

ALGORITMICA I: EPD

EVALUABLE #1

Marcos Velazquez Carmona, Daniel Sanchez-Matamoros Carmona

Contenido

Planteamiento del problema	2
Enunciado.....	2
Objetivos de la Práctica.....	2
Representación en Java.....	3
Funciones a implementar.....	3
Solución propuesta	6
Cálculo de complejidad.....	7
Método recursivo	7
Método iterativo.....	8
Estudio temporal.....	8
Conclusiones	10
Bibliografía	10

Planteamiento del problema

Enunciado

El Mapa del Merodeador muestra los pasillos de Hogwarts y la posición de las puertas mágicas. Harry necesita encontrar el número de caminos posibles para llegar desde la Entrada Principal hasta la Sala de los Menesteres, moviéndose solo hacia abajo o a la derecha en una cuadrícula mágica.

Cada casilla del mapa puede estar:

- Bloqueada (0) → hay una pared encantada.
- Libre (1) → puede moverse.

El programa debe usar recursividad simple para:

1. Calcular el número total de caminos posibles desde (0,0) hasta ($N - 1, N - 1$).
2. Calcular el camino de menor coste si cada celda tiene un valor de energía.
3. Analizar el tiempo de ejecución de la versión recursiva y de una versión iterativa optimizada.

Objetivos de la Práctica

- Aplicar recursividad (solo llamadas hacia abajo y derecha).
- Implementar versiones recursiva pura e iterativa (programación dinámica).
- Realizar un análisis temporal empírico (comparando tiempos y llamadas).
- Comprender la diferencia entre crecimiento exponencial vs. polinómico.
- Trabajar en Java sin objetos, solo con métodos estáticos y arrays.

Representación en Java

El mapa de Hogwarts es una matriz cuadrada $N \times N$ de números enteros:

```
int[][] mapa = {  
    {1, 1, 1, 1},  
    {1, 0, 1, 1},  
    {1, 1, 1, 0},  
    {1, 1, 1, 1}  
};
```

- 1 → pasillo transitable
- 0 → obstáculo mágico

Harry empieza en (0,0) y debe llegar a ($N - 1, N - 1$).

Funciones a implementar

FUNCIÓN 1 – MOSTRAR EL MAPA.

```
void mostrarMapa(int[][] mapa)
```

- Imprime la matriz con formato legible

FUNCIÓN 2 — NÚMERO DE CAMINOS POSIBLES (RECURSIVA).

```
int contarCaminosRec(int[][] mapa, int i, int j)
```

- Devuelve el número total de caminos válidos desde (i, j) hasta la meta, moviéndose solo hacia abajo o derecha

Reglas:

- Si (i, j) está fuera del mapa → devolver 0
- Si (i, j) es una pared (0) → devolver 0
- Si (i, j) es la meta → devolver 1
- En otro caso: Habrá que contar los caminos recursivos

FUNCIÓN 3 — NÚMERO DE CAMINOS (ITERATIVA OPTIMIZADA)

```
int contarCaminosIter(int[][] mapa)
```

- Utiliza programación dinámica con una matriz DP de tamaño $N \times N$:

$dp[i][j] = dp[i-1][j] + dp[i][j-1]$

siempre que $mapa[i][j] == 1$.

FUNCIÓN 4—CAMINO DE MENOR COSTE (RECURSIVA)

En otra versión del mapa, cada casilla de una matriz llamada energía tiene un coste que viene dado por un número natural:

```
int[][] energia = {
    {1, 3, 1},
    {1, 5, 1},
    {4, 2, 1}
};
```

Implementa:

```
int caminoMinimoRec(int[][] energia, int i, int j)
```

- Devuelve el coste mínimo total desde (i,j) hasta la meta:

```
return energia[i][j] + Math.min(
    caminoMinimoRec(energia, i+1, j),
    caminoMinimoRec(energia, i, j+1)
);
```

Condiciones base:

- Si sale del mapa → devuelve Integer.MAX_VALUE
- Si llega a la meta → devuelve $energia[i][j]$

