

Análise Comparativa de Algoritmos de Ordenação

Introdução

Este trabalho tem como objetivo explorar e comparar os principais algoritmos de ordenação, entendendo como eles funcionam, seu desempenho e em que situações são mais indicados.

Sabe-se que os algoritmos de ordenação são essenciais na ciência da computação, pois são amplamente usados em diversas áreas e aplicações práticas. Eles são responsáveis por organizar conjuntos de dados em uma ordem específica — geralmente crescente ou decrescente — para facilitar tarefas como busca, análise e visualização das informações.

Ter dados bem organizados é crucial na melhoria de desempenho de outros algoritmos e sistemas, impactando na organização interna de dados e no uso eficiente de memória, por exemplo.

Fundamentação Teórica

Selection Sort

Consiste em um algoritmo de classificação baseado em comparação. Ele classifica uma matriz por meio da repetida troca de posição do menor elemento não classificado pela menor posição disponível.

Implementação do Selection Sort

- Percorra toda a lista e descubra o **menor elemento**
- Troque de posição o **menor elemento** com o **primeiro elemento**
- Encontre o **menor elemento (segundo menor)** entre os elementos restantes e trocamos com o **segundo elemento** do vetor
- Continue o processo até que todos os elementos estejam classificados corretamente

Complexidade Temporal

Complexidade de tempo: $O(n^2)$

- **Melhor Caso: $O(n^2)$**
- **Caso Intermediário: $O(n^2)$**
- **Pior Caso: $O(n^2)$**

Complexidade Espacial

Complexidade Espacial: $O(1)$ é a **única memória extra utilizada** para variáveis temporárias

Insertion Sort

Consiste na classificação por meio da inserção de forma iterativa de cada elemento de uma lista não classificada em sua posição correta em uma parte classificadas da lista

Implementação

- Começa com o segundo elemento da matriz, considerando o primeiro elemento como classificado
- Compara o segundo elemento com o primeiro, se o segundo for menor, troque-os de posição
- Compare o terceiro elemento com os anteriores e coloque-o na posição correta
- Repita até que toda a matriz esteja classificada

Complexidade Temporal do Insertion Sort

- **Melhor caso: $O(n)$, Conjunto de dados ordenados**
- **Caso médio: $O(n^2)$, Conjunto de dados ordenados aleatoriamente**
- **Pior caso: $O(n^2)$, Conjunto de dados ordenados inversamente**

Complexidade Espacial

$O(1)$, o Insertion Sort requer apenas **$O(1)$** espaço adicional, semelhantemente ao **Selection Sort**

Bubble Sort

Consiste na troca repetidamente dos elementos adjacentes se eles estiverem na ordem errada. Esse algoritmo não é recomendado para grandes volumes de dados, pois a complexidade de tempo média e para o pior caso é alta

Implementação

- Ordenamos o array usando múltiplas passagens. Após a primeira passagem, o elemento máximo vai para o final (sua posição correta). Da mesma forma, após a segunda passagem, o segundo maior elemento vai para a penúltima posição e assim por diante.
- Em cada passagem, processamos apenas os elementos que ainda não foram movidos para a posição correta. Após k passagens, os maiores k elementos devem ter sido movidos para as últimas k posições.
- Em uma passagem, consideramos os elementos restantes, comparamos todos os adjacentes e trocamos de lugar se o elemento maior estiver antes do menor. Se continuarmos fazendo isso, obtemos o maior (dentre os elementos restantes) em sua posição correta.

