
FIAP

PYTHON – AULA 6
LÓGICA DE PROGRAMAÇÃO APLICADA

AGENDA

Aula 1	Instalando e conhecendo o Python e as ferramentas do curso. Revisitando lógica de programação em Python.
Aula 2	Revisitando lógica de programação em Python – continuação.
Aula 3	Trabalhando com listas, tuplas e dicionários. Leitura e escrita de JSON.
Aula 4	Leitura e escrita de arquivos. Introdução a Numpy e Pandas.

AGENDA

Aula 5	Orientação a objeto no Python.
Aula 6	Trabalhando com os algoritmos. Ordenação e Recursão. – Pesquisa em largura.
Aula 7	Algoritmo de Dijkstra.
Aula 8	Algoritmos Genéticos.

PROGRAMANDO EM PYTHON

TRABALHANDO COM OS ALGORITMOS.

ORDENAÇÃO, RECURSÃO E PESQUISA

ALGORITMOS

“Conjunto de regras e operações bem-definidas e ordenadas, destinadas à solução de um problema ou de uma classe de problemas, em um número finito de etapas.”

Os algortimos não se restringem à informática; são de uso geral. Veja o seguinte exemplo de um algoritmo não computacional “Receita de sorvete de chocolate”

INGREDIENTES:

- 1 tablete de chocolate meio amargo
- 1 lata de leite condensado
- 1 lata de leite
- Raspas de chocolate ou granulado

MODO DE PREPARO:

INÍCIO

1. Ponha o chocolate em uma tigela
2. Deixe a tigela no micro-ondas durante um minuto
3. Tire o chocolate do forno e mexa até esfriar
4. Bata no liquidificador com o leite e o leite condensado
5. Despeje tudo em uma forma e espere congelar por 3 horas
6. Distribua o sorvete em taças
7. Decore com raspas de chocolate ou granulado
8. Sirva

FIM

ALGORITMOS

- Existe um algoritmo embutido nas nossas tarefas diárias, independente de ela ser relacionada a programas de computador.

- Entretanto, existem algoritmos mais complexos que precisam ser aprendidos, tais como aqueles ligados à um programa eficiente de pesquisa em banco de dados, por exemplo.

- Algoritmos complexos são essenciais para levar sua lógica de programação à um próximo nível.

Métodos de Ordenação / Classificação

Existem 4 métodos gerais de ordenação:

- ✓ Inserção:
  - ✓ Inserção direta
  - ✓ Incremento decrescente (Shell Sort)
- ✓ Troca:
  - ✓ Método da bolha (Bubble Sort)
  - ✓ Método da troca e partição (Quick Sort)
- ✓ Seleção
  - ✓ Seleção direta (Selection Sort)
  - ✓ Seleção em árvore (Heap Sort)
- ✓ Intercalação
  - ✓ Merge Sort

Inserção direta:

- Dado um vetor para ordenação, percorre elemento por elemento deslocando-o e inserindo-o na posição ordenada.
- A ideia é formar um bloco de valores ordenados e outro de desordenados e ir passando os valores de um bloco a outro.
- Usado para conjunto pequeno de dados.
- Baixa eficiência

Inserção direta:

Vetor original	18	15	7	9	23	16	14
Divisão original	18	15	7	9	23	16	14
	Ordenados	**Desordenado**					
1ª iteração	15	18	7	9	23	16	14
2ª iteração	7	15	18	9	23	16	14
3ª iteração	7	9	15	18	23	16	14
4ª iteração	7	9	15	18	23	16	14
5ª iteração	7	9	15	16	18	23	14
6ª iteração	**7**	**9**	**14**	**15**	**16**	**18**	**23**

- Primeiro elemento do vetor é considerado ordenado e os demais desordenados;
- Busca-se os elementos do bloco dos desordenados e compara-se com os do bloco de ordenados;
- Faz-se a inserção na posição correta; Repete-se o processo até que os elementos do bloco de desordenados tenham passado para o bloco de ordenados.

Implementação Inserção direta:

Este algoritmo é um dos mais simples para ser implementado. A função recebe uma lista e faz a ordenação.

```
import random

def insercao_direta(vetor):
    n = len(vetor)
    for j in range(1, n):
        temp = vetor[j]
        i = j - 1
        while i >= 0 and vetor[i] > temp:
            vetor[i + 1] = vetor[i]
            i = i - 1
        vetor[i + 1] = temp
```

Implementação Inserção direta:

Para testar criamos uma lista com 20 valores aleatórios entre 1 e 100

```
lista = random.sample(range(1, 100), 20)

print(“Lista não ordenada: ", lista)
insercao_direta(lista)
print(“Lista ordenada:", lista)
```

Shell Sort:

- Extensão do algoritmo de inserção direta
- Diferença: o vetor usado no processo de classificação é dividido em vários segmentos (blocos).
- Faz classificações parciais do vetor, aumentando o desempenho nos passos seguintes.

Shell Sort:

Passo 1

1	2	3	4	5	6	7	8	9	10	11	12
17	29	42	15	21	22	47	37	52	43	27	12

1	5	9
17	21	52

2	6	10
29	22	43

3	7	11
42	47	27

4	8	12
15	37	12

Aplicando-se a inserção direta em cada segmento

17	21	52

29	22	43

42	47	27

15	37	12

Obtém-se o vetor

17	22	27	12	21	29	42	15	52	43	47	37

Shell Sort:

Passo 2

17	22	27	12	21	29	42	15	52	43	47	37

1	3	5	7	9	11
17	27	21	42	52	47

2	4	6	8	10	12
22	12	29	15	43	37

Aplicando-se a inserção direta em cada segmento

17	21	27	42	47	52

12	15	22	29	37	43

Obtém-se o vetor

17	12	21	15	27	22	42	29	47	37	52	43

Shell Sort:

Passo Final

Aplicando-se a inserção direta no vetor anterior, obtém-se o vetor:

Implementação Shell Sort:

