Relatório ordenação parcial

Estrutura de Dados II

O objetivo do trabalho em questão é fazer a ordenação parcial de um arquivo de dados inteiros formatado, usando-se de cinco tipos de ordenação diferentes: selection Sort, insertion Sort, shellSort, quickSort e heapSort. A fim de comparar suas vantagens e desvantagens frente aos mais variados tipos de entradas.

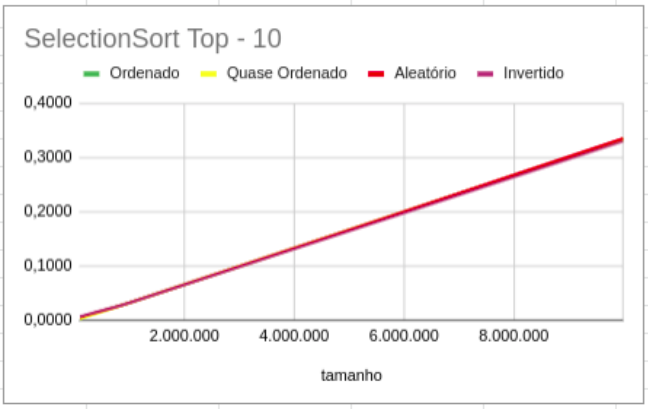
1- Selection Sort:

O selection Sort é um dos mais velozes na ordenação de vetores de tamanhos pequenos, o que não é comum na ordenação parcial, sendo um dos mais lentos para vetores de tamanhos grandes. Sua implementação foi feita de forma a ordenar os tops, com um for externo até os top e um for interno do tamanho da entrada, Complexidade teórica é linear, O(n). Além de ser ótimo em mover poucos os registros, principalmente nessa implementação.

Selection Sort - Top 10				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0,0060	0,0040	0,0070	0,0080
1.000.000	0,0330	0,0330	0,0330	0,0330
10.000.000	0,3340	0,3340	0,3350	0,3300

Selection Sort - Top 10 - Comparações				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	999.945	999.945	999.945	999.945
1.000.000	9.999.945	9.999.945	9.999.945	9.999.945
10.000.000	99.999.945	99.999.945	99.999.945	99.999.945

Selection Sort - top 10 - Trocas				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0	10	10	10
1.000.000	0	3	10	10
10.000.000	0	7	10	10



Conclusão:

Como o esperado, o algoritmo é lento para tamanhos grandes, aumentando o tempo proporcionalmente com a quantidade de dados, além de não fazer distinção entre os tipos de entrada, não ganhando nenhuma vantagem sobre eles, mesmo estando ordenada. Os resultados de número de trocas são iguais, para uma entrada de mesmo tamanho, e bem altos para todas elas. A complexidade dele é visivelmente linear, pois quando a quantidade de dados é multiplicada por 10 o tempo sofre a mesma alteração, como mostrado no gráfico. Destarte, seu único ganho visível é a pouquíssima mobilidade de registros, o que pode ser interessante dependendo da aplicação.

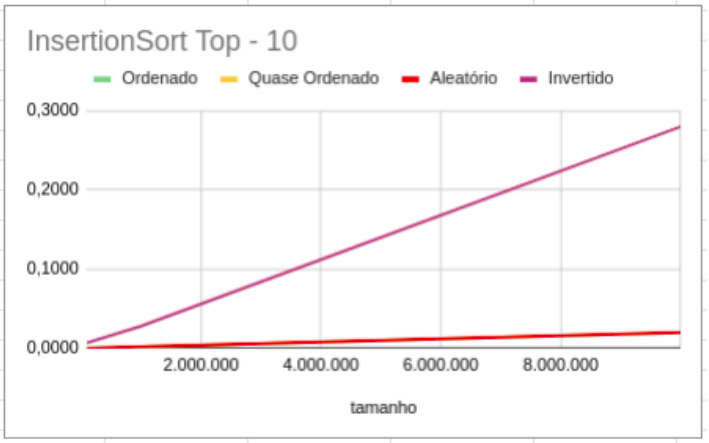
2- Insertion Sort:

O insertion parcial é um algoritmo que contém um for da quantidade de elementos da entrada, e um for mais interno que ordena o elemento entre a lista de tops, logo, inferimos que sua complexidade teórica é da ordem linear, de tops vezes o tamanho da entrada. Porém vemos nas referências abaixo, que no pior caso caso invertido, o algoritmo tem um comportamento mais custoso, o que faz total sentido, pois, nessa ordem, qualquer número será submetido a realizar o número máximo de comparações e trocas possíveis para o insertion parcial.

