Documentação do segundo trabalho prático da disciplina de Estrutura de Dados

Lucas R. P. Machado[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

lucasresende@dcc.ufmg.br

Abstract. *This document has the objective of explaining the methodology and the results gathered during the tests proposed in the practical activity from the Data Structures course, which involves the comparison between different implementations of the sorting algorithm denominated Quicksort in several different scenarios.*

Resumo. *Esse documento tem como objetivo explicar a metodoloia utilizada e os resultados obtidos durante os testes propostos pelo trabalho prático da disciplina de Estrutura de Dados, que envolve a comparação entre diferentes implementações do algoritmo de ordenação denominado como Quicksort em diferentes cenários.*

1. Introdução

O trabalho se baseia em cima de um dos algoritmos de ordenação mais utilizados em problemas gerais, chamado de Quicksort. Esse algoritmo é tão bem sucedido devido a facilidade na sua implementação, seu fácil entendimento e sua complexidade computacinoal baixa.

O algoritmo funciona particionando o vetor a partir de um número que chamaremos de "pivô". Esse número pode ser escolhido de diversas formas, fato que é objetivo do trabalho mostrar qual dessas formas é a mais eficiente. Assim que o pivô é escolhido, o vetor é dividido em duas partes: elementos menores que o pivo vão para as posições à esquerda do pivô e os elementos maiores que o pivô vão para os posições à direita do pivô, deixando assim o pivô na sua posição correta. Logo após essa partição, o mesmo algoritmo é chamado para as duas metades criadas por esse processo, repetindo até que cada uma das partições contenha um elemento.

Quando cada partição possuí um único elemento, a pilha de execução ao voltar pela recursão, coloca cada partição à esquerda ou a direita da divisão que a criou, culminando em um vetor completamente ordenado.

Agora que a ideia geral do algoritmo foi exposta, o problema proposto pelo

trabalho envolve utilizar de sete diferentes implementações desse algortimo em três organizações possíveis de vetores e então comparar os resultados obtidos em cada um deles. As implementações que foram testadas são:

- Quicksort Clássico – A escolha do pivô se dá simplesmente escolhendo o elemento do meio do vetor.
- Quicksort Mediana de Três – A escolha do pivô se dá por meio da seleção do elemento do meio quando o primeiro, o último e o elemento do meio do vetor são comparados.
- Quicksort Primeiro Elemento – A escolha do pivô se dá simplesmente escolhendo o primeiro elemento do vetor.
- Quicksort Inserção 1% - Se utilizando da mesma escolha de vetor do método da mediana de três, quando o tamanho dos subvetores do Quicksort terem 1% do tamanho do vetor original, o algoritmo de ordenação Insertion Sort é utilizado para terminar de ordenar.
- Quicksort Inserção 5% - A mesma ideia da implementação anterior, porém ao invés de esperar até que os subvetores terem 1% do tamanho original, nessa implementação, os vetores deverão ter 5% do tamanho original para começarem a ser ordenador pelo Insertion Sort.
- Quicksort Inserção 10% - Mesma ideia do anterior, porém ordenando com o Insertion Sort a partir do momento que os subvetores terem 10% do tamanho original do vetor.
- Quicksort não Recursivo – Utilizando da seleção de pivo do elemento central do vetor, essa implementação não utiliza a recursão para funcionar. Uma pilha é utilizada para simular a sequẽncia de execuções da recursão.

Já os possíveis estados dos vetores inicais são:

- Ordernado Crescente
- Ordenado Decrescente
- Aleátorio

Para cada um das variações de implementação, serão gerados vetores de 50.000 até 500.000 (em intervalos de 50.000) elementos seguindo um dos tipos de vetores e serão salvos o número de comparações, o número de trocas de chaves e o tempo em microsegundos que foi necessário para a ordenação. Com isso podemos traçar quais as vantagens e desvantangens de cada implementação e talvez decidir em uma implmentação que seria a melhor entre todas as testadas aqui.

2. Implementação

Para o problema, foi escolhido uma abordagem por meio de orientação a objeto visto que além de um código melhor organizado e modularizado, com arquivos com funcionalidades mais específicas, deixaria mais fácil a expansão do escopo caso fosse necessário.

Com isso em mente, o projeto tomou a seguinte estrutura:

```
- src/
    - insertion_sort.hpp
    - main.cpp
    - quick_sort.hpp
    - sort_request.hpp
    - sort.hpp
- obj/
    - main.o
- Makefile
- main
```

Com isso, temos quatro classes principais no projeto:

- Sort (*sort.hpp*) – Classe abstrata que representa um algoritmo de ordenação genérico. Possuí um método abstrato *sort(long long, int, int)* que é o método que será chamado nas classes filhas para ordenar um vetor de valores do tipo *long long*. Também é responsável pela definição das variáveis e métodos que cuidam da contagem das comparações e movimentações de chaves que acontecerão durante a execução do ordenamento.

- InsertionSort (*insertion_sort.hpp*) – Subclasse da classe *Sort* e implementa o algoritmo de ordenação *Insertion Sort*. Esse algoritmo funciona de forma que, para cada posição do vetor, olhamos para o valor do elemento naquela posição e em seguida olhamos todos os elementos anteriores a ele. Quando o elemento que estamos o comparando for menor que ele, significa que ele deve ficar logo depois desse, caso contrário, seguimos percorrendo para a direita.
  Na implementação observamos que:
  ```
  (37)    for (int i = left + 1; i < right; i++){
  ```
  Controla qual a posição do vetor do elemento que colocaremos na posição ideal e que
  ```
  (47)    while (j >= left && arr[j] > key){
  (56)            j = j - 1;
  ```
  Controla o "andar" para a esquerda do elemento até que ele encontre um elemento que seja menor do que ele.

- QuickSort (*quick_sort.hpp*) – Subclasse da classe *Sort* e implementa o algoritmo discutido anteriormente, o *Quicksort*. Porém diferentemente da implementação do *Insertion Sort*, o algoritmo não está compactado inteiramente no método *sort(long long, int, int)* devido tanto a necessidade de modularização do código

para melhor entendimento tanto pelas restrições do projeto que necessitam diferentes implementações do algoritmo.
Inicialmente, uma instância dessa classe não é criada sem antes definirmos alguns parâmetros:

- *int* (*partition)(long long*, int, int)* – Função que recebe um vetor e retorna as duas posições referentes ao pivô.
- *double change_sort_perc* – Quantos por cento do tamanho inicial do vetor os subvetores precisam ter para que ele sejam ordenados pelo método da Inserção.
- *bool iterative* – Se o *Quicksort* utilizado será iterativo ou recursivo.