Para a implementação do shell sort temos uma função que recebe uma lista não ordenada e seu tamanho.

```
import random

def shell_sort(vetor, n):
    intervalo = n // 2
    while intervalo > 0:
        for i in range(intervalo, n):
            temp = vetor[i]
            j = i
            while j >= intervalo and vetor[j - intervalo] > temp:
                vetor[j] = vetor[j - intervalo]
                j -= intervalo
            vetor[j] = temp
        intervalo = intervalo // 2
```

Implementação Shell Sort:

Para testar criamos uma lista com 20 valores aleatórios entre 1 e 100

```
lista = random.sample(range(1, 100), 20)

print("Lista não ordenada: ", lista)
shell_sort(lista, len(lista))
print("Lista ordenada:", lista)
```

Método Bolha (Bubble Sort):

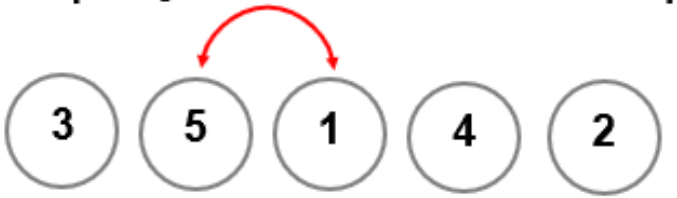
- Simples e fácil implementação.
- Ordenação por troca de valores entre posições consecutivas, fazendo com que os valores mais altos (ou mais baixos) “borbulhem” para o final do arranjo.
- Ele envolve repetidas comparações.
- Seu desempenho é muito ruim.

Método Bolha (Bubble Sort):

Ordenar o arranjo em ordem crescente.

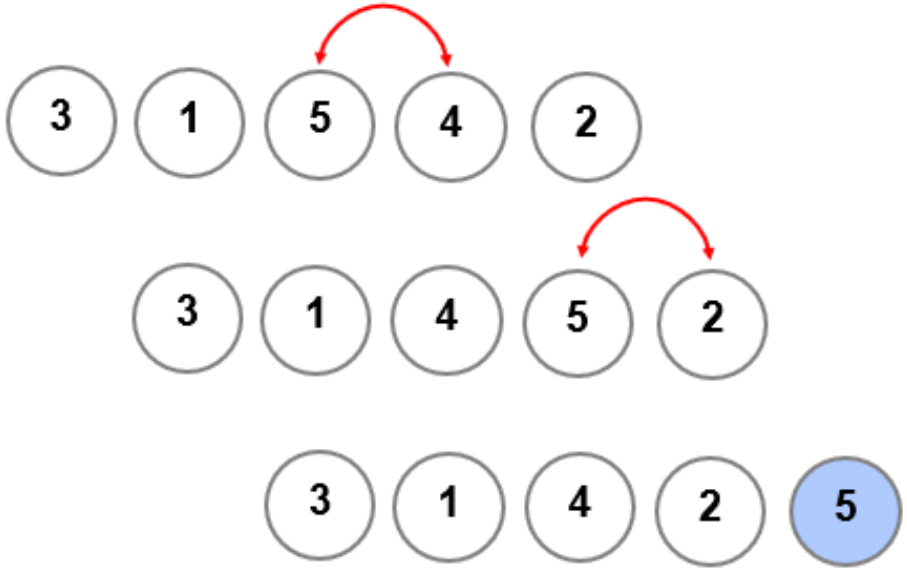
1º Passo: Compara os elementos das primeiras posições.

2º Passo: Ordena-se os elementos e continuamos com as comparações dos elementos subsequentes.



Método Bolha (Bubble Sort):

Próximos Passos:

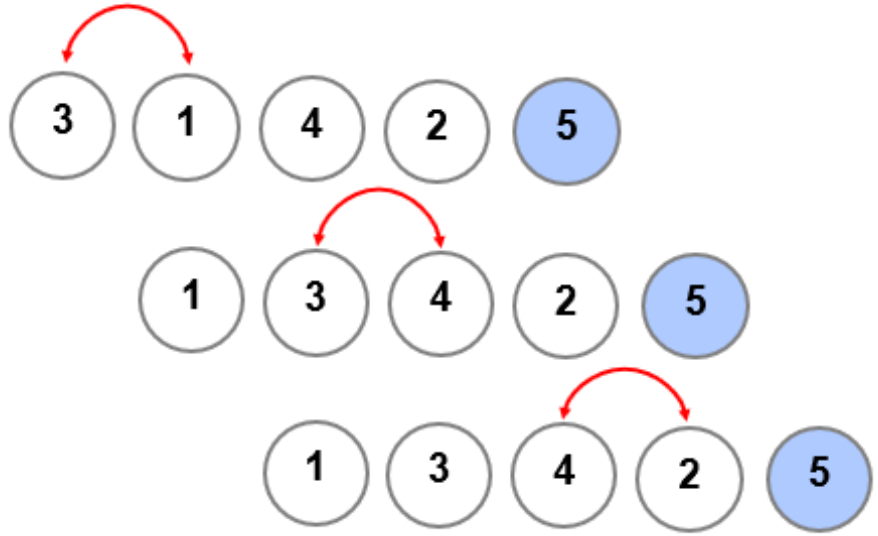


O elemento 5 “borbulhou” para sua posição correta.

