Sorting arrays in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment the task is to compare different sorting algorithms. In last weeks assignment we looked at searching algorithms and concluded that for large arrays we can search much faster when the array is sorted. The worst searching algorithm we saw was O(n) which we managed to improve to one of O(log(n)) for sorted arrays. Now let's see how the cost of sorting compares.

Selection sort

One of the most intuitive ways to sort is to go through your list of numbers, pick the lowest number, place it at the left-most position and move one step to the right and so on...
To determine it's time complexity, we can easily see that for the first position, we have to compare the first element with (n-1) elements, for the second with (n-2) elements, etc. The total number of comparisons are n+(n-1)+...+1 = $\frac{n(n-1)}{2}$, so it's time complexity is $O(n^2)$. To confirm, I ran a benchmark and got the result in the table below.

array size (n)	execution time µs
100	4.8
200	13
400	38
800	125
1600	440
3200	1550
6400	5850
12800	22700

Insertion sort

The next sorting algorithm starts ordering the list from the first index, always inserting the next value to the right spot, so that after the k^{th} step the first k elements are sorted. I have my code below, where `swap()` is just a function that swaps the elements at the given indices in the given array.

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

Benching this gives me the results shown in the table below.

array size (n)	execution time [µs]
100	1.5
200	5
400	15
800	50
1600	180
3200	720
6400	2850
12800	11500

Now this is definitely faster (about half the time it took for selection sort) but it still has a time complexity of $O(n^2)$.

Merge sort

Next we look at an algorithm called merge sort, that takes an entirely different approach. It uses recursion to sort smaller sections of the array at a time and then merges them back together, hence the name. It is a bit harder to understand conceptually, so I included a code snippet from my implementation here.

```
private static void sort(int[] org, int[] aux, int lo, int hi) {
    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);
    }
}
```

Basically it breaks the array into two smaller sections to sort, calls itself on both, so these sections get broken into another two-two sections and so on, until it cannot divide it any more, and then starts merging them back together (if lo==hi it falls out of the current function call and continues with the merge part of the one before,...).

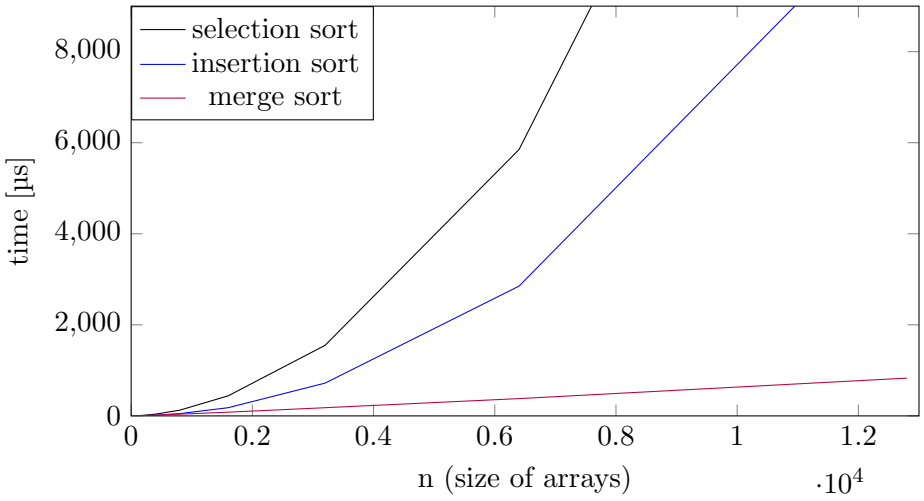
```
private static void merge(int[] org, int[] aux, int lo, int mid,
int hi) {
    for (int i = lo; i<=hi; i++ ) {
        aux[i]=org[i];
    }
    int i = lo;
    int j = mid+1;
    for ( int k = lo; k <= hi; k++) {
    if (i>mid){
        org[k]=aux[j++];
    }
    else if (j>hi){
        org[k]=aux[i++];
    }
    else if (aux[j]<aux[i]){
        org[k]=aux[j++];
    }
    else org[k]=aux[i++];
    }
}
```

Here we first have to copy everything over to the auxiliary array and start two indices, one for the lower half and one for the upper half of our array (since both are individually sorted), then we start building the sorted sequence in the original array. As long as both of our indices are within their range, we only have to compare the two elements at the indices to see which comes next. Since we saw that this algorithm utilizes the divide-and-conquer paradigm, we expect the time complexity to have something to do with log(n). Let's see if my benchmark confirms this.

array size (n)	execution time [µs]	insertion sort [µs]
100	3.3	1.5
200	7.7	5
400	17	15
800	38	50
1600	82	180
3200	180	720
6400	380	2850
12800	830	11500

Merge sort execution time compared to insertion sort

Based on the first few rows we might think that insertion sort is a better algorithm. However, as array sizes grow ($n > 400$), we notice how much slower merge sort follows. I visualise this relationship in the graph below.

