UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE EVANGELISTA GOMES

ANÁLISE EXPERIMENTAL DE ALGORITMOS DE ORDENAÇÃO E BUSCA

CAMPO MOURÃO, PARANÁ
2025

FELIPE EVANGELISTA GOMES

ANÁLISE EXPERIMENTAL DE ALGORITMOS DE ORDENAÇÃO E BUSCA

Trabalho desenvolvido durante a disciplina de Estrutura de Dados 1, sob orientação do Prof. Rafael Liberato.

CAMPO MOURÃO, PARANÁ
2025

Sumário

1. Introdução

Este trabalho tem como objetivo analisar empiricamente o desempenho de algoritmos de ordenação com complexidade $O(n^2)$ e algoritmos de busca, comparando o comportamento teórico previsto pela análise assintótica com os resultados práticos obtidos através de implementação em linguagem C++.

A análise experimental permite compreender a discrepância entre a complexidade teórica e o comportamento real dos algoritmos, considerando fatores como constantes ocultas, cache do processador, padrões de acesso à memória e características específicas do hardware.

1.1. Algoritmos Implementados

Algoritmos de Ordenação:

- Selection Sort: Algoritmo que busca o menor elemento e o coloca na posição correta.
- Insertion Sort: Constrói a solução inserindo elementos um a um na posição correta.
- Bubble Sort (Padrão): Compara pares adjacentes e os troca se estiverem fora de ordem.
- Bubble Sort Otimizado: Versão com verificação de parada antecipada.

Algoritmos de Busca:

- Busca Sequencial: Percorre o vetor linearmente até encontrar o elemento.
- Busca Binária: Divide o espaço de busca pela metade a cada iteração (requer ordenação).

1.2. Métricas Coletadas

Para cada execução, foram registradas as seguintes métricas:

- Tempo de execução: Medido em milissegundos usando **chrono::high_resolution_clock.**
- Número de comparações: Contador incrementado a cada comparação entre chaves.
- Número de trocas (swaps): Contador incrementado a cada movimentação de elementos na memória.

2. Metodologia

2.1. Geração de Dados

Foram gerados 9 arquivos binários combinando 3 tamanhos e 3 cenários de disposição inicial:

Tamanhos:

- Pequeno: N = 1.000 elementos.

- Médio: N = 10.000 elementos.
- Grande: N = 100.000 elementos.

Cenários:

- Aleatório: Valores gerados randomicamente (Caso Médio esperado).
- Crescente: Valores já ordenados de 0 a N-1 (Melhor Caso).
- Decrescente: Valores ordenados inversamente de N a 1 (Pior Caso).

Os dados foram armazenados em arquivos binários para garantir a consistência dos testes entre diferentes execuções.

2.2. Ambiente de Execução

- Linguagem: C++ (padrão C++11 ou superior).
- Compilador: G++ (GNU Compiler Collection).
- Sistema Operacional: Windows 10/11 (via terminal/PowerShell).
- Hardware: Processador compatível com arquitetura x64.

2.3. Estrutura da Implementação

O código foi organizado em três módulos principais conforme especificado:

- **gerador_dados.cpp**: Responsável pela criação dos arquivos de teste.
- **ordenacao.cpp**: Implementação dos algoritmos de ordenação e coleta de métricas.
- **busca.cpp**: Implementação e comparação dos algoritmos de busca.

