UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
IMD0029 - ESTRUTURA DE DADOS BÁSICAS 1
PROF.º JOÃO GUILHERME

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO

SABRINA DA SILVA BARBOSA

NATAL
JUL/2025

INTRODUÇÃO

Este trabalho tem por objetivo expor e analisar o comportamento dos algoritmos de ordenação: Selection Sort, Insertion Sort, Bubble Sort, Quick Sort, Merge Sort e STL (Standard Template Library) – Sort, para a distribuição de dados: aleatórios, quase ordenados (sendo 10% desordenado) e inversamente ordenados, utilizando as seguintes métricas como base de comparações: tempo gasto, quantidade de comparações e de trocas realizadas. Salientando que a coleta dessas informações foi feita em um espaço amostral de dados que vai de 1.000 até 100.000, crescendo em uma escala de 10.000 para o tamanho do vetor. Não obstante, para o STL Sort foi usada apenas a métrica de tempo para comparação.

Nesse sentindo, sabendo que um computador, no geral, funciona à base do registro de informações e, segundo Cormen et al. (2012, p. 131 - PDF), os números que devem ser ordenados raramente são valores isolados, se faz importante entender que, em geral, cada um desses números faz parte de uma coleção de dados denominada registro. Nesse sentido, cada registro contém uma chave, que é o valor a ser ordenado. O restante do registro consiste em dados satélites, que normalmente são transportados junto com a chave. Isso significa que, quando um algoritmo de ordenação permuta as chaves, também deve permutar os dados satélites. Dessa forma, se esses registros incluem uma grande quantidade de dados satélites, muitas vezes será necessário permutar um arranjo de ponteiros para os registros em vez dos próprios registros, minimizando a movimentação de dados.

Portanto, aprender diferentes abordagens para lidar com diversos problemas reais, que vão desde conseguir mapear um funcionário para realizar o pagamento de seu trabalho até conseguir otimizar um algoritmo melhorando seu funcionamento (Cormen et al., 2012), são de extrema importância, especialmente para compreender qual o melhor cenário para aplicar determinadas técnicas de programação. Diante disso, este trabalho surge com esse propósito de identificar as diferenças dos algoritmos de ordenação para conhecer e compreender a importância de seu uso, aprendendo a utilizá-los nos cenários adequados a cada um deles.

FUNDAMENTAÇÃO TEÓRICA

SELECTION SORT

O algoritmo de ordenação por seleção, Selection Sort, é um dos algoritmos mais simples e funciona da seguinte forma:

- Primeiro encontra o menor elemento do array;
- Em seguida, troca de posição com o primeiro elemento do array;
- Repete o procedimento até que todo o array esteja ordenado.

Este algoritmo é chamado de Selection Sort por selecionar várias vezes o menor elemento restante. Possuindo complexidade assintótica N^2 no melhor, médio e pior caso (Sedgewick, 2010, p. 94 ~ 95).

INSERTION SORT

O algoritmo de ordenação Insertion Sort é um algoritmo tão simples quanto o Selection Sort e funciona da seguinte forma:

- Toma o valor da primeira posição como já ordenado;
- Em seguida, compara com o segundo elemento do vetor;
- Para ordenar, "abre" um espaço no vetor onde o valor em questão precisa ser inserido, movendo os elementos maiores e já ordenados uma posição à direita;
- Não realiza necessariamente uma troca direta. Apenas desloca as posições;
- considera os elementos um de cada vez, inserindo cada um em seu devido lugar entre os já considerados (mantendo-os ordenados).

Este algoritmo é chamado de Insertion Sort, porque funciona inserindo o elemento não ordenado na "lista" de elementos já ordenados. Também possui complexidade assintótica N^2, porém no melhor caso tem complexidade assintótica linear N (Sedgewick, 2010, p. 95 ~ 96).

BUBBLE SORT

O algoritmo de ordenação Bubble Sort é um algoritmo muito básico e fácil de entender. Ele é frequentemente ensinado em aulas introdutórias, fazendo uma ordenação por “bolhas”. Funciona da seguinte forma:

- Percorre o vetor de dados, trocando elementos adjacentes, se necessário;
- Pega elementos par a par e compara;
- Quando nenhuma troca for necessária em alguma passagem, o vetor estará ordenado.

É preciso refletir um pouco para se convencer, primeiro, de que isso funciona e, em segundo, de que o tempo de execução é quadrático. Nesse sentido, não está claro o motivo pelo qual tal método é ensinado com tanta frequência, já que a ordenação por inserção (Insertion Sort) parece mais simples e eficiente em quase todos os aspectos. Vale salientar ainda que o loop interno da ordenação por bolhas (Bubble Sort) tem cerca de duas vezes mais instruções do que o Insertion Sort ou o Selection Sort (Sedgewick, 2010, p. 99). Semelhante ao Insertion Sort, possui complexidade assintótica N^2 e, no melhor caso, tem complexidade assintótica linear N.

QUICK SORT

O algoritmo de ordenação Quick Sort é o algoritmo que provavelmente é o mais utilizado do que todos os outros. Ele foi inventado em 1960 por C. A. R. Hoare e tem sido estudado por muitas pessoas desde então. É popular por não ser difícil de implementar, é bom para "propósito geral" (funciona bem em diversas situações) e consome menos recursos do que qualquer outro método de ordenação em muitas situações (Sedgewick, 2010, p. 103). Funciona da seguinte forma:

- É selecionado um pivô;
- Divide-se o vetor com base nesse pivô;
- Ordena cada parte comparando os elementos do vetor com o pivô escolhido para cada pedaço.

