UFES - Universidade Federal do Espírito Santos

Trabalho 1

Ordenação Parcial em Memória Principal

Leonam Morais de Oliveira
Hiuri Carriço Liberato

Vitória, 2022
UFES - Universidade Federal do Espírito Santo

Objetivo:

O objetivo deste trabalho é desenvolver algoritmos em C que consigam ordenar vetores parcialmente, do maior valor ao menor, não importando sua ordenação primária. Para isso foram utilizados cinco métodos de ordenação, sendo eles: ordenação por seleção, inserção, shellsort, quicksort e heapsort. Além disso, o programa consegue imprimir na tela informações tais como: vetor em ordem crescente, x maiores elementos, estatísticas, dados/estatísticas separados por tab. A entrada dos vetores é por arquivos de extensão .txt e a saída é no próprio terminal.

Quick sort é um algoritmo de ordenação baseado em divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedem as chaves "maiores". Em seguida o quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. Em relação à complexidade temos que ela é $\theta(n^2)$.

O **heapSort** é um algoritmo que funciona através de uma árvore binária, a ideia é que haja uma ordem onde os nós filhos sempre sejam menores que os nós pais, ou o contrário conforme decisão de criação da árvore. cada vez que ao montar uma raíz detectamos que um filho é maior que o pai então colocamos o filho no lugar do pai o copiando e prosseguimos com a montagem da árvore.
O heapsort não é um algoritmo de ordenação estável, mas podemos contornar isso adaptando a estrutura a ser ordenada, cada elemento dessa estrutura adaptada deve ficar no formato de um par (elemento original, índice original). Assim, caso dois elementos sejam iguais, o desempate ocorrerá pelo índice na estrutura original.

Insertion Sort, ou ordenação por inserção, é um algoritmo de ordenação que, dado uma estrutura (array, lista) constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação. Tem complexidade de ordem O(nlog(n))

O **shellSort** utiliza a quebra sucessiva da sequência a ser ordenada e implementa a ordenação por inserção na sequência obtida. Devido a sua complexidade possui excelentes desempenhos em N muito grandes, inclusive sendo melhor que o Merge Sort.

Obs: Ordenação por inserção só troca itens adjacentes para determinar o ponto de inserção. São efetuadas n-1 comparações e movimentações quando o menor item

está na última posição. O método de Shell contorna este problema permitindo trocas de registros distantes.
A complexidade do algoritmo ainda não é conhecida por completo, mas em seu melhor caso ela é de ordem O(NlogN).

A ordenação por seleção ou **selection sort** é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os **n-1** elementos restantes, até os últimos dois elementos. A complexidade deste algoritmo será sempre **$O(n^2)$.**

Resultados:

Entrada: 1 milhão aleatório ordenando 100.000 elementos

```
[📦 com.visualstudio.code trab 1]$ ./a.out -a 100000 1m.txt
Algoritmo: Isertion
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 99999
Trocas: 2501190250
Tempo(s): 14.258828

Algoritmo: Selection
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 94999950000
Trocas: 100000
Tempo(s): 237.201007

Algoritmo: Shell
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 48182916
Trocas: 48182916
Tempo(s): 0.551224

Algoritmo: Heap
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 27097366
Trocas: 2500000
Tempo(s): 0.302020

Algoritmo: Quick
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 16496425
Trocas: 4899538
Tempo(s): 0.174184
```

obs: Não sabemos o motivo mas para entradas na casa dos milhões o nome do arquivo fica como (null), não deveria ter relação mas pelo visto tem.

Entrada: 1 milhão aleatório ordenando 1.000 elementos

```
[📦 com.visualstudio.code trab 1]$ ./a.out -a 1000 1m.txt
Algoritmo: Isertion
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 999
Trocas: 253495
Tempo(s): 0.001708

Algoritmo: Selection
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 999499500
Trocas: 1000
Tempo(s): 2.851642

Algoritmo: Shell
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 48009959
Trocas: 48009959
Tempo(s): 0.545649

Algoritmo: Heap
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 27097366
Trocas: 2500000
Tempo(s): 0.304530

Algoritmo: Quick
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 16496425
Trocas: 4899538
Tempo(s): 0.172516
```

