RELATÓRIO
EX 13 - ORDENAÇÃO FINAL
PROFESSOR: VICTOR HAZIN
ALUNO: ARTHUR LINS DA GAMA
PERÍODO: 2
DATA: 28/11/2021

1. COMPLEXIDADE DOS ALGORITMOS

INSERTION SORT

O insertion sort é basicamente pegar cada número “x” e comparar com todos na sua esquerda, assim que for encontrando números menores que ele, ele é colocado atrás desses números. Caso não haja nenhum elemento menor que ele, ele é colocado no início da lista. Além disso, pelo fato desse algoritmo funcionar comparando um número com outros à sua esquerda, o primeiro número da lista não é comparado com nenhum à esquerda. Por funcionar colocando os números em seus devidos lugares comparando com os à esquerda dele:

a. O(n)

O melhor caso é quando se lê um arquivo que já está ordenado em ordem crescente.

b. O(n^2)

O pior caso é quando se lê em ordem decrescente .

c. O(n^2)

O caso médio é quando se lê um que não está ordenado e os números estão de forma aleatória.

SELECTION SORT

O selection sort já funciona classificando o primeiro valor “x” da lista como um valor mínimo, assim comparando com os números à sua direita, e no momento em que encontrar um número menor que ele esse número vai ser classificado como o mínimo. Cada comparação começa a partir do primeiro número não classificado. (Para vermos a complexidade desse algoritmo basta olhar para o número de loops, que são 2). Por esse motivo:

a. O(n^2)

O melhor caso é quando se lê um aarquivo que já está ordenado em ordem crescente, já que o valor mínimo sempre estará no seu devido lugar.

b. O(n^2)

O pior caso é quando se lê um que não está ordenado e os números estão de forma aleatória.

c. O(n^2)

O caso médio é quando se lê em ordem decrescente .

OBS: Uma curiosidade é que por algum motivo não identificado a função selection sort no arquivo de 1000 palavras repete o número 240 e o 993 e exclui os números 998 e 278, até onde eu enxerguei. Então eu rodei ela com os números já ordenados e todos esses números citados estão em seus locais corretamente, porém o número 1 é substituído pelo número 999.

BUBBLE SORT

O bubble sort funciona comparando 2 elementos, o número “x” e o seu posterior, se o posterior for menor que ele, eles trocam de lugar, caso contrário eles se mantém, e isso é feito do primeiro até o último número, logicamente a lista não fica ordenada de primeira, esse processo é feito várias vezes até todos os elementos serem classificados de forma correta. Por esse motivo o bubble sort é o mais demorado entre os 3 vistos e testados nessa atividade.

a. O(n)

O melhor caso é quando se lê um arquivo que já está ordenado em ordem crescente. Obviamente pelo fato da função não precisar percorrer a lista toda várias vezes já que todos já estão na ordem correta.

b. O(n^2)

O pior caso é quando o arquivo se encontra em ordem aleatória, obrigando o bubble sort a encontrar cada erro de ordenação várias e várias vezes.

c. O(n^2)

O caso médio é quando o arquivo está em ordem decrescente.

OBS: Fiquei curioso porque o tempo do Bubble sort para ler o arquivo de 100000 números era enorme batendo mais de 43 segundos e decidi rodar ele lendo um arquivo de 100000 números já ordenado e foram apenas 18.8 segundos, uma diferença gritante. Além disso, eu tive a ideia de colocar milhares de números iguais em um arquivo e usar a função bubble sort nele e foi bem rápido, em 4 segundos já tinha sido finalizado o processo.

OBS: A partir do HeapSort os algoritmos começam a ficar extremamente rápidos quando usados para ordenar os arquivos disponibilizados para nós.

HEAPSORT

O HeapSort funciona com uma relação entre índices de matriz e árvore binária, e ele “faz o heap” basicamente transformando o melhor elemento na raiz ou pai e passando o elemento menor para o filho.

a. O(nlog n)

O melhor caso é quando se lê um arquivo que já está ordenado em ordem crescente.

b. O(nlog n)

O pior caso é quando o arquivo se encontra em ordem aleatória.

c. O(nlog n)

O caso médio é quando o arquivo está em ordem decrescente.

