Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Curso de Graduação em Ciência da Computação

Trabalho Prático 1 - Estrutura de Dados
Métodos de Ordenação

Lucas Affonso Pires

Matrícula: 2023028420

Professor: Wagner Meira Junior

Belo Horizonte, Junho de 2024

1 Introdução

O objetivo deste trabalho é bem simples, demonstrar os diversos métodos de ordenação, implementando-os e comparando-os. Os seguintes métodos serão discutidos aqui: método da bolha, método da inserção, método da seleção, merge sort, quick sort, shell sort, couting sort, bucket sort e radix sort. Ao longo do documento serão apresentados dados que demonstrarão a utilidade de cada um desses métodos de acordo com suas qualidades e problemas.

2 Método

Os algoritmos foram implementados de acordo com os slides fornecidos durante o curso. No trecho de Análise de Complexidade, cada algoritmo indica qual foi o método selecionado para a sua implementação e estudo. Para a parte de Análise Experimental, os algoritmos foram separados em gráficos feitos no google colab condizentes com o seu tempo de execução; o método da bolha, seleção e inserção foram agrupados juntos, assim como o Quicksort, Shell sort e Merge sort, e o Counting sort, Radix sort e Bucket sort. Para cada um dos testes realizados, foram utilizados vetores de 10, 100, 1000, 10000, 100000 ou 1000000 de elementos, que vão de acordo com a necessidade de se identificar um padrão no tempo de execução. Além disso, também são utilizados diferentes vetores nos testes, sendo eles aleatórios, quase ordenados ou inversos, visto que isso pode influenciar na execução de certos algoritmos, principalmente os mais simples, como da bolha, inserção e seleção.

3 Análise de Complexidade

A análise de complexidade de cada método de organização foi relizada utilizando-se a quantidade de comparações, movimentações, espaço e tempo feitas por cada algoritmo. Isso se mantém mesmo a partir das chamadas recursivas em casos onde o algoritmo foi implementado recursivamente. Abaixo, cada método é brevemente sintetizado e então seu custo é discutido.

- Método da Bolha:

  O método da bolha foi implementado não recursivamente, no seu funcionamento ele compara o elemento atual com o elemento seguinte através de dois "for"para poder organizar o vetor, com isso a comparação acontece $n-1, n-2, ..., 1$ vezes e o número de comparações é $\frac{n^2 - n}{2}$,portanto a complexidade em termo de comparações é $O(n^2)$. Quanto as movimentações realizadas por esse método, para cada vez que compararmos um elemento com o próximo e verificarmos qual for maior, iremos trocá-los até que estejam em sua devida posição, sendo assim temos dois possíveis casos: 1- Melhor caso, onde os elementos já estão devidamente ordenados e portanto, não há

movimentações, com complexidade $O(1)$, 2- Pior caso, onde o vetor está inversamente ordenado e em toda comparação, são feitas 3 movimentações para ordenar os elementos, logo, a complexidade é $3 * C(n)$, ou seja $O(n^2)$.

Pode então concluir-se que a complexidade temporal do método da bolha é $O(n^2)$.

Quanto à complexidade espacial, o método da bolha utiliza apenas um inteiro para realizar as movimentações e dois inteiros para iterar, logo a complexidade espacial é constante $O(1)$.

- Método da Inserção:

  O método da inserção é aquele utilizado pelos jogadores de carta, organizando um elemento comparando-o da esquerda para a direita e vendo sua posição correta. Partindo dessa ideia e analisando o número de comparações, é perceptível que o método da inserção possui um melhor e um pior caso, visto que ele possui dois "for"onde, o de fora itera de 1 até $n-1$, para n como o tamanho do vetor, e o interno itera até que o elemento i seja maior que algum dos seus anteriores, portanto, no melhor caso, com o vetor já ordenado, o laço interno ocorrerá apenas uma vez pois i será maior que seu anterior, ou seja a complexidade em termos de comparações é $O(n)$. Por outro lado no pior caso, com o vetor inversamente ordenado, o laço interno tem i comparações, assim a complexidade assume $O(n^2)$.

