R1.04 - Systèmes - TP 5
Filtres

Si vous n'avez pas bien compris le mécanisme des tubes, vous ne pourrez pas avancer sur ce TP. N'attendez pas, **demandez des explications et de l'aide dès maintenant**.

Rappels

Redirections de canaux

Tout processus possède un canal d'entrée **`STDIN`** et un canal de sortie **`STDOUT`**.

Un processus lit sur son **`STDIN`** (par exemple : **`scanf()`** en C) et écrit sur son **`STDOUT`** (par exemple : **`printf()`** en C).

Par défaut, quand on lance une commande depuis un *Terminal*, **`STDIN`** est le clavier et **`STDOUT`** est l'écran.

Si on redirige **`STDIN`** depuis un fichier de cette façon :

```
une_commande < un_fichier
```

ça détache **`STDIN`** du clavier pour l'attacher à un fichier, qui devient la source d'alimentation du **`STDIN`** (des **`scanf()`** en C). Ce qui revient à simuler des frappes qui auraient pu être faites au clavier.

Si on redirige **`STDOUT`** vers un fichier de cette façon :

```
une_commande > un_fichier
```

ça détache **`STDOUT`** de l'écran pour l'attacher à un fichier, qui devient la destination du **`STDOUT`** (des **`printf()`** en C).

Tubes

Sur le même principe que les redirections, on peut brancher les canaux STDIN et STDOUT, non plus avec des fichiers mais avec des canaux d'autres processus.

On branche alors la sortie de l'un avec l'entrée d'un autre, comme on le ferait avec un tuyau. On appelle d'ailleurs cela un tube (pipe en anglais).

La syntaxe d'un tube est la suivante :

```
une_commande | une_autre_commande
```

Le **STDOUT** de **une_commande** alimente le **STDIN** de **une_autre_commande**.

Évidemment, **une_autre_commande**, comme tout processus, a aussi son propre **STDOUT**. Dans l'exemple précédent, il s'agit de l'écran (canal par défaut), mais rien n'empêche de rediriger son **STDOUT** vers un fichier, de cette façon :

```
une_commande | une_autre_commande > un_fichier
```

ou vers une autre commande, de cette façon :

```
une_commande | une_autre_commande | une_troisieme_commande
```

Vous l'aurez compris, ce chaînage peut être poursuivi autant de fois que nécessaire, chaque commande venant s'alimenter de ce que la précédente lui donne.

C'est ce mécanisme qui est la base du sujet de ce TP : **les filtres**.

IMPORTANT :
• Seule la **1ère commande** peut avoir une redirection entrante de la forme : **<**
• Seule la **dernière commande** peut avoir une redirection sortante de la forme : **>**

Ainsi, ceci est autorisé :

```
cmd1 <[1] source | cmd2 | cmd3 | cmd4 >[2] resultat
```

[1] Le **STDIN** de **cmd1** est alimenté par un fichier **source** et le **STDOUT** de **cmd1** alimente le **STDIN** de **cmd2**
[2] le **STDIN** de **cmd4** est alimenté par le **STDOUT** de **cmd3** et son **STDOUT** va dans le fichier **resultat**

Ceci est incohérent et donc interdit :

```
cmd1 >[3] dest | cmd2 | cmd3 | cmd4
cmd1 | cmd2 | cmd3 | cmd4 <[4] source
cmd1 | cmd2 <[5] source | cmd3 | cmd4
cmd1 | cmd2 | cmd3 >[6] dest | cmd4
```

Il est interdit d'écrire ces choses mais si vous le faites, vous n'aurez pas de message d'erreur[7], juste un comportement bizarre, qui peut aller jusqu'au blocage apparent de la commande, dont vous vous sortirez généralement par un **CTRL+C**

Filtres

Nous allons détailler chaque filtre avant de passer à des exercices de mise en pratique.

Par rapport aux TP précédents qui étaient très orientés Tuto, vous allez être moins guidés pour laisser plus de place à la réflexion, maintenant que vous commencez à avoir une petite maîtrise de la ligne de commande et des principes fondamentaux de Linux.

Bien évidemment, votre enseignant.e est toujours là pour vous aider. N'hésitez pas à faire appel à lui/elle pour comprendre ou pour faire confirmer si vous avez un doute. Continuez de prendre des notes de synthèse de ce que vous apprenez, le DS approche…

Principe de fonctionnement

Avant de voir en détail les filtres, il est important de comprendre ce qu'est un filtre et comment il fonctionne.

