Algoritmos e Estruturas de Dados em Python

Technical Report · July 2021

DOI: 10.13140/RG.2.2.21210.26561/1

1 author:

Alexandre L M Levada
Universidade Federal de São Carlos
124 PUBLICATIONS **365** CITATIONS

Departamento de Computação

Centro de Ciências Exatas e Tecnologia

Universidade Federal de São Carlos

Algoritmos e Estruturas de Dados em Python

Complexidade e programação orientada a objetos

Prof. Alexandre Luis Magalhães Levada

Email: alexandre.levada@ufscar.br

Sumário

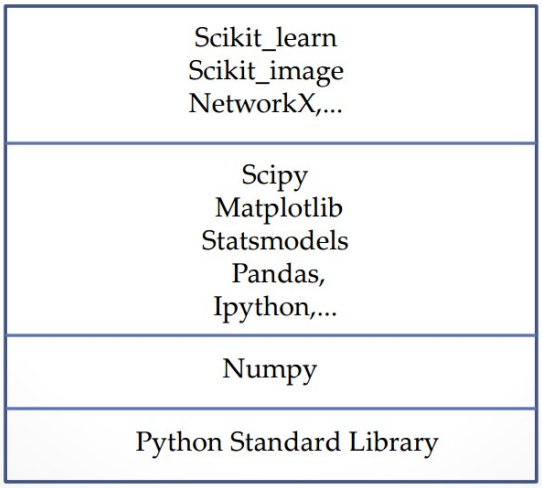
“Experiência não é o que acontece com um homem; é o que ele faz com o que lhe acontece”
(Aldous Huxley)

A linguagem Python

"Python é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. Foi lançada por Guido van Rossum em 1991.[1] Atualmente possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation. Apesar de várias partes da linguagem possuírem padrões e especificações formais, a linguagem como um todo não é formalmente especificada. A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código sobre a velocidade ou expressividade. Combina uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e frameworks desenvolvidos por terceiros. Python é uma linguagem de propósito geral de alto nível, multiparadigma, suporta o paradigma orientado a objetos, imperativo, funcional e procedural. Possui tipagem dinâmica e uma de suas principais características é permitir a fácil leitura do código e exigir poucas linhas de código se comparado ao mesmo programa em outras linguagens". (Wikipedia)

Nesse curso, optamos pela linguagem Python, principalmente pela questão didática, uma vez que a sua curva de aprendizado é bem mais suave do que linguagens de programação como C, C++ e Java. Existem basicamente duas versões de Pyton coexistindo atualmente. O Python 2 e o Python 3. Apesar de existirem poucas diferenças entre elas, é o suficiente para que um programa escrito em Python 2 possa não ser compreendido por interpretador do Python 3. Optamos aqui pelo Python 3, pois além de ser considerado uma evolução natural do Python 2, representa o futuro da linguagem.

Porque Python 3?
- Evolução do Python 2 (mais moderno)
- Sintaxe simples e de fácil aprendizagem
- Linguagem de propósito geral que mais cresce na atualidade
- Bibliotecas para programação científica e aprendizado de máquina



Dentre as principais vantagens de se aprender Python, podemos citar a enorme gama de bibliotecas existentes para a linguagem. Isso faz com que Python seja extremamente versátil. É possível desenvolver aplicações científicas como métodos matemáticos numéricos, processamento de sinais e imagens, aprendizado de máquina até aplicações mais comerciais, como sistemas web com acesso a bancos de dados.

Plataformas Python para desenvolvimento

Para desenvolver aplicações científicas em Python, é conveniente instalar um ambiente de programação em que as principais bibliotecas para computação científica estejam presentes. Isso poupa-nos muito tempo e esforço, pois além de não precisarmos procurar cada biblioteca individualmente, não precisamos saber a relação de dependência entre elas (quais devem ser instaladas primeiro e quais tem que ser instaladas posteriormente). Dentre as plataformas Python para computação científica, podemos citar as seguintes:

a) Anaconda - https://www.anaconda.com/products/individual
Uma ferramenta multiplataforma com versões para Windows, Linux e MacOS. Inclui mais de uma centena de pacotes para programação científica, o que o torna um ambiente completo para o desenvolvimento de aplicações em Python. Inclui diversos IDE's, como o idle, ipython e spyder.
b) WinPython - http://winpython.github.io/
Uma plataforma exclusiva para Windows que contém inúmeros pacotes indispensáveis, bem como um ambiente integrado de desenvolvimento muito poderoso (Spyder).
c) Pyzo (Python to people) - http://www.pyzo.org/
Um projeto que visa simplificar o acesso à plataforma Python para computação científica. Instala os pacotes standard, uma base pequena de bibliotecas e ferramentas de atualização, permitindo que novas bibliotecas sejam incluídas sob demanda.
d) Canopy - https://store.enthought.com/downloads
Mais uma opção multiplataforma para usuários Windows, Linux e MacOS.

Repl.it
Uma opção muito interessante é o interpretador Python na nuvem repl.it
Você pode desenvolver e armazenar seus códigos de maneira totalmente online sem a necessidade de instalar em sua máquina um ambiente de desenvolvimento local.

Nossa recomendação é a plataforma Anaconda, por ser disponível em todos os sistemas operacionais. Ao realizar o download, opte pleo Python 3, que atualmente deve estar na versão 3.7. Para a execução das atividades presentes nessa apostila, o editor IDLE é recomendado. Além de ser um ambiente extremamente simples e compacto, ele é muito leve, o que torna sua execução possível mesmo em máquinas com limitações de hardware, como pouca memória RAM e processador lento.

Após concluir a instalação, basta digitar anaconda na barra de busca do Windows. A opção Anaconda Prompt deve ser selecionada. Ela nos leva a um terminal onde ao digitar o comando idle e pressionarmos enter, seremos diretamente redirecionados ao ambiente IDLE. Um vídeo tutorial mostrando esse processo pode ser assistido no link a seguir:

https://www.youtube.com/watch?v=PWdrdWDmJIY&t=8s

Aula 1 – Programação estruturada em Python

Como forma de recapitular os fundamentos de programação em linguagem Python, essa aula apresenta dois modelos matemáticos para simulação de sistemas complexos: a recorrência logística e os autômatos celulares.

A recorrência logística

Um modelo matemático simples, porém capaz de gerar comportamentos caóticos e imprevisíveis é a recorrência logística (*logistic map*). Imagine que desejamos criar uma equação para modelar o número de indivíduos de uma população de coelhos a partir de uma população inicial. A equação mais simples seria algo do tipo:

$$x_{n+1} = r\, x_n$$

onde r > 0 denota a taxa de crescimento. Porém, na natureza sabemos que devido a limitação de espaço e a disputa pelos recursos, populações não tendem a crescer indefinidamente. Há um ponto de equilíbrio em que o número de indivíduos tende a se estabilizar ao redor. Sendo assim, podemos definir a seguinte equação:

$$x_{n+1} = r\, x_n (1 - x_n)$$

em que o $x_n \in [0,1]$ a porcentagem de indivíduos vivos e o termo $(1 - x_n)$ tende a zero quando essa porcentagem se aproxima do valor máximo de 100%. Esse modelo é conhecido como recorrência logística. Veremos a seguir que fenômenos caóticos emergem desse simples modelo, que aparentemente possui um comportamento bastante previsível.

Primeiramente, note que o número de indivíduos no tempo n+1 é uma função quadrática do número de indivíduos no tempo n, pois:

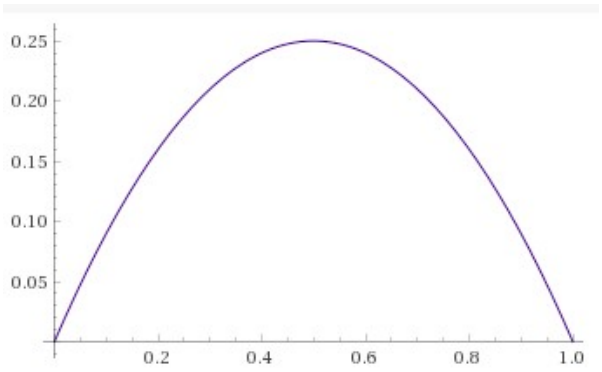
$$x_{n+1} = f(x_n) = -r\, x_n^2 + r\, x_n$$

ou seja, temos uma equação do segundo grau com $a = -r$, $b = r$ e $c = 0$. Como a < 0, a concavidade da parábola é para baixo, ou seja, ela admite um ponto de máximo. Derivando $f(x_n)$ em relação a x_n e igualando a zero, temos o ponto de máximo:

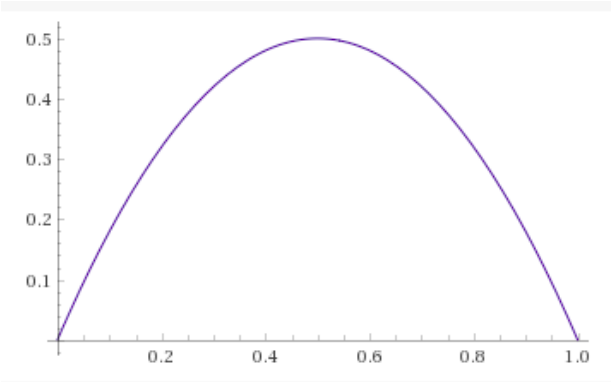
$$-2r\, x_n + r = 0$$

o que nos leva a $x_n^* = \frac{1}{2}$. Note que nesse ponto o valor da função vale: $f(x_n^*) = -\frac{r}{4} + \frac{r}{2} = \frac{r}{4}$

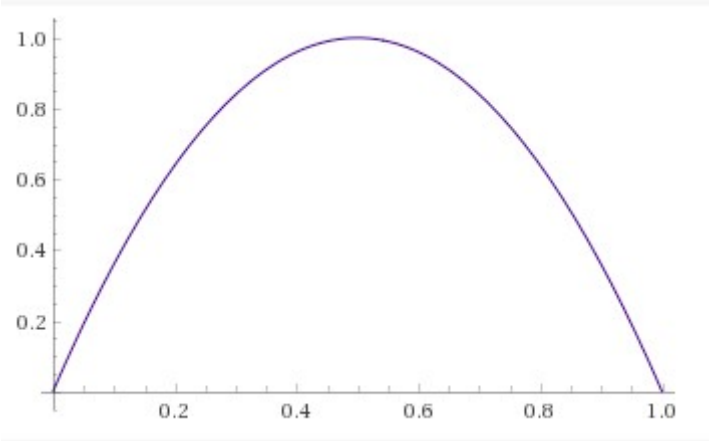
Note também que como c = 0, $f(0) = 0$, ou seja, a parábola passa pela origem. Note ainda que $f(1) = 0$, ou seja a parábola corta o eixo x no ponto x = 1. De forma gráfica temos para alguns valores de r as seguintes parábolas:



r = 1



r = 2



r = 4

Vamos simular várias iterações do método em Python para analisar o comportamento do tamanho da população em função do tempo t. O script em Python a seguir mostra uma implementação computacional do modelo utilizando 100 iterações.

```
import matplotlib.pyplot as plt

# Número de iterações para atingir equilíbrio
MAX = 100

r = float(input('Entre com a constante r: '))
x = float(input('Entre com x0: '))

population = [x]

for i in range(1, MAX):
      x = r*x*(1 - x)
      population.append(x)


print('População no longo prazo: ', population[-1])

# Plota gráfico da população pelo tempo
eixox = list(range(MAX))
plt.figure(1)
plt.plot(eixox, population)
plt.show()
```

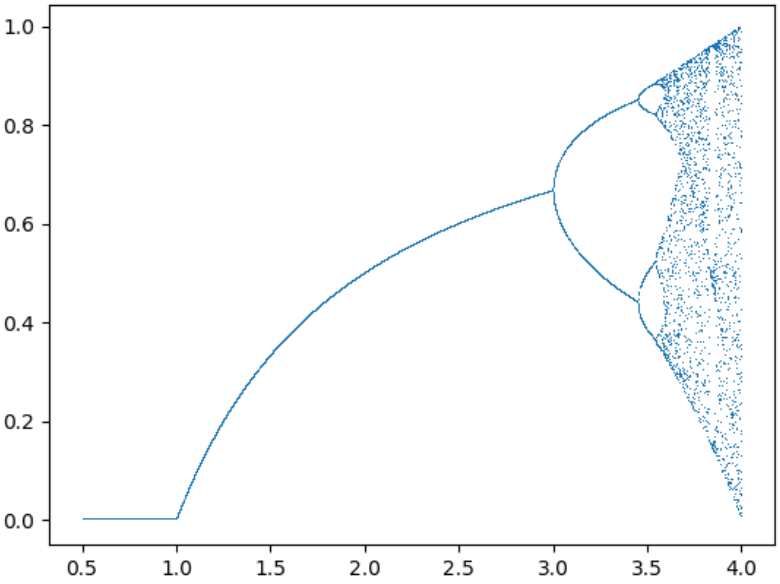
Execute o script e veja o que acontece para as entradas a seguir:

a) r = 1 e x_0 = 0.4 (extinção)
b) r = 2 e x_0 = 0.4 (equilíbrio em 50%)
c) r = 2.4 e x_0 = 0.6 (pequena oscilação, mas atinge equilíbrio em 58%)
d) r = 3 e x_0 = 0.4 (não há equilíbrio, população oscila, mas em torno de uma média)
e) r = 4 e x_0 = 0.4 (comportamento caótico, totalmente imprevisível)

Em seguida, iremos estudar o que acontece com a população de equilíbrio conforme variamos o valor do parâmetro r. A ideia é que no eixo x iremos plotar os possíveis valores de r e no eixo y iremos plotar a população de equilíbrio para aquele valor de r específico. Iremos considerar que a população do equilíbrio é obtida depois de 1000 iterações. O script em Python a seguir mostra a implementação computacional dessa análise.

```
import matplotlib.pyplot as plt
import numpy as np

# Cria um vetor com todos os possíveis valores de r
R = np.linspace(0.5, 4, 20000)

m = 0.5

# Inicializa os eixos x e y vazios
X = []
Y = []
# Loop principal (iterar para todo r em R)
for r in R:
    # Adiciona r no eixo x
    X.append(r)

    # Escolhe um valor aleatório entre 0 e 1
    x = np.random.random()

    # Gera população de equilíbrio
    for l in range(1000):
      x=r*x*(1-x)

    Y.append(x)

# Plota o gráfico sem utilizar retas ligando os pontos
plt.plot(X, Y, ls='', marker=',')
plt.show()
```

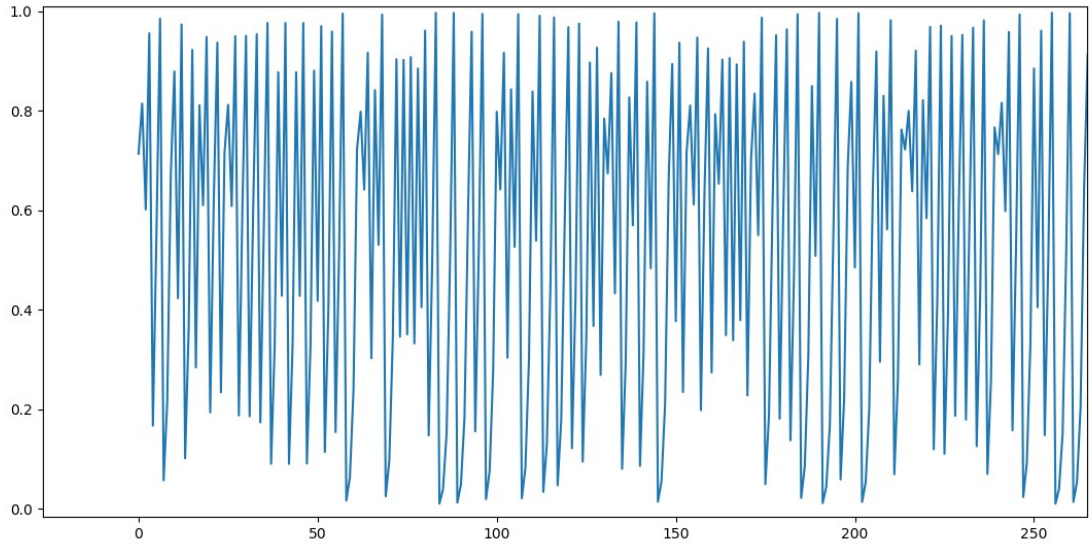
O gráfico plotado pelo script acima é conhecido como *bifurcation map*. Esse fenômeno da bifurcação ocorre como uma manifestação do comportamento caótico da população de equilíbrio para valores de r maiores que 3. Na prática, o que temos é que para um valor de r = 3.49999, a população de equilíbrio é muito diferente daquela obtida para r = 3.50000 por exemplo. Pequenas perturbações no parâmetro r causam um efeito devastador na população de equilíbrio. Esse é o lema da teoria do caos, que pode ser parafraseado pela célebre sentença: o simples bater de asas de uma borboleta pode levar ao surgimento de um furação, conhecido também como o efeito borboleta.



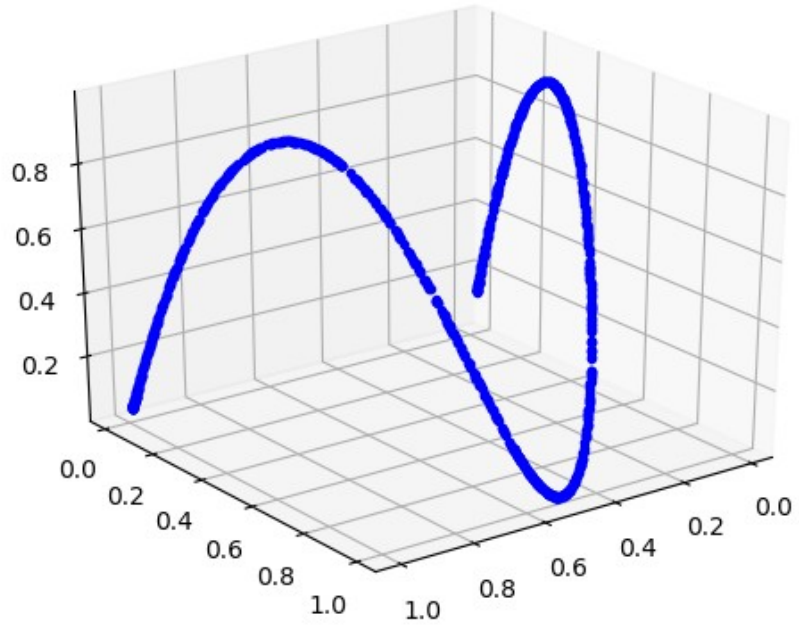
Uma das propriedades do caos é que é possível encontrar ordem e padrões em comportamentos caóticos. Por exemplo, a seguir iremos desenvolver um script em Python para plotar uma sequencia de populações, começando de uma população inicial arbitrária e utilizando o valor de r = 3.99.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

def atrator(X):
    A = X[:len(X)-2]
    B = X[1:len(X)-1]
    C = X[2:]
    #Plota atrator em 3D
    fig = plt.figure(2)
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(A, B, C, '.', c='red')
    plt.show()

# Início do script
r = 3.99
x = np.random.random()
X = [x]

for i in range(1000):
    x = r*x*(1 - x)
    X.append(x)

# Plota o gráfico da sequência
plt.figure(1)
plt.plot(X)
plt.show()
```

Note que o gráfico mostrado na figura 1 parece o de uma sequência totalmente aleatória.

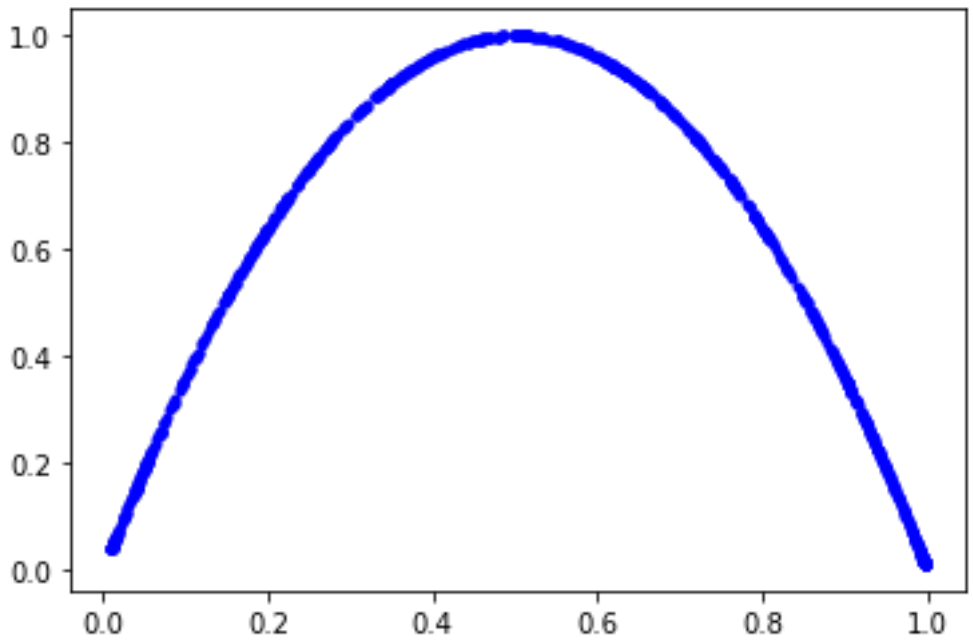


A seguir, iremos plotar cada subsequência (x_n, x_{n+1}, x_{n+2}) como um ponto no R^3. Na prática, isso significa que no eixo X iremos plotar a sequência original, no eixo Y iremos plotar a sequência deslocada de uma unidade e no eixo Z iremos plotar a sequência deslocada de duas unidades. Qual será o gráfico formado? Se de fato a sequência for completamente aleatória, nenhum padrão deverá ser observado, apenas pontos dispersos aleatoriamente pelo espaço. Mas, surpreendentemente, temos a formação do seguinte padrão, conhecido como o atrator do modelo.



Podemos repetir a análise anterior, mas agora plotando o ponto (x_n, x_{n+1}) no plano. Conforme discutido anteriormente, vimos que $x_{n+1}=f(x_n)$ é uma função quadrática, ou melhor, uma parábola. O experimento prático apenas comprova a teoria. Note que o gráfico obtido pelo script a seguir é exatamente a parábola. Conforme a teoria, note que o ponto de máximo ocorre em $x_n = 0.5$, e o valor da função nesse ponto, $f(x_n)$, é praticamente 1 (r/4 = 3.99/4).

```
import matplotlib.pyplot as plt
import numpy as np

r = 3.99
x = np.random.random()     # a população é x minúsculo!
X = [x]                    # a lista é X maiúsculo!

for i in range(1000):
     x = r*x*(1 - x)
     X.append(x)

# Plota o gráfico da sequência
plt.figure(1)
plt.plot(X)
plt.show()

A = X[:len(X)-1]
B = X[1:len(X)]

#Plota atrator em 3D
plt.figure(2)
plt.plot(A, B, '.', c='blue')
plt.show()
```



Interessante, não é mesmo? Dentro do caos, há ordem. Muitos fenômenos que observamos no mundo real parecem ser aleatórios, mas na verdade exibem comportamento caótico. A pergunta que fica é justamente essa: como distinguir um sistema aleatório de um sistema caótico? Como identificar os padrões que nos permitem enxergar a ordem em um sistema caótico? Para responder a esse questionamento precisamos mergulhar fundo na matemática dos sistemas complexos e da teoria do caos.

Autômatos celulares

Modelos de autômatos celulares definem ferramentas computacionais muito importantes para a simulação e estudo de sistemas complexos. Um sistema é dito complexo quando suas propriedades não são uma consequência natural de seus elementos constituintes vistos isoladamente. As propriedades emergentes de um sistema complexo decorrem em grande parte da relação não-linear entre as partes. Costuma-se dizer de um sistema complexo que o todo é mais que a soma das partes. Exemplos de sistemas complexos incluem redes sociais, colônias de animais, o clima e a economia.

Uma pergunta recorrente no estudo de tais sistemas é: porque e como padrões complexos emergem a partir da interação entre os elementos? Como esses padrões evoluem com o tempo? Respostas a essas perguntas não são totalmente conhecidas, mas o estudo de modelos de autômatos celulares nos auxiliam no estudo e análise de tais sistemas. Aplicações práticas são muitas e incluem:
- Autômatos celulares e composição musical
- Autômatos celulares e modelagem urbana
- Autômatos celulares e propagação de epidemias
- Autômatos celulares e crescimento de câncer

Os primeiros modelos de autômatos celulares foram propostos originalmente na década de 40 por John Von Neumann e tinham como objetivos principais:
- Representar matematicamente a evolução natural em sistemas complexos
- Desenvolver máquinas de auto-replicação, através de um conjunto de regras matemáticas objetivas
- Estudar a auto-organização em sistemas complexos

Segundo Wolfram, autômatos celulares são formados por uma rede de células que possuem seus estados alterados num tempo disscreto de acordo com seu estado anterior e o estado de suas células vizinhas. Algumas características importantes e comuns a todos os autômatos celulares são:
- Homogeneidade: as regras são iguais para todas as células
- Estados discretos: cada célula pode estar em um dos finitos estados
- Interações locais: o estado de uma célula depende apenas das células mais próximas (vizinhas)
- Processo dinâmico: a cada instante de tempo as células podem sofrer uma atualização de estado

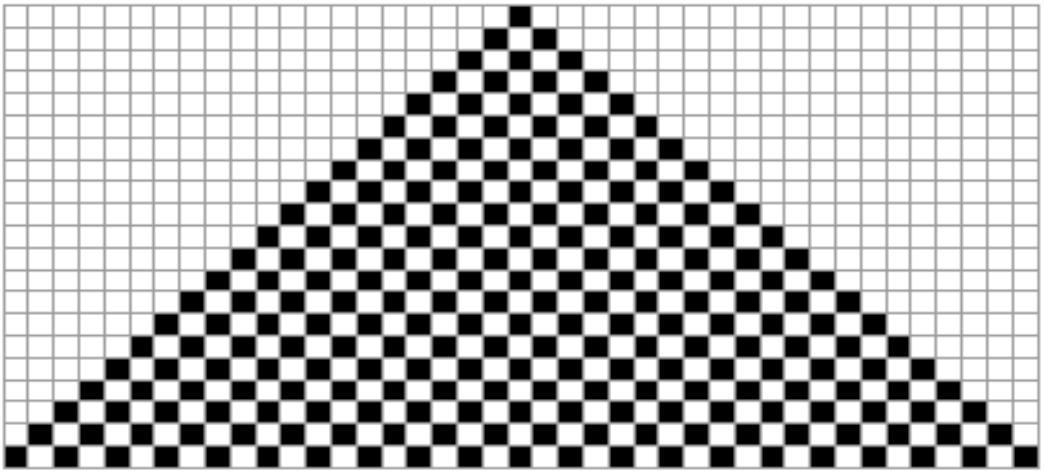
Def: Um autômatos celular é definido por uma 5-tupla de elementos

$A=(R,S,S_0,V,F)$ onde

R é a grade de células (pode ser 1D, 2D, 3D,…)
S é o conjunto de estados de uma célula c_i
S_0 é o estado inicial do sistema
V é conjunto vizinhança (define quem são os vizinhos de cada célula)
F é a função de transição (regras de governam a evolução do sistema no tempo)

Autômatos Celulares Elementares

Um autômato celular é dito elementar se o reticulado de células R é unidimensional (ou seja, pode ser representado por um vetor), o conjunto de estados S = {0, 1} e o conjunto vizinhança engloba apenas duas células: a anterior (i-1) e a posterior (i+1). Tipicamente, se uma célula assume estado 0 dizemos que ela está morta e se ela assume estado 1 dizemos que está viva. A figura a seguir ilustra as primeiras 20 gerações de um autômato celular elementar, em que no início apenas uma célula está viva (preto = vivo, branco = morto)

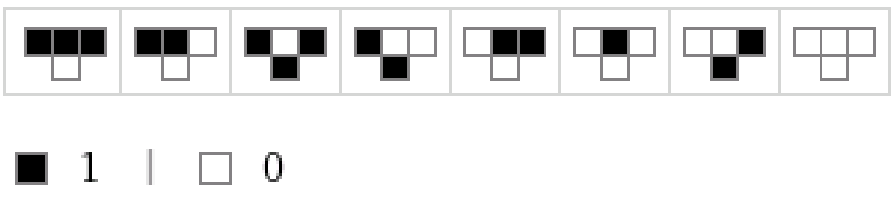


A questão é: como evoluir uma configuração de forma a construir esse padrão? Que regras são aplicadas para definir quais células vivem ou morrem na próxima geração?

A função de transição do autômato da figura é dada pela seguinte tabela.

$x_t(i-1)$	$x_t(i)$	$x_t(i+1)$	$x_{t+1}(i)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Uma forma de resumir toda essa tabela é através da seguinte representação:

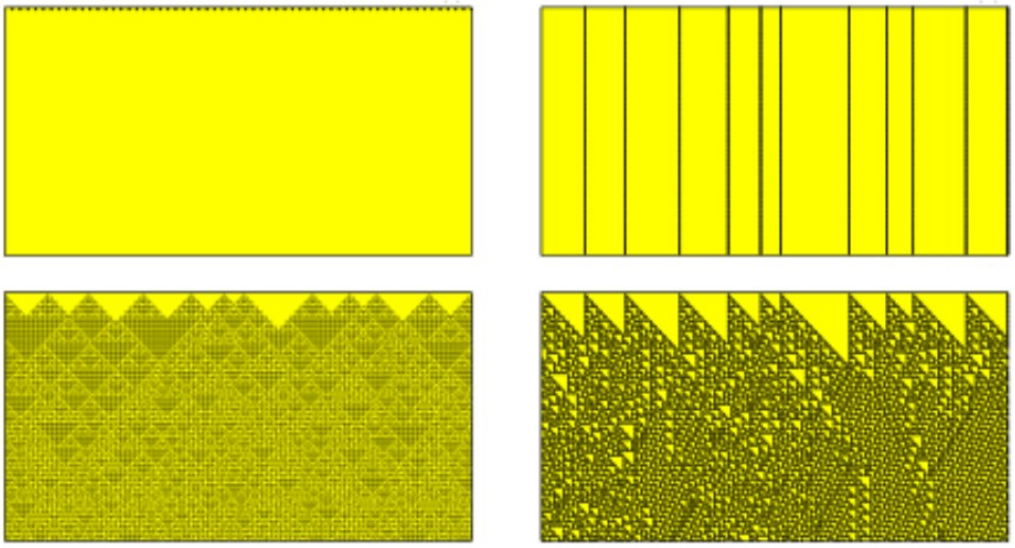


O que essa regra nos diz pode ser sumarizado em 8 fatos:

1) sempre que a célula i for morta e a (i-1) e (i+1) também forem, a célula i permanecerá morta na próxima geração.
2) sempre que a célula i for morta, a (i-1) for morta e a (i+1) for viva, a célula i viverá na próxima geração.
3) sempre que a célula i for viva e a (i-1) e a (i+1) forem mortas, a célula i morrerá na próxima geração.
4) sempre que a célula i for viva, a (i-1) for morta e a (i+1) for viva, a célula i morrerá na próxima geração.
5) sempre que a célula i for morta, a (i-1) for viva e a (i+1) for morta, a célula i viverá na próxima geração.
6) sempre que a célula i for morta e ambas (i-1) e (i+1) forem vivas, a célula i viverá na próxima geração.
7) sempre que a célula i for viva, a (i-1) for viva e a (i+1) for morta, a célula i morrerá na próxima geração.
8) sempre que a célula i for viva e ambas (i-1) e (i+1) forem vivas, a célula i morrerá na próxima geração.

Note que não existem mais combinações possíveis de 0's e 1's usando apenas 3 bits, pois conseguimos contar em binário de 0 a 7, o que resulta em 8 possibilidades. Essa regra tem um nome: é a regra 50, pois o número binário correspondente a última coluna da função de transição vale 00110010, que em binário é justamente o número 50. Sendo assim, quantas possíveis regras existem para um autômato celular elementar? Basta computar 2^8, que resulta em 256. Portanto, o número total de regras distintas é 256. Por essa razão dizemos que existem 256 autômatos celulares elementares distintos, um para cada regra. O interessante é estudar e simular o que acontece com cada um desses autômatos durante sua evolução. De acordo com Wolfram, existem 4 classes de regras para um autômato celular elementar:

- Classe 1: Estado Homogêneo
  Todas as células chegarão num mesmo estado após um número finito de estados
- Classe 2: Estável simples
  As células não possuirão todas o mesmo estado, mas eles se repetem com a evolução temporal
- Classe 3: Padrão irregular
  Não possui padrão reconhecível
- Classe 4: Estrutura complexa
  Estruturas complexas que evoluem imprevisivelmente



As regras mais interessantes são as da classe 4, pois definem um sistema complexo com propriedades dinâmicas interessantes, sendo algumas delas capazes até de simular máquinas de Turing, que são modelos computacionais capazes de serem programadas para realizar diferentes tarefas computacionais. Um exemplo de regra com essa característica é a regra 110. (Referências: https://en.wikipedia.org/wiki/Rule_110, http://www.complex-systems.com/pdf/15-1-1.pdf)

Exercício: Construa a função de transição do autômato celular elementar definido pela regra 30. Aplique a regra para evoluir a condição inicial idêntica a da figura da regra 50 (apenas uma célula viva) por 20 gerações. Repita o exercício mas agora para a regra 110.

A seguir é apresentado um algoritmo para a simulação de autômatos celulares elementares.

```
geração = vetor(N) (N é o número de células)

nova_geração = vetor(N)

evolução = matriz(MAX, N) (MAX é o número de gerações)

Inicializar geração (setar configuração inicial)

para i = 1 até MAX
     evolução[i,:] = geração
     # Percorre cada célula da geração atual
     para j = 1 até N
          Aplicar regra de transição na célula j, gerando nova_geração
     geração = nova_geração

Plotar resultados
```

Ex: Baseado no algoritmo anterior, implementar um script em Python que, dado uma regra (número de 0 a 255), evolua uma configuração inicial de tamanho N = 1000 até a geração 500.

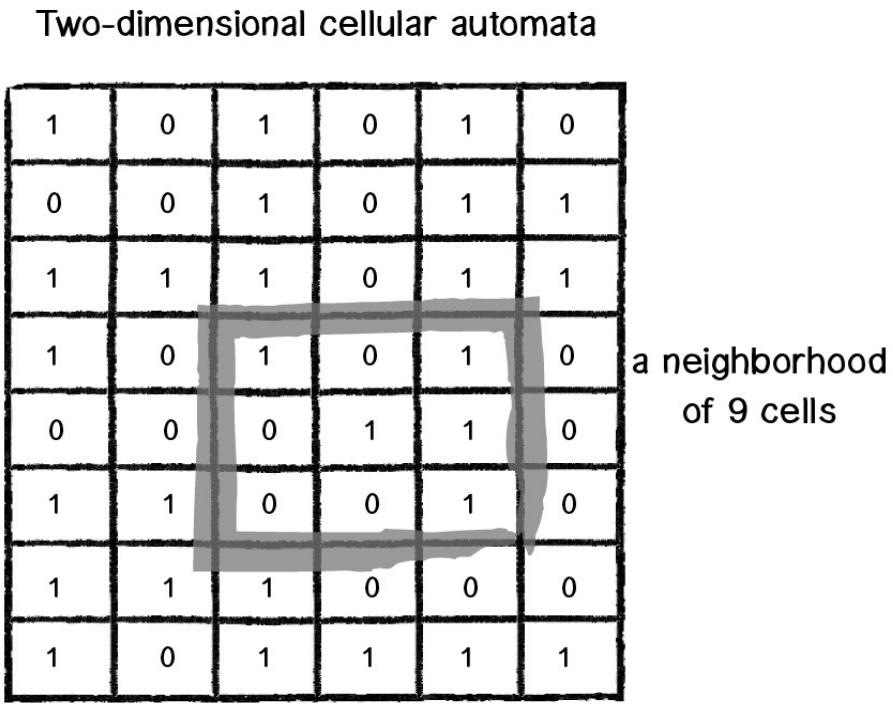
```
import numpy as np
import matplotlib.pyplot as plt

# converte um número inteiro para sua representação binária (0-255)
def converte_binario(numero):
     binario = bin(numero)
     binario = binario[2:]
     if len(binario) < 8:
          zeros = [0]*(8-len(binario))
     binario = zeros + list(binario)
     return list(binario)


# Início do script
MAX = 500
g = np.zeros(1000)
ng = np.zeros(1000)

regra = int(input('Entre com o número da regra: '))
codigo = converte_binario(regra)
```

```
# Matriz em que cada linha armazena uma geraçao do autômato
matriz_evolucao = np.zeros((MAX, len(g)))

# Define geração inicial
g[len(g)//2] = 1

# Laço principal: atualiza as gerações
for i in range(MAX):

     matriz_evolucao[i,:] = g

     # Percorre células da geração atual
     for j in range(len(g)):

          if (g[j-1] == 0 and g[j] == 0 and g[(j+1)%len(g)] == 0):
               ng[j] = int(codigo[7])
          elif (g[j-1] == 0 and g[j] == 0 and g[(j+1)%len(g)] == 1):
               ng[j] = int(codigo[6])
          elif (g[j-1] == 0 and g[j] == 1 and g[(j+1)%len(g)] == 0):
               ng[j] = int(codigo[5])
          elif (g[j-1] == 0 and g[j] == 1 and g[(j+1)%len(g)] == 1):
               ng[j] = int(codigo[4])
          elif (g[j-1] == 1 and g[j] == 0 and g[(j+1)%len(g)] == 0):
               ng[j] = int(codigo[3])
          elif (g[j-1] == 1 and g[j] == 0 and g[(j+1)%len(g)] == 1):
               ng[j] = int(codigo[2])
          elif (g[j-1] == 1 and g[j] == 1 and g[(j+1)%len(g)] == 0):
               ng[j] = int(codigo[1])
          elif (g[j-1] == 1 and g[j] == 1 and g[(j+1)%len(g)] == 1):
               ng[j] = int(codigo[0])

     g = ng.copy()  # se não usar copy ambos vetores tornam-se o mesmo

# plota matriz resultante como imagem
plt.figure(1)
plt.axis('off')
plt.imshow(matriz_evolucao, cmap='gray')
plt.savefig('Automata.png', dpi=300)
plt.show()
```

Autômatos celulares 2D

O Jogo da Vida

O autômato 2D mais conhecido sem dúvida é o jogo da vida, criado por Conway para simular a evolução de sistemas complexos a partir de regras determinísticas. O reticulado 2D é representado computacionalmente por uma matriz geralmente quadrada de células que podem estar vivas ou mortas. A função de vizinhança é definida pela vizinhança de Moore, ou seja, pelas 8 células mais próximas a uma dada célula i, conforme ilustra a figura a seguir.
A função de transição do jogo da vida tem como conceito imitar processos de nascimento de morte. A ideia básica é que um ser vivo necessita de outros seres vivos para sobreviver e procriar, mas um excesso de densidade populacional provoca a morte do ser vivo devido à escassez de recursos.



São 4 regras básicas:

R1 (Sobrevivência) – uma célula viva com 2 ou 3 células vizinhas vivas, permanece viva na próxima geração.

R2 (Morte por isolamento) – uma célula viva com 0 ou 1 vizinho vivo morre de solidão na próxima geração.

R3 (Morte por sufocamento) – uma célula viva co 4 ou mais vizinhos vivos morre por sufocamento na próxima geração.

R4 (Renascimento) – uma célula morta com exatamente 3 vizinhos vivos, renasce na próxima geração.

Isso nos leva a regra de transição conhecida como B3S23, uma vez que no código proposto B significa born (nascer) e S significa survive (sobreviver). Em outras palavras, nesse autômato em particular, uma célula nasce sempre que possui 3 vizinhas vivas ao redor e sobrevive se possui 2 ou 3 vizinhas vivas ao redor. Outras variantes de regras incluem: B15S257, B147S256, B34S4567, etc. Cada uma das regras define um autômato diferente. Ao autômato cuja regra é B3S23 dá-se o nome de Jogo da Vida.

É interessante perceber que a regra B3S23 a partir de diversas inicializações simples exibe um comportamento altamente complexo, onde padrões complexos e “seres vivos” passam a interagir de maneira bastante inesperada. Trata-se de um conjunto de regras totalmente determinísticas que levam a um comportamento completamente imprevisível (ordem x caos).

Para uma coletânea de condições iniciais verifique o link:
https://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/

Um problema comum que afeta simulações computacionais do jogo da vida é o chamado problema de valor de contorno. Isso nada mais é que uma falha ao se definir a função de transição para células na borda do sistema. Para se evitar essa indefinição, considera-se que o reticulado é na verdade um toro. Isso significa que não existem bordas, uma vez que a borda da esquerda é ligada a borda da direita, assim como a inferior é ligada a superior.

Ex: Implementar um script em Python que, dada uma configuração inicial, simule o jogo da Vida num tabuleiro de dimensões 100 x 100 por 200 gerações.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time

# cria inicialização: rpentomino nas coordenadas i, j
def init_config(tabuleiro, padrao, i, j):
     linhas = padrao.shape[0]
     colunas = padrao.shape[1]
     tabuleiro[i:i+linhas, j:j+colunas] = padrao

# Início do script
inicio = time.time()
MAX = 200
SIZE = 100

geracao = np.zeros((SIZE, SIZE))
nova_geracao = np.zeros((SIZE, SIZE))

# Cubo de dados em que cada fatia representa uma geração
matriz_evolucao = np.zeros((SIZE, SIZE, MAX))

# Define geração inicial

unbounded = np.array([[1, 1, 1, 0, 1],
                      [1, 0, 0, 0, 0],
                      [0, 0, 0, 1, 1],
                      [0, 1, 1, 0, 1],
                      [1, 0, 1, 0, 1]])

glider = np.array([[1, 0, 0],
          [0, 1, 1],
          [1, 1, 0]])

r_pentomino = np.array([[0, 1, 1],
               [1, 1, 0],
               [0, 1, 0]])

diehard = np.array([[0, 0, 0, 0, 0, 0, 1, 0],
           [1, 1, 0, 0, 0, 0, 0, 0],
           [0, 1, 0, 0, 0, 1, 1, 1]])

acorn = np.array([[0, 1, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0],
         [1, 1, 0, 0, 1, 1, 1]])

init_config(geracao, r_pentomino, SIZE//2, SIZE//2)
```

```
# Laço principal (atualiza gerações)
for k in range(MAX):

     print('Processando geração %d...' %k)
     matriz_evolucao[:,:,k] = geracao

     # Laço principal: atualiza as gerações
     for i in range(SIZE):

          for j in range(SIZE):

               vivos = geracao[i-1, j-1] + geracao[i-1, j] + \
               geracao[i-1, (j+1)%SIZE] + geracao[i, j-1] + \
               geracao[i, (j+1)%SIZE] + geracao[(i+1)%SIZE, j-1] + \
               geracao[(i+1)%SIZE, j] + geracao[(i+1)%SIZE, (j+1)%SIZE]

               if (geracao[i,j] == 1):
                    if (vivos == 2 or vivos == 3):
                         nova_geracao[i,j] = 1
                    else:
                         nova_geracao[i,j] = 0
               else:
                    if (vivos == 3):
                         nova_geracao[i,j] = 1

     geracao = nova_geracao.copy()

fim = time.time()
print('Tempo gasto na simulação: %.2f s' %(fim-inicio))

# Gera animação da evolução do sistema
fig = plt.figure(1)
plt.axis('off')
lista = []
for i in range(MAX):
     im = plt.imshow(matriz_evolucao[:,:,i], cmap='gray')
     lista.append([im])

ani  =  animation.ArtistAnimation(fig,  lista,  interval=100,  blit=True,
repeat_delay=1000)

ani.save('r_pentomino.gif', writer='imagemagick')

plt.show()
```

Aula 2 – Complexidade de algoritmos

No estudo de programação de computadores, uma característica fundamental dos algoritmos que vamos implementar em uma linguagem de programação é o seu custo computacional. Em outras palavras, se temos um problema P e dois algoritmos A_1 e A_2 que resolvem P, um dos critérios mais utilizados na escolha de qual a melhor opção é a complexidade computacional do algoritmo, que em termos práticos é uma aproximação para o número de operações que serão executadas pelo algoritmo quando o tamanho da entrada é igual a n. Tipicamente, o número de operações necessárias para um algoritmo resolver um problema é uma função crescente de n (tamanho do problema), uma vez que quanto maior for a entrada, maior o número de operações necessárias.

Na verdade, durante a análise de complexidade de um programa, estamos interessados em duas propriedades básicas:
i. quantidade de memória utilizada
ii. tempo de execução

Quanto menores as quantidades definidas acima, melhor será um algoritmo, sendo que o tempo de execução na verdade é estimado indiretamente a partir do número de instruções a serem executadas.

A quantidade de memória disponível para um programa pode ser aumentada de maneira relativamente simples, de modo que isso não representa um grande gargalo (podemos duplicar a memória RAM e tudo bem). Além disso, a memória utilizada pode ser estimada observando as variáveis utilizadas pelo algoritmo. Quanto mais variáveis e quanto maiores elas forem, no caso de listas, vetores e matrizes, mais memória elas exigem. Ou seja, antes mesmo da execução de um programa, é possível ter noção de quanta memória ele precisará para executar. Porém, no que diz respeito a tempo de execução, há alguns fatores limitantes, dentre os quais:

a) Como estimar o tempo de execução de um programa antes de executá-lo?
b) O tempo de execução de um programa depende do processador, ou seja, um mesmo programa pode levar 5 segundos em um PC novo e 10 segundos em um laptop antigo e velho.
c) Podemos lidar com um algoritmo que demoraria 50 anos para executar. Como esperar?

Veja o exemplo a seguir:

```
import time

def sum_of_n(n):
     start = time.time()
     the_sum = 0
     for i in range(1, n+1):
          the_sum = the_sum + i
     end = time.time()
     return (the_sum, end-start)
```

Se executarmos a função 5 vezes, passando o valor de n como 10000, ou seja:

```
for i in range(5):
     print("Sum is %d required %.7f seconds" % sum_of_n(10000))
```

teremos como resultado os seguintes valores:

```
Sum is 50005000 required 0.0018950 seconds
Sum is 50005000 required 0.0018620 seconds
Sum is 50005000 required 0.0019171 seconds
```

```
Sum is 50005000 required 0.0019162 seconds
Sum is 50005000 required 0.0019360 seconds
```

Esse tempo depende diretamente do processador utilizado na execução. O que é certo é que se aumentarmos o valor de n para 1 milhão, ou seja, n = 1000000, o tempo gasto será maior. Portanto, há uma relação entre o tamanho da entrada n e o tempo gasto: isso ocorre porque o número de iterações na repetição aumenta, fazendo com que o número de instruções executadas pelo processador aumente.

```
for i in range(5):
     print("Sum is %d required %.7f seconds" % sum_of_n(1000000))

Sum is 500000500000 required 0.1948988 seconds
Sum is 500000500000 required 0.1850290 seconds
Sum is 500000500000 required 0.1809771 seconds
Sum is 500000500000 required 0.1729250 seconds
Sum is 500000500000 required 0.1646299 seconds
```

Para contornar os problemas citados acima, a análise de algoritmos busca fornecer uma maneira objetiva de estimar o tempo de execução de um programa pelo número de instruções que serão executadas. Dessa forma, quanto maior o número de repetições existentes no programa, maior será o tempo de execução. Claramente, calcular de maneira precisa o exato número de instruções necessárias para um programa pode ser extremamente complicado, ou até mesmo impossível. É aqui que entra a notação Big-O. Conforme mencionado previamente, o número de operações a serem executadas em um programa é uma função do tamanho da entrada, ou seja, f(n). Assim, podemos estudar o comportamento assintótico de tais funções, ou seja, o que ocorre com elas quando n cresce muito (tende a infinito).

Notação Big-O

Ao invés de contar exatamente o número de instruções de um programa, é mais tratável matematicamente analisar a ordem de magnitude da função f(n). Vejamos um simples exemplo com a função definida anteriormente. As instruções mais relevantes para esse tipo de análise são atribuições (=) envolvendo operações aritméticas (+, -, * e /). Note que ao entrar na função temos 1 atribuição. Depois disso, o loop é executado n vezes, pois i inicia com 1 e termina com valor menor que n+1, ou seja, termina em n. A nossa função fica definida como $T(n)=n+1$.

Isso significa que para um problema de tamanho n, essa função executa n + 1 instruções. O que ocorre na prática, é que cientistas da computação verificaram que quando n cresce, apenas uma parte dominante da função é importante. Essa parte dominante é o que motiva a definição na notação O(f(n)). No exemplo da função T(n), no que conforme n cresce o termo 1 torna-se desprezível. Por isso, dizemos que O(T(n)) = n, ou seja, esse é uma algoritmo de ordem linear.

Uma convenção comum na análise de algoritmos é justamente desprezar instruções de inicialização, pois elas ocorrem apenas um número fixo e finito de vezes, não sendo função de n. Assim, a função T(n) também pode ser escrita como T(n) = n. Isso ocorre porque constantes possuem ordem zero, ou seja, nesse caso temos que O(1) = 0.

Para definirmos a notação Big-O, vejamos um outro exemplo. Considere o programa em Python a seguir, em que matrix é uma matriz n x n preenchida com números aleatórios.

```
totalSum = 0
for i in range(n):
      rowSum[i] = 0
      for j in range(n):
            rowSum[i] = rowSum[i] + matrix[i,j]
      totalSum = totalSum + rowSum[i]
```

Podemos opcionalmente a partir de agora desprezar as operações de inicializações, caso necessário, pois como elas são em número constante, não fazem diferença para a ordem de magnitude da função f(n). Vamos calcular o número de instruções a serem executadas por esse programa. Note que no loop mais interno (j) temos n iterações. Assim para cada valor de i no loop mais externo, temos n + 2 operações. Como temos n possíveis valores para i, chegamos em:

$$T(n)=n(n+2)=n^2+2n$$

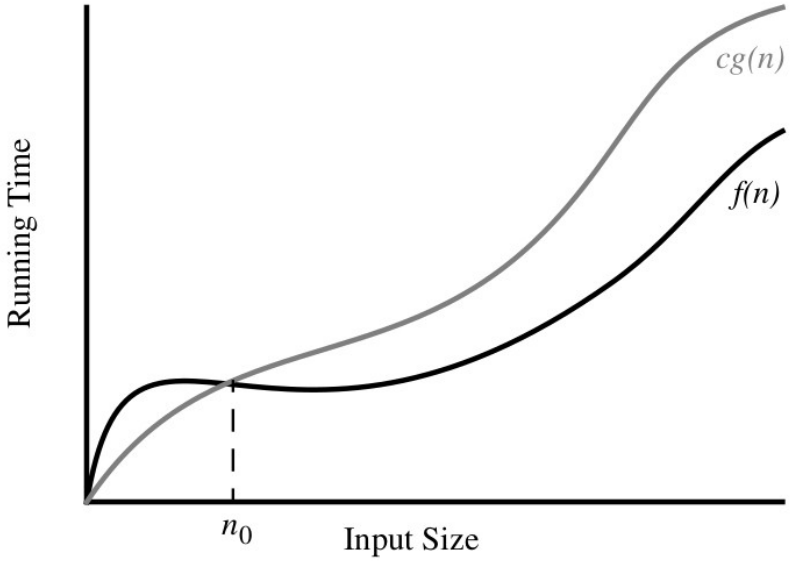
Suponha que exista uma função f(n), definida para todos os inteiros maiores ou iguais a zero, tal que para uma constante c e uma constante m:

$$T(n)\leq c\,f(n)$$

para todos os valores suficientemente grandes $n\geq m$ (quando n é grande). Então, dizemos que esse algoritmo tem complexidade de O(f(n)). A função f(n) indica a taxa de crescimento na qual a execução de uma algoritmo aumenta, conforme o tamanho da entrada n aumenta. Voltemos ao exemplo anterior. Vimos que $T(n)=n^2+n$. Note que para c = 2 e $f(n)=n^2$, temos:

$n^2+2n\leq 2n^2$ para todo n > 1

o que implica em dizer que o algoritmo em questão é $O(n^2)$. A função $f(n)=n^2$ não é a única escolha que satisfaz $T(n)\leq c\,f(n)$. Por exemplo, poderíamos ter escolhido $f(n)=n^3$, o que nos levaria a dizer que o algoritmo tem complexidade $O(n^3)$. Porém, o objetivo da notação Big-O é o limite superior mais apertado ou justo possível. Trata-se de uma maneira de estudar a taxa de crescimento assintótico de funções.