Entrada: 1 milhão ordenado ordenando 100.000 elementos

```
[📦 com.visualstudio.code trab 1]$ ./a.out -a 100000 1mOrd.txt
Algoritmo: Isertion
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 99999
Trocas: 4999949980
Tempo(s): 28.228194

Algoritmo: Selection
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 94999950000
Trocas: 100000
Tempo(s): 252.371161

Algoritmo: Shell
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 5432643
Trocas: 5432643
Tempo(s): 0.118345

Algoritmo: Heap
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 25766222
Trocas: 2500000
Tempo(s): 0.180021

Algoritmo: Quick
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 16951629
Trocas: 1024447
Tempo(s): 0.050600
```

Entrada: 1 milhão ordenado ordenando 1.000 elementos

```
[📦 com.visualstudio.code trab 1]$ ./a.out -a 1000 1mOrd.txt
Algoritmo: Isertion
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 999
Trocas: 499500
Tempo(s): 0.002755

Algoritmo: Selection
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 999499500
Trocas: 1000
Tempo(s): 2.655683

Algoritmo: Shell
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 5346498
Trocas: 5346498
Tempo(s): 0.117211

Algoritmo: Heap
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 25766222
Trocas: 2500000
Tempo(s): 0.180335

Algoritmo: Quick
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 16951629
Trocas: 1024447
Tempo(s): 0.050532
```

Entrada: 1 milhão quase ordenado ordenando 100.000 elementos

```
[📦 com.visualstudio.code trab 1]$ ./a.out -a 100000 1mQord.txt
Algoritmo: Isertion
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 99999
Trocas: 4999849649
Tempo(s): 27.571002

Algoritmo: Selection
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 94999950000
Trocas: 100000
Tempo(s): 250.439892

Algoritmo: Shell
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 5526446
Trocas: 5526446
Tempo(s): 0.119520

Algoritmo: Heap
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 25773979
Trocas: 2500000
Tempo(s): 0.180453

Algoritmo: Quick
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 16952005
Trocas: 1025805
Tempo(s): 0.050448
```

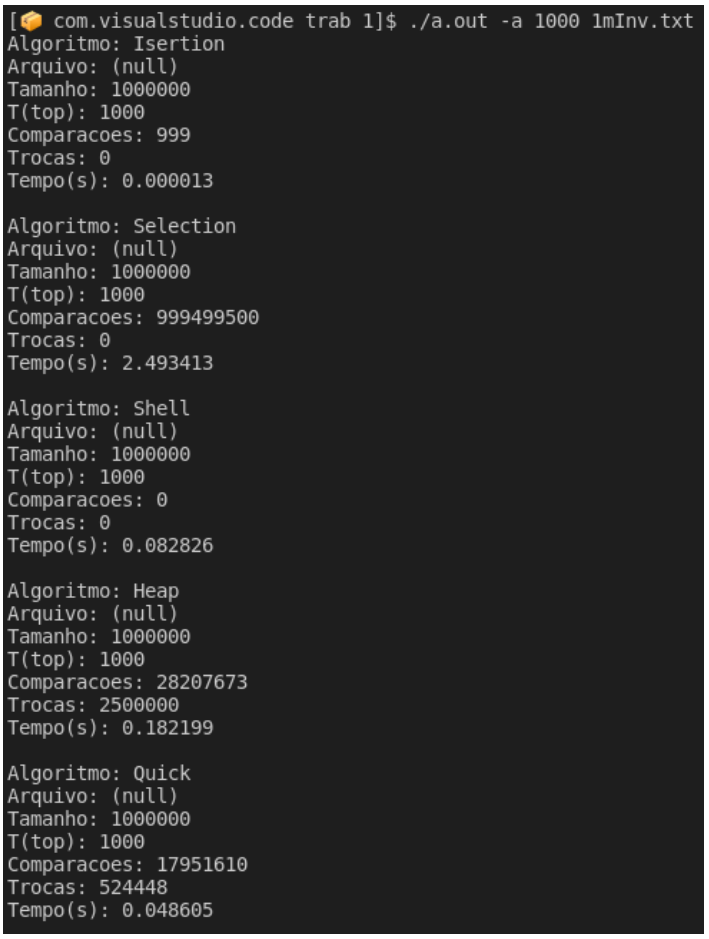
Entrada: 1 milhão quase ordenado ordenando 1.000 elementos

```
[📦 com.visualstudio.code trab 1]$ ./a.out -a 1000 1mQord.txt
Algoritmo: Isertion
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 999
Trocas: 498205
Tempo(s): 0.003333

Algoritmo: Selection
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 999499500
Trocas: 1000
Tempo(s): 2.933483

Algoritmo: Shell
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 5402737
Trocas: 5402737
Tempo(s): 0.117931

Algoritmo: Heap
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 25773979
Trocas: 2500000
Tempo(s): 0.180514

Algoritmo: Quick
Arquivo: (null)
Tamanho: 1000000
T(top): 1000
Comparacoes: 16952005
Trocas: 1025805
Tempo(s): 0.050490
```

