Trabalho Prático

Estrutura de Dados 2024/1

Felipe Dias de Souza Martins - 2023002073
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
felipedsm@ufmg.br

1. Introdução

O problema proposto neste trabalho é o de analisar e categorizar métodos de ordenação, teórica e prática, observando o desenvolvimento e a eficiência desses métodos a partir de alguns cenários. No total, foram analisados 9 métodos conhecidos na computação em mais de 9 cenários diferentes, validando cada resultado, coletando, e comparando-os. Para isso, foi-se desenvolvido um programa em C++ capaz de implementar cada um desses cenários, testando cada algoritmo, e sendo exibido os seus resultados, que, após diversos testes, serão postos para análise, que poderá ser visto, para facilitar o entendimento do leitor, por meio de gráficos, esses que revelarão o comportamento de cada um desses algoritmos, contribuindo para a conclusão de seus métodos.

2. Implementação

O programa foi desenvolvido utilizando a linguagem C++, como já mencionado anteriormente, compilado pelo G++, em um notebook, modelo ideapad S145 da Lenovo, com um processador intel i5, 8gb de memória ram, utilizando WSL 2 com o Visual Studio Code.

2.1 Estrutura de Dados

Para o projeto, optei por utilizar apenas três bibliotecas C + + para compor minhas estruturas de dados.

Iostream: Biblioteca padrão da linguagem, essa que me deu a maioria das ferramentas que utilizei, dentre elas, a mais utilizada foram os arrays, esses que permitem armazenar múltiplos valores do mesmo tipo em uma única variável, as quais são acessadas a partir de um índice.
Um dos desafios do projeto era o de conseguir variar o tamanho, em bytes, dos elementos pertencentes a esse array, para isso, utilizei de templates, os quais permitem que seja possível editar o tamanho dos elementos, podendo variar entre 1 byte até 4 bytes. Além disso, também é importante destacar os conceitos utilizados para variar o tamanho do array, detalhes sobre tamanhos serão discutidos nos próximos tópicos.

cstdlib: Esta biblioteca fornece várias funções úteis para a manipulação de strings, matemática, alocação de memória dinâmica e, principalmente, geração de números aleatórios que seguem uma ordem de valores pré determinado. No contexto do trabalho, o maior destaque vai para a função “rand()”, essa que gera os números aleatórios, um dos critérios pedidos pelo professor para os métodos de ordenação, sendo esse o mais importante, na minha visão.

ctime: Esta biblioteca fornece funcionalidades relacionadas ao tempo e data. Ela inclui definições de tipos de dados para representar a data e a hora, bem como funções de manipulação de tempo. No contexto desse projeto, utilizei-a junto das outras para obter os resultados de cada algoritmo, sendo possível mencionar a função “time(nullptr)”, que mostra o

tempo atual em segundos a partir de uma data, e a função “clock()” essa que simula um cronômetro dentro de um método, possuindo início e fim, e retornando o tempo em segundos de cada execução.

2.2 Métodos e Classes

Para a execução do código, pensando em superar os desafios propostos, foram implementados dois métodos principais.

double medirTempoExecucao: Utilizando todas as bibliotecas mencionadas anteriormente, este método calcula o tempo de execução dos algoritmos, dentro de seu escopo, ele guarda o método que será utilizado,como o quicksort por exemplo, inicia o relógio e logo após aplica o algoritmo no array fornecido, após sua execução, ele para o relógio e retorna um double com os segundos. Cabe ressaltar que, para alguns métodos, a abordagem foi um pouco diferente, tendo que alterar alguns detalhes para sua funcionalidade.

void preencherArray: Utilizando o ctime e cstdlib este método foi criado para preencher os tipos abstratos de dados, de forma aleatória, com os elementos, nele é requisitado o tipo de dado, o tamanho, e o número de bytes. Dentro do seu escopo, ele utiliza funções como “time(nullptr)” e “rand()” para gerar os valores, além de contas para projetar os números a partir dos bytes solicitados, retornando o array, neste caso, preenchido. Assim como a função para medir o tempo de execução, este também teve que ser adaptado para alguns casos, outra observação a ser levada em conta, é que o método mostra quais valores foram adicionados, para que possam ser feitos testes posteriormente.

