Trabalho Prático 2
Algoritmos e Estrutura de Dados 2

Geraldo Arthur Detomi, Rhayan de Sousa Barcelos, Rodrigo José Isidoro Junior

1 Introdução

O objetivo deste trabalho é realizar testes nos principais algoritmos de ordenação - *Quicksort*, *Selectionsort*, *Insertionsort*, *Mergesort*, *Shellsort* e *Heapsort* - com o propósito de comparar seus tempos de execução e analisar seu desempenho em diferentes cenários. Esses testes buscam investigar como esses algoritmos se comportam diante de diferentes tamanhos de dados, considerando casos em que os dados são grandes, pequenos, aleatórios ou já ordenados em ordem crescente ou decrescente.

A fim de avaliar o desempenho dos algoritmos, foram realizados testes repetidos em cada um deles, com variações de tamanho dos dados conforme especificado no trabalho. A média dos tempos de execução em segundos, bem como a média do número de comparações e movimentações realizadas por cada algoritmo, foram calculadas e exibidas a fim de se comparar cada algoritmo proposto.

Concluindo, este trabalho tem como objetivo fornecer uma análise comparativa dos principais algoritmos de ordenação, explorando seu desempenho em diferentes cenários e oferecendo uma compreensão mais aprofundada dessas técnicas fundamentais. Os resultados obtidos podem ser úteis tanto para aprimorar o conhecimento teórico dos algoritmos quanto para aplicá-los de forma eficiente em futuros projetos de programação.

2 Implementação

A implementação foi feita separando os algoritmos e os testes em .h *(header)* e .c, além de enumerações que mostram cada algoritmo de ordenação que o programa trabalha e todas as ordens de dados que ele testa para um código mais limpo e organizado. Além do programa já rodar os testes 10 vezes com todas as variações de tamanho exigidas no trabalho e retornar a média do tempo em segundos automaticamente além da média da quantidade de comparações e movimentações de cada ordenação escolhida. Os algoritmos de ordenação utilizados foram de implementações presentes no livro **Nivio Ziviani.** *Projeto*

de Algoritmos: Implementações em Pascal e C 3ª ed. Editora Cengage Learning, 2011 e no livro do **Backes, André Ricardo.** *Algoritmos e Estruturas de Dados em Linguagem C.* 1ª ed. Editora LTC, 2023., apenas modificados para comparar as *structs* personalizadas do programa. Foi necessário implementar duas funções relacionadas a cada tipo de ordenação, uma para ordenar o *array* de *structs* que contém apenas a chave, e outra para ordenar o *array* de *structs* que contém a chave e uma matriz *char[50][50].*

O programa possui um menu interativo que inicialmente solicita ao usuário o nome do algoritmo, a condição (se é ordenado ou aleatório) e o tamanho. E então passa para os testes, realizando o número de comparações, movimentações, e calculando o tempo gasto. O tempo é calculado usando a função *clock* da biblioteca *"time.h"*.Além disso, como um bônus para facilitar a plotagem dos gráficos, os resultados dos testes são impressos no terminal e também é criada uma planilha *".csv"* com os resultados para uma melhor visualização.

3 Resultados

Analisando a complexidade dos algoritmos, em notação *Big O*, obtemos as seguintes informações:

- *Quicksort*: Melhor caso e caso médio = O(n log n) e pior caso = O(n^2)
- *Selectionsort*: O(n^2) em todos os casos
- *Insertionsort*: Melhor caso = O(n) e caso médio e pior caso = O(n^2)
- *Mergesort*: O(n log n) em todos os casos
- *Shellsort*: O(n $\log(n)^2$) em todos os casos
- *Heapsort*: O(n log(n)) em todos os casos

