Guía de campo `grep` y `sed`

Y *un poco* de expresiones regulares

Autor: Martín René Vilugrón – mrvilugron@unrn.edu.ar

Como siempre, podemos consultar con los manuales autoritativos
https://www.gnu.org/software/grep/manual/grep.html

https://www.gnu.org/software/sed/manual/sed.html

Pueden usar la herramienta https://regexr.com/ para probar las expresiones y obtener una ayuda visual más completa.

Tengan en cuenta que existen múltiples dialectos de expresiones regulares por lo que va a ser necesario probar antes de usar.

¿Que es grep?

Dado uno o más patrones, `grep` busca en los archivos de entrada las coincidencias con los patrones. Cuando encuentra una coincidencia en una línea, la copia a `STDOUT` (por defecto), o produce cualquier otro tipo de salida que haya solicitado con las opciones.

Aunque `grep` espera hacer la coincidencia en el texto, no tiene límites en la longitud de la línea de entrada, aparte de la memoria disponible, y puede coincidir con caracteres arbitrarios dentro de una línea. Si el último byte de un archivo de entrada no es una nueva línea, `grep` proporciona una de forma silenciosa. Dado que la nueva línea es también un separador para la lista de patrones, no hay forma de hacer coincidir caracteres de nueva línea en un texto.

Invocando la herramienta

Entonces, si tenemos un archivo llamado `nombres.txt`:

```
$> cat nombres.txt
Sara
```

```
Miguel
Juan
Juan Pablo
Esteban
Cristina
```

Podemos buscar dentro de la siguiente manera, (en la siguiente sección se explica que es `-F`)

```
$> grep -F Miguel nombres.txt
```

Lo que nos retorna una sola línea:

```
Miguel
```

Aunque técnicamente no es necesario en este ejemplo, también posible buscar vía `STDIN`

```
$> cat nombres.txt | grep -F Miguel
```

Obteniendo el mismo resultado:

```
Miguel
```

Lo potente y más difícil de controlar de `grep` está en buscar por coincidencias parciales.

```
$> grep -F Juan nombres.txt
```

Acá obtenemos dos resultados:

```
Juan
Juan Pablo
```

Ya que la palabra "`Juan`" se encuentra en dos líneas.

Las opciones más importantes

El uso básico de la herramienta consiste en indicar la palabra o frase que queremos buscar:

```
$> grep "frase a buscar" archivo
```

Tenemos una amplia variedad de opciones para utilizar, esta no es una lista exhaustiva.

- `-i` :ignorar si es mayúscula o minúscula ("case insensitive")

Obtenemos la misma coincidencia con "`frase`" que con "`FRASE`"

- `-v` :invertir la coincidencia, mostrando lo que no coincide.

En el ejemplo anterior, obtenemos **toda** la lista menos "`John`"

- `-c` :muestra cuantas veces fue encontrado

Obtenemos cuantas veces aparece la frase a buscar, en el ejemplo anterior, 1.

- `-F` :no usar expresiones regulares, interpretar literalmente el patrón.

A continuación vamos a ver que es esto del patrón, pero es una forma de armar conjuntos de caracteres para buscar y es donde radica el poder de la herramienta.

- `-x` :solo coincidencias exactas

`Juan` no es lo mismo que `Juan Pablo`

- `-m número`:detener la búsqueda luego de `número` veces de ser encontrado.
- `-n` : mostrar el número de línea de la coincidencia.
- `-q` / `--quiet`: Silencioso

No envía nada a `STDOUT`, pero sale con estado `0` en `$?` si fue encontrado y `1` si no.

- `-A líneas`: Muestra contexto al agregar `líneas` antes de la coincidencia.
- `-B líneas`: Muestra contexto al agregar `líneas` después de la coincidencia.
- `-o` : mostrar solo lo que coincide de la línea.

Si estamos buscando `an` en la lista de nombres, obtendremos dos coincidencias, un `an` por `Juan` y otro `an` por `Juan Pablo`.

