**Technical**

UNIX

Project: Data Driven File Management

V2.0

© FDM Group Ltd 2017. All Rights Reserved.

Any unauthorized reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

**Problem**

You have a very large collection of files with random, unhelpful file names; office staff are finding these files impossible to manage. The files must be arranged into a new tree structure based on an item of data within each file.

Once this is done, a verification program is needed to check each file has, in fact, been correctly placed. Once this is in place, a correction program should fix these errors.

Management summaries are needed from data held in a subset of the files.

The files to be processed may be in any location. There should be no dependency in your scripts on the files being in a specific place, just within the current directory. This means your scripts should use relative rather than full pathnames for the files.

## Phase 1 – Basic Functionality: Placing the files into a new directory structure

Create a directory called **filesToSort** in your **$HOME** directory and ‘**cd**’ into it. From there run the command ‘**tar -xf /var/tmp/fts.tar**’. This will create the files you need to process. There are around 200 of them, all with names starting ff\_.

The files have similar data, in the same format throughout. Here is an example (please note that, throughout this document, examples are provided for illustration only; they do not necessarily reflect the actual data you will be processing):

[~/filesToSort] $ cat ./ff\_1683451198

2015-07-21

Tuesday

Leeds

3416293

2001-01-02

Tuesday

St-Helens

106293

Each data file (ff\_xxxx..xx) should be placed in one of a set of subdirectories based on the date in the first line of the file.

For example, the file aboveshould be placed in a sub directory underneath the current directory (in your case, *filesToSort*) ‘**2015/07/21’**. This means it should be in a directory named ‘21’ within a directory named ‘07’ within a directory named ‘2015’ within the current directory.

Directories should only be created when needed so that only the minimum number of directories needed to house the files is actually created.

Write a script called **makeStructure** that moves any files beginning with ‘ff\_’ into the structure described (and creates that structure if required).

After running the script:

[~/filesToSort] $ makeStructure

The tree structure should go from something like this:

[~/filesToSort] $ ls -R

.:

ff\_100344141 ff\_7067218 ff\_15829945 ff\_12023763 ff\_1922301 ff\_1329830596 ff\_14923177 ff\_142819214 ff\_167532759 ff72616519057….

To something like this:

[~/filesToSort] $ ls -R

.:

2000 2001 2002 2003 2004 2005 2006 2007 2008 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019

./2000:

02 03 04 09 10 11 12

./2000/02:

09

./2000/02/09:

ff\_1646921307 ….

## Phase 2 – Data Verification & Correction

#### Scenario

Unfortunately, a glitch has caused corruption to some of the files that have been placed in the new structure. The data corruption is not severe and only affects the date stamp in the file on line 1. This means the date stamp can be regenerated from the file pathname.

#### Putting the damaged data in place

From the directory ‘filesToSort’ run the following commands:

* ‘**pwd’** # To **make sure** you are in filesToSort
* ‘**rm –r 20\* ff\*’** # Delete any unneeded files. Ignore any error messages
* ‘**tar –xf /var/tmp/damagedData.tar** ‘ # Get the files

This will put the damaged files in place and in the appropriate directory structure as discussed in phase 1.

### Phase 2A – Error Detection

Write a script called **checkData** that identifies the files with damaged date entries. The script should produce a list of filenames with the non‑matching date stamps. For example:

[~/filesToSort] $ checkData

./2014/06/14/ff\_1147710362

./2004/05/27/ff\_1525617511

./2004/01/16/ff\_1658910840

./2005/01/18/ff\_1664330311

./2005/02/16/ff\_1467231391

…

This must **not** be done by simply checking to see if the date in the file is in a date format.

### Phase 2B – Error Correction

Write a script called **correctData** that corrects the date entry using information in the file’s pathname. Prove this has worked by running checkData and seeing no output.

## Phase 3 – Reporting Population Statistics by Year

The last line of the data files is an estimated town population. Write a script called **statPopnYear** that calculates statistics for qualifying files.

