COMP 2210 Assignment 3 Part B

Group 44

John C. Carroll James C. Bass

February 24, 2014

Abstract

This document presents an example for any student to follow when identifying sorting algorithms without having the source code available. In Part B of this assignment, five sorting algorithms are provided in the SortingLab class and the goal is determining which type of sort they represent. This experiment extends Part A of Assignment 3 in that, within this experiment, it is most helpful to already have an understanding of how to derive time complexities from any given method. Even so, of the five sorting algorithm possibilities, it is possible for them to share time complexities. Thus, it is imperative to this experiment's goals to show the reader what needs to be tested to identify each of the five sorting algorithms. Within this experiment, there are three tests added that check and distinguish the methods through an average case test(1), a best, and potentially worst case test(2), and a stability test(3). Through these methods, data was yielded. The data that was recorded and analyzed in this assignment resulted in the following results for key(44):

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

1. Problem Overview

For Part B's experiment, as stated in the abstract, it is imperative to have covered and understood Part A. Big-Oh time complexities are essentially the unit being compared in this problem. Only look at part B after part A is reviewed.

In this experiment, five methods of the SortingLab class are provided – sort1, sort2, sort3, sort4, sort5. These five sorting algorithms implemented are merge sort, randomized quicksort, non-randomized quicksort, selection sort, and insertion sort, and are presented in an order specific to an assigned key. This key should be the same as in Part A which is the same as the group number. (i.e. Group 23 would invoke the constructor SortingLab(23)). Also, the Clock class should be used again to gather the experimental data.

The goal for this experiment is to distinguish and identify which sort number belongs to each of the five sorting algorithms said to be given. The knowledge gained from Part A will help again in the gathering of timing data. Although this is crucial to the experimental procedure, it will not be sufficient to distinguish between all of the sorting algorithms.

In this experiment, trials can be ran on the SortingLab class using the provided SortingLabClient, as it has relevant example code for use. Modifications should be made to this client and observable data should then be yielded. Deductions and justified calculations about the class method's big-Oh time complexity for each sort should then be made through its analysis.

It is Part B's experimental procedure that will develop on these distinguishing tools. For identifying and being able to tell the sorting algorithms apart, modifications and additions to the client should be made. There are three tests that should be added that check and distinguish the methods. These tests include: a printing and recording of an average case test for all of the five sorts; a best, and potentially worst case test; along with a stability test. With these tests, and an analysis of data, there should be a no doubt which sort is which sorting algorithm.

2. Experimental Procedure

Experimental environment: a group member's home personal computer. (same as Part A)

Machine specifications:

Operating System

Windows 7 Professional 64-bit SP1

CPU

Intel Core i7 3770K @ 4.2GHz (Over-clocked) 38 °C (average)
52 °C (when running test)

RAM

G.SKILL 16.0GB Dual-Channel DDR3 1866MHz @ 2133MHz (Over-clocked)
Timing (9-10-9-28)

Motherboard

ASUSTeK COMPUTER INC. P8Z77-V (LGA1155) 34 °C

Graphics

3072MB ATI AMD Radeon HD 7950 (Gigabyte™) 43 °C

Storage

232GB Samsung SSD 840 Series (SSD) 33 °C
1863GB Seagate (SATA) 40 °C

Java

Java Runtime Environment
Path C:\Program Files (x86)\Java\jre7\bin\java.exe
Version 7.0
Update 51
Build 13

Java Runtime Environment
Path C:\Program Files\Java\jdk1.7.0_25\bin\java.exe
Version 7.0
Update 25
Build 17

Java Runtime Environment
Path C:\Program Files\Java\jre7\bin\java.exe
Version 7.0
Update 25
Build 17

The procedure for the experiment went as follows:

Created the project inside of the same project folder as the files that were provided and added all of them to the project. With set-up complete, the given SortingLabClient was ran using its default source code.

The client printed the elapsed time value for a starting size of N, then doubled it with each iteration, and ran until it reached or went over the maximum size M.

