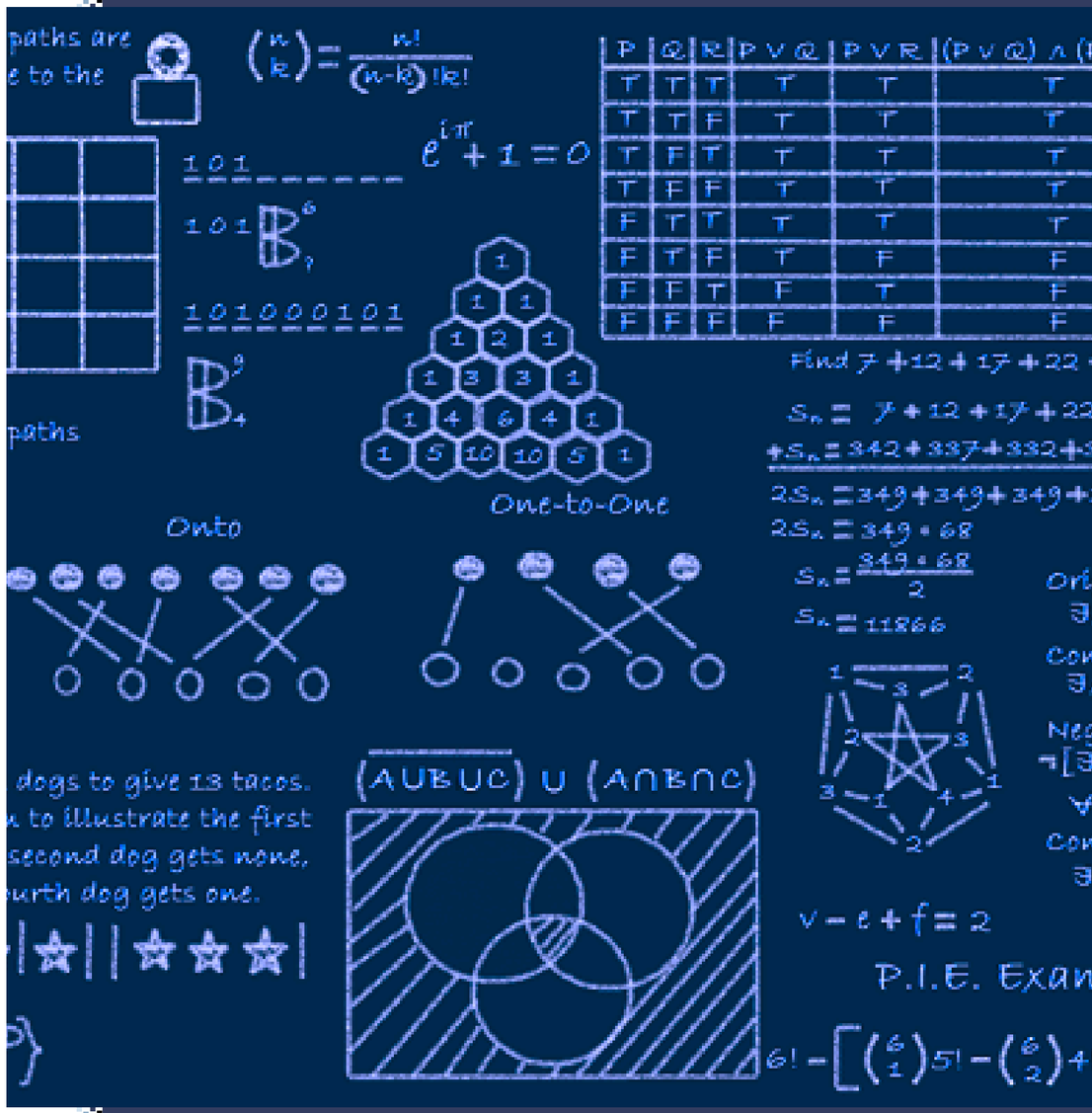
PROJETO E ANÁLISE DE ALGORITMOS

Análise Empírica de Algoritmos de Ordenação

ARMANO BARROS E SOPHIA MENEZES



Tópicos

- Introdução
- Revisão Teórica
- Metodologia
  - Ambiente de Teste
  - Medição do tempo de execução
- Implementação
- Comparação de Resultados

Introdução

A ordenação de dados é uma operação fundamental na computação, essencial para a organização eficiente e rápida recuperação de informações.

Este trabalho acadêmico visa realizar uma análise empírica de diferentes algoritmos de ordenação, explorando aspectos teóricos e práticos.
Serão implementados e comparados seis algoritmos (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort) utilizando listas de tamanhos variados e diferentes distribuições.

A análise envolverá a medição do tempo de execução, número de comparações e número de trocas de cada algoritmo, proporcionando uma comparação detalhada do desempenho.

Revisão Teórica

Bubble Sort

Funcionamento:

Percorre a lista, comparando e trocando elementos adjacentes. Repetido até que a lista esteja ordenada.

Complexidade:

- Melhor caso: (O(n))
- Pior caso: (O(n²))

O Bubble Sort é um algoritmo in-place, pois realiza a ordenação sem a necessidade de memória adicional significativa além da memória para a lista original.

É um algoritmo estável, pois não troca elementos iguais de lugar, mantendo a ordem relativa dos elementos com valores iguais.

Revisão Teórica

Selection Sort

Funcionamento:
Encontra o menor elemento da parte não ordenada e troca com o primeiro não ordenado.

Complexidade:
Melhor, médio e pior caso: ($O(n^2)$)

O Selection Sort é um algoritmo in-place, pois realiza a ordenação sem a necessidade de memória adicional significativa além da memória para a lista original.

