

Ejercicios resueltos de Programación Multimedia y Dispositivos Móviles

Los 15 ejercicios de Programación Multimedia y Dispositivos Móviles de la web, tema a tema: cada uno con su enunciado, los datos que necesitas (código de partida, ejemplos de entrada y salida o la base de datos) y, al final, la solución explicada.

Índice

Tecnologías y Plataformas Móviles

- | | |
|--|--|
| <u>1. Densidades de pantalla y recursos de Android</u> | <u>2. El ciclo de vida de una Activity</u> |
| <u>3. Elegir el minSdk de una app</u> | |

Desarrollo de Aplicaciones Móviles

- | | |
|---|--|
| <u>4. Las diferencias de una lista, como DiffUtil</u> | <u>5. La pila de navegación de una app</u> |
| <u>6. Sincronización sin conexión con conflictos</u> | |

Motores de Juegos

- | | |
|---|---|
| <u>7. Física con paso fijo, acumulador e interpolación</u> | <u>8. Colisiones con rejilla espacial</u> |
| <u>9. El efecto túnel: detección discreta frente a continua</u> | |

Ejercicios de Programación Multimedia y Dispositivos Móviles

- | | |
|---|--|
| <u>10. ¿Quién gana al tres en raya?</u> | <u>11. Una generación del juego de la vida</u> |
| <u>12. Puntuación con combos</u> | <u>13. Camino más corto en un laberinto</u> |

Ejercicios largos

- | | |
|---|---|
| <u>14. Física de un juego de plataformas: gravedad, saltos y colisiones</u> | <u>15. Inteligencia artificial de un enemigo con una máquina de estados</u> |
|---|---|

Tecnologías y Plataformas Móviles

1. Densidades de pantalla y recursos de Android

Medio · Tecnologías y Plataformas Móviles · apuntesdam.com/subject/prog-multimedia/topic/tecnologias-plataformas-movil

Android agrupa las pantallas por densidad para escoger imágenes y por anchura mínima (smallest width) para escoger diseños. Cada línea es un dispositivo «nombre anchoPx altoPx pulgadas». Calcula su densidad real (diagonal en píxeles / pulgadas), el grupo de densidad más cercano (ldpi 120, mdpi 160, hdpi 240, xhdpi 320, xxhdpi 480, xxxhdpi 640; ante empate, el menor), el factor de escala (dpi del grupo / 160), su tamaño en dp (lado corto y lado largo divididos por el factor, redondeados), si es «móvil», «tableta» (lado corto ≥ 600 dp) o «tableta grande» (≥ 720 dp) con su carpeta de diseños (layout, layout-sw600dp o layout-sw720dp) y cuántos píxeles mide un icono de 48 dp. Formato: «nombre: D dpi $\rightarrow$ grupo ($\times F$), $A \times L$ dp, tipo $\rightarrow$ carpeta, icono 48 dp = N px», con el factor sin «.0» si es entero ($\times 1.5$, $\times 2$).

Código de partida (java)

```
import java.util.*;

public class Main {
    static final String[] NOMBRES = {"ldpi", "mdpi", "hdpi", "xhdpi", "xxhdpi", "xxxhdpi"};
    static final int[] DPI = {120, 160, 240, 320, 480, 640};

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" "); // nombre anchoPx altoPx pulgadas
            if (p.length < 4) continue;
            // TODO: calcula densidad, tamaño en dp y carpeta de recursos
        }
    }
}
```

Ejemplo: Móviles y tableta

ENTRADA

```
Pixel7 1080 2400 6.3
GalaxyTabS9 1600 2560 11
MovilBarato 720 1600 6.5
```

SALIDA ESPERADA

```
Pixel7: 418 dpi  $\rightarrow$  xxhdpi ( $\times 3$ ), 360  $\times$  800 dp,
móvil  $\rightarrow$  layout, icono 48 dp = 144 px
GalaxyTabS9: 274 dpi  $\rightarrow$  hdpi ( $\times 1.5$ ), 1067  $\times$  1707
dp, tableta grande  $\rightarrow$  layout-sw720dp, icono 48 dp
= 72 px
MovilBarato: 270 dpi  $\rightarrow$  hdpi ( $\times 1.5$ ), 480  $\times$  1067
dp, móvil  $\rightarrow$  layout, icono 48 dp = 72 px
```

2. El ciclo de vida de una Activity

Difícil · Tecnologías y Plataformas Móviles · apuntesdam.com/subject/prog-multimedia/topic/tecnologias-plataformas-movil

Entender cuándo llama Android a cada método del ciclo de vida evita perder datos. Cada línea es un evento; simula las llamadas y el estado de la pantalla con un campo de texto: «abrir» (onCreate $\rightarrow$ onStart $\rightarrow$ onResume), «escribir TEXTO» (el usuario escribe, sin llamadas), «dialogo» (un diálogo encima: onPause si estaba reanudada) y «cerrar_dialogo» (onResume si estaba pausada), «home» (onPause si estaba reanudada, onStop, onSaveInstanceState: se guarda el texto), «volver» (si estaba detenida: onRestart $\rightarrow$ onStart $\rightarrow$ onResume), «girar» (se destruye y se recrea: onPause $\rightarrow$ onStop $\rightarrow$ onSaveInstanceState $\rightarrow$ onDestroy $\rightarrow$ onCreate $\rightarrow$ onStart $\rightarrow$ onRestoreInstanceState $\rightarrow$ onResume, y el texto se recupera de lo guardado), «matar_proceso» (si estaba detenida, el sistema elimina el proceso: «(proceso eliminado sin más llamadas)», estado NINGUNO y el texto en memoria se pierde), «reabrir» (onCreate $\rightarrow$ onStart $\rightarrow$ onRestoreInstanceState $\rightarrow$ onResume, recuperando lo guardado) y «atras»

(onPause si estaba reanudada, onStop, onDestroy: estado DESTRUIDA y no se guarda nada). Muestra «evento: llamadas separadas por → | estado ESTADO | texto «texto»» (para «escribir», la llamada es «(el usuario escribe «texto»)»); para un evento desconocido, «(evento desconocido)»).

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // Estados: NINGUNO, CREADA, INICIADA, REANUDADA, PAUSADA, DETENIDA, DESTRUIDA
        // TODO: simula el ciclo de vida de una Activity ante cada evento
    }
}
```

Ejemplo: Uso normal

ENTRADA

```
abrir
escribir Hola
dialogo
cerrar_dialogo
home
volver
atras
```

SALIDA ESPERADA

```
abrir: onCreate → onStart → onResume | estado
REANUDADA | texto «»
escribir: (el usuario escribe «Hola») | estado
REANUDADA | texto «Hola»
dialogo: onPause | estado PAUSADA | texto «Hola»
cerrar_dialogo: onResume | estado REANUDADA |
texto «Hola»
home: onPause → onStop → onSaveInstanceState |
estado DETENIDA | texto «Hola»
volver: onRestart → onStart → onResume | estado
REANUDADA | texto «Hola»
atras: onPause → onStop → onDestroy | estado
DESTRUIDA | texto «»
```

Ejemplo: Giro y proceso eliminado

ENTRADA

```
abrir
escribir Pedido 42
girar
home
matar_proceso
reabrir
```

SALIDA ESPERADA

```
abrir: onCreate → onStart → onResume | estado
REANUDADA | texto «»
escribir: (el usuario escribe «Pedido 42») |
estado REANUDADA | texto «Pedido 42»
girar: onPause → onStop → onSaveInstanceState →
onDestroy → onCreate → onStart →
onRestoreInstanceState → onResume | estado
REANUDADA | texto «Pedido 42»
home: onPause → onStop → onSaveInstanceState |
estado DETENIDA | texto «Pedido 42»
matar_proceso: (proceso eliminado sin más
llamadas) | estado NINGUNO | texto «»
reabrir: onCreate → onStart →
onRestoreInstanceState → onResume | estado
REANUDADA | texto «Pedido 42»
```

3. Elegir el minSdk de una app

Muy difícil · Tecnologías y Plataformas Móviles · apuntesdam.com/subject/prog-multimedia/topic/tecnologias-plataformas-movil

Subir el minSdk permite usar APIs más nuevas sin comprobaciones, pero deja fuera a quienes tienen móviles antiguos. La entrada tiene líneas «cuota NIVEL PORCENTAJE» (dispositivos activos con cada nivel de API; puede que no sumen 100, así que hay que normalizar), «api NOMBRE NIVEL» (APIs que usa la app y el nivel en que aparecieron), «minSdk N» y «objetivo P». Muestra «minSdk N: X % de los dispositivos» (los de nivel $\geq N$, 1 decimal); el minSdk más alto que aún cubre al menos el objetivo: «Recomendado para el P %: minSdk M (X %)» (P sin decimales) o «Ningún minSdk

alcanza el P %»; y para cada API, «NOMBRE: disponible siempre (API L)» si $L \leq \text{minSdk}$, o «NOMBRE: comprobar Build.VERSION.SDK_INT $\geq L$ (no disponible en el X % de los dispositivos)» con el porcentaje de dispositivos entre minSdk y L - 1.

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        TreeMap<Integer, Double> cuota = new TreeMap<>();           // nivel de API → % de dispositivos
        List<String[]> apis = new ArrayList<>();                   // nombre nivelMínimo
        int minSdk = 0;
        double objetivo = 0;
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            switch (p[0]) {
                case "cuota" -> cuota.put(Integer.parseInt(p[1]), Double.parseDouble(p[2]));
                case "api" -> apis.add(new String[]{p[1], p[2]});
                case "minSdk" -> minSdk = Integer.parseInt(p[1]);
                case "objetivo" -> objetivo = Double.parseDouble(p[1]);
                default -> { }
            }
        }
        // TODO: cobertura del minSdk, minSdk recomendado y comprobaciones de versión necesarias
    }
}
```

Ejemplo: App de pagos

ENTRADA

```
cuota 24 2.0
cuota 26 4.0
cuota 28 6.5
cuota 29 9.0
cuota 30 13.5
cuota 31 15.0
cuota 33 20.0
cuota 34 18.0
cuota 35 12.0
api Notificaciones 26
api PhotoPicker 33
api PermisoNotificaciones 33
minSdk 26
objetivo 95
```

SALIDA ESPERADA

```
minSdk 26: 98.0 % de los dispositivos
Recomendado para el 95 %: minSdk 26 (98.0 %)
Notificaciones: disponible siempre (API 26)
PhotoPicker: comprobar Build.VERSION.SDK_INT >=
33 (no disponible en el 48.0 % de los
dispositivos)
PermisoNotificaciones: comprobar
Build.VERSION.SDK_INT >= 33 (no disponible en el
48.0 % de los dispositivos)
```

Desarrollo de Aplicaciones Móviles

4. Las diferencias de una lista, como DiffUtil

Medio · Desarrollo de Aplicaciones Móviles · apuntesdam.com/subject/prog-multimedia/topic/desarrollo-aplicaciones-movil

Para no redibujar una lista entera (RecyclerView o LazyColumn) cada vez que cambian los datos, Android calcula las diferencias entre la lista vieja y la nueva (DiffUtil, ListAdapter). La primera línea es la lista antigua y la segunda la nueva, como elementos «id:contenido» separados por espacios (cualquiera puede estar vacía). Muestra, primero, «notifyItemRemoved: id» por cada elemento que ya no está (en el orden de la lista vieja); después, recorriendo la lista nueva, «notifyItemInserted(posición): id» para los nuevos y «notifyItemChanged(posición): id viejo → nuevo» para los que cambian de contenido. Termina con «Sin cambios: A · Insertados: B · Borrados: C · Modificados: D» (añadiendo « ·

Hay elementos movidos» si los elementos que siguen en ambas listas cambian de orden relativo) y con «No hace falta redibujar» o «Se redibujan N de M filas (notifyDataSetChanged redibujaría M)», donde N son insertados más modificados.