É um algoritmo que usa do método de dividir para conquistar, sendo naturalmente recursivo. Todavia, a depender da escolha do pivô, por ser recursivo, pode acabar não sendo muito performático, tendo complexidade assintótica quadrática N^2 no pior caso.

Não obstante, uma versão cuidadosamente ajustada desse algoritmo, provavelmente será executada significativamente mais rápido do que qualquer outro método de classificação na maioria dos computadores fazendo jus ao seu nome (Sedgewick, 2010, p. 104). Dessa forma, possui complexidade assintótica N Log N no melhor e médio caso.

MERGE SORT

O algoritmo de ordenação Merge Sort é um algoritmo que se caracteriza por fazer parte de um grupo de ordenação que é classificada como *externa*, por lidar com o processamento de arquivos muito grandes, grandes demais para caber na memória primária de qualquer computador (Sedgewick, 2010, p. 156). Funciona da seguinte forma:

- Divide o vetor ao meio;
- Realiza as divisões sussecivas até que se obtenha um vetor unitário (tem apenas um elemento);
- Escolhe um lado e ordena "voltando" na pilha de pedaços, comparando os valores de cada pedaço de vetor e ordena os dados em um único vetor fazendo o "merge" deles.
- Faz os mesmos passos para o lado que sobra.

Diante disso, por ser um algoritmo que necessita de um vetor auxiliar para realizar a ordenação, acaba consumindo mais memória do que os outros algoritmos, tendo uma complexidade N para o espaço de memória. Entretanto, tirando esse detalhe, é um algoritmo estável e possui complexidade assintótica N Log N para todos os casos de uso.

METODOLOGIA

Para a comparação dos algoritmos de ordenação e a coleta dos dados, utilizou-se vetores de inteiros (*int)*, porém no arquivo *hpp* referente à quantidade de trocas e comparações, localizado no seguinte caminho includes/qtdComparTroca.hpp do repositório, foi utilizado um artifício que permite gerar dados de diferentes tipos, basta alterar de *int* para o tipo desejado e será possível comparar os algoritmos em diferentes tipos de dados, desde que seja um tipo passível de ordenamento.

Além disso, o espaço amostral de dados foi feito para vetores de 1.000 até 100.000 elementos, sendo preenchidos segundo uma distribuição *aleatória*, usando a função *srand* da *stdlib* padrão, *quase ordenados*, também usando a *srand*, porém iterando de forma sequencialmente parcial (90% ordenado) antes de chamar a *srand* e, por fim, *inversamente ordenados*, em que foi usada apenas uma iteração de modo que os valores maiores eram colocados primeiro, usando o *push_back* a partir do *tamanho – i*.

Diante disso, a coleta de dados foi feita para o *tempo* gasto na execução de cada algoritmo em cada tipo de distribuição apresentada, bem como para a *quantidade de comparações* realizadas e para a *quantidade de trocas* feitas. Nesse sentido, o tempo foi capturado a partir do uso da classe *high_resolution_clock* e, para capturar as informações da quantidade de comparações e trocas realizadas, foi criada a struct *ContaComparEtrocas* contendo os atributos *qtdComparacoes* e *qtdtrocas* usada na implementação de cada função de ordenação.

Após estruturar todo o código e capturar todas as informações exigidas para esta fazedura, foram gerados gráficos e tabelas utilizando o *Jupyter Notebook*, que facilitou a visualização dessas informações, apresentadas na seção seguinte junto de suas devidas análises.

RESULTADOS E ANÁLISE

TABELAS

Para as tabelas, devido o volume de informações, condensou-se em uma amostra de 1.000, 50.000 e 100.000, podendo ser acessar a tabela com todos os tamanhos usados no repositório[1].

Tabela de dados coletados para vetores de tamanho 1.000:

Algoritmo	Tamanho	Distribuição	tempo	comparações	trocas
Selection Sort	1000	aleatorio	0,000945	499500	1998
Insertion Sort	1000	aleatorio	0,000265	253352	253352
Bubble Sort	1000	aleatorio	0,001225	496799	504706

[1] Github – Sabrina Barbosa:
https://github.com/BlackbirdBlina/EDB1/tree/master/Unidade%202/Trabalho2

Quick Sort	1000	aleatorio	8,69E-05	13496	9075
Merge Sort	1000	aleatorio	0,000116	8731	19952
STL (Standard Library)	1000	aleatorio	4,93E-05	0	0
Selection Sort	1000	quase_ordenado	0,000457	499500	1998
Insertion Sort	1000	quase_ordenado	0,000078	77906	77906
Bubble Sort	1000	quase_ordenado	0,000696	498905	153814
Quick Sort	1000	quase_ordenado	5,85E-05	12334	4677
Merge Sort	1000	quase_ordenado	0,00011	8162	19952
STL (Standard Library)	1000	quase_ordenado	2,75E-05	0	0
Selection Sort	1000	inversamente_ordenados	0,000428	499500	1998
Insertion Sort	1000	inversamente_ordenados	0,000413	500499	500499
Bubble Sort	1000	inversamente_ordenados	0,001362	499500	999000
Quick Sort	1000	inversamente_ordenados	6,87E-05	12213	5499
Merge Sort	1000	inversamente_ordenados	9,85E-05	4932	19952
STL (Standard Library)	1000	inversamente_ordenados	1,35E-05	0	0

Tabela 1 – Dados coletados em um espaço amostral de 1.000 dados.
Fonte: Elaborado pela autora

Tabela de dados coletados para vetores de tamanho 50.000:

