Professor
Edson Pedro FERLIN

Estruturas de Dados

Ordenação e Busca

Prof. Edson Pedro Ferlin

Professor
Edson Pedro FERLIN

Estruturas de Dados

AULA 6- Ordenação e Busca

Algoritmos de Ordenação e Busca

Bubble Sort · Quick Sort · Merge Sort

Estruturas de dados · Ordenação e busca · Análise de complexidade

Por que ordenar dados?

ORDENAR ESTÁ EM TODA PARTE

- Agenda telefônica
- Lista de alunos
- Produtos de e-commerce
- Ranking de jogos
- Resultados de pesquisas
- Banco de dados

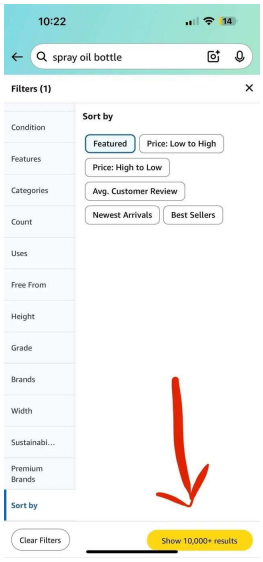
PERGUNTA PARA DISCUSSÃO

É mais fácil encontrar um livro em uma estante organizada ou desorganizada?

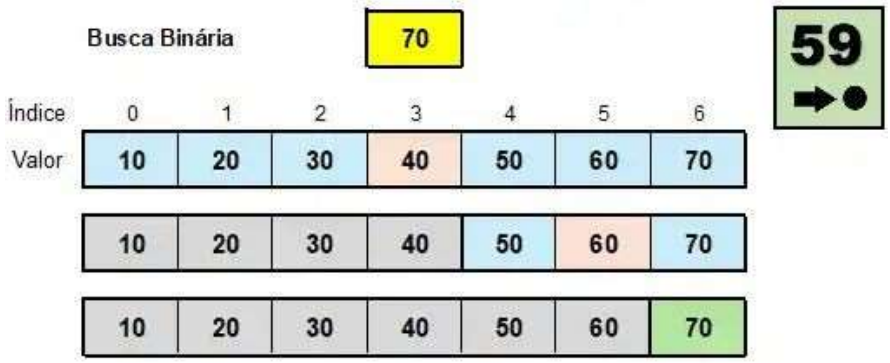
Motivação

- Muitas vezes quando estamos trabalhando com dados desejamos que eles sejam apresentados de maneira ordenada

- Exemplos:
  - Nomes em ordem alfabética
  - Ordenar pelo menor preço
  - Ordenar alunos pelas notas
  - Ordenar arquivos pela data de alteração



- Os algoritmos mais eficientes para realizar buscas em listas são dependentes da ordenação



- Processo de organizar elementos em uma determinada ordem
  - Não se restringe a números
  - Pode ser de forma crescente ou decrescente
- Esse processo é fundamental para eficiência computacional
  - Facilita buscas
  - Organiza informações
  - Melhora desempenho
  - Permite análises mais rápidas

Buscar em dados: por que ordenar ajuda

VETOR ORIGINAL — buscar o número 55

42	10	35	90	18	55

Busca Sequencial

42 → 10 → 35 → 90 → 18 → 55

6 comparações até encontrar o valor

Ordenando primeiro

10	18	35	42	55	90

A Busca Binária reduz drasticamente o número de comparações ao trabalhar sobre dados já ordenados.

Classificação dos algoritmos

Algoritmo	Estratégia
Bubble Sort	Troca
Quick Sort	Divisão e conquista
Merge Sort	Divisão e conquista

Duas grandes famílias

- Trocas — comparam e trocam elementos vizinhos, passo a passo.
- Divisão e conquista — quebram o problema em partes menores e as combinam.

Bubble Sort — conceito

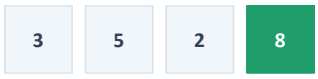
Como funciona

Compara elementos vizinhos e troca suas posições quando estão fora de ordem.

A cada passagem, os maiores elementos "sobem" para o final da lista — como bolhas subindo.

VISUALIZAÇÃO

comparação e troca de vizinhos



o maior elemento fica fixado no fim

Bubble Sort — ilustração passo a passo

Vetor inicial	8	3	5	2
Após a 1ª passagem	3	5	2	**8**
Após a 2ª passagem	3	2	**5**	8
Após a 3ª passagem	2	3	5	8

A cada passagem o maior valor restante é posicionado no fim — até a lista ficar totalmente ordenada.

