

COMP 2210 Assignment 3 Part B

Group 46

Xing Wang, Redatu Semeon

Abstract

In this report, we performed a repeatable experimental procedure to empirically discover the sorting algorithms implemented by the five methods of the SortingLab client — sort1, sort2, sort3, sort4, sort5. The five sorting algorithms implemented are merge sort, randomized quicksort, non-randomized quicksort, selection sort, and insertion sort.

1. Problem Review

In class, we learned four different sorting methods—selection sort, insertion sort, merge sort and quicksort. We studied their properties in two aspects—time complexity and stability. Selection sort is supposed to have a $O(n^2)$ complexity for all cases and is unstable. Insertion sort is supposed to have a $O(n^2)$ complexity for the average case and worst case, but has a $O(n)$ complexity for the best case, in which the array is fully or almost fully sorted. It is a stable sorting. Merge sort is the default sorting method used to sort reference data in Java. It is said to have a $O(n \log(n))$ complexity for all cases and is stable. The quicksort is a little bit more complex than the other sorting methods. Its complexity depends on the choice of pivot. For the average and best case, it picks the median value or a random value which, in most cases, is close to the median as the pivot and will have a $O(n \log(n))$ complexity. For the worst case, it picks the extreme value of the array as the pivot and has a complexity of $O(n^2)$. Regardless of the choice of pivot, quick sort is an unstable sorting method. In this assignment, we used five sorting methods—sort1, sort2, sort3, sort4 and sort5. Each of these methods had to be empirically tested in order to determine the exact sorting method. Note that quicksort have two implementations, both of which choose the left-most element of a partition (e.g., `a[left]`) as the pivot for that partition. Both use the same partition algorithm, but the randomized quicksort implementation makes the worst case probabilistically unlikely by randomly permuting the array elements before the quicksort algorithm begins. The nonrandomized quicksort might expose the algorithm's worst case by never shuffling the array elements for an already sorted array.

2. Experimental Procedures

All calculations were performed on Intel(R) Core™ i5-4210 2.40GHz CPU, with a 6GB RAM in a 64-

bit Windows 10 operating system on a personal HP Pavilion laptop. Our experimental procedures are as follows:

First, we made experimental discovery of running time for all of the 5 sorting methods. To do that, we generated an array of size N containing integer only by using pseudo random number generator, and systematically increased the array size N in *SortingLabClient.java* file and recorded the elapsed time. In class, we learned the growth rate of a time complexity function is a measure of how the amount algorithm does changes as its input size changes. A useful way of thinking about a growth rate for a time complexity function $T(N)$ is to think about the change in $T(N)$ as N doubles. And if we take the ratio of the elapsed time of successive program runs, we will get an approximate value of 2 for a $N \log N$ growth rate (Equation 1), 4 for a N^2 growth rate (Equation 2), and a constant value 2 for a N growth rate (Equation 3). In this case, we started from $N=1000$, then 2000, 4000, 8000, etc. Also note the used inner key is set to be 46, the assigned group number.

$$\text{Equation 1} \quad \frac{2N \log(2N)}{N \log(N)} = 2 \frac{\log(2) + \log(N)}{\log(N)} = 2(1 + x) \approx 2$$

$$\text{Equation 2} \quad \frac{(2N)^2}{N^2} = \frac{4N^2}{N^2} \approx 4$$

$$\text{Equation 3} \quad \frac{2N}{N} = 2$$

Since the big- $O(h)$ actually calculates the worst case time complexity, when performed in experiment, the ratio will be close to these constant instead of being the exact value.

As the order converges, this would give us the average case for the 5 sorting methods. So we can distinguish Selection Sort and Insertion Sort, which have time complexity $O(n^2)$, from the other three, which have time complexity $O(n \log(n))$. To distinguish all the 5 methods, the next thing we need to do is to take a look at the best case as well as the worst case. So we also tested the 5 sorting methods on the already sorted integer arrays of size N (i.e., in increasing order), which we called sorted case, and again, we systematically increased the array size N in *SortingLabClient.java* file and recorded the elapsed time. Then we tested the sorting methods on decreasing order sorted integer arrays, which we called reversed case. We know that in best case, Insertion Sort has running time of $O(n)$, while Selection Sort would always perform as $O(n^2)$, so we can distinguish Selection Sort from Insertion Sort. Also, the non-randomized QuickSort would perform as in its worst case, and we can distinguish it from the other two, i.e., randomized QuickSort and MergeSort.

