

# Algoritmos de ordenação básicos em Python

Prof. Luiz Henrique da Costa Araujo

## Descrição

Apresentação dos conceitos de ordenação, Comparação dos algoritmos de ordenação segundo óticas distintas, Aplicação das métricas de classificação, Apresentação dos algoritmos de ordenação da bolha, da inserção e da seleção implementados em linguagem Python.

## Propósito

Algoritmos de ordenação de uma sequência de chaves é, sem dúvida, um dos problemas mais estudados em computação. Obter conhecimento sobre os algoritmos de ordenação elementares permitirá que você consiga escolher o método de ordenação mais adequado ao problema a ser solucionado, para que possa desenvolver um algoritmo que seja eficiente e rápido.

## Objetivos

---

### Módulo 1

#### Conceitos fundamentais de ordenação para aplicação de métricas

Reconhecer os conceitos fundamentais de ordenação para aplicação de métricas de classificação e escolha adequada do método na resolução de problemas.

---

### Módulo 2

#### Algoritmos de ordenação da bolha, inserção e seleção

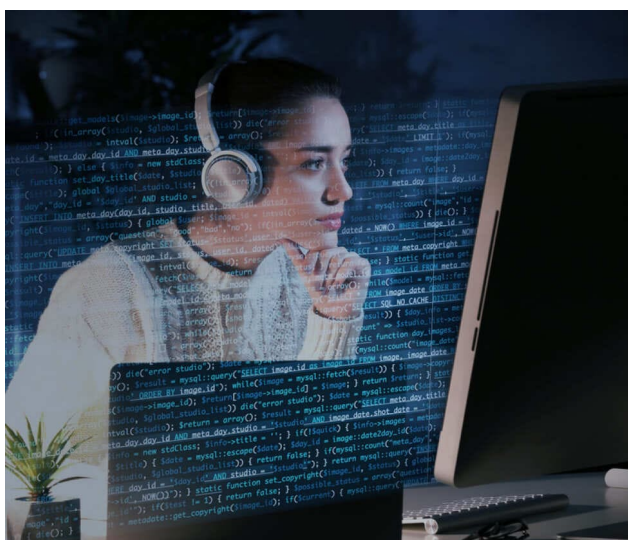
Distinguir os algoritmos de ordenação da bolha, inserção e seleção para desenvolvimento de solução rápida e eficiente para implementação em linguagem Python.

---

# Introdução

A ordenação de dados de computador é um assunto de extrema importância. Já imaginou a quantidade de tempo necessária para buscar um contato na agenda de um celular, se os nomes não estivessem ordenados em ordem alfabética. Ou ainda, buscar um certo valor numérico em uma lista, se esta também não estivesse ordenada. Caso os dados não estejam organizados em uma forma bem definida, teríamos que percorrer toda a lista na hora que fizessemos uma consulta. E isso demandaria muito tempo.

Os algoritmos de ordenação atuam na organização de palavras ou de uma lista de números conforme sua necessidade. Tal procedimento torna a busca por um elemento mais eficiente. Dentro da linguagem Python, vamos abordar os conceitos fundamentais de ordenação, distinguindo os algoritmos de bolha, inserção e seleção.



## 1 - Conceitos fundamentais de ordenação para aplicação de métricas

Ao final deste módulo, você será capaz de reconhecer os conceitos fundamentais de ordenação para aplicação de métricas de classificação e escolha adequada do método na resolução de problemas.

## Conceitos de ordenação

### Ordenação

O problema da ordenação é, sem dúvida, um dos mais antigos e mais estudados em computação. É a ferramenta para a solução de diversos problemas mais complexos e existem diversos algoritmos que resolvem este problema.

Os algoritmos de ordenação podem ser classificados segundo diversos parâmetros:

1. Complexidade computacional.
2. Complexidade de espaço.
3. Ordenação interna ou externa.
4. Caráter recursivo.
5. Estabilidade.

Cada uma destas características é importante e será discutida a seguir.

## Complexidade computacional

O estudo da complexidade computacional é um pré-requisito para o estudo dos algoritmos de ordenação.

Entretanto, em linhas gerais, podemos dizer que determinar a complexidade computacional de um algoritmo é encontrar uma função matemática.

$$f : N \rightarrow R, \text{ tal que } kf(n), \text{ onde } k > 0 \in R \text{ e } n \in N \text{ é o tamanho da instância.}$$

Rotacione a tela. 

A função  $f$  é cota assintótica superior para o número de operações elementares necessárias para um algoritmo  $A$  resolver todas as instâncias de tamanho  $n$ .

A cota assintótica descrita acima define a notação  $O$ , isto é, diz-se que  $g$  é  $O(f)$  se existe um  $n_0 \in N$  e um  $k > 0 \in R$ , tal que para todo  $n > n_0$ ,  $k \cdot f(n) > g(n)$ .

A introdução da notação  $O$  permite que a análise de complexidade computacional classifique os algoritmos em classes que determinam seu desempenho. Os algoritmos mais eficientes são aqueles que executam em um tempo proporcional a uma constante e, a partir daí, temos o aumento do tempo de execução segundo as ordens definidas na tabela a seguir.


Ordem	Tempo de execução
$O(1)$	
$O(\log n)$	
$O(n)$	
$O(n \log n)$	
$O(n^2)$	

Tabela: Complexidade computacional dos algoritmos.  
Elaborada por: Luiz Henrique da Costa Araujo.



