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

Emanuel Catão Montenegro
Lucas José de Sousa Gomes

ALGORITMOS DE ORDENAÇÃO: UMA ANÁLISE EMPÍRICA

PALMAS (TO)
2024

Emanuel Catão Montenegro

Lucas José de Sousa Gomes

Algoritmos de Ordenação: uma Análise Empírica

Trabalho de Ordenação de Algoritmos apresentado à Universidade Federal do Tocantins na disciplina de Projeto e Análise de Algoritmos .

Professor: Dr. Warley Gramacho

PALMAS (TO)

2024

RESUMO

Este trabalho realiza uma análise empírica de seis algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. A implementação dos algoritmos foi feita em Python e os testes foram conduzidos em listas de diferentes tamanhos (1000, 10000, 50000 e 100000 elementos) e distribuições (aleatória, ordenada e inversamente ordenada). Foram analisadas métricas como o número de comparações, trocas e tempo de execução. Os resultados mostram que Merge Sort e Quick Sort são mais eficientes em termos de tempo de execução e número de comparações para listas grandes e desordenadas. Algoritmos mais simples, como Bubble Sort e Selection Sort, mostraram desempenho significativamente inferior, especialmente para listas grandes. Considerações sobre a facilidade de implementação, uso de memória e estabilidade dos algoritmos também foram discutidas. As recomendações sobre o uso apropriado de cada algoritmo em diferentes situações foram apresentadas com base nos resultados obtidos.

Palavra-chave: Algoritmos de Ordenação. Análise de Desempenho. Comparação de Algoritmos. Python. Complexidade Computacional.

ABSTRACT

This study provides an empirical analysis of six sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort. The algorithms were implemented in Python and tested on lists of varying sizes (1000, 10000, 50000, and 100000 elements) and distributions (random, sorted, and reverse-sorted). Key metrics such as the number of comparisons, swaps, and execution time were evaluated. The results indicate that Merge Sort and Quick Sort are more efficient in terms of execution time and number of comparisons for large, unsorted lists. Simpler algorithms like Bubble Sort and Selection Sort exhibited significantly lower performance, especially for large lists. Considerations regarding the ease of implementation, memory usage, and algorithm stability were also discussed. Recommendations for the appropriate use of each algorithm in different scenarios were presented based on the results.

Keywords: Sorting Algorithms. Performance Analysis. Algorithm Comparison. Python. Computational Complexity.

LISTA DE TABELAS

Tabela 1 – Comparação de Algoritmos de Ordenação para Listas Aleatórias de Tamanho 1000. 19
Tabela 2 – Comparação de Algoritmos de Ordenação para Listas Aleatórias de Tamanho 10000. 19
Tabela 3 – Comparação de Algoritmos de Ordenação para Listas Aleatórias de Tamanho 50000. 20
Tabela 4 – Comparação de Algoritmos de Ordenação para Listas Aleatórias de Tamanho 100000. 20
Tabela 5 – Comparação de Algoritmos de Ordenação para Listas Ordenadas de Tamanho 1000. 20
Tabela 6 – Comparação de Algoritmos de Ordenação para Listas Ordenadas de Tamanho 10000. 21
Tabela 7 – Comparação de Algoritmos de Ordenação para Listas Ordenadas de Tamanho 50000. 21
Tabela 8 – Comparação de Algoritmos de Ordenação para Listas Ordenadas de Tamanho 100000. 21
Tabela 9 – Comparação de Algoritmos de Ordenação para Listas Revertidas de Tamanho 1000. 22
Tabela 10 – Comparação de Algoritmos de Ordenação para Listas Revertidas de Tamanho 10000. 22
Tabela 11 – Comparação de Algoritmos de Ordenação para Listas Revertidas de Tamanho 50000. 22
Tabela 12 – Comparação de Algoritmos de Ordenação para Listas Revertidas de Tamanho 100000. 23

SUMÁRIO

1 INTRODUÇÃO 7

2 REVISÃO TEÓRICA 8

2.1 Bubble Sort 8

2.2 Insertion Sort 8

2.3 Selection Sort 9

2.4 Merge Sort 10

2.5 Quick Sort 11

2.6 Heap Sort 12

3 METODOLOGIA 13

3.1 Experimento 13

3.2 Implementação 13

4 RESULTADOS 19

4.1 Listas de Ordenação Aleatória 19

4.2 Listas Ordenadas 20

4.3 Listas Inversamente Ordenadas 22

5 DISCUSSÃO 24

5.1 Considerações sobre a Facilidade de Implementação, Uso de Memória e Estabilidade dos Algoritmos 24

6 CONCLUSÃO 26

6.1 Recomendações sobre o Uso de Cada Algoritmo em Diferentes Situações 26

REFERÊNCIAS 28

1 INTRODUÇÃO

A ordenação de dados é uma operação fundamental em ciência da computação e possui vasta aplicação em diversas áreas (KNUTH, 1998), desde a organização de grandes volumes de dados até a otimização de algoritmos complexos. A ordenação é tida ainda como o problema mais fundamental no estudo dos algoritmos, seja porque em diversos cenários a necessidade de ordenar informações é inerente a uma aplicação, seja porque é um problema para o qual podemos demonstrar um limite inferior não trivial. Algoritmos de ordenação são utilizados para organizar coleções de dados, em geral em forma de coleção, em uma sequência específica, em ordem crescente ou decrescente na maior parte das situações, facilitando assim a busca, a análise, a visualização e até a utilização dessas informações (CORMEN et al., 2009).