Lastly, we are going to check the stability of the sorting methods. To do so, we generated our own data type—Pair, which implemented Comparable interface. It contains two integer elements, but the

compare method only operates on the first integer, ignoring the second. We created 7 new data elements $\{(5,1), (4,2), (3,3), (5,4), (2,5), (1,6), (5,7)\}$. Three of them have the identical first elements but different second ones. After performing a stable sort, it is supposed to be ordered as $\{(1,6), (2,5), (3,3), (4,2), (5,1), (5,4), (5,7)\}$. But after an unstable sorting, it should be something other than that. By doing that, we can distinguish the stable sorting methods from the unstable ones. We know that MergeSort and InsertionSort are stable, while the other three are not.

3. Data Collection and Analysis

The results were put in Table 3.1-Table 3.6 and schemed in Figure 1-Figure 5. Table 3.6 showed the result of the stability test, while table 3.1-Table 3.5 showed the sorting time needed for each sorting methods to sort an integer array of size N. Each table represents the results for random ordered arrays, sorted ordered arrays, and reversed ordered arrays, respectively. And for each part (i.e., the big column), the first sub-column is the size (N) of array in each run, the second column (Time) shows the elapsed time for each run, the third sub-column provides the ratio of the elapsed time(in second) of current run and previous run, and the last sub-column (K) calculates the logarithm of the ratio. Our aim is to find the converging value of K for each sorting method.

3.1 Sort 1 Results

Table 3.1 Results for Sort 1

Sort1											
Random Ordered Array				Sorted Ordered Array				Reverse Ordered Array			
N	Time	Ratio	K	N	Time	Ratio	K	N	Time	Ratio	K
1000	0.003			1000	0.002			1000	0.002		
2000	0.002	0.654	-0.612	2000	0.001	0.631	-0.664	2000	0.001	0.628	-0.671
4000	0.003	1.974	0.981	4000	0.002	2.022	1.016	4000	0.008	7.545	2.916
8000	0.008	2.462	1.300	8000	0.019	8.495	3.087	8000	0.017	2.089	1.062
16000	0.048	5.767	2.528	16000	0.014	0.723	-0.468	16000	0.013	0.723	-0.467
32000	0.073	1.531	0.614	32000	0.054	3.986	1.995	32000	0.039	3.102	1.633
64000	0.027	0.373	-1.423	64000	0.020	0.366	-1.449	64000	0.023	0.576	-0.797
128000	0.075	2.761	1.465	128000	0.039	1.980	0.985	128000	0.047	2.080	1.056
256000	0.113	1.510	0.594	256000	0.062	1.571	0.652	256000	0.071	1.519	0.603
512000	0.213	1.882	0.912	512000	0.168	2.724	1.446	512000	0.169	2.376	1.249
1024000	0.440	2.067	1.048	1024000	0.404	2.398	1.262	1024000	0.428	2.528	1.338
2048000	1.986	4.513	2.174	2048000	0.977	2.422	1.276	2048000	1.034	2.414	1.272
4096000	2.023	1.019	0.027	4096000	1.390	1.422	0.508	4096000	2.312	2.236	1.161
8192000	4.623	2.285	1.192	8192000	2.742	2.132	1.047	8192000	3.665	1.585	0.665

