UNIVERSIDADE FEDERAL DE MINAS GERAIS

Departamento de Ciência da Computação

Bacharelado em Sistemas de Informação

Jean George Alves Evangelista

Matrícula: 2018047021

Estruturas de Dados

Trabalho Prático 2: Biblioteca Digital de Arendelle

Belo Horizonte

Junho/2019

1 INTRODUÇÃO

O Quicksort é um algoritmo de ordenação bastante rápido, eficiente e utilizado para as mais diversas aplicações. Este trabalho tem como objetivo implementar o Quicksort clássico e algumas de suas variações com o intuito de realizar uma análise comparativa, avaliando os melhores e piores casos do algoritmo.

As seguintes implementações deveriam ser realizadas: Quicksort clássico, Quicksort mediana de três, Quicksort primeiro elemento, Quicksort inserção 1%, Quicksort inserção 5%, Quicksort inserção 10% e Quicksort não recursivo. Cada tipo de Quicksort deve ser avaliado utilizando vetores crescentes, decrescentes e aleatórios, com tamanho variando entre 50 mil e 500 mil elementos (em intervalos de 50 mil elementos). Os testes devem ser repetidos no mínimo 20 vezes para obter-se a média de comparações, média de movimentações e mediana do tempo de execução.

A versão final do programa implementado faz exatamente o que está descrito no parágrafo anterior. O tipo do quicksort, tipo do vetor e tamanho do vetor são recebidos via parâmetro. Em seguida, um vetor do tipo e tamanho estabelecidos é ordenado 21 vezes e no fim calcula-se as médias e mediana do tempo. É possível também visualizar os vetores utilizados em cada ordenação, passando um parâmetro adicional. Por fim, os dados são mostrados na tela. Uma descrição melhor da implementação é realizada a seguir.

2 IMPLEMENTAÇÃO

Optou-se por realizar a implementação na linguagem de programação C++, por conta da maior familiaridade com a mesma. O programa possui cinco classes: Cell, Stack, Utils e Quicksort. Tais classes são explicadas de forma detalhada abaixo.

Classes

Cell: Classe que representa cada célula da pilha. Possui como atributos um *dado,* um ponteiro para a próxima célula e um ponteiro para a célula anterior.

Stack: Abstração de uma pilha, necessária para a implementação do Quicksort não recursivo. Possui como atributo uma célula que representa o topo da pilha. Além dos métodos construtor e destrutor, possui um método para inserir no topo da pilha, para remover do topo da pilha e um método que verifica se a pilha está vazia.

Utils: Uma classe que reúne métodos que se mostraram úteis durante a implementação. Nesta classe estão presentes os métodos de geração de número aleatório, geração de vetor crescente, geração de vetor decrescente, geração de vetor aleatório e um método para printar os dados de um vetor.

Quicksort: Classe que reúne todas as variações do Quicksort, bem como o número de movimentações e comparações. Uma explicação melhor de cada variação do Quicksort é dada na seção a seguir.

Cabe salientar que foram colocados contadores para armazenar o número de comparações e movimentações. Além disso, o tempo é calculado utilizando como base o exemplo disponível na especificação. A geração de números aleatórios foi feita através de métodos já presentes na biblioteca padrão do C++.

Quicksort e variações

*Quicksort clássico***:** O Quicksort clássico foi a base de todas as variações (com exceção do não recursivo), a implementação realizada foi baseada nas transparências da disciplina. O Quicksort clássico faz uso de duas funções: *order* e *partition*, sendo que a segunda chamará a primeira. O vetor é particionado, tendo como pivô o elemento central. Em seguida, os elementos menores que o pivô são alocados à sua esquerda e os maiores à sua direita. Após isso, o processo de partição de é repetido, sendo escolhido um novo pivô e realizando-se novas trocas. O processo é repetido várias vezes (chamadas recursivas) até que no fim o vetor esteja ordenado.

Quicksort mediana de três: Implementação semelhante ao Quicksort clássico, mudando apenas na escolha do pivô, que é selecionado analisando-se o valor mediano entre o elemento na primeira posição, na posição central e na última posição. Esta implementação é interessante por evitar o pior caso do Quicksort, quando o pivô escolhido é o menor ou o maior elemento do vetor.

Quicksort primeiro elemento: Implementação semelhante ao Quicksort clássico, mudando apenas a escolha do pivô, que é selecionado sendo sempre o primeiro elemento do vetor. Esta implementação não é interessante por permitir que o pior caso aconteça.

Quicksort inserção 1%: Esta variação se comporta como um Quicksort mediana de três até o momento em que o subvetor atual tenha quantidade igual ou inferior a 1% do vetor original, quando então o vetor é ordenado por meio do método de inserção. O método de inserção ordena o vetor e retorna por parâmetro a quantidade de movimentações e comparações. Este tipo de implementação se mostra bastante interessante, uma vez que a ordenação por inserção trabalha melhor com vetores com poucos elementos. A implementação do algoritmo de inserção também foi realizada seguindo o exemplo disponível nas transparências da disciplina.

Quicksort inserção 5% e 10%: Idênticos Quicksort inserção 1%, mudando apenas a porcentagem que é analisada para saber se o método de inserção deve ser chamado ou não.

