Estimando complexidade de tempo

Fatec 2025

Notação Big O

- Notação especial que diz quão rápido é um algoritmo em função da entrada de dados

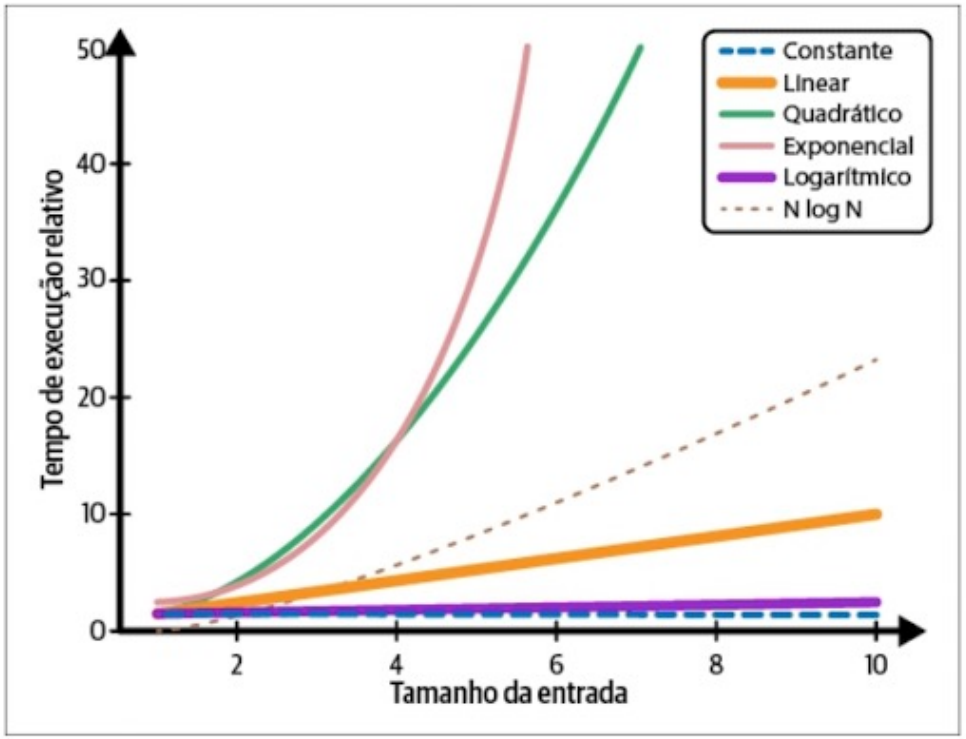
	Pesquisa simples	Pesquisa binária
100 elementos	100ms	7ms
10.000 elementos	10 s	14ms
1.000.000.000 elementos	11 dias	32ms

Notação Big O

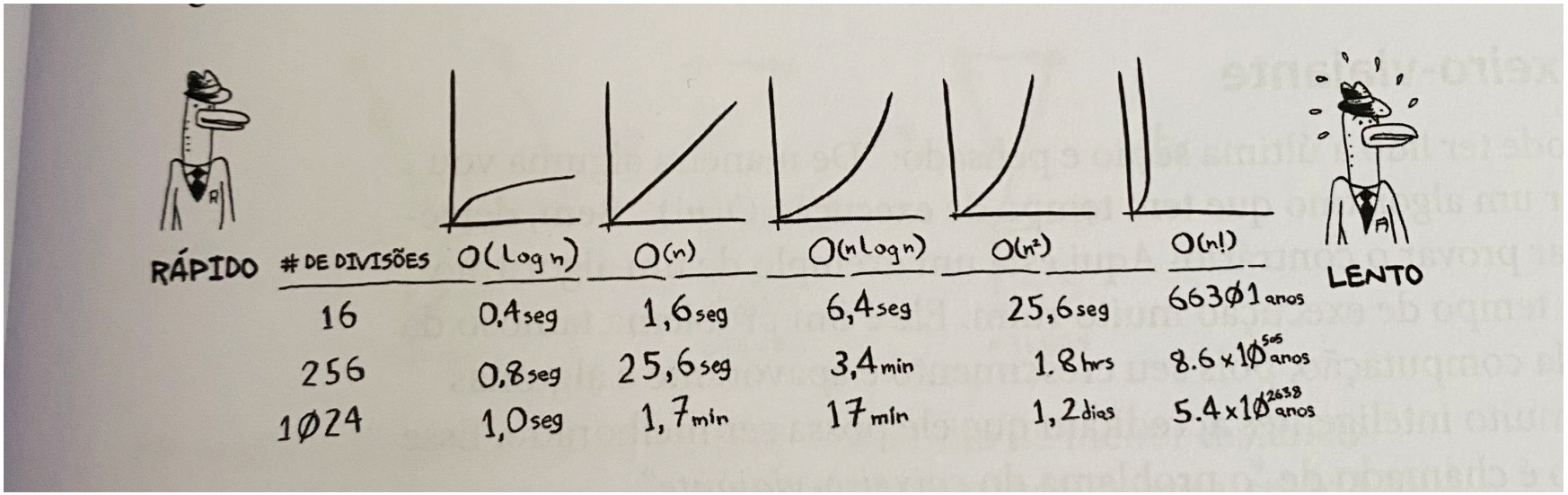
- Estabelece o tempo de execução para a pior hipótese
- Pesquisa simples de um nome em uma lista = O(n)
  - O(n) = no pior caso você irá ter que ler todos os nomes
  - No melhor caso, o nome procurado é o primeiro
- Independe do hardware

