

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 $\rightarrow O(n \log n)$, Fibonacci recursivo simple $\rightarrow O(2^n)$

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 $n \times m$ → $O(n*m)$
- 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 $n \times n$ (teórico)

Supongamos que queremos crear una **matriz de enteros** de tamaño **$n \times n$** 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
```

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

- $n \times n$ → 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

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

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

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

3. Leer una matriz $n \times m$

```
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

```
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

Soluciones a los problemas

Hola Algorithmic

```
# imprime el saludo en la consola
print("Hola, Algorithmic!")
```

¿Número Par o Impar?

```
# leemos un número y lo convertimos a entero
a = int(input())
# si el número es impar
if a % 2 == 1:
    print("Odd") # mostramos "Odd"
else:
    print("Even") # si no, es par y mostramos "Even"
```

Máximo de Dos Enteros

```
# leemos varios números separados por espacios, los convertimos a enteros y
mostramos el mayor
print(max(map(int, input().split())))
```

Contar Cuadrados Perfectos en una Lista

```
# leemos una lista de números enteros separados por espacios
arr = list(map(int, input().split()))
count = 0 # contador de números que son cuadrados perfectos
for num in arr:
    if num >= 0:
        # calculamos la raíz entera y comprobamos si es cuadrado perfecto
        raiz = int(num**(1/2))
        if raiz * raiz == num: # Comprobamos si es un cuadrado perfecto
```

```

        count += 1
# mostramos la cantidad de cuadrados perfectos
print(count)

```

Otra solución utilizando compresión de listas

```

# leemos números separados por espacios y contamos cuántos son cuadrados perfectos
print(sum(
    1 for x in input().split()
    if int(int(x)**(1/2)) * int(int(x)**(1/2)) == int(x) # comprobamos si x es
cuadrado perfecto
))

```

El mensaje secreto

```

def descifrar_mensaje(linea):
    resultado = []
    vocales = "aeiouAEIOU"
    for c in linea:
        if c in vocales:
            continue # saltamos las vocales
        elif c.isalpha():
            resultado.append(c.upper()) # convertimos consonantes a mayúscula
        elif c == ' ':
            resultado.append('_') # reemplazamos espacios por guiones bajos
        else:
            resultado.append(c) # otros caracteres los dejamos igual
    return ''.join(resultado)
# leemos hasta que se ingrese '#'
while True:
    linea = input() # leemos la línea de entrada
    if linea == '#': # si es '#' terminamos el ciclo
        break
    print(descifrar_mensaje(linea))

```

Número Más Frecuente en una Lista

```

# leemos una lista de números separados por espacios
a = list(map(int, input().split()))
d = {} # diccionario para contar cuántas veces aparece cada número
ma = -1 # mayor cantidad de repeticiones encontrada
sol = 0 # número con mayor frecuencia (y menor en caso de empate)
for x in a:
    if x in d:
        d[x] += 1
    else:

```

```

    d[x] = 1
# actualizamos la solución si encontramos más repeticiones
if ma < d[x]:
    ma = d[x]
    sol = x
# si hay empate en repeticiones, elegimos el menor número
elif ma == d[x] and x < sol:
    sol = x
print(sol) # mostramos el número más frecuente

```

Suma de Todos los Elementos en una Matriz

```

# leemos la cantidad de filas y columnas
n = int(input())
m = int(input())
su = 0 # acumulador de la suma
for i in range(n):
    # leemos una fila de números y sumamos sus valores
    su += sum(map(int, input().split()))
print(su) # mostramos la suma total

```

Batalla de Soldados de Juguete

Explicación de la solución

- Ordenamos tus soldados (**a**) y los del otro niño (**b**) de menor a mayor.
- Usamos un puntero **p** para recorrer los soldados del otro niño.
- Para cada uno de tus soldados (en orden creciente):
 - Si tu soldado es **más fuerte** que el soldado actual del otro niño (**x > b[p]**), ganamos la batalla.
 - Avanzamos **p** al siguiente soldado del otro niño para la próxima comparación.
- Esta estrategia asegura **maximizar el número de victorias** porque siempre usamos el soldado más débil posible que aún pueda ganar.
- Finalmente, **sol** contendrá el **número máximo de victorias** que puedes lograr.

```

# leemos la cantidad de soldados
n = int(input())
# leemos las fuerzas de tus soldados y del otro niño
a = list(map(int, input().split()))
b = list(map(int, input().split()))
# ordenamos ambos conjuntos de soldados de menor a mayor
a.sort()
b.sort()
p = 0 # puntero para recorrer los soldados del otro niño
sol = 0 # contador de victorias
# recorremos tus soldados
for x in a:
    if x > b[p]: # si tu soldado es más fuerte que el actual del otro niño
        sol += 1 # ganamos la batalla

```

```
p += 1    # pasamos al siguiente soldado del otro niño
print(sol) # mostramos el número máximo de victorias
```

Columna Máxima por Fila

Idea de la solución

- Para **maximizar la suma**, en cada fila basta con elegir el **elemento más grande**.
- Recorremos cada fila, encontramos su máximo y lo sumamos al total.
- Al final, **sol** contendrá la **suma máxima posible** siguiendo la regla de elegir un elemento por fila.

```
# leemos la cantidad de filas (n) y columnas (m)
n, m = map(int, input().split())
# leemos la matriz fila por fila
a = []
for i in range(n):
    a.append(list(map(int, input().split())))
sol = 0 # Acumulador de la suma máxima
# recorremos cada fila
for i in range(n):
    ma = 0
    # buscamos el elemento más grande de la fila
    for j in range(m):
        ma = max(ma, a[i][j])
    sol += ma # sumamos el máximo de la fila al total
print(sol) # mostramos la suma máxima
```

Bingo Musical

Idea de la solución

- Usamos **conjuntos (set)** para manejar los números de cada jugador.
- **sol1** guarda los números que están en **todas las listas** mediante la **intersección (&=)**.
- **sol2** guarda los números que están en **al menos una lista** mediante la **unión (|=)**.
- Al final imprimimos ambos conjuntos convertidos a cadenas.

```
# leemos el número de jugadores
n = int(input())
# inicializamos los sets
sol1 = set([i for i in range(1, 1001)]) # para intersección: todos los números posibles
sol2 = set() # para unión: empieza vacío
# procesamos cada jugador
for i in range(n):
    numeros = list(map(int, input().split()))[1:] # ignoramos el primer número (cantidad)
    set_numeros = set(numeros) # convertimos la lista en conjunto
```



```
sol1 &= set_numeros # intersección: números que están en todas las listas
hasta ahora
sol2 |= set_numeros # unión: números que aparecen en al menos una lista
# mostramos el resultado
print(" ".join(map(str, sol1))) # números que aparecen en todas las listas
print(" ".join(map(str, sol2))) # números que aparecen en al menos una lista
```