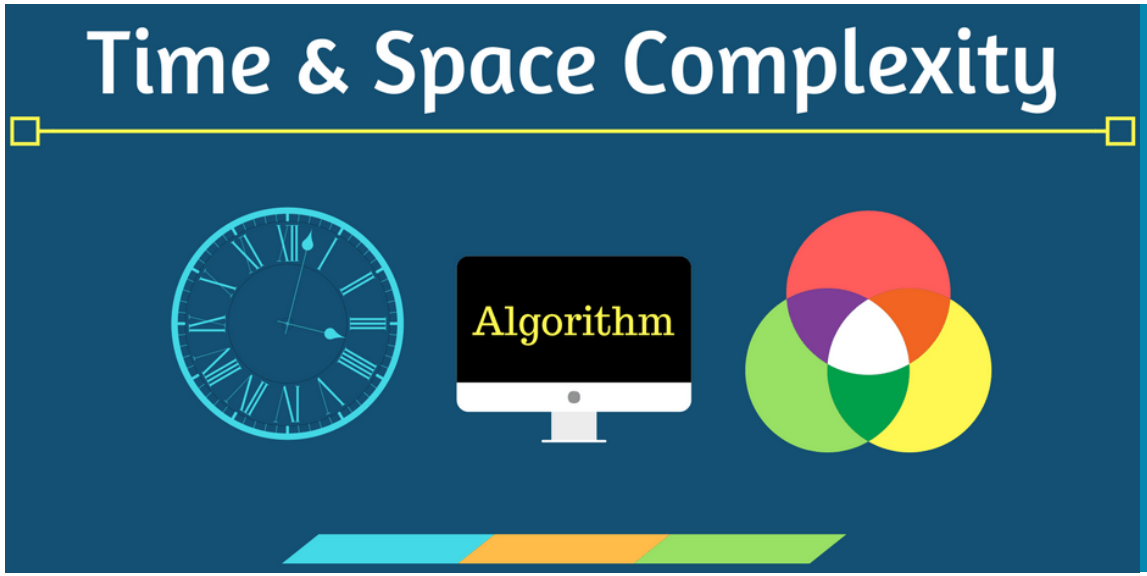


Complejidad Algorítmica

Unidad 1: Comportamiento asintótico, métodos de búsquedas y grafos

Módulo 1: Análisis de Tiempos

Análisis de Algoritmos

Notación Big O

Complejidad Algorítmica

Semana 1 / Sesión 1

MÓDULO 1: Análisis de Tiempos

Contenido

1. Eficiencia de un algoritmo
2. Complejidad temporal vs Complejidad espacial
3. Aproximaciones a la función de tiempo T(n)
4. Notación Asintótica – Notación Big O
5. Clases de Complejidad Algorítmica

Conclusiones y Preguntas

1. Eficiencia de un algoritmo

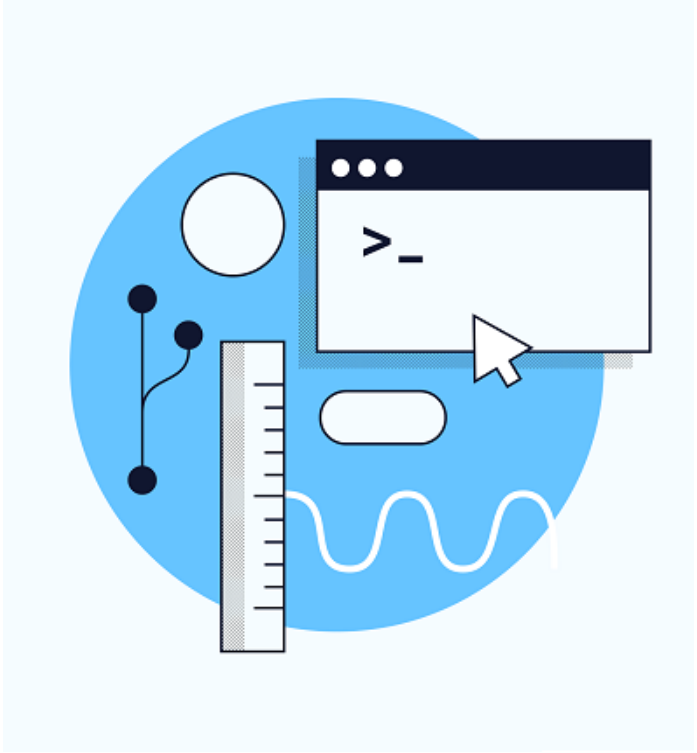
❑ HECHOS: Los algoritmos resuelven problemas