Tempo de execução Big O

- O(1) tempo constante. O tempo é o mesmo, independente do tamanho do conjunto
- O(n) tempo linear. Ex: pesquisa simples
- O(log n) tempo logaritmo. Ex: busca binária
- O(n * log n) S=Algoritmo rápido de ordenação.Ex. QuickSort
- O (n^2) tempo quadrático. Algoritmo lento de ordenação. Ex: Ordenação por seleção
- O(n!) . Algoritmo muito lento. Ex: Caixeiro viajante.



Tempos de execução Big O



# DE DIVISÕES	O(Log n)	O(n)	O(n Log n)	O(n²)	O(n!)
16	0,4 seg	1,6 seg	6,4 seg	25,6 seg	663Ø1 anos
256	0,8 seg	25,6 seg	3,4 min	1.8 hrs	$8.6 \times 1Ø^{505}$ anos
1Ø24	1,0 seg	1,7 min	17 min	1,2 dias	$5.4 \times 1Ø^{2638}$ anos

- A rapidez de um algoritmo não é medido em segundos, mas pelo crescimento do número de operações
- Ou seja, quão rapidamente o tempo de execução aumenta conforme o número de elementos aumenta
- Tempo de execucação em algoritmos é expresso na notação big O
- O(log n) é mais rápido do que O(n), e O(log n) fica ainda mais rápido conforme a lista aumenta

```
import time
import random
import matplotlib.pyplot as plt
def weighted_mean(list_of_numbers, weights):
    running_total = 0
    for i in range(len(list_of_numbers)):
        running_total += (list_of_numbers[i] * weights[i])
    return running_total / sum(weights)

# Função auxiliar para medir tempo
def medir_tempo(tamanho):
    numeros = [random.uniform(1, 100) for _ in range(tamanho)]
    pesos = [random.uniform(1, 10) for _ in range(tamanho)]

    inicio = time.time()
    resultado = weighted_mean(numeros, pesos)
    fim = time.time()

    return fim - inicio
```

```
# Listas para guardar os tamanhos e tempos
tamanhos = [10, 100, 1_000, 10_000, 100_000, 1_000_000]
tempos = []

print("Testando função e coletando dados para o gráfico...\n")
for tamanho in tamanhos:
    tempo_execucao = medir_tempo(tamanho)
    tempos.append(tempo_execucao)
    print(f"Tamanho: {tamanho:>9} | Tempo: {tempo_execucao:.6f} segundos")

# Criando o gráfico
plt.figure(figsize=(10, 6))
plt.plot(tamanhos, tempos, marker='o')
plt.title("Complexidade de Tempo da Função weighted_mean")
plt.xlabel("Tamanho da lista (n)")
plt.ylabel("Tempo de execução (segundos)")
plt.grid(True)
plt.xscale('log')  # Escala logarítmica para melhor visualização
#plt.yscale('log')  # Opcional: mostra que o crescimento é linear em escala log
plt.show()
```

O gráfico mostra uma **linha aproximadamente reta** em escala log-log, reforçando que a complexidade da função é linear **O(n)**.

Busca linear x Busca binária

```
import time
import random
import matplotlib.pyplot as plt
import bisect

# Busca linear: percorre a lista elemento por elemento
def busca_linear(lista, alvo):
    for i in range(len(lista)):
        if lista[i] == alvo:
            return i
    return -1
```

```
# Busca binária: assume que a lista está ordenada
def busca_binaria(lista, alvo):
    pos = bisect.bisect_left(lista, alvo)
    if pos != len(lista) and lista[pos] == alvo:
        return pos
    return -1
```

```
# Função para medir tempo das buscas
def medir_tempo_buscas(tamanho):
    lista = sorted(random.sample(range(tamanho * 10), tamanho))  # Lista ordenada sem repetições
    alvo = lista[-1]  # Pior caso para busca linear (último elemento)

    # Medir tempo busca linear
    inicio = time.time()
    busca_linear(lista, alvo)
    fim = time.time()
    tempo_linear = fim - inicio

    # Medir tempo busca binária
    inicio = time.time()
    busca_binaria(lista, alvo)
    fim = time.time()
    tempo_binaria = fim - inicio

    return tempo_linear, tempo_binaria
```

```
# Tamanhos das listas para o teste
tamanhos = [10, 100, 1_000, 10_000, 100_000, 1_000_000]
tempos_linear = []
tempos_binaria = []

print("Comparando busca linear vs binária...\n")
for tamanho in tamanhos:
    tempo_lin, tempo_bin = medir_tempo_buscas(tamanho)
    tempos_linear.append(tempo_lin)
    tempos_binaria.append(tempo_bin)
    print(f"Tamanho: {tamanho:>9} | Linear:
{tempo_lin:.6f}s | Binária: {tempo_bin:.6f}s")
```

```
# Plotando o gráfico comparativo
plt.figure(figsize=(10, 6))
plt.plot(tamanhos, tempos_linear, marker='o', label='Busca Linear (O(n))')
plt.plot(tamanhos, tempos_binaria, marker='s', label='Busca Binária (O(log n))')
plt.title("Comparação de Complexidade: Busca Linear vs Binária")
plt.xlabel("Tamanho da lista (n)")
plt.ylabel("Tempo de execução (segundos)")
plt.grid(True)
plt.legend()
plt.xscale('log')
plt.yscale('log')  # Mostra claramente a diferença entre O(n) e O(log n)
plt.show()
```

