Introduction à Linux RAQ - Cours 03

Eric Normandeau - 2015-02-23

Plan de cours

1 - Introduction

1.1 - Retour sur les exercices

Réponse aux questions relatives aux exercices de la semaine passée.

1.2 - Retour sur question avec séquences fasta

Nous souhaitons trouver toutes les séquences qui débutent par **ATG** dans le fichier **sequences_wrapped.fasta**. Nous utilisons la commande **grep**.

```
# Séquences avec ATG au début de la ligne
grep -E "^ATG" sequences_wrapped.fasta
```

Malheureusement, les séquences sont repliées sur plusieurs lignes. Nous allons devoir les formater pour que chaque séquence soit sur une seule ligne. Pour cela, nous utilisons un script Python que j'ai écrit pour créer un fichier où les séquences ne prennent qu'une seule ligne.

```
# Déplier les séquences (une ligne par séquence)
fasta_unwrap.py sequences_wrapped.fasta sequences_unwrapped.fasta
```

Nous sommes maintenant prêts à trouver les séquences qui débutent par **ATG**.

```
# Séquences qui débutent par ATG
grep -E "^ATG" sequences_unwrapped.fasta
```

Finalement, nous allons retrouver le nom de ces séquences.

```
# Trouver le nom des séquences
grep -B 1 -E "^ATG" sequences_unwrapped.fasta

grep -B 1 -E "^ATG" sequences_unwrapped.fasta | grep ">"

grep -B 1 -E "^ATG" sequences_unwrapped.fasta | grep ">" | cut -c 2-
```

1.2 - Importer le matériel du cours 03

Nous allons copier un dossier déjà préparé pour le cours 03 avec la commande **cp**, que nous allons revoir plus tard :

```
cd  # Pour retourner dans notre dossier d'utilisateur
cp -r /cours_intro_linux/cours_03 .
```

Toutes les commandes du cours utilisant des fichiers sont lancées à partir du dossier **/home/username/cours_03/01_fichiers**.

2 - sed, awk et perl

Comme nous avons déjà vu, le terminal nous permet de manipuler du texte en utilisant des commandes de base et en créant des pipelines avec le symbole `|`. Afin de pouvoir manipuler le texte de façon encore plus puissante, nous pouvons utiliser les programmes **`sed`**, **`awk`** et **`perl`**. Ils permettent de rechercher et remplacer du texte et d'extraire des portions (lignes ou colonnes) intéressantes. Certaines de leurs utilisations sont très complexes mais même en utilisant seulement les bases, on peut faire beaucoup de choses. Nous allons voir **`sed`** mais dans la plupart des cas nous allons plutôt utiliser **`perl`** puisqu'il peut faire la même chose, ce qui nous rend la tâche plus facile car on n'a qu'une seule syntaxe à apprendre.

2.1 - sed

La commande **`sed`** (pour ***`stream editor`***) permet d'éditer du texte provenant de fichiers ou de l'entrée standard (**`standard input`**). On peut écrire des scripts dans le langage **`sed`** mais nous allons nous contenter de l'utiliser dans le terminal pour faire des commandes d'une ligne, aussi appelés ***`oneliners`***. Sa syntaxe est la suivante ;

```
sed [options] commands [file-to-edit]
```

Typiquement, nous allons utiliser **`sed`** pour :

- Rechercher et remplacer du texte
- Extraire des portions de fichiers

Rechercher et remplacer

```
sed 's/Dodo/Extinct Bird/g' alice.txt

# Pour voir le changement
sed 's/Dodo/Extinct Bird/g' alice.txt | grep "Extinct Bird"
```

Voici comment cette commande fonctionne. Premièrement, on donne le nom de la commande **`sed`**, puis les options (il n'y en a pas dans cette commande), puis le code à exécuter par **`sed`** entre guillemets simples **`'`**. Dans ce code, la lettre **`s`** veut dire que nous allons chercher (*search*) du texte pour le remplacer. Le patron de base est le suivant : **`s/Search/Replace/`**. Entre les deux premiers symboles ***`slash`*** (**`/`**), nous mettons le texte à chercher et entre les deux derniers *slash* le texte pour remplacer. Finalement, la lettre **`g`** à la fin du code veut dire ***`greedy`*** (ou avare), donc **`sed`** va tenter de remplacer le texte le plus grand nombre de fois possible par ligne.