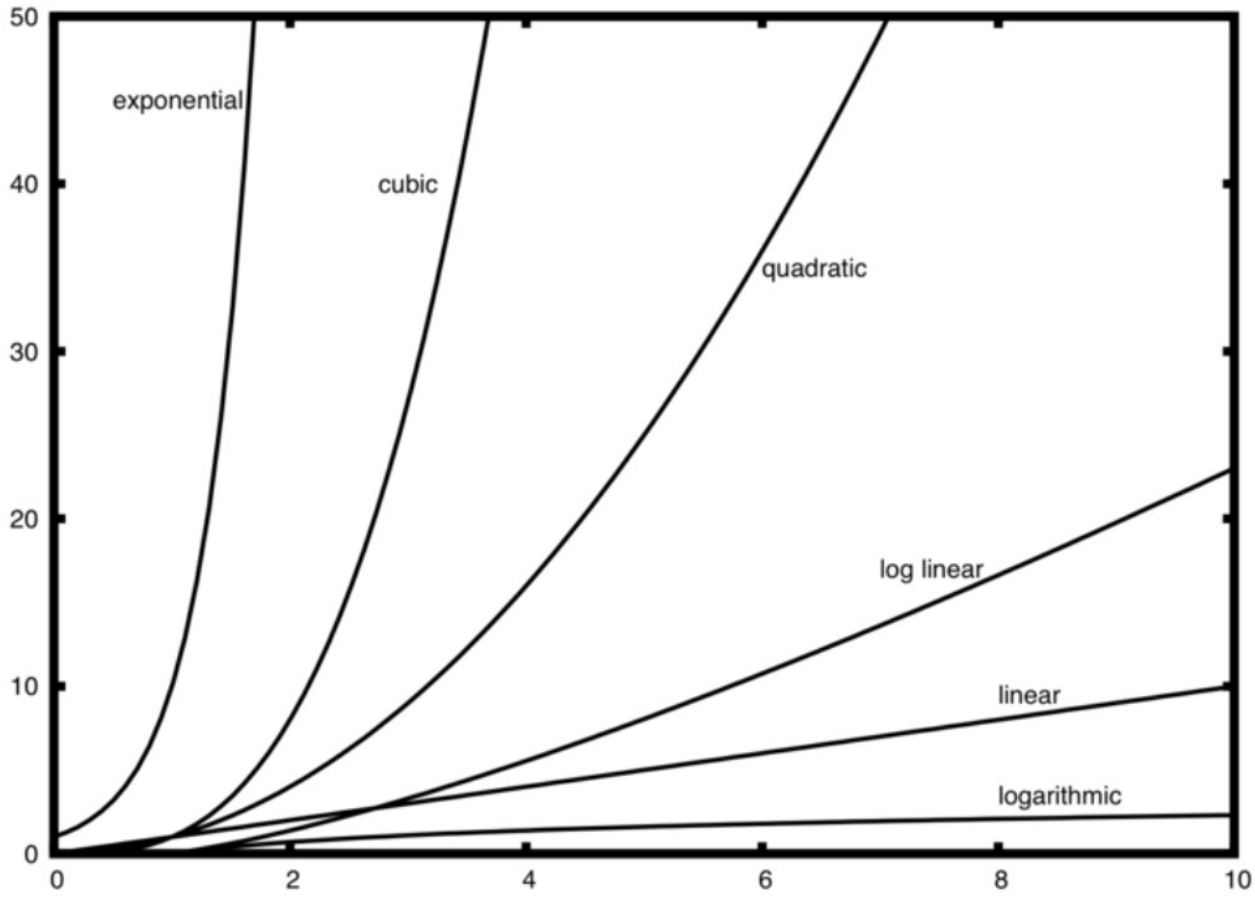


As vezes, o desempenho de um algoritmo não depende apenas do tamanho da entrada, mas também dos elementos que compõem o vetor/matriz. Veremos isso no caso dos algoritmos de ordenação. Se o vetor de entrada está quase ordenado, o algoritmo leva menos tempo, ou seja, é mais rápido. Em

cenários como esse, podemos realizar a análise do algoritmo em três situações distintas: melhor caso (seria o vetor já ordenado), caso médio (valores aleatórios) e pior caso (o vetor em ordem decrescente, totalmente desordenado). Costuma ignorar o melhor caso, pois ele é muito raro de acontecer na prática. Em geral, realizamos as análises no caso médio ou pior caso. Costuma-se dividir os algoritmos nas seguintes classes de complexidade:

f(n)	**Classe**
1	Constante
log n	Logarítmica
n	Linear
n log n	Log-linear
n^2	Quadrática
n^3	Cúbica
2^n	Exponencial

O comportamento assintótico dessas funções é muito diferente. De modo geral, é raro um algoritmo ter complexidade constante (mas há estruturas de dados em que a inserção ou remoção de elementos possui tempo constante), sendo que a classe logarítmica é quase sempre o melhor que podemos obter. Quando um algoritmo é da classe exponencial, ele é praticamente inviável, pois para n = 100, já temos uma valor extremamente elevado de operações, o que seria suficiente para fazer o tempo de execução superar dezenas de anos nos computadores mais rápidos do planeta. A figura a seguir mostra a taxa de crescimento dessas funções.



Construindo T(n)

Ao invés de contar o número exato de comparações lógicas ou operações aritméticas, avaliamos um algoritmo por meio de instruções básica. Para os nossos propósitos, uma instrução básica é toda operação que envolve uma atribuição. Por exemplo, se temos a seguinte linha de código:

```
x = a*b + c*d + e*f
```

então ela toda equivale a uma instrução básica, mesmo envolvendo 3 multiplicações e 2 adições. Assume-se que cada instrução básica possui tempo constante, ou seja, possui O(1). O número total de operações de um algoritmo é dado pela somatória dos tempos individuais $f_i(n) = 1$ sequencialmente, ou seja:

$$T(n)=f_1(n)+f_2(n)+...+f_k(n)$$

Somatórios

Séries e somatórios aparecem com frequência em diversos problemas da matemática e da computação. Na análise de algoritmos é bastante comum termos que resolver somatórios.

Para iniciar com um exemplo simples, suponha que desejamos somar todas as potências de 2, iniciando em 2^1 e terminando em 2^{10}. A maneira explícita de escrever essa soma é:

$$2+2^2+2^3+2^4+...+2^{10}$$

É possível expressar esse somatório como:

$$\sum_{k=1}^{10} 2^k$$

A seguir veremos diversas propriedades úteis na manipulação e resolução de somatórios.

1) Substituição de variáveis

Seja o seguinte somatório:

$$\sum_{k=1}^{n} 2^k$$

Definindo i = k – 1, temos que k = i + 1. Se k inicia em 1, i deve iniciar em zero. Para o limite superior, vale o mesmo. Se k vai até n, i deve ir até n – 1. Dessa forma, podemos expressar o somatório como:

$$\sum_{i=0}^{n-1} 2^{i+1}$$

2) Distributiva: para toda constante c

$$\sum_{k\in A} c\,f(k)=c\left(\sum_{k\in A} f(k)\right)$$

Em outras palavras, é possível mover as constantes para fora do somatório colocando-as em evidência.

3) Associativa: somatórios de somas é igual a somas de somatórios

$$\sum_{k\in A}(f(k)+g(k))=\sum_{k\in A}f(k)+\sum_{k\in A}g(k)$$

4. Decomposição de domínio: Seja A_1 e A_2 uma partição de A. Então,

$$\sum_{k\in A}f(k)=\left[\sum_{k\in A_1}f(k)\right]+\left[\sum_{k\in A_2}f(k)\right]$$

Em outras palavras, podemos decompor o somatório em dois somatórios menores, desde que cada valor de índice apareça no domínio de uma dos subconjuntos. Por exemplo, se A é o conjunto dos naturais, A_1 é o conjunto dos pares e A_2 é o conjunto dos ímpares, todo índice em A_1 não está em A_2.

5. Comutatividade: se p(.) é uma permutação do domínio A, então

$$\sum_{k\in A}f(k)=\sum_{k\in A}f(p(k))$$

ou seja, podemos embaralhar os termos de um somatório que o valor final não muda.

6. Somas telescópicas: considere uma sequencia de números reais $x_1, x_2, x_3, \ldots, x_n, x_{n+1}$. Então, a identidade a seguir é válida:

$$\sum_{k=1}^{n}(x_{k+1}-x_k)=x_{n+1}-x_1$$

ou seja, o valor do somatório das diferenças é igual a diferença entre o último elemento e o primeiro

Prova:

1. Pela propriedade associativa (3), temos:

$$S=\sum_{k=1}^{n}(x_{k+1}-x_k)=\sum_{k=1}^{n}x_{k+1}-\sum_{k=1}^{n}x_k$$

2. Por substituição de variáveis (1), temos:

$$S=\sum_{i=2}^{n+1}x_i-\sum_{k=1}^{n}x_k$$

3. Removendo o último termo do primeiro somatório e o primeiro termo do segundo, temos:

$$S=\sum_{i=2}^{n}x_i+x_{n+1}-x_1-\sum_{k=2}^{n}x_k=x_{n+1}-x_1$$

A prova está concluída.

Analisando códigos em Python

Considere o exemplo a seguir, com duas estruturas de repetição.

```
def ex2(n):
     count = 0
     for i in range(n):
          count += 1
     for j in range(n):
          count += 1
     return count
```

Note que temos uma atribuição inicial (1) e logo dois loops com n iterações. Cada um deles, contribui com n para o total, de modo que no total temos T(n) = 2n + 1, o que resulta em uma complexidade O(n).

Ex: Considere o algoritmo a seguir:

```
def ex3(n):
     count = 0
     for i in range(n):
          for j in range(n):
               count += 1
     return count
```

Nesse caso, o loop interno tem n operações. Como o loop externo é executado n vezes, e temos uma inicialização, o total de operações é $T(n)=n^2+1$, o que resulta em $O(n^2)$.

Note que nem todos os loops aninhados possuem custo quadrático. Considere o código a seguir:

```
def ex3(n):
     count = 0
     for i in range(n):
          for j in range(10):
               count += 1
     return count
```

O loop mais interno é executado 10 vezes (número constante de vezes). Sendo assim, o total de operações é T(n) = 10n + 1, o que resulta em O(n).

Ex: Considere esse código em que o loop interno executa um número variável de vezes.

```
def ex5(n):
     count = 0
     for i in range(n) :
          for j in range(i+1) :
               count += 1
     return count
```

Note que quando i = 1, o loop interno executa uma vez, quando n = 2, o loop interno executa duas vezes, quando n = 3, o loop interno executa 3 vezes, e assim sucessivamente. Assim, o número de vezes que a variável count é incrementada é igual a: 1 + 2 + 3 + 4 + + … n

Devemos resolver esse somatório para calcular a complexidade dessa função.

$$\sum_{k=1}^{n} k$$

Primeiramente, note que

$$(k+1)^2 = k^2 + 2k + 1$$

o que implica em

$$(k+1)^2 - k^2 = 2k + 1$$

Assim, $\sum_{k=1}^{n} [(k+1)^2 - k^2] = \sum_{k=1}^{n} [2k+1]$. Porém, o lado esquerdo é uma soma telescópica e temos:

$$\sum_{k=1}^{n} [(k+1)^2 - k^2] = (n+1)^2 - 1$$

Dessa forma, podemos escrever:

$$(n+1)^2 - 1 = \sum_{k=1}^{n} [2k+1]$$

Aplicando a propriedade associativa, temos:

$$(n+1)^2 - 1 = 2\sum_{k=1}^{n} k + \sum_{k=1}^{n} 1$$

o que nos leva a:

$$2\sum_{k=1}^{n} k = n^2 + 2n + 1 - 1 - n = n^2 + n$$

Finalmente, colocando n em evidência e dividindo por 2, finalmente temos:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Portanto, T(n) é igual a:

$$T(n) = \frac{1}{2}(n^2 + n) + 1$$

o que resulta em $O(n^2)$.

Ex: Considere a função em Python a seguir.

```
def ex6(n):
     count = 0
     i = n
     while i > 1:
          count += 1
          i = i // 2       # divisão inteira
     return count
```

Essa função calcula quantas vezes o número pode ser dividido por 2. Por exemplo, considere a entrada n = 16. Em cada iteração esse valor será dividido por 2, até que atinja o zero.

16 → 8 → 4 → 2 → 1

A variável count termina a função valendo 4, pois $2^4 = 16$.

Se n = 25, temos:

25 → 12 → 6 → 3 → 1

A variável count termina a função valendo 4, pois $2^4 < 25 < 2^5$

Se n = 40, temos:

40 → 20 → 10 → 5 → 2 → 1

A variável count termina a função valendo 5, pois $2^5 < 40 < 2^6$

Portanto, o número de iterações do loop é $\log_2 n$. Dentro do loop existem duas instruções, portanto neste caso teremos:

$T(n) = 1 + 2\lfloor \log_2 n \rfloor$, onde a função piso(x) retorna o maior inteiro menor que x.

o que resulta em $O(\log_2 n)$.

Ex: Considere a função em Python a seguir.

```
def ex7(n):
     count = 0
     for i in range(n):
          count += ex6(n)
     return count
```

Note que, como a função ex6(n) tem complexidade logarítmica, e o loop tem n iterações, temos que a complexidade da função em questão é $O(n \log_2 n)$.

Ex: Verifique a complexidade do algoritmo em Python a seguir, computando primeiramente o número de instruções a serem executadas.

```
a = 5
b = 6
c = 10
for i in range(n):
     for j in range(n):
```

```
          x = i * i
          y = j * j
          z = i * j
for k in range(n):
     w = a * k + 45
     v = b * b
d = 33
```

Iniciamos com os loops aninhados (i e j), onde temos 3 operações de ordem constante, resultando em $3n^2$ operações, pois são n operações no loop mais interno vezes as n vezes do loop mais externo. No segundo loop (k), temos 2 operações de ordem constante, o que resulta em 2n operações. Por fim, há 4 operações de tempo constante fora dos loops. Sendo assim, temos:

$$T(n)=3n^2+2n+4$$

Quando temos uma expressão polinomial, é fácil perceber que o termo com o maior grau domina os demais. Neste caso, o termo dominante é quadrático, portanto a complexidade do algoritmo em questão é $O(n^2)$.

Ex: Escreva duas funções para encontrar o menor elemento de uma lista, uma que compara cada elemento com cada outro elemento da lista e outra que percorre a lista uma única vez. Calcule a complexidade de cada um deles, analisando o pior caso.

```
def menor_A(L):
     n = len(L)
     for i in range(n):
          x = L[i]
          menor = n*[0]
          for j in range(n):
               if x <= L[j]:
                    menor[j] = 1
          if sum(menor) == n:
               return (i, x)
```

```
def menor_B(L):
     pos = 0
     n = len(L)
     menor = L[pos]
     for i in range(n):
          if L[i] < menor:
               pos = i
               menor = L[i]
     return (pos, menor)
```

a) Vamos analisar o algoritmo menor_A: note que, para cada elemento x da lista L, ele verifica se x é menor ou igual a todos os demais. Ele faz a marcação com o número 1 na posição de x na lista menor. Se x for menor ou igual a todos os elementos de L, teremos exatamente n 1's na lista menor, o que fará com que a soma dos elementos de L seja igual a n.

Pior caso: o menor elemento está na última posição de L

Loop mais interno (j) tem n execuções de um comando
Loop mais externo (i) tem n execuções de 2 comandos e do loop mais interno
Inicialização de n é 1 comandos

Assim, temos:

$T(n) = 1 + n(2 + n) = n^2 + 2n + 1$

o que resulta em $O(n^2)$.

b) Vamos analisar o algoritmo menor_B: note que iniciamos o menor elemento como o primeiro elemento da lista L, Então, percorremos a lista verificando se o elemento atual é menor que atual menor. Se ele for, então atualizamos o menor com esse elemento.

Pior caso: menor elemento está na última posição de L.

Loop tem n execuções com 2 instruções
Inicialização de 3 variáveis

Assim, temos:

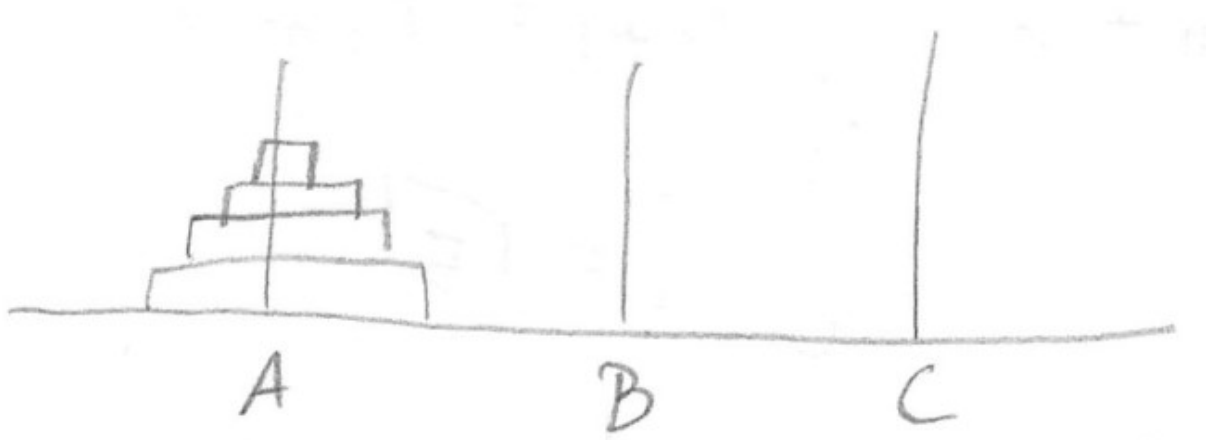
$T(n) = 3 + 2n$

o que resulta em O(n).

Portanto, o algoritmo menor_B é mais eficiente que o algoritmo menor_A.

O problema da torre de Hanói

Imagine que temos 3 hastes (A, B e C) e inicialmente n discos de tamanhos distintos empilhados na haste A, de modo que discos maiores não podem ser colocados acima de discos menores.



O objetivo consiste em mover todos os discos para uma outra haste. Há apenas duas regras:

1. Podemos mover apenas um disco por vez
2. Não pode haver um disco menor embaixo de um disco maior

Vejamos o que ocorre para diferentes valores de n (número de discos).

Se n = 1, basta um movimento: Move A, B

Se n = 2, são necessários 3 movimentos:
Move A, B
Move A, C
Move B, C

Se n = 3, são necessários 7 movimentos:
Move A, B
Move A, C
Move B, C

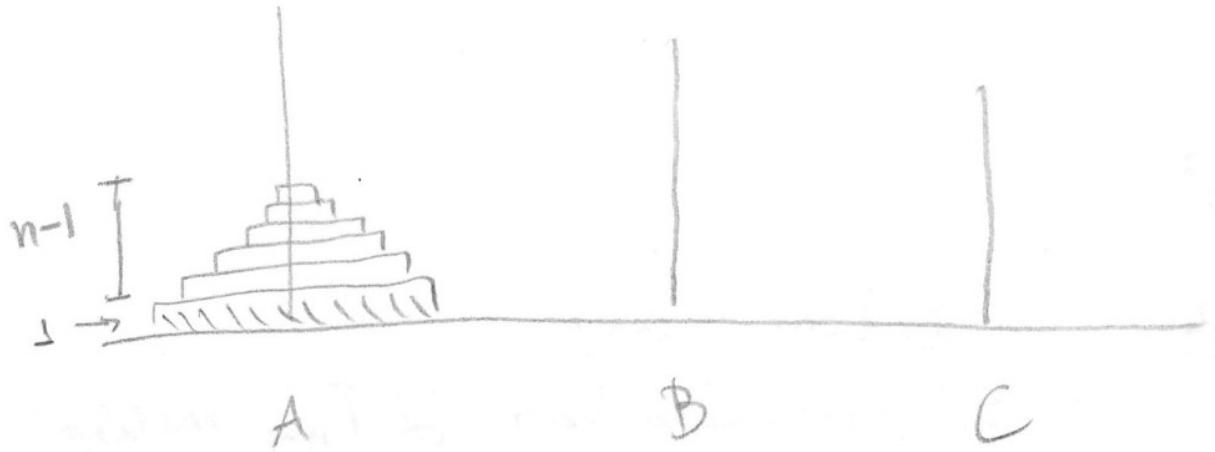
Move A, B
Move C, A
Move C, B
Move A, B

Utilizando uma abordagem recursiva, note que são 3 movimentos para os dois menores discos, 1 para o maior e mais 3 movimentos para os dois menores

Se n = 4, são necessários 15 movimentos: utilizando a abordagem recursiva, temos 7 movimentos para os 3 menores discos, 1 movimento para o maior e mais 7 movimentos para os 3 menores, o que totaliza 7 + 1 + 7 = 15 movimentos

Se n= 5, teremos 15 + 1 + 15 = 31 movimentos

A essa altura deve estar claro que temos a seguinte lógica:



Para mover n – 1 discos menores: T_{n-1} movimentos

Para mover o maior disco: 1 movimento

Para mover de volta os n – 1 discos menores: T_{n-1} movimentos

Assim, a recorrência fica definida como:

$$T_n = 2T_{n-1} + 1$$
$$T_1 = 1$$

onde T_n denota o número de instruções necessárias para resolvermos o problema das n torres de Hanói. Porém, se quisermos descobrir o número de movimentos para n = 100, devemos calcular todos os termos da sequência de 2 até 100.

Pergunta: Como calcular uma função T(n) dada a recorrência?

Como resolver essa recorrência, ou seja, obter uma fórmula fechada? Vamos expandir a recorrência.

$T_1 = 1$
$T_2 = 2T_1 + 1$
$T_3 = 2T_2 + 1$

$T_4 = 2T_3 + 1$
$T_5 = 2T_4 + 1$
$T_6 = 2T_5 + 1$
…
$T_{n-2} = 2T_{n-3} + 1$
$T_{n-1} = 2T_{n-2} + 1$
$T_n = 2T_{n-1} + 1$

A ideia consiste em somar tudo do lado esquerdo e somar tudo do lado direito e utilizar a igualdade para chegar em uma expressão fechada. Porém, gostaríamos que a soma fosse telescópica, para simplificar os cálculos. Partindo de baixo para cima, note que para o termo T_{n-1} ser cancelado, a penúltima equação precisa ser multiplicada por 2. Para que o termo T_{n-2} seja cancelado, a antepenúltima equação precisa ser multiplicada por 2^2. E assim sucessivamente, o que nos leva ao seguinte conjunto de equações:

$2^{n-1}T_1 = 2^{n-1}$
$2^{n-2}T_2 = 2^{n-1}T_1 + 2^{n-2}$
$2^{n-3}T_3 = 2^{n-2}T_2 + 2^{n-3}$
$2^{n-4}T_4 = 2^{n-3}T_3 + 2^{n-4}$
$2^{n-5}T_5 = 2^{n-4}T_4 + 2^{n-5}$
$2^{n-6}T_6 = 2^{n-5}T_5 + 2^{n-6}$
…
$2^2T_{n-2} = 2^3T_{n-3} + 2^2$
$2T_{n-1} = 2^2T_{n-2} + 2$
$T_n = 2T_{n-1} + 1$

Somando todas as linhas, temos uma soma telescópica, pois os mesmos termos aparecem do lado esquerdo e direto das igualdades, o que resulta em:

$T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + 2^{n-4} + \ldots + 2^2 + 2^1 + 2^0$

Esse somatório pode ser escrito como:

$$T(n)=\sum_{k=0}^{n-1} 2^k$$

Note que $2^{k+1}=2^k 2$, o que implica em $2^{k+1}=2^k+2^k$, e portanto, $2^k=2^{k+1}-2^k$.

O somatório em questão fica definido por uma soma telescópica:

$$T(n)=\sum_{k=0}^{n-1}\left(2^{k+1}-2^k\right)$$

Pela definição de somas telescópicas, temos:

$$T(n)=\sum_{k=0}^{n-1}\left(2^{k+1}-2^k\right)=2^n-2^0=2^n-1$$

Portanto, esse é a fórmula fechada para o número de movimentos necessários para resolver a torre de Hanói com n discos. Trata-se de um algoritmo exponencial. Por exemplo, se n = 100, o número de movimentos a ser executados é:

126765060022822940149670320537 6

Fazendo um rápido teste, a função a seguir mede o tempo gasto pelo Python para executar uma única instrução:

```
import time

def tempo():
     inicio = time.time()
     x = 1 + 2 + 3
     print(x)
     fim = time.time()
     return(fim-inicio)
```

A execução da função em um processador Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz demora cerca de 6.556×10^{-5} segundos. Assim, o tempo estimado para resolver esse problema com um programa em Python seria de aproximadamente:

$126765060022822940149670320537 6 \times 6.556 \times 10^{-5} = 8.310 \times 10^{25}$ segundos

o que é igual a

2.308×10^{22} horas = 9.618×10^{20} dias = 2.63×10^{18} anos

Sabendo que a idade do planeta Terra é estimada em 4.543×10^{9} anos, se esse programa tivesse sua execução iniciada no momento da criação do planeta, estaria executando até hoje. Estima-se que o Big Bang (origem do universo) tenha ocorrido a cerca de 13 bilhões de anos. Isso é menos tempo do que seria necessário para resolver a Torre de Hanói com 100 discos. Por essa razão, dizemos que algoritmos exponenciais são inviáveis computacionalmente, pois eles só podem ser executados para valores muito pequenos de n.

Complexidade de algumas funções Python

Em Python, é muito comum utilizarmos funções nativas da linguagem para manipulas listas, vetores e matrizes. A seguir apresentamos uma lista com algumas das principais funções que operam sobre listas e sua respectiva complexidade.

Operação	**Complexidade**	**Descrição**
L[i] = x	O(1)	Atribuição
L.append(x)	O(1)	Insere no final
L.pop()	O(1)	Remove do final (último elemento)
L.pop(i)	O(n)	Remove da posição i
L.insert(i, x)	O(n)	Insere na posição i
x in L	O(n)	Verifica se x pertence a lista (busca)
L[i:j]	O(k)	Retorna sublista dos elementos de i até j-1 (k)
L.reverse()	O(n)	Inverte a lista
L.sort()	O(n log n)	Ordena os elementos da lista

Busca sequencial x Busca binária

Uma tarefa fundamental na computação consiste em dado uma lista e um valor qualquer, verificar se aquele valor pertence a lista ou não. Essa funcionalidade é usada por exemplo em qualquer sistema que exige o login de um usuário (para verificar se o CPF da pessoa está cadastrada). Faça uma função que, dada uma lista de inteiros L e um número inteiro x, verifique se x está ou não em L. A função deve retornar o índice do elemento (posição) caso ele pertença a ele ou o valor lógico False se ele não pertence a L. (isso equivale ao operador in de Python)

```
def busca_sequencial(L, x):
      achou = False
      i = 0
      while i < len(L) and not achou:
            if (L[i] == x):
                  achou = True
                  pos = i
            else:
                  i = i + 1

      if achou:
            return pos
      else:
            return achou
```

Vamos analisar a complexidade da busca sequencial no pior caso, ou seja, quando o elemento a ser buscado encontra-se na última posição do vetor. Por exemplo,

L = [3, 1, 8, 2, 9, 6, 7] e x = 7

Note que o loop executa n - 1 vezes a instrução de incremento no valor de i e uma vez as duas instruções para atualizar os valores de achou e pos.

$T(n) = 2 + n – 1 + 2 = n + 3$

o que resulta em O(n).

A busca binária requer uma lista ordenada de elementos para funcionar. Ela imita o processo que nós utilizamos para procurar uma palavra no dicionário. Como as palavras estão ordenadas, a ideia é abrir o dicionário mais ou menos no meio. Se a palavra que desejamos inicia com uma letra que vem antes, então nós já descartamos toda a metade final do dicionário (não precisamos procurar lá, pois é certeza que a palavra estará na primeira metade.

No algoritmo, temos uma lista com números ordenados. Basicamente, a ideia consiste em acessar o elemento do meio da lista. Se ele for o que desejamos buscar, a busca se encerra. Caso contrário, se o que desejamos é menor que o elemento do meio, a busca é realizada na metade a esquerda. Senão, a busca é realizada na metade a direita. A seguir mostramos um script em Python que implementa a versão recursiva da busca binária.

```
# Função recursiva (ela chama a si própria)
def binary_search(L, x, ini, fim):
    meio = ini + (fim - ini) // 2
    if ini > fim:
        return -1          # elemento não encontrado
    elif L[meio] == x:
        return meio
    elif L[meio] > x:
        print('Buscar na metade inferior')
        return binary_search(L, x, ini, meio-1)
    else:
        print('Buscar na metade superior')
        return binary_search(L, x, meio+1, fim)
```

Uma comparação enter o pior caso da busca sequencial e da busca binária, mostra a significativa diferença entre os métodos. Na busca sequencial, faremos n acessos para encontrar o valor procurado na última posição. Costuma-se dizer que o custo é O(n) (é da ordem de n, ou seja, linear). Na busca binária, como a cada acesso descartamos metade das amostras restantes. Supondo, por motivos de simplificação, que o tamanho do vetor n é uma potência de 2, ou seja, $n = 2^m$, note que:

Acessos		**Descartados**
m = 1	→	n/2
m = 2	→	n/4
m = 3	→	n/8
m = 4	→	n/16

e assim sucessivamente. É possível notar um padrão?

Quantos acessos devemos realizar para que descartemos todo o vetor? Devemos ter $n / 2^m = 1$, o que significa ter $n = 2^m$, o que implica em $m = \log_2 n$, ou seja, temos um custo $O(\log_2 n)$ o que é bem menor do que n quando n cresce muito, pois a função log(n) tem uma curva de crescimento bem mais lento do que a função linear n. Veja que a derivada (taxa de variação) da função linear n é constante e igual a 1 sempre. A derivada da função log(n) é 1/n, ou seja, quando n cresce, a taxa de variação, que é o que controla o crescimento da função, decresce.

Na prática, isso significa que em uma lista com 1024 elementos, a busca sequencial fará no pior caso 1023 acessos até encontrar o elemento desejado. Na busca binária, serão necessários apenas $\log_2 1024 = 10$ acessos, o que corresponde a aproximadamente 1% do necessário na busca sequencial! É uma ganho muito grande.

Porém, na busca binária precisamos gastar um tempo para ordenar a lista! Para isso precisaremos de algoritmos de ordenação, o que é o assunto da nossa próxima aula. Bons estudos e até mais.

Aula 3 - Algoritmos de ordenação de dados

Ser capaz de ordenar os elementos de um conjunto de dados é uma das tarefas básicas mais requisitadas por aplicações computacionais. Como exemplo, podemos citar a busca binária, um algoritmo de busca muito mais eficiente que a simples busca sequencial. Buscar elementos em conjuntos ordenados é bem mais rápido do que em conjuntos desordenados. Existem diversos algoritmos de ordenação, sendo alguns mais eficientes do que outros. Neste curso, iremos apresentar alguns deles: Bubblesort, Selectionsort, Insertionsort, Quicksort e Mergesort.

Bubblesort

O algoritmo *Bubblesort* é uma das abordagens mais simplistas para a ordenação de dados. A ideia básica consiste em percorrer o vetor diversas vezes, em cada passagem fazendo flutuar para o topo da lista (posição mais a direita possível) o maior elemento da sequência. Esse padrão de movimentação lembra a forma como as bolhas em um tanque procuram seu próprio nível, e disso vem o nome do algoritmo (também conhecido como o método bolha)
Embora no melhor caso esse algoritmo necessite de apenas n operações relevantes, onde n representa o número de elementos no vetor, no pior caso são feitas n^2 operações. Portanto, diz-se que a complexidade do método é de ordem quadrática. Por essa razão, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados. A seguir veremos uma implementação em Python desse algoritmo.

```
import numpy as np
import time

# Implementação em Python do algoritmo de ordenação Bubblesort
def BubbleSort(L):
    # Percorre cada elemento da lista L
    for i in range(len(L)-1, 0, -1):
        # Flutua o maior elemento para a posição mais a direita
        for j in range(i):
            if L[j] > L[j+1]:
                L[j], L[j+1] = L[j+1], L[j]

# Início do script
n = 5000
X = list(np.random.random(n))

# Imprime vetor
print('Vetor não ordenado: ')
print(X)
print()

# Aplica Bubblesort
inicio = time.time()
BubbleSort(X)
fim = time.time()

# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

Exemplo: mostre o passo a passo necessário para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem (levar maior elemento para última posição)

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6] em amarelo, não troca
[2, 5, 13, 7, -3, 4, 15, 10, 1, 6] em azul, troca
[2, 5, 7, 13, -3, 4, 15, 10, 1, 6]
[2, 5, 7, -3, 13, 4, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 10, 15, 1, 6]
[2, 5, 7, -3, 4, 13, 10, 1, 15, 6]

[5, 2, 7, -3, 4, 13, 10, 1, 6, 15]

2ª passagem (levar segundo maior para penúltima posição)

[2, 5, 7, -3, 4, 13, 10, 1, 6, 15]
[2, 5, 7, -3, 4, 13, 10, 1, 6, 15]
[2, 5, -3, 7, 4, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 10, 13, 1, 6, 15]
[2, 5, -3, 4, 7, 10, 1, 13, 6, 15]

[2, 5, -3, 4, 7, 10, 1, 6, 13, 15]

3ª passagem (levar terceiro maior para antepenúltima posição)
[2, 5, -3, 4, 7, 10, 1, 6, 13, 15]
[2, -3, 5, 4, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 1, 10, 6, 13, 15]

[2, -3, 4, 5, 7, 1, 6, 10, 13, 15]

4ª passagem

[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 1, 7, 6, 10, 13, 15]

[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]

5ª passagem

[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]

[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]
[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]
[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

6ª passagem

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]
[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]
[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

7ª passagem

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]
[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

8ª passagem

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

9ª passagem

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

Análise da complexidade do Bubblesort

Observando a função definida anteriormente, note que no pior caso o segundo loop vai de zero a i, sendo que na primeira vez i = n – 1, na segunda vez i = n – 2 e até i = 0. Sendo assim, o número de operações é dado por:

T(n) = (n + (n – 1) + (n - 2) + … + 1)

Já vimos na aula anterior que o somatório 1 + 2 + … + n é igual a n(n + 1)/2. Assim, temos:

$$T(n)=\frac{n(n+1)}{2}=\frac{n^2+n}{2}$$

o que nos leva a $O(n^2)$.

No melhor caso, é possível fazer uma pequena modificação no Bubblesort para contar quantas inversões (trocas) ele realiza. Dessa forma, se uma lista já está ordenada e o Bubblesort não realiza nenhuma troca, o algoritmo pode terminar após o primeiro passo. Com essa modificação, se o algoritmo encontra uma lista ordenada, sua complexidade é O(n), pois ele percorre a lista de n

elementos uma única vez. Porém, para fins didáticos, a versão apresentada acima tem complexidade $O(n^2)$ mesmo no melhor caso, pois não faz a checagem de quantas inversões são realizadas.

Finalmente, no caso médio, precisamos calcular a média dos custos entre todos os casos possíveis. O primeiro caso (de menor custo) ocorre quando o laço mais externo realiza uma única iteração e o último caso (de maior custo) ocorre quando o laço externo realiza o número máximo de iterações, isto é, n-1 iterações. Como todos os casos são igualmente prováveis, isto é, todas as diferentes configurações do vetor de entrada têm a mesma probabilidade de ocorrer (distribuição uniforme), então a probabilidade de cada caso é de 1/(n – 1) e a média é dada por:

$$\frac{1}{n-1}\sum_{k=1}^{n-1} f(k)$$

onde f(k) denota o número de execuções do loop mais externo para a k-ésima configuração. Lembrando que quando k = 1 o Bubblesort realiza uma única passagem na lista L, quando k = 2 o Bubblesort realiza duas passagens na lista L, e assim sucessivamente. Dessa forma, podemos definir a função f(k) de maneira a contar o número de trocas em cada configuração possível como:

$$f(k)=\sum_{i=1}^{k}\sum_{j=1}^{i} 1$$

uma vez que o loop mais interno realiza i trocas, sendo que esse número de trocas depende de qual configuração estamos, ou sjea, quanto maior for o k, mais vezes o loop mais interno será executado.

Portanto, podemos escrever:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\left(\sum_{i=1}^{k}\sum_{j=1}^{i} 1\right)$$

Note que o último somatório resulta em i, o que nos permite escrever:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\left(\sum_{i=1}^{k} i\right)$$

Sabemos que o somatório 1 + 2 + 3 + … + k = k(k+1)/2, o que nos leva a:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\frac{k(k+1)}{2}$$

Aplicando a distributiva, temos:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\left(\frac{1}{2}(k^2+k)\right)$$

Como um somatórios de somas é igual as somas dos somatórios, podemos escrever:

$$T(n)=\frac{1}{2(n-1)}\left[\sum_{k=1}^{n-1} k^2+\sum_{k=1}^{n-1} k\right] \quad (*)$$

Vamos chamar o primeiro somatório de A e o segundo de B. O valor de B é facilmente calculado pois sabemos que 1 + 2 + 3 + … + n – 1 = n(n-1)/2. Vamos agora calcular o valor de A, dado por:

$$A=\sum_{k=1}^{n-1} k^2$$

Para isso, iremos utilizar o conceito de soma telescópica. Lembre-se que:

$$(k+1)^3=k^3+3k^2+3k+1$$

de modo que podemos escrever

$$(k+1)^3-k^3=3k^2+3k+1$$

Aplicando somatório de ambos os lados, temos:

$$\sum_{k=1}^{n-1}\left[(k+1)^3-k^3\right]=3\sum_{k=1}^{n-1}k^2+3\sum_{k=1}^{n-1}k+\sum_{k=1}^{n-1}1$$

Note que o somatório do lado esquerdo é uma soma telescópica, ou seja, vale $n^3 – 1$ (é a diferença entre o último elemento e o primeiro, uma vez todos os termos intermediários se cancelam:

$$n^3-1=3\sum_{k=1}^{n-1}k^2+3\frac{n(n-1)}{2}+n-1$$

Dessa forma, isolando o somatório desejado, temos:

$$\sum_{k=1}^{n-1}k^2=\frac{n^3-1-n+1-\frac{3n(n-1)}{2}}{3}=\frac{n^3-n}{3}-\frac{n(n-1)}{2}=\frac{n(n^2-1)}{3}-\frac{n(n-1)}{2}=\frac{n(n-1)(n+1)}{3}-\frac{n(n-1)}{2}$$

Voltando para a equação (*), podemos escrever:

$$T(n)=\frac{1}{2(n-1)}\left[\frac{n(n-1)(n+1)}{3}-\frac{n(n-1)}{2}+\frac{n(n-1)}{2}\right]$$

Note que os dois últimos termos da equação se cancelam pois são idênticos, o que nos leva a:

$$T(n)=\frac{1}{2(n-1)}\left(\frac{n(n-1)(n+1)}{3}\right)$$

Cancelando os termos n – 1, finalmente chegamos em:

$$T(n)=\frac{1}{6}n(n+1)=\frac{1}{6}(n^2+n)$$

o que mostra que a complexidade do caso médio é $O(n^2)$.

Selection sort

A ordenação por seleção é um método baseado em se passar o menor valor do vetor para a primeira posição mais a esquerda disponível, depois o de segundo menor valor para a segunda posição e

assim sucessivamente, com os n – 1 elementos restantes. Esse algoritmo compara a cada iteração um elemento com os demais, visando encontrar o menor. A complexidade desse algoritmo será sempre de ordem quadrática, isto é o número de operações realizadas depende do quadrado do tamanho do vetor de entrada. Algumas vantagens desse método são: é um algoritmo simples de ser implementado, não usa um vetor auxiliar e portanto ocupa pouca memória, é um dos mais velozes para vetores pequenos. Como desvantagens podemos citar o fato de que ele não é muito eficiente para grandes vetores.

```
import numpy as np
import time

# Implementação em Python do algoritmo de ordenação Selectionsort
def SelectionSort(L):
    # Percorre todos os elementos de L
    for i in range(len(L)):
        menor = i
        # Encontra o menor elemento
        for k in range(i+1, len(L)):
            if L[k] < L[menor]:
                menor = k
        # Troca a posição do elemento i com o menor
        L[menor], L[i] = L[i], L[menor]

# Início do script
n = 5000
X = list(np.random.random(n))

# Imprime vetor
print('Vetor não ordenado: ')
print(X)
print()

# Aplica Bubblesort
inicio = time.time()
SelectionSort(X)
fim = time.time()

# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

Análise da complexidade do Selectionsort

Observando a função definida anteriormente, note que no pior caso o segundo loop vai de i +1 até n, sendo que na primeira vez i = 1, na segunda vez i = 2 e até i = n - 1. Sendo assim, o número de operações é dado por:

$$T(n)=\sum_{i=0}^{n-1}\left(2+\sum_{j=i+1}^{n-1}1\right)=\sum_{i=0}^{n-1}2+\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}1$$

Note que:

$$\sum_{i=2}^{5} 1 = (5-2)+1 = 4$$

Então, podemos escrever:

$$T(n) = 2n + \sum_{i=0}^{n-1} (n-1-(i+1)+1) = 2n + \sum_{i=0}^{n-1} (n-i+1)$$

Podemos decompor o somatório em três, de modo que:

$$T(n) = 2n + \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 1 = 2n + n^2 - \frac{n(n-1)}{2} + n = n^2 + 3n - \frac{n^2-n}{2} = \frac{1}{2}n^2 + \frac{7}{2}n$$

o que resulta em $O(n^2)$. Uma desvantagem do algoritmo Selectionsort é que mesmo no melhor caso, para encontrar o menor elemento, devemos percorrer todo o restante do vetor no loop mais interno. Isso significa que, mesmo no melhor caso, a complexidade é $O(n^2)$ Isso implica que no caso médio, a complexidade também é $O(n^2)$.

Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [-3, 2, 13, 7, 5, 4, 15, 10, 1, 6]
2ª passagem: [-3, 2, 13, 7, 5, 4, 15, 10, 1, 6] → [-3, 1, 13, 7, 5, 4, 15, 10, 2, 6]
3ª passagem: [-3, 1, 13, 7, 5, 4, 15, 10, 2, 6] → [-3, 1, 2, 7, 5, 4, 15, 10, 13, 6]
4ª passagem: [-3, 1, 2, 7, 5, 4, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6]
5ª passagem: [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6]
6ª passagem: [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 6, 15, 10, 13, 7]
7ª passagem: [-3, 1, 2, 4, 5, 6, 15, 10, 13, 7] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]
8ª passagem: [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]
9ª passagem: [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

Insertion sort

Insertion sort, ou ordenação por inserção, é o algoritmo de ordenação que, dado um vetor inicial constrói um vetor final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadráticos, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

Podemos fazer uma comparação do Insertion sort com o modo de como algumas pessoas organizam um baralho num jogo de cartas. Imagine que você está jogando cartas. Você está com as cartas na mão e elas estão ordenadas. Você recebe uma nova carta e deve colocá-la na posição correta da sua mão de cartas, de forma que as cartas obedeçam a ordenação.
A cada nova carta adicionada a sua mão de cartas, a nova carta pode ser menor que algumas das cartas que você já tem na mão ou maior, e assim, você começa a comparar a nova carta com todas as cartas na sua mão até encontrar sua posição correta. Você insere a nova carta na posição correta, e, novamente, sua mão é composta de cartas totalmente ordenadas. Então, você recebe outra carta e repete o mesmo procedimento. Então outra carta, e outra, e assim por diante, até você não receber

mais cartas. Esta é a ideia por trás da ordenação por inserção. Percorra as posições do vetor, começando com o índice zero. Cada nova posição é como a nova carta que você recebeu, e você precisa inseri-la no lugar correto no sub-vetor ordenado à esquerda daquela posição.

```
import numpy as np
import time

# Implementação em Python do algoritmo de ordenação Insertionsort
def InsertionSort(L):
    # Percorre cada elemento de L
    for i in range(1, len(L)):
        k = i
        # Insere o pivô na posição correta
        while k > 0 and L[k] < L[k-1]:
            L[k], L[k-1] = L[k-1], L[k]
            k = k - 1

# Início do script
n = 5000
X = list(np.random.random(n))

# Imprime vetor
print('Vetor não ordenado: ')
print(X)
print()

# Aplica Bubblesort
inicio = time.time()
InsertionSort(X)
fim = time.time()

# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

Análise da complexidade do Insertionsort

Observando a função definida anteriormente, note que no pior caso a posição correta do pivô será sempre em k = 0 de modo que o segundo loop vai ter de percorrer todo vetor (de i até 0), sendo que na primeira vez i = 1, na segunda vez i = 2 e até i = n - 1. Sendo assim, o número de operações é dado por:

$$T(n)=\sum_{i=1}^{n-1}\left(1+\sum_{j=0}^{i}2\right)$$

Expandindo os somatórios, temos:

$$T(n)=\sum_{i=1}^{n-1}1+2\sum_{i=1}^{n-1}(i+1)=n-1+2\sum_{i=1}^{n-1}i+2\sum_{i=1}^{n-1}1$$

O valor do primeiro somatório é n(n-1)/2 e o valor do segundo somatório é n – 1, o que nos leva a:

$$T(n)=n-1+2\frac{n(n-1)}{2}+2(n-1)=3(n-1)+n^2-n=n^2+2n-3$$

o que mostra que a complexidade é $O(n^2)$.

Para o melhor caso, note que o pivô sempre está na posição correta, sendo que o loop mais interno não será executado nenhuma vez. Assim temos que dentro do loop mais externo apenas uma instrução será executada, o que nos leva a:

$$T(n)=\sum_{i=1}^{n-1}1=n-1$$

mostrando que a complexidade é linear, ou seja, $O(n)$. Para o caso médio, podemos aplicar uma estratégia muito similar àquela adotada na análise do Bubblesort. A ideia consiste em considerar que todos os casos são igualmente prováveis e calcular uma média de todos eles. Assim, temos:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\left[\sum_{i=1}^{k}\left(1+\sum_{j=0}^{i}2\right)\right]$$

Do pior caso, sabemos que a soma entre colchetes pode ser simplificada, o que nos leva a:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}(k^2+2k-3)$$

Deixaremos o restante dos cálculos para o leitor (basta seguir os mesmos passos algébricos da análise do caso médio do Bubblesort). É possível verificar então que a complexidade é $O(n^2)$.

Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 13, 7, -3, 4, 15, 10, 1, 6]
2ª passagem: [2, 5, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 13, 7, -3, 4, 15, 10, 1, 6]
3ª passagem: [2, 5, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 7, 13, -3, 4, 15, 10, 1, 6]
4ª passagem: [2, 5, 7, 13, -3, 4, 15, 10, 1, 6] → [-3, 2, 5, 7, 13, 4, 15, 10, 1, 6]
5ª passagem: [-3, 2, 5, 7, 13, 4, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6]
6ª passagem: [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6]
7ª passagem: [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 10, 13, 15, 1, 6]
8ª passagem: [-3, 2, 4, 5, 7, 10, 13, 15, 1, 6] → [-3, 1, 2, 4, 5, 7, 10, 13, 15, 6]
9ª passagem: [-3, 1, 2, 4, 5, 7, 10, 13, 15, 6] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

Recursão

Dizemos que uma função é recursiva se ela é definida em termos dela mesma. Em matemática e computação uma classe de objetos ou métodos exibe um comportamento recursivo quando pode ser definido por duas propriedades:

1. Um caso base: condição de término da recursão em que o processo produz uma resposta.

2. Um passo recursivo: um conjunto de regras que reduz todos os outros casos ao caso base.

A série de Fibonacci é um exemplo clássico de recursão, pois:

F(1) = 1 (caso base 1)
F(2) = 1 (caso base 2)

Para todo n > 1, F(n) = F(n - 1) + F(n – 2)

A função recursiva a seguir calcula o n-ésimo termo da sequência de Fibonacci.

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

n = int(input('Enter com o valor de n: '))
resultado = fib(n)

print('Fibonacci(%d) = %d' %(n, resultado))
```

Os dois próximos algoritmos de ordenação que iremos estudar são exemplos de abordagens recursicas, onde a cada passo, temos um subproblema menor para ser resolvido pela mesma função. Por isso, a função é definida em termos de si própria.

Quicksort

O algoritmo Quicksort segue o paradigma conhecido como “Dividir para Conquistar” pois ele quebra o problema de ordenar um vetor em subproblemas menores, mais fáceis e rápidos de serem resolvidos. Primeiramente, o método divide o vetor original em duas partes: os elementos menores que o pivô (tipicamente escolhido como o primeiro ou último elemento do conjunto). O método então ordena essas partes de maneira recursiva. O algoritmo pode ser dividido em 3 passos principais:

1. Escolha do pivô: em geral, o pivô é o primeiro ou último elemento do conjunto.

2. Particionamento: reorganizar o vetor de modo que todos os elementos menores que o pivô apareçam antes dele (a esquerda) e os elementos maiores apareçam após ele (a direita). Ao término dessa etapa o pivô estará em sua posição final (existem várias formas de se fazer essa etapa)

<= c	<= c	<= c	<= c	c	>= c	>=c	>= c	>= c	>=c

3. Ordenação: recursivamente aplicar os passos acima aos sub-vetores produzidos durante o particionamento. O caso limite da recursão é o sub-vetor de tamanho 1, que já está ordenado.
Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1º passo: Definir pivô = 6 (último elemento)

2º passo: Particionar vetor (menores a esquerda e maiores a direita)

[5, 2, -3, 4, 1, 6, 13, 7, 15, 10]

3º passo: Aplicar 1 e 2 recursivamente para as metades

a) 2 metades

Metade 1: [5, 2, -3, 4, 1] → pivô = 1

[-3, 1, 5, 2, 4, 6, 13, 7, 15, 10]

Metade 2: [13, 7, 15, 10] → pivô = 10

[-3, 1, 5, 2, 4, 6, 7, 10, 15, 13]

b) 4 metades

Note que a metade 1 possui um único elemento: [-3] → já está ordenada

Metade 2: [5, 2, 4] → pivô = 4

[-3, 1, 2, 4, 5, 6, 7, 10, 15, 13]

Note que a metade 3 possui apenas um único elemento: [7] → já está ordenadas

Metade 4: [15, 13] → pivô = 13

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

c) 4 metades: Note que cada uma das 4 metades restantes contém um único elemento e portanto já estão ordenadas. Fim.

A seguir veremos uma implementação recursiva em Python para o algoritmo *Quicksort*.

```
# Implementação em Python do algoritmo de ordenação Quicksort
def QuickSort(L):
    if len(L) <= 1:
        return L
    # Pivô é o primeiro elemento da lista pode ser último ou do meio)
    m = L[0]
    # Chamada recursiva
    return  QuickSort([x for x in L if x < m]) + \
                      [x for x in L if x == m] + \
            QuickSort([x for x in L if x > m])
```

No Quicksort há significativas diferenças entre o caso melhor e o pior caso. Veremos primeiramente o pior caso.

a) Pior caso: acontece quando o pivô é sempre o maior ou menor elemento, o que gera partições totalmente desbalanceadas:

Na primeira chamada recursiva, temos uma lista de tamanho n, então para criar as novas listas L, teremos n – 1 elementos na primeira lista, o pivô e uma lista vazia.

Isso nos permite escrever a seguinte relação de recorrência:

$$T(n)=T(n-1)+T(0)+n=T(n-1)+n$$

pois estamos decompondo um problema de tamanho n em um de tamanho zero e outro de tamanho n – 1, mas para realizar a divisão da lista L em sublistas, utilizamos n operações (uma vez que a lista L tem n elementos)

Note que:

T(n) = n + T(n – 1)
T(n - 1) = n - 1 + T(n – 2)
T(n - 2) = n - 2 + T(n – 3)
T(n - 3) = n - 3 + T(n – 4)
…
T(2) = 2 + T(1)
T(1) = 1 + T(0)

onde T(0) = 0.

Somando todas as parcelas, temos o somatório (1 + 2 + 3 + … + n – 1) + n, cuja resposta é:

$$T(n)=\frac{n(n+1)}{2}=\frac{n^2+n}{2}$$

o que nos leva a $O(n^2)$.

b) Melhor caso: acontece quando as partições tem exatamente o mesmo tamanho, ou seja, são n/2 elementos, o pivô e mais n/2 - 1 elementos.

Podemos escrever a seguinte relação de recorrência:

$$T(n)=T\left(\frac{n}{2}\right)+T\left(\frac{n}{2}\right)+n=2T\left(\frac{n}{2}\right)+n$$

pois estamos decompondo um problema de tamanho n em dois problemas de tamanho n/2, mas para realizar a divisão da lista L em sublistas, utilizamos n operações (uma vez que a lista L tem n elementos). Expandindo a recorrência temos:

$$T(n)=2T\left(\frac{n}{2}\right)+n$$
$$T\left(\frac{n}{2}\right)=2T\left(\frac{n}{4}\right)+n$$
$$T\left(\frac{n}{4}\right)=2T\left(\frac{n}{8}\right)+n$$
$$T\left(\frac{n}{8}\right)=2T\left(\frac{n}{16}\right)+n$$

…

Voltando com as substituições, temos:

$$T(n)=2\left[2\left[2\left[2T\left(\frac{n}{16}\right)+n\right]+n\right]+n\right]+n=2^4T\left(\frac{n}{2^4}\right)+4n$$

Generalizando para um valor k arbitrário, podemos escrever:

$$T(n)=2^kT\left(\frac{n}{2^k}\right)+kn$$

Para que tenhamos T(1), é preciso que $n = 2^k$, ou seja, $k=\log_2 n$. Quando $k=\log_2 n$, temos:

$$T(n)=2^{\log_2 n}T(1)+n\log_2 n=nT(1)+n\log_2 n$$

Como T(1) é constante (pois não depende de n), temos finalmente que a complexidade do Quicksort no melhor caso é $O(n\log_2 n)$.

Veremos a seguir que o fato de que o pior caso é $O(n^2)$, não é um problema, pois o caso médio está muito mais próximo do melhor caso do que do pior caso. Para considerar o caso médio, suponha que alternemos as recorrências de melhor caso, M(n), com pior caso, P(n). Então, iniciando com o melhor caso:

$$T(n)=2T\left(\frac{n}{2}\right)+n$$

Expandindo agora o pior caso, temos:

$$T(n)=2\left[T\left(\frac{n}{2}-1\right)+\frac{n}{2}\right]+n$$

Simplificando a expressão acima, podemos escrever:

$$T(n)=2T\left(\frac{n}{2}-1\right)+O(n)$$

que é uma relação de recorrência da mesma forma que a obtida para o melhor caso. A solução da recorrência implica que a complexidade do caso médio é $O(n\log_2 n)$.

