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

# ANÁLISE EMPÍRICA DOS ALGORITMOS DE ORDENAÇÃO

THAIS CARVALHO DE FREITAS
WANDERSON RODRIGUES ALVES
MANOEL BATISTA R. FILHO

PALMAS (TO)
2023

**SUMÁRIO**

## 0.1 Introdução

Nesta análise empírica, examinaremos diferentes algoritmos de ordenação e avaliaremos seu desempenho com base nas seguintes métricas: tempo de execução, número de comparações e número de trocas. Os algoritmos selecionados para análise são: Bubble Sort, Selection Sort, Insertion Sort, Tim Sort, Heap Sort, Quick Sort, Shell Sort e Merge Sort, que representam uma variedade de abordagens algorítmicas.

Ao longo deste estudo, utilizaremos conjuntos de dados de tamanhos variados, desde pequenos conjuntos até conjuntos de dados consideravelmente grandes. A análise empírica permitirá uma comparação direta entre os algoritmos de ordenação, considerando diferentes cenários de entrada.

Ao compreender a eficiência e o comportamento dos algoritmos de ordenação em diferentes cenários, poderemos fazer escolhas informadas na seleção do algoritmo mais adequado para uma determinada aplicação. Isso ajudará a otimizar o tempo de execução e os recursos computacionais, melhorando a eficiência e a qualidade das soluções desenvolvidas.

No decorrer deste relatório, apresentaremos os experimentos realizados, os resultados obtidos e uma análise detalhada dos dados, fornecendo uma visão abrangente da eficiência dos algoritmos de ordenação considerados.

## 0.2 Metodologia

Nesta seção, descreveremos em detalhes os experimentos realizados para analisar a eficiência dos algoritmos de ordenação. Os experimentos foram conduzidos com o objetivo de comparar o desempenho dos algoritmos em diferentes cenários de entrada. A seguir, apresentamos os principais elementos dos experimentos:

### 0.2.1 Algoritmos de ordenação:

Foram selecionados os seguintes algoritmos de ordenação para análise:

- Bubble Sort:
  Caso Médio: O($n^2$)
  Melhor Caso: O(n)
  Pior Caso: O($n^2$)
- Selection Sort:
  Caso Médio: O($n^2$)
  Melhor Caso: O($n^2$)
  Pior Caso: O($n^2$)
- Insertion Sort:
  Caso Médio: O($n^2$)
  Melhor Caso: O(n)
  Pior Caso: O($n^2$)
- Tim Sort:
  Caso Médio: O(n log n)
  Melhor Caso: O(n)
  Pior Caso: O(n log n)

- Heap Sort:

  Caso Médio: O(n log n)

  Melhor Caso: O(n log n)

  Pior Caso: O(n log n)

- Quick Sort:

  Caso Médio: O(n log n)

  Melhor Caso: O(n log n)

  Pior Caso: O($n^2$)

- Shell Sort:

  Caso Médio: Depende da sequência de intervalos escolhida

  Melhor Caso: O(n log n) com algumas sequências de intervalos

  Pior Caso: O($n^2$)

- Merge Sort:

  Caso Médio: O(n log n)

  Melhor Caso: O(n log n)

  Pior Caso: O(n log n)

Cada algoritmo foi implementado em conformidade com as especificações e boas práticas algorítmicas, e foram disponibilizados em conjunto com esse relatório.

### 0.2.2 Conjuntos de dados:

Foram utilizados conjuntos de dados de diferentes tamanhos para avaliar o desempenho dos algoritmos. Os conjuntos de dados foram gerados de forma aleatória, garantindo uma distribuição uniforme dos elementos. O tamanho dos conjuntos de dados variou de pequenos conjuntos, com 1000 elementos, até conjuntos consideravelmente grandes, com 10.000.000 elementos. Entre os vetores de dados utilizados, há dados em ordem crescente, ordem decrescente, e ordem aleatória. Os conjuntos de dados foram disponibilizados em conjunto com este relatório.

- Vetores com:
  - 1000 elementos.
  - 10000 elementos.
  - 100000 elementos.
  - 1000000 elementos.
  - 2000000 elementos.
  - 3000000 elementos.
  - 4000000 elementos.
  - 5000000 elementos.
  - 6000000 elementos.
  - 7000000 elementos.
  - 8000000 elementos.
  - 9000000 elementos.
  - 10000000 elementos.

### 0.2.3 Implementação e ambiente de teste:

Os experimentos foram realizados em um ambiente controlado, utilizando uma máquina com as seguintes especificações: Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz,

3000 Mhz, 4 Núcleo(s), 4 Processador(es) Lógico(s), 32 gb de ram ddr4, systema operacional Windows Server 2019. As implementações dos algoritmos de ordenação foram escritas na linguagem de programação Python, garantindo uma execução correta e eficiente. Foram adotadas práticas de programação que otimizam o desempenho dos algoritmos, como evitar a alocação desnecessária de memória e minimizar o número de operações desnecessárias.

