COMP2700 LAB 2: LINUX COMMANDS & SHELL SCRIPTS

In this lab, we will be looking at a few more advanced concepts and commands in Linux, continuing what we have discussed in Lab 1, and Linux shell scripts. Shell scripts are used pervasively in Unix-like system administration and understanding how they work and their shortcomings are an important part of understanding Unix security. This lab is meant to provide a minimum background necessary to follow more advanced lectures and labs related to Unix/Linux security later in the course; it is not meant to be a crash course on Linux administration or shell script programming. Several references are provided at the end of this lab manual if you would like to learn more about shell scripting.

There are a number of exercises that you are asked to complete in this lab. These exercises are for you to work through the concepts presented in each section. After the lab, you are required to complete a lab quiz, that will be posted online on the Wattle course site. The quiz must be taken within 1 week from the date of the quiz. Details will be provided on the Wattle course site.

One or more exercises are marked with a '*', indicating optional extension exercises that will not be discussed during the lab. You are not required to complete them, but you are encouraged to complete them.

On the completion of this lab, students are expected to:

1. Be able to perform basic input/output operations on shell: creating simple files, writing, reading, or appending contents to files, searching contents of files and redirecting input/output of operations on files.
2. Demonstrate understanding of the concepts of environment variables and how to read and write to environment variables.
3. Be able to perform simple composition of shell commands: sequential composition and piping.
4. Write simple shell scripts featuring basic control commands: if-then-else, strings and integer comparison and loops.

0. LAB SETUP

***Notation convention**: In the following, when writing shell commands, we will use lines starting with the symbol '$' to denote that they are shell commands. For example,*

```
$ ls
```

denotes an execution of the shell command 'ls' (for listing files and directories). The symbol $ is not part of the command.

The lab setup is the same as Lab 1, so we will continue using the lab VM we have set up in Lab 1. We will also be using the same **lab1.tar.gz** that was provided in Lab 1. We repeat here the instructions to set up the directory 'lab1' that we will use again in this lab:

1. Login to the lab VM as user admin2700.
2. Delete the previous copy of lab1 from the previous lab, if any, using the following command:

   ```
   $ sudo rm -rf /home/alice/lab1
   ```

3. Change to user **`alice:`**

   ```
   $ su -l alice
   ```

4. Download the file **lab1.tar.gz** if you have not already downloaded it.

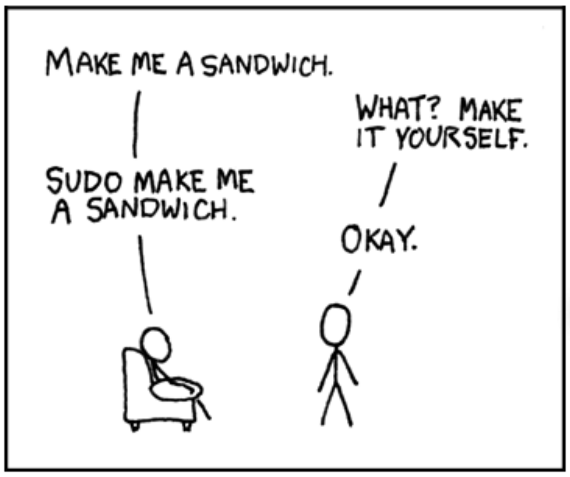
   ```
   $ cd ~/
   $ wget http://users.cecs.anu.edu.au/~tiu/comp2700/lab1.tar.gz
   ```

5. Extract the files in lab1.tar.gz using the following command (assuming you put the file in the the home directory of alice):

   ```
   $ extract-lab lab1.tar.gz
   ```

   This will create a directory called lab1 in /home/alice.

A NOTE ON THE SUDO COMMAND



(Source: xkcd.com)

From time to time you may need to perform some administrative tasks that require an escalated privilege, such as adding/removing files created by another users (as we've seen in lab setup step 2 above), changing ownership of files, changing certain access control policies of executables, etc. In Linux, only the "root" account has unrestricted access to all resources in the system, but you cannot connect to the system as the root user directly. However, an administrator account (such as "admin2700" in the lab

VM) can escalate their privilege to the equivalent of the “root” account using the “sudo” command. Prefixing a command with the word “sudo” causes that command to run under an escalated privilege. To use the sudo command in the lab VM, you need to be logged in as user admin2700.

