Análise da Complexidade de Algoritmos de Ordenação: Comparação entre Merge Sort, Quick Sort e Selection Sort

Adrielson F. Justino[1]

[1]Programa de Pós-Graduação em Engenharia de Computação
e Sistemas (PECS) – Universidade Estadual do Maranhão (UEMA)
São Luís, MA – Brazil

adrielferreira28@gmail.com

1. Introdução

A finalidade deste experimento é comparar o desempenho de três algoritmos de ordenação: *Merge Sort*, *Quick Sort* e *Selection Sort*, em termos de tempo de execução e uso de memória. A análise é realizada em diferentes cenários, incluindo o melhor caso, o pior caso e casos médios, para fornecer uma compreensão detalhada da eficiência e das limitações de cada algoritmo.

A complexidade de algoritmos é um conceito fundamental na ciência da computação, utilizado para descrever a eficiência de um algoritmo em termos de tempo e espaço [Papadimitriou 2014, Bournez et al. 2020]. A análise de complexidade busca determinar a quantidade de recursos computacionais necessários para executar um algoritmo, geralmente em função do tamanho da entrada. Os dois tipos principais de complexidade são a complexidade temporal, que mede o tempo de execução, e a complexidade espacial, que mede o uso de memória [Bournez et al. 2020].

Uma ferramenta amplamente utilizada para descrever a complexidade de algoritmos é a notação Big O [Rubinstein-Salzedo 2018]. A notação Big O fornece uma forma de expressar o comportamento assintótico de um algoritmo, ou seja, como o tempo de execução ou o uso de memória cresce com o aumento do tamanho da entrada [Mala and Ali 2022]. Por exemplo:

O(1): Se uma função f é $O(1)$, isso significa que o valor de f é constante e não cresce com o tamanho da entrada. Um exemplo é $f(x) = e^{-n}$, $n \in N$.

O(n): Se uma função f é $O(n)$, isso significa que f cresce linearmente com o tamanho da entrada. Percorrer uma lista é $O(n)$ porque cada item deve ser visitado.

O(n^2): Se uma função f é $O(n^2)$, isso significa que f cresce de forma quadrática com o tamanho da entrada.

O(2^n): Se uma função f é $O(2^n)$, isso significa que f cresce de forma exponencial com o tamanho da entrada.

O(log n): Se uma função f é $O(\log n)$, isso significa que f cresce de forma logarítmica com o tamanho da entrada. Um exemplo é o algoritmo de busca binária.

O(n log n): Se uma função f é $O(n \log n)$, isso significa que f cresce de forma linear multiplicada pelo logaritmo do tamanho da entrada.

Neste contexto, os algoritmos selecionados para este projeto foram *Merge Sort*, *Quick Sort* e *Selection Sort*, em que cada um possui diferentes complexidades. Dessa

forma, tem-se como objetivo proporcionar uma perspectiva das forças e fraquezas de cada algoritmo. Isso pode auxiliar para também na escolha do algoritmo mais adequado para diferentes tipos de aplicações com diferentes tamanhos de entrada.

Este documento encontra-se organizado conforme segue: Na Seção 2 é detalhado a implementação de cada um dos algoritmos, analisando suas complexidades teóricas. Os resultados experimentais das execuções em diferentes cenários são apresentadas na seção 3. Por fim, na Seção 4 são apresentadas as considerações finais.

2. Descrição do Experimento

A condução do projeto seguiu uma abordagem sistemática, dividida em três etapas principais, a saber: Escolha dos Algoritmos, Implementação, e Análise de Complexidade. Estas etapas são descritas detalhadamente a seguir.

2.1. Escolha dos Algoritmos

Os algoritmos de ordenação foram escolhidos com base em sua popularidade e características distintas de desempenho. Os três algoritmos escolhidos podem ser observados na Tabela 1.

Tabela 1. Complexidade dos Algoritmos de Ordenação

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

O primeiro algoritmo, *Merge Sort*, consiste em um algoritmo de ordenação por divisão e conquista, que divide a lista em sublistas menores, ordena essas sublistas e, em seguida, as combina para formar a lista ordenada final [Rabiu et al. 2021, Taiwo et al. 2020]. O *Merge Sort* é conhecido por sua complexidade $O(n \log n)$, consistente em todos os casos, conforme demonstrado na Tabela 1.

O segundo algoritmo, *Quick Sort* também é um algoritmo de ordenação por divisão e conquista, que realiza a selecção de um pivô e particiona a lista em sublistas de elementos menores e maiores que o pivô [Taiwo et al. 2020]. O *Quick Sort* é eficiente na prática, com complexidade média de $O(n \log n)$, mas pode ter um pior caso de $O(n^2)$ dependendo da escolha do pivô.

Por fim, o *Selection Sort* consiste em algoritmo de ordenação simples, que funciona selecionando repetidamente o menor elemento da lista não ordenada e trocando-o com o primeiro elemento não ordenado [Rabiu et al. 2021]. O *Selection Sort* é de fácil entendimento e implementação, mas tem uma complexidade de $O(n^2)$, o que o torna ineficiente para listas grandes.

2.2. Implementação

Para este estudo, foram utilizadas as seguintes tecnologias: *Python* (3.11.5)[1] para a implementação dos algoritmos, dada sua facilidade de uso e vasta gama de bi-

[1] `https://www.python.org/`

bliotecas disponíveis; as bibliotecas *time*[2] para medir o tempo de execução, *tracemalloc*[3] para medir o uso de memória, e *random*[4] para gerar os casos de teste; e a utilização de arquivos *Comma-separated values* (CSV) para registrar os resultados das medições de desempenho e para a geração de gráficos com a biblioteca *matplotlib*[5]. Além disso, o estudo e o aprimoramento dos algoritmos contaram com o auxílio do ChatGPT[6]. Todo código com suas respectivas funções e materiais adicionais estão disponíveis em material suplementar no seguinte repositório do *Github:* https://github.com/Adrielson/complexidade_algoritmos.git.

