Trabalho Prático 2
Biblioteca Digital de Arendelle

Gabriela Peres Neme (2018054745)

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
gabrielaperesneme@gmail.com

1. Introdução

O objetivo do presente trabalho é analisar o funcionamento e eficiência do algoritmo de ordenação *Quicksort* e de suas variações, através de avaliação de métricas do número de comparações entre elementos do vetor, número de trocas de elementos (movimentações) e tempo de execução.

O algoritmo *Quicksort* clássico funciona através da seleção de um elemento pivô, escolhido na posição central do vetor que se deseja ordenar. Todos os elementos menores que o pivô devem ficar à esquerda do vetor e todos os elementos maiores que ele devem estar à direita. Tal resultado é obtido através da troca sucessiva de elementos menores que o pivô, da esquerda para a direita, com elementos maiores que ele, caminhando da direita para a esquerda. O índice do vetor que separa elementos maiores e menores que o pivô será também o ponto de quebra em dois novos vetores que deverão ser ordenados dessa mesma forma, de maneira recursiva. O algoritmo para quando o número de elementos de cada subvetor é 1.

Além do *Quicksort* clássico (QC), foram analisadas outras seis variações do algoritmo, sendo elas o *Quicksort* mediana de três (QM3), que seleciona o pivô usando a mediana entre os elementos nas posições inicial, central e final do vetor; o *Quicksort* primeiro elemento (QPE), que seleciona o pivô como sendo o primeiro elemento do vetor; o *Quicksort* inserção 1% (QI1), em que o processo de partição do algoritmo *Quicksort* é interrompido quando o subvetor tiver menos que 1% do tamanho do vetor original, a partir de quando o subvetor será organizado pelo método da inserção; o *Quicksort* inserção 5% (QI5), que consiste na mesma ideia do QI1, mas o processo de partição é interrompido quando o subvetor tem 5% do tamanho do vetor original; o *Quicksort* inserção 10% (QI10), que inicia o algoritmo de inserção quando o subvetor tem 10% do tamanho do vetor original; e o *Quicksort* não recursivo (QNR), que utiliza uma pilha para simular as chamadas recursivas do QC, sem alterar nada além disso, em relação ao QC.

A fim de computar as métricas e analisar tais variações do algoritmo *Quicksort*, serão contabilizadas, a cada execução do algoritmo, o número de comparações entre elementos do vetor, o número de trocas realizadas entre elementos do vetor (movimentações), e o tempo de execução. Ademais, com o intuito de evitar que o resultado da análise seja enviesado por execuções ruins, o programa realiza, a cada chamada, 50 vezes o algoritmo de ordenação, e retorna a média de comparações e de movimentações, e a mediana do tempo.

2. Implementação

2.1. Entrada de dados

A entrada de dados do programa deve conter três ou quatro parâmetros. O primeiro consiste no tipo de ordenação desejada (siglas QC, QM3, QPE, etc). O segundo parâmetro é o tipo de vetor que se deseja ordenar. Para vetores com elementos escolhidos aleatoriamente deve-se digitar "Ale". Para vetores crescentes, "OrdC" e para vetores decrescentes, "OrdD". O terceiro parâmetro é o número de elementos do vetor a ser ordenado. Tais vetores serão gerados internamente pelo próprio programa, segundo as especificações da entrada. Por fim, o último parâmetro, opcional, "-p" fará com que o programa imprima também os 50 vetores utilizados pelo algoritmo de ordenação. Para mais detalhes consultar tópico Instruções de compilação e execução.

2.2. Saída de dados

O resultado do programa deverá ser impresso na saída padrão e conterá os três primeiros parâmetros da entrada acrescidos das métricas obtidas, ou seja, a média de comparações, média de movimentações e mediana de tempo, nessa ordem.

2.3. Funcionamento das principais funções e procedimentos

A fim de modularizar o código e deixá-lo organizado, a função *main* está, sozinha, dentro da pasta *program*. Os *headers* (arquivos *.h) estão na pasta *include*, e os códigos fonte (arquivos *.cpp) estão na pasta *cpp*. As duas últimas pastas foram divididas em subpastas com temas específicos.

O procedimento *leEntrada*, como o nome sugere, lê os dados de entrada, inseridos pelo usuário na linha de comando, sendo eles o tipo de variação do *Quicksort* desejado, o tipo de ordenação inicial do vetor, o tamanho do mesmo e, opcionalmente, se os vetores serão impressos.

Em seguida é chamado o procedimento *ordenaVetores*, que inicia a sequência de etapas necessárias para ordenar os vetores e computar as métricas. Desta forma, esse procedimento itera 50 vezes pelos passos seguintes, e armazena em vetores as métricas obtidas em cada uma das 50 vezes que o algoritmo de ordenação for executado. Nesse sentido, a cada iteração, é criado e preenchido o vetor de tamanho n, através do procedimento *preencheVetorNumeros* e, em seguida, o vetor recém-criado é salvo em uma *string*, para que possa ser impresso posteriormente, caso o usuário tenha solicitado (procedimento *concatenaVetores*). Posteriormente, é chamado o procedimento *QuickSort* para que esse ordene o vetor criado.

O procedimento *QuickSort* é responsável pela identificação do tipo de *Quicksort* escolhido pelo usuário (QC, QM3, QPE, etc), e pela apuração do tempo levado para a ordenação. Ao identificar o tipo do *Quicksort*, é chamado o algoritmo correspondente.

O procedimento *QC* é composto por três funções básicas, *QC_Ordena*, *QC_Partição* e *iteraçãoDeTrocas*, responsáveis pela ordenação do vetor e pela contabilização do número de comparações e de movimentações. A função *iteraçãoDeTrocas* também é utilizada pelos métodos de ordenação QM3, QPE, QI1, QI5 e QI10.

