

# Análise de Complexidade dos Métodos de Ordenação\*

Leonardo Aguilar Murça<sup>1</sup>

<sup>1</sup>Departamento de Informática – Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais – Campus Sabará (IFMG)  
CEP - 34.590-390 – Belo Horizonte – MG – Brasil

[leonardoamurca@gmail.com](mailto:leonardoamurca@gmail.com)

**Abstract.** *This article aims to explain complexity analysis of the main sorting methods: BubbleSort, InsertSort, SelectSort, ShellSort and QuickSort. Besides that, it will contain the accounting of the number of comparisons, movements and the average time spent in each method.*

**Resumo.** *Este artigo tem como objetivo explicitar análise de complexidade dos principais métodos de ordenação: Bubble Sort, Insertion Sort, Selection Sort, Shell Sort e Quick Sort. Nessa análise conterà a contabilização do número de comparações, movimentações e o tempo médio gasto em cada método.*

**Todo o código fonte em que este relatório é baseado está hospedado em:**

<https://github.com/leonardoamurca/complexity-sort-methods>

## 1. Introdução

Foi proposto em sala de aula a análise de complexidade dos algoritmos de ordenação mais famosos para a disciplina de *Projeto e Análise de Algoritmos*. Tal proposta busca o aprimoramento das técnicas de análise apresentadas em classe e à prova da relação de comprovação da teoria através da prática.

Foi feita a análise dos seguintes algoritmos: *Bubble Sort, Insertion Sort, Selection Sort, Shell Sort e Quick Sort*.

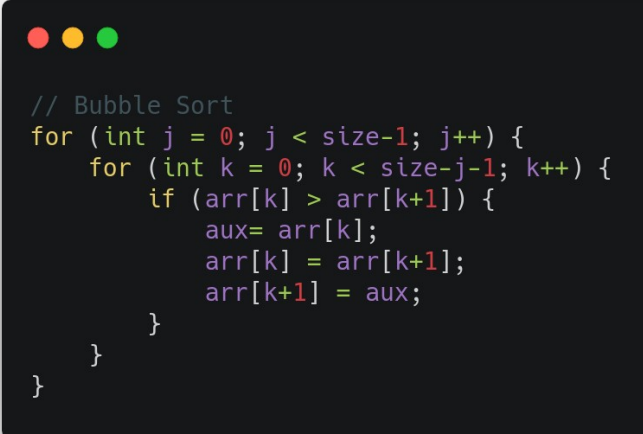
## 2. Descrição dos métodos

### 2.1 Bubble Sort

O conceito central do *Bubble Sort* é percorrer o vetor desordenado várias vezes e a cada passagem colocar ao topo o maior elemento da sequência.

No melhor caso (vetor inicialmente ordenado), é executado  $n$  iterações. Por outro lado, no pior dos casos (vetor ordenado de maneira decrescente), é executado  $n^2$  iterações. Logo, percebe-se que sua ordem de complexidade é  $O(n^2)$ .

Pode ser observado na imagem abaixo a implementação de tal algoritmo utilizando a linguagem de programação C++:



```
// Bubble Sort
for (int j = 0; j < size-1; j++) {
    for (int k = 0; k < size-j-1; k++) {
        if (arr[k] > arr[k+1]) {
            aux = arr[k];
            arr[k] = arr[k+1];
            arr[k+1] = aux;
        }
    }
}
```

*Illustration 1: Implementação do Bubble Sort em C++*

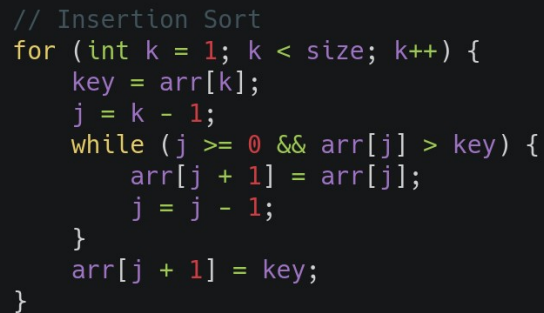
## 2.2 Insertion Sort

O Insertion Sort segue o preceito de ter um elemento chave, que inicialmente começa da segunda posição do vetor, que sempre é comparado ao próximo elemento e à todos anteriores à ele. Sendo o anterior maior que o elemento chave, suas posições são trocadas até que todos os elementos anteriores ao elemento chave estejam ordenados. Logo após, a chave passa a ser o próximo elemento da sequência. Isso tudo ocorre até que todo o vetor esteja completamente ordenado.

Esse método de ordenação demonstra ser eficiente para instâncias pequenas como os algoritmos de ordem quadrática.

