



INTRODUCCIÓN AL ENTORNO DE LINUX

LINUX (GNU/LINUX) es un sistema operativo tipo UNIX que cuenta con numerosas distribuciones. Este sistema es de código abierto y cuenta con una comunidad amplia para sistemas de cómputo y sistemas embebidos en general. Tal como sucede con otros sistemas operativos, las distribuciones de LINUX pueden ser utilizadas a través de una interfaz gráfica o por línea de comandos, siendo la segunda de interés para el curso. En esta guía se darán los comandos básicos que permiten una interacción con el terminal, así como los requerimientos mínimos para la compilación y ejecución de programas en C y Python.

ÍNDICE

1	INTRODUCCIÓN A LINUX	2
2	MANEJO DE ARCHIVOS EN LINUX	3
3	Programación de scripts de BASH	5
3.1	Creación y ejecución de un programa de BASH	5
3.2	Herramientas de programación	5
3.2.1	Ciclo iterativo for	5
3.2.2	Ejecución de líneas de comando para otros programas	6
3.2.3	Medición de tiempo	9
3.3	Ejemplo de aplicación	10
3.3.1	Problema	10
3.3.2	Solución	10
4	Tarea	15
4.1	Problema 1	15
4.2	Problema 2	15
4.3	Problema 3	16

1. INTRODUCCIÓN A LINUX

En un sistema operativo basado Linux, normalmente se utiliza el terminal para navegar y manipular el sistema de directorios. Algunos de los comandos más utilizados son los siguientes:

- a. **pwd**: (print working directory) Imprime el directorio de trabajo. En terminos sencillos te muestra la ruta o directorio en la cual te encuentras cuando estás en una terminal linux.
- b. **cd**: (change directory) Si se coloca solo, dirige al directorio raíz. Adicionalmente, se puede combinar con otras rutas para dirigirse a otros directorios:
 - **cd ~** : ir al directorio home
 - **cd ~/papers**: ir a /home/user/papers
 - **cd ~arqui**: ir a /home/arqui
 - **cd dir**: ir al directorio dir (relativo)
 - **cd /dir1/dir2/dir3**: ir al directorio dir3 (absoluto)
 - **cd -** : regresar al último directorio
 - **cd ..**: subir una jerarquía de directorio
- c. **mkdir** nombre-directorio:(make directory) Crea un directorio con el argumento de entrada el que se quiere crear. Ejemplo: *mkdir prueba* crearía el directorio prueba.
- d. **rmdir** nombre-directorio: (remove directory) Borra el directorio. Considerar que antes de borrar un directorio, éste se debe encontrar vacío, de lo contrario se mostrará un mensaje de error. Ejemplo *rmdir prueba* borrará el directorio prueba.
- e. **tree**: Muestra una estructura de árbol de directorios a partir del directorio actual de trabajo.

2. MANEJO DE ARCHIVOS EN LINUX

Linux maneja básicamente archivos ordinarios, directorios y archivos especiales. Linux maneja tres tipos de archivos/directorios:

- 1) **Archivo ordinario:** También llamado regular. Consiste en textos, imágenes, instrucciones de programas (código fuente), entre otros.
- 2) **Directorios:** Se manejan internamente como si fueran archivos (archivos de directorio), cuyo contenido es conocido por el *file system driver*¹.
- 3) **Archivos especiales:** Tienen un significado especial y generalmente están asociados a los dispositivos entrada y salida (I/O devices) o de comunicación entre procesos (*pipes*).

Así mismo, existen comandos de utilidad que permiten interactuar con estos archivos o directorios:

- a. **find:** Busca a partir del directorio que se le indique, y en forma descendente, cualquier archivo (por nombre, tamaño, permisos, etc.). Se pueden emplear comodines como el * (cualquier grupo de caracteres de longitud mayor o igual a 0) y el ? (un solo carácter).

Ejemplo: `find / -name ing.h -print`

Este comando debería buscar desde el directorio raíz y en forma descendente todos los archivos que contengan los caracteres “ing.h” al final del nombre, y una vez hallados imprime su ubicación.

- b. **cp:** Copia archivos

```
$ cp <nombre_archivo_original> <nombre_archivo_copia>
```

Copia un archivo en uno nuevo con otro nombre.

```
$ cp <archivo1> <archivo2> .... directorio
```

Copia los archivos enumerados al directorio indicado (el cual ya debe existir) con los mismos nombres.

