TP-ALGORITMOS

División de Tareas

Grupo 1: Configuración y manejo de archivo (2 personas)

1. **Responsable 1: Cargar configuraciones del archivo**
   - Función clave: `cargar_configuracion()`
     - **Descripción**: Carga la configuración del juego desde un archivo externo `config.txt`, validando que las configuraciones estén dentro de los parámetros permitidos (tiempo y vidas). Se guarda en una estructura `tConfig`.
     - **Parámetros**:
       - `const char* archivo_configuracion`: Ruta al archivo de configuración.
       - `tConfig* configuraciones`: Estructura donde se almacenan las configuraciones.
     - **Retorno**: Ninguno, pero se debe validar si hay errores en el archivo.
   - Función adicional: `validar_configuracion()`
     - **Descripción**: Valida que las configuraciones cargadas desde el archivo sean correctas (ej. no más de 5 vidas, tiempos no mayores a 20 segundos).
     - **Parámetros**:
       - `tConfig* configuraciones`: Estructura con los parámetros de configuración.
     - **Retorno**: 1 si es válido, 0 si no.
2. **Responsable 2: Guardar resultados del juego**
   - Función clave: `guardar_turno()`
     - **Descripción**: Guarda los detalles del turno actual (secuencia, respuestas del jugador, vidas utilizadas, y puntaje) en un archivo de informe.
     - **Parámetros**:
       - `FILE* archivo_informe`: Puntero al archivo de informe.
       - `tTurno* turno_actual`: Estructura que contiene los datos del turno actual.
     - **Retorno**: Ninguno, pero se debe validar errores de archivo.

Grupo 2: Gestión de secuencias y puntajes (2 personas)

3. **Responsable 3: Generar y mostrar secuencias**
   - Función clave: `generar_secuencia()`
     - **Descripción**: Genera una secuencia de colores de longitud `N` utilizando números aleatorios, y la almacena en una estructura `tPila` (ya desarrollada).
     - **Parámetros**:
       - `tPila* pila_secuencia`: Pila que almacena la secuencia generada.
       - `int longitud`: Longitud de la secuencia.
     - **Retorno**: Ninguno.
   - Función adicional: `mostrar_secuencia()`
     - **Descripción**: Muestra la secuencia de colores/letras almacenada en la pila por un tiempo determinado.
     - **Parámetros**:
       - `tPila* pila_secuencia`: Pila con la secuencia.
       - `unsigned int tiempo_visualizacion`: Tiempo que la secuencia es visible en pantalla (en segundos).
     - **Retorno**: Ninguno.
4. **Responsable 4: Validar respuesta y calcular puntaje**
   - Función clave: `validar_respuesta()`
     - **Descripción**: Compara la secuencia ingresada por el jugador con la secuencia correcta almacenada en `tPila`, indicando si es correcta o no.
     - **Parámetros**:
       - `tPila* pila_secuencia`: Secuencia correcta.
       - `tPila* pila_respuesta`: Secuencia ingresada por el jugador.
     - **Retorno**: 1 si es correcta, 0 si es incorrecta.
   - Función adicional: `calcular_puntaje()`
     - **Descripción**: Calcula el puntaje del jugador según las reglas del juego, tomando en cuenta si se usaron vidas.
     - **Parámetros**:
       - `int* puntaje_actual`: Puntaje actual del jugador.
       - `int secuencia_correcta`: 1 si la secuencia es correcta, 0 si es incorrecta.
       - `int vidas_usadas`: Número de vidas que utilizó el jugador en el turno.
     - **Retorno**: Ninguno.

Grupo 3: Gestión de jugadores y flujo de juego (2 personas)

5. **Responsable 5: Manejo de jugadores y turnos**

- Función clave: `ingresar_jugadores()`
  - **Descripción**: Permite ingresar los nombres de los jugadores y los organiza en una cola para gestionar el orden de juego.
  - **Parámetros**:
    - `tCola* cola_jugadores`: Cola donde se insertan los jugadores.
  - **Retorno**: Número de jugadores ingresados.
- Función adicional: `sortear_orden_jugadores()`
  - **Descripción**: Realiza un sorteo aleatorio para determinar el orden de los jugadores en la cola.
  - **Parámetros**:
    - `tCola* cola_jugadores`: Cola con los jugadores.
    - `unsigned int cant`: Cantidad de jugadores.
  - **Retorno**: Ninguno (modifica el orden de la cola).

6. **Responsable 6: Manejo de vidas y finalización de turnos**
   - Función clave: `gestionar_vidas()`
     - **Descripción**: Administra las vidas restantes cuando los jugadores cometen errores, permitiéndoles retroceder movimientos si tienen vidas disponibles.
     - **Parámetros**:
       - `tPila* pila_secuencia`: Pila con la secuencia actual.
       - `int* vidas_restantes`: Puntero a las vidas restantes del jugador.
     - **Retorno**: Ninguno.
   - Función adicional: `finalizar_turno()`
     - **Descripción**: Al finalizar el turno, actualiza el estado del juego y pasa el turno al siguiente jugador.
     - **Parámetros**:
       - `tJugador* jugador_actual`: Jugador que terminó su turno.
     - **Retorno**: Ninguno.

> Nota: Pueden crear mas funciones que complementen las principales.