Quicksort não recursivo: Esta variação utiliza uma pilha (classe *Stack*) de índices para simular as chamadas recursivas. As implementações tanto do Quicksort como da pilha foram realizadas conforme presente nas transparências da disciplina.

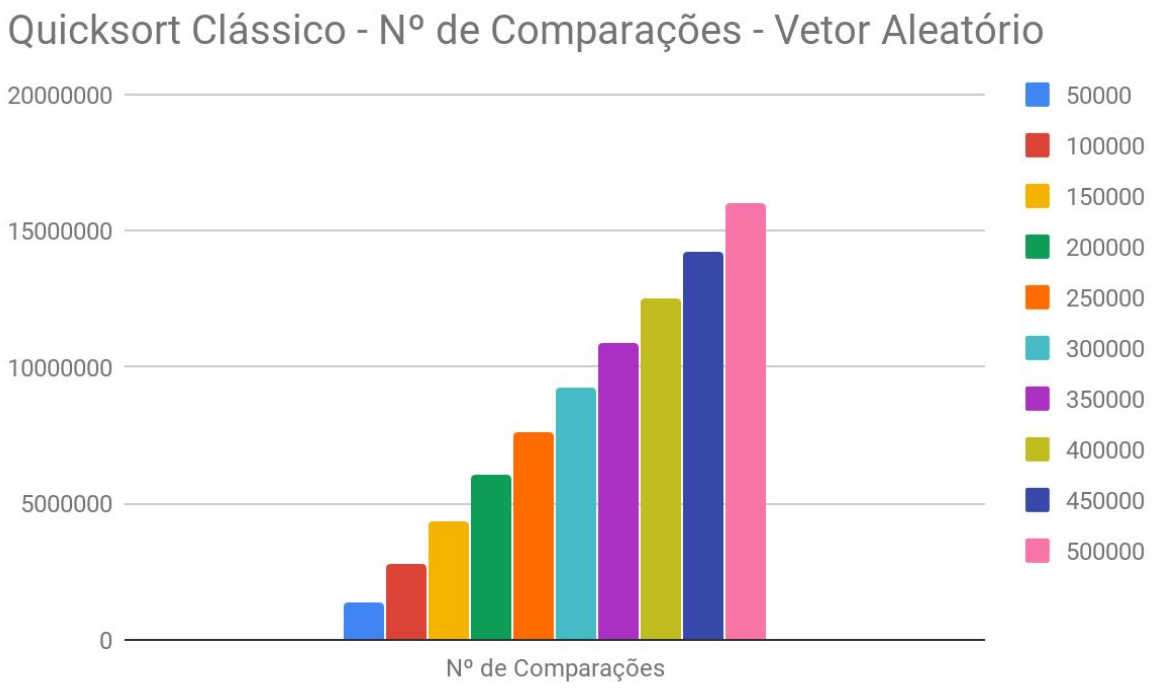
3 INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Compile com make e rode com ./tp2 <variacao> <tipo> <tamanho> [-p]. O makefile utilizado para compilação foi o disponibilizado pelos monitores no Moodle.

