Trabalho Prático 1 - Estrutura de Dados

Lucas Ribeiro da Silva - 2022055564
Universidade Federal de Minas Gerais
Belo Horizonte - Minas Gerais - Brasil

lucasrsilvak@ufmg.br

1 Introducão

O problema proposto foi analisar experimentalmente os métodos de ordenação estudados em sala de aula e constatar a complexidade teórica de cada método, levando em consideração as especificidades de cada algoritmo e também as variações de cada sequência de dados.

2 Método

2.1 Configurações de Máquina e Ambiente

O programa foi desenvolvido na linguagem C e compilado pelo GCC da GNU Compiler Collection. O computador e programas utilizados tem as seguintes especificações:

- Windows 11 Home Single Language
- WSL 2 com Ubuntu 22.04.4 LTS

2.2 Estrutura de Dados

Na implementação desse trabalho, a estrutura de dados utilizada foram as arrays *(array[])*, que terão seus dados ordenadas pelos algoritmos previamente estudados.

2.3 Métodos de Ordenação

Aqui estão os métodos de ordenação implementados:

BubbleSort() O BubbleSort foi implementado com uma flag para identificar se o vetor está previamente ordenado.

InsertionSort() O InsertionSort foi implementado em sua forma canônica.

SelectionSort() O SelectionSort foi implementado em sua forma canônica.

MergeSort() O MergeSort foi implementado em sua forma canônica.

ShellSort() O ShellSort foi implementado em sua forma canônica com um *gap* de 3.

QuickSort() O QuickSort foi implementado com o esquema de partição Hoare, otimizando a pivotação com a mediana de 3.

CountingSort() O CountingSort foi implementado em sua forma canônica.

BucketSort() O BucketSort foi implementado em sua forma canônica, com um BucketCount definido como *ceil(sqrt(Length))* e um InsertionSort para ordenar os Buckets.

RadixSort() O RadixSort foi implementado com a introdução de um QuickSort seguindo o modelo apresentado na aula.

3 Análise de Complexidade

3.1 Tempo

BubbleSort() O BubbleSort caracteriza-se por movimentar os elementos do vetor, comparando-os com os restantes elementos em casas superiores, substituindo caso o inicial seja maior, como sua implementação requer dois laços de repetição aninhados, fazendo com que sua complexidade seja $O(n^2)$.

InsertionSort() O InsertionSort caracteriza-se por partir da posição inicial e percorrer o vetor, comparando as casas posteriores com as últimas até que o objeto adicionado ao vetor tenha seu valor menor que o

valor comparado. Sua implementação requer dois laços de repetição aninhados e logo a complexidade é $O(n^2)$.

SelectionSort() O SelectionSort caracteriza-se por obter o maior elemento do vetor e movimentá-lo para a última posição não modificada, e então, repetir o processo para todos os elementos do vetor. Sua implementação requer dois laços de repetição aninhados e sua complexidade é $O(n^2)$.

MergeSort() O MergeSort caracteriza-se por utilizar da ideia de divisão e conquista, e para isso, particiona repetidamente os vetores em partições menores até o tamanho de 1 e então ordena o vetor principal a partir das partições criadas. Particionar os vetores leva um tempo de $O(\log n)$, enquanto ordenar o vetor a partir das partições leva um tempo $O(n)$, como os processos não são independentes, sua complexidade é $O(n \log n)$.

ShellSort() O ShellSort utiliza-se da ideia de divisão e conquista, particionando o vetor original em pedaços cada vez menores e ordenando-os. Particionar os vetores leva um tempo $O(\log n)$, enquanto ordenar o vetor a partir das partições leva um tempo $O(n)$, como os processos não são independentes, sua complexidade é $O(n \log n)$.

QuickSort() O QuickSort utiliza-se da ideia de usar um pivô para particionar o vetor, e sua complexidade depende da escolha do pivô, se o pivô divide o vetor praticamente no meio, a divisão e conquista é bem sucedida e a complexidade será da ordem de $O(n \log n)$, entretanto, se o pivô for repetidamente um valor extremo as partições serão mal divididas e a complexidade do algoritmo será $O(n^2)$.

