#CátedrasCiber

Módulo II: Estructuras y complejidad

26/02/2025

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

Entidad colaboradora:
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD
UC Universidad de Cantabria

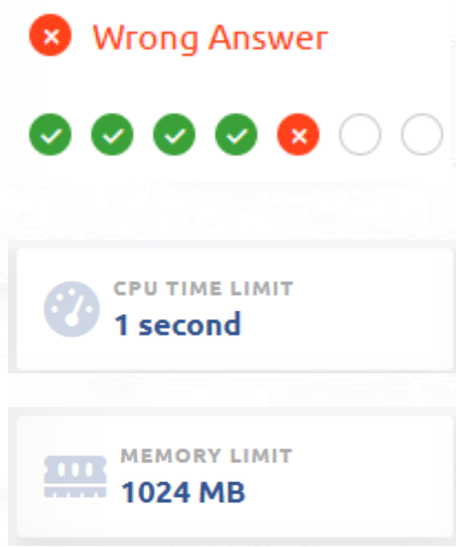
Índice

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

Complejidad

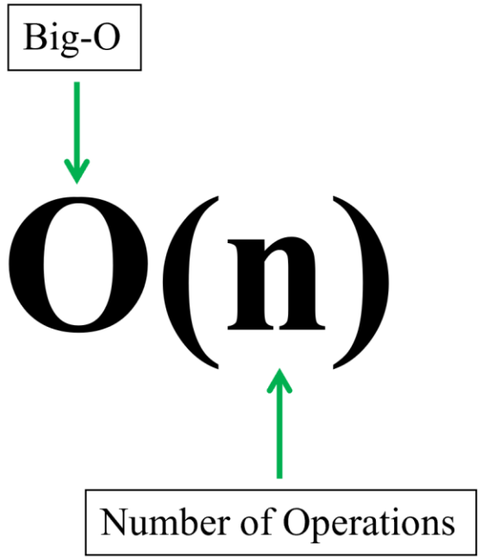
- La **complejidad** de nuestros algoritmos determinará su **eficiencia** a la hora de resolver el problema.
- Este **aspecto** es **crítico** en el contexto de la **programación competitiva**.
- Para obtener un **ACCEPTED** necesitamos cumplir dos aspectos fundamentales.
  - El algoritmo debe ser **correcto**: **para todas** las **entradas** debe proporcionar las **salidas esperadas**.
  - El algoritmo debe ser eficiente en **tiempo** y **memoria**: menor al límite establecido para cada entrada, tanto en **tiempo de ejecución** como en **consumo de memoria**.



Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- Podemos obtener una idea de cuánto va a tardar nuestro código analizando el **número de operaciones** que va a realizar nuestro algoritmo **en relación** con el **tamaño de la entrada (n).**
- Para ello, siempre **consideraremos el peor caso** posible.
- De esta forma, tendremos **una estimación rápida e intuitiva** de la complejidad de nuestro código.



Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- Las **operaciones básicas** se consideran "instantáneas", con un **"coste" asociado de 1**.
  - Operaciones **aritméticas** (1+1, a+a, a*2, b/2...).
  - Operaciones **lógicas** (a or b, a and b, a xor b...).
  - Operaciones **básicas de asignación**, **comparación**, **llamadas** al sistema... (print('hello'), var = 3, if (a == 1)...)
  - Etc.
- Realizar un **número fijo** de operaciones básicas también se considera **O(1).**

O(1)

```
if __name__ == '__main__':
    n = input()
    a = True
    b = False
    c = True
    print(a or b and c)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- Las **operaciones básicas repetidas n veces** sí **dependen** del tamaño de la entrada, es decir, **del valor de n**.
- **O(1*n) = O(n).**
- En resumen: **ejecutar** en bucle **n operaciones O(1)** implica una **complejidad de O(n).**
- El **número de operaciones básicas** que conformen el cuerpo del bucle **no es relevante**.

O(n)

```
if __name__ == '__main__':
    n = input()
    for i in range(n):
        print(i)
```

