Falar de tempo constante é O(1)
• Melhor e mais rápido

O(1) - Constant Time:

```
def get_first_element(arr): return arr[0]
```

[ ]

↳ Independente do input, o algoritmo vai demorar a mesma quantidade de tempo. Por isso é O(1). Por exemplo: Pegar o primeiro elemento de um array.

Binary Search → Tempo de execução ou 2 ou 3 passos

O(log n) - Logarithmic time:

log2(10) = 3.321
log2(20) = 4.321
log2(40) = 5.321

2

$\log_2 n$

7	2	3	4	~~5~~	~~6~~	~~7~~	~~8~~	~~9~~	~~10~~

3 ou 4 passos

$$\log_2 10 = x$$
$$2^x = 10$$
$$x = 3.321$$

↳ Conforme o tamanho do input aumenta n, o tempo de execução aumenta log n. Sendo assim, eles não vão aumentar em uma proporção linear, mas sim em uma proporção logarítmica. Dessa forma o INPUT cresce muito mais que o tempo de execução.

* Lembrando que a representação O(log n) serve para representar o pior caso (que aqui é quando estamos procurando o número 1). Lembrando que o número de passos não pode ser representado por um número decimal, dessa forma 4 passos é o pior cenário.

O(n) - Linear Time:

x → 2x
70 → 20

7 2 3 9 7 2 3 4

O(n log n) - Linearithmic time:

7 2 3 9 7 2 3 4

Ex: MergeSort

O (n^2) - Quadratic Time:

10 → 100

1

7 2 3 9 7 2 3 4

```
for i in range(n)
    for j in range(n)
```

O(2^n) - Exponential Time

```
def fibonacci(n):
    if n ≤ 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

→ A cada novo elemento a quantidade de passos dobra. Tempo exponencial escala terrivelmente mal com o tamanho do input.

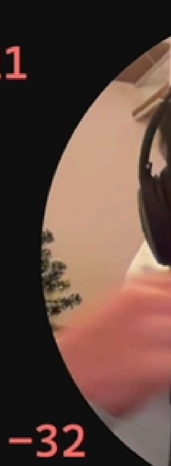
O(n!) - Factorial Time

7 | 2

1 - 2 - 12 - 21

7 | 2 | 3

1 - 2 - 3 - 12 -21 -13 - 31 -23 -32



* Considerar sempre o PIOR dos casos (complexidade pessimista)