No seu melhor caso (vetor inicialmente ordenado), é executado  **$n$**  iterações. Por outro lado, no pior dos casos (vetor ordenado de maneira decrescente), é executado  **$n^2$**  iterações. Logo, percebe-se que sua ordem de complexidade é  **$O(n^2)$** .

Observe logo abaixo a sua implementação em linguagem C++:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in C++ and implements the Insertion Sort algorithm. The code is as follows:

```
// Insertion Sort
for (int k = 1; k < size; k++) {
    key = arr[k];
    j = k - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = key;
}
```

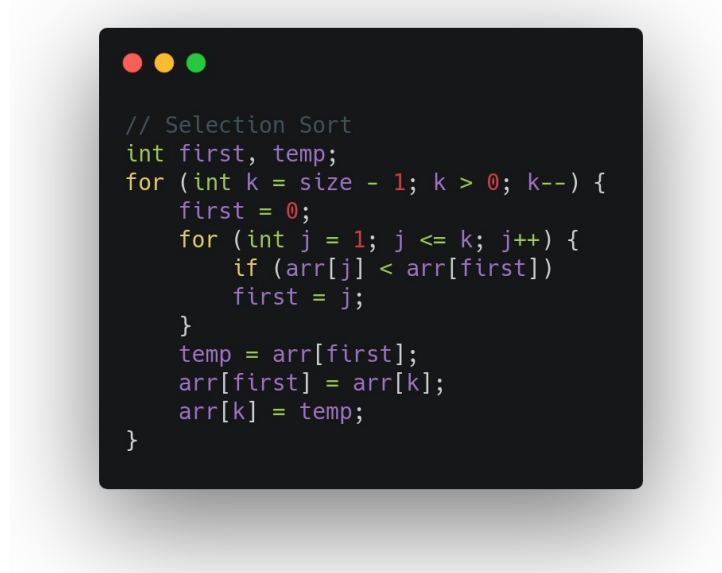
*Illustration 2: Implementação do Insertion Sort em C++*

### 2.3 Selection Sort

O Selection Sort trabalha com a premissa de sempre passar o elemento de menor (ou maior, dependendo da lógica utilizada) valor para a primeira posição, depois o de segundo menor valor para a segunda posição, e assim é feito para os **n-1** elementos de nossa instância.

Esse método compara cada elemento com os demais elementos a cada iteração, ou seja, não há distinção do melhor caso para o pior caso, logo sua complexidade será sempre **O(n<sup>2</sup>)**.

Veja logo abaixo a implementação do Insertion Sort utilizando a linguagem C++:



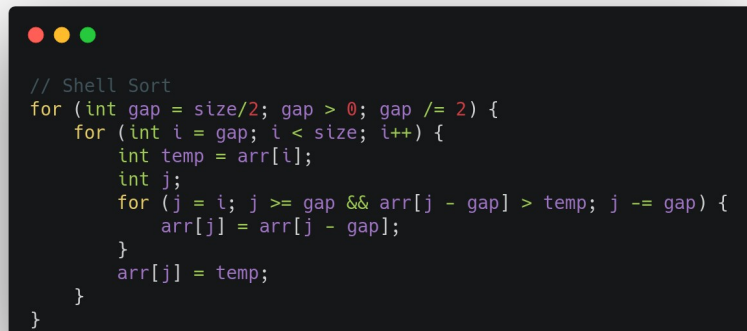
*Illustration 3: Implementação do Selection Sort em C++*

## 2.4 Shell Sort

O Shell Sort utiliza da lógica do Insertion Sort atrelado à estratégia de dividir para conquistar. Ele basicamente passa pelo vetor dividindo o grupo maior em menores, e nesses menores é utilizado a ordenação por inserção (Insertion Sort). A divisão dessas parcelas menores é definida pela lacuna (gap) que será escolhida pelo desenvolvedor.

Sua complexidade, depende diretamente do gap que é utilizado em sua implementação. Porém, para o melhor caso conhecido, utilizando um gap de 2, sua complexidade é  **$O(n \log n)$** . Já para o pior caso, é de ordem quadrática:  **$O(n^2)$** .

Observe abaixo sua implementação:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains C++ code for the Shell Sort algorithm. The code uses nested loops to iterate over the array with a decreasing gap, comparing and shifting elements as needed.

```
// Shell Sort
for (int gap = size/2; gap > 0; gap /= 2) {
    for (int i = gap; i < size; i++) {
        int temp = arr[i];
        int j;
        for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
            arr[j] = arr[j - gap];
        }
        arr[j] = temp;
    }
}
```

## 2.5 Quick Sort

O quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

Sua complexidade para o pior caso é quadrática:  **$O(n^2)$** . Já tanto para o caso médio quanto para o melhor caso, sua complexidade é logarítmica:  **$O(n \log n)$** .

