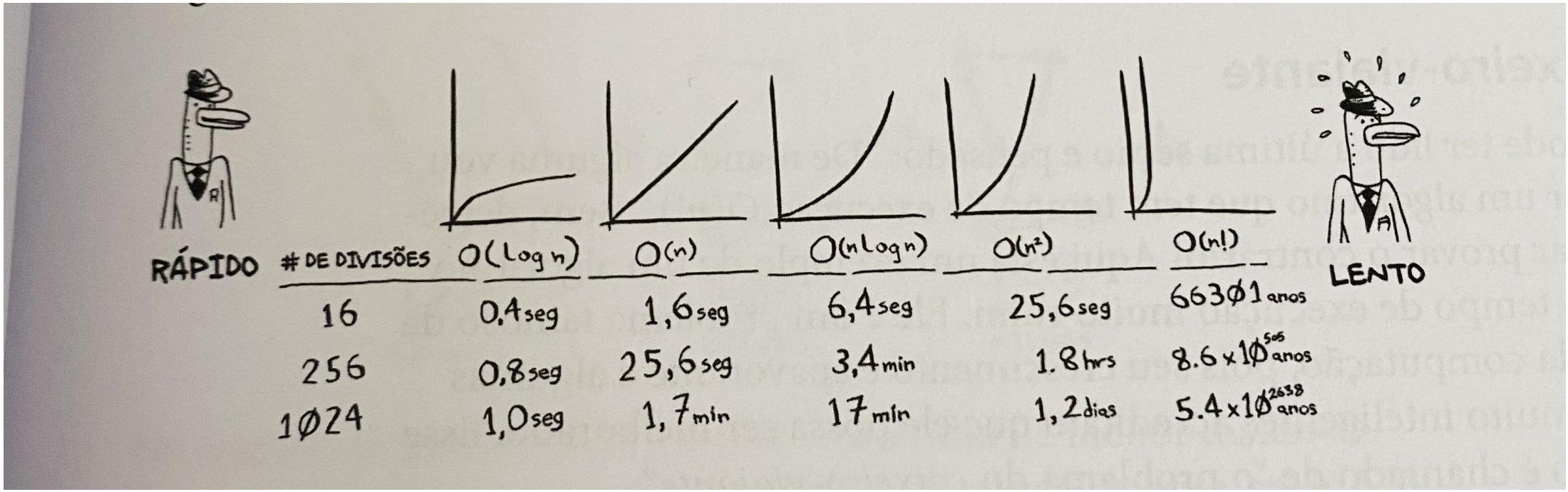
Técnicas de Programação

Usando estruturas de dados de modo eficaz

Métodos para melhorar o desempenho

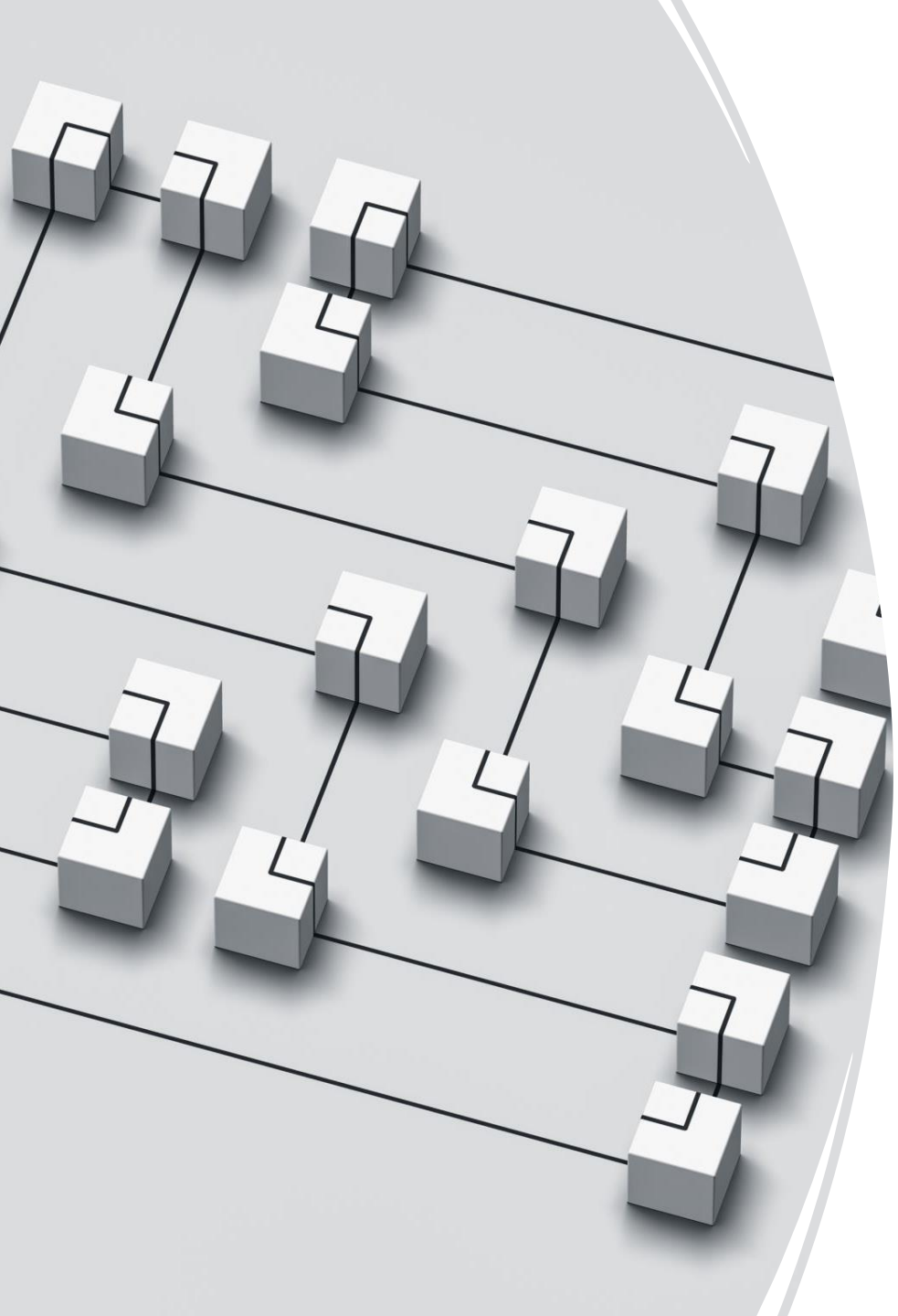
- Escolha do algoritmo
  - Evite iterações desnecessárias
- Escolha da estrutura de dados
  - É mais rápido procurar um valor em um dicionário do que em uma lista
- Use funções built-in
  - Muitas são implementadas em C
- Compilando Python
  - Cython (superconjunto do Pyton), Numba (subconjunto do python) e PyPy (reimplementação do Python)
  - Código Assíncrono
  - Realiza uma tarefa enquanto aguarda outra
- Computação paralela e computação distribuída
  - MapReduce
    - Modelo de programação para BigData

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	$8.6 \times 1Ø^{5Ø5}$ 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



Estruturas de dados

- Usar uma variedade de estruturas de dados
- Estruturas de dados mais comuns:
  - Listas
  - Tuplas
  - dicionários
  - Arrays (NumPy) e
  - dataframes (pandas)

Estruturas de dados

Estrutura	Características principais	Símbolo usado	Exemplo
Tupla	Imutável, tamanho fixo, acesso por índice (O(1))	()	(3.5, 7.2, 0.0)
Lista	Mutável, ordenada, permite vários tipos de dados	[]	['maçã', 'banana', 'laranja']
Array	Estrutura homogênea, mais eficiente para cálculo numérico	array.array() ou np.array()	np.array([1, 2, 3]) (com NumPy)
Dicionário	Chave-valor, acesso rápido por chave (O(1))	{}	{'nome': 'Ana', 'idade': 25}
Set	Conjunto de valores únicos, não ordenado, busca rápida (O(1))	{}	{'maçã', 'banana', 'laranja'}
DataFrame	Tabela (linhas e colunas), ideal para manipulação de dados tabulares	pd.DataFrame()	DataFrame com colunas Nome, Email, CPF

