Trabalho Prático 2
A ordenação da estratégia de dominação do imperador

Euller Saez Lage Silva

2019054501

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

eullersaez@ufmg.br

1. Introdução

O problema proposto consistia em desenvolver um sistema computacional para o imperador de um mundo paralelo da saga star wars no intuito de que pudesse se organizar para dominar novas civilizações, haja visto que mesmo detendo um império, o universo ainda é muito grande e nem todas as outras muitas civilizações o pertence. Assim, o imperador poderia criar um plano com foco na dominação das civilizações mais próximas a sua base, e, em caso de empate entre a distância dessas civilizações, deveria-se priorizar aquelas cujo tamanho da população fossem os maiores primeiro. Dessa forma, para resolver esse problema, deveria-se implementar alguns métodos de ordenação de dados. No entanto, eu era um infiltrado da Aliança Rebelde, e para ajudá-los contra o imperador, deveria implementar métodos pouco eficientes para ele e pensar em métodos muito mais eficientes para meus parceiros, que conseguiriam saber prioritariamente o plano tramado e ter tempo de sobra para bolar uma estratégia.

2. Implementação

O programa foi desenvolvido na linguagem C++ por meio do Visual Studio Code para Windows da Microsoft Corporation, utilizando o Windows Subsystem for Linux (WSL) e compilado pelo compilador G++ da GNU Compiler Collection.

2.1 Estrutura de dados

A implementação dos programas como um todo utilizou a estrutura de dados array. A escolha por ela se deu pelo fato de ser uma estrutura de fácil manipulação para trocas de posições dos elementos.

2.2 Classes

Foram utilizadas cinco classes principais para a implementação do trabalho. Uma estava presente em todos os programas e as outras quatro estavam cada uma em seus respectivos programas. A primeira delas, contida em todos, foi a classe Civilizacoes, que contém um atributo nome do tipo string, um atributo distancia do tipo inteiro e um atributo tamanho do tipo inteiro. Possui também métodos setters e getters, além de um método *imprime*,

responsável por imprimir em uma linha os seus atributos, na mesma ordem em que foram citados.

As demais classes receberam os nomes dos métodos que utilizavam para realizar a ordenação, e inclusive foram baseadas na implementação vista em aula, feitas as devidas adaptações, exceto no caso do Mergesort, que além do visto em sala, foi também baseado na implementação disponível neste site[1].

A Insercao, implementada para o imperador como algoritmo1, possui um único método *ordena*, responsável por receber um array de civilizações, seu número de civilizações e ordená-los. Para isso, conforme comentado no código fonte, utilizou-se da vantagem do algoritmo ser estável, e assim ordena-se primeiramente todas as civilizações em ordem decrescente por tamanho da população e logo após ordena-se novamente, só que agora de forma crescente e utilizando a distância como chave. Isso é possível haja visto que na segunda ordenação não se perde a ordem das posições relativas de cada civilização quando estão a mesma distância.

A Selecao, implementada para o imperador como algoritmo2, também possui um único método *ordena*, que funciona de modo similar ao implementado em aula, no entanto no segundo loop aninhado do tipo for é checado não só se a civilização na posição em que se encontra tem distância menor do que a civilização na posição guardada até então com a menor distância, mas também adiciona um operador OR para o caso de possuir distância igual porém maior tamanho de civilização, o que faz ter prioridade e vir primeiro.

O Quicksort, implementado para a aliança rebelde como algoritmo1, possui o método *particao*, responsável por particionar o array e trocar as posições das civilizações em relação ao pivô, de modo análogo ao utilizado no algoritmo por seleção. Conforme comentado no código, o loop while irá parar quando a civilização na posição i tiver distância maior do que o pivô, ou distância igual porém tamanho de população menor, e quando j tiver distância menor do que o pivô ou distância igual porém tamanho de população maior, e, portanto deverá vir antes, assim como i deverá vir depois. Outro método é o *ordenacao*, responsável por chamar *particao* e a si mesmo de forma recursiva mudando sempre o intervalo no qual o particionamento deverá ser feito. Por fim, o método *ordena* é responsável por chamar *ordenacao* no intervalo adequado do array a ser ordenado.

