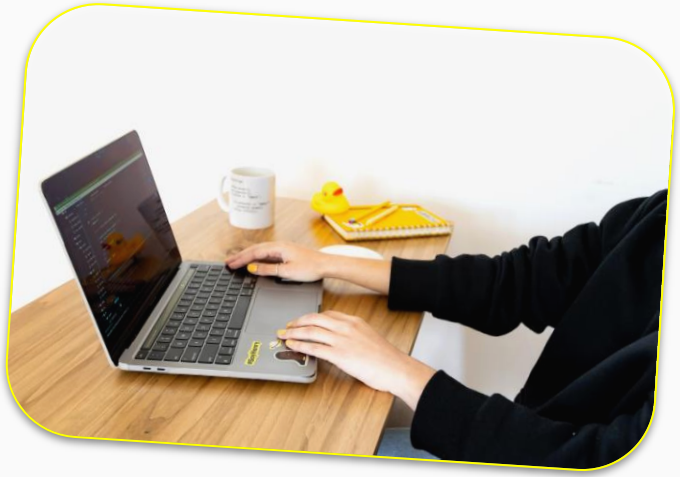
HENRY

Algoritmos I

Data Science



Agenda

- ¿Qué es un algoritmo?
- ¿Qué hace bueno a un algoritmo?
- ¿Cómo medimos la eficiencia de un algoritmo?
- Complejidad de un algoritmo
- Problemas P y NP
- Algoritmos de Ordenamiento y Búsqueda

OBJETIVOS DE LA CLASE

Al finalizar esta lecture estarás en la capacidad de...

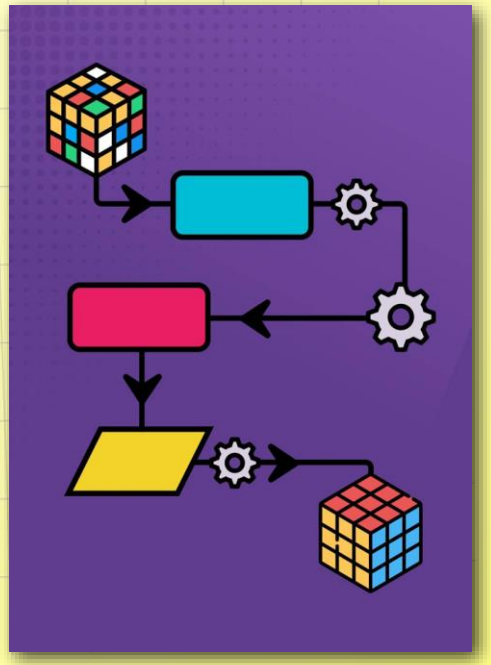
- ➜ Entender el concepto de Algoritmo
- ➜ Conocer el concepto de Complejidad de un Algoritmo y cómo se mide
- ➜ Reconocer qué son los problemas P y NP
- ➜ Identificar los algoritmos de Ordenamiento y Búsqueda más importantes

Algoritmos

¿Qué son?

En Ciencias de la computación los algoritmos van a ser los pasos que la computadora tiene que seguir para poder completar una tarea.

Conocer y encontrar buenos algoritmos y saber cuando utilizarlos es una de las prácticas fundamentales en esta ciencia.



Ejemplo

¿Cómo se hace para transmitir audio y video en tiempo real por internet?

¡Utilizan algoritmos de compresión de audio y video en ambos extremos de la comunicación!

¿Qué hace a un buen Algoritmo?

1. Resuelve un problema: Este es el objetivo principal del algoritmo, fue diseñado para eso. Si no cumple el objetivo, no nos sirve.

2. Debe ser comprensible: El mejor algoritmo del mundo no te va a servir si es demasiado complicado de implementar.

3. Hacerlo eficientemente: No sólo queremos tener la respuesta perfecta (o la más cercana), si no que también queremos que lo haga usando la menor cantidad de recursos posibles.

¿Cómo medimos la eficiencia de un algoritmo?

Contando cuánto tiempo le lleva al algoritmo encontrar la respuesta que buscamos.

Pero eso nos diría la eficiencia de ese algoritmo solamente para la computadora en que corrió, con los datos que tenía y en el lenguaje que se haya implementado.

Para eso se hace un análisis conocido como Asymptotic Analysis, que mide la eficiencia para cualquier situación.

