Trabalho Prático 2: Biblioteca Digital de Arendelle

Guilherme de Abreu Lima Buitrago Miranda
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
guilhermemiranda@dcc.ufmg.br

1. Introdução

O Trabalho Prático 2 da disciplina de Estrutura de Dados consiste numa solução computacional para ordenar os livros da nova biblioteca digital de *Arendelle*, proposta pelo Plano de Democratização do Acesso à Educação (PDAE) proposto por Elsa, atual rainha.

Em razão da enorme quantidade de itens a serem disponibilizados digitalmente, o uso de um algoritmo de ordenação se mostra interessante para a organização da biblioteca. Para encontrar qual a melhor variação do QuickSort para essa tarefa, foi desenvolvido um código na linguagem C++ que, além de ordenar grandes arrays, nos permite comparar os resultados de tempo, número de comparações e número de movimentações.

O algoritmo de ordenação Quick Sort é amplamente utilizado em soluções computacionais, já que tem uma baixa complexidade no caso médio O(n log n) e seu funcionamento é fácil de ser compreendido. De maneira geral, o método de ordenação QuickSort escolhe um elemento do array, denominado pivô e particiona o array em dois, colocando em um array os elementos menores que ele e, no outro array, os elementos maiores. Posteriormente, o método é chamado de maneira recursiva, particionando novamente todos os arrays gerados pelo array original, até que cada partição tenha apenas um elemento. Quando isso acontece, a pilha (stack) de execução volta ordenando os vetores, gerando, ao final, um vetor completamente ordenado.

Para encontrarmos qual o método mais adequado para ordenar os livros da biblioteca digital de *Arendelle*, sete variações do QuickSort foram implementadas e testadas. São elas:

1. **Quicksort clássico**. Seleção de pivô usando o elemento central.
2. **Quicksort mediana de três**. Seleção do pivô usando a “mediana de três” elementos, em que o pivô é escolhido usando a mediana entre a chave mais à esquerda, a chave mais à direita e a chave central (como no algoritmo clássico).
3. **Quicksort primeiro elemento**. Seleção do pivô como sendo o primeiro elemento do subconjunto.
4. **Quicksort inserção 1%**. O processo de partição é interrompido quando o subvetor tiver menos de k = 1% chaves. A partição então deve ser ordenada

usando uma implementação especial do algoritmo de ordenação por inserção, preparada para ordenar um subvetor. Seleção de pivô usando a "mediana de três" elementos, descrita acima.

5. **Quicksort inserção 5%**. Mesmo que o anterior, com k = 5%.
6. **Quicksort inserção 10%**. Mesmo que o anterior, com k = 10%.
7. **Quicksort não recursivo**. Implementação que não usa recursividade. Utiliza pilha para simular as chamadas de função recursivas e identificar os intervalos a serem ordenados a cada momento. A seleção do pivô deve ser feita assim como no Quicksort clássico.

Para comparação de resultados, foram feitos testes com vetores gerados de forma aleatória, vetores em ordem crescente e vetores em ordem decrescente. Além disso, foram testados vetores de 50.000 até 500.000 elementos, em intervalos de 50.000. A solução computacional faz uso do paradigma de orientação a objetos e a estratégia de divisão e conquista. Além disso, implementa a estrutura de dados pilha (stack), utilizada na variação não recursiva do QuickSort.

2. Implementação

Para a implementação das diversas variações do QuickSort, foi escolhida a linguagem de programação C++, já que a mesma permite a implementação segundo o paradigma de orientação a objetos. Dessa forma, pode-se desenvolver um código bastante modularizado e organizado, a fim de facilitar o entendimento de um terceiro.

Dessa forma, além do main, foram implementados as classes "geraVetor", "quickSort" (que é separada em três arquivos principais, explicados posteriormente) e "pilha" (utilizada na implementação do QuickSort não recursivo).

A classe "geraVetor", como evidente em seu próprio nome, tem como objetivo gerar arrays aleatórios, crescentes ou decrescentes, para que as variações do QuickSort sejam devidamente testadas. Além disso, existe nessa classe uma função "clonaVetor", que clona o vetor para que o mesmo seja impresso após a saída padrão do programa, em que são mostrados a mediana do tempo e a média de movimentações e comparações.