We will return to the use of sudo command in the lab on Linux/Unix security. Use this command sparingly, only when needed.

1. FILE OPERATIONS

In Lab 1 we have seen how to perform simple operations on files: copying, deleting and displaying contents of files. We will look at a few more advanced commands.

SEARCHING THE CONTENT OF A FILE

Syntax*:*

> ***grep [-options] Pattern File***
> **Grep** command is used to search for lines of text that match the pattern (specified as a regular expression), and outputs only the matching lines.

Examples:

```
$ grep Chapter ~/lab1/sample_files/comp2700.txt
```

will display lines in comp2700.txt containing the string ‘Chapter’.

The pattern in the first argument of grep can be any *regular expressions*; we will not cover them here, but the man page for grep provides more details if you are interested (use ‘man grep’ to bring up the man page).

The option -n can be used to display the line numbers of matching texts. For example

```
$ grep -n Chapter ~/lab1/sample_files/comp2700.txt
```

will produce a similar result as the previous command, but each line in the output will be prefixed by a number indicating the line number of the matching text in the input file.

As with `ls’ command, the [File] argument can be replaced by a pattern; in that case, all files matching that pattern will be searched.

The option -R can be used to search a directory recursively (to include all files in the subdirectories).

EXERCISE 1. Use grep to search the contents of all files in /etc/ directory (including all its subdirectories) for keyword ‘password’ (ignoring case -- so your search should include those files containing ‘Passwords’, for example). *Note that you may encounter error messages containing ‘permission denied’. You can ignore that for now.*

FINDING FILES

Syntax:

which filename

This command shows the full path of the command ***filename***. For example: running ***which ls*** will produce the answer '/usr/bin/ls'. That is, the command 'ls' refers to the executable file /usr/bin/ls.

This command is useful if you have different executable files with the same name, but in different directories, and you want to find out which version it is that you are invoking when you type the command name.

find

This command searches for files that fit certain patterns and other constraints. It is a very powerful command to find files, with many options that we will not cover here. We will see in later labs more advanced uses of 'find'. A common usage is the following:

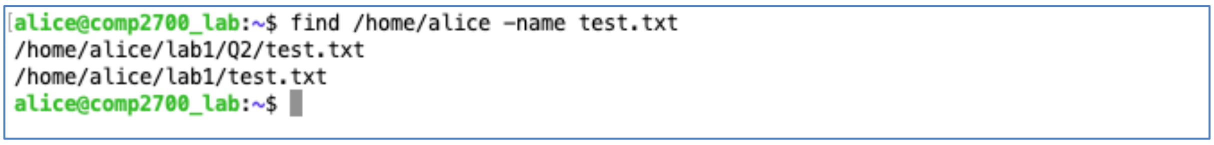
find [path] -name [search-string]

This command examines all files and subdirectories in a given path and print the names of the files containing the search string in their name.

Examples:

```
$ find /home/alice/lab1 -name test.txt
```

searches for a file whose name is 'test.txt' and prints the full path to the matching file(s).



```
alice@comp2700_lab:~$ find /home/alice -name test.txt
/home/alice/lab1/Q2/test.txt
/home/alice/lab1/test.txt
alice@comp2700_lab:~$
```

The search string can be a pattern. For example, the following will find all file/directory with name starting with Q

```
$ find /home/alice -name "Q*"
```

Note that patterns must be surrounded by double-quotes.

EXERCISE 2. Using the "find" command, find all hidden files in /home/alice. In Linux, a hidden file is a file whose name starts with a dot ".", for example, `.bashrc`.

EXECUTING A FILE

To execute a (program) file, simply specify the full path to the file. For example:

```
$ /home/alice/lab1/hello
```

executes the hello program in lab1 directory.

Unlike Windows, for example, executable files in Linux do not need to have any specific extensions (e.g., ‘exe’ or ‘com’). Instead, executable files are indicated by the ‘executable’ permission bit associated with the file -- we will cover this in detail in the lectures on Unix security.

The various shortcuts for directory can be used here as well, e.g., the above command can be executed (by Alice) using

```
$ ~/lab1/hello
```

since ~ expands into /home/alice. Similarly, if the current directory is /home/alice/lab1/ we can simply type

```
$ ./hello
```

since the dot (`.’) translates to the current directory /home/alice/lab1.

