ACTIVIDAD PRÁCTICA: BASH SCRIPTING

Grupo 1: Manipulación básica de variables y cadenas

Objetivo: Familiarizarse con la sintaxis básica de Bash scripting, declarar variables, mostrar su valor y manipular cadenas de texto.

Ejercicio 1.1: Crear un script en Bash que muestre el valor de seis variables de entorno del sistema. Requisitos:

1. El script debe mostrar el valor de las siguientes variables de entorno:
    - LOGNAME: El nombre del usuario con el que se ha iniciado sesión.
    - HOME: El directorio home del usuario.
    - SHELL: El shell predeterminado del usuario.
    - PWD: El directorio de trabajo actual.
    - USER: El nombre de usuario del que se ha iniciado sesión.
    - SSH_TTY: El terminal asociado a la sesión SSH, si está disponible.
2. El script debe mostrar el valor de cada una de estas variables de entorno de forma clara y enunciativa, por ejemplo:
    - "La variable LOGNAME tiene el valor XXX"
    - "La variable HOME tiene el valor YYY"
3. Pasos adicionales:
    - El script puede comenzar limpiando la pantalla para una mejor presentación, aunque esto es opcional.

Ejercicio 1.2: Crear un script en Bash que pida un nombre de usuario y un mensaje, y luego envíe ese mensaje al usuario especificado utilizando el comando write. Requisitos:

1. El script debe pedir al usuario que ingrese un nombre de usuario.
2. Luego debe solicitar el mensaje que quiere enviarse al usuario.
3. El mensaje quedará guardado en un archivo .txt para ser enviado

Pasos del ejercicio:

1. Pedir un nombre de usuario y almacenarlo en una variable.
2. Pedir un mensaje y almacenarlo en una variable.

3. Usar un redireccionamiento para guardar el mensaje en un archivo temporal mensaje.txt. El formato a enviar es "**USUARIO: MENSAJE**"
4. Mostrar el contenido del archivo mensaje.txt para confirmar el mensaje correcto

Ejercicio 1.3: Crear un script en Bash que reciba varios parámetros al ejecutarse y muestre un mensaje indicando el valor de cada uno de los parámetros. Además, debe mostrar el número total de parámetros y el nombre del archivo del script. Requisitos:

1. El script debe aceptar hasta 9 parámetros de entrada.
2. El script debe mostrar un mensaje para cada parámetro indicando el valor de los parámetros $1, $2, ..., $9.
3. El script debe mostrar el número total de parámetros recibidos utilizando $#.
4. El script debe mostrar todos los parámetros concatenados utilizando $*.
5. El script debe mostrar el nombre del archivo ejecutado utilizando $0.

Pasos del ejercicio:

1. Al ejecutar el script, se deben pasar varios parámetros en la línea de comandos.
2. El script debe imprimir un mensaje para cada parámetro indicando su valor en la forma:
   El parámetro $1 es <valor1> El parámetro $2 es <valor2> ...
3. Además de los mensajes de los parámetros, el script debe mostrar:
   - El número total de parámetros.
   - Todos los parámetros juntos.
   - El nombre del archivo que contiene el script.

Ejercicio 1.4: Crear un script en Bash que pida al usuario dos cadenas de texto, verifique si alguna de ellas está vacía y, finalmente, compare ambas cadenas para determinar si son iguales o no. Requisitos:

1. El script debe pedir al usuario que ingrese dos cadenas de texto.
2. El script debe verificar si la primera cadena (cadena1) está vacía. Si está vacía, mostrar un mensaje indicando que "La cadena1 está vacía", de lo contrario, mostrar que "La cadena1 no está vacía".

3. El script debe verificar si la segunda cadena (cadena2) está vacía. Si está vacía, mostrar un mensaje indicando que "La cadena2 está vacía", de lo contrario, mostrar que "La cadena2 no está vacía".

4. El script debe comparar ambas cadenas y mostrar un mensaje indicando si son iguales o no.

Pasos del ejercicio:

1. Al ejecutar el script, el usuario deberá ingresar dos cadenas de texto.
2. El script debe verificar si cada una de las cadenas ingresadas está vacía.
3. El script debe realizar una comparación entre ambas cadenas
4. Según el resultado de la comparación, el script debe indicar si las cadenas son iguales o diferentes.

