Trabalho Prático 2: Biblioteca Digital de Arendelle

Gabriel Victor Carvalho Rocha

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

gabrielcarvalho@dcc.ufmg.br - 2018054907

Abstract. *Digital libraries have a massive amount of accumulated items so that the use of an efficient sorting algorithm becomes critical to the organization of a collection. The present work compares the several sorting algorithms, particularly, seven variations of Quicksort, being: Classic quicksort, median of 3, first element, non-recursive, insertion with 1, 5 and 10%, in which they were submitted to 21 tests with different sizes and types of vectors. Through the analysis and study of the tests, the work also highlights the advantages and disadvantages, the best and worst cases, and recommendations for use.*

Resumo. *As bibliotecas digitais possuem uma quantidade massiva de itens acumulados, de modo com que o uso de um algoritmo de ordenação eficiente se torna fundamental para a organização de um acervo. O presente trabalho compara os diversos algoritmos de ordenação, estes, sete variações do Quicksort, sendo elas: O quicksort clássico, mediana de 3, primeiro elemento, não recursivo, inserção com 1, 5 e 10%, no qual foram submetidas à 21 testes com tamanhos e tipos de vetores diferentes. Através da análise e estudo dos testes, o trabalho também destaca as vantagens e desvantagens, os melhores e piores casos, e recomendações de uso.*

1. Introdução

O Quicksort é um algoritmo de ordenação que utiliza a técnica "dividir para conquistar", no qual ao escolher um pivô (que pode ser feito de várias maneiras) é dividido o vetor original em partições menores, onde à esquerda ficam os valores menores ou iguais ao pivô e à direita os valores maiores ou iguais ao pivô. O problema abordado é a implementação de sete variações deste algoritmo, listados a seguir:

1.1. Clássico

É o Quicksort utilizando como pivô o número central do vetor.

1.2. Mediana de três

Para se evitar divisões do vetor de forma degenerada (onde em um lado só fica um elemento e todo resto fica do outro) utiliza-se a escolha do pivô com a mediana de 3. É escolhido o valor mais a esquerda, o central e o mais à direita do vetor, e como pivô é utilizada a mediana desses três valores.

1.3. Primeiro elemento

Nesta variação, é utilizado como pivô o primeiro elemento do vetor.

1.4. Não recursivo

É o que possui a implementação mais diferente de todos os anteriores. Ao invés de utilizar a recursão para o particionamento dos vetores, aqui é criada uma pilha que irá realizar o trabalho de armazenar os limites do particionamento do vetor. No algoritmo, foi utilizado a partição do elemento central.

1.5. Inserção 1, 5 e 10%

Na implementação utilizada, foi usada a partição mediana de 3, o que o diferencia do Quicksort com Mediana de 3 é que utiliza o algoritmo de ordenação de inserção. Quando alguma partição for menor ou igual à porcentagem definida (1, 5 ou 10%), ao invés de continuar usando a recursão do Quicksort, é utilizado o inserção, que irá ordenar vetores consideravelmente pequenos.

1.6. Funcionamento

Foi analisado o tempo (em microsegundos), o número de comparações e o número de movimentações em três tipos de vetores diferentes (aleatório, ordem crescente e decrescente) possuindo valores entre 50.000 e 500.000 em intervalos de 50.000, testados 21 vezes cada para tirar sua mediana (tempo) e médias (comparações e movimentações): 630 vetores (21 testes x 10 tamanhos x 3 tipos de vetores) para cada variação de Quicksort.

2. Implementação

Todo o programa foi desenvolvido na linguagem C, utilizando o compilador gcc.

2.1. Principais funções

As principais funções do Quicksort são as funções de partições, onde há 3 variações que definem pivôs diferentes.

ParticaoCentral irá particionar o vetor através do vetor central, ficando à esquerda os valores menores ou iguais ao pivô e à direita os valores maiores ou iguais ao pivô. É utilizada pela função OrdenaClas e pelo QuickNR.

ParticaoM3 irá particionar o vetor utilizando o pivô que será a mediana de 3 valores: O mais à esquerda, o central e o mais à direita. Como explicado anteriormente, isso é utilizado para minimizar os casos onde as partições são degeneradas. É utilizada pela função OrdenaM3 e OrdenaI.

ParticaoPE irá particionar o vetor utilizando o pivô como o primeiro elemento do vetor (mais à esquerda). É utilizada pela função OrdenaPE.

Além disso, todas essas funções são chamadas pelas funções de Ordena, possuindo 4 variações. Essas funções são as responsáveis pelas chamadas recursivas do Quicksort.

OrdenaCentral, OrdenaM3 e OrdenaPE são bem similares, diferenciando-se

apenas na chamada da função de partição. Essas funções irão testar se o vetor ainda possui mais de 1 elemento em ambos os lados após ser particionado, caso for verdadeiro para algum dos lados, irá finalizar a execução das chamadas recursivas do lado correspondente, se não, irá se chama recursivamente.

Já a função OrdenaI recebe um parâmetro a mais, que é a porcentagem para testar se irá continuar a ordenação pela função Inserção. Então, antes de particionar e testar se há mais de 1 elemento, será avaliado se o número de elementos naquela chamada é menor do que a porcentagem (0.01, 0.05, 0.1) vezes o número de elementos totais, caso for verdadeiro, irá prosseguir pelo Inserção.

As funções QuickClas, QuickM3, QuickI1, QuickI5, QuickI10 irão apenas chamar a função de Ordena correspondente, realizando o que foi explicado anteriormente. Entretanto, o QuickNR não utiliza a recursão, fazendo-se necessário o uso de uma Pilha para armazenar todas as partições realizadas.

