ID1021: Sorting

Alexander Lundqvist

13-09-2022

1 Introduction

In this report we examine three different search algorithms, measure their respective running times and compare the results. The algorithms in question are selection sort, insertion sort and merge sort.

2 Method

For the experiment we create three different classes for easier testing and to make the code reusable for future projects. The benchmark method is the same for all classes and is defined as:

```
public long benchmark(int size, int iterations) {
    long min = Long.MAX_VALUE;
    for (int i = 0; i < iterations; i++) {
        int[] array = unsortedArray(size);
        long t_start = System.nanoTime();
        sort(array);
        long t = System.nanoTime() - t_start;
        if (t < min) {
            min = t;
        }
    }
    return min;
}
```

The selection sort code was based on the template given in the instructions but I had already implemented both insertion and merge sort the previous year I just used my old code. Instead of bloating the report with code I refer to my github repository.

3 Result

3.1 Selection sort

In figure 1 we see the results of the benchmark. The graph clearly has an upwards curve but it is difficult to see. This is probably because the maximum array size is only 200. The reason for this was mostly to my computer being slow and I reasoned that the result was enough since the time complexity will be explained in the discussion.

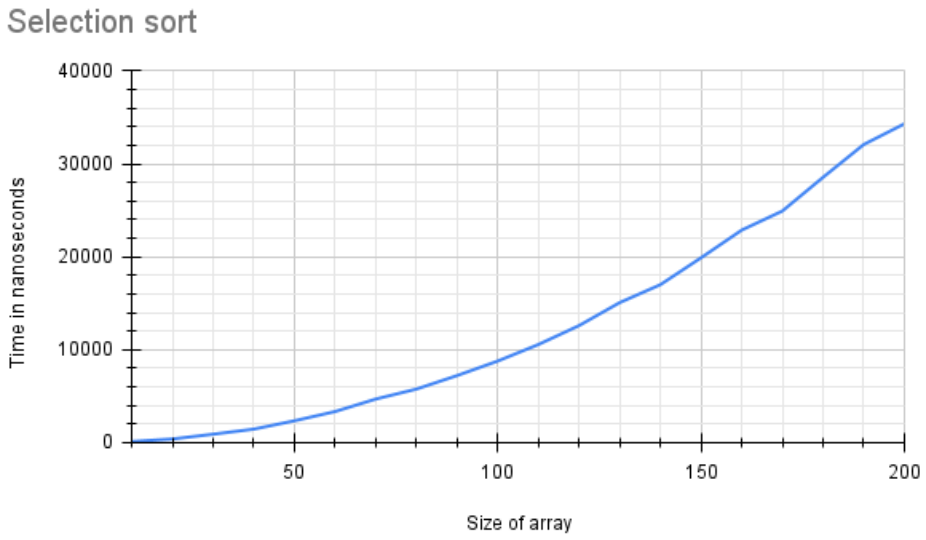

Figure 1: Selection sort running time vs. array size.

3.2 Insertion sort

Again, as with selection sort I determined that the were enough for the discussion. We can however see that the execution time is improved from the selection sort.

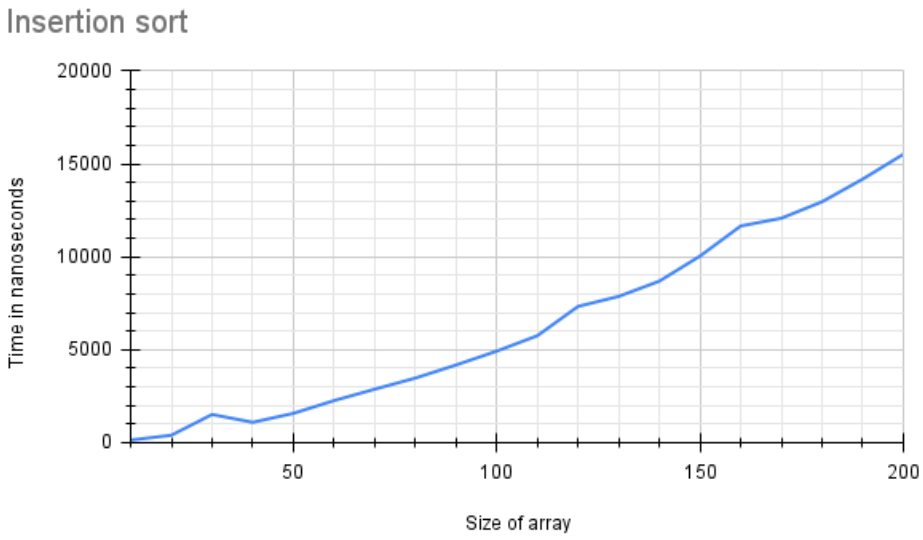

Figure 2: Insertion sort running time vs. array size.