- `-w` : encontrar coincidencias de la palabra completa rodeada de espacios

Expresiones regulares

Además de buscar palabras exactas (`-F`), `grep` puede hacer uso de Expresiones Regulares, que son una forma de expresar patrones de texto.

Las expresiones regulares se construyen de manera análoga a las expresiones aritméticas, utilizando varios operadores para combinar expresiones más pequeñas.

Y aunque los ejemplos a continuación emplean `grep`, también aplican a `sed` y a un gran conjunto de herramientas como editores de textos y herramientas de oficina como planillas de cálculo.

Componentes de una RegEx

En las expresiones regulares, los caracteres '`.?*+{|()[\^$`' son caracteres especiales y tienen los usos que se describen a continuación. Todos los demás caracteres son caracteres ordinarios, y cada carácter ordinario es una expresión regular que coincide consigo misma.

Como regla general, piensen en los patrones como una forma de expresar patrones sobre una sola línea o una fracción de una.

Secuencia de escape

`\` tomar literalmente el carácter siguiente, para separar los símbolos que se usan en la RegEx misma de lo que uno quiere expresar; para buscar algo que necesite de un `\`, es necesario usar `\\` para evitar que la `\` sea tomada como instrucción.

El otro uso tiene que ver con la indicación de caracteres no directamente visibles, como tabuladores (`\t`), fines de línea; newline (`\n`), retorno de carro (`\r`), entre otros.

Esto es conocido en Inglés como “Escape Sequence”. Tengan en cuenta que si es necesario que su patrón tenga coincidencias con cualquier carácter de esta lista, tendrá que ser “escapado” con `\`.

Anclas:

- `^` :coincide al inicio de una línea

En `asa`: `^a` coincide con la primera `a` y no la segunda: `asa`.

- `$` :coincide al final de una línea

En `asa`: `a$` coincide con la segunda `a` y no la primera: `asa`.

- `\b` :coincidencia en los límites de una palabra.

En `pasa`, `a\b` coincide con la segunda `a`, no con la primera.

- `\B` :coincidencia interna en una palabra.

En `masa`, `a\B` coincide con la primera `a` y no con la segunda.

Grupos de captura

`(` y `)` Esto crea un grupo que será usado repetir la búsqueda en otro lugar del patrón que no necesariamente tiene que ser a continuación. Como es posible crear mas de un grupo, estos se referencian por número, a partir del paréntesis más a la izquierda y van hacia la derecha. Es posible crear un grupo dentro de otro.

`\n` Siendo `n` el número del grupo de captura del `1` al `9`, al usarlo, estamos indicando que estamos buscando lo que se encontró en el patrón de captura con anterioridad en otro lugar.

```
$ echo baba | grep -E "(ba)\1"
baba
```

La primera `ba` arma el grupo de captura y `\1` busca la misma coincidencia (`ba`) a continuación.

Si ven un error referido a que no hay una referencia, esto significa que el patrón capturado no obtuvo coincidencia.

> Su uso requiere de la opción `-E` para activar las expresiones regulares extendidas. O utilizar `\(` y `\)`, el ejemplo anterior quedaría `\(ba\)\1`.

Conjuntos

`[ ]`: conjunto de caracteres individuales; coinciden los caracteres individuales que están indicados.
Por ejemplo: `[abcde]` coincide `a`, `b`, `c`, `d` y `e`.
`[^]` : negación de conjunto de caracteres; coinciden los caracteres que no están indicados.
`-` (guion): El guion indica rango en un conjunto de caracteres, por ejemplo `[a-e]` coincide `a`, `b`, `c`, `d` y `e`. (es posible combinarlo con la negación. Se pueden usar múltiples conjuntos con rango. `[a-cf-h]` para `a`, `b`, `c`, `f`, `g` y `h`.
`.` :coincide con un carácter cualquiera, menos el de fin de linea.

