COMP 2210 Assignment 3 Part B

Group 38

Robinson Davis Steven Han

February 25, 2014

Abstract

Part B of activity three consisted of observing five sorting algorithms in the provided SortingLab class and determining which type of sort they represent. Without the source code being available one must derive an accurate, repeatable experiment in order to determine this. Many of the tests relate closely to the one in Part A. However as these algorithms have the potential to possess the same time complexity as their counterparts, other information must be recorded and recorded, i.e. stability. Once all data was collected, recorded, and analyzed one should find that sort1() represents selection sort, sort2() represents a non-randomized quicksort, sort3() represents merge sort, sort4() represents insertion sort, and sort5() represents a randomized quicksort.

1 Problem Overview

One is challenged with not only the task of discovering which sort method represents which sort algorithm but also developing a repeatable test for another person to do so. The experimental procedure in Part A can be used as a stepping stone to accomplish this, as it provides an ample way to find the big-Oh time complexity of each algorithm without seeing source code. However, as stated in abstract, these algorithms may have the same time complexity under certain cases be it best, average, or worse. Therefore one must develop a new test for Part B that will distinguish each algorithm with the same time complexity from one another. For this new test, stability must be recorded and analyzed. There are also more tests to be ran, and more data collected. Simply observing and recording jGRASP output would be inefficient. Microsoft Excel provides excellent tools, such as Excel's built in functions, to record and calculate such data in the form of an easy to read table.

2 Experimental Procedure

These experiments were conducted on a PC with the following specifications:

- Operating System: Windows 8.1 64-bit
- Processor: Intel i7-4700MQ CPU
- RAM: 8GB DDR3L SDRAM
- Java Version: 1.7.0_51

The first step was modifying the provided code to meet computer specification limitations. There were two things that needed to be done. The code provided contains a bit of code that allows iteration through k arrays, which doubles at each iteration. Code to test a randomized array is available for utilization in SortingLabClient.java. Minimum and maximum array size values are also provided, but these are not accurate for all machines. One computer may have a runtime above zero seconds, crucial for information analysis, with an array of one hundred elements while others may not provide a runtime above zero seconds until an array contains one hundred thousand elements. These limitations were changed and a method was

added in order to also test the time complexity of an array that is already sorted in natural order. The code that allows data output and collection is shown in Figure 2(a).

```
// create sorting lab for ints
// also set min/max values for array size
SortingLab<Integer> slint = new SortingLab<>(key);
int M = 10000000; // max capacity for array
int N = 100;  // initial size of array
System.out.println("N Value\t\tRuntime");
//loop to double n each time until max size is reached
for (; N < M; N *= 2) {
// variables
   Integer[] randomArray = getIntegerArray(N, Integer.MAX_VALUE);
   Integer[] sortedArray = sortedArray(N);
   Clock clock = new Clock();
   // sort and calculate time
   slint.sort1(randomArray);
   elapsedTime = clock.elapsedTime();
   // print info
   System.out.print(N + "\t\t");
   System.out.printf("%4.3f\n", elapsedTime);
   ratio = elapsedTime/prevTime;
   prevTime = elapsedTime;
}
```

Figure 2(a) – Source code that provided the data collection for time complexity.

The line Integer[] randomArray = getIntegerArray(N, Integer.MAX_VALUE); is provided for users already and creates an array with size N of random elements. The source for it can be found in Figure 2 (b). It takes a parameter of N for length and max as the maximum number each element may be. The succeeding line was an addition to the provided code, and calls the added method that creates an integer array in natural order. This method takes a parameter N representing the desired size of sortedArray. The source code for it can be found in Figure 2(c). Observation of the source shows the array begins with one followed by powers of two until the desired length is acquired. Note that this is the method called by the Integer[] sortedArray = sortedArray(N); line in Figure 2(a).

```
// method to create an array of random integers
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;
}
```