2.3 Métodos de Execução

Como mencionado na introdução, foram solicitadas algumas variações em relação ao tamanho do vetor, tamanho dos elementos armazenados, e a configuração inicial. Para a análise, o código permite adicionar flags que variam cada uma das configurações:

- Arrays Randomizados, com 8 bytes:
  Com mil, cinco mil e dez mil elementos.

- Arrays Randomizados, com 4 bytes:
  Com vinte mil, cinquenta mil, setenta e cinco mil e cem mil elementos.

- Arrays Ordenados, com 8 byte:
  Com mil, cinco mil e dez mil elementos.

- Arrays Ordenados, com 4 byte:
  Com vinte mil, cinquenta mil, setenta e cinco mil e cem mil elementos.

- Arrays Inversamente Ordenados, com 8 byte:
  Com mil, cinco mil e dez mil elementos.

- Arrays Inversamente Ordenados, com 4 byte:
  Com vinte mil, cinquenta mil, setenta e cinco mil e cem mil elementos.

3. Análise de Complexidade

As análises de complexidade de tempo dependem majoritariamente de dois fatores, o primeiro é o tamanho do array e sua configuração inicial, enquanto que, na complexidade do espaço, é determinante o número de elementos e o tamanho em bytes de cada um destes elementos. Além disso, também é possível analisar se o algoritmo é estável ou não.

Na computação, é visto que algoritmos seguem padrões de classificação quanto ao seu desempenho, como foi possível observar nos slides dados em sala de aula, é comumente aplicadas análises para o melhor caso, caso médio e o pior caso. Em geral, a diferença entre o melhor e o pior caso se dá pela configuração inicial do array de **N** elementos.

Por sua semelhança em relação à complexidade, pode-se classificar os métodos da bolha, de inserção e seleção como iguais, visto que, no melhor caso, onde o array já está ordenado, sua complexidade é linear ***O(*n*)*****,** contudo, para o pior caso, sua complexidade é quadrática ***O(*n^2*)*****,** o que se mantém no caso médio. Contudo, o método de seleção tem uma desvantagem por ser não estável, o que diferencia ele dos outros dois.

O Merge Sort tem complexidade de ***O(*$n \log n$*)*** no melhor, médio e pior caso, tornando-o uma escolha consistente para ordenação de grandes conjuntos de dados, além de que é um método estável, já que se baseia em divisão e conquista.

O Quick Sort é outro algoritmo de ordenação eficiente baseado na estratégia de divisão e conquista. No melhor e no caso médio, sua complexidade é ***O(*$n \log n$*)***, tornando-o comparável ao Merge Sort. Entretanto, no pior caso, quando a escolha do pivô é muito ruim, a complexidade pode degradar para ***O(*n^2*)***. Apesar disso, o Quick Sort é amplamente utilizado devido ao seu desempenho médio superior e sua implementação simples.

O Shell Sort é uma variação do método de seleção que melhora significativamente seu desempenho ao trocar elementos distantes antes de trocar elementos adjacentes. Sua complexidade no pior caso é ***O(n log^2 n)***, tornando-o mais eficiente que os métodos de ordenação quadráticos, mas geralmente menos eficiente que o os outros dois citados anteriormente. Outra desvantagem é que ele requer espaço extra proporcional à n, além de que, para $n/2^i$, ele não é estável.

O Counting Sort é um algoritmo de ordenação não comparativo adequado para classificar números inteiros dentro de um intervalo limitado. Sua complexidade é linear, ***O(n + k)***, onde n é o número de elementos e k é o intervalo dos valores. Isso o torna uma escolha eficiente para classificar grandes conjuntos de dados com uma faixa limitada de valores, além dele ser estável.

Tanto o Bucket Sort quanto o Radix Sort são algoritmos que têm complexidade linear no melhor e no caso médio, mas o pior caso do Bucket Sort pode ser de ***O(*n^2*)*** se todos os elementos forem colocados no mesmo

balde. O Radix Sort tem complexidade linear no pior caso, mas pode exigir mais memória que o Bucket Sort.