```
if __name__ == '__main__':
    n = input()
    for i in range(n):
        a = i*2
        b = a + 3
        print(b-a)
        a = b*a
        print(a*b)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- Si nos **saltamos iteraciones**, **la complejidad final tampoco varía**.
- Si, por ejemplo, reducimos la cantidad de iteraciones a la mitad (n/2), la complejidad será O(n).
  - Ya hemos visto que **no se consideran las constantes**, por lo que **O(n/2) = O(n)**

O(n)

```
if __name__ == '__main__':
    n = input()
    for i in range(0,n,2):
        print(i)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- **Anidar bucles sí afecta** a la complejidad final.
- Hacer **n iteraciones** de un bucle que hace n operaciones **implica** una complejidad de **O(n*n) = O(n^2)**

$$O(n^m)$$

```
if __name__ == '__main__':
    n = input()
    for i in range(n):
        for j in range(n):
            a = i*2
            b = j + 3
            print(b-a)
            a = b*a
            print(a*b)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- ¿Qué complejidad tendrá este código?

```
if __name__ == '__main__':
    n = input()
    for i in range(n):
        print(i)
    for i in range(n):
        print(2*i)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- ¿Qué complejidad tendrá este código?

```
if __name__ == '__main__':
    n = input()
    for i in range(n):
        print(i)
    for i in range(n):
        print(2*i)
```

O(n)

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- ¿Qué complejidad tendrá este código?

```
if __name__ == '__main__':
    n = int(input())
    for i in range(n):
        print(n)
    for i in range(n):
        for j in range(n):
            print(i*j)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- ¿Qué complejidad tendrá este código?

```
if __name__ == '__main__':
    n = int(input())
    for i in range(n):
        print(n)
    for i in range(n):
        for j in range(n):
            print(i*j)
```

$$O(n^2)$$

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

$O(2^n)$

- Otro **escenario habitual** es que, en nuestro algoritmo, **el número de operaciones cambie** de forma dinámica durante las **diferentes iteraciones**.
  - Estamos en un escenario recursivo.

- Por ejemplo, en este código, **por cada iteración** se harán **dos operaciones *solve***, que realizarán otras dos llamadas a la función *solve* hasta que se alcance el caso base (n==0).

n

- O(2*2*2*2…*2) = O(2^n)

```
def solve(n):
    if(n==0):
        print("HOLA")
    else:
        solve(n-1)
        solve(n-1)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

O(logn)

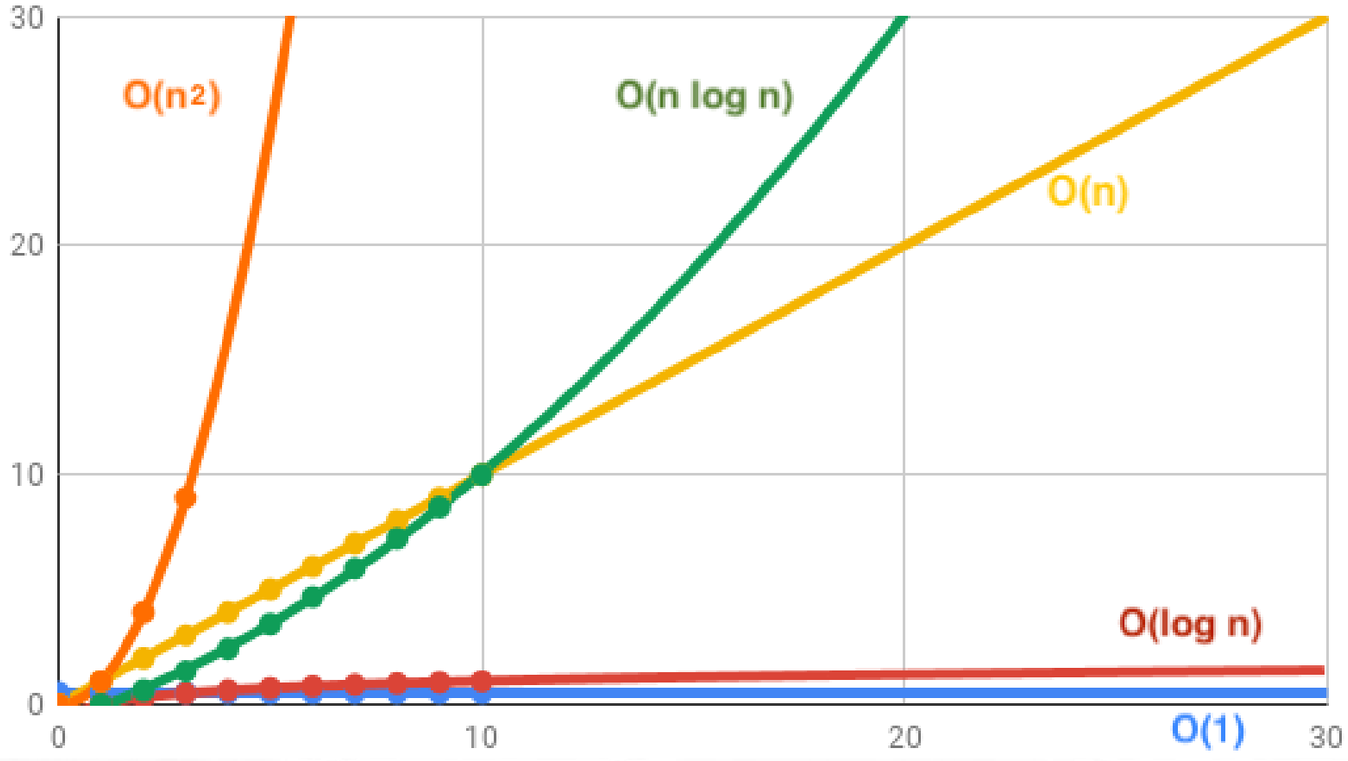
- **Puede** suceder **que este cambio** dinámico se realice **reduciendo el número de operaciones** en cada llamada recursiva.
- Este es un escenario **típico en** la estrategia **divide y vencerás**.
- Por cada **iteración**, el **tamaño de la entrada** se **reduce** (por ejemplo, se divide a la mitad).
- En este caso**, no estamos exponenciando**, sino realizando **la operación contraria**: O(logn)

