Trabalho Prático 2

Biblioteca Digital de Arendelle

Victor Gabriel Ferreira Moraes

2018046734

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

victor.moraes@ufmg.com

1. Introdução

Neste trabalho será abordado à respeito das práticas e conceitos de ordenação, assim como as técnicas e habilidades de análise comparativa de algoritmos, mais especificamente, relativo às variações de implementação do algoritmo de ordenação Quicksort, levando em conta aspectos como: número de comparações de chaves, o número de movimentações de registros, e o tempo de execução de cada variação do referente método de ordenação.

Sendo assim, ao final deste trabalho, espera-se poder tirar conclusões acerca das características (vantagens e desvantagens) de cada variação do Quicksort, por meio de uma análise performática em diferentes casos, para que dessa forma, seja possível compreender a melhor variação a ser utilizada em uma dada situação específica.

Posto isso, torna-se necessário entender, sob um aspecto geral, o funcionamento do algoritmo e suas variações (clássico, mediana de três, primeiro elemento, inserção 1;5;10%, não recursivo).

Dessa forma, por definição, o Quicksort em si, é um método de ordenação não-estável (desenvolvido por Charles Antony Richard Hoare em 1960), que se baseia no conceito de dividir para conquistar a fim de ordenar um conjunto de dados. Esse conceito, é aplicado por meio da estratégia de separar as chaves de menor valor do lado esquerda, e as chaves de maior valor ao lado direito (esse critério é estabelecido com base na escolha de um elemento “parâmetro”, denominado pivô), criando assim subpartições de um conjunto, e repetindo esse processo de maneira recursiva para os demais subconjuntos criados, até que se tenha por fim um vetor ordenado. Por ser considerado um algoritmo não-estável, isso significa que, no Quicksort, a ordem de registros de chaves iguais não é necessariamente preservada.

Em relação à complexidade de tempo do algoritmo, o Quicksort possui como melhor caso uma complexidade da ordem de O(nlogn), que ocorre quando cada partição é dividida em tamanhos iguais. Já o pior caso, corresponde a uma complexidade de $O(n^2)$, e ocorre quando o pivô é escolhido como um dos elementos extremos do conjunto de dados já ordenado.

Em relação às características das variações do Quicksort, tem-se o:

- **Quicksort clássico:** Essa versão consiste em realizar a ordenação de maneira recursiva, escolhendo o pivô como o elemento central do arranjo. É

considerado a forma comum de implementação do algoritmo. Nessa variação o algoritmo pode variar entre o melhor e o pior caso.

- **Quicksort mediana de três:** Neste caso, a ordenação também se dá de maneira recursiva, porém o pivô é escolhido com base no critério da mediana de três dos elementos da partição. Posto isso, evita-se a ocorrência do pior caso nesse tipo de variação.
- **Quicksort primeiro elemento:** Este tipo de variação, também recursiva, escolhe o pivô sempre como o primeiro elemento da partição, logo, por consequência, essa implementação está sujeita ao pior caso do Quicksort sempre que o vetor já estiver pré ordenado.
- **Quicksort inserção 1%:** Assim como as já citadas, essa variação é implementada de maneira recursiva e utiliza o pivô analogamente a variação da mediana de três, porém após dividir o vetor em partições menores que 1% de seu tamanho original, passa-se a utilizar o método de inserção para ordenar essas partições. É uma das versões mais robustas, visto que evita o pior caso ao utilizar o critério da mediana de três, e utiliza do método de inserção para ordenar vetores pequenos, o que é uma das vantagens desse tipo de algoritmo de ordenação
- **Quicksort inserção 5%:** Versão análoga a inserção 1%, porém para partições menores que 5%.
- **Quicksort inserção 10%:** Versão análoga a inserção 1%, porém para partições menores que 10%.
- **Quicksort não recursivo:** Por fim, a versão do Quicksort não recursivo, é uma versão que não se utiliza recursão, mas sim uma pilha como estrutura de dados responsável por armazenar os intervalos de partições, substituindo assim a tarefa demandada pela ausência da pilha de recursão. Observação: neste trabalho o algoritmo não recursivo foi implementado com o pivô sendo o último elemento do conjunto de dados.

2. Implementação

O programa deste trabalho foi desenvolvido na linguagem C++ 11, utilizando como compilador o G++ da GNU Compiler Collection.

2.1 Classes

A implementação do programa foi dividido em apenas duas classes: main.cpp e pilha.cpp (junto de seu template pilha.hpp), de modo que toda a lógica e fluxo do programa se concentra-se na classe main.cpp, sendo a classe pilha.cpp a estrutura de dados implementada para servir de auxílio na implementação da versão do Quicksort não recursivo.