Os procedimentos *QM3* e *QPE* funcionam da mesma maneira que o *QC*, mas apuram o pivô de maneira diferente. Para a execução do QM3 há uma função que calcula a mediana de três números, *QM3_mediana*.

Os métodos de ordenação QI1, QI5 e QI10 utilizam o mesmo procedimento, *QI*, cuja variação é o limite a partir do qual será utilizado o método da inserção para ordenar os vetores.

O algoritmo de ordenação por inserção, por sua vez, foi criado no procedimento *Insercao*, e funciona como uma ordenação de cartas para um jogador de baralho, de forma que o vetor de números é percorrido da esquerda para a direita e, a cada iteração *i* aloca-se o elemento que ocupava a iésima posição em ordem crescente do vetor, já ordenado, à esquerda da posição *i*.

Por fim, o procedimento *QNR* não compartilha nenhuma função com os métodos anteriores, dependendo exclusivamente da estrutura de dado Pilha, implementada através do objeto Pilha, localizada na pasta e arquivo de mesmo nome.

Ao final das 50 ordenações de vetores, o procedimento *OrdenaVetores* apura as médias e a mediana através das funções respectivas, alocadas no arquivo *estatística*.

Após a obtenção das métricas, a função *main* chama o procedimento *imprimeSaida*, para que esse imprima na saída padrão os valores obtidos, conforme especificado no item anterior.

2.4. Decisões

Uma decisão importante que precisou ser tomada durante a criação do programa foi o *trade off* de tempo de execução e modularização das variações do *Quicksort*.

Para que o programa fosse inteiramente modularizado e para que não houvesse repetição de código em relação aos algoritmos de ordenação, seria necessário avaliar o tipo de *Quicksort* escolhido pelo usuário a cada chamada da função Partição. Como foram avaliados vetores com até 500 mil elementos, e a comparação de *strings* é uma operação cara para o computador, optou-se por fazê-la apenas uma vez por vetor. Assim, a métrica de tempo não foi afetada, em nenhum caso, pela comparação de *strings*, e ficou restrita às operações relevantes para a ordenação do vetor.

De fato, observou-se melhora no tempo de execução quando não havia comparações de *strings* a cada chamada da função Partição. Em contrapartida, existem pequenos trechos de código semelhantes nas variações QC, QM3 e QPE que, mesmo assim, foram devidamente modularizados, respeitando as especificações do parágrafo anterior.

Quanto à contabilização das métricas, a troca de elementos de vetor foi contabilizada como apenas uma movimentação, conforme instrução proferida no fórum de dúvidas da disciplina. *Data maxima venia*, acredita-se que seria mais adequado que a troca de elementos fosse contabilizada como três movimentações, especialmente pela existência de movimentações no método de inserção em que não são feitas trocas, mas deslocamento de valores, que deveriam ser contabilizados como uma movimentação. No entanto, não cabe aos alunos a discricionariedade sobre instruções dadas pela

especificação do presente trabalho, motivo pelo qual foi adotada a métrica de uma movimentação para trocas e deslocamentos.

Por fim, qualquer comparação entre elementos de vetores foi contabilizada, incluindo as comparações existentes no método de ordenação por inserção e a escolha da mediana de três elementos, para a variação QM3.

2.5. Compilador utilizado

Foi utilizado o compilador g++ versão 7.3.0 (Ubuntu 7.3.0-27ubuntu~18.04).

3. Instruções de compilação e execução

A compilação do programa é feita através do comando *make* na linha de comandos, que cria o arquivo compilado de cada módulo do programa (cada arquivo *cpp*) na pasta *build*, no formato *.o, e o *main* é criado na pasta *bin*.

Portanto, para executar o programa é necessário rodar o arquivo compilado, intitulado *tp2*, dentro da pasta *bin*, seguido dos parâmetros de entradas estipulados no item 2.1. Entrada de dados, resultando no formato a seguir.

```
./bin/tp2 <variaçãoQS> <tipoVetor> <tamVetor> [-p]
```

Exemplos da forma de execução do programa foram apresentados a seguir.

```
./bin/tp2 QC Ale 500
./bin/tp2 QI5 OrdC 450000
./bin/tp2 QNR OrdD 30000 -p
```

4. Análise experimental

4.1. Metodologia

A fim de analisar o funcionamento e eficiência do algoritmo de ordenação *Quicksort* e de suas variações, utilizou-se o programa descrito nos tópicos anteriores para computar o número médio de comparações entre elementos do vetor, número médio de movimentações entre elementos e a mediana do tempo de execução.

Tal programa foi executado para vetores de tamanho 50.000 a 500.000, em intervalos de 50.000 elementos, para cada uma das sete variações do *Quicksort* analisadas no presente trabalho e para cada uma das três formas de ordenação inicial dos vetores, totalizando 210 testes no total.

Cada um dos 210 testes realizados retorna o número médio de comparações e de movimentações, bem como a mediana do tempo de execução de 50 ordenações. Desta forma, a presente análise experimental abarca 10.500 ordenações de vetores.

4.2. Especificações do computador utilizado

O computador utilizado para realizar os testes foi um notebook Dell Inspiron 14, Série 7000, i14-7460. O processador é um Core i5 7ª geração da Intel e o computador possui 8GB de memória.

4.3. Análise dos gráficos

A partir de um comando *bash*, todos os 210 testes foram impressos e salvos em arquivos específicos, separados por tipo do algoritmo *Quicksort* escolhido. Tais arquivos foram salvos na pasta *out* e possuem extensão de mesmo nome.

A fim de facilitar a interpretação dos dados, todos os arquivos *.out gerados foram convertidos em um único *csv*, a partir do qual foi possível importar as métricas para um *Notebook Jupyter* e, através da biblioteca de *Pandas*, gerar gráficos dos resultados do programa. As figuras que resultaram de tais gráficos foram apresentadas a seguir.

