```
/*-------------------------------------------------------------*/
/* Leer nombre de usuario */
leer_nombre_usuario:
    push {r1, r2, r7, lr}
    ldr r1, =msg_nombre
    bl imprimir_string
    mov r7, #3
    mov r0, #0
    ldr r1, =nombre_usuario
    mov r2, #49
    swi 0
    ldr r1, =nombre_usuario
    mov r2, #0
buscar_newline:
    ldrb r3, [r1, r2]
    cmp r3, #10
    beq eliminar_newline
    cmp r3, #0
    beq fin_leer_nombre
    add r2, #1
    cmp r2, #49
    blt buscar_newline
    b fin_leer_nombre
eliminar_newline:
    mov r3, #0
    strb r3, [r1, r2]
fin_leer_nombre:
    pop {r1, r2, r7, lr}
    bx lr

/*-------------------------------
/* Imprimir nombre de usuario */
imprimir_nombre_usuario:
    push {r0-r2, r7, lr}
    ldr r1, =nombre_usuario
    mov r2, #0
contar_chars:
    ldrb r0, [r1, r2]
    cmp r0, #0
    beq fin_contar
    add r2, #1
    cmp r2, #49
    blt contar_chars
fin_contar:
    mov r7, #4
    mov r0, #1
    ldr r1, =nombre_usuario
    swi 0
    pop {r0-r2, r7, lr}
    bx lr

/*-------------------------------
/* Mostrar tiempo final de jueg
mostrar_tiempo_final_color:
    push {r0-r3, lr}
    mov r3, r2          // r2=color (puntero)
    ldr r1, [sp, #16]  // backup de lr (por si acaso)
    ldr r1, =color_reset
    // Imprime color recibido
    mov r1, r3
    bl imprimir_string
    ldr r1, =msg_tiempo_juego
    bl imprimir_string
    bl imprimir_nombre_usuario
    ldr r1, =msg_dos_puntos
    bl imprimir_string
    bl calcular_tiempo_transcurrido
    bl imprimir_numero
    ldr r1, =msg_tiempo_segundos
    bl imprimir_string
    ldr r1, =msg_gracias
```

Organización del Computador

Trabajo Práctico – ARM: Buscaminas

Profesores:

- **Laura Almada**
- **Ignacio Aranda Bao**
- **Marcelo Castellon**

Alumnos:

- **Cordua Kevin** - (kevincordu2003@gmail.com)
- **Laste German** - (german.laste777@gmail.com)

Comisión 1 - 2025

Universidad Nacional de General Sarmiento

Informe de Desarrollo: Buscaminas en Entorno ARM

Este informe describe el proceso de diseño e implementación de una versión del clásico juego Buscaminas, desarrollada íntegramente en lenguaje ensamblador **ARM** como trabajo final para la materia **Organización del Computador**. El objetivo principal fue profundizar en la comprensión de la arquitectura **ARM** y el manejo de bajo nivel de la memoria, la entrada/salida y la lógica de control, enfrentando los desafíos propios de programar sin las abstracciones de los lenguajes de alto nivel.

El juego presenta una interfaz de usuario basada en caracteres **ASCII**, permitiendo la interacción mediante el ingreso de coordenadas (filas y columnas) a través del teclado. Se brindó especial atención a la experiencia del usuario, incorporando mensajes claros, colores para destacar información relevante y validaciones para evitar errores de ingreso.

Entre las funcionalidades implementadas se destacan:

- Generación aleatoria de minas y gestión eficiente de matrices en memoria.
- Algoritmo de descubrimiento en cascada (flood fill) para casillas vacías.
- Registro de nombre, tiempo y nivel del jugador en un archivo externo (**ranking.txt**) en caso de victoria, y visualización de los últimos ganadores.
- Retroalimentación continua sobre el progreso y el estado del juego.

Este trabajo tuvo como objetivos principales:

- Profundizar en la comprensión del manejo de memoria, entrada/salida y control de flujo a bajo nivel.
- Implementar una interfaz de usuario interactiva basada en texto, con validaciones y mensajes claros.
- Desarrollar rutinas eficientes para la gestión de matrices, generación aleatoria y algoritmos clásicos como **flood fill**.
- Registrar y mostrar información relevante del jugador (nombre, tiempo, nivel) y mantener un ranking persistente.
- Optimizar el código para lograr claridad, robustez y facilidad de mantenimiento, preparando la base para posibles mejoras futuras.

Este proyecto no solo permitió afianzar conceptos teóricos de la materia, sino que también representó un desafío práctico en la resolución de problemas reales de programación a bajo nivel, optimización y manejo de recursos limitados.