In this provided code, Part A's same ratio and $\log_2$(ratio) should be added for added convenience along with run number and a header. Set M for just over 204800, so that it will run a sufficient amount of times for a relatively good read on the ratios. Leave N if felt needed as it is a low number, which is a good starting point. Utilizing the code given, shown in Screenshot 1, output can already be produced for a display.

```
//Used for Average case checker
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;
}
```

Screenshot 1-Default code that is given which creates an array of integers with random values at each index

The random number generator's arrays can be sorted by each of the five unknown sorts. Below in Screenshot 2 is the code edited and introduced for just sort 1 but can be used for all sorts simultaneously to have a long output.

```
//=========================================================
   // Random Case/Average case Checker
//=========================================================
  // N = 100;
System.out.println("Method: sort1"); //***Added
System.out.println("-----Run-----N------Time------Ratio----lgRatio"); //***Added
for (int i = 0; N < M; N *= 2, i++) {
   //Integer[] ai = getWorstArray(N);
   Integer[] ai = getIntegerArray(N, Integer.MAX_VALUE);
   Clock clock = new Clock();

   sli.sort1(ai);//Sort NUMBER ************

   prevTime = elapsedTime;

   elapsedTime = clock.elapsedTime();

   double ratio = elapsedTime/prevTime;
   double lgratio = (Math.log(ratio) / Math.log(2));

   System.out.printf("%7d %7d", i+1, N);
   System.out.printf("%10.3fs %10.3f %10.3f\n", elapsedTime, ratio, lgratio);
}
```

Screenshot 2-This is the needed code to run the first checker for one run.

The first test case we need to make is an average case "checker." This test checks what the sorts time complexities are under an average scenario given that the arrays are going through a random number generator.

The next test case used was a "best" case scenario checker. At this point, there are five sorts. To distinguish them, there are some characteristics that tell them apart.

The only way to tell between Selection sort and Insertion sort is their Best case. This is due to their worst case and average cases both being $O(N^2)$, but Insertion sort's best case is O(N) while Selection's is still $O(N^2)$. This checker can identify them from one another in combination with the average case checker.

This checker also can tell the Randomized quicksort from Non-randomized quicksort. The randomized quicksort implementation makes the worst case probabilistically unlikely by randomizing the array elements before the quicksort algorithm begins. The Non-randomized quicksort exposes the algorithm's worst case because it never shuffles the array elements, and so it can potentially run through all of the elements. This "best" case scenario checker is a worst case scenario checker for the quicksort due to its process. This test case will also find the Non-randomized quicksort.

The code implemented to find the Merge sort is the stability checker. It consists of a method based off of the supplied random number generator but with a few additions.

```
//Used for Stability Checker
private static Tester[] createTester(int length) {

    Tester[] createdArray = new Tester[length];

    java.util.Random rng = new java.util.Random();

    for (int i = 0; i < length; i++) {
        createdArray[i] = new Tester(rng.nextInt(10), rng.nextInt(10));
    }
    return createdArray;
}
```

Screenshot 3-Tester method for stability test Checker

```
1  public class Tester implements Comparable<Tester> {
2
3      int a;
4      int b;
5      public Tester(int int1, int int2) {
6          a = int1;
7          b = int2;
8      }
9      public int compareTo(Tester array) {
10         return this.a - array.a;
11     }
12     public String toString() {
13         String toString = "(" + a + ", " + b + ")";
14         return toString;
15     }
16 }
```

Screenshot 4 - Tester.class which implements Comparable

In conjunction with this Tester class which implements Comparable, the stability checker needs to be placed in main() like so:

```
public final class SortingLabClient {

    public static void main(String[] args) {
        int key = 44;
        // run one sort on an array of Strings
        String[] as = {"D", "B", "E", "C", "A"};

        SortingLab<String> sls = new SortingLab<>(key);
        sls.sort1(as);

        // run a sort on multiple Integer arrays of increasing length
        SortingLab<Integer> sli = new SortingLab<>(key);
        //=========================================================
        // Stability checker
        //=========================================================

        Tester[] stabilityTestArray = createTester(10);
        SortingLab<Tester> slobj = new SortingLab<>(key);

        System.out.println("Array Randomized: ");
        System.out.println(java.util.Arrays.toString(stabilityTestArray));

        slobj.sort5(stabilityTestArray);

        System.out.println("Array Sorted: ");
        System.out.println(java.util.Arrays.toString(stabilityTestArray));
        int something = 0;
        if (something == 0) {
            return;
        }
```

Screenshot 5 - Utilizing the Stability Tester in main for one sort at a time

This is all of the code needed to identify the five sorting algorithms. In the next section, the data collection is reported along with an analysis of it.

