Trabalho Prático de Estrutura de Dados
Análise de Diferentes Implementações do Quicksort

Roberto Gomes Rosmaninho Neto

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

robertogomes@dcc.ufmg.br

1. Introdução

O problema proposto por esse trabalho foi implementar e analisar o desempenho de sete diferentes variações do algoritmo de ordenação Quicksort. A análise das implementações deveria ser feita utilizando diversos vetores com características e tamanhos diferentes a fim de obter-se uma análise consistente. Para isso, as métricas utilizadas deveriam ser: Número de movimentações de registros, número de comparações de chaves e o tempo de execução de cada variação do algoritmo. A entrada do programa deveria ser feita via argumento, pela linha de comando, especificando a variação do algoritmo a ser utilizada, o tipo do vetor de entrada e seu tamanho. Assim ao final da execução o programa deveria retornar não apenas os detalhes de entrada, mas também os resultados das métricas avaliadas e opcionalmente os vetores utilizados nos teste.

O Quicksort foi desenvolvido pelo cientista da computação Tony Hoere em 1960, mas publicado apenas em 1962 com alguns refinamentos[1]. Hoje, ele é um dos algoritmos de ordenação mais conhecidos devido a sua eficiência e sua larga utilização nos mais diversos projetos, tais como, na função da biblioteca padrão de C *qsort* e no método *sort* da biblioteca de *arrays* de Java[2]. O Quicksort é um algoritmo de troca, ou seja, a ordenação dos vetores dá-se a partir da troca de posição entre chaves(ou células do vetor). Além disso, esse algoritmo utiliza o paradigma de programação Dividir para Conquistar. O MergeSort é algoritmo muito conhecido e também incluído nesta categoria de algoritmo. O paradigma Dividir para Conquistar pode ser explicado como uma abordagem recursiva na qual a entrada do algoritmo é ramificada múltiplas vezes a fim de quebrar o problema maior em problemas menores da mesma natureza.[3]

A implementação do Quicksort pode ser dividida em três importantes etapas: Dividir, Conquistar e Combinar; as quais seguem detalhadas a seguir:

- Dividir: Este primeiro passo, conhecido também como Partição(veja a Figura 1) é responsável por escolher o pivô(1), buscar uma chave no intervalo [inicio, pivô] maior que o pivô(2) e outra chave menor que este no intervalo [pivô+1, fim](3), e então fazer a troca dessas duas chaves até que a posição da ”seta”que avalia o primeiro intervalo seja maior que a posição da que avalia o segundo intervalo(4).
- Conquistar: Terminado o Partição é garantido que todos os elementos à esquerda do pivô são menores que este e os da direita, maiores. Assim, é possível dividir o vetor de forma que os dois novos sub vetores são iguais ao primeiro e segundo intervalos citados acima. Além disso, cada um terá seu próprio pivô e a Partição será executada novamente até que haja apenas sub vetores com uma chave.

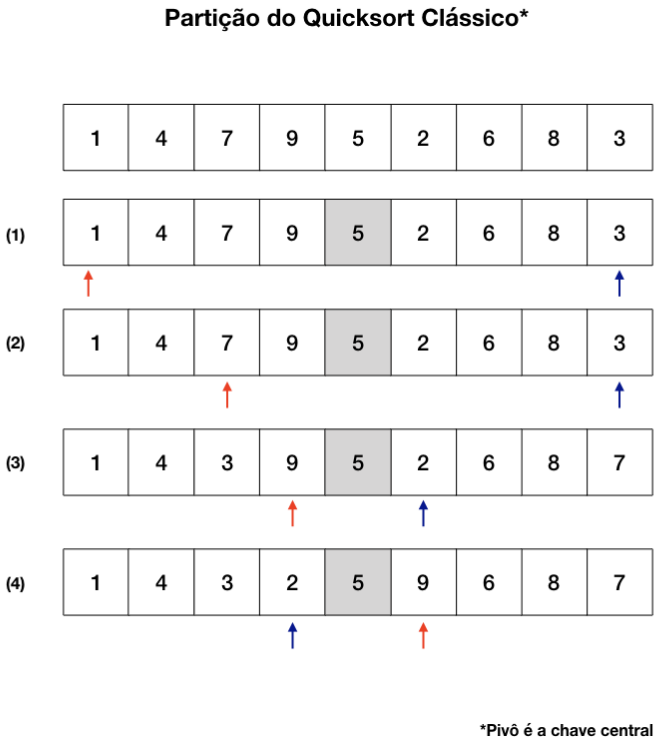

Figura 1. Método de Partição

- Combinar: Após a ordenação de todos os elementos pelas recursivas execuções do método de partição, as chaves devem ser unidas em um único vetor novamente e, assim, o vetor inicial estará ordenado.

A Figura 1 e o Método Partição do item "Dividir" se referem à implementação do **Quicksort Clássico**. Neste, o pivô sempre é o elemento central ou $\lceil (inicio + fim)/2 \rceil$. Vale ressaltar que nesse trabalho outras seis variações do Quicksort também foram implementadas, e a seguir é apresentada a descrição de cada uma delas.

- Pivô é sempre a primeira chave. Essa variação será chamada apenas de "**Primeiro Elemento**".
- O pivô é a mediana das chaves inicial, central e final. Essa variação será chamada apenas de "**Mediana de 3**".
- Utiliza o mesmo pivô da Mediana de 3, porém quando o tamanho dos sub vetores avaliados pelo método partição é menor que 1% do tamanho do vetor original, todos esses sub vetores são ordenados pelo algoritmo de Inserção[4] e não mais mais pelo partição. Essa variação será chamada apenas de "**Inserção 1%**".
- Utiliza o mesmo método que a Mediana de 3, mas quando o tamanho dos sub vetores avaliados pelo método partição é menor que 5% do tamanho do vetor original, esses sub vetores são ordenados pelo algoritmo de Inserção. Essa variação será chamada apenas de "**Inserção 5%**".
- Utiliza o mesmo método que as duas variações anteriores, mas quando o tamanho dos sub vetores é menor que 10% do tamanho do vetor original, esses sub vetores são ordenados pelo algoritmo de Inserção. Essa variação será chamada apenas de "**Inserção 10%**".

- Por fim, a ultima variação utiliza um método iterativo para fazer a Partição e união das chaves do vetor, para isso é utilizado uma Estrutura de Pilha[5] auxiliar. Essa variação será chamada apenas de "**Não Recursivo**"

2. Implementação

Este trabalho prático foi implementado na linguagem C++, utilizando as boas práticas de orientação à objetos. Os Ambientes de testes utilizado para executar o código foram um Macbook Air Core i5, dual core de 1.8Ghz, 8GB de RAM e SSD de 128GB e um Dell OptiPlex 7040M com processador i7vPRO. Os compiladores utilizados para testes foram o Apple LLVM version 10.0.1 (clang-1001.0.46.4) e o GCC 5.4.0 da GNU Compiler Collection.

