Universidade Estadual Paulista Júlio de Mesquita Filho
Faculdade de Ciências e Tecnologia
Presidente Prudente
Ciência da Computação

Nathan Alves da Cruz Silveira
Lucas Henrique Gregorio

PROJETO DE ANÁLISE DE ALGORITMOS:
ANÁLISE EMPÍRICA DE ALGORITMOS DE ORDENAÇÃO

Presidente Prudente
2025

SUMÁRIO

1. METODOLOGIA DO TRABALHO E FUNDAMENTAÇÃO TEÓRICA

O presente trabalho tem como objetivo realizar um estudo comparativo entre diferentes algoritmos de ordenação, analisando seus desempenhos práticos e relacionando-os com suas complexidades teóricas. Para isso, foram conduzidos experimentos empíricos controlados, com foco na medição do tempo de execução em diferentes cenários e tamanhos de entrada.

Espera-se, ao final do estudo, obter resultados que confirmem as previsões teóricas sobre o desempenho relativo de cada algoritmo, validando suas eficiências em termos de tempo de execução e comportamento diante de diferentes distribuições de dados. Além disso, busca-se fornecer uma visão prática e quantitativa sobre como a complexidade algorítmica se manifesta em ambientes reais de execução.

1.1. REFERÊNCIA COMPARATIVA

A fundamentação teórica apoia-se em conceitos de análise de complexidade computacional, que descreve o comportamento assintótico dos algoritmos em função do tamanho da entrada (n). Essa análise teórica é comparada aos resultados empíricos, buscando validar o comportamento esperado de cada método sob diferentes condições experimentais.

Portanto, as referências utilizadas são:

- ELER, Danilo Medeiros. ***Análise de Algoritmos de Ordenação***. Presidente Prudente: FCT/UNESP – Departamento de Matemática e Computação, [s.d.]. Material de aula (PAA – Projeto e Análise de Algoritmos).
- SETIAWAN, R. ***Comparing Sorting Algorithm Complexity Based on Control Flow Structure***. 2016 International Conference on Information Management and Technology (ICIMTech), Bandung, Indonesia, 2016, pp. 224–228. DOI: 10.1109/ICIMTech.2016.7930334.
- CHAUHAN, Yash; DUGGAL, Anuj. ***Different Sorting Algorithms Comparison Based upon the Time Complexity***. Journal of Emerging Technologies and Innovative Research, v. 7, 2020. DOI: 10.1729/Journal.24472.
- SILVA, V. F.; OLIVEIRA, M. **Revisão dos Algoritmos de Ordenação com Aplicação em Busca Binária e Sequencial**. IFSULDEMINAS – Instituto Federal da Ciência e Tecnologia do Sul de Minas Gerais, 2018.

1.2. MÉTODO DE ANÁLISE

A metodologia adotada envolveu a implementação dos algoritmos em linguagem Python, escolhida por sua clareza sintática, ampla difusão no meio acadêmico e facilidade de análise experimental. Fez-se uso da IDE PyCharm para a implementação dos algoritmos e realização dos testes experimentais.

Todos os testes foram executados em um mesmo ambiente computacional, garantindo condições equitativas para todos os algoritmos. As especificações da máquina utilizada estão descritas abaixo:

- Processador (GPU Integrada): AMD Ryzen 4600H;
- Sistema Operacional: Windows 10;
- RAM: 8 GB;

Testamos os algoritmos em diferentes casos: com valores crescentes, decrescentes e aleatórios. Todos algoritmos foram executados 10 vezes, para maior acurácia. Dentre os algoritmos de ordenação, os selecionados foram:

- Bubble-Sort;
  - Otimizado e original.
- Merge-Sort;
- Shell-Sort;
- Quick-Sort;
  - Pivô primeiro.
  - Pivô central.
- Selection-Sort;
- Insertion-Sort;
- Heap-Sort.

2. ANÁLISE DE RESULTADOS

A principal metodologia empregada para a análise dos resultados consiste na apresentação visual e comparativa dos dados obtidos, permitindo uma avaliação mais clara do desempenho dos algoritmos estudados.

Essa abordagem possibilita observar tanto as diferenças de eficiência entre algoritmos distintos quanto as semelhanças de comportamento entre aqueles que compartilham estruturas ou princípios de implementação semelhantes. Por meio de gráficos e tabelas, busca-se facilitar a interpretação dos

resultados e a identificação de padrões que corroboram com as análises teóricas de complexidade. Os códigos utilizados estão ao fim do trabalho, no tópico 3.

2.1. ANÁLISE DE EFICIÊNCIA

Nos tópicos seguintes, analisaremos a eficiência de cada algoritmo, apresentando sua complexidade, juntamente com os resultados dos testes práticos de forma visual e clara. Os testes foram estruturados para capturar o desempenho em diferentes cenários: o melhor caso (tipicamente a lista já ordenada em ordem crescente), o pior caso (frequentemente a lista em ordem decrescente) e o caso médio (gerado com números aleatórios até o valor 100 para o vetor). Todos os algoritmos foram executados 10 vezes para maior acurácia.

É importante notar que, em alguns algoritmos, a definição de melhor e pior caso não se limita estritamente à ordem crescente ou decrescente, sendo definida pela entrada que maximiza ou minimiza o tempo de execução.

2.1.0. BUBBLE-SORT ORIGINAL

Um dos algoritmos mais simples. Percorre repetidamente a lista, compara elementos adjacentes e os troca se estiverem na ordem errada, fazendo com que os maiores elementos "flutuem" para o final (Silva & Oliveira, 2018).

> Similar to selection sort algorithm, there are two loops using “while” loop and one selection using “if” selection in bubble sort algorithm. The first loop repeats n times minus one which n is the number of data that is sorted. “if” selection is used to compare the two neighboring data. (MAHAJA; GHRERA, 2016)

Temos suas complexidades definida a partir de:

- Pior caso: $O(n^2)$;
- Melhor caso: $\Omega(n^2)$;
- Caso médio: $\Theta(n^2)$.

Segue os resultados dos testes práticos, conforme especificado:

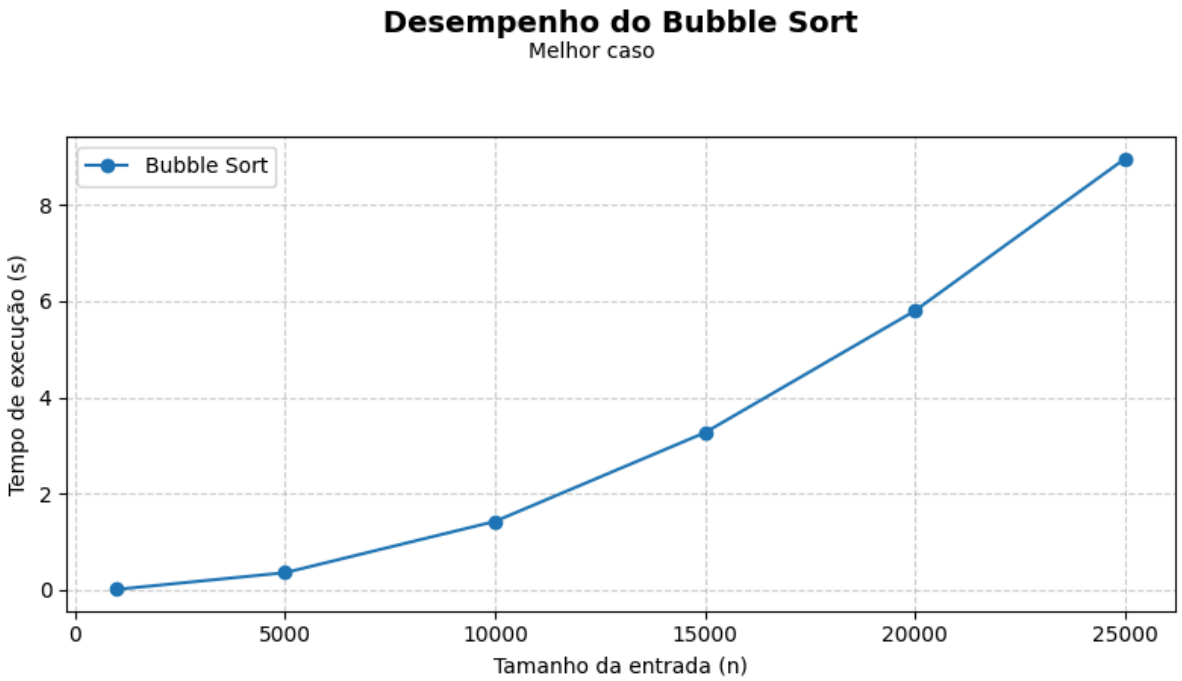


Figura 1: Melhor caso do Bubble Sort.