Nous pouvons obtenir le même effet sans utiliser le pipeline ni la commande **`grep`** si nous utilisons l'option **`-n`** (***`no print`***) et en ajoutant la lettre **`p`** (***`print`***) à la fin du code. Ainsi, seules les lignes qui on eu du remplacement seront affichées.

```
sed -n 's/Dodo/Extinct Bird/gp' alice.txt
```

Ici, le texte à rechercher et remplacer est simple, mais nous pouvons utiliser **sed** en combinaison avec les expressions régulières (**regex**) que nous allons voir plus loin en ajoutant l'option **-r** juste après **sed**.

Spécifier des lignes

Avec **sed**, on peut spécifier des lignes à imprimer ou à utiliser pour rechercher et remplacer du texte. Par exemple, cherchons **Alice** pour remplacer son nom par **the girl** mais seulement dans le premier paragraphe.

```
# Trouver le premier paragraphe
cat -n alice.txt | head -20

# Récupérer seulement le premier paragraphe
sed -n '7,11p' alice.txt

# Rechercher et replacer 'Alice' -> 'the girl'
sed -n '7,11p' alice.txt | sed 's/Alice/the girl/g'
```

Nous allons terminer notre court survol de **sed** ici. Le langage **perl** permet de faire presque tout ce que **sed** peut faire et est plus approprié pour utiliser les expressions régulières complexes. Nous allons donc préférer utiliser **perl** mais beaucoup de gens utilisent **sed**.

2.2 - awk

La commande **awk** est en fait un puissant langage de programmation pour manipuler des fichiers et des calculs arithmétiques. Son nom vient des premières lettres des noms des auteurs (Alfred **A**ho, Peter **W**einberger, and Brian **K**ernighan). On peut écrire des scripts dans le langage de **awk**, mais comme pour sed, nous allons nous contenter de l'utiliser dans le terminal pour faire des ***oneliners***. Sa syntaxe est la suivante ;

```
# Syntaxe simple
awk '{action}' filename

# Syntaxe plus complexe
awk 'BEGIN {start_action} {action} END {stop_action}' filename
```

La plupart du temps, nous allons utiliser la syntaxe simple. Typiquement, nous allons utiliser **awk** pour :

- Extraire des portions de fichiers (surtout des colonnes)
- Faire des calculs sur des colonnes (comme des sommes)

Extraire des colonnes

```
# Affichier fichier seeds_tabs.txt
cat seeds_tabs.txt

# Extraire colonnes 1 et 3
awk '{print $1,$3}' seeds_tabs.txt

# Les séparer par une tabulation
awk '{print $1 "\t" $3}' seeds_tabs.txt

# Mettre la colonne 3 en premier
awk '{print $3 "\t" $1}' seeds_tabs.txt

# Spécifier le séparateur de colonnes
awk -F";" '{print $1,$3}' seeds_semicolon.txt
```

Extraire des lignes

```
# Lignes avec 8 colonnes
awk 'NF == 8' seeds_incomplete.txt

# Lignes avec au moins 7 colonnes
awk 'NF >= 7' seeds_incomplete.txt

# Lignes où Compact est au moins 0.9
awk '$3 >= 0.9' seeds_tabs.txt

# ... et Perim est plus grand que 14
awk '$3 >= 0.9 && $2 > 14' seeds_tabs.txt

# Extraire lignes contenant du texte
awk '/CHAPTER/' alice.txt

# Extraire lignes avec entre 10 et 20 caractères
awk 'length <= 20 && length >= 10' alice.txt
```

Calculs

```
# Additionner deux colonnes
awk '{print $1 + $3}' seeds_tabs.txt

# ... on enlève la première ligne
awk 'NR > 1 {print $1 + $3}' seeds_tabs.txt
```

```
# Somme d'une colonne (en enlevant la première ligne)
awk '!/Area/{sum+=$1} END {print sum}' seeds_tabs.txt

# Moyenne d'une colonne
awk '!/Area/{sum+=$1; n+=1} END {print sum/n}' seeds_tabs.txt
```

La commande **awk** cache un puissant langage de manipulation de texte et de calcul. Cependant, ce dont il faut se rappeler c'est que **awk** est surtout utilisé pour extraire des colonnes. Pour tout le reste, une simple recherche sur **Google** permet de trouver rapidement une recette.