Por fim, a função teste_argumentos chamada no main, irá receber os argumentos inseridos no terminal, onde irá alocar memória para um vetor e para uma matriz que será utilizada caso o parâmetro "-p" for utilizado. Irá entrar no for, rodando 21 vezes (#define num_teste 21) criando o vetor definido no argumento (Ale, OrdC, OrdD) e após isso, é ordenado pela variação de Quicksort escolhido recolhendo todos os números de comparações e números de movimentações (que será tirada a média depois) e armazenando os tempos no vetor tempos_vetor (que será ordenado e tirado sua mediana). Caso o parâmetro "-p" for utilizado, irá chamar a função imprime_vetores imprimindo todos os vetores armazenados na matriz.

2.2. Algumas decisões

Para o número de movimentações, defini que atribuições envolvendo valores do vetor seria equivalente a 1 movimentação (incluindo auxiliares), então, uma troca são 3 movimentações.

No QuickM3, que utiliza a função Mediana3, foi considerado algumas comparações para se achar a mediana: Caso if for verdadeiro, são contabilizadas duas comparações, caso contrário (if else ou else), fará mais duas comparações além do if, contabilizando quatro comparações.

3. Instruções de compilação, execução e formato de entrada/saída

A compilação é feita através de um arquivo Makefile, já a execução é realizada das seguintes formas:

./exe <tipo_quick> <tipo_vetor> <num_itens> <-p [opcional]> , neste caso, a saída será mostrada pelo próprio terminal.

./exe <tipo_quick> <tipo_vetor> <num_itens> <-p [opcional]> >arq.txt/out, que irá imprimir a saída em um arquivo de texto.

O tipo_quick irá aceitar apenas QC, QM3, QPE, QI1, QI5, QI10 e QNR. Caso contrário, irá aparecer uma mensagem informando que o tipo de vetor é inválido.

O tipo_vetor irá aceitar apenas Ale, OrdC e OrdD. Caso contrário, irá aparecer uma mensagem informando que a variação de Quicksort é inválida.

O num_itens foi definida como os valores de 50.000 a 500.000 em intervalos de 50.000. Porém, ele irá aceitar qualquer valor, podendo ocasionar estouro de memória para valores MUITO grandes.

Se o número de argumentos for diferente de 4 ou 5, irá aparecer uma mensagem informando que a quantidade de argumentos é inválida.

Formato de saída:

<tipo_quick> <tipo_vetor> <num_itens> <n_comp> <n_mov> <tempo>

4. Análise experimental

A análise foi feita utilizando gráficos para os tipos de vetores (aleatório, crescente e decrescente) em questão de tempo x tamanho, comparações x tamanho e movimentações x tamanho, que em alguns casos a sobreposição das retas que possuem valores muito próximos ou iguais dificultaram a visualização, mas foi possível ter uma visão melhor de cada situação, podendo então analisar os piores e melhores casos.

Os testes consistiram em escolher uma variação de Quicksort e um tipo de vetor onde, após o vetor ser criado, era armazenado o tempo do início antes de entrar na função do Quicksort e então o vetor era enviado para o Quicksort escolhido contendo também uma variável responsável pelos números de comparações e uma variável para os números de movimentos. Então o vetor era ordenado e imediatamente após sair desta função era armazenado o tempo final que seria utilizado para achar a duração em microssegundos da execução da função. Para cada tamanho, em cada um dos tipos de vetores de uma variação do Quicksort, foi realizado 21 testes. No total foram 4410 testes (21 testes x 3 tipos de vetores x 10 tamanhos x 7 variações de Quicksort).

4.1. Especificações básicas do computador

O computador utilizado possui 8GB de RAM com processador Intel Core i5-8300H CPU @ 2.30GHz × 8 rodando o sistema Ubuntu 18.04.2 LTS 64 bits.

4.2. Vetores Aleatórios

4.2.1. Tempo

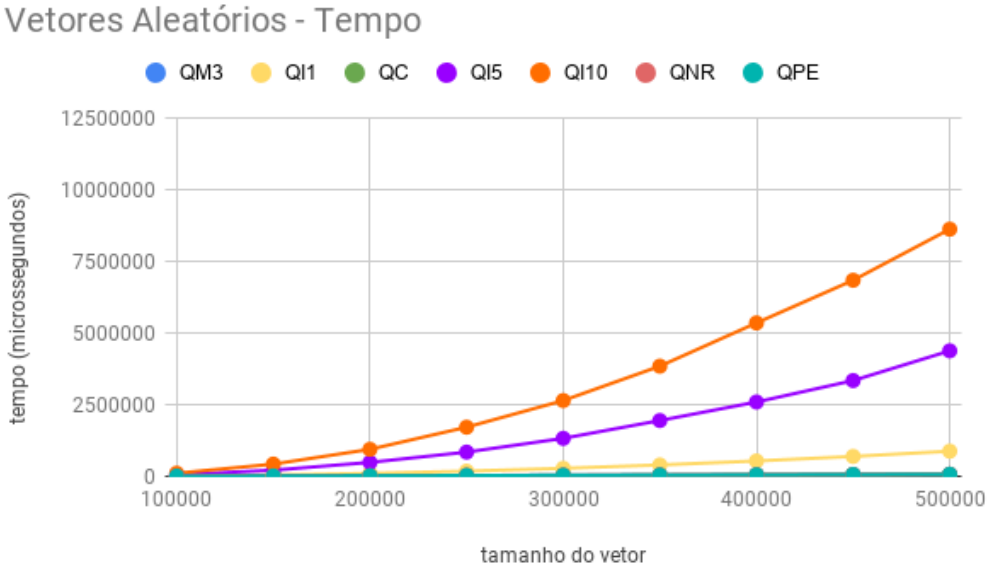

Gráfico 1. Gráfico de vetores aleatórios (tempo)

Analisando o gráfico gerado para os vetores aleatórios em questão de tempo, a pior variação do Quicksort é o Quicksort com Inserção 10%, que possui um crescimento muito grande cada vez que o vetor aumenta de tamanho.

