Scripts Bash — Partie 1

Comprendre et réaliser les scripts 1.1 à 1.5

Guide de réalisation, d'explication et de compréhension

Partie 1 — Scripts Bash pour lister les sous-dossiers

Je reprends ici les cinq scripts corrigés transmis, sans modifier leur fonctionnement. L'objectif est de comprendre :

- dans quel ordre écrire les éléments dans nano ;
- pourquoi cet ordre est logique ;
- le rôle précis de chaque ligne ;
- ce qui évolue d'une question à la suivante ;
- comment vérifier et tester chaque script.

Méthode générale avant chaque script

Place-toi dans le dossier contenant tes scripts :

```
cd ~/TP_scripts
```

Pour ouvrir ou créer un script :

```
nano script1_1.sh
```

Dans nano :

```
Ctrl + O  → enregistrer
Entrée    → confirmer le nom du fichier
Ctrl + X  → quitter
```

Ensuite, rends le script exécutable :

```
chmod +x script1_1.sh
```

Vérifie sa syntaxe :

```
bash -n script1_1.sh
```

Si aucune réponse ne s'affiche, Bash n'a détecté aucune erreur de syntaxe.

Enfin, exécute-le :

```
./script1_1.sh
```

Ordre logique d'un script Bash

1. Indiquer l'interpréteur avec #!/bin/bash
2. Déclarer les variables de départ
3. Examiner les paramètres reçus
4. Choisir les dossiers à traiter
5. Parcourir ces dossiers avec une boucle
6. Vérifier que chaque dossier existe
7. Exécuter la commande principale

8. Traiter les erreurs
9. Terminer avec un code de sortie

Cet ordre est important : le script doit d'abord savoir avec quelles informations travailler avant de lancer find.

Question 1.1 — Lister les dossiers du répertoire courant

Objectif

Afficher les dossiers présents dans le répertoire courant, sans descendre dans leur contenu.

Création

```
nano script1_1.sh
```

Contenu :

```
#!/bin/bash

find . -maxdepth 1 -type d
```

Ce script correspond exactement à la première version transmise.

Explication ligne par ligne

Première ligne

```
#!/bin/bash
```

Cette ligne s'appelle le shebang.

Elle indique que le fichier doit être interprété par Bash.

Décomposition :

```
#!          → indique qu'un interpréteur va être précisé
/bin/bash   → chemin vers l'interpréteur Bash
```

Le shebang doit toujours être placé sur la première ligne du script.

Ligne principale

```
find . -maxdepth 1 -type d
```

Cette commande recherche des dossiers.

Décomposition :

```
find
```

Lance une recherche.

```
.
```

Le point représente le répertoire courant, c'est-à-dire le dossier dans lequel tu te trouves au moment où tu exécutes le script.

```
-maxdepth 1
```

Limite la profondeur de recherche à un seul niveau.

Cela empêche find de parcourir récursivement tous les sous-dossiers.

```
-type d
```

Demande à find de conserver uniquement les éléments de type dossier.

Le d signifie :

directory → dossier

Exécution

```
chmod +x script1_1.sh
bash -n script1_1.sh
./script1_1.sh
```

Pourquoi le point . apparaît-il dans le résultat ?

La commande peut afficher :

```
.
./Documents
./Images
./Téléchargements
```

Le premier point représente le répertoire courant lui-même.

Le script ne retire pas ce résultat, car la commande corrigée est bien :

```
find . -maxdepth 1 -type d
```

À retenir

find → rechercher

. → répertoire courant

-maxdepth 1 → ne pas descendre plus profondément

-type d → afficher uniquement les dossiers

Question 1.2 — Utiliser éventuellement un dossier donné en paramètre

Objectif

Le script doit maintenant :

- travailler dans le répertoire courant si aucun dossier valide n'est fourni ;
- travailler dans le dossier indiqué si le paramètre correspond à un dossier existant.

Création à partir du script précédent

Pour conserver la version 1.1 :

```
cp script1_1.sh script1_2.sh
nano script1_2.sh
```

Contenu

```
#!/bin/bash

# On définit le dossier par défaut
DOSSIER="."

# Si un paramètre est fourni ET si le paramètre fourni est un dossier qui existe
# Je lance la commande find sur ce dossier
if [ -d "$1" ] ; then
   DOSSIER=$1
fi

echo "Voici la liste des sous-dossiers de ${DOSSIER}"

# on exécute la commande find sur le dossier $DOSSIER
find $DOSSIER -maxdepth 1 -type d
```

Cette version définit d'abord le répertoire courant comme valeur par défaut, puis remplace cette valeur si $1 correspond à un dossier existant.