Un filtre reçoit des données (du texte généralement) sur son **STDIN** et leur applique une éventuelle transformation pour produire, sur son **STDOUT**, le résultat de sa transformation (aussi du texte généralement).

STDIN, **STDOUT**, vous l'avez compris, les filtres s'utilisent au moyen de *tubes*.

> Un filtre ne modifie **JAMAIS** la source de données. Il produit sur son **STDOUT** une version (éventuellement) modifiée de ce qui lui a été donné sur son **STDIN**.

[3] Le **STDOUT** de **cmd1** ne peut pas alimenter en même temps un fichier **dest** et une commande **cmd2**
[4] Le **STDIN** de **cmd4** ne peut pas être alimenté à la fois par un fichier **source** et une commande **cmd3**
[5] Le **STDIN** de **cmd2** ne peut pas être alimenté à la fois par le **STDOUT** de **cmd1** et un fichier **source**
[6] Le **STDOUT** de **cmd3** ne peut pas alimenter à la fois le fichier **dest** et le **STDIN** de **cmd4**
[7] Ce n'est pas une erreur de syntaxe mais une erreur de logique.

Analogie du mécanisme

Imaginons un *filtreur* humain dont la compétence est de savoir sortir d'une caisse toutes les choses d'une certaine nature.

Si vous donnez à cette personne une caisse de fruits (des prunes, des pommes, des poires) en lui indiquant que vous avez besoin des pommes, il vous rendra alors uniquement les pommes.

Supposons maintenant que vous ayez un autre *filtreur* humain dont la compétence est de savoir sortir d'une caisse toutes les choses d'une couleur donnée.

Si vous donnez à cette personne un panier de choses quelconques (des fruits, des livres, des brosses à dents) de différentes couleurs (rouges, jaunes, vertes), il vous rendra uniquement les choses rouges, peu importe leur nature.

Maintenant si vous ordonnez à vos deux filtreurs de se mettre en chaîne (l'un fournit le résultat de son travail de filtrage au second), que vous donnez au 1er un panier de fruits de plusieurs couleurs (pommes jaunes, rouges et vertes, et idem avec des prunes et des poires), et que vous demandez à vos filtreurs :

- 1er : des pommes
- 2ème : des choses rouges

vous obtenez alors, en bout de chaîne, uniquement les pommes rouges.

C'est un travail collaboratif, à la chaîne[8].

Avec les filtres c'est exactement la même chose et la mise en chaîne se fait tout simplement par l'usage de tubes.

C'est un mécanisme extrêmement simple et puissant que nous offre Unix.

IMPORTANT : Ne pas confondre filtrer et trier !
• Un filtre réduit (éventuellement) un ensemble de données en un sous-ensemble suivant des critères indiqués. En outre, les données qui en ressortent peuvent être modifiées par rapport à celles qui sont entrées.
• Un tri ordonne un ensemble de données suivant des critères indiqués. La quantité de données reste la même en sortie.

Fichier et STDIN

Notez que les filtres s'utilisent généralement avec des tubes, donc sous cette forme :

[8] C'est finalement le principe du travail sur une chaîne de montage automobile par exemple.

```
une_commande | un_filtre
```

Mais il faut savoir que tous les filtres peuvent aussi s'utiliser seuls, de cette façon :

```
un_filtre < un_fichier
```

ou encore en passant juste un nom de fichier en paramètre :

```
un_filtre un_fichier
```

Pourquoi des filtres

Reprenons l'analogie de nos deux filtres humains précédents. On pourrait aussi apprendre à une troisième personne à sélectionner explicitement un type de fruit d'une couleur donnée. Ca reviendrait au même pour le besoin très spécifique de la sélection des pommes rouges, mais au prix d'une formation spécifique d'une personne qui ne serait alors utile que si on veut sélectionner des fruits en s'intéressant à leur couleur. Comment fait-on si on veut sélectionner les livres verts ? Notre trieur n'a pas la compétence requise.

Plus on rend une compétence spécifique, plus on restreint le champ des possibilités.

Il est plus intéressant de se cantonner à des compétences très basiques mais qu'on peut combiner entre elles.

Il en va de même pour les filtres Unix.

On aurait pu écrire des commandes très spécifiques pour réaliser des tâches spécifiques mais il aurait alors fallu multiplier ces commandes pour remplir tous les besoins communs. Et encore, on aurait obligatoirement fait des choix sur ce qui est utile et commun et ce qui relève d'un besoin plus rare, créant forcément des manques qu'il faudrait combler ponctuellement par le développement de nouvelles commandes. C'est sans fin !

