UNIVERSIDADE FEDERAL DO TOCANTINS
CÂMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CAIO SANTOS SILVA, GUSTAVO GONZAGA DOS SANTOS E THIAGO GONZAGA DOS SANTOS

ANÁLISE EMPÍRICA DE ALGORITMOS DE ORDENAÇÃO

PALMAS (TO)

2024

RESUMO

Este trabalho realiza uma análise empírica de seis algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. Foram implementados e testados esses algoritmos em listas de diferentes tamanhos (1.000, 10.000, 50.000 e 100.000 elementos), com distribuições ordenadas, inversamente ordenadas e aleatórias. A análise focou na medição do tempo de execução, número de comparações e número de trocas realizadas por cada algoritmo. Os resultados mostraram diferenças significativas de desempenho, destacando as vantagens de algoritmos como o Merge Sort e o Quick Sort em listas grandes e desordenadas, enquanto algoritmos simples como o Insertion Sort mostraram eficiência em listas pequenas ou parcialmente ordenadas. Com base nesses resultados, são discutadas as melhores escolhas de algoritmos dependendo do cenário de aplicação.

Palavra-chave: Algoritmos de Ordenação. Complexidade de Tempo. Análise Empírica. Comparações. Eficiência Computacional.

ABSTRACT

This paper conducts an empirical analysis of six sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort. These algorithms were implemented and tested on lists of varying sizes (1,000, 10,000, 50,000, and 100,000 elements) with ordered, reverse ordered, and random distributions. The analysis focused on measuring execution time, the number of comparisons, and the number of swaps performed by each algorithm. The results demonstrated significant performance differences, highlighting the advantages of algorithms such as Merge Sort and Quick Sort for large, unsorted lists, while simpler algorithms like Insertion Sort showed efficiency in smaller or partially ordered lists. Based on these findings, recommendations are made for the best algorithm choices depending on specific sorting scenarios.

Keywords: Sorting Algorithms. Time Complexity. Empirical Analysis. Comparisons. Computational Efficiency.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Importância da ordenação de dados

A ordenação é amplamente considerada um dos problemas mais fundamentais no estudo de algoritmos. Isso se deve a diversos fatores. Em algumas aplicações, a ordenação é uma necessidade intrínseca; por exemplo, bancos precisam organizar cheques por número para preparar extratos de clientes. Além disso, a ordenação é frequentemente usada como sub-rotina em outros algoritmos. Um exemplo comum é em programas que renderizam objetos gráficos, onde é necessário ordenar os objetos de acordo com a relação de profundidade para desenhá-los corretamente, do fundo para a frente. (CORMEN et al., 2022)

1.2 Objetivo

Este trabalho tem como objetivo comparar seis algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. A análise será realizada tanto do ponto de vista teórico quanto empírico, medindo o tempo de execução, número de comparações e trocas em diferentes listas de tamanhos e distribuições.

2 REVISÃO TEÓRICA

Na ciência da computação, algoritmos de ordenação são essenciais para organizar dados de maneira eficiente. A escolha do algoritmo de ordenação adequado pode ter um impacto significativo no desempenho de programas, especialmente em grandes volumes de dados. Cada algoritmo tem suas próprias características, como a complexidade de tempo, o consumo de memória e a estabilidade, que o tornam mais ou menos adequado para diferentes tipos de problemas. Nesta seção, são discutidos seis algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. A seguir, é apresentada a descrição teórica de cada um deles, com ênfase nas propriedades que influenciam sua eficiência em diferentes contextos.

2.1 Bubble Sort

O Bubble Sort é um algoritmo simples que funciona repetidamente comparando pares adjacentes e trocando-os se estiverem na ordem errada. Sua complexidade é $O(n^2)$ no pior caso e $O(n)$ no melhor caso (quando a lista já está ordenada).

- Complexidade: $O(n^2)$.
- In-place: Sim.
- Estabilidade: Sim.

2.2 Selection Sort

O Selection Sort divide o vetor em duas partes: a parte ordenada e a parte não ordenada. A cada iteração, o menor elemento da parte não ordenada é selecionado e colocado na posição correta. Sua complexidade também é $O(n^2)$.

- Complexidade: $O(n^2)$.
- In-place: Sim.
- Estabilidade: Não.

2.3 Insertion Sort

O Insertion Sort constrói uma lista ordenada um elemento de cada vez. Ele tem complexidade $O(n^2)$ no pior caso, mas pode ser $O(n)$ se a lista já estiver quase ordenada.

