Programmation C et Système
Environnement d'exécution des processus (depuis le shell)

Régis Barbanchon

L2 Informatique

Lancement d'une commande : chemin explicite vs PATH

Lorsqu'on lance une commande depuis le shell d'un terminal...

- on peut expliciter le chemin d'accès vers son binaire :

```
$ /bin/date
lundi 5 novembre 2018, 14:44:48 (UTC+0100)
```

- ou on peut utiliser le nom du binaire sans chemin explicite :

```
$ date
lundi 5 novembre 2018, 14:44:48 (UTC+0100)
```

Si ce n'est pas le nom d'une *builtin command* interne au shell, le chemin de la commande est recherché dans la variable `PATH`, qui contient une liste de chemins séparés par des `':'`.

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

Si un binaire de ce nom est dans plusieurs répertoires, alors le chemin le plus à gauche est prioritaire.
Par exemple, `/usr/local/bin/date` *préféré à* `/bin/date` *si 2 binaires* `date` *se trouvaient en* `/usr/local/bin` *et* `/bin`.

Arguments d'une commande : options et usage

Une commande peut avoir des arguments derrière son nom.
La commande `whereis -b <name>` permet de localiser un binaire.
Par exemple, on voit que `date` ne se trouve que dans `/bin` par :

```
$ whereis -b date
date: /bin/date
```

C'est l'option `-b` qui limite la recherche aux binaires. Sans elle, d'autres fichiers sont recherchés (ici, les pages du manuel) :

```
$ whereis date
date: /bin/date /usr/share/man/man1/date.1.gz
```

Lorsque la forme des arguments n'est pas reconnue,
une commande affiche en général une aide abrégée de son usage.
Par exemple, `whereis` utilisé sans argument produit la sortie :

```
$ whereis
Usage:
   whereis [options] <name>

Locate the binary, source, and manual-page files for a command.
Options:
   -b          search only for binaries
   -m          search only for manuals and infos
   -s          search only for sources
   ...
```

Documentation d'une commande : les man pages

La commande `man` affiche une page du manuel d'utilisation.
Le manuel est divisé en sections, les 3 principales étant :

- Section 1 : les commandes exécutables depuis le shell.
- Section 2 et 3 : les fonctions C, resp. système et utilisateur.

Par exemple :

- `man 1 printf` affiche le manuel de la commande `printf`.
- `man 3 printf` affiche le manuel de la fonction C `printf()`.
- `man printf` choisit le manuel de plus petite section (ici `1`).
- Pour plus d'info, voir `man man` qui affiche le manuel de `man`.

```
$ man 1 whereis
```

```
NAME
       whereis  -  locate the binary, source, and manual page for a command.
SYNOPSIS
       whereis [options] name...
DESCRIPTION
       whereis locates the binary, source and manual files for  the  specified
       command names. The search restrictions (options -b, -m, -s) are cumula-
       tive and apply to the subsequent name patterns on the command line.
OPTIONS
       -b     Search for binaries.
       -m     Search for manuals.
       -s     Search for sources.
```

Environnement d'une commande

Parmi les variables du shell, certaines peuvent être exportées vers une commande afin que celle-ci adapte son comportement.
On les appelle alors **variables d'environnement**.
Elles sont exportées par `declare -x` ou `export`.
Par exemple, `LC_TIME` spécifie la langue et l'encodage de la date :

```
$ export LC_TIME=en_US.UTF-8
$ echo $LC_TIME
en_US.UTF-8
$ date
Mon Nov  5 14:44:48 CET 2018
```

```
$ LC_TIME=fr_FR.UTF-8    # export useless here, LC_TIME already exported above
$ echo $LC_TIME
fr_FR.UTF-8
$ date
lundi 5 novembre 2018, 14:44:48 (UTC+0100)
```

On peut aussi transmettre une variable à la commande sans modifier celle du shell, en préfixant la commande par l'assignement de la variable avec la valeur à transmettre :

```
$ LC_TIME=en_US.UTF-8  date                    # variable=value command
Mon Nov  5 14:44:48 CET 2018
$ echo $LC_TIME                                # value unchanged in shell
fr_FR.UTF-8
```

Commandes en avant-plan et en arrière plan

Le shell peut exécuter :

- plusieurs commandes en **arrière-plan** (en les suffixant avec &),
- une unique commande en **avant-plan** (pas de suffixe).