A pilha foi implementada contendo dois atributos privados, sendo eles topo (apontador responsável por indicar a posição do topo da pilha) e pilha[MAX] (vetor responsável por alocar os elementos da pilha, onde MAX é uma constante que vai até 500000, indicando o tamanho máximo da pilha). Além dos atributos, a estrutura criada possui um construtor (Pilha()) e mais quatro métodos: push (responsável por empilhar o novo elemento no topo da pilha); pop (responsável por remover o elemento do topo da

pilha); pilhaVazia (responsável por indicar se a fila está vazia ou não), e getTopo (responsável por retornar a posição do topo da pilha).

Como citado, a classe main.cpp conteve todos os fluxos do programa, sendo encarregada de estabelecer as relações entre as funções e módulos do código.

2.2 Funções e Módulos

O código foi estruturado basicamente em funções, em maior parte, responsáveis por executar a ordenação das variações dos Quicksort já citados. No geral, cada variação recursiva possuía três funções, compondo um módulo que representava cada algoritmo de um dado Quicksort, sendo elas: quicksortNomeVariação (responsável por definir as variáveis de retorno e chamar a execução da função de ordenação do Quicksort); ordenaQuicksortNomeVariação (responsável por realizar a chamada da função de partição e estabelecer as chamadas recursivas); particaoQuicksortNomeVariação (responsável por realizar a partição do vetor em subvetores, com base na ideia de dividir para conquistar).

Outras funções utilizadas na implementação dos métodos de ordenação, foram: quicksortNaoRecursivo e particaoQuicksortNaoRecursivo (responsáveis por implementar o algoritmo sem recursão do Quicksort, utilizando a estrutura de dados pilha.cpp); a função ordenaInsercao (responsável por implementar o método de ordenação de inserção, utilizado nos Quicksorts inserção 1%, 5% e 10%); e a função trocaElementos (responsável por realizar as trocas entre dois elementos durante o processo de ordenação).

Os demais métodos foram responsável por formular as entradas e saídas do programa.

2.3 Entrada e Saída de dados

Em relação ao tratamento da entrada e da saída de dados do código desenvolvido, foram criadas funções para modularizar esses processos, de modo que a etapa de recebimento e emissão dos dados, foi dividida nas seguintes funções:

- **main:** Responsável por inicializar as variáveis respectivas aos parâmetros de entrada, chamar o método executaQuicksort, e por fim, chamar o método imprimeResultado. Os dados recebidos seguem o seguinte formato: <variacaoQuicksort> <tipoVetor> <tamanhoVetor> [-p] (flag responsável por indicar a impressão dos vetores utilizados).
- **executaQuicksort:** Função encarregada de gerar o vetor de dados, baseado nos parâmetros de entrada, por meio da chamada do método inicializaVetor. Além de selecionar e executar a variação de Quicksort em questão, por meio da função selecionaQuicksort.
- **imprimeResultado:** Por fim, essa função é a última chamada do programa, e é responsável por formatar e imprimir os resultados gerados após a execução do algoritmo de ordenação, no seguinte formato: <variacaoQuicksort> <tipoVetor> <tamanhoVetor> <numeroComparacoes> <numeroMovimentacoes> <tempoExecucao> [vetores utilizados] (caso a flag tenha sido informada).

3. Instruções de compilação e execução

O programa desenvolvido, foi executado no Sistema Operacional Ubuntu 18.04.2, rodando com tamanho de pilha alterado para unlimited, para evitar problemas de stack overflow, devido a enorme carga de dados gerada durante a ordenação dos vetores. A compilação se deu por meio da utilização do makefile (disponibilizado na pasta \src do pacote .zip entregue). Dessa forma, seguem as etapas exercidas para compilar e executar o programa:

1. Executar o makefile por meio do comando “make”, estando no diretório do arquivo.
2. Definir pilha de programa linux como unlimited por meio da execução do seguinte comando: “ulimit -s unlimited”
3. Executar o programa por meio do seguinte comando “./tp2 <variacaoQuicksort> <tipoVetor> <tamanhoVetor> [-p]”, estando no diretório do arquivo.

Exemplo das etapas de execução:

```
victor@victor-pc:~/Desktop/TPQuicksort/Victor_Moraes/src$ make
g++  -g -Wall -std=c++11 -O3  -c -o main.o main.cpp
g++ -g -Wall -std=c++11 -O3 -c main.cpp Pilha.cpp
g++ -g -Wall -std=c++11 -O3  main.o  Pilha.o -o tp2
victor@victor-pc:~/Desktop/TPQuicksort/Victor_Moraes/src$ ulimit -s unlimited
victor@victor-pc:~/Desktop/TPQuicksort/Victor_Moraes/src$ ./tp2 QC Ale 10 -p
QC Ale 10 37 12 1
7 0 7 0 4 7 5 9 1 9
9 7 2 9 7 9 3 3 1 3
0 6 2 9 4 1 1 3 7 2
0 2 3 4 3 7 5 4 6 3
0 1 8 6 2 3 4 8 5 5
1 1 0 2 2 3 1 0 3 1
1 7 1 8 4 6 0 5 4 6
1 8 7 9 0 5 0 2 7 2
1 2 4 5 5 8 0 6 7 2
2 8 4 1 2 8 0 7 4 6
2 2 4 9 1 1 8 4 5 5
2 9 7 7 6 1 8 6 3 3
3 9 7 1 2 0 7 5 9 1
3 9 4 5 5 7 3 2 0 2
3 0 1 0 4 2 8 8 2 3
3 7 3 9 9 1 8 0 0 1
4 3 5 4 9 5 0 4 9 0
4 1 5 3 0 1 1 1 4 1
4 9 9 8 3 1 4 2 0 4
5 6 6 8 8 0 9 4 7 1
```