Método Bolha (Bubble Sort):

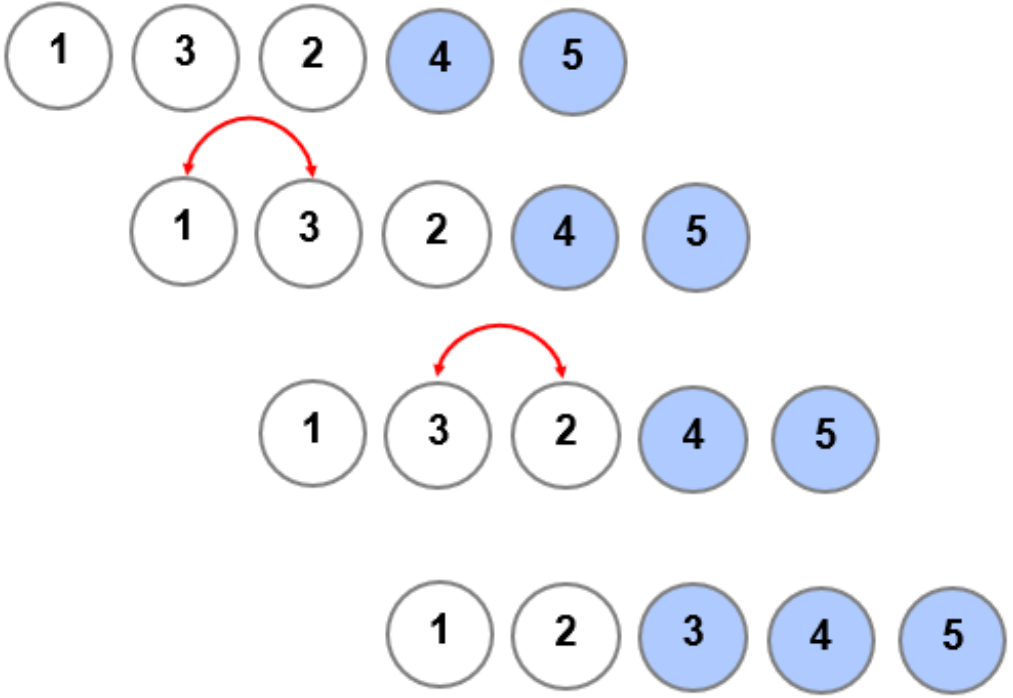
Próximos Passos:

Repete o processo de comparações e trocas desde o início, só que dessa vez não precisará comparar o penúltimo com o último elemento, pois o número “5”, está em sua posição correta no arranjo.

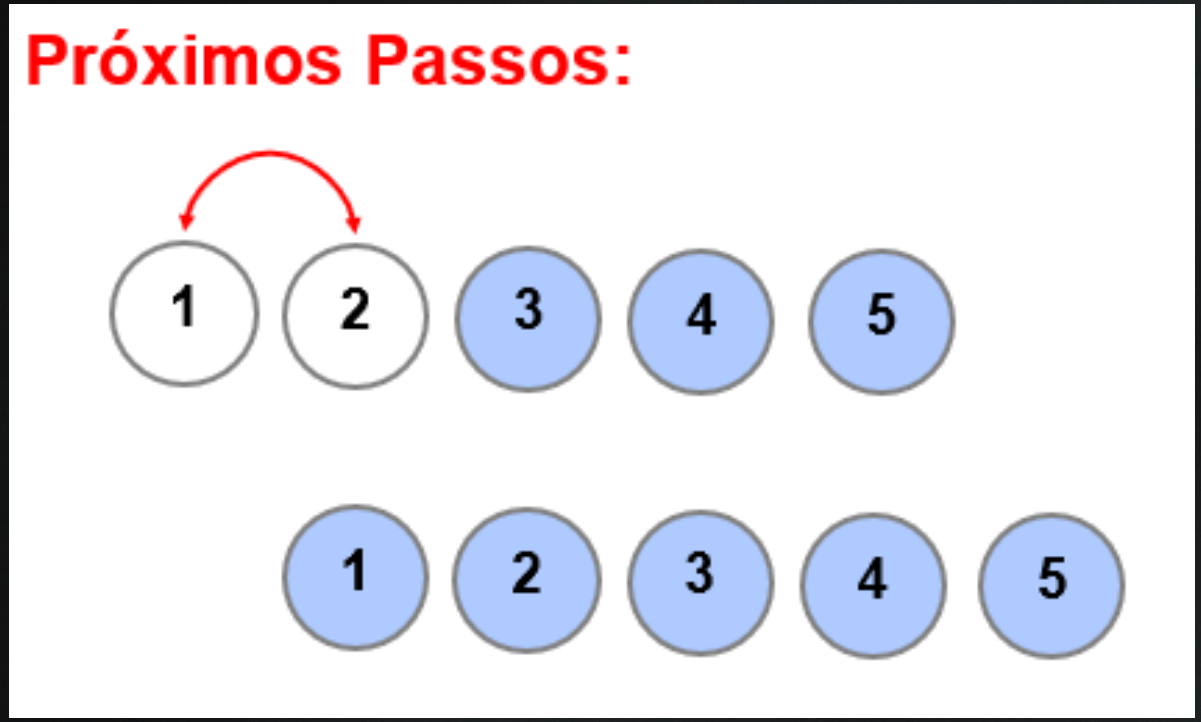


Método Bolha (Bubble Sort):

Próximos Passos:



Método Bolha (Bubble Sort):



Implementação Bubble Sort:

Para o Bubble Sort temos apenas uma função que recebe uma lista não ordenada e faz a ordenação

```
import random

def bubble_sort(vetor):

    for i in range(len(vetor), 0, -1):
        troca = False
        for j in range(0, i - 1):
            if vetor[j] > vetor[j + 1]:
                vetor[j + 1], vetor[j] = vetor[j], vetor[j + 1]
                troca = True
        if not troca:
            break
```