FUNCIÓN 5—MEDICIÓN DE TIEMPOS

```
long medirTiempo(Runnable f)
```

- Ejecuta una función y devuelve el tiempo en nanosegundos:

```
long inicio = System.nanoTime();
f.run();
long fin = System.nanoTime();
return fin - inicio;
```

FUNCIÓN 6 — MAIN

```
public static void main(String[] args)
```

Debe:

1. Leer un mapa desde un CSV (o definirlo manualmente).

2. Ejecutar:

- contarCaminosRec(mapa, 0, 0)
- contarCaminosIter(mapa)
- caminoMinimoRec(energia, 0, 0)

3. Medir el tiempo de ejecución de cada método.

4. Mostrar resultados:

```
--- EL MAPA DEL MERODEADOR ---
```

Tamaño del mapa: 6x6

Caminos posibles (recursivo): 133

Tiempo: 18.230 ms

Caminos posibles (iterativo): 133

Tiempo: 0.421 ms

Coste mínimo del recorrido mágico: 23

Ejemplo de mapa (5x5):

1,1,1,1,1

1,0,1,1,1

1,1,1,0,1

1,1,1,1,1

1,1,1,1,1

Resultado esperado:

- Caminos posibles: 84
- Coste mínimo (si se usa matriz de energía): depende del input.

Solución propuesta

Para resolver este problema, hemos estructurado la solución de tal forma que tenemos una clase principal Main desde la que llamamos a las diferentes funciones, contenidas en archivos separados .java.

Lo hemos decidido así para aportar una solución más modular y limpia, ya que facilita por ejemplo localizar errores más fácilmente y tener los códigos separados lo que ayuda a una mejor organización.

A continuación, pasamos a explicar y detallar las diferentes funciones que hemos desarrollado para nuestro programa, con objetivo de comprender porque lo hemos decidido así y cómo funcionan:

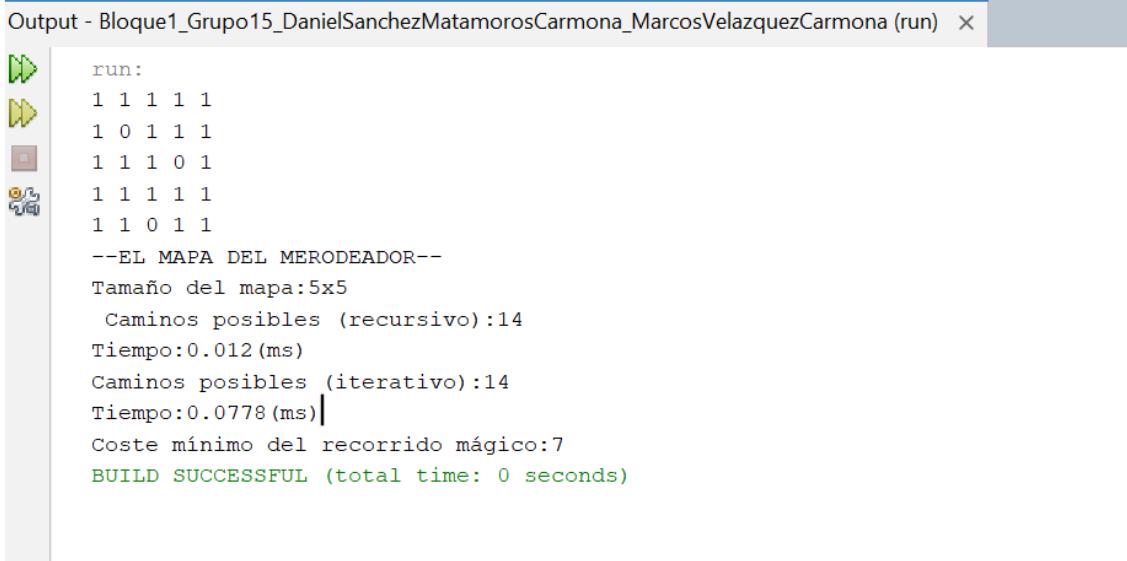
- **leerMatrizCSV:** Utilizamos esta clase para leer los mapas que nos vienen dados en ficheros .CSV. Tenemos un método llamado `leerMapa` en el que implementamos `BufferedReader` que aprendimos en otra asignatura anterior, y allí vamos leyendo línea por línea el fichero con un bucle `while` y el método `readLine()`.
- **MostrarMapa:** Una vez volcado el mapa a un array de tipo `int`, procedemos a recorrerlo con un bucle e imprimirla por pantalla, eso lo hacemos en esta clase.
- **NúmerodeCaminosPosibles_Iter:** El recorrido del mapa de forma iterativa ocurre en esta clase.
- **NúmerodeCaminosPosibles_recursiva:** En esta clase, establecemos un caso base, los casos generales y las llamadas recursivas al método para el cálculo de los diferentes caminos posibles que pueda tener el array pasado por parámetros.
- **CaminoDeMenorCoste:** En base a una matriz, llamada energía, el programa será capaz de identificar que camino es el que cuesta menos, es decir, se irá moviendo hacia abajo y hacia la derecha buscando los valores más pequeños de tal forma que cuando llega al destino final, la suma de dichos valores elegidos para ese camino sean los más pequeños posibles.
- **Energía:** En esta clase simplemente le pasamos la matriz de energía, mencionada anteriormente.
- **MediciónTiempos:** llamamos al método `medirTiempo (Runnable f)` de esta clase donde calculamos el tiempo que tarda en procesar el cálculo que le pasemos por parámetro, ya sea recursiva o iterativa. Simplemente, calculamos el