Este trabalho tem como objetivo analisar e comparar a eficiência de diferentes algoritmos de ordenação, destacando suas características principais, vantagens e desvantagens. Serão analisados os seguintes algoritmos: **Bubble Sort**, **Selection Sort**, **Insertion Sort**, **Merge Sort**, **Quick Sort** e **Heap Sort**.

Cada um desses algoritmos será descrito teoricamente, com ênfase em sua complexidade computacional, se é realizado in place e sua estabilidade. Em seguida, será apresentado um ambiente de teste detalhado, incluindo hardware e software utilizados, seguido pela implementação dos algoritmos e os procedimentos adotados para medir o tempo de execução, número de comparações e trocas realizadas.

Os resultados serão apresentados em forma de tabelas comparativas, permitindo uma visualização clara do desempenho de cada algoritmo em diferentes situações. A discussão dos resultados será realizada com base nas expectativas teóricas, considerando aspectos como facilidade de implementação, uso de memória e estabilidade dos algoritmos. Por fim, serão abordadas as limitações deste estudo e sugestões para trabalhos futuros.

2 REVISÃO TEÓRICA

Essa seção se dedicará a uma rápida descrição teórica de cada algoritmo, nos detendo em dissertar sobre a complexidade, a natureza da ordenação, sua estabilidade e se são in-place. Os algoritmos abordados são Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort e Heap Sort.

2.1 Bubble Sort

O Bubble Sort é um dos algoritmos de ordenação mais simples e conhecidos. Ele funciona repetidamente percorrendo a lista, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. Este processo é repetido até que a lista esteja completamente ordenada. A simplicidade do Bubble Sort o torna fácil de entender e implementar, embora sua ineficiência em listas grandes limite seu uso prático.

A complexidade do Bubble Sort no pior e no caso médio é $O(n^2)$. Isso ocorre porque, em cada passagem, cada elemento deve ser comparado com todos os outros elementos restantes, resultando em $n(n-1)/2$ comparações. No melhor caso, quando a lista já está ordenada, a complexidade é $O(n)$, já que o algoritmo pode detectar que a lista está ordenada e terminar mais cedo.

O Bubble Sort é considerado um algoritmo in-place porque ele requer apenas uma pequena quantidade de memória adicional constante para realizar as trocas. Além disso, é um algoritmo estável, o que significa que elementos iguais mantêm sua ordem relativa original após a ordenação. Esta estabilidade é garantida porque o Bubble Sort apenas troca elementos quando eles estão na ordem errada.

```
1 #Pseudo-codigo representando o algoritmo
2
3 BubbleSort(A)
4     para i de 1 a tamanho(A) - 1
5         para j de 0 a tamanho(A) - i - 1
6             se A[j] > A[j + 1]
7             trocar A[j] e A[j + 1]
```

2.2 Insertion Sort

O Insertion Sort é outro algoritmo de ordenação simples que constrói a lista ordenada um elemento de cada vez, inserindo cada novo elemento na posição correta em relação aos elementos já ordenados. Este processo de inserção é similar à forma como as pessoas organizam cartas em um baralho.

A complexidade do Insertion Sort no pior e no caso médio é $O(n^2)$. No pior cenário, cada novo elemento deve ser comparado com todos os elementos já ordenados, resultando em uma complexidade quadrática. No melhor caso, quando a lista já está ordenada, a complexidade é $O(n)$, uma vez que cada elemento só precisa ser comparado uma vez.

O Insertion Sort é um algoritmo in-place, pois requer apenas uma pequena quantidade de memória adicional constante. Este algoritmo é estável, preservando a ordem relativa dos elementos iguais. A eficiência do Insertion Sort em listas pequenas e quase ordenadas faz com que ele seja útil em algumas situações práticas, especialmente como parte de algoritmos híbridos.

```
#Pseudo-codigo representando o algoritmo

InsertionSort(A)
  para i de 1 a tamanho(A) - 1
    chave = A[i]
    j = i - 1
    enquanto j >= 0 e A[j] > chave
      A[j + 1] = A[j]
      j = j - 1
    A[j + 1] = chave
```

2.3 Selection Sort

O Selection Sort funciona de maneira diferente, dividindo a lista em duas partes: a sublista dos itens já ordenados e a sublista dos itens restantes a serem ordenados. O algoritmo encontra repetidamente o menor (ou maior) elemento da sublista não ordenada e o move para o final da sublista ordenada. Esta abordagem garante que a lista é ordenada passo a passo, um elemento de cada vez.