Complejidad de un algoritmo

En general nos interesa conocer qué tan complejo es un algoritmo, o en realidad, lo contrario: qué tan eficiente es.

Hay muchos aspectos que afectan la complejidad de un algoritmo:

- ➜ Tiempo
- ➜ Espacio
- ➜ Otros recursos:
  - Red
  - Gráficos
  - Hardware (Impresoras, CPUs, Sensores, etc...)

Circunstancias

- Mejor Caso
- Caso Promedio
- Peor Caso: Si no tenemos idea de cómo pueden venir los datos, debemos ver este caso.
- Caso Esperado: Si conocemos el dominio del problema y sabemos con algún grado de certeza como van a venir los datos, nos concentramos en este caso.

Cota superior asintótica
(Big O Notation / Notación O grande)

La notación O grande intenta analizar la complejidad de los algoritmos según crece el número de entradas (N) que tiene que analizar, en general es el tamaño del dataset que usa como entrada.

Y lo que busca es una función que crezca de una forma con respecto a N tal que nuestro algoritmo nunca crezca más rápido que esa función, aunque sí puede crecer más lento.

Ejemplos
(Big O)

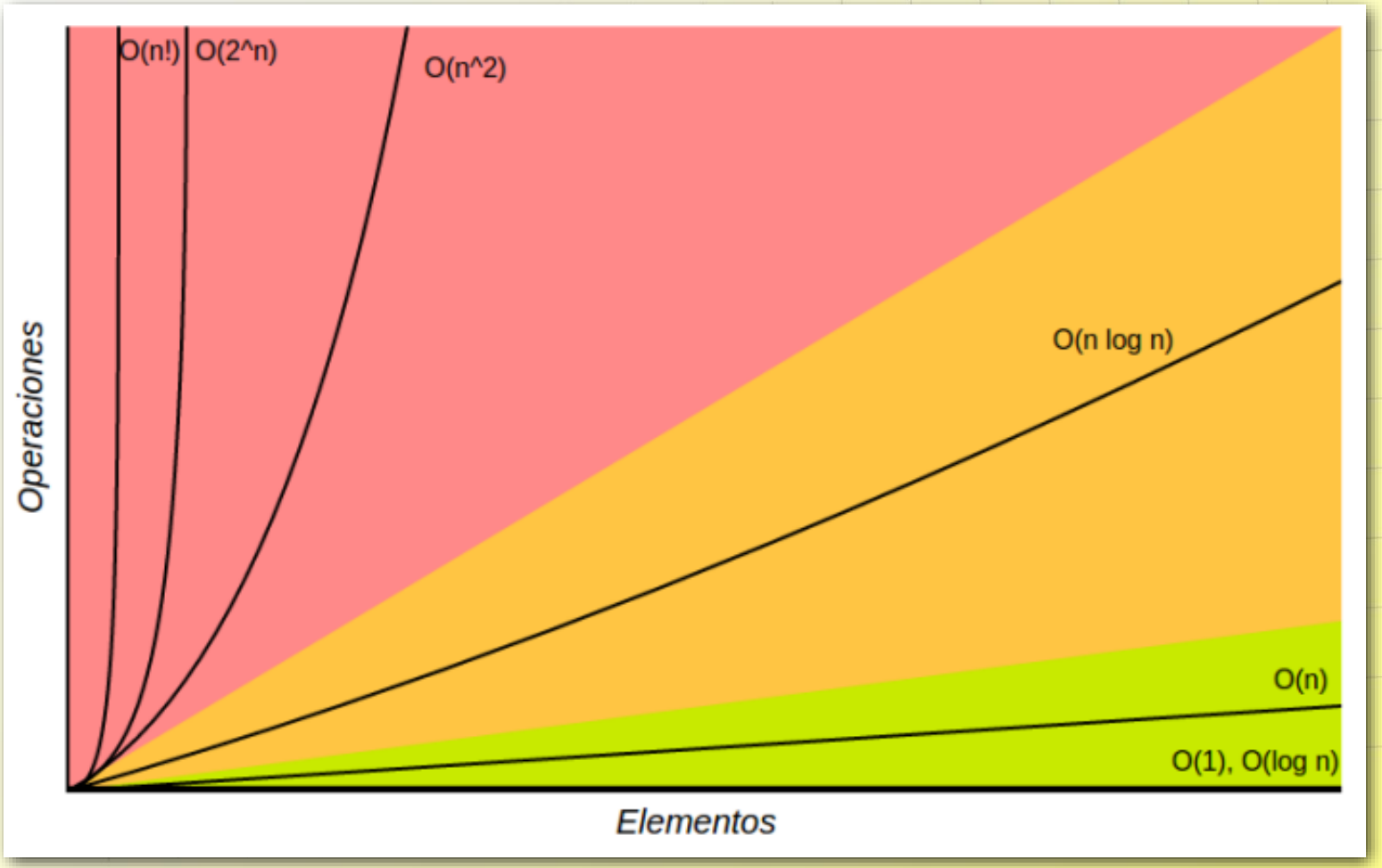
```
>>> max = arreglo[0]
>>> for elemento in arreglo:
>>>   if (elemento > max):
>>>     max = elemento
>>> print(max)
>>> # O ( N )
```

```
>>> max = arreglo[0]
>>> for elemento in arreglo:
>>>   if (elemento > max):
>>>     max = elemento
>>>
>>> min = arreglo[0]
>>> for elemento in arreglo:
>>>   if (elemento < min):
>>>     min = elemento
>>> print(max)
>>> print(min)
>>> # O( N + N) = O(2N)
```

```
>>> max = arreglo[0]
>>> min = arreglo[0]
>>> for elemento in arreglo:
>>>   if (elemento > max):
>>>     max = elemento
>>>   if (elemento < min):
>>>     min = elemento
>>> print(max)
>>> print(min)
>>> # O( N ) = O(N)
```

