Introdução

O trabalho tem como objetivo principal realizar a análise comparativa dos seguintes algoritmos de ordenação: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, avaliando as eficiências e crescimentos dos algoritmos.

Os algoritmos de ordenação são muito importantes, pois eles organizam os dados de forma eficiente e estabelecem critérios/regras que vão trazer mais rapidez numa futura consulta ou busca daqueles dados, trazendo assim mais facilidade e melhor aproveitamento computacional.

Algoritmos

Selection Sort -

A lógica do selection sort é bem simples, varre a lista toda para procurar o menor elemento da lista, assim troca de posição com o elemento da primeira posição, depois procura o próximo menor no restante da lista e faz as trocas novamente. No fim, teremos o mesmo vetor ordenado!

Complexidade:
- Melhor caso: $O(n^2)$
- Caso Médio: $O(n^2)$
- Pior caso: $O(n^2)$

Isso ocorre, pois o algoritmo precisa percorrer o vetor inteiro para encontrar o menor elemento mesmo que a lista já esteja ordenada de fábrica.

Bubble Sort

O algoritmo percorre o vetor várias vezes, comparando pares de elementos vizinhos. Sempre que encontra um par fora de ordem, realiza uma troca, fazendo com que os maiores valores subam para o final do vetor a cada passagem. Esse processo se repete até que não tenham mais trocas possíveis, indicando que o vetor está totalmente ordenado.

Complexidade:
- Melhor caso: $O(n)$
- Caso Médio: $O(n^2)$
- Pior caso: $O(n^2)$

Isso ocorre porque, mesmo sendo possível parar mais cedo se o vetor já estiver ordenado, no pior cenário o algoritmo precisa realizar muitas comparações e trocas

entre elementos adjacentes.

Insertion Sort

A lógica do *insertion sort* é simples, o algoritmo percorre o vetor da esquerda para a direita e, a cada passo, insere o elemento atual na posição correta em relação aos anteriores. Ele compara e move os valores até que todos os elementos à esquerda estejam ordenados.

Complexidade:

Melhor caso: O(n)
Caso Médio: $O(n^2)$
Pior caso: $O(n^2)$

Isso ocorre porque, quando os dados estão quase ordenados, o número de movimentações é pequeno. Porém, se o vetor estiver inversamente ordenado, o algoritmo precisará fazer muitas comparações e deslocamentos.

Merge Sort

O *Merge Sort* funciona da seguinte forma: Ele divide o vetor ao meio recursivamente até que reste apenas um elemento em cada parte. Depois, começa a juntar essas partes de forma ordenada, formando assim listas cada vez maiores e organizadas até reconstruir o vetor completo em ordem.

Complexidade:

Melhor caso: $O(n * \log_2 n)$
Caso Médio: $O(n * \log_2 n)$
Pior caso: $O(n * \log_2 n)$

Isso ocorre porque o algoritmo sempre divide o vetor pela metade e, em cada nível da recursão, precisa percorrer todos os elementos para uni-los novamente.

Quick Sort

O *Quicksort* é baseado na ideia de escolher um pivô, que serve como referência para dividir o vetor em duas partes: uma com elementos menores que o pivô e outra com elementos maiores. Depois, ele aplica o mesmo processo recursivamente em cada parte até que todas estejam ordenadas.

Complexidade:

Melhor caso: O(n log n)
Caso Médio: O(n log n)
Pior caso: $O(n^2)$

Isso ocorre porque, quando o pivô divide bem o vetor, o algoritmo é muito eficiente. Mas se o pivô escolhido for ruim (por exemplo, o menor ou o maior elemento), as divisões ficam desbalanceadas.

Metodologia

Nesta etapa, foram realizados testes experimentais com o objetivo de comparar o desempenho de diferentes algoritmos de ordenação quanto ao tempo de execução, número de comparações e trocas realizadas. Todos os experimentos foram executados utilizando vetores gerados de forma aleatória pelo próprio algoritmo desenvolvido em C++.

