Redirezione, filtri, espressioni regolari

Andrea Passerini
passerini@disi.unitn.it

Informatica

Dispositivi standard di I/O

- I programmi scritti per terminali a carattere (compresi i comandi per shell) usano 3 dispositivi standard di I/O:

  stdin (input)
  stdout (output)
  stderr (error)

- Normalmente `stdin` è collegato alla tastiera, mentre `stdout` e `stderr` sono collegati al terminale video a caratteri
- I dispositivi possono essere "rediretti" su file o in ingresso ad altri comandi tramite gli operatori di redirezione $>$, $>>$, $<$, $|$.

Esempi di redirezione

Redirezionare l'output su file

- Redirezionare l'output su file permette di memorizzare in maniera rapida l'output dei comandi, invece di vederlo e poi perderlo come accade quando va su stdout
- L'operatore $>$ redireziona l'output su un file, fornito come argomento dopo l'operatore
- Il file viene creato se non esiste, o sovrascritto se esistente

```
[andrea@praha Data]$ ls
Fasta  Labels  README  Results  tmp2
[andrea@praha Data]$ ls > ls.out
[andrea@praha Data]$ ls
Fasta  Labels  ls.out  README  Results  tmp2
[andrea@praha Data]$ cat ls.out
Fasta
Labels
ls.out
README
Results
tmp2
```

Esempi di redirezione

Appendere l'output ad un file

- L'operatore >> redireziona l'output su un file, fornito come argomento dopo l'operatore
- Il file viene creato se non esiste, ma se esiste, l'output viene aggiunto in fondo al file, senza sovrascriverne il contenuto

```
[andrea@praha Data]$ ls tmp2/
tmp  tmp3
[andrea@praha Data]$ ls tmp2/ >> ls.out
[andrea@praha Data]$ cat ls.out
Fasta
Labels
ls.out
README
Results
tmp2
tmp
tmp3
```

Redirezione

Filtri

- I sistemi basati su UNIX fanno ampio uso di questi operatori di redirezione
- Esiste una vasta gamma di comandi che opera prendendo del testo dallo standard input, processandolo in qualche modo, e mandando il risultato su standard output
- Tali comandi prendono il nome di *filtri* poiché operano come dei filtri su un flusso di dati
- L'operatore $<$ seguito dal nome di un file, redireziona il contenuto del file sullo standard input di un comando
- Applicata ad un filtro, la redirezione dell'input permette di filtrare il contenuto del file

Ordinamento con `sort`

- Il filtro `sort` restituisce in output una versione dell'input con le righe riordinate lessicograficamente

```
[andrea@praha ~]$ cat disordine
d
a
r
c
f
r
3
[andrea@praha ~]$ sort < disordine
3
a
c
d
f
r
r
```

Pipelines

l'operatore |

- L'operatore | (*pipe*) posto tra due comandi, prende l'output del primo e lo redireziona come input al secondo
- Tale operatore è fondamentale in quanto permette di creare *pipelines* di comandi, in cui vari filtri sono applicati in cascata ciascuno sull'output del filtro precedente

```
[andrea@praha ~]$ cat disordine | sort
3
a
c
d
f
r
r
```

Ordinamento numerico con `sort`

- L'opzione `-g` del filtro `sort` permette di ordinare numericamente invece che lessicograficamente

```
[andrea@praha ~]$ cat bindings.txt | sort -g
idGene  geneSymbol      idUTR   name
4       A2M     126944  uc011mxd.1_3UTR
4       A2M     126944  uc011mxd.1_3UTR
4       A2M     77697   uc001qvj.1_3UTR
4       A2M     77698   uc001qvk.1_3UTR
4       A2M     77699   uc009zgk.1_3UTR
4       A2M     77699   uc009zgk.1_3UTR
12      AACS    80379   uc001uhd.2_3UTR
12      AACS    80380   uc001uhc.2_3UTR
12      AACS    80381   uc009zyi.2_3UTR
18      AADAT   111284  uc003isr.2_3UTR
18      AADAT   111284  uc003isr.2_3UTR
...
```

Eliminazione di replicati con `uniq`

- Il filtro `uniq` permette di eliminare le righe replicate di un file, stampando un solo esemplare per ogni riga.
- Poiché le righe replicate devono essere contigue, il filtro è spesso usato in cascata ad un filtro `sort`

```
[andrea@praha Data]$ cat genes.txt | sort | uniq
AAA1
AAGAB
AAK1
AARSD1
AASDH
AASDHPPT
AASS
AB073649
...
```