4. Análise experimental

4.1 Especificações do computador usado para os testes

O computador utilizado para os testes deste trabalho possui as seguintes especificações: processador Intel Core i5-4460 @ 3.20 GHz; 8,00GB de RAM; 1TB de HD; placa de vídeo NVIDIA GeForce GTX 970; Sistema Operacional Ubuntu 18.04.2.

4.2 Processo de análise

O processo de análise deste trabalho ocorreu por meio de uma metodologia prática de análise de dados, seguindo as etapas de geração, organização, comparação e interpretação dos dados.

Na primeira etapa, de geração dos dados, foi utilizado o software desenvolvido para obter o resultado de número de comparações, número de trocas e tempo de execução, para cada combinação dos algoritmos Quicksort: versão; tipo do vetor; e tamanho do vetor (7x10x3), dessa forma, totalizando 210 combinações de teste. Tendo isso em vista, cada caso de teste foi rodado 20 vezes, para se obter uma estimativa mais precisa com base na média do número de comparações e trocas, e na mediana de tempo, a fim de evitar resultados enviesados.

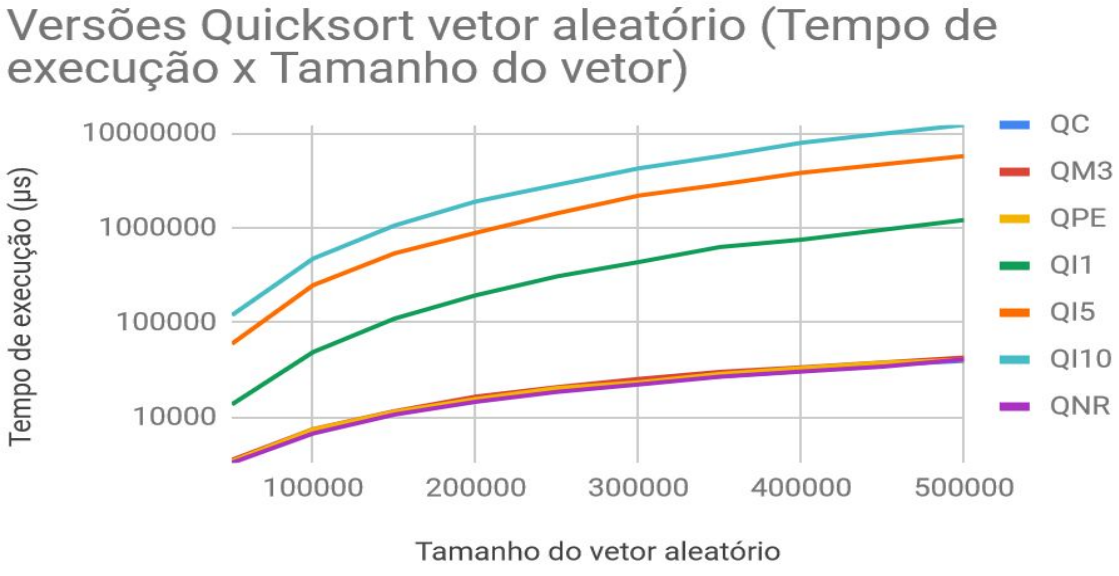
Após gerar os dados, os mesmos foram coletados e organizados em tabelas, de maneira ordenada, visando uma melhor visualização dos dados.

