ESTRUTURA DE DADOS – TRABALHO PRÁTICO II

Luiz Felipe Couto Gontijo – 2018054524

UNIVERSIDADE FEDERAL DE MINAS GERAIS

QUICKSORT

1. Introdução

Esse projeto consiste em desenvolver um programa que implemente certas variações do algoritmo de ordenação *quicksort*, assim como a realização de vários testes e comparações de performance, para que seja possível a avaliação do melhor meio para organizar a biblioteca virtual do reino de Aredelle e democratizar o acesso à informação. Nesse documento, serão discutidos as decisões de implementação, as estruturas de dados criadas para a resolução do problema e também as respectivas comparações entre as diversas variações do algoritmo do *quicksort*.

1.1 Para Compilar

Para compilar o programa, basta rodar o comando - *make* - dentro do diretório onde se encontra o arquivo *main.cpp* . Para executar, temos duas opções:

- Sem a utilização de parâmetros:Apenas rodar *./main*, na qual o programa irá criar nove tabelas referentes a cada varição de quicksort, com todos os tamanhos, sendo 3 tabelas relacionadas ao nº de comparações, 3 relacionadas ao nº de movimentos e outras 3 relacionadas ao tempo de execução(esse comando pode demorar bastante para gerar todas as tabelas, dependendo do ambiente de execução).

- Com os parâmetros: ao adicionar parametros, deve-se incluir a variação do quickSort desejada, o tipo de vetor e o tamanho do vetor(nessa ordem). Pode-se também incluir o parâmetro *-p*, que imprimirá o vetor sempre que houver uma troca de elementos. Segue abaixo um exemplo de entrada:

VARIAÇÃO:

QC - QUICKSORT CLÁSSICO
QM3 - QUICKSORT MEDIANA DE TRÊS
QPE - QUICKSORT PRIMEIRO ELEMENTO
QI1 - QUICKSORT + 1% INSERTION SORT
QI5 - QUICKSORT + 5% INSERTION SORT
QI10 - QUICKSORT + 10% INSERTION SORT
QNR - QUICKSORT NÃO RECURSIVO

TIPO DE VETOR

Ale - VETOR ALEATÓRIO
OrdC - VETOR CRESCENTE
OrdD - VETOR DECRESCENTE

EXEMPLO

./main QPE Ale 100

1.2 Saída

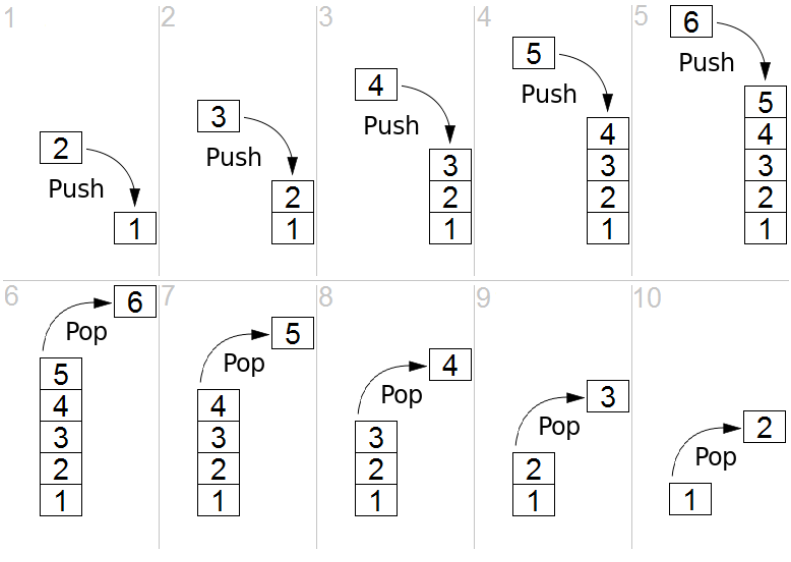
A saída do programa é, além das variações escolhidas acima, o número de comparações realizadas, o número de movimentos realizados e o tempo de execução em microssegundos(tudo na mesma linha). Como dito anteriormente, caso incluso o *-p*, será impresso o vetor a cada modificação realizada. Uma decisão de implementação foi imprimir o vetor (no caso de utilizado o *-p)* antes da impresão final do número de comparações, número de movimentações e tempo. Essa decisão foi tomada por uma questão de eficiência(visando maior rapidez do programa), já que os vetores serão impressos no momento em que eles forem modificados. Tornando, assim, desnecessário o armazenamento desses vetores em outras variáveis e aprimorando a performance do programa.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Estrutura de Dados

A implementação do programa teve como base a criação de uma classe principal QuickSort, onde são definidas as funções de todas as variações do algoritmo assim como todos os elementos (variáveis) necessários para os posteriores resultados dos testes. Exatamante seis das sete funções de *quicksort* são definidas recursivamente, na qual a função chama a si mesmo até que se atinja a condição de parada. A função de *quicksort* não recursiva, por sua vez, utiliza de uma estrutura de dados criada na classe secundária Pilha, que tem por função simular a entrada e saída de elementos até que a Pilha esteja vazia. A imagem abaixo representa essa última situação:



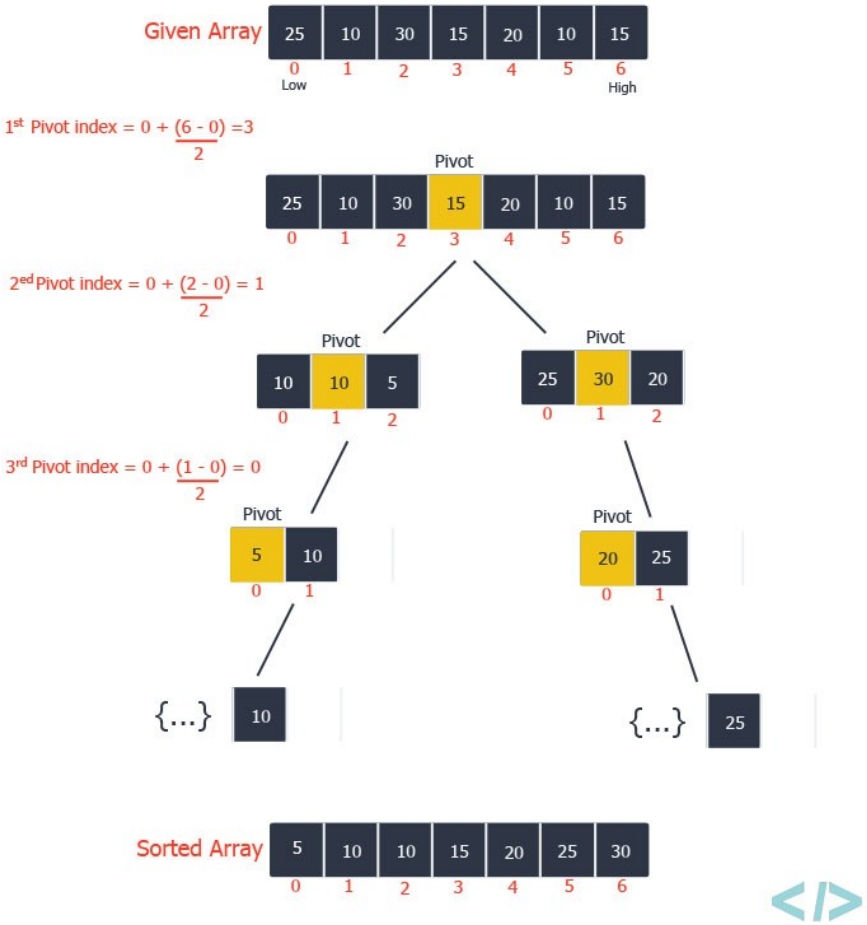
2.2 Classes

Assim como introduzido no tópico anterior, foram criadas uma classe principal QuickSort (implementada nos arquivos *quicksort.cpp* e *quicksort.h*) e uma classe secundária Pilha(implementada nos arquivos *pilha.cpp* e *pilha.h*).

2.2.1 QuickSort

Essa classe é responsável por definir as funções de ordenação:

- *qsort*: implementa o *quicksort clássico*, na qual o pivô é sempre o elemento central da partição. A imagem abaixo mostra como o *quicksort* funciona, valendo também para todas as variações, com mudança apenas do pivô.



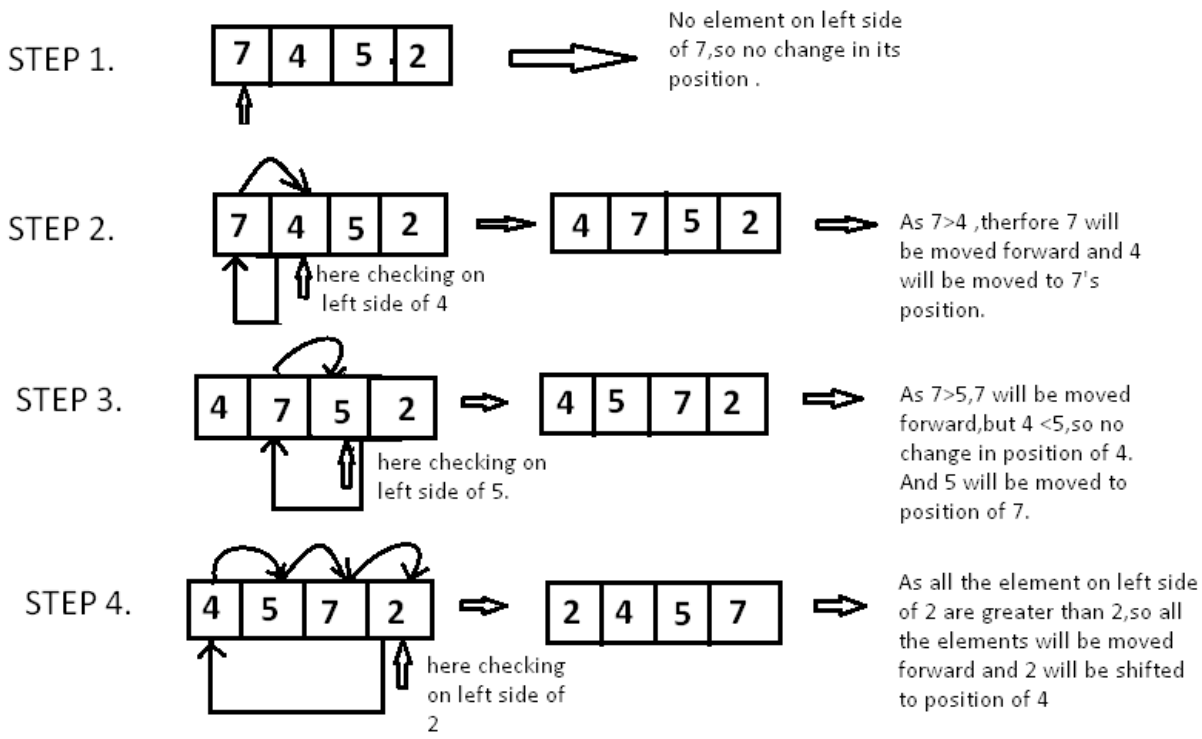
- *qsort_primeiro*: implementa o *quicksort* na qual o pivô é definido pelo primeiro elemento da partição.

