COMP 2210 Empirical Analysis Assignment – Part B

Maxwell Heeschen

September 24, 2016

**Abstract**

The purpose of this experiment was to find which sorting algorithm went with each unknown sort method. This was done by collecting timing data for each of the sorting methods. It was found that **sort1** was **insertion sort**, **sort2** was **selection sort**, **sort3** was the **non-random quicksort**, **sort4** was the **random quicksort**, and **sort5** was the **merge sort**.

1. Problem Overview

The problem that I am trying to solve is that I have an unknown collection of sorting methods labeled sort1-sort5, and I have to determine which sorting algorithm is implemented by each of the five methods. I know that, as implemented in class, selection sort will always have time complexity O(N2) and it is not stable. Insertion sort will have time complexity O(N) in the best case, O(N2) in the worst case, and is stable. Merge sort will always have time complexity O(N log N) and it is stable. Finally, quick sort has time complexity O(N log N) in the best case, O(N2) in the worst case, and it is not stable. I also know that the randomized quick sort method is very unlikely to have a worst case because it randomly permutes the array elements before the sorting starts. I am assuming that each of the sorting methods, except the quick sorts, are implemented in the exact same way as in lecture. The time complexity can be found with the following equation:

T(N)∝ Nk =⇒ T(2N)/T(N) ∝ (2N)­k/Nk = 2kNk/Nk = 2k

Where k is used as the exponent for N in the notation O(Nk).

1. Experimental Procedure

The system specs that I used were OS: Windows 10 Home 64-bit, Processor: AMD FX-8150 Eight-Core Processor ~3.6GHz, and Memory: 8192 MB. I used my Banner ID as the key which is 903532323. I started out by writing a method that could give me an array of size N with the elements in the array ordered up to N-1 so that I could test the best case for each of the sorting methods. I called it getSortedIntArray. Then I made a method that did what getSortedIntArray did but with the elements in reverse order, and I called it getReverseArray. After that I made a section of the main method to test one sort method with increasing array sizes starting with 10000 and going to 10240000, doubling each time:

System.out.println("Using already sorted arrays\n\nN\t\tTime\t\t\tR");  
 for(; N < M; N \*= 2)  
 {  
 Integer[] sorted = getSortedIntArray(N);  
 start = System.nanoTime();  
 sli.sort3(sorted);  
 elapsedTime = (System.nanoTime() - start) / 1000000000d;  
 System.out.print(N + "\t");  
 System.out.printf("%4.3f\t\t", elapsedTime);  
 System.out.printf("%4.3f", (elapsedTime/prevElapsedTime));  
 System.out.println();  
 prevElapsedTime = elapsedTime;  
 }

I outputted the results of each of these runs with N the size of the array, Time the time it took the method to sort, and R the ratio of the current time with the time that the previous run took. Then I did the same using the reversed arrays method on the same sort method; I also reset the value of N to 10000:

N = 10000;  
System.out.println("\nUsing reverse sorted arrays\n\nN\t\tTime\t\t\tR");  
for(; N < M; N \*= 2)  
{  
 Integer[] sorted = getReverseArray(N);  
 start = System.nanoTime();  
 sli.sort3(sorted);  
 elapsedTime = (System.nanoTime() - start) / 1000000000d;  
 System.out.print(N + "\t");  
 System.out.printf("%4.3f\t\t", elapsedTime);  
 System.out.printf("%4.3f",

(elapsedTime/prevElapsedTime));  
 System.out.println();  
 prevElapsedTime = elapsedTime;  
}

Finally, I did the same, finding the average cases for the same sort method:

N = 10000;  
System.out.println("\nAverage cases\n\nN\t\tTime\t\t\tR");  
for (; N < M; N \*= 2) {  
Integer[] ai = getIntegerArray(N, Integer.MAX\_VALUE);  
start = System.nanoTime();  
sli.sort3(ai);  
elapsedTime = (System.nanoTime() - start) / 1000000000d;  
System.out.print(N + "\t");  
System.out.printf("%4.3f\t\t", elapsedTime);  
System.out.printf("%4.3f", (elapsedTime/prevElapsedTime));  
System.out.println();  
prevElapsedTime = elapsedTime;  
}

I did this for each of the sort methods, changing the source code to sort1-sort5. I then made a class called Test that took 2 variables to test the stability of the sort methods:

class Test implements Comparable<Test>  
{  
  
private int hold;  
private int something;  
  
public Test(int A, int B)  
{  
hold = A;  
something = B;  
}  
  
  
public int compareTo(Test that)  
{  
if(this.hold < that.hold)  
{  
return -1;  
}  
else if (this.hold > that.hold)  
{  
return 1;  
}  
else   
return 0;  
}  
}

I then used it in the SortingLabClient class like so:

SortingLab<Test> tsi = new SortingLab<Test>(key);  
Test[] stable = {new Test(1, 4), new Test(1, 1), new Test(2, 7), new Test(2, 2)};  
tsi.sort4(stable);

I didn’t use it until I was down to two unknown sort methods.

