Sorting an Array

Jeffrey Chang

13th September 2022

Introduction

For the course ID1021 - Algorithms and Data Structures, we are given tasks to submit each week for us to learn about the course content. This report concerns the fourth assignment which was about different ways of sorting arrays and their respective efficiencies, benefits and pros and cons, so we know when to use what sort. This is a basic introduction into search algorithms and in this essay, I will explain the runtime as a function of the size of the arrays.

Background

This assignment requires us to understand the fundamentals of searching algorithms and how one can sort data in an array to use for greatest benefit during a task.

There are different ways to sort, such as *Merge, Heap, Bubble, Insertion or Selection Sort* - being the most common ones. It is therefore important to know which method of sorting to choose in order to achieve the best results. This is done in order to limit or keep within time or space complexities required by the assignment or task at hand.

The Tasks

The Not So Efficient

We start this assignment with the simplest of sorts - *Selection Sort*. It is definitely not the most efficient, but it is very simple to implement.

The idea is also very simple, search through the whole array and find the smallest element, then swap with the current index, then continue until the entire array is sorted and searched through. As can be deduced, it is incredibly slow but it still works for smaller array sizes. Implementing the code following the assignment instructions gave me the following code snippet:

```
for (int i = 0; i < array.length - 1; i++) //search whole array
    int cand = i //first candidate = index
    for(int j = i; j < array.length; j++) //if new candidate is smaller - swap
        if(array[cand] > array[j]) //if, then swap
            cand = j;
```

We have actually already explored time complexities of different sorts - very briefly, in the previous course ID1018 - Programming I so I can say with certainty that Selection sort will have a runtime complexity of $O(n^2)$. This is also confirmed through the following benchmark result presented in the table:

Array Size	**nanoTime**
10	298
50	1034
100	3042
500	39266
1000	112053
5000	3767520
10000	13871217

Table 1: Array size vs Selection Sort time

The table clearly indicates an exponential increase in time as the array size increases.

The second form of sorting we implemented was another fairly simple and straightforward sorting algorithm, mainly the *Insertion Sort*. This is similar to Selection sort, except for when we find a "smaller" element, we sort it in an already sorted section of the array - I like to think of it as sorting within the array until the array is fully sorted. The implementation of this, again following instructions gives us the following code:

```
for (int i = 1; i < array.length ; i++) //scan through all
//going from i to 0, if smaller - move in the element
    for (int j = i; j > 0 && array[i] < array[j - 1] ; j--) //if smaller, insert
        array[j] = array[j - 1] //inserting and making space for element to move in
        i = j - 1 //make all in previous position go one step and continue
```

As we can see, it follows very closely to the time of Selection sort, meaning that its complexity lies in $O(n^2)$ as well. They both have double for-loops going through array of size n, causing an n*n time complexity. The main difference is, however that for every array size, small or large, Insertion sort is either much quicker (as shown by an array size of 10,000 elements) or just slightly quicker. There is no instance where Selection sort is faster at sorting any sized array.

Array Size	nanoTime
10	154
50	982
100	3210
500	21316
1000	61919
5000	1632639
10000	6006295

Table 2: Array size vs Insertion Sort time

However, as suggested in the assignment, Insertion sort also requires more space complexity but its not considered in this assignment. I do however know that Insertion sort has a best runtime complexity of O(n), if the array is sorted, which could be considered.

From previous courses, I know that because both Selection and Insertion sort have two for-loops in their "main" code, their worst case runtime complexities are $O(n^2)$ - although Selection sort is always $O(n^2)$. I also know that they both start with an unsorted array and take one element at a time to a sorted section of the array with comparisons done n(n-1)/2 times.

The main difference between the two basic sorts lies in what the inner for-loop does (since the upper just searches the whole array). Whilst in Selection sort, the inner for-loop only loops over the *unsorted* elements to find the next smallest element, in Insertion sort, the inner loop iterates over the *sorted* elements and they are moved around until the loop insertions are over and the array is sorted.

Because of the slow speed of Selection sort it may be more efficient at checking an already sorted array or when there is a memory complexity restriction whilst Insertion sort may be better at actually sorting an unsorted array efficiently, given that there are no memory complexity restriction.

Completely Different

For the second part of this assignment, we are implementing another type of sorting algorithm, this time, one that is much more efficient, has better time complexity and is more robust - namely, the *Merge Sort*. Merge sort can be separated into the sorting part and the merging part, and following the instructions of the assignment, we begin the implementation of this sort by creating the following merge method.

```
//following the exact instructions for the merging, we get, for each part:
if(i > mid) //if i is greater than mid
    org[k] = org[j] //move the j item to the org array
```

```
    j++ //update j

else if (j > hi) //else if j is greater than hi
    org[k] = org[i] //move the i item to the org array, (update i)

else if (org[i] < org[j]) //else if i is smaller than j item
    org[k] = org[i] //move it to the org array, (update i)

else{ // else you can move the j item to the org array, (update j)
    org[k] = org[j];
```

Now that the Merge method has been successfully implemented, we turn towards the sorting method. By using a recursive merge technique, where we halve the array each time until there are only two elements left, we can create a merged array which results in a sorted array.

```
recursive:
if (lo != hi)
    int mid = lo + (hi - lo)/2
    sort(org, aux, lo, mid) //sort the items from lo to mid
    sort(org, aux, mid + 1, hi) //sort the items from mid + 1 to hi
    merge(org, aux, lo, mid, hi) //merge the two sections using the additional...
//recursive first sorts the "subarrays" and then merges them together
//it does this continuously until we get our original array back - sorted.
```

In the following table we have the benchmark of Merge sort time vs different array sizes.

Array Size	**nanoTime**
10	562
50	2504
100	4678
500	15381
1000	30202
5000	127301
10000	231629

Table 3: Array size vs Merge Sort time

If we compare these results to Selection or Insertion sort, it is quite clear that Merge sort is much more efficient at sorting - for large enough array sizes. The benchmark is also a clear indication of the runtime complexity that Merge sort has, as we increase the size of the array, we see that the runtime difference taken to sort increases steadily, by quite small margins,

and then starts to increase a bit more but hold to a maximum increase by factors of 10, so not O(n^2), but not O(logn) either. This means that we have a time complexity of O(nlogn).

This is also quite intuitive since we divide the array into half so we have two n/2-sized arrays instead of one n-sized array, there are then four operations (the if-else loop, i++, k++/j++ and array[k] = left[i] | right[j]) then we continue with their respective suboperations going to n/4, n/8, ..., n/i^2 which adds up to O((n/2)log(n)) which of course simplifies to O(nlogn).

Very simplified, merge sort iterates itself by halving continuously which already indicates that it will be on a logarithmic scale since you can only half until (logn). With the addition of an n sized array to iterate through, we get an O(nlogn) time complexity.