Listas

- Listas em Python são um tipo de array: uma estrutura de dados que tem alguma ordem
- Array dinâmico, pode ser redimensionada
- Pode armazenar diferentes tipos
- Aloca um fragmento contínuo conforme o tamanho da lista: cada elemento da lista fica próximo ao local de memória adjacente do próximo elemento. Isso torna fácil procurar um elemento

Comparando listas

```python
import random
import timeit
# Criar listas de tamanhos diferentes
lista_pequena = random.sample(range(1, 100), 10)
lista_grande = random.sample(range(1, 20000), 10000)
# Elementos a serem buscados (últimos de cada lista)
alvo_pequeno = lista_pequena[-1]
alvo_grande = lista_grande[-1]
# Código para timeit (igual para ambos os casos)
codigo_teste_pequena = 'alvo_pequeno in lista_pequena'
codigo_teste_grande = 'alvo_grande in lista_grande'
# Medir tempo para lista pequena
tempo_pequena = timeit.timeit(stmt=codigo_teste_pequena,
                              globals=globals(), number=1000)
# Medir tempo para lista grande
tempo_grande = timeit.timeit(stmt=codigo_teste_grande,
                             globals=globals(), number=1000)
# Exibir os resultados
print("🔍 Comparação de busca linear (último elemento):")
print(f"Lista com   10 elementos: {tempo_pequena:.6f} s (média: {tempo_pequena/1000:.10f} s)")
print(f"Lista com 10000 elementos: {tempo_grande:.6f} s (média: {tempo_grande/1000:.10f} s)")
```

Conjuntos

- Estrutura de dados sem ordem inerente
- São implementados com uma tabela hash com um conjunto de chaves exclusivas
- Todo elemento de um conjunto deve ser exclusivo
- Operações de incluir, excluir e atualizar em tempo O(1)