A classe também possui quatro métodos principais:

- *sort(long long, int, int)* – Método que realmente irá ser chamado e contém a condição para determinar se o vetor será ordenado por meio do *Quicksort* recursivo ou iterativo.
- *recursive_sort(long long, int, int)* – Método que define a implementação de todos os casos do *Quicksort* que serão recursivos. Esse método possuí basicamente duas partes separadas pelo resultado do *"if"* principal na linha *(180).* Essa é a condição que decide se a partir daquele momento a recursão será interrompida e o resto dos vetores que foram recebidos pela função serão ordenadas utilizando o *Insertion Sort* ou mais uma recursão será chamada. Caso o caminho da recursão for escolhido, o algoritmo funciona normalmente, escolhendo o pivô e chamando a sí própria para as duas metades criadas. Caso contrário, um objeto da classe *InsertionSort* é criada e utilizada para ordenar o vetor recebido.
- *iterative_sort(long long, int, int)* – Método que define a implementação do caso onde o *Quicksort* será iterativo ao invés de recursivo. Essa implementação se baseia na imitação de uma pilha de recursão utilizando uma pilha comum, onde cada elemento da pilha corresponde a um intervalo do vetor que será ordenado na ordem de execução correta. Porém, na implementação utilizada, essa pilha é construida de forma que a cada par de elementos, a posição inicial e final do subvetor é guardada (primeiro a posição inicial e depois a final) para propósitos de simplicidade do código. Não foi implementada uma condição de parada e troca para o algoritmo *Insertion Sort*, visto que não seria utilizada no âmbito desse trabalho.
- *partitionate_array(long long, int, int)* – Método responsável por particionar o vetor baseado em um pivô que será calculado baseado na função recebida pela classe ao ser instanciada. Retorna a primeira

posição à esquerda do pivô com um elemento diferente dele e a primeira posição do vetor à direita dele com um elemento diferente dele. Essa função como explicada anteriormente, coloca todos os elementos menores que o pivô escolhido à esquerda dele no vetor e todos os elementos maiores que o pivô à direita dele no vetor.

Essa classe também possuí os métodos secundários/de ajuda:

- *swap(long long& a, long long& b)* – Tem como objetivo trocar os valores entre as posições de memória de *a* e *b.*
- Funções estáticas que definem cada método de como o pivô será escolhido dado um vetor e retornam tanto a posição do pivô (na primeira posição do vetor de saída) e o número de comparações necessárias para a escolha desse pivô (na segunda posição do vetor de saída).

- SortRequest (*sort_request.hpp*) – Classe responsável por controlar todos os testes relacionados aos inputs recebidos. Ela possuí uma classe principal *run()*, que cria um objeto da classe *Quicksort* da variação especificada, gera 20 vetores do tipo correto para os testes e chama o método de ordenamento do *Quicksort* para cada um deles, armazenando o número de comparações e números de movimentos de chaves feitos além do tempo de execução. Ao final dos 20 testes, é feito a média dos números de comparações e número de movimentação de chaves e a mediana dos tempos de execução, que são exibidos ao usuário na saída padão.

Para fechar a implementação, temos o arquivo *main.cpp* que é responsável exclusivamente por receber as especificações tanto da variação do *Quicksort* e o tipo/tamanho do vetor e assim criar uma instância da classe *SortRequest* e chamar o método de controle *run().*

Alguns detalhes na implementação vem principalmente na contagem de comparações e movimentos de chave. Para o método *swap(long long&, long long&)* foi recomendado pelos monitores na página da disciplina que fosse contabilizado apenas uma movimentação de chaves para todo o método, e assim foi feito. Já em questão do número de comparações, foi tomado o devido cuidado a contabilizar os testes feitos durante a execução de cláusulas *while*/*for* (mesmo a comparação feita logo antes do *while*/*for* ser quebrado foi contabilizada) além dos testes triviais feitos por *if*. Também foram contabilizados o número de comparações necessárias para a escolha dos pivôs, como no caso da mediana de três.

3. Instruções de compilação e execução

Para compilar o programa, deve-se executar no terminal na pasta raiz do projeto:

```
make clean
make
```

E então para executa-lo basta executar o comando:

```
./main <variacao do quicksort> <tipo do vetor> <tamanho do vetor> [-p]
```

Sendo que as opções possíveis para os parâmetros são:

- Variação do Quicksort:
  - *QC* – Quicksort Clássico
  - *QM3* – Quicksort Mediana de Três
  - *QPE* – Quicksort Primeiro Elemento
  - *QI1* – Quicksort Inserção 1%
  - *QI5* – Quicksort Inserção 5%
  - *QI10* – Quicksort Inserção 10%
  - *QNR* – Quicksort não Recursivo
- Tipo do vetor
  - *Ale* – Aleatório
  - *OrdC* – Ordenado Crescente
  - *OrdD* – Ordenado Decrescente
- [-p]
  - Caso incluido no comando, os vetores utilizados para o teste serão impressos na saída padrão

4. Análise experimental

Depois da execução dos experimentos com cada uma das implementações, podemos traçar os gráficos comparando cada uma delas em relação a cada uma das métricas medidas. Isso foi feito por meio de scripts em Python, que facilita o processo de importar os dados e traçar os gráficos.

Uma observação a se levar em conta, é que todos os testes executados foram feitos em um notebook com processador Intel i5-4200U Quad-Core com 1.60GHz por núcleo utilizando o compilador *g++* no Ubuntu 18.04 com a cláusula de otimização *-O3*, logo os resultados do tempo que cada implementação levou para terminar de executar podem variar dependendo das especificações técnicas de seu computador.

4.1. Comparação da métrica de tempo

O tempo foi medido em cada uma das 20 execuções de uma das configurações e no

final, o tempo usado para esses gráficos é a mediana desses 20 tempos. Isso é feito para evitar que execuções sortudas ou azaradas influênciassem demais as medidas.