- No hay una sola forma de resolver un problema, y no todas las soluciones son las mejores.
- No todas las soluciones son capaces de utilizar eficientemente nuestros recursos.
- Por lo tanto, necesitamos encontrar la mejor y más eficiente solución a un problema antes de actuar.

1. Eficiencia de un algoritmo

❑ **En Programación:**

- No podemos dejar sujeto a conjeturas, debemos encontrar el mecanismo para encontrar la mejor solución.
- Necesitamos un estándar claro para evaluar la eficiencia de nuestros algoritmos.
- Aquí es donde interviene el concepto de **complejidad algorítmica**, que nos **ayuda a determinar la eficiencia de un algoritmo en función de los recursos necesarios**.



1. Eficiencia de un algoritmo

¿Qué es complejidad algorítmica?

La **complejidad algorítmica** es una **métrica teórica** que nos ayuda a describir el comportamiento de un algoritmo cuando pasa del mejor de los casos al peor de los casos, en términos de:

- ✓ **Tiempo de ejecución** (tiempo que tarda un algoritmo en resolver un problema)
- ✓ **Memoria requerida** (cantidad de memoria necesaria para procesar las instrucciones que solucionan dicho problema).

Aquí es donde intervienen los conceptos de **complejidad temporal** y **espacial**, que nos ayudan a determinar la eficiencia del algoritmo en función de los recursos necesarios.

1. Eficiencia de un algoritmo

¿En qué nos ayuda la complejidad algorítmica?

La **complejidad algorítmica** nos ayuda a comparar entre la efectividad de un algoritmo y otro, y decidir cuál es el que nos conviene implementar.

2. Complejidad temporal vs Complejidad espacial

Complejidad Temporal

(Tiempo de ejecución)

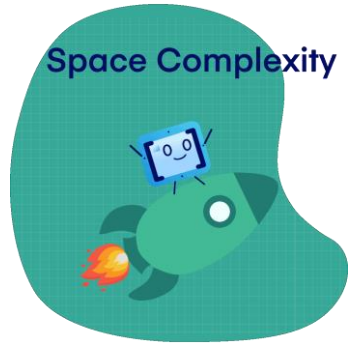
Debemos tener en cuenta que los problemas que debamos resolver mediante el uso de algoritmos, sean capaces de **encontrar una solución en el tiempo esperado**.

Esta solución la expresaremos mediante una **función T** que recibe un input al que llamaremos **n**

T(n) = Tiempo que tardará nuestro algoritmo

Complejidad Espacial

(Memoria requerida)



Conocer cuantos recursos (espacio en memoria) necesitaremos para resolver un problema a través de un algoritmo.

Estos valores se encuentran en función del tamaño del problema
(valor o valores dictados por el número de elementos con los que un algoritmo trabaja, pero a **veces no**….)

3. Aproximaciones a la función de tiempo T(n)

COMPLEJIDAD TEMPORAL: ¿Como podríamos aplicar la función T(n)?

1. **Cronometrando el tiempo en el que se ejecuta un algoritmo**
2. **Contando los pasos con una medida abstracta de operación**

3. Aproximaciones a la función de tiempo T(n)

COMPLEJIDAD TEMPORAL: ¿Como podríamos aplicar la función T(n)?

