UNIVERSIDADE FEDERAL DE ITAJUBÁ

CIÊNCIA DA COMPUTAÇÃO

# COMPARAÇÃO E BENCHMARK ENTRE OS ALGORÍTIMOS DE ORDENAÇÃO ESTUDADOS EM SALA E O COUNTING SORT

**Anna Beatryz Costa**

Algorítimos e Estruturas de Dados II
Bacharelado em Ciência da Computação

INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
UNIFEI - Itajubá

2025

Costa, Anna Beatryz
Comparação e Benchmark entre os algorítimos de ordenação estudados em sala e o Counting Sort / Anna Beatryz Costa. - Itajubá: UNIFEI, 2025.
7 p.
Algorítimos e Estruturas de Dados II (Ciência da Computação) - Universidade Federal de Itajubá, Itajubá, 2025
Orientação: Vanessa Souza
1. Algoritmos de ordenação. 2. Análise Experimental. 3. Comparação de Desempenho. 4. Count Sort. 5. Estruturas de dados. I. Souza, Vanessa, orient. II. Título.

**FOLHA DE APROVAÇÃO**

Costa, Anna Beatryz. **Comparação e Benchmark entre os algorítimos de ordenação estudados em sala e o Counting Sort**. 7 p. Algorítimos e Estruturas de Dados II (Ciência da Computação), Universidade Federal de Itajubá, Itajubá, 2025.

---

Vanessa Souza

UNIVERSIDADE FEDERAL DE ITAJUBÁ
INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

# Comparação e Benchmark entre os algorítimos de ordenação estudados em sala e o Counting Sort

Anna Beatryz Costa

*d2025007883@unifei.edu.br*

*Orientadora*: Vanessa Souza

**RESUMO**
Este trabalho surgiu a partir da premissa de aprofundar os conceitos estudados em sala sobre algoritmos de ordenação e análise de desempenho. Foram implementados três algoritmos — Insertion Sort, Quick Sort e Counting Sort — e, com eles, desenvolvemos um sistema de testes para avaliar suas eficiências em diferentes cenários. Também foi criada uma animação no terminal para visualização do processo de ordenação. Os dados aqui presentes foram organizados em tabelas e gráficos, permitindo uma análise comparativa entre os métodos.

**Palavras-chave:** Algoritmos de ordenação; Análise Experimental; Comparação de Desempenho; Count Sort ; Estruturas de dados

## 1 Introdução

A ordenação de dados é uma operação fundamental na ciência da computação, essencial para otimizar a busca e manipulação de informações em diversas aplicações. A escolha de um algoritmo de ordenação adequado pode impactar significativamente o desempenho de um sistema, visto que diferentes algoritmos apresentam eficiências distintas dependendo das características dos dados de entrada, como volume e ordem inicial. Compreender essas diferenças é crucial para o desenvolvimento de software eficiente.

Este trabalho tem como objetivo realizar uma análise de desempenho comparativa, focada no tempo médio de execução, de três algoritmos de ordenação interna implementados na linguagem C: Insertion Sort, Quick Sort (utilizando a estratégia de particionamento com o primeiro elemento como pivô) e Counting Sort. A análise foi realizada como parte das atividades da disciplina de Algoritmos e Estrutura de Dados II (CTCO02) da Universidade Federal de Itajubá. O estudo avalia como esses algoritmos se comportam ao ordenar conjuntos de dados com tamanhos variando de 100 a 32.400 elementos e com diferentes ordenações iniciais: aleatória, crescente, decrescente e quase ordenada (com 10

Para realizar a comparação, foi desenvolvido um sistema de benchmarking em C que gera os conjuntos de dados necessários, executa cada algoritmo sobre eles (utilizando a média de cinco execuções para entradas aleatórias e quase ordenadas para maior confiabilidade) e mede o tempo de execução utilizando a função clock() da biblioteca <time.h>. Este projeto, originalmente proposto como uma atividade em grupo, foi desenvolvido individualmente, com foco na análise do Counting Sort em comparação aos algoritmos clássicos Insertion Sort e Quick Sort.

