# ALGORITMICA I: EPD EVALUABLE #1

Marcos Velazquez Carmona, Daniel Sanchez-Matamoros Carmona

# Contenido

# 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 $(\mathrm{N}-1, \mathrm{N}-1)$.

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 imprimirlo por pantalla, eso lo hacemos en esta clase.

- `NúmerodeCaminosPosibles_Iter`: El recorrido del mapa de forma iterativa ocurre en esta clase.

- `NumerodeCaminosPosibles_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.

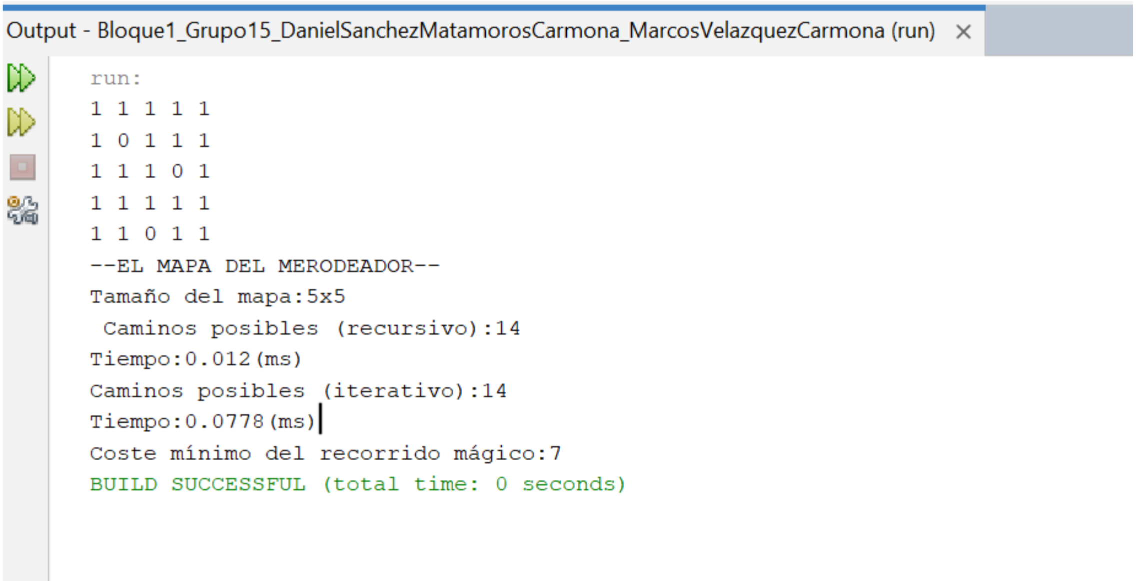
- `Energía`: En esta clase simplemente le pasamos la matriz de energía, mencionada anteriormente.

- `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:




# Cálculo de complejidad

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

## Método recursivo

Llamamos $m = (n - 1)$

- ### Peor de los casos:

  El número de caminos posibles abarca desde $(0,0)$ hasta $(m, m)$ moviéndonos solo hacia abajo y hacia la derecha es: $L(m) = \binom{2m}{m}$.

Cada camino completo corresponde a una ruta completa. La estructura del árbol de llamadas sigue la siguiente estructura (siendo el caso ideal de un árbol binario completo): $nodos = 2 \cdot (caminos - 1)$

Por tanto, la complejidad temporal en el peor caso es $L(i,j) = \frac{i!+j!}{i!j!}$, para el peor de los casos, nos queda: $i = j = (n-1) = m$. Por lo que podemos sustituir y nos queda: $L(m) = \frac{(2m)!}{(m!)^2}, \forall\, m = n - 1$.

Aproximando factorial por la fórmula de Stirling: $m! \approx \sqrt{2\pi m}\left(\frac{m}{e}\right)^m$, y sustituyendo $m = (n-1)$, obtenemos lo siguiente: $T(n) = \left(\frac{4^{n-1}}{\sqrt{n-1}}\right) \in O(4^n)$.

- ## Mejor de los casos

  El mejor caso se da cuando la recursión se corta inmediatamente. Esto implica que solo realizamos una llamada a la función, por lo que $T(n) = 1 \in O(1)$.

- ## Caso promedio

  Para elegir un caso promedio debemos asumir que cada celda es transitable con probabilidad $p \in [0,1]$.

  Por lo que nos queda, $L(m) = \left(\frac{2m!}{(m!)^2}\right)p^{2m}, \forall\, m = n - 1$. El tiempo esperado viene siendo $T(n) = \left(\frac{4p^{2n-2}}{\sqrt{n-1}}\right) \in O(4p^{2n})$. Podemos interpretar que si $0 \le 4p^2 \le 4$, claramente estamos en un caso promedio.

  Para $p = 0.5$ (al ser el punto intermedio del intervalo), nos queda: $T(n) = (4 \cdot 0.5^2)^n = n \in O(n)$

# Método iterativo

El método iterativo viene dado por la fórmula:

$$caminos_{i,j} = \begin{cases} 0, si\ mapa_{i,j} = 0 \\ caminos_{i-1,j} + caminos_{i,j-1}, si\ mapa_{i,j} = 1 \end{cases}$$

- ## Peor de los casos:

El algoritmo recorre la matriz con dos bucles anidados:

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

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

Lo que matemáticamente se puede ver como: $\sum_{i=1}^{n}\sum_{j=1}^{n} c$.