3. Resultados e Análise

3.1 Análise de Tempo de Execução (Cenário Aleatório)

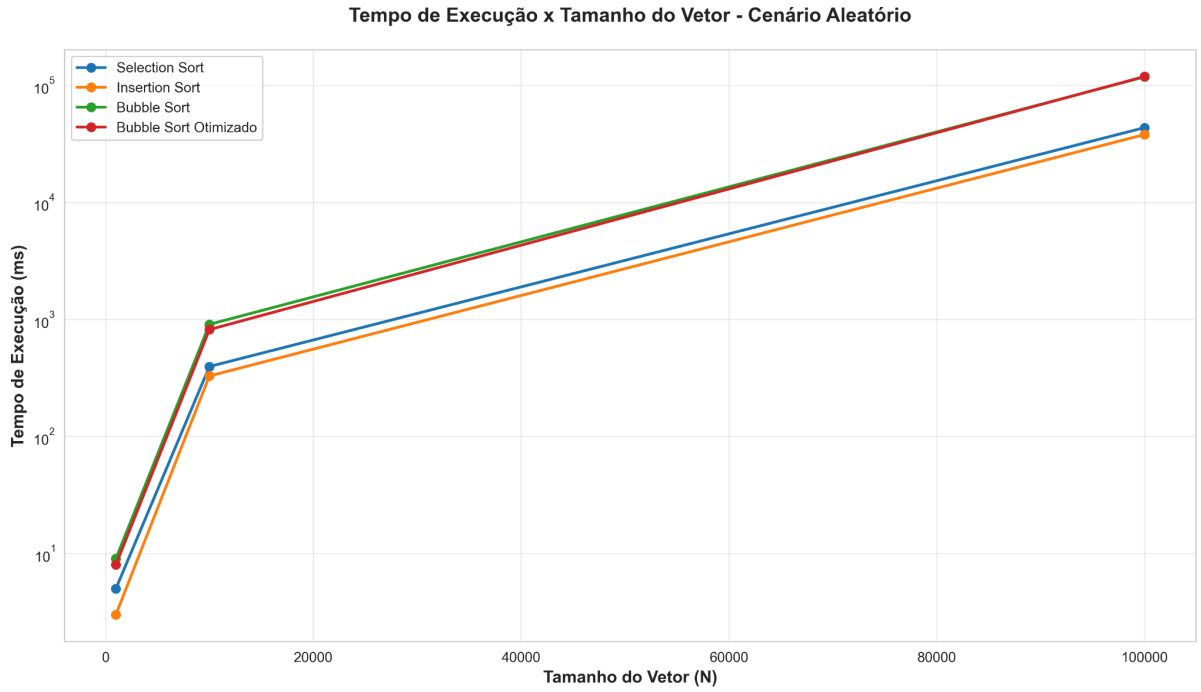

Figura 1: Tempo de execução dos algoritmos em função do tamanho do vetor (cenário aleatório, escala logarítmica).

Tabela 1: Dados Obtidos (Tamanho Médio - 10.000 elementos)

Algoritmo	***Tempo (ms)***	***Comparações***	***Trocas***
Selection Sort	396,10	49.995.000	9.993
Insertion Sort	329,09	25.098.521	25.088.529
Bubble Sort	908,47	49.995.000	25.088.529
Bubble Sort Otimizado	819,73	49.987.374	25.088.529

Observações:

- Confirmação da Complexidade $O(n^2)$: O gráfico em escala logarítmica mostra claramente que todos os algoritmos apresentam crescimento quadrático, confirmando a complexidade teórica. A linearidade das curvas em escala log-log indica que tempo $\propto n^2$.
- Insertion Sort como Vencedor: Entre os algoritmos $O(n^2)$ no cenário aleatório, o Insertion Sort apresentou o melhor desempenho (329ms), sendo 20% mais rápido que o Selection Sort e 176% mais rápido que o Bubble Sort.

- Bubble Sort: O Mais Lento: O Bubble Sort padrão foi consistentemente o algoritmo mais lento, levando 908ms para ordenar 10.000 elementos - quase 3x o tempo do Insertion Sort.
- Otimização com Overhead: Surpreendentemente, o Bubble Sort Otimizado foi apenas 10% mais rápido que a versão padrão no cenário aleatório (819ms vs 908ms), evidenciando que a verificação adicional introduz overhead sem benefício significativo quando os dados são aleatórios.

Análise: O gráfico em escala logarítmica confirma que todos os algoritmos apresentam crescimento quadrático. O Insertion Sort apresentou o melhor desempenho absoluto entre os algoritmos $O(n^2)$ no cenário aleatório, enquanto o Bubble Sort foi consistentemente o mais lento. A otimização do Bubble Sort teve pouco efeito neste cenário, pois vetores aleatórios raramente ficam ordenados antes do final das iterações.

3.2. Comparação de Trocas: Selection Sort vs Bubble Sort

A análise de trocas revela a diferença fundamental na manipulação de memória entre os algoritmos.