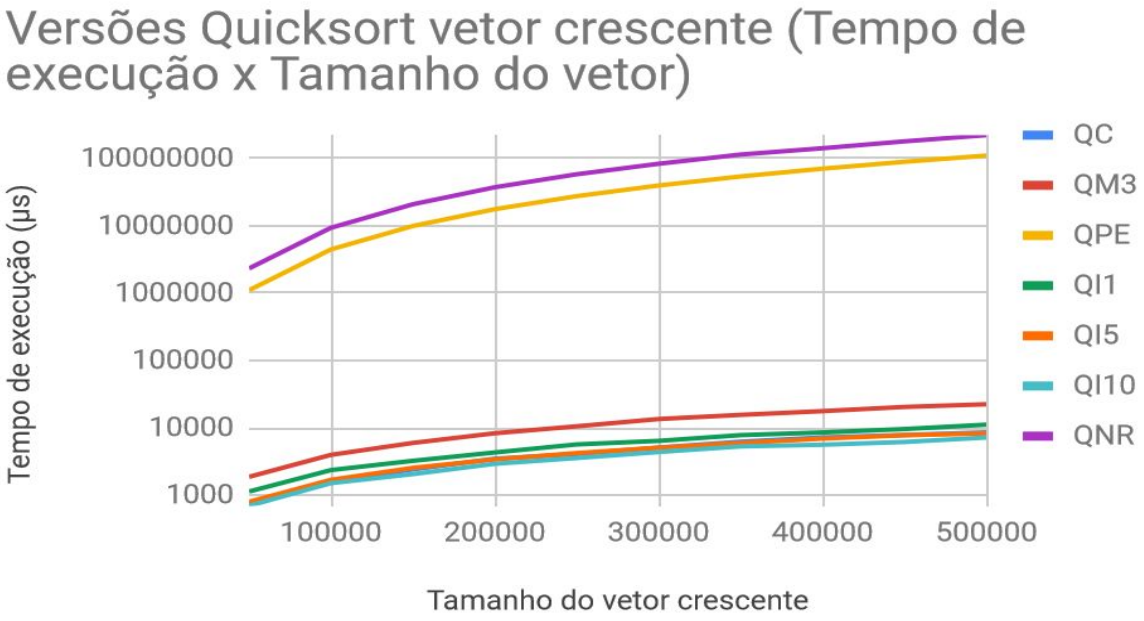
Em seguida, realizou-se uma comparação dos dados, levando em conta diferentes combinações, a fim de selecionar recortes relevantes para a análise, e logo após, esses dados selecionados foram plotados em diferentes tipos de gráfico para interpretação (https://docs.google.com/spreadsheets/d/1yJ6XnaFt3t1okkpBc0nPcybGXfIlk8-5imbqkvBsipk/edit?usp=sharing).

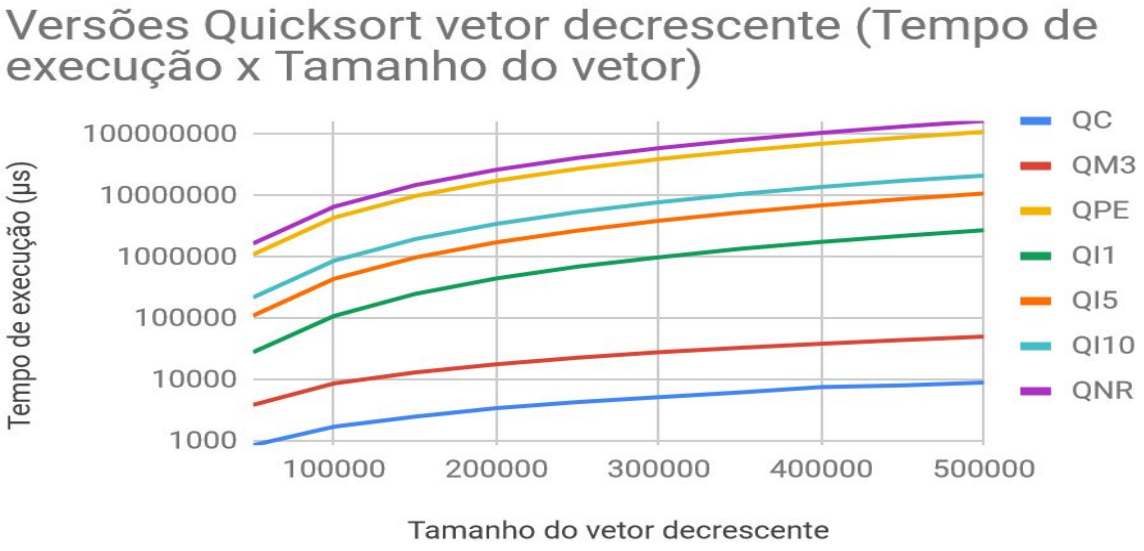
Ao final da interpretação dos gráficos, foi possível tirar algumas conclusões da análise crítica estabelecida em relação às diferentes versões do Quicksort, conforme mostrado a seguir:

4.2.1 Análise do tempo de execução

Os gráficos abaixo, indicam as diferentes versões do Quicksort, representadas de acordo com o tempo de execução conforme o tamanho do vetor, dessa forma, para cada tipo de vetor (aleatório, crescente e decrescente) temos um gráfico.







A análise crítica que pode ser feita em cima desses gráficos, é a de que é possível observar que o tempo de execução cresce de maneira diretamente proporcional ao tamanho do vetor, logo quanto maior o tamanho do conjunto de dados, maior será o tempo gasto para o ordenar esse conjunto.

