Análise de Algoritmos

Samuel de Souza Gomes

[1]Instituto de Ciências Exatas e Aplicadas – Universidade Federal de Ouro Preto (UFOP)
Rua 36, 115 - Loanda, João Monlevade - MG, 35931-008

samuel.gomes2@aluno.ufop.edu.br

Abstract. *This report aims to study some of the main sorting methods. This will make comparisons between them through their execution time, number of assignments and comparisons and also the advantage of using certain method in a given situation.*

Resumo. *Este relatório tem como objetivo o estudo de alguns dos principais métodos de ordenação. Com isso, serão feitas comparações entre eles através do seu tempo de execução, número de atribuições e comparações e também a vantagem de usar determinado método em determinada situação.*

1. Introdução

O estudo em questão será divido em duas etapas, na qual a primeira se trata dos algoritmos mais simples de complexidade O(n^2) (Bubble Sort, Selection Sort, Insertion Sort e suas otimizações) e a segunda parte dos algoritmos ditos eficientes (Shell Sort, Quick Sort, Heap Sort e Merge Sort). Para isso, foi utilizado um programa que gera arrays de várias formas para realizar os testes.

Através dos vários testes foi possível comparar e perceber a diferença entre os algoritmos que no geral realizam a mesma tarefa, porém, de formas, tempo e espaço distintos.

2. Implementação

O programa foi modelado de forma que pudesse extrair os resultados de forma direta e simples para a realização dos testes. O mesmo possui arquivos headers, que é onde se declaram funções em linguagem C com o objetivo de usá-las em qualquer lugar do código. Tais arquivos conhecidos como TADs (Tipo Abstrato de Dados) são utilizados para modularizar o programa, dividir em vários arquivos. Dessa forma, pode-se ler, realizar manutenção e encontrar erros com mais facilidade.

Além dos TADs, o programa dispõe de outros arquivos .c que contém as implementações da funções declaradas nos arquivos headers, além das função principal (main) que executa todo o programa.

Ademais, um arquivo .c em específico (sort.c) contém a implementação de todos os métodos de ordenação em estudo. Esses métodos recebem como parâmetro um ponteiro de array, um ponteiro que contará o número de atribuições e um outro ponteiro que contará a quantidade de comparações do método.

A função principal (main.c), onde tudo funciona, chama todas as funções criadas nos outros arquivos. Certamente, o array precisa alocar um espaço na memória, e portanto, a main chama a função referente à essa tarefa primeiramente. Por conseguinte

chama o método de ordenação que será utilizado, indica o tamanho do array, uma outra função gera a situação do array (ordem crescente, ordem inversa, ordem aleatória ou quase ordenado). Por fim, outra função é chamada para calcular o tempo de execução daquele algoritmo. É importante citar que todos os parâmetros são passados por linha de comando, e apenas dessa forma o programa pode ser executado.

3. Estudo de Complexidade

Como foi dito incialmente, o trabalho foi divido em duas partes, onde a primeira se trata dos algoritmos de complexidade O(n^2) (Selection Bubble, e Insertion) e os algoritmos de complexidade O(nlog(n)) (Shell, Quick, Heap e Merge Sort). Os mesmos têm essas complexidades porque:

3.1. Bubble Sort

- Complexidade de Tempo
Melhor caso: O(n) - versão otimizada (vetor ordenado)
Pior caso: O(n^2)
- Complexidade de Memória
Espaço: O(1)
- Como a complexidade de tempo foi deduzida:

Quando o vetor já está ordenado o método percorre o array apenas 1 vez para a verificação, e esse é o melhor caso O(n). Já no pior caso, se o vetor não está ordenado ele precisa comparar a posição atual com a adjacente n*n vezes para realizar a troca, O(n^2).

No pior caso o método vai realizar n*n comparações com as posições adjacentes da atual, enquanto o vetor não estiver completamente ordenado. Já no melhor caso, o método realiza essas comparações n vezes, pois não precisará ordenar o vetor.

A complexidade tempo no caso médio do Bubble Sort é também O(n^2). Ocorre quando a matriz obtêm os valores aleatórios, nesse caso compara-se novamente somente os valores não ordenados. Pontos Positivos do Bubble: Simples de implementar. Utiliza armazenamento temporário (in-place).

