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

DANIEL VINICIUS DA SILVA, ÉRICK SANTOS MARÇAL, BENEDITO JAIME MELO MORAES JUNIOR

# ANÁLISE EMPÍRICA DE ALGORITMOS DE ORDENAÇÃO

PALMAS (TO)
2024

# 1 INTRODUÇÃO

A ordenação de dados é uma operação fundamental na ciência da computação e em diversas áreas aplicadas. Muitos problemas computacionais, sejam eles simples ou complexos, exigem a ordenação como uma etapa intermediária ou final. Sua importância pode ser destacada pela eficiência em busca e recuperação de dados, pelo preprocessamento necessário para outros algoritmos, pela facilidade de visualização e análise, e pela melhoria de desempenho em sistemas de gerenciamento de banco de dados e outras aplicações de grande escala.

O objetivo deste trabalho é comparar o desempenho de seis algoritmos de ordenação diferentes em termos de tempo de execução, uso de memória e facilidade de implementação. Ao fazer isso, buscamos identificar quais algoritmos são mais adequados para diferentes situações e tipos de dados.

Especificamente, este trabalho analisará os seguintes algoritmos de ordenação:

- **Bubble Sort**: Um algoritmo simples e intuitivo com complexidade temporal de $O(n^2)$. É frequentemente utilizado para fins didáticos devido à sua simplicidade.
- **Selection Sort**: Outro algoritmo de complexidade $O(n^2)$, conhecido pela sua simplicidade, mas não muito eficiente para listas grandes.
- **Insertion Sort**: Também com complexidade $O(n^2)$, é eficiente para listas pequenas ou quase ordenadas e fácil de implementar.
- **Merge Sort**: Um algoritmo de ordenação eficiente com complexidade $O(n \log n)$ que utiliza uma abordagem de divisão e conquista, mas requer espaço adicional.
- **Quick Sort**: Amplamente utilizado devido à sua eficiência média de $O(n \log n)$, embora seu desempenho dependa da escolha do pivô.
- **Heap Sort**: Um algoritmo eficiente com complexidade $O(n \log n)$ que utiliza uma estrutura de dados heap, trabalhando em espaço in-place.

Nos capítulos subsequentes, serão detalhados os resultados experimentais, análises e discussões sobre o desempenho desses algoritmos, assim como sugestões para trabalhos futuros.

# 2 REVISÃO TEÓRICA

## 2.1 Bubble Sort

Bubble Sort é um algoritmo de ordenação in place e estável. Foi descrito por Edward Iverson em 1960. Ele compara repetidamente pares adjacentes de elementos e troca-os se estiverem na ordem errada. Este processo é repetido até que a lista esteja ordenada. Apesar de ser fácil de entender e implementar, o Bubble Sort é ineficiente para listas grandes, com uma complexidade de tempo $O(n^2)$.

## 2.2 Selection Sort

Selection Sort é um algoritmo de ordenação in place e instável. Foi descrito por Donald Knuth em seu livro "The Art of Computer Programming", publicado em 1968. Funciona selecionando repetidamente o menor (ou maior, dependendo da ordem desejada) elemento da lista não ordenada e trocando-o com o primeiro elemento não ordenado. Esse processo é repetido para os elementos restantes. Embora seja simples de implementar, o Selection Sort não é eficiente para listas grandes, pois tem uma complexidade de tempo $O(n^2)$.

## 2.3 Insertion Sort

Insertion Sort é um algoritmo de ordenação in place e estável. As primeiras descrições formais do Insertion Sort datam dos trabalhos de Donald Knuth em 1968. Ele constrói a lista ordenada um elemento por vez, comparando cada novo elemento com os elementos já ordenados e inserindo-o na posição correta. O Insertion Sort é eficiente para listas pequenas ou quase ordenadas, com uma complexidade de tempo média de $O(n^2)$. Ele é frequentemente utilizado em algoritmos híbridos e em aplicações onde a entrada é geralmente pequena ou quase ordenada.