Comparando listas e conjuntos

```python
import random
import timeit
import matplotlib.pyplot as plt
# Gerar dados: lista com 10.000 elementos únicos
dados = random.sample(range(1, 20000), 10000)
alvo = dados[-1]  # Pior caso: último elemento
lista = dados
conjunto = set(dados)
# Medir tempo para diferentes quantidades de buscas
quantidades = [10, 100, 1000, 5000, 10000]
tempos_lista = []
tempos_set = []
for n in quantidades:
    tempo_lista = timeit.timeit(stmt='alvo in lista', globals=globals(), number=n)
    tempo_set = timeit.timeit(stmt='alvo in conjunto', globals=globals(), number=n)
    tempos_lista.append(tempo_lista)
    tempos_set.append(tempo_set)
# Mostrar resultados numéricos
print("🔍 Comparação de tempo total para diferentes quantidades de buscas:")
for i, n in enumerate(quantidades):
    print(f"{n:>5} buscas - list: {tempos_lista[i]:.6f}s | set: {tempos_set[i]:.6f}s")
# Plotar o gráfico
plt.figure(figsize=(10, 6))
plt.plot(quantidades, tempos_lista, marker='o', label='Busca em list (O(n))')
plt.plot(quantidades, tempos_set, marker='s', label='Busca em set (O(1))')
plt.title('Comparação de Tempo de Busca: list vs set')
plt.xlabel('Número de buscas')
plt.ylabel('Tempo total (segundos)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Dicionários

- Baseados em pares chave-valor
- Existem pares de elementos de dados que tem algum vínculo
- Dicionários são melhores para dados que não tem ordenação inerente

Dicionários

- Dicionários em Python são implementados por tabelas hash
- Usa função hash para transformar uma chave no índice de uma lista
- A busca em um dicionário é O(1)

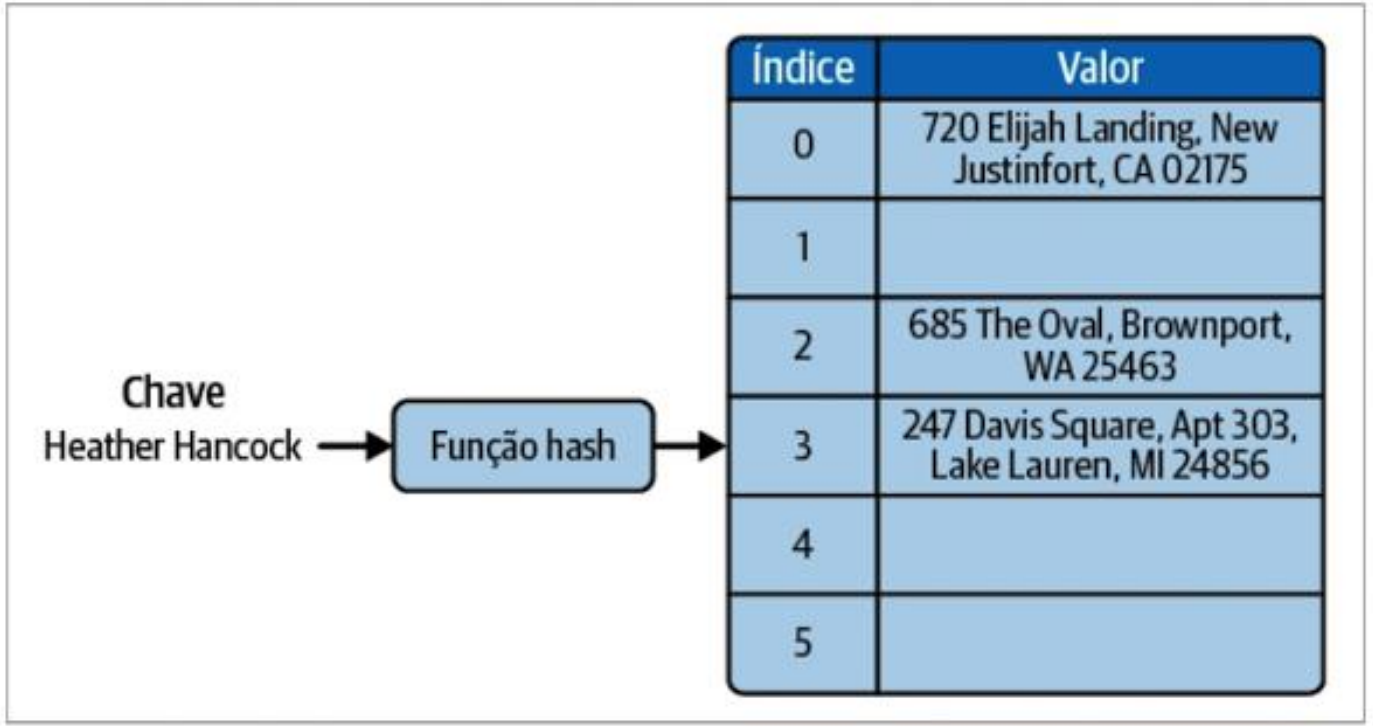


Tabela hash

```python
import matplotlib.pyplot as plt
# Tamanho da tabela hash
tamanho = 5
# Nomes fictícios para simular inserção
entradas = ["João", "Ana", "Bia", "Carlos", "Lucas", "Lia", "Pedro", "Bruno"]
# Criar tabela hash como lista de listas (encadeamento)
hash_slots = [[] for _ in range(tamanho)]
# Função hash simples: soma dos códigos dos caracteres da chave
def hash_simples(chave):
    return sum(ord(c) for c in chave) % tamanho
# Inserir nomes na tabela usando a função hash
for nome in entradas:
    indice = hash_simples(nome)
    hash_slots[indice].append(nome)
# Preparar os dados para o gráfico
labels = [f"Slot {i}" for i in range(tamanho)]
valores = [len(slot) for slot in hash_slots]
conteudo = [", ".join(slot) if slot else "vazio" for slot in hash_slots]
# Plotar o gráfico
fig, ax = plt.subplots(figsize=(10, 6))
bars = ax.bar(labels, valores, color='skyblue')
# Adicionar rótulos com os nomes nas barras
for bar, texto in zip(bars, conteudo):
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2, yval + 0.1, texto, ha='center', fontsize=10)
plt.title("Distribuição de nomes em uma Tabela Hash simples")
plt.ylabel("Número de entradas por slot")
plt.ylim(0, max(valores)+1)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

