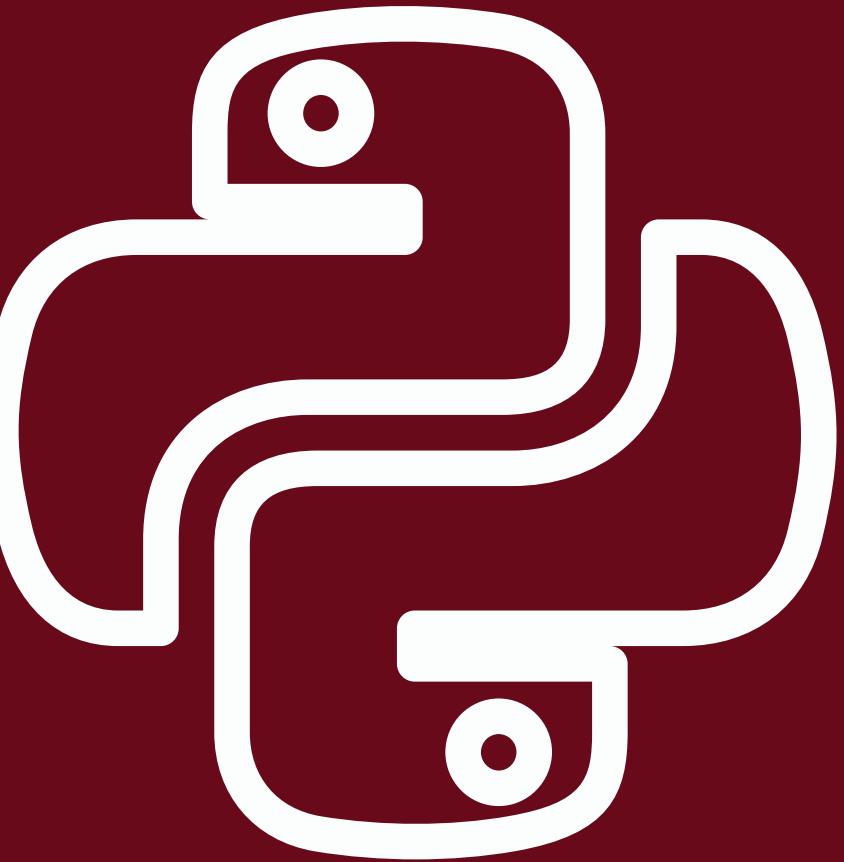


ESTRUTURA DE DADOS



NOTAÇÃO BIG-O

COMO COMPARAR 2 ALGORITMOS?

CONSIDERA DIFERENÇAS ENTRE PODER DE PROCESSAMENTO, SISTEMA OPERACIONAL, LINGUAGEM DE PROGRAMAÇÃO.

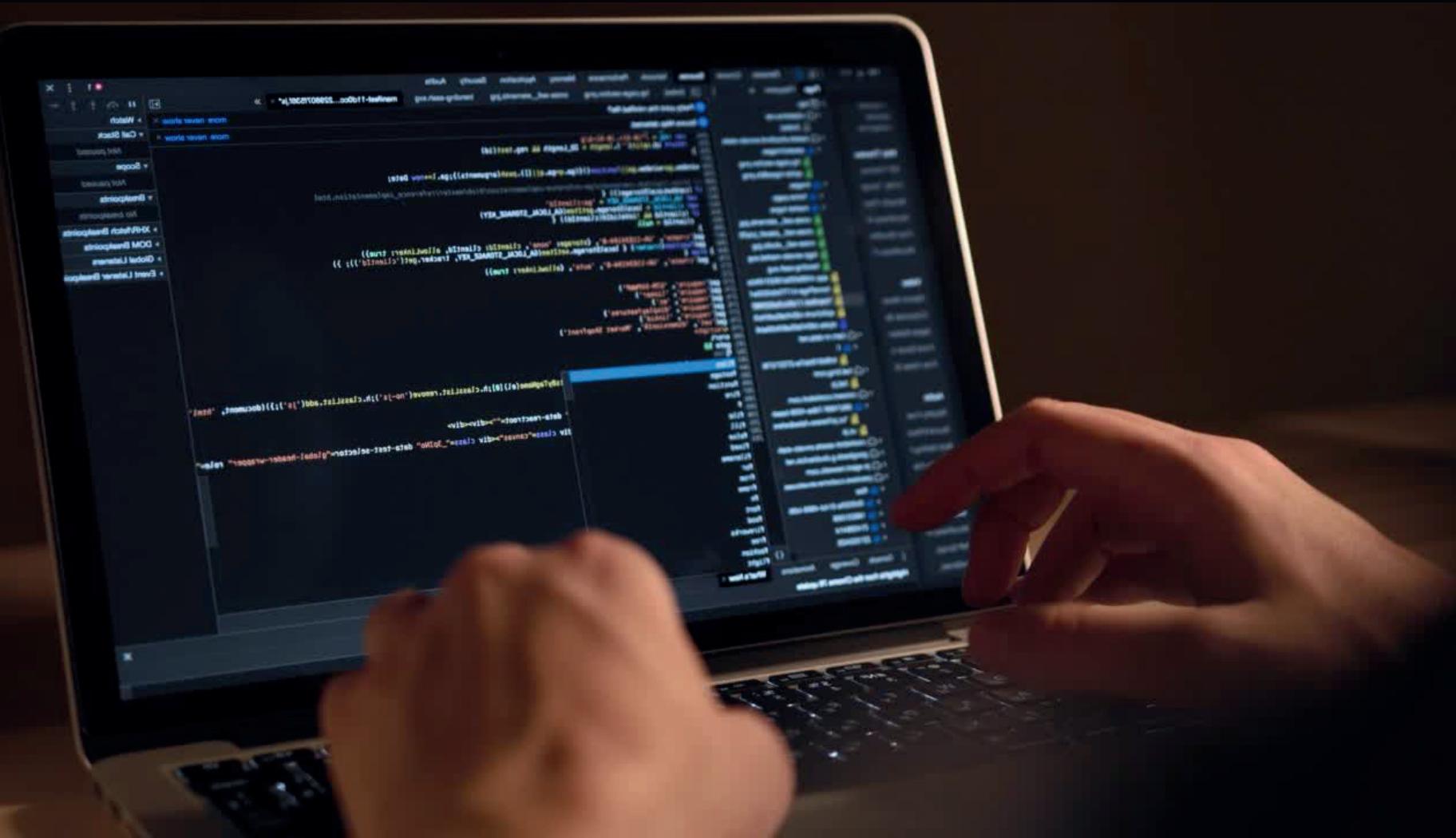
COMPARAÇÃO OBJETIVA ENTRE ALGORITMOS.

O QUANTO A "COMPLEXIDADE" DO ALGORITMO AUMENTA DE ACORDO COM AS ENTRADAS ?