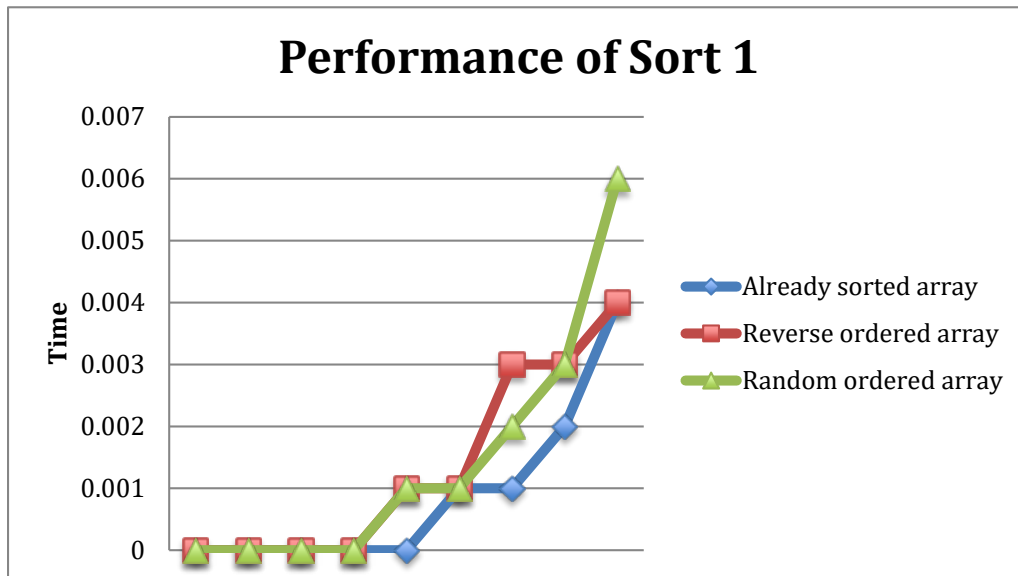


Figure 1

According to table 3.1 and figure 1, we see that sort1 sorted all the 3 types of arrays quite fast. The logarithm value is approximately slight larger than 1. Let us denoted it as 1.x, i.e.,

	Random Ordered Array	Sorted Ordered Array	Reverse Ordered Array
K	1.x	1.x	1.x

3.2 Sort 2 Results

Table 3.2 Results for Sort 2

Sort2											
Random Ordered Array				Sorted Ordered Array				Reverse Ordered Array			
N	Time	Ratio	K	N	Time	Ratio	K	N	Time	Ratio	K
1000	0.018			1000	0.000			1000	0.018		
2000	0.018	0.957	-0.063	2000	0.000	0.338	-1.565	2000	0.024	1.342	0.424
4000	0.026	1.470	0.555	4000	0.000	2.686	1.425	4000	0.053	2.235	1.161
8000	0.099	3.806	1.928	8000	0.001	1.706	0.770	8000	0.247	4.641	2.214
16000	0.353	3.573	1.837	16000	0.002	2.125	1.087	16000	0.699	2.826	1.499
32000	1.374	3.894	1.961	32000	0.005	3.375	1.755	32000	3.404	4.873	2.285
64000	7.009	5.101	2.351	64000	0.004	0.720	-0.474	64000	15.265	4.485	2.165
128000	30.772	4.390	2.134	128000	0.002	0.449	-1.156	128000	65.469	4.289	2.101
256000	169.166	5.497	2.459	256000	0.003	1.515	0.599	256000	509.077	7.776	2.959
				512000	0.001	0.521	-0.941				
				1024000	0.003	2.233	1.159				
				2048000	0.033	10.803	3.433				
				4096000	0.077	2.332	1.222				
				8192000	0.166	2.159	1.111				