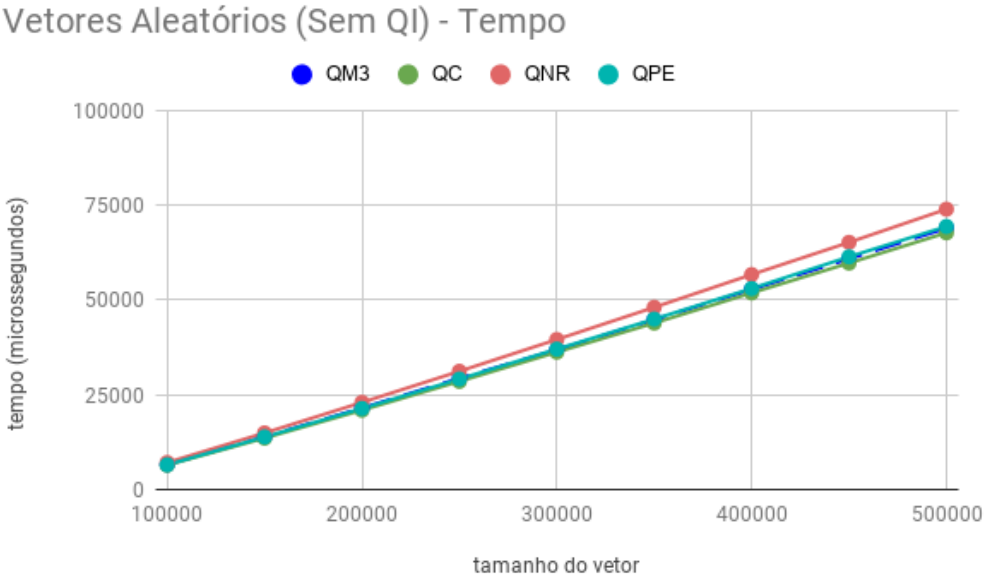

Gráfico 2. Gráfico de vetores aleatórios sem QI (tempo)

Como o crescimento dos Quicksort com Inserção são muito grandes, foi gerado um novo gráfico para obter melhor visualização do Quicksort.

É possível observar que Quicksort Mediana de 3, Quicksort Clássico e Quicksort Primeiro elemento estão bem próximos no gráfico, porém ainda sim o Quicksort Clássico está mais abaixo, tendo um resultado melhor.

4.2.2. Comparação

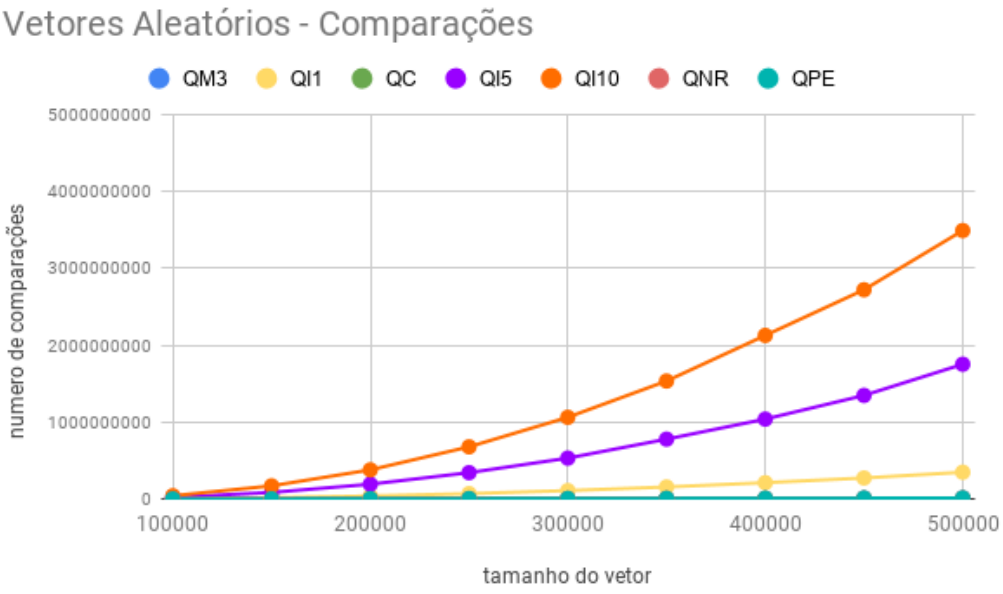

Gráfico 3. Gráfico de vetores aleatórios (comparação)

Em relação às comparações, Quicksort com Inserção 10% continua sendo o menos eficiente, tendo um crescimento muito alto.

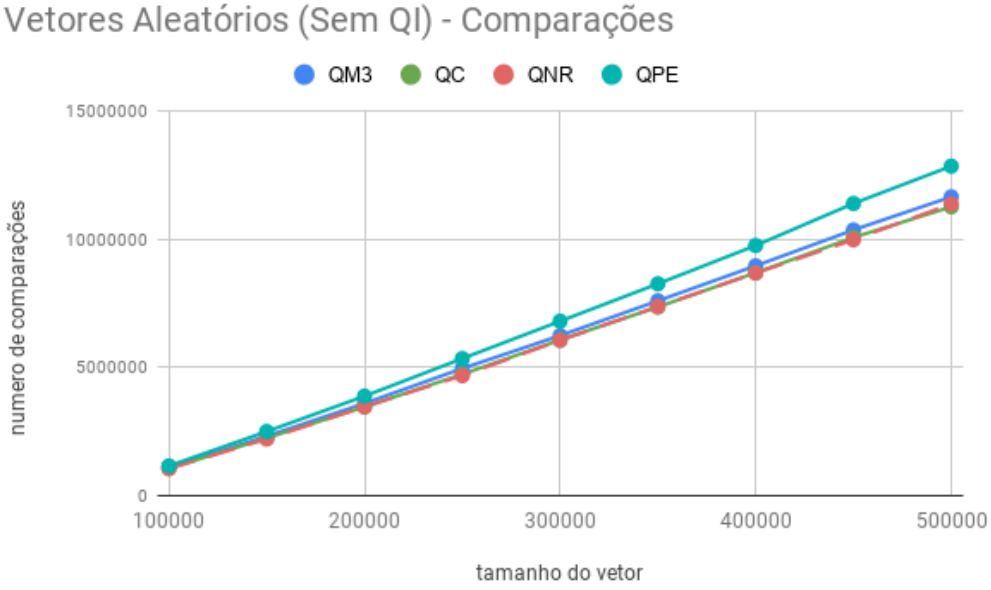

Gráfico 4. Gráfico de vetores aleatórios sem QI (comparação)