Ubicación del Código Fuente:
El código fuente del programa se encuentra disponible en la siguiente ruta en el dispositivo remoto: **/home/orga2021/occ1g2/TrabajoPractico/**

Pseudocódigo:

```
INICIO
   // Inicialización de variables y bienvenida
   generar_semilla_aleatoria()
   mostrar_mensaje_bienvenida()
   leer_nombre_usuario()

   // Selección de tamaño de tablero (si aplica)
   REPETIR
      mostrar_opciones_tamaño()
      leer_opcion_tamaño()
      SI opcion_válida ENTONCES
         configurar_tamaño_tablero()
      SINO
         mostrar_error_opcion_invalida()
      FIN_SI
   HASTA opcion_válida

   // Selección de dificultad
   REPETIR
      mostrar_opciones_dificultad()
      leer_opcion_dificultad()
      SI opcion_válida ENTONCES
         configurar_nivel_dificultad()
      SINO
         mostrar_error_opcion_invalida()
      FIN_SI
   HASTA opcion_válida

   // Preparar matrices y variables de juego
   limpiar_matrices()
   generar_minas_aleatorias()
   calcular_objetivo_descubrir()
   es_primera_jugada = VERDADERO

   mostrar_tablero()
   mostrar_progreso()

   // Bucle principal del juego
   MIENTRAS jugadas_realizadas < objetivo_descubrir
      // Leer y validar coordenadas
      REPETIR
         leer_fila()
         SI fila_válida ENTONCES
            leer_columna()
            SI columna_válida ENTONCES
               SALIR_DEL_BUCLE
            SINO
               mostrar_error_coordenadas()
            FIN_SI

         SINO
            mostrar_error_coordenadas()
         FIN_SI
      HASTA fila_y_columna_válidas

      // Validar jugada
      SI casilla_ya_descubierta ENTONCES
         mostrar_error_casilla_repetida()
         CONTINUAR
      FIN_SI
```

```
      SI hay_mina_en_posicion ENTONCES
         IR_A perder
      FIN_SI

      minas_cercanas = calcular_minas_cercanas(fila, columna)

      SI es_primera_jugada Y minas_cercanas == 0 ENTONCES
         flood_fill(fila, columna)
         es_primera_jugada = FALSO
      SINO
         marcar_casilla_descubierta(fila, columna)
         marcar_en_tablero(fila, columna, minas_cercanas)
      FIN_SI

      mostrar_mensaje_seguro()
      mostrar_tablero()
      mostrar_progreso()
   FIN_MIENTRAS

   // Fin del juego
   SI jugadas_realizadas >= objetivo_descubrir ENTONCES
      IR_A ganar
   FIN_SI

ganar:
   tiempo_fin = obtener_tiempo_actual()
   mostrar_mensaje_victoria()
   calcular_y_mostrar_tiempo_transcurrido()
   guardar_en_ranking()
   mostrar_ranking()
   terminar_programa()

perder:
   tiempo_fin = obtener_tiempo_actual()
   mostrar_mensaje_derrota()
   mostrar_todas_las_minas()
   mostrar_tablero_final()
   calcular_y_mostrar_tiempo_transcurrido()
   terminar_programa()
FIN
```

Implementación del modo de prueba (modo_test):

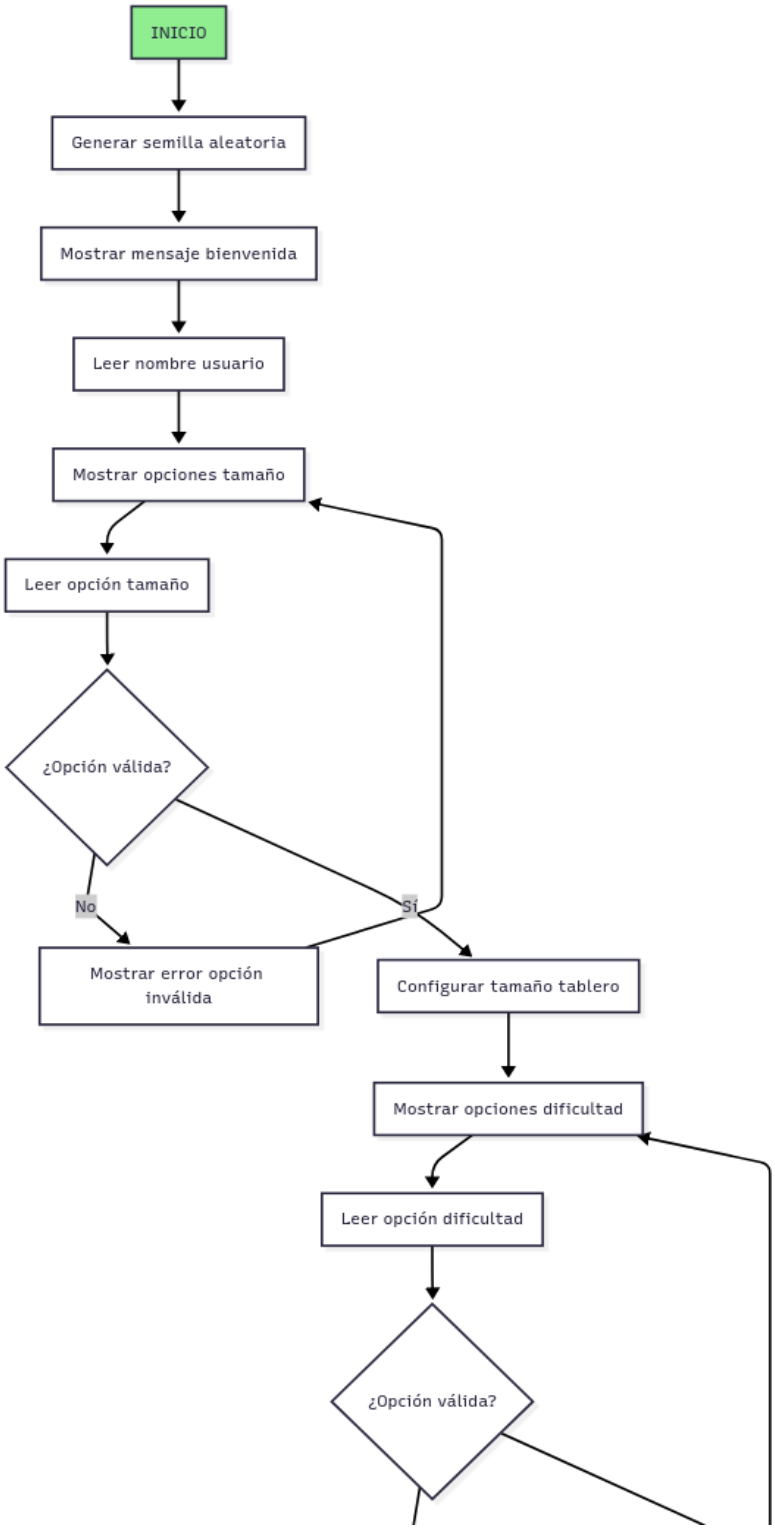
Durante el desarrollo del proyecto, incorporamos una variable denominada **modo_test** con el objetivo de facilitar la verificación y depuración del funcionamiento del juego.

