UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
DEPARTAMENTO DE INFORMÁTICA

Elaine Dias Pires
Filipe Gomes Arante de Souza

1º Relatório de Estrutura de Dados II
Ordenação Parcial em Memória Principal

Fevereiro / 2022

Sumário

1 Introdução

Este trabalho tem o objetivo de estudar o comportamento de alguns dos principais algoritmos de ordenação parcial a fim de fixar o conteúdo ensinado em aula. A ordenação implementada no trabalho será a ordenação decrescente, pois é necessário ordenar os K maiores elementos de um vetor. Assim sendo, os algoritmos analisados serão os seguintes:

- Selection Sort Parcial;
- Insertion Sort Parcial;
- Shell Sort Parcial;
- Quick Sort Parcial;
- Heap Sort Parcial;

O programa desenvolvido neste trabalho é capaz de gerar algumas estatísticas, tais como quantidade de comparações, swaps, tempo de execução e os top K maiores elementos pertencentes a um conjunto de números inteiros.

2 Selection Sort Parcial

2.1 Lógica de implementação

Na ordenação decrescente, o selection sort seleciona o maior elemento do vetor e o coloca na primeira posição; seleciona o segundo maior elemento e o coloca na segunda posição e assim sucessivamente. Uma vez que o trabalho pede uma ordenação parcial, esse procedimento de selecionar os k maiores elementos é realizado k vezes.

2.2 Resultados

2.2.1 Para k = 100

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	9.994.950	100	0,06
	1kk	99.994.950	100	0,25
	10kk	999.994.950	100	2,65
Aleatório 2	100k	9.994.950	100	0,03
	1kk	99.994.950	100	0,25
	10kk	999.994.950	100	2,82
Ordenado	100k	9.994.950	100	0,06
	1kk	99.994.950	100	0,25
	10kk	999.994.950	100	3,07
Parcialmente Ordenado	100k	9.994.950	100	0,03
	1kk	99.994.950	100	0,32
	10kk	999.994.950	100	2,64
Invertido	100k	9.994.950	100	0,04
	1kk	99.994.950	100	0,29
	10kk	999.994.950	100	2,82

Tabela 1: Dados do selection sort parcial para k = 100.

2.2.2 Para k = 1000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	99.499.500	1.000	0,25
	1kk	999.499.500	1.000	2,67
	10kk	1.409.564.908	1.000	25,5
Aleatório 2	100k	99.499.500	1.000	0,23
	1kk	999.499.500	1.000	2,53
	10kk	1.409.564.908	1.000	25,59
Ordenado	100k	99.499.500	1.000	0,25
	1kk	999.499.500	1.000	2,67
	10kk	1.409.564.908	1.000	25,5
Parcialmente Ordenado	100k	99.499.500	1.000	0,25
	1kk	999.499.500	1.000	2,64
	10kk	1.409.564.908	1.000	25,46
Invertido	100k	99.499.500	1.000	0,25
	1kk	999.499.500	1.000	2,59
	10kk	1.409.564.908	1.000	25,51

Tabela 2: Dados do selection sort para k = 1000.

2.2.3 Para k = 10000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	949.995.000	10.000	2,7
	1kk	9.949.995.000	10.000	29,78
	10kk	99.949.995.000	10.000	287,93
Aleatório 2	100k	949.995.000	10.000	2,78
	1kk	9.949.995.000	10.000	29,54
	10kk	99.949.995.000	10.000	282,85
Ordenado	100k	949.995.000	10.000	2,48
	1kk	9.949.995.000	10.000	27,68
	10kk	99.949.995.000	10.000	282,76
Parcialmente Ordenado	100k	949.995.000	10.000	2,71
	1kk	9.949.995.000	10.000	28,34
	10kk	99.949.995.000	10.000	281,98
Invertido	100k	949.995.000	10.000	2,73
	1kk	9.949.995.000	10.000	29,60
	10kk	99.949.995.000	10.000	292,76

Tabela 3: Dados do selection sort para k = 10000.

Note que para o selection sort a configuração inicial do vetor não interfere em sua execução. Para um tamanho n e uma quantidade k de elementos a serem ordenados fixos, a quantidade de comparações e de trocas sempre será a mesma, enquanto o tempo de CPU sempre resultará em valores próximos. Teoricamente, a complexidade deste algoritmo é $O(k * n)$ e pudemos comprovar isso empiricamente, pois quando n aumentava em 10 vezes, as grandezas analisadas também tenderam a crescer em 10 vezes, e este mesmo comportamento também ocorreu com k.

3 Insertion Sort Parcial

3.1 Lógica de implementação

A ideia do insertion sort é de fato inserir um elemento em sua posição correta. Isto é feito da seguinte maneira: para cada elemento x, verificamos se os anteriores são menores que x, caso isso aconteça, são feitas sucessivas trocas até que todos os elementos anteriores a x sejam maiores que ele. Esse procedimento é realizados para todos os elementos do vetor.
Na ordenação parcial, o interesse é ordenar os k maiores elementos. Assim, até a $k - ésima$ iteração não há mudança no algoritmo, porém a partir dela, os próximos elementos do vetor serão comparados apenas com aqueles que estão nas k primeiras posições. Dessa forma, apenas as k primeiras posições estarão ordenadas.