3.3 Merge sort

We can see from figure 3 that merge sort has even better execution time (at size 200). For this benchmark I had to increase the maximum array size to get any meaningful values. The graph looks quite linear but we can see that it slightly bends upwards. If we compare this to known time complexities we can assume that it is $O(n*log(n))$. This will be proven in the discussion.

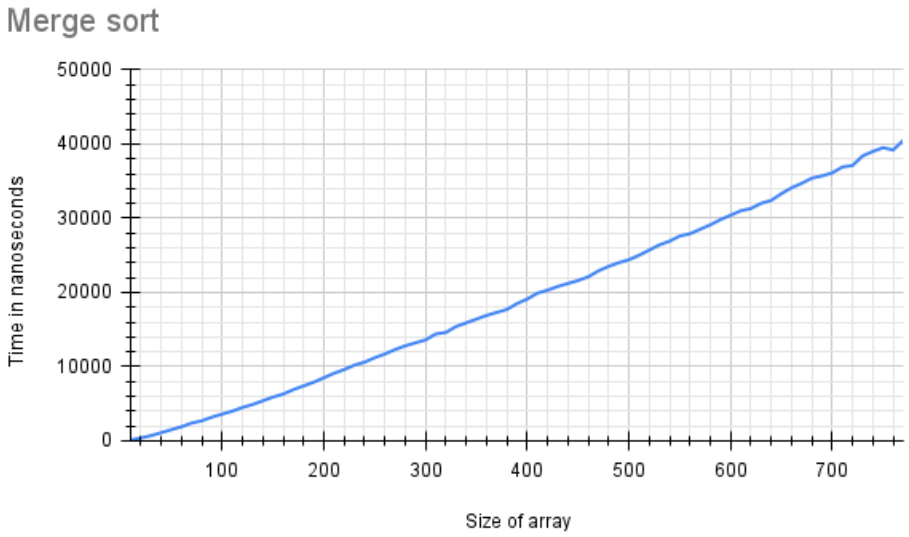

Figure 3: Merge sort running time vs. array size.

4 Discussion

4.1 Selection sort

Selection sort is a very simple algorithm which is very easy to implement. It is however very inefficient for large arrays and there are similar algorithms that are just as simple and perform much better, such as insertion sort. As such, there aren't any reason to use it other than for educational purposes.

By examining the graph in figure 1 we can see that the execution time gradually increases with array size. The time complexities that can produce such graphs are usually some $O(n^x)$ where $x > 1$. The graph is a bit hard to decipher in this regard however, we can assume the time complexity just by examining the code. Since we have two loops nested where the inner runs $1, 2, 3, ..., n-1$ times we can describe the sum of all inner loop executions as $\frac{n(n-1)}{2}$. When we expand this fraction we get $\frac{1}{2} * (n^2 - n)$ and since we can omit both constant factors and n of lower order when dealing with large n,

we get that the time complexity is $O(n^2)$.

4.2 Insertion sort

As mentioned earlier, a better way to sort algorithms is insertion sort. Insertion sort works by "dividing" the array into one that is sorted and one that is not.

The time complexity can be described in the same way as selection sort so no need to write the calculations again. The difference here as opposed to selection sort is that there is a best case scenario where the algorithm only has to do one swap per each outer loop iteration, therefore we would get a $O(n)$ in time complexity. This behaviour might be responsible for the faster execution time when we compare the graphs even though both have worst case $O(n^2)$.

Insertion sort however is known for being useful when dealing with smaller array sizes and partially sorted arrays. It is also stable and in-place, which means it uses no additional memory. It is sometimes used for optimizing more advanced sorting algorithms.

4.3 Merge sort

I had already implemented merge sort from last year so I used my old algorithm. Naturally, I didn't encounter any issue because of this. The only issue I had was with the benchmark method where I got very weird values when dealing with larger arrays. I remembered something that was mentioned earlier in the course that it is sometimes better to measure the minimum value instead of average to avoid abnormal values. I modified my old benchmark method accordingly and received better results.

