Practical 6: Sorting

Task

Write a program that implements several sorting algorithms and use it to demonstrate the comparative performance of the algorithms for a variety of datasets.

SVN check out

Before you begin, you will need to check out the practical directory from the topic repository. This can be achieved by selecting <u>Get from Version Control</u> on the CLion welcome screen. In the window that appears, select <u>Subversion</u> from the dropdown list. If you have not previously connected CLion to the repository, click the <u>add</u> button (**+**) and paste the following URL into the text field (replacing **`FAN`** with your FAN):

```
https://topicsvn.flinders.edu.au/svn-repos/COMP2711/FAN
```

If you are prompted to login, use your University FAN and password. Expand the repository listing and select the **sorting** directory. Click <u>Check Out</u> then select a location to store your project (preferably in a practicals directory) and click <u>Open</u>. From the destination list, choose the second option to create a project directory named **sorting** and then click <u>OK</u> to complete the check out. A dialogue may appear confirming the subversion working format; if so, 1.8 is fine. Finally, when prompted if you would like to open the project, click <u>Yes</u>.

Automated marking

Levels 1 and 2 of this practical are available for automatic marking via quiz **Practical 6 Level 1 and 2**, which is located in the Assessment module on FLO. To assess your solution, first commit your changes to the topic repository. This can be completed within CLion via the <u>Commit</u> option under the <u>VCS</u> menu (or alternatively with Ctrl+K on Windows or ⌘+K on macOS).

You should adhere to good development habits and enter a commit message that describes what you have changed since the last commit. When ready, click <u>Commit</u> to upload your changes and receive a revision number. Enter this revision number into the answer box for the relevant question (level).

There are no penalties for incorrect solutions, so if you do not pass all test cases, check the report output, modify your solution, commit, and try again. Remember to finish and submit the quiz when you are ready to hand in. You may complete the quiz as many times as you like—your final mark for the practical will be the highest quiz mark achieved. If you do start a new quiz attempt, ensure you reassess any levels you have previously completed.

Levels 3 and 4 are about your program's performance not its function, so it does not make sense to use automatic marking. Instead, you will need to hand in your tabular data and plots as a PDF file to the **Practical 6 Level 3 and 4** submission box.

Background

The skeleton program **`sorting.cpp`** contains a main function for testing the operation of several sort algorithms over various data sizes and dataset organisations.

The program understands the following arguments:

`-i`	insertion sort
`-s`	selection sort (default)
`-q`	quicksort
`-a`	(already) sorted dataset
`-v`	reverse-sorted dataset
`-r`	random dataset (default)
`-n`	no sorting
`x`	generate a dataset of size *x* (default 10000)

Level 1: Basic sorts

Implement the **`selectionSort`** and **`insertionSort`** functions. Note that you can base your code on the sample code used in lectures, although you will need to modify it from passing the data using an array and two indexes to passing it using two pointers. The program will check that the final list is sorted correctly.

Level 2: Quicksort

Implement the **`quickSort`** function. The real work of quicksort happens in the partition operation, so it is probably best to define a separate function to do the partitioning.

Level 3: Profiling

Gather data on the number of times that each algorithm compares and swaps items. A simple strategy is to use variables to count the comparisons and swaps. The skeleton code declares **`comparisons`** and **`swaps`** variables for just this purpose. To make use of them, modify your code to increment each variable where appropriate and then uncomment the output statements to display their final values.

When you have profiling working, run the program and record the number of compares and swaps for each combination of algorithm (**`-i`**, **`-s`**, and **`-q`**) and dataset organisation (**`-a`**, **`-v`**, and **`-r`**). Use dataset sizes of 1000, 2000, 5000, 10,000, 50000, and 100000. In the worst case, execution should not take much longer than 30 seconds.

For each run, compute a normalised execution count by dividing the actual count by the dataset size. For example, if quicksort with a random dataset containing 10000 elements performed 161619 comparisons, the normalised count would be 161619/10000 or about 16.2.

Finally, plot the comparison results (for the options **`-ia`**, **`-iv`**, **`-ir`**, **`-sa`**, **`-sv`**, **`-sr`**, **`-qa`**, **`-qv`**, and **`-qr`**) as separate lines on a single set of axes with dataset size on the horizontal axis and execution count on the vertical axis. Repeat for swaps. Choose scales so that the full range of values can be displayed and label the lines so you know which one is which. You may find that using a log-log plot shows the values over the full range better.

Level 4: Improving quicksort

A simple implementation of quicksort is likely to perform badly for certain datasets. For example, the implementation used as an example in lectures performs very poorly if the dataset is in reverse-sorted order.

The poor performance can be avoided by choosing a better pivot for the partitioning step. Common strategies include selecting a random element or using the median of the first, middle, and last elements. In any case, the chosen pivot is swapped with the last element before proceeding to do the partition as before.

Read about and implement one of the improvements to quicksort and show (by adding its profile data to the graphs of Level 3) how much improvement you have achieved. Include a brief description of the method you implemented to improve quicksort's performance.