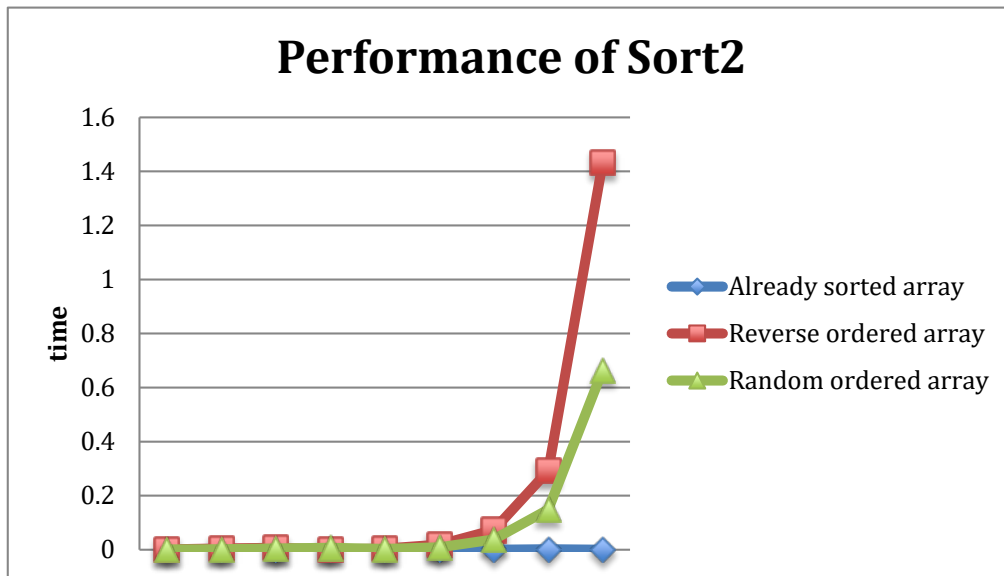


Figure 2

According to table 3.2 and figure 2, we see that for the sorted array, sort2 is quite fast and has time complexity is of order $1.x$. However, that is the only case it performs so well. For random ordered array, as well as reversed ordered array, the order of its time complexity is converging to 2, i.e.,

	Random Ordered Array	Sorted Ordered Array	Reverse Ordered Array
K	2	$1.x$	2

3.3 Sort 3 Results

Table 3.3 Results for Sort 3

Sort3											
Random Ordered Array				Sorted Ordered Array				Reverse Ordered Array			
N	Time	Ratio	K	N	Time	Ratio	K	N	Time	Ratio	K
1000	0.004			1000	0.003			1000	0.003		
2000	0.001	0.371	-1.431	2000	0.001	0.480	-1.060	2000	0.002	0.508	-0.976
4000	0.003	2.130	1.091	4000	0.003	2.231	1.158	4000	0.008	5.383	2.428
8000	0.006	2.249	1.170	8000	0.010	3.333	1.737	8000	0.010	1.232	0.300
16000	0.017	2.652	1.407	16000	0.015	1.526	0.610	16000	0.017	1.695	0.761
32000	0.020	1.207	0.271	32000	0.030	2.066	1.047	32000	0.022	1.269	0.344
64000	0.024	1.166	0.221	64000	0.031	1.023	0.033	64000	0.031	1.426	0.512
128000	0.054	2.266	1.180	128000	0.053	1.703	0.768	128000	0.043	1.404	0.490
256000	0.100	1.860	0.895	256000	0.080	1.513	0.598	256000	0.072	1.650	0.723
512000	0.221	2.204	1.140	512000	0.166	2.066	1.047	512000	0.187	2.614	1.386
1024000	0.435	1.971	0.979	1024000	0.363	2.185	1.128	1024000	0.444	2.369	1.244
2048000	1.065	2.447	1.291	2048000	1.112	3.064	1.616	2048000	0.956	2.155	1.108
4096000	2.376	2.230	1.157	4096000	2.708	2.435	1.284	4096000	1.873	1.959	0.970
8192000	5.585	2.351	1.233	8192000	6.611	2.442	1.288	8192000	4.828	2.578	1.366