3.2 Resultados

3.2.1 Para k = 100

Tipo da entrada	Tam.	Comp.	Trocas	Tempo (s)
Aleatório 1	100k	138.453	38.467	0,00
	1kk	1.048.002	48.021	0,00
	10kk	10.062.780	62.794	0,03
Aleatório 2	100k	137.492	37.509	0,00
	1kk	1.048.885	48.898	0,00
	10kk	10.061.325	61.341	0,03
Ordenado	100k	99.999	0	0,00
	1kk	999.999	0	0,01
	10kk	9.999.999	0	0,03
Parcialmente Ordenado	100k	100.133	135	0,00
	1kk	1.000.129	130	0,01
	10kk	10.000.121	123	0,01
Invertido	100k	9.994.950	9.994.949	0,03
	1kk	99.994.950	99.994.736	0,26
	10kk	999.994.904	999.971.785	2,28

Tabela 4: Dados do insertion sort para k = 100.

3.2.2 Para k = 1000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	2.668.010	2.568.024	0,01
	1kk	4.711.034	3.711.053	0,01
	10kk	14.949.532	4.949.546	0,03
Aleatório 2	100k	2.702.487	2.602.504	0,00
	1kk	4.657.172	2.602.504	0,01
	10kk	14.914.786	3.657.185	0,04
Ordenado	100k	99.999	0	0.00
	1kk	999.999	0	0.01
	10kk	99.999.999	0	0.01
Parcialmente Ordenado	100k	101.033	1.035	0,00
	1kk	1.001.295	1.296	0,01
	10kk	10.001.047	1.049	0,03
Invertido	100k	99.499.500	99.499.499	0,23
	1kk	999.499.500	999.499.286	2,4
	10kk	1.409.564.862	1.409.541.743	24,18

Tabela 5: Dados do insertion sort para k = 1000.

3.2.3 Para k = 10000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	139.844.938	139.844.938	0,32
	1kk	256.822.588	255.822.607	0,53
	10kk	381.775.717	371.775.731	0,81
Aleatório 2	100k	141.074.607	140.974.624	0,29
	1kk	254.894.237	253.894.250	0,51
	10kk	380.658.502	370.658.518	0,84
Ordenado	100k	99.999	0	0,00
	1kk	999.999	0	0,00
	10kk	9.999.999	0	0,03
Parcialmente Ordenado	100k	110.215	10.217	0,00
	1kk	1.010.331	10.332	0,01
	10kk	10.011.724	11.726	0,03
Invertido	100k	949.995.000	949.994.999	2,01
	1kk	9.949.995.000	9.949.994.786	21,03
	10kk	99.949.994.954	99.949.971.835	211,59

Tabela 6: Dados do insertion sort para k = 10000.

O insertion sort teve seus melhores desempenhos quando o vetor estava ordenado e quase ordenado, tendo sua quantidade de comparações proporcional ao tamanho do vetor e efetuando pouquíssimas trocas. Seu pior desempenho foi quando a lista estava invertida, onde a quantidade de trocas foi quase igual a quantidade de comparações.

4 Shell Sort Parcial

4.1 Lógica de implementação

O shell sort funciona como uma extensão do insertion sort, pois ele cria gaps para trazer os maiores elementos para o início do vetor e, consequentemente, os menores para o final do vetor. Assim, quando gap = 1, seu comportamento é equivalente ao insertion sort em um dos seus melhores casos: quando a lista está quase ordenada. A ideia para ordenar com o shell sort parcialmente consiste em analisar quantos elementos serão percorridos no vetor para cada gap. Se essa quantidade for menor que k, o algoritmo continua normalmente. Mas se for maior ou igual a k, os elementos na posição do i-ésimo gap e do k-ésimo gap são comparados e ocorre um swap deles caso o i-ésimo seja maior que k-ésimo, e a execução continua a partir da k-ésima posição.

4.2 Resultados

4.2.1 Para k = 100

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	1.579.135	653.387	0,00
	1kk	17.918.104	6.522.495	0,09
	10kk	199.523.943	64.428.909	1,09
Aleatório 2	100k	1.577.992	652.362	0,00
	1kk	17.932.071	6.536.941	0,07
	10kk	199.623.751	645.29.068	1,07
Ordenado	100k	967.146	0	0,01
	1kk	1.1804.265	0	0,04
	10kk	139.238.328	0	0,68
Parcialmente Ordenado	100k	967.852	718	0,00
	1kk	11.805.174	922	0,06
	10kk	139.239.514	1.202	0,71
Invertido	100k	1.143.216	229.376	0,00
	1kk	13.836.689	2.503.447	0,04
	10kk	15.6182.620	21.244.841	0,73

Tabela 7: Dados do shell sort para k = 100.

4.2.2 Para k = 1000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	2.346.437	1.420.689	0,01
	1kk	26.348.957	14.953.348	0,01
	10kk	287.022.719	151.927.685	1,42
Aleatório 2	100k	2.407.701	1.482.071	0,01
	1kk	26.587.882	15.192.752	0,12
	10kk	286792857	151.698.174	1,43
Ordenado	100k	967.146	0	0,00
	1kk	11.804.265	0	0,06
	10kk	139.238.328	0	0,73
Parcialmente Ordenado	100k	971.949	4.815	0,01
	1kk	11.811.156	6.904	0,04
	10kk	139.247.935	9.623	0,73
Invertido	100k	1.240.009	326.169	0,01
	1kk	14.821.986	3.488.744	0,07
	10kk	168.821.512	33.883.733	0,76

Tabela 8: Dados do shell sort para k = 1000.

