COMP20003: Algorithms and Data Structures
Week 4: Workshop Exercise: Sorting Algorithms

Part A: Elementary sorting algorithms

Objectives

- To implement elementary sorting algorithms in C.
- To revise the use of structures and arrays in C.
- To gain practice in understanding and modifying existing code.

Preparation

- Read this lab sheet.
- Read Week 3 slides on sorting and refer to any basic textbook for algorithms.
- Read through the provided source code.
- Read up on arrays and structures in a C reference book.

Source Code

Download the files from LMS.

The above directory also contains the source code of a program called
``data_gen``. This program may be used to generate random test data.
The source code is in the file ``data_gen.c``.

Part A.1: Complete the ``SORT1`` program

The program ``SORT1`` reads a sequence of whitespace separated integers
from ``stdin`` and sorts them into ascending numerical order.
It reads in a maximum of 5000 integers.
You may supply more but the program will ignore them.

``SORT1`` allows you to sort the input using one of three algorithms:
bubble sort, selection sort, insertion sort.

The program requires one command line argument: the name of the sorting algorithm it should use.
These are ``bubble`` for bubble sort, ``selection`` for selection sort,
``insertion`` for insertion sort.

Invoking the program as follows will cause ``sorts`` to read its
data from the file ``data`` and sort it using bubble sort.::

```
$ ./SORT1 bubble < data
```

As it reads integers, ``sorts``, stores them in a array of ``Record`` structures.
Each ``Record`` structure has two integer fields.
The integer that is read from stdin is stored in the **key** field.

Additionally, as each integer is read in, the program stores a second integer in the **value** field. The **value** field is used to record the order in which integers in the **key** field were read in.

For example, if the sequence of integers that is read in is::

```
        23 19 17 13 11 7 5 3 2
```

the records will look like this:::

```
  23    0
  19    1
  17    2
  13    3
  11    4
  7     5
  3     6
  2     7
```

After each sort run, ``SORT1`` prints out the sorted records (both fields), the number of comparisons that the sort run required, and information regarding whether the sorting run was successful. If the sort was successful it also checks if the sort run was stable.
Note that the fact that some of the sort runs are stable does not necessarily mean that the sorting algorithm is stable in general.

The reason for having the **value** field in each records is so that we can test for stability, by checking that records have the same key have their **value** fields sorted in ascending numerical order after the sort run.

Currently, only bubble sort has been implemented. For the first part of this exercise you need to implement the other two sorting algorithms.

You must use the function ``compare_records`` to compare records; its interface is described in the source code.
This function has been instrumented so that it keeps track of the number of comparisons that have been performed.
You may also want to make use of the ``swap_records`` function.
The ``bubble_sort`` function provides an example of the use of both these functions.

Required:

You need to modify the code in basicsorts.c

Laboratory Tasks 1: Implement selection sort
Implement selection sort and complete the function ``selection_sort``.

Laboratory Tasks 2: Implement insertion sort
Implement insertion sort and complete the function ``insertion_sort``.

Pseudocodes for the functions are given in lecture slides.
Please refer to any basic C textbook or AIA for more explanation.

Optional:

Part A2: Experimental Work

After implementing the above algorithms you should compare the number of comparisons required for each of the four algorithms on different kinds. (In practice, we may also be interested in the number of exchanges or swaps that each algorithm carries out.)

In addition random data, other kinds of data that you should consider might include:

- data that is already sorted
- data that is reverse sorted
- data where every element is identical
- data that consists of only two distinct values

You will need to create your own test data. We have provided a small program, data_gen, that you can use to generate pseudo-random data. The source code is in the file data_gen.c.

The Unix sort command may be useful in generating certain types of test data. To learn how to use it, read its man page.

You should test different sizes of each kind of data.
(Don't forget to consider the case where the file is empty.)
Prepare a table showing the results of your experimentation.
Show this to your lab demonstrator when you are done.

Questions

Finally, answer the following questions:

1) How do your experimental results compare to the known theoretical efficiencies of the algorithms?

2) Is selection sort stable? Create the smallest test case that demonstrates that it is not stable.

3) What sort of data does insertion sort work best on? What sort of data does it work worst on? In both cases, describe the relationship between the number of comparisons and the size of the input.

Part B: Advanced sorting algorithms

Objectives

- To implement two advanced sorting algorithms in C.

Preparation

- Read the sections of the textbook on mergesort and quicksort (Sections 2.3 and 2.4).
- Review the code that you wrote for PartA.

Introduction

In this workshop we will be create program ``SORT2`` by completing necessary code in ``adv_main.c``.
For this exercise you will be implementing the quicksort and mergesort algorithms.

Task: Complete the ``adv_main.c`` program

The program ``SORT2`` works in a similar manner to the ``SORT1`` program.
It reads a sequence of whitespace separated
integers from ``stdin`` and sorts them in ascending numerical order.

``adv_sorts`` will read in a maximum of 5,000,000 integers. You may supply more but the program will ignore them. Each record contains a second integer that is used to implement stability testing - the same mechanism employed by the ``sorts`` program. (The **Record** structure used here is the same as before.)

``adv_sorts`` employs one of three sorting algorithms: quicksort, mergesort

and also the C standard library's ``qsort`` function.

The program requires one command line argument. This argument specifies the sorting algorithm that is to be used. These are ``quicksort`` for quicksort, ``mergesort`` for mergesort, and ``libsort`` for the standard library's ``qsort`` function.

Currently, only the ``libsort`` function has been implemented.

Required :

For this lab you need to complete the function ``basic_quicksort``.
You should follow the pseudocode provided in the textbook.

Optional :

If you are finished, you may continue and complete the ``basic_mergesort``.

Also Optional :

Possible Improvements

The versions of the algorithms that you have implemented in this exercise will serve as a baselines against which we can measure potential improvements. In preparation for a later lab on sorting, please think about the following issues:

- How would you measure the performance of these algorithms? For example, you may wish to instrument the code to count comparisons as we did in the exercise of Part 1, or you may wish to use the actual CPU time.

- What test data sets would you want to use to measure the performance of the algorithms?

Discuss your ideas with your tutor.