1. Introdução
Este relatório apresenta uma análise experimental de diversos algoritmos de ordenação clássicos, implementados em linguagem C++.
O objetivo é comparar o desempenho prático de cada algoritmo com sua análise assintótica teórica, considerando diferentes tipos de entrada:
vetores com valores aleatórios, em ordem crescente e em ordem decrescente.

Os algoritmos analisados foram: Bubble Sort, Bubble Sort Otimizado, Insertion Sort, Selection Sort, Shell Sort, Heap Sort, Merge Sort,
Quick Sort (pivô sendo o primeiro elemento) e Quick Sort (pivô sendo o elemento central).
Os tempos de execução foram medidos em microssegundos e os resultados apresentados em gráficos e tabelas.

2. Metodologia
Cada algoritmo foi implementado em C++ e testado com vetores de diferentes tamanhos, fazendo 10 repetições do algoritmo. Tamanhos usados:
1000, 5000, 10000, 15000, 20000, 25000, 50000 e 100000 elementos.

Três tipos de entrada foram utilizados:
- Aleatória: números inteiros gerados de forma pseudoaleatória;
- Crescente: números ordenados em ordem ascendente;
- Decrescente: números ordenados em ordem descendente.

Os tempos de execução foram obtidos utilizando medições com alta resolução.
Após a coleta dos dados, os resultados foram processados e os gráficos gerados em Python, utilizando a biblioteca Matplotlib.

3. Análise Teórica (Complexidade Assintótica)

Tabela 1 – Complexidade dos algoritmos:

Algoritmo	Melhor Caso	Caso Médio	Pior Caso
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Bubble Sort Otimizado	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Shell Sort	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n^2)$

Heap Sort	O(n log n)	O(n log n)	O(n log n)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort (pivô 1º)	O(n log n)	O(n log n)	$O(n^2)$
Quick Sort (pivô central)	O(n log n)	O(n log n)	O(n log n)

Tabela 2 – Tempos médios de execução (em microssegundos):

Algoritmo	1000	5000	10000	25000	50000	100000
Bubble Sort	6935	241922	898043	5547000	22370000	89750000
Bubble Sort Otimizado	5801	161507	601518	3616400	16032333	60251067
Insertion Sort	2334	52011	208447	1259863	5333853	21901033
Selection Sort	3268	78706	283312	1875106	6905923	27272500
Shell Sort	167	1034	2534	6975	14370	31845
Heap Sort	333	1986	4234	10719	22134	47910
Merge Sort	537	3655	7676	16985	31743	63908
Quick Sort (pivô 1º)	1034	27045	111511	708817	2725524	10621121
Quick Sort (pivô central)	33	641	1187	2921	6246	14211

Tabela 3 – Tempos médios de execução (em segundos):

Algoritmo	1000	5000	10000	25000	50000	100000
Bubble Sort	0.01	0.24	0.90	5.55	22.37	89.75

Bubble Sort Otimizado	0.01	0.16	0.60	3.62	16.03	60.25
Insertion Sort	0.00	0.05	0.21	1.26	5.33	21.90
Selection Sort	0.00	0.08	0.28	1.88	6.91	27.27
Shell Sort	0.00	0.00	0.00	0.01	0.01	0.03
Heap Sort	0.00	0.00	0.00	0.01	0.02	0.05
Merge Sort	0.00	0.00	0.01	0.02	0.03	0.06
Quick Sort (pivô 1º)	0.00	0.03	0.11	0.71	2.73	10.62
Quick Sort (pivô central)	0.00	0.00	0.00	0.00	0.01	0.01

4. Resultados Experimentais

Os tempos médios observados mostraram comportamento coerente com a análise teórica.
Os algoritmos quadráticos (Bubble, Insertion e Selection Sort) apresentaram crescimento acentuado, enquanto os algoritmos O(n log n) escalaram melhor.

5. Comparação Teórica × Experimental (detalhada)

Os resultados experimentais foram coerentes com a análise assintótica conhecida para os algoritmos testados. A seguir são feitas observações por família/algoritmo, relacionando o comportamento prático com o comportamento assintótico.

Algoritmos quadráticos (Bubble, Bubble otimizado, Insertion, Selection)

- Todos esses algoritmos mostraram crescimento de tempo muito mais acentuado quando n aumentou, o que confirma a natureza $O(n^2)$esperada teoricamente.
- Em particular, Bubble Sort (versão sem otimização) e Selection Sort já apresentavam tempos elevados mesmo para tamanhos médios (5k–10k), tornando-os impraticáveis para entradas maiores (50k e 100k).

- Bubble Sort otimizado melhorou substancialmente no caso crescente (melhor caso $O(n)$), mas continua proibitivo para entradas grandes e permutações aleatórias.
- Insertion Sort apresentou bom desempenho no caso crescente (quase imediato), porém degrada para tempos extremamente altos no caso decrescente — exatamente como a análise teórica prevê (melhor caso $O(n)$, pior caso $O(n^2)$).