4.2.3 Para k = 10000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	3.452.140	2.526.392	0,01
	1kk	42.272.005	30.876.396	0,17
	10kk	466.319.900	331.224.866	2,03
Aleatório 2	100k	3.454.404	2.528.774	0,01
	1kk	42.811.255	31.416.125	0,18
	10kk	467.128.803	332.034.120	2,01
Ordenado	100k	967.146	0	0,00
	1kk	11.804.265	0	0,06
	10kk	139.238.328	0	0,76
Parcialmente Ordenado	100k	993.904	26.770	0,01
	1kk	11.851.673	47.421	0,07
	10kk	139.308.366	70.054	0,76
Invertido	100k	1.406.054	492.214	0,01
	1kk	15.522.983	4.189.741	0,06
	10kk	181.370.065	46.432.286	0,81

Tabela 9: Dados do shell sort para k = 10000.

O shell sort, como esperado, teve seus melhores resultados com os vetores ordenados e quase ordenados. A ideia dos gaps fez com que o caso da lista invertida não prejudicasse a execução. Além disso, é possível observar nas tabelas que para entradas aleatórias ele realiza bem mais comparações e trocas comparados aos outros tipos de inputs.

5 Quick Sort Parcial

5.1 Lógica de implementação

O quick sort escolhe um elemento arbitrário como pivô. No contexto deste trabalho, como a ordenação deve ser decrescente, os elementos a esquerda do pivô devem ser maiores que ele, enquanto os elementos a direita devem ser menores do que ele. Após isso, o algoritmo é chamado recursivamente para cada uma das partições do vetor, assim organizando todos os números. Para ordená-los parcialmente, basta observar o tamanho da partição esquerda. Caso ela seja maior que K, podemos ignorar a partição da direita. Neste trabalho, implementamos o quick sort com o pivo sendo o elemento mais a esquerda.

5.2 Resultados

5.3 Para k = 100

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	220.806	120.396	0,01
	1kk	3.278.905	2.278.384	0,01
	10kk	24.087.407	24.087.407	0,11
Aleatório 2	100k	277.522	176.461	0,00
	1kk	1.437.531	437.297	0,00
	10kk	33.291.467	23.287.634	0,12
Ordenado	100k	705.082.703	99.999	8,59
	1kk	1.784.293.663	999.999	862,12
	10kk	-	-	-
Parcialmente Ordenado	100k	-	-	-
	1kk	-	-	-
	10kk	-	-	-
Invertido	100k	5.000.049.999	2.500.099.999	11,20
	1kk	-	-	-
	10kk	-	-	-

Tabela 10: Dados do quick sort para k = 100.

5.4 Para k = 1000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	231.778	126.216	0,00
	1kk	3.289.134	2.283.376	0,00
	10kk	24.096.783	14.092.094	0,15
Aleatório 2	100k	332.043	207.220	0,00
	1kk	1.465.949	451.807	0,00
	10kk	33.303.605	23.293.825	0,15
Ordenado	100k	5.000.049.999	99.999	11,15
	1kk	500.000.499.999	999.999	1270,23
	10kk	-	-	-
Parcialmente Ordenado	100k	-	-	-
	1kk	-	-	-
	10kk	-	-	-
Invertido	100k	-	-	-
	1kk	-	-	-
	10kk	-	-	-

Tabela 11: Dados do quick sort para k = 1000.

5.5 Para k = 10000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	443.048	239.224	0,00
	1kk	3.437.880	2.350.118	0,01
	10kk	24.670.453	14.377.602	0,11
Aleatório 2	100k	535.849	318.497	0,01
	1kk	1.609.262	527.726	0,01
	10kk	34.070.982	23.667.849	0,14
Ordenado	100k	5.000.049.999	99.999	8,70
	1kk	500.000.499.999	999.999	857.92
	10kk	-	-	-
Parcialmente Ordenado	100k	-	-	-
	1kk	-	-	-
	10kk	-	-	-
Invertido	100k	5.000.049.999	2.500.099.999	11,03
	1kk	-	-	-
	10kk	-	-	-

Tabela 12: Dados do quick sort para k = 10000.

O quick sort parcial tem ótimo desempenho quando a entrada é aleatória, pois as partições tendem a ser mais balanceadas. Contudo, em outros tipos de entrada, as partições possuem tamanhos muito discrepantes, o que torna o algoritmo bastante lento. Na entrada ordenada, por exemplo, a cada nível da recursão, só ocorria partição a direita, e na entrada invertida só ocorria partição a esquerda. Alguns valores nas tabelas estão em branco pois a partição estava tão desbalanceada que a pilha deu stackoverflow e não foi possível computar os dados de sua execução.