The shell commands we have seen earlier can be executed without writing the full path, e.g., we simply type "ls" rather than "/usr/bin/ls" to execute the ls file. This is because the programs for these commands are in the search path of the shell environment (more later when we discuss environment variables).

EXERCISE 3. Find where the program that corresponds to the shell command ‘cat’ is located and copy it to Alice’s home directory and rename it to ‘mycat’. Run the ‘mycat’ program to display the content of ~/lab1/ab.txt.

2. INPUT/OUTPUT REDIRECTION

INPUT REDIRECTION (<)

Input redirection allows you to replace the standard input (i.e., keyboard) with a file. For example, consider the grep command we have seen earlier. If no file names are provided as arguments to the command, then it will take input from the standard input, e.g., if you type

```
$ grep aaa
```

then it will wait for input from the standard input and treat it as if it is a file where the string "aaa" is to be searched from. Use Ctrl-D (which signifies the "end of file") to tell the grep command that the "file" (standard input) has ended so it will start the search. You can redirect the standard input to, say, the file ~/lab1/ab.txt

```
$ grep aaa < ~/lab1/ab.txt
```

This command will treat the file ~/lab1/ab.txt as if it is input from keyboard. Of course, you can also use

```
$ grep aaa ~/lab1/ab.txt
```

to directly find the string "aaa" in **`ab.txt`**, but the ability to redirect input will be useful to chain commands, as we will see later with piping.

OUTPUT REDIRECTION (>)

Output redirection is the reverse of input redirection. In this case, you can redirect output to the standard output (i.e., the display) to a file. For example, the command **`echo`** outputs a string to the display. We can redirect it so that it outputs to a file, say output.txt, e.g.,

```
$ echo hello > output.txt
```

This command causes the output of hello to be written to a file named output.txt.

PIPING (|)

We can also redirect the output of program X to the input of program Y. This is called 'piping'; we build a 'pipe' between X and Y. For example, to find all files and directory belonging to user 'alice', we can pipe together the ls command and the grep command:

```
$ ls -l | grep alice
```

The two commands are separated by the pipe `|'. The first command will output a list of files along with their attributes, the piping sends this output to the 'grep' program as the standard input.

APPENDING OUTPUT TO FILE (>>)

The operator >> will append the output of a command to a file. For example:

```
$ echo 'abc' >> test.txt
```

will append the text 'abc' to the end of file 'text.txt'.

EXERCISE 4. cracklib-check is a program to check the strength of passwords. It takes input from the standard input (keyboard). If a password is good, it will output the password and a message OK at the end of the password. For this exercise, use the file 'passwords.txt' in /home/alice/lab1. Use the cracklib-check program, together with the input/output redirection operators, to check the passwords stored in the passwords.txt, and output only the password entries that are OK.

3. ENVIRONMENT VARIABLES

Environment variables are placeholders for values that can be used in the shell environment. These variables can be used as macros to define simple data such as strings, or (shell script) functions. The bash environment has several predefined environment variables. We list a couple of them below.

HOME: This environment variable contains the home directory of the current user.

PATH: This environment variable contains a sequence of directories, separated by the colon (':') character. The directories in the sequence are consulted when the user launches a command without specifying the path to the executable. The bash system will look for the executable file in the directories set in the PATH environment variable.

USER: This environment variable contains the username of the current logged in user.

To see all the currently defined environment variables in the shell use the 'env' command:

```
$ env
```

PRINTING AN ENVIRONMENT VARIABLE

You can access the value of an environment variable by prefixing it with a `$' sign. For example, to display the content of the HOME environment variable, you can use the echo command, e.g.,

```
$ echo $HOME
```

This command prints the value of the HOME environment variable. Similarly, the command

```
$ ls $HOME
```

will list the content of the home directory.

SETTING AN ENVIRONMENT VARIABLE

You can create or change the value of an environment variable via the command 'export'. For example, you can set PATH to point to /home/alice by executing:

```
$ export PATH=/home/alice
```

Caution: This overrides the default search path of the bash environment and will cause the built-in commands not working. A safer way to alter the PATH environment is to add additional paths on top of existing:

```
$ export PATH=$PATH:/home/alice
```

The paths in PATH are separated by a colon ':'.

When a user types in a command in bash shell, bash will search for the program name matching the command based on the content of the PATH variable, in a left-to-right order, and execute the first matching program.