Algoritmo	Tamanho	Distribuição	tempo	comparações	trocas
Selection Sort	50000	aleatorio	0,84779	1,25E+09	99998
Insertion Sort	50000	aleatorio	0,383595	6,26E+08	6,26E+08
Bubble Sort	50000	aleatorio	3,740814	1,25E+09	1,25E+09
Quick Sort	50000	aleatorio	0,004083	975462	657651
Merge Sort	50000	aleatorio	0,005179	718368	1568928
STL (Standard Library)	50000	aleatorio	0,0027	0	0
Selection Sort	50000	quase_ordenado	0,796546	1,25E+09	99998
Insertion Sort	50000	quase_ordenado	0,121128	2,05E+08	2,05E+08
Bubble Sort	50000	quase_ordenado	1,495306	1,25E+09	4,09E+08
Quick Sort	50000	quase_ordenado	0,002494	950431	279540
Merge Sort	50000	quase_ordenado	0,003845	700478	1568928
STL (Standard Library)	50000	quase_ordenado	0,001024	0	0
Selection Sort	50000	inversamente_ordenados	0,731626	1,25E+09	99998
Insertion Sort	50000	inversamente_ordenados	0,766298	1,25E+09	1,25E+09
Bubble Sort	50000	inversamente_ordenados	2,6569	1,25E+09	2,5E+09
Quick Sort	50000	inversamente_ordenados	0,002053	1043323	275115
Merge Sort	50000	inversamente_ordenados	0,002161	382512	1568928
STL (Standard Library)	50000	inversamente_ordenados	0,000421	0	0

Tabela 2 – Dados coletados em um espaço amostral de 50.000 dados.

Fonte: Elaborado pela autora

Tabela de dados coletados para vetores de tamanho 100.000:

Algoritmo	Tamanho	Distribuição	tempo	comparações	trocas
Selection Sort	100000	aleatorio	3,336001	5E+09	199998
Insertion Sort	100000	aleatorio	1,561531	2,5E+09	2,5E+09
Bubble Sort	100000	aleatorio	16,68615	5E+09	5,01E+09
Quick Sort	100000	aleatorio	0,008567	2090118	1375917
Merge Sort	100000	aleatorio	0,011555	1536350	3337856
STL (Standard Library)	100000	aleatorio	0,00934	0	0
Selection Sort	100000	quase_ordenado	3,2604	5E+09	199998
Insertion Sort	100000	quase_ordenado	0,507962	8,2E+08	8,2E+08
Bubble Sort	100000	quase_ordenado	7,149912	5E+09	1,64E+09
Quick Sort	100000	quase_ordenado	0,005171	2051985	570432
Merge Sort	100000	quase_ordenado	0,006332	1502507	3337856
STL (Standard Library)	100000	quase_ordenado	0,002162	0	0
Selection Sort	100000	inversamente_ordenados	2,87694	5E+09	199998
Insertion Sort	100000	inversamente_ordenados	3,113396	5E+09	5E+09
Bubble Sort	100000	inversamente_ordenados	10,70818	5E+09	1E+10
Quick Sort	100000	inversamente_ordenados	0,00443	2175204	550122
Merge Sort	100000	inversamente_ordenados	0,004864	815024	3337856
STL (Standard Library)	100000	inversamente_ordenados	0,000872	0	0

Tabela 3 – Dados coletados em um espaço amostral de 100.000 dados.
Fonte: Elaborado pela autora

GRÁFICOS

Tempo

Os resultados da aplicação dos métodos descritos na seção anterior mostraram que em relação ao *tempo* o algoritmo que mais demorou para concluir a ordenação, para todos os tamanhos de vetor (1.000 a 100.000) e em todos os 3 tipos de distribuição (aleatório, quase ordenados e inversamente ordenados), foi o **Bubble Sort**, demorando aproximadamente 0.001225 segundos no *aleatório* para 1.000 elementos e cerca de

16.68615 segundos para um vetor de 100.000 elementos, conforme gráfico 1. sendo destacadamente visível a discrepância em comparação aos outros algoritmos:

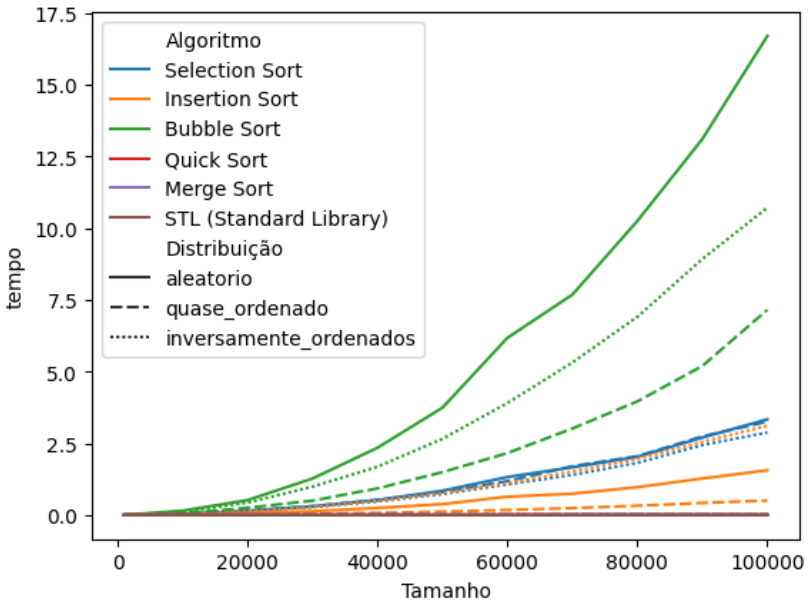

Gráfico 1 – Tempo de execução dos algoritmos de ordenação em diferentes tamanhos de entrada.
Fonte: Elaborado pela autora.