CountingSort() O CountingSort caracteriza-se por utilizar-se do valor máximo do vetor (k) para organizar o vetor original em partições iguais onde serão contabilizados o número de repetições e então monta o vetor original conforme o número de repetições. Tem um laço de repetição para percorrer o vetor inicial e sua complexidade torna-se $O(n + k)$.

BucketSort() O BucketSort é uma variação do CountingSort onde as partições não são únicas para cada valor, mas aceitam um *range* de valores que serão ordenados posteriormente por um InsertionSort. Se as partições são bem divididas, a complexidade será $O(n)$, mas se forem muito longas o InsertionSort dominará e a complexidade será $O(n^2)$.

RadixSort() O RadixSort implementado é uma variação do QuickSort onde o pivô são bem escolhidos: são os próprios números binários. Sua complexidade é $O(n \log n)$.

3.2 Espaço

BubbleSort() O BubbleSort não aloca memória além da inicial e por isso é da ordem $O(1)$.

InsertionSort() O InsertionSort utiliza-se somente de uma variável temporária e por isso, também não aloca memória além da inicial, sendo da ordem $O(1)$.

SelectionSort() O SelectionSort também não aloca memória além da inicial e por isso é da ordem $O(1)$.

MergeSort() O MergeSort requer um espaço adicional do tamanho do inicial para armazenar as partições a serem mescladas, logo é da ordem de complexidade de $O(n)$.

ShellSort() O ShellSort, apesar de particionar o vetor, não tem a necessidade de alocar nova memória, operando na ordem $O(1)$.

QuickSort() O QuickSort também tem a complexidade de espaço baseada em seu pivoteamento, se o pivô for bem escolhido, será da ordem $O(\log n)$, mas pode degradar para $O(n)$ com pivôs mal escolhidos.

CountingSort() O CountingSort necessita de alocar memória baseado no valor máximo do vetor para fazer a contagem depois, logo sua ordem é de $O(n + k)$.

BucketSort() O BucketSort é semelhante ao CountingSort, mas aloca a memória em relações ao número de baldes (m) ao invés do valor máximo do vetor, logo sua complexidade será do tipo $O(n + m)$.

RadixSort() O RadixSort implementado é uma variação do QuickSort onde o pivô é sempre bem escolhido: são os próprios números binários. Sua complexidade de memória é $O(\log n)$.

4 Análise de Robustez

Para incrementar a legibilidade do código, os métodos foram devidamentes padronizados utilizando PascalCase e com as variáveis nomeadas em inglês.

Para aumentar a robustez dos algoritmos, foram adicionadas *flags* que podem diminuir substancialmente o uso de tempo dos algoritmos.

Além disso, foi criada uma biblioteca auxiliar *utils.h* para auxiliar a maximizar o número de testes e para comprovar o funcionamento do algoritmo inclusive em seus casos de borda.

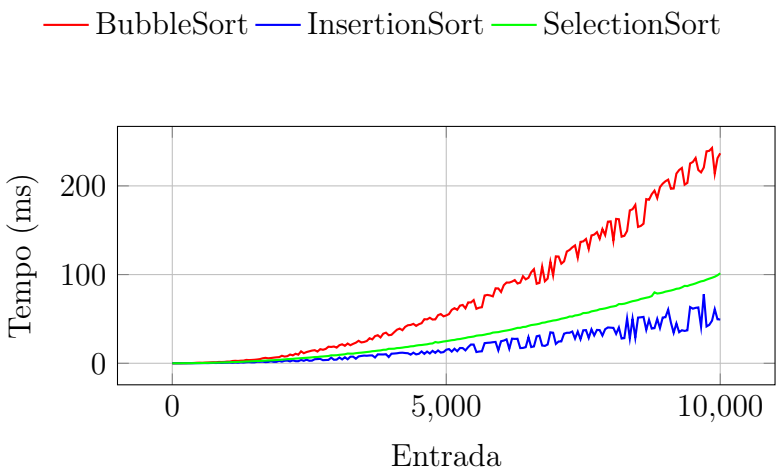
5 Análise Experimental

5.1 Complexidade Experimental

Nessa análise testaremos os algoritmos para os tipos mais variados de entradas, desordenadas, ordenadas crescentemente, decrescentemente, com dados de amplitudes e tamanhos diferentes.