Ces filtres ont des fonctions (des compétences) très basiques et c'est la combinaison des filtres entre eux qui rend le mécanisme très puissant et polyvalent.

Fichiers de test sur Moodle

Récupérez les fichiers de test sur Moodle pour expérimenter les exemples de chaque filtre.

Les fichiers que vous allez exploiter avec les filtres sont généralement des fichiers textes et sont de deux types.

Fichiers “au kilomètre”

Ce sont des fichiers de texte brut, comme peut l’être ce document que vous êtes en train de lire : des paragraphes de texte dont la longueur, le nombre et la structure sont quelconques et impossibles à prévoir.

Exemples : les fichiers **`murphy`** et **`lorem`**.

Fichiers structurés

Ce sont des fichiers de texte qui suivent un formatage homogène.

1. **Exemple 1** - Voici un extrait du fichier **`prod`** :

```
pomme:2:rouge:1,35:nzelande
cerise:6:rouge:3,25:france
```

Chaque ligne[9] correspond à un produit. On appelle cela un **Enregistrement**.

Chaque enregistrement est composé de **Champs** séparés par un **Séparateur**[10]. Si on a deux séparateurs consécutifs (ici ce serait **`::`**), le champ est donc vide.

Voici les champs du fichier **`prod`** :

- Nom du produit
- Quantité
- Couleur
- Prix (numérique)
- Provenance

2. **Exemple 2** - Voici un extrait du fichier **`lang`** :

```
C++         1983 compilé    code natif
Forth       1978 compilé    machine virtuelle
```

[9] Une ligne (un enregistrement) se termine par un **`\n`** (i.e. : **`printf("...\n")`**) Mais la dernière ligne du fichier peut ne pas avoir de **`\n`**, ce sera quand même un enregistrement.
[10] Un caractère quelconque. Ex. : Tabulation (**`\t`**), virgule, point-virgule, deux points (**`:`**), etc.

Encore une fois chaque ligne correspond à un langage de programmation. On appelle toujours cela un **Enregistrement**.

Chaque enregistrement est composé de **Champs** mais ici il n'y a pas de séparateur. Les champs sont à des positions fixes :

- Les 11 premiers caractères = nom du langage (10 symboles et 1 espace)
- Les 5 suivants = date de création (4 chiffres et 1 espace)
- Les 11 suivants = type de langage, **`compilé`** ou **`interprété`**, (10 lettres et 1 espace)
- Et enfin, le reste de la ligne est le mode d'exécution, **`code natif`** ou **`machine virtuelle`**.

Les champs dont le contenu est plus court que l'espace qui leur est alloué sont complétés par des espaces. Par exemple, le 1er champ du 1er enregistrement de **`lang`** vaut **`C++(et 8 espaces)`**.

Conseils

Dans les exemples qui sont donnés pour chaque filtre, **commencez par exécuter seule la commande avant le `|` (`pipe`)[11] afin de comprendre déjà ce qu'elle produit**. Dans un second temps, exécutez l'intégralité de la commande avec la partie **`pipe`** pour avoir le résultat produit par le filtre. Ainsi vous comprendrez mieux ce résultat.

Comme toujours, consultez le **`man`** de chaque filtre pour plus de détails sur les options de chacun.

Vous noterez par écrit les commandes des différentes questions.

wc - Compter[12]

Cette commande compte des… mots, oui, mais pas que des mots[13] ! Elle permet aussi de compter des lignes et des caractères.

On a déjà eu l'occasion de l'utiliser dans un TP précédent.

Syntaxe

`wc -l`	Compte les lignes (**`l`** = lines[14])
`wc -c`	Compte les caractères (**`c`** = characters)
`wc -w`	Compte les mots (**`w`** = words)

[11] Cela signifie : exécutez la commande sans la partie : **`| le_filtre`** qui suit la commande.
[12] **`wc`** signifie **W**ord **C**ount
[13] Encore une bonne blague de nos chers barbus !
[14] C'est un L minuscule, pas un 1.

