COMP 2210 Empirical Analysis Assignment – Part B

Zejian Zhong

July 3, 2017

Abstract

In this assignment, I performed a repeatable experimental procedure that allowed me to empirically discover the sorting algorithms implemented by the five methods of the SortingLab class — sort1, sort2, sort3, sort4, sort5. Those five methods located in a provided .jar file that are invisible to users. The five sorting algorithms implemented are merge sort, randomized quicksort, nonrandomized quicksort, selection sort, and insertion sort.

1 Problem Overview

The biggest problem for me is that I do not know what the code looks like in the .jar file, thus I cannot modify it, either. I need to verify each of those sorting algorithms based on the running time and calculate the k by using following equation:

$$T(N) \propto N^k \Longrightarrow \frac{T(2N)}{T(N)} \propto \frac{(2N)^k}{N^k} = \frac{2^k N^k}{N^k} = 2^k$$

We can know that, the time complexity should be proportional to N_k for int k, and the ratio R should converge to a constant 2^k, so $k = \log_2(R)$. And then we can find the big-Oh as $O(N^k)$.

Below is a comparison table for those five sorting algorithms:

Name	Best	Average	Worst	Stable
Insertion Sort	n	n^2	n^2	Yes
Selection Sort	n^2	n^2	n^2	No
Merge Sort	n log n	n log n	n log n	Yes
Quicksort	n log n	n log n	n^2	No

2 Experimental Procedure

Below are some specs of the used machine:

- System: Windows 10 Pro 64-bit

- Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
- RAM: 16.0 GB
- Java version: Java 8 update 131

My key was my Banner ID:903907977. I changed the initial size of array N and max capacity M to get a reasonable time for my test data. I decided to test each algorithm based on random ordered array, sorted ordered array and reversed ordered array and tried to find more useful information to help me verify the algorithms.

```
/** return an array of random integer values. */
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);
   }
   return a;
}
/** return an ordered array of random integer values. */
private static Integer[] getIntegerArrayO(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);
   return a;
}
/** return a reversed array of random integer values. */
private static Integer[] getIntegerArrayR(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;
}
```

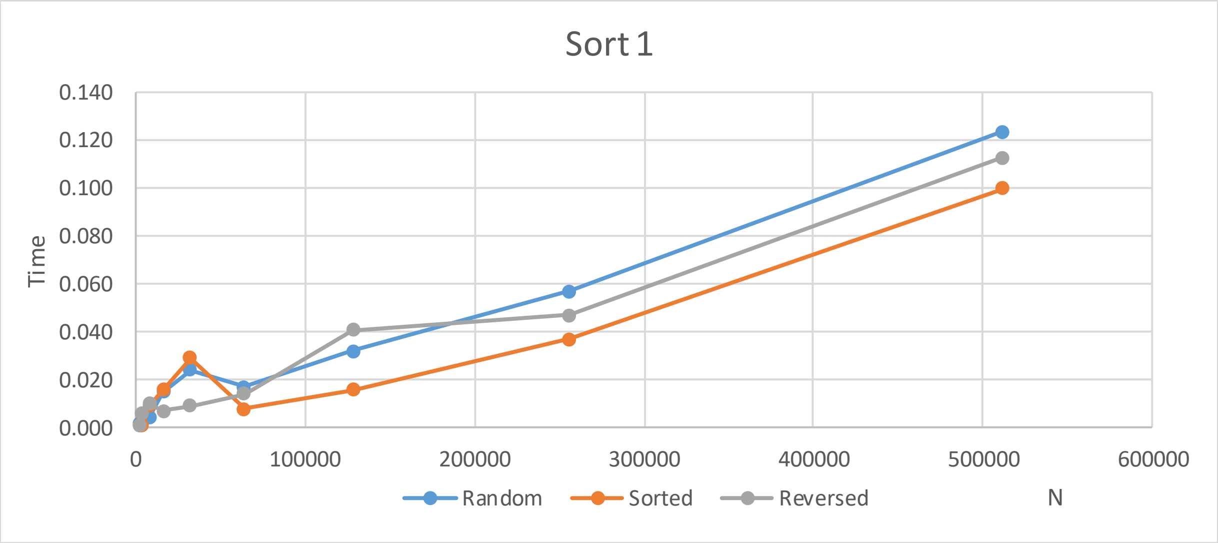
To get data from each sorting algorithm, I changed the class name in main class every time I run it to get the running time based on those three types of array and did the same for every sort.

