

Manual de prácticas de *Computadors*

Fase 1

CandyNDS 1

Versión de texto

2º curso de Grado en Ingeniería Informática

Universitat Rovira i Virgili

Profesor responsable:	Santiago Romaní (santiago.romani@urv.cat)
Profesores de prácticas:	Pere Millán, Cristina Romero, Carlos Soriano, Víctor Navas

Tabla de contenidos

1 Descripción general de la fase 1	3
1.1 Funcionamiento básico	3
1.2 Estructura del proyecto	6
1.3 Tareas de la fase 1	6
1.4 Distribución de tareas.....	8
1.5 Dependencias entre rutinas	9
1.6 Planificación temporal de las tareas.....	10
1.7 Valoración de las tareas.....	11
1.8 Evaluación de las prácticas de Computadores	12
2 Descripción detallada de la fase 1	14
2.1 Ficheros de definiciones.....	14
2.2 Fichero de configuración	15
2.3 El programa principal.....	17
2.4 Directrices para el desarrollo de las rutinas	18
2.5 Tarea 1A: inicializa_matriz()	21
2.6 Tarea 1B: recombina_elementos()	23
2.7 Tarea 1C: hay_secuencia()	26
2.8 Tarea 1D: elimina_secuencias()	28
2.9 Tarea 1E: cuenta_repeticiones().....	31
2.10 Tarea 1F: baja_elementos().....	32
2.11 Tarea 1G: hay_combinacion().....	36
2.12 Tarea 1H: sugerir_combinacion()	40
3 Depuración de programas en la NDS	45
3.1 Depuración de código escrito en lenguaje C.....	45
3.2 Depuración de código escrito en lenguaje ensamblador	47
3.3 Problemas con el Insight.....	53

1 Descripción general de la fase 1

1.1 Funcionamiento básico

Se propone implementar una versión reducida del juego **Candy Crush**, basado en conseguir secuencias de 3 o más elementos del mismo tipo en horizontal o en vertical (ref. [Shariki](#)).

En esta primera fase de la práctica se realizará una versión del juego en modo texto: el tablero de juego se definirá con una matriz de 9x9 casillas, donde cada casilla contendrá un número que representará uno de los siguientes objetos:

1..6	elementos básicos	habitualmente, cada casilla de la matriz contendrá un elemento de uno de los seis tipos disponibles, representado por un número del 1 al 6,
0	casilla vacía	cuando se consiguen secuencias de 3 o más elementos del mismo tipo, dichos elementos se eliminan del tablero poniendo sus casillas a cero, lo cual puede provocar la caída de elementos desde las casillas superiores,
7	bloque sólido	los bloques sólidos no se pueden mover y no permiten la bajada de los elementos de las casillas superiores,
15	hueco	los huecos no se pueden mover, pero sí permiten bajar elementos de casillas superiores, a través de ellos,
9..14	gelatinas simples	son elementos atrapados por una gelatina que se “rompe” cuando se elimina el elemento básico (al generar una secuencia); se representan con la numeración de los tipos básicos +8,
17..22	gelatinas dobles	son elementos atrapados por una doble gelatina que se “rompe” cuando se elimina el elemento básico, pero la gelatina doble se convierte en una gelatina simple; se representan con la numeración de los tipos básicos +16.

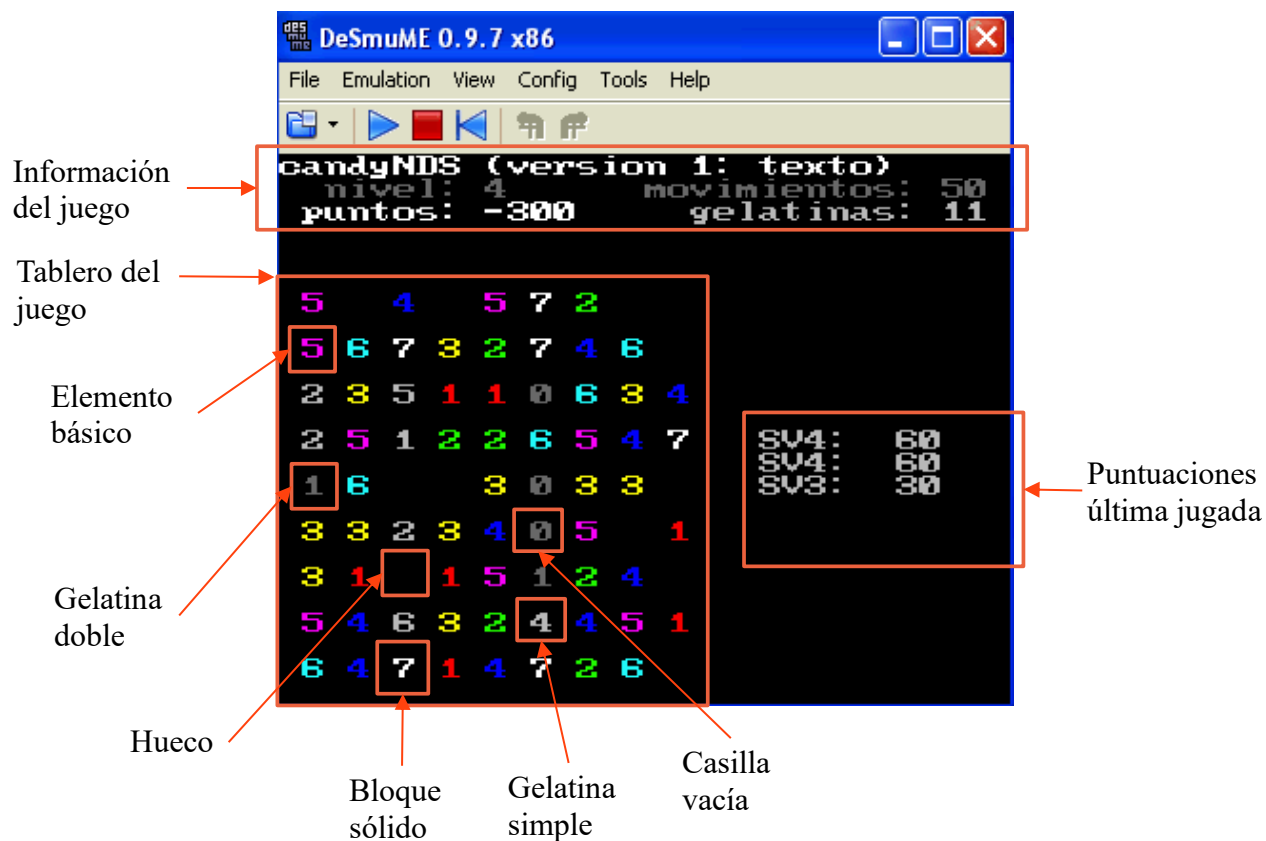
La representación de la matriz en modo texto se realiza con la función `escribe_matriz()` que, junto a otras funciones ya implementadas, ofrecen una visualización por pantalla similar a la siguiente:

```
candyNDS (version 1: texto)
nivel: 4      movimientos: 50
puntos: -300  gelatinas: 11

5  4  5 7 2
5 6 7 3 2 7 4 6
2 3 5 1 1 0 6 3 4
2 5 1 2 2 6 5 4 7
1 6      3 0 3 3
3 3 2 3 4 0 5 1
3 1      1 5 1 2 4
5 4 6 3 2 4 4 5 1
6 4 7 1 4 7 2 6

SV4: 60
SV4: 60
SV3: 30
```

La función `escribe_matriz()` utiliza diversos colores para resaltar los diferentes tipos de elementos, los bloques sólidos y las gelatinas. Las casillas vacías muestran un cero, mientras que los huecos no visualizan ningún carácter.

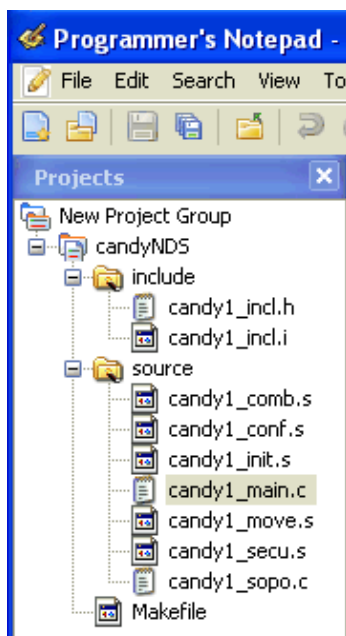


La dinámica del juego se puede resumir en los siguientes puntos:

Inicialización	se configura el tablero de juego según unos mapas de configuración establecidos para cada nivel, generando elementos aleatorios sobre las casillas vacías (evitando secuencias iniciales)
Jugadas	usando el puntero de pantalla, el usuario podrá intercambiar dos elementos colindantes en horizontal o en vertical, si la nueva disposición provoca una secuencia de 3 o más elementos del mismo tipo
Caída	cuando se eliminan los elementos de una secuencia, los elementos de las casillas superiores caen hasta rellenar las casillas que han quedado vacías, excepto si hay bloques sólidos por medio; se crearán nuevos elementos aleatorios para la casilla vacía más alta de cada columna; las casillas con gelatinas permiten que sus elementos bajen a casillas inferiores, pero la característica de gelatina no se mueve
Puntuación	según la combinación de secuencias se obtendrán ciertas puntuaciones, que se sumarán al contador de puntos; para superar cada nivel hay que conseguir que dicho contador sea cero o positivo (inicialmente será negativo o cero)
Gelatinas	cuando una gelatina se “rompe”, el contador de gelatinas se decrementa en una unidad (cada gelatina doble corresponde a 2 gelatinas simples); para superar cada nivel hay que conseguir que dicho contador sea 0
Movimientos	cada nivel tiene un contador de movimientos, que se decrementa con cada jugada; hay que conseguir superar cada nivel (puntos ≥ 0 y gelatinas = 0) antes de que el contador de movimientos llegue a cero; de otro modo, se tendrá que repetir el nivel
Sugerencias	cada cierto tiempo sin que el usuario realice un movimiento (8 segundos, aprox.), el programa realizará una sugerencia, ocultando y mostrando intermitentemente 3 de los elementos que se puedan combinar
Recombinación	se puede llegar a distribuciones de elementos en el tablero que no permitan realizar ninguna secuencia; en tal caso, el programa debe reorganizar aleatoriamente todos los elementos presentes en el tablero hasta conseguir una nueva disposición que permita realizar al menos una secuencia

1.2 Estructura del proyecto

Para facilitar el trabajo se proporciona una estructura específica del proyecto, con los siguientes ficheros:



candy1_incl.h: definiciones de constantes, variables y funciones externas, en C

candy1_incl.i: definiciones de constantes, en GAS (GNU assembler)

candy1_main.c: programa principal de control del juego (ya implementado)

candy1_sopo.c: funciones de soporte al programa principal (ya implementadas)

candy1_conf.s: variables globales de configuración del juego (ya implementadas)

candy1_init.s: rutinas para inicializar y recombinar la matriz de juego

candy1_secu.s: rutinas para detectar y eliminar secuencias

candy1_move.s: rutinas para contar repeticiones y bajar elementos

candy1_comb.s: rutinas para detectar y sugerir combinaciones

1.3 Tareas de la fase 1

Como se puede observar, algunas partes del proyecto ya están implementadas. La mayor parte del trabajo a realizar se localiza en los 4 últimos ficheros, que albergarán diferentes rutinas en lenguaje ensamblador para realizar determinadas tareas del juego.

A continuación se listan dichas tareas, junto con una breve descripción de las rutinas involucradas, que se detallarán más adelante en este mismo manual:

candy1_init.s

Tarea 1A	<code>inicializa_matriz(*matriz, num_mapa)</code> rutina para inicializar la matriz de juego a partir del número de mapa de configuración indicado por parámetro
Tarea 1B	<code>recombina_elementos(*matriz)</code> rutina para generar una nueva matriz de juego mediante la reubicación de los elementos de la matriz original, para crear nuevas jugadas

candy1_secu.s

Tarea 1C	<code>hay_secuencia(*matriz)</code> rutina para detectar si existe, por lo menos, una secuencia de 3 elementos iguales consecutivos, en horizontal o en vertical
Tarea 1D	<code>elimina_secuencias(*matriz, *marcas)</code> rutina para eliminar todas las secuencias de 3 o más elementos del mismo tipo en horizontal, vertical o cruzadas

candy1_move.s

Tarea 1E	<code>cuenta_repeticiones(*matriz, f, c, ori)</code> rutina para contar el número de repeticiones del elemento situado en la posición (f,c) de la matriz, visitando las siguientes posiciones según la orientación que indique el parámetro <code>ori</code>
Tarea 1F	<code>baja_elementos(*matriz)</code> rutina para bajar elementos hacia las posiciones vacías, primero en vertical y después en diagonal

candy1_comb.s

Tarea 1G	<code>hay_combinacion(*matriz)</code> rutina para detectar si existe, por lo menos, una combinación entre dos elementos consecutivos que provoquen una secuencia válida
Tarea 1H	<code>sugiere_combinacion(*matriz, *sug)</code> rutina para detectar alguna combinación entre dos elementos consecutivos que provoquen una secuencia válida, y devolver (por referencia) las coordenadas de las tres posiciones que pueden dar lugar a dicha secuencia

Todas las rutinas se escribirán en el lenguaje ensamblador GAS para procesadores ARM (v5).

Estas rutinas implementan la lógica del juego. En la segunda fase de la práctica se implementarán las rutinas necesarias para la visualización gráfica del juego.