```
def solve(n):
    if(n<1):
        print(n)
    else:
        solve(n/2)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- Todo esto **aplica cuando el tamaño de la entrada crece**.
- En casos de prueba **con tamaños de entrada pequeños**, es difícil que se de el peor caso, nuestro algoritmo **funcionará** de manera **eficaz**.



Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

- Con todo ello, podemos establecer una tabla que nos de una **noción de la complejidad** que se requiere a nuestro algoritmo **en función del tamaño de la entrada**.

Complejidad esperada	Tamaño de la entrada (n)
O(1)	∞
O(logn)	$2^{1000000}$
O(n)	1,000,000
O(n * logn)	100,000
$O(n^2)$	1,000 ~ 3,000
$O(n^3)$	100 ~ 300
$O(2^n)$	20
O(n!)	12

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

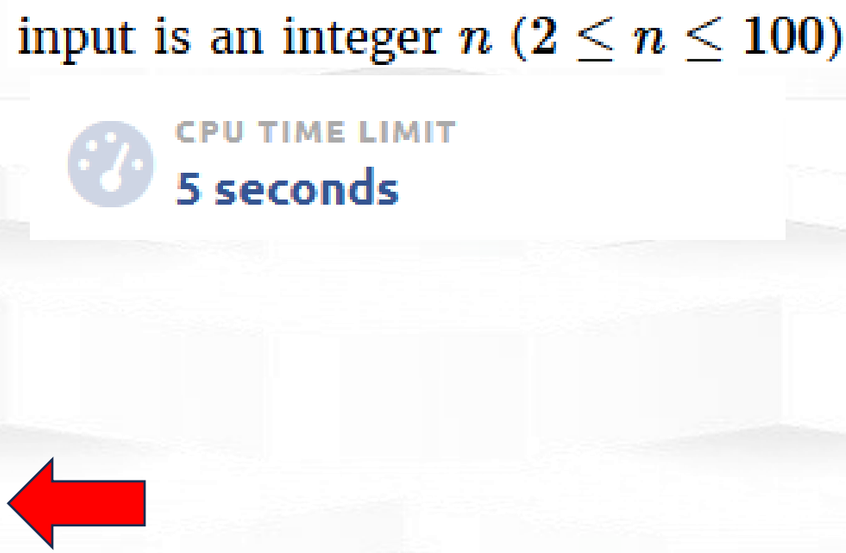
Complejidad esperada	Tamaño de la entrada (n)
O(1)	∞
O(logn)	$2^{1000000}$
O(n)	1,000,000
O(n * logn)	**100,000**
$O(n^2)$	1,000 ~ 3,000
$O(n^3)$	100 ~ 300
$O(2^n)$	20
O(n!)	12

The length of the input string is at least 2 and at most 100 000 characters.

