

Aula 02

Complexidade de Algoritmos



Algoritmos e Estrutura de Dados II

2º Semestre – CDN



Prof. Dr. Dilermando Piva Jr.

Conteúdo Programático - Planejamento

Semana	Data	Temas/Atividades
1	07/08	Acolhimento e Boas-vindas! Introdução a Disciplina. Formas de Avaliação e Percurso Pedagógico.
2	14/08	Tipo de dado abstrato. Introdução a Estrutura de Dados.
3	21/08	Complexidade de Algoritmos
4	28/08	Vetores não-Ordenados e busca sequencial
5	04/09	Vetores Ordenados e busca binária
6	11/09	Revisão de Programação Orientada a Objetos (POO)
7	18/09	Pilhas
8	25/09	Filas
9	02/10	Listas encadeadas
10	09/10	Recursão
11	16/10	Primeira Avaliação Formal. (P1). Correção da Avaliação após o intervalo.
12	18/10	Algoritmos de Ordenação
13	23/10	Algoritmos de Ordenação
14	30/10	Árvores
15	06/11	Grafos
16	13/11	Segunda Avaliação Formal (P2). Correção da Avaliação após o intervalo
17	27/11	Apresentação PI do curso de CDN
18	04/12	Tabela Hash (tabela de espalhamento) – Tópico extra.
19	11/12	Exame / Avaliação Substitutiva. Correção da Avaliação após o intervalo. Finalização Disciplina
20	18/12	Finalização da disciplina.

Cenário... **LOJA VIRTUAL** ... Milhares de clientes acessam o site simultaneamente



Complexidade de Algoritmos...

Eficiência de Algoritmos?

Como podemos medir?



Complexidade de Algoritmos...

Eficiência de Algoritmos?

Existem muitas formas de resolver o mesmo problema...

Como saber qual resolução é a melhor?



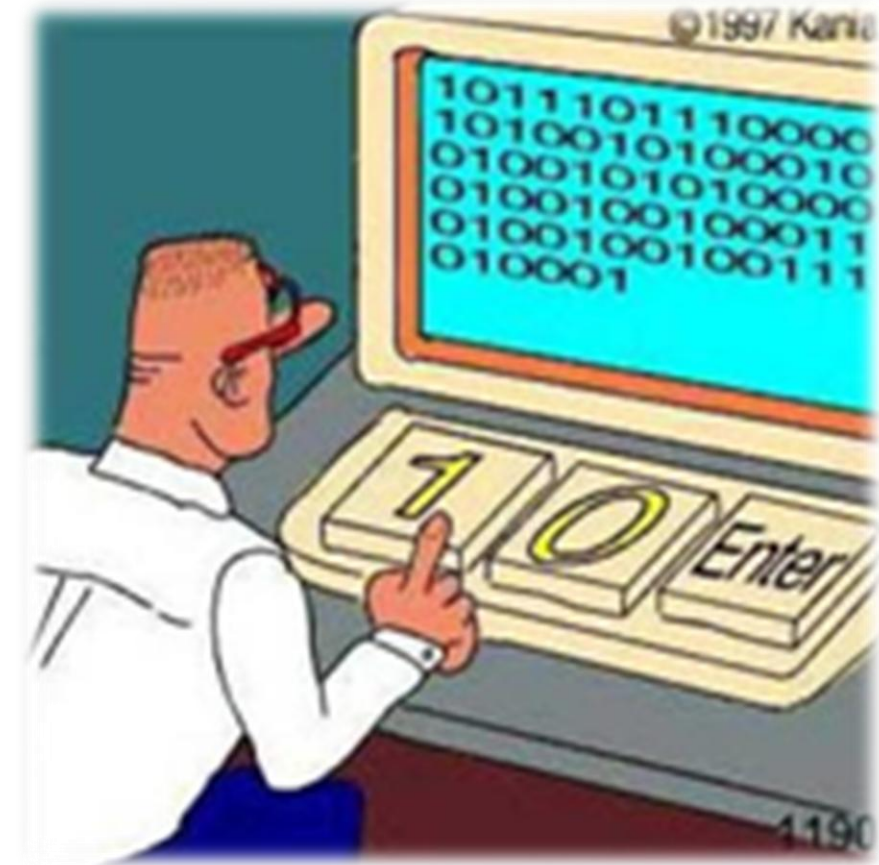
Complexidade de Algoritmos...

Eficiência de Algoritmos?

Como podemos medir?

Tempo

Espaço



Complexidade de Algoritmos...

Eficiência de Algoritmos? Medindo o Tempo...

```
"""
```

```
Arquivo: timing1.py
```

```
Imprime os tempos de execução para tamanhos de problemas que  
dobram, usando um único laço.
```

```
"""
```

```
import time
```

```
problemSize = 10000000
```

```
print("%12s%16s" % ("Problem Size", "Seconds"))
```

```
for count in range(5):
```

```
    start = time.time()
```

```
    # O início do algoritmo
```

```
    work = 1
```

```
    for x in range(problemSize):
```

```
        work += 1
```

```
        work -= 1
```

```
    # O fim do algoritmo
```

```
    elapsed = time.time() - start
```

```
    print("%12d%16.3f" % (problemSize, elapsed))
```

```
    problemSize *= 2
```

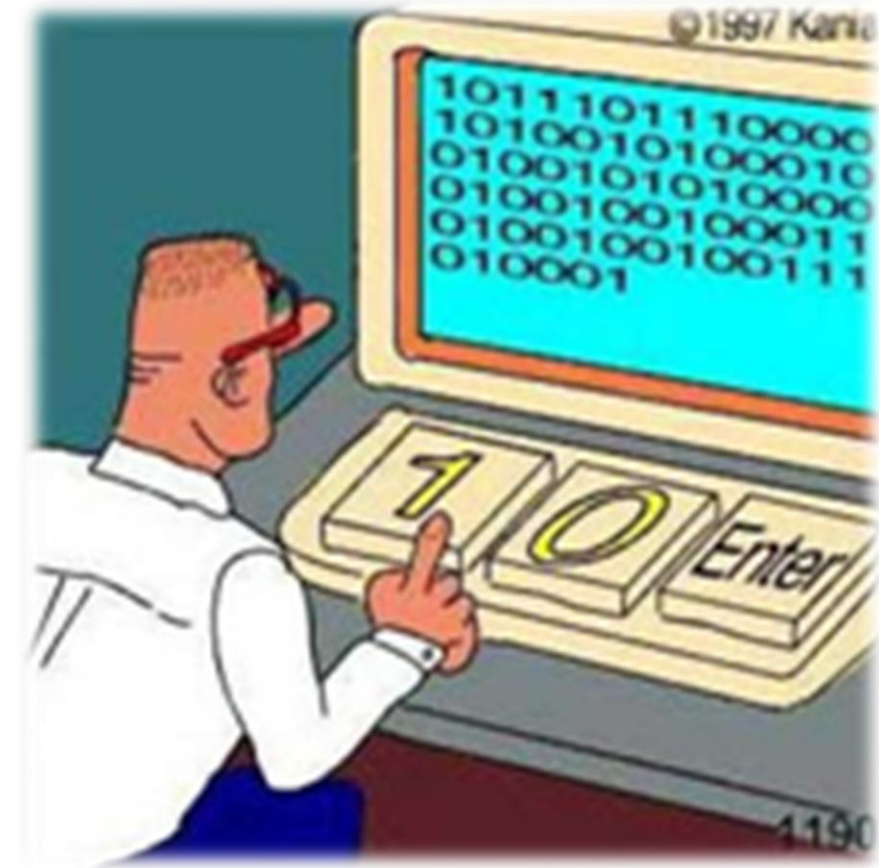


Complexidade de Algoritmos...

Eficiência de Algoritmos? Medindo o Tempo...

A saída do programa de teste

Problem Size	Seconds
10000000	3.8
20000000	7.591
40000000	15.352
80000000	30.697
160000000	61.631



Complexidade de Algoritmos...

Eficiência de Algoritmos? Medindo o Tempo...

- Como outro exemplo, considere a seguinte alteração no algoritmo do programa de teste:

```
for j in range(problemSize):  
    for k in range(problemSize):  
        work += 1  
        work -= 1
```

- As atribuições estendidas foram movidas para um laço aninhado:
 - Esse laço itera pelo tamanho do problema dentro de outro laço que também itera pelo tamanho do problema.
 - os resultados...



Complexidade de Algoritmos...

Eficiência de Algoritmos? Medindo o Tempo...