Tirando os Quicksort com Inserção, podemos analisar o melhor caso em relação às comparações. Dessa vez, tanto quanto Quicksort Clássico como Quicksort Não recursivo possuem o melhor desempenho em relação às comparações

4.2.3. Movimentação

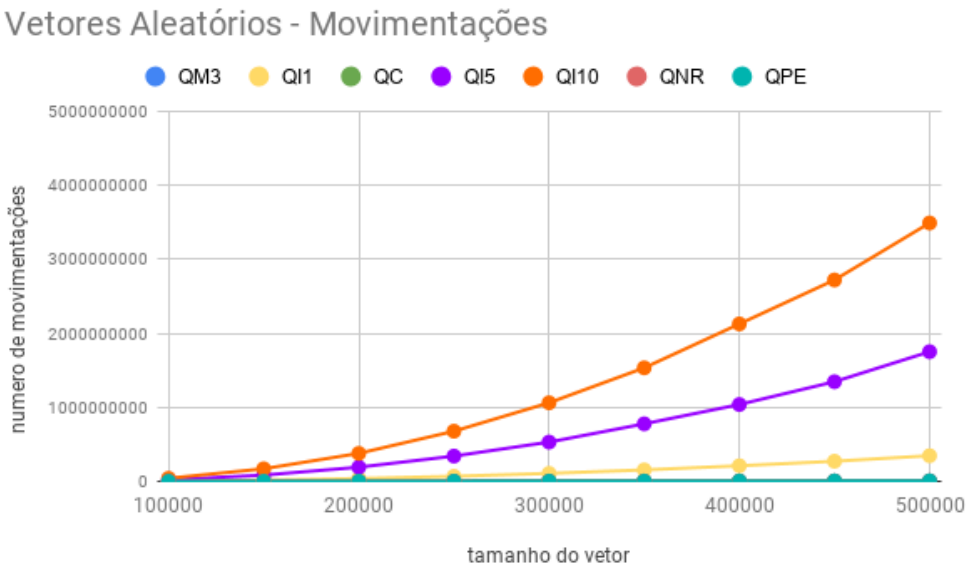

Gráfico 5. Gráfico de vetores aleatórios (movimentação)

Para as movimentações, Quicksort com Inserção 10% é o pior caso do vetor aleatório.

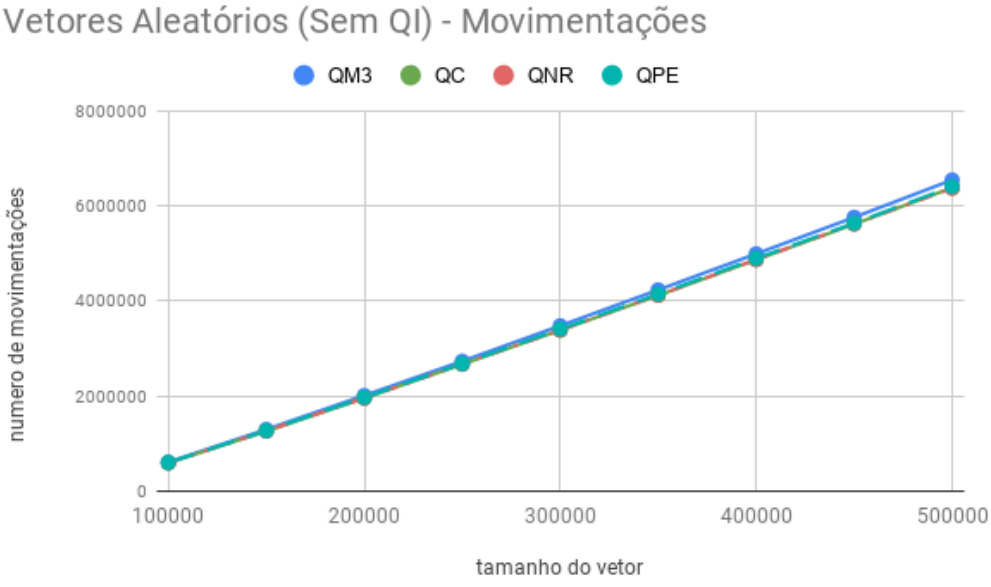

Gráfico 6. Gráfico de vetores aleatórios sem QI (movimentação)

Já para o melhor caso nas movimentações dos vetores aleatórios, todos os 4 possuem desempenho muito similar, porém Quicksort Mediana de 3 está um pouco acima, sendo então Quicksort Clássico, Não recursivo e Primeiro elemento os melhores em relação a movimentação.