1.4 Distribución de tareas

La implementación de todas las tareas se repartirá entre los componentes de cada grupo de prácticas, atendiendo a los siguientes requisitos:

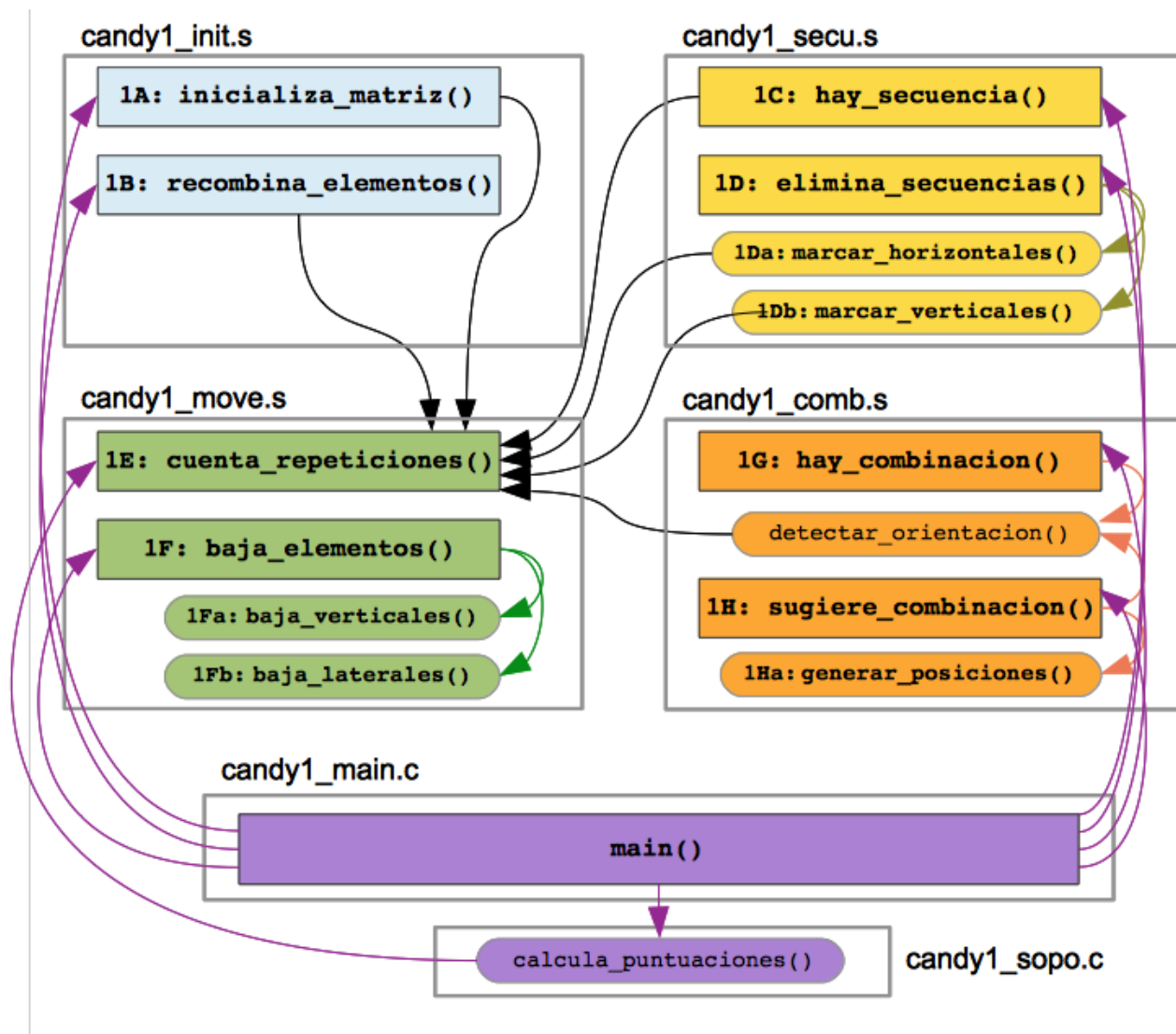
- Los grupos de prácticas podrán ser de 1 a 4 personas, que se identificarán como **progl**, **prog2**, **prog3** y **prog4**.
- Cada tarea tiene una valoración diferente, según su grado de dificultad (ver apartado sobre valoración de las tareas).
- Cada programador tendrá que realizar, como mínimo, 2 tareas.
- En principio, a cada programador se le asignarán las 2 tareas de un mismo fichero, pero esto no será un requisito obligatorio, es decir, es posible realizar tareas de distintos ficheros.
- Sin embargo, las tareas a realizar no se pueden escoger solo entre las más simples de cada fichero, es decir, cada programador deberá presentar, al menos, una tarea simple (**1A**, **1C**, **1E** o **1G**) y una tarea compleja (**1B**, **1D**, **1F** o **1H**).
- Los grupos de menos de 4 programadores podrán presentar una versión incompleta de la práctica, aunque esto implicará una limitación de la nota máxima que podrán obtener (ver apartado sobre valoración de las tareas).

Además, hay que tener en cuenta que las rutinas escritas por todos los programadores se tendrán que unir en **una única versión definitiva** del proyecto. Este trabajo de fusión se tendrá que realizar conjuntamente entre todos los componentes del grupo.

1.5 Dependencias entre rutinas

Algunas rutinas necesitarán invocar otras subrutinas de soporte que tendrán que ser implementadas por el mismo programador que tenga asignada la tarea correspondiente. Sin embargo, la rutina `cuenta_repeticiones()` se tiene que invocar desde otras tareas asignadas a distintos programadores. Por este motivo, será necesario que dicha rutina se complete **lo antes posible** (ver el apartado de planificación temporal de tareas).

El siguiente gráfico muestra un mapa de dependencias entre rutinas:



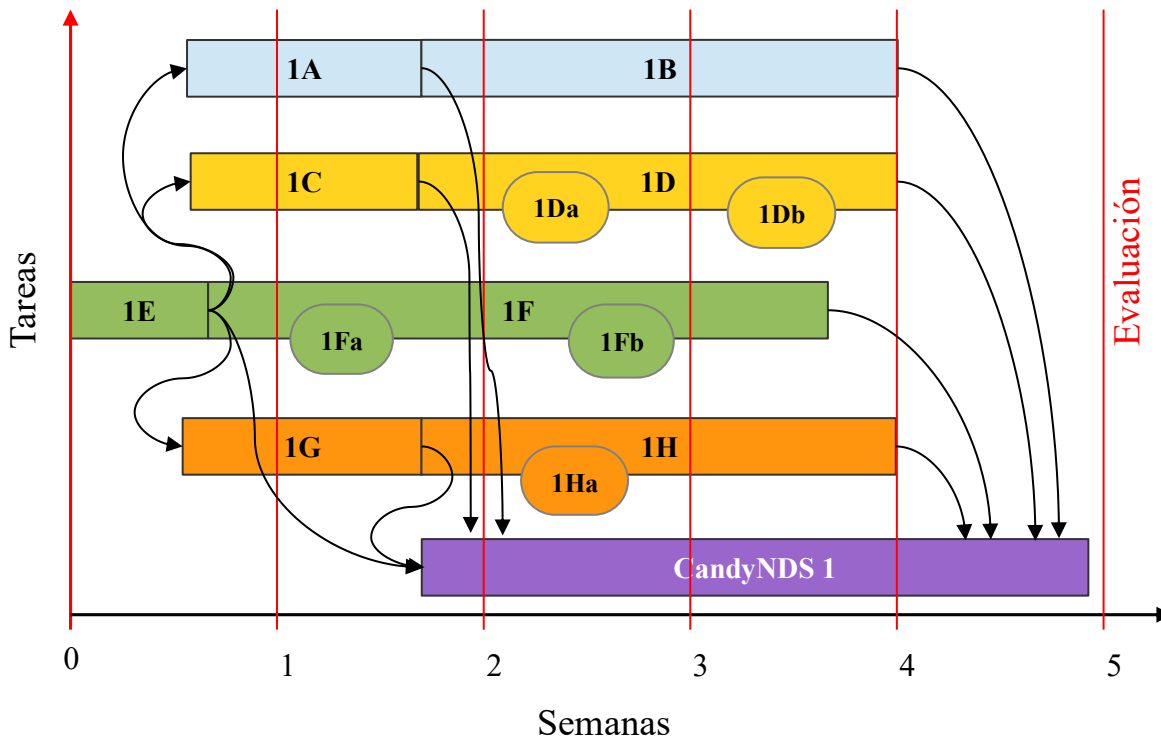
Las subrutinas de soporte se han etiquetado añadiendo una letra minúscula 'a' o 'b' detrás del nombre de la tarea de la cual dependen, a excepción de `detectar_orientacion()`, que ya está implementada.

Además, en el gráfico se muestran también las dependencias con el programa principal y la función de soporte `calcula_puntuaciones()`, lo cual indica la necesidad de fusionar todas las tareas para completar la versión definitiva del proyecto.

1.6 Planificación temporal de las tareas

Para acabar las tareas a tiempo para la evaluación en primera convocatoria, es imprescindible ceñirse al máximo a una planificación temporal.

A continuación se propone una planificación para las 5 semanas de que se dispone, desde que se explica la práctica (inicio del curso) hasta que se realiza la evaluación:



Las flechas que salen de las tareas indican la necesidad de incorporar ese código en las otras tareas, o sea, son las dependencias directas.

La tarea **CandyNDS 1** representa la versión completa del proyecto, sobre la cual hay que ir incorporando el código generado por todos los programadores.

Se hace evidente, por tanto, la necesidad de **acabar las tareas de las rutinas antes de la cuarta semana**, con el fin de tener tiempo suficiente para ajustar la integración de todas las rutinas dentro de la versión definitiva de la fase 1 del proyecto, lo cual **NO** es un proceso trivial en absoluto: es muy probable que las rutinas individuales funcionen correctamente, pero al realizar la fusión surjan errores debidos a interacciones no previstas en el funcionamiento conjunto de las rutinas integradas.

1.7 Valoración de las tareas

A continuación se muestra una tabla con la valoración máxima de cada tarea, agrupadas por ficheros:

candy1_init.s	1A: 4 puntos	1B: 7 puntos
candy1_secu.s	1C: 3 puntos	1D: 8 puntos
candy1_move.s	1E: 3 puntos	1F: 8 puntos
candy1_comb.s	1G: 4 puntos	1H: 7 puntos

La suma de los puntos máximos por cada fichero es 11 (sobre 10), aunque la nota final vendrá limitada por el número de ficheros que se presenten fusionados en la versión definitiva, según la siguiente tabla:

1 fichero	máx. 7 puntos
2 ficheros	máx. 9 puntos
3 ficheros	máx. 10 puntos
4 ficheros	máx. 11 puntos

La limitación del máximo número de puntos está pensada para permitir presentar proyectos incompletos, al mismo tiempo que tiene en cuenta la **menor** dificultad para fusionar menos rutinas del proyecto. De este modo se pueden formar grupos de menos de 4 alumnos.

Sin embargo, **no se contabilizarán los ficheros acabados pero no fusionados**. Es decir, si se presentan 3 ficheros acabados, 2 fusionados y uno no, los alumnos que hayan fusionado sus códigos podrán llegar a 9 puntos, mientras que el alumno que presente su código por separado solo podrá llegar a 7 puntos.

Los grupos de prácticas compuestos por menos de 4 alumnos también pueden optar a máximos superiores si completan (y fusionan) tareas adicionales. Por ejemplo, un grupo de 2 alumnos que presente las tareas (fusionadas) de 3 ficheros podrá optar a un máximo de 10 puntos.

Así mismo, si alguno de los componentes de un grupo **no** cumple con su trabajo, el resto de los componentes pueden asumir sus tareas, con lo cual podrán optar a la nota máxima correspondiente al número de ficheros fusionados, además de acumular los puntos de las tareas extra que realicen (hasta el máximo correspondiente). En este caso, el alumno que no realice una tarea se quedará sin los puntos correspondientes.

Esta observación es especialmente importante para el alumno que tenga asignada la tarea **1E**, puesto que debe terminarla con suficiente antelación para que el resto de los compañeros puedan realizar sus códigos. Un plazo máximo razonable sería **2 semanas desde el inicio del curso**. Si dicho alumno no cumple con este plazo, alguno de sus compañeros podrá asumir la realización de la tarea **1E** y subirla al repositorio remoto, lo cual le dará derecho a quedarse con los puntos de dicha tarea.

Se pide, además, que para cada tarea implementada se suministre un programa de test específico de las rutinas correspondientes. Dentro del fichero **Compus.zip**, disponible en el Campus Virtual (*Moodle*) de la asignatura, se proporciona un fichero denominado **code_tests_1.txt** que contiene ejemplos de programas de test para las tareas simples **1A**, **1C**, **1E** y **1G**. Cada programa de test incluye una función `main()` específica, diferente de la función `main()` final del juego, además de diversos juegos de pruebas en forma de tableros iniciales de juego.

Además, cada programador deberá ampliar la función `main()` de test proporcionada, junto con su correspondiente juego de pruebas, para probar el funcionamiento de la segunda tarea compleja que tenga asignada (**1B**, **1D**, **1F** o **1H**). En caso de **no** proporcionar dicha función `main()` con su juego de pruebas adjunto, **la práctica se considerará suspendida**.

1.8 Evaluación de las prácticas de Computadores

Aunque este manual está dedicado a explicar la primera fase de la práctica, a continuación se detallan los requisitos para aprobar las dos fases en las dos convocatorias disponibles:

- La nota de cada práctica es **individual** por alumno, y **no** depende exclusivamente del correcto funcionamiento del programa, sino que vendrá **significativamente** determinada por la calidad de las respuestas a las preguntas que el profesor de prácticas realizará durante la entrevista de evaluación.
- No se requiere presentar informe.
- Las entregas de las dos partes de la práctica son **independientes**, es decir, si no se entrega la fase 1 en la fecha establecida, **no** se evaluarán esas tareas en la fecha de entrega de la fase 2, aunque para la 2ª entrega ya estén terminadas.
- La nota final de prácticas será la **media aritmética** de las notas de las fases 1 y 2. Sin embargo, para poder aprobar la asignatura es necesario obtener una nota superior o igual a 4 (sobre 10) en cada una de las dos fases.

- Las condiciones de presentación de las dos fases de la práctica en segunda convocatoria son exactamente las mismas que en primera convocatoria.
- Se guardarán notas parciales de cada una de las dos fases de la práctica entre convocatorias, y hasta un máximo de dos cursos académicos, con las reducciones que se establecen en el documento de presentación de la asignatura disponible en el *Moodle*.

Además de estas consideraciones, las dos fases de la práctica se tendrán que presentar **obligatoriamente** a través del servidor **Git** del DEIM. Esto implica nuevas restricciones a la hora de presentar las dos fases de la práctica:

- para evaluar las tareas que ha realizado cada programador, el profesor corrector solo tendrá en cuenta el contenido de los *commits* que dicho programador haya subido al servidor dentro del tiempo límite, es decir, se ignorarán los ficheros y *commits* locales en el ordenador del alumno, aunque los hubiera terminado a tiempo,
- se considerará como versión definitiva del proyecto al último *commit* de la rama **master** del servidor, subido dentro del tiempo límite; las fusiones de los ficheros se tendrán que realizar como *merges* de las ramas de los programadores sobre la rama **master**,
- el programa de test de las tareas de cada programador se deberá subir en un *commit* específico en la rama del programador, con su correspondiente mensaje identificativo, por ejemplo, 'TEST 1A+1B'; los *commits* de test deben contener código fuente que compile y funcione directamente, sin tener que realizar ningún cambio durante la entrevista de evaluación (ni descomentar código, ni cambiar nombre de la función `main()`),
- los *commits* de test **no** deben contener código de la función `main()` final del juego (ni siquiera comentado), puesto que dicho código final ya estará registrado en la versión fusionada del proyecto,
- en el *commit* final del proyecto, todo el código de test debe ser **eliminado** (no comentado), puesto que ya estará registrado en los *commits* de test.

Nota: en la segunda y tercera sesión de laboratorio se explicará el funcionamiento del sistema de control de versiones Git.

2 Descripción detallada de la fase 1

2.1 Ficheros de definiciones

El primer fichero a considerar es `candy1_incl.i`:

```
@;=== candy1_incl.i: definiciones comunes para ficheros en ensamblador ===

@; Rango de los números de filas y de columnas -> mínimo: 3, máximo: 11
ROWS = 9
COLUMNS = 9
```

Este pequeño fichero establece las dimensiones actuales (filas, columnas) del tablero de juego para el resto de los ficheros de código en lenguaje ensamblador. Por ejemplo, la primera línea efectiva (no comentario) del fichero `candy1_init.s` es la siguiente:

```
.include "../include/candy1_incl.i"
```

Esta línea es una directiva de ensamblador que inserta el fichero `candy1_incl.i` dentro del propio fichero `candy1_init.s`. Esta técnica permite definir los dos símbolos `ROWS` y `COLUMNS` en un solo lugar (fichero `.i`) y utilizarlos en cualquier otro fichero `.s` que los necesite.