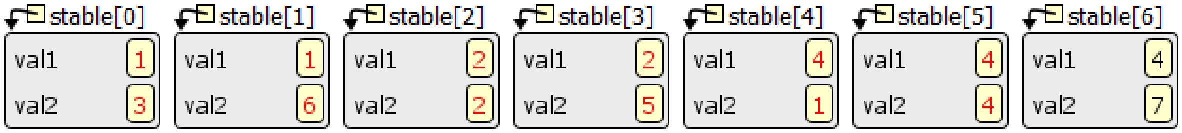
As for stability test, I created a new test file and also an array of integer pairs, which, to be specific, is (4, 1), (2, 2), (1, 3), (4, 4), (2, 5), (1, 6), (4, 7). If the tested sorting is stable ,the result should be (1, 3), (1, 6), (2, 2), (2, 5), (4, 1), (4, 4), (4, 7). I used debug mode in jGRASP to get a quick look of sorted arrays and compare it with the result above to determine the sort is stable or not. By comparing the data and stability test result, we can find out the correct sorting algorithms in the .jar file.

3 Data Collection and Analysis

From my test, I got all the data and made some tables and figures to see it clearly.

Sort 1											
Random				Sorted				Reversed			
N	Time	Ratio	k	N	Time	Ratio	k	N	Time	Ratio	k
2000	0.002			2000	0.001			2000	0.001		
4000	0.002	1.000	0.000	4000	0.001	1.000	0.000	4000	0.006	6.000	2.585
8000	0.004	2.000	1.000	8000	0.009	9.000	3.170	8000	0.010	1.667	0.737
16000	0.015	3.750	1.907	16000	0.016	1.778	0.830	16000	0.007	0.700	-0.515
32000	0.024	1.600	0.678	32000	0.029	1.813	0.858	32000	0.009	1.286	0.363
64000	0.017	0.708	-0.497	64000	0.008	0.276	-1.858	64000	0.014	1.556	0.637
128000	0.032	1.882	0.913	128000	0.016	2.000	1.000	128000	0.041	2.929	1.550
256000	0.057	1.781	0.833	256000	0.037	2.313	1.209	256000	0.047	1.146	0.197
512000	0.124	2.175	1.121	512000	0.100	2.703	1.434	512000	0.113	2.404	1.266