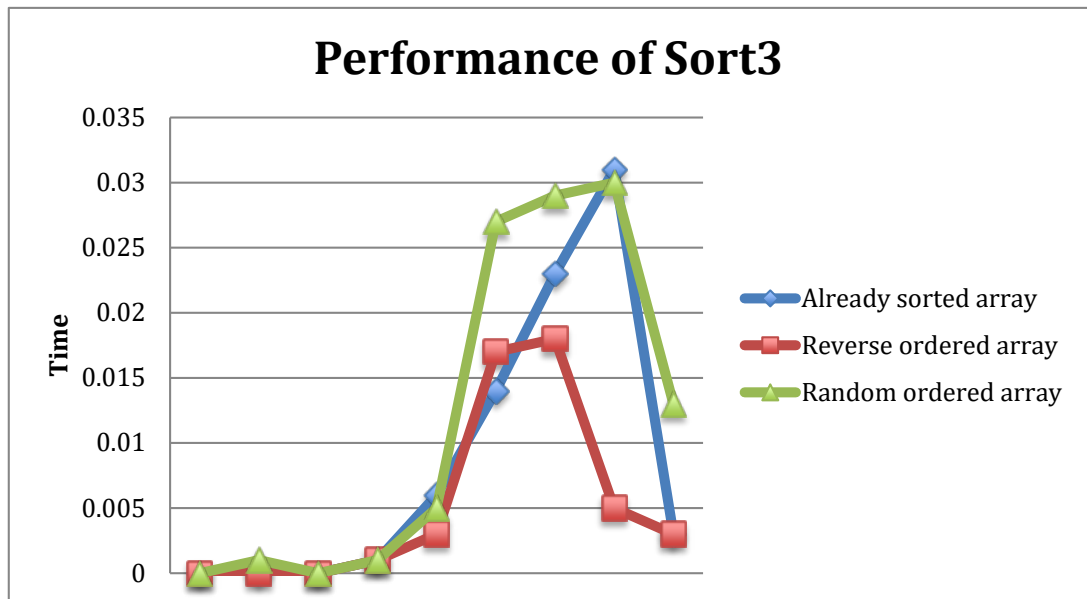


Figure3

According to table 3.3, we see that sort3 also sorted all the 3 types of arrays very fast. The logarithm values are all 1.x, i.e.,

	Random Ordered Array	Sorted Ordered Array	Reverse Ordered Array
K	1.x	1.x	1.x

3.4 Sort 4 Results

Table 3.4 Results for Sort 4

Sort4											
Random Ordered Array				Sorted Ordered Array				Reverse Ordered Array			
N	Time	Ratio	K	N	Time	Ratio	K	N	Time	Ratio	K
1000	0.003			1000	0.012			1000	0.012		
2000	0.001	0.254	-1.976	2000	0.005	0.399	-1.327	2000	0.051	4.358	2.124
4000	0.002	2.835	1.503	4000	0.060	12.704	3.667	4000	0.038	0.755	-0.405
8000	0.005	2.119	1.083	8000	0.049	0.815	-0.296	8000	0.081	2.117	1.082
16000	0.030	6.348	2.666	16000	0.253	5.146	2.363	16000	0.330	4.074	2.027
32000	0.063	2.139	1.097	32000	1.045	4.138	2.049	32000	1.524	4.614	2.206
64000	0.072	1.136	0.184	64000	4.251	4.069	2.025	64000	13.653	8.959	3.163
128000	0.051	0.702	-0.511	128000	20.477	4.817	2.268	128000	36.166	2.649	1.405
256000	0.076	1.503	0.588	256000	183.467	8.960	3.163	256000	322.974	8.930	3.159
512000	0.140	1.848	0.886								
1024000	0.297	2.118	1.083								
2048000	0.691	2.325	1.217								
4096000	1.593	2.305	1.205								
8192000	3.663	2.299	1.201								

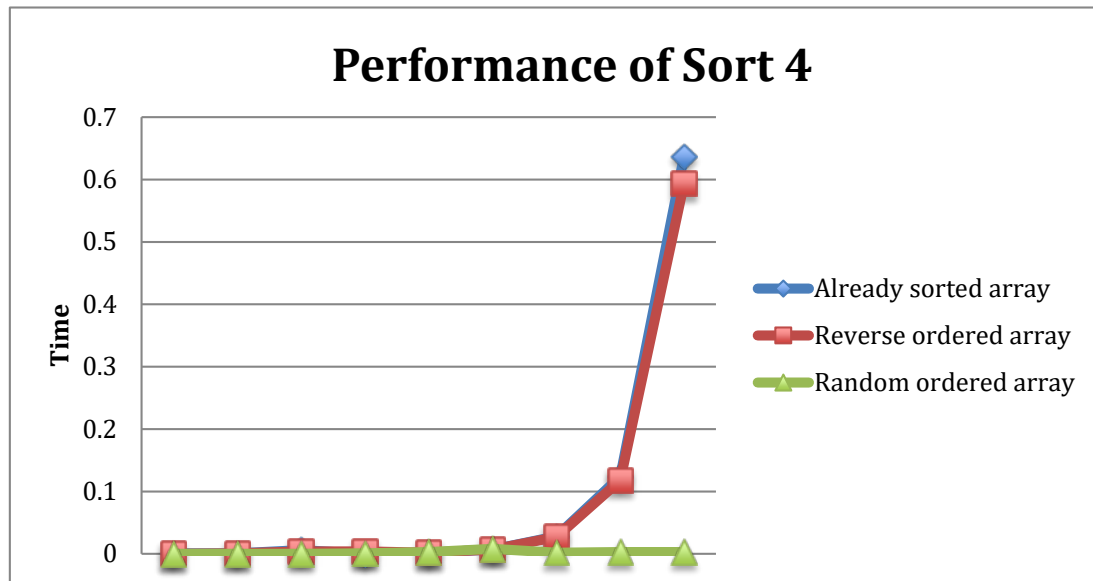


Figure 4

According to table 3.4 and figure 4, we see that for the random ordered array, sort4 is quite fast and has time complexity is of order 1.x. However, when the array becomes ordered, no matter in increased or decreased order, the order of its time complexity is converging to 2, i.e.,

