Big O.

Clª cor, write again so u can understand future sdfj

Notación usado para medir los recursos en tiempo y en espacio.

Big O(n) n = numero elementos

Constant Time O(n) Escalado Lineal - Escalado Lineal

Tiempo cont

```
func ()
    for (1 → n)
        | print (10 * 1000)
```

} Siempre tardara lo mismo independientemente del inicio del for.

| Generalmente los valores constantes se ignoran ya que son constantes en el rendimiento del algoritmo

Constante = no escala con el input, siempre tarda lo mismo

Big O(1): Constant Algorithm

Orden de crecimiento

Good

EFFICIENCY

- → $O(1)$ - Constantes
- → $O(\log n)$ - crecimiento logaritmico
- → $O(n)$ - Crecimiento Lineal
- → $O(n \log n)$ - Linearithmic
- → $O(n^2)$ → cuadrado
- → $O(n^3)$ → cubico
- → $O(2^n)$ exponencial
- → $O(n!)$ Factorial

Bad

Cuando calificamos un algoritmo en Big O siempre nos quedamos con el peor caso, es decir si tenemos:

$O(\log n) + O(n) + O(2^n)$

nos quedamos con el $O(2^n)$ como calificacion del algoritmo ya que es el peor caso

$O(n^2)$ - Square

Ej:

```
func s(n) {
    for(i → n)
        for(j → n)
            log(i,j)
```

Básicamente es un crecimiento cuadrado. Si por ejemplo n fuese 4 la función haría 4 loops que a su vez harían 4 loops ejecutando el algoritmo 16 veces (4^2).

Los ejemplos más comunes son loops anidados.

$O(n^3)$ - Cubic

No necesita mucha explicación es:

O (log n) - Logarithmic

$n^m = x$

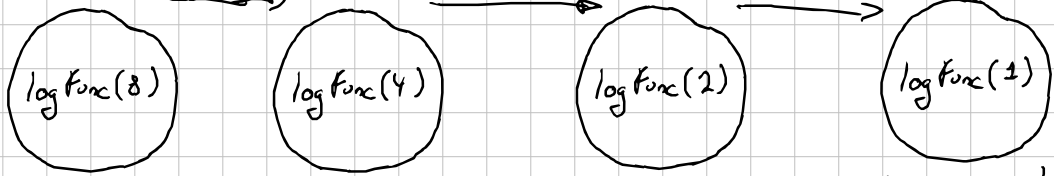
$2^m = 8$ ▷ $\log_2 8 = ?$

$2^3 = 8 \rightarrow \log_2 8 = 3$

Example:

```
func logFunc(n)
    if (n==0) return;
    n = n/2
    return logFunc(n);
```

Ej



Ya no es divisible por 2 así que paramos

Es importante mencionar que en complejidad siempre usamos la base 2 para la medición.

Ejemplo sin recursividad:

```
func logm(n)
    while (n>1)
        n = n/2
```

En el ejemplo anterior el loop pasaría tres veces como en la recursiva también

O(log n) mide como si fuese lo contrario del cuadrado en referencia a cantidad de iteraciones.

Básicamente a qué potencia tendríamos que elevar a 2 para hacer n

Binary search with O(log n)

- Solo funciona en arrais ordenados.

Basicamente vamos a ir partiendo entre dos el array hacia arriba o hacia abajo hasta que encontremos el elemento a buscar en el punto actual

```
f()
  arr[]
  start=0
  end= arr.lenght - 1
  x = 0
  while (true)
    x= (end-start)/2 + start . integer or
    if (arr[x] > n) end= x-1
    elseif (arr[x]<n) start=x+1
    else return arr[x]
```

```
start = 0        x = 0
end= lenght -1
f(arr, start, end, n)
start>end  → x = ((start + end)/2).Int
false        if (arr[x] = n) rtr t
             else if (arr[x] > n)
                f(arr, start, x - 1, n)
             else (arr[x] < n
                f(arr, x+1, end, n)
```

$O(n \log n)$ -

Ej.

```
{ nLogNFunc(n) {
    x=n
    while(n>1)
        n=(n/2).integer
        for(1->x) {
            log(i)
        }
```

Basicamente → hacer $\log n$ veces n

→ podríamos expresar como $O(n \cdot \log n)$ de complejidad

$o = o$