Na implementação das variações do Quicksort uma importante decisão de projeto precisou ser tomada: implementar cada variação de forma separada, sendo cada uma com sua própria função Ordena e Partição, ou utilizar apenas uma implementação principal, mas avaliando cada **caso**, antes de aplicar operações não genéricas entre as variações. Optou-se pela segunda devido à baixa complexidade que seria inserida no programa e a possibilidade de reaproveitar várias operações e até funções inteiras.

2.1. Classes e Funções

Esse programa possui três classes principais: Vector, Pilha, Quicksort. A primeira é responsável principalmente pela geração de vetores, mas também por copiar os elementos de um vetor para outro e ordenar um pequeno vetor utilizando um algoritmo de Inserção como base. Este último é utilizado no para ordenar os tempos de cada execução do Quicksort dentro do programa. A segunda classe é responsável por gerir as execuções necessárias do Quicksort Não Recursivo. Por fim, a terceira classe é responsável pela implementação de todas as variações do Quicksort, bem como a mensura de tempo, comparações e movimentações realizadas a cada execução.

Além dessas classes, divididas em arquivos *.h* e *.cpp*, para a execução do programa é utilizado um arquivo *main.cpp*, onde os vetores são incializados, ordenados, seu custo de tempo, comparações e movimentações é calculado. O funcionamento do programa ocorre por meio de *m* execuções de conjunto de operações com vetores, sendo *m* o número de testes que o usuário define na varável global *NUM_TESTES*. Dessa forma, os resultados mostrados para o usuário na saída são na verdade a **mediana** dos tempos e as **médias** do número de comparações e movimentações dos *n* conjuntos operações com vetores.

Cada uma das três principais classes serão detalhadas a seguir:

2.1.1. Quicksort

Um elemento da classe Quicksort para ser iniciado deve receber um ponteiro para o vetor a ser ordenado e seu tamanho. Cada elemento dessa classe também é inicializado com o número de comparações, movimentações e caso iguais a 0.

A classe Quicksort além de construtor e destrutor possui as seguintes funções:

Além dessas funções, há outras responsáveis por executar cada variação do Quicksort. Todas essas são compostas por apenas dois comandos, um para atribuir um valor do

Tabela 1. Principais Funções da Classe Quicksort

Nome da Função	Parâmetros	Retorno	Descrição
get_vetor	-	Ponteiro pro vetor inicializado com a classe	Retorna o vetor da armazenado no elemento da classe Quicksort
get_tamanho	-	Tamanho do vetor inicializado com a classe	Retorna o tamanho do vetor da armazenado no elemento da classe Quicksort
Mediana3	*inicio, *fim, *vetor	Mediana de três valores	Passados os valores das posições inicial e final do vetor, além deste, essa função obtém o valor central do vetor, e retorna a mediana deste valor com os outros dois recebidos por parâmetro
EscolhePivo	Esq, *inicio, *fim, *vetor	Retorna o pivô de acordo com o caso	Calcula o pivô do vetor de acordo com o caso, ou seja, com a variação do Quicksort selecionada
Particao	Esq, Dir, *inicio, *fim, *vetor	-	Chama o EscolhePivo e utilizando o método de partição, garante que todos os elementos à esquerda com pivô sejam menores que ele e que todos os elementos da direita dele sejam maiores.
Ordena	Esq, Dir, *vetor	-	Testa se o tamanho do vetor atual é menor que 1%, 5% ou 10% do vetor original, se for chama o método Insercao, senão chama o método Particao e, recursivamente, o método Ordena pra cada metade do vetor
Insercao	*vetor, tamanho	-	Ordena o vetor recebido por parâmetro pelo método de Inserção

intervalo [1, 7] à variável _**caso** e outro que chama a função Ordena para o vetor principal.

Por fim, há as funções de retorno de métrica, são elas:

Tabela 2. Funções de Retorno de Métricas da Classe Quicksort

Nome da Função	Parâmetros	Retorno	Descrição
get_comp	-	Número de comparações realizadas	Caso nenhuma comparação tenha sido realizada retorna 0
get_mov	-	Número de movimentações realizadas	Caso nenhuma movimentação tenha sido realizada retorna 0
get_tempo	Caso	Tempo gasto para executar uma variação do Quicksort	Retorna o tempo de acordo com o caso recebido como parâmetro. Cada caso no intervalo de [1, 7] possui uma variação do Quicksort associada. Se um número fora do intervalo [1, 7] for selecionado o usuário receberá uma mensagem de erro e o valor 0 como retorno

2.1.2. Pilha

A classe pilha, nesse programa, apenas será utilizada para gerenciar as chamadas do método Partição dentro da variação Não Recursiva do Quicksort. Dessa forma, apenas as funções necessárias para esse fim foram implementadas. A implementação dessa Pilha foi feita tendo como base uma lista simplesmente encadeada, visando a facilidade que esta oferece por ser alocada em diferentes espaços da memória e, ainda assim, ter seus elementos ordenados.

Um elemento da classe Pilha é inicializado com o seu topo _topo e seu _tamanho iguais a 0. A Pilha também possui construtor e destrutor, este desempilha elementos até que o tamanho da pilha seja 0. Além dessas, essa classe conta com as seguintes funções:

Tabela 3. Funções da Pilha

Nome da Função	Parâmetros	Retorno	Descrição
Empilha	*item	-	Uma nova posição é criada com o item e com um ponteiro para o topo da Pilha
Desempilha	-	Retorna um ponteiro para o item que estava no topo da Pilha	Se a Pilha não estiver vazia retorna o elemento da posição do topo da Pilha
get_tamanho	-	Retorna o tamanho da Pilha	Retorna o tamanho da Pilha
get_top	-	Retorna um ponteiro pra posição no topo da pilha	Se a pilha estiver vazia retorna nullptr, senão retorna a posição do topo da Pilha

2.1.3. Vector

A classe Vector, nesse programa, apenas é utilizada para a manipulação de vetores dentro da *main*, ou seja, apenas é utilizada para geração de vetores, cópia ou ordenação para um caso específico(vetor de Tempo de cada conjunto de execuções). As principais funções dessa classe são:

Tabela 4. Funções de Geração de Vetores da classe Vector

Nome da Função	Parâmetros	Retorno	Descrição
OrdemCrescente	-	Retorna ponteiro pra um vetor	Cria um vetor com os elementos ordenados de forma crescente
OrdemDecrescente	-	Retorna ponteiro pra um vetor	Cria um vetor com os elementos ordenados de forma decrescente
OrdemAleatória	-	Retorna ponteiro pra um vetor	Cria um vetor com os elementos ordenados de forma aleatória

As outras funções da classe Vector apenas são utilizadas para tratar os vetores cujos elementos são métricas dos resultados de cada conjunto de operações com vetores realizadas no programa(Geração, Ordenação, Cálculo de Tempo de Execução, Número de Movimentações e Comparações).

