Esercizi con `grep`, `sed`

1.usa `grep` per selezionare tutte le linee che contengono la parola “when” oppure “ When”

grep ‘[Ww]hen’ file

2.Usa `sed` per selezionare tutte le linee che contengono la parola “when” oppure “ When”

Sed –n ‘/[Ww]hen/p’ file

3.Stampa tutte le linee che iniziano con una lettera maiuscola usando Grep

Grep ‘^[A-Z]’ file

4.Stampa tutte le linee che ***non*** contengono come prima lettera una lettera maiuscola usando Grep

Grep –v ‘^[A-Z]’ file

5.Converti tutte le lettere”C” maiuscole in “c” minuscole usando Sed

Sed –e ‘s /C/c/g’ file

6.Usa `sed` per sostituire ALTRI con STUDENT

Sed –e ‘s /altri/studenti/g’ file

7.Cancella i primi tre caratteri di ciascuna linea del file passwd usando Sed

Sed –e ‘s /???/---/g’ passwd > file.1

8.Usa `sed` per rimuovere tutte le linee che contengono la parola STUDENT

Sed –e ‘s/student/d’ file

9.Stampa con Grep solo le linee 5 e 7 di un file

Grep [5,7] file

10.Ordina con Sort il testo di un file in ordine inverso

Sort –r file

11. Ordina con Sort il testo di un file sulla base del secondo campo

Sort +2 file

12.Lista il file passwd utilizzando Sed

Sed –e 1 /etc/passwd

13.Stampa una lista di utenti che abbiano “F” come primo carattere

14.Stampa una lista di utenti indicando per ciascuno di essi la *line number* del file passwd

15.Stampa il valore del codice utente più piccolo e più grande presenti nel file passwd

grep

Da Wikipedia, l'enciclopedia libera.

grep (general regular expression print) è un comando dei sistemi Unix e Unix-like, e più in generale dei sistemi POSIX[1] e GNU[2], che ricerca in uno o più file di testo le linee che corrispondono ad uno o più modelli specificati con espressioni regolari o stringhe letterali, e produce un elenco delle linee (o anche dei soli nomi di file) per cui è stata trovata corrispondenza. È un tipo di filtro.

grep è comunemente utilizzato per ricercare le occorrenze di una o più parole in una serie di file, spesso in congiunzione con i comandi find e xargs tramite una *pipeline software*.

grep può generalmente essere impiegato anche con file binari, ad esempio per ricercare la presenza di particolari etichette Exif all'interno di file contenenti immagini digitali. In particolare, la versione GNU di *grep* in questo caso elenca i nomi dei file contenenti la stringa o espressione regolare indicata (e non anche la porzione di testo corrispondente, come normalmente accade per i file di testo).

Indice

Uso

La sintassi generale di *grep* è:

```
grep [opzioni] [-e] modello1 [-e modello2 …] [--] [file1 [file2 …] ]
```

I parametri facoltativi *file* indicano i nomi dei file su cui effettuare la ricerca. Se non specificati, la ricerca viene effettuata sui dati letti dallo *standard input*. Specificando più di un parametro *file*, ogni linea per cui è stata trovata una corrispondenza viene preceduta dal nome del file che la contiene e dal suo numero di linea; in caso di un solo parametro *file* (o nessuno) viene invece indicato solo il contenuto della linea stessa.

I parametri *modello* specificano il criterio di ricerca, ed il comportamento predefinito prevede che si tratti di espressioni regolari. Una linea trova corrispondenza se soddisfa almeno uno dei modelli.

Il doppio trattino `--` (facoltativo) indica che i parametri successivi non sono da considerarsi opzioni.

Tra le opzioni principali vi sono:

`-i`
: Ignora le differenze tra lettere maiuscole e minuscole.

`-n`
: Precede ogni linea dei risultati con il numero di linea all'interno del file (partendo da 1).

`-l`
: Indica solo i nomi dei file in cui è stata trovata almeno una corrispondenza (ciascun file è elencato una sola volta, indipendentemente dal numero di corrispondenze in esso trovate).

