Initiation à l'environnement Unix : TP7

novembre 2023 — Pierre Rousselin

1 Tubes

Exercice 1 : Premiers pas

Dans cet exercice, on va se servir du fichier `/etc/passwd` qui liste tous les utilisateurs (personnes physiques ou non) du système et donne certaines informations sur eux.

1. Afficher le contenu du fichier `/etc/passwd`. Comment sont formatées les données ? Quel caractère est utilisé pour délimiter les différents champs ? Lire ce que signifient les différents champs dans

```
$ man 5 passwd
```

2. La commande `cut` permet de sélectionner certains champs et de n'afficher que ceux-ci. La commande suivante (à tester)

```
$ cut -d : -f 1,3-5 /etc/passwd
```

indique (option `-d`) que le délimiteur de champ est `:`, et que l'on veut sélectionner (`-f` comme *fields*, champs) les champs 1, et de 3 à 5.

a) Afficher uniquement les utilisateurs et les shells de connexion des utilisateurs.

b) Afficher uniquement les uid et les gid des utilisateurs.

3. La commande `grep` permet de sélectionner les lignes qui contiennent une certaine chaîne dans un ou plusieurs fichier. La commande suivante (à tester) permet de n'afficher que les lignes qui contiennent le mot `bash`

```
$ grep bash /etc/passwd
```

Afficher uniquement les lignes contenant la chaîne `daemon`.

4. La commande `sort` permet de trier un fichier. Par défaut, c'est l'ordre alphabétique qui est utilisé. Essayer la commande

```
$ sort /etc/passwd
```

Avec l'option `-n`, `sort` trie selon les valeurs numériques. On va combiner les commandes `cut` et `sort` pour obtenir la liste triée des uid sur le système : tester la commande

```
$ cut -d : -f 3 /etc/passwd | sort -n
```

Voir la différence en enlevant l'option `-n`.

5. Afficher la liste triée de tous les shells de connexion. Y a-t-il des doublons ? Filtrer la sortie de `sort` avec les commandes :

a) `uniq`

b) `uniq -c`
Combien d'utilisateurs ont pour shell de connexion `nologin` ?

6. Afficher la liste triée de tous les noms d'utilisateurs dont le shell de connexion est `nologin`.

7. Afficher la liste triée de tous les gid, puis chercher le groupe qui est le plus populaire. Chercher, dans le fichier `/etc/group`, le nom du groupe correspondant.

--- * ---

Correction de l'exercice 1 :

1. Les différents champs sont séparés par des `:`. Le premier est le nom d'utilisateur, suivi d'un éventuel mot de passe (en fait, plus du tout dans ce fichier), de l'uid, du gid, éventuellement du nom complet, du chemin vers le home et du shell de connexion.