Desta forma, a implementação foi dividida em três etapas principais: leitura da entrada, geração dos casos de teste, e medição e registro dos resultados, conforme descrito a seguir.

2.2.1. Leitura do arquivo de entrada

Para este projeto a entrada consiste em um arquivo de texto, *input.txt*, contendo na primeira linha os tamanhos dos *arrays* que serão serem gerados, separados por um espaço em brano. Enquanto na segunda linha deve conter o nome do algoritmo escolhido. Para realizar a leitura do arquivo de texto contendo os tamanhos dos *arrays* e o algoritmo a ser executado, foi utilizada a função *read_input_file*, mostrada na Figura 1.

```python
def read_input_file(filename):
    with open(filename, 'r') as file:
        sizes = list(map(int, file.readline().strip().split()))
        algorithm = file.readline().strip()
    return sizes, algorithm
```

Figura 1. Função para leitura dos tamanhos dos *arrays* e algoritmo a ser testado

Conforme a Figura 1, essa função recebe como entrada um nome de arquivo e retorna os tamanhos dos *arrays* e o algoritmo a ser testado.

2.2.2. Geração dos Casos de Teste

Para cada algoritmo, foram gerados três tipos de casos de teste: melhor caso, pior caso e caso médio. A Figura 2 mostra as funções para geração dos casos de teste do *Merge Sort*, incluindo os casos ordenado, em ordem inversa e embaralhado.

Para o melhor caso do *Merge Sort*, Figura 2 (A), um *array* já ordenado é gerado utilizando a função *generate_best_case_merge_sort*. Esta função cria um *array* ordenado de 0 a $n - 1$, garantindo que o algoritmo execute com eficiência máxima, realizando o menor número de comparações possível.

É importante destacar que a análise do tempo de execução de algoritmos foca majoritariamente no pior caso, estabelecendo um limite superior que garante a máxima

[2] https://docs.python.org/3/library/time.html
[3] https://docs.python.org/3/library/tracemalloc.html
[4] https://docs.python.org/3/library/random.html
[5] https://matplotlib.org/
[6] https://www.openai.com/chatgpt

(A) Melhor caso

```python
def generate_best_case_merge_sort(n):
    return list(range(n))
```

(B) Pior caso

```python
def generate_worst_case_merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = generate_worst_case_merge_sort(arr[:mid])
    right = generate_worst_case_merge_sort(arr[mid:])
    worst_case_arr = []
    left_index, right_index = 0, 0
    while left_index < len(left) and right_index < len(right):
        worst_case_arr.append(left[left_index])
        left_index += 1
        worst_case_arr.append(right[right_index])
        right_index += 1
    worst_case_arr.extend(left[left_index:])
    worst_case_arr.extend(right[right_index:])
    return worst_case_arr
def generate_worst_case_merge_sort_wrapper(n):
    arr = list(range(n))
    return generate_worst_case_merge_sort(arr)
```

(C) Caso médio

```python
def generate_average_case_merge_sort(n):
    arr = list(range(n))
    random.shuffle(arr)
    return arr
```

Figura 2. Funções para geração dos casos de teste do Merge Sort

duração independente da entrada [Cormen et al. 2012]. O pior caso para o *Merge Sort* ocorre quando o *array* é estruturado de tal forma que, em cada etapa, a função *merge* realiza o número máximo de comparações. Para gerar esse *array*, foi criada uma função recursiva chamada *generate_worst_case_merge_sort*, que alterna os elementos entre os subarrays esquerdo e direito para maximizar o número de comparações. Esta função é encapsulada por um *wrapper* que inicializa o *array* e chama a função recursiva, conforme pode ser observado na Figura 2 (B).

Para o caso médio, um *array* embaralhado aleatoriamente é gerado pela função *generate_average_case_merge_sort*, Figura 2 (C). Esta função utiliza a biblioteca *random* para garantir que o *array* esteja em uma ordem aleatória, representando uma entrada típica para o *Merge Sort*.

A Figura 4 contém a descrição das funções utilizadas para gerar esses casos para o algoritmo *Quick Sort*.

Conforme ilustrado na Figura 3, foram implementadas as funções para geração dos casos de teste do *Quick Sort*, incluindo os casos ordenado, em ordem inversa e embaralhado.

Para o melhor caso do *Quick Sort*, Figura 3 (A), onde o pivô é o elemento do meio, o *array* deve ser tal que cada divisão é perfeitamente equilibrada. Na prática, isso é alcançado com um *array* já ordenado ou qualquer distribuição balanceada dos elementos, utilizando a função *generate_best_case_quick_sort*, que simplesmente gera um *array* ordenado de 0 a $n - 1$.

O pior caso para o *Quick Sort* ocorre quando o *array* está ordenado ou inversa-

(A) Melhor caso

```python
def generate_best_case_quick_sort(n):
    return list(range(n))
```

(B) Pior caso

```python
def generate_worst_case_quick_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = generate_worst_case_quick_sort(arr[:mid])
    right = generate_worst_case_quick_sort(arr[mid:])
    return [arr[mid]] + left + right

def generate_worst_case_quick_sort_wrapper(n):
    arr = list(range(n))
    return generate_worst_case_quick_sort(arr)
```

(C) Caso médio

```python
def generate_average_case_quick_sort(n):
    arr = list(range(n))
    random.shuffle(arr)
    return arr
```

Figura 3. Funções para geração dos casos de teste do Quick Sort

mente ordenado, e o pivô sendo o elemento do meio ainda resulta em uma partição desbalanceada. A definição de um caso ruim para dados de entrada pode ser, por exemplo, um que precisa de mais de 1,5 vezes o número médio de comparações [Hartmann 2015]. Para simular esse cenário, foi criada a função recursiva *generate_worst_case_quick_sort*, que gera um *array* especificamente distribuído para forçar o pior caso. Esta função é encapsulada por um *wrapper* que inicializa o *array* e chama a função recursiva, isso é demonstrado na Figura 3 (B).

