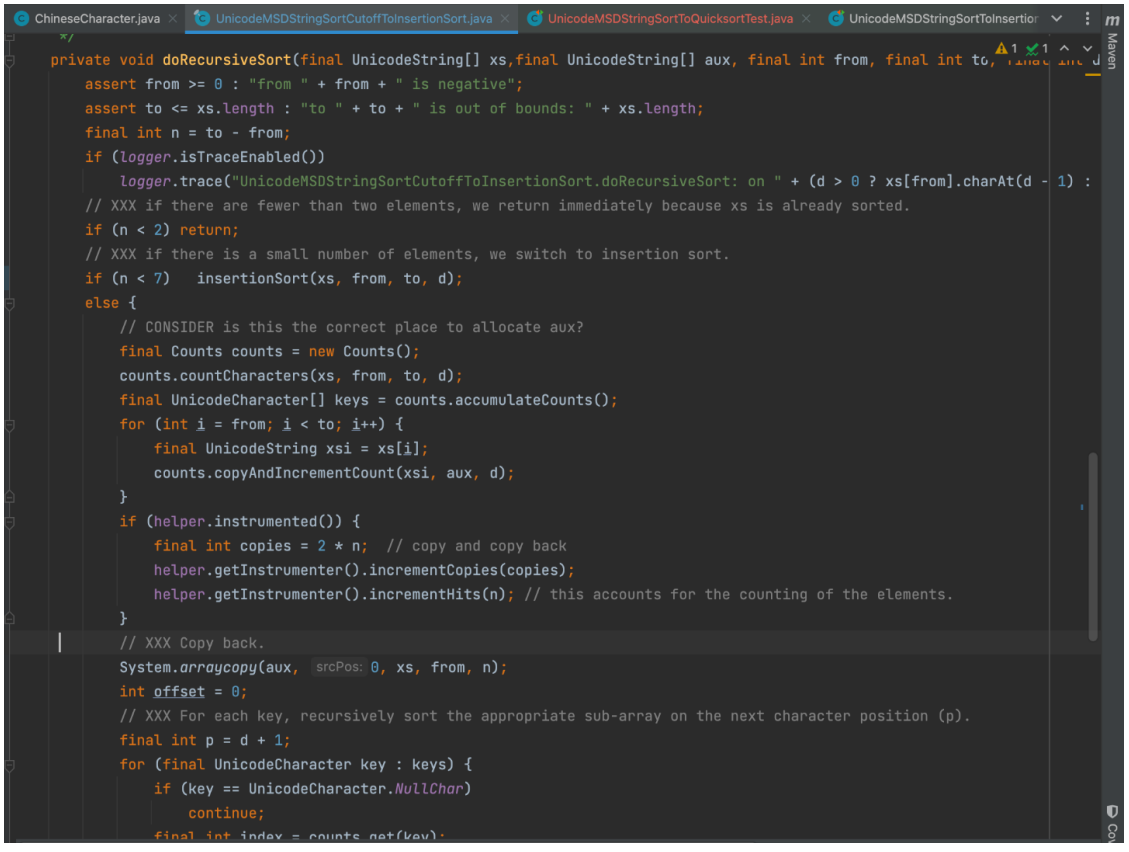
Sorting Mechanism

1. Implement change in the sorting function

To find the proper value for the cutoff, we wrote a “UnicodeMSDStringSortCutoffToInsertionSort.java”. In this file, we push the sorting into insertion sort when n is less than a particular number. And we modified the original “UnicodeMSDStringSort.java”, and removed the insertion sort part, to make it a control group.

2. Compare the performance

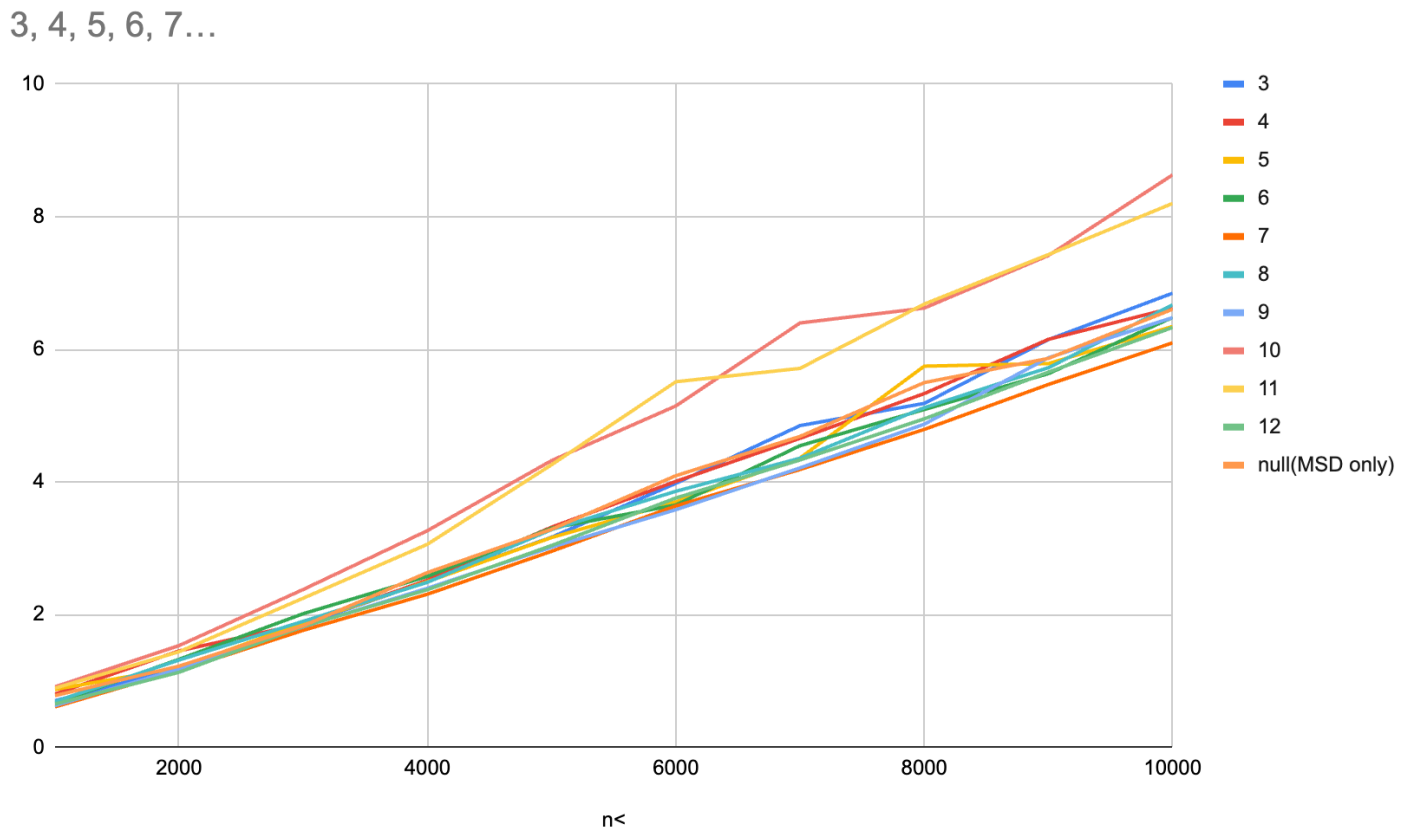
Compare the performance of no cutoff and cutoff from 3 to 12. To inspect if the result would be impacted by the array size, we increased the size of the array from 1000 to 10000 and run the test 1000 times.