4 ANÁLISE EXPERIMENTAL

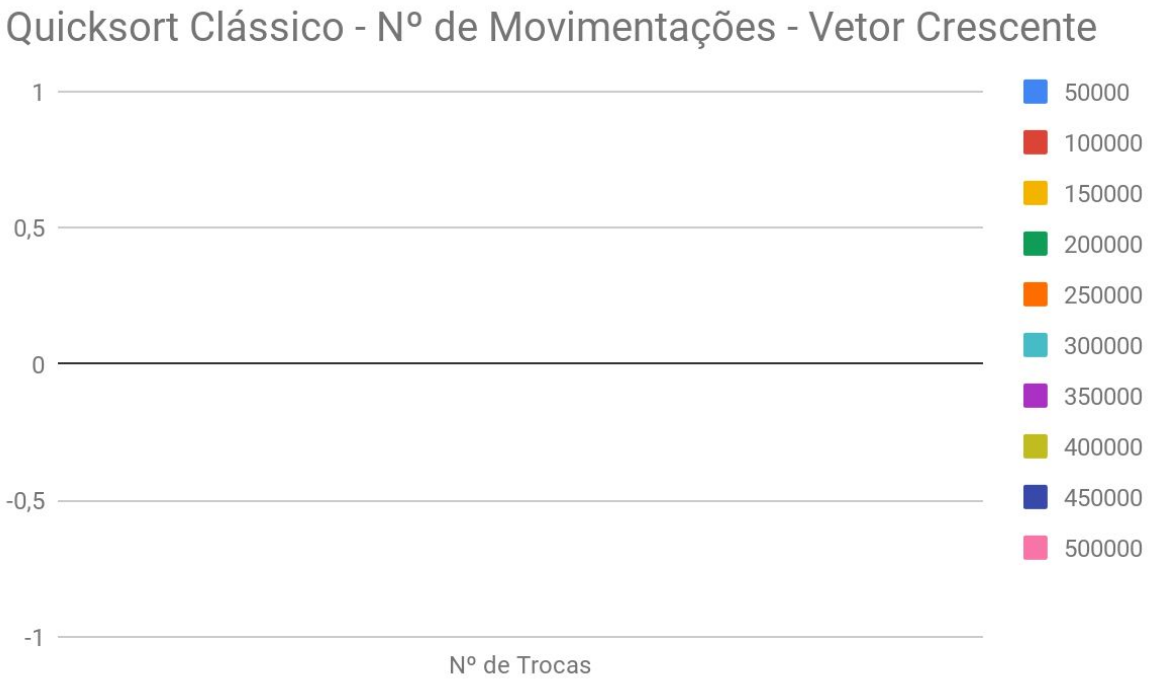
Quicksort clássico

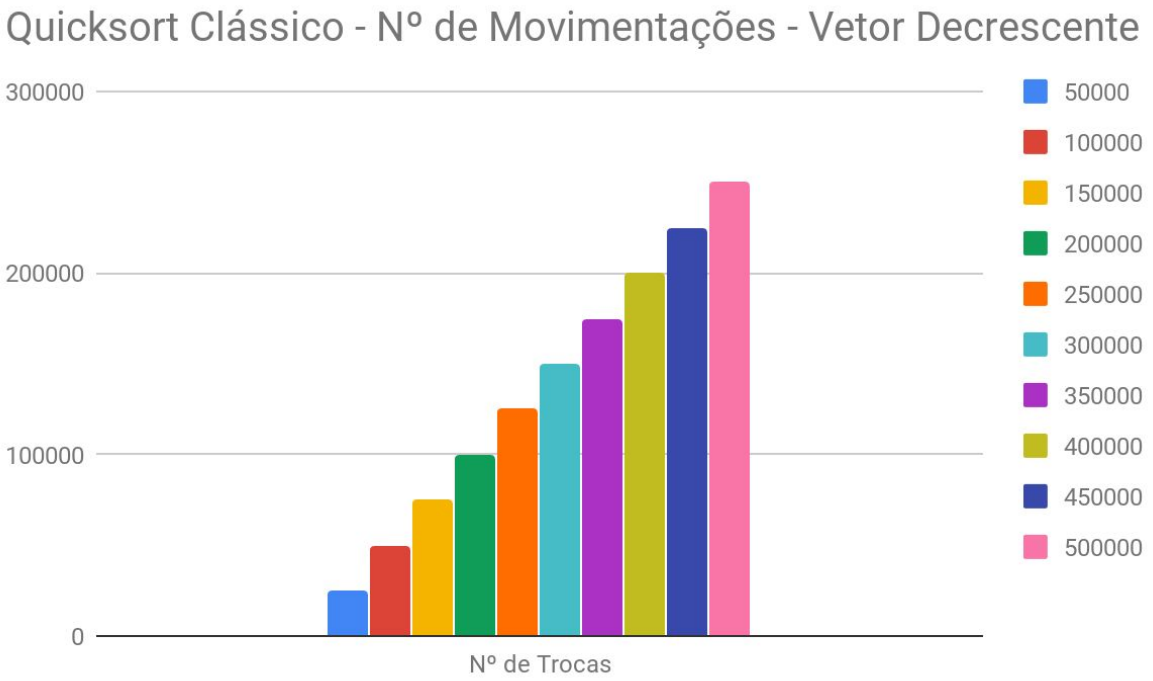
Comparações: Para o Quicksort clássico, o número de comparações se mostrou crescente para todos os tipos de vetores, isto é, o número de comparações é proporcional ao tamanho do vetor. Esse resultado já era esperado, uma vez que um vetor maior requer mais comparações para ser ordenado. Notou-se que o número de comparações segue a seguinte ordem de grandeza: vetor crescente < vetor decrescente < vetor aleatório, embora a diferença entre os vetores crescentes e decrescentes seja pequena. Abaixo segue um gráfico do número de comparações para o Quicksort clássico e vetor aleatório.



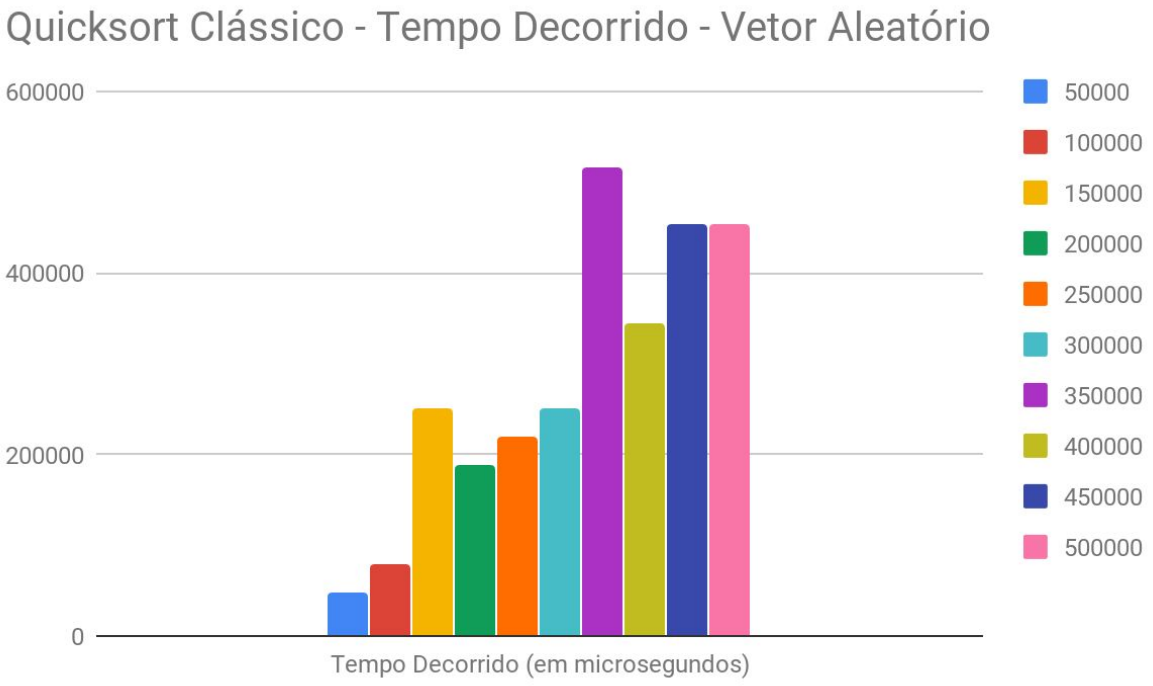
Movimentações: O número de movimentações se mostrou crescente para os vetores aleatórios, variando entre 3,7 a 4,5 vezes o tamanho do vetor (ex: para um vetor de tamanho 50000 foram feitas 3,7*50000 movimentações e para um vetor de tamanho 500000 foram feitas 500000*4,5 movimentações). Tal como esperado para os vetores crescentes o número de movimentações foi zero, uma vez que em vetores ordenados o quicksort não realiza movimentações. Já para os vetores decrescentes