Para cada tipo de vetor foram demonstradas as métricas de comparações, movimentações e tempo, de todos os tipos de *Quicksort*, com o intuito de facilitar o cotejo dos mesmos. Ademais, caso tenha havido dificuldade de visualizar a diferença entre variações do *Quicksort* em razão da escala exigida por uma ou mais variação, ou por eventual sobreposição, serão exibidos dois gráficos, de maneira que seja possível visualizar todas as variações no gráfico da esquerda e, à direita, será demonstrado um subconjunto de tipos de *Quicksort* cuja visualização ficou prejudicada no gráfico da esquerda.

4.3.1. Vetores aleatoriamente gerados

A figura 1 mostra o número de comparações para vetores aleatórios. Nela é evidente que, ao utilizar o método de inserção a partir de um determinado tamanho de vetor, nos aproximamos de uma função quadrática, que cresce à medida que a porcentagem do vetor ordenado pelo método de inserção cresce. Ou seja, foram necessárias mais comparações para a variação QI10 que para a QI1, uma vez que o tamanho dos vetores ordenados pelo método da inserção na variação QI10 foi maior que na QI1.

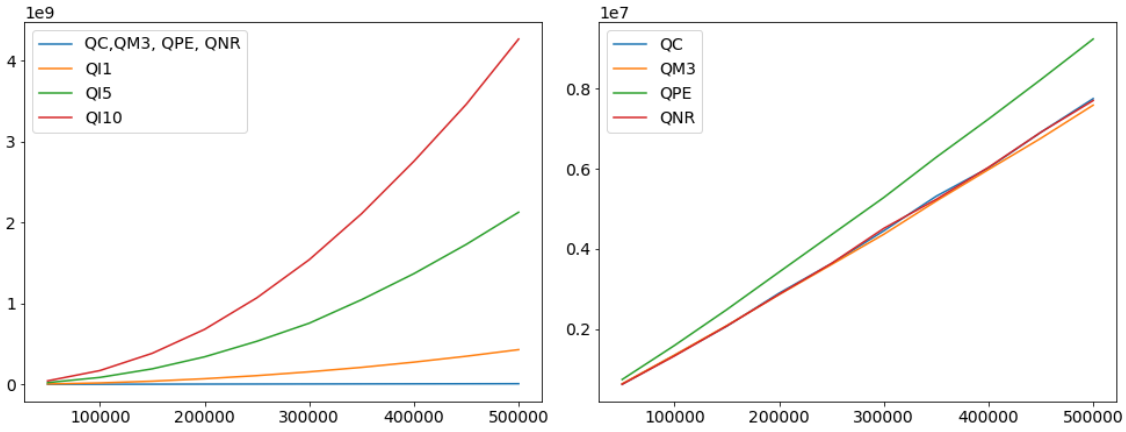

Figura 1 - Número de comparações para vetores aleatórios

Em relação à parte direita da figura 1, observa-se que o QPE teve um número de comparações um pouco superior às demais. Tal fato pode ser explicado pela maior probabilidade de criação de partições degeneradas ao escolher o primeiro elemento do vetor como o pivô, uma vez que é característica do algoritmo colocar os elementos menores à esquerda do vetor, e os maiores à direita.

Apesar de o método QM3 adicionar 2 ou 3 comparações a cada processo de partição, esse aumento foi compensado pela menor probabilidade de ocorrência de

partições degeneradas ao escolher a mediana de três números e, assim, o número de comparações desse método foi um pouco menor que a variação QC. De qualquer forma, não se pode dizer que houve diferença significativa entre QC, QM3 e QNR.

Em relação ao número de movimentações necessários para ordenar um vetor aleatório, cujos gráficos foram exibidos na figura 2, pode-se afirmar o mesmo em relação às variações que usam o método da inserção, ou seja, houve piora significativa no desempenho, uma vez que é possível observar uma tendência à ordem de complexidade quadrática.

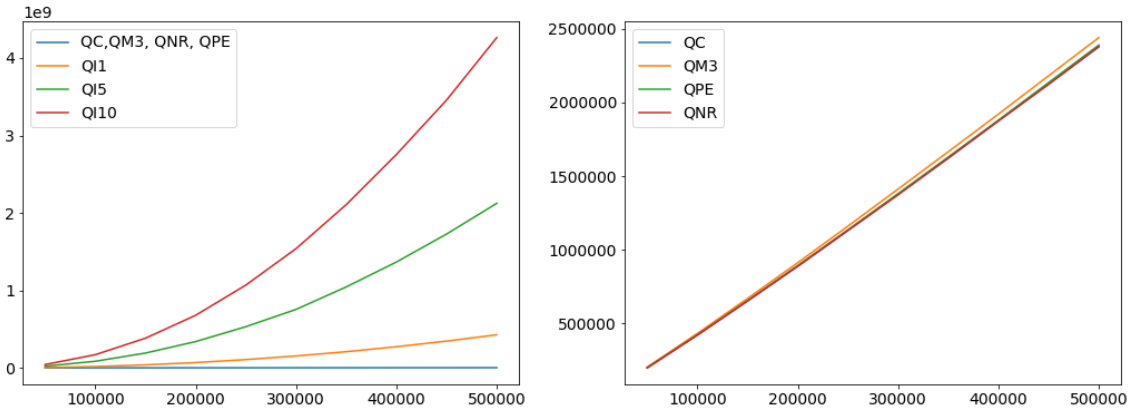

Figura 2 - Número de movimentações para vetores aleatórios

No que diz respeito às variações que não utilizam a ordenação por inserção, tiveram desempenho que tenderam para uma reta, de forma que o aumento do número de movimentações cresceu de maneira linearmente proporcional ao tamanho do vetor ordenado. E essas variações não apresentaram diferença significativa quanto ao número de movimentações entre si.

