

Elementos básicos en GNU/Linux

2

UNO DE LOS COMPONENTES EN TODO COMPUTADOR, es su sistema operativo (OS, por sus siglas en inglés) y las herramientas que brinda para realizar labores específicas. Los sistemas operativos que comunmente se utilizan en física y astronomía son GNU/Linux, Windows, y OS/X. En este libro nos dedicaremos a trabajar con GNU/Linux.

El desarrollo de este capítulo se realizará sobre una distribución Ubuntu de GNU/Linux – útil también para cualquier distribución basada en Debian¹ –, y se asumirá que el lector ya cuenta con alguna distribución GNU/Linux disponible, lo que puede ser a través de máquinas virtuales, instalación única de GNU/Linux, o con sistema operativo compartido. En la actualidad versiones de Windows 10 y 11 cuentan con *Windows Subsystem for Linux* (WSL) que da un soporte íntegro de un sistema GNU/Linux dentro de Windows.

2.1	La terminal	5
2.2	Comandos básicos	7
2.3	Tuberías y Redirección	9
2.4	Manejo de archivos	10
2.5	Edición de Archivos	12
2.6	Ejemplos BASH para terminales	16
2.7	BASH como lenguaje	24

1: Debian es una de las distribuciones más antiguas de GNU/Linux. El proyecto Debian se fundó en Agosto de 1993, más información puede encontrarse en la web oficial <http://www.debian.org>

2.1 La terminal

EXISTEN VARIADOS TIPOS DE TERMINALES EN GNU/Linux, que emulan la antigua terminal de Unix, sistema utilizado mucho en centros de investigación a mediados de los años 70. Estas terminales generan una interfaz directa – mediante instrucciones – entre el usuario y el computador. La mayoría de las terminales no utilizan el *mouse*, sino que se basan en instrucciones que son entregadas a través del teclado por el usuario.

Un terminal clásico presenta un *prompt* como el que se muestra a continuación,

```
1 username@machine:$
```

acá *username* corresponde al nombre del usuario que se encuentra utilizando el sistema, y *machine* corresponde al nombre de la máquina (o computadora) en la que se encuentra trabajando. Usualmente el símbolo \$ indica que el equipo esta a la espera de instrucciones. El **intérprete de comandos** – conocido como shell – conecta al usuario con el sistema operativo, en nuestro caso trabajaremos con el intérprete de comandos BASH.²

2: Existen variados shell para GNU/Linux, Los más comunes son BASH, ZSH, CSH, SH, entre otros. En la mayoría de los sistemas BASH viene instalado por *default*.

Los archivos y directorios que posee el/la usuario/a en el sistema, son almacenados en su directorio personal. Este directorio esta generalmente ubicado en `/home/username/`, en donde *username* es el nombre de el/la usuario/a. Es importante tener en cuenta el árbol de directorios dentro del sistema operativo, comprender las restricciones de el/la usuario/a, y saber donde son almacenados los distintos elementos de nuestro OS.

3: Similar a los formatos NTFS, FAT, y FAT32 que se observa para los discos en Microsoft Windows.

2.1.1 Estructura de directorios en GNU/Linux

Todo OS utiliza un disco para almacenar información, este disco tiene un formato y estructura definida, así como también restricciones dependiendo el tipo de usuario que desea acceder a la información del disco. En GNU/Linux el formato de un disco suele ser EXT4/EXT3/EXT2³, con EXT4 uno de los más utilizados hoy en día. El árbol de GNU/Linux – standard – tiene la forma que se observa en la figura 2.1. De este árbol es posible destacar características generales, entre ellas:

- ▶ `/` : Es el directorio *raíz*, el cual anida a todos los demás directorios del OS.
- ▶ `/bin` : Es el directorio que almacena los principales ejecutables del sistema.
- ▶ `/home` : Es el directorio donde se encuentran ficheros y directorios de cada uno/a de los/as usuarios/as del sistema.
- ▶ `/etc` : Ficheros con datos de la configuración del sistema.
- ▶ `/media` : Lugar donde se montan dispositivos externos en el sistema, tales como memorias usb, o discos ópticos.
- ▶ `/dev` : Almacena los dispositivos del equipo, como discos duros, tarjetas gráficas, etc.

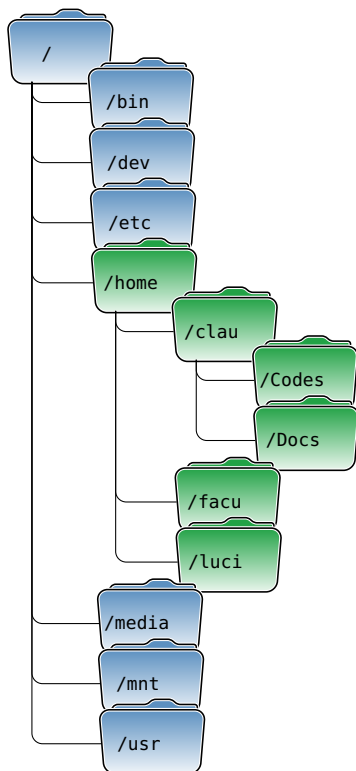


Figure 2.1: Estructura de los directorios en un sistema EXT4 en GNU/Linux. La imagen no muestra archivos, sólo directorios.

Un/Una usuario/a *standard* tiene acceso de escritura sólo a los datos que están almacenados en su directorio personal, y de lectura a gran parte del sistema – pero **no** a todo –. Por ejemplo, en este caso la usuaria *clau*, puede almacenar su información dentro del directorio `/home/clau/`. Es allí – principalmente – donde ella puede crear más directorios y archivos. En EXT4 los nombres de directorios y archivos son *case-sensitive*, es decir distingue entre mayúsculas y minúsculas, esto quiere decir que podemos tener dos archivos llamados `hola.dat` y `Hola.dat`, con lo que el sistema los reconocerá como ficheros completamente distintos. Adicionalmente es posible manejar extensiones y nombres de archivo libremente. Por ejemplo para que un fichero sea ejecutable, no se necesitan extensiones ni nombres especiales, puede ser ejecutado siempre y cuando tenga asignados **permisos** de ejecución ⁴.

Existe sólo una usuaria en GNU/Linux que tiene acceso a todo el sistema, su nombre es *root* (la raíz). Para la usuaria *root*, si bien puede acceder a todo, existen muchas cosas que se encuentran encriptadas, por ejemplo los *passwords* de los/las usuarios/as del sistema, por lo que *root* necesitará *desencriptar* ⁵ para acceder este tipo de información. Es por eso que los/las usuarios/as deben usar *passwords* que incorporen gran cantidad de caracteres, para que sea prácticamente imposible que alguien (con privilegios de tipo *root*) sea capaz de robar esta información. Existen muchas formas para mantener un sistema personal de forma segura [Clinton2020-oq], por si el/la lector/a desea profundizar en esto.

4: Los permisos de cada archivo, tienen 3 factores principales, lectura, escritura, y ejecución, que pueden ser ajustados con el comando `chmod`.

5: Tarea que puede ser imposible, dependiendo de las características de la palabra clave (*password*).

Ahora, para que el/la usuario/a pueda crear y administrar los directorios y archivos en su *home*, tiene un set grande de comandos pertenecientes al intérprete (BASH). Entre estos, existen algunos comandos muy útiles, entre los que destacamos `ls`, `pwd`, y `cd`. Estos comandos permiten listar el contenido de un directorio (`ls`), ver en que directorio estamos ubicados (`pwd`), y cambiarnos de directorio (`cd`). Para la usuaria *clau*, que ya

posee algunos directorios y archivos en el sistema, la ejecución de estos comandos luce algo como:

```

1 clau@machine:$ ls
2 Codes/ Docs/ archivo1.dat archivo2.dat
3 clau@machine:$ pwd
4 /home/clau
5 clau@machine:$ cd Codes
6 clau@machine:$ pwd
7 /home/clau/Codes/

```

Un comando interesante es `cd ~`, el que independiente del directorio en el que nos encontremos trabajando, nos llevará a nuestro `/home`. Existe un número muy grande de instrucciones para administrar nuestros archivos en GNU/Linux, una selección muy reducida de estas instrucciones se presentarán en la siguiente sección.