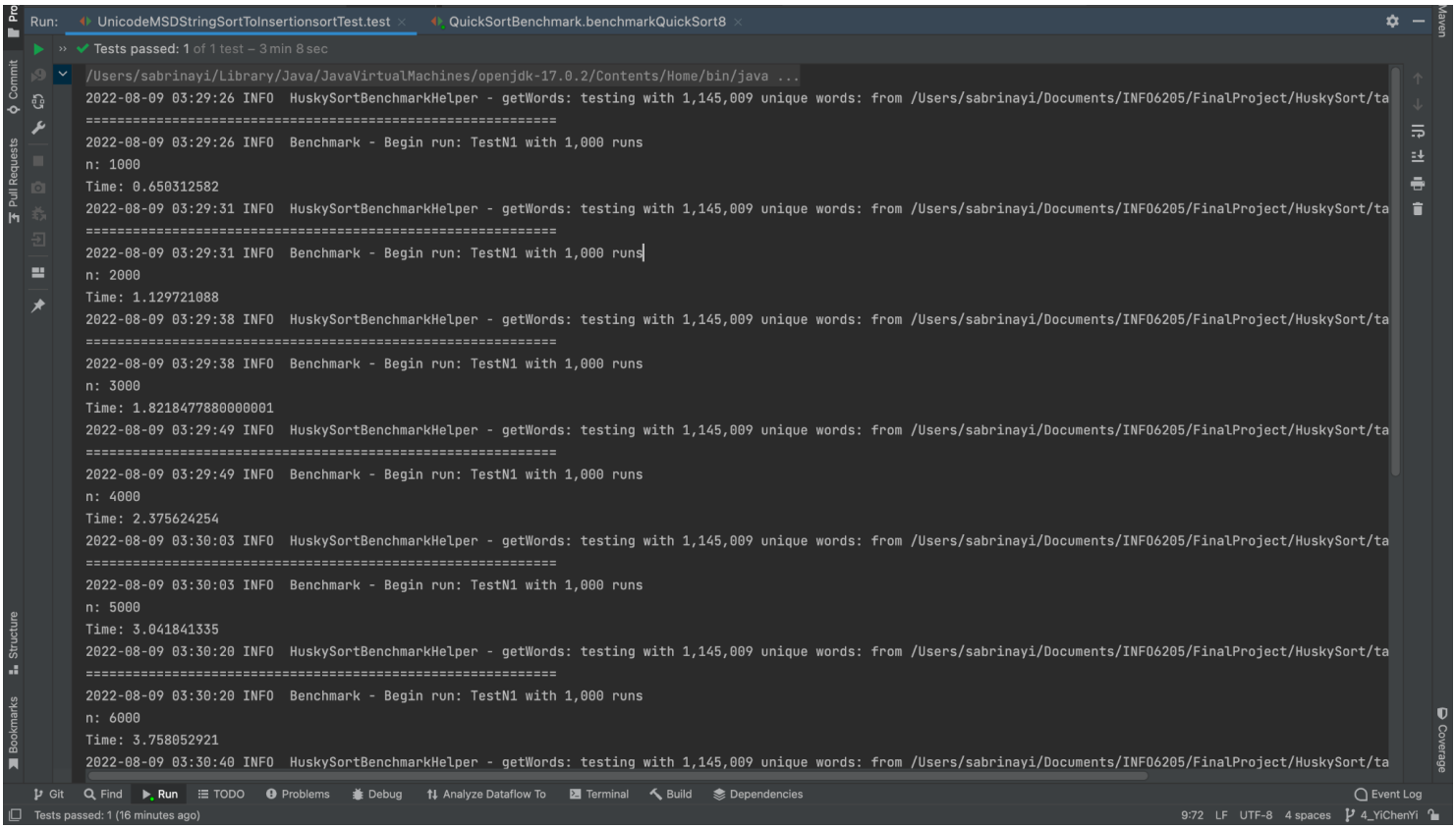
Benchmarking

to - from <	3	4	5	6	7	8	9	10	11	12	null(MSD only)
1000	0.700856232	0.801195419	0.870309867	0.664550698	0.607206842	0.688115924	0.626657446	0.910545218	0.889744963	0.650312582	0.77718728
2000	1.187768257	1.452264381	1.200309833	1.325237215	1.156520497	1.314047429	1.17920442	1.5323302	1.439190249	1.129721088	1.22253792
3000	1.850815038	1.863031921	1.870866034	2.012988161	1.762757582	1.898381878	1.815244099	2.378386748	2.247249803	1.821847788	1.830590069
4000	2.501790874	2.518824747	2.496443925	2.575406941	2.307434636	2.487085098	2.397657228	3.266050034	3.060312448	2.375624254	2.633884383
5000	3.167390254	3.318683694	3.162054682	3.303633551	2.954420494	3.281104168	3.016662569	4.323091633	4.260175956	3.041841335	3.286222209
6000	3.97909236	4.008720128	3.698306185	3.652303963	3.639144128	3.859337231	3.581664052	5.146134784	5.511614423	3.758052921	4.09420748
7000	4.849982256	4.658007544	4.361288126	4.546811208	4.187571668	4.358198826	4.212139405	6.397962969	5.713097562	4.331621917	4.687050855