Lorsque la commande est en **avant-plan** :

- le shell attend sa terminaison pour réafficher son prompt,
- elle possède le terminal pour la saisie au clavier,
- elle possède le terminal pour l'affichage.

```
$ date
lundi 5 novembre 2018, 14:44:48 (UTC+0100)
$                           # prompt displayed after date is terminated
```

Lorsque la commande (typiquement fenêtrée) est en **arrière-plan** :

- le shell réaffiche son prompt immédiatement,
- elle ne peut saisir pas au clavier,
- par défaut, elle peut afficher sur le terminal.

```
$ firefox &
[1] 7773                    # job 1, process 7773
$ thunderbird &             # prompt displayed while firefox still running
[2] 7774                    # job 2, process 7774
$                           # prompt displayed while thunderbird still running
```

Terminaison normale d'une commande

La terminaison d'une commande est dite **normale**
lorsque c'est elle qui décide volontairement de sa terminaison,
(par opposition à une interruption par un signal).

Le programme communique alors son **exit status**.
Il s'agit d'un code entier n compris entre 0 et 255 :

- $n = 0$ pour une terminaison sans erreur ;
- $1 \leq n \leq 255$, au choix du programmeur, pour une erreur.

La variable spéciale ? du shell contient toujours l'**exit status**
de la dernière commande exécutée en **avant-plan** :

```
$ date -d 12/25/2018        # date format is month/day/year
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
$ echo $?
0
$ date -d 25/12/2018        # date format should be month/day/year
date: invalid date '25/12/2018'
$ echo $?
1
$ echo $?  # last foreground command was echo
0
```

Terminaison anormale d'une commande

La terminaison d'une commande est dite **anormale**
lorsque celle-ci est interrompue par un envoi de signal,
(par opposition à une décision volontaire de la commande).

Un programme interrompu par un signal n'a pas de **exit status**.
Cependant, sous les shells Bourne (`sh`) et Bourne Again (`bash`),
après terminaison anomale d'un processus en **avant-plan**,
la variable spéciale `?` aura la valeur `128+s`, où `s` est le n° du signal.

Pour cette raison, il est déconseillé de terminer normalement
avec un exit status `n > 128`. Car dans ce cas, sous `sh` et `bash`,
on ne sait pas distinguer si la commande en avant-plan...

- a bien terminé normalement avec un exit status `n = $?`,
- ou si elle a été interrompue par un signal `s = 128-$?`.

Liste des principaux signaux

La table suivante est extraite de la commande `man 7 signal` :

```
Signal  Value  Action      Comment
------ ------ ----------- ------------------------------------------
 HUP       1   Terminate   Hangup detected on controlling terminal
                           (or death of controlling process)
 INT       2   Terminate   Interrupt from keyboard by CTRL-C

 QUIT      3   Dump Core   Quit from keyboard by CTRL-\
 ILL       4   Dump Core   Illegal Instruction
 TRAP      5   Dump Core   Trace/breakpoint trap
 ABRT      6   Dump Core   Abort signal from abort()
 FPE       8   Dump Core   Floating point exception
 SEGV     11   Dump Core   Invalid memory reference

 KILL      9   Terminate   Kill signal
 USR1     10   Terminate   User-defined signal 1
 USR2     12   Terminate   User-defined signal 2
 PIPE     13   Terminate   Broken pipe: write to pipe with no readers
 ALRM     14   Terminate   Timer signal from alarm()
 TERM     15   Terminate   Termination signal
 STKFLT   16   Terminate   Unused, stack fault

 CHLD     17   Ignore      Child stopped or terminated

 CONT     18   Continue    Continue if stopped

 STOP     19   Stop        Stop process
 TSTP     20   Stop        Stop typed at terminal by CTRL-Z
 TTIN     21   Stop        Terminal  input for background process
 TTOU     22   Stop        Terminal output for background process
```