4. Estratégia de Robustez

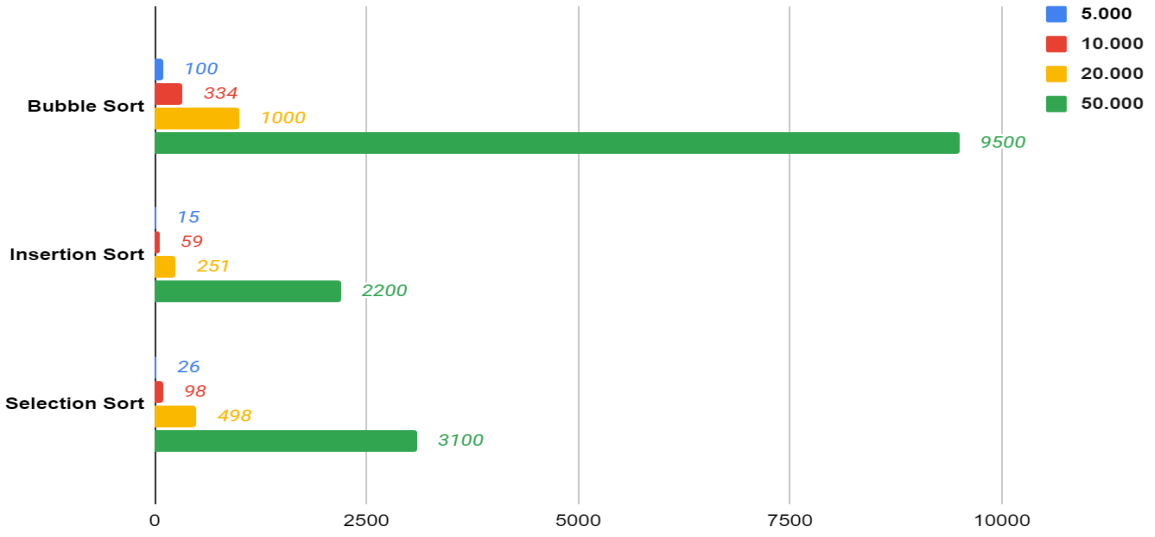
Uma das estratégias de robustez utilizadas é a de clonar os arrays, a partir de uma função, todos os arrays, independentemente de como estejam configurados, são testados, para verificar se os elementos e os tamanhos foram os solicitados pelo usuário, e logo após ele cria no buffer um array cópia, esse que é utilizado para restaurar o estado original antes, durante e depois da aplicação de algum método. Além disso, foi-se desenvolvido estratégias para alocar corretamente o array e verificar se a manipulação das chaves/índices foram feitas de forma correta, a fim de evitar erros de segmentação ou compilação, outra estratégia utilizada foi a de testar todos os métodos, isso só foi possível com a biblioteca “gtest”, que foi aplicada para testar os comportamentos de todos os nove algoritmos.

5. Análise Experimental

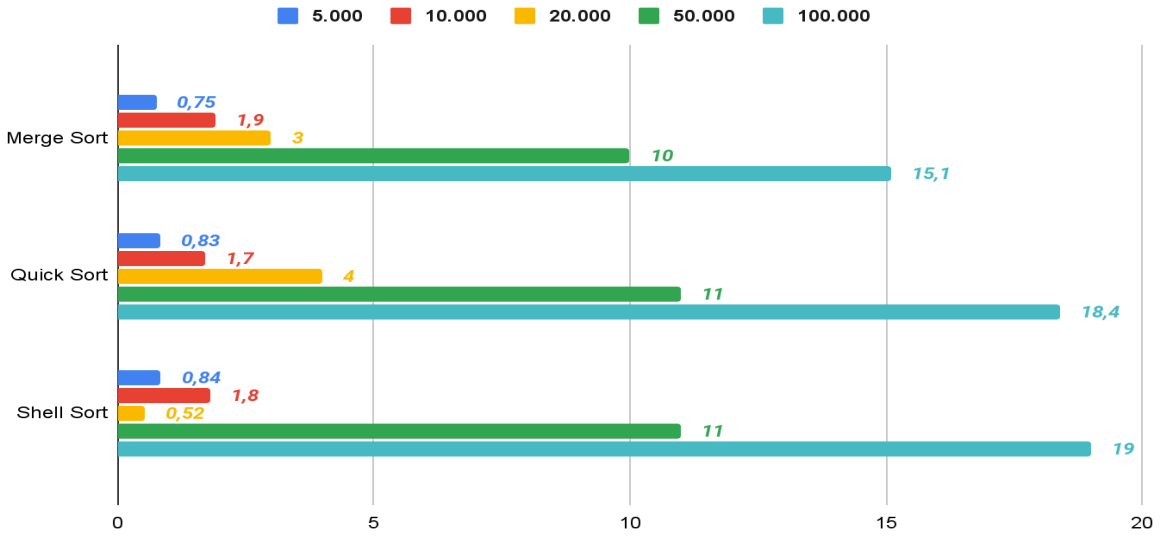
Visando melhorar a visualização, as análises de comparação serão organizadas pela análise de complexidade de cada uma, pois, quando se compara o método da bolha com o "radix sort", principalmente para arrays grandes e aleatórios, os valores dos índices ficam discrepantes e prejudicam a análise. O eixo vertical representa os métodos, o horizontal representa o tempo em milissegundos. As legendas das cores revelam o número de elementos de que foram utilizados.