## 2.4 Merge Sort

Merge Sort é um algoritmo de ordenação estável e baseado na técnica de dividir e conquistar. Foi desenvolvido por John von Neumann em 1945. O algoritmo divide repetidamente a lista ao meio até que cada sublista contenha apenas um elemento. Depois, ele combina essas sublistas de forma ordenada para produzir sublistas maiores, até que a lista inteira esteja ordenada. O Merge Sort é eficiente e tem uma complexidade de tempo O(n log n), sendo bastante usado em situações onde a estabilidade da ordenação é crucial.

## 2.5 Quick Sort

Quicksort é um algoritmo de ordenação in place e instável, descrito pela primeira vez nos trabalhos do programador britânico C. A. R. Hoare em 1962. Baseia-se no princípio de dividir um problema em dois subproblemas mais simples, que são resolvidos recursivamente até que se tornem problemas triviais.

## 2.6 Heap Sort

Heap Sort é um algoritmo de ordenação in place e instável. Foi desenvolvido por J. W. J. Williams em 1964. Baseia-se na estrutura de dados heap, especificamente em um heap máximo ou mínimo, para ordenar os elementos. O algoritmo primeiro constrói um heap a partir da lista de elementos e, em seguida, remove repetidamente o maior (ou menor) elemento do heap, reconstruindo o heap até que todos os elementos estejam ordenados. O Heap Sort tem uma complexidade de tempo O(n log n) e é eficiente tanto em termos de tempo quanto de espaço.

# 3 METODOLOGIA

Para a realização de comparação dos algoritmos foi determinado uma linguagem padrão para a implementação de todos eles, de forma que todos os algoritmos fossem executados em ambientes similares para evitar imprecisões nos dados coletados.

## 3.1 Descrição do Hardware

Os testes foram realizados em dois computadores de especificações diferentes, variando na quantidade de memória RAM e até marcas de processadores.

### 3.1.1 Notebook Dell Inspiron 15 3520

- Processador: Intel(R) Core(TM) i5-1235U
- Memória RAM: 16GB DDR4 (2x8GB)

### 3.1.2 Notebook Lenovo Ideapad Gaming 3i

- Processador: 11th Gen Intel(R) Core(TM) i5-11300H 3.11 GHz
- Memória RAM: 24GB DDR4 (1x8GB, 1x16GB)

## 3.2 Descrição do Software

Em relação ao software utilizado, os sistemas operacionais e a versão da linguagem utilizada foram variadas.

### 3.2.1 Notebook Dell Inspiron 15 3520

- Sistema Operacional: Windows 11 Home 23h2
- Versão do Python: 3.12.3

### 3.2.2 Notebook Lenovo Ideapad Gaming 3i

- Sistema Operacional: Gentoo Linux 23.0 (WSL2) no Windows 11 Pro 23h2
- Versão do Python: 3.12.3

### 3.3 Implementação dos Algoritmos

Os algoritmos foram implementados utilizando a linguagem de programação Python. A escolha do Python se deve à sua simplicidade e ao seu amplo uso em ambientes acadêmicos e de pesquisa. Cada algoritmo foi encapsulado em uma função que recebe como entrada uma lista de números e retorna uma tupla contendo a lista ordenada, o número de comparações realizadas e o número de trocas efetuadas. Essa abordagem permite uma avaliação detalhada do desempenho de cada algoritmo, tanto em termos de tempo de execução quanto em operações realizadas.

```python
def bubble_sort(arr):
    n = len(arr)
    comparisons = 0
    swaps = 0
    for i in range(n-1):
        swapped = False
        for j in range(0, n-i-1):
            comparisons += 1
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swaps += 1
        if not swapped:
            break
    return arr, comparisons, swaps

```

**Figura 1 – Algoritmo Bubble Sort**

```python
def selection_sort(arr):
    comparisons = 0
    swaps = 0

    def swap(i, j):
        nonlocal swaps
        arr[i], arr[j] = arr[j], arr[i]
        swaps += 1

    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            comparisons += 1
            if arr[j] < arr[min_index]:
                min_index = j
        swap(i, min_index)

    return arr, comparisons, swaps
```