Já a classe quickSort, embora tenha apenas um arquivo com extensão ".h", tem três arquivos ".cpp", com a intenção de facilitar o entendimento e a implementação modularizada de todas as sete variações do QuickSort. Além disso, tem também a implementação de um tipo enumeração, para decidir de qual maneira o pivô será escolhido (modo clássico, mediana de três ou primeiro elemento). Por fim, é implementada uma struct que armazena os parâmetros de comparações e de movimentos, essenciais para a análise experimental, assim como a sobrescrita do operador "+=", útil nos momentos de incremento das variáveis supracitadas.

No arquivo "classico.cpp" tem-se a implementação de três métodos. O método "classico" é o primeiro a ser chamado, recebendo o índice do elemento mais

à esquerda, do elemento mais à direita, o vetor e a maneira com a qual o QuickSort deve ser calculado (classico, mediana de três ou primeiro elemento). com isso, o método chama o procedimento de partição e, posteriormente, faz chamadas recursivas, a depender do estado atual do vetor.

O procedimento supracitado “particao” trata de, ao receber um array qualquer, o índice do elemento mais a esquerda, do elemento mais a direita e o pivô, gera dois subarrays ordenados, fazendo a troca (swap) de elementos a fim de obter, na direita, o vetor com elementos maiores que o pivồ e, à esquerda, o vetor com elementos menores que o pivô. Por fim, no mesmo arquivo, existe o método “medianaDeTres”, que calcula a mediana de três elementos e incrementa na variável resultado do tipo “stats” as comparações feitas, retornando a mediana desejada.

No arquivo “inserção.cpp” encontra-se mais dois métodos. O primeiro é homônimo ao arquivo e recebe o índice dos elementos da esquerda e da direita, além do vetor e do número de elementos a serem cortados, a depender da porcentagem k de chaves a serem inseridas por meio do método InsertionSort. Assim, o programa verifica se o número de elementos é menor que o de elementos passados por parâmetro. Caso contrário, é feito a ordenação de maneira já explicada anteriormente, utilizando o método da mediana de três e o procedimento partição, chamando o método inserção recursivamente.

Contudo, caso a condição para a porcentagem dos elementos seja satisfeita, o método “insere” é invocado, inserindo os elementos no array de maneira ordenada segundo o algoritmo InsertionSort, em que o array atual é percorrido da esquerda para a direita, procurando por um elemento maior que o elemento a ser inserido. Quando a afirmação for verdadeira, o elemento é inserido e o índice dos elementos posteriores é acrescido de uma unidade, pois os mesmo são deslocados uma casa para a direita.

O último dos arquivos que implementam as função da classe estática QuickSort é o “nãoRecursivo.cpp”, em que é implementada a variação não recursiva do algoritmo de ordenação referido. Para tal, é feita uma importante mudança no paradigma: as chamadas recursivas são trocadas por inserções em duas pilhas diferentes, que armazenam os subvetores a esquerda e à direita do array principal respectivamente, até que as pilhas se esvaziem e o novo vetor ordenado seja montado completamente.

Como pode ser observado, para a implementação da variação não recursiva do QuickSort é necessária a criação de uma pilha (stack), que foi também implementada. Foram necessários poucos métodos, que podem ser vistos nos arquivos “pilha.h” e “pilha.cpp”. São eles: construtor, destrutor, “nElements”, que retorna o número de elementos da pilha, “push”, que adiciona um elemento no final da pilha e “pop”, que retira o último elemento da pilha.

Por fim, tem-se a implementação do arquivo “main.cpp”. Nele, a função homônima pega os argumentos passados por CLI (linha de comando) e chama o

procedimento de execução, a fim de fazer os testes requeridos. No procedimento, é chamado algum dos métodos da classe QuickSort, a depender da escolha de quem está executando o programa, e o vetor em questão é ordenado vinte vezes, com o objetivo de pegar a mediana dos tempos e a média de número de movimentações e número de comparações retornadas pela classe estática em questão.

Posteriormente, é impressa a saída normal do programa (tipo de algoritmo usado, tipo do vetor, tamanho do vetor, média de comparações, média de movimentos e mediana do tempo), Além disso, se o último argumento passado for "-p", são impressos os vetores originais que passaram pelo algoritmo de ordenação (para tal, foi criado um método que percorre o array imprimindo elemento a elemento).

A entrada dos dados por linha de comanda é feita da seguinte forma: "./nome do programa tipo_do_algoritmo tipo_do_vetor tamanho_do_vetor", sendo os tipos do algoritmo as variações supracitadas do QuickSort e o tipo do vetor variando entre aleatório, ordenado crescente e ordenado decrescente.