Cuantificadores y alternaciones

`+` :repite uno o más la coincidencia de lo inmediatamente a la izquierda.
`*` :repite cero o más la coincidencia de lo inmediatamente a la izquierda.
`{x,y}` :coincide entre `x` e `y` veces lo inmediatamente a la izquierda. `A{1,3}` coincide `A`, `AA`, `AAA`
`{x}` :coincide exactamente `x` veces lo inmediatamente a la izquierda. `A{3}` coincide `AAA`
`{x,}` :coincide más de `x` veces lo inmediatamente a la izquierda. `A{1,}` coincide `A`, `AA`, `AAA`, `AAAA`, etc.
`|` : coincide uno **u** otro; por ejemplo, `ab|ba` coincide `ab` o `ba`.

Conjuntos predefinidos

Estos conjuntos predefinidos son la extensión de Conjuntos.

`[[:alnum:]]`: cualquier carácter alfanumérico, equivale a `[a-zA-Z0-9]`
`[[:alpha:]]`: cualquier carácter alfabético, equivale a `[a-zA-Z]`
`[[:contrl:]]`: cualquier carácter de control
`[[:digit:]]`: cualquier número, equivale a `[0-9]`
`[[:xdigit:]]`: cualquier número hexadecimal, equivale a `[0-9a-f]`
`[[:lower:]]`: cualquier carácter en minúscula, equivale a `[a-z]`
`[[:upper:]]`: cualquier carácter en mayúscula, equivale a `[A-Z]`
`[[:print:]]`: cualquier carácter que se pueda imprimir.
`[[:blank:]]`: Los caracteres de espacio y tabulacion.
`[[:space:]]`: cualquier carácter de espacio, incluyendo saltos de línea, tabuladores, espacios, 'form feed' y otros.

Algunos ejemplos con `grep`

Con el archivo:

```
$>cat archivo.txt
```

Que contiene:

```
Stewart
Christina
- 441
<a href="https://www.google.com">Google</a>
 Jill
54r4h
Shazbot123
lll
221
Item 1, Item 2, Item 3
TABS	TABS	TABS
```

> ! La última línea, tiene tabuladores, que a pesar de estar como espacios en el apunte, son un único carácter.
> ■ Así como los otros resaltes en color, son para hacer visibles caracteres que son normalmente invisibles y aquellos a los que tenemos que prestarle atención.

Insensible a mayúsculas y minúsculas

```
$>grep -i jill archivo.txt
```

obtenemos

```
Jill
```

Obteniendo lo que no coincide

```
$>grep -v jill archivo.txt
```

obtenemos

```
Stewart
Christina
- 441
<a href="https://www.google.com">Google</a>
54r4h
Shazbot123
lll
221
Item 1, Item 2, Item 3
TABS    TABS    TABS
```

Buscando en varios archivos

Podemos buscar en varios archivos simultáneamente, simplemente utilizamos un `glob` para indicar el grupo de archivos.

```
$>grep -i password archivo*.txt
```

Esto nos mostrará algo como:

```
archivo1.txt: password
archivo2.txt: PASSword
```

Buscar lineas que contengan números

Acá podemos usar uno de los atajos de del conjunto `[:digit:]` predefinido.

```
$>grep '[[:digit:]]' names.txt
```

obtendremos

```
 - 441
54r4h
Shazbot123
lll
221
Item 1, Item 2, Item 3
```

> **!** Un detalle muy importante es la cantidad de coincidencias obtenidas; por más de que la herramienta muestre las líneas, la coincidencia es a cada número individual (**¡18 coincidencias!**).

Buscar líneas que comiencen con números

Podemos expresar este patrón utilizando el ancla 'inicio de línea' ^ seguido del conjunto de dígitos

```
$>grep '^[[:digit:]]' archivo.txt
```

obtendremos

```
54r4h
221
```

Búsqueda exacta

En este ejemplo, utilizamos un pipe para pasar un texto

```
$>echo "Jason" | grep -x son
```

Y no obtenemos nada, no hay coincidencias.

```

```

Encontrar direcciones IP

```
$>grep '[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}' direcciones.txt
```

obtenemos

```
23.44.124.67
172.16.23.1
10.10.2.21
```

Mostrar lineas alrededor de la coincidencias

Podemos usar `-A` para las que vengan a continuación

```
$>grep -A 2 tres numeros.txt
```

Obteniendo

```
tres
cuatro
cinco
```

También podemos usar `-B` para mostrar lo que esta antes, o `-C` para mostrar lo que esta antes y después; las líneas que rodean la coincidencia.