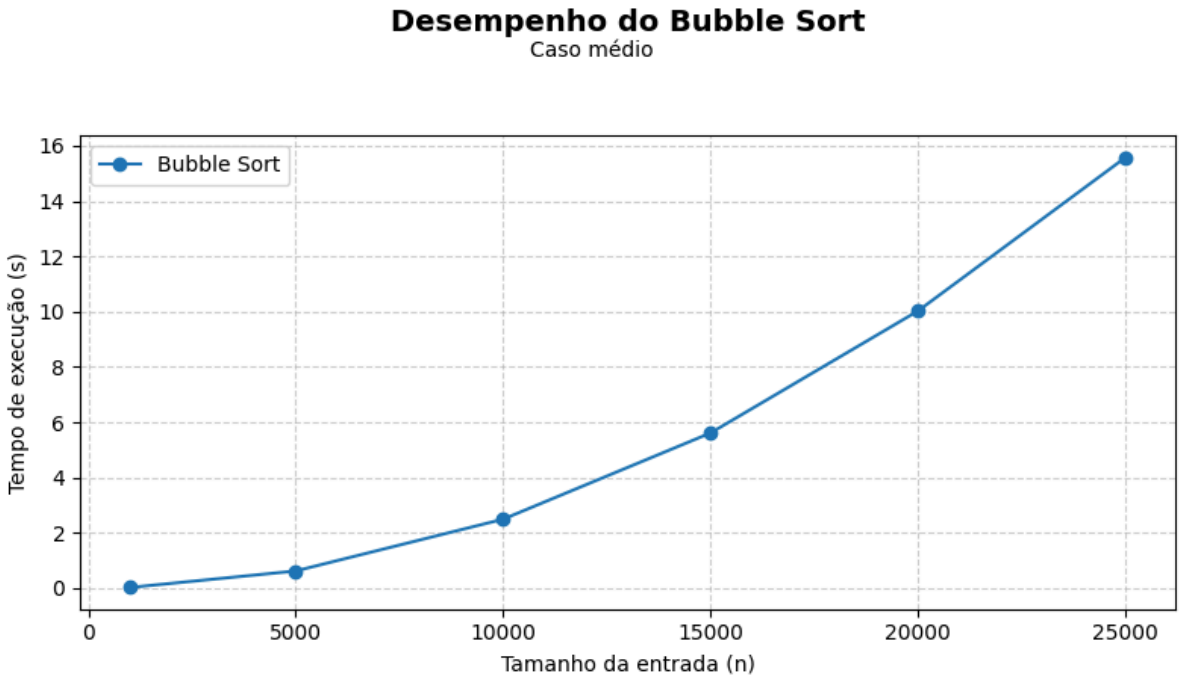


Figura 2: Caso médio Bubble Sort

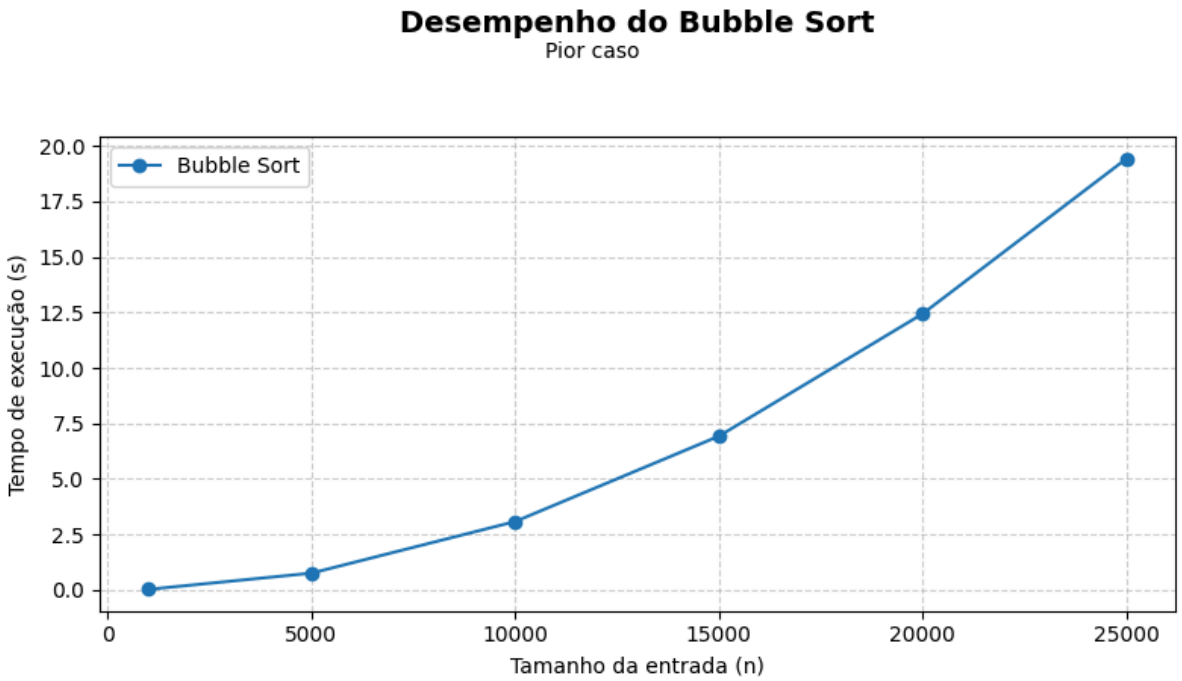


Figura 3: Pior caso Bubble Sort.

2.1.1. BUBBLE-SORT OTIMIZADO

Semelhante ao Bubble-Sort Original, mas inclui um flag (bandeira) para verificar se houve trocas em uma passagem. Se nenhuma troca for feita, a lista está ordenada, e o algoritmo para, melhorando a performance no melhor caso (MAHAJA; GHRERA, 2016).

Temos suas complexidades definida a partir de:

- Pior caso: $O(n^2)$;
- Melhor caso: $\Omega(n^2)$;
- Caso médio: $\Theta(n^2)$.

Segue os resultados dos testes práticos, conforme especificado:

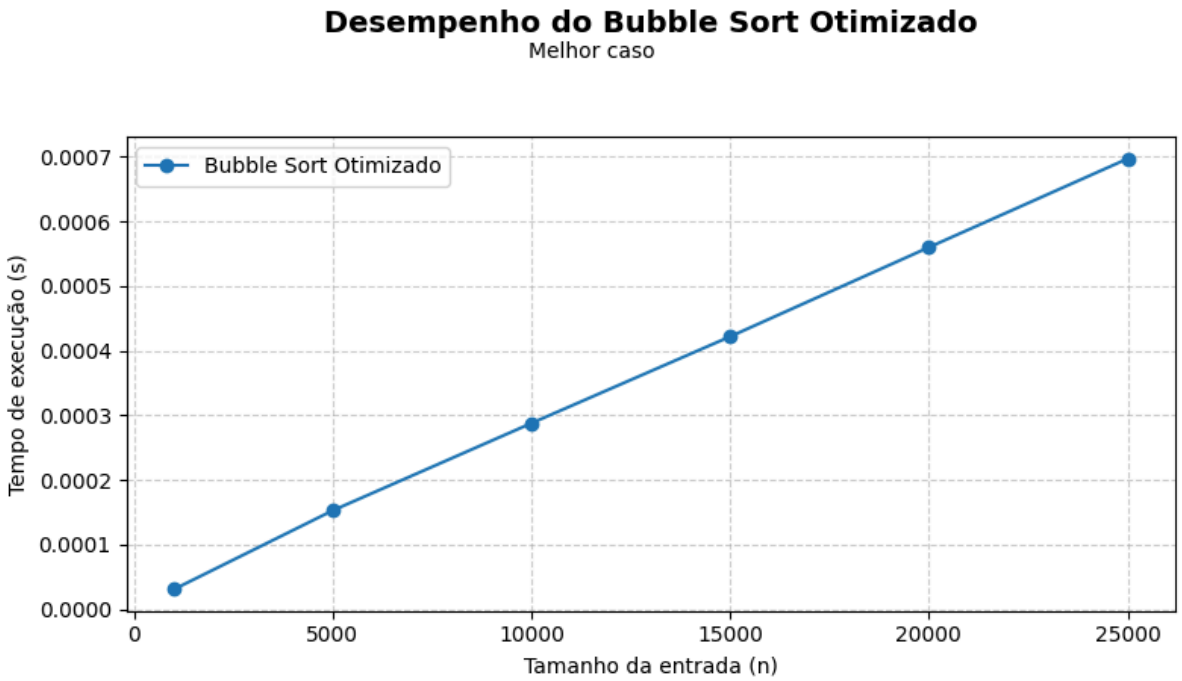


Figura 4: Melhor caso Bubble Sort Otimizado.

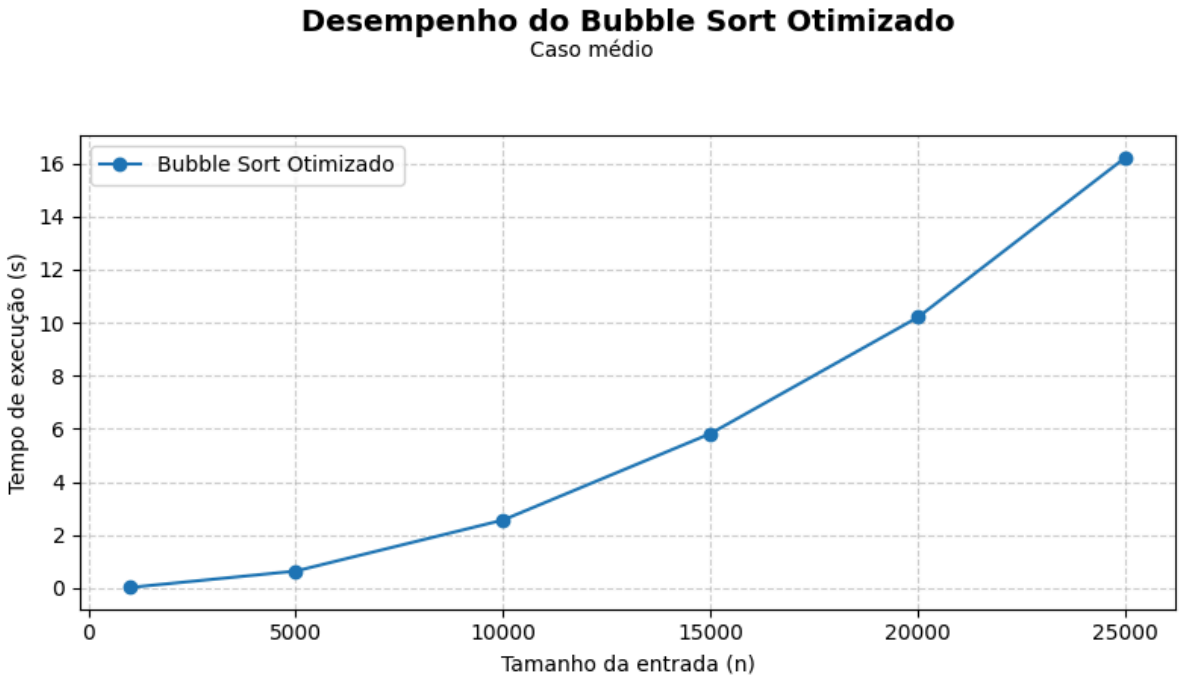


Figura 5: Caso médio Bubble Sort Otimizado.

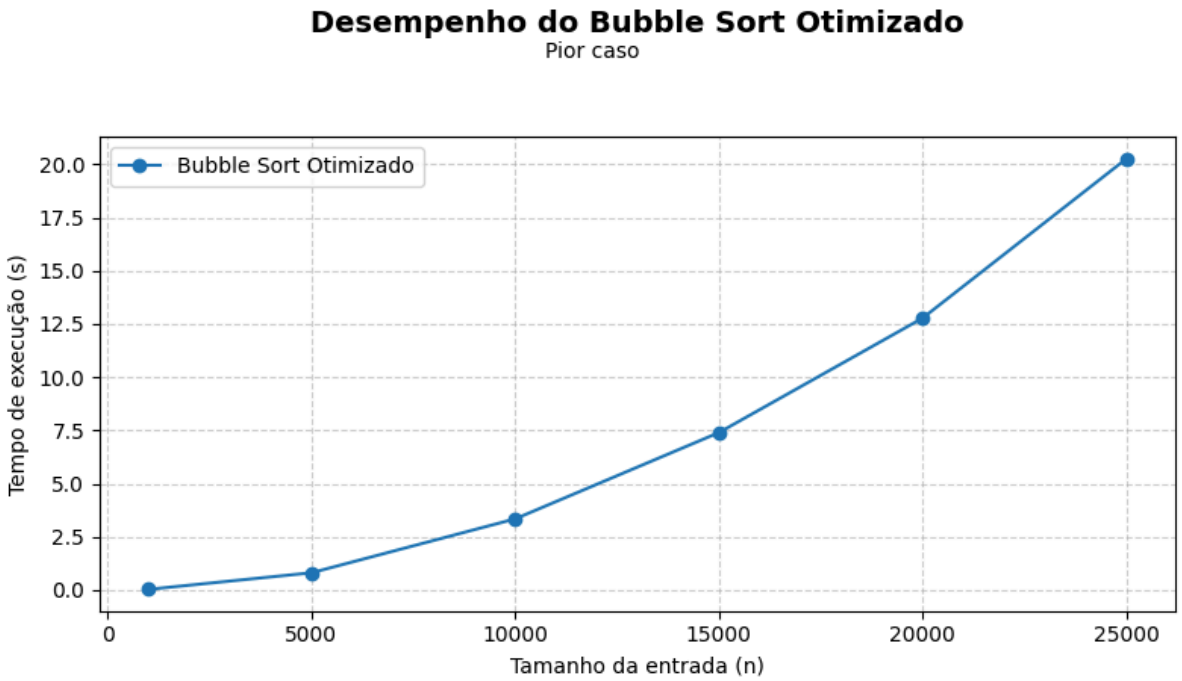


Figura 6: Pior caso Bubble Sort Otimizado.