- c. **rm:** (remove) Elimina archivos.

```
rm <nombre_archivo>
```

- d. **ls:** (list) Muestra el contenido del directorio de trabajo o de cualquier otro directorio que se le indique. Si se le invoca sin argumentos, muestra los nombres de los archivos sin ninguna otra información:

```
ls
```

Pero si se desea una descripción más detallada puede usarse la opción `-l`:

```
ls -l
```

¹ Software que se encarga de administrar y organizar el contenido de archivos en el sistema operativo

Esta opción muestra un listado de archivos en formato largo, en el directorio actual (a menos que se indique alguno como argumento).

- e. **chmod**: (change mode) permite definir los permisos de un archivo respecto a tres tipos de usuario (usuario, grupos, otros). Se suele usar dos formas para expresar los permisos:

- a) La octal asigna los permisos mediante valores en base 8. Ejemplo: *chmod 600 ejemplo*. Esto significa que el grupo de usuarios tiene permisos de lectura y escritura, que el grupo no tiene permisos y que otros tampoco tiene permisos.

0	—	sin permisos
1	-x	ejecución
2	-w-	escritura
3	-wx	escritura y ejecución
4	r—	lectura
5	r-x	lectura y ejecución
6	rw-	lectura y escritura
7	rwX	escritura y ejecución

- b) Ejemplo: *chmod a+rw archivo*

Tipo de usuario	u	Usuario
	g	Grupo
	o	Otros
Operación	+	Añade permiso
	-	Remueve permiso
	=	Asigna permiso
Tipo de permiso	r	(reading) lectura
	w	(writing) escritura
	x	(execution) ejecución

- f. **chown**: permite cambiar el usuario o grupo propietarios de un determinado archivo. El uso más directo para asignar la propiedad de un archivo a un usuario es: *chown usuario archivo*. Si además del usuario, se desea también modificar el grupo propietario del archivo, se emplea: *chown usuario:grupo archivo*.
- g. **mv**: Permite renombrar archivos o mover archivos entre directorios.
- a) *mv archivo_a_renombrar nuevo_nombre*
- b) *mv archivo_a_copiar ruta_destino*
- h. **cat**: Examina el contenido de un archivo. Ejemplo: *cat prueba.txt*

Finalmente, siempre será posible observar todas las funciones de un comando utilizando el comando *help*. Para ello debe colocar el comando, dejar un espacio, colocar dos guiones y la palabra *help*. Ejemplo: *cat - - help*.

3. Programación de scripts de BASH

3.1. Creación y ejecución de un programa de BASH

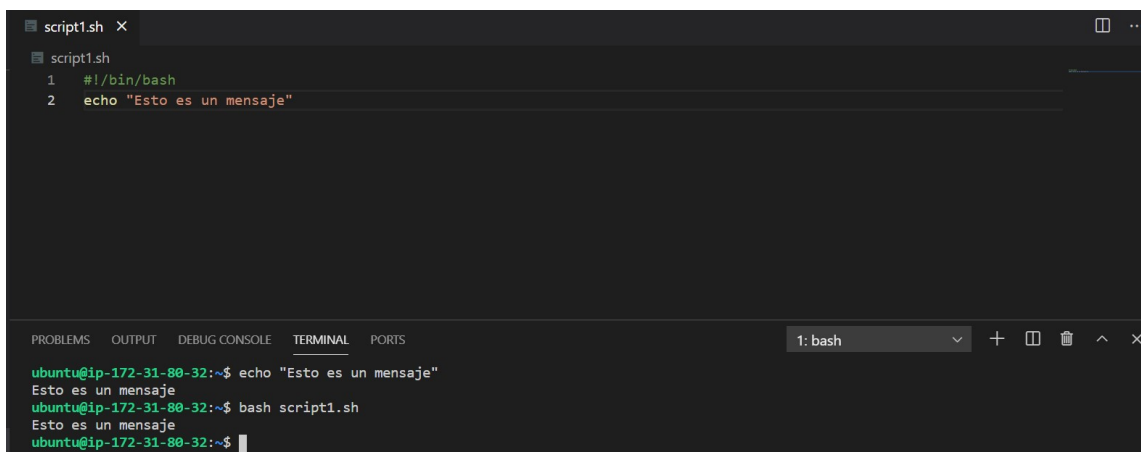
Los scripts de BASH son básicamente un conjunto de comandos secuenciales escritos que se pueden ejecutar a través del terminal al ejecutar el script. Por ejemplo, el comando **echo** realiza una réplica del argumento de entrada. Por tanto, si se quisiera mostrar en pantalla un mensaje se podría realizar lo mostrado en la Figura 1. Por otro lado, es posible



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ubuntu@ip-172-31-80-32:~$ echo "Esto es un mensaje"
Esto es un mensaje
ubuntu@ip-172-31-80-32:~$
```

Fig. 1: Ejemplo de uso del comando **echo**

crear un script que nos permita obtener el mismo resultado y posteriormente ser ejecutado a través del comando **bash**. Por ejemplo, se creará el script *script1.sh* y se ejecutará tal y como se muestra en la Figura 2.