## Analisando a complexidade computacional

Neste vídeo, abordamos os algoritmos de ordenação fazendo uma análise de complexidade de algoritmo.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A frieza da Matemática não traduz com precisão a separação existente nesta classificação. Para exemplificar, vamos comparar uma busca linear em um vetor, que procura por todos os elementos do vetor, com a busca binária.

A ideia básica do algoritmo de busca binária é simples. Seja **V** um vetor ordenado. A busca binária compara a chave buscada com o elemento central do vetor ( **$n/2$** , onde **n** é o tamanho do vetor). Caso a chave seja igual ao elemento buscado, temos o sucesso, caso contrário, a chave buscada pode ser maior ou menor. Conforme o caso, buscamos recursivamente na metade superior ou inferior do vetor.

Este algoritmo tem complexidade  **$O(\log n)$**  e a busca linear  **$O(n)$** .

### Exemplo

Imaginemos agora que desejamos buscar uma chave em um vetor de  $10^6$  elementos.

Com a busca linear, teríamos que realizar  $10^6$  comparações. Aplicando a busca binária, teremos  $\log_2 10^6 \cong 20$  comparações. Isto é, a mudança na classificação de ordem implica em uma mudança significativa na quantidade de operações elementares executadas.

Existe um resultado importante em relação à complexidade computacional dos algoritmos de ordenação, que são baseados na comparação entre dois elementos do vetor que desejamos ordenar. **Existe um limite inferior para complexidade destes algoritmos, que é  $O(n \log n)$ .**

Podemos visualizar este resultado modelando a ordenação como um problema de decisão. Assim, pode-se montar uma árvore de decisão que modele as comparações entre elementos deste vetor. Cada nó interno da árvore de decisão é a comparação entre dois elementos do vetor, e as folhas desta árvore as permutações obtidas pelo algoritmo. Como temos  $n!$  permutações, temos  $n!$  folhas nesta árvore de decisão. Sabemos também que a árvore de decisão é uma árvore binária (a comparação entre dois elementos  $x$  e  $y$  só pode fornecer dois resultados:  **$x < y$  ou  $x > y$** ).

Sendo assim, os algoritmos mais eficientes de ordenação geram árvores de decisão de altura mínima, isto é, árvores completas. Ou seja, se determinarmos a altura da árvore de decisão mínima (**árvore completa**), determinamos a quantidade mínima de comparações para ordenar um vetor.

Como a árvore é completa com  $n!$  folhas, sabe-se que esta árvore possui  **$2(n!)$**  nós, porém  **$2(n!) < n^n$** . Sabemos também que a altura da árvore completa é proporcional a  **$\log k$** , onde  **$k$**  é o número de nós da árvore. Logo, a altura da árvore de decisão é menor que  **$\log(n^n) = n \log n$** , o que mostra o resultado.

## Complexidade de espaço

A complexidade de espaço mede a quantidade de memória necessária para que o algoritmo seja executado. O usual é que o algoritmo execute necessitando  $O(n)$  bytes, isto é, somente o espaço necessário para armazenar o vetor contendo as informações. Entretanto, é possível trocar espaço de memória por complexidade computacional.

Por exemplo, suponhamos que desejamos ordenar números inteiros contidos no intervalo **[0, 100000]**, armazenados em um vetor **V** de tamanho **n**. Podemos instanciar um vetor **AUX** de tamanho **100000**, inicializar este vetor com valor zero, por exemplo, e percorrer o vetor a ser ordenado da seguinte forma:

Pseudocode



```
1 Para i = 1 até n
2   aux [V[i]]=1;
```

Após esta operação, percorremos o vetor auxiliar, colocando em **V** os valores do índice que não possuem valor zero.

Pseudocode



```
1 j=0;
2 para i = 0 ate 100000
```

```
3   se (aux[i] == 1)
4       inicio
5       V[j]=i;
6       j++;
```

Após o término do loop acima, o **vetor V** estará ordenado.

A análise da complexidade computacional nos remete a um algoritmo que executa em  **$O(n)$** . Entretanto, a análise não leva em conta que o algoritmo aloca memória proporcional ao tamanho do conjunto universo das chaves (**observe que o conjunto universo é o intervalo [0-100000]**).

**Ou seja, a análise teórica da complexidade computacional do algoritmo nos indica um algoritmo de ordenação muito eficiente em termos de complexidade de tempo, porém, muito ineficiente em termos de complexidade de espaço, uma vez que necessita de uma memória proporcional ao tamanho do universo.**

Observe que a análise de complexidade de tempo para este caso é tendenciosa. Isto é, despreza o fato de que a memória precisa ser inicializada e este passo consome recursos computacionais. Algoritmos cuja linearidade não é proporcional ao tamanho da instância, e sim ao tamanho do conjunto universo são chamados de algoritmos pseudolineares e, normalmente, possuem complexidade computacional alta.

## Ordenação interna x externa

Esta característica refere-se ao local onde os dados estão armazenados. Como sabemos, todo computador tem a memória principal ou memória RAM e a memória secundária, no caso, disco rígido ou memória de estado sólido.

