UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

GRADO EN INGENIERÍA INFORMÁTICA

Técnicas de Programación Avanzada

Profesor:

- *BORJA MONSALVE PIQUERAS*

Alumnos:

- *PAULA SAENZ DE SANTA MARÍA DÍEZ*
- *ANDRÉS RAMOS GARCÍA*

Ejercicio 1

```java
public class Iguales {
	/**
	 * Función booleana que compara dos matrices de números enteros, aplicando una
	 * estrategía de Divide y Vencerás.
	 *
	 * @param matriz1 Matriz AxB de números enteros
	 * @param m2 Matriz CxD de números enteros
	 * @return true/false en función de si las matrices son iguales
	 */
	private static boolean sonIguales(int[][] m1, int[][] m2, int iniF, int iniC, int filas, int columnas) {
		if (filas <= 0 || columnas <= 0) {
			return true;
		}
		if (filas == 1 && columnas == 1) {

			return m1[iniF][iniC] == m2[iniF][iniC];
		}
		int mitadF = filas / 2;
		int mitadC = columnas / 2;

		if(
		  !sonIguales(m1, m2, iniF, iniC, mitadF, mitadC) ||
		  !sonIguales(m1, m2, iniF, iniC + mitadC, mitadF, columnas - mitadC) ||
		  !sonIguales(m1, m2, iniF + mitadF, iniC, filas - mitadF, mitadC) ||
		  !sonIguales(m1, m2, iniF + mitadF, iniC + mitadC, filas - mitadF, columnas - mitadC)
		  ) {
			return false;
		}
		return true;
	}// sonIguales

	public static boolean iguales_DV(int[][] matriz1, int[][] matriz2) {
		if (
		  matriz1.length != matriz2.length || matriz1[0].length != matriz2[0].length ||
		  !sonIguales(matriz1, matriz2, 0, 0, matriz1.length, matriz1[0].length)
		  ) {
			return false;
		}
		return true;
	}// iguales_DV
```

La función auxiliar *sonIguales* tiene como objetivo principal realizar una comparación recursiva entre dos matrices para determinar si son iguales o no. Se utiliza en la función principal *iguales_DV* para implementar el enfoque de divide y vencerás.

La función *iguales_DV* no podía ser modificada, lo que significa que no podíamos cambiar los argumentos pasados a la función ni la lógica interna. Dado que la función principal requiere trabajar con los argumentos dados, llegamos a la conclusión de que

necesitábamos una manera de realizar la comparación de manera eficiente dentro de estas restricciones.

Por está misma razón creamos la función auxiliar *sonIguales*, ya que al decidir qué parámetros se pasan en los argumentos podríamos llegar a una solución de una manera más eficiente. Al pasar estos parámetros por referencia en las llamadas recursivas, se redimensionan las operaciones para que se ajusten al tamaño de la submatriz actual.

Es por está razón que hemos decidido implementar el uso de la función auxiliar *sonIguales.*

Ejercicio 2.1

Escribir el **pseudocódigo** de la función pedida.

Función: iguales_DV…

```
funcion iguales_DV(matriz1, matriz2: matriz [1..n][1..n] de enteros): booleano
    si (
        matriz1.longitud() != matriz2.longitud() || matriz1[0].longitud() != matriz2[0].longitud() OR
        !sonIguales(matriz1, matriz2, 0, 0, matriz1.longitud(), matriz1[0].longitud)
    ) entonces
        devolver falso
    devolver verdadero
```

Función: sonIguales…

```
función sonIguales(
    m1, m2 : matriz [1…n][1….n] de enteros, iniF, iniC, filas, columnas : enteros
): booleano
    si (filas <= 0 OR columnas <= 0) entonces
        devolver verdadero
    si (filas == 1 AND columnas == 1) entonces
        devolver matriz1(iniF)(iniC) == matriz2(iniF)(iniC)

    mitadF = filas/2
    mitadC = columnas/2

    si (
        !sonIguales(m1, m2, iniF, iniC, mitadF, mitadC) OR
        !sonIguales(m1, m2, iniF, iniC + mitadC, mitadF, columnas - mitadC) OR
        !sonIguales(m1, m2, iniF + mitadF, iniC, filas - mitadF, mitadC) OR
        !sonIguales(m1, m2, iniF + mitadF,iniC + mitadC, filas - mitadF, columnas - mitadC)
    ) entonces
        devolver falso
    devolver true
```

Ejercicio 2.2

Hacer un esquema **gráfico** de la idea de descomposición planteada.

