CSC 225 - SUMMER 2018
ALGORITHMS AND DATA STRUCTURES I
PROGRAMMING ASSIGNMENT 1
UNIVERSITY OF VICTORIA

Due: Sunday, June 17th, 2018 before 11:55pm. **Late assignments will not be accepted.**

1 Assignment Overview

This programming assignment covers an application of algorithms and data structures arising in the implementation of database systems. Databases and data management are covered by later courses, such as CSC 370. In this assignment, you will design and implement an efficient and correct algorithm for **aggregation**, which is a fundamental operation in relational database systems. Since this is not a database course, the algorithm will operate on simple text-based spreadsheet files instead of a database.

The specification for this assignment can be found in Section 2. Sections 1.1 - 1.3 contain a description and several examples of grouping and aggregation, using the following (fictional) table of student numbers and grades.

Table `grades`

id	course_subject	course_num	grade
V00123457	CSC	225	78
V00654322	MATH	110	63
V00654322	CSC	110	75
V00314159	CSC	106	85
V00951413	CSC	106	72
V00951413	MATH	110	68
V00654322	CSC	106	70
V00123456	CSC	110	79
V00314159	CSC	110	47
V00654322	CSC	225	91
V00123456	CSC	106	68
V00123456	CSC	225	89
V00951413	CSC	225	40
V00314159	MATH	100	74
V00654322	MATH	100	71

1.1 Counts and Averages by Student

Consider the task of computing the number of courses taken by each student, or the task of computing the average grade for each student. For both of these tasks, the rows of the table need to be

divided into groups by student number (which is the `id` column of the table). Then, the number of courses per student or average grade per student can be computed by counting or averaging the `grade` values within each group. Applying a function (such as counting or averaging) to collapse a group of rows into a single row is called **aggregation**. The table on the left below shows the groupings of rows by the `id` column, with each group receiving a different colour, and the tables on the right show the counts and averages within each group. Notice that the only columns in the result tables are the grouping criteria (in this case, the `id` column) and the aggregation result. All other columns in the original table are omitted.

Table `grades`

id	course_subject	course_num	grade
V00123457	CSC	225	78
V00654322	MATH	110	63
V00654322	CSC	110	75
V00314159	CSC	106	85
V00951413	CSC	106	72
V00951413	MATH	110	68
V00654322	CSC	106	70
V00123456	CSC	110	79
V00314159	CSC	110	47
V00654322	CSC	225	91
V00123456	CSC	106	68
V00123456	CSC	225	89
V00951413	CSC	225	40
V00314159	MATH	100	74
V00654322	MATH	100	71

Original table, grouped by `id` column.

Aggregation (Counting Rows)

id	count(grade)
V00123457	1
V00654322	5
V00314159	3
V00951413	3
V00123456	3

Result of aggregation by counting the number of `grade` values in each group.

Aggregation (Averaging)

id	avg(grade)
V00123457	78
V00654322	74
V00314159	68.66
V00951413	60
V00123456	78.66

Result of aggregation by averaging the `grade` values in each group.

1.2 Averages by Subject

Consider the task of computing the average grade within each subject (`CSC` and `MATH`). In this case, the grouping criteria is the `course_subject` column and the aggregation result is `avg(grade)`. The tables below show the grouping and aggregation. As above, notice that the other columns are excluded from the result table.

Table `grades`			
id	**course_subject**	**course_num**	**grade**
V00123457	CSC	225	78
V00654322	MATH	110	63
V00654322	CSC	110	75
V00314159	CSC	106	85
V00951413	CSC	106	72
V00951413	MATH	110	68
V00654322	CSC	106	70
V00123456	CSC	110	79
V00314159	CSC	110	47
V00654322	CSC	225	91
V00123456	CSC	106	68
V00123456	CSC	225	89
V00951413	CSC	225	40
V00314159	MATH	100	74
V00654322	MATH	100	71

Original table, grouped by the `course_subject` column.

Aggregation Result	
course_subject	**avg(grade)**
CSC	72.18
MATH	69

Result of aggregation by averaging the `grade` values in each group.

1.3 Averages by Course

Now suppose we want to compute the average grade for each course. It is not sufficient to group by the `course_num` column, since two courses with the same number may not be the same course (for example, consider CSC 110 and MATH 110). The grouping criteria therefore must include two columns: `course_subject` and `course_num`. The tables below show the grouping and aggregation. Again, the only columns in the output table are the grouping criteria and the aggregation result.

Table `grades`			
id	**course_subject**	**course_num**	**grade**
V00123457	CSC	225	78
V00654322	MATH	110	63
V00654322	CSC	110	75
V00314159	CSC	106	85
V00951413	CSC	106	72
V00951413	MATH	110	68
V00654322	CSC	106	70
V00123456	CSC	110	79
V00314159	CSC	110	47
V00654322	CSC	225	91
V00123456	CSC	106	68
V00123456	CSC	225	89
V00951413	CSC	225	40
V00314159	MATH	100	74
V00654322	MATH	100	71

Original table, grouped by the `course_subject` and `course_num` columns.

