Abstract

I distinguished between five sorting algorithms in the **`SortingLab`** class using existing knowledge about the algorithms and empirical timing data. I determined that the identities of the sorting algorithms are the following:

- **`Sort1`**: Merge Sort
- **`Sort2`**: Randomized Quicksort
- **`Sort3`**: Selection Sort
- **`Sort4`**: non-randomized Quicksort
- **`Sort5`**: Insertion Sort

1 Problem Overview

Our task was to experimentally distinguish between five sorting algorithms in the **`SortingLab`** class using both existing knowledge about the algorithms and by collected empirical timing data. Each of the five sorting algorithms were implemented as methods in the **`SortingLab`** class: **`Sort1`**, **`Sort2`**, **`Sort3`**, **`Sort4`**, **`Sort5`**. Each method manipulates a `SortingLab<T super Comparable>` object and returns `void`. The implemented algorithms included merge sort, randomized quicksort, non-randomized quicksort, selection sort, and insertion sort.

2 Experimental Procedure

1. To distinguish between the different algorithms, I employed existing knowledge about both their time complexity and stability. Given that average cases for selection sort and insertion sort run in O(N^2) time while the average cases for quick sort and merge sort run in O(NlogN) time, I can separate the five algorithms into a *slow group* and a *fast group*.

2. To distinguish between the algorithms in the *slow group* (insertion sort and selection sort), I employed existing knowledge about the best case time complexity of each algorithm. Given that selection sort has a best case of O(N^2) and that insertion sort has a best case of O(N), I can determine the identities of the two algorithms by timing their completion of a sorted array.

3. To distinguish between merge sort and the two quick sort algorithms, I employed existing knowledge about the stability of the two algorithms. Given that merge sort is stable while quick sort is not stable, I can determine which of the three *fast* algorithms is merge sort by analyzing the order of *equal* elements before and after the sort. I used `String` objects and sorted them by length.

4. To distinguish between the randomized quicksort and the non-randomized quicksort, I employed existing knowledge about the time complexity of their worst cases. Given that each of the algorithms selects pivots from the

beginning of the array, I can determine the identity of the two algorithms by testing an array in reverse order.

2.1 Experimental materials

The experimental materials include the **`SortingLabClient`** class and the **`resources.jar`** files. The **`SortingLabClient`** is used to collect timing data by accessing the hidden **`SortingLab`** class located in the **`resources.jar`** file. To instantiate a **`SortingLab`** object, I was required to input my Banner ID as a parameter, serving as a **key**.

2.2 Collecting running time data

Using the given **`SortingLabClient`**, I timed the completion of the different sorting algorithms at exponentially scaled input sets ($10000 < \mathbf{N} < 2000000$). The code implementing the timing is below.

```
SortingLab<Integer> sli = new SortingLab<Integer>(key);
    int M = 2000000; // max capacity for array
    int N = 10000;   // initial size of array
    double start;
    double elapsedTime;
    for (; N < M; N = 2) {
        Integer[] ai = getIntegerArray(N, Integer.MAX_VALUE);
        start = System.nanoTime();
        sli.sort1(ai);
        elapsedTime = (System.nanoTime() - start) / 1_000_000_000d;
        System.out.print(N + "\t");
        System.out.printf("%4.3f\n", elapsedTime);
      }
```

I used this algorithm to acquire all of the timing data needed for each step in the Experimental Procedure above. The only variable that I modified between the different tests was the contents and data type of the `Integer[]` and the contents of the print statements.

2.3 Analyzing running time data

I primarily analyzed the scalability of the sorting algorithms by analyzing the run time of large data sets.

3 Data Collection and Analysis

The timing data was gathered by using the **`SortingLabClient`** class. The computer environment in which the data was collected is described below.

- Computer: Microsoft Surface Book (13.5 in. 2015), 2.6GHz Intel i7 Processor,

- Operating System: Microsoft Windows 10 Pro
- Java: 1.8.0
  - **`javac -version`**: **`javac`** 1.8.0
  - **`java -version`**: **`java`** version \1.8.0" Java(TM) SE Runtime Environment (build 1.8.0-b132) Java HotSpot(TM) 64-Bit Server VM (build 25.0-b70, mixed mode)
- `System.nanoTime()`: used to measure elapsed time

3.1 Timing Data

Timing Data was generated using the **`SortingLabClient`** class. For each test at every step, I timed the completion of each sorting algorithm using data sizes ranging from **N** = 10,000 to **N** = 2,000,000 elements with each successive trial containing $2N$ elements when compared to the previous trial.

3.1.1 *Slow* and *Fast* group data

As noted in step 1 of the Experimental Process, I first separated the five algorithms into two groups: the *slow group* and the *fast group*.