8000	5.184795346	5.332971545	5.746501505	5.092590052	4.792111473	5.120149493	4.871447076	6.622232677	6.683014802	4.951691421	5.498247638
9000	6.146714466	6.151224005	5.785331828	5.633760869	5.470572443	5.723836521	5.87453602	7.415183715	7.429633385	5.656918938	5.863887416
10000	6.847111157	6.625434481	6.345315205	6.481606866	6.0979516	6.670367745	6.479245618	8.631863915	8.201064015	6.330007274	6.604884387

The result of the benchmark



The relationship between the different cutoff choice

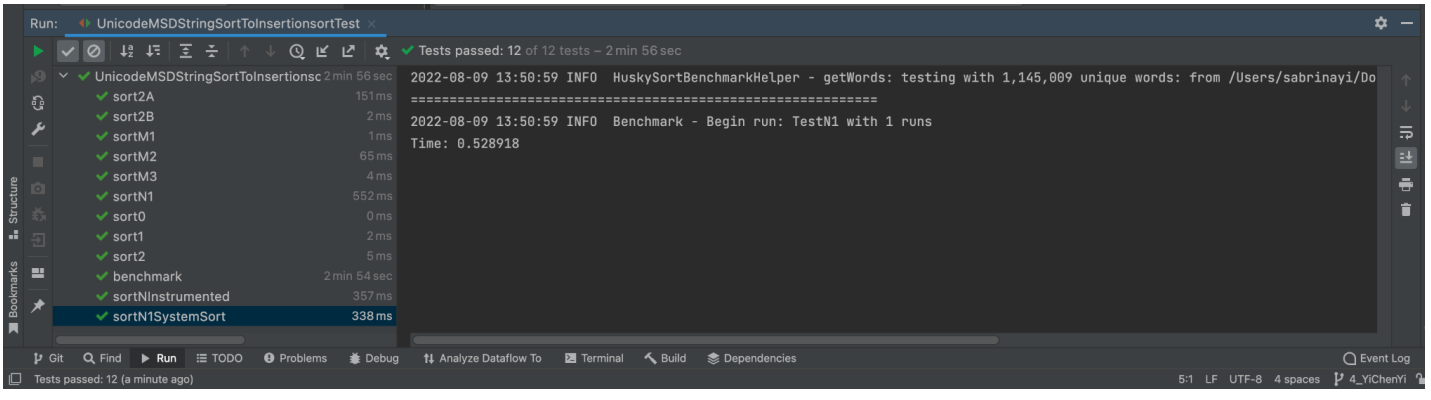


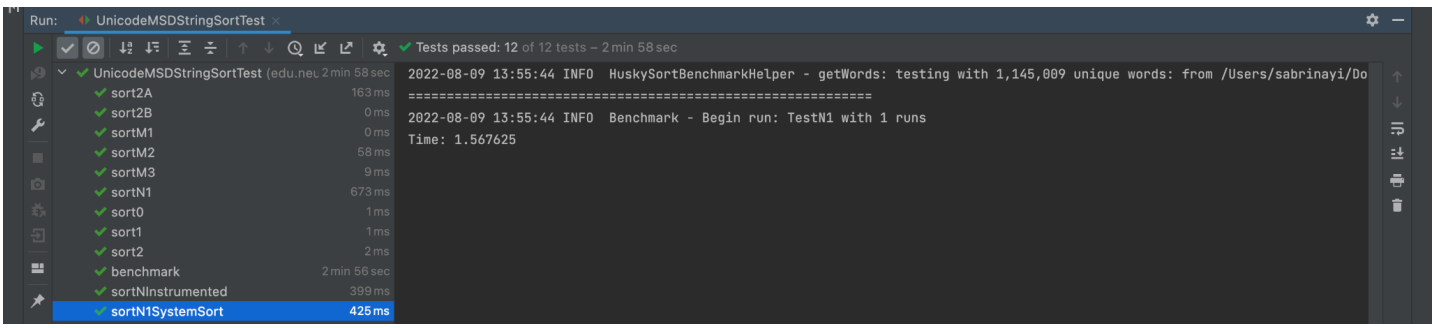
The screenshot of the benchmark log

Conclusions

According to the inspection of the test result, when we set the cutoff to less than 10 or 11, the result is worse. If we set the cutoff to less than 7, the result will be stable and better than other numbers. Thus, if we set the cutoff as n < 7, changing the sort from MSD sort to insertion sort can improve the performance.

