Projeto 02 ED Ordenação

Dupla: Kauan Oliveira Perdigão Lopes - 514867
e Luís Gustavo Rabelo de Oliveira - 540974
Professor: Atílio Gomes Luiz

Semestre 2023.1

1 Descrições gerais de cada algoritmo de ordenação aplicado no projeto

1.1 bubbleSort:

- Complexidade de pior caso: $O(n^2)$, onde **n** é o tamanho da lista a ser ordenada.
- bubbleSort é um algoritmo **instável**, pois não preserva a ordem relativa dos elementos iguais durante a ordenação.
- bubbleSort é um algoritmo **in loco**, pois a ordenação é realizada diretamente da lista original.
- bubbleSort é um algoritmo iterativo, pois atêm-se ao uso de loop's de repetição **while**.
- O bubbleSort vai percorrer a lista repetidamente comparando os pares de nós adjacentes e trocando-os se estiverem fora de ordem.

1.2 selectionSort:

- Complexidade de pior caso: $O(n^2)$, onde **n** é o tamanho da lista a ser ordenada.
- selectionSort é um algoritmo **instável**, pois não preserva a ordem relativa dos elementos iguais durante a ordenação.
- selectionSort é um algoritmo **in loco**, pois a ordenação é realizada diretamente da lista original.
- selectionSort é um algoritmo iterativo, pois atêm-se ao uso de loop's de repetição **while**.
- O selectionSort vai encontrar o menor elemento não ordenado em cada iteração e colocá-lo na posição correta, repetindo esse processo até que toda a lista esteja ordenada.

1.3 insertionSort:

- Complexidade de pior caso: $O(n^2)$, onde **n** é o tamanho da lista a ser ordenada.
- insertionSort é um algoritmo **instável**, pois não preserva a ordem relativa dos elementos iguais durante a ordenação.
- insertionSort é um algoritmo **in loco**, pois a ordenação é realizada diretamente da lista original.
- insertionSort é um algoritmo iterativo, pois atêm-se ao uso de loop's de repetição **while**.
- O insertionSort constrói uma sublista ordenada, inserindo elementos não ordenados na posição correta dentro dessa sublista, deslocando os elementos maiores para a direita.

1.4 quickSort:

- Complexidade de pior caso: $O(n^2)$, onde **n** é o tamanho da lista a ser ordenada.
- quickSort é um algoritmo **instável**, pois não preserva a ordem relativa dos elementos iguais durante a ordenação.
- quickSort é um algoritmo **in loco**, porém, necessita de espaço adicional para a pilha de chamadas recursivas.
- quickSort é um algoritmo **recursivo**.
- O quickSort seleciona um elemento como "pivô"e rearranja os elementos da lista de forma que os elementos menores que o pivô fiquem antes dele e os maiores fiquem depois dele. Em seguida, o processo é aplicado recursivamente às sublistas antes e depois do pivô.

1.5 mergeSort:

- Complexidade de pior caso: O(n log n), onde **n** é o tamanho da lista a ser ordenada.
- mergeSort é um algoritmo **estável** de ordenação. Porque ele preserva a ordem relativa dos elementos com chaves iguais durante o processo de ordenação.
- mergeSort não é um algoritmo **in loco**, pois ele requer memória adicional para armazenar as listas durante o processo de mesclagem.
- mergeSort é um algoritmo **recursivo**.
- O mergeSort divide a lista em duas metades, ordenar cada metade separadamente por meio de chamadas recursivas e, em seguida, mesclar as duas metades ordenadas para obter a lista totalmente ordenada.

1.6 shellSort:

- Complexidade de pior caso: $O(n^2)$, onde **n** é o tamanho da lista a ser ordenada.
- shellSort pode ser implementado tanto como um algoritmo estável quanto como um algoritmo instável, dependendo da forma como é implementado. Se a implementação do shellSort levar em consideração a ordem relativa dos elementos com chaves iguais e garantir que eles não sejam trocados entre si, então o algoritmo pode ser implementado como um algoritmo estável. Nesse caso, a ordem relativa dos elementos com chaves iguais será preservada durante o processo de ordenação. No entanto, se a implementação do shellSort não levar em consideração a estabilidade e permitir que elementos com chaves iguais sejam trocados entre si, então o algoritmo pode ser implementado como um algoritmo instável. Nesse caso, a ordem relativa dos elementos com chaves iguais pode não ser preservada após a ordenação.
- shellSort é um algoritmo **in loco**, pois a ordenação é realizada diretamente da lista original.
- shellSort é um algoritmo iterativo, pois atêm-se ao uso de loop's de repetição **while** e **for**.
- O shellSort realiza uma ordenação por inserção em listas de lacunas (que são chamados de **gaps**) cada vez maiores, reduzindo gradualmente o tamanho das lacunas até que a ordenação final seja realizada com uma lacuna igual a 1. Isso permite que elementos distantes sejam comparados e trocados, facilitando a movimentação dos elementos para suas posições corretas.

2 Gráficos dos tempos de execução

2.1 Gráfico do tempo de execução do bubbleSort:

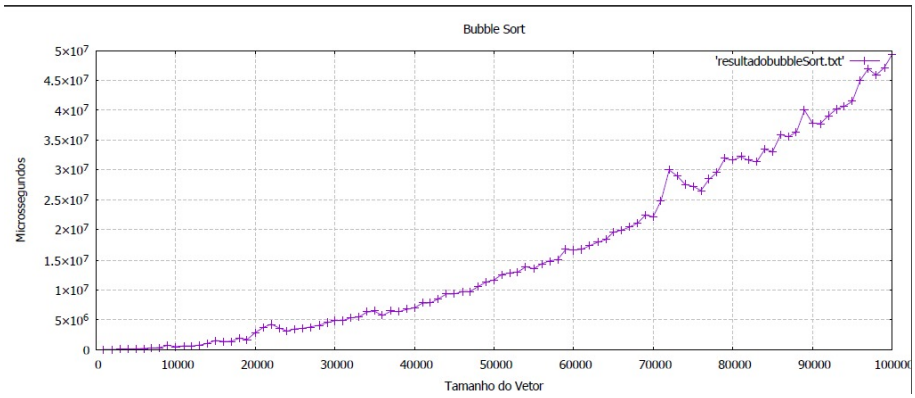

Figura 1: Em geral, podemos observar que à medida que o tamanho do algoritmo aumenta, o tempo de execução do bubbleSort também aumenta significativamente. Isso é consistente com a complexidade quadrática do algoritmo, onde o número de comparações e trocas aumenta exponencialmente com o tamanho dos dados. Portanto, para conjuntos de dados grandes, o bubbleSort pode se tornar impraticável em termos de tempo de execução.

2.2 Gráfico do tempo de execução do selectionSort:

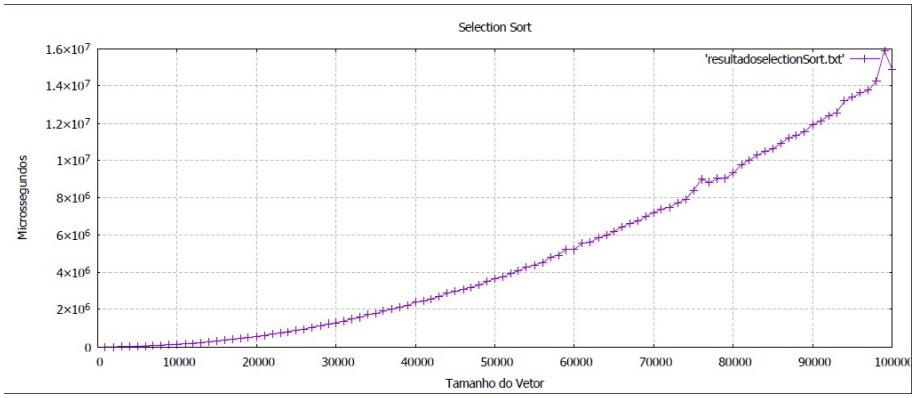

Figura 2: Os dados mostram que o selectionSort tem um desempenho pior em termos de tempo de execução à medida que o tamanho do algoritmo aumenta. Embora seja simples de implementar, o selectionSort não é recomendado para conjuntos de dados grandes devido à sua ineficiência em comparação com outros algoritmos mais eficientes.