5.1 Arrays Com Valores Randômicos Baseados no Número de Elementos

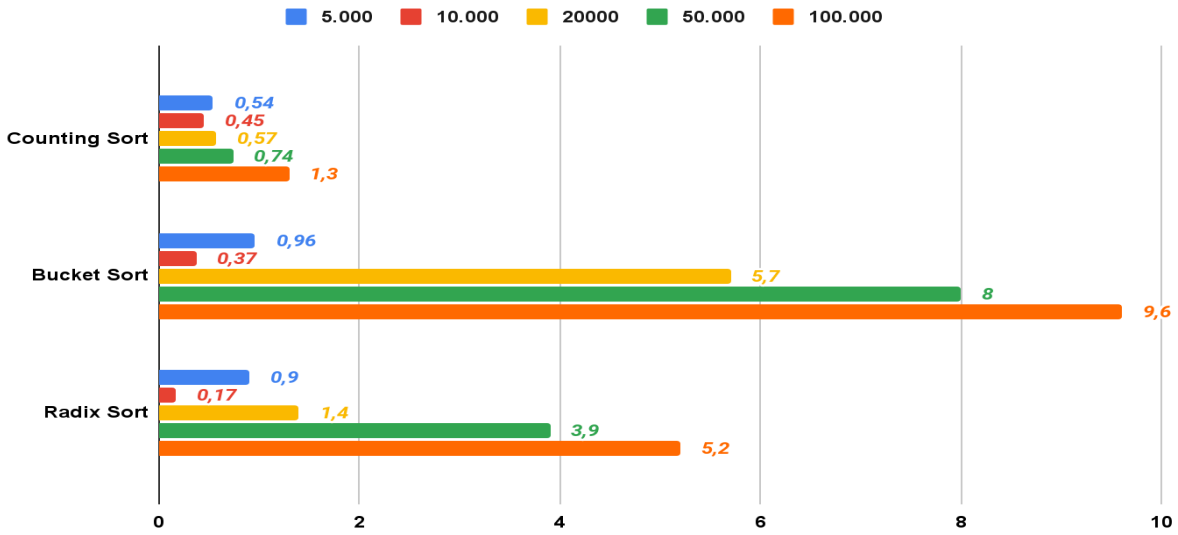
O(n^2) **:**



O$(n \log n)$ **:**

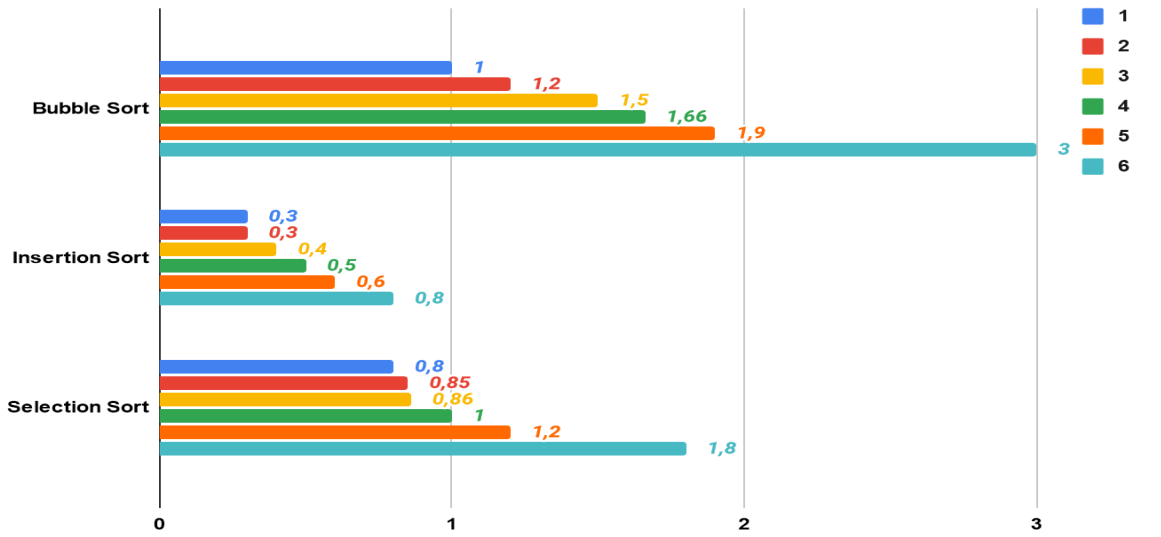


Complexos:

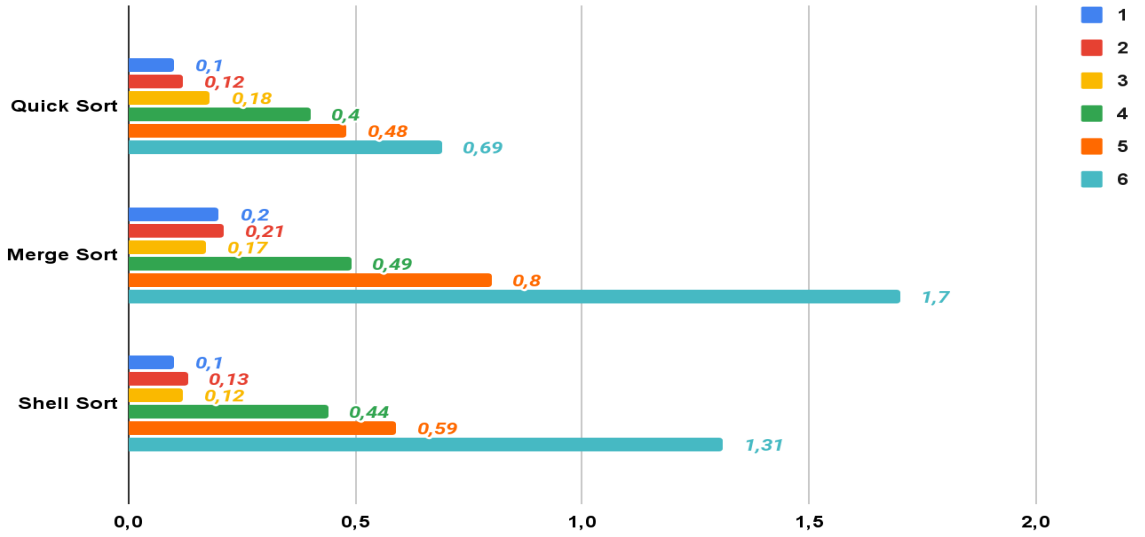


5.2 Arrays Com Valores Randômicos Baseados no Tamanho dos Elementos em Bytes

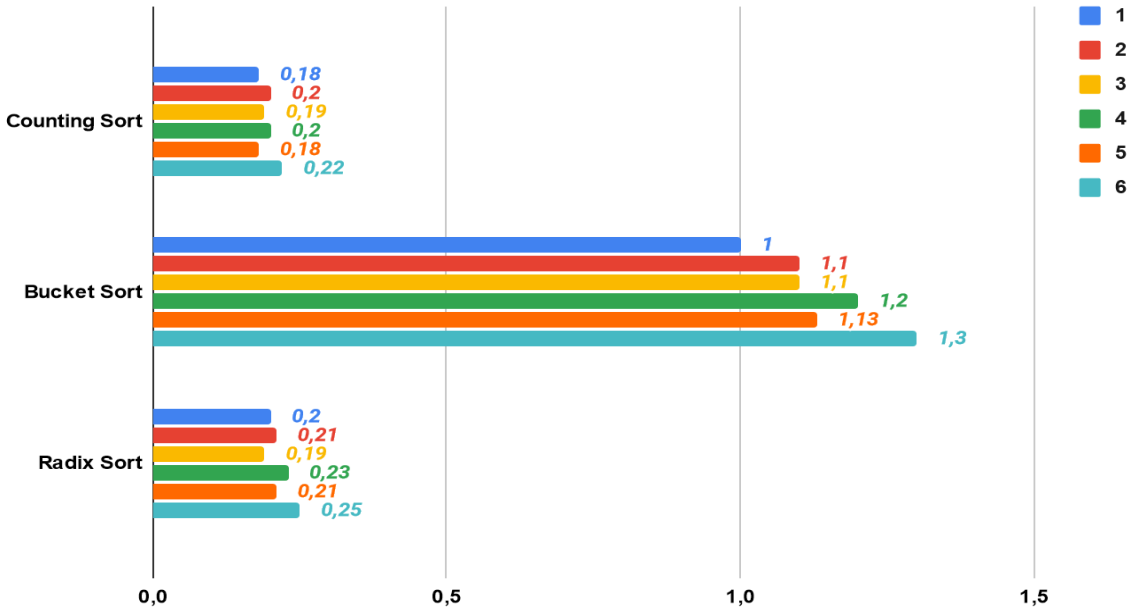
***O*(n^2) :**



$O(n \log n)$:



Complexos:



5.3 Análise Dos Gráficos Por Número de Bytes e Quantidade de Elementos

A partir das imagens, é possível ver que os algoritmos $O(n^2)$ são fortemente impactados pelo número de elementos do array, isso ocorre porque eles têm muitas comparações e movimentações durante o processo do algoritmo inclusive, visitam posições no array mais de uma vez, o que faz com que os três tenham a características de serem diretamente proporcionais