Una **expresión regular** es un patrón que nos permite buscar un texto formado por metacaracteres y caracteres ordinarios.

Los **metacaracteres** son ciertos caracteres con un significado específico dentro de una expresión regular. Estos caracteres tienen un significado que va más allá del símbolo que representan y tienen un comportamiento especial en una expresión regular.

Aquí tenéis una lista de metacaracteres que usamos en expresiones regulares:

- . Significa cualquier caracter.
- ^Indica el principio de una línea.
- $ Indica el final de una línea.
- * Indica cero o más repeticiones del caracter anterior.
- + Indica una o más repeticiones del caracter anterior.
- \< Indica el comienzo de una palabra.
- \> Indica el final de una palabra.
- \ Caracter de escape. Da significado literal a un metacaracter.
- [ ] Uno cualquiera de los caracteres entre los corchetes. Ej: [A-Z] (desde A hasta Z).
- [^] Cualquier caracter distinto de los que figuran entre corchetes: Ej: [^A-Z].
- { } Nos permiten indicar el número de repeticiones del patrón anterior que deben darse.
- | Nos permite indicar caracteres alternativos: Ej: (^|[?&])
- ( ) Nos permiten agrupar patrones. Ej: ([0-9A-F]+:)+

Ojo. En las expresiones regulares se distingue entre mayúsculas y minúsculas.

Si queremos representar un caracter entre la a y la z, lo haremos de la siguiente manera: [a-z]
Dentro del conjunto, podemos especificar todos los caracteres que queramos. Por ejemplo: [a-zA-Z] representaría los caracteres alfabéticos en minúsculas y mayúsculas. Eso sí. El conjunto representa a un sólo caracter.
Si lo que queremos es representar identificar un número o una letra, podríamos hacerlo así: [a-zA-Z0-9]

Los conjuntos pueden representarse, nombrando todos y cada uno de los elementos, o el intervalo. Ej: [0-9] representa lo mismo que [0123456789].

Si queremos representar un número que se compone de cero o más dígitos: [0-9]*

Y si queremos representar un número que se compone de uno o más dígitos: [0-9]+

Si ahora queremos representar cualquier caracter menos los dígitos: [^0-9]

Ahora, imaginemos que queremos representar un número de 5 dígitos: [0-9]{5}

Y si quisieramos representar una palabra que tiene entre dos y cuatro caracteres: [a-zA-Z]{2,4}

Dentro de los conjuntos de caracteres individuales, se reconocen las siguientes categorías:

```
[:alnum:] alfanuméricos
[:alpha:] alfabéticos
[:cntrl:] de control
[:digit:] dígitos
[:graph:] gráficos
[:lower:] minúsculas
```

```
[:print:] imprimibles
[:punct:] de puntuación
[:space:] espacios
[:upper:] mayúsculas
[:xdigit:] dígitos hexadecimales
```

Vamos a ver algunos ejemplos de expresiones regulares:

 # grep '^La' fichero
El comando anterior nos devuelve todas las líneas del fichero que comienzan por **La**.

 # grep '^ *La' fichero
El comando anterior nos devuelve todas las líneas del fichero que comienzan por cualquier número de espacios seguido de **La**.

 # grep '^\..*' fichero
El comando anterior nos devuelve todas las líneas del fichero que comienzan por punto y tienen cualquier número de caracteres.

 # ls -la | grep ' \..*'
El comando anterior nos devuelve la lista de ficheros que comienzan por un espacio seguido de un punto y cualquier número de caracteres, es decir, la lista de ficheros ocultos.

 # ls -l | grep '^d'
El comando anterior nos devuelve la lista de ficheros que comienzan por **d**, es decir, la lista de directorios.

¿Qué es una expresión regular?

Una expresión regular es una serie de caracteres especiales que permiten describir un texto que queremos buscar. Por ejemplo, si quisiéramos buscar la palabra “linux” bastaría poner esa palabra en el programa que estemos usando. La propia palabra es una expresión regular. Hasta aquí parece muy simple, pero, ¿y si queremos buscar todos los números que hay en un determinado fichero? ¿O todas las lineas que empiezan por una letra mayúscula? En esos casos ya no se puede poner una simple palabra. La solución es usar una expresión regular.

Expresiones regulares vs patrones de ficheros.

Antes de empezar a entrar en materia sobre las expresiones regulares, quiero aclarar un malentendido común sobre las expresiones regulares. Una

expresión regular no es lo que ponemos como parámetro en los comandos como rm, cp, etc para hacer referencia a varios fichero que hay en el disco duro. Eso sería un patrón de ficheros. Las expresiones regulares, aunque se parecen en que usan algunos caracteres comunes, son diferentes. Un patrón de fichero se lanza contra los ficheros que hay en el disco duro y devuelve los que encajan completamente con el patrón, mientras que una expresión regular se lanza contra un texto y devuelve las lineas que contienen el texto buscado. Por ejemplo, la expresión regular correspondiente al patrón `*.*` seria algo así como `^.*\..*$`

Tipos de expresiones regulares.