2.2 Comandos básicos

A DIFERENCIA DE OTROS OS, GNU/Linux se construyó pensando en un ambiente de terminales de texto, más que en un ambiente gráfico, es por eso que la administración completa de nuestros archivos puede ser construida mediante la combinación de comandos que nos permiten administrar el contenido de nuestro `home`, o directorio personal. Si bien todos los OS tienen esta facilidad, GNU/Linux fue pensado en una forma eficiente para llevar a cabo estas labores. Hoy en día muchos OS han incorporado mejores herramientas para el manejo de instrucciones mediante línea de comandos, como es el caso de PowerShell⁶ en Microsoft Windows 10 y 11.

6: Lanzada en el año 2006, y actualmente en su versión 7.0.

Los comandos más utilizados en una terminal pueden ser catalogados entre distintos tipos. Aquí se presentarán dos tipos principales: i) comandos relacionados a la administración de archivos, como mover, copiar, eliminar, etc; y ii) comandos que son utilidades, las que pueden ser de gran ayuda para analizar, cortar, separar, y ordenar el contenido de nuestros archivos.

2.2.1 Administración de archivos

La siguiente lista – muy reducida – presenta los **principales** comandos GNU/Linux para administrar nuestros archivos.

- **ls** : Lista el contenido del directorio en el cuál nos encontramos.
- **cp** : Copia un archivo en otra ubicación.
El formato es:
`cp archivo_origen.txt ubicacion/archivo_destino.txt`
- **mv** : Mueve el archivo a otra ubicación. También puede ser utilizado para renombrar archivos.
El formato es:
`mv archivo_origen.txt ubicacion/archivo_destino.txt`

- ▶ **mkdir** : Crea un directorio en la ubicación actual. Recuerde que los nombres deben ser elegidos de una forma ordenada, para la salud mental de la/el propia/o usuaria/o.
El formato es: `mkdir DirNuevo`
- ▶ **cd** : Cambio de Directorio. Nos cambiamos a un Directorio, desde el directorio en el cuál nos encontramos.
El formato es: `cd DirNuevo`
- ▶ **touch** : Genera un archivo vacío, o actualiza un existente.
El formato es: `touch new_file.dat`
- ▶ **rm** : Remueve un fichero o un directorio (con la opción `-r`).
El formato para borrar un archivo es: `rm archivo.dat`.
El formato para borrar un directorio es: `rm -r Directorio/`
- ▶ **tree** : Muestra un árbol de los Directorios y archivos, a partir del directorio actual. Se ejecuta simplemente con `tree`.
- ▶ **tail** : Muestra las últimas 10 líneas –opción por *default*– de un archivo de texto.
- ▶ **head** : Muestra las primeras 10 líneas –opción por *default*– de un archivo de texto.
- ▶ **find** : Realiza búsqueda de archivos. Posee variadas opciones de búsqueda.

2.2.2 Utilidades

7: Para muchas de las utilidades en GNU/Linux que utilizan un despliegue especial en la terminal, se puede salir de ellas presionando la tecla `q`, del inglés *quit*.

La siguiente lista – muy reducida – presenta algunas utilidades⁷ que posee una terminal básica de GNU/Linux.

- ▶ **man** : Es el acceso al manual para cualquier comando del sistema. Por ejemplo, puede utilizar `man ls`, para acceder al manual del comando `ls`.
- ▶ **cal** : Muestra el calendario del mes actual.
- ▶ **date** : Muestra la fecha y la hora en la terminal.
- ▶ **sudo** : Ejecuta el comando como administradora del sistema –root–.
- ▶ **apt** : Comando para administrar la instalación de software. Esto es sólo válido para distribuciones GNU/Linux basadas en Debian. Para aquellas basadas en RedHat, un análogo es el comando `yum`. Ejemplo: `sudo apt install tree`
- ▶ **git** : Sistema de control para desarrollo de software
- ▶ **cat** : Muestra el contenido de un fichero en la terminal
- ▶ **less** : Muestra el contenido de un fichero, paginado.
- ▶ **sort** : Ordena el contenido de un fichero, alfa o numérico.
- ▶ **alias** : Redefine o Genera nuevos comandos, que son un *alias* a combinaciones de comandos mas complejas.
- ▶ **tar** : Permite comprimir/descromprimir nuestros archivos en variados formatos.
- ▶ **top** : Muestre el estado actual del sistema. Existen versiones modernas como `htop`.
- ▶ **ssh** : Permite conectarse a un computador remoto.
- ▶ **scp** : Permite copiar desde/hacia un computador remoto.
- ▶ **lastlog** : Despliega la información de *logging* del usuario.
- ▶ **wc** : Muestra el número de líneas, palabras y caracteres de un archivo.
- ▶ **sed** : Editor para contenido de filas en tiempo real.

- ▶ **awk** : Lenguaje de programación, diseñado para el manejo de columnas en archivos.
- ▶ **echo** : Muestra un texto o variables en la terminal. Pruebe, por ejemplo: `echo "Hola mundo"`.

De esta lista básica de comandos GNU/Linux veremos en secciones posteriores algunos ejemplos particulares con `awk`, y `sed`. La ventaja de estos comandos, es que son en sí mismos lenguajes de programación, lo que nos ayudará a generar estructuras de análisis sobre los ficheros en lo que podríamos estar trabajando.

2.3 Tuberías y Redirección

LAS TUBERÍAS EN GNU/Linux SON UNA DE LAS GRANDES HERRAMIENTAS que posee el sistema. Su uso se enmarca en la filosofía de unir muchas herramientas pequeñas para generar herramientas más complejas. Un repaso histórico sobre esta idea de *pipes* en UNIX, puede ser visitada en [Kernighan2019-hd].

El sistema de tuberías se utiliza mediante el símbolo *pipe* `|`, y tiene como objetivo tomar la salida – o resultado – de un comando, y entregarlo como entrada para un nuevo comando.

Consideremos el siguiente ejemplo: acá mostramos – con el comando `ls` – los archivos con extensión `.dat`, y luego contamos el número total de archivos que hay con esa extensión, usando el comando `wc`⁸. Notemos que `wc` **no** cuenta las líneas que poseen los archivos `.dat`, sino que toma la salida del comando `ls`, y eso lo considera **un nuevo contenido** al que cuenta sus líneas.

8: Recordemos que este comando puede contar, líneas, caracteres, y/o palabras en un archivo.

```

1 clau@machine:$ ls -l *.dat
2 archivo1.dat
3 archivo2.dat
4 clau@machine:$ ls -l *.dat | wc -l
5 2
6 clau@machine:$
```

Así en una sola línea de comando, podemos contar cuantos ficheros de cierto tipo tenemos, y sin la necesidad de arrastrar un *mouse* por distintos lugares de la pantalla. Combinaciones de multiples *pipes* se pueden realizar para llevar a cabo análisis más complejos, que veremos en ejemplos más adelante.

Otro elemento práctico es la redirección de las salidas de nuestros comandos a nuevos archivos, o dispositivos. La redirección de un comando típicamente esta dado por el símbolo `>`, o `>>`, en donde escribe o añade respectivamente, contenido a un archivo o dispositivo. También existen redirecciones en sentido contrario, como `<`, pero no las cubriremos en esta sección.