Complexidade Temporal

- **Melhor caso: $O(n)$, Conjunto de dados ordenados**
- **Caso médio: $O(n^2)$, Conjunto de dados ordenados aleatoriamente**
- **Pior caso: $O(n^2)$, Conjunto de dados ordenados inversamente**

Complexidade Espacial

$O(1)$, o Bubble Sort requer apenas **$O(1)$** espaço adicional, semelhantemente ao **Selection Sort** e ao **Insertion Sort**. Porém o **Bubble Sort** precisa de **quantidade constante de espaço adicional**

Quick Sort

Consiste em um algoritmo de classificação baseada no **Dividir para conquistar** que escolhe um elemento como um pivô e particiona a matriz fornecida em torno do pivô escolhido, colocando o pivô em sua posição correta na matriz classificada

Implementação

- Escolha um pivô
- Partição da matriz: Reorganiza a matriz ao redor do **pivô**. Após o particionamento, todos os elementos **menores** que o **pivô** estarão à sua **esquerda** e os **maiores** à sua **direita**. O pivô fica na posição correta e obtemos o índice do pivô
- **Chame recursivamente**: Aplique recursivamente o mesmo processo aos conjunto de dados **particionadas**
- **Caso base**: A recursão para quando resta apenas um elemento no subconjunto, pois um único elemento já está classificado

Complexidade

- **Melhor caso: $O(\log n)$, Partições balanceadas** - Cada partição com $n/2$ elementos
- **Caso médio: $O(n \log n)$, O pivô divide a matriz em duas partes, mas não necessariamente iguais**
- **Pior caso: $O(n^2)$** , Ocorre quando o **menor ou o maior elemento é sempre escolhido como pivô** (vetor classificado)

Complexidade espacial

$O(n)$, devido a pilha de chamadas recursivas

Complexidade dos algoritmos

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Espaço Auxiliar	Estável	In-place
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Não	Sim
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Sim	Não
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sim	Sim
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Não	Sim
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sim	Sim

Metodologia

Tipos de Dados

Foram utilizados nos testes conjuntos de dados com as seguintes configurações:

- **Ordenados Aleatoriamente**
- **Parcialmente Ordenados**
- **Inversamente Ordenados**

Tamanho dos Conjuntos de dados

Foram utilizados vetores de **1000** elementos, com valores entre **1** e **1000**

Métricas

Coletou-se informações à respeito do tempo de execução de cada algoritmo de classificação para cada tipo de conjunto de dados (aleatorio, parcialmente ordenado e inversamente ordenado).

Método de medição

Para a medição de tempo de execução, fora utilizado a função `tempo_medio_exec` no arquivo `main.cpp`. Essa função utiliza da biblioteca padrão do C++ `<chrono>` para a medição do tempo. A função recebe como parâmetro um dos algoritmos de ordenação declarados no arquivo `include/includes.hpp` e o vetor que será ordenado. A função retorna o tempo médio de execução da implementação em 100 aplicações diferentes. Utilizou-se dessa metodologia para aumentar a fidedignidade dos dados, já que apenas uma ou duas aplciações do mesmo algoritmo para o mesmo vetor não seriam suficientes para levantar dados próximos da realidade de cada algoritmo, já que a apicação estaria sujeita a forte influencia de questões como hardware e memória do dispositivo.

```
#include <chrono>

template<typename Func>
long long tempo_medio_exec(const vector<int>& vetor, Func sort_func){
    long long duracao = 0;

    for (int i = 0; i < 100; i++) {
        // Garantia que o vetor que iremos ordenar não já foi ordenado em
        // uma aplicação anterior do loop
        vector<int> copia_vetor = vetor;
        auto inicio = high_resolution_clock::now();
        // Algoritmo passado como parâmetro
        sort_func(copia_vetor);
        auto fim = high_resolution_clock::now();
        // Calculo do tempo de execução total
        duracao += duration_cast<microseconds>(fim - inicio).count();
    }
    // Tempo médio
    return duracao / 100;
}
```

Resultados e Análises

Tabelas

Conjunto de Dados Ordenados Aleatoriamente

Método de Ordenação	Tempo Médio (µs)
Selection Sort	198.054
Insertion Sort	103.486

Método de Ordenação	Tempo Médio (µs)
Bubble Sort	331.593
Merge Sort	3.283
Quick Sort	1.249

