**DV1625 - Task A**
**Sorting and Analysis in Python**

## Summary

Must implement two hybrid sorting algorithms through a combination of the sorting algorithms Insertionsort, Quicksort, and Mergesort.

The three (3) original algorithms must first be implemented, test run, and runtime results must be generated. The results are analyzed and used to identify the breakpoints where the algorithms Quicksort and Mergesort are faster than Insertion Sort, and vice versa, by using theoretical knowledge regarding time complexity and asymptotic notation.

Finally, create two hybrid sorting algorithms based on the breakpoints, and empirically demonstrate that hybrid sorting algorithms' running time is generally faster than the original algorithms.

## Implementation and Report

The task is to create two hybrid sorting algorithms and write a report on the approach and the results obtained. This is done by first implementing Insertionsort, Quicksort, and Mergesort and then analyzing their performance to then create the two hybrids.

A report template is available alongside this document, pdf file. also find next to this document the file sorting.py as well as a zip file containing tests.py and a folder named Testdata. In sorting.py there are the following function templates to be implemented, preferably by further calling own functions:

Report writing and implementation
(a) Insertionsort
(b) Quicksort
(c) Mergesort
(d) Hybrid sorting, including Insertionsort and Quicksort
(e) Hybrid sorting, including Insertionsort and Mergesort

```
• insertionsort(lst: list) -> None
#- Sorts using the algorithm
# Insertion Sort and return nothing.
• quicksort(lst: list) -> None
#- Sorts the list lst using the Quicksort algorithm and
#returns nothing.
• mergesort(lst: list) -> None
#- Sorts the list lst using the Mergesort algorithm and
returns nothing.
• quicksort_hybrid(lst: list) -> None
# - Sorts the list lst using an algorithm that is a hybrid of
Quicksort and Insertionsort.
• mergesort_hybrid(lst: list) -> None
#- Sorts the list lst using an algorithm that is a hybrid of
Mergesort and Insertionsort.
```

In order to generate results that will then be analyzed, you need to create a program that runs the sorting algorithms on the files of different sizes in Testdata. The files to be used are 10 random.txt, 100 random.txt,
1000 random.txt, 10000 random.txt, and 100000 random.txt.

Each algorithm must be run at least five (5) times per data file and then an average value can be calculated and used. Note! Calculations that are performed must be reported in a clear manner.

**Checklista**

1. Here you will find a checklist of the parts of the task that need to be carried out. The report should be completed afterwards to facilitate writing. • Implement Insertionsort, Quicksort, and Mergesort and verify that they work by passing the tests in tests.py

2. Create a test program that runs each algorithm five (5) times per data file and prints each run time in the appropriate format. Timing is conveniently done with the module time and specifically the function perf_counter(). Calculate the average value of the runs and use this in your calculations, but don't forget to show all your run times and average values in appendices. Tip: There is a lot of time and headache to be saved if a good test program is created, where calculations are done and printing is done in a simple format for your reports.

3. Use the running times to calculate the constant c in the equation $T(n) = O(f(n)) \rightarrow T(n) \leq c * f(n)$ for the respective algorithm. Assume the average error for each algorithm as the data in Test Data is random.

4. Identify the breakpoints for n where Insertionsort starts to become slower than Quicksort and Mergesort. This must be done mathematically by identifying the point of intersection between the two functions and their constants. Tip: When do the functions become equal?

5. Implement the hybrid sorting algorithms based on the breakpoints and verify that they work.

6. Run the hybrid sort algorithms on all data files and compare against the original algorithms and analyze their running times. Investigate whether your estimations of the breakpoint are correct or if the running time can be further improved by changing the respective breakpoint.