De maneira análoga ao que pôde ser observado nas análises de movimentações e comparações, o tempo também foi significativamente pior para as variações que utilizam o método de inserção, conforme demonstrado pela figura 3, cuja metade esquerda é extremamente semelhante às figuras 1 e 2. Desta forma, a explicação para o comportamento é a mesma.

Em contrapartida, enquanto o número de comparações e movimentações para os métodos QC, QM3, QPE e QNR tenham sido muito semelhantes, o tempo entre eles apresentou comportamento diferente, tendo sido o QC o mais rápido.

Tal variação pode ser explicada pela maior probabilidade de partições degeneradas no método QPE, pelo tempo adicional de apurar a mediana no método QM3, e pela maior velocidade do compilador, processador ou computador em lidar com chamadas recursivas, no método QNR, de forma que a pilha gerada internamente para as chamadas recursivas é mais eficiente que a implementada no presente trabalho, o que resultou em um tempo de execução maior para o método QNR, que utiliza o objeto Pilha criado nesse trabalho, para esse fim.

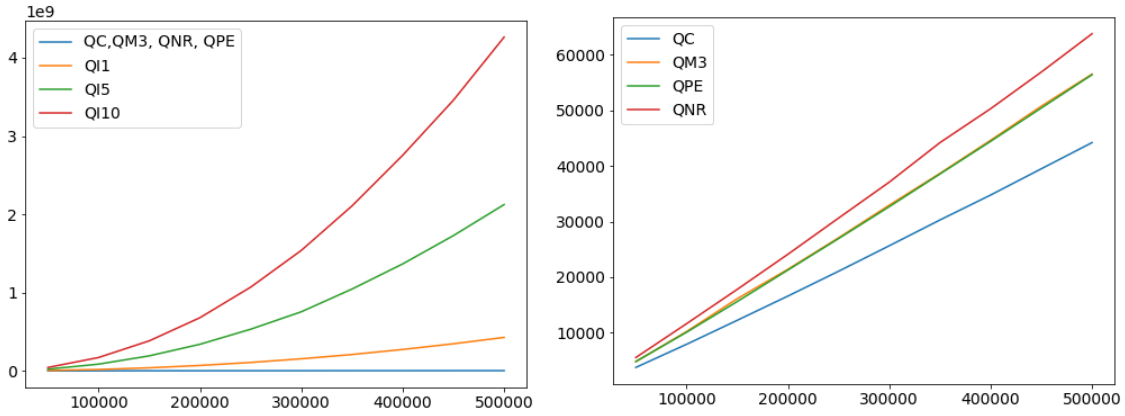

Figura 3 - Tempo de execução para vetores aleatórios

Desta forma, conforme demonstrado através das figuras 1, 2 e 3, o método mais eficiente para vetores aleatórios é o *Quicksort* clássico, que teve o menor tempo mediano de execução e os menores números de comparação e movimentações. Além disso, não é interessante ordenar nenhum trecho do vetor pelo método da inserção para vetores aleatórios.

Vale observar que o tempo de execução foi medido através da mediana, motivo pelo qual é muito provável que o método QM3 apresente um tempo significativamente menor para os piores casos, quando comparado ao QC, QPE e QNR, exatamente pela menor probabilidade de partições degeneradas.

4.3.2. Vetores ordenados de forma crescente

A figura 4 mostra o número de comparações para uma ordenação de vetores inicialmente ordenados. Conforme observado nessa figura, o número de comparações da variação QPE foi significativamente maior que os demais, tendo atingido, inclusive, a ordem de complexidade quadrática no presente caso.

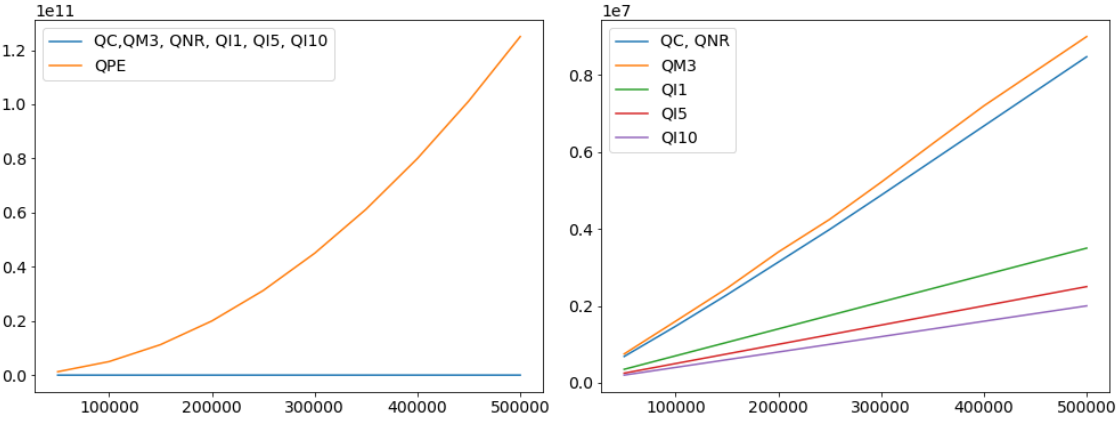

Figura 4 - Número de comparações para vetores ordenados

Essa diferença pode ser explicada pelo fato de que o método QPE, ao escolher o primeiro elemento do vetor como pivô, sempre realiza partições degeneradas no método *Quicksort* para vetores ordenados, o que ocasiona um número de comparações quadrático, ao invés do n.log(n). Ou seja, cada uma das n operações de partição irá comparar n elementos.

