**UNIBH UNIDADE CRISTIANO MACHADO**

JOSÉ CLÁUDIO DOS SANTOS JÚNIOR

MATEUS COSTA AVELAR

WESLEI DA SILVA

**ALGORITMOS DE ORDENAÇÃO**

Belo Horizonte

Abr/2020

Sumário

[**1-Introdução** 3](#_Toc38380086)

[**2-Ambiente experimental.** 3](#_Toc38380087)

[**3-Como foi feito:** 3](#_Toc38380088)

[**4-Resultados.** 4](#_Toc38380089)

[**4.1.Insertion sort** 5](#_Toc38380090)

[**4.2.Merge Sort** 5](#_Toc38380091)

[**4.3.Quick sort** 6](#_Toc38380092)

[**4.4.Bubble sort** 6](#_Toc38380093)

[**4.5.Selection sort** 7](#_Toc38380094)

[**6.Conclusão** 8](#_Toc38380095)

[**7. Apêndice** 9](#_Toc38380096)

# **1-Introdução**

O Quão importante é manter a organização dos seus dados ? Organizando seus dados você possui controle maior deles possibilitando tratativas de problemas de forma mais rápida entregando soluções para sua empresa mais robustas e eficaz. Nem sempre manter seus dados organizados seja por uma ordem crescente, decrescente ordenada que seja vai ser fácil dependendo do volume de dados que você possui, pelo menos é isso que muitos pensam quando os dados passam as casas dos milhares. Realizamos testes de ordenação de dados utilizando vários métodos já introduzidos em aula tais como Insertion Sort, Merge Sort, Quick Sort, Bubble Sort, Selection Sort onde utilizamos vetores de 5 até 100 mil posições e realizamos o cálculo de quanto tempo levou para que seja realizada a ordenação dos vetores buscando identificar o pior, médio e melhor caso para determinada quantidade de vetores. Com estes dados é possível realizar a melhor escolha possuindo uma máquina de porte mediano para ordenação de seus dados de acordo com demanda ou necessidade sua ou de seu cliente.

# **2-Ambiente experimental.**

Os dados foram obtidos usando a mesma máquina, e para maior confiabilidade todos os serviços não essenciais foram desativados para a realização dos testes, características da máquina:

Processador: Intel(R) Core(™) i7-3770 CPU @ 3.40GHz.

Memória instalada (RAM): 8,00GB.

Tipo do sistema: Sistema Operacional de 64 bits, processador com base em x64.

Ambiente: Windows 10 Pro.

GPU: GTX 1050Ti 4GB.

# **3-Como foi feito:**

Os experimentos foram realizados utilizando programação em C++ onde foi implementado vários métodos de ordenação sendo eles, bubble-sort, insertion-sort, merge-sort, quick-sort e selection-sort.

O preenchimento dos vetores foi realizado pelo próprio código que o preenche automaticamente as posições no vetor através de for e variáveis de controle com atribuição de valor toda vez que é "rodado" já preenchendo um valor.

Função para criar arrays preenchendo eles aleatórios e para salvar eles a fim de trabalhar esses dados posteriormente. Foi utilizado contadores para medir o tempo somente de ordenação, o tempo de preenchimento dos vetores, não foi contabilizado o tempo de preenchimento dos vetores.

Cada um dos métodos utilizados foi realizada as alterações dos vetores de forma diferente,

o **bubble sort** trocando os números de lugar se necessário após validar se o número da frente é menor que o anterior.

O **Merge Sort**, divide ao meio os dados do vetor após preenchidos pela função no inicio do codigo e vai ordenando ele em paralelo as divisões menores e juntando as mesmas novamente.

**Quick Sort**, a ordenação é realizada a partir do momento que se decide um pivor onde vão separar os elementos a esquerda e a direita do pivor e utilizar duas variáveis “i” e “j” onde avaliamos no código os elementos que são maiores e menores que o pivor, após feito isso jogamos o lado maior para o lado direito e o menor para a esquerda a recursividade continua enquanto o esquerdo for menor que o direito e trabalhamos esses dados de forma recursiva afim de realizar a ordenação.

**Selection Sort** Procura o menor elemento e joga ele para um dos lados depois valida os outros procurando o segundo menor elemento e também o joga para o lado até organizar todo o vetor, validando assim as posições do vetor e ordenando cada valor do menor para o maior ocupando as posições do arraw.  é utilizado dois for com contadores realizando os procedimentos citados anteriormente.