O compilador utilizado para os testes foi o GCC - g++ versão 8.3.0 e mostrou-se uma boa ideia utilizar a flag -O3, com a intenção de otimizar da melhor maneira possível o código desenvolvido, já que a execução de todos os 210 casos de teste é consideravelmente demorada.

3. Instruções de compilação e execução

Os testes foram todos realizados em ambiente Linux, e o compilador recomendado é o g++. Para compilação e execução, deve se executar, primeiramente, o comando make, que utiliza do arquivo makefile para compilar os arquivos em c++ 11 e com todas as otimizações possíveis, chamando o arquivo final de tp2. Posteriormente, para a execução de uma ordenação, deve-se digitar no terminal: "./tp2 <variacao> <tipo> <tamanho> [-p]". Os valores possíveis para variação são: QC (clássico), QM3 (mediana de três), QPE (primeiro elemento), QI1 (inserção 1%), QI5 (inserção 5%),QI10 (inserção 10%) e QNR (não recursivo) e para o tipo são Ale (aleatório), OrdC (ordenado crescente) e OrdD (ordenado decrescente).

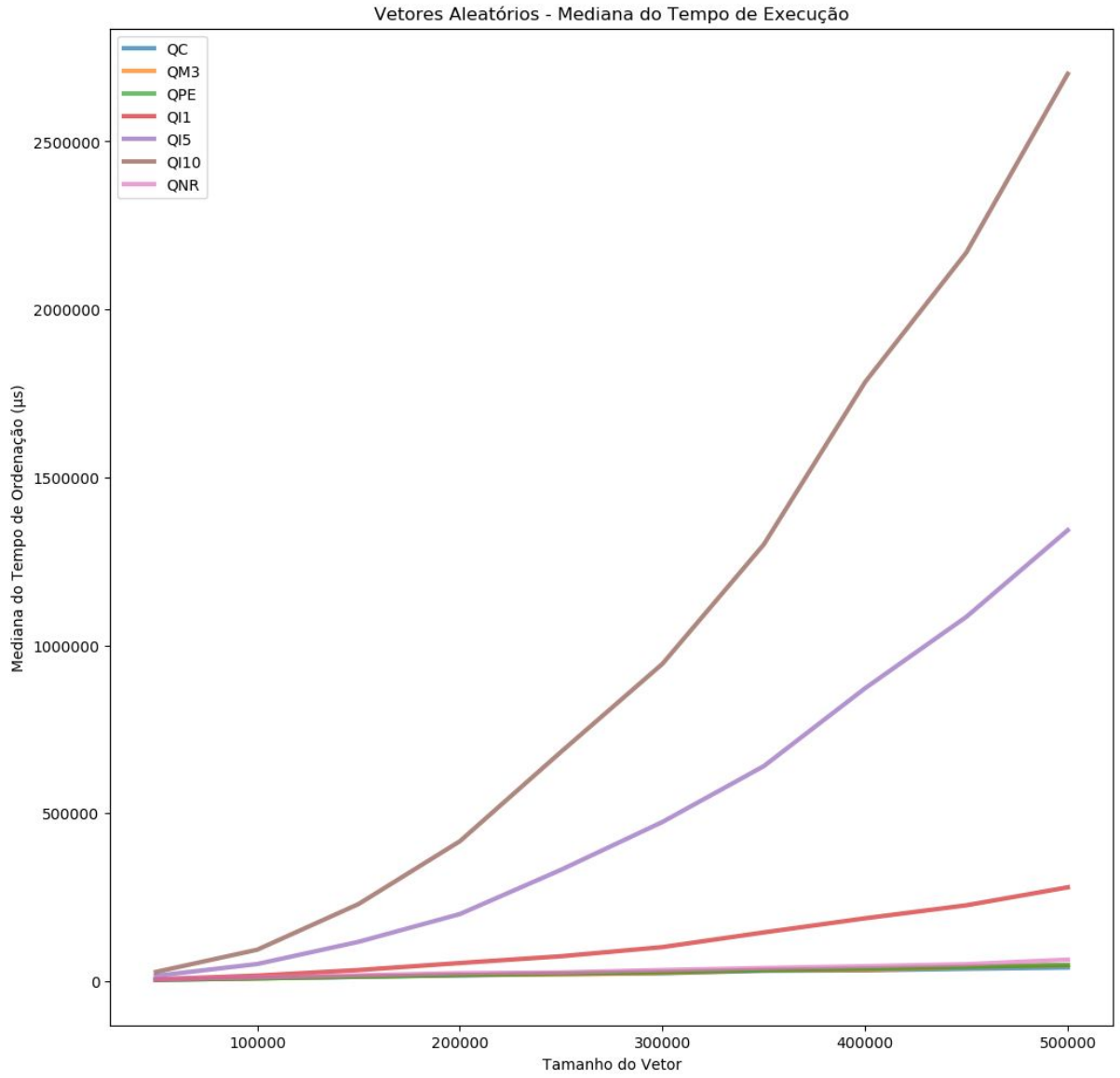
A saída do programa será: <variacao> <tipo> <tamanho> <n comp> <n mov> <tempo>, em que nComp e nMov serão a média do número de comparações e do número de movimentações, enquanto o tempo será a mediana das vinte execuções.