Este relatório está estruturado da seguinte forma: a Seção 2 detalha o funcionamento, complexidade e características do algoritmo Counting Sort. A Seção 3 descreve a metodologia empregada na construção do sistema de benchmarking e na geração dos dados. A Seção 4 apresenta e discute os resultados obtidos na análise comparativa de tempo de execução. Por fim, a Seção 5 apresenta as considerações finais sobre o desempenho observado dos algoritmos e o desenvolvimento do trabalho.

## 2 Algoritmo Counting Sort

O Counting Sort (Ordenação por Contagem) é um algoritmo de ordenação linear que se distingue dos métodos de ordenação baseados em comparação, pois determina a posição ordenada dos elementos contando a frequência de cada valor dentro de um intervalo específico Backes (2023); w3s (2025). É particularmente eficiente para ordenar conjuntos de dados cujas chaves são inteiros dentro de um intervalo $[0, k]$ conhecido e relativamente pequeno.

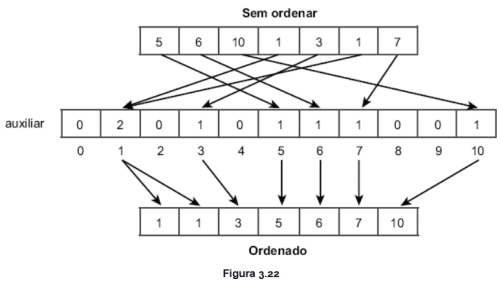


Figura 1: Ilustração do Funcionamento do Counting Sort. Retirado do livro Algoritmos e Estruturas de Dados em Linguagem C Backes (2023)

A complexidade de tempo do Counting Sort é determinada pela soma dos tempos de cada etapa. As etapas 1 e 4 levam tempo O(n), e as etapas 2 e 3 levam tempo O(k) e O(n) respectivamente (considerando o percurso dos vetores). Portanto, a complexidade total de tempo é O(n+k). A complexidade de espaço também é O(n+k), devido aos vetores auxiliares C (tamanho k+1) e B (tamanho n).

O Counting Sort apresenta vantagens significativas quando o intervalo k dos valores de entrada é da mesma ordem de grandeza ou menor que o número de elementos n, resultando em um tempo de execução linear (O(n)). No entanto, torna-se ineficiente em termos de espaço e potencialmente tempo (devido à inicialização e percurso do vetor C) se o intervalo k for muito maior que n. É importante notar que o Counting Sort, na sua forma apresentada, é um algoritmo estável, o que significa que elementos com o mesmo valor mantêm sua ordem relativa original no vetor ordenado. Por essas características, ele é frequentemente utilizado como sub-rotina em outros algoritmos de ordenação, como o Radix Sort.

## 2.1 Funcionamento Geral

O algoritmo opera em três etapas principais, utilizando um vetor auxiliar de contagem ‘C‘ de tamanho $k + 1$ e um vetor auxiliar de saída ‘B‘ de tamanho $n$ (onde $n$ é o tamanho do vetor de entrada ‘A‘), conforme descrito por Backes (2023, p. 27) Backes (2023):

1. **Contagem de Frequências:** O vetor ‘C‘ é inicializado com zeros. Em seguida, o vetor de entrada ‘A‘ é percorrido, e para cada elemento ‘A[i]‘, a contagem em ‘C[A[i]]‘ é incrementada. Ao final, ‘C[x]‘ armazena quantas vezes o valor $x$ ocorre em ‘A‘.

2. **Soma Acumulada:** O vetor ‘C‘ é modificado para que cada posição ‘C[x]‘ contenha o número de elementos menores ou iguais a $x$. Isso é feito acumulando as contagens: ‘C[x] = C[x] + C[x-1]‘ para $x$ de 1 até $k$. O valor ‘C[x]‘ agora representa a posição final do último elemento de valor $x$ no vetor ordenado.