Interruption au clavier de la commande en avant-plan

On voit que la commande en avant-plan peut être interrompue par l'utilisateur via une combinaison de touches au clavier :

- CTRL-C (noté ^C) : termine par envoi du signal 2 (INT),

```
$ sleep 1               # sleeps for 1 second in foreground
$ echo $?
0
$ sleep 100             # sleeps for 100 seconds in foreground
^C
$ echo $?               # CTRL-C sends signal 2 (INT) and 130=128+2
130
```

- CTRL-\ (noté ^\) : termine par envoi du signal 3 (QUIT),

```
$ sleep 100             # sleeps for 100 seconds in foreground
^\Quit (core dumped)
$ echo $?                # CTRL-\ sends signal 3 (QUIT) and 131=128+3
131
```

- CTRL-Z (noté ^Z) : stoppe par envoi du signal 20 (TSTP).

Une commande stoppée n'est pas terminée. Elle sera continuée :

- en avant-plan avec la commande fg (foreground job).
- en arrière-plan avec la commande bg (background job).

Continuation en avant-plan avec `fg`

Le shell maintient les commandes en cours dans une table de jobs.

```
$ sleep 100 & sleep 500 & sleep 200 &
[1] 21332
[2] 21333
[3] 21334
```

Les numéros entre crochets sont les ids de job dans la table.
(Ils sont suivis des ids des processus en cours d'exécution.)
La commande `jobs` affiche l'états des jobs du shells :

```
$ jobs
[1]   Running                 sleep 100 &
[2]-  Running                 sleep 500 &
[3]+  Running                 sleep 200 &
```

La commande `fg %jobid` ramène un job en avant-plan :

```
$ fg %3
sleep 200
```

On peut stopper la commande en avant-plan avec `CTRL+Z` :

```
^Z
[3]+  Stopped                 sleep 200
```

La commande n'est pas terminée, elle *"dort"* jusqu'à son réveil.

Continuation en arrière-plan avec bg

La table des jobs est alors dans l'état suivant :

```
$ jobs
[1]   Running                 sleep 100 &
[2]-  Running                 sleep 500 &
[3]+  Stopped                 sleep 200
```

On peut la réveiller, en arrière-plan ou en avant-plan.
Par exemple en arrière-plan, avec `bg` ou `bg %jobid` :

```
$ bg %3
[3]+ sleep 200 &
```

La table des jobs repasse alors dans l'état initial :

```
$ jobs
[1]   Running                 sleep 100 &
[2]-  Running                 sleep 500 &
[3]+  Running                 sleep 200 &
```

Terminons les au clavier via `CTRL-C` après mise en avant-plan :

```
$ fg
sleep 200
^C
```

```
$ fg
sleep 500
^C
```

```
$ fg
sleep 100
^C
```

La table des jobs est maintenant vide :

```
$ jobs
$
```

Interruption par envoi de signal via la commande `kill`

La commande `kill -s signum pid` ou `kill -signum pid` envoie le signal identifié par `signum` au processus identifié par `pid`. `signum` peut être un symbole (par exemple `INT`, `QUIT` ou `KILL`) ou bien l'entier correspondant (par exemple `2`, `3` ou `9`).

```
$ sleep 100 & sleep 500 & sleep 200 &
[1] 21913
[2] 21914
[3] 21915
```

Tuons d'abord le premier job avec le signal `2` (`INT`) :

```
$ kill -2 21913   # or: kill -INT 21913
$
[1]   Interrupt                 sleep 100
```

Tuons ensuite le deuxième job avec le signal `3` (`QUIT`) :

```
$ kill -3 21914   # or: kill -QUIT 21914
$
[2]-  Quit                      (core dumped) sleep 500
```

Tuons enfin le troisième job avec le signal `9` (`KILL`) :

```
$ kill -9 21915   # or: kill -KILL 21916
$
[3]+  Killed                    sleep 200
$
```

Sortie standard et sortie d'erreur d'un processus

Pour un processus lancé depuis un terminal,
il y a deux façons principales d'écrire sur ce terminal :

- écrire sur la sortie standard (`stdout` pour *standard output*),
- écrire sur la sortie d'erreur (`stderr` pour *standard error*).