O Mergesort, implementado para a aliança rebelde como algoritmo2, possui os métodos *mergeDistancia, mergeTamanho, mergeSortDistancia, mergeSortTamanho* e *ordena*. Assim como na implementação por Inserção, também aproveitou-se do fato de que o MergeSort é um algoritmo estável e utilizou-se a mesma estratégia de ordenar primeiro por tamanhos de população das civilizações em ordem decrescente e, posteriormente por distâncias em ordem crescente. Assim, os métodos *mergeDistancia* e *mergeTamanho* seguem exatamente a mesma lógica de dividir o array em outros dois e ir re-introduzindo as civilizações em cada parte adequada, com a diferença de que o primeiro combina por menor distância e o segundo por maior tamanho. Da mesma forma, *mergeSortDistancia* e *mergeSortTamanho* também seguem as mesmas lógicas, e chamam a si próprias de forma

[1] https://www.geeksforgeeks.org/merge-sort/

recursiva para cada primeira e segunda partes do array, e logo em seguida chamam os métodos de *mergeDistancia* ou *mergeTamanho* para combinar os resultados de forma ordenada. *ordena*, então chama *mergeSortTamanho* e *mergeSortDistancia*.

3. Instruções de compilação e execução

3.1 Compilação

Para compilar o programa, primeiro é necessário abrir o diretório euller_silva/diretorio_algoritmo no terminal do linux e digitar o comando "make', onde "diretorio_algoritmo" é o nome do diretório de qual algoritmo deseja-se compilar. Por exemplo, o diretório para o algoritmo que implementa o quicksort seria euller_silva/src_ordenacao_alianca_rebelde_algoritmo1.

3.2 Execução

Para executar o programa, após compilado, ainda dentro do mesmo diretório euller_silva/diretorio_algoritmo no terminal do linux, basta digitar o comando "./tp2".

3.3 Limpeza de arquivos temporários

Para limpar todos os arquivos temporários gerados durante a compilação, ainda dentro do mesmo diretório euller_silva/diretorio_algoritmo no terminal do linux, basta digitar o comando "make clean".