Eliminazione di replicati con `uniq`

- L'opzione `-c` di `uniq` permette di anteporre ad ogni riga il numero di duplicati trovati

```
[andrea@praha Data]$ cat genes.txt | sort | uniq -c
   1 AAA1
   1 AAGAB
   1 AAK1
   1 AARSD1
   1 AASDH
   6 AASDHPPT
   1 AASS
   1 AB073649
   1 AB074162
   2 AB1
   3 AB209061
   ...
```

Eliminazione di replicati con `uniq`

- L'opzione `-d` di `uniq` permette di stampare solo le righe duplicate

```
[andrea@praha Data]$ cat genes.txt | sort | uniq -d
AASDHPPT
AB1
AB209061
ABCA12
ABCC1
ABCC3
ABCC4
ABCC5
ABCD4
ABCG1
...
```

Redirezione su file

- E' sempre possibile, dopo aver applicato una serie di filtri in pipeline, redirezionare l'uscita finale su file

```
[andrea@praha Data]$  cat genes.txt | sort | uniq -c
 > uniq.txt
[andrea@praha Data]$ cat uniq.txt
AASDHPPT
AB1
AB209061
ABCA12
ABCC1
ABCC3
ABCC4
ABCC5
ABCD4
ABCG1
...
```

Lettura in reverse con `tac`

- `tac` esegue la concatenazione di file come `cat`, ma stampa ciascuno di essi dall'ultima alla prima riga
- Entrambi possono prendere dati da input con pipe invece che da file

```
[andrea@praha Data]$ ls | cat
Fasta
Labels
README
Results
tmp2
[andrea@praha Data]$ ls | tac
tmp2
Results
README
Labels
Fasta
```

Prime righe con `head`

- `head` legge le prime *n* righe dell'input, dove *n* viene specificato con l'opzione $-<n>$ (o 10 di default)

```
[andrea@praha Data]$ head -5 interactions.txt
elementName name evidenceType
ACO1 hg19_uc001dte.3_3UTR RIP-chip
ACO1 hg19_uc001dtf.2_3UTR RIP-chip
ACO1 hg19_uc001dtg.3_3UTR RIP-chip
ACO1 hg19_uc001nsu.2_3UTR RIP-chip
```

Ultime righe con `tail`

- `tail` legge le ultime *n* righe dell'input, dove *n* viene specificato con l'opzione $-<n>$ (o 10 di default)

```
[andrea@praha Data]$ tail -5 interactions.txt
ZFP36L2 hg19_uc003hhh.1_3UTR
ZFP36L2 hg19_uc003kwe.1_3UTR
ZFP36L2 hg19_uc003kwf.2_3UTR
ZFP36L2 hg19_uc003nui.2_3UTR
ZFP36L2 hg19_uc003nuj.2_3UTR
```

Righe “a partire da” con `tail`

- con la sintassi `-n +<x>`, `tail` stampa tutte le righe a partire dalla numero `<x>`

```
[andrea@praha Data]$ tail -n +2 interactions.txt
ACO1    hg19_uc001dte.3_3UTR    RIP-chip
ACO1    hg19_uc001dtf.2_3UTR    RIP-chip
ACO1    hg19_uc001dtg.3_3UTR    RIP-chip
ACO1    hg19_uc001nsu.2_3UTR    RIP-chip
ACO1    hg19_uc001rxc.3_3UTR    RIP-chip
ACO1    hg19_uc001rxd.3_3UTR    RIP-chip
ACO1    hg19_uc001rxe.3_3UTR    RIP-chip
...
```

Scelta di colonne con `cut`

- `cut` permette di stampare solo una selezione delle colonne in ingresso
- l'opzione `-f <list>` permette di specificare una lista di colonne da stampare, come:
  - lista di numeri di colonna separati da virgola (e.g. `cut -f 1,4,5,7`)
  - range di numeri di colonna, indicando i limiti (e.g. `cut -f 1-4 cut -f 5-9`)
  - range di numeri di colonna, indicando un solo limite (e.g. `cut -f 5-, cut -f -7`)
  - una qualunque combinazione delle modalità precedenti (e.g. `cut -f -5,7-9,10,11,15-`)

