Universidade Federal do Tocantins
Ciência da Computação

Projeto e Análise de Algoritmos

Análise e comparação de algoritmos de ordenação

Professor: Warley Gramacho

Autores: Patryck Henryck Moreira Silva, Luis Felipe Krause, Gabryel Soares Marques

Palmas - TO, Outubro de 2024

Conteúdo

1 Introdução

O objetivo deste trabalho é realizar uma análise aprofundada dos principais algoritmos de ordenação, destacando a sua importância no campo da ciência da computação e sua aplicação prática em diversas áreas. A ordenação de dados é um processo central para otimizar a eficiência de vários outros algoritmos e sistemas, como busca de informações, compressão de dados e organização de memória, sendo uma ferramenta indispensável para o funcionamento de sistemas computacionais modernos e grandes volumes de dados. Ao longo do estudo, será possível entender como a escolha do algoritmo correto pode impactar significativamente a performance de aplicações práticas.

Para isso, o trabalho buscará compreender o funcionamento interno de cada algoritmo, detalhando os passos que envolvem suas operações principais, como comparações, trocas e divisões de dados. Será dada atenção especial a como cada algoritmo lida com diferentes tipos de entradas, analisando a eficiência tanto em cenários bons quanto ruins. Através da explicação técnica de cada um, será possível desmistificar como os algoritmos processam dados, facilitando a compreensão de sua lógica interna e estrutura de execução.

Outro aspecto central do trabalho é a análise de complexidade de tempo e espaço de cada algoritmo. Será avaliada a eficiência em termos de número de operações necessárias, levando em consideração os três casos clássicos: melhor, pior e caso médio. Além disso, será feita uma discussão sobre o uso de memória de cada algoritmo, comparando aqueles que necessitam de espaço adicional com aqueles que operam in-place. Isso permitirá uma análise crítica de como esses fatores influenciam o desempenho geral.

O estudo também buscará comparar os algoritmos em diferentes cenários de teste, incluindo listas já ordenadas, desordenadas de maneira aleatória e em ordem inversa. Ao fazer isso, será possível identificar em quais contextos cada algoritmo se destaca, proporcionando insights sobre sua adequação em situações práticas e reais. A ideia é entender quais técnicas de ordenação oferecem melhor desempenho em diferentes tipos de entrada, e como isso se traduz em ganhos de eficiência computacional.

Por fim, o trabalho examinará as vantagens e desvantagens de cada algoritmo, discutindo aspectos como simplicidade de implementação, robustez e escalabilidade. Também serão exploradas as aplicações práticas em que cada algoritmo se adequa melhor, como no processamento de grandes volumes de dados em bancos de dados e redes. O objetivo é fornecer uma visão global dos métodos de ordenação, evidenciando, após análise empírica, o quanto a teoria de complexidade de algoritmos é importante.

Algoritmos de Ordenação

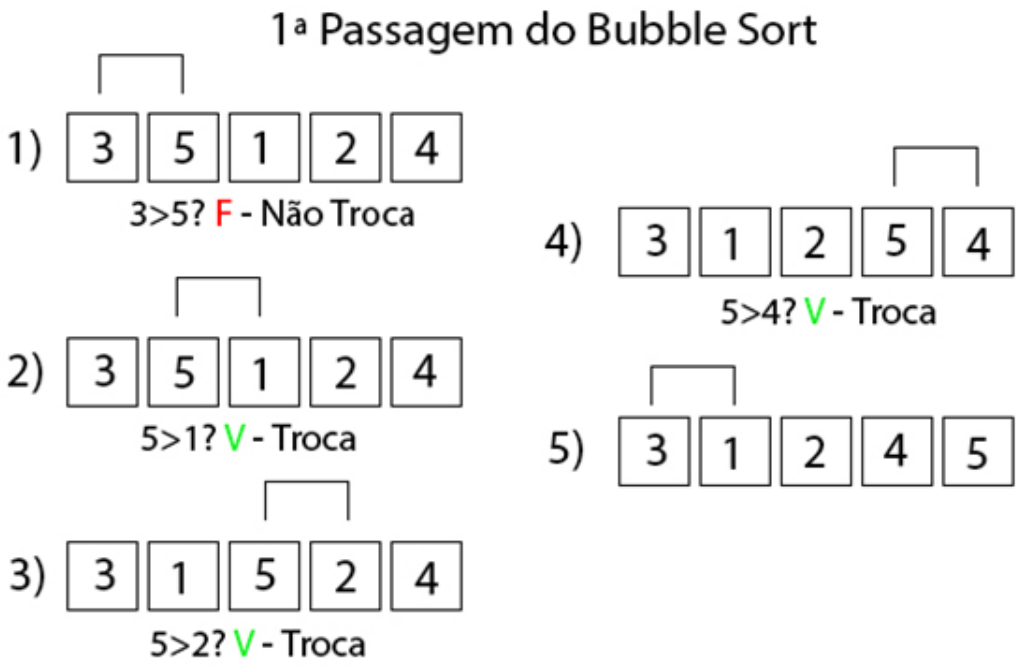
2 Descrição dos Algoritmos

2.1 Bubble Sort

O Bubble Sort é um algoritmo de ordenação simples que compara pares de elementos adjacentes e os troca se estiverem na ordem errada. Esse processo é repetido até que a lista esteja completamente ordenada. A cada passagem, o maior elemento "borbulha"para o fim da lista.

- **Complexidade:** O Bubble Sort possui complexidade de tempo $O(n^2)$ no pior e no caso médio, onde n é o número de elementos da lista. No melhor caso (quando a lista já está ordenada), a complexidade pode ser $O(n)$, desde que uma otimização seja implementada para interromper o algoritmo se não houver trocas em uma passagem.
- **In place:** Sim, o Bubble Sort é um algoritmo in-place, pois realiza a ordenação utilizando apenas uma quantidade constante de memória extra $O(1)$.
- **Estabilidade:** Sim, o Bubble Sort é estável, ou seja, elementos com valores iguais mantêm a ordem relativa original na lista.

  Exemplo de funcionamentos:



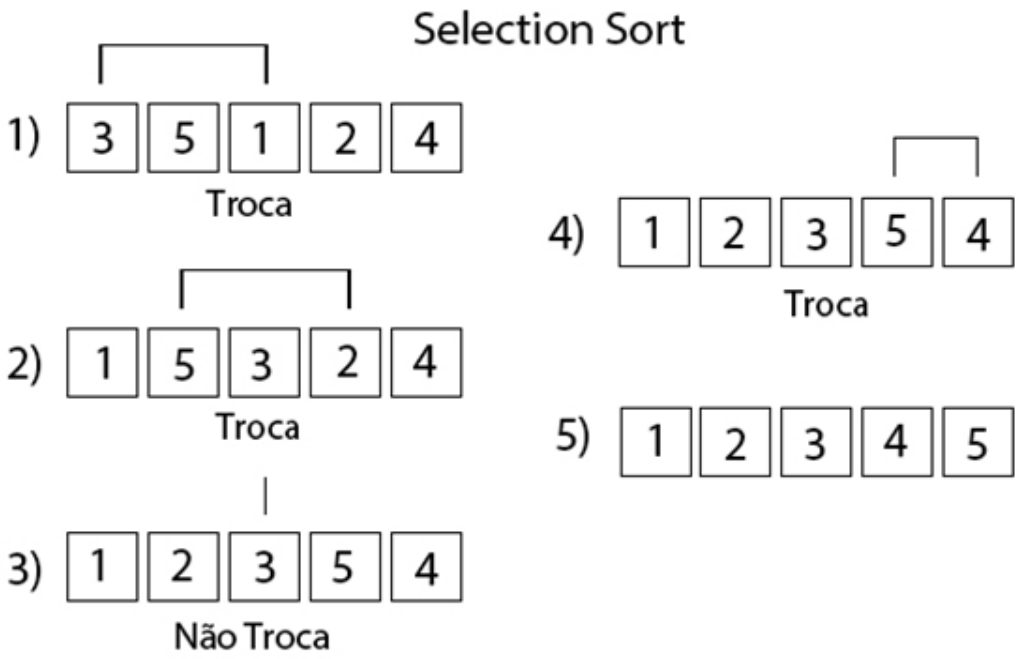
- **Resultado esperado:** Execução ineficiente devido a quantidade de comparações e trocas realizadas.

2.2 Selection Sort

O Selection Sort funciona ao dividir a lista em duas partes: a parte ordenada e a parte não ordenada. Ele encontra o menor elemento da parte não ordenada e o troca com o primeiro elemento da parte não ordenada, movendo o limite entre as duas partes. Este processo se repete até que toda a lista esteja ordenada.

- **Complexidade:** A complexidade de tempo do Selection Sort é $O(n^2)$ tanto no pior quanto no melhor e no caso médio, já que ele realiza $n(n-1)/2$ comparações independentemente da ordenação inicial da lista.
- **In place:** Sim, o Selection Sort é in-place, utilizando $O(1)$ de memória adicional.
- **Estabilidade:** Não, o Selection Sort não é estável, pois ao fazer trocas de elementos, a ordem relativa de elementos iguais pode ser alterada.

  Exemplo de funcionamentos:



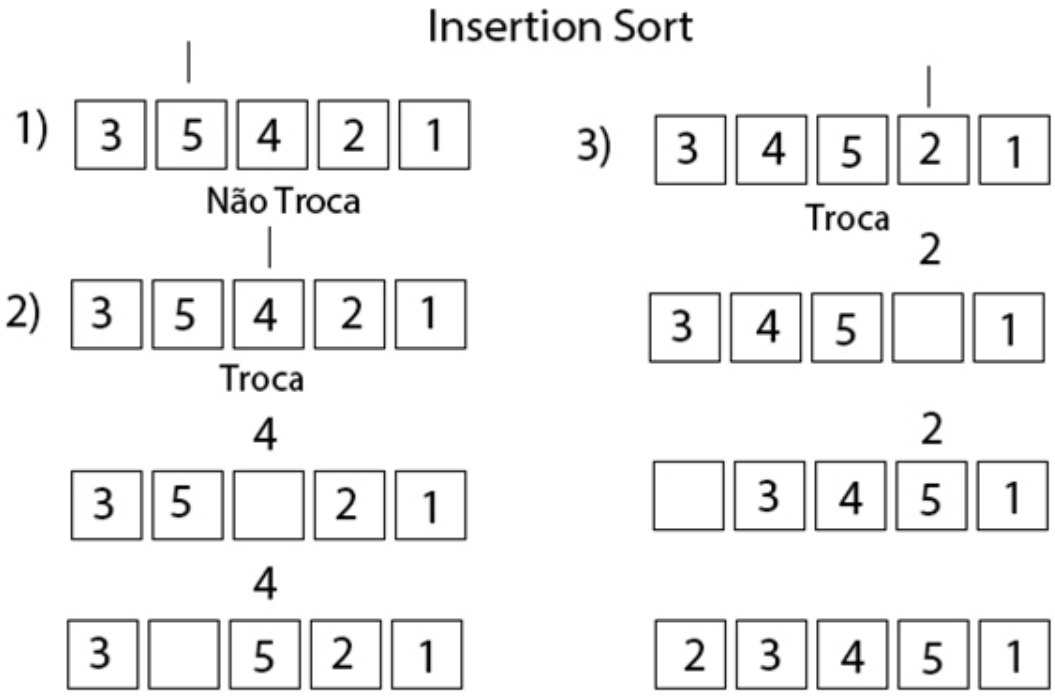
- **Resultado esperado:** Execução lenta para listas inversamente ordenadas devido ao grande número de trocas necessárias.

2.3 Insertion Sort

O Insertion Sort constrói a lista ordenada um elemento por vez, inserindo cada novo elemento em sua posição correta na lista ordenada. É comparado ao modo como as cartas de um baralho são ordenadas à medida que novas cartas são adicionadas.

- **Complexidade:** No pior caso e no caso médio, o Insertion Sort tem uma complexidade de tempo de $O(n^2)$, enquanto no melhor caso (lista já ordenada) ele tem uma complexidade de $O(n)$.
- **In place:** Sim, o Insertion Sort é in-place, com uso de memória extra $O(1)$.
- **Estabilidade:** Sim, o Insertion Sort é estável, já que elementos iguais mantêm sua ordem relativa na lista.

  Exemplo de funcionamentos:



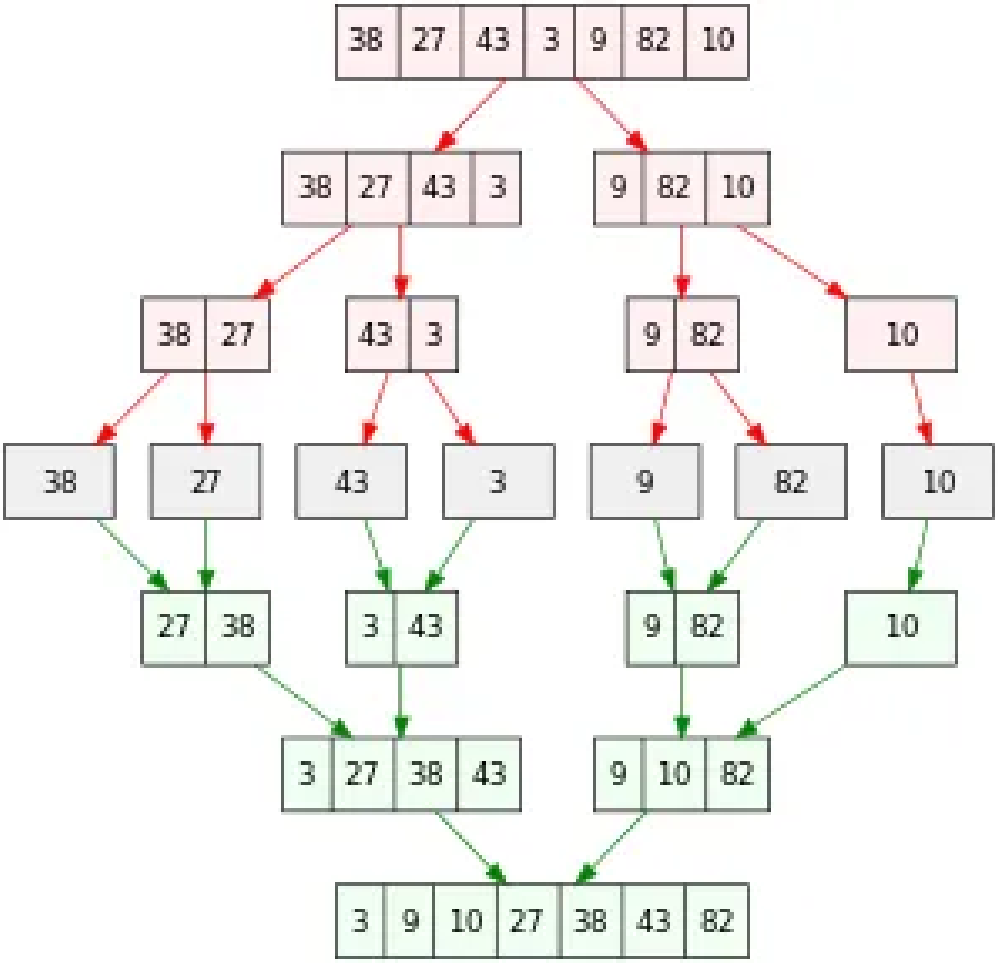
- **Resultado esperado:** Execução muito rápida para listas ordenadas ou parcialmente ordenadas, porém execução extremamente lenta para listas inversamente ordenadas.