2.
```
$ cut -d : -f 1,7 /etc/passwd
$ cut -d : -f 3,4 /etc/passwd
```

3.
```
$ grep daemon /etc/passwd
```

4.
5. ```
   $ cut -d : -f 7 | sort
   $ cut -d : -f 7 | sort | uniq
   $ cut -d : -f 7 | sort | uniq -c
   ```
6. ```
   $ grep nologin | cut -d : -f 1 | sort
   ```
7. ```
   $ cut -d : -f 4 | sort -n | uniq -c
   ```
   On voit le gid le plus utilisé et on peut
   ```
   $ grep <ce gid> /etc/group
   ```

--- * ---

Exercice 2 :

Traduire en langage naturel les commandes suivantes puis vérifier en les exécutant :

1. `head -n 5 /usr/include/stdio.h | tr a-z A-Z`
2. `head -n 10 /usr/include/stdio.h | tail -n 3`
3. `head -n 10 /usr/include/stdio.h | tail -n 3 | tr -s ' ' '\n'` Remarque : cette dernière commande `tr` change les espaces en saut de ligne en supprimant (option `-s`) les sauts de ligne doublons ajoutés de cette manière.
4. `grep FILE /usr/include/stdio.h | wc -l`
5. `grep FILE /usr/include/*.h | wc -l`
6. `printf '%s\n' /usr/include/std*.h | wc -l`
7. `wc -c /usr/bin/* | sort -nr | head -n 20`
8. (Note : il y a de bien meilleures façon d'obtenir les informations suivante, c'est juste pour l'intérêt pédagogique, et pour rigoler) :
   `ls -l /etc | tail -n +2 | cut -c 1 | sort | uniq -c`

--- * ---

Correction de l'exercice 2 :

1. Afficher les 5 premières lignes, en majuscules, du fichier `stdio.h`.
2. Afficher les lignes 8 à 10 du fichier `stdio.h`.
3. Afficher les mots contenus dans les lignes 8 à 10 du fichier `stdio.h`, à raison d'un mot par ligne.
4. Affiche le nombre de lignes de `/usr/include/stdio.h` contenant `FILE`.
5. Affiche le nombre de lignes de tous les fichiers `.h` de `/usr/include` contenant le mot `FILE`.
6. Affiche le nombre de fichiers du répertoire `/usr/include/` dont le nom commence par `std` et se termine par `.h` (noter le développement de noms de fichiers et la réutilisation du format du `printf`)
7. Affiche les 20 fichiers les plus gros, avec leurs tailles en nombre d'octet, du répertoire `/usr/bin/` (en fait il y a aussi la ligne `Total` qui pollue l'affichage et nous casse les pieds).
8. Affiche le nombre de répertoires (après `d`), de fichiers normaux (après `-`) et de liens symboliques (après `l`) dans contenus dans le répertoire `/etc`. La commande `tail -n +2` ne sert qu'à supprimer la méchante ligne du total.

--- * ---

Exercice 3 : Frankenstein

1. Télécharger le roman Frankenstein de Mary Shelley (mis à disposition par le site merveilleux `https://www.gutenberg.org/`) avec l'une des deux commandes suivantes :
   ```
   $ wget https://www.math.univ-paris13.fr/~rousselin/unix/frankenstein.txt
   ```
   ```

ou bien

```
$ curl https://www.math.univ-paris13.fr/~rousselin/unix/frankenstein.txt \
  -o frankenstein
```

2. Le but du jeu est maintenant d'écrire une commande qui affiche les 100 mots les plus utilisés (avec leurs effectifs) dans le roman *Frankenstein* de Mary Shelley. Télécharger `Frankenstein.txt` sur l'ENT (source : `https://www.gutenberg.org/`). On vous aide :
   — la commande `tr -cs '[:alpha:]' '\n'` transforme les caractères **non** (option `-c` pour complémentaire) alphabétiques de son entrée en fins de lignes, puis transforme chaque suite de plusieurs `'\n'` consécutifs en un seul `'\n'` (option `-s`) ;
   — il vaut mieux changer les majuscules en minuscules pour bien compter les occurrences de chaque mot ;
   — l'option `-r` (pour *reverse*) de `sort` permet d'inverser l'ordre.
3. Combien de fois apparaît le mot « frankenstein » ? et le mot « moon » ?
4. Combien de mots apparaissent une seule fois ? Indice : utiliser le filtre `awk '$1 == 1'` qui sélectionne les lignes dont le premier champ est `1`.

--- * ---

Correction de l'exercice 3 :

1. `tr -cs '[:alpha:]' '\n' <frankenstein.txt | tr '[:upper:]' '[:lower:]' | sort | uniq -c | sort -nr | head -n 100`
2. Même chose en remplaçant `head -n 100` par `grep frankenstein` puis par `grep moon`.
3. `tr -cs '[:alpha:]' '\n' <frankenstein.txt | tr '[:upper:]' '[:lower:]' | sort | uniq -c | sort -nr | awk '$1 == 1' | wc -l`