`sed`, el filtro en el caño

`sed` es un editor de textos programable que podemos insertar en una secuencia de caños. La programación de las modificaciones pueden emplear expresiones regulares, permitiendo todo tipo de cambios en el texto, como separar en líneas, buscar y reemplazar.

Esta herramienta puede ser usada para buscar sin modificar la salida, de la misma manera que `grep`, pero es preferible mantener todo lo más simple posible.

La herramienta admite *scripts* en su propio lenguaje y múltiples comandos para modificar archivos, aquí veremos solo lo básico.

El comando esencial: `s` para la sustitución

El comando de sustitución cambia todas las ocurrencias de la expresión regular con un nuevo valor. Un ejemplo sencillo es cambiar `dia` en el archivo `viejo` por `noche` en el archivo `nuevo`:

```
$> sed s/dia/noche/ viejo >nuevo
```

y para los que quieran probar esto

```
$> echo day | sed s/day/night/
```

Esto dará como resultado `night`.

> ! No puse comillas alrededor del argumento porque este ejemplo no las necesitaba. Sin embargo, te recomiendo que uses siempre comillas. Cuando el comando es más complejo, es casi seguro que contendrá caracteres que la consola intentará interpretar.

Usando comillas simples, sería

```
$> sed 's/día/noche/' < viejo > nuevo
```

Es importante remarcar que `sed` cambia exactamente lo que le indiques, por lo que si ejecutás:

```
$> echo domingo | sed 's/min/max/'
```

Esto mostraría la palabra `domaxgo` porque sed encontró la cadena `min` en la entrada.

Otro concepto importante es que sed está *orientado a trabajar con líneas*. Suponga que tiene el archivo de entrada

```
one two three, one two three
four three two one
one hundred
```

y ha utilizado el comando

```
$> sed 's/one/ONE/' <file
```

La salida sería

```
ONE two three, one two three
four three two ONE
ONE hundred
```

Tenga en cuenta que esto cambió `one` a `ONE` una vez en cada línea. La primera línea tenía `uno` dos veces, pero sólo se cambió la primera ocurrencia. Este es el comportamiento por defecto. Si quiere algo diferente, tendrá que usar algunas de las opciones que están disponibles. Las explicaré más adelante.

Así que continuemos.

Hay cuatro partes en este comando de sustitución:

`s/one/ONE/`	
`s`	Comando de sustitución
`/../../`	Delimitadores
`one`	Patrón de Expresión Regular Patrón de Búsqueda
`ONE`	Cadena de sustitución

El patrón de búsqueda está en la parte izquierda y la cadena de sustitución está en la parte derecha.

Hemos cubierto las comillas y las expresiones regulares. Eso ya es el 90% del esfuerzo necesario para aprender el comando de sustitución. Para decirlo de otra manera, usted ya sabe cómo manejar el 90% de los usos más frecuentes de sed. Hay unos ... pocos puntos finos que cualquier futuro experto en sed debería conocer.

La barra inclinada como delimitador

El carácter que sigue a la `s` es el delimitador. Es convencionalmente una barra, porque es lo que usan `ed`, `more` y `vi`. Sin embargo, puede ser cualquier cosa que desee. Si quiere cambiar un nombre de ruta que contiene una barra - digamos `/usr/local/bin` a `/common/bin` - puede usar la barra invertida para citar la barra:

```
$> sed 's/\/usr\/local\/bin/\/common\/bin/' <viejo >nuevo
```

Gulp. Algunos llaman a esto 'Picket Fence' y es feo. Es más fácil de leer si utiliza un subrayado en lugar de una barra como delimitador:

```
$> sed 's_/usr/local/bin_/common/bin_' <viejo >nuevo
```

Algunos utilizan dos puntos:

```
$> sed 's:/usr/local/bin:/common/bin:' <viejo >nuevo
```

Otros utilizan el carácter "|".

```
sed 's|/usr/local/bin|/common/bin|' <viejo >nuevo
```

Elige uno que te guste. **Mientras no esté en la cadena que buscas, todo vale.** Y recuerda que necesitas tres delimitadores. Si obtienes un `"unterminated `s' command"` es porque te falta uno.