2.3 Gráfico do tempo de execução do insertionSort:

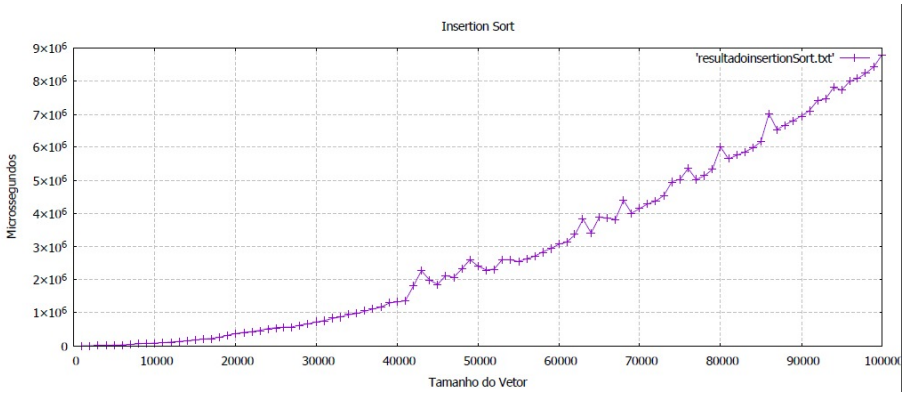

Figura 3: Para tamanhos menores de algoritmo, como 1000 e 2000, o tempo de execução é relativamente baixo. No entanto, à medida que o tamanho do algoritmo aumenta, o tempo de execução cresce consideravelmente. O insertionSort tem um desempenho melhor do que o bubbleSort e o selectionSort para conjuntos de dados maiores. No entanto, ainda é menos eficiente em comparação com algoritmos como o quickSort ou o mergeSort. Uma característica interessante do insertionSort é que seu desempenho melhora à medida que o tamanho do algoritmo aumenta. Isso pode ser visto nas diferenças de tempo de execução entre tamanhos próximos, como 8000 e 9000, onde a diferença é menor em comparação com tamanhos menores, como 2000 e 3000.

2.4 Gráfico do tempo de execução do quickSort:

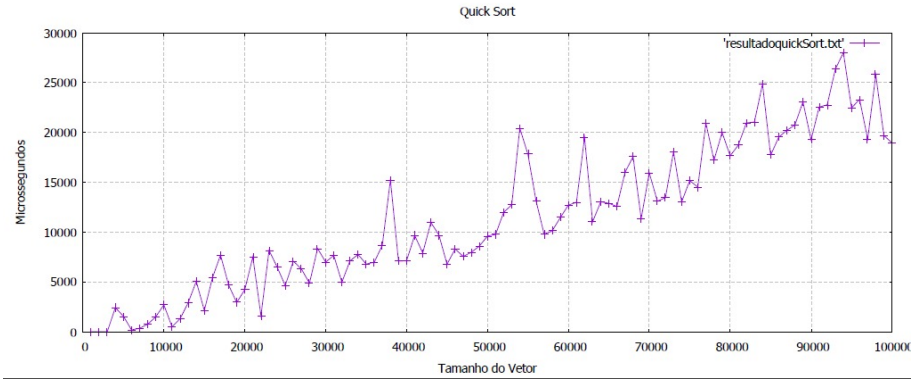

Figura 4: Ao analisar os dados, podemos observar algumas características do quickSort. Para os tamanhos menores de algoritmo, como 1000, 2000 e 3000, o tempo de execução é 0, indicando que o algoritmo é altamente eficiente para tamanhos menores de entrada. Conforme o tamanho do algoritmo aumenta, o tempo de execução também aumenta, mas não de forma linear. Em alguns casos, como para o tamanho do algoritmo de 6000, o tempo de execução é significativamente menor em comparação com tamanhos menores. Isso pode ser resultado de uma boa escolha de pivô ou outros fatores específicos do conjunto de dados. No entanto, a partir de um determinado tamanho do algoritmo, o tempo de execução começa a aumentar novamente. Isso pode indicar que, para conjuntos de dados maiores, o quickSort pode encontrar casos em que a escolha de pivô não é tão eficiente ou que a recursão leva a um número maior de comparações e trocas.