Tabela 5. Funções de Geração de Vetores da classe Vector

Nome da Função	Parâmetros	Retorno	Descrição
OrdenaDouble	tamanho, *tempo	-	Ordena um vetor de elementos do tipo double pelo algoritmo de Inserção
CopiaVetor	-	Retorna ponteiro para um novo vetor	Copia o vetor do elemento Vector para um novo vetor e o retorna

A primeira função é responsável por ordenar os Tempos de Execução para que a mediana possa ser obtida posteriormente. A segunda função é responsável por copiar os elementos do vetor em Vector para um novo, assim é possível salvar os estados inciais de cada vetor para ser mostrado ao usuário posteriormente caso seja solicitado.

2.2. Fases de Implementação

A implementação do programa ocorreu em cinco fases: Geração de Vetores, Implementação das Variações do Quicksort, Mensura de Tempo, Mensura de Comparações e Movimentações e por fim, Adaptação para Geração dos Testes. Cada fase será detalhada posteriormente, assim como as decisões que justificam a adoção desse formato de implementação.

2.2.1. Geração de Vetores

A Geração de vetores aleatórios, ordenados de forma crescente e decrescente é realizada pela classe *Vector* utilizando os métodos *OrdemCrescente()*, *OrdemDecrescente()* e *OrdemAleatoria()*.

- *OrdemCrescente()*: É composto por um simples loop que itera a variável *i* de 0 a n e armazena em cada posição *i* o valor de *i+1*.
- *OrdemDecrescente()* : Também composto por um simples loop que itera uma variável *k* de n a 1 e uma variável *i* de 0 a n, assim cada posição *i* recebe o valor de *k*.
- *OrdemAleatoriia()*: Utiliza a biblioteca *random* do C++ para gerar um número aleatório dentro do intervalo [1, n], também é utilizado um loop que itera a variável *i* de 0 a n para armazenar em cada posição *i* o número aleatório gerado.[6]

Dessa forma, todo o intervalo [1, n] será armazenado nos vetores gerados pelos primeiros dois métodos, sendo n o tamanho do vetor solicitado pelo usuário como argumento antes da execução do programa. A complexidade de cada um desses métodos é O(n).

2.2.2. Implementação das Variações do Quicksort

A ideia chave para a implementação das diversas variações do Quicksort foi o uso de uma variável que por meio de *if-else* e *switches* é responsável por executar apenas as operações necessárias a cada variação. Nesse programa, essa variável é um inteiro chamado **_caso** e seu uso é dado da seguinte forma: Durante a chamada da variação desejada **_caso** recebe um valor entre 1 e 7, cada número representa uma variação seguindo a seguinte ordem: Clássico, Primeiro Elemento, Mediana de 3, Inserção 1%, Inserção 5%, Inserção 10% e Não Recursivo.

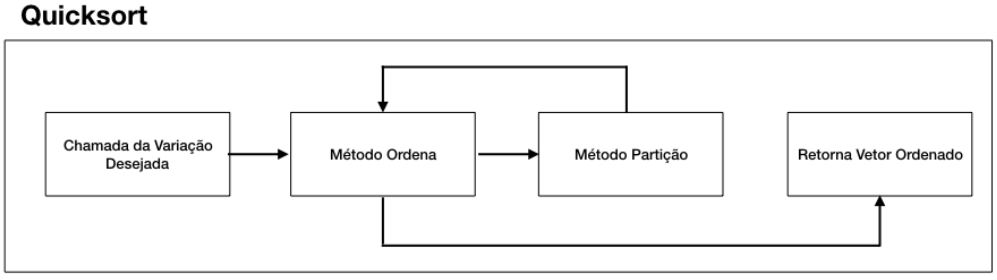

Figura 2. Diagrama do Quicksort

Uma vez que a variação do Quicksort foi escolhida, o método Ordena é então chamado e logo depois o Método Partição até que certa condição seja satisfeita e assim será retornado um ponteiro para o vetor ordenado assim como sugere a Figura 2. Essa é uma visão genérica do Quicksort, logo essa estrutura é utilizada por todas as variações do Quicksort, justificando assim a oportunidade de mesclar as variações do algoritmo.

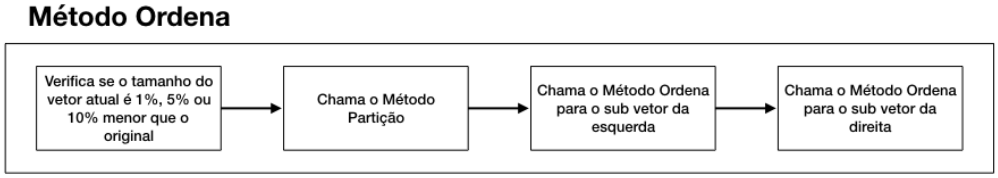

Figura 3. Diagrama do Método Ordena

O método Ordena, o primeiro a ser chamado, é onde começam as diferenças entre as implementações. A primeira parte desse método verifica se a soma dos índices recebidos por parâmetro é menor que certa porcentagem do vetor original, pois esse método é chamado recursivamente cada vez com índices que representam partes menores do vetor original, assim o conjunto de elementos do vetor entre esses índices de é chamado de sub vetor. Nessa primeira parte que as variações Inserção 1%, Inserção 5% e Inserção 10% podem realizar suas operações específicas, ou seja, caso o tamanho do sub vetor em análise seja 1% do tamanho do vetor original, então o método Inserção é chamado para ordenar esse sub vetor, analogamente ocorre com as variações Inserção 5% e Inserção 10%.