Bubble Sort — algoritmo

Ideia do algoritmo

1. Enquanto houver trocas
2. Comparar pares vizinhos
3. Trocar quando estiverem fora de ordem

O QUE DESTACAR NA ANIMAÇÃO

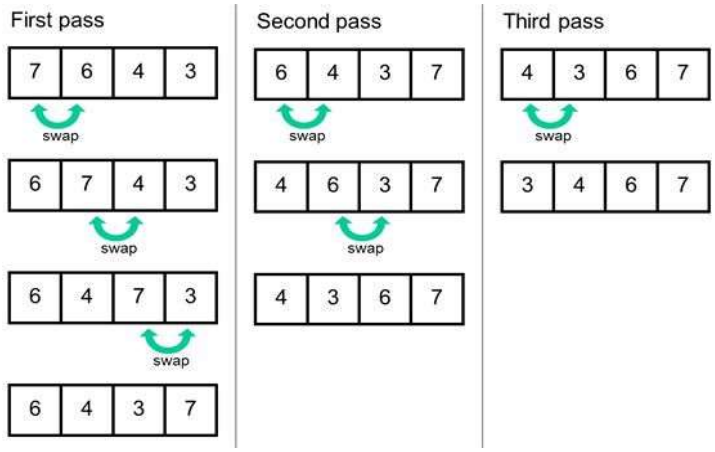
- **Comparação**
- **Troca**
- **Elemento fixado**

Ideia inicial

- Procedimento:
  - Comparar elementos vizinhos
  - Trocar quando necessário
  - Repetir até ordenar
- Analogia:
  - "Os maiores valores sobem como bolhas."
- Esse algoritmo é conhecido como Bubble Sort
- Se trata do algoritmo de ordenação mais intuitivo e simples de se implementar

para $i = 1, \ldots n$ **faça**
  para $j = 1, \ldots n - 1$ **faça**
    se $L[j]\,.\,chave > L[j+1]\,.\,chave$ **então**
      $trocar(L[j], L[j+1])$

Bubble Sort



Funcionamento do Bubble Sort

- O Bubble Sort funciona de maneira que:
  - 1ª passagem → maior vai para o final
  - 2ª passagem → segundo maior se posiciona
  - 3ª passagem → terceiro maior se posiciona
  - ...
- **Desvantagem: Baixa eficiência**
  - Inviável seu uso para grande volume de dados
  - Complexidade $O(n^2)$
- Para melhorar um pouco a eficiência podemos implementar um critério de parada quando a lista já estiver ordenada
  - Não altera a complexidade de pior caso

Bubble Sort em Python

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

Exemplo

```
dados = [8, 3, 5, 2, 9]
print(bubble_sort(dados))
```

Resultado

```
[2, 3, 5, 8, 9]
```


Bubble Sort — complexidade

MELHOR CASO	CASO MÉDIO	PIOR CASO
$O(n)$	$O(n^2)$	$O(n^2)$

Simples de implementar, mas ineficiente para grandes volumes: o custo cresce com o quadrado do número de elementos.

Estratégia - Dividir para conquistar

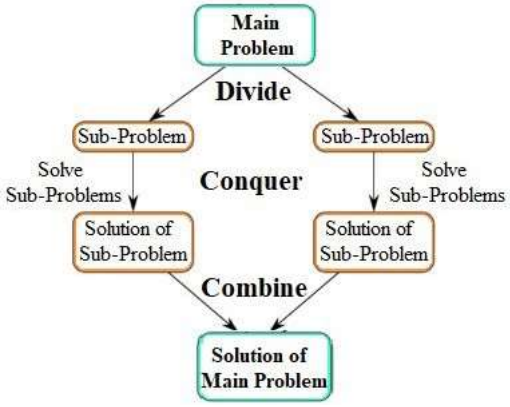
Muitos algoritmos eficientes quebram um problema grande em subproblemas menores, resolvem cada um e depois combinam as respostas.

Dividir	Conquistar	Combinar
quebra em partes menores	resolve cada parte (recursão)	junta as soluções parciais

É a base do Quick Sort e do Merge Sort: dividir bem torna a ordenação muito mais rápida.