No todos los programas utilizan las mismas expresiones regulares. Ni mucho menos. Existen varios tipos de expresiones regulares más o menos estándar, pero hay programas que cambian ligeramente la sintaxis, que incluyen sus propias extensiones o incluso que utilizan unos caracteres completamente diferentes. Por eso, cuando queráis usar expresiones regulares con algún programa que no conozcáis bien, lo primero es mirar el manual o la documentación del programa para ver cómo son las expresiones regulares que reconoce.

En primer lugar, existen dos tipos principales de expresiones regulares, que están recogidas en el estándar POSIX, que es el que usan las herramientas de Linux. Son las expresiones regulares básicas y las extendidas. Muchos de los comandos que trabajan con expresiones regulares, como grep o sed, permiten usar estos dos tipos. Más abajo hablare de ellos. También están las expresiones regulares estilo PERL, y luego hay programas como vim o emacs que usan variantes de estas. Según lo que queramos hacer puede ser más adecuado usar unas u otras.

Probando expresiones regulares.

La sintaxis de las expresiones regulares no es nada trivial. Cuando tengamos que escribir una expresión regular complicada estaremos delante de una ristra de caracteres especiales imposibles de entender a primera vista, así que para aprender a usarlas es imprescindible contar con una forma de hacer todas las pruebas que queramos y ver los resultados fácilmente. Por eso voy a poner ahora varios comandos con los que podremos hacer las pruebas y experimentar todo lo que necesitemos hasta que tengamos las expresiones regulares dominadas.

El primero de ellos es el comando grep. Este es el comando que usaremos con más frecuencia para hacer búsquedas. La sintaxis es la siguiente:

```
grep [-E] 'REGEX' FICHERO
COMANDO | grep [-E] 'REGEX'
```

Recomiendo poner siempre las expresiones regulares entre comillas simples para que el shell no nos haga de las suyas. La primera forma sirve para buscar una expresión regular en un fichero. La segunda permite filtrar la salida de un comando a través de una expresión regular. Por defecto, grep usa expresiones regulares básicas. La opción -E es para usar expresiones regulares extendidas.

Un truco que nos puede ayudar a ver el funcionamiento de las expresiones regulares es activar el uso del color en el comando grep. De esa manera, aparecerá resaltada la parte del texto que empareja con la expresión regular que estamos usando. Para activar el color en el comando grep basta con asegurarse que la variable de entorno `GREP_OPTIONS` contenga en valor `--color`, cosa que se puede hacer con este comando:

```
GREP_OPTIONS=--color
```

Podemos ponerlo en el .bashrc para tenerlo activado siempre.

Otra forma de usar expresiones regulares es mediante el comando sed. Este es más adecuado para reemplazar texto, pero también puede usarse para hacer búsquedas. La sintaxis para ello sería así:

```
sed -n[r] '/REGEX/p' FICHERO
COMANDO | sed -n[r] '/REGEX/p'
```

El comando sed también usa expresiones regulares básicas por defecto, se pueden usar expresiones regulares extendidas con la opción -r.

Otro comando que también quiero nombrar es awk. Este comando puede usarse para muchas cosas, ya que permite escribir scripts en su propio lenguaje de programación. Si lo que queremos es buscar una expresión regular en un fichero o en la salida de un comando, la forma de usarlo sería la siguiente:

```
awk '/REGEX/' FICHERO
COMANDO | awk '/REGEX/'
```

Este comando usa siempre expresiones regulares extendidas.

Para hacer nuestras pruebas también necesitaremos un texto que nos sirva como ejemplo para hacer búsquedas en él. Podemos utilizar el siguiente texto:

```
- Lista de páginas wiki:



ArchLinux: https://wiki.archlinux.org/

Gentoo: https://wiki.gentoo.org/wiki/Main_Page

CentOS: http://wiki.centos.org/
```

```
Debian: https://wiki.debian.org/

Ubuntu: https://wiki.ubuntu.com/



- Fechas de lanzamiento:



Arch Linux: 11-03-2002

Gentoo: 31/03/2002

CentOs: 14-05-2004 03:32:38

Debian: 16/08/1993

Ubuntu: 20/10/2004



Desde Linux Rulez.
```

Este es el texto que usaré para los ejemplos del resto del post, así que os recomiendo que lo copiéis en un fichero para tenerlo a mano desde la terminal. Podéis poner el nombre que queráis. Yo lo he llamando regex.

Entrando en materia.