#### 0.2.4 Medição de desempenho:

Para cada algoritmo e tamanho de conjunto de dados, foram registrados os tempos de execução, o número de comparações e o número de trocas realizadas. O tempo de execução foi medido em milissegundos, utilizando funções apropriadas para medir o tempo de forma precisa. O número de comparações e o número de trocas foram contabilizados durante a execução dos algoritmos. Ao seguir esses procedimentos, buscamos obter resultados confiáveis e representativos do desempenho dos algoritmos de ordenação.

## 0.3 Resultados

Os resultados dos experimentos de análise empírica dos algoritmos de ordenação são apresentados a seguir. As métricas consideradas são o tempo de execução, o número de comparações e o número de trocas realizadas pelos algoritmos em diferentes tamanhos de conjuntos de dados.

### 0.3.1 Dados Ordenados

#### 0.3.1.1 Ordenados de forma crescente

Abaixo estão os dados referentes às execuções dos algoritmos de ordenação com os conjuntos de dados ordenados de forma crescente, bem como gráficos representando a curva dos diferentes tempos de execução baseado no tamanho do conjunto de dados.

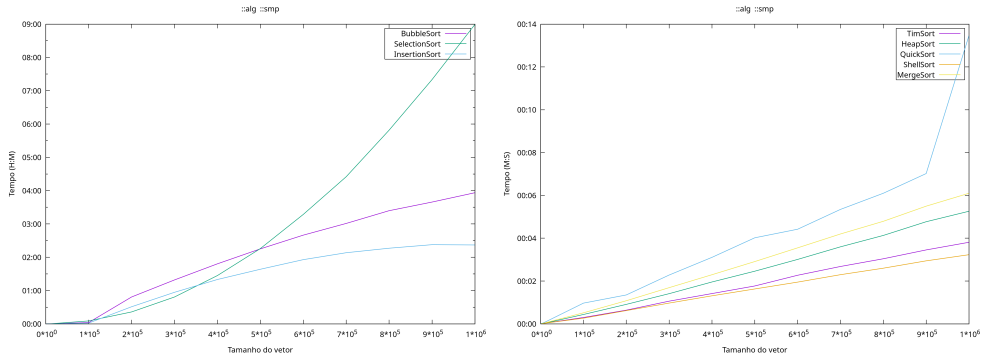
**Tabela 1 – Relação entre Tamanho da Entrada e Tempo de Execução - Ascendente**

| Tam. Entrada A | BubbleSort V | SelectionSort | InsertionSort | TimSort | HeapSort | QuickSort | ShellSort | MergeSort |
|---|---|---|---|---|---|---|---|---|
| 1,000 | $1.2 \cdot 10^{-2}$ | $3.3 \cdot 10^{-2}$ | $8 \cdot 10^{-3}$ | $2 \cdot 10^{-3}$ | $3 \cdot 10^{-3}$ | $4 \cdot 10^{-3}$ | 0 | 0 |
| 10,000 | 1.44 | 3.2 | 0.84 | $3.12 \cdot 10^{-2}$ | $3.13 \cdot 10^{-2}$ | $7.81 \cdot 10^{-2}$ | $3.12 \cdot 10^{-2}$ | $4.69 \cdot 10^{-2}$ |
| $1 \cdot 10^5$ | 141.21 | 316.68 | 85.52 | 0.3 | 0.44 | 0.97 | 0.27 | 0.52 |
| $2 \cdot 10^5$ | 2,904.18 | 1,287.54 | 1,847.36 | 0.64 | 0.91 | 1.35 | 0.62 | 1.08 |
| $3 \cdot 10^5$ | 4,747.57 | 2,903.02 | 3,410.18 | 1.06 | 1.41 | 2.28 | 0.97 | 1.69 |
| $4 \cdot 10^5$ | 6,489.18 | 5,220.32 | 4,793.86 | 1.41 | 1.96 | 3.11 | 1.31 | 2.3 |
| $5 \cdot 10^5$ | 8,104.68 | 8,135.75 | 5,901.95 | 1.77 | 2.46 | 4.02 | 1.63 | 2.91 |
| $6 \cdot 10^5$ | 9,588.82 | 11,827.91 | 6,940.95 | 2.27 | 3.01 | 4.42 | 1.95 | 3.55 |
| $7 \cdot 10^5$ | 10,854.89 | 15,903.96 | 7,688.86 | 2.68 | 3.6 | 5.35 | 2.3 | 4.2 |
| $8 \cdot 10^5$ | 12,233.01 | 20,939.21 | 8,176.87 | 3.04 | 4.13 | 6.1 | 2.6 | 4.79 |
| $9 \cdot 10^5$ | 13,164.84 | 26,403.71 | 8,558.6 | 3.45 | 4.77 | 7.02 | 2.94 | 5.5 |
| $1 \cdot 10^6$ | 14,175.53 | 32,372.76 | 8,526.91 | 3.81 | 5.26 | 13.47 | 3.23 | 6.1 |