```
script1.sh x
script1.sh
1  #!/bin/bash
2  echo "Esto es un mensaje"

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ubuntu@ip-172-31-80-32:~$ echo "Esto es un mensaje"
Esto es un mensaje
ubuntu@ip-172-31-80-32:~$ bash script1.sh
Esto es un mensaje
ubuntu@ip-172-31-80-32:~$
```

Fig. 2: Codificación del script de BASH usando nano.

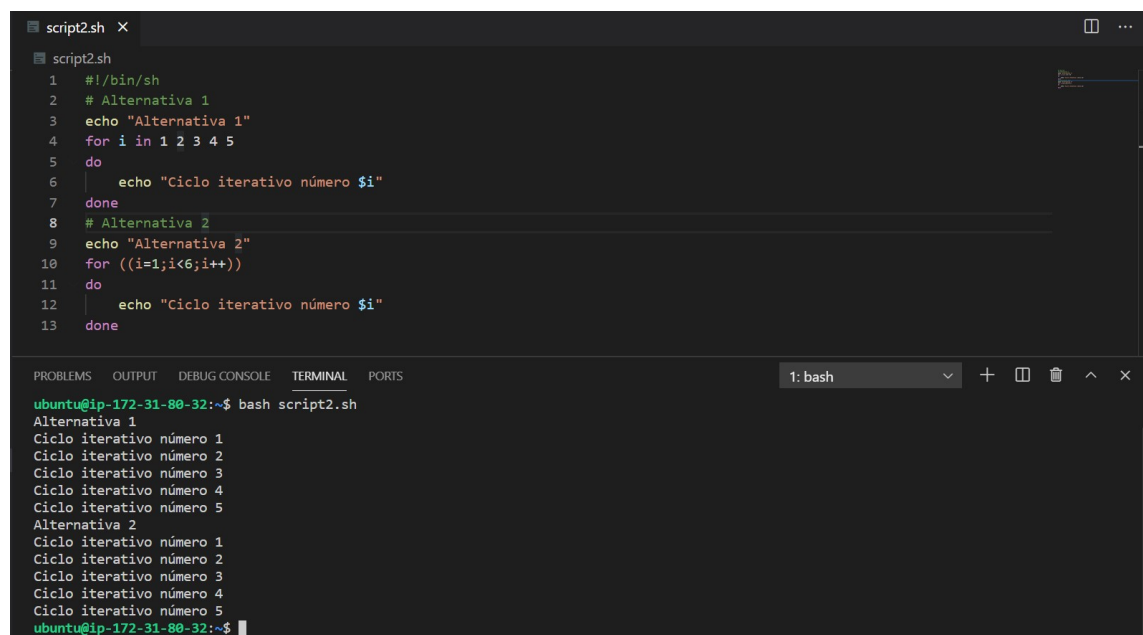
3.2. Herramientas de programación

La programación de un script de BASH es como cualquier lenguaje de programación. Solo dependerá de la sintaxis y de la lógica que hay detrás de lo que se quiere codificar. Por ello, en esta sección se verán herramientas y ejemplos útiles para lograr una ejecución de programa adecuada.

3.2.1. Ciclo iterativo for

Los ciclos iterativos son parte fundamental de cualquier lenguaje de programación. Siendo así, la programación de un script de BASH no es ajeno a este concepto. En esta

guía se presentan dos alternativas para su codificación (Figura 3). En la primera alternativa, se colocan los valores necesarios que recorrerá el ciclo `for`. No es necesario que sean consecutivos, tampoco que sean números, podrían ser letras. Es importante recalcar que la variable después del `for` (*i*) es quien tomará el valor de cada uno de estos elementos y que si se requiere hacer uso de ella, se puede hacer a través de `$i`. Así mismo, considerar que el ciclo inicia con el comando `do`. No olvidar cerrar el ciclo con el comando `done`. Por otro lado, la segunda alternativa recoge elementos de programación en C. Para ello, se coloca la operación entre doble paréntesis, y el compilador de BASH entenderá que se está usando como mecanismo auxiliar para que se interprete como lo haría el compilador de C. Al igual que la primera alternativa, también se debe abrir el loop con `do` y cerrar con `done`.