Figure 2(b) – Method to create an array of integers with random values at each index

```java
// method to create an array in ascending order
private static Integer[] sortedArray(int length) {
    Integer[] sortedArray = new Integer[length];
    for (int i = 0; i < length; i++) {
        if (i != 0) {
            sortedArray[i] = i * 2;
        }
        else {
            sortedArray[i] = 1;
        }
    }
    return sortedArray;
}
```

Figure 2(c) – Method that creates an array of N length with elements in natural order

To test for stability another new method was also added to SortingLabClient.java, and an entire new class was created. This new class simply creates an element that has symbolic representation as (x, y) where x and y are any real numbers. This is useful because a stable sort algorithm will sort (x, y) keeping each element that has the same x value, or duplicates, stay in the same relative order. An unstable algorithm may randomize these elements. An example for a stable algorithm is provided in Figure 2(d) and unstable in Figure 2(e).

```
----jGRASP exec: java -ea SortingLabClient

Array Randomized:
[(4, 4), (0, 4), (4, 0), (5, 3), (0, 3), (2, 5), (3, 1), (1, 4), (4, 4), (1, 3)]

Array Sorted:
[(0, 4), (0, 3), (1, 4), (1, 3), (2, 5), (3, 1), (4, 4), (4, 0), (4, 4), (5, 3)]

----jGRASP: operation complete.
```

Figure 2(d) – Example of a stable sort. [(0, 4) and (0, 3) stay in relative order.]

```
----jGRASP exec: java -ea SortingLabClient

Array Randomized:
[(3, 1), (5, 3), (1, 4), (4, 3), (1, 1), (4, 3), (3, 3), (4, 0), (4, 0), (4, 5)]

Array Sorted:
[(1, 1), (1, 4), (3, 3), (3, 1), (4, 3), (4, 0), (4, 0), (4, 5), (4, 3), (5, 3)]

----jGRASP: operation complete.
```

Figure 2(e) – Example of a non-stable sort. [(1, 1) and (1, 4) are not in relative order.]

IntPair.java contains the source that builds each element in the above while the method to create an array of IntPair objects was added in the provided SortingLabClient.java. Figure 2(f) shows the source of IntPair.java while Figure 2(g) shows the source of the method that builds the array. Notice IntPair.java implements Compareable<IntPair>. This allows the sorting methods provided the ability to be called on the created array of IntPair objects.

```java
public class IntPair implements Comparable<IntPair> {

// declerations
 - int firstInt;
 - int secondInt;
// constructor
  public IntPair(int int1, int int2) {
    firstInt = int1;
    secondInt = int2;
  }
// override compareTo
  public int compareTo(IntPair inputArray) {
    return this.firstInt - inputArray.firstInt;
  }
// create a toString
  public String toString() {
    String output = "";
    output += ("(" + firstInt + ", " + secondInt + ")");
    return output;
  }

}
```

Figure 2(f) – IntPair.java source.

```java
  //method to create IntPair array
  private static IntPair[] createIntPair(int length) {
    IntPair[] createdArray = new IntPair[length];
    java.util.Random rng = new java.util.Random();
    for (int i = 0; i < length; i++) {
      createdArray[i] = new IntPair(rng.nextInt(6), rng.nextInt(6));
    }
    return createdArray;
  }
```

Figure 2(g) – Method to create an array of IntPair objects

The final step was to create a snippet of code that would call createIntPair(), print out the returned array, sort the returned array, and then print it again sorted. This code is shown in Figure 2(h). Note the final line is System.exit(0);. The code in Figure 2(h) executes before the code shown in Figure 2(a) as they are both in SortingLabClient.java. System.exit(0) provides a way of stopping the program from outputting dozens of lines from the code in Figure 2(a) that would be unhelpful in finding stability. This could be done by "commenting out" Figure 2(a) code, but this would be tedious and time consuming. System.exit(0) means only one line of code to be commented or uncommented as the output from Figure 2(h) does not print enough lines to make observing the output of Figure 2(a) difficult.

```
// create array to test stability, and sorting lab for IntPair
IntPair[] stabilityTestArray = createIntPair(10);
SortingLab<IntPair> slobj = new SortingLab<>(key);

// print the randomly generated array
System.out.println("Array Randomized: ");
System.out.println(java.util.Arrays.toString(stabilityTestArray));
System.out.println("");
// sort array
slobj.sort1(stabilityTestArray);
// print sorted array
System.out.println("Array Sorted: ");
System.out.println(java.util.Arrays.toString(stabilityTestArray));
// exit program
System.exit(0);
```

