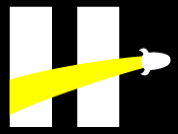
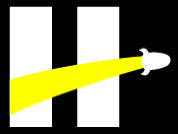
Algoritmos I



Algoritmos

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.

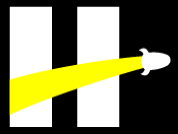
Algunos ejemplos de la vida real: ¿Cómo se hace para transmitir audio y video en tiempo real por internet? Utilizan algoritmos de compresión de audio y video en ambos extremos de la comunicación!



Algoritmos

¿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.

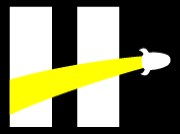


Algoritmos

¿Cómo medimos la eficiencia de un algoritmo?

Contando cuanto 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 tenia y en el lenguaje que se haya implementado

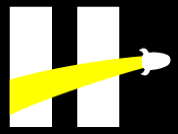
Para eso se hace un análisis conocido como **Asymptotic Analysis**



Algoritmos

Adivinando un número en un arreglo de 16 elementos

- Búsqueda Linear por fuerza bruta: en promedio vamos a ganar el juego en 8 veces.
- Que alguien sepa el número y nos dice si el que elegimos es mayor o menor. Sabemos que el número máximo de veces que vamos a necesitar sale de la cantidad de veces que podemos dividir la lista en dos hasta que el resultado sea uno:
  - 1 = N / 2x // N es el largo de la lista y x el número buscado
  - Al ser 1 y 4 los casos extremos en promedio vamos a necesitar tan sólo de 2 jugadas hasta adivinar el número.

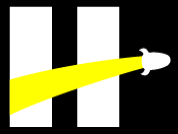


Algoritmos

Complejidad de un algoritmo

En general nos interesa conocer qué tan complejos son los algoritmos, o en realidad, lo contrario: que tan eficiente es un algoritmo. Hay muchos aspectos que afectan la complejidad de un algoritmo:

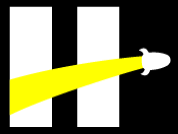
- Tiempo
- Espacio
- Otros recursos:
  - Red
  - Gráficos
  - Hardware (Impresoras, Cpus, Sensores, etc...)



Algoritmos

Circunstancias

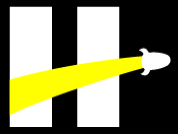
- Mejor Caso
- Caso Promedio
- Peor Caso <- Si no tenemos idea de como pueden venir los datos, debemos ver este caso.
- Caso Esperado <- Si conocemos el dominio del problema y sabemos con algún grado de certeza como van a venir los datos, nos concentramos en este caso.



Algoritmos

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 si puede crecer más lento.