São duas as principais diferenças entre estes tipos de memória:

1. A forma de acesso.
2. A velocidade de acesso.

A memória principal é acessada aleatoriamente, isto é, pode-se acessar byte a byte da memória independentemente. Já o acesso à memória secundária é feito por blocos, isto é, a menor porção que pode ser recuperada é um bloco, normalmente múltiplo de **512 Kbytes**.

Em relação à velocidade, a memória principal é muito mais rápida que a secundária. O tempo de acesso à memória principal é da ordem de nanosegundo  $1 \times 10^{-9}$ , enquanto a memória secundária é acessada em milissegundo  $1 \times 10^{-3}$ .

Os algoritmos de ordenação interna executam somente com todos os dados presentes na memória principal, enquanto os algoritmos de ordenação externa são capazes de ordenar dados contidos em memória secundária.

### Atenção!

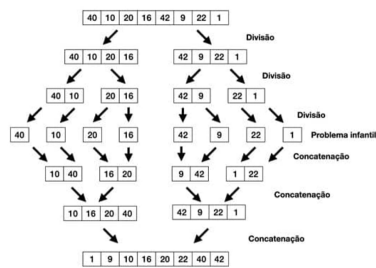
Vale destacar que os algoritmos de ordenação externa não são capazes de abstrair completamente a memória principal, isto é uma impossibilidade no paradigma de von Neumann. Para a execução de um programa, o código e os dados devem estar em memória principal. Assim sendo, um algoritmo de ordenação externa copia parte dos dados para a memória principal, trata estes dados e armazena o resultado (a sequência ordenada) em memória secundária.

## Caráter recursivo

Outra característica dos algoritmos de ordenação é o caráter recursivo. Alguns algoritmos são recursivos, normalmente aplicando a estratégia dividir para conquistar. Outros são sequenciais.

Um exemplo de algoritmo recursivo de ordenação é o Merge Sort. A ideia do algoritmo é dividir o vetor a ser ordenado em duas metades, ordenar estas metades e intercalar o resultado da ordenação obtendo o vetor ordenado.

A ordenação das metades do vetor é feita aplicando o algoritmo recursivamente. A imagem a seguir mostra o exemplo do processo.



Merge Sort.

A imagem acima ilustra o processo do algoritmo recursivo. Na primeira linha do diagrama, temos o vetor original de oito elementos, que é dividido em duas metades de quatro elementos. Cada um destes vetores de quatro elementos é dividido em dois vetores de dois elementos (terceira linha do diagrama). Os vetores de dois elementos são divididos em vetores de um único elemento, os problemas infantis, isto é, que naturalmente já estão resolvidos.

Em seguida, inicia-se o passo da concatenação. No passo da concatenação, juntam-se, sempre dois a dois, vetores do nível anterior. Isto é, vetores de um elemento são aglutinados em vetores de dois elementos, de dois elementos em quatro elementos e de quatro elementos no vetor final.

O processo de concatenação de vetores é simples. Basta selecionar o menor elemento entre os dois vetores e colocar na primeira posição do novo vetor. Em seguida, pega-se o segundo menor elemento, colocando na segunda posição. Repete-se este processo até todos os elementos serem concatenados em um único vetor. O resultado final é o vetor ordenado.

Em particular, este algoritmo é baseado na comparação entre dois elementos do vetor, isto é, feito no passo de concatenação e, por isso, o algoritmo não pode ter complexidade inferior a  $O(n \log n)$ . A complexidade deste algoritmo é, de fato,  $O(n \log n)$ . Para realizar a análise, não é necessário estudar o código do algoritmo. A imagem é um bom instrumento de análise.

Na imagem Merge Sort, cada linha representa a divisão do vetor em duas metades e de suas metades recursivamente (daí o caráter recursivo do algoritmo). Vejamos quantas linhas temos até a linha dos problemas infantis.

**Linha 1** vetor original  $n/2^0$

**Linha 2** metade do vetor original  $n/2^1$

**Linha 3** metade da metade do vetor original  $n/2^2$

.....

**Linha k** dos problemas infantis  $n/2^{k-1}$

Porém, sabemos que na linha  $k$  os problemas são de tamanho  $= 1$ , assim  $1 = n/2^{k-1}$ ,  $\log n = 2^{k-1}$ . Aplicando  $\log_2$  temos  $\log_2 n = \log_2 2^{k-1}$ , que fornece  $k - 1 = \log_2 n$ ,  $k = \log_2 n + 1$

Ou seja, temos  $\log_2 n + 1$  linhas. Na Figura 1,  $n = 8$ ,  $\log_2 8 = 3$ ,  $k = 4$  linhas até o problema infantil. O processo se repete na concatenação. Porém, na concatenação acessamos uma vez cada elemento do vetor, assim, em cada linha temos uma complexidade de  $n$  como temos  $\log_2 n$  linhas para concatenar. A complexidade do algoritmo é  $O(n \log n)$ .

## Estabilidade

Diz-se que um algoritmo de ordenação é estável quando elementos que são apresentados já na ordem correta são mantidos durante a execução do algoritmo. Assim, se for apresentada uma sequência já ordenada para um algoritmo estável, o algoritmo não irá realizar nenhuma operação de troca.

Um efeito da estabilidade é a possível redução do número de operações elementares necessárias para execução do algoritmo. Porém, o efeito da estabilidade é percebido nas instâncias com os melhores casos e não nos piores casos. Assim, a estabilidade do algoritmo não tem impacto na complexidade computacional teórica.