1. Cronometrar el tiempo en el que se ejecuta un algoritmo

Calcular el tiempo en el que se ejecuta un algoritmo. Sin embargo no es una buena forma de medir los algoritmos, ya que no se puede predecir cuanto demorará a medida que crecen nuestros pasos (instrucciones).

Y las diferencias de tiempo dependerán de varios factores:

- La velocidad de los CPU utilizados (¿tardará lo mismo en ejecutarse en un procesador Pentium III que en un Octacore i7?).
- Los distintos lenguajes de programación utilizados (¿un código en JavaScript demorará lo mismo que uno en C++?).
- Otros procesos que este ejecutando el sistema operativo.

3. Aproximaciones a la función de tiempo T(n)

COMPLEJIDAD TEMPORAL – Ejemplo de cronometrar ejecución del algoritmo

Para una realizar una medida temporal simplemente calculamos la diferencia del tiempo previo y posterior de la ejecución del algoritmo.

```python
import time

def factorial(n):
    respuesta = 1

    while n > 1:
        respuesta *= n
        n -= 1

    return respuesta


def factorial_r(n):
    if n == 1:
        return 1

    return n * factorial(n - 1)
```

```python
if __name__ == '__main__':
    n = 200000

    comienzo = time.time()
    factorial(n)
    final = time.time()
    print(final - comienzo)

    comienzo = time.time()
    factorial_r(n)
    final = time.time()
    print(final - comienzo)
```

- La complejidad del tiempo no se trata de cuántos segundos lleva ejecutar el algoritmo.
- **Se trata de medir el número de operaciones que se ejecutan**.
- Estas operaciones se ven afectadas por el tamaño de la entrada y cómo se organizan sus elementos.

Como veremos a continuación…

3. Aproximaciones a la función de tiempo T(n)

COMPLEJIDAD TEMPORAL: ¿Como podríamos aplicar la función T(n)?

2. Contar los pasos con una medida abstracta de operación

Cada operación aritmética, comparaciones, asignaciones, etc. que ejecute el algoritmo será contabilizada.

Ejemplo:

En el siguiente ejemplo la variable **respuesta** contendrá los números de pasos que realiza nuestro código al ejecutarse.

```
def f(x):
    respuesta = 0

    for i in range(1000):
        respuesta += 1

    for i in range(x):
        respuesta += x

    for i in range(x):
        for j in range(x):
            respuesta += 1
            respuesta += 1

    return respuesta
```

Contemos los pasos:

1 asignación

1000 veces se ejecuta independientemente del valor de x

x veces se ejecuta dependiendo del valor de x

x . x veces se ejecuta cada elemento de rango j por cada elemento de i → x . x = x^2

2 operaciones → **$2x^2$**

1 retorno

Crecimiento de nuestra función es un polinomio

$= 1002 + x + 2x^2$

Que sucede si **x** es igual a 1, a 100 o 1000?

El crecimiento de nuestra función se aproxima al infinito

3. Aproximaciones a la función de tiempo T(n)

ENTONCES:

Solo la complejidad temporal ❯ “No llega a ser una buena métrica para evaluar la efectividad de un algoritmo, ya que parece obvio que cierto algoritmo tardará la mitad de tiempo en ejecutarse en un procesador lo doble de rápido….”

La medición del crecimiento de nuestra función ❯ Debiera estar libre de aquellos términos que no son relevantes para poder medir el algoritmo (y que son variables en el polinomio)

Para lograrlo, nos introducimos a la **Notación Asintótica**.

4. Notación Asintótica – Notación Big O

¿Qué es el crecimiento asintótico?

Crecimiento Asintótico = significa que conforme se va hacia el infinito o el input en un algoritmo crece hacia el infinito.