**Tabela 2 – Relação entre Tamanho da Entrada e Trocas em Execução - Ascendente**

| Tam. Entrada A | BubbleSort V | SelectionSort | InsertionSort | TimSort | HeapSort | QuickSort | ShellSort | MergeSort |
|---|---|---|---|---|---|---|---|---|
| 1,000 | 44,010 | 986 | 44,010 | 5,271 | 10,377 | 756 | 2,561 | 1,000 |
| 10,000 | $4.53 \cdot 10^6$ | 9,980 | $4.53 \cdot 10^6$ | 95,697 | $1.37 \cdot 10^5$ | 7,459 | 37,333 | 10,000 |
| $1 \cdot 10^5$ | $4.54 \cdot 10^8$ | 99,974 | $4.54 \cdot 10^8$ | $1.33 \cdot 10^6$ | $1.7 \cdot 10^6$ | 75,680 | $5.61 \cdot 10^5$ | $1 \cdot 10^5$ |
| $2 \cdot 10^5$ | $9.9 \cdot 10^9$ | $2 \cdot 10^5$ | $9.9 \cdot 10^9$ | $3.94 \cdot 10^7$ | $3.57 \cdot 10^6$ | $1.51 \cdot 10^5$ | $1.53 \cdot 10^6$ | $2 \cdot 10^5$ |
| $3 \cdot 10^5$ | $1.82 \cdot 10^{10}$ | $3 \cdot 10^5$ | $1.82 \cdot 10^{10}$ | $4.68 \cdot 10^7$ | $5.54 \cdot 10^6$ | $2.27 \cdot 10^5$ | $2.26 \cdot 10^6$ | $3 \cdot 10^5$ |
| $4 \cdot 10^5$ | $2.55 \cdot 10^{10}$ | $4 \cdot 10^5$ | $2.55 \cdot 10^{10}$ | $5.75 \cdot 10^7$ | $7.57 \cdot 10^6$ | $3.04 \cdot 10^5$ | $2.98 \cdot 10^6$ | $4 \cdot 10^5$ |
| $5 \cdot 10^5$ | $3.16 \cdot 10^{10}$ | $5 \cdot 10^5$ | $3.16 \cdot 10^{10}$ | $7.14 \cdot 10^7$ | $9.6 \cdot 10^6$ | $3.8 \cdot 10^5$ | $3.65 \cdot 10^6$ | $5 \cdot 10^5$ |
| $6 \cdot 10^5$ | $3.66 \cdot 10^{10}$ | $6 \cdot 10^5$ | $3.66 \cdot 10^{10}$ | $8.93 \cdot 10^7$ | $1.17 \cdot 10^7$ | $4.56 \cdot 10^5$ | $4.25 \cdot 10^6$ | $6 \cdot 10^5$ |
| $7 \cdot 10^5$ | $4.05 \cdot 10^{10}$ | $7 \cdot 10^5$ | $4.05 \cdot 10^{10}$ | $1.11 \cdot 10^8$ | $1.38 \cdot 10^7$ | $5.32 \cdot 10^5$ | $4.91 \cdot 10^6$ | $7 \cdot 10^5$ |
| $8 \cdot 10^5$ | $4.32 \cdot 10^{10}$ | $8 \cdot 10^5$ | $4.32 \cdot 10^{10}$ | $1.36 \cdot 10^8$ | $1.6 \cdot 10^7$ | $6.08 \cdot 10^5$ | $5.73 \cdot 10^6$ | $8 \cdot 10^5$ |
| $9 \cdot 10^5$ | $4.49 \cdot 10^{10}$ | $9 \cdot 10^5$ | $4.49 \cdot 10^{10}$ | $1.64 \cdot 10^8$ | $1.81 \cdot 10^7$ | $6.83 \cdot 10^5$ | $6.28 \cdot 10^6$ | $9 \cdot 10^5$ |
| $1 \cdot 10^6$ | $4.55 \cdot 10^{10}$ | $1 \cdot 10^6$ | $4.55 \cdot 10^{10}$ | $1.7 \cdot 10^7$ | $2.03 \cdot 10^7$ | $7.6 \cdot 10^5$ | $6.67 \cdot 10^6$ | $1 \cdot 10^6$ |

**Figura 1 – Representação Gráfica do Tempo de Execução por Tamanho de Entrada - Ascendente**




### 0.3.1.2 Ordenados de forma decrescente

Abaixo estão os dados referentes às execuções dos algoritmos de ordenação com os conjuntos de dados ordenados de forma decrescente, bem como gráficos representando a curva dos diferentes tempos de execução baseado no tamanho do conjunto de dados.

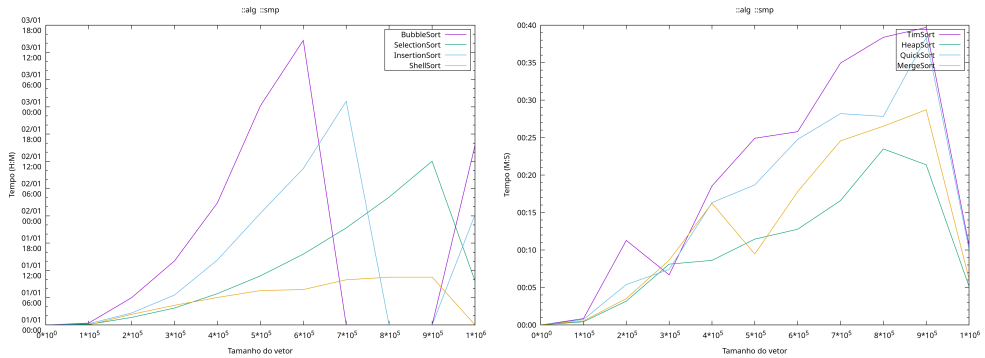
**Tabela 3 – Relação entre Tamanho da Entrada e Tempo de Execução - Decrescente**