- *q_sort_med3*: implementa o *quicksort* que define o pivõ como sendo a mediana entre o primeiro elemento, o elemento do meio e o último elemento de cada partição.

- *qsort_one_percent*: essa variação do quicksort escolhe o pivõ igual à anterior. Porém, quando as partições possuem menos de 1% de elementos em relação ao total de elementos

do vetor inicial, é utilizado o algoritmo de ordenação por inserção, também implementado nessa mesma classe QuickSort.

- *insertion_sort*: implementa o algoritmo de ordenação por inserção. A imagem abaixo exemplifica essa funcionalidade.



- *qsort_five_percent*: similar ao *quicksort_one_percent*, porém com 5% dos elementos.

- *qsort_ten_percent:* similar ao *quicksort_one_percent*, porém com 10% dos elementos.

- *qsort_non_recursive*: Variação que utiliza a classe secundária Pilha, na qual ao invés da pilha ser criada automaticamente(como ocorre nas chamadas recursivas), ela é criada manualmente, implementando-se todas as funcionalidades necessárias (adicionar elemento, retirar elemento e verificar se a pilha está vazia).

A classe QuickSort também possui algumas funções que são chamadas por aquelas que ordenam os vetores:

- *swap*: Troca dois elementos do vetor.

- *Print*: Funções que imprimem alguma variável da classe, como o número de movimentações ou númumero de comparações.

- *Get*: Funções que retornam alguma variável da classe, como o número de movimentações ou númumero de comparações.

2.2.2 Pilha

Para implementar a pilha, foram adicionadas apenas as funcionalidades mais básicas, como inicializar, adicionar e remover um elemento da pilha. O construtor inicia a pilha com nenhum elemento, e o tamanho é definido dinamicamente. A função *Push()* adiciona um elemento e a função *Retira()* remove um elemento (seguindo a lógica que o primeiro a entrar é o último a sair). Já a função *is_empty()* retorna *true* caso a pilha esteja vazia, e retorna *false* caso contrário.

2.2.3 Main

Mesmo não sendo uma classe, a função principal foi adicionada nessa secção pois nela também são implementadas as funções que ordenam os vetores utilizados para retirar a mediana dos 21 testes realizados para cada combinação de tamanho x variação de quicksort:

- *Ordenar_M()*: implementa um *quicksort* clássico para ordenar o vetor com os valores dos 21 testes em relação ao número de comparações e de movimentos, para ser possível indicar a mediana posteriormente.

- *Ordenar_Time()*: implementa um *quicksort* clássico para ordenar o vetor com os valores dos 21 testes em relação ao tempo de execução, para ser possível indicar a mediana posteriormente.

3. Análise de performance

Para fazer as comparações foram criados 9 gráficos:

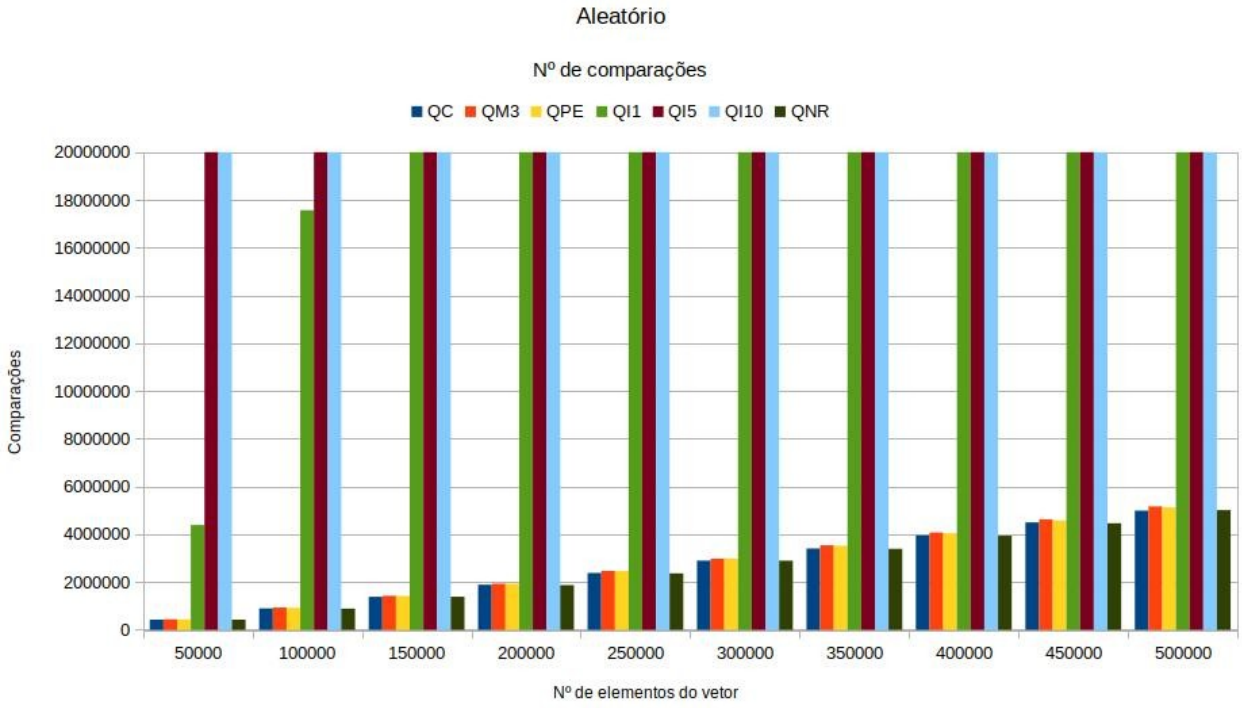
- 3 se referem às ordenações com vetores aleatórios (1 para o nº de comparações, 1 para o nº de movimentos e 1 para o tempo de execução)