Unit Tests Result





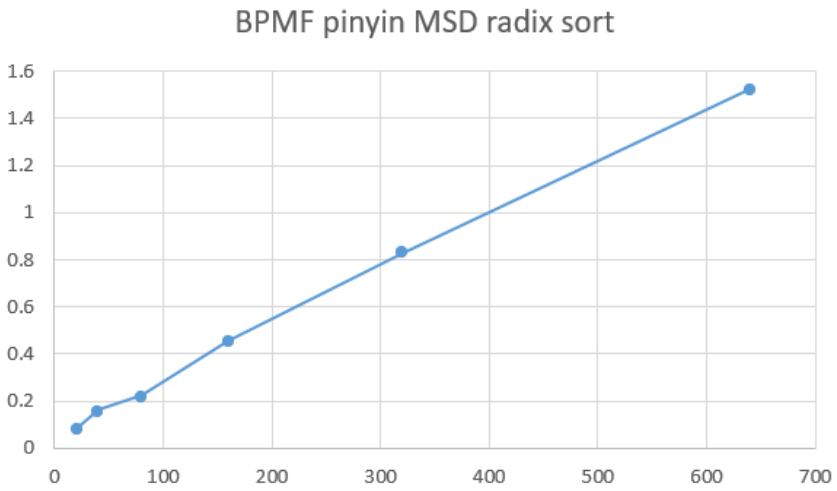
Part 2

Sorting Mechanism

For each Chinese character converted into BPMF pinyin has four parts: initial medial, final sound and tone and each part has different pinyin order. So, we use four Hashmaps to build this pinyin order logic. In this case, we can use a smaller array to count the frequencies, this saves space. For each Chinese character can be converted to the same length pinyin, then it is very easy to use MSD sort on it. And we also use Hashmap to build the relationship between Chinese characters and pinyin. For those repeating Chinese characters or Chinese characters with similar pinyin, we add “” behind them to distinguish to make sure we have corrected the length Character string after sorting.

Benchmarking

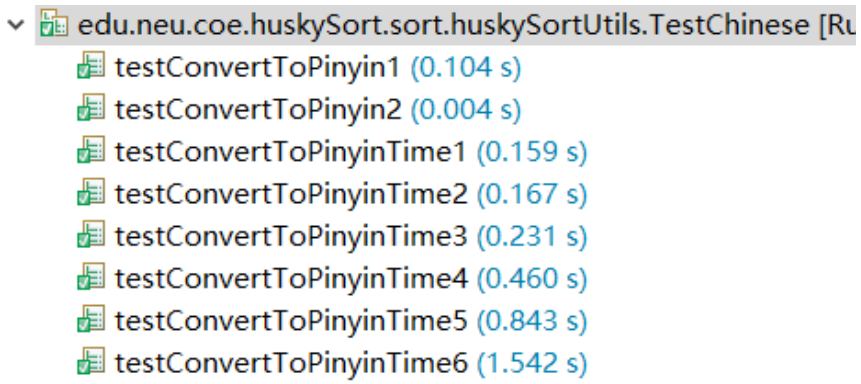
```
============================================================
2022-08-09 14:23:56 INFO  Benchmark - Begin run: result with 1,000 runs
............................................................
Sort 20 Chinese characters: 0.08060500000000001 ms
============================================================
2022-08-09 14:23:56 INFO  Benchmark - Begin run: result with 1,000 runs
............................................................
Sort 40 Chinese characters: 0.16108039999999998 ms
============================================================
2022-08-09 14:23:56 INFO  Benchmark - Begin run: result with 1,000 runs
............................................................
Sort 80 Chinese characters: 0.2246184 ms
============================================================
2022-08-09 14:23:56 INFO  Benchmark - Begin run: result with 1,000 runs
............................................................
Sort 160 Chinese characters: 0.4531062 ms
============================================================
2022-08-09 14:23:57 INFO  Benchmark - Begin run: result with 1,000 runs
............................................................
Sort 320 Chinese characters: 0.8291914 ms
============================================================
2022-08-09 14:23:58 INFO  Benchmark - Begin run: result with 1,000 runs
............................................................
Sort 20 Chinese characters: 1.5237415 ms
```

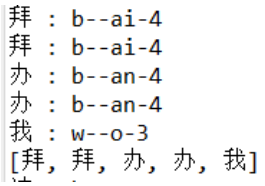


Conclusions

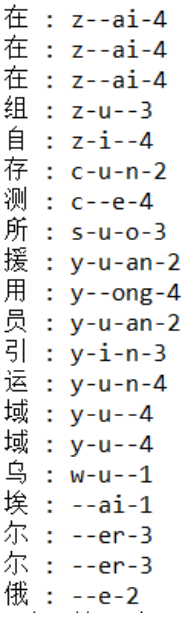
Using BPMF pinyin as a sorting order can save space for counting frequencies in MSD radix sorting.

Unit Tests Result





short array and sorted result



[波, 棒, 波, 不, 炮, 炮, 炮, 片, 米, 风, 发, 乏, 辐, 附, 动, 电, 电, 电, 弹, 的, 的, 的, 的, 达, 队, 对, 涅, 兰, 凌, 料, 了, 了

long array and sorted result

Task: General Benchmarks

Goal

To find the extent that execution time can be predicted based solely on the number of array accesses?

Conclusions

As the number of array accesses increases, there is a linear relationship between the number of array accesses and time. When the number of array accesses is greater than 20,000 to 40,000, the execution time can be predicted solely using the number of array accesses

