# Métodos de Ordenação

**Augusto Daniel Rodrigues[1], Natan Ogliari[2]**

[1]Graduando em engenharia eletrônica, IFSC – Câmpus Florianópolis

[2] Graduando em engenharia eletrônica, IFSC – Câmpus Florianópolis

{augusto.dr@aluno.ifsc.edu.br , natan.o@aluno.ifsc.edu.br}

***Resumo.*** *Os métodos de ordenação são algoritmos utilizados para organizar um conjunto de dados em um padrão específico. Neste trabalho, o padrão utilizado foi uma sequência crescente de números inteiros. Foram calculadas as médias e desvios padrão para vinte iterações dos seguintes tipos de ordenação: BubbleSort, QuickSort, Inserção Direta, Inserção Binária, Seleção Direta, HeapSort e MergeSort. Para cada iteração de cada método, preenche-se um vetor de um milhão de elementos com números aleatórios e executa-se a ordenação . Este trabalho foi testado em um computador com as seguintes configurações: (HP) AMDx64 Phenom™ II X4 B97 @3227Mhz, memória física de 3.595MB. Ao final serão apresentados os resultados de cada algoritmo, permitindo a comparação entre os métodos desta implementação e, levando em conta as configurações do computador usado, a comparação entre diferentes implementações.*

## 1. BubbleSort

O método de ordenação *BubbleSort* funciona fazendo troca entre elementos consecutivos do vetor, após comparação do maior conteúdo entre eles. Neste caso, como deseja-se ordenar de forma crescente, após a comparação, o maior valor é movido para a posição de maior índice, e o menor valor, para a posição de menor índice, obviamente.

O algoritmo implementado para executar a ordenação *BubbleSort* consiste em dois laços de iteração *for*, sendo que no laço *for* interno, há uma estrutura *if* para executar a comparação entre os elementos consecutivos em análise. Se o conteúdo da posição atual for maior que o conteúdo da próxima posição, a troca é efetuada. O laço externo garante que todos os elementos do vetor sejam selecionados para serem submetidos ao processo de comparação e troca adjacente. O laço interno executa as comparações e trocas necessárias do elemento escolhido com seus elementos consecutivos.

O grau de complexidade no caso médio e no pior caso é de $O(n^2)$. Nesta estrutura, foi obtida uma média igual a 1302,559 segundos, com um desvio padrão associado de 2,0267 segundos.

## 2. QuickSort

O método de ordenação Quicksort utiliza a estratégia de divisão e conquista. Um elemento do vetor é escolhido como pivô e é removido do vetor. São criadas duas sub-sequências do vetor, sendo que a primeira sub-sequência vai do início do vetor até a posição anterior do pivô e a segunda sub-sequência vai da posição após o pivô até o fim do vetor. É feita então uma ordenação, que consiste em colocar todos os elementos menores que o conteúdo do pivô na primeira subsequência, e todos os elementos maiores que o conteúdo do pivô na segunda subsequência. Com esta divisão feita, o mesmo método é aplicado de forma recursiva nas duas partes criadas, resultando em quatro novas subsequências, que serão também particionadas e ordenadas, e assim sucessivamente. Quando as subsequências atingirem o menor tamanho possível e o pivô coincidir com o início da subsequência, a ordenação está completa.

Para escolher o pivô de forma a aprimorar a execução do algoritmo, é feita uma comparação entre os conteúdos do início, do centro e do final do vetor original e de cada subsequência, e o valor médio entre estes três é colocado no centro do vetor ou subsequência, e essa posição é selecionada como pivô.

O grau de complexidade no pior caso é de ($O(n^2)$) e no melhor caso ($O(n*\log n)$). Para que seja respeitado o algebrismo citado faz-se necessário que a mediana de três seja linear ($O(n)$). Nesta estrutura, foi obtida uma média de 0,115 segundos  e desvio padrão 0,0006 segundos.