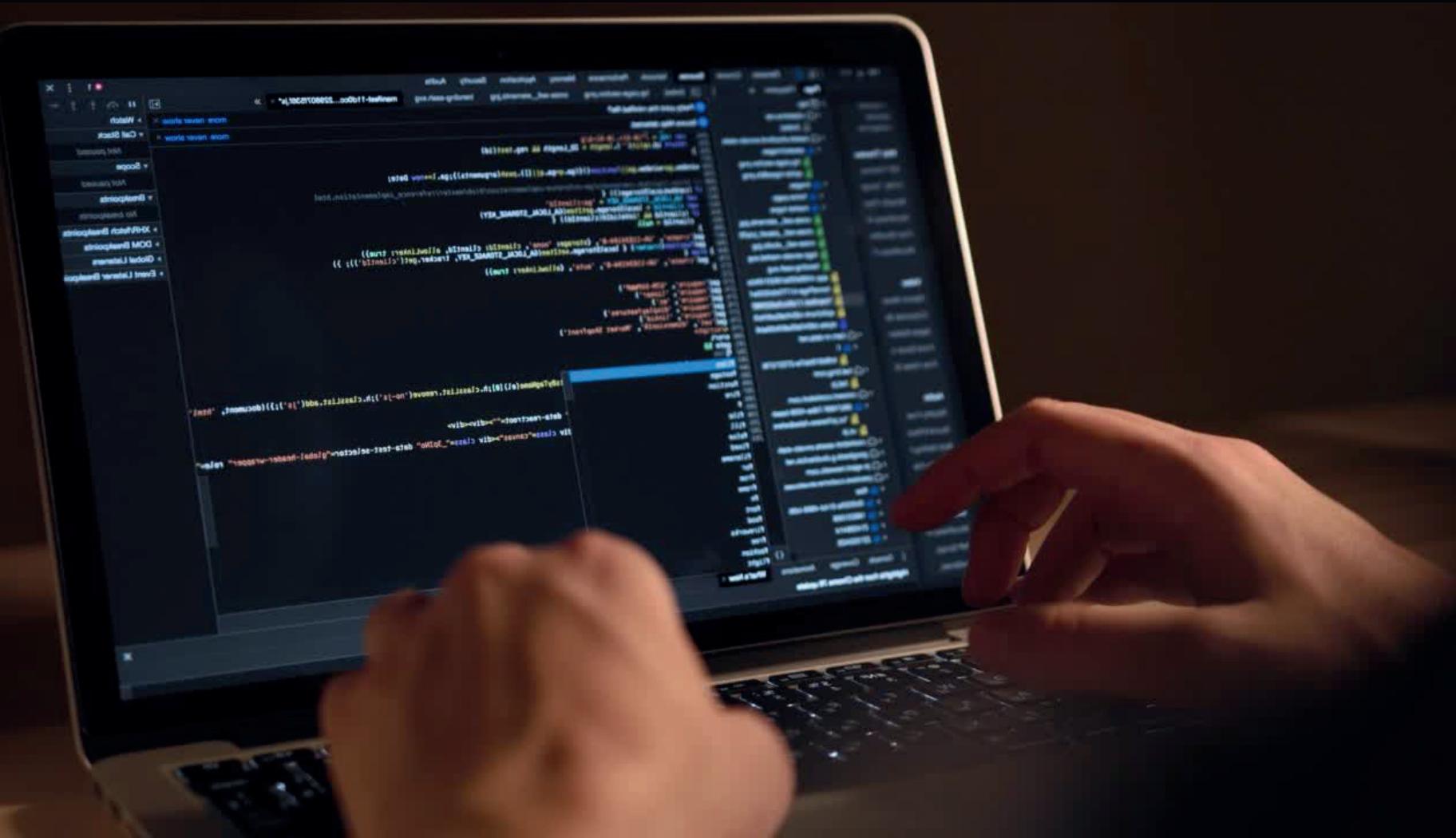
Um exemplo “prático”.

Vamos supor que você pede para que um desenvolvedor crie uma **função** para fazer o **somatório** de determinados valores e você passa esse mesmo problema para um outro desenvolvedor.



Você deve passar um número, vamos supor que você passe o número 10 e o objetivo é criar uma função que vai receber o número 10 e vai fazer um somatório de todos os algarismos a contar do zero até chegar ao número informado. Exemplo: $0+1+2+3+4+5+6+7+8+9+10$ que é igual a 55.

Uma função simples, não?!



Podem haver funções “**diferentes**” ! O programador **A** faz uma função com determinada estrutura e o programador **B** também faz essa função para fazer o somatório dos valores, porém com uma estrutura diferente.
Como você vai saber qual é a melhor abordagem para resolver esse problema?

Para matar a dúvida de qual seria o melhor algoritmo a ser utilizado, fazemos a análise de complexidade com notação Big-O.

Na notação Big-O mascaramos toda a parte de poder de processamento, SO e linguagem de programação para analisarmos somente o algoritmo comparando-os de forma objetiva.

Outro premissa que temos é o quanto de “complexidade” aumenta de acordo com as entradas... coisa que já iremos verificar com algumas implementações utilizando a nossa situação problema da função soma.

Desenvolvedor A

O mesmo criou a função abaixo.

Função 1 - $O(n)$

```
1 # 11 passos
2 def soma1(n):
3     soma = 0
4     for i in range(n + 1):
5         soma += i
6
7     return soma
```

] ✓ 0.0s

Desenvolvedor A

Passou o parâmetro.



A screenshot of a terminal window with a dark background. On the left, there are navigation icons: a play button, a dropdown arrow, and a square icon. The main area displays two lines of code output:

```
1 soma1(10)
[2] ✓ 0.0s
```

Below this, three dots indicate more output, followed by the number 55.

Desenvolvedor A

Vamos agora analisar o tempo de execução dessa função.



366 ns \pm 12.6 ns per loop (mean \pm std. dev. of 7 runs, 1,000,000 loops each)

366 ns \pm 12,6 ns por loop (média \pm desvio padrão de 7 execuções, 1.000.000 loops cada)
ns = nanosegundo.

Desenvolvedor B

O mesmo criou a função abaixo.

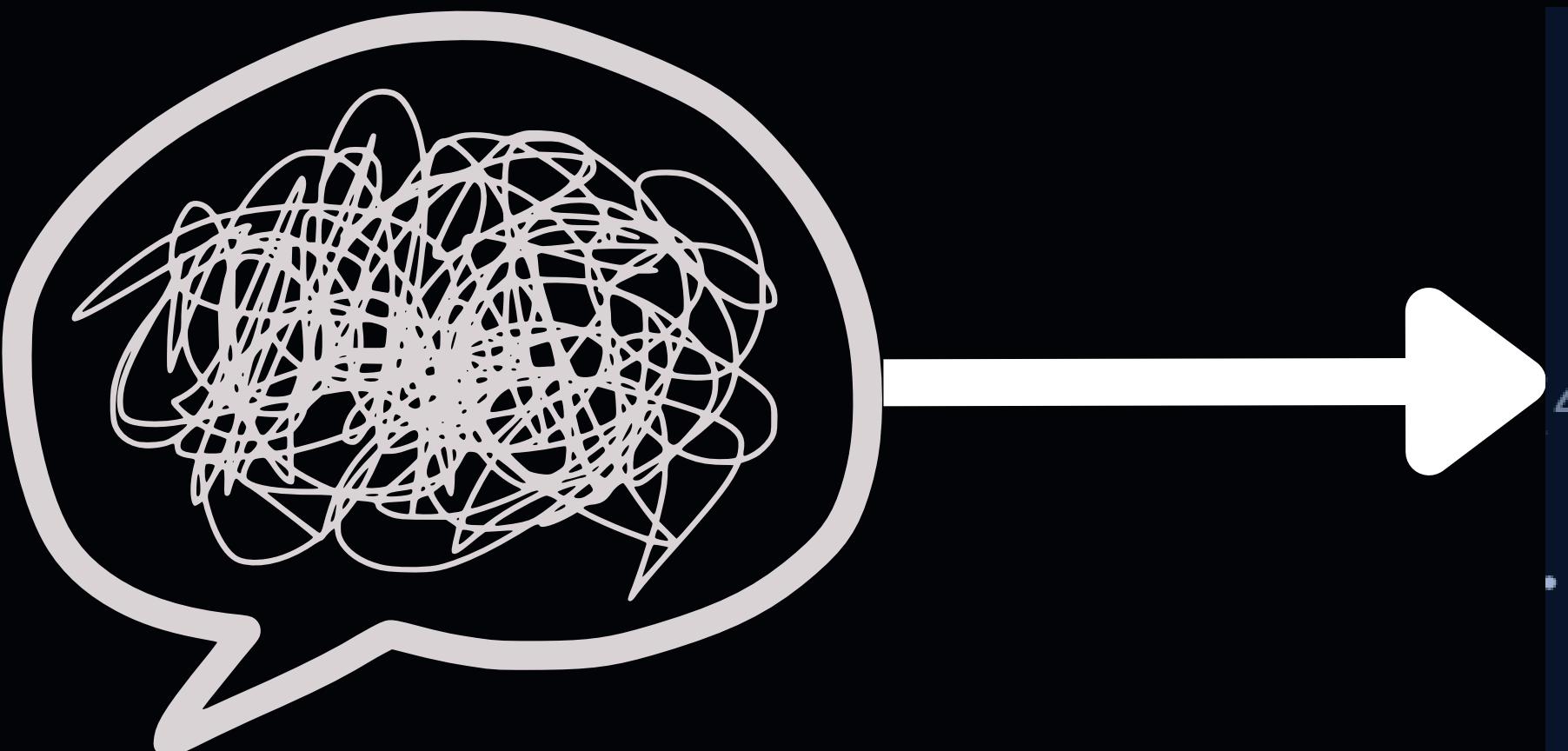
+ Code

+ Markdown

```
1 # 3 passos
2 # O(3)
3 def soma2(n):
4     return (n * (n + 1)) / 2
```

✓ 0.0s

“Caio, não entendi a função do dev B... traduz ai que to nerfado em matemática hoje”



```
4] ✓ 0.0s
..      55.0
1 (10 * 11) / 2
```

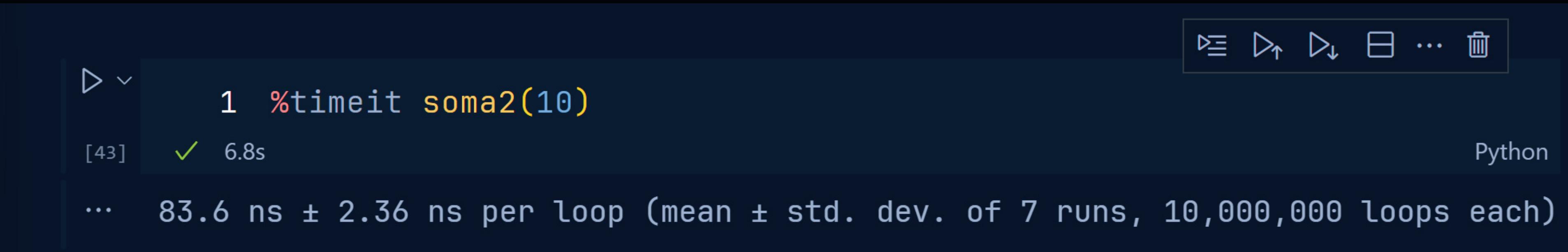
Desenvolvedor B

Passou o parâmetro.

```
▶ 1 soma2(10)
[6] ✓ 0.0s
...
55.0
```

Desenvolvedor B

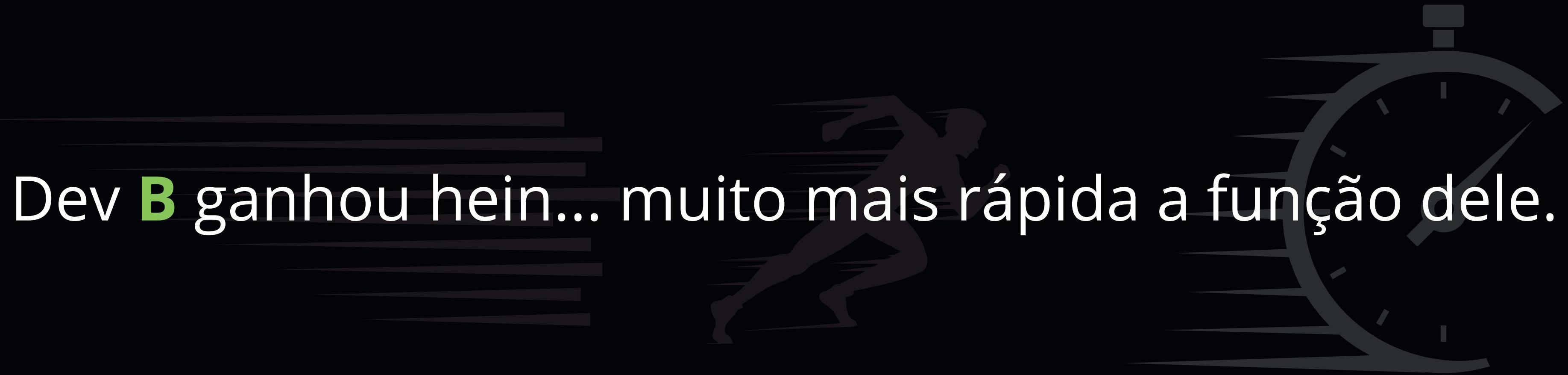
Vamos agora analisar o tempo de execução dessa função.



```
1 %timeit soma2(10)
[43]    ✓ 6.8s
... 83.6 ns ± 2.36 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

ns = nanosegundo.





Dev B ganhou hein... muito mais rápida a função dele.

Resumindo

Na primeira função o Big-O foi **O(n)**, onde o **n** foi a quantidade de passos dados para execução completa dela, ficando **O(11)**. Onde o primeiro passo seria setar a variável `soma=0` e os outros 10 passos estariam diretamente ligados ao valor do número passado por parâmetro que iria iterando com a estrutura escolhida pelo dev **A** que foi a **for**.

Suponhamos que o `soma1` tivesse o parâmetro setado em **50.000**. Logo o seu Big-O seria **O(50001)** pois pela estrutura escolhida (`for`) ele precisou de 50 mil e um passos... o que resultaria em muito tempo para executar.

Resumindo

Na segunda função o Big-O foi **O(n)**, onde o **n** foi a quantidade de passos dados para execução completa dela, ficando **O(3)**. Onde o primeiro passo seria uma multiplicação, o segundo um somatório e o terceiro uma divisão.

Suponhamos que eu passe o número 1.000 como parâmetro na função soma2... ela continuaria com 3 passos da mesma forma.

Resumindo

Enfim, a primeira função **não é uma função escalável** pois quanto **maior o número passado... mais tempo** ela vai demorar para processar.





Outro exemplo “prático”
utilizando listas!

Bora mais uma!

E vaaamos a outra partida desta luta!

Será que o Dev A novamente perderá para o Dev B?
Ou a revanche será agora?!

ODD pagando bem, façam suas apostas!!



Os códigos precisam ser eficientes!

Não adianta de nada que escrevamos um código que resolva algum problema se quando formos colocá-lo em produção ele demora eras para executar... consumindo muitos recursos de hardware por exemplo.



O desafio desta vez é criar uma função que adicione 1.000 números a uma lista.



Não, não é um To-do list,
só usei o gif pois achei
maneiro.

Desenvolvedor A

```
1  # O(1000) → O(n)
2  def lista1():
3      lista = []
4      for i in range(1000):
5          lista += [i]
6      return lista
8]    ✓  0.0s
```

Desenvolvedor A

Desenvolvedor A

```
1 %timeit lista1()
```

```
[11] ✓ 7.0s
```

Py

```
86.9 µs ± 929 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

ns = nanosegundo.

us = microsegundo

10^{-6}	micro	µ	0,000 001
10^{-9}	nano	n	0,000 000 001

Desenvolvedor B

```
1 def lista2():
2     return range(1000)
] ✓ 0.0s
```

```
1 l = lista2()
] ✓ 0.0s
```

Desenvolvedor B



```
1 %timeit lista2()
```

✓ 9.3s

Python

```
117 ns ± 9.47 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

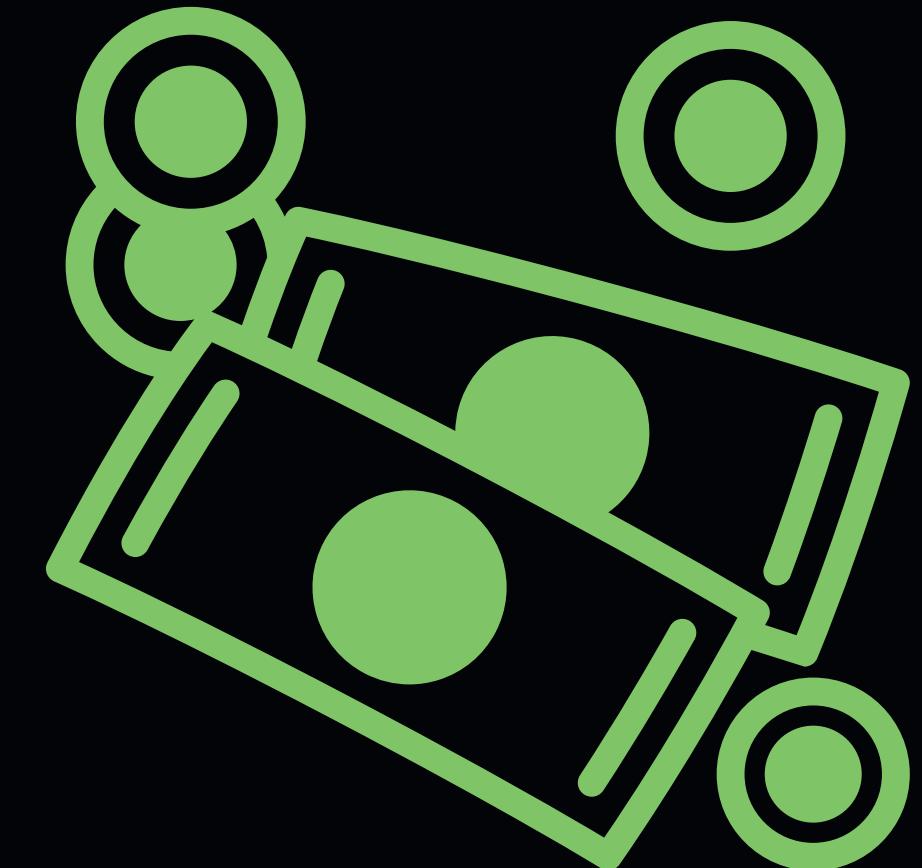
ns = nanosegundo.

us = microsegundo

10^{-6}	micro	μ	0.000 001
10^{-9}	nano	n	0.000 000 001

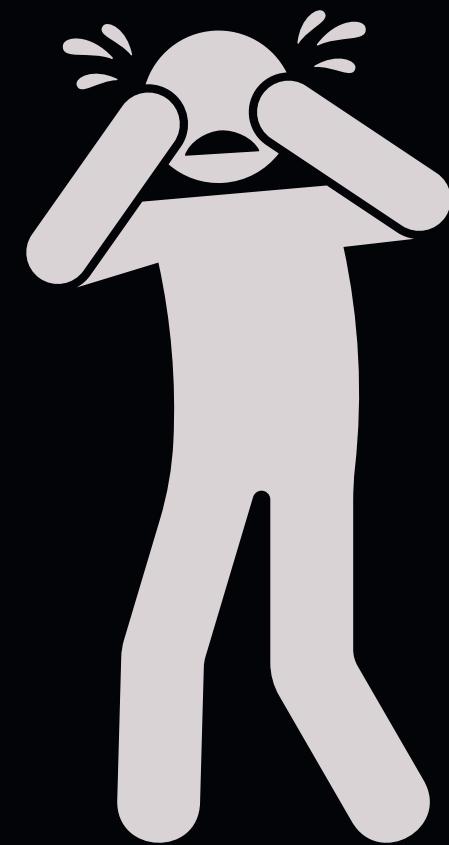
Resultado...R\$

Desenvolvedor **B** ganhou novamente!



Resultado...R\$

Chorou quem apostou na revanche do Dev A



Resumindo

Na primeira função o Big-O foi **O(n)**, onde o n foi a quantidade de passos dados para execução completa dela, ficando **O(1001)**. Onde o primeiro passo seria setar a variável lista = [] (vazia) e os outros 1000 passos estariam diretamente ligados ao número passado que iria iterando com a estrutura escolhida pelo dev A que foi a **for**.

Na segunda função o Bi função o Big-O foi **O(n)**, onde o n foi a quantidade de passos dados para execução completa dela, ficando **O(1)**. Onde o primeiro e único passo foi usar a função **range()** para atribuir os 1.000 números.



Funções Big-O

Alternative Big O notation:

$O(1)$ = O(yeah)

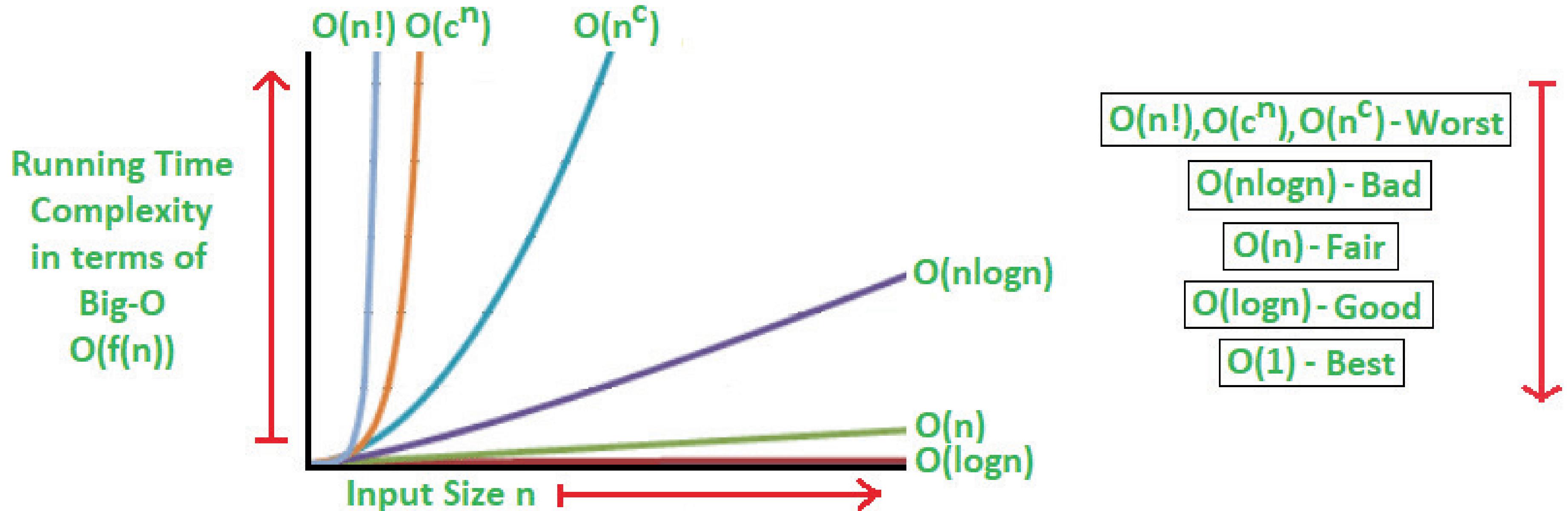
$O(\log n)$ = O(nice)

$O(n)$ = O(ok)

$O(n^2)$ = O(my)

$O(2^n)$ = O(no)

$O(n!)$ = O(mg!)



Conceituando

A complexidade $O(1)$ (constante) é aquela em que não há crescimento do número de operações, pois não depende do volume de dados de entrada (n). Por exemplo: o acesso direto a um elemento de uma matriz.

A complexidade $O(\log n)$ (logaritmo) é aquela em que o crescimento do número de operações é menor do que o do número de itens. Exemplo: caso médio do algoritmo de busca em árvores binárias ordenadas.

A complexidade $O(n)$ (linear) é aquela em que o crescimento no número de operações é diretamente proporcional ao crescimento do número de itens. Por exemplo: o algoritmo de busca em uma lista/vetor.

Conceituando

A complexidade $O(n \log n)$ (linearitmica ou quasilinear) é aquela em que é resultado das operações ($\log n$) executada n vezes. Exemplo: o caso médio do algoritmo de ordenação Quicksort.

A complexidade $O(n^2)$ (quadrático) é aquela que ocorre quando os itens de dados são processados aos pares, muitas vezes com repetições dentro da outra. Com dados suficientemente grandes, tendem a se tornar muito ruim. Por exemplo: o processamento de itens de uma matriz bidimensional.

A complexidade $O(2^n)$ (exponencial) é aquela em que a medida que n aumenta, o fator analisado (tempo ou espaço) aumenta exponencialmente. Não é executável para valores muito grandes e não são úteis do ponto de vista prático. Exemplo: busca em uma árvore binária não ordenada.

Conceituando

A complexidade $O(n!)$ (fatorial) é aquela em que o número de instruções executadas cresce muito rapidamente para um pequeno número de dados. Por exemplo: um algoritmo que gere todas as possíveis permutações de uma lista.



Vejamos alguns exemplos



```
1 from math import log  
2 import numpy as np  
3 import matplotlib.pyplot as plt  
4  
5 n = np.linspace(1, 10, 100)
```

✓ 1.1s

```
1 len(n)
```

✓ 0.0s

100

Gera números espaçados
linspace(de, até, qts
números)

Logo... de 1 até 10 e 100
números. Ele vai gerar 100
números entre 1 e 10 esses
números estarão espaçados
da mesma forma.



[17]

...

1 len(n)

✓ 0.0s

100

Confirmado! Ele gerou os 100
números!

Vamos agora ver quais números temos dentro de **n**.

```
1 n
✓ 0.0s Python
array([[ 1.          ,  1.09090909,  1.18181818,  1.27272727,  1.36363636,
       1.45454545,  1.54545455,  1.63636364,  1.72727273,  1.81818182,
       1.90909091,  2.          ,  2.09090909,  2.18181818,  2.27272727,
       2.36363636,  2.45454545,  2.54545455,  2.63636364,  2.72727273,
       2.81818182,  2.90909091,  3.          ,  3.09090909,  3.18181818,
       3.27272727,  3.36363636,  3.45454545,  3.54545455,  3.63636364,
       3.72727273,  3.81818182,  3.90909091,  4.          ,  4.09090909,
       4.18181818,  4.27272727,  4.36363636,  4.45454545,  4.54545455,
       4.63636364,  4.72727273,  4.81818182,  4.90909091,  5.          ,
       5.09090909,  5.18181818,  5.27272727,  5.36363636,  5.45454545,
       5.54545455,  5.63636364,  5.72727273,  5.81818182,  5.90909091,
       6.          ,  6.09090909,  6.18181818,  6.27272727,  6.36363636,
       6.45454545,  6.54545455,  6.63636364,  6.72727273,  6.81818182,
       6.90909091,  7.          ,  7.09090909,  7.18181818,  7.27272727,
       7.36363636,  7.45454545,  7.54545455,  7.63636364,  7.72727273,
       7.81818182,  7.90909091,  8.          ,  8.09090909,  8.18181818,
       8.27272727,  8.36363636,  8.45454545,  8.54545455,  8.63636364,
       8.72727273,  8.81818182,  8.90909091,  9.          ,  9.09090909,
       9.18181818,  9.27272727,  9.36363636,  9.45454545,  9.54545455,
       9.63636364,  9.72727273,  9.81818182,  9.90909091,  10.        ])
```

+ Code + Markdown

Bora dar uma brincada!

```
1 np.ones(2)
```

```
✓ 0.0s
```

```
array([1., 1.])
```

Cria (dois) números 1

```
1 np.ones(3)
```

```
✓ 0.0s
```

```
array([1., 1., 1.])
```

Cria (três) números 1



```
1 labels = [
2     "Constant",
3     "Logarithmic",
4     "Linear",
5     "Log linear",
6     "Quadratic"
7     "Cubic",
8     "Exponential",
9 ]
10 big_o = [np.ones(n.shape), np.log(n), n, n * np.log(n), n**2,
n**3, 2**n]
```

6] ✓ 0.0s Pytl

Foram criadas duas listas distintas de exemplos para os tipos de notação Big-O serem melhores visualizados quando formos gerar gráficos de tempo de execução.

```
1 np.ones(n.shape) # Constant
```

Uma matriz de n números 1.
`shape()` mostra o tamanho da matriz (linhas x colunas)

≡ ⌂ ⌃ ⌄ ⌅ ⌆

▷
 1 np.log(n) # Logarithmic

[22] ✓ 0.0s

Python

```
... array([0.          , 0.08701138, 0.16705408, 0.24116206, 0.31015493,
       0.37469345, 0.43531807, 0.49247649, 0.54654371, 0.597837  ,
       0.64662716, 0.69314718, 0.73759894, 0.78015856, 0.82098055,
       0.86020127, 0.89794159, 0.93430924, 0.96940056, 1.00330211,
       1.03609193, 1.06784063, 1.09861229, 1.12846525, 1.15745279,
       1.18562367, 1.21302264, 1.23969089, 1.26566637, 1.29098418,
       1.31567679, 1.33977435, 1.36330484, 1.38629436, 1.40876722,
       1.43074612, 1.45225233, 1.47330574, 1.49392503, 1.51412773,
       1.53393036, 1.55334845, 1.57239664, 1.59108877, 1.60943791,
       1.62745642, 1.645156  , 1.66254774, 1.67964217, 1.69644929,
       1.71297859, 1.72923911, 1.74523945, 1.76098781, 1.776492  ,
       1.79175947, 1.80679735, 1.82161243, 1.83621123, 1.85059997,
       1.8647846 , 1.87877085, 1.89256417, 1.90616982, 1.91959284,
       1.93283807, 1.94591015, 1.95881355, 1.97155258, 1.98413136,
       1.99655388, 2.00882397, 2.02094533, 2.03292153, 2.04475598,
       2.05645202, 2.06801285, 2.07944154, 2.0907411 , 2.1019144 ,
       2.11296423, 2.1238933 , 2.13470422, 2.14539951, 2.15598162,
       2.16645292, 2.17681571, 2.18707221, 2.19722458, 2.20727491,
       2.21722524, 2.22707754, 2.23683372, 2.24649563, 2.25606508,
       2.26554382, 2.27493356, 2.28423595, 2.29345261, 2.30258509])
```

```
1 n
```

✓ 0.0s

Pyth

```
array([ 1.          ,  1.09090909,  1.18181818,  1.27272727,  1.36363636,
       1.45454545,  1.54545455,  1.63636364,  1.72727273,  1.81818182,
       1.90909091,  2.          ,  2.09090909,  2.18181818,  2.27272727,
       2.36363636,  2.45454545,  2.54545455,  2.63636364,  2.72727273,
       2.81818182,  2.90909091,  3.          ,  3.09090909,  3.18181818,
       3.27272727,  3.36363636,  3.45454545,  3.54545455,  3.63636364,
       3.72727273,  3.81818182,  3.90909091,  4.          ,  4.09090909,
       4.18181818,  4.27272727,  4.36363636,  4.45454545,  4.54545455,
       4.63636364,  4.72727273,  4.81818182,  4.90909091,  5.          ,
       5.09090909,  5.18181818,  5.27272727,  5.36363636,  5.45454545,
       5.54545455,  5.63636364,  5.72727273,  5.81818182,  5.90909091,
       6.          ,  6.09090909,  6.18181818,  6.27272727,  6.36363636,
       6.45454545,  6.54545455,  6.63636364,  6.72727273,  6.81818182,
       6.90909091,  7.          ,  7.09090909,  7.18181818,  7.27272727,
       7.36363636,  7.45454545,  7.54545455,  7.63636364,  7.72727273,
       7.81818182,  7.90909091,  8.          ,  8.09090909,  8.18181818,
       8.27272727,  8.36363636,  8.45454545,  8.54545455,  8.63636364,
       8.72727273,  8.81818182,  8.90909091,  9.          ,  9.09090909,
       9.18181818,  9.27272727,  9.36363636,  9.45454545,  9.54545455,
       9.63636364,  9.72727273,  9.81818182,  9.90909091, 10.        ])
```

array n original