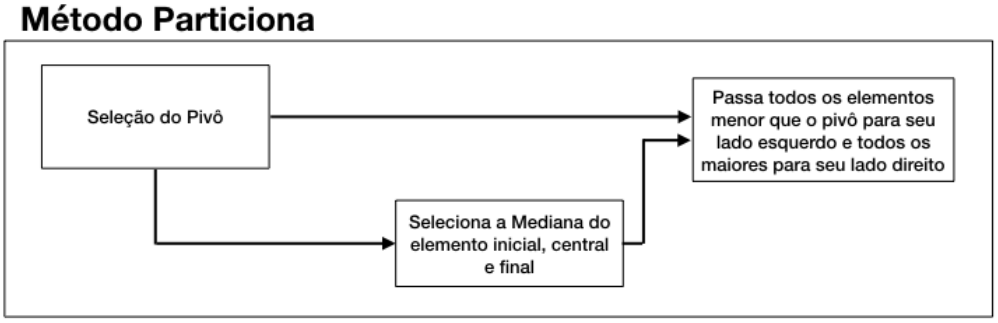

Figura 4. Diagrama do Método Partição

O método Particiona, mesmo nas implementações da literatura[1] é o método mais importante do Quicksort, nessa implementação não é diferente. A primeira parte chama a função que seleciona o Pivô, este é selecionado de acordo com o **_caso**, as variações Mediana de 3, Inserção 1%, Inserção 5% e Inserção 10% ainda chamam a função Mediana3 para auxiliar na seleção do Pivô. O pivô pode ter os seguintes valores de acordo com a variação escolhida:

- Pivô Clássico e Não Recursivo: O pivô é o elemento central de cada vetor ou sub vetor.
- Pivô Primeiro Elemento O pivô é sempre o primeiro elemento de cada vetor ou sub vetor.

- Pivô Mediana de 3: O primeiro elemento, o elemento central e o último elemento de cada vetor ou sub vetor são ordenados e a mediana deles é selecionada como pivô.

Determinado qual será o pivô, então o partição organizará o vetor, como mostrado no início desse documento, até que todos os elementos à esquerda do pivô sejam menores que ele e todos os elemento à sua direita sejam maiores ou iguais a ele.

Após o fim do método Partição o método Ordena é chamado para a parte esquerda recursivamente e depois para a parte direita recursivamente. No final do método ordena é garantido que todos os elementos do vetor original estão ordenados e assim um ponteiro para este é retornando.

2.2.3. Mensura de Tempo

Uma das principais métricas necessárias para avaliar o desempenho de um algoritmo é seu custo de tempo. Neste trabalho, além de implementar diferentes variações do Quicksort é necessário fazer uma análise de cada um e, se possível, determinar a melhor variação e, para isso o custo do tempo de execução é imprescindível. Assim, o programa possui uma função específica que utilizando do valor de **_caso** retorna o tempo para ordenar um vetor por meio da variação desejada. O tempo é calculado utilizando funções da biblioteca *chrono* de C++. Para isso, o horário que precede imediatamente a execução é armazenado em uma variável ***t1***, depois o horário que sucede imediatamente a ordenação é salvo em uma variável ***t2***, a função então retorna a diferença entre essas variáveis. O retorno é feito em uma variável do tipo *double* e a representação do tempo é feita em microssegundos.

2.2.4. Mensura de Comparações e Movimentações

Outras duas métricas muito importantes para se avaliar o desempenho de um algoritmo de ordenação são os número de comparações de chaves e movimentações de registros durante a execução deste. Nesse programa, essas duas métricas podem ser avaliadas nos métodos de Inserção, Partição e Mediana3 dentro da classe Quicksort, visto que apenas nessas três classes há de fato manipulação de vetores, ou seja, somente nelas um elemento do vetor é comparado com outro e pode ocorrer a de troca de posições entre eles.No caso do Inserção a comparação é feita com o elemento anterior ao avaliado, respeitando as características do algoritmo, assim como no Quicksort a comparação é feita com o pivô, mas outros elementos além do pivô podem trocaR de posições entre si. Essas métricas foram obtidas por meio do incremento de variáveis ***_mov*** e ***_comp*** que são atributos da classe Quicksort.

2.2.5. Adaptação para Geração dos Testes

Por fim, o programa precisou ser adaptado para que cada variação selecionada pudesse ser executada várias vezes e, assim garantir que o resultado da execução de uma

variação do Quicksort não seja enviesada ou fortemente influenciada por fatores externos à implementação. Esse programa possui uma variável ***NUM_TESTES*** que define o número de vezes a execução de cada conjunto de operações com vetores (Geração, Ordenação e Captura de Métricas). No início do programa é criado um vetor com o tamanho igual ao ***NUM_TESTES*** para cada métrica e cada posição desses vetores recebe o resultado de uma execução. A saída para o usuário é mediana dos tempos das execuções e a média do número de comparações e movimentações das execuções.

2.3. Entradas e Saídas

O programa aceitar quatro até quatro parâmetros como entrada. Sendo eles: A variação do Quicksort, o tipo do vetor a ser ordenado, o tamanho do vetor a ser ordenado e uma parâmetro opcional para exibir na saídas os vetores utilizados para em cada conjunto de operações antes de sua ordenação. Cada entrada dever seguir os seguintes padrões:

- Variação: **`QC`** - Clássica, **`QPE`** - Primeiro Elemento, **`QM3`** - Mediana de 3, **`QI1`** - Inserção 1%, **`1I5`** - Inserção 5%, **`QI10`** - Inserção 10%, **`QNR`** - Não Recursivo.
- Tipo do Vetor: **`Ale`** - Vetor Aleatório, **`OrdC`** - Ordem Crescente, **`OrdD`** - Ordem Decrescente.
- Tamanho do Vetor: Inteiros não negativos, os testes foram realizados com tamanhos variando entre 50 mil e 500 mil elementos.
- Exibir Vetores Utilizados: Se a string "-p" estiver presente uma lista de vetores será apresentada na saída.

Logo a execução do programa deverá ser feita seguindo o modelo padrão abaixo:

./nomedoprograma <variacao> <tipo> <tamanho> [-p]

Por exemplo:

./tp2 `QC` `Ale` 50000 -p

ou

./tp2 `QI10` `OrdD` 100000 > qi10 ordd 100000.txt

No primeiro caso, serão gerados vetores com ordenação aleatória com 50 mil elementos de números do intervalo [1,50000] e cada um será ordenado pela variação Clássica do Quicksort, após a execução do programa além da saída padrão, todos os vetores criados deverão ser mostrados na saída. Já no segundo caso, serão gerados vetores com 100 mil elementos de números no intervalo [1,100000] ordenados de forma decrescente e cada vetor será ordenado utilizando a variação Inserção 10% e a saída padrão, sem os vetores utilizados, será gravada no arquivo "ordd 100000.txt".

O padrão de saída de cada execução será a seguinte:

<variacao> <tipo> <tamanho> <n comp> <n mov> <tempo>
< Se o parâmetro "-p" for utilizados os vetores gerados devem aparecer aqui>

Os resultados devem ser impressos na saída padrão(*stdout*). Na saídas os três primeiros valores devem ser os mesmos recebidos como parâmetros para execução, os três último devem ser, respectivamente, a média do número de comparações, a média do número de movimentações e a mediana do tempo doas testes realizados durante a execução.