O Selection Sort é instável, pois pode trocar elementos iguais de lugar.

Revisão Teórica

Insertion Sort

Funcionamento:

Constrói a lista ordenada um elemento de cada vez, retirando um elemento da lista de entrada e inserindo-o na posição correta na lista já ordenada.

Complexidade:

- Melhor caso: (O(n))
- Médio e pior caso: ($O(n^2)$)

O Insertion Sort é um algoritmo in-place, pois realiza a ordenação sem a necessidade de memória adicional significativa além da memória para a lista original.

O Insertion Sort é estável, pois não troca elementos iguais de lugar.

Revisão Teórica

Merge Sort

Funcionamento:
Divide a lista em duas, ordena cada metade e mescla.

Complexidade:
Melhor, médio e pior caso: (O(n log n))

O Merge Sort não é in-place, pois requer memória adicional para as listas temporárias utilizadas durante o processo de mesclagem.

O Merge Sort é estável, pois preserva a ordem relativa dos elementos iguais durante o processo de mesclagem.

Revisão Teórica

Quick Sort

Funcionamento:
Seleciona um pivô, particiona a lista e ordena recursivamente.

Complexidade:
- Melhor e médio caso: (O(n log n)), quando os pivôs são escolhidos de forma a balancear as partições.
- Pior caso: ($O(n^2)$), quando os pivôs escolhidos resultam em partições altamente desbalanceadas.

O Quick Sort é um algoritmo in-place, pois realiza a ordenação utilizando a memória da lista original, com uso adicional de memória para a pilha de recursão.

O Quick Sort é instável, pois pode trocar elementos iguais de lugar.

Revisão Teórica

Heap Sort

Funcionamento:
Baseado em uma estrutura de dados chamada heap binário, converte a lista em um heap máximo e remove a raiz repetidamente.

Complexidade:
Melhor, médio e pior caso: (O(n log n))

O Heap Sort é um algoritmo in-place, pois realiza a ordenação utilizando a memória da lista original, sem a necessidade de memória adicional significativa além da lista.

O Heap Sort é instável, pois pode trocar elementos iguais de lugar durante o processo de construção do heap e remoção da raiz.

Metodologia

Ambiente de Testes

Google Colaboratory (Colab)

- Serviço de nuvem gratuito do Google.
- Ambiente de notebook interativo para código, texto e gráficos.
- Problemas: Lentidão com listas pequenas, inviável para listas grandes (50.000 ou 100.000 elementos).

Teste Final: Computador Local

- Configurações:
  - Processador: Ryzen 7 5800x
  - Placa de Vídeo: GTX 1660 Super 6GB
  - Placa Mãe: B550 Elite
  - Memória RAM: DDR4 Vegeance RGB 3600 2x8GB

Metodologia

Medição do Tempo

Para medir o tempo de execução dos algoritmos, foi implementado uma função que receberia como parâmetro a função de ordenação e a executaria internamente, assim, utilizando da biblioteca time do Python, foi capturado o valor inicial de tempo ao início do algoritmo e no fim, e por fim, retornando a diferença entre o fim e o início, além das comparações e trocas realizadas no algoritmo.

Implementação

Bubble Sort

```
def bubble_sort(lista):
    trocas = 0
    comparacoes = 0
    n = len(lista)
    for i in range(n):
        trocado = False
        for j in range(0, n-i-1):
            comparacoes += 1
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
                trocado = True
                trocas += 1
        if not trocado:
            break
    print("Bubble:")
    return comparacoes, trocas
```

- Percorrer a lista do início ao fim repetidamente.
- Em cada passagem, comparar cada par de elementos adjacentes.
- Se os elementos estiverem na ordem errada (primeiro maior que o segundo), trocá-los.
- Continuar repetindo as passagens até que nenhuma troca seja necessária, indicando que a lista está ordenada.

Implementação

Selection Sort

```python
def selection_sort(lista):
    comparacoes = 0
    trocas = 0
    n = len(lista)
    for i in range(n):
        indice_menor = i
        for j in range(i+1, n):
            comparacoes += 1
            if lista[j] < lista[indice_menor]:
                indice_menor = j
        if i != indice_menor:
            lista[i], lista[indice_menor] = lista[indice_menor], lista[i]
            trocas += 1
    return comparacoes, trocas
```

- Dividir a lista em duas partes: a sub lista ordenada (inicialmente vazia) e a sub lista não ordenada.
- Encontrar o menor elemento na sub lista não ordenada.
- Trocar esse menor elemento com o primeiro elemento da sub lista não ordenada.
- Mover o limite entre a sub lista ordenada e não ordenada uma posição à direita.
- Repetir os passos até que toda a lista esteja ordenada.

Implementação

Insertion Sort

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

- Iniciar com a segunda posição da lista.
- Comparar o elemento atual com os elementos anteriores.
- Inserir o elemento atual na posição correta entre os elementos anteriores.
- Repetir o processo para cada elemento da lista até que todos os elementos estejam ordenados.

```python
def merge_sort(lista):
    comparacoes = 0
    trocas = 0
    if len(lista) > 1:
        meio = len(lista) // 2
        metade_esquerda = lista[:meio]
        metade_direita = lista[meio:]

        comparacoes_esquerda, trocas_esquerda = merge_sort(metade_esquerda)
        comparacoes_direita, trocas_direita = merge_sort(metade_direita)
        comparacoes += comparacoes_esquerda + comparacoes_direita
        trocas += trocas_esquerda + trocas_direita

        i = j = k = 0

        while i < len(metade_esquerda) and j < len(metade_direita):
            comparacoes += 1
            if metade_esquerda[i] < metade_direita[j]:
                lista[k] = metade_esquerda[i]
                i += 1
            else:
                lista[k] = metade_direita[j]
                j += 1
                trocas += 1  # Conta a troca de posição
            k += 1

        while i < len(metade_esquerda):
            lista[k] = metade_esquerda[i]
            i += 1
            k += 1

        while j < len(metade_direita):
            lista[k] = metade_direita[j]
            j += 1
            k += 1

    return comparacoes, trocas
```