Usar & como cadena de coincidencia

A veces se quiere buscar un patrón y añadir algunos caracteres, como paréntesis, alrededor o cerca del patrón encontrado. Es fácil hacer esto si está buscando una cadena en particular:

```
$> sed 's/abc/(abc)/' <viejo >nuevo
```

Esto no funcionará si no sabes exactamente lo que vas a encontrar. ¿Cómo puede poner la cadena que ha encontrado en la cadena de reemplazo si no sabe lo que es?

La solución requiere el carácter especial "&". Corresponde al patrón encontrado.

```
$> sed 's/[a-z]*/(&)/' <viejo >nuevo
```

Puede tener cualquier número de "&" en la cadena de sustitución. También puede duplicar un patrón, por ejemplo, el primer número de una línea:

```
$> echo "123 abc" | sed 's/[0-9]*/& &/'
123 123 abc
```

En sed, las búsquedas tienen que ser tan precisas como la modificación, por lo que es necesario establecer límites exactos a las coincidencias, limitando la “codicia” de la expresión.

Permítame modificar ligeramente este ejemplo. Sed coincidirá con la primera cadena, y la hará lo más codiciosa posible. Lo explicaré más adelante. Si no quiere que sea tan codicioso (es decir, que limite las coincidencias), necesita poner restricciones a la coincidencia.

La primera coincidencia para `[0-9]*` es el primer carácter de la línea, ya que coincide con cero o más números. Así que si la entrada fuera `abc 123` la salida no cambiaría (bueno, excepto por un espacio antes de las letras). Una mejor manera de duplicar el número es asegurarse de que coincide con un número:

```
$> echo "123 abc" | sed 's/[0-9][0-9]*/& &/'
123 123 abc
```

La cadena `abc` no se ha modificado, porque no ha sido igualada por la expresión regular. Si quiere eliminar `abc` de la salida, debe ampliar la expresión regular para que coincida con el resto de la línea y excluir explícitamente parte de la expresión utilizando `(`, `)` y `\1`, que es lo que viene a continuación.

Expresiones regulares extendidas

Quisiera añadir un comentario rápido aquí porque hay otra forma de escribir el script anterior. `[0-9]*` coincide con cero o más números. `[0-9][0-9]*` coincide con uno o más números. Otra forma de hacer esto es usar el metacarácter `+` y usar el patrón `[0-9]+`, ya que el `+` es un carácter especial cuando se usan "expresiones regulares extendidas". Estas poseen una mayor potencia, pero los scripts sed que trataran el `+` como un carácter normal se romperían. Por lo tanto, debe habilitar explícitamente esta extensión con una opción de línea de comandos.

GNU sed activa esta característica si se utiliza la opción de línea de comandos `-r`. Así que lo anterior también podría escribirse usando

```
$> echo "123 abc" | sed -r 's/[0-9]+/& &/'
123 123 abc
```

Uso de \1 para mantener parte del patrón

En sed podemos utilizar Grupos de captura para conservar una parte de la coincidencia de la expresión regular, en donde el `\1` es el primer patrón recordado, y el `\2` es el segundo patrón recordado. Sed tiene hasta nueve patrones recordados.

Si quiere mantener la primera palabra de una línea, y borrar el resto de la línea, marque la parte importante con el paréntesis:

```
$> sed 's/\([a-z]*\).*/\1/'
```

Debería explicar esto con más detalle. Las expresiones regulares son codiciosas, e intentan coincidir con todo lo posible. La expresión regular `[a-z]*` coincide con cero o más letras minúsculas, e intenta coincidir con el mayor número de caracteres posible. El `.*` coincide con

cero o más caracteres después de la primera coincidencia. Dado que la primera coge todas las letras minúsculas contiguas, la segunda coincide con cualquier otra cosa. Por lo tanto, si escribe

```
$> echo abcd123 | sed 's/\([a-z]*\).*/\1/'
```

Esto mostrará `abcd` y borrará los números.