Código de partida (java)

```
import java.util.*;

public class Main {
    record Elemento(String id, String contenido) { }

    static List<Elemento> leer(String linea) {
        List<Elemento> l = new ArrayList<>();
        if (linea.isBlank()) return l;
        for (String par : linea.trim().split(" ")) {
            String[] p = par.split(":");
            l.add(new Elemento(p[0], p[1]));
        }
        return l;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<Elemento> antes = leer(sc.nextLine()), despues = leer(sc.hasNextLine() ? sc.nextLine() :
            "");
        // TODO: calcula los cambios entre las dos listas como DiffUtil
    }
}
```

Ejemplo: Lista de tareas

ENTRADA

```
1:Pan 2:Leche 3:Huevos 4:Fruta
1:Pan 3:Huevos(12) 5:Café 4:Fruta
```

SALIDA ESPERADA

```
notifyItemRemoved: 2
notifyItemChanged(1): 3 Huevos → Huevos(12)
notifyItemInserted(2): 5
Sin cambios: 2 · Insertados: 1 · Borrados: 1 ·
Modificados: 1
Se redibujan 2 de 4 filas (notifyDataSetChanged
redibujaría 4)
```

Ejemplo: Reordenada

ENTRADA

```
a:x b:y c:z
c:z a:x b:y
```

SALIDA ESPERADA

```
Sin cambios: 3 · Insertados: 0 · Borrados: 0 ·
Modificados: 0 · Hay elementos movidos
Se redibujan 0 de 3 filas (notifyDataSetChanged
redibujaría 3)
```

Ejemplo: Sin cambios

ENTRADA

```
a:1 b:2
a:1 b:2
```

SALIDA ESPERADA

```
Sin cambios: 2 · Insertados: 0 · Borrados: 0 ·
Modificados: 0
No hace falta redibujar
```

5. La pila de navegación de una app

Difícil · Desarrollo de Aplicaciones Móviles · apuntesdam.com/subject/prog-multimedia/topic/desarrollo-aplicaciones-movil

El componente Navigation de Android mantiene una pila de pantallas (back stack). La primera línea es «inicio DESTINO»; después vienen órdenes «navigate DESTINO [popUpTo=X] [inclusive] [singleTop]» y «back». navigate apila el destino; con popUpTo=X, antes se desapilan las pantallas hasta dejar X en la cima (con inclusive, también X), lo que

se usa por ejemplo tras iniciar sesión para que «atrás» no vuelva al acceso; con singleTop, si el destino ya está en la cima no se apila otra vez («orden: ya estás en DESTINO (singleTop)»). Si X no está en la pila, «ERROR: X no está en la pila» y no se hace nada. back desapila; si solo queda una pantalla, «back: se cierra la app desde DESTINO» y se termina. Tras cada orden que cambia la pila (y tras un back que no cierra), muestra la orden y «→ [base > ... > cima]».

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Deque<String> pila = new ArrayDeque<>(); // la cima es la pantalla visible
        pila.push(sc.nextLine().trim().split(" ")[1]); // inicio DESTINO
        // TODO: procesa navigate y back como el componente Navigation
    }
}
```

Ejemplo: Inicio de sesión

ENTRADA

```
inicio splash
navigate login popUpTo=splash inclusive
navigate inicio popUpTo=login inclusive
navigate detalle
navigate detalle singleTop
navigate carrito
back
back
back
```

SALIDA ESPERADA

```
navigate login popUpTo=splash inclusive → [login]
navigate inicio popUpTo=login inclusive →
[inicio]
navigate detalle → [inicio > detalle]
navigate detalle singleTop: ya estás en detalle
(singleTop)
navigate carrito → [inicio > detalle > carrito]
back → [inicio > detalle]
back → [inicio]
back: se cierra la app desde inicio
```

6. Sincronización sin conexión con conflictos

Muy difícil · Desarrollo de Aplicaciones Móviles · apuntesdam.com/subject/prog-multimedia/topic/desarrollo-aplicaciones-movil

Una app «offline first» guarda los cambios hechos sin conexión y los envía al recuperarla. La entrada tiene tres secciones: [servidor] con el estado actual «id valor tiempo», [base] con la versión que tenía el móvil cuando se quedó sin conexión, y [cambios] con las operaciones pendientes «crear id valor tiempo», «editar id valor tiempo» o «borrar id tiempo» (los valores no tienen espacios). Para cada cambio, en orden: crear algo que ya existe → «CONFLICTO id: ya existe en el servidor (se conserva «valor»)»; editar o borrar algo que ya no existe → «IGNORADO id: ya no existe en el servidor»; si el elemento ha cambiado en el servidor desde la base (distinto tiempo) y el cambio del servidor es más reciente → «CONFLICTO id: gana el servidor «valor» (tiempo) frente al cambio local (tiempo)» (estrategia «gana el último»); en otro caso se aplica: «CREADO id = «valor»», «BORRADO id» o «EDITADO id = «valor»», añadiendo «(gana el cambio local, más reciente)» o «(sobre un cambio del servidor más antiguo)» si había cambio en el servidor. Termina con «Servidor tras sincronizar:», cada elemento « id = «valor» (tiempo)» por orden de id y «Aplicados: A · Conflictos: C».

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, String[]> servidor = new TreeMap<>(), local = new TreeMap<>(); // id → {valor,
        marca de tiempo}
        List<String[]> pendientes = new ArrayList<>(); // cambios hechos sin
        conexión
        String seccion = "";
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.isEmpty()) continue;
            if (l.startsWith("[") ) { seccion = l; continue; }
            String[] p = l.split(" ");
            switch (seccion) {
                case "[servidor]" -> servidor.put(p[0], new String[]{p[1], p[2]});
                case "[base]" -> local.put(p[0], new String[]{p[1], p[2]});
                case "[cambios]" -> pendientes.add(p); // op id valor tiempo (op: crear,
                editar, borrar)
                default -> { }
            }
        }
        // TODO: aplica los cambios pendientes al servidor resolviendo conflictos y muestra el resultado
    }
}
```

Ejemplo: Tareas

ENTRADA

```
[servidor]
t1 Comprar_pan 100
t2 Llamar_al_banco 250
t3 Entregar_la_práctica 120
[base]
t1 Comprar_pan 100
t2 Llamar_al_banco 110
t3 Entregar_la_práctica 120
[cambios]
editar t1 Comprar_pan_integral 300
editar t2 Llamar_a_las_9 200
crear t4 Recoger_paquete 210
borrar t3 220
crear t1 Duplicada 230
```

SALIDA ESPERADA

```
EDITADO t1 = «Comprar_pan_integral»
CONFLICTO t2: gana el servidor «Llamar_al_banco»
(250) frente al cambio local (200)
CREADO t4 = «Recoger_paquete»
BORRADO t3
CONFLICTO t1: ya existe en el servidor (se
conserva «Comprar_pan_integral»)
Servidor tras sincronizar:
    t1 = «Comprar_pan_integral» (300)
    t2 = «Llamar_al_banco» (250)
    t4 = «Recoger_paquete» (210)
Aplicados: 3 · Conflictos: 2
```

Motores de Juegos

7. Física con paso fijo, acumulador e interpolación

Medio · Motores de Juegos · apuntesdam.com/subject/prog-multimedia/topic/motores-juegos

Los motores dibujan a la frecuencia que pueden, pero calculan la física en pasos fijos (FixedUpdate en Unity, `_physics_process` en Godot). La entrada tiene «paso P» (ms de cada paso de física), «velocidad V» (px/s de un objeto que avanza en x desde 0) y una línea con la duración en ms de cada fotograma. Para cada fotograma: suma su duración a un acumulador (como máximo 250 ms por fotograma, para que un parón no obligue a simular cientos de pasos), ejecuta pasos de física mientras el acumulador llegue a P (guardando la posición anterior y avanzando $V \cdot P / 1000$) y dibuja interpolando entre la posición anterior y la actual con $\alpha = \text{acumulador} / P$. Muestra «Fi dt=D[(limitado a 250)]: N paso(s), sobran A ms, x=X, se dibuja en Y (alfa Z)» con un decimal en las posiciones y dos en alfa, y al final «Fotogramas: F · Pasos de física: N · Tiempo simulado: T ms».

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int paso = Integer.parseInt(sc.nextLine().trim().split(" ")[1]); // «paso 20» (ms)
        double velocidad = Double.parseDouble(sc.nextLine().trim().split(" ")[1]); // «velocidad 120»
        (px/s)
        String[] deltas = sc.nextLine().trim().split(" "); // ms de cada
        fotograma
        final int MAXIMO = 250; // evita la
        «espiral de la muerte»
        double x = 0, xAnterior = 0;
        int acumulado = 0, totalPasos = 0;
        // TODO: bucle con paso fijo, acumulador e interpolación
    }
}
```

Ejemplo: 60 FPS con tirones

ENTRADA

```
paso 20
velocidad 120
16 16 17 16 50 16 300 8
```

SALIDA ESPERADA

```
F1 dt=16: 0 pasos, sobran 16 ms, x=0.0, se dibuja
en 0.0 (alfa 0.80)
F2 dt=16: 1 paso, sobran 12 ms, x=2.4, se dibuja
en 1.4 (alfa 0.60)
F3 dt=17: 1 paso, sobran 9 ms, x=4.8, se dibuja
en 3.5 (alfa 0.45)
F4 dt=16: 1 paso, sobran 5 ms, x=7.2, se dibuja
en 5.4 (alfa 0.25)
F5 dt=50: 2 pasos, sobran 15 ms, x=12.0, se
dibuja en 11.4 (alfa 0.75)
F6 dt=16: 1 paso, sobran 11 ms, x=14.4, se dibuja
en 13.3 (alfa 0.55)
F7 dt=300 (limitado a 250): 13 pasos, sobran 1
ms, x=45.6, se dibuja en 43.3 (alfa 0.05)
F8 dt=8: 0 pasos, sobran 9 ms, x=45.6, se dibuja
en 44.3 (alfa 0.45)
Fotogramas: 8 · Pasos de física: 19 · Tiempo
simulado: 380 ms
```

8. Colisiones con rejilla espacial

Difícil · Motores de Juegos · apuntesdam.com/subject/prog-multimedia/topic/motores-juegos

Comprobar todos los objetos contra todos cuesta $n \cdot (n-1)/2$ comprobaciones: con 1000 balas y enemigos, medio millón por fotograma. Los motores primero descartan los pares lejanos (fase amplia) con estructuras como una rejilla espacial. La primera línea es «celda C» y las siguientes cajas «nombre x y ancho alto» (enteros, y hacia abajo). Calcula los choques por fuerza bruta (AABB, ya dado en chocan) y con la rejilla: cada caja se apunta en todas las celdas que ocupa, de $\text{floorDiv}(x, C)$ a $\text{floorDiv}(x+\text{ancho}-1, C)$ en x e igual en y; solo se comprueban los pares distintos que comparten al menos una celda. Muestra «Celdas ocupadas: K · La más llena: (cx,cy) con M» (la primera creada si hay empate), «Choques: a-b c-d...» (cada par en el orden de entrada, ordenados por la primera caja y luego por la segunda; «ninguno» si no hay), «Comprobaciones: fuerza bruta N · rejilla R (ahorro P %)» (P redondeado; 0 si N es 0) y «Mismo resultado: sí».

Código de partida (java)