Scelta di colonne con `cut`: esempio

```
[andrea@praha Data]$ cat interactions.txt | cut -f 1
elementName
ACO1
ACO1
ACO1
ACO1
ACO1
ACO1
ACO1
ACO1
ACO1
ACO1
ACO1
...
```

Filtri

Esercizio

```
[andrea@praha Data]$ cat interactions.txt
elementName     name    evidenceType
ACO1    hg19_uc001dte.3_3UTR    RIP-chip
ACO1    hg19_uc001dtf.2_3UTR    RIP-chip
ACO1    hg19_uc001dtg.3_3UTR    RIP-chip
ACO1    hg19_uc001nsu.2_3UTR    RIP-chip
ACO1    hg19_uc001rxc.3_3UTR    RIP-chip
ACO1    hg19_uc001rxd.3_3UTR    RIP-chip
ACO1    hg19_uc001rxe.3_3UTR    RIP-chip
ACO1    hg19_uc001rxg.1_3UTR    RIP-chip
ACO1    hg19_uc001rxh.1_3UTR    RIP-chip
ACO1    hg19_uc001rxi.2_3UTR    RIP-chip
ACO1    hg19_uc001rxj.1_3UTR    RIP-chip
ACO1    hg19_uc001rxk.1_3UTR    RIP-chip
...
```

Calcolare il numero di interazioni di ciascun UTR, ed ordinarli in base a tale numero

Soluzione

```
[andrea@praha Data]$ cat interactions.txt | tail -n +2 |
 cut -f 2 | sort | uniq -c | sort -g
...
  65 hg19_uc002hdy.3_3UTR
  65 hg19_uc004ebm.1_5UTR
  68 hg19_uc001euz.2_3UTR
  72 hg19_uc003owk.2_3UTR
  72 hg19_uc011khw.1_3UTR
  79 hg19_uc002liu.1_3UTR
  86 hg19_uc001opa.2_3UTR
  87 hg19_uc010lez.2_3UTR
```

Scelta di colonne con `cut`

- di default, `cut` usa il TAB come delimitatore tra colonne
- E' possibile specificare un delimitatore diverso tramite l'opzione `-d`
- Tipici delimitatori sono lo spazio (e.g. `cut -d ' ' -f 1-5`), la virgola (e.g. `cut -d ',' -f 7-`).

```
[andrea@praha Data]$ ls -l
total 28
drwxr-xr-x 2 andrea andrea 4096 2008-12-22 13:06 Fasta
-r--r--r-- 1 andrea andrea   50 2009-01-14 17:55 README
drwxr-xr-x 4 andrea andrea 4096 2008-12-22 13:57 Results
dr-xr-xr-x 4 andrea andrea 4096 2009-01-14 17:42 tmp2
-rw-r--r-- 1 andrea andrea  797 2009-01-14 19:01 uniq.txt
[andrea@praha Data]$ ls -l | cut -d ' ' -f 3

andrea
andrea
andrea
andrea
andrea
```

Attenzione

- `cut` considera come colonna ciò che si trova tra due delimitatori successivi
- se compaiono più delimitatori in sequenza (e.g. una sequenza di spazi se si usa `-d ' '`), `cut` stamperà delle colonne vuote

```
[andrea@praha Data]$ ls -l | cut -d ' ' -f 5,8

4096 Fasta
 2009-01-14
4096 Results
4096 tmp2
 19:01
```

Modifica di caratteri con `tr`

- `tr` (`TRanslate`) permette di modificare tutte le occorrenze di un carattere
- L'uso più semplice consiste nel convertire un carattere in un altro

```
[andrea@praha Data]$ cat seq.txt | tr T U
GAUGAUGGGACUCA
GUAUUUUUUAUUUCAGUCACUCCUGAUA
AUUUGGAGAAAUAAA
UCUUCCGUGU
```