Implementação

Merge Sort

- Dividir a lista em duas metades iguais (ou quase iguais).
- Aplicar recursivamente Merge Sort em cada metade.
- Mesclar as duas metades ordenadas para formar uma lista única ordenada.
- Continuar dividindo e mesclando até que toda a lista esteja ordenada.

```python
def quick_sort(lista):
    def particionar(lista, inicio, fim):
        pivo = lista[fim]
        i = inicio - 1
        comparacoes = 0
        trocas = 0
        for j in range(inicio, fim):
            comparacoes += 1
            if lista[j] <= pivo:
                i += 1
                lista[i], lista[j] = lista[j], lista[i]
                trocas += 1
        lista[i + 1], lista[fim] = lista[fim], lista[i + 1]
        trocas += 1
        return i + 1, comparacoes, trocas

    def quick_sort_iterativo(lista):
        stack = [(0, len(lista) - 1)]
        total_comparacoes = 0
        total_trocas = 0

        while stack:
            inicio, fim = stack.pop()
            while inicio < fim:
                pivo_index, comparacoes, trocas = particionar(lista, inicio, fim)
                total_comparacoes += comparacoes
                total_trocas += trocas

                # Determine the smaller subarray
                if pivo_index - inicio < fim - pivo_index:
                    stack.append((pivo_index + 1, fim))
                    fim = pivo_index - 1
                else:
                    stack.append((inicio, pivo_index - 1))
                    inicio = pivo_index + 1

        return total_comparacoes, total_trocas

    return quick_sort_iterativo(lista)
```

Implementação

Quick Sort

- Escolher um pivô (usualmente o último elemento da lista).
- Particionar a lista em duas sub listas: elementos menores que o pivô e elementos maiores que o pivô.
- Colocar o pivô na posição correta.
- Aplicar recursivamente Quick Sort nas duas sub listas.
- Continuar particionando e ordenando até que toda a lista esteja ordenada.

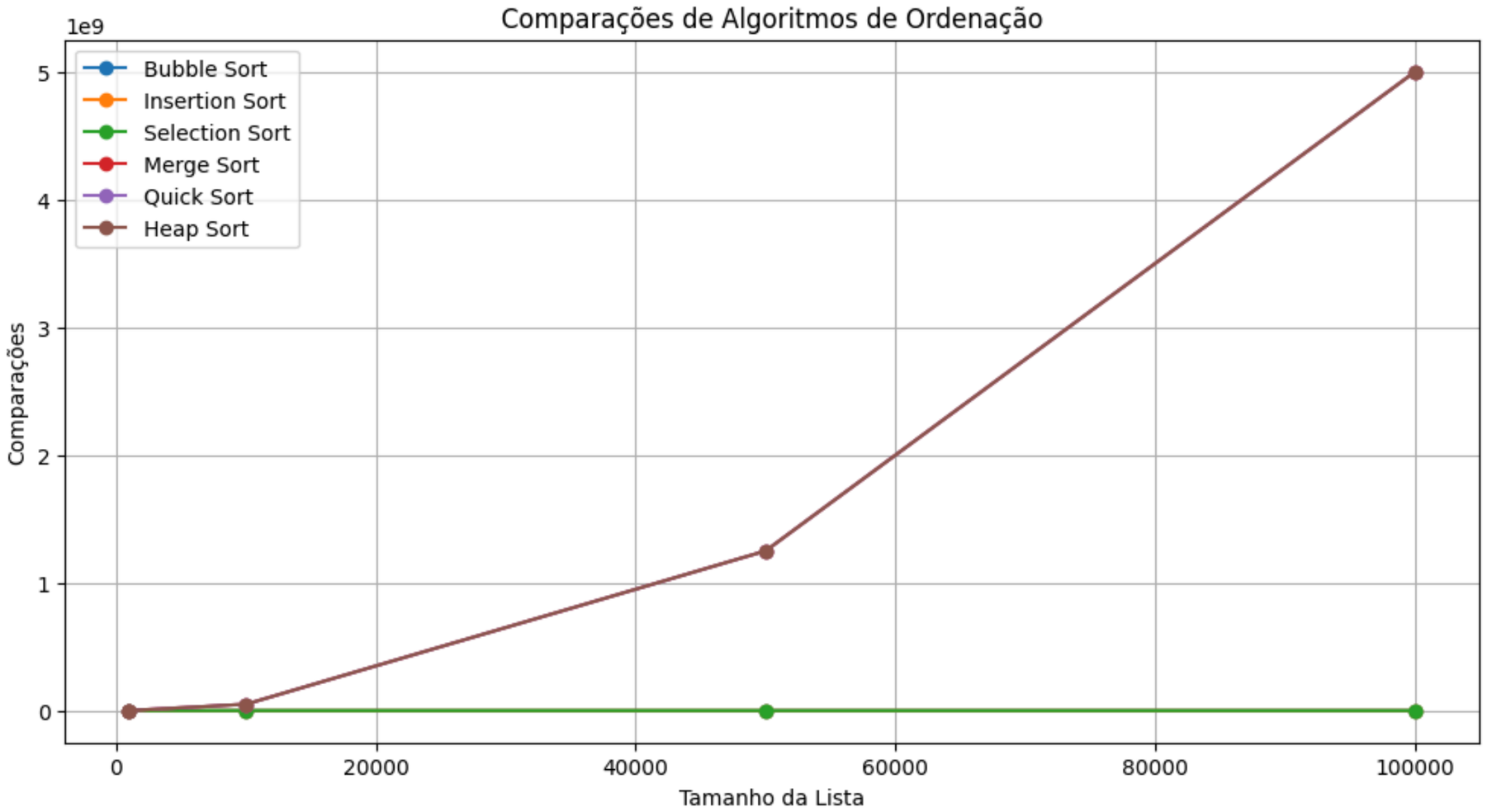
```python
def heapify(lista, n, i):
    comparacoes = 0
    trocas = 0
    maior = i
    esquerda = 2 * i + 1
    direita = 2 * i + 2

    if esquerda < n:
        comparacoes += 1
        if lista[esquerda] > lista[maior]:
            maior = esquerda

    if direita < n:
        comparacoes += 1
        if lista[direita] > lista[maior]:
            maior = direita

    if maior != i:
        lista[i], lista[maior] = lista[maior], lista[i]
        trocas += 1
        trocas += heapify(lista, n, maior)[1]
        comparacoes += heapify(lista, n, maior)[0]

    return comparacoes, trocas

def heap_sort(lista):
    trocas = 0
    comparacoes = 0
    n = len(lista)

    for i in range(n // 2 - 1, -1, -1):
        comparacoes += heapify(lista, n, i)[0]
        trocas += heapify(lista, n, i)[1]

    for i in range(n-1, 0, -1):
        lista[i], lista[0] = lista[0], lista[i]
        trocas += 1
        trocas += heapify(lista, i, 0)[1]
        comparacoes += heapify(lista, i, 0)[0]

    return comparacoes, trocas
```