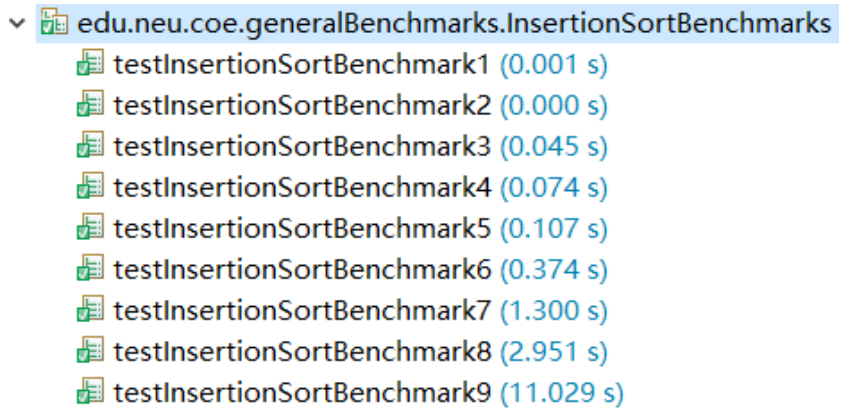
Benchmarking and Unit Tests Result

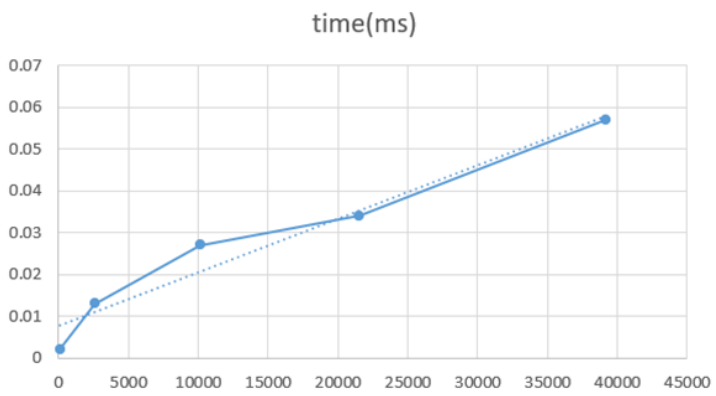
1.Insertion Sort

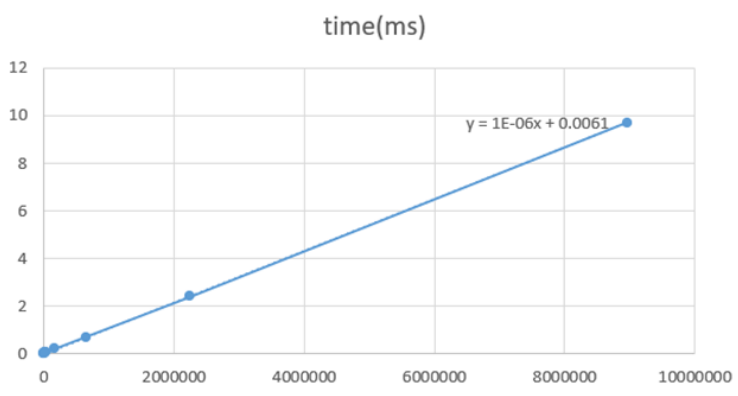
Benchmark:

```
------------------------------------------------------------
2022-07-31 15:50:44 INFO  Benchmark - Begin run: Insertion sort has 114 hits for 10 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:44 INFO  InsertionSortBenchmarks - 0.002 ms
============================================================
2022-07-31 15:50:44 INFO  Benchmark - Begin run: Insertion sort has 2604 hits for 50 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:44 INFO  InsertionSortBenchmarks - 0.013 ms
============================================================
2022-07-31 15:50:44 INFO  Benchmark - Begin run: Insertion sort has 10162 hits for 100 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:44 INFO  InsertionSortBenchmarks - 0.027 ms
============================================================
2022-07-31 15:50:44 INFO  Benchmark - Begin run: Insertion sort has 21544 hits for 150 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:44 INFO  InsertionSortBenchmarks - 0.034 ms
============================================================
2022-07-31 15:50:44 INFO  Benchmark - Begin run: Insertion sort has 39190 hits for 200 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:44 INFO  InsertionSortBenchmarks - 0.057 ms
============================================================
2022-07-31 15:50:44 INFO  Benchmark - Begin run: Insertion sort has 159478 hits for 400 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:44 INFO  InsertionSortBenchmarks - 0.199 ms
============================================================
2022-07-31 15:50:44 INFO  Benchmark - Begin run: Insertion sort has 650736 hits for 800 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:45 INFO  InsertionSortBenchmarks - 0.680 ms
============================================================
2022-07-31 15:50:45 INFO  Benchmark - Begin run: Insertion sort has 2248986 hits for 1500 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:47 INFO  InsertionSortBenchmarks - 2.414 ms
============================================================
2022-07-31 15:50:47 INFO  Benchmark - Begin run: Insertion sort has 8960430 hits for 3000 Integers with 1,000 runs
.....................................................
2022-07-31 15:50:57 INFO  InsertionSortBenchmarks - 9.694 ms
```

Unit test:







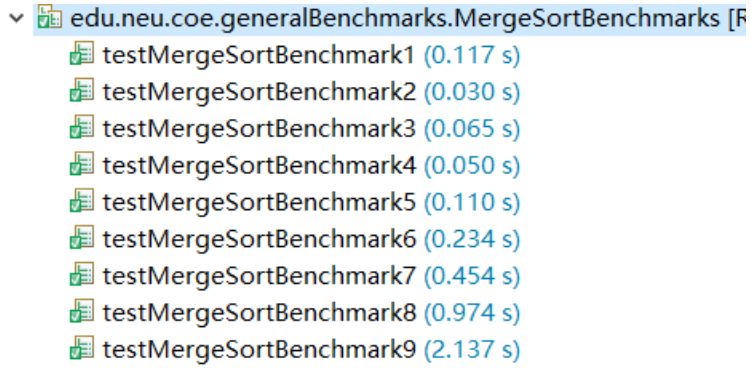
The time has a linear relationship with the number of array accesses. The larger the number of array accesses, the more accurate it is. From about 20,000 array accesses, the time can be solely predicted by the number of array accesses.

