Comparing RunTimes between QuickSort, Insertion Sort, MergeSort, and HeapSort

Homework #5

By

John Kiyak

CS303

October 1, 2018

**Problem Specification**

The goal of this assignment was to write a program that would implement the quicksort algorithm. Input files were downloaded from canvas ranging from 16 to 500000, and were each read into the main function and passed into the quicksort array. The methods were then used to sort the arrays of differing sizes, and outputting the execution time for each array specified.

**Program Design**

For the regular quicksort algorithm, the main method reads in the text file and passes it into an array. It then starts the timer, calls the quicksort method, and stops the timer when the quicksort method is completed, and outputs the time the quicksort method takes. The quicksort class takes in the arguments of the arr, the low number ( which is 0), and the high which is the length of the array - 1. If the high variable was higher than the low variable, it set the middle variable to a call to the partition method with the same arguments of the quicksort method. It then calls quicksort two more times, with the arr, low, and middle variable -1, and arr, middle plus 1 and high variables. The partition method creates a pivot variable and compares it to the lowest in the array, and swaps if the lowest is lower than the highest.

For the median of 3 quicksort method, the main method was the same. For the quicksort, a variable N was set to the high-low + 1 on the array. A variable m was used to call the median3 class, using the arguments of the arr itself, the low value, the low value + (N /2), and high). Then, the m’th number in the variable and the high number in the variable were swapped, and then the rest of the quicksort proceeded as normal from the last one. For the median of 3 method, the arguments of the arr, i,j, and k were initialized. The arrays in the i,j, and k placements were compared to each other, and whichever one was the largest was returned.

**Testing Plan**

The testing plan for this lab was pretty straight forward. For the first portion, I tested the normal quicksort class on arrays that were already sorted, ones that were reverse sorted, and a random sorted array. Then, I tested the quicksort and median classes on array says going from 16 to 500000, and wrote down the execution times for each of them. Finally, so that all test cases were uniform, I went back and tested all of the previous sorts (insertion, merge, and heapsort), and compare the execution times for all of those as well.

**Test Cases**

The test cases are shown in the table below:

For the Sorted, Reverse Sorted, and Random Arrays:

|  |  |
| --- | --- |
| Arrays | Time Taken (s) |
| Sorted | .14058 |
| Reverse Sorted | .00780 |
| Random | .005 |

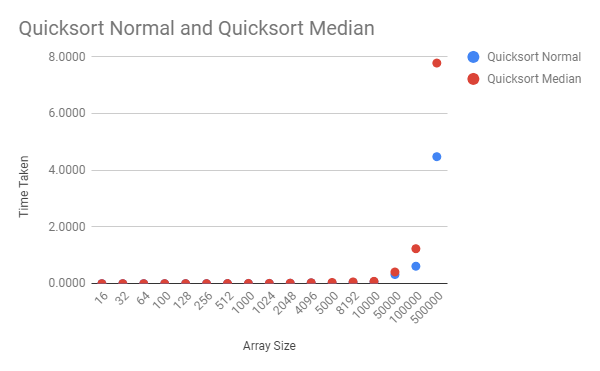
For the Normal Quicksort vs the Median Quicksort:

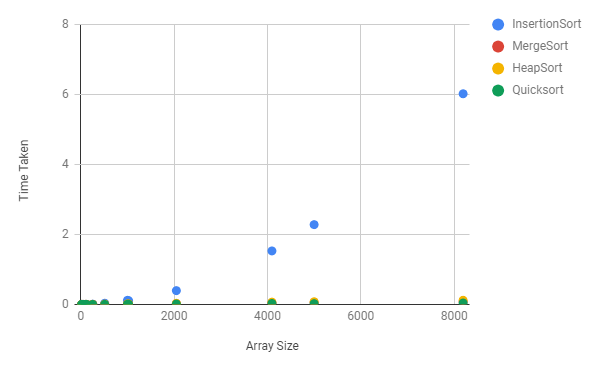
|  |  |  |
| --- | --- | --- |
| Array Size | Quicksort Normal | Quicksort Median |
| 16 | 0.0000 | 0.0000 |
| 32 | 0.0000 | 0.0010 |
| 64 | 0.0000 | 0.0000 |
| 100 | 0.0000 | 0.0010 |
| 128 | 0.0010 | 0.0009 |
| 256 | 0.0009 | 0.0009 |
| 512 | 0.0020 | 0.0029 |
| 1000 | 0.0060 | 0.0069 |
| 1024 | 0.0060 | 0.0079 |
| 2048 | 0.0120 | 0.0150 |
| 4096 | 0.0290 | 0.0309 |
| 5000 | 0.0240 | 0.0409 |
| 8192 | 0.0410 | 0.0599 |
| 10000 | 0.0620 | 0.0790 |
| 50000 | 0.3190 | 0.4100 |
| 100000 | 0.6122 | 1.2330 |
| 500000 | 4.4800 | 7.7833 |

And finally, for the comparison of InsertionSort, MergeSort, HeapSort, and QuickSort:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | InsertionSort | MergeSort | HeapSort | Quicksort |
| 16 | 0.0000 | 0.0000 | 0 | 0.0000 |
| 32 | 0.0000 | 0.0009 | 0 | 0.0000 |
| 64 | 0.0009 | 0.0009 | 0.0009 | 0.0000 |
| 100 | 0.0010 | 0.0000 | 0.0009 | 0.0000 |
| 128 | 0.0019 | 0.0009 | 0.0019 | 0.0010 |
| 256 | 0.0080 | 0.0029 | 0.003 | 0.0009 |
| 512 | 0.0340 | 0.0049 | 0.0069 | 0.0020 |
| 1000 | 0.1179 | 0.0109 | 0.0149 | 0.0060 |
| 1024 | 0.1099 | 0.0119 | 0.0149 | 0.0060 |
| 2048 | 0.3949 | 0.0250 | 0.0329 | 0.0120 |
| 4096 | 1.5299 | 0.0460 | 0.0659 | 0.0290 |
| 5000 | 2.2810 | 0.0570 | 0.0769 | 0.0240 |
| 8192 | 6.0219 | 0.0800 | 0.1189 | 0.0410 |

And here are graphical representations of the above data:





**Analysis and Conclusions**

Considering the worst case time complexity of quicksort (n^2), and the average and best both being (n log n), the worst case time complexity appears to only appear when the array list in question is sorted already. In regards to sorted, unsorted, and randomly sorted array, the array performs best when the array is randomly sorted, and slower than the randomly sorted, and faster than the sorted array when unsorted. Compared to all of the previous algorithims, it has the best execution time thus far, therefore being the best sorting algorithm to use in a non-sorted array list.

**References**

I used geeks for geeks to get started on my quicksort and for help on time complexities. I also used the textbook for the big O times for the algorithms.

**Time Complexity:**

**Pseudocode:**

I = n

While ( i >=1) {

J = i

While (j < n) {

// some work here

J = j \* 2

}

I = i/2

}

For this pseudocode, every while loop has n runtime. For everyone nested while loop, you add another n to the product. So for this, it has one while loop nested inside another one, so the time complexity for this pseudocode would be log(n^2), since the loop is divided/multiplied by a constant amount.