Os métodos QC e QNR foram idênticos em relação ao número de comparações, pois executam exatamente as mesmas operações, quando submetidos a vetores idênticos. Já a QM3 executou mais comparações em razão da necessidade do cômputo da mediana a cada iteração de partição.

Para as variações que utilizam o método da inserção, dessa vez, o número de comparações foi significativamente menor, e decrescente à medida que o tamanho do vetor ordenado pelo método da inserção cresce. Isso acontece porque, para vetores já ordenados, o método da inserção compara n-1 elementos, enquanto o *Quicksort* compara n.log(n). Tal fato ocorre porque o método da inserção compara o i-ésimo elemento apenas com o elemento i-1, caso i-1 seja menor que i. Como isso acontece para todos os i números do vetor ordenado, são feitas apenas n-1 comparações. Já o critério de parada do *Quicksort* é chegar a um vetor de tamanho 1, e isso não será minimizado no caso em que o vetor já está ordenado. Em razão disso, quanto maior a porcentagem do vetor ordenado pelo método da inserção, menor o número de comparações executadas.

A figura 5 mostra o número de movimentações para vetores ordenados.

No gráfico à esquerda, nota-se que, mais uma vez, o QPE apresentou o pior desempenho. Dessa vez, no entanto, teve um crescimento linear de movimentações, e não quadrático, como ocorreu com as comparações. Isso acontece porque o vetor já está ordenado. Em razão disso, o método QPE apenas realiza trocas quando as variáveis i e j se encontram, trocando o elemento com ele mesmo, a cada vez que o procedimento partição é chamado, o que acontece exatamente o número de elementos do vetor.

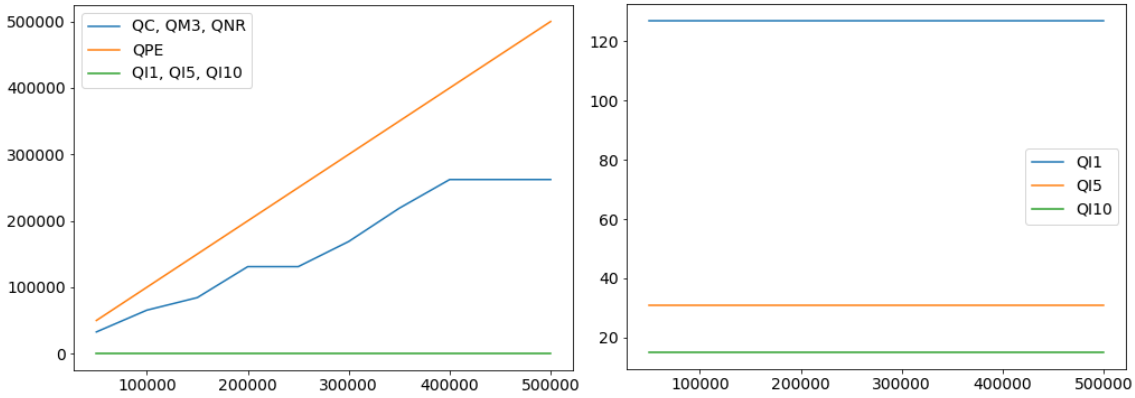

Figura 5 - Número de movimentações para vetores ordenados

Os métodos QC, QM3 e QNR tiveram exatamente o mesmo número de movimentações, quando consideramos vetores de mesmo tamanho. Isso acontece porque realizam precisamente as mesmas trocas quando submetidos aos mesmos vetores, pois QC e QNR são equivalentes, e a escolha do pivô do QM3 sempre resultará no mesmo escolhido pelos dois métodos anteriores.

O formato da curva desses métodos pode ser explicado pelas diferentes maneiras de se atingir vetores de tamanho 1 nas separações do algoritmo *Quicksort*. Nesse caso, em que o pivô sempre será a mediana do vetor, o número de partições será log(n), e vetores de diferentes tamanhos podem gerar o mesmo número de movimentações porque geram subvetores finais de tamanhos 2 ou 3. Ao subdividir-se vetores de tamanho 2, teremos zero movimentações, e ao subdividir vetores de tamanho 3 teremos

uma movimentação. Como esses números não são proporcionais, o crescimento do gráfico de movimentações não é linear, o que ocasionou o formato diferenciado.

Quanto às variações que utilizam o método de ordenação por inserção, QI1, QI5 e QI10, acontece um número fixo de operações de movimentações, independentemente do tamanho do vetor, porque o número de iterações até o início do método de ordenação depende do tamanho do vetor. Em razão disso, e por haver log(n) partições, o número de movimentações é fixo, enquanto o vetor estiver sendo ordenado pelo algoritmo *Quicksort*. Já quando é iniciada a fase de ordenação por inserção, o número de movimentações é zero, pois o elemento i é sempre maior que o i-1 e, por isso, não ocorrerão trocas a partir desse ponto do algoritmo.

Assim, quanto maior o trecho do vetor executado pelo método da inserção, menor o número de movimentações.

Em relação ao tempo de execução, a figura 6, à esquerda, deixa clara a ineficiência do método QPE para vetores ordenados. Como relatado para a métrica de comparação, a ordem de complexidade de tempo é n^2, motivo pelo qual a execução de vetores grandes, com 500 mil elementos, demorou mais de uma hora, enquanto as demais variações levaram segundos.