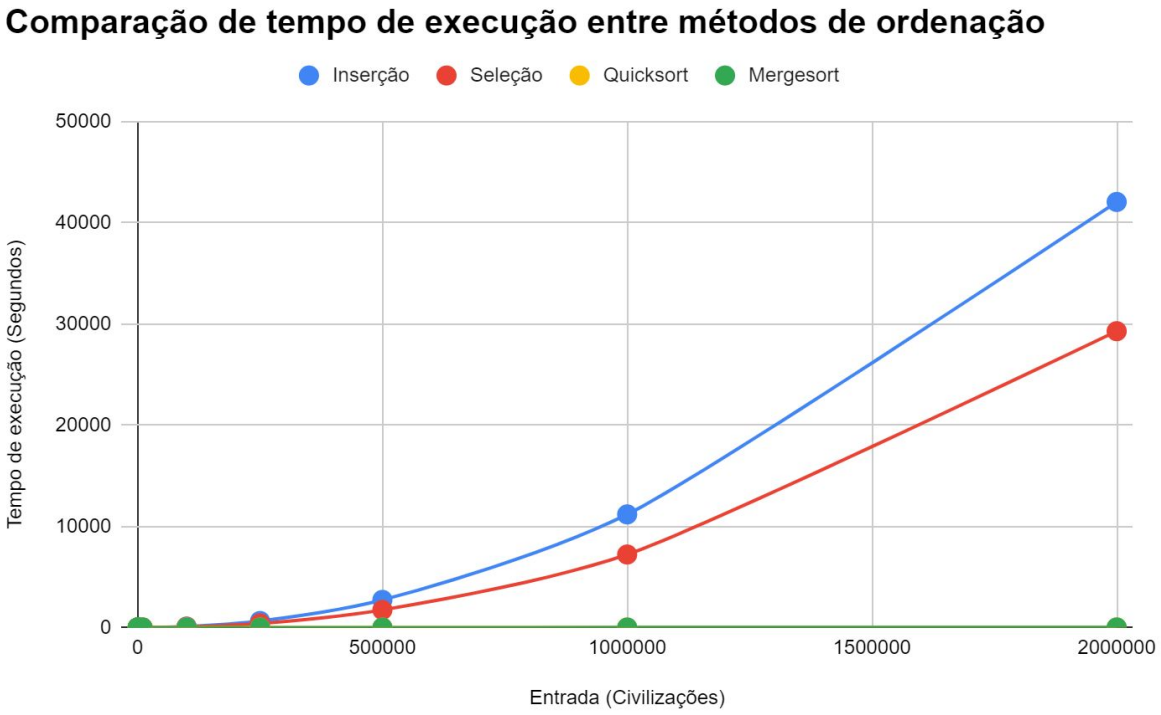
4. Análise de complexidade

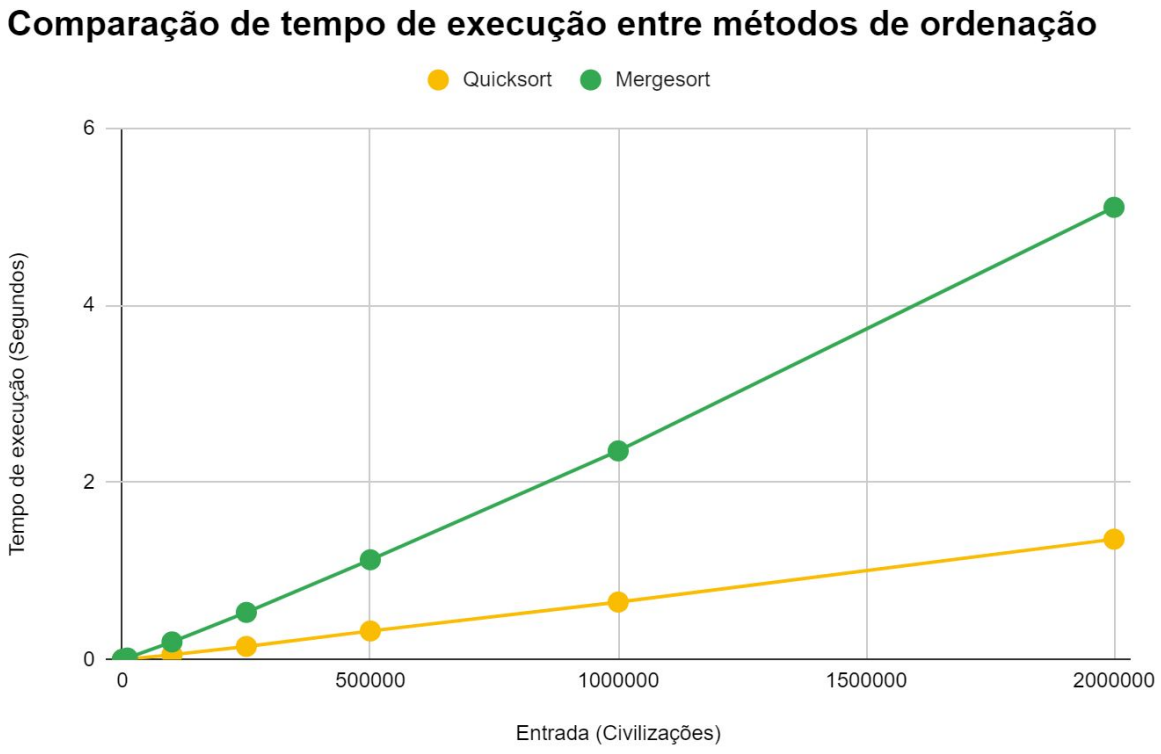
4.1 Tempo

Para comparação da análise da complexidade de tempo entre os algoritmos, foi coletado os seguintes tempos abaixo em segundos com o auxílio do código disponível neste site[2].