Figure 2(h) – Code that allows the testing of stability.

Now that code allowing for all needed data to be collected has been created, the process of actually collecting and recording the data can be performed. When the data has been recorded it can be analyzed and each sort algorithm can be deciphered.

3 Data Collection and Analysis 4

To analyze any data it must be known which time complexities and stability data represents which sort algorithm. This can be found in Figure 3(a).

	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?	No	Yes	Yes	No
Adaptive?	No	Yes	No	No

Figure 3(a) – Table indicating the time complexity of each sort algorithm and whether or not it is stable.

Data collection for time complexity meant nothing more than executing the code in Figure 2(a) with only minor alterations for reach test. The line in Figure 2(a) which calls the sort method was changed each time to represent the current sort method being tested. The parameter was also modified each test to allow each sort method to be called on a randomized array (created by the method show in Figure 2(b)) and an array in natural order (created by the method shown in Figure 2(c)). Again, this code was executed five

times for each test in order to confirm consistency. Once consistency was confirmed the final test data for each sort method on a randomized and nonrandomized array was recorded and stored for analyzing later. The result for each run can found in the figures below. Please note that each column labeled "Log Ratio" is actually the $\log_2$ of each ratio, and not $\log_{10}$.

	A	B	C	D	E	F	G	H	I
1									
2	Random(Average) For Sort 1					Array sorted Increasing For Sort 1			
3									
4	N Value	Run Time	Ratio	Log Ratio		N Value	Run Time	Ratio	Log Ratio
5	6400	0.029				6400	0.025		
6	12800	0.119	4.103448276	2.036836768		12800	0.101	4.04	2.014355293
7	25600	0.564	4.739495798	2.244733589		25600	0.402	3.98019802	1.992840208
8	51200	2.536	4.496453901	2.168787678		51200	1.607	3.997512438	1.999102522
9	102400	10.863	4.28351735	2.098795932		102400	6.423	3.996888612	1.998877367
10	204800	46.615	4.291171868	2.101371684		204800	25.486	3.96792776	1.98838576

Figure 3(b) – Time complexity data for sort1()

	A	B	C	D	E	F	G	H	I
1									
2	Random(Average) For Sort 2					Array sorted Increasing For Sort 2			
3									
4	N Value	Run Time	Ratio	Log Ratio		N Value	Run Time	Ratio	Log Ratio
5	200000	0.03				800	0.013		
6	400000	0.071	2.366666667	1.242856524		1600	0.01	0.769230769	-0.378511623
7	800000	0.161	2.267605634	1.181169759		3200	0.004	0.4	-1.321928095
8	1600000	0.361	2.242236025	1.164938149		6400	0.015	3.75	1.906890596
9	3200000	0.924	2.559556787	1.355894015		12800	0.063	4.2	2.070389328
10	6400000	2.154	2.331168831	1.221053493		25600	0.259	4.111111111	2.039528364

Figure 3(c) – Time complexity data for sort2()