```

script2.sh
1  #!/bin/sh
2  # Alternativa 1
3  echo "Alternativa 1"
4  for i in 1 2 3 4 5
5  do
6      echo "Ciclo iterativo número $i"
7  done
8  # Alternativa 2
9  echo "Alternativa 2"
10 for ((i=1;i<6;i++))
11 do
12     echo "Ciclo iterativo número $i"
13 done

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
1: bash
ubuntu@ip-172-31-80-32:~$ bash script2.sh
Alternativa 1
Ciclo iterativo número 1
Ciclo iterativo número 2
Ciclo iterativo número 3
Ciclo iterativo número 4
Ciclo iterativo número 5
Alternativa 2
Ciclo iterativo número 1
Ciclo iterativo número 2
Ciclo iterativo número 3
Ciclo iterativo número 4
Ciclo iterativo número 5
ubuntu@ip-172-31-80-32:~$

```

Fig. 3: Ejemplo de las alternativas de ciclos iterativos con número específico de iteraciones.

3.2.2. Ejecución de líneas de comando para otros programas

En las Figuras 4 y 5 se ha codificado el producto de dos números en C² (`producto.c`) y Python³ (`producto.py`), respectivamente. Para ambos, los números se ingresan como argumentos de entrada desde el terminal.

En la Figura 6 se observa el código que permite llamar a ambas funciones. En el caso de C, se va a utilizar el compilador `gcc`.

²argv toma los argumentos de entrada del terminal, atoi cambia de string a int

³sys.argv cumple la misma función que argv en C

```
C producto.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char const *argv[])
5  {
6      int a = atoi(argv[1]);
7      int b = atoi(argv[2]);
8      int c = a*b;
9      printf("El producto es %d \n", c);
10     return 0;
11 }
```

Fig. 4: Código en C.

```
python3 producto.py > ...
1  import sys
2
3  a = int (sys.argv[1])
4  b = int (sys.argv[2])
5
6  c = a*b
7
8  print("El producto es ", c)
```

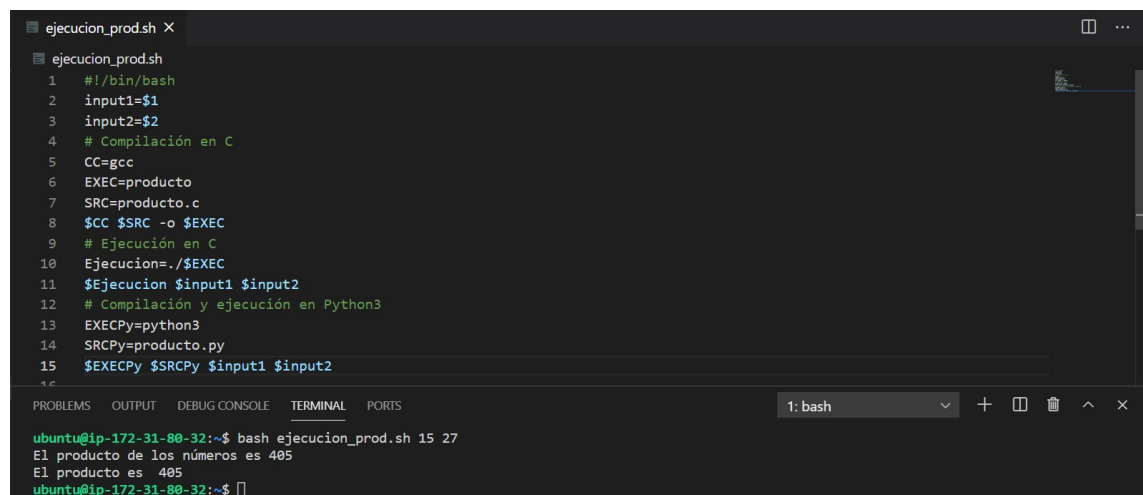
Fig. 5: Código en Python.

Explicación del llamado a C: Si se quisiera ejecutar el programa *producto.c* desde el terminal, se debería colocar lo siguiente: `gcc prod.c -o prod && ./prod 15 27`⁴. En este caso, 15 y 27 son números de prueba para realizar el producto ya que el programa hace el producto de los argumentos de entrada.

Explicación del llamado a Python 3: Si se quiere ejecutar el programa *prod.py* desde el terminal, se debería colocar lo siguiente: `python3 prod.py 15 27`. Notar que 15 y 27 siguen la misma lógica del programa en C.

Explicación del código de BASH: A modo de prueba, se colocará cada instrucción en una variable para validar la secuencialidad del programa de BASH. Los argumentos de entrada de BASH se pueden referenciar en estricto orden de llamado: Primer argumento: \$1, Segundo argumento: \$2, etc. Para nuestro ejemplo, se almacenarán el primer y segundo argumento en las variables `input1` e `input2`, respectivamente.

Para entender el potencial de BASH, se almacenarán los nombres de los comandos y programas de utilidad en variables. Por ello, se utilizarán las variables `CC`, `EXEC` y `SRC`, `EXECpy` y `SRCpy` para almacenar los comandos y programas de interés. Finalmente, se utilizarán los argumentos de entrada 15 y 27 para validar que el programa de BASH funciona.