3.2. Selection Sort

- Complexidade de Tempo
Melhor caso: O(n^2)
Pior caso: O(n^2)
- Complexidade de Memória
Espaço: O(1)
- Como a complexidade de tempo foi deduzida:

O selection sort compara a cada iteração um elemento com os outros, visando encontrar o menor. Dessa forma, pode-se entender que não existe um melhor caso, mesmo que o vetor esteja ordenado ou em ordem inversa serão executados os dois laços do algoritmo, o externo e o interno. Portanto, em ambos os casos a complexidade será O(n^2).

Como não tem um melhor caso, o algoritmo tem o mesmo comportamento sempre. Na primeira iteração do laço externo o índice começa de 0 e a cada iteração ele soma uma unidade até o final do vetor e o laço mais interno percorre o vetor começando desse

índice externo + 1 até o final do vetor. Dentro dos laços o Selection ordena um vetor ao repetidamente encontrar o menor elemento (considerando a ordem crescente) da parte não ordenada e colocando-o no início.

A complexidade de tempo no caso médio do Selection Sort também é O(n^2), pois em todos os casos o algoritmo se comporta da mesma forma. Pontos positivos: é um algoritmo simples de ser implementado em comparação aos demais; Não necessita de um vetor auxiliar (in-place); Por não usar um vetor auxiliar para realizar a ordenação, ele ocupa menos memória; É um dos mais velozes na ordenação de vetores de tamanhos pequenos. Pontos negativos: ele faz sempre (n^2 - 2) / 2 comparações, independente do vetor está ordenado ou não; Ele é um dos mais lentos para vetores de tamanhos grandes; Ele não é estável.

3.3. Insertion Sort

- Complexidade de Tempo
Melhor caso: O(n)
Pior caso: O(n^2)
- Complexidade de Memória
Espaço: O(1)
- Como a complexidade de tempo foi deduzida:

Quando o vetor já está ordenado o método percorre o array apenas 1 vez para a verificação, e esse é o melhor caso O(n). Já no pior caso, o método faz n * n verificações, pois se o vetor não está ordenado ele precisa comparar uma posição com as outras.

O melhor caso acontece quando o vetor já está todo ordenado, e o comportamento do Insertion Sort nesse caso será percorrer o vetor apenas 1 vez para verificar se está realmente ordenado. O pior caso acontece quando o vetor está em ordem inversa e o método precisa ordenar cada posição realizano n * n comparações.

A complexidade de tempo no caso médio do Insertion Sort é O(n^2). Acontece quando o vetor contêm valores aleatórios sem ordem de classificação (crescente ou descrescente).

Positivos positivos: É o método a ser utilizado quando o vetor está "quase" ordenado; É um bom método quando se desejar adicionar poucos elementos em um arquivo já ordenado, pois seu custo é linear; O algoritmo é estável. Negativos: Alto custo de movimentação de elementos no vetor.

3.4. Shell Sort

- Complexidade de Tempo
Melhor caso: O(n*log(n))
Pior caso: O(n*log(n^2))
- Complexidade de Memória
Espaço: O(1)
- Como a complexidade de tempo foi deduzida:

Como neste algoritmo a classificação é aplicada no grande intervalo de elementos e, em seguida, o intervalo é reduzido em uma sequência, o tempo de execução do tipo Shell é altamente dependente da sequência de lacunas que ele usa. Ou seja, melhor caso no

tipo shell é quando o array já está classificado. O número de comparações é menor. É um algoritmo de ordenação in-place, pois não requer espaço adicional.Shell é um algoritmo instável, pois a ordem relativa de elementos com valores iguais pode mudar. O caso médio depende da sequência de lacunas (gaps).

3.5. Quick Sort

- Complexidade de Tempo
Melhor caso: O(n*log(n))
Pior caso: O(n^2))
- Complexidade de Memória
Espaço: O(log(n))
- Como a complexidade de tempo foi deduzida:

O pior caso de particionamento ocorre quando o elemento pivô divide a lista de forma desbalanceada, ou seja, divide a lista em duas sub listas: uma com tamanho 0 e outra com tamanho n - 1 (no qual n se refere ao tamanho da lista original). Isso pode ocorrer quando o elemento pivô é o maior ou menor elemento da lista, ou seja, quando a lista já está ordenada, ou inversamente ordenada.

O melhor caso de particionamento acontece quando ele produz duas listas de tamanho não maior que n/2, uma vez que uma lista terá tamanho [n/2] e outra tamanho [n/2] - 1. Nesse caso, o quicksort é executado com maior rapidez.