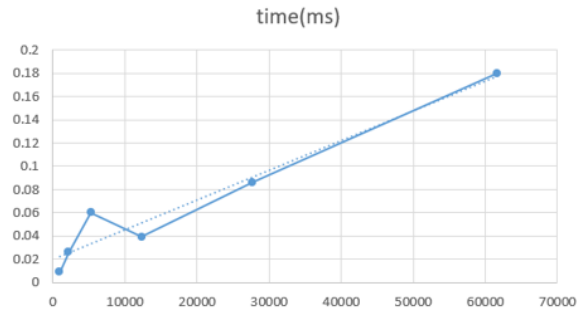
2.Merge Sort

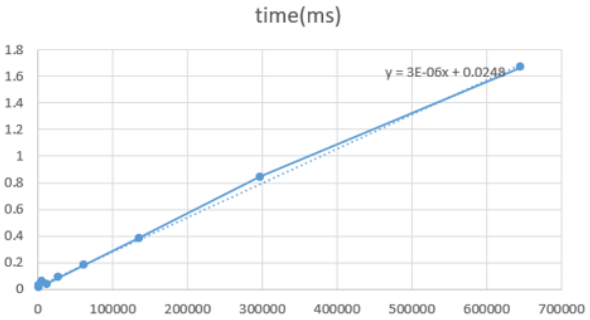
Benchmark:

```
2022-07-31 16:13:05 INFO  Benchmark - Begin run: Merge sort has 938 hits for 50 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:05 INFO  InsertionSortBenchmarks - 0.009 ms
=============================================================
2022-07-31 16:13:05 INFO  Benchmark - Begin run: Merge sort has 2218 hits for 100 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:05 INFO  InsertionSortBenchmarks - 0.026 ms
=============================================================
2022-07-31 16:13:05 INFO  Benchmark - Begin run: Merge sort has 5318 hits for 200 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:05 INFO  InsertionSortBenchmarks - 0.060 ms
=============================================================
2022-07-31 16:13:05 INFO  Benchmark - Begin run: Merge sort has 12366 hits for 400 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:05 INFO  InsertionSortBenchmarks - 0.039 ms
=============================================================
2022-07-31 16:13:05 INFO  Benchmark - Begin run: Merge sort has 27740 hits for 800 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:05 INFO  InsertionSortBenchmarks - 0.086 ms
=============================================================
2022-07-31 16:13:05 INFO  Benchmark - Begin run: Merge sort has 61676 hits for 1600 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:05 INFO  InsertionSortBenchmarks - 0.180 ms
=============================================================
2022-07-31 16:13:05 INFO  Benchmark - Begin run: Merge sort has 135586 hits for 3200 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:06 INFO  InsertionSortBenchmarks - 0.382 ms
=============================================================
2022-07-31 16:13:06 INFO  Benchmark - Begin run: Merge sort has 296886 hits for 6400 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:07 INFO  InsertionSortBenchmarks - 0.843 ms
=============================================================
2022-07-31 16:13:07 INFO  Benchmark - Begin run: Merge sort has 645626 hits for 12800 Integers with 1,000 runs
.............................................................
2022-07-31 16:13:09 INFO  InsertionSortBenchmarks - 1.670 ms
```

Unit test:







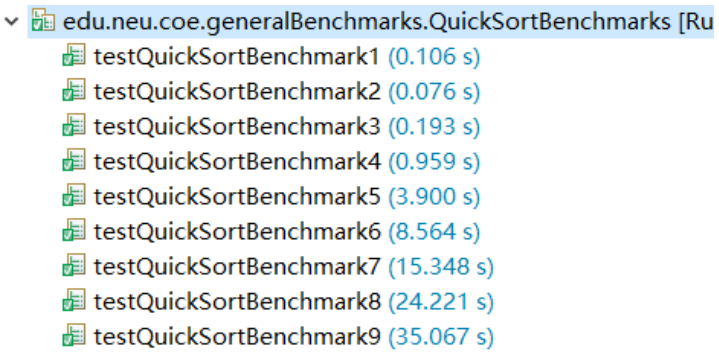
The time has a linear relationship with the number of array accesses. The larger the number of array accesses, the more accurate it is. From about 20,000 array accesses, the time can be solely predicted by the number of array accesses.

