Exercice 2 - Contrôle de mot de passe

💡 Notions abordées

- Instruction conditionnelle `if`
- Lecture d'une valeur au clavier

```
#!/bin/bash

PASSWORD="secret"

read -s -p "Saisissez votre mot de passe : " input

if [ _$input == _$PASSWORD ]; then
        echo -e "\nVous êtes authentifié !"
else
        echo -e "\nErreur: mot de passe incorrect"
fi
```

💡 Auto-test

- Pourquoi ajoute-t-on un caractère (par exemple `_` comme ici) avant les variables dans le `if` ?
- Quelle est l'utilité de l'option `-e` de `echo` ?

Exercice 3 - Expressions rationnelles

💡 Notions abordées

- Passage d'argument à un script
- Fonctions Bash
- Valeur de retour d'une fonction
- Expressions rationnelles

```bash
#!/bin/bash

function is_number()
{
    re='^[+-]?[0-9]+([.][0-9]+)?$'

    if ! [[ $1 =~ $re ]] ; then
        return 1
    else
        return 0
    fi
}

is_number $1

if [ $? -eq 0 ]; then
    echo "Le nombre est un nombre réel !"
else
    echo "Le nombre n'est pas un nombre réel !"
fi
```

💡 Explications

La fonction `is_number` renvoie 1 si le paramètre qui lui est envoyé **n'est pas** un réel (la condition dans le `if` vérifie si le paramètre est conforme à l'expression rationnelle, et on prend la négation de cette condition avec le symbole `!`), et 0 sinon.

La négation n'est bien sûr pas utile (on pourrait l'enlever et inverser les deux `return`), elle n'avait ici qu'un but pédagogique.

💡 Auto-test

- Comprendre ce que représente et comment fonctionne l'expression rationnelle `re`.
- Comprendre la différence entre `echo` et `return` pour renvoyer une valeur dans une fonction.
- A quoi correspond la variable spéciale `$?` ?

Exercice 4 - Contrôle d'utilisateur

💡 Notions abordées

- Appel à une commande dans un script

- Redirections

```
#!/bin/bash

if [ -z $1 ];then
    echo "Utilisation : $0 nom_utilisateur"
else
    id $1 > /dev/null 2>&1

    if [ $? -eq 0 ];then
        echo "L'utilisateur $1 existe"
    else
        echo "L'utilisateur $1 n'existe pas"
    fi
fi
```

💡 Explications

On commence par tester qu'un paramètre a bien été fourni au script (si ce n'est pas le cas, on affiche un message rappelant comment la syntaxe correcte).

Si on a bien un paramètre, on l'utilise pour vérifier s'il correspond à un utilisateur existant. Pour cela on utilise la commande `id` ; comme la plupart des commandes Linux, `id` renvoie `0` si elle se termine avec succès (l'utilisateur a été trouvé) et une valeur positive en cas d'échec.

Afin de ne pas "polluer" l'affichage avec les messages produits par la commande `id`, on redirige ces messages vers le fichier `/dev/null`.

💡 Auto-test

- Que se passe-t-il quand on écrit dans le fichier `/dev/null` ?
- Que signifie `2>&1` ?

Exercice 5 - Factorielle

💡 Notions abordées

- Boucle `for`
- Opérateur de calcul arithmétique

```bash
#!/bin/bash

factorielle=1

for i in `seq 1 $1`;
do
    factorielle=$(($factorielle * $i))
done

echo "Le factoriel du nombre est $factorielle"
```

💡 Auto-test

- Par définition, 0! = 1. Le programme ci-dessus est-il correct ?

Exercice 6 - Juste prix

💡 Notions abordées

- Nombres aléatoires
- Boucles `while` / `until`

```bash
#!/bin/bash

nb=$(( $RANDOM  %  1000 + 1 ));
essais=0
input=0

while [ $nb -ne $input ]
do
    read -p 'Tapez un nombre entre 1 et 1000 : ' input
    ((essais++))

    if [ $nb -gt $input ];then
        echo "C'est plus grand !"
    else
        echo "C'est plus petit !"
    fi
done

echo "Gagné ! Vous avez gagné en $essais essais !"
```

💡 **Explications**

- L'expression `$RANDOM % n` renvoie un nombre aléatoire dans l'intervalle [0; n-1]. Donc on ajoute 1 pour obtenir un nombre dans l'intervalle [1 ; n].
- Le nombre d'essais est incrémenté à chaque tentative ; c'est pourquoi on initialise la variable `essais` à 0.
- Pour être certain de rentrer au moins une fois dans le `while`, la condition `$nb -ne $input` ne doit pas être vérifiée. Puisque `1 <= nb <= 1000`, on peut initialiser `input` à 0.