Aggregation Result		
course_subject	**course_num**	**avg(grade)**
CSC	225	74.50
MATH	110	65.50
CSC	110	67
CSC	106	73.75
MATH	100	72.50

Result of aggregation by averaging the `grade` values in each group.

1.4 Student Counts by Subject

Finally, consider the problem of computing the number of different students enrolled in each subject (CSC or MATH). The result should have one row for each subject code, along with the number of different students enrolled in any course for that subject. A particular student might be enrolled in multiple CSC courses, but that student should only be counted once toward the CSC subject code. Therefore, the `count_distinct` aggregation function should be used instead of the regular `count` function. The `count_distinct` function returns the number of different values of the aggregation column within each group. For example, if the `course_subject` column is used as the grouping criteria and the `id` column is used as the aggregation column, the group for the 'MATH' subject code would contain the `id` values

V00654322, V00951413, V00314159, V00654322

The `count` aggregation function would return the number of items in the group (4), but this is not the number of students taking MATH courses (since one `id` value appears more than once). The `count_distinct` function returns the number of distinct values in the group (3).

Table `grades`			
id	**course_subject**	**course_num**	**grade**
V00123457	CSC	225	78
V00654322	MATH	110	63
V00654322	CSC	110	75
V00314159	CSC	106	85
V00951413	CSC	106	72
V00951413	MATH	110	68
V00654322	CSC	106	70
V00123456	CSC	110	79
V00314159	CSC	110	47
V00654322	CSC	225	91
V00123456	CSC	106	68
V00123456	CSC	225	89
V00951413	CSC	225	40
V00314159	MATH	100	74
V00654322	MATH	100	71

Original table, grouped by the `course_subject` column.

Aggregation Result	
course_subject	**count_distinct(id)**
CSC	5
MATH	3

Result of aggregation by counting the number of distinct `id` values in each group.

2 Specification

Your task for this assignment is to write a Java program `Aggregate` which reads a table from a text file and performs grouping and aggregation with one of four aggregation functions: `count` (count the number of rows in each group, including rows which happen to be duplicates of over rows in the group), `sum` (add up all elements in the aggregation column within each group), `avg` (compute the average of all elements in the aggregation column within each group) and `count_distinct` (count the number of rows in each group, with duplicates ignored). To receive full marks, the entire implementation must be correct, allow any number of columns to be used as grouping criteria, and run in $O(n \log_2 n)$ time in the worst case on any input table with n rows. See Section 3 for more details on the evaluation process.

2.1 Data Format

Tabular data can be stored in a variety of ways. One simple format for text-based tables and spreadsheets is the comma-separated value (CSV) format. Files in CSV format normally have the extension `.csv` or `.txt`. A CSV spreadsheet consists of a line of text for each row of data, with each column separated by a comma. In this assignment, column headings will be given as the first row of the spreadsheet, and no comma will appear after the last column on each line.

The `grades` table in Section 1 would be represented in CSV format as follows.

```
id,course_subject,course_num,grade
V00123457,CSC,225,78
V00654322,MATH,110,63
```

```
V00654322,CSC,110,75
V00314159,CSC,106,85
V00951413,CSC,106,72
V00951413,MATH,110,68
V00654322,CSC,106,70
V00123456,CSC,110,79
V00314159,CSC,110,47
V00654322,CSC,225,91
V00123456,CSC,106,68
V00123456,CSC,225,89
V00951413,CSC,225,40
V00314159,MATH,100,74
V00654322,MATH,100,71
```

Formally, the CSV files used in this assignment will conform to the following specification.

- Every CSV file must contain at least one non-blank line, which will contain the column headings.
- Blank lines (consisting entirely of whitespace characters such as spaces and tabs) will be completely ignored (and will not count as a row of the table).
- You may assume that every line contains the same number of columns (an input file with an inconsistent number of columns per line will be considered invalid).
- You may assume that the length of each line is at most 100000 characters (and therefore that the number of columns is at most 100001).
- There is no limit to the number of rows which may appear in the file.
- The data in the file may contain letters, numbers, spaces and punctuation, with one exception: comma characters will only appear as column separators (they may not appear in the data).
- The number of columns is equal to the number of commas in the line plus one. The data in a column may be blank. For example, the line '`Hello,,World,`' contains four columns. The second and fourth columns are blank.

