Universidade Federal de Viçosa
Campus Florestal
CCF 211 - Algoritmos e Estruturas de Dados I

Trabalho Prático 3

João Francisco Hecksher - 3893
Pedro Augusto Maia Silva - 3878
Guilherme Melos - <matrícula>

Florestal 2019

Sumário

1 Introdução

O terceiro trabalho de Algoritmos e Estruturas de Dados foi proposto com objetivo de melhorar nossas noções quanto a utilização de algoritmos de ordenação, analisando as diferenças de performance entre algoritmos e observando até que ponto é mais vantajosa uma abordagem simples em comparação com uma complexa.

No caso em questão foram utilizados o código de sorting de Seleção e o código de QuickSort, para que com as respectivas medidas de comparações e movimentos estes pudessem ser comparados.

2 Desenvolvimento

Os TAD’s biblioteca, texto e palavra foram feitas em duas variedades, tendo listas encadeadas ou vetores como estrutura de dados, com operações básicas de inserção, inicialização, remoção, impressão e tamanho nas suas duas formas implementadas.

A geração pseudo-aleatória de palavras para formar textos ocorreu com base na função rand(), que com sua seed de geração inicializada corretamente (em nosso caso, usando a função time() da biblioteca time.h) gera novas combinações a cada execução do programa. Como no trabalho ocorriam TADs que continham outros TADs, decidiu-se que a geração destes deveria se dividir em diversas funções, que poderiam chamar umas as outras. A mais baśica construía e retornava palavras através de caracteres minúsculos do alfabeto na tabela ASCII, que correspondem a uma faixa de números inteiros (97-122). Esta função era usada por outra, que a chamava várias vezes, adicionando determinada quantidade de palavras a um texto e retornando-o. Por fim, uma função de criar bibliotecas chamava a função de criação de textos, assim criando e preenchendo uma biblioteca.

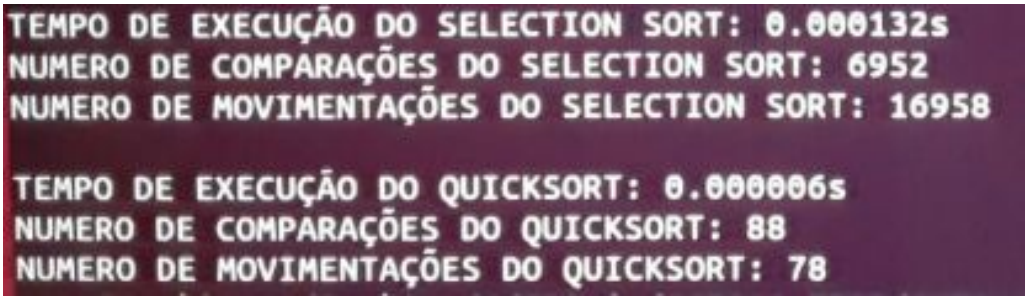
Após a implementação dos TAD's em seus dois tipos (vetor e lista encadeada) e a formação dos textos, restava a implementação das ordenações. Nessa etapa, a maior dificuldade foi a implementação correta do quicksort para bibliotecas encadeadas, devido a mistura de ponteiros, recursividade e diversas funções estando envolvidas num mesmo processo. Após um período de investigação e estudo do código, foram consertados os erros e daí em diante eram possíveis todas as ordenações requeridas.

Obtidos os textos e tendo eles ordenados, foi possível fazer a análise de tempo de execução do código através da biblioteca “time.h”, com a função clock(), que tem precisão de frações de segundo. Também foram analisados o número de comparações e movimentações, medidos a partir de um contador passado como ponteiro, que são acrescidos dentro da função a cada operação de movimento ou comparação.

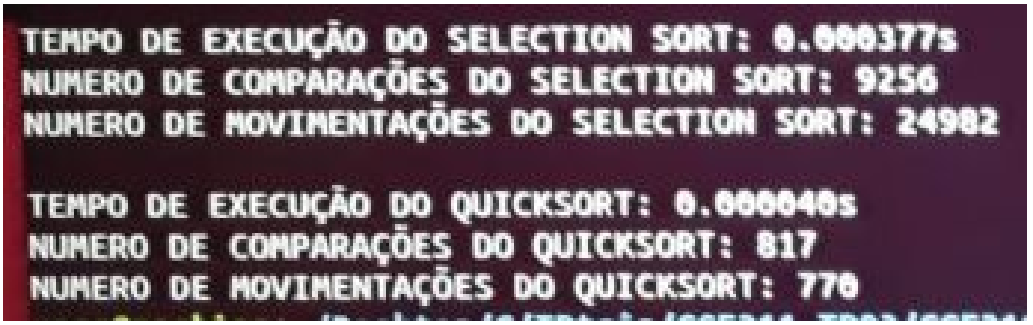
Por fim, uma interface foi criada de maneira a facilitar a criação e manipulação dos TADs, utilizando de diversas funções separadas que correspondiam aos respectivos “branches” requeridos nas instruções do trabalho. Essa abordagem se mostrou mais vantajosa do que a utilização de “switch & case”, já que está se tratava de uma interface mais complexa.