3.6. Heap Sort

- Complexidade de Tempo
Melhor caso: O(n*log(n))
Pior caso: O(n*log(n))
- Complexidade de Memória
Espaço: O(1)
- Como a complexidade de tempo foi deduzida:

Para empilhar completamente um elemento cujas subárvores já são max-heaps, precisamos continuar comparando o elemento com seus filhos da esquerda e da direita e empurrando-o para baixo até que ele atinja um ponto em que ambos os filhos sejam menores que ele. Na pior das hipóteses, precisaremos mover um elemento da raiz para o nó folha fazendo um múltiplo de comparações e swaps de log(n).

Durante a etapa de ordenação, trocamos o elemento raiz pelo último elemento e empilhamos o elemento raiz. Para cada elemento, isso toma novamente o pior momento do log(n), pois talvez tenhamos que trazer o elemento todo o caminho da raiz para a folha. Como repetimos isso n vezes, o passo heap-sort também é n*log(n).

4. Listagem de Testes Executados

Seguem os testes realizados com os respectivos algoritmos e suas otimizações.

4.1. Bubble, Selection, Insertion

- Vale a pena inserir a verificação de ordenação no Bubble Sort?
Não, pois realizará o trabalho de ordenação da mesma forma, porém, se não tiver a

Bubble Sort / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	81	9801	998001	99980001	9999800001	-
Atribuições	0	135	14847	1498500	149984142	14999841981	-
Tempo	0.000006 s	0.000010 s	0.000242 s	0.018481 s	0.964442 s	96.221172 s	Mais de 1 hora
Bubble Sort / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	81	9801	998001	99980001	9999800001	-
Atribuições	0	0	0	0	0	0	0
Tempo	0.000005 s	8	0.000163 s	0.010243 s	0.606161 s	60.027634 s	Mais de 1 hora
Bubble Sort / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	81	9801	998001	99980001	9999800001	-
Atribuições	0	55	6346	683650	67301029	6786830000	-
Tempo	0.000005 s	0.000010 s	0.000231 s	0.01413667 s	0.63068233 s	104.058684 s	Mais de 1 hora
Bubble Sort / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	81	9801	998001	99980001	9999800001	-
Atribuições	0	0	271	17164	1815909	197772366	-
Tempo	0.000002 s	0.000008 s	0.000132 s	0.001174 s	0.635226 s	63.194079 s	Mais de 1 hora

Figure 1. Bubble Sort

Bubble Sort Otimizado / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	135	14847	1498500	149984142	14999841981	-
Tempo	0.000006 s	0.000010 s	0.000242 s	0.018481 s	0.642893 s	63.853122 s	Mais de 1 hora
Bubble Sort Otimizado / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	0	0	0	0	0	0
Tempo	0.000005 s	0.000006 s	0.000046 s	0.005174 s	0.303583 s	30.320871 s	Mais de 1 hora
Bubble Sort Otimizado / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	44	7118	662311	68018541	6786830000	-
Tempo	0.000005 s	0.000010 s	0.000137 s	0.004414 s	0.629317 s	72.339384 s	Mais de 1 hora
Bubble Sort Otimizado / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	0	195	18356	1884192	191065485	-
Tempo	0.000002 s	0.000008 s	0.000132 s	0.001174 s	0.318897 s	31.629878 s	Mais de 1 hora

Figure 2. Bubble Sort Otimizado

Selection Sort / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	15	155	1500	15858	158019	-
Tempo	0.000004 s	0.000010 s	0.000087 s	0.006009 s	0.308108 s	30.199934 s	Mais de 1 hora
Selection Sort / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	0	0	0	0	0	0
Tempo	0.000005 s	0.000008 s	103	0.005174 s	0.303583 s	30.320871 s	Mais de 1 hora
Selection Sort / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	18	238	2525	25509	253293	-
Tempo	0.000005 s	0.000010 s	0.000137 s	0.004414 s	0.296133 s	29.446325 s	Mais de 1 hora
Selection Sort / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	0	3	30	321	3207	-
Tempo	0.000002 s	0.000008 s	84	0.005852 s	0.318897 s	31.629878 s	Mais de 1 hora

Figure 3. Selection Sort

verificação economiza no número de comparações, mas o tempo é o mesmo.

- Vale a pena utilizar o elemento sentinela no Insertion Sort?