A complexidade do Selection Sort é $O(n^2)$ em todos os casos, pois independentemente da ordem inicial, sempre é necessário comparar cada elemento com todos os outros elementos restantes. Esse comportamento torna o Selection Sort menos eficiente que alguns outros algoritmos de ordenação para listas grandes.

O Selection Sort é um algoritmo in-place, pois requer apenas uma pequena quantidade de memória adicional constante. No entanto, ele não é estável, o que significa que a ordem relativa dos elementos iguais pode ser alterada. Isso ocorre porque o algoritmo troca o menor elemento encontrado com o primeiro elemento da sublista não ordenada, o que pode mudar a posição relativa dos elementos iguais.

```
#Pseudo-codigo representando o algoritmo

SelectionSort(A)
```

```
4       para i de 0 a tamanho(A) - 1
5           min_idx = i
6           para j de i + 1 a tamanho(A)
7           se A[j] < A[min_idx]
8               min_idx = j
9           trocar A[i] e A[min_idx]
```

2.4 Merge Sort

O Merge Sort é um algoritmo de ordenação eficiente que segue o paradigma de divisão e conquista. Ele divide repetidamente a lista pela metade, ordena cada metade de forma recursiva e depois mescla as duas metades ordenadas. Este processo de divisão e mesclagem garante uma ordenação eficiente mesmo para listas grandes.

A complexidade do Merge Sort é $O(n \log n)$ em todos os casos. Isso ocorre porque a lista é repetidamente dividida ao meio, resultando em $\log n$ divisões, e cada divisão requer a ordenação de todos os elementos, resultando em n comparações. Essa complexidade torna o Merge Sort muito eficiente para grandes volumes de dados.

O Merge Sort não é um algoritmo in-place, pois requer memória adicional proporcional ao tamanho da lista para armazenar as sublistas temporárias. No entanto, ele é um algoritmo estável, mantendo a ordem relativa dos elementos iguais após a ordenação. A eficiência e a estabilidade do Merge Sort o tornam amplamente utilizado em sistemas que lidam com grandes volumes de dados. Tende a ser bastante útil em situações que a estabilidade é muito importante, a exemplo de ordenações por múltiplos critérios: ordenar lista de contatos por nome e sobrenome; ordernar transações bancárisa por valores, mantendo a ordenação de datas, entre outros.

```
1 #Pseudo-codigo representando o algoritmo
2
3 MergeSort(A)
4     se tamanho(A) <= 1
5         retornar A
6     meio = tamanho(A) // 2
7     esquerda = MergeSort(sublista(A, 0, meio))
8     direita = MergeSort(sublista(A, meio, tamanho(A)))
9     retornar merge(esquerda, direita)
10
11 merge(esquerda, direita)
12     resultado = lista_vazia()
13     enquanto esquerda nao esta vazia e direita nao esta vazia
14         se primeiro(esquerda) <= primeiro(direita)
15             adicionar primeiro(esquerda) a resultado
16             remover primeiro de esquerda
17         senao
```

```
            adicionar primeiro(direita) a resultado
            remover primeiro de direita
    adicionar todos os elementos restantes de esquerda a resultado
    adicionar todos os elementos restantes de direita a resultado
    retornar resultado
```

2.5 Quick Sort

O Quick Sort é um algoritmo de ordenação eficiente que também segue o paradigma de divisão e conquista. Ele seleciona um pivô, particiona a lista em elementos menores e maiores que o pivô, e depois ordena as sublistas de forma recursiva. A escolha do pivô é crucial para a eficiência do Quick Sort. (HOARE, 1962)

A complexidade do Quick Sort no melhor e no caso médio é $O(n \log n)$. Isso ocorre quando o pivô divide a lista em partes aproximadamente iguais. No pior caso, a complexidade é $O(n^2)$, ocorrendo quando a lista já está quase ordenada e o pivô escolhido é o menor ou o maior elemento. No entanto, o uso de técnicas de escolha de pivô, como o pivô mediano, pode mitigar este problema.

O Quick Sort é um algoritmo in-place, pois requer apenas uma pequena quantidade de memória adicional constante. No entanto, ele não é estável, o que significa que a troca de elementos pode mudar a ordem relativa dos elementos iguais. Apesar dessa limitação, a eficiência do Quick Sort em muitas situações práticas faz dele um dos algoritmos de ordenação mais utilizados. Ele é amplamente utilizado quando o critério mais importante na escolha do algoritmo é a eficiência em tempo médio, em detrimento à estabilidade, a exemplo de: ordenação de logs de servidor para análise; simulações numéricas e modelagem computacional, como a simulação de partículas em física computacional, entre outros.

```
#Pseudo-codigo representando o algoritmo

QuickSort(A, baixo, alto)
    se baixo < alto
        pi = partition(A, baixo, alto)
        QuickSort(A, baixo, pi - 1)
        QuickSort(A, pi + 1, alto)

partition(A, baixo, alto)
    pivot = A[alto]
    i = baixo - 1
    para j de baixo a alto - 1
        se A[j] <= pivot
            i = i + 1
            trocar A[i] e A[j]
    trocar A[i + 1] e A[alto]
```

```
17      retornar i + 1
```

2.6 Heap Sort

O Heap Sort é um algoritmo de ordenação que utiliza uma estrutura de dados chamada heap. Ele converte a lista em um heap, remove o maior elemento do heap e o coloca no final da lista, repetindo esse processo até que o heap esteja vazio. A estrutura de heap garante que a operação de remoção do maior elemento seja eficiente.