```
import java.util.*;

public class Main {
    record Caja(String nombre, int x, int y, int ancho, int alto) { }

    static boolean chocan(Caja a, Caja b) {
        return a.x() < b.x() + b.ancho() && a.x() + a.ancho() > b.x()
            && a.y() < b.y() + b.alto() && a.y() + a.alto() > b.y();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int celda = Integer.parseInt(sc.nextLine().trim().split(" ")[1]); // «celda 64»
        List<Caja> cajas = new ArrayList<>();
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            if (p.length < 5) continue;
            cajas.add(new Caja(p[0], Integer.parseInt(p[1]), Integer.parseInt(p[2]),
                Integer.parseInt(p[3]), Integer.parseInt(p[4])));
        }
        // TODO: fuerza bruta frente a rejilla espacial
    }
}
```

Ejemplo: Nivel de plataformas

ENTRADA

```
celda 64
jugador 40 300 24 32
suelo 0 330 640 20
moneda1 60 300 12 12
moneda2 400 200 12 12
enemigo 380 196 30 30
bala 600 40 6 6
plataforma 360 226 120 16
```

SALIDA ESPERADA

```
Celdas ocupadas: 16 · La más llena: (6,3) con 3
Choques: jugador-suelo jugador-moneda1 moneda2-
enemigo
Comprobaciones: fuerza bruta 21 · rejilla 5
(ahorro 76 %)
Mismo resultado: sí
```

Ejemplo: Coordenadas negativas

ENTRADA

```
celda 50
a -30 -30 40 40
b 0 0 10 10
c 200 200 10 10
```

SALIDA ESPERADA

```
Celdas ocupadas: 5 · La más llena: (0,0) con 2
Choques: a-b
Comprobaciones: fuerza bruta 3 · rejilla 1
(ahorro 67 %)
Mismo resultado: sí
```

9. El efecto túnel: detección discreta frente a continua

Muy difícil · Motores de Juegos · apuntesdam.com/subject/prog-multimedia/topic/motores-juegos

Si una bala avanza 25 píxeles por fotograma y la pared mide 10, puede estar delante en un fotograma y detrás en el siguiente sin que nunca se detecte el choque (efecto túnel). Por eso los motores ofrecen detección continua (Continuous en el Rigidbody de Unity, CCD en Godot). La entrada tiene «pared x y ancho alto», «fps F» y balas «nombre x y vx vy» (velocidades en px/s, y hacia abajo). Simula 1 segundo. Discreta: en cada fotograma $k = 1 \dots F$ la bala está en $(x + vx \cdot k/F, y + vy \cdot k/F)$ y choca si está estrictamente dentro de la pared. Continua: calcula con el método de las franjas el instante $t \in [0, 1]$ en que el segmento recorrido entra en la pared (si la velocidad en un eje es 0, la bala debe estar estrictamente dentro de la franja de ese eje; si no, no hay impacto). Muestra «nombre: discreta → choca en el fotograma K ($t=T$ s) | no detecta nada | continua → impacto en $t=T$ s en (X, Y) por la cara izquierda/derecha/superior/inferior | no hay impacto | empieza dentro de la pared», añadiendo « ¡ATRAVIESA LA PARED!» si la continua ve un impacto (sin empezar dentro) que la discreta no detecta; al final «Efecto túnel: N de M balas». Tiempos con tres decimales y posiciones con uno.

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String[] m = sc.nextLine().trim().split(" "); // «pared x y ancho alto»
        double px = Double.parseDouble(m[1]), py = Double.parseDouble(m[2]);
        double pw = Double.parseDouble(m[3]), ph = Double.parseDouble(m[4]);
        int fps = Integer.parseInt(sc.nextLine().trim().split(" ")[1]); // «fps 60»
        int tuneles = 0, balas = 0;
        while (sc.hasNextLine()) {
            String[] b = sc.nextLine().trim().split(" ");
            if (b.length < 5) continue;
            String nombre = b[0];
            double x0 = Double.parseDouble(b[1]), y0 = Double.parseDouble(b[2]);
            double vx = Double.parseDouble(b[3]), vy = Double.parseDouble(b[4]);
            balas++;
            // TODO: detección discreta frente a continua (barrido)
        }
        System.out.println("Efecto túnel: " + tuneles + " de " + balas + " balas");
    }
}
```

Ejemplo: Pared fina

ENTRADA

```
pared 300 0 10 200
fps 60
lenta 5 100 600 0
rapida 0 50 1500 0
vertical 305 -50 0 400
diagonal 100 250 600 -300
falla 0 300 900 0
```

SALIDA ESPERADA

```
lenta: discreta → choca en el fotograma 30
(t=0.500 s) | continua → impacto en t=0.492 s en
(300.0, 100.0) por la cara izquierda
rapida: discreta → no detecta nada | continua →
impacto en t=0.200 s en (300.0, 50.0) por la cara
izquierda ¡ATRAVIESA LA PARED!
vertical: discreta → choca en el fotograma 8
(t=0.133 s) | continua → impacto en t=0.125 s en
(305.0, 0.0) por la cara superior
diagonal: discreta → no detecta nada | continua →
impacto en t=0.333 s en (300.0, 150.0) por la
cara izquierda ¡ATRAVIESA LA PARED!
falla: discreta → no detecta nada | continua → no
hay impacto
Efecto túnel: 2 de 5 balas
```

Ejercicios de Programación Multimedia y Dispositivos Móviles

10. ¿Quién gana al tres en raya?

Fácil · Ejercicios de Programación Multimedia y Dispositivos Móviles · apuntesdam.com/subject/programacion-multimedia/topic/ejercicios-pmdm

La entrada son 3 líneas con el tablero (X, O o . para las casillas vacías). Muestra «Gana X», «Gana O», «Empate» (tablero lleno sin ganador) o «Sigue la partida».

Código de partida (java)

```
import java.util.*;

public class Main {

    static String resultado(char[][] t) {
        // TODO
        return "Sigue la partida";
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        char[][] t = new char[3][];
        for (int i = 0; i < 3; i++) t[i] = sc.next().toCharArray();
        System.out.println(resultado(t));
    }
}
```

Ejemplo: Gana X en diagonal

ENTRADA

X.O
OX.
..X

SALIDA ESPERADA

Gana X

Ejemplo: Empate

ENTRADA

XOX
XOO
OXX

SALIDA ESPERADA

Empate

Ejemplo: Sigue

ENTRADA

X..
.O.
...

SALIDA ESPERADA

Sigue la partida

11. Una generación del juego de la vida

Medio · Ejercicios de Programación Multimedia y Dispositivos Móviles · apuntesdam.com/subject/prog-multimedia/topic/ejercicios-pmdm

La primera línea tiene filas y columnas; después, el tablero con # (célula viva) y . (muerta). Calcula la siguiente generación del juego de la vida de Conway y muéstrala: una célula viva con 2 o 3 vecinas vivas sobrevive; una muerta con exactamente 3 vecinas vivas nace; el resto muere o sigue muerta. Los bordes no se conectan.

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int f = sc.nextInt(), c = sc.nextInt();
        char[][] t = new char[f][];
        for (int i = 0; i < f; i++) t[i] = sc.next().toCharArray();
        // TODO: calcula y muestra la siguiente generación
        for (char[] fila : t) System.out.println(new String(fila));
    }
}
```

Ejemplo: Parpadeador

ENTRADA

```
5 5
.....
..#..
..#..
..#..
.....
```

SALIDA ESPERADA

```
.....
.....
.###.
.....
.....
```

Ejemplo: Bloque estable

ENTRADA

```
4 4
....
.##.
.##.
....
```

SALIDA ESPERADA

```
....
.##.
.##.
....
```

12. Puntuación con combos

Medio · Ejercicios de Programación Multimedia y Dispositivos Móviles · apuntesdam.com/subject/prog-multimedia/topic/ejercicios-pmdm

Por la entrada llega la secuencia de eventos de una partida, uno por línea: MONEDA (10 puntos), ENEMIGO (50 puntos), GOLPE (pierdes una vida y el combo vuelve a 0). Cada MONEDA o ENEMIGO seguidos aumentan el combo en 1 y los puntos del evento se multiplican por el multiplicador actual: 1 con combo 0-4, 2 con combo 5-9 y 3 con combo 10 o más (el combo se incrementa antes de calcular). Empiezas con 3 vidas; con 0 vidas la partida termina y se ignoran los eventos siguientes. Muestra «Puntos: N», «Vidas: V» y «Mejor combo: C».

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int puntos = 0, vidas = 3, combo = 0, mejor = 0;
        while (sc.hasNext()) {
            String evento = sc.next();
            // TODO
        }
        System.out.println("Puntos: " + puntos);
        System.out.println("Vidas: " + vidas);
        System.out.println("Mejor combo: " + mejor);
    }
}
```

Ejemplo: Combo hasta x2

ENTRADA

```
MONEDA
MONEDA
MONEDA
MONEDA
MONEDA
ENEMIGO
```

SALIDA ESPERADA

```
Puntos: 160
Vidas: 3
Mejor combo: 6
```

Ejemplo: Golpe reinicia el combo

ENTRADA

ENEMIGO
GOLPE
MONEDA

SALIDA ESPERADA

Puntos: 60
Vidas: 2
Mejor combo: 1

13. Camino más corto en un laberinto

Difícil · Ejercicios de Programación Multimedia y Dispositivos Móviles · apuntesdam.com/subject/prog-multimedia/topic/ejercicios-pmdm

La IA de un enemigo necesita saber cuántos pasos hay hasta el jugador. La primera línea tiene filas y columnas; después el mapa con # (muro), . (suelo), E (enemigo) y J (jugador). Moviéndose en 4 direcciones, muestra el número mínimo de pasos de E a J, o «Sin camino» si no se puede llegar.