Para conseguir un código fuente fácilmente adaptable se deben utilizar símbolos en vez de números fijos para referirse a valores constantes. En el caso de esta práctica, las dimensiones de la matriz de juego van a ser diferentes en la segunda fase, de modo que, si utilizáramos números fijos, el proceso de cambio sería mucho más costoso:

Bien	Mal
<code>cmp r2, #COLUMNS</code>	<code>cmp r2, #9</code>

El fichero de cabeceras `candy1_incl.h` define las dimensiones del tablero de juego para los ficheros escritos en lenguaje C, así como otras definiciones de constantes, variables y funciones externas:

```
/*-----

$Id: candy1_incl.h $

Definiciones externas en C para la versión 1 del juego (modo texto)
```

```

----- */

// Rango de los números de filas y de columnas:
// mínimo: 3, máximo: 11
#define ROWS      9                // dimensiones de la matriz de juego
#define COLUMNS  9
#define DFIL      (24-ROWS*2)     // desplazamiento vertical de filas

#define MAXLEVEL  9                // nivel máximo (niveles 0..MAXLEVEL-1)
...

```

2.2 Fichero de configuración

El fichero `candy1_conf.s` define ciertas variables globales que permiten configurar cada nivel de juego, por ejemplo, el número máximo de movimientos y los puntos a conseguir (en negativo):

```

...
    .global max_mov
max_mov:    .byte 20, 27, 31, 45, 52, 32, 21, 90, 50
...

    .align 2
    .global pun_obj
pun_obj:    .word -1000, -830, -500, 0, -240, -500, -200, -900, 0
...

```

A continuación, se definen los mapas de configuración de la matriz de juego para cada nivel. Por ejemplo, el mapa proporcionado para el nivel 4 es el siguiente:

```

...
    .global mapas
mapas:
...
    @; mapa 4: gelatinas dobles
    .byte 0,15,0,15,0,7,0,15,15
    .byte 0,0,7,0,0,7,0,0,15
    .byte 10,3,8,1,1,8,3,3,0
    .byte 10,1,9,0,0,20,3,4,7
    .byte 17,2,15,15,3,19,4,3,15
    .byte 3,2,10,0,0,20,0,15,0
    .byte 2,3,15,0,0,16,0,0,15
    .byte 0,0,8,0,0,8,0,0,0
    .byte 0,4,7,0,0,7,0,0,15
...

```

Este mapa contiene ejemplos de todos los tipos de valores de configuración que podemos introducir, los cuales determinarán el contenido inicial de cada posición de la matriz de juego:

0	casilla vacía	el programa deberá generar un tipo de elemento (número del 1 al 6) de forma aleatoria, para crear un nuevo elemento
1-6	elementos básicos prefijados	se copiarán directamente en la matriz de juego
7	bloque sólido	se copiará directamente en la matriz de juego
8	gelatina simple vacía	el programa deberá generar un tipo de elemento (número del 1 al 6) de forma aleatoria, que sumado a 8 creará un nuevo elemento con gelatina simple
9-14	gelatinas simples prefijadas	se copiarán directamente en la matriz de juego
15	hueco	se copiará directamente en la matriz de juego
16	gelatina doble vacía	el programa deberá generar un tipo de elemento (número del 1 al 6) de forma aleatoria, que sumado a 16 creará un nuevo elemento con gelatina doble
17-22	gelatinas dobles prefijadas	se copiarán directamente en la matriz de juego

Con estos mapas se pueden definir diferentes configuraciones iniciales (aleatorias o no) para poder probar diferentes aspectos de las rutinas a implementar, es decir, nos permiten definir diferentes **juegos de pruebas**.

Atención: se exigirá a los alumnos que modifiquen los mapas de configuración proporcionados inicialmente a modo de ejemplo, para que aprendan a crear sus propios juegos de pruebas.

2.3 El programa principal

El fichero `candy1_main.c` contiene el código del programa principal que controla todo el juego. Como es bastante largo, a continuación se muestra un esquema para poder comprender mejor su estructura:

```
int main(void)
{
    ...                // definición de variables locales e inicializaciones

    do                  // bucle principal del juego
    {
        if (initializing)      //// SECCIÓN DE INICIALIZACIÓN ////
        {
            // inicializar matriz
            // comprobar y eliminar secuencias iniciales

        }
        else if (falling)      //// SECCIÓN BAJADA DE ELEMENTOS ////
        {
            // bajar elementos (1 posición)
            // si final de bajada, comprobar y eliminar secuencias

        }
        else                  //// SECCIÓN DE JUGADAS ////
        {
            // detectar jugada (touchscreen)
            // comprobar y eliminar secuencias

        }
        if (!falling)          //// SECCIÓN DE DEPURACIÓN ////
        {
            // detectar teclas 'B'/'START' para cambio manual de nivel

        }
        if (change)            //// SECCIÓN CAMBIO DE NIVEL ////
        {
            // detectar fin movimientos o combinaciones u objetivos
            // si no hay combinación, recombina elementos

        }
        if (lapse >= 192)      //// SECCIÓN DE SUGERENCIAS ////
        {
            // si tiempo > 8 segundos, sugiere combinación
            // parpadeo de elementos sugeridos

        }
    }
    while (1);              // bucle infinito

    return(0);              // nunca retornará del main
}
```

Básicamente, se trata de un bucle infinito que, en cada iteración, ejecuta una o varias secciones del código, según el estado de diversas variables de control (*initializing*, *falling*, *change* y *lapse*).

Se han coloreado las distintas partes de las secciones según las tareas que están involucradas. De este modo, **cada programador debe eliminar las partes del programa principal que no invocan a sus rutinas**, para definir sus propios programas de test de sus rutinas. Solo cuando se realice la fusión de todas las tareas de todos los programadores se podrá incluir todo el código del programa principal proporcionado.

2.4 Directrices para el desarrollo de las rutinas

En el apartado 1.5 de este manual se exponen las dependencias debidas a llamadas explícitas entre rutinas. Sin embargo, a medida que se desarrolle la práctica aparecerán otras dependencias indirectas debidas a la necesidad de fusionar y probar el código en su conjunto. Algunas de estas dependencias se pueden observar en el esquema del programa principal expuesto en el apartado anterior.

En la sección de inicialización, por ejemplo, no se puede comprobar si hay secuencias iniciales (tarea 1C) si no se realiza previamente la inicialización de la matriz (tarea 1A). En este caso, para evitar tener que esperar a que el programador correspondiente termine la tarea 1A, el programador que se encargue de la tarea 1C puede modificar el programa principal para llamar a la función de soporte *copia_mapa()* en vez de llamar a *inicializa_matriz()*, de modo que pueda copiar directamente el contenido de un mapa de configuración (especificado por parámetro) dentro de la matriz de juego. La invocación desde la sección de inicialización podría ser como sigue:

```
...
do                                // bucle principal del juego
{
    if (initializing)             //// SECCIÓN DE INICIALIZACIÓN ////
    {
        //inicializa_matriz(matrix, level);
        copia_mapa(matrix, level);
        escribe_matriz(matrix);
    }
    ...

    if (hay_secuencia(matrix))    // si hay secuencias
    {
        ...
    }
}
```

De este modo **no** se generan elementos aleatorios, pero se pueden introducir elementos fijos manualmente en distintos mapas de configuración específicos (diferentes de los proporcionados por los profesores) para probar la rutina `hay_secuencia()`, sin tener que esperar a que la rutina `inicializa_matriz()` esté terminada.

El resto de las rutinas también se tendrán que probar utilizando mapas de configuración específicos y modificando adecuadamente el programa principal. Éste es el procedimiento habitual de desarrollo de cualquier proyecto informático complejo; **nunca** hay que pretender escribir el código de todas las rutinas para probarlo todo a la vez, sino que **hay que ir probando (a fondo) cada rutina por separado**.

Otra directriz que facilitará las llamadas entre diferentes rutinas escritas en lenguaje ensamblador consiste en acordar un convenio común a la hora de utilizar los registros que albergan los parámetros más habituales de las rutinas.

Concretamente, adoptaremos el convenio de utilizar el registro `R0` para pasar la dirección base de la matriz que representará el tablero de juego, los registros `R1` y `R2` para pasar la fila y la columna de una posición de la matriz, y el registro `R3` para pasar algún otro parámetro de la rutina. Por ejemplo, en la cabecera de la rutina `cuenta_repeticiones()` se lee lo siguiente:

```
@;   Parámetros:
@;       R0 = dirección base de la matriz
@;       R1 = fila 'f'
@;       R2 = columna 'c'
@;       R3 = orientación 'ori' (0 -> Este, 1 -> Sur, 2 -> Oeste, 3 -> Norte)
```

Por lo tanto, en las rutinas que recorran toda la matriz será una buena práctica utilizar los registros `R1` y `R2` como índices de filas y columnas. Además, se recomienda copiar la dirección base de la matriz sobre `R4`, ya que el registro `R0` se utiliza habitualmente para retornar el resultado de la llamada a otras rutinas, y utilizar otro registro (como el `R6`) como desplazamiento lineal de las posiciones de la matriz. De este modo, un bucle para recorrer todos los elementos de la matriz podría tener la siguiente estructura:

```
...
    mov r4, r0           @;R4 es dirección base de la matriz de juego
    mov r6, #0           @;R6 es desplazamiento de las posiciones
    mov r1, #0           @;R1 es índice de fila
.L_for1:
    mov r2, #0           @;R2 es índice de columna
.L_for2:
    ldrb r5, [r4, r6]     @;R5 = valor de la casilla (R1,R2)

    ...                  @;procesamiento del elemento actual
```

```

add r6, #1           @;avanza desplazamiento de posiciones
add r2, #1
cmp r2, #COLUMNS   @;recorre todas las columnas
blo .L_for2

add r1, #1
cmp r1, #ROWS        @;recorre todas las filas
blo .L_for1
...

```

En el código anterior, el registro **R5** captura el código del objeto (espacio vacío, elemento básico, bloque sólido, hueco, gelatina) contenido en la posición (**R1**, **R2**). Para procesarlo de manera eficiente, hay que tener en cuenta que la codificación de los posibles objetos está basada en la separación de información a nivel de bits:

	bits 4..3 = 00	bits 4..3 = 01	bits 4..3 = 10
bits 2..0 = 000	00 000 → espacio vacío	01 000 → gel.s. vacía	10 000 → gel.d. vacía
bits 2..0 = 001	00 001 → elem. tipo 1	01 001 → gel.s. tipo 1	10 001 → gel.d. tipo 1
010	00 010 → elem. tipo 2	01 010 → gel.s. tipo 2	10 010 → gel.d. tipo 2
...
110	00 110 → elem. tipo 6	01 110 → gel.s. tipo 6	10 110 → gel.d. tipo 6
bits 2..0 = 111	00 111 → bloque sólido	01 111 → hueco	10 111 → no utilizado

De este modo, para detectar determinadas condiciones sobre el código de los objetos **se exigirá** trabajar con instrucciones de lenguaje máquina que permitan operar sobre los dos grupos de bits **usando máscaras de bits**. Las siguientes instrucciones son ejemplos de cómo procesar los campos de bits:

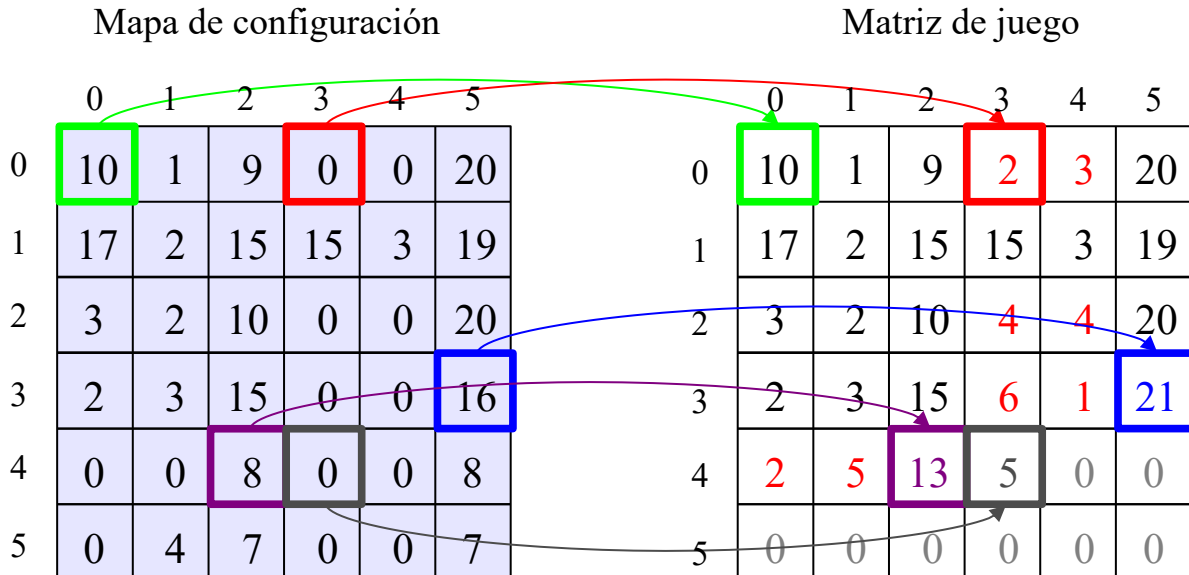
and r5, #0x07	R5 = bits 2..0
mov r9, r5, lsr #3 and r9, #0x03	R9 = bits 4..3
tst r5, #0x07	FZ = 1 si bits 2..0 = 000
etc.	

2.5 Tarea 1A: *inicializa_matriz()*

La rutina de inicialización de la matriz recibe dos parámetros: la dirección base de la matriz (**R0**) y el número de mapa de configuración (**R1**). Se propone el siguiente algoritmo:

- 1 recorrer todas las posiciones del mapa de configuración especificado por parámetro,
- 2 si una posición contiene un objeto fijo (1-6, 7, 9-14, 15, 17-22), copiarlo directamente en la matriz de juego,
- 3 si una posición contiene un objeto variable (0, 8, 16), generar un número aleatorio del 1 al 6 con la rutina `mod_random()`, sumarlo al código base y copiar el resultado en la matriz de juego,
- 4 el paso anterior se debe repetir mientras se detecte alguna secuencia de 3 elementos del mismo tipo consecutivos, en horizontal o en vertical, que termine sobre la posición actual, para lo cual se debe llamar a la rutina `cuenta_repeticiones()` con las orientaciones Oeste (2) y Norte (3).

El siguiente gráfico ejemplifica estas acciones:



- (0, 0): la línea verde es una copia de un valor literal (gelatina simple tipo 2),
- (0, 3): la línea roja es una generación aleatoria de un elemento básico,
- (3, 5): la línea azul es una generación aleatoria de una gelatina doble,
- (4, 2): la línea morada es una generación aleatoria de una gelatina simple.

Además, la línea gris que sale de la casilla (4, 3) representa la posición que se está calculando actualmente, lo cual se puede deducir porque las posiciones siguientes de la matriz de juego todavía están a cero. En este caso, la rutina de inicialización debe generar un elemento básico aleatoriamente, pero debe verificar que las dos casillas de la izquierda o las dos casillas superiores no formen una secuencia inicial de 3 elementos del mismo tipo. Si suponemos que aleatoriamente se ha generado un elemento de tipo 5, y se realizan las comprobaciones llamando a la rutina `cuenta_repeticiones()` se obtendría lo siguiente:

	1	2	3
2			4
3			6
4	5	13	5

\updownarrow `cuenta_repeticiones(mat, 4, 3, 3) = 1`
 \longleftrightarrow `cuenta_repeticiones(mat, 4, 3, 2) = 3`

En el caso de la dirección vertical no hay problema, pero en el caso horizontal hay una secuencia inicial de 3 elementos de tipo 5, puesto que el elemento de la casilla (4, 2) es una gelatina simple del mismo tipo (8+5). En este caso, el programa debe generar otro número aleatorio y volver a realizar las comprobaciones oportunas para evitar secuencias iniciales en la matriz de juego.