  Quanto às movimentações do método da inserção, no melhor caso, em que não se entra no laço interno, as movimentações são duas para cada laço externo, sendo $2(n-1)$ e portanto $O(n)$, já no pior caso, em que o laço interno sempre é acessado temos que são realizadas $i-1$ movimentaçoes, como i varia de 1 a $n-1$ temos que internamente são feitas $\frac{(n-1)n}{2}$ comparações e externamente $2(n-1)$logo no pior caso as movimentações tem complexidade $O(n^2)$.

  Pode então concluir-se que a complexidade temporal do método da inserção é $O(n^2)$.

  Quanto a complexidade espacial, ela é definida pela criação de três inteiros, ou seja custo constante $O(1)$.

- Método da Seleção:

  O método da seleção é um método que seleciona o n-ésimo menor (ou maior) elemento e troca sua posição para que ele esteja na n-ésima posição. Assim como no método da bolha, o método da seleção possui dois "for", o "de fora"com i que vai de 0 até $n-2$ e o de dentro j ,que vai de $i+1$ até $n-1$, logo ele realiza $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)$ comparações e portanto seu custo é $O(n^2)$.

  Qunato às movimentações, ele realiza três movimentações para cada iteração do "for"de fora, fazendo uma troca de 2 elementos do vetor, portanto, temos $3 * (n-1)$

movimentações e a complexidade de movimentações é $O(n)$.

Quanto à complexidade temporal, ela é dada por $O(n^2)$. Ademais, pela mesma razão explicada no método de inserção a complexidade espacial é $O(1)$.

- Merge Sort:

  O merge sort é um algoritmo que utiliza a ideia de divisão e conquista para ordernar o vetor. Ele foi implementado de maneira recursiva e realiza basicamente duas tarefas, a primeira é utilizando recursão para dividir partição atual em duas, chamando um merge sort para cada uma delas e no fim de cada um desses dois merges a função chama outra funçao merge para juntar todas as partições menores em uma só, ordenando o vetor com custo linear n. Dessa maneira o custo é dado por $T(n) = 2T(n/2) + n$, e como em $f(n) = n = n^{(log_2 2)}$ pode aplicar-se o caso 2 do Teorema Mestre, o custo temporal é dado por $T(n) = \Theta(n * logn)$.

  Quanto à complexidade espacial o Merge Sort utiliza dois vetores de tamanho $n/2$, independentemente de n ser par ou ímpar, portanto a complexidade espacial é $O(n)$.

- Quick Sort:

  O quick sort , assim como o merge sort, é um algoritmo que utiliza a ideia de divisão e conquista para a ordenação do vetor. Ele foi implementado de maneira padrão utilizando a escolha arbritária de pivô, e também utilizando a mediana de três para alterar seu funcionamento. Na versão padrão, temos 3 possíveis casos para a seleção aleatória do pivô: 1 - O pior caso, onde o pivô escolhido é o maior ou menor elemento do vetor e portanto o procedimento de ordenação é chamado n vezes, eliminando apenas um elemento por chamada e tendo custo $T(n) = T(n-1) + n$, e portanto $O(n^2)$, 2 - O melhor caso, onde a cada partição o conjunto é dividido em 2 partes iguais e seu custo é $T(n) = 2T(\frac{n}{2}) + n$, e portanto $O(nlogn)$, 3 - O caso médio, dado por $C(n) = 1,386nlogn - 0,846n$, que significa em média custo $O(nlogn)$. Na versão com mediana de 3, o pior caso é evitado e isso melhora na maioria dos casos o funcionamento do algoritmo.

  Quanto à complexidade espacial, para o quicksort é criado um número fixo e finito de variáveis, logo, a complexidade espacial é $O(1)$.

- Shell Sort:

  O shell sort é, resumindo em poucas palavras, uma melhoria do método de inserção, pois conserta seu principal problema, que é poder trocar apenas elementos vizinhos, passando a trocá-los em itens separados por h. O método de inserção é melhor para listas quase ordenadas e de menor tamanho, porém o shell sort se mostra mais efetivo conforme o tamanho e a desorganização aumentam. Para o algoritmo, algumas sequências para o valor do h podem ser utilizadas e com isso a complexidade de tempo varia, mas a escolhida aqui foi unicamente a sequência $\frac{n}{2^i}$, onde a complexidade é

dada por $O(n^2)$.

Quanto à complexidade espacial, o shell sort é um algoritmo de ordenação in-place, logo seu custo espacial é $O(1)$.