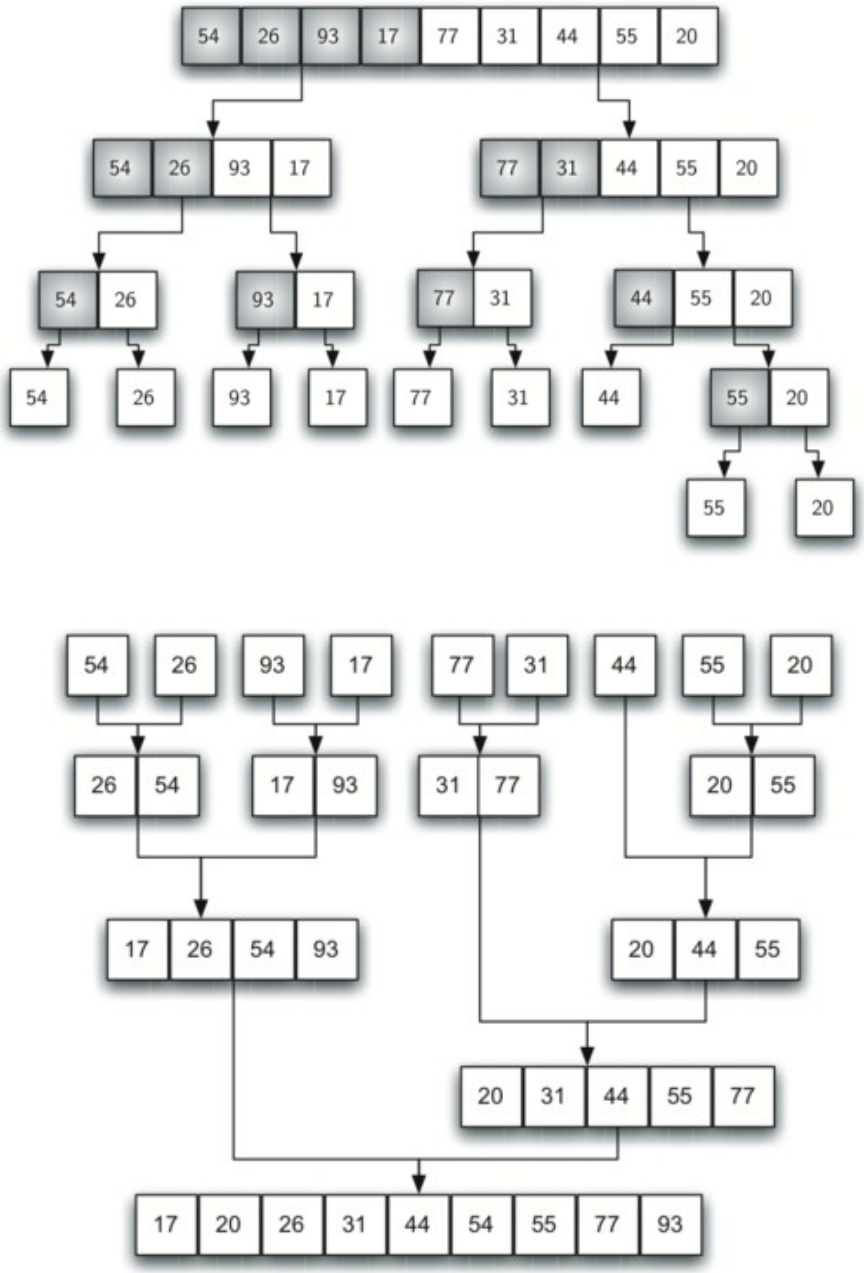
Uma estratégia empírica que, em geral, melhora o desempenho do algoritmo Quicksort consiste em escolher como pivô a mediana entre o primeiro elemento, o elemento do meio e o último elemento.

Mergesort (ordenação por intercalação)

O algoritmo Mergesort utiliza a abordagem Dividir para Conquistar. A ideia básica consiste em dividir o problema em vários subproblemas e resolver esses subproblemas através da recursividade e depois conquistar, o que é feito após todos os subproblemas terem sido resolvidos através da união das resoluções dos subproblemas menores.

Trata-se de um algoritmo recursivo que divide uma lista continuamente pela metade. Se a lista estiver vazia ou tiver um único elemento, ela está ordenada por definição (o caso base). Se a lista tiver mais de um elemento, dividimos a lista e invocamos recursivamente um Mergesort em ambas as metades. Assim que as metades estiverem ordenadas, a operação fundamental, chamada de **intercalação**, é realizada. Intercalar é o processo de pegar duas listas menores ordenadas e combiná-las de modo a formar uma lista nova, única e ordenada.

A figura a seguir ilustra as duas fases principais do algoritmo Mergesort: a divisão e a intercalação.



A seguir apresentamos um código em Python para implementar o algoritmo MergeSort.

```
# Implementação em Python do algoritmo de ordenação MergeSort
def MergeSort(L):

    if len(L) > 1:

        meio = len(L)//2
        LE = L[:meio]   # Lista Esquerda
        LD = L[meio:]   # Lista Direita

        # Aplica recursivamente nas sublistas
        MergeSort(LE)
        MergeSort(LD)

        # Quando volta da recursão inicia aqui!
        i, j, k = 0, 0, 0
        # Faz a intercalação das duas listas (merge)
        while i < len(LE) and j < len(LD):
            if LE[i] < LD[j]:
                L[k] = LE[i]
                i += 1
            else:
                L[k] = LD[j]
                j += 1
            k += 1

        while i < len(LE):
            L[k] = LE[i]
            i += 1
            k += 1

        while j < len(LD):
            L[k] = LD[j]
            j += 1
            k += 1

# Início do script
n = 5000
X = list(np.random.random(n))

# Imprime vetor
print('Vetor não ordenado: ')
print(X)
print()

# Aplica Bubblesort
inicio = time.time()
MergeSort(X)
fim = time.time()

# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

A função MergeSort mostrada acima começa perguntando pelo caso base. Se o tamanho da lista form menor ou igual a um, então já temos uma lista ordenada e nenhum processamento adicional é necessário. Se, por outro lado, o tamanho da lista for maior do que um, então usamos a operação de slice do Python para extrair a metades esquerda e direita. É importante observar que a lista pode não ter um número par de elementos. Isso, contudo, não importa, já que a diferença de tamanho entre as listas será de apenas um elemento.

Quando a função MergeSort retorna da recursão (após a chamada nas metades esquerda, LE, e direita, LD), elas já estão ordenadas. O resto da função (linhas 11-31) é responsável por intercalar as duas listas ordenadas menores em uma lista ordenada maior. Note que a operação de intercalação coloca um item por vez de volta na lista original (`L`) ao tomar repetidamente o menor item das listas ordenadas.

A seguir apresentamos um exemplo ilustrativo passo a passo da aplicação do MergeSort.

Ex: Mostre o passo a passo da ordenação do vetor a seguir pelo algoritmo MergeSort

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

Passo 1: Dividir em subproblemas

1ª divisão: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

meio = 10 // 2 = 5

2ª divisão: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

meio = 5 // 2 = 2

3ª divisão: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

meio = 2 // 2 = 1 ou meio = 3 // 2 = 1

4ª divisão: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

meio = 2 // 2 = 1

Parte 2: Intercalar listas (Merge) – as últimas a serem divididas serão as primeiras a fazer o merge

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

[5, 2, 13, -3, 7, 4, 15, 10, 1, 6]

[2, 5, -3, 7, 13, 4, 15, 1, 6, 10]

[-3, 2, 5, 7, 13, 1, 4, 6, 10, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Análise da complexidade do MergeSort

Os três passos úteis dos algoritmos de dividir para conquistar, ou *divide and conquer*, que se aplicam ao MergeSort são:

1. Dividir: Calcula o ponto médio do sub-arranjo, o que demora um tempo constante O(1);
2. Conquistar: Recursivamente resolve dois subproblemas, cada um de tamanho n/2, o que contribui com T(n/2) + T(n/2) para o tempo de execução;
3. Combinar: Unir os sub-arranjos em um único conjunto ordenado, que leva o tempo O(n);

Assim, podemos escrever a relação de recorrência como:

$$T(n)=\begin{cases} O(1), & se \quad n=1 \\ 2T\left(\frac{n}{2}\right)+O(n), & se \quad n>1 \end{cases}$$

Trata-se da mesma recorrência resolvida no algoritmo Quicksort. Note que expandindo a recorrência, temos:

$$T(n)=2T\left(\frac{n}{2}\right)+n$$
$$T\left(\frac{n}{2}\right)=2T\left(\frac{n}{4}\right)+n$$
$$T\left(\frac{n}{4}\right)=2T\left(\frac{n}{8}\right)+n$$
$$T\left(\frac{n}{8}\right)=2T\left(\frac{n}{16}\right)+n$$

…

Voltando com as substituições, podemos escrever:

$$T(n)=2\left[2\left[2\left[2T\left(\frac{n}{16}\right)+n\right]+n\right]+n\right]+n=2^4T\left(\frac{n}{2^4}\right)+4n$$

Generalizando para um valor k arbitrário, podemos escrever:

$$T(n)=2^kT\left(\frac{n}{2^k}\right)+kn$$

Para que tenhamos T(1), é preciso que $n = 2^k$, ou seja, $k=\log_2 n$. Quando $k=\log_2 n$, temos:

$$T(n)=2^{\log_2 n}T(1)+n\log_2 n=nT(1)+n\log_2 n$$

Como T(1) é O(1), temos que a complexidade do Quicksort no melhor caso é $O(n\log_2 n)$.

Uma das maiores limitações do algoritmo MergeSort é que esse método passa por todo o longo processo mesmo se a lista L já estiver ordenada. Por essa razão, a complexidade de melhor caso é idêntica a complexidade de pior caso, ou seja, $O(n\log n)$.

Para o caso de n muito grande, e listas compostas por números gerados aleatoriamente, pode-se mostrar que o número médio de comparações realizadas pelo algoritmo MergeSort é aproximadamente αn menor que o número de comparações no pior caso, onde:

$$\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645$$

Em resumo, a tabela a seguir faz uma comparação das complexidades dos cinco algoritmos de ordenação apresentados aqui, no pior caso, caso médio e melhor caso.

Algoritmo	**Tempo**		
	Melhor	**Médio**	**Pior**
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Como pode ser visto, fica claro que os dois melhores algoritmos são O QuickSort e o MergeSort.

Existem vários outros algoritmos de ordenação que não apresentaremos aqui. Dentre eles, podemos citar o HeapSort, o ShellSort e RadixSort. Para os interessados em aprender mais sobre o assunto, a internet contém diversos materiais sobre algoritmos de ordenação.

Há várias animações que demonstram o funcionamento dos algoritmos de ordenação. A seguir indicamos alguns links interessantes:

15 sorting algorithms in 6 minutes (Animações sonorizadas muito boas para visualização)
https://www.youtube.com/watch?v=kPRA0W1kECg

Animações passo a passo dos algoritmos
https://visualgo.net/en/sorting

Comparação em tempo real dos algoritmos
https://www.toptal.com/developers/sorting-algorithms

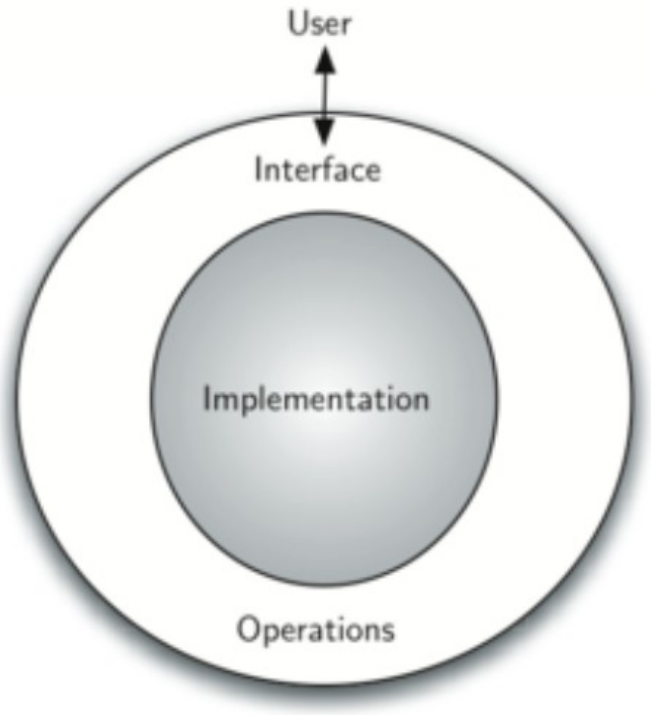
Algoritmos de ordenação como danças
https://www.youtube.com/user/AlgoRythmics

Aula 4 - Programação orientada a objetos em Python

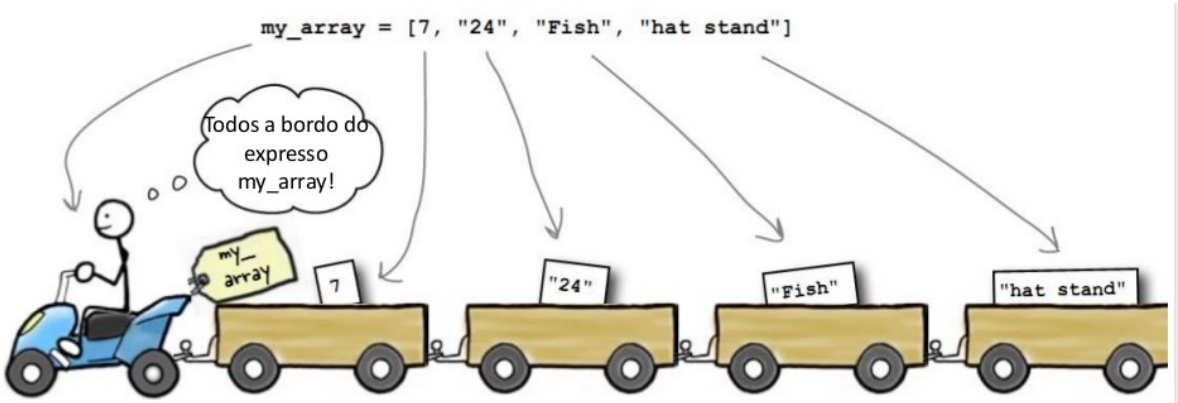
Na programação de computadores, o conceito de abstração é fundamental para o desenvolvimento de software de alto nível. Um exemplo são as funções, que são abstrações de processos. Uma vez que uma função é criada, toda lógica é encapsulada do usuário. Sempre que o programador necessitar, basta invocar a função, sem que ele precise conhecer os detalhes da implementação. Conforme a complexidade dos programas aumenta, torna-se necessário definir abstrações para dados. É para essa finalidade que foram criados os Tipos Abstratos de Dados (TAD's), assunto que iremos explorar em detalhes a seguir.

Tipos Abstratos de Dados (TAD's)

Um Tipo Abstrato de Dados, ou TAD, é um tipo de dados definido pelo programador que especifica um conjunto de variáveis que são utilizadas para armazenar informação e um conjunto de operações bem definidas sobre essas variáveis. TAD's são definidos de forma a ocultar a sua implementação, de modo que um programador deve interagir com as variáveis internas a partir de uma interface, definida em termos do conjunto de operações. A Figura a seguir ilustra essa ideia.



Um exemplo de TAD implementado nativamente pela linguagem Python são as listas. Uma lista em Python consiste basicamente de uma variável composta heterogênea (pode armazenar informações de tipos de dados distintos) utilizada para armazenar as informações mais um conjunto de operações para manipular essa variável.



O trem de dados my_array é uma única variável

Como exemplo de operações encapsuladas em um TAD lista, temos:

`L.append(x)`: adiciona x no final da lista L
`L.pop()`: remove o último elemento da lista L
`L.pop(i)`: remove o elemento da posição i
`L.insert(i, x)`: insere elemento x na posição i
`L.reverse()`: inverte a lista L
`L.sort()`: ordena os elementos da lista L

Note que para o programador, a implementação desses processos fica encapsulada, sendo que não é preciso saber os detalhes, basta conhecer a interface das funções, ou seja, quais os parâmetros necessários para invocá-las. Por exemplo, na função L.insert(i, x) o primeiro parâmetro deve ser o índice da posição do elemento na lista e o segundo parâmetro deve ser o valor a ser armazenado.

Há uma diferença entre os processos encapsulados em um TAD e funções genéricas. Por exemplo, ao trabalharmos com uma lista L, em diversas ocasiões desejamos saber quantos elementos estão no momento em L. Podemos utilizar a função

```
len(L)
```

para retornar essa informação. Porém, a função len() não está encapsulada na definição do TAD. Trata-se de uma função genérica, como uma macro da linguagem, uma vez que ela opera não somente sobre listas, mas outros tipos de coleções também (como conjuntos, dicionários,…)

Outro exemplo de TAD que podemos citar são os vetores definidos pelo pacote Numpy. Em Python, vetores são variáveis compostas homogêneas (pois todos elementos devem ser do mesmo tipo de dados: int ou float), com diversas funções encapsuladas. Algumas das principais são listadas aqui:

`v.max()` - retorna o maior elemento do vetor
`v.min()` - retorna o menor elemento do vetor
`v.argmax()` - retorna o índice do maior elemento do vetor
`v.argmin()` - retorna o índice do menor elemento do vetor
`v.sum()` - retorna a soma dos elementos do vetor
`v.mean()` - retorna a média dos elementos do vetor
`v.prod()` - retorna o produto dos elementos do vetor
`v.T` – retorna o vetor transposto
`v.clip(a, b)` – o que é menor que a vira a e o que é maior que b vira b
`v.shape()` - retorna as dimensões do vetor/matriz

Existem diversas vantagens de se trabalhar com TAD's:

1. Foco na resolução do problema e não nos detalhes de implementação
2. Redução de erros pelo encapsulamento de código validado
3. Correção de bugs e manutenção de código (podemos modificar a parte interna de um TAD sem se preocupar com o programa que utiliza o TAD)
4. Redução da complexidade no desenvolvimento de software, pois é mais fácil dividir um programa muito extenso em pequenos módulos separados de modo que times possam trabalhar de maneira independente

Um TAD é uma abstração, mas sua implementação em uma linguagem de programação é chamada de classe. Uma classe é composta por um conjunto de atributos, que são as variáveis internas utilizadas para armazenar informações, e um conjunto de métodos, que são as operações (funções) utilizadas para processar seus atributos (variáveis internas). Uma instância de uma classe é o que chamamos de objeto.

A ideia desse conceito é bastante simples. Por exemplo, podemos definir uma classe Carro, com os seguintes atributos: nome, marca, cor, ano, km e valor. Podemos criar alguns métodos, como por exemplo: muda_cor(nova_cor), muda_valor(novo_valor), verifica_marca(), verifica_km(), etc. Quando definimos um objeto específico da classe Carro, temos uma instância dessa classe. Por exemplo, um objeto dessa classe poderia ter as seguintes informações:

nome = 'Onix'
marca = 'Chevrolet'
cor = 'Prata'
ano = 2020
km = 20000
valor = 55.600,00

Em resumo, a classe define a implementação do TAD, enquanto que o objeto é a variável alocada na memória, que representa uma instância específica da classe.

Classes em Python

Iremos criar nossa primeira classe em Python utilizando como exemplo, o TAD que define uma fração matemática. Lembe-se que toda fração é composta por dois números números: um numerador e um denominador. O numerador pode assumir qualquer valor, mas o denominador não pode ser zero.

Para que um objeto da classe Fracao possa ser criado, é necessário definirmos um método construtor, que é a função chamada toda vez que um novo objeto for instanciado. Iremos apresentar a segui a definição da Fracao com dois métodos: o construtor, que em Python deve se chamar __init__ e receber como parâmetro o próprio objeto, denominado de self, e a função show() que imprime na tela a fração em forma de string.

```
class Fracao:
     # Construtor (usado para instanciar novos objetos)
     def __init__(self, numerador, denominador):
          self.num = numerador
          self.den = denominador
     # Imprime fração na tela como string
     def show(self):
          print('%s/%s' %(self.num, self.den))
```

Dessa forma, ao carregar o arquivo .py no ambiente, podemos criar um objeto da classe Fracao e imprimir na tela como:

```
frac = Fracao(3, 5)

frac.show()
```

Note que se usarmos o comando print diretamente objeto, não iremos ver o conteúdo de suas variáveis:

```
print(frac)

<__main__.Fraction object at 0x40bce9ac>
```

O comando print retorna o endereço na memória dessa instância específica da classe, ou seja, desse objeto. Há uma maneira de dizer para o interpretador Python que quando fizermos uma referência ao objeto dentro de um comando print, desejamos imprimir alguma informação interna como texto. Trata-se do método __str__, que nesse caso pode ser definido como:

```
def __str__(self):
     return str(self.num) + "/" + str(self.den)
```

Na verdade, estamos fazendo uma sobrecarga na função __str__ que já existe por padrão em todo objeto criado em Python. Porém, por padrão, essa função imprime o endereço de memória do objeto. Ao redefini-la, podemos imprimir as informações internas da maneira que desejarmos. A classe então pode ser definida como:

```
class Fracao:

     # Construtor (usado para instanciar novos objetos)
     def __init__(self, numerador, denominador):
          self.num = numerador
          self.den = denominador

     # Imprime fração na tela como string
     def show(self):
          print('%s/%s' %(self.num, self.den))

     # Imprime fração na tela como string pelo comando print()
     def __str__(self):
          # Devemos retornar a string que será exibida no print()
          return str(self.num) + "/" + str(self.den)
```

Vamos tentar calcular o produto entre duas matrizes com o operador *. Note que se executarmos:

```
f1 = Fraction(1,4)
f2 = Fraction(1,2)
f1 * f2
Traceback (most recent call last):
File "<pyshell#26>", line 1, in <module>
f1 * f2
TypeError: unsupported operand type(s) for +: 'Fraction' and
'Fraction'
```

veremos uma mensagem de erro. Isso porque precisamos dizer ao interpretador Python como calcular o produto entre duas frações com o operador *. Para isso, devemos criar o método __mul__ Por exemplo, utilizar `f1 * f2` será equivalente a chamar `f1.__add__(f2)`.

```
# Implementa a multiplicação de duas frações
def __mul__(self, other):
     novo_num = self.num * other.num
     novo_den = self.den * other.den
     return Fracao(novo_num, novo_den)
```

Para testar o método, basta executar o arquivo .py em que se encontra a definição da classe Fracao, para carregar as definições na memória e digitar os comandos no modo interativo:

```
f1 = Fracao(3, 5)
print(f1)
f2 = Fracao(2, 3)
print(f2)
f3 = f1 * f2
print(f3)
```

Vamos agora implementar uma função para calcular a soma de duas frações. Da mesma forma que fizemos com a multiplicação, podemos criar a função __add__ para sobrecarregar o operador +. Note que matematicamente a definição da soma é dada por:

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+bc}{bd}$$

Dessa forma, a implementação em Python da função fica:

```
def __add__(self, other):
     novo_num = self.num * other.den + self.den*other_fraction.num
     novo_den = self.den * other_fraction.den
     return Fracao(novo_num, novo_den)
```

Lembre-se de executar o script para carregar as alterações feitas na classe para a memória.

```
f1 = Fracao(1, 4)
print(f1)
1/4

f2 = Fracao(1, 2)
print(f2)
1/2

f3 = f1 + f2
print(f3)
6/8
```

Note que o resultado está correto, mas a fração não encontra-se em sua forma simplificada. Se dividirmos tanto o numerador quanto o denominador pelo MDC (máximo divisor comum) entre eles, teremos a fração em sua forma simplificada. Vamos criar um método chamado simplifica, que utiliza o algoritmos de Euclides para encontrar o MDC. O algoritmo de Euclides nos diz que o MDC entre dois inteiros m e n é o próprio n se n divide m. Caso contrário, a resposta deve ser o

MDC de n e o resto da divisão de m por n. A função a seguir implementa a simplificação de uma fração utilizando essa ideia:

```
# Implementa uma função para simplificar uma fração
def simplifica(self):
     m = self.num
     n = self.den
     while n > 0:
          m, n = n, m % n
     mdc = m
     return Fracao(self.num//mdc, self.den//mdc)
```

Executando os seguintes comandos, podemos ver que a fração de fato é simplificada.

```
f1 = Fracao(1, 4)
f2 = Fracao(1, 2)
f3 = f1 + f2
print(f3)
6/8

print(f3.simplifica())
3/4
```

Como podemos verificar se duas frações são iguais? Basta checarmos se as suas formas simplificadas são idênticas, ou seja, os dois numeradores devem ser iguais e os denominadores também devem ser iguais. Para que possamos utilizar o operador == para comparar frações, temos que definir o nome da função como __eq__ (sobrecarga do operador ==).

```
# Implementa uma função para verificar se duas frações são iguais
def __eq__(self, other):
     f1 = self.simplifica()
     f2 = other.simplifica()
     return ((f1.num == f2.num) and (f1.den == f2.den))
```

Testando a função, podemos ver que ela funciona corretamente:

```
f1 = Fracao(1, 2)
f2 = Fracao(2, 4)

f1 == f2
True

f3 = Fracao(3, 5)

f1 == f3
False

f2 == f3
False
```

Por fim, a definição completa da classe Fração é listada a seguir.

```
class Fracao:

     # Construtor (usado para instanciar novos objetos)
     def __init__(self, numerador, denominador):
          # É aqui que criamos todas as variáveis que definem o TAD
          self.num = numerador
          self.den = denominador

     # Imprime fração na tela como string
     def show(self):
          print('%s/%s' %(self.num, self.den))

     # Imprime fração na tela como string usando comando print()
     def __str__(self):
          # Devemos retornar a string que será exibida no print()
          return str(self.num) + "/" + str(self.den)

     # Implementa a multiplicação de duas frações com operador *
     def __mul__(self, other):
          novo_num = self.num * other.num
          novo_den = self.den * other.den
          return Fracao(novo_num, novo_den)

     # Implementa a soma de duas frações com operador +
     def __add__(self, other):
          novo_num = self.num * other.den + self.den*other.num
          novo_den = self.den * other.den
          return Fracao(novo_num, novo_den)

     # Implementa uma função para simplificar uma fração
     def simplifica(self):
          m = self.num
          n = self.den
          while n > 0:
               m, n = n, m % n
          mdc = m
          return Fracao(self.num//mdc, self.den//mdc)

     # Implementa uma função para verificar se duas frações são iguais
     def __eq__(self, other):
          f1 = self.simplifica()
          f2 = other.simplifica()
          return ((f1.num == f2.num) and (f1.den == f2.den))
```

Podemos continuar criando métodos para operar sobre frações, como por exemplo, subtração de duas frações, inverter uma fração, elevar uma fração a uma potência, calcular a raiz quadrada de uma fração, racionalizar uma fração (tornar o denominador inteiro), etc. Não iremos fazer isso aqui, mas aos interessados, é um bom exercício para praticar a programação orientada a objetos na linguagem Python.

Antes de passarmos para o próximo exemplo, uma observação importante: no exemplo anterior, nada impede que alguém acesse as variáveis internas da classe (num e den) de maneira direta, pois por padrão essas informações são públicas em Python. Se desejarmos torná-las privadas, ou seja, elas só podem ser acessadas diretamente a partir de um método interno a classe, devemos adicionar o prefixo __ (dois underscores antes do nome da variável). Em aplicações de pequeno porte, como

as que veremos no curso, não iremos nos preocupar muito com essa questão, pois podemos deixar todas as variáveis públicas. Porém, ao tornar todas as informações públicas, é mais fácil ocorrer acessos de fora, no sentido de que manter a consistência interna dos dados torna-se bem mais complicado. No exemplo a seguir, adotaremos atributos privados. Para isso, devemos criar métodos get e set, para serem interfaces que permitem ao desenvolvedor acessar as variáveis privadas. O programador pode assim controlar quais tipos de valores podem ser atribuídos as variáveis internas, o que é útil para evitar bugs e problemas indesejados no futuro. A seguir iremos fazer alguns exercícios que envolvem outros exemplos numéricos de programação orientada a objetos. Começaremos com uma classe para equações do segundo grau.

Fórmula de Bhaskara: Seja uma equação do segundo grau $a x^2+b x+c=0$, com a, b e c números reais arbitrários. Então, as soluções x_1 e x_2 são dadas por:

$$x_1=\frac{-b-\sqrt{b^2-4 a c}}{2 a} \qquad \text{e} \qquad x_2=\frac{-b+\sqrt{b^2-4 ac}}{2 a}$$

Prova:

1. Partindo da equação do segundo grau $a x^2+b x+c=0$, podemos dividir tudo por a (pois a não pode ser zero) e isolar os termos em x, chegando em:

$$x^2+\frac{b}{a} x=-\frac{c}{a}$$

2. Note que podemos completar o quadrado, adicionando $\frac{b^2}{4 a^2}$ em ambos os lados:

$$x^2+\frac{b}{a} x+\frac{b^2}{4 a^2}=\frac{b^2}{4 a^2}-\frac{c}{a}$$

3. Simplificando a expressão acima, chegamos em:

$$\left(x+\frac{b}{2 a}\right)^2=\frac{b^2-4 ac}{4 a^2}$$

4. Aplicando a raiz quadrada, temos:

$$x+\frac{b}{2 a}=\pm \frac{\sqrt{b^2-4 a c}}{2 a}$$

5. Isolando x, chegamos finalmente a:

$$x=\frac{-b \pm \sqrt{b^2-4 a c}}{2 a}$$

Ex: Implemente uma classe em Python para representar uma equação do segundo grau. Ela deve ter como atributos três números reais - a, b e c - que denotam os coeficientes da equação quadrática, delta, x1 e x2. Inicialmente, delta, x1 e x2 recebem o valor nan (que significa Not a Number).

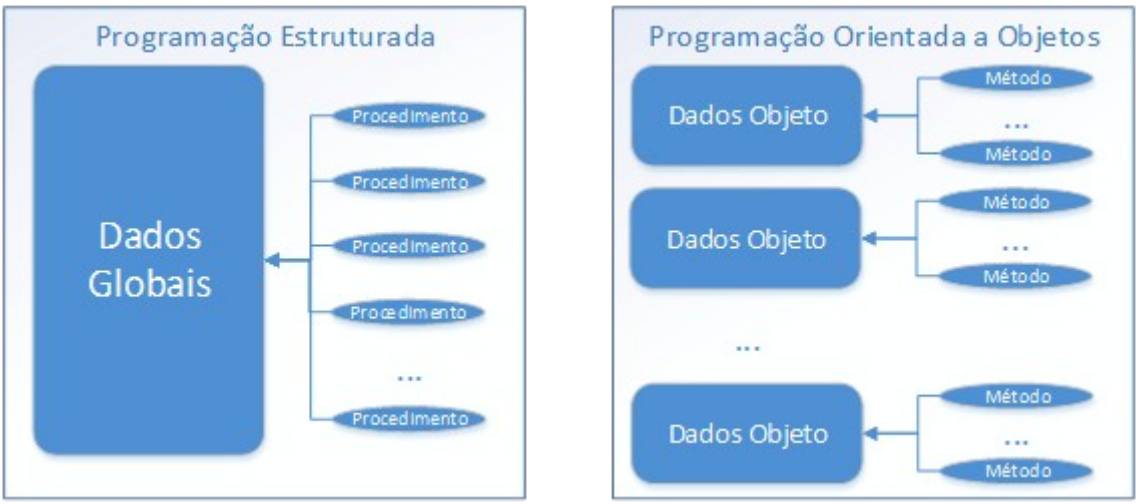
```
from math import sqrt
from math import isnan

class Equacao_Quadratica:

      # Construtor (usado para instanciar novos objetos)
      def __init__(self, a, b, c):
            # __ indica que variável é privada (só visível dentro da classe)
            self.__a = float(a)
            self.__b = float(b)
            self.__c = float(c)
            self.__delta = float('nan')
            self.__x1 = float('nan')
            self.__x2 = float('nan')

      # Obtém valor de a
      def getA(self):
            return self.__a

      # Obtém valor de b
      def getB(self):
            return self.__b

      # Obtém valor de c
      def getA(self):
            return self.__c

      # Obtém valor de delta
      def getDelta(self):
            return self.__delta

      # Atribui valor para a
      def setA(self, a):
            self.__a = float(a)

      # Atribui valor para b
      def setB(self, b):
            self.__b = float(b)

      # Atribui valor para c
      def setC(self, c):
            self.__c = float(c)
      # Retorna string para imprimir equação com comando print
      def __str__(self):
            return str(self.__a)+'x^2 + '+str(self.__b)+'x +'+ str(self.__c)

      # Verifica se é equação do segundo grau (a não é zero)
      def eh_quadratica(self):
            if self.__a != 0:
                  return True

      # Calcula o valor de delta
      def calcula_delta(self):
            self.__delta = self.__b**2 - 4 * self.__a * self.__c

      # Raíz real: verifica se a equação possui raízes reais
      def raiz_real(self):
            if self.__delta >= 0:
                  return True
```

```
      # Resolve equação do 2o grau
      def resolve(self):
            # Copiando variaveis para código menor
            a = self.__a
            b = self.__b
            # Verifica se é equação quadrática
            if self.eh_quadratica():
                  self.calcula_delta()
                  delta = self.__delta
                  # Verifica se as raízes são reais
                  if self.raiz_real():
                        self.__x1 = (-b-sqrt(delta))/(2*a)
                        self.__x2 = (-b+sqrt(delta))/(2*a)
                        return (self.__x1, self.__x2)
                  else:
                        print('A equação não admite raízes reais')
            else:
                  print('A equação não é do segundo grau (coef. a = 0)')


if __name__ == '__main__':
      # Cria equação do segundo grau
      equacao = Equacao_Quadratica(1, -5, 6)
      # Note que se tentarmos acessar o valor de a diretamente
      # ocasionará um erro, pois variável é privada
      # print(equacao.__a)
      # Imprime na tela
      print(equacao)
      # Resolve a equação utilizando a fórmula de Bhaskara
      print(equacao.resolve())
```

Podemos renomear as variáveis removendo o prefixo __ que o os métodos continuarão funcionando normalmente, porém todas as variáveis irão se tornar públicas e poderão ser acessadas diretamente por qualquer usuário no ambiente Python. A classe Equacao_Quadratica define um TAD para a representação e resolução de uma equação do segundo grau. Mais adiante veremos métodos para resolver equações arbitrárias, polinomiais ou não, através de algoritmos para encontrar o zero de funções na seção de aplicações matemáticas. Em particular, definiremos uma classe chamada RootFinder, que representa uma função arbitrária, e sua derivada, juntamente com os métodos de Newton e da Secante. Por hora, nos concentraremos nos princípios básicos da programação orientada a objetos.

Os princípios da programação orientada a objetos

A seguir iremos discutir um pouco mais sobre os princípios da programação orientada a objetos (POO). Na programação estruturada, os dados a serem manipulados são globais e diversas funções operam sobre eles. Na orientação a objetos, como cada objeto tem seus próprios métodos, eles são aplicados somente aos dados daquele objeto. O diagrama a seguir ilustra essa diferença fundamental entre os paradigmas.



FONTE:https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264

Os quatro conceitos fundamentais da programação orientada a objetos (POO) são:

1. Abstração: a ideia é que uma classe seja a implementação computacional de um TAD, com um conjunto de variáveis que representam o estado interno e métodos que operam sobre esses dados

2. Encapsulamento: consiste em ocultar as variáveis que armazenam as informações internas dos objetos, tornando-as acessíveis apenas através dos métodos. Assim, cria-se uma espécie de caixa preta com a qual podemos interagir a partir de suas interfaces.

3. Herança: é basicamente um mecanismo da POO que permite criar novas classes, mais especializadas, a partir de classes mais gerias já existentes. Essa característica é muito útil, pois promove um grande reaproveitamento de código. Por exemplo, podemos definir uma superclasse Pessoa, que possui os atributos, nome, peso, altura e data de nascimento. Em seguida, podemos criar uma subclasse Funcionário, que herda os atributos e métodos já existentes em uma pessoa, e adiciona novos atributos como cargo, salário e ano de admissão, além de métodos como receber_abono(), receber_promocao(), etc...

4. Polimorfismo: é o princípio pelo qual duas ou mais classes derivadas da mesma superclasse podem invocar métodos que têm a mesma assinatura, mas comportamentos distintos. Em outras palavras, consiste na alteração do funcionamento interno de um método herdado de um objeto pai. Isso significa que um método com o mesmo nome em duas classes, pode ser definido de maneira diferente em cada uma delas. É a ideia da sobrecarga dos operadores que vimos no exemplo da classe Fracao, em que utilizamos o operador + e * de forma diferente do que eles funcionam com objetos da classe int ou float.

Para ilustrar os conceitos de herança e polimorfismo, veremos alguns exemplos práticos de como implementá-los em Python a seguir.

Inicialmente, vamos definir uma classe CreditCard, que cria um cartão de crédito com base nos atributos: cliente, banco, conta e limite. Além disso, criamos os métodos get() para acessar as variáveis (que são privadas), além dos métodos compra, que adiciona um valor a fatura do cartão e pagamento, que remove um valor da fatura do cartão.

```
class CreditCard:

     # Construtor (usado para instanciar novos objetos)
     def __init__(self, cliente, banco, conta, limite):
          self._cliente = cliente
          self._banco = banco
          self._conta = conta
          self._limite = limite
          self._fatura = 0   # Valor da fatura sempre inicia com zero
          print('Cartão criado com sucesso')

     def imprime_dados(self):
          print('Cliente: %s' %self._cliente)
          print('Banco: %s' %self._banco)
          print('Conta: %s' %self._conta)
          print('Limite: %s' %self._limite)
          print()

     # Obtém cliente
     def get_cliente(self):
          return self._cliente

     # Obtém banco
     def get_banco(self):
          return self._banco

     # Obtém conta
     def get_conta(self):
          return self._conta

     # Obtém limite
     def get_limite(self):
          return self._limite

     # Obtém fatura
     def get_fatura(self):
          return self._fatura

     # Função que realiza compra (lança valor)
     def compra(self, preco):
          if preco + self._fatura > self._limite:
               print('Limite insuficiente')
               return False
          else:
               self._fatura += preco
               print('Compra de %.3f reais realizada' %preco)
               return True

     # Função que realiza pagamento de fatura
     def pagamento(self, valor):
          if valor <= self._fatura:
               self._fatura -= valor
               print('Pagamento de %.3f reais da fatura' %valor)
```

```
# Esse comando diz ao interpretador se estamos executando o script
# Se for uma execução, ele entra aqui e prossegue
# Caso contrário, se eu incluo esse arquivo usando import, ele ignora
if __name__ == '__main__':
     # Instancia objeto e faz operações
     cartao = CreditCard('Alexandre', 'BB', '11432-5', 1000)
     cartao.imprime_dados()
     print(cartao.get_fatura())
     cartao.compra(250)
     cartao.compra(100)
     cartao.compra(200)
     print(cartao.get_fatura())
     cartao.pagamento(500)
     print(cartao.get_fatura())
```

Para ilustrar o conceito de herança, iremos criar uma subclasse chamada PredatoryCreditCard, que além dos atributos originais, contém uma taxa de juros anual, além de um método adicional processa_mes(), que aplica os juros no final de cada mês. Além disso, para ilustra o conceito de polimorfismo, o método compra será modificado para aplicar uma penalização de 10 reais sempre que o valor da compra ultrapassar o limite de 1000 reais. Note que na definição da classe CreditCard, os atributos possuem um único underscore (_) como prefixo. Isso indica ao Python, que essas informações são protegidas (nem públicas, nem privadas). Variáveis protegidas são visíveis não apenas dentro da classe base (superclasse), mas também dentro de todas as suas subclasses, ou seja, aquelas classes mais especializadas que herdam da superclasse. Iremos considerar juros próximos aos cobrados no Brasil, cerca de 300% ao ano!

```
from CreditCard import CreditCard

# Essa declaração diz que estamos herdando da superclasse CreditCard
class PredatoryCreditCard(CreditCard):

     # Construtor (usado para instanciar novos objetos)
     def __init__(self, cliente, banco, conta, limite, juros):
          super().__init__(cliente, banco, conta, limite)
          self._juros = juros

     def imprime_dados(self):
          print('Cliente: %s' %self._cliente)
          print('Banco: %s' %self._banco)
          print('Conta: %s' %self._conta)
          print('Limite: %.2f' %self._limite)
          print('Juros/ano: %.2f' %self._juros)
          print()

     # As funções get são herdadas da superclasse

     # Função que realiza compra (lança valor)
     def compra(self, preco):
          sucesso = super().compra(preco)
          if not sucesso:
               self._fatura += 10
               print('Multa por ultrapassar limite: + R$ 10')
          return sucesso
```

```
     # Função que aplica juros ao final do mês
     def processa_mes(self):
          if self._fatura > 0:
               juros_mes = (1 + self._juros)**(1/12)  # juros no mês
               self._fatura *= juros_mes
               print('Fatura corrigida = %f' %self._fatura)

# Esse comando diz ao interpretador se estamos executando o script
if __name__ == '__main__':
     # Instancia objeto e faz operações
     cartao = PredatoryCreditCard('Fulano','HSBC','99372-5', 500, 300)
     cartao.imprime_dados()
     print(cartao.get_fatura())
     cartao.compra(250)
     cartao.compra(100)
     print(cartao.get_fatura())
     cartao.processa_mes()
     cartao.get_fatura()
     cartao.compra(500)
     cartao.get_fatura()
     cartao.pagamento(200)
     print(cartao.get_fatura())
```

Aplicações matemáticas

Nesta seção, mostraremos algumas implementações baseadas em programação orientada a objetos (POO) utilizando a linguagem Python em aplicações matemáticas. Iniciaremos com uma aplicação para implementar sequências matemáticas como progressões aritméticas e geométricas.

Sequências e recorrências

Recorrências são sequências matemáticas que obedecem a uma lei de formação, ou seja, o próximo elemento da série é uma função de um ou mais elementos anteriores. Existem diversas recorrências que surgem naturalmente na matemática e outras ciências exatas como a física e a computação. Um dos exemplos mais conhecidos de recorrência é a sequência de Fibonacci, em que dados os dois primeiros termos iguais a 1, cada novo termo é computado pela soma dos 2 termos anteriores.

Def: Recorrência
Uma recorrência é uma regra que nos permite calcular um termo qualquer de uma sequência em função de termos anteriores.

Duas sequências de extrema importância na matemática são progressões aritméticas e geométricas.

Def: Progressão aritmética
Uma progressão aritmética é uma recorrência em que:

$$T_{n+1} = T_n + r$$
$$T_1 = a$$

onde a é o primeiro elemento da sequência e r é a razão. Sendo assim, uma P. A. é da forma:

S = (a, a+r, a+2r, a+3r, …, a+(n-1)r, …)

Def: Progressão geométrica
Uma progressão geométrica é uma recorrência em que:

$$T_{n+1} = qT_n$$
$$T_1 = a$$

onde a é o primeiro elemento da sequência e q é a razão. Sendo assim, uma P. G. é da forma:

S = (a, aq, aq^2, aq^3, …, aq^{n-1}, …)

Def: Sequência de Fibonacci
Uma das primeiras aplicações da sequência de Fibonacci foi no estudo de populações de coelhos. A ideia era modelar o número de indivíduos ao longo dos meses. A ideia era bastante simples: por mês, cada casal adulto de coelhos gera um casal de filhotes. Denota-se por c um casal de filhotes e por C um casal de adultos. De um mês para o outro, um casal de filhotes passa a ser um casal de adultos. Suponha que iniciemos com um casal de filhotes:

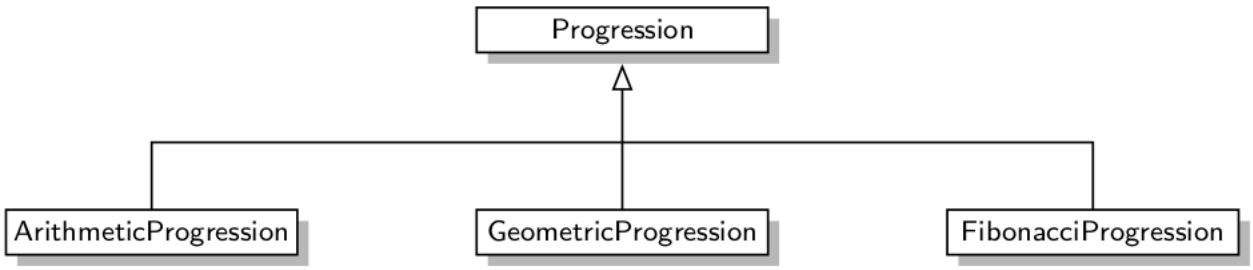
c → C → Cc → CCc → CCCcc → CCCCCccc → CCCCCCCCccccc → ...
1 → 1 → 2 → 3 → 5 → 8 → 13 → ...

Quantos casais existirão no n-ésimo mês?

A recorrência é bastante intuitiva. A expressão matemática é dada por:

$$F_{n+2} = F_{n+1} + F_n$$, com $F_1 = 1$ e $F_2 = 1$

Iremos criar uma superclasse Progression que gera todo o conjuntos dos números naturais: 0, 1, 2, 3 ,… etc. Em seguida, iremos criar 3 subclasses, uma para cada sequência específica: progressão aritmética, progressão geométrica e sequência de Fibonacci, conforme ilustra a figura a seguir.



Para essa aplicação em particular, iremos construir um tipo de classe especial: um iterador. Toda classe que implementa um iterador deve conter um método __next__, que retorna o próximo da coleção ou gera uma exceção do tipo StopIteration, que indica que não há mais elementos. A seguir apresentamos a implementação da superclasse Progression.

```
class Progression:

     # Construtor (usado para instanciar novos objetos)
     def __init__(self, inicio):
          self._valor = inicio       # Função

     # Calcula próximo elemento da sequência
     # prefixo _ apenas indica que iremos modificar
     # essa função nas subclasses
```

```
    def _avancar(self):
        # Por convenção, se valor == None, fim da sequência
        self._valor += 1

    # Retorna o próximo elemento
    def __next__(self):
        if self._valor is None:
            raise StopIteration() # chegamos ao fim
        else:
            resposta = self._valor
            self._avancar()      # atualiza valor corrente
            return resposta

    # Toda classe do tipo iterador tem que conter
    # o método __iter__ (indica que é um iterador)
    def __iter__(self):
        # Por convenção, ietrador retorna si próprio
        return self

    # Imprime os n primeiros elementos da sequência
    def gera_sequencia(self, n):
        L = []
        for j in range(n):
            L.append(next(self))
        return L

if __name__ == '__main__':
    # Instancia objeto
    sequencia = Progression(0)
    print(sequencia.gera_sequencia(20))
```

A execução do código acima mostra o seguinte resultado:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Iremos agora definir uma subclasse que implementa uma progressão aritmética. Note que o único método que precisará ser modificado é o _avancar, que aplica a regra para geração do próximo termo da PA. Essa é a vantagem da herança, que permite um grande reuso de código.

```
# from nome do arquivo import nome da classe
from Progression import Progression

# Herança a partir da superclasse Progression
class ArithmeticProgression(Progression):

    # Construtor (usado para instanciar novos objetos)
    def __init__(self, inicio, razao):
        super().__init__(inicio)
        self._razao = razao

    # Mesmo nome, mas comportaemto diferente da superclasse
    # O conceito de polimorfismo sendo implementado na prática
    # Sobrecarga da função herdada da superclasse (classe pai)
    def _avancar(self):
        # Por convenção, se valor == None, fim da sequência
        self._valor += self._razao
```

```
if __name__ == '__main__':
     # Instancia objeto
     a0 = int(input('Entre com o valor de a0: '))
     razao = int(input('Entre com o valor da razão: '))
     n = int(input('Enter com o número de termos: '))
     PA = ArithmeticProgression(a0, razao)
     print(PA.gera_sequencia(n))
```

Note que utilizamos o construtor da classe pai, adicionando apenas a variável razão, que armazena o incremento dado a cada passo na PA. Note que estamos utilizando aqui o conceito de polimorfismo para sobrecarregar o método _avancar herdado da classe pai. É o mesmo método, mas com comportamentos distintos em cada classe.

A classe definida a seguir mostra a implementação de uma progressão geométrica. Assim como a PA, iremos herdar da classe Progression.

```
# from nome do arquivo import nome da classe
from Progression import Progression

# Herança a partir da superclasse Progression
class GeometricProgression(Progression):

     # Construtor (usado para instanciar novos objetos)
     def __init__(self, inicio, razao):
          super().__init__(inicio)
          self._razao = razao

     # Mesmo nome, mas comportaemto diferente da superclasse
     # O conceito de polimorfismo sendo implementado na prática
     # Sobrecarga da função herdada da superclasse (classe pai)
     def _avancar(self):
          # Por convenção, se valor == None, fim da sequência
          self._valor *= self._razao

if __name__ == '__main__':
     # Instancia objeto
     a0 = int(input('Entre com o valor de a0: '))
     razao = float(input('Entre com o valor da razão: '))
     n = int(input('Enter com o número de termos: '))
     PG = GeometricProgression(a0, razao)
     print(PG.gera_sequencia(n))
```

A classe definida a seguir mostra a implementação de uma sequencia de Fibonacci, também herdade da classe pai Progression (superclasse).

```
# from nome do arquivo import nome da classe
from Progression import Progression

# Herança a partir da superclasse Progression
class FibonacciProgression(Progression):

     # Construtor (usado para instanciar novos objetos)
     def __init__(self, primeiro=0, segundo=1):
          super().__init__(segundo)
          self._previo = primeiro
```

```
     # Mesmo nome, mas comportaemto diferente da superclasse
     # O conceito de polimorfismo sendo implementado na prática
     # Sobrecarga da função herdada da superclasse (classe pai)
     def _avancar(self):
          # Por convenção, se valor == None, fim da sequência
          self._previo,self._valor=self._valor,self._previo+self._valor

if __name__ == '__main__':
     # Instancia objeto
     n = int(input('Entre com o número de termos: '))
     Fib = FibonacciProgression()
     print(Fib.gera_sequencia(n))
```

Ex: Implemente uma classe derivada de Progression para gerar os termos de uma recorrência linear do tipo:

$$T_{n+2} = aT_{n+1} + bT_n$$

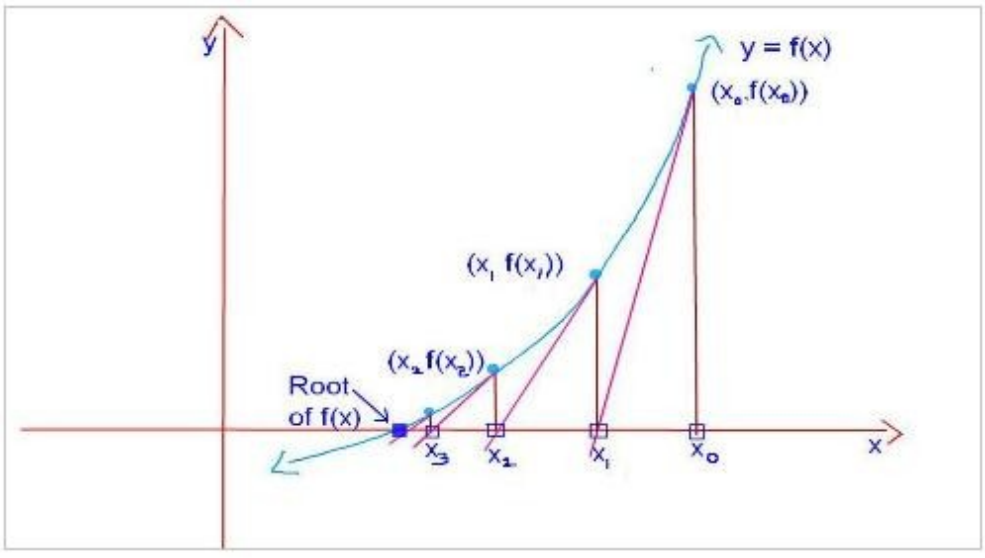
onde a e b são números inteiros arbitrários (positivos ou negativos) e a condição inicial é dada por:

$$T_0 = x$$
$$T_1 = y$$

com x e y sendo inteiros arbitrários.

O Método de Newton

Um problema de grande importância na matemática consiste em encontrar as raízes de uma função f(x), quando elas existem. A ideia básica do método consiste em, a partir de uma escolha inicial x_0 relativamente próxima a verdadeira raiz, aproximar a função pela reta tangente a $(x_0, f(x_0))$ e então computar o ponto em que essa reta intercepta o eixo x, que tipicamente será uma melhor aproximação a raiz da função.



A equação da reta que passa pelo ponto (x_0, $f(x_0)$) e é tangente a curva nesse ponto tem inclinação igual a $m=f'(x_0)$ (coeficiente angular é a derivada). Dessa forma, temos a seguinte equação:

$$y-y_0=m(x-x_0)$$

Substituindo os valores, temos:

$$y=f(x_0)+f'(x_0)(x-x_0)$$

Como queremos o ponto x em que a reta intercepta o eixo x, então $y=0$:

$$f(x_0)+f'(x_0)(x-x_0)=0$$

Dividindo ambos os lados por $f'(x_0)$:

$$\frac{f(x_0)}{f'(x_0)}+x-x_0=0$$

o que implica em

$$x=x_0-\frac{f(x_0)}{f'(x_0)}$$

Como o processo é iterativo (deve ser repetido várias vezes), chega-se na seguinte relação de recorrência:

$$x_{k+1}=x_k-\frac{f(x_k)}{f'(x_k)}$$

onde x0 pode ser escolhido arbitrariamente (idealmente próximo da raiz).

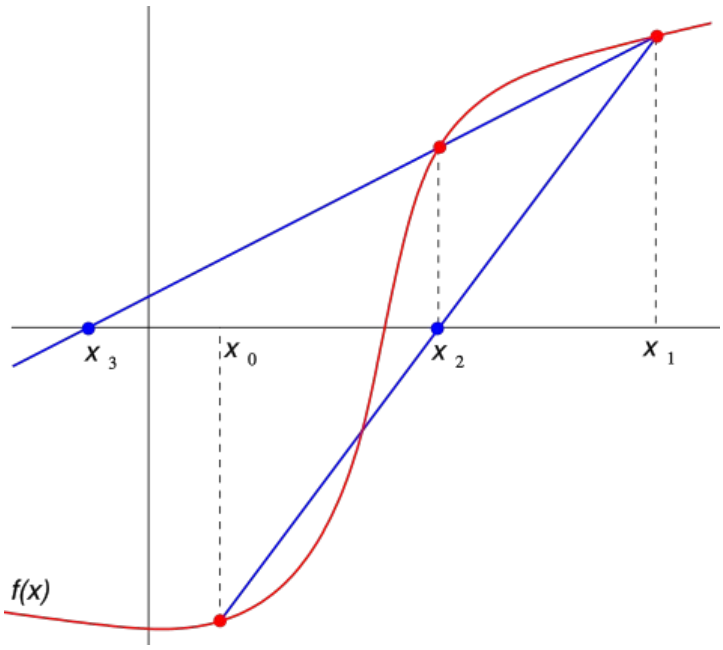
Aplicação: cálculo da raiz quadrada de um número.

Suponha a equação $f(x)=x^2-a=0$. Assim, a derivada vale $f'(x)=2x$ e temos:

$$x_{k+1}=x_k-\frac{x_k^2-a}{2x_k} \quad \rightarrow \quad x_{k+1}=x_k-\frac{x_k}{2}+\frac{a}{2x_k} \quad \rightarrow \quad x_{k+1}=\frac{1}{2}\left(x_k+\frac{a}{x_k}\right)$$

Método da secante

Um problema com o método de Newton é que ele depende explicitamente da derivada da função f(x). Em alguns casos, ela pode ser difícil ou até mesmo impossível de calcular. Um outro método para encontrar raízes de funções que não requer derivadas é o método das secantes. Esse método pode ser pensado como uma aproximação do método de Newton utilizando a técnica de diferenças finitas para o computo numérico das derivadas.



Iniciando pelos pontos x_0 e x_1 é possível construir uma linha entre $(x_0, f(x_0))$ e $(x_1, f(x_1))$, como indicado na figura acima. A equação dessa reta é dada por:

$$y - f(x_1) = m(x - x_1)$$

onde $m = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$ (inclinação da reta)

Assim, temos

$$y = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1)$$

Para encontrar o ponto em que essa reta intercepta o eixo x, basta atribuir valor zero a y:

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1) = 0$$

Multiplicando ambos os lados por $\frac{x_1 - x_0}{f(x_1) - f(x_0)}$ temos:

$$x - x_1 + f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)} = 0$$

Isolar x nos leva a:

$$x = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

Como o processo é iterativo e deve ser repetido por várias vezes, chega-se a seguinte relação de recorrência:

$$x_k = x_{k-1} - f(x_{k-1}) \frac{x_{k-1} - x_{k-2}}{f(x_{k-1}) - f(x_{k-2})}$$

A seguir apresentamos uma classe que implementa os métodos para encontrar raízes de funções não lineares arbitrárias: Newton e Secante. Utilizamos as funções lambda (lambda functions) para definir a função cuja raiz desejamos encontrar.