Ejercicio 1.5: Crear un script que realice las cuatro operaciones básicas (suma, resta, multiplicación, división) con dos números proporcionados como parámetros. El script debe comprobar la cantidad de parámetros y realizar diferentes acciones según el número de parámetros ingresados, mostrando mensajes adecuados en cada caso. Requisitos:

1. El script debe aceptar dos parámetros numéricos desde la línea de comandos.
2. El script debe verificar la cantidad de parámetros y comportarse de la siguiente manera:
    - Si **no se ha introducido ningún parámetro**, preguntará al usuario si desea introducirlos ahora, y si la respuesta es afirmativa, solicitará los dos números.
    - Si **se ha introducido solo un parámetro**, preguntará al usuario si desea introducir el segundo, y si la respuesta es afirmativa, usará el primer parámetro proporcionado y solicitará el segundo.
    - Si **se han introducido más de dos parámetros**, el script tomará los dos primeros parámetros y mostrará un mensaje indicando que se han proporcionado demasiados parámetros.
    - Si **se han introducido exactamente dos parámetros**, el script procederá a realizar las operaciones con esos dos números.
3. Una vez que se han obtenido los dos números, el script debe realizar y mostrar los resultados de las cuatro operaciones básicas:
    - **Suma**: número1 + número2
    - **Resta**: número1 - número2
    - **Multiplicación**: número1 * número2

- **División**: número1 / número2 (debe manejarse adecuadamente la división por cero)

Mensajes a mostrar según la cantidad de parámetros:

- Si no se introducen parámetros:
  "No ha introducido ninguno. ¿Quiere ahora s/n?"
- Si se introduce solo un parámetro:
  "Ha introducido uno. ¿Quiere ahora s/n?"
- Si se introducen más de dos parámetros:
  "Demasiados parámetros, tomo los dos primeros."
- Si se introducen exactamente dos parámetros:
  "CORRECTO"

Pasos del ejercicio:

1. Utilizar la variable especial $# para determinar cuántos parámetros se han pasado al script.
2. Dependiendo de la cantidad de parámetros, mostrar los mensajes correspondientes y utilizar el comando read para obtener los números del usuario.
3. Utilizar $(()) para realizar las operaciones aritméticas.
4. Imprimir los resultados de cada operación de forma clara.
5. Incluir una condición para evitar la división por cero, mostrando un mensaje de error en ese caso.

Grupo 2: Estructuras de control de flujo

Objetivo: Aprender a utilizar condicionales if-else y bucles (while, until) para controlar el flujo de ejecución de un script.

Ejercicio 2.1: Crear un script en Bash que reciba una nota numérica e imprima la calificación correspondiente en formato alfabético, siguiendo el sistema de calificación estándar:

- **Sobresaliente** para notas de 9 o más.
- **Notable** para notas de 7 o más, pero menores que 9.
- **Bien** para notas de 6 o más, pero menores que 7.
- **Suficiente** para notas de 5 o más, pero menores que 6.
- **Insuficiente** para notas menores que 5.

Requisitos:

1. El script debe solicitar al usuario que ingrese una nota numérica.
2. Debe validar que la entrada es un número entero positivo.

3. Según la nota proporcionada, debe mostrar el correspondiente nivel de calificación alfabética (Sobresaliente, Notable, Bien, Suficiente, Insuficiente).
4. Si el usuario ingresa un valor no numérico, el script debe mostrar un mensaje de error e indicar que se debe introducir un número válido.

Mensajes:

- Si la entrada no es un número válido, mostrar: "Por favor, introduce un número válido."
- Dependiendo de la nota ingresada, mostrar el correspondiente nivel de calificación (por ejemplo, "Sobresaliente", "Notable", etc.).

Ejercicio 2.2: Crear un script en Bash que simule un menú interactivo con las siguientes opciones:

1. Calcular el área de un rectángulo.
2. Calcular el perímetro de un rectángulo.
3. Salir.

Se deberán cumplir los siguientes requisitos:

- Mostrar un menú con las opciones disponibles.
- Leer la opción seleccionada por el usuario.
    - Si la opción es 1, solicitar la base y la altura del rectángulo y calcular su área.
    - Si la opción es 2, solicitar la base y la altura del rectángulo y calcular su perímetro.
    - Si la opción es 3, mostrar un mensaje de despedida y terminar el script.
    - Si la opción es inválida, mostrar un mensaje de error.

Ejercicio 2.3: Crear un script en Bash que permita al usuario introducir números de manera continua hasta que se ingrese el número 999. Después, se le preguntará si desea ver los números que ha introducido. Si la respuesta es afirmativa, se le permitirá seleccionar un orden para mostrar los números (orden establecido, ascendente o descendente). Requisitos**:**

1. El script debe permitir al usuario ingresar números uno por uno, hasta que el número 999 sea introducido. El número 999 no debe ser registrado.
2. Después de que el usuario termine de ingresar los números, el script debe preguntarle si desea ver los números introducidos. La respuesta debe ser **sí** (s/S) o **no** (n/N). Si elige **no**, el script terminará.