Pongamos un ejemplo de una matriz 4x4 de color azul.

[0, 0]	[0, 1]	[0, 2]	[0, 3]
[1, 0]	[1, 1]	[1, 2]	[1, 3]
[2, 0]	[2, 1]	[2, 2]	[2, 3]
[3, 0]	[3, 1]	[3, 2]	[3, 3]

Una vez llamada la recursividad, ésta se divide en 4 partes, como puede comprobarse de colores amarillo, verde, morado y rojo.

[0, 0]	[0, 1]	[0, 2]	[0, 3]
[1, 0]	[1, 1]	[1, 2]	[1, 3]
[2, 0]	[2, 1]	[2, 2]	[2, 3]
[3, 0]	[3, 1]	[3, 2]	[3, 3]

Como todavía no se llega a entrar al condicional para comparar las matrices (porque tiene que llegar a ser una submatriz de 1x1), vuelve a hacer la división en 4 partes, representadas en los colores anteriores.

[2, 0]	[2, 1]
[3, 0]	[3, 1]

Al llegar a este punto, entra en el condicional y compara el elemento que tiene dentro de ambas matrices de la misma posición.

Si son iguales devuelve true, si no devuelve false. Si devuelve false, gracias a la lógica de las llamadas recursivas, sale del bucle y no termina de comparar todos los elementos, ya que cuando uno es distinto, ya no pueden ser iguales nunca.

Ejercicio 3

Calcular la complejidad del código generado, utilizando la reducción correspondiente y desarrollando los razonamientos.

Caso base Caso general

```
private static boolean sonIguales(int[][] m1, int[][] m2, int inF, int iniC, int filas, int columnas) {
    if (filas <= 0 || columnas <= 0) {
        return true;
    }
    if (filas == 1 && columnas == 1) {
        return m1[iniF][iniC] == m2[iniF][iniC];
    }
    int mitadF = filas / 2;
    int mitadC = columnas / 2;
    if(
        !sonIguales(m1, m2, iniF, iniC, mitadF, mitadC ) ||
        !sonIguales(m1, m2, iniF, iniC+ mitadC , mitadF , columnas - mitadC ) ||
        !sonIguales(m1, m2, iniF + mitadF , iniC, filas - mitadF , mitadC ) ||
        !sonIguales(m1, m2, iniF+ mitadF , iniC+ mitadC , filas - mitadF , columnas - mitadC)
    ) {
        return false;
    }
    return true;
}// sonIguales

public static boolean iguales_DV(int[][] matriz1, int[][] matriz2) {
    if (
        matriz1.length != matriz2.length || matriz1[0].length != matriz2[0].length ||
        !sonIguales(matriz1, matriz2, 0, 0, matriz1.length, matriz1[0].length)
    ){
        return false;
    }
    return true;
}// iguales_DV
```

Para calcular la complejidad de este código, primero analizamos su estructura. Observamos que la función "SonIguales" realiza llamadas recursivas, lo que nos indica que tenemos que seguir sus reglas para calcular su complejidad.

Luego, debemos elegir la técnica adecuada para abordar el análisis, ya sea reducción por sustracción o por división. En este caso, concluimos que la reducción por división es la apropiada, dado que cada subproblema resultante de la llamada recursiva representa una cuarta parte del problema original, como se indicó en el ejercicio 2.

Por lo tanto, empleamos:

$$T(n) = \begin{cases} c_0 * n^{k_0} & si\ n >= k \\ a * T(n/b) + c_1 * n^k & si\ n >= k \end{cases}$$

Para poder solucionar esto, lo primero que hacemos es separar nuestro código en caso base y caso general:

Los dos primeros condicionales (subrayados en amarillo en el codigo) pertenecen al caso base, ya que por definición el caso base es aquel que proporciona la condición de parada de la recursividad, y en este caso si el tamaño de la submatriz es cero o negativo, o si es una matriz de 1x1, no se realiza más subdivisión y la función devuelve un resultado directamente.