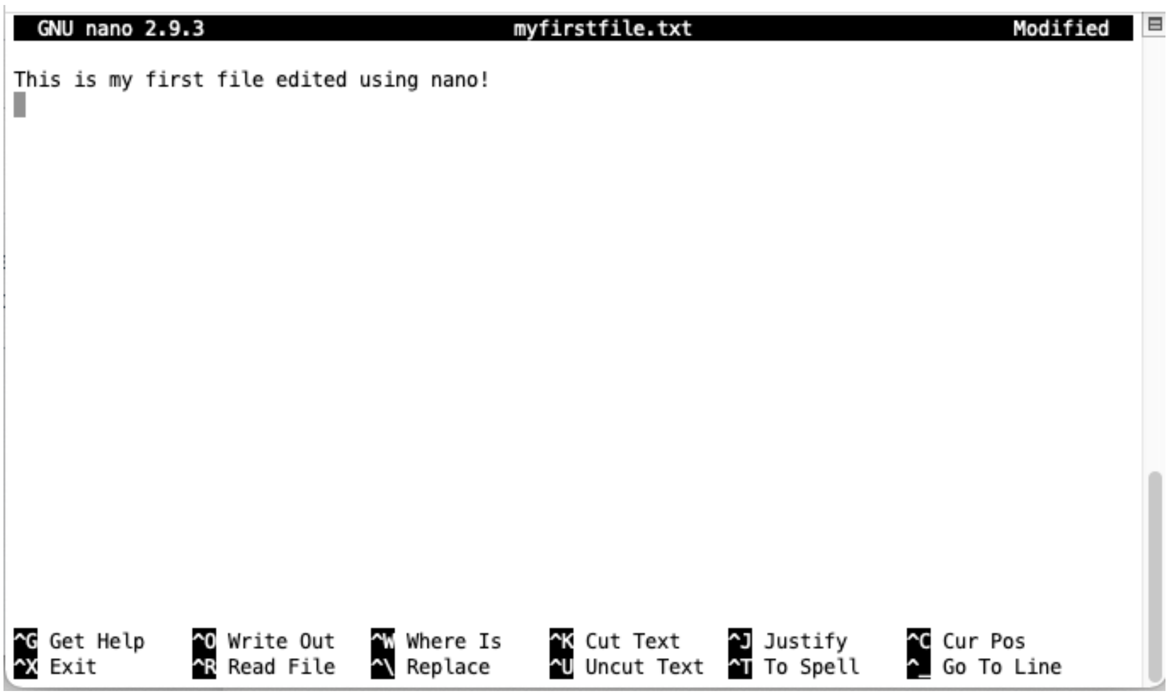
EXERCISE 5. Rename the file ~/lab1/hello to ~/lab1/ls and change the behaviour of your shell so that the command 'ls' points to ~/lab1/ls rather than /usr/bin/ls. What command(s) do you use to achieve that?

EDITING FILES

So far we have not shown how to edit files directly in the command line interface (CLI). There are several popular text editors in CLI, for example, nano, vim and emacs. For beginners, we recommend using nano, as it is relatively user friendly compared to some other editors. To edit a file, say, myfirstfile.txt, simply type the command:

```
$ nano myfirstfile.txt
```

If the file already exists, it will open it. Otherwise, it will be treated as a new file. The interface looks something like the following.



```
  GNU nano 2.9.3                  myfirstfile.txt                      Modified

This is my first file edited using nano!

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Spell    ^_ Go To Line
```

The bottom of the window in the figure above shows a few main commands of nano. The ^ sign next to a command, e.g., ^X, denotes the control key. For example, to exit the editor, simply press the control key and the X key at the same time. The commands are self-explanatory. We will not explain in details features of nano. For creating and editing shell scripts in the following exercises, it is sufficient to know how to open/write/read a file and exiting the text editor.

4. SHELL SCRIPTS

WHAT IS SHELL SCRIPTING?

In addition to the interactive mode, where the user types one command at a time, with immediate execution and feedback, Bash (like many other shells) also has the ability to run an entire script of commands, known as a "Bash shell script" (or "Bash script" or "shell script" or just "script"). A shell script may contain just a very simple list of commands — or even just a single command — or it may contain functions, loops, conditional constructs.

In your Bash Shell Script, the first line will always start with **#!/bin/bash**. This tells the system your script is a bash script.

