**FACULDADE DE COMPUTAÇÃO E INFORMÁTICA**

**CIÊNCIAS DA COMPUTAÇÃO E SISTEMAS DE INFORMAÇÃO**

**Linguagem de Programação I**

**AULA 07: ORDENAÇÃO EM LISTAS**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | Nossos objetivos nesta aula são:   * Conhecer o problema da ordenação em listas. * Implementar as estratégias de ordenação pelos métodos de inserção (*InsertionSort*), Bolha (*BubbleSort*) e Merge (*MergeSort*) | |  | 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 *InsertionSort* e *BubbleSort*. Os outros métodos serão vistos posteriormente e em outras disciplinas (como Estruturas de Dados, do terceiro semestre).

# ORDENAÇÃO POR INSERÇÃO (INSERTION SORT)

A ideia básica do algoritmo de ordenação por inserção é semelhante ao processo de ordenar as cartas de um baralho.

Considera-se que os dados estejam divididos em 2 segmentos: um já ordenado e outro a ser ordenado. Inicialmente o 1º segmento tem apenas 1 elemento, considerado já ordenado. O 2º. Segmento é formado pelos dados restantes. A cada iteração, um elemento do 2º. segmento é transferido para o 1º. segmento, sendo inserido em sua posição correta em relação àqueles que lá já se encontravam.

Uma introdução lúdica a este algoritmo pode ser vista em:

* https://www.youtube.com/watch?v=ROalU379l3U

A figura abaixo ilustra o funcionamento deste algoritmo:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 |  | 12 |  | 12 |  | 12 |  | 6 |
| 12 |  | 44 |  | 42 |  | 42 |  | 12 |
| 42 |  | 42 |  | 44 |  | 44 |  | 42 |
| 94 |  | 94 |  | 94 |  | 94 |  | 44 |
| 6 |  | 6 |  | 6 |  | 6 |  | 94 |

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

**Algoritmo** InsertionSort (L)

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

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

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

2. D = L[i]

3. j = i - 1

4. # encontra a posição certa do elemento D

5. **enquanto** j >=0 E L[j] > D **faça**

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

7. j = j - 1

8. **fimEnquanto**

9. L[j+1] = D #coloca D na posição correta

10.**fimpara**

# EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Implemente o algoritmo de ordenação por inserção na função abaixo:

def insertionSort(L):

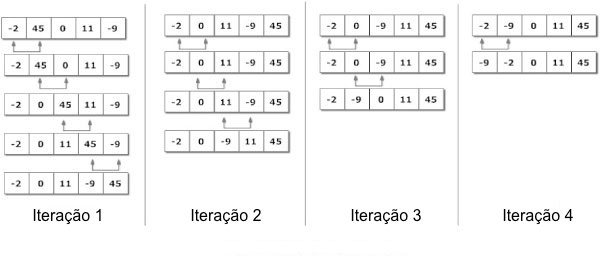
**Questão:** se o tamanho do vetor a ser ordenado for n, quantas comparações faz o algoritmo de ordenação por inserção ?

# 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 sobre para a superfície. Uma introdução lúdica a este algoritmo pode ser vista em:

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

A figura abaixo ilustra o funcionamento deste algoritmo:



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?

**ATIVIDADE DE LABORATÓRIO**

**(1)** O Método de Ordenação por Intercalação **(Mergesort)** é um outro algoritmo clássico de ordenação de listas. A ideia básica deste algoritmo é ir subdividindo a lista até que se encontre um subvetor que saibamos ordenar facilmente (por exemplo, uma lsita com apenas um elemento). Uma vez que os sublistas estejam ordenadas, o algoritmo faz uso de uma rotina de intercalação (merge) para juntá-los. Para uma introdução lúdica ao algoritmo MergeSort, veja o vídeo abaixo:

https://www.youtube.com/watch?v=XaqR3G\_NVoo

Nosso objetivo nesta atividade será implementar o MergeSort, utilizando os passos abaixo.

1. Implemente um procedimento chamado **merge**, que recebe dois trechos ordenados de um vetor e produz a intercalação ordenada destes dois trechos, usando a descrição abaixo:

**Algoritmo** merge (V,L,R)

**Entrada**: A lista L e as sublistas L e R

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

1. nL=len(L)
2. nR=len(R)
3. i=0
4. j=0
5. k=0
6. **enquanto** (i<nL e j<nR) **faça**
7. **se** (L[i]<=R[j])
8. **então** V[k]=L[i]
9. i=i+1
10. **senão** V[k]=R[j]
11. j=j+1
12. **fimse**
13. k=k+1
14. **fimenquanto**
15. **enquanto** (i<nL) **faça**
16. V[k]=L[i]
17. i=i+1
18. k=k+1
19. **fimenquanto**
20. **enquanto** (j<nR) **faça**
21. V[k]=R[j]
22. j=j+1
23. k=k+1
24. **fimequanto**
25. Implemente o Algoritmo Mergesort descrito abaixo:

**Algoritmo MergeSort (V)**

**Entrada**: Uma lista V

**Saída**: A lista L ordenada segundo a estratégia de ordenação por intercalação

1. n=len(V)
2. Se n>1 **então**
3. **meio=n//2**
4. **para i de 0 até meio-1** **faça**
5. left[i]=V[i]
6. **fimpara**
7. **para i de meio até n-1** **faça**
8. right[i-meio]=V[i]
9. **fimpara**
10. mergeSort(left)
11. mergeSort(right)
12. merge(V,left, right)
13. **Problema: PISCINA E BALDES**

Andrea, Carlos e Marcelo são muito amigos e passam todos os finais de semana à beira da piscina. Enquanto Andrea se bronzeia ao sol, os dois ficam jogando Bolhas. Andrea, uma desenvolvedora de jogos muito esperta, já disse a eles que não entende por que passam tanto tempo jogando um jogo tão primário.