`-v`
: Nega i modelli specificati, producendo un elenco delle linee che non soddisfano alcun modello.

`-E`
: I modelli sono espressioni regolari estese invece che espressioni regolari di base.

`-F`
: I modelli sono stringhe che vanno ricercate in maniera letterale.

`-c`
: Produce per ciascun file solo il conteggio del numero di linee che corrispondono.

La versione GNU di *grep* (disponibile ad esempio sui sistemi GNU/Linux) supporta tra le altre anche le seguenti opzioni:

`-`*`numero`*
: Ogni linea per cui è stata trovata una corrispondenza viene elencata insieme al *numero* specificato di linee ad essa adiacenti (per fornire il contesto). Ogni linea è elencata una sola volta, per cui anche se vi fossero corrispondenze nelle linee adiacenti, esse non saranno ripetute.

`-A` *`numero`*
: Fa seguire ogni linea per cui è stata trovata una corrispondenza dal *numero* specificato di linee che la seguono (la "A" sta per *after* - dopo).

`-B` *`numero`*
: Fa precedere ogni linea per cui è stata trovata una corrispondenza dal *numero* specificato di linee che la precedono (la "B" sta per *before* - prima).

Varianti

Storicamente esistono anche varianti di *grep* chiamate **egrep** e **fgrep**, le quali interpretano i *modelli* rispettivamente come espressioni regolari estese e come stringhe letterali.

L'uso delle opzioni `-E` e `-F` equivale all'uso di queste varianti.

Origine del nome *grep*

Il nome del programma deriva dal comando `g/re/p` dell'editor di testo ed che svolge una funzione simile, ovvero ricercare **g**lobalmente (ovvero in tutto il file e non in una sola linea) una espressione regolare (***r**egular **e**xpression*) e di mostrare (***p**rint*) le corrispondenze.

Punti di attenzione

Ancoraggio del testo da ricercare

Un'espressione regolare che sia priva di ancoraggi può trovare corrispondenza in un punto qualsiasi della linea, e quindi anche nel mezzo di una parola. Questo può essere fonte di risultati inattesi se quello che si intendeva ricercare era in realtà un'intera parola. Ad esempio l'espressione regolare "10" trova corrispondenza anche in "100", "101", "320103" e così via.

Soluzione con GNU *grep*

La versione GNU di *grep*, oltre che per gli ancoraggi a inizio e fine linea, ha supporto anche per particolari metacaratteri che rappresentano l'inizio e/o la fine di una qualsiasi parola, e possono essere usati per *ancorare* il resto dell'espressione regolare. Nello specifico la sequenza `\<` corrisponde al punto d'inizio di una parola, `\>` al punto in cui termina una parola, e `\b` al punto d'inizio o di termine di una parola. Ad esempio, l'espressione regolare `\<10\>` trova corrispondenza solo in linee che contengono "10" come parola a sè stante, in maniera visivamente isolata, e non "100" o "210".

Soluzione con espressioni regolari POSIX

Gli ancoraggi previsti dallo standard POSIX sono solo quelli che rappresentano l'inizio e la fine della linea, rispettivamente l'accento circonflesso `^` ed il simbolo del dollaro `$`; in questo caso un possibile rimedio consiste nell'estendere l'espressione regolare circondandola con `[^[:alnum:]]`, ad esempio con `[^[:alnum:]]10[^[:alnum:]]`; ciò tuttavia non copre i casi in cui vi sia corrispondenza all'inizio o alla fine della linea (in cui non vi sono caratteri precedenti o caratteri successivi) e nemmeno il caso in cui l'espressione sia l'intera linea. Per considerare anche queste situazioni occorre espandere manualmente i quattro casi, ricorrendo a più opzioni `-e`. Ad esempio la riga di comando di *grep* diventerebbe:

```
grep -e "[^[:alnum:]]10[^[:alnum:]]" -e "[^[:alnum:]]10$" -e "^10[^[:alnum:]]" -
e "^10$" …
```

Oppure si può anche ricorrere alle espressioni regolari estese (opzione di *grep* `-E`) e al metacarattere `|` per indicare più espressioni alternative, ad esempio:

```
 grep -E "[^[:alnum:]]10[^[:alnum:]]|[^[:alnum:]]10$|^10[^[:alnum:]]|^10$" …
```

Stringhe letterali cercate come espressioni regolari

Il comportamento predefinito di *grep* prevede che i modelli usati per la ricerca siano delle espressioni regolari e non stringhe letterali (per le quali occorre specificare l'apposita opzione `-F`), ma è facile scordarsi della distinzione poiché non capita spesso di dover ricercare testo contenente dei metacaratteri come il punto `.`.