- Couting Sort:

  O counting sort é um dos chamados algoritmos de ordenação sem comparação, ele utiliza o maior elemento do vetor dado (previamente conhecido) e cria um espaço de memória referente ao valor deste maior elemento, e após percorrer todo o vetor armazenando os elementos menores, ordena-os. Implementado de maneira padrão e considerando k o maior elemento do vetor, a complexidade de espaço é dada por $O(n+k)$, sendo a passagem pelo vetor para contagem $O(n)$ e a passagem pelo contador para imprimir os elementos $O(k)$.

  Quanto à complexidade espacial, o counting sort reserva um espaço referente ao maior elemento k e portanto seu custo é $O(k)$.

- Bucket Sort:

  O ucket sort é um algoritmo que separa os elementos do vetor original em buckets de tamanho menor, e, utilizando algum outro algoritmo ou listas encadeadas, ordena os elementos dos buckets. Implementado utilizando listas encadeadas para a ordenação dos buckets, a complexidade de tempo para inserir cada elemento é dada por $\frac{n}{k}$ e a complexidade total é $O(\frac{n^2}{k})$.

  Quanto à complexidade espacial, o bucket sort tem complexidade de $O(n + k)$, para k listas com n elementos no total.

- Radix Sort:

  O radix sort é um algoritmo de ordenação que utiliza a representação binária dos elementos para a ordenação, com funcionamento similar ao quicksort, só que comparando os bits ao invés de chaves. Implementado de forma padrão, o radix sort tem complexidade dada pela ordenação dos dígitos $O(n + k)$ (k é equivalente ao número de diferentes valores assumidos pelo elemento, no caso estudado ele tem 2 possíveis valores, 0 e 1) e pelo número de passes d, logo sua complexidade de tempo total é $O(d.(n + k))$.

  Quanto à complexidade espacial, o radix sort utiliza dois arrays para realizar a ordenação, um de contagem $O(k)$ e um auxiliar $O(n)$, logo, sua complexidade total é $O(n + k)$.

4 Estratégias de Robutez

Entre algumas das estratégias de robustez utilizadas, estão aquelas que evitam o uso desnecessário de memória e o uso desnecessário de tempo na execução. Quanto a essas es-

tratégias, destacam-se a identificação de padrões nos vetores utilizados, evitando assim que um possível vetor já ordenado execute de maneira completa (isso foi feito principalmente no método da bolha, seleção e inserção, onde se em uma passagem do vetor não é efetuada trocas, o algoritmo é interrompido pois o vetor já está ordenado). Além disso, em algoritmos onde a memória utilizada pode atingir valores altos, como o bucket sort e o counting sort, sempre é garantido que o algoritmo pare caso a memória não seja suficiente, evitando assim valores incorretos nos testes feitos.

5 Análise experimental

Comparação dos métodos da bolha, seleção e inserção, com vetor aleatório, quase ordenado e inversamente ordenado:

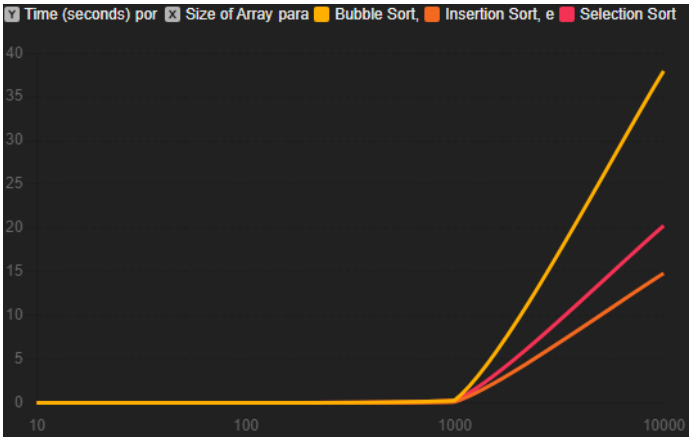

Figura 1: Gráfico com vetor aleatório

Size	Bubble Sort (s)	Insertion Sort (s)	Selection Sort (s)
10	0.000059	0.000052	0.000039
100	0.003482	0.002050	0.001610
1000	0.312546	0.143576	0.260148
10000	37.975309	14.820652	20.255598