Un ejemplo podría ser la búsqueda de todos los ficheros `.txt` de mi directorio personal usando el comando `find`, guardar esa información en un fichero llamado `mis.txt.dat`. Así, nuestro comando sería:

```
1 clau@machine:$ find . -iname "*.txt" > mistxt.dat
```

9: Recuerde que para finalizar la revisión de un fichero con el comando `less`, debe presionar la tecla `q`.

En el ejemplo anterior, una lista con todos los ficheros con extensión `.txt` es almacenada en `mistxt.dat`, y para ver la lista podemos usar el comando `less mistxt.dat`⁹.

Tanto las tuberías como las redirecciones pueden ser utilizadas en sistemas más complejos de `bash`. Podríamos por ejemplo, renombrar todos nuestros ficheros terminados en `.txt` en ficheros `.dat`, usando:

```
1 clau@machine:$ find . -iname "*.txt" | sed 's/\.txt//g' | sed 's/./mv &.txt &.dat/g'
2 mv archivo3.txt archivo3.dat
3 mv archivo4.txt archivo4.dat
4 clau@machine:$ find . -iname "*.txt" | sed 's/\.txt//g' | sed 's/./mv &.txt &.dat/g'
   ↪ | sh
```

10: No lo intente, eliminará sus archivos de forma permanente.

11: Existen algunas distribuciones GNU/Linux que generan una estructura básica de directorios en el home del usuario

en donde se observan tres y cuatro comandos respectivamente separados por `|`. Si bien es un comando muy complejo que se comprenderá mejor al final del capítulo, podemos comentar sus partes esenciales separadas por `|`. En el primer comando se buscan los ficheros de interés (con `find`), luego con el segundo comando se elimina su extensión (`sed`) y finalmente con el tercer comando (`sed`) se escribe a partir de esta información un texto –que corresponde a comandos nuevos– para cada uno de los archivos. Así se consigue un *set* de líneas de comandos que deseamos ejecutar, pero estas no se ejecutarán hasta que el cuarto y último comando `sh`, sea incluido. Al final del capítulo se revisarán algunas combinaciones útiles en `bash`, con el uso de tuberías.

Antes de continuar con comandos y utilidades avanzadas de BASH, revisaremos el manejo de archivos dentro de nuestro sistema, mediante el uso de las herramientas básicas.

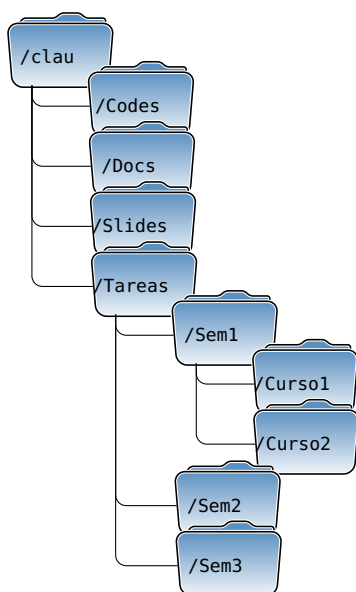


Figure 2.2: Estructura de los directorios de un usuario.

2.4 Manejo de archivos

GENERAR UNA ESTRUCTURA ORDENADA DE NUESTROS ARCHIVOS es fundamental para poder acceder a ellos de forma eficiente, no sobrecargar la cantidad de ficheros, y evitar catástrofes como la pérdida de archivos, el borrado accidental, y la sobre-escritura, entre otros. En GNU/Linux, la mayoría de los comandos no solicitan confirmación, por lo que combinaciones simples como `rm -fr *`¹⁰ pueden simplemente **destruir todos sus archivos**. Por eso siempre hay que pensar fríamente las instrucciones de usuario/a que involucren comandos como `rm`, y `mv`, entre otros.

A continuación revisaremos, brevemente, la forma en que los directorios son creados y con esto la administración de archivos. Lo primero es considerar cual es la estructura de directorios que deseamos construir¹¹ inicialmente –la que somos libres de modificar–, para ello consideremos como ejemplo, un set de directorios como el que se muestra en la figura 2.2, para la usuaria `clau`. Con esta información podemos construir los directorios fácilmente con una secuencia de comandos `mkdir`.

```

1 clau@machine:~$ ls
2 clau@machine:~$ mkdir Codes; mkdir Docs; mkdir Slides
3 clau@machine:~$ mkdir Tareas
4 clau@machine:~$ cd Tareas
5 clau@machine:~/Tareas$ mkdir -p Sem1/Curso1
6 clau@machine:~/Tareas$ mkdir Sem1/Curso2
7 clau@machine:~/Tareas$ mkdir Sem{2..3}
8 clau@machine:~/Tareas$ cd ~
9 clau@machine:~$
10 clau@machine:~$ ls
11 Codes/ Docs/ Slides/ Tareas/

```

De acá podemos desglosar: Primero, es posible separar distintos comandos de bash, en una sola línea, utilizando el símbolo `;`. Además es posible generar la existencia de un Directorio y Subdirectorio a la vez, utilizando la opción `-p` en el comando `mkdir`. Es decir, si el Directorio no existe, entonces lo crea. Por último destacar el uso de símbolos para la creación de múltiples directorios de forma simultánea, acá es posible crear dos directorios, `Sem2` y `Sem3`, utilizando simplemente `mkdir Sem{2..3}`¹², es decir bash comprende de inmediato lo indicado y numera desde 2 a 3 la creación de directorios.

12: Pruebe combinaciones como `mkdir Sem{01..15}`

Cuando deseamos crear archivos que contengan datos, códigos, y/u otra información, podemos utilizar un editor de texto para terminal, en donde podemos editar/crear este tipo de ficheros. Sin embargo también es posible crear ficheros vacíos de una manera rápida, para poder observar o hacer pruebas con los permisos de lectura/escritura en el directorio en que nos encontramos. Observe las siguientes instrucciones de bash:

```

1 clau@machine:$ ls
2 Codes/ Docs/ Slides/ Tareas/
3 clau@machine:$ touch datos.txt
4 clau@machine:$ ls
5 Codes/ Docs/ Slides/ Tareas/ datos.txt
6 clau@machine:$ touch /nuevo.txt
7 touch: cannot touch '/nuevo.txt': Permission denied

```

Como se puede observar el comando `touch`, actualiza o crea un fichero, lo que puede ser realizado en cualquier ubicación en la cuál el usuario tenga los permisos adecuados, como se observa en los comandos anteriores, la usuaria no puede escribir en el directorio raíz `/` del sistema, recibiendo un error de *Permission denied*, permiso denegado.

Para eliminar los archivos y directorios, existen comandos especializados, estos son `rm` y `rmdir` para eliminar archivos y directorios respectivamente. Muchas utilidades tienen opciones adicionales, por ejemplo en la eliminación de archivos tenemos la opción `-i`, como se muestra a continuación


```

1 clau@machine:$ ls
2 clau@machine:$ ls
3 Codes/ Docs/ Slides/ Tareas/ datos.txt
4 clau@machine:$ rm -i datos.txt
5 rm: remove regular empty file 'datos.txt'?

```

Acá podemos observar que el sistema reacciona de forma *interactiva* (-i), lo que implica que se nos consulte si estamos seguros sobre la eliminación de este archivo, a la que podemos contestar con y/n (yes/no). Si estamos seguro de nuestros comandos, podemos evitar la opción -i. En nuestro bash shell podemos definir un alias si queremos que nuestro comando se comporte siempre de alguna forma particular, o bien definir nuestros propios comandos. Por ejemplo:

```

1 clau@machine:$ alias rm='rm -i'
2 clau@machine:$ ls
3 Codes/ Docs/ Slides/ Tareas/ datos.txt
4 clau@machine:$ rm datos.txt
5 rm: remove regular empty file 'datos.txt'?