Busca linear x Busca binária

A **busca linear** precisa examinar cada elemento até encontrar o alvo → **O(n)**

- A **busca binária**, por ser baseada em divisão sucessiva de listas ordenadas, localiza rapidamente → **O(log n)**
- Com o gráfico em escala log-log, fica evidente como a busca binária cresce muito mais lentamente mesmo em listas enormes

Buble Sort

- https://www.geeksforgeeks.org/bubble-sort-algorithm/
- https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm
- https://www.youtube.com/watch?v=lyZQPjUT5B4

Insertion Sort

- https://www.geeksforgeeks.org/insertion-sort/
- https://visualgo.net/en/sorting
- https://www.youtube.com/watch?v=ROalU379l3U

Buble Sort x Insertion Sort x TimSort

Buble Sort e Insertion Sort são didáticos mas ineficientes para grande volume de dados
TimSort é otimizado para uso prático

```
import time
import random
import matplotlib.pyplot as plt

# Bubble Sort (O(n²))
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
```

```python
# Insertion Sort (O(n²))
def insertion_sort(arr):
    for i in range(1, len(arr)):
        chave = arr[i]
        j = i - 1
        while j >= 0 and chave < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = chave
    return arr
```

```python
# Função para medir tempos de ordenação
def medir_tempo_ordenacao(tamanho):
    lista_original = [random.randint(0, tamanho) for _ in range(tamanho)]
    # Bubble Sort
    lista = lista_original.copy()
    inicio = time.time()
    bubble_sort(lista)
    tempo_bubble = time.time() - inicio
    # Insertion Sort
    lista = lista_original.copy()
    inicio = time.time()
    insertion_sort(lista)
    tempo_insertion = time.time() - inicio
    # TimSort (built-in sorted)
    lista = lista_original.copy()
    inicio = time.time()
    sorted(lista)
    tempo_timsort = time.time() - inicio

    return tempo_bubble, tempo_insertion, tempo_timsort
```

```
# Tamanhos para teste (limitados para evitar tempos longos)
tamanhos = [10, 100, 500, 1000, 2000] # Pode aumentar se quiser, mas Bubble é lento
tempos_bubble = []
tempos_insertion = []
tempos_timsort = []
print("Comparando algoritmos de ordenação...\n")
for tamanho in tamanhos:
    t_b, t_i, t_t = medir_tempo_ordenacao(tamanho)
    tempos_bubble.append(t_b)
    tempos_insertion.append(t_i)
    tempos_timsort.append(t_t)
    print(f"Tamanho: {tamanho:>5} | Bubble: {t_b:.6f}s | Insertion: {t_i:.6f}s |
TimSort: {t_t:.6f}s")
```

```
# Gráfico comparativo
plt.figure(figsize=(10, 6))
plt.plot(tamanhos, tempos_bubble, marker='o', label='Bubble Sort (O(n²))')
plt.plot(tamanhos, tempos_insertion, marker='s', label='Insertion Sort (O(n²))')
plt.plot(tamanhos, tempos_timsort, marker='^', label='TimSort - sorted() (O(n log n))')
plt.title("Comparação de Algoritmos de Ordenação")
plt.xlabel("Tamanho da lista (n)")
plt.ylabel("Tempo de execução (segundos)")
plt.grid(True)
plt.legend()
plt.show()
```

QuestÕes:

1- O que você observa no gráfico em relação ao tempo de execução do Bubble Sort e do Insertion Sort?

2. Qual algoritmo se comportou melhor conforme o tamanho da lista aumentou? Por quê?

3. O Timsort foi mais rápido? Com base na teoria, qual sua complexidade de tempo e por que isso faz diferença?

4. Comparando os algoritmos:
   - Quais algoritmos têm complexidade quadrática ($O(n^2)$)?
   - Qual tem complexidade log-linear ($O(n \log n)$)?

- 5. Execute o teste com uma lista de tamanho 3000 ou 4000 (adicione no vetor tamanhos).
  - O que acontece com o tempo do Bubble Sort?
  - Por que isso ocorre?