Código de partida (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int f = sc.nextInt(), c = sc.nextInt();
        char[][] m = new char[f][];
        for (int i = 0; i < f; i++) m[i] = sc.next().toCharArray();
        // TODO: BFS desde E hasta J
        System.out.println("Sin camino");
    }
}
```

Ejemplo: Laberinto sencillo

ENTRADA

3 5
E..#.
.#...
...#J

SALIDA ESPERADA

6

Ejemplo: Camino directo

ENTRADA

1 4
E..J

SALIDA ESPERADA

3

Ejemplo: Sin camino

ENTRADA

3 3
E#.
##.
..J

SALIDA ESPERADA

Sin camino

Ejercicios largos

14. Física de un juego de plataformas: gravedad, saltos y colisiones

Muy difícil · JavaScript · 100 minutos · apuntesdam.com/ejercicios/javascript/fisica-de-un-juego-de-plataformas

Los juegos de plataformas 2D (de Super Mario a Celeste) se mueven con unas pocas reglas que se aplican en cada fotograma: leer los controles, mover al personaje, aplicar la gravedad y corregir su posición si ha atravesado una pared o el suelo. Motores como Unity o Godot traen esto hecho, pero entenderlo es lo que permite ajustar el «tacto» del juego y depurar los fallos de colisiones.

El nivel es un mapa de casillas de 16×16 píxeles: `#` es suelo o pared, `C` una moneda, `E` la meta, `P` la posición inicial del jugador y `.` aire. El jugador es una caja de 12×14 píxeles (una AABB, caja alineada con los ejes) y todo se calcula con números enteros, como en las consolas antiguas: así el resultado es exacto y repetible.

El truco clásico de las colisiones es mover y corregir cada eje por separado: primero en horizontal (si choca con una pared, se pega a ella) y después en vertical (si cae sobre el suelo, se queda encima y puede volver a saltar). Hacerlo con los dos ejes a la vez provoca que el personaje se enganche en las esquinas.

Requisitos

- La entrada es el mapa (filas del mismo ancho con `.`, `#`, `C`, `E` y un solo `P`), una línea `---` y las acciones separadas por espacios en una o varias líneas: `R` (derecha), `L` (izquierda), `J` (salto), combinaciones como `RJ`, `.` (nada) y repeticiones como `R*10`. Errores: Mapa no válido: falta el separador `---`, `...: las filas no miden lo mismo`, `...: carácter desconocido` y `...: tiene que haber un jugador (P)` (se comprueban en ese orden).
- El jugador empieza en $x = \text{columna} \times 16 + 2$ e $y = \text{fila} \times 16 + 2$ (la casilla de `P`, apoyado en su borde inferior), con velocidad vertical 0 y fuera del suelo.
- En cada fotograma, en este orden: (1) la velocidad horizontal es +2 con `R`, -2 con `L` (0 con las dos o ninguna); se suma a x , se limita al mapa (0 a ancho×16-12) y, si la caja toca alguna casilla `#`, se pega a ella (a su izquierda si iba hacia la derecha, a su derecha si no); (2) si hay `J` y está en el suelo, la velocidad vertical pasa a -9; después se le suma la gravedad (1) con un máximo de 8, se suma a y , y si toca un `#`: si bajaba, se coloca encima (y queda en el suelo); si subía, se coloca debajo; en los dos casos la velocidad vertical pasa a 0; (3) las monedas que toque la caja se recogen (una vez cada una), la meta termina el nivel y, si y llega a filas×16, el jugador ha caído.
- Una casilla «toca» la caja si se solapan: la caja ocupa de x a $x+11$ y de y a $y+13$. Fuera del mapa no hay suelo (se puede saltar por encima de la fila superior y caer por debajo de la inferior).
- Mensajes: Fotograma N: moneda (k/total) al recoger cada moneda; Fotograma N: ¡meta! Nivel superado con k/total monedas (y termina); Fotograma N: el jugador ha caído al vacío con k/total monedas (y termina). Si se acaban las acciones: Fin de las acciones (N fotogramas): posición (x, y), en el suelo|en el aire, k/total monedas.

Formato de la entrada

- El mapa, una línea `---` y las acciones (`R`, `L`, `J`, combinaciones, `.` y repeticiones con `*`).

Constantes del juego

Constante	Valor
Tamaño de casilla	16 px
Caja del jugador	12 × 14 px
Velocidad horizontal	2 px por fotograma
Impulso del salto	−9 px por fotograma
Gravedad	+1 px por fotograma ²
Velocidad máxima de caída	8 px por fotograma

Ejemplo: Recorrer el nivel saltando

ENTRADA

```
.....C.....
.....E.
.....####.
....C...####.
P.####.
#####
---
R*20 RJ R*12 . . R*24 RJ R*14 R*34 RJ R*16 R*30
```

SALIDA ESPERADA

```
Fotograma 42: moneda (1/2)
Fotograma 111: moneda (2/2)
Fotograma 146: ¡meta! Nivel superado con 2/2 monedas
```

Ejemplo: Mapas que no valen

ENTRADA

```
.....C.....
.....E.
.....####.
....C...####.
P.####.
#####
---
R*20 RJ R*12 . . R*24 RJ R*14 R*28 RJ R*16 R*30
```

SALIDA ESPERADA

```
Fotograma 42: moneda (1/2)
Fotograma 106: moneda (2/2)
Fin de las acciones (149 fotogramas): posición
(284, 66), en el suelo, 2/2 monedas
```

Código de partida (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const T = 16; // tamaño de casilla en píxeles
const ANCHO = 12, ALTO = 14; // caja del jugador
const VEL = 2, SALTO = -9, GRAVEDAD = 1, CAIDA_MAX = 8;

function main() {
  // TODO: lee y valida el mapa, lee las acciones y simula fotograma a fotograma
}

main();
```

15. Inteligencia artificial de un enemigo con una máquina de estados

Difícil · JavaScript · 80 minutos · apuntesdam.com/ejercicios/javascript/ia-enemiga-con-maquina-de-estados

Los enemigos de la mayoría de los videojuegos no piensan: siguen una máquina de estados. En cada momento están en un estado (patrullar, perseguir, atacar...) que decide qué hacen, y unas reglas de transición deciden cuándo cambian a otro. Es fácil de programar, de depurar y de ajustar, y por eso motores como Unity (con su Animator) y Godot la usan a todas horas.

Este guardia vigila un pasillo en un mapa de casillas. Patrulla entre dos puntos; si ve al jugador (a 5 casillas o menos y sin que el jugador vaya en sigilo) lo persigue; si lo tiene al lado, ataca; si lo pierde de vista, va al último sitio donde lo vio y espera un poco antes de volver a patrullar. Y si el jugador le baja la vida por debajo de 30, huye hasta poder recuperarse.

La simulación es por turnos: cada línea de la entrada dice dónde está el jugador en ese turno y si ha golpeado al guardia. El programa escribe cada cambio de estado con su motivo, que es exactamente el registro que se activa en un juego para depurar la IA.

Requisitos

- La primera línea es `PATRULLA x1,y1 x2,y2` (si no, La primera línea debe ser `PATRULLA x1,y1 x2,y2` y termina). El guardia empieza en el primer punto, en estado `PATRULLA`, con 100 de vida. Cada línea siguiente es un turno: `x,y` del jugador y, opcionalmente, `SIGILO` y `GOLPE n`. Las líneas vacías no son turnos; una línea con otra forma escribe `t=N: línea no válida` (y cuenta como turno).
- En cada turno: primero se resta el golpe (mínimo 0; si la vida llega a 0, `t=N: el enemigo cae derrotado` y la simulación termina). Después se calcula la distancia Manhattan al jugador: el guardia lo ve si es 5 o menos y no va en sigilo, y entonces recuerda su posición.
- Transiciones, se aplica solo la primera que se cumpla: cualquier estado salvo `HUIDA` → `HUIDA` si la vida es 30 o menos (`vida V`); `PATRULLA` → `PERSECUCION` si lo ve (`te ve a distancia D`); `PERSECUCION` → `ATAQUE` si la distancia es 1 o menos (`a tu lado`); `PERSECUCION` → `BUSCAR` si la distancia es mayor que 8 (`te pierde a distancia D`); `ATAQUE` → `PERSECUCION` si la distancia es mayor que 1 (`te alejas a distancia D`); `BUSCAR` → `PERSECUCION` si lo ve (`te vuelve a ver a distancia D`); `HUIDA` → `PATRULLA` si la vida es 60 o más (`recuperado con vida V`). Cada cambio escribe `t=N: ANTES → DESPUÉS (motivo)`.
- Acción del estado (ya cambiado): `PATRULLA`, un paso hacia el punto de destino (al llegar a uno, el destino pasa a ser el otro); `PERSECUCION`, un paso hacia el jugador; `ATAQUE`, 10 de daño al jugador; `BUSCAR`, un paso hacia la última posición vista y, una vez allí, espera: al cuarto turno de espera escribe `t=N: BUSCAR → PATRULLA (no te encuentra)` y vuelve a patrullar; `HUIDA`, un paso alejándose del jugador si está a 8 o menos, o recupera 10 de vida (hasta 100) si está más lejos.
- Un paso mueve una casilla en el eje en el que queda más distancia (en horizontal si empatan); alejarse es lo mismo en sentido contrario. Al final: `Posición final: (x, y) · vida V · daño causado D` y `Turnos por estado: PATRULLA n, ...` con los estados que han tenido algún turno, en el orden `PATRULLA`, `PERSECUCION`, `ATAQUE`, `BUSCAR`, `HUIDA`. Cada turno cuenta para el estado en el que termina (el turno en que cae derrotado, para el estado que tenía).

Formato de la entrada

- Línea 1: `PATRULLA x1,y1 x2,y2`. Resto: un turno por línea, `x,y [SIGILO] [GOLPE n]`.

Estados y transiciones

Estado	Qué hace	Sale hacia
PATRULLA	va y vuelve entre los dos puntos	PERSECUCION (te ve) · HUIDA (vida $\leq$ 30)
PERSECUCION	un paso hacia ti	ATAQUE (distancia $\leq$ 1) · BUSCAR (distancia $>$ 8) · HUIDA
ATAQUE	10 de daño por turno	PERSECUCION (distancia $>$ 1) · HUIDA
BUSCAR	va a donde te vio y espera 3 turnos	PERSECUCION (te ve) · PATRULLA (no te encuentra) · HUIDA
HUIDA	se aleja o, lejos, recupera 10 de vida	PATRULLA (vida $\geq$ 60)

Ejemplo: Te ve, te persigue, ataca y te pierde

ENTRADA

```
PATRULLA 0,0 4,0
9,5
8,4
6,2
4,1
3,2
3,2
3,2
3,2
5,3
9,6
12,8
14,9
15,9
15,9
15,9
15,9
15,9
15,9
```

SALIDA ESPERADA

```
t=4: PATRULLA → PERSECUCION (te ve a distancia 2)
t=7: PERSECUCION → ATAQUE (a tu lado)
t=9: ATAQUE → PERSECUCION (te alejas a distancia 4)
t=10: PERSECUCION → BUSCAR (te pierde a distancia 10)
t=16: BUSCAR → PATRULLA (no te encuentra)
Posición final: (4, 1) · vida 100 · daño causado 20
Turnos por estado: PATRULLA 7, PERSECUCION 4,
ATAQUE 2, BUSCAR 6
```

Ejemplo: Sigilo, golpes y huida

ENTRADA

```
PATRULLA 0,0 0,3
2,0 SIGILO
1,0 SIGILO GOLPE 40
1,0 GOLPE 35
2,0
3,0
4,0
5,0
12,0
12,0
12,0
12,0
12,0
```

SALIDA ESPERADA

```
t=3: PATRULLA → HUIDA (vida 25)
t=10: HUIDA → PATRULLA (recuperado con vida 65)
Posición final: (0, 2) · vida 65 · daño causado 0
Turnos por estado: PATRULLA 5, HUIDA 7
```

Código de partida (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const VISION = 5, PERDER = 8, ESPERA = 3, VIDA_MAX = 100, HUIR = 30, RECUPERADO = 60;

const distancia = (a, b) => Math.abs(a.x - b.x) + Math.abs(a.y - b.y);

function main() {
  // TODO: lee la ruta de patrulla y, turno a turno, aplica las transiciones y la acción de cada estado
}

main();
```

Soluciones

1. Densidades de pantalla y recursos de Android

Medio · Tecnologías y Plataformas Móviles · apuntesdam.com/subject/prog-multimedia/topic/tecnologias-plataformas-movil

Solución (java)

```
import java.util.*;

public class Main {
    static final String[] NOMBRES = {"ldpi", "mdpi", "hdpi", "xhdpi", "xxhdpi", "xxxhdpi"};
    static final int[] DPI = {120, 160, 240, 320, 480, 640};

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" "); // nombre anchoPx altoPx pulgadas
            if (p.length < 4) continue;
            int ancho = Integer.parseInt(p[1]), alto = Integer.parseInt(p[2]);
            double dpi = Math.hypot(ancho, alto) / Double.parseDouble(p[3]);

            int g = 0; // grupo más cercano (en empate, el menor)
            for (int i = 1; i < DPI.length; i++) {
                if (Math.abs(dpi - DPI[i]) < Math.abs(dpi - DPI[g])) g = i;
            }
            double factor = DPI[g] / 160.0;
            String textoFactor = factor == Math.floor(factor) ? String.valueOf((long) factor) :
                String.valueOf(factor);

            long corto = Math.round(Math.min(ancho, alto) / factor), largo = Math.round(Math.max(ancho,
                alto) / factor);
            String tipo = corto >= 720 ? "tableta grande" : corto >= 600 ? "tableta" : "móvil";
            String carpeta = corto >= 720 ? "layout-sw720dp" : corto >= 600 ? "layout-sw600dp" :
                "layout";