A complexidade do Heap Sort é $O(n \log n)$ em todos os casos. Isso ocorre porque a operação de heapificação requer $O(\log n)$ operações e é feita n vezes. Essa complexidade torna o Heap Sort eficiente para grandes listas de dados.

O Heap Sort é um algoritmo in-place, pois requer apenas uma pequena quantidade de memória adicional para o heap. No entanto, ele não é estável, o que significa que a construção do heap pode mudar a ordem relativa dos elementos iguais. Apesar dessa limitação, o Heap Sort é útil em muitas aplicações devido à sua eficiência e ao fato de não requerer memória adicional significativa.

```
1  #Pseudo-codigo representando o algoritmo
2
3  HeapSort(A)
4      n = tamanho(A)
5      para i de n // 2 - 1 a 0
6          heapify(A, n, i)
7      para i de n - 1 a 1
8          trocar A[i] e A[0]
9          heapify(A, i, 0)
10
11 heapify(A, n, i)
12     maior = i
13     esquerda = 2 * i + 1
14     direita = 2 * i + 2
15     se esquerda < n e A[esquerda] > A[maior]
16         maior = esquerda
17     se direita < n e A[direita] > A[maior]
18         maior = direita
19     se maior != i
20         trocar A[i] e A[maior]
21         heapify(A, n, maior)
```

3 METODOLOGIA

3.1 Experimento

Esse experimento foi implementado em Python3, valendo-se da biblioteca PyTest para a execução controlada dos testes. Foi conduzido em um dispositivo Intel i5-8265U @ 1.6GHz a 1.8 GHz, com 4 cores e 8 threads. O dispositivo possui ainda 16 Gb de memória RAM, ainda que, para a natureza desse experimento, a exigência de grande quantidade de memória RAM não exista. (BENTLEY, 1986)

A escolha do uso do PyTest se deu em razão das facilidade que a biblioteca fornece, como a possibilidade de parametrizar os testes de antemão, evitando escrita de código desnecessária e que possivelmente impactaria negativamente na execução de toda a bateria de testes.

Em razão da invariabilidade de hardware em nossos testes, optamos por executar cada algoritmo um total de dez vezes, por tipo de ordenação e por tamanho das listas, totalizando assim um total de 720 execuções divididas em 72 testes diferentes, um para cada combinação de algoritmo, tipo de ordenação e tamanho da lista de teste.

Além disso, como forma de registrar os testes utilizados, objetivando o possível reaproveitamento do conjunto de testes ante a algum erro na execução de todo o conjunto, foi utilizado a serialização de objetos python com o módulo Pickle. Isso teve como resultado 3 arquivos de extensão ”pkl”, um para cada tipo de ordenação das listas teste que estão disponíveis para uso público no repositório desse experimento, no GitHub.

3.2 Implementação

Quanto à implementação, algumas precauções foram tomadas antes do início dos testes, para evitar ao máximo situações como o estouro do limite de recursão ou mesmo execuções desnecessárias. Para isso, a adoção de soluções como o ”early exit”em algoritmos de complexidade quadrática e o uso da programação interativa em parte da implementação de algoritmos que valem de recursões, foram de grande importância.

Apesar dessa precaução tomada, tivemos problemas com a execução do ”Quick Sort”para grandes listas inversamente ordenadas, o que nos levou a modificar a forma com que implementamos esse algoritmo.

O número de comparações e trocas são retornados pelos algoritmos da maneira que melhor represente esse valor. Para métodos de ordenação como o Merge Sort a ideia da troca é meramente conceitual, já que na prática ela não existe, dada a natureza de suas operações. Para esses casos, adotamos a forma que melhor represente isso, o que consideramos, para o Merge Sort ser cada operação de reinserção dos valores no array de

onde originaram.