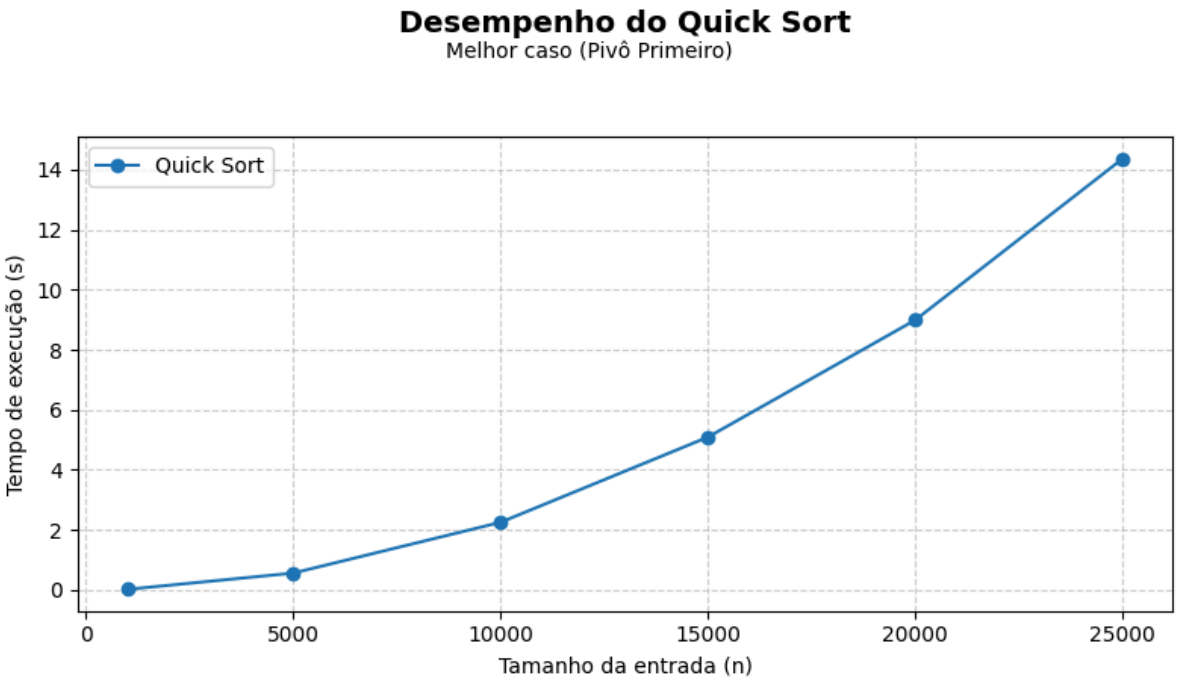
2.1.2. QUICK-SORT (PIVÔ PRIMEIRO)

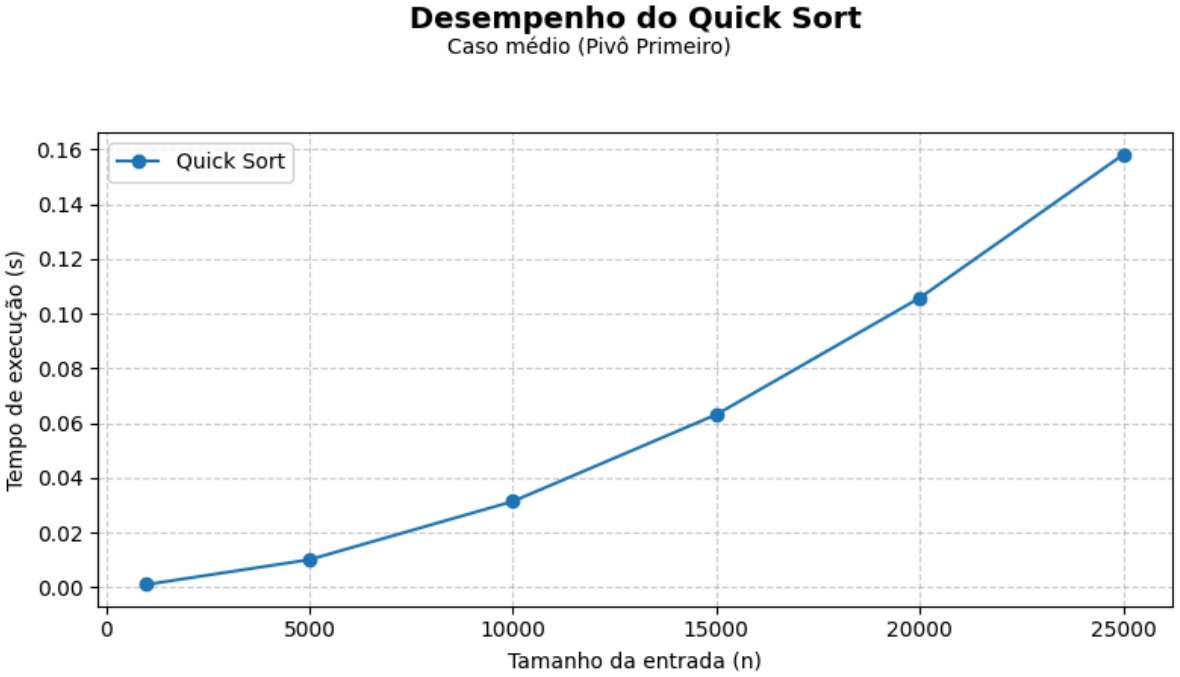
Implementa o paradigma Dividir e Conquistar. Escolhe o primeiro elemento como pivô e particiona o array em dois sub-arrays: elementos menores que o pivô e elementos maiores que o pivô. Recorre para ordenar os sub-arrays.

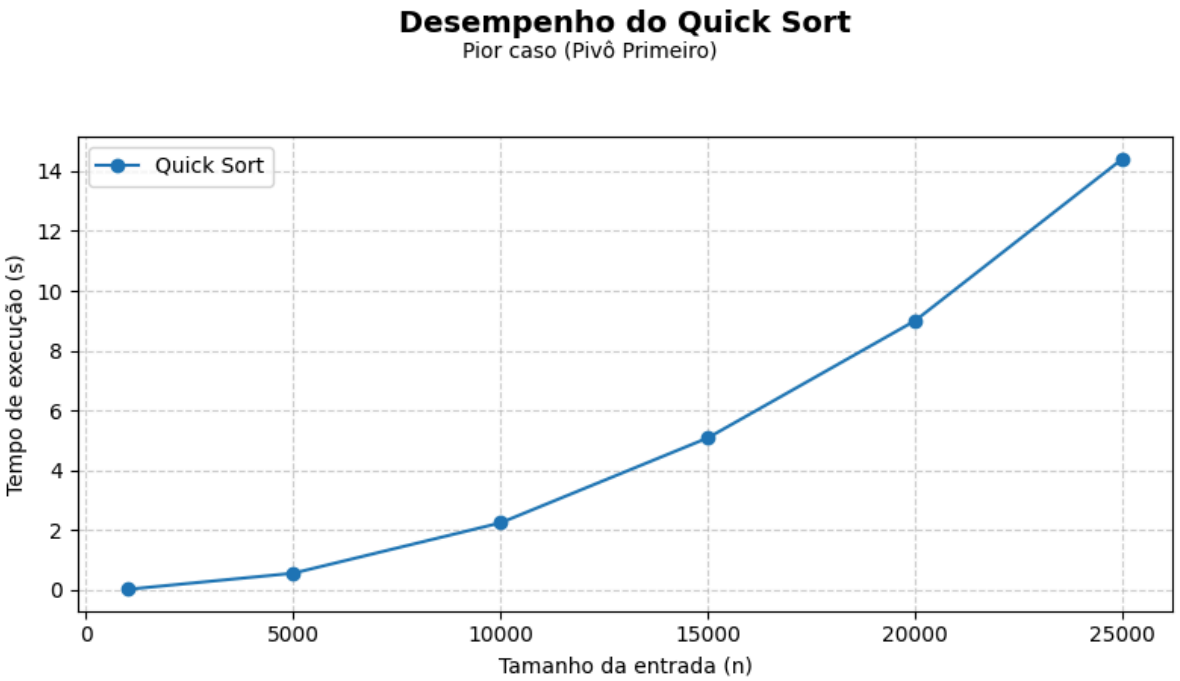
> Trabalha como o Merge Sort e realiza múltiplas chamadas recursivas, a cada chamada seleciona um elemento como pivô e divide vet[n] em duas partes [4], e para cada uma das partes faz a ordenação individual. [3] possui complexidade O($n\ log\ n$). (SILVA; OLIVEIRA, 2018)

Temos suas complexidades definida a partir de:

- Pior caso: $O(n^2)$;
- Melhor caso: $\Omega(n\ logn)$;
- Caso médio: $\Theta(n\ logn)$.


Desempenho do Quick Sort
Melhor caso (Pivô Primeiro)
Quick Sort
Tempo de execução (s)
Tamanho da entrada (n)
0
2
4
6
8
10
12
14
0
5000
10000
15000
20000
25000


Desempenho do Quick Sort
Caso médio (Pivô Primeiro)
Quick Sort
Tempo de execução (s)
Tamanho da entrada (n)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
0
5000
10000
15000
20000
25000



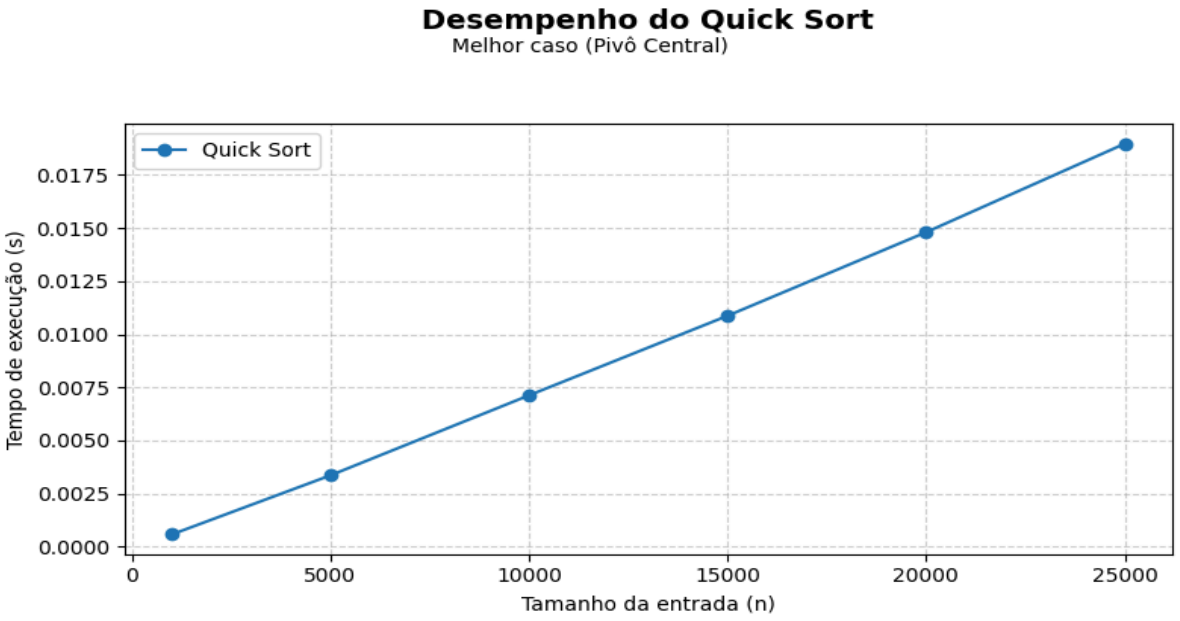
2.1.3. QUICK-SORT (PIVÔ-CENTRAL)

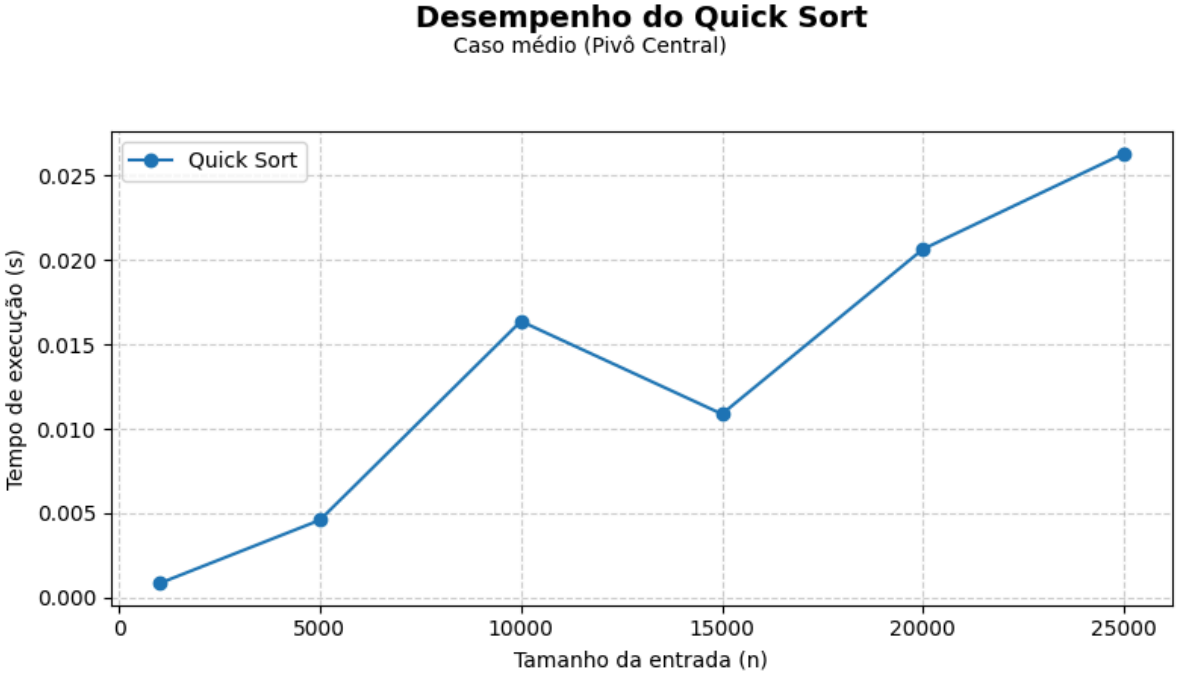
Muito semelhante ao Quick-Sort (Pivô Primeiro), também usa o paradigma Dividir e Conquistar e realiza múltiplas chamadas recursivas, selecionando um elemento como pivô e dividindo o vetor em duas partes, que são ordenadas individualmente. A diferença reside na escolha do pivô, onde o elemento central é selecionado. Esta abordagem visa reduzir a probabilidade do pior caso.

