COMP 2210 Empirical Analysis Assignment

Cameron L. Mathis

September 27, 2018

**Abstract**

I must apply what I know about the *time complexity* of each sorting algorithm, along with *stability* in the case of merge sort and quicksort, to match each sorting method with the sorting algorithm that it implements. Analysis of the data determined that **sort1** uses non-randomized Quicksort, **sort2** uses Merge Sort, **sort3** uses Insertion Sort, **sort4** uses randomized Quicksort, and **sort5** uses Selection Sort.

# Problem Overview

The assignment required us to discover the sorting algorithm for each of the **sort** methods in the **SortingLab** class. These methods take a single parameter in the form of an array. The constructor of the **SortingLab** class takes a single int parameter (the *key*) that is used to map each **sort** method to one of various different sorting algorithms. The key is my Banner ID: 903828685.

Selection sort has *O*(*N2*) time complexity for its best, average, and worst cases. Insertion sort has *O*(*N*) time complexity for its best case and has *O*(*N2*) time complexity for its average and worst cases. Non-randomized Quicksort has *O*(*N log*(*N*)) time complexity for its best and average cases has *O*(*N2*) time complexity for its worst case. Merge sort has *O*(*N log*(*N*)) time complexity for its best, average, and worst cases. Randomized Quicksort sort has *O*(*N log*(*N*)) time complexity for its best, average, and worst cases.

An array of increasing integers would invoke the best case scenario for Insertion sort and the worst case scenario for non-randomized Quicksort. An array of random integers would invoke the average case scenario for each sort algorithm.

Insertion sort and Merge sort are both stable algorithms, while Selection sort and both versions of Quicksort are unstable algorithms.

The table below allows us to record running times for *m* calls to each **sort**(**N**) method, with the size of **N** doubling on each successive call (i.e., *Ni* = 2*Ni−*1), and then calculate the ratios. This data is described in the table below, where column *N* records the problem size, column *T* (*N*) records the time required by the **sort** method on the current problem size, and column *R* is the ratio of the time required for current run to the time required for the previous run (i.e., *T* (*Ni*)*/T* (*Ni−*1)). If the values of *R* converge to 4, then the time complexity of that

specific **SortingLab** method will be *O*(*N* 2).

|  |  |  |  |
| --- | --- | --- | --- |
| *N* | *Time* | | *R* |
| *N0* | *T* (*N0*) | – | |
| *N1*  *N2* | *T* (*N1*)  *T* (*N2*) | *T* (*N1*)*/T* (*N0*)  *T* (*N2*)*/T* (*N1*) | |
| … | … | … | |
| *Ni*  *…*  *Nm* | *T* (*Ni*)  …  *T* (*Nm*) | *T* (*Ni*)*/T* (*Ni−*1)  …  *T* (*Nm*)*/T* (*Nm−*1) | |

Table 1: Description of running time data

The table below helps display what time complexity each sorting algorithm has when given different types of arrays. It also helps display where or not each sorting method is stable or unstable.

Table 2: Description of each sorting algorithm

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Algorithm* | *Sorted Array* | *Random Array* | *Stable* | |
| Selection Sort | *O(N2)* | *O(N2)* | No |
| Insertion Sort | *O(N)* | *O(N2)* | Yes |
| Merge Sort | *O(N log(N))* | *O(N log(N))* | Yes |
| Randomized Quicksort  Non-randomized Quicksort | *O(N log(N))*  *O(N2)* | *O(N log(N))*  *O(N log(N))* | No  No |

# Experimental Procedure

The following is an outline of my experimental procedure.

* 1. Collect running time data for each **sort** method when both an array of increasing integers and an array of random integers are passed through to determine time complexity, as described in Table 1.
  2. Analyze the timing data to identify if it represents *O*(*N2*) time complexity.
  3. Collect runtime data for each **sort** method to determine stability.
  4. Repeat for all five sort methods.

The following subsections discuss the experimental materials and context and elaborate on each step of the experimental procedure.

## Experimental materials

Two Java classes constitute the experimental materials: **SortingLab**, which contains the **sort** methods under study, and **SortingLabClient**, which generates timing data for the **SortingLab** methods. **SortingLabClient** implements step 1 of the experimental procedure.

We were provided with a jar file (**resources**.**jar**) that contained the **SortingLab** class. The methods under study are methods of this class. The **SortingLab** constructor takes an int parameter (the *key*) that is used to map the **SortingLab** method to a particular sorting method. I used my Banner ID number (903828685) as the key when instantiating the **SortingLab** object.