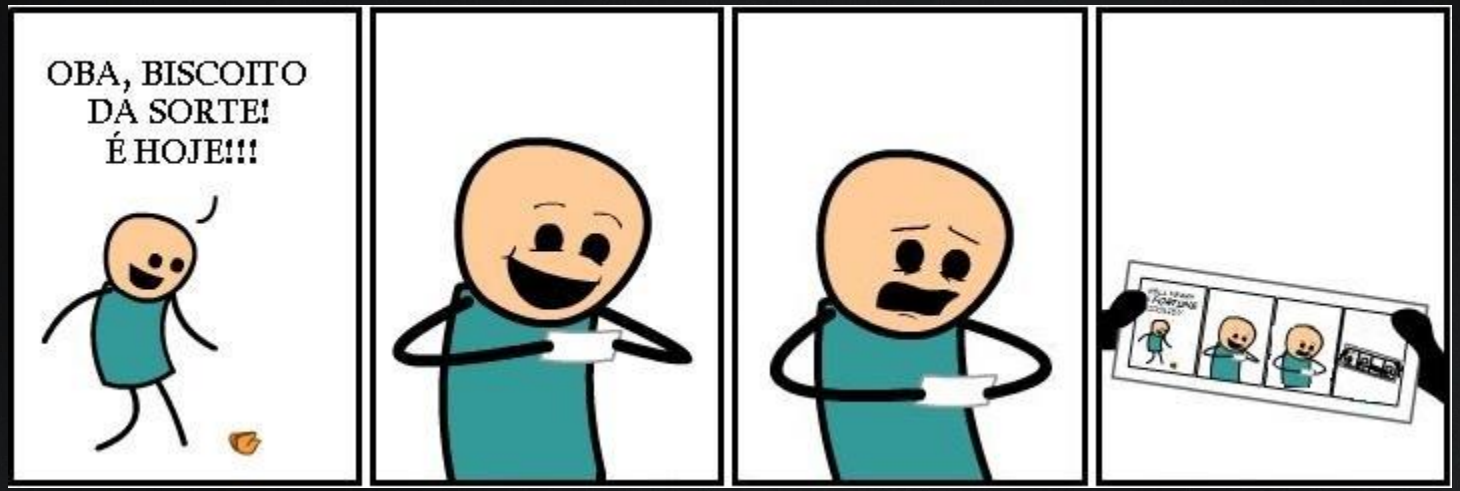
Implementação Bubble Sort:

Para testar criamos uma lista com 20 valores aleatórios entre 1 e 100

```
lista = random.sample(range(1, 100), 20)

print("Lista não ordenada: ", lista)
bubble_sort(lista)
print("Lista ordenada:", lista)
```

Recursividade

- *É uma forma de resolver problemas dividindo-o em partes menores.*
- *Quando uma função chama ela mesma para resolver um problema.*



https://geraldoferraz.blogspot.com/2011/11/recursividade.html

Exemplo: função para somar uma sequência numérica

```python
def soma1(n):
    soma = 0
    for i in range(n+1):
        soma += i
    return soma

```

Com FOR

```python
#recursão
def soma2(n):
    if n == 0:
        return 0
    return n + soma2(n-1)
```

Com recursividade

Exercícios

1) *Desenvolva uma função para calcular a série Fibonacci de uma quantidade de termos fornecidos pelo usuário.*

2) *Desenvolva uma função utilizando a recursividade para calcular o valor fatorial de um numero inteiro.*

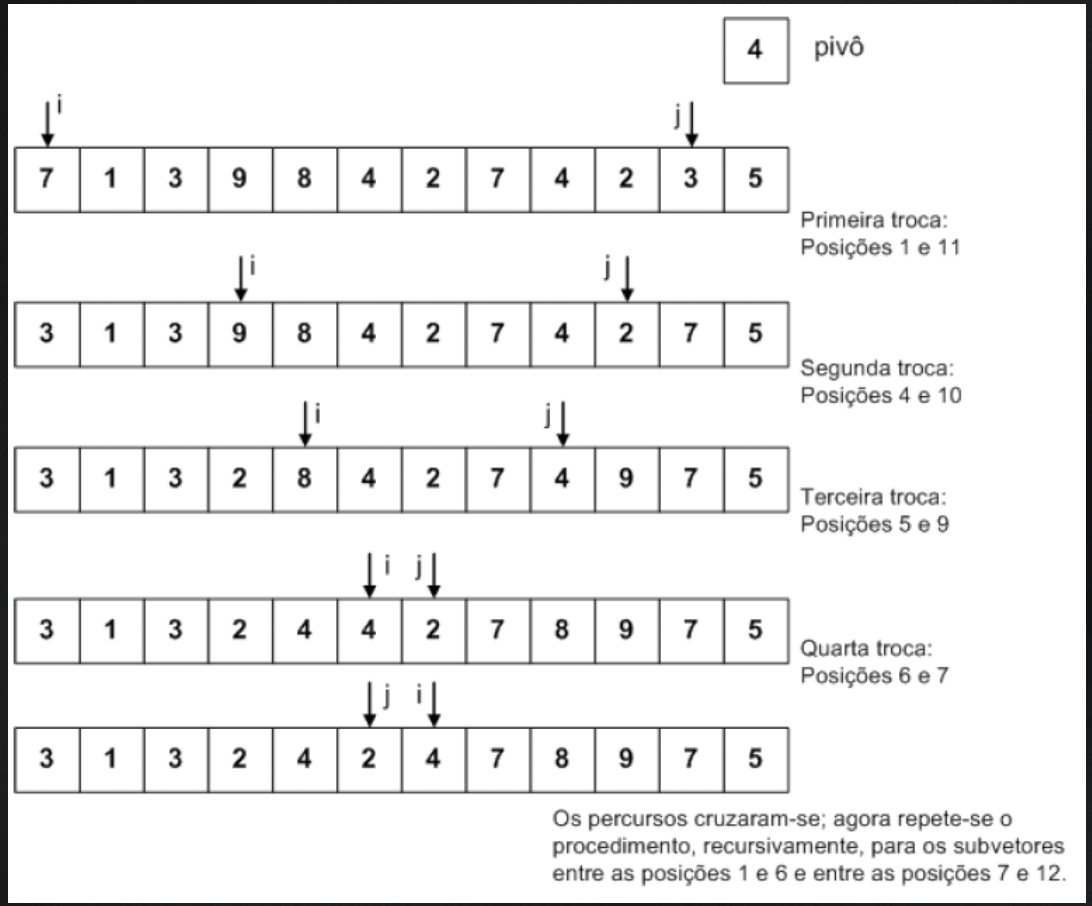
3) *Desenvolva uma função que recebe um número e verifica se esse valor é igual a 1. Se for, deve retornar 1, afinal: soma(1) = 1. Entretanto, se não for 1, deve retornar: n + soma(n-1)*

Método Quick Sort:

- É baseado em uma estratégia de dividir para conquistar
- É um dos algoritmos de ordenação mais populares.
- Seu desempenho é melhor na maioria das vezes.

