26-2-2024

Ejercicios Unidad 1 TPA

ALBERTO RAMOS LOPEZ

Ejercicio 1

Calcular el tiempo de ejecución de los siguientes fragmentos de código

A	sum = 0;
	for (int i = 0; i < n; i++)
	  for (int j = 0; j < n*n; j++)
	    sum ++;
		B	sum = 0;
			for (int i = 0; i < n; i++)
			  for (int j = 0; j < i; j++)
			    sum ++;

C	sum = 0;
	for (int i = 0; i < n; i++)
	  for (int j = 0; j < i*i; j++)
	    for (int k = 0; k < j; k++)
	      sum++;
		D	sum = 0;
			for (int i = 0; i < n; i++)
			  for (int j = 0; j < i*i; j++)
			    if (j % 2 == 0)
			      for (int k = 0; k < j; k++)
			        sum ++;

E	int i = 1;
	int x = 0;
	while (i <= n) {
	  x++;
	  i += 2;
	}		

A:

```
sum = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n * n; j++)
        sum++;
```

- Este fragmento contiene dos bucles anidados. El bucle exterior se ejecuta `n` veces, y el bucle interior se ejecuta `n^2` veces.
- El tiempo de ejecución es **O(n^3)**.

B:

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

- Aquí, el bucle interior se ejecuta solo hasta `i`, lo que reduce la cantidad de iteraciones.
- El tiempo de ejecución es **O(n^2)**.

C:

```
sum = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < i * i; j++)
        for (int k = 0; k < j; k++)
            sum++;
```

- El bucle externo se ejecuta n veces, el segundo bucle se ejecuta `i^2` veces para cada valor de i, y el tercer bucle se ejecuta j veces para cada valor de j.
- Tiene sentido que al introducir un valor de 2 de 0 ya que este se va reduciendo n-1 veces hasta quedar en 0 y al introducir 3 este se queda en 1.
- $\sum i = 0\ n - 1 \sum j = 0\ i^2 - 1 \sum k = 0\ j - 1\ \ 1$ esta sería la ecuación que la corresponde.
- El tiempo de ejecución es **O(n^3)**.

E:

```
int i = 1;
int x = 0;
while (i <= n) {
    x++;
    i += 2;
}
```

- Este fragmento utiliza un bucle `while` que incrementa `i` en 2 en cada iteración.
- El tiempo de ejecución es **O(n/2)**, que se simplifica a **O(n)**.

Ejercicio 2

Primo.java

```java
1 public class Primo {
2 
3     public static boolean esPrimo(int numero) {
4         if (numero <= 1) {
5             return false;
6         }
7         for (int i = 2; i <= Math.sqrt(numero); i++) {
8             if (numero % i == 0) {
9                 return false;
10            }
11        }
12        return true;
13    }
14 
15    public static void main(String[] args) {
16        int numero = 17;
17        if (esPrimo(numero)) {
18            System.out.println(numero + " es primo.");
19        } else {
20            System.out.println(numero + " no es primo.");
21        }
22    }
23 }
24 
```

En la línea 7 es posible cambiar el ***Math.sqrt*** por ***i * i*** ya que también valida la propiedad que un numero multiplicado por sí mismo sea indivisible por otro que no sea si mismo.

Ejercicio 3:

Calcular la complejidad de la función A indicada a continuación:

```
public static int funcion_A (int n) {
        int sum = 0;
        for (int i=0; i<n; i++) {
            if (esPrimo(n))
                sum = sum + n;
            else
                sum ++;
        }
        return sum;
}
```

- Si asumimos que `esPrimo(n)` tiene una complejidad de **O($\sqrt{n}$)**, entonces la complejidad total de `funcion_A` sería **O($n * \sqrt{n}$)**.

  (Creo que debería de ser así pero no me queda muy claro)

Ejercicio 4:

NumeroPerfecto.java

```java
public class NumeroPerfecto {
    public static boolean esNumeroPerfecto(int numero) {
        if (numero <= 1) {
            return false;
        }

        int sumaDivisores = 1;

        for (int i = 2; i <= Math.sqrt(numero); i++) {
            if (numero % i == 0) {
                sumaDivisores += i;
                if (i != numero / i) {
                    sumaDivisores += numero / i;
                }
            }
        }

        return sumaDivisores == numero;
    }

    public static void main(String[] args) {
        int numero = 28;

        if (esNumeroPerfecto(numero)) {
            System.out.println(numero + " es un número perfecto.");
        } else {
            System.out.println(numero + " no es un número perfecto.");
        }
    }
}
```

- El bucle **for** se ejecuta hasta la raíz cuadrada de **numero** (**i <= Math.sqrt(numero)**), lo que reduce el número de iteraciones y mejora la eficiencia.
- En el bucle, verificamos si **i** es un divisor de **número**. Si es así, actualizamos la suma de divisores y también consideramos el divisor complementario (**numero / i**).
- La complejidad de esta función es O(√n), donde n es el número dado. Esto se debe a que la verificación de los divisores se realiza hasta la raíz cuadrada de **número**.

Ejercicio 5:

MatricesIguales.java

```java
public class MatricesIguales {

    public static boolean sonMatricesIguales(int[][] matriz1, int[][] matriz2) {

        if (matriz1 == null || matriz2 == null) {
            return false;
        }

        int n = matriz1.length;

        if (matriz2.length != n) {
            return false;
        }

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matriz1[i][j] != matriz2[i][j]) {
                    return false;
                }
            }
        }

        return true;
    }

    public static void main(String[] args) {

        int[][] matrizA = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
        int[][] matrizB = { { 1, 2, 3 }, { 4, 5, 4 }, { 7, 8, 9 } };

        boolean resultado = sonMatricesIguales(matrizA, matrizB);
        System.out.println("Las matrices son iguales: " + resultado);
    }
}
```

- La función utiliza dos bucles anidados para recorrer todos los elementos de las matrices. Ambos bucles tienen una complejidad de O(N^2), donde N es el orden de las matrices.
- La función realiza un número constante de operaciones dentro de los bucles, como comparaciones de elementos.
- Por lo tanto, la complejidad total de la función es O(N^2), donde N es el orden de las matrices.