Abaixo estão listadas as implementações de cada um desses algoritmos:

```python
#Implementacao do algoritmo Bubble Sort

def bubble_sort(arr):
    n = len(arr)
    comparisons = 0
    swaps = 0
    for i in range(n):
        swapped = False
        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
                swapped = True
        if not swapped:
            break
    return comparisons, swaps, arr
```

```python
#Implementacao do algoritmo Insertion Sort

def insertion_sort(arr):
    comparisons = 0
    swaps = 0
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        if arr[j] <= key:
            comparisons += 1
            continue
        while j >= 0 and arr[j] > key:
            comparisons += 1
            arr[j + 1] = arr[j]
            j -= 1
            swaps += 1
        arr[j + 1] = key
        if j >= 0:
            comparisons += 1
    return comparisons, swaps, arr
```

```python
#Implementacao do algoritmo Selection Sort

def selection_sort(arr):
    n = len(arr)
    comparisons = 0
    swaps = 0
```

```
    for i in range(n):
        min_idx = i
        swapped = False
        for j in range(i+1, n):
            comparisons += 1
            if arr[j] < arr[min_idx]:
                min_idx = j
        if min_idx != i:
            arr[i], arr[min_idx] = arr[min_idx], arr[i]
            swaps += 1
            swapped = True
        if not swapped:
            break
    return comparisons, swaps, arr
```

```
#Implementacao do algoritmo Merge Sort

def merge_sort(arr: list):
    comparisons = [0]
    swaps = [0]

    def _merge_sort(arr: list, left: int, right: int, comparisons: list,
        swaps: list):
        if left < right:
            mid = (left + right) // 2
            _merge_sort(arr, left, mid, comparisons, swaps)
            _merge_sort(arr, mid + 1, right, comparisons, swaps)
            merge(arr, left, mid, right, comparisons, swaps)

    def merge(arr: list, left: int, mid: int, right: int, comparisons:
       list, swaps: list):
        n1 = mid - left + 1
        n2 = right - mid
        L = [0] * n1
        R = [0] * n2
        for i in range(n1):
            L[i] = arr[left + i]
        for i in range(n2):
            R[i] = arr[mid + 1 + i]
        i = 0
        j = 0
        k = left
        while i < n1 and j < n2:
            comparisons[0] += 1
            if L[i] <= R[j]:
                swaps[0] += 1
                arr[k] = L[i]
```

```
                i += 1
            else:
                swaps[0] += 1
                arr[k] = R[j]
                j += 1
            k += 1
        while i < n1:
            swaps[0] += 1
            arr[k] = L[i]
            i += 1
            k += 1
        while j < n2:
            swaps[0] += 1
            arr[k] = R[j]
            j += 1
            k += 1


    _merge_sort(arr, 0, len(arr) - 1, comparisons, swaps)
    return comparisons[0], swaps[0], arr
```

```
#Implementacao do algoritmo Quick Sort

def quick_sort(arr):
    comparisons = [0]
    swaps = [0]

    def _quick_sort(items, low, high):
        if low < high:
            pi = partition(items, low, high)
            _quick_sort(items, low, pi - 1)
            _quick_sort(items, pi + 1, high)

    def median_of_three(items, low, high):
        mid = (low + high) // 2
        if items[low] > items[mid]:
            items[low], items[mid] = items[mid], items[low]
            swaps[0] += 1
        if items[low] > items[high]:
            items[low], items[high] = items[high], items[low]
            swaps[0] += 1
        if items[mid] > items[high]:
            items[mid], items[high] = items[high], items[mid]
            swaps[0] += 1
        return mid

    def partition(items, low, high):
```

```
27          median_index = median_of_three(items, low, high)
28          items[median_index], items[high] = items[high], items[
                median_index]
29          swaps[0] += 1
30          pivot = items[high]
31          i = low - 1
32          for j in range(low, high):
33              comparisons[0] += 1
34              if items[j] < pivot:
35                  i += 1
36                  items[i], items[j] = items[j], items[i]
37                  swaps[0] += 1
38          items[i + 1], items[high] = items[high], items[i + 1]
39          swaps[0] += 1
40          return i + 1
41
42      _quick_sort(arr, 0, len(arr) - 1)
43      return comparisons[0], swaps[0], arr
```

```
1  #Implementacao do algoritmo Heap Sort
2
3  def heap_sort(arr):
4      n = len(arr)
5      comparisons = 0
6      swaps = 0
7      for i in range(n // 2 - 1, -1, -1): # Aqui construimos o max heap
8          comparisons, swaps = heapify(arr, n, i, comparisons, swaps)
9      for i in range(n-1, 0, -1): # aplicamos o heap sort em si
10         arr[i], arr[0] = arr[0], arr[i]
11         swaps += 1
12         comparisons, swaps = heapify(arr, i, 0, comparisons, swaps) #
               heapify o heap reduzido, para manter a propriedade de max
               heap
13     return comparisons, swaps, arr
14
15 def heapify(arr, n, i, comparisons, swaps):
16     largest = i
17     left = 2 * i + 1
18     right = 2 * i + 2
19     if left < n:
20         comparisons += 1
21         if arr[left] > arr[largest]:
22             largest = left
23     if right < n:
24         comparisons += 1
25         if arr[right] > arr[largest]:
26             largest = right
```

```
27      if largest != i:
28          arr[i], arr[largest] = arr[largest], arr[i]
29          swaps += 1
30          comparisons, swaps = heapify(arr, n, largest, comparisons, swaps
                )
31      return comparisons, swaps
```

Algumas decisões de implementação requerem uma explicação adicional. Como mencionado anteriormente, o Quick Sort se monstrou problemático para alguns casos e isso está diretamente ligado a escolha de um bom pivô. Em casos em que a má escolha do pivô é evidente, teremos que lidar com uma complexidade da ordem de $O(n^2)$, que prejudica a performance do algoritmo grandemente. Uma solução comum para esse problema com o Quick Sort é o método da mediana de três, em que três valores são escolhidos do array e é retornado aquele com maior potencial de provocar um particionamento uniforme, que é o termo médio, ou mediana entre os três valores. Em nossos testes, essa abordagem livrou os erros que haviam se manisfestado na execução do Quick Sort e o sacrifício de performance esperado com essa etapa intermediária na escolha do pivô não foi evidente. Outra importante adaptação se deu na implementação do Merge Sort. Como visto anteriormente o Merge Sort tem uma implementação bastante simples e baseada fortemente na recursão. Para nossos testes optamos por uma versão que não se valesse tanto dessa característica, evitando atingir o limite da recursão.