Modifica di caratteri con `tr`

- E' possibile specificare come carattere anche caratteri speciali. I più utili sono il ritorno a capo (`\n`) e il TAB (`\t`)
- I caratteri speciali devono essere racchiusi tra apici

```
[andrea@praha Examples]$ cat fasta/*
>1a02f
RRIRRERNKMAAAKSRNRRRELTDTLQAETDQLEDEKSALQTEIANLLKEKEK>1a2xb
EEKRNRAITARRQHLKSVMLQIAATELEKEE>1abaa
MFKVYGYDSNIHKCGPCDNAKRLLTVKKQPFEFINIMPEKGVFDDEKIAELLT>1aqza
ATWTCINQQLEDKRLLYSQAKAESNSHHAPLSDGKTGSSYPHWFTNGYDGNGK>1ay7b
...
[andrea@praha Examples]$ cat fasta/* | tr '>' '\n'

1a02f
RRIRRERNKMAAAKSRNRRRELTDTLQAETDQLEDEKSALQTEIANLLKEKEK
1a2xb
EEKRNRAITARRQHLKSVMLQIAATELEKEE
1abaa
MFKVYGYDSNIHKCGPCDNAKRLLTVKKQPFEFINIMPEKGVFDDEKIAELLT
1aqza
ATWTCINQQLEDKRLLYSQAKAESNSHHAPLSDGKTGSSYPHWFTNGYDGNGK
...
```

Modifica di caratteri con `tr`

- E' possibile specificare set di caratteri invece che caratteri singoli.
- Il modo più semplice è specificando il range, come abbiamo visto per le wildcards (e.g. 0-9, a-z)
- Esempio: convertire minuscole in maiuscole

```
[andrea@praha Examples]$ ls -l | tr a-z A-Z
TOTAL 32
-RW-R--R-- 1 ANDREA ANDREA  5231 2009-01-12 15:35 DATA.DST
-RW-R--R-- 1 ANDREA ANDREA 10549 2009-01-12 15:09 DATA.GENBANK
-RW-R--R-- 1 ANDREA ANDREA    10 2009-01-19 11:38 P
-RW-R--R-- 1 ANDREA ANDREA   865 2009-01-12 14:35 SP.FASTA
```

- Esempio: convertire numeri in lettere

```
[andrea@praha Examples]$ ls -l | tr 0-9 a-z
total dc
-rw-r--r-- b andrea andrea  fcdb caaj-ab-bc bf:df data.dst
-rw-r--r-- b andrea andrea bafej caaj-ab-bc bf:aj data.genbank
-rw-r--r-- b andrea andrea    ba caaj-ab-bj bb:di p
-rw-r--r-- b andrea andrea   igf caaj-ab-bc be:df sp.fasta
```

Cancellazione di caratteri con `tr`

- L'opzione `-d` permette di specificare un set di caratteri da cancellare, invece che sostituirlo con altro

```
[andrea@praha Examples]$ cat fasta/seq.f
>1icfi
LTKCQEEVSHIPAVHPGSFRPKCDENGNYLPLQCYGSIGYCWCVFPNGTEVPNTRSR
[andrea@praha Examples]$ cat fasta/seq.f  | head -1
 | tr -d '>'
1icfi
```

Eliminazione di duplicati con `tr`

- L'opzione `-s` (*squeeze*) permette di sostituire sequenze di un certo carattere con una singola occorrenza

```
[andrea@praha Examples]$ ls -l | tr -s ' '
total 28
drwxr-xr-x 2 andrea andrea 4096 2008-12-22 13:06 Fasta
-r--r--r-- 1 andrea andrea 50 2009-01-14 17:55 README
drwxr-xr-x 4 andrea andrea 4096 2008-12-22 13:57 Results
dr-xr-xr-x 4 andrea andrea 4096 2009-01-14 17:42 tmp2
-rw-r--r-- 1 andrea andrea 797 2009-01-14 19:01 uniq.txt
```

Esempio

- Esempio: recupero di dimensione e nome dei file, ordinati per dimensione:

```
[andrea@praha Data]$ ls -l
total 28
drwxr-xr-x 2 andrea andrea 4096 2008-12-22 13:06 Fasta
-r--r--r-- 1 andrea andrea   50 2009-01-14 17:55 README
drwxr-xr-x 4 andrea andrea 4096 2008-12-22 13:57 Results
dr-xr-xr-x 4 andrea andrea 4096 2009-01-14 17:42 tmp2
-rw-r--r-- 1 andrea andrea  797 2009-01-14 19:01 uniq.txt
[andrea@praha Data]$ ls -l | tail -n +2  | tr -s ' '
 | cut -d ' ' -f 5,8 | sort -g
50 README
797 uniq.txt
4096 Fasta
4096 Results
4096 tmp2
```

Estrazione di statistiche con `wc`

- `wc` (`WordCount`) permette di estrarre statistiche sul numero di righe, parole e caratteri in ingresso
- Ad esempio per calcolare il numero di righe, parole e caratteri di un file

```
[andrea@praha Data]$ cat data.txt  | wc
     55     225     797
```