4.3. Vetores Crescentes

4.3.1. Tempo

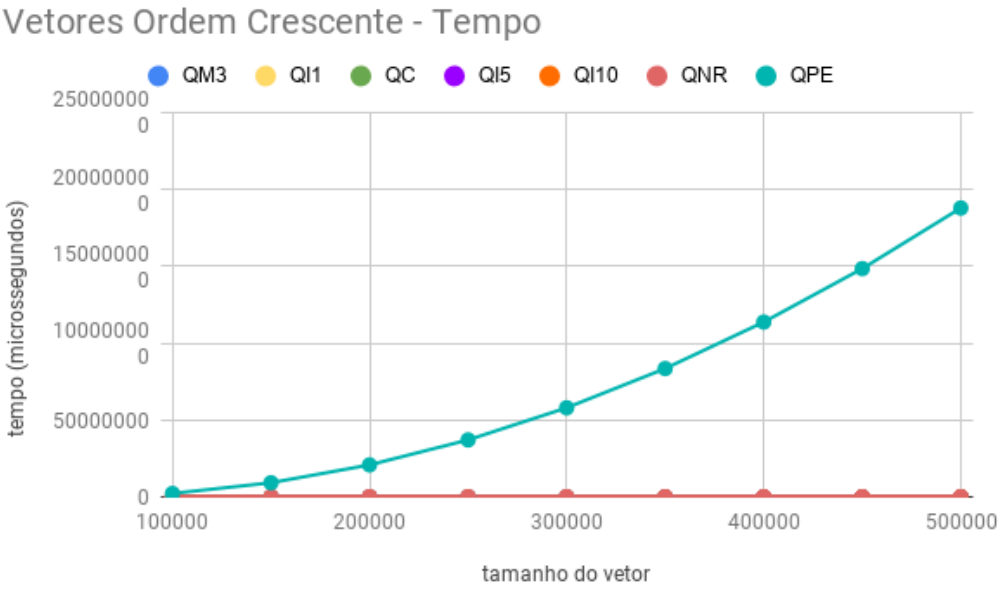

Gráfico 7. Gráfico de vetores em ordem crescente (tempo)

Em relação ao tempo, Quicksort Primeiro elemento em vetores crescentes é um dos piores caso do Quicksort, pois toda chamada recursiva resultará em uma partição degenerada, possuindo apenas 1 elemento de um lado e todo o resto do outro, e por esse motivo sua complexidade é $O(n^2)$.

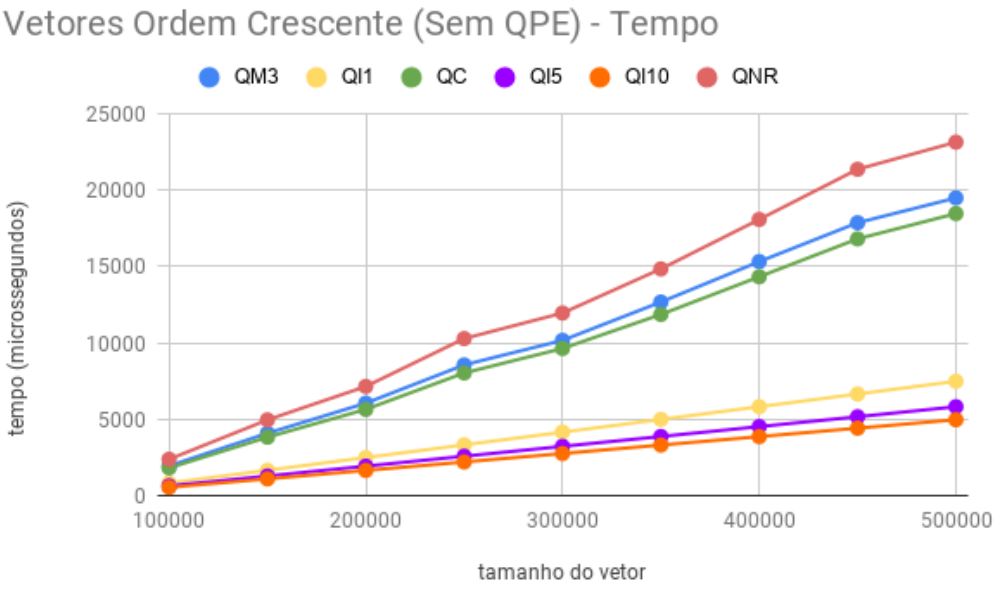

Gráfico 8. Gráfico de vetores em ordem crescente sem QPE (tempo)

O melhor caso nesse tipo de vetor é o do Quicksort com Inserção 10%, pois em vetores ordenados de forma crescente o inserção possui complexidade O(n).

4.3.2. Comparação

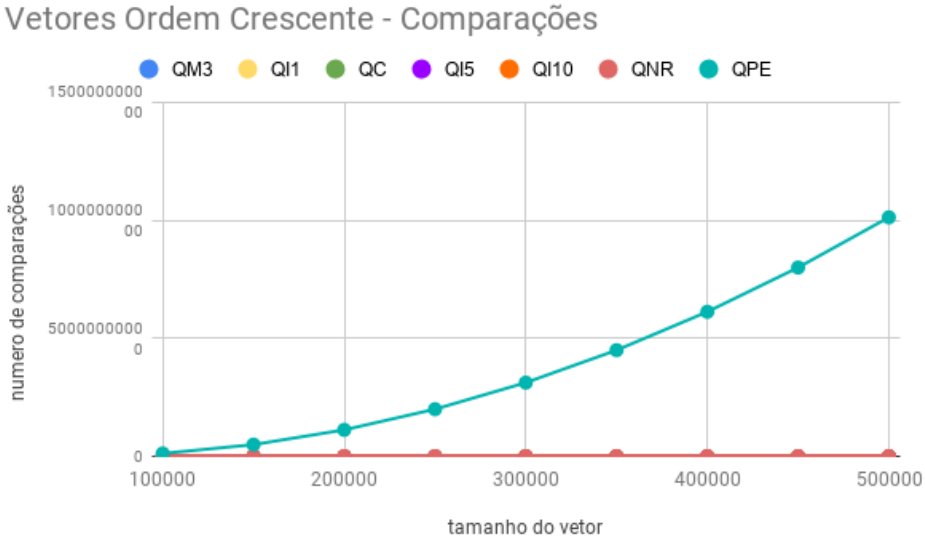

Gráfico 9. Gráfico de vetores em ordem crescente (comparação)

Em questão de comparação, o Quicksort Primeiro elemento também é o pior caso em vetores crescentes.