**Figura 2 – Algoritmo Selection Sort**

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

**Figura 3 – Algoritmo Insertion Sort**

```python
def merge_sort(arr):
    comparisons = 0
    swaps = 0

    def merge(left, right):
        nonlocal comparisons, swaps
        merged = []
        l = r = 0
        while l < len(left) and r < len(right):
            comparisons += 1
            if left[l] < right[r]:
                merged.append(left[l])
                l += 1
            else:
                merged.append(right[r])
                r += 1
        merged.extend(left[l:])
        merged.extend(right[r:])
        return merged

    def sort(arr):
        if len(arr) <= 1:
            return arr
        mid = len(arr) // 2
        left = sort(arr[:mid])
        right = sort(arr[mid:])
        return merge(left, right)

    sorted_arr = sort(arr)
    return sorted_arr, comparisons, swaps
```

**Figura 4 – Algoritmo Merge Sort**

```python
def heap_sort(arr):
    comparisons = 0
    swaps = 0

    def heapify(n, i):
        nonlocal comparisons, swaps
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2
        comparisons += 1
        if l < n and arr[i] < arr[l]:
            largest = l
        comparisons += 1
        if r < n and arr[largest] < arr[r]:
            largest = r
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            swaps += 1
            heapify(n, largest)

    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        swaps += 1
        heapify(i, 0)
    return arr, comparisons, swaps
```

**Figura 5 – Algoritmo Quick Sort**

```python
def heap_sort(arr):
    comparisons = 0
    swaps = 0

    def heapify(n, i):
        nonlocal comparisons, swaps
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2
        comparisons += 1
        if l < n and arr[i] < arr[l]:
            largest = l
        comparisons += 1
        if r < n and arr[largest] < arr[r]:
            largest = r
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            swaps += 1
            heapify(n, largest)

    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        swaps += 1
        heapify(i, 0)
    return arr, comparisons, swaps
```

**Figura 6 – Algoritmo Heap Sort**

### 3.4 Medição do Tempo de Execução

Para medir o tempo de execução dos algoritmos foi utilizado a função time da biblioteca padrão do Python para medir o tempo de execução dos algoritmos em milissegundos. Esse método é portátil e funciona em qualquer sistema operacional. A função é chamada antes do algoritmo de ordenação ser iniciado e após o algoritmo ser finalizado.

### 3.5 Número de Comparações e Trocas

Em cada função que encapsula os algoritmos, foram adicionadas duas variáveis do tipo inteiro para a realização da contagem do número de comparações e trocas, que foram incrementadas nos pontos apropriados do código.

### 3.6 Automatização dos testes

Para a automatização dos testes, foi criado um programa que itera a cada algoritmo uma série testes com parâmetros diferentes. Foram utilizados três tipos de listas: ordenada, inversa e aleatório. No qual cada tipo foi executado em listas com mil, dez mil, cinquenta mil e cem mil elementos. Como cada algoritmo retorna o número de comparações e trocas, e o programa retorna o tempo em milissegundos, esses dados foram adicionados em um arquivo para análise posterior.

# 4 RESULTADOS

Os resultados dos testes de desempenho dos algoritmos de ordenação são apresentados a seguir. Foram medidos o tempo de execução, o número de comparações e o número de trocas para cada algoritmo, utilizando listas de diferentes tamanhos e tipos.

| **Algoritmo** | **Tipo de Lista** | **Tempo (ms)** | **Comparações** | **Trocas** |
|---|---|---|---|---|
| Bubble Sort | Ordenada | 0.0281 | 999 | 0 |
| | Inversa | 67.4055 | 499500 | 499500 |
| | Aleatória | 45.6985 | 498526 | 251276 |
| Selection Sort | Ordenada | 20.6921 | 499500 | 1000 |
| | Inversa | 22.2454 | 499500 | 1000 |
| | Aleatória | 23.0286 | 499500 | 1000 |
| Insertion Sort | Ordenada | 0.0501 | 999 | 0 |
| | Inversa | 62.1723 | 499500 | 500499 |
| | Aleatória | 29.8390 | 250200 | 250213 |
| Merge Sort | Ordenada | 0.8444 | 4932 | 0 |
| | Inversa | 0.7347 | 5044 | 0 |
| | Aleatória | 1.7973 | 8697 | 0 |
| Quick Sort | Ordenada | 0.7896 | 7987 | 4449 |
| | Inversa | 4.6695 | 14205 | 8339 |
| | Aleatória | 1.0910 | 9129.5 | 4809 |
| Heap Sort | Ordenada | 10.5016 | 20416 | 9708 |
| | Inversa | 4.6642 | 17632 | 8316 |
| | Aleatória | 1.5496 | 19216 | 9108 |