2.4 Merge Sort

O Merge Sort é um algoritmo de ordenação baseado na técnica de divisão e conquista. Ele divide a lista em sublistas menores até que cada sublista tenha apenas um elemento, e então as combina (merge) de volta em ordem crescente.

- **Complexidade:** A complexidade de tempo do Merge Sort é $O(n \log n)$ no pior, melhor e caso médio, devido à necessidade de dividir a lista $\log n$ vezes e realizar n operações para mesclar as sublistas.
- **In place:** Não, o Merge Sort não é in-place, pois ele requer memória extra para armazenar as sublistas temporárias durante o processo de mesclagem, com complexidade de espaço adicional $O(n)$.
- **Estabilidade:** Sim, o Merge Sort é estável, preservando a ordem relativa de elementos iguais.

  Exemplo de funcionamentos:



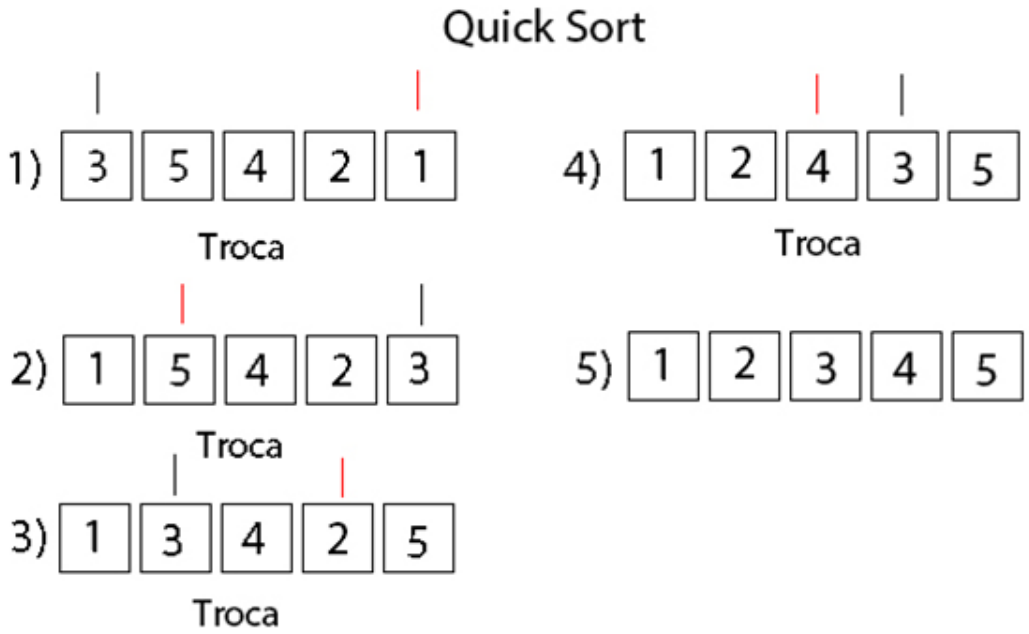
- **Resultado esperado:** Visto que sempre separa o vetor e faz comparação de cada item, se espera uma execução diretamente proporcional ao tamanho da entrada, independente se a lista está ordenada, inversamente ordenada ou aleatória.

2.5 Quick Sort

O Quick Sort é um algoritmo eficiente de ordenação que também utiliza a técnica de divisão e conquista. Ele seleciona um pivô, rearranja os elementos para que os menores fiquem à esquerda e os maiores à direita, e recursivamente aplica o mesmo procedimento nas sublistas geradas.

- **Complexidade:** A complexidade de tempo do Quick Sort é $O(n \log n)$ no caso médio e no melhor caso, mas pode ser $O(n^2)$ no pior caso, que ocorre quando o pivô escolhido é o menor ou maior elemento da lista, resultando em partições desequilibradas.
- **In place:** Sim, o Quick Sort é in-place, utilizando $O(\log n)$ de memória para as chamadas recursivas.
- **Estabilidade:** Não, o Quick Sort não é estável, pois pode alterar a ordem relativa de elementos iguais durante o processo de partição.

  Exemplo de funcionamentos:



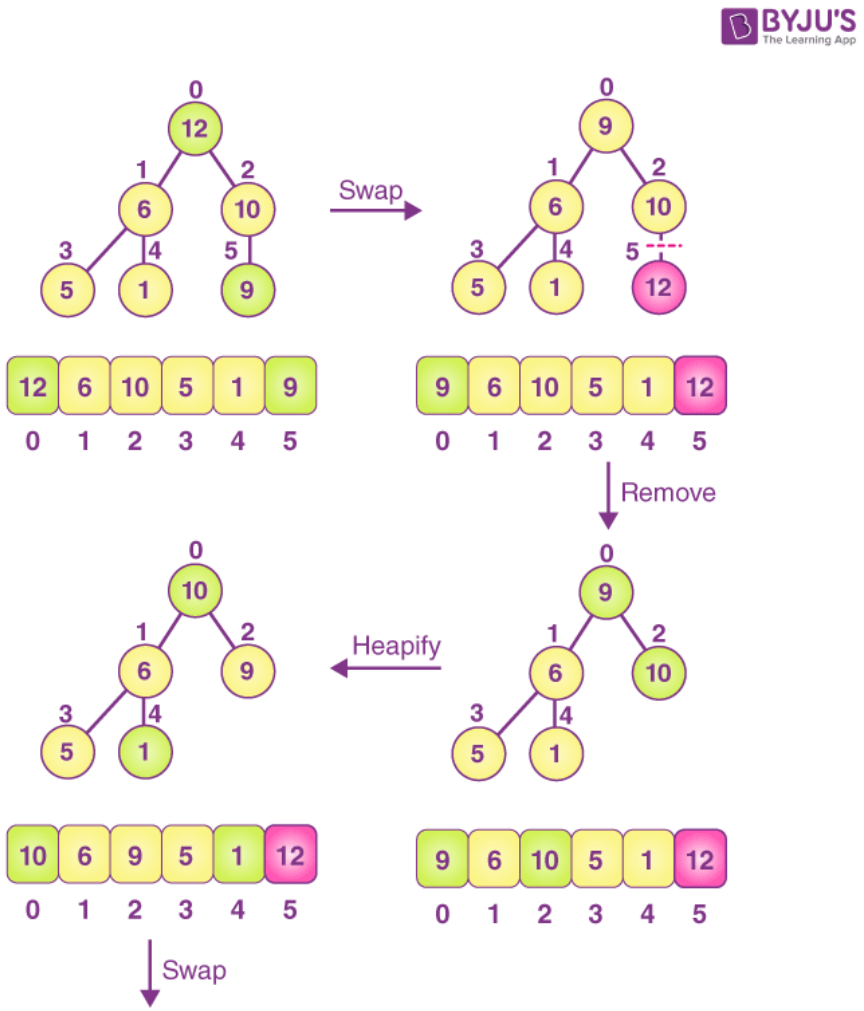
- **Resultado esperado:** Visto que utiliza uma das extremidades como pivô, se espera uma execução lenta em casos como a lista estando ordenada ou inversamente ordenada, mas aparenta ter uma boa eficiência em listas aleatórias.

2.6 Heap Sort

O Heap Sort utiliza uma estrutura de dados chamada heap binário para ordenar os elementos. O algoritmo constrói um heap máximo (onde o maior elemento é a raiz) a partir da lista e, então, remove o maior elemento repetidamente, reconstruindo o heap a cada remoção.

- **Complexidade:** A complexidade de tempo do Heap Sort é $O(n \log n)$ no pior, melhor e caso médio, devido ao processo de construção do heap e as remoções subsequentes, que têm custo logarítmico.
- **In place:** Sim, o Heap Sort é in-place, pois a estrutura de heap pode ser construída utilizando apenas a lista original, com memória extra $O(1)$.
- **Estabilidade:** Não, o Heap Sort não é estável, já que a reorganização dos elementos no heap pode mudar a ordem relativa de elementos iguais.

  Exemplo de funcionamentos:



- **Resultado esperado:** Como utiliza uma estrutura auxiliar baseado em árvore se espera que seja construida uma hierarquia a qual minimize a quantidade de comparações ou trocas. Portanto se espera uma execução razoavelmente eficiente para todos os casos.

3 Descrição do Ambiente de Teste

3.1 Hardware

- Sistema Operacional: Microsoft Windows 11 Home Single Language (Versão 10.0.22631 Compilação 22631)
- Fabricante do sistema: Acer
- Modelo do sistema: Nitro AN515-55
- Processador: Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz, 4 núcleos, 8 threads
- Memória RAM: 8 GB
- Armazenamento: 512 GB SSD
- Placa de Vídeo: GeForce GTX 1650
- Versão da BIOS: Insyde Corp. V2.06
- Modo da BIOS: UEFI
- Estado da Inicialização Segura: Desativado

3.2 Software

Para realizar os testes dos algoritmos de ordenação, foi desenvolvido um script em Python que implementa cada um dos algoritmos estudados, além de um módulo para medir o tempo de execução, o número de comparações e o número de trocas realizadas. O ambiente de desenvolvimento utilizado foi o Python 3.10, que é amplamente utilizado para programação e prototipagem rápida devido à sua simplicidade e legibilidade.

Configuração do Ambiente

As bibliotecas padrão do Python foram utilizadas para a implementação dos algoritmos, incluindo:

- `time`: utilizada para medir o tempo de execução dos algoritmos.
- `random`: utilizada para gerar listas aleatórias para os testes.

O código foi estruturado de maneira a permitir a fácil inclusão e execução de diferentes algoritmos de ordenação, além de permitir a variação do tamanho da lista a ser ordenada.

Testes Realizados

Os testes foram realizados em três cenários diferentes para cada algoritmo de ordenação:

1. **Lista Ordenada:** Uma lista que já está ordenada em ordem crescente, permitindo observar o desempenho em um caso ideal.
2. **Lista Inversamente Ordenada:** Uma lista que está ordenada em ordem decrescente, o que representa o pior caso para alguns algoritmos de ordenação.

3. **Lista Aleatória:** Uma lista com elementos dispostos aleatoriamente, que serve como um caso médio para avaliar o desempenho dos algoritmos.

Os resultados de cada teste foram registrados, incluindo o tempo de execução, o número total de comparações e o número total de trocas realizadas durante a execução de cada algoritmo. Esses dados foram utilizados para analisar e comparar a eficiência de cada método de ordenação.

Procedimento para Medir o Desempenho

Para implementar os algoritmos de ordenação e realizar a análise de desempenho, utilizaremos uma abordagem que envolve a contagem do número de comparações e trocas, além da medição do tempo de execução para cada algoritmo. Isso nos permitirá uma avaliação precisa da eficiência de cada método.

Para medir o tempo de execução dos algoritmos de ordenação, utilizamos a função `time` da biblioteca `time` do Python. Essa função permite calcular o tempo total gasto para ordenar uma lista de elementos. O número de comparações e trocas é rastreado por variáveis que são incrementadas a cada operação relevante dentro do algoritmo.

O procedimento de medição segue os seguintes passos:

1. **Inicialização:** Uma lista de números aleatórios é gerada com um tamanho predefinido, simulando diferentes cenários de entrada, como listas ordenadas, inversamente ordenadas e aleatórias.

2. **Execução do Algoritmo:** O algoritmo de ordenação é executado sobre essa lista, enquanto o tempo de início e término é registrado para calcular o tempo total de execução.

3. **Contagem de Operações:** Durante a execução, variáveis de contagem para comparações e trocas são atualizadas a cada vez que uma comparação ou troca é realizada.

4. **Registro dos Resultados:** Ao final da execução, o tempo total, o número de comparações e o número de trocas são exibidos, permitindo a análise de desempenho do algoritmo.