	A	B	C	D	E	F	G	H	I
1									
2	Random(Average) For Sort 3					Array sorted Increasing For Sort 3			
3									
4	N Value	Run Time	Ratio	Log Ratio		N Value	Run Time	Ratio	Log Ratio
5	204800	0.04				204800	0.017		
6	409600	0.092	2.3	1.201633861		409600	0.04	2.352941176	1.234465254
7	819200	0.218	2.369565217	1.244622369		819200	0.085	2.125	1.087462841
8	1638400	0.496	2.275229358	1.186011986		1638400	0.176	2.070588235	1.050040682
9	3276800	1.184	2.387096774	1.255257055		3276800	0.416	2.363636364	1.2410081
10	6553600	2.555	2.157939189	1.10965421		6553600	0.682	1.639423077	0.713188211

Figure 3(d) – Time complexity data for sort3()

	A	B	C	D	E	F	G	H	I
1									
2	Random(Average) For Sort 4					Array sorted Increasing For Sort 4			
3									
4	N Value	Run Time	Ratio	Log Ratio		N Value	Run Time	Ratio	Log Ratio
5	6400	0.021				204800	0.001		
6	12800	0.086	4.095238095	2.033947332		409600	0.001	1	0
7	25600	0.384	4.465116279	2.158697746		819200	0.003	3	1.584962501
8	51200	1.755	4.5703125	2.192292814		1638400	0.004	1.333333333	0.415037499
9	102400	7.477	4.26039886	2.090988503		3276800	0.009	2.25	1.169925001
10	204800	31.063	4.154473719	2.054665731		6553600	0.014	1.555555556	0.637429921

Figure 3(e) – Time complexity data for sort3()

	A	B	C	D	E	F	G	H	I
1									
2	Random(Average) For Sort 5					Array sorted Increasing For Sort 5			
3									
4	N Value	Run Time	Ratio	Log Ratio		N Value	Run Time	Ratio	Log Ratio
5	204800	0.044				204800	0.037		
6	409600	0.112	2.545454545	1.347923303		409600	0.092	2.486486486	1.31410859
7	819200	0.291	2.598214286	1.377520421		819200	0.242	2.630434783	1.395301281
8	1638400	0.705	2.422680412	1.276604104		1638400	0.517	2.136363636	1.095157233
9	3276800	1.631	2.313475177	1.210061619		3276800	1.192	2.305609284	1.20514805
10	6553600	3.782	2.318822808	1.213392581		6553600	2.658	2.229865772	1.156956869

Figure 3(f) – Time complexity data for sort5()

The first set of data labeled "Random(Average) for Sort X" was conducted by calling each sort method on a randomized array (created by calling the method in Figure 2(b). This was effective for testing the average case of each sort algorithm. The set of data labeled "Array sorted Increasing For Sort X" was conducted by calling each sort method on an array in natural order (created by calling the method in Figure 2(c)). This was generally useful for testing "best" cases, although this was not true for the randomized quicksort. Randomized quicksort always permuted the elements in the parameter array which made the worst case of quicksort probabilistically unlikely. Non-randomized quicksort however did not permute elements in the parameter array which exposes quicksort's worst case. Also not that many "N Value" columns had differing values, this is the same concept explained in the first paragraph of section two. Some sorting algorithms ran too close to 0 seconds to get accurate data. Therefore the N Values were modified where applicable to gather meaningful timing data. This does not affect time complexity.

From the data above we can make multiple observations about each sort method:

- Sort 1 has a time complexity of $O(N^2)$ both a randomized and natural order array.
- Sort 2 has a time complexity of O(N log N) for a randomized array, and a time complexity of $O(N^2)$ for a natural order array.
- Sort 3 has a time complexity of O(N log N) both a randomized and natural order array.
- Sort 4 has a time complexity of $O(N^2)$ for a randomized array and a time complexity of O(N) for a natural order array.
- Sort 5 has a time complexity of O(N log N) both a randomized and natural order array.

