Complejidad y Eficiencia en Algoritmos

Explicación detallada con ejemplos en Python

Complejidad de los algoritmos

La complejidad de un algoritmo se refiere a la cantidad de recursos que dicho algoritmo necesita para ejecutarse. Los recursos generalmente se miden en tiempo (complejidad temporal) y memoria utilizada (complejidad espacial).

Introducción a la complejidad computacional

La complejidad computacional estudia los recursos necesarios para resolver problemas con algoritmos. Se divide en:

- Complejidad temporal: tiempo necesario para ejecutar el algoritmo.
- Complejidad espacial: memoria necesaria para ejecutar el algoritmo.

Complejidad Temporal

```
import time

def suma_n_numeros(n):
    suma = 0
    for i in range(1, n + 1):
        suma += i
    return suma

n = 100000

start_time = time.time()

suma_n_numeros(n)

end_time = time.time()

print(f"Tiempo de ejecución: {end_time - start_time} segundos")
```

Este ejemplo muestra cómo medir el tiempo de ejecución de un algoritmo que suma los primeros n números. La complejidad temporal de este algoritmo es O(n), ya que el tiempo de ejecución crece linealmente con n.

Complejidad Espacial

```
def crear_lista(n):
    lista = []
    for i in range(n):
        lista.append(i)
    return lista
n = 100000
lista = crear_lista(n)
print(f"Memoria utilizada: {lista.__sizeof__()} bytes")
```

La complejidad espacial de este algoritmo es O(n), ya que el uso de memoria crece linealmente con n.

Eficiencia y recursos

La eficiencia de un algoritmo se mide por la cantidad de recursos que consume. Los principales recursos como dijimos anteriormente son tiempo y memoria. Un algoritmo eficiente es aquel que consume menos recursos.

Comparación de eficiencia

```
def suma_directa(n):
return n * (n + 1) // 2
n = 100000
start_time = time.time()
suma_directa(n)
end_time = time.time()
print(f"Tiempo de ejecución: {end_time - start_time} segundos")
```

Este algoritmo tiene una complejidad temporal O(1), ya que el tiempo de ejecución es constante sin importar el valor de n, lo que lo hace más eficiente que el algoritmo de suma n números.

Orden de complejidad de los algoritmos

El orden de complejidad se clasifica comúnmente en notación Big O (O grande), que describe el comportamiento asintótico del algoritmo:

- O(1): constante.
- O(log n): logarítmica.
- O(n): lineal.
- O(n log n): linealítmica.
- O(n^2): cuadrática.
- O(2^n): exponencial.

La regla de la suma y de la multiplicación

La regla de la suma y la multiplicación nos ayuda a calcular la complejidad de los algoritmos.

- Regla de la suma: Si un algoritmo consiste en dos partes consecutivas, la complejidad total es la suma de las complejidades de ambas partes.
- Regla de la multiplicación: Si un algoritmo consiste en dos partes anidadas, la complejidad total es el producto de las complejidades de ambas partes.

Regla de la Suma

```
def regla_suma(n):
# Parte 1: O(n)
for i in range(n):
    print(i)
# Parte 2: O(n^2)
    for i in range(n):
        for j in range(n):
```

print(i, j)

 # Complejidad total: O(n + n^2) = O(n^2)

regla_suma(10)

Regla de la Multiplicación

```
def regla_multiplicacion(n):
# Parte 1: O(n)
for i in range(n):
print(i)
# Parte 2: O(n)
for i in range(n):
print(i)
# Complejidad total: O(n * n) = O(n^2)
regla_multiplicacion(10)
```

Orden de Complejidad

```
def algoritmo_constante():
return 42  # O(1)
def algoritmo_lineal(n):
for i in range(n):
print(i)  # O(n)
def algoritmo_cuadratico(n):
for i in range(n):
for j in range(n):
print(i, j)  # O(n^2)
# Ejecución de ejemplos
```

```
n = 10

algoritmo_constante()

algoritmo_lineal(n)

algoritmo_cuadratico(n)
```

Algoritmos básicos de ordenamiento y búsqueda.

Dos algoritmos comunes son la búsqueda binaria y el ordenamiento de burbuja.

- Búsqueda binaria: O(log n).
- Ordenamiento de burbuja: $O(n^2)$.

Búsqueda Binaria

```
def busqueda_binaria(lista, objetivo):
        inicio = 0
        fin = len(lista) - 1
        while inicio <= fin:
                medio = (inicio + fin) // 2
                if lista[medio] == objetivo:
                        return medio
                elif lista[medio] < objetivo:
                        inicio = medio + 1
                else:
                        fin = medio - 1
        return -1


lista = [1, 3, 5, 7, 9]
```

```
objetivo = 7

print(busqueda_binaria(lista, objetivo))  # Devuelve el índice 3
```

Ordenamiento de Burbuja

```
def ordenamiento_burbuja(lista):
    for i in range(len(lista) - 1):
        for j in range(len(lista) - 1 - i):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista


lista = [5, 3, 8, 4, 2]

print(ordenamiento_burbuja(lista))  # Devuelve [2, 3, 4, 5, 8]
```