Observe abaixo sua implementação:

```

void quickSort(int arr[], int l, int h) {
    comparisons++;
    if (l < h) {
        int p = partition(arr, l, h);
        quickSort(arr, l, p - 1);
        quickSort(arr, p + 1, h);
    }
}

int partition(int arr[], int l, int h) {
    int x = arr[h];
    int i = (l - 1);
    int aux;
    for (int j = l; j <= h - 1; j++) {
        c++;
        if (arr[j] <= x) {
            i++;
            aux = arr[i];
            arr[i] = arr[j];
            arr[j] = aux;
        }
    }
    aux = arr[i+1];
    arr[i+1] = arr[h];
    arr[h] = aux;

    return (i + 1);
}

```

### 3. Análise dos resultados

Foram feitas simulações de ordenação com vetores de tamanho 100, 1000 e 10000 elementos. Além disso, foram gerados vetores crescentes, decrescentes, randômicos e parcialmente randômicos. Os resultados gerados estão todos na pasta *OutputFiles* deste projeto.

Foi observado que todos os métodos simulados condizeram com suas respectivas equações para o número de comparações e movimentações, consequentemente se equivalendo à sua ordem de complexidade.

Para todas as simulações do Bubble Sort, obteve-se os resultados equivalentes:

Número de comparações:  $n(n-1)/2$ .

Resultados simulados para vetor decrescente de 100 elementos:

Sort Method: Bubble

Array Type: OrdD

Runtime: 0.032 ms

Comparisons: 4950

Movimentations: 14850

Substituindo o tamanho do vetor na equação de comparações obtivemos:

$$(100 * (100 - 1)) / 2 = 4950$$

Para todas as simulações do Selection Sort, obteve-se os resultados equivalentes:

Número de comparações:  $(n^2 - n) / 2$

Resultados simulados para vetor decrescente de 100 elementos:

Sort Method: Insertion

Array Type: OrdD

Runtime: 0.039 ms

Comparisons: 4950

Movimentations: 297

Substituindo o número de elementos do vetor na equação obtivemos:

$$(100^2 - 100) / 2 = 4950$$

Já para as simulações do Selection Sort, temos os seguintes resultados equivalentes:

Número de comparações:  $n - 1$

Resultados simulados para vetor decrescente de 100 elementos:

Sort Method: Selection

Array Type: OrdD

Runtime: 0.056 ms

Comparisons: 99

Movimentations: 5148

Substituindo o número de elementos do vetor na equação obtivemos:

$$100 - 1 = 99$$

Para as simulações do Shell Sort, não é definida uma complexidade fixa, pois ele depende diretamente do gap a ser utilizado. Porém, há um intervalo em que podemos classificar tal análise.

Para seu pior caso sua complexidade equivale a  $O(n \log n)$  e em seu melhor caso tem-se  $O(n \log^2 n / (\log \log n)^2)$ .

Por fim, a complexidade do Quick Sort também será de ordem quadrática para o pior caso. Além disso, tal algoritmo utiliza uma quantidade de memória da máquina menor do que os demais métodos.

#### 4. Conclusão

Ao implementar e analisar os principais métodos de ordenação, foi observado que não existe o melhor e mais eficiente método de ordenação em detrimento de todos os outros, mas sim aquele método que resolve o problema específico da maneira mais eficiente. Para que isso aconteça, deve-se ponderar em qual contexto cada algoritmo terá uma melhor performance. Por exemplo: o Bubble Sort é extremamente ineficiente para instâncias muito grandes, diferentemente do Quick Sort, que perfoma de uma maneira muito eficiente para tais tipos de instâncias.

Além da experiência na análise dos algoritmos, esse Trabalho Prático proporcionou o aprendizado de várias técnicas de implementação na linguagem C++, como: Classes, Objetos, contextos para passagem de parâmetros, gerenciamento de memória e abstração de dados.

#### Referências

Knuth, D. E. (1984), *The TeXbook*, Addison Wesley, 15<sup>th</sup> edition.

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

Khanacademy, Analysis of Quicksort. Available at:  
<<https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort>>

Devmedia, Algoritmos de ordenação: análise e comparação. Disponível em:  
<<https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261>>

Naugarduko, Vilnius . An empirical study of the gap sequences for Shell sort.

Shellsort & Algorithmic Comparisons, 2008. Available at:  
<<https://www.cs.wcupa.edu/rkline/ds/shell-comparison.html>>