Ahora ya tenemos todo lo necesario para empezar a probar las expresiones regulares. Vayamos poco a poco. Voy a poner varios ejemplos de búsquedas con expresiones regulares en los que iré explicando para qué sirve cada carácter. No son ejemplos muy buenos, pero como me va a quedar un post muy largo no quiero complicarlo más. Y eso que sólo voy a arañar la superficie de lo que se puede hacer con expresiones regulares.

Lo más sencillo de todo es buscar una palabra concreta, por ejemplo, supongamos que queremos buscar todas las lineas que contengan la palabra "Linux". Esto es lo más fácil, ya que sólo tenemos que escribir:

```
grep 'Linux' regex
```

Y podremos ver el resultado:

```
ArchLinux: https://wiki.archlinux.org/

Arch Linux: 11-03-2002

Desde Linux Rulez.
```

Estas son las tres lineas que contienen la palabra "Linux" la cual, si hemos usado el truco del color, aparecerá resaltada. Fijaros que reconoce la palabra que estamos buscando aunque forme parte de una palabra más larga como en "ArchLinux". Sin embargo, no resalta la palabra "linux" que aparece en la URL "https://wiki.archlinux.org/". Eso es porque ahí aparece con la "l" minúscula y la hemos buscado en mayúscula. El comando grep tiene opciones para esto, pero no voy a hablar de ellas en un artículo que trata sobre expresiones regulares.

Con esta sencilla prueba podemos sacar la primera conclusión:

- Un carácter normal puesto en una expresión regular empareja consigo mismo.

Lo que viene a decir que si pones la letra "a" buscará la letra "a". Parece lógico, ¿verdad?

Supongamos ahora que queremos buscar la palabra "CentO" seguida de cualquier carácter, pero sólo un único carácter. Para esto podemos usar el carácter ".", que es un comodín que empareja con un carácter cualquiera, pero sólo uno:

```
grep 'CentO.' regex
```

Y el resultado es:

```
CentOS: http://wiki.centos.org/

CentOs: 14-05-2004 03:32:38
```

Lo que significa que incluye la "S" de "CentOS" aunque en un caso es mayúscula y en otro minúscula. Si apareciera en ese lugar cualquier otro carácter también lo incluiría. Ya tenemos la segunda regla:

- El carácter "." empareja con cualquier carácter.

Ya no es tan trivial como parecía, pero con esto no podemos hacer mucho. Avancemos un poco más. Vamos a suponer que queremos encontrar las líneas en las que aparece el año 2002 y el 2004. Parecen dos búsquedas, pero se pueden hacer de una sola vez así:

```
grep '200[24]' regex
```

Lo que quiere decir que queremos buscar el número 200 seguido del 2 o el 4. Y el resultado es este:

```
Arch Linux: 11-03-2002

Gentoo: 31/03/2002

CentOs: 14-05-2004 03:32:38

Ubuntu: 20/10/2004
```

Lo que nos lleva a la tercera regla:

- Varios caracteres encerrados entre corchetes emparejan con cualquiera de los caracteres que hay dentro de los corchetes.

Los corchetes dan más juego. también se pueden usar para excluir caracteres. Por ejemplo, supongamos que queremos buscar los sitios en los que aparece el carácter ":", pero que no vaya seguido de "/". El comando sería así:

```
grep ':[^/]' regex
```

Se trata simplemente de poner un "^" como primer carácter dentro del corchete. Se pueden poner a continuación todos los caracteres que se quieran. El resultado de este último comando es el siguiente:

```
ArchLinux: https://wiki.archlinux.org/

Gentoo: https://wiki.gentoo.org/wiki/Main_Page

CentOS: http://wiki.centos.org/

Debian: https://wiki.debian.org/

Ubuntu: https://wiki.ubuntu.com/

Arch Linux: 11-03-2002

Gentoo: 31/03/2002

CentOs: 14-05-2004 03:32:38

Debian: 16/08/1993

Ubuntu: 20/10/2004
```

Ahora aparecen resaltados los ":" que hay detrás de los nombres de las distros, pero no los que hay en las URL porque los de las URL llevan "/" a continuación.

- Poner el carácter "^" al principio de un corchete empareja con cualquier carácter excepto con los demás caracteres del corchete.

Otra cosa que podemos hacer es especificar un rango de caracteres. Por ejemplo, para buscar cualquier número seguido de un "-" sería así:

```
grep '[0-9]-' regex
```

Con esto estamos especificando un carácter entre 0 y 9 y, a continuación, un signo menos. Veamos el resultado:

```
Arch Linux: 11-03-2002

CentOs: 14-05-2004 03:32:38
```

Se pueden especificar varios rangos dentro de los corchetes a incluso mezclar rangos con caracteres sueltos.

