

Ing. Edison Meneses Mg.

efmenesest@pucesa.edu.ec

0983506849

Semana 1 - Complejidad Algorítmica

Introducción a la Complejidad Algorítmica

En el mundo de la programación y la informática, no basta con que un algoritmo funcione correctamente; también es fundamental que lo haga de manera eficiente. Aquí es donde entra en juego el concepto de complejidad algorítmica, una herramienta clave para analizar el rendimiento de los algoritmos en función del tiempo que tardan en ejecutarse o del espacio que utilizan en memoria.

La complejidad algorítmica permite estimar cuánto crecerá el tiempo de ejecución o el uso de recursos a medida que aumenta el tamaño de la entrada del problema. Esta estimación se expresa comúnmente mediante la notación Big O (O-grande), que proporciona una forma estándar de describir el comportamiento asintótico del algoritmo, es decir, cómo se comporta cuando el tamaño de los datos tiende a crecer.

Por ejemplo, si un algoritmo tiene una complejidad de $O(n)$, significa que su tiempo de ejecución crecerá linealmente con el tamaño de los datos de entrada. En cambio, un algoritmo con complejidad $O(1)$ se ejecuta en el mismo tiempo, sin importar cuántos datos procese. Otros niveles comunes incluyen $O(n^2)$ (cuadrática), $O(\log n)$ (logarítmica) y $O(n \log n)$ (lineal-logarítmica).

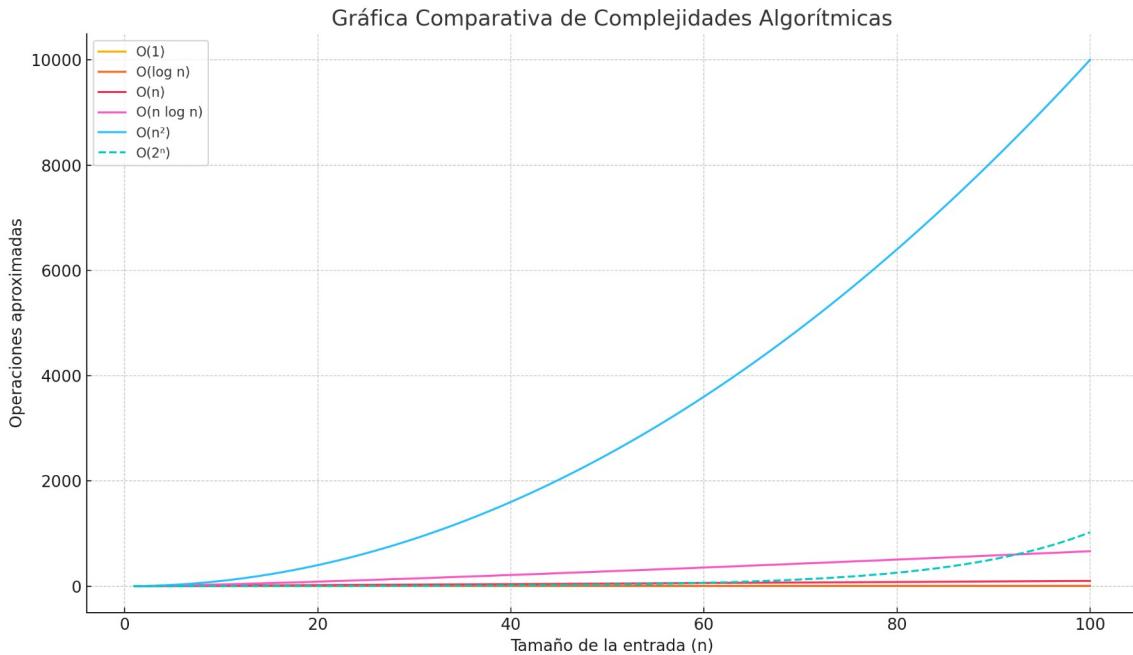
Estudiar la complejidad algorítmica permite a los desarrolladores:

Comparar diferentes soluciones a un mismo problema.

Identificar cuellos de botella en sus programas.

Tomar decisiones informadas sobre qué estructuras y técnicas usar.

En resumen, la complejidad algorítmica no solo nos ayuda a crear algoritmos correctos, sino también a crear algoritmos óptimos y escalables, que puedan ejecutarse eficientemente en el mundo real, donde los datos suelen ser grandes y los recursos limitados.



Aquí tienes la gráfica comparativa de las principales complejidades algorítmicas. Te permite visualizar cómo crecen diferentes funciones en relación al tamaño de entrada n :

- $O(1)$: Constante – no cambia con el tamaño del problema.
- $O(\log n)$: Crece lentamente – ideal para búsquedas rápidas (como búsqueda binaria).
- $O(n)$: Lineal – crece proporcional al tamaño de entrada.
- $O(n \log n)$: Usado en algoritmos eficientes de ordenamiento como Merge Sort.
- $O(n^2)$: Cuadrática – se vuelve lenta con muchos datos (como burbuja).
- $O(2^n)$: Exponencial – se vuelve impracticable rápidamente.

