Iván Miguélez García 100383387

José Manuel Frías Salvachúa 100383533

**Exercise 1**

a)The idea to solve this problem is to get an input parameter and start to compare it with the text inside the selected file, so that when there is a match, the program gives us some information. In order to do it, first we need to take care of a very important part: the parsing. When the program reads the file in “/etc/passwd”, it’s getting a bunch of information that we need to manage in some way. The solution implemented is reading line by line and saving in different fields the information, which is separated by “:”. As we need the program to look for a user, and this information is stored at the beginning of the line, the program will compare the input parameter with the first field of each line. If there is a correlation, then the required information will be printed.

Otherwise, an error will be printed

b)This problem is basically the same as the previous one. In “/etc/group”, the information is also separated by “:”. We just need to tell the program to compare the input parameter (GID), with the position were that information can be found (in this case, the third field). In the same way as before, if there is a correlation, the required information is printed, and an error if there is no correlation.

**Exercise 2**

a)Show us the N more repeated lines in a file of our choice after changing any symbol different from a letter, number or space (such as a \*,\ or similar) into a new line.

**tr -c [:alnum:] [\\n\\*] < $1:** take standard input (what has been written by keyboard), find if there’s any complement of the set formed by all numbers and letters and change it with an space.

**sort:**rearrange all the lines in ascending order

**uniq -c:** Compress repeated statements and puts how many times each of it is repeated

**sort -nr:** sorts the lines in reverse order. This makes that the most repeated lines are first

**head -$2:** shows the first N lines being N the number introduced in the input line after the name of the file.

b)For controlling how many arguments have been passed we look at $#. If this number is different from 2, the command has been bad introduced and we print an usage error telling our user what is the correct input format. IF the number is correct we execute the command line established before

Tests:

**correct input:** Correct number of lines was printed.

**Correct file but number bigger than number of lines in the file:** Only number of lines available in the sorted list appear. No extra lines.

**Correct file but various numbers after it:** Usage message appears telling us the correct format.

**Introduce 2 parameters but both are numbers:** “No such file or directory”.

**No introduction of parameters:** Usage message (because we first take into account the number of parameters before trying to execute the command line).

**Exercise 3**

To solve this exercise, the main problem remains in parsing and manipulating the information given by the function “factor”. To do it, the program uses some piped command, and the output is stored inside a variable for later comparisons. Appart from “factor”, the command “cut” is used. The command factor returns an output with the following format: “n: m …”, where n is the number we want to factor, and the remaining letters are the factors of the number. There is only a special case, which is number “1”, that has no factors so only”1:” is printed.

If a number is prime, the returning output should be “n: n”, so with the command “cut” we can get the factors in the output and compare them to the number that is being tested. In this way, if the number is the same as the factors part of the output, then the program has found a prime number.

**Exercise 4**

We are told to, given an input directory change all of it’s jpg images that have a file size bigger than 1Mb. For this, we first store in a variable 1024 (1024 kilobytes is the same as 1 Mb). Next we execute a for loop that invoques all the files inside the directory introduced by standard input, storing each file of the loop in a variable. We compare what’s in that variable and search if it has a termination in jpg or derivatives (in the computers we worked on this format was given with different terminations: .jpg, .JPG, .jpeg and .JPEG; so we have if conditions for each of this possibilities. Next, we store in a variable the size of the file (du help us see how much space does this file use in kilobytes, then cut for erasing the name of the file and staying only with the numerical value), then compare this value with our comparing variable established before. If this new size is bigger than the established one we proceed to to the transformation of the image with convert and resize and show in the screen the file that we have modify (we simply use input file because in this variable is already stored the path to that file). We have use convert but by giving as output name the same one we introduce, we modify the original image (thanks to rewriting) so mogrify could have been used instead.

Test

We introduce a folder with images from all types. Only jpg (and derivatives) with size bigger than 1024 were changed. Other jpg images with smaller size remained equal and other types of images as gif or webp didn't change either. If we introduce an invalid folder, nothing happens and the shell waits for another command.

**Exercise 5**

a)The file exercise5.sh is much more bigger that its repackage version.

b)The script is using the dialog dependency to give the user some options to choose.

The first line is storing the dialog with all the different options.

“--backtittle” is setting “exercise5.sh” as tittle of the dialog.

”--output-fd 1” is setting the standard output as the output file descriptor of the dialog.

“--checklist” is setting the different options given to the user

In SIZE, the command stat is storing the total amount of data to be transfered, and its stored into $0.

Then, if the string WHAT is not empty, the command “tail -n +8 $0” gets the last 8 lines in $0, and pass them as input. “pv” won’t do anything with that information, so it will be the input of the next command. “pv” starts a progress bar, that will have a visual integer percentage with the option “-n”, uses “-s SIZE” to know the total amount of data to be transferred, and “-i 0.25” to wait that amount of seconds between updates.

After that, “base64 -d” decodes data returned by “tail”, and “tar -xzf” is extracting a gzipped archive from the file $WHAT.

What “2>&1” is doing, is basically redirecting the standard error (2) to the standard output (1). “&” is used in order to indicate that 1 is not a file name, but a file descriptor (output).

c) The script gets a directory from dialog. If it is empty, then the file is extracted to the working directory. If not, then it is extracted to the designed directory. We have to change some parameters in repackage such as the sed command parameters, in order for the file to be uncompressed properly

d)The script creates a temporary directory and saves the name “.\exercise5” in a variable.

Then, a compressed file is created with the name “.\exercise5”. After that, the file is truncated and some archives are added to it..

Finally, the temporary directory is deleted.