Insertion Sort - Top 10				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0,0000	0,0000	0,0000	0,0070
1.000.000	0,0020	0,0020	0,0020	0,0280
10.000.000	0,0200	0,0200	0,0200	0,2800

Insertion Sort - Top 10 - Comparações				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	100.035	100.053	100.989	999.945
1.000.000	1.000.035	1.000.053	1.001.178	9.999.945
10.000.000	10.000.035	10.000.062	10.001.178	99.999.945

Insertion Sort - top 10 - Trocas				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0	22	605	999.944
1.000.000	0	12	759	9.999.731
10.000.000	0	16	788	99.976.780



Conclusão: O insertion já se mostra mais vantajoso em comparação ao selection sort nos tipos ordenados, quase ordenados e aleatórios, em tempo de execução. Possui complexidade linear para as entradas, e seu pior caso, invertido, deve ser evitado devido ao tempo e comparações elevadas, além da troca de registros super alta. Todavia, as comparações gerais melhoram muito em relação ao último algoritmo citado, contudo a quantidade de trocas aumenta consideravelmente em contraste.

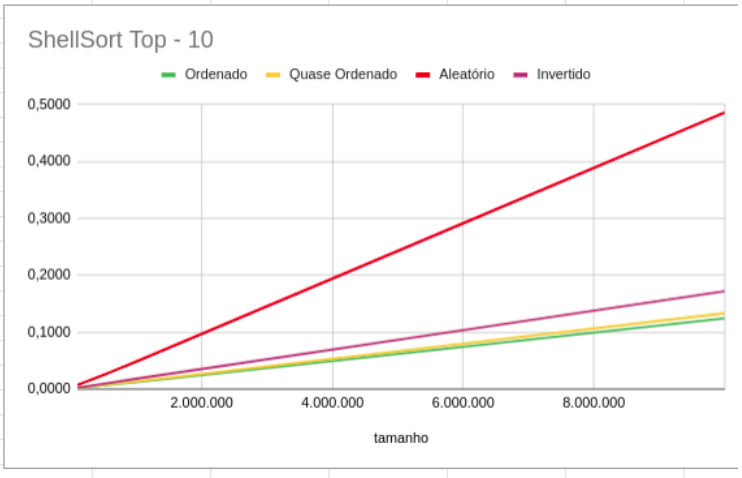
3- Shell Sort:

Implementado com a chave 3x+1, o shell é uma variação do insertion sort e tem sua vantagem principalmente no caso invertido que o método acima tem grandes dificuldades de resolver. A ideia da aplicação da parcialidade nesse algoritmo foi, aproveitar-se de sua característica de ordenar elementos a uma distância h, e a cada vez que o algoritmo faz isso ele cria sub vetores ordenados, assim, a cada passagem, podemos considerar os top * gap posições como o total a ser ordenado na próxima iteração, que na realidade são as top maiores posições de cada sub vetor gerado pela divisão do gap.

ShellSort - Top 10				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0,0020	0,0020	0,0070	0,0020
1.000.000	0,0120	0,0130	0,0480	0,0180
10.000.000	0,1240	0,1330	0,4860	0,1720

ShellSort - Top 10 - Comparações				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	314.706	314.958	654.495	453.623
1.000.000	3.132.795	3.133.118	6.425.265	4.466.557
10.000.000	31.195.668	31.196.057	63.364.265	43.114.262

ShellSort - top 10 - Trocas				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0	264	381.187	192.223
1.000.000	0	336	3.701.126	1.804.785
10.000.000	0	405	36.311.891	16.219.143



Conclusão: Esse algoritmo é difícil na obtenção de sua complexidade, mas podemos ver duas características marcantes nele. A primeira, é seu ganho no tipo de entrada invertida, que o insertion tem muita dificuldade em resolver, esse resultado se reflete no tempo, nas comparações e nas trocas bem mais baixas, logo mais vantajosas, sendo uma boa escolha nessa situação. E, a segunda, como o caso aleatório é nitidamente o pior caso deste algoritmo, o que fica ainda mais claro na representação gráfica dele. Por fim, seus demais casos não apresentam ganhos em relação à última opção apresentada, mais uma vez, fazendo a ressalva do caso invertido.