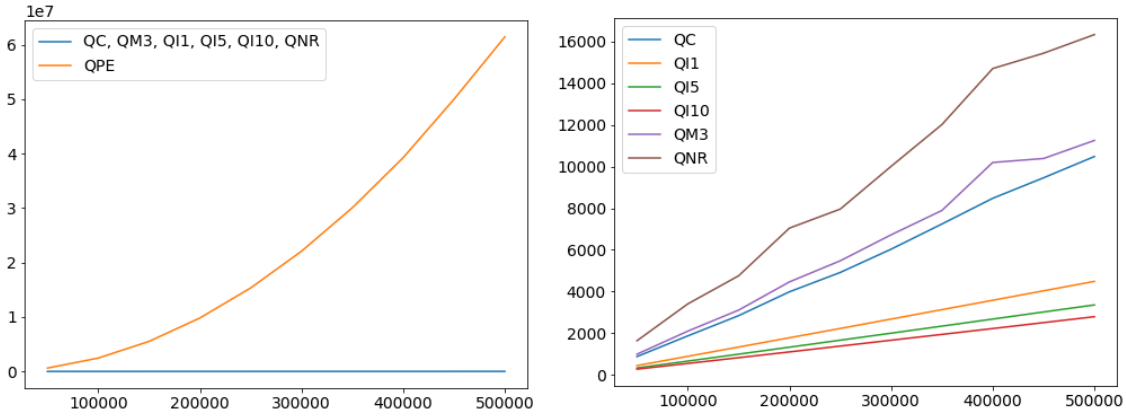

Figura 6 - Tempo de execução para vetores ordenados

Por outro lado, quando comparamos a execução para os demais métodos, através do lado direito da figura 6, observamos, também para a métrica de tempo, que o uso do algoritmo de inserção é benéfico para vetores ordenados de forma crescente, de maneira que as execuções mais rápidas foram as que usaram o método de inserção para porcentagens maiores do tamanho do vetor.

Ademais, é possível observar, mais uma vez, o impacto negativo no tempo da realização de uma mediana, pois o QM3 demorou mais que o QC. Da mesma maneira, é possível observar a maior eficiência das chamadas recursivas em lidar com as pilhas, quando comparado ao objeto criado, pilha.

Por fim, foi possível observar que o uso do método da inserção em vetores já ordenados é vantajoso, pois reduz a ordem de complexidade n.log(n) do *Quicksort* para a complexidade linear do método de inserção, fazendo uso do melhor casso desse último algoritmo.

É importante observar que, para vetores ordenados, em nenhuma hipótese acontecem partições degeneradas para os métodos QC, QM3, QNR, QI1, QI5 e QI10, enquanto todas as partições do método QPE são degeneradas.

4.3.3. Vetores ordenados de forma decrescente

Para vetores ordenados de forma decrescente, as métricas obtidas foram muito semelhantes àquelas de vetores ordenados de forma crescente, motivo pelo qual não serão repetidas, nesse item, informações e explicações apresentadas no item anterior.

Nesse sentido, a figura 7 é extremamente semelhante à figura 4, e evidencia, mais uma vez, as partições degeneradas do método QPE, que sempre seleciona como pivô o maior elemento do vetor.

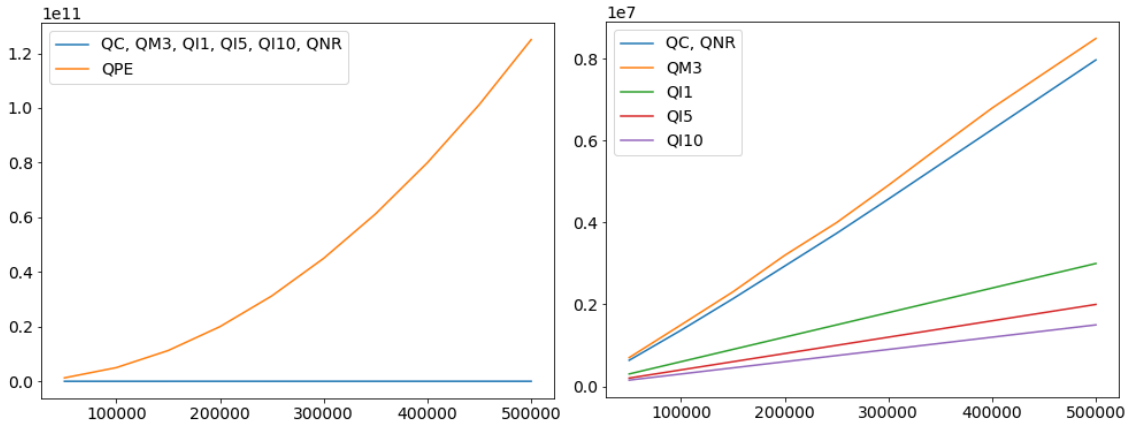

Figura 7 - Número de comparações para vetores decrescentes

Da mesma maneira, os métodos que utilizam a inserção realizam o menor número de comparações, pois utilizam da complexidade linear do método da inserção para vetores ordenados.

É importante observar, portanto, que após a primeira iteração do *Quicksort*, para qualquer um dos métodos QC, QNR, QM3, QI1, QI5, QI10, o vetor terá sido ordenado, pois o primeiro elemento será trocado com o último, o segundo com o penúltimo, e assim sucessivamente. Desta forma, após a primeira operação de partição, o vetor terá sido ordenado e aplica-se a esse caso as mesmas características observadas na ordenação do vetor crescente.

A figura 8, por sua vez, mostra o número de movimentações realizadas. Dessa vez foi apresentada apenas uma figura, que possui todas as variações do *Quicksort*, porque uma nova figura não seria acrescentaria nenhuma informação relevante, já que há uma sobreposição integral das curvas QC, QM3 e QNR, e das QI1, QI5 e QI10.

Para o caso específico de vetores decrescentes, o número de movimentações é igual ao número de movimentações do vetor crescente acrescido do tamanho do vetor sobre dois. Pois, como descrito anteriormente, na primeira iteração o vetor será ordenado, a partir de quando se aplica a ele todas as métricas e respectivas justificativas do item anterior.