## Collecting running time data for time complexity

**SortingLabClient** was used to generate timing data as shown in the Tables above. Specifically, **SortingLabClient** records the running time required by the **SortingLab** method on problem sizes that are successively doubled.

The code that generated the timing data is shown below. The variable **ai** is an array that doubles in length after each execution. To execute **SortingLab sort1**, we would issue the following command (assuming **resources**.**jar** is in the current working directory): **sli.sort1(ai);**

The following code is how I measured how **sort1** handles an array of increasing values.

*public static void main(String[] args) {  
   
 // instantiate the SortingLab class  
 // using your Banner ID number as the key value  
 int key = 903828685;*

*// generate timing data for one sort method using  
// the "doubling strategy" from lecture and lab  
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);  
}*

*}  
 /\*\* return an array of increasing integer values. \*/  
 private static Integer[] getIntegerArray(int N, int max) {  
 Integer[] a = new Integer[N];   
 for (int i = 0; i < N; i++) {  
 a[i] = i;  
 }  
 return a;  
 }*

The following code is how we could measure how **sort1** handles an array of random values.

*public static void main(String[] args) {  
   
 // instantiate the SortingLab class  
 // using your Banner ID number as the key value  
 int key = 903828685;*

*// generate timing data for one sort method using  
// the "doubling strategy" from lecture and lab  
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);  
}*

*}*

*/\*\* 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;  
 }*

Note that, per the assignment requirements, **System**.**nanoTime**()was used to measure elapsed time.

## Analyzing running time data for time complexity

Analysis of the running time data produced by **SortingLabClient** involves computing the ratio of the running time for the current run to the running time of the previous run across all runs, and then estimating the average value of this ratio (*R* ). If the values of *R* converge to 4, the time complexity of the **sort** method will be *O*(*N* 2).

## Collecting data for stability

**SortingLabClient** was used to sort an array of with two field values. Specifically, **SortingLabClient** sorted an array by the integer field and prints the sorted array.

The code that generated the sorted array is shown below. The variable **ad** is an array with field value for string and integer. To execute **SortingLab sort1**, we would issue the following command (assuming **resources**.**jar** is in the current working directory): **sli.sort1(ai);**

*public static void main(String[] args) {  
   
 // instantiate the SortingLab class  
 // using your Banner ID number as the key value  
 int key = 903828685;*

*SortingLab<Data> sld = new SortingLab<Data>(key);  
   
 // use to determine stability  
 Data[] ad = {new Data("A", 2), new Data("A", 4), new Data("A", 5),*

*new Data("B", 2), new Data("A", 3), new Data("A", 1)};  
sld.sort1(ad);*

*for (Data element : ad) {*

*System.out.println(element);*

*}*

## Analyzing data for stability

If (A, 2) is printed before (B, 2), then the **sort** method is stable.

# Data Collection and Analysis

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

* Computer: MacBook Pro (13-inch Mid 2012), 2.5GHz Intel Core i5 processor, 16GB 1600 MHz DDR3 memory
* Operating system: OS X 10.13.6
* Java: 9.0.1
  + **jGRASP** -**version**: 2.0.5\_01
  + **java** -**version**: java version “9.0.1”
  + **System**.**nanoTime**() used to measure elapsed time

## 3.1 Timing Data

Timing data was generated by running **SortingLabClient** with array sizes 10000 to 1280000 inclusive. Each method is first called with an array of increasing values and then called again with an array of random values. The outputs of each method call is shown below. All times are reported in seconds.

**Method:** *sli.sort1(ai); //increasing*

**Run N Time**

1 10000 0.157

2 20000 0.437

3 40000 1.694

4 80000 7.387

5 160000 29.333

6 320000 121.146

7640000 482.950

8 1280000 2091.039

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 3: Running time data with an increasing array

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.157 | - |
| 20000 | 0.437 | 2.783 |
| 40000 | 1.694 | 3.876 |
| 80000 | 7.387 | 4.361 |
| 160000  320000  640000  1280000 | 29.333  121.146  482.950  2091.039 | 3.971  4.130  3.987  4.329 |

It is clear that *R* ≈ 4. Therefore, **sort1** has *O*(*N2*) time complexity when given an array of increasing values.

**Method:** *sli.sort1(ai); //random*

**Run N Time**

1 10000 0.009

2 20000 0.041

3 40000 0.037

4 80000 0.022

5 160000 0.050

6 320000 0.101

7640000 0.225

8 1280000 0.566

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 4: Running time data with a random array

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.009 | - |
| 20000 | 0.041 | 4.556 |
| 40000 | 0.037 | 0.902 |
| 80000 | 0.022 | 0.595 |
| 160000  320000  640000  1280000 | 0.050  0.101  0.225  0.566 | 2.273  2.020  2.228  2.471 |

It is clear that *R* !≈ 4. Therefore, **sort1** has *O*(*N log*(*N*)) time complexity when given an array of random values.