Selection Otimizado Sort / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	15	153	1500	15858	158019	-
Tempo	0.000004 s	0.000006 s	0.000079 s	0.005847 s	0.304538 s	30.199934 s	Mais de 1 hora
Selection Otimizado Sort / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	0	0	0	0	0	0
Tempo	0.000005 s	0.000008 s	0.000064 s	0.005174 s	0.290644 s	28.580221 s	Mais de 1 hora
Selection Otimizado Sort / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	18	243	2469	25476	254097	-
Tempo	0.000005 s	0.000010 s	0.000098 s	0.005664 s	0.285627 s	28.356134 s	Mais de 1 hora
Selection Otimizado Sort / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	0	3	30	315	3228	-
Tempo	0.000011 s	0.000008 s	0.000069 s	0.005852 s	0.290166 s	28.346906 s	Mais de 1 hora

Figure 4. Selection Sort Otimizado

Insertion Sort / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	135	14847	1498500	149948142	14999841981	-
Tempo	0.000004 s	0.000010 s	0.000087 s	0.006009 s	0.308108 s	40813760 s	Mais de 1 hora
Insertion Sort / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	0	0	0	0	0	0
Atribuições	0	18	198	1998	19998	199998	1999998
Tempo	0.000005 s	0.000008 s	0.000009 s	0.000022 s	0.000196 s	0.000900 s	0.012032 s
Insertion Sort / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	25	2327	214997	22697859	2262436306	-
Atribuições	0	75	6981	644991	68093577	2262436306	-
Tempo	0.000005 s	0.000010 s	0.000048 s	0.003445 s	183.530	18.629.317	Mais de 1 hora
Insertion Sort / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	0	107	9178	543856	63210980	6610192459
Atribuições	0	0	321	27534	1631568	189632940	6612192457
Tempo	0.000002 s	0.000008 s	0.000012 s	0.000172 s	0.004642 s	523.663	54.680079 s

Figure 5. Insertion Sort

Insertion Sort Otimizado / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	45	4950	499500	49995000	4999950000	-
Atribuições	0	63	5147	501498	50014712	5000147325	-
Tempo	0.000004 s	0.000010 s	0.000110 s	0.007855 s	0.456401 s	44.155246 s	Mais de 1 hora
Insertion Sort Otimizado / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	0	0	0	0	0	0
Atribuições	0	18	198	1998	19998	199998	1999998
Tempo	0.000005 s	0.000008 s	0.000009 s	0.000022 s	0.000196 s	0.000900 s	0.012032 s
Insertion Sort Otimizado / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	21	2327	214997	22774697	2262436306	-
Atribuições	0	39	2544	225691	22794695	2262436306	-
Tempo	0.000005 s	0.000006 s	0.000048 s	0.004529 s	0.218067 s	18.629.317	Mais de 1 hora
Insertion Sort Otimizado / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	0	13	4596	543856	63210980	6597180689
Atribuições	0	18	211	6594	603632	65328552	6612192457
Tempo	0.000002 s	0.000008 s	0.000012 s	0.000141 s	0.004642 s	0.612322 s	54.680079 s

Figure 6. Insertion Sort Otimizado

Sim, pois ele aumenta o número de atribuições, mas diminui o tempo de execução.

- Vale a pena inserir uma verificação (min == i) para evitar troca, no Selec-

tion Sort?
Como o Selection tem a mesma complexidade em todos os casos e analisando os testes, essa verificação não fez tanta diferença, principalmente quando o array já está todo ordenado ou quando é gerado em ordem inversa. Contudo, em um array muito grande e gerado de forma aleatório, a diferença no número de atribuições pode ser significante, então, nesse caso vale a pena.

Shell Sort / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	15	235	3926	53891	621157	6248662
Atribuições	0	43	915	14834	204368	2555438	29857179
Tempo	0.000004 s	0.000007 s	0.000019 s	0.000200 s	0.002862 s	0.016781 s	194.485
Shell Sort / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	2	6	6	582	5328	9677
Atribuições	0	30	686	10914	151059	1939609	23618194
Tempo	0.000005 s	0.000008 s	0.000017 s	0.000168 s	0.001755 s	0.013121 s	135.530
Shell Sort / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	17	413	8696	157071	3271693	6248662
Atribuições	0	45	1093	19604	307548	5205974	29857179
Tempo	0.000005 s	0.000006 s	0.000033 s	0.000524 s	0.007231 s	0.059465 s	0.194485 s
Shell Sort / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	2	86	2112	67828	1271280	23448584
Atribuições	0	30	766	13020	218305	3205561	47057101
Tempo	0.000002 s	0.000008 s	0.000027 s	0.000237 s	0.003704 s	0.036716 s	0.548105 s