📝 **Remarque**

Il existe en Bash la boucle `until` : elle fonctionne exactement comme la boucle `until` mais sur la condition contraire. Ici, on aurait donc plus remplacer `while [ nb -ne input ]` par `until [ nb -eq input ]`.

Exercice 7 - Statistiques

💡 **Notions abordées**

- Synthèse de tout ce qui précède
- Déréférencement de variable
- Tableaux
- Instruction `exit`

📝 **Remarque**

Pour cet exercice en particulier, les solutions proposées sont *des* solutions parmi beaucoup d'autres !

Question 1

Une première version, volontairement pas la plus simple, sans le test de validité des paramètres :

```bash
#!/bin/bash

min=101
max=-101
somme=0

for i in `seq 1 3`; do
    if [ ${!i} -lt $min ]; then min=${!i}; fi
    if [ ${!i} -gt $max ]; then max=${!i}; fi
    somme=$((somme + ${!i}))
done

echo "Le nombre minimum est $min, le max est $max et la moyenne est $((somme / 3))"
```

💡 Explications

- La boucle `for` écrite ici est l'équivalente de la boucle `for(int i = 1; i <= 3; i++)` qu'on aurait pu rencontrer dans d'autres langages, comme C ou Java.
- L'expression `$i` renvoie la valeur de `i` et non la valeur du *i-ème paramètre*. C'est pourquoi on doit *déréférencer* la variable `i`.

Test de validité des paramètres

Pour tester la validité des paramètres, on peut commencer par écrire une fonction qui teste la validité d'**un** paramètre. Par exemple :

```bash
function verifParam() {
    if [ -z $1 ] || ! [[ $1 =~ ^-?[0-9]+$ ]] || [ $1 -lt -100 ] || [ $1 -gt 100 ]; then
        return 1
    else
        return 0
    fi
}
```

💡 Explications

1. Cette fonction commence par tester que le paramètre n'est pas nul
2. Ensuite elle vérifie si la valeur de ce paramètre est un nombre entier, à l'aide de l'expression rationnelle indiquée (c'est une version simplifiée de l'expression de l'exercice 3, qui ne tient pas compte de la partie décimale)

3. Enfin, elle teste si cet entier ce trouve bien dans l'intervalle attendu

Si l'un de ces tests échoue, la fonction renvoie le code d'erreur 1. En cas de succès, comme d'habitude, elle renvoie 0.

Pour tester **tous** les paramètres, comme ici on n'en a que trois, on pourrait éventuellement se contenter de la fonction suivante :

```
function verifTousParams() {
    if ! verifParam $1  || ! verifParam $2  || ! verifParam $3; then
        echo "Les 3 paramètres doivent être compris entre -100 et 100 !"
        exit 1
    fi
}

verifTousParams $1 $2 $3
```

💡 Auto-test

Il est important de bien comprendre que la notation des paramètres :

- La fonction `verifTousParams` est appelée avec **trois** paramètres ; à l'intérieur de cette fonction, les valeurs de ces paramètres sont identifiées respectivement par `$1`, `$2` et `$3`
- Ces trois paramètres sont donnés **successivement** à la fonction `verifParam` qui, elle, n'attend qu'**un seul** paramètre. Donc, **quel que soit le paramètre qu'on lui transmet**, sa valeur sera toujours identifiée par `$1` dans cette fonction.
- Quelle est la différence entre `return 1` et `exit 1` ?

Question 2

La version précédente a un problème majeure : elle n'est pas générique ; le nombre de paramètres (3), est présent plusieurs fois en dur, et il est donc impossible d'utiliser ce programme avec deux valeurs ou avec dix valeurs.

On peut résoudre ce problème très simplement, en itérant sur la *liste des paramètres* (ce qu'on aurait d'ailleurs déjà pu faire à la question précédente, mais la solution proposée avait l'intérêt pédagogique d'illustrer le déréférencement). La boucle `for` devient ainsi :

```
for param in $*; do
    if [ $param -lt $min ]; then min=$param; fi
    if [ $param -gt $max ]; then max=$param; fi
    somme=$((somme + $param))
done


echo "Le nombre minimum est $min, le max est $max et la moyenne est $((somme / $#))"
```