```
import numpy as np

class RootFinder:

     # Construtor (usado para instanciar novos objetos)
     def __init__(self, f, df):
          self.f = f           # Função
          self.df = df         # Derivada da função


     # Método de Newton
     # x0: chute inicial
     # epson: tolerância
     def Newton(self, x0, epson):
          x = x0
          novo_x = np.random.random()
          erro = abs(x - novo_x)

          while  erro >= epson:
               novo_x = x - f(x)/df(x)
               erro = abs(x - novo_x)
               print('x : %.10f ******** Erro: %.10f' %(novo_x, erro))
               x = novo_x

          return x

     # Método da Secante
     # x0, x1: chutes iniciais
     # epson: tolerância
     def Secante(self, x0, x1, epson):
          erro = abs(x0 - x1)

          while  erro >= epson:
               novo_x = x1 - f(x1)*(x1 - x0)/(f(x1) - f(x0))
               erro = abs(x1 - novo_x)
               print('x : %.10f ******** Erro: %.10f' %(novo_x, erro))
               x0, x1 = x1, novo_x

          return novo_x

# Esse comando diz ao interpretador se estamos executando o script
# Se for uma execução, ele entra aqui e prossegue
# Caso contrário, se eu incluo esse arquivo usando import, ele ignora
if __name__ == '__main__':
     # Cria a função cuja raiz será encontrada
     f = lambda x: x**3 + x – 1
     # Cria a função que representa a derivada de f (para Newton)
     df = lambda x: 3*x**2 + 1
```

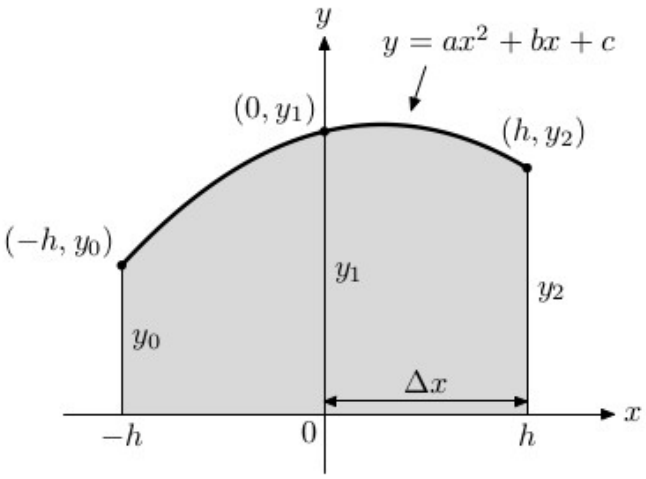
```
metodo = RootFinder(f, df)
metodo.Newton(0, 10**(-8))
print()
metodo.Secante(0, 1, 10**(-8))
```

Integração numérica

Ser capaz de calcular a integral definida de uma função numericamente é muito importante, pois nos permite computar áreas sob curvas. Dentre aos métodos mais conhecidos para esse fim, encontra-se a regra de Simpson. Veremos a seguir como essa regra funciona e como podemos criar uma classe Integration para resolver esse problema com a linguagem de programação Python.

A Regra de Simpson

A regra de Simpson é um método numérico que aproxima o valor de uma integral definida através de polinômios quadráticos (parábolas). É muito utilizado como uma forma computacional de se calcular a área sob uma curva. Primeiro, iremos derivar uma fórmula para calcular a área sob uma parábola definida pela equação $y = ax^2 + bx + c$ passando por 3 pontos: (-h, y_0), (0, y_1) e (h, y_2), conforme ilustra a figura a seguir.



A área sob a curva nada mais é que a integral definida da função y = f(x) de -h a h:

$$
\begin{aligned}
A &= \int_{-h}^{h} (ax^2 + bx + c)\, dx \\
&= \left(\frac{ax^3}{3} + \frac{bx^2}{2} + cx \right) \Bigg|_{-h}^{h} \\
&= \frac{2ah^3}{3} + 2ch \\
&= \frac{h}{3} \left(2ah^2 + 6c \right)
\end{aligned}
$$

Como os 3 pontos (-h, y_0), (0, y_1) e (h, y_2) pertencem a parábola, eles satisfazem a equação $y = ax^2 + bx + c$ e portanto:

$$y_0 = ah^2 - bh + c$$
$$y_1 = c$$
$$y_2 = ah^2 + bh + c$$

Note porém que a seguinte igualdade é válida:

$$y_0 + 4y_1 + y_2 = (ah^2 - bh + c) + 4c + (ah^2 + bh + c) = 2ah^2 + 6c.$$

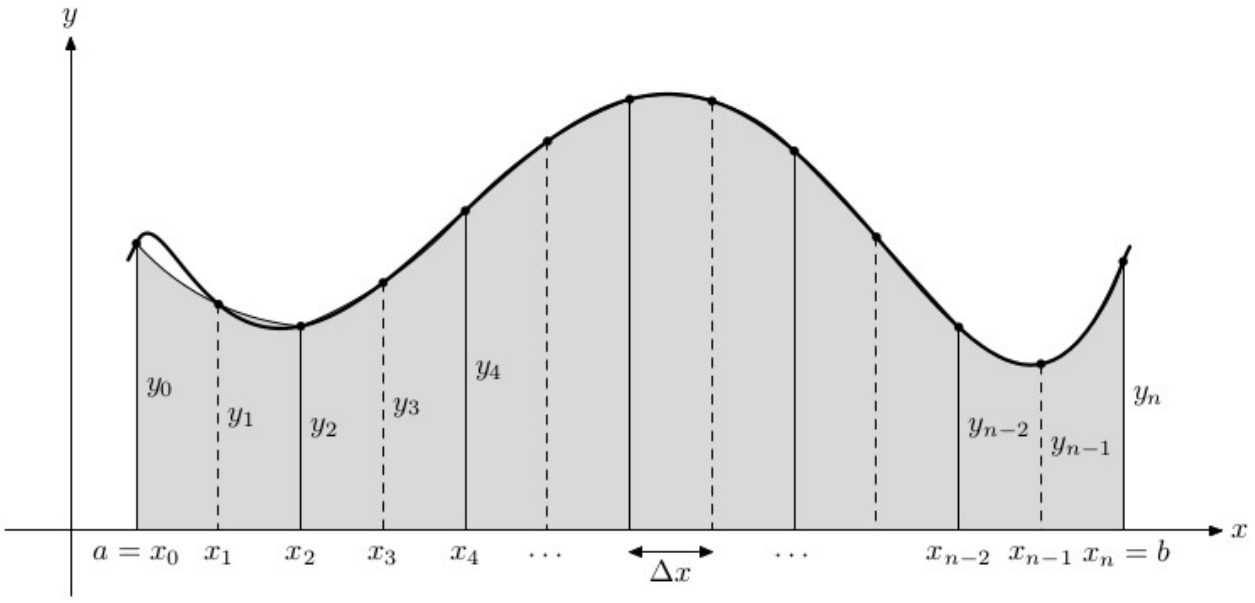
Assim, a área sob a parábola pode ser reescrita como:

$$A = \frac{h}{3}(y_0 + 4y_1 + y_2) = \frac{\Delta x}{3}(y_0 + 4y_1 + y_2)$$

Para aplicar a regra de Simpson para a integração numérica de uma função f(x) qualquer, deseja-se resolver a seguinte integral: $\int_a^b f(x)$. Assumindo f(x) contínua no intervalo [a, b] e dividindo o intervalo em um número par n de subintervalos de tamanhos iguais a $\Delta x = \frac{b-a}{n}$, definimos n+1 pontos, para os quais podemos computar os valores da função f(x):

$$x_0 = a, \quad x_1 = a + \Delta x, \quad x_2 = a + 2\Delta x, \quad \ldots, \quad x_n = a + n\Delta x = b.$$

$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \ldots, \quad y_n = f(x_n).$$



Podemos estimar a integral pela soma das áreas sob os arcos parabólicos formados por cada 3 pontos sucessivos:

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(y_0 + 4y_1 + y_2\right) + \frac{\Delta x}{3}\left(y_2 + 4y_3 + y_4\right) + \cdots + \frac{\Delta x}{3}\left(y_{n-2} + 4y_{n-1} + y_n\right)$$

Simplificando a expressão anterior, chega-se a:

$$\boxed{\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_{n-1} + y_n\right)}$$

Exercício: Utilizando uma calculadora, aplique a regra de Simpson com n = 6 para aproximar a integral

$$\int_1^4 \sqrt{1+x^3}\,dx$$

Para n = 6, temos subintervalos de largura $\Delta x = \frac{b-a}{n} = \frac{4-1}{6} = 0.5$. Assim, os pontos ficam:

x	1	1.5	2	2.5	3	3.5	4
$y = \sqrt{1+x^3}$	$\sqrt{2}$	$\sqrt{4.375}$	3	$\sqrt{16.625}$	$\sqrt{28}$	$\sqrt{43.875}$	$\sqrt{65}$

Portanto, o valor final da integral pode ser computado por:

$$\begin{aligned}\int_1^4 \sqrt{1+x^3}\,dx &\approx \frac{0.5}{3}\left(\sqrt{2} + 4\sqrt{4.375} + 2(3) + 4\sqrt{16.625} + 2\sqrt{28} + 4\sqrt{43.875} + \sqrt{65}\right)\\ &\approx \boxed{12.871}\end{aligned}$$

Existe uma variação da regra de Simpson que utiliza interpolação cúbica ao invés de interpolação quadrática, denominada de regra de Simpson 3/8. A expressão para a integral definida de uma função f(x) no intervalo [a, b] fica:

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{3h}{8}\left[f(x_0) + 3f(x_1) + 3f(x_2) + 2f(x_3) + 3f(x_4) + 3f(x_5) + 2f(x_6) + \cdots + 3f(x_{n-2}) + 3f(x_{n-1}) + f(x_n)\right].\\ &= \frac{3h}{8}\left[f(x_0) + 3\sum_{i\neq 3k}^{n-1} f(x_i) + 2\sum_{j=1}^{n/3-1} f(x_{3j}) + f(x_n)\right] \qquad \text{For: } k \in \mathbb{N}_0\end{aligned}$$

onde h = (b – a)/n. A lógica aqui consiste em multiplicar por 3 todos os valores da função nos pontos xi, exceto os pontos em que i é múltiplo de 3, como i = 3, 6, 9, … etc, o que gera um padrão simples de ser programado computacionalmente.

A seguir implementamos a classe Integration, que armazena a função, os valores de a, b e n, além do métodos simpson e simpson_3_8.

```
import numpy as np

class Integration:

	# Construtor (usado para instanciar novos objetos)
	def __init__(self, f):
		self.f = f		# Função

	# Método de Simpson
	def Simpson(self, a, b, n):
		h = (b - a)/n
		soma_pares, soma_impares = 0, 0
		# Soma os pares
		for i in range(2, n, 2):
			k = a + i*h
			soma_pares += self.f(k)
		# Soma os ímpares
		for i in range(1, n, 2):
			k = a + i*h
			soma_impares += self.f(k)
		area = (h/3)*(self.f(a) + 4*soma_impares + \
					2*soma_pares + self.f(b))
		return area

	# Método de Simpson 3/8
	# n deve ser múltiplo de 3
	def Simpson_3_8(self, a, b, n):
		h = (b - a)/n
		soma_mult_3, soma_restante = 0, 0
		# Soma os pares
		for i in range(1, n):
			k = a + i*h
			# Se é multiplo de 3
			if i % 3 == 0:
				soma_mult_3 += self.f(k)
			else:
				soma_restante += self.f(k)
		area = (3*h/8)*(self.f(a) + 2*soma_mult_3 + \
					3*soma_restante + self.f(b))
		return area


if __name__ == '__main__':
	# Instancia objeto e faz operações
	f = lambda x: np.sqrt((1 + x**3))
	g = lambda x: 1/np.sqrt((1 + x**4))
	# Densidade Normal(0, 1)
	p = lambda x: (1/(2*np.pi)**0.5)*np.exp(-0.5*x**2)

	# Função f
	metodo = Integration(f)
	print('Simpson: ', metodo.Simpson(1, 4, 20))
	print('Simpson 3/8: ', metodo.Simpson_3_8(1, 4, 21))
	print()
```

```
# Função g
metodo = Integration(g)
print('Simpson: ', metodo.Simpson(0, 2, 20))
print('Simpson 3/8: ', metodo.Simpson_3_8(0, 2, 21))
print()

# Função p
metodo = Integration(p)
print('Simpson: ', metodo.Simpson(0, 3, 20))
print('Simpson 3/8: ', metodo.Simpson_3_8(0, 3, 21))
print()
```

Ex: A regra de Simpson estendida é uma formulação alternativa para quando o número de pontos n é maior que 8. Ela é dada por:

$$\int_a^b f(x)\,dx \approx \frac{h}{48}\left[17f(x_0) + 59f(x_1) + 43f(x_2) + 49f(x_3) + 48\sum_{i=4}^{n-4} f(x_i) + 49f(x_{n-3}) + 43f(x_{n-2}) + 59f(x_{n-1}) + 17f(x_n)\right].$$

Os 4 primeiros elementos e os 4 últimos possuem pesos simétricos e os elementos do miolo possuem todos o mesmo peso. Modifique a classe Integration definida anteriormente, criando um novo método chamado Simpson_Estendido que calcula numericamente a integral utilizando a expressão acima. Verifique se n > 8. Caso não seja, imprima na tela a mensagem “Número de pontos insuficiente”.

Aula 5 – Estruturas de dados lineares: Pilhas, Filas

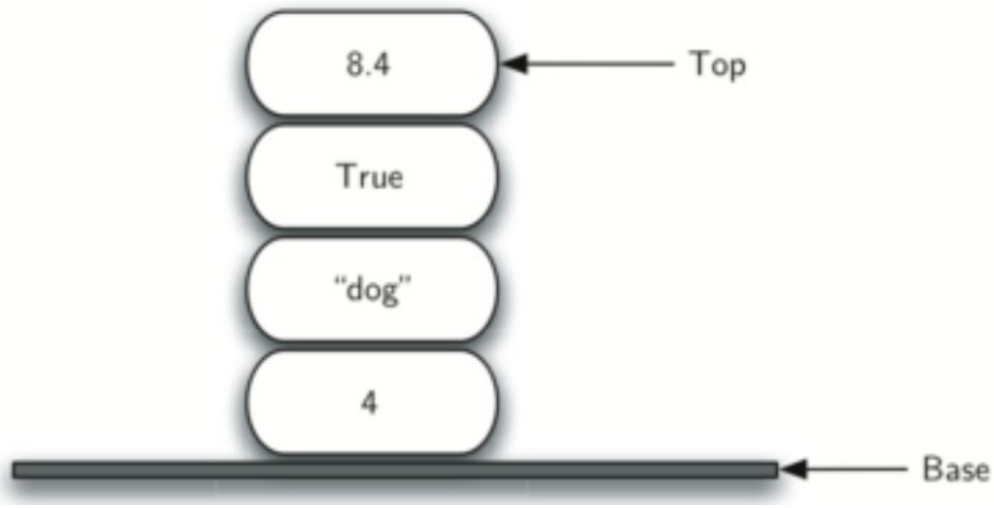
Na programação de computadores, estruturas de dados são abstrações lógicas criadas para organizarmos dados na memória de maneira eficiente para o posterior acesso e manipulação. Do ponto de vista prático, estruturas de dados são compostas por coleções de elementos, o relacionamento entre esses elementos e as funções/operações que podem ser aplicadas nesses elementos.

Iniciaremos nosso estudo com Pilhas e Filas, que são coleções em que os elementos são organizados dependendo de como eles são adicionados ou removidos do conjunto. Uma vez que um elemento é adicionado ele permanece na mesma posição relativa ao elemento que veio antes e o elemento que veio depois. Essa é a característica comum de todas as estruturas lineares.

Sendo bastante simplista, estruturas de dados lineares são aquelas que possuem duas extremidades: início e fim, ou esquerda e direita, ou base e topo. A nomenclatura não é relevante para nós, pois o que realmente importa é como são realizadas as inserções e remoções de elementos na estrutura.

Pilhas (LIFO)

Uma pilha (stack) é uma estrutura de dados linear em que a inserção e a remoção de elementos é realizada sempre na mesma extremidade, comumente de chamada de topo. O oposto do topo é a base. Quanto mais próximo da base está um elemento, há mais tempo ele está armazenado na estrutura. Por outro lado, um item inserido agora estará sempre no topo, o que significa que ele será o primeiro a ser removido (se não empilharmos novos elementos antes). Por esse princípio de ordenação inerente das pilhas, elas são conhecidas como a estrutura **LIFO**, do inglês, *Last In First Out*, ou seja, primeiro a entrar é último a sair. Essa intuição faz total sentido com uma pilha de livros por exemplo. O primeiro livro a ser empilhado fica na base da pilha e será o último a ser retirado. A ordem de remoção é o inverso da ordem de inserção. A figura a seguir ilustra uma pilha de objetos em Python.



Para que possamos implementar um classe Pilha, devemos ter em mente quais suas variáveis internas e quais as operações que podemos aplicar sobre os seus elementos. A listagem a seguir mostra como podemos construir a pilha da figura acima, a partir de uma pilha inicialmente vazia.
Note que podemos utilizar uma lista P para armazenar os elementos da pilha.

Operação	Conteúdo	Retorno	Descrição
s.is_empty()	[]	True	Verifica se pilha está vazia
s.push(4)	[4]		Insere elemento no topo
s.push('dog')	[4, 'dog']		Insere elemento no topo
s.peek()	[4, 'dog']	'dog'	Consulta o elemento do topo (mantém)
s.push(True)	[4, 'dog', True]		Insere elemento no topo
s.size()	[4, 'dog', True]	3	Retorna número de elementos da pilha
s.is_empty()	[4, 'dog', True]	False	Verifica se pilha está vazia
s.puch(8.4)	[4, 'dog', True, 8.4]		Insere elemento no topo
s.pop()	[4, 'dog', True]	8.4	Remove elemento do topo
s.pop()	[4, 'dog']	True	Remove elemento do topo
s.size()	[4, 'dog']	2	Retorna número de elementos da pilha

Para implementar uma pilha em Python, iremos utilizar como atributo uma lista chamada items, que inicia vazia. Definiremos os métodos push() e pop() para inserção e remoção de elementos do topo, bem como peek(), size() e is_empty(), para consultar o elemento do topo, obter o número de elementos da pilha e verifica se a pilha está vazia. Note que na nossa implementação de pilha, tanto a inserção quanto a remoção tem complexidade O(1).

```
# Implementação da classe Pilha
class Stack:

    # Inicia com uma pilha vazia
    def __init__(self):
        self.itens = []

    # Verifica se pilha está vazia
    def is_empty(self):
        return self.itens == []

    # Adiciona elemento no topo (topo é o final da lista)
    def push(self, item):
        self.itens.append(item)
        print('PUSH %s' %item)

    # Remove elemento do topo (final da lista)
    def pop(self):
        print('POP')
        return self.itens.pop()

    # Obtém o elemento do topo (mas não remove)
    def peek(self):
        # Em Python, índice -1 retorna último elemento (topo)
        return self.itens[-1]

    # Retorna o número de elementos da pilha
    def size(self):
        return len(self.itens)

    # Imprime pilha na tela
    def print_stack(self):
        print(self.itens)
```

```
if __name__ == '__main__':

    S = Stack()
    S.print_stack()
    S.push(1)
    S.push(2)
    S.push(3)
    S.print_stack()
    S.pop()
    S.pop()
    S.print_stack()
    S.push(7)
    S.push(8)
    S.push(9)
    S.print_stack()
    print(S.is_empty())
```

Ex: Uma aplicação interessante que utiliza uma estrutura de dados do tipo Pilha é a verificação do balanceamento dos parêntesis de uma expressão matemática. Uma expressão matemática é bem formada se o número de parêntesis é par, sendo que para cada (deve existir um). Por exemplo, a expressão a seguir é válida:

((1 + 2) * (3 + 4)) - (5 + 6)

Já expressão a seguir não é válida:

((1 + 2) * (3 + 4) – (5 + 6)

Como implementar uma função em Python que verifique se uma dada expressão é válida ou não? E se a expressão permitir parêntesis, colchetes e chaves, como em:

{ [(1 + 2) + (3 + 4)] * 2 }

Iremos apresentar um código Python para resolver esses problemas. A ideia consiste em utilizar uma pilha para empilhar cada abre parêntesis que aparece na expressão e ao encontrar um fecha parêntesis, devemos desempilhar o seu par da pilha. Se ao final do processo a pilha estiver vazia, ou seja, para cada (existe um respectivo), então a fórmula é considerada válida. No caso de expressões compostas, em que existem vários tipos de símbolos, como {, [e (, ao desempilhar o fechamento, devemos nos atentar se o par é bem formado, ou seja, se temos ( ), [ ] ou { }.

```
from Pilha import Stack

# Apenas parêntesis são permitidos na expressão: (, )
def verifica_expressao_simples(expressao):
    # Cria pilha vazia
    s = Stack()
    balanceado = True
    indice = 0  # primeira posição na expressão
    n = len(expressao)  # tamanho da expressão (qtde de caracteres)
```

```
    # Enquanto não chegar no final e estiver balanceado
    while indice < n and balanceado:
        # Obtém o símbolo atual
        simbolo = expressao[indice]
        # Se for abre parênstesis, empilha
        if simbolo == '(':
            s.push(simbolo)
        # Se for fecha parêntesis, o abre tem que estar na pilha
        # Se não estiver na pilha, não está balanceado
        elif simbolo == ')':
            if s.is_empty():
                balanceado = False
            else:   # Se abre parêntesis está na pilha, desempilha
                s.pop()
        # Passa para o próximo caracter da expressão
        indice += 1

    # Se ao final estiver balanceado e não sobrou nada na pilha, OK
    if balanceado and s.is_empty():
        return True
    else:
        return False

# Expressão completa permite {, [, (, ), ], }
def verifica_expressao_composta(expressao):
    # Cria pilha vazia
    s = Stack()
    balanceado = True
    indice = 0  # primeira posição na expressão
    n = len(expressao) # tamanho da expressão (qtde de caracteres)

    # Enquanto não chegar no final e estiver balanceado
    while indice < n and balanceado:
        # Obtém o símbolo atual
        simbolo = expressao[indice]
        # Se símbolo é um dos abre parêntesis, empilha
        if simbolo in '([{':
            s.push(simbolo)
        # Se for algum fechamento, o abre tem que estar na pilha
        # Se não estiver na pilha, não está balanceado
        elif simbolo in ')]}':
            if s.is_empty():
                balanceado = False
            else: # Se abre está na pilha, verifica se são iguais
                abre = s.pop()
                # Se o abre e o fecha não são iguais, está desbalanceado
                if (simbolo == '}' and abre != '{') or \
                   (simbolo == ']' and abre != '[') or \
                   (simbolo == ')' and abre != '('):
                    balanceado = False
        indice += 1

    if balanceado and s.is_empty():
        return True
    else:
        return False
```

```
if __name__ == '__main__':

    print('Expressões simples')
    print(verifica_expressao_simples('((1 + 2) * (3 + 4)) - (5 + 6)'))
    print()
    print(verifica_expressao_simples('((1 + 2) * (3 + 4))))'))
    print()
    print(verifica_expressao_simples('(((((1 + 2) * (3 + 4))'))
    print()

    print('Expressões compostas')
    print(verifica_expressao_composta('{ [ (1 + 2) + (3 + 4) ] * 2 }'))
    print()
    print(verifica_expressao_composta('{ [ (1 + 2} + [3 + 4} ) * 2 )'))
```

Um outro problema interessante que pode ser resolvido com uma estrutura de dados do tipo Pilha é a conversão de um número decimal (base 10) para binário (base 2). A representação de números inteiros por computadores digitais é feita na base 2 e não na base 10. Sabemos que um número na base 10 (decimal) é representado como:

$23457 = 2 \times 10^4 + 3 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$

Veja o quanto mais a esquerda estiver o dígito, maior o seu valor. O 2 em 23457 vale na verdade 20000 pois é igual a 2×10^4 .

Analogamente, um número binário (base 2) pode ser representado como:

$110101 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 53$

O bit mais a direita é o menos significativo e portanto o seu valor é $1 \times 2^0 = 1$
O segundo bit a partir da direita tem valor de $0 \times 2^1 = 0$
O terceiro bit a partir da direita tem valor de $1 \times 2^2 = 4$
O quarto bit a partir da direita tem valor de $0 \times 2^3 = 0$
O quinto bit a partir da esquerda tem valor de $1 \times 2^4 = 16$
Por fim, o bit mais a esquerda tem valor de $1 \times 2^5 = 32$
Somando tudo temos: 1 + 4 + 16 + 32 = 5 + 48 = 53.

Essa é a regra para convertermos um número binário para sua notação decimal.

Veremos agora o processo inverso: como converter um número decimal para binário. O processo é simples. Começamos dividindo o número decimal por 2:

53 / 2 = 26 e sobra resto **1** → esse 1 será nosso bit mais a direita (menos significativo no binário)

Continuamos o processo até que a divisão por 2 não seja mais possível:

26 / 2 = 13 e sobra resto **0** → esse 0 será nosso segundo bit mais a direita no binário

13 / 2 = 6 e sobra resto **1** → esse 1 será nosso terceiro bit mais a direita no binário

6 / 2 = 3 e sobra resto **0** → esse 0 será nosso quarto bit mais a direita no binário

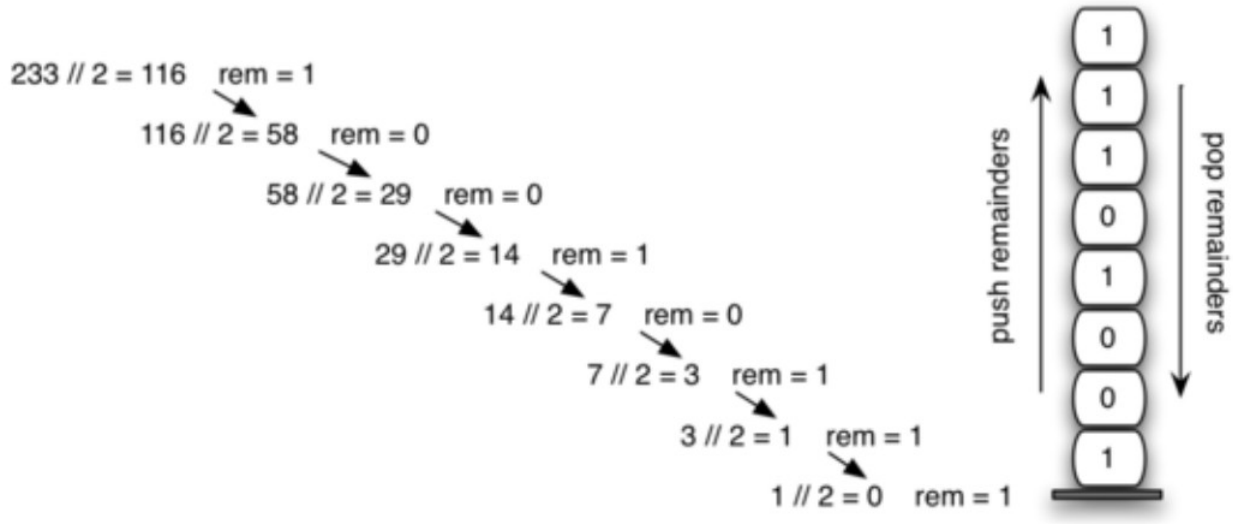
3 / 2 = 1 e sobra resto **1** → esse 1 será nosso quinto bit mais a direita no binário

1 / 2 = 0 e sobra resto **1** → esse 1 será o nosso último bi (mais a esquerda)

Note que de agora em diante não precisamos continuar com o processo pois

0 / 2 = 0 e sobra 0
0 / 2 = 0 e sobra 0

ou seja a esquerda do sexto bit teremos apenas zeros, e como no sistema decimal, zeros a esquerda não possuem valor algum. Portanto, 53 em decimal equivale a 110101 em binário. Note que se empilharmos os restos em uma pilha, ao final, basta desempilharmos para termos o número binário na sua forma correta. A figura a seguir ilustra essa ideia.



O código em Python a seguir mostra a implementação de uma função que converte um número inteiro arbitrário na base 10 (decimal) para a representação binária.

```
from Pilha import Stack

# Função que converte um número decimal para binário
def decimal_binario(numero):
    s = Stack()

    while numero > 0:
        resto = numero % 2
        s.push(resto)
        numero = numero // 2
    binario = ''
    while not s.is_empty():
        binario = binario + str(s.pop())

    return binario

if __name__ == '__main__':
    n = int(input('Entre com um número inteiro: '))
    print(decimal_binario(n))
```

Ex: Escreva uma função para determinar se uma cadeia de caracteres (string) é da forma: xCy onde x e y são cadeias de caracteres compostas por letras ‘A’ ou ‘B’, e y é o inverso de x. Isto é, se:

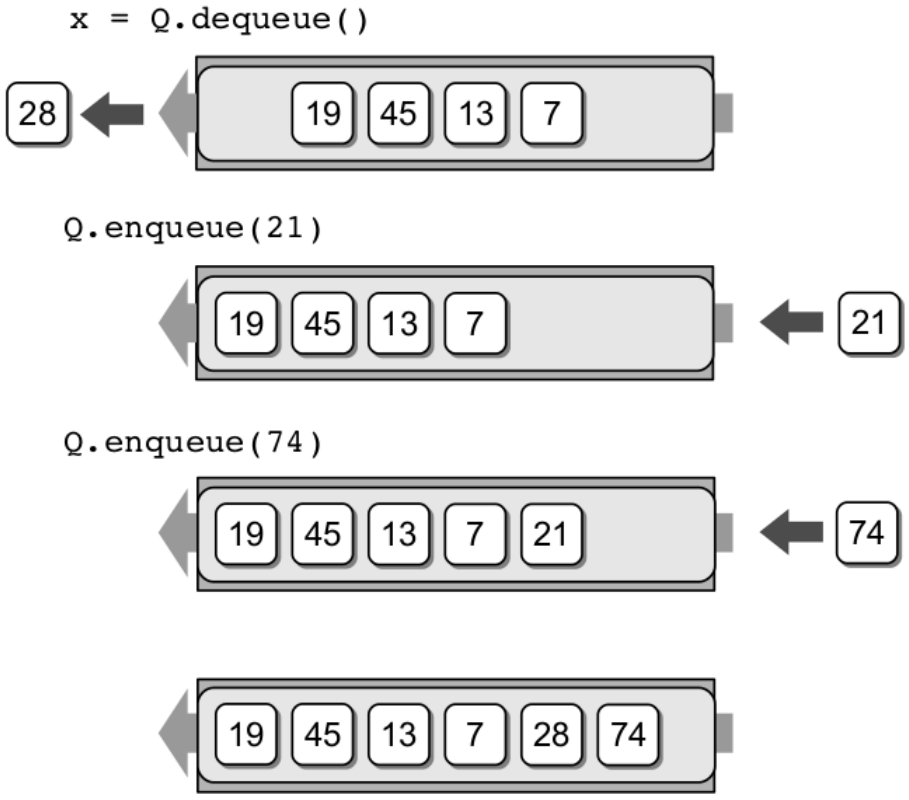
x = ‘ABABBA’

y deve equivaler a ‘ABBABA’

Em cada ponto, você só poderá ler o próximo caractere da cadeia (use uma pilha).

Filas (FIFO)

Uma fila é uma estrutura de dados linear em que a inserção de elementos é realizada em uma extremidade (final) e a remoção é realizada na outra extremidade (início). Assim como em uma fila de pessoas no mundo real, o primeiro a entrar será o primeiro a sair, o que define o princípio de ordenação FIFO, ou do inglês, *First In First Out*. Além disso, filas devem ser restritivas no sentido de que um elemento não pode passar na frente de seu antecessor. A figura abaixo ilustra o processo.



Na ciência da computação há diversos exemplos de aplicações que utilizam filas para gerenciar a ordem de acesso aos recursos. Por exemplo, um servidor de impressão localizado em um laboratório de pesquisa em um departamento da universidade precisar lidar com o gerenciamento da ordem das impressões. Para isso, é usual o software da impressora criar uma fila de impressões. Assim, múltiplas requisições são tratadas sequencialmente de acordo com a ordem em que chegam.
Para que possamos implementar um classe Fila, devemos ter em mente quais suas variáveis internas e quais as operações que podemos aplicar sobre os seus elementos. A listagem a seguir mostra como podemos construir uma fila a partir de uma lista inicialmente vazia. As operações são bastante parecidas com as operações de uma pilha.

Operação	Conteúdo	Retorno	Descrição
q.is_empty()	[]	True	Verifica se fila está vazia
q.enqueue(4)	[4]		Insere elemento no final
q.enqueue('dog')	['dog', 4]		Insere elemento no final
q.enqueue(True)	[True, 'dog', 4]		Insere elemento no final
q.size()	[True, 'dog', 4]	3	Retorna número de elementos da fila
q.is_empty()	[True, 'dog', 4]	False	Verifica se fila está vazia
q.enqueue(8.4)	[8.4, True, 'dog', 4]		Insere elemento no final
q.dequeue()	[8.4, True, 'dog']	4	Remove elemento do início
q.dequeue()	[8.4, True]	'dog'	Remove elemento do início
q.size()	[8.4, True]	2	Retorna número de elementos da fila

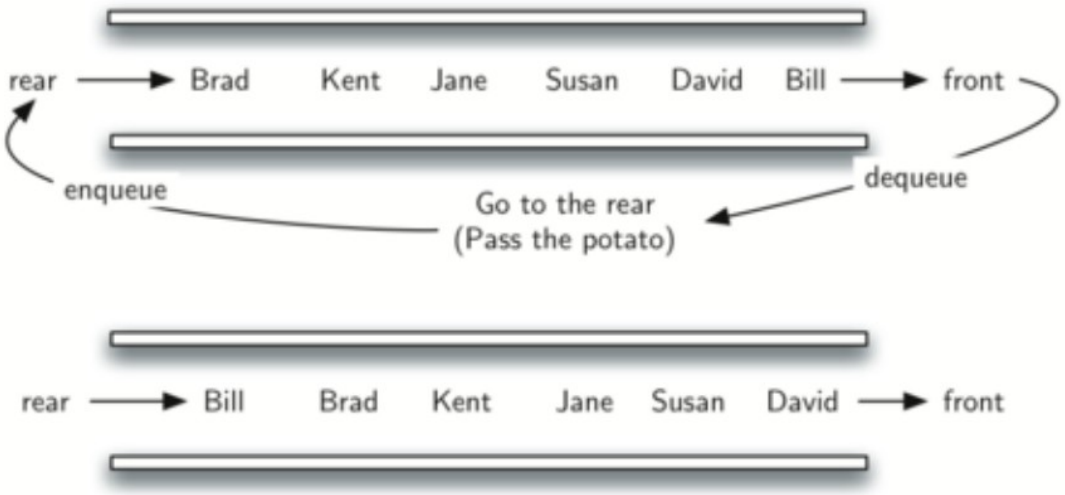
Para implementar uma fila em Python, iremos utilizar a mesma estratégia utilizada com a pilha: como atributo teremos uma lista chamada items, que inicia vazia. Definiremos os métodos enqueue() e dequeue() para inserção de elementos no final e remoção de elementos no início, bem como, size() e is_empty(), obter o número de elementos da fila e verificar se a fila está vazia. Note que em nossa implementação de fila uma fila, a inserção tem custo O(n), pois para inserir na posição zero (início), utilizamos o método insert de uma lista, que é O(n) (é como se precisasse mover todos os elementos da lista uma posição a direita), mas a remoção tem custo O(1).

```
# Implementação da classe Fila
class Queue:

    # Inicia com uma fila vazia
    def __init__(self):
        self.itens = []

    # Verifica se fila está vazia
    def is_empty(self):
        return self.itens == []

    # Adiciona elemento no início da fila
    def enqueue(self, item):
        self.itens.insert(0, item)
        print('ENQUEUE %s' %item)

    # Remove elemento do final da fila
    def dequeue(self):
        print('DEQUEUE')
        return self.itens.pop()

    # Retorna o número de elementos da fila
    def size(self):
        return len(self.itens)

    # Imprime a fila na tela
    def print_queue(self):
        print(self.itens)


if __name__ == '__main__':

    Q = Queue()
    Q.print_queue()
    Q.enqueue(1)
```

```
    Q.enqueue(2)
    Q.enqueue(3)
    Q.print_queue()
    Q.dequeue()
    Q.dequeue()
    Q.print_queue()
    Q.enqueue(7)
    Q.enqueue(8)
    Q.enqueue(9)
    Q.print_queue()
    print(Q.is_empty())
```

Para ilustra algumas aplicações que utilizam estruturas de dados do tipo Fila. A primeira delas é uma simples simulação do jogo infantil Batata Quente. Neste jogo, as crianças formam um círculo e passam um item qualquer (batata) cada um para o seu vizinho da frente o mais rápido possível. Em um certo momento do jogo, essa ação é interrompida (queimou) e a criança que estiver com o item (batata) na mão é excluída da roda. O jogo então prossegue até que reste apenas uma única criança, que é a vencedora.

Para simular um círculo (roda), utilizaremos uma fila da seguinte maneira: a criança que está com a batata na mão será sempre a quela que estiver no início da fila. Após passar a batata, a simulação deve instantaneamente remover e inserir a criança, colocando-a novamente no final da fila. Ela então vai esperar até que todas as outras assumam o início da fila, antes de assumir essa posição novamente. Após um número pré estabelecido MAX de operações enqueue/dequeue, a criança que ocupar o início da fila será removida e outro ciclo da brincadeira é realizado. O processo continua até que a fila tenha possua tamanho um. A figura a seguir ilustra o processo.



A implementação em Python da simulação é apresentada a seguir.

```
# Importa definição da classe
from Fila import Queue

# Simula o jogo batata_quente
def batata_quente(nomes, MAX):
    # Cria fila para simular roda
    fila = Queue()
    # Coloca os N nomes em cada posição
    for nome in nomes:
        fila.enqueue(nome)
```

```
    # Inicia a lógica do jogo
    while fila.size() > 1:
        # Para simular MAX passagens da batata
        for i in range(MAX):
            # Remove o primeiro e coloca no final
            fila.enqueue(fila.dequeue())
        # Quem parar no ínicio da fila, está com batata
        # Deve ser eliminado da fila
        fila.dequeue()
    # Após N-1 rodadas, retorna a fila com o vencedor
    vencedor = fila.dequeue()

    return vencedor
if __name__ == '__main__':
    # Chama a função para simular o jogo
    v = batata_quente(['Alex','Julia','Carlos','Maria','Ana','Caio'], 7)
    print('O vencedor é %s' %v)
```

Ex: Para um dado número inteiro n > 1, o menor inteiro d > 1 que divide n é chamado de fator primo. É possível determinar a fatoração prima de n achando-se o fator primo d e substituindo n pelo quociente n / d, repetindo essa operação até que n seja igual a 1. Utilizando uma das estruturas de dados lineares (pilha ou fila) para auxiliá-lo na manipulação de dados, implemente uma função que compute a fatoração prima de um número imprimindo os seus fatores em ordem decrescente. Por exemplo, para n=3960, deverá ser impresso 11 * 5 * 3 * 3 * 2 * 2 * 2. Justifique a escolha do TAD utilizado.

Ex: Modifique a simulação do jogo batata quente de modo a permitir que o número passagens MAX seja um número aleatório de 4 até 15. Assim, em cada rodada, teremos um valor de MAX diferente.

Ex: Usando uma pilha P inicialmente vazia, implemente um método para inverter uma fila Q com n elementos utilizando apenas os métodos is_empty(), push(), pop(), enqueue() e dequeue().

```
from Fila import Queue
from Pilha import Stack

def inverte_fila(Q):
    # Cria pilha vazia
    S = Stack()
    # Enquanto tiver elementos na fila
    while not Q.is_empty():
        S.push(Q.dequeue())
    # Enquanto tiver elementos na pilha
    while not S.is_empty():
        Q.enqueue(S.pop())

    return Q
if __name__ == '__main__':
    # Cria fila vazia
    Q = Queue()
    # Adiciona elementos na fila
    Q.enqueue(1)
    Q.enqueue(2)
    Q.enqueue(3)
```

```
    Q.enqueue(4)
    Q.enqueue(5)
    # Imprime fila
    Q.print_queue()
    # Cria fila invertida
    IQ = inverte_fila(Q)
    # imprime fila invertida
    IQ.print_queue()
```

Deque: Double-Ended Queue (Fila de duas extremidades)

Deque (pronuncia-se deck para diferenciar da operação dequeue), ou Double-Ended Queue, nome que traduziremos como Dupla Fila, é uma estrutura de dados linear bastante similar a uma fila tradicional, porém com dois inícios e dois finais. O que a torna diferente da fila é justamente a possibilidade de inserir e remover elementos tanto do início quanto do fim da lista. Neste sentido, esse TAD híbrido prove todas as funcionalidades de filas e pilhas em uma única estrutura de dados. Apenas para deixar claro, denotaremos por front a parte da frente da fila (direita) e por rear a parte de trás da fila (esquerda), ou seja:

rear → [1, 2, 3, 4, 5] ← front

Operação	Conteúdo	Retorno	Descrição
d.is_empty()	[]	True	Verifica se fila está vazia
d.add_rear(4)	[4]		Insere elemento na esquerda
d.add_rear(‘dog’)	[‘dog’, 4]		Insere elemento na esquerda
d.add_front(‘cat’)	[‘dog’, 4, ‘cat’]		Insere elemento na direita
d.add_front(True)	[‘dog’, 4, ‘cat’, True]		Insere elemento na direita
d.size()	[‘dog’, 4, ‘cat’, True]	4	
d.is_empty()	[‘dog’, 4, ‘cat’, True]	False	Verifica se fila está vazia
d.add_rear(8.4)	[8.4, ‘dog’, 4, ‘cat’, True]		Insere elemento na esquerda
d.remove_rear()	[‘dog’, 4, ‘cat’, True]	8.4	Remove elemento da esquerda
d.remove_front()	[‘dog’, 4, ‘cat’]	True	Remove elemento da direita
d.size()	[‘dog’, 4, ‘cat’]	3	Retorna número de elementos

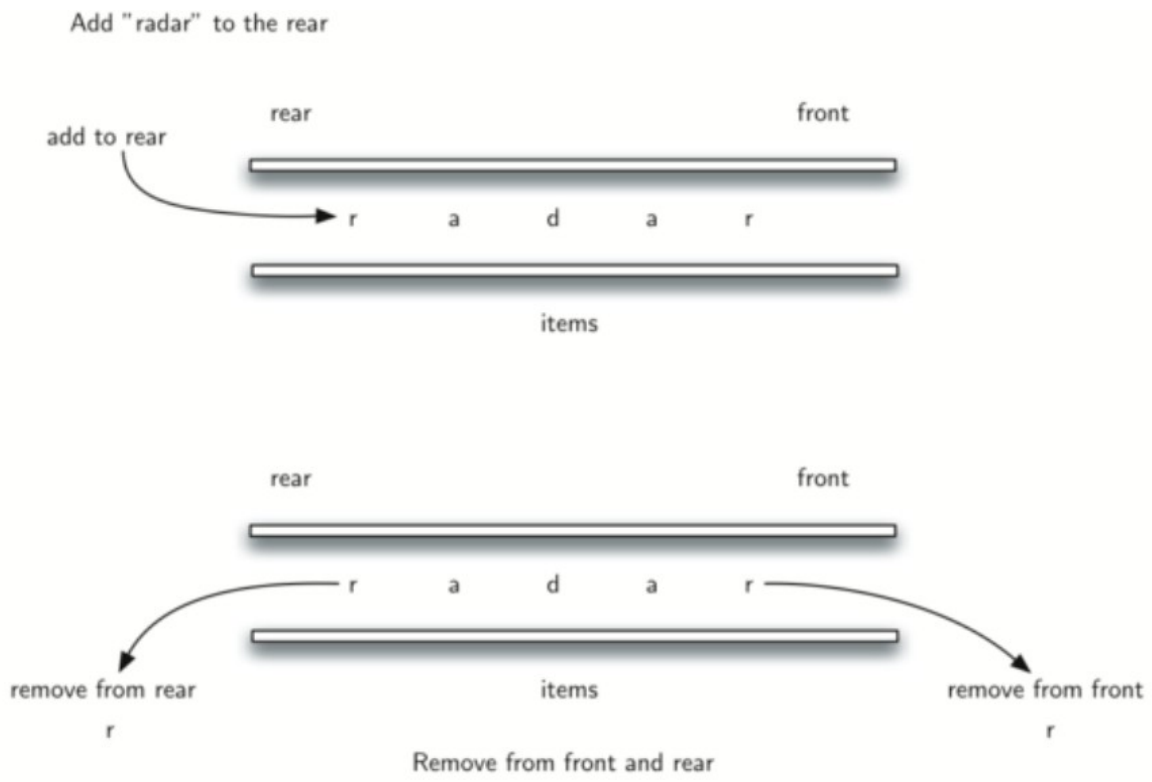
A seguir apresentamos uma implementação em Python para a classe Deque.

```
# Implementação da classe Deque (Double-Ended queue)
class Deque:

    # Inicia com uma fila vazia
    def __init__(self):
        self.itens = []

    # Verifica se fila está vazia
    def is_empty(self):
        return self.itens == []
    # Adiciona elemento na direita da fila
    def add_front(self, item):
        self.itens.append(item)
        print('ADD FRONT %s' %item)

    # Adiciona elemento na esquerda da fila
    def add_rear(self, item):
```

```
        self.itens.insert(0, item)
        print('ADD REAR %s' %item)

    # Remove elemento na direita da fila
    def remove_front(self):
        print('REMOVE FRONT')
        return self.itens.pop()

    # Remove elemento na esquerda da fila
    def remove_rear(self):
        print('REMOVE REAR')
        return self.itens.pop(0)

    # Retorna o número de elementos da fila
    def size(self):
        return len(self.itens)

    # Imprime fila na tela
    def print_deque(self):
        print(self.itens)


if __name__ == '__main__':

    D = Deque()
    D.print_deque()
    D.add_front(1)
    D.add_front(2)
    D.add_rear(3)
    D.add_rear(4)
    D.print_deque()
    D.remove_front()
    D.remove_rear()
    D.print_deque()
    D.add_front(7)
    D.add_rear(8)
    D.print_deque()
    print(D.is_empty())
```

Um aspecto interessante com essa implementação que utiliza como base a classe lista pré-definida em Python é que inserção e remoção de elementos na direita da fila (add_front e remove_front) possuem complexidade O(1), enquanto que a inserção e remoção de elementos na esquerda da fila (add_rear e remove_rear) possuem complexidade O(n) (pois precisa deslocar os outros elementos).

Um exemplo de aplicação de uma estrutura linear do tipo Deque é no problema de verificar se uma dada palavra é palíndroma, ou seja, se ela é igual a sua versão reversa. Um exemplo de palavra palíndroma é RADAR. A figura a seguir ilustra como uma estrutura Deque pode ser utilizada nesse tipo de problema.



A ideia consiste em processar cada caractere da string de entrada da esquerda para a direita adicionando cada caractere na esquerda do Deque. No próximo passo, iremos remover os caracteres das duas extremidades (esquerda e direita), comparando-as. Se os caracteres forem iguais repetimos o processo ate que não haja mais caracteres no Deque, se a string tiver um número par de caracteres, ou sobre apenas 1 caractere, se a string tiver um número ímpar de caracteres. A função a seguir implementa a solução em Python.

```
from Deque import Deque

# Função que verifica que palavra é palíndroma
def palindrome(palavra):
    # Cria Deque
    D = Deque()
    # Percorre string de entrada
    for c in palavra:
        # Adiciona na esquerda
        D.add_rear(c)
    # Verifica se letras da esquerda e da direita são iguais
    iguais = True
    # Enquanto houver mais de 1 letra e extremos forem iguais
    while D.size() > 1 and iguais:
        # Obtém extremo da esquerda
        esquerda = D.remove_rear()
        # Obtém extremo da direita
        direita = D.remove_front()
        # Se não forem iguais, deve retornar False
        if esquerda != direita:
            iguais = False

    return iguais
```

```
if __name__ == '__main__':
    print(palindrome('radar'))
    print(palindrome('arara'))
    print(palindrome('ovo'))

    print(palindrome('bicicleta'))
    print(palindrome('bola'))
    print(palindrome('carro'))
```

Fila de prioridades (Priority Queue)

Uma fila de prioridades é uma extensão da fila tradicional em que, para cada elemento inserido, uma prioridade p deve ser associada. Por convenção, inteiros positivos são utilizados para representar a prioridade, sendo que quanto menor o inteiro, maior a prioridade (ou seja, p = 2 tem prioridade sobre p = 5).

A principal diferença em relação a fila tradicional é o método enqueue(), que todo elemento deve ser inserido no final da fila, mas a remoção não é feita no início da fila e sim descobrindo o elemento que possui a maior prioridade. Quanto maior a prioridade, menor o inteiro que a codifica.

A primeira coisa que devemos fazer é criar uma classe para definir a estrutura interna de um elemento da fila de prioridades. Isso é necessário pois além de armazenar a informação referente ao elemento em si (um número, uma string, etc), precisamos de uma variável para armazenar o prioridade. O código em Python a seguir implementa uma fila de prioridades.

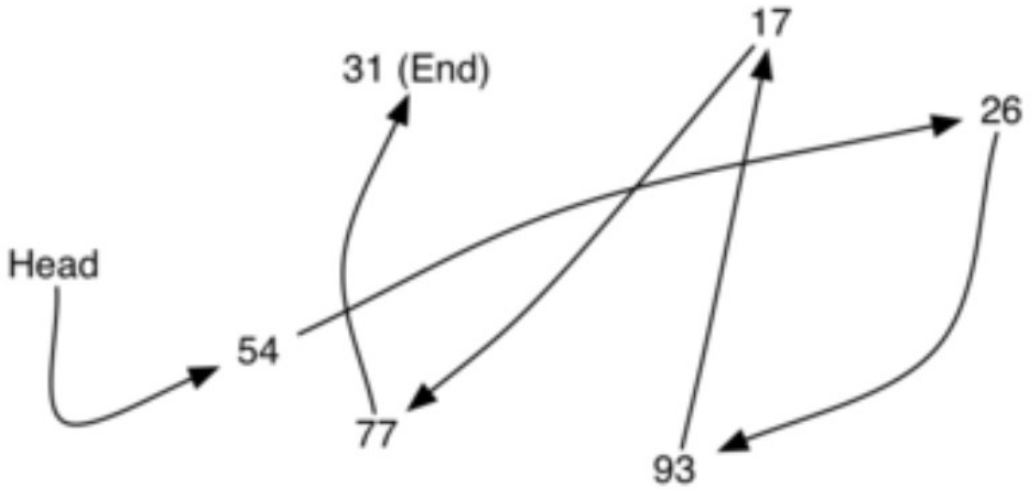
```
# Implementa classe que será utilizada como elemento da fila
class Element:
    def __init__(self, valor, prioridade):
        self.item = valor
        self.prioridade = prioridade

    def __str__(self):
        return str(self.item) + ' ' + str(self.prioridade)

# Implementação da classe fila
class PriorityQueue:

    # Inicia com uma fila vazia
    def __init__(self):
        self.itens = []

    # Verifica se fila está vazia
    def is_empty(self):
        return self.itens == []

    # Retorna o número de elementos da fila
    def size(self):
        return len(self.itens)

    # Adiciona elemento no final da fila com sua prioridade
    def enqueue(self, item, prioridade):
        elemento = Element(item, prioridade)
        self.itens.insert(0, elemento)
        print('ENQUEUE %s' %elemento)
```

```
    # Remove elemento com maior prioridade
    def dequeue(self):
        if self.is_empty():
            print('EMPTY QUEUE')
        else:
            print('DEQUEUE')
            posicao = 0
            menor = self.itens[posicao].prioridade
            # Encontra õ elemento de maior prioridade
            for i in range(self.size()):
                if self.itens[i].prioridade < menor:
                    posicao = i
                    menor = self.itens[i].prioridade
            # Remove elemento da fila
            removido = self.itens.pop(posicao)

            return removido.item

    # Imprime pilha na tela
    def print_queue(self):
        L = []
        for x in self.itens:
            L.append(x.item)
        print(L)


if __name__ == '__main__':

    PQ = PriorityQueue()
    PQ.print_queue()
    PQ.enqueue('a', 4)
    PQ.enqueue('b', 2)
    PQ.enqueue('c', 3)
    PQ.print_queue()
    PQ.dequeue()
    PQ.print_queue()
```

Nesta aula, vimos as principais estruturas de dados lineares: Pilha, Fila, Deque e Fila de prioridades. Na próxima aula estudaremos outra estrutura linear muito importante para a computação, as listas encadeadas. Listas encadeadas diferem de vetores tradicionais em um aspecto primordial: enquanto em um vetor tradicional os elementos subsequentes são armazenados de maneira contígua na memória, em uma lista encadeada, cada nó da estrutura é armazenado independente dos demais e a conexão entre os nós é realizada por um encadeamento lógico.

Aula 6 – Listas Encadeadas

Na linguagem de programação Python é possível utilizar objetos listas de maneira dinâmica e totalmente transparente. Por maneira dinâmica queremos dizer uma lista pode ser facilmente aumentada ou diminuída em tempo de execução a partir da inserção ou remoção de elementos. Em várias linguagens de programação mais antigas, como C, C++ e Pascal, essa característica não é algo transparente, pois precisa ser implementada pelo programador. Tipicamente, em linguagens compiladas, quando criamos uma variável composta como um vetor por exemplo, a alocação de memória é feito em tempo de compilação, o que significa que aquela estrutura deverá ter tamanho fixo durante toda a execução do programa. Na prática, nessas linguagens, não é possível aumentar ou diminuir o tamanho do vetor posteriormente a sua criação. Para permitir esse tipo de flexibilidade, elas oferecem mecanismos para alocação dinâmica de memória. Nesse contexto, as listas encadeadas são estruturas dinâmicas que utilizam um esquema de encadeamentos lógicos sequenciais de modo a permitir que elementos vizinhos não precisem ocupar posições contíguas da memória, como mostrado na figura a seguir. Em Python, é importante estudar listas encadeadas para entendermos de forma completa como as listas realmente funcionam, ou seja, para que possamos compreender todos os processos ocultos que ficam implícitos aos usuários das listas.



Primeiramente, antes de definirmos uma lista encadeada, devemos definir o bloco básico de construção de uma lista: o nó. Cada nó de uma lista encadeada deve conter duas informações: o dado propriamente dito e uma referência para o próximo nó da lista (para quem esse nó aponta). Podemos definir a classe Node, que representa um nó da lista, como segue:

```
class Node:
     # Construtor
     def __init__(self, init_data):
          self.data = init_data
          self.next = None # inicialmente não aponta para ninguém

     # Obtém o dado armazenado
     def get_data(self):
          return self.data

     # Retorna o próximo elemento (para quem nó aponta)
     def get_next(self):
          return self.next
```

```
    # Armazena uma nova informação (atualiza dados)
    def set_data(self, new_data):
        self.data = new_data

    # Aponta o nó para outro nó
    def set_next(self, new_next):
        self.next = new_next
```

Assim, podemos criar um objeto nó como:

```
temp = Node(93)

temp.get_data()

93
```

Internamente, ao criarmos um nó, temos uma representação típica como o diagrama a seguir.

Listas Encadeadas não ordenadas

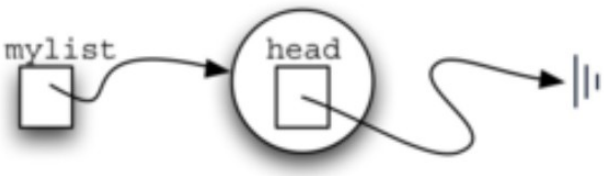
Em uma lista ordenada, a principal característica é que a inserção de um novo nó é feita sempre no início ou no final da lista, ou seja, os elementos do conjunto não encontram-se ordenados. Iniciaremos apresentando como criar uma lista encadeada não ordenada vazia. Adotaremos o seguinte construtor, em que head (cabeça) é uma referência para o primeiro nó da lista:

```
def __init__(self):
    self.head = None
```

Podemos instanciar um objeto dessa classe como:

```
mylist = UnorderedList()
```

o que pode ser representado pelo digrama a seguir:

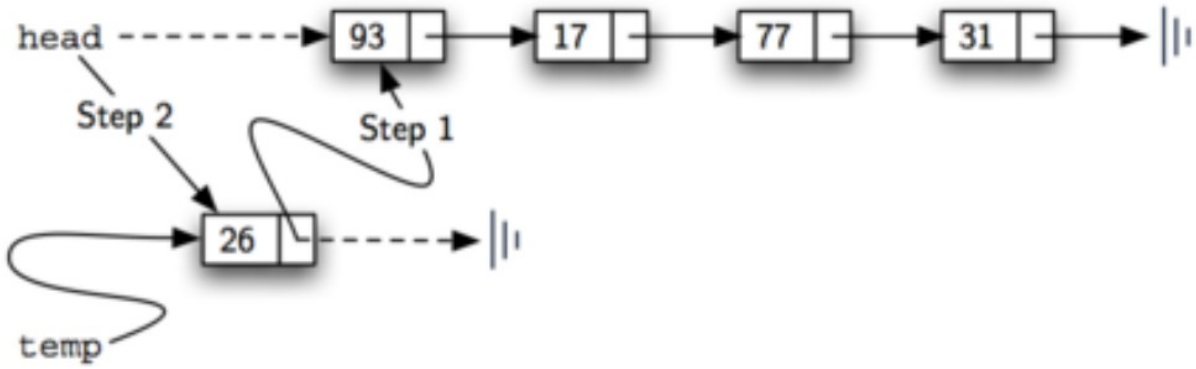


O primeiro método que iremos desenvolver é o mais simples deles. Ele verifica se a lista é vazia, condição que requer que a cabeça da lista seja igual a None.