A saída do segundo programa de teste

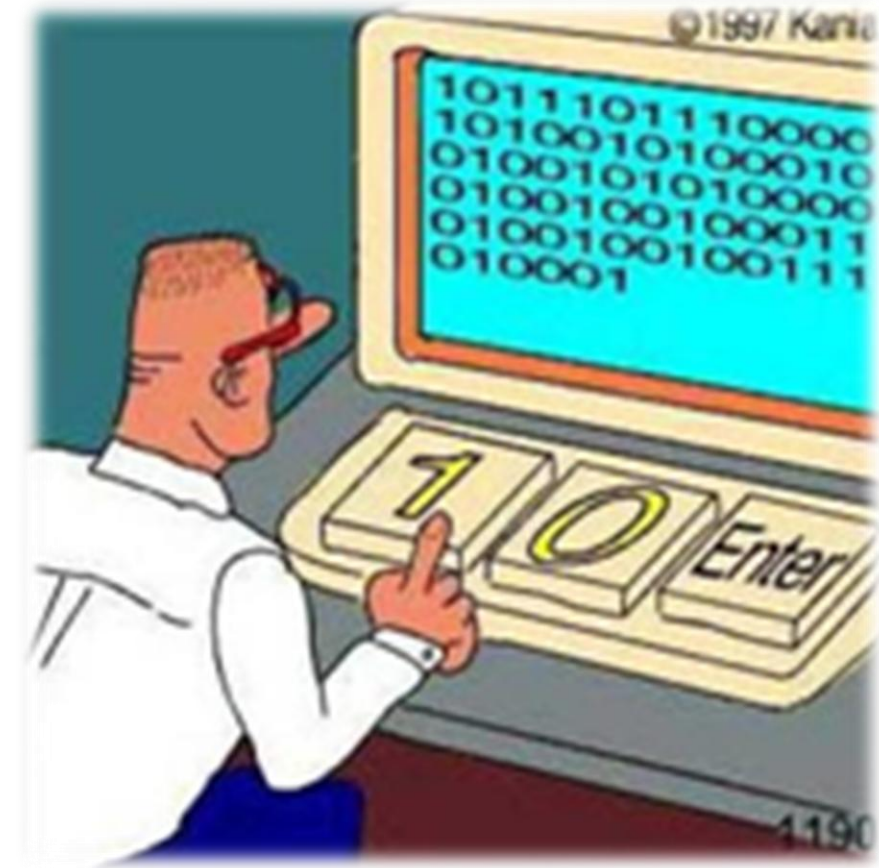
Problem Size	Seconds
1000	0.387
2000	1.581
4000	6.463
8000	25.702
16000	102.666



Complexidade de Algoritmos...

Eficiência de Algoritmos? Medindo o Tempo...

- Esse método permite previsões precisas dos tempos de execução de muitos algoritmos. Entretanto, existem dois problemas principais nessa técnica:
 - **Diferentes plataformas** de hardware têm diferentes velocidades de processamento, portanto, os tempos de execução de um algoritmo variam entre uma máquina e outra.
 - É impraticável determinar o tempo de execução de alguns algoritmos **com conjuntos de dados muito grandes**.



Complexidade de Algoritmos...

Eficiência de Algoritmos? Qtd de Instruções Executadas...

- Outra técnica usada para estimar a eficiência de um algoritmo
 - **Contar as instruções executadas** com problemas de tamanhos diferentes.
- Ao analisar um algoritmo dessa forma, você distingue entre duas classes de instruções:
 - Instruções que executam o mesmo número de vezes, independentemente do tamanho do problema.
 - Instruções cuja contagem de execução varia de acordo com o tamanho do problema.
- As instruções na segunda classe normalmente são encontradas em laços ou funções recursivas:
 - No caso de laços, você também se concentra nas instruções executadas em quaisquer laços aninhados.



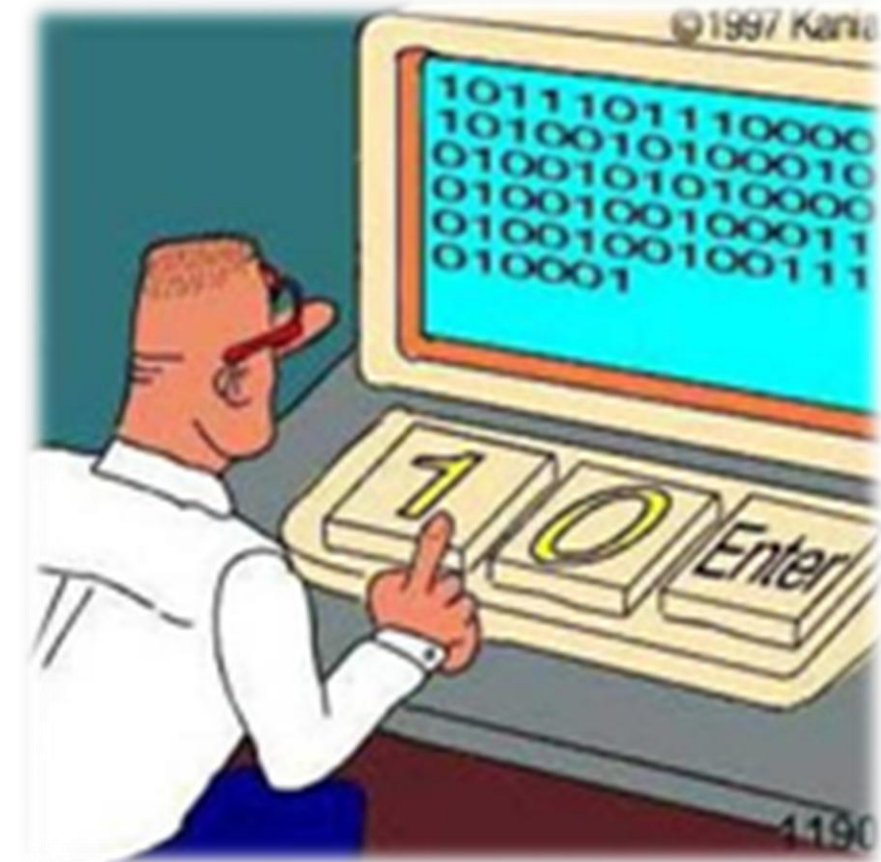
Complexidade de Algoritmos...

Eficiência de Algoritmos? Qtd de Instruções Executadas...

```
"""
Arquivo: counting.py
Imprime o número de iterações para tamanhos de
problema que dobram, usando um laço aninhado.
"""

problemSize = 1000
print("%12s%15s" % ("Problem Size", "Iterations"))
for count in range(5):
    number = 0
    # O início do algoritmo
    work = 1
    for j in range(problemSize):
        for k in range(problemSize):
            number += 1
            work += 1
            work -= 1

    # O fim do algoritmo
    print("%12d%15d" % (problemSize, number))
    problemSize *= 2
```



Complexidade de Algoritmos...

Eficiência de Algoritmos? Qtd de Instruções Executadas...

A saída de um programa de teste que conta iterações

Problem Size	Iterations
1000	1000000
2000	4000000
4000	16000000
8000	64000000
16000	256000000



Complexidade de Algoritmos...

Eficiência de Algoritmos? Qtd de Instruções Executadas...

```
"""
Arquivo: countfib.py
Imprime o número de chamadas de uma função Fibonacci
recursiva com tamanhos de problema que dobram.
"""

from counter import Counter

def fib(n, counter):
    """Conta o número de chamadas da função Fibonacci."""
    counter.increment()
    if n < 3:
        return 1
    else:
        return fib(n - 1, counter) + fib(n - 2, counter)

problemSize = 2
print("%12s%15s" % ("Problem Size", "Calls"))
for count in range(5):
    counter = Counter()
    # O início do algoritmo
    fib(problemSize, counter)
    # O fim do algoritmo
    print("%12d%15s" % (problemSize, counter))
    problemSize *= 2
```



Complexidade de Algoritmos...

Eficiência de Algoritmos? Qtd de Instruções Executadas...

A saída de um programa de teste que executa a função Fibonacci

Problem Size	Calls
2	1
4	5
8	41
16	1973
32	4356617



Complexidade de Algoritmos...

Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE

- Complexidade constante (1)
- Complexidade Logarítmica ($\log n$)
- Complexidade Linear (n)
- Complexidade Quadrática (n^2)
- Complexidade Exponencial (2^n)

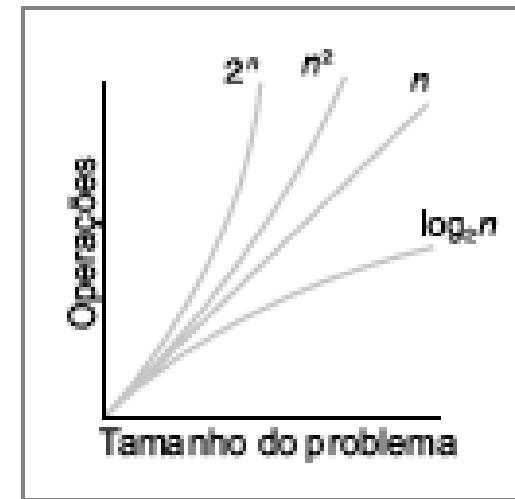


Complexidade de Algoritmos...