Entrada	Inserção	Seleção	Quicksort	Mergesort
50	0.000115	0.000089	0.000082	0.000117
100	0.000178	0.000149	0.000104	0.000175
500	0.002496	0.001563	0.000263	0.00062
1000	0.009745	0.006308	0.000451	0.001261
10000	0.964803	0.572768	0.005219	0.017165
100000	95.91229	59.251542	0.054776	0.199445
250000	644.442853	400.121811	0.148548	0.532856
500000	2731.196778	1737.670084	0.322653	1.127275
1000000	11180.00894	7215.64707	0.649481	2.356898
2000000	42055.98978	29285.46922	1.359635	5.107586

[2] https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/





Como é possível perceber pelos gráficos, existe uma diferença significativa no tempo de execução de cada algoritmo, especialmente quando as entradas crescem e se tornam mais robustas.

No caso do algoritmo que utiliza o método de Inserção, sempre que se recebe um array a ser ordenado haverá um loop externo do tipo for que será percorrido n vezes, onde n é

sempre o tamanho do array de civilizações. O loop interno do tipo while ditará quando o algoritmo cairá no melhor ou pior caso. Isso porque caso já esteja ordenado terá um custo constante de verificar a condição do while e logo passará para a próxima iteração do loop externo, sendo esse o seu melhor caso. No seu pior caso, terá sempre que executar o while da posição i até a posição 0 do array, o que ocorre quando todas as civilizações estão na ordem inversa a qual se deseja ordenar. Além disso, como essas etapas são realizadas duas vezes, para ordenar primeiro com relação ao tamanho e depois com relação a distância, teremos que o número de comparações C(n) do algoritmo será a soma do número de comparações de cada um dos loops aninhados. Assim, em seu pior caso, quando tanto os tamanhos quanto as distâncias estão na ordem reversa, tem-se que

$$C(n) = O(n^2) + O(n^2) = O(n^2)$$

Enquanto em seu melhor caso

$$C(n) = O(n) + O(n) = O(n)$$

Quanto ao número de movimentações M(n), no loop externo sempre há 2 movimentações para cada vez que é executado. Assim, o pior e melhor caso novamente são definidos pela execução do loop interno while, que será executado i-1 vezes em seu pior caso para cada posição i do loop externo. Seguindo a mesma lógica anterior dadas as duas vezes que a ordenação é feita, tem-se que no pior caso:

$$M(n) = O(n^2) + O(n^2) = O(n^2)$$

Enquanto em seu melhor caso

$$M(n) = O(n) + O(n) = O(n)$$

O algoritmo de Seleção, por sua vez sempre percorre dois loops aninhados do tipo for. O primeiro será executado n-1 vezes e o segundo sempre executa n-i vezes para cada posição i do loop externo. Assim, em qualquer caso o número de comparações C(n) será

$$C(n) = O(n^2)$$

Além disso, para cada execução do loop externo são feitas 3 movimentações, e, portanto, o número de movimentações M(n) será sempre

$$M(n) = O(n)$$

O número de comparações executado pelo Quicksort sempre irá depender da seleção inicial do pivô no método *particao*. Caso aconteça de ser um dos extremos, ou seja, o menor ou maior dos elementos, cairá em seu pior caso pois o método *ordenacao* será chamado recursivamente n vezes, deixando em ordem apenas um dos elementos por vez. Assim, cai-se

numa equação de recorrência $T(n) = T(n-1) + n$ cuja solução resultará no número de comparações C(n)

