1 Assignment No. 3: Analysis & Comparison of Advanced Sorting Methods – Heapsort and Quicksort / QuickSelect

Allocated time: 2 hours

1.1 Implementation

You are required to implement **correctly** and **efficiently** *Quicksort, Hybrid Quicksort* and *Quick-Select (Randomized-Select)*. You are also required to analyze comparatively of the complexity of *Heapsort* (implemented in Assignment No. 2) and *Quicksort*.

You may find any necessary information and pseudo-code in your course notes, or in the book(**?**):

- *Heapsort*: chapter 6 (Heapsort)
- *Quicksort*: chapter 7 (Quicksort)
- *Hybridization for quicksort using iterative insertion sort* - in quicksort, for array sizes < threshold, insertion sort should be used (use insertion sort from Assignment No. 1)
- *Randomized-Select*: chapter 9

1.2 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.***

1.3 Requirements

1.3.1 QuickSort: implementation (2p)

You will have to prove your algorithm(s) work on a small-sized input.

1.3.2 QuickSort: average, best and worst case analysis (3p)

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

This is how the analysis should be performed:

- 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 method; run the method, counting the operations (assignments and comparisons, may be counted together).

> **!** Only the assignments and comparisons performed on the input structure and its corresponding auxiliary variables matter.

1.3.3 Quicksort and Heapsort: comparative analysis of average case (2p)

You are required to compare the two sorting procedures in the **average** case. Remember that for the **average** case you have to repeat the measurements m times (m=5) and report their average; also for the **average** case, make sure you always use the **same** input sequence for the two methods – to make the comparison fair.

Generate a chart which compares the two methods under the total number of operations, in the **average** case.

If one of the curves cannot be visualized correctly because the other has a larger growth rate, place that curve on a separate chart as well. Name the chart and curves appropriately.

1.3.4 Implementation of quicksort hybridization (1p)

You will have to prove your algorithm(s) work on a small-sized input.

1.3.5 Determination of an optimal threshold used in hybridization + proof (graphics/ measurements) (1p)

You should vary the threshold value of quicksort hibridization for which insertion sort is applied.

Compare the results from the performance (number of operations and execution time) perspective for determination of the optimum threshold. You can use 10.000 as the fixed size of the vector that is being sorted and vary the threshold between [5,50] with an increment of 1 to 5.

The number of tests (nr_tests from the example) has to be chosen based on your processor and the compile mode used. We suggest bigger values such as 100 or 1000.

1.3.6 Comparative analysis (between *quicksort and quicksort hibridization*) from the operations and runtime perspective (1p)

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

For quicksort hybridization, you have to use the iterative insertion sort from the first assignment if the size of the vector is small (we suggest using insertion sort if the vector has less than 30 elements). Compare *runtime and the number of operations* (assignments + comparisons) for quicksort implemented in the third assignment with the hybrid one.

For measuring the runtime you can use Profiler similar to the example below.

```
profiler.startTimer("your_function", current_size);
for(int test=0; test<nr_tests; ++test) {
   your_function(array, current_size);
}
profiler.stopTimer("your_function", current_size);
```

When you are measuring the execution time make sure all the processes that are not critical are stopped.

1.3.7 Bonus: QuickSelect - Randomized-Select (0.5p)

You will have to prove your algorithm(s) work on a small-sized input.

For QuickSelect (Randomized-Select) no explicit complexity analysis needs to be performed, only the correctness needs to be demonstrated on sample inputs.