Conjunto de Dados Parcialmente Ordenados

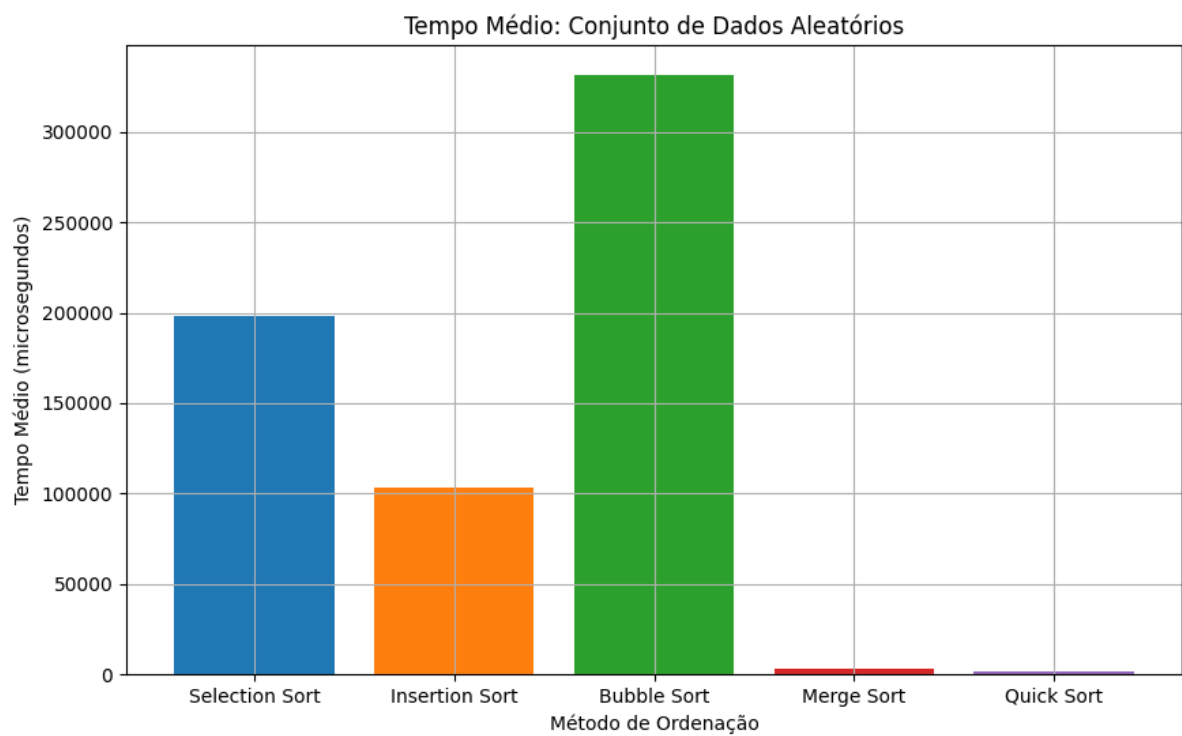
Método de Ordenação	Tempo Médio (µs)
Selection Sort	199.267
Insertion Sort	78.318
Bubble Sort	287.804
Merge Sort	3.024
Quick Sort	1.188