3. Compilação e Execução

Esse projeto possui um *Makefile* responsável por compilar o programa e executá-lo. Este segue o mesmo modelo disponibilizado pelo monitor, salvo uma diferença no uso de *flags* para compilar o programa. Esse projeto optou por não utilizar nenhuma otimização de compilação para que os resultados realmente refletissem a eficiência da implementação. Acreditamos que algumas otimizações, principalmente em *loops* prejudicam algumas variações do Quicksort, como no caso da variação Não Recursiva, essa diferença será apresentada posteriormente na seção de Análise Experimental.

Para compilar o programa apenas digite:

```
$ make
```

Após executar este comando além dos arquivos binários ".*o*" um executável "***quicksort***" será criado. Assim, é possível executar esse programa de três formas, estas serão detalhadas a seguir:

Utilizando o *Makefile*, executando diretamente pelo linha de comando ou executando o *Bash Script* ***run.sh***.

Executando pelo *Makefile* os parâmetros padrões de testes serão utilizados, ou seja, o programa executará a variação Clássica com um vetor aleatório de 10 elementos e irá imprimir na saída os vetores utilizados nessa execução juntamente com o resultado das métricas. Para executar essa desse modom apenas digite:

```
$ make run
```

Executar pela linha de comando lhe dá mais flexibilidade para testar as diferentes variações do Quicksort com os diferentes tipos de vetores de qualquer tamanho. Para executar desse modo, apenas digite os comando seguindo o padrão de entrada definidos anteriormente. Por exemplo:

```
$ ./quicksort QPE Ale 500000
```

Por fim, o *Bash Script* disponibilizado nesse projeto é responsável por testar todas as combinações entre variações do Quicksort, tipos de Ordenação de Vetores e tamanhos no intervalo [50000, 500000] sempre com 50mil elementos de diferença. Para cada tipo de vetor é criado um arquivo *.txt* com o resultado das execuções sem o parâmetro "-p ". Nesses arquivos, há o resultado cada variação executada 10 vezes, sendo uma vez com cada tamanho diferente utilizando o tipo de vetor correspondente ao nome do arquivo. Este pode varias entre "*OrdemCrescente.txt*", "*OrdemDerescente.txt*" e "*OrdemAleatória.txt*". Além disso esses arquivos serão salvos ao fim da execução em uma pasta chamada *Resultados* no mesmo diretório da pasta *src*. Para executar esse *Script*, apenas digite":

```
$ ./run
```

4. Análise Experimental

Os objetivo desse trabalho, como mencionado anteriormente e especificado no enunciado deste, pode ser dividido entre a implementação das variações do Quicksort e na análise de cada uma delas. Nessa seção, serão discutidas a metodologia utilizada para a análise e os resultados obtidos.

4.1. Premissas

Os resultados analisados nesse documento se referem aos testes realizado no OptiPlex 7040M com processador i7vPRO, o programa foi compilado com o GCC 5.4.0 no sistema operacional Ubuntu 16.04.10.

4.2. Metodologia

Essa análise se baseia, principalmente, nas métricas implementadas no programa, são elas:

- Número de comparações entre chaves
- Número de movimentações de registros
- Tempo de Execução

Assim como requisitado no enunciado do trabalho, foram testados 3 tipos de ordenações de vetores,são elas:

- Aleatória
- Crescente
- Decrescente

Por fim, cada tipo de vetor foi testado dez vezes em intervalos de 50 mil elementos, variando desde 50 mil elementos até 500 mil elemento.

Os três tipos de vetores utilizados nos testes representam bem as situações reais de aplicação do Quicksort, visto que é possível obter o pior caso de algumas variações e, ainda assim inferir um comportamento médio do algoritmo por meio dos casos randômicos. A variação do número de elementos e o alto valor de cada um se justificam de duas maneiras. Na primeira, devido a necessidade de se encontrar um padrão de comportamento que explique o algoritmo é necessário utilizar diversas entradas[7]. Na segunda, dada a eficiência do Quicksort é necessário um largo vetor para se obter um tempo de execução expressivo, mesmo para o pior caso. Além disso, só é possível verificar a diferença da eficiência entre as variações que ordenam parte do vetor utilizando o método Inserção com vetores com muitos elementos.

Uma importante questão quanto à analise de tempo de execução surge da possibilidade de fatores externos ao programa influenciarem seu desempenho. Dito isso, foram executadas 21 conjunto de operações em vetores.Logo, há 21 resultados diferentes, estes são guardadas em um vetor que ao fim das execuções é ordenado, no caso do tempo de execução, e a mediana é selecionada para ser mostrada ao usuário. Já no caso dos números de comparações e movimentações, todo os valores do vetor são somados e divididos pelo número de testes, assim a média aritmética desses números é obtida e os mostrada ao usuário conforme o padrão de saída do programa. Dessa forma, a possibilidade de um processo interno da máquina, onde os testes estão sendo executados, interferir na execução do programa é diminuída. Além disso, todos os números gerados aleatoriamente, os números nos vetores ordenados de forma crescente e decrescente estão dentro do intervalo [1, n], onde n é o tamanho do vetor.

Finalmente, para comparar as variações do Quicksort serão utilizadas tabelas e gráficos para demonstrar visualmente os resultados obtidos. Os gráficos apresentados nesse documento seguem o seguinte padrão: No eixo $\boldsymbol{X}$: Tamanho do Vetor e no eixo $\boldsymbol{Y}$: Valor da métrica (Tempo em Microssegundos, Número de Comparações ou Número de

Movimentações). As análises serão feitas em blocos, de acordo com o tipo de ordenação do vetor, visto que os vetores ordenados de forma aleatória são os que melhor representam melhor o comportamento assintótico médio de cada variação, mas são nos vetores já ordenados, seja de forma crescente ou não, que os piores casos podem ser observados. Para fins didáticos e de melhor visualização nem todas as variações estarão presentes em um mesmo gráfico, isso porque há casos em que apenas uma variação desloca os eixos do gráfico de tal forma que é impossível diferenciar o comportamento das outras seis. Consequência disto, os gráficos possuem diferentes escalas no eixo ***Y***.

