Sorting an array

Fatima Mohammad Ali

Fall 15/09/2023

Introduction

The purpose of this assignment was to explore *selection*, *insertion*, and *merge* sort. For each of the algorithms, the run time complexity (as a function of the array size) was determined and analysed.

Selection and Insertion sort

Selection sort

Selection sort is an algorithm that constantly finds the minimum value of the unsorted part of the array and swaps with the first unsorted item which will gradually sort the entire sequence.

The implementation of this algorithm started by setting up nested loops. The first element at index `i` of the outer loop was compared to the rest of the unsorted array by iterating an inner loop (iterator `j`) that started at `i`. While iterating the inner loop, the element at `j` was constantly compared to the element at a temporary index '`candidate`' (initially set equal to `i`). If the item at `j` was smaller than the one at `candidate`, the index `candidate` was updated to `j` and this procedure carried on until `candidate` held the minimum value of the unsorted part of the array. After reaching the end of the array, the positions of the current and minimum values were swapped and the current index `i` was incremented to sort the next item. Thus, the array gradually became sorted while moving forward.

Insertion sort

Insertion sort works by considering an element, comparing it to the elements on the left, and inserting it at the right place to obtain a sorted sequence.

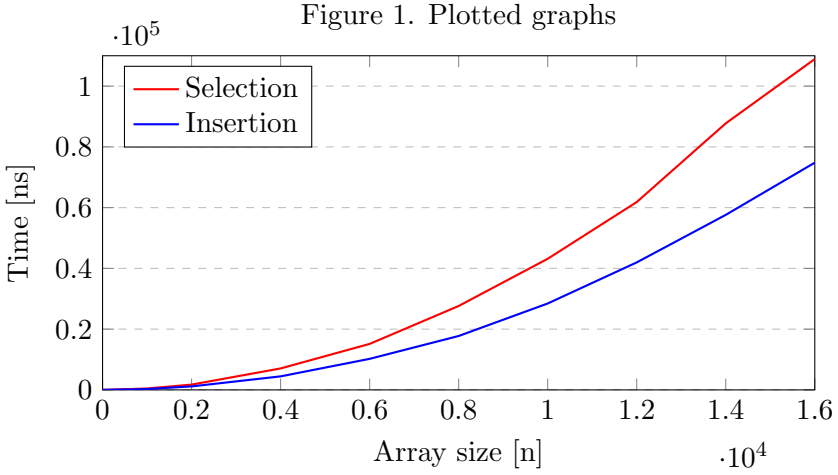
The implemented algorithm used nested loops with the outer loop (iterator `i`) moving forward and the inner loop (iterator `j`) moving from the `i`-th position to the left. In each iteration of the inner loop, a comparison between the considered element and the element to the left was done. If the

adjacent element was larger, it was shifted/swapped to the right. The shifting was done until a smaller element was found, and the considered element was inserted at its correct position. The next unsorted elements followed the same procedure until the entire array became sorted.

Results

The benchmark results of both selection and insertion sort were plotted as graphs in figure 1.

Figure 1. Plotted graphs



Array sizes n were given by 1000, 2000, 4000, 6000, ..., 16000, and both algorithms were given the same randomized arrays.

Discussion

As coordinates, both selection and insertion sort displayed square function qualities where each doubling of size n resulted in the time becoming four times larger. This could also be confirmed by the quadratic appearance of the graphs in figure 1, which only indicates that the run-time complexity of both algorithms is $O(n^2)$.

The number of comparisons done in selection sort is quadratic since each current element (all n-elements) has to be compared to the rest of the array (rest being (n-1),(n-2), etc., which is proportional to n^2). The amount of swaps is linear since at worst case, n-1 elements need to be placed in their correct position to sort the array. Thus, this algorithm could be generally expressed with $t(n) = (n(n-1))/2$, but since n^2 is the dominating term, the time complexity becomes $O(n^2)$.

For insertion sort, the number of comparisons and swaps are quadratic since the inner loop has to make n^2 comparisons and swaps at most. This goes to show that insertion sort has the big-O notation $O(n^2)$.

Further analysis of the graphs also shows that insertion sort is faster than selection sort. This could be explained by several factors, one being the amount of comparisons and swaps needed when given a partially/entirely sorted array. At average/best case, insertion sort would only need n comparisons to determine that the array is already sorted. Selection sort however, would still go through each element and compare it to the remainder of the array, making its best case for comparisons n^2. This shows how insertion sort is more useful when given a partially/entirely sorted array, which is one of the reasons it is better than selection sort.

Merge sort

Merge sort is an algorithm that uses an additional array to store temporary results before presenting the final sorted array. The algorithm constantly divides the given array in halves that are respectively sorted and merged together, creating a fully sorted array at last.

To implement this algorithm, a `sort` method that utilizes recursion was used. The recursion procedure split the array into two halves after each call of itself until the array could not be divided anymore (until lowest index = last index, this stop condition prevented an endless loop). Every half was sorted respectively then merged into one sorted array as follows:

```
private static void sort(int[] org, int[] aux, int lo, int hi) {
    if (lo != hi) { //stop condition for the recursion
        int mid = (lo + hi)/2;
        // recursive call, sorts the items from lo to mid
        sort(org, aux, lo, mid);

        // recursive call, sorts the items from mid+1 to hi
        sort(org, aux, mid+1, hi);

        // merges the two sections using the additional array
        merge(org, aux, lo, mid, hi);
    }
```

The merging started when reaching the base case, i.e. reaching one element. The procedure consisted of going through two arrays, item by item, and selecting the smallest item found to place it next in the array. The following `merge` method (most of it was given in the assignment) was used.

```
private static void merge(int[] org, int[] aux, int lo, int mid, int hi) {
    // copies all items from lo to hi from org to aux
    //initializes int i = lo and int j = mid+1
```

```
:
// for all indices from lo to hi
for (int k = lo; k <= hi; k++) {
    if(i > mid)
        org[k] = aux[j++];
    else if(j > hi)
        org[k] = aux[i++];
    else if(aux[i] < aux[j])
        org[k] = aux[i++];
    else
        org[k] = aux[j++];
    }
}
```

Selection and Insertion sort vs Merge sort

The benchmark results of all sorting algorithms can be seen in table 1.

n	Selection sort	Insertion sort	Merge sort
1000	200	130	33
2000	780	500	79
4000	3000	2000	210
8000	12000	7700	460
16000	47000	32000	1100
32000	190000	130000	2200

Table 1: *T*imestamps are given in microseconds and rounded to two sign. figures.

For merge sort, an array of size n is divided into $log(n)$ parts and the merging of all subarrays is linear ($O(n)$). Thus, the time complexity for this algorithm $O(n * log(n))$

As previously analysed, both selection and insertion sort have run-time complexity of $O(n^2)$, with the latter overtaking its counterpart in regards of performance. Comparing merge sort with these two algorithms only shows how the last algorithm wins by a feat, making it the most efficient of the three.