4 RESULTADOS

Essa seção se dedica à exposição dos resultados obtidos na execução dos testes nas condições impostas na descrição do experimento.

4.1 Listas de Ordenação Aleatória

Os resultados para listas com elementos dispostos de maneira aleatória são apresentados na Tabela 1, Tabela 2, Tabela 3 e Tabela 4. Podemos observar que, em geral, algoritmos como Merge Sort e Quick Sort apresentam um desempenho significativamente melhor em termos de tempo de execução e número de comparações quando comparados aos algoritmos Bubble Sort e Selection Sort, especialmente para listas maiores.

Tabela 1 – Comparação de Algoritmos de Ordenação para Listas Aleatórias de Tamanho 1000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	498904.8	249405.4	74.0
selection sort	324380.3	520.3	25.0
insertion sort	250397.8	249405.4	38.0
merge sort	8695.2	9976.0	5.0
quick sort	10984.2	6264.3	6.0
heap sort	16848.8	9077.6	4.0

Tabela 2 – Comparação de Algoritmos de Ordenação para Listas Aleatórias de Tamanho 10000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	49985634.1	25009452.7	8336.0
selection sort	29589895.4	4273.6	2474.0
insertion sort	25019440.7	25009452.7	4264.0
merge sort	120446.0	133616.0	54.0
quick sort	158972.6	86089.4	34.0
heap sort	235340.8	124156.9	57.0

Tabela 3 – Comparação de Algoritmos de Ordenação para Listas Aleatórias de Tamanho 50000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	1249924249.1	624546014.6	235805.0
selection sort	1012972265.1	34515.6	98378.0
insertion sort	624596002.6	624546014.6	108042.0
merge sort	718189.2	784464.0	303.0
quick sort	957767.5	511963.2	192.0
heap sort	1409771.1	737516.4	344.0

Tabela 4 – Comparação de Algoritmos de Ordenação para Listas Aleatórias de Tamanho 100000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	4999877450.7	2496899964.4	880109.0
selection sort	3105845501.9	45482.4	262712.0
insertion sort	2496999951.0	2496899964.4	440164.0
merge sort	1536357.3	1668928.0	638.0
quick sort	2022662.7	1084734.5	417.0
heap sort	3019691.1	1574982.3	727.0

4.2 Listas Ordenadas

Os resultados para listas previamente ordenadas são apresentados na Tabela 5, Tabela 6, Tabela 7 e Tabela 8. Algoritmos como Bubble Sort, Selection Sort e Insertion Sort mostraram desempenho superior neste cenário, apresentando um número significativamente menor de comparações e trocas, assim como tempos de execução reduzidos. Em contraste, algoritmos como Merge Sort e Heap Sort mostraram menor sensibilidade à ordenação prévia dos dados.

Tabela 5 – Comparação de Algoritmos de Ordenação para Listas Ordenadas de Tamanho 1000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	999.0	0.0	0.097
selection sort	999.0	0.0	0.079
insertion sort	999.0	0.0	0.101
merge sort	5044.0	9976.0	3.390
heap sort	17583.0	9708.0	3.890
quick sort	7987.4	4960.8	3.647

Tabela 6 – Comparação de Algoritmos de Ordenação para Listas Ordenadas de Tamanho 10000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	9999.0	0.0	1.048
selection sort	9999.0	0.0	0.792
insertion sort	9999.0	0.0	1.062
merge sort	69008.0	133616.0	44.401
heap sort	244460.0	131956.0	60.936
quick sort	113643.9	66485.7	31.730

Tabela 7 – Comparação de Algoritmos de Ordenação para Listas Ordenadas de Tamanho 50000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	49999.0	0.0	5.625
selection sort	49999.0	0.0	4.175
insertion sort	49999.0	0.0	5.879
merge sort	401952.0	784464.0	248
heap sort	1455438.0	773304.0	339
quick sort	684577.2	398356.0	198.843

Tabela 8 – Comparação de Algoritmos de Ordenação para Listas Ordenadas de Tamanho 100000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	99999.0	0.0	11.210
selection sort	99999.0	0.0	8.199
insertion sort	99999.0	0.0	11.563
merge sort	853904.0	1668928.0	521
heap sort	3112517.0	1650854.0	717
quick sort	1469876.9	847408.0	434.177

4.3 Listas Inversamente Ordenadas

Os resultados para listas ordenadas de forma decrescente são apresentados na Tabela **??**, Tabela **??**, Tabela 11 e Tabela 12. Este cenário é geralmente mais desafiador para algoritmos de ordenação, especialmente para Bubble Sort e Insertion Sort, que têm complexidade $O(n^2)$ no pior caso. Isso pode ser observado pelo alto número de comparações e trocas, além de tempos de execução significativamente maiores.