4.1.1. Vetor ordenado crescente

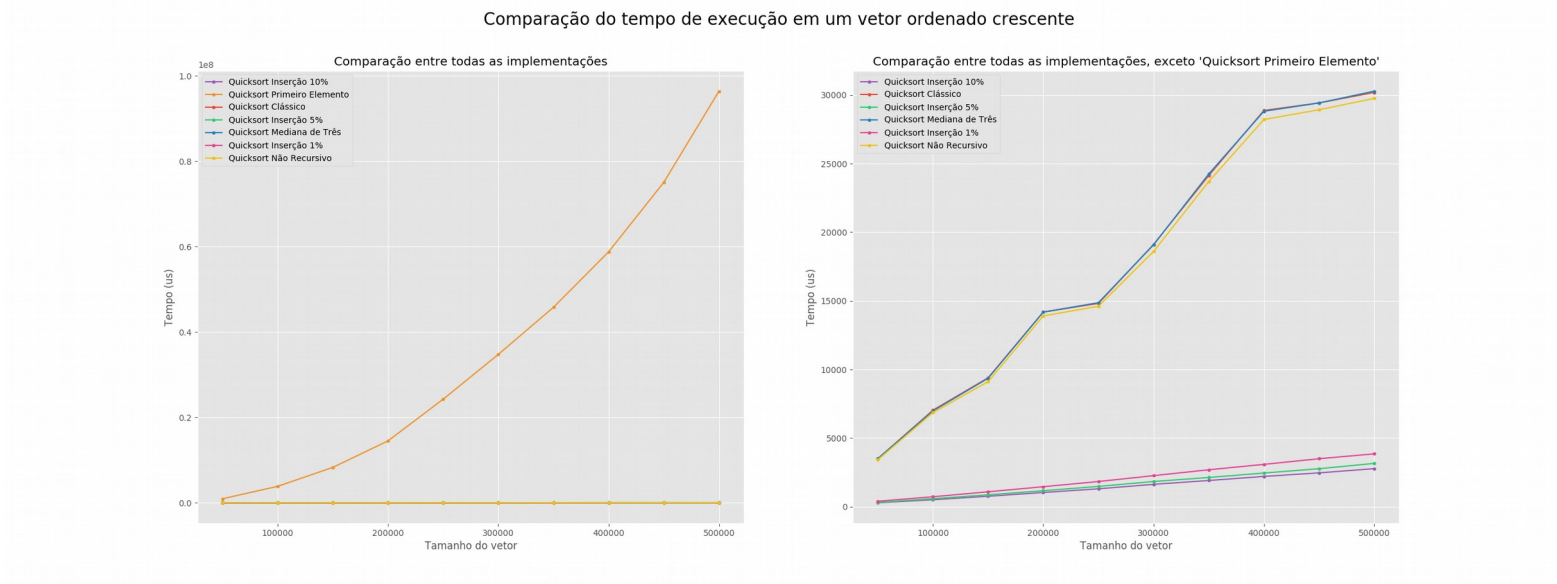


Figura 1 – Comparação dos tempos de execução em um vetor ordenado crescente dentre cada uma das implementações

Como pode-se observar, a implementação onde o pivô escolhido foi o primeiro elemento do vetor teve resultados muito piores que as demais implementações. Isso é o comportamento esperado visto que escolhendo o pivô na primeira posição de um vetor ordenado significa que o pivô escolhido vai ser sempre o menor elemento, e quando as partições forem feitas, a partição da esquerda ficará vazia (chamamos isso de partição degenerada) visto que não existe elemento menor que ele, e a partição da direita terá todos os outros elementos do vetor. Nesse caso, o algoritmo do Quicksort se encontra no seu pior caso, que tem complexidade $O(n^2)$, se comportando da maneira que se observa no gráfico.

Quanto às implementações que usam o Inserção a partir de certo ponto na ordenação, é esperado que elas sejam as mais rápidas. Isso se dá pois o algoritmo *Insertion Sort* tem complexidade de tempo linear para vetores já ordenados, que é melhor que a complexidade do *Quicksort* no mesmo caso - $O(n\log(n))$. É possível perceber que os resultados no gráfico respeitam essa lógica visto que o melhor tempo de execução vem da implementação que utiliza o *Insertion Sort* mais cedo (Insertion 10%).

Já as outras implementações seguem praticamente juntas, visto que em geral nenhuma tem vantagem sobre a outra. O Quicksort clássico sempre criará ambas as partições com números de elementos iguais (que é o melhor caso para o algoritmo), o Quicksort mediana de três vai se comportar da mesma forma que o Quicksort clássico visto que na escolha do pivô, ao fazer a mediana entre o primeiro, o último e o elemento do meio, o elemento do meio sempre vai ser o escolhido, fazendo o pivô ser

sempre igual à outra implementação. Quanto ao Quicksort não recursivo, ele possuí uma vantagem marginal sobre os outros dois de seu grupo visto que recursões são geralmente mais custosas ao computador do que um algoritmo iterativo, mesmo que esse algoritmo simule a pilha de execução da recursão.

4.1.2. Vetor ordenado decrescente

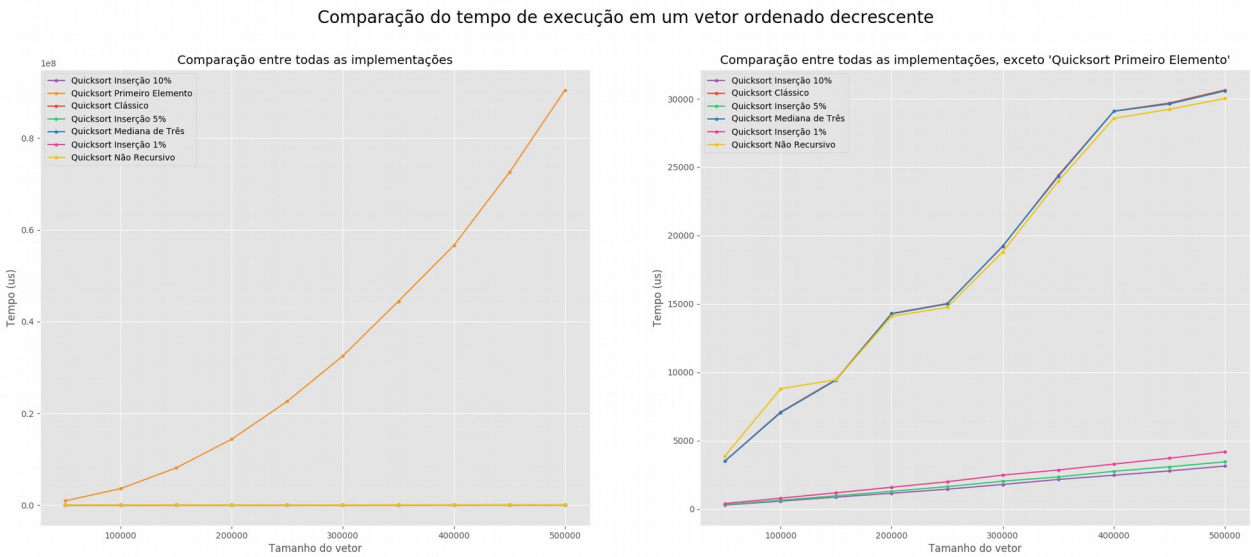


Figura 2 – Comparação dos tempos de execução em um vetor ordenado decrescente dentre cada uma das implementações

O caso para o vetor ordenado decrescente é praticamente idêntico ao caso do vetor crescente, tanto pelos resultados dos gráficos tanto pelo funcionamento dos algoritmos. Uma diferença que válida ser citada é a do Quicksort Primeiro Elemento. No caso do vetor crescente, o pivô escolhido era sempre o menor elemento do vetor. No caso atual é o inverso, o pivô é sempre o maior elemento do vetor. Essa diferença não gera mudança no tempo de execução pois a partição degenerada ainda será criada, porém ao invés de estar do lado esquerdo do pivô, estará do lado direito, não afetando em nada a performance do algoritmo. Essa similaridade com os vetores ordenados crescentes se manterá em geral para as outras métricas.