- Colocar dos caracteres separados por "-" dentro de los corchetes empareja con cualquier carácter dentro del rango.

Vamos a ver ahora si podemos seleccionar la primera parte de las URL. La que pone "http" o "https". Sólo se diferencian en la "s" final, así que vamos a hacerlo de la siguiente manera:

```
grep -E 'https?' regex
```

La interrogación sirve para hacer que el carácter que hay a su izquierda sea opcional. Pero ahora hemos añadido la opción -E al comando. Esto es porque la interrogación es una característica de las expresiones regulares extendidas. Hasta ahora estábamos usando expresiones regulares básicas, así que no hacía falta poner nada. Veamos el resultado:

```
ArchLinux: https://wiki.archlinux.org/

Gentoo: https://wiki.gentoo.org/wiki/Main_Page

CentOS: http://wiki.centos.org/
```

```
Debian: https://wiki.debian.org/

Ubuntu: https://wiki.ubuntu.com/
```

O sea que ya tenemos una nueva regla:

- Un carácter seguido de "?" empareja con ese carácter o con ninguno. Esto sólo es válido para expresiones regulares extendidas.

Ahora vamos a buscar dos palabras completamente diferentes. Vamos a ver cómo buscar las lineas que contengan tanto la palabra "Debian" como "Ubuntu".

```
grep -E 'Debian|Ubuntu' regex
```

Con la barra vertical podemos separar dos o más expresiones regulares diferentes y buscar las lineas que emparejen con cualquiera de ellas:

```
Debian: https://wiki.debian.org/

Ubuntu: https://wiki.ubuntu.com/

Debian: 16/08/1993

Ubuntu: 20/10/2004
```

- El carácter "|" sirve para separar varias expresiones regulares y empareja con cualquiera de ellas. También es específico de las expresiones regulares extendidas.

Continuemos. Ahora vamos a buscar la palabra "Linux", pero sólo donde no esté pegada a otra palabra por la izquierda. Podemos hacerlo así:

```
grep '\
```

Aquí el carácter importante es "<", pero es necesario escaparlo colocando "\" delante para que grep lo interprete como un carácter especial. El resultado es el siguiente:

```
Arch Linux: 11-03-2002

Desde Linux Rulez.
```

También se puede usar "\>" para buscar palabras que no estén pegadas a otras por la derecha. Vamos con un ejemplo. Probemos este comando:

```
grep 'http\>' regex
```

El resultado que produce es este:

```
CentOS: http://wiki.centos.org/
```

Ha salido "http", pero no "https", porque en "https" todavía hay un carácter a la derecha de la "p" que puede formar parte de una palabra.

- Los caracteres "<" y ">" emparejan con el principio y el final de una palabra, respectivamente. Hay que escapar estos caracteres para que no se interpreten como caracteres literales.

Vamos con cosas un poco más complicadas. El carácter "+" empareja con el carácter que aparece a su izquierda repetido al menos una vez. Este carácter sólo está disponible con las expresiones regulares extendidas. Con él podemos buscar, por ejemplo, secuencias de varios números seguidos que empiecen con ":".

```
grep -E ':[0-9]+' regex
```

Resultado:

```
CentOs: 14-05-2004 03:32:38
```

Queda resaltado también el número 38 porque también empieza con ":".

- El carácter "+" empareja con el carácter que aparece a su izquierda repetido al menos una vez.

También se puede controlar el número de repeticiones usando "{" y "}". La idea es colocar entre llaves un número que indica el número exacto de repeticiones que queremos. También se puede poner un rango. Vamos a ver ejemplos de los dos casos.

Primero vamos a buscar todas las secuencias de cuatro dígitos que haya:

```
grep '[0-9]\{4\}' regex
```

Fijaros en que hay que escapar las llaves si estamos usando expresiones regulares básicas, pero no si usamos las extendidas. Con extendidas sería así:

```
grep -E '[0-9]{4}' regex
```

Y el resultado en los dos casos sería este:

```
Arch Linux: 11-03-2002

Gentoo: 31/03/2002

CentOs: 14-05-2004 03:32:38

Debian: 16/08/1993

Ubuntu: 20/10/2004
```

- Los caracteres "{" y "}" con un número entre ellos emparejan con el carácter anterior repetido el número de veces indicado.