2.2 `Aggregate` program interface

Your `Aggregate` program will accept command line arguments in the form

```
$ java Aggregate <function> <aggregation column> <input file> <group column 1> <group column 2> ...
```

where

- `<function>` is one of '`count`', '`sum`', '`avg`' or '`count_distinct`',
- `<aggregation column>` is the column name of the column to aggregate,
- `<input file>` is the name of the input CSV file, and
- the `<group column x>` arguments are column names of grouping columns (at least one must be specified, but in a complete implementation, any number of grouping columns should be allowed).

For example, if the table from Section 1 is contained in the file `table_of_grades.csv`, the number of grades per student (from Section 1.1) would be computed with the command

```
$ java Aggregate count grade table_of_grades.csv id
```

and the average grade in each course (from Section 1.3) would be computed with the command

```
$ java Aggregate avg grade table_of_grades.csv course_subject course_num
```

The program will report an error if any of the following conditions arise.

- The `<function>` argument is not one of '`count`', '`sum`', '`avg`' or '`count_distinct`'.
- The input file does not exist, cannot be opened or is not in the correct format.
- The aggregation column or any of the grouping columns do not exist in the input file.
- A column is used as both a grouping column and an aggregation column.
- When the aggregation function is `sum` or `avg`, an error will be reported if the aggregation column contains any non-numerical data (except the column header). Other columns may contain non-numerical data. When the aggregation function is `count` or `count_distinct`, the aggregation column may contain any data (and its contents should be treated as `String` values).

If none of the above errors occur, the program will perform the aggregation and output the resulting table to standard output (that is, the console) in the CSV format described in the previous section, including column headers. The **only** columns that will appear in the output table are the grouping columns and the aggregation result. The grouping columns must appear first, with the last column containing the aggregation result. The column name of the aggregation result column should contain the function name (`count`, `sum` or `avg`) and the name of the original column (e.g. `count(grade)` or `avg(temperature)`). The rows of the result table may appear in any order, as long as the header row is first (so you do not have to ensure that the order matches the examples in this document). For example, with `table_of_grades.csv` defined as in the examples above, the output of the command

```
$ java Aggregate avg grade table_of_grades.csv course_subject course_num
```

would contain all of the rows below (with the header row first, but with the other rows possibly in a different order).

```
course_subject,course_num,avg(grade)
CSC,106,73.75
CSC,110,67.0
CSC,225,74.5
MATH,100,72.5
MATH,110,65.5
```

2.3 Running Time Requirements

You will be expected to justify the running time of your implementation (as a whole) in terms of the number of rows n in the input file. For full marks (see Section 3), your implementation must have a running time in $O(n \log_2 n)$ (and you must be able to justify this fact during your in-person demo). You may assume that the input and output functionality provided by the standard library is linear in the number of characters read or written (that is, reading k characters from a file can be assumed to take $O(k)$ time). If your implementation uses any other features of the standard library (including sorting algorithms or data structures like lists), you must be prepared to justify the running time of any operations used, so you should consult the official Java documentation to verify that the running times of each operation are suitable for the task.

As noted in Section 2.1, you may assume that the maximum length of any line in the input file is 100000 characters. This assumption ensures that the number of columns will always be bounded by

a constant value, so the number of columns in the input file will have no effect on the asymptotic running time of your code (which will simplify your analysis). Another consequence of the 100000 character limit is that you may assume that any operation performed on a fixed number of rows (e.g. comparing two rows, or splitting a single row into columns) will have a constant running time. Therefore, you should focus your algorithm design and analysis on the overall grouping and aggregation of rows (since the number of rows is not bounded by a constant), rather than the internal operations needed to process individual rows.

2.4 Using Outside Code

You are encouraged to use the features of the Java Standard Library (including any of the data structures it provides) in your code. If you use a standard library data structure, make sure you are aware of the running times of the operations you use, since that will be important for determining the running time of your program.

There should be no need to use large volumes of code from other sources (such as outside libraries or the internet) in this assignment. If you believe that your implementation requires an outside library, talk to your instructor.

If you find a small snippet of code on the internet that you want to use (for example, in a StackOverflow thread), put a comment in your code indicating the source (including a complete URL). Remember that using code from an outside source without citation is plagiarism.

You are encouraged to discuss the assignment with your peers, and even to share implementation advice or algorithm ideas. However, you are not permitted to use any code from another CSC 225 student under any circumstances, nor are you permitted to share your code in any way with any other student (or the internet) until after the marking is complete. Sharing your code with others before marking is completed, or using another student's code for assistance (even if you do not directly copy it) is plagiarism.

2.5 Implementation Advice

Although this assignment may seem daunting, most of the complexity lies in the specification (since this application is likely new to you) and in the finishing touches needed to make the algorithm asymptotically fast. The actual volume of code needed may be significantly less than assignments you have completed in the past. Your instructor's solution required fewer than 200 lines of Java code.