Exemples

`cat lang prod | wc -l` Compte le nombre total de la globalité des lignes des fichiers **`lang`** et **`prod`** : **`cat lang prod`** permettant d'envoyer sur son **`STDOUT`** le contenu des fichiers **`lang`** et **`prod`** qui est ensuite consommé par **`wc`** sur son **`STDIN`**, pour produire finalement le résultat de son comptage (de lignes) sur son **`STDOUT`**

Q.1 Quelle commande vous permet d'afficher le nombre total de caractères de la globalité de ces deux fichiers ?

Q.2 Quelle commande vous permet d'afficher le nombre total de mots de la globalité de ces deux fichiers ?

sort - Trier

Cette commande trie des lignes de texte suivant des critères passés en option.

NB : Relire le bloc encadré noir de la page 4 concernant la différence entre filtre et tri.

Syntaxe

`sort -t <sep> -k <champ_deb,champ_fin>`

Trie les lignes (enregistrements) sur les valeurs du ou des champs dont les numéros vont de **`champ_deb`** à **`champ_fin`**. Le séparateur de champs est **`sep`**. Si **`-t <sep>`** n'est pas spécifié, chaque bloc de caractères *blancs*[15] est un séparateur. Bloc = si on rencontre plusieurs blancs consécutifs, ils seront globalement pris comme un unique séparateur entre deux champs.

Options utiles

`-r` Tri inversé

`-n` Tri numérique. Par défaut, les champs sont considérés comme du texte. Ainsi **`10`** arrive avant **`2`** car le caractère **`1`** est avant le caractère **`2`**[16], comme **`Andromaque`** arrive avant **`Belphégor`**. Avec cette option, les champs sont considérés comme numériques et **`10`** arrivera alors après **`2`** dans ce cas.

[15] Un caractère *blanc* = espace, tabulation, tabulation verticale

[16] C'est l'ordre lexicographique pour tous les symboles (table ASCII) ou ordre alphabétique quand on n'a que des lettres. Il est donc différent de l'ordre numérique qui s'intéresse à la valeur numérique et non pas aux caractères.

-u	Ne produit que des lignes uniques, les autres lignes en double sur les valeurs de leurs champs de tri sont omises.

Exemples

sort -t':' -k 3 < prod	Trie ce qui arrive sur **STDIN** (ici le fichier **prod**) et envoie sur **STDOUT** le résultat du tri en utilisant le caractère **:** comme séparateur de champ et en triant sur le 3ème champ (la couleur) et les suivants.

A savoir

Le **sep**arateur doit être un caractère unique, pas une séquence, pas un mot.

Le **sep**arateur peut être collé au **-t**. Exemple : **-t':'** et **-t ':'** sont équivalents.

Il n'est pas nécessaire de placer le **sep**arateur entre apostrophes mais c'est conseillé.

Il est **absolument nécessaire** de placer le **sep**arateur entre apostrophes s'il s'agit d'un caractère blanc. Exemple : **-t ' '**

Si on ne spécifie aucune option **-k**, alors les lignes sont triées sur le contenu intégral de chaque ligne.

Si on ne spécifie pas de **champ_fin**, alors le tri se fait en utilisant tous les champs à partir de **champ_deb**.

Le tri peut se faire sur plusieurs champs : d'abord 1er champ puis, si égalité entre deux enregistrements sur le 1er champ, on utilise le 2ème champ pour les départager etc. La syntaxe est alors de faire figurer plusieurs options **-k**. Exemple : **-k3,3 -k1,1** permet de trier sur le champ **3** et ensuite sur le champ **1**.

Lors de l'utilisation du **-u** (unique), seuls les champs impliqués dans le tri[17] sont pris en considération pour déterminer l'unicité. Ainsi, même si les autres champs sont différents, ils n'entrent pas en compte dans l'élimination des doublons.

Q.3 Quelle commande vous permet de trier **prod** sur la provenance ?

Q.4 Quelle commande vous permet de trier **prod** sur la provenance et ensuite sur le nom ?

Q.5 Quelle commande vous permet de trier **prod** sur la quantité ? (Attention, piège)

Q.6 Quelle commande vous permet de trier **prod** sur la provenance en ne gardant qu'une seule ligne par pays ?

[17] Les champs listés dans les **-k**

uniq - Dédoublonner

Cette commande produit en sortie ce qui lui est donné en entrée mais en omettant les lignes qui seraient en double[18].

Attention, cette commande ne fonctionne que sur une source déjà triée.

Par défaut, lors du traitement d'unicité, les lignes sont prises dans leur intégralité, il n'y a pas de notion de champ[19] pour cette commande.

Syntaxe

`uniq`	On ne peut pas faire plus simple !