## Algoritmos não são baseados em comparação entre elementos do vetor

Os algoritmos de ordenação mais comuns se baseiam na comparação entre dois elementos da sequência a ser ordenada e alguma operação, normalmente a troca de posição, sobre os elementos comparados. Entretanto, existem algoritmos capazes de ordenar uma sequência sem comparar os elementos da sequência dois a dois.

Para estes algoritmos, o teorema que determina o limite inferior da ordenação como  $O(n \log n)$  não se aplica, uma vez que sua premissa é invalidada, isto é, a árvore de decisão não é "montada" pelo algoritmo. Portanto, é teoricamente possível existirem algoritmos de ordenação desta classe que possuem complexidade inferior a  **$O(n \log n)$** .

## Exemplo

Um algoritmo de ordenação não baseado em comparações entre elementos da sequência é o bucket sort ou método do balde. Este algoritmo aplica-se quando desejamos ordenar uma sequência de inteiros, na qual conhecemos a quantidade máxima de dígitos desses inteiros.

O princípio de funcionamento do algoritmo é simples. Executamos uma iteração para cada ordem numérica, da maior ordem para a menor ordem. Em cada iteração, separamos por dígito (**de 0 até 9**) pertencente à ordem analisada. Na próxima iteração, analisamos a ordem numérica inferior. Ao final, concatenamos a sequência obtendo a ordenação.

Por exemplo, seja a sequência com números de três dígitos: 005, 235, 014, 236, 423, 456, 890.

## Etapa 1

Na primeira iteração, analisamos a ordem mais significativa, que é a centena, separando os números por centena.

Centena 0: 005, 014

Centena 1: -

Centena 2: 235, 236

Centena 3: -

Centena 4: 423, 456

Centena 5: -

Centena 6: -

Centena 7: -

Centena 8: 890

Centena 9: -

## Etapa 2

Para cada centena, separamos por dezena:

Centena 0:

Dezena 0: 005

Dezena 1: 014

Centena 2:

Dezena 3: 235, 236

Centena 4:

Dezena 2: 423

Dezena 5: 456

Centena 8:

Dezena 9: 890

## Etapa 3

Para cada dezena, dentro das centenas, separamos por unidade:



Centena 0:

Dezena 0:

Unidade 5: 005

Dezena 1:

Unidade 4: 014

Centena 2:

Dezena 3:

Unidade 5: 235

Unidade 6: 236

Centena 4:

Dezena 2:

Unidade 3: 423

Dezena 5:

Unidade 6: 456

Centena 8:

Dezena 9:

Unidade 0: 890

**Concatenando tudo na ordem (de cima para baixo): 005, 014, 235, 236, 423, 456, 890.**

Alguns autores citam a complexidade deste algoritmo como  $O(n)$ . A argumentação é que, como a quantidade de dígitos  $k$  é conhecida e constante, e para cada ordem numérica percorremos a sequência a ser ordenada uma vez, temos  $k * n$  que representa uma complexidade de  $O(n)$ .

Entretanto,  $k$  é a quantidade máxima de dígitos do número e a função matemática que fornece a quantidade de dígitos de um número em **base 10** é  $\log(n)$ . Por esta razão, alguns autores citam este algoritmo com complexidade  $O(n \log n)$ .

## Falta pouco para atingir seus objetivos.

### Vamos praticar alguns conceitos?

#### Questão 1

Analise as afirmativas abaixo e marque a opção correta.

1 – Não existe algoritmo de ordenação capaz de executar em um tempo inferior à complexidade de  $O(n \log n)$ .

2 – Métodos de ordenação interna executam em memória principal (RAM) somente, enquanto métodos de ordenação externa executam em memória secundária, porém podem usar memória principal também.

3 – Somente algoritmos de ordenação não baseados em comparação podem executar em um tempo inferior à  $O(n \log n)$ .

A

Somente a afirmativa 1 está correta.

B

Somente a afirmativa 2 está correta.

C

Somente a afirmativa 3 está correta.

D	As afirmativas 2 e 3 estão corretas.
E	As afirmativas 1, 2 e 3 estão corretas.

Parabéns! A alternativa D está correta.

Letra D, a afirmativa 1 está incorreta, algoritmos não baseados em comparação entre elementos da sequência a ser ordenada não estão sujeitos ao limite de complexidade de  $O(n \log n)$ .

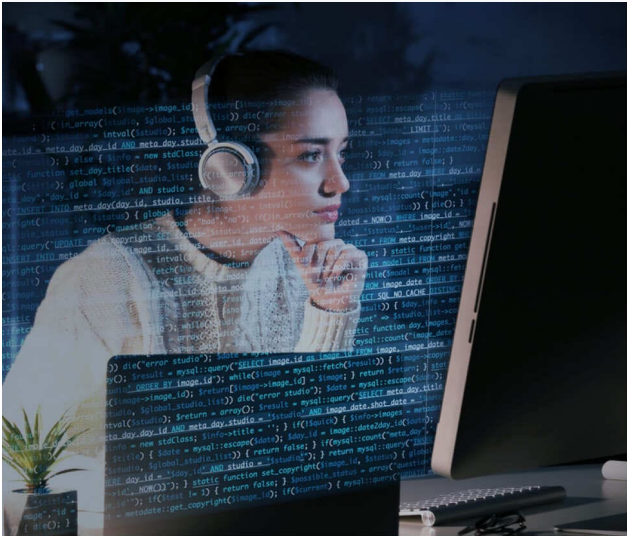
Questão 2

Marque a alternativa correta em relação à estabilidade de um algoritmo de ordenação.