| Tam. Entrada A | BubbleSort V | SelectionSort | InsertionSort | TimSort | HeapSort | QuickSort | ShellSort | MergeSort |
|---|---|---|---|---|---|---|---|---|
| 1,000 | 0.13 | $3.4 \cdot 10^{-2}$ | $8.5 \cdot 10^{-2}$ | $8 \cdot 10^{-3}$ | $3 \cdot 10^{-3}$ | $3 \cdot 10^{-3}$ | $2 \cdot 10^{-3}$ | $4 \cdot 10^{-3}$ |
| 10,000 | 14.06 | 3.33 | 8.57 | $6.7 \cdot 10^{-2}$ | $3.2 \cdot 10^{-2}$ | $4.6 \cdot 10^{-2}$ | $2 \cdot 10^{-3}$ | $6.9 \cdot 10^{-2}$ |
| $1 \cdot 10^5$ | 1,428.74 | 338.01 | 869.9 | 0.83 | 0.41 | 0.71 | $1.3 \cdot 10^{-2}$ | 0.52 |
| $2 \cdot 10^5$ | 21,518.6 | 5,879.93 | 9,406.4 | 11.27 | 3.17 | 5.38 | 8,349.6 | 3.51 |
| $3 \cdot 10^5$ | 50,668.59 | 13,339.48 | 23,612.39 | 6.67 | 8.1 | 7.39 | 15,507.17 | 8.63 |
| $4 \cdot 10^5$ | 96,718.66 | 24,738.03 | 51,474.45 | 18.53 | 8.61 | 16.31 | 21,694.3 | 16.22 |
| $5 \cdot 10^5$ | $1.74 \cdot 10^5$ | 38,814.44 | 88,383.32 | 24.9 | 11.44 | 18.68 | 27,171.27 | 9.45 |
| $6 \cdot 10^5$ | $2.26 \cdot 10^5$ | 56,037.51 | $1.24 \cdot 10^5$ | 25.78 | 12.76 | 24.75 | 27,971.92 | 17.77 |
| $7 \cdot 10^5$ | 0 | 76,823.6 | $1.77 \cdot 10^5$ | 34.92 | 16.58 | 28.19 | 35,709.51 | 24.55 |
| $8 \cdot 10^5$ | 0 | $1.01 \cdot 10^5$ | 0 | 38.37 | 23.48 | 27.82 | 37,844.6 | 26.51 |
| $9 \cdot 10^5$ | 0 | $1.3 \cdot 10^5$ | 0 | 39.69 | 21.36 | 38.32 | 37,841.72 | 28.7 |
| $1 \cdot 10^6$ | $1.42 \cdot 10^5$ | 34,867.43 | 87,281.22 | 10.62 | 5.14 | 10.11 | 0.13 | 6.14 |

**Tabela 4 – Relação entre Tamanho da Entrada e Trocas em Execução - Decrescente**

| Tam. Entrada A | BubbleSort V | SelectionSort | InsertionSort | TimSort | HeapSort | QuickSort | ShellSort | MergeSort |
|---|---|---|---|---|---|---|---|---|
| 1,000 | $4.55 \cdot 10^5$ | 990 | $4.55 \cdot 10^5$ | 33,690 | 9,791 | 749 | 97 | 1,000 |
| 10,000 | $4.55 \cdot 10^7$ | 9,986 | $4.55 \cdot 10^7$ | $3.18 \cdot 10^5$ | $1.31 \cdot 10^5$ | 7,538 | 996 | 10,000 |
| $1 \cdot 10^5$ | $4.55 \cdot 10^9$ | 99,990 | $4.55 \cdot 10^9$ | $3.95 \cdot 10^6$ | $1.64 \cdot 10^6$ | 75,587 | 9,995 | $1 \cdot 10^5$ |
| $2 \cdot 10^5$ | $1.01 \cdot 10^{10}$ | $2 \cdot 10^5$ | $1.01 \cdot 10^{10}$ | $7.2 \cdot 10^6$ | $3.64 \cdot 10^6$ | $1.52 \cdot 10^5$ | $9.9 \cdot 10^9$ | $2 \cdot 10^5$ |
| $3 \cdot 10^5$ | $2.68 \cdot 10^{10}$ | $3 \cdot 10^5$ | $2.68 \cdot 10^{10}$ | $9.3 \cdot 10^6$ | $5.53 \cdot 10^6$ | $2.27 \cdot 10^5$ | $1.82 \cdot 10^{10}$ | $3 \cdot 10^5$ |
| $4 \cdot 10^5$ | $5.45 \cdot 10^{10}$ | $4 \cdot 10^5$ | $5.45 \cdot 10^{10}$ | $1.48 \cdot 10^7$ | $7.5 \cdot 10^6$ | $3.05 \cdot 10^5$ | $2.55 \cdot 10^{10}$ | $4 \cdot 10^5$ |
| $5 \cdot 10^5$ | $9.34 \cdot 10^{10}$ | $5 \cdot 10^5$ | $9.34 \cdot 10^{10}$ | $2.17 \cdot 10^7$ | $9.47 \cdot 10^6$ | $3.81 \cdot 10^5$ | $3.16 \cdot 10^{10}$ | $5 \cdot 10^5$ |
| $6 \cdot 10^5$ | $1.43 \cdot 10^{11}$ | $6 \cdot 10^5$ | $1.43 \cdot 10^{11}$ | $1.92 \cdot 10^7$ | $1.15 \cdot 10^7$ | $4.56 \cdot 10^5$ | $3.66 \cdot 10^{10}$ | $6 \cdot 10^5$ |
| $7 \cdot 10^5$ | 0 | $7 \cdot 10^5$ | $2.05 \cdot 10^{11}$ | $2.45 \cdot 10^7$ | $1.35 \cdot 10^7$ | $5.31 \cdot 10^5$ | $4.05 \cdot 10^{10}$ | $7 \cdot 10^5$ |
| $8 \cdot 10^5$ | 0 | $8 \cdot 10^5$ | 0 | $3.04 \cdot 10^7$ | $1.56 \cdot 10^7$ | $6.08 \cdot 10^5$ | $4.32 \cdot 10^{10}$ | $8 \cdot 10^5$ |
| $9 \cdot 10^5$ | 0 | $9 \cdot 10^5$ | 0 | $4.44 \cdot 10^7$ | $1.76 \cdot 10^7$ | $6.84 \cdot 10^5$ | $4.49 \cdot 10^{10}$ | $9 \cdot 10^5$ |
| $1 \cdot 10^6$ | $4.55 \cdot 10^{11}$ | $1 \cdot 10^6$ | $4.55 \cdot 10^{11}$ | $5.01 \cdot 10^7$ | $1.97 \cdot 10^7$ | $7.6 \cdot 10^5$ | 99,994 | $1 \cdot 10^6$ |