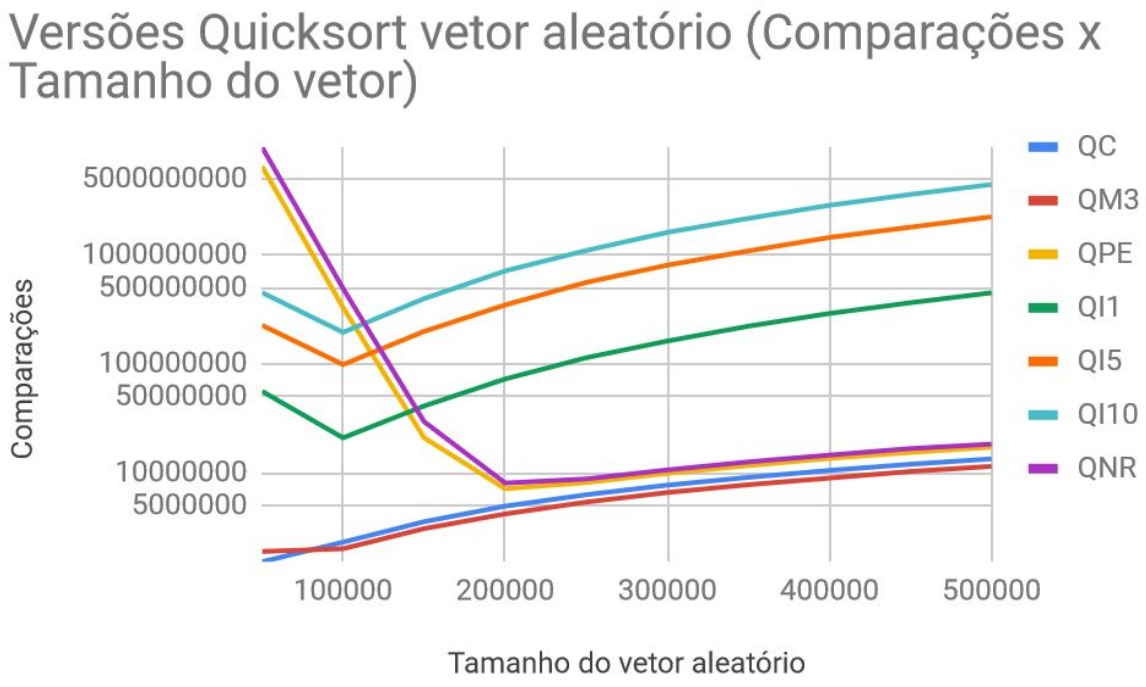
Outro comportamento observável é que no caso do arranjo aleatório, os métodos QI10, QI5, QI1, são os que demandam mais tempo respectivamente, enquanto os outros algoritmos possuem uma duração semelhante entre si. Isso ocorre pois, o algoritmo de Inserção é em geral da ordem $O(n^2)$ para conjuntos de dados grandes e não ordenados, o que ocorre frequentemente nos conjuntos aleatórios, já o algoritmo de Quicksort costuma ser da ordem de O(nlogn).

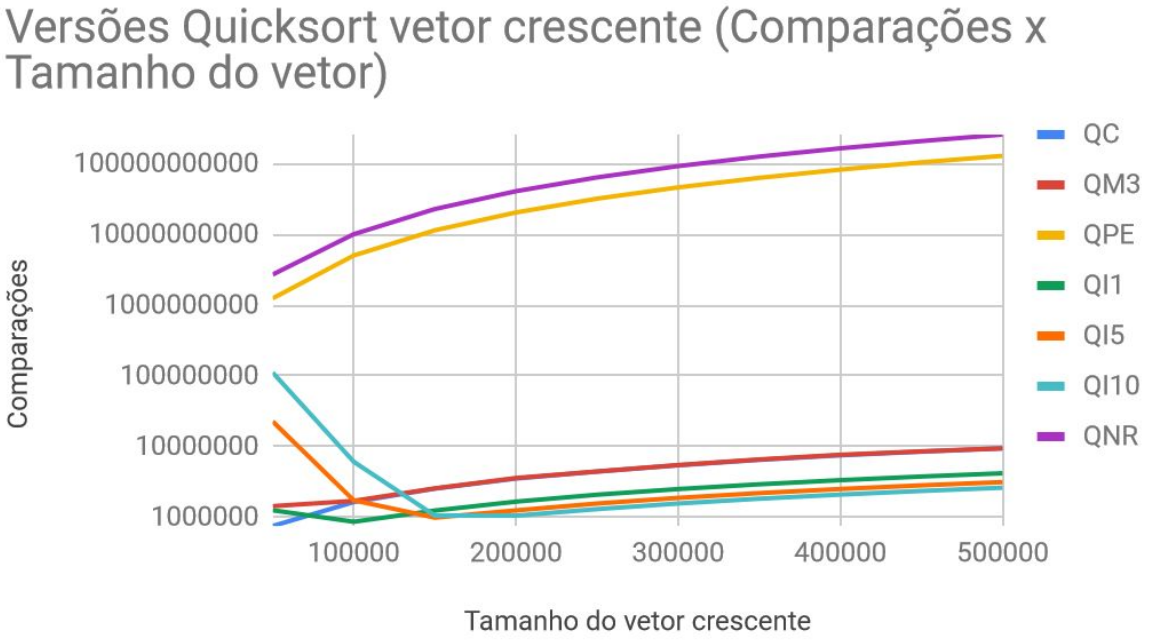
Por fim, pode-se observar também que, nos casos em que o vetor está ordenado ou de forma decrescente, as versões QPE e QNR, possuem maior tempo de execução, visto