Implementação

Heap Sort

- Construir um heap máximo a partir da lista.
- Trocar Repetidamente o maior elemento (na raiz do heap) com o último elemento do heap.
- Reduzir o tamanho do heap e aplicar heapify na raiz para restaurar a propriedade do heap.
- Continuar trocando e heapificando até que o tamanho do heap seja 1, resultando na lista ordenada.

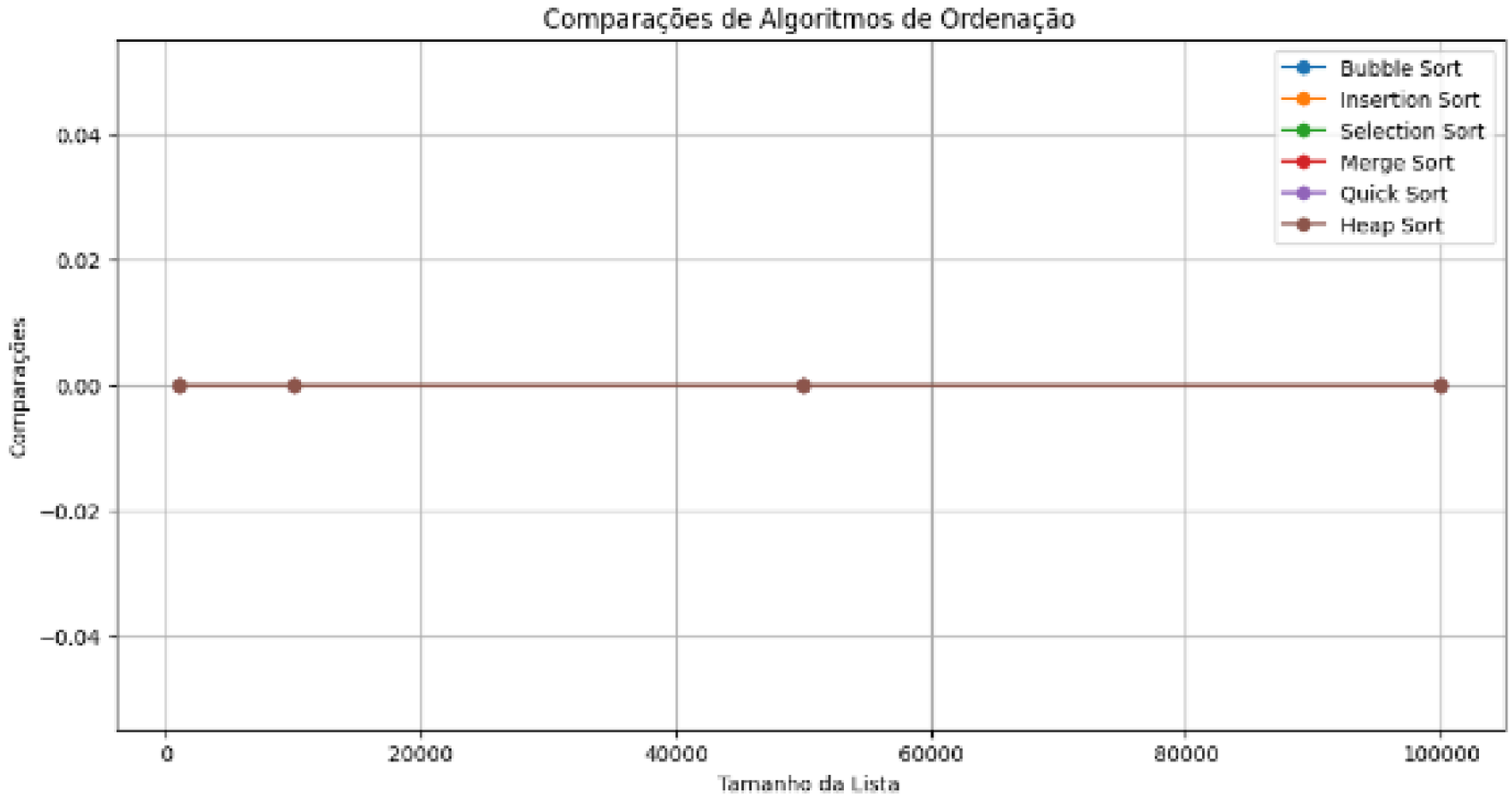
Comparação de Resultados



Quase todos os algoritmos apresentaram o número de comparações relativos com o tamanho da lista de entrada, os que apresentaram variações são os que realizam checagem de ordenarem dentro da lista na hora da comparação.