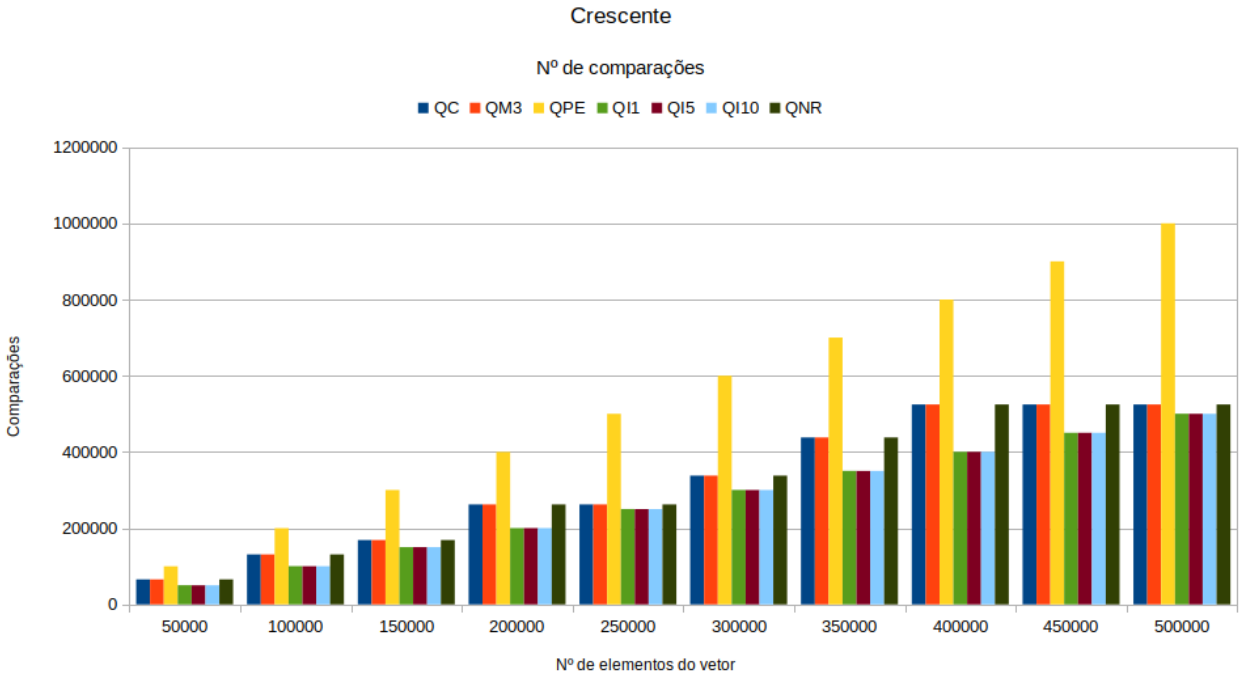
- 3 se referem às ordenações com vetores em ordem crescente (1 para o nº de comparações, 1 para o nº de movimentos e 1 para o tempo de execução)

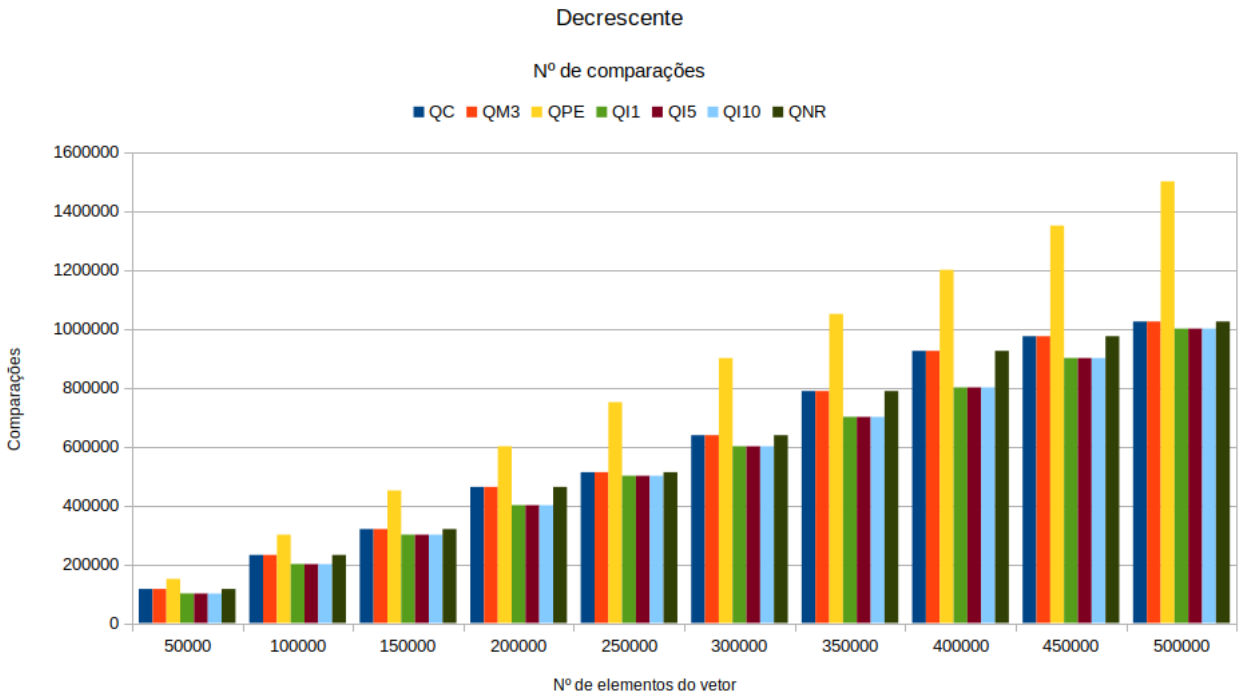
- 3 se referem às ordenações com vetores ordem decrescente (1 para o nº de comparações, 1 para o nº de movimentos e 1 para o tempo de execução)