o número de movimentações foi exatamente n/2, resultado também esperado dado que o Quicksort funciona comparando e trocando pares e como o pivô é o elemento central, metade dos elementos são maiores que o pivô e metade são menores. Abaixo estão os gráficos do número de movimentações para vetores crescentes e decrescentes, respectivamente.





Tempo: Para todos os casos o tempo aumenta quando aumenta-se o tamanho do vetor, o que era esperado. Os gráficos mostram tempos de certa forma oscilante, o que pode ser explicado pelo fato de algumas execuções serem “melhores” que outras, mas mesmo assim a diferença entre os tempos é muito pequena (milisegundos). Dentre os tipos de vetores, não é possível notar muita diferença no tempo dentro do Quicksort clássico. Abaixo segue um gráfico que apresenta os dados do tempo para o vetor aleatório.



Conclusão: O algoritmo do Quicksort clássico se mostrou bastante eficiente para ordenar vetores de todos os tipos e tamanhos.

Quicksort mediana de três

Comparações: O Quicksort mediana três apresentou comportamento bem semelhante ao clássico quanto às comparações. Se comparado ao clássico, apresentou uma pequena melhora somente ao ordenar um vetor aleatório. Isso já era esperado, uma vez que ao realizar a escolha do pivô através da mediana de três elimina-se a possibilidade se escolher o menor ou maior elemento, o que pode acontecer no clássico. Não é interessante exibir algum gráfico aqui porque o comportamento é quase idêntico ao clássico no aspecto comparações.

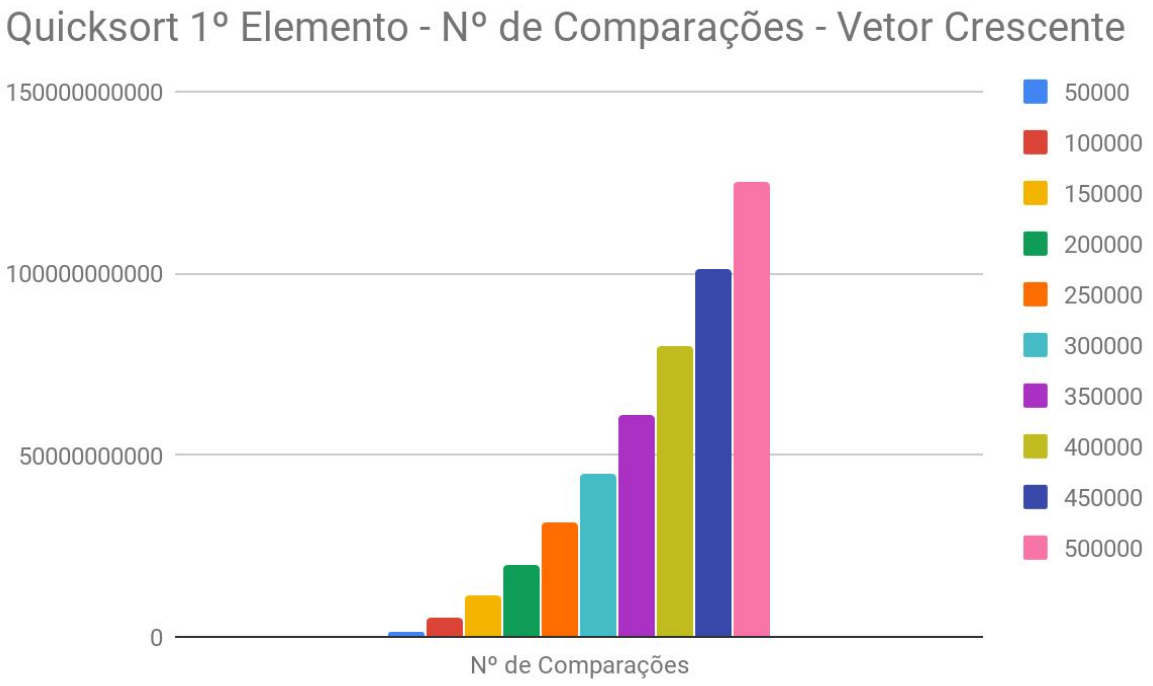
Movimentações: As movimentações seguem o mesmo padrão das comparações, semelhante ao Quicksort clássico mas com uma leve melhora. O número de movimentações num vetor crescente novamente foi zero.