MERGESORT

O MergeSort divide o array em dois e faz o Sort em cada uma dessas metades, depois disso ele faz a junção dos arrays classificados, fazendo isso até todos estarem classificados.

a. O (n * log n)

O melhor caso é quando se lê um arquivo que já está ordenado em ordem crescente.

b. O (n * log n)

O pior caso é quando o arquivo se encontra em ordem aleatória.

c. O (n * log n)

O caso médio é quando o arquivo está em ordem decrescente.

QUICKSORT

O QuickSort funciona da seguinte forma, um elemento chamado de pivô é selecionado a partir de uma divisão de uma matriz em submatrizes, e a classificação dos números funcionam baseado nesse pivô, pois todos os elementos menores que ele se mantenham à esquerda dele e os maiores fiquem a direita dele. E esse procedimento acontece com todos os subarrays separados até cada um conter apenas um elemento.

a. O (n * log n)

O melhor caso é quando o arquivo se encontra em ordem aleatória

b. O (n^2)

O pior caso é quando se lê um arquivo que já está ordenado em ordem crescente

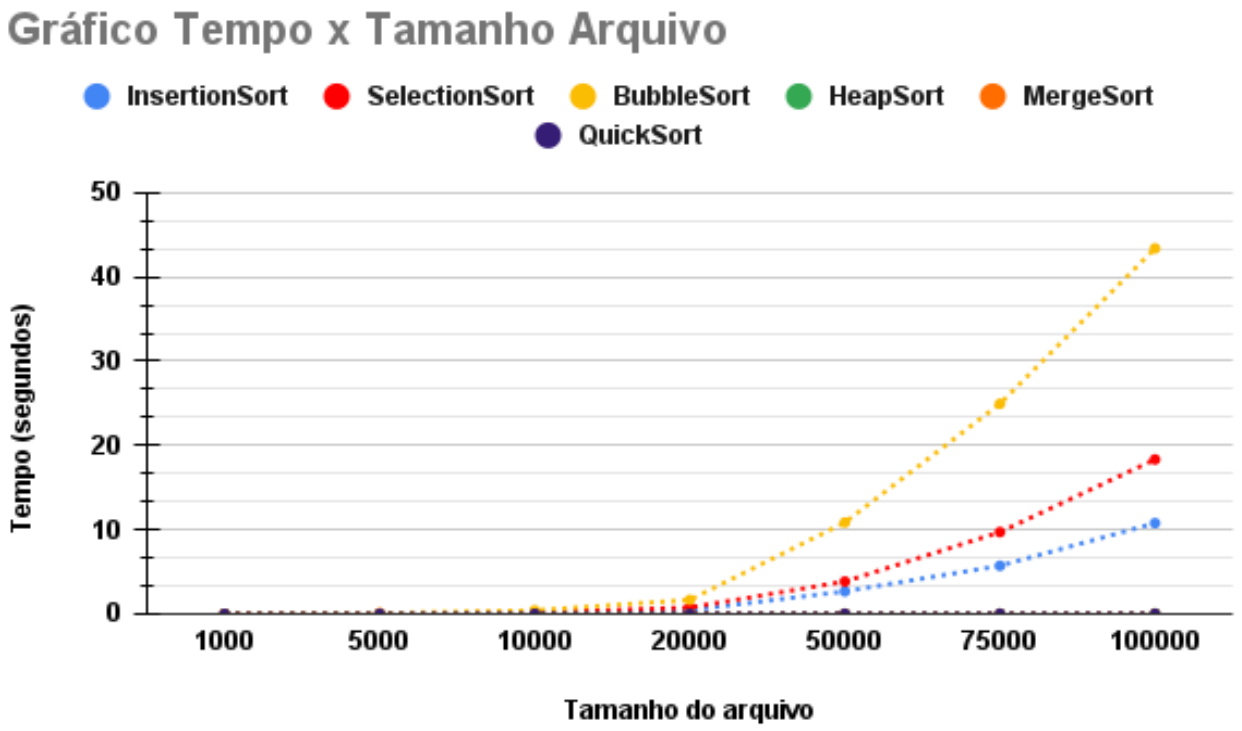
c. O (n * log n)

O caso médio é quando o arquivo está em ordem decrescente.