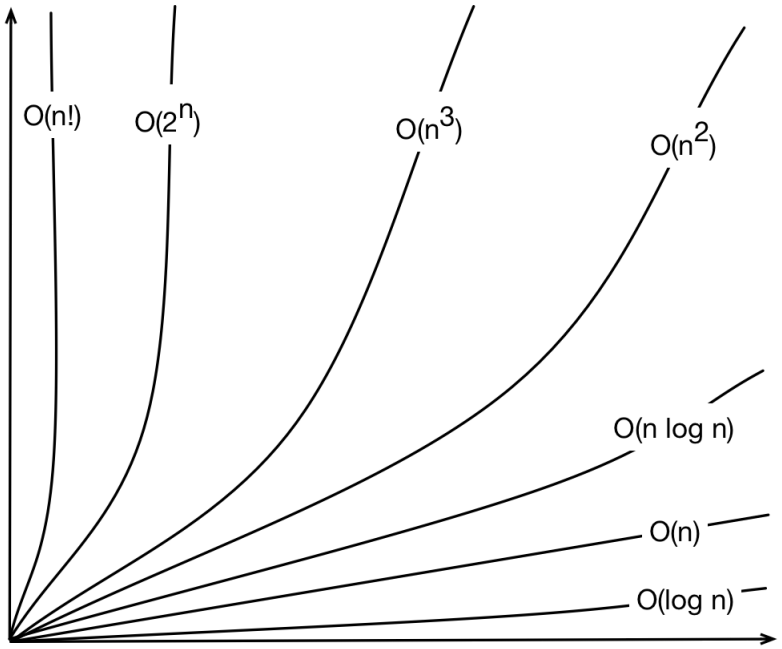

Figura 1: Influência do tamanho da entrada no tempo de ordenação

O gráfico acima ilustra como o tamanho da entrada influencia no tempo de ordenação, de acordo com a complexidade dos algoritmos. Essa visualização nos permite ter uma noção preliminar sobre quais algoritmos podem ser mais eficientes em diferentes cenários.

Além disso, ao realizar os testes, foram obtidos outros gráficos relevantes para cada algoritmo e condição de dados testada. Esses resultados detalhados estão disponíveis para uma melhor visualização e análise.

4 Análise de Eficiência

Foram considerados três aspectos para a análise de eficiência: tempo de execução, número de comparações e número de trocas realizadas.

Com entradas aleatórias:

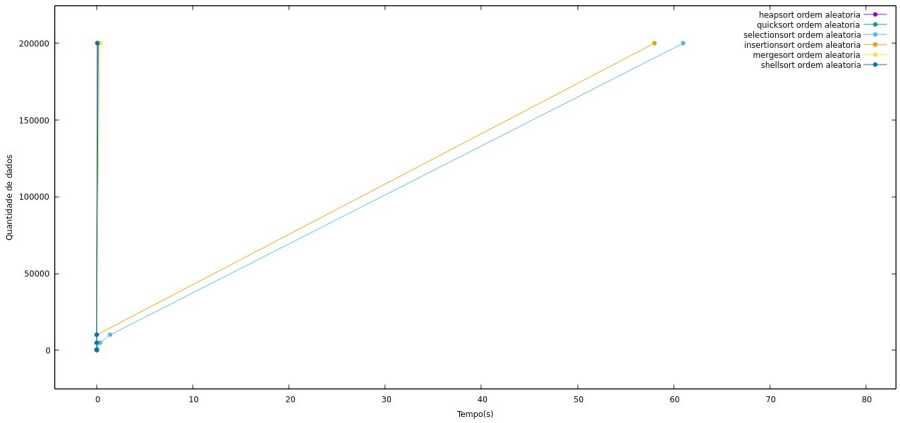

Figura 2: Gráfico de tempo de execução dos algoritmos com entradas aleatórias

É perceptível que, ao analisar pequenas entradas, as diferenças entre os tempos dos algoritmos são mínimas, sendo praticamente insignificantes nos testes. No entanto, à medida que a entrada aumenta, as diferenças entre eles se tornam evidentes. Destaca-se a ineficiência do *Insertion Sort* e do *Selection Sort* em uma *struct* com chave de comparação e um *array char*[50][50], enquanto os demais apresentam desempenho semelhante.

Com entradas em ordem crescente:

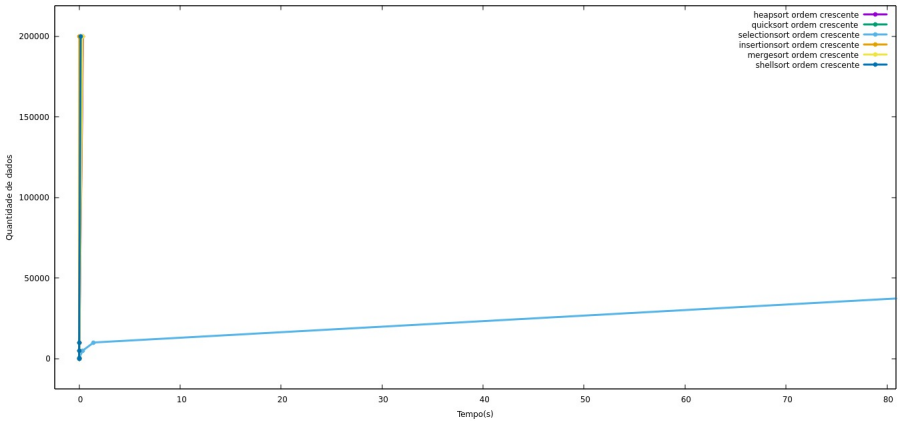

Figura 3: Gráfico de tempo de execução dos algoritmos com entradas em ordem crescente

É bastante claro a ineficiência do *Selection Sort* nesse caso, devido à sua complexidade $O(n^2)$. Por outro lado, o *Insertion Sort* é eficiente nessa situação, pois, com o vetor já ordenado, ele se torna o mais rápido, com complexidade O(n) no melhor caso. Os demais algoritmos apresentam desempenho semelhante

e também são eficazes.