Il problema è che la distinzione in realtà sussiste, e che spesso le stringhe contenenti dei metacaratteri (come ad esempio un indirizzo IP numerico `10.10.1.1`) sono anche delle valide espressioni regolari, per cui *grep* non segnala alcun errore, ma può fornire risultati del tutto inattesi. Ad esempio l'espressione regolare `10.1.1.1` trova corrispondenza anche in `10.101.1` o in `1091a1b1` o altro ancora, e unitamente al fatto che non sia ancorata aumenta la possibilità di risultati inattesi.

Una possibile soluzione consiste appunto nell'usare l'opzione `-F` in modo da effettuare ricerche letterali; ciò tuttavia impedisce di ancorare il testo (si ricorda che la stringa letterale `10.1.1.1` trova corrispondenza anche in `10.1.1.100` o anche `210.1.1.1`).

Se ciò fosse un problema, occorre ricorrere ancora una volta alle espressioni regolari, indicando che i metacaratteri vanno considerati in maniera letterale prefissandoli uno ad uno con la barra inversa `\` e poi procedendo come per il caso in cui si necessita di ancoraggio. Ad esempio, con la versione GNU di *grep*:

```
grep -e '\<10\.1\.1\.1\>' …
```

Utilizzo della barra inversa

Le *shell testuali* dei sistemi Unix e Unix-like effettuano sostituzioni sull'intera riga di comando prima di eseguirla, tra le quali vi è anche quella delle sequenze di caratteri che iniziano con una barra inversa `\` quando non sono specificate tra virgolette doppie o apici singoli. Ad esempio, la riga di comando

```
grep -e 10\.1\.1\.1
```

viene trasformata dalla shell in

```
grep -e 10.1.1.1
```

e quindi *grep* si ritroverebbe ad operare ricerche con l'espressione regolare `10.1.1.1`, che probabilmente non era nelle intenzioni originali.

È quindi opportuno specificare le espressioni regolari tra virgolette doppie o apici singoli, come ad esempio in

```
grep -e '10\.1\.1\.1'
```

Utilizzo con find

Per effettuare ricerche in più file all'interno di una gerarchia di directory si usa spesso *grep* in combinazione con il comando find, ad esempio con:

```
find . -type f -name "*.c" -exec grep -e "espressione" {} +
```

Così facendo esiste tuttavia la possibilità che *grep* sia invocato da *find* con un unico file da esaminare (ad esempio se *find* trovasse un solo file), nel qual caso *grep* procede a elencare le linee corrispondenti senza prefissarle con il nome file a cui esse appartengono (che è il comportamento predefinito nel caso di un unico file), quindi offrendo un risultato diverso da quello normalmente atteso.

Per rimediare si può esplicitare direttamente tra i parametri di *grep* il nome di un primo file, in modo che *grep* sia invocato sempre con almeno due nomi di file da esaminare. Allo scopo torna comodo `/dev/null`, che è sempre presente e non contiene mai dati, ed è quindi l'ideale come file "riempitivo" che non influenza le ricerche. Ad esempio:

```
find . -type f -name "*.c" -exec grep -e "espressione" /dev/null {} +
```

SED

Sed è un editor di riga non interattivo. Riceve un testo come input, o dallo `stdin` o da un file, esegue alcune operazioni sulle righe specificate, una alla volta, quindi invia il risultato allo `stdout` o in un file. Negli script di shell, sed è, di solito, una delle molte componenti di una pipe.