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Complejidad

Complejidad esperada	Tamaño de la entrada (n)
O(1)	∞
O(logn)	$2^{1000000}$
O(n)	1,000,000
O(n * logn)	100,000
$O(n^2)$	1,000 ~ 3,000
$O(n^3)$	**100 ~ 300**
$O(2^n)$	20
O(n!)	12



Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de Datos

Estructuras de datos y complejidad

- Uno de los **elementos determinantes en la complejidad** final de nuestro algoritmo es la elección de las **Estructuras de Datos**.
- **Cada operación** (inserción, eliminación, búsqueda...) tiene una **complejidad diferente** en cada estructura de datos.
- Aunque para nosotros como usuarios de las ED's parezca que todas las operaciones son inmediatas, la realidad es que estaremos **llamando a una función**, por lo que la **operación** será tan **compleja** como la **función** a la que se llama.
- Por todo ello, **es vital elegir una buena estructura de datos** para modelar nuestro problema en función del tipo y número de operaciones que vayamos a realizar.

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos - Listas

- Son estructuras de datos dinámicas que permiten **almacenar una secuencia de elementos**.
- En Python, estos elementos pueden ser de tipos diferentes (no es lo habitual en CP).

```
1   # Definición de una lista
2   mi_lista = [1, 2, 3, 4, 5]
3
4   print(mi_lista)  # Salida: [1, 2, 3, 4, 5]
5
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos - Listas

- Normalmente, se localizan en un bloque contiguo de memoria.
- Se redimensionan automáticamente cuando requieren de más espacio para añadir más elementos.
  - Se dobla su tamaño, copiando los elementos al nuevo bloque de memoria (O(n)).

Operación	Complejidad
Búsqueda mediante índice	O(1)
Búsqueda mediante valor	O(n)
Eliminar un elemento	O(n)
Insertar al final (append)	O(1)
Insertar en una posición concreta	O(n)
Modificar una posición	O(1)

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos - Pilas

- Son estructuras de datos que siguen un esquema **LIFO** (***L****ast* ***i****n*, ***F****irst* ***O****ut*).
- En Python están implementadas en el módulo collections, en concreto como la clase deque.
- Esta implementación se basa en una lista doblemente enlazada, siendo más eficiente que la lista simplemente enlazada.
- Esta estructura de datos es muy útil para implementar recursividad o *backtracking* de forma iterativa.

```
2   from collections import deque
3
4   pila = deque()
5
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos - Pilas

- Las funciones principales de la pila serán: **apliar** un elemento, **desapilar** un elemento, comprobar cuál fue el **último elemento añadido** a la pila (cima de la pila) y comprobar si **está vacía**.

Operación	Complejidad
Apilar (append)	O(1)
Desapilar (pop)	O(1)
Ver cima (pila[-1])	O(1)
Comprobar si está vacía (len(pila)==0)	O(1)

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos - Colas

- Son estructuras de datos que siguen un esquema **FIFO** (***F****irst* ***i****n*, ***F****irst* ***O****ut*).
- En Python están implementadas en el módulo collections, en concreto como la clase deque.
- Esta implementación se basa en una lista doblemente enlazada, siendo más eficiente que la lista simplemente enlazada.
- Esta estructura de datos es muy útil para implementar representaciones de colas en el mundo real o recorridos sobre grafos (BFS).

```
2   from collections import deque
3
4   pila = deque()
5
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos - Colas

- Las funciones principales de la cola serán: **encolar** un elemento, **desencolar** un elemento, **comprobar** cuál fue el **primer elemento** añadido a la cola (frente de la cola) y **comprobar** si está **vacía**.

Operación	Complejidad
Encolar (append)	O(1)
Desencolar (popleft)	O(1)
Ver frente (cola[0])	O(1)
Comprobar si está vacía (len(cola)==0)	O(1)

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos – Colas de prioridad

- Un caso especial de cola bastante útil es la **cola de prioridad**.
- Las operaciones son similares a las de una cola normal, con la diferencia de que **añadir un elemento a la cola requiere de su inserción ordenada**, en función de la **prioridad** otorgada a este elemento.
- En Python, la prioridad se suele representar como el primer elemento de una dupla (pos1, pos2), donde pos1 representará la prioridad del elemento y pos2 el elemento en sí.