Tempo: Os tempos também foram muito parecidos com o Quicksort clássico.

Conclusão: Bastante eficiente e ligeiramente melhor que o clássico. Sem dúvidas um dos mais confiáveis por evitar o pior caso.

Quicksort primeiro elemento

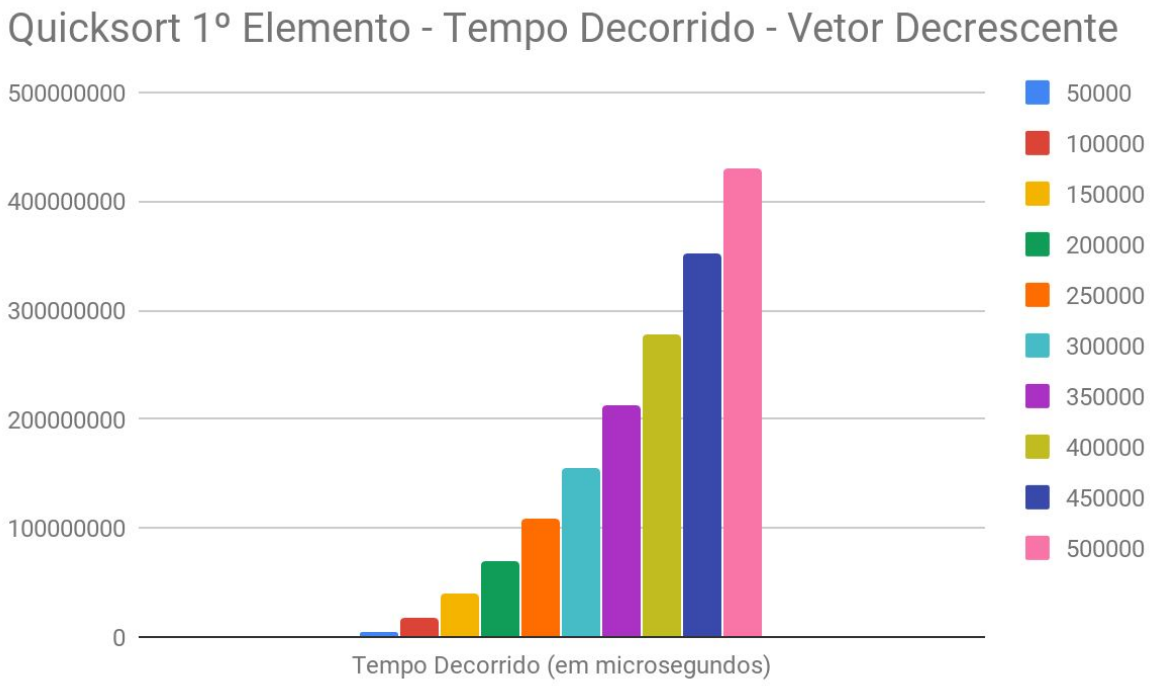
Comparações: Leve piora em relação ao Quicksort clássico e mediana de três para vetores aleatórios. Para vetores crescentes e decrescentes observou-se um número extremamente elevado de comparações, o que era esperado por se tratar dos piores casos. O número de comparações para vetores crescentes e decrescentes foi de aproximadamente $n^2 / 2$, o que nos leva a uma complexidade $O(n^2)$ e é justamente essa a complexidade do Quicksort no pior caso. Abaixo está o gráfico do Quicksort primeiro elemento para comparações e vetor crescente.



Movimentações: Número de movimentações bastante parecido com o Quicksort clássico e mediana de três para vetores aleatórios. Zero movimentações em vetores crescentes. n / 2 movimentações em vetores decrescentes.

Tempo: Quase idêntico ao clássico e mediana de três para vetores aleatórios, apresentando uma leve piora. Tempo extremamente elevado para vetores