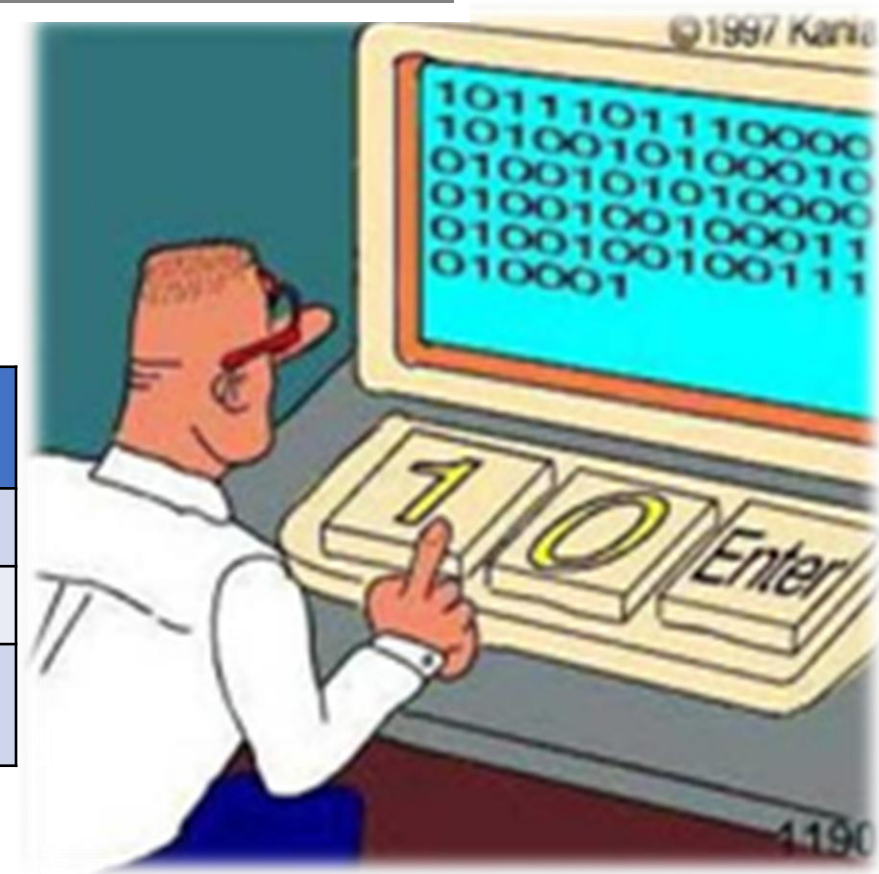
Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE

- Complexidade constante (1)
- Complexidade Logarítmica ($\log n$)
- Complexidade Linear (n)
- Complexidade Quadrática (n^2)
- Complexidade Exponencial (2^n)



n	Logarítmico ($\log_2 n$)	Linear (n)	Quadrático (n^2)	Exponencial (2^n)
100	7	100	10.000	Fora do gráfico
1000	10	1000	1.000.000	Fora do gráfico
1.000.000	20	1.000.000	1.000.000.000.000	Realmente fora do gráfico



ORDENS DE COMPLEXIDADE



Cenário: Análise de Sentimentos em Redes Sociais

Imagine que você trabalha em uma empresa que quer monitorar o que as pessoas estão falando sobre seus produtos nas redes sociais.

O objetivo é analisar milhares de tweets para determinar se o sentimento geral é positivo, negativo ou neutro.

Para fazer isso, você precisa usar algoritmos que ajudem a processar e analisar grandes volumes de dados rapidamente.

Passo 1: Coleta de Dados

Você coleta 1.000.000 de tweets sobre o produto.

Esses tweets precisam ser processados para identificar o sentimento.

ORDENS DE COMPLEXIDADE



Passo 2: Pré-processamento dos Tweets

Você precisa limpar e preparar os tweets, removendo palavras irrelevantes, links e caracteres especiais.

Isso é uma etapa de **pré-processamento**.

Para tanto...

Você usa um script simples para limpar cada tweet, removendo palavras irrelevantes, links e caracteres especiais.

- Você não mede a complexidade de tempo.
- Observação empírica: Percebe que seu script leva, em média, 1 segundo para processar 100 tweets.
- Estimativa empírica: Para 1.000.000 de tweets, leva aproximadamente 10.000 segundos (1 segundo para 100 tweets x 10.000).

ORDENS DE COMPLEXIDADE



Passo 3: Análise de Sentimentos

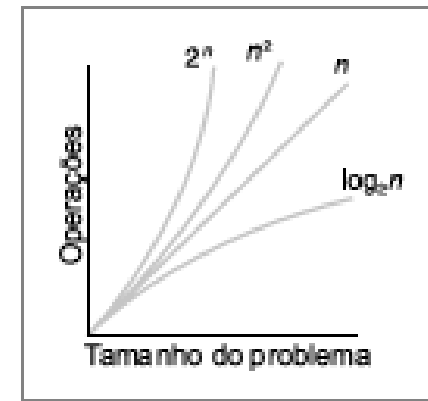
Método 1: Análise com Dicionário de Palavras

- Você cria um dicionário de palavras positivas e negativas.
- Observação empírica: O script leva cerca de 0,1 segundo para analisar cada tweet.
- Estimativa empírica: Para 1.000.000 de tweets, leva aproximadamente 100.000 segundos (0,1 segundo por tweet x 1.000.000).

Método 2: Análise com Machine Learning

- Você usa um modelo de machine learning pré-treinado.
- Observação empírica: O modelo leva cerca de 0,05 segundo para analisar cada tweet.
- Estimativa empírica: Para 1.000.000 de tweets, leva aproximadamente 50.000 segundos (0,05 segundo por tweet x 1.000.000).

ORDENS DE COMPLEXIDADE



Passo 4: Relatório de Resultados

Você gera o relatório com a porcentagem de tweets positivos, negativos e neutros.

Resumo Empírico:

- Pré-processamento: 10.000 segundos
- Análise com Dicionário: 100.000 segundos
- Análise com Machine Learning: 50.000 segundos
- 160.000 segundos = 2.667 minutos = 44 horas =
- quase 2 dias de trabalho ininterrupto.

PURAMENTE EMPÍRICO →

REALIZADO PELA MEDIÇÃO E OBSERVAÇÃO DIRETA

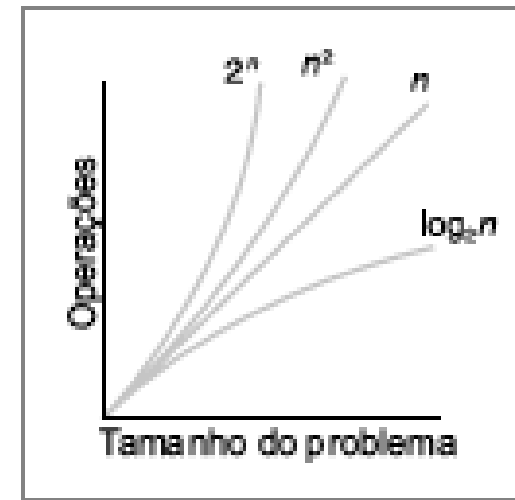
Complexidade de Algoritmos...

Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE

Notação **Big O**

- A notação Big O é utilizada para descrever a complexidade de algoritmos em termos de seu comportamento assintótico.
((uma grandeza física se aproxima de um valor limite a medida que uma variável tende a um valor específico))
- Isso significa que ela se concentra no comportamento do algoritmo quando o tamanho da entrada se torna muito grande.

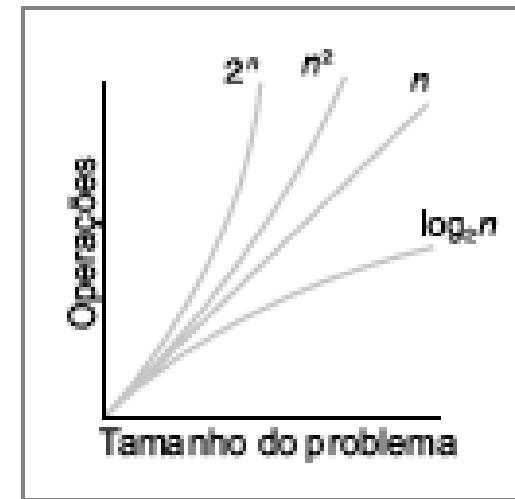


Complexidade de Algoritmos...

Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE

Notação **Big O**



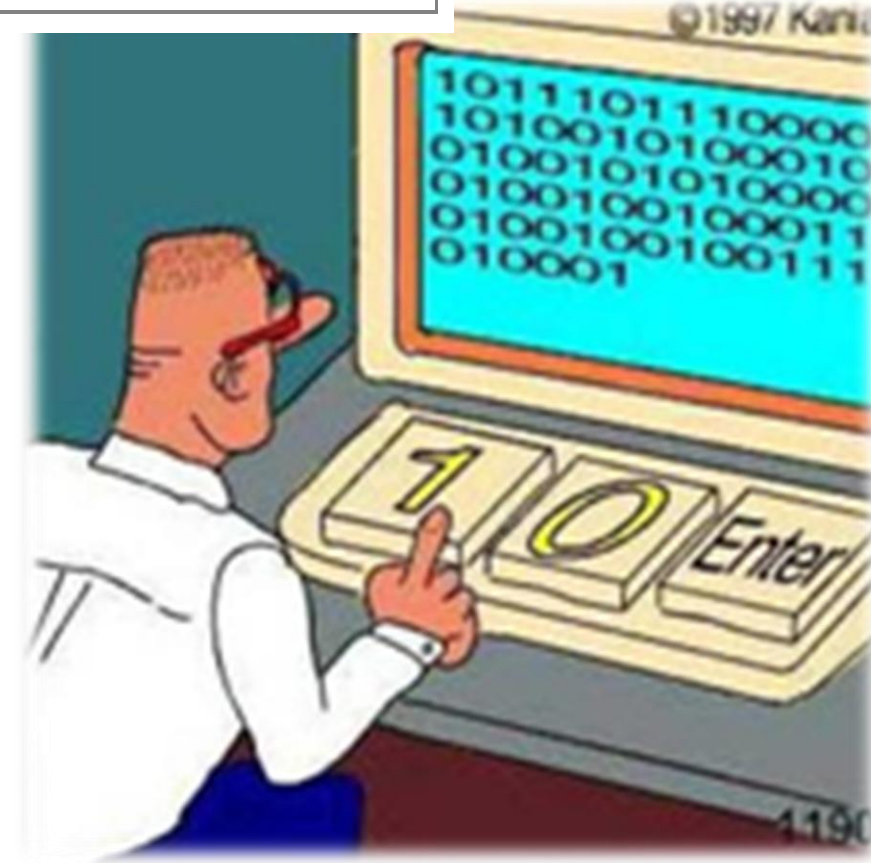
Tempo/Custo de
processamento



Entrada / Volume
de elementos



O(n)



Atividade com IA

- Vamos saber mais sobre:
 - O que significa “Notação Big-O”?
 - Por quê utilizar essa notação?
 - Quais são os tipos de complexidade?
- Faça individualmente, e depois compartilhe com o seu colega esses conceitos.
 - Faça as perguntas acima para a IA e compartilhe o resultado com seu colega... Resuma e prepare uma apresentação simples (1 parágrafo) para cada pergunta.



Complexidade de Algoritmos...

Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE

Notação **Big O** (razões)

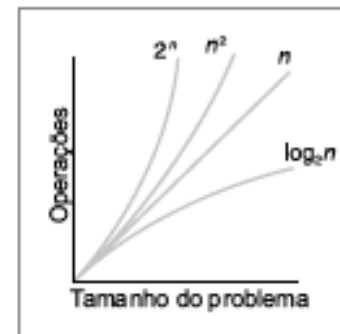


- **Abstração de Detalhes Menores:** *Big O* abstrai constantes e termos de ordem inferior, focando-se no fator de crescimento dominante. Isso facilita a comparação entre algoritmos.
- **Exemplo:** Se um algoritmo tem complexidade $5n+35n+35n+3$, a notação *Big O* simplifica isso para $O(n)$, ignorando constantes e termos menores.
- **Comparação de Algoritmos:** Facilita a comparação de diferentes algoritmos com base em seu desempenho teórico.
- **Exemplo:** Um algoritmo $O(n \log n)$ é geralmente mais eficiente que um algoritmo $O(n^2)$ para grandes volumes de dados.

Complexidade de Algoritmos...

Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE



Notação **Big O** (razões)

- **Escalabilidade:** Ajuda a entender como um algoritmo escala com o aumento do tamanho dos dados, essencial para problemas de grande escala.
- **Exemplo:** Algoritmos $O(\log n)$ e $O(n)$ são preferíveis para grandes entradas comparados a $O(n^2)$ ou $O(2^n)$.
- **Planejamento e Otimização:** Permite engenheiros e cientistas de dados planejarem e otimizarem sistemas de forma mais eficiente.
- **Exemplo:** Saber que uma função crítica é $O(n^2)$ pode levar à busca de alternativas mais eficientes para otimizar o desempenho.

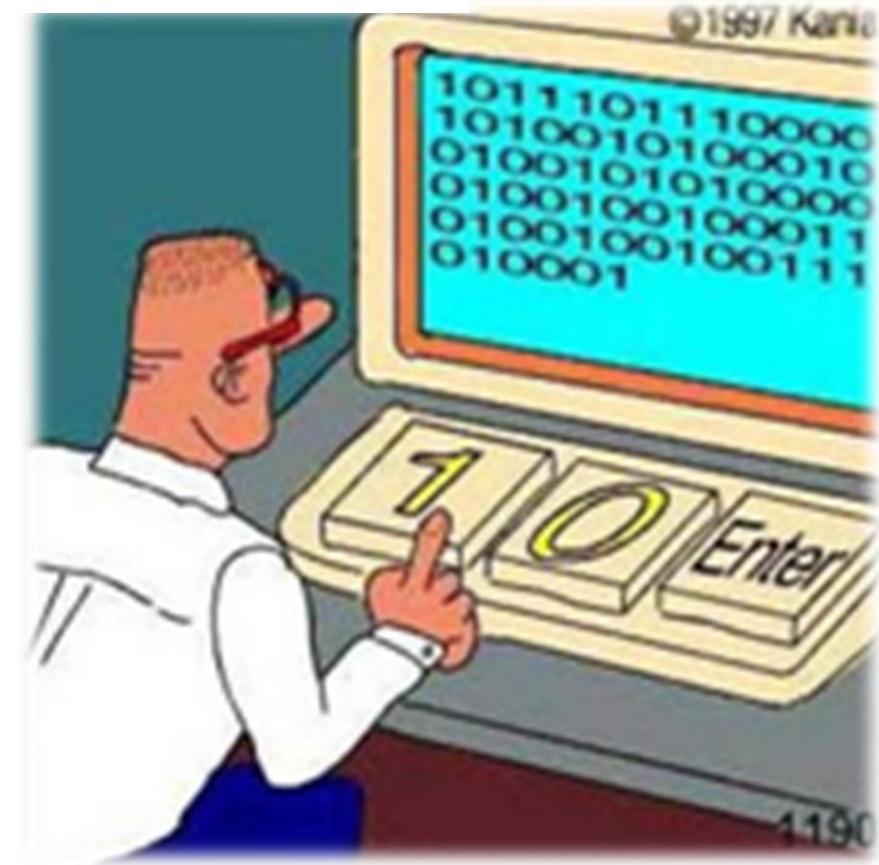
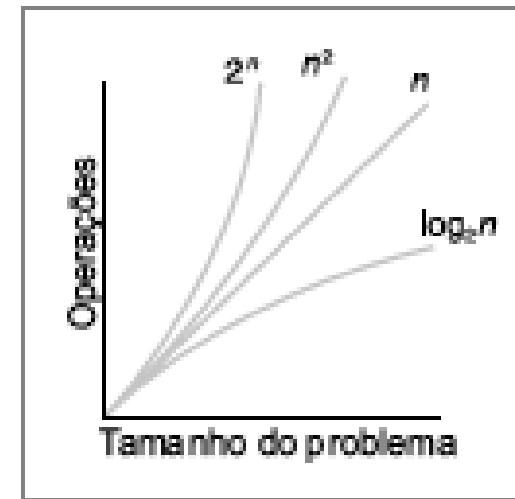
Complexidade de Algoritmos...

Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE

Notação **Big O**

- Complexidade constante (1)
 - **$O(1)$**
- Complexidade Logarítmica ($\log n$)
 - **$O(\log n)$**
- Complexidade Linear (n)
 - **$O(n)$**
- Complexidade Quadrática (n^2)
 - **$O(n^2)$**
- Complexidade Exponencial (2^n)
 - **$O(2^n)$**



Complexidade de Algoritmos...

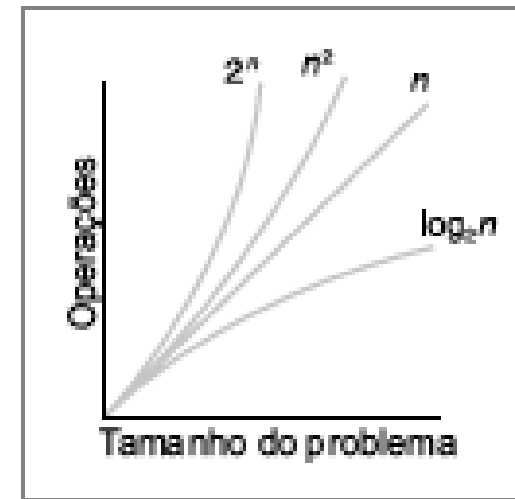
Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE

Significado da Notação Big O

A notação **Big O** nos fornece uma maneira de quantificar a eficiência dos algoritmos em termos de tempo e espaço.