Sed determina le righe dell'input, su cui deve operare, tramite un *indirizzo* che gli è stato passato. [1] Questo indirizzo può essere rappresentato sia da un numero di riga sia da una verifica d'occorrenza. Ad esempio, *`3d`* indica a sed di cancellare la terza riga dell'input, mentre *`/windows/d`* segnala a sed che si vogliono cancellare tutte le righe dell'input contenenti l'occorrenza "windows".

Di tutte le operazioni a disposizione di sed, vengono focalizzate, in primo luogo, le tre più comunemente usate. Esse sono **p**rint (visualizza allo `stdout`), **d**elete (cancella) e **s**ubstitute (sostituisce).

Tabella C-1. Operatori sed di base

Operatore	Nome	Effetto
`[indirizzo]/p`	print	Visualizza [l'indirizzo specificato]
`[indirizzo]/d`	delete	Cancella [l'indirizzo specificato]
`s/modello1/modello2/`	substitute	Sostituisce in ogni riga la prima occorrenza della stringa modello1 con la stringa modello2

Operatore	Nome	Effetto
`[indirizzo]/s/modello1/modello2/`	substitute	Sostituisce, in tutte le righe specificate in *`indirizzo`*, la prima occorrenza della stringa modello1 con la stringa modello2
`[indirizzo]/y/modello1/modello2/`	transform	sostituisce tutti i caratteri della stringa modello1 con i corrispondenti caratteri della stringa modello2, in tutte le righe specificate da *`indirizzo`* (equivalente di **tr**)
`g`	global	Agisce su *tutte* le verifiche d'occorrenza di ogni riga di input controllata

Se l'operatore `g` (*global*) non è accodato al comando *substitute*, la sostituzione agisce solo sulla prima verifica d'occorrenza di ogni riga.

Sia da riga di comando che in uno script di shell, un'operazione sed può richiedere il quoting e alcune opzioni.

```
sed -e '/^$/d' $nomefile #  L'opzione -e indica che la stringa successiva deve
essere interpretata come #+ un'istruzione di editing. #  (se a "sed" viene
passata un'unica istruzione, "-e" è facoltativo.) #  Il quoting "forte" ('')
protegge i caratteri speciali delle ER, presenti #+ nell'istruzione, dalla
reinterpretazione da parte dello script. # (Questo riserva solo a sed
l'espansione delle ER.) # # Agisce sul testo del file $nomefile.
```

In certi casi, un comando di editing **sed** non funziona in presenza degli apici singoli.

```
nomefile=file1.txt modello=INIZIO    sed "/^$modello/d" "$nomefile"  # Funziona
come indicato. # sed '/^$modello/d' "$nomefile"    dà risultati imprevisti. #
In questo esempio, il quoting forte (' ... '), #+       impedisce a "$modello"
di espandersi a "INIZIO".
```

Sed utilizza l'opzione `-e` per indicare che la stringa che segue è un'istruzione, o una serie di istruzioni. Se la stringa contiene una singola istruzione, allora questa opzione può essere omessa.

```
sed -n '/xzy/p' $nomefile #  L'opzione -n indica a sed di visualizzare solo
quelle righe che verificano #+ il modello. #  Altrimenti verrebbero visualizzate
tutte le righe dell'input. #  L'opzione -e, in questo caso, non sarebbe
necessaria perché vi è una sola #+ istruzione di editing.
```

Tabella C-2. Esempi di operatori sed

Notazione	Effetto
`8d`	Cancella l'ottava riga dell'input.
`/^$/d`	Cancella tutte le righe vuote.
`1,/^$/d`	Cancella dall'inizio dell'input fino alla prima riga vuota compresa.
`/Jones/p`	Visualizza solo le righe in cui è presente "Jones" (con l'opzione -n).
`s/Windows/Linux/`	Sostituisce con "Linux" la prima occorrenza di "Windows" trovata in ogni riga dell'input.
`s/BSOD/stabilità/g`	Sostituisce con "stabilità" tutte le occorrenze di "BSOD" trovate in ogni riga dell'input.
`s/ *$//`	Cancella tutti gli spazi che si trovano alla fine di ogni riga.
`s/00*/0/g`	Riduce ogni sequenza consecutiva di zeri ad un unico zero.
`/GUI/d`	Cancella tutte le righe in cui è presente "GUI".
`s/GUI//g`	Cancella tutte le occorrenze di "GUI", lasciando inalterata la parte restante di ciascuna riga.