ao tamanho do vetor, e que também acontece com o número de bytes, que aumenta com o tamanho de bytes.

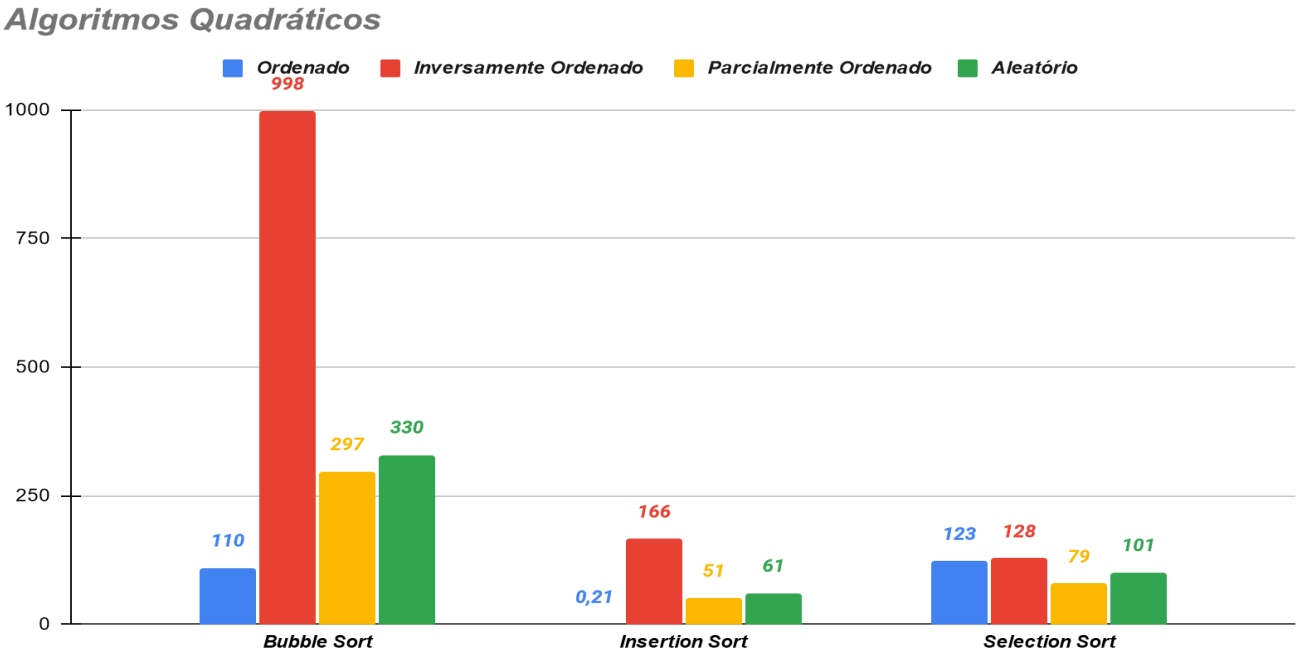
Nos algoritmos $O(n \log n)$ a abordagem é diferente, como pode ser observado, eles são mais eficientes do que os algoritmos de ordenação quadrática, isso ocorre porque eles, de forma leviana, dividem para conquistar, isto é, eles tem a estratégia de criar sub vetores e ordená los de forma separada isso faz com que eles sejam mais eficientes e resistentes quanto ao tamanho dos elementos. Contudo, esses algoritmos solicitam memória extra, o que pode gerar um gasto computacional maior.