Entrada: 1 milhão invertido ordenando 100.000 elementos

```
[📦 com.visualstudio.code trab 1]$ ./a.out -a 100000 1mInv.txt
Algoritmo: Isertion
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 99999
Trocas: 0
Tempo(s): 0.001175

Algoritmo: Selection
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 94999950000
Trocas: 0
Tempo(s): 233.561352

Algoritmo: Shell
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 0
Trocas: 0
Tempo(s): 0.083107

Algoritmo: Heap
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 28207673
Trocas: 2500000
Tempo(s): 0.180338

Algoritmo: Quick
Arquivo: (null)
Tamanho: 1000000
T(top): 100000
Comparacoes: 17951610
Trocas: 524448
Tempo(s): 0.057178
```

Entrada: 1 milhão invertido ordenando 1.000 elementos



Conclusão:

Pudemos perceber que os algoritmos selection sort, insertion sort e shellsort para entradas pequenas possuem bom desempenho, por exemplo para entradas de mil dados em forma aleatória. Passando desse valor, o tempo passa a ser um fator limitante para o funcionamento do algoritmo. Já no caso do selection sort, por exemplo, em entradas de até cem mil dados, ele é capaz de realizar a ordenação independente da organização do vetor usado ficando com um tempo inferior a 15s. Mas, quando o valor sobe para a casa dos milhões, é possível observar o pior caso e um tempo de execução inviável. No entanto, para os outros dois, insertion sort e shellsort, com entradas na ordem dos milhões, só é possível observar uma execução viável se os dados estiverem em uma ordenação prévia favorável, ou seja, parcialmente ou completamente ordenados dado as suas complexidades O(N) e O(Nlog(N)) respectivamente.

Em contrapartida, ficou claro que os algoritmos quicksort e heapsort possuem melhor desempenho. Enquanto os três algoritmos citados anteriormente gastam cerca de 10s em média para execução da ordenação, os dois permanecem com menos de 1s em média. Para entradas aleatórias, o quicksort sai na frente do heapsort no quesito tempo, independente do tamanho de entrada. Porém, quando há uma ordem pré-definida, não

importando entre normal, inversa ou quase ordenada, o quicksort se torna bem menos eficiente, tendo um tempo inviável, cedendo espaço para o heapsort como algoritmo de melhor eficiência. Vantagem essa graças ao nível de complexidade que permanece O(NlogN) em todas as possibilidades de execução, enquanto o quicksort assume $O(N^2)$ em seus piores casos.

Portanto, concluímos que para escolher o melhor algoritmo de ordenação, primeiro é preciso ter bom conhecimento do problema em que ele será inserido para que, assim, se obtenha os melhores resultados possíveis em relação às limitações de cada um e seus cenários de melhor desempenho. Mesmo nos casos de ordenações parciais. Quando o objetivo é apenas obter uma parte/quantidade específica de números do vetor, não se julga necessário ordená-lo por completo para obter o resultado, mas sim ordenar apenas o suficiente para obter os valores desejados. Dependendo também da quantidade de itens ou elementos a serem buscados, com quantidades muito grandes, alguns algoritmos serão melhores e mais indicados do que outros de acordo com sua complexidade.