**FACULDADE DE COMPUTAÇÃO E INFORMÁTICA**

**CIÊNCIAS DA COMPUTAÇÃO E SISTEMAS DE INFORMAÇÃO**

**Linguagem de Programação I**

**AULA 04: ORDENAÇÃO EM LISTAS**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | Nossos objetivos nesta aula são:   * Conhecer o problema da ordenação em listas. * Implementar as estratégias de ordenação pelos métodos de seleção (*SelectionSort*) e Bolha (*BubbleSort*) | |  | As referências para esta aula são: | |  | O **Capítulo 11** do livro:  DIERBACH, C. *Introduction to Computer Science Using Python: A Computational Problem Solving Focus.* 1st Edition, New York: Wiley, 2012. | |  |  | | Macintosh HD:Users:lucianosilva:Desktop:6780034g1.jpg | O Capítulo 14, Seção 14.6 (Classificação) do livro:  HORSTMANN, C. Conceitos de Computação com Java. 5.ed. Porto Alegre: Bookman, 2009. | |

# PROBLEMA DE ORDENAÇÃO DOS ELEMENTOS DE UMA LISTA

O problema de ordenação de elementos de uma lista consiste em colocar os elementos de uma lista L em ordem crescente ou decrescente, conforme mostra o exemplo abaixo:

|  |  |
| --- | --- |
| Lista não ordenada | Lista ordenada |
|  |  |

# ORDENAÇÃO DOS ELEMENTOS DE UMA LISTA EM PYTHON

Na aula sobre LISTAS EM PYTHON já estudamos alguns recursos em Python que permitem realizar a ordenação dos elementos de uma lista.

Vimos que podemos colocar os elementos de uma lista **L** em ordem crescente utilizando o método **sort()**.

L.sort()

Exemplo:

lista = [25, 44, 99, 3, 4]

lista.sort()

print(lista)

A execução do trecho acima irá apresentar a lista em ordem crescente:

[3, 4, 25, 44, 99]

O método **reverse()** inverte a ordem atual dos elementos da lista.

L.reverse()

Caso seja necessário ordenar os elementos em ordem decrescente, podemos chamar o método **reverse()** logo após a execução do método **sort()**. Exemplo:

lista = [25, 44, 99, 3, 4]

lista.sort()

lista.reverse()

print(lista)

A execução do trecho acima irá apresentar a lista com os elementos em **ordem decrescente**.

[99, 44, 25, 4, 3]

# IMPLEMENTAÇÃO DE ALGORITMOS de ORDENAÇÃO

Apesar da linguagem Python já possuir o método de ordenação de elementos em uma lista, é necessário que um estudante da área de Computação entenda o funcionamento dos algoritmos de ordenação. Por este motivo, nesta aula você irá implementar os algoritmos de ordenação e não utilizará o método já existente em Python para efetuar esta operação.

Existem diversos algoritmos para se ordenar listas:

* ordenação por inserção (*InsertionSort*)
* ordenação por seleção (*SelectionSort*)
* ordenação pelo método da bolha (*BubbleSort*)
* ordenação por intercalação (*MergeSort*)
* ordenação por heap (*HeapSort*)
* ordenação rápida (*QuickSort*)

Neste primeiro contato com ordenação, veremos apenas os métodos *BubbleSort* e *SelectionSort*. Os outros métodos serão vistos posteriormente em outras disciplinas (como Estruturas de Dados).