--- * ---

2 Statut de sortie

Exercice 4 : Statut de sortie de `ls` et de `mkdir`

1. Pour chacune des commandes suivantes, l'éxécuter, puis voir son statut de sortie en développant le paramètre spécial `$?`.
   a) `ls /`
   b) `ls / /azerty`

   Laquelle de ces commandes se passe anormalement d'après `ls` et pourquoi ? Quel est le statut de sortie correspondant ?
2. Créer l'arborescence suivante (composée uniquement de répertoires).

```
a
|-- b
|   |-- q
|   `-- s
`-- c
    |-- q
    `-- s
```

   Lancer la commande `ls -R a` et voir son statut de sortie. À quoi sert l'option `-R` ?
3. Même chose après vous être enlevé la permission de lire le répertoire `a/b`
4. Lancer la commande `mkdir a/z` et voir son statut de sortie.
5. Lancer la commande `mkdir a/z` une deuxième fois et voir son statut de sortie.

6. Enlever la permission `w` sur `a/c` (pour l'utilisateur propriétaire), lancer la commande `mkdir a/c/g` et voir son statut de sortie.

--- * ---

Correction de l'exercice 4 : Le statut de sortie de `ls` est :

— 0 si tout s'est bien passé ;

— 2 si impossible d'accéder à un des arguments (par exemple un fichier qui n'existe pas) ;

— 1 si impossible d'accéder à un sous-répertoire d'un des arguments alors que l'option `-R` est présente.

--- * ---

Exercice 5 : Statut de sortie de `grep`

1. Lancer la commande `grep FILE /usr/include/stdio.h` puis voir son statut de sortie.

2. Même chose pour la commande `grep bidule /usr/include/stdio.h`.

3. Même chose pour la commande `grep bidule /hop/plop`.

4. Quels sont les différents statuts de sortie de `grep` et dans quel contexte sont-ils utilisés ?

--- * ---

Correction de l'exercice 5 : Le statut de sortie de `grep` est

— 0 si tout s'est passé normalement **et** `grep` a trouvé des correspondances ;

— 1 si tout s'est passé normalement **et** `grep` n'a pas trouvé des correspondances ;

— 2 si une erreur est survenue (par exemple un fichier qui n'existe pas).

--- * ---

3 `if`

Exercice 6 : Une histoire d'âge

1. Recopier le script suivant dans un fichier nommé `age` :

```
#!/bin/sh

printf 'Bonjour, entrer votre âge : '
IFS= read -r age

if [ $age -gt 30 ]; then
        printf 'Vous êtes trop vieux ! Vous sucrez les fraises !\n'
elif [ $age -lt 3 ]; then
        printf 'Menteur !\n'
else
        printf 'Vous êtes trop jeune ! Vous ne connaissez rien à la vie !\n'
fi

printf "Au revoir !\n"
```

Le rendre exécutable. Rappel : la commande

```
IFS= read -r age
```

met la prochaine ligne de l'entrée standard dans la variable `age`. Les subtilités désagréables sont :

— On vide l'`IFS` pour cette commande (et juste celle-ci) de façon à ce que `read` ne fasse pas de séparation en champs et mette bien toute la ligne dans `age`.
— L'option `-r` permet de s'assurer que les contre-obliques (\) ne sont pas interprétées spécialement.

2. Tenter de prévoir ce qu'affiche la commande `./age` lorsque, lors de son exécution, l'utilisateur entre la valeur :

a) 35
b) 2
c) 18

Vérifier par l'expérience.

3. Modifier le script de façon à ce que, si l'utilisateur entre votre âge, le script réponde

`Ah, c'est le plus bel âge de la vie !`

et ce, peu importe dans quel intervalle il se trouve. Vous aurez sans doute à utiliser `help test`.