```

Ahora bien, para una verdadera edición de archivos, debemos utilizar un editor de archivos en la terminal. Existen en la actualidad muchos editores disponibles para una terminal, entre ellos se destacan: **vim**, **nano**, **emacs**, **jed**, **vi**, **nice**, **neovim**, entre muchos otros. Acá daremos una revisión rápida a **vim**, ya que es un editor de texto que se encuentra ampliamente disponible en los sistemas GNU/Linux, y si bien tiene una curva de aprendizaje más elevada que el resto de los editores, sus capacidades son muy amplias.

2.5 Edición de Archivos

A LA HORA DE ESCRIBIR CONTENIDO EN UN ARCHIVO debemos contar con un editor que nos permita, de forma rápida y simple, modificar o crear archivos. En la actualidad existen muchos editores de texto para terminales GNU/Linux sin embargo existen algunos que mantienen su *supremacía* en los terminales, y la han mantenido por mucho tiempo, entre ellos se encuentran **vi**, **vim**, **emacs**, y **nano** principalmente.

En esta sección veremos **vim** como nuestro editor predeterminado, motivados por su versatilidad, y amplio uso en programación y en clusters de cómputo¹³. Es, de hecho, un editor creado por programadores, para programadores.

13: Un clúster de cómputo corresponde a un arreglo grande de muchos computadores utilizados para distintas problemáticas, como en investigación, bases de datos, mercado bursátil, entre otros.

2.5.1 vim

14: Si no lo encuentra, puede instalarlo con: `sudo apt install vim`.

El editor **vim**, puede ser invocado en un terminal de texto, simplemente ejecutando el comando **vim**¹⁴ seguido, opcionalmente, por el nombre del archivo que deseamos editar. Si el fichero no existe, entonces **vim** lo creará una vez guardado; y si el fichero existe, entonces **vim** lo editará. Para todos estos casos es importante notar que se debe contar con los

Una vez en el modo COMANDO, existen diversos *comandos* o *instrucciones* que se pueden realizar, mediante combinación de teclas hasta teclas simples. Por ejemplo presionar la letra **u** para deshacer algún cambio. Hasta cosas más complejas, en donde un set de instrucciones debe darse para que vim ejecute lo que le ordenamos, en estos casos los comandos suelen iniciar con la tecla **:**. Por ejemplo para guardar el fichero, primero ingresamos al modo comando (**Esc**) y luego tipeamos **:w** (note que viene del inglés, *write*). Y si queremos grabar y salir al terminal, ejecutamos **:wq** (*write and quit*). Existe en la actualidad una gran cantidad de libros y guías para el uso de vim¹⁵. La mejor forma de aprender a utilizar este, o algún otro editor, es utilizarlo de forma continua para nuestro trabajo.

15: Puede incluso aprender vim usando tutores online como <https://www.openvim.com/tutorial.html>

2.5.2 Tips básicos de VIM

A continuación mostramos los elementos básicos para el uso de vim para un manejo adecuado en edición de ficheros.

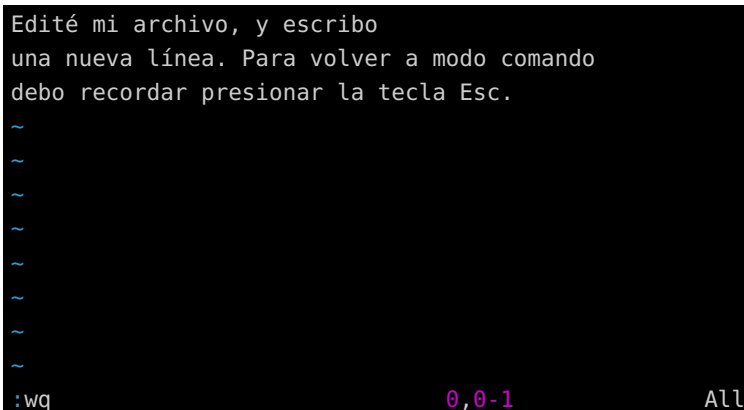
1. Abrir / Guardar / Salir

Para abrir archivos existentes, o nuevos

```
1 clau@machine:$ vim archivo.txt
```

En donde `archivo.txt` es el archivo que deseamos editar, si el archivo no existe entonces será creado, si el archivo existe entonces se editará. Recuerde que podemos usar la extensión que deseamos, por ejemplo `txt` para archivos de texto simple, `dat` para archivos con datos, o `py` para archivos de programas escritos en Python.

Una vez que editamos nuestro archivo – recuerde que para modo edición debe presionar **i** –, y realizamos todos los cambios deseados, entonces podemos salir y guardar, llendo a modo comando – presionando la tecla **Esc** –, y ejecutando en vim: **:wq**.



```
Edité mi archivo, y escribo
una nueva línea. Para volver a modo comando
debo recordar presionar la tecla Esc.
~
~
~
~
~
~
~
~
:wq                                0,0-1      All
```

Luego de presionar **Enter**, volvemos al terminal.

Para los casos de guardar, no salir, salir sin guardar y otras se pueden usar – en modo comando –:

- ▶ **:w** → Grabar y continuar en vim
- ▶ **:q!** → Salir sin grabar
- ▶ **:e nuevo.dat** → Editar un nuevo fichero llamado `nuevo.dat`

2. **Borrar / Deshacer / Saltar a línea** Dentro del modo edición, uno puede borrar simplemente como un editor de texto normal, así como moverse entre las líneas y caracteres del texto. Por lo que la lista de *tips* que se muestra a continuación, operan únicamente sobre el **modo COMANDO**.

- ▶ `dd` → Borra la línea completa sobre la cual se encuentra el cursor.
- ▶ `d$` → Borra desde la ubicación del cursor hasta el final de la línea.
- ▶ `dw` → Borra la palabra desde donde se encuentra el cursor.
- ▶ `x` → Borra el carácter donde se encuentra el cursor.
- ▶ `[N]dd` → Borra las siguientes `N` líneas desde donde esta el cursor.
- ▶ `[N]gg` → Nos movemos a la línea `N`, por ejemplo `5gg`, nos lleva el cursor a la línea 5.
- ▶ `u` → Deshace la última acción realizada.
- ▶ `Ctrl-r` → Rehace, en caso de haber deshecho una acción por error.

3. **Búsqueda / Reemplazo / Copiar y Pegar**

Para búsqueda y reemplazo utilizaremos el modo **COMANDO**. Por otro lado para copiar y pegar, accederemos a un modo especial, el modo **VISUAL**. Revisemos en primer lugar algunos comandos útiles de búsqueda y reemplazo:

- ▶ `:/WORD` → Buscamos la palabra `WORD` en el archivo, para continuar la búsqueda presionamos la tecla `n` (next), para las ocurrencias previas usamos `N`.
- ▶ `:%s/search_term/replace_term/g` → Busca `search_term` y reemplaza por `replace_term`. La `g` al final de la instrucción indica que debe hacerlo de forma Global en el documento, sin `g` solo reemplazará en la primera ocurrencia.
- ▶ `:%s/search_term/replace_term/gc` → Reemplaza `search_term` por `replace_term`. Acá `gc` al final de la instrucción indica que debe hacerlo de forma Global en el documento y confirmar cada caso. En este caso uno debe indicar mediante `y/n` si debe o no hacer el reemplazo en el archivo.
- ▶ `:1,10s/ubuntu/debian/g` → En este caso realiza un reemplazo de la palabra `ubuntu`, por `debian` entre solamente las líneas 1 y 10 del fichero.

Para los casos de copiar/pegar/cortar es preferible acceder a un modo especializado de [vim](#), el modo **VISUAL**. Al modo **VISUAL** se puede acceder desde el modo **COMANDO** y presionando la tecla `v`. Una vez en modo visual usted podrá moverse con las flechas libremente para seleccionar texto en el archivo.