En cambio el caso general es aquel en el cual se hace el problema cada vez más pequeño (intentando que todos los subproblemas sean parecidos en cuanto a tamaño), Las variables mitadF y mitadC, es donde se divide el número de filas y columnas respectivamente por 2, se realiza para la división recursiva del problema original en subproblemas más pequeños. Las 4 llamadas recursivas comparan una submatriz de la primera matriz con la submatriz correspondiente de la segunda matriz. Por último, dentro del caso general tenemos ambos return (true y false). El return false se utiliza cuando se encuentra al menos un elemento que no coincide en las comparaciones de las submatrices. Esto detiene la ejecución y devuelve false, indicando que las matrices completas no son iguales. Por otro lado, el return true se ejecuta cuando todas las comparaciones de las submatrices han sido exitosas.

Una vez hecha la división de los casos, calculamos la complejidad:
Para el caso base todas las operaciones realizadas son de orden constante $O(1)$.

Para el caso general, tenemos que asignar los valores de a, b y k:
a: Representa el número de llamadas recursivas idénticas en cada invocación del programa. En este caso, hay cuatro llamadas recursivas idénticas. **(a = 4)**
b: Representa el tamaño de los subproblemas generados. En este caso, el tamaño de los subproblemas generados es ¼ del tamaño original de la matriz, ya que la matriz se divide en cuatro partes iguales en cada llamada recursiva. **(b = 4)**
k: Representa el costo de las instrucciones que no son llamadas recursivas. En nuestra función podemos observar que todo aquello que no es llamada recursiva tiene un orden constante ($O(1)$), por lo que sería 1, haciendo que k sea igual a 0. **(k = 0)**

Con estos valores, vemos cuál es la que se adapta a nuestro caso:

$$T(n) \in O(n^k)\ si\ a < b^k$$
$$T(n) \in O(n^k log_b\ n)\ si\ a = b^k$$
$$T(n) \in O(n^{log_b a})\ si\ a > b^k$$

Siendo la respuesta : $T(n) \in O(n^{log_b a})\ si\ a > b^k$, por lo que nuestro resultado es:

$$O(n^{log_4 4}) = O(n^1) = O(n)$$

Siendo *n* el número total de elementos en las matrices de entrada.

Ejercicio 4

Explicar cuál sería la complejidad de haber seguido un enfoque que no fuera "Divide y vencerás".

La principal diferencia con respecto al enfoque "Divide y vencerás" es la utilización de bucles "while" para su correcto funcionamiento. Esto genera un cambio de complejidad y por ende una diferente velocidad de procesamiento, interfiriendo en la eficiencia.

Al no recurrir a la recursividad y sí a los bucles "while", deben usarse métodos más simples para calcular su eficiencia, ya que ésta depende de los bucles internos de la función. Para ello deben usarse el siguiente método en los bucles:

$$\sum_{i=1}^{n} = O(Condición) + O(Cuerpo)$$

Dependiendo del resultado que nos dé la primera fórmula, se usarán uno de estos métodos:

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Si ponemos en práctica estos métodos en nuestro código, sustituyendo cada valor por el que nos corresponde, el resultado final acaba siendo:

$$ab - a + 2b + 3 \in O(ab)$$

Teniendo en cuenta que **a = matrizA[0].length** y **b = matrizA.length**, sacando en conclusión que la eficiencia depende de la cantidad de filas y columnas de las matrices.

Pero aún no se ha acabado, hay que tener en cuenta que tenemos que sacar el **PEOR CASO** en cada código, de esta forma podemos ver la peor opción posible y saber cuál es el límite de tiempo máximo que tarda en ejecutarse. La peor opción posible sería que las matrices tengan el **mismo número de filas y de columnas**, o lo que es lo mismo, **a = b**. Por lo tanto, la ecuación final cambiaría un poco.

$$a^2 + a + 3 \in O(a^2)$$

Teniendo como resultado que la complejidad es del siguiente orden: $O(a^2)$

Ejercicio 5

Argumentar cuál de las dos versiones sería la mejor.

Tras haber sacado el resultado del código que utiliza el “Divide y vencerás”, siendo $O(n)$ y del código que no usa esta metodología, siendo $O(a^2)$. Lllegamos a la conclusión de que, en este caso, la opción que utiliza el “Divide y Vencerás” para su funcionamiento es más eficiente que la que utiliza bucles “while”, ya que requiere menos tiempo de ejecución que su contraparte.

$$O(n) < O(n^2)$$