Notación Big O

- O(1): Siempre tarda lo mismo
- O(n): Hace una acción por cada entrada
- $O(n^2)$: Por cada entrada, recorre todas las entradas de nuevo
- O(Nc): Es el concepto general del anterior, por cada entrada el algoritmo recorre todas las demás entradas c veces.
- O(Log n): En cada paso recorre la mitad de las entradas que quedan.
- O(N!): Esta complejidad en general aparece en algoritmos que acomodan ítems, porque hay N! Formas de acomodar N ítems.


O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operaciones
Elementos

Ejemplo

Si tuviéramos una computadora que es capaz de ejecutar 1.000.000 instrucciones por segundo, ¿cuánto tiempo tardarían algoritmos de distinta complejidad en terminar de correr con un N de entrada de 1000?

Runtime	F(1,000)	Time
O(Log N)	10	0.00001 sec
$O(\sqrt{N})$	32	0.00003 sec
N	1,000	0.001 sec
N^2	1,000,000	1 sec
2^N	1.07×10^{301}	3.40×10^{287} years
N!	4.02×10^{2567}	1.28×10^{2554} years

Problemas **P** y **NP**

Problemas P

Tiempo polinómico: Si la cantidad de operaciones que necesita un algoritmo para terminar es un polinomio.

Además, si para llegar al resultado realiza una cierta cantidad de pasos, y siempre va a realizar los mismos, podemos decir que el algoritmo es determinístico.