Una vez seleccionado el texto, usted podrá :

- ▶ `y` → *Yank*, o copiar el texto.
- ▶ `x`, o `d` → Borrar, o cortar el texto.
- ▶ `p` → Pegar el texto previamente cortado o pegado.

Estos son elementos básicos del modo **VISUAL**, las capacidades son mucho mayores, por ejemplo copiar muchas líneas similares que sólo se

16: Por ejemplo, puede visitar el sitio web: <https://dev.to/igcredible/mastering-visual-mode-in-vim-15pl>

17: Por ejemplo: <https://www.youtube.com/watch?v=TmNa4y-K5Z8>

diferencian en un número, donde es posible además el automatizar el proceso ¹⁶.

Si bien lo revisado hasta ahora son elementos muy básicos en el uso de `vim`, es recomendable que el usuario visite variada documentación que existe hoy disponible *online*, para aprender mucho más de este poderoso editor ¹⁷

Ahora que ya conocemos lo básico para el manejo y edición de archivos podemos ir a trabajar con comandos más especializados de BASH en nuestro terminal de trabajo.

2.6 Ejemplos BASH para terminales

A CONTINUACIÓN SE PRESENTAN ALGUNOS EJEMPLOS AVANZADOS EN TERMINALES que pueden ser útiles a la hora de trabajar con grandes cantidades de archivos, o con archivos que poseen gran cantidad de contenido que deseamos filtrar. Estos ejemplos no se enfocan a ningún tópico en particular y su finalidad es servir como guía para resolver problemáticas específicas, que pueden servir como base para otros problemas.

2.6.1 Descargando ficheros online

El primer paso para revisar algunos de los ejemplos es la utilización de ficheros adicionales, muchos de estos ficheros pueden ser descargados online. Pero para evitar las descargas online accediendo a sitios web mediante el uso de *browsers*, la terminal cuenta con la herramienta `wget` para descargar ficheros directamente en un terminal desde cualquier dirección url.

Como ejemplo, podemos descargar el fichero de prueba ubicado en <https://gitarra.cl/lectures/modcomp/-/raw/main/sample.dat>.

Para ello basta con ejecutar en nuestro terminal

```
1 clau@machine:$ mkdir prueba
2 clau@machine:$ cd prueba
3 clau@machine:~/prueba$ wget https://gitarra.cl/lectures/modcomp/-/raw/main/sample.dat
4 ...
5 clau@machine:~/prueba$ ls
6 sample.dat
```

Ahora podemos inspeccionar nuestro fichero descargado con comandos tales como `less`, `cat`, o `vim` por ejemplo.

```
1 clau@machine:~/prueba$ cat sample.dat
2 Este fichero
3 es un fichero de ejemplo
4 que solo contiene texto.
5 clau@machine:~/prueba$
```

2.6.2 Descomprimir/Comprimir archivos o directorios

Tanto la compresión como la descompresión de ficheros es importante para poder transmitir grandes cantidades de datos mediante una red, ya que nos ayuda a evitar fallas debido a la estabilidad de la conexión y adicionalmente nos ayuda a acelerar el proceso de transferencia.

Es por esto que comprimir y descomprimir ficheros es importante dentro de un ambiente de trabajo de GNU/Linux. Uno de los comandos principales con soporte de compresión en los sistemas GNU/Linux es `tar`. Si bien el formato `tar` se refiere a un formato ampliamente utilizado en entornos UNIX, este formato sólo se encarga de juntar archivos, no de comprimirlos. Pese a esto el comando `tar` en GNU/Linux, tiene soporte para compresión en variados formatos hoy en día.

Entre las extensiones típicas utilizados en `tar` se encuentran

- ▶ `tar.gz` o `.tgz` : Archivo tar comprimido utilizando GNU-Zip.
- ▶ `tar.bz` o `.tbz` : Archivo tar comprimido utilizando GNU-BZip2.
- ▶ `tar.xz` : Archivo tar comprimido utilizando XZ.

Consideremos comprimir un directorio, para ello construyamos nuestro directorio de pruebas, y generemos muchos archivos en él.

```
1 clau@machine:$ cd ~
2 clau@machine:$ mkdir to_compress
3 clau@machine:$ cd to_compress
4 clau@machine:~/to_compress$ touch file{0..100}.txt
5 clau@machine:~/to_compress$ cd ../
6 clau@machine:$
```

Ahora, para comprimir este directorio, vamos a utilizar el comando:

```
1 clau@machine:$ tar -cvzf to_compress.tar.gz
   ↪ to_compress/
```

Las opciones asociadas al comando `tar`, son: `c` que indica la creación; `v` que indica modo verbose, es decir que muestre información sobre el proceso en pantalla; `z` que indica el modo de compresión en este caso GNU-Zip; y `f` que es la opción de nombre de archivo que se utilizará de salida, en este caso un fichero llamado `to_compress.tar.gz`.

Así, para comprimir con otros formatos sería

- ▶ `tar -cvf to_compress.tar to_compress` → Tar de archivos, sin compresión
- ▶ `tar -cvjf to_compress.tar.bz to_compress` → Compresión de archivos usando BZIP.
- ▶ `tar -cvJf to_compress.tar.xz to_compress` → Compresión de archivos usando XZ.

Mayor información actualizada siempre se puede encontrar en el manual del comando, al que puede acceder con `man tar`. O también en distintos sitios online, con ejemplos y casos de uso.

Para la descromprensión de archivos comprimidos, directamente en nuestra terminal, basta con el reemplazo de la opción `c` –asociada a la creación– de nuestro `tar`, por la opción `x` que está relacionada a la extracción del contenido. Por ejemplo para descromprimir el fichero `sample.tar.gz`, que puede descargar desde https://gitarra.cl/lectures/modcomp/-/raw/main/sample_data.tar.gz, lo hacemos con:

```
1 clau@machine:$ wget https://gitarra.cl/lectures/modcomp/-/raw/main/sample_data.tar.gz
2 clau@machine:$ tar -xvzf sample_data.tar.gz
```

Una opción interesante es la opción `t`, que muestra el contenido de un fichero sin necesidad de descromprimirlo. Por ejemplo para mostrar el contenido de un archivo, y paginarlo con `less`, debemos ejecutar:

```
1 clau@machine:$ tar -tvf sample_data.tar.gz | less
```

Un buen sitio con excelente documentación sobre ejemplos de `tar` se encuentra en el sitio <https://www.tecmint.com/18-tar-command-examples-in-linux/>. O simplemente con el manual de GNU/Linux, usando `man tar`.

2.6.3 Renombrando Archivos

Consideremos ahora un ejemplo un poco más complejo. Imaginemos que tenemos muchos archivos llamados `archivo01.txt`, `archivo02.txt`, ..., `archivo99.txt`. Y vamos a asumir que necesitamos transformarlos a otro tipo de archivos, por ejemplo podríamos simplemente renombrarlos. Ejecutar 99 comandos para renombrar cada uno de estos archivos podría ser tedioso, por lo que combinaremos el comando `find` para encontrar los archivos sobre los que deseamos trabajar, el comando `sed` para reemplazar texto – o editar – y generar estos 99 comandos. Un directorio con 100 ficheros como ejemplo, puede ser descargado en https://gitarra.cl/lectures/modcomp/-/raw/main/sample_100_files.tar.gz. A continuación se muestra la forma de generar estos 99 comandos:

```
1 clau@machine:$ wget https://gitarra.cl/lectures/modcomp/-/raw/main/sample_100_files.tar.gz
2 ...
3 clau@machine:$ tar -xvzf sample_100_files.tar.gz
4 ...
5 clau@machine:$ cd sample_files/
6 clau@machine:~/sample_files$ find . -iname "file???.txt" | sed 's/\.\.txt//g' | sed 's/././mv &.txt &.dat/g'
7 mv file01.txt file01.dat
8 mv file02.txt file02.dat
9 ...
10 mv file99.txt file99.dat
11 clau@machine:~/sample_files$ find . -iname "file???.txt" | sed 's/\.\.txt//g' | sed 's/././mv &.txt &.dat/g' |
   ↪ sh
```

Tenemos 3 partes fundamentales en el proceso, separadas por `pipe` `|`:

- `find`: Realiza una búsqueda en el directorio actual (`.`), busca los archivos sin importar si tienen mayúsculas o minúsculas (`-iname`), y selecciona aquellos archivos llamados `file???.txt`, en donde `??` representa dos caracteres cualesquiera.

