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[])* de inteiros 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 *passo* 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 $O(n \times b)$ onde b é o número de bits.

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, como é o caso do BubbleSort() e nos métodos de ordenação que requerem alocamento de memória foram utilizados métodos de alocação dinâmica, como *calloc* e *malloc*. Além disso, foi criada uma biblioteca auxiliar *utils.h* que possui métodos para verificar se o vetor foi ordenado e auxiliar a maximizar o número de testes e comprovar o funcionamento do algoritmo inclusive em seus casos de borda.

5 Análise Experimental

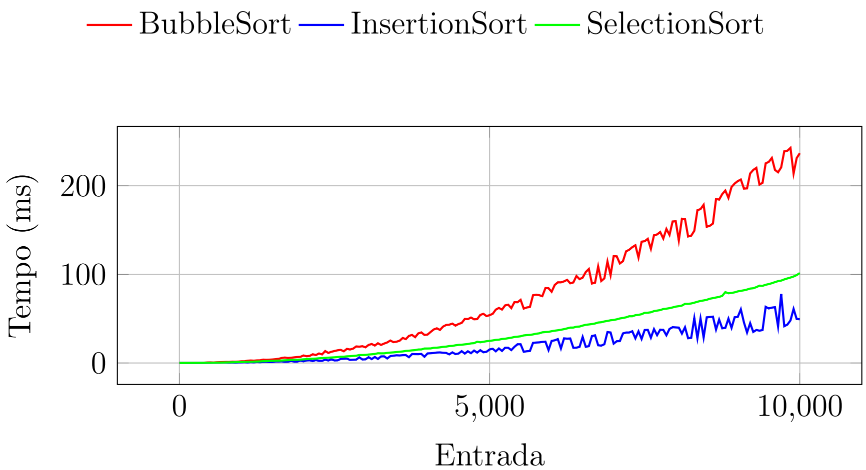
5.1 Complexidade Experimental

Na análise de Complexidade Experimental testaremos os nove algoritmos para os tipos mais variados de entradas, com os gráficos para O(n^2), O(n log n) e O(n) possuindo entradas de ordenação variada (entre 0-99%), amplitudes de dados variadas (entre 100 e 1000000) e tamanho de entrada variada (entre 2^6 e 2^{24} bits).

Depois, compararemos a resposta de alguns algoritmos dado as variações entre os tipos de entrada e analisaremos sua estabilidade e seus melhores e piores casos.

5.1.1 0(n^2)

Os algoritmos de ordem quadrática tendem a subir o tempo de execução de maneira quadrática conforme o número de entradas aumenta, isso os torna extremamente ineficientes com valores maiores de entrada.

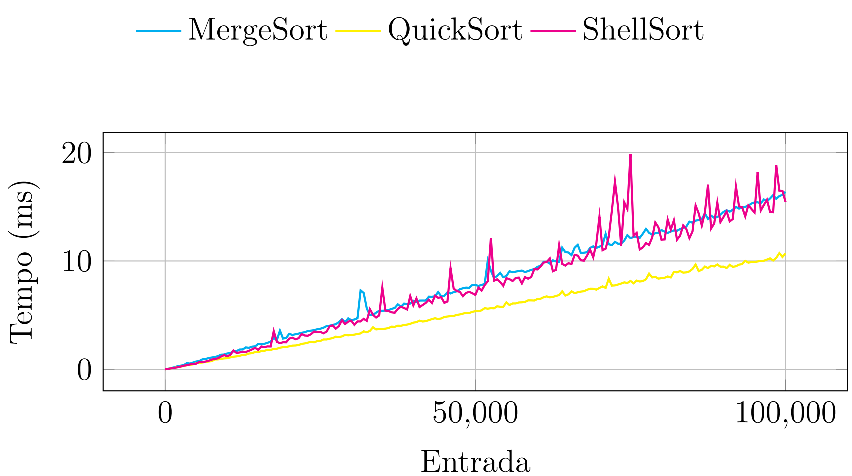


É possível perceber que o SelectionSort() permaneceu estável embora as cargas sejam variadas o que é comprovado pela teoria. Podemos perceber que o InsertionSort() obteve variações significantes, o que pode que a variação da ordenação foi percebida.

Outra conclusão possível é que entre os algoritmos quadráticos o BubbleSort() tem um pior desempenho, o que indica constantes maiores e indica menor desempenho para entradas pequenas.