Ela nos ajuda a prever como um algoritmo se comportará à medida que o tamanho da entrada aumenta, permitindo escolher a solução mais adequada para o problema em questão.



Complexidade de Algoritmos...

Eficiência de Algoritmos?

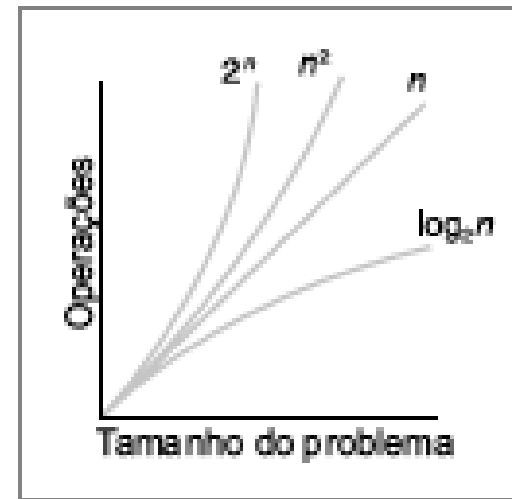
ORDENS DE COMPLEXIDADE

Exemplo de Aplicação: Big O

Levando em consideração que em linguagem Python em 1 segundo é possível executar 10^9 instruções...

Se você tiver uma entrada de n valores, onde $n = 10^6$, e tem apenas 1 segundo para rodar....

Se um algoritmo tem complexidade $O(n^2)$ sua execução será possível?



Complexidade de Algoritmos...

Eficiência de Algoritmos?

ORDENS DE COMPLEXIDADE

Exemplo de Aplicação: Big O

Levando em consideração que em linguagem Python em 1 segundo é possível executar 10^9 instruções...

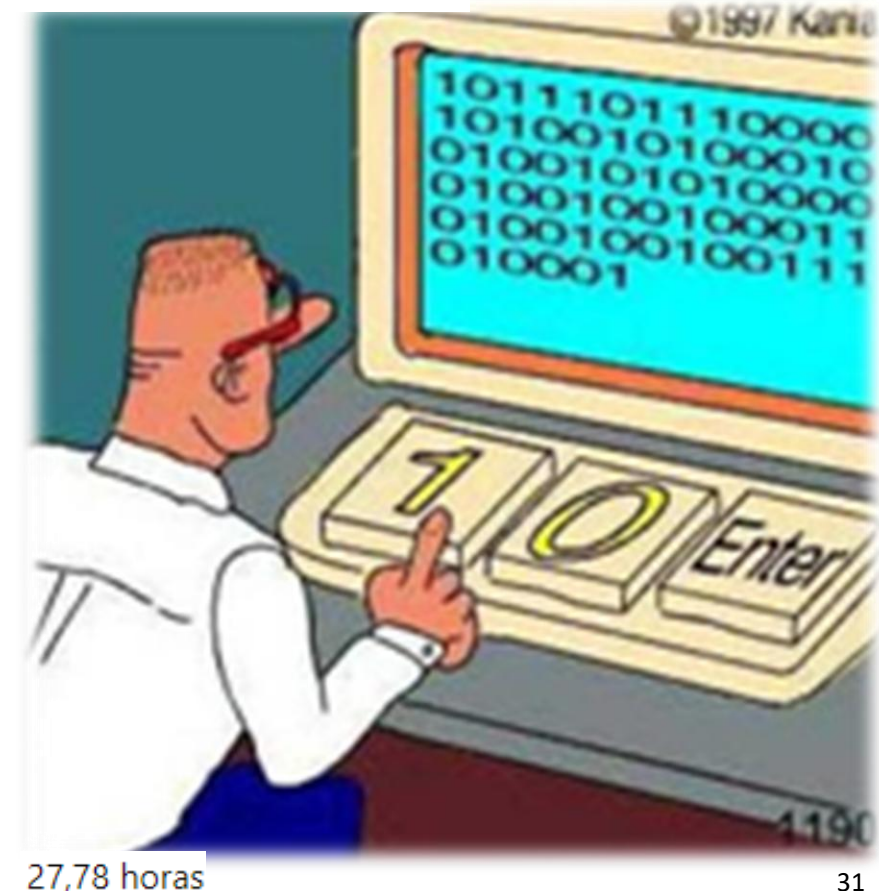
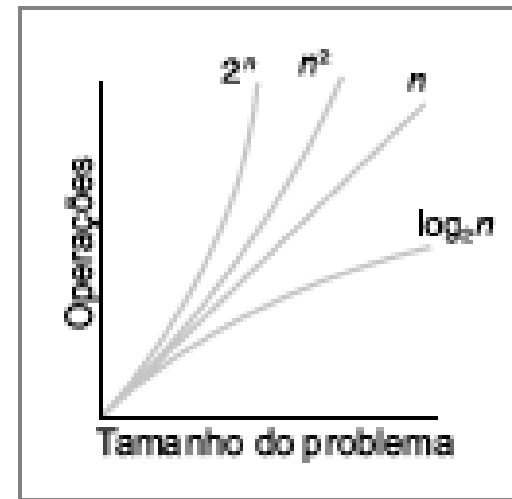
Se você tiver uma entrada de n valores, onde $n = 10^6$, e tem apenas 1 segundo para rodar....

Se um algoritmo tem complexidade $O(n^2)$ sua execução será possível?

Resposta: NÃO

Pois... $O(n^2)$ resultará em 10^{12} interações... Muito maior que 10^9

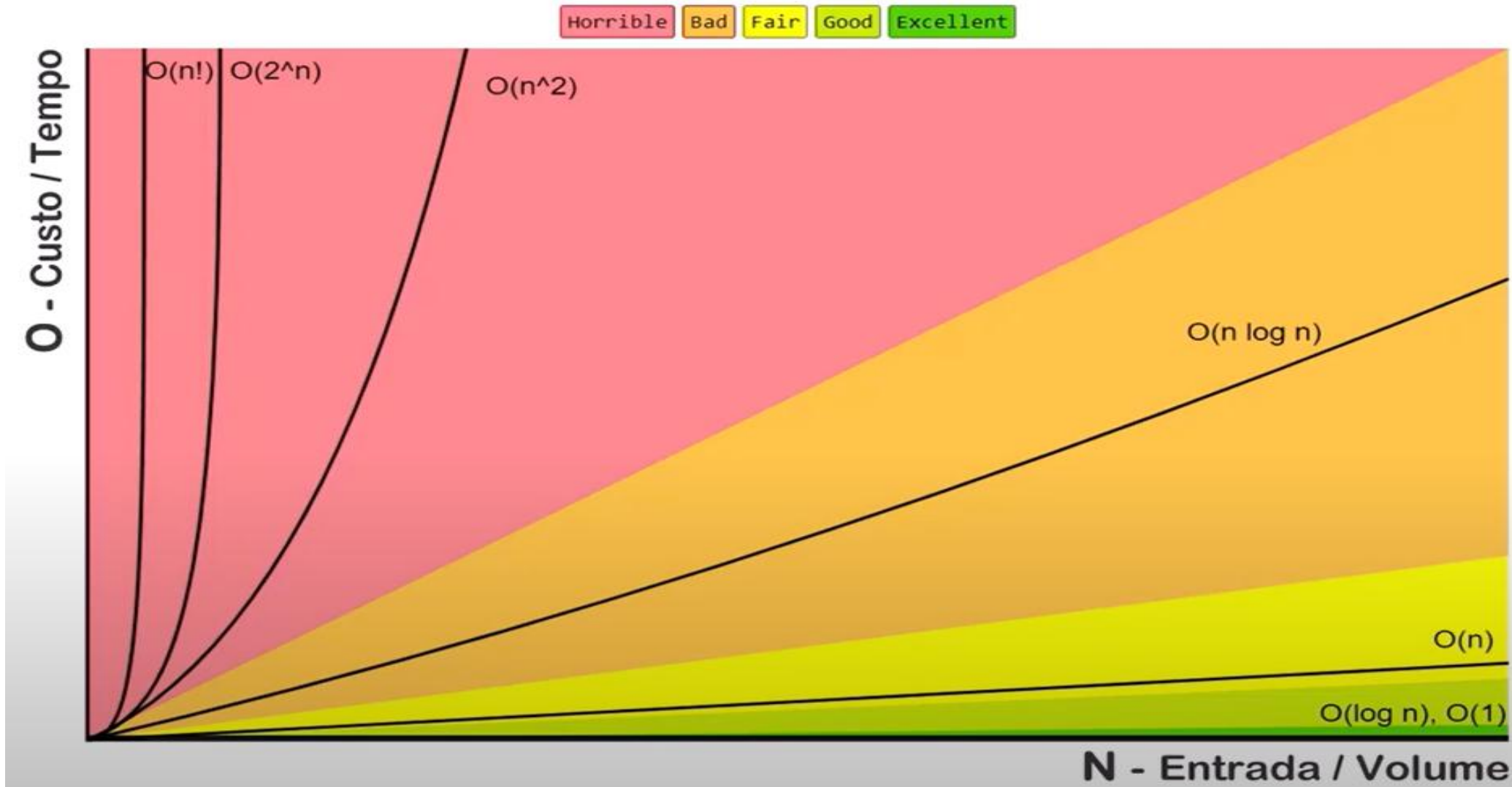
$$\text{Tempo necessário} = \frac{10^{12}}{10^9} = 10^3 \text{ segundos} \quad 100.000 \text{ s}$$