Dessa vez, no entanto, o número de movimentações não foi maior no método QPE. Isso se deu porque a partição degenerada do QPE continua realizando trocas de forma linear, ou seja, a cada partição gerada, é trocado o primeiro com o último

elemento do vetor, uma única vez. Como são geradas n-1 partições, são feitas n-1 trocas. Já os métodos QC, QM3 e QNR realizam n/2 trocas na primeira partição do *Quicksort* e, para cada partição seguinte, segue realizando a troca do pivô com ele mesmo. Assim, esses métodos acabaram resultando em um maior número de movimentações que o QPE.

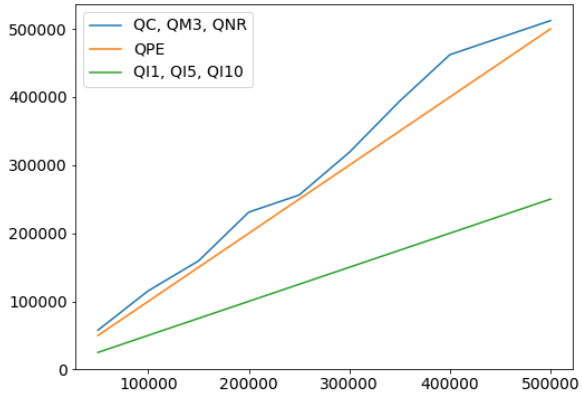

Figura 8 - Número de movimentações para vetores decrescentes

Em relação à métrica de tempo, observa-se que o gráfico se assemelha ao de vetores ordenados de forma crescente, como esperado. Assim, as justificativas que se aplicam ao caso anterior, se aplicam também ao presente caso.

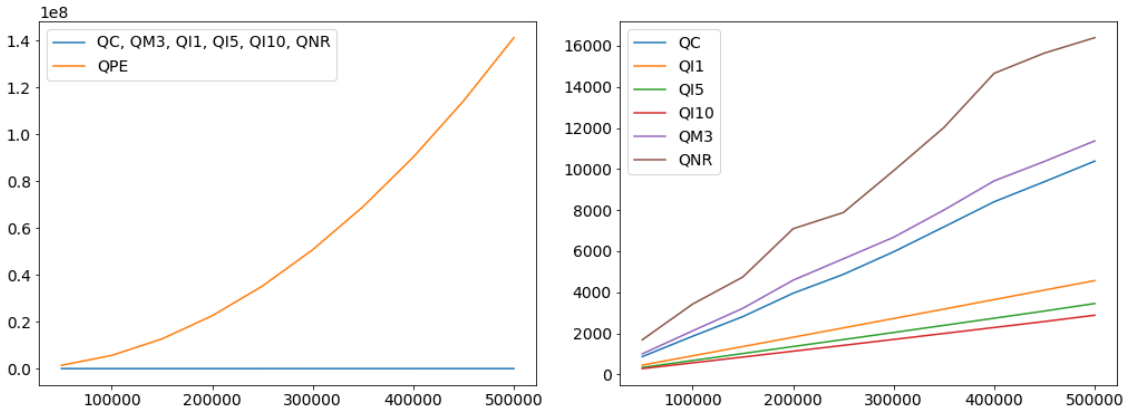

Figura 9 - Tempo de execução para vetores decrescentes

Por conseguinte, fica evidente que a utilização do método de inserção é útil quando usado em conjunto com o *Quicksort*, para vetores ordenados de forma decrescente, pois, para vetores ordenados (o que acontece após a primeira iteração), a ordem de complexidade do método da inserção é linear.

É importante observar, no entanto, que caso o método da inserção fosse utilizado sozinho, esse seria o pior caso do algoritmo, uma vez que seriam necessárias movimentações e comparações com ordem de complexidade n^2.

Desta maneira, método menos eficaz, mais uma vez, foi o QPE, em razão das partições degeneradas. E os mais eficientes foram os que utilizaram o método de inserção.

4.4. Análise das variações do Quicksort

Na presente seção serão avaliadas, individualmente, as diferentes variações do *Quicksort*, bem como suas vantagens, desvantagens, melhores e piores casos.

4.4.1. Quicksort clássico (QC)

O *Quicksort* clássico é, para uma distribuição de probabilidade uniforme, o mais rápido, em média, dentre os métodos de ordenação apresentados neste trabalho, conforme evidenciado, principalmente, pela análise de vetores aleatórios.

É importante notar, no entanto, que existe uma probabilidade, mesmo que baixa, de que haja uma sucessão de partições degeneradas nesse método, o que ocasionaria uma aproximação à ordem de complexidade quadrática.

Nota-se, por fim, que não há variação de desempenho desse método em função do tipo de vetor a ser ordenado, com a ressalva descrita no parágrafo anterior.

4.4.2. Quicksort mediana de três (QM3)

O *Quicksort* mediana de três tem comportamento bastante semelhante ao QC, pois não sofre variação de desempenho em função do tipo de vetor ordenado e, em média, apresenta um bom desempenho.

A principal diferença do QM3 em relação ao QC é a redução da probabilidade de partições degeneradas, o que afasta esse método de possíveis execuções muito lentas, ocasionadas por essas partições.

Em contrapartida, existe um custo adicional de comparações e de tempo para esse método, por mais que, eventualmente, o número de movimentações seja menor em função das partições possivelmente mais igualitárias.

4.4.3. Quicksort primeiro elemento (QPE)

O *Quicksort* primeiro elemento possui desempenho um pouco pior que o QC para vetores aleatórios, ocasionado, principalmente, pela maior probabilidade de que a partição não seja igualitária, mas, ao contrário, o pivô seja tendencioso a elementos menores. Tal fato é consequência direta do método de funcionamento do *Quicksort*, que atribui ao lado esquerdo do vetor números menores, na fase de partição.