- Complexidade: $O(n^2)$.

- In-place: Sim.
- Estabilidade: Sim.

2.4 Merge Sort

O Merge Sort utiliza o método de divisão e conquista para ordenar os elementos. Ele divide a lista em sublistas até que cada sublista tenha um único elemento e, em seguida, as combina. Sua complexidade é $O(n \log n)$.

- Complexidade: $O(n \log n)$.
- In-place: Não.
- Estabilidade: Sim.

2.5 Quick Sort

O Quick Sort também utiliza o método de divisão e conquista. Ele seleciona um "pivô"e particiona os outros elementos em duas sublistas de acordo com se eles são menores ou maiores que o pivô. Sua complexidade média é $O(n \log n)$.

- Complexidade: $O(n \log n)$ no melhor caso, $O(n)$ no pior caso.
- In-place: Sim.
- Estabilidade: Não.

2.6 Heap Sort

O Heap Sort transforma a lista em uma estrutura de heap e então extrai os elementos da heap para formar a lista ordenada. Sua complexidade é $O(n \log n)$.

- Complexidade: $O(n \log n)$.
- In-place: Sim.
- Estabilidade: Não.

3 METODOLOGIA

Para a realização dos testes empíricos, a metodologia adotada inclui a implementação dos algoritmos de ordenação, a geração de listas de diferentes tamanhos e distribuições, e a medição do desempenho dos algoritmos. A seguir, são apresentados os detalhes de cada etapa.

3.1 Ambiente de Teste

- Hardware: AMD Ryzen 7, 64GB memória RAM, Windows 11 64 bits.
- Software: Linguagem de programação Python e bibliotecas .

3.2 Implementação

Para a implementação dos algoritmos de ordenação, foram desenvolvidos dois códigos em Python. O primeiro código é responsável pela geração das listas, enquanto o segundo realiza a ordenação dessas listas utilizando os diferentes algoritmos de ordenação. Além disso, o segundo código calcula o tempo de execução e o número de comparações realizadas durante o processo de ordenação.

As listas para os testes foram geradas com diferentes distribuições (ordenadas, inversamente ordenadas e aleatórias). O código abaixo ilustra o processo de geração das listas e como são salvas em arquivos:

```
1  import random
2
3  def gerar_listas(tamanho):
4      lista_ordenada = list(range(1, tamanho + 1))
5      lista_inversa = list(range(tamanho, 0, -1))
6      lista_aleatoria = lista_ordenada[:]
7      random.shuffle(lista_aleatoria)
8      return lista_ordenada, lista_inversa, lista_aleatoria
9
10 def salvar_listas(tamanho):
11     lista_ordenada, lista_inversa, lista_aleatoria = gerar_listas(
           tamanho)
12     nome_arquivo = f'lista_{tamanho}.txt'
13     with open(nome_arquivo, 'w') as f:
14         f.write(', '.join(map(str, lista_ordenada)) + '\n')
15         f.write(', '.join(map(str, lista_inversa)) + '\n')
16         f.write(', '.join(map(str, lista_aleatoria)) + '\n')
17
18 for tamanho in [1000, 10000, 50000, 100000]:
```

```
19      salvar_listas(tamanho)
20      print(f"Arquivo 'lista_{tamanho}.txt' gerado com sucesso!")
```

Esse código gera três tipos de listas para cada tamanho de dados testado (1.000, 10.000, 50.000 e 100.000 elementos):

- Lista Ordenada: Elementos em ordem crescente.
- Lista Inversamente Ordenada: Elementos em ordem decrescente.
- Lista Aleatória: Elementos dispostos em uma ordem aleatória.

Cada lista foi salva em um arquivo .txt, contendo as três versões (ordenada, inversa e aleatória) para cada tamanho. Esses arquivos serviram de entrada para os algoritmos de ordenação.

Foram implementados seis algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. A implementação de cada algoritmo contabiliza o número de comparações e trocas realizadas, além de medir o tempo de execução. O código para execução dos algoritmos é descrito abaixo.

