UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
**School of Computer Science & Applied Mathematics**

# Introduction to Algorithms & Programming
# Better sorting

## 1 Introduction

In this lab, we'll be extneding the previous lab to implement a few more of the sorting procedures we saw in the lecture notes. The main aim here is to measure the running time (wall-clock time) of each sort method empirically as the size of the input increases.

It will be useful to build on the code from the previous lab, so please go back and finish that (and read the hints and tips) if you haven't done so already!

**Please make sure to read exactly what needs to be uploaded to Moodle to be marked!**

## 2 Sorting algorithms

Build on the previous lab and additionally implement insertion sort, mergesort and quicksort, as presented in the class notes. It would be a good idea to verify that it works by testing it with some input and verifying that it does in fact sort the numbers. Each of these sorts should be written as their own separate function that accept a vector of integers and sort them. You should then have 6 sorting algorithms implemented — three from the previous lab, and the three mentioned here.

**Hint: For mergesort, be sure to allocate the temporary memory used in the merge procedure only once at the beginning! If you allocate new space every time the mergesort function is called, it will result in a massive slowdown of your code. If you have done everything correctly, the mergesort and quicksort algorithms should have almost the same performace!**

### 2.1 Time the algorithms

Use the pseudocode provided in the previous lab to time how long each of the 6 algorithms take as a function of the input size. Timing should be measured in nanoseconds. The input size should start at $N = 1000$ and go up to (and including) $N = 20000$ in steps of $1000$. Note that all algorithms should be given the same data. For example, for an vector of size $1000$, all algorithms should be given as input that exact same vector (not different vectors of size $1000$.) Average these results over $5$ runs (the vectors between runs should differ, so the vector of size $1000$ on run $1$ should be different from the same sized vector on run $2$).

### 2.2 Save the data

Save the results in a text file as a comma separated list. The text file should therefore contain 6 lines (one for each algorithm) and each line should contain 20 values, separated by commas. You may save the results manually, or use C++ to write them to file (the latter is obviously preferred).

### 2.3 Plot the data

Use the provided Python file on Moodle and modify it by providing the relevant information. Run this Python file to produce a plot of the running times of the six algorithms which will be displayed and saved to file (you will need to have matplotlib installed). Verify that the running times are as to be expected (with potentially different constants/lower-order terms). If they are not, you'll need to fix the mistake somewhere in your code!

### 2.4 Upload to Moodle

Once you are happy and satisfied with everything, upload the following to Moodle. **Each of these files will be marked, so missing files will mean missing marks! Please upload everything!**

1. A single C++ source file containing all the code you used to generate the timing data. Name this file `sorts2.cpp`.
2. The text file where the timing data was saved. Name this file `data2.txt`.
3. The Python file used to plot the graphs. Name this file `plot2.py`.
4. The PDF file with the actual plots. Name this file `sorting2.pdf`.