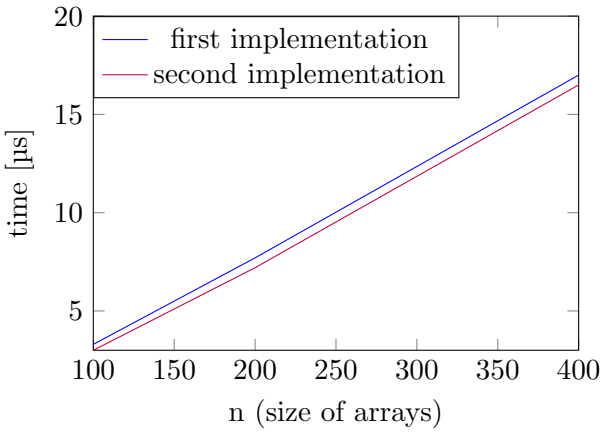


We can see that it is pretty fast, but definitely not O(log(n)), in fact, it grows steeper than linear. If you think about it, the partition part occurs log(n) number of times and the merging part of the algorithm is O(n), so the time complexity of the entire merge sort has to be $O(n \log(n))$. It is much faster than the previous two, however, it requires an additional array to work in, so there is a trade-off between time and space complexity.

Merge sort - just a little bit different

We later saw another implementation of this algorithm that gets rid of a lot of the copying between the two arrays by sorting in one and returning the result in the other (recursively, so always switching between the two). This trick is a bit harder to understand and doesn't significantly improve

the program, but it did save us a few µs, shown on the graph below.



Quick sort

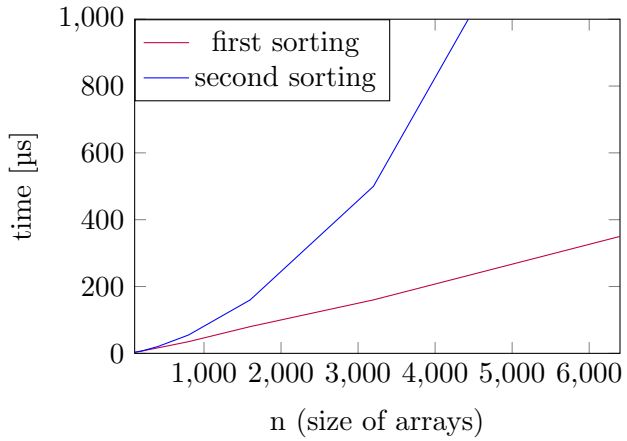
Another widely used algorithm that uses a divide-and-conquer strategy is quicksort. It first takes a pivot element and places everything lower than that to the left and greater than that to the right, then divides it into to and calls itself on both halves. Since it doesn't seem to be part of the assignment, I'm not going to go into too much detail here, but I implemented it using the first element as a pivot and Hoare's partitioning scheme. I set up a benchmark and decided to test it with random values, as I did for all the previous algorithms. I compare my results with merge sort in the table below.

array size (n)	quicksort [µs]	merge sort (ver2) [µs]
100	2.8	3
200	6.4	7.2
400	14.5	16.5
800	32.5	37
1600	70	80
3200	155	180
6400	340	390
12000	720	850

I did not expect this much of an improvement, especially since to my knowledge merge sort is supposed to be faster for larger arrays. The only explanation I could come up with is that my implementation of the merge sort is not a very efficient one. However, quicksorts arguably greatest drawback is that it is not always $O(n \log(n))$. Its worst case scenario (when the array is

already sorted) is $O(n^2)$. To see this in practice, I decided to run the sorting again on the already sorted array. The results are in the table below.

array size (n)	first sort [µs]	second sort [µs]
100	3	3
200	7.2	7.6
400	16.3	20.3
800	35	55
1600	80	160
3200	160	500
6400	350	1800



Another difference that may be important, is that merge sort (if implemented right) is a stable sorting algorithm (meaning that among equivalent values it doesn't change their initial order) whereas quicksort is not. This doesn't matter if we are only sorting integers for example, since we cannot differentiate between 5 and 5, but if we were working with objects that store multiple values that we can sort by, it is important to think about the algorithms stability.

Summary

Now that we saw the time complexity of searching and sorting algorithms, we can conclude that the question whether to sort before searching depends on how the data is handled. Unsorted search is $O(n)$, but sorting (using merge sort) and then binary search has a combined time complexity of $O(n \log(n))$, so if you only look up a few keys in the array before updating it, it's probably better to just search through it. On the other hand, if you search a lot compared to how often you write to it, it's probably worth to sort it and then maybe keep it sorted when writing.