Método Quick Sort:

- Baseia-se na divisão de um vetor em dois subvetores, de tal forma que todos os elementos do vetor 1 sejam < = a todos os elementos do vetor 2.
- Estabelecida a divisão, o problema estará resolvido, pois aplica-se recursivamente a mesma técnica a cada um dos subvetores, no final o vetor estará ordenado ao se obter um subvetor de apenas 1 elemento.



Extraído de Marcos Laureano

Implementação Quick Sort:

- Para este algoritmo, temos uma função quick sort que recebe a lista para ser ordenada e outras duas funções para dividir e trocar posições do vetor

```
import random

def quick_sort(vetor, inicial, final):
  if inicial < final:
      resultado_divisao = dividir(vetor, inicial, final)
      quick_sort(vetor, inicial, resultado_divisao - 1)
      quick_sort(vetor, resultado_divisao + 1, final)
```

```
def dividir(vetor, inicial, final):
    x = vetor[inicial]
    i = inicial
    j = inicial + 1
    while(j<=final):
        if vetor[j] < x:
            i += 1
            trocar(vetor, i, j)
        j += 1
    trocar(vetor, inicial, i)
    return i

def trocar(vetor, n, m):
    temp = vetor[n]
    vetor[n] = vetor[m]
    vetor[m] = temp
```

Implementação Quick Sort:

Para testar criamos uma lista com 20 valores aleatórios entre 1 e 100

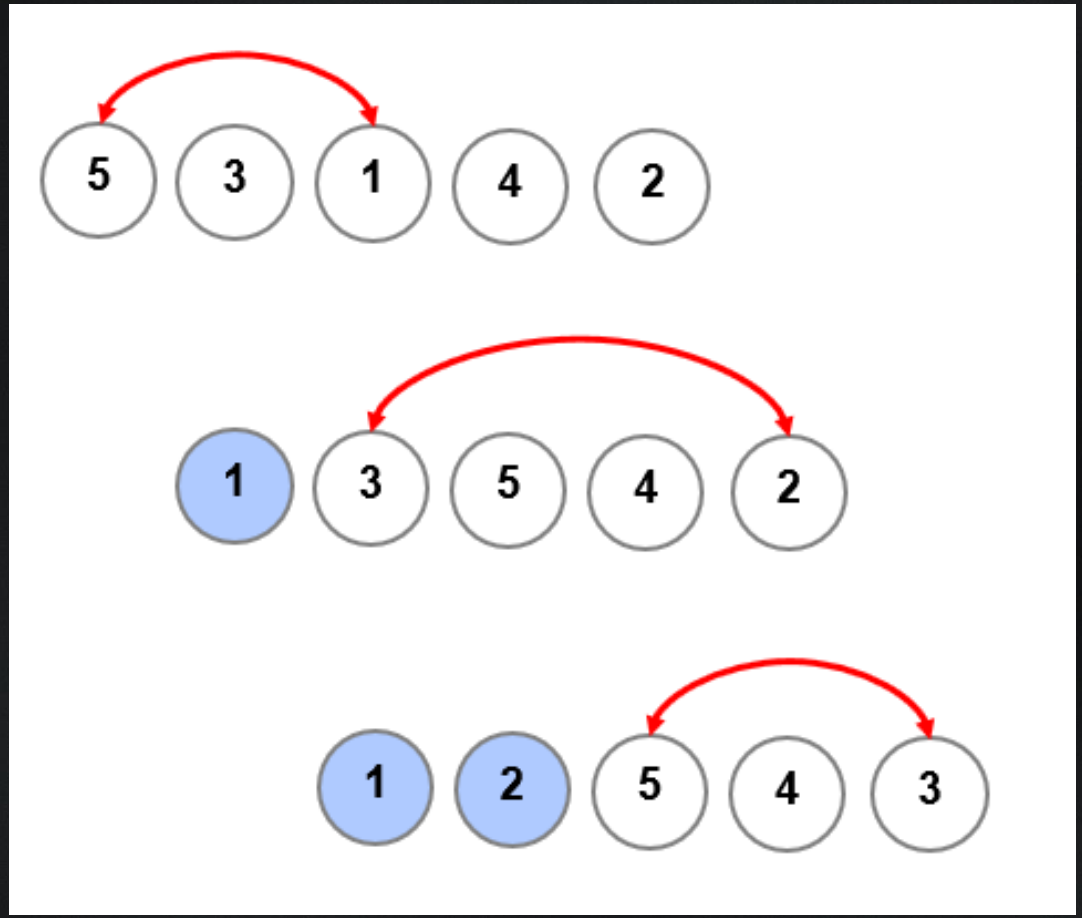
```
lista = random.sample(range(1, 100), 20)

print(“Lista não ordenada: ", lista)
quick_sort(lista, 0 , len(lista)-1)
print(“Lista ordenada:", lista)
```

Métodos de Seleção (Selection Sort)

- Selection Sort é um dos algoritmos de ordenação mais simples.
- Percorre o vetor ao longo das iterações e seleciona o menor elemento atual e o troca de lugar.