3. Si el usuario elige **sí** (s/S), el script debe preguntarle el orden en el que quiere ver los números:
    - **O** (Orden de ingreso): Los números se mostrarán en el orden en que fueron introducidos.
    - **A** (Ascendente): Los números se mostrarán de menor a mayor.
    - **D** (Descendente): Los números se mostrarán de mayor a menor.
4. El script debe manejar las entradas correctamente, asegurándose de que el número 999 no se incluya en la lista de números y de que el usuario pueda elegir el tipo de orden de los números.
5. El archivo donde se almacenan temporalmente los números introducidos debe eliminarse al final del script.

Mensajes:

- Después de ingresar todos los números, se debe mostrar el siguiente mensaje: "¿Quieres ver los números introducidos?(s/n)"
- Si el usuario elige **sí**, debe preguntarse: "¿ Orden de ingreso, ascendente o descendente?(o/a/d)"
- Si el usuario elige una opción válida de orden, se mostrarán los números según esa opción. En caso de que elija una opción no válida, se debe mostrar: "Opción no válida"
- El script terminará con el mensaje: "Hasta la vista"

Ejercicio 2.4: Crear un script que realice la misma pregunta tres veces: "¿Cuánto es 2 + 2?". Si el usuario responde correctamente, el script debe mostrar un mensaje de "Correcto" y finalizar. Si el usuario responde incorrectamente, el script debe indicarle cuántos intentos le quedan y permitirle intentar nuevamente. Después de tres intentos fallidos, el script debe mostrar un mensaje de "Game Over" y finalizar.

Requisitos:

1. El script debe hacer la pregunta "¿Cuánto es 2 + 2?" al usuario hasta tres veces.
2. Si el usuario responde correctamente (es decir, si su respuesta es **4**), el script debe mostrar el mensaje "CORRECTO, acertado en el intento X", donde X es el número de intentos que el usuario ha hecho. Luego, el script debe finalizar inmediatamente.

3. Si el usuario responde incorrectamente, el script debe indicar cuántos intentos le quedan con el mensaje "Te quedan X intentos", donde X es el número de intentos restantes.
4. El script debe permitir un máximo de tres intentos. Si el usuario no acierta en tres intentos, el script debe mostrar el mensaje "Game Over" y finalizar.

Ejercicio 2.5: Crear un script que lea un archivo llamado **pregyresp.txt**, que contiene preguntas matemáticas junto con las respuestas correctas. El script debe presentar cada pregunta al usuario, permitirle introducir una respuesta y luego comparar su respuesta con la correcta. Al final, debe mostrar el número total de aciertos.

Descripción del archivo pregyresp.txt:

El archivo **pregyresp.txt** contiene una serie de preguntas matemáticas y sus respuestas correctas, con el siguiente formato en cada línea:

```
pregunta;respuesta_correcta
```

Donde:

- **pregunta** es una operación matemática.
- **respuesta_correcta** es la solución correcta a la operación matemática.

Formato del archivo (ejemplo):

```
2+2;4
3+3;6
4×4;16
3-1;2
```

Requisitos:

1. El script debe leer el archivo **pregyresp.txt** línea por línea.
2. Para cada línea, el script debe separar la pregunta y la respuesta correcta (utilizando el delimitador ;).
3. El script debe mostrar la pregunta al usuario y permitirle ingresar una respuesta.
4. El script debe comparar la respuesta del usuario con la respuesta correcta.
5. El script debe contar cuántas respuestas son correctas y, al finalizar, mostrar el número total de aciertos.

Mensajes esperados:

- El script debe mostrar la pregunta al usuario y esperar su respuesta. Ejemplo:

2+2?

- Si la respuesta es correcta, el script incrementará el contador de aciertos.
- Al final, el script debe mostrar el número total de aciertos, por ejemplo:

Tienes 3 aciertos

Grupo 3: Manipulación de archivos y directorios

Objetivo: Aprender a trabajar con archivos y directorios, incluyendo la comprobación de su existencia, tipo y permisos, así como la lectura y escritura de datos.

Ejercicio 3.1: Crear un script que, dado el nombre de un archivo, determine y muestre el tipo de archivo y sus permisos. El script debe verificar si el archivo existe, identificar si es un archivo regular, directorio, archivo especial, entre otros, y además, verificar los permisos de lectura, escritura y ejecución. Requisitos:

1. El script debe pedir al usuario que ingrese la ruta de un archivo.
2. Si el archivo existe, el script debe:
    - Mostrar el tipo de archivo (por ejemplo, archivo de bloques, archivo de caracteres, directorio, archivo ordinario, archivo simbólico).
    - Verificar y mostrar los permisos del archivo: lectura, escritura y ejecución.
3. Si el archivo no existe, el script debe indicar que el archivo no existe.

Mensajes esperados:

1. Si el archivo existe:
    - Indicar el tipo de archivo.
    - Indicar los permisos de lectura, escritura y ejecución si están habilitados.
2. Si el archivo no existe:
    - Indicar que el archivo no existe.