Ordre logique dans nano

1. Le shebang

```
#!/bin/bash
```

Toujours en première ligne.

2. La valeur par défaut

```
DOSSIER="."
```

Avant de tester un paramètre, le script prévoit déjà un dossier à utiliser.

Ici :

```
DOSSIER → nom de la variable
=       → affectation
"."     → valeur enregistrée
```

Il ne faut pas mettre d'espace autour du signe =.

Correct :

```
DOSSIER="."
```

Incorrect :

```
DOSSIER = "."
```

3. Vérifier le premier paramètre

```
if [ -d "$1" ] ; then
```

Décomposition :

```
if
```

Commence une condition.

```
[ -d "$1" ]
```

Teste si le premier paramètre correspond à un dossier existant.

`-d` → le chemin est-il un dossier ?

`$1` → premier paramètre transmis au script

Exemple :

```
./script1_2.sh /home
```

Dans cette commande :

`$0` → `./script1_2.sh`

`$1` → `/home`

Les guillemets autour de "$1" évitent certains problèmes lorsque la valeur contient des espaces ou lorsqu'elle est vide.

```
then
```

Signifie : si la condition est vraie, exécuter ce qui suit.

4. Remplacer la valeur par défaut

```
DOSSIER=$1
```

Si $1 est un dossier existant, sa valeur remplace le point.

Exemple :

```
Valeur de départ : DOSSIER="."
Paramètre reçu :  /home
Nouvelle valeur : DOSSIER=/home
```

5. Fermer la condition

```
fi
```

fi termine une structure if.

On peut retenir que fi correspond à if écrit à l'envers.

6. Afficher une phrase

```
echo "Voici la liste des sous-dossiers de ${DOSSIER}"
```

echo affiche du texte à l'écran.

```
${DOSSIER}
```

Permet d'utiliser la valeur de la variable.

Les accolades délimitent clairement son nom.

Si DOSSIER=/home, le résultat sera :

```
Voici la liste des sous-dossiers de /home
```

7. Exécuter la recherche

```
find $DOSSIER -maxdepth 1 -type d
```

La commande find travaille maintenant sur la valeur contenue dans DOSSIER.

Tests

Sans paramètre

```
./script1_2.sh
```

Le test suivant est faux :

```
[ -d "$1" ]
```

car $1 est vide.

La variable conserve donc sa valeur initiale :

```
DOSSIER="."
```

Avec un dossier valide

```
./script1_2.sh /home
```

/home est un dossier existant, donc :

```
DOSSIER=/home
```

Avec un chemin invalide

```
./script1_2.sh /dossier/inexistant
```

Le test est faux. Le script conserve alors :

```
DOSSIER="."
```

Cette version ne produit pas encore de message d'erreur. Cette amélioration arrive dans la question 1.3.

Évolution entre 1.1 et 1.2
1.1 → travaille toujours dans le répertoire courant
1.2 → accepte éventuellement un dossier valide en paramètre

Question 1.3 — Distinguer l'absence de paramètre d'un paramètre invalide

Objectif

Le script doit maintenant distinguer trois situations :

```
Aucun paramètre      → utiliser le répertoire courant
Dossier valide       → utiliser ce dossier
Paramètre invalide   → afficher une erreur et arrêter le script
```

Création

```
cp script1_2.sh script1_3.sh
nano script1_3.sh
```

Contenu

```
#!/bin/bash

# Si un paramètre est fourni
if [ -z "$1" ]; then
   # pas de paramètre
   DOSSIER="."
else
   # paramètre présent
   # si le paramètre fourni est un dossier qui existe
   if [ -d "$1" ] ; then
      DOSSIER=$1
   else
      echo "Le dossier $1 n'existe pas"
      exit 1
   fi
fi

echo "Voici la liste des sous-dossiers de ${DOSSIER}"

# on exécute la commande find sur le dossier $DOSSIER
find $DOSSIER -maxdepth 1 -type d
```

La nouveauté principale est le test -z, qui permet de vérifier si le premier paramètre est vide.

Pourquoi commencer par tester -z ?