Quick Sort (pivô = primeiro elemento) vs Quick Sort (pivô = central)

- O Quick Sort com pivô central foi consistentemente um dos mais rápidos nos testes práticos, com tempos muito baixos e escalabilidade próxima de $O(n\log\ n)$experimental — este comportamento é esperado quando a escolha de pivô evita partições muito desbalanceadas.
- Já o Quick Sort com pivô fixo (primeiro elemento) apresentou degradação severa para vetores já ordenados (crescente/decrescente), aproximando-se do pior caso $O(n^2)$. Isso aparece claramente nos dados experimentais: tempos muito altos nos casos ordenados, confirmando a fragilidade dessa escolha de pivô.

Algoritmos $O(n\log\ n)$estáveis (Merge Sort, Heap Sort)

- Merge Sort e Heap Sort mostraram comportamento previsível e escalável, com crescimento suave dos tempos à medida que n aumentou.
- Ambos são robustos a entradas ordenadas ou parcialmente ordenadas (não sofrem degradação a $O(n^2)$), o que corresponde à análise teórica. Merge Sort tende a usar memória adicional (por causa das cópias), enquanto Heap Sort tem vantagem de ser in-place — escolha entre eles depende de restrições de memória e estabilidade (Merge é estável, Heap não é).

Shell Sort

- O Shell Sort apresentou desempenho intermediário: bem melhor que os quadráticos para entradas maiores, mas geralmente inferior a Merge/Heap/Quick com pivô central. Isso condiz com a teoria, já que a complexidade do Shell depende da sequência de gaps; práticas comuns dão algo entre $O(n\log\ n)$e $O(n^{1.25})$em média, mas sem garantia firme como Merge/Heap.

Comparação global

- Para entradas grandes (≥ 50k), os vencedores práticos foram Quick Sort (pivô central), Merge Sort e Heap Sort, nessa ordem de eficiência nos nossos testes.
- Algoritmos $O(n^2)$(Bubble, Selection, Insertion) se tornaram rapidamente impraticáveis conforme n cresceu, concordando com a análise teórica.

- O comportamento observado em cada caso (aleatório, crescente, decrescente) confirma as hipóteses teóricas sobre melhor/médio/pior caso de cada algoritmo: quando o algoritmo tem melhor caso linear (ex.: Insertion, Bubble otimizado) isso aparece claramente em entradas já ordenadas; quando o algoritmo tem pior caso quadrático em entradas ordenadas (Quick com pivô fixo), o experimento também mostrou isso.

Limitações e fontes de ruído experimental

- As medições podem sofrer ruído do sistema operacional, cache, alocação de memória e outros processos concorrentes.
- Pequenas diferenças entre algoritmos com complexidade similar podem depender de detalhes de implementação, otimizações do compilador e do hardware (CPU, cache, memória).
- Recomendação: repetir cada experimento várias vezes (já feito) e usar a média/mediana para diminuir ruído. Para comparações finais, também é útil apresentar barras de erro (desvio padrão) nos gráficos.

---

6. Conclusão (recomendações e fechamento)

Este estudo experimental confirmou que a complexidade assintótica é um excelente guia para prever o desempenho prático dos algoritmos de ordenação. Principais conclusões:

1. Algoritmos $O(n\log\ n)$(Quick — com boa escolha de pivô, Merge e Heap) são as melhores opções para entradas grandes. Entre eles, o Quick Sort com pivô central obteve o melhor desempenho prático em nossos testes, seguido por Merge e Heap.
2. Quick Sort com pivô fixo (primeiro elemento) deve ser evitado para entradas que possam estar ordenadas ou quase ordenadas, pois seu pior caso é $O(n^2)$e isso foi confirmado experimentalmente. Se for usar Quick Sort, usar seleção de pivô mais robusta (pivô central, pivot randomizado, ou mediana-de-tres) é recomendado.
3. Algoritmos quadráticos (Bubble, Selection, Insertion) só são indicados para conjuntos pequenos (ex.: n $\lesssim$ 1.000) ou para fins didáticos. O Insertion Sort pode ser útil como algoritmo final em hybrid sorts (ex.: usar insertion em subarrays pequenos em Quick/Merge) devido ao seu baixo overhead em tamanhos pequenos.
4. Shell Sort é uma escolha razoável quando se busca algo simples e mais eficiente que quadráticos, porém não tão robusto ou previsível quanto Merge/Heap/Quick.

5. Recomendações práticas:
    - Para uso geral em aplicações que exigem alto desempenho: Quick Sort (com pivô robusto) ou Merge Sort (se estabilidade for necessária).
    - Para restrições de memória: Heap Sort (in-place).
    - Para dados pequenos ou parcialmente ordenados: Insertion Sort como fallback.