Importante destacar, também, a necessidade de aumentar o tamanho da stack padrão para a execução dos casos maiores, que pode ser feita segundo o comando "ulimit -s unlimited" no terminal. Além disso, para garantir a inexistência de vazamentos de memória (memory leaks), o utilitário valgrind foi largamente utilizado, com o objetivo de manter as boas práticas de programação.

4. Análise Experimental

Para realizar os 210 testes, foi desenvolvido um script bash executado num terminal linux numa máquina com as seguintes especificações: Ryzen 1700 3.0GHz (3.7GHz Turbo), 8-Core 16-Thread contando com 8GB RAM DDR4 2400Mhz. Além disso, para evitar a troca de contexto da CPU durante a execução do programa, foi instalada a distribuição linux Manjaro Architect, que mostrou-se bastante adequada para a tarefa, já que não tem nem mesmo interface gráfica, gerando o mínimo de interferências possível.

A execução dos casos de teste demorou 1 hora, 21 minutos e 43 segundos e vários gráficos foram gerados posteriormente (utilizando os resultados obtidos disponíveis em anexo no arquivo out.txt). Interessante observar, por exemplo, o tempo médio gasto para ordenar um vetor aleatório:

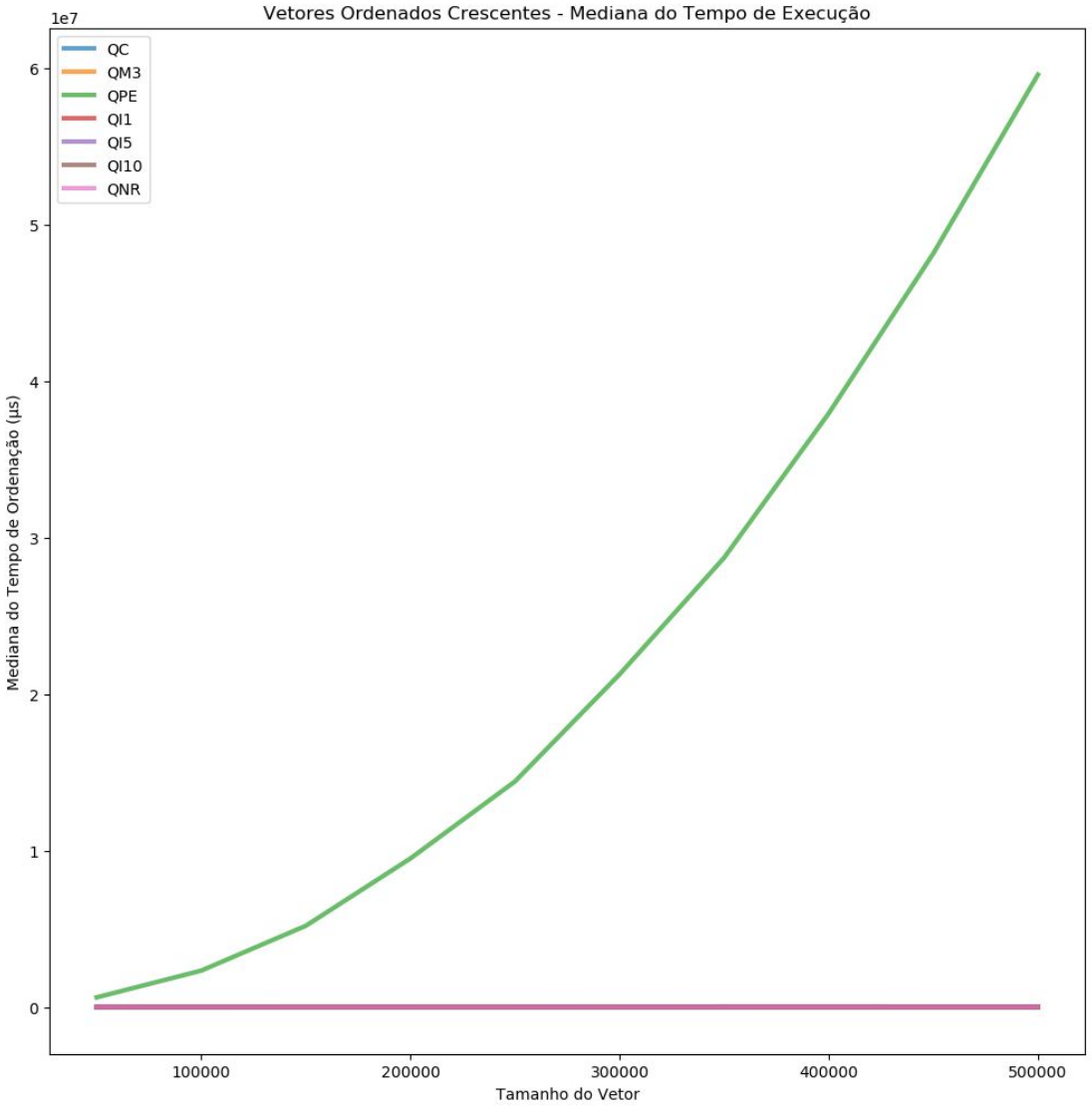


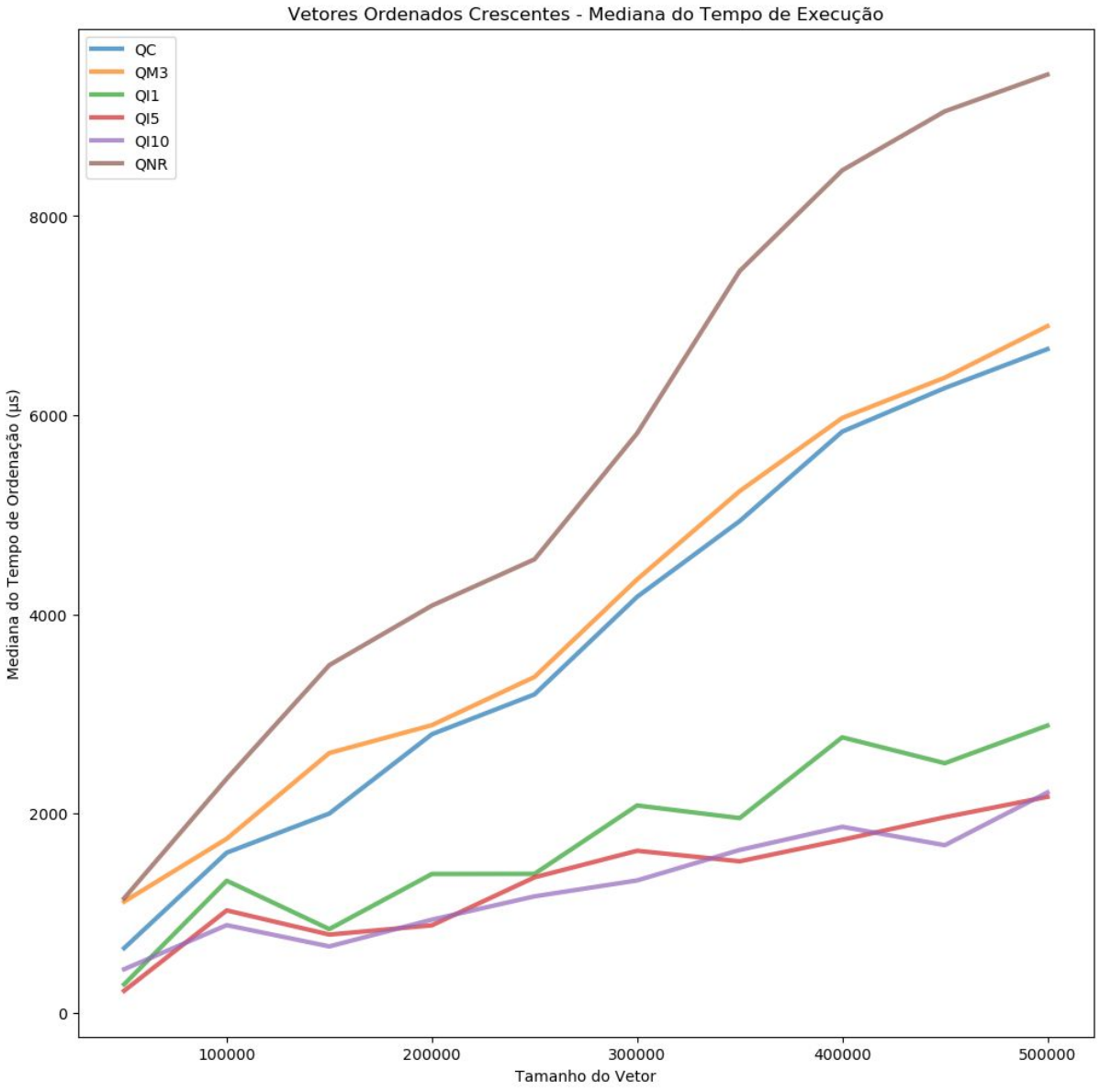
Pode-se observar, por exemplo, como, para um vetor aleatório, quanto maior a porcentagem utilizada na variação de inserção, maior o tempo gasto, sendo esses as três variações mais lentas. O número de movimentações e comparações segue a