Le script doit d'abord répondre à cette question :

```
Un premier paramètre a-t-il été fourni ?
```

Il utilise :

```
if [ -z "$1" ]; then
```

-z signifie :

```
La chaîne contient-elle zéro caractère ?
```

Autrement dit :

```
La valeur est-elle vide ?
```

Mémo

-z → zéro caractère → chaîne vide

-n → chaîne non vide

Premier cas — Aucun paramètre

```
if [ -z "$1" ]; then
   DOSSIER="."
```

Si l'utilisateur exécute :

```
./script1_3.sh
```

alors $1 est vide.

Le test est vrai et le script utilise :

```
DOSSIER="."
```

Deuxième cas — Un paramètre est présent

```
else
```

Le else signifie :

```
Sinon, c'est qu'une valeur est présente dans $1.
```

Le script doit maintenant vérifier si cette valeur représente réellement un dossier.

Condition imbriquée

```
if [ -d "$1" ] ; then
   DOSSIER=$1
```

Cette deuxième condition se trouve à l'intérieur de la première.

On parle de condition imbriquée.

La logique complète est :

```
Si $1 est vide :
    utiliser le répertoire courant
Sinon :
    si $1 est un dossier :
        utiliser ce dossier
    sinon :
        afficher une erreur
```

Cas du paramètre invalide

```
else
   echo "Le dossier $1 n'existe pas"
   exit 1
```

Si $1 ne correspond pas à un dossier existant :

```
echo "Le dossier $1 n'existe pas"
```

affiche un message.

Puis :

```
exit 1
```

arrête immédiatement le script.

Convention des codes de sortie :

`exit 0 → succès`

`exit différent de 0 → erreur`

Ici, exit 1 indique qu'une erreur s'est produite.

Comme le script s'arrête, la commande find placée plus bas ne sera pas exécutée.

Pourquoi la commande find est-elle placée après les conditions ?

```
find $DOSSIER -maxdepth 1 -type d
```

À ce moment-là, deux possibilités seulement subsistent :

`DOSSIER contient "."`

`ou`

`DOSSIER contient un dossier valide`

Si le paramètre était invalide, exit 1 aurait déjà arrêté le script.

Cela évite donc de lancer find sur un chemin incorrect.

Tests

Aucun paramètre

```
./script1_3.sh
```

Dossier valide

```
./script1_3.sh /home
```

Dossier inexistant

```
./script1_3.sh /dossier/inexistant
```

Résultat attendu :

```
Le dossier /dossier/inexistant n'existe pas
```

Évolution entre 1.2 et 1.3

1.2 → un mauvais paramètre est silencieusement ignoré

1.3 → un mauvais paramètre provoque un message et exit 1

Question 1.4 — Traiter plusieurs dossiers et enregistrer les erreurs

Objectif

Le script doit maintenant :

- accepter zéro, un ou plusieurs dossiers ;
- traiter chaque dossier séparément ;
- continuer même si l'un des dossiers est invalide ;
- créer un fichier de journalisation des erreurs ;
- compter le nombre d'erreurs ;
- retourner ce nombre à la fin.

Création

```
cp script1_3.sh script1_4.sh
nano script1_4.sh
```

Contenu

```
#!/bin/bash

SORTIE=0
ERRORFILE="errors_$(date '+%Y-%m-%d %H:%M:%S').log"

# Si un paramètre est fourni
if [ $# -eq 0 ]; then
   # pas de paramètre
   DOSSIERS[0]="."
else
   # paramètres présents
   DOSSIERS=$@
fi

for DOSSIER in $DOSSIERS; do

   # si le paramètre fourni est un dossier qui existe
   if [ -d "$DOSSIER" ] ; then
      echo "Voici la liste des sous-dossiers de ${DOSSIER}"

      # on exécute la commande find sur le dossier $DOSSIER
      if ! find $DOSSIER -maxdepth 1 -type d 2>> "$ERRORFILE" ; then
         echo "Erreur d'exécution sur le dossier $DOSSIER"
         let SORTIE=$SORTIE+1
      fi
   else
      # le dossier n'existe pas
      echo "Le dossier $DOSSIER n'existe pas"
      echo "Le dossier $DOSSIER n'existe pas" >> "$ERRORFILE"
      let SORTIE=$SORTIE+1
   fi
done

exit $SORTIE
```

Cette version ajoute une boucle for, un fichier de journalisation et un compteur d'erreurs.

Première partie — Préparer les variables

Compteur d'erreurs

```
SORTIE=0
```

La variable SORTIE commence à zéro.

Elle servira à compter les erreurs.

Exemple :

```
Aucune erreur  → SORTIE=0
Une erreur     → SORTIE=1
Deux erreurs   → SORTIE=2
```

Elle est déclarée en haut car elle doit exister avant le traitement des dossiers.