            System.out.println(p[0] + ": " + Math.round(dpi) + " dpi → " + NOMBRES[g] + " (x" +
                textoFactor + ")", "
                + corto + " x " + largo + " dp, " + tipo + " → " + carpeta + ", icono 48 dp = " +
                Math.round(48 * factor) + " px");
        }
    }
}
```

La densidad real es la diagonal en píxeles (Pitágoras, con `Math.hypot`) dividida entre la diagonal en pulgadas. Android no usa ese valor exacto: redondea al grupo más cercano, y el factor de ese grupo respecto a mdpi (160) es el que convierte dp en píxeles. Por eso un icono de 48 dp mide $48 \times \text{factor}$ píxeles.

Al dividir los píxeles entre el factor se obtiene el tamaño en dp, que es parecido en todos los móviles (unos 360 dp de ancho) aunque sus píxeles sean muy distintos. La anchura mínima en dp decide la carpeta de diseños: así una tableta puede usar una interfaz de dos columnas.

2. El ciclo de vida de una Activity

Difícil · Tecnologías y Plataformas Móviles · apuntesdam.com/subject/prog-multimedia/topic/tecnologias-plataformas-movil

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // Estados: NINGUNO, CREADA, INICIADA, REANUDADA, PAUSADA, DETENIDA, DESTRUIDA
        String estado = "NINGUNO", texto = "", guardado = null; // guardado = el Bundle de
        onSaveInstanceState
        while (sc.hasNextLine()) {
            String linea = sc.nextLine().trim();
            if (linea.isEmpty()) continue;
            String[] p = linea.split(" ", 2);
            List<String> llamadas = new ArrayList<>();
            switch (p[0]) {
                case "abrir" -> {
                    llamadas.addAll(List.of("onCreate", "onStart", "onResume"));
                    estado = "REANUDADA";
                    texto = "";
                }
                case "escribir" -> {
                    texto = p.length > 1 ? p[1] : "";
                    llamadas.add("el usuario escribe «" + texto + "»");
                }
                case "dialogo" -> {
                    if (estado.equals("REANUDADA")) { llamadas.add("onPause"); estado = "PAUSADA"; }
                }
                case "cerrar_dialogo" -> {
                    if (estado.equals("PAUSADA")) { llamadas.add("onResume"); estado = "REANUDADA"; }
                }
                case "home" -> {
                    if (estado.equals("REANUDADA")) llamadas.add("onPause");
                    llamadas.addAll(List.of("onStop", "onSaveInstanceState"));
                    guardado = texto;
                    estado = "DETENIDA";
                }
                case "volver" -> {
                    if (estado.equals("DETENIDA")) { llamadas.addAll(List.of("onRestart", "onStart",
                        "onResume")); estado = "REANUDADA"; }
                }
                case "girar" -> {
                    llamadas.addAll(List.of("onPause", "onStop", "onSaveInstanceState", "onDestroy",
                        "onCreate", "onStart", "onRestoreInstanceState", "onResume"));
                    guardado = texto; // se guarda antes de destruirla...
                    texto = guardado; // ...y se recupera en la nueva instancia
                    estado = "REANUDADA";
                }
                case "matar_proceso" -> {
                    if (estado.equals("DETENIDA")) {
                        llamadas.add("proceso eliminado sin más llamadas");
                        estado = "NINGUNO";
                        texto = ""; // lo que había en memoria se pierde; el Bundle no
                    }
                }
                case "reabrir" -> {
                    llamadas.addAll(List.of("onCreate", "onStart"));
                    if (guardado != null) llamadas.add("onRestoreInstanceState");
                    llamadas.add("onResume");
                    texto = guardado != null ? guardado : "";
                    estado = "REANUDADA";
                }
                case "atras" -> {
                    if (estado.equals("REANUDADA")) llamadas.add("onPause");
                    llamadas.addAll(List.of("onStop", "onDestroy"));
                    estado = "DESTRUIDA";
                    texto = "";
                    guardado = null; // el usuario la cerró: no se guarda nada
                }
                default -> llamadas.add("(evento desconocido)");
            }
        }
    }
}
```

```

    }
    if (llamadas.isEmpty()) llamadas.add("(sin llamadas)");
    System.out.println(p[0] + ": " + String.join(" → ", llamadas) + " | estado " + estado + " |
        texto «" + texto + "»");
    }
}
}

```

Se guardan tres cosas: el estado de la Activity, el texto que hay en memoria y el que se guardó en

`onSaveInstanceState` (el «Bundle»). La diferencia entre los dos textos es la lección del ejercicio: lo que está solo en memoria se pierde si el sistema mata el proceso, y lo guardado en el Bundle se recupera al recrear la pantalla.

Cada evento añade a una lista las llamadas que haría Android, algunas solo si el estado lo permite (por ejemplo, `onPause` solo si la pantalla estaba reanudada). Un giro destruye y recrea la Activity: sin `onSaveInstanceState` el usuario perdería lo que estaba escribiendo. En cambio, al pulsar «atrás» el usuario cierra la pantalla a propósito y no se guarda nada.

3. Elegir el minSdk de una app

Muy difícil · Tecnologías y Plataformas Móviles · apuntesdam.com/subject/prog-multimedia/topic/tecnologias-

Solución (java)

```

import java.util.*;

public class Main {
    static String numero(double v) {
        return v == Math.floor(v) ? String.valueOf((long) v) : String.valueOf(v);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        TreeMap<Integer, Double> cuota = new TreeMap<>(); // nivel de API → % de dispositivos
        List<String[]> apis = new ArrayList<>(); // nombre nivelMinimo
        int minSdk = 0;
        double objetivo = 0;
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            switch (p[0]) {
                case "cuota" -> cuota.put(Integer.parseInt(p[1]), Double.parseDouble(p[2]));
                case "api" -> apis.add(new String[]{p[1], p[2]});
                case "minSdk" -> minSdk = Integer.parseInt(p[1]);
                case "objetivo" -> objetivo = Double.parseDouble(p[1]);
                default -> { }
            }
        }

        double total = 0;
        for (double c : cuota.values()) total += c;
        final double suma = total;
        // Porcentaje (normalizado) de dispositivos con nivel en [desde, hasta)
        java.util.function.BiFunction<Integer, Integer, Double> entre = (desde, hasta) -> {
            double s = 0;
            for (double c : cuota.subMap(desde, true, hasta, false).values()) s += c;
            return s / suma * 100;
        };

        System.out.printf(Locale.US, "minSdk %d: %.1f %% de los dispositivos%n", minSdk,
            entre.apply(minSdk, Integer.MAX_VALUE));

        Integer recomendado = null;
        for (int nivel : cuota.descendingKeySet()) { // del más alto al más bajo
            if (entre.apply(nivel, Integer.MAX_VALUE) >= objetivo) { recomendado = nivel; break; }
        }
        if (recomendado == null) System.out.println("Ningún minSdk alcanza el " + numero(objetivo) + "
            %");
        else System.out.printf(Locale.US, "Recomendado para el %s %: minSdk %d (%.1f %)%n",
            numero(objetivo), recomendado, entre.apply(recomendado, Integer.MAX_VALUE));

        for (String[] api : apis) {
            int nivel = Integer.parseInt(api[1]);
            if (nivel <= minSdk) System.out.println(api[0] + ": disponible siempre (API " + nivel + ")");
            else System.out.printf(Locale.US, "%s: comprobar Build.VERSION.SDK_INT >= %d (no disponible
                en el %.1f %% de los dispositivos)%n",
                api[0], nivel, entre.apply(minSdk, nivel));
        }
    }
}

```

Todo se reduce a una operación: qué porcentaje de dispositivos tiene un nivel de API dentro de un intervalo. Con un `TreeMap`, `subMap(desde, true, hasta, false)` devuelve justo ese tramo de niveles, y dividir entre la suma total normaliza las cuotas aunque no sumen 100.

La cobertura de un `minSdk` es el tramo `[minSdk, ∞)`; el recomendado se busca recorriendo los niveles de mayor a menor (`descendingKeySet`) y quedándose con el primero que aún llega al objetivo. Para cada API más nueva que el `minSdk`, los dispositivos que no la tienen son los del tramo `[minSdk, nivel de la API)`: para ellos hace falta la comprobación de versión.

4. Las diferencias de una lista, como DiffUtil

Medio · Desarrollo de Aplicaciones Móviles · apuntesdam.com/subject/prog-multimedia/topic/desarrollo-aplicaciones-movil

Solución (java)

```
import java.util.*;

public class Main {
    record Elemento(String id, String contenido) { }

    static List<Elemento> leer(String linea) {
        List<Elemento> l = new ArrayList<>();
        if (linea.isBlank()) return l;
        for (String par : linea.trim().split(" ")) {
            String[] p = par.split(":");
            l.add(new Elemento(p[0], p[1]));
        }
        return l;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<Elemento> antes = leer(sc.nextLine()), despues = leer(sc.hasNextLine() ? sc.nextLine() :
            "");

        Map<String, String> viejos = new LinkedHashMap<>(), nuevos = new LinkedHashMap<>();
        for (Elemento e : antes) viejos.put(e.id(), e.contenido());
        for (Elemento e : despues) nuevos.put(e.id(), e.contenido());

        int borrados = 0, insertados = 0, modificados = 0, iguales = 0;
        for (Elemento e : antes) {
            if (!nuevos.containsKey(e.id())) {
                System.out.println("notifyItemRemoved: " + e.id());
                borrados++;
            }
        }
        for (int i = 0; i < despues.size(); i++) {
            Elemento e = despues.get(i);
            String viejo = viejos.get(e.id());
            if (viejo == null) {
                System.out.println("notifyItemInserted(" + i + "): " + e.id());
                insertados++;
            } else if (!viejo.equals(e.contenido())) {
                System.out.println("notifyItemChanged(" + i + "): " + e.id() + " " + viejo + " → " +
                    e.contenido());
                modificados++;
            } else {
                iguales++;
            }
        }

        // ¿Los que siguen en las dos listas mantienen su orden relativo?
        List<String> comunesAntes = new ArrayList<>(), comunesDespues = new ArrayList<>();
        for (Elemento e : antes) if (nuevos.containsKey(e.id())) comunesAntes.add(e.id());
        for (Elemento e : despues) if (viejos.containsKey(e.id())) comunesDespues.add(e.id());
        boolean movidos = !comunesAntes.equals(comunesDespues);

        System.out.println("Sin cambios: " + iguales + " · Insertados: " + insertados + " · Borrados: " +
            borrados
            + " · Modificados: " + modificados + (movidos ? " · Hay elementos movidos" : ""));
        if (insertados + modificados + borrados == 0 && !movidos) System.out.println("No hace falta
            redibujar");
        else System.out.println("Se redibujan " + (insertados + modificados) + " de " + despues.size()
            + " filas (notifyDataSetChanged redibujaría " + despues.size() + ")");
    }
}
```

La identidad de cada elemento es su id, no su posición: dos mapas id → contenido permiten saber en una consulta si un elemento es nuevo (no estaba antes), ha desaparecido (no está después) o ha cambiado (está en los dos con distinto contenido). Es la misma distinción que hace DiffUtil con `areItemsTheSame` y `areContentsTheSame`.

Para detectar movimientos se comparan las listas de ids que están en las dos versiones, cada una en su orden: si no coinciden, algún elemento ha cambiado de sitio. Solo las filas insertadas o modificadas necesitan dibujarse otra vez, que es lo que se ahorra frente a `notifyDataSetChanged`.