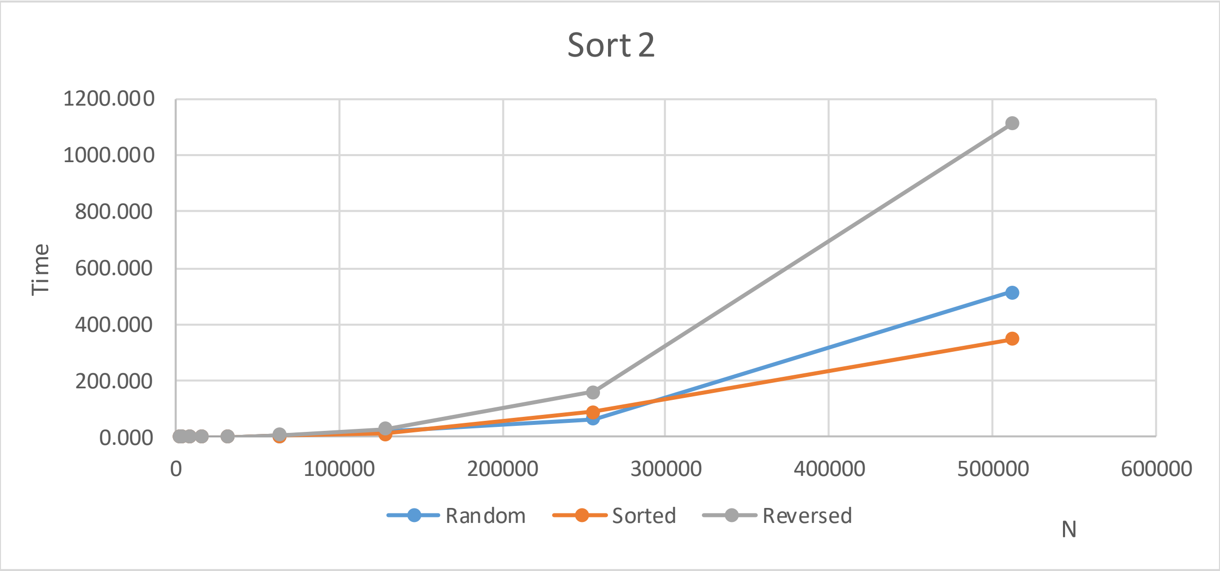


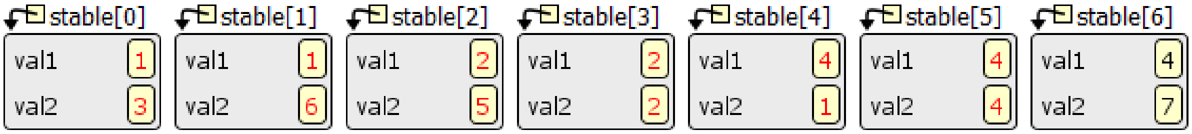


stable[0]	stable[1]	stable[2]	stable[3]	stable[4]	stable[5]	stable[6]
val1 1	val1 1	val1 2	val1 2	val1 4	val1 4	val1 4
val2 3	val2 6	val2 2	val2 5	val2 1	val2 4	val2 7

Sort 1 is stable, k is generally less than 2.

Sort 2											
Random				Sorted				Reversed			
N	Time	Ratio	k	N	Time	Ratio	k	N	Time	Ratio	k
2000	0.009			2000	0.009			2000	0.009		
4000	0.014	1.556	0.637	4000	0.008	0.889	-0.170	4000	0.016	1.778	0.830
8000	0.058	4.143	2.051	8000	0.032	4.000	2.000	8000	0.067	4.188	2.066
16000	0.248	4.276	2.096	16000	0.147	4.594	2.200	16000	0.342	5.104	2.352
32000	1.015	4.093	2.033	32000	0.764	5.197	2.378	32000	1.338	3.912	1.968
64000	4.372	4.307	2.107	64000	3.361	4.399	2.137	64000	7.910	5.912	2.564
128000	19.775	4.523	2.177	128000	13.764	4.095	2.034	128000	30.549	3.862	1.949
256000	65.459	3.310	1.727	256000	89.818	6.526	2.706	256000	161.554	5.288	2.403
512000	515.055	7.868	2.976	512000	350.000	3.897	1.962	512000	1113.803	6.894	2.785