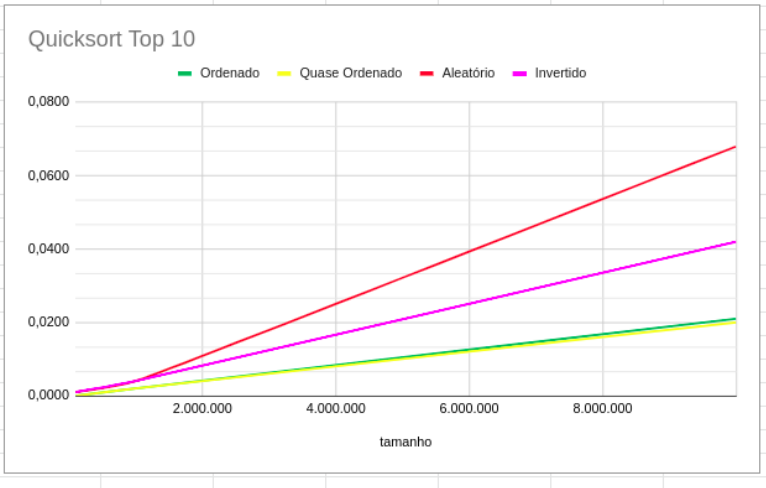
4- Quicksort:

Sem dúvidas, o algoritmo mais recomendado para os mais diversos tipos de ordenação genérica é o quick sort, e sua versão parcial foi adaptada de forma a ordenar a parte direita do vetor de cada chamada a ordenar, apenas se ela fizer parte das tops posições a ordenar. De acordo com o próprio livro do Ziviani : Projeto de Algoritmos, “A análise do Quicksort Parcial é difícil. O comportamento é muito sensível à escolha do pivô, podendo cair no melhor caso O(k log k), ou cair em algum valor entre o melhor caso e O(n log n)”.

QuickSort - Top 10				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0,0000	0,0000	0,0010	0,0010
1.000.000	0,0020	0,0020	0,0040	0,0040
10.000.000	0,0210	0,0200	0,0680	0,0420

QuickSort - Top 10 - Comparações				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	40	86	70.974	100.038
1.000.000	44	76	323.360	1.000.042
10.000.000	52	104	5.338.390	10.000.050

QuickSort - top 10 - Trocas				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0	22	35.468	50.000
1.000.000	0	17	161.665	500.000
10.000.000	0	28	2.669.180	5.000.000



Conclusão: Até o momento é o algoritmo mais balanceado e rápido dentre as opções apresentadas. Consegue tirar vantagem dos tipos de entrada, por exemplo, a ordenada e até mesmo a quase ordenada, sendo veloz e mantendo número baixíssimo de comparações e trocas. Na aleatória é um pouco menos ágil que o insertion e faz um número muito mais alto de trocas comparado a ele, o que pode ser uma desvantagem. E, na invertida, tem resultados mais promissores que o ShellSort. Sendo assim, sem dúvidas é o algoritmo mais indicado até o momento por conseguir tirar proveito de tipos de entradas no mínimo um pouco ordenadas, ser veloz e, principalmente, ter equilíbrio frente a qualquer situação, por mais que o aleatório, que é um caso comum, não tenha a melhor performance. Mas, vale lembrar que a escolha do pivô afeta diretamente esse algoritmo.