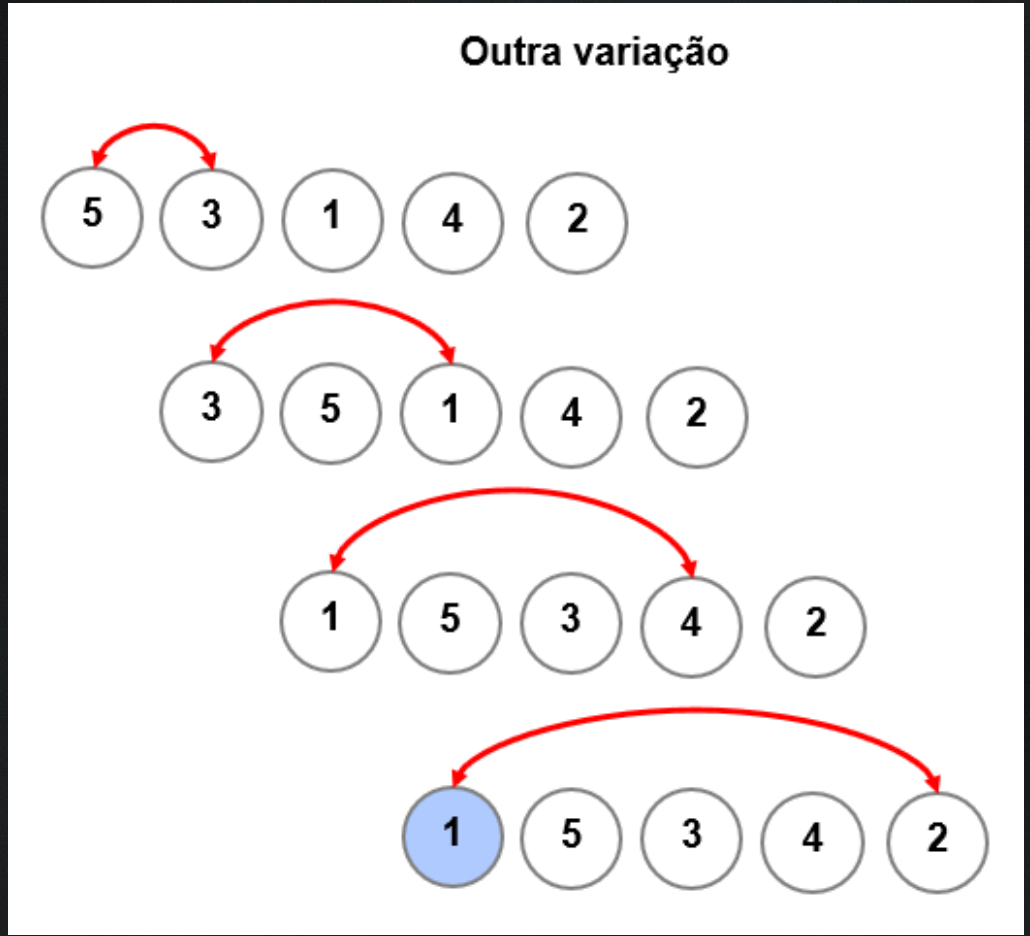
Selection Sort

- Consiste em encontrar a menor chave por pesquisa sequencial.
- Encontrando a menor chave, essa é permutada com a que ocupa a posição inicial do vetor, que fica então reduzido a um elemento.
- O processo é repetido para o restante do vetor, sucessivamente, até que todas as chaves tenham sido selecionadas e colocadas em suas posições definitivas.



Selection Sort

- Outra variação consiste em posicionar-se no primeiro elemento e ir testando-o com todos os outros (segundo)... (ultimo), trocando cada vez que for encontrado um elemento menor do que o que esta na primeira posição.
- Em seguida passa-se para a segunda posição do vetor repetindo novamente todo o processo.



Implementação Selection Sort:

```
import random

def selection_sort(vetor):
    if not vetor:
        return vetor
    for i in range(len(vetor)):
        min = i
        for j in range(i + 1, len(vetor)):
            if vetor[j] < vetor[min]:
                min = j
        vetor[i], vetor[min] = vetor[min], vetor[i]
```

Implementação Selection Sort:

Para testar criamos uma lista com 20 valores aleatórios entre 1 e 100

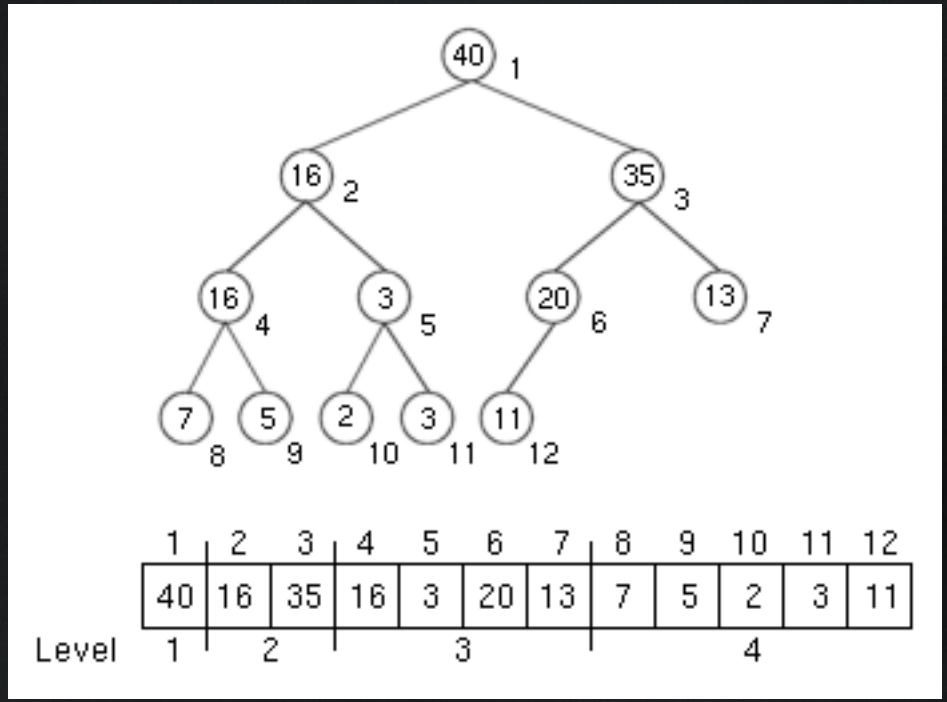
```
lista = random.sample(range(1, 100), 20)

print(“Lista não ordenada: ", lista)
selection_sort(lista)
print(“Lista ordenada:", lista)
```

Método Heap Sort:

- Foi desenvolvido com o objetivo de ordenar dados de uma ótima maneira, obtendo um consumo bastante reduzido de memória.
- É um método de ordenação cujo princípio de funcionamento é o mesmo princípio utilizado para a ordenação por seleção.
- Heap é o tipo de estrutura (árvore binária quase completa) e temos que saber qual estrutura estamos usando: Max-heap ou Min-heap.
- Uma propriedade sempre tem que ser mantida: NUNCA UM FILHO DEVE SER MAIOR QUE SEU PAI!