Sin embargo, cuando se copian literalmente los elementos del mapa de configuración **no** se deben realizar comprobaciones de secuencias iniciales. Esto puede provocar casos como el de la casilla (2, 5):

	3	4	5
0			20
1			19
2	4	4	20

\updownarrow `cuenta_repeticiones(mat, 2, 5, 3) = 1`
 \longleftrightarrow `cuenta_repeticiones(mat, 2, 5, 2) = 3`

En este caso, el código 20 (gelatina doble de tipo 4) se ha copiado literalmente del mapa de configuración, lo cual provoca una secuencia horizontal de 3 elementos del

mismo tipo. Los 2 cuatros anteriores se generaron aleatoriamente, pero en ese momento no se detectó ninguna secuencia inicial.

Además, puede que el mapa de configuración contenga secuencias iniciales utilizando elementos literales, con el fin de realizar pruebas específicas de detección y eliminación de secuencias (ver mapas de configuración 5 y 6, en el fichero `candy1_conf.s`).

Es decir, la rutina `inicializa_matriz()` solo debe evitar secuencias iniciales cuando genera elementos aleatorios, pero no cuando copia literalmente códigos literales del mapa de configuración.

2.6 Tarea 1B: *recombina_elementos()*

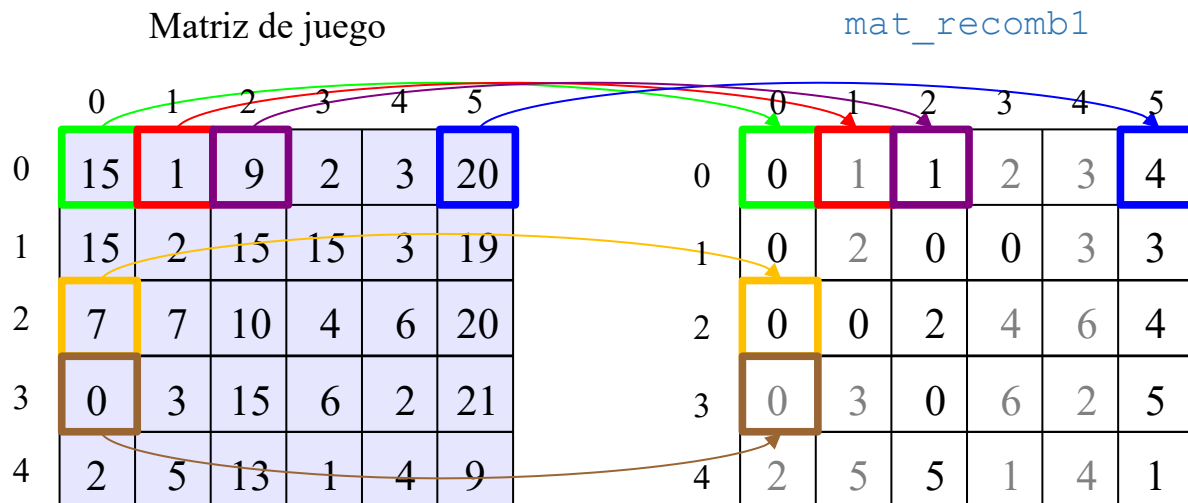
La rutina de recombinación de los elementos de la matriz recibe un solo parámetro, la dirección base de la matriz (`R0`). Se propone el siguiente algoritmo:

1. copiar todo el contenido actual de la matriz de juego sobre la variable global `mat_recomb1`, pero transformando los bloques sólidos (7), huecos (15) y gelatinas vacías (8, 16) en ceros, y los elementos con gelatina simple (9-14) o doble (17-22) a su correspondiente código básico de elemento (1-6); de esta forma tendremos en `mat_recomb1` todos los códigos básicos de elemento y el resto de las casillas a cero,
2. copiar todo el contenido actual de la matriz de juego sobre la variable global `mat_recomb2`, pero transformando los elementos básicos (1-6) en ceros, y las casillas con gelatina (8-14, 16-22) a su correspondiente código básico de gelatina (8 o 16); de esta forma tendremos en `mat_recomb2` todos los códigos de bloque sólido, hueco y gelatina (sin elemento), y el resto de las casillas a cero,
3. recorrer linealmente (filas / columnas) todas las posiciones de la matriz de juego,
4. si la posición actual en la matriz de juego es un espacio vacío (0, 8 o 16), un bloque sólido o un hueco, ignorar dicha posición,
5. en caso contrario, seleccionar una posición aleatoria de `mat_recomb1` que contenga un código de elemento (valor $\neq 0$),

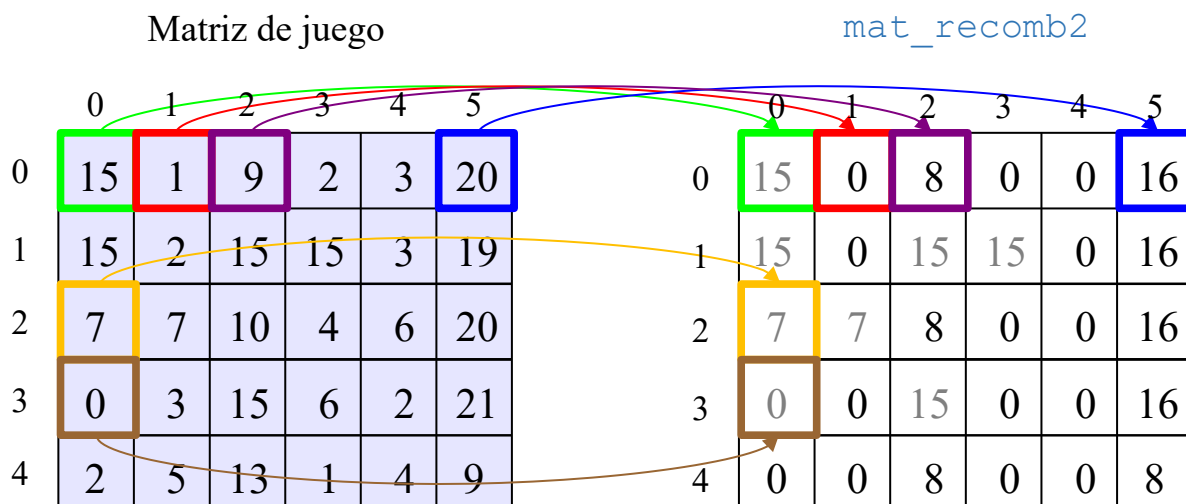
6. añadir el código de elemento obtenido de `mat_recomb1` a la posición actual en `mat_recomb2`, sumándole el posible código de gelatina guardado en el paso 2, y comprobar que no genera ninguna secuencia horizontal ni vertical; en caso de que haya secuencia, restituir el valor anterior de la posición actual en `mat_recomb2` y volver al paso 5,
7. cuando se consiga un código válido para la posición actual en `mat_recomb2`, fijar a cero la posición de `mat_recomb1` donde se ha obtenido el código de elemento, para evitar reutilizar dicho código en las siguientes posiciones de `mat_recomb2`,
8. una vez generada la nueva matriz de juego en `mat_recomb2`, copiar todo su contenido sobre la matriz de juego pasada por referencia.

Los siguientes gráficos ejemplifican algunas de las acciones anteriores.

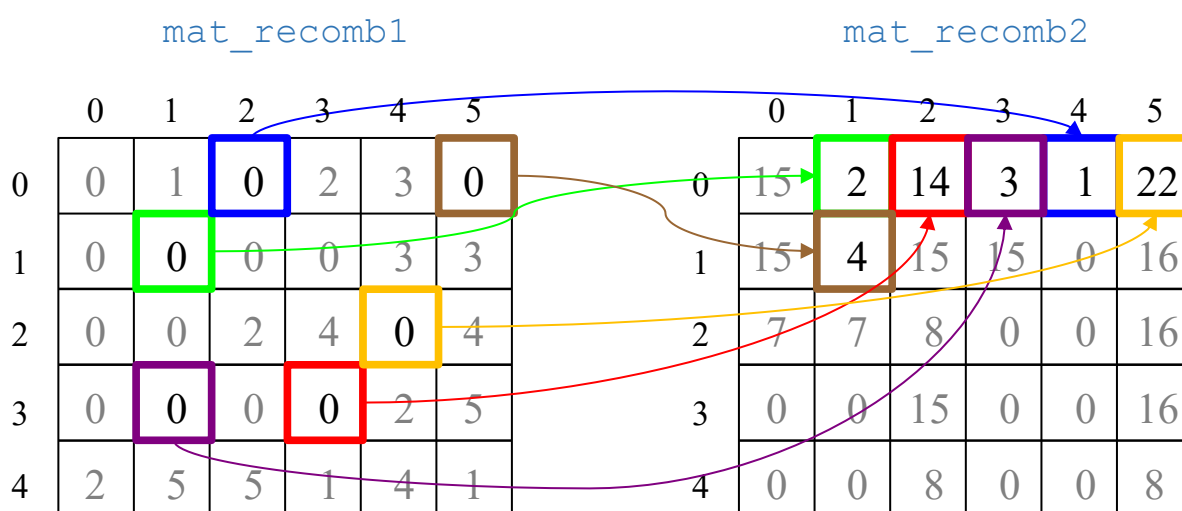
Las flechas del primer gráfico muestran cómo se copia la matriz de juego sobre `mat_recomb1`, fijando a cero las posiciones de un hueco (0, 0) y un bloque sólido (2, 0), dejando a cero un espacio vacío (3, 0), copiando un elemento básico (0, 1) y eliminando los códigos de gelatinas simple (0, 2) y doble (0, 5):



Las flechas del segundo gráfico muestran cómo se trasladan las mismas posiciones anteriores de la matriz de juego sobre `mat_recomb2`, conservando los códigos del hueco (0, 0), del bloque sólido (2, 0) y del espacio vacío (3, 0), fijando a cero la casilla del elemento básico (0, 1), y fijando el código base de gelatina en las posiciones (0, 2) y (0, 5):



El tercer gráfico muestra cómo se va recorriendo linealmente (filas / columnas) las posiciones de `mat_recomb2` y, para las posiciones donde antes había un elemento, se escoge aleatoriamente una posición de `mat_recomb1` que contenga un código diferente de cero; este código se suma al contenido de la posición destino en `mat_recomb2`, y la posición origen en `mat_recomb1` se pone a cero:



Por último, hay que tener en cuenta situaciones en que los últimos códigos disponibles en `mat_recomb1` (casillas con valor diferente de cero) no se puedan colocar en las últimas posiciones de `mat_recomb2` sin generar una secuencia de 3 elementos del mismo tipo: **esto generará un bucle infinito en los pasos 5 y 6 del algoritmo propuesto**. Por lo tanto, se debe adaptar dicho algoritmo para que, dado este caso, empiece todo el proceso de nuevo desde el paso 1.

2.7 Tarea 1C: hay_secuencia()

La rutina para detectar secuencia recibe la dirección base de la matriz por `R0` y devuelve cierto (`R0 = 1`) si existe alguna secuencia de 3 o más elementos del mismo tipo consecutivos, en vertical o en horizontal, o falso (`R0 = 0`) en caso contrario. Se propone el siguiente algoritmo:

1. recorrer linealmente (filas / columnas) todas las posiciones de la matriz de juego,
2. si el código de la posición actual es un espacio vacío (0, 8 o 16), un bloque sólido o un hueco, ignorar dicha posición,
3. si la columna actual es anterior a la penúltima, contar las repeticiones del elemento actual en orientación Este (0), saliendo de la rutina con `R0 = 1` en el caso de que el número de repeticiones sea superior o igual a 3,
4. si la fila actual es anterior a la penúltima, contar las repeticiones del elemento actual en orientación Sur (1), saliendo de la rutina con `R0 = 1` en el caso de que el número de repeticiones sea superior o igual a 3,
5. si se ha recorrido toda la matriz y no se ha detectado ninguna secuencia de 3 o más elementos, salir de la rutina con `R0 = 0`.

El siguiente gráfico muestra una matriz de juego de ejemplo con diferentes colores de fondo para indicar los posibles casos de análisis de posiciones, según se analice en horizontal y vertical (blanco), solo en vertical (azul claro), solo en horizontal (naranja claro) o sin análisis posible (violeta):

Matriz de juego

	0	1	2	3	4	5
0	15	1	9	2	3	20
1	15	2	15	15	3	4
2	7	7	10	4	6	19
3	0	5	13	21	5	9
4	2	3	15	6	2	21

Además, se han remarcado algunas casillas individuales (rojo, azul, verde) para ejemplificar el proceso de análisis en tres regiones distintas de la matriz y con diferentes tipos de elementos. A continuación se muestran los resultados de estos análisis para cada casilla, que se obtendrán invocando a `cuenta_repeticiones()`:

- La casilla roja se encuentra en la zona blanca, es decir, hay que analizar en horizontal y en vertical. En el análisis horizontal del número de repeticiones el resultado es 2, puesto que el código 9 es una gelatina simple de tipo 1, de modo que hay 2 unos consecutivos. En el análisis vertical el resultado es 1, puesto que la casilla inferior contiene un elemento diferente a 1:

	1	2	3	
0	1	9	2	$\text{cuenta_repeticiones}(\text{mat}, 0, 1, 0) = 2$
1	2	15	15	
2	7	10	4	

$\text{cuenta_repeticiones}(\text{mat}, 0, 1, 1) = 1$

- La casilla azul se encuentra en la zona azul claro, es decir, solo hay que analizar en vertical. El número de repeticiones es 2 porque 20 es una gelatina doble de tipo 4, pero 19 no lo es:

	3	4	5	
0	2	3	20	$\text{cuenta_repeticiones}(\text{mat}, 0, 5, 1) = 2$
1	15	3	4	
2	4	6	19	

- La casilla verde se encuentra en la zona naranja claro, es decir, solo hay que analizar en horizontal. En este caso hay 4 cincos seguidos, dos de ellos con gelatinas, una simple (13) y otra doble (21):

	1	2	3	4	5	
2	7	10	4	6	19	
3	5	13	21	5	9	$\text{cuenta_repeticiones}(\text{mat}, 3, 1, 0) = 4$
4	3	15	6	2	21	

2.8 Tarea 1D: *elimina_secuencias()*

La rutina para eliminar secuencias de elementos recibe la dirección de la matriz de juego por `R0` y otra dirección de una matriz de marcas por `R1`. Su cometido es eliminar todas las secuencias de 3 o más elementos consecutivos del mismo tipo, en horizontal y en vertical, poniendo los elementos básicos a cero y reduciendo el nivel de las gelatinas. Además, debe almacenar un identificador único para cada secuencia sobre la matriz de marcas.