4 Implementação dos Algoritmos

4.1 Bubble Sort

O código a seguir apresenta a implementação do algoritmo Bubble Sort, onde monitoramos as comparações e trocas realizadas:

```
import time

def bubble_sort(arr):
    n = len(arr)
    comparisons = 0
    swaps = 0
    for i in range(n):
        for j in range(0, n-i-1):
            comparisons += 1
            if arr[j] > arr[j+1]:
```

```
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swaps += 1
    return comparisons, swaps
```

4.2 Selection Sort

A seguir, apresentamos a implementação do algoritmo Selection Sort, que ordena uma lista de elementos selecionando repetidamente o menor elemento da parte não ordenada e movendo-o para a parte ordenada. O código também monitora o número de comparações e trocas realizadas durante a execução do algoritmo.

```
from random import shuffle
from time import time

def selection_sort(lista):
    n = len(lista)
    comparacoes = 0  # Inicializa o contador de comparações
    trocas = 0       # Inicializa o contador de trocas
    for i in range(n):
        # Encontra o menor elemento na sublista [i:n]
        indice_minimo = i
        for j in range(i + 1, n):
            comparacoes += 1  # Incrementa o contador de comparações
            if lista[j] < lista[indice_minimo]:
                indice_minimo = j

        # Troca o menor elemento encontrado com o primeiro elemento não ord
        if indice_minimo != i:  # Verifica se uma troca é necessária
            lista[i], lista[indice_minimo] = lista[indice_minimo], lista[i]
            trocas += 1  # Incrementa o contador de trocas

    return lista, comparacoes, trocas  # Retorna a lista ordenada, número d
```

4.3 Insertion Sort

A seguir, apresentamos a implementação do algoritmo Insertion Sort. Este algoritmo insere cada elemento da lista na sua posição correta, construindo gradualmente uma lista ordenada. O código também monitora o número de comparações e trocas realizadas durante a execução do algoritmo.

```
import time

def insertion_sort(arr):
    comparisons = 0
    swaps = 0

    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
```

```
        # Move elements of arr[0..i-1] that are greater than key
        # to one position ahead of their current position
        while j >= 0 and key < arr[j]:
            comparisons += 1
            arr[j + 1] = arr[j]
            j -= 1
            swaps += 1
        comparisons += 1  # for the failed comparison when exiting the whi
        arr[j + 1] = key

    return comparisons, swaps
```

4.4 Merge Sort

A seguir, apresentamos a implementação do algoritmo Merge Sort, que utiliza a técnica de divisão e conquista para ordenar uma lista. O código também monitora o número de comparações e trocas realizadas durante a execução do algoritmo.

```
from time import time
from random import shuffle

def merge_sort(lista, contador_comparacoes):
    if len(lista) > 1:
        meio = len(lista) // 2  # Encontra o meio da lista
        metade_esquerda = lista[:meio]  # Divide a lista na metade esquerda
        metade_direita = lista[meio:]   # Divide a lista na metade direita

        # Recursivamente divide as duas metades
        merge_sort(metade_esquerda, contador_comparacoes)
        merge_sort(metade_direita, contador_comparacoes)

        # Índices para as sublistas
        i = j = k = 0

        # Ordena as duas metades
        while i < len(metade_esquerda) and j < len(metade_direita):
            contador_comparacoes += 1  # Incrementa o contador de comparaçõ
            if metade_esquerda[i] < metade_direita[j]:
                lista[k] = metade_esquerda[i]
                i += 1
            else:
                lista[k] = metade_direita[j]
                j += 1
            k += 1
            trocas += 1

        # Verifica se ainda há elementos na metade esquerda
        while i < len(metade_esquerda):
            lista[k] = metade_esquerda[i]
```

```
            i += 1
            k += 1
            trocas += 1

        # Verifica se ainda há elementos na metade direita
        while j < len(metade_direita):
            lista[k] = metade_direita[j]
            j += 1
            k += 1
            trocas += 1
    return lista, contador_comparacoes  # Retorna a lista ordenada e o núme
```

4.5 Quick Sort

A seguir, apresentamos a implementação do algoritmo Quick Sort, que utiliza a técnica de divisão e conquista para ordenar uma lista. O código também monitora o número de comparações realizadas durante a execução do algoritmo.

```
from time import time
from random import shuffle

def quicksort(arr):
    if len(arr) <= 1:
        return arr, 0  # Se a lista tiver 1 ou menos elementos, não há comp

    pivot = arr[0]
    left = []
    right = []
    comparacoes = 0  # Inicializa o contador de comparações

    for x in arr[1:]:
        comparacoes += 1  # Incrementa o contador de comparações
        if x < pivot:
            left.append(x)
        else:
            right.append(x)

    left_sorted, left_comparacoes = quicksort(left)
    right_sorted, right_comparacoes = quicksort(right)

    # Total de comparações feitas
    total_comparacoes = comparacoes + left_comparacoes + right_comparacoes
    return left_sorted + [pivot] + right_sorted, total_comparacoes  # Reto
```

4.6 Heap Sort

A seguir, apresentamos a implementação do algoritmo Heap Sort, que realiza a ordenação de uma lista de elementos utilizando a estrutura de heap binário. O código também monitora as comparações e trocas realizadas durante a execução do algoritmo.

```
import time

def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    comparisons = 0
    swaps = 0

    comparisons += 1
    if left < n and arr[i] < arr[left]:
        largest = left

    comparisons += 1
    if right < n and arr[largest] < arr[right]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        swaps += 1
        additional_comparisons, additional_swaps = heapify(arr, n, largest)
        comparisons += additional_comparisons
        swaps += additional_swaps

    return comparisons, swaps

def heap_sort(arr):
    n = len(arr)
    total_comparisons = 0
    total_swaps = 0

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        comparisons, swaps = heapify(arr, n, i)
        total_comparisons += comparisons
        total_swaps += swaps

    # Extract elements one by one
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        total_swaps += 1
        comparisons, swaps = heapify(arr, i, 0)
        total_comparisons += comparisons
        total_swaps += swaps

    return total_comparisons, total_swaps
```

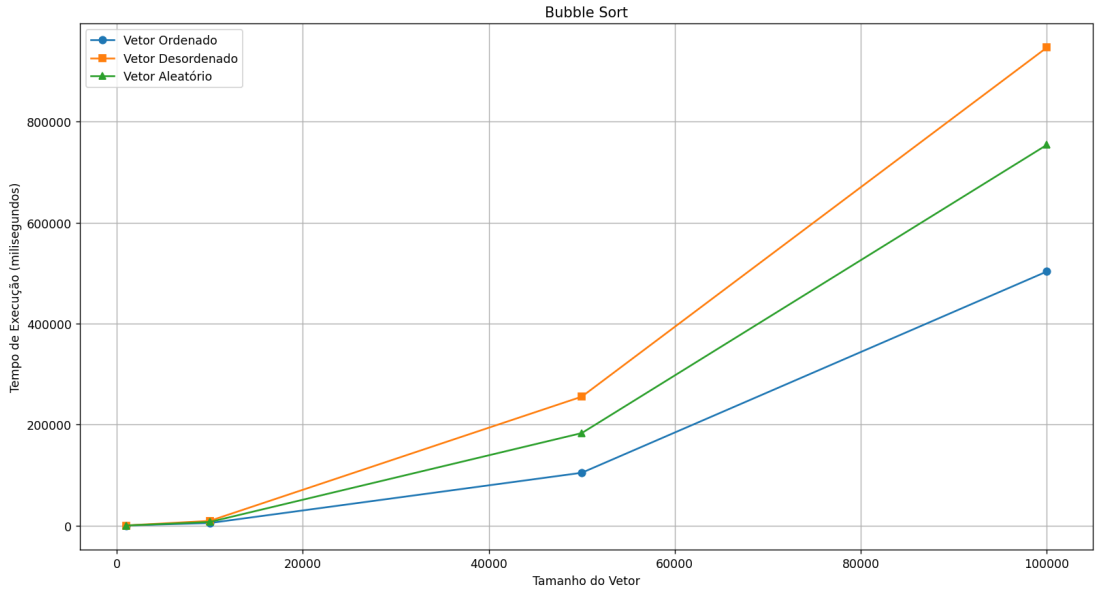
Resultados

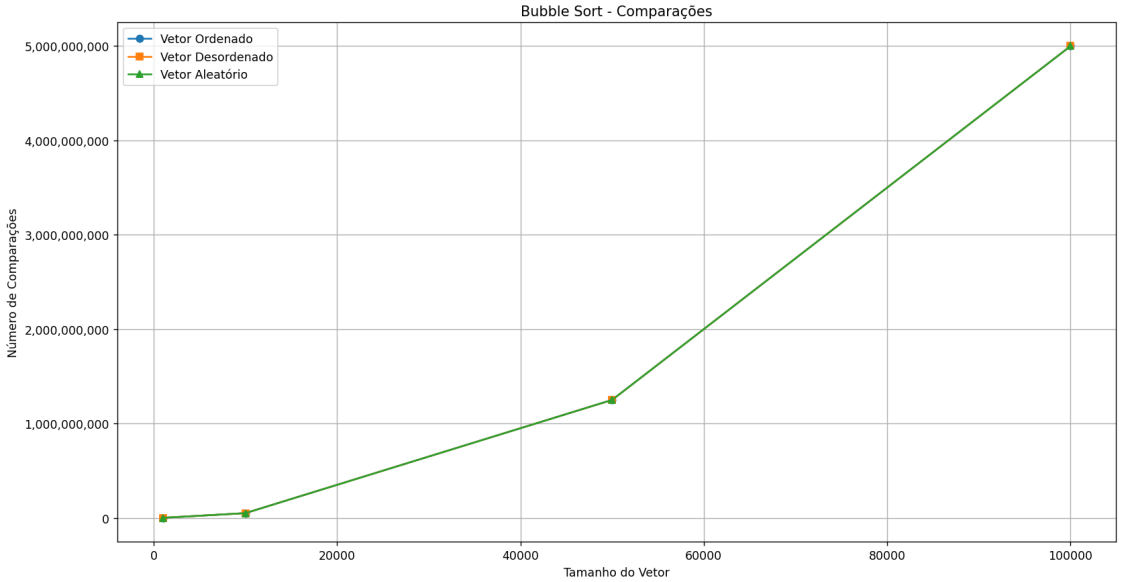
Bubble Sort

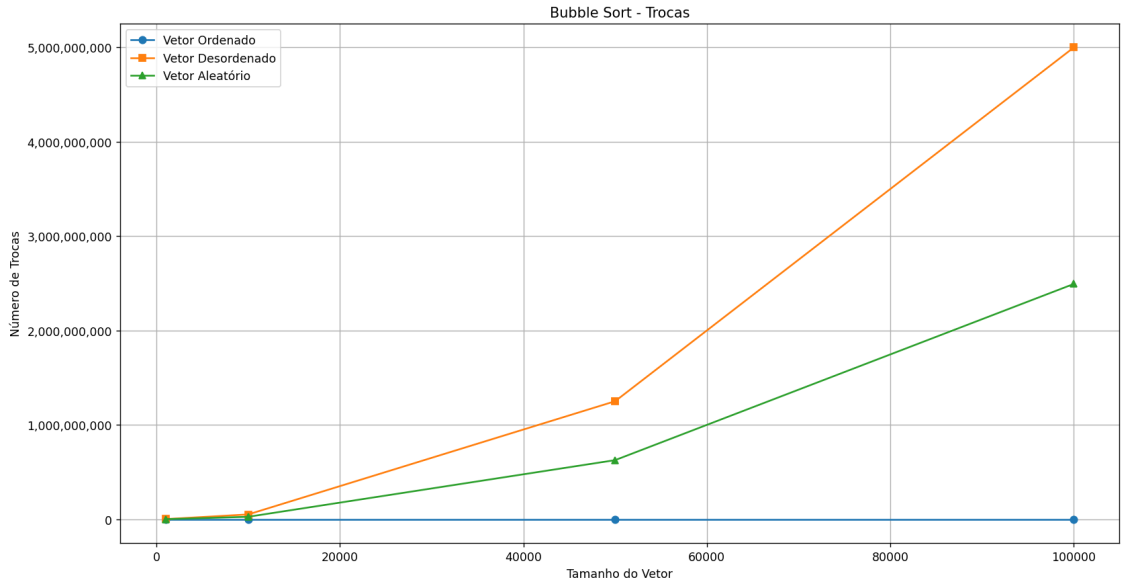
Tipo de Lista	Tamanho da Lista	Tempo de Execução (ms)	Comparações	Trocas
Ordenada	1.000	48.00	499500	0
	10.000	4952.43	49995000	0
	50.000	104474.13	1249975000	0
	100.000	502986.18	4999950000	0
Inv-Ordenada	1.000	100.03	499500	499500
	10.000	9191.03	49995000	49995000
	50.000	255175.62	1249975000	1249975000
	100.000	946506.68	4999950000	4999950000
Aleatória	1.000	58.00	499500	249320
	10.000	7004.00	49995000	25105282
	50.000	183060.39	1249975000	625609651
	100.000	753795.36	4999950000	2493252675

Pelos resultados percebemos que:

O melhor caso é quando a lista está ordenada, neste caso o algoritmo apenas realiza as comparações, o pior caso é quando a lista está ordenada inversamente, neste caso o algoritmo realiza o máximo de comparações e trocas, e no caso da lista ordenada aleatóriamente o algoritmo pode realizar menos trocas que o ordenado inversamente e assim ter um tempo menor, porém ainda é ineficiente. Portanto temos que seu uso é recomendado apenas para listas ordenadas e de tamanho pequeno


Bubble Sort
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Tempo de Execução (milisegundos)
0
200000
400000
600000
800000
0
20000
40000
60000
80000
100000
Tamanho do Vetor


Bubble Sort - Comparações
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Comparações
0
1,000,000,000
2,000,000,000
3,000,000,000
4,000,000,000
5,000,000,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor

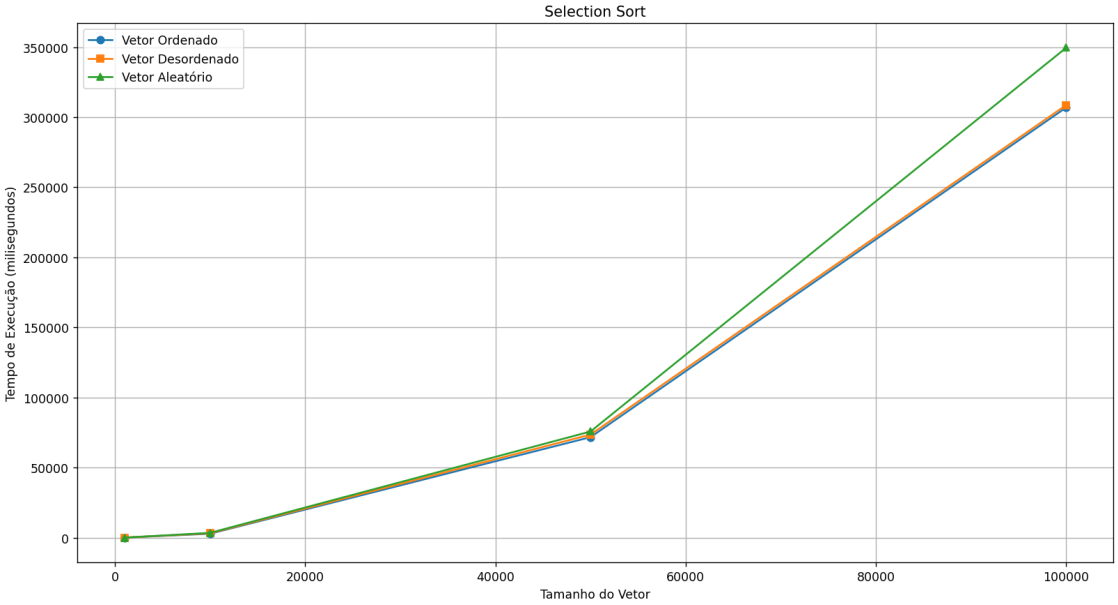

Bubble Sort - Trocas
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Trocas
0
1,000,000,000
2,000,000,000
3,000,000,000
4,000,000,000
5,000,000,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor

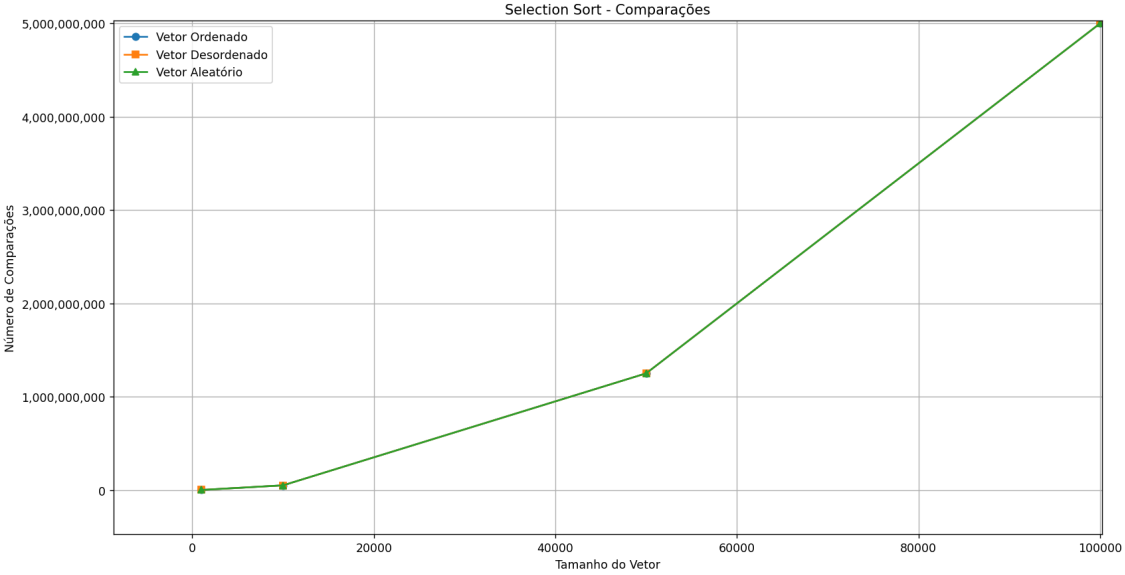
Selection Sort

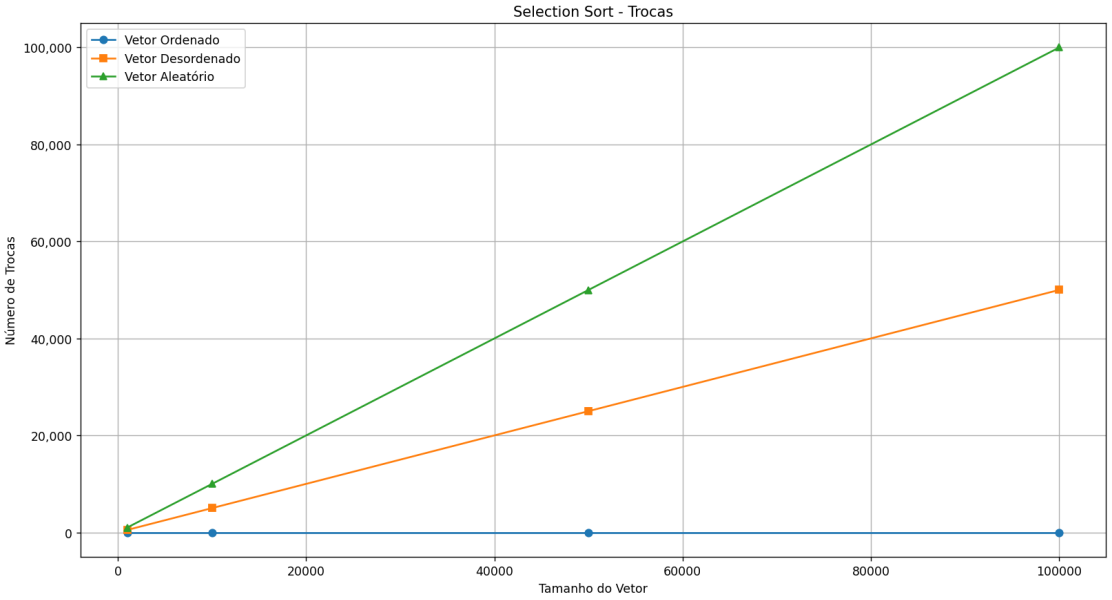
Tipo de Lista	Tamanho da Lista	Tempo de Execução (ms)	Comparações	Trocas
Ordenada	1.000	31.43	499500	0
	10.000	2800.98	49995000	0
	50.000	71703.06	1249975000	0
	100.000	306896.76	4999950000	0
Inv-Ordenada	1.000	32.00	499500	500
	10.000	3095.04	49995000	5000
	50.000	73555.49	1249975000	25000
	100.000	308650.97	4999950000	50000
Aleatória	1.000	31.00	499500	991
	10.000	3423.09	49995000	9984
	50.000	75736.24	1249975000	49993
	100.000	349523.77	4999950000	99991

Pelos resultados percebemos que:

O algoritmo assim como o bubble sempre realiza o máximo de comparações, nota-se que para todo o tipo de lista a execução ocorre em um tempo ligeramente diferente entre cada tipo sendo o melhor caso a lista ordenada e o pior caso a lista aleátoria. Apesar de ser mais eficiente que o bubble sort este algoritmo é ineficiente para listas muito grandes mesmo em seu melhor caso. Apesar disto percebe-se que este algoritmo realiza uma quantidade mínima de trocas, algo que pode ser vantajoso dependendo da sua aplicabilidade.


Selection Sort
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Tempo de Execução (milisegundos)
Tamanho do Vetor
0
50000
100000
150000
200000
250000
300000
350000
0
20000
40000
60000
80000
100000


Selection Sort - Comparações
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Comparações
Tamanho do Vetor
0
1,000,000,000
2,000,000,000
3,000,000,000
4,000,000,000
5,000,000,000
0
20000
40000
60000
80000
100000

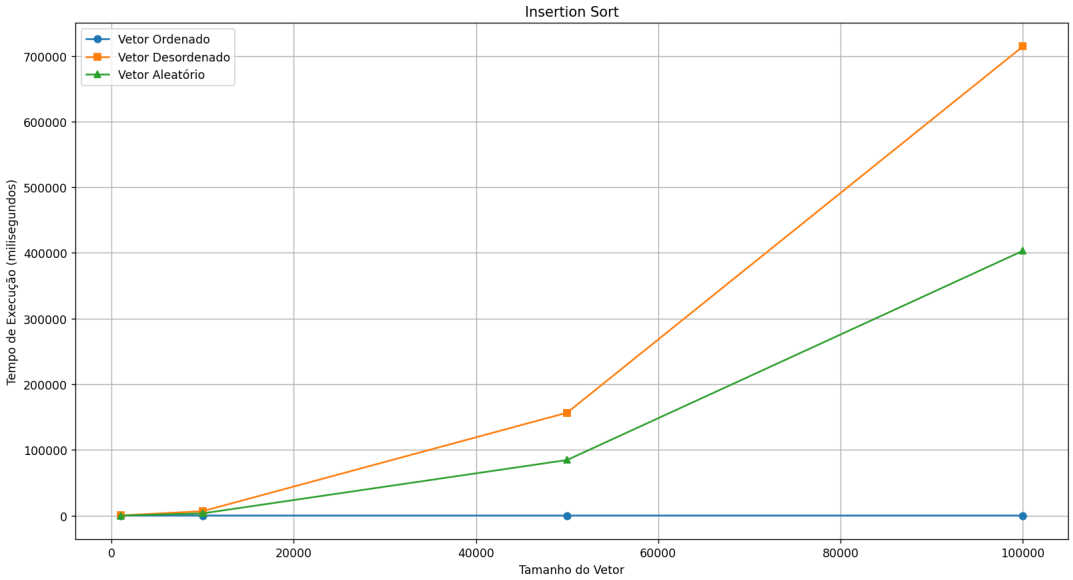

Selection Sort - Trocas
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Trocas
Tamanho do Vetor
0
20,000
40,000
60,000
80,000
100,000
0
20000
40000
60000
80000
100000

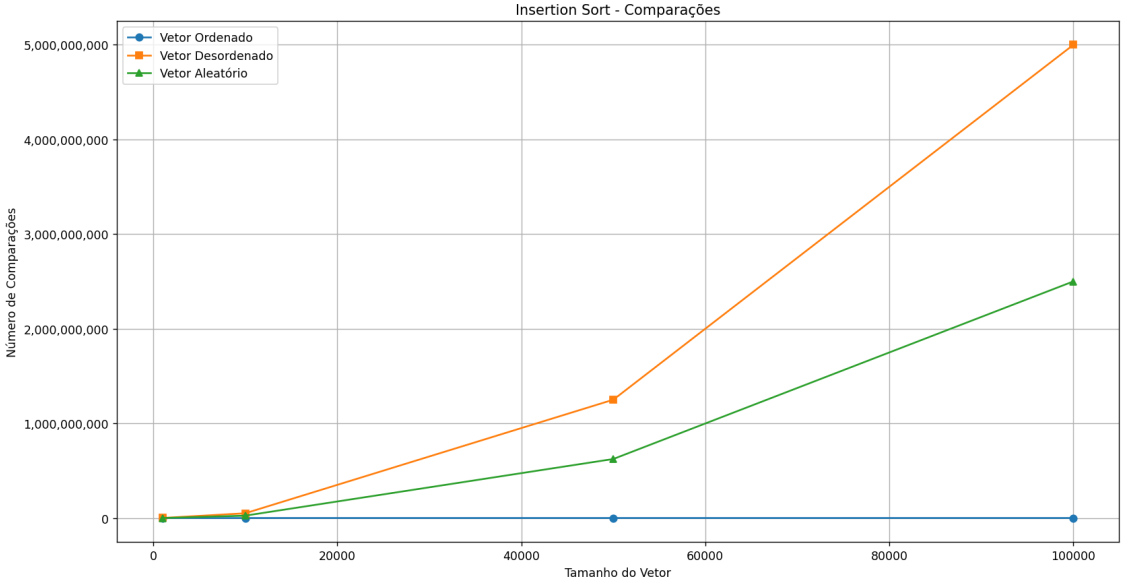
Insertion Sort

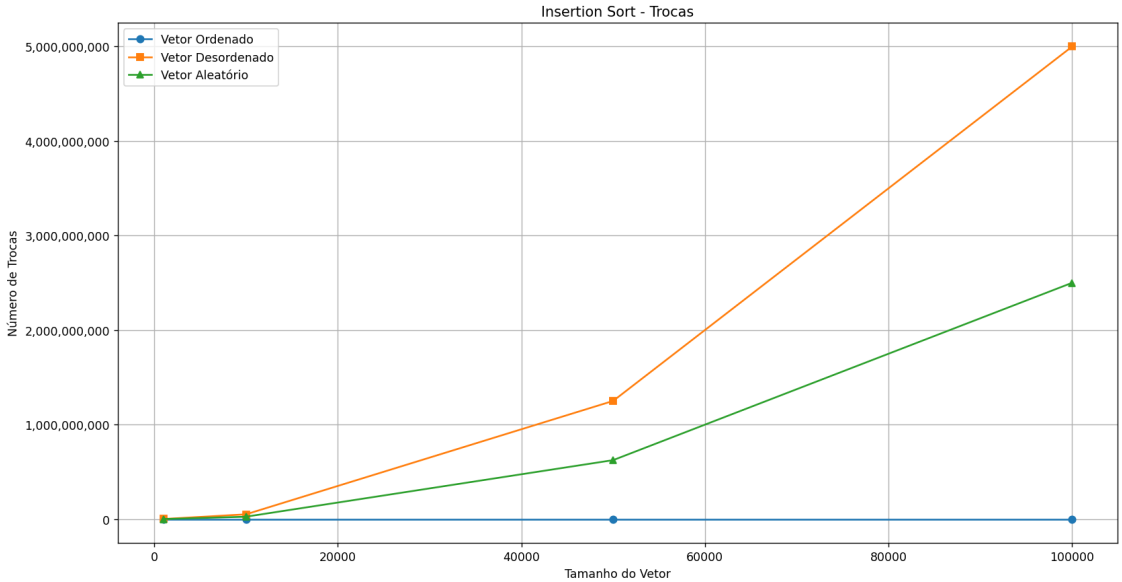
Tipo de Lista	Tamanho da Lista	Tempo de Execução (ms)	Comparações	Trocas
Ordenada	1.000	0.00	999	0
	10.000	1.00	9999	0
	50.000	8.51	49999	0
	100.000	14.68	99999	0
Inv-Ordenada	1.000	72.49	499500	500499
	10.000	6576.56	49995000	50004999
	50.000	156676.73	1249975000	1250024999
	100.000	714465.28	4999950000	5000049999
Aleatória	1.000	28.60	247906	247909
	10.000	3259.56	24869550	24869550
	50.000	84600.83	623463262	623463263
	100.000	403173.81	2500055839	2500055839

Pelos resultados percebemos que:

Este algoritmo se apresenta ser extremamente eficiente em listas já ordenadas, realizando o mínimo de comparações dado por n-1 e 0 trocas e executa incrivelmente rápido, sendo esse o melhor caso. Quando na lista inversamente ordenada ele realiza o máximo de comparações e o máximo de trocas, sendo este o pior caso. Na lista aleatório o algoritmo basicamente age como em um caso médio, onde não é tão lento como o inversamente ordenado, porém há um custo computacional mais elevado que o ordenado e apresenta uma crescente inconsistência para maiores conjuntos de dados.