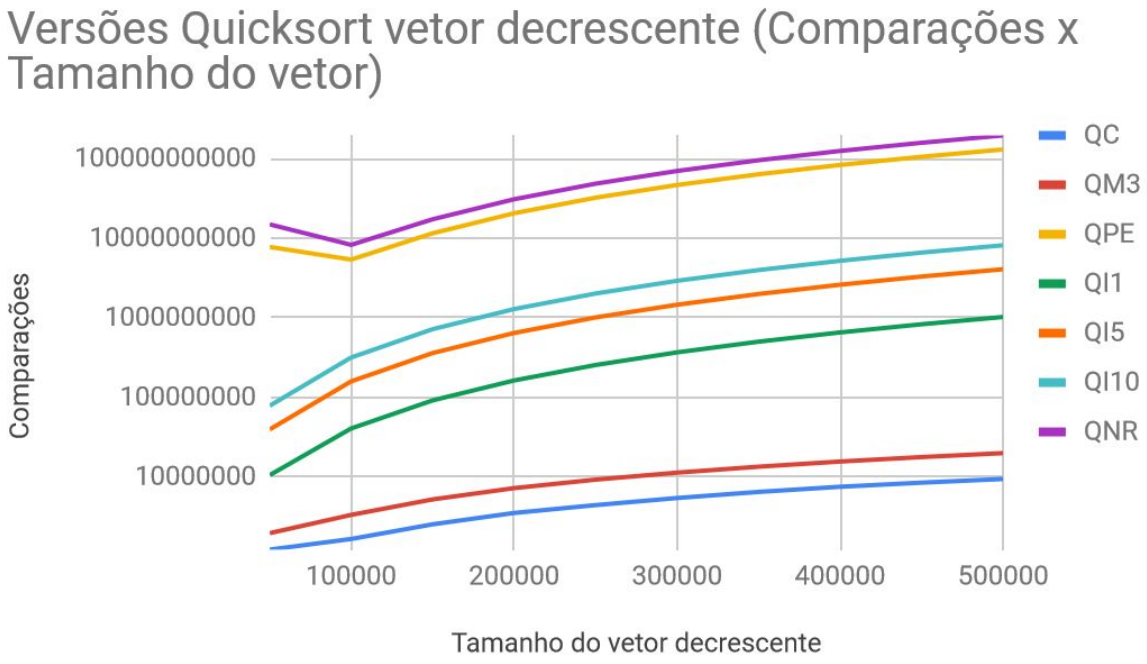
que por tomarem o pivô das partições como os extremos (primeiro elemento para o QPE e último elemento para o QNR), tem-se um dos piores casos do Quicksort, sendo o pior entre eles, o de quando o vetor já está ordenado. O tempo extra que o QNR gasta em relação ao QPE, é justificado devido ao uso da pilha como substituição da recursão, o que demanda maior tempo de processamento.

4.2.2 Análise de comparações

Os gráficos abaixo, indicam as diferentes versões do Quicksort, representadas de acordo com a quantidade de comparações conforme o tamanho do vetor, dessa forma, para cada tipo de vetor (aleatório, crescente e decrescente) temos um gráfico.