4.1.3. Vetor aleatório

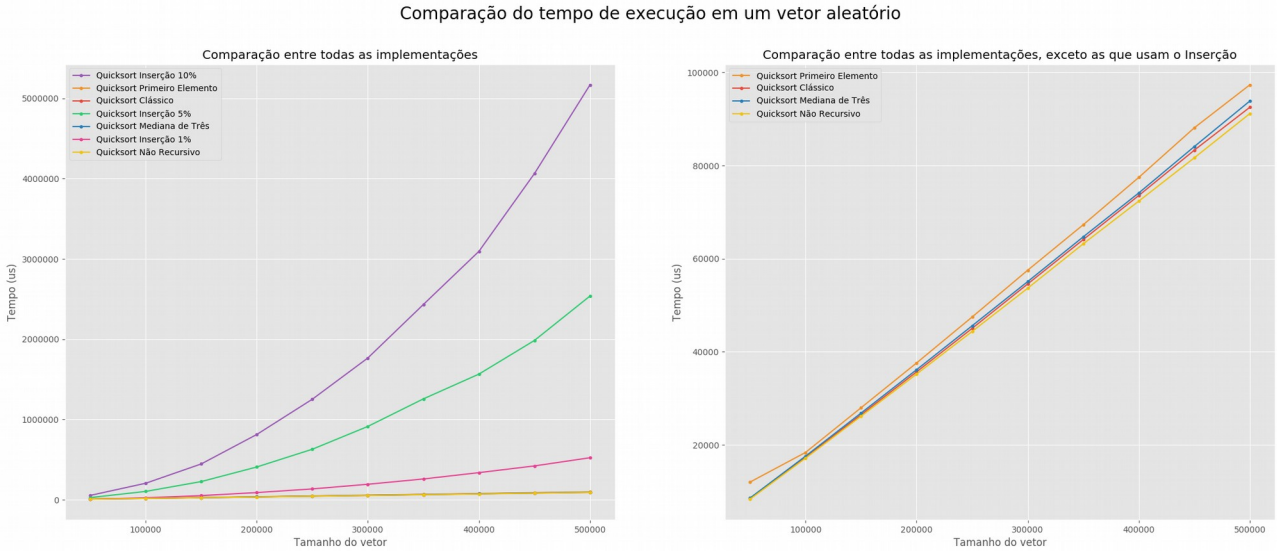


Figura 3 – Comparação dos tempos de execução em um vetor aleatório dentre cada uma das implementações

Em um vetor aleatório a similaridade com os vetores ordenados já não é tão presente. Nesse caso as piores implementações são aquelas que utilizam o *Insertion Sort* para ordenar o vetor a partir de um ponto. Isso é contra intuitivo visto que uma das otimizações do Quicksort é usar esse método. O problema aqui é que mesmo a implementação mais conservadora em quando utilizar o Inserção – o Inserção 1% - para o caso mais fácil - o de 50.000 elementos - começa a ordenar utilizando o novo algoritmo de ordenação quando os subvetores tem 500 elementos, o que é muito grande para que um algoritmo de complexidade quadrática ser eficiênte. Em implementações realmente otimizadas do Quicksort, o inserção é utilizado apenas em vetores geralmente menores que 15 elementos. Isso explica a baixa eficiência dessas implmentações.

Quanto ao restante dos algoritmos, pode-se ver que nenhum tem uma grande vantagem sobre o outro. Vale ressaltar que apesar de por pouco, o Quicksort Primeiro Elemento é o pior dentre todos os outros visto que ele não faz testes antes de escolher o pivô, logo é possível que o pivô escolhido seja um dos menores ou maiores elementos do vetor e isso implica que o algoritmo vai estar executando próximo ao seu pior caso de complexidade quadrática (ou até mesmo nele). Outro detalhe é a marginal vantagem do Quicksort não Recursivo visto que, como dito antes, um algoritmo não recursivo geralmente é ligeiramente mais rápido que um algoritmo recursivo.

4.2. Comparação do número de comparações realizados

Nesse teste, para cada uma das 20 execuções, foram armazenados o número de comparações feito por cada algoritmo. Os resultados a seguir se referem a média do resultado das 20 execuções.

4.2.1. Vetor ordenado crescente

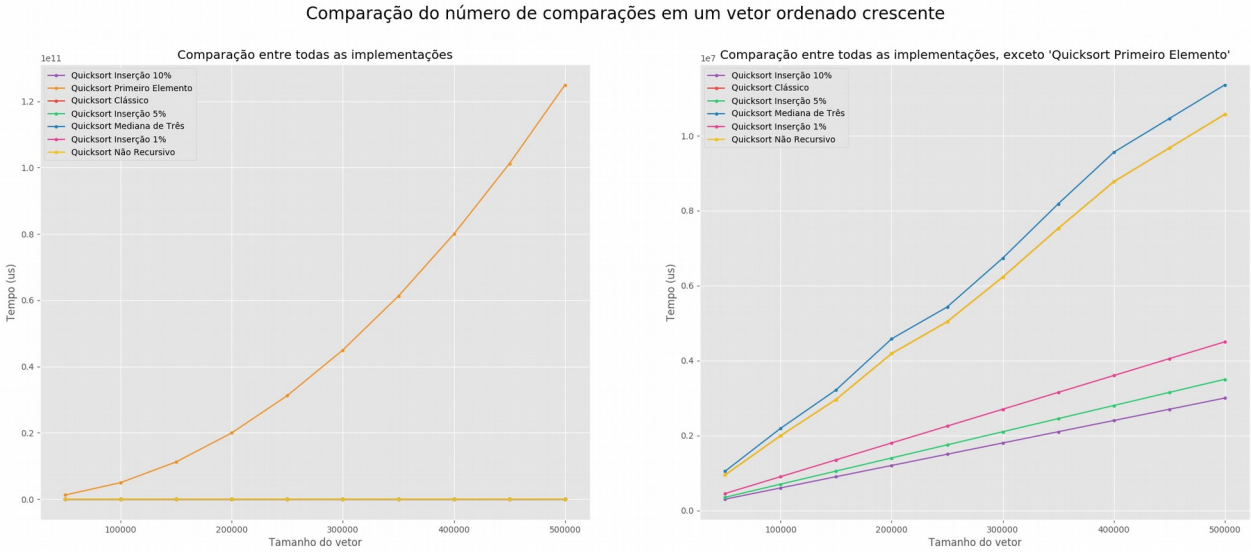


Figura 4 – Comparação dos números de comparações em um vetor ordenado crescente dentre cada uma das implementações

De forma muito parecida à analise de tempo, o Quicksort Primeiro Elemento é o pior dentre todos, os métodos que usam o Inserção a partir de um ponto são os melhores e o restante se encontra agrupado em um meio termo entre os dois, tendendo mais ao melhor deles.