Com entradas em ordem decrescente:

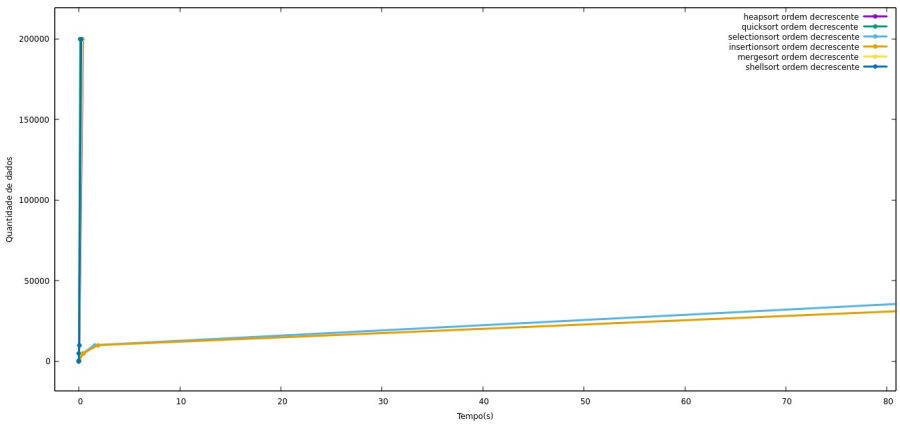

Figura 4: Gráfico de tempo de execução dos algoritmos com entradas em ordem decrescente

O vetor ordenado em ordem decrescente pode ser considerado o pior caso possível para a ordenação, pois está oposto ao esperado, exigindo a inversão de todos os elementos. Os algoritmos com pior desempenho foram o *Insertion Sort* - pior caso com complexidade $O(n^2)$ - seguido pelo *Selection Sort* - complexidade sempre será $O(n^2)$. Os demais algoritmos tiveram desempenho eficiente e quase idêntico, levando menos de um segundo. A ineficiência do *Insertion Sort* deve-se à necessidade de percorrer todo o vetor da esquerda para a direita e ordenar os elementos durante as movimentações. Já a ineficiência do *Selection Sort* ocorreu devido a esse ser o pior caso para sua atuação, com trocas ocorrendo em todas as movimentações.

Número de movimentações:

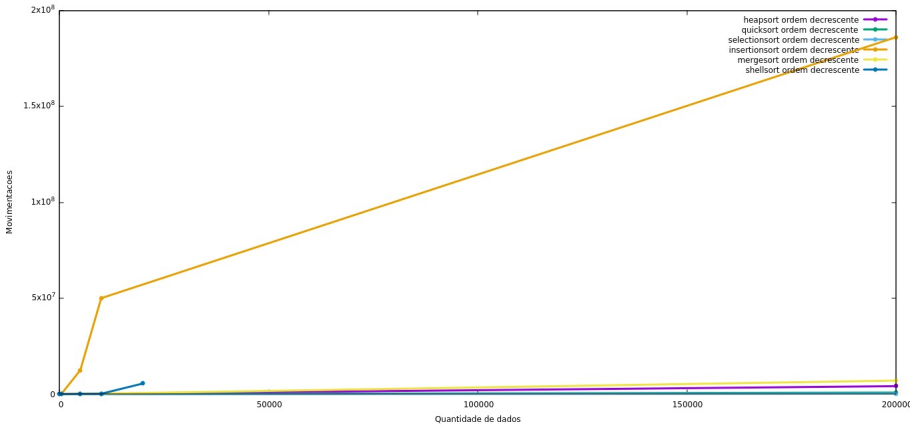

Figura 5: Gráfico de movimentações durante a execução dos algoritmos

O *Insertion Sort* teve o pior desempenho nesse aspecto, enquanto os demais tiveram desempenho similar. Isso ocorre, pois as movimentações ocorrem quando um valor é atribuído a uma variável, fazendo a cópia de todo o conteúdo. O *Insertion sort* realiza a mudança de todo o vetor para frente, para só depois, atribuir um valor. Em comparação, o *Selection sort* encontra o menor e realiza apenas três movimentações em cada processo. A diferença entre os testes com elementos de uma única *struct* e elementos com uma *struct* com chave de comparação e um *array char*[50][50] só se diferencia na questão de movimentações, já que, quanto maior o elemento analisado, maior o custo para movimentar o elemento.