Ejemplo de ejecución:

```
$ ./script3.1_tipoarchivo

Archivo: /dev/sda

El archivo /dev/sda existe

Es un archivo especial de bloques

Tiene permiso de escritura

Tiene permiso de ejecución
```

Detalles del ejercicio:

- El script debe identificar correctamente el tipo de archivo utilizando los operadores de prueba de Bash, como -e (para verificar si el archivo existe), -b (para archivos especiales de bloques), -c (para archivos de caracteres), -d (para directorios), -f (para archivos ordinarios), y -h (para archivos simbólicos).
- El script debe verificar los permisos utilizando los operadores de prueba como -r (lectura), -w (escritura), y -x (ejecución).
- En caso de que el archivo no exista, debe mostrar un mensaje adecuado.

Notas adicionales:

- El archivo puede ser un archivo regular, un directorio o un archivo especial como un archivo de bloques o caracteres.
- Los permisos se pueden verificar en archivos regulares o directorios, pero no en todos los tipos de archivos especiales.

Ejercicio 3.2: Crear un script que, dada una extensión de archivo, busque todos los archivos que coincidan con dicha extensión en el directorio actual. Para cada archivo encontrado, el script debe mostrar su nombre, el contenido del archivo y un separador de líneas. Requisitos:

1. El script debe solicitar al usuario que ingrese una extensión de archivo (por ejemplo, .txt, .log, .sh).
2. El script debe buscar todos los archivos en el directorio actual que tengan la extensión proporcionada.
3. Para cada archivo encontrado, el script debe:

- Mostrar el nombre del archivo.
- Mostrar el contenido del archivo.
- Imprimir un separador de líneas consistente (por ejemplo**,** ====== o ==========).

Mensajes esperados:

1. Para cada archivo que coincida con la extensión:
   - El nombre del archivo.
   - El contenido del archivo.
   - Un separador de líneas.

Ejemplo de ejecución:

```
$ ./script3.2
Extension del archivo: txt
Nombre del archivo: archivo1.txt
Contenido del archivo 1
==========
Nombre del archivo: archivo2.txt
Contenido del archivo 2
==========
```

Detalles del ejercicio:

- El script debe utilizar un bucle for para recorrer los archivos que coinciden con la extensión proporcionada por el usuario.
- Para mostrar el contenido de cada archivo, se utilizará el comando cat.
- El separador debe ser una línea de caracteres (por ejemplo, ==========) para diferenciar claramente los contenidos de los archivos.

Grupo 4: Combinación de comandos y redirecciones

Objetivo: Aprender a combinar diferentes comandos de Bash utilizando tuberías (pipes) y redirecciones para procesar información de forma eficiente.

Ejercicio 4.1: Crear un script que combine la información de dos archivos de texto (arch1.txt y arch2.txt) para generar un archivo de salida (union.txt) con el siguiente formato:

- En **arch1.txt**, cada línea contiene el nombre de un equipo de fútbol y el nombre de su estadio, separados por una coma.
- En **arch2.txt**, cada línea contiene los colores del equipo y el nombre del estadio, separados por un punto y coma.
- El script debe combinar la información de ambos archivos para generar un nuevo archivo, **union.txt**, donde cada línea tiene la siguiente estructura:

[Nombre del equipo];[Colores del equipo];[Estadio]

Requisitos:

1. El script debe leer ambos archivos **arch1.txt** y **arch2.txt**.
2. Debe combinar la información de los archivos de tal manera que:
    - Para cada línea en **arch1.txt**, se obtenga el nombre del equipo y el estadio.
    - Para cada equipo de **arch1.txt**, debe buscarse el color correspondiente de camiseta en **arch2.txt**, utilizando el nombre del equipo.
    - El archivo **union.txt** debe contener, por cada línea, el nombre del equipo, los colores del equipo y el nombre del estadio en el formato indicado.

Ejemplo de entrada:

arch1.txt:

```
BOCA,BOMBONERA
TALLERES,BOUTIQUE
```

arch2.txt:

```
AZUL Y BLANCO;TALLERES
AZUL Y AMARILLO;BOCA
```

Ejemplo de salida en union.txt:

```
BOCA;AZUL Y AMARILLO;BOMBONERA
TALLERES;AZUL Y BLANCO; BOUTIQUE
```

Detalles del ejercicio:

1. El script debe usar un bucle para recorrer todas las líneas de **arch1.txt**.
2. Para cada equipo de **arch1.txt**, debe buscar su nombre en **arch2.txt** y extraer los colores correspondientes.
3. El formato de la salida debe ser el siguiente: [Equipo];[Colores];[Estadio].
4. El archivo **union.txt** debe generarse y guardarse con la información combinada.

Notas adicionales:

- Si un equipo de **arch1.txt** no tiene una coincidencia en **arch2.txt**, no se debe escribir nada para ese equipo.
- El archivo **union.txt** debe ser creado desde cero al inicio del script (usando rm union.txt para eliminar cualquier archivo previo).