Estratégia Dividir para Conquistar

- A estratégia de dividir para conquistar consiste em dividir o problema em problemas mais simples (menores)
- Após resolver os problemas mais simples, eles são combinados para solucionar o problema inicial



QuiCk Sort

- Funcionamento
  - Escolher um elemento chamado pivô
  - Separar os menores e maiores em relação ao pivô
    - Elementos menores para esquerda e maiores para direita
  - Ordenar cada parte recursivamente (repetir o processo para menores e maiores)
- Muito utilizado na prática
- Pior caso depende de escolha do pivô

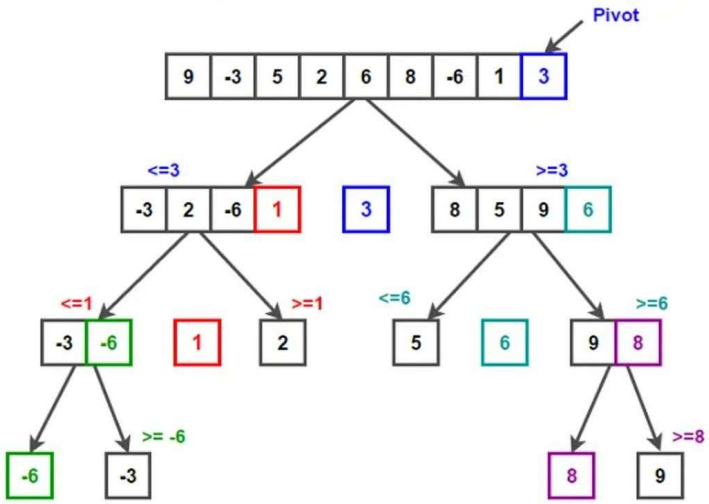
Quick Sort — a ideia

Escolhe um pivô e divide a lista em três partes, ordenando cada parte recursivamente.

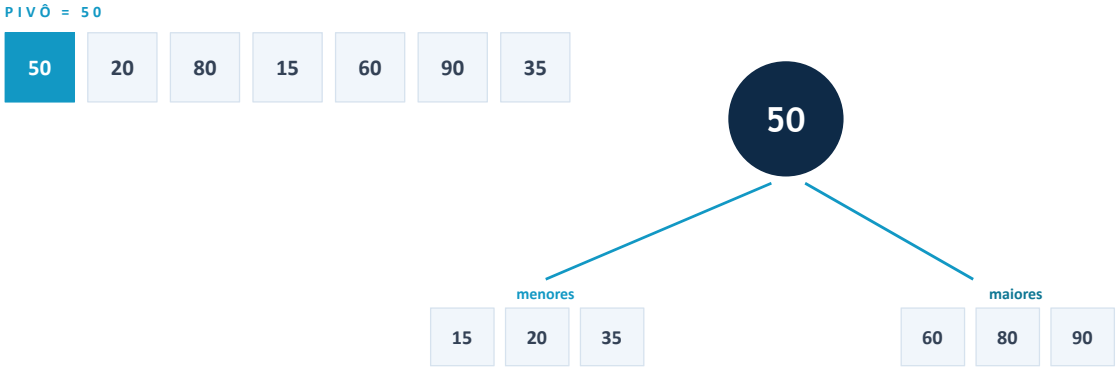
menores	pivô	maiores
valores < pivô	elemento de referência	valores > pivô

Ordena recursivamente cada sublista até que todas tenham um único elemento.

QuiCk Sor



Quick Sort — divisão visual



Quick Sort em Python

```
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    pivo = lista[len(lista) // 2]
    menores = [x for x in lista if x < pivo]
    iguais  = [x for x in lista if x == pivo]
    maiores = [x for x in lista if x > pivo]
    return quicksort(menores) + iguais + quicksort(maiores)
```

Exemplo

```
dados = [8, 3, 5, 2, 9]
print(quicksort(dados))
```

```
[2, 3, 5, 8, 9]
```


Quick Sort — complexidade

MELHOR CASO	CASO MÉDIO	PIOR CASO
$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Muito rápido na prática. O pior caso $O(n^2)$ ocorre com pivôs mal escolhidos em listas já quase ordenadas.

