COMP26120 Lab Exercise 8 Lexicographic Sorting

Duration: 1 lab session

For this assignment, **you may use code taken from other sources** (but remember to use comments to fully credit sources of information in your code, and to understand it so that you can explain it and to check that it works correctly).

Aims

To gain experience from implementing an efficient sorting algorithm. To gain understanding of the concept of stability in sorting. To use the concept of stability to achieve a lexicographic ordering of a set of multidimensional objects.

Learning Outcomes

On successful completion of this exercise, a student will:

1. Have implemented an efficient and **stable** sorting algorithm
2. Have implemented the algorithm so that it works on structured objects, which can be sorted on various different keys
3. Have written a program that achieves a lexicographic sorting of objects, by sorting them in multiple passes (first by one key, then by another)
4. Have sorted a list of books lexicographically by price, user-rating and relevance
5. Have a further appreciation of properties of different sorting methods
6. Have used qsort and the Unix sort

Summary

The task is to write a program that can sort a set of complex items, first by one selected attribute (or key) and then by another.

1. You will write a sorting function that can operate on an array of structs, and can sort by any of the member variables of the struct.
2. The sort function must be **stable**. You will test this.
3. You will use your sorting function in a program to achieve a lexicographic sorting of an array of structs representing books from an online book store.
4. You will compare your method to one we provide which uses quicksort.
5. You will also compare the result of your method to the inbuilt qsort function, and to the Unix sort facility

Deadlines: The unextended deadline is the end of your scheduled lab. If you need it, if you attend the lab you will get an automatic extension to 5pm on the last day of the semester (you must use submit to prove you finished in time and get it marked at the start of your next scheduled lab).

Description

Amazon, the online bookstore, provides users with various tools to help browse or search its collection of books. A search by keyword or category is the basic tool, but results will frequently span many pages. To further aid browsing, the user may subsequently choose to sort the books (the "hits") in different ways: by star-ratings; by sales-rank; by relevance to the search terms initially entered; or by price; etc.

In this lab, you will implement a sorting algorithm that operates on an array of structs representing books. The variables in the struct represent the different attributes of the book that a user may like to sort on.

Going beyond what is currently possible on Amazon, your program will allow a user to sort on *several* keys, not just one. The user will be asked what is the most important key, the second most, and so on. (Initially we will use up to three keys only).

The desired effect of sorting on several keys is to achieve a *lexicographic* ordering of the books. What this means is that items will be sorted in order of the most important key, but in the case of ties, the tied items will be sorted in order of the next most important key, and so on.

Here is an example of a lexicographically ordered list:

Star-rating	Relevance	Price
5	8	7.99
5	8	22.49
5	7	6.49
5	6	18.99
4.5	9	12.99
4.5	7	11.49
4.5	2	7.49
4	9	9.99
4	4	6.99

The most important key was the Star-rating, then the Relevance, then the Price (with the price being sorted from low to high).

One (convenient) way to achieve a lexicographic ordering of a set of items is to sort them first by the **least** important key, then re-sort them by the next least important key, and so on, finally sorting them by the most important key. This will work reliably **if and only if** the sorting method is **stable**.

What does it mean for a sorting algorithm to be stable? (See Goodrich and Tamassia, pages 242-243). It means that if two items to be sorted have an equal key, then in the sorted array they will be **in the same order** as they were in the original unsorted array.

At this point, you should now check that you understand why (i) we need a stable sort method to achieve lexicographic ordering by doing multiple sorting passes, and (ii) why the passes are done in the reverse order of the importance of the sorting key.

Part 1: Write the Sorting Function

Useful files will be found in /opt/info/courses/COMP26120/problems/ex8.

You will need an efficient sorting algorithm that is also stable. We want you to use a comparison-based algorithm (i.e. not a bucket sort) because the attributes to sort on could include strings or other keys that are not easy to place in a bucket array.

By efficient, we mean that on most inputs it will have complexity O(n log n) or better, so e.g. bubblesort is not allowed.

Having decided on an appropriate sort algorithm, write a function to go in the file, **part1.c**.