1. Data Collection and Analysis

I first ran my custom SortingLabClient.java file on the sort1 method to try to figure out the time complexity. N is the number of elements in an array, Time is the time the method took, and R is the ratio between the current run and the previous run.

**Sort1**

|  |  |  |
| --- | --- | --- |
| Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.005 | -- |
| 20000 | 0.001 | 0.267 |
| 40000 | 0.000 | 0.116 |
| 80000 | 0.000 | 2.493 |
| 160000 | 0.001 | 1.641 |
| 320000 | 0.005 | 7.384 |
| 640000 | 0.003 | 0.594 |
| 1280000 | 0.005 | 1.573 |
| 2560000 | 0.011 | 2.361 |
| 5120000 | 0.027 | 2.447 |
| 10240000 | 0.038 | 1.403 |

|  |  |  |
| --- | --- | --- |
| Reverse Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.000 | -- |
| 20000 | 0.000 | 1.968 |
| 40000 | 0.000 | 2.000 |
| 80000 | 0.000 | 2.861 |
| 160000 | 0.001 | 2.062 |
| 320000 | 0.001 | 1.466 |
| 640000 | 0.003 | 3.289 |
| 1280000 | 0.005 | 1.357 |
| 2560000 | 0.009 | 1.999 |
| 5120000 | 0.019 | 2.087 |
| 10240000 | 0.037 | 1.949 |

|  |  |  |
| --- | --- | --- |
| Average Cases | | |
| N | Time | R |
| 10000 | 0.210 | -- |
| 20000 | 0.943 | 4.492 |
| 40000 | 3.965 | 4.207 |
| 80000 | 16.609 | 4.189 |
| 160000 | 77.951 | 4.693 |
| 320000 | 436.990 | 5.606 |

R

Time

**Sort2**

|  |  |  |
| --- | --- | --- |
| Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.175 | -- |
| 20000 | 0.619 | 3.537 |
| 40000 | 1.342 | 2.168 |
| 80000 | 5.365 | 3.998 |
| 160000 | 24.509 | 4.568 |
| 320000 | 107.160 | 4.372 |
| 640000 | 551.501 | 5.147 |

|  |  |  |
| --- | --- | --- |
| Reverse Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.179 | -- |
| 20000 | 0.666 | 3.740 |
| 40000 | 1.372 | 2.062 |
| 80000 | 5.427 | 3.955 |
| 160000 | 24.872 | 4.583 |
| 320000 | 110.365 | 4.437 |

|  |  |  |
| --- | --- | --- |
| Average Cases | | |
| N | Time | R |
| 10000 | 0.198 | -- |
| 20000 | 0.752 | 3.800 |
| 40000 | 2.155 | 2.866 |
| 80000 | 9.265 | 4.300 |
| 160000 | 54.049 | 5.834 |
| 320000 | 258.462 | 4.912 |

R

Time

**Sort3**

|  |  |  |
| --- | --- | --- |
| Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.073 | -- |
| 20000 | 0.233 | 3.198 |
| 40000 | 0.888 | 3.818 |
| 80000 | 2.241 | 2.525 |
| 160000 | 8.219 | 3.667 |
| 320000 | 17.805 | 2.166 |
| 640000 | 54.703 | 3.072 |
| 1280000 | 111.011 | 2.029 |
| 2560000 | 285.465 | 2.572 |
| 5120000 | 569.086 | 1.994 |

|  |  |  |
| --- | --- | --- |
| Reverse Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.077 | -- |
| 20000 | 0.236 | 3.082 |
| 40000 | 0.852 | 3.612 |
| 80000 | 2.262 | 2.654 |
| 160000 | 8.150 | 3.602 |
| 320000 | 17.885 | 2.195 |
| 640000 | 55.445 | 3.100 |
| 1280000 | 111.067 | 2.003 |

|  |  |  |
| --- | --- | --- |
| Average Cases | | |
| N | Time | R |
| 10000 | 0.027 | -- |
| 20000 | 0.026 | 0.954 |
| 40000 | 0.010 | 0.389 |
| 80000 | 0.014 | 1.377 |
| 160000 | 0.030 | 2.149 |
| 320000 | 0.071 | 2.400 |
| 640000 | 0.154 | 2.157 |
| 1280000 | 0.369 | 2.398 |
| 2560000 | 0.844 | 2.290 |
| 5120000 | 1.892 | 2.241 |
| 10240000 | 4.227 | 2.234 |