Para o caso médio, um *array* embaralhado aleatoriamente é gerado pela função *generate_average_case_quick_sort*, Figura 3 (C). Esta função utiliza a biblioteca *random* para garantir que o *array* esteja em uma ordem aleatória, representando uma entrada típica para o *Quick Sort*.

Na Figura 4 são descritas as funções utilizadas para gerar esses casos para o algoritmo *Selection Sort*.

Conforme ilustrado na Figura 4, foram implementadas as funções para geração dos casos de teste do *Selection Sort*, incluindo os casos ordenado, em ordem inversa e embaralhado.

Para o melhor caso do *Selection Sort*, onde o *array* já está ordenado, o algoritmo deve realizar todas as comparações, mas não faz trocas desnecessárias, Figura 4 (A). Apesar disso, a complexidade permanece $O(n^2)$, mas o número de trocas é mínimo. A função *generate_best_case_selection_sort* gera um *array* ordenado de 0 a $n-1$.

O pior caso para o *Selection Sort* ocorre quando o *array* está em ordem inversa, forçando o algoritmo a realizar o máximo número de trocas, Figura 4 (B). A função *generate_worst_case_selection_sort* gera um *array* em ordem inversa, de $n-1$ a 0.

(A) Melhor caso

```python
def generate_best_case_selection_sort(n):
    return list(range(n))
```

(B) Pior caso

```python
def generate_worst_case_selection_sort(n):
    return list(range(n-1, -1, -1))
```

(C) Caso médio

```python
def generate_average_case_selection_sort(n):
    arr = list(range(n))
    random.shuffle(arr)
    return arr
```

Figura 4. Funções para geração dos casos de teste do Selection Sort

Para o caso médio, um *array* embaralhado aleatoriamente é gerado pela função *generate_average_case_selection_sort*, Figura 4 (C). Esta função utiliza a biblioteca *random* para garantir que o *array* esteja em uma ordem aleatória, representando uma entrada típica para o *Selection Sort*.

2.2.3. Medição e Registro dos Resultados

A medição do tempo de execução e do uso de memória dos algoritmos foi realizada utilizando as funções implementadas para cada tipo de caso. Os resultados foram registrados em um arquivo CSV para análise posterior. A função principal do experimento, *main()*, é responsável por ler os tamanhos dos *arrays* e o algoritmo a ser testado, gerar os casos de teste, medir o desempenho e registrar os resultados. A Figura 5 contém parte da implementação da função principal do experimento, a função completa e está disponível no material suplementar.

```python
def main():
    sizes, algorithm_name = read_input_file('input.txt')
    sort_function = get_sort_function(algorithm_name)
    log_file = 'sort_performance_log.csv'

    if not os.path.exists(log_file):
        with open(log_file, 'w', newline='') as csvfile:
            fieldnames = ['timestamp', 'algorithm', 'size', 'case', 'time', 'memory']
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
            writer.writeheader()

    case_generators = get_case_generators(algorithm_name)
    cases = ['melhor', 'pior', 'medio']

    for size in sizes:
        for case, gen_case in zip(cases, case_generators):
            if case == 'medio':
```

Figura 5. Função principal do experimento

A função *main()* é responsável por chamar as funções de leitura do arquivo de entrada com os tamanhos dos *arrays* e o algoritmo a ser testado, função para gerar os

casos de teste, função que mede o desempenho e por fim registra os resultados em um arquivo CSV.

2.3. Análise de Complexidade

A análise da complexidade dos algoritmos foi realizada para entender o comportamento assintótico de cada um. A complexidade de tempo e espaço foi determinada utilizando a notação Big O, que descreve o crescimento do tempo de execução ou do uso de memória em função do tamanho da entrada.

2.4. Merge Sort

A implementação do algoritmo *Merge Sort* pode ser observada na Figura 6. Para a análise do algoritmo *Merge Sort*, primeiro a divisão do *array* é feita calculando o ponto médio com a operação (linha 3), que tem complexidade $O(1)$. Em seguida, as metades esquerda e direita são criadas (linhas 4 e 5), cada uma com complexidade $O(n)$. As chamadas recursivas (linhas 6 e 7) têm complexidade $T(n/2)$ cada.

```
1  def merge_sort(arr):
2      if len(arr) > 1:  # O(1)
3          mid = len(arr) // 2  # O(1)
4          left_half = arr[:mid]  # O(n)
5          right_half = arr[mid:]  # O(n)
6          merge_sort(left_half)  # T(n/2)
7          merge_sort(right_half)  # T(n/2)
8          i, j, k = 0, 0, 0  # O(1)
9          while i < len(left_half) and j < len(right_half):  # O(n)
10             if left_half[i] < right_half[j]:  # O(1)
11                 arr[k] = left_half[i]  # O(1)
12                 i += 1  # O(1)
13             else:
14                 arr[k] = right_half[j]  # O(1)
15                 j += 1  # O(1)
16             k += 1  # O(1)
17         while i < len(left_half):  # O(n/2)
18             arr[k] = left_half[i]  # O(1)
19             i += 1  # O(1)
20             k += 1  # O(1)
21         while j < len(right_half):  # O(n/2)
22             arr[k] = right_half[j]  # O(1)
23             j += 1  # O(1)
24             k += 1  # O(1)
```

Figura 6. Código do Merge Sort

A combinação das sublistas começa com a inicialização dos índices i, j, k, que é $O(1)$ (linha 8). O laço $while$ percorre os elementos das sublistas (linha 9) com complexidade $O(n)$. As comparações e atribuições dentro do laço if (linha 10), $arr[k] = left\_half[i]$ (linha 11), $arr[k] = right\_half[j]$ (linha 14), são todas $O(1)$. Os laços $while$ (linhas 17 e 21) copiam os elementos restantes das metades esquerda e direita, cada um com complexidade $O(n/2)$.