Ahora el otro ejemplo con las llaves. Supongamos que queremos encontrar palabras que tengan entre 3 y 6 letras minúsculas. Podríamos hacer lo siguiente:

```
grep '[a-z]\{3,6\}' regex
```

Y el resultado sería este:

```
- Lista de páginas wiki:

ArchLinux: https://wiki.archlinux.org/

Gentoo: https://wiki.gentoo.org/wiki/Main_Page

CentOS: http://wiki.centos.org/

Debian: https://wiki.debian.org/

Ubuntu: https://wiki.ubuntu.com/

- Fechas de lanzamiento:

Arch Linux: 11-03-2002

Gentoo: 31/03/2002

CentOs: 14-05-2004 03:32:38

Debian: 16/08/1993

Ubuntu: 20/10/2004

Desde Linux Rulez.
```

Que, como veis, no se parece mucho a lo que queríamos. Eso es porque la expresión regular encuentra las letras dentro de otras palabras que son más largas. Probemos con esta otra versión:

```
grep '\<[a-z]\{3,6\}\>' regex
```

Resultado:

```
- Lista de páginas wiki:
```

```
ArchLinux: https://wiki.archlinux.org/

Gentoo: https://wiki.gentoo.org/wiki/Main_Page

CentOS: http://wiki.centos.org/

Debian: https://wiki.debian.org/

Ubuntu: https://wiki.ubuntu.com/
```

Esto ya se parece más a lo queríamos. Lo que hemos hecho es exigir que la palabra empiece justo delante de la primera letra y termine justo detrás de la última.

- Los caracteres "{" y "}" con dos números entre ellos separados por una coma emparejan con el carácter anterior repetido el número de veces indicado por los dos números.

Veamos ahora un carácter que es primo de "+". Se trata de "*" y su funcionamiento es muy parecido sólo que empareja con cualquier número de caracteres incluido cero. O sea que hace lo mismo que el "+" pero no exige que el carácter de su izquierda aparezca en el texto. Por ejemplo, probemos a buscar esas direcciones que empiezan en wiki y acaban en org:

```
grep 'wiki.*org' regex
```

Vamos a ver el resultado:

```
ArchLinux: https://wiki.archlinux.org/

Gentoo: https://wiki.gentoo.org/wiki/Main_Page

CentOS: http://wiki.centos.org/

Debian: https://wiki.debian.org/
```

Perfecto.

Ahora el último carácter que vamos a ver. El carácter "\" sirve para escapar el carácter de su derecha de manera que pierda el significado especial que tiene. Por ejemplo: Supongamos que queremos localizar la lineas que terminen en punto. Lo primero que se nos podría ocurrir podría ser esto:

```
grep '.$' regex
```

El resultado no es el que buscamos:

```
- Lista de páginas wiki:

ArchLinux: https://wiki.archlinux.org/

Gentoo: https://wiki.gentoo.org/wiki/Main_Page

CentOS: http://wiki.centos.org/

Debian: https://wiki.debian.org/

Ubuntu: https://wiki.ubuntu.com/

- Fechas de lanzamiento:

Arch Linux: 11-03-2002

Gentoo: 31/03/2002

CentOs: 14-05-2004 03:32:38

Debian: 16/08/1993

Ubuntu: 20/10/2004

Desde Linux Rulez.
```

Esto es porque el carácter "." empareja con cualquier cosa, así que esa expresión regular empareja con el último carácter de cada linea sea cual sea. La solución es esta:

```
grep '\.$' regex
```

Ahora el resultado sí que es el que queremos:

```
Desde Linux Rulez.
```

5.2 El comando `fgrep`

El comando `fgrep` es similar a `grep`, pero con tres diferencias principales: se puede utilizar para buscar varios objetivos al mismo tiempo, no permite utilizar expresiones regulares para buscar patrones y es más rápida que `grep`. Cuando se busca en un archivo grande o en varios pequeños, la diferencia de velocidad puede ser significativa.

Con `fgrep` se pueden buscar las líneas que contengan uno cualquiera de varios objetivos alternativos. Por ejemplo, la siguiente orden busca las entradas en el archivo `recetas.txt` que contengan las palabras "`pollo`" o "`pavo`".

```
$ fgrep "pollo
> pavo" recetas.txt
```

Cuando a `fgrep` se dan varios objetivos de búsqueda, cada uno debe estar en una línea separada. En este ejemplo, si no se hubiera puesto `pavo` en la siguiente línea hubiera buscado "pollo pavo".

`fgrep` no acepta expresiones regulares, los objetivos deben ser cadenas de texto.

Con la opción `-f` se puede decir a `fgrep` que tome los objetivos de búsquda de un archivo, en lugar de tenerlos que teclear directamente. Si tuviese una gran lista de direcciones denominada `direcciones.txt`, con nombres de clientes y direcciones y un pequeño archivo denominado `especial.txt` que contuviese los nombres de los clientes especiales, podría utilizar esta opción para seleccionar e imprimir las direcciones de los clientes especiales a partir de la lista completa:

```
$ fgrep -f especial.txt clientes.txt
```

El shell de linux: Comando sort

sort es uno de los comandos que utilizamos mucho a la hora de realizar scripts.