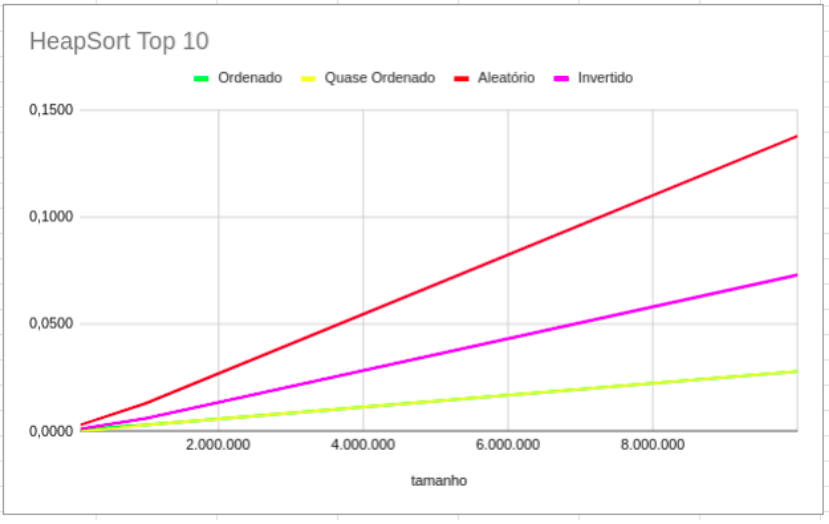
5- HeapSort:

Por último, o heapsort, que é baseado em seleção, é obtido parcialmente para selecionar os maiores números da seguinte forma: constrói-se o heap de forma a colocar na raiz o maior número da lista, depois este é colocado na última posição do vetor, e o heap é reconstruído. Repete esse procedimento até ordenar os tops maiores no fim do vetor. Assim, sua complexidade fica: O(n) para construir o heap, e top vezes O(log n) para reconstruí-lo de acordo com a lógica acima. Dessa forma, sua complexidade teórica é O(n + top log n);

HeapSort - Top 10				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	0,0010	0,0000	0,0030	0,0010
1.000.000	0,0030	0,0030	0,0130	0,0060
10.000.000	0,0280	0,0280	0,1380	0,0730

HeapSort - Top 10 - Comparações				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	150.425	150.479	282.575	300.397
1.000.000	1.500.512	1.500.566	2.821.333	3.000.474
10.000.000	15.000.602	15.000.683	28.223.168	30.000.577

HeapSort - top 10 - Trocas				
tamanho	Ordenado	Quase Ordenado	Aleatório	Invertido
100.000	152	171	74.725	100.142
1.000.000	181	200	743.579	1.000.167
10.000.000	211	239	7.440.692	10.000.203



Conclusão: Não apresenta vantagens em relação ao favorito do momento, o quicksort, pois a construção do Heap agrega um custo O(n) ao algoritmo e prejudica, principalmente, o desempenho das entradas menores. Além dele aumentar muito tanto a quantidade de comparações quanto de trocas devido a sua lógica. E por fim, faz trocas no caso de uma entrada já ordenada, o que nenhum outro algoritmo fez até então, sendo uma desvantagem única.

6- Conclusão:

O relatório e principalmente o código em questão podem ser muito úteis para tomada de decisões de projetos sobre qual algoritmo de ordenação parcial usar, já que diferentes aplicações requerem distintos olhares para analisar os dados referenciados. Porém, de forma macro, é possível inferir as características mais marcantes de cada algoritmo. Como por exemplo, o selection possuir grande vantagem em quando o importante é mover pouco os registros, ou selecionar pouco tops, como top 1,2 ou 3 em uma base de dados muito grande. O insertion ser o melhor entre os dois básicos da ordenação, selection e insertion, porém com a ressalva de que o caso invertido dele ainda é defasado. Que o shell, baseado em inserção, consegue resolver o caso invertido até então com mais facilidade que seu antecessor. E que o Heap acaba por não trazer vantagens maiores frente aos estudos dos anteriores e principalmente do quicksort, que mostra-se vantajoso, tanto na ordenação total, quanto na parcial devido a sua agilidade, redução na quantidade de comparações, de conseguir tirar proveito de entradas mais fáceis de ordenar e principalmente de se equilibrado nos mais diversos cenários.

Por fim, decidi fazer o teste com o top 10 pela motivação do trabalho, que foi o exemplo de busca na web e padronizei esse valor de forma a conseguir ter uma visão melhor ao comparar os resultados obtidos a cada algoritmo utilizado.