Intermediate Linux: Exercises

In these instructions the *first* character "**$**" in the command examples should not be typed, but it denotes the command prompt.

Some command lines are too long to fit a line in printed form. These are indicated by a backslash "\" at the end of line. It should not be included when typing in the command. For example

```
$ example command \
continues \
and continues
```

Should be typed in as:

```
example command continues and continues
```

0 Download and unpack the exercise files (do that first time only):

Go to the workshop home page (www.csc.fi -> click at the workshop name link at the right column, scroll down and download the tar archive from the link, save as …)

Open a terminal, **cd** to the folder where you downloaded the archive, unzip and untar the file:

```
$ tar xzvf Intermediate_Linux-exercises.tgz
```

The **v** (verbose) flag in the tar command shows the files with the path that are untarred. You see that you'll get a number of files in a subdirectory named linux-exercises. Go to that directory with the **cd** command.

1 Find shell scripts

Metadata: commands in this exercise: **file**, **grep**, **wc**

Metadata: Demonstrate the usage of pipes in simple tasks like finding the number of certain files in a directory.

1) How many shell script files there are in /usr/bin directory?

There are quite a bunch of files in */usr/bin*, but which of those files are actually shell scripts? To find out which type a file is, you will need the **file** command, which quite accurately will tell you what is inside a particular file.

2 Writing a conversion script

Metadata: commands in this exercise: **cp, convert, tar, animate.**

Metadata: The aim is to present a scripting solution of systematically manipulating a large set of similarly named files

0) Unpack the jpegs.tar file

```
$ cd FileRename
$ tar xvf jpegs.tar
```

1) Find out what directories and files were created

```
$ ls *.jpg
```

You find out that the naming is numbered, but the numbering is not in order. Hence, if you produce an animated gif from this bunch of files using the ImageMagik command convert:

```
$ convert –delay 30 –loop 2 *.jpg animation_small.gif
```

you will get strange resulst, as the input jpeg's are ordered by the first digit, i.e. {0, 10, 11, …, 19, 1, 20, 21, …, 29, 2, 30, …, 50, 5, 6, 7, 8, 9}

```
$ animate animation_small.gif
```

2) Resize and convert files

This means we want to add a heading 0 to the single-digit numbers to produce the correct order {00,01,02,03,…,09,10,11,…,50}. Doing that by hand would be tedious, as one would have to make 10 shell commands similar to that on

```
$ mv 0_singleframe_small.jpg 00_singleframe_small.jpg
```

Additionally, we also can resize the output using the ImageMagik command convert. In a combined way that would read for single digit numbers (0-9, here using 0):

```
$ convert –resize 200% 0_singleframe_small.jpg 00_singleframe_large.jpg
```

as well as for double digit numbers (10-50, here using 10):

```
$ convert –resize 200% 10_singleframe_small.jpg 10_singleframe_large.jpg
```

which would force us to give 51 shell commands. But we can use the power of loops within bash! You need to embed the commands using a variable for the counter within two loops, one from 0 to 9 and one from 10 to 50. Use the following syntax as a starting point and morph in the commands above:

```
$ for i in {0..9}; do ls ${i}_singleframe_small.jpg;\
echo "converting to 0${i}_singleframe_small.jpg"; done
```

You can (later on!!) look up the solution in convert.sh.

3 Creating a simple cryptography function

Metadata: command in this exercise: **cat, tr**

Metadata: The aim is to create a shell function, which can be used to crypt and decrypt text from both the standard input and files passed as command line arguments.

Metadata: ROT13 ("rotate by 13 places", sometimes hyphenated ROT-13) is a simple letter substitution cipher that replaces a letter with the letter 13 letters after it in the alphabet. ROT13 is a special case of the Caesar cipher, developed in ancient Rome. The algorithm provides virtually no cryptographic security, and is often cited as a canonical example of weak encryption. *(Wikipedia)*

1 Implement the crypto algorithm

The **`tr`** command is handy for translating or deleting characters from the input stream. You provide **`set1`** to be translated as the first argument to the command and **`set2`** for the translation table as the second argument:

```
$ echo "Make it right for once and for all" | tr [A-Za-z0-9] [N-ZA-Mn-za-m3-90-2]
```

Here we specify that all letters *A-Z* will be translated so that the first 13 letters (*A-M*) will become *N-Z* and the rest (*N-Z*) will become *A-M*. Thus, letter *A* becomes *N*, letter *B* becomes *O* and so on. Similar rules will be applied for letters *a-z* and for numbers *0-9*. Since there are 26 letters in the (English) alphabet, the same algorithm will also work for de-ciphering – you can try it out by entering the ciphered text for the **`tr`** command again.

2 Make it a shell function

In order to avoid typing the long **`tr`** command over and over again, try making it a function so that one could simply type:

```
$ echo "Make it right for once and for all" | rot13
```

3. Improve the function so that it can read files, too

The function now reads its input only from standard input. Modify the function so that one can pass file(s) to crypt as argument(s) to the function, as follows:

```
$ rot13 crypt_input1 crypt_input2
```

(Note: **`tr`** does only read from *stdin* so you need to feed the files e.g. with **`cat`** command through a pipe.)

4. Improve further so that the function work both for stdin and for files

If there are arguments to the function, then it should read input from the file(s), otherwise it should use the standard input as a source.

```
$ rot13 <<< "Make it right once and for all"
$ rot13 crypt_input1 crypt_input2
```

Hint: check first if there are any arguments by checking **$1**. Other variables you need are **$@** and **$***.

The solutions to each step (2..4) can be found from the files Solution(2..4), respectfully.