5.1.2 O(n log n)

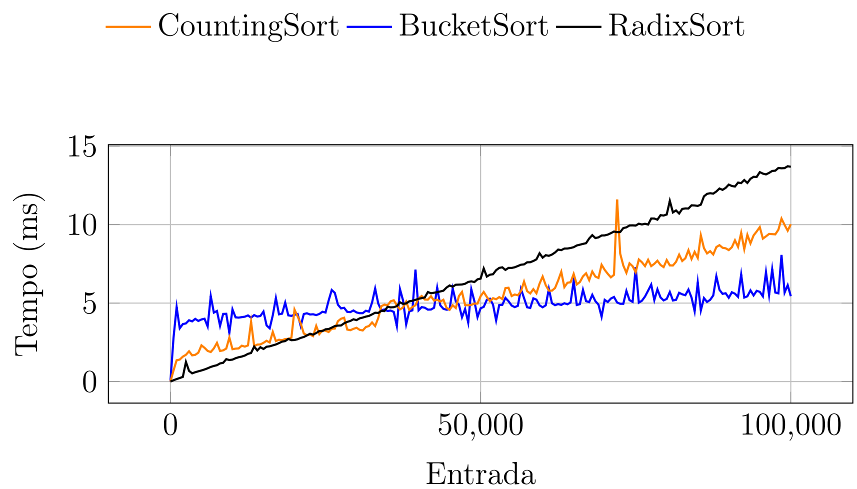
Os algoritmos de ordem n log n estudados tendem a ser relativamente rápidos para números de entradas razoáveis além de manter relativa estabilidade.



É possível observar que os algoritmos MergeSort() e ShellSort() tem um desempenho parecido, embora o ShellSort() tenha tido variações maiores em relação as cargas diferentes de entradas. O QuickSort() foi bem mais rápido e estável que os demais algoritmos, o que demonstra uma maior robustez e constantes menores.

5.1.3 O(n)

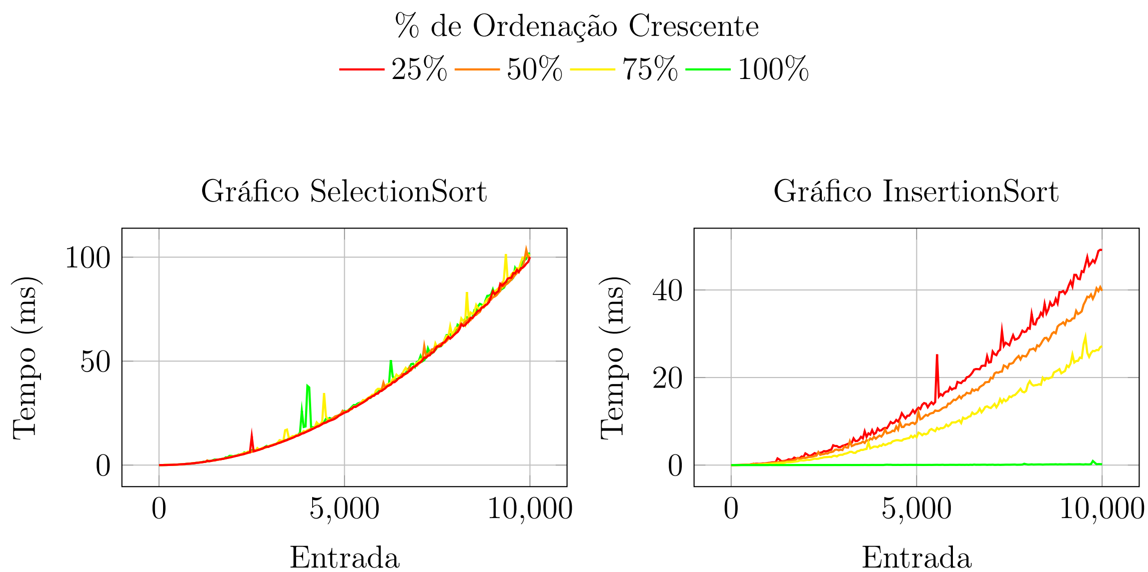
Algoritmos de ordem linear tendem a ser extremamente rápidos, mas enfrentam limitações quanto a amplitude dos dados. No exemplo do gráfico, a amplitude foi configurada em 10^7.