Nom du fichier d'erreurs

```
ERRORFILE="errors_$(date '+%Y-%m-%d %H:%M:%S').log"
```

Cette variable contient le nom du fichier dans lequel seront enregistrées les erreurs.

Substitution de commande

```
$(date '+%Y-%m-%d %H:%M:%S')
```

Exécute la commande date et récupère son résultat.

Décomposition du format :

```
%Y → année sur quatre chiffres
%m → mois
%d → jour
%H → heure
%M → minutes
%S → secondes
```

Exemple de nom obtenu :

```
errors_2026-07-28 11:35:42.log
```

La date et l'heure permettent de créer un journal différent pour chaque exécution.

Deuxième partie — Examiner le nombre de paramètres

```
if [ $# -eq 0 ]; then
```

$# représente le nombre de paramètres transmis au script.

-eq signifie :

égal à

Donc :

```
[ $# -eq 0 ]
```

signifie :

Le nombre de paramètres est-il égal à zéro ?

Aucun paramètre

```
DOSSIERS[0]="."
```

Le script place le répertoire courant dans la première case de DOSSIERS.

DOSSIERS[0] → première case

"." → répertoire courant

Un ou plusieurs paramètres

```
else
   DOSSIERS=$@
```

$@ représente tous les paramètres reçus.

Exemple :

```
./script1_4.sh /home /etc /var
```

On obtient conceptuellement :

$1 → /home

$2 → /etc

$3 → /var

$@ → /home /etc /var

La variable DOSSIERS reçoit donc la liste des chemins à traiter.

Troisième partie — Parcourir les dossiers

```
for DOSSIER in $DOSSIERS; do
```

La boucle for traite chaque valeur contenue dans DOSSIERS.

Exemple avec :

DOSSIERS=/home /etc /var

La boucle effectue successivement :

1er passage → DOSSIER=/home

2e passage → DOSSIER=/etc

3e passage → DOSSIER=/var

```
do
```

marque le début des instructions répétées.

```
done
```

marque la fin de la boucle.

Quatrième partie — Vérifier chaque dossier

```
if [ -d "$DOSSIER" ] ; then
```

Pour chaque valeur, le script vérifie si elle correspond à un dossier existant.

Dossier valide

```
echo "Voici la liste des sous-dossiers de ${DOSSIER}"
```

Affiche le dossier en cours de traitement.

Puis :

```
if ! find $DOSSIER -maxdepth 1 -type d 2>> "$ERRORFILE" ; then
```

exécute find.

Comprendre le !

```
!
```

Inverse le résultat de la commande.

Une commande réussie retourne normalement 0.

Avec !, la condition devient vraie lorsque find échoue.

La logique est donc :

```
Si la commande find échoue :
    afficher un message
    ajouter 1 au compteur d'erreurs
```

Comprendre 2>>

```
2>> "$ERRORFILE"
```

Les programmes possèdent notamment :

```
1 → sortie standard
2 → sortie d'erreur
```

Le chiffre 2 désigne donc les messages d'erreur.

`>>` → ajouter à la fin du fichier

Ainsi :

```
2>> "$ERRORFILE"
```

signifie :

Ajouter les erreurs produites par find à la fin du fichier de journalisation.

Le contenu existant n'est pas écrasé.

Incrémenter le compteur

```
let SORTIE=$SORTIE+1
```

Ajoute 1 à la variable SORTIE.

Exemple :

Avant → SORTIE=0

Après → SORTIE=1

À chaque erreur, le compteur augmente.

Cinquième partie — Dossier inexistant

```
else
   echo "Le dossier $DOSSIER n'existe pas"
```

Affiche le message à l'écran.

```
echo "Le dossier $DOSSIER n'existe pas" >> "$ERRORFILE"
```

Écrit également le message dans le fichier d'erreurs.

Puis :

```
let SORTIE=$SORTIE+1
```

augmente le compteur.

Le script ne contient pas exit dans la boucle : il continue donc avec le dossier suivant.

Dernière ligne

```
exit $SORTIE
```

Le script se termine avec la valeur du compteur.

Exemple :

Aucune erreur → exit 0

Une erreur → exit 1

Deux erreurs → exit 2

Pour consulter le code de sortie juste après l'exécution :

```
echo $?
```

Tests

Sans paramètre

```
./script1_4.sh
```

Le script travaille sur .