Para cada dado, foram feitos 21 testes e foi utilizada a mediana dos mesmos. Alguns gráficos ultrapassam o limite da imagem, pois possuem valores muitos grandes em relação à outras variações de quicksort. Mesmo que não permita a comparação exata entre esses valores muito grandes, eles têm a função de destacar a discrepância entre os diversos tipos de algoritmos de ordenação utilizados.

3.1 Número de comparações

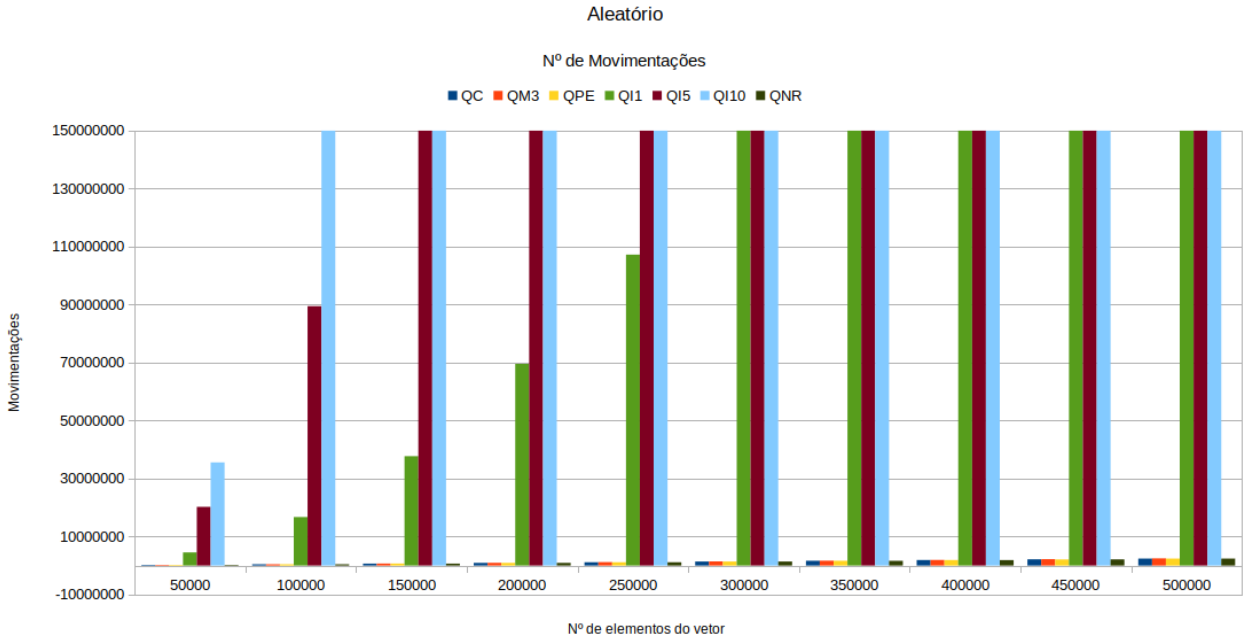


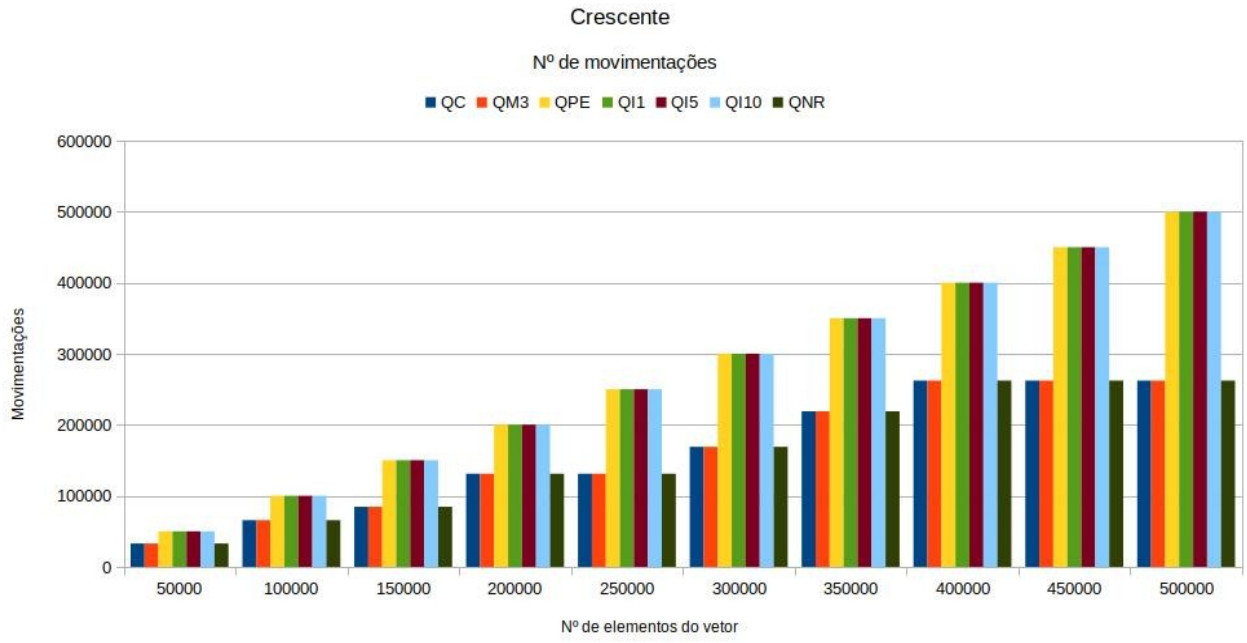


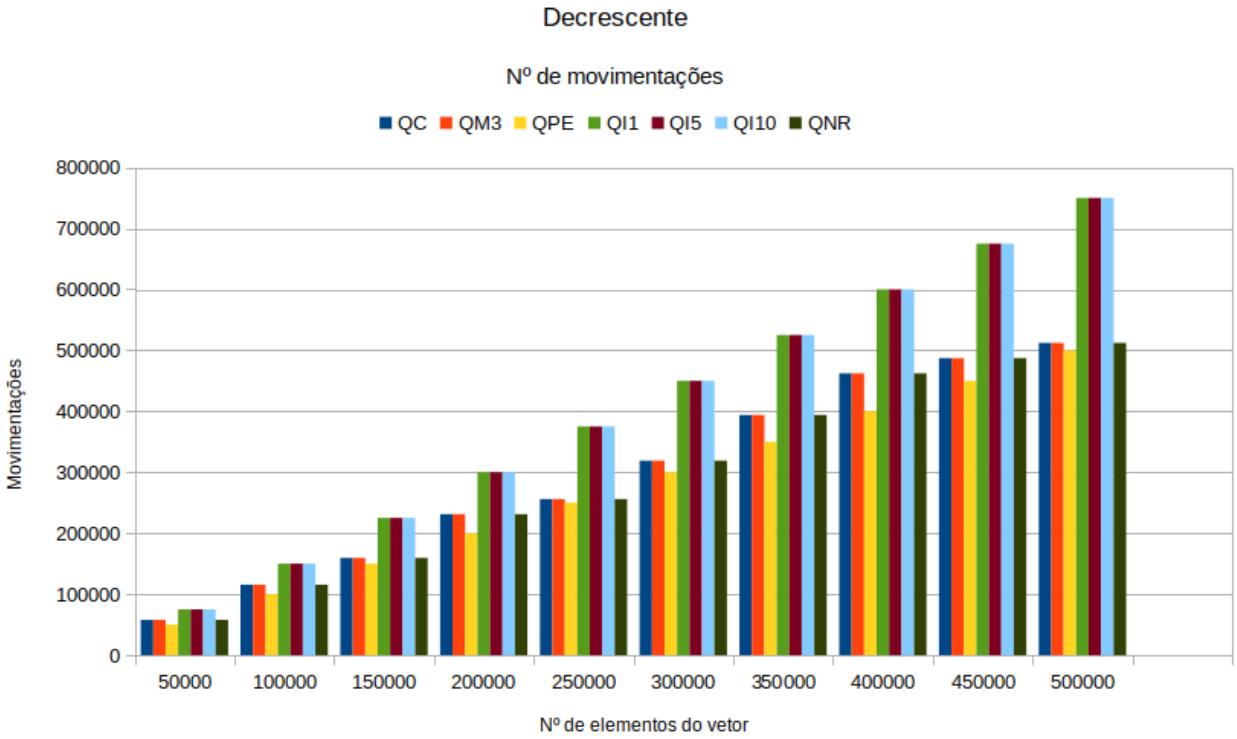


Analisando as informações contidas nos gráficos acima, é possível perceber que as comparações foram , de forma geral, maiores em vetores decrescentes, porém não é tão discrepante dos vetores aleatórios. Com relação ao aos *quicksorts*, é visível que aquele que menos fez comparações foi o Quicksort Clássico, bem próximo também ao Quicksort não recursivo. A diferença entre esses dois está na implementação da pilha. Como no não recursivo a pilha é criada manualmente, temos uma certa perda de eficácia quando comparada a pilha gerada automaticamente pelo QuickSort Clássico(de modo recursivo). Em relação ao maior número de comparações, vemos que, quanto mais se usa o método de ordenação por inserção, maior é o número de comparações. Isso ocorre, pois se realizam muito mais comparações por iteração na ordenação de inserção quando comparado ao quicksort. Desse modo, o "campeão" de comparações se torna o QI10, que quase não aparece no gráfico apresentado devido aos enormes valores. Algo interessante é que os QI(QuickSort + Inserção) possuem valores de comparações bem menores em vetores que já estão previamente ordenados do que aqueles aleatórios.