- Bubble Sort

```
1 def bubble_sort(arr):
2     comparacoes, trocas = 0, 0
3     for i in range(len(arr)):
4         for j in range(0, len(arr) - i - 1):
5             comparacoes += 1
6             if arr[j] > arr[j + 1]:
7                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
8                 trocas += 1
9     return comparacoes, trocas
```

- Selection Sort

```
1 def selection_sort(arr):
2     comparacoes, trocas = 0, 0
3     for i in range(len(arr)):
4         min_idx = i
5         for j in range(i + 1, len(arr)):
6             comparacoes += 1
7             if arr[j] < arr[min_idx]:
8                 min_idx = j
9         if min_idx != i:
10            arr[i], arr[min_idx] = arr[min_idx], arr[i]
11            trocas += 1
12    return comparacoes, trocas
```

- Insertion Sort

```python
def insertion_sort(arr):
    comparacoes, trocas = 0, 0
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        comparacoes += 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            trocas += 1
            j -= 1
            comparacoes += 1
        arr[j + 1] = key
    return comparacoes, trocas
```

- Merge Sort

```python
def merge_sort(arr):
    comparacoes = [0]
    def merge(arr, l, m, r):
        L = arr[l:m + 1]
        R = arr[m + 1:r + 1]
        i = j = k = 0
        while i < len(L) and j < len(R):
            comparacoes[0] += 1
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

    def sort(arr, l, r):
        if l < r:
            m = (l + r) // 2
            sort(arr, l, m)
            sort(arr, m + 1, r)
            merge(arr, l, m, r)
```

```
31
32      sort(arr, 0, len(arr) - 1)
33      return comparacoes[0], 0
```

- Quick Sort

```
1  def quick_sort(arr):
2      comparacoes = [0]
3      def partition(arr, low, high):
4          pivot = arr[high]
5          i = low - 1
6          for j in range(low, high):
7              comparacoes[0] += 1
8              if arr[j] < pivot:
9                  i += 1
10                 arr[i], arr[j] = arr[j], arr[i]
11         arr[i + 1], arr[high] = arr[high], arr[i + 1]
12         return i + 1
13
14     def sort(arr, low, high):
15         if low < high:
16             pi = partition(arr, low, high)
17             sort(arr, low, pi - 1)
18             sort(arr, pi + 1, high)
19
20     sort(arr, 0, len(arr) - 1)
21     return comparacoes[0], 0
```

- Heap Sort

```
1  def heap_sort(arr):
2      comparacoes = [0]
3      def heapify(arr, n, i):
4          largest = i
5          left = 2 * i + 1
6          right = 2 * i + 2
7          if left < n and arr[left] > arr[largest]:
8              largest = left
9          if right < n and arr[right] > arr[largest]:
10             largest = right
11         if largest != i:
12             arr[i], arr[largest] = arr[largest], arr[i]
13             heapify(arr, n, largest)
14
15     n = len(arr)
16     for i in range(n // 2 - 1, -1, -1):
17         heapify(arr, n, i)
```

```
18      for i in range(n - 1, 0, -1):
19          arr[i], arr[0] = arr[0], arr[i]
20          heapify(arr, i, 0)
21
22      return comparacoes[0], 0
```

3.3 Procedimento para Medir o Tempo de Execução, Comparações e Trocas

Os dados foram gerados utilizando a função gerar_listas(tamanho), que cria três tipos de listas para cada tamanho: ordenada, inversamente ordenada e aleatória. Os tamanhos de listas considerados foram 1.000, 10.000, 50.000 e 100.000 elementos.

Os algoritmos de ordenação foram executados em cópias dessas listas. O tempo de execução foi calculado com a função time.time() antes e depois da execução de cada algoritmo, e os valores foram multiplicados por mil para serem apresentados em milissegundos. Além disso, variáveis específicas em cada algoritmo foram usadas para rastrear o número de comparações (interações que comparam dois elementos) e trocas (operações de troca de posições entre elementos).

Os resultados obtidos foram organizados e analisados de maneira comparativa para cada algoritmo em relação ao tamanho da lista e à sua distribuição.

4 RESULTADOS

Nesta seção, serão apresentados os resultados obtidos nos testes de desempenho dos algoritmos de ordenação. Os algoritmos foram executados em listas de diferentes tamanhos (1.000, 10.000, 50.000 e 100.000 elementos), com três tipos de organização: listas ordenadas, inversamente ordenadas e aleatórias. Os tempos de execução, número de comparações e número de trocas foram medidos para cada caso.