$$C(n) = O(n^2)$$

Enquanto no seu melhor caso o pivô escolhido é de modo que o array sempre será subdividido em partes iguais, caindo na equação de recorrência $T(n) = 2T(n/2) + n$ cuja solução resultará no número de comparações C(n)

$$C(n) = O(nlogn)$$

Por fim, o algoritmo Mergesort, assim como o Inserção, também se aproveita de ser estável e executa duas ordenações, sendo a primeira por tamanho e a segunda por distância. Assim, para os métodos *mergeDistancia* e *mergeTamanho*, o número de comparações C(n) é sempre linear $O(n)$. Os métodos *mergeSortDistancia* e *mergeSortTamanho* por sua vez sempre vão chamar a si próprios recursivamente para cada parte do array recebido e no final chamam uma vez o método *mergeDistancia* ou *mergeTamanho*. Desse modo, para cada ordenação, cai-se numa equação de recorrência $T(n) = 2T(n/2) + O(n)$ cuja solução resultará no número de comparações $C(n) = O(nlogn)$. Portanto, tem-se que o número de comparações final do método *ordena* será

$$C(n) = O(nlogn) + O(nlogn) = O(nlogn)$$

4.2 Espaço

Vamos considerar que cada civilização ocupa uma unidade de espaço no programa.

No caso dos algoritmos por Inserção e Seleção, sempre teremos manipulações diretas no array recebido e apenas algumas mesmas variáveis auxiliares são utilizadas em todos os casos. Assim, sempre terão custo de espaço constante $O(1)$.

O Quicksort, por ser implementado recursivamente terá em cada etapa da pilha de execução $O(n)$ civilizações armazenadas, e, portanto, terá custo de espaço linear $O(n)$.

O Mergesort por sua vez sempre tem que alocar, dentre outras variáveis com custo constante que sempre são declaradas, nos métodos *mergeDistancia* ou *mergeTamanho* dois arrays auxiliares que ocuparão $O(n)$ unidades de espaço. Assim, como são chamados no método *mergeSortDistancia* e *mergeSortTamanho,* o custo total de espaço será linear $O(n)$.

5. Conclusão

Após o término do trabalho, ficou bem evidente a importância de se estudar cada algoritmo de ordenação e ponderar muito sobre seus prós e contras para poder tomar melhores decisões de quando implementá-los.

Em geral, a discrepância no tempo de execução de algoritmos como o quicksort contra o de inserção, por exemplo, faria parecer com que esperar segundos contra horas respectivamente nos testes mais robustos para ordenar exatamente as mesmas entradas seria extremamente inviável. De fato, na maior parte do tempo isso será verdade. No entanto, o

método de inserção não é inválido por isso, e inclusive tem algumas vantagens sobre o quicksort aqui implementado. Uma delas é a sua implementação mais simples, que poupa tempo do programador. Além disso, ser estável também se mostrou um ponto muito positivo, pois neste trabalho foi possível ordenar a princípio de uma forma, e logo após ordenar de outra, mantendo posições relativas importantes da primeira ordenação. Em sistemas que não recebem entradas muito grandes talvez não seja necessário se preocupar com tanta eficiência de tempo de execução e assim abre-se a possibilidade de aproveitar benefícios de métodos não tão eficientes, como o inserção, em detrimento do quicksort, por exemplo.

Dessa forma, reforça-se a necessidade de verificar bem cada algoritmo e determinar de modo analítico qual se adequa mais a cada aplicação.

References

CHAIMOWICZ, L. , PRATES, R. Slides da disciplina Ordenação: Métodos Simples - Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais (UFMG). 2020

CHAIMOWICZ, L. , PRATES, R. Slides da disciplina Ordenação: QuickSort - Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais (UFMG). 2020

CHAIMOWICZ, L. , PRATES, R. Slides da disciplina Ordenação: MergeSort - Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais (UFMG). 2020