Usando o computador portátil dela, os dois informam um inteiro N e uma sequência aleatória de inteiros de tamanho N é gerada.

O jogo então começa, cada jogador faz um movimento, e a jogada passa para o outro jogador. Marcelo é sempre o primeiro a começar a jogar. Um movimento de um jogador consiste na escolha de um par de elementos **consecutivos** da sequência que estejam fora de ordem e em inverter a ordem dos dois elementos. Por exemplo, dada a sequência gerada aleatoriamente 4, 4, 7, 3, 8, 5, o jogador pode inverter as posições de **7 e 3** ou de **8 e 5**, mas não pode inverter as posições de 3 e 8, nem de 4 e 7, também não pode inverter 4 e 4 pois são iguais. Continuando com o exemplo, se o jogador decide inverter as posições de 7 e 3 então a nova sequência será 4, 4, 3, 7, 8, 5.

Mais cedo ou mais tarde, a sequência ficará ordenada crescente. Perde o jogador impossibilitado de fazer um movimento. Andrea, com algum desdém, sempre diz que seria mais simples jogar cara ou coroa, com o mesmo efeito. Sua missão é ler a partir do teclado um inteiro N indicando o tamanho da sequência aleatória, gerar a sequência aleatória e determinar quem ganha o jogo, dada a sequência.

**Entrada**

A entrada contém vários casos de teste. Os dados de cada caso de teste estão numa única linha, e são inteiros separados por um espaço em branco. Cada linha contém um inteiro N seguido da sequência (X1 , X2 , . . . , XN )de N inteiros distintos dois a dois, veja o exemplo de entrada:

5 1 5 3 4 2

5 5 1 3 4 2

5 1 2 3 4 5

6 3 5 2 1 4 6

5 5 4 3 2 1

6 6 5 4 3 2 1

**Saída**

Para cada caso de teste da entrada seu programa deve imprimir uma única linha, com o nome do vencedor, igual a Carlos ou a Marcelo, sem espaços em branco. Veja o exemplo de saída:

Marcelo

Carlos

Carlos

Carlos

Carlos

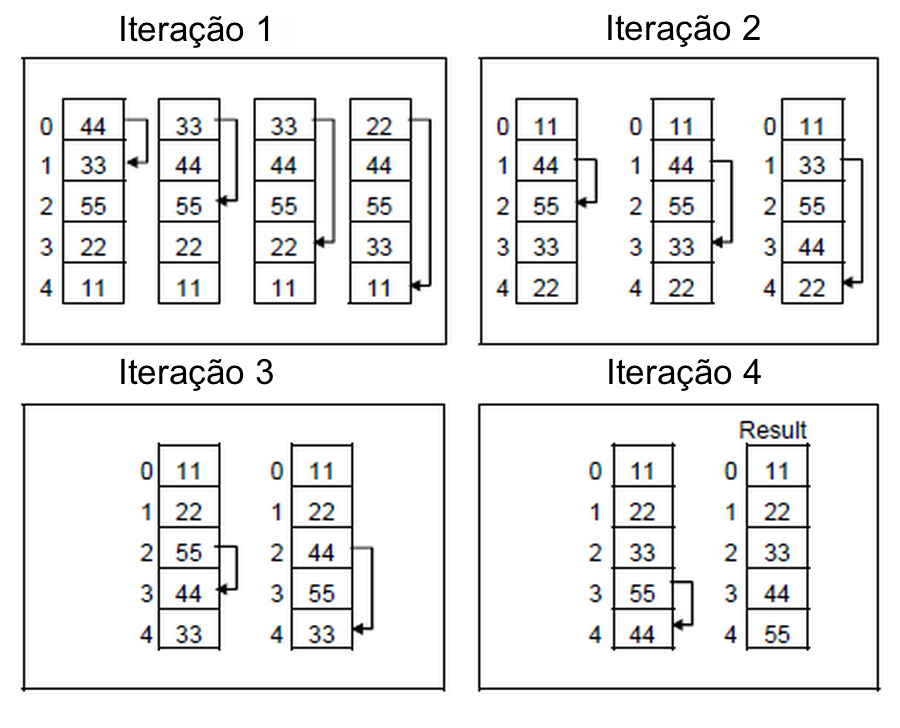
Marcelo

**EXERCÍCIOS EXTRA-CLASSE**

1. Modifique o programa de **ordenação por inserção** 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. Modifique o programa **MergeSort** 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.
4. Compare os gráficos obtidos nos exercícios (1),(2) e (3). Qual dos três algoritmos é o mais eficiente ?
5. A ideia básica do algoritmo de ordenação por seleção é, a cada iteração, selecionar o menor elemento do trecho do vetor que está sendo ordenado, supondo-se que queiramos ordená-lo em ordem crescente. Uma introdução lúdica a este algoritmo pode ser vista em:

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

A figura abaixo ilustra o funcionamento deste algoritmo:



O algoritmo de ordenação por seleção 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** // iteração  2. **para** j=i+1 até len(L)-1 **faça**// encontra o menor elemento do trecho a ser ordenado  3. **se** L[j]<L[i] // troca os elementos  4. **então** aux=L[i]  5. L[i]=L[j]  6. L[j]=aux  7. **fimse**  8. **fimpara**  9. **fimpara** |

Implemente a versão iterativa e uma versão recursiva do selectionSort e escreva uma função para testar as funções.