3 Resultados

Os casos de teste propostos nos mostra fortemente a grande diferença de performance entre algoritmos simples e algoritmos mais complexos e esse contraste ,cada vez mais, aumenta a partir do aumento do tamanho das estruturas analisadas como podemos observar nas imagens abaixo:



|----------Biblioteca com 100 textos de tamanho entre 1 e 100 palavras---------|

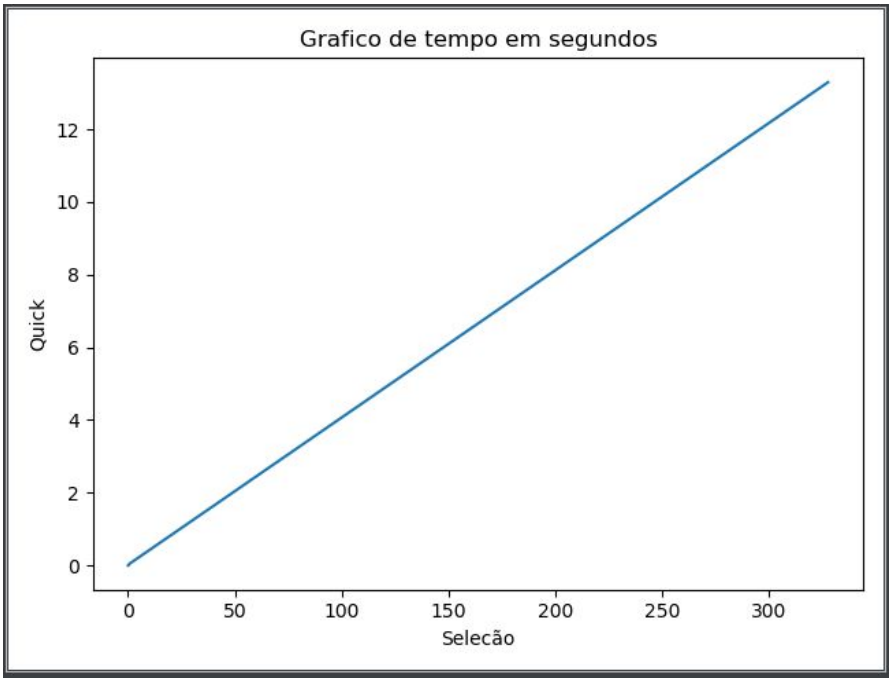


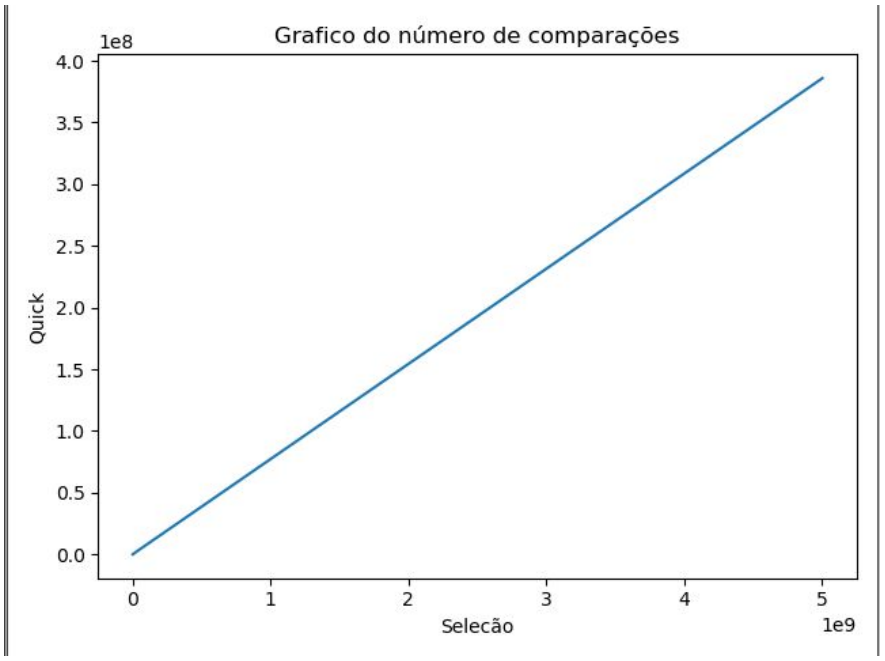
|----Biblioteca com 100 textos de tamanho entre 50.000 e 100.000 palavras----|