tiempo al inicio con `System.nanoTime` y calculamos de nuevo al finalizar, de tal forma que restamos $fin - inicio$.

Luego lo ajustamos para que se muestre en milisegundos.

Por último, mostramos los resultados por pantalla, utilizando un `System.out.println`.

Adjuntamos una captura:



```
Output - Bloque1 Grupo15_DanielSanchezMatamorosCarmona_MarcosVelazquezCarmona (run) ×
run:
1 1 1 1 1
1 0 1 1 1
1 1 1 0 1
1 1 1 1 1
1 1 0 1 1
--EL MAPA DEL MERODEADOR--
Tamaño del mapa:5x5
Caminos posibles (recursivo):14
Tiempo:0.012 (ms)
Caminos posibles (iterativo):14
Tiempo:0.0778 (ms)
Coste mínimo del recorrido mágico:7
BUILD SUCCESSFUL (total time: 0 seconds)
```

Cálculo de complejidad

Vamos a analizar las llamadas recursivas e iterativas para poder clasificarlas:

Método recursivo

El algoritmo explora todas las posibles combinaciones de movimientos.

Cada llamada recursiva genera a su vez tres nuevas llamadas, salvo el caso base. Por tanto, en el peor de los casos, sin obstáculos ni salidas prematuras, el número de llamadas crecería exponencialmente:

$$T(n) = 3T(n - 1)$$

Por tanto: $T(n) \in O(3^n)$

Método iterativo

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        ...
    }
}
```

Analizando el código de la llamada iterativa tenemos un bucle doble `for`, que nos lleva a concluir:

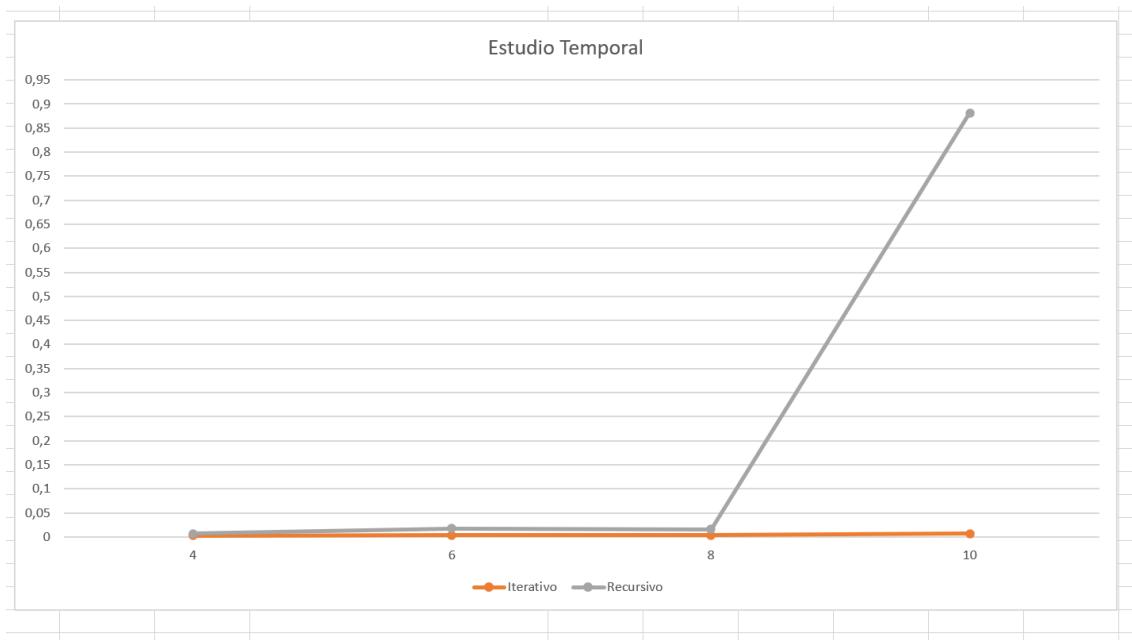
$$n \times n = n^2$$

Por tanto: $T(n) = O(n^2)$

Estudio temporal

6x6	Iterativo	0,004	0,0041	0,0046	0,0042	0,0043	0,0041	0,004	0,0051	0,0044
	Recursivo	0,0174	0,0178	0,0232	0,0194	0,0182	0,0176	0,0178	0,0217	0,0204
8x8	Iterativo	0,0042	0,0042	0,0045	0,0041	0,0047	0,0043	0,0041	0,0042	0,0043
	Recursivo	0,017	0,0188	0,0188	0,0171	0,0201	0,0174	0,0169	0,0178	0,0183
4x4	Iterativo	0,0032	0,0032	0,0035	0,0034	0,0035	0,0032	0,003	0,003	0,0034
	Recursivo	0,0082	0,0077	0,0086	0,0089	0,0076	0,0077	0,0075	0,0073	0,0076
10x10	Iterativo	0,0066	0,007	0,0069	0,0071	0,0067	0,0091	0,0071	0,0102	0,007
	Recursivo	1	1,0611	0,9186	0,9578	0,9788	1,0457	1,0369	0,9142	0,8826

Tamaño	Iterativo	Recursivo
4	0,00294	0,00711
6	0,00388	0,01735
8	0,00386	0,01622
10	0,00677	0,88158



Conclusiones

Para concluir que el programa funciona según lo esperado, hemos probado metiéndole varias matrices diferentes a través de los ficheros .CSV, de tal forma nos aseguramos de que el resultado es consistente.

Por otra parte, nos aseguramos de que los caminos posibles sean los mismos, ya sea el cálculo de forma recursiva como iterativa, porque no tendría sentido que se mostraran valores diferentes, de hecho, en algún momento nos pasó, y el haber llegado a esta conclusión, nos ayudó a encontrar el error.

Interpretando los resultados, hemos probado a cargar el programa 10 veces con la misma matriz de 6×6 y anotar los tiempos, lo mismo para 4×4 , 8×8 , y 10×10 : Lo cual corresponde con la primera captura adjunta en el estudio temporal. Luego, hemos obtenido las medias de dichos valores y hemos generado una gráfica lineal donde en el eje de abscisas se representa el tamaño y en el eje de ordenadas se representan los tiempos. Los tiempos iterativos son siempre más bajos que los recursivos, y son tiempos generalmente muy parecidos, mientras que los recursivos han ido creciendo a medida que las matrices son más grandes, siendo la matriz de 10×10 un tiempo bastante mayor. Aunque también ha influido que hemos añadido más 0 (puertas) en la matriz, con lo cual se incrementa el número de caminos posibles y por tanto, aumentan los cálculos y el tiempo de ejecución.

En la matriz de energía al igual que hicimos con la carga de los .CSV, hemos probado a cambiar los valores para probar que funcione como se espera.

Bibliografía

Para el desarrollo de esta EPD evaluable no hemos necesitado consultar fuentes externas más allá de consultar los apuntes de EB subidos al aula virtual para un soporte teórico y consultar los ejercicios resueltos de las EPDs para un soporte práctico.