Cuando **modo_test** está activado **(modo_test: .word 1)**, el programa ajusta automáticamente la cantidad de casillas a descubrir para que la partida pueda finalizar en una sola jugada exitosa.

Esto nos permitió comprobar rápidamente el flujo de victoria, el registro en el ranking y la correcta actualización de los mensajes y archivos, sin necesidad de jugar una partida completa cada vez que realizábamos una modificación.

Esta funcionalidad fue utilizada exclusivamente durante la etapa de pruebas y no afecta el comportamiento normal del juego para el usuario final.

Diagrama Lógico de la Estrategia de Diseño del Juego



Inicio del programa

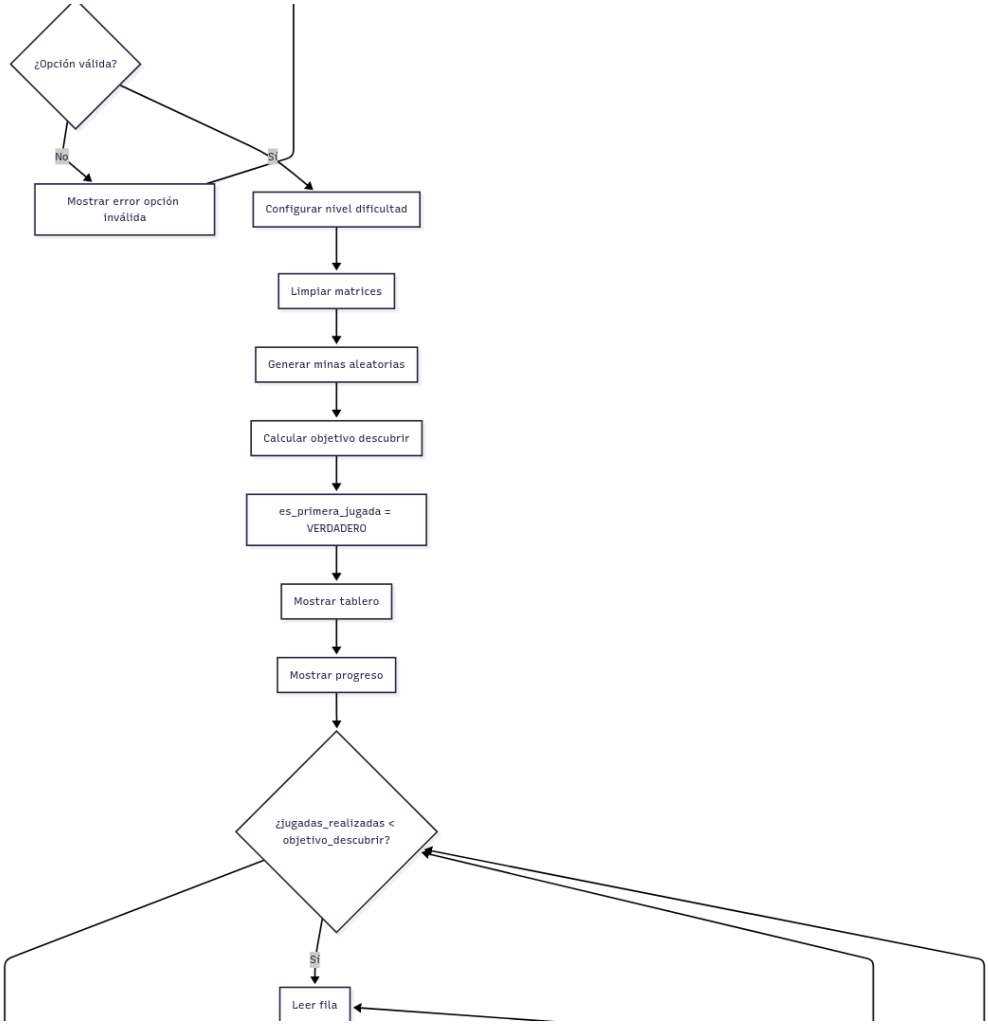
- Se genera una semilla aleatoria.
- Se presenta un mensaje de bienvenida al usuario.
- Se solicita y registra el nombre del usuario para personalizar la experiencia.