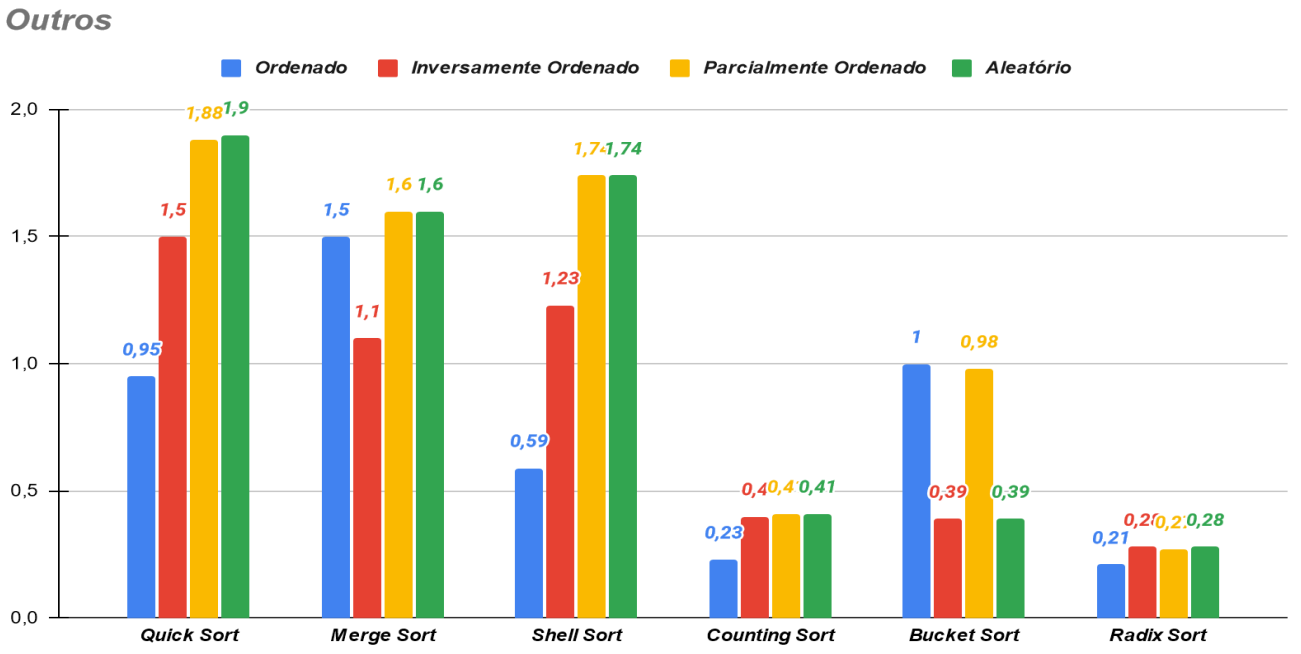
Os métodos de ordenação sem comparação são mais peculiares, o Counting sort e o Bucket sort são sensíveis com arrays mal ordenados e com arrays de vários elementos, enquanto o Radix aparenta ser um pouco mais resistente. Contudo, eles são pouco influenciados em relação ao tamanho do número que armazenam, sendo quase que estáveis mesmo com o aumento de bytes.