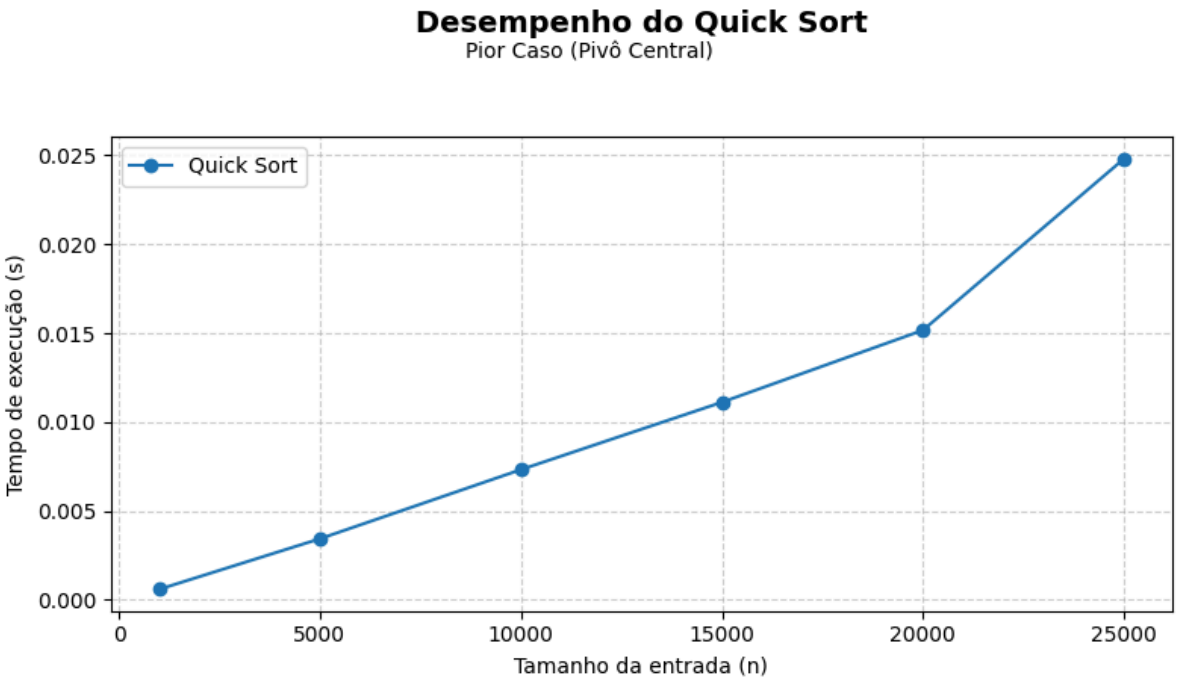
> Para remediar o pior caso, uma das estratégias é "escolher o pivot como sendo a mediana entre o primeiro, o elemento central e o último elemento do array" (BRITO, 2019, p. 5). (SILVA; OLIVEIRA, 2018)

Temos suas complexidades definida a partir de:

- Pior caso: $O(n^2)$;
- Melhor caso: $\Omega(n\ logn)$;
- Caso médio: $\Theta(n\ logn)$.


Desempenho do Quick Sort
Melhor caso (Pivô Central)
Quick Sort
Tempo de execução (s)
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
0
5000
10000
15000
20000
25000
Tamanho da entrada (n)


Desempenho do Quick Sort
Caso médio (Pivô Central)
Quick Sort
Tempo de execução (s)
0.000
0.005
0.010
0.015
0.020
0.025
0
5000
10000
15000
20000
25000
Tamanho da entrada (n)



2.1.4. INSERTION-SORT

Constroi a lista ordenada elemento por elemento. A cada iteração, retira um elemento da entrada e o insere na posição correta da sub-lista já ordenada, de forma análoga a ordenar cartas na mão (Silva & Oliveira, 2018).

Temos suas complexidades definida a partir de:

- Pior caso: $O(n^2)$;
- Melhor caso: $\Omega(n)$;
- Caso médio: $\Theta(n^2)$.

Segue os resultados dos testes práticos, conforme especificado:

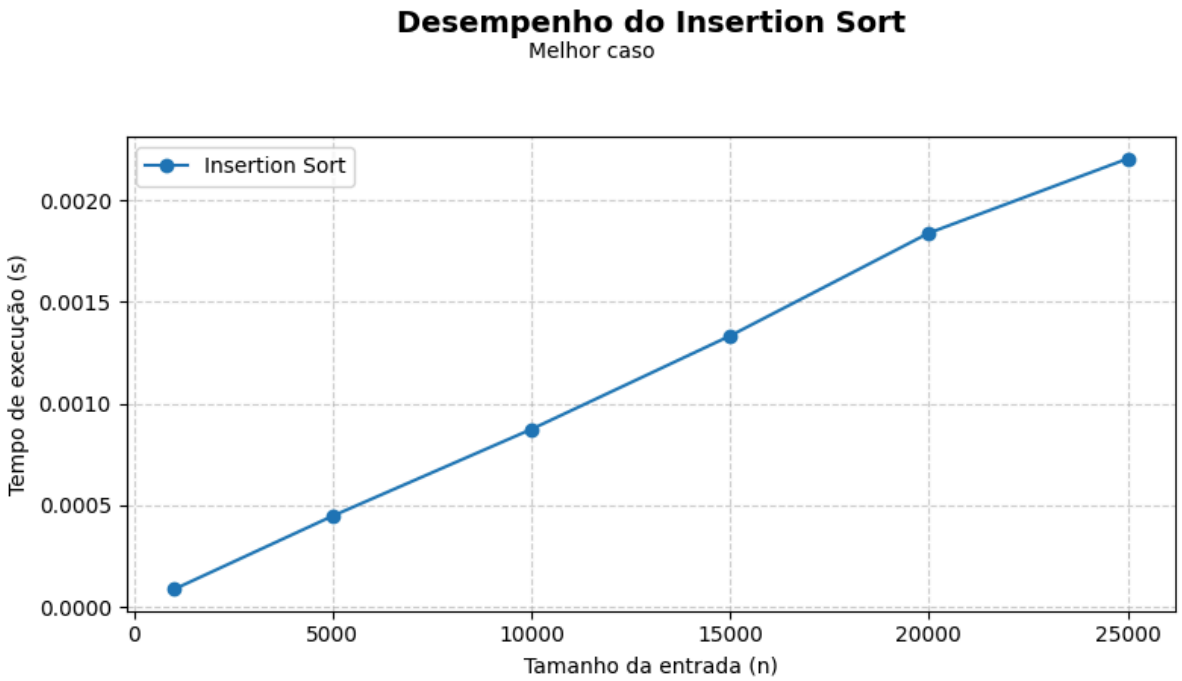


Figura 7: Melhor caso Insertion Sort.

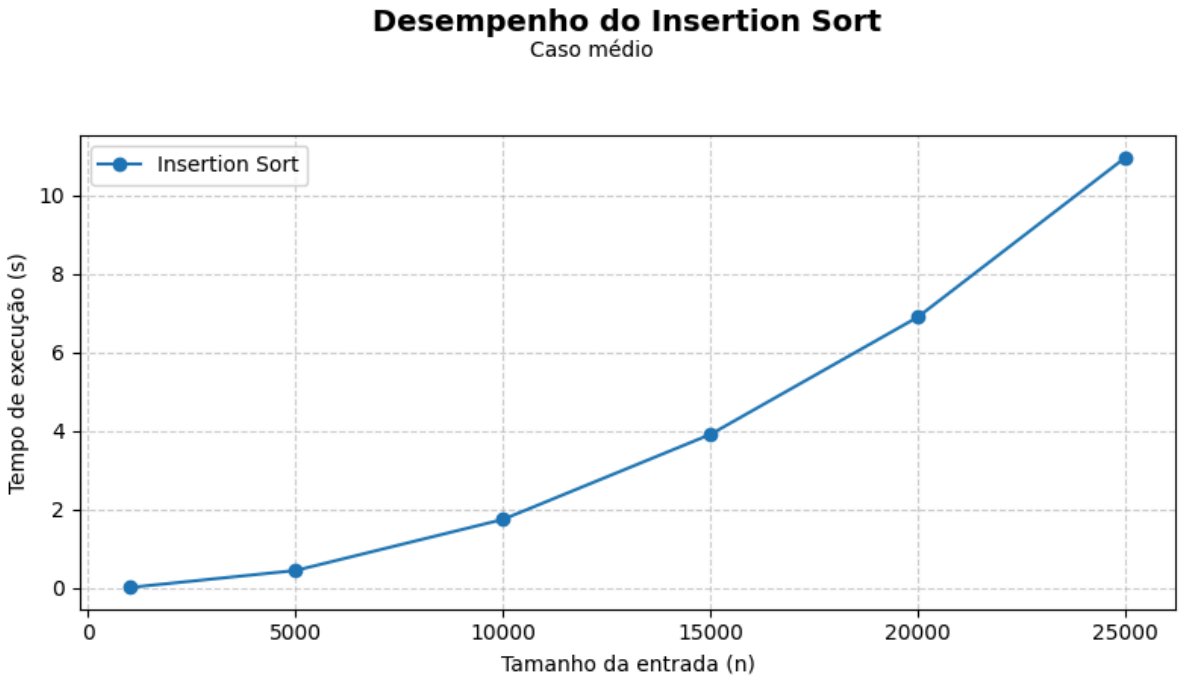


Figura 8: Caso médio Insertion Sort.

Figura 7: Caso médio Insertion Sort.

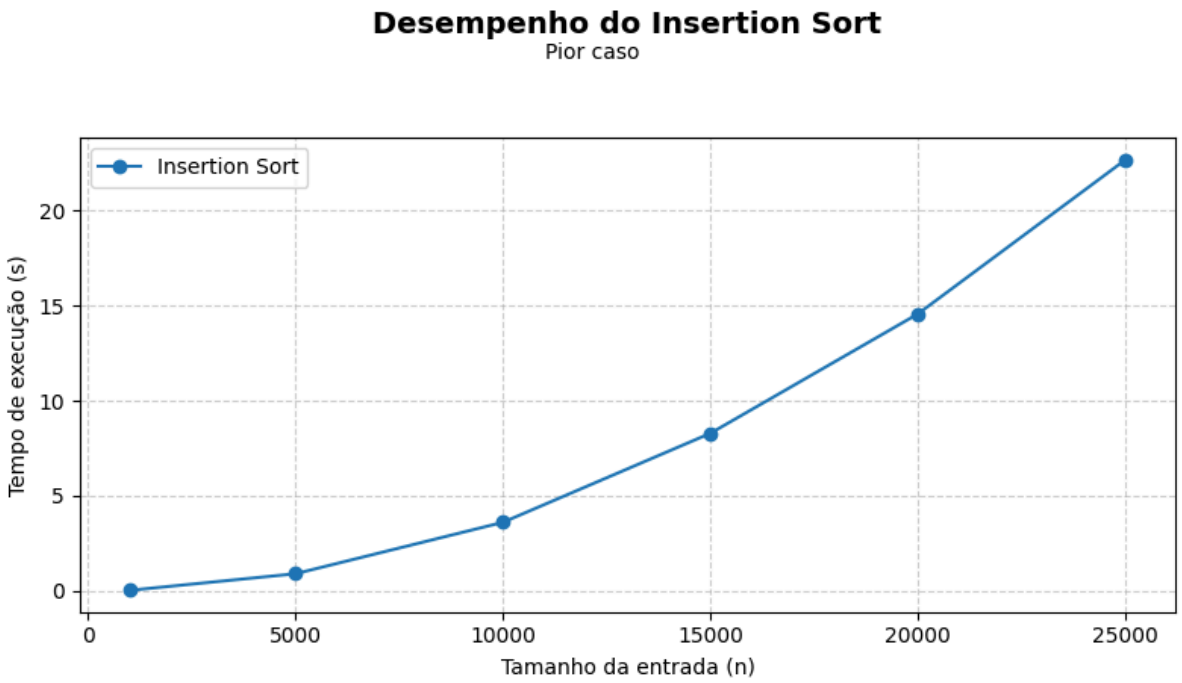


Figura 8: Pior caso Insertion Sort.