Tabela de tempo Algoritmo x Arquivo txt

	1000	**5000**	**10000**	**20000**	**50000**	**75000**	**100000**
Insertion Sort	0.000858	0.020712	0.092617	0.453396	2.669562	5.708915	10.770486
Selection Sort	0.001785	0.055415	0.179368	0.751731	3.841470	9.710299	18.330541
Bubble Sort	0.003299	0.078655	0.417076	1.661549	10.851360	24.934360	43.378658
HeapSort	0.000303	0.001272	0.003288	0.006623	0.019129	0.025968	0.034994
MergeSort	0.000203	0.001351	0.002361	0.004994	0.015318	0.019928	0.029613
QuickSort	0.000135	0.000813	0.001932	0.004074	0.012791	0.019043	0.019964

Gráfico



2. **COMENTÁRIOS SOBRE A TABELA E O GRÁFICO:**

Pela forma que cada algoritmo ordena os arquivos já era de se esperar uma boa diferença entre eles, podemos notar que o insertion sort é o mais rápido, o selection sort um pouco mais lento que ele e o bubble sort nos arquivos de até 20000 números tem uma pequena diferença para os demais, mas a partir de 50000 números podemos uma diferença gritante dele para esses 2.

Como comentado anteriormente o insertion sort compara um determinado número com todos a sua esquerda sendo mais eficiente por não precisar passar pelo arquivo todo mais de uma vez por isso se olharmos a tabela e o gráfico podemos ver que a linha azul que representa ele não sobe tanto e fica sempre abaixo dos demais algoritmos. Além de que a variação de tempo entre o arquivo de menor quantidade de números e o arquivo de maior quantidade é de apenas um pouco mais de 10 segundos.

No selection sort como são necessárias várias iterações para cada interação ter seu valor mínimo, ele é um pouco mais demorado que o insertion sort mas não tanto, começando a se diferenciar mais a partir do arquivo de 50000 números

Já o bubble sort é o que mais me deixou intrigado pelo fato de demorar bastante, no primeiro teste que eu fiz com esse algoritmo eu utilizei já de início o arquivo de 100000 e para ser sincero achei que não estava funcionando ou que tinha dado algum problema

no arquivo, mas eu esperei um pouco e ele funcionou perfeitamente, mas com 43.37 segundos para ser concluído. Desde os arquivos de menos números ele já se diferencia dos outros algoritmos, mesmo que sendo em milissegundos, mas a partir do 50000 é que conseguimos notar essa diferença com mais nitidez.

Já no heapsort, mergesort e quicksort, os tempos são muito pequenos nem chegando a aparecer a variação no gráfico nessa escala que usamos, nenhum deles passa de um segundo sequer. Comparado aos anteriores, esses 3 algoritmos praticamente não variam de tempo. Apenas olhando o código rodar sem ver a contagem de tempo, dá a impressão de que eles ordenam o arquivo de 1000 e de 100000 no mesmo tempo.

A função de contagem de tempo faz a gente perceber essa diferença de milissegundos entre os algoritmos, porém quanto mais números um arquivo possui mais é perceptível a diferença de tempo entre eles sem precisar olhar para o contador de tempo nos 3 primeiros algoritmos, já nos 3 últimos não.