CHEAT SHEET

A useful list of commands: https://devhints.io/bash

SOME SHELL SCRIPT EXAMPLES

Here are some examples of shell scripts. We will not be writing advanced scripts in this lab. The purpose here is to introduce students to the concept of a shell script to facilitate discussions about security restrictions on such scripts. For a more comprehensive introduction to shell script, please refer to the references at the end of this guide.

The sample scripts below are located in **/home/alice/lab1/shell** directory in the lab VM. The following examples assume you have made that directory your current directory, e.g., using the command

```
$ cd /home/alice/lab1/shell
```

HELLO, WORLD!

Here is an example of a hello-world program (in hello.sh file) to print 'Hello, World' to standard output.

hello.sh

```
#!/bin/bash
echo "Hello World!"
```

One may need to turn it into an executable file, by using the command

```
$ chmod a+x hello.sh
```

(we will cover chmod command in detail in lectures on Unix security).

Then to execute it, run:

```
$ ./hello.sh
```

VARIABLES

Variables in bash script need not be explicitly declared and can be read by prefixing it with a $ sign. To initialize a variable, use the = operator.

For example:

```
$ X=1
$ echo $X
```

will initialize the variable X to 1 and displays it on standard output.

ARITHMETIC

Arithmetic operations must be enclosed in double-parentheses.

For example: the following script

```
#!/bin/bash

X=1
((X=X+1))
echo $X
```

initialises X to 1, increment it by 1, and prints its content (which will be '2').

QUOTES

In bash, single quotes tell the bash environment to treat everything in between the quotes as it is. In particular, the $ sign will be ignored so the (environment) variables referred to in the quotes are not expanded.

For example:

```
$ echo 'The current user is $USER.'
```

will output exactly:

```
The current user is $USER
```

Double quotes tell bash that some characters in between the quotes have special meaning. In particular, the $ sign will be interpreted by bash as indicating what follows the sign is a variable and will be expanded to its actual value.

For example:

```
$ echo "The current user is $USER."
```

will output

```
The current user is alice.
```

SEQUENTIAL COMPOSITION

The operator ; (semicolon) tells bash that the commands separated by the semicolon are to be executed sequentially (from left to right).

For example:

```
$ echo hello; echo world
```

will execute two echo commands, one printing hello, and the other printing world.

This is useful when writing several commands in one line, e.g., if they are run directly from a command prompt.

THE TEST COMMAND AND THE LEFT SQUARE BRACKET [

One of the most confusing aspects of bash scripts is the use of brackets and parentheses. Unlike most other programming languages, brackets/parentheses are not intended to be used to group expressions. The left square bracket [actually refers to the file /usr/bin/[(yes, the file name is actually the left bracket "["). A more traditional name for "[" is the "test" command (/usr/bin/test). To see the full explanation of the test command, run "man test".

When the left bracket [is used in a command, it checks that its last argument is a right bracket "]". This serves no purpose other than giving the illusion of a grouping of expressions. More precisely, the syntax of the command /usr/bin/[is as follows

```
[ EXPRESSION ]
```

where EXPRESSION is any test. Two forms of tests that are commonly used are:

- String comparisons:
  - Testing for equality: e.g., $x = "hello"
  - Testing for inequality: e.g., $x != "hello"
- Integer comparisons:
  - Equality: e.g., $x -eq 10
  - Less-than: e.g., $x -lt 10
  - Less-than or equal: e.g., $x -le 10
  - Greater-than: $x -gt 10
  - Greater-than or equal: $x -ge 10

The command [can be replaced by the test command, omitting the right bracket]. For example, the following two commands are interchangeable.

```
[ $x -lt 10 ]
```

```
test $x -lt 10
```

The test/[command plays an important role in conditionals and loops.

Test/[commands can be combined using && (logical AND), || (logical OR). For example,

```
[ $x -lt 10 ] && [ $x -gt 5 ]
```

checks whether the variable x is in the range (5,10).

Equivalently, using the test command:

```
test $x -lt 10 && test $x -gt 5
```

IF-THEN-ELSE

The if-then-else command takes the form

```
if test-command
then
   commands
else
  commands
fi
```

where test-command is any command formed using either the 'test' or the '[' command discussed in the previous section.