Table 1: Random Set: **`Sort1`**

Trial	Data Size (N)	Time (s)
1	10000	0.007
2	20000	0.009
3	40000	0.022
4	80000	0.047
5	160000	0.103
6	320000	0.229
7	640000	0.447
8	1280000	1.02

Table 2: Random Set: **`Sort2`**

Trial	Data Size (N)	Time (s)
1	10000	0.004
2	20000	0.008
3	40000	0.020
4	80000	0.037
5	160000	0.180
6	320000	0.310
7	640000	0.638
8	1280000	1.742

Trial	Data Size (N)	Time (s)

Table 3: Random Set: **`Sort3`**

Trial	Data Size (N)	Time (s)
1	10000	0.414
2	20000	1.612
3	40000	6.871
4	80000	28.343
5	160000	191.407
6	320000	1065.986
7	640000	14179.546
8	1280000	

Table 4: Random Set: **`Sort4`**

Trial	Data Size (N)	Time (s)
1	10000	0.007
2	20000	0.010
3	40000	0.021
4	80000	0.053
5	160000	0.092
6	320000	0.178
7	640000	0.427
8	1280000	1.031

Table 5: Random Set: **`Sort5`**

Trial	Data Size (N)	Time (s)
1	10000	0.372
2	20000	1.661
3	40000	6.576
4	80000	35.584
5	160000	209.364
6	320000	4898.029
7	640000	10190.337
8	1280000	

I stopped **`Sort5`** and **`Sort3`** after several hours due to their prolonged run time.

Given that there are two of the algorithms that process with O(N^2) and three algorithms process with O(N), the *slow* algorithms are **`Sort3`** and **`Sort5`** and the *fast* algorithms are **`Sort1`**, **`Sort2`**, and **`Sort4`**.

3.1.2 Insertion sort versus selection sort

As noted in step 2 of the Experimental Procedure, I differentiated between the insertion sort algorithm and the selection sort algorithm by timing their respective completions of a sorted array.

Table 6: Best Case: **`Sort3`**

Trial	Data Size (N)	Time (s)
1	10000	0.672
2	20000	2.715
3	40000	10.940
4	80000	46.385
5	160000	187.843
6	320000	1220.590
7	640000	6041.340
8	1280000	11482.717

Table 7: Best Case: **`Sort5`**

Trial	Data Size (N)	Time (s)
1	10000	0.002
2	20000	0.002
3	40000	0.001
4	80000	0.001
5	160000	0.003
6	320000	0.005
7	640000	0.009
8	1280000	0.018

Given that insertion sort has a best case of O(N) and that selection sort has a best case of O(N^2) , **`Sort5`** is obviously insertion sort due to its fast completion when compared to **`Sort3`** which is selection sort.

3.1.3 Distinguish Merge Sort Using Stability

Given that merge sort does not affect the order of *equal* elements (stable), I distinguished merge sort from the two quicksorts by checking if *equal* elements changed position from the algorithm. Because time was not a factor in this test, I was able to use a smaller data set (`N = 10`).

Table 8: Stability Test

Algorithm	Input	Output	Stability
`Sort1`	`[A, B, C, D, E, F, G]`	`[A, B, C, D, E, F, G]`	Stable
`Sort2`	`[A, B, C, D, E, F, G]`	`[A, G, C, B, E, D, F]`	Unstable
`Sort4`	`[A, B, C, D, E, F, G]`	`[F, E, G, A, B, D, C]`	Unstable

Given that merge sort is stable while quicksort is not stable, **`Sort1`** is merge sort while **`Sort2`** and **`Sort4`** are versions of quicksort.

3.1.4 Distinguish Quicksorts

Given that non-randomized quicksort has a worst case of O(N^2) while randomized quicksort has a worst case of O(N), I could distinguish between the two algorithms by inputing a descending array.

Table 9: Worst Case: **`Sort2`**

Trial	Data Size (N)	Time (s)
1	10000	0.008
2	20000	0.012
3	40000	0.032
4	80000	0.063
5	160000	0.102
6	320000	0.188
7	640000	0.527
8	1280000	1.123

Table 10: Worst Case: **`Sort4`**

Trial	Data Size (N)	Time (s)
1	10000	0.235
2	20000	0.572
3	40000	1.300
4	80000	2.698
5	160000	5.912
6	320000	12.904
7	640000	19.048
8	1280000	41.078

Because **`Sort4`** was significantly slower than **`Sort2`**, I can determine that **`Sort4`** is the non-randomized quicksort while **`Sort2`** is the randomized quicksort.

4 Interpretation

As outlined in Timing data, I distinguished between the different sorting algorithms using both knowledge about the algorithms and experimental data. The identities that I have determined for the algorithms are as follows:

- `Sort1`: Merge Sort
- `Sort2`: Randomized Quicksort
- `Sort3`: Selection Sort
- `Sort4`: non-randomized Quicksort
- `Sort5`: Insertion Sort