2.5 Gráfico do tempo de execução do mergeSort:

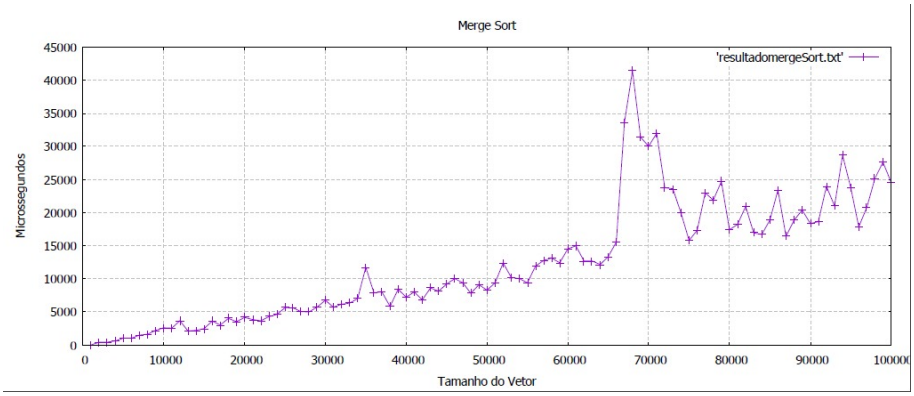

Figura 5: À medida que o tamanho do algoritmo aumenta, o tempo de execução também tende a aumentar, indicando uma tendência de crescimento. No entanto, essa relação não é estritamente linear. Há variações e flutuações no tempo de execução à medida que o tamanho do algoritmo aumenta. Por exemplo, nos primeiros pontos de dados, vemos um aumento gradual no tempo de execução à medida que o tamanho do algoritmo aumenta de 1000 para 6000. No entanto, a partir do ponto em que o tamanho do algoritmo é 6000, o tempo de execução começa a variar consideravelmente, não seguindo uma tendência clara de aumento ou diminuição.

2.6 Gráfico do tempo de execução do shellSort:

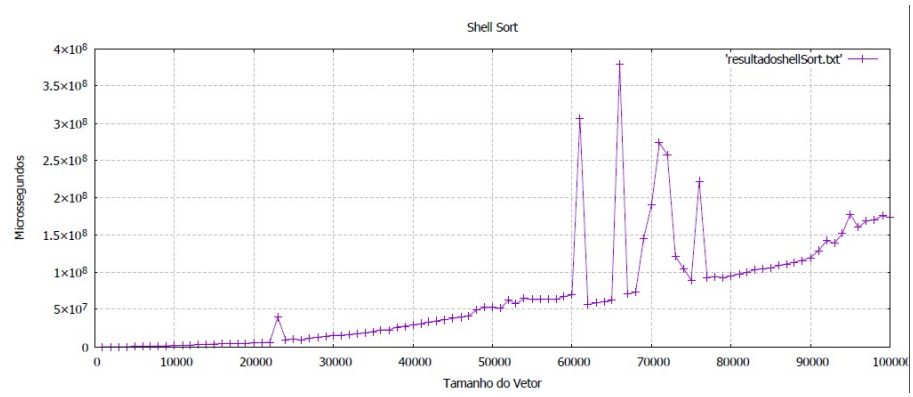

Figura 6: O tempo de execução do algoritmo varia de acordo com o tamanho do algoritmo, mas não segue um padrão de crescimento linear ou quadrático. Existem flutuações e variações nos tempos de execução para diferentes tamanhos de entrada. Em geral, o shellSort apresenta um bom desempenho para tamanhos menores de entrada, com tempos de execução relativamente baixos. No entanto, à medida que o tamanho do algoritmo aumenta, o tempo de execução também aumenta consideravelmente. Notavelmente, há flutuações nos tempos de execução em alguns tamanhos de entrada, como o pico no tempo de execução em 23000, seguido de uma queda nos tamanhos subsequentes.