Com isso, a equação de recorrência para o tempo de execução $T(n)$ é:

$$T(n) = 2T(n/2) + O(n)$$

Resolvendo essa equação, tem-se:

$$T(n) = O(n \log n)$$

Portanto, a complexidade de tempo do *Merge Sort* é $O(n \log n)$.

2.5. Quick Sort

A implementação do algoritmo *Quick Sort* é ilustrada na Figura 7.

```
1  def quick_sort(arr):
2      if len(arr) <= 1:  # O(1)
3          return arr  # O(1)
4      else:
5          pivot = arr[len(arr) // 2]  # O(1)
6          left = [x for x in arr if x < pivot]  # O(n)
7          middle = [x for x in arr if x == pivot]  # O(n)
8          right = [x for x in arr if x > pivot]  # O(n)
9          return quick_sort(left) + middle + quick_sort(right)  # T(n)
```

Figura 7. Código do Quick Sort

No algoritmo *quick sort* primeiramente uma condição base verifica se o *array* tem um ou nenhum elemento (linha 2), com complexidade $O(1)$, e retorna o *array* se ele já estiver ordenado (linha 3), também $O(1)$.

A escolha do pivô (linha 4) tem complexidade $O(1)$. Enquanto o particionamento do *array* em sublistas (linhas 5 a 7) tem complexidade $O(n)$ para cada operação. Já as chamadas recursivas e a combinação dos resultados (linha 8) têm complexidade $T(n)$.

Dessa forma, a equação de recorrência para o melhor e o caso médio é:

$$T(n) = 2T(n/2) + O(n)$$

Resolvendo essa equação, tem-se:

$$T(n) = O(n \log n)$$

No pior caso, o particionamento é desequilibrado:

$$T(n) = T(n-1) + O(n)$$

Resolvendo essa equação, é obtido:

$$T(n) = O(n^2)$$

Portanto, a complexidade de tempo do *quick sort* é $O(n \log n)$ para o caso médio e melhor caso. Já para pior caso a complexidade de tempo é $O(n^2)$.

2.6. Selection Sort

A implementação do algoritmo *Selection Sort* é demonstrada na Figura 8.

```
1  def selection_sort(arr):
2      for i in range(len(arr)):  # O(n)
3          min_idx = i  # O(1)
4          for j in range(i+1, len(arr)):  # O(n)
5              if arr[j] < arr[min_idx]:  # O(1)
6                  min_idx = j  # O(1)
7          arr[i], arr[min_idx] = arr[min_idx], arr[i]  # O(1)
```

Figura 8. Código do Selection Sort

Para a análise do *Selection Sort* inicialmente o laço externo itera sobre cada elemento do *array* (linha 2), com complexidade $O(n)$. A seleção do mínimo começa com a inicialização do índice do menor elemento (linha 3), $O(1)$. O laço interno itera sobre os elementos restantes do *array* (linha 4), $O(n)$. As comparações e atualizações (linhas 5 e 6) têm complexidade $O(1)$ cada. A troca de elementos (linha 7) tem complexidade $O(1)$.

Desta forma, a complexidade de tempo é calculada somando o número de comparações realizadas:

$$T(n) = \sum_{i=1}^{n}(n-i) = \frac{n(n-1)}{2} = O(n^2)$$

Portanto, a complexidade de tempo do *Selection Sort* é $O(n^2)$.

3. Resultados e Discussões

Nesta seção, são apresentados os resultados das execuções dos algoritmos de ordenação abordando o tempo de execução e o uso de memória para cada tamanho de vetor, casos (pior, médio e melhor) e algoritmo.

3.1. Merge Sort

Os resultados das execuções do algoritmo *Merge Sort*, em termos de tempo de execução e consumo de memória, estão sintetizados na Tabela 2.

Para análise complementar da Tabela 2, a Figura 9 contém a ilustração do tempo de execução do *Merge Sort* nos diferentes cenários. Observa-se que o tempo de execução aumenta linearmente com o tamanho do vetor, conforme esperado pela complexidade teórica $O(n \log n)$. Para vetores pequenos (100 elementos), o tempo de execução é negligível, enquanto para vetores maiores, como 10.000.000 de elementos, o tempo de execução varia de 968.7257 segundos no melhor caso a 1062.3636 segundos no caso médio, destacando a eficiência do *Merge Sort* em grandes volumes de dados.

Enquanto na Figura 10, pode ser observado o uso de memória do *Merge Sort*. Conforme pode ser visto, a memória utilizada pelo algoritmo também aumenta com o tamanho do vetor, refletindo a complexidade espacial $O(n)$. Para um vetor de 100 elementos, o consumo de memória é mínimo (0.0027 MB no melhor caso). Em contrapartida, para um vetor de 10.000.000 de elementos, o consumo de memória chega a 229.0258 MB. Este aumento é consistente com a necessidade do *Merge Sort* de utilizar espaço adicional para armazenar as sublistas durante o processo de ordenação.

