Sorting an array

Mahdi Nazari

September 2022

Introduction

In this assignment three different sorting algorithms will be presented, namely Selection sort, Insertion sort and Merge sort. These algorithms will be implemented on arrays with integer type of elements. As it will be discussed later every of these algorithms has its advantages and disadvantages and can be useful in different situations.

Selection sort

The first sorting algorithm is Selection sort. This algorithm works as follows: The index of the first element in the array will be assigned as a candidate for the smallest value in the array. Then program will loop in the array and compare every element with the first element and if it is smaller than the first element the candidate will be changed to the index of that element. This procedure will go on until the program find the smallest element in the array then it will swap the smallest element with the first element. In the second iteration the candidate for smallest element will be the second element in the array and program will do the same procedure as in the first iteration and swap the elements if it finds the smallest element. At the end when the program is done, a sorted array will be generated by the program.

The time complexity of this algorithm will be as follows: If an unsorted array of length (n) is entered as input to the program. The program will do (n-1) comparisons in the first iteration and eventually do the swap, in the second iteration the number of comparison will be (n-2) and in the third iteration (n-3) etc. This can be expressed as follows:
(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = (n(n-1))/2 = $n^2/2$ - n/2
This means that the time complexity of this algorithm will be an $O(n^2)$.

Insertion sort

The second algorithm is Insertion sort. This algorithm works as follows: The element will be inserted to the already sorted part of the array if the element is smallest. In the first iteration pointer 1(just to make it clear number 1 and 2 will be used) will point at the first element in the array and pointer 2 will point to the element after. These elements will be compared to each other and if the second element is smaller than the first element they will be swapped with each other. In the second iteration the pointer 1 will point to the second element and pointer 2 will point to the third element. The third element will be compared to the elements before it and will be moved toward beginning of the array if it is smaller than the elements before. This comparison will go on until the pointer 2 reaches the start of the array. In the third iteration the pointer 1 will move one step to the right and pointer 2 will point at the next element to the right of the pointer 1. The same procedure as the first and second iteration will happen. The program will reach its end when the pointer 1 is pointing to penultimate element and by that time the elements in the array are sorted. A part of the program code is as follows:

```
for (int i = 0; i < array.length; i++) {
            for (int j = i; j > 0 && array[j] < array[j-1]; j--) {
                int tempurary = array[j];
                array[j] = array[j-1];
                array[j-1] = tempurary;
            }
        }
```

Time complexity of this algorithm can be explained as follows: In the first iteration the program does 1 comparison and eventually 1 swap. In the second iteration the program does 2 comparisons and eventually 1 swap. In the third iteration 3 comparisons and ev. 1 swap etc. In the last iteration the program does (array.length -1) comparisons and ev. 1 swap. The time complexity for an array with (n) elements can be determined as follows:
$1 + 2 + 3 + ... + (n-1) = (n(n-1))/2 = n^2/2 - n/2$
This means that the time complexity of this algorithm will be an $O(n^2)$ (OBS although this time complexity is based on the worst case scenario, but it is still reliable since the time complexity even i other cases are an $O(n^2)$ type)

Merge sort

The last algorithm is Merge sort. This algorithm can be described as follows: In short, in a function called ”sort” an unsorted array will be divided

into two subarrays those subarrays will be sorted and the merge function will merge those two sorted subarrays into one sorted array. For sorting these two subarrays a recursive programming will be used and for each of these two subarrays will the function "sort" be called. Then every of these two subarrays will be divided into two other subarrays again. The "sort" function will be called again and divide these subarrays once again into two other subarrays which are even shorter. This recursive procedure will go on until the subarrays consist of just one element and then the merge function will combine these subarrays into one sorted subarray. Merge function will return the sorted subarray to the etapp before and do the same and return the sorted subarray to the stage before etc, at the end it reaches the first stage and merge the two sorted subarrays into one sorted array and then the program is done. The array which the program returns will be a sorted array.

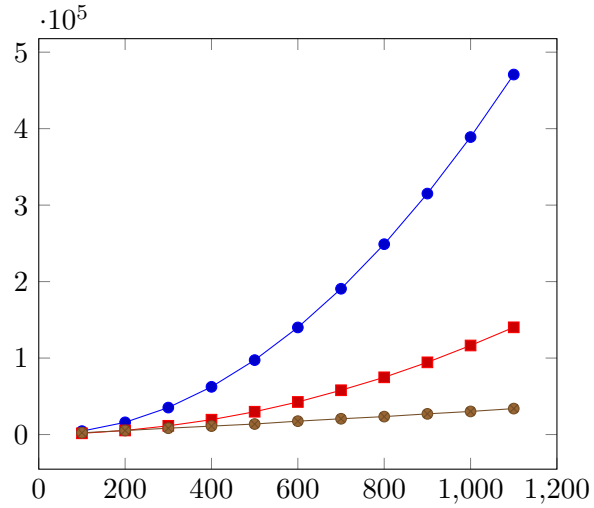
Time complexity of Merge sort consists of two part, dividing arrays and then merging the subarrays. The dividing part will have a time complexity of O(log n) type (for the same reason as the binary search, dividing by 2 will have a t.c. of (log n) type) and the merging part will have a time complexity of O(n) type (the t.c. for merging subarrays into one sorted array will be linear and proportional to the arrays size) . Totally the Merge sort will have a time complexity of O(n*log n) type. A function which roughly describes the execution time for Merge sort can be as follows:
t(n) = n * log n (t=time complexity in ns, n=number of elements in the array)

Benchmark

In this section these three algorithms will be benchmarked with different array sizes. Look at the Table 1 and Graph 1. (the benchmark was done by 11 different arrays sizes but to save spaces just 6 of them are presented in the Table 1)

number	**Selection sort**	**Insertion sort**	**Merge sort**
100	4500	1600	2300
200	16000	5300	5300
300	35300	11400	8300
400	62400	19400	11000
500	97300	30000	13800
600	140000	42600	17500

Table 1: The runtime for benchmarking of Selection sort, Insertion sort and Merge sort with different array sizes.



Graph 1: Results from benchmarking of Selection sort(blue line) , Insertion sort(red line) and Merge sort(brown line). x-axis, size of arrays, y-axis execution time in ns.

Conclusion

The result from benchmarking shows that the runtime efficiency of these three algorithms differ significantly from each others. The execution time for the Selection sort is much longer than the two other algorithms. The execution time for the Insertion sort is longer than the Merge sort when the array size is larger than 200 elements but it is shorter when the array size is up to 200 elements. (look at the Table 1, columns 2 and 3) For larger data the Merge sort is much efficient regard to the execution time compared to the two other algorithms since the time complexity for this algorithm is O(n*log n) and for a large (n) the log n can be seen as a constant and the time complexity will behave as a linear function. (look at the brown line in Graph 1. OBS! the brown line can be mistaken as a constant function because of the scale in the y-axis is much larger than the x-axis. But the brown line is a linear function)