Então eu decidi fazer uma tabela a parte com os algoritmos lendo os arquivos de 100000 números em forma crescente, decrescente e de modo aleatório para checar a diferença de tempo entre eles e a variação de tempo dos próprios algoritmos dependendo do modo em que o arquivo esteja, e uma tabela de cada algoritmo lendo um arquivo com milhares de números iguais. (Lembrando que são testes novos por isso os tempos serão um pouco diferentes da outra tabela)

Tabela de tempo Algoritmo x Arquivo 100000.txt

Algoritmo	Insertion Sort	Selection Sort	Bubble Sort	HeapSort	MergeSort	QuickSort
Crescente	0.000384	17.756465	22.253116	0.029985	0.016377	44.630911
Decrescente	23.172166	18.469560	33.598356	0.033122	0.019534	31.402329
Aleatório	12.160873	18.570248	45.712625	0.036866	0.033715	0.023451

Podemos notar que a variação no insertion e no bubble sort são praticamente as mesmas, e o bubble sort diminui praticamente a metade entre o melhor e o pior caso. Já o selection sort praticamente não varia. Além disso o insertion sort é o único que tem o pior caso sendo a ordem decrescente tendo um tempo maior que o bubble sort em ordem crescente. Assim como o selection sort, o heapsort e o mergesort não variam praticamente nada, menos ainda que o próprio selection sort.

O quicksort apesar de ser o mais rápido em ordenar em ordem aleatória, é o mais lento ordenando quando o arquivo já está ordenado em ordem crescente quase chegando ao bubble sort em ordem aleatória, e é o segundo mais lento quando já está ordenado em ordem decrescente. Sendo o que mais varia em comparação a todos os outros.