Nos permite ordenar los registros o líneas de uno o más archivos.
La ordenación se puede hacer por el primer carácter, por el primer campo de la línea o por un campo distinto al primero en el caso de ficheros estructurados.

Podemos ordenar el contenido de un fichero de la siguiente manera:

sort fichero
Se realizaría la ordenación y el resultado se mostraría por pantalla. Así que, si lo que queremos es obtener el resultado de la ordenación en un fichero, haríamos:

sort fichero > ficheroordenado
Si lo que queremos es ordenar varios ficheros y añadir el resultado a otro, podemos indicar varios ficheros en la línea de entrada:

sort fichero1 fichero2 > fichero3
Y si lo que queremos es ordenar un fichero y dejar el resultado de la ordenación en el mismo fichero, podemos hacerlo con el parámetro -o (output):
sort -o f1 f1

Veamos una lista de los parámetros que pueden resultarnos más útiles a la hora de usar este comando:

- -f : Este parámetro nos sirve para indicar que las mayúsculas y las minúsculas se van a tratar de forma diferente y que por tanto se va a seguir un ordenamiento alfabético.
- -n : Este parámetro nos sirve para ordenar los campos numéricos por su valor numérico.
- -r : Nos permite realizar una ordenación inversa, es decir, de mayor a menor.
- +número : Este parámetro nos sirve para indicar la columna o campo por el que vamos hacer la ordenación. Esta sintaxis está en desuso y se va a eliminar. En su lugar se utilizará la siguiente sintaxis:
- -k numero : De este modo especificaremos por qué columna o campo vamos a realizar la ordenación en las versiones más recientes de Linux.
- --field-separator= separador. Normalmente, se usa como delimitador de campos el espacio en blanco. Podemos utilizar el parámetro --field-separator para indicar que vamos a usar otro delimitador de campo cualquiera. Ej: --field-separator=, La opción abreviada de --field-separator es -t.
- -u : Nos permite suprimir todas las líneas repetidas después de realizar la ordenación.

Y algunos ejemplos con dichos parámetros:

Obtener un listado de los ficheros del directorio actual, ordenado por tamaño de archivo:
$ ls -l | sort +4n

Obtener un listado de los ficheros del directorio actual, ordenado de mayor a menor por tamaño de archivo:
$ ls -l | sort -r +4n

Obtener un listado de los ficheros del directorio actual, ordenado por nombre del archivo:
$ ls -l | sort +7
Ordenar un fichero eliminando las líneas repetidas:
$ sort -u fichero
Ordenar un fichero pen el que los campos están separados por comas, por el campo número 3:
$ sort -t, +3

Veamos un ejemplo en el que ordenemos usando la sintaxis actual para ordenar por columnas:
Imaginemos que queremos ver un listado de usuarios del fichero /etc/passwd ordenado por uid:
$ cat /etc/passwd| sort -t":" -k3n

Con -k3 le indicamos a sort que queremos ordenar por la columna 3. Y, al añadir la opción -n le indicamos que ordene por orden numérico.

Y si quisiéramos realizar la ordenación de mayor a menor:
$ cat /etc/passwd| sort -t":" -k3nr

Un ejemplo que uso mucho, cuando quiero eliminar las líneas repetidas de un archivo y dejar el contenido en el mismo archivo:
$ sort -o fichero -u fichero

Ejemplos de uso del comando find

FEB 3RD, 2014

A la hora de nombrar comando útiles en linux uno de los primero que se vendrá a la mente es el comando *find* el cual nos permite buscar archivos/directorios bajo una serie de criterios que nosotros le argumentemos, esto con la finalidad de que la busqueda sea lo mas exacta posible. Como las funcionalidades operacionales del comando son muchísimas nos remitiremos a un uso bástante simple en este post.

. Encontrar todos los archivos que terminen en .php

```
Miguel@MacBook:$ find . -type f -name *php
./dir1/app.php
./dir1/calm.php
./dir1/test1.php
./dir2/foo.php
```

. Encontrar todos los archivos que no terminen con .php

```
Miguel@MacBook:$ find . -type f ! -name *php
./dir3/caz.html
```

Esta busqueda también se puede realizar con el comando

```
Miguel@MacBook:$ find . -type f -not -name *php
./dir3/caz.html
```

. Es posible generar busquedas utilizando criterios multiples, en este caso vamos a buscar todos los arhivos con nombre “ca*” pero que no terminen con .php

```
Miguel@MacBook:$ find . -type f -name "ca*" ! -name *php
./dir3/caz.html
```

también, podemos utilizar el operador “AND/OR” cuando utilizamos el comando find y con ello solo los archivos/directorios que cumplen con los criterios serán listados.

```
Miguel@MacBook:$ find . -name '*.php' -o -name '*.txt' -o -name '*.html'
./dir1/app.php
./dir1/calm.php
./dir1/sun.txt
./dir1/test1.php
./dir2/foo.php
./dir3/caz.html
./dir3/caz.php
```