2.1.5. SHELL-SORT

É uma otimização do Insertion-Sort. Ordena elementos separados por um intervalo (gap), que é progressivamente reduzido. Ele permite que elementos distantes sejam movidos para o lugar mais rapidamente do que no Insertion Sort.

Temos suas complexidades definida a partir de:

- Pior caso: $O(n\ log^2 n)$;
- Melhor caso: $\Omega(n\ log n)$;
- Caso médio: $O(n^2)$.

Segue os resultados dos testes práticos, conforme especificado:

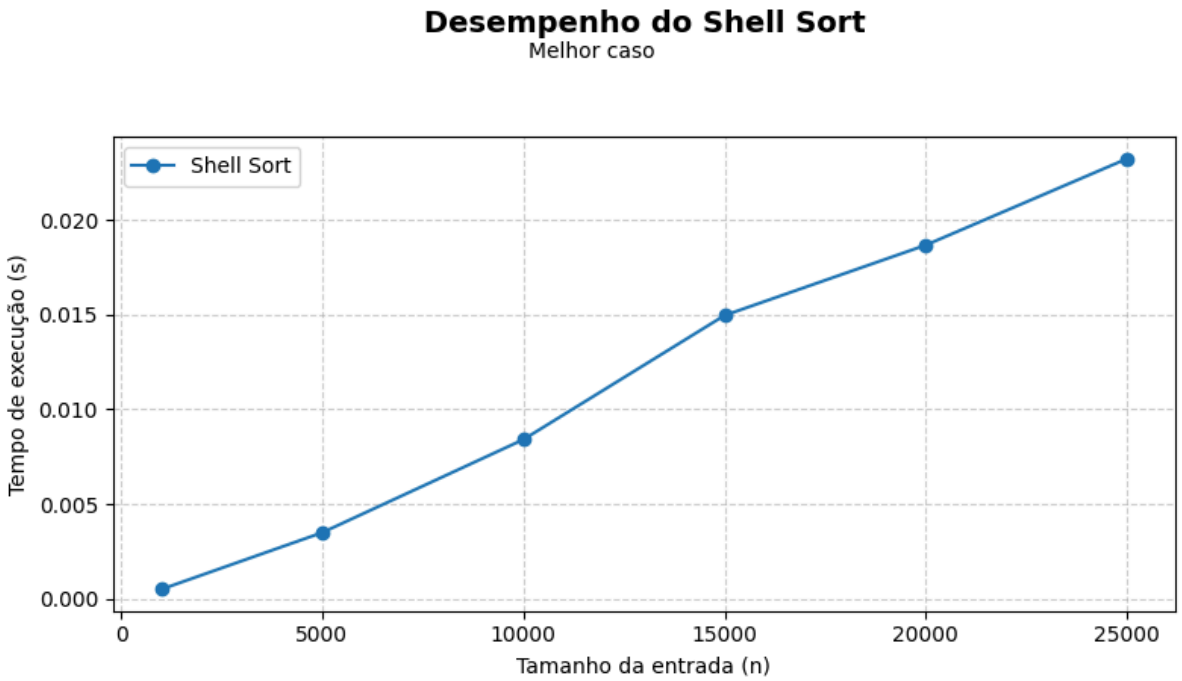


Figura 9: Melhor caso Shell Sort.

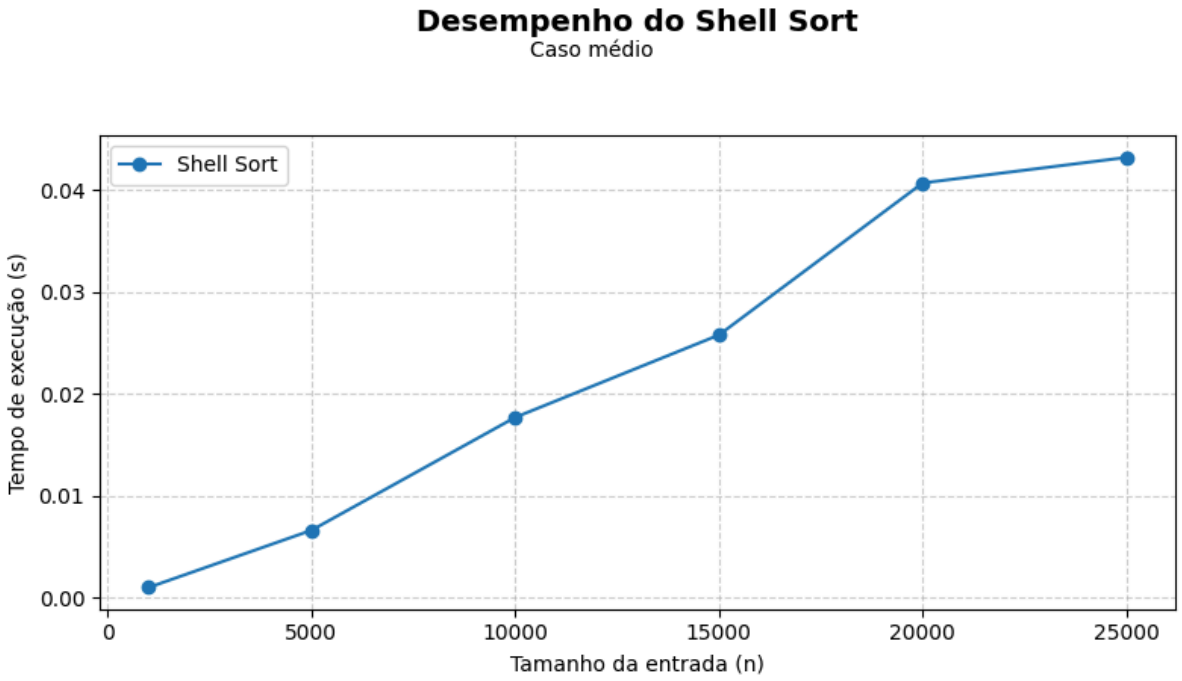


Figura 10: Caso médio Shell Sort.

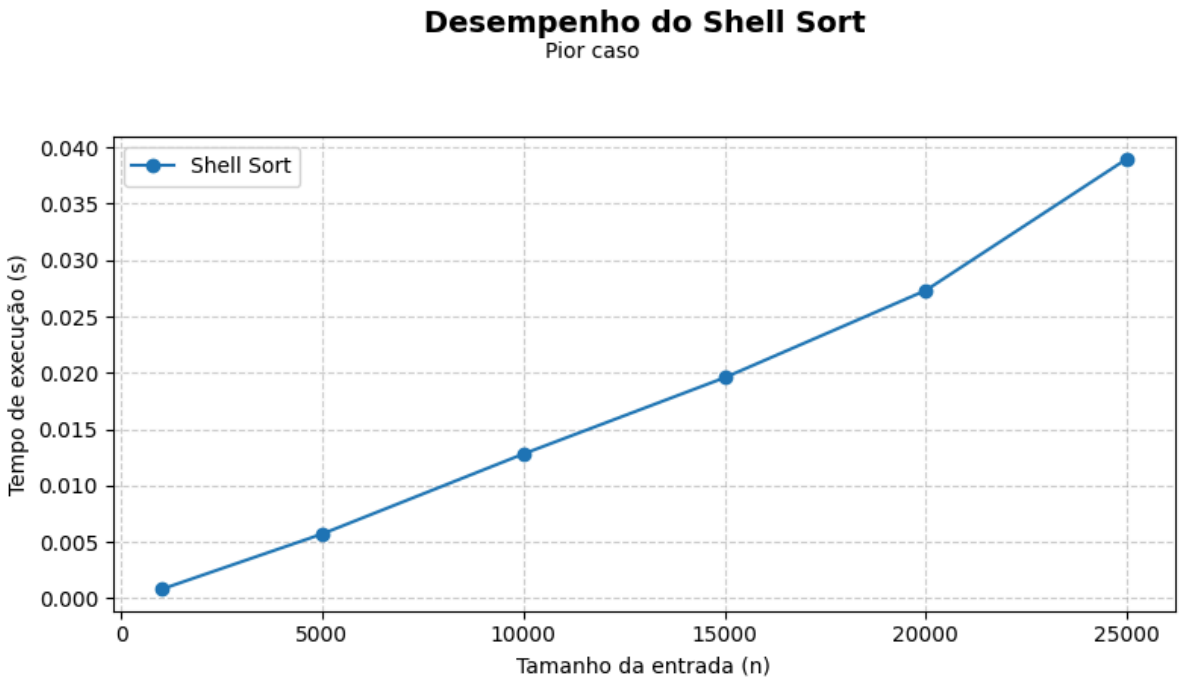


Figura 11: Pior caso Shell Sort.

2.1.6. SELECTION-SORT

Percorre a lista para encontrar o menor elemento (ou maior) e o troca com o elemento na primeira posição não ordenada. Repete este processo para o restante da lista (MAHAJA; GHRERA, 2016).

Temos suas complexidades definida a partir de:

- Pior caso: $O(n^2)$;
- Melhor caso: $\Omega(n^2)$;
- Caso médio: $\Theta(n^2)$.

Segue os resultados dos testes práticos, conforme especificado:

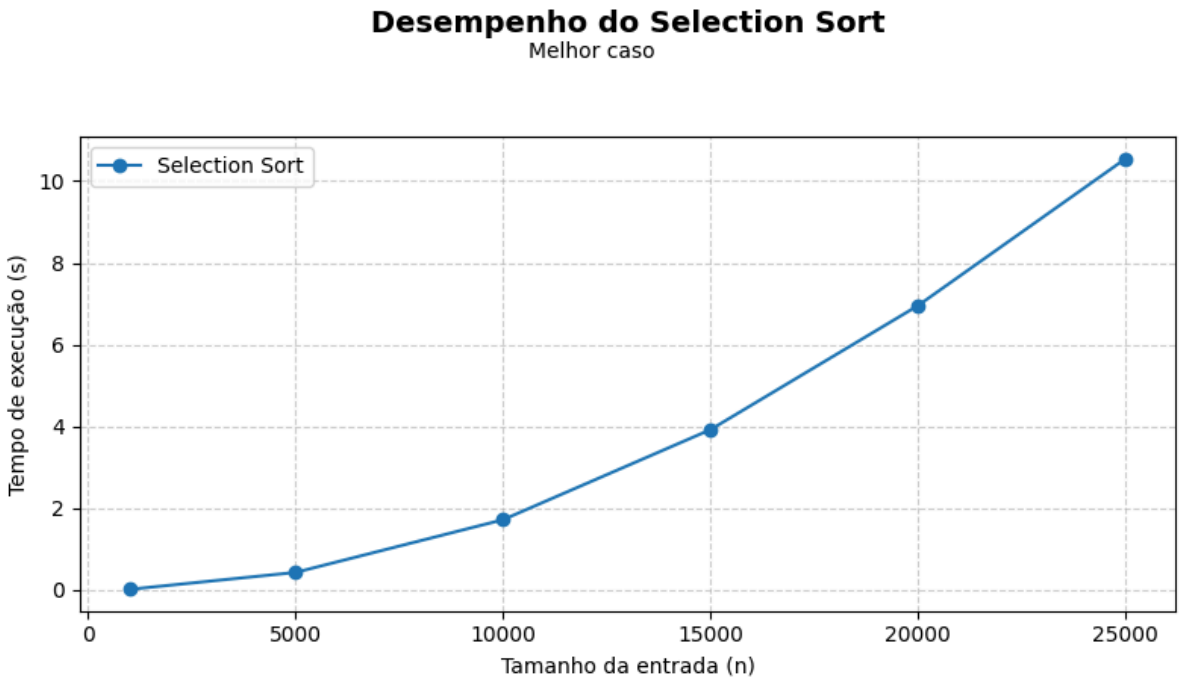


Figura 12: Melhor caso Selection Sort.