	Random Ordered Array	Sorted Ordered Array	Reverse Ordered Array
K	<i>1.x</i>	<i>2</i>	<i>2</i>

3.5 Sort 5 Results

Table 3.5 Results for Sort 5

Sort5											
Random Ordered Array				Sorted Ordered Array				Reverse Ordered Array			
N	Time	Ratio	K	N	Time	Ratio	K	N	Time	Ratio	K
1000	0.012			1000	0.018			1000	0.015		
2000	0.009	0.783	-0.352	2000	0.006	0.316	-1.662	2000	0.005	0.319	-1.649
4000	0.032	3.455	1.789	4000	0.013	2.241	1.164	4000	0.018	3.666	1.874
8000	0.112	3.506	1.810	8000	0.062	4.855	2.279	8000	0.084	4.644	2.215
16000	0.484	4.339	2.117	16000	0.204	3.304	1.724	16000	0.265	3.168	1.663
32000	1.666	3.439	1.782	32000	0.928	4.541	2.183	32000	1.032	3.892	1.960
64000	6.819	4.094	2.034	64000	4.236	4.565	2.191	64000	4.444	4.306	2.106
128000	21.251	3.116	1.640	128000	23.034	5.437	2.443	128000	20.701	4.658	2.220
256000	197.504	9.294	3.216	256000	190.953	8.290	3.051	256000	193.411	9.343	3.224

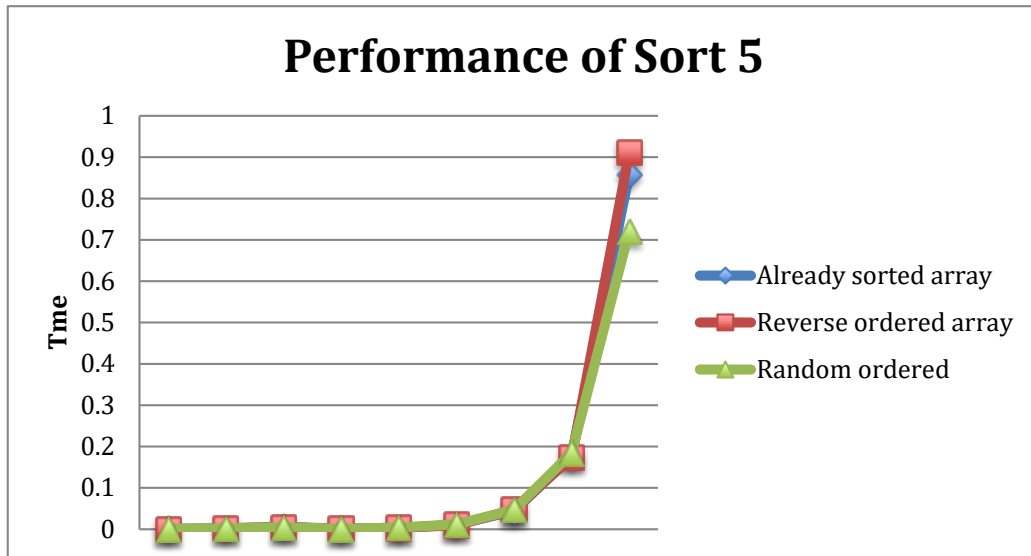


Figure 5

Sort 5 seems to be a quite slow method all the time, and in all 3 cases, the order of its time complexity is converging to 2, i.e.,

	Random Ordered Array	Sorted Ordered Array	Reverse Ordered Array
K	2	2	2

Note that since we know the best comparison based sorting algorithm has $O(n \log(n))$ complexity, we conclude that the 1.x order is of complexity $O(n \log(n))$. Also, the order 2 methods has complexity $O(n^2)$.

Also note that since we gathered data for several times to make sure the performance to be consistent, the data in the figures is not exactly same as the data in the tables. However, the trend for each part is the same.