3. **Distribuição Ordenada:** O vetor de saída ‘B‘ é preenchido percorrendo o vetor de entrada ‘A‘ da última para a primeira posição (garantindo estabilidade). Para cada ‘A[i]‘, o elemento é colocado em ‘B[C[A[i]]-1]‘. A contagem em ‘C[A[i]]‘ é então decrementada para posicionar corretamente o próximo elemento com o mesmo valor. Finalmente, o conteúdo do vetor auxiliar ‘B‘ é copiado de volta para o vetor original ‘A‘.

## 2.2 Implementação (Pseudocódigo)

O pseudocódigo apresentado na Figura 1 (adaptado de Backes (2023) Backes (2023)) ilustra os passos descritos.

## 2.3 Complexidade e Características

Conforme mencionado, a complexidade de tempo do Counting Sort é $O(n+k)$. Isso o torna extremamente eficiente, com tempo linear $O(n)$, quando $k$ (o intervalo dos valores) é da ordem de $n$ ou menor ($k = O(n)$) Backes (2023). A complexidade de espaço também é $O(n+k)$ devido aos vetores auxiliares ‘B‘ e ‘C‘.

Suas principais limitações são a necessidade de conhecer o valor máximo $k$ previamente e o alto consumo de memória se $k$ for muito grande comparado a $n$. O algoritmo é estável, preservando a ordem relativa de elementos com chaves iguais Backes (2023).

**Algoritmo 1:** Counting Sort (Adaptado de Backes (2023) Backes (2023))

**Dados:** Vetor $A[1..n]$, Vetor $B[1..n]$, inteiro $k$ (valor máximo em A)
**Resultado:** Vetor A ordenado (através do vetor B)

```
/* Passo 1: Inicializar vetor de contagem C */
para i ← 0 até k faça C[i] ← 0;
fim-para;
/* Passo 2a: Contar ocorrências */
para j ← 1 até n faça C[A[j]] ← C[A[j]] + 1;
fim-para;
/* Passo 2b: Calcular frequência acumulada */
para i ← 1 até k faça C[i] ← C[i] + C[i − 1];
fim-para;
/* Passo 3: Distribuir elementos em B (ordenado) */
para j ← n até 1 faça B[C[A[j]]] ← A[j];
C[A[j]] ← C[A[j]] − 1;
fim-para;
/* Passo 4: Copiar B para A (opcional, depende da implementação) */
para j ← 1 até n faça A[j] ← B[j];
fim-para;
```

## 3 Metodologia do Benchmarking

Para avaliar e comparar o desempenho dos algoritmos de ordenação propostos, foi desenvolvido um sistema de benchmarking utilising a linguagem C. O ambiente de desenvolvimento e teste consistiu em *[Mencione seu Sistema Operacional, ex: Windows 11]* utilizando o compilador `GCC` *[Se souber a versão, mencione]*. O foco da análise foi o tempo médio de execução para diferentes cenários de entrada.

## 3.1 Algoritmos Avaliados

Os seguintes algoritmos de ordenação interna foram implementados e avaliados neste estudo, baseados nos conceitos apresentados em aula Souza (2025):

- **Insertion Sort:** Implementação clássica do método de inserção direta.
- **Quick Sort:** Implementação recursiva utilizando a estratégia de particionamento onde o primeiro elemento do sub-vetor é selecionado como pivô, conforme exemplo adaptado das aulas teóricas Souza (2025).
- **Counting Sort:** Implementação do método de ordenação por contagem, adequado para chaves inteiras dentro de um intervalo conhecido.

As implementações focaram na corretude e na medição de desempenho. Os códigos-fonte dos algoritmos e do sistema de benchmarking estão disponíveis no repositório do projeto **?**.