With this information, the following can be derived:

- Sort 1 = Selection.
- Sort 2 = Non-randomized quicksort.
- Sort 3 = Sort 5 (unknown).
- Sort 4 = Insertion.
- Sort 5 = Sort 3 (unknown)

As sort3() and sort5() have the same time complexity, the sort algorithms randomized quicksort and merge are left to be found. How can they be differed from each other? The answer is stability. Shown below are figures representing stability tests for each sort algorithm. The figures are in numerical order (i.e. sort1(), sort2(), sort3(), etc.).

```
----jGRASP exec: java -ea SortingLabClient

Array Randomized:
[(1, 5), (4, 4), (2, 1), (4, 4), (1, 3), (1, 5), (4, 1), (4, 1), (4, 5), (0, 2)]

Array Sorted:
[(0, 2), (1, 3), (1, 5), (1, 5), (2, 1), (4, 4), (4, 1), (4, 1), (4, 5), (4, 4)]

----jGRASP: operation complete.
```

Figure 3(g) – Stability results for sort1().

```
----jGRASP exec: java -ea SortingLabClient

Array Randomized:
[(1, 5), (3, 4), (5, 2), (2, 3), (1, 2), (0, 0), (1, 3), (5, 2), (3, 1), (1, 4)]

Array Sorted:
[(0, 0), (1, 3), (1, 2), (1, 4), (1, 5), (2, 3), (3, 4), (3, 1), (5, 2), (5, 2)]

----jGRASP: operation complete.
```

Figure 3(h) – Stability results for sort2().

```
----jGRASP exec: java -ea SortingLabClient

Array Randomized:
[(4, 4), (0, 4), (4, 0), (5, 3), (0, 3), (2, 5), (3, 1), (1, 4), (4, 4), (1, 3)]

Array Sorted:
[(0, 4), (0, 3), (1, 4), (1, 3), (2, 5), (3, 1), (4, 4), (4, 0), (4, 4), (5, 3)]

----jGRASP: operation complete.
```

Figure 3(i) – Stability results for sort3(). *This is also the same figure as Figure 2(d).*

```
----jGRASP exec: java -ea SortingLabClient

Array Randomized:
[(0, 2), (0, 4), (5, 3), (3, 2), (1, 4), (1, 4), (4, 2), (4, 0), (2, 3), (2, 4)]

Array Sorted:
[(0, 2), (0, 4), (1, 4), (1, 4), (2, 3), (2, 4), (3, 2), (4, 2), (4, 0), (5, 3)]

----jGRASP: operation complete.
```

Figure 3(j) – Stability results for sort4().

```
----jGRASP exec: java -ea SortingLabClient

Array Randomized:
[(3, 1), (5, 3), (1, 4), (4, 3), (1, 1), (4, 3), (3, 3), (4, 0), (4, 0), (4, 5)]

Array Sorted:
[(1, 1), (1, 4), (3, 3), (3, 1), (4, 3), (4, 0), (4, 0), (4, 5), (4, 3), (5, 3)]

----jGRASP: operation complete.
```

Figure 3(k) – Stability results for sort5(). *This is also the same as Figure 2(e).*

So now we have stability information on each sort algorithm:

- Sort 1: Not stable.
- Sort 2: Not stable.
- Sort 3: Stable.
- Sort 4: Stable.
- Sort 5: Not stable.

There are no differences in the time complexity between sort3() and sort5(). Now, however, we see that sort3() is a stable algorithm and sort5() is a non-stable algorithm. This states that sort3() must be merge, and sort5() must be randomized quicksort. With this information each sort method can now be deciphered and displayed. Observing and comparing time complexities and stability each sort method can correctly be labeled as:

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

4 Interpretation

The goal was to decipher five sort algorithms. By finding time complexity and stability for each of the sort methods, this goal was accomplished. The time complexities exposed three search algorithms, sort1() as selection, sort2() as non-randomized quicksort, and sort4() as insertion. Time complexities did not, however, expose the algorithm of sort3() and sort5() as they shared the same time complexity. Stability tests answered this problem, showing sort3() to be stable and sort5() to be not stable. Knowing that merge and randomized quicksort were the remaining options and the fact that merge is sable and quicksort is now, they were able to be differed from each other and accurately labeled.