Dans l'exemple ci-dessous, `date` écrit son résultat sur `stdout`...

```
$ date -d 12/25/2018        # date format is month/day/year
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
```

alors que ci-dessous, `date` écrit son erreur sur `stderr`...

```
$ date -d 25/12/2018        # date format should be month/day/year
date: invalid date '25/12/2018'
```

et dans les 2 cas, les sorties s'affichent par défaut sur le terminal.
On peut cependant les **rediriger** vers des destinations différentes :

- vers un fichier,
- vers un *"trou noir"* (pour ignorer la sortie),
- vers un autre terminal,
- vers l'entrée d'un autre processus, etc.

Redirection de `stdout` et `stderr` vers un fichier

Soit `dates.txt` un fichier contenant 4 dates dont 2 invalides :

```
$ ls
dates.txt
$ cat ./dates.txt
12/25/2018
25/12/2018
31/01/2019
01/31/2019
```

Voici la sortie de `date -f` pour ce fichier sur le terminal :

```
$ date -f ./dates.txt
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
date: invalid date '25/12/2018'
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
date: invalid date '31/01/2019'
```

Redirigeons `stdout` avec `1>` et `stderr` avec `2>`, vers 2 fichiers :

```
$ date -f ./dates.txt   1> ./results.log   2> ./errors.log
$ ls
dates.txt   errors.log   results.log
```

Examinons les 2 fichiers `./results.log` et `./errors.log` :

```
$ cat ./results.log
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
$ cat ./errors.log
date: invalid date '25/12/2018'
date: invalid date '31/01/2019'
```

Redirection vers un fichier en ajout

Les opérateurs de redirection `[n]>` fonctionnent en écrasement :

- `command 1> file` redirige `stdout` vers `file` en écrasement,
- `command 2> file` redirige `stderr` vers `file` en écrasement.

C'est-à-dire que si `file` existait déjà, son contenu initial est perdu.

Si l'on veut conserver son contenu initial, il faut utiliser `[n]>>` :

- `command 1>> file` redirige `stdout` vers `file` en ajout,
- `command 2>> file` redirige `stderr` vers `file` en ajout.

```
$ rm -f results.log errors.log
$ date -d 12/25/2018   1>> results.log   2>> errors.log
$ date -d 25/12/2018   1>> results.log   2>> errors.log
$ date -d 31/01/2019   1>> results.log   2>> errors.log
$ date -d 01/31/2019   1>> results.log   2>> errors.log
```

À nouveau, si l'on examine `results.log` et `errors.log` :

```
$ cat ./results.log
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
$ cat ./errors.log
date: invalid date '25/12/2018'
date: invalid date '31/01/2019'
```

Redirection des sorties vers le *"trou noir"* /dev/null

Le pseudo-fichier /dev/null ignore tout ce qu'on y écrit.
Reprenons dates.txt, fichier contenant 4 dates dont 2 invalides :

```
$ ls
dates.txt
$ cat ./dates.txt
12/25/2018
25/12/2018
31/01/2019
01/31/2019
```

Si l'on veut ignorer stderr, on peut la rediriger vers /dev/null :

```
$ date -f ./dates.txt   2> /dev/null
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

Et de même si l'on veut ignorer stdout :

```
$ date -f ./dates.txt   1> /dev/null
date: invalid date '25/12/2018'
date: invalid date '31/01/2019'
```

Rien n'est jamais écrit dans /dev/null :

```
$ cat /dev/null
$
```

Redirection d'une sortie vers un autre terminal

À chaque terminal correspond un pseudo-fichier `/dev/pts/[xx]`.
La commande `tty` permet d'afficher son chemin.

Soit deux terminaux ouverts, un à gauche, un à droite :

```
$ tty
/dev/pts/11
```

```
$ tty
/dev/pts/12
$ cat
```

On peut écrire sur le terminal de droite depuis celui de gauche :

