Lab 2 explanatory text and time measurement graphs

Examples from executions: *examples from executions*

Question 1

For this question I made a classic insertionSort method that sorted whatever int array is input to it. This is done by going through every element in the array using a for-loop and seeing if the number to the left of it is smaller (using a loop in the for-loop), if so we switch places on them and continue until the element to the left isn't smaller or there are no more numbers to the left. The total amount of swaps is calculated by adding 1 to a counter every time an element is swapped. Printing out the array in several places is done using a method that goes through the input parameter array using a loop and printing out every element, with a comma after it.

Question 2

This question was solved by checking every number from left to right in the array with a for-loop. When checking a number I also checked every number after it using a for-loop inside the previously mentioned for-loop. The inner for-loop also checked if the number checked that is to the right was bigger than the one the outer loop was on, if so it added the numbers and indexes to a string and also added 1 to a counter. This was so that we would know the number of inversions and what elements on what indexes would be inverted.

Question 3

This question was solved by having a function that takes an int array and an int as parameters, this is the array that has to be sorted and the length of it.

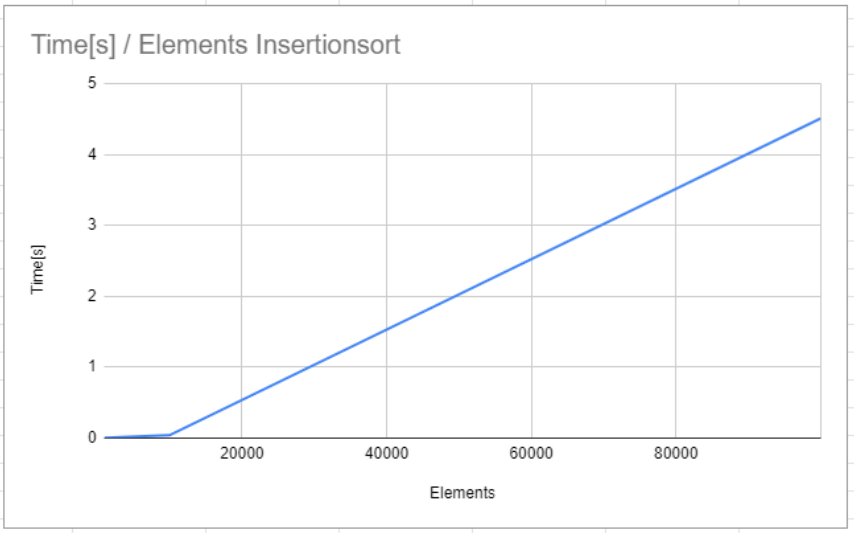
Question 4

The insertion sort algorithm is almost exactly the same as the one in question 1 but without printing out every step it makes.

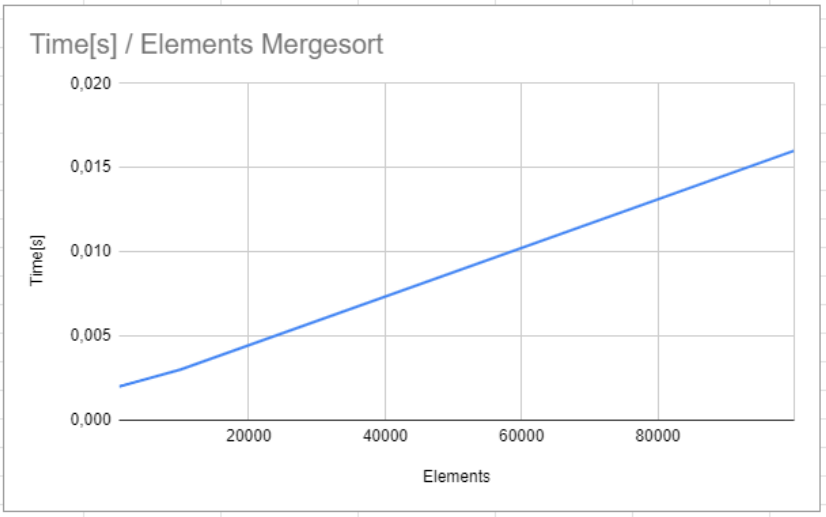
The mergesort was work like a mergesort usually does. So it first breaks the input into two halves and does this until you are left with arrays that can't be halved. Then we sort these arrays and merge the halves that were sorted recursively, when merging we make sure that they are merged so that they are in order of course.

For the quicksort I chose to have the pivot element as the element most to the right to begin with, since there wasn't any need to choose any specific pivot element. What quicksort does is that it uses this pivot and from it it sorts so all elements that are lower than the pivot are to the left and all elements that are bigger than the pivot are to the right and then the pivot is set in the middle of these. This is done recursively with the number lower and higher than the pivot element until all of these are sorted and we end up with a sorted array.

