CSc 560 — Database Systems Implementation
Fall 2022 (McCann)

http://u.arizona.edu/~mccann/classes/560

Program #2: External Multi–Way Merge Sort

(100 points)

Due Date: October 6th, 2022, *at the beginning of class*

Overview: We've covered indices, and learned that both primary and clustered ordered indices require data, in both the index and the database, to be in sorted order. We're about to talk about how relational DBMSes process relational and set operators, and will learn that some of those algorithms rely on sorted data. We can maintain data in sorted order, and can access data in sorted order via an index, but we can also incur the cost of sorting the data. A major problem: Often, the data is much larger than the available memory, making common in–memory sorting algorithms unsuitable. The solution: An external sorting algorithm, which, as the name suggests, holds the data on disk much of the time. Our text describes a common external sorting algorithm. In this assignment, you'll implement it, with a few modifications.

Assignment: Using your choice of programming language, write a complete, well–documented program that uses a version of the "external sort–merge" algorithm presented in our text in Section 15.4.1 (pages 701 and 702) to sort CSV files on any attribute selected by the user.

Here's our version of that algorithm. We will be using pseudo–blocks of data; the user will provide the first stage's (Pass 0's) output run–length (R) (that is, the quantity of CSV file lines to be sorted by each call to an internal sorting algorithm), as well as the default quantity of runs to be merged (W) (the 'way' of a W–way merging of sorted runs) in the second stage (Passes 1 — n). We'll re–use R in the second stage as our 'block' size for the number of lines of CSV data to read at a time.

```
Pass 0:         r <-- 0
                Loop:
                    Read R lines from the provided CSV file
                    Sort the lines using the provided field
                    Write out the sorted lines to file "run0[r].csv"
                    Increment r
                Until the provided CSV file is exhausted

Passes 1 - ?:   p <-- 1
                While the number of sorted runs from pass p-1 is > 1:
                    r <-- 0
                    While there are still unread runs from pass p-1:
                        Loop:
                            Read R lines from each of the next W run[p-1][?].csv files
                            Merge the content of those blocks into run[p][r].csv,
                              refilling the R lines from a run as the previous R lines
                              are exhausted
                        Until the longest of those W run[p-1][?].csv files is exhausted
                        Increment r
                    End while
                    Increment p
                End while

Clean-Up:       Make a copy of the completed file under the name "...-sorted.csv"
                Verify that the completed file's content is sorted
```

(Continued...)

A few notes on the algorithm:

1. As in Program #1, we will assume that all fields are strings, meaning that the lines are to be sorted using string comparisons.
2. A clarification of the naming scheme of the files of runs is definitely needed: The naming scheme for those run files is `run[p][r].csv`, where `p` is the pass number and `r` is the run number within that pass, without the brackets. Thus, the sorted runs created by pass 0 are named `run00.csv`, `run01.csv`, etc. In pass 1, we start by merging the content of the first W runs from pass 0 into the file `run10.csv`, the next W runs into `run11.csv`, etc.
3. The sorting of the runs in Pass 0 can be accomplished with any internal sorting algorithm/routine you wish to use, including those provided with your language.
4. The question marks are there only to suggest that the exact values at those positions are hard to describe in pseudocode, but will be easy to figure out as you play around with small examples and think about what needs to be accomplished.
5. The name of the copy of the sorted file is to be the same filename as the given file, but with the string "-sorted" appended. For example, if the input file is named "chips.csv", the sorted file is to be named "chips-sorted.csv".