```python
import heapq

queue = []
heapq.heapify(queue)
element = (1, 1)
heapq.heappush(queue, element)
heapq.heappop(queue)
```

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos – Colas de prioridad

- Debido a este comportamiento, **la complejidad** de las operaciones **se ve modificada**.

Operación	Complejidad
Encolar (append)	O(logn)
Desencolar (popleft)	O(logn)
Ver frente (cola[0])	O(1)
Comprobar si está vacía (len(cola)==0)	O(1)

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos – Conjuntos y mapas

- Otras estructuras de datos útiles son los conjuntos y los mapas.
- Los conjuntos se basan en el concepto matemático de conjunto: almacenan elementos no repetidos sin ningún tipo de orden.
- Los mapas se basan en el concepto de diccionario: almacenan pares clave-valor, y cada valor es accesible a través de su clave.
- La implementación depende del tipo de conjunto o mapa que se utilice y del lenguaje de programación elegido.
  - Por simplicidad, asumiremos las implementaciones basadas en tablas hash.
- Los conjuntos son muy útiles para realizar búsquedas rápidas de elementos, eliminar duplicados o realizar operaciones de conjuntos.
- Los mapas son útiles para representar elementos relacionados entre sí y acceder a las correspondencias de forma rápida.

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Estructuras de datos – Conjuntos y mapas

- La **complejidad** de los conjuntos y los mapas es **similar** en las diferentes operaciones.

Operación	Complejidad
Buscar (in)	O(1)
Añadir (add)	O(1)
Eliminar (remove/del)	O(1)
Operaciones de conjuntos	O(n)

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Vulnerabilidades, fallos comunes y mitigación

Vulnerabilidades, fallos comunes y mitigaciones

- Como sabemos, como **desarrolladores** es nuestra responsabilidad **generar un código seguro**.
  - No basta con generar un software **funcional y eficiente**, sino que también es importante generar **software seguro**.
- Cada lenguaje de programación, framework de desarrollo, etc. Tiene sus **particularidades** y, por lo tanto, **debemos ser conocedores de ellas para programar de manera segura**.
- Sin embargo, hay **fallos comunes** a todos los lenguajes que se pueden mitigar siguiendo buenas prácticas de desarrollo.

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Vulnerabilidades, fallos comunes y mitigaciones

- La mayoría de estas vulnerabilidades se dan cuando, ante determinadas entradas del usuario, se produce un comportamiento no definido para nuestro algoritmo (***undefined behavior***).
- Este tipo de vulnerabilidades pueden llevar a la materialización de diferentes amenazas:
  - Buffer overflow y sus variantes
  - SQL Injection
  - Command injection
  - Remote Code Execution

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

Vulnerabilidades, fallos comunes y mitigaciones

- Para evitar este tipo de problemas, debemos:
  - **Sanear siempre la entrada de los usuarios**, comprobando que los valores introducidos se corresponden con el tipo esperado y están dentro de sus límites.
  - **Aplicar metodologías de integración continua** que evalúen la calidad de nuestro código y nos informen de las potenciales vulnerabilidades que puede presentar (analizadores estáticos y dinámicos, CD/CI, etc.).
  - Seguir las **guías de desarrollo** del *framework*, lenguaje de programación, etc. Que estemos utilizando.

Raúl Martín Santamaría
Isaac Lozano Osorio
Sergio Pérez Peló

#CátedrasCiber

Módulo II: Estructuras y complejidad

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

Entidad colaboradora:
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD
UC Universidad de Cantabria