Se propone el siguiente algoritmo:

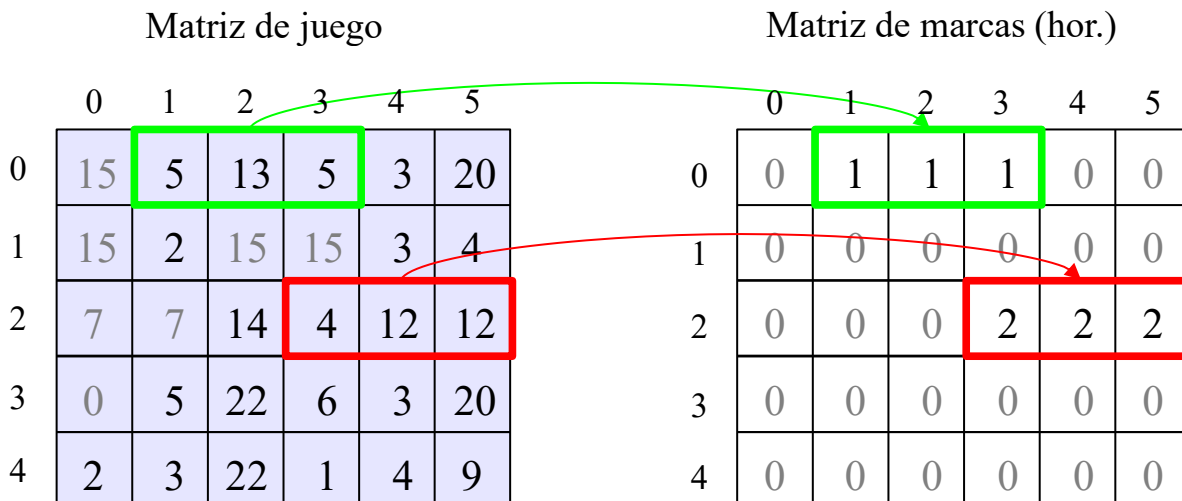
1. poner toda la matriz de marcas a cero,
2. marcar las secuencias horizontales de elementos (con o sin gelatina) con un identificador único para cada secuencia, empezando por 1,
3. marcar las secuencias verticales de elementos (con o sin gelatina) con el identificador de la primera secuencia horizontal que intersecte, o bien con un nuevo identificador si la secuencia vertical no intersecta con ninguna secuencia horizontal. Los identificadores nuevos deberán continuar a partir del último identificador horizontal,
4. para todas las posiciones marcadas, eliminar el tipo de elemento y decrementar su nivel de gelatina, si lo tiene.

Este algoritmo es bastante general, ya que los procesos de los pasos 2 y 3 son suficientemente complejos como para requerir sus propios algoritmos, que se encapsularán en unas rutinas de soporte cuyas cabeceras se encuentran definidas al final del fichero `candy1_secu.s`: `marcar_horizontales()` y `marcar_verticales()`.

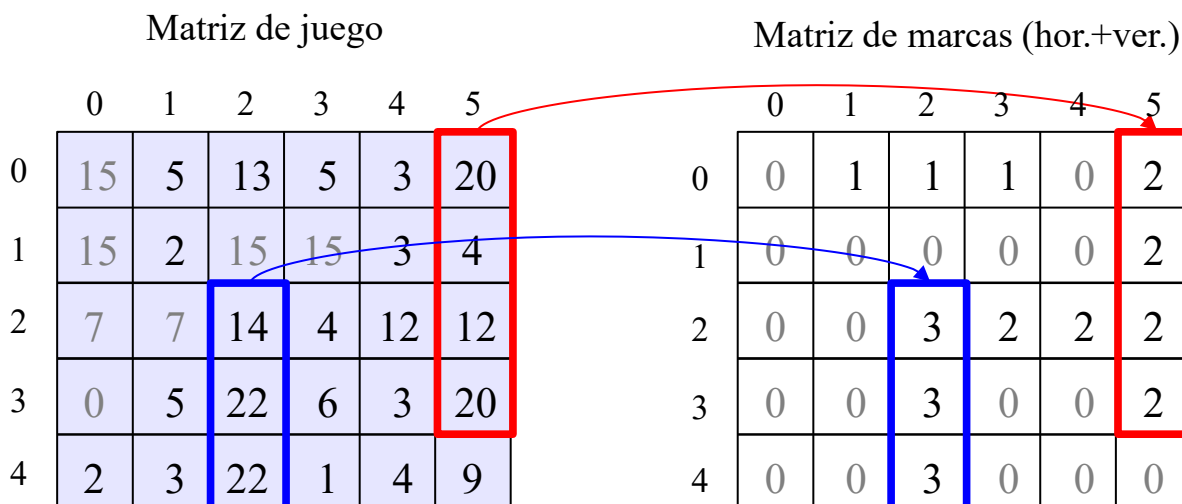
Evidentemente, los algoritmos para marcar secuencias horizontales y verticales deben **ignorar** las posiciones de la matriz de juego con códigos de bloque sólido (7), hueco (15), o de casilla vacía, con o sin gelatina (0, 8, 16).

A continuación se muestran ejemplos gráficos de su funcionamiento.

El primer gráfico es un ejemplo del funcionamiento de `marcar_horizontales()`, donde se muestra la detección de dos secuencias horizontales de 3 elementos consecutivos del mismo tipo. La primera secuencia recibe el identificador 1 y la segunda recibe el identificador 2:



El segundo gráfico es un ejemplo del funcionamiento de `marcar_verticales()`. La primera secuencia vertical intersecta con la secuencia horizontal con identificador 2, de modo que debe adquirir este identificador para marcar la fusión de las secuencias. La segunda secuencia vertical no intersecta con ninguna secuencia horizontal, de modo que se le debe asignar un nuevo identificador, que en este caso será el 3:



El último gráfico ejemplifica la eliminación de los elementos de las posiciones marcadas anteriormente, poniendo a cero los códigos de elemento (con o sin gelatina) y reduciendo el nivel de gelatina ($8 \rightarrow 0$, $16 \rightarrow 8$):

Matriz de marcas

	0	1	2	3	4	5
0	0	1	1	1	0	2
1	0	0	0	0	0	2
2	0	0	3	2	2	2
3	0	0	3	0	0	2
4	0	0	3	0	0	0

Matriz de juego anterior

	0	1	2	3	4	5
0	15	5	13	5	3	20
1	15	2	15	15	3	4
2	7	7	14	4	12	12
3	0	5	22	6	3	20
4	2	3	22	1	4	9

Matriz de juego actualizada

	0	1	2	3	4	5
0	15	0	0	0	3	8
1	15	2	15	15	3	0
2	7	7	0	0	0	0
3	0	5	8	6	3	8
4	2	3	8	1	4	9

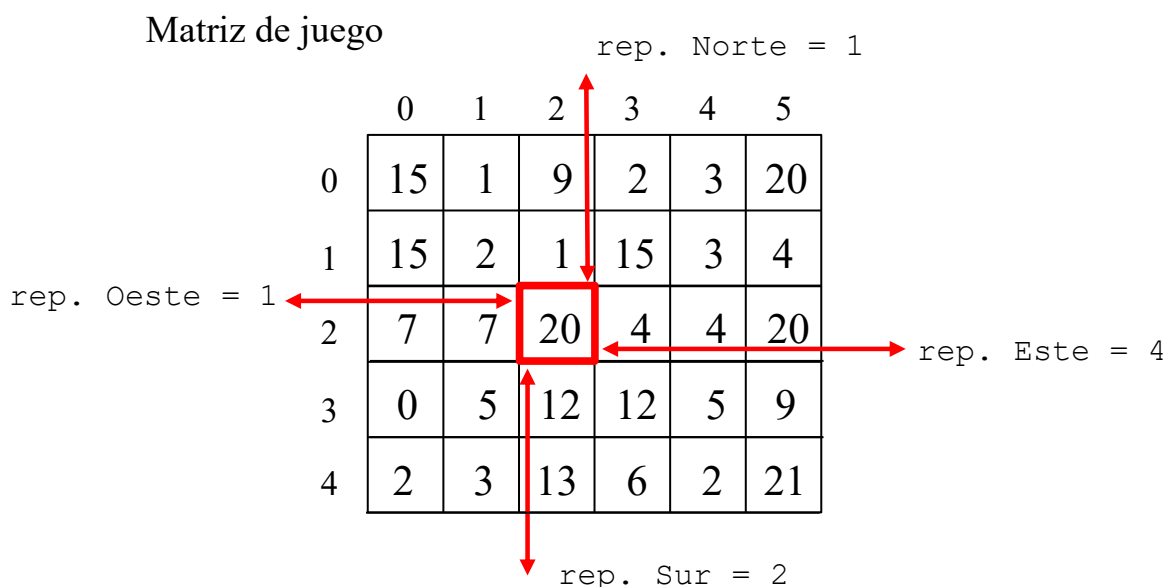
Hay que observar que los elementos básicos y las gelatinas simples se ponen a cero, mientras que las gelatinas dobles se convierten en gelatinas simples vacías (8). La rutina de bajada de elementos permitirá colocar nuevos elementos dentro de estas nuevas gelatinas simples, del mismo modo que bajará (si puede) elementos sobre las posiciones vacías creadas.

2.9 Tarea 1E: cuenta_repeticiones()

La rutina para contar repeticiones (de códigos de elemento) recibe la dirección base de la matriz por `R0`, fila y columna de la posición inicial por (`R1`, `R2`) y la orientación a seguir por `R3` (0: Este, 1: Sur, 2: Oeste, 3: Norte). Devuelve por `R0` el número de elementos consecutivos del mismo tipo que el elemento de la posición inicial, ignorando los códigos base de las gelatinas. El propio elemento inicial también se cuenta, de modo que, como mínimo, el resultado será 1. Se propone el siguiente algoritmo:

1. obtener el tipo de elemento de la posición inicial, filtrando las posibles marcas de gelatina (quedarse con los 3 bits de menor peso),
2. para cada orientación, realizar un recorrido de las posiciones mientras no se llegue al límite del tablero y mientras el tipo de elemento de las posiciones visitadas coincida con el inicial,
3. cuando no se cumpla alguna de las condiciones de recorrido, retornar de la rutina devolviendo en `R0` el número de elementos detectados.

El siguiente gráfico muestra un ejemplo de resultado para cada orientación desde la posición inicial (2, 2), enmarcada con un cuadrado rojo:



El elemento inicial 20 es una gelatina doble de tipo 4. El recorrido hacia el Este (4 repeticiones) termina en el borde del tablero. El recorrido hacia el Sur (2 repeticiones) termina en la posición (4, 2), que contiene una gelatina simple de tipo 5. Los recorridos

hacia el Oeste (1 repetición) y hacia el Norte (1 repetición) no detectan ninguna coincidencia.

Atención:

la rutina `cuenta_repeticiones()` **NO** debe descartar los códigos iniciales 0, 7, 8, 15 y 16. Si se invoca sobre una posición vacía (0, 8, 16), un bloque sólido (7) o un hueco (15), la rutina contabilizará los elementos consecutivos del mismo grupo, es decir, posiciones vacías (0, 8, 16) o bloque sólido/hueco (7, 15). El motivo de esta directriz es permitir algo más de flexibilidad en el funcionamiento de la rutina, ya que se puede utilizar para otros fines además de contar códigos de elemento, como por ejemplo, contar marcas de secuencia, tal como realiza la función ya implementada `calcula_puntuaciones()`. Por lo tanto, si una rutina de tarea requiere ignorar las posiciones vacías, los bloques sólidos o los huecos, debe realizar las comprobaciones en la propia rutina, antes de llamar a `cuenta_repeticiones()`.

2.10 Tarea 1F: *baja_elementos()*

La rutina para bajar elementos solo recibe la dirección base de la matriz por `R0`. Esta rutina baja **una única posición** todos los elementos de las posiciones superiores a las que contengan un espacio vacío (0) o un código de gelatina vacío (8 o 16), primero en vertical y luego en diagonal. Además, genera un nuevo elemento aleatorio en la posición vacía más alta de cada columna. Devuelve cierto (`R0 = 1`) si ha habido movimientos o falso (`R0 = 0`) en caso contrario. Se propone el siguiente algoritmo:

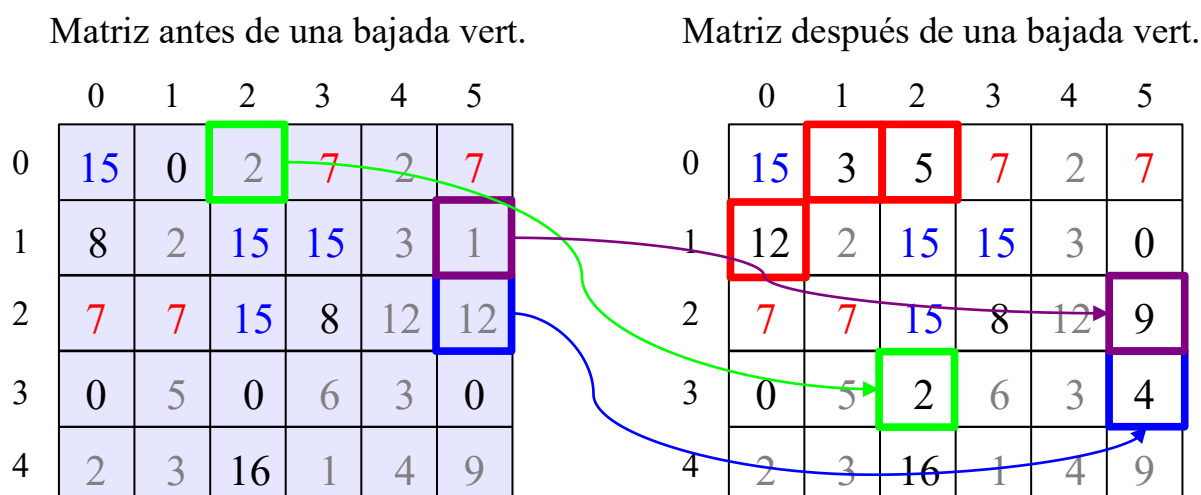
1. bajar elementos en sentido vertical y devolver cierto si ha habido algún movimiento,
2. si no ha habido ningún movimiento en vertical, bajar elementos en diagonal y devolver cierto o falso según haya habido movimientos en diagonal.

Este algoritmo es muy genérico, pero separa claramente las tareas de bajar en vertical y bajar en diagonal, las cuales se pueden encapsular en dos rutinas de soporte adicionales.

La rutina `baja_verticales()` recibe por parámetro la dirección base de la matriz en `R4`, y devuelve cierto (`R0 = 1`) si ha conseguido bajar algún elemento en vertical, o falso (`R0 = 0`) en caso contrario. Se propone el siguiente algoritmo:

1. recorrer toda la matriz de juego en sentido inverso, empezando por la última fila y columna, decrementando el índice de columna y después el de fila,
2. observar solo las posiciones vacías (0, 8, 16), ignorando el resto de posiciones,
3. para las posiciones vacías que no sean la más alta de cada columna, buscar un código de elemento en la posición superior (misma columna), saltando los posibles huecos superiores (consecutivos) hasta llegar a un bloque sólido, a una posición vacía, o salir del tablero por arriba,
4. si se ha encontrado un código de elemento válido, poner a cero la casilla superior donde se ha encontrado ese elemento y guardarlo en la casilla actual, teniendo en cuenta que los códigos base de gelatina no se mueven de sus posiciones originales,
5. para la posición vacía más alta de cada columna, generar un nuevo elemento de forma aleatoria (número del 1 al 6, a sumar al posible código base de gelatina de la posición); esta posición vacía más alta de cada columna será la de la primera fila o de una fila inferior, si en las posiciones superiores hay huecos (15); si en las posiciones superiores hay algún bloque sólido (7), entonces no deberá generarse ningún elemento aleatorio.

En el siguiente gráfico se muestra un ejemplo de bajada de un movimiento vertical. Los códigos de bloques sólidos y los huecos se han escrito en rojo y azul, respectivamente, porque tienen un papel importante en el algoritmo. Además, las posiciones vacías están en negro y las posiciones no vacías están en gris. Las bajadas se visualizan con cuadros y flechas de colores entre dos estados consecutivos de la matriz de juego, es decir, antes y después de la bajada en vertical:



El cuadro verde es un ejemplo de bajada que atraviesa dos huecos (baja 3 filas). El cuadro azul extrae un elemento de tipo 4 de una gelatina simple y lo guarda en la posición inferior. El cuadro violeta extrae un elemento básico de tipo 1 y lo añade al código base de gelatina simple de la posición inferior, quedándose la casilla original a cero porque encima hay un bloque sólido (7).