```
ejecucion_prod.sh X
ejecucion_prod.sh
1  #!/bin/bash
2  input1=$1
3  input2=$2
4  # Compilación en C
5  CC=gcc
6  EXEC=producto
7  SRC=producto.c
8  $CC $SRC -o $EXEC
9  # Ejecución en C
10 Ejecucion=./$EXEC
11 $Ejecucion $input1 $input2
12 # Compilación y ejecución en Python3
13 EXECpy=python3
14 SRCpy=producto.py
15 $EXECpy $SRCpy $input1 $input2
16
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
1: bash
ubuntu@ip-172-31-80-32:~$ bash ejecucion_prod.sh 15 27
El producto de los números es 405
El producto es 405
ubuntu@ip-172-31-80-32:~$
```

Fig. 6: Código de BASH y ejecución con argumentos de entrada.

Este ejemplo, sirve para ilustrar que los scripts de BASH pueden almacenar tanto números como comandos o nombres de programas. Sin embargo, es importante recalcar que, al menos para este ejercicio, la forma más fácil de realizar esto sería la codificación directa (Figura 7).

⁴Cuando se coloca `&&` en el terminal, se ejecutan los comandos uno después de otro


```
ejecucion_prod_fast.sh
1  #!/bin/bash
2  # Compilación en C
3  gcc producto.c -o producto
4  # Ejecución en C
5  ./producto $1 $2
6  # Compilación y ejecución en C
7  python3 producto.py $1 $2

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

ubuntu@ip-172-31-80-32:~$ bash ejecucion_prod_fast.sh 15 27
El producto es 405
El producto es 405
ubuntu@ip-172-31-80-32:~$
```

Fig. 7: Código de BASH y ejecución con argumentos de entrada.

3.2.3. Medición de tiempo

Los sistemas computacionales tienen como principal motivación hacer más rápido las tareas encomendadas. Siendo así, es importante calcular el tiempo que toma una ejecución, un proceso, u otra actividad que realice nuestro sistema. El comando **time**. Dicho comando brinda la información de salida del tiempo real, usuario y de sistema. La información detallada se resume de la siguiente manera:

- Real:** Tiempo desde el inicio al final de la llamada. Incluye los segmentos de tiempo utilizados por otros procesos y el tiempo que el proceso pasa bloqueado (i.e. tiempo de espera de I/O)
- User:** Tiempo que el CPU gasta en el código en modo usuario (fuera del kernel) dentro del proceso. No cuenta para el tiempo que pasa bloqueado el proceso.
- Sys:** Tiempo que el CPU se pasa en el kernel dentro del proceso. Ejecuta el tiempo de CPU para llamadas al sistema dentro del kernel.

Para fines prácticos, se utilizará el tiempo arrojado por el valor real. A medida que se avance en el curso se explicarán más detalles de cómo usar las salidas user y sys. En la Figura 8, se puede observar un ejemplo de cómo usar el comando **time**. En la primera línea se observa el tiempo que crear el directorio *directorio* toma 0 minutos con 0.002 segundos. Para el segundo ejemplo, se puede observar que el tiempo para acceder al directorio *shell* se encuentra en un orden de magnitud menor a 10^{-4} , por lo cual no es visible. Finalmente, se utiliza el script de BASH *cbash.sh* para hallar su tiempo de ejecución. En este caso, se observa que el tiempo fue de 0 minutos con 0.069 segundos.

```
ubuntu@ip-172-31-87-186:~$ time mkdir directorio
real    0m0.002s
user    0m0.002s
sys     0m0.000s
ubuntu@ip-172-31-87-186:~$ time cd shell/
real    0m0.000s
user    0m0.000s
sys     0m0.000s
ubuntu@ip-172-31-87-186:~/shell$ time bash cbash.sh 15 27
El producto hallado desde C es 405
El producto hallado desde python es 405