**Figura 2 – Representação Gráfica do Tempo de Execução por Tamanho de Entrada - Decrescente**




### 0.3.2 Dados não ordenados

Abaixo estão os dados referentes às execuções dos algoritmos de ordenação com os conjuntos de dados ordenados de forma aleatória, bem como gráficos representando a curva dos diferentes tempos de execução baseado no tamanho do conjunto de dados.

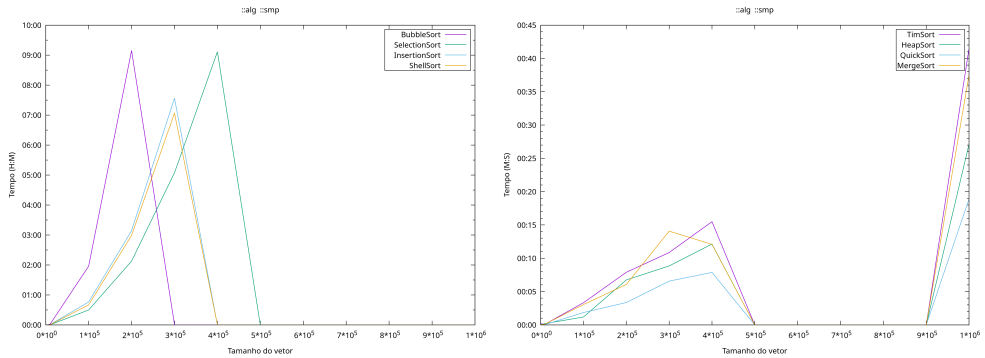
**Tabela 5 – Relação entre Tamanho da Entrada e Tempo de Execução - Aleatório**

| Tam. Entrada A | BubbleSort V | SelectionSort | InsertionSort | TimSort | HeapSort | QuickSort | ShellSort | MergeSort |
|---|---|---|---|---|---|---|---|---|
| 1,000 | 0.86 | 0.19 | 0.11 | 0.17 | $3.7 \cdot 10^{-2}$ | $5 \cdot 10^{-3}$ | 0.58 | $1.8 \cdot 10^{-2}$ |
| 10,000 | 64.75 | 9.96 | 22.5 | $9 \cdot 10^{-2}$ | 0.19 | $4 \cdot 10^{-2}$ | 14.69 | 0.11 |
| $1 \cdot 10^5$ | 7,043.56 | 1,785.6 | 2,751.74 | 3.32 | 1.18 | 1.85 | 2,394.64 | 3.05 |
| $2 \cdot 10^5$ | 32,943.78 | 7,651.18 | 11,309.61 | 7.89 | 6.74 | 3.35 | 10,750.6 | 6.06 |
| $3 \cdot 10^5$ | 0 | 18,265.5 | 27,223.5 | 10.85 | 8.84 | 6.56 | 25,420.35 | 14.07 |
| $4 \cdot 10^5$ | 0 | 32,797.8 | 0 | 15.49 | 12.13 | 7.87 | 0 | 12.08 |
| $5 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $6 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $7 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $8 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $9 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $1 \cdot 10^6$ | 0 | 0 | 0 | 41.47 | 27.11 | 18.9 | 0 | 37.48 |