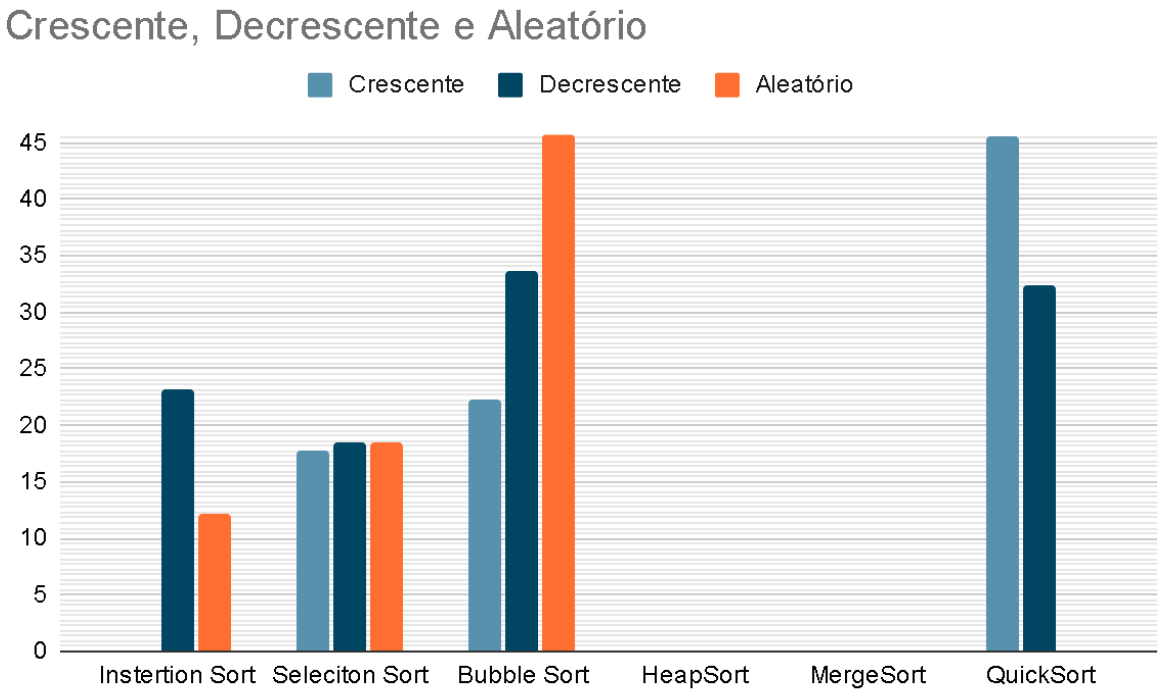
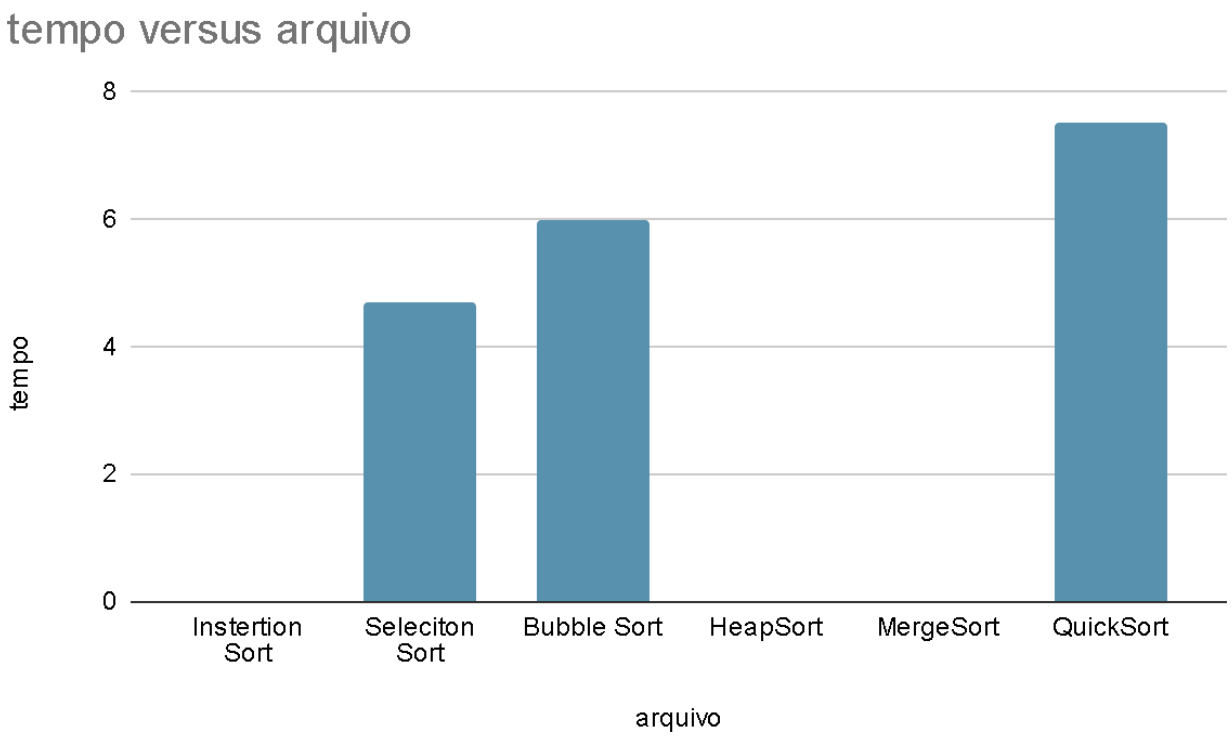


Tabela de tempo Algoritmo x Arquivo de números iguais

Algoritmo	Insertion Sort	Selection Sort	Bubble Sort	HeapSort	MergeSort	QuickSort
Tempo	0.000188	4.706669	5.977192	0.000798	0.005466	7.523829

Mesmo todos os números sendo iguais, o selection e o bubble sort ainda demoram um pouco pelo fato de terem várias interações em ambas, diferente do insertion sort.
Assim como o selection e o bubble sort o quicksort demora um pouco com arquivos de números iguais, já o heap e o merge sort continuam rápidos como em todas as situações testadas por mim.
(Os arquivos tinham a mesma quantidade de inteiros).


tempo versus arquivo
8
6
4
2
0
tempo
Instertion Sort
Seleciton Sort
Bubble Sort
HeapSort
MergeSort
QuickSort
arquivo