Tabela 2. Tempo e Memória do Merge Sort

Timestamp	Algoritmo	Tamanho	Caso	Tempo (s)	Memória (MB)
2024-07-08 13:50:09	Merge Sort	100	Melhor	0.0	0.0027
2024-07-08 13:50:09	Merge Sort	100	Pior	0.0	0.0023
2024-07-08 13:50:09	Merge Sort	100	Médio	0.0005	0.0023
2024-07-08 13:50:09	Merge Sort	1000	Melhor	0.0	0.0229
2024-07-08 13:50:09	Merge Sort	1000	Pior	0.0080	0.0229
2024-07-08 13:50:09	Merge Sort	1000	Médio	0.0048	0.0229
2024-07-08 13:50:09	Merge Sort	10000	Melhor	0.3105	0.3530
2024-07-08 13:50:09	Merge Sort	10000	Pior	0.3001	0.3331
2024-07-08 13:50:11	Merge Sort	10000	Médio	0.2974	0.2780
2024-07-08 13:50:16	Merge Sort	100000	Melhor	4.8074	2.4312
2024-07-08 13:50:21	Merge Sort	100000	Pior	5.2079	2.4298
2024-07-08 13:50:47	Merge Sort	100000	Médio	5.1284	2.3740
2024-07-08 13:51:57	Merge Sort	1000000	Melhor	69.2669	23.0309
2024-07-08 13:53:15	Merge Sort	1000000	Pior	74.4777	23.0304
2024-07-08 14:00:04	Merge Sort	1000000	Médio	81.1062	23.0304
2024-07-08 14:16:14	Merge Sort	10000000	Melhor	968.7257	229.0258
2024-07-08 14:34:19	Merge Sort	10000000	Pior	1030.4405	229.0258
2024-07-08 16:03:44	Merge Sort	10000000	Médio	1062.3636	229.0257

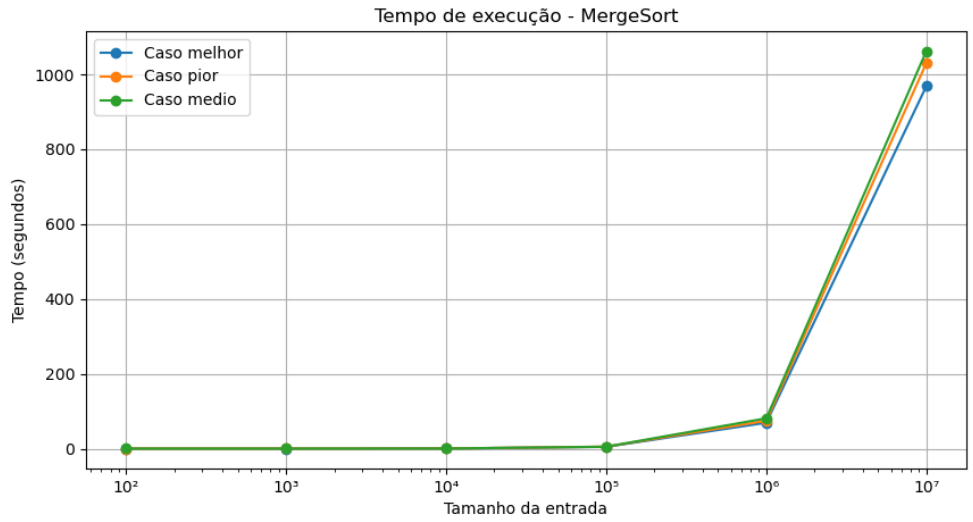

Figura 9. Tempo de execução do Merge Sort

3.2. Quick Sort

Nesta seção são apresentados os resultados obtidos a partir da execução do algoritmo *Quick Sort*. O resumo das execuções está contido na Tabela 3, onde também são considerados os tempos de execução e os consumos de memória para diferentes tamanhos de vetores e tipos de casos (melhor, pior e médio).

Analisando a Tabela 3 que contém o tempo de execução do *Quick Sort* em diferentes cenários e observando a Figura 11, pode ser notado que o tempo de execução aumenta