A	Um algoritmo instável possui necessariamente complexidade computacional alta.
B	Um algoritmo instável tem complexidade de espaço superior à $O(n)$ .
C	A estabilidade pode reduzir o número de operações em algumas instâncias, porém não reduz a complexidade computacional do algoritmo.
D	Algoritmos estáveis sempre executam em uma complexidade de $O(n \log n)$ .
E	Algoritmos internos sempre são estáveis.

Parabéns! A alternativa C está correta.

A estabilidade pode reduzir o número de operações em algumas instâncias, porém não reduz a complexidade computacional do algoritmo.



## 2 - Algoritmo de ordenação da bolha, inserção e seleção

Ao final deste módulo, você será capaz de distinguir os algoritmos de ordenação da bolha, inserção e seleção para desenvolvimento de solução rápida e eficiente para implementação em linguagem Python.

### Algoritmos de ordenação

#### Método da bolha (Bubble sort)

Seja  $S = s_1, s_2, \dots, s_n$  uma sequência de números inteiros distintos. Uma forma de verificar se a sequência  $S$  está ordenada é realizar o seguinte conjunto de operações: Para  $1 \leq i < n$  verificar se  $s_i < s_{i+1}$ , caso isto ocorra, para todos os valores de  $i$ , a sequência está ordenada.

##### Exemplo

Por exemplo, para a sequência  $s = s_1 = 10, s_2 = 12, s_3 = 15, s_4 = 20, s_5 = 25$ , temos que  $s_1 < s_2, s_2 < s_3, s_4 < s_5$ . Por esta razão, podemos afirmar que a sequência está ordenada.

A ideia do método da bolha é explorar esta propriedade com o objetivo de ordenar o vetor.

No exemplo acima,  $s = s_1 = 10, s_2 = 12, s_3 = 15, s_4 = 20, s_5 = 25$ , realizamos uma iteração, ou seja, todas as comparações, e verificamos que a propriedade é válida para todo par de elementos adjacentes. Isto garante a ordenação da sequência. De fato, se  $s_i < s_{i+1}$  e  $s_{i+1} < s_{i+2}$  então  $s_i < s_{i+2}$  ou seja, a comparação é transitiva e assim se garante que a relação é válida para qualquer par da sequência  $s_i < s_j$ , se  $i < j$ .

O método da bolha explora esta propriedade realizando a troca de posição caso a comparação entre elementos adjacentes falhe.

Vejamos um exemplo, seja a sequência  $S = 15, 20, 8, 16, 40$ . No ábaco abaixo, vamos destacar o par que está sendo comparado a cada passo. Caso a comparação seja bem-sucedida, isto é,  $s_i < s_{i+1}$ , nada é realizado, caso contrário, efetuamos a troca.

Na iteração completa que não ocorra troca, teremos a sequência ordenada.

#### Etapa 1

Início da 1ª iteração

Passo 1:  $15 < 20, 8, 16, 40$

Passo 2:  $15, 20 > 8, 16, 40$

Passo 3:  $15, 8, 20 > 16, 40$

Passo 4:  $15, 8, 16, 20 < 40$

Fim da 1ª iteração

#### Etapa 2

Início da 2ª iteração

Passo 1:  $15 > 8, 16, 20, 40$

Passo 2:  $8, 15 < 16, 20, 40$

Passo 3:  $8, 15, 16 < 20, 40$

Passo 4:  $8, 15, 16, 20 < 40$

Fim da 2ª iteração

## Etapa 3

Início da 3ª iteração

Passo 1: 8 < 15, 16, 20, 40

Passo 2: 8, 15 < 16, 20, 40

Passo 3: 8, 15, 16 < 20, 40

Passo 4: 8, 15, 16, 20 < 40

Fim da 3ª iteração – Como não houve troca, a sequência está ordenada.

Entendido o princípio de funcionamento do algoritmo, vejamos a complexidade computacional para obter a sequência ordenada.

O primeiro passo para o estudo da complexidade computacional do algoritmo é a determinação do pior caso. Se analisarmos o ábaco acima, poderemos verificar que cada iteração tem quatro passos, isto é, **n-1** comparações, e que o algoritmo para as n-1 comparações têm sucesso.

Outra característica interessante é que, em uma iteração, o valor que é trocado vai para a posição dele no vetor ordenado. Isto é, na primeira iteração, o maior valor vai para última posição, na segunda iteração, o segundo maior valor vai para penúltima posição, e assim sucessivamente. Deste modo, o pior caso para o método da bolha é quando apresentamos a sequência ordenada em ordem reversa na entrada do algoritmo.