Foram utilizados três tipos distintos de vetores para avaliar o comportamento dos algoritmos sob diferentes condições:

Vetor aleatório: contém valores inteiros gerados de forma totalmente randômica, representando um caso “base” ou mais comum.
Vetor quase ordenado: inicia em ordem crescente, mas apresenta pequenas alterações ocasionais, simulando uma lista parcialmente organizada.
Vetor inversamente ordenado: apresenta os elementos em ordem decrescente, representando a pior condição para alguns algoritmos.

Tamanhos dos vetores testados:
Foram definidos quatro tamanhos de entrada, com o objetivo de observar o impacto da quantidade de elementos no desempenho de cada algoritmo. Sendo eles:
1.000, 5.000, 10.000 e 20.000 elementos.

Métricas coletadas:
Durante a execução dos experimentos, foram registradas as seguintes métricas:

Tempo de execução (ms)**:** calculado utilizando a biblioteca <ctime>, com a diferença entre os instantes inicial e final de execução de cada algoritmo.

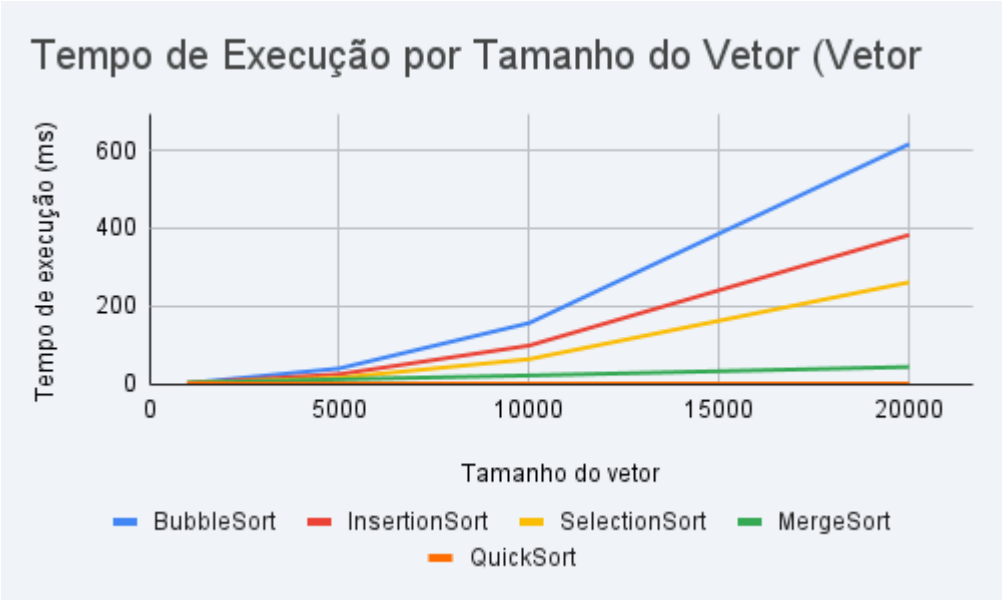
Número de comparações: contabiliza quantas vezes os elementos do vetor foram comparados entre si durante o processo de ordenação.

Número de trocas: representa quantas vezes os elementos foram efetivamente movimentados no vetor.

Os resultados foram exportados automaticamente para um arquivo csv, dessa forma, permite uma análise posterior em planilhas e geração de gráficos comparativos.