Figure 7. Shell Sort

Quick Sort / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	1	5	50	500	5000	50000	500000
Atribuições	3	15	150	1500	15000	150000	1500000
Tempo	0.000004 s	0.000005 s	0.000008 s	0.000017 s	0.000230 s	0.000710 s	0.007087 s
Quick Sort / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	1	1	1	1	1	1	1
Atribuições	3	3	3	3	3	3	3
Tempo	0.000005 s	0.000008 s	0.000012 s	0.000015 s	0.000068 s	0.000527 s	0.003952 s
Quick Sort / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	1	2	13	202	1921	21523	210433
Atribuições	3	3	36	603	5763	64566	631299
Tempo	0.000005 s	0.000006 s	0.000008 s	0.000033 s	0.000257 s	0.001463 s	0.014170 s
Quick Sort / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	1	1	1	6	52	511	4920
Atribuições	3	3	3	18	156	1533	14760
Tempo	0.000002 s	0.000008 s	0.000007 s	0.000015 s	0.000133 s	0.000429 s	0.004898 s

Figure 8. Quick Sort

- Até que tamanho de entrada vale a pena usar algoritmos O(n^2) com relação aos algoritmos O(nlog(n))?
De acordo com o estudo feito em cima desses algoritmos, percebe-se que cada método tem suas características, apesar de todos realizarem o mesmo trabalho. Com isso, é interessante analisar antes que tipo de trabalho o usuário/programador quer realizar, pois o melhor método utilizado vai sempre depender do caso. Ou seja, os algoritmos de ordenação simples para arrays pequenos são muito úteis, contudo, à partir de 1000 elementos a diferença de complexidade começa ser bastante significativa.

Obs.: Não foi possível implementar o Merge Sort porque ele necessita de uma

Heap Sort / Ordem Inversa							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	19	478	7991	113380	1463422	18001530
Atribuições	2	40	851	13266	182896	2329680	28266210
Tempo	0.000009 s	0.000007 s	0.000043 s	0.000421 s	0.004725 s	0.30485 s	0.359407 s
Heap Sort / Ordem Crescente							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	23	548	8813	122274	1556835	18864295
Atribuições	2	54	1055	15645	206490	2571965	30728747
Tempo	0.000005 s	0.000007 s	0.000044 s	0.000440 s	0.005436 s	0.031477 s	0.371348 s
Heap Sort / Ordem Aleatória							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	21	530	8468	118059	1514896	18445795
Atribuições	2	52	997	14790	197043	2472117	29724658
Tempo	0.000005 s	0.000009 s	0.000041 s	0.000587 s	0.004882 s	0.043399 s	0.553351 s
Heap Sort / Quase Ordenado							
Tamanho	1	10	100	1000	10000	100000	1000000
Comparações	0	23	548	8793	122190	1555652	18858580
Atribuições	2	54	1051	15616	206342	2569118	30713630
Tempo	0.000006 s	0.000008 s	0.000049 s	0.000327 s	0.003586 s	0.033294 s	0.377512 s

Figure 9. Heap Sort

outra função recursiva. Com isso, não consegui alterar o código suporte disponibilizado para que ele pudesse ser executado e testado.

5. Conclusão

Após vários testes, foi possível notar que não existe o melhor método de ordenação, isso vai depender do tamanho do array, se o algoritmo necessita de memória extra, da quantidade de atribuições e comparações. É um trabalho simples e objetivo que ajuda entender um pouco mais do funcionamento dos métodos e que deixa mais clara a importância de cada um deles.

Contudo, surgiu um problema na execução do programa, no qual logo foi resolvido, que era a passagem de parâmetros por linha de comando. A execução de tal funcionalidade era desconhecida até o desenvolvimento do trabalho em questão, o que também serviu de aprendizado.

6. Referências

WIKIPÉDIA. Algoritmo de Ordenação. Disponível em: https://pt.wikipedia.org/wiki/Algoritmo-de-ordenação. Acesso em: 06 jun. 2019.

ZIVIANI, Nivio. Solução e Documentação de Trabalhos Práticos. Disponível em: https://homepages.dcc.ufmg.br/ nivio/cursos/aed2/roteiro/. Acesso em: 06 jun. 2019

FORTES, Reinaldo. Ordenação: Bubble, Selection, Insertion Sort. Disponível em: http://www.decom.ufop.br/reinaldo/site-media/uploads/2013-02-bcc202/aula12bubblesort-selectionsort-insertionsort-(v1).pdf. Acesso em: 06 jun. 2019