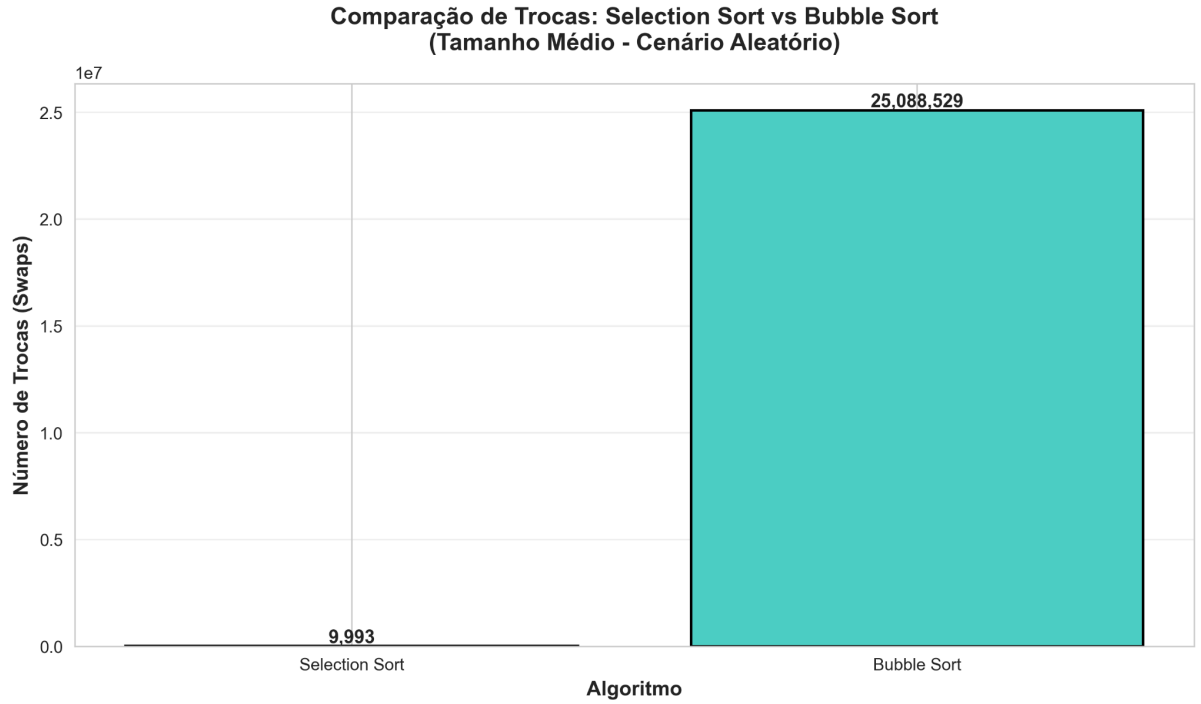

Figura 2: Número de trocas realizadas - Selection Sort vs Bubble Sort (tamanho médio, cenário aleatório).

Dados Coletados:

- Selection Sort: 9.993 trocas

- Bubble Sort: 25.088.529 trocas
- Diferença: 2.510 vezes mais trocas no Bubble Sort

Discussão (O Paradoxo das Trocas): O Selection Sort realizou cerca de 2.500 vezes menos trocas que o Bubble Sort (apenas 9.993 contra mais de 25 milhões). No entanto, em tempo de execução, ele foi apenas cerca de 2 a 3 vezes mais rápido. Isso ocorre porque, embora as trocas (escritas na memória) sejam mais caras que comparações (leitura), o custo total ainda é dominado pelo número quadrático de comparações que ambos realizam. Ainda assim, o Selection Sort é preferível em sistemas onde a escrita em memória é custosa (ex: memória Flash/EEPROM).