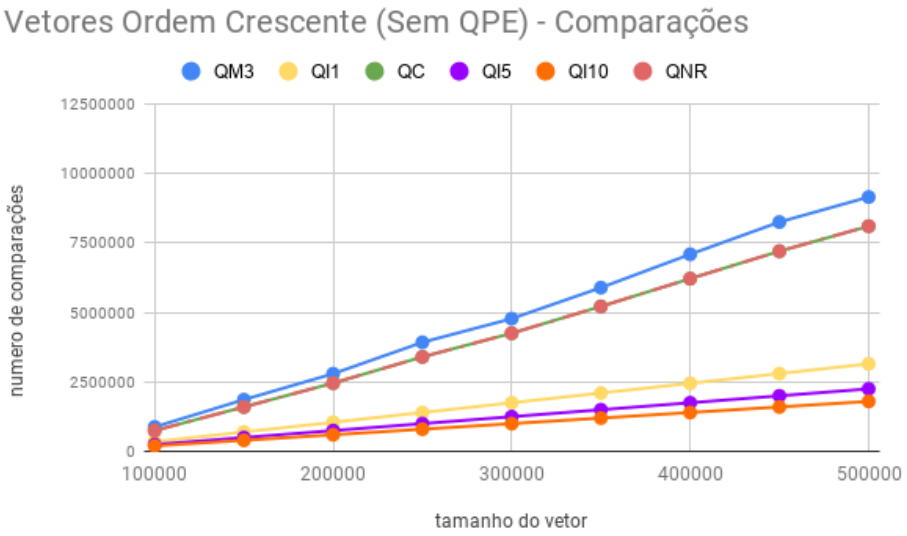

Gráfico 10. Gráfico de vetores em ordem crescente sem QPE (comparação)

O melhor caso em comparações é o Quicksort com Inserção 10% novamente.

4.3.3. Movimentação

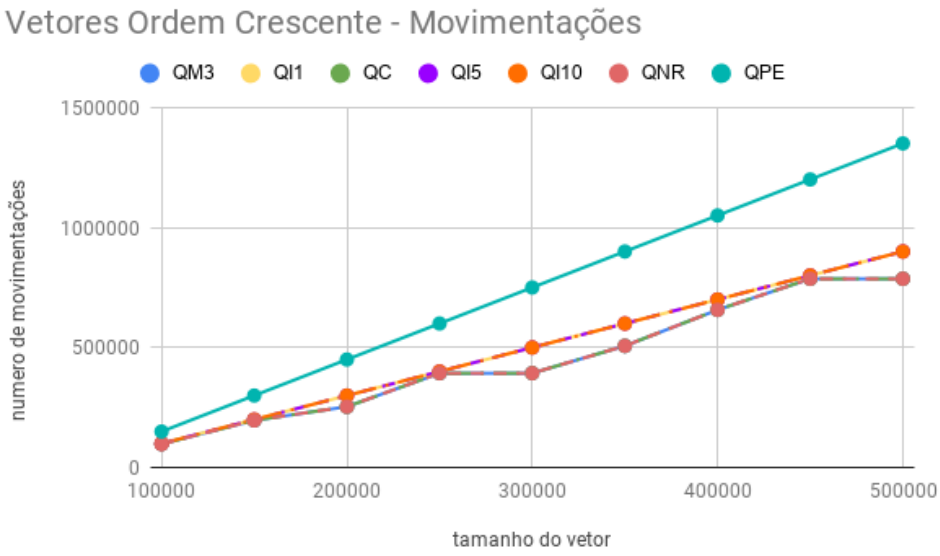

Gráfico 11. Gráfico de vetores em ordem crescente (movimentação)

Já em questão de movimentação, o pior caso é novamente o Quicksort Primeiro elemento. Já os melhores são o Quicksort Clássico, o Quicksort Não recursivo e o Quicksort Mediana de 3, que possuem um crescimento parecido com os Quicksort com Inserção, porém em alguns momentos apresentam um crescimento menor.

4.4. Vetores Decrescentes

4.4.1. Tempo

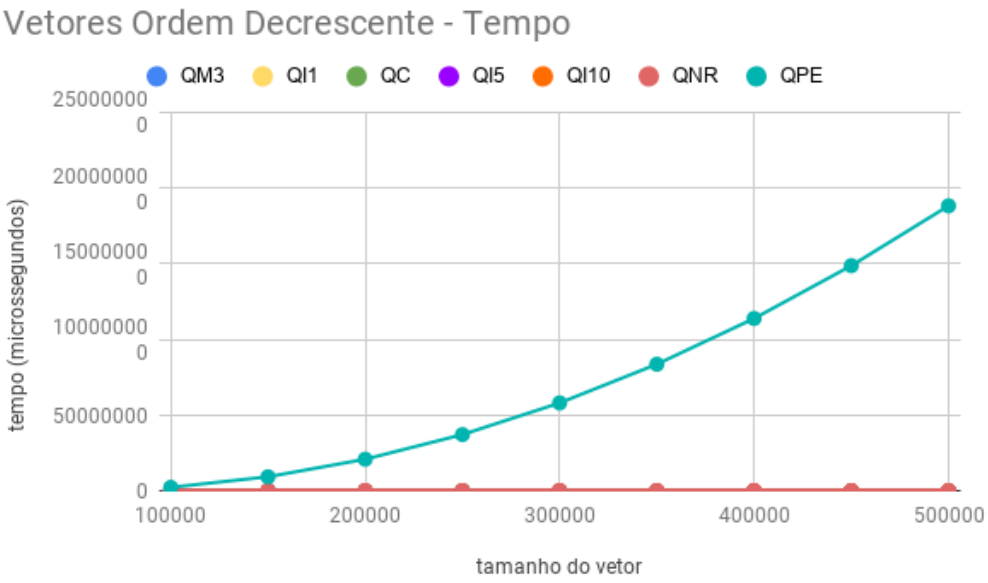

Gráfico 12. Gráfico de vetores em ordem decrescente (tempo)

Este é novamente o pior caso do Quicksort Primeiro elemento, que possui o tempo um pouco pior do que nos vetores crescente, pois além de gerar as partições degeneradas, precisa fazer mais trocas.