Características:

- No importan las variaciones pequeñas dentro de la ecuación.
- El enfoque se centra en lo que sucede conforme el tamaño del problema se acerca al infinito.
- Se tiene que tener en cuenta el mejor de los casos, promedio, peor de los casos. Para medir el algoritmo, consideramos el peor de los casos.
- Usando la **notación Big O** únicamente importa el término de mayor tamaño (dentro de la ecuación), esto es, el término que crece más rápido.

4. Notación Asintótica – Notación Big O

Ejemplos aplicando la Notación Big O

#1

O(n)

```
# Ley de la suma

def f(n):
    for i in range(n):
        print(i)

    for i in range(n):
        print(i)

# En este caso el mayor término es n
# O(n) + O(n) = O(n + n) = O(2n) = O(n)
```

#2

O(n²)

```
# Ley de la suma

def f(n):
    for i in range(n):
        print(i)

    for i in range(n * n):
        print(i)

# En este caso el mayor término es n²
# O(n) + O(n * n) = O(n + n²) = O(n²)
```

#3

O(n²)

```
# Ley de la multiplicación

def f(n):

    for i in range(n):

        for i in range(n):
            print(i, j)

# En este caso el mayor término es n²
# O(n) + O(n * n) = O(n * n) = O(n²)
```

#4

O(2n)**

```
# Recursividad múltiple

def fibonacci(n):

    if n == 0 or n == 1:
        return 1

    return fibonacci(n - 1) +  fibonacci(n - 2)

# En este caso el mayor término es 2**n
  (el símbolo ** denota "elevado a"),
# ya que realiza recursividad 2 veces por n veces.
# O(2**n)
```