Comparação de Resultados

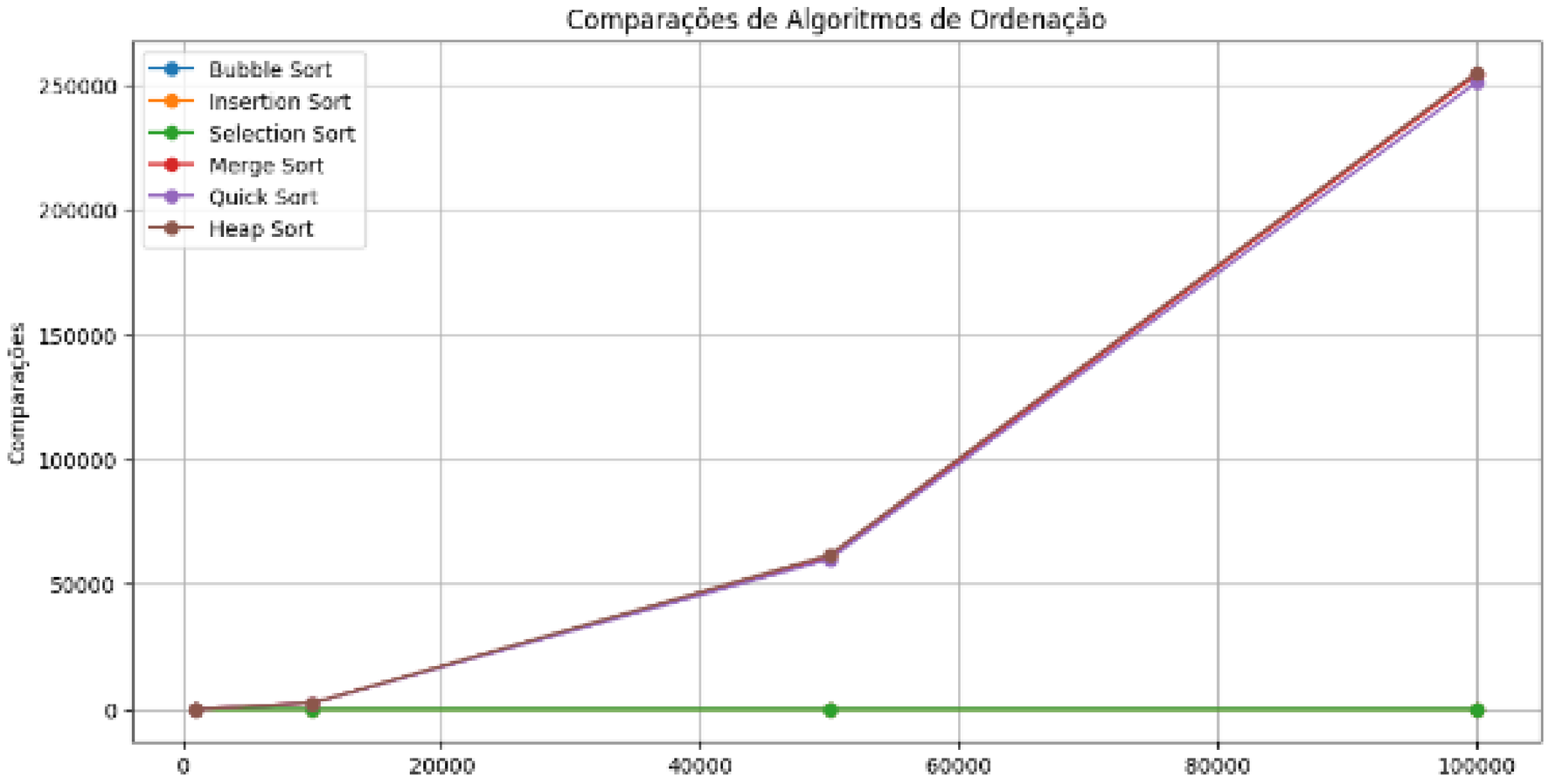
Figura 2 – Gráfico de trocas realizadas com lista ordenada



Como esperado, a quantidade de trocas realizadas pelos algoritmos foi de 0. Isso está correto, pois, todas as listas recebidas pelos algoritmos já estavam previamente ordenadas.