Comportamiento esperado:

1. El script procesará ambos archivos.
2. El archivo **union.txt** se generará correctamente con la información combinada en el formato adecuado.
3. El script debe ser capaz de manejar archivos con múltiples líneas y combinar la información correctamente.

Ejercicio 4.2: Crear un script de gestión de una agenda de clubes, donde se puedan realizar diversas acciones sobre un archivo llamado **agenda.txt**. Este archivo contiene los siguientes datos de cada club: nombreclub, provincia, localidad, codigoclub.

El script debe ofrecer un menú interactivo con las siguientes opciones:

1. **Ver club:** Permite buscar y ver los datos de un club existente.
2. **Gestionar:** Permite realizar acciones de gestión sobre los clubes, que incluyen:
    - Insertar un nuevo club.
    - Eliminar un club existente.
    - Modificar los datos de un club existente.
3. **Salir:** Termina la ejecución del script.

Detalles del ejercicio:

1. **Menú principal:**
    - El script debe mostrar el siguiente menú:

```
1. Ver club
2. Gestionar
3. Salir
```

    - El usuario selecciona una opción ingresando un número.

2. **Opción 1: Ver club**
    - El usuario debe poder buscar un club por su nombre. Si el club existe en el archivo **agenda.txt**, se mostrarán los datos del club.

- Si el club no existe, se debe mostrar un mensaje indicando que no se encontró el club.

3. **Opción 2: Gestionar**
   - Al seleccionar esta opción, se debe mostrar un submenú con las siguientes opciones:

```
1. Insertar club
2. Eliminar club
3. Modificar
4. Salir
```

   - El usuario puede elegir una opción para:
     - **Insertar un club**: El script debe comprobar que el club no exista ya en la agenda. Si no existe, se le pedirá al usuario los datos del nuevo club (nombre, provincia, localidad, código) y se añadirá al archivo **agenda.txt**.
     - **Eliminar un club**: El script debe comprobar si el club existe. Si existe, el club será eliminado del archivo **agenda.txt**.
     - **Modificar un club**: El script debe permitir modificar los datos de un club existente. Si el club existe, se eliminarán sus datos actuales y se actualizarán con la nueva información proporcionada por el usuario.
4. **Validación de existencia:**
   - El script debe comprobar la existencia de los clubes antes de realizar cualquier operación (ya sea ver, insertar, eliminar o modificar). Para ello, el nombre del club se utilizará como clave primaria, ya que se supone que no puede haber dos clubes con el mismo nombre.

Formato de entrada en el archivo agenda.txt:

```
nombreclub,provincia,localidad,codigoclub
```

Ejemplo de archivo agenda.txt:

```
TALLERES,CORDOBA,CORDOBA,123
UNION,SANTA FE,SANTA FE,456
```

Ejemplo de ejecución:

```
1. Ver club
2. Gestionar
3. Salir
Elige una opción: 1
```

```
Ingrese el club: TALLERES
TALLERES,CORDOBA,CORDOBA,123

1. Insertar club
2. Eliminar club
3. Modificar
4. Salir
Elige una opción: 1
Ingrese el club: REAL MADRID
Ingrese el nombre de su provincia: Madrid
Ingrese su localidad: Madrid
Ingrese su código: 789
```

Requisitos adicionales:

- Si un club ya existe, no se podrá insertar de nuevo y se debe informar al usuario.
- Si el club no existe, no podrá ser eliminado ni modificado, y se debe informar al usuario de la ausencia del club.

Ejercicio 4.3: A partir de un archivo llamado **puntuaciones.txt**, que contiene el nombre de varios jugadores y sus puntuaciones actuales, el script debe permitir al usuario introducir las nuevas puntuaciones obtenidas por cada jugador en una carrera. Posteriormente, debe actualizar el archivo con las nuevas puntuaciones y mostrar la lista de jugadores ordenada de mayor a menor según sus puntuaciones totales.

Detalles del ejercicio:

1. **Archivo de entrada:** El archivo **puntuaciones.txt** tiene el siguiente formato, con cada línea representando el nombre del jugador y su puntuación actual separada por una coma:

```
ALONSO,12
COLAPINTO,10
HAMILTON,3
```

2. **Proceso:**
    - El script debe leer cada línea del archivo **puntuaciones.txt**, mostrar al usuario el nombre de cada jugador y pedirle que introduzca los puntos obtenidos por ese jugador en la carrera.

- Luego, el script debe sumar los puntos obtenidos por el jugador a los puntos anteriores y actualizar la puntuación total de cada jugador.
- Finalmente, el script debe reescribir el archivo **puntuaciones.txt** con las puntuaciones actualizadas y mostrar la lista de jugadores ordenada de mayor a menor puntuación.