3.2 Número de movimentações

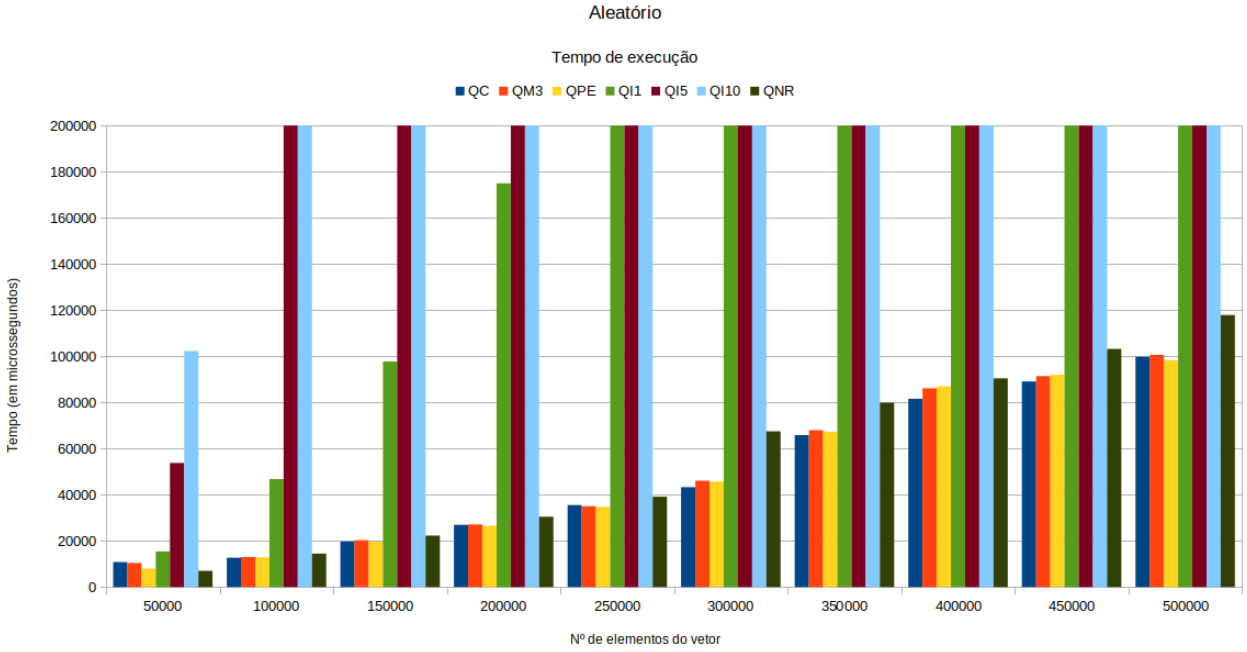


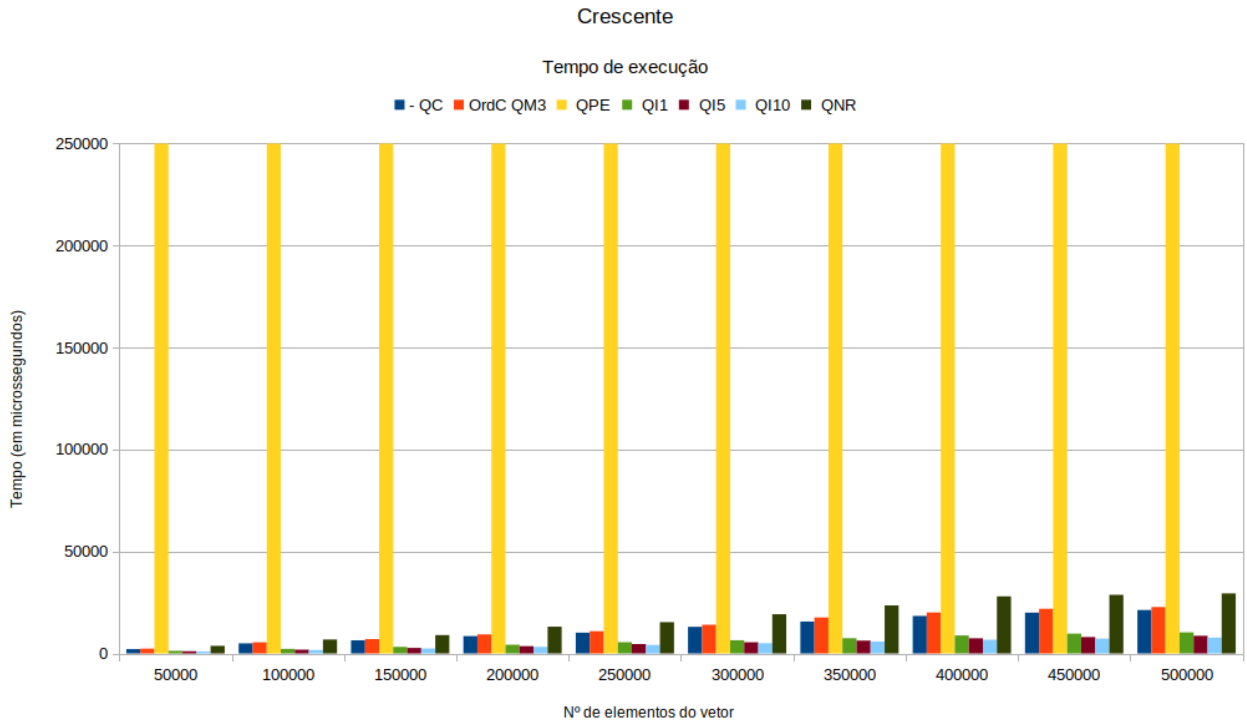


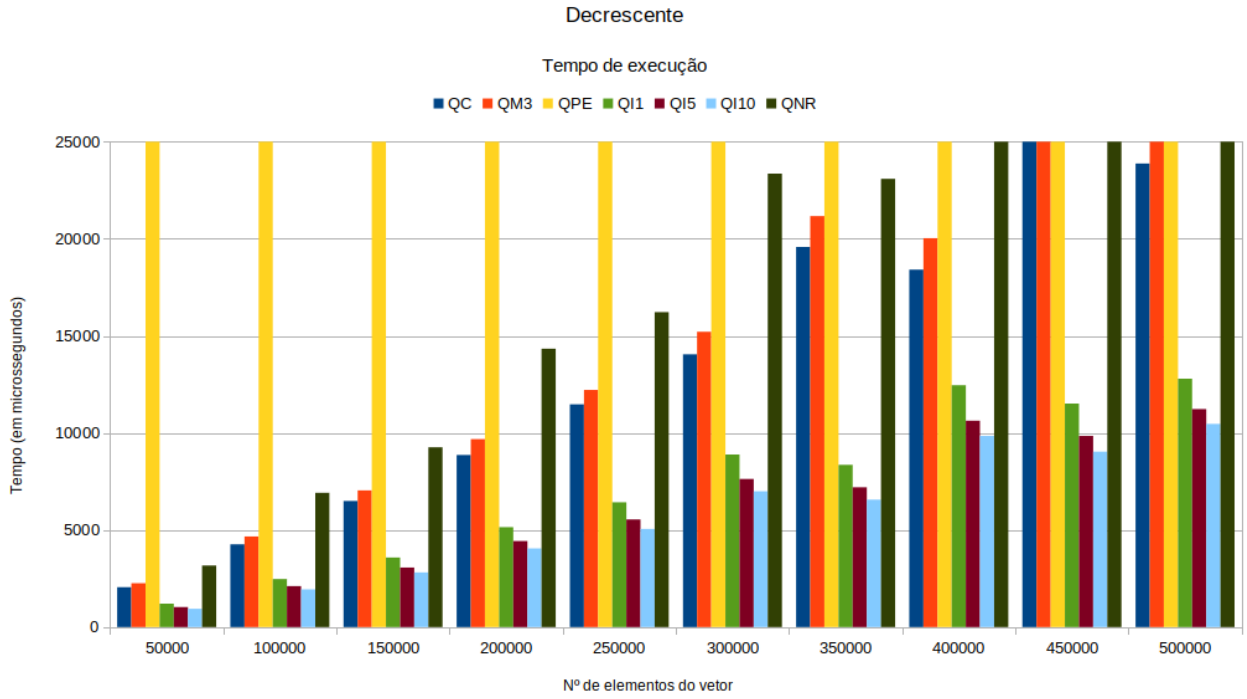


Aqui, é visível que o Insertion Quicksort foi aquele com o maior número de movimentações. É um resultado esperado ja que o método da inserção utiliza de bastantes movimentos dentro do vetor a ser ordenado. O maior (mesmo não aparecendo no gráfico devido aos enormes valores), é com certeza o QI10, já que utiliza mais do método de inserção do que o QI5 e QI1. Em relação ao menor número de movimentos, vemos que o QC, QNR e QM3 possuem valores bem similares, dando um destaque para os vetores já ordenados que, como esperado, possuem números de movimentação menores.

3.3 Tempo de execução







Avaliando os gráficos, é possível ver que o QPE (First element QuickSort) é o mais lento (com uma boa margem de diferença em relação aos demais). O pior cenário neste QS é o OrdC, mas podemos perceber que, quanto maior os vetores, mais o tempo de OrdC e OrdD ficam parecidom. Um ponto que vale destacar é que, quando utilizamos de um vetor aleatório, o tempo de QPE é realmente muito bom, em comparação com os demais(mantendo um nível razoável). Isso ocorre porque, como estamos usando o primeiro elemento como o pivô, no pior dos casos (vetor ordenado), ao invés de ir até a metade do array e, em seguida, dividir, O (n log n), o vetor é apenas dividido. Então temos O(n2).

4. Conclusão

A partir de todos os dados apresentados e análises feitas, é plausível concluir que o QuickSort mais eficiente para o caso de Arendelle é o QuickSort clássico (pivô como elemento central), que teve um menor tempo de execução e também número de movimentações e comparações mais “equilibrados”. Vale ressaltar que tanto o QM3 como o QNR também obtiveram resultados bastante satisfatórios, com uma performance bastante similar.

Elsa talvez prefira o QC já que ele é mais simples de ser implementado, não sendo necessária a criação da pilha manualmente como ocorre no QNR. Tendo, assim, um ganho de performance.

5. Referências

Ziviani, N. (2006).Projetos de Algoritmos com Implementações em Java e C++:Cap ́ıtulo 3: Estruturas de Dados B ́asicas. Editora Cengage.