Conjunto de Dados Inversamente Ordenados

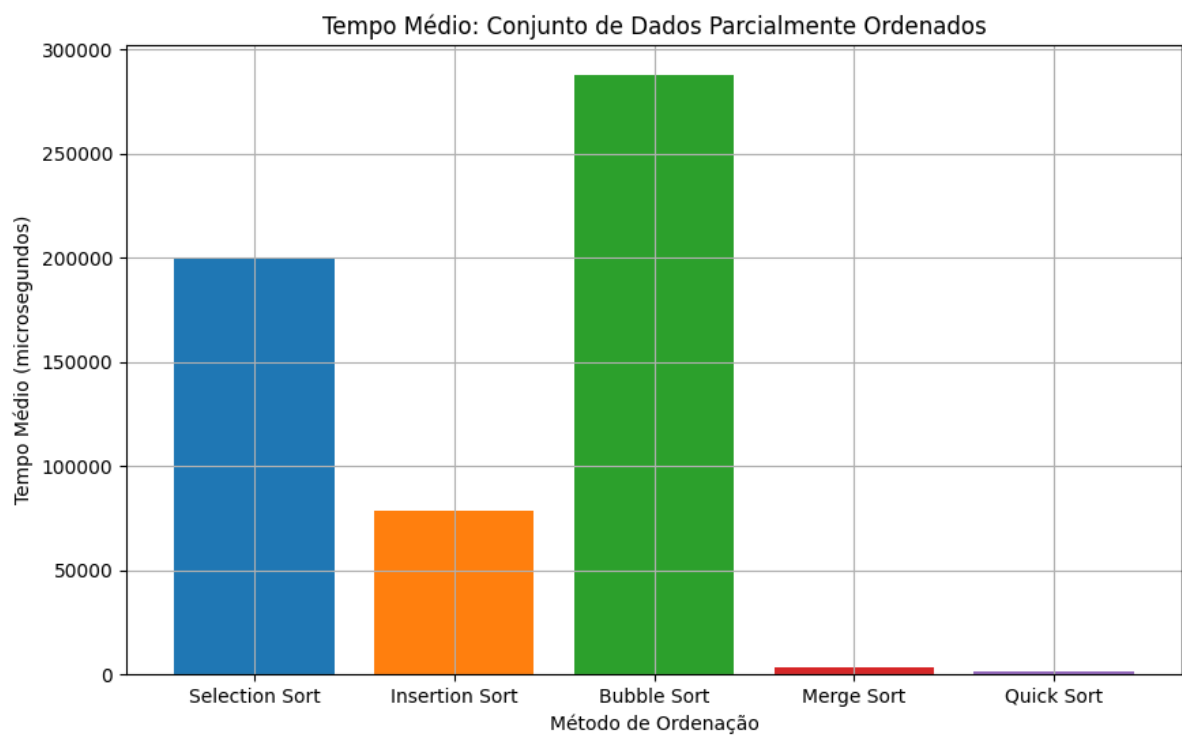
Método de Ordenação	Tempo Médio (µs)
Selection Sort	209.896
Insertion Sort	210.913
Bubble Sort	487.754
Merge Sort	2.556
Quick Sort	47.483

Graficos

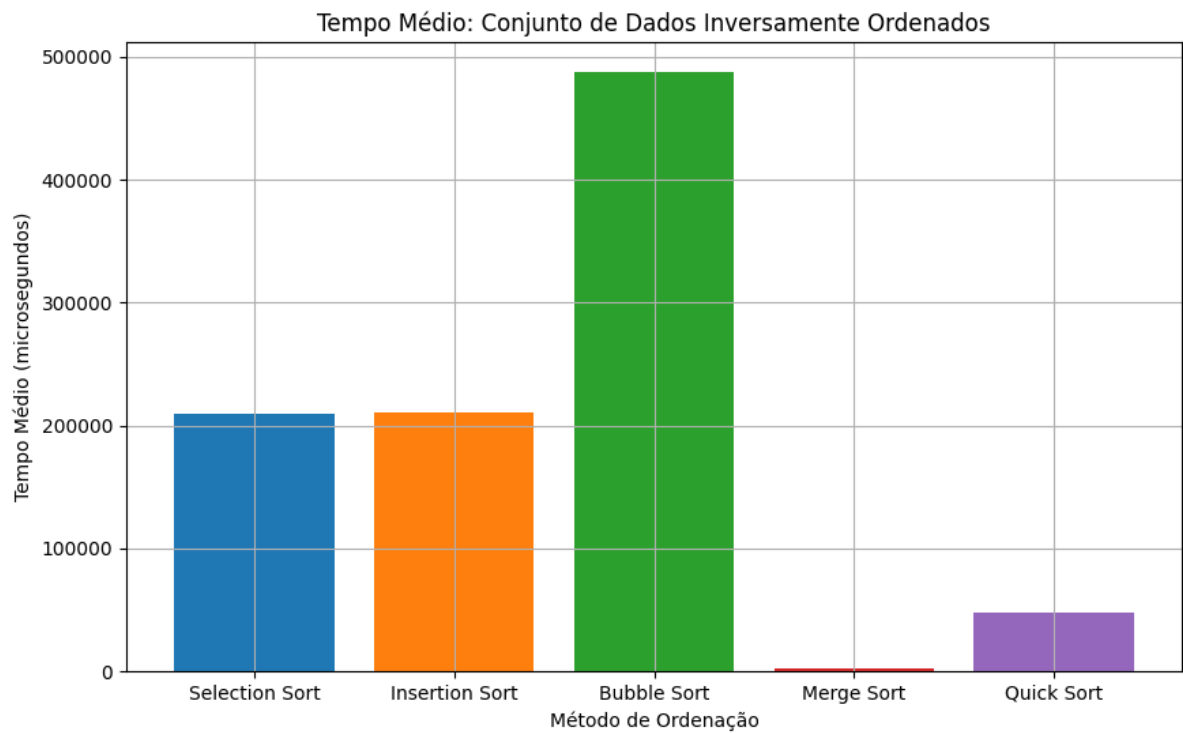
Conjunto de Dados Ordenados Aleatoriamente