Heap Sort

- Quando a propriedade é ferida, deve-se haver uma troca;
- Se ocorrer e essa troca ferir a propriedade heap deve-se trocar novamente até que a propriedade seja estabelecida.
- Quando a propriedade for mantida, a raiz troca com a última posição da árvore e preenche a última posição do vetor que começa ser ordenado da direita para esquerda.
- Repete a operação até que o vetor seja ordenado.



Extraído de: https://commons.wikimedia.org/wiki/File:Heap_mat_entspriechendem_Tableau_dozou.png

Implementação do Heap Sort:

Temos uma função heap sort que recebe a lista para ser ordenada e uma função de apoio max heapify (empilhamento máximo)

```python
def heap_sort(vetor):
    n = len(vetor)
    for i in range(n, -1, -1):
        max_heapify(vetor, n, i)
    for i in range(n - 1, 0, -1):
        vetor[0], vetor[i] = vetor[i], vetor[0]
        max_heapify(vetor, i, 0)
```

```python
import random

def max_heapify(vetor, n, i):
    j = 2 * i + 1
    k = 2 * i + 2
    if j < n and vetor[j] > vetor[i]:
        maior = j
    else:
        maior = i
    if k < n and vetor[k] > vetor[maior]:
        maior = k
    if maior != i:
        vetor[i], vetor[maior] = vetor[maior], vetor[i]
        max_heapify(vetor, n, maior)
```

Implementação Heap Sort:

Para testar criamos uma lista com 20 valores aleatórios entre 1 e 100

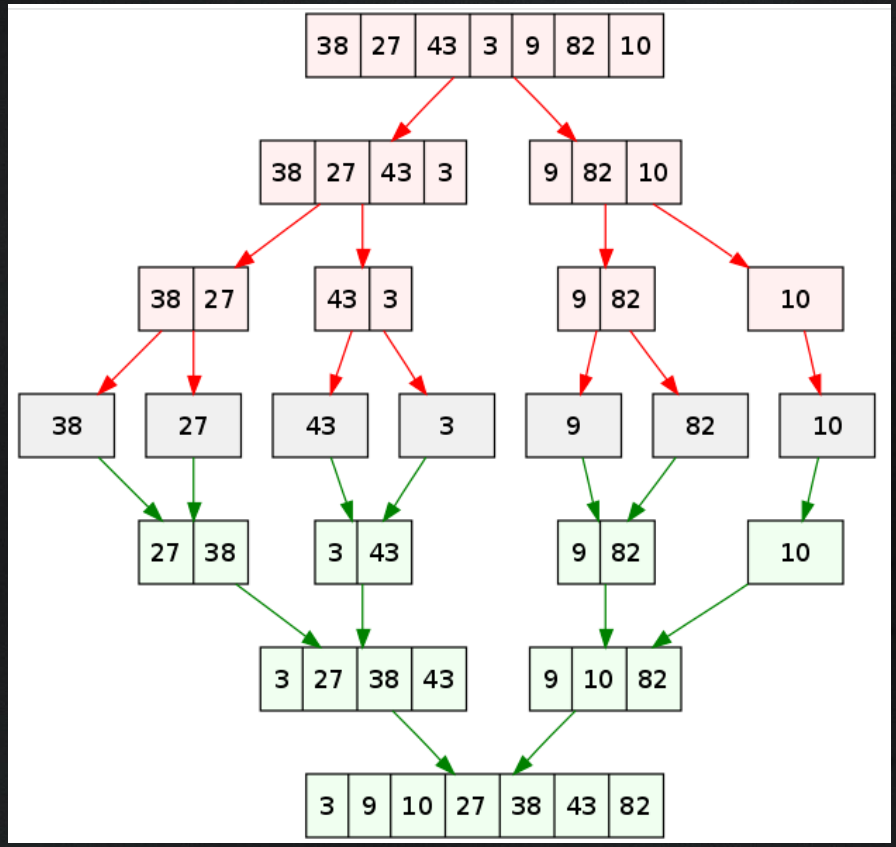
```
lista = random.sample(range(1, 100), 20)

print(“Lista não ordenada: ", lista)
heap_sort(lista)
print(“Lista ordenada:", lista)
```

Método Merge Sort:

- É um algoritmo que emprega a técnica de dividir e conquistar.
- Divide o vetor de entrada em duas metades e depois mescla as duas metades já ordenadas.
- A maior vantagem deste algoritmo diz respeito a complexidade temporal para ordenar um vetor inteiro, sendo muito melhor do que a complexidade temporal para execução do bubble sort e do insertion sort.

Merge Sort:

- *O processo de ordenação de um vetor pode ser resumido em três etapas:*
- *Dividir o vetor em duas metades.*
- *Ordenar a metade esquerda e a metade direita usando o mesmo algoritmo de forma recursiva.*
- *Mesclar as metades ordenadas*



https://commons.wikimedia.org/wiki/File:Merge_sort_algorithm_diagram.svg

Implementação do Merge Sort:

Criamos uma função merge_sort que vai receber a lista para ser ordenada.

```
def merge_sort(vetor, compare = lambda x, y: x <
y):
  if len(vetor) < 2:
    return vetor[:]
  else:
    meio = len(vetor) // 2
    esq = merge_sort(vetor[:meio], compare)
    dir = merge_sort(vetor[meio:], compare)
    return merge(esq, dir, compare)
```

Implementação do Merge Sort:

Criamos uma função merge que vai receber os dados pré-processados pela função merge_sort.

```
def merge(esq, dir, compare):
  resultado = []
  i = 0
  j = 0
  while (i < len(esq) and j < len(dir)):
    if compare(esq[i], dir[j]):
      resultado.append(esq[i])
      i += 1
    else:
      resultado.append(dir[j])
      j += 1
  while (i < len(esq)):
    resultado.append(esq[i])
    i += 1
  while (j < len(dir)):
    resultado.append(dir[j])
    j += 1
  return resultado
```

Pesquisa Binária

- É um algoritmo de busca que utiliza uma lista ordenada de elementos.
- Imagine que você precisa fazer uma busca em um dicionário e a palavra começa com L. Você pode começar na primeira página e percorrer o dicionário até encontrar a palavra ou pode começar pela metade.
- Quais das duas abordagens parece mais eficiente?