- `sed` : El primer comando `sed` toma el texto `.txt` y lo reemplaza por nada (i.e. lo borra). Note que lo hace de forma **global**, es decir para todas las ocurrencias.
- `sed` : El segundo comando `sed` toma cada una de las líneas (`.*`), y las reemplaza por el texto `mv &.txt &.dat`, acá el valor de `&` corresponde a cada una de las líneas originales.

Estos tres comandos generan una lista de líneas que contienen instrucciones para cada uno de los 99 casos de interés: `mv file01.txt file01.dat ... mv file99.txt file99.dat`. Estas líneas son solo desplegadas en pantalla, **no son ejecutadas**. Para ejecutar cada uno de estos comandos, se debe indicar que la salida sea ejecutada por la shell del sistema usando para ello la instrucción final `| sh`, lo que ejecutará cada una de las líneas generadas previamente.

Si bien, existen comandos específicos que se pueden utilizar para renombrar archivos, comprender la lógica del uso de tuberías *–pipes–* `|` es importante para poder desarrollar comandos mucho más complejos en el futuro. Hay que recordar siempre la idea de unir pequeñas herramientas para construir herramientas más complejas.

2.6.4 Filtrado por columnas

Una de las herramientas potentes para el uso de análisis por columnas en GNU/Linux es `awk`. El comando `awk` es por sí un lenguaje de programación, y muchos análisis pueden ser desarrollados directamente con `awk`, en esta sección se cubrirán sólo cosas básicas para el uso de `awk`, tanto por si mismo, como combinado con otros comandos.

El comando `awk` es un programa para el manejo de columnas principalmente, sin embargo, su evolución y fácil programación lo hacen el programa ideal para el manejo de archivos. Es idiomático, y conviene pensarlo siempre así.

Por ejemplo si queremos imprimir sólo las líneas que cumplen cierta condición en un archivo, uno deberá entregar estas opciones al comando `awk` como :

```
1 clau@machine:$ awk 'conditions {actions}' file
```

De esta forma cuando un patrón se cumple en alguna fila, podemos verlo con :

```
1 clau@machine:$ awk '/patron/ {print $0}' file
```

Adicionalmente `awk` posee valores por defecto, la principal acción por defecto es `print $0`, por lo que el comando anterior puede reducirse a:

```
1 clau@machine:$ awk '/patron/' file
```

Esta es una lista con algunas cosas útiles de `awk`.

- ▶ `awk 'NR % 6' → Imprime todo excepto líneas divisibles por 6.`
- ▶ `awk 'NR > 5' → Imprime a partir de la línea 6.`
- ▶ `awk '$2 == "foo"' → Imprime líneas cuyo segundo campo es foo.`
- ▶ `awk 'NF >= 6' → Imprime líneas con 6 o más campos.`
- ▶ `awk '/foo/ && /bar/' → Imprime líneas que contengan /foo/ and /bar/.`
- ▶ `awk '/foo/ && !/bar/' → Imprime líneas que contengan /foo/ y no /bar/.`
- ▶ `awk '/foo/ || /bar/' → Imprime líneas que contengan /foo/ o /bar/.`
- ▶ `awk '/foo/, /bar/' → Imprime desde línea con /foo/ hasta línea con /bar/.`
- ▶ `awk 'NF' → Imprime sólo líneas no vacías.`
- ▶ `awk 'NF--' → Remueve el último campo e imprime la línea.`
- ▶ `awk '$0 = NR' '$0' → Antepone número de línea.`

Como se puede observar de la lista anterior en la mayoría de los casos estamos tomando la *action* por defecto que corresponde a `print $0`, es decir imprimir la línea completa cuando la condición se cumple. Además existen valores propios de `awk`, como los son: i) `NR` que corresponde al número de *records*, es decir el número de línea que se está analizando; ii) `NF` Número de campos (*fields*) esto corresponde a los campos – o columnas – que posee una línea. Los contenidos de un *record* pueden accederse mediante los campos numerando de la forma `$1`, `$2`, ..., y `$N`, que corresponde a la columna uno, dos, y hasta la *n*-ésima columna.

A continuación veremos algunos ejemplos básicos sobre columnas utilizando `awk`.

2.6.5 Operaciones básicas en columnas

Haciendo uso de los 100 ficheros descargados y renombrados previamente, realizaremos algunos análisis sobre estos mediante el uso del comando `awk`. Es recomendable que antes de comenzar inspecciones los archivos, por ejemplo con `less`, o `vim` para ver como están formados estos archivos¹⁸, cuantas columnas tienen, que tipo de datos poseen, etc. A continuación se muestran algunos casos de interés. Pruebe los comandos a continuación sobre estos ficheros, para comprender mejor las ventajas de `awk`.

Mostrar sólo la columna 1 de un archivo de datos.

```

1 clau@machine:~/sample_files$ awk '{print $1}'
  ↪ file01.dat
2 ....
3 340
4 886
5 clau@machine:~/sample_files$
```

Mostrar la columna 2 y 5, separadas por espacios.

18: Puede utilizar por ejemplo `less file01.dat`.

```

1 clau@machine:~/sample_files$ awk '{print $2 " " $5}'
  ↪ file02.dat
2 ...
3 947 454
4 496 853
5 clau@machine:~/sample_files$

```

Multiplicar dos columnas, la columna 1 y la 2, e imprimir eso en la salida.

```

1 clau@machine:~/sample_files$ awk '{print $1*$2}' file01.dat
2 ...
3 188048
4 282200
5 848788
6 clau@machine:~/sample_files$

```

Sumar dos columnas, en este caso sumar las columnas 2 y 3.

```

1 clau@machine:~/sample_files$ awk '{print $2+$3}' file02.dat
2 ...
3 602
4 1685
5 887
6 clau@machine:~/sample_files$

```

Sumar todos los términos de una columna, en este caso la columna 3.

```

1 clau@machine:~/sample_files$ awk '{sum += $3} END {print sum}' file02.dat
2 499500
3 clau@machine:~/sample_files$

```

Promedia los valores de una columna, en este caso la columna 2.

```

1 clau@machine:~/sample_files$ awk
  ↪ '{ sum += $2; n++ } END { if (n > 0) print sum / n; }' file02.dat
2 499.5
3 clau@machine:~/sample_files$

```

Para quienes están más familiarizados con la programación notarán que las instrucciones END, if, y print, están ligadas a procesos lógicos de lenguajes de programación.

2.6.6 Edición al vuelo

La edición al *vuelo* se refiere a la capacidad de modificar textos mientras los comandos van siendo ejecutados. En ejemplos previos ya lo hemos realizado mediante la herramienta [sed](#). Ahora cubriremos un poco más de detalles de [sed](#) con algunos ejemplos específicos. El programa [sed](#)

es utilizado para el manejo de filas, su sintaxis resumida lo hacen el programa ideal para edición *al vuelo*.