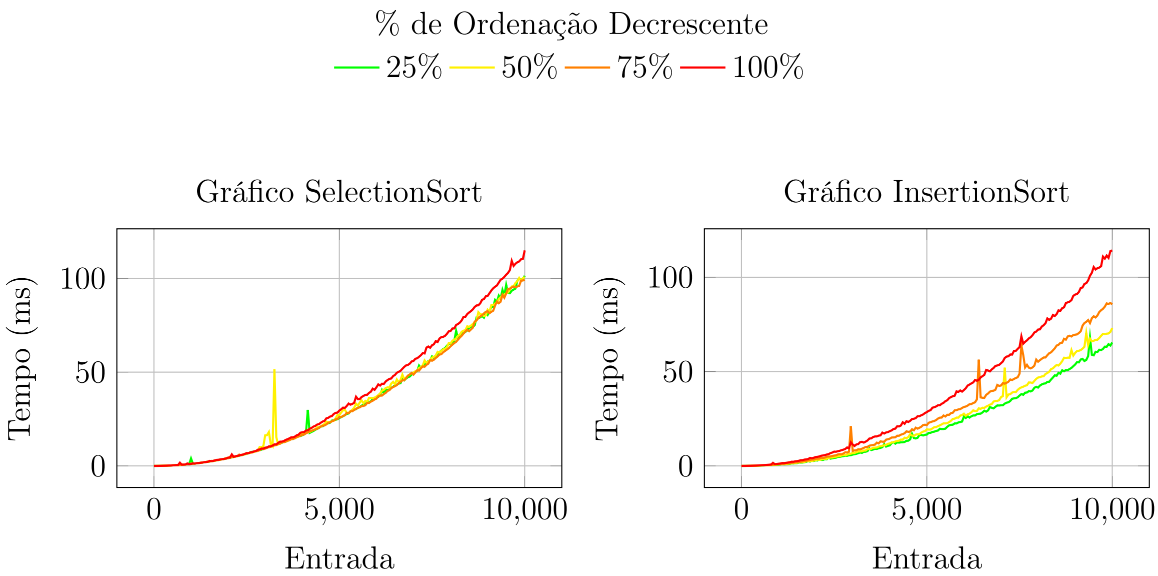


É possível observar que o CountingSort() e o RadixSort() aumentaram consideravelmente o tempo de execução conforme a entrada, enquanto o BucketSort(), apesar de ter começado com um tempo de execução pior, manteve praticamente o mesmo intervalo de tempo durante toda a execução, o que pode indicar que o número de baldes foi ideal para o intervalo de entradas escolhido.

5.2 Disposição dos Dados

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

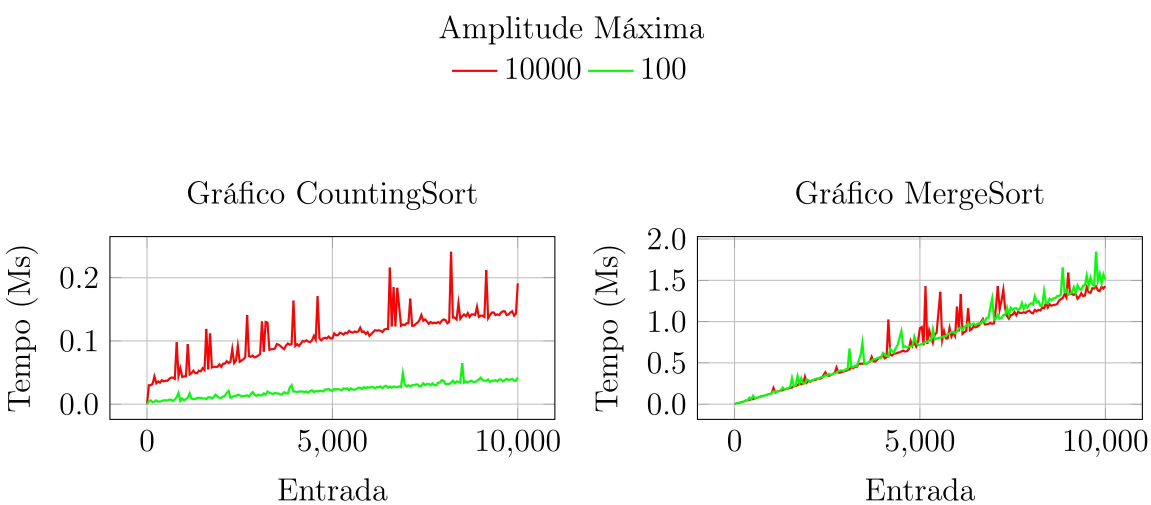




Podemos ver, pelos gráficos que algoritmos como o SelectionSort() tendem a ser mais estáveis conforme a ordenação dos dados, o que é comprovado pela teoria, pois realiza sempre as mesmas operações, enquanto o InsertionSort() tende a melhorar proporcionalmente a ordenação crescente dos dados, o que também é comprovado pela teoria, pois realiza menos operações quando está ordenado positivamente.