- Per stampare il numero di parole distinte presenti in un file

```
[andrea@praha Data]$ cat data.txt  | tr -s ' '
  | tr ' ' '\n' | sort | uniq | wc
     24      23      62
```

(NB: ci sono 24 righe e 23 parole, poiché una delle righe è vuota, quindi non contiene alcuna parola)

Selezione di righe tramite `grep`

- `grep` è un potente filtro che permette di selezionare righe che soddisfano determinate espressioni
- Nell'uso più semplice, `grep` seleziona le righe che contengono una determinata sequenza di caratteri

```
[andrea@praha Data]$ less ligands | grep HEM
101m_ H HEM H HEM
1a7vA C HEM C HEM H HEM
1cl6A H HEM H HEM C HEM
1cxyA C CYS H HEM C CYS H HEM
1dt6A E SM E SM H HEM C HEM
1e7pC H HEM H HEM H HEM H HEM
1izoA H HEM H HEM C HEM
1mj4A H HEM H HEM H HEM
1oj6A H HEM H HEM
1q16C H HEM H HEM H HEM H HEM
1vf5A C HEM H HEM H HEM H HEM H HEM
```

Uso di *espressioni regolari* tramite `grep`

- In generale, `grep` permette di specificare delle *espressioni regolari* che devono essere soddisfatte dalle righe da selezionare (con l'opzione `-E`)
- Le wildcards viste precedentemente sono semplici esempi di espressioni regolari. `grep` supporta espressioni regolari più complesse (ma alcune caratteristiche sono comuni)
- Le espressioni regolari sono uno strumento estremamente utile per eseguire operazioni su sequenze, e sono usate diffusamente in bioinformatica
- Come le espressioni aritmetiche, le espressioni regolari sono costruite combinando espressioni più semplici tramite degli *operatori*

Espressioni regolari

Espressioni regolari di base

- La più semplice espressione regolare è un singolo carattere, che rappresenta sé stesso

```
[andrea@praha Data]$ cat fasta/* | grep W

PPVWYPGGQC
IVNGEEAVPGSWPWQVS
MSYEKEFLKDFEDWVKTQIQV
SNATTDKAQBQTSINLALSTINGKW
```

- Il punto `.` è un'espressione che rappresenta un qualsiasi carattere (ed ha senso combinato in espressioni complesse)
- Una *classe di carettere* più essere specificata racchiudendone la rappresentazione tra parentesi quadre
- Come per le wildcards, è possibile rappresentare esplicitamente insiemi di caratteri, o specificare dei range

Espressioni regolari

Principali classi di caratteri

`[CDE]`	uno qualsiasi dei caratteri specificati
`[A-Z]`	un carattere nel range dei caratteri specificati
`[^0-9]`	un carattere che non sia nel range dei caratteri specificati (NOTA: nelle wildcards si usa `!` invece di `^`)
`[[:space:]]`	qualsiasi carattere di spaziatura (e.g. `\t\n` o spazio)

Espressioni regolari

Ancore

- Il carattere speciale `^` indica l'inizio della riga

```
[andrea@praha Data]$ cat fasta/* | grep ^C
```

```
CRYLLVRSLQTFSQAWFTCRRCYRGN
CKYKFENWGACDGGTGTKVRQGTLKKA
CRKEQGKFYDHLLRDCISCASICGQHP
CYCRIPACIAGERRYTCIYQGRLWAFCC
```

- Il carattere speciale `$` indica la fine della riga

```
[andrea@praha Data]$ cat fasta/* | grep C$
```

```
SEAVKFLTNETREREVFDRLGMIYTVGYSVC
GHACYRNCWREGNDEETCKERC
CKYKFENWGACDGGTGTKVRQGTLKKARYNAQCQETIRVTKPC
MDLAPQMLRELQETNAALQDVRELLRQQVKEITFLKNTVMECDAC
```

Espressioni regolari

Ripetizione

- Una certa espressione regolare può essere seguita da un operatore di *ripetizione*

?	l'espressione è opzionale, e può essere soddisfatta al massimo una volta
*	l'espressione può essere soddisfatta da zero ad un qualsiasi numero di volte
+	l'espressione deve essere soddisfatta almeno una volta, o un qualsiasi numero maggiore di volte
{n}	espressione soddisfatta esattamente *n* volte
{n,}	espressione soddisfatta *n* volte o più
{,m}	espressione soddisfatta al massimo *m* volte
{n,m}	espressione soddisfatta tra *n* ed *m* volte