Avec plusieurs dossiers valides

```
./script1_4.sh /home /etc /var
```

Avec un dossier invalide

```
./script1_4.sh /home /inexistant /etc
```

Puis :

```
echo $?
```

Et pour voir le fichier d'erreurs :

```
ls errors_*.log
```

Puis, en utilisant son nom exact :

```
cat "errors_2026-07-28 11:35:42.log"
```

Évolution entre 1.3 et 1.4
1.3 → un seul dossier
1.4 → plusieurs dossiers

1.3 → s'arrête à la première erreur
1.4 → continue avec les autres dossiers

1.3 → pas de journal d'erreurs
1.4 → crée un fichier errors_....log

1.3 → exit 1 en cas d'erreur
1.4 → exit correspond au nombre d'erreurs

Question 1.5 — Accepter un dossier, plusieurs dossiers ou un fichier texte

Objectif

La version finale accepte désormais :

```
Aucun paramètre
→ travailler sur le répertoire courant


Un dossier
→ travailler sur ce dossier


Un fichier texte
→ lire la liste des dossiers contenue dans ce fichier


Plusieurs paramètres
→ traiter tous les paramètres comme des dossiers
```

Création

```
cp script1_4.sh script1_5.sh
nano script1_5.sh
```

Contenu

```
#!/bin/bash

SORTIE=0
ERRORFILE="errors_$(date '+%Y-%m-%d %H:%M:%S').log"

# Si un paramètre est fourni
if [ $# -eq 0 ]; then
   # pas de paramètre
   DOSSIERS[0]="."
elif [ $# -eq 1 ]; then
   # un seul paramètre
   if [ -d $1 ]; then
      # le paramètre est le dossier sur lequel travailler
      DOSSIERS[0]=$1
   elif [ -f $1 ]; then
      # le paramètre est le fichier texte qui contient la liste
      DOSSIERS=`cat $1`
   else
      # le paramètre est invalide
      echo "Paramètre invalide $1"
      exit 1
   fi
else
   # paramètres présents
   DOSSIERS=$@
fi

for DOSSIER in $DOSSIERS; do

   # si le paramètre fourni est un dossier qui existe
   if [ -d "$DOSSIER" ] ; then
      echo "Voici la liste des sous-dossiers de ${DOSSIER}"

      # on exécute la commande find sur le dossier $DOSSIER
      if ! find $DOSSIER -maxdepth 1 -type d 2>> "$ERRORFILE" ; then
         echo "Erreur d'exécution sur le dossier $DOSSIER"
         let SORTIE=$SORTIE+1
      fi
   else
      # le dossier n'existe pas
      echo "Le dossier $DOSSIER n'existe pas"
      echo "Le dossier $DOSSIER n'existe pas" >> "$ERRORFILE"
      let SORTIE=$SORTIE+1
   fi
done

exit $SORTIE
```

Cette version reprend tout le fonctionnement de la 1.4 et ajoute le cas du fichier texte.

Comprendre les trois branches principales

Premier cas — Aucun paramètre

```
if [ $# -eq 0 ]; then
   DOSSIERS[0]="."
```

Si le nombre de paramètres est égal à zéro :

`DOSSIERS contient le répertoire courant.`

Deuxième cas — Exactement un paramètre

```
elif [ $# -eq 1 ]; then
```

elif signifie :

```
Sinon, si...
```

Le script sait qu'il a reçu exactement une valeur. Il doit maintenant déterminer sa nature.

Le paramètre est-il un dossier ?

```
if [ -d $1 ]; then
   DOSSIERS[0]=$1
```

-d vérifie si le paramètre est un dossier.

Exemple :

```
./script1_5.sh /home
```

Alors :

```
DOSSIERS[0]=/home
```

Le paramètre est-il un fichier ?

```
elif [ -f $1 ]; then
```

-f vérifie si le chemin correspond à un fichier classique.

Si c'est un fichier :

```
DOSSIERS=`cat $1`
```

La commande :

```
cat $1
```

lit le contenu du fichier.

Les accents graves :

```
`commande`
```

récupèrent le résultat de la commande.

Ainsi, si chemins.txt contient :

```
/home
/etc
/var
```

alors :

```
DOSSIERS=`cat chemins.txt`
```

place conceptuellement dans DOSSIERS :

```
/home /etc /var
```

La boucle for pourra ensuite traiter chaque chemin.

Le paramètre n'est ni un dossier ni un fichier

```
else
   echo "Paramètre invalide $1"
   exit 1
```

Le script affiche une erreur et s'arrête.