**Tabela 1 – Desempenho médio dos algoritmos de ordenação em diferentes tipos de listas de mil elementos**

| **Algoritmo** | **Tipo de Lista** | **Tempo (ms)** | **Comparações** | **Trocas** |
|---|---|---|---|---|
| Bubble Sort | Ordenada | 11.2090 | 99999 | 0 |
| | Inversa | 1186765.762 | 4999950000 | 4999950000 |
| | Aleatória | 618881.305 | 4999831171.5 | 2496960764.5 |
| Selection Sort | Ordenada | 228737.5215 | 4999950000 | 100000 |
| | Inversa | 1082539.8876 | 4999950000 | 100000 |
| | Aleatória | 264958.66765 | 4999950000 | 100000 |
| Insertion Sort | Ordenada | 23.0365 | 99999 | 0 |
| | Inversa | 669413.1012 | 4999950000 | 5000049999 |
| | Aleatória | 356693.93 | 2500578588 | 2500578589.5 |
| Merge Sort | Ordenada | 133.6524 | 815024 | 0 |
| | Inversa | 142.0869 | 853904 | 0 |
| | Aleatória | 248.6933 | 1536410 | 0 |
| Quick Sort | Ordenada | 164.4981 | 1468946 | 780565 |
| | Inversa | 272.5545 | 2940915 | 1690835 |
| | Aleatória | 182.9271 | 1697238 | 894868 |
| Heap Sort | Ordenada | 592.7821 | 3401708 | 1650854 |
| | Inversa | 542.1710 | 3094868 | 1497434 |
| | Aleatória | 573.198 | 3249966 | 1574983 |

**Tabela 2 – Desempenho médio dos algoritmos de ordenação em diferentes tipos de listas de cem mil elementos**

# 5 DISCUSSÃO

## 5.1 Desempenho dos Algoritmos

### 5.1.1 Quick Sort

O Quick-Sort apresentou o melhor desempenho em termos de tempo de execução, independente da máquina usada, como esperado devido à sua complexidade média de $O(n \log n)$. No entanto, o desempenho do Quick Sort pode variar dependendo da escolha do pivô e da presença de muitos elementos iguais.

### 5.1.2 Bubble Sort

O Bubble-Sort, com complexidade $O(n^2)$, foi o mais lento, especialmente para listas grandes. É simples de implementar e entender, mas muito ineficiente para listas grandes.

### 5.1.3 Merge Sort

O Merge-Sort, embora eficiente em termos de tempo com complexidade $O(n \log n)$, requer espaço adicional para a lista temporária, tornando-o menos adequado para sistemas com memória limitada.

### 5.1.4 Insertion Sort

O Insertion Sort, apesar de ter um pior desempenho para listas grandes com complexidade $O(n^2)$, é simples de implementar e usa pouca memória. Ele é mais eficiente para listas pequenas ou quase ordenadas.

### 5.1.5 Selection Sort

O Selection Sort, com complexidade $O(n^2)$, também é simples de implementar, mas igualmente ineficiente para listas grandes.

### 5.1.6 Heap Sort

O Heap Sort mostrou-se eficiente em termos de tempo com complexidade $O(n \log n)$. Ele é moderadamente complexo devido à manipulação da estrutura de heap, mas é eficiente e usa espaço in-place.

## 5.2 Uso de Memória

Os algoritmos variam no uso de memória, conforme descrito abaixo:

- **Bubble Sort, Selection Sort, Insertion Sort:** Utilizam memória in-place, ou seja, não requerem espaço adicional significativo além da lista original.
- **Merge Sort:** Requer espaço adicional para as sublistas, o que pode ser um fator limitante em sistemas com memória restrita.
- **Quick Sort:** Utiliza memória in-place, mas a recursão pode aumentar o uso da pilha de chamadas.
- **Heap Sort:** Utiliza memória in-place, mas a manipulação do heap pode ser mais complexa.

