Quicksort Algorithm Lab

Homework #5

By:

Pratyusha Thundena

CS 303L – L1A

October 9, 2018

### Problem Specification

The objectives included implementing basic quick sort algorithm and implementing the other quick sort while the utilizing the median of 3 partition on various input files. Then the two quicksort methods’ performance was compared to insertion sort, merge sort, and heap sort.

### Program Design

This program required the following: text file with 100 random numbers (input\_100.txt), text file with 1,000 random numbers (input\_1000.txt), text file with 5,000 random numbers (input\_5000.txt), text file with 10,000 random numbers(input\_10000.txt), text file with 50,000 random numbers(input\_50000.txt), text file with 100,000 random numbers (input\_100000.txt), text file with 500,000 random numbers (input\_500000.txt), Input\_Random.txt, Input\_ReversedSorted.txt, Input\_Sorted.txt, java.util package and IOException class.

The following steps were required to develop this program:

1. scan the following text files: input\_100.txt, input\_1000.txt, input\_5000.txt, input\_10000.txt, input\_50000.txt, input\_100000.txt, input\_500000.txt, Input\_Random.txt, Input\_ReversedSorted.txt, Input\_Sorted.txt
2. save the text files in a known variable
3. read the text files
4. input the values from the text files into the array
5. sort each array by calling quicksort method
6. sort each array by calling quicksort method using median of 3 partitioning
7. record the time taken to sort the elements of the text file
8. display the recorded time (in nanoseconds) of the sorted text file to the user
9. compare the execution time of quicksort with execution time of insertion sort, merge sort, and heap sort

The following constructors and methods were defined within the class:

1. QUICKSORT(List<Integer> A, int p, int r)

Method for calling partitioning in basic quicksort.

b) PARTITION(List<Integer>A, int p, int r)

Method that contains the partitioning algorithm for quicksort.

c) Quicksort (List<Integer> A, int p, int r)

Method for quicksort algorithm using the median of 3 partitioning.

d) Median3(List<Integer> A, int p, int i, int r)

Method that returns the median value from the 3-way partitioning quicksort.

e) parseFile (String filename)

Method that runs the parser over a set of files.

i) main ()

Driver method that displays information to the user.

c) ArrayList<String ()

Constructor that allows for reading and adding files to the program.

The println method of the System.out object was used to display the inputs and results for the driver program.

### Testing Plan

For this lab, the testing plan involved the following files: input\_100.txt, input\_1000.txt, input\_5000.txt, input\_10000.txt, input\_50000.txt, input\_100000.txt, input\_500000.txt., Input\_Random.txt, Input\_ReversedSorted.txt, and Input\_Sorted.txt. The first test included comparing the execution time of the basic quicksort algorithm with the 3-way partitioning quicksort algorithm for seven input files (100.txt -5,000, 000.txt). The second test required comparing the two quicksort algorithms’ execution times to the execution times of insertion sort, merge sort, and heap sort.

### Results

**Fig 1: Basic Quicksort of Various Array Sizes**

|  |  |
| --- | --- |
| Text File | Recorded Time to Sort File (in nanoseconds): |
| input\_100.txt | **0.0521** |
| input\_1000.txt | **1.0682E-4** |
| input\_5000.txt | **7.1278E-5** |
| input\_10000.txt | **5.8573E-5** |
| input\_50000.txt | **6.2659E-5** |
| input\_100000.txt | **5.8376E-5** |
| input\_500000.txt | **7.679E-5** |
| Input\_Random.txt | **7.7172E-5** |
| Input\_ReversedSorted.txt | **6.8732E-5** |
| Input\_Sorted.txt | **1.16011E-4** |

**Fig 2: 3-Way Partitioning QuickSort of Various Array Sizes**

|  |  |
| --- | --- |
| Text File | Recorded Time to Sort File (in nanoseconds): |
| input\_100.txt | **1.073E-4** |
| input\_1000.txt | **8.128E-5** |
| input\_5000.txt | **6.988E-5** |
| input\_10000.txt | **1.039E-4** |
| input\_50000.txt | **8.583E-5** |
| input\_100000.txt | **6.236E-5** |
| input\_500000.txt | **6.510E-5** |
| Input\_Random.txt | **0.0065** |
| Input\_ReversedSorted.txt | **2.179E-4** |
| Input\_Sorted.txt | **1.437E-4** |

**Fig 3: Heap Sort of Various Array Sizes**

|  |  |
| --- | --- |
| Text File | Recorded Time to Sort File (in nano seconds ): |
| input\_100.txt | 0.0681 |
| input\_1000.txt | 0.0740 |
| input\_5000.txt | 0.3203 |
| input\_10000.txt | 0.6887 |
| input\_50000.txt | 3.8389 |
| input\_100000.txt | 7.8123 |
| input\_500000.txt | 38.383 |

**Fig 4: Merge Sort of Various Array Sizes**

|  |  |
| --- | --- |
| Text File | Recorded Time to Sort File (in nanoseconds): |
| input\_100.txt | 0.0 |
| input\_1000.txt | 15.0 |
| input\_5000.txt | 48.0 |
| input\_10000.txt | 40.0 |
| input\_50000.txt | 323.0 |
| input\_100000.txt | 1722.0 |
| input\_500000.txt | 54380 |

**Fig 5: Insertion Sort of Various Array Sizes**

|  |  |
| --- | --- |
| Text File | Recorded Time to Sort File (in nanoseconds): |
| input\_100.txt | 115337.0 |
| input\_1000.txt | 9793153.0 |
| input\_5000.txt | 5.588623 E7 |
| input\_10000.txt | 2.679420 E7 |
| input\_50000.txt | 4.713254 E8 |
| input\_100000.txt | 1.943406 E9 |
| input\_500000.txt | 6.409374 E10 |

### 5. Analysis and Conclusions

In basic quicksort, the larger input files (5000.txt-500000.txt) took a shorter amount of time to sort than when the input size was small (100.txt). The three-way partitioning quicksort was even faster than the basic quicksort; specifically, because the method completed a lot more extra rearrangements and required O(nlog(n)) time. Additionally, quicksort performed a lot faster than heap sort (i.e. time complexity of O(nlog(n)), merge sort (i.e. time complexity of O(nlog(n)), and insertion sort (i.e. average time complexity of On2). One of the primary reasons that quicksort was faster than all other sorting techniques was because its inner loop could be efficiently implemented on most architectures.

### 6. References

The pseudocodes for quicksort was provided in the homework assignment and textbook, while the sample report provided by Dr. Purushotham Bangalore was used to do the lab report.

### 7. Homework

Determine the time complexity *T(n)* for the given pseudo code:

i = n

while (i >= 1) {

j = i

while (j <= n) {

// some work here

j = j \* 2

} i = i / 2

}

T(n) = ((1 + (1+1) + (1+1+1) + (1+1+1+1) + …………… +1)) (k +1)

= 1 + 2 + 3 + 4 + …... + (k+1)

= ((k+1) (k+1+1)) / 2

= ((k+1) (k+2))/2

= O(k2)

= O(n2)