## 3.2 Geração dos Conjuntos de Dados

Para simular diferentes cenários de uso real, foram gerados conjuntos de dados (vetores de inteiros) com características distintas, utilizando as funções implementadas no módulo `ES.c`:

- **Tamanhos (N):** Foram testados vetores com os seguintes tamanhos: 100, 1000, 10.000, 20.000, 25.000 e 32.400 elementos. Os valores dos elementos gerados estavam no intervalo compatível com os algoritmos, especialmente o Counting Sort (ex: $[0, N]$ ou $[0, k]$ se usou um k fixo).
- **Tipos de Entrada:** Para cada tamanho N, foram gerados quatro tipos de vetores:
  - *Aleatório:* Elementos gerados pseudo-aleatoriamente (usando `rand()` com sementes diferentes para cada repetição).
  - *Crescente:* Elementos pré-ordenados em ordem crescente (0, 1, 2,... N-1).
  - *Decrescente:* Elementos pré-ordenados em ordem decrescente (N, N-1,... 1).
  - *Quase Ordenado:* Um vetor inicialmente crescente onde 10

A geração utilizou funções como `geraAleatorios()`, `geraOrdenados()` e `geraQuaseOrdenados()`.

## 3.3 Medição de Desempenho

A métrica primária de desempenho avaliada foi o **Tempo Médio de Execução** em milissegundos (ms). A medição foi realizada utilizando a função `clock()` da biblioteca `<time.h>` da linguagem C, calculando a diferença entre o tempo registrado antes e depois da chamada da função de ordenação específica.

Para os tipos de entrada *Aleatório* e *Quase Ordenado*, o processo de geração do vetor e ordenação foi repetido 5 (cinco) vezes, utilizando sementes diferentes para a geração aleatória em cada repetição. O tempo final registrado para esses cenários foi a média aritmética dos tempos obtidos nessas cinco execuções, visando reduzir o impacto de flutuações momentâneas do sistema ou de conjuntos de dados aleatórios particularmente "fáceis"ou "difíceis". Para os tipos *Crescente* e *Decrescente*, que são determinísticos, apenas uma execução foi realizada por tamanho de vetor.

Os resultados consolidados, contendo o tempo médio para cada combinação de algoritmo, tamanho e tipo de entrada, foram armazenados no arquivo `benchmark_QIC.csv` para posterior análise e geração de gráficos.

# 4 Resultados e Discussão

Nesta seção, são apresentados e discutidos os resultados do benchmarking de tempo de execução dos algoritmos Insertion Sort, Quick Sort e Counting Sort. A análise compara o desempenho dos algoritmos para os diferentes tipos de entrada de dados e tamanhos de vetor (N) definidos na metodologia (Seção 3).

Inicialmente, as Figuras 2 e 3 fornecem uma visão geral comparativa do tempo de execução dos três algoritmos considerando todos os tipos de entrada agregados ou de forma combinada.

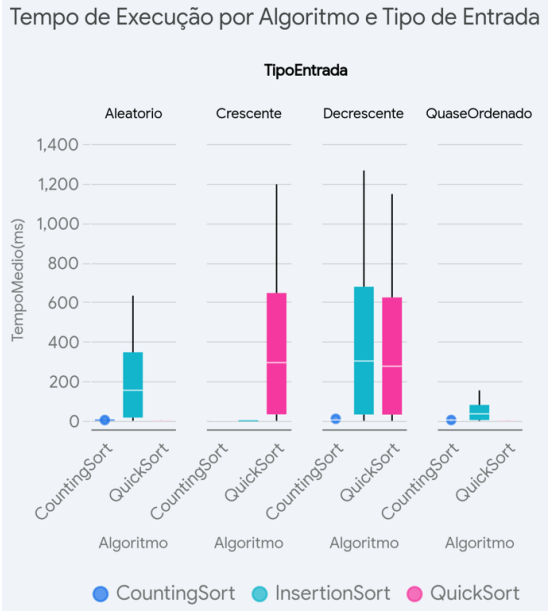


Figura 2: Visão Geral do Desempenho dos Algoritmos (1).