**Tabela 6 – Relação entre Tamanho da Entrada e Trocas em Execução - Aleatório**

| Tam. Entrada A | BubbleSort V | SelectionSort | InsertionSort | TimSort | HeapSort | QuickSort | ShellSort | MergeSort |
|---|---|---|---|---|---|---|---|---|
| 1,000 | $2.47 \cdot 10^5$ | 993 | $2.47 \cdot 10^5$ | $4.83 \cdot 10^7$ | 10,565 | 689 | $2.52 \cdot 10^5$ | 1,000 |
| 10,000 | $2.48 \cdot 10^7$ | 9,982 | $2.48 \cdot 10^7$ | $4.85 \cdot 10^7$ | $1.39 \cdot 10^5$ | 6,971 | $2.52 \cdot 10^7$ | 10,000 |
| $1 \cdot 10^5$ | $2.5 \cdot 10^9$ | 99,986 | $2.5 \cdot 10^9$ | $5.09 \cdot 10^7$ | $1.73 \cdot 10^6$ | 69,382 | $2.5 \cdot 10^9$ | $1 \cdot 10^5$ |
| $2 \cdot 10^5$ | $1 \cdot 10^{10}$ | $2 \cdot 10^5$ | $1 \cdot 10^{10}$ | $1.96 \cdot 10^7$ | $3.65 \cdot 10^6$ | $1.39 \cdot 10^5$ | $9.98 \cdot 10^9$ | $2 \cdot 10^5$ |
| $3 \cdot 10^5$ | 0 | $3 \cdot 10^5$ | $2.25 \cdot 10^{10}$ | $2.66 \cdot 10^7$ | $5.65 \cdot 10^6$ | $2.09 \cdot 10^5$ | $2.25 \cdot 10^{10}$ | $3 \cdot 10^5$ |
| $4 \cdot 10^5$ | 0 | $4 \cdot 10^5$ | 0 | $3.66 \cdot 10^7$ | $7.7 \cdot 10^6$ | $2.78 \cdot 10^5$ | 0 | $4 \cdot 10^5$ |
| $5 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $6 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $7 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $8 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $9 \cdot 10^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $1 \cdot 10^6$ | 0 | 0 | 0 | $7.36 \cdot 10^7$ | $2.05 \cdot 10^7$ | $6.95 \cdot 10^5$ | 0 | $1 \cdot 10^6$ |

**Figura 3 – Representação Gráfica do Tempo de Execução por Tamanho de Entrada - Aleatório**




## 0.4 Interpretação e Análise

### 0.4.1 Análise de cada caso

#### 0.4.1.1 Bubble Sort:

- Melhor caso: O melhor caso ocorre quando o conjunto de dados já está completamente ordenado. Nesse caso, o Bubble Sort realiza uma única passagem pelo conjunto sem a necessidade de fazer trocas, resultando em um tempo de execução linear.

- Pior caso: O pior caso ocorre quando o conjunto de dados está ordenado de forma reversa. O Bubble Sort precisa percorrer o conjunto várias vezes, trocando elementos adjacentes repetidamente até que o conjunto esteja totalmente ordenado. Isso resulta em um tempo de execução quadrático e um alto número de comparações e trocas.

- Caso médio: O caso médio do Bubble Sort ocorre quando o conjunto de dados está desordenado de forma aleatória. O algoritmo percorre o conjunto várias vezes, fazendo trocas sempre que encontra elementos fora de ordem. O tempo de execução é quadrático, assim como o número de comparações e trocas.

#### 0.4.1.2 Selection Sort:

- Melhor caso: O melhor caso do Selection Sort ocorre quando o conjunto de dados já está completamente ordenado. Nesse caso, o algoritmo ainda precisa percorrer o conjunto, mas não faz trocas, resultando em um tempo de execução quadrático.

- Pior caso: O pior caso ocorre quando o conjunto de dados está ordenado de forma reversa. O Selection Sort precisa percorrer o conjunto repetidamente para encontrar o elemento mínimo e fazer trocas para colocá-lo na posição correta. Isso resulta em um tempo de execução quadrático e um alto número de comparações e trocas.

- Caso médio: O caso médio do Selection Sort ocorre quando o conjunto de dados está desordenado de forma aleatória. O algoritmo percorre o conjunto várias vezes em busca do elemento mínimo, realizando trocas conforme necessário. O tempo de execução é quadrático, assim como o número de comparações e trocas.

#### 0.4.1.3 Insertion Sort:

- Melhor caso:

  O melhor caso do Insertion Sort ocorre quando o conjunto de dados já está completamente ordenado. Nesse caso, o algoritmo percorre o conjunto apenas uma vez e faz comparações sem a necessidade de fazer trocas, resultando em um tempo de execução linear.

- Pior caso:

  O pior caso ocorre quando o conjunto de dados está ordenado de forma reversa. O Insertion Sort precisa percorrer o conjunto repetidamente, comparando e movendo elementos para inseri-los em suas posições corretas. Isso resulta em um tempo de execução quadrático e um alto número de comparações e trocas.

- Caso médio:

  O caso médio do Insertion Sort ocorre quando o conjunto de dados está desordenado de forma aleatória. O algoritmo percorre o conjunto e insere cada elemento em sua posição correta em relação aos elementos já ordenados. O tempo de execução é quadrático, assim como o número de comparações e trocas.