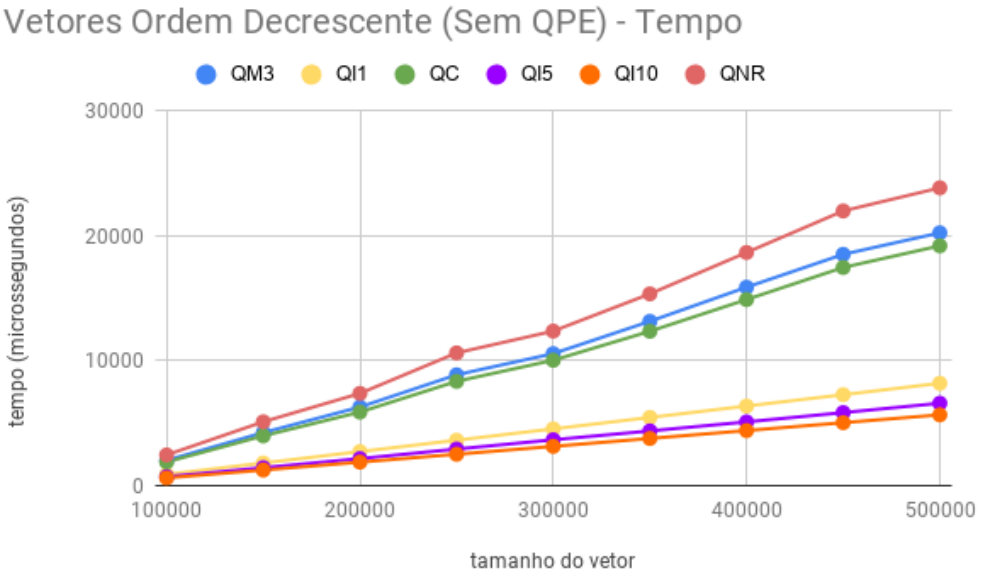

Gráfico 13. Gráfico de vetores em ordem decrescente sem QPE (tempo)

Já o melhor caso, é o Quicksort com Inserção 10%.

4.4.2. Comparação

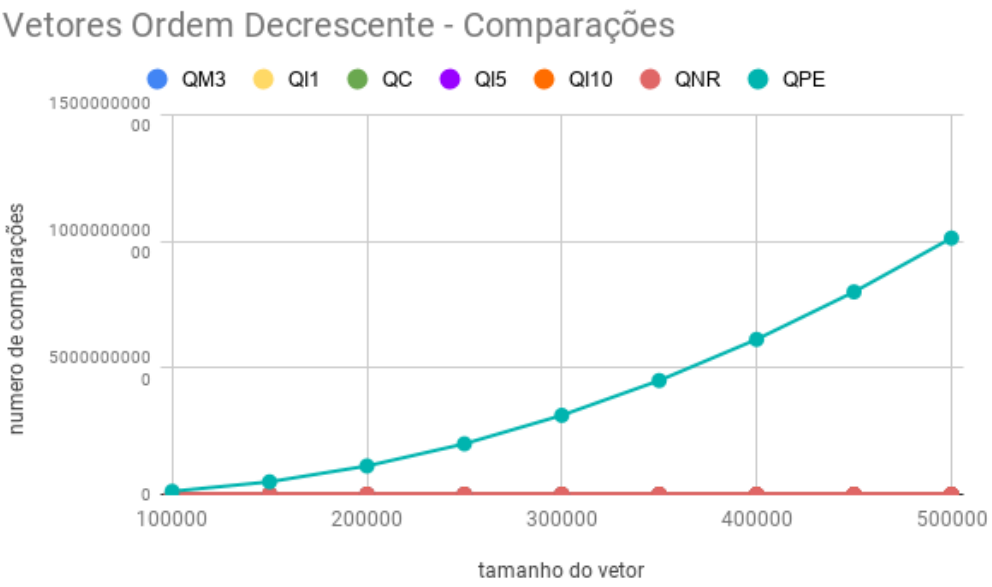

Gráfico 14. Gráfico de vetores em ordem decrescente (comparações)

Em comparação, Quicksort Primeiro elemento também é o pior caso.

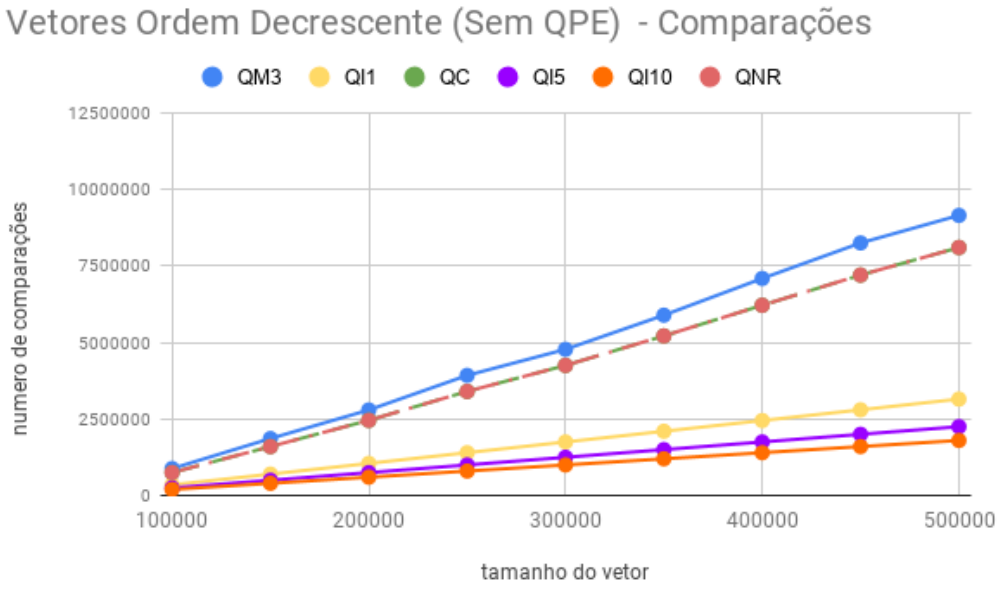

Gráfico 15. Gráfico de vetores em ordem decrescente sem QPE (comparação)

E novamente o Quicksort com Inserção 10% é o melhor caso.