Vale frisar ainda que, apesar dos tempos gastos nas distribuições de *quase ordenados* e *inversamente ordenados*, terem sindo consideravelmente menor em relação ao seu desempenho de tempo para o *aleatório*, ele ainda não performou tão bem quanto os outros algoritmos.

Apesar disso, ao gerar o gráfico sem o Bubble Sort ainda há uma diferença gritante entre os algoritmos, especialmente devido a alta performance do STL Sort, sendo possível visualizar no gráfico 2. Entretanto é válido ressaltar que, retirando apenas o Bubble Sort ainda não fica clara a diferença entre os algoritmos Quick Sort, Merge Sort e STL Sort, sendo necessário realizar um recorte dos 3.

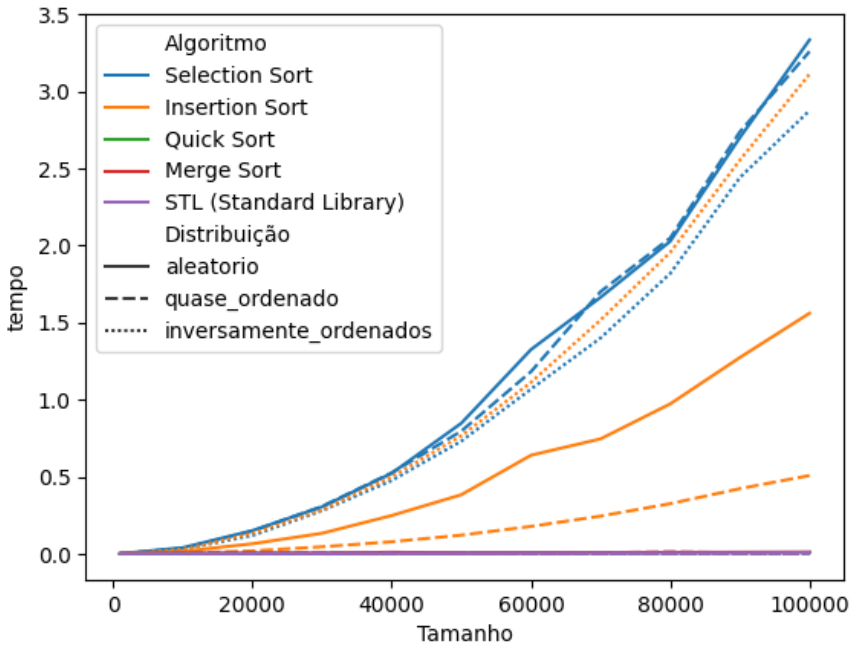

Gráfico 2 – Tempo de execução dos algoritmos de ordenação sem o Bubble Sort.
Fonte: Elaborado pela autora.

Não obstante, o que obteve maior rapidez foi o **STL Sort**, demorando aproximadamente apenas 0.000049 segundos, em um vetor de 1.000 elementos distribuídos aleatóriamente, e aproximadamente 0.0093396 segundos para 100.000 elementos.

Nesse sentido, é importante destacar que, apesar de seu melhor desempenho de tempo, é possível perceber um crescimento acentuado no tempo gasto para ordenar entre os tamanhos de 80.000 a 100.000 elementos, onde o **Quick Sort** aparentemente tende a diminuir seu tempo conforme o aumento considerável no tamanho do vetor de dados para a distribuição *aleatória*, podendo ser visto no recorte do gráfico 3 para os 3 melhores algoritmos, nos 3 tipos de distribuição, em relação ao tempo.

Além disso, para o **Merge Sort** também é possível fazer uma leitura semelhante ao **Quick Sort**, porém apenas em relação às distribuições de *quase ordenados* e *inversamente ordenados*, em que apresenta uma queda acentuada no tempo gasto no mesmo intervalo de 80.000 a 100.000 elementos:

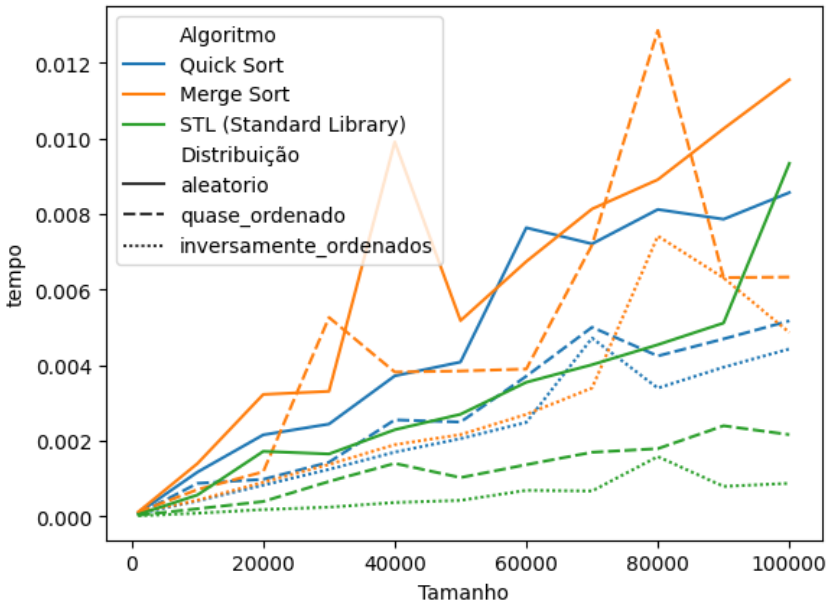

Gráfico 3 – Algoritmos de ordenação que tiveram melhor desempenho de tempo.
Fonte: Elaborado pela autora.