3. **Interacción con el usuario:** El script debe solicitar la puntuación de cada jugador de la siguiente manera:
4.

```
Puntos de ALONSO: ___
Puntos de COLAPINTO: ___
Puntos de HAMILTON: ___
```

5. **Salida esperada:** Al finalizar la actualización de las puntuaciones, el script debe mostrar la lista de jugadores ordenada de mayor a menor puntuación, como en el siguiente ejemplo:

```
HAMILTON,15
COLAPINTO,14
ALONSO,13
```

Requisitos adicionales:

- El archivo **puntuaciones.txt** debe ser actualizado con las nuevas puntuaciones.
- La lista de jugadores debe ser mostrada ordenada de mayor a menor puntuación al final del script.

Grupo 5: Funciones y modularidad

Objetivo: Familiarizarse con la sintaxis básica de Bash scripting, declarar variables, mostrar su valor y manipular cadenas de texto.

Ejercicio 5.1: Repetir el ejercicio 4.3, pero utilizando funciones para organizar y modularizar el código. El objetivo es leer las puntuaciones de los jugadores, permitir que el usuario ingrese nuevas puntuaciones para cada uno y actualizar el archivo **puntuaciones.txt** con los resultados. Posteriormente, se debe mostrar el archivo de puntuaciones ordenado de mayor a menor.

Detalles del ejercicio:

1. **Archivo de entrada:** El archivo **puntuaciones.txt** contiene el nombre de varios jugadores y sus puntuaciones actuales, separadas por una coma, de la siguiente forma:

```
ALONSO,12
COLAPINTO,10
HAMILTON,3
```

2. **Funciones del script:**
   - **lee_jugador_y_puntos(linea):** Esta función recibe un número de línea como parámetro, lee esa línea del archivo **puntuaciones.txt**, y extrae el nombre del jugador y sus puntos actuales. Retorna el nombre del jugador y su puntuación actual.
   - **actualizar_puntos(jugador, puntosantiguos, puntos):** Esta función recibe el nombre del jugador, sus puntos anteriores y los nuevos puntos obtenidos. Calcula los nuevos puntos sumando los anteriores con los nuevos, y los guarda en un archivo temporal **puntuaciones.temp**.
   - **ordenar_y_mostrar():** Esta función ordena el archivo **puntuaciones.txt** de mayor a menor puntuación y muestra la lista resultante.
   - **procesar_puntuaciones():** Función principal que coordina todo el proceso. Recorre el archivo **puntuaciones.txt**, llama a las funciones correspondientes para obtener los datos de cada jugador, solicitar las nuevas puntuaciones y actualizar los registros. Finalmente, reemplaza el archivo original con los datos actualizados y llama a la función que ordena y muestra los resultados.
3. **Interacción con el usuario:** El script solicita al usuario que introduzca las nuevas puntuaciones para cada jugador. La interacción para cada jugador será similar a:

```
Puntos de ALONSO: ___
Puntos de COLAPINTO: ___
Puntos de HAMILTON: ___
```

4. **Salida esperada:** Al finalizar la actualización de las puntuaciones, el script debe mostrar la lista de jugadores ordenada de mayor a menor puntuación, como en el siguiente ejemplo:

```
HAMILTON,15
COLAPINTO,14
ALONSO,13
```

Requisitos adicionales:

- El archivo **puntuaciones.txt** debe ser actualizado con las nuevas puntuaciones.

- El código debe estar modularizado utilizando funciones para mejorar la legibilidad y reutilización.
- La lista de jugadores debe ser mostrada ordenada de mayor a menor puntuación al final del script.

Grupo 6: Administración básica del sistema

Objetivo: Aprender a realizar tareas básicas de administración del sistema, como la gestión de usuarios y la comprobación de información del sistema.

Ejercicio 6.1: Crear un script que permita gestionar usuarios en el sistema. El script debe permitir añadir nuevos usuarios y eliminar usuarios existentes, siempre y cuando el usuario que ejecute el script tenga privilegios de administrador (root). Además, se debe verificar si el usuario que se quiere añadir ya existe en el sistema antes de proceder con su creación.

Detalles del ejercicio:

1. **Menú principal:**
   - El script debe mostrar un menú con tres opciones principales:
     - **1. Información de usuario:** Solicitar el nombre de un usuario y mostrar su información usando el comando finger.
     - **2. Gestión de usuario:** Redirigir a otro script (script20_gestion.sh) para gestionar los usuarios (añadir, eliminar, etc.).
     - **3. Salir:** Finalizar el programa.
2. **Gestión de usuarios (en script20_gestion.sh):**
   - El script debe mostrar un menú con tres opciones:
     - **A. Añadir usuario:** Verificar que el usuario que ejecuta el script es **root** antes de permitir añadir un nuevo usuario. Además, comprobar si el usuario a añadir ya existe en el sistema antes de crearlo.
     - **E. Eliminar usuario:** Comprobar que el usuario que ejecuta el script es **root** y verificar si el usuario a eliminar existe antes de proceder con su eliminación.
     - **V. Volver:** Volver al menú principal.
3. **Comprobaciones y validaciones:**
   - **Comprobación de existencia de usuario:** Antes de añadir un nuevo usuario, el script debe verificar que el nombre del usuario no esté ya en el sistema (usando /etc/passwd). Si el usuario ya existe, debe mostrar un mensaje indicándolo.