Para fins de análise, seja a sequência  $S = s_1 < s_2 < s_3 < \dots < s_n$ . Se ela for apresentada em ordem reversa para o algoritmo da bolha, teremos na entrada  $S' = s_n, \dots, s_3, s_2, s_1$ . Assim, a cada iteração teremos os seguintes resultados parciais:

**1ª iteração:**  $s_{n-1}, s_{n-2}, \dots, s_3, s_2, s_1, s_n$  – após n-1 comparações e trocas

**2ª iteração:**  $s_{n-2}, \dots, s_3, s_2, s_1, s_{n-1}, s_n$  – após n-1 comparações e trocas

**n-2ª iteração:**  $s_2, s_1, s_3, \dots, s_{n-2}, s_{n-1}, s_n$  – após n-1 comparações e trocas

**n-1ª iteração:**  $s_1, s_2, s_3, \dots, s_{n-2}, s_{n-1}, s_n$  – após n-1 comparações e nenhuma troca

A operação elementar do algoritmo é a comparação, não a troca, e a comparação sempre ocorre. Assim, para ordenar o vetor, o algoritmo executa  $(n-1)^2$  comparações, isto é, o algoritmo executa em  $O(n^2)$ . Ainda em relação à complexidade, o método da bolha só executa  $O(n^2)$  operações no pior caso.

### Comentário

O estudo de complexidade do algoritmo é feito sempre no pior caso, entretanto, a quantidade de comparações pode variar, chegando, no melhor caso (que é o fornecimento do vetor já ordenado em ordem crescente), a n-1 comparações.

Vejamos as outras características do algoritmo: É estável, ou seja, caso um elemento esteja posicionado corretamente em relação ao seu sucessor e antecessor na sequência, o algoritmo não efetua troca. É não recursivo. Tem baixa complexidade de espaço  $O(n)$ , isto é, só utiliza a memória necessária para armazenar a sequência a ser ordenada e é um método de ordenação interna.

O código em Python do algoritmo é o seguinte:

Método da bolha em Python.

 TUTORIAL  COPIAR

Python3

```
1 def bolha(vetor):
2     for n in range(len(vetor)-1, 0, -1):
3         trocou = False # para indicar se houve trocas
4
5         for i in range(n):
6             if vetor[i] > vetor[i+1]:
```

null

null

## Método da seleção (Selection Sort)

O método da ordenação por seleção é um algoritmo simples.

O algoritmo é iterativo e parte do princípio que a sequência não está ordenada e que todos os elementos da sequência estão fora de sua posição.

Na primeira iteração, o algoritmo analisa a sequência  $s = s_1, s_2, \dots, s_n$  e determina o menor elemento desta sequência. Seja  $s_k = s'_1$  este elemento. Em seguida, o algoritmo troca  $s'_1$  com  $s_1$ . Obtendo a sequência  $S = s'_1, s_2, \dots, s_1, \dots, s_n$ .

Na segunda iteração, como  $s'_1$  é o menor elemento,  $s'_1$  está na posição correta. Ou seja,  $s'_1$  é o primeiro elemento da sequência ordenada. Nesta iteração, o algoritmo irá determinar o menor elemento entre  $s_2, \dots, s_1, \dots, s_n$ , que será o segundo menor elemento da sequência ordenada. Seja  $s_j$  o menor elemento entre  $s_2, \dots, s_1, \dots, s_n$ , seja  $s'_2 = s_j$ . Trocando  $s_j = s'_2$  com  $s_2$ , temos a sequência  $s'_1, s'_2, s_3, \dots, s_1, \dots, s_2, \dots, s_n$ . Ao término da segunda iteração,  $s'_1, s'_2$  correspondem aos dois primeiros termos da sequência ordenada.

Repetimos cada iteração  $n - 1$  vezes, assim teremos a sequência ordenada completando a execução do algoritmo. Utilizando a sequência 13, 25, 8, 19, 7, 52 como exemplo, vamos executar o algoritmo.

**1ª iteração: 13, 25, 8, 19, 7, 52; 7 é o menor elemento**  
**2ª iteração: 7, 25, 8, 19, 13, 52; 8 é o menor elemento**  
**3ª iteração: 7, 8, 25, 19, 13, 52; 13 é o menor elemento**  
**4ª iteração: 7, 8, 13, 19, 25, 52; 19 é o menor elemento**  
**5ª iteração: 7, 8, 13, 19, 25, 52; 25 é o menor elemento**  
**6ª iteração: 7, 8, 13, 19, 25, 52; 52 é o menor elemento**  
**Fim de execução**

A complexidade computacional do algoritmo é  $O(n^2)$ . Este fato pode ser provado analisando o número de iteração e a quantidade de operações por iteração. O primeiro passo para a análise é determinar a operação fundamental do algoritmo. A cada iteração, necessitamos o valor mínimo de uma sequência de  $n$  elementos. A operação básica para isto é a comparação.

Na primeira iteração, temos que determinar o menor elemento de uma sequência de  $n$  elementos. Para isto, é necessário realizar  $n$  comparações. Na segunda iteração, temos que determinar o menor elemento de uma sequência de  $n-1$  elementos, e assim por diante. Assim, temos:

**1ª iteração:  $n-1$  comparações**  
**2ª iteração:  $n-2$  comparações**  
...  
 **$n-1$ ª iteração: 1 comparação**

Logo, o número de operações elementares é:  $n-1 + n-2 + \dots + 1 = (n)(n-1)/2$  que é  $O(n^2)$ . Um aspecto relevante na análise de complexidade do método da seleção é que não existe um tipo de instância de pior caso, para todas as instâncias o algoritmo executa de mesma forma em  $O(n^2)$ .