The script should accept a year number followed by a keyword that shows the information to be displayed. The year number should correspond to one of those available in the tree. The keyword should be one of ***average***, ***minimum***, ***maximum*** or ***all***.

The script should terminate with an error message and error status if:

* The number of arguments supplied is not two
* The year number does not correspond to one available in the tree
* The keyword is not one of those listed

The output should be similar to the following examples (remember that the data might be different in the sample data you are provided, so do not use these figures to check your calculations are correct):

[~/filesToSort] $ statPopnYear 2000 all

average is 934512

maximum is45921

minimum is 18320

[~/filesToSort] $ statPopnYear 2000 average

The average population for the year 2000 was 934512

[~/filesToSort] $ statPopnYear 2000 maximum

The maximum population for the year 2000 was 45921

[~/filesToSort] $ statPopnYear 2000 minimum

The minimum population for the year 2000 was 18320

## Phase 4 – Creating and Processing Indexed Data

### Phase 4A – Creating an Index File

Index files are often used in file systems as a quicker, more efficient way of searching through large numbers of files for predetermined data.

The third line of the data files contains a town name on which the population data is based.

Write a script called **createTownIndex** that will create a file index called **townFileIndex** for the towns in the data. This file will contain a list of towns and the files that are associated with that town.

It should output the data in the format **town:filepath** and the data should be stored alphabetically by town name. It should look something like this:

[~/filesToSort] $ cat townIndex

Aberdeen:./2016/12/23/ff\_2182327097

Aldershot:./2017/08/23/ff\_349812947

Altrincham:./2006/01/23/ff\_298983010 …

Note – Some towns may be repeated due to there being data for multiple dates. It is acceptable to have repeated town names in your index file.

### Phase 4B – Reporting Yearly Population Data Town Using Index File Information

Create a script called **getPopTown** to retrieve the population data for each file containing a given town.

Your script should **not** search through all the files in the filesToSort directory but rather use the file index you have just created to retrieve information from only the relevant files.

The script should adhere to the following rules:

* If there are no arguments, the script should exit with an error message and appropriate exit status
* If the town does not exist in the records, your script should output an error message saying “Cannot find this town in the current records”
* It should accept multiple arguments

The output should look like the following:

[~/filesToSort] $getPopTown Bristol Burnley Cheadle

Bristol:

2003-08-23:534898

2006-05-03:510992

2008-12-31:496044

Burnley:

2016-08-05:87231

Cheadle:

Can not find this town in the current records

### Phase 4C – Reporting population statistics by town

Create a script called **statPopnTown** to automatically report the average, maximum and minimum population in a town aggregated for all years.

The script should adhere to the following rules:

* If there are no arguments, the script should exit with an error message and appropriate exit status
* If the town does not exist in the records, your script should output an error message saying “Cannot find this town in the current records”
* Should accept multiple arguments
* The script should create the file index required if the file does not already exist

[~/filesToSort] $statPopnTown Stockport

Stockport:

The average population is 13319

The minimum population is 8765

The maximum population is 14572

Make sure to look up the town names using the townFileIndex. Make use of previous code you have written to create an efficient script that minimized code duplication..

## Coding Standards

The coding standards the script should conform to are:

* Use of comments to document the script
* Meaningful identifier names
* Tidy code using blank lines, indentation and no unnecessary statements
* Use of functions for repeated code

You must only use commands taught in this course (you can use any option associated to those commands) in order to fulfil the requirements.

## Backups

Ensure you take regular copies of your scripts. It is recommended you take a backup after you have completed and tested a working version.

## Marking

The marking process involves running your script to check it works. Your trainer will award you 0 points if they cannot get it working.

Your trainer will also be reviewing your code to ensure it meets the coding standards listed above. Additionally, any script whose logic cannot be defended will be awarded 0 points.

Although you may discuss concepts with fellow classmates, answers and code cannot be shared. Under no circumstances are you allowed to work at another’s workstation while they are logged in.