Conjunto de Dados Parcialmene Ordenados



Conjunto de Dados Inversamente Ordenados



Discussão


Por meio do levantamento de dados, foram obtidos os retornos esperados para a maioria dos algoritmos, com exceção do `bubble_sort`. Esperava-se que esse algoritmo apresentasse resultados semelhantes aos do `selection_sort`, uma vez que ambos possuem a mesma complexidade temporal para os tipos de conjuntos de dados analisados ($O(n^2)$). Ressalta-se que não foi testada a aplicação dos algoritmos em um vetor já ordenado (caso em que o `bubble_sort` possui complexidade $O(n)$).


Entretanto, embora o `bubble_sort` e o `selection_sort` apresentem a mesma complexidade nos casos testados (vetor parcialmente ordenado, aleatorio e inversamente ordenado), o `bubble_sort` registrou tempos de execução significativamente superiores aos do `selection_sort` nos três tipos de conjuntos de dados avaliados.

Desconsiderando essa questão, os demais algoritmos de ordenação apresentaram os resultados esperados. Destaca-se a eficiência do `merge_sort` e do `quick_sort`: o `merge_sort` manteve desempenho constante nos três tipos de conjuntos de dados, enquanto o `quick_sort` obteve os menores tempos de execução entre todos os algoritmos, exceto no caso do conjunto de dados inversamente ordenado — comportamento já previsto.

Conclusão

Ranking

 Posição	Algoritmo	Tempo Médio (µs)	Observações
1º	Merge Sort	16.640	Estável e com tempo médio consistentemente baixo.

 Posição	Algoritmo	Tempo Médio (µs)	Observações
2º	Quick Sort	8.863	Mais rápido que o Merge Sort na média, porém seu pior caso compromete o desempenho.
3º	Insertion Sort	130.905	Simples e eficiente para listas pequenas ou quase ordenadas.
4º	Selection Sort	202.405	Fácil de implementar, mas apresenta baixa eficiência em geral.
5º	Bubble Sort	369.050	O menos eficiente — deve ser evitado em listas grandes.

Recomendações:

- **Quick Sort** : Recomendado quando há certeza de que o vetor não está inversamente ordenado
- **Merge Sort** : Indicado quando não se tem certeza sobre a ordenação dos dados
- **Insertion Sort** : Adequado para vetores pequenos ou quase ordenados, especialmente útil em ambientes com restrições de memória
- **Selection Sort** : Pode ser utilizado em arrays pequenos, mas, em geral, o **Insertion Sort** oferece melhor desempenho na prática
- **Bubble Sort** : Apenas para fins didáticos