O Quicksort Primeiro Elemento é ruím basicamente pelo mesmo motivo da analise de tempo: como ele cria partições degeneradas, não existe a vantagem de ir diminuindo pela metade o tamanho do vetor em cada iteração da recursão. O que acontece é que em cada iteração, a recursão é chamada com um vetor de tamanho $n-1$ já que o único elemento que não está presente nesse novo vetor é o pivô. Logo o cálculo do pivô, que já é linear em função do número de comparações, vai ser repetido n vezes, tornanando o número de comparações uma função de complexidade quadrática.

Já os métodos que utilizam o inserção são bons também pelo mesmo motivo do caso da analise de tempo. Como o método da inserção tem complexidade linear em função do número de comparações feitas em um vetor já ordenado, isso se torna melhor que usar o Quicksort até o final, fazendo com que essas implementações tenham vantagem sobre as outras que não usam o algoritmo da inserção em algum ponto de sua execução.

Os outros métodos sempre tem complexidade $O(n\log(n))$ de comparações e nunca cairão no pior caso de escolher o pivô como o maior ou o menor elemento visto que sempre escolherão o elemento do meio no vetor ordenado. No gráfico não é possível ver o traço do Quicksort Clássico visto que ele coincide exatamente com o traço do Quicksort mediana de três, já que ao fazer a mediana de três, o elemento do

meio sempre será escolhido, que é o método que o clássico usa. Denovo é possível ver a pequena vantagem que o algoritmo iterativo tem sobre os recursivos.

4.2.2. Vetor ordenado descrescente

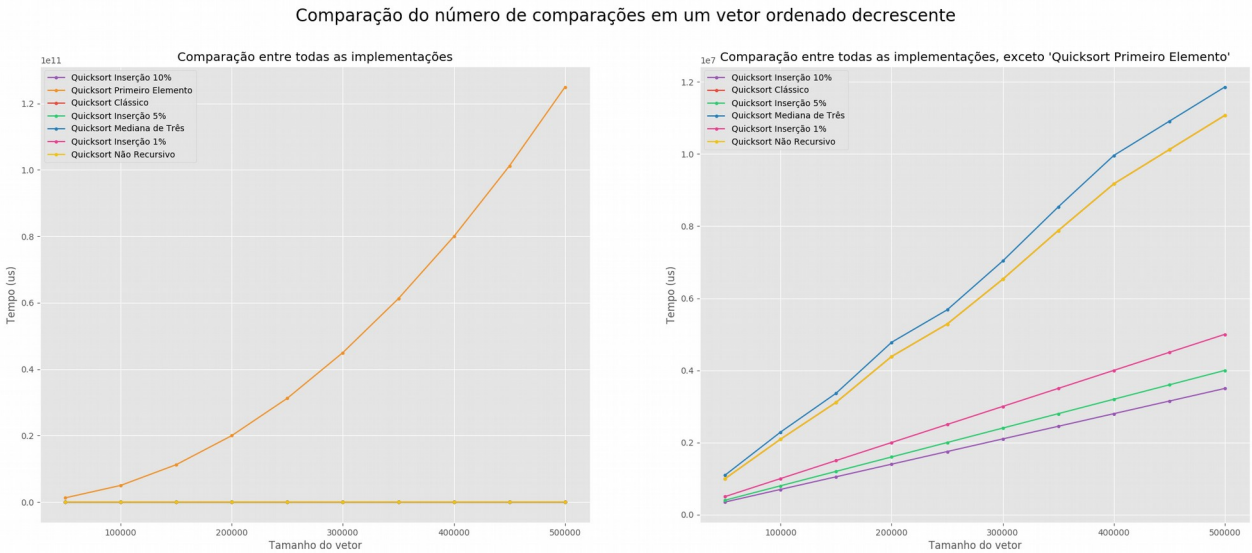


Figura 5 – Comparação dos números de comparações em um vetor ordenado decrescente dentre cada uma das implementações

Esse caso é análogo ao caso anterior do vetor ordenado crescente. O único comentário é o fato discutido também na seção 4.1.3 onde ao invés do Quicksort Primeiro Elemento escolher sempre o menor elemento do vetor, nesse caso ele sempre escolherá o maior elemento, porém isso não altera o fato dele estar no seu pior caso, de complexidade quadrática.

4.2.3. Vetor aleatório

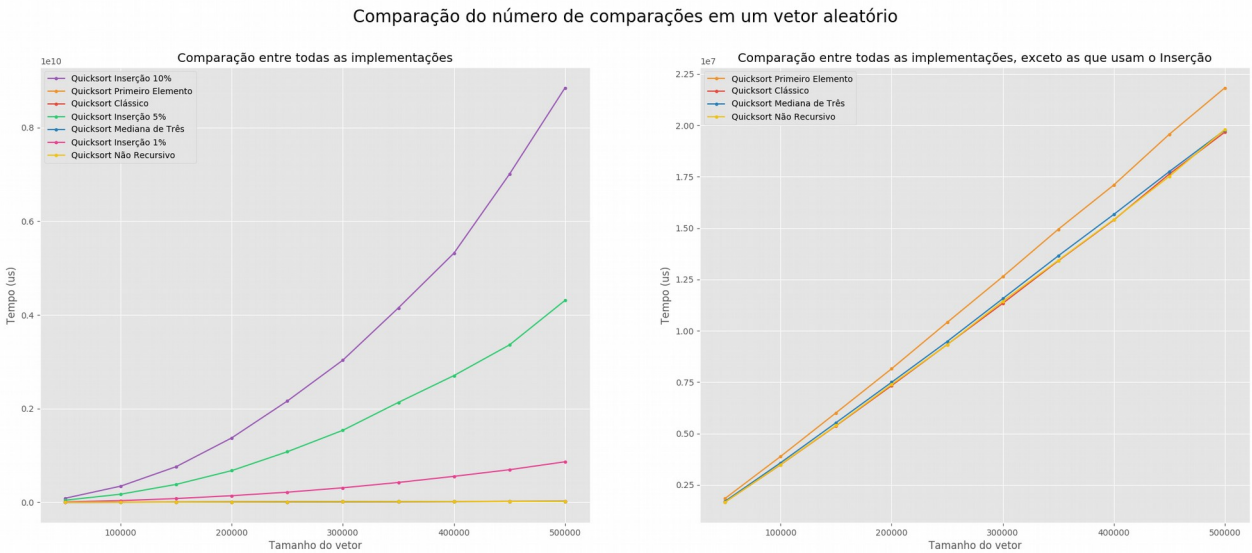


Figura 6 – Comparação dos números de comparações em um vetor aleatório dentre cada uma das implementações

