Entrega 4

34 Un algoritmo es un conjunto finito de instrucciones o reglas bien definidas y no ambiguas que describen un proceso o procedimiento específico que debe seguirse para resolver un problema o llevar a cabo una tarea en un número finito de pasos.

35
```
public static int suma(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + suma(n - 1);
    }
}
```

36 En términos de límites de cocientes de funciones, la notación O(n) se refiere a la complejidad computacional de un algoritmo que crece linealmente con respecto al tamaño de la entrada (n).

37 La complejidad temporal de este cálculo en función de t sería O(1), es decir, constante.

38 Ambos métodos tienen complejidad temporal constante (O(1)) porque el tiempo requerido para acceder a un elemento en una posición específica de un ArrayList no depende del tamaño de la lista, sino del índice específico proporcionado.

39 La complejidad temporal de este algoritmo es O(n), donde n es el número proporcionado como entrada. Esto se debe a que el número de operaciones realizadas es proporcional al tamaño de la entrada, que en este caso es n.

40 La complejidad de tiempo del algoritmo dado es O(n), donde 'n' representa el tamaño del array. Esto significa que el tiempo que lleva ejecutar el algoritmo aumenta linealmente con el tamaño del array.

41 La complejidad del algoritmo sigue siendo O(n), donde 'n' representa el tamaño del array.

42 A × A será n^2.

43 Si tenemos una matriz con f filas y c columnas, y el algoritmo ejecuta una operación para cada elemento de la matriz, entonces el tiempo de ejecución del algoritmo se puede expresar como O(f * c).

44
```
public static boolean buscar(int e, int[] array) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == e) {
```

```
            return true; // Se encontró el elemento
        }
    }
    return false; // No se encontró el elemento
}
45 import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int e = 6;
        System.out.println("¿El número " + e + " está en el array? " + complejidad(array,
e));
    }

    public static boolean complejidad (int[] array, int e) {
        System.out.println(Arrays.toString(array)); // Imprime el array en cada iteración
para propósitos de demostración

        if (array.length == 0) {
            return false; // Si el array está vacío, el elemento no está presente
        }

        int puntoMedio = array.length / 2;
        System.out.println("//////////////////");
        System.out.println("Punto medio: " + puntoMedio + ": " + array[puntoMedio]);
        System.out.println("//////////////////");

        if (array[puntoMedio] == e) {
```

```
            return true; // Si el elemento en el punto medio es igual a 'e', lo hemos encontrado
        } else if (array[puntoMedio] > e) {
            // Si el elemento en el punto medio es mayor que 'e', buscamos en la mitad izquierda del array
            return complejidad (Arrays.copyOfRange(array, 0, puntoMedio), e);
        } else {
            // Si el elemento en el punto medio es menor que 'e', buscamos en la mitad derecha del array
            return complejidad (Arrays.copyOfRange(array, puntoMedio + 1, array.length), e);
        }
    }
}
```

46 Complejidad temporal: O(2^n) en el peor caso.

Complejidad espacial: O(n) en el peor caso.

47 Complejidad temporal: Tiene un comportamiento similar a O(2^n), ya que el tiempo de ejecución aumenta exponencialmente con el tamaño de la entrada.

48 Complejidad temporal: Tiene un comportamiento similar a O(n^2), ya que el tiempo de ejecución aumenta exponencialmente más rápido que el tamaño de los inputs.

49 Se refiere a la cota superior del crecimiento de una función $T(n)$ en comparación con otra función $f(n)$ multiplicada por una constante k.

50 1 para todo $n \geq 1$, tenemos $T(n) \leq 4 \cdot \log_2(n)$, lo que demuestra que $T(n)$ es de orden $O(\log_2(n))$.

2 aunque $T(n)$ puede estar acotada superiormente por $\log_2(n)$, esto no implica que también esté acotada superiormente por n.

3 no podemos concluir que si $T(n)$ está acotada superiormente por $\log_3(n)$, también lo esté por $\log_2(n)$.