**Method:** *sli.sort2(ai); //increasing*

**Run N Time**

1 10000 0.009

2 20000 0.051

3 40000 0.062

4 80000 0.021

5 160000 0.074

6 320000 0.151

7640000 0.324

8 1280000 0.752

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 5: Running time data with an increasing array

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.009 | - |
| 20000 | 0.051 | 5.667 |
| 40000 | 0.062 | 1.216 |
| 80000 | 0.021 | 0.339 |
| 160000  320000  640000  1280000 | 0.074  0.151  0.324  0.752 | 3.524  2.041  2.146  2.321 |

It is clear that *R* !≈ 4. Therefore, **sort2** has *O*(*N log*(*N*)) time complexity when given an array of increasing values.

**Method:** *sli.sort2(ai); //random*

**Run N Time**

1 10000 0.008

2 20000 0.088

3 40000 0.128

4 80000 0.042

5 160000 0.114

6 320000 0.282

7640000 0.545

8 1280000 1.123

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 6: Running time data with a random array

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.008 | - |
| 20000 | 0.088 | 11.000 |
| 40000 | 0.128 | 1.455 |
| 80000 | 0.042 | 0.328 |
| 160000  320000  640000  1280000 | 0.114  0.282  0.545  1.123 | 2.714  2.474  1.933  2.061 |

It is clear that *R* !≈ 4. Therefore, **sort2** has *O*(*N log*(*N*)) time complexity when given an array of random values.

**Method:** *sli.sort3(ai); //increasing*

**Run N Time**

1 10000 0.001

2 20000 0.002

3 40000 0.004

4 80000 0.003

5 160000 0.002

6 320000 0.007

7640000 0.003

8 1280000 0.005

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 7: Running time data with an increasing array

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.001 | - |
| 20000 | 0.002 | 2.000 |
| 40000 | 0.004 | 2.000 |
| 80000 | 0.003 | 0.750 |
| 160000  320000  640000  1280000 | 0.002  0.007  0.003  0.005 | 0.667  3.500  0.429  1.667 |

It is clear that the time is constant. Therefore, **sort3** has *O*(*N*) time complexity when given an array of increasing values.

**Method:** *sli.sort3(ai); //random*

**Run N Time**

1 10000 0.237

2 20000 1.111

3 40000 3.739

4 80000 20.439

5 160000 157.787

6 320000 1042.278

7640000 -

8 1280000 -

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 8: Running time data with a random array

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.237 | - |
| 20000 | 1.111 | 4.678 |
| 40000 | 3.739 | 3.410 |
| 80000 | 20.439 | 5.466 |
| 160000  320000  640000  1280000 | 157.787  1042.278  -  - | 7.719  6.606  -  - |

Even though I did not record the time for 640000 or 1280000 (because it would have taken over an hour for both), it is clear that *R* ≈ 4. Therefore, **sort3** has *O*(*N2*) time complexity when given an array of random values.

**Method:** *sli.sort4(ai); //increasing*

**Run N Time**

1 10000 0.011

2 20000 0.019

3 40000 0.038

4 80000 0.058

5 160000 0.075

6 320000 0.148

7640000 0.280

8 1280000 0.836

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 9: Running time data with an increasing array

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.011 | - |
| 20000 | 0.019 | 1.727 |
| 40000 | 0.038 | 2.000 |
| 80000 | 0.058 | 1.526 |
| 160000  320000  640000  1280000 | 0.075  0.148  0.280  0.836 | 1.293  1.973  1.892  2.986 |

It is clear that *R* !≈ 4. Therefore, **sort4** has *O*(*N log*(*N*)) time complexity when given an array of increasing values.

**Method:** *sli.sort4(ai); //random*

**Run N Time**

1 10000 0.021

2 20000 0.016

3 40000 0.043

4 80000 0.045

5 160000 0.119

6 320000 0.165

7640000 0.402

8 1280000 0.977

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 10: Running time data with a random array

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.021 | - |
| 20000 | 0.016 | 0.762 |
| 40000 | 0.043 | 2.688 |
| 80000 | 0.045 | 1.047 |
| 160000  320000  640000  1280000 | 0.119  0.165  0.402  0.977 | 2.644  1.387  2.436  2.430 |

It is clear that *R* !≈ 4. Therefore, **sort4** has *O*(*N log*(*N*)) time complexity when given an array of random values.

