Análisis Teórico y Notación Big-O

1. Introducción al Análisis Teórico

El **análisis teórico** es un enfoque matemático para evaluar la eficiencia de un algoritmo sin necesidad de implementarlo. Este método se basa en el **pseudocódigo** (usaremos Python como lenguaje) del algoritmo y permite calcular una **función temporal T(n)**, que representa el número de operaciones que realiza el algoritmo para una entrada de tamaño **n**.

Ventajas del análisis teórico:

- No depende del hardware o software específico.
- Permite considerar todas las posibles entradas, no solo un conjunto limitado.
- Es más general y abstracto que el análisis empírico.

2. Cálculo de la Función Temporal T(n)

Operaciones primitivas:

- Asignaciones, accesos a arrays, evaluaciones de expresiones, etc.
- Cada operación primitiva se cuenta como 1.

Ejemplos:

- Asignar valor a una variable: **x=2**
- Indexar un elemento en un array: **vector[3]**
- Devolver un valor en una función: **return x**
- Evaluar una expresión aritmética: **x+3**
- Evaluar una expresión lógica: **0<i<index**

Operación	#operaciones primitivas
"x = 2"	1
"x = y + 3"	2 (1 suma + 1 asignación)
"return a[0] + 1"	3 (1 acceso + 1 suma + 1 return)
"return node is None or node.elem >5"	5 (1 node is None + 1 obtener node.elem + 1 node.elem >5 + 1 or + 1 return)
"print(“Hola”)"	1

Estructuras de control:

- **Secuencia:** Si nuestro algoritmo se compone de varios bloques, su función T(n), será la suma de las funciones T(n) de cada bloque.

B1
B2
..
Bn

T(n)=T(B1)+T(B2)+...+T(Bn)

- **Condicionales (if-else):** Sólo uno de los bloques (B1 , B2 , ... Bk) se ejecutará.

```
if condition1:
    B1
elif condition2:
    B2 ...
else:
    Bk
```

Tif-else (n)= max(TB1 (n), TB2 (n), ..., TBk (n))

- **Bucles:** La función T(n) para un bucle se calculará como el número de veces que se ejecuta el bucle (número de iteraciones) por la función T(n) del bloque interno B.

```
while condition:
    B
for x in ... :
    B
```

T loop (n) = T(B) * número de iteraciones

- **Bucles anidados:** Producto del número de iteraciones de cada bucle.

```
for i in range(n):

    for i in range(n):

        print(i*j)          #T(n) = 2
```

Su función temporal T(n) es: **$T(n) = n * n * 2 = 2n^2$**

Es decir, 2 del bucle interno, por el número de iteraciones del bucle interno, por el número de iteraciones del bucle externo.

Ejemplo 1: Suma de los primeros n números

```
def sumar_numeros(n):
    resultado = 0           # 1 operación
    for i in range(1, n+1): # n iteraciones
        resultado += i      # 2 operaciones (suma y asignación)
    return resultado        # 1 operación
```

$T(n) = 1 + n * 2 + 1 = 2n + 2$.

Ejemplo 2: Bucles anidados

```
for i in range(n):          # n iteraciones
        for j in range(n):      # n iteraciones
            print(i * j)        # 2 operación
```

- **$T(n) = n * n * 2 = 2n^2$**.

3. Introducción a la Notación Big-O

La **notación Big-O** es una forma de describir el comportamiento asintótico de una función, es decir, cómo crece cuando el tamaño de la entrada tiende a infinito. Se utiliza para simplificar la comparación de algoritmos eliminando constantes y términos de menor orden.

Pasos para calcular Big-O:

1. Identificar el término de mayor crecimiento en T(n).

2. Eliminar constantes y coeficientes.

Ejemplos:

- $T(n) = 3n^2 + 2n + 1 \rightarrow O(n^2)$.
- $T(n) = 5n + 10 \rightarrow O(n)$.
- $T(n) = 7 \rightarrow O(1)$.

4. Órdenes de Complejidad Comunes

Notación Big-O	Nombre	Descripción
O(1)	Constante	El tiempo no depende de la entrada.
O(log n)	Logarítmica	Típico en algoritmos de búsqueda binaria.
O(n)	Lineal	Tiempo proporcional al tamaño de la entrada.
O(n log n)	Lineal-logarítmica	Típico en algoritmos de ordenación eficientes.
$O(n^2)$	Cuadrática	Típico en bucles anidados.
O(2^n)	Exponencial	Típico en algoritmos de fuerza bruta.
O(n!)	Factorial	Típico en problemas de permutaciones.

5. Ejemplos Prácticos de Big-O

Ejemplo 1: Suma de los primeros n números (bucle)

```
def sumar_numeros(n):

    resultado = 0           # 1 operación
    for i in range(1, n+1): # n iteraciones
        resultado += i      # 2 operaciones (suma y asignación)
    return resultado        # 1 operación
```

- $T(n) = 2n + 2 \rightarrow O(n)$.

Ejemplo 2: Suma de Gauss

```
def sumGausN(n):

    resultado=n*(n+1)/2         # 3 operación

    return resultado            # 1 operación
```

- T(n) = 4 → O(1).

Ejemplo 3: Búsqueda binaria

```
def busqueda_binaria(lista, objetivo):
    """
    Realiza una búsqueda binaria en una lista ordenada.

    Parámetros:
    lista (list): Lista ordenada en la que se busca el elemento.
    objetivo: Elemento a buscar en la lista.

    Retorna:
    int: Índice del elemento si se encuentra, -1 si no está presente.
    """
    izquierda, derecha = 0, len(lista) - 1  # O(1), inicialización de
variables

    while izquierda <= derecha:  # O(log n), ya que reducimos el espacio
de búsqueda a la mitad
        medio = (izquierda + derecha) // 2  # O(1), cálculo del índice
medio
        if lista[medio] == objetivo:  # O(1), comparación
            return medio  # O(1), retorno del índice si se encuentra
        elif lista[medio] < objetivo:  # O(1), comparación
            izquierda = medio + 1  # O(1), actualización del límite
inferior
        else:
            derecha = medio - 1  # O(1), actualización del límite
superior

    return -1  # O(1), retorno si el elemento no está presente
```

- T(n) = log n → O(log n).

Ejemplo 4: Ordenación por burbuja

```
def ordenamiento_burbuja(lista):
    """
    Implementa el algoritmo de ordenamiento burbuja.
```

```
    Parámetros:
    lista (list): Lista de elementos a ordenar.

    Retorna:
    list: Lista ordenada de menor a mayor.
    """
    n = len(lista)  # O(1), obtener la longitud de la lista

    for i in range(n - 1):  # O(n), primer bucle
        for j in range(n - 1 - i):  # O(n), segundo bucle (cada vez itera
menos)
            if lista[j] > lista[j + 1]:  # O(1), comparación
                lista[j], lista[j + 1] = lista[j + 1], lista[j]  # O(1),
intercambio

    return lista  # O(1), retorno de la lista ordenada
```

- $T(n) = n^2 \rightarrow O(n^2)$.

6. Conclusión

- El análisis teórico nos permite calcular la función temporal T(n) de un algoritmo.
- La notación Big-O simplifica la comparación de algoritmos al enfocarse en el término de mayor crecimiento.
- Es una herramienta esencial para elegir el algoritmo más eficiente en función del tamaño de la entrada.