4.1 Apresentação dos Resultados

Os testes foram realizados em três tipos de listas: ordenadas, inversamente ordenadas e aleatórias. Abaixo, é apresentada uma síntese dos resultados obtidos para listas de 1.000, 10.000, 50.000 e 100.000 elementos. As tabelas abaixo apresentam os tempos de execução (em milissegundos), o número de comparações e o número de trocas para cada algoritmo.

Tabela 1 – Resultados para Lista de 1.000 Elementos

Algoritmo	**Tipo de Lista**	**Tempo (ms)**	**Comparações**	**Trocas**
Bubble Sort	Ordenada	42.006	499.500	0
Bubble Sort	Inversamente Ordenada	79.014	499.500	499.500
Bubble Sort	Aleatória	54.010	499.500	248.718
Selection Sort	Ordenada	26.006	499.500	0
Selection Sort	Inversamente Ordenada	33.008	499.500	500
Selection Sort	Aleatória	26.013	499.500	994
Insertion Sort	Ordenada	0.000	999	0
Insertion Sort	Inversamente Ordenada	71.015	500.499	499.500
Insertion Sort	Aleatória	29.014	249.717	248.718
Merge Sort	Ordenada	1.999	5.044	0
Merge Sort	Inversamente Ordenada	2.996	4.932	0
Merge Sort	Aleatória	2.007	8.708	0
Quick Sort	Ordenada	89.029	499.500	500.499
Quick Sort	Inversamente Ordenada	75.017	499.500	250.499
Quick Sort	Aleatória	2.001	11.504	6.184
Heap Sort	Ordenada	3.999	17.583	9.708
Heap Sort	Inversamente Ordenada	3.000	15.965	8.316
Heap Sort	Aleatória	4.009	16.877	9.108

Tabela 2 – Resultados para Lista de 10.000 Elementos

Algoritmo	**Tipo de Lista**	**Tempo (ms)**	**Comparações**	**Trocas**
Bubble Sort	Ordenada	3.306.740	49.995.000	0
Bubble Sort	Inversamente Ordenada	7.581.698	49.995.000	49.995.000
Bubble Sort	Aleatória	5.692.274	49.995.000	25.013.104
Selection Sort	Ordenada	2.633.588	49.995.000	0
Selection Sort	Inversamente Ordenada	2.886.655	49.995.000	5.000
Selection Sort	Aleatória	2.603.590	49.995.000	9.988
Insertion Sort	Ordenada	1.007	9.999	0
Insertion Sort	Inversamente Ordenada	6.090.353	50.004.999	49.995.000
Insertion Sort	Aleatória	3.090.692	25.023.103	25.013.104
Merge Sort	Ordenada	20.004	69.008	0
Merge Sort	Inversamente Ordenada	19.009	64.608	0
Merge Sort	Aleatória	24.006	120.481	0
Quick Sort	Ordenada	9.038.023	49.995.000	50.004.999
Quick Sort	Inversamente Ordenada	6.518.456	49.995.000	25.004.999
Quick Sort	Aleatória	26.006	155.790	86.837
Heap Sort	Ordenada	52.004	244.460	131.956
Heap Sort	Inversamente Ordenada	46.010	226.682	116.696
Heap Sort	Aleatória	47.012	235.304	124.124

Tabela 3 – Resultados para Lista de 50.000 Elementos

Algoritmo	**Tipo de Lista**	**Tempo (ms)**	**Comparações**	**Trocas**
Bubble Sort	Ordenada	83.494.682	1.249.975.000	0
Bubble Sort	Inversamente Ordenada	191.703.618	1.249.975.000	1.249.975.000
Bubble Sort	Aleatória	142.511.136	1.249.975.000	621.102.944
Selection Sort	Ordenada	64.881.952	1.249.975.000	0
Selection Sort	Inversamente Ordenada	72.980.335	1.249.975.000	25.000
Selection Sort	Aleatória	66.764.053	1.249.975.000	49.990
Insertion Sort	Ordenada	5.001	49.999	0
Insertion Sort	Inversamente Ordenada	153.862.284	1.250.024.999	1.249.975.000
Insertion Sort	Aleatória	76.812.194	621.152.943	621.102.944
Merge Sort	Ordenada	108.024	401.952	0
Merge Sort	Inversamente Ordenada	111.016	382.512	0
Merge Sort	Aleatória	136.031	718.132	0
Quick Sort	Ordenada	223.883.434	1.249.975.000	1.250.024.999
Quick Sort	Inversamente Ordenada	160.785.406	1.249.975.000	625.024.999
Quick Sort	Aleatória	132.036	903.944	471.616
Heap Sort	Ordenada	304.068	1.455.438	773.304
Heap Sort	Inversamente Ordenada	279.014	1.385.604	718.724
Heap Sort	Aleatória	287.031	1.415.724	742.936