Diante disso, é possível concluir que para performar em cima de tempo de execução, os algoritmos mais indicados são o STL Sort, o Quick sort ou, por último, mas não menos importante, o Merge Sort.

Comparações

Em relação à quantidade de comparações realizadas pelos algoritmos, o **Selection Sort** gerou a maior quantidade de comparações, sendo 499.500 comparações na menor quantidade de dados (1.000) e 4.999.950.000 na maior (100.000) para distribuição de dados aleatórios. Nesse sentido, se faz necessário desconsiderá-lo para poder visualizar melhor a quantidade de comparações dos outros algoritmos, conforme Gráfico 4:

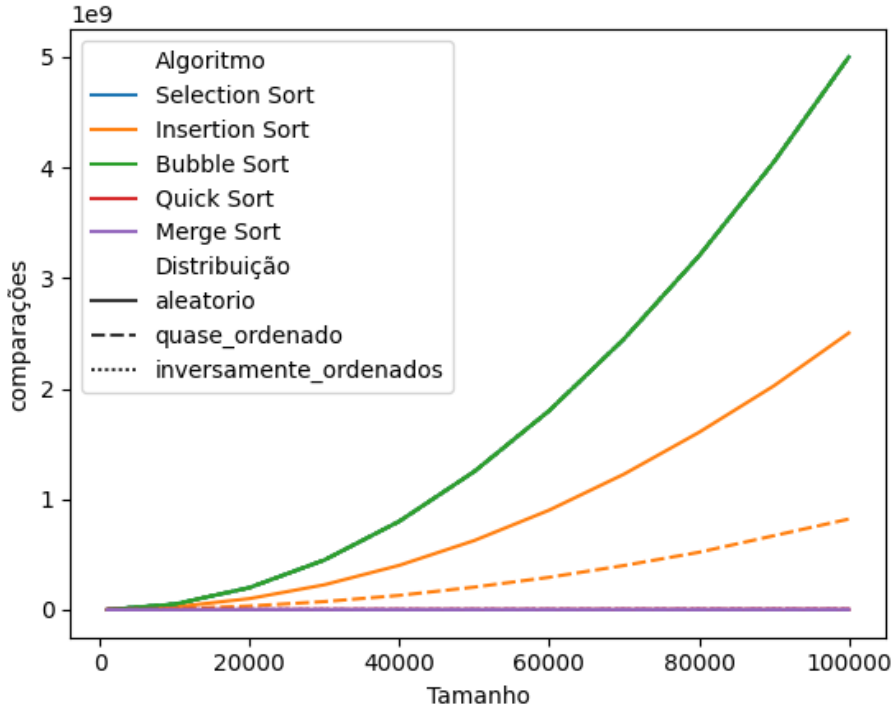

Gráfico 4 – Quantidade de comparações realizadas pelos algoritmos de ordenação.
Fonte: Elaborado pela autora.

Destaca-se ainda que o **Bubble Sort** realizou praticamente a mesma quantidade de comparações que o **Selection Sort**, sendo igualmente desconsiderável, como se pode ver nos gráficos 5 e 6, respectivamente. Permanecendo praticamente no mesmo nível para os outros tipos de distribuição de dados – *quase ordenados* e *inversamente ordenados*.

Além disso, o **Insertion Sort** foi o que obteve o terceiro pior desempenho, realizando 253352 comparações para um vetor de 1.000 dados e aproximadamente 2503406 comparações, em um vetor de 100.000 dados distribuídos aleatóriamente, vide Gráfico 7:

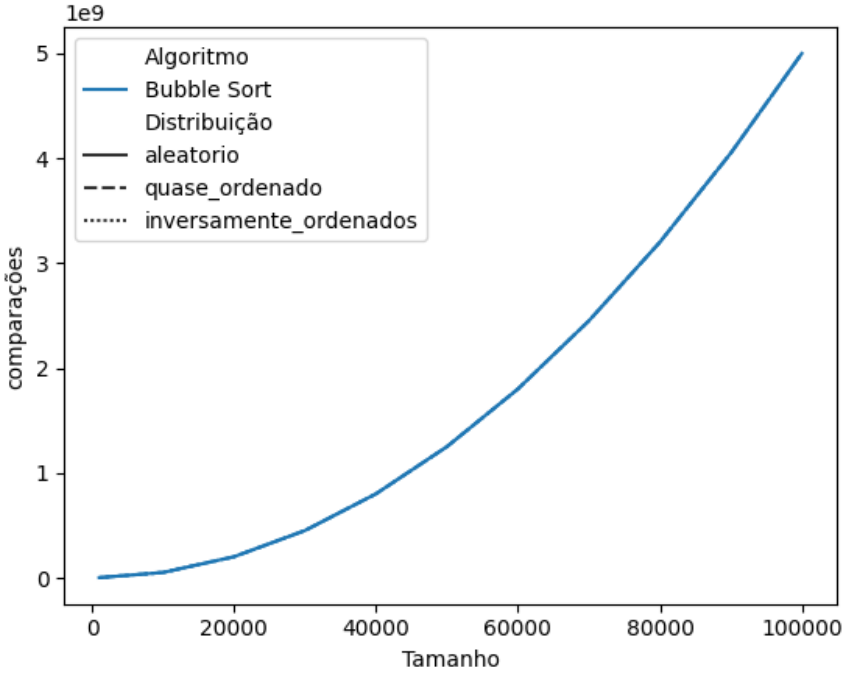

Gráfico 5 – Quantidade de comparações realizadas pelo Bubble Sort.
Fonte: Elaborado pela autora.