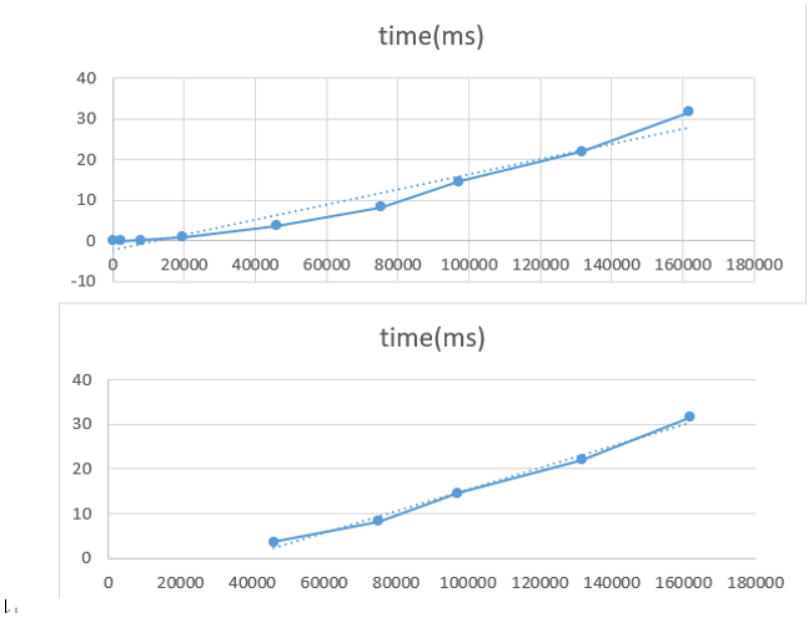
3.Quick Sort

Benchmark:

```
------------------------------------------------------------
2022-07-31 16:47:19 INFO  Benchmark - Begin run: Qucik sort has 78 hits for 10 Integers with 1,000 runs
.....................................................
2022-07-31 16:47:19 INFO  InsertionSortBenchmarks - 0.004 ms
============================================================
2022-07-31 16:47:19 INFO  Benchmark - Begin run: Qucik sort has 2352 hits for 100 Integers with 1,000 runs
.....................................................
2022-07-31 16:47:19 INFO  InsertionSortBenchmarks - 0.070 ms
============================================================
2022-07-31 16:47:19 INFO  Benchmark - Begin run: Qucik sort has 7736 hits for 200 Integers with 1,000 runs
.....................................................
2022-07-31 16:47:19 INFO  InsertionSortBenchmarks - 0.181 ms
============================================================
2022-07-31 16:47:19 INFO  Benchmark - Begin run: Qucik sort has 19662 hits for 500 Integers with 1,000 runs
.....................................................
2022-07-31 16:47:20 INFO  InsertionSortBenchmarks - 0.905 ms
============================================================
2022-07-31 16:47:20 INFO  Benchmark - Begin run: Qucik sort has 45958 hits for 1000 Integers with 1,000 runs
.....................................................
2022-07-31 16:47:24 INFO  InsertionSortBenchmarks - 3.586 ms
============================================================
2022-07-31 16:47:24 INFO  Benchmark - Begin run: Qucik sort has 75320 hits for 1500 Integers with 1,000 runs
.....................................................
2022-07-31 16:47:32 INFO  InsertionSortBenchmarks - 8.036 ms
============================================================
2022-07-31 16:47:32 INFO  Benchmark - Begin run: Qucik sort has 97224 hits for 2000 Integers with 1,000 runs
.....................................................
2022-07-31 16:47:46 INFO  InsertionSortBenchmarks - 14.172 ms
============================================================
2022-07-31 16:47:46 INFO  Benchmark - Begin run: Qucik sort has 131814 hits for 2500 Integers with 1,000 runs
.....................................................
2022-07-31 16:48:09 INFO  InsertionSortBenchmarks - 22.065 ms
============================================================
2022-07-31 16:48:09 INFO  Benchmark - Begin run: Qucik sort has 161882 hits for 3000 Integers with 1,000 runs
.....................................................
2022-07-31 16:48:41 INFO  InsertionSortBenchmarks - 31.642 ms
```

Unit test:





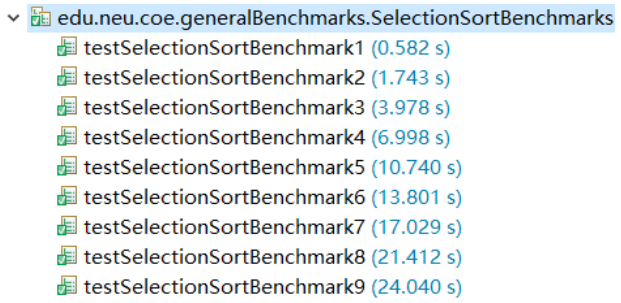
The time has a linear relationship with the number of array accesses. The larger the number of array accesses, the more accurate it is. From about 40,000 array accesses, the time can be solely predicted by the number of array accesses.

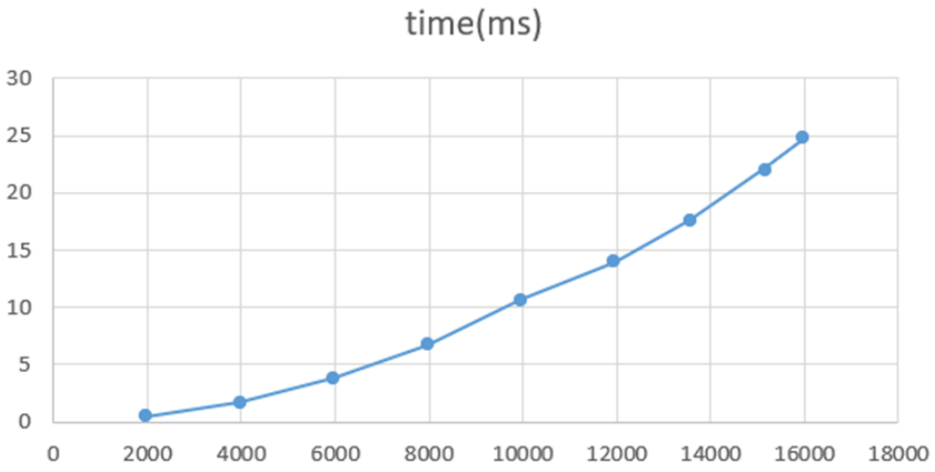
4.Selection Sort

Benchmark:

```
2022-07-31 17:52:10 INFO  Benchmark - Begin run: Selection sort has 1968 hits for 500 Integers with 1,000 runs
.............................................................
2022-07-31 17:52:10 INFO  InsertionSortBenchmarks - 0.440 ms
=============================================================
2022-07-31 17:52:11 INFO  Benchmark - Begin run: Selection sort has 3988 hits for 1000 Integers with 1,000 runs
.............................................................
2022-07-31 17:52:12 INFO  InsertionSortBenchmarks - 1.694 ms
=============================================================
2022-07-31 17:52:12 INFO  Benchmark - Begin run: Selection sort has 5976 hits for 1500 Integers with 1,000 runs
.............................................................
2022-07-31 17:52:16 INFO  InsertionSortBenchmarks - 3.838 ms
=============================================================
2022-07-31 17:52:16 INFO  Benchmark - Begin run: Selection sort has 7976 hits for 2000 Integers with 1,000 runs
.............................................................
2022-07-31 17:52:23 INFO  InsertionSortBenchmarks - 6.737 ms
=============================================================
2022-07-31 17:52:23 INFO  Benchmark - Begin run: Selection sort has 9952 hits for 2500 Integers with 1,000 runs
.............................................................
2022-07-31 17:52:34 INFO  InsertionSortBenchmarks - 10.636 ms
=============================================================
2022-07-31 17:52:34 INFO  Benchmark - Begin run: Selection sort has 11956 hits for 3000 Integers with 1,000 runs
.............................................................
2022-07-31 17:52:48 INFO  InsertionSortBenchmarks - 13.978 ms
=============================================================
2022-07-31 17:52:48 INFO  Benchmark - Begin run: Selection sort has 13560 hits for 3400 Integers with 1,000 runs
.............................................................
2022-07-31 17:53:06 INFO  InsertionSortBenchmarks - 17.692 ms
=============================================================
2022-07-31 17:53:06 INFO  Benchmark - Begin run: Selection sort has 15164 hits for 3800 Integers with 1,000 runs
.............................................................
2022-07-31 17:53:28 INFO  InsertionSortBenchmarks - 22.152 ms
=============================================================
2022-07-31 17:53:28 INFO  Benchmark - Begin run: Selection sort has 15980 hits for 4000 Integers with 1,000 runs
.............................................................
2022-07-31 17:53:53 INFO  InsertionSortBenchmarks - 24.755 ms
```