No caso de vetores aleatórios os piores se tornam os algoritmos que utilizam o inserção. Como discutido na seção 4.1.3 onde isso também ocorre, o motivo disso apesar do uso do algoritmo Inserção ser considerado uma otimização é que mesmo no melhor caso (vetor de 50.000 elementos e a implementação Inserção 1%) o Inserção começa a ser utilizado em vetores de tamanho 500, o que é muito grande para um vetor quadrático ser utilizado. Em implementações otimizadas, o Inserção só começa a ser usado geralmente em vetores de tamanho menor que 15 elementos.

O Quicksort Primeiro Elemento ainda é o pior desconsiderando os que usam o Inserção visto que é normal que, sem nenhum teste como o da mediana, ele acabe escolhendo um pivô ruim e acabe ficando próximo ou até no pior caso do algoritmo, o caso de complexidade quadrática.

As outras implementações, visto que nenhuma tem vantagens aparentes sobre a outra ficam juntas em um grupo.

4.3. Comparação do número de movimentos de chaves

Nesse teste, para cada uma das 20 execuções, foram armazenados o número de movimentações de chave feito por cada algoritmo. Os resultados a seguir se referem a média do resultado das 20 execuções.

4.3.1. Vetor ordenado crescente

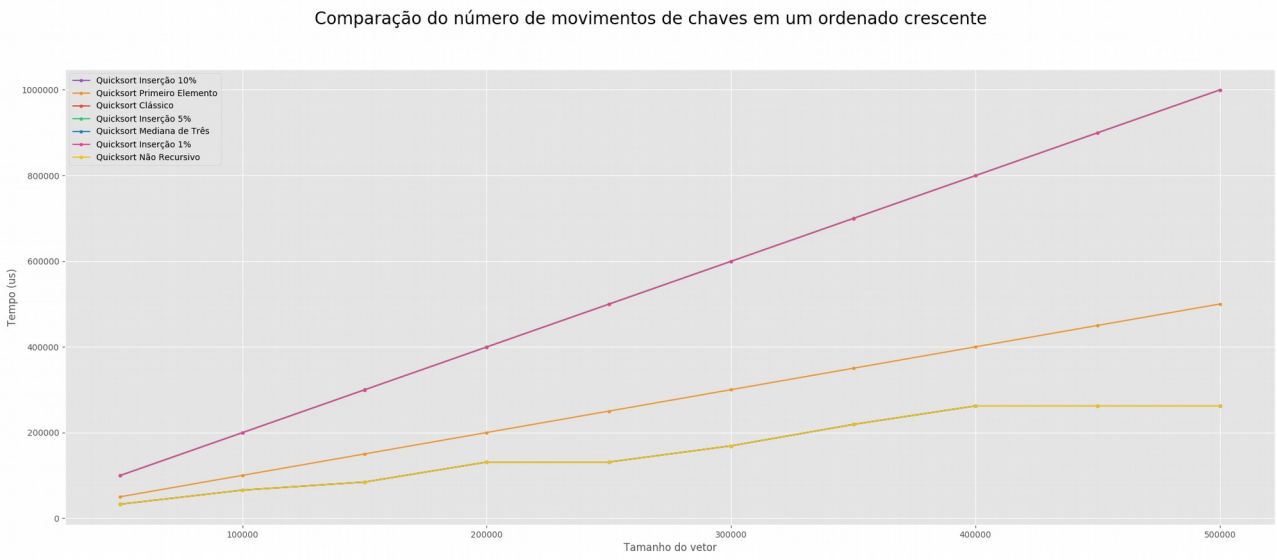


Figura 7 – Comparação dos números de movimentos de chave em um vetor ordenado crescente dentre cada uma das implementações

Nesse caso, temos três grupos de implementações que tem resultados praticamente iguais.

Um dos grupos (traçado pelo rosa do Inserção 1% no gráfico) é composto pelos três algoritmos que utilizam o inserção em algum ponto da execução. O número de movimentos não é exatamente igual entre eles visto que o inserção também faz movimentos e dependendo de quando essa mudança de algoritmos acontecer, essa

diferenciação pode resultar em um número ligeiramente maior ou menor de movimentações, como pode ser visto na aproximação do gráfico anterior:

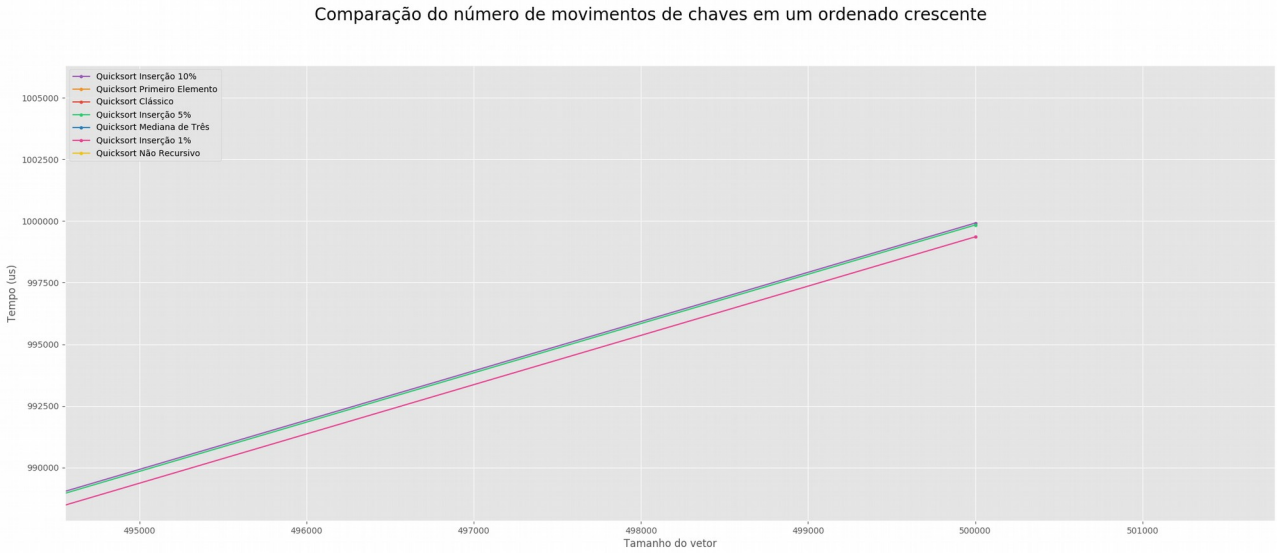


Figura 8 – Zoom na comparação dos números de movimentos de chave em um vetor ordenado destacando os algoritmos que usam o inserção

Essas implementações são as piores pelo mesmo motivo já dito antes, que como os vetores são grandes, começar a usar o inserção, mesmo que nos vetores de 1% do tamanho original, resulta em um uso do algoritmo quadrático em vetores grandes. E como a movimentação de chaves no inserção também é quadrática, isso gera um maior número de movimentações nesse caso também.

