Quicksort

Aditya Gupta

October 2022

1 Introduction

In the previous assignments, we looked at various sorting algorithms with different amounts of efficiency. We have looked at insertion sort, selection sort and merge sort so far. In this assignment, we look at the quick sort algorithm which is similar to the merge sort algorithm in the way that it also makes use of a divide-and-conquer strategy and uses recursion to sort the data structures. We will implement this algorithm for an array and a linked list in this assignment and compare the time complexity to do the same task in both cases.

2 Quicksort on Arrays

The quicksort algorithm makes use of a pivot variable that is set to the last element in the array. Before we do anything we check if the lower index and the higher index passed to the function are such that the lower index value is less than the higher index value. Otherwise, we return. After this, we create 2 local variables that are the left and the right pointer. We keep iterating through the array which we want to sort while the left pointer's value is lower than the value of the right pointer.

After this, we keep incrementing the value of the left pointer until the value in the array at that index(taking the left pointer as the index) is larger than the pivot value. We do something similar for the right pointer where until we reach a value for the right pointer such that the value in the array at that point is smaller than the value of the pivot variable, we decrement the value of the right pointer.

After this, we swap the values at the left and right pointers. The reason for doing all this is so that we alter the original array so that all values at the left of the pivot are smaller in value compared to it while all values to its right in the array are larger than it. After we exit the outer while loop we do another swap of the values at the left and right pointers. After this, we recursively do the same process as above on the subarrays to the left and right of the pivot value. This way we end up sorting the entire array by dividing the sorting problems into smaller arrays to the left and right of the pivot that are further broken down when this algorithm is applied to them.

```
public static void quickSort(int [] array, int lowIndex, int highIndex){
        if(lowIndex >= highIndex){
            return;
        }
        int pivot = array[highIndex];

        int leftPointer = lowIndex;
        int rightPointer = highIndex;
        while (leftPointer < rightPointer){
            while(array[leftPointer] <= pivot && leftPointer < rightPointer){
                leftPointer++;
            }
            while(array[rightPointer] >= pivot && leftPointer < rightPointer){
                rightPointer--;
            }
            swap(array,leftPointer,rightPointer);
        }
        swap(array,leftPointer,highIndex);

        quickSort(array,lowIndex,leftPointer-1);
        quickSort(array,leftPointer+1,highIndex);
    }
```

There is a way to optimize this algorithm which can be done by choosing a random pivot instead of always taking the last element in the array for that variable. In the average case, this should work slightly better than our current algorithm.

3 Quicksort on Linked Lists

Quicksort on Linked Lists is not very different from how we implemented the algorithm before. In this case, rather than swapping values, we change references to the values in such a way that all the nodes are linked in a sorted fashion. This time we add another function called the partition function which is used to divide the problem of sorting this linked list into smaller subsets. We rearrange node values, and return the node previous to end node passed as a parameter. This returned node will then be used to partition the linked list.

```
public void quickSortLL(Node start, Node end){
    if (start != null && start != end && start != end.next){
        Node node_before_pivot = partitionLL(start, end);
        Node pivot = node_before_pivot.next;
        quickSortLL(start, node_before_pivot);
        quickSortLL(pivot.next, end);
    }
}
public Node partitionLL(Node start, Node end){
    Node current  = start;
    Node previous = start;
    Node iterator = start;
    int pivot = end.data;
    while (iterator != end){
        if(iterator.data < pivot){
            swapNode(current,iterator);
            previous = current;
            current = current.next;
        }
        iterator = iterator.next;
    }
    swapNode(current,end);
    return previous;
}
```

Essentially we go about sorting everything to the right of the pivot followed by sorting everything to the left of the pivot. We have chosen the end node as our choice of pivot. Based on its value over the course of the execution of the algorithm this node will be relocated in such a way that all nodes that precede it will have a smaller value than it and every node after it will have a larger value compared to this node. We do this recursively over the entire list to sort it.

4 Benchmarks

Here we present the data gathered for the time taken to do quicksort on different sized arrays and linked lists and then comparing the two cases to check which one performs better than the other.

4.1 Quick Sort for Arrays

The data we get upon benchmarking the quicksort algorithm for different-sized arrays have been tabulated below:

Case	n	Time(in ms)
1	250	23.7
2	500	48.0
3	1000	98.2
4	2000	210.0
5	4000	464.9
6	8000	910.3
7	16000	1926.2
8	32000	4065.5
9	64000	8180.8
10	128000	19971.4
11	256000	61129.0
12	512000	136913.4
13	1024000	269023.4