6 Heap Sort Parcial

6.1 Lógica de implementação

O heap sort traz um princípio semelhante ao selection sort, pegar o maior elemento e colocá-lo em uma das posições extremas do vetor, nesse caso, no final do vetor, repetindo este procedimento várias vezes. Outra diferença, é que no heap sort é utilizada a estrutura de dados heap, cuja busca do maior elemento é mais eficiente. Para ordenar o vetor, o heap é construído, sendo que os nó pais são sempre maiores ou iguais aos nós filhos. Assim, o primeiro elemento do vetor é trocado com o último, ou seja, o maior número é direcionado ao final da lista. Com os itens restantes, refaz-se o heap e, novamente coloca-se o maior elemento no final do vetor, repetindo este processo K vezes até ordenar o vetor parcialmente.

6.2 Resultados

6.2.1 Para k = 100

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	378.624	76.208	0,00
	1kk	3.736.089	745.363	0,01
	10kk	37.328.331	7.442.777	0.12
Aleatório 2	100k	378.165	76.055	0.00
	1kk	3.739.467	746.489	0,01
	10kk	37.326.606	7.442.202	0,12
Ordenado	100k	155.004	1.668	0,00
	1kk	1.505.991	1.997	0,01
	10kk	15.006.981	2.327	0,04
Parcialmente Ordenado	100k	155.055	1.685	0,00
	1kk	1.506.048	2.016	0,00
	10kk	15.007.059	2.353	0,03
Invertido	100k	455.028	101.676	0,00
	1kk	4.505.919	1.001.973	0,01
	10kk	45.007.143	10.002.381	0,12

Tabela 13: Dados do heap sort para k = 100.

6.2.2 Para k = 1000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	422.898	90.966	0,00
	1kk	3.789.279	763.093	0,01
	10kk	37.390.575	7.463.525	0,11
Aleatório 2	100k	422.565	90.855	0,01
	1kk	3.792.594	764.198	0,00
	10kk	37.388.919	7.462.973	0,12
Ordenado	100k	199.605	16.535	0,00
	1kk	1.559.778	19.926	0,00
	10kk	15.069.603	23.201	0,03
Parcialmente Ordenado	100k	199.656	16.552	0,00
	1kk	1.559.832	19.944	0,00
	10kk	15.069.681	23.227	0,03
Invertido	100k	500.568	116.856	0,01
	1kk	4.559.574	1.019.858	0,01
	10kk	45.071.727	10.023.909	0,12

Tabela 14: Dados do heap sort para k = 1000.

6.2.3 Para k = 10000

Tipo da entrada	Tam.	Comp.	Trocas	Tempo(s)
Aleatório 1	100k	864.129	238.043	0,01
	1kk	4.321.101	940.367	0,01
	10kk	38.013.213	7.671.071	0,14
Aleatório 2	100k	864.090	238.030	0,0
	1kk	4.324.230	941.410	0,0
	10kk	38.011.533	7.670.511	0,18
Ordenado	100k	646.896	165.632	0,00
	1kk	2.097.495	199.165	0,00
	10kk	15.699.411	233.137	0,04
Parcialmente Ordenado	100k	646.950	165.650	0,01
	1kk	2.097.552	199.184	0,01
	10kk	15.699.489	233.163	0,04
Invertido	100k	955.968	268.656	0,00
	1kk	5.096.199	1.198.733	0,01
	10kk	45.717.279	10.239.093	0,14

Tabela 15: Dados do heap sort para k = 10000.

O heap sort, assim como o insertion sort, desempenha melhor quando o vetor já está ordenado ou quase ordenado, pois nesse caso, o algoritmo faz menos movimentações para construir e reconstruir o heap. Uma vantagem do Heap Sort é que em seu pior caso, quando a lista está invertida, os resultados não são muito discrepantes em relação a lista aleatória. Assim, o heap sort é muito bom para lidar com diversos tipos de entrada.

7 Comparação entre os algoritmos

Os algoritmos serão comparados abaixo de acordo com o tipo da entrada.

7.1 Entrada aleatória

7.1.1 Tamanho 100.000

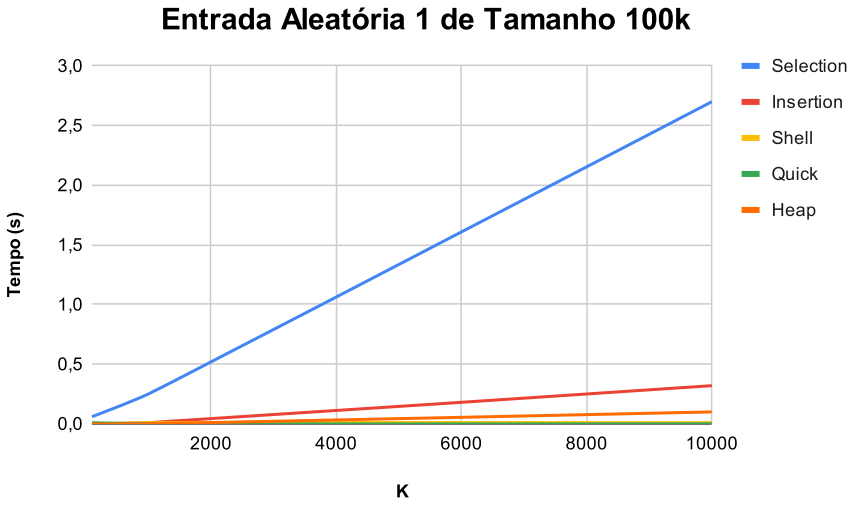

Figura 1: Comparação para entrada aleatória de tamanho 100k.