Elección del tamaño del tablero (falta implementar):

- Se exhiben las opciones disponibles para el tamaño del tablero.
- El usuario selecciona e ingresa una opción.
- El sistema valida la opción ingresada:
    - Si es válida: se establece la configuración del tamaño del tablero.
    - Si no es válida: se emite un mensaje de error y se reitera el proceso.

Elección de la dificultad:

- Se presentan las opciones de dificultad, se procesa la entrada del usuario y se realiza la validación.
- En caso de una entrada incorrecta, el sistema retorna al paso previo.



Inicio del juego:

1. Si la opción de dificultad es válida:
    - Se **configura el nivel de dificultad**
    - Luego, se **limpian las matrices** del juego.
    - Se **generan minas aleatorias** en el tablero.

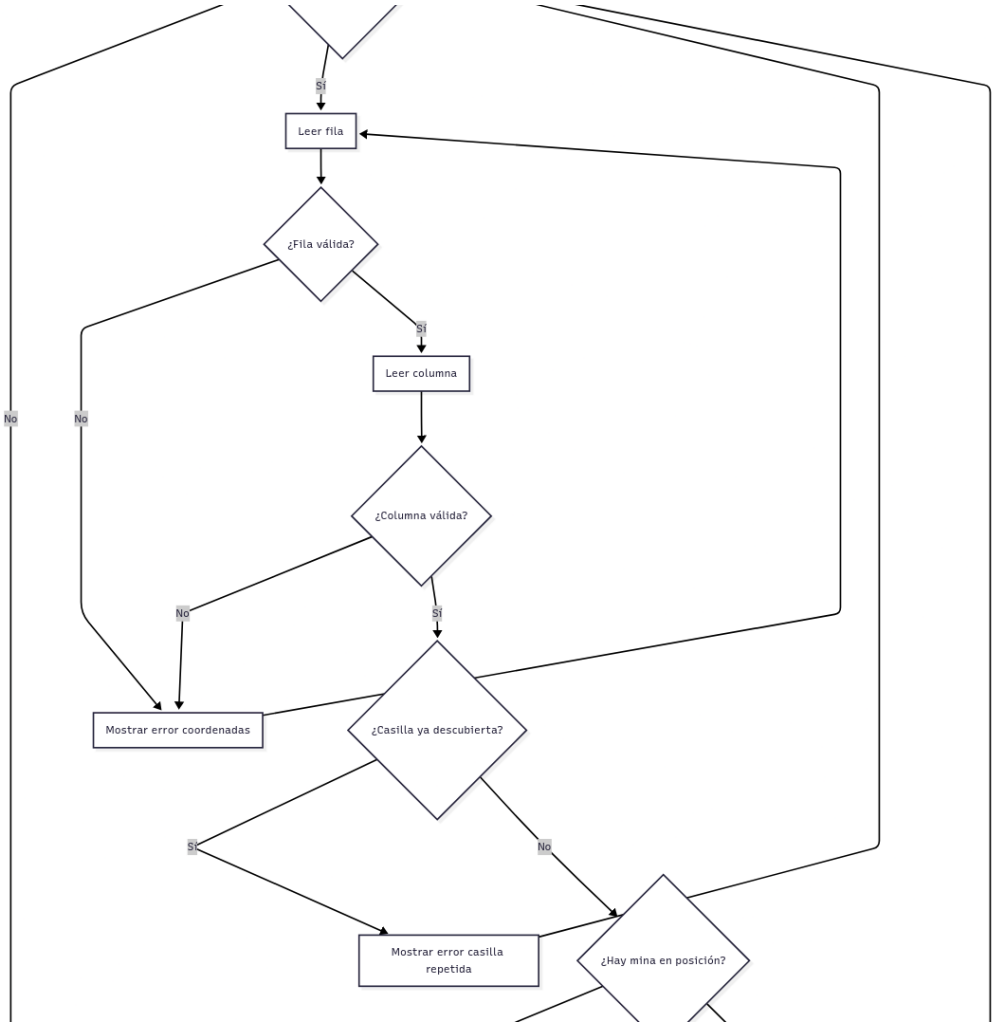
- Se **calcula el objetivo a descubrir**, es decir, cuántas casillas libres (sin minas) el jugador necesita descubrir para ganar.