mesma tendência, ou seja, é mais alto para as maiores variações de inserção, e pode ser observado no diretório “gráficos” na pasta do projeto.

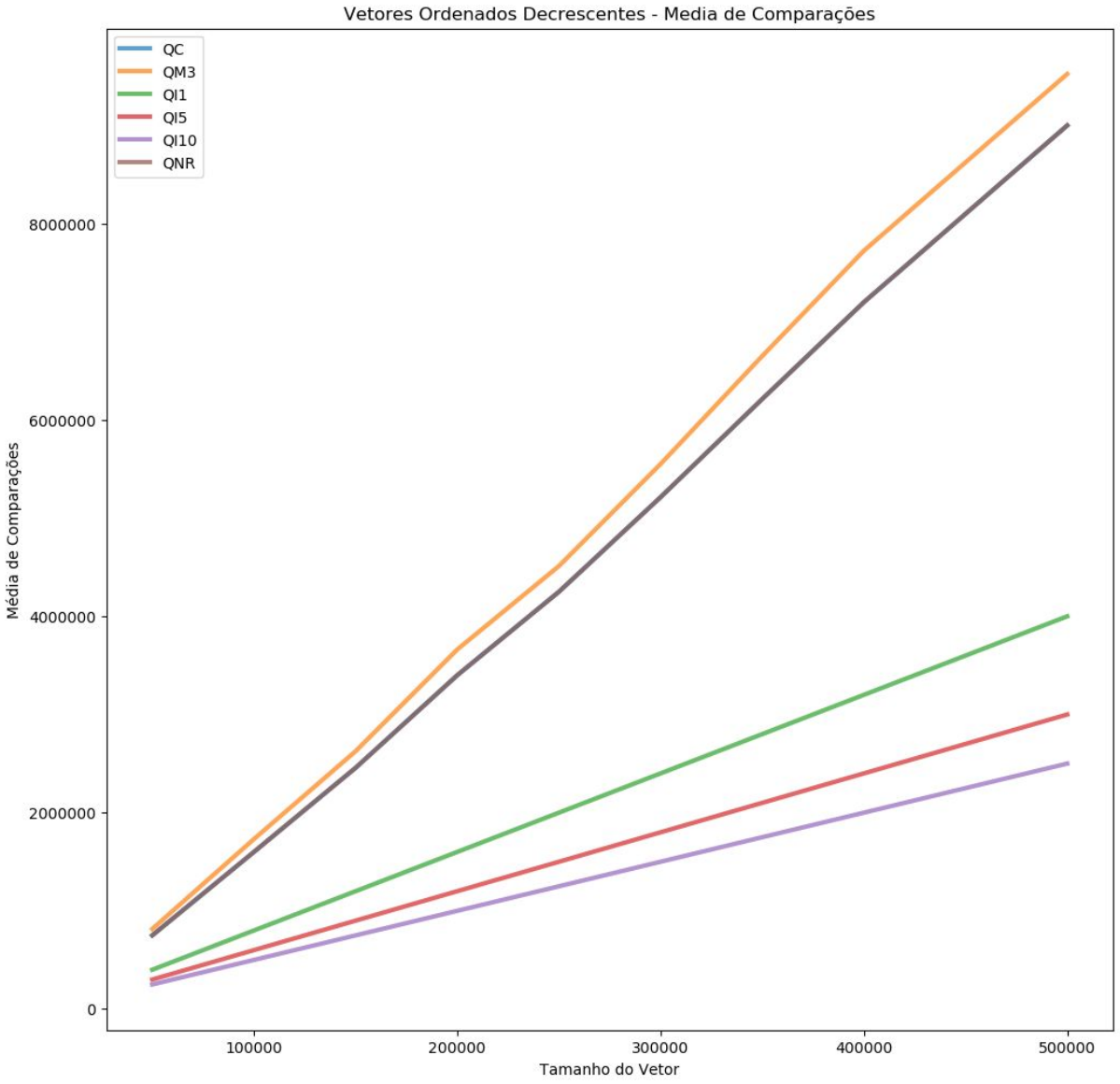
Já para os vetores ordenados é interessante notar que, como todas as variações do QuickSort chamam o método “particiona” e o mesmo faz a troca (swap) dos elementos maiores e menores do array comparando com o pivô, com exceção da variação que o pivô é o primeiro elemento, em todos os outros, logo após a primeira chamada, temos dois arrays já ordenados de maneira crescente, o que permite apontar dois pontos importantes: 1) A variação QuickSort primeiro elemento (QPE) é péssima para vetores ordenados (crescentes ou decrescentes) e 2) as variações de inserção lidam bem com dados já ordenados.