Já o Quicksort Primeiro Elemento é o próximo, sozinho em seu grupo que fica entre o grupo das melhores implementações e das piores. Isso acontece pois apesar dele não possuir o número quadrático de movimentações, as partições degeneradas fazem com que o número total de iterações da recursão gerem um número maior de movimentos.

O último e melhor grupo, composto pelo Quicksort não recursivo, mediana de três e o clássico são os melhores pois unem o fato de nunca cairem no pior caso e não possuírem parte quadrática do inserção.

4.3.2. Vetor ordenado descrescente

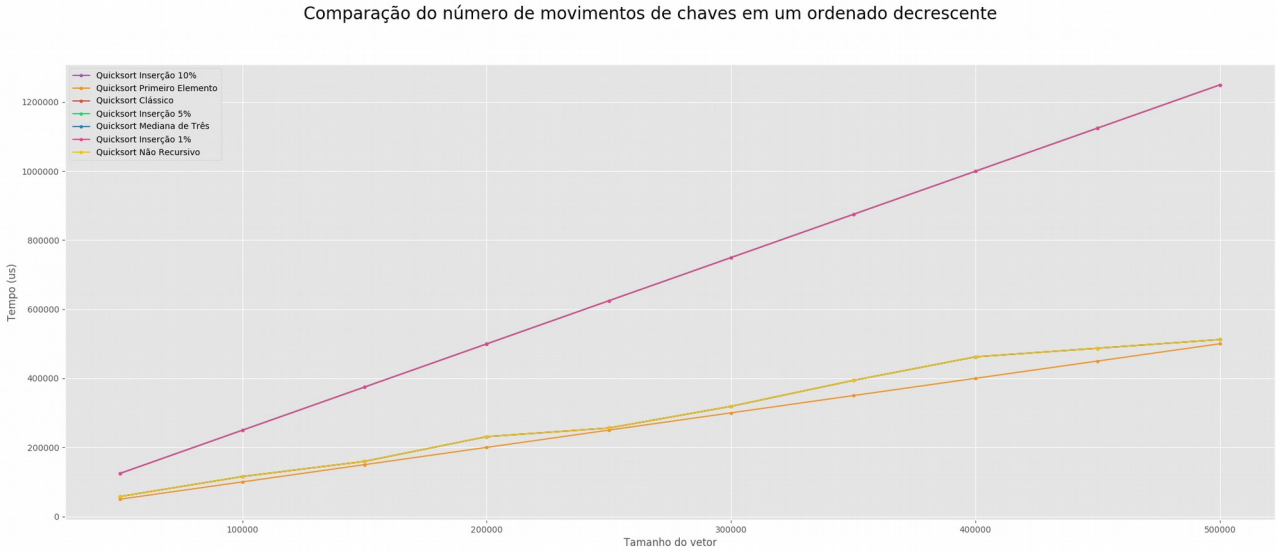


Figura 9 – Comparação dos números de movimentos de chave em um vetor ordenado decrescente dentre cada uma das implementações

No caso onde o vetor é ordenado decrescente, os algoritmos que utilizam o inserção a partir de certo ponto continuam os piores. E da mesma forma que no caso anterior, eles são muito próximos entre sí mas não exatamente iguais, como o zoom mostra:

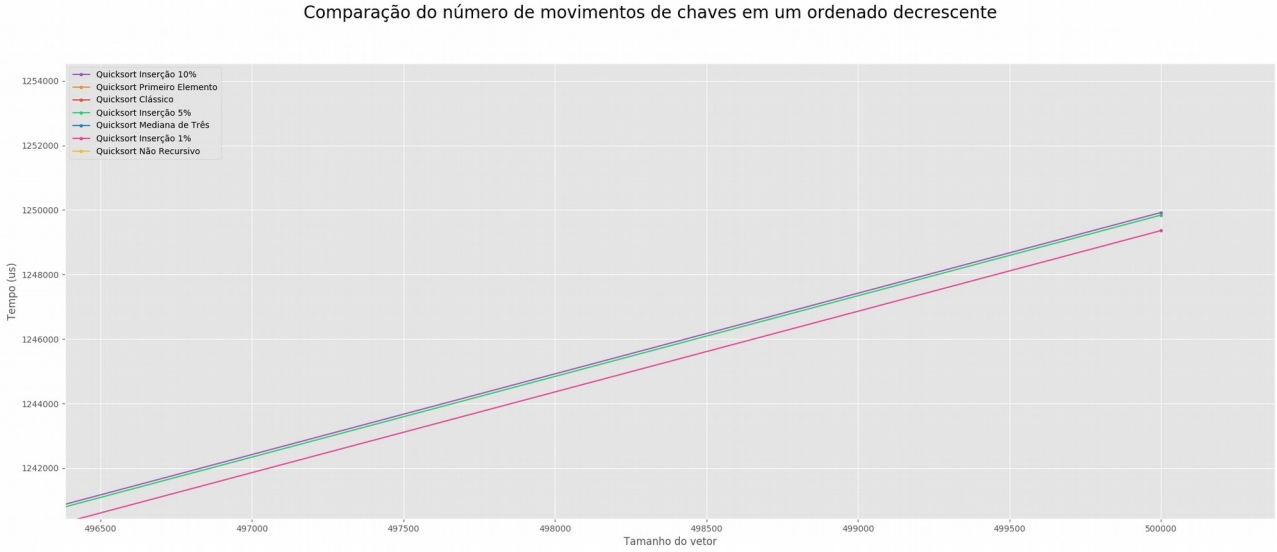


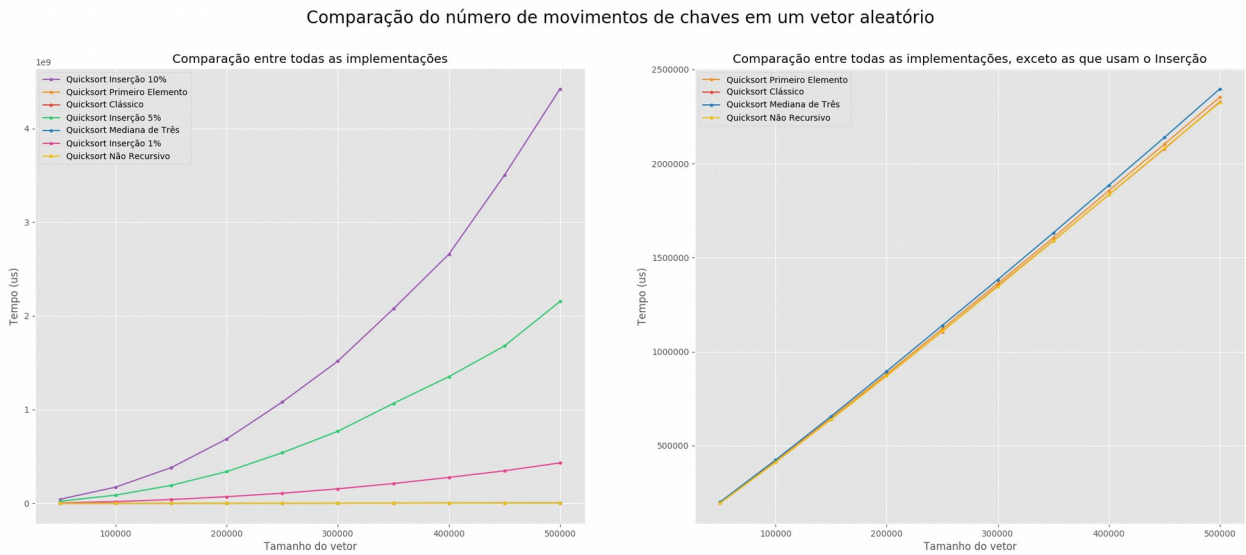
Figura 10 – Zoom na comparação dos números de movimentos de chave em um vetor ordenado destacando os algoritmos que usam o inserção

Quanto às implementações que não utilizam o inserção, agora houve uma inversão. O Quicksort Primeiro Elemento, que era o pior entre os outros agora se tornou o melhor. Isso se da pois na verdade, o Quicksort Primeiro Elemento, como era de se esperar, não mudou seu número de movimentos (é possível ver que nos gráficos dessa seção e na seção anterior, o número de movimentos dessa implementação é praticamente idêntico um ao outro), porém as outras implementações (clássico, mediana de três e recursivo) nesse caso gastam mais movimentações do que no caso anterior.

Esse aumento do número de trocas para as implementações que usam o pivô no

meio do vetor ordenado se deve ao fato de que, para toda iteração da recursão (ou no caso do não recursivo, da iteração do *while*), ao invés de verificar que tudo já está na posição ideal, os algoritmos tem que trocar todas as vezes os elementos da direita do pivô com os da esquerda, aumentando o total de trocas.

4.3.3. Vetor aleatório



Nesse tipo de vetor o resultado se parece muito com o resultado com os resultados das outras métricas no mesmo tipo de vetor. As implementações que usam o inserção sendo consideravelmente piores por usarem o algoritmo em vetores grande demais enquanto as outras implementações estão juntas em um grupo praticamente juntas visto que a diferença na escolha do pivô não é um fator tão grande nessa métrica.

5. Conclusão

É difícil escolher uma das implementações como a melhor de todas. Essa escolha depende muito do tipo de dados que o problema de oferece. Porém podemos tirar conclusões interessantes como a importância de definir um bom limite para a troca para o inserção visto que em diversos casos a escolha inoportuna desse limite fez esse método ser o pior de todos.

Outra informação a ter em mente é a importância de definir um bom pivô, principalmente quando existe uma chance grande dos dados de entrada levarem o algoritmo ao pior caso, como é possível ver quando observamos os desempenho da implementação usando o primeiro elemento como pivô em vetores já ordenados.

Outra métrica não tão comum de fazer diferença, mas que em sistemas mais específicos pode vir a ser importante é o custo de comparações e movimentos de memória. Como foi possível ver, algumas implementações são significamente mais eficiêntes caso essas métricas forem importantes.

Porém de forma geral, é possível ver que uma escolha inteligente do pivô (com

o método da mediana de três por exemplo), uma implementação iterativa e possívelmente a troca para o método da Inserção depois de um limite de tamanho bem estabelecido pode ser a diferença de um algoritmo bem otimizado e um que frequentemente cai no seu pior caso.

Uma das dificuldades encontradas durante o desenvolvimento dos testes é o tempo considerável para a execução de alguns testes, principalmente os envolvendo o método que escolhe o pivô como o primeiro elemento nos vetores ordenados crescentemente e decrescentemente, chegando a tomar mais de 6 horas de execução. Isso é um problema principalmente visto que com o tamanho dos vetores que precisamos gerar, foi necessário descobrir depois de um bom tempo de testes que o limite da memória stack do Ubuntu não era o suficiente, gerando erros e causando perda de execuções.

Algo a ser considerado também é a incoerência gerada pelo que foi dito sobre o número de movimentos da função *swap* utilizada pela método de partição do Quicksort e o número de movimentos no Insertion Sort. No método *swap* fomos instruidos a considerar todo o método como um movimento de chaves, sendo que existem duas movimentações de posições: uma para colocar o valor de *a* em *b* e outro para colocar o valor de *b* em *a*. Isso gera uma incoerência visto que no método do Insertion Sort fazemos uma atribuição de uma posição de um vetor na outra, ou seja, objetivamente apenas uma movimentação e o contabilizamos também como uma movimentação de chaves sendo que em teoria temos uma movimentação a menos do que no método *swap*. Mas visto que essa quantidade não mudaria os resultados de forma geral, não é um problema.

6. Referências bibliográficas

Halim, S., Halim, F. (2011) “Visualgo”, Acesso em Junho, 2019: https://visualgo.net/bn/sorting

Clodoveu A. (2012) “Quicksort”, Acesso em Junho, 2019: https://homepages.dcc.ufmg.br/~clodoveu/files/AEDS2/AEDS2.12%20Quicksort.pdf

Coelho, H., Félix, N. (2004) “Métodos de Ordenação: Selection, Insertion, Bubble, Merge (Sort)”, Acesso em Junho, 2019: http://www.inf.ufg.br/~hebert/disc/aed1/AED1_04_ordenacao1.pdf

Jain, S. “GeeksforGeeks”, Acesso em Junho, 2019: https://www.geeksforgeeks.org/

Cormen, T., Balkcom, D. “Analysis of Quicksort”, Acesso em Junho, 2019: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Wayne, K., Sedgewick, R. “Quicksort”, Acesso em Junho, 2019: https://algs4.cs.princeton.edu/lectures/23Quicksort.pdf