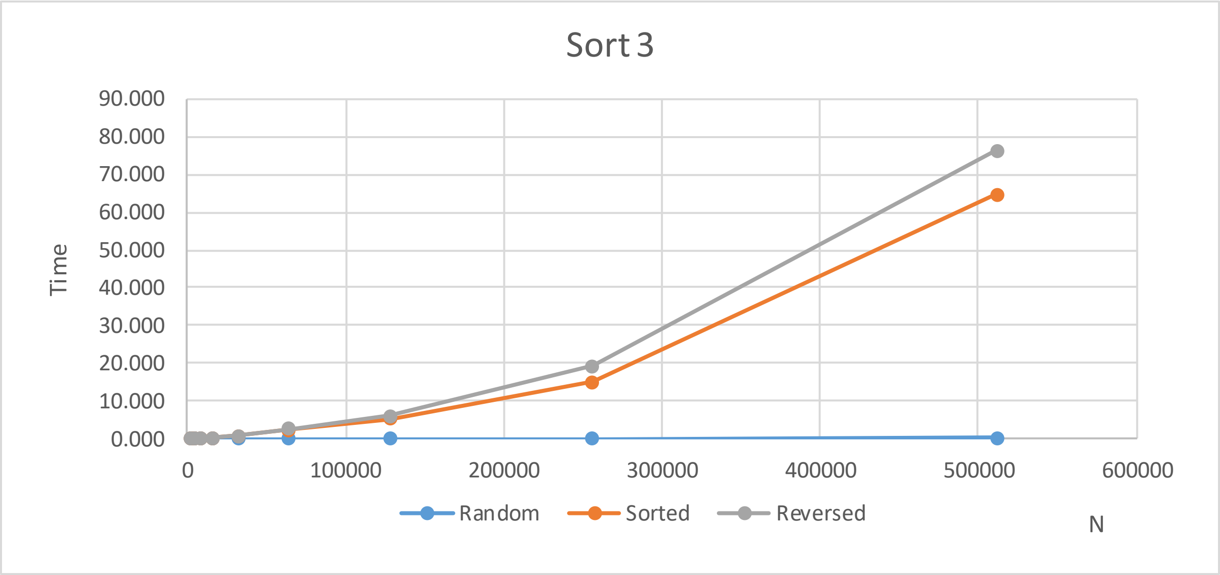


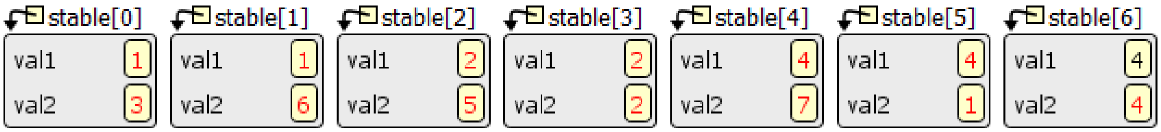


	stable[0]	stable[1]	stable[2]	stable[3]	stable[4]	stable[5]	stable[6]
val1	1	1	2	2	4	4	4
val2	3	6	5	2	1	4	7

Sort 2 is not stable, and has relative long sorting time, k is approaching to 2.

Sort 3											
Random				Sorted				Reversed			
N	Time	Ratio	k	N	Time	Ratio	k	N	Time	Ratio	k
2000	0.002			2000	0.027			2000	0.022		
4000	0.001	0.500	-1.000	4000	0.011	0.407	-1.295	4000	0.027	1.227	0.295
8000	0.003	3.000	1.585	8000	0.051	4.636	2.213	8000	0.038	1.407	0.493
16000	0.090	30.000	4.907	16000	0.246	4.824	2.270	16000	0.180	4.737	2.244
32000	0.020	0.222	-2.170	32000	0.861	3.500	1.807	32000	0.843	4.683	2.228
64000	0.016	0.800	-0.322	64000	2.238	2.599	1.378	64000	2.517	2.986	1.578
128000	0.021	1.313	0.392	128000	5.114	2.285	1.192	128000	5.915	2.350	1.233
256000	0.037	1.762	0.817	256000	14.881	2.910	1.541	256000	19.241	3.253	1.702
512000	0.085	2.297	1.200	512000	64.742	4.351	2.121	512000	76.272	3.964	1.987