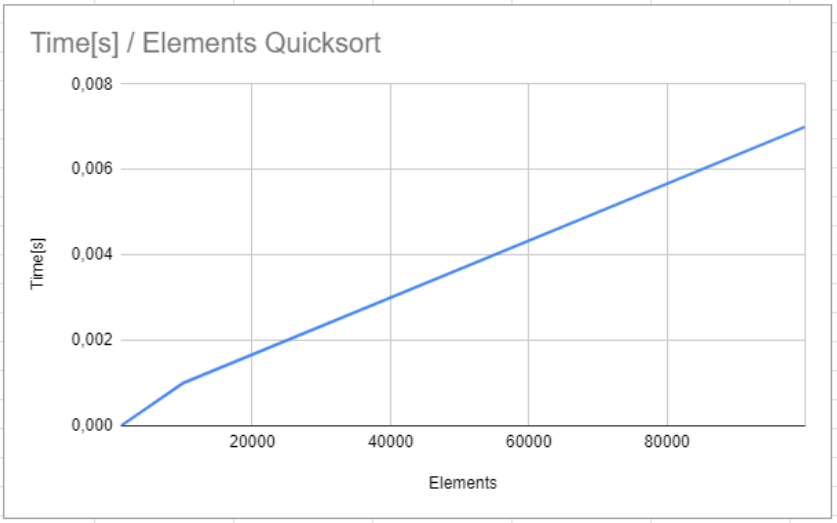
Elements	Time[s]
1000	0,003
10000	0,037
100000	4,514



Elements	Time[s]
1000	0,002
10000	0,003
100000	0,016



Elements	Time[s]
1000	0
10000	0,001
100000	0,007

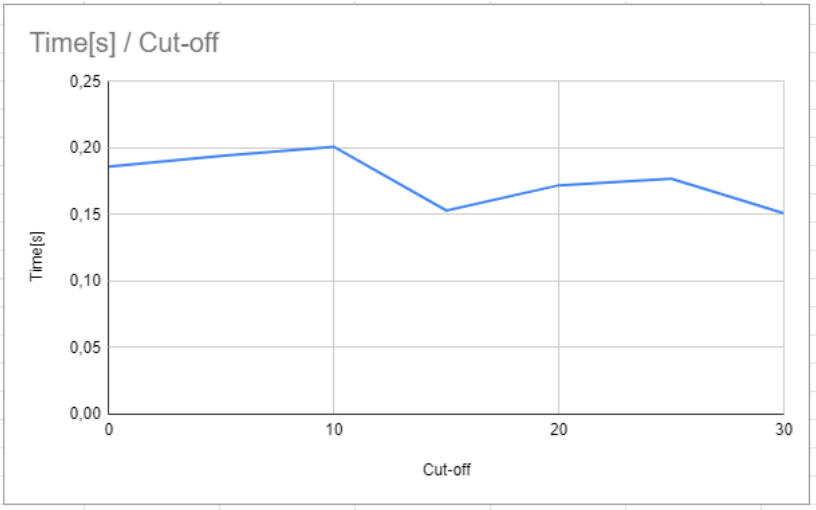


As we can see in the graphs above, mergesort and quicksort didn't get affected a lot from the higher number of elements in the array. However insertion sort was fast for a small number of elements but for more elements it got very slow in comparison to the other sorting algorithms

Question 5

The algorithm is very similar to the one in question 4 so I will focus on discussing the time results.

Cut-off	Time[s]
0	0,186
5	0,194
10	0,201
15	0,153
20	0,172
25	0,177
30	0,151



What we can see is that with the values between 0 and 30 it seems to be the worst with values under 10. After this it goes down a bit and seems to stabilize a bit. So what we can learn from this is that a cutoff value can be good but it shouldn't be too low. We haven't done a more extensive study so we can't be sure but that is what this graph tells us likely is true.

Question 7

To solve this question I went through every element in the array before sorting it and made all of them their reverse (so positives became negative and negatives become positive). After I had sorted I reversed it back. Of course this does so the array is sorted in reverse order since the numbers are more or less reversed since they are their negatives. My solution added O(N) operations since going through all elements took N operations and I did it twice. This gives me N + N which is O(N) since if N is really big N+N is very similar to N and not similar to for example N^2

Generating input

I did this to generate a random file:

The first argument sends in a time in nanoseconds so we get a random number from it basically, this is a good seed so we get a random seed. A random seed is better because if we use the same all the time we will get the same random numbers. The two other arguments are just how long every element should be at max and how many elements should exist. And at the end we tell the console what the text file should be called. Random is better than rand since it is improved so that Random is more random for the lower bits and higher bits, rand isn't as random for both lower and higher bits.