4. Modifier votre script de façon à ce que, si l'utilisateur entre un nombre supérieur à 130, le script réponde

`Je pense qu'en fait vous êtes mort.`

--- * ---

Correction de l'exercice 6 : Noter que la première condition parmi les `if` et `elif` à être satsifaite provoquera l'exécution du bloc correspondant et seulement de celui-ci.

```
#!/bin/sh

printf 'Bonjour, entrer votre âge : '
IFS= read -r age
mon_age=39
if [ $age -eq $mon_age ]; then
        printf "Ah, c'est le plus bel âge de la vie !\n"
elif [ $age -gt 130 ]; then
        printf "Je pense qu'en fait vous êtes mort.\n"
elif [ $age -ge 30 ]; then
        printf 'Vous êtes trop vieux ! Vous sucrez les fraises !\n'
elif [ $age -lt 3 ]; then
        printf 'Menteur !\n'
else
        printf 'Vous êtes trop jeune ! Vous ne connaissez rien à la vie !\n'
fi

printf "Au revoir !\n"
```

--- * ---

Exercice 7 : Le script `perms`

Nous allons utiliser la construction `if` et la commande `test` pour reproduire des fonctionnalités de la commande `ls` avec l'option `-l`.

Voici une version 0 de ce script :

```
#!/bin/sh

for arg; do
        if ! [ -e "$arg" ]; then
```

```
                printf '!!!! %s\n' "$arg"
                continue
        fi
        if [ -d "$arg" ]; then
                printf 'd'
        else
                printf '-'
        fi
        printf ' %s\n' "$arg"
done
```

Rappel : le mot clé `!` (à voir comme une *négation*) avant une commande change un statut de sortie nul en 1 et un statut de sortie non nul en 0.

1. Recopier ce script dans un fichier nommé `perms`. Le rendre exécutable et l'essayer, par exemple avec la commande

```
$ ./perms perms /usr /bidule
```

2. Modifier le script de façon à ce qu'il affiche, pour chacun de ses arguments, interprété comme un nom de chemin, sur une ligne :

1. la chaîne `!!!!` si le nom de chemin ne correspond pas à un fichier existant ;
2. une autre chaîne de 4 caractères sinon, où :
   a) le premier caractère est `d` si le fichier est un répertoire et `-` si c'est un fichier normal ;
   b) le deuxième caractère est `r` si l'utilisateur qui exécute le script a la permission `r` sur ce fichier, `-` sinon ;
   c) le troisième caractère est `w` si l'utilisateur qui exécute le script a la permission `w` sur ce fichier, `-` sinon ;
   d) le quatrième caractère est `x` si l'utilisateur qui exécute le script a la permission `x` sur ce fichier, `-` sinon ;
   e) Enfin, un espace et le chemin.

   Vous aurez sûrement à chercher de l'aide avec la commande `help test`.

3. Pour finir, faire en sorte que, lancé sans argument, le script `perms` affiche les permissions des fichiers non cachés du répertoire courant (bonus : gérer le cas où le répertoire courant ne contient aucun fichier non caché).

Voici un exemple, précédé de la commande `ls -l` (pour voir les fichiers présents et leurs permissions) :

```
$ ls -l
---x------. 2 pierre pierre   4096 22 avril 09:01 a
-rwx------. 2 pierre pierre   4096 22 avril 09:01 b
dr-x------. 2 pierre pierre   4096 22 avril 09:01 trucs
$ ./perms a b trucs machins /usr/bin/
---x a
-rwx b
dr-x trucs
!!!! machins
dr-x /usr/bin
$ ./perms
---x a
-rwx b
dr-x trucs
```

--- * ---

Correction de l'exercice 7 :

```
#!/bin/sh

case $# in
0)
        set -- *
        # cas où le répertoire courant ne contient aucun fichier non caché
        # alors * se développe en "*", voir si "$1" existe
        if ! [ -e "$1" ]; then
                exit 0
        fi
esac

for chemin; do
        if ! [ -e "$chemin" ]; then
                printf "!!!! %s\n" "$chemin"
                continue
        fi

        if [ -d "$chemin" ]; then
                printf "d"
        else
                printf "-"
        fi

        if [ -r "$chemin" ]; then
                printf "r"
        else
                printf "-"
        fi

        if [ -w "$chemin" ]; then
                printf "w"
        else
                printf "-"
        fi

        if [ -x "$chemin" ]; then
                printf "x"
        else
                printf "-"
        fi
        printf ' %s\n' "$chemin"
done
```