Isso pode ser visto experimentalmente conforme os dois gráficos abaixo. O primeiro mostra como a variação QPE é muito mais lenta que todas as outras e a segunda mostra como, ignorando a variação QPE para que os dados sejam melhor visualizados, quanto maior a porcentagem k para iniciar o InsertionSort na variação de inserção do quicksort, mais eficiente é a solução.

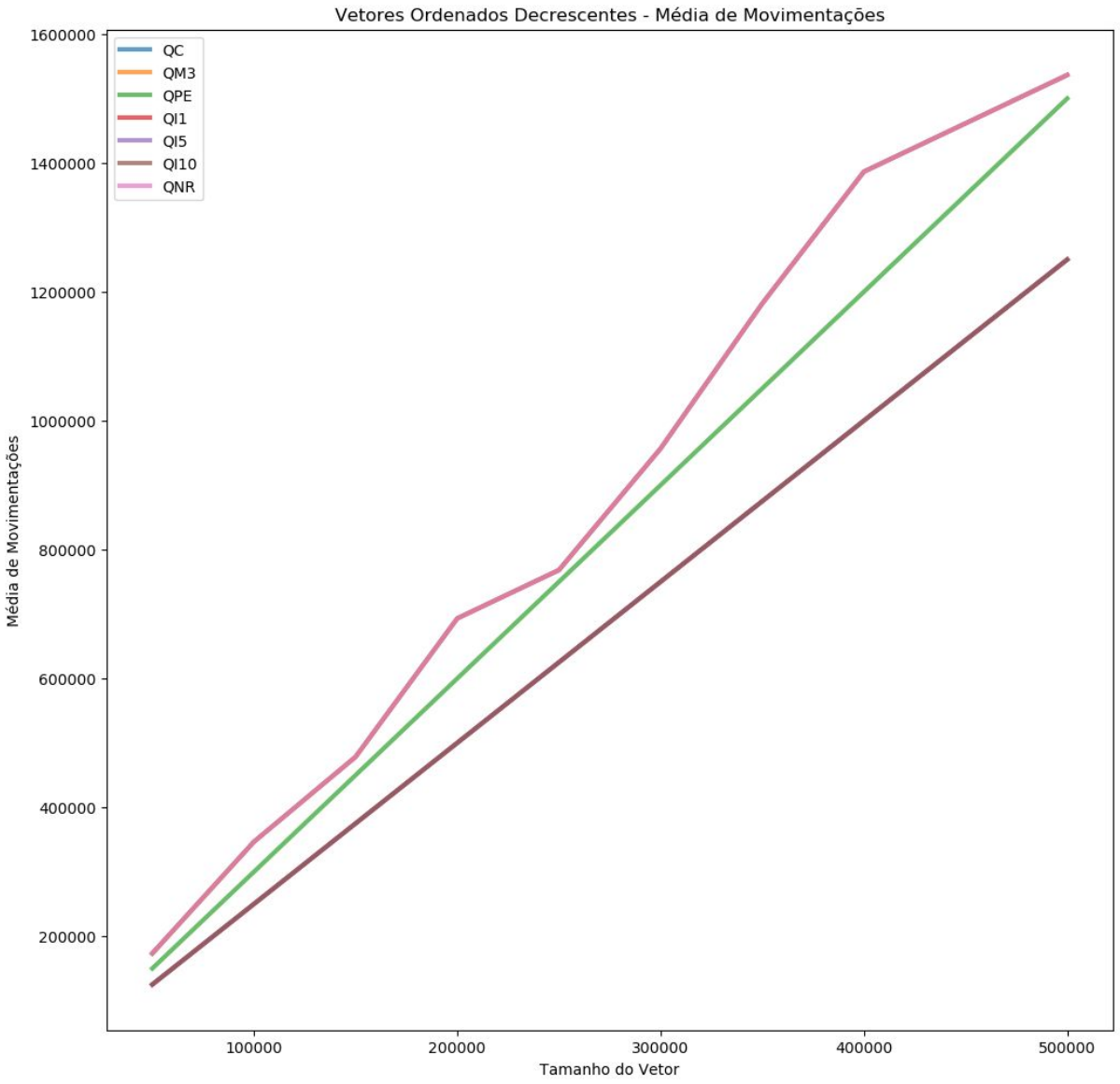




Importante perceber que, assim como nos vetores aleatórios, nos vetores ordenados (crescentes ou decrescentes), a média de comparações segue a tendência de crescimento ou decrescimento do tempo, conforme pode ser observado nos gráficos abaixo (novamente, desconsiderando a variação QPE, que inviabiliza a visualização dos dados)



Já a média de movimentações, apesar de seguir a mesma tendência, devido a implementação do algoritmo, gera um gráfico um pouco diferente, mas que tem pouca diferença em termos práticos relativos à eficiência das variações.



5. Conclusão

O desenvolvimento do algoritmo mostrou-se bastante proveitoso para, além de consolidar os conhecimentos adquiridos a respeito do QuickSort clássico durante as aulas de Estrutura de Dados, perceber e reconhecer, também, as várias diferentes implementações utilizando o mesmo paradigma de ordenação.

Com a obtenção e análise experimental dos dados também foi possível distinguir quais as melhores implementações do QuickSort para cada contexto diferente. Se considerarmos um conjunto aleatório de dados, que são comuns de maneira geral, a variação Classica e Mediana de Três são as melhores escolhas, pois ambas são executadas de maneira mais rápida (conforme o primeiro gráfico do documento), caindo no caso de complexidade médio O(n log n).