**Method:** *sli.sort5(ai); //increasing*

**Run N Time**

1 10000 0.133

2 20000 0.500

3 40000 1.428

4 80000 8.158

5 160000 32.827

6 320000 137.066

7640000 544.133

8 1280000 2506.833

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 11: Running time data with an increasing array.

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.133 | - |
| 20000 | 0.500 | 3.759 |
| 40000 | 1.428 | 2.856 |
| 80000 | 8.158 | 5.713 |
| 160000  320000  640000  1280000 | 32.827  137.066  544.133  2506.833 | 4.024  4.175  3.969  4.607 |

It is clear that *R* ≈ 4. Therefore, **sort5** has *O*(*N2*) time complexity when given an array of increasing values.

**Method:** *sli.sort5(ai); //random*

**Run N Time**

1 10000 0.176

2 20000 0.724

3 40000 2.118

4 80000 10.814

5 160000 58.447

6 320000 379.133

7640000 2037.079

8 1280000 –

The table below shows the raw data along with computations of each running time ratio (column *R*).

Table 12: Running time data with a random array.

|  |  |  |
| --- | --- | --- |
| *N* | *Time* | *R* |
| 10000 | 0.176 | - |
| 20000 | 0.724 | 4.114 |
| 40000 | 2.118 | 2.925 |
| 80000 | 10.814 | 5.106 |
| 160000  320000  640000  1280000 | 58.447  379.133  2037.079  - | 5.405  6.487  5.373  - |

Even though I did not record the time for 1280000 (because it would have taken over an hour), it is clear that *R* ≈ 4. Therefore, **sort5** has *O*(*N2*) time complexity when given an array of random values.

## 3.2 Stability Data

Stability data was generated by running each **sort** method with array of data [(“A”, 2), (“A”, 4), (“A”, 5), (“B”, 2), (“A”, 3), (“A”, 1)]. The array was sorted by integer value. The outputs of each method call is shown below.

**Method:** *sli.sort1(ad); //data*

(A, 1)

(B, 2)

(A, 2)

(A, 3)

(A, 4)

(A, 5)

(A, 2) did not remain before (B, 2) during sorting. Therefore, **sort1** is unstable.

**Method:** *sli.sort2(ad); //data*

(A, 1)

(A, 2)

(B, 2)

(A, 3)

(A, 4)

(A, 5)

(A, 2) did remain before (B, 2) during sorting. Therefore, **sort2** is stable.

**Method:** *sli.sort3(ad); //data*

(A, 1)

(A, 2)

(B, 2)

(A, 3)

(A, 4)

(A, 5)

(A, 2) did remain before (B, 2) during sorting. Therefore, **sort3** is stable.

**Method:** *sli.sort4(ad); //data*

(A, 1)

(B, 2)

(A, 2)

(A, 3)

(A, 4)

(A, 5)

(A, 2) did not remain before (B, 2) during sorting. Therefore, **sort4** is unstable.

**Method:** *sli.sort5(ad); //data*

(A, 1)

(B, 2)

(A, 2)

(A, 3)

(A, 4)

(A, 5)

(A, 2) did not remain before (B, 2) during sorting. Therefore, **sort5** is unstable.

# Interpretation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Algorithm* | *Sorted Array* | *Random Array* | *Stable* | |
| sort1 | *O(N2)* | *O(N log(N))* | No |
| sort2 | *O(N log(N))* | *O(N log(N))* | Yes |
| sort3 | *O(N)* | *O(N2)* | Yes |
| sort4  sort5 | *O(N log(N))*  *O(N2)* | *O(N log(N))*  *O(N2)* | No  No |

The method **sort1** has *O*(*N2*) time complexity for the sorted array, *O(N log(N))* for the random array, and is unstable, thus ituses the non-randomized Quicksort algorithm.

The method **sort2** has *O(N log(N))* time complexity for the sorted array, *O(N log(N))* for the random array, and is unstable, thus ituses the Merge algorithm.

The method **sort3** has *O*(*N*) time complexity for the sorted array, *O*(*N2*) for the random array, and is unstable, thus ituses the Insertion algorithm.

The method **sort4** has *O(N log(N))* time complexity for the sorted array, *O(N log(N))* for the random array, and is unstable, thus ituses the randomized Quicksort algorithm.

The method **sort5** has *O*(*N2*) time complexity for the sorted array, *O*(*N2*) for the random array, and is unstable, thus ituses the Selection algorithm.