5. La pila de navegación de una app

Difícil · Desarrollo de Aplicaciones Móviles · apuntesdam.com/subject/prog-multimedia/topic/desarrollo-aplicaciones-movil

Solución (java)

```
import java.util.*;

public class Main {
    static String mostrar(Deque<String> pila) {
        List<String> l = new ArrayList<>();
        pila.descendingIterator().forEachRemaining(l::add);    // de la base a la cima
        return "[" + String.join(" > ", l) + "]";
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Deque<String> pila = new ArrayDeque<>();    // la cima es la pantalla visible
        pila.push(sc.nextLine().trim().split(" ")[1]);    // inicio DESTINO
        while (sc.hasNextLine()) {
            String orden = sc.nextLine().trim();
            if (orden.isEmpty()) continue;
            String[] p = orden.split(" ");
            if (p[0].equals("back")) {
                if (pila.size() == 1) {
                    System.out.println("back: se cierra la app desde " + pila.peek());
                    return;
                }
                pila.pop();
                System.out.println("back → " + mostrar(pila));
                continue;
            }
            // navigate DESTINO [popUpTo=X] [inclusive] [singleTop]
            String destino = p[1], hasta = null;
            boolean inclusive = false, singleTop = false;
            for (int i = 2; i < p.length; i++) {
                if (p[i].startsWith("popUpTo=")) hasta = p[i].substring(8);
                else if (p[i].equals("inclusive")) inclusive = true;
                else if (p[i].equals("singleTop")) singleTop = true;
            }
            if (hasta != null) {
                if (!pila.contains(hasta)) {
                    System.out.println("ERROR: " + hasta + " no está en la pila");
                    continue;
                }
                while (!pila.peek().equals(hasta)) pila.pop();
                if (inclusive) pila.pop();
            }
            if (singleTop && destino.equals(pila.peek())) {
                System.out.println(orden + ": ya estás en " + destino + " (singleTop)");
                continue;
            }
            pila.push(destino);
            System.out.println(orden + " → " + mostrar(pila));
        }
    }
}
```

La pila de navegación es un `Deque` usado como pila: `push` apila la nueva pantalla, `pop` la quita y `peek` dice cuál se ve. Para mostrarla de la base a la cima se recorre al revés con `descendingIterator`.

`popUpTo` desapila hasta que `X` queda arriba (y, con `inclusive`, también `X`) antes de apilar el destino. Así, tras iniciar sesión, la pantalla de acceso desaparece de la pila y «atrás» no vuelve a ella. `singleTop` evita apilar dos veces la misma pantalla si ya es la visible.

6. Sincronización sin conexión con conflictos

Muy difícil · Desarrollo de Aplicaciones Móviles · apuntesdam.com/subject/prog-multimedia/topic/desarrollo-

Solución (java)

```

import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, String[]> servidor = new TreeMap<>(), local = new TreeMap<>(); // id → {valor,
        marca de tiempo}
        List<String[]> pendientes = new ArrayList<>(); // cambios hechos sin
        conexión
        String seccion = "";
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.isEmpty()) continue;
            if (l.startsWith("(")) { seccion = l; continue; }
            String[] p = l.split(" ");
            switch (seccion) {
                case "[servidor]" -> servidor.put(p[0], new String[]{p[1], p[2]});
                case "[base]" -> local.put(p[0], new String[]{p[1], p[2]});
                case "[cambios]" -> pendientes.add(p); // op id valor tiempo (op: crear,
                editar, borrar)
                default -> { }
            }
        }

        int aplicados = 0, conflictos = 0;
        for (String[] c : pendientes) {
            String op = c[0], id = c[1];
            String valor = op.equals("borrar") ? null : c[2];
            long tiempo = Long.parseLong(c[c.length - 1]);
            String[] actual = servidor.get(id);

            if (op.equals("crear")) {
                if (actual != null) {
                    System.out.println("CONFLICTO " + id + ": ya existe en el servidor (se conserva «" +
                        actual[0] + "»)");
                    conflictos++;
                } else {
                    servidor.put(id, new String[]{valor, String.valueOf(tiempo)});
                    System.out.println("CREADO " + id + " = «" + valor + "»");
                    aplicados++;
                }
                continue;
            }
            if (actual == null) {
                System.out.println("IGNORADO " + id + ": ya no existe en el servidor");
                continue;
            }
            String[] base = local.get(id);
            boolean cambioEnServidor = base == null || !base[1].equals(actual[1]);
            long tiempoServidor = Long.parseLong(actual[1]);
            if (cambioEnServidor && tiempoServidor > tiempo) { // gana el último
                System.out.println("CONFLICTO " + id + ": gana el servidor «" + actual[0] + "» (" +
                    tiempoServidor
                    + ") frente al cambio local (" + tiempo + ")");
                conflictos++;
                continue;
            }
            if (op.equals("borrar")) {
                servidor.remove(id);
                System.out.println("BORRADO " + id + (cambioEnServidor ? " (sobre un cambio del servidor
                    más antiguo" : ""));
            } else {
                servidor.put(id, new String[]{valor, String.valueOf(tiempo)});
                System.out.println("EDITADO " + id + " = «" + valor + "»" + (cambioEnServidor ? " (gana
                    el cambio local, más reciente" : ""));
            }
            aplicados++;
        }
    }
}

```

```

        System.out.println("Servidor tras sincronizar:");
        servidor.forEach((id, v) -> System.out.println(" " + id + " = «" + v[0] + "» (" + v[1] + ")"));
        System.out.println("Aplicados: " + aplicados + " · Conflictos: " + conflictos);
    }
}

```

La clave está en la versión base: comparando la marca de tiempo del servidor con la que tenía el móvil al quedarse sin conexión se sabe si alguien más ha cambiado ese elemento mientras tanto. Si no ha cambiado, el cambio local se aplica sin más; si ha cambiado, hay que resolver el conflicto.

La estrategia «gana el último» compara las marcas de tiempo: si el cambio del servidor es más reciente, se conserva y se descarta el local. Los casos que no tienen sentido (crear algo que ya existe, editar algo que ya se borró) se detectan antes, mirando si el elemento está en el servidor.

7. Física con paso fijo, acumulador e interpolación

Medio · Motores de Juegos · apuntesdam.com/subject/prog-multimedia/topic/motores-juegos

Solución (java)

```

import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int paso = Integer.parseInt(sc.nextLine().trim().split(" ")[1]); // «paso 20» (ms)
        double velocidad = Double.parseDouble(sc.nextLine().trim().split(" ")[1]); // «velocidad 120» (px/s)
        String[] deltas = sc.nextLine().trim().split(" "); // ms de cada fotograma
        final int MAXIMO = 250; // evita la «espiral de la muerte»
        double x = 0, xAnterior = 0;
        int acumulado = 0, totalPasos = 0;

        for (int f = 0; f < deltas.length; f++) {
            int dt = Integer.parseInt(deltas[f]);
            acumulado += Math.min(dt, MAXIMO);
            int pasos = 0;
            while (acumulado >= paso) { // tantos pasos fijos como quepan
                xAnterior = x;
                x += velocidad * paso / 1000.0;
                acumulado -= paso;
                pasos++;
            }
            totalPasos += pasos;
            double alfa = (double) acumulado / paso; // cuánto llevamos del siguiente paso
            double dibujo = xAnterior + (x - xAnterior) * alfa;
            System.out.printf(Locale.US, "F%d dt=%d ms: %d %s, sobran %d ms, x=%.1f, se dibuja en %.1f (alfa %.2f)%n",
                f + 1, dt, dt > MAXIMO ? " (limitado a " + MAXIMO + ")" : "", pasos, pasos == 1 ? "paso" : "pasos",
                acumulado, x, dibujo, alfa);
        }
        System.out.println("Fotogramas: " + deltas.length + " · Pasos de física: " + totalPasos + " · Tiempo simulado: " + totalPasos * paso + " ms");
    }
}

```

El acumulador guarda el tiempo real que la física todavía no ha simulado. Cada fotograma le suma su duración y se ejecutan pasos fijos mientras quepa uno entero; lo que sobra se queda para el siguiente fotograma. Así la física avanza siempre igual, vaya el juego a 30 o a 144 FPS.

Como la posición de la física solo cambia en saltos de un paso, se dibuja interpolando entre la posición anterior y la actual con `alfa = acumulado / paso`: el movimiento se ve suave aunque la física vaya a otro ritmo. El tope de 250 ms evita que, tras un parón, el juego tenga que simular tantos pasos que vuelva a atascarse.

8. Colisiones con rejilla espacial

Difícil · Motores de Juegos · apuntesdam.com/subject/prog-multimedia/topic/motores-juegos

Solución (java)

```
import java.util.*;

public class Main {
    record Caja(String nombre, int x, int y, int ancho, int alto) { }

    static boolean chocan(Caja a, Caja b) {
        return a.x() < b.x() + b.ancho() && a.x() + a.ancho() > b.x()
            && a.y() < b.y() + b.alto() && a.y() + a.alto() > b.y();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int celda = Integer.parseInt(sc.nextLine().trim().split(" ")[1]); // «celda 64»
        List<Caja> cajas = new ArrayList<>();
        while (sc.hasNextLine()) {
            String[] p = sc.nextLine().trim().split(" ");
            if (p.length < 5) continue;
            cajas.add(new Caja(p[0], Integer.parseInt(p[1]), Integer.parseInt(p[2]),
                Integer.parseInt(p[3]), Integer.parseInt(p[4])));
        }
        int n = cajas.size();

        // Fuerza bruta: todos los pares
        Set<String> brutos = new LinkedHashSet<>();
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                if (chocan(cajas.get(i), cajas.get(j))) brutos.add(i + "-" + j);

        // Rejilla: cada caja se apunta en todas las celdas que toca
        Map<String, List<Integer>> rejilla = new LinkedHashMap<>();
        for (int i = 0; i < n; i++) {
            Caja c = cajas.get(i);
            for (int cx = Math.floorDiv(c.x(), celda); cx <= Math.floorDiv(c.x() + c.ancho() - 1, celda);
                cx++)
                for (int cy = Math.floorDiv(c.y(), celda); cy <= Math.floorDiv(c.y() + c.alto() - 1,
                    celda); cy++)
                    rejilla.computeIfAbsent(cx + "," + cy, k -> new ArrayList<>()).add(i);
        }
        Set<List<Integer>> candidatos = new HashSet<>(); // pares distintos que comparten alguna
            celda
        String masLlena = null;
        for (Map.Entry<String, List<Integer>> e : rejilla.entrySet()) {
            List<Integer> l = e.getValue();
            if (masLlena == null || l.size() > rejilla.get(masLlena).size()) masLlena = e.getKey();
            for (int a = 0; a < l.size(); a++)
                for (int b = a + 1; b < l.size(); b++) candidatos.add(List.of(l.get(a), l.get(b)));
        }
        List<int[]> choques = new ArrayList<>();
        for (List<Integer> par : candidatos)
            if (chocan(cajas.get(par.get(0)), cajas.get(par.get(1)))) choques.add(new int[]{par.get(0),
                par.get(1)});
        choques.sort(Comparator.<int[]>comparingInt(c -> c[0]).thenComparingInt(c -> c[1]));

        List<String> textos = new ArrayList<>(), claves = new ArrayList<>();
        for (int[] c : choques) {
            textos.add(cajas.get(c[0]).nombre() + "-" + cajas.get(c[1]).nombre());
            claves.add(c[0] + "-" + c[1]);
        }
        long total = (long) n * (n - 1) / 2;
        System.out.println("Celdas ocupadas: " + rejilla.size() + " · La más llena: ("
            + (masLlena == null ? "-" : masLlena) + ") con " + (masLlena == null ? 0 :
                rejilla.get(masLlena).size()));
        System.out.println("Choques: " + (textos.isEmpty() ? "ninguno" : String.join(" ", textos)));
        System.out.println("Comprobaciones: fuerza bruta " + total + " · rejilla " + candidatos.size()
            + " (ahorro " + (total == 0 ? 0 : Math.round(100.0 * (total - candidatos.size()) /
                total)) + " %)");
    }
}
```

```
        System.out.println("Mismo resultado: " + (new HashSet<>(claves).equals(brutos) ? "sí" : "no"));
    }
}
```

La fase amplia reparte las cajas en celdas: cada caja se apunta en todas las celdas que toca, calculadas con `Math.floorDiv` (que, a diferencia de `/`, redondea hacia abajo también con coordenadas negativas). Después solo se comprueban los pares que comparten celda; dos cajas en celdas distintas no pueden tocarse.

