TD2 – Langages de script

Abdallah Ammar

19 janvier 2026

Il est fortement recommandé de taper les commandes à la main afin de se familiariser avec le terminal.

Ce TP introduit de nombreuses notions nouvelles. Il est fortement recommandé de prendre des notes au fur et à mesure.

Avant de regarder les solutions proposées dans ce TD, prenez le temps de réfléchir et de tenter de résoudre les questions par vous-même. L'erreur fait partie du processus d'apprentissage.

1 Globbing

Le *globbing* est un mécanisme du shell permettant de faire correspondre des motifs (*patterns*) à des noms de fichiers.

Le globbing est effectué par le shell avant l'exécution de la commande. Les caractères `*`, `?` et `[...]` ont une signification particulière et peuvent être désactivés à l'aide de guillemets ou d'un antislash (\).

— Créez un répertoire nommé `TD2` dans votre répertoire personnel (`$HOME`).

```
$ # d'abord, placez-vous dans votre répertoire personnel
$ cd $HOME
$
$ # ensuite, on crée le répertoire TD2
$ mkdir TD2
$
$ # optionnel :
$ pwd
$ ls
```

— Placez-vous dans le répertoire `TD2` et créez un sous-répertoire nommé `Q1`.

```
$ # pour aller dans TD2
$ cd TD2
$ # ou bien :
$ cd $HOME/TD2
$
$ mkdir Q1
$
$ # optionnel :
```

```
$ pwd
$ ls
```

— Placez-vous dans le répertoire `Q1` et créez les fichiers suivants :

```
$ # pour aller dans Q1
$ cd Q1
$
$ touch a.txt b.txt c.txt aa.txt bb.txt cc.txt
$ touch a.java b.java c.java
$ touch ba.txt ab.txt ca.txt ac.txt
$ touch f1.txt f2.txt f3.txt f12.txt f21.txt f13.txt f31.txt
$ touch note.pdf image.png report.doc index.html
```

— Analysez les résultats des commandes suivantes :

```
$ ls
$ ls *
$ echo *
$ echo * *
$ echo * * *
$ ls *.txt
$ ls *.java
$ ls *.doc
$ ls *.*
$ ls *.TXT
```

Le caractère `*` correspond à n'importe quelle suite de caractères (dans un nom de fichier).

La commande `rm *` supprime tous les fichiers (non cachés) du répertoire courant. Elle n'affiche pas de confirmation par défaut et il n'y a pas de "corbeille". `rm -r *` supprimerait en plus les sous-répertoires de manière récursive. Vérifiez toujours votre motif avant d'utiliser `rm`.

— Analysez les résultats des commandes suivantes :

```
$ ls *.txt
$ ls "*.txt"
$ ls '*.txt'
```

Lorsque le motif est entouré de guillemets, le globbing n'est pas interprété par le shell.

- Les guillemets simples `' '` désactivent l'interprétation de tous les caractères spéciaux.
- Les guillemets doubles `" "` désactivent le globbing, mais permettent toujours l'expansion des variables (avec `$`) et la substitution de commande (avec guillemets inversés `` `...` `` ou `$(...)`).

— Exécutez les commandes suivantes :

```
$ ls *
$ ls .*
```

Par défaut, le globbing ne correspond pas aux fichiers cachés (commençant par un point).

— Analysez les résultats des commandes suivantes :

```
$ ls ?.txt
$ ls a?.txt
$ ls A?.txt
$ ls ?c.txt
$ ls ??.txt
$ ls ???.txt
```

Le caractère `?` correspond à exactement un caractère (ni plus, ni moins).

— Analysez les résultats des commandes suivantes :

```
$ ls [ab].txt
$ ls [ba].txt
$ ls [bc].txt
$ ls [a-z].txt
$ ls [b-e].txt
$ ls a[a-z].txt
$ ls [a-z]a.txt
$ ls f[0-9].txt
$ ls [a-z][0-9].txt
$ ls [a-z][!0-9].txt
$ ls [a-z][a-z].txt
$ ls *.[!t][!x][!t]
```

Les crochets `[ ]` permettent de rechercher un caractère parmi plusieurs possibilités. Par exemple, `[ab]` correspond soit à `a` soit à `b`. On peut aussi utiliser des intervalles, comme `[0-9]` ou `[a-z]`, et la négation, par exemple `[!0-9]` (tout caractère qui n'est pas un chiffre) ou `[^0-9]`.