4 Geographic data manipulation

Metadata: command in this exercise: **wc**, **head**, **tail**, **cat**, **sort**, **gnuplot**, pipes

Metadata: The aim is to create use shell text utilities to find certain values in a dataset.

We will use the dataset from NSDIC that is https://nsidc.org/data/NSIDC-0119/versions/1 and systematically investigate some properties (max elevation, ice thickness) using UNIX tools, only. The area (should you be interested) we investigate is part of the Antarctic ice sheet (see picture right), called Marie Bird Land. If you want to further use the data in any way, please check the NSDIC web services for their conditions. Data could be downloaded using the command (just as an example. **DON'T DO THAT**! – we do not want to blow up our and NSDIC's network).

```
$ wget ftp://sidads.colorado.edu/pub/DATASETS/AGDC/luyendyk_nsidc_0119/*
```

INSTEAD untar-gzip the provided file:

```
$ tar xvzf luyendyk_nsidc_0119.tgz
```

Now we have two ASCII files:

```
$ ls -l *.txt
-rwxrwxrwx 1 root root 170M Mar  9  2004 srfelev.txt
-rwxrwxrwx 1 root root 114M Mar  9  2004 icethick.txt
```

They are large. Check the number of entries (= number of lines) using the **wc** command. We just want to work with a smaller sub-set of the data. Hence we reduce the size to the first 10000 lines, only.

```
$ head –n 10000 srfelev.txt > srfelev_reduced.txt
$ head –n 10000 icethick.txt > icethick_reduced.txt
```

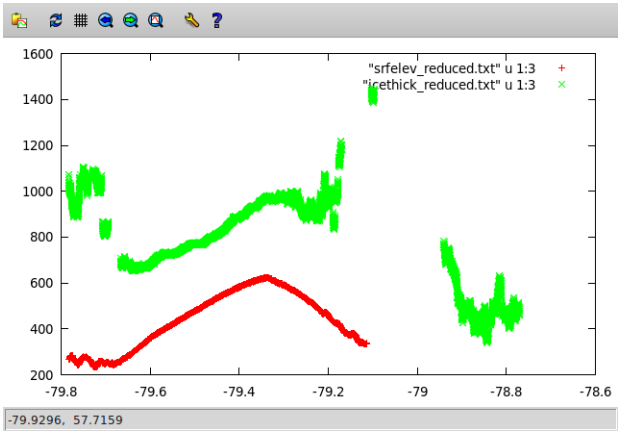
The problem now is, that we do not have a consistent dataset at places (see gnuplot script)

```
$ gnuplot showdata1.gp
```

If nothing shows, you might have to install the X11 version of gnuplot, which is done with

```
$ sudo apt-get install gnuplot-X11
```

(just answer “Y” if asked to download the packages and install).



Hence we would like to reduce the range of both datasets to be confined within -79.6 and -79.2 degrees (southern latitude). Let’s inquire the position within the file for the lower bound:

```
$ cat -n srfelev_reduced.txt |grep "79.6000"
```

Output:

```
  2879      -79.600072 -144.39008  360.9 1137.4 1998 358 10256.30 RTZ8/32\
Wy-Y11a
```

And the same for ice thickness:

```
$ cat -n icethick_reduced.txt |grep "79.6000"
  1962      -79.600072 -144.39008  681.5 1998 358 10256.30 RTZ8/32\
 Wy-Y11a
```

So, we have to use the last (10000 – 2879) = 7121 and (10000 – 1962) = 8038 entries.

```
$ tail -n 7121 srfelev_reduced.txt > srfelev_reduced2.txt
$ tail -n 8038 icethick_reduced.txt > icethick_reduced2.txt
$ gnuplot showdata2.gp
```

Now, the same for the upper bound of -79.2 degrees:

```
$ cat -n srfelev_reduced2.txt |grep "79.2000"
  5892      -79.200076 -147.74868  …
$ cat -n icethick_reduced2.txt |grep "79.2000"
  5739      -79.200076 -147.74868  …
```

But now we need the heading lines, not the trailing and spare us the math!

```
$ head -n 5892 srfelev_reduced2.txt > srfelev_reduced3.txt
$ head -n 5739 icethick_reduced2.txt > icethick_reduced3.txt
$ gnuplot showdata3.gp
```

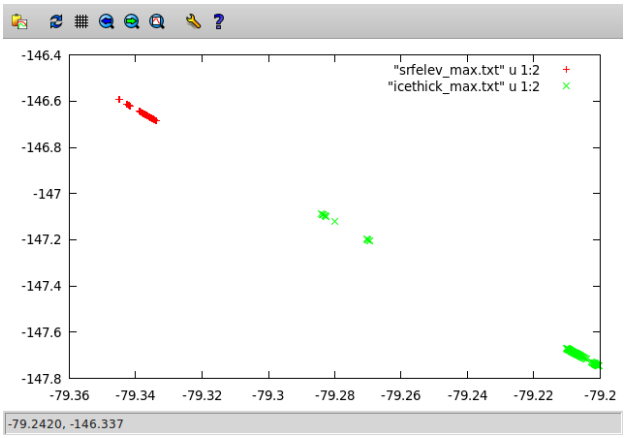
NB.: from the deviation of these two figures you can already tell that you still have 153 missing ice thickness entries.

Let's sort the lines according to the thickest ice and the highest elevation and extract the 100 maximum values

```
$ sort -n -k3 -r srfelev_reduced3.txt|head -n 100 > srfelev_max.txt
$ sort -n -k3 -r icethick_reduced3.txt|head -n 100 > icethick_max.txt
```

Let's check how max ice thickness and elevation correlate in their positions (they don't)

```
$ gnuplot showdata4.gp
```



Give the exact value of maximum thickness and elevation.

Do the same for minimum values and check this correlation.