Merge sort is as stated in the instructions, a very efficient sorting algorithm. It very much better at handling large arrays than the previous mentioned algorithms due to the fact that it splits the list into progressively smaller chunks which are easier to handle. A thing to mention is that the algorithm is stable, which means that it preserve the general order of the elements, i.e if we have two identical elements in the unsorted array, they will have the same relative position in the sorted array. This is relevant in the case of sorting arrays with objects.

Now to the complexities of the algorithm. Since we are dealing with an auxiliary array for the sorting process and this array is a copy of the source array, we have to allocate extra memory. The memory is dependent on the size of the array and thus the memory complexity of the algorithm is $O(n)$. This means that the algorithm is not in-place. As for the time complexity it was mentioned in the result that it should be $O(n * log(n))$. We can prove this by looking at the algorithm. Say we start with array size 2, then the algorithm has to split the array once before merging. If we continue

with size 4, we get two splits. Size 8 gives us 3 splits. Size 16 gives us 4 splits. As we go higher we can see a pattern that the amount of splits that need to be done can be determined by $n = 2^{splits}$ which can be written as $splits = log(n)$. As a split operation has complexity $O(1)$ we can say that the process of dividing the array has the time complexity $O(log(n))$. When the array has been completely divided it is time to recursively merge and sort it. The merge algorithm in itself is dependant on the size of the input and has a time complexity of $O(n)$. With each merge step the sum of all elements to merge are always n, therefore each step takes n time. Since the amount of steps can be calculated with $log(n)$ we get that the total time complexity is $O(n * log(n))$.

Figure 4 illustrates this process.

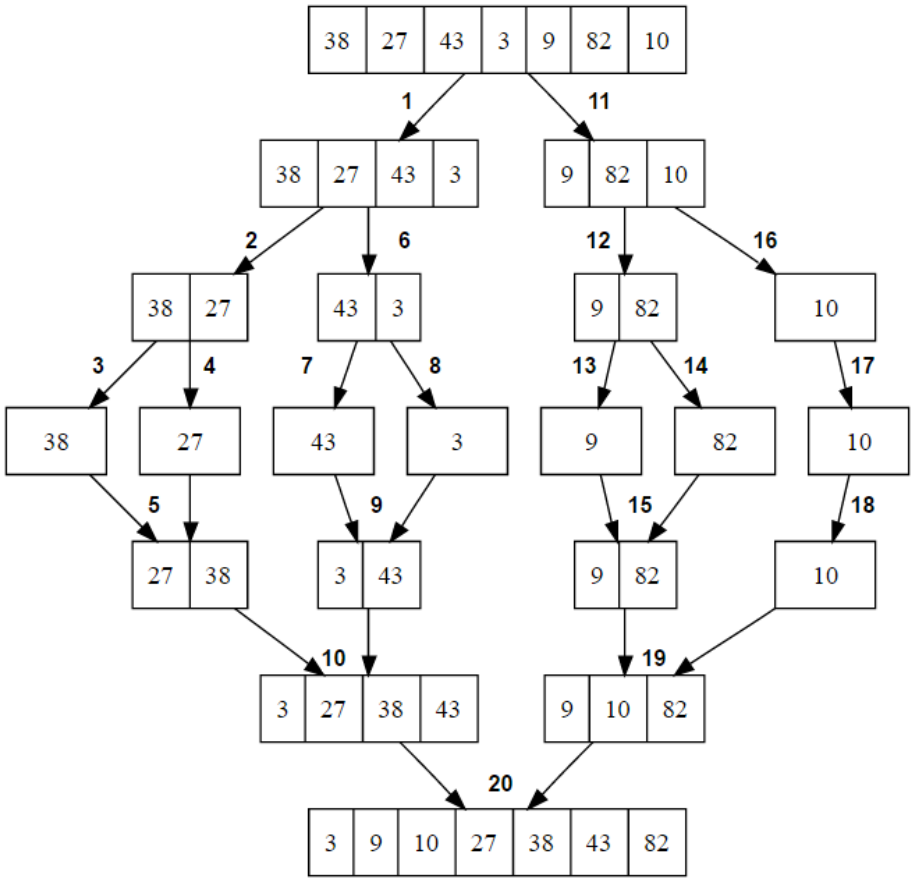

Figure 4: Merge sort steps.

Merge sort can also be optimized by using insertion sort for smaller algorithms as we mentioned earlier. We do this by introducing a cutoff value where we switch to insertion sort. This cutoff value is largely dependent on

what system the algorithm is run on but from testing I did in a previous course round the value is around 10.