Si quiere cambiar dos palabras, puede recordar dos patrones y cambiar el orden

```
$> sed 's/\([a-z]*\\) \([a-z]*\)/\2 \1/'
```

Observe el espacio entre los dos patrones recordados. Esto se utiliza para asegurarse de que se encuentran dos palabras. Sin embargo, esto no hará nada si se encuentra una sola palabra, o cualquier línea sin letras. Puede insistir en que las palabras tengan al menos una letra utilizando

```
$> sed 's/\([a-z][a-z]*\) \([a-z][a-z]*\)/\2 \1/'
```

o utilizando expresiones regulares extendidas (tenga en cuenta que `(` y `)` ya no necesitan tener una barra invertida):

```
$> sed -r 's/([a-z]+) ([a-z]+)/\2 \1/' # Usando GNU sed
```

El `\1` no tiene que estar en la cadena de sustitución (en la parte derecha). Puede estar en el patrón que está buscando (en la parte izquierda). Si quiere eliminar las palabras duplicadas, puede probar

```
$> sed 's/\([a-z]*\\) \1/\1/'
```

Si quiere detectar palabras duplicadas, puede utilizar

```
$> sed -n '/\([a-z][a-z]*\) \1/p'
```

o con expresiones regulares extendidas

```
$> sed -rn '/([a-z]+) \1/p'
```

Esto, cuando se usa como filtro, imprimirá las líneas con palabras duplicadas.

El valor numérico puede tener hasta nueve valores: `\1` a `\9`. Si desea invertir los tres primeros caracteres de una línea, puede utilizar

```
sed 's/^\(.\)\(.\)\(.\)/\3\2\1/'
```

Indicadores del patrón Sed

Puede añadir banderas adicionales después del último delimitador. Habrás notado que usé una '`p`' al final del comando sustituto anterior. También añadí la opción '`-n`'. Primero cubriremos la '`p`' y otras banderas de patrón. Estas banderas pueden especificar lo que sucede cuando se encuentra una coincidencia.

/g - Reemplazo global

La mayoría de las utilidades de UNIX trabajan sobre archivos, leyendo una línea a la vez. Sed, por defecto, es de la misma manera. Si le dice que cambie una palabra, sólo cambiará la primera ocurrencia de la palabra en una línea. Usted puede querer hacer el cambio en cada palabra de la línea en lugar de la primera.

Por ejemplo, pongamos paréntesis alrededor de las palabras de una línea. En lugar de usar un patrón como "`[A-Za-z]*`" que no coincidirá con palabras como "no", usaremos un patrón, "`[^ ]*`", que coincide con todo excepto con un espacio. Bueno, esto también coincidirá con cualquier cosa porque "`*`" significa cero o más.

Como solución, hay que evitar que la cadena nula coincida con la bandera "`g`" de sed. Un ejemplo de solución es: "`[^ ][^ ]*.`" Lo siguiente pondrá paréntesis alrededor de la primera palabra:

```
sed 's/[^ ]*/(&)/' <antiguo >nuevo
```

Si quiere que haga cambios para cada palabra, añada una "`g`" después del último delimitador y utilice la solución

```
sed 's/[^ ][^ ]*/(&)/g' <viejo >nuevo
```

¿Sed es recursivo?

Sed sólo opera sobre los patrones encontrados en los datos de entrada. Es decir, se lee la línea de entrada, y cuando se encuentra un patrón, se genera la salida modificada, y se escanea el resto de la línea de entrada. El comando s no escanea la salida recién creada. Es decir, no tiene que preocuparse por expresiones como

```
$> sed 's/lazo/lazo del lazo/g' <viejo >nuevo
```

Esto no causará un bucle infinito. Si se ejecuta un segundo comando s, podría modificar los resultados de un comando anterior.

