Department of Computing and Information Systems

COMP10002 Foundations of Algorithms
Semester 1, 2014

Assignment 1

Learning Outcomes

In this project you will demonstrate your understanding of arrays, pointers, input processing and functions that make use of them. You will also extend your skills in terms of program design, testing and debugging.

The Story…

Data Science, Big Data, Analytics are transforming the way industries garner insights from data, generating billions for the economy annually. Mines forecast demand and predict equipment failure, banks process billions of customer transactions and balance investment risk, medical research identifies genetic features that influence predisposition to disease.

These disciplines rest on the shoulders of machine learning and databases, where the basic tasks of summarising and "slicing and dicing" data – selecting records that satisfy a condition or attributes of interest across a whole dataset – are extremely important. Your task in this project is to develop a *data wrangler* that works with datasets of records, each a sequence of positive integers. The wrangler has a simple interface in which data can be stored and manipulated. For example, a session with your program (where `>` is the prompt from the system) might look like:[1]

```
netbook: ./proj1 4
> a 11, 12, 13, 14
Added record:
11, 12, 13, 14
> a 21, 22, 23, 24
Added record:
21, 22, 23, 24
> ?
11, 12, 13, 14
21, 22, 23, 24
> p 3, 1
13, 11
23, 21
> ^D
netbook:
```

Note the extremely simple *syntax*: records can be added with an `a` command, the dataset of records can be printed with an `?`, and the dataset can be projected to select columns using `p`. The interface is a simple form of interpreter like you would have used when learning python, and like the popular R statistical computing program. The interface prompts the user for the next command, and provides useful feedback (including error messages when appropriate; it does not simply "quit" when it encounters something invalid), and the result of processing commands. The commands (particularly project) are simple versions of what is offered by database systems like MySql or Microsoft SqlServer. The *parsing* of syntax should be robust to any amount of space between components of commands (and indeed work without space, since like C the components of commands in this "language" is not delimited by space but rather commas and reserved keywords).

[1] The program was launched from the command-prompt with argument `4` indicating records will have four components. Two records are added to the dataset; the dataset is printed; and then "projected" to column `3` followed by `1` (counting `1` from the left).

You are to extend the wrangler from a starting point that implements the above, to cover additional commands and functionality that mirror some important analytics features found in databases.[2]

Stage 1 – Warm Up (marks up to 3/15)

Obtain a copy of the skeleton file using the link on the LMS, and spend some time (hours!) studying the way it is constructed, including compiling and executing it on the example input. It only supports a couple of very limited commands but is already quite a complex program! Read through this project specification before starting, so that you can plan elements of your program design.

Note all records added to the dataset via the add `a` command must have a number of components or columns as specified in the command-line argument passed to main. This is needless and arguably clumsy. Modify the program to remove the command-line argument, and instead have the required number of columns determined by the number of components in the first record added (up to a maximum `COLS`).

Stage 2 – Selecting Records by Property (marks up to 7/15)

Note the project command `p`, how it is implemented and called within the skeleton program. Where this command selects columns by their index, you are to design and implement a *select* command `s` which selects rows satisfying a given condition. For example for the dataset above:

```
> ?
11, 12, 13, 14
21, 22, 23, 24
> s 2 > 12
21, 22, 23, 24
>
```

The first component to the select is a column index (starting 1 from left); the second a binary operator `<`,`>`,`=` for (strictly) less than, greater than, and equality respectively; the third component is a positive integer. You are to process this command by printing out the result of keeping only those rows whose specified column value satisfy the comparison.

Stage 3 – Composing Commands (marks up to 11/15)

So far, your wrangler can only process stored data. This stage has you refactor and extend processing to running commands on the output of other commands. This is to be accomplished by introducing a unix-like pipe `|`. Single commands should run as before, but your system should support piping out of and into the project and select commands (it is illegal to pipe add and print commands). Only the output of the final command in a chain of pipes should be printed. Thus, piping allows you to *compose* complex commands made from our simple building blocks.

```
> ?
11, 12, 13, 14
21, 22, 23, 24
31, 32, 33, 34
> p 4, 1 | s 2 > 11
24, 21
34, 31
>
```

[2] Databases go much further than what we'll cover. They store several datasets in *tables*, which can be related to one another. Data structures called *indexes* built for that admit efficient *joins* matching up rows of tables. Databases process complex *query languages* and are highly optimised.