Por tanto, tenemos que $T(n) = c \cdot n^2 \in O(n^2)$.

- ### Mejor de los casos:

  El mejor caso es cuando estamos en la meta, por lo que $T(n) = 1 \in O(1)$.

- ### Caso promedio:

  El caso promedio, viene siendo igual que el peor de los casos, pero sin multiplicar por constantes escalares. $T(n) = n^2 \in O(n^2)$.

# Estudio temporal

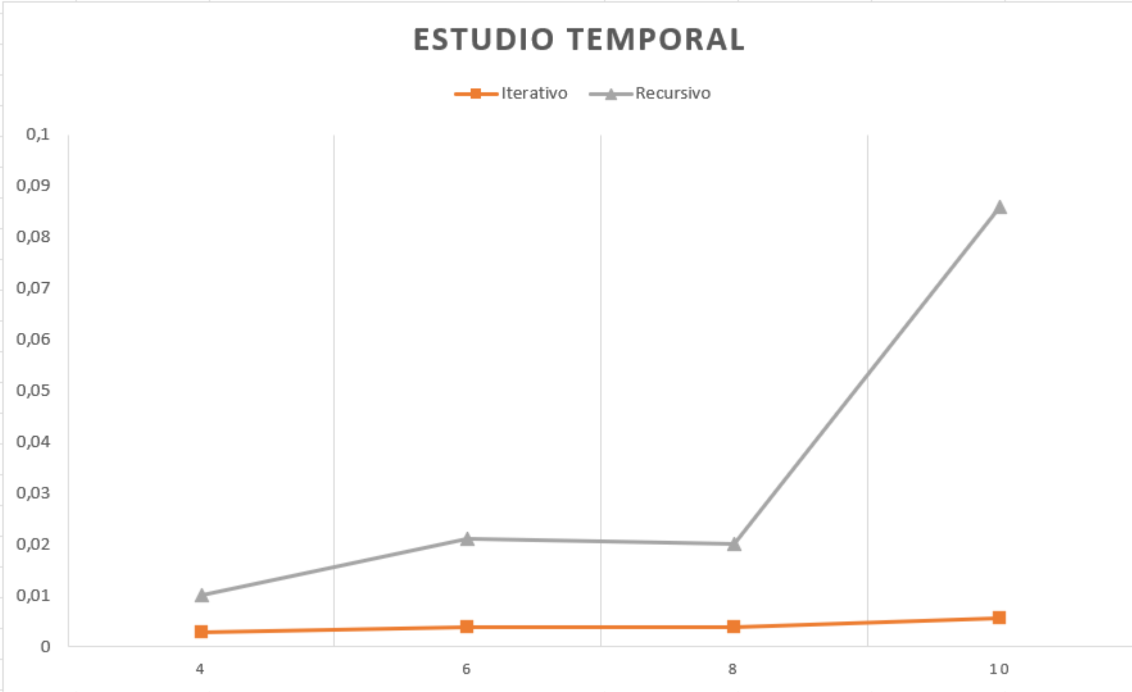
La siguiente tabla muestra los valores para un método iterativo y recursivo para matrices cuadradas de tamaños $4 \times 4$, $6 \times 6$, $8 \times 8$, y $10 \times 10$.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 | 0,006101 | 0,006299 | 0,006399 | 0,0071 | 0,006099 | 0,006099 | 0,005999 | 0,0058 | 0,0059 |
| | Recursivo | 0,0891 | 0,09 | 0,086301 | 0,0911 | 0,0897 | 0,0893 | 0,0877 | 0,0902 | 0,089601 |

La siguiente tabla muestra el promedio de los valores.

| Tamaño | Iterativo | Recursivo |
|---|---|---|
| 4 | 0,00294 | 0,00711 |
| 6 | 0,00388 | 0,01735 |
| 8 | 0,00386 | 0,01622 |
| 10 | 0,0055796 | 0,0803002 |

La siguiente gráfica muestra el estudio temporal de ambos métodos:



# 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 $6 \times 6$ y anotar los tiempos, lo mismo para $4 \times 4$, $8 \times 8$, y $10 \times 10$: 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, obteniendo con la matriz de $10 \times 10$ un tiempo bastante mayor.

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 práctica evaluable hemos consultado los apuntes de EB Y EPD de la asignatura *Algoritmica I* tanto como para un soporte práctico como teórico. También hemos recurrido a otras asignaturas del grado, como pueden ser *Ingeniería del Software I*, para recordar cómo leer archivos `.csv` que contienen una matriz, y transformar este archivo a una matriz de enteros leíble por el `JDK`.

También hemos tenido que recurrir a la asignatura *Cálculo* de primero, para recordar procedimientos matemáticos que utilizamos para calcular la complejidad, tanto de la versión recursiva como de la iterativa, como pueden ser la fórmula de Stirling y la simplificación que se utiliza a través de dicha fórmula.