4.4.3. Movimentação

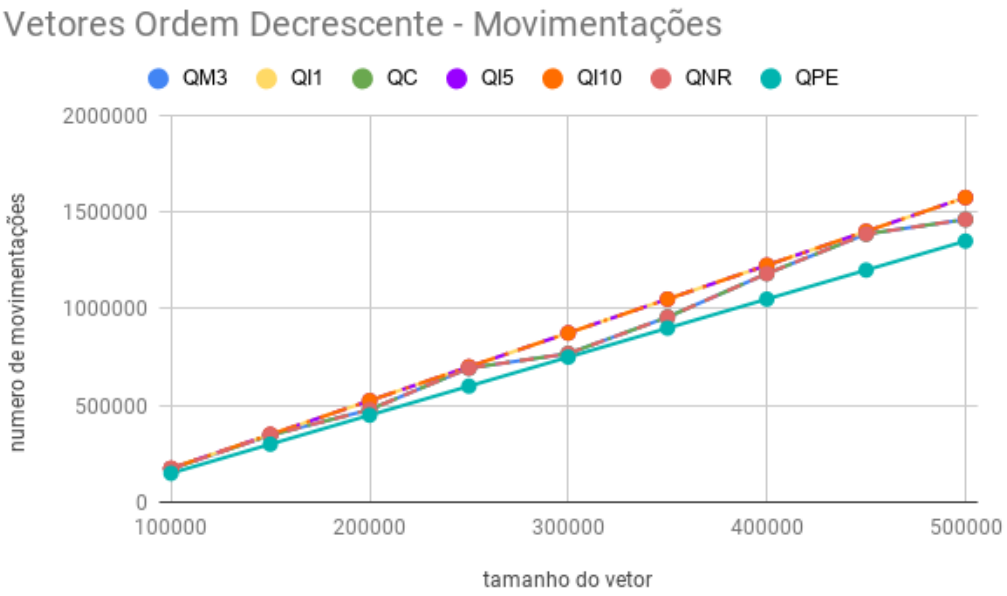

Gráfico 16. Gráfico de vetores em ordem decrescente (movimentação)

E por fim, na movimentação com vetores decrescentes, todos os Quicksort com Inserção (seja, 1, 5 ou 10%) possuem o pior caso. Já o melhor caso dessa vez é do Quicksort Primeiro elemento.

4.5. Resultado

Com os resultados obtidos na análise de todos os gráficos foi possivel gerar a tabela abaixo:

Tipo do vetor	Tempo		Comparação		Movimentação	
	Melhor	Pior	Melhores	Pior	Melhores	Piores
Aleatório	QC	QI10	QC, QNR	QI10	QC, QNR, QPE	QI10
Crescente	QI10	QPE	QI10	QPE	QC, QNR, QM3	QPE
Decrescente	QI10	QPE	QI10	QPE	QPE	QI1, QI5, QI10

5. Conclusão

Após a análise realizada anteriormente, algumas conclusões podem ser apontadas:

Para vetores que estão ordenados em Ordem Crescente ou em Ordem Decrescente, os algoritmos do Quicksort que utilizam o Inserção se saem muito melhor

se comparados aos outros, pois o Inserção em vetores ordenados possui complexidade O(n), tendo destacado o Quicksort com Inserção 10% no teste executado. O pior desempenho é o Quicksort Primeiro elemento pelo motivo de que neste caso apenas gera partições degeneradas tendo complexidade $O(n^2)$, devendo ser evitado.

Para vetores aleatórios, o Quicksort Clássico apresenta um melhor desempenho, tendo o menor tempo, menor número de comparações e o menor número de trocas. Os piores casos são os Quicksort com Inserção, que devem ser evitados neste tipo de vetor.

Já para casos em que não se sabe qual o tipo de vetor, o melhor e mais consistente é o próprio Quicksort Clássico, que apresenta bons resultados em todos os casos. O Quicksort Mediana de 3 também é recomendável, mesmo tendo um número de comparações um pouco maior, apresenta um bom tempo e além disso, este irá evitar o pior caso do Quicksort (partições degeneradas). E por último, o Quicksort Não recursivo também pode ser interessante mesmo apresentando um tempo um pouco maior comparado ao Quicksort Clássico por utilizar a pilha de modo iterativo, possui os mesmos resultados em comparações e movimentações, porém evitando o custo de várias chamadas recursivas.

Ao longo do trabalho, algumas dificuldades foram enfrentadas, como a memória da stack, que por padrão é definida como 8192 kbytes e causava Segment fault durante a execução do Quicksort Primeiro elemento no seu pior caso, que foi resolvido utilizando ulimit -s unlimited. Além disso, a execução teve uma duração extensa, executando por mais de 5 horas.

6. Bibliografia

ZIVIANI, N. Projeto de Algoritmos com Implementações em Pascal e C. 3ª Edição, Cengage Learning, 2011.

CASAVELLA, Eduardo. Argumentos na linha de comando. Disponível em: <http://linguagemc.com.br/argumentos-em-linha-de-comando/>. Acesso em: 27 de maio. 2019.

WIKIPEDIA. Insertion Sort. Disponível em: <https://en.wikipedia.org/wiki/Insertion_sort>. Acesso em: 29 de maio. 2019.