5.3 Quantidade de Dados Diferentes

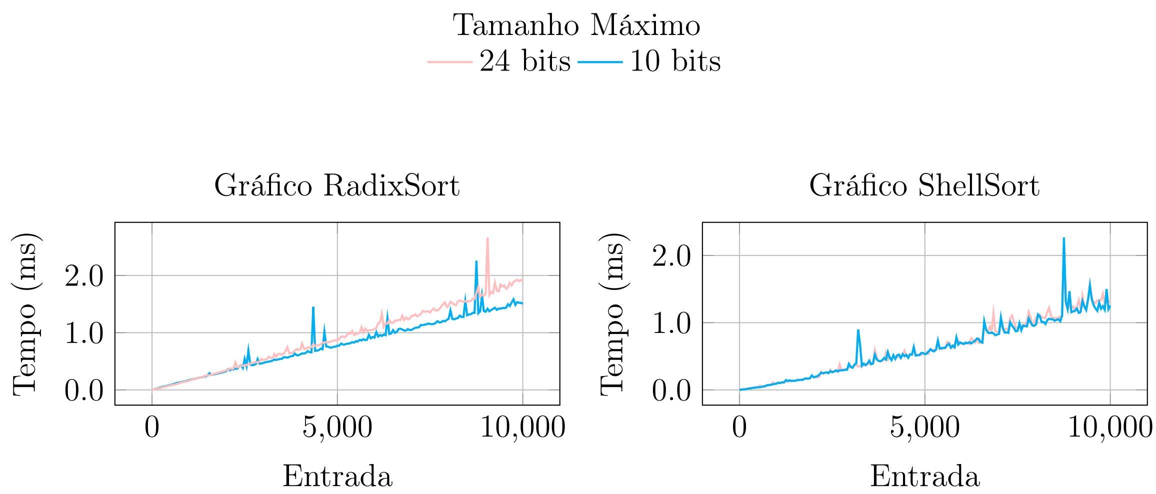
Nessa análise observaremos os algoritmos dado a amplitude dos dados.



Podemos observar que algoritmos como o CountingSort() pioram com a a amplitude dos dados, o que é comprovado pela teoria, pois o CountingSort() depende do máximo valor do conjunto de dados, enquanto isso o MergeSort() permanece praticamente constante, o que era esperado devido sua natureza estável.

5.4 Tamanho dos Dados

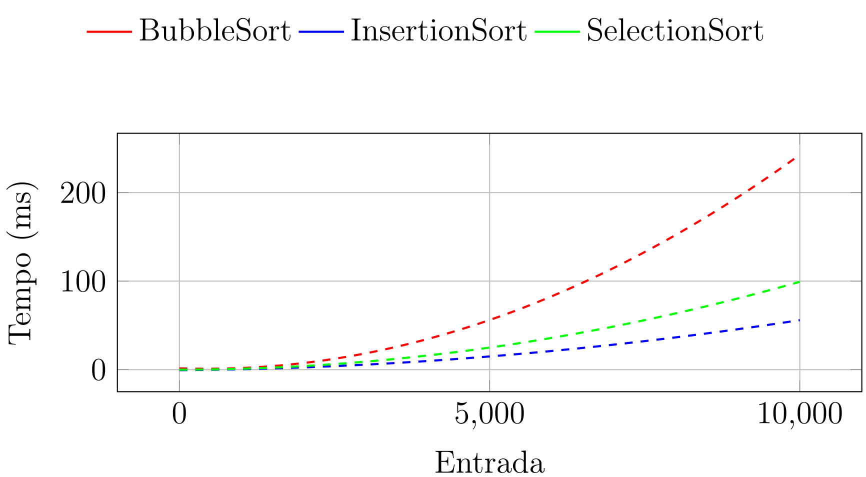
Nessa análise observaremos os algoritmos dado o tamanho dos dados.