stable[0]
val1 1
val2 3

stable[1]
val1 1
val2 6

stable[2]
val1 2
val2 5

stable[3]
val1 2
val2 2

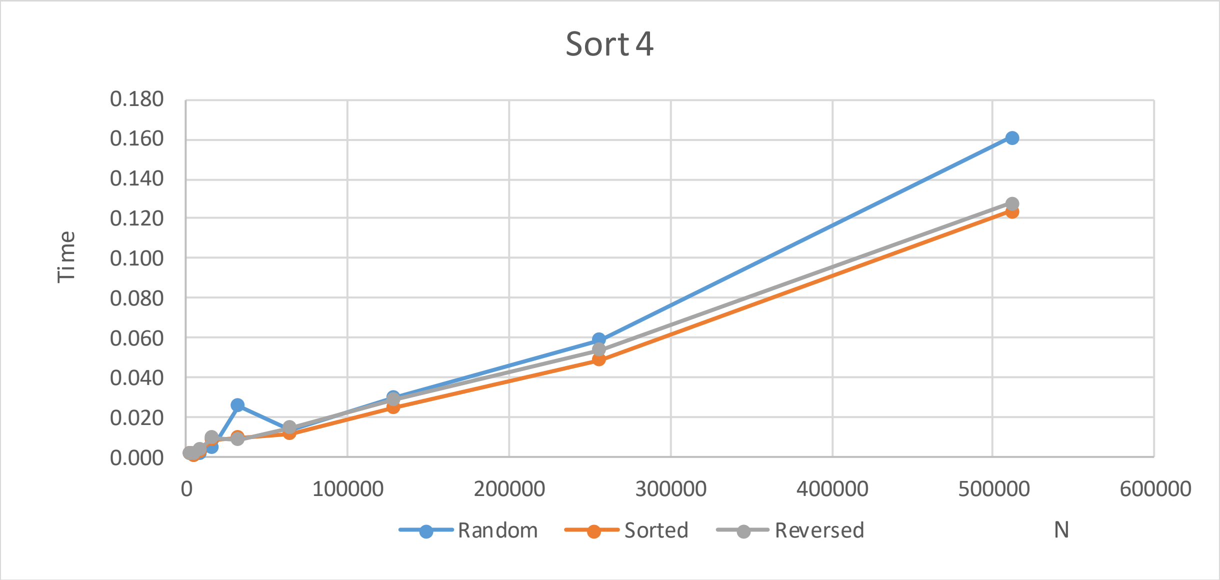
stable[4]
val1 4
val2 7

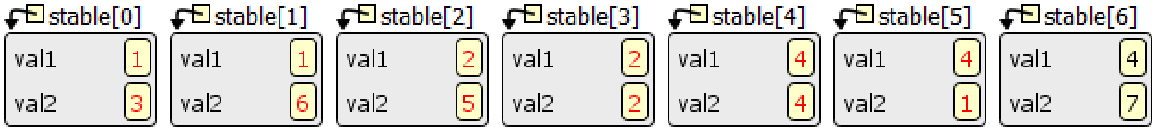
stable[5]
val1 4
val2 1

stable[6]
val1 4
val2 4

Sort 3 is not stable, and has worse sorting speed when array is not random.

Sort 4											
Random				Sorted				Reversed			
N	Time	Ratio	k	N	Time	Ratio	k	N	Time	Ratio	k
2000	0.002			2000	0.002			2000	0.002		
4000	0.001	0.500	-1.000	4000	0.001	0.500	-1.000	4000	0.002	1.000	0.000
8000	0.002	2.000	1.000	8000	0.003	3.000	1.585	8000	0.004	2.000	1.000
16000	0.005	2.500	1.322	16000	0.009	3.000	1.585	16000	0.010	2.500	1.322
32000	0.026	5.200	2.379	32000	0.010	1.111	0.152	32000	0.009	0.900	-0.152
64000	0.014	0.538	-0.893	64000	0.012	1.200	0.263	64000	0.015	1.667	0.737
128000	0.030	2.143	1.100	128000	0.025	2.083	1.059	128000	0.029	1.933	0.951
256000	0.059	1.967	0.976	256000	0.049	1.960	0.971	256000	0.054	1.862	0.897
512000	0.161	2.729	1.448	512000	0.124	2.531	1.339	512000	0.128	2.370	1.245