— Analysez les résultats des commandes suivantes :

```
$ echo {1..5}
$ echo {5..1}
$ echo {15..-10}
$ echo {a..z}
$ echo {h..c}
```

La syntaxe `{début..fin}` permet de générer une séquence de nombres ou de caractères, selon l'ordre croissant ou décroissant.

— Créez 12 répertoires nommés `moi_1`, `moi_2`, ..., `moi_12` en utilisant une seule commande.

```
$ mkdir moi_{1..12}
$ # pour vérifier
$ ls
```

— Sans changer de répertoire, créez 30 fichiers nommés `jour_1.txt`, `jour_2.txt`, ..., `jour_30.txt` dans chacun des répertoires `moi_1` à `moi_12`, toujours avec une seule commande.

```
$ touch moi_{1..12}/jour_{1..30}.txt
$ # pour vérifier
$ ls *
```

2 Expressions régulières (regex) avec `grep`

Dans cette exercice, nous allons utiliser des expressions régulières (*regex*) avec la commande `grep` pour rechercher des motifs dans des fichiers texte.

Globbing vs Regex

— Le globbing (`*`, `?`, `[...]` dans le shell) sert à faire correspondre des motifs à des noms de fichiers.
— Les expressions régulières décrivent des motifs à l'intérieur du contenu des fichiers (les lignes de texte).

— Créez un sous-répertoire `Q3` dans `TD3` et placez-vous dedans.

```
$ cd $HOME/TD3
$ mkdir Q3
$ cd Q3
```

— Telechargez le fichier `donnees.txt`

2.1 Commande `grep`

— Affichez toutes les lignes contenant le mot `user`.

```
$ grep 'user' donnees.txt
```

— Affichez toutes les lignes contenant une adresse e-mail quelconque (celles qui contiennent le caractère `@`).

```
$ grep '@' donnees.txt
```

— Analysez les résultats des commandes suivantes :

```
$ grep 'info' donnees.txt
$ grep -i 'info' donnees.txt
```

Par défaut, `grep` distingue les majuscules et les minuscules. Avec l'option `-i` (*ignore case*), la recherche devient insensible à la casse.

2.2 Ancrages `^` (début de ligne) et `$` (fin de ligne)

— Analysez les résultats des commandes suivantes :

```
$ grep '$' donnees.txt
$ grep '\$' donnees.txt
$ grep 'H$' donnees.txt
$ grep 'HH$' donnees.txt
$ grep '000$' donnees.txt
```

Dans une expression régulière, le symbole `$` ne signifie pas “caractère dollar” : il signifie “fin de ligne”. Pour chercher un vrai `$` dans le texte, il faut utiliser `\$`.

— Analysez les résultats des commandes suivantes :

```
$ grep '^' donnees.txt
$ grep '\^' donnees.txt
$ grep '^e' donnees.txt
$ grep -i '^e' donnees.txt
$ grep -i '^er' donnees.txt
```

En expression régulière, le symbole `^` ne signifie pas “caractère `^` littéral” : il signifie “début de ligne”. Pour chercher un vrai `^` dans le texte, il faut utiliser `\^`.

— Analysez les résultats des commandes suivantes :

```
$ grep '[\$]' donnees.txt
$ grep '[$]' donnees.txt
$ grep '[^]' donnees.txt
$ grep '[\^]' donnees.txt
$ grep '^M' donnees.txt
$ grep '^[M]' donnees.txt
$ grep '^[^M]' donnees.txt
$ grep '[^Z]' donnees.txt
$ grep '[Z^]' donnees.txt
```

`^` et `$` servent d’ancres (début/fin de ligne) uniquement *en dehors* des crochets `[ ]`. À l’intérieur, ils sont soit de la syntaxe de classe négative (`[^...]`) soit de simples caractères normaux.

2.3 Regex de base (`.`, `[ ]`, `*`)

— Analysez les résultats des commandes suivantes :

```
$ grep '.' donnees.txt
$ grep '\.' donnees.txt
$ grep 'se.' donnees.txt
$ grep 'se\.' donnees.txt
$ grep '2.:' donnees.txt
$ grep '.2.1.w' donnees.txt
$ grep ':..:' donnees.txt
$ grep ':...:' donnees.txt
```

Dans une expression régulière, le caractère `.` signifie “un caractère quelconque”. Pour chercher un *vrai* point, on l’échappe avec un antislash `\`.