Options utiles

`-i`	Minuscules et majuscules sont équivalentes
`-d`	Fait l'inverse : produit en sortie uniquement les lignes qui apparaissent plusieurs fois[20].

Exemples

`uniq < depts`	Élimine en sortie les lignes en double
`uniq -d < depts`	Affiche en sortie uniquement les lignes en double, mais 1 seul exemplaire par doublon, peu importe le nombre de fois où le doublon est présent.

A savoir

Comme **`uniq`** nécessite une source de données triée, il est fréquent de préparer la source à l'aide d'un **`sort`**. Or, **`sort`** possède aussi une option (**`-u`**) qui permet de ne conserver qu'un exemplaire de chaque ligne en doublon. Du coup, **`uniq`** est d'une utilité relative. Elle n'est utile que si on a déjà la certitude absolue que la source est triée.

Q.7 En vous aidant du **`man`**uel, quelle commande vous permet d'afficher les lignes uniques du fichier **`depts`** en affichant devant chacune le nombre de fois où elle apparaît dans le fichier ?

[18] En cas de doublon, un seul exemplaire de la ligne est produite en sortie.
[19] Il existe une option **`-f`** qui permet de travailler sur des champs.
[20] **`-d`** : duplicate

colrm[21] - Supprimer des colonnes

Cette commande est utile principalement sur des fichiers textes avec des champs à positions fixes, comme le fichier **lang**.

Elle permet de supprimer[22] des colonnes de caractères dans ce qui est produit en sortie.

Syntaxe

colrm col1 [col2]	Omet en sortie, tout ce qui est situé entre les colonnes **col1** et **col2**. Si **col2** n'est pas spécifié, elle omet tout entre **col1** et la fin de chaque ligne.

Exemples

colrm 12 16 < lang	Omet en sortie la partie date du fichier **lang**

A savoir

La 1ère colonne est numérotée **1**.

Q.8 Quelle commande permet d'afficher la liste des départements du fichier **depts**, sans les numéros, en ne gardant juste que les noms ?

cut - Couper des champs

Cette commande est un faux-ami de la précédente. Il ne faut pas les confondre.

Elle permet, non pas de supprimer (omettre) des parties de lignes mais de couper[23] les champs qu'on souhaite conserver dans ce qui est produit en sortie.

Les champs qui ne sont pas coupés sont simplement omis du résultat produit en sortie, sur **STDOUT**.

Contrairement à **colrm** qui travaille sur des fichiers à champs fixes, **cut** travaille, comme **sort**, sur des fichiers avec des champs séparés par des séparateurs. Le fichier **prod** est un exemple de ce type de fichiers.

Attention au piège : même si **cut** et **sort** utilisent tous deux des champs pour travailler, les options de ces deux commandes sont différentes. Il est facile et fréquent de les confondre, donc notez bien cela dans votre fiche de synthèse !

[21] **colrm** signifie **Col**umns **R**e**m**ove
[22] Rappel : rien n'est supprimé de la source. "Supprimer" signifie plutôt : ne pas conserver dans ce qui est produit en sortie.
[23] Un peu dans l'idée, même si ce n'est pas cela, d'un couper-coller dans un éditeur de texte.

Syntaxe

`cut -d <sep> -f <champs>`

Extrait de chaque ligne (enregistrements) les valeurs du ou des champs dont les numéros sont donnés par **`champs`**. Le séparateur de champs est **`sep`**. Si **`-d <sep>`** n'est pas spécifié, la tabulation (**`\t`**) est le séparateur.

Exemples

`cut -d':' -f 3 < prod` Extrait de ce qui arrive sur **`STDIN`** (ici le fichier **`prod`**) uniquement le champ **`3`** (la couleur) et envoie cet unique donnée restante sur **`STDOUT`**, en utilisant le caractère **`:`** comme séparateur de champs.

`cut -d':' -f 3- < prod` Extrait les champs à partir du **`3`**ème (la couleur et ceux qui suivent)

A savoir

Le **`sep`**arateur doit être un caractère unique, pas une séquence, pas un mot.

Les champs peuvent être spécifiés sous différentes formes :

- **`N`** : Le **`N`**ème
- **`N-`** : Le **`N`**ème et les suivants
- **`N,M`** : Les **`N`**ème et **`M`**ème (on peut en ajouter d'autres séparés par des virgules)
- **`N-M`** : Du **`N`**ème au **`M`**ème (inclus)
- **`N-M,X-Y`** : Du **`N`**ème au **`M`**ème et du **`X`**ème au **`Y`**ème (et ainsi de suite)

Q.9 Quelle commande permet d'extraire du 1er au 3ème champ dans le fichier **`prod`** ?