3. Data Collection and Analysis

To better analyze the data that this experiment outputs, it is crucial in understanding the differences between the sorting algorithms' time complexities. This chart is found in COMP 2210's lecture notes: Sorting.pptx.pdf.

	Selection	Insertion	Mergesort	Quicksort
Worst case	$O(N^2)$	$O(N^2)$	O(N log N)	$O(N^2)$
Average case	$O(N^2)$	$O(N^2)$	O(N log N)	O(N log N)
Best case	$O(N^2)$	O(N)	O(N log N)	O(N log N)
In-place?	Yes	Yes	No	Yes
Stable?	Yes	Yes	Yes	No
Adaptive?	No	Yes	No	No

This experiment's output is based off of the code from the Experimental Procedure. Below are screenshots, graphs, and tables pertaining to the results. The elapsed times in the data below do not begin run from N = 100 to N = 204800. Screenshot 6 is from the Average case test.

Average Case Test Results

Method: sort1

Run	N	Time	Ratio	lgRatio
1	100	0.000s	NaN	NaN
2	200	0.002s	Infinity	Infinity
3	400	0.002s	1.000	0.000
4	800	0.010s	5.000	2.322
5	1600	0.013s	1.300	0.379
6	3200	0.007s	0.538	-0.893
7	6400	0.032s	4.571	2.193
8	12800	0.123s	3.844	1.943
9	25600	0.505s	4.106	2.038
10	51200	2.130s	4.218	2.076
11	102400	8.832s	4.146	2.052
12	204800	36.867s	4.174	2.062

Method: sort2

Run	N	Time	Ratio	lgRatio
1	100	0.000s	0.000	-Infinity
2	200	0.001s	Infinity	Infinity
3	400	0.000s	0.000	-Infinity
4	800	0.001s	Infinity	Infinity
5	1600	0.001s	1.000	0.000
6	3200	0.003s	3.000	1.585
7	6400	0.001s	0.333	-1.585
8	12800	0.002s	2.000	1.000
9	25600	0.003s	1.500	0.585
10	51200	0.006s	2.000	1.000
11	102400	0.012s	2.000	1.000
12	204800	0.031s	2.583	1.369

Method: sort3

Run	N	Time	Ratio	lgRatio
1	100	0.000s	0.000	-Infinity
2	200	0.000s	NaN	NaN
3	400	0.000s	NaN	NaN
4	800	0.001s	Infinity	Infinity
5	1600	0.003s	3.000	1.585
6	3200	0.006s	2.000	1.000
7	6400	0.012s	2.000	1.000
8	12800	0.024s	2.000	1.000
9	25600	0.005s	0.208	-2.263
10	51200	0.012s	2.400	1.263
11	102400	0.014s	1.167	0.222
12	204800	0.031s	2.214	1.147

Method: sort4

Run	N	Time	Ratio	lgRatio
1	100	0.000s	0.000	-Infinity
2	200	0.001s	Infinity	Infinity
3	400	0.002s	2.000	1.000
4	800	0.000s	0.000	-Infinity
5	1600	0.001s	Infinity	Infinity
6	3200	0.006s	6.000	2.585
7	6400	0.022s	3.667	1.874
8	12800	0.092s	4.182	2.064
9	25600	0.421s	4.576	2.194
10	51200	1.880s	4.466	2.159
11	102400	7.985s	4.247	2.087
12	204800	35.379s	4.431	2.148

Method: sort5

Run	N	Time	Ratio	lgRatio
1	100	0.000s	0.000	-Infinity
2	200	0.000s	NaN	NaN
3	400	0.000s	NaN	NaN
4	800	0.000s	NaN	NaN
5	1600	0.000s	NaN	NaN
6	3200	0.000s	NaN	NaN
7	6400	0.001s	Infinity	Infinity
8	12800	0.001s	1.000	0.000
9	25600	0.002s	2.000	1.000
10	51200	0.005s	2.500	1.322
11	102400	0.011s	2.200	1.138
12	204800	0.022s	2.000	1.000