. Busqueda en multiples rutas de forma conjunta

```
Miguel@MacBook:$ find dir1/ dir2/ -name *php
dir1//app.php
dir1//calm.php
dir1//test1.php
dir2//foo.php
```

. Encontrando archivos ocultos

```
Miguel@MacBook:$ find . -name ".*"
./dir3/.hid.php
```

. Encontrando archivos con algún criterio de permisos. En este caso el criterio de permiso será 775

```
Miguel@MacBook:$ find . -type f -perm 0775
./dir1/kol.txt
./dir3/.hid.php
./dir3/hup.txt
```

Y si comprobamos los permisos, veremos que cumplen el criterio

```
Miguel@MacBook:$ ll ./dir1/kol.txt
-rwxrwxr-x  1 Miguel  staff     0B Feb  3 16:52 ./dir1/kol.txt
Miguel@MacBook:$ ll ./dir3/.hid.php
-rwxrwxr-x  1 Miguel  staff     0B Feb  3 16:49 ./dir3/.hid.php
Miguel@MacBook:$ ll ./dir3/hup.txt
-rwxrwxr-x  1 Miguel  staff     0B Feb  3 16:51 ./dir3/hup.txt
```

. Buscar por un usuario en particular

```
Miguel@MacBook:$ find dir1/ -user Miguel
dir1/
dir1//app.php
dir1//calm.php
dir1//kol.txt
dir1//sun.txt
dir1//test1.php
dir1//tool
```

Además del nombre podemos utilizar algún criterio adicional como el nombre, permiso, tipo, etc

```
Miguel@MacBook:$ find dir1/ -user Miguel -type f -name *txt
dir1//kol.txt
dir1//sun.txt
```

. Buscar por un grupo propietario en particular

```
Miguel@MacBook:$ find /var/ftp -group admin -type d
/var/ftp/site
```

. Encontrar todos los archivos modificados en la última hora

```
[root@mail-old ~]$ find /var/log/ -mmin -60
/var/log/clamav/clamd.log
/var/log/secure
/var/log/wtmp
/var/log/lastlog
/var/log/maillog
/var/log/cron
/var/log/sa/sa03
```

. Encontrar todos los archivos de hasta 50 MG de tamaño

```
[root@mail-old ~]$ find /var/tmp/ -size 50M
/tmp/nohup
```

. Realizar una búsqueda y ejecutar un comando sobre ella

```
Miguel@MacBook:$ find . -type f -name *txt -exec ls -l {} \;
-rwxrwxr-x  1 Miguel  staff  0 Feb  3 16:52 ./dir1/kol.txt
-rw-r--r--  1 Miguel  staff  21 Feb  3 17:21 ./dir1/sun.txt
-rwxrwxr-x  1 Miguel  staff  0 Feb  3 16:51 ./dir3/hup.txt
```

Como fue posible ver el comando *find* nos pemirte realizar busquedas de forma bastante simple e incluir una serie de criterios que nos ayudan a que esta sea lo más eficaz posible. Todas estas salidas las podemos complementar con otros comandos (sort, awk, cut, etc).

tr es un filtro que nos permite cambiar una determinada información de un archivo por otra. Cambia cada uno de los caracteres especificados en el conjunto inicial por los caracteres especificados en el conjunto final.
El fichero de origen o fichero destino lo especificamos con los caracteres de redirección: < ó >.

Veamos un par de ejemplos o tres:

tr ':' ' ' < /etc/passwd > ficheropasswd
tr '[a-z]' '[A-Z]' <> listaalumnosmayusculas
tr ' ' '\n' <> lineasusuarios
tr -s " " <> prueba2
Parámetros útiles:

- -s : Sustituye un conjunto de caracteres repetidos por uno sólo. Es muy útil cuando hay secuencias de caracteres que queremos borrar:

tr -s " " <> ficherodestino

- -c : Hace que se traduzcan todos los caracteres que no se encuentren especificados en el primer parámetro. En el siguiente ejemplo se traduce por una ? todo lo que no sean letras o números.

tr -c '[a-z][A-Z][0-9]' ? <> ficherodestino

- -d : Borra los caracteres que especifiquemos.

tr -d '[a-z][0-9]' ? <> ficherodestino

dd: clona y graba discos duros fácilmente

El comando dd (Dataset Definition), es una herramienta sencilla, útil, y sorprendente, a la vez que desconocida por muchos. Esta aplicación fue creada a mediados de los 70, en principio para Unix, simplemente porque no existía. Pero al contrario que otras herramientas que desde su creación se han ido sofisticando, ésta se ha ido simplificando, hasta el punto de poder hacer lo mismo que buenos programas

comerciales como Norton Ghost o libres como CloneZilla, con sólo una pequeña orden en la línea de comandos.