27,78 horas

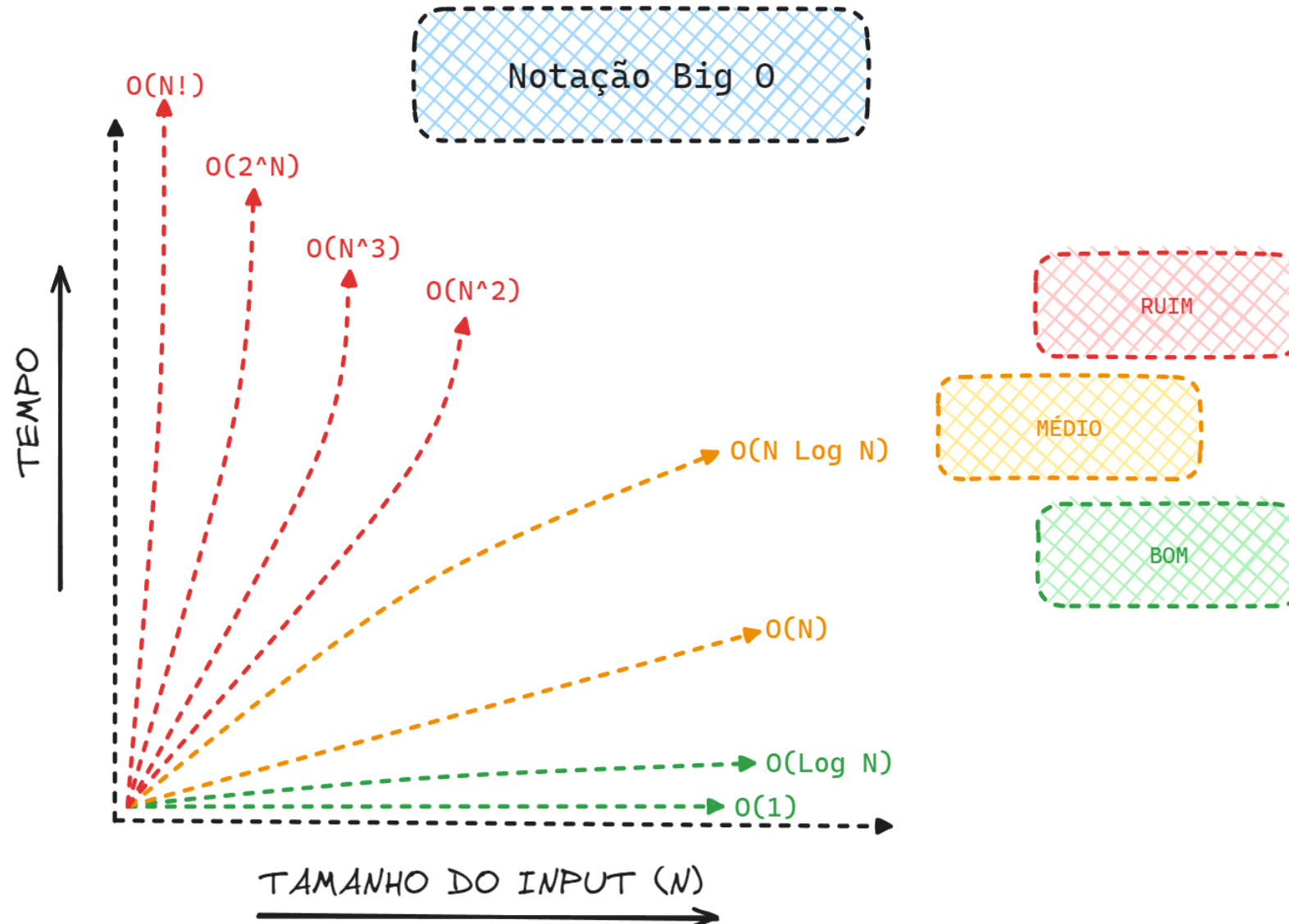
Complexidade de Algoritmos...

Eficiência de Algoritmos? ORDENS DE COMPLEXIDADE



Complexidade de Algoritmos...

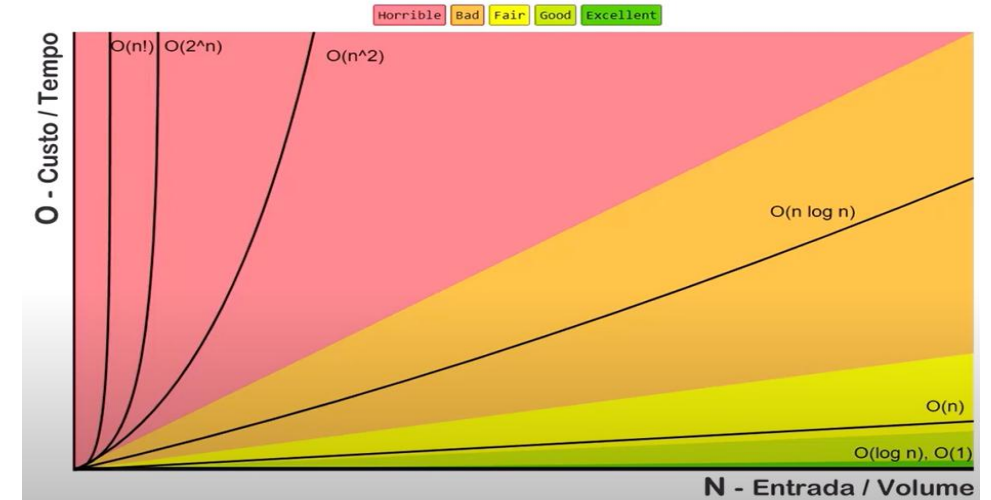
Eficiência de Algoritmos? ORDENS DE COMPLEXIDADE



Complexidade de Algoritmos...

Eficiência de Algoritmos? ORDENS DE COMPLEXIDADE

Big-O	Alternativa
$O(1)$	O(yeah)
$O(\log n)$	O(nice)
$O(n)$	O(ok)
$O(n \log n)$	O(uch)
$O(n^2)$	O(my)
$O(2^n)$	O(no)
$O(n!)$	O(mg!)



Complexidade de Algoritmos...

Eficiência de Algoritmos? ORDENS DE COMPLEXIDADE

<https://pt.wikipedia.org/wiki/Grande-O>

Notação	Nome	Ideia intuitiva	Definição informal: para um n suficientemente grande...	Definição formal
$f(n) = O(g(n))$	O-grande; Omicron-grande ^[12]	f é limitada superiormente por g (até no máximo um fator constante) assintoticamente	$ f(n) \leq k \cdot g(n) $ para algum k positivo.	$\exists k > 0 \exists n_0 \forall n > n_0 f(n) \leq k \cdot g(n) $
$f(n) = \Omega(g(n))$	Omega-grande	: Teoria dos números: f não é dominada por g assintoticamente Teoria da complexidade: f é limitada por baixo por g assintoticamente	Teoria dos números: $f(n) \geq k \cdot g(n)$ por infinitos valores de n e para algum k positivo Teoria da complexidade: $f(n) \geq k \cdot g(n)$ para algum k positivo	Teoria dos números: $\exists k > 0 \forall n_0 \exists n > n_0 f(n) \geq k \cdot g(n)$ Teoria da complexidade: $\exists k > 0 \exists n_0 \forall n > n_0 f(n) \geq k \cdot g(n)$
$f(n) = \Theta(g(n))$	Theta-grande	f é limitada por cima e por baixo por g assintoticamente	$k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ para algum k_1 e algum k_2 positivos	$\exists k_1 > 0 \exists k_2 > 0 \exists n_0 \forall n > n_0 k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$
$f(n) = o(g(n))$	o-pequeno	f é dominado por g assintoticamente	$ f(n) \leq k \cdot g(n) $, para todo número positivo k	$\forall k > 0 \exists n_0 \forall n > n_0 f(n) \leq k \cdot g(n) $
$f(n) = \omega(g(n))$	Omega-pequeno	f domina g assintoticamente	$ f(n) \geq k \cdot g(n) $, para todo número positivo k	$\forall k > 0 \exists n_0 \forall n > n_0 f(n) \geq k \cdot g(n) $
$f(n) \sim g(n)$	Na ordem de	f é igual a g assintoticamente	$f(n)/g(n) \rightarrow 1$	$\forall \epsilon > 0 \exists n_0 \forall n > n_0 \left \frac{f(n)}{g(n)} - 1 \right < \epsilon$

ORDENS DE COMPLEXIDADE



Cenário 2: Análise de Sentimentos em Redes Sociais

Imagine que você trabalha em uma empresa que quer monitorar o que as pessoas estão falando sobre seus produtos nas redes sociais.