5.4 Análise Por Ordenação dos Vetores

Os gráficos a seguir analisam os algoritmos por sua estrutura, seguindo os moldes anteriores, com a representação em milissegundos, com cada ordenação presente na legenda (Ordenado, Inversamente Ordenado, Aleatório e Parcialmente Ordenado).

Utilizei 10 mil elementos para cada, com 4 bytes de tamanho, e com o mesmo array para todos os algoritmos.





Para esta análise, para que fosse possível observar com mais clareza as comparações de tempo, optei por separar os três primeiros algoritmos do restante, visto que a proporção seria de, no mínimo, 100 vezes maior para qualquer modo de ordenação.

Nessa abordagem, torna-se mais condizente analisar os gráficos separando por estilo do array, e não pelo conjunto dos algoritmos como feito nos dois tópicos anteriores.

Para os ordenados, o destaque negativo é do Bubble Sort e do Selection Sort, visto que eles visitam todo o array da mesma maneira e fazem todas as comparações, diferentemente do Insertion Sort que, entre todos os algoritmos, é o melhor. Os outros possuem bom desempenho, podendo se dizer que são estáveis.

Para os inversamente ordenados, o Bubble e o Insertion foram os piores, visto que estão no pior caso, enquanto que os algoritmos de contagem, Radix, Bucket e Counting, foram os melhores, justamente por não terem sido tão afetados quanto à ordem presente dos elementos.

Para os parcialmente ordenados, os algoritmos que comparam foram péssimos, enquanto os outros foram mais estáveis e sem muita variação, com o destaque indo para o Radix Sort.

E finalmente, para os aleatórios, os algoritmos de divisão e conquista foram na média, estáveis, enquanto que os de contagem se propuseram melhor nessa modalidade.

6 Conclusões

A partir do trabalho, foi se possível tirar várias conclusões interessantes e ver na prática o que foi falado em sala de aula, sendo algumas delas:

- Cada algoritmo tem sua propriedade, vantagens e desvantagens, em uma implementação real é necessário conhecer como são as propriedades dos vetores. Os algoritmos de contagem, por exemplo, que se destacaram na última análise, apresentam problemas em arrays extremamente grandes e o Bucket, em especial, precisa ter um bom critério de divisão para ir bem, enquanto que o Merge e o Quick apresentam estabilidade na eficiência, porém, são “caros” computacionalmente, enquanto que os quadráticos tem um desempenho reduzido e quase que linear ao aumentar o tamanho de elementos e do array, porém, para vetores ordenados, se saíram bem.
- Todos os fatores unidos tem grande impacto nas análises, isto quer dizer que, não basta apenas olhar o tamanho do array, ou somente o número de bytes dos elementos, mas sim o conjunto de ordenação, tamanho e quantidade.
- Os algoritmos de contagem foram os mais complexos para serem construídos, foram os que mais apresentaram erros quando foram colocados em situações extremas, principalmente de tamanho de elementos.
- Um fator muito importante na hora de considerar os algoritmos pode ser a estabilidade, e neste quesito, o Merge, Counting e Radix foram

os melhores, por terem boa eficiência e não alterarem as chaves ao longo da ordenação

- Para conjuntos pequenos de elementos, o Counting e o Shell podem ser os mais racionais, enquanto que para grandes, Radix, Quick e Merge, dependendo da memória do computador, podem ser boas escolhas.
- O que mais me surpreendeu nesta análise foi o Insertion Sort para arrays já ordenados.

7 Bibliografia

Slides em sala de aula.
Vídeos disponibilizados no moodle.
Algoritmos - Cormem.