real    0m0.069s
user    0m0.059s
sys     0m0.010s
ubuntu@ip-172-31-87-186:~/shell$
```

Fig. 8: Resultado en el terminal tiempo.

3.3. Ejemplo de aplicación

3.3.1. Problema

Codificar un script de bash basado en la distribución de directorios mostrados en la Figura 9 considerando los permisos de lectura, escritura y ejecución que se muestran al costado de cada directorio. Para ello, específicamente, se le pide lo siguiente:

- Utilizar el comando **tree** para validar la creación de carpetas.
- Utilizar el comando **ls -l** para validar los permisos de los directorios existentes en la raíz y dentro de la carpeta *lab*.
- Similar al inciso anterior, utilizar **ls -l lab/prog**, **ls -l lab/example**, **ls -l lab/data** para validar los permisos de cada carpeta. Comentar si es posible ver los permisos o existe algún impedimento. Justificar su respuesta.
- Utilizar el comando **cd** para ingresar a los directorios *prog*, *data*, *example* (en ese orden) ¿Es posible ingresar a todos los directorios? Justificar su respuesta.

3.3.2. Solución

Se creará el programa *script_tree.sh*. La codificación del programa se muestra en la Figura 10. El código básicamente se encarga de crear las carpetas de forma secuencial desde el directorio con mayor jerarquía. Esto tiene sentido ya que, por ejemplo, no sería

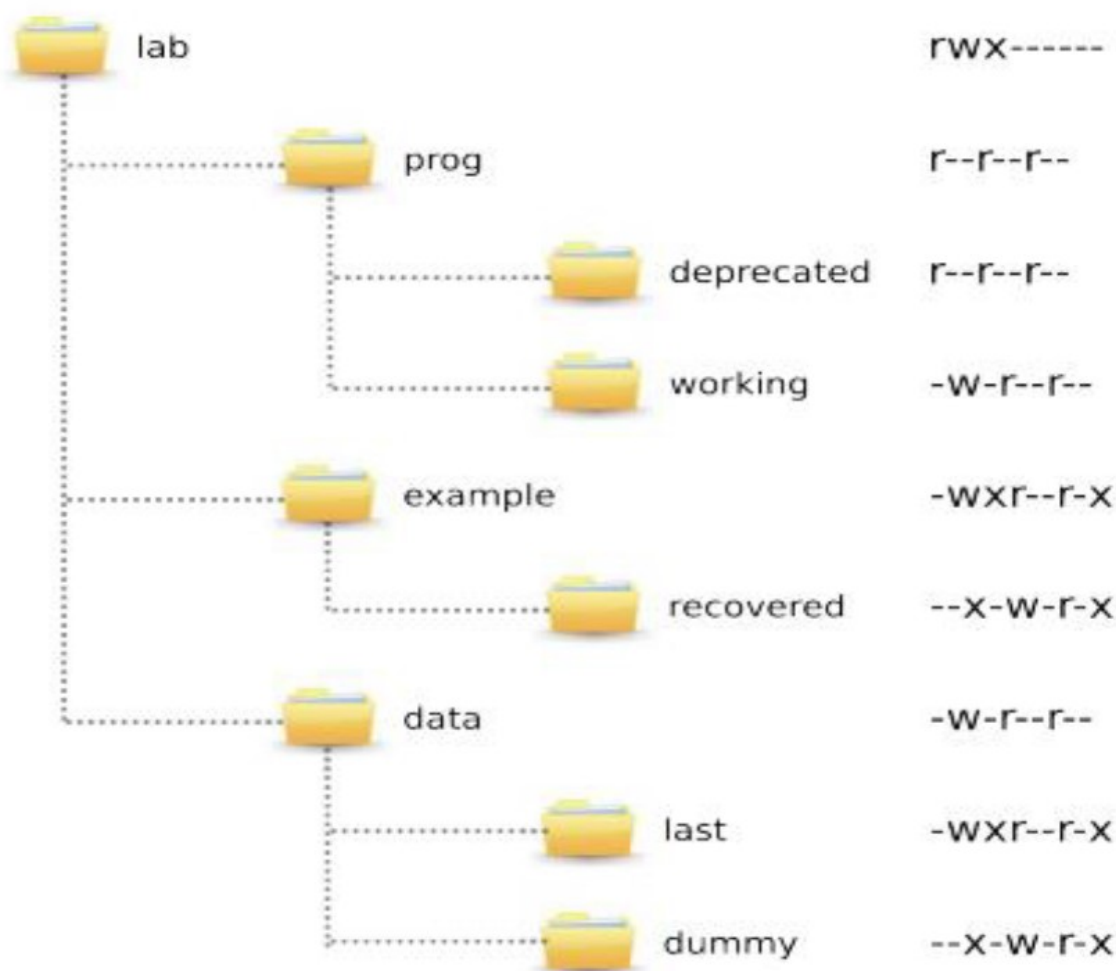


Fig. 9: Tree del folder resultante

posible utilizar **mkdir lab/prog/working**, sin antes haber utilizado **mkdir lab**. Por otro lado, la asignación de permisos empieza desde las carpetas con menor jerarquía. Esto también tiene sentido ya que si a la carpeta de mayor jerarquía no se le asigna ningún permiso no sería posible darle permisos a uno de menor jerarquía. Por ejemplo, si realizamos primero **chmod 444 lab**, ya no se podría realizar **chmod lab prog/working**. Sugerencia: Validar estas afirmaciones realizando los casos que se indican que no se podrían hacer.

Respuesta a.

En la Figura 11, se puede observar el resultado de ejecutar el programa *script_tree.sh*. Tal como se puede observar, al hacer uso del comando **tree**, se obtiene la ramificación desde el directorio raíz como se requería en la pregunta.

