Uppsala University

Algorithms and Data Structures I
1DL210 Autumn 2020 - Group 13

Assignment 2

Copyright authors:
Henrik Schulze, Neethu Joseph

3rd October 2020

ABSTRACT

- Unless you have asked for and received special permission, solutions must be prepared in pairs, triples or quadruple. If you got special permission for the first assignment, you also have special permission for this assignment.

- The provided skeleton code is supposed to be run on the Unix/Linux system.

- If you are not using the python skeleton code, make sure that your program behaves similarly. In this case, you probably need to re-implement yourself a test() function to test your program.

- When you have completed the assignment, you should have four files: three sorting programs and one Pdf document explaining the differences between them.

- Submit the resulting these files to Studentportalen. Only one of you needs to do this.

1 Implementation of Sorting Algorithms

In this section, you are going to implement three different sorting algorithms based on their textbook descriptions that you can also find in the lecture slides. Each algorithm is to be implemented as its own program.

- Your program should read a file called `nums.txt`, which can be assumed to contain integers separated by newlines, sort them using the algorithm in question, and then output a file `nums_sorted.txt` of the same form but with all numbers sorted.

- Additionally, your program should verify that your result matches the output of the already given *bubblesort* algorithm.

Note that the arrays in the textbook are indexed from 1, whereas in most programming languages they are indexed from 0.

1.1 Skeleton Code

We provide some skeleton code in Python, which is the language we recommend for the assignments in this course. You are of course free to implement the algorithms in the language of your choice as long as the programs meet the requirements. The skeleton code consists of the following:

- `rangen.py` takes an integer argument n and outputs a file `nums.txt` containing n rows with random numbers in the range $\{0, \ ..., \ n-1\}$. Example of usage: `python rangen.py 61`.

- `sort.py` contains an implementation of the *bubblesort* algorithm along with the function test that verifies the output of your implementation matches the output of the provided sorting algorithm *bubblesort*. Example of usage: `python sort.py`. Look at function test for the code for time measurement, and use it in your implementation. Also, you need to use `import time` at the beginning of your file.

1.2 Insertion sort (2p)

Implement *insertion sort*.

Answer.
See the file `insertionsort.py` attached as a separate file.

1.3 Heapsort (3p)

Implement the functions *Max-Heapify*, *Build-Max-Heap*, and *Heapsort* as you have seen in the lectures.

Answer.
See the file `heapsort.py` attached as a separate file.

1.4 QuickSort (3p)

Start by implementing *Partition*. Then use it to implement *QuickSort*.

Answer.
See the file `quicksort.py` attached as a separate file.

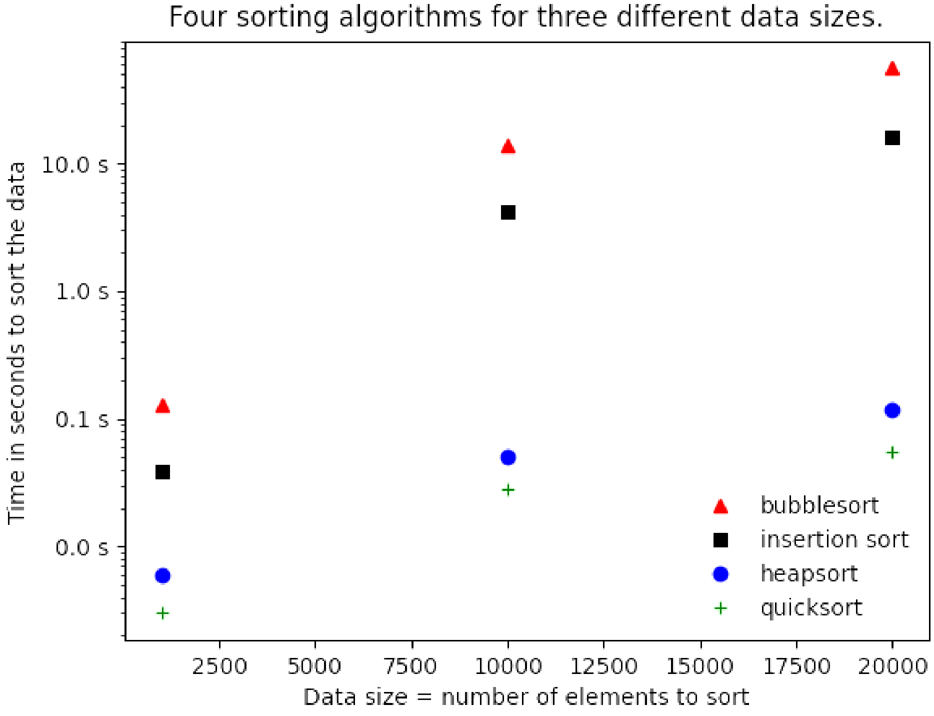
1.5 Comparison (2p)

Run your implementation of *InsertionSort*, *HeapSort* and *QuickSort* for five times for each category of input sizes 1000, 10000 and 20000. Measure the running times, calculate the average running time for each category of inputs. Do the same for *Bubblesort*. Compare average running times of four algorithms for each input category and plot a graph. What can we deduce?

Answer.
For *bubblesort*, see the file `bubblesort.py` attached as a separate file.

Our results are as shown in the following table, where all times are given in seconds.

	1000	10000	20000
`bubblesort`	0.128	13.881	55.969
`insertionsort`	0.039	4.212	15.926
`heapsort`	0.006	0.051	0.119
`quicksort`	0.003	0.028	0.055



We conclude that *heapsort* and *quicksort* are considerably faster than both *insertion sort* and *bubblesort* for large data sizes. This is well in line with what we should expect from the theoretical result saying that *heapsort* and *quicksort* both belong to the time complexity class $\Theta(n \log(n))$, where n is the size of the data (in the case of *quicksort* when data is “well shuffled”). Whereas *insertion sort* and *bubblesort* belong to $\Theta(n^2)$.