4.3. Estudo Comparativo

Nessa seção será discutida as potencialidades e fraquezas das variações de acordo com cada métrica analisada. Estas, como dito anteriormente, serão discutidas em diferentes tópicos para que a análise do comportamento de cada variação seja bem detalhada. As métricas observadas foram as seguintes:

4.3.1. Tempo de Execução

Primeiramente, será analisado comportamento das variações do Quicksort em vetores aleatórios. Nos gráficos da figura 5 é possível visualizar que a maioria das implementações possuem complexidade aproximadamente linear, salvo as variações Inserção 5% e Inserção 10%. A primeira, apresenta leve curvatura nos vetores maiores, isso por que a quantidade de elementos ordenados pelo método Inserção também fica maior de acordo com o vetor. Na segunda, isso é evidente, a variação Inserção 10% possui um complexidade muito próximo de $O(n^2)$, visto que a quantidade de elementos ordenado pelo Inserção é grande, esse é um comportamento esperado para essa variação.

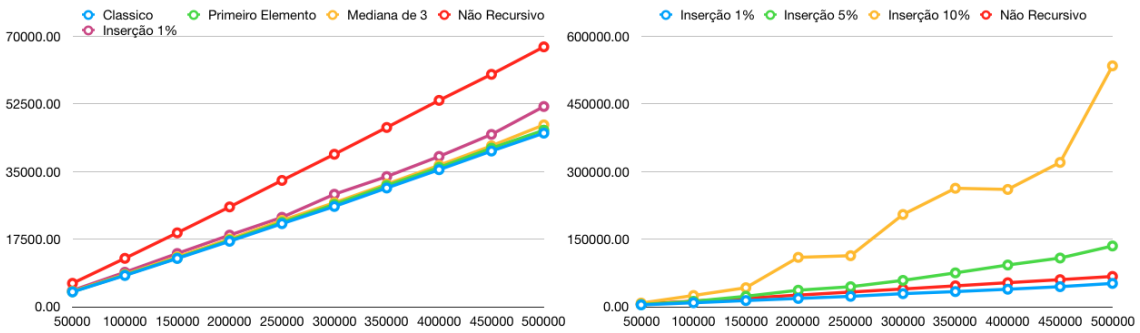

Figura 5. Tempo de Execução para Vetores em Ordem Aleatória

Importante observar que apesar de a variação Inserção 1% ser um pouco mais lenta que a maioria, pelo menos em vetores suficientemente grandes, ela ainda é melhor que a variação Não Recursiva durante um grande intervalo de tamanho de vetores. Esta possui um mal desempenho quanto ao tempo de execução quando comparada às outras implementações que não utilizam o método Inserção nenhuma vez. Isso se dá pelo gerenciamento de memória no uso da pilha implementada para essa variação, pois como será detalhado adiante o número de comparações e movimentações desta é semelhante às outras variações.

Por outro lado, nos vetores já ordenados é possível notar o pior caso do Quicksort, ou seja, quando o pivô selecionado em cada sub vetor é sempre o menor elemento deste e está na primeira ou na última posição do sub vetor. Esse caso caso ocorre tanto em vetores ordenado de forma Crescente, quanto em vetores de ordem Decrescente. Isso ocorre pois o número de comparações necessárias em cada sub vetor é sempre a maior possível, o tamanho n do sub vetor, além disso as partições ocorrem n vezes, assim como nas outras implementações. Logo, a variação Primeiro Elemento que obrigatoriamente segue esse padrão descrito possui complexidade assintótica quadrática, ou seja, $O(n^2)$ quando o vetor está ordenado.

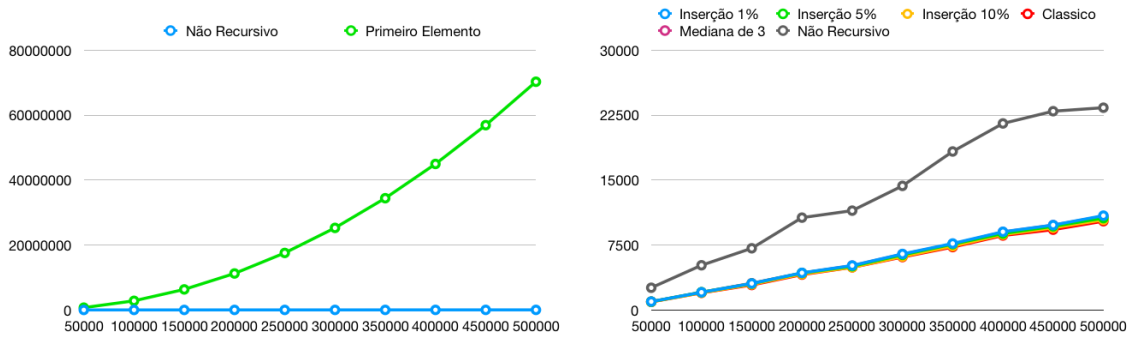

Figura 6. Tempo de Execução para Vetores em Ordem Crescente e Decrescente

É possível observar esse comportamento no primeiro gráfico da figura 6 na curva verde. A curva Azul, por sua vez, representa a variação Não Recursiva e é possível notar a grande diferença do tempo de execução entre essas duas implementações, visto que esta é a segunda com o maior tempo de execução, as outras seguem um padrão uniforme conforme apresentado no segundo gráfico dessa mesma figura. No segundo gráfico ainda, é possível observar que a maioria das variações seguem um mesmo padrão, ou seja, o mesmo do caso médio e melhor caso, $O(nlog(n))$ para vetores ordenados de Forma Crescente e Decrescente. Novamente, a variação Não recursiva tem seu comportamento diferente das demais, assim como ocorre em vetores aleatórios. Logo, parte da hipótese sobre o tempo de execução é conformada, visto que essa diferença independe da ordenação do vetor.

Importante observar também que o comportamento em tempo de execução é muito semelhante em vetores ordenados de forma crescente e decrescente, posteriormente será explicitado o motivo desse comportamento.

4.3.2. Número de Movimentações

Nesse subtópico e no próximo serão apresentados os dados que justificam os tempos de execução mostrados acima. O número de movimentações é uma importante métrica, pois mede quantas vezes o vetor precisou ser alterado até que ficasse ordenado. Os gráficos a seguir explicam a variação do tempo de execução para cada implementação do algoritmos.