The else part can be omitted if necessary:

```
if test-command
then
   command
fi
```

Here is an example of if-then-else (~/lab1/shell/**check_num.sh**):

```
#!/bin/bash
echo "Type the input integer, followed by [Enter]:"
read x
if [ $((x%2)) -eq 0 ]
      then echo "$x is even"
      else echo "$x is odd"
fi
```

The command '**read x**' reads an input from the standard input (eg, keyboard) and stores it in the variable x.

The command **elif** (i.e., "else if") can be used to write a nested if-then-else concisely. For example, the following two scripts (testbinary.sh and testbinary2.sh) are equivalent:

testbinary.sh

```
#!/bin/bash

read x

if [ $x -eq 0 ]
then
   echo zero
elif [ $x -eq 1 ]
then
   echo one
else
   echo 'not binary'
fi
```

testbinary2.sh

```
#!/bin/bash

read x

if [ $x -eq 0 ]
then
   echo zero
else
   if [ $x -eq 1 ]
   then
     echo one
   else
     echo 'not binary'
  fi
fi
```

EXERCISE 6. Write a shell script to convert a numerical mark, read from the standard input, into ANU grading scale: HD (80 - 100), D (70 - 79), CR (60 - 69), P (50 - 59), N (0 - 49).

FOR-LOOPS

Syntax of for-loops:

```
for loop-variable in value-range
do
   command
done
```

The value-range can be enumerated explicitly, as in:

```
for i in 1 2 3 4
do
   echo "iteration $i"
done
```

or specified as an interval:

```
for i in {1..4}
do
   echo "iteration $i"
done
```

Or we can use a more familiar C or Java-like syntax:

```
for ((i=1; i<=4; ++i))
do
  echo "iteraton $i"
done
```

WHILE-LOOPS

Syntax for while-loops:

```
while test-command
do
    command
done
```

For example:

```
i=1
while [ $i -le 4 ]
do
 echo "iteration $i"
 ((i++))
done
```

EXERCISE 7. Write a shell script to print the first n elements of the Fibonacci sequence, where n is read from the standard input.

MORE SHELL SCRIPT EXAMPLES AND EXERCISES (OPTIONAL)

If you find the previous exercises too easy, here are some advanced examples and exercises for you to try. They are optional and will not be covered during the lab. None of the assessment items in this course rely on these examples so you may safely skip this part.

loop_examples.sh:

```
#!/bin/bash
#the following for loop will clear all files and folders
#under folder_to_be_cleared directory

for i in $(ls folder_to_be_cleared/)
do
    rm -rf folder_to_be_cleared/$i
done

#syntax {Start..End..Increment}
for i in {1..5..1}
do
    echo "welcome $i times"
done

#calculate 1+2+...+10
counter=1
sum=0
while [ $counter -le 10 ]
do
    sum=$((sum+counter))
   ((counter++))
done
echo "1+2+...+10=$sum"
```

There are 3 loops contained in this file. First loop will find out all files and folders listed under **folder_to_be_cleared** directory and use these to construct the **rm** command to get rid of everything underneath. Second loop is a simple for loop, if you have studied C/Java/Python before. Last is the while loop which adds all the number from 1 to 10.

read_nums.sh:

```
#!/bin/bash
#usage: ./read_nums.sh [Input File]
Filename=$1
if [ -z $1 ];
       then echo "no input found"
else
num=( $(<$Filename) )
numlen=${#num[*]}

echo "There are totally "$numlen" numbers in the source file. They are:"
echo ${num[*]};
echo "And in reverse order, they are:"

for ((i=1;i<=$numlen;i++))
       do
       echo -ne ${num[$numlen-$i]}" "
       done
echo
fi
```

Execute:

```
$ ./read_nums.sh input_nums.txt
```

In this example the input_nums.txt contains 8 numbers. The program read all the numbers and output those in a reverse order.

EXERCISE 8 (*). In the directory ~/lab1/ there is a file, **numbers.txt,** containing a list of numbers. Write a bubble/selection sort shell script to sort the number in ascending order and save it in sorted.txt.

REFERENCES

- Stephen G. Kochan and Patrick Wood. Shell Programming in Unix, Linux and OS X (4th edition). Addison-Wesley Professional, 2016.
- Machtelt Garrels. “Introduction to Linux: A Hands on Guide”. 2010. http://tille.garrels.be/training/tldp/
- Arnold Robins. “Unix in a Nutshell”, 4th Edition, O’Reilly, 2005.
- Mike G. “BASH Programming: Introduction HOW-TO”, http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html