Insertion Sort
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Tempo de Execução (milisegundos)
0
100000
200000
300000
400000
500000
600000
700000
0
20000
40000
60000
80000
100000
Tamanho do Vetor


Insertion Sort - Comparações
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Comparações
0
1,000,000,000
2,000,000,000
3,000,000,000
4,000,000,000
5,000,000,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor

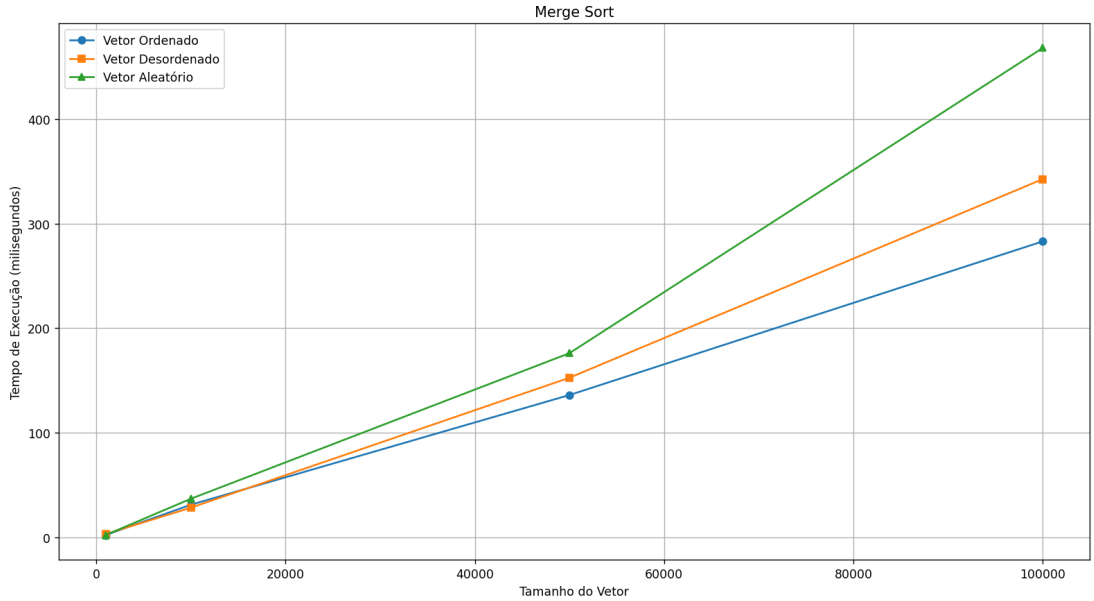

Insertion Sort - Trocas
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Trocas
0
1,000,000,000
2,000,000,000
3,000,000,000
4,000,000,000
5,000,000,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor

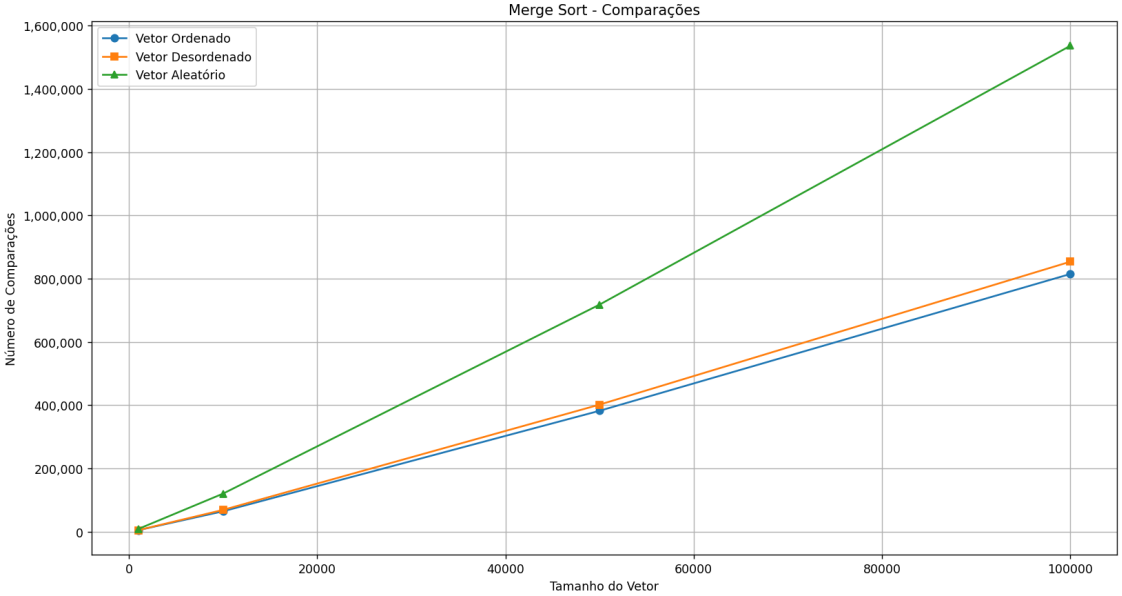
Merge Sort

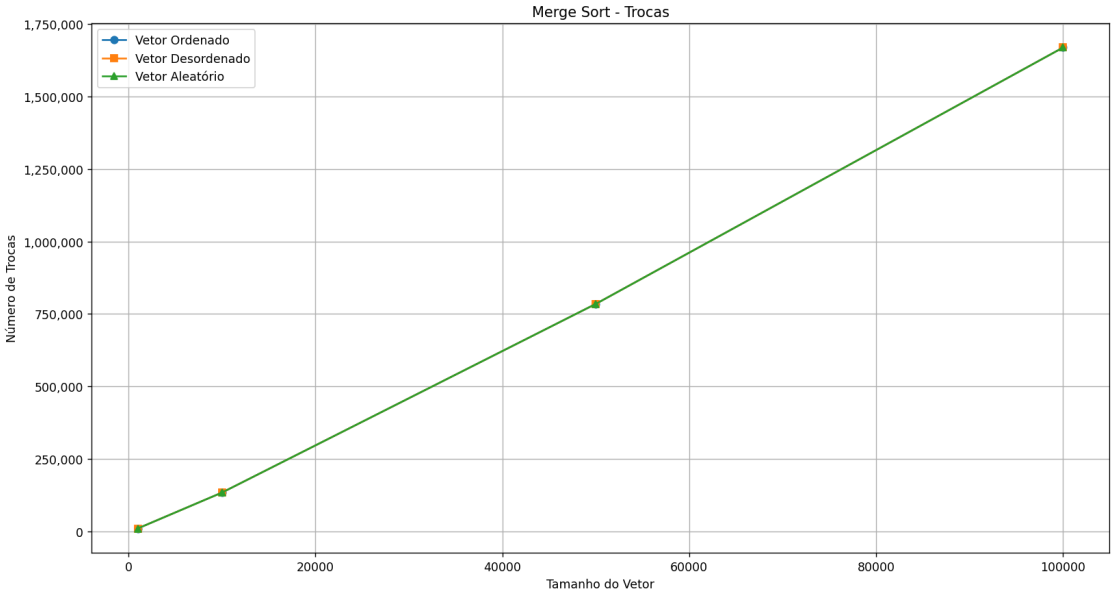
Tipo de Lista	Tamanho da Lista	Tempo de Execução (ms)	Comparações	Trocas
Ordenada	1.000	1.99	4932	9976
	10.000	31.00	64608	133616
	50.000	136.00	382512	784464
	100.000	283.03	815024	1668928
Inv-Ordenada	1.000	3.00	5044	9976
	10.000	28.16	69008	133616
	50.000	152.55	401952	784464
	100.000	342.63	853904	1668928
Aleatória	1.000	2.00	8739	9976
	10.000	36.71	120419	133616
	50.000	176.06	718281	784464
	100.000	468.16	1536679	1668928

Pelos resultados percebemos que:

O algoritmo consegue ser razoavelmente bom em listas ordenadas, inversamente ordenadas e aleatórias, por sempre dividir a lista em sublistas sempre hávera trocas e estas serão sempre constantes para o tamanho da lista, é possível notar um ligeiro crescimento do número de comparações entre as listas ordenadas e inversamente ordenadas, porém há uma grande diferença quando em listas aleatórias aumentando bastante o número de comparações e assim o tempo de execução, portanto temos como melhor caso as listas ordenadas e pior caso as listas aleatórias e o merge sort se apresenta como um algoritmo basntante razoável independente do formato inicial da lista.


Merge Sort
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Tempo de Execução (milisegundos)
0
100
200
300
400
0
20000
40000
60000
80000
100000
Tamanho do Vetor


Merge Sort - Comparações
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Comparações
0
200,000
400,000
600,000
800,000
1,000,000
1,200,000
1,400,000
1,600,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor

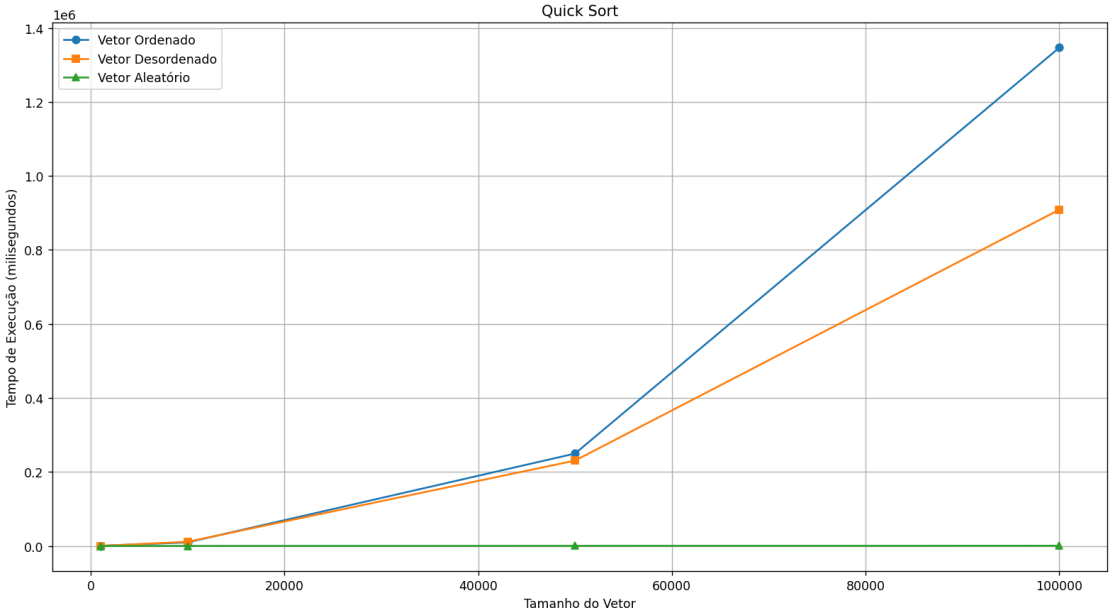

Merge Sort - Trocas
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Trocas
0
250,000
500,000
750,000
1,000,000
1,250,000
1,500,000
1,750,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor

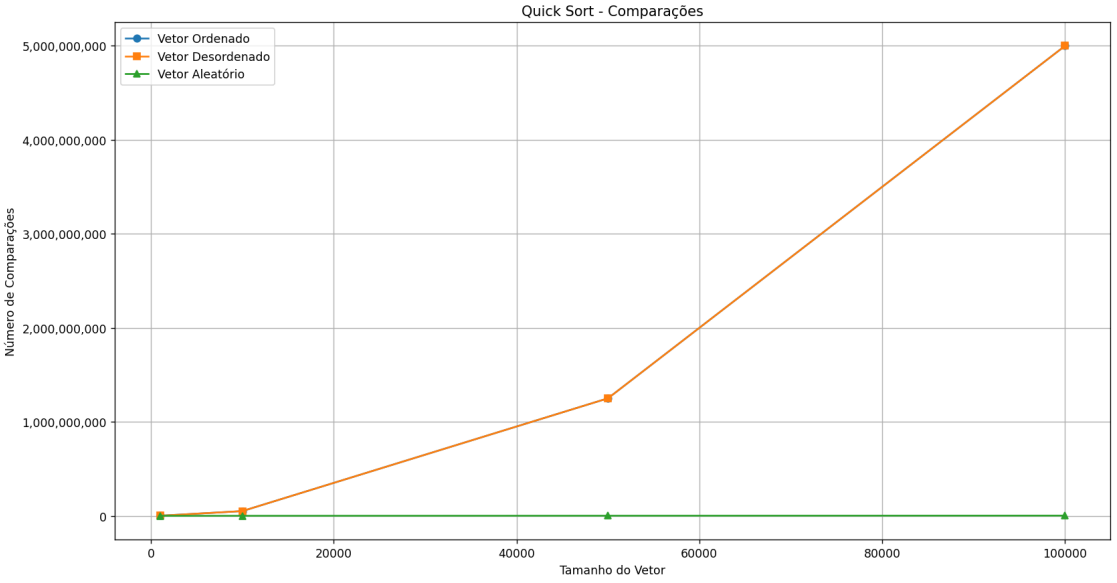
Quick Sort

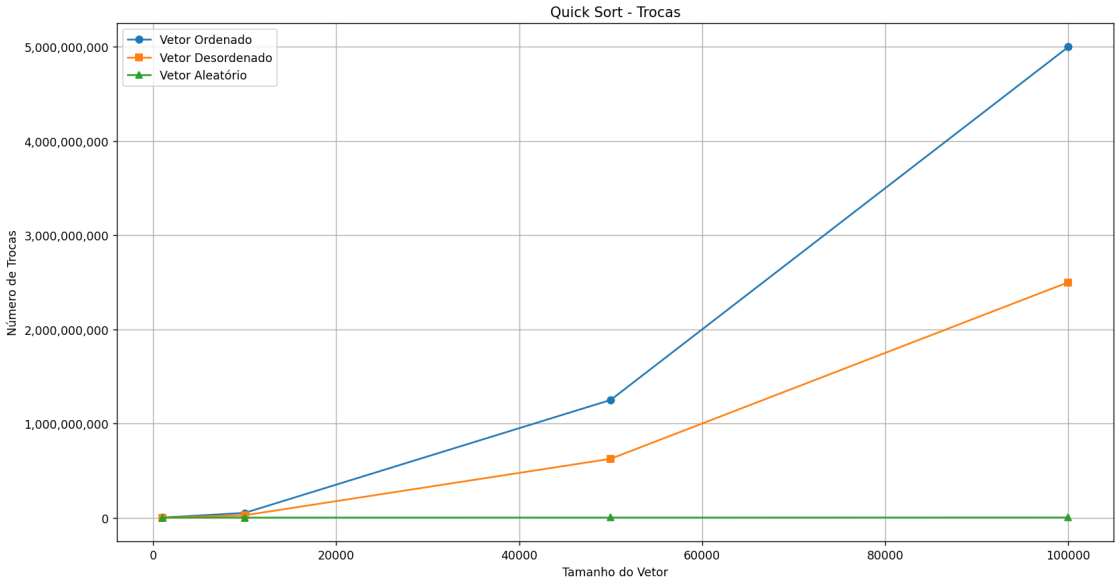
Tipo de Lista	Tamanho da Lista	Tempo de Execução (ms)	Comparações	Trocas
Ordenada	1.000	90.26	499500	500499
	10.000	9526.55	49995000	50004999
	50.000	249599.33	1249975000	1250024999
	100.000	1346738.26	4999950000	5000049999
Inv-Ordenada	1.000	81.08	499500	250499
	10.000	11160.00	49995000	25004999
	50.000	230571.66	1249975000	625024999
	100.000	908865.37	4999950000	2500049999
Aleatória	1.000	0.00	11565	5831
	10.000	37.08	162791	85894
	50.000	264.23	974993	519252
	100.000	528.51	2038026	1140211

Pelos resultados percebemos que:

Visto que o funcionamento do quick short se baseia em o uso de um pivô para balancear a lista, se a escolha do pivô não for optimizada (como no nosso caso) o algoritmo acaba desbalanceando uma lista já balanceada, isto ocorre na execução da lista ordenada e da lista inversamente ordenada percebe-se a alta quantidade de comparações e trocas para estes casos os quais realizam o máximo de comparações, os dois casos se assemelham muito como o pior caso porém nota-se que conforme a entrada cresce a lista inversamente ordenada executa mais rapidamente que a ordenada, portanto temos que a lista ordenada representa o pior caso para grandes entradas. No caso da lista aleatório o algoritmo executa incrivelmente rápido tendo em vista que a lista passa pro algoritmo desbalanceada e o algorito a balanceia, neste caso o algoritmo realiza um número razoável de comparações e trocas, logo este se apresenta como o melhor caso.