🧠 Ejemplos prácticos de complejidad algorítmica con if, while, y for

1. if – Complejidad $O(1)$ (Constante)

```
In [19]: # Solo una condición se evalúa, sin importar el tamaño de la entrada

def verificar_par(n):
    if n % 2 == 0:
        return "Par"
    else:
        return "Impar"

# Solo se ejecuta una vez
print(verificar_par(10)) # O(1)
```

Par

- Complejidad: O(1) – siempre hace una sola evaluación.
-

2. for – Complejidad O(n) (Lineal)

In [20]: # Recorre una Lista una vez

```
def imprimir_lista(lista):
    for elemento in lista:
        print(elemento)

# Si hay n elementos, hace n impresiones
imprimir_lista([1, 2, 3, 4, 5]) # O(n)
```

1
2
3
4
5

- Complejidad: O(n) – depende del tamaño de la lista.
-

3. while – Complejidad O(n) (Lineal)

In [21]: # Hace una cuenta regresiva hasta Llegar a 0

```
def cuenta_regresiva(n):
    while n > 0:
        print(n)
        n -= 1

cuenta_regresiva(5) # O(n)
```

5
4
3
2
1

- Complejidad: O(n) – número de repeticiones depende del valor inicial.
-

4. for anidado – Complejidad $O(n^2)$ (Cuadrática)

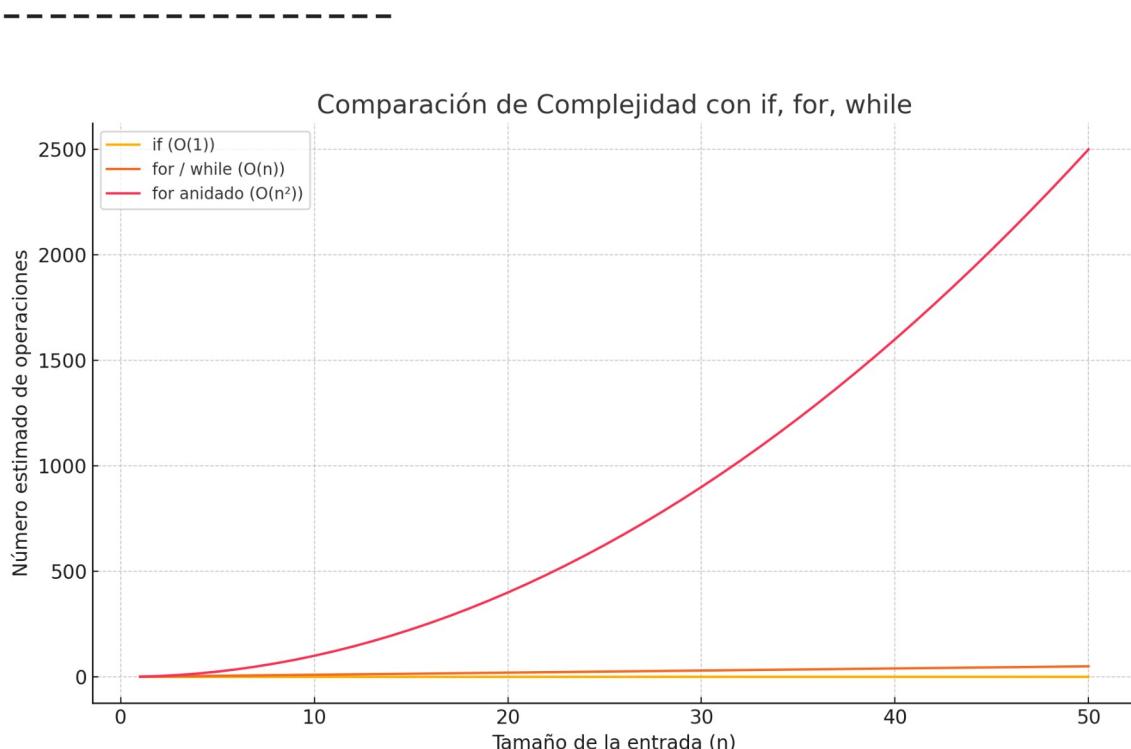
```
In [22]: # Compara cada elemento con los demás

def buscar_duplicados(lista):
    for i in range(len(lista)):
        for j in range(i + 1, len(lista)):
            if lista[i] == lista[j]:
                return True
    return False

buscar_duplicados([1, 2, 3, 4, 1]) # O(n2)
```

Out[22]: True

Complejidad: $O(n^2)$ – dos ciclos anidados aumentan rápidamente el tiempo de ejecución.