7.1.2 Tamanho 1.000.000

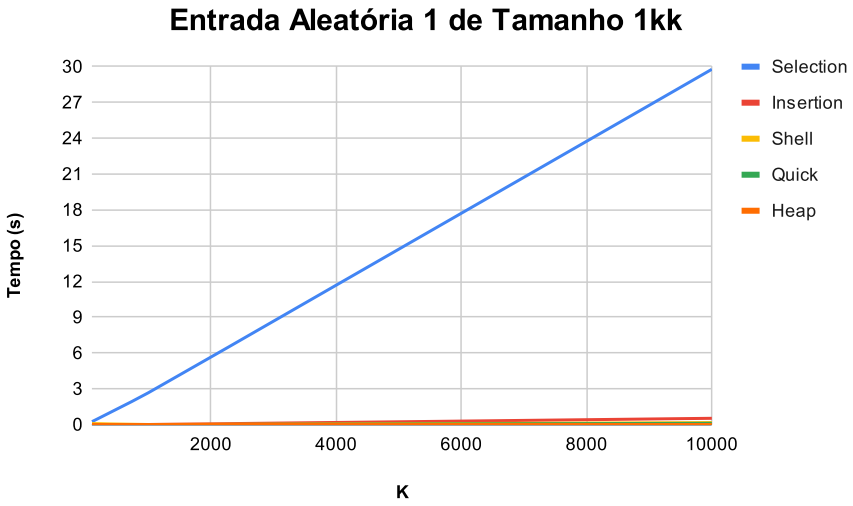

Figura 2: Comparação para entrada aleatória de tamanho 1kk.

7.1.3 Tamanho 10.000.000

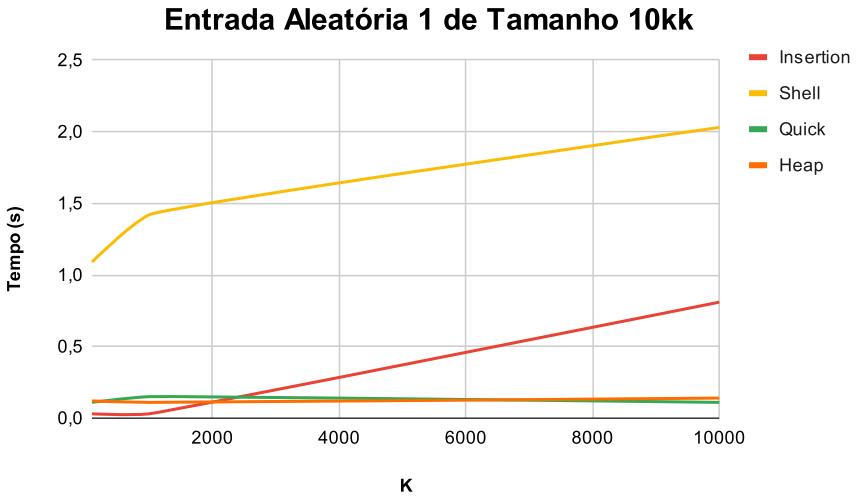

Figura 3: Comparação para entrada aleatória de tamanho 10kk.

OBS: Retirou-se o Selection Sort da comparação para melhor visualização dos demais algoritmos, uma vez que já foi inferido que o Selection apresenta desempenho inferior aos demais na entrada Aleatória.

Para entradas aleatórias, o Quicksort mostrou o melhor resultado, devido a suas partições tenderem a serem balanceadas neste caso. Outro algoritmo que também foi muito eficiente para essa entrada foi o Heap Sort.

7.2 Entrada ordenada

7.2.1 Tamanho 100.000

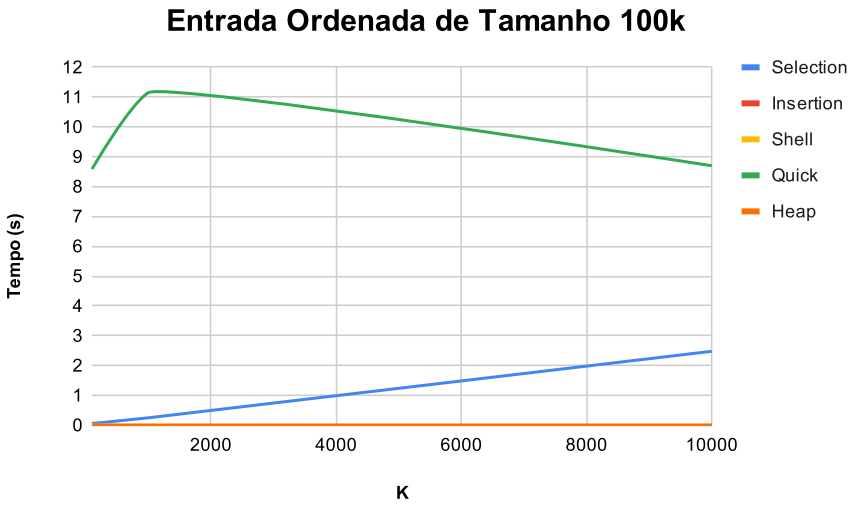

Figura 4: Comparação para entrada ordenada de tamanho 100k.