Par exemple, pour replier les séquences fasta du fichier **sequences_wrapped.fasta**, on peut utiliser la commande **awk** suivante :

```
awk '/^>/&&NR>1{print "";}{ printf "%s",/^>/ ? $0"\n":$0 }' \
    sequences_wrapped.fasta
```

Il y a un coût évident à avoir une syntaxe si puissante : la lisibilité en souffre. C'est pourquoi je vous recommande d'utiliser **awk** seulement pour des opérations plus simples, comme extraire des colonnes, et d'écrire des scripts ou d'utiliser des programmes qui ont bien été testés pour faire les opérations plus complexes.

2.3 - perl

Le langage de programmation **perl** a été à l'origine inventé pour remplacer des commandes spécialisées comme **sed** et **awk** par un seul programme. Le langage **perl** a été le langage le plus utilisé pour écrire des scripts sur les systèmes UNIX pendant environ 20 ans. Ses capacités sont presque illimitées mais sa syntaxe est parfois très difficile à lire. C'est pourquoi nous allons utiliser **perl** seulement pour lancer des commandes d'une ligne dans le terminal (commandes appelées : ***oneliners***). Nous allons utiliser **perl** surtout pour rechercher et remplacer du texte.

Rechercher et remplacer

```
# Exemple de la syntaxe la plus utilisée
perl -pe 's/Search/Replace/g' file

# Cecilia au Pays des Merveilles (début du texte)
perl -pe 's/Alice/Cecilia/g' alice.txt | head -20
```

La commande est similaire à ce qu'on utilisait avec **sed** plus tôt mais nous ajoutons les options **-p** (pour ***print***) et **-e** (pour ***execute***). En combinant ces deux options, **perl** lit chaque ligne, exécute la commande entre les guillemets (') et imprime le résultat. Nous

pourrions aussi vouloir imprimer seulement les lignes où il y a eu un remplacement. Noter l'option **-n** qui dit de **ne pas** imprimer. Dans la portion exécutée, nous spécifions d'imprimer seulement si il y a eu un remplacement.

```
# Cecilia au Pays des Merveilles (seulement lignes remplacées)
perl -ne 'print if s/Alice/Cecilia/g' alice.txt | head -20
```

Nous passons tout de suite à la section suivante sur les expressions régulières car c'est grâce à elles que **grep**, **sed** et **perl** sont des commandes si puissantes.

3 - Expressions régulières

Les expressions régulières (souvent appelées ***regex***) sont un mini langage spécialisé pour la recherche et le remplacement de texte. Sa syntaxe est très compacte et donc difficile à lire, mais elle permet de faire des recherches complexes avec un minimum de caractères. Il existe plusieurs versions des expressions régulières, mais celles utilisées par **`perl`** sont devenu le standard. C'est une autre raison pour utiliser **`perl`** plutôt que **`sed`**. Certaines commandes, comme **`grep`**, on même une options (**`-P`** pour **`grep`**) pour spécifier qu'on veut utiliser les expressions régulières comme dans **`perl`**. Dans cette section, nous allons explorer la syntaxe particulière des **`regex`** avec des exemples. Nous allons utiliser **`perl`** ou **`grep -P`** pour montrer le résultat.

3.1 - Classes de caractères

Certaines classes de caractères peuvent être spécifiées par un code spécial débutant par **`\`** ou à l'aide d'un groupe de caractères entre crochets **`[ ]`**. Utiliser ces raccourcis permet d'écrire des expressions régulières plus courtes et parfois plus lisibles.

- Mots : **`\w`** ou **`[A-Za-z0-9_]`**. Inverse : **`\W`** ou **`[^A-Za-z0-9_]`**
- Alphabétique : **`[A-Za-z]`** ou **`[[:alpha:]]`**. Inverse : **`[^A-Za-z]`**
- Espace, tabulation et fin de ligne : **`\s`** ou **`[[:space:]]`** . Inverse : **`\S`**
- Nombres : **`\d`** ou **`[0-9]`**. Inverse : **`\D`** ou **`[^0-9]`**

3.2 - Caractères spéciaux

3.2.1 - N'importe quoi (`.`)

Le symbole du point (`.`) correspond à *n'importe quel* caractères.

```
# N'importe quel caractère suivi de 'lice'
grep -P '.lice' alice.txt
grep -Po '.lice' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.2 - Groupe de caractères (`[ ]`. Inverse : `[^ ]`)

Les crochets permettent de chercher n'importe quel des caractères qui s'y trouvent. Si le premier caractère entre les crochets est `^`, alors on cherche n'importe quel des caractères qui **n'est pas présent** entre les crochets.

