📘 Clase 1 – Introducción y Fundamentos

🎯 Objetivos de la clase

- Conocer la **estructura del curso**.
- Entender la **complejidad computacional** y la notación **Big-O** (tiempo de ejecución).
- Entender la **complejidad espacial** (uso de memoria).
- Estudiar **entrada y salida básica**.

⏱️ Complejidad computacional (Notación Big-O)

La **complejidad computacional** mide cuántas operaciones realiza un algoritmo en función del tamaño de la entrada `n`.
Se utiliza la **notación Big-O** para describir el **peor caso**, es decir, el máximo número de pasos que puede tomar un algoritmo.

Complejidades comunes

Algoritmo / Operación	**Big-O**	**Descripción breve**	**Valores de `n` aproximados**
Acceso a un elemento	O(1)	Acceder a un índice de un arreglo o lista	Hasta 10^8
Búsqueda lineal	O(n)	Recorre todos los elementos de una lista	Hasta 10^7
Búsqueda binaria	O(log n)	Divide el problema por la mitad cada vez	Hasta 10^{18} o más
Ordenamiento rápido (Quicksort)	O(n log n)	Algoritmo eficiente de ordenación	Hasta 10^6
Fuerza bruta / pares	$O(n^2)$	Compara todos los pares posibles	Hasta 10^3 – 10^4
Exponencial simple	$O(2^n)$	Algoritmos que generan todas las combinaciones de elementos	Hasta 20
Fuerza bruta permutaciones	O(n!)	Generar todas las permutaciones de `n` elementos	Hasta 10

Cómo calcular la complejidad de un algoritmo

Observa el bloque de código que más se repite o hace más trabajo. Ese bloque domina la complejidad.

1. **Operaciones con coste O(1)**

   Algunas **operaciones básicas tienen coste O(1)** (tiempo constante), es decir, no dependen de `n`:

   - Asignaciones simples: `x = 5`
   - Sumas, restas, multiplicaciones y divisiones simples: `x + y`

- Comparaciones: `x > y`
- Acceso a un elemento de una lista: `arr[i]`
- Llamadas a funciones que son O(1) por definición

  Ejemplo:

  ```
  x = 5          # O(1)
  y = x + 2      # O(1)
  print(y)       # O(1)
  ```

2. **Sumar bucles anidados**

   - Cada bucle anidado multiplica las operaciones:

   ```
   for i in range(n):       # O(n)
       for j in range(n):   # O(n)
           print(i, j)      # O(1)
   # Complejidad total: O(n²)
   ```

3. **Ignorar constantes y términos menores**

   - Al calcular la complejidad, **no importa multiplicar por constantes ni sumar términos menores**.
   - Ejemplos:
     - $O(3n + 5) \rightarrow O(n)$
     - $O(n^2 + n) \rightarrow O(n^2)$

4. **Calcular complejidades de recursividad** 🦉

   - No entraremos en detalle en este curso, pero en recursión se usan **ecuaciones de recurrencia** para estimar el número de operaciones.
   - La idea es ver **cuántas veces se llama la función** y cuánto trabajo hace cada llamada.
   - Ejemplo típico: mergesort → $O(n \log n)$, Fibonacci recursivo simple → $O(2^n)$

Ejemplo práctico: combinación de bucles y condiciones

```
def ejemplo_complejidad(arr):
    total = 0
    # Ciclo simple con varios if
    for x in arr:           # Se ejecuta n veces → O(n)
        if x % 2 == 0:
            total += x
        if x % 3 == 0:
            total += 2*x
        if x % 5 == 0:
            total += 3*x
```

```
    # Dos ciclos for anidados
    n = len(arr)
    for i in range(n):    # O(n)
        for j in range(n):# O(n)
            total += i*j  # O(1)
    return total

# Complejidad total:
# Primer for: O(n)
# Segundo for anidado: O(n²)
# Total: O(n + n²) → O(n²)
```

💾 Complejidad espacial

La **complejidad espacial** mide **cuánta memoria utiliza un algoritmo** en función del tamaño de la entrada `n`. Es tan importante como la complejidad temporal, sobre todo en problemas donde la memoria es limitada.

Factores principales que afectan la complejidad espacial

- Variables simples (enteros, floats, booleanos) → O(1)
- Arreglos o listas de tamaño `n` → O(n)
- Matrices de tamaño n × m → O(n*m)
- Estructuras de datos adicionales (pilas, colas, diccionarios) → depende del número de elementos
- Funciones recursivas → la pila de llamadas también consume memoria

> **Nota:** La complejidad espacial **no siempre coincide con la temporal**. Por ejemplo, algunos algoritmos rápidos usan más memoria para almacenar estructuras auxiliares.

Ejemplo: calcular memoria de una matriz n x n (teórico)

Supongamos que queremos crear una **matriz de enteros** de tamaño `n × n` y que cada entero ocupa **32 bytes**.

```
n = 1000            # tamaño de la matriz
matriz = [[0 for _ in range(n)] for _ in range(n)] # Crear matriz
```

$$ \text{Memoria total (bytes)} = n \times n \times \text{tamañoEntero} $$

- `n × n` → número total de elementos
- `tamaño_entero` → memoria de cada entero (32 bytes aproximadamente)

2. Convertir a megabytes (MB)

$$ \text{Memoria (MB)} = \frac{\text{Memoria total (bytes)}}{1024 \times 1024} $$

- 1 KB = 1024 bytes
- 1 MB = 1024 KB = 1024 × 1024 bytes

3. Ejemplo con n = 1000

- Número de elementos: $1000 \times 1000 = 1,000,000$
- Memoria en bytes: $1,000,000 \times 32 = 32,000,000$ bytes
- Memoria en MB: $32,000,000 \div (1024 \times 1024) \approx 30.52$ MB

> **Nota:** Este cálculo es teórico. En Python, los enteros son objetos y usan más memoria debido al overhead.

🖥️ Entrada y salida en Python

1. Leer un solo número

```python
# Leer un entero
x = int(input())
print("Número leído:", x)
```

2. Leer una lista de números en una sola línea separada por espacios

```python
# Entrada: "1 2 3 4 5"
arr = list(map(int, input().split()))
print("Lista leída:", arr)
```

3. Leer una matriz `n x m`

```python
n = int(input()) # Número de filas
m = int(input()) # Número de columnas:

matriz = []
for _ in range(n):
    fila = list(map(int, input().split()))
    matriz.append(fila)

print("Matriz leída:")
for fila in matriz:
    print(fila)
```

4. Leer `n` números cuando `n` está dado

```python
n = int(input("Número de elementos: "))
numeros = []
for _ in range(n):
    numeros.append(int(input()))

print("Números leídos:", numeros)
```

5. Leer hasta el final de la entrada (desconocido)

```
import sys

numeros = []
for linea in sys.stdin:
    print("Leí:", line.strip())
```

📝 Problemas: Clase 1 – Introducción y Fundamentos