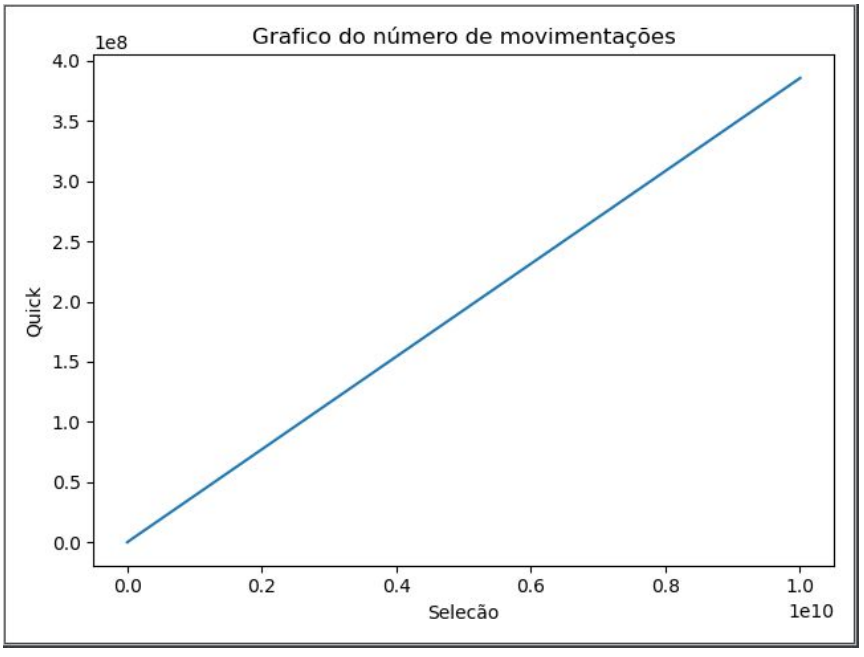
```
TEMPO DE EXECUÇÃO DO SELECTION SORT: 736.105396s
NUMERO DE COMPARAÇÕES DO SELECTION SORT: 5005770250
NUMERO DE MOVIMENTAÇÕES DO SELECTION SORT: 10020306738

TEMPO DE EXECUÇÃO DO QUICKSORT: 11.251516s
NUMERO DE COMPARAÇÕES DO QUICKSORT: 240302637
NUMERO DE MOVIMENTAÇÕES DO QUICKSORT: 240246170
```

|-----Biblioteca com 100.000 textos de tamanho entre 1 e 100 palavras------|

Para mostrar os contrastes de forma mais nítida foram feitos 3 gráficos comparando esses valores na ordenação de textos entre o código de Seleção e Quicksort que se encontram abaixo:

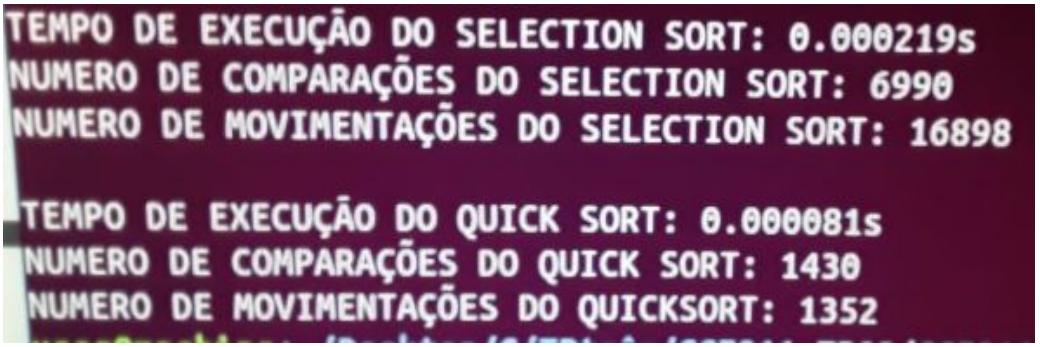




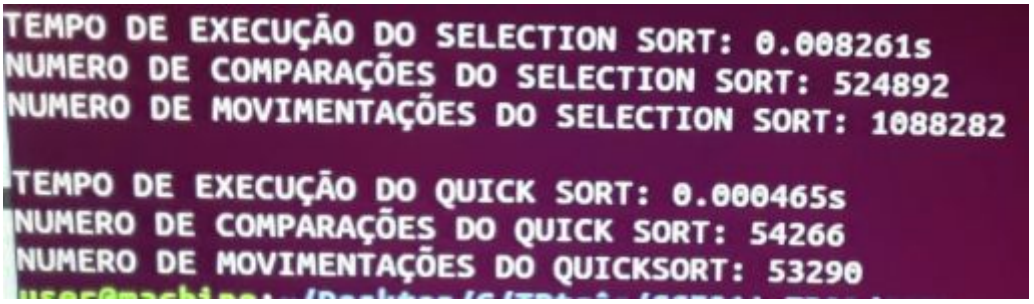


Os gráficos (Quick x Seleção) compara os valores tempo, comparações e movimentações de textos de tamanho 10,100,1.000,10.000,100.000 palavras (vale ressaltar que “le(X)” equivale a dizer que o número no eixo em questão é multiplicado por 10 elevado a X).

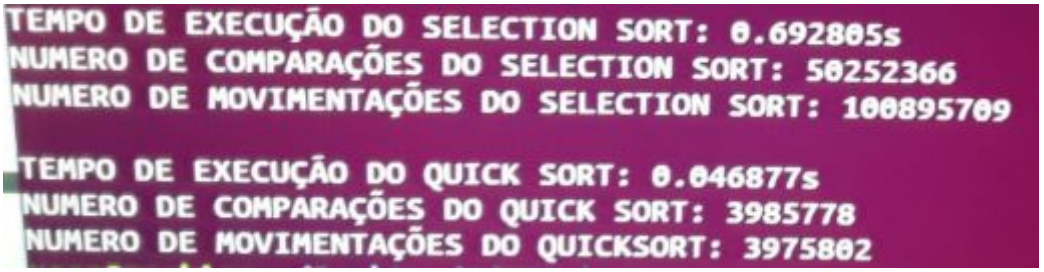
Aqui se encontram os resultados dos testes com lista encadeada dos valores tempo, comparações e movimentações de textos de tamanho 10,100,1.000,10.000,100.000 palavras:



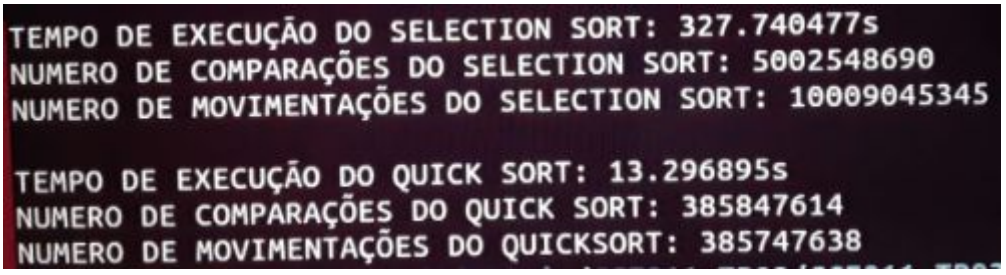
|----------Texto com 100 palavras---------|



