Taller de Complejidad Algorítmica (Big O)

Nombre: Juan Andres Salazar Urdinola
Código: 2262431-2724

Objetivo: Demostrar el crecimiento asintótico de funciones mediante límites y analizar/escribir algoritmos eficientes en Python.

Módulo 1: El Rigor Matemático (Criterio del Límite)

Para cada ejercicio, utiliza la fórmula:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Si L=0, entonces f(n) = O(g(n)). Si L=c, entonces f(n) = Θ(g(n)).

Ejercicio 1.1 (Nivel Básico):

Demuestre que $F(n) = 7n^2 + 5n + 2$ es de orden $O(n^2)$

$$\lim_{n \to \infty} \frac{F(n)}{g(n)} = \frac{(7n^2 + 5n + 2)}{n^2} = 7 + \frac{5}{n} + \frac{2}{n^2}$$

$$\lim_{n \to \infty} = \frac{5}{n} = 0 \qquad \lim_{n \to \infty} = \frac{2}{n^2} = 0 = \text{Lim} = 7$$

Como el limite es un numero Finito

$\text{Lim} = 7$ (constante $\neq 0$) $= F(n) = O(n^2)$, por lo tanto tambien $O(n^2)$

Ejercicio 1.2 (Nivel Intermedio):

Demuestra que la función lineal g(n) = n es una cota superior para la función logarítmica

Demuestre que la funcion lineal $g(n) = n$ es una cota superior para la funcion logaritmica

$f(n) = Ln(n)$ es decir, demuestra que $ln(n) = O(n)$

$Ln(n) = O(n)$ $\quad \lim_{n \to \infty} \frac{Ln(n)}{n}$

$\frac{d}{dn}(Ln\, n) = \frac{1}{n} = \frac{d}{dn}(n) = 1$

$\lim_{n \to \infty} = \frac{1}{n} = 0$ Como el limite es 0

el logaritmo crece mas lento que la funcion lineal

Ejercicio 1.3 (Nivel Avanzado):

Determina la relacion asintótica entre $f(n) = n!$ (factorial) y $g(n) = 2^n$ ¿Cual crece mas rapido?

