Etapa 9 — Busca e Ordenação (Searching and Sorting)

Trilha: Lógica de Programação

Introdução

Em programação, **busca (search)** e **ordenação (sorting)** são operações fundamentais.
Elas permitem **organizar e localizar informações** de forma eficiente dentro de listas, vetores ou bancos de dados.

- **Busca (Search):** encontrar um elemento específico dentro de uma coleção.
- **Ordenação (Sorting):** organizar os elementos em uma determinada ordem (geralmente crescente ou decrescente).

Essas operações aparecem em quase todos os sistemas computacionais — desde **tabelas de cadastro** até **algoritmos de inteligência artificial**.

Tipos de Busca

Existem dois tipos principais de algoritmos de busca:

1. Busca Sequencial (Linear Search)

É o método mais simples.
Percorre a lista elemento por elemento até encontrar o valor procurado.

Pseudocódigo genérico:

```
funcao busca_linear(vetor, valor)
  para i de 1 até tamanho(vetor) faça
    se vetor[i] = valor entao
      retorna i
    fimse
  fimpara
  retorna -1 // não encontrado
```

```
fimfuncao
```

Vantagens:

- Fácil de implementar
- Funciona para listas **não ordenadas**

Desvantagens:

- Ineficiente para listas grandes
- Complexidade: **O(n)**

2. Busca Binária (Binary Search)

A busca binária é muito mais eficiente, mas **exige que a lista esteja ordenada**.
Ela divide o conjunto de dados ao meio a cada passo, descartando metade dos elementos.

Pseudocódigo genérico:

```
funcao busca_binaria(vetor, valor)
  inicio ← 1
  fim ← tamanho(vetor)
  enquanto inicio <= fim faça
    meio ← (inicio + fim) / 2
    se vetor[meio] = valor entao
      retorna meio
    senao se vetor[meio] < valor entao
      inicio ← meio + 1
    senao
      fim ← meio - 1
    fimse
  fimenquanto
```

```
  retorna -1
fimfuncao
```

Vantagens:

- Muito mais rápida: **O(log n)**
- Ideal para grandes volumes de dados ordenados

Desvantagens:

- Requer lista **pré-ordenada**
- Implementação um pouco mais complexa

Tipos de Ordenação (Sorting)

Ordenar significa colocar os dados em uma sequência lógica, como **do menor para o maior**.
Existem muitos algoritmos de ordenação, com diferentes níveis de eficiência e complexidade.

1. Bubble Sort

Compara pares de elementos vizinhos e os troca se estiverem fora de ordem.
Após cada passagem, o maior elemento vai "subindo" até o final da lista.

Pseudocódigo:

```
procedimento bubble_sort(vetor)
  para i de 1 até tamanho(vetor) - 1 faça
    para j de 1 até tamanho(vetor) - i faça
      se vetor[j] > vetor[j + 1] entao
        troque(vetor[j], vetor[j + 1])
      fimse
    fimpara
  fimpara
```

```
fimprocedimento
```

Vantagens: fácil de implementar.

Desvantagem: lento para grandes listas (O(n²)).

2. Selection Sort

Procura o menor valor e o coloca na primeira posição, depois repete o processo com o restante.

Pseudocódigo:

```
procedimento selection_sort(vetor)
  para i de 1 até tamanho(vetor) - 1 faça
    min ← i
    para j de i + 1 até tamanho(vetor) faça
      se vetor[j] < vetor[min] entao
        min ← j
      fimse
    fimpara
    troque(vetor[i], vetor[min])
  fimpara
fimprocedimento
```

Vantagem: poucas trocas.
Desvantagem: ainda é O(n²).

3. Insertion Sort

Constrói a lista ordenada passo a passo, inserindo cada novo elemento na posição correta.

Pseudocódigo:

```
procedimento insertion_sort(vetor)
  para i de 2 até tamanho(vetor) faça
```

```
    chave ← vetor[i]
    j ← i - 1
    enquanto j > 0 e vetor[j] > chave faça
      vetor[j + 1] ← vetor[j]
      j ← j - 1
    fimenquanto
    vetor[j + 1] ← chave
  fimpara
fimprocedimento
```

Vantagem: ótimo para listas pequenas ou quase ordenadas.
Desvantagem: ineficiente em grandes volumes ($O(n^2)$).

4. Merge Sort

Segue o princípio de **dividir e conquistar (divide and conquer)**.
Divide o vetor em duas partes, ordena cada uma e as combina.

Ideia:

1. Divida a lista ao meio.
2. Ordene recursivamente cada metade.
3. Junte as duas partes de forma ordenada.

Complexidade: $O(n \log n)$
Muito eficiente e estável.

5. Quick Sort

Também utiliza **divisão e conquista**, mas escolhe um elemento pivô (pivot) e separa os menores à esquerda e os maiores à direita.

Etapas:

1. Escolher o pivô.

2. Particionar a lista.

3. Aplicar o algoritmo recursivamente.

Complexidade média: O(n log n)

Pior caso: O(n²), se o pivô for mal escolhido.

Comparativo

Algoritmo	Complexidade Média	Estável	Estrutura
Bubble Sort	O(n²)	Sim	Simples
Selection Sort	O(n²)	Não	Simples
Insertion Sort	O(n²)	Sim	Simples
Merge Sort	O(n log n)	Sim	Recursivo
Quick Sort	O(n log n)	Não	Recursivo

Boas Práticas

- Ordene os dados antes de usar **busca binária**
- Prefira algoritmos mais eficientes (Merge / Quick) para grandes listas
- Teste tempos de execução com diferentes tamanhos de entrada
- Analise o tipo de dado e a frequência de uso

Exercícios de Múltipla Escolha

1. A busca binária só pode ser usada quando:
a) A lista é pequena
b) A lista está ordenada

c) O vetor está invertido
d) Há valores duplicados

2. O algoritmo mais simples, porém mais lento para ordenar, é:
a) Quick Sort
b) Merge Sort
c) Bubble Sort
d) Insertion Sort

3. O algoritmo que insere cada elemento na posição correta é:
a) Selection Sort
b) Insertion Sort
c) Bubble Sort
d) Merge Sort

4. O algoritmo com complexidade média **O(log n)** é:
a) Busca Linear
b) Busca Binária
c) Bubble Sort
d) Merge Sort

5. O algoritmo que divide e conquista, juntando as partes ordenadas, é:
a) Insertion Sort
b) Merge Sort
c) Selection Sort
d) Bubble Sort

🧾 Gabarito

1. b) Lista ordenada
2. c) Bubble Sort
3. b) Insertion Sort
4. b) Busca Binária
5. b) Merge Sort