Figura 3: Visão Geral do Desempenho dos Algoritmos (2).

## 4.1 Desempenho Individual por Algoritmo

Antes de comparar os algoritmos lado a lado para cada tipo de entrada, as Figuras 4, 5 e 6 ilustram o comportamento individual de cada algoritmo sob as diferentes condições de ordenação inicial testadas.

Estes gráficos individuais reforçam as características de cada algoritmo: a eficiência do Insertion Sort para dados quase ordenados (Figura 4), a vulnerabilidade do Quick Sort a dados ordenados (Figura 5) e a estabilidade de desempenho do Counting Sort independentemente da ordem inicial (Figura 6).

## 4.2 Comparativo por Tipo de Entrada: Aleatório

A Figura 7 apresenta o comparativo de tempo de execução para vetores com elementos em ordem aleatória.

Observa-se que, para entradas aleatórias, o **Insertion Sort** apresenta um crescimento de tempo acentuado com o aumento de N, consistente com sua complexidade esperada de

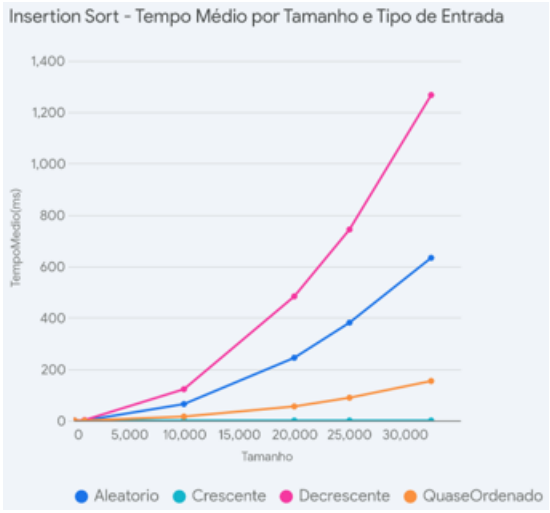


Figura 4: Desempenho do Insertion Sort para Diferentes Tipos de Entrada.

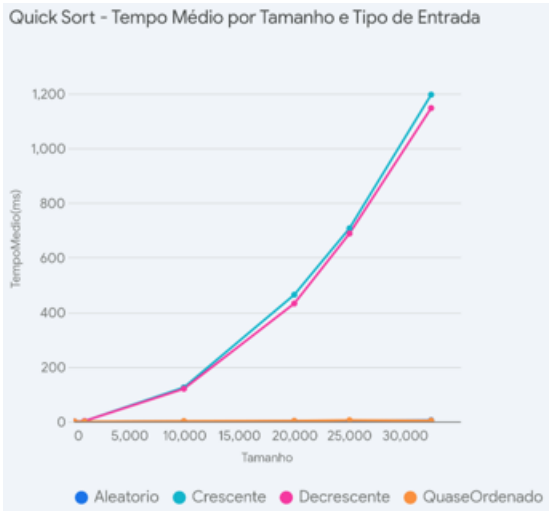


Figura 5: Desempenho do Quick Sort para Diferentes Tipos de Entrada.

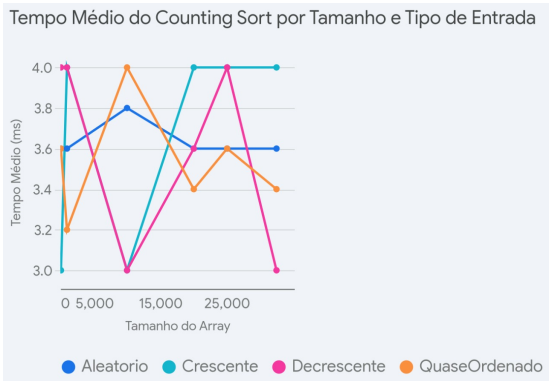


Figura 6: Desempenho do Counting Sort para Diferentes Tipos de Entrada.

$O(n^2)$ para o caso médio. Em contraste, tanto o **Quick Sort** quanto o **Counting Sort** demonstram desempenho significativamente superior. O Quick Sort reflete sua complexidade média eficiente de $O(n \log n)$, enquanto o Counting Sort exibe seu comportamento $O(n + k)$, ambos resultando em tempos

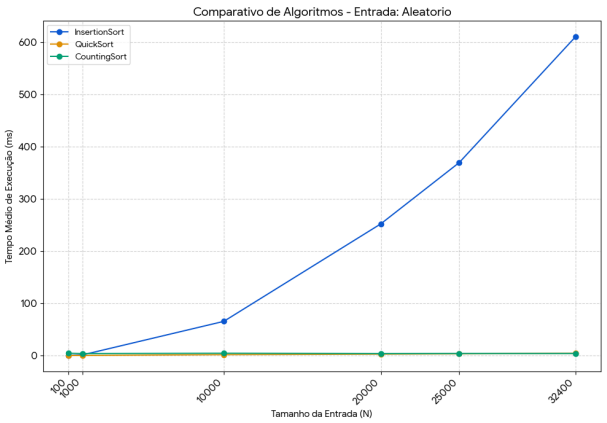


Figura 7: Comparativo de Tempo (ms) vs Tamanho (N) para Entrada Aleatória.

de execução muito baixos nesta escala para os tamanhos testados. Para N=32400, o Insertion Sort levou aproximadamente 610 ms, enquanto Quick Sort e Counting Sort levaram cerca de 4 ms.

## 4.3 Comparativo por Tipo de Entrada: Crescente

O desempenho dos algoritmos para entradas já ordenadas crescentemente é ilustrado na Figura 8.

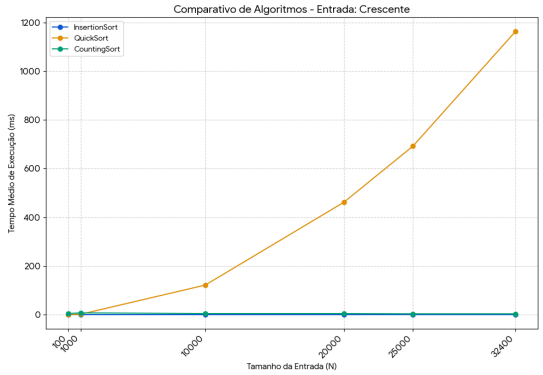


Figura 8: Comparativo de Tempo (ms) vs Tamanho (N) para Entrada Crescente.

Neste cenário, o **Insertion Sort** apresenta seu melhor caso, com complexidade $O(n)$, resultando em tempos de execução praticamente nulos (próximos a 0 ms) para todos os tamanhos testados, pois poucas comparações e nenhuma movimentação significativa são necessárias. O **Counting Sort** mantém sua eficiência $O(n+k)$, com tempos baixos e estáveis. Em contrapartida, o **Quick Sort**, com a estratégia de pivô implementada (primeiro elemento), exibe seu pior caso de complexidade $O(n^2)$. Isso ocorre porque o pivô (o menor elemento) gera partições completamente desbalanceadas a cada chamada recursiva, levando a um aumento drástico no tempo de execução, superando até mesmo o Insertion Sort (que está em seu melhor caso) para N maiores, chegando a mais de 1100 ms para N=32400 Souza (2025).

## 4.4 Comparativo por Tipo de Entrada: Decrescente

A Figura 9 mostra o comportamento dos algoritmos para entradas ordenadas de forma decrescente.

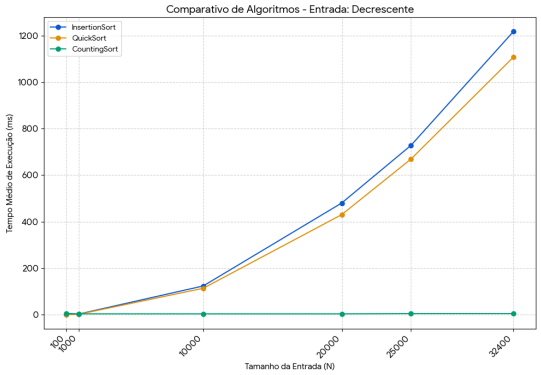


Figura 9: Comparativo de Tempo (ms) vs Tamanho (N) para Entrada Decrescente.

Este cenário representa o pior caso tanto para o **Insertion Sort** ($O(n^2)$) quanto para o **Quick Sort** ($O(n^2)$) com a estratégia de pivô utilizada. Ambos apresentam tempos de execução elevados e com crescimento quadrático, como pode ser observado no gráfico. O Quick Sort, novamente, sofre com o desbalanceamento das partições geradas pelo pivô (agora o maior elemento). O **Counting Sort**, por não ser baseado em comparações e depender apenas das frequências e do intervalo $k$, mantém seu desempenho eficiente $O(n + k)$, sendo ordens de magnitude mais rápido que os outros dois algoritmos neste cenário.

## 4.5 Comparativo por Tipo de Entrada: Quase Ordenado

Finalmente, a Figura 10 compara os algoritmos para dados quase ordenados (10

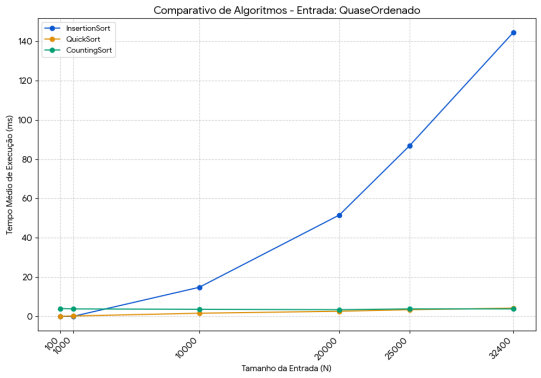


Figura 10: Comparativo de Tempo (ms) vs Tamanho (N) para Entrada Quase Ordenada (10% desordem).

Para dados quase ordenados, o **Insertion Sort** mostra um desempenho consideravelmente melhor do que no caso aleatório, aproximando-se de um comportamento linear para a porção ordenada, embora ainda apresente crescimento notável. O **Quick Sort** e o **Counting Sort** continuam muito eficientes, com tempos de execução baixos, refletindo suas complexidades $O(n \log n)$ e $O(n + k)$, respectivamente. O Quick Sort se beneficia do fato de que a entrada não é perfeitamente ordenada, evitando seu pior caso.

## 4.6 Discussão Geral

Os resultados experimentais corroboram as análises teóricas de complexidade para os algoritmos estudados. O Counting Sort demonstrou ser o algoritmo mais rápido em quase todos os cenários testados, evidenciando a vantagem de algoritmos não baseados em comparação quando as condições de entrada (inteiros em intervalo conhecido e razoável) são satisfeitas. Sua complexidade $O(n + k)$ o torna ideal para essas situações.

O Quick Sort confirmou sua reputação de ser muito eficiente em média (entradas aleatórias e quase ordenadas), com desempenho $O(n \log n)$. No entanto, sua sensibilidade à escolha do pivô e à ordenação inicial dos dados ficou clara nos testes com entradas crescentes e decrescentes, onde seu desempenho degradou para $O(n^2)$. Estratégias de pivoteamento mais robustas (como mediana de três ou pivô aleatório) poderiam mitigar esse pior caso, mas não foram objeto deste estudo.

O Insertion Sort, apesar de sua simplicidade e excelente desempenho em seu melhor caso ($O(n)$ para entradas crescentes), mostrou-se ineficiente ($O(n^2)$) para entradas maiores nos casos médio e pior, tornando-o inadequado para grandes volumes de dados não ordenados ou inversamente ordenados. Sua vantagem para dados quase ordenados foi observada, mas ainda assim foi superado por Quick Sort e Counting Sort nos tamanhos maiores testados.

## 5 Considerações Finais

Este trabalho apresentou uma análise comparativa de desempenho, focada no tempo de execução, entre os algoritmos de ordenação Insertion Sort, Quick Sort e Counting Sort, implementados em linguagem C. O objetivo foi avaliar o comportamento desses algoritmos sob diferentes condições de entrada, variando o tamanho do vetor e a ordem inicial dos elementos (aleatória, crescente, decrescente e quase ordenada).

Os resultados obtidos através do benchmarking experimental corroboraram, em grande parte, as expectativas baseadas na análise teórica de complexidade de cada algoritmo. O **Counting Sort** demonstrou ser consistentemente o algoritmo mais eficiente em termos de tempo de execução para quase todos os cenários testados, dada a natureza dos dados (inteiros em um intervalo razoável), confirmando a eficácia de sua complexidade $O(n + k)$. Sua performance manteve-se estável independentemente da ordem inicial dos dados.

O **Quick Sort**, implementado com a estratégia de pivô no primeiro elemento, apresentou excelente desempenho ($O(n \log n)$) para entradas aleatórias e quase ordenadas, sendo competitivo com o Counting Sort nesses casos. No entanto, sua vulnerabilidade ao pior caso ($O(n^2)$) foi claramente evidenciada em entradas ordenadas (crescente e decrescente), tornando-o significativamente mais lento nessas situações específicas.

O **Insertion Sort**, apesar de sua simplicidade e ótimo desempenho em seu melhor caso ($O(n)$ para entradas crescentes), mostrou-se o menos eficiente ($O(n^2)$) para entradas maiores nos casos médio (aleatório) e pior (decrescente), limitando

sua aplicabilidade prática a conjuntos de dados pequenos ou quase ordenados.

Com base nestas observações, pode-se recomendar o uso do Counting Sort quando se trabalha com chaves inteiras dentro de um intervalo conhecido e de tamanho gerenciável. Para dados genéricos e com distribuição mais aleatória, o Quick Sort continua sendo uma escolha robusta, embora seja prudente considerar estratégias de pivoteamento que mitiguem seu pior caso. O Insertion Sort pode ser útil em contextos específicos de dados quase ordenados ou como parte de algoritmos híbridos para subproblemas pequenos.

É importante notar algumas limitações deste estudo, como a faixa específica de tamanhos de vetor testada (até 32.400 elementos) e a dependência dos resultados em relação ao ambiente de hardware e software utilizado. Além disso, apenas uma variante do Quick Sort foi avaliada.

O desenvolvimento deste trabalho, realizado individualmente, proporcionou uma valiosa experiência prática na implementação, teste e análise de algoritmos de ordenação fundamentais, reforçando a compreensão da relação entre a teoria de complexidade e o desempenho observado na prática. A análise dos diferentes cenários de entrada permitiu visualizar concretamente as vantagens e desvantagens de cada abordagem de ordenação.

A implementação de um sistema de animação no terminal contribuiu para a compreensão visual dos passos de cada algoritmo, destacando-se como recurso didático complementar.

O código-fonte completo do projeto, incluindo todas as funções auxiliares, sistema de benchmarking e visualização animada, encontra-se disponível em: https://github.com/Cheeshiiree/Trabalho1_AEDII_Ordenacao_2025007883.git

## Referências

(2025). Dsa counting sort. W3Schools. Disponível em: https://www.w3schools.com/dsa/dsa_algo_countingsort.php. Acesso em: abr. 2025.

Backes, A. R. (2023). *Algoritmos e Estruturas de Dados em Linguagem C*. Rio de Janeiro: LTC. E-book. Acesso via Minha Biblioteca.

Souza, V. C. O. d. (2025). Aulas de algoritmos e estrutura de dados ii (ctco02). Material de Aula. Universidade Federal de Itajubá. Aulas 01 a 07.