/1, /2, etc. Especificación de la ocurrencia

Sin banderas, se modifica la primera sustitución coincidente. Con la opción "g", se modifican todas las coincidencias. Si quiere modificar un patrón en particular que no es el primero de la línea, puede usar \(y \) para marcar cada patrón, y usar \1 para poner el primer patrón sin cambios. El siguiente ejemplo mantiene la primera palabra en la línea, pero borra la segunda:

```
$> sed 's/\([a-zA-Z]*\) \([a-zA-Z]*\) /\1 /' <viejo >nuevo
```

Qué asco. Hay una manera más fácil de hacer esto. Puedes añadir un número después del comando de sustitución para indicar que sólo quieres que coincida con ese patrón en particular. Por ejemplo:

```
$> sed 's/[a-zA-Z]* //2' <viejo >nuevo
```

Puedes combinar un número con la bandera g (global). Por ejemplo, si quiere dejar la primera palabra, pero cambiar la segunda, tercera, etc. para que sean BORRADAS, utilice /2g:

```
$> sed 's/[a-zA-Z]* /BORRADAS /2g' <viejo >nuevo
```

No confundir /2 y \2. El /2 se utiliza al final. El \2 se usa dentro del campo de reemplazo.

```
$> sed 's/[^ ]*//2' <viejo >nuevo
```

pero esto también se come CPU. Si esto funciona en tu ordenador, y lo hace en algunos sistemas UNIX, podrías eliminar la contraseña encriptada del archivo de contraseñas:

```
sed 's/[^:]*//2' </etc/passwd >/etc/password.new
```

Pero esto no me funcionó la vez que escribí esto. Usar `[^:][^:]*` como solución no ayuda porque no coincidirá con una contraseña inexistente, y en su lugar borrará el tercer campo, ¡que es el ID del usuario! En su lugar, tiene que utilizar el feo paréntesis

```
sed 's/^\([^:]*\):[^:]:/\1::/'  </etc/passwd >/etc/password.new
```

También puedes añadir un carácter al primer patrón para que no coincida con el patrón nulo:

```
sed 's/[^:]*:/:/2'  </etc/passwd >/etc/password.new
```

El indicador de número no está restringido a un solo dígito. Puede ser cualquier número del `1` al `512`. Si quieres añadir dos puntos después del carácter `80` en cada línea, puedes escribir

```
sed 's/./&:/80' <viejo >nuevo
```

También puede hacerlo de la manera más difícil, utilizando 80 puntos:

```
sed
's/^.............................................................................
......./&:/' <viejo >nuevo
```

Comandos múltiples

Anteriormente, sólo hemos utilizado un comando de sustitución. Si necesitas hacer dos cambios, y no quieres leer el manual, puedes juntar varios comandos sed:

```
sed 's/BEGIN/begin/' <viejo | sed 's/END/end/' >nuevo
```

Esto tal vez sea ineficiente y exista una forma de hacerlo en una sola llamada a `sed`, pero para mantener el apunte simple no lo trataremos aquí.

Conclusión

Las expresiones regulares son una herramienta sumamente poderosa, pero que muy rápidamente pueden transformarse en bloques de símbolos difíciles de leer [1].

Esto no es un mensaje para dejar de utilizarlas, es simplemente una advertencia. De no abusar de esta poderosa herramienta .

Bibliografía

[1] *Regular expressions: Now you have two problems*. (2008, junio 27). Coding Horror. https://blog.codinghorror.com/regular-expressions-now-you-have-two-problems/
[2] *Sed, a stream editor*. (s. f.). Recuperado 7 de abril de 2022, de https://www.gnu.org/software/sed/manual/sed.html
[3] *The absolute bare minimum every programmer should know about regular expressions—I'm Mike*. (2009, febrero 9). http://web.archive.org/web/20090209182018/http://immike.net/blog/2007/04/06/the-absolute-bare-minimum-every-programmer-should-know-about-regular-expressions/
[4] *Top[GNU Grep 3.7]*. (s. f.). Recuperado 7 de abril de 2022, de https://www.gnu.org/software/grep/manual/html_node/index.html
[5]