Unit test:





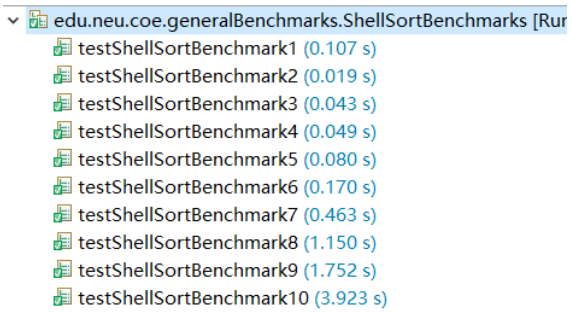
The time has a linear relationship with the number of array accesses. The larger the number of array accesses, the more accurate it is. From about 10,000 array accesses, the time can be solely predicted by the number of array accesses.

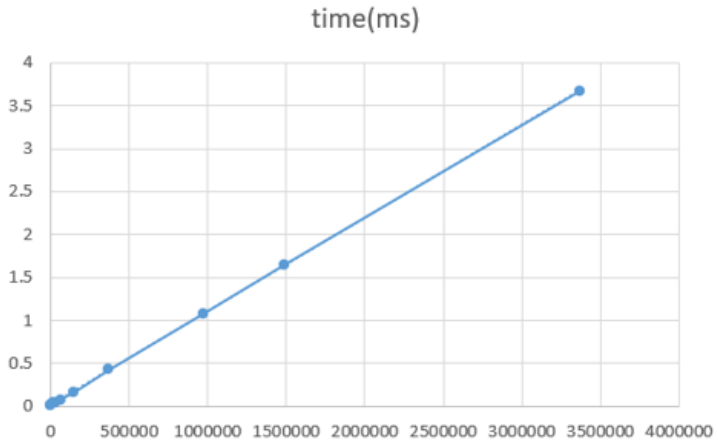
5.Shell Sort

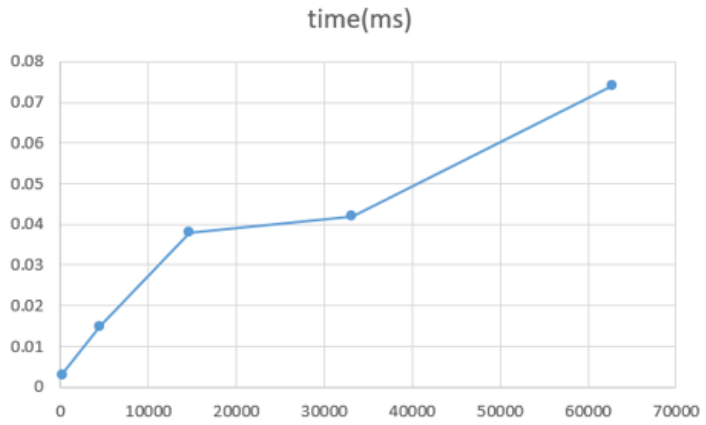
Benchmark:

```
2022-07-31 17:56:34 INFO  Benchmark - Begin run: Shell sort has 196 hits for 10 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:34 INFO  InsertionSortBenchmarks - 0.003 ms
=============================================================
2022-07-31 17:56:34 INFO  Benchmark - Begin run: Shell sort has 4496 hits for 50 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:34 INFO  InsertionSortBenchmarks - 0.015 ms
=============================================================
2022-07-31 17:56:34 INFO  Benchmark - Begin run: Shell sort has 14688 hits for 100 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:34 INFO  InsertionSortBenchmarks - 0.036 ms
=============================================================
2022-07-31 17:56:34 INFO  Benchmark - Begin run: Shell sort has 33140 hits for 150 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:34 INFO  InsertionSortBenchmarks - 0.041 ms
=============================================================
2022-07-31 17:56:34 INFO  Benchmark - Begin run: Shell sort has 62724 hits for 200 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:34 INFO  InsertionSortBenchmarks - 0.072 ms
=============================================================
2022-07-31 17:56:34 INFO  Benchmark - Begin run: Shell sort has 145748 hits for 300 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:34 INFO  InsertionSortBenchmarks - 0.156 ms
=============================================================
2022-07-31 17:56:34 INFO  Benchmark - Begin run: Shell sort has 372222 hits for 500 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:35 INFO  InsertionSortBenchmarks - 0.431 ms
=============================================================
2022-07-31 17:56:35 INFO  Benchmark - Begin run: Shell sort has 975310 hits for 800 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:36 INFO  InsertionSortBenchmarks - 1.086 ms
=============================================================
2022-07-31 17:56:36 INFO  Benchmark - Begin run: Shell sort has 1492872 hits for 1000 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:38 INFO  InsertionSortBenchmarks - 1.696 ms
=============================================================
2022-07-31 17:56:38 INFO  Benchmark - Begin run: Shell sort has 3371986 hits for 1500 Integers with 1,000 runs
.............................................................
2022-07-31 17:56:42 INFO  InsertionSortBenchmarks - 3.777 ms
```

Unit test:







The time has a linear relationship with the number of array accesses. The larger the number of array accesses, the more accurate it is. From about 30,000 array accesses, the time can be solely predicted by the number of array accesses.

References

[1] Quicksort — An Example

http://homepages.math.uic.edu/~leon/cs-mcs401-s08/handouts/quicksort.pdf

[2]Hillyard, R. C., & Liaozheng, Y. (2020). Huskysort. arXiv preprint arXiv:2012.00866.

[3] Conrado Martínez (1996). Randomized Binary Search Trees. Algorithms Seminar, Universitat Politecnica de Catalunya, Spain.