Quick Sort
1e6
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Tempo de Execução (milisegundos)
Tamanho do Vetor


Quick Sort - Comparações
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Comparações
Tamanho do Vetor

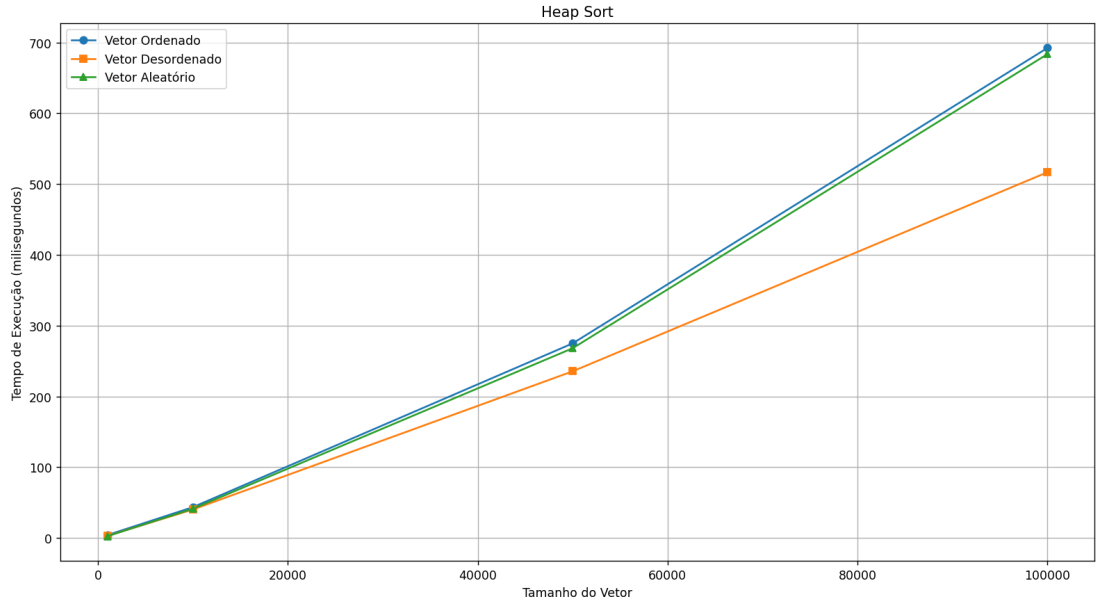

Quick Sort - Trocas
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Trocas
Tamanho do Vetor

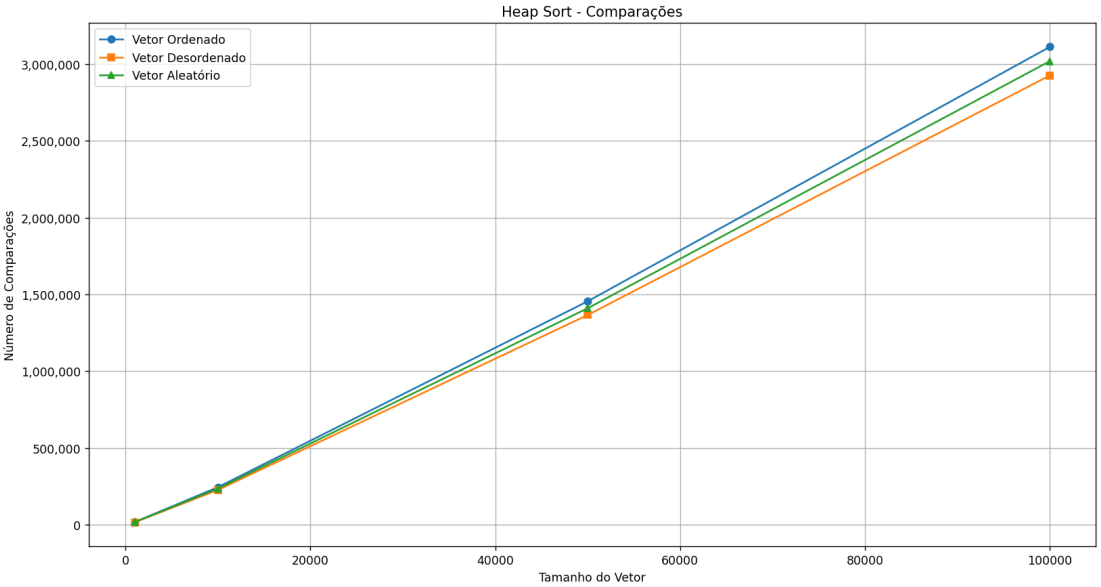
Heap Sort

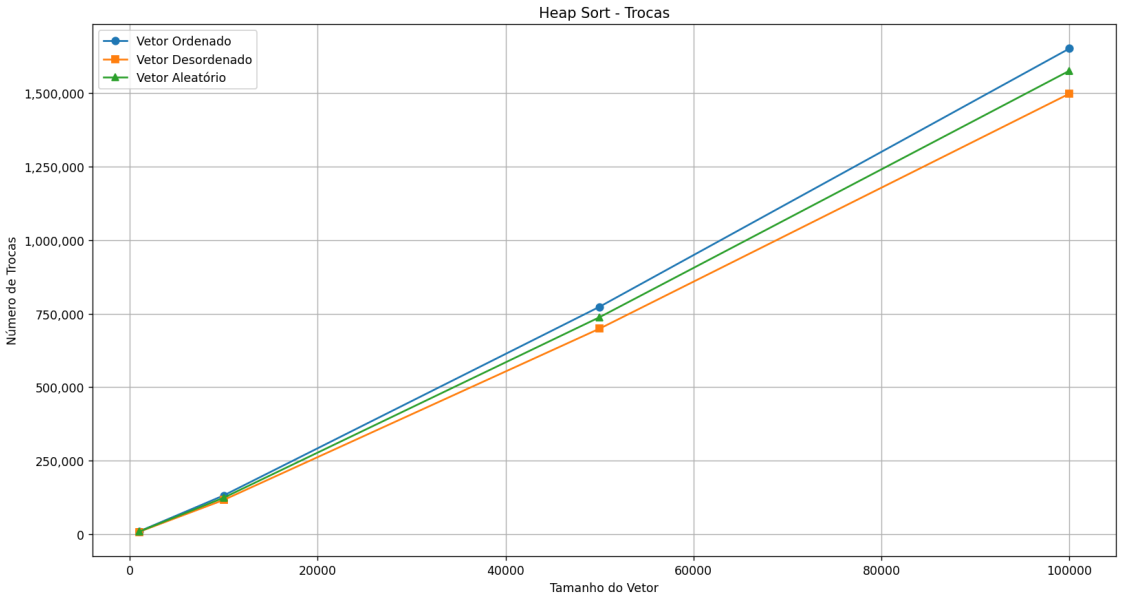
Tipo de Lista	Tamanho da Lista	Tempo de Execução (ms)	Comparações	Trocas
Ordenada	1.000	4.04	17583	9708
	10.000	43.45	244460	131956
	50.000	274.85	1455438	773304
	100.000	692.25	3112517	1650854
Inv-Ordenada	1.000	2.85	15965	8316
	10.000	40.13	226682	116696
	50.000	235.41	1366047	698892
	100.000	516.64	2926640	1497434
Aleatória	1.000	2.51	16857	9062
	10.000	41.13	235398	124205
	50.000	268.00	1410103	737727
	100.000	683.49	3019563	1575369

Pelos resultados percebemos que:

Independentemente se a lista está ordenada, inversamente ordenada ou aleatória, o algoritmo mantêm uma estabilidade em relação ao tempo de executação, ao número de comparações e ao número de trocas não diferenciando muito entre os casos. Portanto temos este como um algoritmo que roda razoavelmente bem qualquer caso, sendo o melhor caso o algoritmo inversamente ordenado e o pior caso o algoritmo ordenado.


Heap Sort
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Tempo de Execução (milisegundos)
Tamanho do Vetor
0
100
200
300
400
500
600
700
0
20000
40000
60000
80000
100000


Heap Sort - Comparações
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Comparações
Tamanho do Vetor
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
0
20000
40000
60000
80000
100000


Heap Sort - Trocas
Vetor Ordenado
Vetor Desordenado
Vetor Aleatório
Número de Trocas
Tamanho do Vetor
0
250,000
500,000
750,000
1,000,000
1,250,000
1,500,000
0
20000
40000
60000
80000
100000

5 Discussão

5.1 Expectativas

Do ponto de vista de estudantes de ciência da computação, podemos afirmar que muito do que nos é ensinado em sala de aula é limitado à teoria. Mais uma vez, a teoria da análise de complexidade de algoritmos foi passada pelo professor em sala, a fim de nos ajudar a compreender os conceitos de algoritmos eficientes e ineficientes. Contudo, esse trabalho foi proposto justamente com o propósito de aplicar algoritmos conhecidos para ordenação de itens e demonstrar a veracidade da teoria apresentada em sala.

As expectativas para esse trabalho eram boas, uma vez que não seria difícil implementar os algoritmos de ordenação (pois são conhecidos) e desenvolver um relatório completo acerca dos resultados ajuda-nos a verificar facilmente a utilidade da teoria de complexidade de algoritmos.

Como resultado, esperávamos que os algoritmos que seriam testados não demorariam tanto na execução, contudo fomos surpreendidos, pois o tempo de execução em alguns casos (em geral, nos piores casos) foi maior do que o esperado. De forma análoga, esperávamos que os algoritmos mais eficientes fossem mais rápidos, o que de fato aconteceu.

Sendo assim, desde o início da implementação foi possível notar a importância do projeto e análise de algoritmos baseados na complexidade, e a expectativa para verificar o funcionamento era alta.

5.2 Desafios e Dificuldades

Durante a implementação e execução de um algoritmo de ordenação, o Quick Sort, enfrentamos um problema significativo relacionado ao limite de recursão. Esse problema ocorreu ao tentar ordenar listas de 50.000 e 100.000 itens, em que o número de chamadas recursivas excedeu o limite permitido pela linguagem Python, resultando na interrupção da execução. Esse comportamento evidenciou uma limitação prática do uso de algoritmos recursivos em situações que envolvem grandes volumes de dados.

Embora, teoricamente, esses algoritmos tenham complexidade $O(nlogn)$ e sejam altamente eficientes em termos de tempo de execução, a profundidade das chamadas recursivas impõe uma restrição que não é diretamente refletida na análise de complexidade. Na prática, ao atingir grandes listas, o sistema enfrenta dificuldades com a pilha de chamadas, o que interrompe a execução normal do algoritmo. Esse problema é agravado pelo fato de que a recursão em Python tem um limite relativamente baixo comparado a linguagens que permitem um controle mais específico da alocação de memória e da profundidade da pilha.

Devido a esse problema, foi necessário utilizar outra abordagem para realizar a ordenação usando o Quick Sort. Optamos, após pesquisar sobre as possibilidades, por usar o algoritmo iterativo do Quick Sort. Depois que fizemos a mudança, o algoritmo funcionou perfeitamente, mantendo, inclusive, a eficiênca esperada, como consta nos resultados apresentados anteriormente.

5.3 Resultados Obtidos

A prática realizada no estudo dos algoritmos de ordenação revela uma clara correspondência com a teoria da complexidade de algoritmos apresentada em sala de aula. A análise prática, que incluiu a implementação e teste de diversos algoritmos, reforça os conceitos teóricos de eficiência e ineficiência com base nas entradas e nos algoritmos testados.

No caso dos algoritmos Bubble Sort, Selection Sort e Insertion Sort, todos com complexidade de tempo $O(n^2)$ no pior caso, os testes confirmaram as limitações teóricas desses métodos de

listas de tamanho grande. Como previsto, esses algoritmos tiveram um desempenho insatisfatório, especialmente em cenários de pior caso, como listas inversamente ordenadas. O tempo de execução aumentou consideravelmente conforme o número de elementos na lista cresceu, refletindo exatamente o que é esperado teoricamente para algoritmos de complexidade quadrática. Embora os três algoritmos compartilhem a mesma complexidade no pior caso, houve variações de desempenho. O Insertion Sort, por exemplo, demonstrou uma eficiência notável em listas que já estavam ordenadas ou quase ordenadas. Isso é consistente com a teoria, que prevê uma complexidade de $O(n)$ no melhor caso para o Insertion Sort, devido ao fato de que, nessas condições, o algoritmo realiza o mínimo de comparações e não efetua trocas. Em contraste, o Bubble Sort se destacou negativamente por seu desempenho consistentemente ruim, mesmo sendo frequentemente utilizado para fins didáticos devido à sua simplicidade. O Selection Sort, embora ligeiramente mais eficiente em termos de trocas, ainda apresentou tempos de execução semelhantes ao Bubble Sort, confirmando que sua complexidade de $O(n^2)$ o torna inadequado para grandes volumes de dados, mesmo com o número reduzido de trocas.

O Merge Sort e o Quick Sort, por sua vez, se destacaram em termos de eficiência, como esperado devido à sua complexidade de tempo $O(nlogn)$ tanto no melhor quanto no caso médio. O Merge Sort, sendo um algoritmo baseado na técnica de divisão e conquista, demonstrou uma performance estável em todos os cenários testados. Independentemente de a lista estar ordenada, inversamente ordenada ou aleatória, o Merge Sort apresentou tempos de execução consistentes, confirmando sua robustez teórica e sua aplicabilidade em situações práticas que envolvem grandes conjuntos de dados. A eficiência do Merge Sort reflete sua capacidade de lidar com grandes listas de maneira eficaz, sem sofrer grandes penalizações em termos de tempo de execução, mesmo quando comparado a listas desordenadas.