Un par puede compartir varias celdas (dos cajas grandes), así que los candidatos se guardan en un `Set` para no comprobarlos dos veces; su tamaño es el número de comprobaciones. La fuerza bruta se calcula también para demostrar que la rejilla encuentra exactamente los mismos choques con muchas menos comprobaciones.

9. El efecto túnel: detección discreta frente a continua

Muy difícil · Motores de Juegos · apuntesdam.com/subject/prog-multimedia/topic/motores-juegos

Solución (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String[] m = sc.nextLine().trim().split(" "); // «pared x y ancho alto»
        double px = Double.parseDouble(m[1]), py = Double.parseDouble(m[2]);
        double pw = Double.parseDouble(m[3]), ph = Double.parseDouble(m[4]);
        int fps = Integer.parseInt(sc.nextLine().trim().split(" ")[1]); // «fps 60»
        int tuneles = 0, balas = 0;
        while (sc.hasNextLine()) {
            String[] b = sc.nextLine().trim().split(" ");
            if (b.length < 5) continue;
            String nombre = b[0];
            double x0 = Double.parseDouble(b[1]), y0 = Double.parseDouble(b[2]);
            double vx = Double.parseDouble(b[3]), vy = Double.parseDouble(b[4]);
            balas++;

            // Discreta: solo se mira la posición al final de cada fotograma
            String discreta = "no detecta nada";
            for (int k = 1; k <= fps; k++) {
                double x = x0 + vx * k / fps, y = y0 + vy * k / fps;
                if (x > px && x < px + pw && y > py && y < py + ph) {
                    discreta = String.format(Locale.US, "choca en el fotograma %d (t=%.3f s)", k,
                        (double) k / fps);
                    break;
                }
            }

            // Continua: método de las franjas sobre el segmento recorrido en [0, 1]
            String continua;
            boolean impacto = false;
            if (x0 > px && x0 < px + pw && y0 > py && y0 < py + ph) {
                continua = "empieza dentro de la pared";
            } else {
                double[] fx = franja(x0, vx, px, px + pw), fy = franja(y0, vy, py, py + ph);
                double entrada = fx == null || fy == null ? 0 : Math.max(fx[0], fy[0]);
                double salida = fx == null || fy == null ? 0 : Math.min(fx[1], fy[1]);
                if (fx == null || fy == null || entrada > salida || entrada < 0 || entrada > 1) {
                    continua = "no hay impacto";
                } else {
                    impacto = true;
                    String cara = fx[0] >= fy[0] ? (vx > 0 ? "izquierda" : "derecha") : (vy > 0 ?
                        "superior" : "inferior");
                    continua = String.format(Locale.US, "impacto en t=%.3f s en (%.1f, %.1f) por la cara
                        %s",
                        entrada, x0 + vx * entrada, y0 + vy * entrada, cara);
                }
            }
            boolean tunel = impacto && discreta.equals("no detecta nada");
            if (tunel) tuneles++;
            System.out.println(nombre + ": discreta → " + discreta + " | continua → " + continua + (tunel
                ? " ¡ATRAVIESA LA PARED!" : ""));
        }
        System.out.println("Efecto túnel: " + tuneles + " de " + balas + " balas");
    }

    // Intervalo de tiempo [entrada, salida] en que la coordenada está dentro de (min, max); null si
    // nunca
    static double[] franja(double p0, double v, double min, double max) {
        if (v == 0) {
            return p0 > min && p0 < max ? new double[]{Double.NEGATIVE_INFINITY,
                Double.POSITIVE_INFINITY} : null;
        }
        double t1 = (min - p0) / v, t2 = (max - p0) / v;
        return new double[]{Math.min(t1, t2), Math.max(t1, t2)};
    }
}
```

```
}  
}
```

La detección discreta solo mira dónde está la bala al final de cada fotograma; si la pared es más fina que lo que avanza la bala en un fotograma, puede saltársela entera. La continua mira el segmento completo que recorre la bala.

El método de las franjas calcula, en cada eje, el intervalo de tiempo en que la bala está entre los dos bordes de la pared: $(\text{borde} - p_0) / v$. La bala está dentro cuando está a la vez en las dos franjas, así que el impacto es el último instante de entrada (`max` de las entradas) siempre que sea anterior al primer instante de salida y esté dentro del segundo simulado. El eje que entra más tarde es la cara por la que choca.

Si la velocidad en un eje es 0, la franja de ese eje es «siempre» o «nunca», según la bala ya esté o no entre sus bordes; el método `franja` devuelve un intervalo infinito o `null`.

10. ¿Quién gana al tres en raya?

Fácil · Ejercicios de Programación Multimedia y Dispositivos Móviles · apuntesdam.com/subject/programacion-multimedia/topic/ejercicios-pmdm

Solución (java)

```
import java.util.*;  
  
public class Main {  
  
    static boolean linea(char[][] t, char j, int f0, int c0, int df, int dc) {  
        for (int k = 0; k < 3; k++) if (t[f0 + k * df][c0 + k * dc] != j) return false;  
        return true;  
    }  
  
    static boolean gana(char[][] t, char j) {  
        for (int i = 0; i < 3; i++) {  
            if (linea(t, j, i, 0, 0, 1) || linea(t, j, 0, i, 1, 0)) return true;    // fila i o columna i  
        }  
        return linea(t, j, 0, 0, 1, 1) || linea(t, j, 0, 2, 1, -1);                // diagonales  
    }  
  
    static String resultado(char[][] t) {  
        if (gana(t, 'X')) return "Gana X";  
        if (gana(t, 'O')) return "Gana O";  
        for (char[] fila : t) for (char c : fila) if (c == '.') return "Sigue la partida";  
        return "Empate";  
    }  
  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        char[][] t = new char[3][3];  
        for (int i = 0; i < 3; i++) t[i] = sc.next().toCharArray();  
        System.out.println(resultado(t));  
    }  
}
```

Las ocho líneas ganadoras (tres filas, tres columnas y dos diagonales) son todas «tres casillas a partir de una, avanzando en una dirección». El método `linea` lo comprueba con un punto de partida y un desplazamiento `(df, dc)`, y así no hay que escribir ocho condiciones distintas.

Si nadie gana, la partida sigue mientras quede alguna casilla vacía; si no queda ninguna, es empate.

11. Una generación del juego de la vida

Medio · Ejercicios de Programación Multimedia y Dispositivos Móviles · apuntesdam.com/subject/programacion-multimedia/topic/ejercicios-pmdm

Solución (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int f = sc.nextInt(), c = sc.nextInt();
        char[][] t = new char[f][];
        for (int i = 0; i < f; i++) t[i] = sc.next().toCharArray();

        char[][] nueva = new char[f][c];
        for (int i = 0; i < f; i++) {
            for (int j = 0; j < c; j++) {
                int vecinas = 0;
                for (int di = -1; di <= 1; di++)
                    for (int dj = -1; dj <= 1; dj++) {
                        int a = i + di, b = j + dj;
                        if ((di != 0 || dj != 0) && a >= 0 && a < f && b >= 0 && b < c && t[a][b] == '#')
                            vecinas++;
                    }
                boolean viva = t[i][j] == '#';
                nueva[i][j] = (viva && (vecinas == 2 || vecinas == 3)) || (!viva && vecinas == 3) ? '#' :
                    '.';
            }
        }
        for (char[] fila : nueva) System.out.println(new String(fila));
    }
}
```

La siguiente generación se escribe en una matriz nueva: todas las células cambian a la vez, y si se modificara el tablero mientras se recorre, las células ya cambiadas alterarían el recuento de sus vecinas.

Las ocho vecinas se cuentan con dos bucles de -1 a 1 , saltando la propia célula y las posiciones fuera del tablero (los bordes no se conectan). Las reglas se escriben en una sola expresión: viva con 2 o 3 vecinas, o muerta con exactamente 3.

12. Puntuación con combos

Medio · Ejercicios de Programación Multimedia y Dispositivos Móviles · apuntesdam.com/subject/prog-

Solución (java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int puntos = 0, vidas = 3, combo = 0, mejor = 0;
        while (sc.hasNext()) {
            String evento = sc.next();
            if (vidas == 0) break; // partida terminada: se ignora el resto
            switch (evento) {
                case "MONEDA", "ENEMIGO" -> {
                    combo++;
                    mejor = Math.max(mejor, combo);
                    int multiplicador = combo >= 10 ? 3 : combo >= 5 ? 2 : 1;
                    puntos += (evento.equals("MONEDA") ? 10 : 50) * multiplicador;
                }
                case "GOLPE" -> {
                    vidas--;
                    combo = 0;
                }
                default -> { }
            }
        }
        System.out.println("Puntos: " + puntos);
        System.out.println("Vidas: " + vidas);
        System.out.println("Mejor combo: " + mejor);
    }
}
```

Cada evento que puntúa incrementa primero el combo y después calcula el multiplicador con él, como pide el enunciado: por eso la quinta moneda seguida ya vale el doble. El mejor combo se actualiza con `Math.max` en el mismo momento.

La comprobación de las vidas va al principio del bucle: en cuanto llegan a 0, `break` termina la partida y los eventos posteriores no cuentan.

13. Camino más corto en un laberinto

Difícil · Ejercicios de Programación Multimedia y Dispositivos Móviles · apuntesdam.com/subject/prog-

Solución (java)