La casilla (3, 0) también se queda vacía (0) porque está debajo de un bloque sólido. La casilla (4, 2), de momento, se queda vacía (16), porque no tenía ningún elemento encima en el momento en que se trató de realizar la bajada vertical sobre esa posición. En la próxima activación de la rutina `baja_verticales()` ya recibirá el elemento superior (2).

Por último, los cuadros rojos indican la generación de nuevos elementos aleatorios. El nuevo elemento en la casilla (0, 2) se genera porque el elemento que había anteriormente (2) se ha desplazado hacia abajo previamente. El nuevo elemento en la casilla (0, 1) se genera porque es la casilla vacía más alta de la columna 1. También el nuevo elemento en la casilla (1, 0) es debido a que se trata de la casilla vacía más alta de la columna 0, pero en este caso hay un hueco (15) encima, de modo que se simula una "caída" de un nuevo elemento aleatorio, a través de ese hueco. Sin embargo, sobre la casilla (2, 3) no puede caer un elemento aleatorio porque lo impide el bloque sólido (7) que hay encima del hueco.

La rutina `baja_laterales()` recibe por parámetro la dirección base de la matriz en `R4`, y devuelve cierto (`R0 = 1`) si ha conseguido bajar algún elemento en diagonal, o falso (`R0 = 0`) en caso contrario. Se propone el siguiente algoritmo (simplificando el algoritmo anterior):

1. recorrer toda la matriz de juego en sentido inverso, empezando por la última fila y columna, decrementando el índice de columna y después el de fila,
2. para las posiciones vacías que no sean la más alta de cada columna, obtener un código de elemento entre las posiciones superior-izquierda o superior-derecha (visitar solamente una fila superior), de forma aleatoria cuando las dos posiciones diagonales existan (estén dentro del tablero) y contengan un elemento válido para desplazar (código de elemento básico o con gelatina),
3. con el elemento de la posición superior elegida, proceder como en el paso 4 de la rutina de bajada en vertical.

En el siguiente gráfico se muestra un ejemplo de bajada de un movimiento lateral. El cuadro verde es una bajada desde la posición superior-izquierda a la (1, 5), mientras que el cuadro azul es una bajada desde la posición superior-derecha a la (2, 3), en este caso con adición de código de gelatina simple. Las posiciones (0, 4) y (1, 4) se quedan a cero, de momento, pero ya se rellenarán en las próximas bajadas verticales.

Matriz después de las bajadas verticales

	0	1	2	3	4	5
0	15	3	4	7	2	7
1	12	2	15	15	3	0
2	7	7	15	8	12	9
3	0	5	5	6	3	4
4	2	3	18	1	4	9

Matriz con una bajada diagonal

	0	1	2	3	4	5
0	15	3	4	7	0	7
1	12	2	15	15	0	2
2	7	7	15	11	12	9
3	0	5	5	6	3	4
4	2	3	18	1	4	9

La posición (3, 0) no permite bajadas verticales ni laterales, porque está cubierta por dos bloques sólidos superiores.

2.11 Tarea 1G: *hay_combinacion()*

La rutina para determinar si hay combinación recibe la dirección base de la matriz por `R0` y devuelve cierto (`R0 = 1`) si existe alguna combinación entre 2 posiciones consecutivas (jugada) que pueda formar una secuencia de 3 o más elementos del mismo tipo, en horizontal o en vertical, o falso (`R0 = 0`) en caso contrario. Se propone el siguiente algoritmo:

1. recorrer linealmente (filas / columnas) todas las posiciones de la matriz de juego,
2. si el código de la posición actual es un espacio vacío (0, 8 o 16), un bloque sólido o un hueco, ignorar dicha posición,
3. si la columna actual es anterior a la última (hacia la derecha), el código de elemento de la posición derecha es diferente del código de elemento de la posición actual y no es un espacio vacío, ni un bloque sólido, ni un hueco, intercambiar los códigos de la posición actual con la posición derecha, verificar si hay alguna orientación en la que se produzca una secuencia válida sobre alguna de las dos posiciones de intercambio, volver a intercambiar los códigos y, si se ha detectado alguna secuencia, retornar con `R0 = 1`,
4. si la fila actual es anterior a la última (hacia abajo), el código de elemento de la posición inferior es diferente del código de elemento de la posición actual y no es un espacio vacío, ni un bloque sólido, ni un hueco, intercambiar los códigos de la posición actual con la posición inferior, verificar si hay alguna orientación en la que se produzca una secuencia válida sobre alguna de las dos posiciones de intercambio, volver a intercambiar los códigos y, si se ha detectado alguna secuencia, retornar con `R0 = 1`,
5. si se ha recorrido toda la matriz y no se ha detectado ninguna posible combinación, salir de la rutina con `R0 = 0`.

A continuación se muestra una matriz de ejemplo, con diversos códigos de elementos, y la misma matriz sin códigos de gelatinas, para poder observar mejor las posibles secuencias:

Matriz de juego						
	0	1	2	3	4	5
0	15	1	2	9	1	15
1	15	2	13	19	3	4
2	2	5	10	7	5	19
3	7	5	18	22	6	9
4	2	7	13	5	2	21

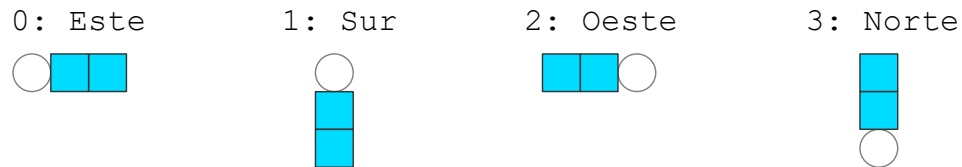
Matriz sin códigos gel.						
	0	1	2	3	4	5
0	15	1	2	1	1	15
1	15	2	5	3	3	4
2	2	5	2	7	5	3
3	7	5	2	6	6	1
4	2	7	5	5	2	5

En los tableros anteriores, las casillas blancas indican que se puede realizar un intercambio horizontal y otro vertical, como en la posición (1, 1), que se ha enmarcado en verde junto con sus casillas de intercambio. En las casillas azul claro solo es posible realizar un intercambio vertical, como en la posición (1, 5), que se ha enmarcado en azul junto con su casilla inferior. En las casillas naranja claro solo es posible realizar un intercambio horizontal, como en la posición (4, 4), que se ha enmarcado en rojo junto con su casilla derecha. Por último, en la casilla violeta (4, 5) no es posible realizar ningún intercambio, aunque el código de dicha posición sí puede intervenir en intercambios con las casillas vecinas.

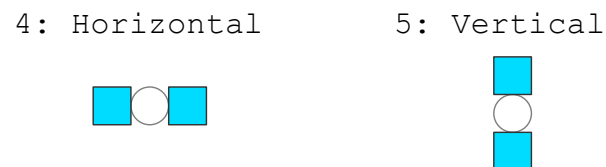
Para simplificar la implementación del algoritmo de detectar combinaciones, se proporciona una rutina de soporte ya implementada de nombre `detectar_orientacion()`, que recibe por parámetro la dirección de la matriz de juego (`R4`) y las coordenadas (fila, columna) de una posición inicial a inspeccionar (`R1`, `R2`), y devuelve un código de orientación que indica la existencia de una secuencia válida, según la ubicación de la posición inicial dentro de la secuencia. Hay que tener en cuenta que esta rutina llama a `contar_repeticiones()` para realizar su trabajo, de modo que no funcionará hasta que la tarea **1E** esté implementada.

A continuación se muestran todos los códigos de orientación posibles, junto con unos gráficos que representan la posición inicial como un círculo, y las otras dos posiciones como cuadrados azul turquesa:

- 0..3 → la posición inicial está al principio de una secuencia:

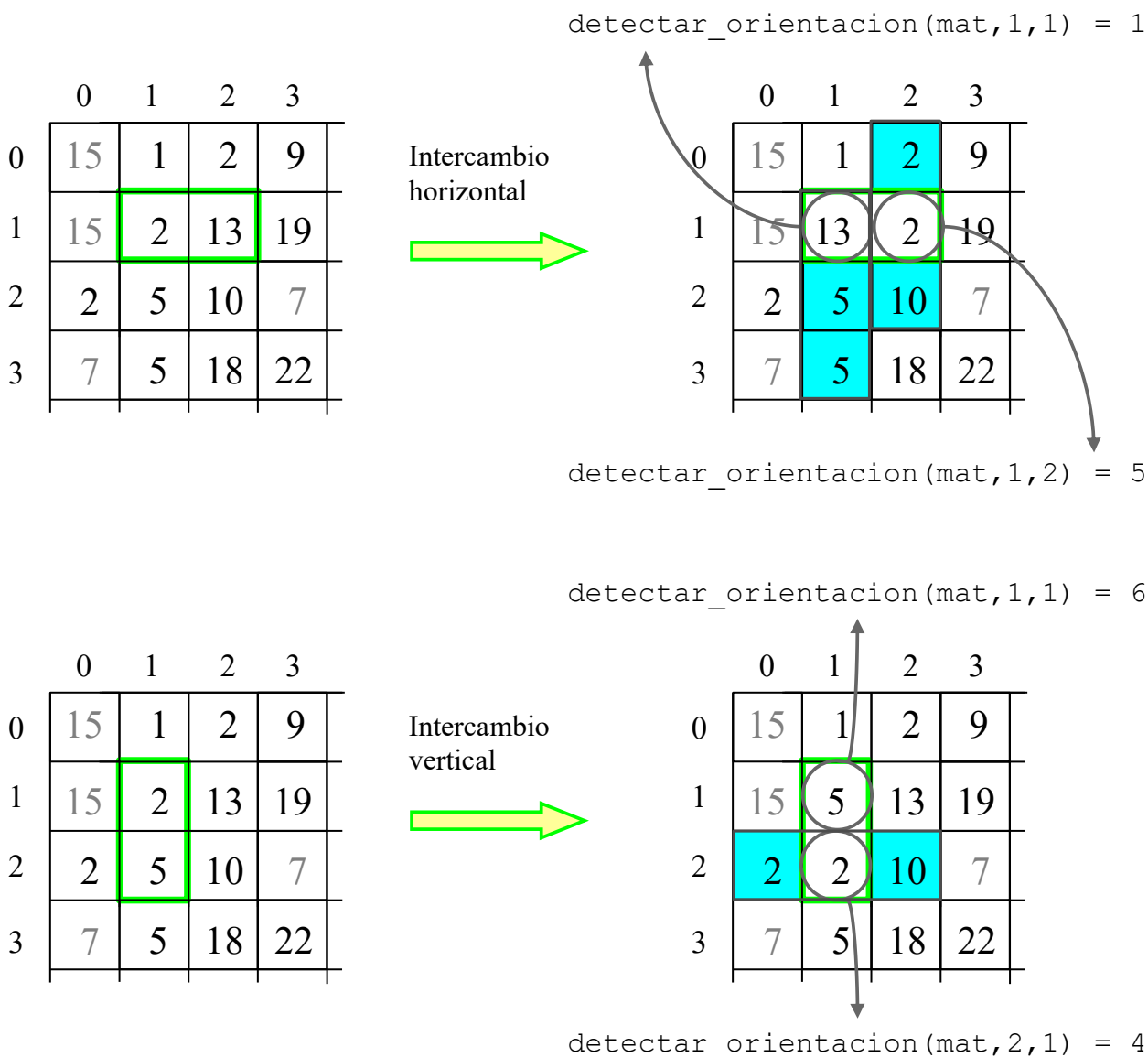


- 4..5 → la posición inicial está en medio de una secuencia:



- 6 → la posición inicial no está dentro de ninguna secuencia.

A continuación se muestran ejemplos de posibles intercambios en horizontal y en vertical, mostrando el resultado de la llamada a la rutina `detectar_orientación()` sobre las casillas (posiciones iniciales) que han participado en el intercambio:



En definitiva, la rutina `hay_combinacion()` tiene que devolver cierto al primer intercambio que detecte una posible orientación de secuencia (resultado entre 0 y 5), y falso si las dos posiciones de todos los posibles intercambios obtienen un resultado igual a 6 en el cálculo de detectar orientación.

2.12 Tarea 1H: *sugerir_combinacion()*

La rutina para sugerir combinación recibe la dirección base de la matriz por `R0` y la dirección base de un vector de posiciones por `R1`, sobre el cuál se devolverá (por referencia) una lista de tres posiciones (`x1`, `y1`, `x2`, `y2`, `x3`, `y3`) correspondientes a las coordenadas (`x`:columna, `y`:fila) de tres casillas de una posible combinación.

Atención:

en todo este manual, cuando se hace referencia a un código de posición de la matriz se utilizan coordenadas (fila, columna), pero la especificación de la tarea **1H** pide explícitamente que para generar la lista de posiciones de una combinación se utilice el orden opuesto (`x`: columna, `y`: fila).


Se propone el siguiente algoritmo, en base a algunos pasos del algoritmo de `hay_combinacion()`:

1. escoger una posición aleatoria de la matriz para buscar una posible combinación,
2. si se detecta una combinación sobre la posición actual o una colindante (derecha o inferior) según los pasos 3 o 4 del algoritmo anterior, generar el vector de posiciones de 3 casillas que formen la combinación, de acuerdo con la posición inicial, el tipo de intercambio y el código de orientación de secuencia detectada,
3. si no se detecta ninguna combinación, pasar a la siguiente posición y, en caso de llegar a la última, volver a empezar por el principio del tablero; se puede suponer que, al llamar a la rutina, existirá al menos una combinación.

El algoritmo se ha detallado poco porque utiliza procedimientos similares al algoritmo de la tarea **1G**. La diferencia más significativa reside en la generación de las posiciones de la combinación detectada, para lo cual se propone implementar otra rutina de soporte `generar_posiciones()`, que recibirá por parámetro la dirección del vector de posiciones (`R0`), las coordenadas de la posición inicial (`R1`:fila, `R2`:columna), el código de orientación de la secuencia devuelto por `detectar_orientacion()` (`R3`) y un código del tipo de posición inicial donde se ha detectado la secuencia (`R4`).