Comparação de Resultados

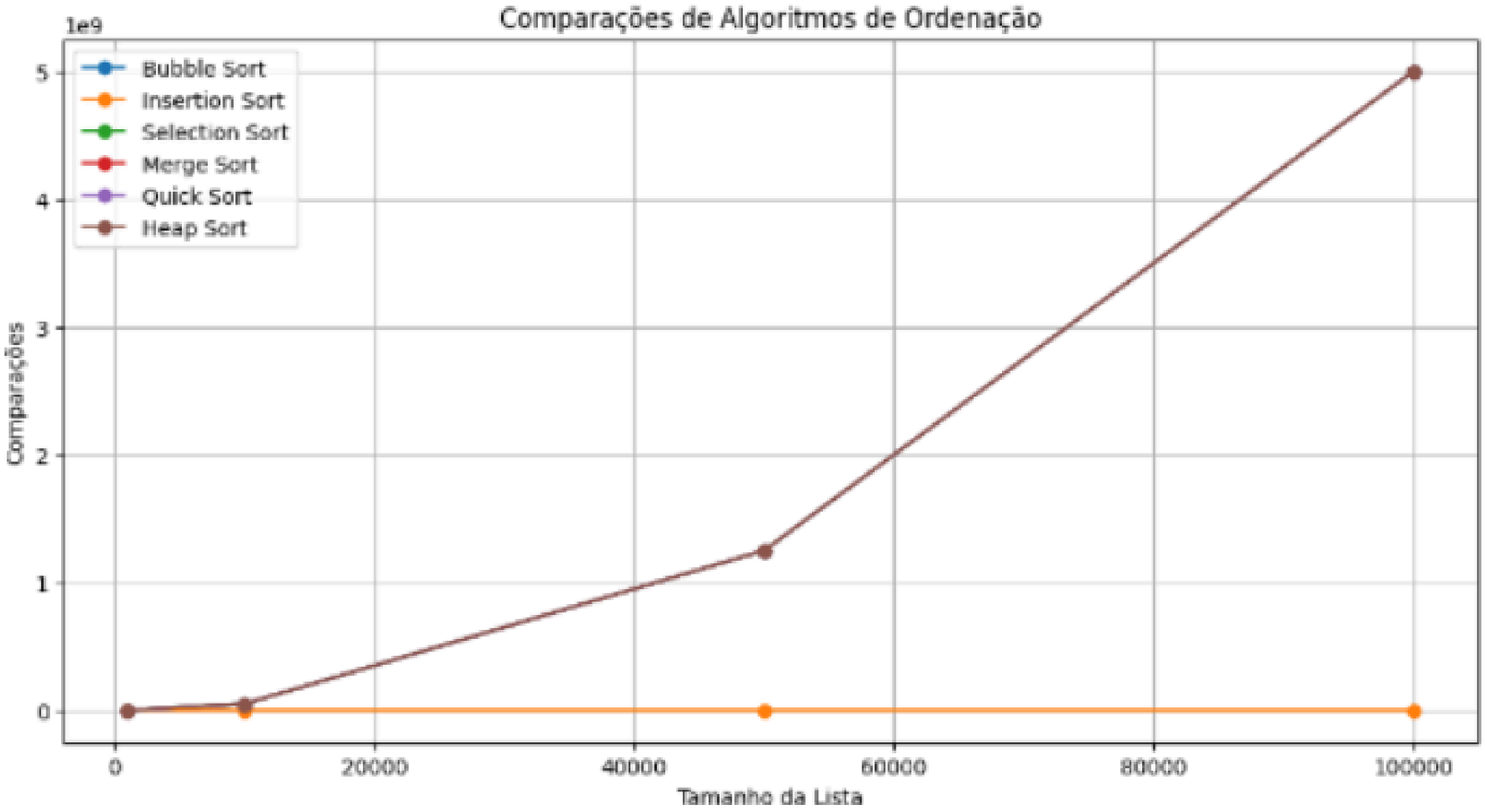
Figura 3 – Gráfico de tempo de execução dos algoritmos com lista ordenada



Percebe-se que os algoritmos, na maioria dos testes, rodaram no tempo bem abaixo dos 50.000 milissegundos, porém, algoritmos como o Quick e Heap, ultrapassaram esse valor quando a lista era de 50.000 ou 100.000 elementos, totalizando um tempo de execução superior a 250.000 milissegundos.