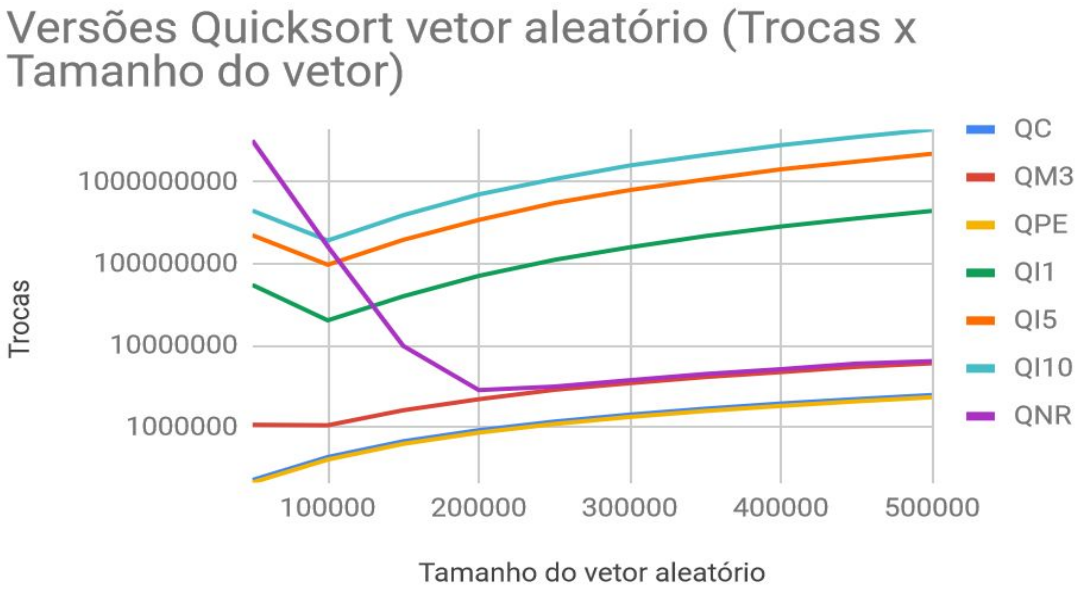


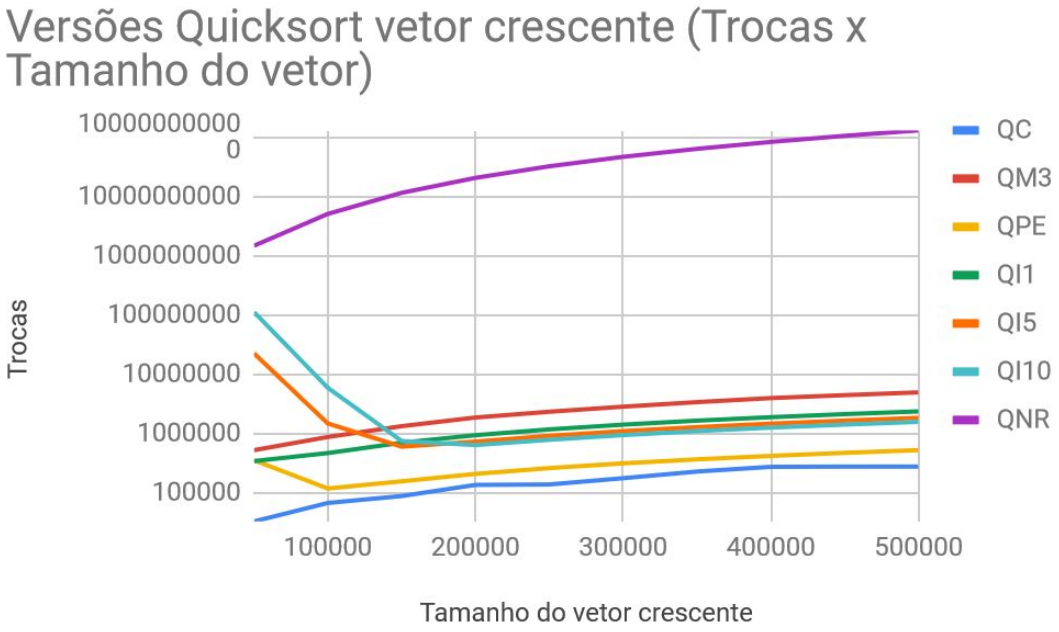
Ao analisar os gráficos, podemos observar que o número de comparações não é necessariamente proporcional ao tamanho do vetor, mas que o mesmo depende da forma em que o conjunto de dados está ordenado.

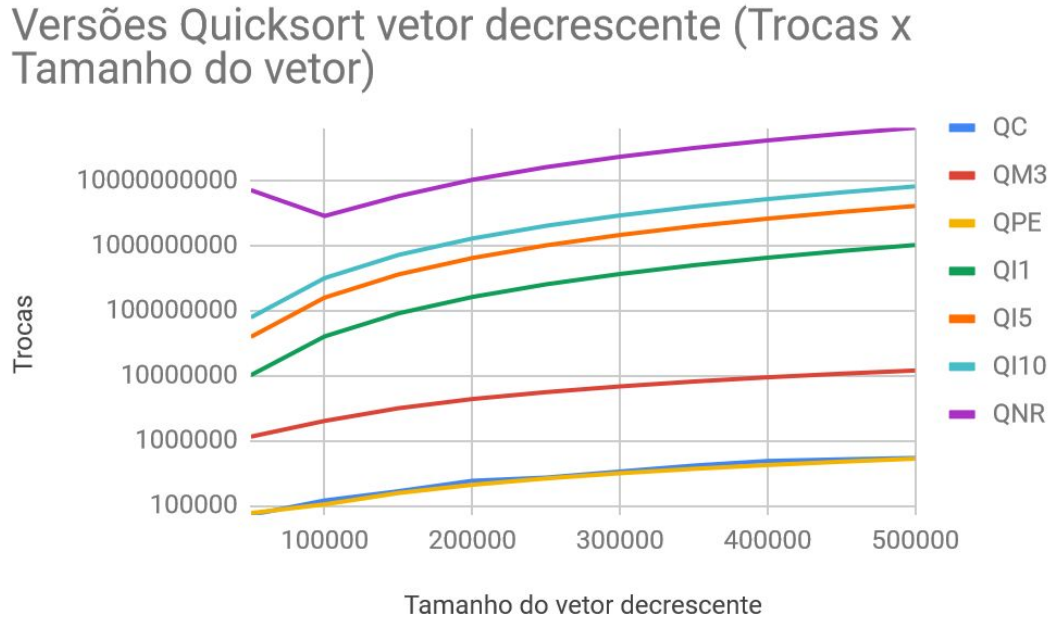
Se repararmos no gráfico de vetor decrescente, é possível notar que a partir de 100000 elementos, os algoritmos passam a comparar de maneira diretamente proporcional ao tamanho do vetor, porém para valores abaixo de 100000, esse comportamento difere para cada versão do Quicksort em questão. Além disso, nesse tipo vetor (decrescente), o número de comparações é bem distinto de algoritmo para algoritmo, já no vetor crescente, QPE e QNR se assemelham entre si e se diferem dos demais.