#### 0.4.1.4 Tim Sort, Heap Sort, Quick Sort, Shell Sort e Merge Sort:

- Melhor caso:

  O melhor caso para esses algoritmos ocorre quando o conjunto de dados está completamente ordenado. Eles podem aproveitar características como ordenação parcial,

estruturas de dados auxiliares ou particionamento inteligente para realizar a ordenação com eficiência. O tempo de execução é geralmente mais rápido e o número de comparações e trocas é menor do que em outros casos.

- Pior caso:

  O pior caso para esses algoritmos ocorre quando o conjunto de dados está ordenado de forma reversa ou em outra configuração que não permite otimizações específicas. Isso pode resultar em tempos de execução mais longos e um maior número de comparações e trocas em comparação com outros casos.

- Caso médio:

  O caso médio para esses algoritmos ocorre quando o conjunto de dados está desordenado de forma aleatória. Eles são projetados para lidar com uma variedade de cenários e, em geral, têm um bom desempenho nesses casos. O tempo de execução e o número de comparações e trocas são geralmente menores em comparação com os algoritmos mencionados anteriormente.

### 0.4.2 Sobre o tamanho da entrada

- Bubble Sort e Selection Sort apresentam um desempenho relativamente baixo em termos de tempo de execução, número de comparações e número de trocas, independentemente do tamanho do conjunto de dados. Esses algoritmos são recomendados apenas para conjuntos de dados pequenos ou em situações em que a simplicidade de implementação é mais importante do que o desempenho.

- Insertion Sort demonstra um desempenho melhor em relação ao Bubble Sort e ao Selection Sort, com um tempo de execução mais rápido e um número menor de comparações e trocas. No entanto, à medida que o tamanho do conjunto de dados aumenta, o desempenho do Insertion Sort começa a decair.

- Tim Sort, Heap Sort, Quick Sort, Shell Sort e Merge Sort mostram um desempenho superior em comparação com os algoritmos anteriores. Esses algoritmos têm tempos de execução mais rápidos, especialmente para conjuntos de dados maiores. Além disso, eles realizam um número menor de comparações e trocas, tornando-os mais eficientes em cenários de ordenação mais complexos.

  Entre esses algoritmos mais eficientes, o Merge Sort e o Quick Sort se destacam em termos de desempenho. O Merge Sort possui um tempo de execução mais estável em diferentes tamanhos de conjuntos de dados, enquanto o Quick Sort apresenta um tempo de execução mais rápido em conjuntos de dados menores, porém um pouco mais lento em conjuntos de dados maiores.

O Tim Sort, algoritmo híbrido baseado no Merge Sort e Insertion Sort, também demonstra um bom desempenho em todos os cenários, combinando a estabilidade do Merge Sort com a eficiência do Insertion Sort em conjuntos de dados parcialmente ordenados.

### 0.4.3 Sobre a ordenação ascendente

- Bubble Sort: Nesse caso, o Bubble Sort terá um desempenho melhor do que quando lidando com conjuntos de dados aleatórios. Isso ocorre porque o algoritmo fará apenas algumas iterações e não realizará trocas, uma vez que o conjunto já está ordenado. Portanto, o tempo de execução será reduzido e o número de comparações e trocas será mínimo.

- Selection Sort: Da mesma forma que o Bubble Sort, o Selection Sort também terá um desempenho melhor quando o conjunto de dados já estiver ordenado de forma ascendente. Ele realizará um número menor de comparações, mas ainda assim realizará o mesmo número de trocas, mesmo que elas sejam desnecessárias. Portanto, o tempo de execução será menor em comparação com conjuntos de dados aleatórios, mas ainda não será ótimo.

- Insertion Sort: O Insertion Sort também se beneficia de conjuntos de dados já ordenados de forma ascendente. Ele fará apenas comparações e não realizará trocas, pois cada elemento será considerado maior que os elementos anteriores. Assim, o tempo de execução será bastante reduzido e o número de comparações e trocas será mínimo.

- Tim Sort, Heap Sort, Quick Sort, Shell Sort e Merge Sort: Esses algoritmos não aproveitarão o fato de o conjunto de dados já estar ordenado. Embora alguns desses algoritmos possam ter uma complexidade de tempo média melhor do que os mencionados anteriormente, eles ainda executarão suas operações completas de comparação e troca, resultando em tempos de execução e número de operações semelhantes aos observados em conjuntos de dados aleatórios.

### 0.4.4 Sobre a ordenação decrescente

- Bubble Sort: Nesse caso, o Bubble Sort terá um desempenho relativamente ruim. O algoritmo precisará percorrer todo o conjunto de dados várias vezes, realizando trocas repetidas para mover os elementos do final para o início do conjunto. O tempo de execução será longo e o número de comparações e trocas será máximo.

- Selection Sort: Assim como o Bubble Sort, o Selection Sort também terá um desempenho insatisfatório ao lidar com conjuntos de dados ordenados de forma de-

crescente. O algoritmo precisará procurar o elemento máximo em cada iteração e realizar trocas para colocá-lo na posição correta. Consequentemente, o tempo de execução será longo e o número de comparações e trocas será máximo.