Ni que decir tiene que toda la información de dd la podéis consultar con el comando *man dd* e *info dd*, también dos grandes olvidados…

Al lío…

Lo primero siempre es tener claro el disco duro de origen y el de destino, algo que averiguamos fácilmente con el comando (como root) *fdisk -l*.

La sintaxis más básica, sería ésta [como root]:

dd if=[origen] of=[destino]

Por lo que si quisiéramos clonar un disco duro:

dd if=/dev/hda of=/dev/hdb bs=1M con esto clonaríamos el disco hda en hdb. (discos IDE)

O:

dd if=/dev/sda of=/dev/sdb bs=1M para discos SATA

Con *bs=1M*, estamos diciendo que tanto la lectura como la escritura se haga en bloques de 1 megabyte (menos, sería más lento pero más seguro, y con más nos arriesgamos a perder datos por el camino).

Hay que tener en cuenta que de esta forma grabarás el disco “tal cual”, MBR, tabla de particiones, espacio vacío, etc., por lo que sólo podrás grabar en un disco del mismo o mayor tamaño.

Vamos a ver algunos ejemplos prácticos y opciones de este comando:

dd if=/dev/hda1 of=/dev/hdb bs=1M

Grabaríamos sólo la primera partición del disco de origen en el de destino.

dd if=/dev/hda of=/dev/hdb1 bs=1M

Grabaríamos el disco completo en la primera partición del disco de destino.

dd if=/dev/hda of=/home/hda.bin

Crear una imagen del disco duro, puede ser bin o iso (a partir de ahora utilizaré nuestro home como ejemplo). Como root:

dd if=/dev/hda | gzip > /home/hda.bin.gz

Crearíamos con el anterior comando una imagen del disco comprimida, (podemos utilizar gzip, bzip o bzip2.)

Crea una imagen de un CD:

dd if=/dev/cdrom of=/home/imagendeCD.iso

Para montar la imagen del CD:

mount -o loop imagedeCD.iso /mnt/home

Copiar el Master Boot Record:

dd if=/dev/hda of=mbr count=1 bs=512

Para restaurar el MBR:

dd if=mbr of=/dev/hda

Copiar el Volume Boot Sector (VBS):

dd if=/dev/hda of=/home/sector_arranque_hda count=1 bs=512

Para restaurar el VBS:

dd if=/home/sector_arranque_hda of=/dev/hda

Algunas curiosidades:

Recuperar un DVD rayado:

dd if=/dev/cdrom of=/home/dvd_recuperado.iso conv=noerror,sync

Esto no recupera todo el DVD, en este caso, sólo los sectores legibles. Sirve también para discos duros defectuosos.
La opción *noerror* sirve para obviar los errores de lectura en cualquier situación. Otro ejemplo sería:

dd conv=noerror if=/dev/hda of=~/home/imagen_disco_con_errores.iso

Grabaríamos con ello una imagen del disco duro en nuestro home saltándonos los errores del disco (muy útil para discos que se están muriendo).

Limpia nuestro MBR y la tabla de particiones:

dd if=/dev/zero of=/dev/hda bs=512 count=1

Limpia el MBR pero no toca la tabla de particiones (muy útil para borrar el GRUB sin perder datos en las particiones):

dd if=/dev/zero of=/dev/hda bs=446 count=1

Crea un archivo vacío de 1 Mb, una opción muy interesante como ahora veremos:

dd if=/dev/zero of=archivo_nuevo_vacio bs=1024 count=1024

Crear un archivo swap de 2Gb así de fácil:

sudo dd if=/dev/zero of=/swapspace bs=4k count=2048M
mkswap /swapspace
swapon /swapspace

Al borde de la paranoia… Convierte todas las letras en mayúsculas:

dd if=miarchivo of=miarchivo conv=ucase

Cambia en todo el disco, la palabra Puigcorbe por Slqh, (puedes cambiar rápidamente tu nombre a todos los archivos del disco):

dd if=/dev/sda | sed ‘s/Puigcorbe/Slqh/g’ | dd of=/dev/sda

Llena el disco con caracteres aleatorios cinco veces. No va a quedar ni rastro de información en el disco :

for n in {1..5}; do dd if=/dev/urandom of=/dev/hda bs=8b conv=notrunc; done

(*) Nota final: utiliza este comando con precaución, y asegúrate siempre del orden y nombre de tus discos duros, porque lo mismo que te clona un disco te lo borra ‘en un plis’.