3 Comparação gráfica dos tempos de execução dos 6 algoritmos

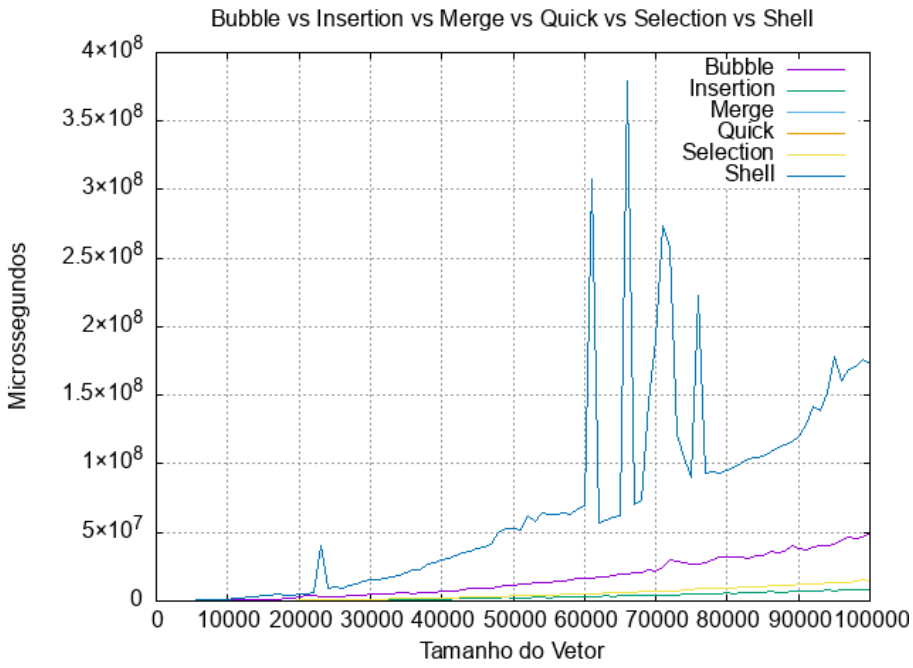

Figura 7: Em geral, o mergeSort e o quickSort apresentam desempenho mais estável e rápido para uma ampla variedade de tamanhos de entrada. O bubbleSort tem um desempenho extremamente ruim em todos os tamanhos de entrada, enquanto o shellSort mostra uma variação em seu desempenho dependendo do tamanho do algoritmo e da sequência de lacunas utilizada. A escolha do algoritmo de ordenação a ser utilizado dependerá do tamanho do algoritmo e da eficiência desejada, levando em consideração as características e complexidades de cada algoritmo.

4 Como foi realizada a divisão do trabalho entre a dupla

O trabalho foi igualmente feito pela dupla, onde ambos os integrantes programaram as funções-membro e todo o código fonte juntos, de maneira paralela, utilizando a **IDE Visual Studio Code**. O relatório também foi igualmente feito pela dupla, onde tanto o relatório quanto o código programado foram feitos com reuniões da dupla via Discord.

5 As dificuldades encontradas

Tivemos dificuldades com a função **void gera_dados()** que não estava conseguindo encontrar os diretórios especificados **"dados"e "resultados"**, onde deveriam ser salvos os dados necessários para plotagem dos gráficos contendo o tamanho da lista e tempo de execução, e então decidimos acessar o sistema de arquivos diretamente no código para criação dos diretórios já citados, que encontram-se na pasta **output**. Também foi encontrada dificuldade na plotagem dos gráficos, especificamente no gráfico que compara todos os algoritmos ao mesmo tempo, onde após algumas tentativas fracassadas de fazer, utilizando o **gnplot**, conseguimos através da criação de um arquivo **.p** e o carregando no terminal do **gnplot**.

6 Referências utilizadas pela dupla como suporte ao desenvolvimento do projeto

Referências

Harvey M Deitel and Paul J Deitel. *C++. Fondamenti di programmazione*. Maggioli Editore, 2014.
Deitel and Deitel [2014]
https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao?gclid=Cj0KCQjwtamlBhI
wcB
https://www.youtube.com/watch?v=2UNw5-jfqhc
https://www.youtube.com/playlist?list=PL5TJqBvpXQv4l7nH-08fMfyl7aDFNW_fC
https://www.geeksforgeeks.org/shellsort/
https://acervolima.com/criar-diretorio-ou-pasta-com-o-programa-c-c/
https://www.youtube.com/watch?v=0ENnVZsMjL8
https://drive.google.com/drive/folders/1wP9D2TqQfspgCnZsD7u5KengSOeaweNu?usp=sharing