O objetivo é analisar milhares de tweets para determinar se o sentimento geral é positivo, negativo ou neutro.

Para fazer isso, você precisa usar algoritmos que ajudem a processar e analisar grandes volumes de dados rapidamente.

Passo 1: Coleta de Dados

Você coleta 1.000.000 de tweets sobre o produto.

Esses tweets precisam ser processados para identificar o sentimento.

ORDENS DE COMPLEXIDADE

Passo 2: Pré-processamento dos Tweets

Você analisa a complexidade de tempo do seu algoritmo de pré-processamento.

- Complexidade: **$O(n)$** (linear)
- Explicação: *Cada tweet é processado uma vez, então o tempo cresce linearmente com o número de tweets.*
- Estimativa teórica: Para 1.000.000 de tweets, o tempo de execução é proporcional a 1.000.000.



ORDENS DE COMPLEXIDADE



Passo 3: Análise de Sentimentos

Método 1: Análise com Dicionário de Palavras

- Complexidade: **$O(n)$** (linear)
- Explicação: *Cada tweet é comparado com o dicionário de palavras uma vez.*
- Estimativa teórica: Para 1.000.000 de tweets, o tempo de execução é proporcional a 1.000.000.

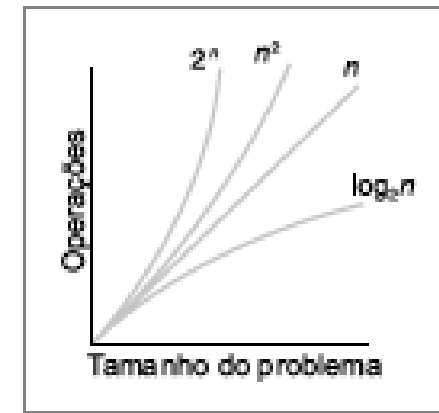
Método 2: Análise com Machine Learning

- Complexidade: **$O(n \log n)$** (linear-logarítmica)
- Explicação: *O modelo de machine learning processa cada tweet de forma mais eficiente.*
- Estimativa teórica: Para 1.000.000 de tweets, o tempo de execução é proporcional a $1.000.000 \log(1.000.000)$.

ORDENS DE COMPLEXIDADE

Passo 4: Relatório de Resultados

Você gera o relatório com a porcentagem de tweets positivos, negativos e neutros.



Resumo Teórico:

- Pré-processamento: **$O(n)$**
- Análise com Dicionário: **$O(n)$**
- Análise com Machine Learning: **$O(n \log n)$**

Utilizando a notação Big O, você pode prever de maneira mais precisa como o algoritmo se comportará com grandes volumes de dados.

Isso ajuda a escolher o algoritmo mais eficiente antes de implementá-lo e realizar medições empíricas.

ORDENS DE COMPLEXIDADE

Exemplos – $O(1)$

Algoritmo de Complexidade Constante

Acesso a um elemento de uma lista

```
def get_element_at_index(arr, index):  
    return arr[index]
```

```
# Exemplo de uso  
array = [10, 20, 30, 40, 50]  
print(get_element_at_index(array, 2)) # Saída: 30
```



ORDENS DE COMPLEXIDADE

Exemplos – $O(\log n)$

Algoritmo de Complexidade Logarítmica

Busca Binária

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Exemplo de uso

```
array = [10, 20, 30, 40, 50]
```

```
print(binary_search(array, 30)) # Saída: 2
```



ORDENS DE COMPLEXIDADE

Exemplos – $O(n)$

Algoritmo de Complexidade Linear

Soma de Elementos de uma lista

```
def sum_of_elements(arr):  
    total = 0  
    for num in arr:  
        total += num  
    return total
```

```
# Exemplo de uso  
array = [10, 20, 30, 40, 50]  
print(sum_of_elements(array)) # Saída: 150
```



ORDENS DE COMPLEXIDADE

Exemplos – $O(n^2)$

Algoritmo de Complexidade Quadrática

Ordenação Buble Sort

```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        for j in range(0, n-i-1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]  
    return arr
```

Exemplo de uso

```
array = [50, 20, 30, 10, 40]
```

```
print(bubble_sort(array)) # Saída: [10, 20, 30, 40, 50]
```



ORDENS DE COMPLEXIDADE

Exemplos – $O(2^n)$

Algoritmo de Complexidade Exponencial

Fibonacci (Recursivo)

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

```
# Exemplo de uso  
print(fibonacci(10)) # Saída: 55
```



VAMOS PARA A PRÁTICA ?!!!



ex1. Qual a ordem de complexidade ?

RETORNA O MAIOR VALOR EM UMA LISTA

```
def encontrar_maximo(lista):  
    valor_maximo = lista[0]  
    for numero in lista:  
        if numero > valor_maximo:  
            valor_maximo = numero  
    return valor_maximo
```

Exemplo de uso

```
lista = [10, 20, 30, 40, 50]  
print(encontrar_maximo(lista))  #  
Saída: 50
```

ex1. Qual a ordem de complexidade ?

Análise da Ordem de Complexidade

1. Inicialização do valor_maximo:

- `valor_maximo = lista[0]`
- Esta linha é executada uma vez, independentemente do tamanho da entrada.
- **Complexidade: $O(1)$**

```
def encontrar_maximo(lista):  
    valor_maximo = lista[0]  
    for numero in lista:  
        if numero > valor_maximo:  
            valor_maximo = numero  
    return valor_maximo  
  
# Exemplo de uso  
lista = [10, 20, 30, 40, 50]  
print(encontrar_maximo(lista)) # Saída: 50
```

ex1. Qual a ordem de complexidade ?

2. Loop for sobre a lista:

- `for numero in lista:`
- Este loop percorre todos os elementos da lista. Se a lista tiver n elementos, o loop executará n vezes.
- **Complexidade: $O(n)$**

```
def encontrar_maximo(lista):  
    valor_maximo = lista[0]  
    for numero in lista:  
        if numero > valor_maximo:  
            valor_maximo = numero  
    return valor_maximo  
  
# Exemplo de uso  
lista = [10, 20, 30, 40, 50]  
print(encontrar_maximo(lista)) # Saída: 50
```

ex1. Qual a ordem de complexidade ?

3. Retorno do valor_maximo:

- `return valor_maximo`
- Esta linha é executada uma vez, independentemente do tamanho da entrada.
- **Complexidade: $O(1)$**

```
def encontrar_maximo(lista):  
    valor_maximo = lista[0]  
    for numero in lista:  
        if numero > valor_maximo:  
            valor_maximo = numero  
    return valor_maximo
```

Exemplo de uso

```
lista = [10, 20, 30, 40, 50]
```

```
print(encontrar_maximo(lista)) # Saída: 50
```