Pesquisa Binária:

- Imagine outra situação, na qual você deve adivinhar um número de 1 a 100, começando em 1.
- A cada tentativa, você é informado se chutou para cima, para baixo ou corretamente.
- Em cada tentativa você está eliminando apenas um número e isso se chama pesquisa simples.
- Na pesquisa binária comece pela metade (50) e será informado se o chute foi correto, alto ou baixo; se foi alto ou baixo, você eliminou praticamente a metade dos números.

Pesquisa Binária:

- A cada tentativa, faz-se novo corte pela metade.
- Em resumo, na pesquisa binária, você faz um chute intermediário e elimina a metade restante dos elementos de cada vez.
- Suponha a busca de um termo entre 10.000 termos distintos. Na pesquisa simples seriam necessárias 10.000 etapas na pior das hipóteses; na pesquisa binária seriam necessárias cerca de 13 etapas.
- Em uma lista de n números, a pesquisa binária precisa de $\log_2^n$ para devolver o valor correto.

Implementação da Pesquisa Binária

```python
def penquisa_binaria(lista, num):
    baixo = 0
    alto = len(lista)-1

    while baixo <= alto:
        meio = (baixo+alto) // 2
        chute = lista[meio]
        if chute == num:
            return meio
        if chute > num:
            alto = meio - 1
        else:
            baixo = meio + 1
    return None

lista = [1,3,5,7,9,11,13,15]

#retorna o índice do vetor
print(penquisa_binaria(lista, 5))
print(penquisa_binaria(lista, 15))
```

Notação Big O

- Essa notação informa o quão rápido é o algoritmo.
- É importante, pois o tempo de execução cresce a taxas diferentes.
- Conforme o número de itens aumenta, a pesquisa binária aumenta um pouco seu tempo de execução, já a pesquisa simples aumenta bastante.
- Portanto, não basta saber o quão rápido é o algoritmo, você precisa saber se o tempo de execução aumenta caso a lista aumente.

Notação Big O

- A notação Big O não fornece o tempo em segundos, mas em etapas, permitindo que você compare o número de operações e o quão rapidamente um algoritmo cresce.
- Veja a lista de alguns exemplos de tempo de execução Big O

Notação	Descrição
O(log *n*)	Tempo logarítmico. Pesquisa binária.
O(*n*)	Tempo linear. Pesquisa simples.
O(*n** log *n*)	Algoritmo de ordenação rápido. Quick sort
O(n^2)	Algoritmo de ordenação lento. Selection sort
O(*n*!)	Algoritmo muito lento. Caixeiro viajante

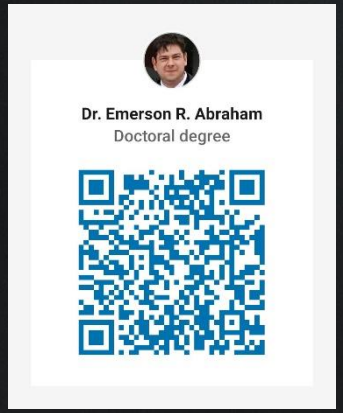
BIBLIOGRAFIA BÁSICA

- BEAZLEY, David. Python Essential Reference, 2009.
- BHARGAVA, ADITYA Y. Entendendo Algoritmos. Um guia ilustrado para programadores e outros curiosos. São Paulo: Ed. Novatec, 2017
- CORMEN, THOMAS H. et al. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, 2002
- COSTA, Sérgio Souza. Recursividade. Professor Adjunto da Universidade Federal do Maranhão.
- DOWNEY, ALLEN B. Pense em Python. Pense como um cientista da computação. São Paulo: Ed. Novatec, 2016
- GRANATYR, Jones; PACHOLOK, Edson. IA Expert Academy. Disponível em: https://iaexpert.academy/
- KOPEC, DAVID. Problemas clássicos de ciência da computação com Python. São Paulo: Ed. Novatec, 2019
- LINDEN, Ricardo. Algoritmos Genéticos. 3 edição. Rio de Janeiro: Editora Moderna, 2012
- MCKINNEY, WILLIAM WESLEY. Python para análise de dados. Tratamento de dados com Pandas, Numpy e Ipython. São Paulo: Ed. Novatec, 2018

BIBLIOGRAFIA BÁSICA

- TENEMBAUM, Aaron M. Estrutura de Dados Usando C. Sao Paulo: Makron Books do Brasil, 1995.
- VELLOSO, Paulo. Estruturas de Dados. Rio de Janeiro: Ed. Campus, 1991.
- VILLAS, Marcos V & Outros. Estruturas de Dados: Conceitos e Tecnicas de implementacao. RJ: Ed. Campus, 1993.
- PREISS, Bruno P. Estrutura de dados e algoritmos: Padrões de projetos orientados a objetos com Java. Rio de Janeiro: Editora Campus, 2001.
- PUGA, Sandra; RISSETTI, Gerson. Lógica de programação e estruturas de dados. 2016.
- SILVA, Osmar Quirino. Estrutura de Dados e Algoritmos Usando C. Fundamentos e Aplicações. Rio de Janeiro: Editora Ciência Moderna, 2007.
- ZIVIANI, N. Projeto de algoritmos com implementações em pascal e C. São Paulo: Editora Thomsom, 2002.

OBRIGADO



FIAP


FIAP