COMP 2210 Empirical Analysis Assignment

Spencer Downey

September 28, 2018

1. **Problem Overview**

This Assignment required us to determine the big-Oh time complexity of different sorting methods based on the ratio determined from data gathered from running them.

The ratio is found by dividing any time other than the first by the time before it.

1. **Experimental Procedure**

The following is an outline of the experimental procedure.

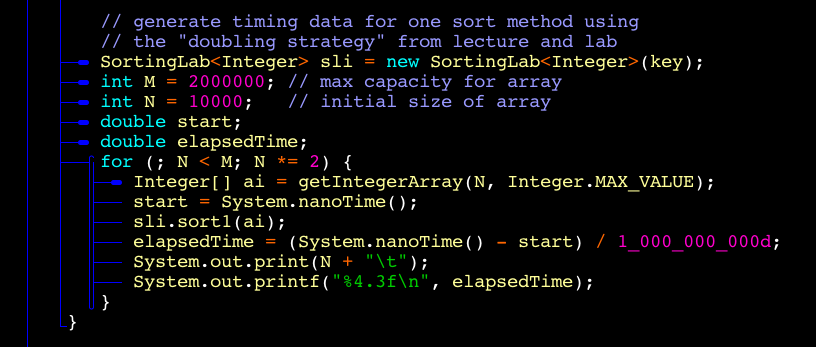
1. Collect running time data for the various sorting methods
2. Analyze the timing data to identify the ratio.
3. Use the ratio to determine the big-Oh time complexity by setting it equal to 2k and solving for k.
4. Use the big-Oh time complexity to determine which type of sorting is used.
   1. **- Experimental Materials**

One Java class is used to gather the running time data: **SortingLabClient**. This class uses a file called **resources.jar** which contains the sorting methods. We were required to use our Banner ID number as the key when instantiating a **SortingLab** object.

* 1. **– Collecting running time data**

**SortingLabClient** is used to generate the timing data. Specifically, the class records the running time required by the **SortingLab** method on problem sizes that successively doubled.

The code that generated the timing data is shown below.

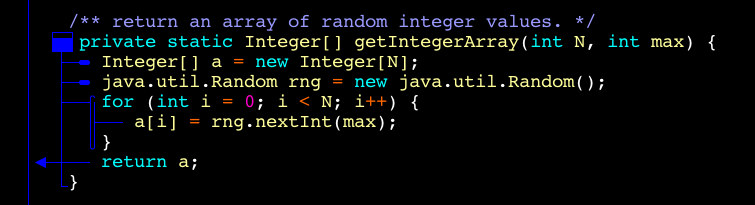


1. **Data Collection and Analysis**

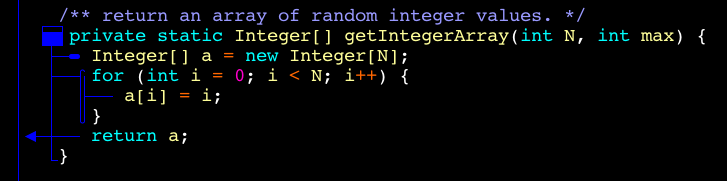
Timing data was generated by **SortingLabClient.** The computer environment in which it was run is described below.

* Computer: MacBook Air (11-inch, Mid 2013), 1.3 GHz Intel Core i5 processor, 4 GB 1600 MHz DDR3 memory
* Operating System: macOS High Sierra Version 10.13.

1. **Timing Data and Interpretation**
   1. Timing data and big-Oh time complexities were found using the base code to create an array of random numbers:



* 1. They were then compared to times and ratios found using code that created an array of continually increasing numbers that would determine the best-case time complexity for each sort:



Sort 1 – O(NlogN) both times – Merge Sort

|  |  |  |
| --- | --- | --- |
| N | Time | Range |
| 10000 | 0.011 | - |
| 20000 | 0.043 | 3.9 |
| 40000 | 0.049 | 1.13 |
| 80000 | 0.046 | 0.938 |
| 160000 | 0.119 | 2.58 |
| 320000 | 0.186 | 1.56 |
| 640000 | 0.464 | 2.49 |
| 1280000 | 0.985 | 2.12 |

* This sort was determined through both the original code and the edited code to have a time complexity of O(NlogN). Since the edited code is meant to determine the best-case time-complexity of the sorting method, and both cases were O(NlogN), this sort was determined to be Mergesort.

Sort 2 – O(NlogN) Best Case, O(N2) Worst Case – Randomized Quicksort

|  |  |  |
| --- | --- | --- |
| N | Time | Range |
| 10000 | 0.009 | - |
| 20000 | 0.045 | 5 |
| 40000 | 0.065 | 1.44 |
| 80000 | 0.042 | 0.646 |
| 160000 | 0.053 | 1.26 |
| 320000 | 0.104 | 1.96 |
| 640000 | 0.250 | 2.4 |
| 1280000 | 0.612 | 2.45 |

* Based on the times given from the original code, this sort had a time complexity of O(NlogN). The time complexity based in the edited code was O(N2). This sort was determined to be Randomized Quicksort because it was originally optimized to make the worst-case time-complexity unlikely. This was bypassed, however, when the array created in the edited code was one of increasing numbers, thus exposing that it was randomized quicksort.

Sort 3 – O(N) – Insertion Sort

|  |  |  |
| --- | --- | --- |
| N | Time | Range |
| 10000 | 0.012 | - |
| 20000 | 0.049 | 4.08 |
| 40000 | 0.133 | 2.7 |
| 80000 | 0.042 | .315 |
| 160000 | 0.112 | 2.66 |
| 320000 | 0.236 | 2.1 |
| 640000 | 0.508 | 2.15 |
| 1280000 | 1.155 | 2.27 |

* This sorting method garnered a time complexity of O(N) in both the original and edited code. The only sorting method that can have a time complexity of O(N) is Insertion Sort, so that must be what this sort is.

Sort 4 – O(N^2) both times – Non-Randomized Quicksort

|  |  |  |
| --- | --- | --- |
| N | Time | Range |
| 10000 | 0.149 | - |
| 20000 | 0.67 | 4.49 |
| 40000 | 2.740 | 4.08 |
| 80000 | 12.372 | 4.5 |
| 160000 | 57.818 | 4.67 |

* This sorting method had a time complexity of O(N2) in both the original and edited code. It was determined to be Non-Randomized Quicksort due to the fact that this version of quicksort was optimized to always expose quicksort’s worst-case time complexity. No matter what, this sort always displayed quicksort’s worst-case scenario.

Sort 5 – O(N^2) worst case, O(1) best case – Selection Sort

|  |  |  |
| --- | --- | --- |
| N | Time | Range |
| 10000 | 0.346 | - |
| 20000 | 1.004 | 2.9 |
| 40000 | 5.053 | 5.03 |
| 80000 | 21.8 | 4.31 |
| 160000 | 164.256 | 7.53 |

* This sorting method had a time complexity of O(N2) when using the original code. When using the edited code that created an array of increasing numbers (meaning it was already sorted), this sort rarely took longer than 0.004 seconds. This led to a time-complexity of roughly O(1). Considering that the array was already sorted, this sorting method was determined to be Selection Sort. The sort simply didn’t have anything to do when it was called on the increasing array.