O pior caso desse método é bastante claro e provável, vetores ordenados de forma crescente e decrescente. Nesses casos, a ordem de complexidade torna-se quadrática, e o elevado tempo de execução, ocasionado pelo altíssimo número de comparações, torna esse método inviável.

Como a probabilidade de um vetor estar ordenado é consideravelmente maior que qualquer outra configuração, considera-se que não há motivos para utilizar o QPE em nenhuma situação específica, pois não foi possível verificar nenhum caso em que ele apresentasse desempenho e métricas melhores que o QC, por exemplo.

4.4.4. Quicksort com inserção (QI1, QI5 e QI10)

O *Quicksort* com inserção apresentou desempenho consideravelmente pior que os métodos QC, QMR, QPE e QNR para vetores aleatórios, pois ordena uma parte do vetor através de um método com complexidade quadrática, o método da inserção. Assim, quanto maior a porcentagem do vetor ordenado pelo método quadrático, pior foi o desempenho observado.

Em contrapartida, para vetores ordenados de forma crescente, esse método é extremamente eficiente, pois faz uso do melhor caso do método de inserção, que possui ordem de complexidade linear. Nesse caso, quanto maior a porcentagem do vetor ordenado pela inserção, mais rápida será a execução do algoritmo, com menor número de comparações e movimentações.

Desta maneira, talvez fosse interessante a utilização desse método para situações em que, majoritariamente, os vetores de entrada já estão ordenados de forma crescente ou decrescente. Recomendar-se-ia, no entanto, que fosse utilizado o QI1, para evitar os altos tempos de execução do QI5 e do QI10 para vetores aleatórios.

4.4.5. Quicksort não recursivo (QNR)

O *Quicksort* não recursivo é idêntico ao QC no que tange número de comparações e movimentações, pois se comportam de maneira idêntica, quando submetidos aos mesmos vetores.

Assim, esse método também possui uma probabilidade de sucessivos casos degenerados, o que resultaria em uma ordem de complexidade quadrática, bem como apresentou desempenho semelhante independentemente do tipo de vetor de entrada.

A diferença entre essas variações se resume à eficiência do computador ao lidar com pilhas implementadas internamente, ao realizar chamadas recursivas, quando comparado a uma pilha implementada na forma de uma estrutura de dado. Como era esperado, as pilhas nativas funcionam de forma mais eficiente e, consequentemente, mais rápidas, de maneira que houve uma diferença significativa no tempo de execução entre o QNR e o QC, sendo o segundo (recursivo), consideravelmente mais rápido.

5. Conclusão

O presente trabalho teve como objetivo analisar o desempenho do algoritmo de ordenação *Quicksort*, bem como diversas variações dele, através de métricas de número de comparações, número de movimentações e tempo de execução.

Para atingir tal objetivo foi criado, inicialmente, um programa capaz de ordenar vetores de tamanhos variados e através dos diversos métodos de ordenação que se desejava comparar. Ademais, esse programa deveria computar as métricas de mediana do tempo de execução, média do número de comparações e média do número de movimentações entre elementos do vetor.

Após concluída a fase de implementação do programa, foi iniciada a fase de testes, nos quais foram submetidas 210 chamadas ao programa, que, para cada chamada, executava a ordenação de 50 vetores. Para os casos de ordenação cuja complexidade era n.log(n) ou menor, a ordenação se deu em segundos, mesmo para vetores muito grandes, com 500 mil elementos, por exemplo. Já para casos de partições degeneradas e execuções de complexidade quadrática, houve chamadas que demoraram mais de uma hora para serem concluídas.

Em razão disso, foi necessário executar uma sequência de comandos de uma vez, através de um arquivo *bash*, de forma a permitir, eficientemente, a execução de todos os comandos, sequencialmente, de um dia para outro.

Após realizar todas as 210 chamadas ao programa e obter os resultados das métricas desejadas, esses dados foram organizados na forma de um arquivo *csv* e transformados em gráficos que permitiram visualização mais rápida das informações e análises mais precisas das mesmas.

O resultado obtido permitiu entender, com sucesso, o funcionamento do algoritmo *Quicksort* e de suas variações quando submetidos a diferentes tipos e tamanhos de vetores.

Ademais, foi possível observar que o *Quicksort* clássico é, em média, o mais eficiente, mas que existem situações em que outra variação pode trazer resultados melhores.

Além disso, em nenhuma variação do algoritmo é possível garantir a inexistência de partições degeneradas. Mesmo quando o objetivo da modificação do *Quicksort* clássico é esse, como é o caso da *Quicksort* mediana de três, apenas garante-se a diminuição da probabilidade de partições degeneradas, sem nunca garantir que elas não ocorrerão.

Por fim, pôde-se observar que não existe um único algoritmo de ordenação ideal para todas as situações. Para cada aplicação, contexto, dados de entrada e distribuição de probabilidade existirá uma solução ótima, que deve ser escolhida com uma metodologia adequada, esmero e critérios claros.

Assim, aplicar uma metodologia semelhante à empregada no presente trabalho para escolha de algoritmos que melhor se adaptam à situação em questão é uma excelente prática e conduta, sempre com o intuito de conhecer profundamente o problema com o qual se está lidando e aplicar a solução mais adequada, respeitando os melhores e piores casos em cada situação, bem como a tolerância a riscos.

6. Referências

Chaimowicz, L., Prates, R., (2019), Pilhas e Filas, In Estrutura de Dados, Departamento de Ciência da Computação, UFMG.

Chaimowicz, L., Prates, R., (2019), Ordenação: Quicksort, In Estrutura de Dados, Departamento de Ciência da Computação, UFMG.

Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2009), Introduction to Algorithms, MIT Press, 3rd edition.

Sedgewick, R., Wayne, K. (2018), "2.3 Quicksort", In Algorithms, Princeton, 4th edition, https://algs4.cs.princeton.edu/23quicksort/, junho.

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