3.6 Stability Test Results

Table 3.6 Stability Test Results for Sorting Methods

Array index	0	1	2	3	4	5	6	Stability
Sort1	(1, 6)	(2, 5)	(3, 3)	(4, 2)	(5, 1)	(5, 4)	(5, 7)	Stable
Sort2	(1, 6)	(2, 5)	(3, 3)	(4, 2)	(5, 1)	(5, 4)	(5, 7)	Stable
Sort3	(1, 6)	(2, 5)	(3, 3)	(4, 2)	(5, 7)	(5, 1)	(5, 4)	Unstable
Sort4	(1, 6)	(2, 5)	(3, 3)	(4, 2)	(5, 7)	(5, 1)	(5, 4)	Unstable
Sort5	(1, 6)	(2, 5)	(3, 3)	(4, 2)	(5, 4)	(5, 1)	(5, 7)	Unstable

According to table 3.6, sort1 and sort2 generated the same output while the second elements were kept in the original order, i.e., sort1 and sort 2 kept the original order of the second elements of $\{(5,1),(5,4) \text{ and } (5,7)\}$. So one of them will be insertion sort and the other is merge sort. Also from table 1, we

notice that sort3, sort4 and sort5 were unstable sorting methods, since the order of the second elements for the same first elements has been changed.

4. Interpretation

Based on the results shown in part 3, we could now make our inference.

Sort 5 always got a logarithm of ratio approaching to constant 2 for all the three cases when N was systematically doubled. We know the selection sort method had a time complexity of $O(N^2)$ for all of the best, the average and the worst case. So sort 5 was the selection sort.

The overall performance of sort 2 is not quite well. However, when working on the sorted array, its performance is prominent. We know that the Insertion Sort has this property, which has a linear running time for best case, but $O(N^2)$ performance for both average and worst cases.

Sort 4's performance for the already sorted array and reverse ordered array is quite poor, both running time are about $O(N^2)$, but it performed well for the random array. We say sort 4 must be non-randomized quick sort. The performance of quicksort depends on the choice of pivot. A best choice of pivot is the median or some value close to the median. A worst choice will be the extreme value, such as minimum or maximum. The implementation of non-randomized quick sort use the most left element of the array, which will leads to worst choice of pivot in the already sorted array or the reverse ordered array. In this case, the time complexity is $O(N^2)$, while for the random case, it has time complexity of $O(n \log(n))$. Randomized quicksort permutes the array elements before the algorithm begins. This makes choice of the worst pivot be probabilistically impossible. Thus for randomized quicksort, we will always have best case, in probability.

The left two methods now are Merge Sort and Randomized Quick Sort. According to the experimental data, they both performs very well in all cases, which we say they both have $O(n \log(n))$ performance in all cases. However, with the help of the stability tests, we can easily distinguish them. We know that Merge Sort is a stable algorithm which does not switch the original relative order of the duplicate elements, while Quick Sort is not. During the experiment, we see that sort1 kept the original order of the second elements of $\{(5,1),(5,4) \text{ and } (5,7)\}$, while sort 3 did not. So sort 1 must be merge sort and therefore, sort 3 is Randomized Quick Sort, which completes our argument.

To make it clear, we summarize our interpretation and conclusion in the following table:

	Random	Sorted	Reversed	Stable ?	
Sort 1	<i>1.x</i>	<i>1.x</i>	<i>1.x</i>	Yes	Merge Sort
Sort 2	<i>2</i>	<i>1.x</i>	<i>1.x</i>	Yes	Insertion Sort
Sort 3	<i>1.x</i>	<i>1.x</i>	<i>1.x</i>	No	Randomized Quick Sort
Sort 4	<i>1.x</i>	<i>1.x</i>	<i>1.x</i>	No	Non-Randomized Quick Sort
Sort 5	<i>2</i>	<i>2</i>	<i>2</i>	No	Selection Sort

In conclusion, an empirical analysis of algorithm of SortingLab.jar was performed to get the big-O running time complexity of and stability of five sorting methods—merge sort, selection sort, insertion sort, random quicksort and non-randomized quicksort. Based on the results, we successfully identified the five sorting methods. Sort 1 is merge sort, sort 2 is insertion sort, and sort 3 is randomized quicksort, sort4 is non-randomized quick sort, while sort 5 was selection sort.