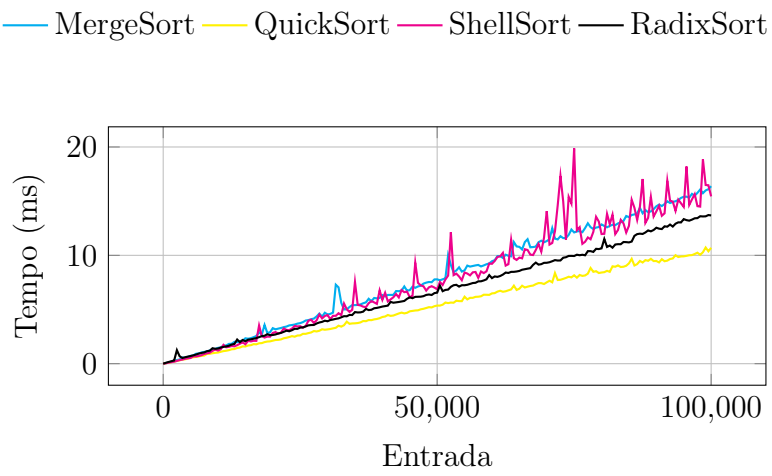
5.1.1 $0(n^2)$

Os algoritmos de ordem quadrática tendem a subir o tempo de execução rapidamente conforme o número de entradas aumenta.



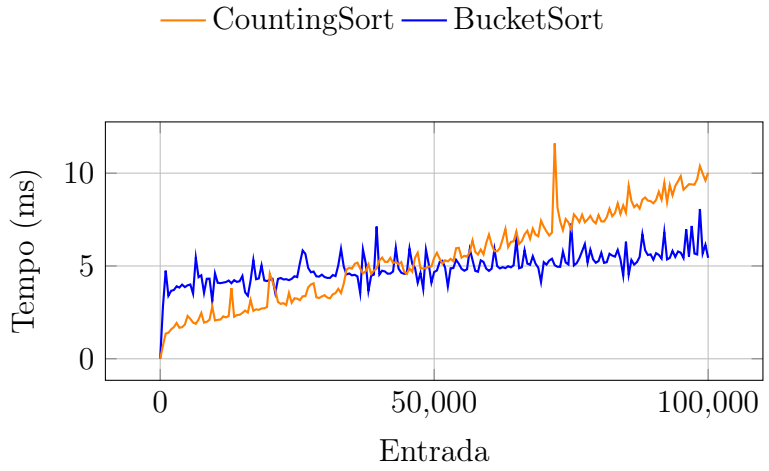
5.1.2 O(n log n)

Os algoritmos de ordem n log n estudados tendem a ser mais rápidos para números de entradas razoáveis além de manter relativa estabilidade.