## 5.3 Estabilidade dos Algoritmos

Os algoritmos também variam em termos de estabilidade:

- **Estáveis:** Bubble Sort, Insertion Sort e Merge Sort mantêm a ordem relativa dos elementos iguais.
- **Instáveis:** Selection Sort, Quick Sort e Heap Sort não garantem a manutenção da ordem relativa dos elementos iguais.

## 5.4 Limitações e Sugestões para Estudos Futuros

Este trabalho teve algumas limitações, como a realização dos testes em um número limitado de configurações de hardware e a exclusão de outros algoritmos de ordenação avançados. Para estudos futuros, seria interessante:

- Testar os algoritmos em diferentes configurações de hardware para observar a variação de desempenho.
- Implementar e comparar outros algoritmos, como Radix Sort, Counting Sort e algoritmos híbridos como TimSort.
- Explorar o impacto de diferentes estratégias de escolha de pivô no Quick Sort.
- Analisar o desempenho dos algoritmos em dados de entrada específicos, como listas quase ordenadas ou com muitos elementos repetidos.

# 6 CONCLUSÃO

Com este trabalho deu para avaliar bem o desempenho dos seis algoritmos de ordenação principais em termos de tempo de execução, número de comparações e trocas. Os resultados confirmam que os algoritmos com complexidade O(n log n) (Merge Sort, Quick Sort e Heap Sort) são significativamente mais eficientes para listas grandes do que os algoritmos de complexidade $O(n^2)$.

**Recomendações**

- **Bubble Sort e Selection Sort**: Adequados apenas para listas muito pequenas ou fins educacionais.
- **Insertion Sort**: Útil para listas pequenas ou quase ordenadas.
- **Merge Sort**: Recomendado quando a estabilidade é crucial e há memória suficiente.
- **Quick Sort**: Geralmente a melhor escolha para desempenho rápido em listas grandes, mas cuidado com a escolha do pivô.
- **Heap Sort**: Boa escolha para um algoritmo de ordenação eficiente e in-place, mas menos usado que Quick Sort e Merge Sort.

# REFERÊNCIAS

Anany Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd edition, Pearson, 2011.

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10-16, 1962.

Sultanullah Jadoon, Salman Faiz Solehria, Mubashir Qayum. Optimized Selection Sort Algorithm is faster than Insertion Sort Algorithm: a Comparative Study. *Department of Information Technology, Hazara University, Haripur Campus, 2. Sarhad University, Peshawar, 3. National University of Computer and Emerging Sciences, Peshawar Campus*. 115002-3838 IJECS-IJENS © April 2011 IJENS.

Owen Astrachan. *Bubble Sort: An Archaeological Algorithmic Analysis*. Computer Science Department, Duke University, December 9, 2003.

Rohit Joshi, Govind Singh Panwar, Preeti Pathak. *Analysis of Non-Comparison Based Sorting Algorithms: A Review*. Dept. of CSE, GEHU, Dehradun, India. International Journal of Emerging Research in Management & Technology, ISSN: 2278-9359, Volume 2, Issue 12, December 2013.

Paul Biggar, David Gregg. *Sorting in the Presence of Branch Prediction and Caches: Fast Sorting on Modern Computers*. Technical Report TCD-CS-2005-57, Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland, August 2005.

Ramesh Chand Pandey. *Study and Comparison of Various Sorting Algorithms*. Master's thesis, Thapar University, Patiala, July 2008.

Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. *A Comparison of Sorting Algorithms for the Connection Machine CM-2*. Carnegie Mellon University, MIT, NEC Research Institute, University of Texas, and Thinking Machines Corp, 2000.

Adarsh Kumar Verma and Prashant Kumar, *A New Approach for Sorting List to Reduce Execution Time*. Department of Computer Science and Engineering, Galgotias College of Engineering and Technology, Greater Noida, India. Octuber 2013.