1 Assignment No. 2: Analysis & Comparison of Bottom-up and Top-down Build Heap Approaches

Allocated time: 2 hours

1.1 Implementation

You are required to implement **correctly** and **efficiently** two methods for building a heap, namely the *bottom-up* and the *top-down* strategies. Moreover, you are required to implement the *heapsort* algorithm.

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

- *Bottom-up*: section 6.3 (Building a heap)
- *Heapsort*: chapter 6.4 (The heapsort algorithm)
- *Top-down*: section 6.5 (Priority queues) and problem 6-1 (Building a heap using insertion)

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 Comparative analysis of *one* of the sorting algorithms from L1 (you choose) in *iterative* vs *recursive* version. The analysis should be performed based on the *number of operations* and the *runtime* (2p)

For the comparative analysis of the iterative vs recursive version pick one of the 3 algorithms from Assignment 1 (bubble sort, insertion or selection). Use the iterative version that you already implemented (corrected, if needed, based on the feedback received from the teacher) and implement the same algorithm in the recursive version.

You must measure the total effort and the running time of the two versions (iterative and recursive) => two charts, each of them comparing the two versions of the algorithm.

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);
```

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

1.3.2 Implement bottomup build heap procedure (2p)

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

1.3.3 Implement topdown build heap procedure (2p)

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

1.3.4 Comparative analysis of the two build heap methods in the average case (2p)

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

You are required to compare the two build heap 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.

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 (n), generate the appropriate input sequence for the method; run the method, counting the operations (assignments and comparisons, may be counted together for this assignment).

> **!** Only the assignments and comparisons performed on the input structure and its corresponding auxiliary variables matter.
>
> 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 your chart and the curves on it appropriately.

1.3.5 Comparative analysis of the two build heap methods in the worst case (1p)

1.3.6 Implement and exemplify correctness of heapsort (1p)

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