```
GNU nano 2.9.3 script_tree.sh

#!/bin/bash

mkdir lab
mkdir lab/prog
mkdir lab/prog/deprecated
mkdir lab/prog/working

mkdir lab/example
mkdir lab/example/recovered

mkdir lab/data
mkdir lab/data/last
mkdir lab/data/dummy

chmod 444 lab/prog/deprecated
chmod 244 lab/prog/working
chmod 444 lab/prog

chmod 125 lab/example/recovered
chmod 345 lab/example

chmod 345 lab/data/last
chmod 125 lab/data/dummy
chmod 244 lab/data

chmod 700 lab

^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 Linter  ^_ Go To Line
```

Fig. 10: Solución codificada en el script

Respuesta b.

En la Figura 11, se observa como al ejecutar el comando **ls -l** nos brinda los permisos para el directorio **lab** y para **script_tree.sh**. Esto considerando, que el script se encuentra en dicha directorio. Así mismo, al utilizar **ls -l lab**, se puede observar los permisos de las carpetas **data**, **example** y **prog**

Respuesta c.

En la Figura 13 se muestra la ejecución de lo requerido. Se puede observar que si bien es posible acceder a **prog**, no se observan los permisos de **working** y **deprecated**. Esto tiene sentido, ya que **prog** tiene permisos de lectura por lo que permite enlistar lo que hay dentro mediante de **ls**; sin embargo, no permite ejecutar la acción de brindar permisos. Para el caso de **example** y **data** simplemente no es posible brindar una lista ya que no tiene permisos de lectura.

```
ubuntu@ip-172-31-87-186:~$ bash script_tree.sh
ubuntu@ip-172-31-87-186:~$ tree

.
├── lab
│   ├── data
│   │   ├── dummy
│   │   └── last
│   ├── example
│   │   └── recovered
│   └── prog
│       ├── deprecated
│       └── working
└── script_tree.sh

9 directories, 1 file
```

Fig. 11: Ramificación de la carpeta raíz. Se puede observar los directorios y subdirectorios.

```
ubuntu@ip-172-31-87-186:~$ ls -l
total 8
drwx----- 5 ubuntu ubuntu 4096 Apr  2 17:15 lab
-rw-rw-r-- 1 ubuntu ubuntu  407 Apr  2 17:59 script_tree.sh
ubuntu@ip-172-31-87-186:~$ ls -l lab
total 12
d-w-r--r-- 4 ubuntu ubuntu 4096 Apr  2 17:15 data
d-wxr--r-x 3 ubuntu ubuntu 4096 Apr  2 17:15 example
dr--r--r-- 4 ubuntu ubuntu 4096 Apr  2 17:15 prog
```

Fig. 12: Revisión de permisos mediante el uso del comando `ls -l`

Respuesta d.

En la Figura 14 se observa que no es posible ingresar a `lab/prog` y tampoco `lab/data`. Esto es consistente con los permisos ya que no es posible ejecutar el comando `cd` porque ninguno tiene permisos de ejecución. Por otro lado, cuando se usa `cd` sí es posible ingresar a la carpeta debido a que sí se tiene permisos de ejecución.


```
ubuntu@ip-172-31-87-186:~$ ls -l lab/prog
ls: cannot access 'lab/prog/working': Permission denied
ls: cannot access 'lab/prog/deprecated': Permission denied
total 0
d????????? ? ? ? ?      ? deprecated
d????????? ? ? ? ?      ? working
ubuntu@ip-172-31-87-186:~$ ls -l lab/example
ls: cannot open directory 'lab/example': Permission denied
ubuntu@ip-172-31-87-186:~$ ls -l lab/data
ls: cannot open directory 'lab/data': Permission denied
```

Fig. 13: Revisión de permisos mediante el uso del comando **ls -l** para cada carpeta dentro de *lab*