Espressioni regolari

Ripetizione: esempi

```
[andrea@praha Data]$ cat fasta/* | grep -E [A-D]{2}

  RAEVQIARKLQCIADQFHRLHT
  NTVGYLEQKMFAAMVADNQMAMVMLNPK
  GEGKVVAAAYPDLYADBDAIBIIVKLAN
```

Espressioni regolari

Ripetizione: problema dell'espansione di shell

- Abbiamo visto che l'interprete di comandi *espande* le wildcards nella lista dei nomi di file che le soddisfano
- Questo avviene *prima* che il comando venga eseguito
- Nel caso di un comando che prende un'espressione regolare, come `grep`, se l'espressione regolare contiene dei caratteri da espandere, l'interprete li sostituirà con i nomi compatibili, e *dopo* eseguirà il comando.

Espressioni regolari

Ripetizione: protezione dall'espansione di shell

- Di norma è sempre utile proteggere l'espressione regolare, per evitare risultati inattesi dovuti ad un'espansione non prevista

```
[andrea@praha Data]$ cat Fasta/* | grep -E A{2,3}
grep: A3: No such file or directory
```

- La shell espande l'espressione in:

```
[andrea@praha Data]$ cat Fasta/* | grep -E A2 A3
grep: A3: No such file or directory
```

- L'espressione va quindi protetta

```
[andrea@praha Data]$ cat Fasta/* | grep -E "A{2,3}"
```

```
MADAAVHGHG
AAILGDEYLW
AAVILESIFL
GSAAEVMKKY
AIGPAASLVV
```

Espressioni regolari

Espressioni regolari vs Wildcards

- Le wildcards permettono di costruire semplici espressioni regolari per l'interprete di comandi
- Le espressioni regolari usate da `grep` sono più complesse
- Nonostante vari aspetti in comune, la sintassi dei due tipi di espressioni differisce

Espressioni regolari vs Wildcards

Esempio

- Elenca tutti i file che cominciano per `R`

```
[andrea@praha Data]$ ls -l R*
-r--r--r-- 1 andrea andrea   50 2009-01-14 17:55 README
drwxr-xr-x 4 andrea andrea 4096 2008-12-22 13:57 Results
```

- Seleziona le righe che contengono `R` zero o più volte

```
[andrea@praha Data]$ ls -l | grep 'R*'
total 28
drwxr-xr-x 2 andrea andrea 4096 2008-12-22 13:06 Fasta
drwxr-xr-x 2 andrea andrea 4096 2009-01-14 17:37 Labels
-rw-r--r-- 1 andrea andrea   49 2009-01-14 18:30 ls.out
-r--r--r-- 1 andrea andrea   50 2009-01-14 17:55 README
drwxr-xr-x 4 andrea andrea 4096 2008-12-22 13:57 Results
drwxr-xr-x 4 andrea andrea 4096 2009-01-19 14:37 tmp2
-rw-r--r-- 1 andrea andrea  797 2009-01-14 19:01 uniq.txt
```

Espressioni regolari vs Wildcards

Esempio

- Elenca tutti i file che cominciano per `12` oppure `13`

```
[andrea@praha TmpDownloads]$ ls 1{2,3}*
12.1.1.135.3092.pdf  132843970425343 8_article.pdf
```

- Seleziona le righe che contengono `1` ripetuto due o tre volte

```
[andrea@praha TmpDownloads]$ ls | grep -E "1{2,3}"
10.1.1.74.5119.pdf
112843970425343 8_article.pdf
```

Espressioni regolari

Nota

- Gli operatori di ripetizione permettono di espandere la *sottostringa* che deve soddisfare l'espressione
- Il numero massimo di ripetizioni vincola la sottostringa che soddisferà l'espressione, NON la riga intera.
- Se una riga contiene una sottostringa che soddisfa l'espressione, `grep` la restituirà in uscita, anche se tale sottostringa è seguita da un'ulteriore ripetizione oltre al numero consentito dall'espressione
- Vedremo che questo farà differenza quando si *concatenano* espressioni

Espressioni regolari

Esempio

```
[andrea@praha Data]$ cat Fasta/* | grep -E "A?" \
> | grep -E "A{2}"

  MADAAVHGHG
  AAILGDEYLW
  AAVILESIFL
  GSAAEVMKKY
  AIGPAASLVV
  GTNAAMRKAF
```