Remarque, nous considérons (comme dans le reste du cours) qu'il n'existe que deux types de fichier, les répertoires et les fichiers normaux. Il y a cependant d'autres types de fichiers (sockets (s), tubes nommés (p), liens symboliques (l), périphériques de type bloc (b) et caractère (c)).

--- * ---

4 `while`

Exercice 8 :

Le script suivant appelé `facto` sert à calculer la factorielle de son unique argument :

```
#!/bin/sh
N=$1
res=1
while [ "$N" -gt 1 ]; do
        res=$(($res * $N))
        N=$(($N - 1))
done
printf '%d!=%d\n' "$1" "$res"
```

1. Le recopier, le rendre exécutable et l'essayer, par exemple avec
   ```
   $ ./facto 5
   $ ./facto 10
   $ ./facto 20
   ```
2. En faire la trace, à la main, lorsque l'argument est `4`. Vérifier en ajoutant la commande `set -x` au début du script puis en l'exécutant.
3. En s'inspirant de `facto`, écrire `puissance` qui prend deux arguments numériques n et k et écrit sur sa sortie la valeur de n^k.

--- * ---

Correction de l'exercice 8 :

```
#!/bin/sh

n=$1
k=$2
res=1

while [ "$k" -ge 1 ]; do
        res=$(($res * $n))
        k=$(($k - 1))
done
printf '%d^%d=%d\n' "$1" "$2" "$res"
```

--- * ---

Exercice 9 : L'idiome `while read`

Pour lire l'entrée standard ligne par ligne, on utilise la commande `read` (qui lit une ligne d'entrée et la met dans une variable) dans la partie *condition* d'une boucle `while` : ainsi, s'il n'y a plus de ligne à lire, `read` a un statut de sortie non nul et on sort de la boucle. Voici un exemple :

```
#!/bin/sh
i=0
while IFS= read -r ligne; do
        i=$(($i + 1))
        printf '%6d %s\n' "$i" "$ligne"
done
```

1. Recopier ce script dans un fichier nommé `mon_nl`, le rendre exécutable et le tester, par exemple de la façon suivante :
   ```
   $ ./mon_nl <mon_nl
   ```
   Puis sans redirection :

```
$ ./mon_nl
```

en écrivant quelques lignes d'entrée sur le terminal puis en appuyant sur `Ctrl+D` (fin de fichier) pour mettre fin à la saisie.

2. En s'inspirant de `mon_nl`, écrire le script `longues_lignes` qui n'imprime sur sa sortie que les lignes de son entrée qui sont de longueur supérieure à `80` (rappel : longueur de la chaîne contenue dans la variable `var` : `${#var}`).
3. Bonus : à l'aide notamment d'une boucle `for` et de redirections, compter le nombre de longues lignes (longueur supérieure à 80) dans les fichiers `.h` du répertoire `/usr/include`. Comparer avec le nombre total de lignes dans ces fichiers et dire si le principe de ne pas avoir de longue ligne dans son code est globalement bien respecté.