```
ubuntu@ip-172-31-87-186:~$ cd lab/prog
-bash: cd: lab/prog: Permission denied
ubuntu@ip-172-31-87-186:~$ cd lab/data
-bash: cd: lab/data: Permission denied
ubuntu@ip-172-31-87-186:~$ cd lab/example
ubuntu@ip-172-31-87-186:~/lab/example$
```

Fig. 14: Uso del comando **cd** para ingresar a las carpetas.

4. Tarea

4.1. Problema 1

Se quiere observar la diferencia de tiempos entre un programa en C y otro en Python. Para ello, se debe codificar lo siguiente:

- Un programa en C que calcule la media armónica de un arreglo desde un número A hasta un número B, que incrementa de 1 en 1, ingresados por terminal.
- Un programa en Python que calcule la media armónica desde un número A hasta un número B, que incrementa de 1 en 1, ingresados por terminal.
- Codificar un programa de BASH que arroje el valor de los tiempos de ejecución. Sugerencia: *Incluir el comando time en el script de bash.*
- Responder qué programa ha sido más fácil de codificar y por qué
- Utilizando A=10 y B=20, responder qué programa ha sido más rápido. Además, responder si esto tiene sentido y porqué podría ser.
- Codificar un programa de BASH que llame al programa de BASH del inciso C y que mediante un ciclo iterativo se ejecuten los programas en C y Python considerando los siguientes pares para {A,B}: {100, 200}, {150,250} y {200,300}.
- De los 3 tiempos mostrados, ¿se mantienen los mismos números?

4.2. Problema 2

Cuando colocamos desde el terminal la frase:

```
$ echo 'Arqui es chevere' >> texto.txt
```

automáticamente, estamos creando un archivo llamado texto.txt que contiene la frase "Arqui es chevere". Esto se puede validar abriendo el archivo texto.txt con el editor nano. Siendo así, se le pide lo siguiente:

- Crear un script de BASH llamado *creador.sh* que cree y ejecute un programa en C que se llame *hola.c*. Este programa en C debe generar el mensaje "Hola amigos del curso de archi". **Nota:** Cuando requiera que echo genere comillas debe anteponer el signo de backslash (\).
- Editar el script de BASH *creador.sh* para que haga lo siguiente:
 - Después de ejecutar el programa *hola.c*, ejecute el comando **ls -l** y se puedan ver los permisos de todos los programas.
 - Que cambie los permisos de *hola.c* y de su ejecutable para que ya no se pueda leer, escribir o ejecutar.

- III. Que ejecute el comando **ls -l** y que valide que los permisos han sido cambiados.
- c. ¿Cuál es la diferencia entre denegar permisos de escritura y ejecución?
 - d. ¿Si se niega todos los permisos en el archivo *hola.c*, el compilador **gcc** aún podría generar el ejecutable? Justificar.
 - e. ¿Cuáles son los permisos mínimos que debería tener el archivo *hola.c* para que se pueda crear su ejecutable?

4.3. Problema 3

Se requiere ejecutar un programa que calcule si un número es primo. Para ello, se le pide lo siguiente:

- a. Codificar el programa en C **primo.c** el cual solo imprima el mensaje “**El número X es primo**”, siendo X un número ingresado por terminal. **Sugerencia: Utilizar el operador %**.
- b. Codificar un programa en Python3 **primo.py** el cual que solo imprima el mensaje “**El número X es primo**”, siendo X un número ingresado por terminal. **Sugerencia: Utilizar el operador %**.
- c. Codificar el programa de BASH **exp3.sh** que halle los números primos del 1 al 100 utilizando los programas de los incisos a y b. Considerar que debe generar un for-loop para cada uno. Calcular el tiempo de ejecución de cada uno **sin utilizar el comando time**. Nota: Para el programa en C, utilizar únicamente el ejecutable.
- d. Responder cuál ha sido más rápido y su posible justificación. A continuación, se le pide codificar el programa de BASH **exp3p.sh** el cual realice, de forma secuencial, las siguientes acciones:
 - e. Mostrar los permisos que tiene el script **exp3.sh**.
 - f. Crear la carpeta **exp3** y copie todos los archivos de dicha experiencia a la carpeta creada y que ingrese a dicha carpeta.
 - g. Dentro de la carpeta **exp3**, cambiar los permisos del script **exp3.sh** habilitando, al menos, permisos de lectura, escritura y ejecución de usuario.
 - h. Posteriormente, agregar la línea **./exp3.sh** en su código.
 - i. Después de lo colocado en el inciso anterior, retirar los permisos y ejecutar nuevamente **./exp3.sh**.
 - j. Finalmente, ejecutar el script **exp3p.sh**. Responder las siguientes preguntas, considerando lo que se encuentra en el **folder exp3**: ¿Qué ha sucedido con **exp3.sh** en



este script? ¿exp3.sh fue, en algún momento un programa ejecutable? ¿se puede editar exp3.sh?