Sostituire una stringa con un'altra di lunghezza zero (nulla) equivale a cancellare quella stringa nella riga di input. Questo lascia intatta la parte restante della riga. L'espressione `s/GUI//` applicata alla riga

```
Le parti più importanti di ogni applicazione sono le sue GUI e gli effetti
sonori
```

dà come risultato

```
Le parti più importanti di ogni applicazione sono le sue e gli effetti sonori
```

La barra inversa costringe il comando di sostituzione **sed** a continuare sulla riga successiva. L'effetto è quello di usare il carattere di *a capo* alla fine della prima riga come *stringa di sostituzione*.

```
s/^  */\ /g
```

In questo modo, tutti gli spazi che si trovano all'inizio della riga vengono sostituiti con un carattere di a capo. Il risultato finale è la sostituzione di tutte le indentazioni dei paragrafi con righe vuote poste tra gli stessi paragrafi.

Un indirizzo seguito da una, o più, operazioni può richiedere l'impiego della parentesi graffa aperta e chiusa, con un uso appropriato dei caratteri di a capo.

```
/[0-9A-Za-z]/,/^$/{ /^$/d }
```

Questo cancella solo la prima di ogni serie di righe vuote. Potrebbe essere utile per effettuare la spaziatura singola di un file di testo mantenendo, però, la/e riga/he vuota/e tra i paragrafi.

Esempio 33-1. ***Shell wrapper***

```
#!/bin/bash  # Questo è un semplice script che rimuove le righe vuote da un
file. # Nessuna verifica d'argomento. # # Sarebbe meglio aggiungere qualcosa
come:  # E_NOARG=65 # if [ -z "$1" ] # then #  echo "Utilizzo: `basename $0`
nome-file" #  exit $E_NOARG # fi   # È uguale a #    sed -e '/^$/d' nomefile #
invocato da riga di comando.  sed -e /^$/d "$1" #  '-e' significa che segue un
comando di "editing" (in questo caso opzionale). #  '^' indica l'inizio della
riga, '$' la fine. #  Verifica le righe che non contengono nulla tra il loro
inizio e la fine, #+ vale a dire, le righe vuote. #  'd' è il comando di
cancellazione.  #  L'uso del quoting per l'argomento consente di #+ passare nomi
di file contenenti spazi e caratteri speciali.  #  Va notato che lo script, in
realtà, non modifica il file di riferimento. #  Se avete questa necessità,
effettuate la redirezione dell'output.  exit 0
```

Esempio 33-2. Uno ***shell wrapper*** **leggermente più complesso**

```
#!/bin/bash  #  "subst", uno script per sostituire un nome #+ con un altro
all'interno di un file, #+ es. "subst Smith Jones letter.txt".  ARG=3          #
Lo script richiede tre argomenti. E_ERR_ARG=65   # Numero errato di argomenti
passati allo script.  if [ $# -ne "$ARG" ] # Verifica il numero degli argomenti
(è sempre una buona idea). then   echo "Utilizzo: `basename $0` vecchio-nome
nuovo-nome nomefile"   exit $E_ERR_ARG fi  vecchio_nome=$1 nuovo_nome=$2  if [ -
f "$3" ] then   nome_file=$3 else   echo "Il file \"$3\" non esiste."   exit
$E_ERR_ARG fi  #  Ecco dove viene svolto il lavoro principale.  # --------------
-------------------------------- sed -e "s/$vecchio_nome/$nuovo_nome/g"
$nome_file # ----------------------------------------------  #  's' è,
naturalmente, il comando sed di sostituzione, #+ e /modello/ invoca la ricerca
di corrispondenza. #  L'opzione "g", o globale, provoca la sostituzione di
*tutte* #+ le occorrenze di $vecchio_nome in ogni riga, non solamente nella
prima. #  Leggete i testi riguardanti 'sed' per una spiegazione più
approfondita.  exit 0    # Lo script invocato con successo restituisce 0.
```