|----------Texto com 1.000 palavras---------|

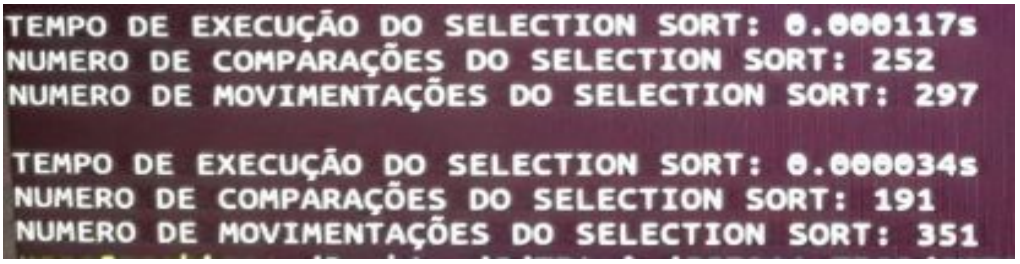


|----------Texto com 10.000 palavras---------|

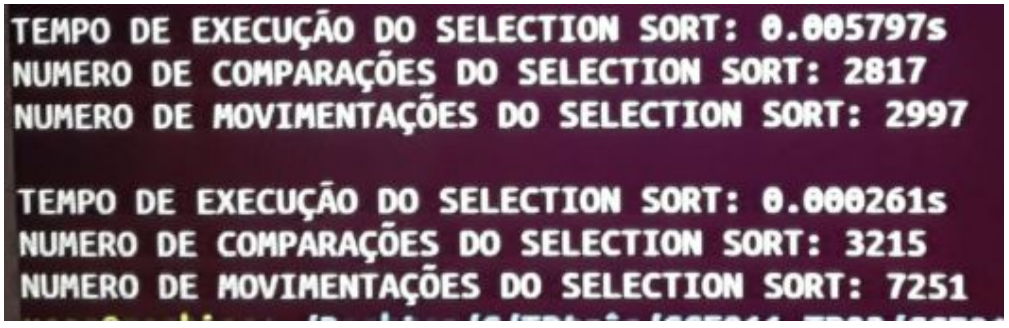


|----------Texto com 100.000 palavras---------|

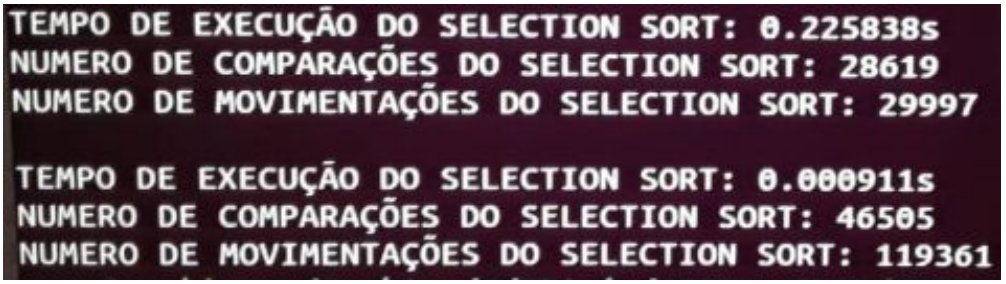
Aqui se encontram os resultados dos testes com vetor dos valores tempo, comparação e movimentação de textos com tamanho 10,100,1.000,10.000,100.000 palavras:



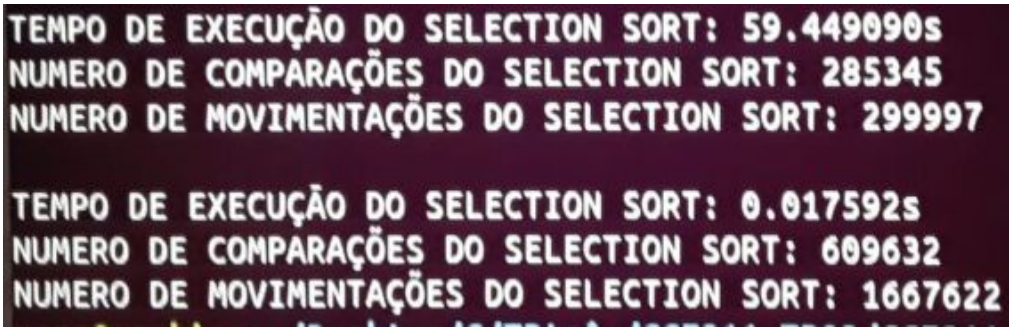
|----------Texto com 100 palavras---------|



|----------Texto com 1.000 palavras---------|



|----------Texto com 10.000 palavras---------|



|----------Texto com 100.000 palavras---------|

Dessa forma , não sobra dúvidas de que o algoritmo de ordenação Quicksort é cada vez mais indicado em relação ao algoritmo de seleção a partir do aumento das estruturas.

Os resultados que obtemos com o Quicksort nos mostram que este gasta menos tempo de execução, menos comparações e menos movimentações, ou seja , menos custo computacional.

4 Conclusão

Até que ponto podemos utilizar algoritmos de ordenação simples?

Durante a realização desse trabalho, praticamos o que foi dito em aula, comprovando que para conjuntos de dados considerados pequenos, algoritmos de ordenação simples acabam podendo ser mais indicados que os complexos. Vale ressaltar que os algoritmos levados em consideração neste trabalho foram os de seleção (simples) e quicksort (complexo), e que para quantidades pequenas de dados, o processo mais compartimentado, que requer mais etapas de um algoritmo complexo acaba por não valer a pena. No entanto, como pudemos observar, mesmo para conjuntos “médios”, a eficiência de um quicksort já se destaca.

Para vetores maiores que 100, o número de comparações necessárias para a utilização de algoritmos de ordenação simples ,neste caso , o custo inicial para fazer o particionamento do vetor seria pago , pois ele reduziria consideravelmente o número de comparações e assim se torna mais eficiente em relação aos mais simples e agora entregando mais velocidade e menor custo para ordenação neste caso.