El uso general de `sed` es de la forma:

```
1 clau@machine:$ sed opciones 'script' archivo
```

Sus opciones son muchas, en la siguiente lista se muestran algunos casos de utilidad.

- ▶ `sed -n '5,10p' file.txt` → Imprime entre la línea 5 y la 10.
- ▶ `sed '20,35d' file.txt` → Imprime todo excepto entre la línea 20 y 35.
- ▶ `sed -n -e '5,7p' -e '10,13p' file.txt` → Imprime líneas 5-7 y 10-13
- ▶ `sed 's/version/story/g' file.txt` → reemplaza `version` por `story` en el archivo.
- ▶ `sed 's/ */ /g' file.txt` → Reemplaza múltiples espacios por un espacio simple.
- ▶ `sed '30,40 s/version/story/g' file.txt` → Reemplaza en un rango específico.
- ▶ `sed G myfile.txt` → Inserta espacios entre líneas.
- ▶ `sed '$d' file.txt` → Borra la última línea del archivo.
- ▶ `sed 'nd' file.txt` → Borra la *n*-ésima línea del archivo.

Al momento de ejecutar estas instrucciones, `sed` lo que hace es ir leyendo línea a línea e integrando las opciones que nosotros hayamos especificados, luego imprime en pantalla cada uno de los resultados. Así conseguimos editar mientras el archivo original es observado.

Una de las ventajas de `sed` es que puede tomar la salida de otros comando y utilizarlo como entrada para programar análisis más complejos.

2.6.7 Operaciones básicas con sed

Al igual que antes, vamos a utilizar los 100 ficheros ya descargados y renombrados para ver las capacidades de `sed` en algunos casos. Pero también veremos casos simples de reemplazo y edición utilizando un texto de un cuento, que puede descargar en <https://gitarra.cl/lectures/modcomp/-/raw/main/gcb.txt?inline=false>. Recuerde que para descargar este archivo de texto –en el mismo directorio que tiene los 100 archivos renombrados–, basta con:

```
1 clau@machine:~/sample_files$ wget
↳ https://gitarra.cl/lectures/modcomp/-/raw/main/gcb.txt
```

Antes de revisar los ejemplos también inspeccione el fichero con –por ejemplo– `less gcb.txt`, y lea el cuento, para entender mejor los ejemplos a continuación.

Búsqueda por una coincidencia en un archivo, y reemplazar. En este caso reemplazamos la palabra `Juan` en el documento, por `Joaquin`. Note que al final, en lugar de utilizar `| less` para observar los cambios, puede utilizar `> nuevo.txt` para guardar los cambios en un nuevo fichero.

```
1 clau@machine:~/sample_files$ sed 's/Juan/Joaquin/g'
  ↪ gcb.txt | less
```

Búsqueda por coincidencia y añadir texto previamente.

```
1 clau@machine:~/sample_files$ sed 's/\(Juan\)/Don \1 /'
  ↪ gcb.txt | less
2 clau@machine:~/sample_file$
```

Añadir líneas antes o después de una coincidencia.

```
1 clau@machine:$ sed '/rey/ s/^/---En la siguiente linea se habla del rey---.\n/'
  ↪ gcb.txt | less
2 clau@machine:$ sed 's/rex/&\n---En la linea anterior se habló del rey---.\n/' gcb.txt
  ↪ | less
```

Borrar líneas vacías

```
1 clau@machine:$ sed '/^$/d' gcb.txt
```

Borrar líneas que tenga un patrón en dos oportunidades.

```
1 clau@machine:$ sed 's/Juan/dl/i2;t' cgb.txt | sed
  ↪ '/dl/d' | less
```

Muchos de los ejemplos anteriores, pueden parecer algo crípticos en la simbología que usa frecuentemente sed, esto esta relacionado al uso de expresiones regulares (*regular expressions*), que poseen mucha documentación *online*¹⁹.

19: Puede visitar por ejemplo <https://www.sitepoint.com/learn-regex/>

2.6.8 Pegado de archivos

Cuando trabajamos con archivos, especialmente del tipo que contienen datos, en ocasiones quisieramos combinar columnas de un archivo, con columnas de otro archivo, o juntar columnas en una sola nueva columna. Si bien esto es posible con softwares de ofimática²⁰, nuestro objetivo es hacerlo de forma más simple en la terminal, para lo que contamos con los comandos `cat`, y `paste`.

20: Tales como LibreOffice, o Microsoft Excel.

Concateida en la tenar dos archple y rápivos

```
1 clau@machine:$ cat file1.dat file2.dat > nuevo.dat
2 clau@machine:$
```

Pegar como columnas nuevas

```
1 clau@machine:$ paste file1.dat file2.dat > nuevo.dat
2 clau@machine:$
```

Como veremos en ejemplos más adelante, es posible entregar al comando `paste`, comandos provenientes de otros análisis lo que ayuda a acelerar la manipulación de estas columnas.

2.7 BASH como lenguaje

EN MUCHAS OPORTUNIDADES TENEMOS DIRECTORIOS CON FICHeros QUE DESEAMOS ANALIZAR, y hemos visto hasta ahora como, mediante `find` y `sed`, generar sistemas de análisis –o la generación de comandos complejos– sobre archivos.

Sin embargo, para sistemas que requieren múltiples análisis llevar a cabo estas implementaciones en línea de comando puede ser algo críptico, por lo que el uso de iteradores puede ayudarnos a darle una sintaxis más simple a nuestros comandos, así como también el uso de herramientas matemáticas en BASH.

2.7.1 El ciclo FOR en BASH

El ciclo `for` en bash es una de las primeras herramientas para llevar a cabo análisis más complejos, por ejemplo consideremos el siguiente comando

```
1 clau@machine:$ for i in 1 2 3;do echo $i; done
2 1
3 2
4 3
5 clau@machine:$
```

Como se puede observar la instrucción `for` pertenece a BASH, y lo que hace es iterar sobre una variable –en este caso `i`– sobre distintos valores –en este caso 1, 2 y 3–. En el proceso de iteración la instrucción a realizar es subsecuente a la instrucción `do`. Así lo que se realizará es ejecutar un `echo` en la pantalla de la variable, ahora llamada con `$i`, y el ciclo es cerrado con la instrucción `done`.

Se puede por ejemplo, mostrar la tabla de multiplicar de 9, usando simplemente:

```
1 clau@machine:$ for i in `seq 1 9`; do print "$i*9 = $((($i*9))"; done
```

Ejecute el comando para ver los resultados. Note el doble paréntesis dentro de la instrucción `echo` para la operación matemática. Esto será clave para realizar operaciones en *scripts* sobre todo en casos simples.

Uno de los elementos claves en la ejecución anterior es el uso de apóstrofes ``` para encerrar un comando adicional sobre el cual vamos a iterar. Por ejemplo si sólo ubiéramos ejecutado el comando `seq` veríamos que

```

1 clau@machine:$ seq 1 9
2 1
3 2
4 ....
5 9
6 clau@machine:$

```

sólo se obtendría el resultado de una secuencia numérica de 1 a 9 (revise el manual de [seq](#), para ver más opciones). Entonces el uso de estos apóstrofes nos permite iterar no sólo sobre valores numéricos sino que además sobre resultados provenientes de otros comandos.

Descarguemos y descomprimamos el fichero de <http://ejemplo> y veamos que ocurre cuando buscamos y analizamos los datos con el siguiente comando.

```

1 clau@machine:$ wget http://ejemplo
2 clau@machine:$ tar -xvzf archivo.tgz
3 clau@machine:$ cd Data_sample
4 clau@machine:$ for i in `find . -iname *.dat`; do echo "Analizando archivo $i"; done

```

Podrá observar usted que con el último comando ejecutado, esta realizando una iteración sobre los archivos *.dat presentes en el directorio de trabajo.