```
def is_empty(self):
    return self.head == None
```

A próxima função é a responsável por adicionar um elemento no início da lista. A lógica dessa operação consiste em apontar o novo nó para a cabeça da lista (head) e fazer a cabeça da lista apontar para esse novo n´o recém inserido (pois ele será o primeiro elemento da lista). O diagrama a seguir ilustra o resultado da execução dos comandos a seguir:

```
mylist.add_head(31)
mylist.add_head(77)
mylist.add_head(17)
mylist.add_head(93)
mylist.add_head(26)
```

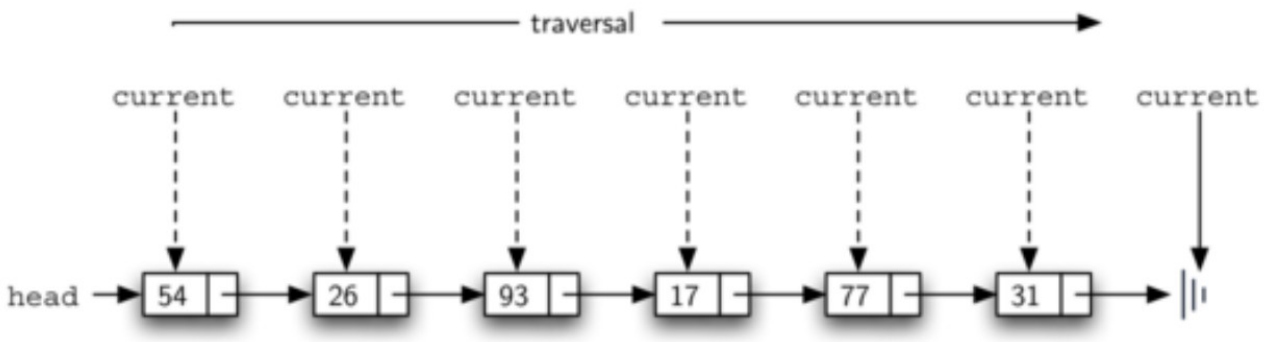


O código fonte a seguir mostra a implementação desse método em Python.

```
def add_head(self, item):
     # Cria novo nó
     temp = Node(item)
     # Aponta novo nó para cabeça da lista
     temp.set_next(self.head)
     # Atualiza a cabeça da lista
     self.head = temp
```

De modo análogo, podemos realizar a inserção no final da lista (tail). Para isso, devemos criar um novo nó e posicionar uma referência no último elemento da lista. Para isso, é preciso apontar temp para a cabeça da lista e percorrer a lista até atingir um nó tal que o próximo elemento seja definido como None. Isso significa que estamos no último elemento da lista.

Ao percorrermos uma lista encadeada, devemos iniciar na cabeça da lista (head) e a cada iteração fazer a referência apontar para o seu sucessor (get_next). A figura a seguir ilustra esse processo.



A função a seguir implementa essa funcionalidade em Python.

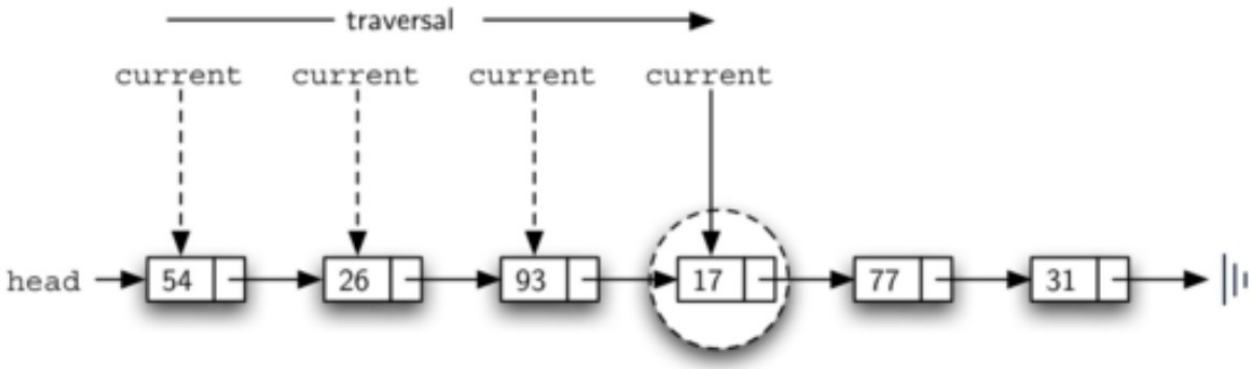
```
def add_tail(self, item):
```

```
      # Cria novo nó
      tail = Node(item)
      # Usa referência temporária para percorer lista (aponta na cabeça)
      temp = self.head
      # Percorre a lista até o último elemento
      while temp.next != None:
            temp = temp.get_next()
      # Aponta tail (ultimo elemento) para novo nó
      temp.set_next(tail)
```

Precisamos ainda implementar uma função para retornar quantos elementos existem na lista encadeada. Como ela é dinâmica, precisamos percorrê-la toda vez que desejarmos contar o número de elementos. A ideia é simples: iniciando na cabeça da lista, devemos percorrer todos os nós, sendo que a cada nó visitado, incrementamos uma unidade no contador. A função a seguir mostra uma implementação em Python.

```
def size(self):
      # Aponta para cabeça da lista
      temp = self.head
      count = 0
      while temp != None:
            count = count + 1
            temp = temp.get_next()

      return count
```

Outra funcionalidade importante consiste em buscar um elemento na lista encadeada, ou seja, verificar se um dado elemento pertence ao conjunto. Para isso, devemos percorrer a lista até encontrar o elemento desejado (e retornar True, uma vez que o elemento desejado pertence a lista), ou até atingir o final da lista (e retornar False, pois o elemento não pertence ao conjunto). A figura a seguir ilustra esse processo.

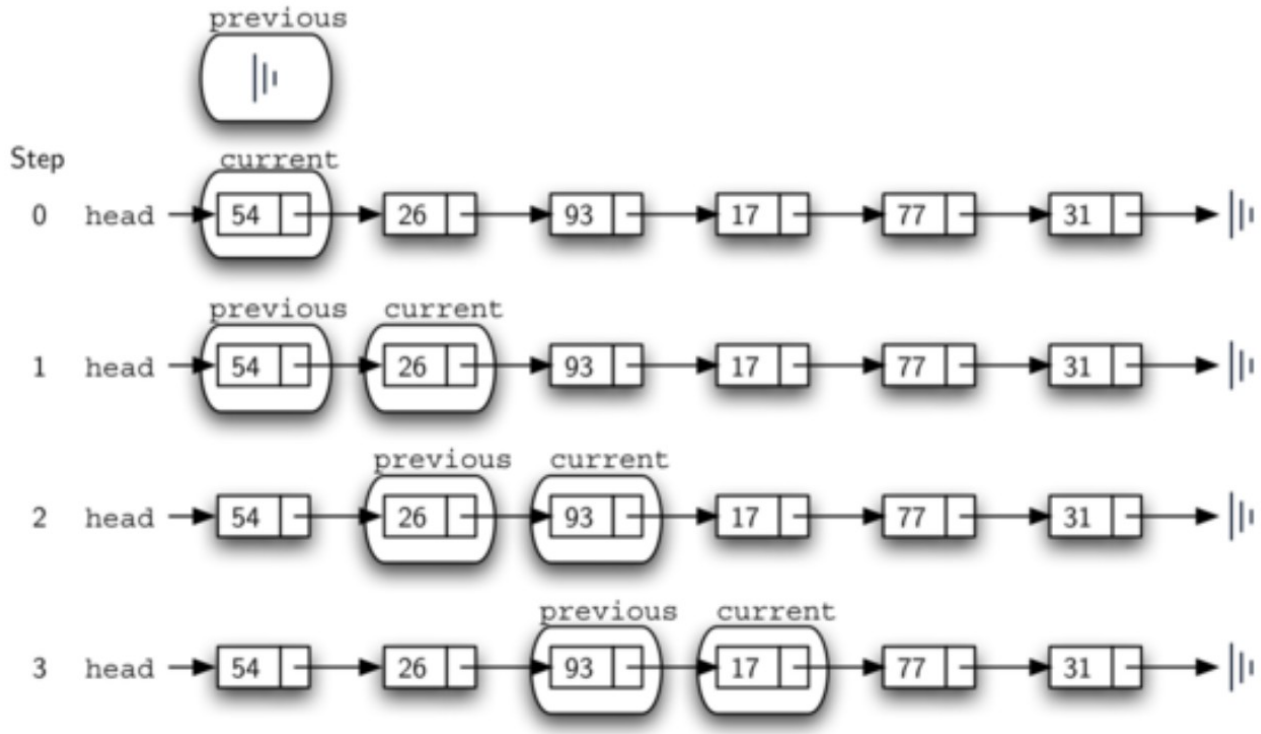


A função a seguir mostra uma implementação em Python.

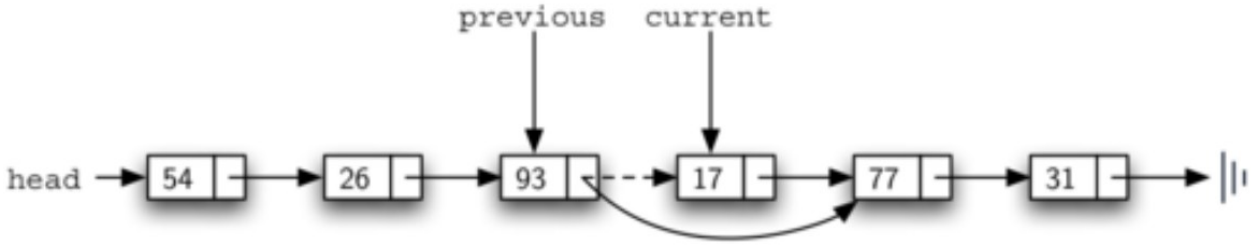
```
# Busca pelo elemento na lista
def search(self,item):
      # Inicia na cabeça da lista
      temp = self.head
      found = False

      # Percorre a lista até achar elemento u chegar no final
      while temp != None and not found:
            # Se achar atual nó contém elemento, found == True
```

```
            if temp.get_data() == item:
                  found = True
            else:
                  # Se não, aponta para o sucessor
                  temp = temp.get_next()

      return found
```

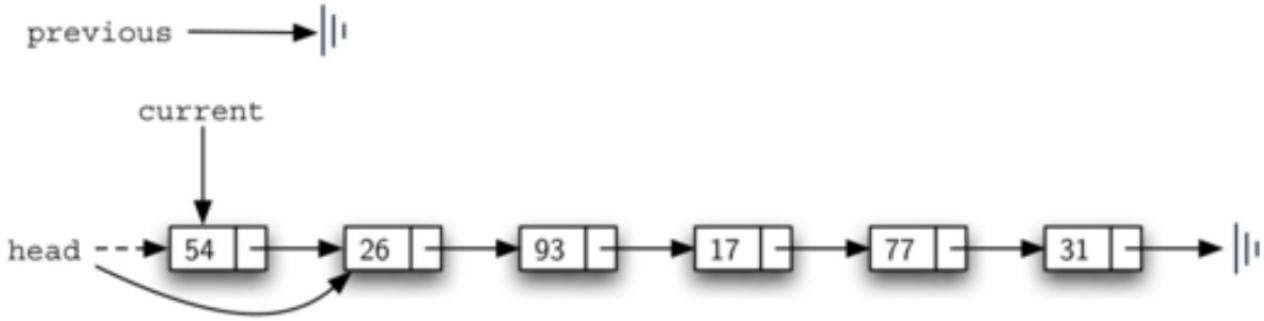
Por fim, uma operação importante é a remoção de um dado elemento da lista encadeada. Note que ao remover um nó da lista, precisamos religar o antecessor com o sucessor, de modo a evitar que elementos fiquem inacessíveis pela quebra do encadeamento sequencial. Primeiramente, preciamos ter duas referências se movendo ao longo da lista: current, que aponta para o elemento corrente da lista encadeada e previous, que aponta para seu antecessor. Eles devem se mover conjuntamente, até que current aponte diretamente para o nó a ser removido. A figura a seguir ilustra o processo.



Em seguida, devemos apontar o valor de next da referência previous para o mesmo local apontado pelo valor de next da referência corrente. Por fim, apontamos o valor de next da referência corrente para None, de modo a excluir completamente o nó da lista encadeada. A figura a seguir mostra uma ilustração gráfica do processo.



Porém, há um caso particular a ser considerado: se o nó que desejamos remover é o primeiro da lista. Neste caso, previous aponta para None, então devemos manipular a referência head, ou seja a cabeça da lista, conforme ilustra a figura a seguir.



O código em Python a seguir apresenta uma implementação para o método remove().

```
# Remove um nó da lista encadeada
def remove(self, item):
     # Aponta a referência corrente para cabeça de L
     current = self.head
     # Aponta referência previous para None
     previous = None
     found = False
     # Enquanto não encontrar o valor a ser removido
     while not found:
          # Se nó corrente armazena o item desejado, encontrou
          if current.get_data() == item:
               found = True
          else:
               # Se no corrente não é o que buscamos
               # Atualiza o previous e o corrente
               previous = current
               current = current.get_next()
     # Se nó a ser removido for o primeiro da lista
     # Altera a cabeça da lista
     if previous == None:
          self.head = current.get_next()
     else:
          # Caso não seja primeiro nó, liga o previous com o próximo
          previous.set_next(current.get_next())
     # Desliga nó corrente
     current.set_next(None)
```

E assim, a classe ListaEncadeada está completa. É interessante notar as complexidades das operações de uma lista encadeada. A função is_empty() e a função add-hear() são O(1), enquanto as funções add_tail(), size(), search() e remove() são todas O(n). A seguir encontra-se a listagem completa do código fonte em Python para a classe UnorderedList.

```
# Implementação da classe nó
class Node:
     # Construtor
     def __init__(self, init_data):
          self.data = init_data
          self.next = None  # inicialmente não aponta para ninguém

     # Obtém o dado armazenado
     def get_data(self):
          return self.data
```

```
     # Retorna o próximo elemento (para quem nó aponta)
     def get_next(self):
          return self.next

     # Armazena uma nova informação (atualiza dados)
     def set_data(self, new_data):
          self.data = new_data

    # Aponta o nó para outro nó
     def set_next(self, new_next):
          self.next = new_next

# Implementa a classe UnorderedList
# (Lista encadeada não ordenada)
class UnorderedList:
    # Construtor (aponta cabeça da lista para None)
    def __init__(self):
        self.head = None

    # Verifica se lista está vazia
    def is_empty(self):
        return self.head == None

    # Adiciona elemento no início da lista
    def add_head(self, item):
        # Cria novo nó
        temp = Node(item)
        # Aponta novo nó para cabeça da lista
        temp.set_next(self.head)
        # Atualiza a cabeça da lista
        self.head = temp

    # Adiciona elemento no final da lista
    def add_tail(self, item):
        # Cria novo nó
        tail = Node(item)
        # Usa referência temporária para percorer lista (inicio=cabeça)
        temp = self.head
        # Percorre a lista até o último elemento
        while temp.next != None:
            temp = temp.next
        # Aponta tail (ultimo elemento) para novo nó
        temp.set_next(tail)

    # Imprime elementos da lista
    def print_list(self):
        # Aponta referência para cabeça
        temp = self.head
        X = []
        # Percorre lista adicionando elementos em X
        while temp != None:
            X.append(temp.data)
            temp = temp.get_next()
        return X
```

```
    # Calcula o número de elementos na lista
    def size(self):
        # Aponta para cabeça da lista
        temp = self.head
        count = 0
        # Percorre lista contando elementos
        while temp != None:
            count = count + 1
            temp = temp.get_next()
        return count

    # Busca pelo elemento na lista
    def search(self,item):
        # Inicia na cabeça da lista
        temp = self.head
        found = False
        # Percorre a lista até achar elemento u chegar no final
        while temp != None and not found:
            # Se achar atual nó contém elemento, found == True
            if temp.get_data() == item:
                found = True
            else:
                # Senão, aponta para o sucessor
                temp = temp.get_next()
        return found

    # Remove um nó da lista encadeada
    def remove(self, item):
        # Aponta a referência corrente para cabeça de L
        current = self.head
        # Aponta referência previous para None
        previous = None
        found = False
        # Enquanto não encontrar o valor a ser removido
        while not found:
            # Se nó corrente armazena o item desejado, encontrou
            if current.get_data() == item:
                found = True
            else:
                # Se no corrente não é o que buscamos
                # Atualiza o previous e o corrente
                previous = current
                current = current.get_next()
        # Se nó a ser removido for o primeiro da lista
        # Altera a cabeça da lista
        if previous == None:
            self.head = current.get_next()
        else:
            # Caso não seja primeiro nó, liga o previous com o próximo
            previous.set_next(current.get_next())
        # Desliga nó corrente
        current.set_next(None)

if __name__ == '__main__':
    # Cria lista vazia
    L = UnorderedList()
```

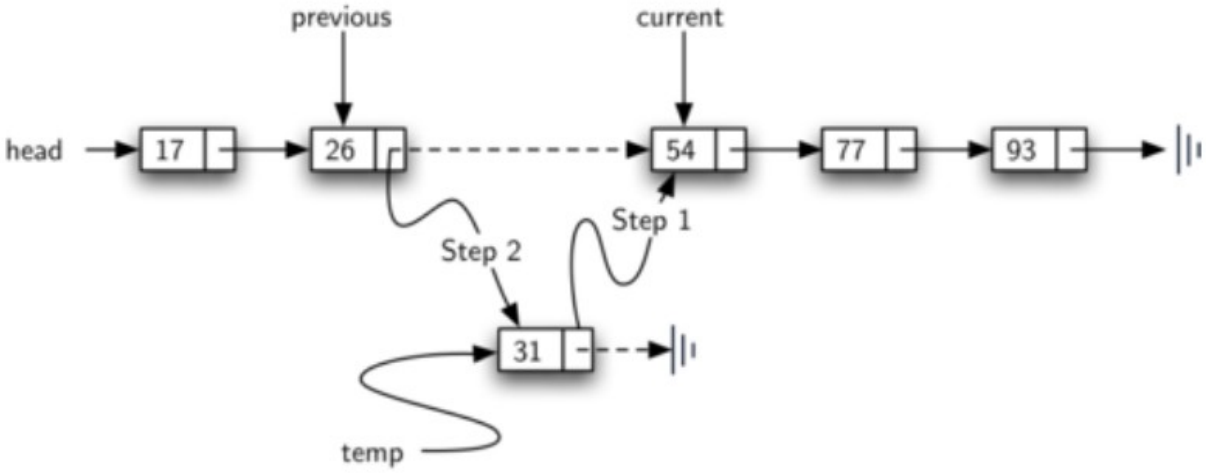
```
print(L.is_empty())
# Insere no início
L.add_head(1)
L.add_head(2)
L.add_head(3)
print(L.print_list())
print(L.size())
print(L.is_empty())
# Insere no final
L.add_tail(4)
L.add_tail(5)
L.add_tail(6)
L.add_tail(12)
print(L.print_list())
print(L.size())
print(L.search(5))
print(L.search(29))
L.remove(5)
print(L.print_list())
print(L.size())
```

Listas ordenadas

Quando trabalhamos com listas ordenadas, os dois métodos que precisam de ajustes em relação as listas encadeadas não ordenadas são search() e add(). Na inserção de um novo nó, devemos primeiramente encontrar sua posição na lista. Na busca pelo elemento, não precisamos percorrer toda a lista encadeada, pois se o elemento buscado é maior que o atual e ainda não o encontramos, significa que ele não pertence a lista. A seguir apresentamos a função que verifica se um elemento faz parte de uma lista ordenada ou não.

```
def search(self, item):
     # Inicio da lista
     current = self.head
     found = False
     stop = False
     # Enquanto não atingir o final da lista e não encontrou e não parou
     while current != None and not found and not stop:
          # Se nó atual é o elemento, encontrou
          if current.get_data() == item:
               found = True
          else:
               Senão, se elemento atual é maior que valor buscado, pare
               if current.get_data() > item:
                    stop = True
               else:
                    # Caso contrário, vai para o próximo
                    current = current.get_next()
     return found
```

Devemos também modificar o método add(), que insere um novo elemento a lista ordenada. A ideia consiste em encontrar a posição correta do elemento na lista ordenada, então para isso é mais fácil iniciar pela cabeça da lista. A figura a seguir ilustra o processo.



Para encontrar a posição correta precisamos de duas referências, assim como na remoção de um elemento. A posição correta da inserção na lista ordenada ocorre exatamente quando o valor da referência prévia é menor que o valor do novo elemento, que por sua vez é menor que o valor da referência atual. Note na figura que 31 está entre 26 e 54.

```
# Adiciona elemento na posição correta da lista
def add(self, item):
      # Inicia na cabeça da lista
      current = self.head
      previous = None
      stop = False

      # Enquanto não chegar no final e não parou
      while current != None and not stop:
            # Se valor do corrente for maior elemento desejado
            # Parar, pois elemento não pertence a lista
            if current.get_data() > item:
                  stop = True
            else:
                  # Senão, move o prévio e o corrente para o próximo
                  previous = current
                  current = current.get_next()
      # Cria novo nó
      temp = Node(item)
      # Se for primeiro elemento, prévio = None
      if previous == None:
            temp.set_next(self.head)
            self.head = temp
      else:
            # Senão, estamnis no meio ou último
            temp.set_next(current)
            previous.set_next(temp)
```

A diferença em relação a complexidade da lista encadeada não ordenada é que na lista ordenada inserção é O(n).

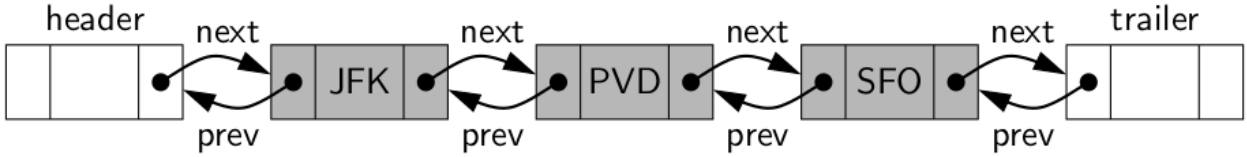
Listas duplamente encadeadas

Conforme vimos anteriormente, a inserção no final de uma lista encadeada tem complexidade O(n). Uma maneira de melhorar essa operação consiste na definição de listas duplamente encadeadas. Em listas duplamente encadeadas, tanto a inserção quanto a remoção no final são operações O(1).

Em uma lista duplamente encadeada, cada nó possui uma informação e duas referências: uma para o nó antecessor e outra para o nó sucessor.

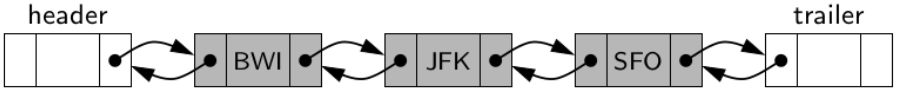
```
class Node:
     # Construtor
     def __init__(self, init_data, prev, next):
          self.data = init_data
          self.prev = prev     # inicialmente não aponta para ninguém
          self.next = next

     # Obtém o dado armazenado
     def get_data(self):
          return self.data

     # Atualiza dado armazenado
     def set_data(self, new_data):
          self.data = new_data
```

Assim como a lista encadeada possui uma cabeça (head) que sempre aponta para o início do encadeamento lógico, uma lista duplamente encadeada possui duas referências especiais: a própria cabeça, que chamaremos de header, e a cauda, que chamaremos de trailer. A figura a seguir ilustra a estrutura de uma lista duplamente encadeada.

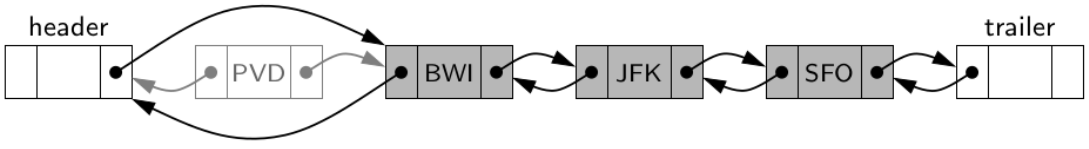


Para essa classe, adotaremos a estratégia de a cada nó inserido, incrementar em uma unidade o seu tamanho e a cada nó removido, decrementar em uma unidade o seu tamanho, assim não precisamos de uma função para contar quantos elementos existem na lista.

Com relação a operação de inserção, a principal diferença em relação a lista encadeada é que aqui devemos ligar o novo nó tanto ao seu elemento sucessor quanto ao seu elemento antecessor, conforme ilustra a figura a seguir.

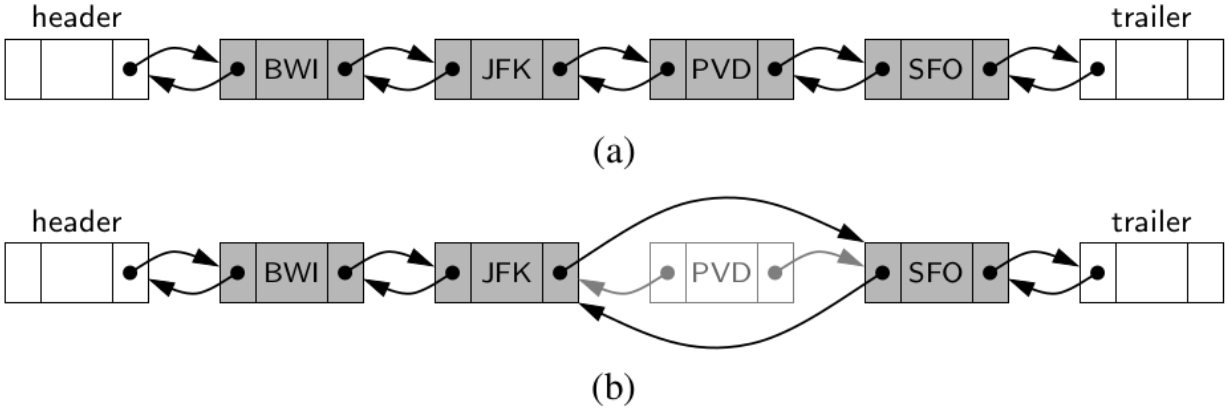


(a)



(b)

A mesma observação vale para a remoção de um nó. Para desconectá-lo completamente da lista duplamente encadeada, devemos ligar o antecessor ao sucessor e vice-versa, conforme ilustra a figura a seguir.



A implementação completa da classe DoubleList em Python encontra-se a seguir.

```
# Implementação da classe nó
class Node:
     # Construtor
    def __init__(self, init_data, prev, prox):
        self.data = init_data
        self.prev = prev        # inicialmente não aponta para ninguém
        self.next = prox

    # Obtém o dado armazenado
    def get_data(self):
        return self.data

    # Atualiza dado armazenado
    def set_data(self, new_data):
        self.data = new_data

# Implementa a classe UnorderedList
# (Lista encadeada não ordenada)
class DoubleList:
    # Construtor (aponta cabeça da lista para None)
    def __init__(self):
        # Cria nós iniciais e finais
        self.header = Node(None, None, None)
        self.trailer = Node(None, None, None)
        # trailer é no final
        self.header.next = self.trailer
        # header é no início
        self.trailer.prev = self.header
        self.size = 0

    # Verifica se lista está vazia
    def is_empty(self):
        return self.size == 0

    # Retorna o número de elementos na lista
    # devemos aplicar a função len()
```

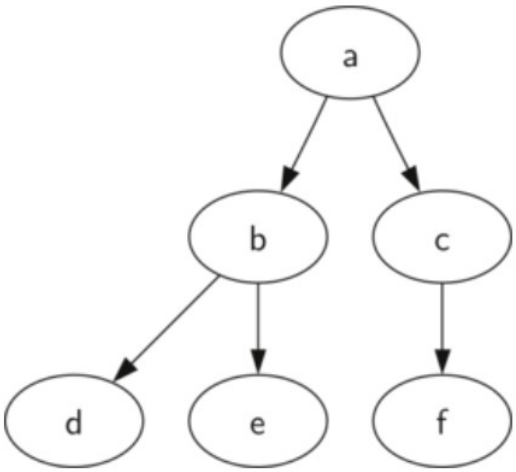
```
def __len__(self):
    return self.size

# Insere novo nó entre dois nós existentes
def insert_between(self, item, predecessor, successor):
    newest = Node(item, predecessor, successor)
    predecessor.next = newest
    successor.prev = newest
    self.size += 1
    return newest

# Remove um nó intermediário da lista
# Header e trailer nunca podem ser removidos!
def delete_node(self, node):
    predecessor = node.prev
    successor = node.next
    predecessor.next = successor
    successor.prev = predecessor
    self.size -= 1
    # Armazena o elemento removido
    element = node.data
    node.prev = node.next = node.element = None
    return element

# Insere elemento no início
def insert_first(self, data):
    # nó deve entrar entre header e header.next
    self.insert_between(data, self.header, self.header.next)

# Insere elemento no final
def insert_last(self, data):
    # nó deve entrar entre trailer.prev e trailer
    self.insert_between(data, self.trailer.prev, self.trailer)

# Remove elemento no início
def delete_first(self):
    if self.is_empty():
        raise Empty('Lista está vazia!')
    return self.delete_node(self.header.next)

# Remove elemento no final
def delete_last(self):
    if self.is_empty():
        raise Empty('Lista está vazia!')
    return self.delete_node(self.trailer.prev)

# Imprime elementos da lista
def print_list(self):
    # Aponta referência para cabeça
    temp = self.header.next
    X = []
    # Percorre lista adicionando elementos em X
    while temp.next != None:
        X.append(temp.data)
        temp = temp.next
    return X
```

```
if __name__ == '__main__':
    # Cria lista vazia
    L = DoubleList()
    print(L.is_empty())
    # Insere no início
    L.insert_first(1)
    L.insert_first(2)
    L.insert_first(3)
    print(L.print_list())
    print(len(L))
    print(L.is_empty())
    # Insere no final
    L.insert_last(4)
    L.insert_last(5)
    L.insert_last(6)
    print(L.print_list())
    print(len(L))
    L.delete_first()
    L.delete_last()
    print(L.print_list())
    print(len(L))
```

Até o presente momento, estudamos estruturas de dados lineares, como listas, pilhas e filas. A partir de agora, iremos estudar estruturas consideradas não lineares, no sentido de que a o encadeamento lógico dos elementos permite organizações mais complexas e otimizadas para a busca. Esse é o caso das árvores binárias, assunto das próximas aulas.

Aula 7 – Árvores binárias

Árvores são estruturas de dados muito utilizadas em diversas áreas da ciência da computação, como sistemas operacionais, bancos de dados e redes de computadores. Uma estrutura de dados do tipo árvore possui uma raiz a partir da qual diferentes ramos conectam um conjunto de nós intermediários até as folhas da árvore. A figura a seguir ilustra uma simples árvore composta por 6 nós, sendo que a é a raiz e d, e, f são as folhas.



A seguir apresentamos uma nomenclatura para as principais componentes de uma árvore.

1. Nós: são as componentes fundamentais de uma árvore, onde a informação é armazenada.
2. Arestas: são as conexões entre os diferentes nós da árvore
3. Raiz: a raiz de uma árvore é o único nó que não possui um nó antecessor, ou seja, não possui nenhuma aresta de entrada, apenas de saída
4. Caminho: é uma sequência ordenada de nós conectados por arestas. Por exemplo, a → b → e
5. Nós filhos: são todos os nós que possuem arestas de entrada proveniente do mesmo no. Por exemplo, os filhos do nó b são d, e
6. Nó pai: um nó é pai de todos os nós conectados a ele pelas arestas de saída. Por exemplo, o nó b é pai dos nós d, e
7. Nós irmãos: são nós que possuem o mesmo pai. Por exemplo, os nós b e c são irmãos
8. Subárvore: uma subárvore é o conjunto de nós e arestas composto por um pai e todos os seus descendentes. A subárvore da direita em relação à raiz, composta por b, d, e, é um exemplo.
9. Nó folha: é todo nó que não possui filhos. Neste caso, os nós d, e, f são folhas
10. Nível de um nó: é o número de arestas no caminho da raiz até o nó em questão. Os nós b, c estão no nível 1 e os nós d, e, f estão no nível 2. A raiz, por definição, pertence ao nível zero.
11. Altura: é o maior nível de um nó pertencente à árvore. Neste exemplo, a altura da árvore é 2.

Sendo assim, podemos definir formalmente uma árvore. Existem diversas formas de se definir o que é uma árvore. Veremos aqui duas delas: uma baseada nas propriedades e outra definição recursiva.

Def: Uma árvore consiste em um conjunto de nós e um conjunto de arestas que conectam pares de nós. Uma árvore possui as seguintes propriedades:

i) Toda árvore tem um nó designado de raiz, por onde a busca, a inserção e a remoção de elementos deve iniciar. Em outras palavras, é a porta de entrada para o conjunto de dados.

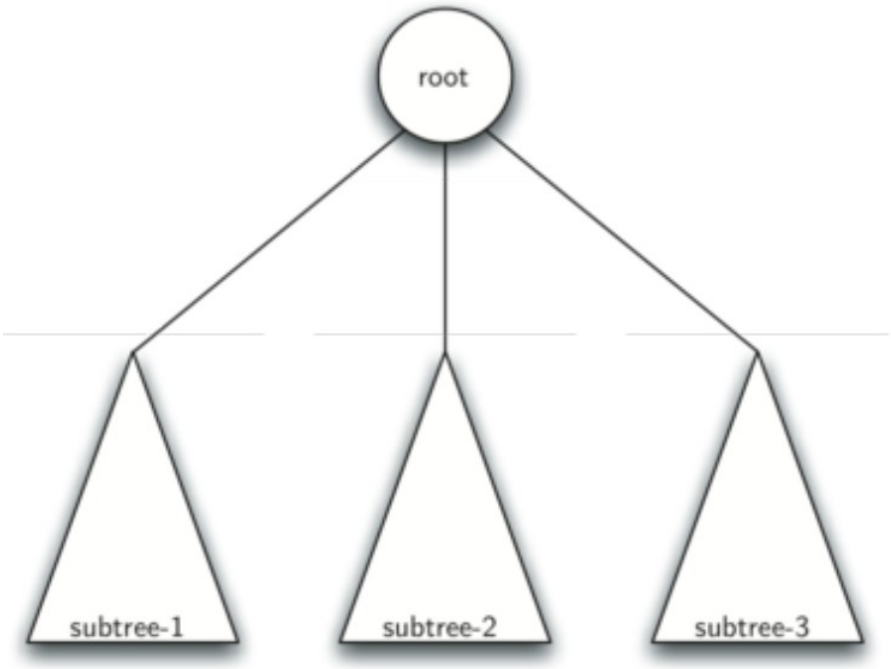
ii) Todo nó n, com exceção da raiz, é conectado por uma aresta a exatamente um nó pai p. Ou seja, cada nó da árvore tem precisamente um único pai.

Iii) Existe um único caminho saindo da raiz e chegando em um nó arbitrário da árvore. Ou seja, sempre que desejarmos acessar um determinado nó, iremos sempre pelo mesmo caminho.

iv) Se cada nó da árvore possui no máximo dois nós filhos, dizemos que a árvore é uma árvore binária.

A seguir apresentamos uma definição recursiva de árvore.

Def: Uma árvore ou é vazia ou consiste de uma raiz com zero ou mais subárvores, cada uma sendo uma árvore. A raiz de cada subárvore é conectada a raiz da árvore pai por uma aresta.



Pela definição recursiva, sabemos que a árvore acima possui pelo menos 4 nós, uma vez que cada triângulo representando uma subárvore deve possuir uma raiz. Na verdade, ela pode ter muito mais nós do que isso, mas não sabemos pois não conhecemos a estrutura interna de cada subárvore.

Nessa aula, por motivos didáticos, o foco será no estudo e implementação de árvores binárias. Existem basicamente duas formas de implementar árvores binárias em Python: utilizando uma lista de sublistas, ou utilizando encadeamento lógico (como na lista encadeada).

Antes de implementarmos uma árvore binária em Python, vamos listar os principais métodos que essa estrutura de dados deve suportar.

Métodos	**Descrição**
BinaryTree()	cria uma nova instância da árvore binária
get_left_child()	retorna a subárvore a esquerda do nó corrente
get_right_child()	retorna a subárvore a direita do nó corrente
set_root_val(val)	armazena um valor no nó corrente
get_root_val()	retorna o valor armazenado no nó corrente
insert_left(val)	cria uma nova árvore binária a esquerda do nó corrente
insert_right(val)	cria uma nova árvore binária a direita do nó corrente

Representação com lista de listas

Em uma representação de lista de sublistas, o valor armazenado na raiz é sempre o primeiro elemento da lista. O segundo elemento da lista será a sublista que representa a subárvore a esquerda. Analogamente, o terceiro elemento da lista será a sublista que representa a subárvore a direita. Para ilustrar como fica a representação da árvore mostrada na figura 1, apresentamos o código a seguir.

```
# Representação de árvore como lista de sublistas
my_tree =   ['a',   # raiz
                ['b',   # subárvore esquerda
                    ['d', []  []],
                    ['e'. [], []]
                ],
                ['c',   # subárvore direita
                    ['f', [], []],
                    []
                ]
            ]
```

Uma propriedade muito interessante desta representação é que a estrutura de uma lista representando uma subárvore possui a mesma estrutura de uma árvore (lista), de modo que define uma representação recursiva.

```
print(my_tree)
print('Subárvore esquerda = ', my_tree[1])
print('Raiz = ', my_tree[0])
print('Subárvore direita = ', my_tree[2])
```

A implementação a seguir ilustra um conjunto de funções que nos permite manipular uma lista de listas de modo a simular o comportamento de uma árvore binária.

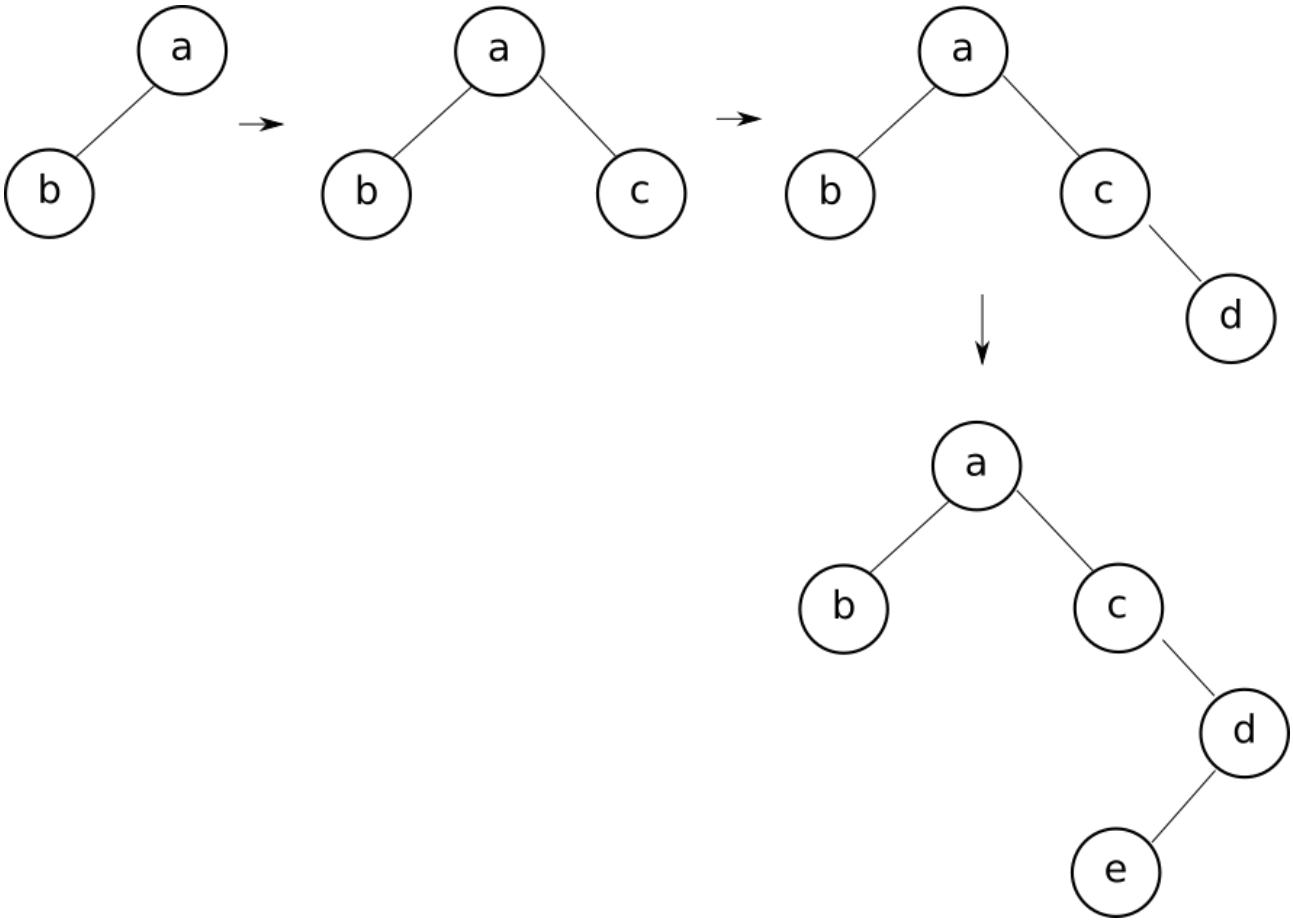
```
# Cria árvore binária com raiz r
def binary_tree(r):
    return [r, [], []]

# Insere novo ramo a esquerda da raiz
def insert_left(root, new_branch):
    # Analisa a subárvore a esquerda
    t = root.pop(1)
    # Se a subárvore a esquerda não é vazia
    if len(t) > 1:
        # Insere na posição 1 da raiz (esquerda)
        # Novo ramo será a raiz da subárvore a esquerda
        # Adiciona t na esquerda do novo ramo
        root.insert(1, [new_branch, t, []])
    else:
        # Se t for vazia, não há subárvore a esquerda
        root.insert(1, [new_branch, [], []])

    return root
# Insere novo ramo a direita da raiz
def insert_right(root, new_branch):
    # Analisa a subárvore a direita
    t = root.pop(2)
    # Se a subárvore a direita não é vazia
```

```
    if len(t) > 1:
        # Insere na posição 2 da raiz (direita)
        # Novo ramo será a raiz da subárvore a direita
        # Adiciona t na direita do novo ramo
        root.insert(2, [new_branch, [], t])
    else:
        # Se t for vazia, não há subárvore a direita
        root.insert(2, [new_branch, [], []])
    return root

def get_root_val(root):
    return root[0]

def set_root_val(root, new_val):
    root[0] = new_val

def get_left_child(root):
    return root[1]

def get_right_child(root):
    return root[2]

if __name__ == '__main__':
    # Cria árvore binária
    r = binary_tree(3)
    # Adiciona subárvore a esquerda
    insert_left(r, 4)
    # Adiciona subárvore a esquerda
    insert_left(r, 5)
    # Adiciona subárvore a direita
    insert_right(r, 6)
    # Adiciona subárvore a direita
    insert_right(r, 7)
    print(r)

    # Obtém subárvore a esquerda da raiz
    l = get_left_child(r)
    print(l)
    # Muda a raiz da subárvore a esquerda
    set_root_val(l, 9)
    print(r)
    # Insere a esquerda da subárvore a esquerda
    insert_left(l, 11)
    print(r)
```

Ex: Considere o seguinte código em Python.

```
x = binary_tree('a')
insert_left(x,'b')
insert_right(x,'c')
insert_right(get_right_child(x), 'd')
insert_left(get_right_child(get_right_child(x)), 'e')
```

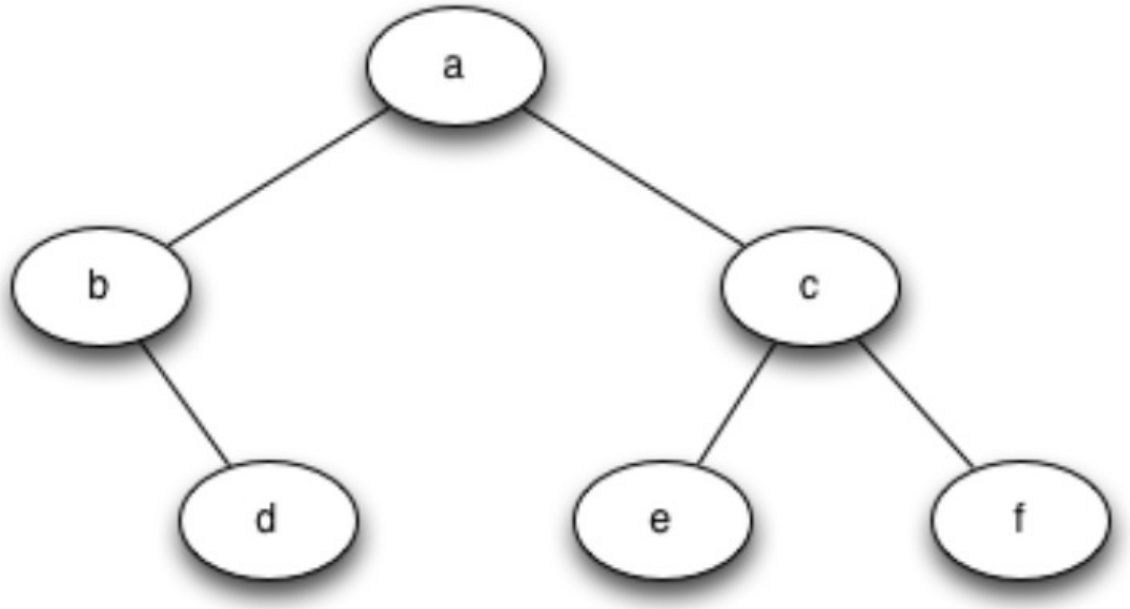
Desenhe a árvore resultante e forneça a representação usando lista de listas.



Representação com lista de listas:

```
['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []]]]
```

Ex: Escreva a representação utilizando lista de listas da árvore a seguir.



Representação com referências

Apesar de interessante, a representação de árvores binárias utilizando lista de listas não é muito intuitiva, principalmente quando o número de nós da árvore cresce. O número de sublistas fica tão elevado que é muito fácil cometer erros e equívocos durante a manipulação da estrutura. Sendo assim, veremos agora, como representar árvores binárias utilizando referências e um encadeamento lógico similar ao adotado nas listas duplamente encadeadas, em que cada nó possui duas referências: uma para o nó filho a esquerda e outra para o nó filho a direita. O código fonte em Python a seguir ilustra uma implementação da classe BinaryTree utilizando referências.

```
# Define a classe árvore binária utilizando referências
class BinaryTree:
    # Construtor
    def __init__(self, valor):
        self.key = valor
        self.left_child = None
        self.right_child = None

    # Insere nó a esquerda
    def insert_left(self, valor):
        # Se nó corrente não tem filho a esquerda, OK
        if self.left_child == None:
            self.left_child = BinaryTree(valor)
        else:
            # Se tem filho a esquerda, pendura subárvore
            # a esquerda do nó corrente na esquerda do
            # novo nó recem criado
            temp = BinaryTree(valor)
            temp.left_child = self.left_child
            self.left_child = temp

    # Insere nó a direita
    def insert_right(self, valor):
        # Se nó corrente não tem filho a direita, OK
        if self.right_child == None:
            self.right_child = BinaryTree(valor)
        else:
            # Se tem filho a direita, pendura subárvore
            # a direita do nó corrente na direita do
            # novo nó recem criado
            temp = BinaryTree(valor)
            temp.right_child = self.right_child
            self.right_child = temp

    # Obtém filho a direita
    def get_right_child(self):
        return self.right_child

    # Obtém filho a esquerda
    def get_left_child(self):
        return self.left_child

    # Atualiza valor do nó corrente
    def set_root_val(self, valor):
        self.key = valor
```

```
    # Obtém valor do nó corrente
    def get_root_val(self):
        return self.key

if __name__ == '__main__':
    # Cria nó com valor 'a'
    r = BinaryTree('a')
    print(r.get_root_val())
    print(r.get_left_child())
    print(r.get_right_child())
    # Insere nó com valor 'b' a esquerda da raiz
    r.insert_left('b')
    print(r.get_left_child().get_root_val())
    # Insere nó com valor 'c' na direita da raiz
    r.insert_right('c')
    print(r.get_right_child().get_root_val())
    # Insere nó com valor 'd' a esquerda no filho a esquerda da raiz
    r.get_left_child().insert_left('d')
    print(r.get_left_child().get_left_child().get_root_val())
    # Insere nó com valor 'e' a direita no filho a esquerda da raiz
    r.get_left_child().insert_right('e')
    print(r.get_left_child().get_right_child().get_root_val())
```

Percorrendo uma árvore

Há basicamente 3 formas de navegar por uma árvore binária: preorder, inorder e postorder.

Preorder: visitamos primeiramente a raiz da árvore, depois recursivamente a subárvore a esquerda e finalmente recursivamente a subárvore a direita.

Inorder: visitamos recursivamente a subárvore a esquerda, depois passamos pela raiz da árvore e por fim visitamos recursivamente a subárvore a direita.

Postorder: visitamos recursivamente a subárvore a esquerda, depois visitamos recursivamente a subárvore a direita e por fim passamos pela raiz.

O código a seguir ilustra uma implementação recursiva do método preorder da classe BinaryTree.

```
# Percorre uma árvore binária em Preorder
def preorder(self):
     # Imprime valor da raiz
     print(self.key)
     # Visita subárvore a esquerda
     if self.left_child:
          self.left.preorder()
     # Visita subárvore a direita
     if self.right_child:
          self.right.preorder()
```

Note que, antes de realizar a chamada recursiva, devemos verificar se as subárvores a esquerda e a direita existem, ou seja, se não são vazias. Essa é a condição de parada da recursão. A seguir apresentamos uma implementação recursiva do método inorder.