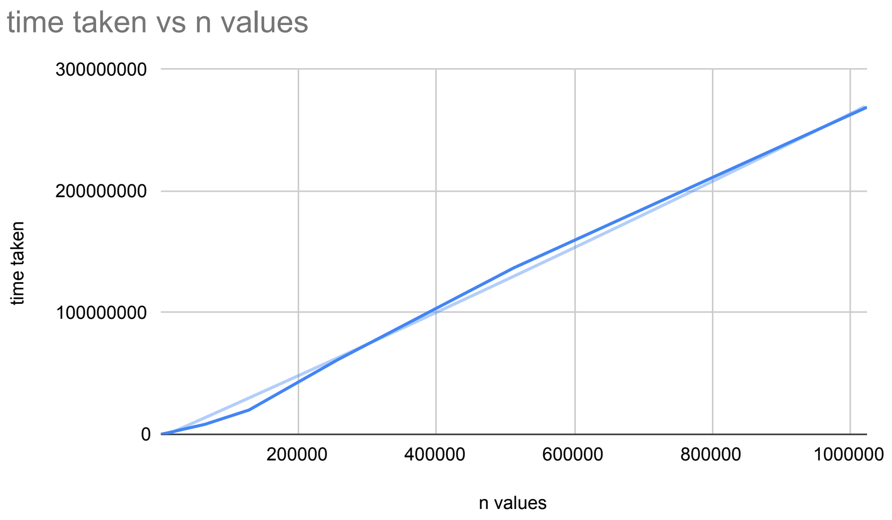

Figure 1: Graph illustrating the Quick Sort Algorithm for Arrays

Here the time complexity of this algorithm is given by O(nlog(n)). From the graph it appears to be linear but the we will not be able to observe the effect of the log(n) factor as long as we don't have an extremely large value of n. The above benchmark was obtained after doing this process for 1000 iterations on a different randomly generated array each time.

4.2 Quick Sort for Linked Lists

Case	n	Time(in ms)
1	250	12.0
2	500	27.4
3	1000	62.4
4	2000	147.7
5	4000	324.0
6	8000	1239.7
7	16000	2593.4
8	32000	5017.2

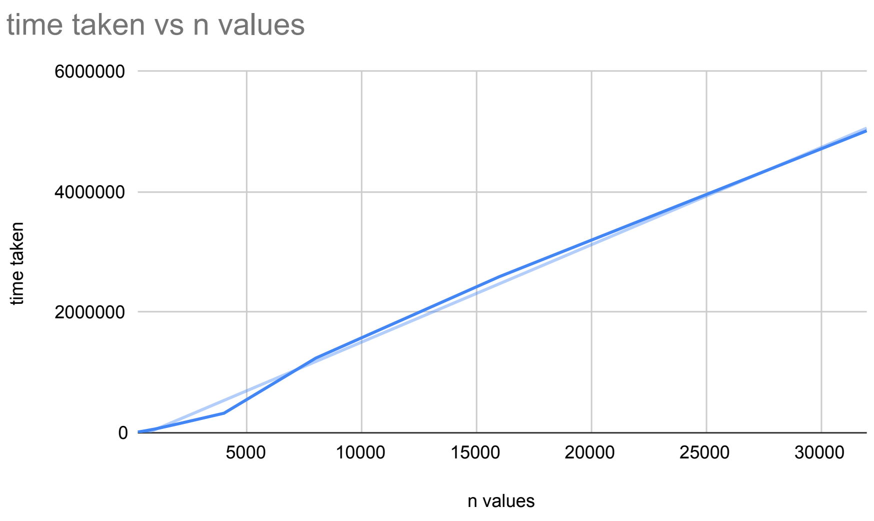

Figure 2: Graph illustrating the Quick Sort Algorithm for Linked Lists

Here the time complexity for the algorithm remains the same as the time taken to do it in case of the arrays i.e. O(nlog(n)). Here the benchmarks were obtained by executing this algorithm over 1000 iterations for randomly generated lists.

4.3 Quick Sort Comparision between Arrays and Linked Lists

Case	n	Time QSL(in ms)	Time QSA(in ms)
1	250	12.0	23.7
2	500	27.4	48.0
3	1000	62.4	98.2
4	2000	147.7	210.0
5	4000	324.0	464.8
6	8000	1239.7	910.3
7	16000	2593.4	1926.2
8	32000	5017.2	4065.5

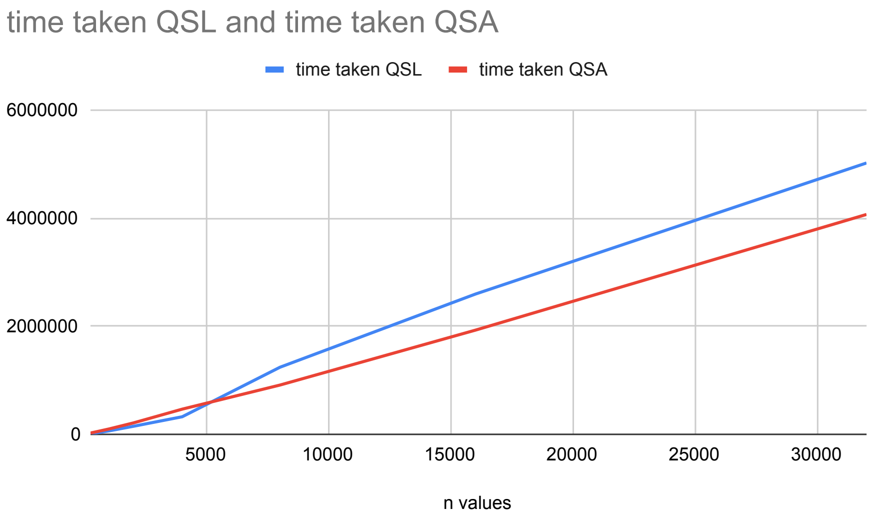

Figure 3: Graph illustrating the Quick Sort Algorithm for Linked Lists as well as Arrays

Finally we compare the performance of the quicksort algorithm for arrays as well as linked lists. We observe that the time taken to perform the same algorithm for the linked lists takes more time than it took to perform the task for arrays. At the beginning its a bit faster than doing the algorithm on arrays (when performed for smaller data sets i.e. <= 4000 elements). This is most probably as we have direct access to any index inside of an array while in case of a linked list we have to traverse the list to get a specific node. Thus we can conclude that this algorithm performs better when implemented for arrays rather than linked lists.