```
$ echo hello > /dev/pts/12
$ echo bye   > /dev/pts/12
$
```

```
hello
bye
[CTRL-D]
$
```

Remarque 1 : la commande `cat` à droite n'a pas d'autre utilité que d'éviter que `hello` soit affiché sur le prompt du shell.

Remarque 2 : `CTRL-D` génère une fin de fichier sur `stdin`, ce qui a pour effet de terminer `cat` qui lit (rien) sur `stdin`.
On réobtient ainsi le prompt du terminal de droite.

Redirection des deux sorties vers une même destination

Pour les sorties d'une même commande :

- `command 1>&2` permet de rediriger `stdout` vers `stderr`,
- `command 2>&1` permet de rediriger `stderr` vers `stdout`.

On peut ainsi rediriger `stdout` et `stderr` vers le même fichier :

```
$ cat ./dates.txt
12/25/2018
25/12/2018
31/01/2019
01/31/2019
```

```
$ date -f ./dates.txt   1> ./output.log   2>&1
$ cat ./output.log
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
date: invalid date '25/12/2018'
date: invalid date '31/01/2019'
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

Attention à l'ordre : si on redirige `stderr` vers `stdout` trop tôt, c-à-d avant que `stdout` soit redirigée vers le fichier, on a :

```
$ date -f ./dates.txt   2>&1   1> ./output.log
date: invalid date '25/12/2018'
date: invalid date '31/01/2019'
$ cat ./output.log
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

Groupement de commandes pour partager les sorties

On peut grouper une séquence de plusieurs commandes avec :
`{ command-1 ; command-2 ; ... ; command-n ; }`

Ceci permet de rediriger leurs sorties vers une même destination.

Exemple avec la redirection de `stdout` vers un fichier :

```
$ { date -d 12/25/2018 ; date -d 01/31/2019 ; }   1> ./results.log
$ cat ./results.log
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

Exemple analogue avec la redirection de `stderr` vers un fichier :

```
$ { date -d 25/12/2018 ; date -d 31/01/2018 ; }   2> ./errors.log
$ cat ./errors.log
date: invalid date '25/12/2018'
date: invalid date '31/01/2019'
```

Remarque : attention à ne pas oublier le dernier point-virgule juste avant l'accolade fermante du groupe.

Substitution de commande (par sa sortie standard)

On peut substituer une commande par sa sortie standard via la syntaxe `$(command)` ou anciennement `‘command‘`.
(Il s'agit de backquotes ici, et non d'apostrophes).

Les deux utilisations principales sont. . .

- rediriger la sortie d'une commande dans une variable :

```
VARIABLE=$( command )
```

- utiliser la sortie d'une commande en argument d'une autre :

```
command-1  $( command-2 )
```

```
$ TTY=$( tty )
$ echo "$TTY"
/dev/pts/14
$ DATE=$( date -d 01/31/2019 )
$ echo "$DATE"
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

Entrée standard d'un processus

L'entrée standard (*standard input* ou `stdin`) est un flux de caractères que peut lire un processus.

Lorsque le processus est lancé par une commande depuis un terminal, `stdin` permet la saisie au clavier depuis ce terminal.

Dans l'exemple suivant `date -f -` lit deux dates saisies au clavier.
La notation `-` est ici une convention pour dénoter `stdin` :

```
$ date -f -                                          # - denotes stdin here
01/31/2019                                           # input by user on stdin
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)           # normal output on stdout
31/01/2019                                           # input by user on stdin
date: invalid date '31/01/2019'                      # errr output on stderr
[CTRL-D]                                             # end of file on stdin
$
```

On signale la fin de fichier `EOF` (*end of file*) sur `stdin`
par la combinaison de touches `CTRL-D`.
Le processus se termine et le shell réaffiche le prompt.

Redirection de `stdin` vers un fichier

On redirige l'entrée standard d'une commande vers un fichier par :

- `command < file`

Le processus qui lit `stdin` lit alors le contenu du fichier
au lieu de lire un flux de caractères saisi au clavier.

Soit le fichier de dates `./dates.txt` suivant :

```
$ cat ./dates.txt
12/25/2018
25/12/2018
31/01/2019
01/31/2019
```

La sortie de `date -f -`, si `stdin` est redirigée vers ce fichier :