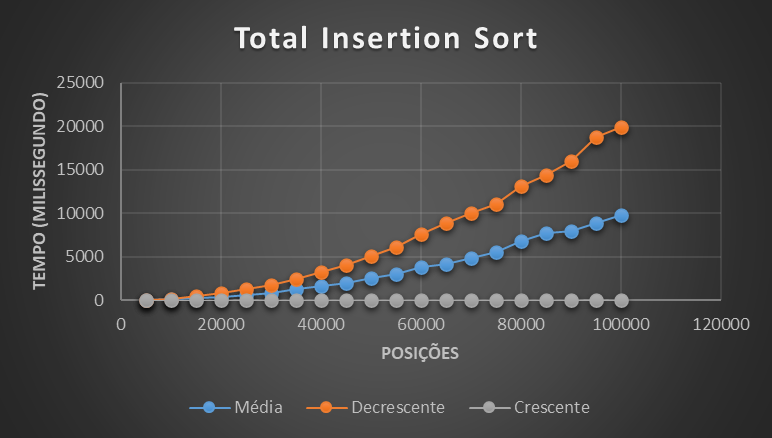
**Insertion Sort** utilizado dois vetores onde o primeiro contém os dados e no segundo vetor ja são adicionados os dados de forma organizada já entrando cada um deles em sua ordem correta nesse novo vetor. Começa na posição 1 do vetor e vai até a última posição, recebe a posição que está passando no for, enquanto o valor que está passando em vtemp for menor que a posição vtemp -1 realiza a troca, quando vtemp chegar na posição 0 o while para.

# **4-Resultados.**

Devido ao ambiente de obtenção dos dados não ser ideal é esperado de se encontrar distorções nos resultados quando em comparação com a curva esperada teoricamente.

Tais distorções podem ser geradas por temperatura, serviços rodando em segundo plano, entre outros possíveis fatores.

## **4.1.Insertion sort**

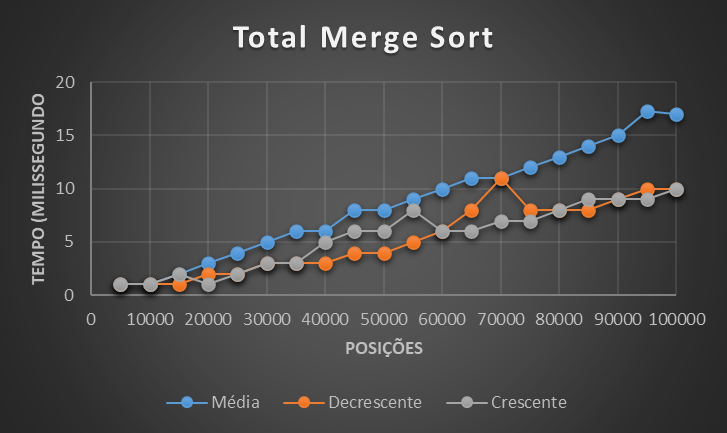
****Figura

*Resultado esperado: melhor caso: O(n); pior caso: O(n²); caso médio: O(n²).*

Crescente - O algoritmo crescente roda em uma velocidade muito alta, quase não gasta tempo para ser executado, na unidade utilizada para comparação de tempo o valor é imperceptível devido à grande velocidade de execução.

Decrescente e Media – à medida que as posições aumentam o tempo cresce juntamente a eles, porém a decrescente aumenta o tempo de execução com uma velocidade duas vezes maior que a média.

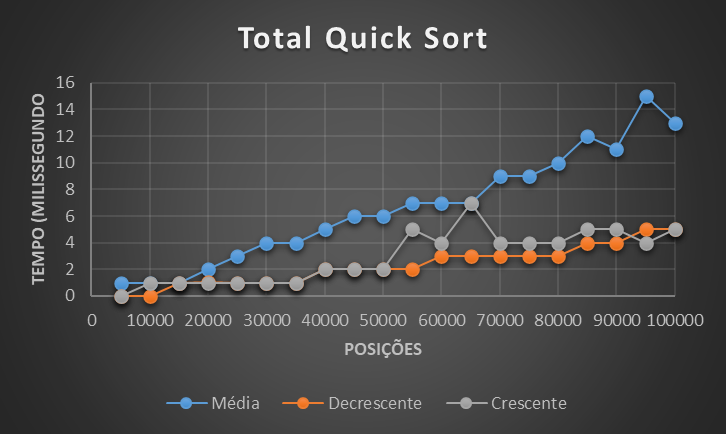
## **4.2.Merge Sort**

Figura

*Resultado esperado: melhor caso: O n log(n); pior caso: O n log(n); caso médio: O n log(n).*

Crescente, decrescente e media: crescem de forma similar, à medida que aumentam as posições aumenta o tempo. Porém de acordo com o número elevado de posições depois da posição de 75.000 mil a média passa a gastar quase o dobro de tempo das demais.