Stage 4 – Data Aggregation (marks up to 15/15)

An important task in wrangling data is summarisation. Next you will implement four types of summarisation via a new aggregate command g.

```
> ?
11, 12, 13, 14
21, 22, 23, 24
31, 32, 33, 34
> g a 1
21.0
> g s 1
8.164966
> g m 1
21.0
> g c 1, 2
1.0
```

This command can be piped, but only at the end of a chain of pipes (its output is not a dataset per se). In turn, the four aggregation operators are

- **Average**. Compute the floating-point average $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$ of the values $x_1, \dots, x_n$ found in the specified column.
- **Standard deviation.** Compute the floating-point population standard deviation $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$ of the values $x_1, \dots, x_n$ found in the specified column. Here μ represents the average.
- **Median.** Of the ordered values in the specified column, compute the floating-point middle value if there are an odd number of rows; otherwise the floating-point average of the two middle values. *E.g.* the median of 5,1,3,2 is 2.5
- **Correlation.** Compute the floating-point correlation between two specified columns of values $x_1, \dots, x_n$ and $x_1, \dots, y_n$ as $\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\mu_x)(y_i-\mu_y)}{\sigma_x\sigma_y}$ where the μ and σ designate the respective mean and standard deviations.

The syntax for these four variations of the aggregate command are given in the example above: g followed by a letter specifying the type of aggregation either a,s,m,c then for the first three cases a column index single positive integer, and in the last case a comma-delimited pair of positive integers.

Stage 5 – Plotting (marks up to 15/15)

In the interests of understanding your data, it is often nice to visualise what you have with simple plots. For two columns we might like to produce a scatter plot with the first column on the y-axis and the second on the x-axis, for example:

```
> ?
1, 5
10, 2
3, 7
2, 5
2, 4
1, 3
> v 1, 2
Scatter plot of Columns 1 (y) and 2 (x)
  | *
9 |
8 |
7 |
6 |
5 |
4 |
3 |      *
2 |   **
1 |  * *
  +-------
   1234567
```

The axes always begin at 1 which is the minimum possible value in our data. There is no max for a column, so the plot stretches to accommodate the data. The axis "ticks" run from 1 up to the minimum of 9 and the max value (here column 1 had a maximum of 10, so the max tick shown is 9; while column 2 only went to 7). Each record is visualised as an asterisk.

If you are successful in this stage, you will be able to earn back up to two marks you lost in the earlier stages (assuming that you lost some, you can't go past 15/15 in total). But *please* don't start this stage until your program is complete through to Stage 4 and is (in your opinion) perfect. (Still keen? Visualise histogram of a single column *e.g.,* `v 1`)

The boring stuff…

This project is worth 15% of your final mark. You don't have to do Stage 5 in order to get 15/15.

You need to submit your program for assessment; instructions on how to do that will be posted on the LMS. You can (and should) submit both early and often – to get used to the way it works, and ensure you have a submission in before the deadline. Only the last submission that you make before the deadline will be marked.

You may discuss your design/plans during your workshop, and with others in the class, but what gets typed into your program must be individual work, not from anyone else. So, do not give hard copy or soft copy of your work to anyone else; do not "lend" your memory stick to others; and do not ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say "are you out of your mind?" if they ask for a copy of, or to see, your program, pointing out that your refusal, and their acceptance of it, is the only thing that will preserve your friendship. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode.

Deadline: Programs not submitted by 2:00pm on Monday 12 May will lose penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email Ben, benjamin.rubinstein@unimelb.edu.au.

And remember, algorithms are fun!