Análise Final

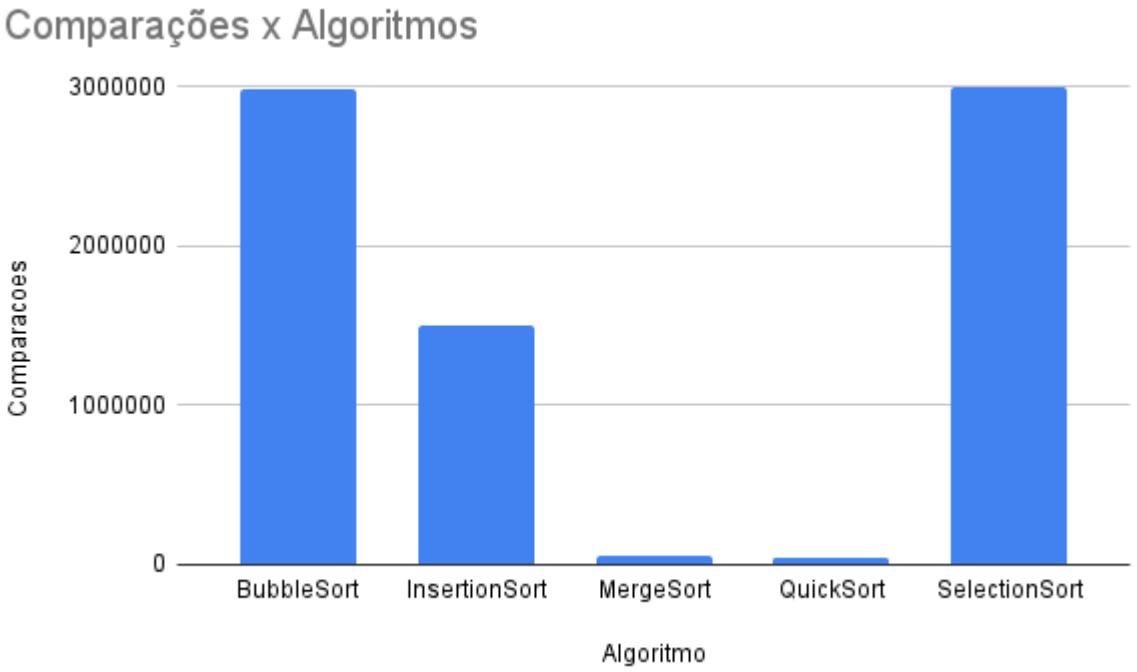
Os dados coletados foram armazenados no arquivo resultados.csv, contendo algumas colunas, sendo elas: o algoritmo utilizado (*bubblesort, insertionsort, mergesort, quicksort* ou *selectionsort*), o tamanho do vetor (1000, 5000, 10000 ou 20000 elementos), o tipo de vetor (aleatório, quase ordenado ou inversamente ordenado), o tempo de execução (em milissegundos), além do número de comparações e trocas realizadas durante o processo de ordenação.



Tempo de execução por tamanho do Vetor (Vetor Ordenado)

O gráfico mostra o tempo de execução dos algoritmos em função do tamanho do vetor, considerando vetores já ordenados.

Algoritmos de complexidade quadrática, como Bubble sort, Insertion sort e Selection sort, aumentam significativamente o tempo de execução conforme o tamanho do vetor cresce, sendo o Bubble o mais lento. Já o Merge sort e o Quicksort apresentaram desempenho muito superior, mantendo tempos baixos e estáveis, o que confirma suas complexidades(n log n). Assim, observa-se que os resultados práticos estão de acordo com a teoria, evidenciando a eficiência dos algoritmos de divisão e conquista em comparação aos métodos mais simples.



O gráfico acima mostra que os algoritmos Bubble Sort, Insertion Sort e Selection Sort, de complexidade quadrática, aumentam rapidamente o tempo de execução conforme o tamanho do vetor cresce, com o Bubble sendo o mais lento. Já o Merge Sort e o Quick Sort mantêm tempos de execução baixos e estáveis, confirmando suas complexidades O(n log n). Assim, os resultados experimentais estão de acordo com a teoria, evidenciando que os algoritmos de divisão e conquista são mais eficientes para vetores maiores e ordenados.

Conclusões

Por fim, com base nos resultados obtidos, o MergeSort e o QuickSort apresentaram o melhor desempenho geral, confirmando sua eficiência teórica de O(n log n) e sendo indicados para grandes volumes de dados. O InsertionSort mostrou bom desempenho em vetores pequenos ou quase ordenados, enquanto o SelectionSort manteve tempo constante, porém ineficiente para grandes entradas. Já o BubbleSort foi o mais lento em todos os cenários. Assim, conclui-se que algoritmos de divisão e conquista são mais adequados para aplicações que exigem alta performance, enquanto os métodos simples são mais indicados para vetores curtos ou situações teóricas.