- **Permisos de administrador (root):** Solo los usuarios con privilegios de **root** deben poder añadir o eliminar usuarios. Si un usuario sin privilegios de administrador intenta realizar estas acciones, el script debe mostrar un mensaje indicando que no tiene permisos suficientes.

4. **Interacción con el usuario:**
   - Cuando se selecciona la opción de añadir un usuario, el script debe pedir el nombre del nuevo usuario y verificar si ya existe. Si no existe, se debe crear el usuario.
   - Al seleccionar la opción de eliminar un usuario, el script debe pedir el nombre del usuario a eliminar y verificar si existe en el sistema antes de proceder con la eliminación.
   - El script debe permitir al usuario volver al menú principal en cualquier momento.

Requisitos adicionales:

- El script debe ser sencillo de usar e interactivo.
- Los usuarios sin privilegios de root no deben poder añadir ni eliminar usuarios.

Ejemplo de salida:

```
Menú de gestión de usuarios
A. Añadir
E. Eliminar
V. Volver
Elige una opción: A
No tiene permiso de administrador.

Menú de gestión de usuarios
A. Añadir
E. Eliminar
V. Volver
Elige una opción: A
Nombre del usuario: nuevo_usuario
Usuario nuevo_usuario añadido con éxito.
```

Nota: Para poder utilizar el comando finger en el script, se puede necesitar instalarlo primero en el sistema con el siguiente comando:

```
sudo apt-get install finger
```

ANEXO: ayudas a la resoluciòn

Grupo 1: Manipulación básica de variables y cadenas

1. **echo**: Muestra texto o el valor de variables en la consola.
    - **echo "Hola Mundo"**: Muestra el texto "Hola Mundo".
    - **echo "La variable HOME tiene el valor" $HOME**: Muestra el texto concatenado con el valor de la variable de entorno HOME.
2. **read**: Lee la entrada del usuario y la almacena en una variable.
    - **read nombre**: Lee una línea de la entrada estándar y la guarda en la variable "nombre".
3. **>**: Redirecciona la salida estándar a un archivo (sobreescribe el archivo).
    - **echo "Hola" > saludo.txt**: Escribe "Hola" en el archivo saludo.txt, sobrescribiendo el contenido si existe.
4. **>>**: Redirecciona la salida estándar a un archivo, añadiendo al final.
    - **echo "Mundo" >> saludo.txt**: Añade "Mundo" al final del archivo saludo.txt.
5. **|**: Tubería (pipe). Redirecciona la salida estándar de un comando a la entrada estándar de otro.
    - **ls -l | grep "txt"**: Lista los archivos y directorios, luego filtra los que contienen "txt".
6. **cat**: muestra el contenido de un archivo
    - **cat mensaje.txt**: muestra en terminal el contenido de mensaje.txt
7. **$variable**: Accede al valor de una variable.
    - **echo $HOME**: Muestra el valor de la variable de entorno HOME.
8. **$1, $2, ...**: Accede a los parámetros posicionales del script.
    - **./script.sh uno dos tres**: $1 sería "uno", $2 sería "dos", etc.
9. **$#**: Muestra el número de parámetros del script.
10. **$***: Muestra todos los parámetros del script.
11. **$0**: Muestra el nombre del script.
12. **-z**: Operador de prueba que verifica si una cadena está vacía.
    - **if [-z "$cadena"]; then echo "La cadena está vacía"; fi**: Verifica si la variable cadena está vacía.
13. **==**: Operador de comparación de cadenas.
    - **if ["$cadena1" == "$cadena2"]; then echo "Las cadenas son iguales"; fi**: Compara dos cadenas.

Grupo 2: Estructuras de control de flujo

1. **if-elif-else**: Estructura condicional.
    - **if [condición]; then comandos; elif [condición]; then comandos; else comandos; fi**: Estructura condicional con múltiples ramas.
2. **case**: Estructura de selección múltiple.
    - **case variable in patrón1) comandos ;; patrón2) comandos ;; *) comandos ;; esac**: Realiza una comparación con múltiples patrones.
3. **while**: Bucle que se ejecuta mientras la condición sea verdadera.
    - **while [condición]; do comandos; done**: Ejecuta los comandos mientras la condición sea verdadera.
4. **until**: Bucle que se ejecuta hasta que la condición sea verdadera.
    - **until [condición]; do comandos; done**: Ejecuta los comandos hasta que la condición sea verdadera.
5. **((...))**: Permite realizar operaciones aritméticas y comparaciones numéricas.
    - **if ((num1 > num2)); then echo "$num1 es mayor que $num2"; fi**: Realiza una comparación numérica.