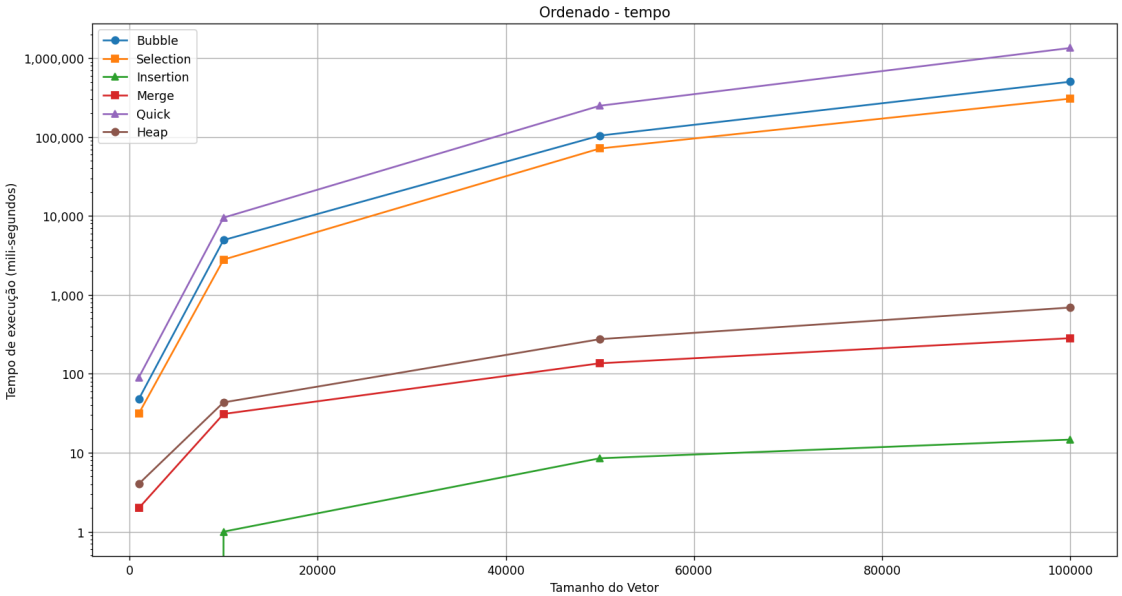
O Quick Sort, outro algoritmo que utiliza divisão e conquista, apresentou resultados mistos nos testes práticos. Em listas aleatórias, seu desempenho foi excelente, conforme esperado, pois o algoritmo consegue particionar a lista de forma relativamente equilibrada, o que mantém a complexidade $O(nlogn)$. No entanto, o pior caso teórico do Quick Sort, que ocorre quando a lista já está ordenada ou inversamente ordenada, foi confirmado nos experimentos. Nessas situações, o algoritmo apresentou um número elevado de comparações e trocas, resultando em uma complexidade de $O(n^2)$, como previsto pela teoria. A ineficiência do Quick Sort em listas ordenadas está diretamente relacionada à escolha inadequada do pivô, que resulta em partições muito desequilibradas. Esse comportamento destaca a necessidade de estratégias de escolha de pivô mais sofisticadas para otimizar o desempenho do Quick Sort em cenários reais.

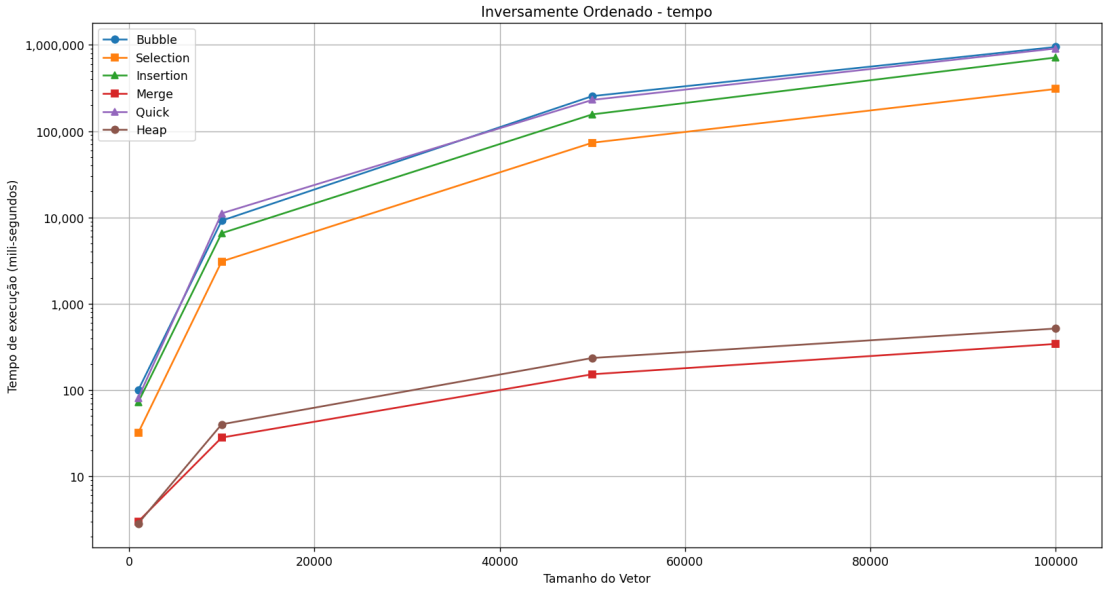
O Heap Sort, por sua vez, confirmou sua robustez em relação à teoria. Com uma complexidade de $O(nlogn)$ no pior, melhor e caso médio, o Heap Sort se mostrou uma opção eficiente em todos os tipos de listas testadas. Independentemente de a lista estar ordenada, inversamente ordenada ou aleatória, o algoritmo manteve um desempenho estável, demonstrando que sua abordagem, baseada na estrutura de heap binário, é eficaz para grandes volumes de dados. Além disso, o Heap Sort também possui a vantagem de ser um algoritmo in-place, o que significa que ele não requer espaço adicional significativo além da lista original, tornando-o uma escolha atrativa em cenários onde o uso de memória é uma preocupação.

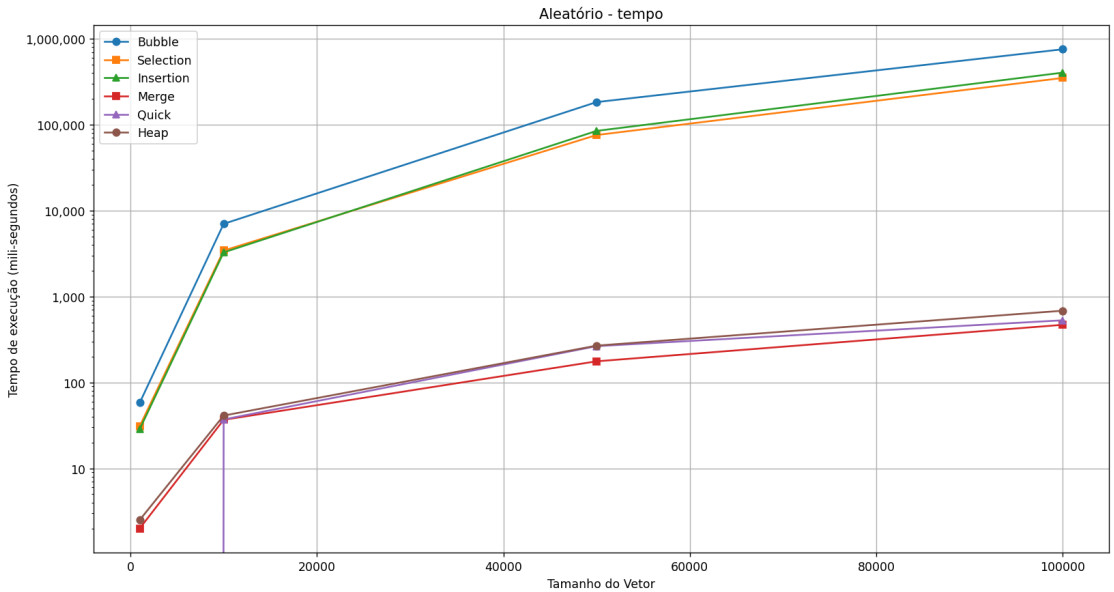
Ao comparar a teoria estudada em sala com os resultados obtidos na prática, fica claro que a teoria da complexidade de algoritmos é uma ferramenta extremamente útil para prever o desempenho de algoritmos em diferentes cenários. Os algoritmos de complexidade quadrática, como o Bubble Sort e o Selection Sort, se mostraram ineficazes para listas grandes, confirmando as limitações impostas por sua complexidade teórica. Por outro lado, algoritmos como o Merge Sort e o Quick Sort, com complexidade $O(nlogn)$, mostraram-se mais eficientes para lidar com grandes volumes de dados, com exceção do Quick Sort em seu pior caso.

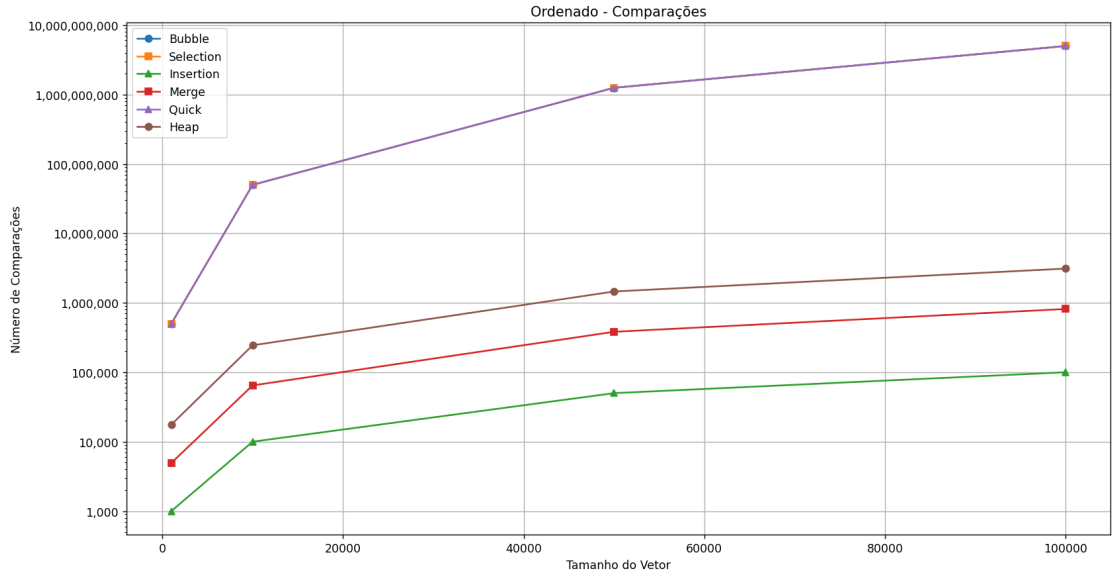
Esses resultados evidenciam a importância de escolher o algoritmo de ordenação correto com base nas características da entrada. A teoria de complexidade, ao prever a quantidade de operações necessárias para diferentes tipos de entradas, fornece uma base sólida para tomar decisões informadas sobre qual algoritmo utilizar. Na prática, isso significa que a escolha de um algoritmo de ordenação deve levar em consideração não apenas a eficiência teórica, mas também a natureza dos dados que serão processados, a fim de garantir o melhor desempenho possível. O Merge Sort, por exemplo, se destaca como uma excelente escolha para grandes listas, enquanto o Insertion Sort pode ser preferido em casos onde as listas já estão ordenadas ou quase ordenadas.

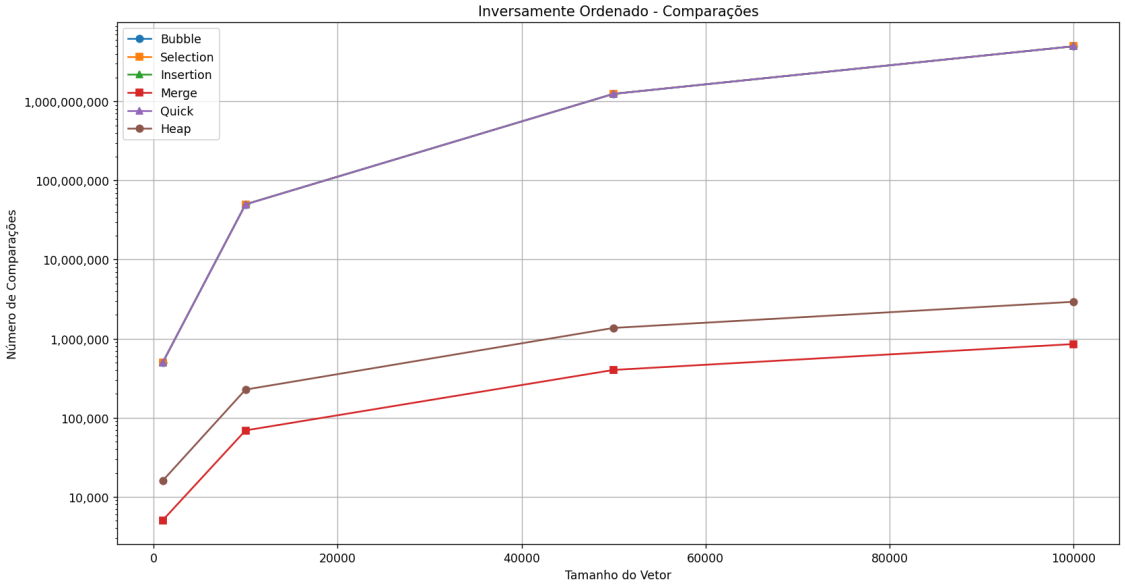
Em resumo, a experiência prática validou amplamente os conceitos teóricos de complexidade de algoritmos, destacando tanto as limitações quanto as vantagens de cada método de ordenação. Esse exercício de implementação e análise empírica demonstrou que a teoria de complexidade não apenas explica o comportamento dos algoritmos, mas também fornece insights cruciais para seu uso eficiente em aplicações reais.