Best Case Test Results

Method: sort1

Run	N	Time	Ratio	lgRatio
1	100	0.000s	NaN	NaN
2	200	0.002s	Infinity	Infinity
3	400	0.003s	1.500	0.585
4	800	0.010s	3.333	1.737
5	1600	0.010s	1.000	0.000
6	3200	0.008s	0.800	-0.322
7	6400	0.026s	3.250	1.700
8	12800	0.100s	3.846	1.943
9	25600	0.402s	4.020	2.007
10	51200	1.609s	4.002	2.001
11	102400	6.436s	4.000	2.000
12	204800	25.980s	4.037	2.013

Method: sort2

Run	N	Time	Ratio	lgRatio
1	100	0.000s	0.000	-Infinity
2	200	0.000s	NaN	NaN
3	400	0.000s	NaN	NaN
4	800	0.000s	NaN	NaN
5	1600	0.002s	Infinity	Infinity
6	3200	0.004s	2.000	1.000
7	6400	0.002s	0.500	-1.000
8	12800	0.002s	1.000	0.000
9	25600	0.004s	2.000	1.000
10	51200	0.006s	1.500	0.585
11	102400	0.013s	2.167	1.115
12	204800	0.027s	2.077	1.054

Method: sort3

Run	N	Time	Ratio	lgRatio
1	100	0.000s	0.000	-Infinity
2	200	0.000s	NaN	NaN
3	400	0.000s	NaN	NaN
4	800	0.000s	NaN	NaN
5	1600	0.003s	Infinity	Infinity
6	3200	0.005s	1.667	0.737
7	6400	0.011s	2.200	1.138
8	12800	0.021s	1.909	0.933
9	25600	0.007s	0.333	-1.585
10	51200	0.003s	0.429	-1.222
11	102400	0.007s	2.333	1.222
12	204800	0.013s	1.857	0.893

Method: sort4

Run	N	Time	Ratio	lgRatio
1	100	0.000s	0.000	-Infinity
2	200	0.000s	NaN	NaN
3	400	0.000s	NaN	NaN
4	800	0.000s	NaN	NaN
5	1600	0.000s	NaN	NaN
6	3200	0.000s	NaN	NaN
7	6400	0.000s	NaN	NaN
8	12800	0.000s	NaN	NaN
9	25600	0.000s	NaN	NaN
10	51200	0.000s	NaN	NaN
11	102400	0.000s	NaN	NaN
12	204800	0.000s	NaN	NaN

Method: sort5

Run	N	Time	Ratio	lgRatio
1	100	0.000s	NaN	NaN
2	200	0.000s	NaN	NaN
3	400	0.000s	NaN	NaN
4	800	0.000s	NaN	NaN
5	1600	0.001s	Infinity	Infinity
6	3200	0.004s	4.000	2.000
7	6400	0.017s	4.250	2.087
8	12800	0.065s	3.824	1.935
9	25600	0.264s	4.062	2.022
10	51200	1.054s	3.992	1.997
11	102400	4.196s	3.981	1.993
12	204800	17.197s	4.098	2.035

It has said that sorting algorithms can share time complexities. Here is an excellent example of that. From observing Screenshot 6, it appears sort1 and sort4 are very similar, and the other three are similar to each other. With this the Best case checker, it will be possible to better identify the sorts.

So far, from the data above, there can be multiple observations made about each sort method.

- Sort 1: has a time complexity of $O(N^2)$ in both a randomized (average) and sorted array (best case).
- Sort 2: has a time complexity of O(NlogN) for a randomized array, and a time complexity of O(N2) for a natural order array.
- Sort 3: has a time complexity of O(N log N) both a randomized and natural order array. has a time complexity of O(N log N) both a randomized and natural order array.
- Sort 4: has a time complexity of O(N2) for a randomized array and a time complexity of O(N) for a natural order array.
- Sort 5: has a time complexity of O(NlogN) for a randomized array, and a time complexity of O(N2) for a natural order array.