Grupo 3: Manipulación de archivos y directorios

1. **-e**: Verifica si un archivo o directorio existe.
2. **-b**: Verifica si un archivo es un archivo especial de bloque.
3. **-c**: Verifica si un archivo es un archivo especial de carácter.
4. **-d**: Verifica si un archivo es un directorio.
5. **-f**: Verifica si un archivo es un archivo regular.
6. **-h**: Verifica si un archivo es un enlace simbólico.
7. **-r**: Verifica si un archivo tiene permiso de lectura.
8. **-w**: Verifica si un archivo tiene permiso de escritura.
9. **-x**: Verifica si un archivo tiene permiso de ejecución.
10. **cat**: Muestra el contenido de un archivo.
    - **cat archivo.txt**: Muestra el contenido de archivo.txt.
11. **for**: Bucle que itera sobre una lista de elementos.
    - **for variable in lista; do comandos; done**: Ejecuta los comandos para cada elemento de la lista.

Grupo 4: Combinación de comandos y redirecciones

1. **wc -l**: Cuenta el número de líneas de un archivo.
    - **wc -l archivo.txt**: Cuenta las líneas en archivo.txt.
2. **head**: Muestra las primeras líneas de un archivo.
    - **head -n 5 archivo.txt**: Muestra las primeras 5 líneas de archivo.txt.
3. **tail**: Muestra las últimas líneas de un archivo.
    - **tail -n 2 archivo.txt**: Muestra las últimas 2 líneas de archivo.txt.
4. **cut**: Extrae secciones de cada línea de un archivo.
    - **cut -d "," -f1 archivo.csv**: Extrae el primer campo de cada línea de archivo.csv, usando la coma como delimitador.
5. **grep**: Busca patrones en un archivo.
    - **grep "palabra" archivo.txt**: Muestra las líneas de archivo.txt que contienen "palabra".
6. **sort**: Ordena las líneas de un archivo.
    - **sort archivo.txt**: Ordena las líneas de archivo.txt alfabéticamente.
    - **sort -n archivo.txt**: Ordena las líneas de archivo.txt numéricamente.
    - **sort -r archivo.txt**: Ordena las líneas de archivo.txt en orden inverso.
7. **mv**: Mueve o renombra archivos y directorios.

Grupo 5: Funciones y modularidad

1. **function nombre_funcion() { comandos; }**: Define una función.
    - **function saludo() { echo "Hola"; }**: Define la función saludo que imprime "Hola".
2. **local variable**: Declara una variable local dentro de una función.
3. **IFS**: Internal Field Separator. Controla cómo se dividen las cadenas en palabras.
    - **IFS=":"**: Cambia el delimitador de campos a los dos puntos.
4. **<<<**: Redirecciona una cadena a la entrada estándar de un comando.
    - **grep "texto" <<< "Hola mundo"**: Busca "texto" en "Hola mundo".

Ejemplo para recorrido de archivos con campos separados por punto y coma:

```
while IFS=';' read -r pregunta respuesta_correcta <&3 ; do

{ CUERPO DEL BUCLE}


done 3<  "$archivo"
```

El bucle while lee cada línea del archivo especificado ($archivo). Para cada línea:

- *IFS=';'*: Separa la línea en partes, usando ";" como separador.
- *read -r pregunta respuesta_correcta*: Asigna la primera parte a la variable pregunta y la segunda a respuesta_correcta. El -r evita que los backslashes sean interpretados.
- **Uso de descriptor de archivo diferente:** En lugar de leer directamente del archivo en el bucle while, usamos el descriptor de archivo 3 (<&3 y 3< "$archivo"). Esto evita que el read interno para el usuario se confunda con la lectura del archivo.
- **Lectura separada para cada read:**
    - El while lee del descriptor 3 (el archivo).
    - El read interno lee de la entrada estándar (el usuario).

El bucle continúa hasta que no hay más líneas en el archivo.

Grupo 6: Administración básica del sistema

1. **finger**: Muestra información sobre un usuario.
2. **adduser**: Crea un nuevo usuario.
3. **deluser**: Elimina un usuario.
4. **/etc/passwd**: Archivo que contiene información sobre los usuarios del sistema.
5. **useradd**: Crea una nueva cuenta de usuario.
6. **userdel**: Elimina una cuenta de usuario.
7. **grep -i**: Busca un patrón sin distinguir entre mayúsculas y minúsculas.
    - **grep -i "usuario" archivo.txt**: Busca "usuario" sin importar si está en mayúsculas o minúsculas.