# ORDENAÇÃO PELO MÉTODO DA BOLHA (BUBBLESORT)

O algoritmo de ordenação pelo método da bolha é um método baseado em trocas e a ideia é ir comparando pares consecutivos. O termo bolha vem do fato que os maiores números subam na lista como uma bolha para a superfície. Uma introdução lúdica a este algoritmo pode ser vista em:

* <https://www.youtube.com/watch?v=lyZQPjUT5B4>

Execute o passo a passo do algoritmo de ordenação **BubbleSort** para o vetor [50, 30, 40, 20, 10]

|  |  |  |
| --- | --- | --- |
| **COMPARAÇÃO** | **VETOR** | **TROCA /NÃO TROCA** |
| 50 > 30 | 50, 30, 40, 20, 10 | Troca |
|  | 30, 50, 40, 20, 10 |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

O algoritmo de ordenação pelo método da bolha é mostrado abaixo:

**Algoritmo** BubbleSort (L)

**Entrada**: Lista L com elementos comparáveis

**Saída**: Lista L ordenada em ordem crescente

1. **para** i=1 até len(L)-1 **faça**

2. **para** j=0 até len(L)-1-i **faça**

3. **se** L[j]>L[j+1] **então #** troca os elementos

4. aux = L[j]

5. L[j] = L[j+1]

6. L[j+1] = aux

7. **fimse**

8. **fimpara**

9. **fimpara**

# EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Implemente o algoritmo de ordenação **BubbleSort** na função abaixo:

def bubbleSort(L):

**Questão**: se o tamanho do vetor a ser ordenado for n, quantas comparações faz o algoritmo de ordenação Bubblesort no pior caso?

# ORDENAÇÃO POR SELEÇÃO (SELECTION SORT)

A ideia básica do algoritmo SelectionSort é varrer o vetor comparando todos os seus elementos com o primeiro elemento. Caso o primeiro elemento esteja ordenado em relação ao elemento que está sendo comparado, então é feita a troca. Ao chegar no final do vetor, teremos o menor valor (ou maior), conforme comparação na primeira posição do vetor.

O método SelectionSort é levemente mais eficiente que o método BubbleSort, ainda que se trate de um algoritmo apenas para estudo e ordenação de pequenos vetores.

Uma introdução lúdica a este algoritmo pode ser vista em:

* <https://www.youtube.com/watch?v=Ns4TPTC8whw>

Execute o passo a passo do algoritmo de ordenação **SelectionSort** para o vetor [50, 30, 40, 20, 10]

|  |  |  |
| --- | --- | --- |
| **COMPARAÇÃO** | **VETOR** | **TROCA /NÃO TROCA** |
| 50 > 30 | 50, 30, 40, 20, 10 | Troca |
|  | 30, 50, 40, 20, 10 |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

O algoritmo de ordenação por inserção pode ser escrito da seguinte forma:

**Algoritmo** SelectionSort (L)

**Entrada**: Uma lista L com elementos comparáveis

**Saída**: A lista L ordenada em ordem crescente

1. **para** i=0 até len(L) -1 **faça**

2. **para** j=i+1 até len(L) **faça**

3. # compara se o elemento é menor que o primeiro

4. se L[j] < L[i] então

5. aux <- L[j]

6. L[j] <- L[i]

7. L[i] <- aux

8. **fimSe**

9. **fimpara**

10.**fimpara**

# EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Implemente o algoritmo de ordenação por seleção na função abaixo:

def SelectionSort(L):

**Questão:** se o tamanho do vetor a ser ordenado for n, quantas comparações faz o algoritmo de ordenação por inserção ?

**EXERCÍCIOS EXTRA-CLASSE**

1. Modifique o programa de **ordenação por seleção** para **contar quantas comparações** são feitas durante a ordenação. Teste a sua modificação e conte quantas comparações são feitas para vetores de tamanho 10, 50, 100, 500, 1000, 5000, 10000, 50000 e 100000 números. Gere um gráfico tamanho do vetor x número de comparações.
2. Modifique o programa **BubbleSort** para contar quantas comparações são feitas durante a ordenação. Teste a sua modificação e conte quantas comparações são feitas para vetores de tamanho 10, 50, 100, 500, 1000, 5000, 10000, 50000 e 100000 números. Gere um gráfico tamanho do vetor x número de comparações.
3. Compare os gráficos obtidos nos exercícios (1) e (2). Qual dos três algoritmos é o mais eficiente?