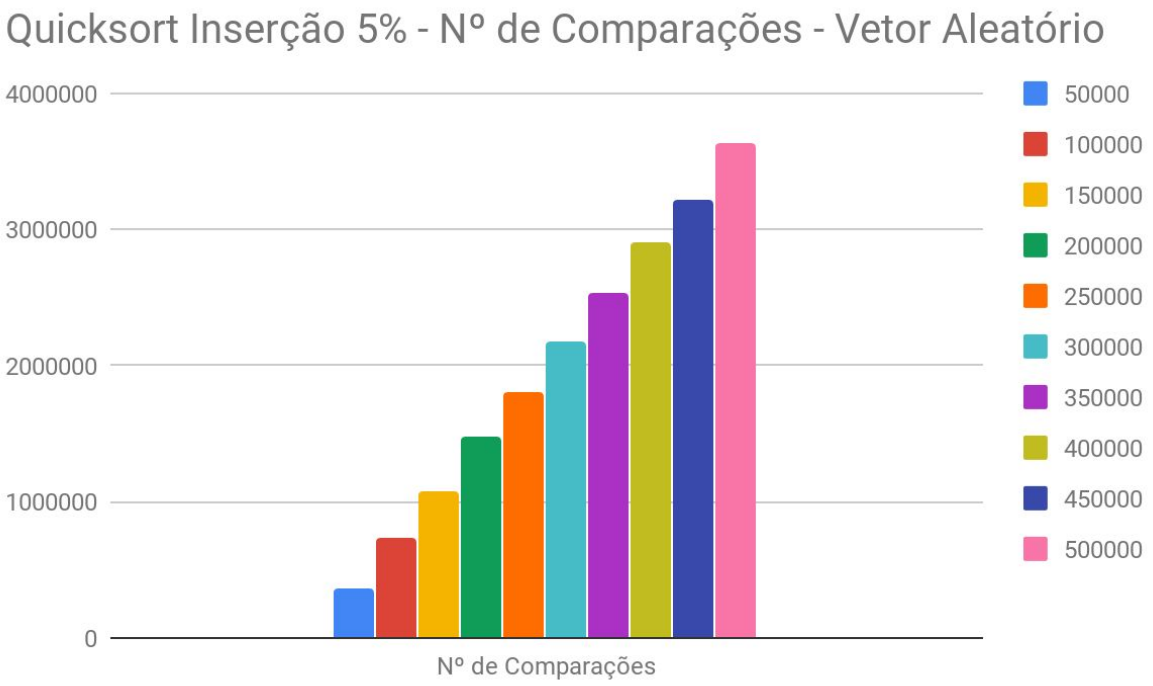
crescentes e decrescentes. O tempo chegou a 7 minutos para 500000 elementos decrescentes. Abaixo está um gráfico do tempo em microssegundos e vetor decrescente.

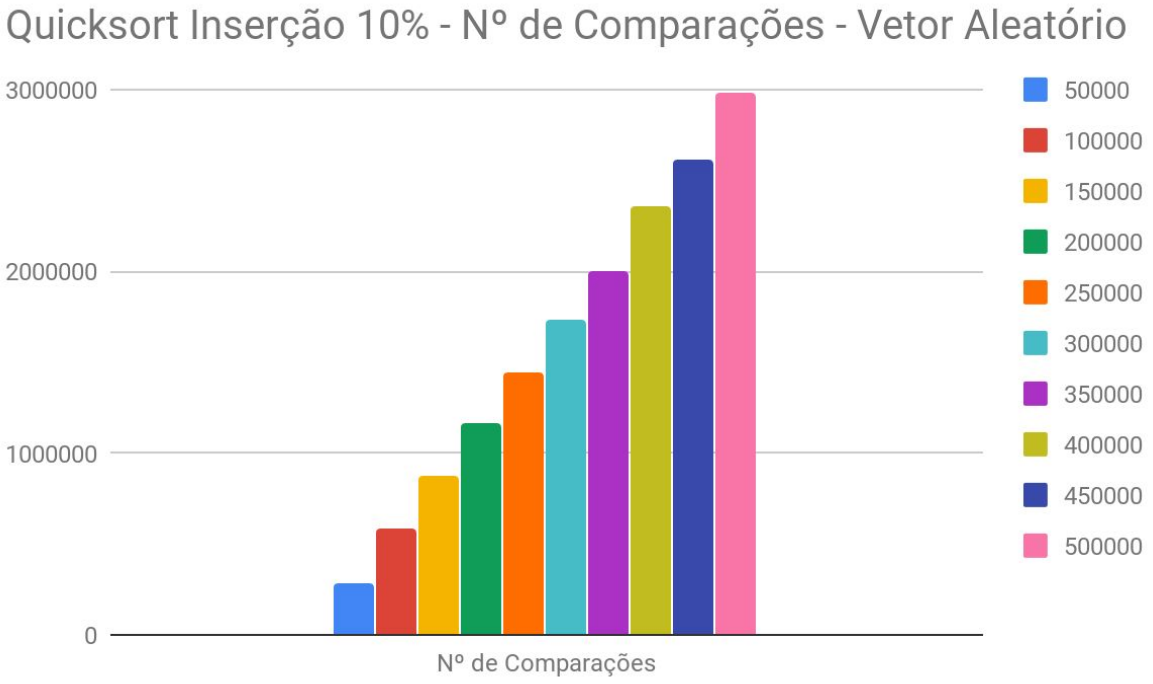


Conclusão: Algoritmo extremamente ruim caso o vetor já esteja ordenado (crescente ou decrescente).