— Analysez les résultats des commandes suivantes :

```
$ grep '*' donnees.txt
$ grep '\*' donnees.txt
$ grep 'K*' donnees.txt
$ grep '*K' donnees.txt
$ grep '\*K' donnees.txt
```

```
$ grep 'V*' donnees.txt
$ grep 'V\*' donnees.txt
$ grep 'V*V' donnees.txt
$ grep 'V*VV' donnees.txt
$ grep 'tel: 01*' donnees.txt
```

Dans une expression régulière, le caractère `*` n'est pas un caractère normal : c'est un "quantificateur" qui signifie "zéro ou plusieurs occurrences de l'élément précédent". Pour chercher un *vrai* caractère `*`, on l'échappe avec un antislash `\` (si `*` est precede par un autre character).

— Analysez les résultats des commandes suivantes :

```
$ grep '[' donnees.txt
$ grep '\[' donnees.txt
$ grep '[[]' donnees.txt
$ grep '[aeiou]' donnees.txt
$ grep '[x-z]' donnees.txt
$ grep '[x-zX-Z]' donnees.txt
$ grep '[0-3]' donnees.txt
$ grep '[^a-zA-Z0-9]' donnees.txt
```

Dans une expression régulière, `[...]` définit une "classe de caractères" : la regex correspond à *un seul caractère* parmi ceux listés entre les crochets.

- `[aeiou]` : une lettre parmi `a`, `e`, `i`, `o`, `u`.
- `[0-9]` : un chiffre (de 0 à 9).
- `[A-Z]` : une lettre majuscule.
- `[A-Za-z0-9_]` : lettre majuscule, ou minuscule, ou chiffre, ou `_`.

On peut aussi utiliser une "classe négative" avec `^` en premier caractère :

- `[^0-9]` : un caractère qui n'est pas un chiffre.

2.4 Quantificateurs : `+`, `?`, `{m,n}`

— Analysez les résultats des commandes suivantes :

```
$ grep 'a*' donnees.txt
$ grep 'a\+' donnees.txt
$ grep 'a+' donnees.txt
$ grep -E 'a\+' donnees.txt
$ grep -E 'a+' donnees.txt
$ grep -E 's+s+' donnees.txt
$ grep -E 'ss+' donnees.txt
$ grep -E '1+2+3+' donnees.txt
$ grep -E '123+' donnees.txt
$ grep -E 'a+t' donnees.txt
```

Dans les regex, `+` désigne "une ou plusieurs occurrences" du motif précédent.
Avec `grep` (regex basiques), il faut l'écrire `\+`. Avec `grep -E` (regex étendues), on écrit simplement `+`.

— Analysez les résultats des commandes suivantes :

```
$ grep '4' donnees.txt
$ grep '4\?' donnees.txt
$ grep -E '4?' donnees.txt
$ grep -E '4\?' donnees.txt
```

Dans les regex, `?` signifie "zéro ou une occurrence" du motif précédent.
Avec `grep` (regex basiques), il faut l'écrire `\?`. Avec `grep -E` (regex étendues), on écrit simplement `?`.

— Analysez les résultats des commandes suivantes :

```
$ grep '[0-9]' donnees.txt
$ grep '[0-9][0-9][0-9][0-9][0-9]' donnees.txt
$ grep '[0-9]{5}' donnees.txt
$ grep '[0-9]\{5\}' donnees.txt
$ grep -E '[0-9]{5}' donnees.txt
$ grep -E '[0-9]\{5\}' donnees.txt
$ grep -E '[0-9]{3,6}' donnees.txt
$ grep -E '[0-9]{4}-[0-9]{2}-[0-9]{2}' donnees.txt
$ grep -E '[0-9]{4}-[0-9]{2}-[0-9]{2}$' donnees.txt
$ grep -E '[A-Z]{6,8}: \$[0-9]{4}' donnees.txt
$ grep -E '[0-9]{2}/[0-9]{2}/[0-9]{4}' donnees.txt
```

Avec `grep` *(sans `-E`)*, on utilise les "regex basiques". La répétition se fait avec `\{m\}` ou `\{m,n\}`.
Avec `grep -E`, on active les "regex étendues". Ici, `{ }` sont directement des quantificateurs, *sans antislash*.

- `{m}` = exactement `m` répétitions.
- `{m,n}` = entre `m` et `n` répétitions.

2.5 Alternatives `|` et groupes `( )`

— Analysez les résultats des commandes suivantes :

```
$ grep 'com' donnees.txt
$ grep 'org' donnees.txt
$ grep 'com\|org' donnees.txt
$ grep -E 'com|org' donnees.txt
$ grep -E 'com|org|fr' donnees.txt
```

Avec `grep` (regex basiques), l'alternative (signifie "ou") s'écrit `\|`. Avec `grep -E` (regex étendues), on écrit directement `|`.

— Analysez les résultats des commandes suivantes :

```
$ grep -E 'date: 2021|2022' donnees.txt
$ grep -E 'date: (2021|2022)' donnees.txt
$ grep 'date: \(2021\|2022\)' donnees.txt
$ grep -E 'date: (2021|2022|2023)' donnees.txt
```

Avec `grep -E`, `( )` servent à regrouper les alternatives. Avec `grep` (sans `-E`), il faut échapper ces symboles : `\( \| \)`.

2.6 Inverser la sélection

— Analysez les résultats des commandes suivantes :

```
$ grep '1' donnees.txt
$ grep -v '1' donnees.txt
$ grep '11' donnees.txt
$ grep -v '11' donnees.txt
$ grep -E '[0-9]' donnees.txt
$ grep -Ev '[0-9]' donnees.txt
```

`grep -v motif` affiche les lignes qui ne correspondent pas au motif. C'est l'inverse de `grep motif`.

— Affichez toutes les lignes qui ne contiennent pas des lettres majuscules.

```
$ grep -Ev '[A-Z]' donnees.txt
```

— Affichez toutes les lignes qui ne contiennent pas des lettres (majuscules ou minusclues) entre `A` et `D`.

```
$ grep -Ev '[A-Da-d]' donnees.txt
```

3 Introduction à `sed` pour de simples remplacements

`sed` est un éditeur de flux, très puissant.
Commande de base pour un remplacement :
- `sed 's/ancien/nouveau/'` remplace la première occurrence de `ancien` par `nouveau` sur chaque ligne.
- `sed 's/ancien/nouveau/g'` remplace toutes les occurrences sur chaque ligne.

Par défaut, `sed` écrit le résultat sur la sortie standard (n'écrase pas le fichier original).

— Affichez le fichier `donnees.txt` en remplaçant `ERROR` par `ERREUR` dans les lignes de logs.

```
$ sed 's/ERROR/ERREUR/g' donnees.txt
```

— Affichez seulement les lignes d'e-mails, en supprimant le préfixe `email:` au début de la ligne. (Nous utilisons ici l'option `-n` et la commande `p` (*print*) pour n'afficher que les lignes modifiées.)

```
$ sed -n 's/^email: //p' donnees.txt
```

— Affichez les lignes de code postal en mettant le texte `code postal:` en majuscules (`CODE POSTAL:`), sans modifier le reste.

```
$ sed 's/^code postal:/CODE POSTAL:/' donnees.txt
```

4 Substitutions, pipes et filtres

Dans cet exercice, nous allons pratiquer plusieurs fonctionnalités importantes du shell : *brace expansion*, substitution de commande, variables, pipes, tests, opérateurs logiques, et quelques commandes classiques de traitement de texte.

4.1 Préparation

— Créez un répertoire `TD3` dans votre répertoire personnel et placez-vous dedans.

```
$ cd $HOME
$ mkdir TD3
$ cd TD3
```

4.2 Brace expansion `{start..end}` et substitution de commande `$(...)`

Brace expansion : `fichier_{1..5}.txt` est développé par le shell en `fichier_1.txt fichier_2.txt fichier_3.txt fichier_4.txt fichier_5.txt`
Substitution de commande : `$(commande)` est remplacé par la sortie de `commande`. Exemple : `AUJ=$(date)`.

— Créez un sous-répertoire `Q1` et placez-vous dedans.

```
$ mkdir Q1
$ cd Q1
```

— Utilisez la *brace expansion* pour créer d'un seul coup 12 fichiers `rapport_01.txt` à `rapport_12.txt`.

```
$ touch rapport_01..12.txt
```

— Vérifiez qu'ils ont bien été créés :

```
$ ls
```

— Utilisez maintenant la substitution de commande pour stocker la date du jour dans une variable.

```
$ AUJOURDHUI=$(date +%F)
$ echo $AUJOURDHUI
```

— À partir de cette variable, créez un fichier de log nommé `log-$AUJOURDHUI.txt`.

```
$ touch log-$AUJOURDHUI.txt
$ ls
```

— Stockez dans une variable le nombre de fichiers `rapport_*.txt` présents dans le répertoire, en utilisant `ls` et `wc -l` avec une substitution de commande.

```
$ N_RAPPORTS=$(ls rapport_*.txt | wc -l)
$ echo $N_RAPPORTS
```

4.3 Pipes (|)

L'opérateur de pipe `|` envoie la sortie standard de la commande de gauche comme entrée standard de la commande de droite.
Exemple : `ls | wc -l` compte le nombre de fichiers listés par `ls`.

— Assurez-vous d'être dans `$HOME/TD3/Q1`.

```
$ cd $HOME/TD3/Q1
$ ls
```

— Comptez le nombre de fichiers `rapport_*.txt` en une seule commande en utilisant `ls`, le pipe `|` et `wc -l`.

```
$ ls rapport_*.txt | wc -l
```

— Stockez ce nombre dans une variable `N_RAPPORTS` en combinant le pipe et la substitution de commande `$(...)`.

```
$ N_RAPPORTS=$(ls rapport_*.txt | wc -l)
$ echo $N_RAPPORTS
```

— Créez un petit fichier texte `noms.txt` contenant quelques prénoms (un par ligne), puis utilisez un pipe pour compter le nombre de lignes.

```
$ cat > noms.txt alice bob charlie diane ^D
$ cat noms.txt | wc -l
```

— Affichez le contenu de `noms.txt` en remplaçant toutes les lettres minuscules par des majuscules, en utilisant `tr` via un pipe.

```
$ cat noms.txt | tr 'a-z' 'A-Z'
```

4.4 Variables, tests `[ ]` / `[[ ]]`, `&&` et `||`

Variables : `X=valeur`, utilisation avec `$X`.
Test simple : `[ condition ]` est une commande qui réussit ou échoue.
Exemples : `[ -f fichier ]` (fichier existe), `[ "$A" -eq 3 ]` (entiers), `[ "$S" = "abc" ]` (chaînes).
Test avancé : `[[ ... ]]` supporte `<`, `>` sur les chaînes.
`&&` et `||` : `cmd1 && cmd2` exécute `cmd2` seulement si `cmd1` réussit. `cmd1 || cmd2` exécute `cmd2` seulement si `cmd1` échoue.

— Créez une variable contenant une commande, par exemple une variante de `ls`.

```
$ LSCMD="ls -1 rapport_*.txt"
$ echo "$LSCMD"
```

— Exécutez la commande contenue dans la variable (en la faisant précéder par `$`).

```
$ $LSCMD
```

— Utilisez la commande `test` ou les crochets `[ ]` pour vérifier que le fichier `rapport_01.txt` existe.

```
$ [ -f rapport_01.txt ] && echo "rapport_01.txt existe"
```

— Testez maintenant si le nombre de rapports (`$N_RAPPORTS`) est égal à 12, en utilisant une comparaison d'entiers (`-eq`) et `&&`.

```
$ [ "$N_RAPPORTS" -eq 12 ] && echo "Tous les 12 rapports sont là"
```

— Testez si l'utilisateur courant n'est pas `root`, en utilisant une comparaison de chaînes.

```
$ [ "$USER" != "root" ] && echo "Vous n'êtes pas root"
```

— Utilisez `[[ ]]` pour faire une comparaison lexicographique entre deux chaînes, par exemple vérifier si `"alice"` est « plus petite » que `"bob"`.

```
$ [[ "alice" < "bob" ]] && echo "alice < bob"
```

— Utilisez `&&` et `||` ensemble pour afficher : `"OK"` si `$N_RAPPORTS` est égal à 12, et `"ERREUR"` sinon.

```
$ [ "$N_RAPPORTS" -eq 12 ] && echo "OK" || echo "ERREUR"
```

4.5 `grep`, `sort`, `uniq`, `wc -l`, `tr`

`grep motif` : affiche les lignes contenant `motif`.
`sort` : trie les lignes.
`uniq` : supprime les doublons consécutifs (souvent après `sort`).
`wc -l` : compte le nombre de lignes.
`tr` : transforme des caractères (par ex. minuscules → majuscules).

— Revenez dans `TD3` et créez un sous-répertoire `Q2`.

```
$ cd $HOME/TD3
$
$ mkdir Q2
$
$ cd Q2
```

— Créez un fichier de logs simulant des connexions à un système :

```
$ cat > logins.log « EOF 2023-03-01 09:01:03 user01 SUCCESS 2023-03-01
09:05:10 user02 FAIL 2023-03-01 09:10:22 user01 SUCCESS 2023-03-01 09:15:47
user03 SUCCESS 2023-03-01 09:20:03 user02 FAIL 2023-03-01 09:25:30 user04
SUCCESS 2023-03-01 09:30:00 user01 FAIL 2023-03-01 09:35:42 user03 SUCCESS
2023-03-01 09:40:11 user02 SUCCESS 2023-03-01 09:45:55 user04 FAIL EOF
```

— Affichez le contenu du fichier :

```
$ cat logins.log
```

— Comptez, avec une seule ligne de commande utilisant un pipe, le nombre de lignes contenant le mot `SUCCESS`.

```
$ grep SUCCESS logins.log | wc -l
```

— Comptez le nombre de lignes contenant le mot `FAIL`.

```
$ grep FAIL logins.log | wc -l
```

— Affichez toutes les lignes contenant `user01`, triées par ordre alphabétique.

```
$ grep user01 logins.log | sort
```

— Affichez toutes les lignes du fichier `logins.log`, triées et sans doublons.

```
$ sort logins.log | uniq
```

— Comptez le nombre de lignes distinctes dans le fichier, en utilisant `sort`, `uniq` et `wc -l`.

```
$ sort logins.log | uniq | wc -l
```

— Affichez le contenu de `logins.log` en transformant toutes les lettres majuscules en minuscules, à l'aide de `tr`.

```
$ cat logins.log | tr 'A-Z' 'a-z'
```

4.6 Extension : expansion arithmétique `$(( ... ))`, `head`, `tail`, `cut`, `xargs`

4.6.1 Expansion arithmétique `$(( ... ))`

`$(( ... ))` permet de faire des calculs sur des entiers.
Exemples : `X=$((2 + 3))`, `Y=$((X * 10))`.

— Dans le répertoire `Q1`, définissez deux variables `A` et `B` puis calculez leur somme.

```
$ cd $HOME/TD3/Q1 $ A=5 $ B=3 $ SOMME=$((A + B)) $ echo $SOMME
```

— Calculez le nombre de secondes dans une journée (24 heures).

```
$ SECONDES_PAR_JOUR=$((24 * 60 * 60)) $ echo $SECONDES_PAR_JOUR
```

— En une seule ligne de commande, incrémentez une variable `N` de 1 et affichez sa nouvelle valeur.

```
$ N=10 $ N=$((N + 1)) && echo $N
```

4.6.2 `head` et `tail`

`head -n N fichier` : affiche les N premières lignes.
`tail -n N fichier` : affiche les N dernières lignes.
`tail -n +K fichier` : affiche à partir de la ligne K.

— Placez-vous dans le répertoire `Q2`.

```
$ cd $HOME/TD3/Q2
```

— Affichez les 3 premières lignes de `logins.log`.

```
$ head -n 3 logins.log
```

— Affichez les 2 dernières lignes de `logins.log`.

```
$ tail -n 2 logins.log
```

— Affichez les 5 premières lignes contenant le mot `FAIL`.

```
$ grep FAIL logins.log | head -n 5
```

— Affichez les 3 dernières lignes contenant `SUCCESS`.

```
$ grep SUCCESS logins.log | tail -n 3
```

4.6.3 `cut` : extraction de colonnes

`cut` permet d'extraire des colonnes d'un texte.
Exemple : `cut -d ';' -f2 fichier` : deuxième colonne, séparateur `;`.

— Créez un fichier `ventes.csv` :

```
$ cat > ventes.csv « EOF date;client;montant 2023-03-01;CLIENTA;1200
2023-03-01;CLIENTB;800 2023-03-02;CLIENTA;500 2023-03-02;CLIENTC;2300
2023-03-03;CLIENTB;1500 EOF
```

— Affichez tout le fichier pour vérifier son contenu.

```
$ cat ventes.csv
```

— Affichez uniquement la colonne des clients (deuxième colonne).

```
$ cut -d ';' -f2 ventes.csv
```

— Affichez uniquement la colonne des montants (troisième colonne).

```
$ cut -d ';' -f3 ventes.csv
```

— Affichez la liste des clients sans doublons, triée par ordre alphabétique.

```
$ cut -d ';' -f2 ventes.csv | tail -n +2 | sort | uniq
```

— Comptez le nombre total de ventes (lignes de données, en excluant l'en-tête).

```
$ tail -n +2 ventes.csv | wc -l
```

Bonus : `xargs`

`xargs` lit des mots sur l'entrée standard et les utilise comme arguments d'une commande.
Exemple : `echo fichier1 fichier2 | xargs cat` = `cat fichier1 fichier2`.

— Dans `Q1`, créez un fichier `liste_rapports.txt` contenant la liste des fichiers `rapport_*.txt` (un par ligne).

```
$ cd $HOME/TD3/Q1 $ ls rapport_*.txt > liste_rapports.txt $ cat
liste_rapports.txt
```

— Affichez le contenu de tous les fichiers listés dans `liste_rapports.txt` en une seule commande utilisant `xargs` et `cat`.

```
$ cat liste_rapports.txt | xargs cat
```

— Dans `Q2`, affichez uniquement les lignes contenant `user01` pour tous les fichiers dont la liste est fournie sur l'entrée standard, en utilisant `xargs` et `grep`. (Par exemple avec `echo logins.log`.)

```
$ cd $HOME/TD3/Q2 $ echo logins.log | xargs grep user01
```

5 Mise en pratique : navigation dans un dépôt d'entreprise

Dans cet exercice, vous allez télécharger un petit jeu de données qui simule les fichiers d'une entreprise (rapports, factures, logs, sauvegardes, etc.), puis utiliser le globbing pour retrouver certaines informations.

— Téléchargez l'archive suivante dans votre répertoire `TD2` :

```
$ cd $HOME/TD2
$
$ # Je définis le lien sur plusieurs lignes car il est trop long
$ UrlGit="https://github.com/AbdAmmar/LDS/blob"
$ Commit="03eff847f0b55cdf2cc8558b6654b5cd7b2cf282"
$ Rep="TD/TD2/data/globbing/entreprise_data.tar.gz"
$
$ # On télécharge maintenant
$ wget "${UrlGit}/${Commit}/${Rep}"
```

Si la commande `wget` n'est pas disponible, installez-la avec :

```
$ # pour Linux :
$ sudo apt update
$ sudo apt install wget
$
$ # pour macOS :
$ brew install wget
```

— Décompressez l'archive et placez-vous dans le répertoire `entreprise_data` :

```
$ tar xf entreprise_data.tar.gz
$ cd entreprise_data
$ ls
```

— Le répertoire contient plusieurs sous-répertoires (par exemple `finances`, `rh`, `ventes`, `it/logs`, `backups`), chacun avec de nombreux fichiers aux noms structurés.

Dans toutes les questions suivantes, utilisez uniquement le globbing (`*`, `?`, `[...]`).
Vous pouvez bien sûr combiner le globbing avec des commandes comme `ls`, `echo`, voire `ls | wc -l` pour compter des fichiers.

1. Lister tous les fichiers PDF du service `finances`. (Rapports trimestriels et factures.)
2. Lister uniquement les rapports trimestriels de 2023 dans `finances` (les fichiers dont le nom contient `rapport-finances` et commence par `2023-`).
3. Parmi ces rapports de 2023, lister uniquement les versions `v1` (les fichiers dont le nom se termine par `-v1.pdf`).

4. Lister toutes les factures 2023 pour le client `CLIENTA` dans le répertoire `finances`.
5. Lister tous les fichiers de salaires (fichiers contenant `salaires` dans leur nom) dans le répertoire `rh`.
6. Toujours dans `rh`, lister uniquement les fichiers de salaires marqués comme `confidentiel` (nom contenant `salaires-confidentiel`).
7. Lister tous les fichiers CSV de chiffres d'affaires (se terminant par `.csv`) dans le répertoire `ventes`, tous mois et toutes régions confondus.
8. Lister uniquement les fichiers de `ventes` correspondant au mois de mars 2023 (nom contenant `2023-03-`).
9. Dans `it/logs`, lister tous les logs de production non compressés (fichiers `app-prod-...` se terminant par `.log`, mais pas `.log.gz`).
10. Dans `it/logs`, lister les logs de production du 15 au 19 janvier 2023 (jours `15`, `16`, `17`, `18`, `19` dans le nom de fichier).
11. Dans `backups`, lister toutes les sauvegardes complètes (`full`) de la base de données pour mars 2023 (fichiers contenant `backup-db-2023-03-` et `-full.tar.gz`).
12. Lister tous les fichiers cachés à la racine du dépôt `entreprise_data` (fichiers dont le nom commence par un point).
13. Lister tous les fichiers qui ne sont *pas* des fichiers texte PDF (c'est-à-dire tous les fichiers dont l'extension n'est pas `.pdf`). Vous pouvez ici utiliser un motif avec négation dans les crochets, par exemple `.[!p][!d][!f]` ou similaire.

6 Récupération des fichiers de données

Dans ce TD, vous allez travailler sur des fichiers texte fournis par l'enseignant. Ces fichiers contiennent des données fictives destinées à l'apprentissage des commandes de filtrage et d'analyse de texte.