Los siguientes gráficos muestran los 4 códigos de posición inicial posibles, según el tipo de intercambio (horizontal / vertical) y la ubicación de la posición inicial relativa al intercambio (izquierda / derecha, arriba / abajo); la posición inicial se representa como un círculo y la posición a intercambiar como un cuadrado naranja:

- 0..1 → intercambio horizontal:

0: Horizontal Izquierda 

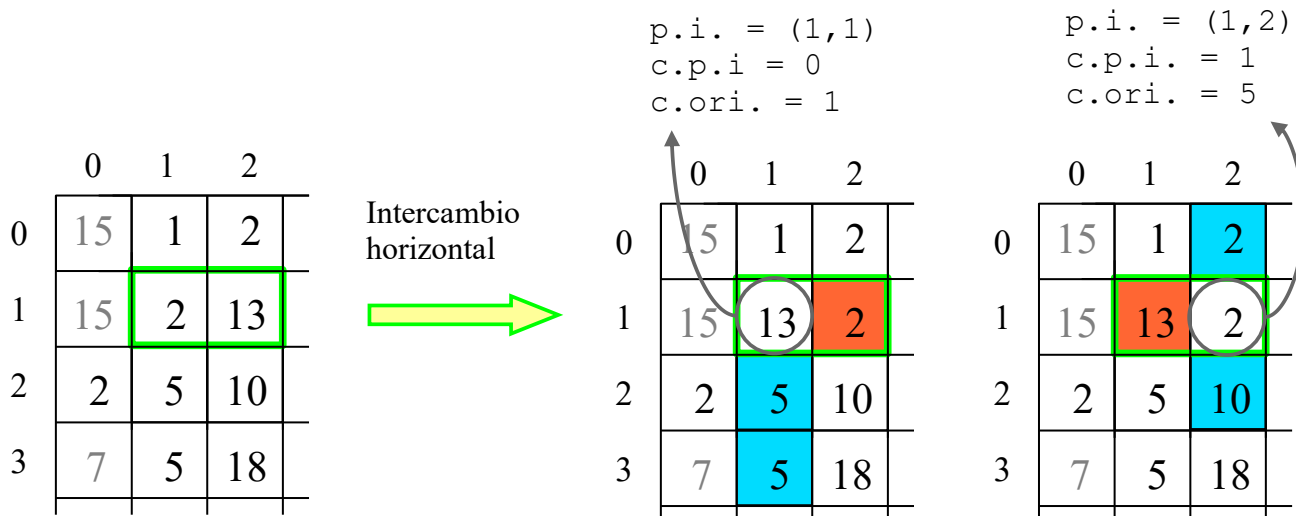
1: Horizontal Derecha 

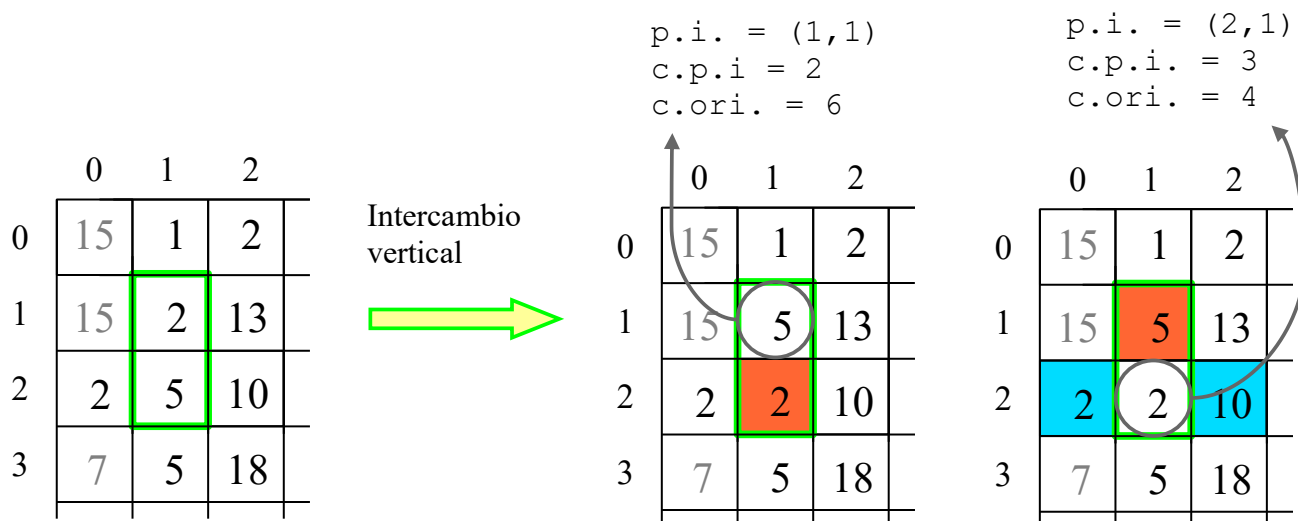
- 2..3 → intercambio vertical:

2: Vertical Arriba 

3: Vertical Abajo 

A continuación se muestran los ejemplos de intercambio del apartado anterior, pero combinando el resaltado de los cuadrados azules de secuencia detectada con el cuadrado naranja de la posición de intercambio, además del círculo para la posición inicial de análisis de la secuencia. Hay que observar que la posición inicial no siempre coincide con la posición actual del recorrido del tablero, (1, 1) en los ejemplos, ya que puede corresponder a una posición vecina (1, 2) o (2, 1). Para cada ejemplo se adjuntan los valores de posición inicial (*p.i.*), código de posición inicial (*c.p.i.*) y código de orientación de la secuencia detectada (*c.ori.*):





En caso de que se pueda crear secuencia, una combinación se describe con la posición resaltada en naranja y las posiciones resaltadas en azul. La posición inicial se utilizará para detectar si hay secuencia después del intercambio de códigos. El código de posición inicial (*c.p.i.*) servirá para deducir la posición origen del código que se ha intercambiado.

Se propone el siguiente algoritmo para `generar_posiciones()`:

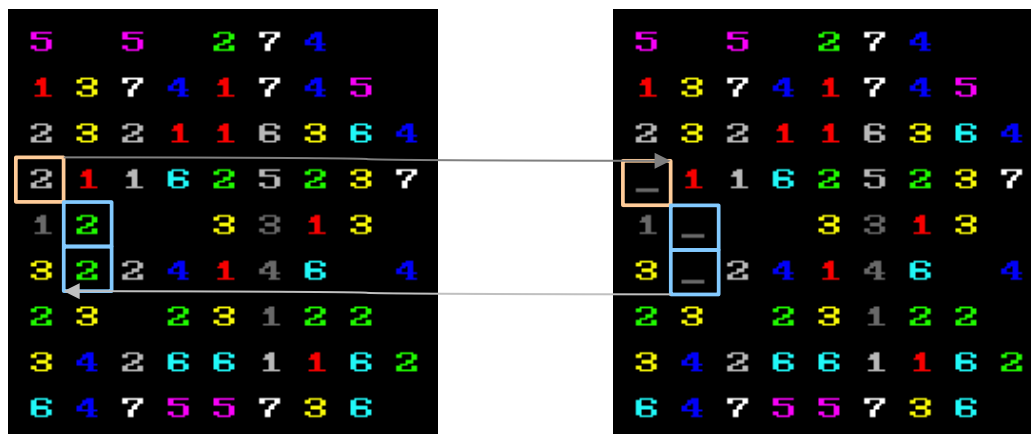
1. en el vector de posiciones, guardar las coordenadas (*x1*, *y1*) correspondientes a la posición donde se encontraba el código intercambiado, según la *p.i.* y el *c.p.i.*; por ejemplo, si *c.p.i.* es 0 (Horizontal Izquierda), la posición original del código intercambiado será la de la derecha a la *p.i.*,
2. en el vector de posiciones, guardar las coordenadas (*x2*, *y2*, *x3*, *y3*) correspondientes a las dos posiciones que completan la secuencia detectada, según la *p.i.* y el *c.ori.*; por ejemplo, si *c.ori.* es 1 (Sur), hay que registrar las dos posiciones inferiores de *p.i.*

Debido a que hay 4 códigos de posición inicial y 6 códigos de orientación, surgen 24 combinaciones teóricas, aunque algunas de ellas son incompatibles, como $c.p.i. = 0$ y $c.ori. = 0$, ya que no es posible que el cuadrado naranja se superponga con uno de los cuadrados azules. La siguiente tabla muestra las 16 combinaciones compatibles:

		$c.ori.$					
		0	1	2	3	4	5
$c.p.i.$	0						
	1						
	2						
	3						

Las combinaciones incompatibles se han marcado con fondo gris. Para el resto, se puede observar que, para cualquier fila de la tabla anterior, la posición del cuadrado naranja respecto a la circunferencia siempre es la misma, y que, para cualquier columna, la posición de los dos cuadrados azules respecto a la circunferencia siempre es la misma. Esta observación es importante para entender que el algoritmo se puede dividir en los dos pasos mencionados (posición de intercambio, posiciones de secuencia), de forma independiente (4+6 casos independientes, en lugar de 16 casos individuales).

Para completar este apartado, a continuación se muestra un ejemplo de resaltado de una combinación sugerida en pantalla, correspondiente a $p.i. = (3, 1)$, $c.ori. = 1$, $c.p.i. = 1$, donde se han añadido unos recuadros y unas flechas para representar el cambio de estado en las posiciones sugeridas:



En este ejemplo, el vector de posiciones sugeridas sería $(0, 3, 1, 4, 1, 5)$, o sea, las posiciones $(3, 0)$, $(4, 1)$ y $(5, 1)$. El programa principal propuesto se encarga de realizar el resaltado mediante la sustitución temporal de los códigos de los elementos contenidos en las posiciones sugeridas por una marca especial (-1) , la cual es representada en pantalla como un subrayado '_' por la rutina `escribe_matriz()`; al cabo de medio segundo se restituyen los códigos originales y se vuelve a escribir la matriz. Esto se repite mientras el usuario no realiza ningún movimiento. Hay que observar que el proceso de sugerencia de combinación también incluye a los códigos con gelatinas.

3 Depuración de programas en la NDS

3.1 Depuración de código escrito en lenguaje C

En el caso de tener que probar código escrito en lenguaje C, el depurador *Insight/gdb* resulta poco eficaz debido a que está preparado explícitamente para analizar y ejecutar código máquina, de modo que tiene un comportamiento aparentemente errático cuando se ejecuta un trozo de programa en C paso a paso, aunque puede ser de utilidad si se usan *breakpoints* u otras funcionalidades (inspección de registros y memoria, etc.).

Por este motivo, la técnica más básica que nos permite hacer un seguimiento del código escrito en C es el uso de la función `printf()`, combinado con códigos ANSI de posicionamiento en pantalla ("`\x1b[nf;ncH`", donde `nf` y `nc` son los números de fila y de columna) y cambio de color ("`\x1b[ccm`", donde `cc` es el código de color).

Para encontrar ejemplos se puede inspeccionar el código de la función para contabilizar los puntos de las secuencias, ubicada en el fichero `candy1_sopo.c`:

```
...
int calcula_puntuaciones(char mar[][COLUMNS])
{
    ...
    puntos = detectar_combo(nh,nv,texto);
    printf("\x1b[%dm\x1b[%d;20H %s", (37 + num_pun%3), (12+ult_tex), texto);
}
...
```

Aunque en este ejemplo la llamada a la función `printf()` está pensada para mostrar información útil al usuario (contabilización de puntos parciales), se pueden incluir otras llamadas similares con fines de testeo, bien sea para marcar que se ha pasado por cierto punto del programa, bien sea para escribir por pantalla el valor de algunas variables.

Además, también es interesante poder modificar del comportamiento del programa en tiempo de ejecución, con el fin de realizar diferentes tipos de pruebas. En la función `main()` suministrada en `candy1_main.c` se ha añadido expresamente una sección que tiene el propósito de permitir pasar de nivel o repetir el nivel con la pulsación de dos teclas específicas ('B' y 'START', respectivamente):

```
...
    if (!falling)          ///// SECCIÓN DE DEPURACIÓN /////
    {
        swiWaitForVBlank();
        scanKeys();
        if (keysHeld() & KEY_B)          // forzar cambio de nivel
        {
            points = 0;
            geles = 0;                    // superado
            change = 1;
        }
        else if (keysHeld() & KEY_START)
        {
            movements = 0;                // repetir
            change = 1;
        }
        lapse++;
    }
...

```

Por último, si se requiere detener la ejecución del programa para poder analizar pausadamente los resultados en pantalla, también se puede hacer uso del teclado, de modo que no se continúe hasta que no se pulse cierta tecla, por ejemplo:

```
...
    printf("\x1b[39m\x1b[10;20H (SELECT)");
    while (!(keysDown() & KEY_SELECT))    // esperar tecla SELECT
    {
        swiWaitForVBlank();
        scanKeys();
    }
    printf("\x1b[10;20H          ");
    while (keysDown() & KEY_SELECT)        // esperar soltar SELECT
    {
        swiWaitForVBlank();
        scanKeys();
    }
...

```

Obviamente, todos los trozos de código de test se tendrán que retirar de la versión definitiva del proyecto, incluidas las instrucciones para forzar el cambio de nivel.

3.2 Depuración de código escrito en lenguaje ensamblador

La forma más eficaz para detectar fallos en un programa es la ejecución instrucción a instrucción (o paso a paso), aunque a veces se requiera mucho tiempo para probar todas las instrucciones relativas a un determinado fallo.

Para depurar un programa escrito en lenguaje ensamblador del ARM podemos ejecutar el depurador *Insight / gdb* sobre el fichero `.elf` correspondiente, como se hacía en la asignatura *Fonaments de Computadors*. Sin embargo, para depurar un programa escrito explícitamente para NDS (fichero `.nds`) también será necesario ejecutar una versión del emulador *DeSmuME* para depuración, que se denomina de *DeSmuME_dev*, donde el sufijo `'_dev'` hace referencia a *'developer'*.

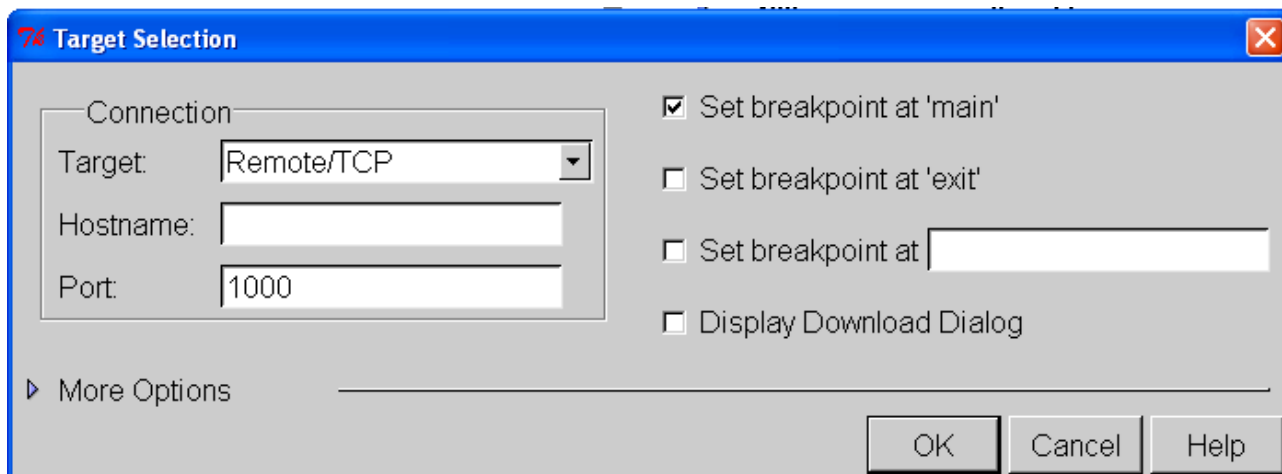
En el fichero `Makefile` que se proporciona dentro del proyecto para realizar esta práctica hay una entrada específica para la depuración del programa:

```
#-----
debug : $(TARGET).nds $(TARGET).elf
    @echo "testing $(TARGET).nds/.elf with DeSmuME_dev/Insight (gdb) through
TCP port=1000"
    @$ (DESMUME) /DeSmuME_dev.exe --arm9gdb=1000 $(TARGET).nds &
    @$ (DEVKITPRO) /insight/bin/arm-eabi-insight $(TARGET).elf &
...
```

Cuando se invoca la regla `debug`, se lanza la ejecución del programa `DeSmuME_dev.exe`, que permite la depuración del código emulado mediante un protocolo específico de mensajes que se envían a través de un puerto TCP. En nuestro caso, hemos especificado la opción `'--arm9gdb=1000'`, puesto que suponemos que el puerto número 1000 está libre en el ordenador local (en otro caso se podría cambiar).