Quicksort inserção 1%, 5% e 10%

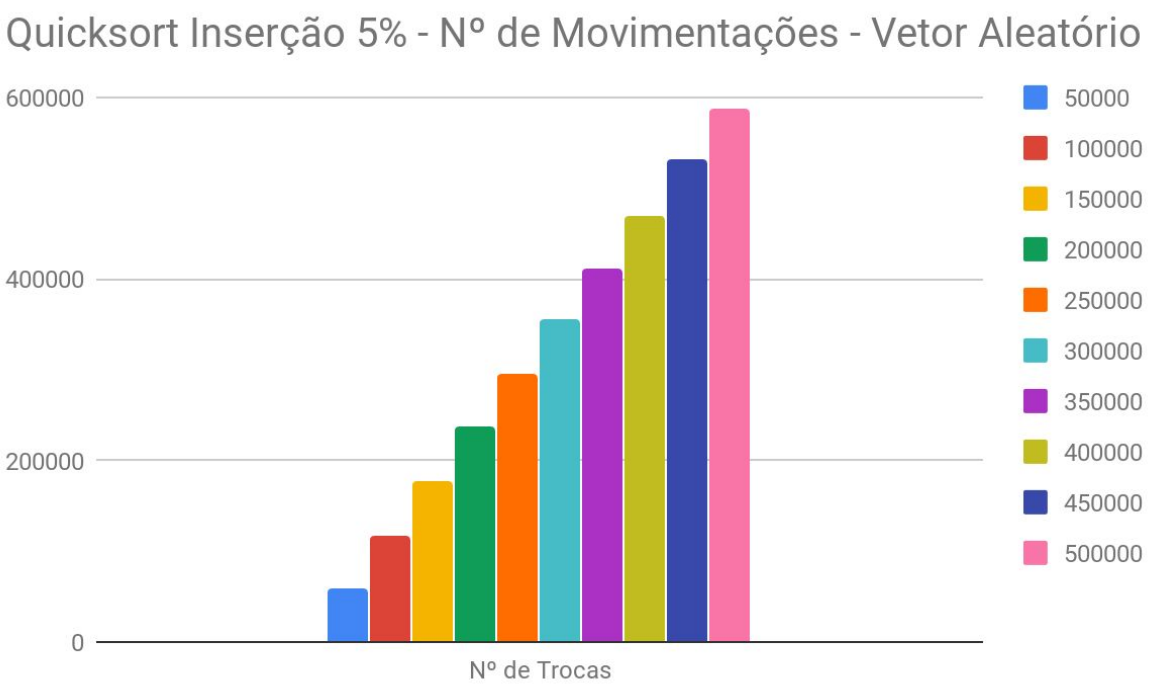
Comparações: Apresenta uma melhora significativa em relação aos algoritmos já apresentados em todos os tipos de vetores. Em média, o número de comparações foi de duas à três vezes menor que os demais. Observou-se que ao aumentar a porcentagem a partir da qual o Inserção seria aplicado ocorreu uma diminuição no número de comparações. O resultado descrito era de certa forma esperado, uma vez que o Inserção trabalha muito melhor com pequenos vetores que o Quicksort. Abaixo estão gráficos do número de comparações para vetores aleatórios para os níveis 5% e 10%, respectivamente. Como dito, pode ser observado uma queda no número de comparações ao variar de 5% para 10%.

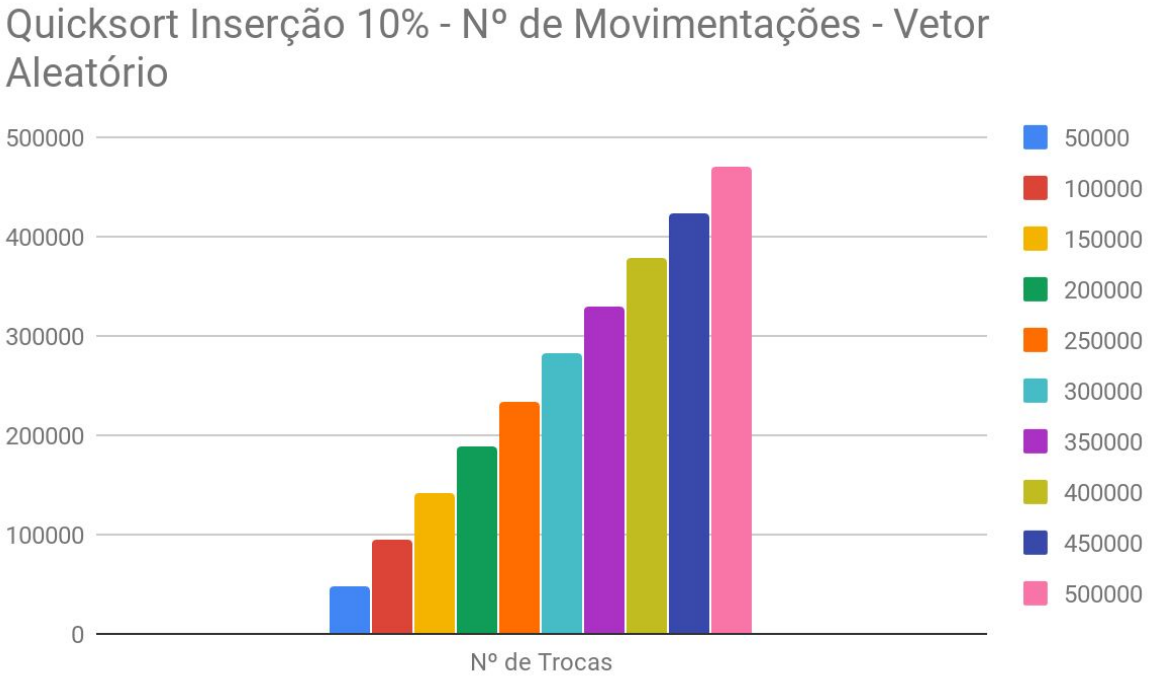




Movimentações: As movimentações em vetores crescentes e decrescentes foram iguais às demais implementações (zero e n/2). Já para os vetores aleatórios observou-se uma queda brusca no número de movimentações. Abaixo estão os

dados do número de movimentações para vetores aleatórios no algoritmo Quicksort inserção 5% e 10%, respectivamente.