Figura 2: Tempo de execução

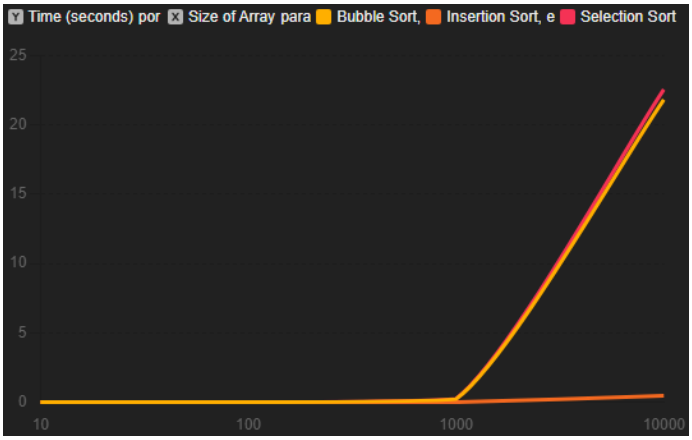

Figura 3: Gráfico com vetor semi-ordenado

Size	Bubble Sort (s)	Insertion Sort (s)	Selection Sort (s)
10	0.000085	0.000158	0.001260
100	0.002461	0.000359	0.005005
1000	0.227570	0.004389	0.251151
10000	21.801012	0.477526	22.539043

Figura 4: Tempo de execução semi-ordenado

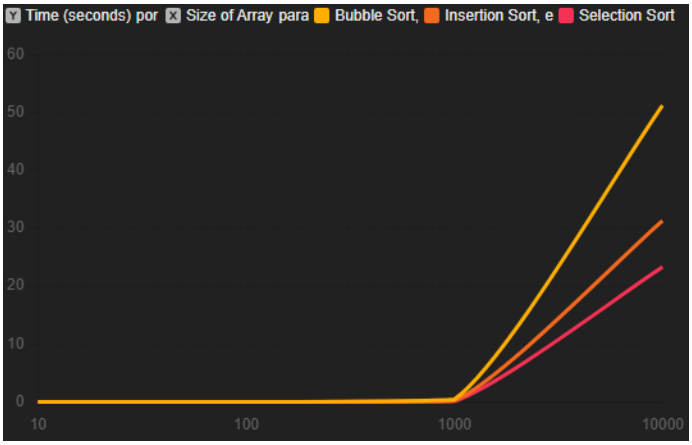

Figura 5: Gráfico com vetor inverso

Size	Bubble Sort (s)	Insertion Sort (s)	Selection Sort (s)
10	0.000166	0.000084	0.000216
100	0.007247	0.003219	0.002388
1000	0.456216	0.217351	0.154114
10000	51.159482	31.284420	23.272604

Figura 6: Tempo de execução inverso

Comparação, com vetores aleatórios, do Quicksort, Shellsort e Mergesort:

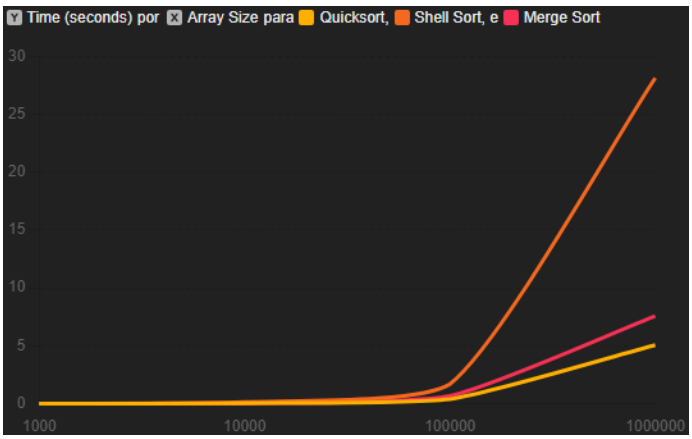

Figura 7: Gráfico vetores aleatórios

	Array Size	Quicksort (s)	Shell Sort (s)	Merge Sort (s)
1	1000	0.002748966217041 0156	0.005397558212280 2734	0.004112720489501 953
2	10000	0.039633035659790 04	0.138855934143066 4	0.046175479888916 016
3	100000	0.405801296234130 86	1.694027662277221 7	0.683494806289672 9
4	1000000	5.069836854934692	28.11667895317077 6	7.571078062057495