Tabela 4 – Resultados para Lista de 100.000 Elementos

Algoritmo	**Tipo de Lista**	**Tempo (ms)**	**Comparações**	**Trocas**
Bubble Sort	Ordenada	343.701.025	4.999.950.000	0
Bubble Sort	Inversamente Ordenada	783.362.112	4.999.950.000	4.999.950.000
Bubble Sort	Aleatória	586.004.951	4.999.950.000	2.486.512.382
Selection Sort	Ordenada	268.322.085	4.999.950.000	0
Selection Sort	Inversamente Ordenada	301.012.078	4.999.950.000	50.000
Selection Sort	Aleatória	276.631.095	4.999.950.000	99.990
Insertion Sort	Ordenada	12.012	99.999	0
Insertion Sort	Inversamente Ordenada	614.842.174	5.000.049.999	4.999.950.000
Insertion Sort	Aleatória	306.152.193	2.486.562.381	2.486.512.382
Merge Sort	Ordenada	219.047	853.904	0
Merge Sort	Inversamente Ordenada	232.053	809.048	0
Merge Sort	Aleatória	289.079	1.529.123	0
Quick Sort	Ordenada	911.763.242	4.999.950.000	5.000.049.999
Quick Sort	Inversamente Ordenada	654.051.003	4.999.950.000	2.500.049.999
Quick Sort	Aleatória	274.051	1.908.130	998.104
Heap Sort	Ordenada	638.142	3.112.517	1.650.854
Heap Sort	Inversamente Ordenada	593.016	2.964.716	1.540.736
Heap Sort	Aleatória	607.036	3.034.192	1.592.612

4.2 Gráficos Comparativos

Para melhor visualizar o desempenho dos algoritmos, os gráficos a seguir ilustram a comparação de tempo de execução entre os diferentes algoritmos para cada tamanho de lista e tipo de organização.