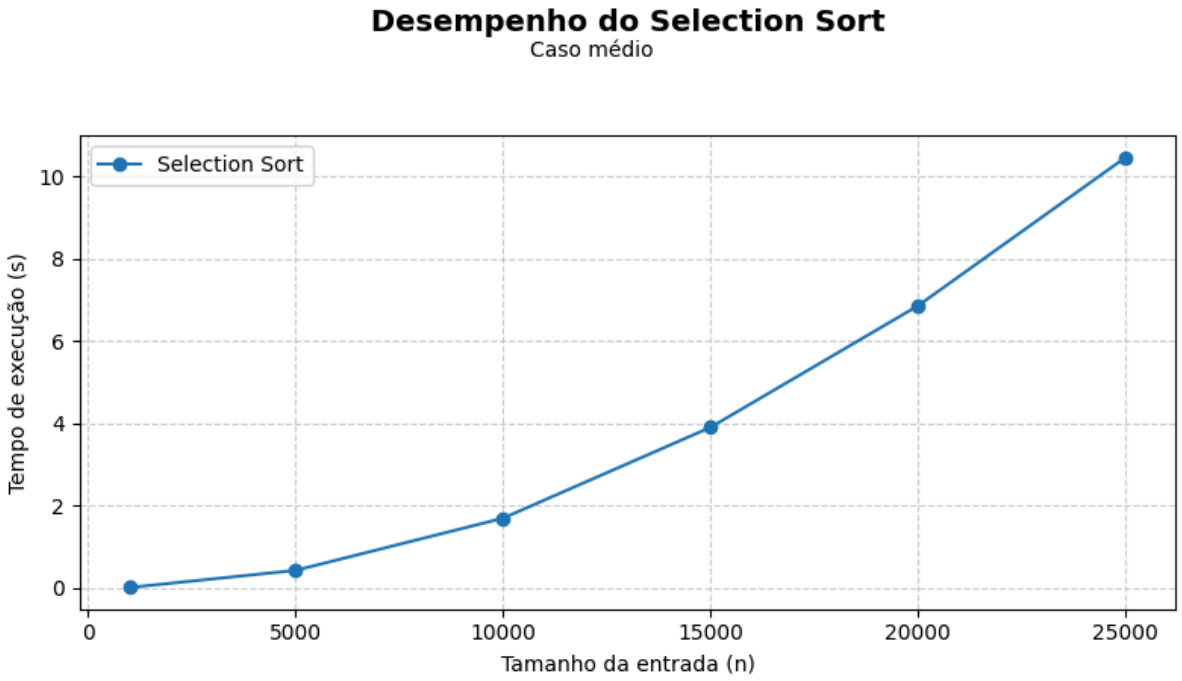


Figura 13: Caso médio Selection Sort.

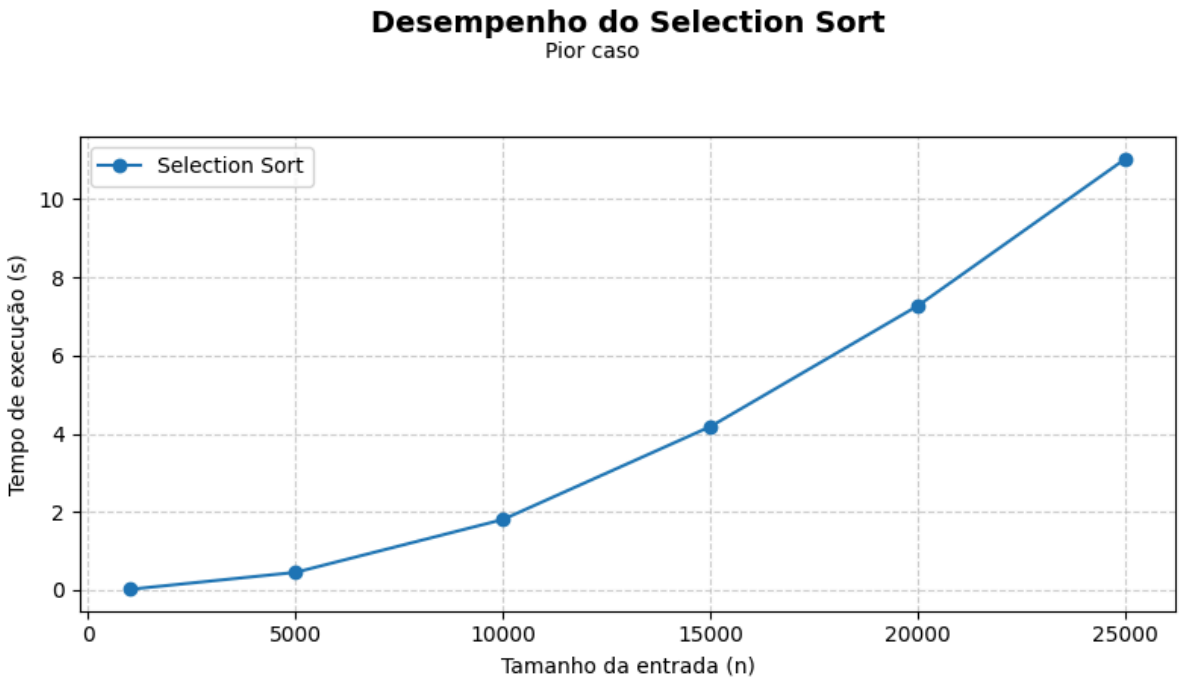


Figura 14: Pior caso Selection Sort.

2.1.7. HEAP-SORT

Algoritmo que utiliza uma estrutura de dados conhecida como Heap Binário (ou Max Heap). Constroi o heap e, em seguida, remove repetidamente o elemento máximo (a raiz), colocando-o na posição final do array ordenado.

Temos suas complexidades definida a partir de:

- Pior caso: $O(n\ logn)$;
- Melhor caso: $\Omega(n\ logn)$;
- Caso médio: $\Theta(n\ logn)$.

Segue os resultados dos testes práticos, conforme especificado:

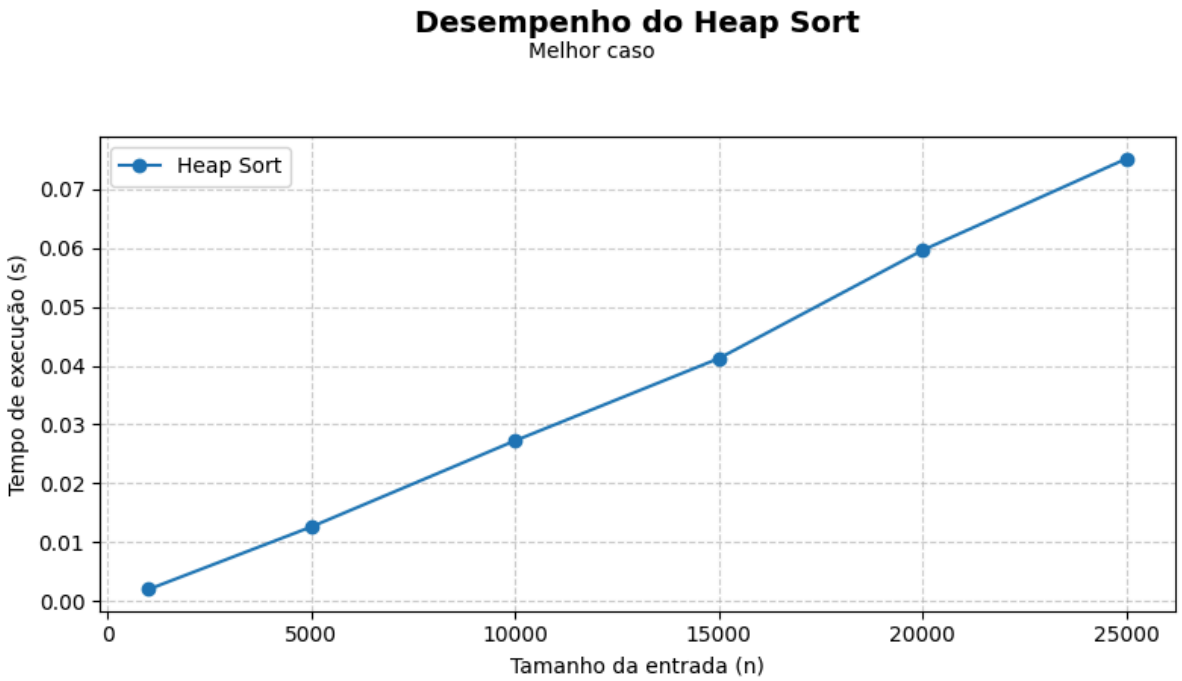


Figura 15: Melhor caso Heap Sort.

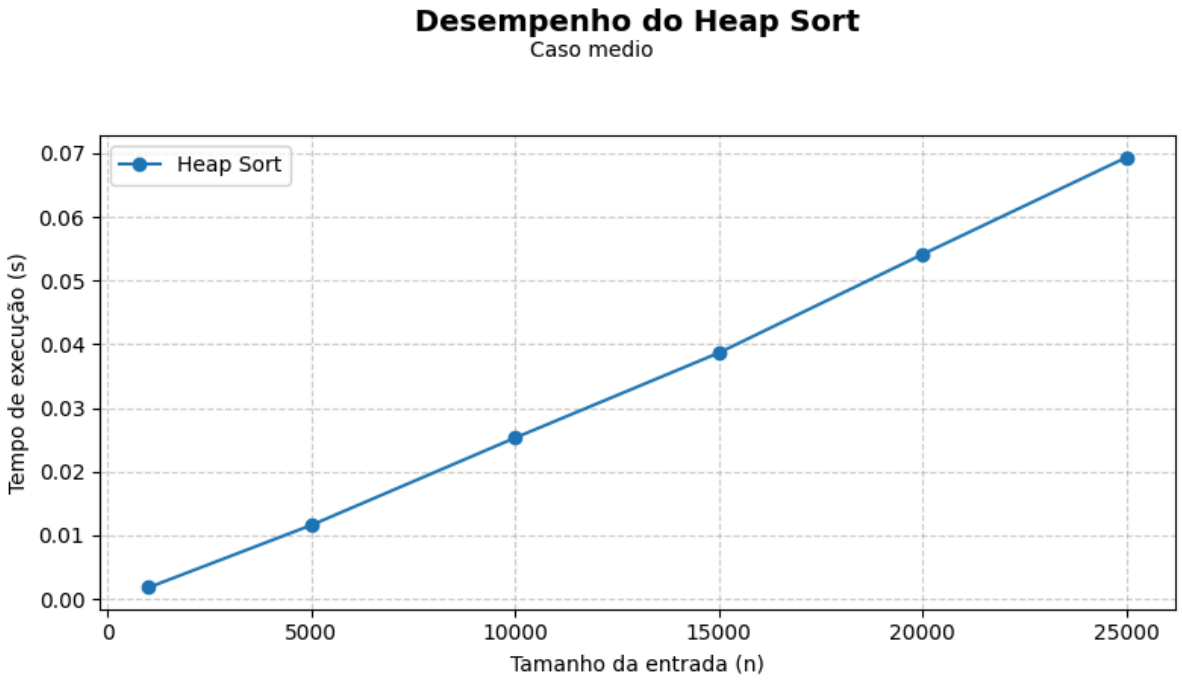


Figura 16: Caso médio Heap Sort.

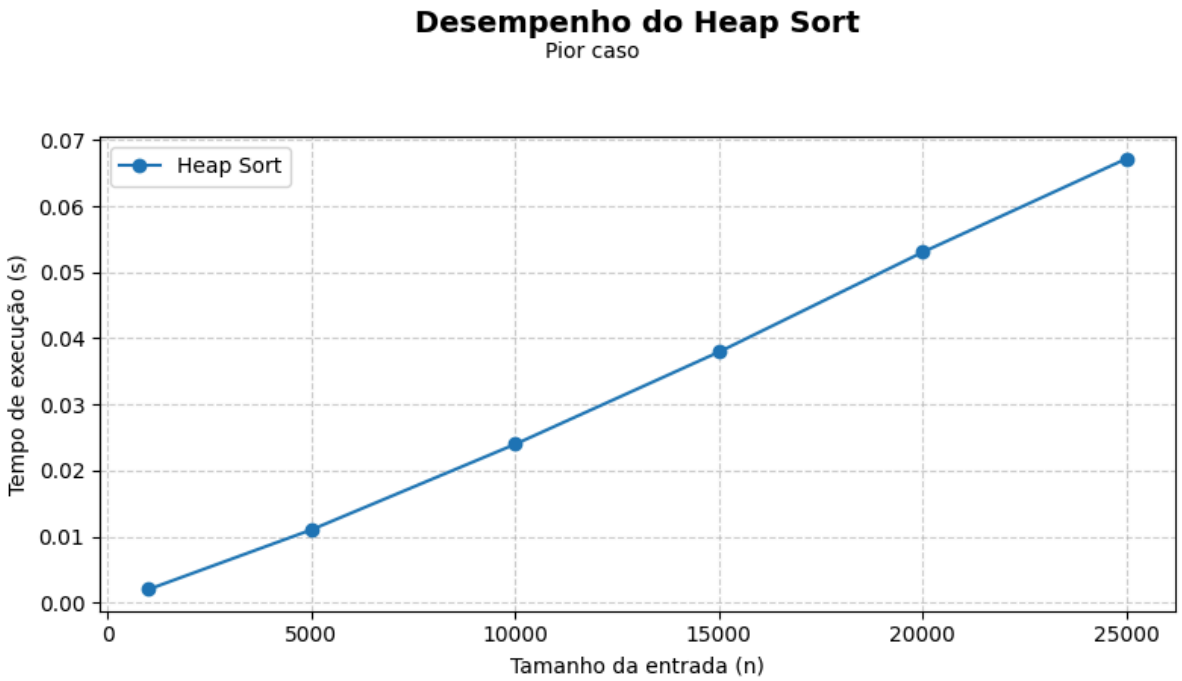


Figura 17: Pior caso Heap Sort.

2.1.8. MERGE-SORT

Algoritmo que utiliza uma estrutura de dados conhecida como Heap Binário (ou Max Heap). Constroi o Heap e, em seguida, remove repetidamente o elemento máximo (a raiz), colocando-o na posição final do array ordenado.

Temos suas complexidades definida a partir de:

- Pior caso: $O(n\ logn)$;
- Melhor caso: $\Omega(n\ logn)$;
- Caso médio: $\Theta(n\ logn)$.

Segue os resultados dos testes práticos, conforme especificado:

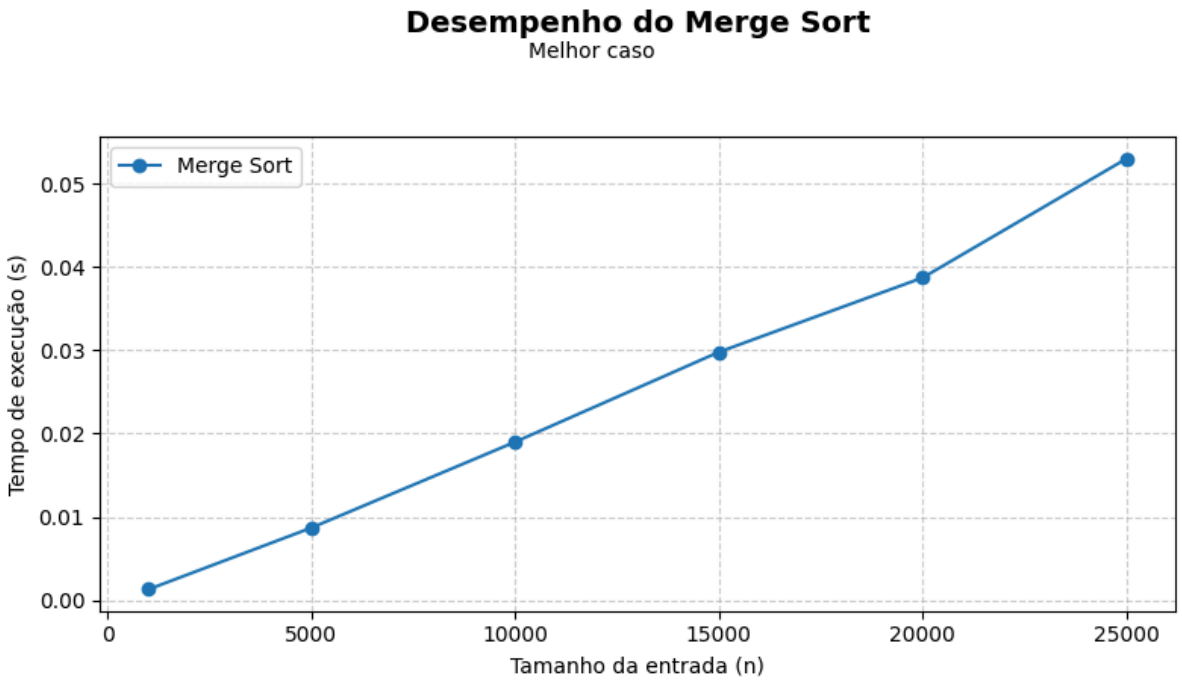


Figura 18: Melhor caso Merge Sort.

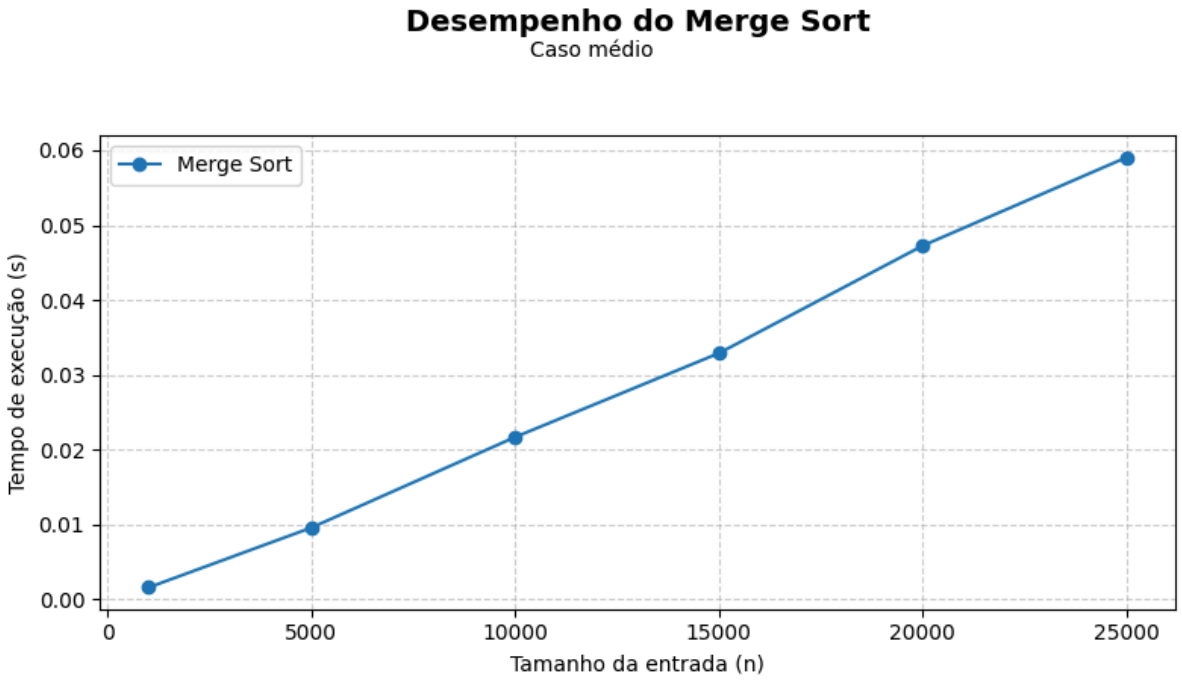


Figura 19: Caso médio Merge Sort.

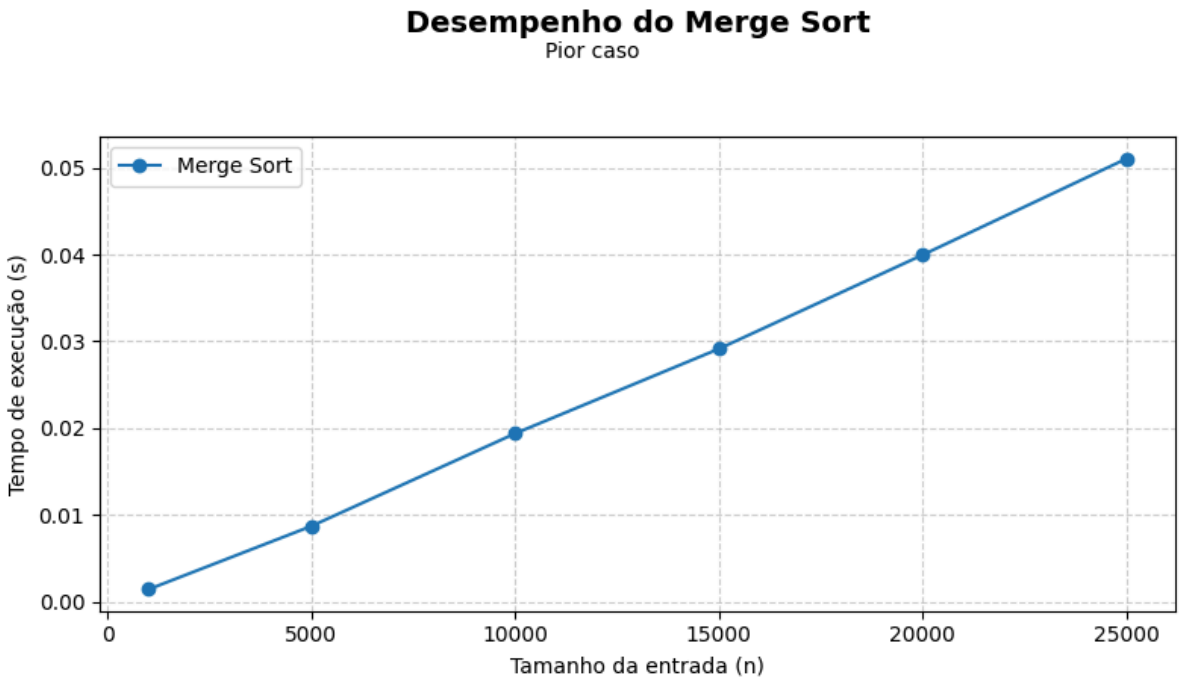


Figura 20: Pior caso Merge Sort.

3. CÓDIGOS

Abaixo segue os códigos utilizados para cada algoritmo:

- **Bubble-Sort:**

```python
def bubble_sort_original(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

- **Bubble-Sort Otimizado:**

```python
def bubble_sort_otimizado(arr):
    n = len(arr)
    for i in range(n - 1):
```

```
        trocou = False  # Flag para verificar se houve troca
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                trocou = True
        if not trocou:  # Nenhuma troca -> vetor já ordenado
            break
    return arr
```

- **Merge-Sort:**

Python

```
def merge_sort(vetor):
    if len(vetor) > 1:
        meio = len(vetor) // 2
        esquerda = vetor[:meio]
        direita = vetor[meio:]

        merge_sort(esquerda)
        merge_sort(direita)

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

- **Shell-Sort:**

```python
def shell_sort(vetor):
    n = len(vetor)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            temp = vetor[i]
            j = i
            while j >= gap and vetor[j - gap] > temp:
                vetor[j] = vetor[j - gap]
                j -= gap
            vetor[j] = temp
        gap //= 2
```

- **Quick-Sort (Pivô Central):**

```python
def quick_sort_central(vetor, inicio=0, fim=None):
    if fim is None:
        fim = len(vetor) - 1
    if inicio < fim:
        p = particiona_central(vetor, inicio, fim)
        quick_sort_central(vetor, inicio, p)
        quick_sort_central(vetor, p + 1, fim)

def particiona_central(vetor, inicio, fim):
    meio = (inicio + fim) // 2
    pivo = vetor[meio]
    i = inicio
    j = fim
    while True:
        while vetor[i] < pivo:
            i += 1
        while vetor[j] > pivo:
            j -= 1
        if i >= j:
            return j
        vetor[i], vetor[j] = vetor[j], vetor[i]
        i += 1
        j -= 1
```

- **Quick-Sort (Pivô Inicial):**

```python
def particiona(vetor, inicio, fim):
    pivo = vetor[inicio]
    i = inicio + 1
    j = fim

    while True:
        # Avança o i enquanto os elementos forem menores ou iguais ao pivô
        while i <= j and vetor[i] <= pivo:
            i += 1
        # Regride o j enquanto os elementos forem maiores que o pivô
        while i <= j and vetor[j] > pivo:
            j -= 1
        # Se os índices se cruzarem, encerra
        if i <= j:
            vetor[i], vetor[j] = vetor[j], vetor[i]
        else:
            break

    # Coloca o pivô em sua posição correta
    vetor[inicio], vetor[j] = vetor[j], vetor[inicio]
    return j  # Retorna o índice final do pivô


def quick_sort_primeiro(vetor, inicio=0, fim=None):
    if fim is None:
        fim = len(vetor) - 1
    if inicio < fim:
        p = particiona(vetor, inicio, fim)
        quick_sort_primeiro(vetor, inicio, p - 1)
        quick_sort_primeiro(vetor, p + 1, fim)
```

- **Selection-Sort:**

```python
def selection_sort(vetor):
    n = len(vetor)
    for i in range(n - 1):
```

```
        min_index = i
        for j in range(i + 1, n):
            if vetor[j] < vetor[min_index]:
                min_index = j
        vetor[i], vetor[min_index] = vetor[min_index], vetor[i]
```

- **Insertion-Sort:**

```
Python
def insertion_sort(vetor):
    for i in range(1, len(vetor)):
        chave = vetor[i]
        j = i - 1
        while j >= 0 and vetor[j] > chave:
            vetor[j + 1] = vetor[j]
            j -= 1
        vetor[j + 1] = chave
```

- **Heap-Sort:**

```
Python
def heapify(vetor, n, i):
    maior = i
    esq = 2 * i + 1
    dir = 2 * i + 2
    if esq < n and vetor[esq] > vetor[maior]:
        maior = esq
    if dir < n and vetor[dir] > vetor[maior]:
        maior = dir
    if maior != i:
        vetor[i], vetor[maior] = vetor[maior], vetor[i]
        heapify(vetor, n, maior)

def heap_sort(vetor):
    n = len(vetor)
    for i in range(n // 2 - 1, -1, -1):
        heapify(vetor, n, i)
```

```
    for i in range(n - 1, 0, -1):
        vetor[0], vetor[i] = vetor[i], vetor[0]
        heapify(vetor, i, 0)
```

4. CONCLUSÃO

A análise consolidada dos gráficos de desempenho, abrangendo o melhor caso, o pior caso e o caso médio, oferece uma visão clara sobre os trade-offs fundamentais entre os algoritmos estudados. Os dados demonstram que nenhuma solução é universalmente superior; a eficiência de um algoritmo está intrinsecamente ligada ao cenário de execução.

Para maior visualização e consolidação do estudo realizado, as figuras abaixo detalham a comparativa entre todos algoritmos estudados - abrangendo o melhor caso, caso médio e pior caso:

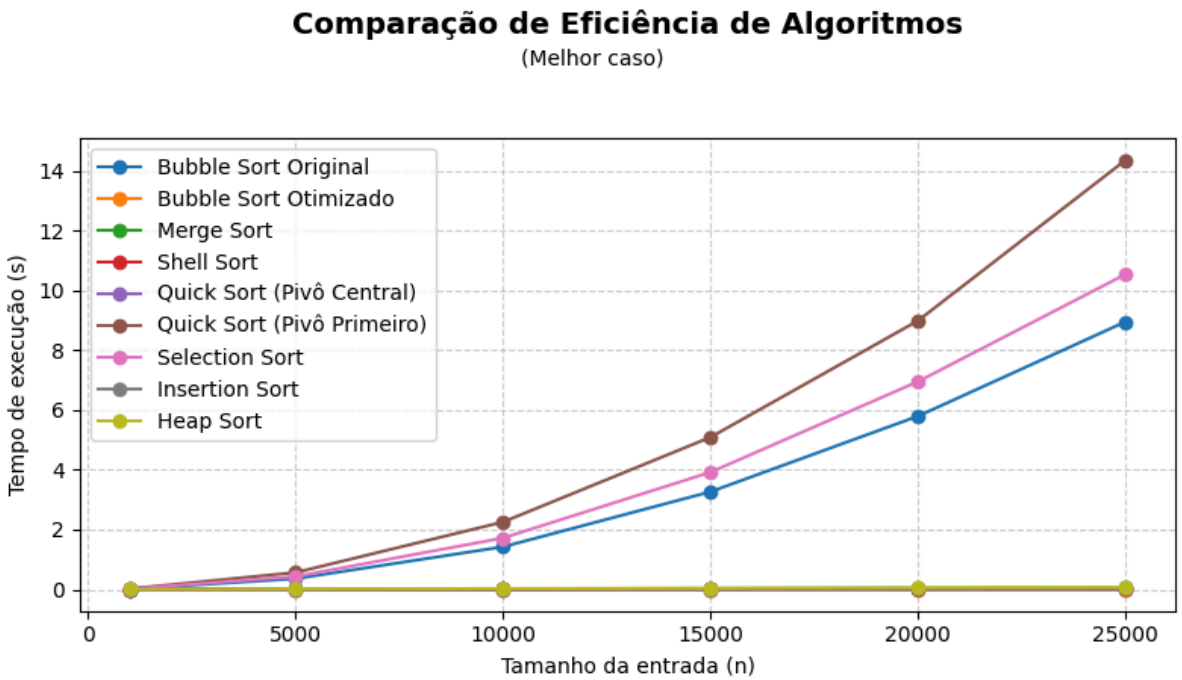


Figura 21: Comparativo algoritmos em melhor caso.

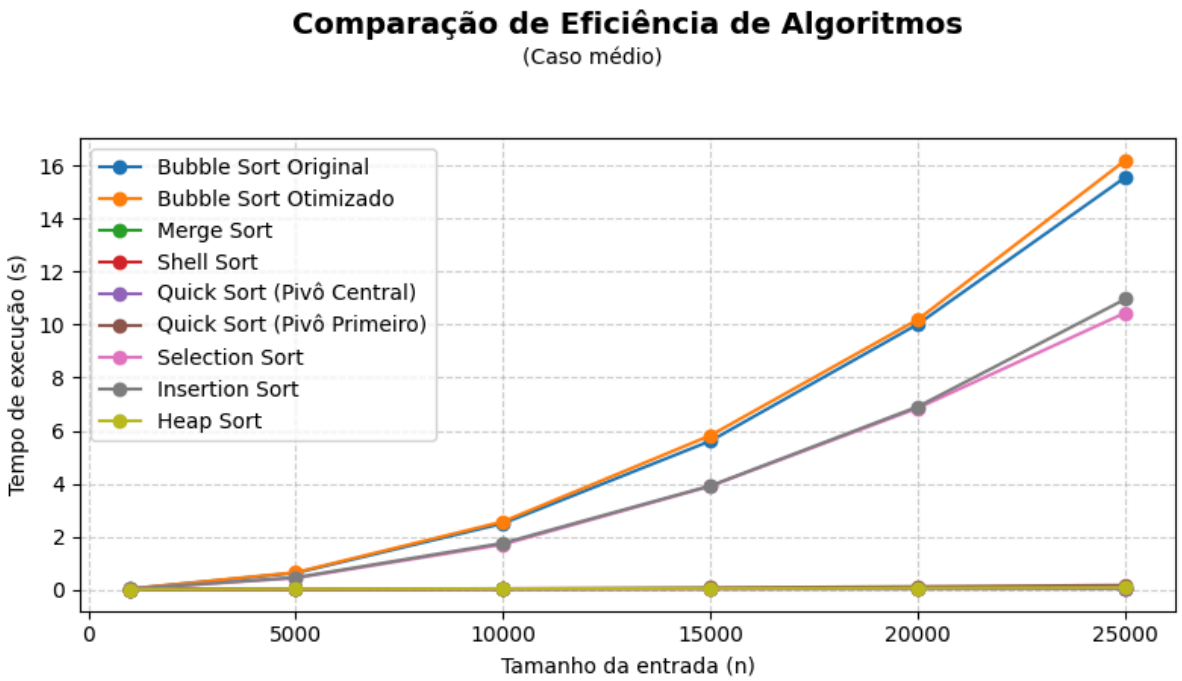


Figura 22: Comparativo algoritmos em caso médio.

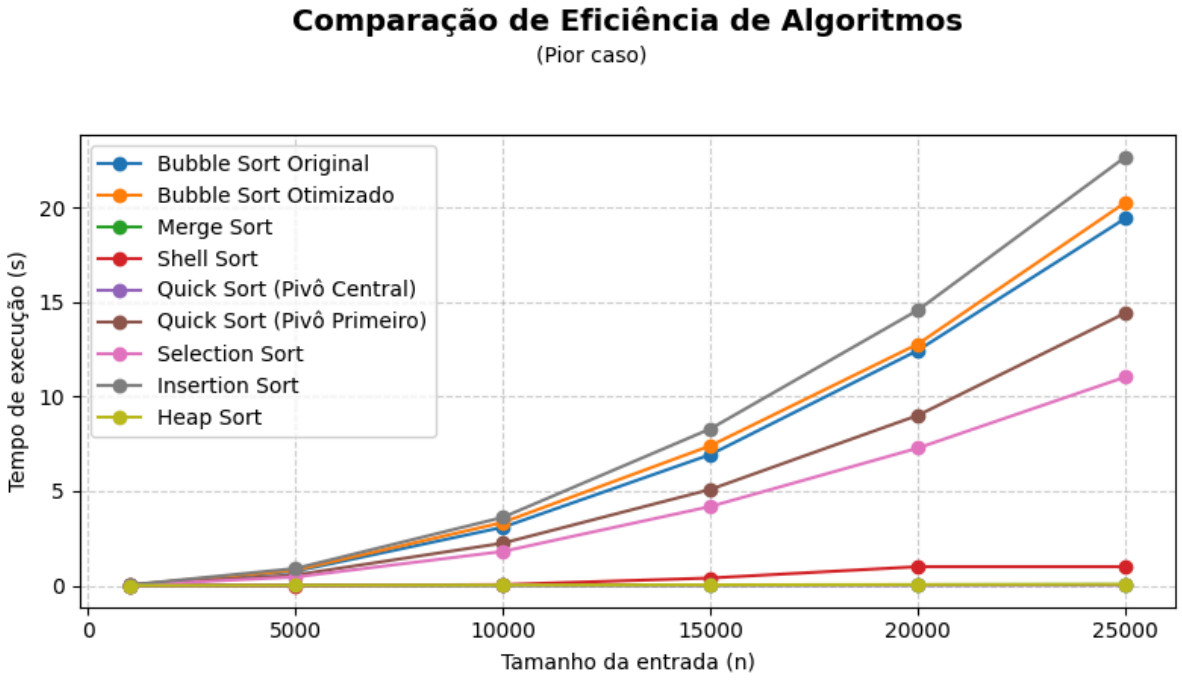


Figura 23: Comparativo algoritmos em pior caso.

De tal forma, a análise do caso médio revela quais algoritmos oferecem maior estabilidade e previsibilidade prática. Conclui-se, portanto, que a seleção de um algoritmo eficiente não deve ser baseada em apenas um cenário, mas sim em uma compreensão ponderada de como cada um reage a diferentes distribuições de dados de entrada, reforçando a importância da análise assintótica completa para o desenvolvimento de software robusto.