Reference

1. Dr. Heandrix Dean, *Algorithm_Analysis.pptx*,Page24-27.
2. Venugopal, S. (2006). *Data Structures Outside-In with Java* (1st edition.). Prentice Hall. ISBN 0-13-198619-8.
3. Lewis&Loftus ,*Java Software solutions foundations of program design* (7th edition),Addison Welsey. ISBN 978-0-13-214918-1.

Appendix A

● Random Ordered Client

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Comparator;

public final class SortingLabClient {

    /** Drives execution. */
    public static void main(String[] args) {
        int key = 46;
        // run one sort on an array of Strings
        /*String[] as = {"D", "B", "E", "C", "A"};
        SortingLab<String> sls = new SortingLab<String>(key);
        sls.sort1(as);*/

        Pair[] a = {new Pair(5,1), new Pair(4,2), new Pair(3,3),
                    new Pair(5,4), new Pair(2,5), new Pair(1,6), new Pair(5,7)};

        SortingLab<Pair> sl2 = new SortingLab<>(key);
        sl2.sort1(a);
        System.out.println("=====sort 1=====");
        for (int i = 0; i < 7; i++){
            System.out.print(a[i] + "\t");
        }

        // run a sort on multiple Integer arrays of increasing length
        SortingLab<Integer> sli = new SortingLab<Integer>(key);
        int M = 500000; //2000000; // max capacity for array
        int N = 1000;    //10000;    // initial size of array
        double start;
        double elapsedTime;

        double[] t = new double[100];
        double R;
        double k;

        //for (; N < M; N *= 2) {
        for (int i = 0; i < 16; i++) {
            Integer[] ai = getIntegerArray(N, Integer.MAX_VALUE);
            start = System.nanoTime();
```

```

        sli.sort5(ai);
        elapsedTime = (System.nanoTime() - start) / 1000000000d;
        System.out.print(N + "\t");
        System.out.printf("%4.3f\t", elapsedTime);

        t[i] = elapsedTime;
        if (i >= 1) {
            R = t[i] / t[i - 1];
            k = Math.log(R) / Math.log(2);
            System.out.printf("%4.3f\t", R);
            System.out.printf("%4.3f\n", k);
        }
        else {System.out.print("\n");}
        N *= 2;
    }
}

/**
 * Returns an array of size N filled with Integer values
 * in the range 0 .. max - 1.
 */
private static Integer[] getIntegerArray(int N, int max) {
    Integer[] a = new Integer[N];
    java.util.Random rng = new java.util.Random();
    for (int i = 0; i < N; i++) {
        a[i] = rng.nextInt(max);
    }
    /*Arrays.sort(a);
    Integer[] b = new Integer[N];
    for (int i = 0; i < N; i++) {
        b[i] = a[N - i - 1];
    }
    return b;*/
    return a;
}
}

```

● Ordered Array Method

```
private static Integer[] getIntegerArray(int N, int max) {
    Integer[] a = new Integer[N];
    java.util.Random rng = new java.util.Random();
    for (int i = 0; i < N; i++) {
        a[i] = rng.nextInt(max);
    }
    Arrays.sort(a);
    /*Integer[] b = new Integer[N];
    for (int i = 0; i < N; i++) {
        b[i] = a[N - i - 1];
    }
    return b;*/
    return a;
}
```

● Reversed Order Method

```
private static Integer[] getIntegerArray(int N, int max) {
    Integer[] a = new Integer[N];
    java.util.Random rng = new java.util.Random();
    for (int i = 0; i < N; i++) {
        a[i] = rng.nextInt(max);
    }
    Arrays.sort(a);
    Integer[] b = new Integer[N];
    for (int i = 0; i < N; i++) {
        b[i] = a[N - i - 1];
    }
    return b;
    //return a;
}
```

● Pair Class

```
public class Pair implements Comparable<Pair> {  
    private int val1;  
    private int val2;  
  
    public Pair(int v1, int v2) {  
        this.val1 = v1;  
        this.val2 = v2;  
    }  
  
    public int compareTo(Pair other) {  
        //return (this.val1).compareTo(other.val1);  
        if (this.val1 < other.val1)  
            return -1;  
        else if (this.val1 > other.val1)  
            return 1;  
        else  
            return 0;  
    }  
  
    public String toString() {  
        return String.format("(" + val1 + ", " + val2 + ")");  
    }  
}
```