$f(n) = n!$ $\qquad g(n) = 2^n$

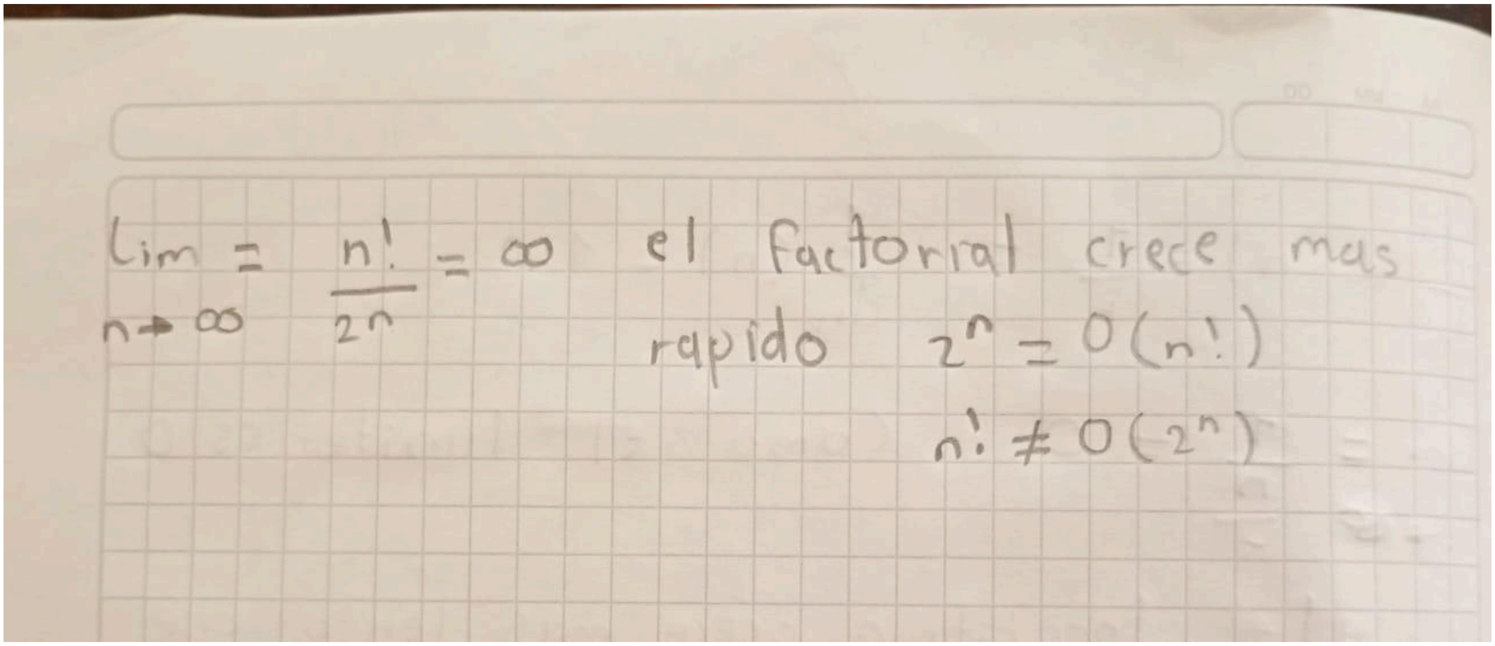
para $n > 2$

$$\lim_{n \to \infty} = \frac{n!}{2^n} = \frac{1 \cdot 2 \cdot 3 \cdot 4 \; 5}{2 \; 2 \; 2 \; 2 \; 2} = \frac{3}{2} > 1$$

$$\frac{4}{2} > 1$$

$$\frac{5}{2} > 1$$

Se puede Observar que mediante numeros mayores que 1 por ende crece sin limite



Módulo 2: Análisis de Algoritmos (Lectura de Código)

Analiza los siguientes fragmentos de código en Python. Para cada uno, identifica: Mejor Caso, Peor Caso y la Notación Big O.

Ejercicio 2.1: El buscador de pares

```python
def buscar_par_especifico(lista, objetivo):
    # Analiza este código
    pasos = 0
    for i in range(len(lista)):
        pasos += 1
        if lista[i] % 2 == 0 and lista[i] == objetivo:
            return True, pasos
    return False, pasos
```

Mejor Caso O(1): El primer elemento de la lista es par e igual al objetivo. El bucle se ejecuta una sola vez y retorna inmediatamente.

Peor Caso O(n): El objetivo no existe en la lista o está en la última posición. Se recorren todos los n elementos.

Big O General: O(n) — El algoritmo realiza una sola pasada lineal sobre la lista. El número de operaciones es proporcional al tamaño de la entrada.

Ejercicio 2.2: El salto de índices

```python
def algoritmo_misterioso(n):
    # Analiza el crecimiento de 'i'
    i = 1
    operaciones = 0
    while i < n:
        operaciones += 1
        i = i * 2
    return operaciones
```

La clave está en cómo crece i . La secuencia de valores es: 1, 2, 4, 8, 16, ..., 2^k

El bucle termina cuando 2^k ≥ n, es decir, cuando k ≥ log 2(n).

Mejor Caso O(1): Si n ≤ 1, el while nunca se ejecuta (0 iteraciones).

Peor Caso O(log n): El número de iteraciones es exactamente $\lfloor \log_2(n) \rfloor$.

Big O General: O(log n) — Cada iteración duplica el valor de i, reduciendo a la mitad el espacio restante. Este patrón de crecimiento geométrico produce complejidad logarítmica

Módulo 3: Implementación (Programación)

En esta parte debes escribir código que cumpla con las restricciones de eficiencia solicitadas.

Ejercicio 3.1: Intersección de Listas

Escribe una función en Python llamada interseccion(lista1, lista2) que devuelva los elementos comunes.

```python
ef interseccion(lista1, lista2):

    return list(set(lista1) & set(lista2))



# Prueba

lista1 = [1, 2, 3, 4, 5]

lista2 = [3, 4, 5, 6, 7]


print(interseccion(lista1, lista2))
```

Salida

```
PS C:\Users\juan_\OneDrive\Desktop\Taller_Complejidad_Algoritmica>
& C:/Users/juan_/AppData/Local/Python/bin/python.exe "c:/Users/juan_/OneDrive/Desktop/Taller_Complejidad_Algoritmica/Ejercicios 3.py"
[3, 4, 5]
```