2. Se inicializa una variable:
   - **es_primera_jugada = VERDADERO**: útil para el flood_fill

3. Se muestra el **tablero actual** y luego el **progreso del jugador**

Bucle de jugadas:

4. Se entra a un ciclo que se repite **mientras las jugadas realizadas sean menores al objetivo**:
   - Se verifica: **¿jugadas_realizadas < objetivo_descubrir?**
     - Sí sí: se continúa el juego leyendo una jugada (en el siguiente paso del diagrama).
     - Si **no**: el jugador ha descubierto todas las casillas necesarias y el juego terminaría con victoria

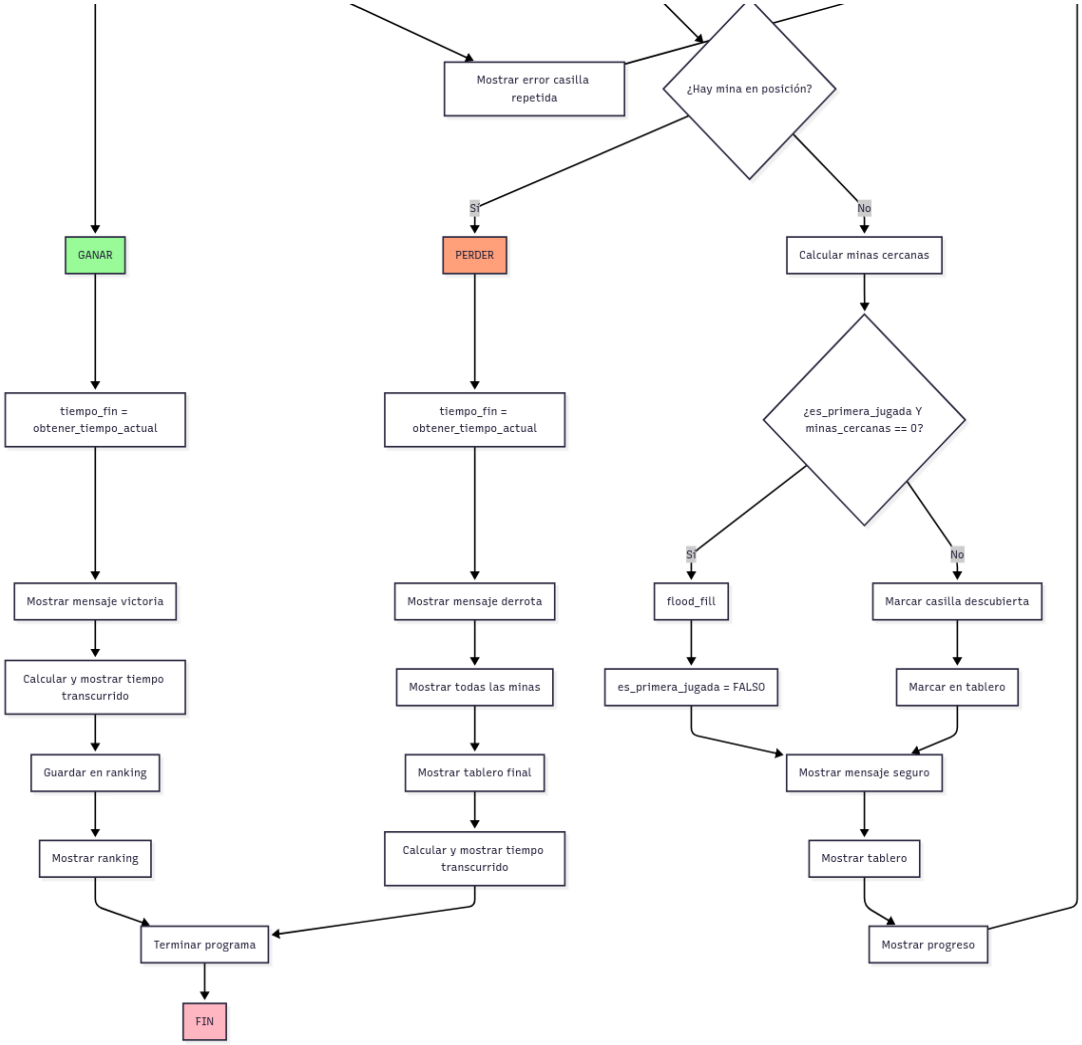


Ingreso de coordenadas y validación:

1. El jugador **ingresa la fila** de la jugada.
2. Se valida: **¿Fila válida?**
    - **No válida** → Se muestra un mensaje de **error de coordenadas** y se vuelve a pedir.
    - **Válida** → Se continúa.
3. El jugador **ingresa la columna**.
4. Se valida: **¿Columna válida?**
    - **No válida** → Se muestra error de coordenadas.
    - **Válida** → Se continúa.