## 3. Inserção Direta

No algoritmo de ordenação por inserção direta ou linear, são usados dois laços de iteração *for*. O laço interno faz uma varredura regressiva, da posição atual até o início do vetor e, a cada decremento na posição, compara o conteúdo da posição atual com o conteúdo da posição anterior. Se o valor da posição atual for menor e não se chegou ao início do vetor, é feita a troca entre esses dois elementos. Isso cria um sub-vetor ordenado dentro do vetor desordenado que está sendo varrido. O laço *for* externo redefine a posição inicial da varredura do laço interno, ou seja, redefine o tamanho do sub-vetor ordenado, iniciando no índice 1 do vetor (segunda posição) e indo até a última posição, e assim sucessivamente. Quando redefinido para a última posição do vetor original, o sub-vetor ordenado corresponde ao próprio vetor ordenado.

Na possibilidade de visualizar a ordenação, seriam vistos, ocorrendo do início para o fim do vetor (da esquerda para direita), os menores elementos do

vetor sendo “empurrados” para as primeiras posições, até que o vetor fique completamente ordenado.

O grau de complexidade no caso médio e no pior caso é de $O(n^2)$ e, no melhor caso, de O(n). Nesta estrutura, foi obtida uma média igual a 1083,35 segundos, com um desvio padrão associado de 2,237 segundos.

## 4. Inserção Binária

O algoritmo de ordenação por inserção binária cria um sub-vetor ordenado em forma crescente no início do vetor de números aleatórios, que vai crescendo, até atingir o tamanho do próprio vetor, quando todos seus elementos estão ordenados. O algoritmo é constituído de 3 laços de iteração, sendo um *while* e um *for* aninhados em outro *for* externo.

Basicamente, o laço *for* externo é responsável pela varredura linear progressiva dos elementos do vetor. No início de cada iteração deste laço, copia-se o conteúdo da posição atual para uma variável auxiliar temporária, define-se a esquerda como zero e a direita como a posição atual. Então o laço *while* usa a variável auxiliar temporária para buscar a posição correta de inserção dentro do sub-vetor, porém a busca é binária, ou seja, a variável auxiliar é comparada com o elemento central do sub-vetor ordenado. Se o auxiliar é menor, a posição correta está na primeira metade do sub-vetor, senão está na segunda metade. Assim, compara-se a variável com a elemento central da metade que foi identificada como contendo a posição correta. Esse procedimento é repetido pelo laço *while* até ser encontrada a posição correta. Quando a posição correta é encontrada, o laço *for* após o laço *while* faz as trocas necessárias para a inserção do elemento em análise na posição correta encontrada com a busca binária.

O grau de complexidade no caso médio e no pior caso é de $O(n^2)$ e, no melhor caso, de O( n*log(n)). Nesta estrutura, foi obtido uma média igual a 1346,035 segundos, com um desvio padrão associado de 0,920 segundos.

## 5. Seleção Direta

No método de ordenação por seleção direta, a cada varredura completa do vetor, o maior valor encontrado por comparação sucessiva é colocado na posição final e o apontador de fim de vetor é decrementado, para que na próxima varredura, que encontrará o segundo maior elemento, este seja colocado na penúltima posição do vetor, e assim sucessivamente.

A implementação deste método é feita com duas estruturas *for*, sendo que a estrutura externa redefine a posição final de varredura e zera a variável que

guarda o índice do último maior elemento lido, e a estrutura interna faz a varredura e comparação dos elementos do vetor, salvando o maior lido na posição final.

O grau de complexidade deste método é de $O(n^2)$. Nesta estrutura, foi obtida uma média igual á 799,151 segundos, com um desvio padrão associado de 1,2712 segundos.

## 6. Seleção HeapSort

No método de ordenação *heapsort*, é criada uma estrutura de dados que contém o tamanho do vetor a ser ordenado, um ponteiro para o vetor a ser ordenado e o comprimento atual do vetor, que inicia igual ao tamanho do vetor.

Em seguida, o vetor é ordenado no formato de uma árvore binária. O elemento anterior à metade do vetor é selecionado como folha-pai (ou vértice-pai), e é usada a relação de 2*i + 1 para localizar a folha-filha esquerda e a relação 2*i +2 para localizar a folha-filha direita, sendo "i" a posição da folha-pai. Os conteúdos desses três endereços são então comparados, e o maior é colocado na posição da folha-pai, e a folha-pai colocada na antiga posição do maior. Este processo, chamado de heapify na implementação, é executado de forma recursiva no elemento que foi trocado de posição com o antigo pai, e também nos próximos elementos que possivelmente sejam trocados com um elemento pai.

Dessa forma, ao final das execuções recursivas do processo *heapify*, o vetor está ordenado no formato *heap*, ou seja, uma árvore binária, cujo cada folha-pai (posição i) é maior que suas folhas-filhas (posições 2*i+1 para esquerda e 2*i+2 para direita).

Em seguida, é feita uma troca entre o primeiro e o último elemento do vetor, logo, o maior elemento está no final. O comprimento do vetor é decrementado em 1, e o *heapify* é aplicado da raiz da árvore até o comprimento do vetor. Esse processo de troca e *heapify* é feito até que o comprimento do vetor seja 1. Dessa forma, o vetor foi ordenado de forma crescente, porém de trás pra frente, colocando os maiores elementos no final e se movendo em direção ao início.

O grau de complexidade deste método é de $O(n*\log(n))$. Nesta estrutura, foi obtida uma média igual a 0,302 segundos, com um desvio padrão associado de 0,0047 segundos.

## 7. MergeSort

O método de ordenação mergesort também utiliza o princípio da divisão e conquista. Na implementação feita, a divisão do vetor em duas partes iguais, a ordenação dessas duas metades e a intercalação dos elementos de forma ordenada é feita de forma recursiva. Quando as subdivisões do vetor atingem o menor tamanho possível, as metades são ordenadas e intercaladas recursivamente, de modo a obter um conjunto ordenado de cada par de metades separadas previamente, e ao final da execução é obtido um único conjunto ordenado, ou seja, o vetor foi ordenado.

O grau de complexidade deste método é de O(n*log(n)). Nesta estrutura, foi obtida uma média igual a 0,109 segundos, com um desvio padrão associado de 0,0007 segundos.

## CONCLUSÃO

Foi visto que cada método possui um sistema diferente de ordenação. Para ser possível uma comparação entre os métodos, todos foram executados vinte vezes, ordenando um vetor de um milhão de elementos gerado de forma pseudo-aleatória. Com as vinte medições de tempo de cada método, foi calculada a média de tempo e o desvio padrão. Na tabela abaixo podem ser observados os resultados agrupados.

Tabela 1 - Média e desvio padrão de cada método.

| Método de Ordenação | N° de aquisição | Média [s] | Desvio Padrão [s] |
|---|---|---|---|
| **BubbleSort** | 20 | 1302,559 | 2,0267 |
| **QuickSort** | 20 | 0,115 | 0,0006 |
| **Inserção Direta** | 20 | 1083,35 | 2,237 |
| **Inserção Binária** | 20 | 1346,035 | 0,920 |
| **Seleção Direta** | 20 | 799,151 | 1,2712 |
| **Seleção HeapSort** | 20 | 0,302 | 0,0047 |
| **MergeSort** | 20 | 0,109 | 0,0007 |

Com estes resultados, conclui-se que o método mais rápido para o contexto definido é o *mergesort*, com o *quicksort* logo em seguida, com quase a

mesma média de tempo. Apesar de serem os mais rápidos, esses algoritmos pagam o preço na quantidade de memória utilizada, devido a quantidade de recursões necessárias e, no caso do *mergesort,* a necessidade de um vetor temporário.

O *heapsort* fica em terceiro lugar, com quase o triplo de tempo dos dois primeiros. Apesar de não ser a mais rápida, seu consumo de memória em relação aos dois primeiros é menor, logo trata-se de um ótimo custo benefício.

Esses três primeiros métodos tiveram ótimos tempos, convergindo em menos de um segundo. Já o método seleção direta ficou na casa de centenas de segundos, e os restantes chegaram a passar de mil segundos. Logo, é possível concluir que os últimos quatro métodos não são recomendados para ordenação de grandes quantidades de dados. Já em pequenas quantidades eles podem obter desempenhos melhores que os três primeiros citados, porém é necessária uma análise bem pontual para definir a partir de qual quantidade cada método vale a pena.

Os códigos implementados para este projeto, podem ser conferidos neste link: https://github.com/OgliariNatan/Sorting_methods.