Criando dicionário

```python
from faker import Faker

fake = Faker('pt_BR')

# Criar um dicionário com dados de 5 clientes
clientes = {}

for _ in range(5):
    nome = fake.name()
    dados = {
        'CPF': fake.cpf(),
        'Email': fake.email(),
        'Endereço': fake.address(),
        'Data de Nascimento': fake.date_of_birth(minimum_age=18, maximum_age=65)
    }
    clientes[nome] = dados

# Exibir os dados
for nome, info in clientes.items():
    print(f"Cliente: {nome}")
    for chave, valor in info.items():
        print(f"  {chave}: {valor}")
    print("-" * 40)
```

Tempo em dicionário

```
! pip install faker
from faker import Faker
import timeit
import random

fake = Faker()

# Criar dicionário com 10 pares
dict_10 = {fake.name(): fake.email() for _ in range(10)}
chave_10 = list(dict_10.keys())[-1]  # última chave (pior caso hipotético)

# Criar dicionário com 10.000 pares
dict_10000 = {fake.name(): fake.email() for _ in range(10000)}
chave_10000 = list(dict_10000.keys())[-1]

# Teste de tempo para 10 buscas
tempo_10 = timeit.timeit(stmt="chave_10 in dict_10", globals=globals(), number=10000)
tempo_10000 = timeit.timeit(stmt="chave_10000 in dict_10000", globals=globals(), number=10000)

print("🔍 Tempo total para 10.000 buscas:")
print(f"Dicionário com     10 elementos: {tempo_10:.6f} segundos")
print(f"Dicionário com 10.000 elementos: {tempo_10000:.6f} segundos")
```

Dicionários

Limitação	Explicação
Uso de memória	Dict consome mais memória que listas por conta do hash
Colisões do hash	Muitas colisões podem degradar a performance
Acesso somente por chave	Você precisa conhecer a chave exata

Tuplas

- Array que tem tamanho específico e valores Imutáveis
- Úteis quando temos apenas alguns itens que queremos armazenar em uma estrutura de dados, e eles não serão alterados
- É armazenado no cache e não na memória
- Busca é O(1)

Tuplas

```python
# Criando uma tupla com três elementos
coordenadas = (3.5, 7.2, 0.0)
# Exibindo a tupla
print("Tupla:", coordenadas)
# Tentando alterar um valor (isso vai gerar erro!)
try:
    coordenadas[0] = 10
except TypeError as e:
    print("Erro ao tentar alterar a tupla:", e)
# Acessando um valor (busca O(1))
print("Segundo valor (índice 1):", coordenadas[1])
# Verificando se um valor existe (ainda é O(n), mas acesso direto por índice é O(1))
print("Existe o valor 7.2?", 7.2 in coordenadas)
# Tuplas são úteis quando os dados são fixos, ex:
PI = (3.14159,)  # vírgula é obrigatória se for uma tupla com 1 item
RGB_BRANCO = (255, 255, 255)
dias_da_semana = ('segunda', 'terça', 'quarta', 'quinta', 'sexta', 'sábado', 'domingo')
# Tuplas podem ser usadas como chaves em dicionários (por serem imutáveis)
cidades = {
    ('São Paulo', 'SP'): 12_000_000,
    ('Rio de Janeiro', 'RJ'): 6_700_000
}
print("População de São Paulo:", cidades[('São Paulo', 'SP')])
```