Gráfica que muestra cómo varía la complejidad algorítmica dependiendo de la estructura de control:

if: ejecuta una sola operación, sin importar el tamaño de entrada ($O(1)$).
for y while: crecen linealmente con n ($O(n)$).
for anidado: el número de operaciones crece rápidamente al cuadrado ($O(n^2)$).



Explicación de complejidad algorítmica

suma_iterativa(n) recorre todos los números desde 1 hasta n, haciendo una suma en cada paso.

👉 Complejidad: $O(n)$ (lineal), ya que crece proporcional al tamaño de n.

suma_formula(n) utiliza una fórmula matemática conocida.

👉 Complejidad: $O(1)$ (constante), porque solo realiza una operación sin importar el valor de n.

```
In [23]: # Este programa suma los primeros N números naturales
# usando dos métodos diferentes para analizar su complejidad algorítmica.

# Método 1: Suma usando un bucle (complejidad O(n))
def suma_iterativa(n):
    suma = 0
    for i in range(1, n + 1):
        suma += i # Se realiza una suma por cada iteración
    return suma

# Método 2: Suma usando fórmula matemática (complejidad O(1))
def suma_formula(n):
    return n * (n + 1) // 2 # Una sola operación matemática

# Probamos ambos métodos
n = 10
print("Suma iterativa:", suma_iterativa(n)) # Resultado esperado: 55
print("Suma con fórmula:", suma_formula(n)) # Resultado esperado: 55
```

Suma iterativa: 55

Suma con fórmula: 55

Reflexión 1:

¿Cuál método sería mejor si n fuera un número muy grande, como 1,000,000?

 Respuesta: El método de la fórmula (suma_formula) sería mucho mejor en este caso. Esto se debe a que realiza solo una operación matemática, sin importar el tamaño de n . Por lo tanto, incluso si n fuera un millón o un billón, el tiempo de ejecución seguiría siendo el mismo (constante).  Reflexión 2:

¿Por qué crees que la eficiencia importa en problemas reales?

 Respuesta: La eficiencia es crucial en problemas reales porque permite que los programas:

-  Se ejecuten más rápido, especialmente cuando se manejan grandes volúmenes de datos.
-  Usen menos recursos del sistema (memoria, CPU), lo que mejora el rendimiento general.
-  Sean más escalables y puedan ejecutarse en múltiples dispositivos o entornos con diferentes capacidades.

Un algoritmo ineficiente puede hacer que un programa sea inútil en la práctica, incluso si es correcto.

 Ejercicio: ¿Existe un número duplicado en una lista?

Vamos a resolver este problema de dos formas: una con complejidad cuadrática ($O(n^2)$) y otra con complejidad lineal ($O(n)$).

```
In [24]: #  Método 1: Comparación de cada elemento con los demás (fuerza bruta)
# Complejidad: O(n^2)

def tiene_duplicados_bruto(lista):
    for i in range(len(lista)):
        for j in range(i + 1, len(lista)):
            if lista[i] == lista[j]:
                return True # Se encontró un duplicado
    return False # No se encontraron duplicados

#  Método 2: Uso de un conjunto (set) para detectar duplicados
# Complejidad: O(n)

def tiene_duplicados_set(lista):
    elementos_vistos = set()
    for elemento in lista:
        if elemento in elementos_vistos:
            return True # Se encontró un duplicado
        elementos_vistos.add(elemento)
    return False # No hay duplicados
```

💡 Prueba de ambos métodos

```
In [25]: lista_prueba = [3, 7, 1, 9, 3]

print("Método fuerza bruta:", tiene_duplicados_bruto(lista_prueba)) # True
print("Método con set:", tiene_duplicados_set(lista_prueba)) # True
```

Método fuerza bruta: True

Método con set: True

📊 Comparación de complejidades

Método	Complejidad	Descripción
tiene_duplicados_bruto	O(n^2)	Compara todos contra todos
tiene_duplicados_set	O(n)	Usa memoria adicional para hacerlo más rápido

💡 Reflexión 2:

¿Es correcto decir que un algoritmo es "mejor" solo porque usa menos líneas de código?

✓ Respuesta: No necesariamente. Un algoritmo con pocas líneas puede ser menos eficiente si tiene una alta complejidad. Lo importante es analizar qué tan bien se comporta con grandes cantidades de datos, no solo si es corto o "bonito". La claridad y la eficiencia deben ir de la mano. 💬 Reflexión 3:

¿Cuándo sería útil sacrificar tiempo por memoria, o viceversa?

✓ Respuesta:

Si estamos en un sistema con poca memoria (como un dispositivo embebido), puede ser necesario usar un algoritmo más lento pero que consuma menos memoria.

Si el tiempo de respuesta es crítico (por ejemplo, en sistemas en tiempo real), conviene usar más memoria para obtener resultados más rápidos.

⌚ Ejemplo concreto: El método con set usa más memoria (porque guarda los elementos vistos), pero es más rápido. El método de fuerza bruta usa menos memoria, pero es mucho más lento en listas grandes.