Espressioni regolari

Concatenazione

- Le espressioni regolari possono essere *concatenate* per creare espressioni più complesse.
- L'espressione risultante sarà soddisfatta se si riesce a trovare una sequenza di stringhe contigue ciascuna delle quali soddisfa la corrispondente espressione
- Ad esempio, una parola è una semplice concatenazione di espressioni che consistono in singoli caratteri
- Quando una espressione contiene operatori di ripetizione, le sottostringhe compatibili con tali operatori vengono provate l'una dopo l'altra, verificando se ciò che rimane dopo la sottostringa è compatibile con le espressioni successive

Concatenazione: esempi

```
[andrea@praha Data]$ cat Fasta/* | grep -E "A?Q"
  MVLSEGEWQL
  MQKGNFRNQR
  AQCEATIESN
  GERRRSQLDR
  MALTNAQILA
  QTDVIAQRKA
  MHPRFQTAFA
  QNEGHECQCQ

[andrea@praha Data]$ cat Fasta/* | grep -E "TS+K"
  MAKKTSSKGK

[andrea@praha Data]$ cat Fasta/* | grep -E "S.*E{2,3}V"
  MTGMSREEVE
  MSSSEEVSWI
```

Espressioni regolari

Alternative

- E' possibile specificare più espressioni regolari *alternative*, separandole con |
- La riga sarà soddisfatta se una qualsiasi delle espressioni alternative è soddisfatta
- Alternative e concatenazione di espressioni possono essere combinate a piacere

Espressioni regolari

Alternative: esempi

```
[andrea@praha Data]$ cat Fasta/* \
> | grep -E "G.*D{2,}|^E.*F"
  ETFEIPESVT
  EEEFQFLRCQ
  GPQTDDPRNK

[andrea@praha Data]$ cat Fasta/* | \
> grep -E "EI+.*E$|H[CDE]{2}"
  QNEGHECQCQ
  MELTKEIISE
  LDAFQEIPLE
  GSHMRVYHEC
```

Espressioni regolari

Precedenza

- Quando si combinano più espressioni regolari, il modo con cui si interpreta l'espressione risultante dipende dalle *regole di precedenza*
- Nelle operazioni aritmetiche, la precedenza tra operatori ci dice ad esempio che

$$a + b \times c = a + (b \times c)$$

- Nelle espressioni regolari, l'ordine di precedenza è :
  1. ripetizione
  2. concatenazione
  3. alternativa

Espressioni regolari

Precedenza: esempi

```
"A?Q" = "(A?)Q"
"TS+K" = "T(S+)K"
"S.*E{2,3}V" = "S(.*)(E{2,3})V"
"G.*D{2,}|^E.*F" =  "(G(.*)(D{2,}))|(^E(.*)F)"
"EI+.*E$|H[CDE]{2}" = "(E(I+)(.*)E$)|(H([CDE]{2}))"
```

Espressioni regolari

Precedenza

- Come per le espressioni algebriche, è possibile stabilire una precedenza diversa racchiudendo le espressioni tra parentesi tonde
- Tale possibilità è utile per forzare precedenze diverse da quelle di default, tipo:
  - applicare un operatore di ripetizione ad una espressione complessa
  - far precedere o seguire un'alternativa da altre espressioni

Espressioni regolari

Precedenze: esempi

```
[andrea@praha Data]$ cat Fasta/* | grep -E "E(I+.)*E$"
  MEFDYVICEE
  MELTKEIISE
  MLMREVTKEE
  MDLSALRVEE

[andrea@praha Data]$ cat Fasta/* \
> | grep -E "D(K+E|[CDE]{2}).*T"
  MDKDCEMKRT
  PKYTIVDKET
```

Righe che NON soddisfano l'espressione

- L'opzione `-v` permette di selezionare le righe che NON soddisfano l'espressione indicata

```
[andrea@praha Data]$ cat alignments
# STOCKHOLM 1.0
#=GF ID   RRM_1
#=GF AC   PF00076.17
#=GF DE   RNA recognition motif.
#=GF PI   rrm;
...
[andrea@praha Data]$ cat alignments | grep ^# -v
C1GKH7_PARBD/164-234 ......................IFIQNL....
C1H4D7_PARBA/94-166  ......................IFIQNL....
...
```