Comparação de Resultados

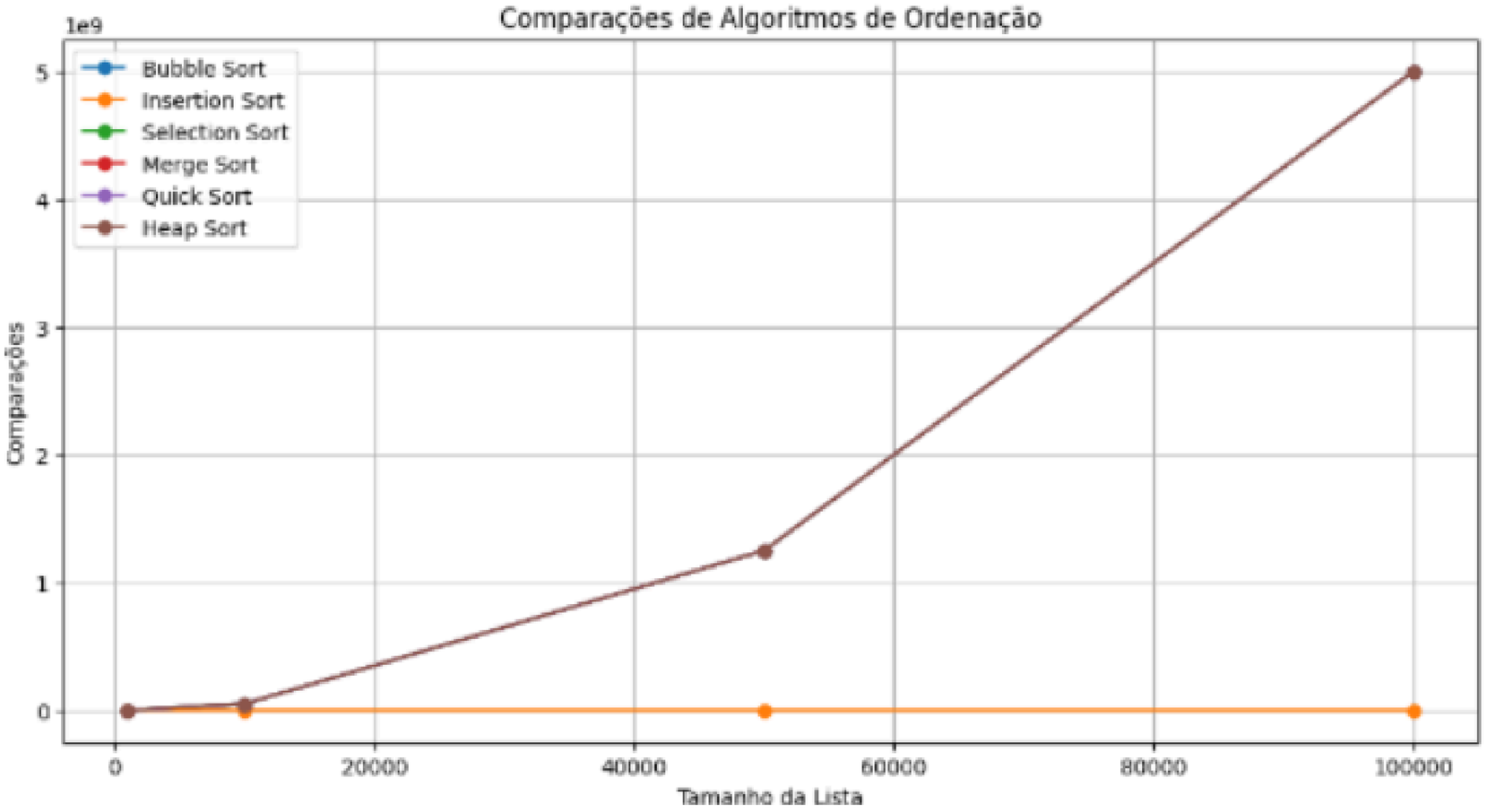
Figura 4 – Gráfico de comparações realizadas com lista ordenada inversamente



Com a lista de entrada ordenada inversamente, mostra-se que alguns algoritmos como o Insertion Sort realizaram pouquíssimas comparações quando comparadas com o Heap, que acabou chegando em mais de 500.000 comparações realizadas.

Comparação de Resultados

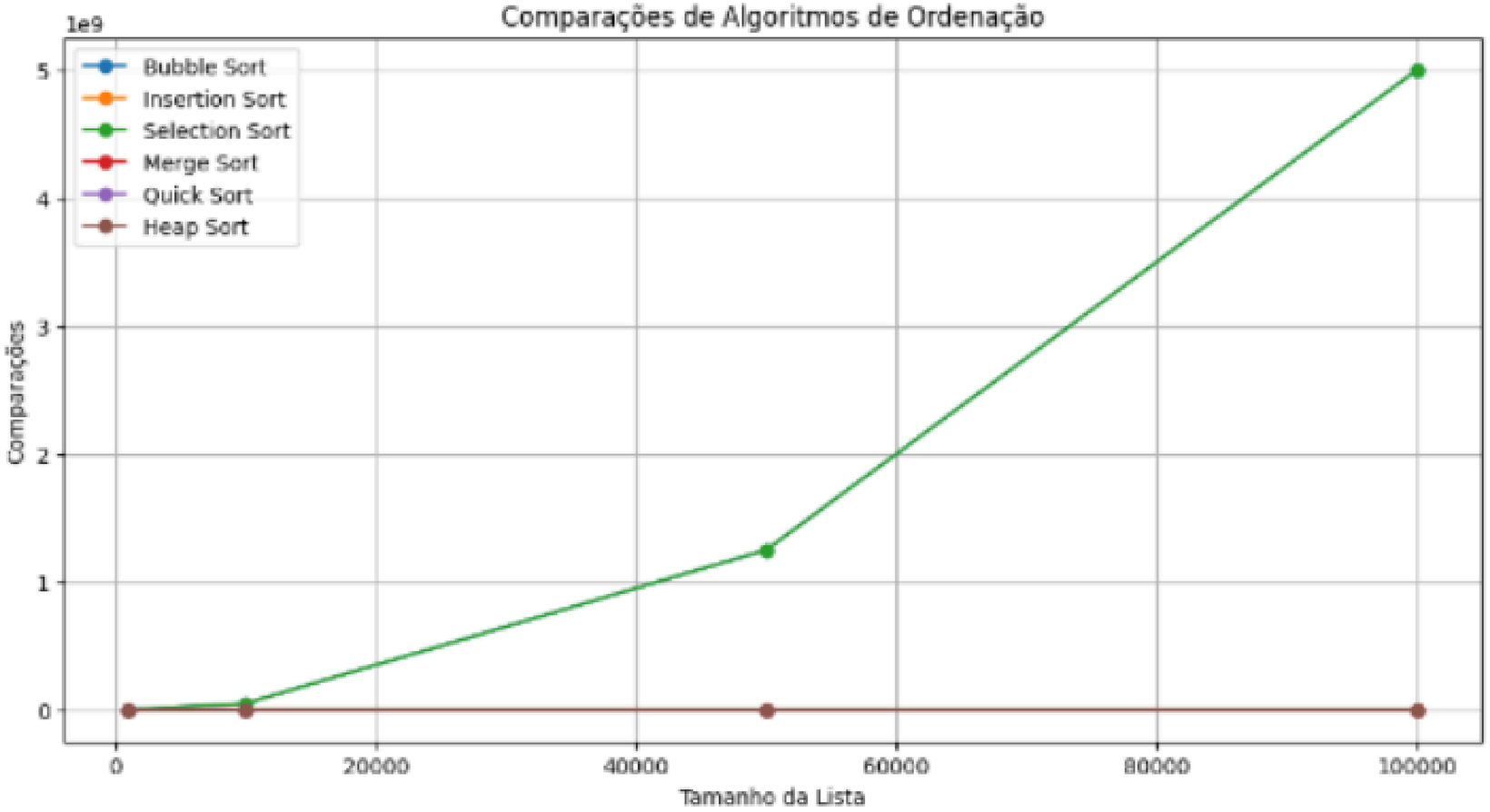
Figura 4 – Gráfico de comparações realizadas com lista ordenada inversamente