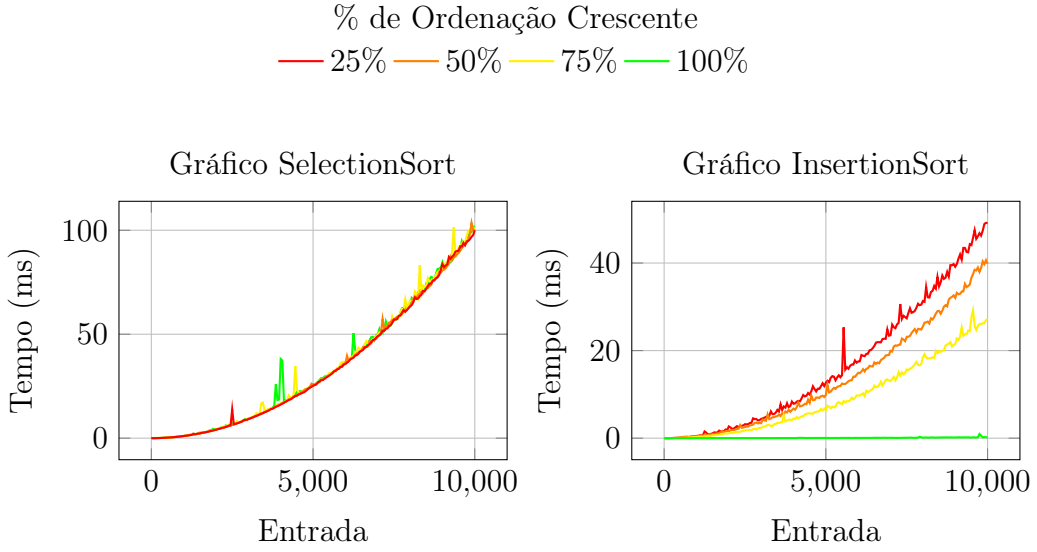
5.1.3 O(n)

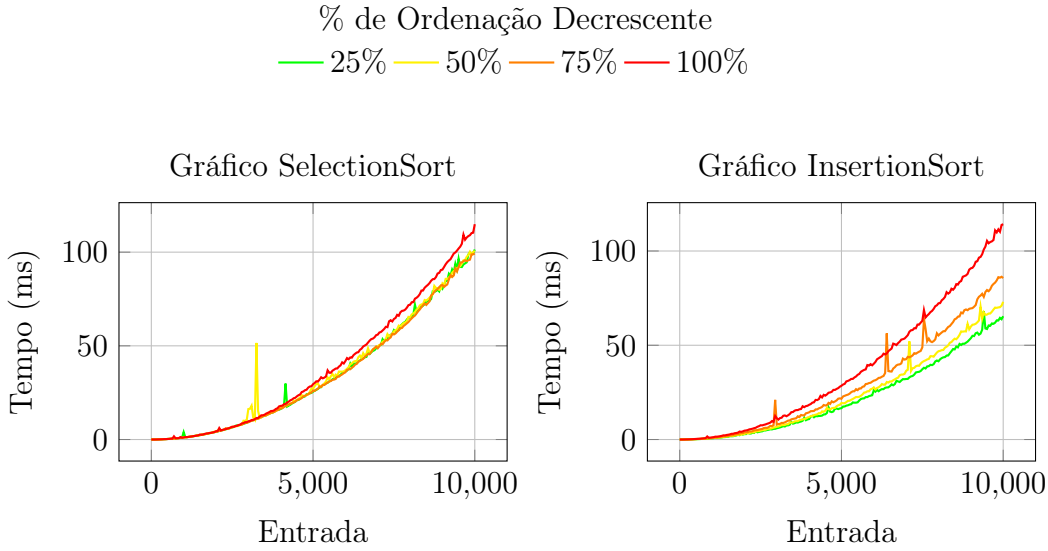
Algoritmos de ordem linear tendem a ser extremamente rápidos, mas enfrentam limitações quanto a amplitude dos dados. No exemplo do gráfico, amplitude utilizada de 10^7.



5.2 Disposição dos Dados

Nessa análise observaremos como os algoritmos se comportam dado a disposição dos dados num determinado vetor.

