Fragmento 1 — Complejidad O(1)

Pseudocódigo

```
funcion ejemploConstante(n):
  a ← 5
  b ← 8
  c ← a + b
  imprimir(c)
```

Hipótesis inicial

El código no parece depender del tamaño de la entrada n, así que probablemente tenga complejidad constante.

Razonamiento paso a paso

No hay ciclos ni recursión. Cada instrucción se ejecuta una sola vez, independientemente del valor de n. El tiempo de ejecución siempre es el mismo.

Conclusión

La complejidad temporal es O(1).

Fragmento 2 — Complejidad O(n)

Pseudocódigo

```
funcion ejemploLineal(n):
  para i ← 1 hasta n:
    imprimir(i)
```

Hipótesis inicial

El ciclo recorre todos los valores desde 1 hasta n, así que el tiempo de ejecución crece con la entrada.

Razonamiento paso a paso

El ciclo se ejecuta exactamente n veces y dentro solo hay una operación constante. El número total de operaciones es proporcional a n.

Conclusión

La complejidad temporal es O(n).

Fragmento 3 — Complejidad $O(n^2)$

Pseudocódigo

```
funcion ejemploCuadratico(n):
  para i ← 1 hasta n:
    para j ← 1 hasta n:
      imprimir(i, j)
```

Hipótesis inicial

Hay dos ciclos anidados que dependen de n, lo que sugiere una complejidad cuadrática.

Razonamiento paso a paso

El ciclo externo se ejecuta n veces. Por cada iteración del ciclo externo, el ciclo interno también se ejecuta n veces. En total se realizan n × n operaciones.

Conclusión

La complejidad temporal es $O(n^2)$.

Fragmento 4 — Complejidad O(log n)

Pseudocódigo

```
funcion ejemploLogaritmico(n):
  mientras n > 1:
    n ← n / 2
```

Hipótesis inicial

El valor de n se reduce a la mitad en cada iteración, lo que apunta a una complejidad logarítmica.

Razonamiento paso a paso

Cada iteración reduce el tamaño del problema. El número de veces que se puede dividir n entre 2 antes de llegar a 1 es proporcional a $\log_2(n)$.

Conclusión

La complejidad temporal es O(log n).

Fragmento 5 — Caso engañoso (parece $O(n^2)$, pero no lo es)

Pseudocódigo

```
funcion ejemploEnganoso(n):
  para i ← 1 hasta n:
    para j ← 1 hasta 5:
      imprimir(i, j)
```

Hipótesis inicial

A primera vista parece haber dos ciclos anidados, lo que podría sugerir complejidad cuadrática.

Razonamiento paso a paso

El ciclo externo depende de n, pero el ciclo interno tiene un límite constante. Siempre se ejecuta 5 veces, sin importar el tamaño de la entrada. El número total de operaciones es 5n, y las constantes se ignoran en Big-O.

Conclusión

La complejidad temporal es O(n).

Fragmento 6 — Ejemplo complejo (combinación de estructuras)

Pseudocódigo

```
funcion ejemploComplejo(n):
  para i ← 1 hasta n:
    x ← 1
    mientras x < n:
```

```
        x ← x * 2

  para j ← 1 hasta n:
    imprimir(j)
```

Hipótesis inicial

El código tiene varias partes, por lo que es necesario analizar cada bloque por separado.

Razonamiento paso a paso

Primer bloque:
El ciclo externo se ejecuta n veces. Dentro hay un ciclo while que duplica el valor de x en cada iteración, por lo que su complejidad es O(log n). Al estar anidados, este bloque cuesta O(n log n).

Segundo bloque:
Es un ciclo simple que se ejecuta n veces, por lo que cuesta O(n).

Complejidad total:
Al ser bloques secuenciales, se suman las complejidades:
O(n log n) + O(n).
En Big-O se conserva el término dominante.

Conclusión

La complejidad temporal total es O(n log n).

Reflexión final

El fragmento que resultó más difícil de analizar fue el último, porque combina ciclos anidados con bloques secuenciales. Es fácil cometer el error de multiplicar todas las complejidades sin separar correctamente qué partes van una dentro de otra y cuáles se ejecutan una después de la otra.

Los puntos clave que se refuerzan con estos ejemplos son:

- Los ciclos anidados multiplican complejidades.
- Los bloques secuenciales suman complejidades.
- Los límites constantes no afectan el Big-O.
- Cuando el tamaño del problema se reduce por división, aparece una complejidad logarítmica.
- En la notación Big-O siempre domina el término de mayor crecimiento.