7.2.2 Tamanho 1.000.000

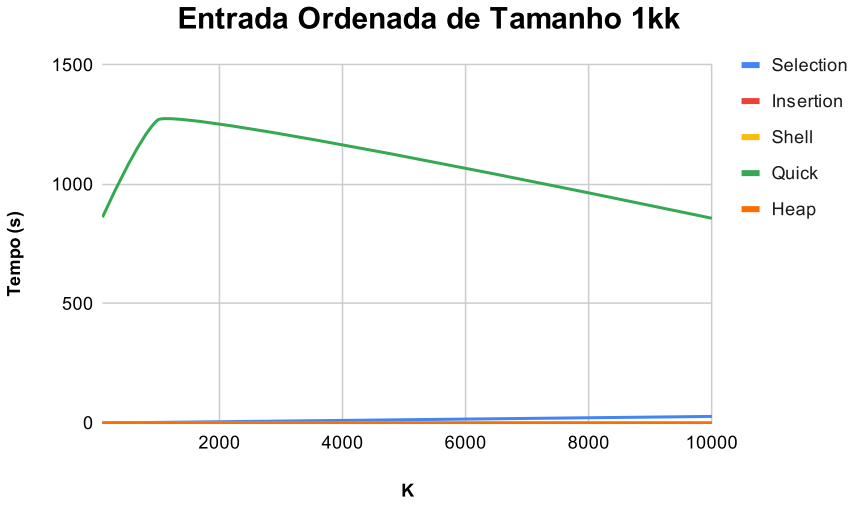

Figura 5: Comparação para entrada ordenada de tamanho 1kk.

7.2.3 Tamanho 10.000.000

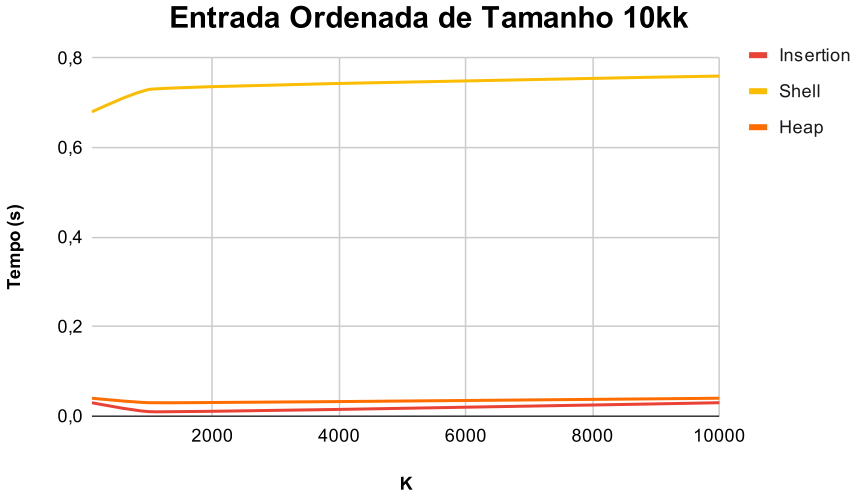

Figura 6: Comparação para entrada ordenada de tamanho 10kk.

OBS: Nesse gráfico, não há dados do Quicksort, pois não foi possível inferi-los. Além disso, optou-se por retirar o selection para melhor visualização dos demais algoritmos.

Para entradas Ordenadas, o Insertion sort apresentou o melhor resultado. O Heapsort também foi bastante satisfatório e o Shellsort foi satisfatório. Em contrapartida, o Quicksort e o Selection sort apresentaram grande tempo de CPU, especialmente o Quicksort que teve o pior desempenho para a entrada Ordenada. Isso ocorreu devido a implementação com o pivô sendo o elemento mais a esquerda da partição, fazendo que a partição ficasse desbalanceada.
À medida que o tamanho da entrada aumentava, era esperado que o Shell Sort desempenhasse melhor que o Insertion, pois é o que ocorre na ordenação total. Contudo, isso não aconteceu, pois a ordenação parcial ordena menos elementos, e por isso o Insertion sort permanceu como melhor algoritmo para a entrada Ordenada.

7.3 Entrada parcialmente ordenada

Nessa entrada, o Quicksort apresentava uma previsão de tempo muito elevada, por isso não foi possível medir o tempo de CPU desse algoritmo.

7.3.1 Tamanho 100.000

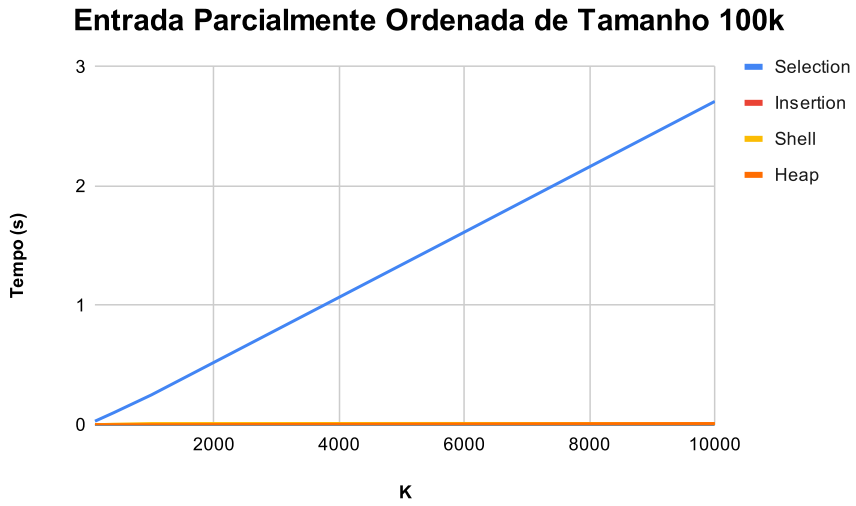

Figura 7: Comparação para entrada quase ordenada de tamanho 100k.