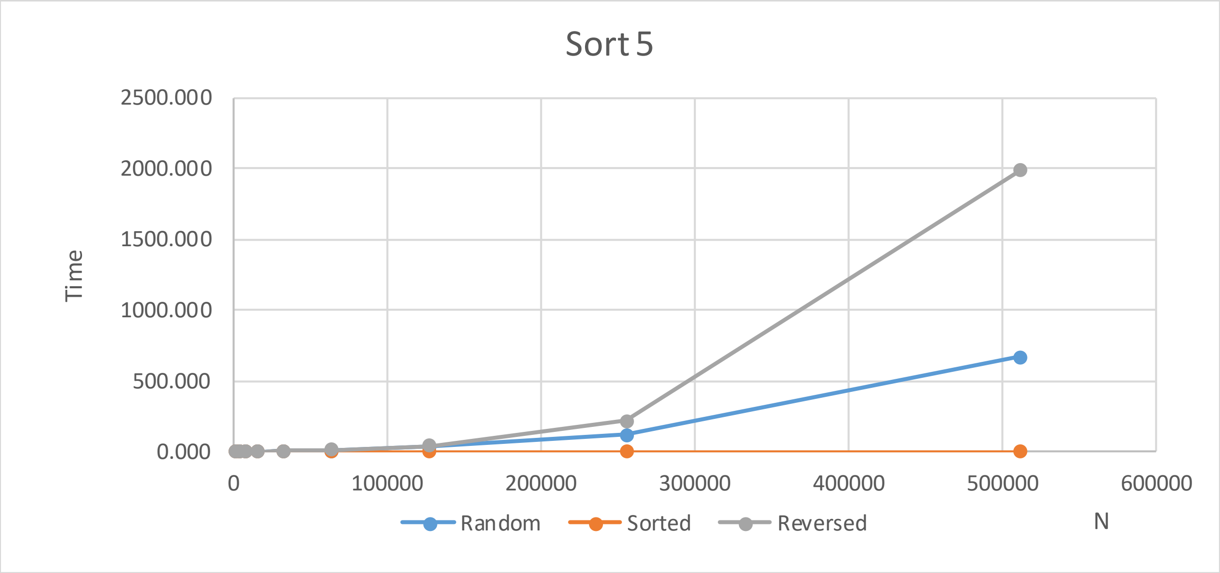


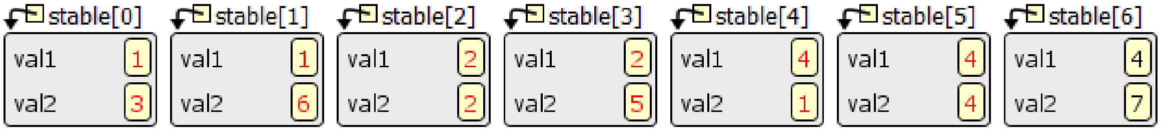


	stable[0]	stable[1]	stable[2]	stable[3]	stable[4]	stable[5]	stable[6]
val1	1	1	2	2	4	4	4
val2	3	6	5	2	4	1	7

Sort 4 is not stable, k is generally less than 2.

Sort 5											
Random				Sorted				Reversed			
N	Time	Ratio	k	N	Time	Ratio	k	N	Time	Ratio	k
2000	0.018			2000	0.000			2000	0.017		
4000	0.023	1.278	0.354	4000	0.000			4000	0.034	2.000	1.000
8000	0.058	2.522	1.334	8000	0.000			8000	0.127	3.735	1.901
16000	0.228	3.931	1.975	16000	0.000			16000	0.499	3.929	1.974
32000	1.013	4.443	2.152	32000	0.001			32000	2.703	5.417	2.437
64000	5.973	5.896	2.560	64000	0.002	2.000	1.000	64000	11.030	4.081	2.029
128000	32.034	5.363	2.423	128000	0.001	0.500	-1.000	128000	41.437	3.757	1.909
256000	116.664	3.642	1.865	256000	0.001	1.000	0.000	256000	217.156	5.241	2.390
512000	668.810	5.733	2.519	512000	0.004	4.000	2.000	512000	1988.007	9.155	3.195





	stable[0]	stable[1]	stable[2]	stable[3]	stable[4]	stable[5]	stable[6]
val1	1	1	2	2	4	4	4
val2	3	6	2	5	1	4	7

Sort 5 is stable, and has the shortest running time when the array is already sorted.

4 Interpretation

From the data shown above, we noticed that sort 3 has better performance when the array is random. We know that non-random quicksort has this kind of property because it always starts from the most left, which will cause the worst situation when the array is sorted or reversed ordered. Sort 3 is also non-stable, so it must be **non-randomized quicksort**.

Sort 2 is not stable, and the k is approaching to 2 for all three types of array. **Selection sort** is not stable, and the time complexity is O(N^2), which matches the case of sort 2.

Sort 1 is stable, and k is generally less than 2, so the time complexity of it could not be O(N^2). According to the table, the only option left is **merge sort**.

Because we only have two stable sorts, besides sort 1, another stable sort is sort 5, and it must be **insertion sort**. Insertion sort has best case when the array is already sorted, so it proves that our guess is correct.

Then the last one, sort 4, must be **randomized quicksort**. It is not stable, and the k is not approaching to 2, so the time complexity should be n log n, which is the property of randomized quicksort.

Here is the result of our discussion:

Sort 1	Merge Sort
Sort 2	Selection Sort
Sort 3	Non-randomized Quicksort
Sort 4	Randomized Quicksort
Sort 5	Insertion Sort