```
# Percorre uma árvore binária em Inorder
def inorder(self):
     # Visita subárvore a esquerda
     if self.left_child:
          self.left.inorder()
     # Imprime valor da raiz
     print(self.key)
     # Visita subárvore a direita
     if self.right_child:
          self.right.inorder()
```

Por fim, a seguir apresentamos uma implementação recursiva do método postorder.

```
# Percorre uma árvore binária em Postorder
def postorder(self):
     # Visita subárvore a esquerda
     if self.left_child:
          self.left.postorder()
     # Visita subárvore a direita
     if self.right_child:
          self.right.postorder()
     # Imprime valor da raiz
     print(self.key)
```

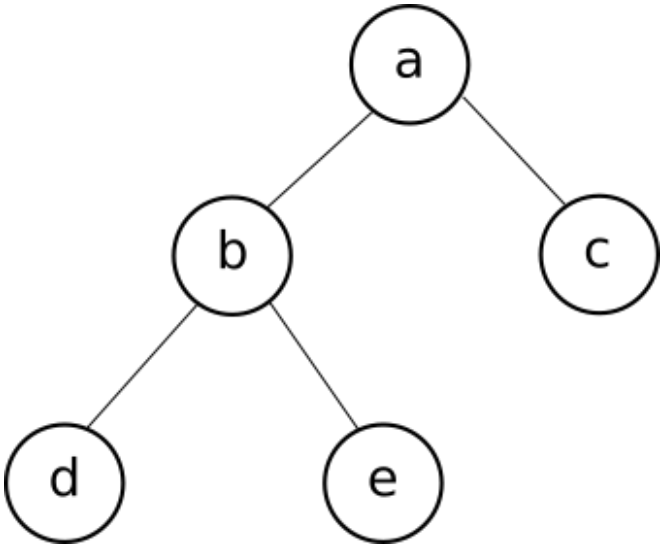
Assim, o código em Python para a classe BinaryTree completa é listado a seguir.

```
# Define a classe árvore binária utilizando referências
class BinaryTree:
    # Construtor
    def __init__(self, valor):
        self.key = valor
        self.left_child = None
        self.right_child = None

    # Insere nó a esquerda
    def insert_left(self, valor):
        # Se nó corrente não tem filho a esquerda, OK
        if self.left_child == None:
            self.left_child = BinaryTree(valor)
        else:
            # Se tem filho a esquerda, pendura subárvore
            # a esquerda do nó corrente na esquerda do
            # novo nó recem criado
            temp = BinaryTree(valor)
            temp.left_child = self.left_child
            self.left_child = temp

    # Insere nó a direita
    def insert_right(self, valor):
        # Se nó corrente não tem filho a direita, OK
        if self.right_child == None:
            self.right_child = BinaryTree(valor)
        else:
            # Se tem filho a direita, pendura subárvore
            # a direita do nó corrente na direita do
            # novo nó recem criado
```

```
        temp = BinaryTree(valor)
        temp.right_child = self.right_child
        self.right_child = temp

# Obtém filho a direita
def get_right_child(self):
    return self.right_child

# Obtém filho a esquerda
def get_left_child(self):
    return self.left_child

# Atualiza valor do nó corrente
def set_root_val(self, valor):
    self.key = valor

# Obtém valor do nó corrente
def get_root_val(self):
    return self.key

# Percorre uma árvore binária em Preorder
def preorder(self):
    # Imprime valor da raiz
    print(self.key)
    # Visita subárvore a esquerda
    if self.left_child:
        self.left_child.preorder()
    # Visita subárvore a direita
    if self.right_child:
        self.right_child.preorder()

# Percorre uma árvore binária em Inorder
def inorder(self):
    # Visita subárvore a esquerda
    if self.left_child:
        self.left_child.inorder()
    # Imprime valor da raiz
    print(self.key)
    # Visita subárvore a direita
    if self.right_child:
        self.right_child.inorder()

# Percorre uma árvore binária em Postorder
def postorder(self):
    # Visita subárvore a esquerda
    if self.left_child:
        self.left_child.postorder()
    # Visita subárvore a direita
    if self.right_child:
        self.right_child.postorder()
    # Imprime valor da raiz
    print(self.key)
```

```
if __name__ == '__main__':
    # Cria nó com valor 'a'
    r = BinaryTree('a')
    print(r.get_root_val())
    print(r.get_left_child())
    print(r.get_right_child())
    # Insere nó com valor 'b' a esquerda da raiz
    r.insert_left('b')
    print(r.get_left_child().get_root_val())
    # Insere nó com valor 'c' na direita da raiz
    r.insert_right('c')
    print(r.get_right_child().get_root_val())
    # Insere nó com valor 'd' a esquerda no filho a esquerda da raiz
    r.get_left_child().insert_left('d')
    print(r.get_left_child().get_left_child().get_root_val())
    # Insere nó com valor 'e' a direita no filho a esquerda da raiz
    r.get_left_child().insert_right('e')
    print(r.get_left_child().get_right_child().get_root_val())
    # Percurso preorder
    print('Preorder')
    r.preorder()
    # Percurso inorder
    print('Inorder')
    r.inorder()
    # Percurso postorder
    print('Postorder')
    r.postorder()
```

Note que a árvore criada no exemplo acima é ilustrada pela figura a seguir.



Como esperado, os percursos obtidos com os métodos preorder, inorder e postorder foram:

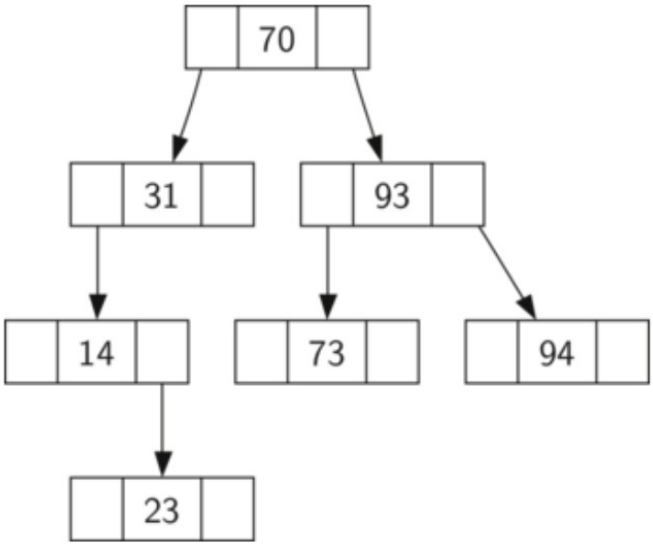
preorder: a, b, d, e, c
inorder: d, b, e, a, c
postorder: d, e, b, c, a

Árvores binárias de busca

Apesar de funcional, a estrutura de dados implementada pela classe BunaryTree não é otimizada para busca de elementos no conjunto. Para essa finalidade, veremos que existe uma classe de árvores mais adequada: as árvores binárias de busca (BynarySearchTree). Uma árvore binária de busca implementa um TAD Map, que é uma estrutura que mapeia uma chave a um valor. Neste tipo de estrutura de dados, nós não estamos interessados na localização exata dos elementos na árvore, mas sim em utilizar a estrutura da árvore binária para realizar busca de maneira eficiente. A seguir apresentamos os principais métodos da classe BinarySearchTree.

Método	Operação
Map()	cria um mapeamento vazio
put(key, val)	adiciona um novo par chave-valor ao mapeamento (se chave já existe, atualiza o valor referente a ela)
get(key)	retorna o valor associado a chave
del map[key]	deleta o par chave-valor do mapeamento
len()	retorna o número de pares chave-valor no mapeamento
in	retorna True se chave pertence ao mapeamento (key in map)

Propriedade chave: em uma árvore binária de busca, chaves menores que a chave do nó pai devem estar na subárvore a esquerda e chaves maiores que a chave do nó pai devem estar na subárvore a direita. Por exemplo, suponha que desejamos criar uma árvore binária de busca com os seguintes valores: 70, 31, 93, 94, 14, 23, 70, nessa ordem.



Antes de definirmos a classe BinarySearchTree, vamos definir a classe TreeNode responsável por organizar todas as informações e métodos relacionados a um nó da arvore de busca.

```
import random

# Classe para armazenar um nó da árvore de busca
class TreeNode:
    # Construtor
    def __init__(self, key, val, left = None, right = None, parent = None):
        self.key = key              # chave
        self.payload = val          # valor
        self.left_child = left      # filho a esquerda
```

```
        self.right_child = right    # filho a direita
        self.parent = parent        # nó pai

    # Verifica se tem filho a esquerda
    def has_left_child(self):
        # Se retornar None, não tem
        return self.left_child

    # Verifica se tem filho a direita
    def has_right_child(self):
        return self.right_child

    # Verifica se nó é filho a esquerda de alguém
    def is_left_child(self):
        # Tem que ter um nó pai e ser filho a esquerda desse nó pai
        return self.parent and self.parent.left_child == self

    # Verifica se é filho a direita
    def is_right_child(self):
        # Tem que ter um nó pai e ser filho a direita desse nó pai
        return self.parent and self.parent.right_child == self

    # Verifica se nó é raiz
    def is_root(self):
        # Raiz não pode ter pai
        return not self.parent

    # Verifica se é nó folha
    def is_leaf(self):
        # Folha não tem'filho a esquerda nem a direita
        return not (self.right_child or self.left_child)

    # Verifica se nó tem algum filho
    def has_any_children(self):
        # Basta ter um filho a direita ou a esquerda
        return self.right_child or self.left_child

    # Verifica se tem ambos os filhos
    def has_both_children(self):
        # Deve ter filho a direita e a esquerda
        return self.right_child and self.left_child

    # Atualiza dados do nó
    def replace_node_data(self, key, value, lc, rc):
        self.key = key              # nova chave
        self.payload = value        # novo valor
        self.left_child = lc        # novo filho a esquerda
        self.right_child = rc       # novo filho a direita
        if self.has_left_child():   # é pai de seu novo filho a esquerda
            self.left_child.parent = self
        if self.has_right_child():  # é pai de seu novo filho a direita
            self.right_child.parent = self

# Implementa a classe BinarySearchTree
class BinarySearchTree:
    # Construtor
    def __init__(self):
        self.root = None
        self.size = 0
```

```
# Retorna número de nós da árvore
def length(self):
    return self.size

# Permite usar a função built-in len() do Python
def __len__(self):
    return self.size

'''Esse método irá checar se a árvore já tem uma raiz. Se ela não
tiver, então será criado um novo TreeNode e ele será a raiz da árvore.
Se uma raiz já existe, então o método chama a função auxiliar _put para
procurar o local correto do elemento na árvore de maneira recursiva.'''
def put(self, key, val):
    # Se raiz existe
    if self.root:
        # Adiciona elemento a partir da raiz (vai achar posição correta)
        self._put(key, val, self.root)
    else:
        # Se não tem raíz, cria novo nó raiz
        self.root = TreeNode(key, val)
    # Incrementa número de nós
    self.size = self.size + 1

# Função auxiliar recursiva para inserção na árvore de busca
def _put(self, key, val, current):
    # Se chave é menor, olha na subárvore a esquerda
    if key < current.key:
        # Se já tem filho a esquerda, dispara função recursiva
        if current.has_left_child():
            self._put(key, val, current.left_child)
        else:
            # Encontrou a posição correta
             current.left_child = TreeNode(key,val,parent=current)
    else:
        # Aqui a chave é maior ou igual, então subárvore a direita
        # Se já tem filho a direita, dispara função recursiva
        if current.has_right_child():
            self._put(key, val, current.right_child)
        else:  # Encontrou a posição correta
            current.right_child = TreeNode(key, val, parent=current)

# Podemos agora sobrecarregar o operador colchetes como em
# T['a'] = 123 para inserir o elemento 123 na chave 'a'
def __setitem__(self, k, v):
    self.put(k, v)

# Busca pelo elemento com a chave key
def get(self, key):
    # Se árvore tem raiz
    if self.root:
        # Dispara função recursiva auxiliar de busca
        res = self._get(key, self.root)
        # Se retorna elemento diferente de None
        if res:
            return res.payload
        else:
            return None
    else:
        return None
```

```
# Função auxiliar recursiva para busca de elemento na árvore
def _get(self, key, current):
    # Se nó corrente não existe, não existe elemento
    if not current:
        return None
    # Se chave do elemento igual a chave de busca, encontrou
    elif current.key == key:
        return current
    # Se chave menor que chave do nó
    elif key < current.key:
        # Buscar na subárvore a esquerda
        return self._get(key, current.left_child)
    else:
        # Buscar na subárvore a direita
        return self._get(key, current.right_child)

# Podemos agora sobrecarregar o operador colchetes como em
# T['a'] para retornar o valor cuja chave é 'a'
def __getitem__(self, key):
    return self.get(key)

# Permite utilizar o operador in para realizar a busca
# como em 'a' in T
def __contains__(self, key):
    if self._get(key, self.root):
        return True
    else:
        return False

# Percorre uma árvore binária em Preorder
def preorder(self, current_node):
    # Imprime valor da raiz
    print(current_node.key)
    # Visita subárvore a esquerda
    if current_node.left_child:
        self.preorder(current_node.left_child)
    # Visita subárvore a direita
    if current_node.right_child:
        self.preorder(current_node.right_child)

# Percorre uma árvore binária em Inorder
def inorder(self, current_node):
    # Visita subárvore a esquerda
    if current_node.left_child:
        self.inorder(current_node.left_child)
    # Imprime valor da raiz
    print(current_node.key)
    # Visita subárvore a direita
    if current_node.right_child:
        self.inorder(current_node.right_child)

# Percorre uma árvore binária em Postorder
def postorder(self, current_node):
    # Visita subárvore a esquerda
    if current_node.left_child:
        self.postorder(current_node.left_child)
    # Visita subárvore a direita
    if current_node.right_child:
        self.postorder(current_node.right_child)
```

```
        # Imprime valor da raiz
        print(current_node.key)

if __name__ == '__main__':
    # Cria nova árvore de busca
    T = BinarySearchTree()
    # Insere 10 elementos na árvore com chaves aleatórias
    for i in range(10):
        chave = random.randint(0, 100)
        # Se a chave ainda não pertence a árvore
        if not chave in T:
            # Armazena o valor de i
            T[chave] = random.random()
    # Percorre a árvore, imprimindo as chaves
    print('Percurso inorder')
    T.inorder(T.root)
```

Note que em uma árvore binária de busca, ao percorrermos os nós inorder, as chaves aparecem ordenadas do menor para o maior.

```
Percurso inorder: 9 24 25 34 35 52 66 67 77 86
```

Uma observação acerca da complexidade das operações é que para a inserção, busca e remoção de nós, pode-se mostrar que no caso médio elas são $O(\log_2 n)$, onde $h = \log_2 n$ representa a altura da árvore binária. Basta notar que o número de nós no nível d da árvore é sempre 2^d. Sendo assim, $n = 2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = 2^h$. Logo, $h = \log_2 n$.
No pior caso, quando a árvore se degenera para uma lista encadeada (quando as chaves são inseridas em ordem crescente), a complexidade das operações torna-se O(n). A principal vantagem das árvores de busca sobre as listas encadeadas é justamente quando os dados estão bagunçados, ou seja, sem uma ordem específica. Em outras palavras, dados armazenados em estruturas de dados do tipo árvore de busca não precisam ser ordenados, pois elas são otimizadas para a recuperação da informação armazenada.

Remoção de elementos

A operação de remoção de nós de uma árvore de busca é bastante complicada, pois exige a análise de diversos casos específicos. Se a árvore tem mais de um nó, o primeiro passo consiste em pesquisamos com a função get() qual é o nó a ser removido. Se a árvore tem apenas um único nó (raiz), devemos nos certificar que ela contém o elemento a ser removido. Em ambos os casos, se o elemento não for encontrado, a operação deve retornar uma mensagem de erro.

```
# Deleta um elemento de um nó da árvore
def delete(self, key):
    # Se árvore contém mais de um nó
    if self.size > 1:
        # Procura pela chave a ser removida
        node_to_remove = self._get(key, self.root)
        # Se encontrou a chave na árvore
        if node_to_remove:
            # Remove nó da árvore (analisar vários casos)
            self.remove(node_to_remove)
            # Decrementa o número de nós da árvore
            self.size = self.size - 1
        else:
```

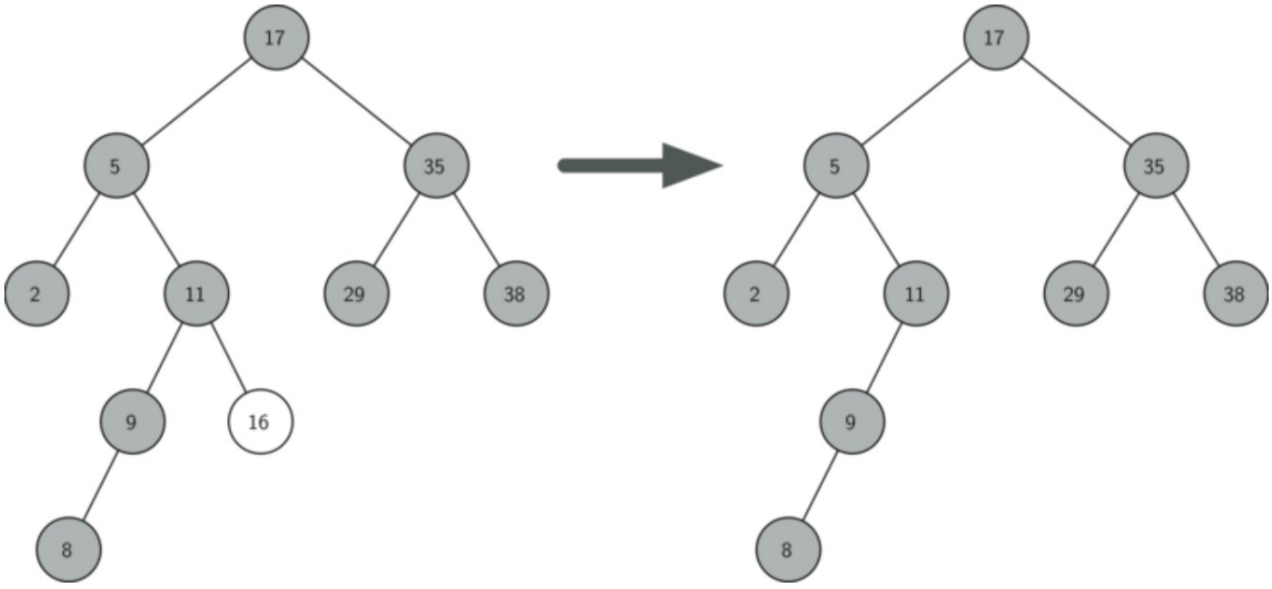
```
            # Senão, retorna erro (chave não existe)
            raise KeyError('Erro, chave não pertence a árvore')
    # Se árvore tem apenas um nó = raiz
    # Verificar se chave a ser removida é = a chave da raiz
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size = self.size - 1
    else:
        # Senão, retorna erro (chave não existe)
        raise KeyError('Erro, chave não pertence a árvore')

# Para utilizar o operador del
def __delitem__(self, key):
    self.delete(key)
```

Agora, o próximo passo consiste na implementação da função remove(). Para isso, existem 3 casos que devem ser analisados:

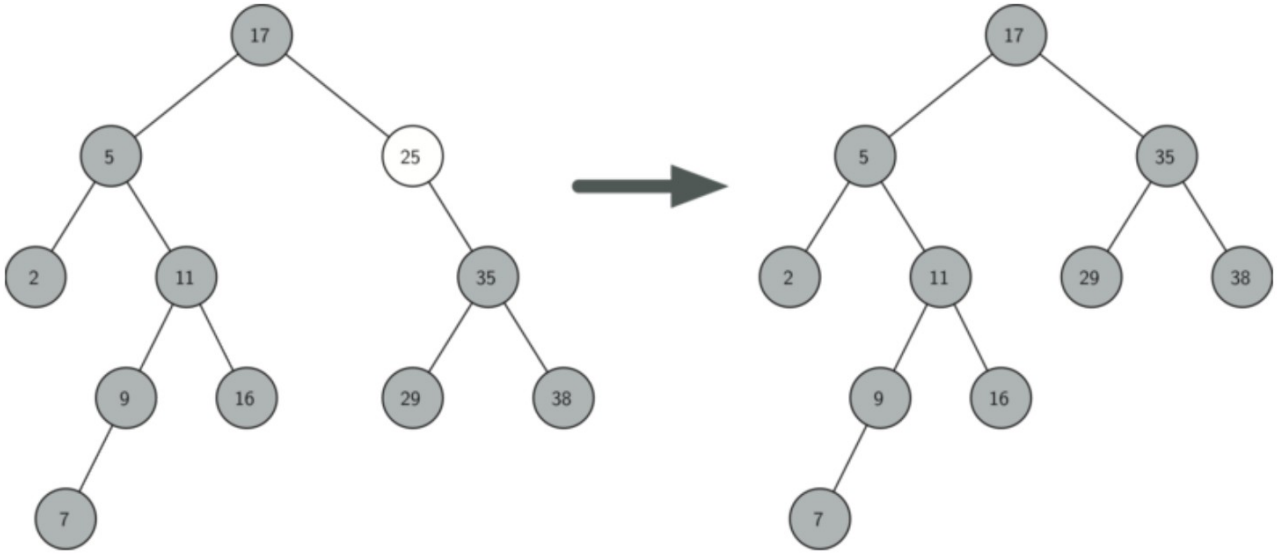
i. o nó a ser removido não tem filhos (é um nó folha);
ii. o nó a ser removido tem apenas um filho;
iii. O nó a ser removido tem dois filhos;

O primeiro caso é trivial pois basta remover a referência a essa folha no nó pai.

```
# Se nó a ser removido é folha
if current_node.is_leaf():
     # Verifica se o nó é um filho a esquerda ou a direita do nó pai
     if current_node == current_node.parent.left_child:
          current_node.parent.left_child = None
     else:
          current_node.parent.right_child = None
```

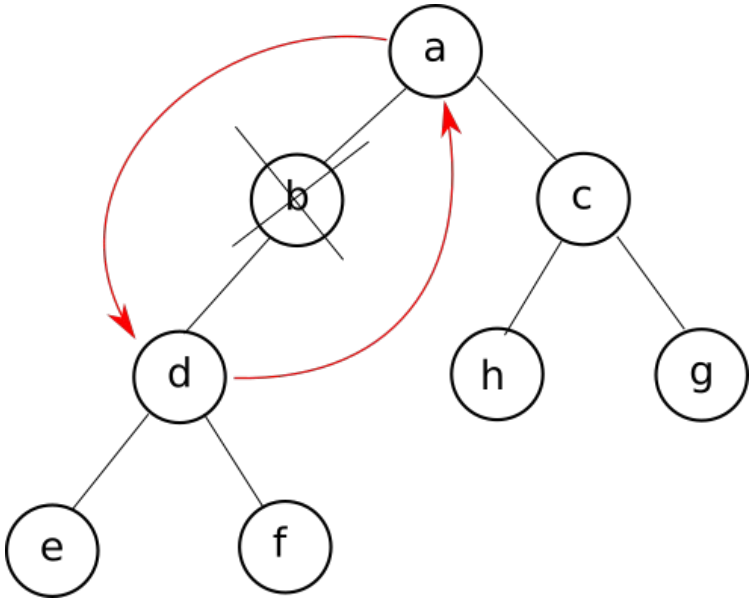


O segundo caso é um pouco mais complicado, mas ainda assim é relativamente fácil de tratar. Se o nó a ser removido possui apenas um único filho, podemos simplesmente promover esse filho para ocupar a posição do pai que será removido, conforme ilustra a figura a seguir.

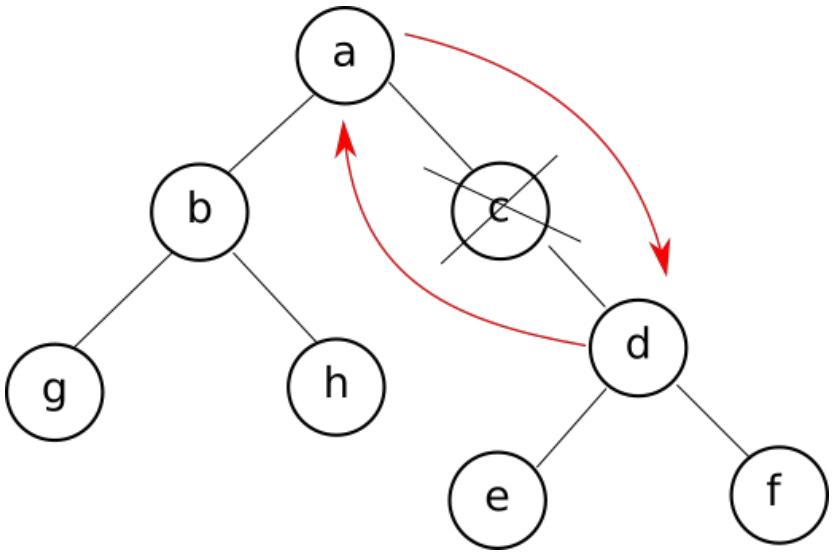


Existem 6 casos a serem tratados, porém como eles são simétricos em relação ao nó corrente ter um filho a esquerda ou um filho a direita, iremos discutir apenas o caso em que o nó corrente possui um filho a esquerda. Então, o processo deve ser o seguinte:

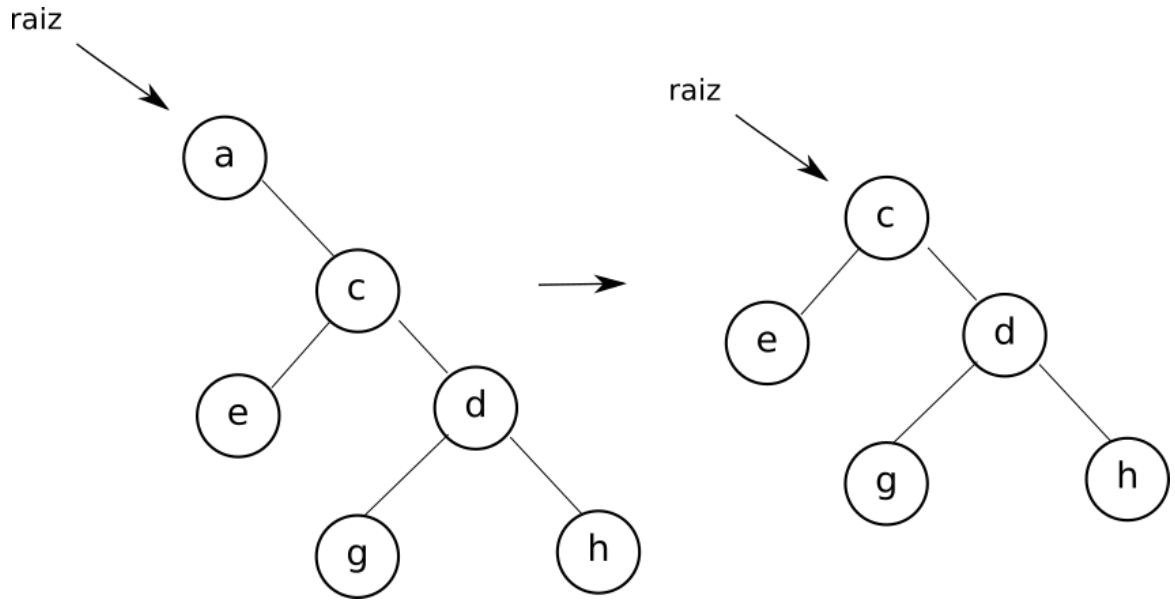
i. Se o nó a ser removido é um filho a esquerda, precisamos apontar a referência parent do nó filho a esquerda para o pai do nó a ser removido e também atualizar a referência left_child do nó pai para apontar para o filho a esquerda do nó a ser removido.



ii. Se o nó a ser removido é um filho a direita, precisamos apontar a referência parent do nó filho a direita para o pai do nó a ser removido e também atualizar a referência right_child do nó pai para apontar para o filho a direita do nó a ser removido.



iii. Se o nó a ser removido não tem pai, então se trata da raiz da árvore. Neste caso, apenas atualizaremos os campos key e payload com os valores dos campos de seu único filho (esquerda ou direita), e atualizamos as referências left_child e right_child para aquelas do seu único filho.



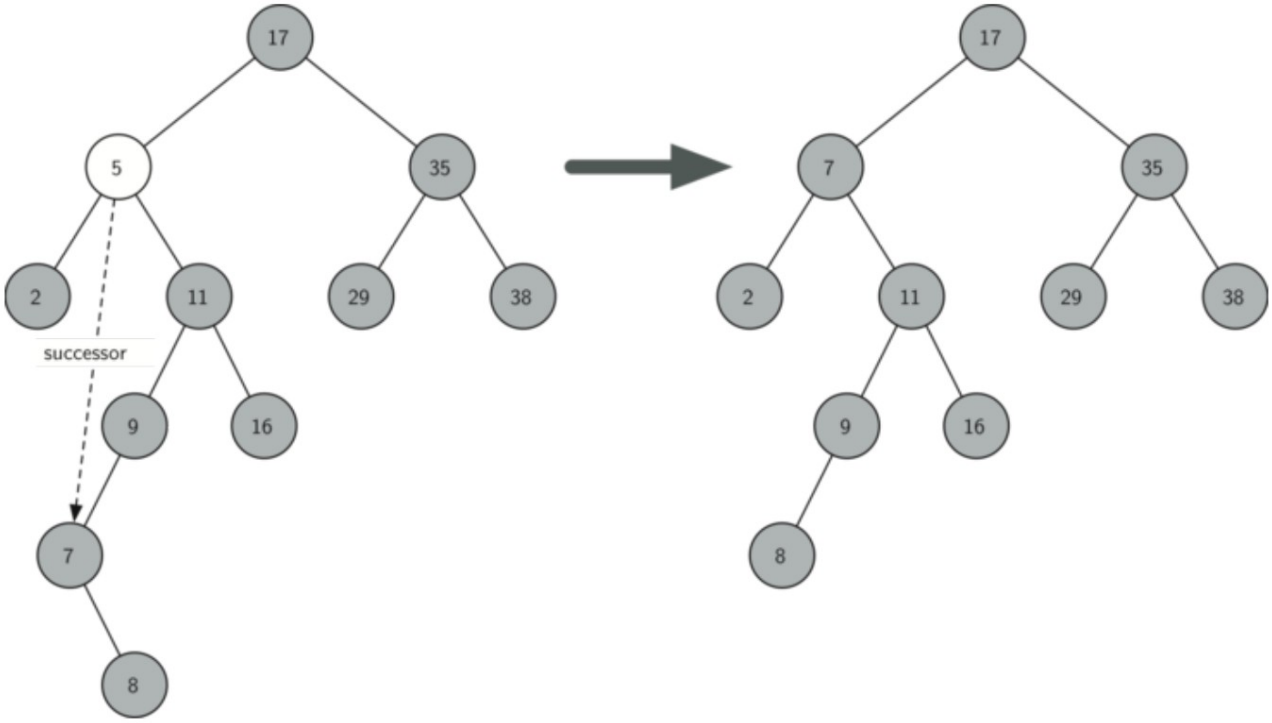
O código em Python a seguir ilustra a implementação dessa operação.

```
# Se nó corrente tem filho a esquerda
if current_node.has_left_child():
     # Se nó corrente é filho a esquerda do nó pai
     if current_node.is_left_child():
          # referência parent do filho esquerdo aponta para o pai do nó
          # corrente a ser removido
          current_node.left_child.parent = current_node.parent
          # referência left_child do pai do nó corrente aponta para o
          # filho esquerdo do nó corrente a ser removido
          current_node.parent.left_child = current_node.left_child
     # Se nó corrente é filho a direita do pai
     elif current_node.is_right_child():
          # referência parent do filho esquerdo aponta para o pai do nó
          # corrente a ser removido
```

```
            current_node.left_child.parent = current_node.parent
            # referência right_child do pai do nó corrente aponta para o
            # filho esquerdo do nó corrente a ser removido
            current_node.parent.right_child = current_node.left_child
      else:
            # Se não tiver filho a esquerda, nem a direita, é raiz
            # Puxa dados do filho a esquerda
            current_node.replace_node_data(current_node.left_child.key,
                                           current_node.left_child.payload,
                                           current_node.left_child.left_child,
                                    current_node.left_child.right_child)
else:  # Se nó corrente tem filho a direita
      # Se nó corrente é filho a esquerda do nó pai
      if current_node.is_left_child():
            # referência parent do filho direito aponta para o pai do nó
            # corrente a ser removido
            current_node.right_child.parent = current_node.parent
            # referência left_child do pai do nó corrente aponta para o
            # filho direito do nó corrente a ser removido
            current_node.parent.left_child = current_node.right_child
      # Se nó corrente é filho a direita do pai
      elif current_node.is_right_child():
            # referência parent do filho direito aponta para o pai do nó
            # corrente a ser removido
            current_node.right_child.parent = current_node.parent
            # referência right_child do pai do nó corrente aponta para o
            # filho direito do nó corrente a ser removido
            current_node.parent.right_child = current_node.right_child
      else:
            # Se não tiver filho a esquerda, nem a direita, é raiz
            # Puxa dados do filho a direita
            current_node.replace_node_data(current_node.right_child.key,
                                           current_node.right_child.payload,
                                    current_node.right_child.left_child,
                                    current_node.right_child.right_child)
```

O terceiro caso é o mais complexo de todos e surge quando o nó a ser removido possui ambos os filhos esquerdo e direito. Nesse caso, devemos procurar na árvore por um nó que possa substituir o nó a ser removido. O nó substituto deve preservar as propriedades da árvore binária de busca tanto para a subárvore a esquerda quanto para a subárvore a direita. O nó que permite preservar essa propriedade é o nó que possui com a menor chave que seja maior que a chave do nó corrente (a ser removido). Denotaremos esse nó especial como o sucessor (*successor*) e iremos discutir como encontrá-lo mais adiante. Uma propriedade interessante é que o *successor* tem no máximo um único filho (0 ou 1 filho), de modo que já sabemos como removê-lo utilizando os dois casos discutidos previamente. Após a remoção do *successor*, nós simplesmente o colocamos no lugar do nó a ser removido.

Uma ilustração gráfica do processo pode ser visualizada na figura a seguir, em que o nó a ser removido é o 5, e o *successor* neste caso é o nó 7 (menor chave que seja maior que o nó a ser removido). As informações do nó 7 são copiadas para o nó 5 e ele então é removido da árvore.



Sendo assim, o código em Python para tratar a remoção de um nó com dois filhos segue abaixo.

```
if current_node.has_both_children():
      succ = current_node.find_successor()
      succ.splice_out()
      current_node.key = succ.key
      current_node.payload = succ.payload
```

Note que estamos utilizando duas funções auxiliares: find_successor() e splice_out(). Primeiro, vamos discutir o funcionamento do método para encontrar o sucessor. Essa função é implementada como um método da classe TreeNode e utiliza as propriedades das árvores binárias de busca. Note que se estamos neste caso, é porque o nó corrente (que deve ser removido) possui obrigatoriamente dois filhos (um a esquerda e outro a direita). Sendo assim, o elemento que desejamos é o que possui a menor chave dentre todos pertencentes a subárvore a direita do nó corrente. Veja que na figura anterior, o nó com chave 7 é o menor elemento da subárvore a direita. O código em Python a seguir implementa o método em questão.

```
def find_successor(self):
      succ = self.right_child.find_min()
```

A função find_min() simplesmente visita o filho a esquerda até que um nó não tenha mais filho a esquerda. Note que pela definição de árvore binária de busca, esse será o menor elemento da subárvore a direita.

```
def find_min(self):
      current = self
      while current.has_left_child():
            current = current.left_child
      return current
```

Após encontrar o menor elemento da subárvore a direita, devemos removê-lo. Para isso, utilizamos a função splice_out(), definida a seguir.

```
# Remove o nó successor
def splice_out(self):
      # Se é nó folha
      if self.is_leaf():
            # Se é filho a esquerda
            if self.is_left_child():
                  self.parent.left_child = None
            else:
                  # senão, é filho a direita
                  self.parent.right_child = None
      # Caso o nó tenha algum filho
      elif self.has_any_children():
            # Se nó tem filho a direita
            if self.has_left_child():
                  # Se está a esquerda do pai
                  if self.is_left_child():
                        self.parent.left_child = self.left_child
                  else:
                        # Senão, está a direita do pai
                        self.parent.right_child = self.left_child
                  self.left_child.parent = self.parent
            else:
                  # Senão, o nó tem filho a direita
                  Se está a esquerda do pai
                  if self.is_left_child():
                        self.parent.left_child = self.right_child
                  else:
                        # Senão, está a direita do pai
                        self.parent.right_child = self.right_child
                  self.right_child.parent = self.parent
```

Dessa forma, a função remove completa é apresentada a seguir. Note que ela é a união de todos os casos discutidos previamente. Esse é um método da classe BinarySearchTree.

```
def remove(self, current_node):
      if current_node.is_leaf(): #leaf
            if current_node == current_node.parent.left_child:
                  current_node.parent.left_child = None
            else:
                  current_node.parent.right_child = None
      elif current_node.has_both_children(): # interior
            succ = current_node.find_successor()
            succ.splice_out()
            current_node.key = succ.key
            current_node.payload = succ.payload
      else: # this node has one child
            if current_node.has_left_child():
                  if current_node.is_left_child():
                     current_node.left_child.parent = current_node.parent
                     current_node.parent.left_child = current_node.left_child
                  elif current_node.is_right_child():
                     current_node.left_child.parent = current_node.parent
                     current_node.parent.right_child = current_node.left_child
                  else:
                     current_node.replace_node_data(current_node.left_child.key,
                                                    current_node.left_child.payload,
                                                 current_node.left_child.left_child,
                                                current_node.left_child.right_child)
```

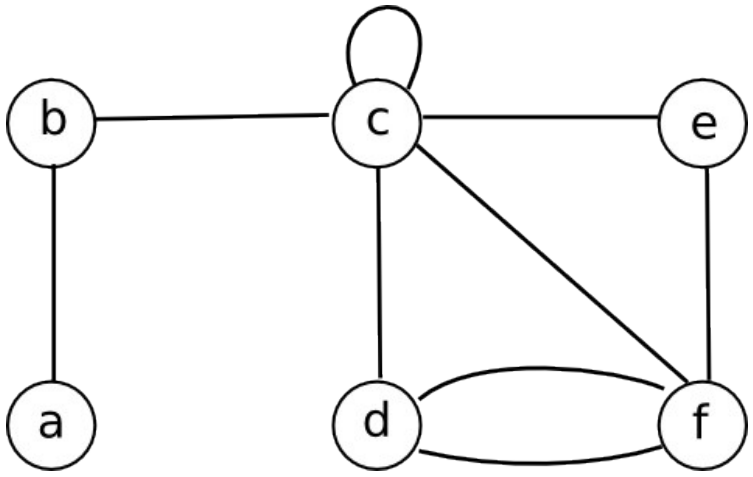
```
else:
      if current_node.is_left_child():
         current_node.right_child.parent = current_node.parent
         current_node.parent.left_child =
         current_node.right_child
      elif current_node.is_right_child():
         current_node.right_child.parent = current_node.parent
         current_node.parent.right_child = current_node.right_child
      else:
        current_node.replace_node_data(current_node.right_child.key,
                                       current_node.right_child.payload,
                                    current_node.right_child.left_child,
                                   current_node.right_child.right_child)
```

Aula 8 – Grafos: Fundamentos básicos

Grafos são estruturas matemáticas que representam relações binárias entre elementos de um conjunto finito. Em termos gerais, um grafo consiste em um conjunto de vértices que podem estar ligados dois a dois por arestas. Se dois vértices são unidos por uma aresta, então eles são vizinhos. É uma estrutura fundamental para a computação, uma vez que diversos problemas do mundo real podem ser modelados com grafos, como encontrar caminhos mínimos entre dois pontos, alocação de recursos e modelagem de sistemas complexos.

Def: G = (V,E) é um grafo se:
i) V é um conjunto não vazio de **vértices**
ii) $E \subseteq V \times V$ é uma relação binária qualquer no conjunto de vértices (não precisa ser apenas equivalência ou ordem parcial, pode ser qualquer coisa): conjunto de **arestas**

Obs: Convenção
Grafo não direcionado: (a,b) = (b,a)
Grafo direcionado ou dígrafo: <a,b> != <b,a>



Grafo ou Multigrafo
- Existem loops
- Existem arestas paralelas

Grafo básico simples
- Não existem loops
- Não existem arestas paralelas

Denotamos por N(v) o conjunto vizinhança do vértice v. Por exemplo, N(b) = {a, c}

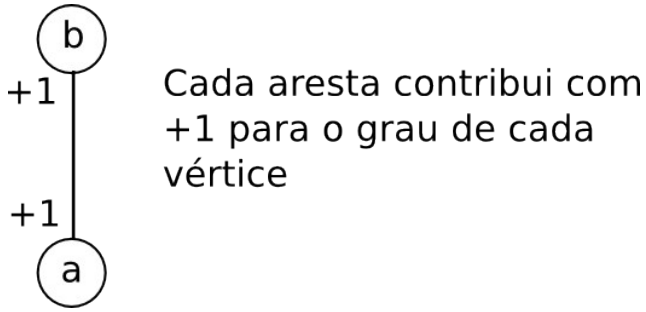
Def: Grau de um vértice v: d(v)
É o número de vezes que um vértice v é extremidade de uma aresta. Num grafo básico simples, é o mesmo que o número de vizinhos de v. Ex: d(a) = 1, d(b) = 2, d(c) = 6, d(d) = 3, …

Lista de graus de G: graus dos vértices de G em ordem crescente
L_G = (1, 2, 2, 3, 4, 6)

Handshaking Lema: A soma dos graus dos vértices de G é igual a duas vezes o número de arestas.

$$\sum_{i=1}^{n} d(v_i) = 2m$$ (condição de existência para grafos)

onde n = |V| e m = |E| denotam respectivamente o número de vértices e arestas.



Prova: (por indução)
Seja o predicado P(n) definido como:

$$P(n): \sum_{i=1}^{n} d(v_i) = 2m$$

BASE: Note que nesse caso, n = 1, o que implica dizer que temos um único vértice. Então, para todo $m \geq 0$ (número de arestas), a soma dos graus será sempre um número par pois ambas as extremidades das arestas incidem sobre o único vértice de G. (OK)

PASSO DE INDUÇÃO: Para k arbitrário, mostrar que P(k) → P(k+1)

Note que para k vértices, temos:

$$P(k): \sum_{i=1}^{k} d(v_i) = 2m$$

Ao adicionarmos exatamente um vértice a mais, temos k + 1 vértices:

$$P(k+1): \sum_{i=1}^{k+1} d(v_i) = 2m'$$

Ao passar de k para k + 1 vértices, temos duas opções: a) o grau do novo vértice é zero; b) o grau do novo vértice é maior que zero.

Caso a): Nessa situação, temos o número de arestas permanece inalterado, ou seja, m = m'. Pela hipótese de indução e sabendo que o grau no novo vértice é zero, podemos escrever:

$$\sum_{i=1}^{k+1} d(v_i) = \sum_{i=1}^{k} d(v_i) + d(v_{k+1}) = 2m + 0 = 2m' \qquad \text{(OK)}$$

ou seja, P(k + 1) é válida.

Caso b): Nessa situação, temos que o número de arestas m' > m. Seja m' = m + a, onde a denota o número de arestas adicionadas ao inserir o novo vértice k + 1. Então, pela hipótese de indução e sabendo que o grau do novo vértice será a, podemos escrever:

$$\sum_{i=1}^{k+1} d(v_i) = \sum_{i=1}^{k} d(v_i) + d(v_{k+1}) + \underbrace{1 + 1 + 1 + \ldots + 1}_{\text{a vezes } 1}$$

pois para cada extremidade das arestas no novo vértice, haverá outra extremidade em algum outro vértice. Isso implica em:

$$\sum_{i=1}^{k+1} d(v_i) = 2m + a + a = 2m + 2a = 2(m+a) = 2m' \qquad \text{(OK)}$$

ou seja, P(k + 1) é valida. Note que mesmo que alguma das arestas inseridas possuam ambas as extremidades no novo vértice k + 1, a soma dos graus também será igual a 2m + 2 a, o que continuará validando P(k + 1). Portanto, a prova está concluída.

Teorema: Em um grafo G = (V, E) o número de vértices com grau ímpar é sempre par.
Podemos particionar V em 2 conjuntos: P (grau par) e I (grau ímpar). Assim,

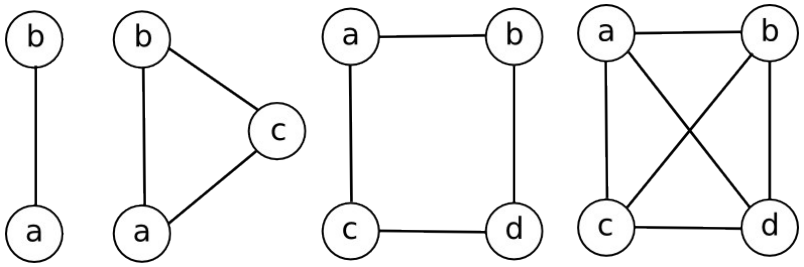
$$\sum_{i=1}^{n} d(v_i) = \sum_{v \in P} d(v) + \sum_{u \in I} d(u) = 2m$$

Isso implica em

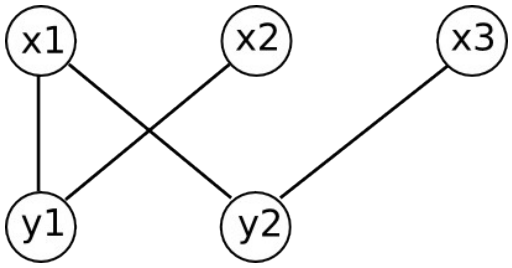
$$\sum_{u \in I} d(u) = 2m - \sum_{v \in P} d(v)$$

Como 2m é par e a sima de números pares é sempre par, resulta que a soma dos números ímpares também é par. Para que isso ocorra temos que ter |I| par (número de elementos do conjunto I é par)

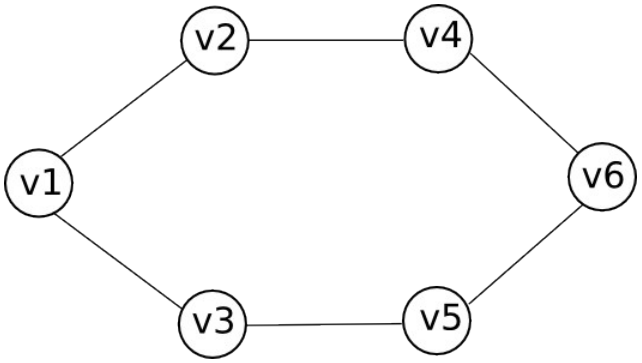
Def: G é k-regular $\Leftrightarrow$ $\forall v \in V\,(d(v) = k)$, ou seja, L_G = (k, k, k, k,..., k)



Def: Grafo Bipartido
G = (V, E) é bipartido $\Leftrightarrow$ $V = X \cup Y$ com $X \cap Y = \emptyset$ tal que
$\forall e \in E(e = (a,b) / a \in X \wedge b \in Y)$



Ex: O grafo a seguir é bipartido ou não? Justifique sua resposta

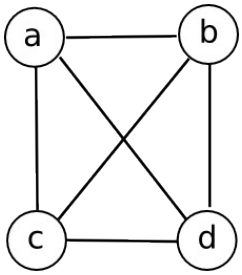


SIM, é bipartido!
Mesmo parecendo que não.

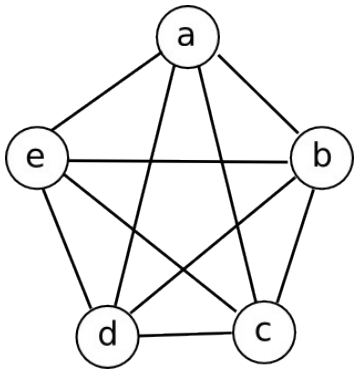
Como decidir se grafo G é bipartido? Seja R = {0, 1} o conjunto de rótulos e seja $r \in R$ um rótulo arbitrário e $\bar{r} \in R$ o seu complementar, ou seja, $\bar{r} = 1 - r$.

1) Escolha um vértice inicial v e rotule-o como r .
2) Para todos os vértices u vizinhos de v ainda não rotulados, rotule-os como $\bar{r}$. Ou seja, se um vértice recebe rótulo r , todos seus vizinhos não rotulados devem receber rótulo $\bar{r}$ e vice-versa.
3) Condição de parada: Pare quando todos os vértices do grafo estiverem rotulados.
4) Se ao fim do processo toda aresta do grafo for do tipo $(r, \bar{r})$ então o grafo G é bipartido. Caso contrário, o grafo não é bipartido.

Def: Grafo completo
G é um grafo completo de n vértices, denotado por K_n, se cada vértice é ligado a todos os demais, ou seja, se L_G = (n-1, n-1, n-1,..., n-1)



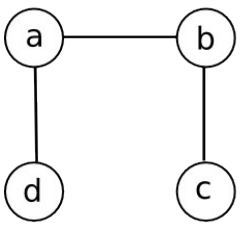
K4



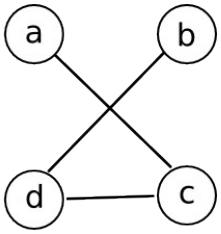
K5

O número de arestas do grafo K_n é dado por $\binom{n}{2} = \frac{n!}{(n-2)!\,2!} = \frac{n(n-1)}{2}$

Def: Complementar de um grafo G: $\bar{G} = K_n - G$



G



$\bar{G}$

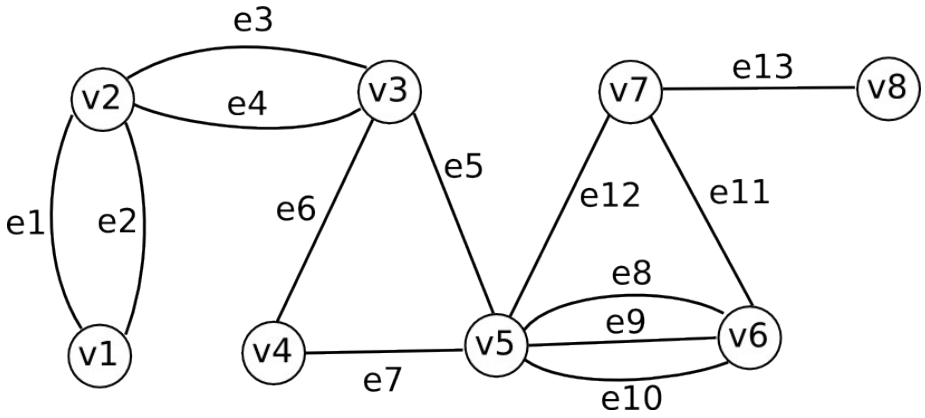
Obs: $G + \bar{G} = K_n$

Def: Subgrafo
Seja G = (V, E) um grafo. Dizemos que H = (V', E') é um subgrafo de G se $V' \subseteq V$ e $E' \subseteq E$

Em outras palavras, é todo grafo que pode ser obtido a partir de G através de remoção de vértices e/ou arestas.

Def: Subgrafos disjuntos não possuem vértices em comum
Subgrafos arestas disjuntos: não possuem aresta em comum

Caminhos e ciclos



a) Passeio: não tem restrição alguma quanto a vértices e arestas

P = v1 e1 v2 e1 v1 e1 v2

b) Trilha: não há repetição de arestas

T = v1 e1 v2 e3 v3 e4 v2

Se trilha é fechada, temos um circuito. Ex: T = v1 e1 v2 e3 v3 e4 v2 e2 v1

c) Caminho: não há repetição de vértices

C = v1 e1 v2 e3 v3 e6 v4 e7 v5

Se caminho é fechado, temos um ciclo

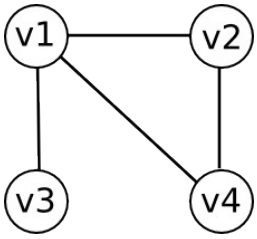
Obs: O comprimento/tamanho de um caminho/trila/passeio é o número de arestas percorridas
Obs: O caminho/trilha/passeio trivial é aquele composto por zero arestas

Representações computacionais de grafos

1) Matriz de adjacências A: matriz quadrada n x n definida como:

a) Grafos básicos simples

$$A_{i,j} = \begin{cases} 1, & i \leftrightarrow j \\ 0, & c.c \end{cases}$$

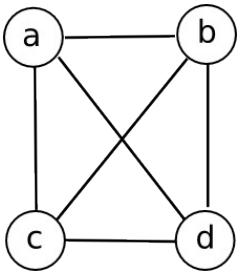


$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Propriedades básicas

i) diag(A) = 0
ii) matriz binária
iii) $A = A^T$ (com exceção de dígrafos)
iv) $\sum_j A_{i,j} = d(v_i)$
v) Esparsa
vi) $O(n^2)$ em espaço

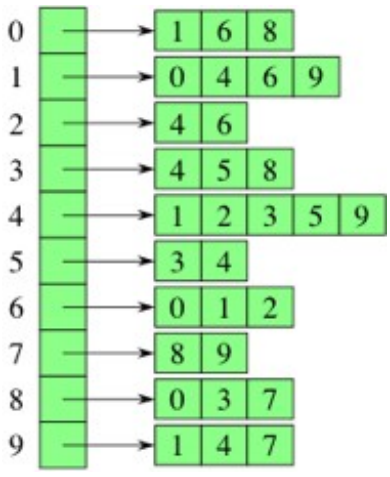
2) Matriz de Incidência M: matriz n x m em que as linhas referem-se aos vértices e as colunas referem-se as arestas



$$M = \begin{array}{c} \begin{matrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} a \\ b \\ c \\ d \end{matrix}$$

Obs: Loops são indicados pelo número 2 no vértice em questão

3) Lista de adjacências: estrutura do tipo hash-table (mais eficiente em termos de memória)



OBS: Em Python, na biblioteca NetworkX a classe Graph implementa grafos básicos simples e a classe MultiGraph implementa multigrafos (permitem arestas paralelas).

A estrutura de dados básica utilizada é um dicionário de dicionários de dicionários, que simula uma tabela hash na forma de uma lista de adjacências. As chaves do dicionário são os nós de modo que G[u] retorna um dicionário cuja chave é o extremo da respectiva aresta e o campo valor é um dicionário para os atributos da aresta. A expressão G[u][v] retorna o dicionário que armazena os atributos da aresta (u,v). O código em Python a seguir mostra como podemos criar e manipular um grafo utilizando a biblioteca NetworkX.

