1 Assignment No. 1: Analysis & Comparison of Direct Sorting Methods

Allocated time: 2 hours

1.1 Implementation

You are required to implement **correctly** and **efficiently** 3 direct sorting methods (*Bubble Sort*, *Insertion Sort* – using either linear or binary insertion and *Selection Sort*)

- Input: sequence of numbers $< a_1,\ a_2,\ \ldots,\ a_n >$
- Output: an ordered permutation of the input sequence $< a'_1 \leq a'_2 \leq \ldots \leq a'_n >$

You may find any necessary information and pseudo-code in the **Seminar no. 1 notes** (Insertion Sort is also presented in the **book[1]– Section 2.1**). Make sure that you implement the efficient version for each of the required sorting methods (if more than one version has been provided to you).

Minimal requirements for grading

- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- Prepare a demo for each algorithm implemented.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- ***The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and the correct answer to the questions from the teacher.***

1.2 Requirements

1.2.1 Implementation of direct sorting method (5.5p)

- Bubble sort (1.5p)
- Insertion sort (2p)
- Selection sort (2p)

You will have to prove your algorithm(s) work, so you should also prepare a demo on a small-sized input (which may be hard-coded in your *main* function).

1.2.2 Evaluate algorithms for the average case (1.5p – 0.5p for each algorithm)

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm!

You are required to compare the three sorting algorithms, in the **best**, **average** and **worst** cases. Remember that for the **average** case you have to repeat the measurements **m** times (m=5 should suffice) and report their average; also, for the **average** case, make sure you always use the **same** input sequence for all three sorting methods. To make the comparison fair, make sure you know how to generate the **best/worst** case input sequences for all three methods.

This is how the analysis should be performed for a sorting method, in any of the three cases (**best**, **average** and **worst**):

- vary the dimension of the input array (n) between [100...10000], with an increment of maximum 500 (we suggest 100).

- for each dimension, generate the appropriate input sequence for the sorting method; run the sorting method counting the operations (i.e., number of assignments, number of comparisons and their sum).

> **!** Only the assignments (=) and comparisons (<, ==, >, !=) which are performed on the input structure and its corresponding auxiliary variables matter.

1.2.3 Evaluate algorithm for best and worst case (3p - 0.5p for each case of each algorithm)

For each analysis case (**best**, **average,** and **worst**), generate charts which compare the three methods; use different charts for the number of comparisons, number of assignments and total number of operations. If one of the curves cannot be visualized correctly because the others have a larger growth rate (e.g., a linear function might seem constant when placed on the same chart with a quadratic function), place that curve on a separate chart as well. Name your charts and the curves on each chart appropriately.

1.2.4 Bonus: Binary insertion sort (0.5p)

You will have to prove your algorithm(s) work, so you should also prepare a demo on a small-sized input (which may be hard-coded in your main function).

You will have to compare binary insertion sort against all other sorting methods (bubble, insertion, selection) on all cases (best, average, worst).

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.