## **4.3.Quick sort**

****Figura

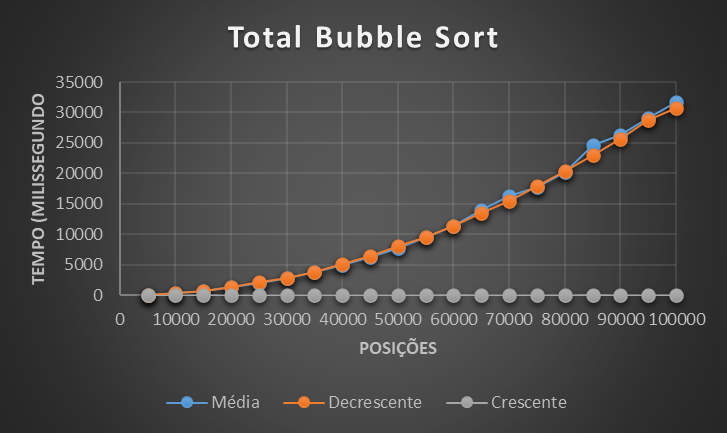
*Resultado esperado: melhor caso: O n log(n); pior caso: O(n²); caso médio: O n log(n).*

Crescente, decrescente e media, elas crescem gradativamente de acordo com a inserção de posições.

A média continua crescendo de forma gradativa, porém gasta quase três vezes mais tempo que as outras.

Crescente e decrescente: entre os pontos de 15.000 mil e 50.000 mil posições gastam o tempo praticamente igual, apresentando variação mínima. Após a posição de 50.000 mil a crescente passa a gastar mais tempo.

## **4.4.Bubble sort**

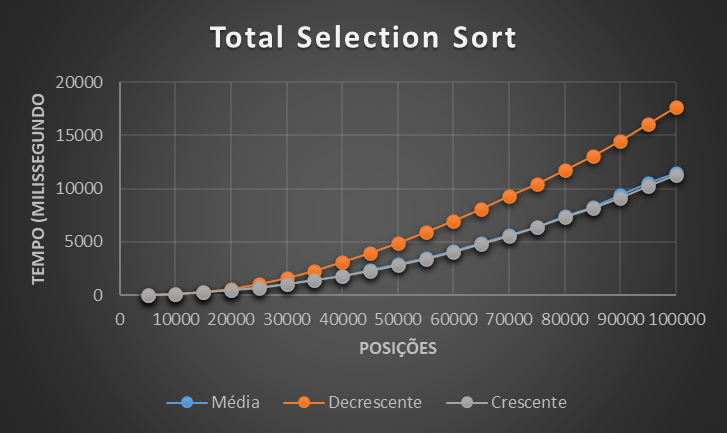
****Figura

*Resultado esperado: melhor caso: O(n); pior caso: O(n²); caso médio: O(n²)*

A Crescente é executada em alta velocidade utilizando um tempo mínimo de execução, praticamente sem variações.

Media e decrescente: gastam quase o mesmo tempo, crescem de forma igual. De acordo com a quantidade de posições inseridas o tempo vai aumentando.

## **4.5.Selection sort**

Figura

*Resultado esperado: melhor caso: O(n²); pior caso: O(n²); caso médio: O(n²)*

Crescente, decrescente e media aumentam o tempo de execução de acordo com a quantidade de posições inseridas. A crescente e media gastam quase o mesmo tempo de execução.

A decrescente inicia igual a demais mas passa a gastar mais tempo após a posição de 30.000 mil, mas ainda crescendo de forma exponencial.

# **6.Conclusão**

Analisando de uma forma geral os algoritmos Merge Sort e Quick Sort utilizam menor tempo de execução (decrescente e media ) entre os demais. Comparando o Merge e quick sort, o Quick Sort ainda é melhor para execução (decrescente e media ) com menor tempo entre os dois.

Por outro lado, o bubble Sort e insertion sort na forma crescente executam o algoritmo de forma tão rápida que não é possível analisar o tempo gasto. Mas a média e decrescente gastam mais tempo que os demais algoritmos para execução.

Relacionando o selection ao demais ele é o que tem menos a oferecer por não ter um melhor caso, todos os casos são executados de forma parecida. Mesmo que ele gaste menos tempo para executar MÉDIO e DECRESCENTE Comparado ao insertion e bubble sort.

Entre todos os algoritmos proposto concluímos que, o Quick Sort é que mostrou melhor desempenho entre eles, mesmo que sua execução na forma crescente gaste mais tempo do que o Bubble Sort e Insertion Sort. Mas ainda apresenta um tempo baixo de 5 Milissegundo na sua posição de maior inserção.

# **7. Apêndice**

Os algoritmos e os executáveis utilizados nesse trabalho podem ser encontrados no repositório do GITHUB no endereço: (https://github.com/Jjoclas/Trabalho-AED-Sort ).