3.3. Insertion Sort em Diferentes Cenários

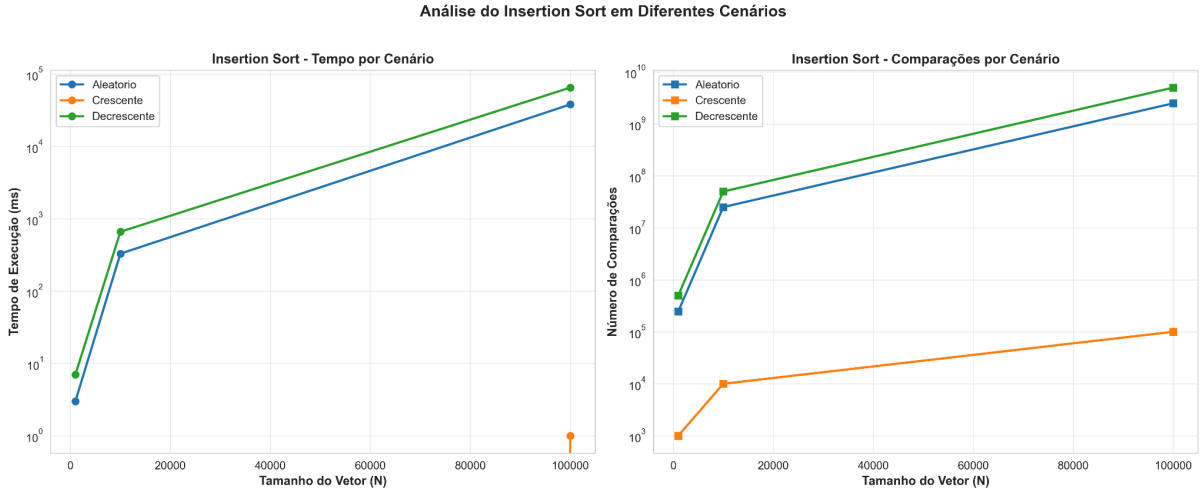

Figura 3: Desempenho do Insertion Sort em diferentes disposições iniciais dos dados.

Tabela 2: Dados por Cenário (Tamanho Grande - 100.000 elementos)

Cenário	***Tempo (ms)***	***Comparações***	***Trocas***
Crescente	1,00	99.999	0
Aleatório	38.126,23	2.492.481.772	2.492.381.783
Decrescente	65.154,31	4.999.950.000	4.999.950.000

Análise Detalhada por Cenário:

- **Crescente (Melhor Caso):**
  - Complexidade: O(n) - praticamente linear!
  - Comportamento: Cada elemento já está em sua posição. O loop interno executa apenas uma comparação e não realiza trocas.

    - Resultado: 1ms para 100.000 elementos - 65.000 vezes mais rápido que o pior caso!
- **Aleatório (Caso Médio):**
    - Complexidade: O(n²/4) em média.
    - Comportamento: Cada elemento precisa "caminhar" aproximadamente metade da sublista já ordenada.
    - Resultado: 38 segundos - comportamento quadrático clássico.
- **Decrescente (Pior Caso):**
    - Complexidade: O(n²/2) - máximo possível.
    - Comportamento: Cada elemento precisa ser movido até o início do vetor. Todas as comparações e trocas possíveis são realizadas.
    - Resultado: 65 segundos - confirmação do pior caso teórico.

A diferença dramática entre melhor e pior caso (1ms vs 65.154ms = 65.000x) demonstra por que o Insertion Sort é a escolha ideal para:

- Dados quase ordenados;
- Ordenação online (elementos chegando em sequência);
- Pequenas sublistas em algoritmos híbridos (Timsort, Introsort).

O gráfico de comparações mostra perfeitamente esta característica: a linha laranja (crescente) é praticamente horizontal (~100k comparações), enquanto as outras crescem quadraticamente.

