# 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 microssegundos 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 abordar o problema 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, 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 dele, 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 direita 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.

Temos 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 decisão 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 caso 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 o vetor recebido.
- *iterative_sort(long long, int, int)* – Método que define a implementação do caso onde o *Quicksort* será iterativo ou 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 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 e retornar 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 ele 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 que definem como o pivô será escolhido dado um vetor

- 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 corretamente uma instância da classe *SortRequest* e chamar o método de controle *run()*.

## 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 tela

## 4. Análise experimental

Todos os testes executados foram feitos em um computador com processador i5-4200U Quad-Core com 1.60GHz por núcleo.

## 5. References

Bibliographic references must be unambiguous and uniform. We recommend giving the author names references in brackets, e.g. [Knuth 1984], [Boulic and Renault 1991]; or dates in parentheses, e.g. Knuth (1984), Smith and Jones (1999).

The references must be listed using 12 point font size, with 6 points of space before each reference. The first line of each reference should not be indented, while the subsequent should be indented by 0.5cm.

## References


Boulic, R. and Renault, O. (1991) "3D Hierarchies for Animation", In: New Trends in Animation and Visualization, Edited by Nadia Magnenat-Thalmann and Daniel Thalmann, John Wiley & Sons ltd., England.

Dyer, S., Martin, J. and Zulauf, J. (1995) "Motion Capture White Paper", http://reality.sgi.com/employees/jam_sb/mocap/MoCapWP_v2.0.html, December.

Holton, M. and Alexander, S. (1995) "Soft Cellular Modeling: A Technique for the Simulation of Non-rigid Materials", Computer Graphics: Developments in Virtual Environments, R. A. Earnshaw and J. A. Vince, England, Academic Press Ltd., p. 449-460.

Knuth, D. E. (1984), The TeXbook, Addison Wesley, 15th edition.

Smith, A. and Jones, B. (1999). On the complexity of computing. In *Advances in Computer Science*, pages 555–566. Publishing Press.