Em relação às demais características do algoritmo: É estável, porém, dependendo de como for implementado, esta característica pode ser perdida, entretanto não há impacto na complexidade, não recursivo, tem complexidade de espaço de  $O(n)$ .

O código em Python do algoritmo é o seguinte:

Python3

```
1 def selecao(vetor):
2     for i in range(len(vetor)):
3         for j in range(i + 1, len(vetor)):
4             if vetor[i] > vetor[j]:
5                 # realizando a troca
6                 vetor[i], vetor[j] = vetor[j], vetor[i]
```

null

null



O código acima apresenta a versão menos estável do algoritmo. Observe que trocamos o valor armazenado na posição  $i$  do vetor com o menor valor da sequência  $i+1$  até o tamanho do vetor. Uma forma de fazer isto é trocar todo elemento  $v[j] < v[i]$ , está conceitualmente correto, porém reduz a estabilidade do algoritmo.

Podemos melhorar a estabilidade do algoritmo realizando a execução em dois passos: No primeiro, elege-se o elemento mínimo e, no segundo passo, realiza-se a troca com o elemento na posição  $i$ . O programa *Método de seleção, implementação estável* está implementado desta forma.

Python3

```
1 def selecao(vetor):
2     for i in range(len(vetor)):
3         for j in range(i + 1, len(vetor)):
4             if vetor[i] > vetor[j]:
5                 # realizando a troca
6                 vetor[i], vetor[j] = vetor[j], vetor[i]
```

null

null



## Método da inserção (Insertion Sort)

O método de ordenação da inserção é um algoritmo simples e eficiente para listas “quase” ordenadas.

O princípio de funcionamento do algoritmo é dividir a lista em duas partes:

- Os elementos já ordenados.

- Os elementos a ordenar.

Inicialmente, a parte da lista que contém os elementos já ordenados é o primeiro elemento (uma lista unitária é sempre uma lista ordenada). Em seguida, devemos inserir o primeiro elemento da lista não ordenada (segundo elemento da lista na primeira iteração) na posição correta do segmento já ordenado da sequência. O processo é repetido até que não tenhamos mais elementos na lista não ordenada.

**Vamos analisar a execução do algoritmo através de um exemplo. Seja a sequência: 16, 8, 20, 18, 9, 2.**

**Início: 16, 8, 20, 18, 9, 2**

**1ª iteração: 8, 16, 20, 18, 9, 2**

**2ª iteração: 8, 16, 20, 18, 9, 2**

**3ª iteração: 8, 16, 18, 20, 9, 2**

**4ª iteração: 8, 9, 16, 18, 20, 2**

**5ª iteração: 2, 8, 9, 16, 18, 20**

O método de ordenação da inserção é um algoritmo simples e eficiente para listas “quase” ordenadas.

O princípio de funcionamento do algoritmo é dividir a lista em duas partes:

- Os elementos já ordenados.
- Os elementos a ordenar.

Inicialmente, a parte da lista que contém os elementos já ordenados é o primeiro elemento (uma lista unitária é sempre uma lista ordenada). Em seguida, devemos inserir o primeiro elemento da lista não ordenada (segundo elemento da lista na primeira iteração) na posição correta do segmento já ordenado da sequência. O processo é repetido até que não tenhamos mais elementos na lista não ordenada.

**Vamos analisar a execução do algoritmo através de um exemplo. Seja a sequência: 16, 8, 20, 18, 9, 2.** Antes da primeira iteração, consideramos que o primeiro elemento “16” é a sequência ordenada e os elementos “8, 20, 18, 9, 2” os não ordenados.

## Etapa 1

Na primeira iteração, analisamos o primeiro elemento da sequência não ordenada colocando-o na sua posição correta. Assim, passamos a ter “8, 16”, que é a sequência ordenada, e “20, 18, 9, 2”, que é a sequência não ordenada.

## Etapa 2

Na segunda iteração, vamos colocar o elemento “20” na posição correta que, por coincidência, já está posicionado. Sendo assim, passamos a ter “8, 16, 20” como sequência ordenada e “18, 9, 2” como não ordenada.

## Etapa 3

Na terceira iteração, analisamos a chave “18”, colocando-a na sua posição correta na sequência ordenada “8, 16, 18, 20”. Restando “9, 2” como não ordenada.

## Etapa 4

Na quarta iteração, analisamos "9", obtendo "8, 9, 16, 18, 20" e "2".

Na última iteração, a chave "2" é colocada em sua posição, terminando a execução.

A complexidade deste algoritmo também é  $O(n^2)$ . Assim como o método da bolha, o algoritmo tem o pior caso bem formado, a saber, as instâncias ordenadas em ordem reversa da desejada. Seja a sequência  $S' = s_1, s_2, s_3, \dots, s_n$  onde  $s_1 < s_2 < s_3 < \dots < s_n$ . Supondo que inicialmente apresentamos ao algoritmo a sequência ordenada em ordem reversa, isto é,  $S' = s_n, s_{n-1}, \dots, s_3, s_2, s_1$ , teremos a seguinte execução:

### Início:

$s_n, s_{n-1}, \dots, s_3, s_2, s_1$	início
$s_{n-1}, s_n, \dots, s_3, s_2, s_1$	1 troca 1ª iteração
$s_{n-2}, s_{n-1}, s_n, \dots, s_3, s_2, s_1$	2 trocas 2ª iteração
...	
$s_3, s_4, \dots, s_n, s_2, s_1$	n-3 trocas (n-3)ª iteração
$s_2, s_3, \dots, s_n, s_1$	n-2 trocas 3ª (n-2)ª iteração
$s_1, s_2, \dots, s_{n-1}, s_n$	n-1 trocas 3ª (n-1)ª iteração - fim, vetor ordenado!