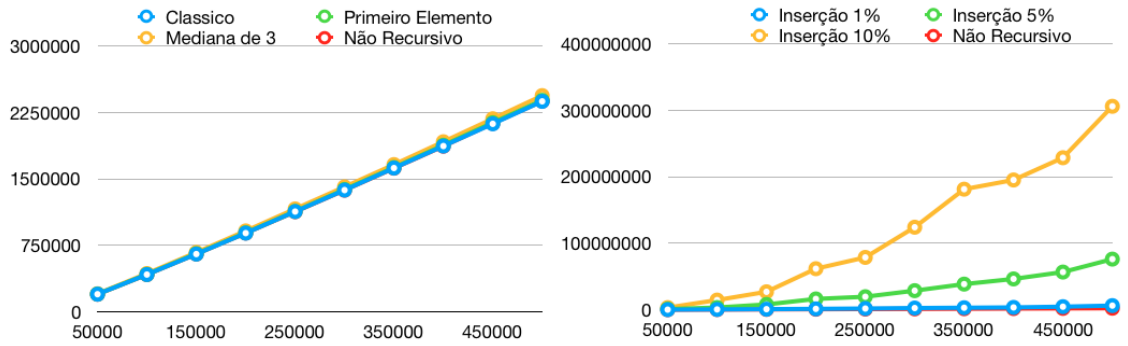

Figura 7. Número de Movimentações para Vetores em Ordem Aleatória

No segundo gráfico da figura 7, é possível observar que o comportamento das curvas das variações que utilizam o método Inserção para ordenar parte dos vetores é semelhante as curvas para o tempo de execução destas. Isso ocorre devido a relação direta do tempo de ordenação e a quantidade de vezes que vetor precisou ser alterado. Isso ocorre, pois enquanto maior o tamanho dos sub vetores, mais vezes o método inserção será utilizado e, como vimos anteriormente, este possuí complexidade assintótica. O comportamento semelhante entre as outras varições é facilmente explicado visto que todas elas utilizam o método partição para movimentação de chaves durante toda a ordenação do vetor e os vetores são sempre muito aleatórios.

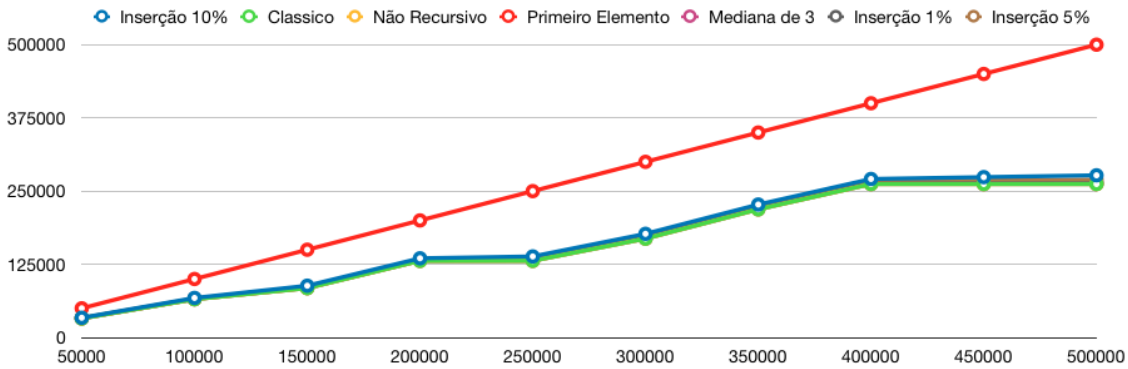

Figura 8. Número de Movimentações para Vetores em Ordem Decrescente

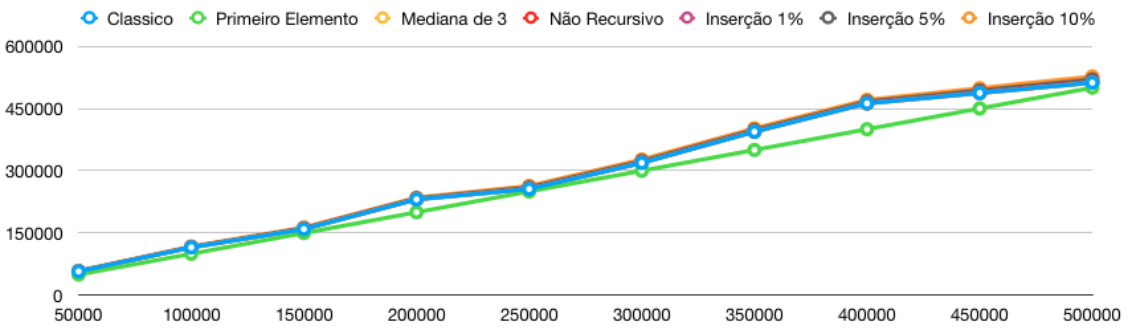

Figura 9. Número de Movimentações para Vetores em Ordem Decrescente

No caso dos vetores em ordem crescente e decrescente o comportamento das curvas novamente é semelhante,mas valores bem diferentes. Tanto na figura 8, quanto na figura 9 podemos observar que a varição Primeiro Elemento possui um comportamento estritamente linear. No caso do vetor ordenado de forma crescente, essa é uma grande desvantagem frente às outras implementações, porém no vetor ordenado de forma decrescente, essa linearidade oferece leve vantagem sobre as demais variações. Esse comportamento ocorre, pois a cada chamada do método partição só há uma ”movimentação”, a do pivô com ele mesmo.

4.3.3. Número de Comparações

Nesse subtópico serão apresentados mais dados para justificar o tempo de execução das variações do Quicksort. As comparações são peças importantes para definir o pior caso do Quicksort, tanto no vetor crescente, quanto no decrescente. Além disso, dados dos vetores aleatórios reforçarão os pontos abordados no subtópico anterior.

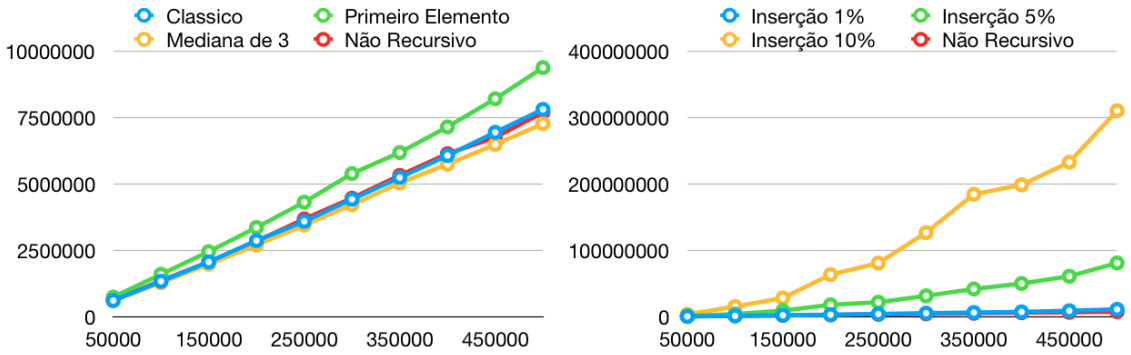

Figura 10. Número de Comparações para Vetores em Ordem Aleatória

Na figura 10, podemos observar, novamente a semelhança do comportamento das curvas nas variações que utilizam o método Inserção. Isso ocorre, como já explicado anteriormente, devido as características desse método. A novidade nessa situação é na variação Primeiro Elemento, visto que o comportamento da curva se diferencia das demais variações de acordo com o crescimento do número de elementos.

