1 General remarks

All answers can be put into a simple text file. Questions are enumerated **Q1**, **Q2**, and so on.

Topic of this lab: We will study implementations of algorithms which were discussed earlier during the week in the course. The main tasks will usually be to understand these algorithms, and to make some experiments with them.

This week, we will look at an implementation of the INSERTIONSORT algorithm. We will try to empirically verify our runtime bounds on upwards sorted lists, downwards sorted lists and random lists, which we computed theoretically in the lecture.

2 Java implementation of InsertionSort

Read the code of the program `InsertionSort.java`, which you find in `Canvas/Modules` under "Programs for labs". You may download it, compile it with `javac InsertionSort.java`, and run it on your own, but you don't need to, since below all outputs are already provided.

This command-line program reads the size of the array n, and measures the runtime in μs (microseconds, that is, 10^{-6}s) for three types of arrays:

- already sorted (i.e., ascending order)
- reversely sorted (i.e., descending order)
- random order.

For example for $n = 1000$:

```
> java InsertionSort 1000
1000 1.887000 322.068000 159.842000
```

Q1 The sorting algorithm is implemented in the function `sort`. What are the two key differences to the implementation presented in the lecture (see the script for Week 1)?

Q2 Time measurement takes place in the function `experiment`. How does the time-measurement work? Is this what you expected, and if not, what might be the reasons for doing it this way?

3 Numerical evaluation

Running the implementation on the above machine for $n = 1000, \ldots, 5000$, we got:[1]

[1] See the Appendix for the command run here.

```
1000 1.889000 322.067000 159.542000
2000 3.709000 1253.275000 636.955000
3000 5.534000 2791.553000 1376.475000
4000 7.365000 4952.496000 2417.846000
5000 9.194000 7695.331000 3755.836000
```

The task is now to explain the data, connecting it to the theoretical results from Week 1. You might use a calculator (as available in every desktop environment), but it is perfectly fine, and perhaps even preferrable, to do rough calculations in your head (or simple calculations on paper): getting an overview, that's the task.

An important tool here is to compare the five lines above via *quotients*:

1. The main quotient is given by the parameter n, the first column. Say you compare the fourth and the second line: the quotient q_1 is $q_1 = \frac{4000}{2000} = 2$.
2. For the three other quotients here we get:

$$q_2 = \frac{7.365000}{3.709000} = 1.98571,\ q_3 = \frac{4952.496000}{1253.275000} = 3.951643,\ q_4 = \frac{2417.846000}{636.955000} = 3.795945.$$

Q3 What is the rough empirical picture for the second column (the already sorted case)? That is, how do the values depend on n?

What is the underlying theoretical result?

Q4 Consider a runtime function $f(n) = \Theta(n^2)$. Consider two input-sizes n_1, n_2 with a quotient $n_2/n_1 = q$ (thus $n_2 = q \cdot n_1$). What is the "average" value of

$$q' := \frac{f(n_2)}{f(n_1)}$$

?

Q5 How does the third column depend on n? What is the underlying theoretical result?

Q6 How does the fourth column depend on n? What is the underlying theoretical result?

Q7 How do columns four and three relate?

Q8 Consider the table "Time to solve a problem instance of size n" from the script of Week 1: how do these calculations relate to the above data?

A Some technical details

The command-line instruction on Linux for the little experiment in Section 3 is:

```
> for n in {1000..5000..1000}; do java InsertionSort $n; done
```