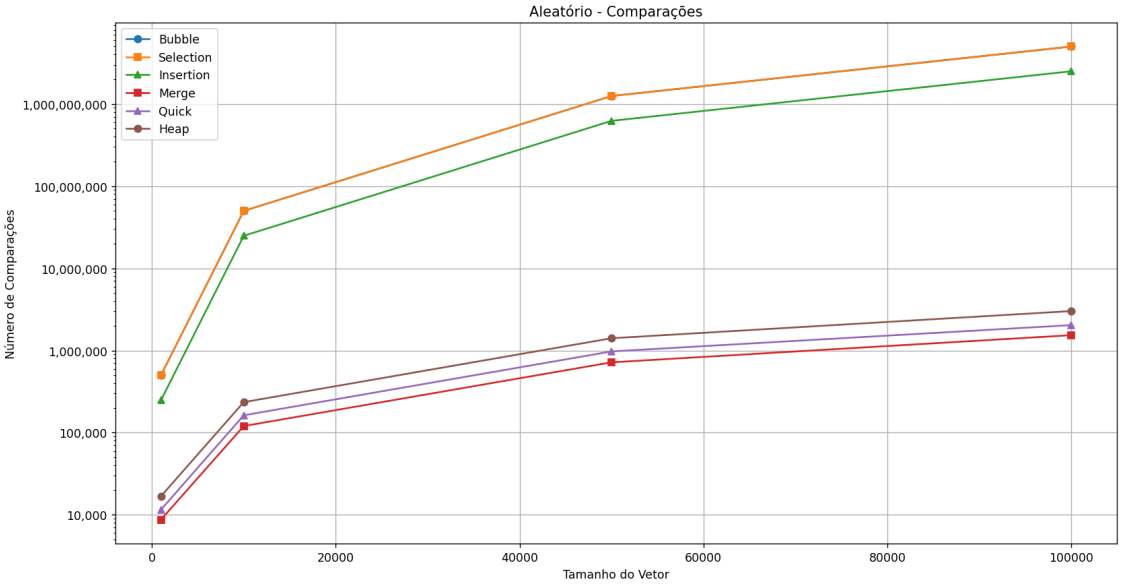

Ordenado - tempo
Bubble
Selection
Insertion
Merge
Quick
Heap
Tempo de execução (mili-segundos)
1,000,000
100,000
10,000
1,000
100
10
1
0
20000
40000
60000
80000
100000
Tamanho do Vetor

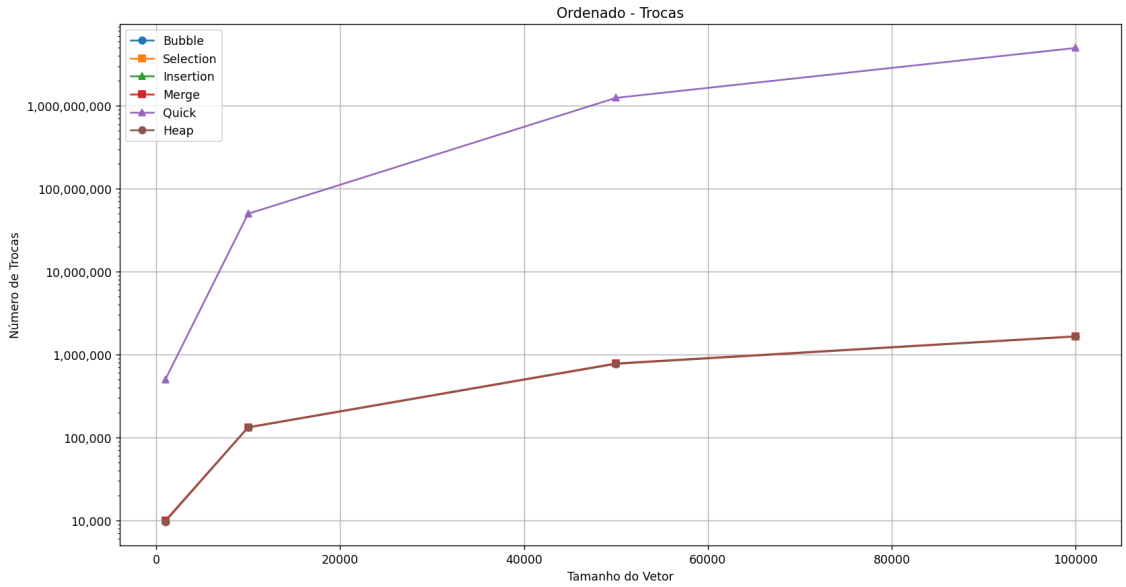

Inversamente Ordenado - tempo
Bubble
Selection
Insertion
Merge
Quick
Heap
Tempo de execução (mili-segundos)
1,000,000
100,000
10,000
1,000
100
10
0
20000
40000
60000
80000
100000
Tamanho do Vetor

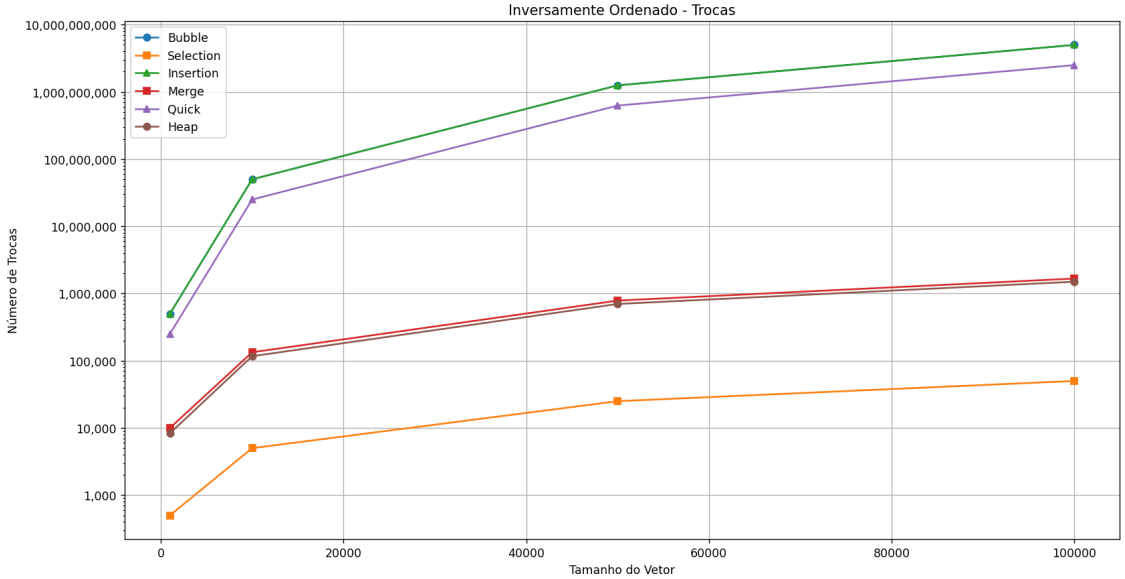

Aleatório - tempo
Bubble
Selection
Insertion
Merge
Quick
Heap
Tempo de execução (mili-segundos)
1,000,000
100,000
10,000
1,000
100
10
0
20000
40000
60000
80000
100000
Tamanho do Vetor

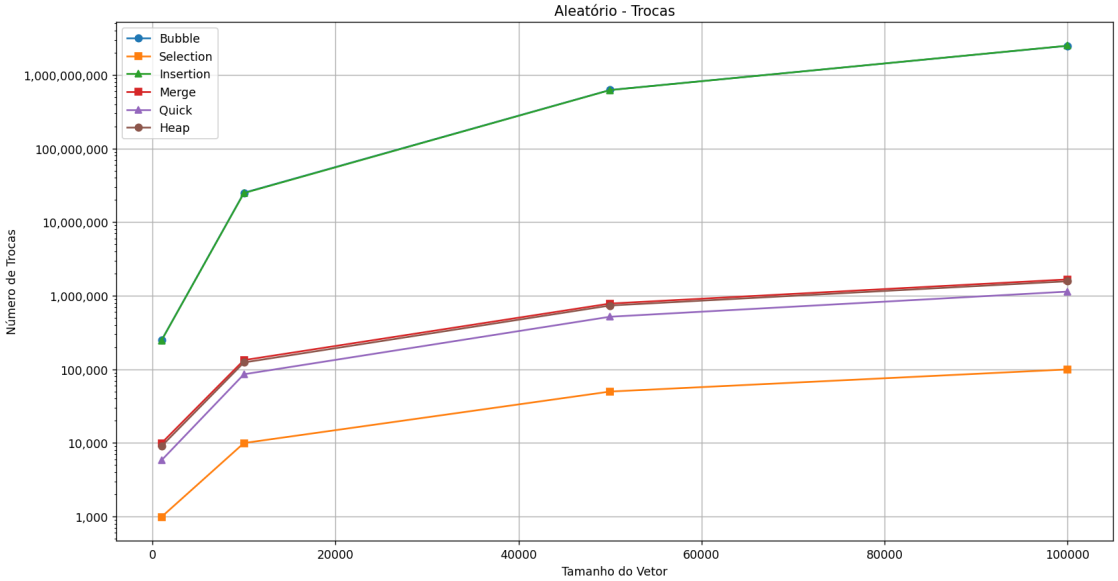

Ordenado - Comparações
Bubble
Selection
Insertion
Merge
Quick
Heap
Número de Comparações
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor


Inversamente Ordenado - Comparações
Bubble
Selection
Insertion
Merge
Quick
Heap
Número de Comparações
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor


Aleatório - Comparações
Bubble
Selection
Insertion
Merge
Quick
Heap
Número de Comparações
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor


Ordenado - Trocas
Bubble
Selection
Insertion
Merge
Quick
Heap
Número de Trocas
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor


Inversamente Ordenado - Trocas
Bubble
Selection
Insertion
Merge
Quick
Heap
Número de Trocas
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor


Aleatório - Trocas
Bubble
Selection
Insertion
Merge
Quick
Heap
Número de Trocas
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
0
20000
40000
60000
80000
100000
Tamanho do Vetor

6 Conclusão

De acordo com os dados obtidos:

- Bubble Sort: Apesar da fácil implementação é muito ineficiente para ser usado. Pois em todos os 3 tipos de listas ele teve um dos piores desempenhos em relação ao tempo de execução
- Selection: Apesar de ser melhor que o Bubble Sort este algoritmo apresentou-se ser ineficiente para entradas medianas a grandes de dado, porém este tem poucas trocas, fator que pode ser útil dependendo da sua aplicabilidade.
- Insertion: Este algoritmo é extremamente eficiente para usar em listas ordenadas ou praticamente ordenadas realizando o mínimo de comparações e poucas trocas.
- Merge: Por se utilizar do método dividir para conquistar, este é um ótimo algoritmo para listas grandes, porém para listas pequenas este algoritmo pode ter uma execução mais lenta em relação a outros.
- Quick: Como no merge a lista é dividida em sublistas para a ordenação, por tentar balancear o vetor a partir de um pivô das extremidades este algoritmo é extremamente ineficaz para listas ordenadas e desordenadas listas as quais já estão balanceadas, mostrando seu verdadeiro valor em listas aleatórias.
- Heap: Este apresentou uma boa execução em todos os casos, não foi foi apresentado este teste no relatório, porém o heap sort apresenta um ótimo funcionamento para listas com repetição de elementos, visto que nesses casos há menos trocas em comparação com outros algoritmos.

De forma geral:

O melhor algoritmo em relação ao tempo de execução é o Merge Sort, porém nota-se que o Heap Sort apresenta também um ótimo funcionamento de modo geral. O pior é o Bubble Sort o qual apresenta um bom funcionamento apenas para fins de explicação de algoritmos de ordenação.

O melhor algoritmo em relação ao número de comparações é o Merge Sort e novamente temos o Heap Sort apresentando um número aproximado de comparações. Os piores são assumidos por Bubble e Selection que sempre realizam o máximo de comparações.

O melhor algoritmo em relação ao número de trocas é o Selection Sort. Os piores são assumidos por Bubble e Selection, porém caso o vetor esteja ordenado o Quick é tido como o pior.

De forma particular:

Tipo de Lista	Tamanho da Lista	Tempo de Execução (ms)	Comparações	Trocas	Algoritmo
Ordenada	1.000	0.00	999	0	Insertion Sort
	10.000	1.00	9999	0	Insertion Sort
	50.000	8.51	49999	0	Insertion Sort
	100.000	14.68	99999	0	Insertion Sort
Inv-Ordenada	1.000	2.85	15965	8316	Heap Sort
	10.000	40.13	226682	116696	Heap Sort
	50.000	235.41	1366047	698892	Heap Sort
	100.000	516.64	2926640	1497434	Heap Sort
Aleatória	1.000	0.00	11565	5831	Quick Sort
	10.000	37.08	162791	85894	Quick Sort
	50.000	264.23	974993	519252	Quick Sort
	100.000	528.51	2038026	1140211	Quick Sort

Tabela 1: Melhores algoritmos em termos de tempo de execução para diferentes tipos e tamanhos de listas

7 Referências

Repositório com Algoritmos

- **LUISFELIPEKRAUSE. GitHub - LuisFelipeKrause/AlgoritmosOrdenacaoPAA: Repositório utilizado no desenvolvimento de trabalho prático da disciplina de Projeto e Análise de Algoritmos.** Disponível em: https://github.com/LuisFelipeKrause/Algoritmos_Ordenacao_PAA. Acesso em: 10 out. 2024.

Listagem das Fontes Utilizadas

- **Algoritmos de ordenação explicados com exemplos em Python, Java e C++.** Disponível em: https://www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-e
- **GABI DEUTNER. Dominando Algoritmos de Ordenação: Exemplos Práticos em Python para Desenvolvedores.** Disponível em: https://medium.com/@deutnerg/dominando-algor C3%A7%C3%A3o-exemplos-pr%C3%A1ticos-em-python-para-desenvolvedores-c69 Acesso em: 9 out. 2024
- **Conheça os principais algoritmos de ordenação.** Disponível em: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/.
- **FREITAS, R. Desempenho e Complexidade: Um Estudo Comparativo dos Algoritmos de Ordenação.** Disponível em: https://medium.com/@ricardo.macedo/desempenho-e-complexidade-um-estudo-comparativo-dos-algoritmos-de-orde C3%A7%C3%A3o-b488bcaba3ff/. Acesso em: 9 out. 2024.
- **Algoritmos de Ordenação: Análise e Comparação.** Disponível em: https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261
- **PACIFICO, A. Quicksort vs Mergesort vs Heapsort: Análise de desempenho - ProgramAi! - Dicas, cursos e tutoriais de programação.** Disponível em: https://programai.com.br/quicksort-vs-mergesort-vs-heapsort/ Acesso em: 06 out. 2024.