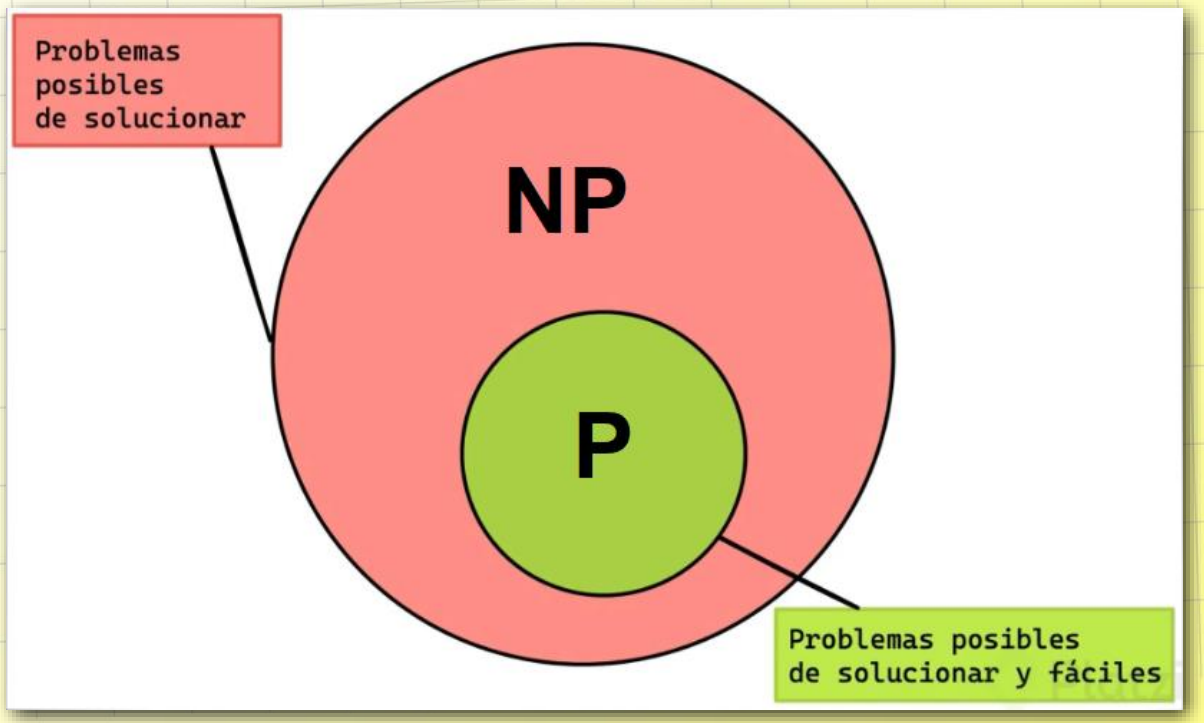
Problemas NP

Tiempo polinómico no determinístico: Encontrar la solución real nos puede llegar a tomar muchísimo tiempo!

Imaginemos que nos dan un set de números y nos preguntan si algún subset del set suma 0:

```
{-10, 60, 95, 25, -70, -50}
solucion 1: -10 + -50 + 60 = 0
```

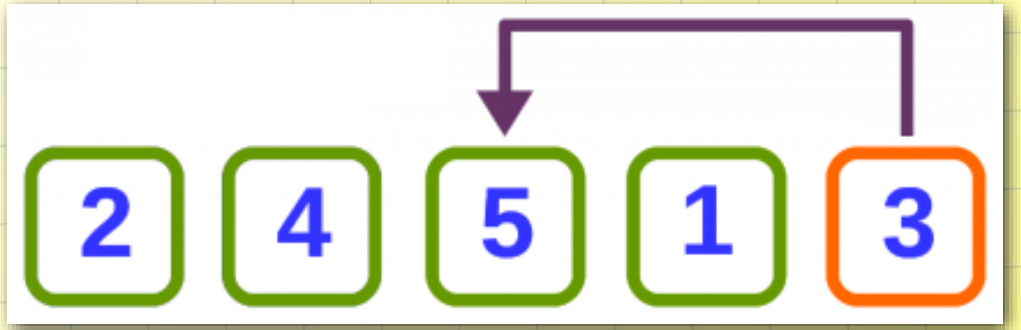
Problemas P y NP



Algoritmos de **ordenamiento**

Inserción (Insertion Sort)

Extrae el elemento del conjunto y lo agrega en la posición que le corresponde.



Selección (selection sort)

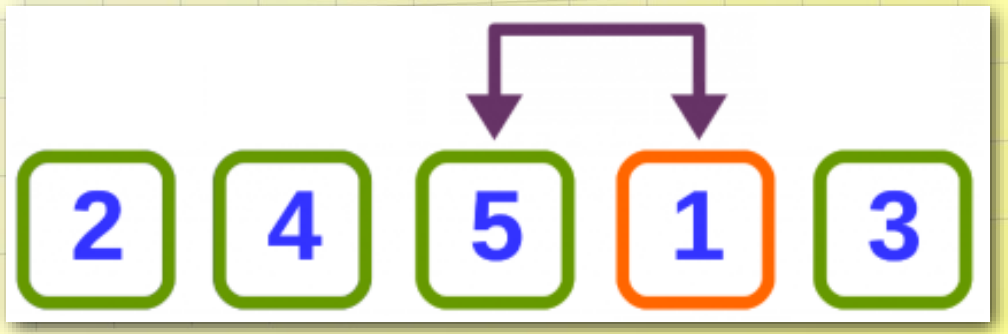
Encuentra el menor elemento y lo intercambia con el que está en la primera posición.

Sigue con el más pequeño de los que quedan, y así sucesivamente.

Burbujeo (Bubble Sort)

Va ordenando de a dos elementos adyacentes.

Recorre todo hasta que quedan todos los elementos ordenados.




HENRY