```
# 'A' ou 'p' suivi de 'l' suivi de 'i' ou 'a' suivi de 'ce'
grep -Po '[Ap]l[ia]ce' alice.txt | sort | uniq -c | sort -nr | head
```

```
# On peut spécifier un range de caractères
grep -Po '[a-zA-Z]l[a-z]ce' alice.txt | sort | uniq -c | sort -nr | head
grep -Po '[[:alpha:]]l[a-z]ce' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.3 - Début de ligne (^)

Le symbole ^, lorsqu'il n'est pas entre crochets, représente le début de la ligne. On peut donc chercher des patrons qui se trouvent au début des lignes.

```
# 'A' en début de ligne, suivi de 4 . et un espace ' '
grep -P '^A.... ' alice.txt
grep -Po '^A.... ' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.4 - Fin de ligne ($)

Le symbole **$** représente la fin de la ligne.

```
# 'ice' en fin de ligne
grep -P '..ice$' alice.txt
grep -Po '..ice$' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.5 - Zéro ou plusieurs fois (*)

Le symbole * se place après un patron pour signifier qu'on doit le trouver zéro ou plusieurs fois.

```
# Trouver tous les mots commençant par 'A' (pas très bon)
grep -Po ' A.*? ' alice.txt | sort | uniq -c | sort -nr | head
```

Dans le contexte ici, le symbole **?** veut dire de prendre la partie la plus courte qui satisfait le patron. Par défaut, la recherche retourne la partie la plus longue qui satisfait le patron de recherche.

3.2.6 - Une ou plusieurs fois (+)

Le symbole + se place après un patron pour signifier qu'on doit le trouver une ou plusieurs fois.

```
# Trouver tous les mots de 2+ lettres commençant par 'A' (pas très bon)
grep -Po ' A.+? ' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.7 - Patron trouvé zéro ou une fois (?)

Le symbole **?** se place après un patron pour signifier qu'on doit le trouver zéro ou une fois.

```
# Mots commencant par 'T' 0-1 fois, puis 'he', puis '.' 0-1 fois
grep -Po ' T?he.? ' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.8 - Patron trouvé n fois ({n})

On utilise **{n}** pour spécifier le nombre de fois qu'un caractère ou patron doit être trouvé.

```
# Espace, 'A' suivi de 4 lettres minuscules '[a-z]' et un espace
grep -Po ' A[a-z]{4} ' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.9 - Patron trouvé de m à n fois ({m, n})

On utilise **{m, n}** pour spécifier le nombre de fois minimal et maximal qu'un caractère ou patron doit être trouvé.

```
# Espace, majuscule suivi de 4 à 8 lettres minuscules '[a-z]' et un espace
grep -Po ' [A-Z][a-z]{4,8} ' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.10 - Groupe de recherche (( )) pour remplacement

On utilise les parenthèses **( )** pour spécifier un groupe de recherche. Ces groupes sont utiles pour rechercher une patron a **ou** un patron b (voir 3.1.11) et pour récupérer une portion du patron dans recherché avec **perl**.

```
# Alice suivi de 2 mots
grep -Po 'Alice (\w+ )(\w+ )' alice.txt | sort | uniq -c | sort -nr | head

# Utiliser les groupes pour remplacer Alice et inverser les 2 mots
perl -ne 'print if s/.+?Alice (\w+) (\w+) .+/The Girl \2 \1/g' alice.txt
```

3.2.11 - Patron a ou patron b (|)

On utilise **|** pour spécifier un patron **ou** un autre.

```
# Alice ou place
grep -Po ' (Ali|pla)ce ' alice.txt | sort | uniq -c | sort -nr | head
```

3.2.12 - Exemples finaux et conclusion

Voici deux exemples plus complexes.

```
# Distribution du nombre de mot par phrase dans alice.txt
perl -pe 's/\n/ /' alice.txt | \
    perl -pe 's/[.!?;][ \n]+/.\n\n/g' | \
    perl -pe 's/ +/ /g' | \
    grep -vE '^$' | \
    awk -F" " 'length <= 200 {print length/10}' | \
    perl -pe 's/\..*//' | \
    sort -n | \
    perl -pe 's/$/0/' | \
    uniq -c

# Fréquence des mots avec un '-' au centre
grep -Po '\W*\w+-\w+\s*' alice.txt | \
    perl -lpe 's/[^\w-]+//g; s/[^\w]+$//; s/^[^\w]+//' | \
    sort | uniq -c | sort -nr | head -20
```