OBS: Nesse gráfico, não há dados do Quicksort, pois não foi possível computá-los.

7.3.2 Tamanho 1.000.000

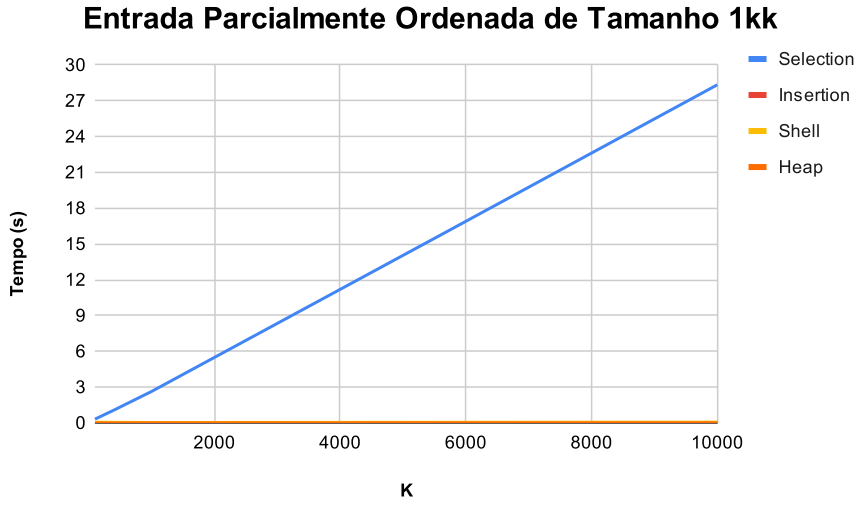

Figura 8: Comparação para entrada quase ordenada de tamanho 1kk.

OBS: Nesse gráfico, não há dados do Quicksort, pois não foi possível computá-los.

7.3.3 Tamanho 10.000.000

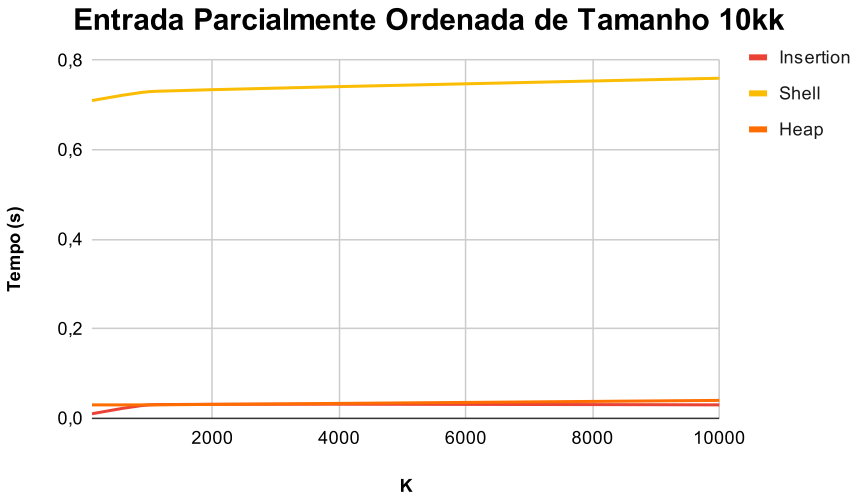

Figura 9: Comparação para entrada quase ordenada de tamanho 10kk.

OBS: Nesse gráfico, não há dados do Quicksort, pois não foi possível computá-los. Além disso, decidiu-se retirar o Selection da comparação para melhor análise dos demais algoritmos.

As entradas Parcialmente Ordenadas seguiram o mesmo padrão das entradas Ordenadas, isto é, o Quicksort teve o pior resultado seguido do Selection sort, enquanto o Insertion sort teve o melhor resultado, seguido do Heapsort e posteriormente do Shell sort.

7.4 Entrada invertida

Nessa entrada, o Quicksort apresentava uma previsão de tempo muito elevada, por isso não foi possível medir o tempo de CPU desse algoritmo para cada k.