Somando-se a quantidade de comparações e trocas:  $1 + 2 + 3 + \dots + n-1 = n(n-1)/2$  que é  $O(n^2)$ .

Assim como os outros, o algoritmo é estável, não recursivo e com complexidade de espaço de  $O(n)$ . O programa em linguagem Python que implementa o algoritmo é apresentado no programa a seguir.

#### Insertion Sort

 TUTORIAL  COPIAR

Python3

```
1 def insercao(vetor):
2     for i in range(1, len(vetor)):
3         valor_atual = vetor[i] # guardar o valor da posição 'i'
4         j = i - 1 # guardar a posição anterior à posição 'i'
5         while j >= 0 and valor_atual < vetor[j]:
6             vetor[j+1] = vetor[j]
```

null

null



A linguagem Python conta com muitas funções que fazem parte de sua biblioteca nativa. Elas são muito úteis para o desenvolvimento de algoritmos com menos linhas de códigos e mais característicos da linguagem. Segue abaixo um exemplo de algumas dessas funções, as quais facilitariam a codificação de alguns dos algoritmos de ordenação estudados. Execute o código presente no emulador abaixo e veja por si mesmo!

#### Facilidades da linguagem

 TUTORIAL  COPIAR

Python3

```
1 # demonstrando troca de posição entre 2 elementos
2 # de uma lista, array etc.
```



```
3 i = 3 # índice on ocorrerá a troca
4 vetor = [1, 2, 4, 8, 16]
5 print("vetor antes da troca de posições:",vetor)
```

null

null



Que tal um pequeno desafio? Reescreva os algoritmos de ordenação vistos neste conteúdo, dessa vez fazendo uso das facilidades que a linguagem Python nos propicia, como foi demonstrado no Emulador anterior. Vamos lá!

Emulador de códigos - Linguagem Python

 TUTORIAL  COPIAR

Python3

1

null

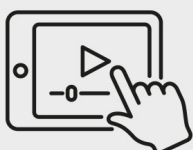
null



## Aplicando os algoritmos de ordenação primária em python

Para acompanhar o vídeo, você pode baixar o código fonte do programa sinalizado no Explore +.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



# Falta pouco para atingir seus objetivos.

## Vamos praticar alguns conceitos?

### Questão 1

Quando apresentamos uma sequência já ordenada para os algoritmos da bolha e para o Insert Sort, é correto afirmar que:

A

O método da bolha irá executar mais rápido, uma vez que, para sequências ordenadas, o método da bolha executa  $n-1$  comparações, enquanto o Insert Sort executa  $n(n-1)/2$  comparações.

B

O Insert Sort irá executar mais rápido, uma vez que, para sequências ordenadas, o método executa  $n-1$  comparações, enquanto o método da bolha sempre executa  $n(n-1)/2$  comparações.

C

Ambos executam sempre em  $O(n^2)$  operações.

D

Ambos executam sempre em tempo linear para as instâncias já ordenadas.

E

A sequência ordenada não é o melhor caso dos métodos analisados.

Parabéns! A alternativa D está correta.

O método da bolha e o Insert Sort executam em tempo linear para o melhor caso. Em ambos algoritmos, o melhor caso é a sequência ordenada.

### Questão 2

Comparando o Selection Sort com o Bubble Sort e o Insert Sort, é correto afirmar que:

A

Os três têm a mesma complexidade computacional.

B

Podemos afirmar que o Bubble Sort e o Insert Sort são mais eficientes que o Selection Sort.

C

Podemos afirmar que o Selection Sort e o Insert Sort são mais eficientes que o Bubble Sort.

D

Podemos afirmar que o Selection Sort e Bubble Sort são mais eficientes que o Insert Sort.

E

O Bubble Sort é o mais eficiente entre os três.

Parabéns! A alternativa A está correta.

Não há base, além da complexidade computacional, para afirmar que um algoritmo é melhor que outro. Como os três têm a mesma complexidade, são considerados equivalentes.

## Considerações finais

Neste tema, viajamos pelo mundo dos algoritmos de ordenação elementares. Inicialmente, estudamos o problema da ordenação, como analisar e classificar os algoritmos de ordenação e como as diversas métricas de classificação devem ser aplicadas para a escolha do algoritmo de ordenação.

Finalmente, estudamos os métodos de ordenação elementares empregando os algoritmos de ordenação da bolha, ordenação por seleção e por inserção, assim como a implementação desses algoritmos em linguagem Python.

No podcast a seguir, falaremos um pouco mais sobre os algoritmos de ordenação.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



## Referências

FRIEND, E. **Sorting on electronic computer systems**. J. ACM 3 (1956), 134–168. Consultado na internet em: 25 mar. 2022.

## Explore +

Para saber mais sobre os assuntos tratados neste tema, leia:

**A Survey, Discussion and Comparison of Sorting Algorithms**, de Ashok Kumar Karunanithi.