--- * ---

Correction de l'exercice 9 :

1.
2.
```
#!/bin/sh
while IFS= read -r ligne; do
        if [ ${#ligne} -gt 80 ]; then
                printf '%s\n' "$ligne"
        fi
done
```
3. Nombre total de longues lignes :
```
$ for f in /usr/include/*.h; do ./longues_lignes <"$f"; done | wc -l
```
Nombre total de lignes :
```
$ wc -l /usr/include/*.h | tail -n 1
```
L'ordre de grandeur est d'environ 5 longue ligne pour 100 lignes.
Remarque : `awk` c'est bien :
```
$ awk 'length($0) > 80' /usr/include/*.h | wc -l
```

--- * ---

Exercice 10 : `boum` version 2

1. Écrire le script `boum`, qui prend un argument, interprété comme un nombre positif et qui fait un compte à rebours en partant de ce nombre, en attendant une seconde entre chaque affichage (`sleep 1`) puis, après être arrivé à 1 affiche la chaîne `BOUM !` sur le terminal.
Exemple :
```
$ ./boum 3
3
2
1
BOUM !
```
2. À l'aide de constructions `if` et de la commande `test`, modifier ce script de façon à ce que :
   a) sans aucun argument, le nombre de secondes à attendre est 5 par défaut ;
   b) avec deux arguments ou plus, le script s'interrompt avec un message d'erreur (et un code de retour approprié)
   c) avec un seul argument négatif, le script s'interrompt avec un message d'erreur (et un code de retour approprié).
3. Discuter de la pertinence de l'utilisation d'une (ou plusieurs) construction `case` à la place de la ou des constructions `if`. Y a-t-il encore des cas d'erreur à traiter ?

--- * ---

Correction de l'exercice 10 :

```
#!/bin/sh

if [ $# -gt 1 ]
then
        printf "Vous devez fournir 0 ou 1 argument.\n"
        exit 1
fi

if [ $# -eq 0 ]
then
        N=3
else
        if [ $1 -lt 0 ]
        then
                printf "L'argument doit être positif ou nul !\n"
                exit 2
        fi
        N=$1
fi

i=$N
while [ $i -gt 0 ]
do
        printf "$i\n"
        sleep 1
        i=$(( $i - 1 ))
done
printf "BOUM !\n"
```

L'utilisation d'une construction `case` portant sur `$#` est sans doute plus claire et plus concise. En général, dans les scripts shell, on préfère `case` à `if`.

Enfin, on pourrait tester l'argument donné (avec une construction `case` par exemple) pour vérifier qu'il est numérique.

--- * ---

Exercice 11 : Nombre secret et nombre secret inverse

1. Écrire un script `nb_secret` qui fait deviner un nombre secret compris entre 0 et 1000 à l'utilisateur. Pour obtenir un tel nombre aléatoire, vous pourrez utiliser la commande

   ```
   awk 'BEGIN { srand(); print int(rand()*1001) }'
   ```

   comme par exemple dans

   ```
   $ awk 'BEGIN { srand(); print int(rand()*1001) }'
   883
   $ awk 'BEGIN { srand(); print int(rand()*1001) }'
   536
   ```

   ou bien, si votre shell le permet (`zsh`, `bash`, ...) la variable `RANDOM` et un développement arithmétique :

   ```
   $ echo $RANDOM
   15969
   ```

```
$ echo $RANDOM
29050
$ echo $(($RANDOM % 1001))
90
```

Puis, si l'utilisateur :

- donne un nombre trop petit, le script affiche la chaîne `>` (pour dire « c'est plus grand ! » puis redemande le nombre silencieusement ;
- donne un nombre trop grand, le script affiche la chaîne `<` puis redemande le nombre ;
- donne le nombre secret, le script affiche `=` puis se termine après avoir affiché le nombre d'itérations nécessaires à l'utilisateur pour deviner le nombre secret.

Voici un exemple de déroulement :

```
$ ./nb_secret
500
<
250
<
125
<
50
<
25
>
37
<
31
>
14
>
34
<
32
>
33
=
nb_secret : Vous avez trouvé 33 en 11 coups
```