Q.10 Quelle commande permet d'extraire le nom du département dans le fichier **`depts`** ? (Attention au piège)

head - Garder des lignes de tête

Cette commande permet de produire en sortie les lignes se trouvant en tête de la source donnée en entrée.

Syntaxe

`head -n <nb>` Produit en sortie une extraction des **`nb`** lignes du début de la source d'entrée.

`head -<nb>` Une autre façon de faire la même chose, on peut omettre le **`-n`**.

Exemples

`head -n 2 < prod`	Extrait et affiche (sur **`STDOUT`**) les 2 premières lignes de **`STDIN`** (ici le fichier **`prod`**).
`head -3 < depts`	Idem avec les 3 premières lignes **`depts`** (autre forme, sans le **`-n`**).

A savoir

Sans option **`-<nb>`** ou **`-n <nb>`**, le nombre de lignes par défaut est **`10`**.

Si le nombre de lignes demandé dépasse le nombre total de lignes de la source, il n'y a pas d'erreur.

Q.11 Quelle combinaison de commandes[24] permet d'extraire les 3 premiers langages de **`lang`**, par ordre alphabétique ? Attention, cette fois-ci on complique un tout petit peu le jeu.

tail - Garder des lignes de queue

Cette commande est le pendant de **`head`** mais cette fois-ci elle concerne les lignes de la fin de la source de données.

Syntaxe

`tail -n <nb>`	Produit en sortie une extraction des **`nb`** lignes de la fin de la source d'entrée.
`tail -<nb>`	Une autre façon de faire la même chose, on peut omettre le **`-n`**.

Options utiles

`-f`	Permet de laisser la commande **`tail`** tourner sans fin pour qu'elle puisse afficher ensuite toute nouvelle ligne qui apparaîtra, au fil du temps, dans la source de données. Pratique pour afficher des logs qui arrivent de temps en temps. Un **`CTRL+C`** met fin à la commande.

Exemples

`tail -5 < murphy`	Extrait les **`5`** dernières lignes de **`STDIN`** (ici le fichier **`murphy`**)
`tail -f < lorem`	Extrait les **`10`** dernières lignes de **`STDIN`** (ici le fichier **`lorem`**) et reste en attente de nouvelles lignes qui pourraient être

[24] Combinaison mais en une seule ligne !

ajoutées à la source. Essayez d'ajouter 2-3 lignes à la main (depuis VSC) pour observer l'effet sur cette commande. Terminez-la ensuite par un **CTRL+C**.

A savoir

Sans option **-<nb>** ou **-n <nb>**, le nombre de lignes par défaut est **10**.

Si le nombre de lignes demandé dépasse le nombre total de lignes de la source, il n'y a pas d'erreur.

Q.12 Quelle combinaison de commandes permet d'extraire les lignes 4 à 6 du fichier **depts** ?

tr - Convertir

Cette commande permet de convertir des caractères de la source de données en d'autres caractères.

Syntaxe

tr liste_cars_source liste_cars_rempl

Convertit chaque caractère présent dans la liste **liste_cars_sources** par le caractère de **liste_cars_rempl** situé à la même position.

tr -d liste_cars Supprime chaque caractère présent dans la liste **liste_cars**.

Options utiles

-s Permet de supprimer toute redondance successive du même caractère de la **liste_cars** pour n'en garder qu'un seul à chaque fois.

Exemples

tr , ' ' < murphy Remplace par des espaces toutes les virgules des données arrivant sur **STDIN** (ici le fichier **murphy**).

tr '0123456789' _ < depts

Remplace les chiffres de la source **depts** par des **_**. Les 2 listes (**source** et **rempl**) peuvent être de tailles différentes. Si **rempl** est plus courte, le dernier caractère est celui qui remplace ceux manquants. Donc ici le **_** sert comme unique caractère de remplacement pour tous.

A savoir

Les listes de caractères peuvent utiliser des intervalles similaires à ceux des jokers du Shell, sans les crochets **`[]`**. Ainsi : **`tr a-z A-Z`** convertit les minuscules en majuscules.

Q.13 Quelle commande permet de produire l'affichage d'une version du fichier **`lorem`** entièrement en majuscules ?