Reto A: Impleméntalo de forma "ingenua" usando bucles anidados (O(n^2)).

```python
def interseccion(lista1, lista2):
    resultado = []

    for elemento1 in lista1:                        # Recorre lista1 → O(n)
        for elemento2 in lista2:                    # Por cada elemento, recorre lista2 → O(m)
            if elemento1 == elemento2:              # Compara uno a uno
                if elemento1 not in resultado:      # Evita duplicados
                    resultado.append(elemento1)

    return resultado
lista1 = [1, 2, 3, 9, 5]
lista2 = [3, 4, 8, 6, 7]
print(interseccion(lista1, lista2))
```

Salida

```
PS C:\Users\juan_\OneDrive\Desktop\Taller_Complejidad_Algoritmica>
& C:/Users/juan_/AppData/Local/Python/bin/python.exe "c:/Users/juan_/OneDrive/Desktop/Taller_Complejidad_Algoritmica/Ejercicios 3.py"
[3]
```

Reto B: Investiga cómo hacerlo usando un set (conjunto) en Python para lograr una complejidad de O(n).

```python
def interseccion_set(lista1, lista2):
    set1 = set(lista1)    # Convierte lista1 a conjunto → O(n)
    set2 = set(lista2)    # Convierte lista2 a conjunto → O(m)
    return list(set1 & set2)
# Prueba
lista1 = [1, 2, 4, 9, 5]
lista2 = [1, 4, 8, 6, 7]
print(interseccion_set(lista1, lista2))
```

Salida

```
PS C:\Users\juan_\OneDrive\Desktop\Taller_Complejidad_Algoritmica>
& C:/Users/juan_/AppData/Local/Python/bin/python.exe "c:/Users/juan_/OneDrive/Desktop/Taller_Complejidad_Algoritmica/Ejercicios 3.py"
[1, 4]
```

Documentación: Explica por qué la versión con set es más rápida basándote en lo visto en el Módulo 1.

R/Un set en Python implementa internamente una tabla hash. Esto garantiza que la operación "in" tenga complejidad $O(1)$ promedio en lugar de $O(n)$ como en una lista.

Aplicando el criterio del límite del Módulo 1:

$\lim_{n\to\infty} n / n^2 = \lim_{n\to\infty} 1/n = 0$

Como $L = 0$, se confirma que $O(n) \subset O(n^2)$ (la solución con set es asintóticamente inferior y mucho más eficiente para entradas grandes).

Ejercicio 3.2: El desafío del crecimiento exponencial

Escribe una función recursiva para calcular el número de Fibonacci: $F(n) = F(n-1) + F(n-2)$.

```python
def fibonacci(n):
    # Casos base
    if n < 0:
        raise ValueError("El número debe ser mayor o igual a 0")
    if n == 0:
        return 0
    if n == 1:
        return 1

    # Caso recursivo: F(n) = F(n-1) + F(n-2)
    return fibonacci(n - 1) + fibonacci(n - 2)


for i in range(10):
    print(f"F({i}) = {fibonacci(i)}")
```

Salida

```
PS C:\Users\juan_\OneDrive\Desktop\Taller_Complejidad_Algoritmica> & C:/Users/juan_/AppData/Local/Python/bin/python.exe c:/Users/juan_/OneDrive/Desktop/Taller_Complejidad_Algoritmica/1.py
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
```

Analiza por qué esta implementación simple es O(2^n).

R/Cada llamada a fibonacci(n) genera exactamente 2 llamadas recursivas. El árbol de recursión tiene profundidad n y se expande geométricamente:

T(n) = T(n-1) + T(n-2) + O(1) → T(n) ≈ 2^n

Además, el algoritmo recalcula los mismos subproblemas múltiples veces. Por ejemplo, F(3) se calcula en varias ramas del árbol.

Pregunta técnica: Si n=50, ¿por qué tu computadora probablemente se bloquee al intentar calcularlo?

R/Con n = 50, el número de llamadas recursivas es aproximadamente:

2^50 ≈ 1,125,899,906,842,624 (≈ 1 cuatrillón de operaciones)

Incluso a 10^9 operaciones/segundo (procesador moderno), esto tomaría ~13 días. La memoria de la pila de llamadas (stack) también se agota por la profundidad de recursión.