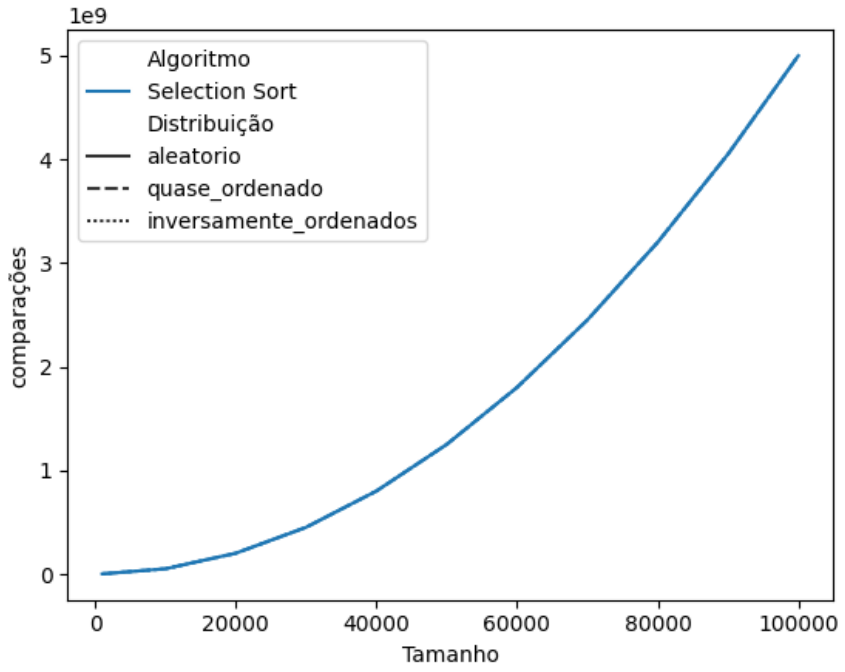

Gráfico 6 – Quantidade de comparações realizadas pelo Selection Sort.
Fonte: Elaborado pela autora.

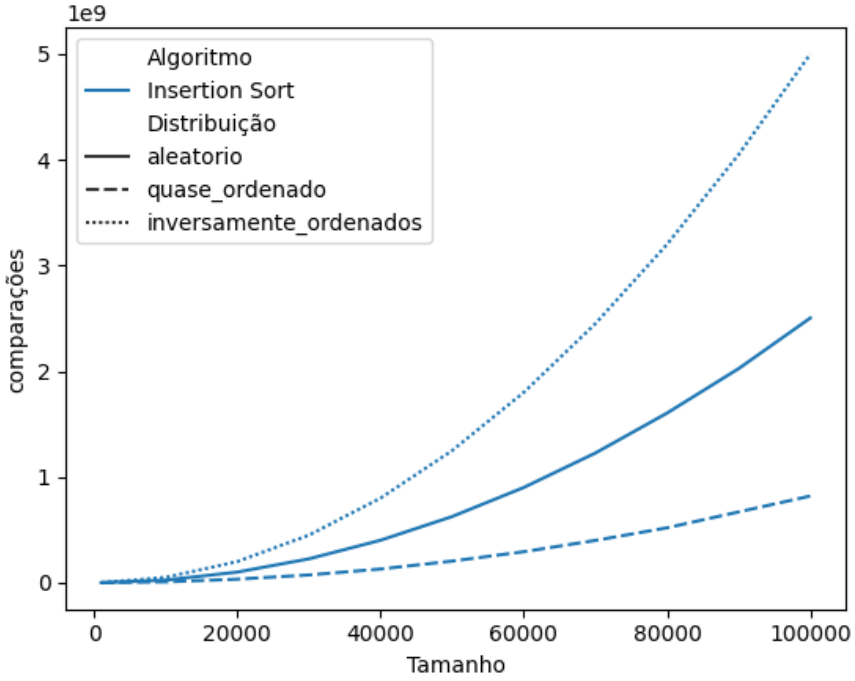

Gráfico 7 – Quantidade de comparações realizadas pelo Insertion Sort.
Fonte: Elaborado pela autora.

Dessa forma, observando o Gráfico 7, é possível perceber com mais clareza o desempenho do **Insertion Sort**. Nesse sentido, nota-se que, embora simples e eficiente para pequenas quantidades de dados ou dados quase ordenados, esse algoritmo apresenta uma quantidade de comparações significativamente maior nas distribuições *inversamente ordenadas* e *aleatórias* à medida que o tamanho do vetor cresce. Evidenciando sua limitação para grandes volumes de dados desordenados.

Em contrapartida, o **Quick Sort** e o **Merge Sort** apresentaram quantidades de comparações bem menores em todos os casos, o que reforça sua eficiência algorítmica. Sendo assim, o Quick Sort, apesar de não garantir a menor quantidade absoluta de comparações, se destacou pelo equilíbrio entre desempenho e baixa complexidade. Não obstante, o **Merge Sort**, por sua vez, manteve um comportamento estável, inclusive nas distribuições inversamente ordenadas, podendo ser visto no Gráfico 8:

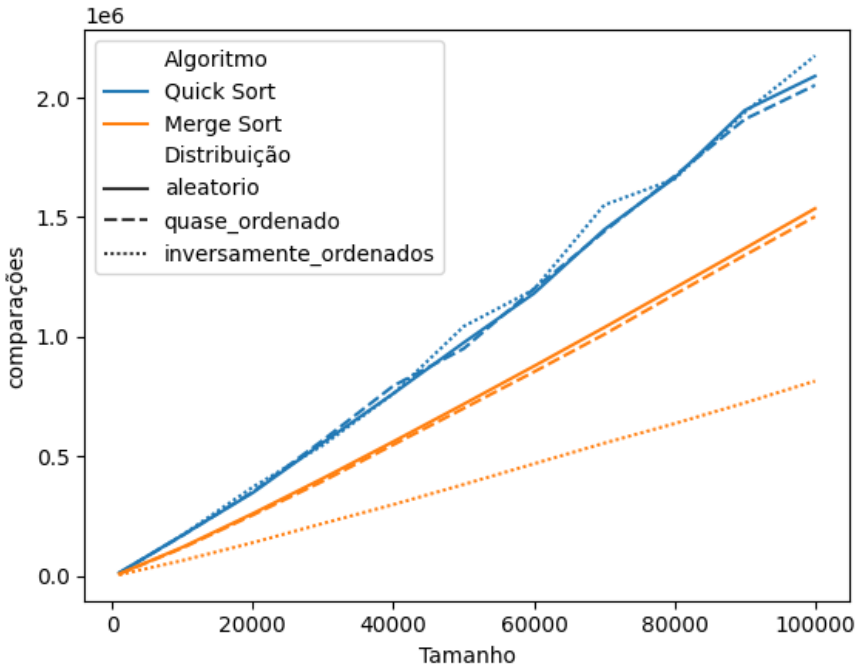

Gráfico 8 – Quantidade de comparações realizadas pelo Quick Sort e pelo Merge Sort.
Fonte: Elaborado pela autora.

Com isso, é possível concluir que os algoritmos Quick sort e o Merge Sort, são os mais indicados para quando **não** se deseja maiores gastos computacionais em acesso aos dados.

Trocas

Em relação à quantidade de trocas, o algoritmo que mais se destacou negativamente foi o **Bubble Sort**, pois diante da sua natureza de trocar elementos sempre que necessário, faz com que se torne extremamente ineficiente para dados desordenados. Para tanto, foram registradas até 9.999.900.000 trocas para vetores inversamente ordenados com 100.000 elementos, o que representa um crescimento exponencial no custo computacional, podendo ser observado no Gráfico 9 a seguir:

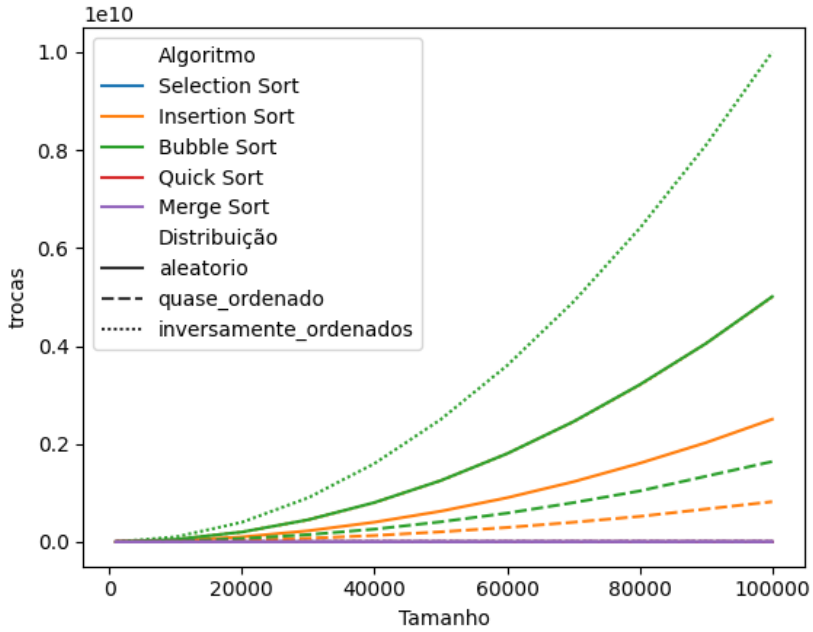

Gráfico 9 – Quantidade de trocas realizadas pelos algoritmos de ordenação.
Fonte: Elaborado pela autora.

Além disso, outro algoritmo com quantidade alta de trocas foi o **Insertion Sort**, especialmente em dados inversamente ordenados, onde cada inserção exige o deslocamento de praticamente todos os elementos anteriores, sendo melhor visualizado no Gráfico 10:

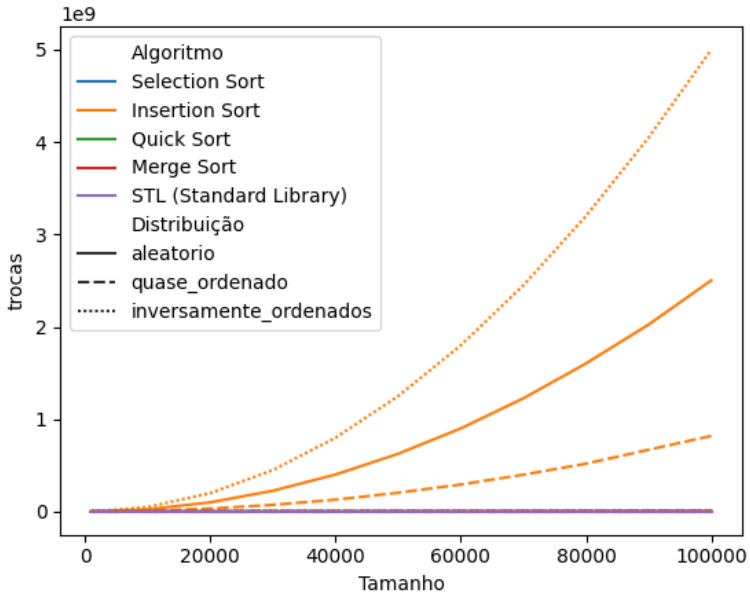

Gráfico 10 – Quantidade de trocas realizadas pelos algoritmos de ordenação (sem o Bubble Sort).
Fonte: Elaborado pela autora

Apesar disso, o **Selection Sort**, mesmo apresentando um número constante de comparações (independente da ordenação dos dados), teve um número significativamente menor de trocas, sendo uma troca por iteração no máximo, de acordo com o Gráfico 11. Nessa perspectiva, esse comportamento é útil quando o custo de trocar elementos é alto.