4.2.3 Análise de trocas

Os gráficos abaixo, indicam as diferentes versões do Quicksort, representadas de acordo com a quantidade de trocas conforme o tamanho do vetor, dessa forma, para cada tipo de vetor (aleatório, crescente e decrescente) temos um gráfico.







No caso dos gráficos de número de trocas, também se observa algo parecido com os de número de comparações, onde para vetores decrescentes, os algoritmos se distinguem bastante, porém, nesse caso, tem-se a exceção do QPE e QC, que são praticamente idênticos em relação às trocas realizadas. Já no caso do vetor ordenado (crescente), o número de trocas entre os algoritmos é bem parecido, salvo o caso da versão QNR, que realiza um número de trocas bem superior às demais versões do Quicksort.

5. Conclusão

Ao finalizar o trabalho, foi possível perceber três etapas distintas durante o processo de seu desenvolvimento, a primeira delas correspondeu à implementação do programa responsável por implementar as diferentes versões do algoritmo de Quicksort com base nos diferentes parâmetros. A segunda fase, foi após o término do desenvolvimento do software, e consistiu na coleta dos dados resultantes da ordenação do Quicksort, por meio da execução do programa. Por fim, a última etapa do trabalho, foi organizar os dados resultantes em tabelas, plotá-los em gráficos e realizar uma análise crítica dos resultados feitas.

Dessa forma, foi possível reconhecer as partes mais desafiadores de cada etapa. Na primeira, o obstáculo consistia basicamente de entender o funcionamento das variações do Quicksort e encarar as peculiaridades de implementação de cada caso. Na etapa de coleta, o desafio era conseguir um ambiente estável para rodar grande quantidade de dados e um período de tempo considerável, isso foi solucionado pelo uso do sistema operacional ubuntu com pilha modificada. Já no caso da etapa de análise, o grande desafio era reconhecer as informações e cruzamento de dados relevantes e tirar as conclusões a partir dos mesmos.

Sendo assim, com a conclusão deste trabalho, foi possível aprender mais a fundo sobre as diferentes implementações do Quicksort, de modo a saber utilizá-las nas situações em que são vantajosos e evitá-las em casos que possuem um desempenho ruim. Por fim, também foi possível adquirir conhecimento acerca do uso de diferentes sistemas operacionais e a prática de análise de dados.

6. Referências

- Felipe, Henrique. Quicksort mediana de três. [Último acesso em 13/06/2019]. Disponível em: <https://www.blogcyberini.com/2018/08/quicksort-mediana-de-tres.html>.
- Sach Gadgets. Quicksort to sort elements in ascending order using the first element as the Pivot. [Último acesso em 13/06/2019] . Disponível em: <https://sachgadgets.blogspot.com/2012/03/quicksort-to-sort-elements-in-ascending.html>.
- Geek Community. Insertion Sort. [Último acesso em 13/06/2019]. Disponível em: <https://www.geeksforgeeks.org/insertion-sort/>.
- Geek Community. Stack Data Structure (Introduction and Program). [Último acesso em 13/06/2019]. Disponível em: <https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>.
- Geek Community. Iterative Quick Sort. [Último acesso em 13/06/2019]. Disponível em: <https://www.geeksforgeeks.org/iterative-quick-sort/>.
- AZEREDO, Paulo A. (1996). *Métodos de Classificação de Dados e Análise de suas Complexidades*. Rio de Janeiro: Campus.
- Chaimowicz, Luiz; Prates, Raquel. Slide “Ordenação: Quicksort” Departamento de Ciência da Computação (UFMG).