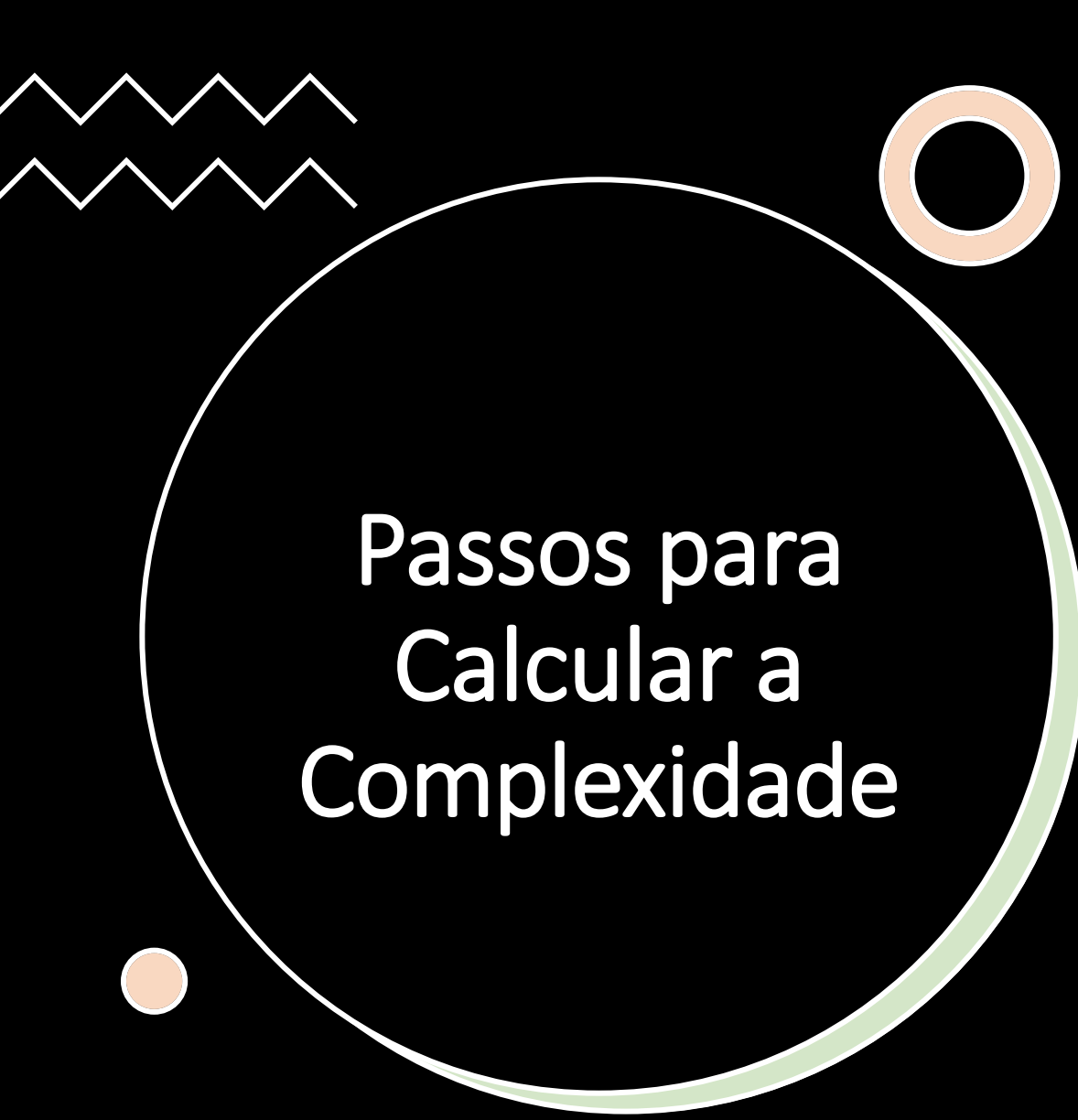

ex1. Qual a ordem de complexidade ?

```
def encontrar_maximo(lista):  
    valor_maximo = lista[0]  
    for numero in lista:  
        if numero > valor_maximo:  
            valor_maximo = numero  
    return valor_maximo
```


```
# Exemplo de uso  
lista = [10, 20, 30, 40, 50]  
print(encontrar_maximo(lista)) #  
Saída: 50
```

Para determinar a complexidade total do algoritmo, observamos a parte que mais contribui para o tempo de execução à medida que o tamanho da entrada “n” cresce...

Portanto, a complexidade desse algoritmo é:
 $O(n)$



Passos para Calcular a Complexidade

1. Levar em consideração apenas as repetições do código
 2. Verificar a complexidade das funções/métodos próprios da linguagem (se utilizado)
 3. Ignorar as constantes e utilizar o termo de maior grau
- 

Ex2. Qual a ordem de complexidade ?

SOMA TODOS OS ELEMENTOS DE UMA LISTA

```
def somar_elementos(lista):  
    soma = 0  
    for numero in lista:  
        soma += numero  
    return soma
```

```
# Exemplo de uso  
lista = [10, 20, 30, 40, 50]  
print(somar_elementos(lista))  
# Saída: 150
```

**AGORA É COM
VOCÊ !!!!**



Ex2. Qual a ordem de complexidade ?

Análise da Ordem de Complexidade

1. Inicialização da soma:

- `soma = 0`
- Esta linha é executada uma vez, independentemente do tamanho da entrada.
- **Complexidade: $O(1)$**

```
def somar_elementos(lista):  
    soma = 0  
    for numero in lista:  
        soma += numero  
    return soma  
  
# Exemplo de uso  
lista = [10, 20, 30, 40, 50]  
print(somar_elementos(lista)) # Saída: 150
```

Ex2. Qual a ordem de complexidade ?

2. Loop for sobre a lista:

- `for numero in lista:`
- Este loop percorre todos os elementos da lista. Se a lista tiver n elementos, o loop executará n vezes.
- **Complexidade: $O(n)$**

```
def somar_elementos(lista):  
    soma = 0  
    for numero in lista:  
        soma += numero  
    return soma  
  
# Exemplo de uso  
lista = [10, 20, 30, 40, 50]  
print(somar_elementos(lista)) # Saída: 150
```


Ex2. Qual a ordem de complexidade ?

3. Retorno do valor_maximo:

- `return soma`
- Esta linha é executada uma vez, independentemente do tamanho da entrada.
- **Complexidade: $O(1)$**

```
def somar_elementos(lista):  
    soma = 0  
    for numero in lista:  
        soma += numero  
    return soma  
  
# Exemplo de uso  
lista = [10, 20, 30, 40, 50]  
print(somar_elementos(lista)) # Saída: 150
```

Ex2. Qual a ordem de complexidade ?

```
def somar_elementos(lista):  
    soma = 0  
    for numero in lista:  
        soma += numero  
    return soma
```

```
# Exemplo de uso  
lista = [10, 20, 30, 40, 50]  
print(somar_elementos(lista))  
# Saída: 150
```

O laço for é o que mais contribui... PASSO 1...

Não existem outras funções (PASSO 2)

Não existem constantes (PASSO 3)

A complexidade desse algoritmo é:

$O(n)$

**AGORA É COM
VOCÊ !!!!**



Ex3. Qual a ordem de complexidade ?

```
def exemplo(lista):  
    tamanho = len(lista)  
    for i in range(tamanho):  
        for j in range(tamanho):  
            if (i != j):  
                return True  
    return False
```

PASSO 1:

PASSO 2:

PASSO 3:

A complexidade desse algoritmo é:

$O()$

$\text{len()} \text{ e } \text{range()} \rightarrow O(1)$

Ex3. Qual a ordem de complexidade ?

```
def exemplo(lista):  
    tamanho = len(lista)  
    for i in range(tamanho):  
        for j in range(tamanho):  
            if (i != j):  
                return True  
    return False
```

PASSO 1: $O(n) * O(n)$

PASSO 2: $O(1)$ e $O(1)$

PASSO 3: $O(n^2)$

A complexidade desse algoritmo é:

$O(n^2)$

$\text{len()} \text{ e } \text{range()} \rightarrow O(1)$

Ex4. Qual a ordem de complexidade ?

Qual dos dois algoritmos abaixo apresenta menor complexidade?

1

```
def existem_2_menores(idades):  
    tamanho = len(idades)  
    menor_idade = 200  
    for i in range(tamanho):  
        if idades[i] < menor_idade:  
            menor_idade = idades[i]  
  
    cont = 0  
    for i in range(tamanho):  
        if idades[i] == menor_idade:  
            cont += 1  
  
    return cont > 1
```

2

```
def exemplo6(idades):  
    idades.sort()  
    return idades[0] == idades[1]
```

Complexidade da função sort()
 $O(n \log n)$

Ex4. Qual a ordem de complexidade ?

Qual dos dois algoritmos abaixo apresenta menor complexidade?

1

```
def existem_2_menores(idades):  
    tamanho = len(idades)  
    menor_idade = 200  
    for i in range(tamanho):  
        if idades[i] < menor_idade:  
            menor_idade = idades[i]  
  
    cont = 0  
    for i in range(tamanho):  
        if idades[i] == menor_idade:  
            cont += 1  
  
    return cont > 1
```

$O(n)$

$O(n) + O(n) = 2 O(n) \rightarrow O(n)$

2

```
def exemplo6(idades):  
    idades.sort()  
    return idades[0] == idades[1]
```

$O(n \log n)$

Ex4. Qual a ordem de complexidade ?

Qual dos dois algoritmos abaixo apresenta **menor complexidade**?

1

```
def existem_2_menores(idades):  
    tamanho = len(idades)  
    menor_idade = 200  
    for i in range(tamanho):  
        if idades[i] < menor_idade:  
            menor_idade = idades[i]  
  
    cont = 0  
    for i in range(tamanho):  
        if idades[i] == menor_idade:  
            cont += 1  
  
    return cont > 1
```

$O(n)$

$O(n)$

2

```
def existem_2_menores(idades):  
    idades.sort()  
    return idades[0] == idades[1]
```

$O(n \log n)$

$O(n) + O(n) = 2 O(n) \rightarrow O(n)$

Próxima Aula



- Ler o capítulo 3 do livro “Estrutura de Dados com Python”



Boa semana e bons estudos!!