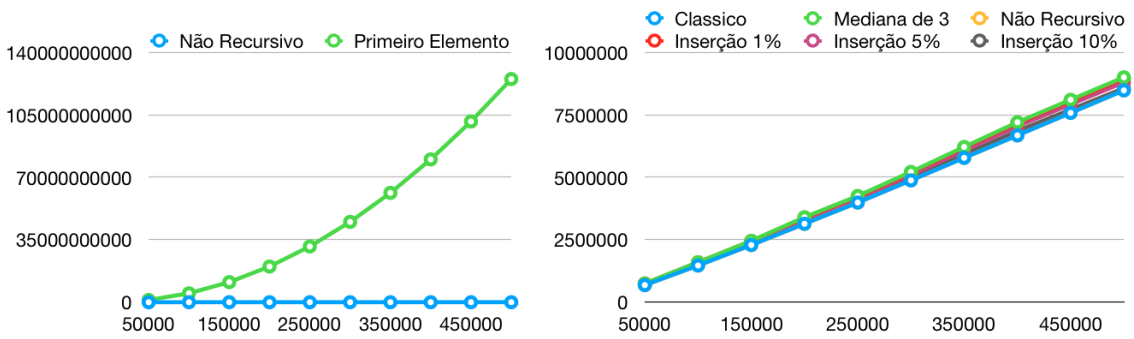

Figura 11. Número de Comparações para Vetores em Ordem Crescente

A figura 11, por outro lado, demonstra perfeitamente o comportamento quadrático da varição Primeiro Elemento, visto que a segunda condição do pior caso do Quicksort é justamente o vetor já estar ordenado, seja de forma crescente ou não. Esse comportamento ocorre pois a cada etapa do partição o pivô é comparado com todos os elementos antes de ser comparado com ele mesmo. Assim, sendo *n* o tamanho do vetor ou sub vetor, a cada chamada do partição há *n* comparações e durante o método ordena há, pelo menos, n chamadas do método partição. Assim, podemos verificar que a complexidade do pior caso do Quicksort é, de fato $O(n^2)$.

Uma vez que, nenhuma outra variação goza das mesmas caraterísticas da Primeiro Elemento e, dado um vetor ordenado, seu comportamento é bem parecido, mesmo que utilizado o método Inserção. Temos que as outras seis variações possuem números de comparações parecidos, conforme o segundo gráfico da figura 10.

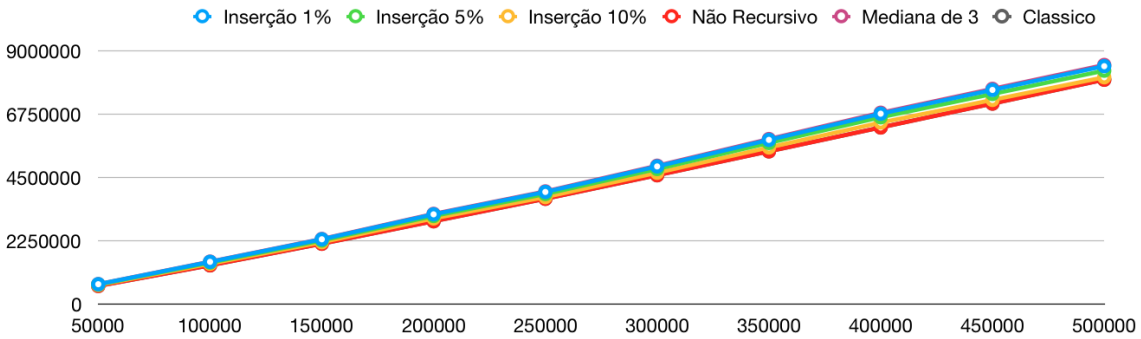

Figura 12. Número de Comparações para Vetores em Ordem Decrescente

A figura 12 representa o comportamento das variações quando os vetores estão organizados de forma decrescente. A variação Primeiro Elemento não se encontra nesse gráfico, pois ela possui exatamente o mesmo comportamento já apresentado no 1 gráfico da figura 11. Neste gráfico apenas é possível destacar que o número de comparações realizadas pelas variações que utilizam o método Inserção é inversamente proporcional ao tamanho dos vetores em que estes métodos são utilizados, pois em uma visão geral, essas curvas possuem as mesmas características das curvas já mencionadas no segundo gráfico da figura 11.

5. Conclusão

Este trabalho abordou sete diferentes implementações do algoritmo de ordenação Quicksort e analisou cada um deles sob diferentes métricas e utilizações. Logo, foi possível observar que algumas variações Quicksort destoam do seu comportamento esperado, mesmo nos piores casos. Além disso, foi possível comprovar empiricamente que o pior caso do desse algoritmo é de fato quando o vetor está ordenado e o elemento escolhido como pivô é sempre o primeiro, atingindo a complexidade assintótica de O(n^2), onde n é o tamanho do vetor.Além disso, foi possível verificar que enquanto mais o método Inserção for utilizado, pior será o desempenho do algoritmo. Por fim, é possível concluir que as variações mais estáveis sãs a clássica, a Mediana de 3 e a Não Recursiva, além de na maioria dos casos, obterem melhor desempenho, mesmo que a implementação da pilha no caso Não Recursivo possa prejudicá-lo em alguns casos.

Referências

[1] Wikipédia. *Quicksort*, https://pt.wikipedia.org/wiki/Quicksort. Acesso em: 6 Jun. 2019.

[2] Pedro Vasconcelos. *Ordenação Quicksort*, https://www.dcc.fc.up.pt/∼pbv/aulas/progimp/teoricas/teorica18.html. Acesso em: 8 Jun. 2019.

[3] Gustavo Pantuza. *O Algoritmo de Ordenação Quicksort*, https://blog.pantuza.com/artigos/o-algoritmo-de-ordenacao-quicksort. Acesso em: 9 Jun. 2019.

[4] Paulo Feofiloff. *Análise da ordenação por inserção*, https://www.ime.usp.br/∼pf/analise_de_algoritmos/aulas/insert.html Acesso em: 9 Jun. 2019.

[5] Paulo Feofiloff. *Pilhas*, https://www.ime.usp.br/∼pf/algoritmos/aulas/pilha.html Acesso em: 9 Jun. 2019.

[6] CPlusPlus. *Random*, http://www.cplusplus.com/reference/random/ Acesso em: 10 Jun. 2019.

[7] Wikipédia. *Profiling (computer programming)*, https://en.wikipedia.org/wiki/Profiling_(computer_programming) Acesso em: 10 Jun. 2019.