Merge Sort

- Funcionamento
  - Divide a lista ao meio
  - Continua dividindo recursivamente
  - Junta os elementos (merge) ordenando
- Ganha eficiência pois não precisamos comparar todos elementos entre si no momento de fazer o merge
- Muito eficiente e estável em desempenho, porém utiliza muita memória
- Também muito usado na prática

Merge Sort — dividir e mesclar

DIVISÃO SUCESSIVA

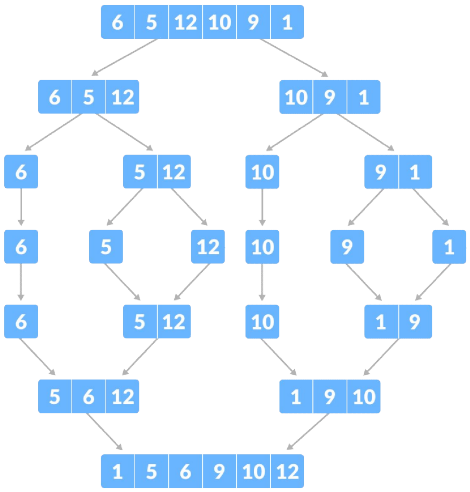
A lista é dividida ao meio repetidamente até restarem elementos individuais; depois eles são mesclados em ordem, formando sublistas cada vez maiores já ordenadas.

Merge Sort — mesclagem

A intercalação combina duas sublistas ordenadas em uma só, sempre escolhendo o menor elemento disponível.

Elementos isolados	8	3		
Primeira mesclagem	3	8		
Outra sublista	2	5		
Resultado final	2	3	5	8

Merge Sort



Merge Sort em Python

```
def mergesort(lista):
    if len(lista) <= 1:
        return lista
    meio = len(lista) // 2
    esq = mergesort(lista[:meio])
    dir = mergesort(lista[meio:])
    return merge(esq, dir)
```

```
def merge(esq, dir):
    resultado = []
    i = j = 0
    while i < len(esq) and j < len(dir):
        if esq[i] < dir[j]:
            resultado.append(esq[i]); i += 1
        else:
            resultado.append(dir[j]); j += 1
    resultado.extend(esq[i:])
    resultado.extend(dir[j:])
    return resultado
```


Merge Sort — complexidade

MELHOR CASO	CASO MÉDIO	PIOR CASO
O(n log n)	O(n log n)	O(n log n)

Desempenho garantido em todos os casos e algoritmo estável — ao custo de memória extra para as sublistas.

Crescimento do número de operações

n	Bubble	Merge	Quick
100	10.000	664	664
1.000	1.000.000	9.966	9.966
10.000	100 milhões	132.877	132.877

Com n = 10.000, o Bubble Sort faz ~100 milhões de operações; Merge e Quick, cerca de 133 mil — uma diferença de quase 1.000 vezes.

Memória e estabilidade

USO DE MEMÓRIA

Algoritmo	Memória
Bubble	Baixa
Quick	Média
Merge	Alta

ESTABILIDADE

Algoritmo	Estável?
Bubble	Sim
Merge	Sim
Quick	Não (tradicional)

Estável significa preservar a ordem relativa de elementos com chaves iguais — importante ao ordenar por múltiplos critérios.

Comparação geral

Característica	Bubble	Quick	Merge
Complexidade média	O(n²)	O(n log n)	O(n log n)
Melhor caso	O(n)	O(n log n)	O(n log n)
Pior caso	O(n²)	O(n²)	O(n log n)
Estável	Sim	Não	Sim
Memória extra	Não	Baixa	Sim
Fácil de implementar	Muito	Média	Média

Comparação dos Algoritmos de Ordenação

Algoritmo	Melhor Caso	Caso Médio	Pior Caso
Bubble Sort	o(n)	o(n²)	o(n²)
Quick Sort	o(nlog(n))	o(nlog(n))	o(n²)
Merge Sort	o(nlog(n))	o(nlog(n))	o(nlog(n))

- https://www.youtube.com/watch?v=ZZuD6iUe3Pc

Busca sequencial

```
def busca(lista, valor):
    for i in lista:
        if i == valor:
            return True
    return False
```

COMPLEXIDADE

O(n)

Percorre todos os elementos, um a um, até encontrar o valor — funciona em qualquer lista, ordenada ou não.

Busca binária — como funciona

PRÉ-REQUISITO a lista precisa estar ordenada.

Buscar 50 em uma lista ordenada — a cada passo, descarta metade dos elementos:

10	20	30	40	50	60	70

1. Compara com o meio (40) → alvo é maior, busca à direita
2. Compara com o meio (60) → alvo é menor, busca à esquerda
3. Compara com o meio (50) → encontrado!

Busca binária em Python

```
def busca_binaria(lista, valor):
    ini = 0
    fim = len(lista) - 1
    while ini <= fim:
        meio = (ini + fim) // 2
        if lista[meio] == valor:
            return True
        elif valor < lista[meio]:
            fim = meio - 1
        else:
            ini = meio + 1
    return False
```

COMPLEXIDADE

O(log n)

A cada passo, o espaço de busca é dividido pela metade.

Sequencial x binária — o impacto

Busca	Complexidade
Sequencial	O(n)
Binária	O(log n)

Ordenar antes de buscar transforma um milhão de passos em cerca de vinte.

Buscar 75 em 1 milhão de registros

SEQUENCIAL

≈ 1.000.000

comparações

BINÁRIA

≈ 20

comparações

Atividade - Ordenação

Conte operações e ordene dados reais aplicando o que você aprendeu

- Conte quantas comparações o Bubble Sort faz para ordenar a lista ao lado.
- Ordene os jogadores pelo número de gols, do maior para o menor.
- Depois, ordene os mesmos jogadores pelo número de títulos.
- Otimize o Bubble com um critério de parada: por que ele ainda funciona?

DADOS DOS EXERCÍCIOS

```
lista = [3, 1, 6, 7, 6, 8, 10, 2, 4]

# nome: (gols, títulos)
jogadores = {
  "Messi":   (850, 44),
  "Ronaldo": (630, 35),
  "Neymar":  (440, 31),
  "Mbappé":  (330, 18),
  "Haaland": (260, 10),
}
```


Conexão com conteúdos anteriores

Vetores	Principal estrutura para implementar os algoritmos de ordenação.
Árvores Binárias de Busca	O percurso em ordem (in-order) já produz os elementos ordenados.
Tabelas Hash	Acesso rápido por chave, mas não mantêm os dados ordenados.

Exercícios

1) Implemente o algoritmo bubble sort
2) Implemente o algoritmo merge sort
3) Implemente o algoritmo quick sort
4) Quantas operações cada um dos 3 algoritmos estudados utiliza para ordenar a seguinte lista?

3,1,6,7,6,8,10,2,4

5) Otimize sua implementação do Bubble Sort executando uma operação a menos a cada rodada. Essa implementação ainda funciona? Por que?

6) Considere o seguinte dicionário:

```
jogadores = {"Messi": {"gols": 850, "time": "Inter Miami", "titulos": 44}, "Cristiano Ronaldo": {"gols": 630, "time": "Al Nassr", "titulos": 35}, "Neymar": {"gols": 440, "time": "Santos", "titulos": 31},"Mbappé": {"gols": 330, "time": "Real Madrid", "titulos": 18}, "Haaland": {"gols": 260, "time": "Manchester City", "titulos": 10}}
```

a) Utilize um algoritmo de ordenação para mostrar os nomes dos jogadores ordenados por gols.
b) Utilize outro algoritmo de ordenação para mostrar os nomes dos jogadores ordenados por títulos.

Exercício desafio

Gere números aleatórios e compare os três algoritmos de ordenação.

10.000	50.000	100.000
números aleatórios	números aleatórios	números aleatórios

Compare

Bubble Sort · **Quick Sort** · **Merge Sort**

Projeto integrador

Comparação experimental de algoritmos de ordenação

- Gere listas aleatórias de tamanhos variados (100, 1.000, 10.000 e 100.000).
- Ordene os dados com Bubble, Quick e Merge Sort.
- Meça o tempo de execução de cada algoritmo com o módulo time.
- Apresente os resultados em uma tabela e em um gráfico comparativo.

MEDINDO O TEMPO

```
import time

inicio = time.perf_counter()
# chamada do algoritmo
fim = time.perf_counter()
print(f"Tempo: {fim - inicio:.6f} s")
```


Sugestão didática: demonstração em sala

Use cartões numerados ou alunos representando os elementos do vetor para simular visualmente cada algoritmo.

Bubble Sort	Quick Sort	Merge Sort
As trocas sucessivas entre elementos vizinhos	A escolha do pivô e a divisão em menores e maiores	O processo de divisão e intercalação das sublistas

A vivência ajuda a compreender os algoritmos antes da implementação em código.

Contato

eferlin@live.com

(BLOG) professorferlin.blogspot.com

(SITE) professorferlin.com.br

(YOUTUBE) ProfEdsonPedroFerlin