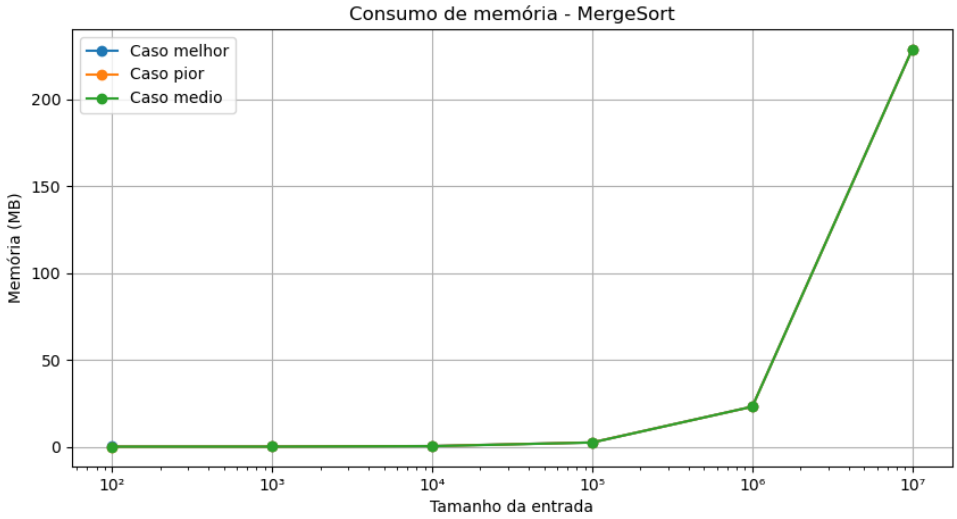

Figura 10. Uso de memória do Merge Sort

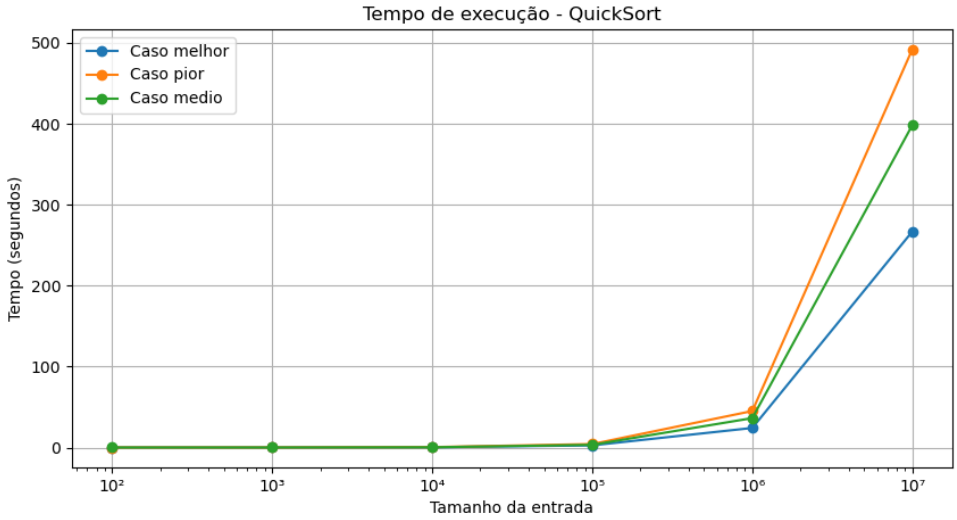

Figura 11. Tempo de execução do Quick Sort

de forma significativa com o tamanho do vetor, especialmente no pior caso.

No caso dos vetores pequenos (100 elementos), o tempo de execução é praticamente instantâneo. Para vetores maiores, como os de 10.000.000 de elementos, o tempo de execução varia de 267.0118 segundos no melhor caso a 492.1881 segundos no pior caso. Esta variação significativa entre os diferentes casos de teste é um reflexo da natureza do *Quick Sort*, cuja eficiência depende fortemente da escolha do pivô e da ordem inicial dos elementos. Ressalta-se que a geração do vetor para o pior caso foi implementada de maneira a maximizar a quantidade de partições desbalanceadas, em que o pivô escolhido fosse sempre o pior possível, resultando em um número máximo de comparações e trocas, levando ao pior desempenho possível do *Quick Sort*.

Tabela 3. Tempo e Memória do Quick Sort

Timestamp	Algoritmo	Tamanho	Caso	Tempo (s)	Memória (MB)
2024-07-08 18:04:35	Quick Sort	100	Melhor	0.0	0.0037
2024-07-08 18:04:35	Quick Sort	100	Pior	0.0	0.0072
2024-07-08 18:04:35	Quick Sort	100	Médio	0.0016	0.0052
2024-07-08 18:04:35	Quick Sort	1000	Melhor	0.0179	0.0320
2024-07-08 18:04:35	Quick Sort	1000	Pior	0.0448	0.0649
2024-07-08 18:04:35	Quick Sort	1000	Médio	0.0240	0.0448
2024-07-08 18:04:36	Quick Sort	10000	Melhor	0.2600	0.3148
2024-07-08 18:04:36	Quick Sort	10000	Pior	0.4386	0.7740
2024-07-08 18:04:38	Quick Sort	10000	Médio	0.2894	0.5519
2024-07-08 18:04:40	Quick Sort	100000	Melhor	2.6368	3.3403
2024-07-08 18:04:45	Quick Sort	100000	Pior	4.2409	6.4107
2024-07-08 18:05:02	Quick Sort	100000	Médio	3.3077	4.4362
2024-07-08 18:05:26	Quick Sort	1000000	Melhor	24.1476	31.3216
2024-07-08 18:06:14	Quick Sort	1000000	Pior	45.0735	62.7396
2024-07-08 18:09:23	Quick Sort	1000000	Médio	36.3079	41.7871
2024-07-08 18:13:51	Quick Sort	10000000	Melhor	267.0118	318.2954
2024-07-08 18:22:30	Quick Sort	10000000	Pior	492.1881	641.1869
2024-07-08 18:57:21	Quick Sort	10000000	Médio	398.4535	478.6161

Quanto ao consumo de memória, a Tabela 3 e a Figura 12 mostram que o uso de memória pelo *Quick Sort* aumenta com o tamanho do vetor, mas de forma mais dramática no pior caso.

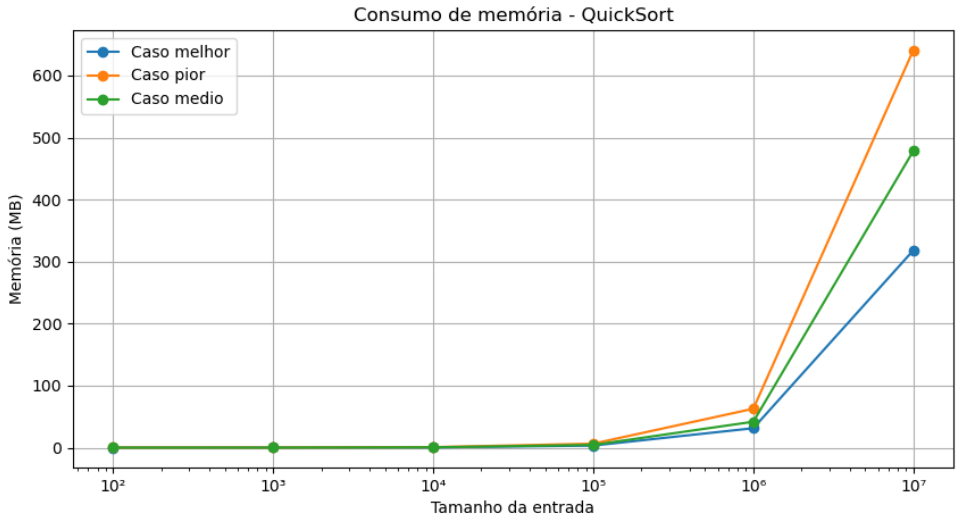

Figura 12. Uso de memória do Quick Sort

Para um vetor de 100 elementos, o consumo de memória é relativamente baixo (0.0037 MB no melhor caso). No entanto, para um vetor de 10.000.000 de elementos, o consumo de memória no pior caso pode chegar a 641.1869 MB, destacando a potencial ineficiência do *Quick Sort* em termos de uso de espaço quando comparado ao *Merge Sort*.