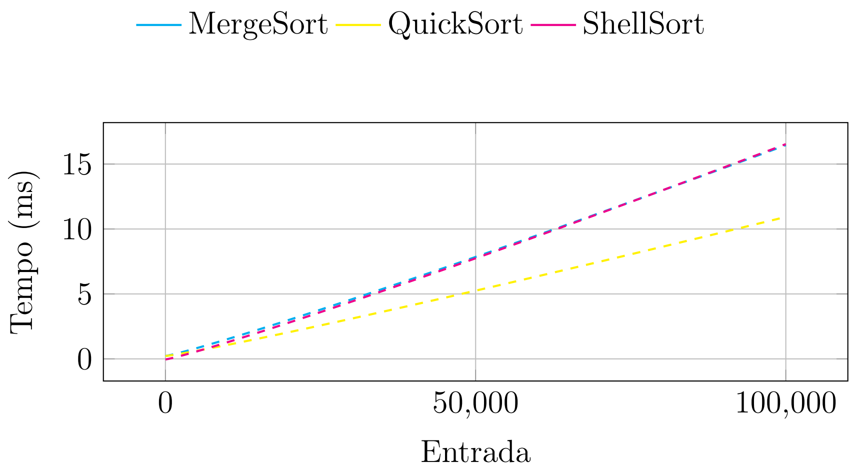


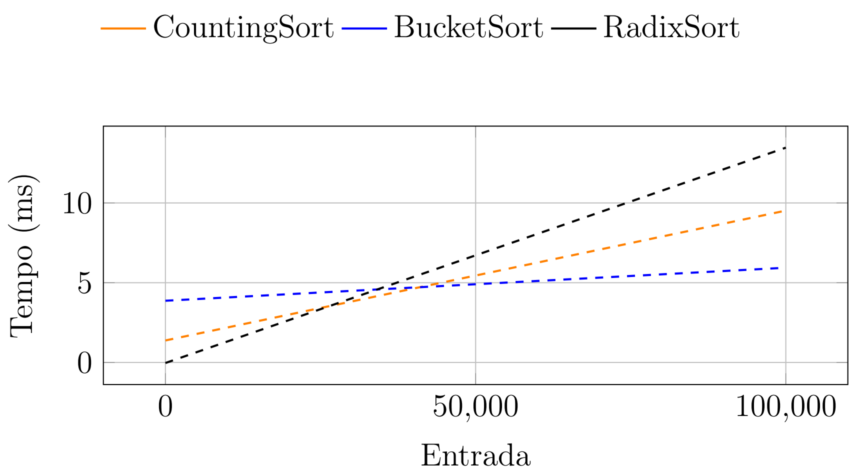
Podemos observar que algoritmos como o RadixSort() tendem a piorar conforme o tamanho dos dados aumenta, o que é comprovado pela teoria, pois o número de bits é um dos parametros da complexidade do mesmo, enquanto o ShellSort() permanece praticamente constante, o que era esperado.

5.5 Ajuste de Curvas

Para os códigos observados, utilizaremos curvas para aproximar as funções experimentais observadas utilizando o método de Ajuste de Curvas e calcularemos o Coeficiente de Determinação (R^2).







5.5.1 Coeficientes

Método	R^2	β_2	β_1	β_0
$Tempo(n) = \beta_2 n^2 + \beta_1 n + \beta_0\ ms$				
BubbleSort	0,99311	2,64e-6	-2,28e-3	1,47
InsertionSort	0,91380	5,1e-7	5,49e-4	-5,89e-1
SelectionSort	0,99985	9.86e-7	6.56e-5	-5,49e-2
$Tempo(n) = \beta_1 n \ln n + \beta_0\ ms$				
MergeSort	0,99400	-	1,41e-5	2,21e-1
QuickSort	0,99630	-	9,3e-6	2,21e-1
ShellSort	0,94989	-	1,44e-5	-4,54e-2
$Tempo(n) = \beta_1 n + \beta_0\ ms$				
CountingSort	0,95406	-	8,14e-5	1,38
BucketSort	0,48346	-	2,07e-5	3,87
RadixSort	0,99766	-	1,35e-4	-3,35e-2

Tabela 1: Tabela de R^2 e coeficientes $\beta_2, \beta_1, \beta_0$ para métodos de ordenação.

Pelo Ajuste de Curvas foi possível identificar as constantes dos algoritmos, é possível também observar que no caso de algoritmos muito flutuantes, o ajuste tende a não ser confiável, como por exemplo no caso do BucketSort, enquanto no caso do SelectionSort, o ajuste é extremamente preciso por causa da alta estabilidade do mesmo.

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 e comprovando a complexidade já anteriormente estudada por meio de experimentos.

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 distintas situações, dependendo de fatores como número de entrada, organização dos dados, tipos de dados, utilização de *flags*, métodos diferentes, entre outros.

7 Bibliografia

Referências

[1] 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 em: https://virtual.ufmg.br/

[2] Campos Filho, F. (2007). *Algoritmos Numéricos*. 2ª edição. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte.