Relatório Avaliação de Ordenação e Pesquisa de Dados

ATD2

Integrantes grupo:

Lucas Fadini Carbonaro
Abner de Oliveira Magalhães
Nichollas Henrique Lázaro Magalhães
João Michelin

Algoritmos: Merge Sort e Quick Sort

Conjunto de Dados:

- **Melhor caso:**
  {1, 10, 13, 14, 29, 37}
- **Caso médio:**
  {29, 10, 14, 37, 13, 1}
- **Pior caso:**
  {37, 29, 14, 13, 10, 1}

Implementação:

- Linguagem de programação de sua preferência (C++).

Codigo disponível no github:

- https://github.com/NichollasMagalhaes/ATD2-Ordenacao-e-Pesquisa-de-dados2025.git

MERGE SORT - DECRESCENTE

```python
# --- MERGE SORT (ordem crescente) ---
def merge_sort(arr):
    if len(arr) > 1:
        meio = len(arr) // 2
        esquerda = arr[:meio]
        direita = arr[meio:]

        merge_sort(esquerda)
        merge_sort(direita)

        i = j = k = 0
        while i < len(esquerda) and j < len(direita):
            if esquerda[i] < direita[j]:
                arr[k] = esquerda[i]
                i += 1
            else:
                arr[k] = direita[j]
                j += 1
            k += 1

        while i < len(esquerda):
            arr[k] = esquerda[i]
            i += 1
            k += 1

        while j < len(direita):
            arr[k] = direita[j]
            j += 1
            k += 1
    return arr
```

```python
# --- VETORES ---
melhor_caso = [1, 10, 13, 14, 29, 37]
caso_medio = [29, 10, 14, 37, 13, 1]
pior_caso = [37, 29, 14, 13, 10, 1]
```

```python
# --- EXECUÇÃO ---
print("=== MERGE SORT (CRESCENTE) ===")
for nome, vetor in [("Melhor caso", melhor_caso),
                    ("Caso médio", caso_medio),
                    ("Pior caso", pior_caso)]:
    print(f"{nome}: {merge_sort(vetor.copy())}")
```

Saida Ordenada:

```
=== MERGE SORT (CRESCENTE) ===
Melhor caso: [1, 10, 13, 14, 29, 37]
Caso médio: [1, 10, 13, 14, 29, 37]
Pior caso: [1, 10, 13, 14, 29, 37]
```

Análise fundamentada e relatório sobre o uso, as vantagens e desvantagens do Merge Sort

COMO FUNCIONA?

O algoritmo divide o vetor em dois vetores menores para utilizar da RECURSÃO que consiste no chamamento da própria função do Merge Sort para que ela resolva os problemas dos vetores menores.

EX: pior caso = [29, 10, 14, 37, 13, 1]
1° - [29, 10, 14] [37, 13, 1]
2° - [29, 10] [14] [37, 13] [1]
3° - [29] [10] [14] [37] [13] [1]
4° - reagrupar os vetores até que eles estejam separados em dois vetores ordenados, para assim serem comparados e juntados.
[10, 14, 29] [1, 13, 37]
5° - faz a comparação dos índices equivalentes de cada vetor ordenando-os em forma crescente.
[10] e [1], [14] e [13], [29] e [37] >> [1, 10, 13, 14, 29, 37]

Vantagens:

Muito rápido na prática: geralmente o mais veloz em dados comuns.

Usa pouca memória: trabalha “no lugar” (in-place), sem precisar de muitos vetores auxiliares.

Ideal para grandes listas em memória.

Desvantagens:

Pior caso ruim: se o pivô for mal escolhido, a complexidade vira $O(n^2)$.

Não é estável: pode alterar a ordem de elementos iguais.

Mais sensível à escolha do pivô.

Quick Sort:- DECRESCENTE

```python
# --- QUICK SORT (ordem decrescente) ---
def quick_sort_decrescente(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivo = arr[0]
        maiores = [x for x in arr[1:] if x > pivo]
        menores = [x for x in arr[1:] if x <= pivo]
        return quick_sort_decrescente(maiores) + [pivo] + quick_sort_decrescente(menores)
```

```python
# --- VETORES ---
melhor_caso = [1, 10, 13, 14, 29, 37]
caso_medio = [29, 10, 14, 37, 13, 1]
pior_caso = [37, 29, 14, 13, 10, 1]
```

```python
# --- EXECUÇÃO ---
print("\n=== QUICK SORT (DECRESCENTE) ===")
for nome, vetor in [("Melhor caso", melhor_caso),
                    ("Caso médio", caso_medio),
                    ("Pior caso", pior_caso)]:
    print(f"{nome}: {quick_sort_decrescente(vetor.copy())}")
```

Saída Ordenada:

```
=== QUICK SORT (DECRESCENTE) ===
Melhor caso: [37, 29, 14, 13, 10, 1]
Caso médio: [37, 29, 14, 13, 10, 1]
Pior caso: [37, 29, 14, 13, 10, 1]
```

Análise fundamentada e relatório sobre o uso, as vantagens e desvantagens do Quick Sort:

COMO FUNCIONA?

Cada etapa o algoritmo escolhe um pivô e divide a lista em partes menores. Essas partes são ordenadas recursivamente, chamando o próprio Quick Sort novamente. No final, tudo é reagrupado de forma ordenada de forma decrescente.

EX: melhor caso = [29, 10, 14, 37, 13, 1]
1º – Escolhe o **pivô = 37**
Maiores: [] nenhum é maior que 37
Menores: [29, 14, 13, 10, 1]
Junta: [37] + quick_sort [29, 14, 13, 10, 1]
2º – Agora ordena [29, 14, 13, 10, 1]
Pivô = 29
Maiores: []
Menores: [14, 13, 10, 1]
Junta: [29] + quick_sort [14, 13, 10, 1]
3º – Ordenando [14, 13, 10, 1]
Pivô = 14
Maiores: []
Menores: [13, 10, 1]
Junta: [14] + quick_sort [13, 10, 1]
4º – Ordenando [13, 10, 1]
Pivô = 13
Maiores: []
Menores: [10, 1]
Junta: [13] + quick_sort [10, 1]
5º – Ordenando [10, 1]
Pivô = 10
Maiores: []
Menores: [1]
Junta: [10] + [1]
6º – Reagrupando os resultados, voltando das chamadas recursivas:
[10, 1]
[13, 10, 1]
[14, 13, 10, 1]
[29, 14, 13, 10, 1]
[37, 29, 14, 13, 10, 1]

Vantagens:

Muito rápido na prática: geralmente o mais veloz em dados comuns.

Usa pouca memória: trabalha “no lugar” (in-place), sem precisar de muitos vetores auxiliares.

Ideal para grandes listas em memória.

Desvantagens:

Pior caso ruim: se o pivô for mal escolhido, a complexidade vira $O(n^2)$.

Não é estável: pode alterar a ordem de elementos iguais.

Mais sensível à escolha do pivô.