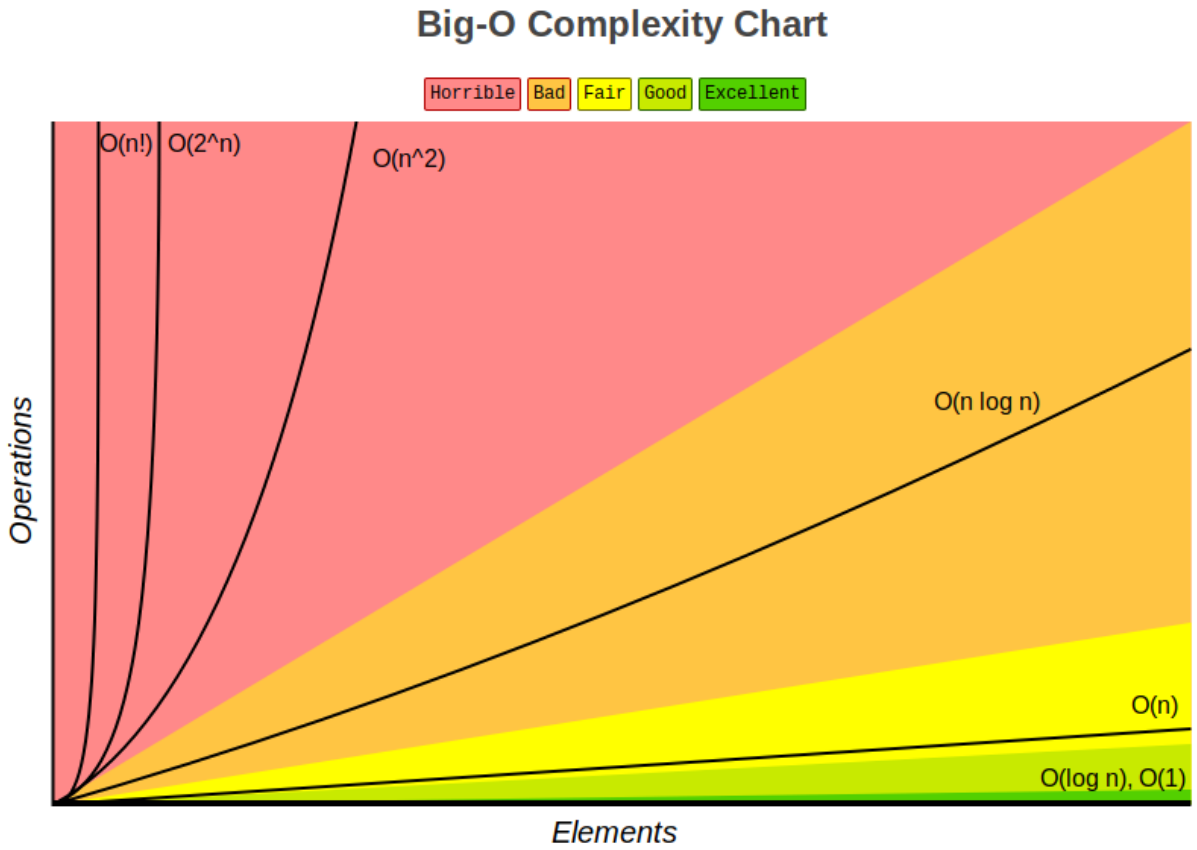
5. Clases de Complejidad Algorítmica

Existen distintos tipos o clases de complejidad algorítmica (llamados también **orden de complejidad**).

En la siguiente tabla se agrupan todas las complejidades que crecen de igual forma, es decir, que pertenecen al mismo orden.

$O(1)$	Orden constante
$O(\log n)$	Orden logarítmico
$O(n)$	Orden lineal
$O(n \log n)$	Orden cuasi-lineal
$O(n^2)$	Orden cuadrático
$O(n^3)$	Orden cúbico
$O(n^a)$	Orden polinómico
$O(2^n)$	Orden exponencial
$O(n!)$	Orden factorial

Órdenes de complejidad comunes.



5. Clases de Complejidad Algorítmica

Existen distintos tipos o clases de complejidad algorítmica (llamados también **orden de complejidad**).

Abreviatura	Orden de Complejidad	Descripción	
O(1)	**Constante**	No importa la cantidad de input que reciba el algoritmo, siempre demorará el mismo tiempo (el tiempo de ejecución es constante)..	Excelente/bueno
O(log n)	**Logarítmica**	La función crecerá de forma logarítmica con respecto al input. Esto significa que en un inicio crecerá rápido, pero luego se estabilizará.	
O(n)	**Lineal**	La complejidad crecerá de forma proporcional a medida que crezca el input.	Justo
O(n log n)	**Log lineal ó Cuasi-lineal**	La función crecerá de forma logarítmica pero junto con una constante.	Malo
O(n²)	**Cuadrático**	La función crece de forma cuadrática. No es recomendable a menos que el input de datos sea pequeño.	Horrible
O(2^n)	**Exponencial**	La función crecerá de forma exponencial, por lo que la carga es muy alta. Para nada recomendable en ningún caso, solo se utiliza para análisis conceptual.	
O(n!)	**Factorial**	La función crece de forma factorial, por lo que al igual que la O exponencial, su carga es muy alta, por lo que jamás se debe utilizar algoritmos de este tipo.	

Conclusiones

- La complejidad algorítmica o ritmo de crecimiento **es una métrica que nos permite como programadores evaluar la factibilidad de las diferentes soluciones de un problema**, y poder decidir con un argumento matemático cuál es mejor mediante comparaciones.

- Este tipo de herramienta teórica cobra importancia cuando se trabaja con **Big Data**, **Machine Learning** o hardware con recursos muy limitados como Arduino donde la optimización es crucial.

- Deberemos elegir aquellos algoritmos que se comportarán mejor al crecer los datos de entrada.

- La complejidad algorítmica nos permite dimensionar y pensar acerca del comportamiento de nuestras soluciones y la manera en la que se codifica, y no solo codear un algoritmo que termine costándonos más en un futuro.

- Se cree que la complejidad algorítmica tiene aparentemente poca utilidad en el día a día de un programador habitual (que tiene poca importancia a nivel práctico), pero definitivamente, quien la domina, adquiere competencias que algunas empresas valoran mucho y solicitan.

PREGUNTAS

Dudas y opiniones