```
1 n # Linear
```

✓ 0.0s

Pyth

```
array([ 1.          ,  1.09090909,  1.18181818,  1.27272727,  1.36363636,
       1.45454545,  1.54545455,  1.63636364,  1.72727273,  1.81818182,
       1.90909091,  2.          ,  2.09090909,  2.18181818,  2.27272727,
       2.36363636,  2.45454545,  2.54545455,  2.63636364,  2.72727273,
       2.81818182,  2.90909091,  3.          ,  3.09090909,  3.18181818,
       3.27272727,  3.36363636,  3.45454545,  3.54545455,  3.63636364,
       3.72727273,  3.81818182,  3.90909091,  4.          ,  4.09090909,
       4.18181818,  4.27272727,  4.36363636,  4.45454545,  4.54545455,
       4.63636364,  4.72727273,  4.81818182,  4.90909091,  5.          ,
       5.09090909,  5.18181818,  5.27272727,  5.36363636,  5.45454545,
       5.54545455,  5.63636364,  5.72727273,  5.81818182,  5.90909091,
       6.          ,  6.09090909,  6.18181818,  6.27272727,  6.36363636,
       6.45454545,  6.54545455,  6.63636364,  6.72727273,  6.81818182,
       6.90909091,  7.          ,  7.09090909,  7.18181818,  7.27272727,
       7.36363636,  7.45454545,  7.54545455,  7.63636364,  7.72727273,
       7.81818182,  7.90909091,  8.          ,  8.09090909,  8.18181818,
       8.27272727,  8.36363636,  8.45454545,  8.54545455,  8.63636364,
       8.72727273,  8.81818182,  8.90909091,  9.          ,  9.09090909,
       9.18181818,  9.27272727,  9.36363636,  9.45454545,  9.54545455,
       9.63636364,  9.72727273,  9.81818182,  9.90909091, 10.        ])
```

```
1 n * np.log(n) # Log linear
```

✓ 0.0s

Python

```
array([ 0.          ,  0.0949215 ,  0.19742755,  0.30693353,  0.42293854,
       0.54500865,  0.67276429,  0.80587061,  0.94403004,  1.08697637,
       1.23447004,  1.38629436,  1.54225234,  1.70216413,  1.86586489,
       2.03320299,  2.20403846,  2.3782417 ,  2.55569238,  2.73627848,
       2.91989544,  3.10644547,  3.29583687,  3.48798351,  3.68280433,
       3.88022291,  4.08016706,  4.28256852,  4.4873626 ,  4.69448793,
       4.90388623,  5.11550205,  5.32928257,  5.54517744,  5.76313861,
       5.98312015,  6.20507813,  6.42897049,  6.65475693,  6.88239878,
       7.11185894,  7.34310174,  7.57609291,  7.81079943,  8.04718956,
       8.28523267,  8.52489925,  8.7661608 ,  9.00898983,  9.25335976,
       9.49924492,  9.74662045,  9.99546233,  10.24574726, 10.49745271,
      10.75055682, 11.00503838, 11.26087685, 11.51805227, 11.77654526,
      12.03633699, 12.29740918, 12.55974403, 12.82332425, 13.08813301,
      13.35415392, 13.62137104, 13.88976884, 14.15933216, 14.43004627,
      14.70189677, 14.97486963, 15.24895116, 15.52412802, 15.80038715,
      16.07771582, 16.3561016 , 16.63553233, 16.91599615, 17.19748143,
      17.47997684, 17.76347127, 18.04795386, 18.33341399, 18.61984125,
      18.90722547, 19.19555668, 19.48482511, 19.7750212 , 20.06613557,
      20.35815906, 20.65108265, 20.94489752, 21.23959501, 21.53516665,
      21.8316041 , 22.12889919, 22.42704392, 22.7260304 , 23.02585093])
```

```
1 n**2 # Quadratic
```

```
✓ 0.0s
```

```
array([[ 1.          ,  1.19008264,  1.39669421,  1.61983471,
        1.85950413,  2.11570248,  2.38842975,  2.67768595,
        2.98347107,  3.30578512,  3.6446281 ,  4.          ,
        4.37190083,  4.76033058,  5.16528926,  5.58677686,
        6.02479339,  6.47933884,  6.95041322,  7.43801653,
        7.94214876,  8.46280992,  9.          ,  9.55371901,
       10.12396694, 10.7107438 , 11.31404959, 11.9338843 ,
       12.57024793, 13.2231405 , 13.89256198, 14.5785124 ,
       15.28099174, 16.          , 16.73553719, 17.48760331,
       18.25619835, 19.04132231, 19.84297521, 20.66115702,
       21.49586777, 22.34710744, 23.21487603, 24.09917355,
       25.          , 25.91735537, 26.85123967, 27.80165289,
       28.76859504, 29.75206612, 30.75206612, 31.76859504,
       32.80165289, 33.85123967, 34.91735537, 36.          ,
       37.09917355, 38.21487603, 39.34710744, 40.49586777,
       41.66115702, 42.84297521, 44.04132231, 45.25619835,
       46.48760331, 47.73553719, 49.          , 50.28099174,
       51.5785124 , 52.89256198, 54.2231405 , 55.57024793,
       56.9338843 , 58.31404959, 59.7107438 , 61.12396694,
       62.55371901, 64.          , 65.46280992, 66.94214876,
       68.43801653, 69.95041322, 71.47933884, 73.02479339,
       74.58677686, 76.16528926, 77.76033058, 79.37190083,
       81.          , 82.6446281 , 84.30578512, 85.98347107,
       87.67768595, 89.38842975, 91.11570248, 92.85950413,
       94.61983471, 96.39669421, 98.19008264, 100.        ])
```

▷ ▾

1 n**3 # Cubic

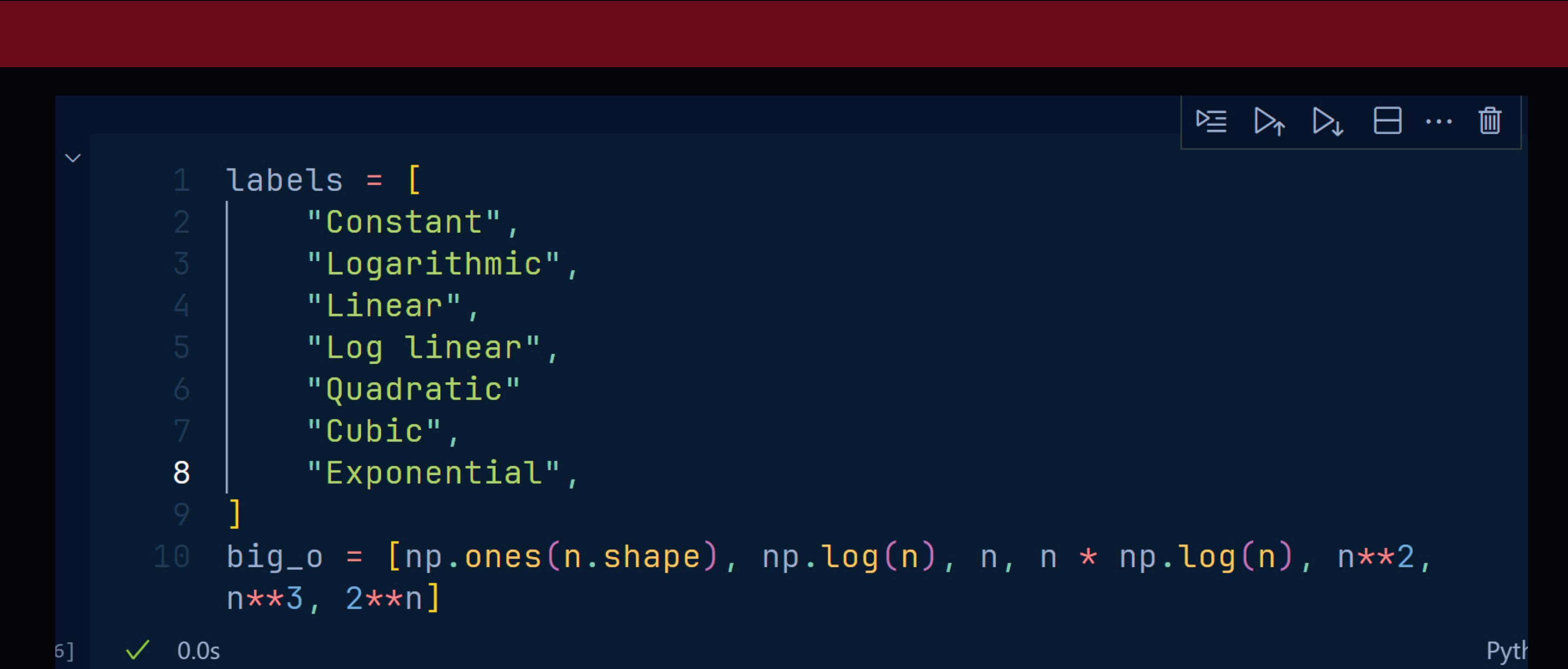
[50] ✓ 0.0s

```
... array([ 1.          ,  1.29827198,  1.65063862,  2.06160781,
       2.53568745,  3.07738542,  3.69120962,  4.38166792,
       5.15326822,  6.01051841,  6.95792637,  8.          ,
       9.14124718,  10.38617581, 11.73929376, 13.20510894,
      14.78812923, 16.49286251, 18.32381668, 20.28549962,
      22.38241923, 24.6190834 , 27.          , 29.52967693,
      32.21262209, 35.05334335, 38.05634861, 41.22614576,
      44.56724267, 48.08414726, 51.78136739, 55.66341097,
      59.73478588, 64.          , 68.46356123, 73.12997746,
      78.00375657, 83.08940646, 88.39143501, 93.91435011,
      99.66265965, 105.64087153, 111.85349361, 118.30503381,
     125.          , 131.94290008, 139.13824192, 146.59053343,
     154.30428249, 162.28399699, 170.53418482, 179.05935387,
     187.86401202, 196.95266717, 206.3298272 , 216.          ,
     225.96769346, 236.23741548, 246.81367393, 257.70097671,
     268.90383171, 280.42674681, 292.2742299 , 304.45078888,
     316.96093163, 329.80916604, 343.          , 356.5379414 ,
     370.42749812, 384.67317806, 399.27948911, 414.25093914,
     429.59203606, 445.30728775, 461.4012021 , 477.878287 ,
     494.74305034, 512.          , 529.65364388, 547.70848986,
     566.16904583, 585.03981968, 604.32531931, 624.03005259,
     644.15852742, 664.71525169, 685.70473328, 707.13148009,
     729.          , 751.3148009 , 774.08039068, 797.30127724,
     820.98196844, 845.1269722 , 869.74079639, 894.82794891,
     920.39293764, 946.44027047, 972.9744553 , 1000.        ])
```

```
1 2**n # Exponential
```

```
✓ 0.0s
```

```
array([ 2.          ,  2.13008218,  2.26862504,  2.41617889,
       2.5733298 ,  2.74070197,  2.91896021,  3.10881256,
       3.31101312,  3.52636502,  3.75572364,  4.          ,
       4.26016436,  4.53725009,  4.83235778,  5.14665959,
       5.48140394,  5.83792042,  6.21762513,  6.62202624,
       7.05273004,  7.51144729,  8.          ,  8.52032872,
       9.07450018,  9.66471556, 10.29331918, 10.96280788,
      11.67584085, 12.43525025, 13.24405248, 14.10546008,
      15.02289457, 16.          , 17.04065743, 18.14900036,
      19.32943111, 20.58663837, 21.92561576, 23.35168169,
      24.87050051, 26.48810496, 28.21092016, 30.04578914,
      32.          , 34.08131486, 36.29800071, 38.65886222,
      41.17327674, 43.85123151, 46.70336338, 49.74100102,
      52.97620991, 56.42184032, 60.09157828, 64.          ,
      68.16262972, 72.59600142, 77.31772444, 82.34655347,
      87.70246302, 93.40672676, 99.48200203, 105.95241983,
     112.84368064, 120.18315656, 128.          , 136.32525945,
     145.19200284, 154.63544888, 164.69310695, 175.40492604,
     186.81345353, 198.96400407, 211.90483965, 225.68736128,
     240.36631313, 256.          , 272.6505189 , 290.38400568,
     309.27089777, 329.38621389, 350.80985209, 373.62690705,
     397.92800813, 423.8096793 , 451.37472256, 480.73262626,
     512.          , 545.30103779, 580.76801136, 618.54179554,
     658.77242778, 701.61970418, 747.25381411, 795.85601627,
     847.6193586 , 902.74944512, 961.46525252, 1024.        ])
```



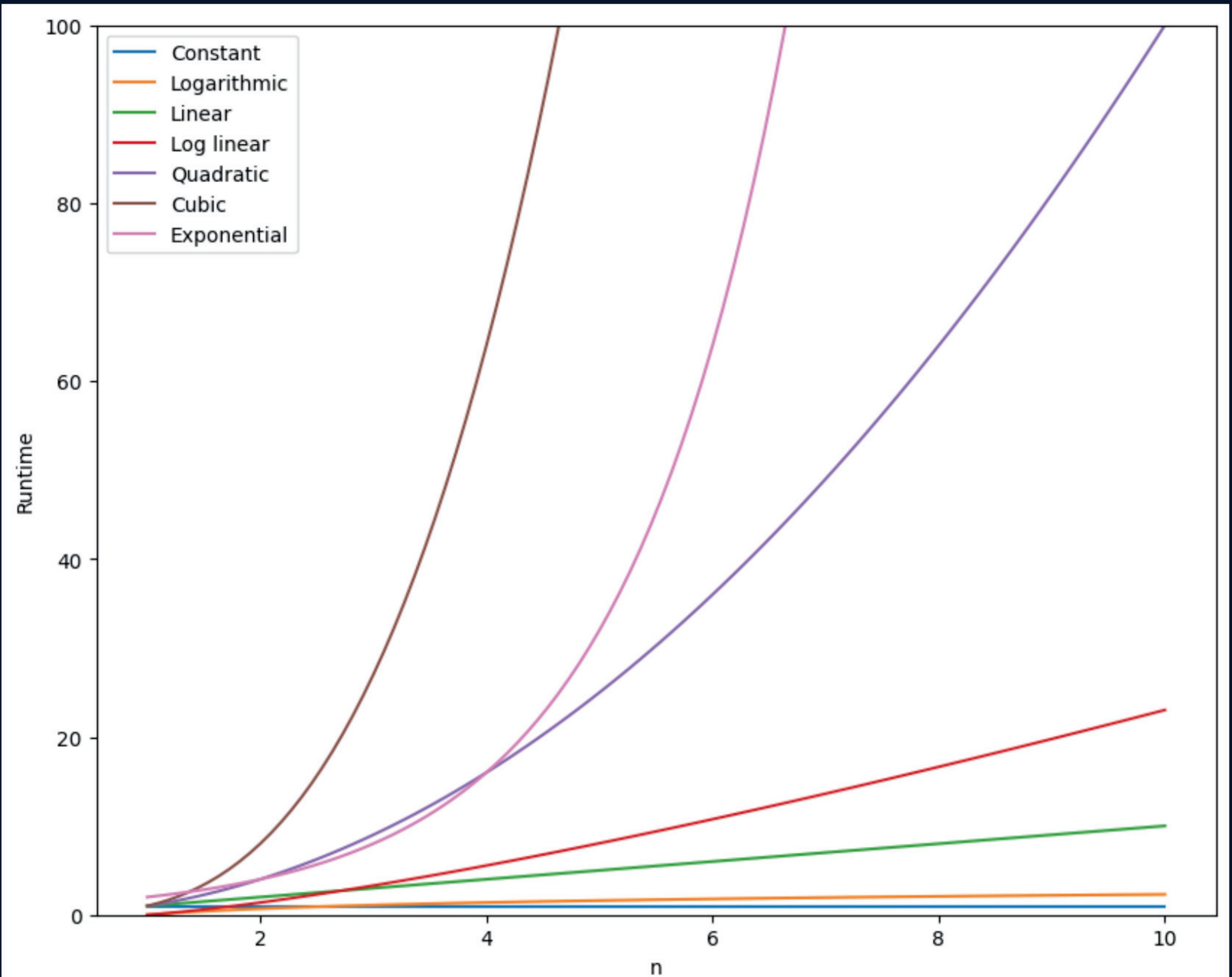
```
1 labels = [
2     "Constant",
3     "Logarithmic",
4     "Linear",
5     "Log linear",
6     "Quadratic"
7     "Cubic",
8     "Exponential",
9 ]
10 big_o = [np.ones(n.shape), np.log(n), n, n * np.log(n), n**2,
n**3, 2**n]
```

6] ✓ 0.0s Pyth

Foram criadas duas listas distintas de exemplos para os tipos de notação Big-O serem melhores visualizados quando formos gerar gráficos de tempo de execução.

Gerando os gráficos