Le but aujourd'hui n'est pas d'avoir tout compris à propos des expressions régulières mais bien d'avoir appris qu'elles existent, comment les reconnaître, et comment les utiliser. Dans le futur, vous pourrez toujours vous tourner vers Google ou une feuille de référence sur les **regex** pour créer les patrons dont vous aurez besoin.

4 - Compression de fichiers

Les programmes **gzip**, **zip** et **tar** sont utilisés pour compresser des fichiers et dossiers. Les commandes **gunzip**, **unzip** et **tar** servent à décompresser leurs archives respectives. La commande **gzip**s'utilise seulement sur des fichiers mais on peut utiliser **tar** pour créer une archive non compressée d'un dossier qui peut par la suite être compressée comme un fichier. La commande **tar** peut faire les deux étapes d'un coup.

4.1 - gzip et gunzip

4.1.1 - Compresser

Le programme **gzip** est le programme par défaut pour compresser des fichiers sur Linux. Pour compresser un fichier avec **gzip**, il suffit de taper la commande suivante en remplaçant **filename** par le nom du fichier à compresser :

```
gzip filename
```

NOTE : Le fichier sera compressé et la copie original effacée. Pour garder la copie original, il faut utiliser l'option **-c** et rediriger la sortie dans un fichier au nom de notre choix. Il est fortement recommandé d'utiliser l'extension **.gz** et de garder le même nom pour les fichiers compressés avec **gzip**. Ainsi, lorsque le fichier sera décompressé, il aura à nouveau le même nom.

```
gzip -c filename > filename.gz
```

4.1.2 - Décompresser

Pour décompresser un fichier compressé avec **gzip**, on utilise **gunzip**.

```
gunzip filename.gz
```

L'archive est alors décompressée et effacée. Pour garder l'archive intacte, on utilise encore une fois l'option **-c**.

```
gunzip -c filename.gz > filename
```

4.2 - zip et unzip

Le programme **zip** est surtout utilisé sur Windows. Il permet de compresser à la fois des fichiers et des dossiers. Il est utile de pouvoir créer et décompresser des archives **zip** si on collabore avec des gens qui utilisent Windows. Je vous conseille cependant d'utiliser **gzip** sur Linux.

4.2.1 - Compresser

```
# Pour un fichier
zip filename.zip filename

# Pour un dossier
zip -r dossier dossier
```

Une copie du fichier original est conservée.

4.2.2 - Décompresser

```
# Fichier
unzip filename.zip

# Dossier
unzip dossier.zip
```

Une copie de l'archive est conservée.

4.3 - tar

Comme mentionné, la commande **gzip** ne peut pas compresser de dossiers. Par contre, on peut utiliser **tar** qui, avec les bonnes options, utilisera à son tour **gzip**.

4.3.1 - Compresser

Pour les dossiers compressés, on peut utiliser une des deux extensions suivantes : **.tar.gz** ou **.tgz**. Je préfère la seconde mais les deux sont acceptables.

```
tar cvfz dossier.tgz dossier
```

Voici ce que les différentes options font :

- **c** : compresser
- **v** : verbose (optionnel : affiche des informations durant la compression)
- **f** : fichier où envoyer l'archive (**dossier.tgz**)
- **z** : utiliser **gzip** pour la compression

4.3.2 - Décompresser

```
tar xvfz dossier.tgz
```

Les options sont les mêmes à l'exception de **x**, qui veut dire d'extraire et qui remplace le **c** pour compresser.

ATTENTION ! : Lorsqu'on décompresse un dossier ou un fichier, **tar** va écraser les fichiers ou dossiers du même nom qui se trouvent dans le dossier où l'archive est décompressée !

5 - Mot de la fin

5.1 - Aujourd'hui, nous avons vu :

- Comment trouver un patron dans des séquences fasta
- Les bases des **`sed`**, **`awk`** et **`perl`**
- Un survol des expressions régulières (**`regex`**)
- Comment compresser et décompresser des fichiers

Nous avons à présent vu la grande majorité des commandes et options qui sont utilisées fréquemment. Nous avons même survolé **`perl`** et les expressions régulières. Avec ces outils, nous pouvons préparer et lancer des analyses simples.