Por outro lado, se o conjunto de dados a ser tratado está completamente (ou em sua maioria) ordenado de alguma forma, as variações de inserção são opções mais interessantes. Em suma, quanto mais ordenados estão os dados, maior deve ser o k escolhido, a fim de cair no melhor caso do algoritmo de inserção O(n).

Contudo, se os dados estão aleatorizados, as variações com inserção são extremamente perigosas, pois caem no pior caso do algoritmo de inserção (O(n^2)).

Além disso, as variações com o pivô no primeiro elemento e não recursiva também são interessantes de se analisar. Ambos são bastante parecidos com as outras variações para um conjunto de dados aleatórios, porém, como pode ser visto nos gráficos, são extremamente custosos caso o conjunto de dados esteja ordenado (o QPE é ainda mais custoso que o QNR nesse caso, mas existem opções mais eficientes que as duas na situação descrita).

A maior dificuldade do trabalho foi a implementação de soluções para a análise estatística dos casos de teste, ainda que a implementação das sete diferentes variações tenha sido bastante trabalhosa. Outrossim, foi necessária a pesquisa a respeito de como tratar argumentos passados ao programa por CLI (linha de comando) e a utilização da biblioteca de tempo (std::chrono).

Não obstante, o trabalho prático também foi muito proveitoso para treinar a implementação da estrutura pilha (stack), assim como a lógica de programação e a maior familiaridade com a linguagem C++. Interessante ainda revisar alguns conceitos como troca de contexto de CPU e overflow de tipos, já que apenas o int não foi capaz armazenar o número de comparações em alguns casos de teste.

6. Referências

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática 3a. Edição, Elsevier, 2012.

MS LATHA SHILVANTH. Java: For Programming (2018). Createspace independent Publishing Platform. ISBN-10: 1985254603

Command line arguments in C/C++. Disponível em: <https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/>. Acesso em 5 jun. 2019.