Speaking of small examples, here's one to help give you a mental image of what you'll need to accomplish. Here, R = 3 and W = 3. The details of loading, reading, and reloading the 'blocks' in Passes 1 and 2 are not shown, and the lines of hyphens are included only for clarity; they are not to be stored in any files.

```
   (a)          (b)                  (c)                  (d)

                After                After                After
Given Data      Pass 0:              Pass 1:              Pass 2:
==========      =======              =======              =======
                 -----                -----                -----
   L,8           A,9                  A,9                  A,9
   A,9           L,8    run00.csv     E,4                  C,7
   Q,5           Q,5                  G,5                  D,2
   R,1           -----                H,0                  E,4
   H,0           G,5                  L,8    run10.csv     G,5
   G,5           H,0    run01.csv     M,6                  H,0
   M,6           R,1                  Q,5                  L,8    run20.csv
   E,4           -----                R,1                  M,6
   T,1           E,4                  T,1                  Q,5
   C,7           M,6    run02.csv     -----                R,1
   D,2           T,1                  C,7                  T,1
   V,0           -----                D,2    run11.csv     V,0
   Z,6           C,7                  V,0                  Z,6
                 D,2    run03.csv     Z,6                  -----
                 V,0                  -----
                 -----
                 Z,6    run04.csv
                 -----
```

Figure (a) shows the CSV file records to be sorted on the first field. Pass 0 reads groups of R records (R CSV file lines) into memory, one group at a time, and sorts them into runs (Figure (b)). Notice that the final run may be shorter than the others. Pass 1 does a three-way (W–way) merge of the first three runs from Pass 0. As Figure (c) suggests, run11.csv is the result of a two–way merge because there are only two runs left to be merged. Pass 2 is the final pass because only one run is produced, as shown in Figure (d).

(Continued...)

Data: Write your program to accept the following command–line arguments, in this order:

1. The complete pathname of the CSV file whose lines are being sorted
2. The # of the field on which the file is to be sorted (where the first field is field 1)
3. R, the Pass 0 run–length (# of records to be sorted per group)
4. W, the "way" for Passes 1 – n (maximum initial # of runs being merged at a time)

For example, if the `chips.csv` file from Program #1 is being sorted on the number of transistors, sorting five lines per group in Pass 0, and using a two–way merge, the command line would be (assuming Python3, just to pick something):

```
$ python3 /home/cs560/fall22/chips.csv 4 5 2
```

To keep things a bit simpler, we will again assume that all field values are ASCII strings. When a field's value is missing, assume that its value is the empty string (e.g., `""`). As you did in Program #1, assume that the first line of any provided CSV file is metadata — do not sort it.

Output: Apart from making the copy of the completed file as discussed above, end your program by displaying the total number of runs that were created by each pass, the name of that completed file, the quantity of lines in the completed file, and a message affirming that the completed file is, in fact, sorted on the requested field. That is, something like this:

```
Pass 0 created 5 runs.
Pass 1 created 2 runs.
Pass 2 created 1 run.
The completed file's name is "chips-sorted.csv".
It contains 4854 lines.
VERIFIED: The completed file is in sorted order by field 4.
```

Do not just print the verification message – you need to write code to read and test the ordering of the lines of the completed file.

Hand In: You are required to submit your completed program file(s) using the `turnin` facility on lectura. The submission folder is `cs560p2`. Instructions are available from the document of submission instructions linked to the class web page. In particular, because I will be grading your program on lectura, it needs to run on lectura, so be sure to test it on lectura. Feel free to split up your code over additional files if doing so is appropriate to achieve good code modularity. Submit all files as–is, *without* packaging them into `.zip`, `.jar`, `.tar`, etc., files.

Other Requirements and Hints:

- I strongly recommend that begin this assignment by working through the algorithm by hand (yup, pencil and paper!) on a larger example than the one provided here, so that you understand how to handle the refilling of our 'blocks' that is required during the merging of longer runs. It's hard to write code to do something that you don't know how to do yourself!
- The basics from Program #1 apply here, too: You may use any language you wish to implement your program, so long as it is installed on lectura; you can consult resources that explain how to use your language but not resources that supply solutions to all or parts of the assignment; sharing output with your classmates is encouraged; test your code thoroughly on multiple inputs; document well; late days can be used if you have some remaining; and, as always, **Start early!**