7.4.1 Tamanho 100.000

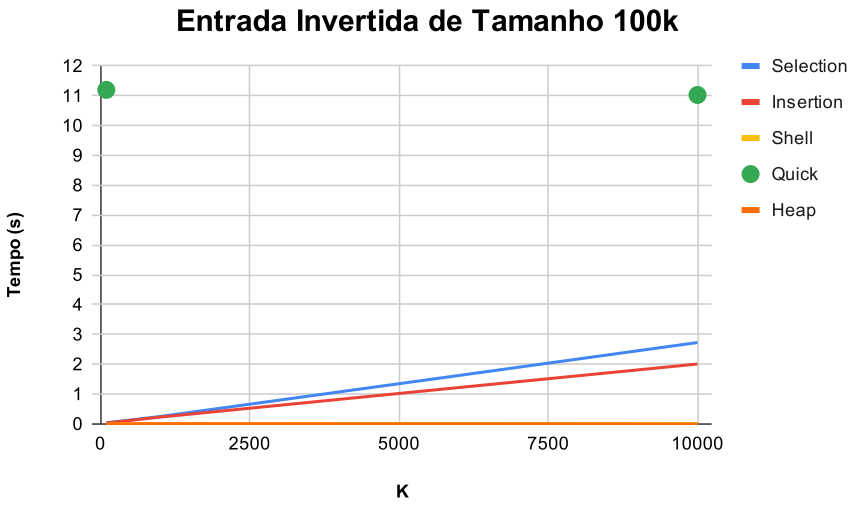

Figura 10: Comparação para entrada invertido de tamanho 100k.

OBS: Nesse gráfico, não foi possível medir o tempo do Quicksort para k = 1000, devido a isso apenas 2 pontos estão representados para esse algoritmo.

7.4.2 Tamanho 1.000.000

Figura 11: Comparação para entrada aleatória de tamanho 1kk.

OBS: Nesse gráfico, não há dados do Quicksort, pois não foi possível computá-los

7.4.3 Tamanho 10.000.000

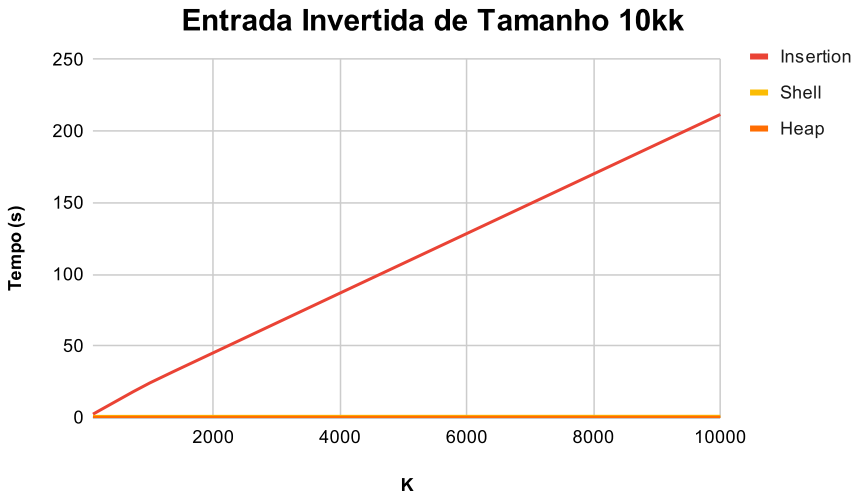

Figura 12: Comparação para entrada invertida de tamanho 10kk.

OBS: Nesse gráfico, não há dados do Quicksort, pois não foi possível computá-los. Além disso, decidiu-se retirar o Selection da comparação para melhor análise dos demais algoritmos.

Para o último tipo entrada, a entrada Invertida, o Heapsort e o Shell Sort apresentaram os melhores resultados, enquanto o Quicksort teve o pior resultado, seguido do Selection e do Insertion. Nessa entradas, o Shellsort obteve melhor performance que o Insertion, pois os gaps permitiram que o vetor ficasse quase ordenado mais rápido para o momento que o gap fosse 1

7.5 Conclusões das comparações

A partir da análise dos dados, infere-se que, no geral, o Selection Sort apresenta o pior resultado e que para entradas Aleatórias, o Quicksort é excelente, enquanto que para as outras entradas, seu resultado é bem inferior aos demais algoritmos. Percebe-se também que o Insertion, o Shellsort e o Heapsort costumam ter desempenhos mais constantes, sendo que o Insertion sort e Shell sort possuem melhor desempenho para as entradas Ordenadas/Parcialmente Ordenadas e Invertidas, respectivamente. Desses três algoritmos, Insertion, Heap sort e Shell sort, o Heapsort é o que teve menor variação, apresentando excelentes resultados para todas as entradas.

8 Conclusão

Neste trabalho consolidamos os conhecimentos ensinados em aula. Foi possível aprender a lógica de ordenação parcial do Selection sort, Insertion sort, Shell Sort, Quicksort e Heap Sort, bem como os pontos fortes e fracos de cada um deles. Assim, aprofundamos nosso conhecimentos tanto na ordenação total quanto na ordenação parcial.

9 Referências

9.1 Livros

- ZIVIANI, Nivio. **Projeto de Algoritmos com Implementações em Pascal e C**: 3ª edição revista e ampliada. 3. ed. Brasil: Cengage Learning, 2011.

9.2 Vídeos

- INSERT-SORT with Romanian folk dance. 2011.;
- SHELL-SORT with Hungarian (Székely) folk dance. 2011.PB.;
- HEAP Sort — GeeksforGeeks. 2017.;

9.3 Sites

- "Algorithms". GeeksforGeeks, https://www.geeksforgeeks.org/fundamentals-of-algorithms/. Acessado 3 de fevereiro de 2022.;
- "java - Min Heapify method- Min heap algorithm". Stack Overflow, https://stackoverflow.com/questions/15493056/min-heapify-method-min-heap-algorithm. . Acessado 4 de fevereiro de 2022.;