Simultáneamente, también se lanza la ejecución del programa `arm-eabi-insight.exe`, es decir, el depurador *Insight / gdb* convencional, cargando el fichero `.elf` del proyecto.


Cuando se inicia la ejecución con el comando **Run** del *Insight*, aparece el siguiente diálogo:



En la opción '**Target**' hay que indicar `Remote/TCP`, asegurándose de que el número de puerto coincide con el especificado para el *DeSmuME_dev*. No hay que especificar la opción `Simulator`, que es la que se utilizaba en la asignatura *Fonaments de Computadors*; en el contexto actual no funcionaría, puesto que el *gdb* solo simula la ejecución de instrucciones del procesador ARM sobre una memoria principal genérica, pero no está preparado para simular toda la estructura interna de la plataforma NDS (memorias específicas, controladores de E/S, etc.).

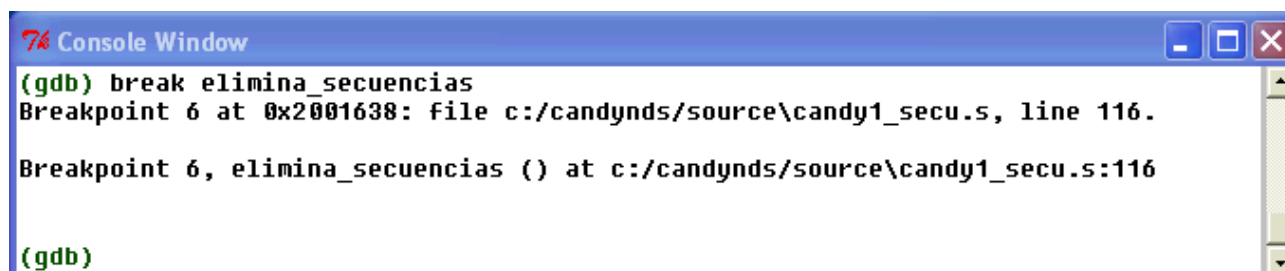
Al arrancar la simulación, el *DeSmuME_dev* se queda con las pantallas en blanco, puesto que está esperando comandos del *Insight / gdb*. Por su parte, el *Insight / gdb* está a la espera de que se inicie la simulación con el comando **Run**, cosa que provocará la carga del programa e inicialización del entorno de depuración.

Por defecto, el *Insight / gdb* crea un *breakpoint* al principio de la función `main()`, lo cual es necesario para que la ejecución del programa se pare justo al principio.

Supongamos que queremos depurar el código de la rutina `elimina_secuencias()`; en este caso, deberemos colocar un *breakpoint* al inicio de la rutina. Esto se puede conseguir mediante la consola del *gdb*, que se abre con el botón  de la barra de herramientas del *Insight*. En dicha consola podemos escribir comandos *gdb*. El primer comando que escribiremos es el siguiente:

```
(gdb) break elimina_secuencias
```

El resultado por la consola será similar al siguiente:



```

74 Console Window
(gdb) break elimina_secuencias
Breakpoint 6 at 0x2001638: file c:/candynds/source\candy1_secu.s, line 116.


Breakpoint 6, elimina_secuencias () at c:/candynds/source\candy1_secu.s:116

(gdb)

```

El comando `break` permite crear un nuevo *breakpoint* en el punto de programa que nos interese, ya sea el inicio de una rutina, ya sea en una instrucción concreta. Para conseguir este segundo tipo de especificación del *breakpoint* podemos utilizar el mismo comando `break` junto con el nombre del fichero fuente seguido por el carácter dos puntos ':' y el número de línea del código fuente donde se encuentra la instrucción. Por ejemplo, supongamos que queremos parar el programa después de realizar las marcas horizontales y verticales de las secuencias detectadas, y que, en el código fuente, la siguiente instrucción después de llamar a `marcar_verticales()` está situada en la línea 127 del fichero `candy1_secu.s`; en este caso, el comando que debemos invocar por la consola es el siguiente:

```
(gdb) break candy1_secu.s:127
```

En este momento podemos indicar al *Insight* que continúe la ejecución hasta que encuentre el siguiente *breakpoint*, con el botón  de la barra de herramientas. Esto nos permitirá interactuar libremente con el juego, hasta el momento en que una jugada genere una o varias secuencias y se invoque la rutina `elimina_secuencias()`.

La siguiente imagen muestra un ejemplo de combinación, resaltada en amarillo a posteriori con un programa de edición de imágenes:



La figura siguiente muestra la ventana principal del **Insight / gdb** cuando éste ha parado la ejecución del programa en la segunda instrucción de la rutina (`mov r6, #0`); de hecho, debería estar parado en la primera instrucción (`push {...}`), pero a veces el **Insight / gdb** no puede establecer el punto exacto de ejecución del programa, lo cual no debe interpretarse como un fallo del programa, teniendo en cuenta la simulación del avance del PC (*Program Counter*) que realiza el procesador (cuando ejecuta una instrucción, el PC está apuntando dos instrucciones más adelante):

```

105 @; de que alguna casilla se encuentre en dicho modo;
106 @; además, la rutina marca todos los conjuntos de secuencias sobre una matriz
107 @; de marcas que se pasa por referencia, utilizando un identificador único para
108 @; cada conjunto de secuencias (el resto de las posiciones se inicializan a 0).
109 @; Parámetros:
110 @;     R0 = dirección base de la matriz de juego
111 @;     R1 = dirección de la matriz de marcas
112 .global elimina_secuencias
113 elimina_secuencias:
114     push {r6-r9, lr}
115
116     mov r6, #0
117
118
119
120
121
122
123
124     bl marcar_horizontales
125     bl marcar_verticales
126
127     mov r8, #0
128
129
130
131
132
133

```

Program is running. 0x2001638 116

A partir de este momento, podemos realizar todas las tareas de análisis y ejecución paso a paso del programa con las correspondientes herramientas del **Insight / gdb**; mientras tanto, el **DeSmuME_dev** irá emulando el estado de la NDS según la ejecución de las instrucciones que el **Insight / gdb** le indique. Este estado puede que se refleje en el contenido de las pantallas de la NDS que genera el **DeSmuME_dev**, aunque el refresco en la visualización de dicho contenido puede que no sea inmediato.

El análisis más común consiste en visualizar el contenido de los registros del ARM:

Register	Value	Register	Value
r0	0x201f410	f0	0
r1	0x201f46c	f1	0
r2	0x2008	f2	0
r3	0x2008	f3	0
r4	0x0	f4	0
r5	0x3e5	f5	0
r6	0x0	f6	0
r7	0x201f404	f7	0
r8	0x201f408	fps	0x0
r9	0x201e034	cpsr	0x2000001f
r10	0x201f40c		
r11	0x1		
r12	0xa		
sp	0xb003cac		
lr	0x2016e2c		
pc	0x2001638		

A partir de aquí podemos observar la dirección de memoria de la matriz de juego, dentro del registro R0, y del mapa de marcas, dentro del registro R1; con estas dos direcciones podemos inspeccionar los trozos de memoria de la NDS que albergan las dos matrices:

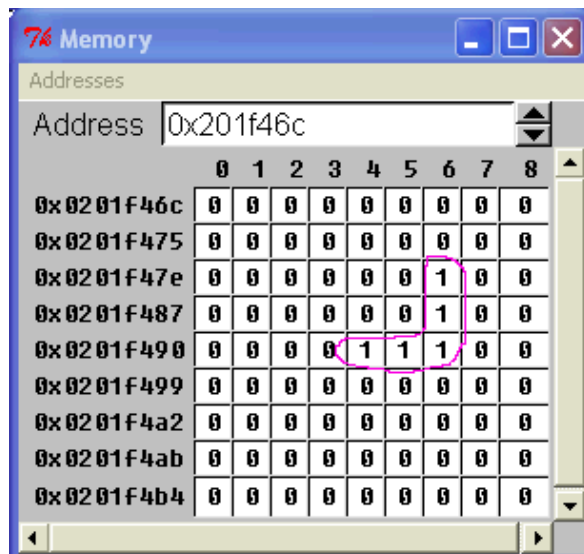
Address	0	1	2	3	4	5	6	7	8
0x0201f410	2	15	5	15	5	7	4	15	15
0x0201f419	4	1	7	3	1	7	4	5	15
0x0201f422	10	3	10	1	1	10	3	3	6
0x0201f42b	10	1	9	3	5	20	3	4	7
0x0201f434	17	2	15	15	3	19	3	4	15
0x0201f43d	3	2	10	1	1	20	1	15	6
0x0201f446	2	3	15	2	2	21	4	5	15
0x0201f44f	2	6	13	5	2	9	2	3	1
0x0201f458	5	4	7	2	6	7	1	2	15

Address	0	1	2	3	4	5	6	7	8
0x0201f46c	0	0	0	0	0	0	0	0	0
0x0201f475	0	0	0	0	0	0	0	0	0
0x0201f47e	0	0	0	0	0	0	0	0	0
0x0201f487	0	0	0	0	0	0	0	0	0
0x0201f490	0	0	0	0	0	0	0	0	0
0x0201f499	0	0	0	0	0	0	0	0	0
0x0201f4a2	0	0	0	0	0	0	0	0	0
0x0201f4ab	0	0	0	0	0	0	0	0	0
0x0201f4b4	0	0	0	0	0	0	0	0	0


En la matriz de juego (izquierda) se observan los códigos de las casillas del tablero. El mapa de marcas está todo a ceros, de momento. Se han resaltado las posiciones de las combinaciones en magenta para indicar dónde se deben producir las detecciones.

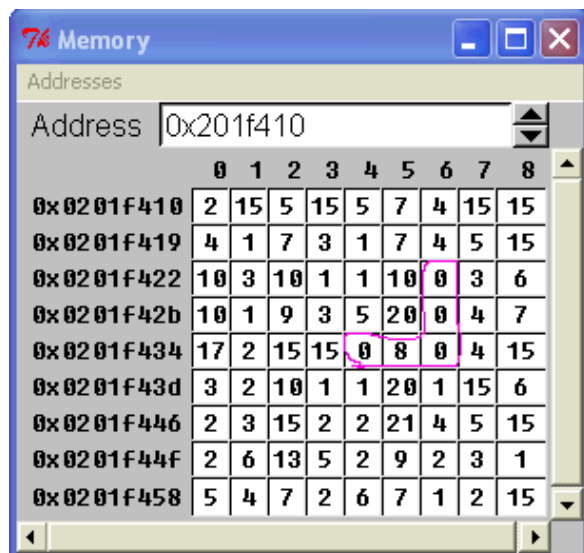
Dado que ya hemos establecido otro *breakpoint* después de la llamada a la rutina `marcar_verticales()`, podemos indicar al *Insight / gdb* que continúe la ejecución

hasta ese punto. Cuando se vuelva a parar la ejecución, la matriz de marcas quedará como sigue:



Addresses	0	1	2	3	4	5	6	7	8
0x0201f46c	0	0	0	0	0	0	0	0	0
0x0201f475	0	0	0	0	0	0	0	0	0
0x0201f47e	0	0	0	0	0	0	1	0	0
0x0201f487	0	0	0	0	0	0	1	0	0
0x0201f490	0	0	0	0	1	1	1	0	0
0x0201f499	0	0	0	0	0	0	0	0	0
0x0201f4a2	0	0	0	0	0	0	0	0	0
0x0201f4ab	0	0	0	0	0	0	0	0	0
0x0201f4b4	0	0	0	0	0	0	0	0	0

Ahora se podría continuar ejecutando el programa paso a paso, para observar como se eliminan los elementos de las casillas marcadas. El proceso puede resultar tedioso, ya que hay que recorrer una matriz de 9x9 posiciones (81 iteraciones). Como alternativa a fijar un nuevo *breakpoint*, podemos utilizar el botón de continuar hasta el final de la rutina actual , de modo que el resultado sobre la matriz de juego quedaría del siguiente modo:




Addresses	0	1	2	3	4	5	6	7	8
0x0201f410	2	15	5	15	5	7	4	15	15
0x0201f419	4	1	7	3	1	7	4	5	15
0x0201f422	10	3	10	1	1	10	0	3	6
0x0201f42b	10	1	9	3	5	20	0	4	7
0x0201f434	17	2	15	15	0	8	0	4	15
0x0201f43d	3	2	10	1	1	20	1	15	6
0x0201f446	2	3	15	2	2	21	4	5	15
0x0201f44f	2	6	13	5	2	9	2	3	1
0x0201f458	5	4	7	2	6	7	1	2	15

En definitiva, la combinación del depurador con el emulador permite analizar el funcionamiento interno del programa con gran precisión, lo cual resulta muy útil para detectar y corregir los errores que, inevitablemente, surgen en el desarrollo de cualquier proyecto.

3.3 Problemas con el *Insight*

Aunque el uso combinado del *Insight / gdb* y el *DeSmuME_dev* es un sistema muy potente para encontrar fallos en un programa para NDS, este entorno presenta algunas problemáticas que vamos a comentar a continuación, junto con sus posibles soluciones:

<i>Problema</i>	<i>Posible solución</i>
A veces no se establece correctamente la comunicación entre el <i>Insight / gdb</i> y el <i>DeSmuME_dev</i>	Además de verificar que el puerto de comunicaciones es correcto, se puede probar de restablecer la comunicación y la carga del programa con las opciones 'Connect to target', 'Download' (descargar programa) y 'Run' del menú ' Run '; en caso negativo, hay un enlace en el <i>Moodle</i> que explica cómo establecer la comunicación directamente mediante comandos <i>gdb</i> .
No es posible acceder a una rutina de un fichero escrito en ensamblador, o no es posible fijar <i>breakpoints</i> directamente sobre la ventana principal del <i>Insight</i>	Si se visualiza el programa en modo ' SRC + ASM ', es posible ver todas las instrucciones y fijar los <i>breakpoints</i> en los puntos del programa en la zona de ensamblador. Alternativamente, siempre se puede abrir la consola del <i>gdb</i> y utilizar el comando ' <i>break</i> '
La ejecución paso a paso no funciona correctamente porque, a veces, salta dos instrucciones a la vez	No es exactamente un error, sino que es un problema de sincronización <i>Insight / gdb</i> y <i>DeSmuME_dev</i> : no es posible evitar este problema.

Finalmente, hay que remarcar que el depurador *Insight* solo es una interfaz gráfica del depurador *gdb*, que se maneja mediante línea de comandos. Aunque resulte costoso aprender comandos *gdb*, a veces es conveniente saber algunos comandos básicos para manejar el proceso desde la consola () , como los siguientes:

```
(gdb) target sim           // se conecta al simulador incorporado del gdb
(gdb) target remote :1000  // se conecta al puerto TCP 1000 local
(gdb) load                 // cargar el código ejecutable desde fichero
(gdb) break *address       // fija un breakpoint en una dirección de memoria
(gdb) break function       // fija un breakpoint al inicio de una función
(gdb) break file:line      // fija un breakpoint en una línea de un fichero
(gdb) delete breakpoints   // elimina uno o varios breakpoints
(gdb) step                 // ejecuta una instrucción
(gdb) next                 // ejecuta una instrucción o una llamada a rutina
(gdb) nexti                // ejecuta una instrucción de Lenguaje Máquina
(gdb) continue             // continúa la ejecución del programa
(gdb) finish               // termina la ejecución de la rutina actual
(gdb) info breakpoints     // muestra todos los breakpoints
(gdb) help                 // ayuda sobre comandos gdb
```