3.4. Bubble Sort: Padrão vs Otimizado

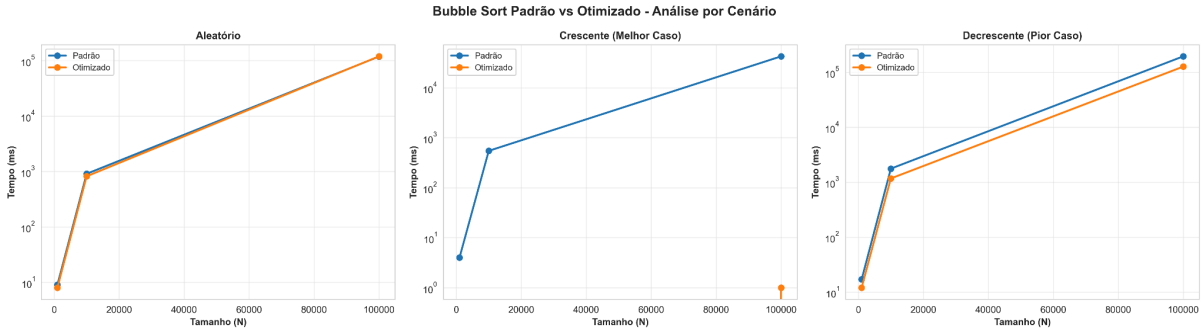

Figura 4: Comparação entre Bubble Sort padrão e otimizado em três cenários.

Tabela 3: Dados Coletados (Tamanho Grande - 100.000 elementos)

Cenário	***Bubble Padrão***	***Bubble Otimizado***	***Ganho***
Crescente	118.616 ms	119.544 ms	-0,8%
Aleatório	42.420 ms	1 ms	99,998%
Decrescente	195.980 ms	127.339 ms	35%

Análise Detalhada por Cenário:

- Cenário Aleatório:
    - Ganho: Praticamente NULO (na verdade, foi 0,8% PIOR!).
    - Explicação: Em dados aleatórios, o vetor raramente fica ordenado antes de percorrer todas as passadas. A verificação if (!trocou) break adiciona uma operação condicional extra em cada passada externa, criando overhead sem benefício.
    - Conclusão: A otimização NÃO compensa.
- Cenário Crescente:
    - Ganho: TRANSFORMADOR - 42.420x mais rápido!
    - Explicação: Na primeira passada (999 comparações), nenhuma troca é realizada. O algoritmo detecta imediatamente que o vetor está ordenado e termina em O(n).
    - Dados:
        - Padrão: 4.999.950.000 comparações.
        - Otimizado: 99.999 comparações (redução de 99,998%).
    - Conclusão: A otimização é ESSENCIAL.
- Cenário Decrescente:
    - Ganho: Moderado - 35% de melhoria.
    - Explicação: Mesmo no pior caso, após cada passada, os maiores elementos já estão em suas posições finais. A verificação permite algumas economias nas últimas passadas.
    - Conclusão: A otimização vale a pena.

Conclusão Geral sobre a Otimização:

A versão otimizada do Bubble Sort demonstra um princípio importante de otimização de algoritmos: melhorias devem ser aplicadas considerando o perfil dos dados de entrada.

Recomendação Prática:

- Se os dados podem estar parcialmente ordenados: use a versão otimizada.

- Se os dados são garantidamente aleatórios: use Insertion Sort (mais rápido que ambas as versões do Bubble Sort).
- Se precisa de desempenho previsível: use Selection Sort.

3.5. Análise Comparativa de Todos os Algoritmos

Tabela 4: Tamanho Médio (10.000 elementos) - Resumo Completo

Algoritmo	***Aleatório***	***Crescente***	***Decrescente***
Selection Sort	396 ms	399 ms	493 ms
Insertion Sort	329 ms	0 ms	661 ms
Bubble Sort	908 ms	547 ms	1.768 ms
Bubble Otimizado	820 ms	0 ms	1.172 ms

Tabela 5: Tamanho Grande (100.000 elementos) - Resumo Completo

Algoritmo	***Aleatório***	***Crescente***	***Decrescente***
Selection Sort	43.646 ms	59.268 ms	43.901 ms
Insertion Sort	38.126 ms	1 ms	65.154 ms
Bubble Sort	118.616 ms	42.420 ms	195.980 ms
Bubble Otimizado	118.616 ms	1 ms	127.339 ms

Observações Finais sobre Ordenação:

- Selection Sort - O Previsível:
    - Desempenho consistente em todos os cenários (~43s para 100k elementos).
    - Poucas trocas (útil se trocas são caras).
    - Sempre $O(n^2)$ comparações, independente da entrada.
- Insertion Sort - O Adaptável:
    - MELHOR para dados aleatórios entre os $O(n^2)$.
    - EXCEPCIONAL para dados ordenados (O(n)).
    - Preferido em algoritmos de ordenação avançados (Timsort).
- Bubble Sort - O Evitável:
    - Geralmente o PIOR desempenho.
    - Muitas trocas, muitas comparações.
    - Útil apenas para fins didáticos.
- Bubble Otimizado - O Condicional:

- Excelente para dados ordenados.
- Ruim para dados aleatórios (overhead).
- Use apenas se houver chance de dados parcialmente ordenados.