Com a lista de entrada ordenada inversamente, mostra-se que alguns algoritmos como o Insertion Sort realizaram pouquíssimas comparações quando comparadas com o Heap, que acabou chegando em mais de 500.000 comparações realizadas.

Comparação de Resultados

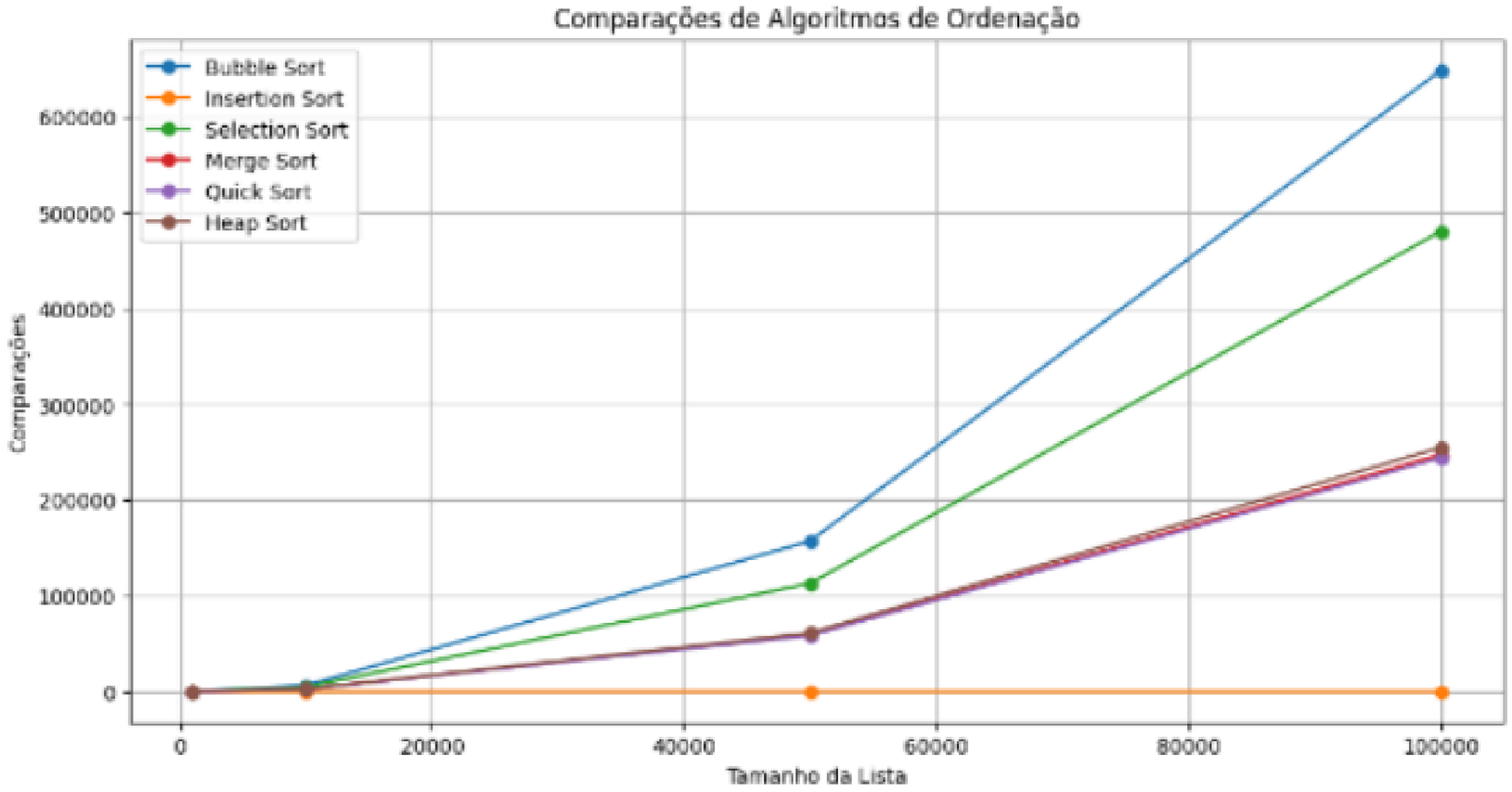
Figura 5 – Gráfico de trocas realizadas com lista ordenada inversamente



Vê-se uma grande diferença quando comparamos com o gráfico de trocas quando a lista já vem previamente ordenada, onde nesse gráfico com a lista ordenada inversamente apresenta inúmeras trocas realizadas, ultrapassando a faixa dos 500000.

Comparação de Resultados

Figura 6 – Gráfico de tempo de execução dos algoritmos com lista ordenada inversamente



Percebe-se a diferença do tempo de execução de cada algoritmo, onde quando o tamanho da lista era de 1000 elementos, a diferença de tempo era irrisória, porém com o aumento da quantidade de elementos podemos ver uma sequência bem definida de tempo.