2. Une fois que votre script fonctionne, jouez avec et essayez de trouver une méthode pour trouver rapidement.

3. On inverse les rôles ! Maintenant c'est *vous* qui pensez à un nombre entre 0 et 1000 et c'est le script `nb_secret_inverse` qui va chercher à deviner le nombre secret. À chaque itération, il vous propose un nombre et vous lui répondez `<` si le nombre secret est plus petit, ou bien `>` s'il est plus grand et enfin `=` si c'est le nombre secret. Alors, le script s'arrête et dit en combien de coups il a trouvé. Bien sûr c'est mieux si le script a une méthode rapide pour trouver. Voici un exemple de déroulement :

```
$ ./nb_secret_inverse # je pense à 764
500
>
750
>
875
<
812
```

```
<
781
<
765
<
757
>
761
>
763
>
764
=
nb_secret_inverse : J'ai trouvé le nombre secret 764 en 10 coups
```

4. Si vous avez fait les choses correctement (en particulier, avez été rigoureux sur les entrées/sorties) vous pouvez faire jouer les deux programmes ensemble ! On utilise pour cela des tubes anonymes et la commande `tee` ainsi qu'un tube nommé (hors-programme de ce cours, mais vous pouvez poser des questions dessus ce n'est pas vraiment compliqué) :

   ```
   $ mkfifo mon_tube
   $ ./nb_secret <mon_tube | tee /dev/tty | ./nb_secret_inverse | tee mon_tube
   ```

   Quand vous aurez fini de vous amuser, vous pourrez :

   ```
   $ rm mon_tube
   ```

--- * ---

Correction de l'exercice 11 : Programme `nb_secret` :

```
#!/bin/sh

nb_sec=$(($RANDOM % 1001))
i=1
while IFS= read -r ligne; do
        if [ "$ligne" -eq "$nb_sec" ]; then
                printf "=\n"
                break;
        elif [ "$ligne" -lt "$nb_sec" ]; then
                printf ">\n"
        else
                printf "<\n"
        fi
        i=$(($i + 1))
done
printf 'nb_secret : Vous avez trouvé %s en %d coups\n' $ligne $i
```

Programme `nb_secret_inverse` :

```
#!/bin/sh

bas=0
haut=1000
i=1
while true; do
        mil=$((($bas + $haut)/2))
        printf '%d\n' $mil
```

```
        IFS= read -r reponse
        if [ "$reponse" = "=" ]; then
                break
        elif [ "$reponse" = "<" ]; then # on cherche un nombre plus petit
                haut=$mil
        else # on cherche un nombre plus grand
                bas=$mil
        fi
        i=$(($i + 1))
done

printf 'nb_secret_inverse : J'\''ai trouvé le nombre secret %d en %d coups\n' "$mil" "$i"
```

--- * ---

Exercice 12 : La ligne la plus longue

Pour lire un fichier ligne par ligne en shell, on peut utiliser la construction suivante :

```
while IFS= read -r ligne; do
    # faire d'autres choses, notamment avec ligne
done <mon_fichier
```

Le fichier `mon_fichier` est alors ouvert en lecture, une seule fois pour toute la boucle, et l'entrée standard de toutes les commandes de la construction (y compris `read`) est redirigée vers ce fichier, plutôt que vers l'entrée du terminal.
Ainsi, toutes les lignes du fichier seront lues les unes après les autres. À la fin du fichier, `read` échouera et la boucle sera terminée.

Écrire un script qui cherche la ligne la plus longue (ou l'une d'entre elles s'il y en a plusieurs) dans un fichier. Affiche à la fin cette ligne, son numéro de ligne et sa taille.

Rappel : pour obtenir la longueur de la chaîne contenue dans une variable ***var****, il suffit d'utiliser le développement* `${#var}`.

--- * ---

Correction de l'exercice 12 :

```
#!/bin/sh

pluslongue=""
long_max=0
num=0

while IFS= read -r ligne; do
        i=$(($i + 1))
        if [ ${#ligne} -gt $long_max ]; then
                long_max=${#ligne}
                pluslongue=$ligne
                num=$i
        fi
done <"$1"
printf 'Ligne la plus longue :\n%s\n' "$pluslongue"
printf 'de taille %d, numéro de ligne %d\n' $long_max $num
```

--- * ---