Cette vérification intervient avant la boucle, car le paramètre unique ne peut pas être utilisé correctement.

Troisième cas — Plusieurs paramètres

```
else
   DOSSIERS=$@
```

Si le nombre de paramètres est supérieur à un, tous les paramètres sont considérés comme des dossiers à traiter.

Exemple :

```
./script1_5.sh /home /etc /var
```

Créer le fichier texte de test

```
nano chemins.txt
```

Contenu :

```
/home
/etc
/var
/dossier/inexistant
```

Enregistrer :

```
Ctrl + O
```

```
Entrée
```

```
Ctrl + X
```

Exécuter :

```
./script1_5.sh chemins.txt
```

Le script :

1. reconnaît que chemins.txt est un fichier ;
2. lit son contenu avec cat ;
3. place les chemins dans DOSSIERS ;
4. parcourt les chemins avec for ;
5. vérifie chaque dossier ;
6. lance find sur les dossiers valides ;
7. journalise les dossiers invalides ;
8. compte les erreurs ;
9. termine avec exit $SORTIE.

Tests complets de la version 1.5

Aucun paramètre

```
./script1_5.sh
```

Un dossier

```
./script1_5.sh /home
```

Un fichier texte

```
./script1_5.sh chemins.txt
```

Plusieurs dossiers

```
./script1_5.sh /home /etc /var
```

Un paramètre invalide

```
./script1_5.sh /chemin/invalide
```

Vérifier le code de sortie

À faire immédiatement après l'exécution :

```
echo $?
```

Évolution complète des scripts

1.1

→ Recherche uniquement dans le répertoire courant.

1.2

→ Utilise le répertoire courant par défaut.

→ Accepte un dossier valide en paramètre.

→ Un paramètre invalide est ignoré.

1.3

→ Distingue l’absence de paramètre d’un paramètre invalide.

→ Affiche une erreur.

→ Utilise exit 1.

1.4

→ Accepte plusieurs dossiers.

→ Utilise une boucle for.

→ Continue malgré les erreurs.

→ Crée un fichier de journalisation.

→ Compte les erreurs.

1.5

→ Accepte zéro, un ou plusieurs paramètres.

→ Un paramètre unique peut être un dossier ou un fichier texte.

→ Le fichier texte contient la liste des dossiers à traiter.

Mémo des éléments utilisés

```
$0  → nom du script
$1  → premier paramètre
$#  → nombre de paramètres
$@  → tous les paramètres

-z  → chaîne vide
-d  → chemin correspondant à un dossier
-f  → chemin correspondant à un fichier
-eq → égal à

if    → commencer une condition
then  → instructions si la condition est vraie
elif  → sinon, tester une autre condition
else  → instructions si les conditions précédentes sont fausses
fi    → terminer la condition

for   → parcourir une liste
do    → commencer les instructions de la boucle
done  → terminer la boucle

echo  → afficher du texte
find  → rechercher des fichiers ou des dossiers
cat   → afficher ou lire le contenu d'un fichier
date  → afficher la date et l'heure
let   → effectuer ici une opération arithmétique
exit  → terminer le script

!     → inverser le résultat d'une commande
>>    → ajouter à la fin d'un fichier
2>>   → ajouter la sortie d'erreur à la fin d'un fichier
$?    → code de sortie de la dernière commande
```

Ordre à retenir pour construire la version finale dans nano

1. Écrire #!/bin/bash.
2. Initialiser SORTIE.
3. Construire le nom de ERRORFILE.
4. Tester le nombre de paramètres avec $#.
5. Remplir DOSSIERS selon la situation.
6. Fermer la condition avec fi.
7. Commencer la boucle for.
8. Vérifier chaque dossier avec -d.
9. Afficher le dossier traité.
10. Exécuter find.
11. Rediriger les erreurs avec 2>>.
12. Incrémenter SORTIE en cas d'erreur.
13. Fermer la condition.
14. Fermer la boucle avec done.
15. Terminer avec exit $SORTIE.

Limite à connaître dans les scripts transmis

Les versions 1.4 et 1.5 utilisent notamment :

```
DOSSIERS=$@
```

puis :

```
for DOSSIER in $DOSSIERS
```

Cette écriture sépare les éléments selon les espaces. Elle est donc adaptée aux chemins simples utilisés dans le TP, comme :

```
/home
/etc
/var
```

Un chemin contenant des espaces pourrait être découpé en plusieurs morceaux. Je n’ai pas modifié cette écriture puisqu’elle fait partie des scripts corrigés transmis.