```
# Classes e funções para manipulação de grafos
import networkx as nx
# Classes e funções para plotagem de gráficos
import matplotlib.pyplot as plt

# Cria grafo vazio
G = nx.Graph()

# Adiciona vértices
G.add_node('v1')
G.add_node('v2')
G.add_node('v3')
G.add_node('v4')
G.add_node('v5')

# Adiciona arestas
G.add_edge('v1', 'v2')
G.add_edge('v2', 'v3')
G.add_edge('v3', 'v4')
G.add_edge('v4', 'v5')
G.add_edge('v5', 'v1')
G.add_edge('v2', 'v4')

# Lista os vértices
print('Lista de vértices')
print(G.nodes())
input()

# Percorre o conjunto de vértices
print('Percorrendo os vértices')
for v in G.nodes():
     print(v)
input()

# Lista as arestas
print('Lista de arestas')
print(G.edges())
input()

# Percorre o conjunto de arestas
```

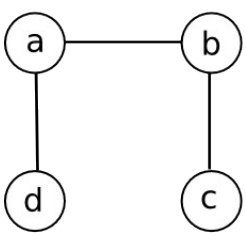
```
print('Percorrendo as arestas')
for e in G.edges():
     print(e)
input()

# Mostra a lista de graus
print('Lista de graus de G')
print(G.degree())
input()
# Acessa o grau do vértice v2
print('O grau do vértice v2 é %d' %G.degree()['v2'])
print()

# Grafo como lista de adjacências
print('Grafo como lista de adjacências')
print(G['v1'])
print(G['v2'])
print(G['v3'])
print(G['v4'])
print(G['v5'])
input()

# Obtém a matriz de adjacências do grafo G
print('Matriz de adjacências de G')
A = nx.adjacency_matrix(G)        # retorna uma matriz esparsa para
economizar memória
print(A.todense())        # converte para matriz densa (padrão)

# Adiciona um campo peso em cada aresta do grafo
G['v1']['v2']['peso'] = 5
G['v2']['v3']['peso'] = 10
G['v3']['v4']['peso'] = 2
G['v4']['v5']['peso'] = 7
G['v5']['v1']['peso'] = 4
G['v2']['v4']['peso'] = 8

# Lista cada aresta e seus respectivos pesos
print('Adicionando pesos as arestas')
for edge in G.edges():
     u = edge[0]
     v = edge[1]
     print('O peso da aresta', edge, 'vale ', G[u][v]['peso'])
input()
print()

print('Plotando o grafo como imagem...')

plt.figure(1)
# Há vários layouts, mas spring é um dos mais bonitos
nx.draw_networkx(G, pos=nx.spring_layout(G), with_labels=True)
plt.show()
```

Um excelente guia de referência oficial para a biblioteca NetworkX pode ser encontrada em: https://networkx.github.io/documentation/stable/_downloads/networkx_reference.pdf

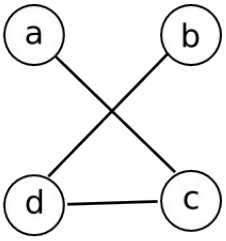
O leitor poderá encontrar diversos exemplos práticos de inúmeras funções da biblioteca.

O Problema do Isomorfismo

Um problema recorrente no estudo dos grafos consiste em determinar sob quais condições 2 grafos são de fato idênticos, ou seja, queremos saber se G1 "é igual" a G2. Dizemos que grafos que satisfazem essa condição de igualdade são isomorfos (iso = mesma, morfos = forma).



G



$\bar{G}$

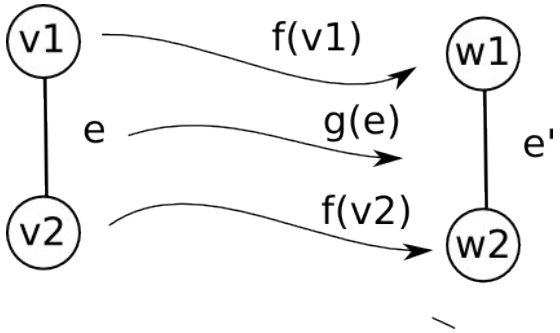
Def: G1 = (V1, E1) e G2 = (V2, E2) são isomorfos se:

i) $\exists f : V_1 \rightarrow V_2$ tal que f é bijetora (mapeamento 1 para 1)

ii) $\exists g : E_1 \rightarrow E_2$ tal que g é bijetora

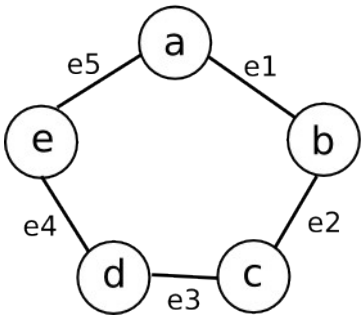
satisfazendo a seguinte restrição (*)

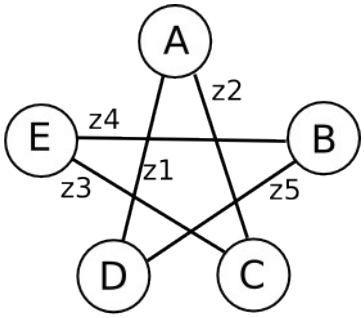
$$v_1 \leftarrow e \rightarrow v_2 \Leftrightarrow f(v_1) \leftarrow g(e) \rightarrow f(v_2)$$



Em termos práticos, G1 e G2 são isomorfos se é possível obter G2 a partir de G1 sem cortar e religar arestas, ou seja, apenas movendo seus vértices (transformação isomórfica).

Exercício: Os grafos abaixo são isomorfos? Prove ou refute.





Passo 1: Encontrar mapeamento f

v	a	b	c	d	e
f(v)	A	C	E	B	D

Passo 2: Encontrar mapeamento g, sujeito a restrição (*)

e	e1	e2	e3	e4	e5
f(e)	z2	z3	z4	z5	z1

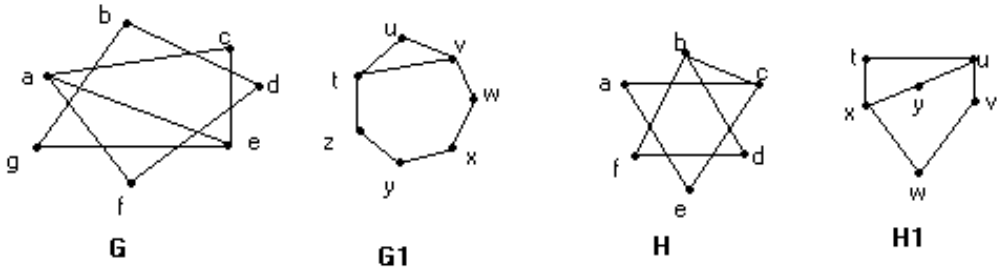
i) e1 = (a, b) → g(e1) deve ser a aresta que une f(a) com f(b): (A, C) = z2
ii) e2 = (b, c) → g(e2) deve ser a aresta que une f(b) com f(c): (C, E) = z3
iii) e3 = (c, d) → g(e3) deve ser a aresta que une f(c) com f(d): (C, E) = z4
iv) e4 = (d, e) → g(e4) deve ser a aresta que une f(d) com f(e): (B, D) = z5
v) e3 = (e, a) → g(e5) deve ser a aresta que une f(e) com f(a): (D, A) = z1
Portanto, os grafos G1 e G2 são isomorfos.

Propriedades Invariantes

Na determinação de isomorfismo entre grafos torna-se útil o estudo de propriedades invariantes. Em outras palavras, ao se obter medidas invariantes a transformações isomórficas, pode-se facilmente verificar que dois grafos não são isomorfos se tais medidas não forem idênticas. Sejam G1 = (V1, E1) e G2 = (V2, E2) dois grafos isomorfos. Então,

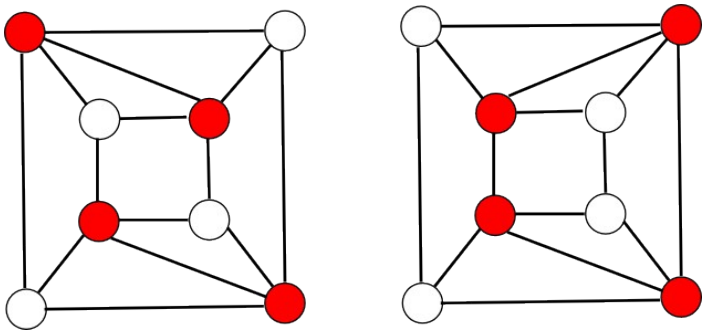
1) |V1| = |V2| (o número de vértices é igual)
2) |E1| = |E2| (o número de arestas é igual)
3) $\omega(G1)=\omega(G2)$ (mesmo número de componentes conexos)
4) $L_{G1} = L_{G2}$ (listas de graus são idênticas)
5) Ambos G1 e G2 admitem ciclos de comprimento k, para k <= n

Ex:



Note que G é isomorfo a G1. Porém, H não é isomorfo a H1 pois enquanto H admite ciclo de comprimento 3, H1 não admite.

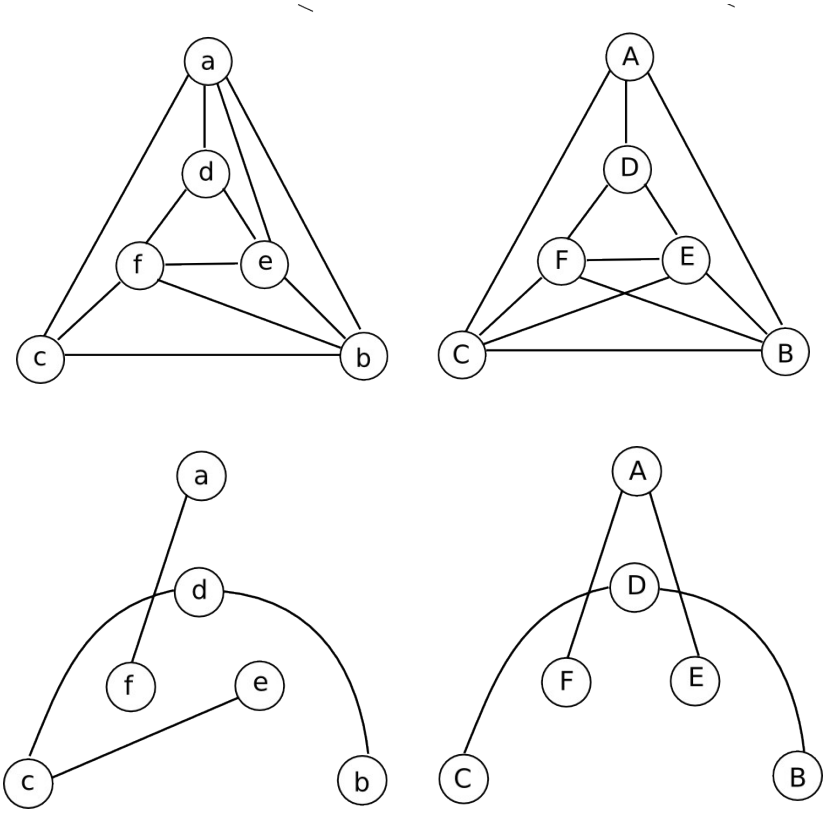
Ex:



Número de vértices: 8
Número de arestas: 8
Lista de graus: (3, 3, 3, 3, 4, 4, 4, 4)
Ciclos de comprimentos 3, 4, 5, 6, 7 e 8 → OK

Note que, apesar de não ferir nenhuma das propriedades invariantes, não podemos afirmar que os grafos são isomorfos. Na verdade, os grafos em questão não são isomorfos. Note que em um deles, todos os vértices de grau 4 (vermelho) possuem como vizinhos exatamente um outro vértice vermelho, enquanto no outro, cada vértice de grau 4 possui exatamente 2 vértices vermelhos. Isso significa uma ruptura na topologia, o que implica em corte e posterior religação de aresta.

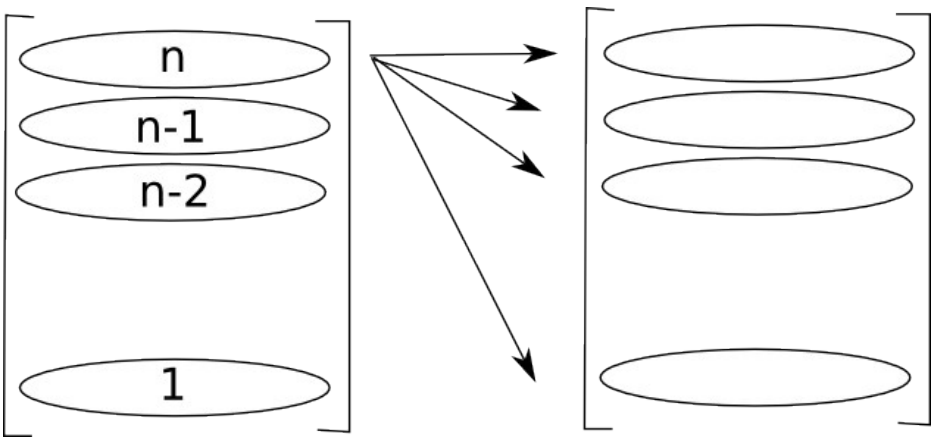
Teorema: $G_1 \equiv G_2 \Leftrightarrow \bar{G}_1 \equiv \bar{G}_2$
Dois grafos são isomorfos se e somente se seus complementares também o forem.



Note que no primeiro grafo os vértices de grau 2 são adjacentes, o que não ocorre no segundo.

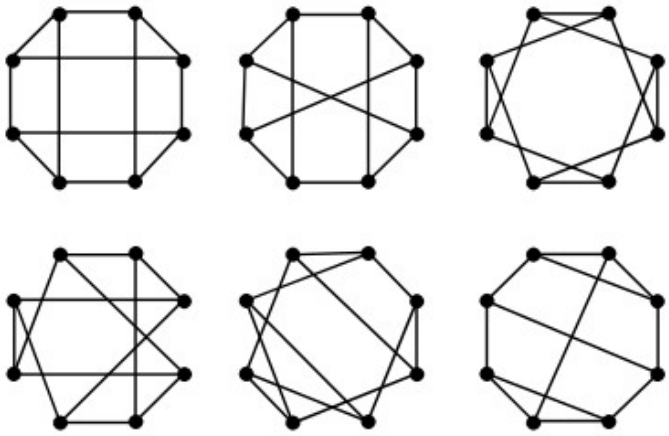
Teorema: $G_1 \equiv G_2 \Leftrightarrow A_1 = A_2$ para alguma sequencia de permutações de linhas e colunas.

Algoritmo: Dadas duas matrizes de adjacências A1 e A2, permutar linhas e colunas de A1 de modo a chegar em A2. Em termos de complexidade, note que para as linhas temos o seguinte cenário



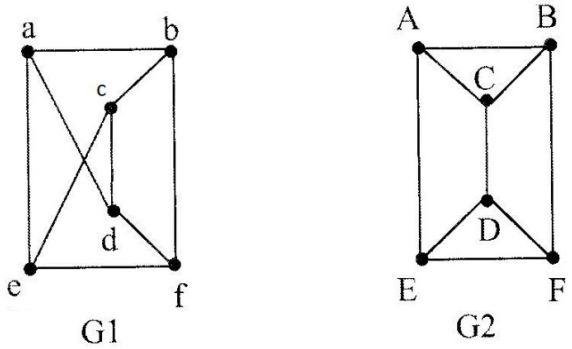
o que resulta num total de n! possibilidades. O mesmo vale para as colunas, de forma que a complexidade do algoritmo é da ordem de O(n!), tornando o método inviável para a maioria dos grafos. Atualmente, ainda não se conhecem algoritmos polinomiais para esse problema. Para algumas classes de grafos o problema é polinomial (árvores, grafos planares). Esse é o primeiro dos problemas que veremos que ainda não tem solução: o problema do isomorfismo entre grafos pode ser resolvido em tempo polinomial? Não se conhece resposta completa para a pergunta.

Ex: Os seis grafos a seguir consistem de três pares isomorfos. Quais são esses pares?

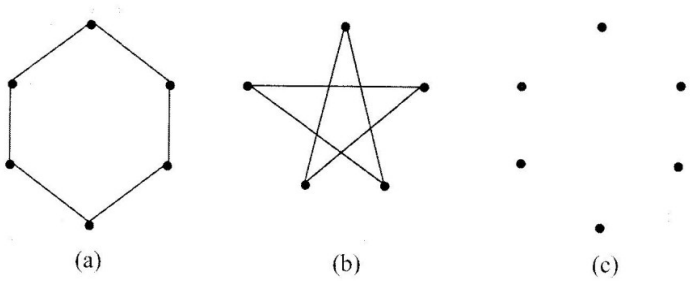


Ex: Os dois grafos a seguir são isomorfos ou não? Prove sua resposta.

c)



Ex: Um grafo simples G é chamado de autocomplementar se ele for isomorfo ao seu próprio complemento. Quais dos grafos a seguir são autocomplementares?



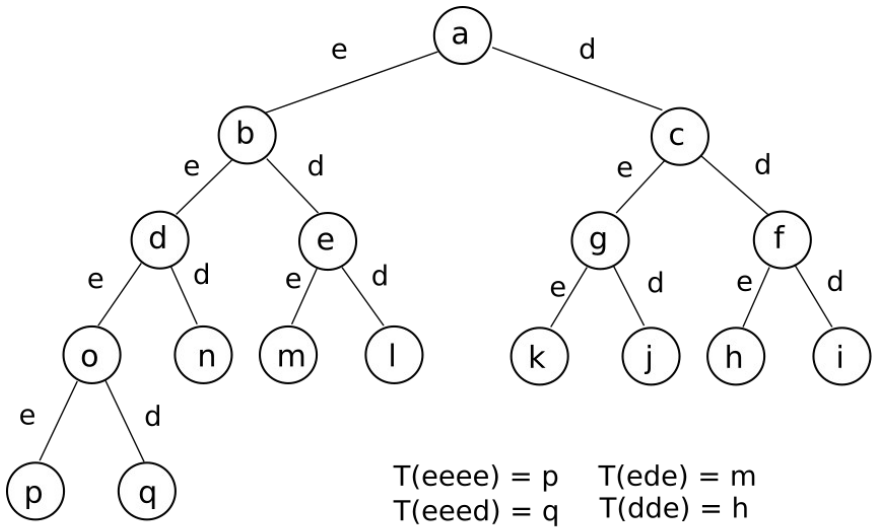
Prove que, se G é um grafo autocomplementar com n vértices, então n = 4t ou n = 4t + 1, para algum inteiro t (dica: considere o número de arestas do K_n).

Aula 9 - Busca em grafos (BFS e DFS)

Importância: como navegar em grafos de maneira determinística de modo a percorrer um conjunto de dados não estruturado, visitando todos os nós exatamente uma vez.
Objetivo: acessar/recuperar todos os elementos do conjunto V
Questões: De quantas maneiras podemos fazer isso? Como? Qual a melhor maneira?

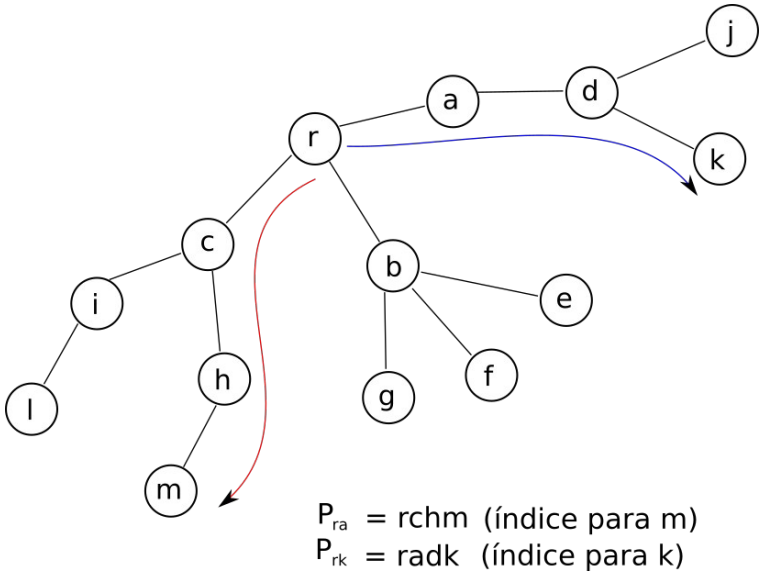
Relação entre busca em grafos e árvores

Buscar elementos num grafo G é basicamente o processo de extrair uma árvore T a partir de G. Mas porque? Como busca se relaciona com uma árvore? Isso vem de uma das propriedades das árvores
Numa arvore existe um único caminho entre 2 vértices u, v. Considere uma árvore binária



índices identificam unicamente os elementos

Pode-se criar um esquema de indexamento baseado nos nós a esquerda e a direita. Cada elemento do conjunto possui um índice único que o recupera. No caso de árvores genéricas, o caminho faz o papel do índice único



Portanto, dado um grafo G, extrair uma árvore T com raiz r a partir dele, significa indexar unicamente cada elemento do conjunto.

Busca em Largura (Breadth-First Search – BFS)

Ideia geral: a cada novo nível descoberto, todos os vértices daquele nível devem ser visitados antes de prosseguir para o próximo nível

Definição das variáveis (pseudo-código)

i) v. color : status do vértice v (existem 3 possíveis valores)
- a) WHITE: vértice v ainda não descoberto (significa que v ainda não entrou na fila Q)
- b) GRAY: vértice já descoberto (significa que v está na fila Q)
- c) BLACK: vértice finalizado (significa que v já saiu da fila Q)

ii) $\lambda(v)$: armazena a menor distância de v até a raiz

iii) $\pi(v)$: predecessor de v (onde estava quando descobri v)

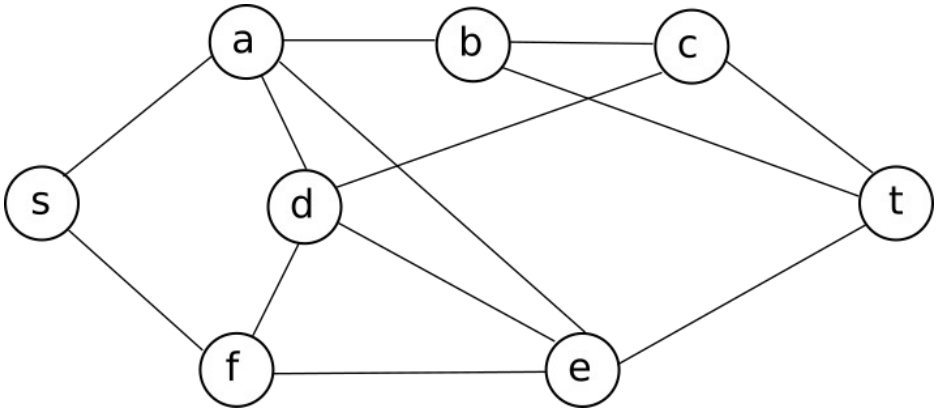
iv) Q: Fila (FIFO)

2 primitivas
- a) pop: remove elemento do início da fila
- b) push: adiciona um elemento no final da fila

PSEUDOCÓDIGO

```
BFS(G, s)
{
        for each   v∈V−{s}   {
                v.color = WHITE
                  λ(v)=∞
                  π(v)=nil
         }
         s.color = GRAY
           λ(s)=0
           π(v)=nil
           Q=∅
         push(Q, s)
         while   Q≠∅   {
                u = pop(Q)
                for each   v∈N(u)   // para todo vizinho de u
                {
                        if v.color == WHITE  // se ainda não passei por aqui, processo vértice v
                        {
                                λ(v)=λ(u)+1        // v é descendente de u então distancia +1
                                π(v)=u
                              v.color = GRAY
                              push(Q, v)                    // adiciona v no final da fila
                        }
                }
                u. color = BLACK              // Após explorar todos vizinhos de u, finalizo u
        }
 }
```

O algoritmo BFS recebe um grafo não ponderado G e retorna uma árvore T, conhecida como BFS-tree. Essa árvore possui uma propriedade muito especial: ela armazena os menores caminhos da raíz s a todos os demais vértices de T (menor caminho de s a v, $\forall v \in V$)



Trace do algoritmo BFS

i	u = pop(Q)	V'={ $v \in N(u)$ | v.color = WHITE}	$\lambda(v)$	$\pi(v)$
0	s	{a, f}	$\lambda(a)=\lambda(s)+1=1$	$\pi(a)=s$
			$\lambda(f)=\lambda(s)+1=1$	$\pi(f)=s$
1	a	{b, d, e}	$\lambda(b)=\lambda(a)+1=2$	$\pi(b)=a$
			$\lambda(d)=\lambda(a)+1=2$	$\pi(d)=a$
			$\lambda(e)=\lambda(a)+1=2$	$\pi(e)=a$
2	f	$\emptyset$	---	---
3	b	{c, t}	$\lambda(c)=\lambda(b)+1=3$	$\pi(c)=b$
			$\lambda(t)=\lambda(b)+1=3$	$\pi(t)=b$
4	d	$\emptyset$	---	---
5	e	$\emptyset$	---	---
6	c	$\emptyset$	---	---
7	t	$\emptyset$	---	---

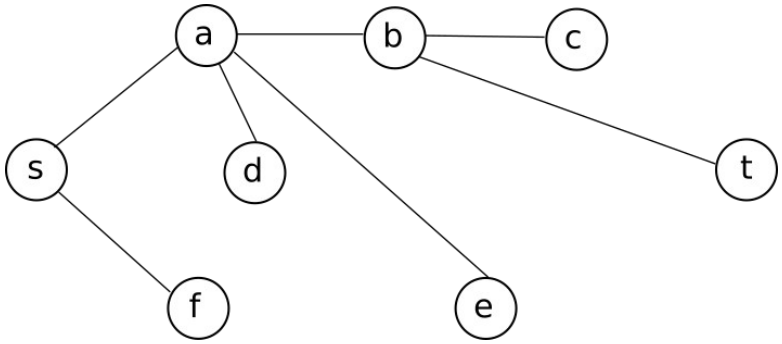
FILA

$Q^{(0)}=[s]$
$Q^{(1)}=[a,f]$
$Q^{(2)}=[f,b,d,e]$
$Q^{(3)}=[b,d,e]$
$Q^{(4)}=[d,e,c,t]$
$Q^{(5)}=[e,c,t]$
$Q^{(6)}=[c,t]$
$Q^{(7)}=[t]$
$Q^{(8)}=\emptyset$

A complexidade da busca em largura é O(n + m), onde n é o número de vértices e m é o número de arestas. É fácil perceber que cada vértice e aresta serão acessados exatamente uma vez.

Árvore BFS

v	s	a	b	c	d	e	f	t
$\pi(v)$	---	s	a	b	a	a	s	b
$\lambda(v)$	0	1	2	3	2	2	1	3



Note que a árvore nada mais é que a união dos caminhos mínimos de s (origem) a qualquer um dos vértices do grafo (destinos). A BFS-tree geralmente não é única, porém todas possuem a mesma profundidade (mínima distância da raiz ao mais distante)

Pergunta: Como podemos implementar um script para computar o diâmetro de um grafo G usando a BFS? Pense em termos do cálculo da excentricidade de cada vértice.

Teorema: A BFS sempre termina com $\lambda(v)=d(s,v)$ para $\forall v \in V$, onde $d(s,v)$ é a distância geodésica (menor distância entre s e v).

Prova por contradição:

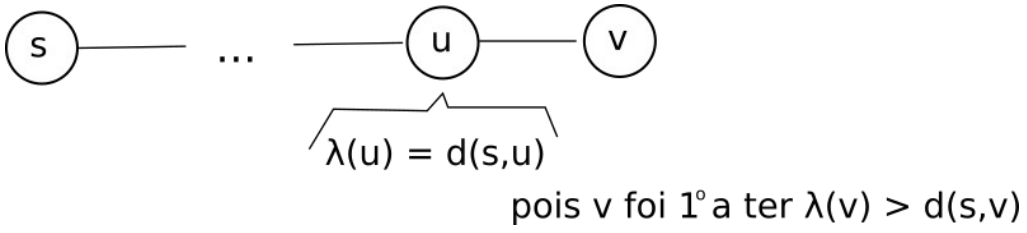
1. Sabemos que na BFS $\lambda(v) \geq d(s,v)$

2. Suponha que $\exists v \in V$ tal que $\lambda(v) > d(s,v)$, onde v é o primeiro vértice que isso ocorre ao sair da fila Q

3. Então, existe caminho P_{sv} pois senão $\lambda(v)=d(s,v)=\infty$ (contradiz 2)

4. Se existe P_{sv} então existe um caminho mínimo P^*_{sv}

5. Considere $u \in V$ como predecessor de v em P^*_{sv}



6. Então, $d(s,v)=d(s,u)+1$ (pois u é predecessor de v)

7. Assim, temos

$\lambda(v) > d(s,v) = d(s,u) + 1 = \lambda(u) + 1$
(2) (6) (5)

e portanto $\lambda(v) > \lambda(u) + 1$ (*) , o que é uma contradição pois só existem 3 possibilidades quando u sai da fila Q, ou seja, u = pop(Q)

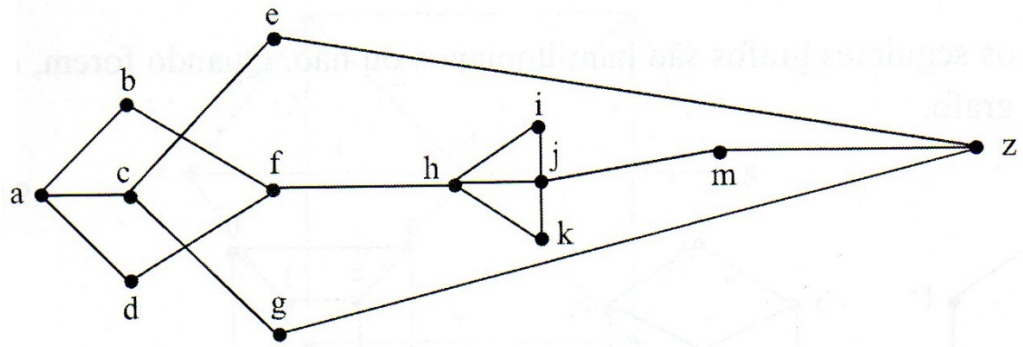
i) v é WHITE: $\lambda(v) = \lambda(u) + 1$ (contradição)

ii) v é BLACK: se isso ocorre significa que v sai da fila Q antes de u, ou seja, $\lambda(v) < \lambda(u)$ (contradição)

iii) v é GRAY: então v foi descoberto por um w removido de Q antes de u, ou seja, $\lambda(w) \leq \lambda(u)$. Além disso, $\lambda(v) = \lambda(w) + 1$. Assim, temos $\lambda(w) + 1 \leq \lambda(u) + 1$, o que finalmente implica em $\lambda(v) \leq \lambda(u) + 1$ (contradição)

Portanto, $\nexists v \in V$ tal que $\lambda(v) > d(s,v)$.

Exercício: Obtenha a BFS-Tree do grafo a seguir. Qual a profundidade da árvore. Obtenha o número de caminhos mínimos de a até m.



Busca em Profundidade (Depth-First Search – DFS)

Ideia geral: a cada vértice descoberto, explorar um de seus vizinhos não visitados (sempre que possível). Imita exploração de labirinto, aprofundando sempre que possível.

Definição das variáveis

i) v.d: discovery time (tempo de entrada em v)
v.f: finishing time (tempo de saída de v)

ii) v. color : status do vértice v (existem 3 possíveis valores)
a) WHITE: vértice v ainda não descoberto
b) GRAY: vértice já descoberto
c) BLACK: vértice finalizado

iii) $\pi(v)$: predecessor de v (onde estava quando descobri v)

iv) Q: Pilha (LIFO)
Para simular a pilha de execução, pode-se utilizar um recurso computacional: Recursão!
Assim, não é necessário implementar de fato essa estrutura de dados (vantagem)

Porém, em casos extremos (tamanho muito grande), recursão pode gerar problemas.

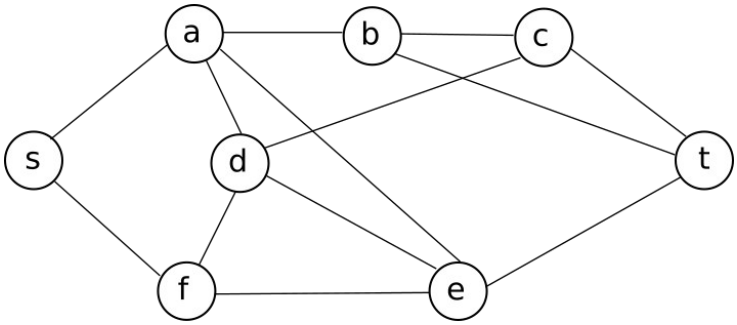
PSEUDOCÓDIGO (versão recursiva)

```
DFS(G, s)
{
        for each   u∈V
        {
                u.color = WHITE
                  π(u)=nil
        }
        time = 0                    // variável global para armazenar o tempo
        DFS_visit(G, s)
}

// Função recursiva que é chamada sempre que um vértice é descoberto
DFS_visit(G, u)
{
        time++
        u.d = time
        u.color = GRAY
        for each   v∈N(u)
        {
                if v.color == WHITE
                {
                        π(v)=u
                      DFS_visit(G, v)              // chamada recursiva
                }
        }
        u.color = BLACK
        time++
        u.f = time
}
```

Para implementar a versão iterativa (não recursiva) da Busca em Profundidade (DFS), basta usar o mesmo algoritmo da Busca em Largura (BFS) trocando a fila por uma pilha.

Da mesma forma que o algoritmo BFS, esse método recebe um grafo G não ponderado e retorna uma árvore, a DFS_tree.

Ex:



u	u.color	u.d	V' = { $v \in N(u)$ / v.color = WHITE}	$\pi(u)$	u.f
s	G	1	{a, f}	--	16
a	G	2	{b, d, e}	s	15
b	G	3	{c, t}	a	14
c	G	4	{d, t}	b	13
d	G	5	{e, f}	c	12
e	G	6	{f, t}	d	11
f	G	7	$\varnothing$	e	8
t	G	9	$\varnothing$	e	10

Diferenças ente BFS e DFS

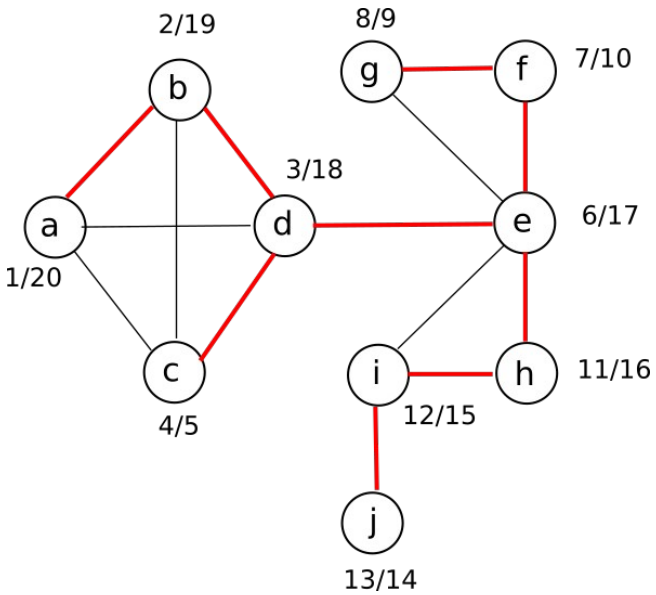
BFS	**x**	**DFS**
- aspecto espacial		- aspecto temporal
- caminhos mínimos		- vértices de corte, ordenação topológica
- Fila		- Pilha

Propriedades da árvore da busca em profundidade (DFS_tree)

1) A rotulação tem o seguinte significado:
a) [u.d, u.f] $\subset$ [v.d, v.f]: u é descendente de v
b) [v.d, v.f] $\subset$ [u.d, u.f]: v é descendente de u
c) [v.d, v.f] e [u.d, u.f] são disjuntos: estão em ramos distintos da árvore

2) Após a DFS, podemos classificar as arestas de G como:
a) tree_edges: $e \in T$
b) fb_edges (forward-backward edge): $e \notin T$

Def: v é um vértice de corte $\Leftrightarrow$ v tem um filho s tal que $\nexists$ fb_edge ligando s ou qualquer descendente de s a um ancestral de v



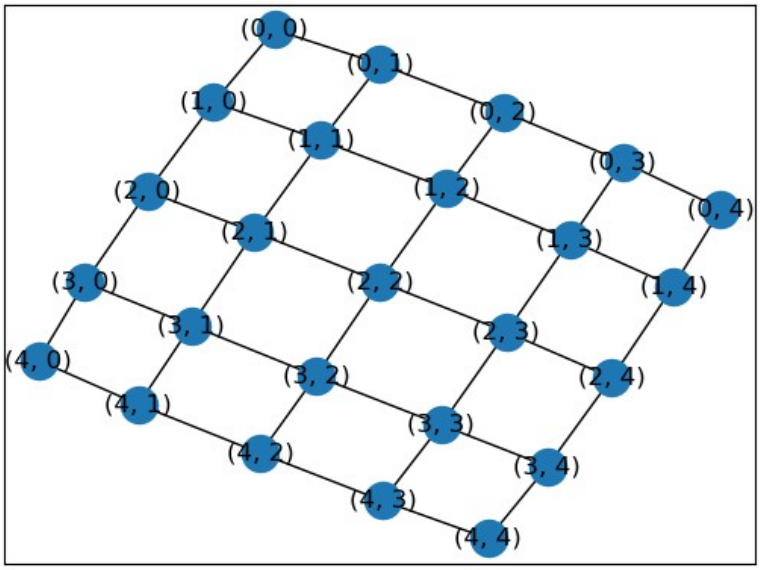
Perguntas:
a) b é vértice de corte? Não pois fb_edge (a, d) liga um sucessor a um antecessor
b) d é vértice de corte? Sim, pois não há fb_edge entre sucessor e antecessor
c) e é vértice de corte? Sim, pois não há fb_edge

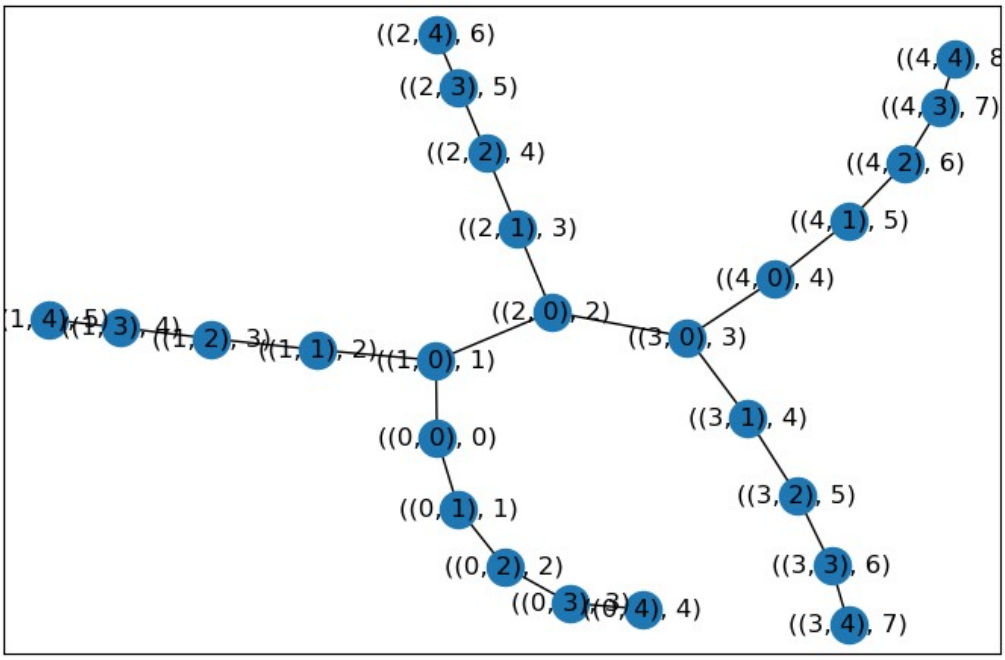
O código em Python a seguir ilustra uma implementação do algoritmo da Busca em Largura. Conforme mencionado anteriormente, para a Busca em Profundidade, basta trocarmos a estrutura de dados fila por uma pilha.

```
# Adiciona bibliotecas auxiliares
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque

def bfs(G, s):

    # dicionário para armazenar o mapa de predecessores
    P = {} # estrutura de dados que mapeia uma chave a um valor
    # inicialização do algoritmo
    for v in G.nodes():
        G.nodes[v]['color'] = 'white'
        G.nodes[v]['lambda'] = float('inf')

    # iniciar cor da raiz como cinza
    G.nodes[s]['color'] = 'gray'
    # custo para a raiz é 0
    G.nodes[s]['lambda'] = 0
    # iniciar fila Q vazia
    Q = deque()
    # inserir nó raiz no início da fila
    Q.append(s)

    # enquanto fila não estiver vazia
    while (len(Q) > 0):
        # obter o primeiro elemento da fila
        u = Q.popleft()
        # para cada vertice adjacênte a u
        for v in G.neighbors(u):
            # se v é branco
            if (G.nodes[v]['color'] == 'white'):
                # atualizar custo de v
                G.nodes[v]['lambda'] = G.nodes[u]['lambda'] + 1
                # adicionar u como antecessor de v
                P[v] = u
                # atualizar cor de v
                G.nodes[v]['color'] = 'gray'
                # incluir v em  Q
                Q.append(v)

        # atualizar cor de u
        G.nodes[u]['color'] = 'black'

    # retorna a lista de antecessores
    return P
```

```
if __name__ == '__main__':
    # Cria um grafo de exemplo
    G = nx.grid_2d_graph(5, 5)

    print('Plotando grafo...')
    # Cria figura para plotagem do grafo
    plt.figure(1)
    # Há vários layouts, mas spring é um dos mais bonitos
    nx.draw_networkx(G, pos=nx.spring_layout(G), with_labels=True)
    # Exibir figura
    plt.show()

    # Aplicando Busca em Largura
    P = bfs(G, (0, 0))      # retorna as arestas que compõem a BFS_tree
    # Cria um grafo vazio para árvore da busca em largura
    T = nx.Graph()
    # Inserir arestas retornadas pelo algoritmo em T
    T.add_edges_from([ (u, v) for u, v in P.items() ])
    # Armazena as distâncias em um dicionário (lambda de cada vértice)
    dist = { v: (v, data['lambda']) for v, data in G.nodes(data=True) }

    # Cria figura para plotagem da árvore
    plt.figure(2)
    # Define o layout
    pos = nx.spring_layout(T)
    # Plotar vertices de T
    nx.draw_networkx_nodes(T, pos)
    # Plotar distâncias
    nx.draw_networkx_labels(T, pos, labels=dist)
    # Plotar arestas de T
    nx.draw_networkx_edges(T, pos)
    # Exibir figura
    plt.show()
```

As figuras a seguir mostram os resultados da execução do script anterior: na primeira vemos o grafo reticulado 2D com 25 vértices (grade 5 x 5), e na segunda vemos a árvore de busca em largura.



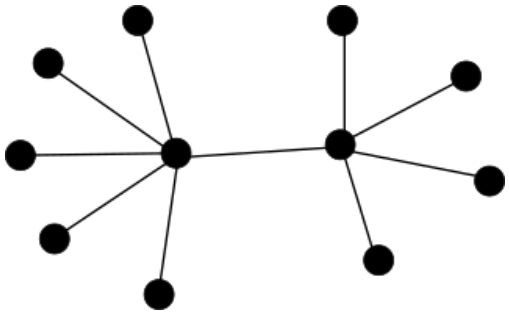


Note que a raiz da árvore é o vértice (0, 0). Além disso, note que a árvore resultante não é binária, no sentido de que todo nó tem exatamente dois filhos. A definição mais geral de árvore é a seguinte: Uma árvore T = (V, E) é um grafo acíclico (sem ciclos) e conexo (existe um caminho entre qualquer par de vértices). Note que o grafo indicado na figura acima satisfaz essas duas condições, sendo portanto uma árvore.

Aula 10 – Grafos: O problema da árvore geradora mínima

Árvores são grafos especiais com diversas propriedades únicas. Devido a essas propriedades são extremamente importantes na resolução de vários tipos de problemas práticos. Veremos ao longo do curso que vários problemas que estudaremos se resumem a: dado um grafo G, extrair uma árvore T a partir de G, de modo que T satisfaça uma certa propriedade (como por exemplo, mínima profundidade, máxima profundidade, mínimo peso, mínimos caminhos, etc).

Def: Um grafo G = (V, E) é uma árvore se G é acíclico e conexo.

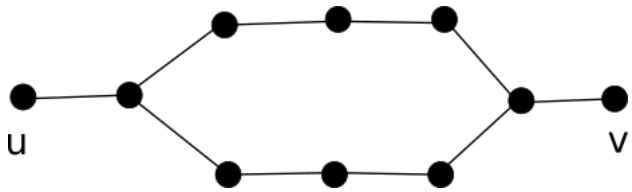


Teoremas e propriedades

Teorema: G é uma árvore ⇔ ∃ um único caminho entre quaisquer 2 vértices $u, v \in V$

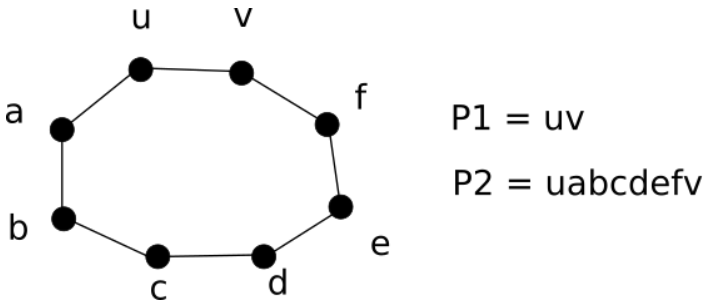
(ida) p → q = !q → !p
∄ um único caminho entre quaisquer u,v → G não é uma árvore

a) Pode existir um par u,v tal que ∄ caminho (zero caminhos). Isso implica em G desconexo, o que implica que G não é uma árvore
b) Pode existir um par u,v tal que ∃ mais de um caminho.

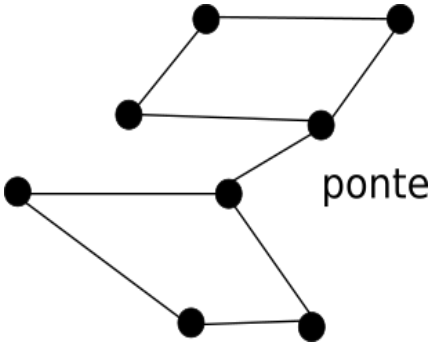


Porém neste caso temos a formação de um ciclo e portanto G não pode ser árvore.

(volta) q → p = !p → !q
G não é árvore → ∄ único caminho entre quaisquer u,v
Para G não ser árvore, G deve ser desconexo ou conter um ciclo. Note que no primeiro caso existe um par u,v tal que não há caminho entre eles. Note que no segundo caso existem 2 caminhos entre u e v, conforme ilustra a figura



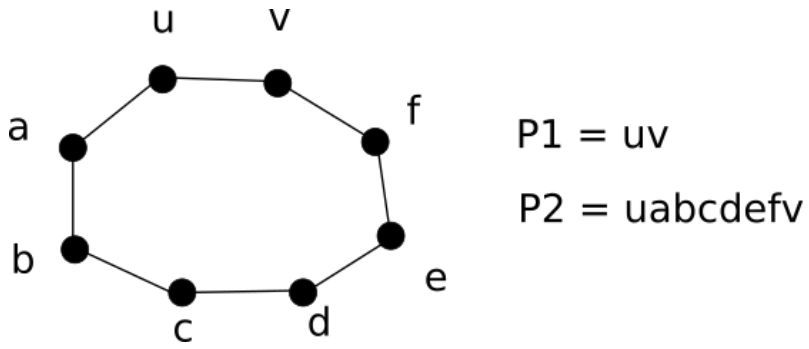
Def: Uma aresta $e \in E$ é ponte se $G - e$ é desconexo
Ou seja, a remoção de uma aresta ponte desconecta o grafo



Como identificar arestas ponte?

Teorema: Uma aresta $e \in E$ é ponte $\Leftrightarrow$ aresta não pertence a um ciclo C

(ida) p → q = !q → !p
$e \in C$ → aresta não é ponte



Como aresta pertence a um ciclo C, há 2 caminhos entre u e v. Logo a remoção da aresta e = (u,v) não impede que o grafo seja conexo, ou seja, G – e ainda é conexo. Portanto, e não é ponte

(volta) q → p = !p → !q
aresta não é ponte → $e \in C$
Se aresta não é ponte então G – e ainda é conexo. Se isso ocorre, deve-se ao fato de que em G – e ainda existe um caminho entre u e v que não passa por e. Logo, em G existem existem 2 caminhos, o que nos leva a conclusão de que a união entre os 2 caminhos gera um ciclo C.

Teorema: G é uma árvore $\Leftrightarrow$ Toda aresta é ponte
(ida) p → q = !q → !p
$\exists$ aresta não ponte → G não é árvore

A existência de uma aresta não ponte implica na existência de ciclo. A presença de um ciclo C faz com que G não seja um árvore

(volta) q → p = !p → !q
G não á árvore → $\exists$ aresta não ponte

Para G não ser uma árvore, deve existir um ciclo em G. Logo, todas as arestas pertencentes ao ciclo não são pontes.

Teorema: Se G = (V, E) é uma árvore com |V| = n então |E| = n – 1

Prova por indução

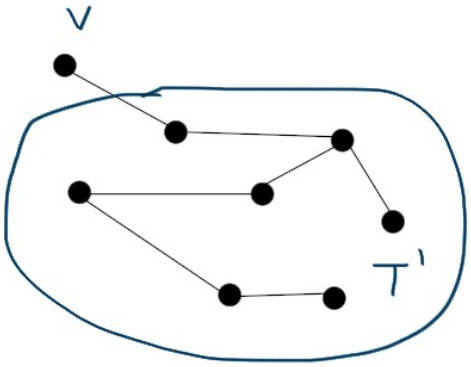
P(n): Toda árvore de n vértices tem n – 1 arestas

Base: P(1): n = 1 → 0 arestas (OK)

Passo de indução: P(k) → P(k+1) para k arbitrários

P(k+1): Toda árvore de n + 1 vértices tem n arestas

1. Seja $T = (V_T, E_T)$ com $|V_T| = n + 1$
2. Seja $v \in V_T$ uma folha em T
3. Seja $T' = (V'_T, E'_T)$ a árvore obtida removendo v e a única aresta incidente a v: T' = T – v



4. Note que T' é uma árvore, pois como T é conexo, T' também é e como T é acíclico, T' também é. Logo, podemos dizer que $T' = (V'_T, E'_T)$ possui n' vértices e m' arestas: $|V'_T| = n'$ e $|E'_T| = m'$.
5. Mas pela hipótese de indução, toda árvore de n' vértices tem n'-1 arestas, então m' = n' – 1
6. Como T' tem exatamente uma aresta e um vértice a menos que T:

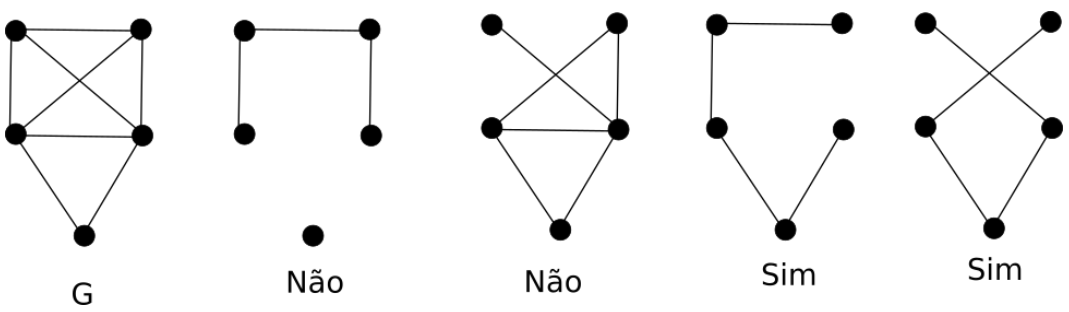
$$m' = m - 1 = n' - 1 = (n - 1) - 1$$

o que implica em m – 1 = n – 1 – 1, ou seja, m = n – 1. A prova está concluída.

Teorema: A soma dos graus de uma árvore de n vértices não depende da lista de graus, sendo dada por 2n – 2

$$\sum_{i=1}^{n} d(v_i) = 2|E| = 2(n-1) = 2n-2$$

Def: Árvore geradora (spanning tree)
Seja G = (V, E) um grafo. Dizemos que $T = (V, E_T)$ é uma árvore geradora de G se T é um subgrafo de G que é uma árvore (ou seja tem que conectar todos os vértices)



Como pode ser visto, um grafo G admite inúmeras árvores geradoras. A pergunta que surge é: Quantas árvores geradoras existem num grafo G = (V, E) de n vértices?

Árvores Geradoras Mínimas (Minimum Spanning Trees - MST's)

Até o presente momento, estamos lidando com grafos sem pesos nas arestas. Isso significa que dado um grafo G, temos inúmeras formas de obter uma árvore geradora T (vimos que existem muitas dessas árvores num grafo de n vértices). A partir de agora, iremos lidar com grafos ponderados, ou seja, grafos em que as arestas possuem um peso/custo de conexão. O objetivo da seção em questão consiste em fornecer algoritmos para resolver o seguinte problema: dado um grafo ponderado G, obter dentre todas as árvores geradoras possíveis, aquela com o menor peso.

Definição do problema: Dado G = (V, E, w), onde w: E → R^+ (peso da aresta e), obter a árvore geradora T que minimiza o seguinte critério:

$w(T)=\sum_{e\in T} w(e)$ (soma dos pesos das arestas que compõem a árvore)

Obs: A matriz de adjacência de um grafo ponderado é dada por:

$$A_{i,j}=\begin{cases} w_{ij}, & i \leftrightarrow j \\ \infty, & c.c \end{cases}$$

Exemplo: Interligação banda larga dos bairros de São Carlos (NET)

Ideia geral: a cada passo escolher a aresta de menor peso que seja segura
Aresta segura = aresta que ao ser inserida não faz a árvore deixar de ser árvore
Como determinar se uma aresta é segura?
Cada algoritmo propõe suas especificações próprias para isso

Algoritmo de Kruskal

Objetivo: escolher a cada passo a aresta de menor peso que não forme um ciclo
Entrada: G = (V, E, w) conexo
Saída: T = (V, E_T)
1. Defina T = (V, E_T) como o grafo nulo de n vértices
2. Enquanto $|E_T| < n - 1$
    a)T = T + {e}, com e sendo a aresta de menor peso em G que não forma m ciclo em T
    b) E = E - {e}

Segundo Cormen, uma metodologia para detecção de ciclos pode ser implementada utilizando a seguinte ideia: inicialmente cada vértice é colocado num grupo distinto e cada vez que uma aresta é inserida, os dois vértices extremidades passam a fazer parte do mesmo grupo. Assim, arestas que ligam vértices do mesmo grupo, formam ciclos e devem ser evitadas.

Para isso, Cormen apresenta 3 primitivas básicas:

make_set(v): cria um grupo contendo um único vértice, v
find_set(v): retorna o grupo a que o vértice v pertence
union(u, v): faz a união dos grupos de u e v, criando um único grupo

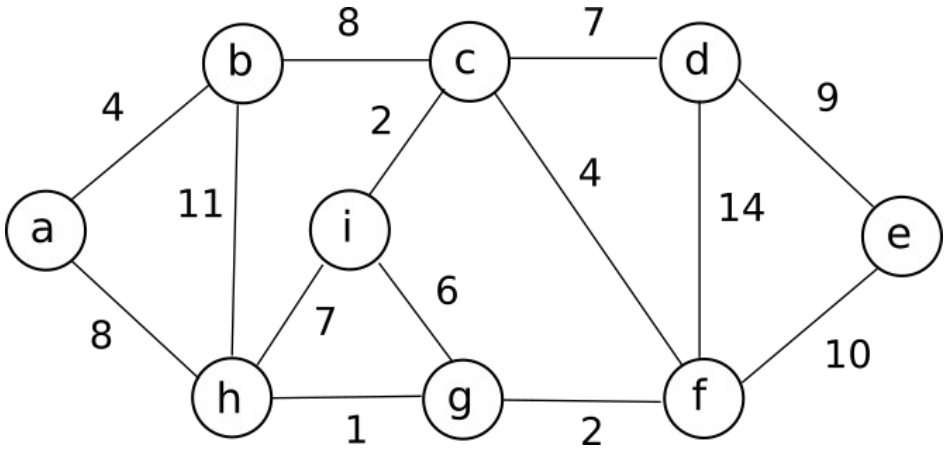
PSEUDOCÓDIGO (Cormen)