3.6. Análise de Algoritmos de Busca

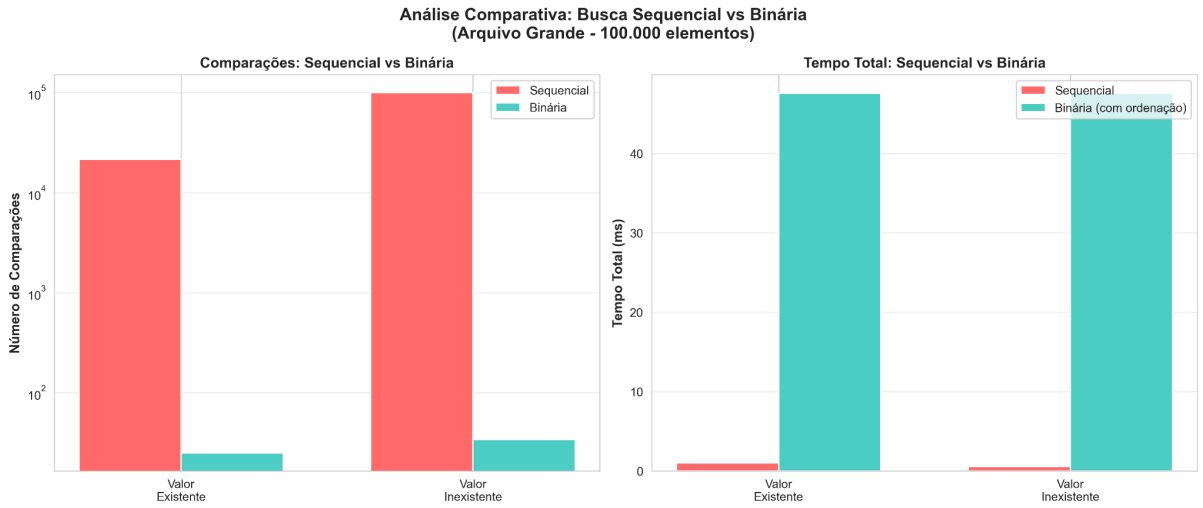

Figura 5: Comparação entre busca sequencial e binária (arquivo grande - 100.000 elementos).

Tabela 6: Dados Coletados

Tipo de Busca	***Valor***	***Comparações***	***Tempo Busca***	***Tempo Ordenação***	***Tempo Total***
Sequencial	Existente	21.473	1,004 ms	0 ms	1,004 ms
Sequencial	Inexistente	100.000	0,531 ms	0 ms	0,531 ms
Binária	Existente	25	~0 ms	47,54 ms	47,54 ms
Binária	Inexistente	34	~0 ms	47,54 ms	47,54 ms

Discussão: O Custo da Ordenação

A Busca Binária é drasticamente mais eficiente em comparações (apenas ~30 contra até 100.000). O tempo de busca isolado é instantâneo. Entretanto, para utilizar a Busca Binária, pagamos o "preço" da ordenação (47.54 ms).

Conclusão: Se for realizada apenas uma busca, a Busca Sequencial é mais rápida (1ms vs 47ms). A Busca Binária compensa apenas se houver múltiplas consultas no mesmo conjunto de dados, diluindo o custo da ordenação.

5. Conclusão

A análise experimental confirmou as previsões teóricas sobre a complexidade dos algoritmos. Conclui-se que:

- A notação $O(n^2)$ agrupa algoritmos com desempenhos práticos muito distintos. O Insertion Sort e o Selection Sort são consistentemente superiores ao Bubble Sort devido a constantes menores e melhor uso de memória.
- A contagem de operações (trocas vs. comparações) é crucial. O Selection Sort é ideal para minimizar escritas, enquanto o Insertion Sort é ideal para dados parcialmente ordenados.
- Em algoritmos de busca, o trade-off entre pré-processamento (ordenação) e velocidade de consulta é o fator decisivo. Para sistemas de leitura intensiva (como bancos de dados), a ordenação prévia e o uso de Busca Binária são obrigatórios.