Ahora, podríamos pensar en algo más complejo, por ejemplo tomar cada uno de estos archivos y contar cuantas palabras tiene cada uno utilizando el comando [wc](#), con la opción -w para sólo imprimir el número de palabras de cada archivo.

```

1 clau@machine:$ for i in `find . -iname *.dat`; do wc -w $i; done

```

Al ejecutar el comando previo veremos el número de palabras de cada archivo. Ahora, si quisiéramos saber el número de palabras promedio, podríamos reconstruir la salida de nuestro comando anterior y combinarlo con, por ejemplo, [awk](#) para calcular el promedio. Así, nuestro comando quedaría de la forma

```

1 clau@machine:$ for i in `find . -iname *.dat`; do wc -w $i; done \
2 | awk '{sum += $1; n++} END {print sum/n}'

```

El símbolo \ representa continuidad de la línea. Es importante notar que a partir de un comando para buscar, junto a un iterador, junto a un comando para contar palabras, y un comando para analizar columnas de datos podamos finalmente obtener nuestro resultado combinando de forma adecuada cada uno de estos comandos. Esto es en esencia el uso de BASH para problemáticas cotidianas.

Si bien la combinación en ocasiones puede ser tediosa, y algo críptica para el mismo usuario, uno puede desarrollar todos estos comandos mediante la creación de *scripts* que son archivos ejecutables que poseen todos los comandos que necesitamos.

2.7.2 Operaciones Matemáticas

Antes de continuar con la creación de *scripts* para sistemas más complejos, veamos ejemplos muy simples utilizando operaciones matemáticas en BASH.

Las operaciones elementales en BASH, tales como suma, resta, y multiplicación de números enteros se pueden tratar directamente con el uso de `$((OPERACIONES))`, veamos por ejemplo los comandos

```
1 clau@machine:$ echo $((5+5))
2 10
3 clau@machine:$ echo $((6*5))
4 30
5 clau@machine:$ echo $((90-32))
6 58
```

Si bien en general es simple tratar estas operaciones con números enteros, puede ser más complejo cuando hablamos de números reales, por lo que en estos casos se suele usar una herramienta más adecuada como lo es `bc`. El comando `bc` es un language de claculadora con precisión arbitraria en bash. Por ahora lo utilizaremos sólo para operaciones básicas. Veamos algunos casos en particular.

```
1 clau@machine:$ echo "5.2*6.9" | bc
2 35.8
3 clau@machine:$ echo "2.4*2.4+3" | bc
4 8.7
5 clau@machine:$ echo "2/5" | bc
6 2
7 clau@machine:$ echo "2/5" | bc -l
8 2.500000000
```

Así, las opciones de operaciones matemáticas dentro de nuestros comandos se complejizan cuando combinamos estos con la herramienta `bc`.

2.7.3 Scripts

Automatizar procesos puede ser útil desde un punto de vista práctico. Por ejemplo si queremos cada cierto tiempo revisar que no tengamos archivos temporales, hacer copias de seguridad de nuestros datos, o cualquier otra tarea automatizada, podemos diseñar *scripts* que es simplemente un set de instrucciones de la SHELL –en nuestro caso BASH–, que cumplen con esa labor.

Consideremos por ejemplo un análisis simple que consiste en tomar dos columnas de archivos distintos, realizar la suma de estos datos, y luego calcular el promedio, desviación estándar, y media de la columna final.

Lo primero que debemos hacer es pensar en cuál es el proceso que debemos realizar sobre los archivos `a.dat` y `b.dat`, consideremos lo siguiente:

- ▶ Primero debemos saber que columnas tomaremos de cada archivo.
- ▶ Luego debemos sumar estas columnas para tener una columna final con los valores sumados.
- ▶ Debemos sumar los datos de la columna final, que nos servirá para calcular su promedio.
- ▶ Debemos operar –conocer como calcular la desviación estándar– sobre los datos de la columna final para calcular la desviación estándar, y del mismo modo calcular la media.

Tomemos entonces los archivos –de ejemplo– del sitio <http://aca> y descomprimos el directorio en nuestro home folder

```
1 clau@machine:$ wget http://aca
2 clau@machine:$ tar -xvzf aca.tbz
3 clau@machine:$ cd aca/
```

Bien, de estos dos archivos, que podemos inspeccionar con `less`, `cat` o `vim`, vamos a tomar la columna 2 para el archivo `a.dat` y la columna 5 para el archivo `b.dat`, y sumaremos estas columnas generando una nueva columna. Pero a diferencia de antes, vamos a generar estos comandos en un *script* que contendrá cada una de las instrucciones.

La construcción de un *script* consiste en crear un nuevo archivo que va a contener todas las instrucciones. Si lo editamos con `vim`, las primeras líneas serán de la forma

```
#!/bin/sh

paste <(awk '{print $2}' a.dat) <(awk '{print $5}' b.dat ) > tmp.dat

awk '{print $1 + $2}' tmp.dat
```

Como se puede observar, el fichero contiene una primera línea `#!/bin/sh` que hace un llamado al shell en caso de que le asignemos permisos de ejecución a nuestro fichero. Las siguientes líneas se encargan de pegar la columna 2 del fichero `a.dat` con la columna 5 del fichero `b.dat` en un nuevo fichero que llamaremos `tmp.dat`. Finalmente `awk` se encarga en sumar las columnas 1 y 2 –del nuevo fichero `tmp.dat`– y mostrar esta suma en pantalla. Una vez que guardamos y salimos, tendremos nuestro nuevo fichero `script.sh` en el directorio de trabajo.

Ahora, lo que necesitamos es operar sobre el resultado de la suma de estas columnas por lo que procederemos a guardar esta suma en un fichero nuevo que llamaremos `sum.dat`. Con el que finalmente podremos calcular de manera correcta la suma, promedio, desviación estándar y media.

Así nuestro fichero `scripts.sh` queda de la forma

```
#!/bin/sh

paste <(awk '{print $2}' a.dat) <(awk '{print $5}' b.dat ) > tmp.dat

awk '{print $1 + $2}' tmp.dat > sum.dat
```

```
SUMA=`awk '{sum+=$1} END {print sum}'`
PROMEDIO=`awk '{sum+=$1; n++;} END {print sum/n}'`
STD=`awk '{sum+=$1; sumsq+=$1*$1} END {print sqrt(sumsq/NR - (sum/NR)^2)}'`

echo "La suma es = $SUMA"
echo "El promedio es = $PROMEDIO"
echo "La Desviacion es = $STD"
```

Así, con nuestro archivo completado, podemos asignarle permisos de ejecución para poder ejecutarlo de forma simple. Para cambiar los permisos utilizamos el comando `chmod`.

```
1 clau@machine:$ chmod 755 script.sh
```

21: `chmod` tiene muchas posibilidades para la asignación de permisos, para más ayuda puede recurrir al manual o visitar [some-web-site](#).

Una vez que posee los permisos de ejecución²¹, podemos ejecutar nuestro *script* en nuestro terminal

```
1 clau@machine:$ ./script.sh
2 La suma es
3 El promedio es
4 La Desviacion es
```

Así hemos incorporado un set de comandos de BASH en un fichero con cada una de las instrucciones, es decir hemos escrito un programa –set de instrucciones, que resuelven un problema– en el lenguaje BASH de GNU/Linux. Tenemos un primer programa de computadora, que se encarga de analizar unos datos. Ahora nos enfocaremos en un lenguaje de programación más avanzado y con muchas funcionalidades adicionales, específicamente revisaremos Python3.

Si bien muchas de las problemáticas podemos resolverlas con lenguajes de programación, es bueno tener en cuenta que con BASH podemos hacer trabajos importantes y reducir el esfuerzo de programación en lenguajes más avanzados.