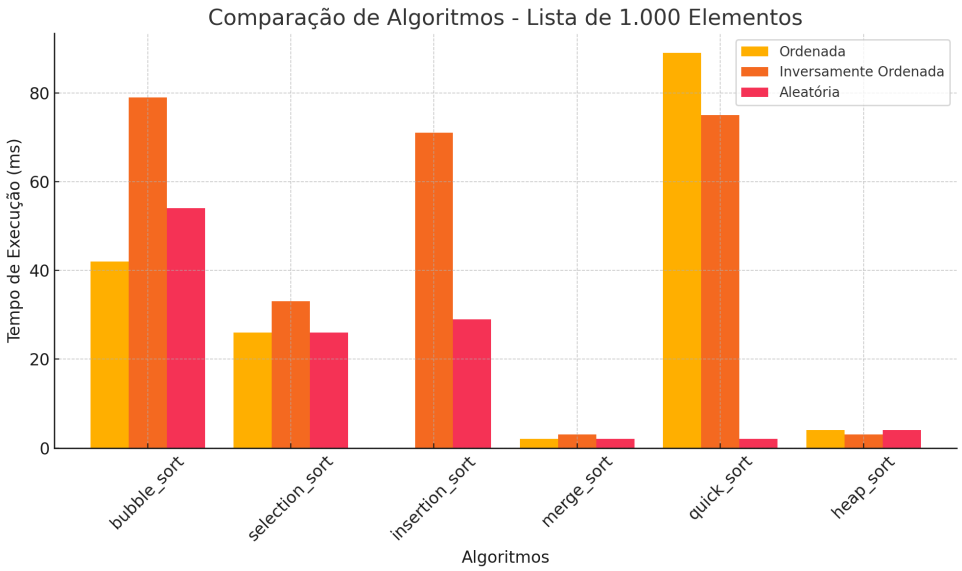

Figura 1 – Gráfico da execução na lista de 1.000 elementos

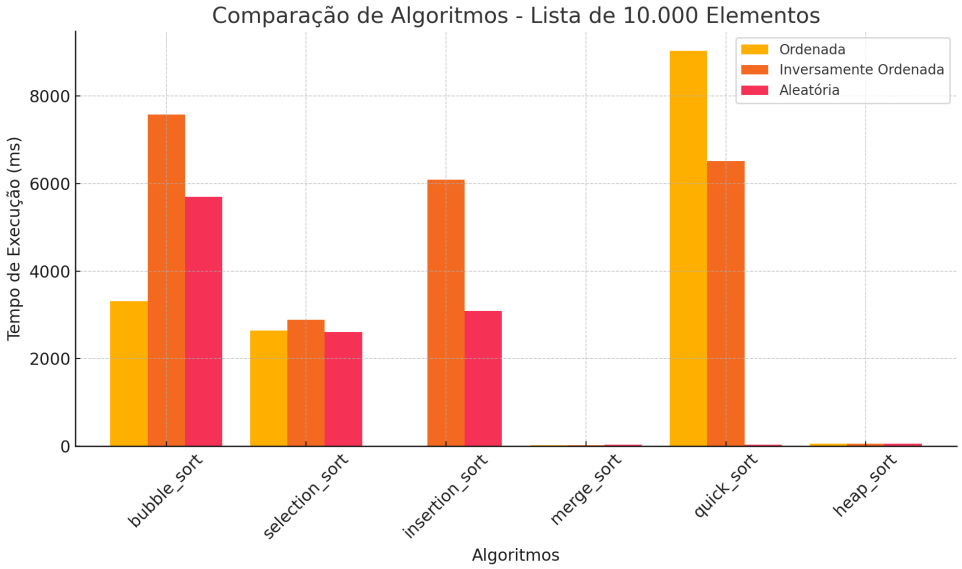

Figura 2 – Gráfico da execução na lista de 10.000 elementos

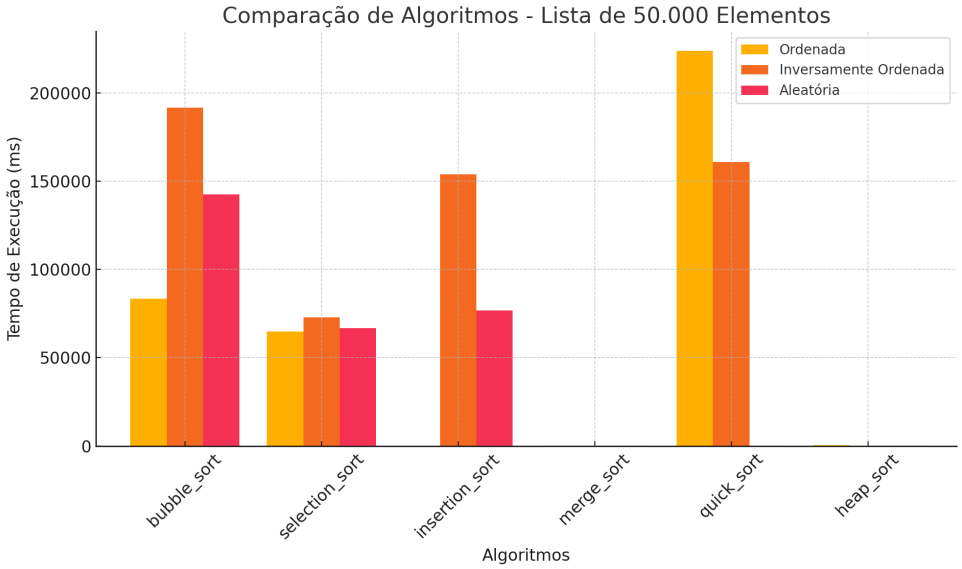

Figura 3 – Gráfico da execução na lista de 50.000 elementos

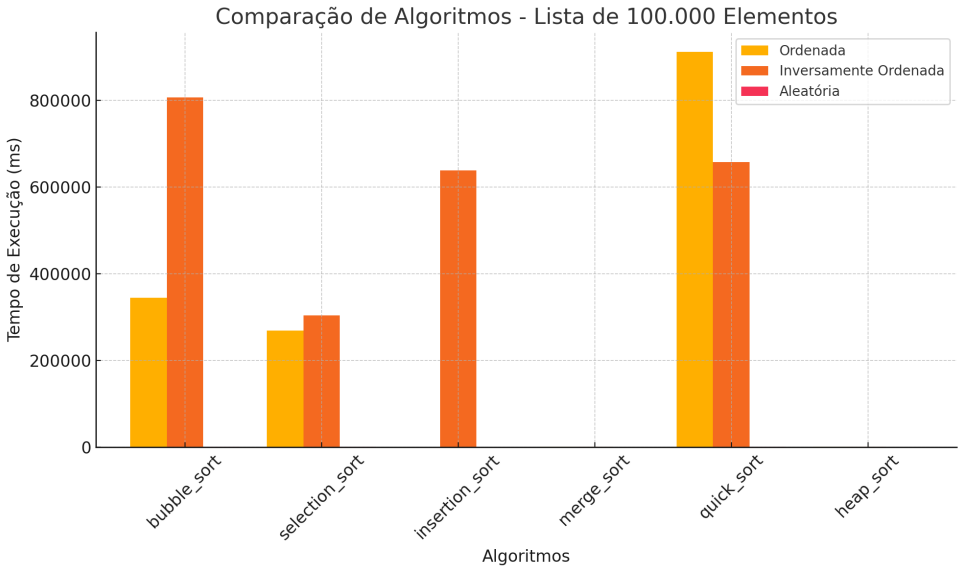

Figura 4 – Gráfico da execução na lista de 100.000 elementos

5 DISCUSSÃO

5.1 Análise

A análise dos resultados mostra que a eficiência dos algoritmos de ordenação varia significativamente conforme o tamanho e a organização das listas. De modo geral, os algoritmos com complexidade $O(n^2)$ (Bubble Sort, Selection Sort e Insertion Sort) apresentaram desempenhos inadequados para listas maiores, especialmente quando estas estavam inversamente ordenadas. Já os algoritmos com complexidade $O(n \log n)$ (Merge Sort, Quick Sort e Heap Sort) demonstraram um desempenho superior, com tempos de execução significativamente menores, especialmente para listas aleatórias.

5.2 Comparação Teórica

Conforme esperado, os algoritmos de complexidade quadrática apresentaram desempenho satisfatório apenas em listas pequenas e parcialmente ordenadas, como o Insertion Sort, que teve excelente desempenho em listas previamente ordenadas. No entanto, quando aplicados a listas maiores ou inversamente ordenadas, esses algoritmos apresentaram uma grande quantidade de comparações e trocas, resultando em tempos de execução extremamente elevados.

O Merge Sort, com sua complexidade $O(n \log n)$, mostrou ser consistentemente eficiente em todos os cenários. O Quick Sort, embora eficiente para listas aleatórias, apresentou resultados piores em listas ordenadas devido ao número de trocas excessivo. O Heap Sort apresentou desempenho estável em todos os testes, destacando-se como uma alternativa eficiente e de fácil implementação.

