Laboratory # 2

Sorting Algorithm Time Complexity analysis

Ivan Gastelum

Professor Olac Fuentes

TA: Aninditha

CS 2302 Data Structures

University of Texas at El Paso

**INTRODUCTION:**

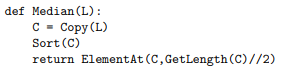
The second laboratory project of Data structures is dedicated to study the concept of Time complexities (running times) of different algorithms. As it can be seen, there many, if not infinite, ways an algorithm can be put into code, and that’s the good thing about coding. One programmer can perform an algorithm in different way as another programmer, but the question is, which code is more efficient? This brings us to another question: how do we define efficiency in a code or algorithm? There are two ways to compare two algorithms/codes: by defining its running time, and the space it takes from memory. This laboratory as mentioned, will be to analyze running times, meaning that the algorithm that takes less time to perform the task will result to be more efficient than the other. This laboratory will compare three famous sorting algorithms: Merge Sort, Quick Sort and Bubble Sort. Each algorithm was put into code and perform various experiments at different inputs and recorded their running times and see which is the most efficient.

In theory, when sorting an array or list of random values, bubble sort seems to be not a good idea as it has the highest Big Oh notation out of the three sorting algorithms. The variable n will be defined as the number of values that the user inputs into the code, so its running time will depend on *n.*

**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION:**

**Defining the task:**

The median of a list L is the element such that half of the elements in L are smaller than a and half of them are larger than a. For example, the median of list L = {20, 10, 45, 1, 12} is 12, since 12 is greater than 1 and 10 and smaller than 20 and 45. An easy way to find the median is to sort the list and return the element in the middle:



1. Sort list using bubble sort, then return the element in the middle.

2. Sort list using merge sort, then return the element in the middle.

3. Sort list using quicksort, then return the element in the middle.

**Solutions:**

Finding the median is easy once the list is already sorted, so the hard part is to sort the list in ascending order by using these three algorithms into practice. Below will show the procedures I took in order to get every algorithm to work in code.

Bubble Sort:

Bubble sort usually is performed using nested loops and not recursion. A brief description of how bubble sort works, you take the front value of the list, in this case the head, and start comparing its item to each one of the consecutive values. Comparing first value with second, third, fourth, and so on as you are traversing the list until you find a value greater, and that’s when you swap places, reordering the values until you get to the end of the list. So the solution is straight forward, a nested while loop when the first level while loop will be to define the comparing value, while the second one is used to compare and swap each value when needed. Code shown in Appendix.

Merge Sort:

Bubble sort is a recursive algorithm that’s more efficient algorithm than bubble sort, but a more complicated code. The description of how merge sort works: the list is broken down into halves with a recursive call until we have several lists of one or zero elements inside, once we reach to this point, we merge back list while comparing one list with the other. We keep doing that with a recursive call until we have one sorted list.

I broke down the sorting algorithm into their main functions: Splitting Lists, and Merging Lists, and I created one main method to incorporate the two methods and to return the final sorted list. The method ‘MergeSort’ as the main, ‘Split\_Lists’ as splitting lists into halves, and ‘Merge\_Lists’ as merging back the lists. And by using the functions given by the professor such as IsEmpty, Append, Remove, Getlength I was able to manage to perform each function. See Appendix for code.

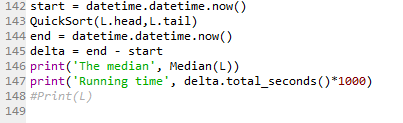
Quick Sort:

Quick sort is also a recursive algorithm that’s as efficient as merge sort. Additionally, coding quick sorting was found to be performed more easily, by inputting less lines of code than merge sort. For Quick Sort, one value should be chosen as the pivot so its value can be used to compare two pointers that are traversing the list, one starting from the right, the other staring from the left. While these pointers are traversing the list, when the left pointer fins a value that is smaller than the pivot and when the right pointer finds a values that is greater than the pivot, these two values swap. The procedure continues until both pointers cross, and then the pivot goes back into its original place, now recursively, the list is broken down into two lists, and perform the same traversal procedure.

Solution is same a merge sort. Breaking down each instructional step into methods, and perform each functionality recursively. See Appendix for code.

**EXPERIMENTAL RESULTS:**

I used datetime as one of my imports to record the start and end times in order to measure running times for each code sorting segment. See piece of code below on how to implement it. The output of time comes out in terms of milliseconds.



|  |  |  |  |
| --- | --- | --- | --- |
| nth values | Bubble Sort (miliseconds) | Merge Sort (miliseconds) | Quick Sort (miliseconds) |
| 10 | 4.012 | 0.001 | 0.001 |
| 50 | 15.622 | 0.001 | 0.001 |
| 100 | 11.996 | 4.998 | 15.624 |
| 200 | 31.248 | 15.624 | 15.624 |
| 500 | 109.368 | 46.873 | 46.873 |
| 1000 | 1246.078 | 186.885 | 171.87 |
| 5000 | 12795.795 | 4584.927 | maximum recursion depth exceeded in comparison |
| 8000 | 31183.1 | 12051.421 | maximum recursion depth exceeded in comparison |
| 10000 | 57502.4889 | 17313.36 | maximum recursion depth exceeded in comparison |

If we compare these results as shown above, it can be analyzed that Quick Sort has come to be most efficient method out of the three. But there is only one exception, after inputting n to be greater than 5000, it reaches a maximum recursion depth and returns an error. So, for bigger input, merge sort can be the best choice.

**CONCLUSIONS:**

To sum up, we can conclude that for smaller values of n, quick sort can be the best choice to sort values of a linked list of objects. However, it shows to be limited, for bigger values, merge sort

We can also conclude that in addition to their efficiencies, theoretically each sorting code for each algorithms has proven to be in their respective big Oh notation. For bubble sort (blue line) it clearly shows how after n = 500, its running times increase close to a parabola, making it seem to be O(n^2). While Merge and Quick sort keep to be competitive with each other and having a trend similar, or less to O(n log n).

**APPENDIX:**