Tempo: Os tempos de execução apresentaram uma leve melhora em relação aos algoritmos já apresentados. Ao aumentar a porcentagem diminui-se levemente o tempo de execução.

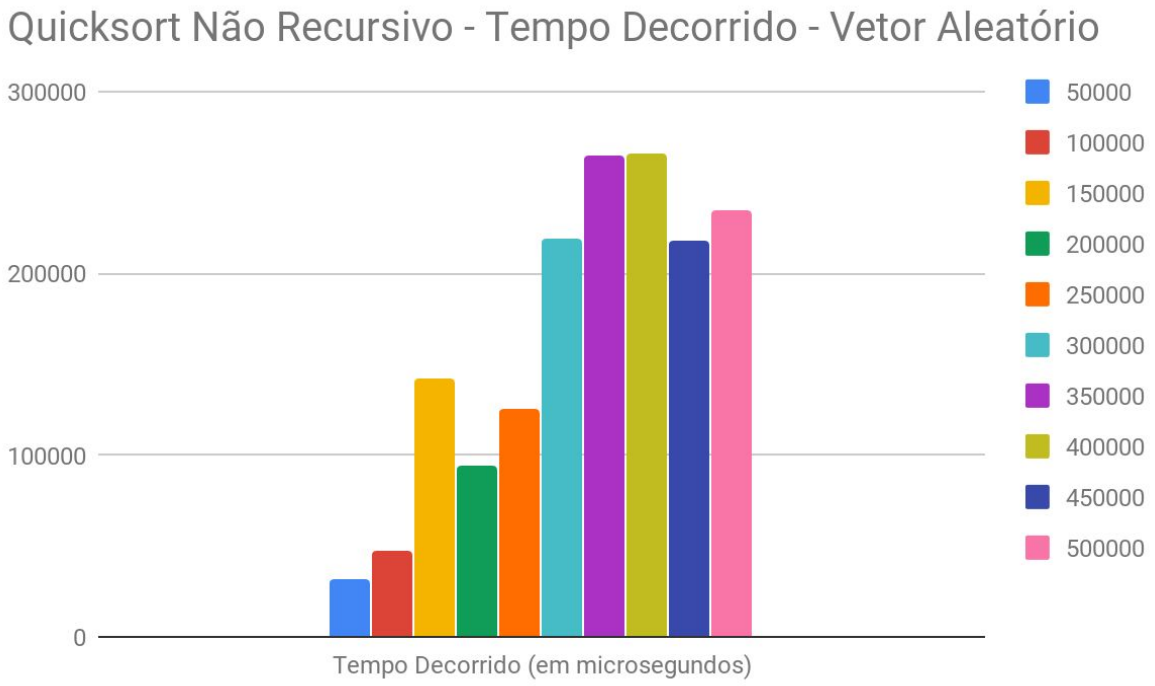
Conclusão: Algoritmos bastante usuais por melhorem significativamente o Quicksort mediana de três, mantendo seus aspectos positivos.

Quicksort não recursivo

Comparações: Para comparações o Quicksort não recursivo se mostrou muito semelhante ao Quicksort clássico, portanto não há muito o que analisar.

Movimentações: Quase idêntico ao Quicksort clássico.

Tempo: Em média, apresentou os números mais baixos se comparado aos demais. Abaixo está o tempo de execução para vetores aleatórios.



Conclusão: Apresenta-se como boa alternativa ao Quicksort clássico, por diminuir o tamanho da pilha de execução e consequentemente o tempo de execução.

4 CONCLUSÃO

O Quicksort de fato se mostrou um algoritmo bastante eficiente para ordenação de vetores de números inteiros. Algumas das variações testadas se mostraram como boas melhorias ao algoritmo clássico, tais como a escolha do pivô como mediana de três para evitar o pior caso e o uso de um outro algoritmo para ordenar as partições menores (Insertion sort). O Quicksort primeiro não se mostrou interessante, por permitir que o pior dos casos aconteça. Já o Quicksort não recursivo se apresentou como uma boa alternativa ao algoritmo clássico, eliminando a necessidade de chamadas recursivas. Entretanto, o não recursivo requer a implementação de uma estrutura de dados do tipo pilha adicional e possui uma implementação complicada e por isso deve ser ponderado se vale a pena ou não utilizá-lo.

Por fim, conclui-se que o trabalho contribuiu bastante para o aprendizado, permitindo exercitar o conteúdo visto em sala de aula.

BIBLIOGRAFIA

[1] StackOverflow. Disponível em: <https://stackoverflow.com/>. Acesso em: 13 jun. 2019.

[2] C++ Language. Disponível em: <http://www.cplusplus.com/doc/tutorial/>. Acesso em: 13 jun. 2019.

[3] Geeks for Geeks. Disponível em: <https://www.geeksforgeeks.org>. 13 jun. 2019.

[4] **Transparências da disciplina "Estruturas de Dados"**.UFMG Virtual. Disponível em: <virtual.ufmg.br>. Acesso em: 13 jun. 2019.

[5] Quicksort. Departamento de Ciência da Computação da UFMG. Disponível em: <https://www.dcc.fc.up.pt/~pbv/aulas/progimp/teoricas/teorica18.html>. Acesso em: 13 jun. 2019.