```
1 plt.figure(figsize=(10, 8)) # tamanho da figura
2 plt.ylim(0, 100) #limite do eixo y do gráfico
3 for i in range(len(big_o)): #plotar todas as linhas num mesmo
gráfico
4     plt.plot(n, big_o[i], label=labels[i]) #n para os dados, os
valores big_o para tempo de execução e os labels para os
nomes dos tipos de big-O
5 plt.legend() #para termos legenda
6 plt.ylabel("Runtime") #os dados dos gráficos ficarão no eixo y.
Runtime é tempo de execução.
7 plt.xlabel("n") #n é o numero de entradas que passamos para o
algoritmo.
```



Melhores casos (do melhor para o “menos” melhor)

1. Constant
2. Logarithmic
3. Linear
4. Log Linear

Piores casos (do pior para o “menos” pior)

1. Cubic
2. Exponential
3. Quadratic

Curiosidade: tanto a quadrática quanto a cúbica são chamadas também de polinomiais

```
1 plt.figure(figsize=(10, 8))
2 plt.ylim(0, 100)
3 for i in range(len(big_o)):
4     plt.plot(n, big_o[i], label=labels[i])
5 plt.legend()
6 plt.ylabel("Runtime")
7 plt.xlabel("n")
```

✓ 0.2s

Gerando os gráficos

Vejamos outros exemplos de funções:

- Constant - $O(1)$
- Linear - $O(n)$
- Quadratic - $O(n^2)$ - polynomial
- Combination



Constant - O(1)

Um único passo.

```
1 lista = [1, 2, 3, 4, 5]
```

[3] ✓ 0.0s

```
1 def constant(n):  
2 | print(n[0])  
3
```

[2] ✓ 0.0s

```
1 constant(lista)
```

[4] ✓ 0.0s

• 1

Linear - O(n)

```
1 lista = [1, 2, 3, 4, 5]  
✓ 0.0s
```

Como a lista possui 5 elementos... o **for** vai rodar 5x.

Será então uma função O(n)... sendo o n = 5

```
1 def linear(n):  
2     for i in n:  
3         print(i)  
36] ✓ 0.0s
```

```
> v  
1 linear(lista)  
37] ✓ 0.0s  
.. 1  
   2  
   3  
   4  
   5
```

Quadratic - $O(n^2)$ - polynomial

O n será percorrido 2x...
sendo um loop dentro de
outro loop.

Nesse caso ele mostra o
número 1 e corre todos os
elementos da lista, depois
mostra 2 e corre todos os
elementos e assim por
diante.

```
1 def quadratic(n):  
2     for i in n:  
3         for j in n:  
4             print(i, j)  
5             print("___")
```

✓ 0.0s

Quadratic - $O(n^2)$ - polynomial

Assim fica melhor de
visualizar.



```
1 quadratic(lista)
```

```
✓ 0.0s
```

```
1 1  
1 2  
1 3
```

```
1 4  
1 5
```

```
---
```

```
2 1
```

```
2 2
```

```
2 3
```

```
2 4
```

```
2 5
```

```
---
```

```
3 1
```

```
3 2
```

```
3 3
```

```
3 4
```

```
3 5
```

```
---
```

```
4 1
```

```
4 2
```

```
4 3
```

```
4 4
```

```
4 5
```

```
---
```

```
5 1
```

```
...
```

Combination

Devemos nesse caso quebrar o código parte por parte individualmente.
Ao final devemos **somar!**

O resultado **n** pode variar até o infinito... então $2x$ infinito será sempre infinito, logo neste caso podemos ignorar as constantes pois elas já estão dentro do infinito. Portanto a complexidade deste algoritmo diz-se $O(n)$

```
1 # O(1) + O(5) + O(n) + O(n) + O(3)
2 # O(9) + O(2n) → O(n)
3
4
5 def combination(n):
6     # O(1)
7     print(n[0])
8
9     # O(5)
10    for i in range(5):
11        print("test ", i) Constante
12
13    # O(n)
14    for i in n:
15        print(i) n
16
17    # O(n)
18    for i in n:
19        print(i) n
20
21    # O(3)
22    print("Python")
23    print("Python")
24    print("Python") Constante
```

Combination

```
1 combination(lista)
```

```
✓ 0.0s
```

```
1  
test 0  
test 1  
test 2  
test 3  
test 4
```

```
1  
2  
3  
4  
5  
1  
2  
3  
4  
5
```

```
Python  
Python  
Python
```

Quiz time!

Para que é utilizada a notação Big-O?

- Para aumentar a velocidade de execução de algoritmos utilizando recursos de Inteligência Artificial
- Para objetivamente comparar algoritmos
- Para aplicar recursos de Deep Learning em algoritmos inteligentes
- Para diminuir o tempo de execução de algoritmos

Quiz time!

Para que é utilizada a notação Big-O?

Para aumentar a velocidade de execução de algoritmos utilizando recursos de Inteligência Artificial

Para objetivamente comparar algoritmos

Para aplicar recursos de Deep Learning em algoritmos inteligentes

Para diminuir o tempo de execução de algoritmos

Quiz time!

Qual é o Big-O do seguinte trecho de código?

```
1 | for a in range(n):  
2 |     for b in range(n):  
3 |         print('test')
```

O(n^2)

O(1)

O(log(n))

O(2)

Quiz time!

Qual é o Big-O do seguinte trecho de código?

```
1 | for a in range(n):  
2 |     for b in range(n):  
3 |         print('test')
```

O(n^2)

O(1)

O(log(n))

O(2)

Quiz time!

Qual é o Big-O para o seguinte trecho de código?

```
1 | for i in range(n):  
2 |     print(i)
```

O(**n**)

O(**1**)

O(**n²**)

O(**n³**)

Quiz time!

Qual é o Big-O para o seguinte trecho de código?

```
1 | for i in range(n):  
2 |     print(i)
```

O(**n**)

O(**1**)

O(**n²**)

O(**n³**)

Quiz time!

Qual é o Big-O do seguinte trecho de código?

```
1 | l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 | for i in range(l):
3 |     print(l[0])
4 |     break
```

O(n)

O(n^10)

O(10)

O(1)

Quiz time!

Qual é o Big-O do seguinte trecho de código?

```
1 | l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 | for i in range(1):
3 |     print(l[0])
4 |     break
```

O(n)

O(n^{10})

O(10)

O(1)

Quiz time!

Dado o trecho de código abaixo

```
1 | for i in range(5):  
2 |     print(i)
```

Qual é o tipo de função Big-O?

Quadrática

Constante

Polinomial

Exponencial

Quiz time!

Dado o trecho de código abaixo

```
1 | for i in range(5):  
2 |     print(i)
```

Qual é o tipo de função Big-O?

Quadrática

Constante

Polinomial

Exponencial

That's all Folks!

ATÉ A PRÓXIMA!