3.3. Selection Sort

Os resultados obtidos a partir das execuções do algoritmo *Selection Sort* são demonstrados a seguir. É importante ressaltar que a execução dos testes foi realizada parcialmente, pois não foi possível processar entradas superiores a 100.000 de elementos dentro do prazo estipulado para a tarefa.

Conforme a Tabela 4, os resultados indicam que o algoritmo apresenta um desempenho significativamente inferior em termos de tempo de execução quando comparado aos outros algoritmos analisados, especialmente para entradas maiores.

Tabela 4. Tempo e Memória do Selection Sort

Timestamp	Algoritmo	Tamanho	Caso	Tempo (s)	Memória (MB)
2024-07-08 19:12:40	Selection Sort	100	melhor	0.0	0.0009
2024-07-08 19:12:40	Selection Sort	100	pior	0.0010	0.0009
2024-07-08 19:12:40	Selection Sort	100	médio	0.0009	0.0009
2024-07-08 19:12:40	Selection Sort	1000	melhor	0.9739	0.1502
2024-07-08 19:12:42	Selection Sort	1000	pior	1.1161	0.1507
2024-07-08 19:12:43	Selection Sort	1000	médio	1.0591	0.1505
2024-07-08 19:14:33	Selection Sort	10000	melhor	110.6916	0.2187
2024-07-08 19:16:25	Selection Sort	10000	pior	111.5872	0.2187
2024-07-08 19:18:16	Selection Sort	10000	médio	111.3172	0.2183
2024-07-08 20:55:16	Selection Sort	100000	pior	5819.6914	0.9054
2024-07-08 22:38:34	Selection Sort	100000	melhor	6198.2111	0.9053
2024-07-09 00:16:56	Selection Sort	100000	médio	5901.6086	0.9049

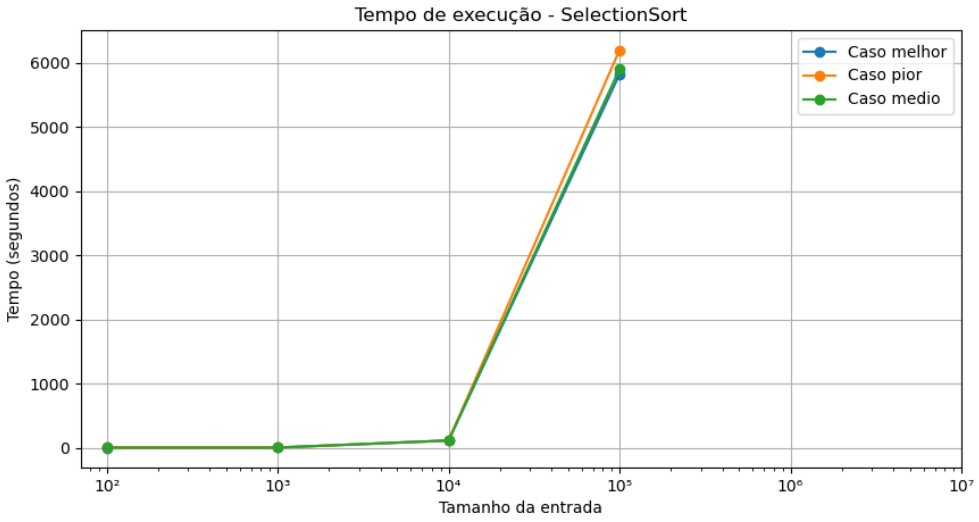

Figura 13. Tempo de execução do Selection Sort

A Figura 13 apresenta o tempo de execução do Selection Sort para diferentes tamanhos de vetores e tipos de casos. Observa-se que o tempo de execução cresce de forma exponencial com o aumento do tamanho do vetor, confirmando a complexidade teórica $O(n^2)$ do algoritmo.

Para vetores pequenos (100 elementos), o tempo de execução é quase instantâneo. No entanto, para vetores maiores, como os de 100.000 elementos, o tempo de execução chega a mais de 6.000 segundos (aproximadamente 1 hora e 40 minutos) no melhor caso, destacando a ineficiência do Selection Sort para grandes volumes de dados. Devido ao prazo da tarefa, não foi possível executar o algoritmo para vetores de 10.000.000 elementos.

O consumo de memória, conforme exposto na Tabela 4, é ilustrado na Figura 14. Observa-se que o consumo de memória do Selection Sort é relativamente constante e baixo, mesmo para vetores maiores, devido à sua complexidade espacial $O(1)$. Também é possível observar que o consumo de memória do Selection Sort permanece baixo (aproximadamente 0.9 MB para vetores de 100.000 elementos), independentemente do tamanho do vetor ou do tipo de caso.

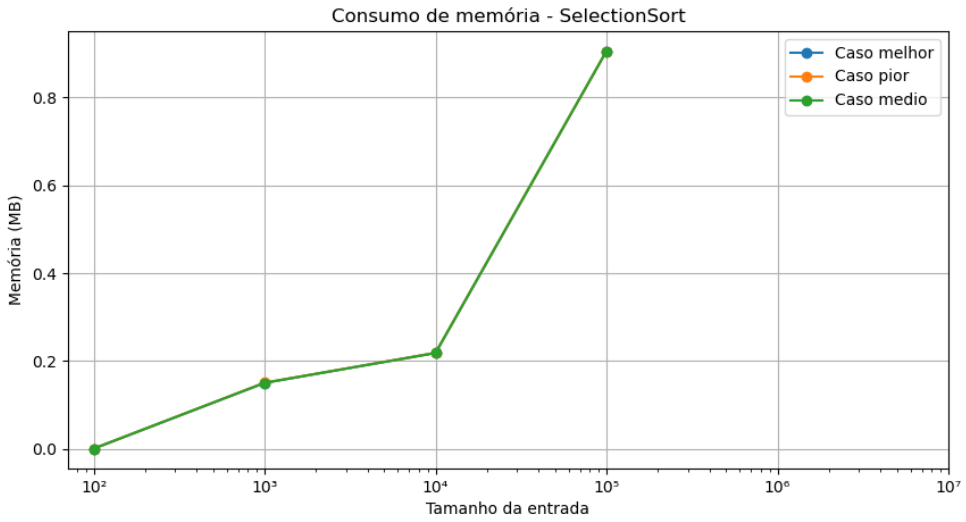

Figura 14. Uso de memória do Selection Sort

3.4. Comparação de Desempenho

Nesta seção, é realizada a comparação geral entre os três algoritmos analisados para determinar o mais adequado para diferentes tipos de aplicações. A Tabela 5 resume os resultados de tempo de execução e consumo de memória para os três algoritmos analisados: *Merge Sort, Quick Sort* e *Selection Sort*.

O *Merge Sort* mostrou-se consistentemente eficiente em termos de tempo de execução, com variações mínimas entre os diferentes cenários de teste (melhor, pior e médio). O *Quick Sort*, embora mais rápido no melhor e médio caso, apresentou um desempenho significativamente pior no pior caso. Isso ocorre devido à natureza do algoritmo, que pode levar a uma degradação de $O(n^2)$ se os pivôs escolhidos forem os piores possíveis repetidamente. O *Selection Sort*, por outro lado, demonstrou-se ineficaz para grandes volumes de dados, com tempos de execução extremamente longos.

Em termos de consumo de memória, o *Selection Sort* se destacou por sua baixa utilização de espaço, mantendo um consumo constante e mínimo. O *Merge Sort*, apesar de

Tabela 5. Comparação de Desempenho dos Algoritmos de Ordenação

Algoritmo	Tamanho	Caso	Tempo (s)	Memória (MB)
Merge Sort	1000000	Melhor	69.2669	23.0309
Merge Sort	1000000	Pior	74.4777	23.0304
Merge Sort	1000000	Médio	81.1062	23.0304
Quick Sort	1000000	Melhor	24.1476	31.3216
Quick Sort	1000000	Pior	45.0735	62.7396
Quick Sort	1000000	Médio	36.3079	41.7871
Selection Sort	100000	Melhor	6198.2111	0.9053
Selection Sort	100000	Pior	5819.6914	0.9054
Selection Sort	100000	Médio	5901.6086	0.9049

eficiente em termos de tempo, apresentou um consumo de memória linear com o tamanho do vetor, o que pode ser uma limitação em sistemas com recursos de memória restritos. O *Quick Sort*, embora geralmente mais eficiente em termos de tempo, pode apresentar um consumo de memória elevado no pior caso.

Com base nos resultados obtidos, pode-se sugerir que o *Merge Sort* pode funcionar melhor para aplicações onde a previsibilidade e a estabilidade do tempo de execução são críticas, especialmente em ambientes onde o consumo de memória não é um fator limitante. Enquanto *Quick Sort* pode se adequar melhor para situações onde a entrada está próxima da aleatoriedade ou onde podem ser adotadas estratégias para garantir uma boa escolha de pivô. No entanto, deve-se ter cautela com entradas quase ordenadas ou inversamente ordenadas. Por fim, o *Selection Sort*, devido à sua simplicidade e baixo consumo de memória, pode ser útil para conjuntos de dados relativamente pequenos, mas não é recomendado para grandes volumes de dados devido à sua ineficiência em termos de tempo de execução.

4. Conclusão

Este estudo comparou três algoritmos de ordenação *(Merge Sort, Quick Sort e Selection Sort)* em termos de tempo de execução e consumo de memória em diferentes cenários (melhor caso, pior caso e caso médio). Os resultados destacaram as vantagens e limitações de cada algoritmo, proporcionando uma base sólida para a escolha do algoritmo mais adequado para diferentes tipos de aplicações. O *Merge Sort* e o *Quick Sort* demonstraram-se mais eficientes para grandes conjuntos de dados, enquanto o *Selection Sort* é mais adequado para conjuntos de dados menores onde a simplicidade e o baixo consumo de memória são desejáveis.

Referências

Bournez, O., Dowek, G., Gilleron, R., Grigorieff, S., Marion, J., Perdrix, S., and Tison, S. (2020). Theoretical computer science: Computational complexity. *A Guided Tour of Artificial Intelligence Research*.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2012). Algoritmos-teoria e prática (3a. ediçao). *Editora Campus*.

Hartmann, G. (2015). A numerical analysis of quicksort: How many cases are bad cases? *arXiv preprint arXiv:1507.04220*.

Mala, F. A. and Ali, R. (2022). The big-o of mathematics and computer science. *Journal of Applied Mathematics and Computation*.

Papadimitriou, C. (2014). Algorithms, complexity, and the sciences. *Proceedings of the National Academy of Sciences*, 111:15881 – 15887.

Rabiu, A., Garba, E., Baha, B., and Mukhtar, M. (2021). Comparative analysis between selection sort and merge sort algorithms. *Nigerian Journal of Basic and Applied Sciences*, 29(1):43–48.

Rubinstein-Salzedo, S. (2018). Big o notation and algorithm efficiency. In *Cryptography*, pages 75–83. Springer.

Taiwo, O. E., Christianah, A. O., Oluwatobi, A. N., Aderonke, K. A., et al. (2020). Comparative study of two divide and conquer sorting algorithms: quicksort and mergesort. *Procedia Computer Science*, 171:2532–2540.