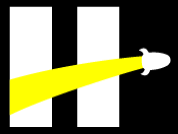


Algoritmos

```
>>> 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)
```

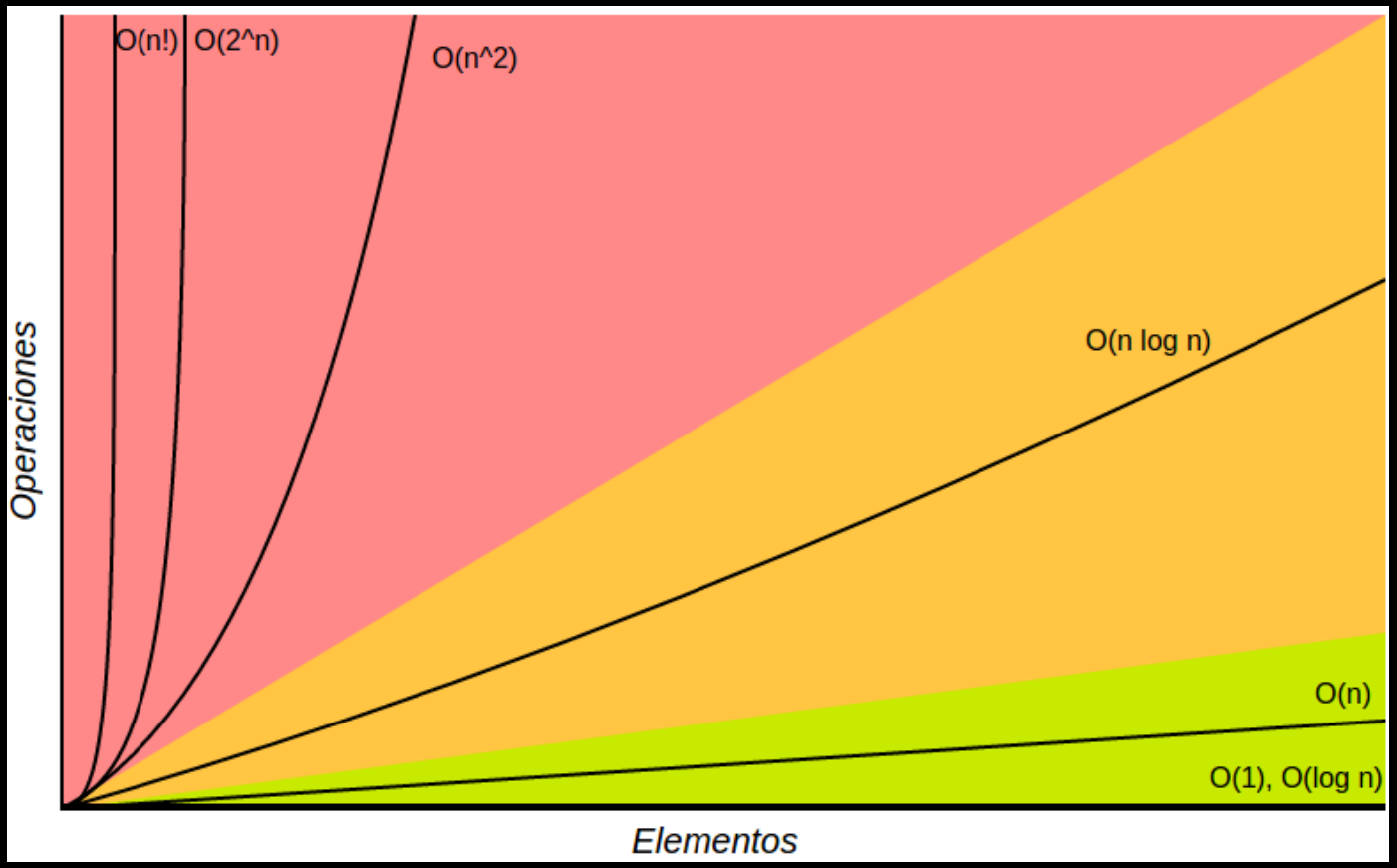
```
>>> max = arreglo[0]
>>> min = arreglo[0]
>>> for elemento in arreglo:
>>>   if (elemento > max):
>>>     max = elemento
>>>   if (elemento < min):
>>>     min = elemento
>>> print(max)
>>> print(min)
>>> # O( N ) = O(N)
```

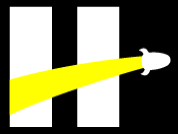


Algoritmos

- 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.

Algoritmos

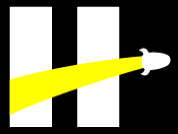




Algoritmos

Si tuvieramos una compu 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



Algoritmos

Problemas P y NP

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**.

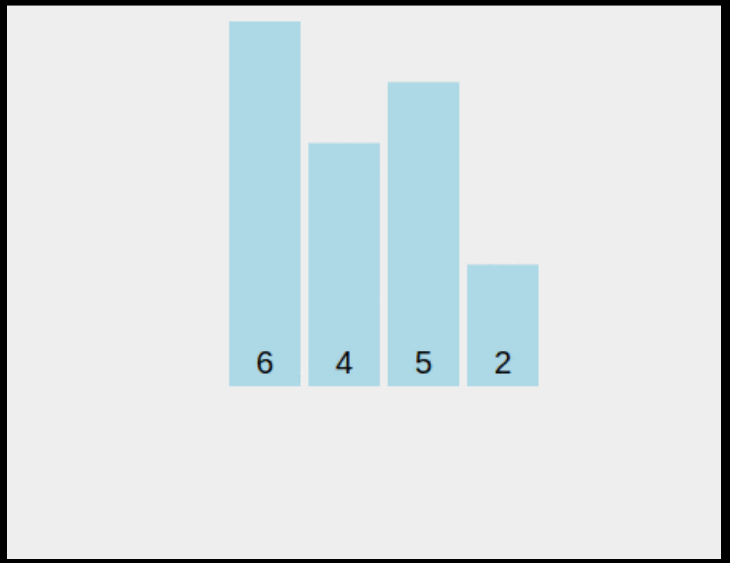
NP - Tiempo polinómico no determinístico: Encontrar la solución real nos puede llegar a tomar muchísimo tiempo! Imaginen que nos dan un set de números y nos preguntan si algún subset del set suma 0:

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

Algoritmos de Ordenamiento

Inserción

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



Algoritmos de Ordenamiento

Selección

Empieza en la posición mínima y busca el elemento que corresponde a ese lugar.

Algoritmos de Ordenamiento

Burbujeo

Recorre los elementos N veces y si dos ítems adyacentes están desordenados los intercambia.
Cuando no hay más intercambios termina.