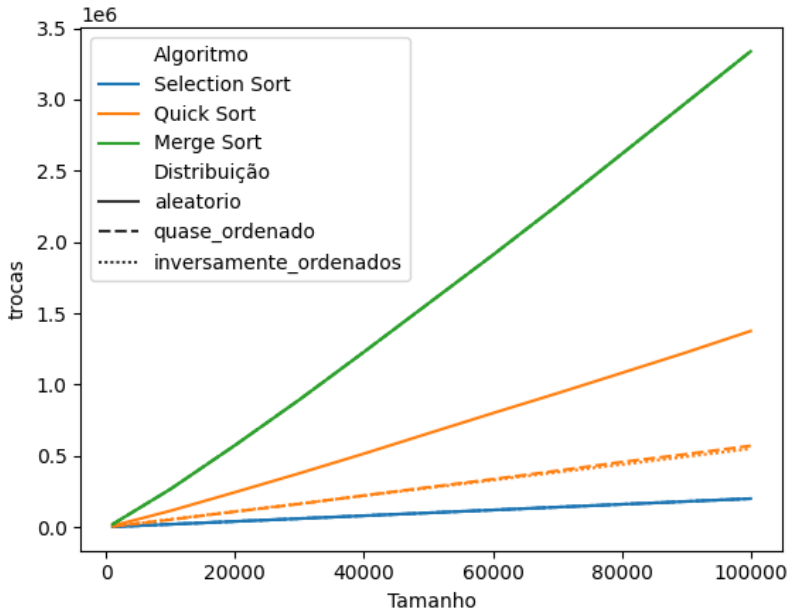

Gráfico 11 – Quantidade de trocas realizadas pelos algoritmos Selection Sort, Quick Sort e Merge Sort.
Fonte: Elaborado pela autora

Por fim, é cabível destacar que o **Quick Sort** e o **Merge Sort**, mais uma vez, demonstraram eficiência, dado que o **Quick Sort** teve um crescimento proporcional, porém moderado, no número de trocas, especialmente em dados aleatórios e quase ordenados. Já o **Merge Sort** manteve um número estável de trocas em todas as distribuições, indicando que sua abordagem baseada em vetores auxiliares contribui para uma movimentação otimizada dos elementos, todavia ainda realizando muito mais trocas do que o **Quick** e o **Selection Sort**.

RANKING

De acordo com os dados coletados, o ranking para os algoritmos que obtiveram o melhor desempenho de tempo está descrito na Tabela 4. Salientando que essas informações foram geradas para a maior quantidade de dados e na distribuição inversa dos dados:

	Algoritmo	tempo
1º	STL (Standard Library)	0,000872
2º	Quick Sort	0,00443
3º	Merge Sort	0,004864
4º	Selection Sort	2,87694
5º	Insertion Sort	3,113396
6º	Bubble Sort	10,70818

Tabela 4 – Ranking dos algoritmos de ordenação.
Fonte: Elaborado pela autora

Conclusão

A análise comparativa entre os algoritmos de ordenação Selection Sort, Insertion Sort, Bubble Sort, Quick Sort, Merge Sort e STL Sort permitiu observar comportamentos distintos em relação a tempo de execução, quantidade de comparações e trocas realizadas, com base em diferentes distribuições de dados e tamanhos de entrada. Ressaltando que foi constatado que algoritmos com complexidade quadrática, como Bubble, Selection e Insertion Sort, não são recomendados para grandes volumes de dados desordenados, devido a sua baixa escalabilidade. Nesse sentido, o Bubble Sort, em especial, foi o menos eficiente nos três critérios analisados.

Contudo, os algoritmos Quick Sort, Merge Sort e especialmente o STL Sort demonstraram ótimo desempenho. Assim, o STL Sort obteve os melhores tempos de execução, enquanto o Merge Sort mostrou-se mais estável em diferentes distribuições e o Quick Sort apresentou eficiência combinada entre tempo e trocas.

Dessa forma, diante do exposto, dependendo do cenário e da métrica mais relevante (tempo, trocas ou comparações), a escolha do algoritmo pode variar. Contudo, de forma geral, os algoritmos com complexidade assintótica N Log N são os mais apropriados para aplicações práticas em larga escala, especialmente quando otimizados, como no caso da STL.

REFERÊNCIAS BIBLIOGRÁFICAS

CORMEN, Thomas H. ... [et al.]. Algoritmos; [tradução Arlete Simille Marques]. - Rio de Janeiro: Elsevier, 2012. il. Tradução de: Introduction to algorithms, 3rd ed.

SEDGEWICK, Robert. Algorithms. Addison-Wesley Publishing Company, 2010.

GUILHERME, João. Estrutura de Dados Básicas 1 - Aula_13_-Revisão_Algoritmos_de_Ordenação. – Natal, 2025.