Tipos de Datos Utilizados

1. **tConfig**: Para manejar configuraciones del juego como nivel de dificultad, tiempos y cantidad de vidas.

```
typedef struct {
    char nivel;  // F = fácil, N = normal, D = difícil
    unsigned int tiempo_muestra;
    unsigned int tiempo_turno;
```

```
    unsigned cant_vidas;
} tConfig;
```

2. **tJugador**: Información del jugador, incluyendo su nombre, puntaje y vidas.

```
typedef struct {
    char nombre[16];
    unsigned int puntaje_turno;
    unsigned int puntaje_total;
    unsigned int vidas_consumidas;
    int vidas_total;
} tJugador;
```

3. **tTurno**: Contiene los datos de cada turno, como el jugador actual y la secuencia correspondiente.

```
typedef struct {
    tJugador* jugador_actual;
    const char* secuencia;
} tTurno;
```

Convención de Estilo de Código

1. Nombres de Variables, Funciones y Estructuras

- **Funciones**:
  - **Formato**: Utiliza `camelCase` (la primera palabra en minúscula, las siguientes en mayúscula).
  - **Ejemplo**:

```
void cargarConfiguracion(const char* archivoConfiguracion,
tConfig* configuraciones);
void mostrarSecuencia(tPila* pilaSecuencia);
```

- **Variables**:
  - **Formato**: Usa `snake_case` (todas las letras en minúscula y separadas por guiones bajos).
  - **Ejemplo**:

```
unsigned int vidas_restantes;
char nombre_jugador[16];
```

```
int puntaje_total;
```

- **Estructuras**:
  - **Formato**: Usa `PascalCase` (todas las palabras comenzando con mayúscula), con el prefijo `t` para indicar que es un tipo.
  - **Ejemplo**:

```
typedef struct {
    char nombre[16];
    unsigned int puntaje_total;
    int vidas_restantes;
} tJugador;

typedef struct {
    char nivel;
    unsigned int tiempo_muestra;
    unsigned int tiempo_turno;
    unsigned int cant_vidas;
} tConfig;
```

2. Formato del Código

- **Indentación**: Usa 2 espacios por nivel de indentación. No utilices tabulaciones. Se puede configurar el editor y el formateador de Codeblocks para hacerlo automáticamente.
- **Longitud de línea**: Mantén las líneas de código con un máximo de 80 caracteres. Si una línea se extiende más allá de esto, divídela en múltiples líneas.
- **Llaves**: Las llaves deben colocarse en líneas separadas y alineadas con el código.

```
if (condicion)
{
  // código aquí
}
else
{
  // código aquí
}
```

3. Estructura del Código

- **Prototipos de funciones**: Coloca los prototipos de funciones al inicio del archivo `.c` o en un archivo de cabecera (`.h`) separado.
- **Agrupación lógica**: Agrupa funciones relacionadas dentro de módulos. Por ejemplo, todas las funciones relacionadas con secuencias estarán en el módulo de secuencias.

4. Comentarios

- **Comentarios de una línea**: Usa `//` para comentarios cortos explicando el propósito de bloques de código o líneas específicas.
  - Ejemplo:

```
// Se carga la configuración del archivo config.txt
cargarConfiguracion("config.txt", &config);
```

- **Comentarios de funciones**: Coloca un bloque de comentarios al inicio de cada función, describiendo brevemente lo que hace, los parámetros que recibe y lo que devuelve.

```
/**
 * cargarConfiguracion
 * Carga la configuración inicial desde un archivo.
 *
 * @param archivoConfiguracion: Nombre del archivo de configuración.
 * @param configuraciones: Estructura tConfig donde se guardan las
configuraciones.
 * @return void
 */
void cargarConfiguracion(const char* archivoConfiguracion, tConfig*
configuraciones) {
    // Código de la función
}
```

5. Manejo de Errores

- **Comprobación de errores**: Cada función que realice una operación de E/S o que pueda fallar (como la lectura de archivos o la asignación de memoria) debe manejar errores de manera explícita.
  - Ejemplo:

```
if (fopen(archivo, "r") == NULL) {
    printf("Error al abrir el archivo\n");
```

```
    return -1;
}
```

6. Estructuras de Datos

- **Declaración de estructuras**: Declara las estructuras de manera que sus nombres sean descriptivos. Usa `typedef` para facilitar el manejo de tipos.

```
typedef struct {
    char nivel;
    unsigned int tiempo_muestra;
    unsigned int tiempo_turno;
    unsigned int cant_vidas;
} tConfig;
```

7. Funciones y Parámetros

- **Uso de punteros**: Cuando sea necesario modificar una variable pasada a una función, utiliza punteros y especifica claramente que esa variable será modificada.
  - Ejemplo:

```
void calcularPuntaje(int* puntaje_total, int vidas_usadas) {
    // Se modifica el valor de puntaje_total
}
```

- **Constantes en parámetros**: Usa `const` para indicar que un parámetro no será modificado dentro de la función.
  - Ejemplo:

```
void mostrarMenu(const tConfig* configuraciones);
```

8. Modularidad

- **Modularidad**: Cada función debe hacer una sola cosa. Divide las tareas complejas en varias funciones más pequeñas y reutilizables.
- **Archivos de cabecera (`.h`)**: Las declaraciones de las funciones y las definiciones de tipos de datos (estructuras, constantes) deben estar en archivos de cabecera. Los archivos `.c` solo deben contener la implementación de las funciones.

9. Espaciado y Separación

- Deja una línea en blanco entre bloques lógicos de código para mejorar la legibilidad.
- No coloques múltiples sentencias en la misma línea.

10. Compilación y Warnings

- **Compilación sin warnings**: El código debe compilar sin advertencias (`-Wall` en GCC).
- **Revisión del código**: Antes de enviar una parte del proyecto, asegúrate de que el código esté limpio y sin errores.

11. Nombrado de archivos

- Los archivos `.c` y `.h` deben tener nombres descriptivos relacionados con su contenido.
  - Ejemplo:
    - `configuracion.c`, `configuracion.h` para la gestión de la configuración del juego.
    - `secuencia.c`, `secuencia.h` para las funciones relacionadas con las secuencias de colores.