Figura 8: Tempo de execução

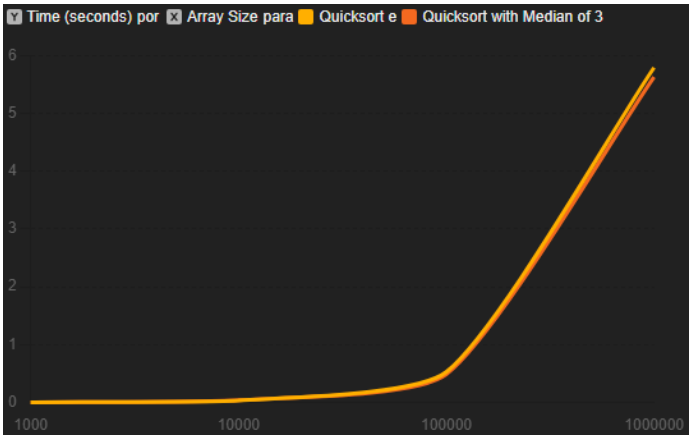

Figura 9: Comparação Quicksort e Quicksort med3

Comparação, com vetores aleatórios, do Radix sort, Counting sort e Bucket sort:

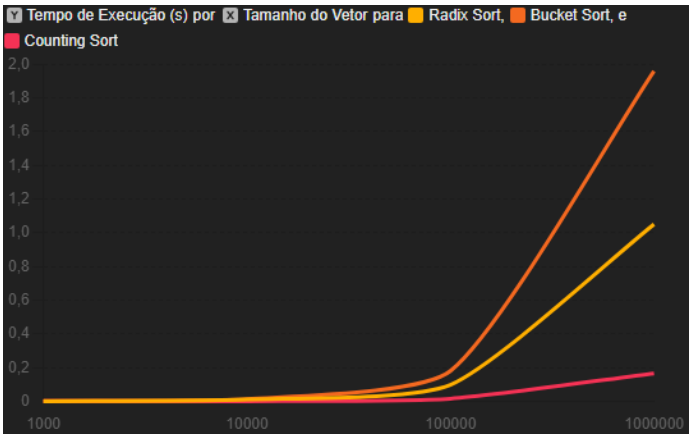

Figura 10: Gráfico vetores aleatórios

	Tamanho do Vetor	Radix Sort (s)	Bucket Sort (s)	Counting Sort (s)
1	1000	0.000879764556884 7656	0.003945827484130 859	0.000214338302612 3047
2	10000	0.011250495910644 531	0.014313459396362 305	0.001228570938110 3516
3	100000	0.098876237869262 7	0.180787324905395 5	0.016504287719726 562
4	1000000	1.050214767456054 7	1.959301471710205	0.166067123413085 94

Figura 11: Tempo de execução

6 Conclusões

Diante dos dados expostos acima, é possível fazer certas conclusões. Cada algoritmo tem sua individualidade, sendo melhor utilizado em certas situações, mas de maneira geral os algoritmos que não utilizam comparação se mostram mais eficientes que o restante, diminuindo o tempo de execução por utilizar um maior espaço na memória, especialmente o counting sort, que ao alocar memória equivalente ao maior elemento, pode rapidamente realizar a ordenação no menor tempo.

Considerando os algoritmos com menor necessidade de utilização da memória, aqueles que utilizam divisão e conquista são bem mais rápidos que os algoritmos mais simples. Entre esses, o quicksort acabou apresentando os melhores resultados. Além disso, analisando apenas o quicksort, a utilização de ferramentas que evitam pior caso como a mediana de 3 melhoram o resultado geral do algoritmo.

Entre os algoritmos mais simples, como o bolha, seleção e inserção, para vetores aleatórios e quase ordenados o método de inserção se mostrou o mais eficiente, porém com a inversão da ordenação do vetor, o método de seleção acabou mostrando um tempo menor de execução.

Concluindo, a utilização de cada método de ordenação pode ser eficiente caso o vetor e o hardware utilizado sejam conhecidos. Caso a finalidade seja economizar espaço ou apenas executar a ordenação no menor tempo possível, é possível a utilização de diversos métodos para alcançar essa finalidade, tudo depende da necessidade imposta na hora da ordenação.

7 Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados.

https://medium.com/@romualdo.v/análise-de-desempenho-e-complexidade-dos-algoritmos-de-ordenação-f47449e93b33