Sorting an array in Java

Emanuel Paraschiv

Fall 2024

Introduction

As we saw before, sorted data is crucial for a faster execution time. In this assignment we will test different types of algorithms for sorting, observe their behavior and decide which is the best for a given practical situation.

Selection Sort

We will begin with Selection Sort. This algorithm is straightforward: in an array of n elements, we start at the first position, search for the smallest element until the end, and then swap it. Then we start again from the second position and follow the same procedure.

This way, if the size is n, then the complexity should be once n as we go through each position and n again as we search for the smallest element to come in that position $\rightarrow n \cdot n \rightarrow \mathcal{O}(n^2)$. After benchmarking different array sizes, I obtained the following execution times:

Input Size	**Time (μs)**
10	0.4
100	7
200	20
400	80
1000	400
2000	1500
5000	8400

Table 1: Selection Sort Benchmark

An equation to predict the execution time given the size of the array is:

$$f(x) = 2x^2 + 458x + 893 \tag{1}$$

Insertion Sort

Insertion sort is another algorithm: it works by picking each element and shifting the larger ones to make space, then inserts it into its correct position. This way, the sorted section grows from the start to the end of the list:

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

As we can see, each element is taken as a candidate and checked with the previous one. If it is the case for a swap (checked inside the nested `for()` loop), then we call the `swap()` method to do the swapping:

```
int temp = array[index1];
    array[index1] = array[index2];
    array[index2] = temp;
```

Both Selection and Insertion Sort algorithms have a complexity of $O(n^2)$. However,an interesting case comes with Insertion Sort: if the array is already sorted, its time complexity drops to $O(n)$. This occurs because Insertion Sort will traverse the array once (of size n) without doing any swaps.

The first table shows benchmarks of Insertion Sort on an unsorted array, while the second table presents results for an already sorted array:

Input Size	**Time (μs)**
10	0.6
100	3
200	10
400	40
1000	200
2000	800
5000	4800

Table 2: Insertion Sort Benchmark

Input Size	**Time (μs)**
10	0.1
100	0.2
200	0.3
400	0.4
1000	0.6
2000	1
5000	2

Table 3: For sorted array

The execution time dropped to nearly half compared to Selection Sort, proving the efficiency of Insertion Sort. The execution time as a function of input size is given by:

$$f(x) = 0.7x^2 + 200x - 650 \tag{2}$$

However, Insertion Sort will work worse on arrays sorted in reverse order, as it will need to perform many swaps to arrange the elements.

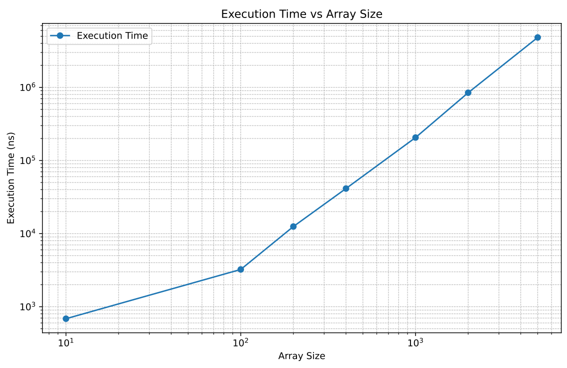

Figure 1: Plot of Insertion Sort with different input sizes

Thus, Insertion Sort excels with nearly sorted arrays due to its adaptability, while Selection Sort is better when we need to minimize memory writes or maintain a consistent performance.

Merge Sort

A more interesting algorithm for sorting, different from the previous 2 is Merge Sort. This one works by breaking an array in half, sorting each individual segment and merging them in the end in order.

The magic comes in the sorting of each half: instead of using any other procedure to sort the parts, the algorithm will call itself recursively until the array is split in individual entries:

```
if (lo < hi) {
    int mid = (lo + hi) / 2;
    sort(org, aux, lo, mid);
    sort(org, aux, mid + 1, hi);
    merge(org, aux, lo, mid, hi);
```

These elements are then merged in ascending order.