You are encouraged to try designing a solution from scratch, since the design process is often the most difficult part of a software project. However, if you are stuck, or feeling uninspired, consider implementing your solution in the following steps.

1. Write a simple program to read a CSV file into an array or list structure (possibly a multidimensional array or list), then output the data in CSV format. Test this thoroughly to ensure that it is correct. To make things easier later, use separate methods for the input and output code. You may use outside libraries to read/write CSV data, but it is likely easier to write this component yourself.

2. Write a method which takes a table and a set of column names as input, then produces a new table containing only the provided columns (for example, given the `grades` table used in previous examples and the column names '`id`' and '`grade`', the method would make a new table containing only the '`id`' and '`grade`' columns). Once this feature has been written, verify that your code can read an input file and prune away all columns besides the group columns and aggregation column, then print the resulting table (without any aggregation being performed). Note that you will receive some marks if you can reach this point (see Section 3).
3. Write a simple (but possibly slow) aggregation algorithm and verify that it works. Various algorithm ideas will be covered in lectures and labs. It is far more important to have working code than fast code (if your code is not correct, you will receive no marks for its performance). You are advised to focus on the `count`, `sum` and `avg` aggregate functions at first (and handle the more difficult `count_distinct` function later).
4. Once you have a working implementation (even if it is slow), make sure you can analyse and explain its running time. Add comments and other documentation if necessary to clarify complicated parts of the code. Hand in your working implementation.
5. If your code does not have a worst case running time of $O(n \log_2 n)$, try implementing an improved algorithm. Again, it is better to submit a slow but correct implementation than a fast but incorrect one. Additionally, a fast implementation will not receive full marks unless you can justify its running time.

Note: You should not use a hash table (or any hashing-based Java data structures, like `HashMap`) for your implementation if you want your code to have a **worst case** $O(n \log_2 n)$ running time. You will not receive marks for the $O(n \log_2 n)$ running time if you use hashing-based structures and are unable to demonstrate that their worst case (**not** expected case) running time is $O(n \log_2 n)$. Additionally, you may lose marks for your running time justification if you mischaracterize the running time of a hashing data structure. Although hash tables are very useful in most cases, they have a relatively poor worst-case running time.

3 Evaluation

Submit all `.java` files needed to compile your assignment electronically via conneX. Your code must compile and run correctly in the Linux environment in ECS 242. If your code does not compile as submitted, you will receive a mark of zero.

This assignment is worth 10% of your final grade and will be marked out of 20 during an interactive demo with an instructor. You will be expected to explain the running time of your implementation to the evaluator, and may also be asked to explain or justify some aspects of your code. Demos must be scheduled in advance (through an electronic system available on conneX, which will be provided shortly before the due date). If you do not schedule a demo time, or if you do not attend your scheduled demo, you will receive a mark of zero.

The marks are distributed among the components of the assignment as follows.

Marks	Component
5	The input file is read successfully and an output file containing the correct columns is generated (even if no aggregation is performed).
5	Aggregation is correct when a single grouping column is used.
4	Aggregation is correct for an arbitrary number of grouping columns.
3	You are able to explain and justify the running time of your code to the evaluator. Note that the running time does not have to be optimal for you to receive these marks (as long as your explanation is clear and correct). If you use algorithms or data structures from the Java standard library, you will be expected to know their running times.
3	The worst case running time of the entire program is $O(n \log_2 n)$ for an input with n rows. These marks will only be given if the implementation is correct for multiple grouping columns and if you can justify the running time.

If your code is not well organized, or if it is poorly documented, the evaluator may ask you explain any aspects that are unclear. If you are unable to do so, up to 4 marks may be deducted. To be clear, you are not required to have spotless, perfectly organized code, but you should be prepared to explain any hard-to-read parts of your code.

You are permitted to delete and resubmit your assignment as many times as you want before the due date, but no submissions or resubmissions will be accepted after the due date has passed. You will receive a mark of zero if you have not officially submitted your assignment (and received a confirmation email) before the due date.

Your main program must be contained in a file called `Aggregate.java`. You may use additional files if needed by your solution (as long as the program can be invoked from the command line using the syntax in Section 2.2). Ensure that each submitted file contains a comment with your name and student number.

Ensure that all code files needed to compile and run your code in ECS 242 are submitted. Only the files that you submit through conneX will be marked. The best way to make sure your submission is correct is to download it from conneX after submitting and test it. You are not permitted to revise your submission after the due date, and late submissions will not be accepted, so you should ensure that you have submitted the correct version of your code before the due date. conneX will allow you to change your submission before the due date if you notice a mistake. After submitting your assignment, conneX will automatically send you a confirmation email. **If you do not receive such an email, you did not submit the assignment.** If you have problems with the submission process, send an email to the instructor **before** the due date.