Revisión del estado de la casilla:

5. Se verifica: **¿Casilla ya descubierta?**
    - **Sí** → Se muestra un **error por casilla repetida** (el jugador eligió una posición que ya fue descubierta antes).
    - **No** → Se sigue con la lógica del juego.



¿Pisa una mina?

Después de validar la jugada:

- Se evalúa: **¿Hay mina en posición?**
    - **Sí** → el jugador **pierde**:
        - Se guarda el tiempo de finalización.
        - Se muestra un mensaje de **derrota**.
        - Se revelan todas las minas en el tablero.
        - Se muestra el tablero final.
        - Se calcula el **tiempo transcurrido**.
        - El juego termina.

¿No pisa una mina?

- Se calculan las **minas cercanas** a la casilla.
- Se evalúa una condición especial:
  ¿Es la primera jugada y minas_cercanas == 0?
  - **Sí**:
    - Se realiza un **flood fill** (descubrimiento automático en cadena de casillas vacías).
    - Se cambia la variable **es_primera_jugada** a **FALSO**.
  - **No**:
    - Se **marca la casilla descubierta** y se actualiza el tablero con la jugada.

En ambos casos:

- Se muestra un mensaje de que la casilla es segura.
- Se muestra el **tablero actualizado**.
- Se muestra el **progreso**.
- Se regresa al bucle principal de jugadas.

¿Se alcanza el objetivo?

- Si el número de **jugadas realizadas** alcanza el **objetivo** de casillas descubiertas:
  - El jugador **gana**:
    - Se guarda el tiempo final.
    - Se muestra mensaje de **victoria**.
    - Se calcula y muestra el **tiempo total jugado**.
    - Se guarda la puntuación en un **ranking**.
    - Se muestra el **ranking**.
    - El programa termina.

Dificultades encontradas y cómo las superamos

Gestión de matrices y acceso a memoria

Dificultad: En ensamblador, no existen estructuras de datos de alto nivel como matrices o arreglos multidimensionales. Todo debe manejarse como bloques lineales de memoria. Esto complica el acceso a elementos, ya que hay que calcular manualmente la posición de cada casilla usando la fórmula **índice = fila * tamanio_tablero + columna**. Además, cualquier error en el cálculo puede provocar que se acceda a posiciones inválidas, generando bugs difíciles de detectar.

¿Cómo lo superamos?
Definimos variables para el tamaño del tablero **(tamanio_tablero)** y utilizamos subrutinas para calcular los índices de acceso a las matrices de minas y casillas descubiertas. Esto permitió centralizar la lógica y reducir errores. Además, agregamos validaciones para evitar accesos fuera de rango, mejorando la robustez del código.

```
verificar_mina_simple:
    push {r2, r3, r4, lr}
    mov r2, #8
    mul r3, r0, r2
    add r3, r3, r1
    ldr r4, =minas
    ldrb r0, [r4, r3]
    pop {r2, r3, r4, lr}
    bx lr
```

Este patrón se repite en otras subrutinas como **verificar_casilla_repetida** y **marcar_casilla_descubierta**.

Inicialización y visualización del tablero

Dificultad: Mantener sincronizados el tablero visual (lo que ve el usuario) y las matrices lógicas (minas y descubiertas) fue un reto. En particular, actualizar el tablero visual después de cada jugada y asegurarse de que refleje correctamente el estado interno del juego requiere mucho cuidado, ya que cualquier desincronización puede confundir al usuario.

¿Cómo lo superamos?
Separando la lógica del tablero visual de la lógica de juego. Creamos subrutinas específicas para marcar casillas en el tablero visual **(marcar_tablero)** y para mostrar el tablero completo **(mostrar_tablero)**. Así, cada vez que se actualiza el estado de una casilla, se actualiza también el tablero visual de forma controlada. Además, inicializamos el tablero visual con puntos y saltos de línea, y lo actualizamos con números, 'O' o '*' según corresponda.

Ejemplo ficticio y breve para no ocupar mucho espacio en el informe:

```
// Marca una casilla en el tablero visual
mov r3, #9
mul r4, fila, r3
add r4, r4, columna
strb simbolo, [tablero, r4]
```

Generación de minas aleatorias sin repeticiones

Dificultad: Colocar minas de forma aleatoria sin que se repitan posiciones es un problema clásico, pero en ensamblador no hay estructuras como listas o sets para verificar fácilmente si una posición ya tiene una mina. Además, la generación de números aleatorios debe ser eficiente y segura.

¿Cómo lo superamos?
Utilizamos una rutina que, para cada mina a colocar, genera coordenadas aleatorias y verifica si ya hay una mina en esa posición usando la subrutina **verificar_mina_simple**. Si la posición ya está ocupada, se repite la generación hasta encontrar una libre. Esto garantiza que no haya repeticiones, aunque puede ser ineficiente si hay muchas minas, pero es suficiente para tableros pequeños.