Esempio 33-3. Uno ***shell wrapper*** **generico che effettua una registrazione in un file di log**

```
#!/bin/bash #  Uno shell wrapper generico che effettua una/delle operazione/i #+
registrandola/e in un file di log.  # Si devono impostare le variabili seguenti.
OPERAZIONE= #         Può essere una serie complessa di comandi, #+        per
esempio uno script awk o una pipe . . . LOGFILE= #         File di log.
OPZIONI="$@" #         Argomenti da riga di comando, se ce ne fossero, per
operazione.  # Registrazione. echo "`date` + `whoami` + $OPERAZIONE "$@"" >>
$LOGFILE # Ora l'esecuzione. exec $OPERAZIONE "$@"  # È necessario effettuare la
registrazione prima dell'esecuzione. # Perché?
```

Esempio 33-4. Uno *shell wrapper* per uno script awk

```
#!/bin/bash # pr-ascii.sh: Visualizza una tabella di caratteri ASCII.  INIZIO=33
# Intervallo dei caratteri ASCII stampabili (decimali). FINE=125  echo "
Decimale  Esa     Carattere"   # Intestazione. echo " --------  ---     --------
-"  for ((i=INIZIO; i<=FINE; i++)) do   echo $i | awk '{printf("  %3d       %2x
%c\n", $1, $1, $1)}' # In questo contesto, il builtin Bash printf non funziona:
#     printf "%c" "$i" done  exit 0   #  Decimale  Esa     Carattere #  --------
---     --------- #    33       21         ! #    34       22         " #    35
23         # #    36       24         $ # #    . . . # #   122       7a
z #   123       7b         { #   124       7c         | #   125       7d
}   #  Redirigete l'output dello script in un file #+ o collegatelo con una pipe
a "more":  sh pr-asc.sh | more
```

Esempio 33-5. Uno *shell wrapper* per un altro script awk

```
#!/bin/bash  # Aggiunge la colonna specificata (di numeri) nel file indicato.
ARG=2 E_ERR_ARG=65  if [ $# -ne "$ARG" ] #  Verifica il corretto nr. di
argomenti da riga                        #+ di comando. then    echo "Utilizzo:
`basename $0` nomefile numero-colonna"    exit $E_ERR_ARG fi  nomefile=$1
numero_colonna=$2  #  Il passaggio di variabili di shell allo script awk
incorporato  #+ è un po' complicato. #  Un metodo consiste nell'applicare il
quoting forte alla variabile dello  #+ script Bash all'interno dello script awk.
#     $'$VAR_SCRIPT_BASH' #      ^                ^ #  È ciò che è stato fatto
nello script awk incorporato che segue. #  Vedete la documentazione awk per
maggiori dettagli.  # Uno script awk che occupa più righe viene invocato con:
awk ' ..... '   # Inizio dello script awk. # ------------------------------ awk '
{ totale += $'"${numero_colonna}"' } END {       print totale }  ' "$nomefile" #
------------------------------ # Fine dello script awk.   #   Potrebbe non essere
sicuro passare variabili di shell a uno script awk #+  incorporato, così
Stephane Chazelas propone la seguente alternativa: #   --------------------------
-------------- #   awk -v numero_colonna="$numero_colonna" ' #   { totale +=
$numero_colonna #   } #   END { #       print totale #   }' "$nomefile" #   ----
-----------------------------------   exit 0
```

Per quegli script che necessitano di un unico strumento tuttofare, un coltellino svizzero informatico, esiste Perl. Perl combina le capacità di **sed** e **awk**, e, per di più, un'ampia parte di quelle del **C**. È modulare e supporta qualsiasi cosa, dalla programmazione orientata agli oggetti fino alla preparazione del caffè. Brevi script in Perl si prestano bene ad essere inseriti in script di shell e si può anche dichiarare, con qualche ragione, che Perl possa sostituire completamente lo scripting di shell stesso (sebbene