Número de comparações:

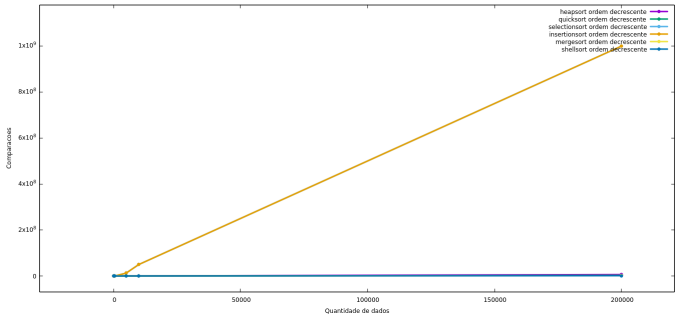

Figura 6: Gráfico do número de comparações durante a execução dos algoritmos

O *Insertion Sort* apresentou um desempenho inferior em comparação aos demais, com um grande crescimento no número de comparações conforme a entrada aumenta, pois são feitas comparações a cada movimentação ao longo do vetor. Para entradas com um vetor ordenado em ordem crescente, ele mostrou-

se mais eficiente que os demais, porém, ao analisarmos um vetor ordenado em ordem decrescente, sua ineficiência é destacada. Para entradas de um único *struct* e *struct* com comparações de *arrays* e *char*[50][50] o gráfico é o mesmo. Seguindo a curva de crescimento no número de comparações nos dois casos.

Com valores maiores, o *Quick Sort* se destaca positivamente em eficiência em relação aos demais, executando a tarefa em menos tempo. Por outro lado, o *Selection Sort* possui uma curva de crescimento mais acentuada, revelando-se menos eficiente que os demais.

Portanto, com base nas análises, podemos concluir que, para uma pequena quantidade de dados, o *insertion sort* é uma opção adequada. Ele é eficiente tanto para dados já ordenados quanto para dados parcialmente ordenados. Além disso, o *insertion sort* é um algoritmo estável, o que significa que mantém a ordem dos elementos iguais, e pode ser aplicado *online*, ou seja, é possível ordenar os dados durante a inserção. Contudo, é importante ressaltar que, para elementos com *structs* que ocupem uma grande quantidade de memória, outra opção a ser considerada é o *selection sort*, uma vez que ele requer menos movimentações. Embora ambos, o *insertion sort* e o *selection sort*, tenham complexidade de tempo $O(n^2)$, em uma pequena quantidade de dados, a diferença de desempenho entre eles pode ser insignificante.

Para lidar com uma grande quantidade de dados, os melhores algoritmos são aqueles que possuem uma complexidade de tempo O(n log n). Quando se trata de algoritmos críticos, nos quais a variação de tempo não é permitida, destaca-se o *heap sort*. Ele mantém a estabilidade dos dados e possui uma complexidade de tempo O(n log n) em todos os casos, além de não requerer memória adicional.

No entanto, se o requisito principal for velocidade e não houver restrições críticas, a melhor opção é o *quicksort*. O *quicksort* é conhecido por sua eficiência em geral, com uma complexidade média de tempo O(n log n). Ele também possui uma boa adaptação a diferentes situações e pode superar outros algoritmos em termos de velocidade de ordenação.

O *merge sort* e o *shell sort* são boas escolhas e possuem complexidade O(n log n). No entanto, é importante observar que o *merge sort* requer um gasto adicional de memória devido ao processo de mesclagem, em comparação com outros métodos de ordenação. Por outro lado, o *shell sort* é uma opção adequada quando a restrição de memória é uma preocupação, pois não é um algoritmo recursivo e não requer um gasto extra de memória.

5 Conclusão

De forma geral, este trabalho cumpriu seu objetivo ao proporcionar uma compreensão aprimorada dos algoritmos de ordenação, tanto para os autores como para os leitores da documentação. Através da implementação e análise dos diferentes algoritmos, foi possível observar seus desempenhos em diversas situações, considerando tamanhos variados de dados, ordens diferentes e características específicas. Essa análise proporcionou *insights* valiosos sobre a eficiência e complexidade de cada algoritmo, auxiliando na escolha da abordagem mais adequada

para diferentes cenários.

Logo, o trabalho não apenas ofereceu uma melhor compreensão dos algoritmos de ordenação mas também estabeleceu uma base sólida para a aplicação prática dessas técnicas. O conhecimento adquirido por meio deste projeto certamente será útil no desenvolvimento de soluções eficientes e otimizadas em futuros desafios de programação e análise de dados.

6 Bibliografia

CPLUSPLUS, 2023. Disponível em: https://cplusplus.com/reference/ Acesso em: 26 maio. 2023.

Backes, André Ricardo. *Algoritmos e Estruturas de Dados em Linguagem C.* 1ª ed. Editora LTC, 2023.

Nivio Ziviani. *Projeto de Algoritmos: Implementações em Pascal e C* 3ª ed. Editora Cengage Learning, 2011