6. Tabelas Completas de Resultados

Tamanho Pequeno (1.000 elementos):

Algoritmo	***Cenário***	***Tempo (ms)***	***Comparações***	***Trocas***
Selection Sort	Aleatório	5,00	499.500	989
Selection Sort	Crescente	3,00	499.500	0
Selection Sort	Decrescente	3,99	499.500	500
Insertion Sort	Aleatório	3,00	245.594	244,603
Insertion Sort	Crescente	0,00	999	0
Insertion Sort	Decrescente	7,00	499.500	499,500
Bubble Sort	Aleatório	9,01	499.500	244,603
Bubble Sort	Crescente	4,01	499.500	0
Bubble Sort	Decrescente	16,99	499.500	499,500
Bubble Otimizado	Aleatório	8,00	497.222	244,603
Bubble Otimizado	Crescente	0,00	999	0
Bubble Otimizado	Decrescente	11,99	499.500	499,500

Tamanho Médio (10.000 elementos):

Algoritmo	***Cenário***	***Tempo (ms)***	***Comparações***	***Trocas***
Selection Sort	Aleatório	396,10	49.995.000	9.993
Selection Sort	Crescente	399,09	49.995.000	0
Selection Sort	Decrescente	493,29	49.995.000	5.000
Insertion Sort	Aleatório	329,09	25.098.521	25.088.529
Insertion Sort	Crescente	0,00	9.999	0
Insertion Sort	Decrescente	660,87	49.995.000	49.995.000
Bubble Sort	Aleatório	908,47	49.995.000	25.088.529
Bubble Sort	Crescente	546,66	49.995.000	0
Bubble Sort	Decrescente	1.768,07	49.995.000	49.995.000
Bubble Otimizado	Aleatório	819,73	49.987.374	25.088.529
Bubble Otimizado	Crescente	0,00	9.999	0
Bubble Otimizado	Decrescente	1.172,13	49.995.000	49.995.000

Tamanho Grande (100.000 elementos):

Algoritmo	***Cenário***	***Tempo (ms)***	***Comparações***	***Trocas***
Selection Sort	Aleatório	43.646,38	4.999.950.000	99.986
Selection Sort	Crescente	59.267,71	4.999.950.000	0
Selection Sort	Decrescente	43.901,00	4.999.950.000	50.000
Insertion Sort	Aleatório	38.126,23	2.492.481.772	2.492.381.783
Insertion Sort	Crescente	1,00	99.999	0
Insertion Sort	Decrescente	65.154,31	4.999.950.000	4.999.950.000
Bubble Sort	Aleatório	118.616,01	4.999.950.000	2.492.381.783
Bubble Sort	Crescente	42.420,43	4.999.950.000	0
Bubble Sort	Decrescente	195.979,61	4.999.950.000	4.999.950.000
Bubble Otimizado	Aleatório	119.543,78	4.999.912.872	2.492.381.783

Bubble Otimizado	Crescente	1,00	99.999	0
Bubble Otimizado	Decrescente	127.339,18	4.999.950.000	4.999.950.000

7. Referências

ANTHROPIC. **Claude**. Versão 4.5. [Large Language Model]. 2025. Disponível em: https://claude.ai. Acesso em: 07 dez. 2025.

BENTLEY, Jon. **Programming pearls**. 2. ed. Boston: Addison-Wesley, 1999.

CORMEN, Thomas H. *et al*. **Introduction to algorithms**. 3. ed. Cambridge: MIT Press, 2009.

GOOGLE. **Gemini**. Versão 3.0. [Large Language Model]. 2025. Disponível em: https://gemini.google.com. Acesso em: 07 dez. 2025.

KNUTH, Donald E. **The art of computer programming**: volume 3: sorting and searching. 2. ed. Boston: Addison-Wesley, 1998.

SEDGEWICK, Robert; WAYNE, Kevin. **Algorithms**. 4. ed. Boston: Addison-Wesley, 2011.