```
$ date -f -   < ./dates.txt
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
date: invalid date '25/12/2018'
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
date: invalid date '31/01/2019'
```

Pipeline de processus

Un job peut être un pipeline de commandes.
Elles sont connectées par l'opérateur `|` (*pipe*) comme suit :

- `command-1 | command-2 | ... | command-n`

Pour `command-j | command-k`, un *pipe* est créé, c-à-d,
une file de caractères de capacité limitée est allouée en mémoire par le système, avec une entrée et une sortie qui sont accessibles comme des fichiers par `command-j` et `command-k`, puis :

- `command-j` redirige `stdout` sur la sortie du *pipe*,
- `command-k` redirige `stdin` sur l'entrée du *pipe*,

`command-k` lit donc sur `stdin` ce que `command-j` écrit sur `stdout`.

```
$ cat ./dates.txt
12/25/2018
25/12/2018
31/01/2019
01/31/2019
```

```
$ cat ./date.txt   |   date -f -
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
date: invalid date '25/12/2018'
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
date: invalid date '31/01/2019'
```

Tee : écriture sur la sortie standard + un fichier (1/2)

Avec la redirection de `stdout` sur un fichier, par exemple...

```
$ date -d 01/31/2019   1> ./results.log
```

on ne voit plus la sortie s'afficher dans le terminal,
puisque celle-ci est redirigée dans le fichier `results.log` :

```
$ cat ./results.log
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

La commande `tee file` lit `stdin` et la réécrit à la fois :

- sur sa sortie standard `stdout`,
- et sur le fichier `file`, ouvert en écrasement par défaut.

De sorte que le pipeline `command | tee file`
est analogue à la redirection `command 1> file`,
mais affiche aussi sur le terminal la sortie écrite dans `file`.

```
$  date -d 01/31/2019   | tee ./results.log
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
$ cat ./results.log
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

Tee : écriture sur la sortie standard + un fichier (2/2)

Remarque : `tee -a file` ouvre file en mode ajout (*append*), de sorte que le pipeline `command | tee -a file` est analogue à la redirection `command 1>> file`.

À titre d'exemple, comparer la sortie de...

```
$ date -d 12/25/2018   1>  ./results.log
$ date -d 01/31/2019   1>> ./results.log
```

```
$ cat ./results.log
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

avec la sortie de...

```
$ date -d 12/25/2018   | tee    ./results.log
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
$ date -d 01/31/2019   | tee -a ./results.log   # do not owerwrite content
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

```
$ cat ./results.log
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
jeudi 31 janvier 2019, 00:00:00 (UTC+0100)
```

Substitution de processus (par le nom de sa sortie)

La substitution de processus `<( cmd )` crée un fichier temporaire de la forme `/dev/fd/xx`, y redirige la sortie standard de `cmd`, et affiche le nom du fichier temporaire `/dev/fd/xx` :

```
$ echo <( date )          # today's date is written to /dev/fd/63 and then lost
/dev/fd/63
```

Utilisation : Soit les séquences d'entiers suivantes et leurs tris...

```
$ seq 9 +1 11
9
10
11
```

```
$ seq 11 -1 9
11
10
9
```

```
$ seq 11 -1 9 | sort
10
11
9
```

```
$ seq 11 -1 9 | sort -n
9
10
11
```

On teste les tris lexicographiques et numériques via `diff -qs`, commande qui teste si des fichiers sont identiques ou différents :

```
$ seq 9  +1 11          > ints
$ seq 11 -1  9 | sort > lex-sort
$ diff -qs ints lex-sort ; echo $?
Files ints and lex-sort differ
1
```

```
$ seq 9  +1 11             > ints
$ seq 11 -1  9 | sort -n > num-sort
$ diff -qs ints num-sort ; echo $?
Files ints and num-sort are identical
0
```

Avec une substitution de processus, on peut écrire directement :

```
$ diff -qs   <( seq 9 +1 11 )   <( seq 11 -1 9 | sort -n )   ; echo $?
Files /dev/fd/63 and /dev/fd/62 are identical
0
```