Ejemplo:

```
colocar_minas_loop:
    cmp r8, r6
    bge fin_generar_minas
    bl generar_aleatorio
    and r0, r0, #7
    mov r4, r0
    bl generar_aleatorio
    and r0, r0, #7
    mov r5, r0
    mov r0, r4
    mov r1, r5
    bl verificar_mina_simple
    cmp r0, #1
    beq colocar_minas_loop
    mov r2, #8
    mul r3, r4, r2
    add r3, r3, r5
    ldr r7, =minas
    mov r0, #1
    strb r0, [r7, r3]
    add r8, r8, #1
    b colocar_minas_loop
```

Validación y manejo de entradas del usuario

Dificultad: El usuario puede ingresar datos inválidos (letras, números fuera de rango, etc.), lo que puede romper la lógica del juego o provocar accesos fuera de los límites de las matrices. Además, en ensamblador, el manejo de cadenas y conversión de caracteres a números es más laborioso que en lenguajes de alto nivel.

¿Cómo lo superamos?
Implementamos una rutina de lectura de números **(leer_numero)** que valida que la entrada sea numérica y esté dentro del rango permitido. Si la entrada es inválida, se muestra un mensaje de error y se solicita nuevamente la entrada. Esto evita errores y mejora la experiencia del usuario.

Ejemplo ficticio y breve para no ocupar mucho espacio en el informe:

```
// Valida que la entrada sea un número entre 0 y 7
cmp entrada, #0
blt invalido
cmp entrada, #7
bgt invalido
```

Implementación del algoritmo flood fill

Dificultad: El flood fill es un algoritmo recursivo que descubre todas las casillas vacías conectadas. En ensamblador, la recursión es peligrosa porque la pila es limitada y no hay protección contra desbordamientos. Además, controlar las condiciones de corte y evitar bucles infinitos es más difícil.

¿Cómo lo superamos?
Implementamos el flood fill como una subrutina recursiva, pero limitando la expansión solo a las 4 direcciones cardinales (arriba, abajo, izquierda, derecha) y agregando condiciones estrictas de corte: no expandir fuera de los límites del tablero, no expandir sobre casillas ya descubiertas y detenerse si hay minas cercanas. Además, deshabilitamos el flood fill después de la primera jugada para evitar problemas de rendimiento y pila.

Ejemplo ficticio y breve para no ocupar mucho espacio en el informe:

```
// Expande solo si la casilla está vacía y dentro de los límites
cmp fila, #0
blt fin
cmp fila, #8
bge fin
// ... (verifica y expande en 4 direcciones)
```

Gestión de archivos para el ranking

Dificultad: Guardar y leer el ranking de ganadores implica trabajar con archivos desde ensamblador, lo cual requiere usar **syscalls** y manejar buffers manualmente. Además, hay que armar las líneas de texto y asegurarse de no sobrescribir datos.

¿Cómo lo superamos?
Creamos subrutinas específicas para armar las líneas de ranking y para leer/escribir archivos usando las **syscalls** correspondientes. Utilizamos buffers temporales y controlamos cuidadosamente los **offsets** y los tamaños para evitar errores de escritura o lectura.

Ejemplo ficticio y breve para no ocupar mucho espacio en el informe:

```
// Escribe una línea en el archivo de ranking
mov r7, #5      // syscall open
swi 0
mov fd, r0
mov r7, #4      // syscall write
swi 0
mov r7, #6      // syscall close
swi 0
```

Manejo de colores y mensajes en la terminal

Dificultad: Mostrar mensajes en colores y con formato requiere enviar secuencias ANSI a la terminal, lo que puede ser confuso y difícil de depurar en ensamblador.

¿Cómo lo superamos?
Definimos los códigos de color como cadenas en la sección de datos y creamos subrutinas para imprimirlas junto con los mensajes. Así, pudimos mejorar la experiencia visual del usuario y hacer el juego más atractivo.

Ejemplo ficticio y breve para no ocupar mucho espacio en el informe:

```
// Imprime un mensaje en color
ldr r1, =color_verde
bl imprimir_string
ldr r1, =mensaje
bl imprimir_string
ldr r1, =color_reset
bl imprimir_string
```

Para cerrar dicho informe, consideramos oportuno destacar los siguientes puntos sobre la realización de este proyecto:

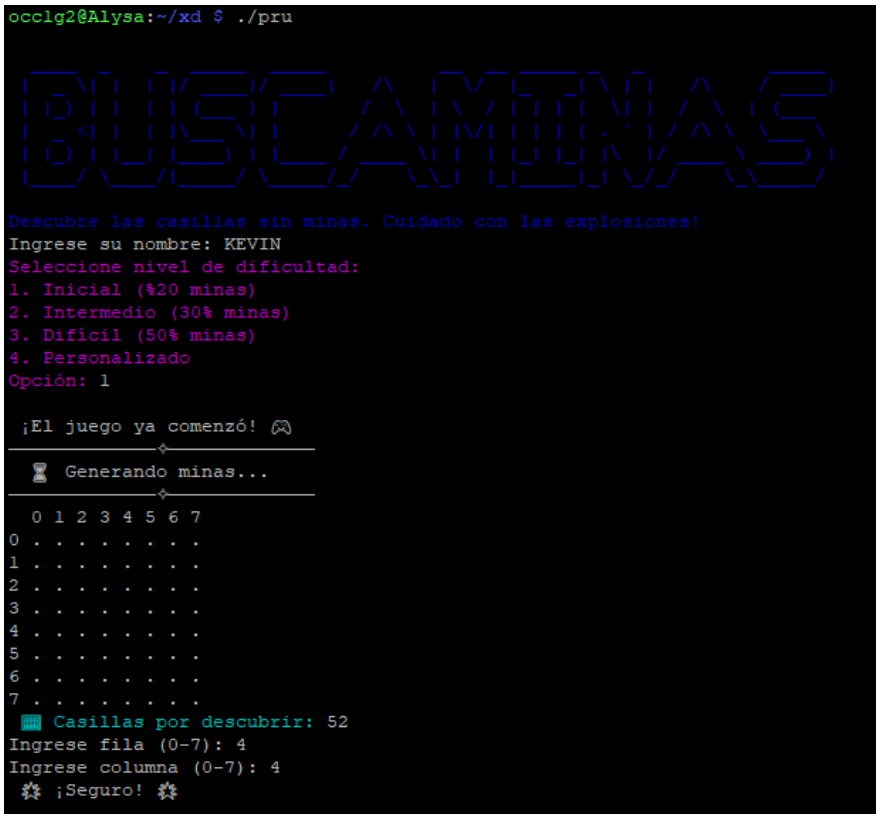
- **Aprendizaje:** Trabajar en ensamblador nos permitió comprender en profundidad cómo se gestionan la memoria y los datos a bajo nivel, así como la importancia de la optimización y la claridad en la estructura del código.

- **Modularidad:** Separar la lógica en subrutinas (mostrar tablero, leer número, flood fill, etc.) facilitó el desarrollo, la depuración y la futura ampliación del juego.

- **Preparación para el futuro:** Aunque el juego está implementado para un tablero 8x8, estructuramos el código para soportar fácilmente tableros de tamaño variable, demostrando buenas prácticas de programación y previsión para futuras mejoras.
.
- **Desafío grupal:** El mayor reto fue adaptar algoritmos clásicos (como flood fill y la generación de minas) a un entorno tan limitado como el ensamblador, donde cada instrucción cuenta y los errores pueden ser difíciles de rastrear.

- **Satisfacción:** Ver el juego funcionando correctamente, con una interfaz amigable y lógica robusta, resultó muy gratificante y nos brindó confianza para abordar proyectos más complejos en bajo nivel.

Imágenes de nuestro trabajo:



```
Ingrese fila (0-7): 4
Ingrese columna (0-7): 1
 ¡Seguro!

  0 1 2 3 4 5 6 7
0 . . . . . . . .
1 1 . . . . . . .
2 0 1 . . . . . .
3 0 0 1 2 . . . .
4 0 0 0 0 2 . . .
5 0 1 1 0 0 2 . .
6 2 . . 1 1 . . .
7 . . . . . . . .
 Casillas por descubrir: 31
```

```
Ingrese fila (0-7): 4
Ingrese columna (0-7): 4
 ¡Seguro!

  0 1 2 3 4 5 6 7
0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0
2 1 . 1 0 0 0 0 0
3 0 1 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0 0
 Casillas por descubrir: 0

 ¡FELICITACIONES! GANASTE

 Tiempo de juego para KEVIN: 4 seg

 RANKING GANADORES

Nombre: KEVIN | Tiempo: 2 seg | Nivel: 4
Nombre: KEVIN | Tiempo: 4 seg | Nivel: 4
Nombre: KEVIN | Tiempo: 4 seg | Nivel: 4

==========

+---------------------------------------------+
|                                             |
|        ¡Gracias por jugar!                  |
|      Desarrollado por Grupo 02              |
|   https://github.com/Corduu/BuscaminasARM   |
|                                             |
+---------------------------------------------+
```

```
Ingrese fila (0-7): 4
Ingrese columna (0-7): 4

 ¡BOOM!
 Has perdido.

Las minas estaban en:
  0 1 2 3 4 5 6 7
0 . * * * . * . .
1 . . . . . * * .
2 * . . . * * * *
3 * * . . . * . *
4 . . . * * . * *
5 * * . . . . * .
6 . . * . . . * .
7 . . * * * * * *

 Tiempo de juego para KEVIN: 6 seg

+---------------------------------------------+
|                                             |
|        ¡Gracias por jugar!                  |
|      Desarrollado por Grupo 02              |
|   https://github.com/Corduu/BuscaminasARM   |
|                                             |
+---------------------------------------------+
```