From those observations, a few of the sorts can be identified as such:

- Sort 1: Selection Sort
- Sort 2: Unknown
- Sort 3: Unknown
- Sort 4: Insertion Sort
- Sort 5: Non-Randomized Quicksort

The next step is to observe the stability check's results between sorts 2 and 3 to identify Merge sort from Randomized Quicksort.

Stability Check:

Sort2's results:

```
Array Randomized:
[(7, 7), (0, 1), (1, 5), (4, 8), (8, 9), (6, 5), (6, 5), (7, 5), (1, 0), (0, 5)]
Array Sorted:
[(0, 5), (0, 1), (1, 5), (1, 0), (4, 8), (6, 5), (6, 5), (7, 7), (7, 5), (8, 9)]
```

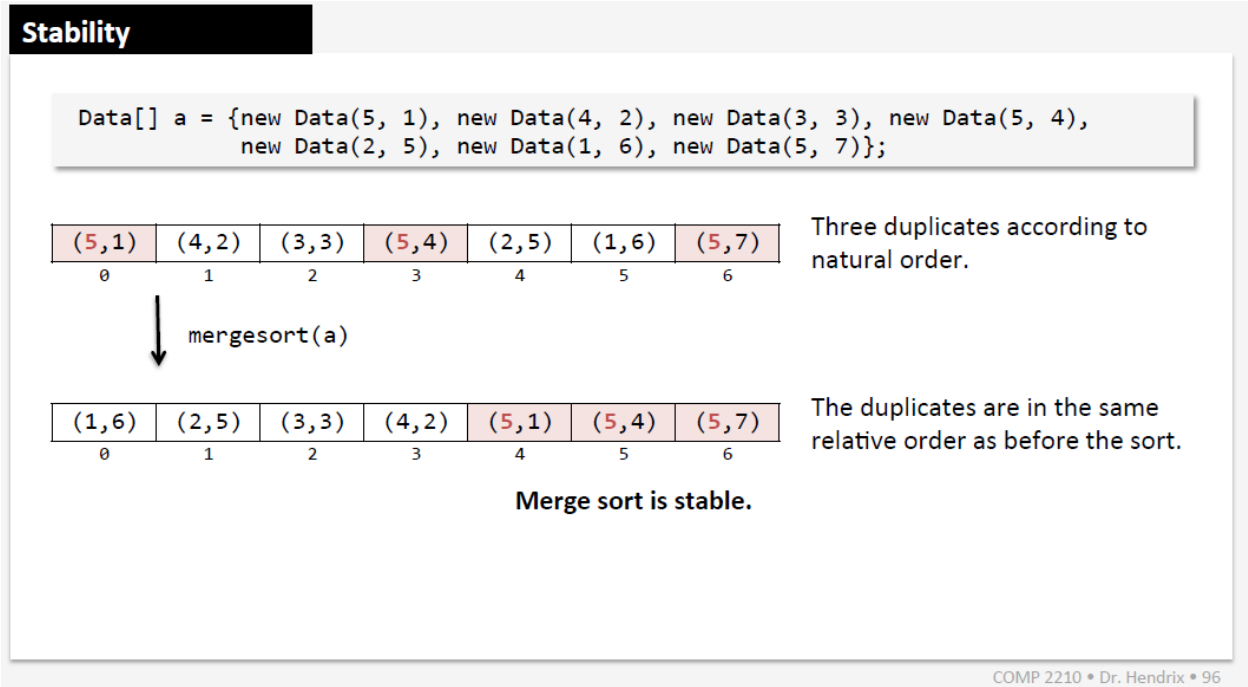
Screenshot 6-output from Sort2

Sort3's results:

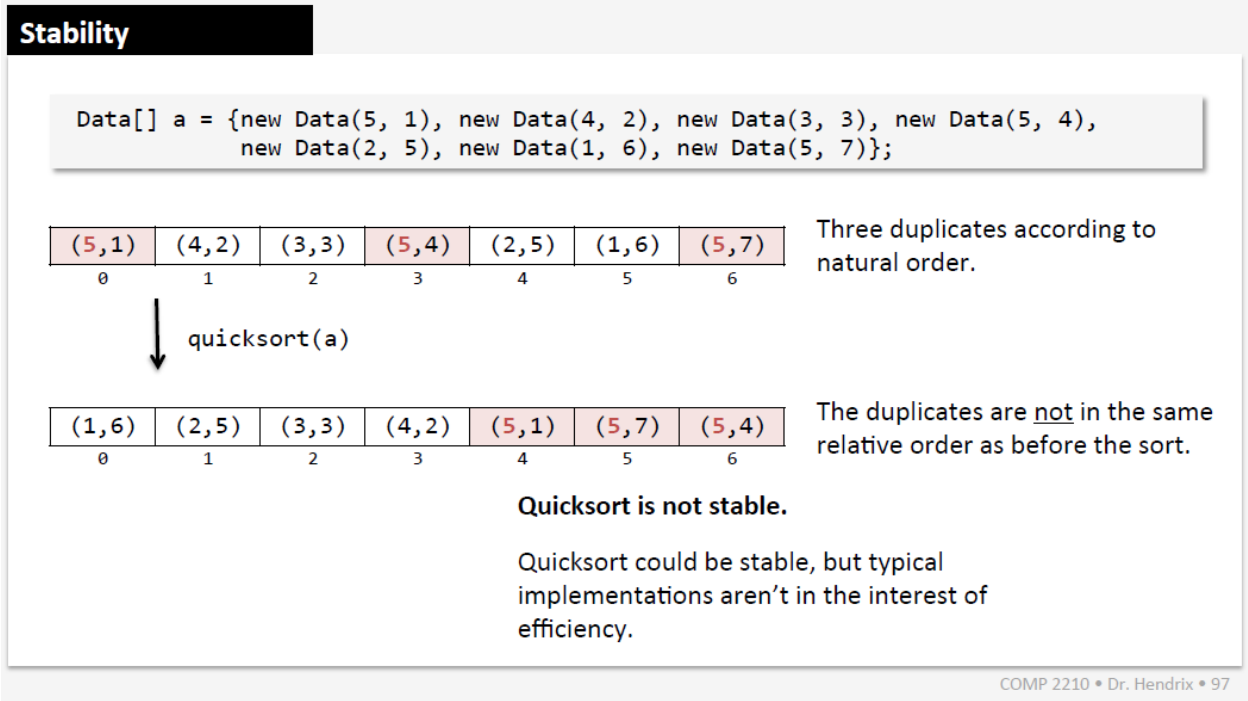
```
Array Randomized:
[(3, 0), (8, 0), (4, 1), (9, 5), (9, 3), (9, 8), (5, 8), (4, 3), (0, 1), (8, 9)]
Array Sorted:
[(0, 1), (3, 0), (4, 1), (4, 3), (5, 8), (8, 0), (8, 9), (9, 5), (9, 3), (9, 8)]
```

Screenshot 7-output from sort3

This stability check sorts the first number in the pair and leaves the second number not sorted whatsoever besides leaving it with the number it is paired with. In sort 2, the duplicates' second numbers are <u>not</u> in the same relative order as before they sorted. However, in sort3, the duplicates' second numbers <u>are</u> in the same relative order as before the sort, therefore it is stable by definitions presented via that same powerpoint from Comp 2210.



Screenshot 8-From the 2210 powerpoint showing Merge sort is stable



Screenshot 9-From the 2210 powerpoint showing Quicksort is not stable

Now that stability check has identified the final two unknowns, the data gives the following results for key(44):

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

4. Interpretation

This experiment was to test if it was possible to decipher five sorting algorithms without their source codes available. Through outputting all of the sorting methods by way of the average case test, it was found that there were two distinct classes based upon timing data. The Selection sort and Insertion sorts' average sorts were the same and the other three were near identical to each other. Through the exposure of some of the sorts' best and worst cases by way of using a natural order array in test form, three of the sorts were able to be identified. Selection sort could be told apart from Insertion sort, and the Non-randomized Quicksort as well. This is due to

observing both of the last two test's data, side-by-side. The last two, Merge sort and Randomized Quicksort, went through a stability check. This brought Merge sort forward as stable, and identified which of the two was not.

With these characteristics observed and confirmed, the result of this experiment was conclusive as it satisfied the beginning goal. All of the sorts were identified as so following results for key(44):

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