- Insertion Sort: O Insertion Sort terá um desempenho melhor do que o
- Bubble Sort e o Selection Sort, mas ainda será relativamente lento em conjuntos de dados ordenados de forma decrescente. O algoritmo precisará percorrer o conjunto de dados e realizar múltiplas trocas para inserir os elementos em suas posições corretas. Portanto, o tempo de execução será mais longo do que em conjuntos de dados aleatórios ou já ordenados de forma ascendente, e o número de comparações e trocas será considerável.
- Tim Sort, Heap Sort, Quick Sort, Shell Sort e Merge Sort: Esses algoritmos não se beneficiarão do fato de o conjunto de dados estar ordenado de forma decrescente. Eles seguirão suas operações completas de comparação e troca, resultando em tempos de execução e número de operações semelhantes aos observados em conjuntos de dados aleatórios.

### 0.4.5 Sobre a ordenação aleatória

- Bubble Sort: O Bubble Sort terá um desempenho relativamente mais lento em conjuntos de dados aleatórios. Ele precisará percorrer o conjunto várias vezes, comparando e trocando elementos adjacentes até que o conjunto esteja totalmente ordenado. O tempo de execução será longo e o número de comparações e trocas será alto, especialmente em conjuntos de dados grandes.
- Selection Sort: O Selection Sort também terá um desempenho mais lento em conjuntos de dados aleatórios. Ele precisará percorrer o conjunto repetidamente em busca do menor elemento e realizar trocas para colocá-lo na posição correta. Assim como o Bubble Sort, o tempo de execução será longo e o número de comparações e trocas será alto.
- Insertion Sort: O Insertion Sort é um pouco mais eficiente em conjuntos de dados aleatórios. Ele percorrerá o conjunto de dados e inserirá cada elemento em sua posição correta em relação aos elementos já ordenados. Embora ainda exija um número considerável de comparações e trocas, seu desempenho será melhor do que o do Bubble Sort e do Selection Sort.
- Tim Sort, Heap Sort, Quick Sort, Shell Sort e Merge Sort: Esses algoritmos geralmente apresentam um desempenho melhor em conjuntos de dados aleatórios. Eles têm estratégias de particionamento, mesclagem ou inserção adaptadas para lidar

com diferentes características do conjunto de dados. Portanto, em geral, eles terão tempos de execução mais rápidos e um menor número de comparações e trocas em comparação com o Bubble Sort, o Selection Sort e o Insertion Sort.

## 0.5 Conclusões

Com base nessas observações, podemos concluir que a escolha do algoritmo de ordenação depende do tamanho e das características do conjunto de dados, bem como das necessidades específicas da aplicação. Algoritmos mais eficientes, como o Merge Sort, Quick Sort e Tim Sort, são recomendados para conjuntos de dados grandes, enquanto algoritmos como o Insertion Sort podem ser adequados para conjuntos de dados menores ou parcialmente ordenados.

Alguns algoritmos, como o Tim Sort e o Merge Sort, possuem otimizações que podem lidar melhor com conjuntos de dados quase ordenados ou parcialmente ordenados em ordem decrescente.

Ao lidar com conjuntos de dados ordenados de forma ascendente, é recomendado considerar algoritmos adaptados para esse cenário ou aproveitar as características favoráveis de algoritmos como o Bubble Sort, Selection Sort e Insertion Sort.

A eficiência dos algoritmos de ordenação varia de acordo com o tamanho e as características do conjunto de dados. Algoritmos mais simples, como o Bubble Sort e o Selection Sort, são adequados para conjuntos de dados pequenos, enquanto algoritmos mais complexos, como o Merge Sort, Quick Sort e Tim Sort, têm melhor desempenho em conjuntos de dados maiores.

Algoritmos como o Bubble Sort e o Selection Sort têm um desempenho mais lento e exigem um maior número de comparações e trocas em comparação com outros algoritmos de ordenação. Portanto, eles são mais adequados para fins educacionais ou quando a simplicidade de implementação é mais importante do que a eficiência, que no caso chegaram a levar dias para completar a tarefa ocasionando até atraso nas outras tarefas.

Algoritmos como o Insertion Sort e o Shell Sort mostram um desempenho intermediário, sendo mais eficientes do que o Bubble Sort e o Selection Sort, mas ainda não tão rápidos quanto o Merge Sort, Quick Sort e Tim Sort.

O Merge Sort, Quick Sort e Tim Sort são algoritmos mais eficientes para a maioria dos cenários de ordenação. Eles têm tempos de execução mais rápidos e requerem um menor número de comparações e trocas em comparação com os algoritmos mencionados anteriormente.

A escolha do algoritmo de ordenação adequado depende das características específicas do conjunto de dados, como tamanho, distribuição e ordem inicial. É importante considerar esses fatores ao selecionar um algoritmo, bem como levar em conta outros aspectos, como estabilidade, requisitos de memória e facilidade de implementação.

Algoritmos adaptados para casos específicos, como o Insertion Sort em conjuntos de dados quase ordenados ou o Tim Sort em conjuntos de dados parcialmente ordenados, podem oferecer um desempenho ainda melhor em certos cenários.