Tabela 9 – Comparação de Algoritmos de Ordenação para Listas Revertidas de Tamanho 1000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	499500.0	499500.0	99.0
selection sort	375249.0	500.0	33.0
insertion sort	499500.0	499500.0	75.0
merge sort	4932.0	9976.0	3.373
heap sort	15965.0	8316.0	3.406
quick sort	14220.8	9589.8	3.884

Tabela 10 – Comparação de Algoritmos de Ordenação para Listas Revertidas de Tamanho 10000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	49995000.0	49995000.0	11176.0
selection sort	37502499.0	5000.0	3098
insertion sort	49995000.0	49995000.0	8002.0
merge sort	64608.0	133616.0	40.0
heap sort	226682.0	116696.0	51.0
quick sort	218527.4	139151.1	86.400

Tabela 11 – Comparação de Algoritmos de Ordenação para Listas Revertidas de Tamanho 50000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	1249975000.0	1249975000.0	270403.0
selection sort	937512499.0	25000.0	76.851
insertion sort	1249975000.0	1249975000.0	201334.0
merge sort	382512.0	784464.0	253.0
heap sort	1366047.0	698892.0	308.0
quick sort	1344200.9	826886.1	379.83

Os resultados de todas essa amonstragem evidencia que algoritmos como Merge Sort e Quick Sort apresentam desempenho superior em termos de tempo de execução e número de comparações, especialmente para listas grandes e em diferentes tipos de ordenação.

Tabela 12 – Comparação de Algoritmos de Ordenação para Listas Revertidas de Tamanho 100000.

Algoritmo	Comparações	Trocas	Tempo (ms)
bubble sort	4999950000.0	4999950000.0	1393058.0
selection sort	3750024999.0	50000.0	383358.0
insertion sort	4999950000.0	4999950000.0	201334.0
merge sort	853904.0	1668928.0	521.0
heap sort	3112517.0	1650854.0	717.0
quick sort	2942429.8	1798914.8	761.310

Ainda, Algoritmos como Bubble Sort, Selection Sort e Insertion Sort se beneficiam significativamente quando as listas estão previamente ordenadas, resultando em um número mínimo de trocas e comparações. Em paralelo, o Quick Sort, utilizando a estratégia de ”median of three”para escolha do pivô, mostrou um desempenho consistente tanto para listas ordenadas quanto para listas revertidas, confirmando sua robustez.

Outra importante observação é que o Merge Sort e o Heap Sort mostraram menor sensibilidade à ordenação prévia dos dados. O Merge Sort, apesar de seu maior uso de memória, manteve um desempenho estável. O Heap Sort apresentou um número elevado de comparações e trocas, mas manteve tempos de execução aceitáveis.

Por fim, as listas revertidas representaram o pior cenário para a maioria dos algoritmos, com exceção do Merge Sort e do Quick Sort, que lidaram de forma mais eficiente com essas listas devido às suas características intrínsecas.

Os resultados também destacam a importância de selecionar o algoritmo de ordenação apropriado com base nas características dos dados de entrada e nos requisitos de desempenho do sistema.

5 DISCUSSÃO

Os resultados empíricos apresentados no capítulo anterior confirmam, em grande parte, as expectativas teóricas sobre os algoritmos de ordenação analisados. Algoritmos como Bubble Sort, Selection Sort e Insertion Sort apresentaram um desempenho significativamente inferior em termos de tempo de execução e número de comparações, especialmente para listas grandes e desordenadas. Isso era esperado, dado que esses algoritmos têm complexidade de tempo $O(n^2)$ no pior caso.

Por outro lado, Merge Sort e Quick Sort demonstraram um desempenho superior, como esperado, devido à sua complexidade de tempo $O(nlogn)$ no pior caso. A estratégia de "median of three"para escolha do pivô no Quick Sort ajudou a manter um desempenho consistente mesmo em listas ordenadas e revertidas. Heap Sort, apesar de também ter complexidade $O(nlogn)$, apresentou um número elevado de comparações e trocas, mas manteve tempos de execução aceitáveis. (MEHLHORN; SANDERS, 2008)

5.1 Considerações sobre a Facilidade de Implementação, Uso de Memória e Estabilidade dos Algoritmos

Os algoritmos Bubble Sort, Selection Sort e Insertion Sort são conhecidos por sua simplicidade de implementação e são frequentemente usados como exemplos introdutórios em cursos de algoritmos devido à sua clareza e facilidade de compreensão. Esses algoritmos têm a vantagem de serem algoritmos "in-place", não requerendo memória extra além da necessária para armazenar o array de entrada.