5.3 Considerações sobre Implementação e Memória

Em termos de facilidade de implementação, o Bubble Sort, Selection Sort e Insertion Sort são algoritmos simples, mas pouco eficientes em grandes volumes de dados. O Merge Sort requer mais memória por ser um algoritmo de ordenação por divisão e conquista, o que pode ser uma desvantagem em ambientes com restrição de recursos. O Quick Sort, embora eficiente em muitos casos, pode ter um comportamento ineficiente em listas já ordenadas ou inversamente ordenadas.

5.4 Limitações do Trabalho e Sugestões Futuros

O trabalho limitou-se a analisar o tempo de execução, o número de comparações e o número de trocas. Em trabalhos futuros, seria interessante explorar o impacto do uso

de memória e realizar análises com outras variações de algoritmos de ordenação, como o Quick Sort com pivôs aleatórios e o Merge Sort in-place. Além disso, investigar o desempenho em diferentes arquiteturas de hardware e ambientes multithread pode ser uma boa sugestão para aprofundar o estudo.

6 CONCLUSÃO

Os resultados deste estudo mostram que algoritmos de complexidade $O(n^2)$, como o Bubble Sort, Selection Sort e Insertion Sort, são ineficazes para listas grandes, especialmente em cenários onde os dados estão desordenados ou inversamente ordenados. Algoritmos com complexidade $O(n \log n)$, como Merge Sort, Quick Sort e Heap Sort, demonstraram um desempenho muito mais eficiente, com destaque para o Merge Sort e Heap Sort em termos de estabilidade e uso de memória. Recomenda-se o uso de algoritmos $O(n \log n)$ para listas grandes ou aleatórias, enquanto algoritmos como o Insertion Sort podem ser mais adequados para listas pequenas ou previamente ordenadas.

REFERÊNCIAS

CORMEN, T. H. et al. **Introduction to Algorithms**. [S.l.]: MIT Press, 2022. 158 p.