⚠ Ce programme présente désormais un bug potentiel ! Comme on divise la somme par le nombre de paramètre, le script peut planter si on l'appelle sans paramètre (division par 0). On corrige ce problème en rajoutant un test avant de rentrer dans la boucle :

```
if [ $# -eq 0 ]; then
   echo "Utilisation: $0 nb1 [nb2...]"
   exit 1
fi
```

L'autre problème est qu'on ne peut plus utiliser la fonction `verifTousParams`, qui fonctionnait avec trois paramètres. En réalité, on peut complètement se passer de cette fonction : il suffit d'appeler "nous-mêmes" la fonction `verifParam`. Ce qui était d'abord un problème devient ainsi un avantage, car on a un code plus concis.

```
for param in $*; do
    verifParam $param  # <== on teste la validité du paramètre courant
    if [ $param -lt $min ]; then min=$param; fi
    if [ $param -gt $max ]; then max=$param; fi
    somme=$((somme + $param))
done
```

💡 La logique entre les deux solutions est légèrement différente :

- auparavant, on testait la validité de **tous** les paramètres **avant** d'effectuer les calculs
- ici, on teste la validité de chaque paramètre **au fur et à mesure** des calculs.

⚠ Dans le cas présent, cette deuxième solution ne pose pas de problème car les calculs à effectuer sont très rapides. Mais dans le cas de calculs très longs, il vaut mieux s'assurer de la validité de tous les paramètres avant de démarrer les calculs !!! Dans ce cas, la meilleure solution serait de conserver la fonction `verifTousParams` mais en y insérant une boucle qui teste **tous** les paramètres.

La solution complète :

```
#!/bin/bash

function verifParam() {
    if [ -z $1 ] || ! [[ $1 =~ ^-?[0-9]+$ ]] || [ $1 -lt -100 ] || [ $1 -gt 100 ]; then
        echo "Les paramètres doivent être compris entre -100 et 100 !"
        exit 1
    fi
}

if [ $# -eq 0 ]; then
   echo "Utilisation: $0 nb1 [nb2...]"
   exit 1
fi

min=101
max=-101
somme=0

for param in $*; do
    verifParam $param
    if [ $param -lt $min ]; then min=$param; fi
    if [ $param -gt $max ]; then max=$param; fi
    somme=$((somme + $param))
done


echo "Le nombre minimum est $min, le max est $max et la moyenne est $((somme / $#))"
```

Question 3

Ici, les valeurs ne sont plus données en paramètres mais saisies au fur et à mesure et stockées dans un tableau (bien qu'on puisse s'en passer ici, l'idée de l'exercice était d'aborder les tableaux Bash).

```bash
#!/bin/bash

function verifParam() {
    if [ -z $1 ] || ! [[ $1 =~ ^-?[0-9]+$ ]] || [ $1 -lt -100 ] || [ $1 -gt 100 ]; then
        echo "Les paramètres doivent être compris entre -100 et 100 !"
        return 1
    fi
}

val=0
tab=()

while [ $val != 'q' ]; do
    read -p "Saisissez un nombre entre -100 et 100, ou 'q' pour arrêter la saisie : " val

    if [ $val != 'q' ]; then
        verifParam $val
        if [ $? -eq 0 ]; then
            tab[${#tab[@]}]=$val
        fi
    fi
done;

min=101
max=-101
somme=0

for val in ${tab[*]}
do
    if [ $val -lt $min ]; then min=$val; fi
    if [ $val -gt $max ]; then max=$val; fi
    somme=$((somme + $val))
done

if [ ${#tab[@]} -gt 0 ]; then
    echo "Valeurs saisies : ${tab[*]}"
    echo "Le nombre minimum est $min, le max est $max et la moyenne est $((somme /
${#tab[@]}))"
fi
```

💡 Explications

j'ai légèrement changé la logique du programme :

- les valeurs sont d'abord toutes stockées dans le tableau avant de lancer les calculs
- le programme ne s'arrête plus quand on saisie une mauvaise valeur, elle n'est juste pas stockée dans le tableau (d'où le remplacement de `exit 1` par `return 1` dans la fonction).

Exercice 8

```
#!/bin/bash

echo -n "FG \ BG   "

# Couleur du texte
for clfg in {0..7} {30..37} ; do
#for clfg in 0 1 4 5 7 {30..37} ; do
    if [ $clfg -ne 0 ]; then echo -en "$clfg\t"; fi
    # Couleur de fond
    for clbg in {40..47}; do
        if [ $clfg -eq 0 ]; then
            echo -n "$clbg    ";
        else
            echo -en "\e[${clbg};${clfg}m Bash \e[0m"
        fi
    done
    echo #Newline
done
```