Em uma segundo análise, embora mais complexo que os anteriores, o Merge Sort é relativamente fácil de implementar, principalmente devido à sua abordagem recursiva e ao uso de sub-rotinas claras para dividir e mesclar. Ele não é um algoritmo in-place, pois requer memória adicional para armazenar os arrays temporários durante o processo de mesclagem, como mencionado na revisão teórica. Isso pode ser uma desvantagem em sistemas com memória limitada.

O Quick Sort, por sua vez, pode ter uma implementação mais complexa devido à necessidade de escolher um pivô de maneira eficiente e de particionar o array corretamente. A estratégia de "median of three"adiciona um pouco mais de complexidade, mas melhora a eficiência. Em sua forma in-place, o Quick Sort não requer memória adicional significativa além da pilha de recursão. No entanto, a profundidade da recursão pode ser um problema em casos extremos, como destacado na seção de implementação.

Por fim, o Heap Sort tem um implementação que é moderadamente complexa, pois envolve a construção de um heap e a manutenção das propriedades do heap durante a ordenação, o que exige um maior conhecimento de estrutura de dados. Ele não requer

memória adicional significativa, sendo classificado com ”in-place”.

6 CONCLUSÃO

Este trabalho realizou uma análise empírica de seis algoritmos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort. Foram testadas listas de diferentes tamanhos (1000, 10000, 50000 e 100000 elementos) e distribuições (random, sorted e reversed). As principais métricas avaliadas foram o número de comparações, o número de trocas e o tempo de execução.

Os principais achados deste trabalho incluem o desempenho superior dos algoritmos Merge Sort e Quick Sort, tanto em tempo de execução, quanto em número de comparações, especialemente para listas grandes e desordenadas. Ambos mantiveram a complexidade de $O(nlogn)$, mesmo nos piores casos, confirmando sua eficiencia.

Além disso, os algoritmos Bubble Sort, Selection Sort e Insertion Sort apresentaram desempenho significativamente inferior, especialmente em listas grandes. A alta complexidade desses algoritmos resultou em tempos de execução elevados e número excessivo de comparações e trocas. Por outro lado, esses mesmos algoritmos mostraram uma sensibilidade significativa à ordenação prévia dos dados, apresentando um número mínimo de comparações e trocas em listas já ordenadas, superando a performance do Merge Sort e Heap Sort.

6.1 Recomendações sobre o Uso de Cada Algoritmo em Diferentes Situações

Com base nos resultados obtidos, as seguintes recomendações sobre o uso dos algoritmos de ordenação podem ser feitas.

Recomenda-se evitar o uso de Bubble Sort para listas grandes devido à sua complexidade $O(n^2)$. Pode ser utilizado para listas muito pequenas ou como exemplo didático para introdução ao conceito de ordenação. Similar ao Bubble Sort, Selection Sort não é eficiente para listas grandes. Pode ser utilizado em situações onde a estabilidade não é um requisito e a simplicidade de implementação é importante.

O Selection Sort é adequado para listas pequenas ou quando se precisa de um algoritmo simples e eficiente em termos de implementação, sendo interessante sua utilização em sistemas embarcados ou onde os recursos são limitados.

Uma boa situação para o Insertion Sort é em sistemas onde a lista está quase sempre ordenada e novos elementos são inseridos gradualmente, como em gerenciadores de listas de tarefas ou listas de contatos.

O Merge Sort tem uma aplicabilidade mais forte, e é costumeiramente utilizado em sistemas de gerenciamento de grandes volumes de dados, como bancos de dados, processamento de arquivos grandes e algoritmos de busca complexos. Ideal para aplicações de ordenação externa onde os dados não cabem na memória principal. Além disso, por

ser estável, se mostra um bom algoritmo de ordenação para situações em que mais de um critério de ordenação precisa ser aplicado a um conjunto de dados.

Por outro lado, o Quick Sort pode ser utilizado em sistemas que necessitam de ordenação rápida e eficiente, como em motores de busca, jogos (para ordenar objetos no cenário), e sistemas de recomendação. Ele se mostra ideal para aplicações onde a memória é um recurso crítico, dado seu uso eficiente de espaço.

Em última análise, o Heap Sort se mostra adequado para sistemas embarcados e aplicações em tempo real onde a memória é limitada, sendo utilizado em algoritmos de prioridade, como filas de prioridade, schedulers de sistemas operacionais e sistemas de cache.

REFERÊNCIAS

BENTLEY, J. L. **Programming Pearls**. [S.l.]: Addison-Wesley, 1986.

CORMEN, T. H. et al. **Algoritmos: Teoria e Prática**. 3rd. ed. Cambridge, MA: Elsevier, 2009.

HOARE, C. A. R. Quicksort. **The Computer Journal**, Oxford University Press, v. 5, n. 1, p. 10–16, 1962.

KNUTH, D. E. **The Art of Computer Programming, Volume 3: Sorting and Searching**. [S.l.]: Addison-Wesley Professional, 1998.

MEHLHORN, K.; SANDERS, P. **Algorithms and Data Structures: The Basic Toolbox**. [S.l.]: Springer Science & Business Media, 2008.