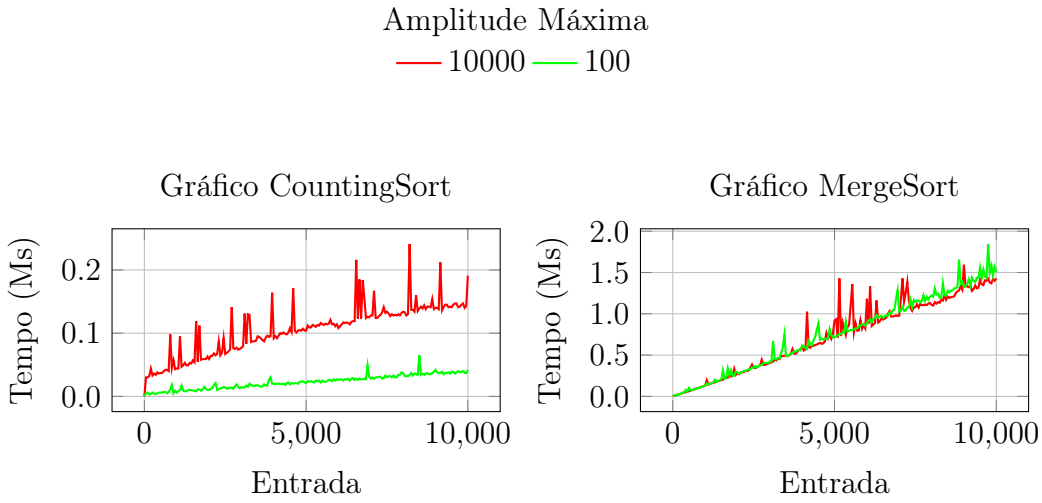




Podemos ver, pelos gráficos que algoritmos como o SelectionSort tendem a ser mais estáveis conforme a ordenação dos dados, enquanto o InsertionSort tende a melhorar proporcionalmente a ordenação crescente dos dados.

5.3 Quantidade de Dados Diferentes

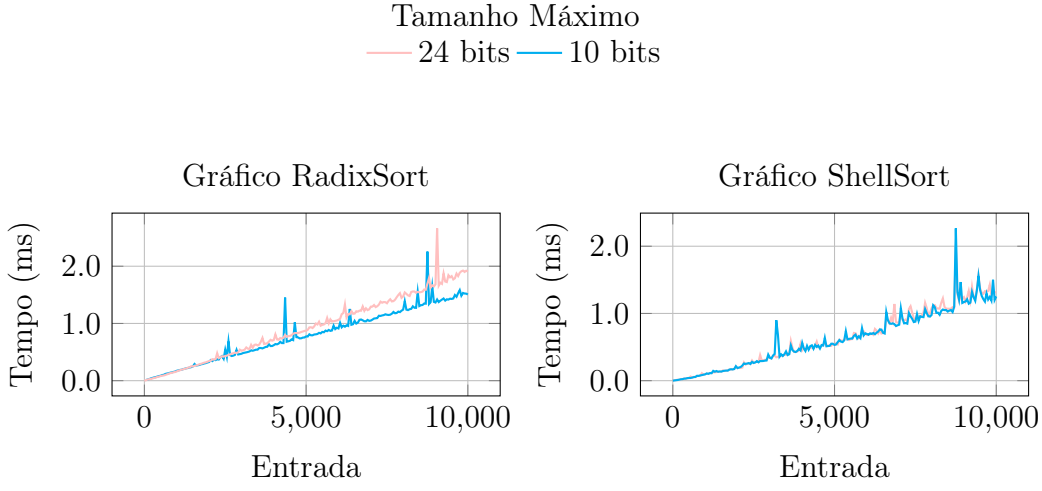
Nessa análise observaremos os algoritmos dado a amplitude dos dados.



Podemos observar que algoritmos sem comparação pioram com a a amplitude dos dados, enquanto o MergeSort permanece praticamente constante.

5.4 Tamanho dos Dados

Nessa análise observaremos os algoritmos dado o tamanho dos dados.



Podemos observar que algoritmos como o RadixSort tendem a piorar conforme o tamanho dos dados aumenta, enquanto o ShellSort permanece praticamente constante.

6 Conclusões

O trabalho lidou com diferentes métodos de ordenação com dados em diferentes situações com o intuito de obter análises experimentais sobre os algoritmos utilizados, ampliando os conhecimentos sobre complexidade e análise de algoritmos.

O trabalho permitiu entender experimentalmente as análises teóricas feitas em sala de aula e reforçar a ideia de que não há algoritmo ótimo e todos tem seus usos e aplicações nas mais diversas situações distintas, dependendo de fatores como número de entrada, organização dos dados, tipos de dados, entre outros.

7 Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides da Disciplina de Estruturas de Dados, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte. Disponibilizado via https://virtual.ufmg.br/