Commencez par créer un répertoire `data`, puis téléchargez les fichiers à l'aide des commandes suivantes :

```
$ mkdir data
$ cd data
$ wget https://raw.githubusercontent.com/AbdAmmar/LDS/main/TD/TD2/data/users.txt
$ wget https://raw.githubusercontent.com/AbdAmmar/LDS/main/TD/TD2/data/logs.txt
$ wget https://raw.githubusercontent.com/AbdAmmar/LDS/main/TD/TD2/data/notes.txt
```

Vérifiez que les fichiers ont bien été téléchargés. Si la commande `wget` n'est pas disponible, installez-la avec :

```
$ sudo apt update
$ sudo apt install wget
```

7 Lire et compter

Travail à faire

1. Affichez le contenu du fichier `users.txt`.
2. Comptez le nombre de lignes du fichier.
3. Affichez le contenu du fichier `logs.txt`.
4. Comptez le nombre de lignes du fichier `notes.txt`.

Commandes utiles

```
$ cat users.txt
$ wc -l users.txt
```

8 Introduction aux pipes

Un *pipe* permet d'envoyer la sortie d'une commande en entrée d'une autre.

Travail à faire

1. Comptez le nombre de lignes de `users.txt` sans afficher le contenu du fichier.
2. Expliquez le rôle du symbole `|`.

Commande clé

```
$ cat users.txt | wc -l
```

9 Rechercher des informations avec `grep`

Travail à faire

À partir du fichier `users.txt` :

1. Affichez les lignes contenant `bash`.
2. Comptez le nombre d'utilisateurs utilisant `bash`.
3. Affichez les lignes ne contenant pas `bash`.

Commandes utiles

```
$ grep bash users.txt
$ grep -c bash users.txt
$ grep -v bash users.txt
```

10 Extraire des informations

Le fichier `users.txt` est structuré : les champs sont séparés par le caractère `:`.

Travail à faire

1. Affichez uniquement la colonne des noms d'utilisateurs.
2. Affichez uniquement la colonne des shells.

Commandes utiles

```
$ cut -d: -f1 users.txt
$ cut -d: -f2 users.txt
```

11 Trier et compter

Travail à faire

1. Affichez la liste des shells utilisés.
2. Triez cette liste.
3. Comptez combien de fois chaque shell apparaît.

Pipeline attendu

```
$ cut -d: -f2 users.txt | sort | uniq -c
```

12 Analyse de fichiers de logs

Travail à faire

À partir du fichier `logs.txt` :

1. Affichez uniquement les lignes contenant `ERROR`.
2. Comptez le nombre d'erreurs.
3. Affichez les noms des utilisateurs ayant généré une erreur.

Commandes utiles

```
$ grep ERROR logs.txt
$ grep -c ERROR logs.txt
$ grep ERROR logs.txt | cut -d= -f2
```

13 Mini-analyse

Travail à faire

Sans utiliser d'éditeur de texte :
— Quel est le shell le plus utilisé ?
— Quel utilisateur apparaît le plus souvent dans les logs ?
Justifiez vos réponses à l'aide de commandes Linux.

14 Les flux standards

Ce TD a pour objectif d'introduire la gestion des sorties et des erreurs sous Linux. Sous Linux, un programme communique avec l'extérieur à l'aide de flux :
— l'entrée standard (stdin) ;
— la sortie standard (stdout) ;
— la sortie d'erreur (stderr).
Par défaut, la sortie standard et la sortie d'erreur sont affichées à l'écran.

15 Rediriger la sortie standard

Travail à faire

1. Listez le contenu de votre répertoire personnel.
2. Redirigez cette sortie dans un fichier `liste.txt`.
3. Vérifiez le contenu du fichier.

Commandes

```
$ ls
$ ls > liste.txt
$ cat liste.txt
```

16 Générer et observer une erreur

Travail à faire

1. Essayez d'afficher un fichier qui n'existe pas.
2. Observez le message affiché.

```
$ cat fichier_inexistant.txt
```

Expliquez pourquoi le message n'est pas redirigé avec `>`.

17 Rediriger la sortie d'erreur

La sortie d'erreur peut être redirigée séparément.

Travail à faire

1. Redirigez l'erreur précédente dans un fichier `erreur.txt`.
2. Vérifiez le contenu du fichier.

```
$ cat fichier_inexistant.txt 2> erreur.txt
$ cat erreur.txt
```

18 Rediriger sortie et erreur

Il est possible de rediriger à la fois la sortie standard et la sortie d'erreur.

Travail à faire

1. Lancez une commande produisant à la fois une sortie et une erreur.
2. Redirigez les deux flux dans un même fichier.

```
$ ls fichier_inexistant > sortie.txt 2> erreur.txt
```

Ou en une seule commande :

```
$ ls fichier_inexistant > tout.txt 2>&1
```