You must write your own function, not use a built-in one (such as qsort), but you are allowed to adapt code from the Internet or a book, if you like. However, if you do that, you MUST clearly indicate the source of the code in your comments.

part1.c already provides the struct for the array of items, a function to read in a list of items, and some comparison functions to compare items by rating, relevance, and price.

Your sort function should include a comparison function as an argument, as follows:
```
void mysort(..., ..., ..., int(*compar)(const void *, const void *))
```
to take advantage of the comparison functions provided. (This is the same way the qsort function is defined).

Test the function by sorting shortbooks.txt by price. To do this, call the function from the `user_interface(int N)` function. Check that your sorting algorithm is stable by inspecting the ID column. The IDs should remain in order amongst each group of prices that are tied (equal).

Note: the print_results() function assumes that "relevance" and "rating" are sorted in ascending order, and "price" descending. It then prints the results in the reverse of this order to obtain the typical search engine format, where the best results are at the top. Only the top 20 results are shown.

Part 2: Find the Book Anna Wants

Copy your code from part1.c to part2.c

Now re-write the function, `user_interface(int N)`, so that is asks the user if they would like to do a lexicographic sort, and then asks them to provide their most important field (or key), next most important, up to three fields. (You may do this in any sensible way). The function should then call your sort function as appropriate.

Anna would like to buy a book on (or by) PG Wodehouse for her brother. She will buy the least expensive book from amongst those with the highest relevance to the search term, "Wodehouse", from amongst those with the highest star-rating. (In other words, star rating is the most important and price the least important sort key).

Use your program on the file Wodehouse.txt. **Place the top 20 results in a file: top20.txt.** Which book does Anna buy (what is its ID number?) ?

In the event, Anna does not buy the book at the top of the list, but instead buys the cheapest book with a star rating of 5 and relevance above 9.5. Which book is this?

Part 3: Quicksort, qsort and Unix sort

Quicksort

So far you have done parts 1 and 2 using your own implementation of an efficient sorting algorithm. Now we are going to run some code that uses its own implementation of quicksort.

The executable of this program is "books-quick". Run books-quick on the Wodehouse file
```
./books-quick 1000 Wodehouse.txt
```
which will (automatically) repeat the lexicographic sorting for Anna.

Did you obtain the same result as with your method? Explain.

The qsort() stdandard library function

Copy your code from part2.c to **part3.c**. Comment out the calls to your sort function and replace them with calls to qsort(). Then re-run the sort for Anna on Wodehouse.txt again. What result do you get? What do you infer about qsort() ?

The Unix sort function

It is possible to achieve the same lexicographic sort of Wodehouse.txt using the command line sort function. Here is a faulty attempt:

```
sort -nk 2,2 -s Wodehouse.txt | sort -k 3,3 | sort -n | head -20
```

Look at the manual pages for the command line sort function and correct the command line. Then use the output to check your result from Part 1 above. (You can used "diff" to do this). Place your corrected command in a Unix shell script called **unixsort.sh** and make it executable by all.

Part 4: Bonus Part

Copy part2.c to **part4.c**. Then,

Either:

Expand the struct storing infomation about the books. It could include sales rank, author names, titles. Add a comparison function for each of these member variables.

Add appropriate columns to a section (a few lines) of Wodehouse.txt, update the read_file() function, and try out your program.

Or:

Add any other interesting functionality to help a user sort the books in Wodehouse.txt

Marking Process

You should have in your directory COMP26120/ex8 the following files: `part1.c part2.c part3.c part4.c top20.txt unixsort.sh`. Both labprint and submit will look for these six files.

The marks are awarded as follows:

```
Part 1:
3 marks: implementation of an efficient, stable sort function
1 mark: working correctly on shortbooks.txt
Part 2:
2 marks: implementation of lexicographic sort interface
1/2 mark: correct answer to Anna's book search
1/2 mark: correct answer to Anna's book search (second part)
Part 3:
1 mark: qsort() implementation and understanding of result
1 mark: Unix command line sort correct
Part 4:
1 mark: additional sorting terms added
Total 10
```