As we can see, the execution time is even smaller than of the previous, Insertion Sort, algorithm (See Fig.2 below). The time complexity is also different as it can be described by $\mathcal{O}(n \log_2(n))$. In the following figure we can see a graph which compares the result to the normalized plot of the function $n \log_2(n)$.(See Fig.3)

An equation to describe the execution time given the array input size is:

$$f(x) = 90 \cdot x \cdot \log_2(x) - 3300 \tag{3}$$

Input Size	Time in μs
10	1
100	20
200	30
400	60
1000	250
2000	550
5000	1000

Figure 2: Merge Sort Benchmark

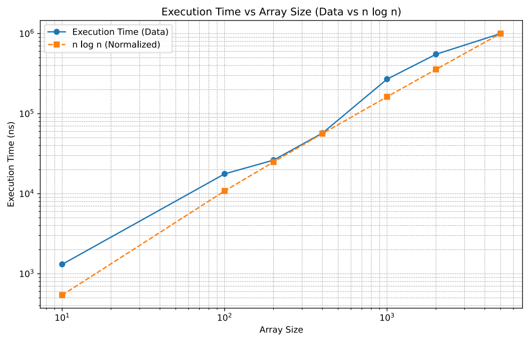

Figure 3: Execution time compared to $n \log_2(n)$

Updated Merge Sort

There is more we can do. Even though merge sort is a great algorithm, it has a disadvantage; For each merge operation, the elements from the original array are copied to an auxiliary array:

```
private static void merge(int[] org, int[] aux, int lo, int mid, int hi)
    for (int k = lo; k <= hi; k++)
        aux[k] = org[k];
    ...
```

This copying happens every time the merge step is performed, and it can be a problem when dealing with larger arrays.

However, we can eliminate unnecessary copying during the merge step by swapping the roles of the two arrays (org and aux) in each recursive call.

We start with the original array and make only one copy of it to the auxiliary:

```
public static void sort(int[] org)
    ...
    // copy org array to aux array
    for (int i = 0; i < org.length; i++)
        aux[i] = org[i];
    ...
```

Given this, in the recursive sort method we no longer need to copy in every merge step. Instead, we alternate between sorting in the original array (org) and the auxiliary array (aux): first we sort into `aux` and merge into `org` and then sort into `org` and merge into `aux`. By switching the roles of the original and auxiliary arrays during recursive calls, the updated algorithm eliminates the need for redundant copying and results in a more efficient sorting process.

This change is especially great for large arrays, where constant copying can become a problem.

After running the benchmark again with this improved Merge Sort, I was able to notice a difference in the execution time, which is displayed in the following figure:

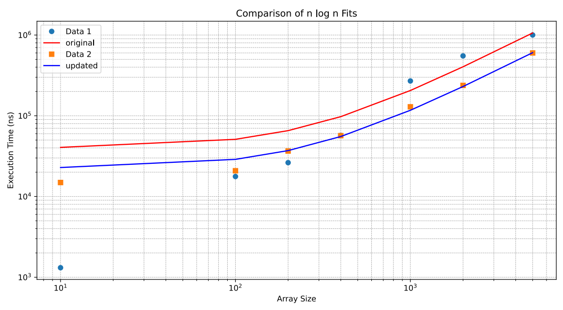

Figure 4: Comparison: Old and New Merge Sort Implementation

Conclusion

Thus, we explored different sorting algorithms and observed their performance. **Selection Sort** and **Insertion Sort** are simple to implement, but they are constrained by their $\mathcal{O}(n^2)$ time complexity and are therefore not suitable for larger arrays. **Merge Sort** came to solve this problem, not only with its improved time efficiency but also with its stability, as it preserves the relative order of elements .

In the end, by reducing the need for excessive copying we made our algorithm more memory efficient, resulting in faster execution times.