```

import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int f = sc.nextInt(), c = sc.nextInt();
        char[][] m = new char[f][];
        for (int i = 0; i < f; i++) m[i] = sc.next().toCharArray();

        int[][] dist = new int[f][c];
        for (int[] fila : dist) Arrays.fill(fila, -1); // -1 = sin visitar
        Deque<int[]> cola = new ArrayDeque<>();
        for (int i = 0; i < f; i++)
            for (int j = 0; j < c; j++)
                if (m[i][j] == 'E') { dist[i][j] = 0; cola.add(new int[]{i, j}); }

        int[][] dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
        while (!cola.isEmpty()) {
            int[] p = cola.poll();
            if (m[p[0]][p[1]] == 'J') {
                System.out.println(dist[p[0]][p[1]]);
                return;
            }
            for (int[] d : dirs) {
                int a = p[0] + d[0], b = p[1] + d[1];
                if (a >= 0 && a < f && b >= 0 && b < c && m[a][b] != '#' && dist[a][b] == -1) {
                    dist[a][b] = dist[p[0]][p[1]] + 1;
                    cola.add(new int[]{a, b});
                }
            }
        }
        System.out.println("Sin camino");
    }
}

```

La búsqueda en anchura (BFS) explora el mapa por capas de distancia: primero las casillas a 1 paso de E, luego las que están a 2, etc. Por eso, la primera vez que saca a J de la cola, la distancia guardada es la mínima.

La matriz `dist` sirve a la vez para marcar las casillas visitadas (-1 si no lo están) y para guardar a cuántos pasos están. Si la cola se vacía sin encontrar a J, no hay camino.

14. Física de un juego de plataformas: gravedad, saltos y colisiones

Muy difícil · JavaScript · 100 minutos · apuntesdam.com/ejercicios/javascript/fisica-de-un-juego-de-plataformas

Solución (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const T = 16; // tamaño de casilla en píxeles
const ANCHO = 12, ALTO = 14; // caja del jugador
const VEL = 2, SALTO = -9, GRAVEDAD = 1, CAIDA_MAX = 8;

function main() {
  const corte = lineas.findIndex((l) => l.trim() === "---");
  const mapa = (corte < 0 ? lineas : lineas.slice(0, corte)).map((l) => l.trimEnd()).filter((l) => l.length);
  if (corte < 0 || !mapa.length) return console.log("Mapa no válido: falta el separador ---");
  if (mapa.some((l) => l.length !== mapa[0].length)) return console.log("Mapa no válido: las filas no miden lo mismo");
  if (mapa.some((l) => /^[^#CPE]/.test(l))) return console.log("Mapa no válido: carácter desconocido");
  if (mapa.join("").split("P").length !== 2) return console.log("Mapa no válido: tiene que haber un jugador (P)");

  const filas = mapa.length, columnas = mapa[0].length;
  const solida = (f, c) => f >= 0 && f < filas && c >= 0 && c < columnas && mapa[f][c] === "#";
  // Las casillas que toca una caja: de la primera a la última fila y columna que ocupa
  const casillas = (x, y) => {
    const r = [];
    for (let f = Math.floor(y / T); f <= Math.floor((y + ALTO - 1) / T); f++)
      for (let c = Math.floor(x / T); c <= Math.floor((x + ANCHO - 1) / T); c++) r.push([f, c]);
    return r;
  };

  const inicio = mapa.findIndex((l) => l.includes("P"));
  let x = mapa[inicio].indexOf("P") * T + 2, y = inicio * T + (T - ALTO);
  let vy = 0, enSuelo = false;
  const monedas = new Set();
  const total = mapa.join("").split("C").length - 1;

  // Acciones: R, L, J, RJ, LJ o «.», y «R*5» repite cinco fotogramas
  const acciones = [];
  for (const token of lineas.slice(corte + 1).join(" ").split(/\s+/).filter(Boolean)) {
    const [accion, veces] = token.split("*");
    for (let i = 0; i < (veces ? Number(veces) : 1); i++) acciones.push(accion.toUpperCase());
  }

  for (let n = 1; n <= acciones.length; n++) {
    const a = acciones[n - 1];
    // 1. Horizontal: mover y, si choca, pegarse a la pared
    const vx = (a.includes("R") ? VEL : 0) - (a.includes("L") ? VEL : 0);
    x = Math.max(0, Math.min(columnas * T - ANCHO, x + vx));
    for (const [f, c] of casillas(x, y)) {
      if (!solida(f, c)) continue;
      x = vx > 0 ? c * T - ANCHO : (c + 1) * T;
      break; // corregida la posición, ya no toca ninguna otra casilla de esa columna
    }
    // 2. Vertical: salto solo desde el suelo, gravedad con velocidad máxima de caída
    if (a.includes("J") && enSuelo) vy = SALTO;
    vy = Math.min(vy + GRAVEDAD, CAIDA_MAX);
    y += vy;
    enSuelo = false;
    // La dirección se decide antes de corregir: después vy vale 0 y ya no se sabría si bajaba
    const bajando = vy > 0;
    for (const [f, c] of casillas(x, y)) {
      if (!solida(f, c)) continue;
      if (bajando) { y = f * T - ALTO; enSuelo = true; } else { y = (f + 1) * T; }
      vy = 0;
      break;
    }
    // 3. Monedas, meta y caída
    for (const [f, c] of casillas(x, y)) {
```

```

    if (f < 0 || f >= filas || c < 0 || c >= columnas) continue;
    if (mapa[f][c] === "C" && !monedas.has(`${f},${c}`)) {
        monedas.add(`${f},${c}`);
        console.log(`Fotograma ${n}: moneda (${monedas.size}/${total})`);
    }
    if (mapa[f][c] === "E") return console.log(`Fotograma ${n}: ¡meta! Nivel superado con
        ${monedas.size}/${total} monedas`);
}
if (y >= filas * T) return console.log(`Fotograma ${n}: el jugador ha caído al vacío con
    ${monedas.size}/${total} monedas`);
}
console.log(`Fin de las acciones (${acciones.length} fotogramas): posición (${x}, ${y}), ${enSuelo ?
    "en el suelo" : "en el aire"}, ${monedas.size}/${total} monedas`);
}

main();

```

El programa es un bucle de juego en miniatura: cada vuelta es un fotograma que lee la acción, actualiza la física y comprueba los eventos. Todos los motores de juegos tienen este bucle, aunque lo ocultan detrás de métodos como `Update()` o `_physics_process()`.

Resolver las colisiones eje por eje es lo que hace que el personaje se deslice por las paredes y aterrice limpio en el suelo. Si se mueve en diagonal y se corrige todo a la vez, no se sabe si ha chocado con la pared o con el suelo, y el personaje se queda enganchado en las esquinas.

La física con enteros y un paso fijo por fotograma hace el juego determinista: las mismas acciones producen siempre el mismo resultado. Es lo que permite grabar repeticiones guardando solo las teclas, y lo que hace posibles estas pruebas automáticas.

Las constantes (velocidad, salto, gravedad, caída máxima) definen el «tacto» del juego. Cambiar el salto de -9 a -10 hace que llegue a plataformas que antes no alcanzaba: en un juego real se ajustan probándolo muchas veces.

15. Inteligencia artificial de un enemigo con una máquina de estados

Difícil · JavaScript · 80 minutos · apuntesdam.com/ejercicios/javascript/ia-enemiga-con-maquina-de-estados

Solución (javascript)

```
const lineas = require("fs").readFileSync(0, "utf8").split("\n");

const VISION = 5, PERDER = 8, ESPERA = 3, VIDA_MAX = 100, HUIR = 30, RECUPERADO = 60;

const distancia = (a, b) => Math.abs(a.x - b.x) + Math.abs(a.y - b.y);
// Un paso en el eje con más distancia (horizontal si empatan), hacia el objetivo o (signo -1) alejándose
function paso(desde, hacia, signo = 1) {
  const dx = hacia.x - desde.x, dy = hacia.y - desde.y;
  if (dx === 0 && dy === 0) return { ...desde };
  return Math.abs(dx) >= Math.abs(dy) ? { x: desde.x + signo * Math.sign(dx || 1), y: desde.y } : { x:
    desde.x, y: desde.y + signo * Math.sign(dy) };
}

function main() {
  const cab = (lineas[0] ?? "").trim().match(/^PATRULLA (-?\d+),(-?\d+) (-?\d+),(-?\d+)$/);
  if (!cab) return console.log("La primera línea debe ser PATRULLA x1,y1 x2,y2");
  const ruta = [{ x: +cab[1], y: +cab[2] }, { x: +cab[3], y: +cab[4] }];
  let yo = { ...ruta[0] }, destino = 1, estado = "PATRULLA", vida = VIDA_MAX, visto = null, esperando =
    0;
  let danoHecho = 0;
  const ticks = { PATRULLA: 0, PERSECUCION: 0, ATAQUE: 0, BUSCAR: 0, HUIDA: 0 };

  const cambiar = (t, nuevo, motivo) => {
    console.log(`t=${t}: ${estado} → ${nuevo} (${motivo})`);
    estado = nuevo;
  };

  let t = 0;
  for (const linea of lineas.slice(1)) {
    const p = linea.trim().split(/\s+/);
    if (!p[0]) continue;
    t++;
    const m = p[0].match(/^(-?\d+),(-?\d+)$/);
    if (!m) { console.log(`t=${t}: línea no válida`); continue; }
    const jugador = { x: +m[1], y: +m[2] };
    const sigilo = p.includes("SIGILO");
    const golpe = p.indexOf("GOLPE") >= 0 ? Number(p[p.indexOf("GOLPE") + 1]) || 0 : 0;
    if (golpe) vida = Math.max(0, vida - golpe);
    if (vida === 0) {
      console.log(`t=${t}: el enemigo cae derrotado`);
      ticks[estado]++;
      break;
    }

    const d = distancia(yo, jugador);
    const ve = d <= VISION && !sigilo;
    if (ve) visto = { ...jugador };

    // Transiciones: la vida baja manda sobre todo lo demás
    if (estado !== "HUIDA" && vida <= HUIR) cambiar(t, "HUIDA", `vida ${vida}`);
    else if (estado === "PATRULLA" && ve) cambiar(t, "PERSECUCION", `te ve a distancia ${d}`);
    else if (estado === "PERSECUCION" && d <= 1) cambiar(t, "ATAQUE", "a tu lado");
    else if (estado === "PERSECUCION" && d > PERDER) cambiar(t, "BUSCAR", `te pierde a distancia ${d}`);
    else if (estado === "ATAQUE" && d > 1) cambiar(t, "PERSECUCION", `te alejas a distancia ${d}`);
    else if (estado === "BUSCAR" && ve) cambiar(t, "PERSECUCION", `te vuelve a ver a distancia ${d}`);
    else if (estado === "HUIDA" && vida >= RECUPERADO) cambiar(t, "PATRULLA", `recuperado con vida
      ${vida}`);

    // Acción del estado actual
    if (estado === "PATRULLA") {
      if (yo.x === ruta[destino].x && yo.y === ruta[destino].y) destino = 1 - destino;
      yo = paso(yo, ruta[destino]);
    } else if (estado === "PERSECUCION") {
      yo = paso(yo, jugador);
    } else if (estado === "ATAQUE") {
      danoHecho += 10;
    }
  }
}
```

```

    } else if (estado === "BUSCAR") {
        if (yo.x !== visto.x || yo.y !== visto.y) {
            yo = paso(yo, visto);
        } else if (++esperando > ESPERA) {
            esperando = 0;
            cambiar(t, "PATRULLA", "no te encuentra");
        }
    } else if (estado === "HUIDA") {
        if (d <= PERDER) yo = paso(yo, jugador, -1);
        else vida = Math.min(VIDA_MAX, vida + 10);
    }
    ticks[estado]++;
}

console.log(`Posición final: (${yo.x}, ${yo.y}) · vida ${vida} · daño causado ${danoHecho}`);
console.log("Turnos por estado: " + Object.entries(ticks).filter(([ , n]) => n).map(([e, n]) => `${e} ${n}`).join(", "));
}

main();

```

Una máquina de estados separa el qué hago (la acción de cada estado) del cuándo cambio (las transiciones). Cada estado es pequeño y fácil de entender por sí solo, y añadir un comportamiento nuevo es añadir un estado y sus transiciones sin tocar el resto.

La prioridad de las transiciones resuelve los conflictos: si el guardia ve al jugador pero tiene poca vida, huye. Escribirlas como una cadena de `else if` ordenada hace explícita esa prioridad.

El guardia tiene memoria (la última posición vista) y temporizadores (los turnos de espera), dos ingredientes que hacen que una IA sencilla parezca inteligente. Con la distancia Manhattan y pasos de una casilla, el movimiento es el de los juegos de rejilla clásicos.

En un motor real, las transiciones se comprobarían en cada fotograma y la acción movería al personaje poco a poco. Para comportamientos más complejos se usan árboles de comportamiento o planificadores, pero casi siempre combinados con máquinas de estados como esta.