R

Time

**Sort4**

|  |  |  |
| --- | --- | --- |
| Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.034 | -- |
| 20000 | 0.012 | 0.349 |
| 40000 | 0.014 | 1.157 |
| 80000 | 0.026 | 1.879 |
| 160000 | 0.041 | 1.571 |
| 320000 | 0.106 | 2.621 |
| 640000 | 0.237 | 2.224 |
| 1280000 | 0.591 | 2.497 |
| 2560000 | 1.292 | 2.184 |
| 5120000 | 2.914 | 2.256 |
| 10240000 | 6.246 | 2.144 |

|  |  |  |
| --- | --- | --- |
| Reverse Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.002 | -- |
| 20000 | 0.004 | 2.900 |
| 40000 | 0.008 | 1.855 |
| 80000 | 0.019 | 2.284 |
| 160000 | 0.044 | 2.325 |
| 320000 | 0.096 | 2.165 |
| 640000 | 0.219 | 2.296 |
| 1280000 | 0.587 | 2.678 |
| 2560000 | 1.279 | 2.179 |
| 5120000 | 2.878 | 2.250 |
| 10240000 | 6.304 | 2.190 |

|  |  |  |
| --- | --- | --- |
| Average Cases | | |
| N | Time | R |
| 10000 | 0.002 | -- |
| 20000 | 0.004 | 2.714 |
| 40000 | 0.010 | 2.293 |
| 80000 | 0.019 | 1.877 |
| 160000 | 0.043 | 2.305 |
| 320000 | 0.112 | 2.588 |
| 640000 | 0.292 | 2.608 |
| 1280000 | 0.706 | 2.419 |
| 2560000 | 1.692 | 2.397 |
| 5120000 | 3.723 | 2.200 |
| 10240000 | 8.233 | 2.211 |

R

Time

**Sort5**

|  |  |  |
| --- | --- | --- |
| Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.037 | -- |
| 20000 | 0.037 | 1.011 |
| 40000 | 0.015 | 0.396 |
| 80000 | 0.027 | 1.852 |
| 160000 | 0.020 | 0.717 |
| 320000 | 0.043 | 2.189 |
| 640000 | 0.090 | 2.096 |
| 1280000 | 0.192 | 2.143 |
| 2560000 | 0.393 | 2.050 |
| 5120000 | 0.812 | 2.066 |
| 10240000 | 1.729 | 2.128 |

|  |  |  |
| --- | --- | --- |
| Reverse Sorted Arrays | | |
| N | Time | R |
| 10000 | 0.001 | -- |
| 20000 | 0.002 | 2.240 |
| 40000 | 0.004 | 2.035 |
| 80000 | 0.009 | 2.035 |
| 160000 | 0.019 | 2.228 |
| 320000 | 0.041 | 2.144 |
| 640000 | 0.543 | 13.232 |
| 1280000 | 0.183 | 0.336 |
| 2560000 | 0.758 | 4.152 |
| 5120000 | 0.847 | 1.118 |
| 10240000 | 6.652 | 7.849 |

|  |  |  |
| --- | --- | --- |
| Average Cases | | |
| N | Time | R |
| 10000 | 0.044 | -- |
| 20000 | 0.009 | 0.210 |
| 40000 | 0.008 | 0.889 |
| 80000 | 0.017 | 2.126 |
| 160000 | 0.037 | 2.142 |
| 320000 | 0.089 | 2.401 |
| 640000 | 0.198 | 2.228 |
| 1280000 | 0.469 | 2.367 |
| 2560000 | 1.087 | 2.317 |
| 5120000 | 2.500 | 2.299 |
| 10240000 | 5.705 | 2.282 |

R

Time

This data was graphed as a line graph for both time versus the number of elements in the array, and R versus the number of elements in the array. With Time being the time that each operation took and R being the ratio of the current time with the previous time.

1. Interpretation

On **sort1** the sorted graph is converging to 2, which means that the time complexity for it is O(N). The average case is converging to time complexity O(N2). The only supplied algorithm with O(N) complexity for best case and O(N2) for the average case is **insertion sort** so **sort1** must be an implementation of **insertion sort**. For a **non-random quicksort**, the worst case would be if the minimum or the maximum was selected as the pivot. Because the **quicksort** in this instance always selects the left-most element in the array, then the worst case would be if it had the greatest time on the ordered and reverse-ordered graph. This is the case for **sort3** where the average cases have much better times than the ordered and reverse-ordered cases. This means that **sort3** is the **non-random quicksort**. In the **sort2** section, graphs 10-12 appear to be converging to 4 which means that the time complexity for the average and best case are O(N2). Graphs 7-9 also appear to have similar run times. The only algorithm with these conditions is **selection sort** so **sort3** must be implementing **selection sort**. Then, to determine which remaining methods were stable, I used the Test class made earlier and ran it through the debugger. I tested **sort4** and found out that the array of Tests that I sent it was not keeping the total order. This means that **sort4** is the **randomized-quicksort** and the remaining **sort5** must be the **merge sort.**