5.2 - Au prochain cours, nous verrons :

- Les éditeurs de texte dans le terminal
- Écrire des scripts bash / perl / python
- Changer les permissions des fichiers
- Créer des alias de commandes
- screen et tmux

5.3 - Questions et suggestions

N'hésitez pas à me poser vos questions durant les cours ou par courriel. Je vais tenter d'y répondre durant les cours. Je vais aussi prendre vos suggestions en note pour tenter d'améliorer le cours.

6 - Exercices

6.1 - sed

Dans ‘**alice.txt** :

- Rechercher **`Queen`** et remplacer par **`Lady`** (commande : **`sed`**)
- ... Imprimer seulement les lignes où il y a eu un remplacement (commande : **`sed`**)
- ... Faire seulement sur les 2000 premières lignes (commande : **`sed`**)

6.2 - awk

Dans **`seeds_tabs.txt`** :

- Extraire les colonnes 3, 4 et 5 (commande : **`awk`**)
- ... Seulement quand la colonne 2 est supérieure à 15 (commande : **`awk`**)
- Additionner les 3 premières colonnes sans imprimer la première ligne (commande : **`awk`**)

6.3 - perl

Dans **`alice.txt`**, comme pour **`sed`** :

- Rechercher **`Queen`** et remplacer par **`Lady`** (commande : **`perl`**)
- ... Imprimer seulement les lignes où il y a eu un remplacement (commande : **`perl`**)

6.4 - Expressions régulières

- Trouver les mots les plus fréquents qui se terminent en **`ing`** dans **`alice.txt`** (commande : **`grep`**, **`sort`**, **`uniq`**, **`head`**)
- Trouver les mots de 8 lettres les plus communs (commande : **`grep`**, **`perl`**, **`awk`**, **`sort`**, **`uniq`**, **`head`**)

6.5 - Compression de fichiers

Avec gzip

- Compresser le fichier **`alice.txt`** (commande : **`gzip`**)
- Valider le changement (commande : **`ls`**)
- ... Décompresser l'archive **`alice.txt`** (commande : **`gunzip`**)

Avec zip

- Compresser le fichier **alice.txt** (commande : **zip**)
- Valider le changement (commande : **ls**)
- ... Décompresser l'archive **alice.txt** (commande : **unzip**)
- ... Effacer l'archive **.zip** (commande : **rm**)
- Compresser le dossier **big_folder** (commande : **zip**)
- Valider le changement (commande : **ls**)
- Renommer **big_folder** en **big_folder_copy** (commande : **mv**)
- ... Décompresser l'archive (commande : **unzip**)
- ... Effacer l'archive **.zip** (commande : **rm**)
- ... Effacer la copie du dossier (commande : **rm**)

Avec tar

- Compresser le dossier **big_folder** (commande : **tar**)
- Valider le changement (commande : **ls**)
- Renommer **big_folder** en **big_folder_copy** (commande : **mv**)
- ... Décompresser l'archive (commande : **tar**)
- ... Effacer l'archive **.zip** (commande : **rm**)
- ... Effacer la copie du dossier (commande : **rm**)

7 - Liste de commandes importantes

Voici une courte liste des commandes que nous avons utilisée aujourd'hui. Entre parenthèses, vous trouverez le nom en anglais de la commande (pour vous aider à retenir la commande). Entre crochets, vous trouverez les options les plus souvent utilisées :

7.1 - sed, awk et perl

- **`sed`** : Rechercher et remplacer lignes **`[-n, -r]`**
- **`awk`** : Extraire et modifier colonnes **`[-e, -F]`**
- **`perl`** : Tout (c'est un langage de programmation complet) **`[-n, -p, -e]`**

7.2 - Expressions régulières

Voir section 3

7.3 - Compression de fichiers

- **`gzip`** : Compresser des fichiers (GNU zip) **`[-c]`**
- **`gunzip`** : Compresser des fichiers (GNU unzip) **`[-c]`**
- **`zip`** : Compresser des fichiers ou dossiers **`[-r]`**
- **`unzip`** : Compresser des fichiers ou dossiers
- **`tar`** : Compresser/décompresser des fichiers et dossiers (tape archive) **`[x, c, v, f, z]`**