```
MST_Kruskal(G, w)    // Entrada: grafo G ponderados
{
        A = ∅
        for each    v ∈ V
                make_set(v)
        //sort edges into nondecreasing order
        for each    e ∈ E   (ordered)
        {
                if (find_set(u)     ≠     find_set(v))
                {
                        A = A ∪ {(u, v)}
                        union(u,v)
                }
        }
}
```

Trata-se de um algoritmo Guloso (segue estratégia de resolver problemas fazendo sempre a escolha ótima em cada passo. Nesse caso, sempre tenta a aresta de menor peso). A seguir iremos realizar um trace do algoritmo (simulação passo a passo). Para isso iremos considerar a seguinte notação:

E^- : conjunto das arestas de peso mínimo não seguras (find_set(u) = find_set(v))
E^+ : conjunto das arestas de peso mínimo seguras (find_set(u) != find_set(v))
e_k : aresta escolhida no passo k

Ex: Suponha que os vértices representem bairros e as arestas com pesos os custos de interligação desses bairros (fibra ótica)



E = [1, 2, 2, 4, 4, 6, 7, 7, 8, 8, 9, 10, 11, 14]

k	E^-	E^+	e_k
1	-	{(g,h)}	(g,h)
2	-	{(c,i), (f,g)}	(c,i)
3	-	{(g,f)}	(g,f)
4	-	{(a,b), (c,f)}	(a,b)

5	-	{(c,f)}	(c,f)
6	{(g,i)}	-	-
7	{(h,i)}	{(c,d)}	(c,d)
8	-	{(a,h), (b,c)}	(a,h)
9	{(b,c)}	-	-
10	-	{(d,e)}	(d,e)

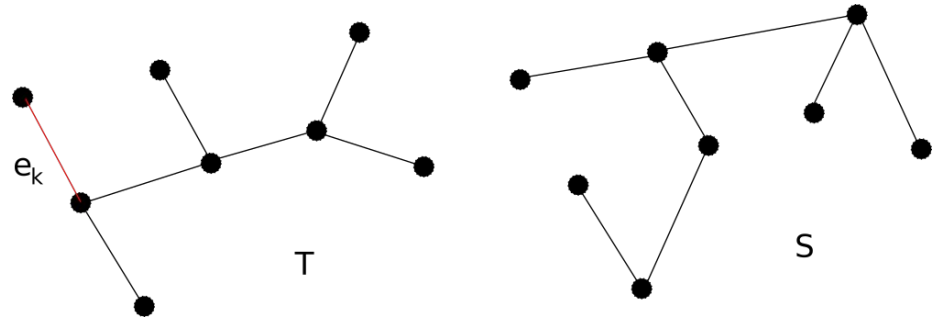
A seguir iremos demonstrar a otimalidade do algoritmo de Kruskal.

Teorema: Toda árvore T gerado pelo algoritmo de Kruskal é uma MST de G
Garante que o algoritmo sempre funciona e é ótimo (retorna sempre a solução ótima)
Prova por contradição

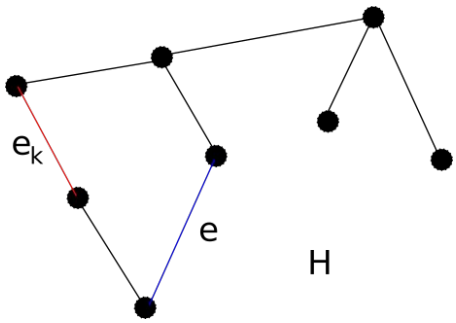
Seja T é a árvore retornada por Kruskal.

1. Supor que $\exists S \neq T$ tal que $w(S) < w(T)$ (S é uma árvore)

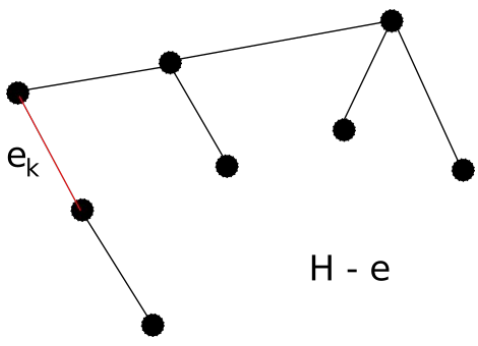
2. Seja $e_k \in T$ a primeira aresta adicionada em T que não está em S (pois árvores são diferentes)



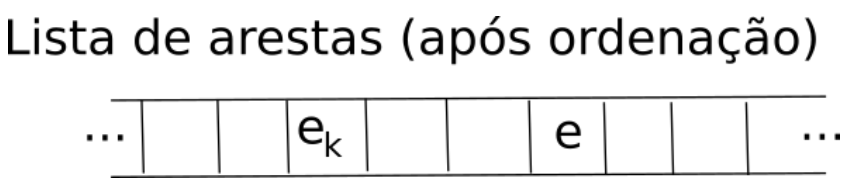
3. Faça $H = (S + e_k)$. Note que H não é mais uma árvore e contém um ciclo



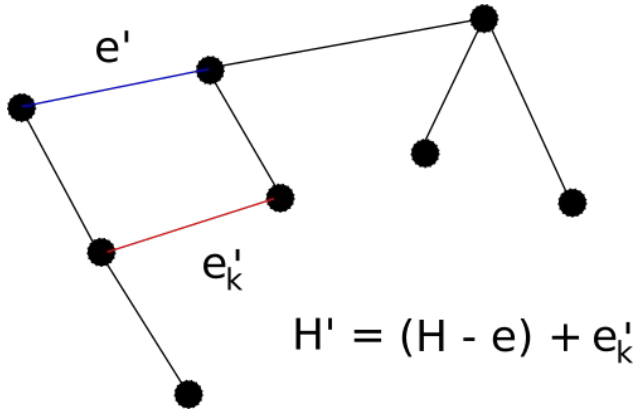
4. Note que no ciclo C, $\exists e \in S$ tal que $e \notin T$ (pois senão C existiria em T). O subgrafo $H - e$ é conexo, possui n - 1 arestas e define uma árvore geradora de G.



5. Porém, $w(e_k) \leq w(e)$ e assim $w(H-e) \leq w(S)$ (pois de acordo com Kruskal e_k vem antes de e na lista ordenada de arestas, é garantido pela ordenação)



6. Repetindo o processo usado para gerar $H-e$ a partir de S é possível produzir uma sequência de árvores que se aproximam cada vez mais de T.



S → (H - e) → (H' - e') → (H'' - e'') → … → T

de modo que

$w(S) \geq w(H-e) \geq w(H'-e') \geq ... \geq w(T)$ (contradição a suposição inicial)

Portanto, não existe árvore com peso menor que T, mostrando que T tem peso mínimo. (Não há como S ter peso menor que T). A complexidade do algoritmo de Kruskal depende das primitivas utilizadas, mas pode-se mostrar que é possível ter complexidade O(m log n), com |V| = n e |E| = m.

Algoritmo de Prim

Ideia geral: Inicia em uma raiz r. Enquanto T não contém todos os vértices de G, o algoritmo iterativamente adiciona a T a aresta de menor peso que sai do conjunto dos vértices finalizados (S) e chega no conjunto dos vértices em aberto (V - S)

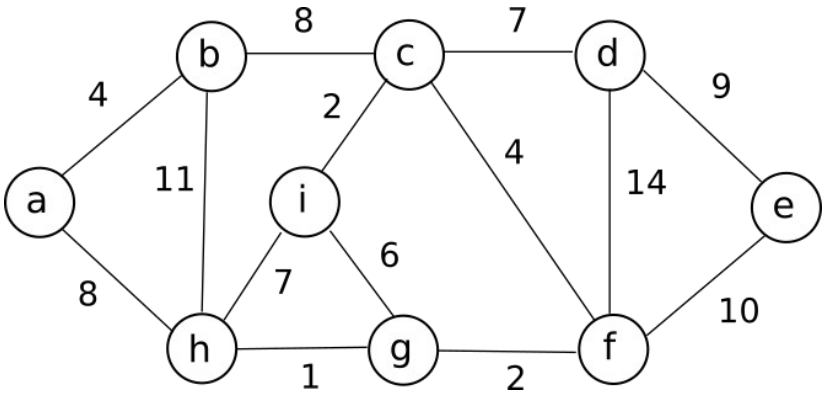
Assim como o Kruskal, também é considerado um algoritmo guloso.

Definições de variáveis
$\lambda(v)$: menor custo estimado de entrada para o vértice v (até o presente momento)
$\pi(v)$: predecessor de v na árvore (vértice pelo qual entrei em v)
Q: fila de prioridades dos vértices (maior prioridade = menor $\lambda(v)$)

Como sei que atingi o menor valor de entrada para um vértice v ?
Basta olhar na fila Q. Quando v é removido da fila, seu custo é mínimo ($\lambda(v)$ não muda mais). Para todo vértice v inserido em S, é garantido que o menor $\lambda(v)$ já foi encontrado.

PSEUDOCÓDIGO

```
MST_Prim(G, w, r) {
        for each  v∈V  {
                λ(v)=∞
                π(v)=nil
        }
          λ(r)=0      (raiz deve iniciar com custo zero pois é primeira a sair da fila)
        Q = V                 (fila de prioridades inicial é todo conjunto de vértices)
          S=∅
        while  Q≠∅  {
                u = ExtractMin(Q)                (remove da fila o vértice de menor prioridade)
                  S=S∪{u}                        (já obtive o menor custo de entrada para u)
                for each  v∈N(u)  {
                (se não estiver na fila Q, não pode mais modificar  λ(v)  )
                        if (  v∈Q  and  λ(v)>w(u,v)  ) {        (entrada de menor custo)
                                λ(v)=w(u,v)                     (atualiza o custo para o menor valor)
                                π(v)=u                          (muda o predecessor)
                        }
                }
        }
}
```



OBS: Note que os códigos a seguir fazem exatamente a mesma operação

```
if (  v∈Q  and  λ(v)>w(u,v)  ) {
        λ(v)=w(u,v)
        π(v)=u
}
```

```
if  v∈Q  {
        λ(v)=min{λ(v),w(u,v)}
        if (  λ(v)  was updated )
                π(v)=u
}
```

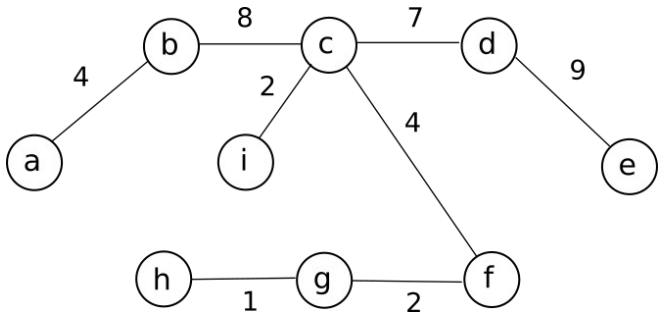
Fila

	a	b	c	d	e	f	g	h	i
$\lambda^{(0)}(v)$	0	∞	∞	∞	∞	∞	∞	∞	∞
$\lambda^{(1)}(v)$		4	∞	∞	∞	∞	∞	8	∞
$\lambda^{(2)}(v)$			8	∞	∞	∞	∞	8	∞
$\lambda^{(3)}(v)$				7	∞	4	∞	8	2

$\lambda^{(4)}(v)$				7	∞	4	6	7	
$\lambda^{(5)}(v)$				7	10		2	7	
$\lambda^{(6)}(v)$				7	10			1	
$\lambda^{(7)}(v)$					9				

Ordem de acesso aos vértices

u	$V'=\{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, h}	$\lambda(b)=min\{\lambda(b), w(a,b)\}=min\{\infty, 4\}=4$	$\pi(b)=a$
		$\lambda(h)=min\{\lambda(h), w(a,h)\}=min\{\infty, 8\}=8$	$\pi(h)=a$
b	{c, h}	$\lambda(c)=min\{\lambda(c), w(b,c)\}=min\{\infty, 8\}=8$	$\pi(c)=b$
		$\lambda(h)=min\{\lambda(h), w(b,h)\}=min\{8, 11\}=8$	---
c	{d, f, i}	$\lambda(d)=min\{\lambda(d), w(c,d)\}=min\{\infty, 7\}=7$	$\pi(d)=c$
		$\lambda(f)=min\{\lambda(f), w(c,f)\}=min\{\infty, 4\}=4$	$\pi(f)=c$
		$\lambda(i)=min\{\lambda(i), w(c,i)\}=min\{\infty, 2\}=2$	$\pi(i)=c$
i	{h, g}	$\lambda(h)=min\{\lambda(h), w(i,h)\}=min\{8, 7\}=7$	$\pi(h)=i$
		$\lambda(g)=min\{\lambda(g), w(i,g)\}=min\{\infty, 6\}=6$	$\pi(g)=i$
f	{d, e, g}	$\lambda(d)=min\{\lambda(d), w(f,d)\}=min\{7, 14\}=7$	---
		$\lambda(e)=min\{\lambda(e), w(f,e)\}=min\{\infty, 10\}=10$	$\pi(e)=f$
		$\lambda(g)=min\{\lambda(g), w(f,g)\}=min\{6, 2\}=2$	$\pi(g)=f$
g	{h}	$\lambda(h)=min\{\lambda(h), w(g,h)\}=min\{7, 1\}=1$	$\pi(h)=g$
h	$\emptyset$	---	---
d	{e}	$\lambda(e)=min\{\lambda(e), w(d,e)\}=min\{10, 9\}=9$	$\pi(e)=d$
e	$\emptyset$	---	---



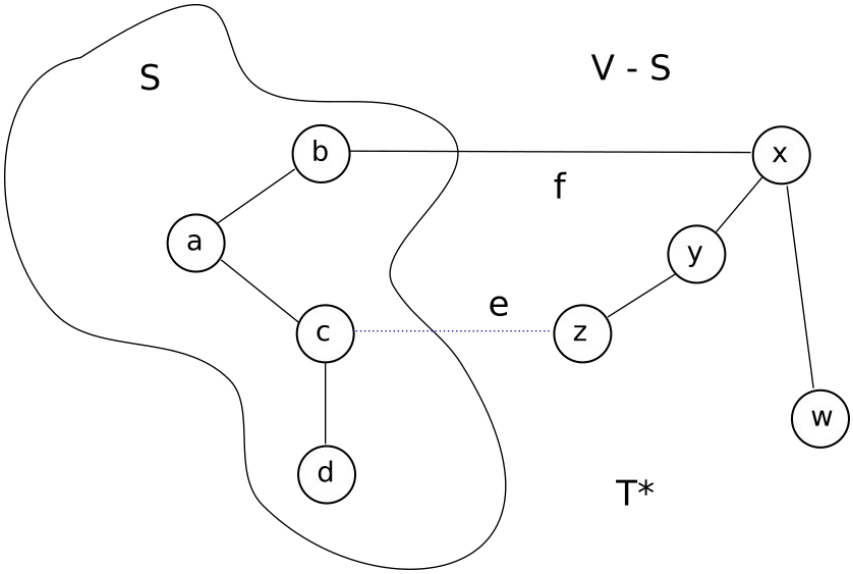
MST
w(T) = 37

Mapa de predecessores (árvore final)

v	a	b	c	d	e	f	g	h	i
$\pi(v)$	--	a	b	c	d	c	f	g	c
$\lambda(v)$	0	4	8	7	9	4	2	1	2

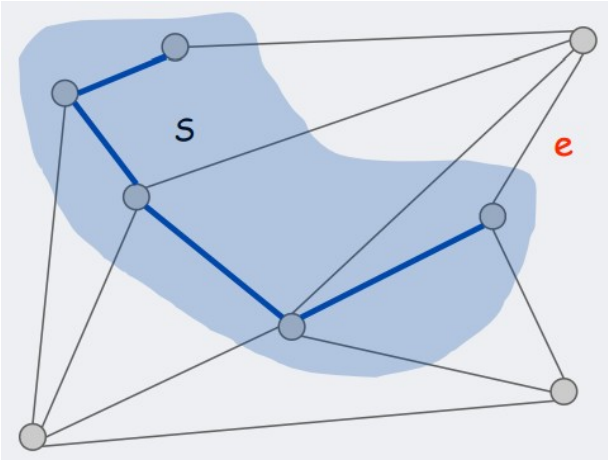
A seguir veremos um resultado fundamental para provar a otimalidade do algoritmo de Prim.

Propriedade do corte: Seja G = (V, E, w) um grafo, S um subconjunto qualquer de V e $e \in E$ a aresta de menor custo com exatamente uma extremidade em S. Então, a MST de G contem e.



Prova por contradição: Seja T* uma MST de G. Suponha que $e \notin T^*$. Ao adicionar e em T* cria-se um único ciclo C. Para que T* - e fosse uma MST, tem que haver alguma outra aresta f com apenas uma extremidade em S (senão T* seria desconexo). Então $T = T^* + e - f$ também é árvore geradora. Como, w(e) < w(f), segue que T tem peso mínimo. Portanto, T* não pode ser MST de G.

Teorema: A árvore T obtida pelo algoritmo de Prim é uma MST de G.
Seja S o subconjunto de vértices de G na árvore T (definido pelo algoritmo). O algoritmo de Prim adiciona em T a cada passo a aresta de menor custo com apenas um vértice extremidade em S. Portanto, pela propriedade do corte, toda aresta e adicionada pertence a MST de G.



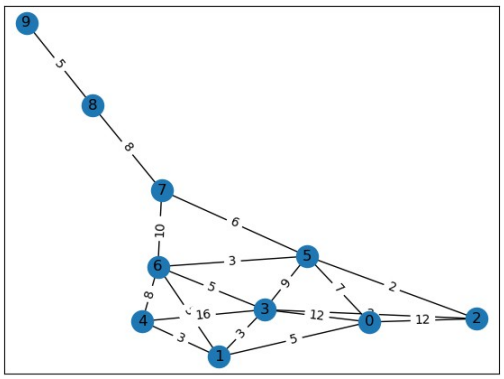
Quanto a complexidade computacional do algoritmo de Prim, pode-se mostrar que no pior caso, em que temos um grafo completo Kn, onde cada vértice se liga a todos os outros vértices, em cada iteração são verificadas n – 1 arestas. Como temos n iterações, o custo computacional é quadrático, ou seja $O(n^2)$.

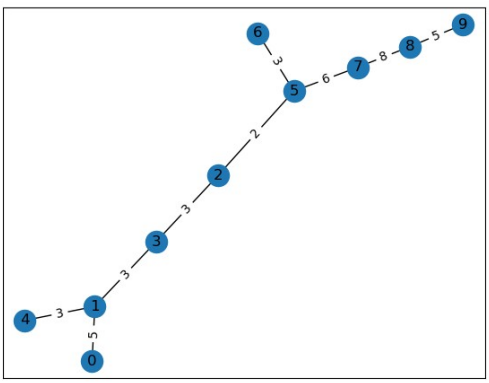
O código em Python a seguir mostra uma implementação do algoritmo de Prim para encontrar a MST de uma grafo em que as arestas são ponderadas aleatoriamente.

```
# Adiciona bibliotecas auxiliares
import random
import networkx as nx
import matplotlib.pyplot as plt
# Extrai o vértice de menor lambda da fila Q
def extractMin(Q, H):
    # Encontra o vértice de menor lambda em Q
    u = Q[0]
    for node in Q:
        # O vértice u armazena o vértice de menor lambda até o momento
        if H.nodes[node]['lambda'] < H.nodes[u]['lambda']:
            u = node
    # Remove u da fila
    del Q[Q.index(u)]
    # Retorna u
    return u

def Prim(G, s):
    # Dicionário para armazenar o mapa de predecessores
    P = {}      # Estrutura de dados que mapeia uma chave a um valor
    # Inicialização do algoritmo
    for v in G.nodes():
        G.nodes[v]['lambda'] = float('inf')
    # Custo para a raiz é 0
    G.nodes[s]['lambda'] = 0
    # Iniciar fila Q com todos os vértices
    Q = list(G.nodes())
    # Enquanto fila não estiver vazia
    while (len(Q) > 0):
        # Obter o primeiro elemento da fila
        u = extractMin(Q, G)
        # Para cada vertice adjacente a u
        for v in G.neighbors(u):
            # Se v está na fila e possui lambda > que peso de (u, v)
            if (v in Q and G.nodes[v]['lambda'] > G[u][v]['weight']):
                # Atualizar custo de v
                G.nodes[v]['lambda'] = G[u][v]['weight']
                # Adicionar u como antecessor de v
                P[v] = u

    # Retorna a lista de predecessores
    return P


if __name__ == '__main__':
    # Cria um grafo de exemplo
    G = nx.krackhardt_kite_graph()
    # Adicionando peso nas arestas
    for u, v in G.edges():
        G[u][v]['weight'] = random.randint(1, 10)


    print('Plotando grafo...')
    # Cria figura para plotagem
    plt.figure(1)
```

```
# Há vários layouts, mas spring é um dos mais bonitos
pos = nx.spring_layout(G)
nx.draw_networkx(G, pos, with_labels=True)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
# Exibir figura
plt.show()

# Aplicando Busca em Largura
P = Prim(G, 0)      # retorna as arestas que compõem a BFS_tree
# Cria um grafo vaxio para árvore da busca em largura
T = nx.Graph()
# Inserir arestas em T
T.add_edges_from([ (u, v) for u, v in P.items() ])
# Obtém os pesos das arestas da árvore a partir de G
for u, v in T.edges():
    T[u][v]['weight'] = G[u][v]['weight']

# Cria figura para plotagem
plt.figure(2)
# Define o layout
pos = nx.spring_layout(T)
# Plotar vertices de T
nx.draw_networkx(T, pos, with_labels=True)
# Plotar pesos das arestas
labels = nx.get_edge_attributes(T, 'weight')
nx.draw_networkx_edge_labels(T, pos, edge_labels=labels)
# Exibir figura
plt.show()
```

As figuras a seguir ilustram o grafo de entrada e sua respectiva MST.





Aula 11 – Grafos: Caminhos mínimos e o algoritmo de Dijkstra

Na busca por caminhos mínimos em grafos,existem 3 subproblemas principais:
i) Menor caminho de s a t (1 para 1): solução ótima é dada pelo algoritmo de Dijkstra
ii) Árvore de caminhos mínimos de s a todos os demais vértices de G (1 para N): solução ótima é dada pelo algoritmo de Dijkstra
iii) Matriz de distâncias geodésicas ponto a ponto (N para N): solução ótima pode ser obtida através de N execuções do algoritmo de Dijkstra

Portanto, com o algoritmo de Dijkstra podemos resolver qualquer tipo de problema de caminhos mínimos em grafos ponderados. Essa é uma das grandes vantagens desse método.

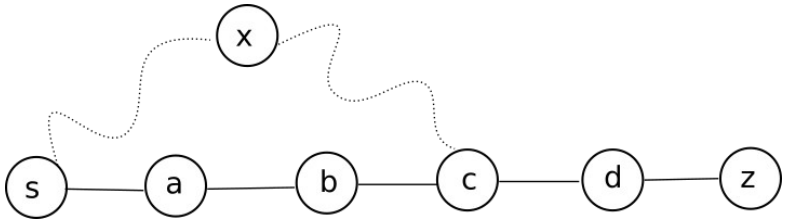
Def: Caminho ótimo
Seja G = (V, E) e $w: E \rightarrow R^+$ uma função de custo para as arestas. Um caminho P* de v0 a vn é ótimo se seu peso

$$w(P^*) = \sum_{i=0}^{n-1} w(v_i, v_{i+1}) = w(v_0, v_1) + w(v_1, v_2) + ... + w(v_{n-1}, v_n)$$ (soma dos pesos das arestas)

é o menor possível.

Teorema: Todo subcaminho de um caminho ótimo é ótimo
Prova por contradição

1. Suponha no grafo a seguir que o caminho ótimo de s a z é P* = sabcdz



2. Suponha que subcaminhos de caminhos ótimos não sejam ótimos (negação da conclusão), ou seja, que exista um caminho mínimo de s a c que não passa por a e b mas sim por x, ou seja, P' = sxc.

3. Ora, se isso é verdade, então claramente P* não é ótimo pois é possível minimizá-lo ainda mais, criando P = sxcdz, o que gera uma contradição. Assim, subcaminhos de caminhos ótimos devem ser ótimos

Importante resultado na obtenção de algoritmos de programação dinâmica (problema com subestrutura ótima, ou seja, é possível utilizar partes de soluções já obtidas na construção da solução final). Em outras palavras, é isso que permite a utilização de programação dinâmica em algoritmos de caminhos mínimos. Antes de introduzirmos os algoritmos, iremos apresentar uma primitiva comum a todos eles. Trata-se da função relax, que aplica a operação conhecida como relaxamento a uma aresta de um grafo ponderado.

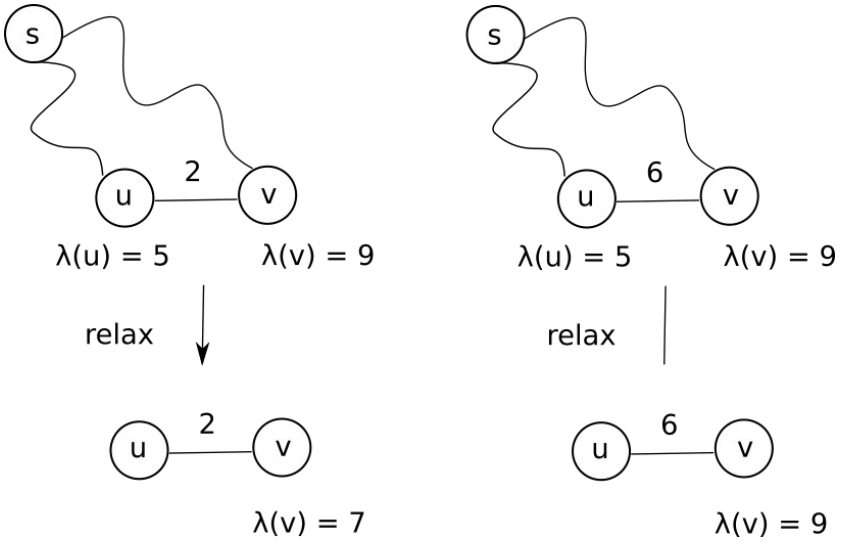
Primitiva relax

relax(u, v, w): relaxar a aresta (u, v) de peso w

Para entender o que significa relaxar a aresta (u,v) de peso w, precisamos definir $\lambda(u)$ e $\lambda(v)$

Quem é $\lambda(u)$? É o custo atual de sair da origem s e chegar até u
Quem é $\lambda(v)$? É o custo atual de sair da origem s e chegar até v

Ideia geral: é uma boa ideia passar por u para chegar em v sabendo que o custo de ir de u até v é w?



Obs: A operação relax(u, v, w) nunca aumenta o valor de $\lambda(v)$, apenas diminui

PSEUDOCODIGO

```
relax(u, v, w)
{
        if  λ(v)>λ(u)+w(u,v)
        {
                λ(v)=λ(u)+w(u,v)
                π(v)=u
        }
}
```

```
relax(u, v, w)
{
        λ(v)=min{λ(v),λ(u)+w(u,v)}
        if  λ(v)  was updated
                π(v)=u
}
```

O que varia nos diversos algoritmos para encontrar caminhos mínimos são os seguintes aspectos:
i) Quantas e quais arestas devemos relaxar?
ii) Quantas vezes devemos relaxar as arestas?
iii) Em que ordem devemos relaxar as arestas?

A seguir veremos um algoritmo muito mais eficiente para resolver o problema: o algoritmo de Dijkstra. Basicamente, esse algoritmo faz uso de uma política de gerenciamento de vértices baseada em aspectos de programação dinâmica. O que o método faz é basicamente criar uma fila de prioridades para organizar os vértices de modo que quanto menor o custo $\lambda(v)$ maior a prioridade do vértice em questão. Assim, a ideia é expandir primeiramente os menores ramos da árvore de caminhos mínimos, na expectativa de que os caminhos mínimos mais longos usarão como base os subcaminhos obtidos anteriormente. Trata-se de um mecanismo de reaproveitar soluções de subproblemas para a solução do problema como um todo.

Definição das variáveis
$\lambda(v)$: menor custo até o momento para o caminho s-v
$\pi(v)$: predecessor de v na árvore de caminhos mínimos

Q: fila de prioridades dos vértices (maior prioridade = menor $\lambda(v)$)

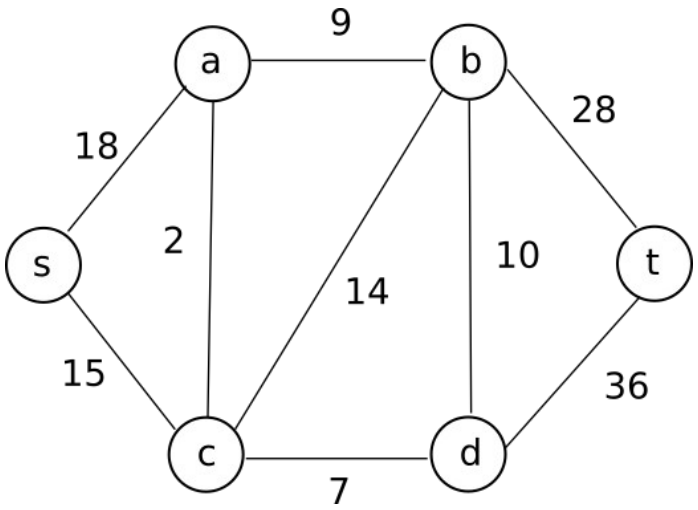
PSEUDOCODIGO

```
Dijkstra(G, w, s)
{
        for each   v∈V
        {
                λ(v)=∞
                π(v)=nil
        }
         λ(s)=0
          π(s)=nil
        Q = V          (fila de prioridades)
        while   Q≠Ø
        {
                u = ExtractMin(Q)
                  S=S∪{u}
                for each   v∈N(u)
                        relax(u, v, w)          // relaxa toda aresta incidente a u
        }                                       // usa sub-caminho ótimo para gerar o caminho maior
}                                               // por isso não precisa relaxar todas as arestas
```

Algoritmos e estruturas de dados
BFS – Fila
DFS – Pilha
Dijkstra – Fila de prioridades

Obs: Note que o algoritmo de Dijkstra é uma generalização da busca em largura
Ambos crescem primeiro os menores ramos da árvore. A BFS toda aresta tem mesmo tamanho, no Dijkstra esse tamanho é variável.

Ex:



Fila

	s	a	b	c	d	t
$\lambda^{(0)}(v)$	0	∞	∞	∞	∞	∞
$\lambda^{(1)}(v)$		18	∞	15	∞	∞
$\lambda^{(2)}(v)$		17	29		22	∞
$\lambda^{(3)}(v)$			26		22	∞
$\lambda^{(4)}(v)$			26			58
$\lambda^{(5)}(v)$						54

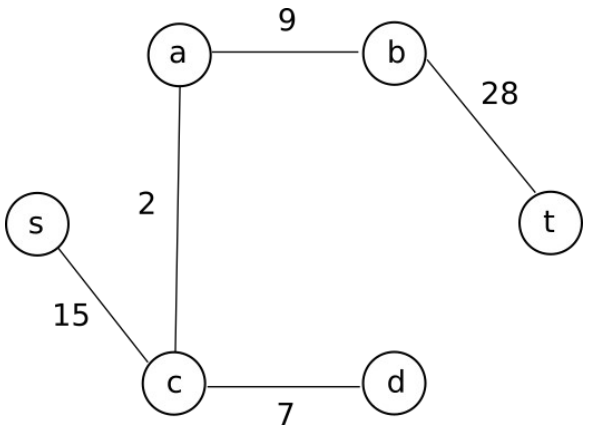
Ordem de acesso aos vértices

u	$V'=\{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
s	{a, c}	$\lambda(a)=min\{\lambda(a),\lambda(s)+w(s,a)\}=min\{\infty,18\}=18$	$\pi(a)=s$
		$\lambda(c)=min\{\lambda(c),\lambda(s)+w(s,c)\}=min\{\infty,15\}=15$	$\pi(c)=s$
c	{a, b, d}	$\lambda(a)=min\{\lambda(a),\lambda(c)+w(c,a)\}=min\{18,17\}=17$	$\pi(a)=c$
		$\lambda(b)=min\{\lambda(b),\lambda(c)+w(c,b)\}=min\{\infty,29\}=29$	$\pi(b)=c$
		$\lambda(d)=min\{\lambda(d),\lambda(c)+w(c,d)\}=min\{\infty,22\}=22$	$\pi(d)=c$
a	{b}	$\lambda(b)=min\{\lambda(b),\lambda(a)+w(a,b)\}=min\{29,26\}=26$	$\pi(b)=a$
d	{b, t}	$\lambda(b)=min\{\lambda(b),\lambda(d)+w(d,b)\}=min\{26,32\}=26$	---
		$\lambda(t)=min\{\lambda(t),\lambda(d)+w(d,t)\}=min\{\infty,58\}=58$	$\pi(t)=d$
b	{t}	$\lambda(t)=min\{\lambda(t),\lambda(b)+w(b,t)\}=min\{58,54\}=54$	$\pi(t)=b$
t	$\emptyset$	---	---

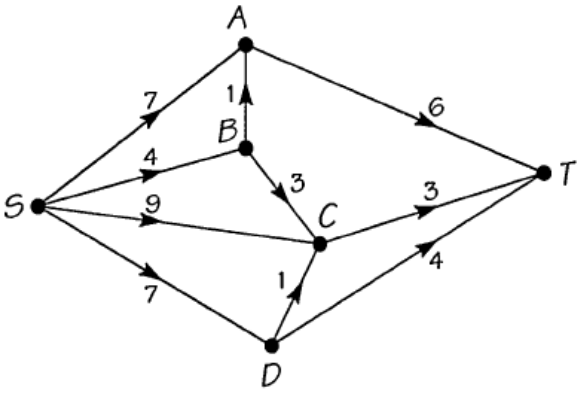
Mapa de predecessores (árvore final)

v	s	a	b	c	d	t
$\pi(v)$	---	c	a	s	c	b

Árvore de caminhos mínimos (armazena os menores caminhos de s a todos os demais vértices)



Ex: Utilizando o algoritmo de Dijkstra, construa a árvore de caminhos mínimos (trace completo)



A seguir iremos demonstrar a otimalidade do algoritmo de Dijkstra.

<u>Teorema:</u> O algoritmo de Dijkstra termina com $\lambda(v)=d(s,v), \forall\, v\in V$
(Prova por contradição)

Obs: Note que sempre $\lambda(v)\geq d(s,v)$ (*)

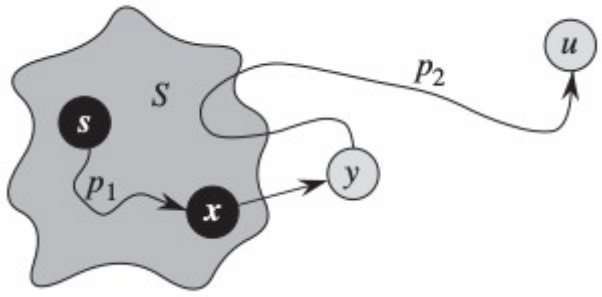
1. Suponha que u seja o 1º vértice para o qual $\lambda(u)\neq d(s,u)$ quando u entra em S.

2. Então, $u\neq s$ pois senão $\lambda(s)=d(s,s)=0$

3. Assim, existe um caminho P_{su} pois senão $\lambda(u)=d(s,u)=\infty$. Portanto, existe um caminho mínimo P^*_{su}

4. Antes de adicionar u a S, P^*_{su} possui $s\in S$ e $u\in V-S$

5. Seja y o 1º vértice em P^*_{su} tal que $y\in V-S$ e seja x seu predecessor ($x\in S$)



P^*_{su} = s → xy → u
p1 p2

Obs: Note que tanto p1 quanto p2 não precisam ter arestas

6. Como $x\in S$, $\lambda(x)=d(s,x)$ e no momento em que ele foi inserido a S, a aresta (x,y) foi relaxada, ou seja:

$$\lambda(y)=\lambda(x)+w(x,y)=d(s,x)+w(x,y)=d(s,y)$$

7. Mas y antecede a u no caminho e como $w: E \rightarrow R^{+}$ (pesos positivos), temos:

$$d(s,y) \leq d(s,u)$$

e portanto

$$\lambda(y) = d(s,y) \leq d(s,u) \leq \lambda(u)$$
$$\quad (6) \qquad (7) \qquad (*)$$

8. Mas como ambos y e u pertencem a V – S, quando u é escolhido para entrar em S temos $\lambda(u) \leq \lambda(y)$

9. Como $\lambda(y) \leq \lambda(u)$ e $\lambda(u) \leq \lambda(y)$ então temos que $\lambda(u) = \lambda(y)$, o que implica em:

$$\lambda(y) = d(s,y) = d(s,u) = \lambda(u)$$

o que gera uma contradição. Portanto $\nexists u \in V$ tal que $\lambda(u) \neq d(s,u)$ quando u entra em S.

Assim como o algoritmo de Prim, no pior caso, o algoritmo de Dijkstra possui complexidade quadrática, ou seja, O(n2), uma vez que em grafos completos, cada vértice se liga a todos os demais (possuem grau n – 1) e como temos n vértices na fila de prioridades, o número de operações é uma função quadrática do número de vértices n

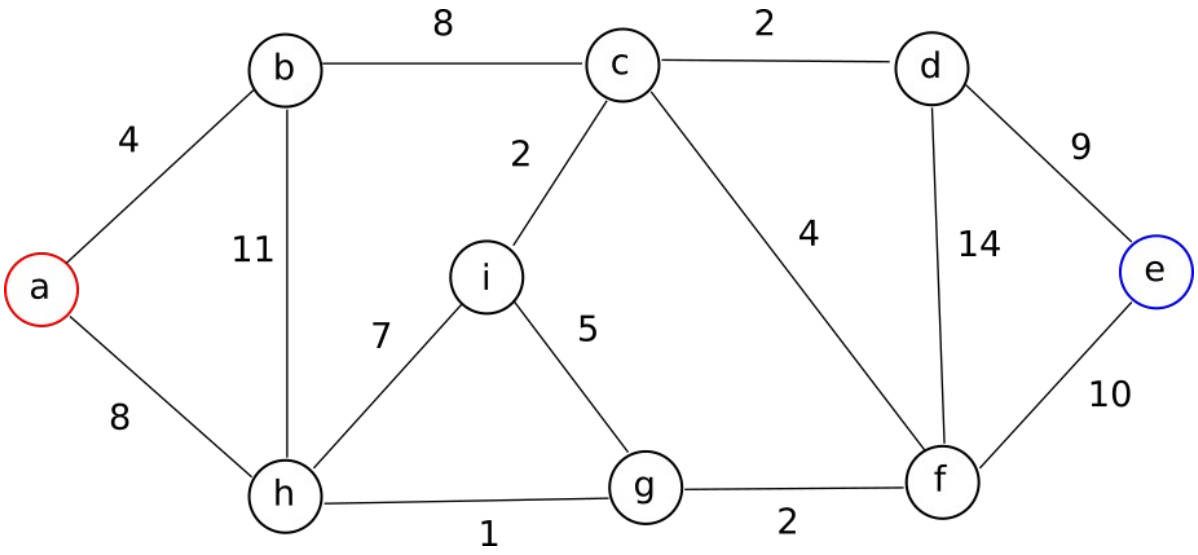
Dijkstra multisource

Ideia: utilizar múltiplas sementes/raízes

Processo de competição: cada vértice pode ser conquistado por apenas uma das sementes (pois ao fim, um vértice só pode estar pendurado em uma única árvore)
Durante a execução do algoritmo, nesse processo de conquista, uma semente pode “roubar” um nó de seus concorrentes, oferecendo a ele um caminho menor que o atual

Ao final temos o que se chama de floresta de caminhos ótimos, composta por várias árvores (uma para cada semente)

Cada árvore representa um agrupamento/comunidade.



Desejamos encontrar 2 agrupamentos. Para isso, utilizaremos 2 sementes: os vértices A e E. Na prática, isso significa inicializar o algoritmo de Dijkstra com $\lambda(a)=\lambda(e)=0$

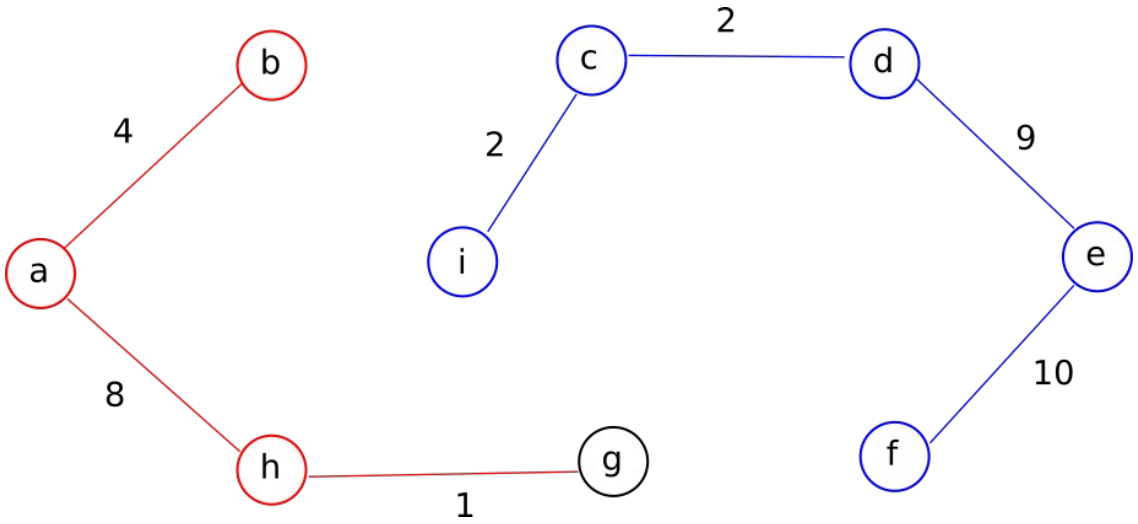
Fila

	a	b	c	d	e	f	g	h	i
$\lambda^{(0)}(v)$	0	∞	∞	∞	0	∞	∞	∞	∞
$\lambda^{(1)}(v)$		4	∞	∞	0	∞	∞	8	∞
$\lambda^{(2)}(v)$		4	∞	9		10	∞	8	∞
$\lambda^{(3)}(v)$			12	9		10	∞	8	∞
$\lambda^{(4)}(v)$			12	9		10	9		15
$\lambda^{(5)}(v)$			11			10	9		15
$\lambda^{(6)}(v)$			11			10			14
$\lambda^{(7)}(v)$			11						14
$\lambda^{(8)}(v)$									13

Ordem de acesso aos vértices

u	$V'=\{v\in N(u)\wedge v\in Q\}$	$\lambda(v),\forall v\in V'$	$\pi(v)$
a	{b, h}	$\lambda(b)=min\{\infty,4\}=4$	$\pi(b)=a$
		$\lambda(h)=min\{\infty,8\}=8$	$\pi(h)=a$
e	{d, f}	$\lambda(d)=min\{\infty,9\}=9$	$\pi(d)=e$
		$\lambda(f)=min\{\infty,10\}=10$	$\pi(f)=e$
b	{c, h}	$\lambda(c)=min\{\infty,4+8\}=12$	$\pi(c)=b$
		$\lambda(h)=min\{8,4+11\}=8$	----------
h	{i, g}	$\lambda(i)=min\{\infty,8+7\}=15$	$\pi(i)=h$
		$\lambda(g)=min\{\infty,8+1\}=9$	$\pi(g)=h$
d	{c, f}	$\lambda(c)=min\{12,9+2\}=11$	$\pi(c)=d$
		$\lambda(f)=min\{10,9+14\}=10$	----------
g	{f, i}	$\lambda(f)=min\{10,9+14\}=10$	----------
		$\lambda(i)=min\{15,9+5\}=14$	$\pi(i)=g$
f	{c}	$\lambda(c)=min\{11,10+4\}=11$	----------
c	{i}	$\lambda(i)=min\{14,11+2\}=13$	$\pi(i)=c$
i	$\emptyset$	----------	----------

Floresta de caminhos ótimos



A heurística A*

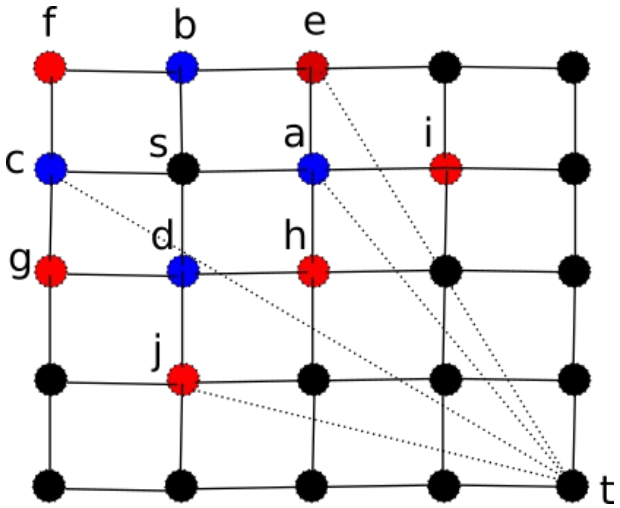
É uma técnica aplicada para acelerar a busca por caminhos mínimos em certos tipos de grafos. Pode ser considerado uma generalização do algoritmo de Dijkstra. Um dos problemas com o algoritmo de Dijkstra é não levar em consideração nenhuma informação sobre o destino. Em grafos densamente conectados esse problema é amplificado devido ao alto número de arestas e aos muitos caminhos a serem explorados. Em suma, o algoritmo A* propõe uma heurística para dizer o quanto estamos chegando próximos do destino através da modificação da prioridades dos vértices na fila Q. É um algoritmo muito utilizado na IA de jogos eletrônicos.

Ideia: modificar a função que define a prioridade dos vértices

$\alpha(v)=\lambda(v)+\gamma(v)$ onde

$\lambda(v)$: custo atual de ir da origem s até v
$\gamma(v)$: custo estimado de v até o destino t (alvo)

Desafio: como calcular $\gamma(v)$?
- É viável apenas alguns casos específicos



Considere o grafo acima. O vértice s é a origem e o vértice t é o destino. Neste caso temos:

$$\lambda(a)=\lambda(b)=\lambda(c)=\lambda(d)=1$$

o que significa que no Dijkstra, todos eles teriam a mesma prioridade. Note porém que, utilizando a distância Euclidiana para obter uma estimativa de distância até a origem, temos:

$$\gamma(a)=\gamma(d)=\sqrt{4+9}=\sqrt{13}$$
$$\gamma(b)=\gamma(c)=\sqrt{9+16}=\sqrt{25}=5$$

Ou seja, no A*, devemos priorizar a e d em detrimento de b e c uma vez que

$$1+\sqrt{13}<1+5$$

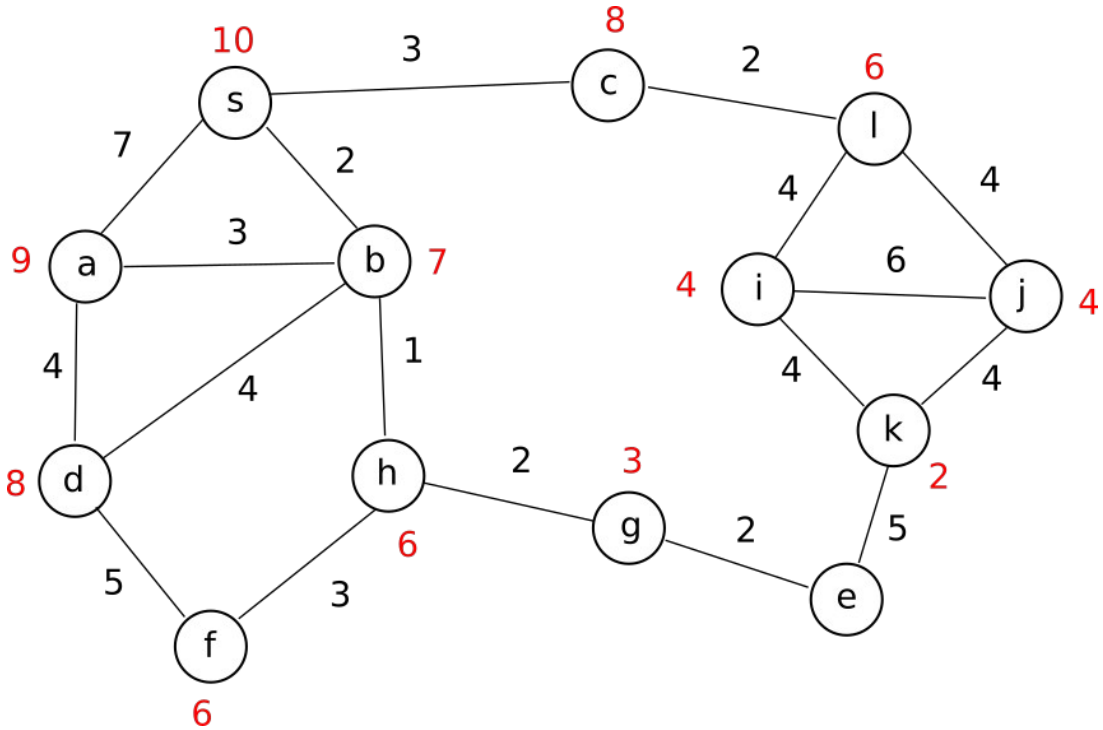
e portanto a e d saem da fila de prioridades antes. Isso ocorre pois no A* eles são considerados mais importantes. O mesmo ocorre nos demais níveis

$$\lambda(e)=\lambda(f)=\lambda(g)=\lambda(h)=\lambda(i)=\lambda(j)=2$$

$$\gamma(e)=\sqrt{18} \qquad \gamma(j)=\sqrt{10} \qquad \gamma(h)=\sqrt{8}$$

A ideia é que o alvo t atraia o caminho. Se t se move, a busca por caminhos mínimos usando A* costuma ser bem mais eficiente que o Dijkstra em casos como esse.

Ex: O grafo ponderado a seguir ilustra um conjunto de cidades e os pesos das arestas são as distâncias entre elas. Estamos situados na cidade s e deseja-se encontrar um caminho mínimo até a cidade e. O números em vermelho indicam o valor de $\gamma(v)$, ou seja, são uma estimativa para a distância de v até o destino e. Execute o algoritmo A* para obter o caminho mínimo de s a e.



Fila

	s	a	b	c	d	e	f	g	h	i	j	k	l
$\gamma^{(0)}(v)$	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$\gamma^{(1)}(v)$		7+9	2+7	3+8	∞	∞	∞	∞	∞	∞	∞	∞	∞
$\gamma^{(2)}(v)$		5+9		3+8	6+8	∞	∞	∞	3+6	∞	∞	∞	∞
$\gamma^{(3)}(v)$		5+9		3+8	6+8	∞	6+6	5+3					
$\gamma^{(4)}(v)$		5+9		3+8	6+8	7	6+6						

Ordem de acesso aos vértices

u	$V'=\{v\in N(u)\wedge v\in Q\}$	$\lambda(v),\forall v\in V'$	$\pi(v)$
s	{a, b, c}	$\lambda(a)=min\{\infty,7\}=7$	$\pi(a)=s$
		$\lambda(b)=min\{\infty,2\}=2$	$\pi(b)=s$
		$\lambda(c)=min\{\infty,3\}=3$	$\pi(c)=s$
b	{a, d, h}	$\lambda(a)=min\{7,2+3\}=5$	$\pi(a)=b$
		$\lambda(d)=min\{\infty,2+4\}=6$	$\pi(d)=b$
		$\lambda(h)=min\{\infty,2+1\}=3$	$\pi(h)=b$
h	{f, g}	$\lambda(f)=min\{\infty,3+3\}=6$	$\pi(f)=h$
		$\lambda(g)=min\{\infty,3+2\}=5$	$\pi(g)=h$
g	{e}	$\lambda(e)=min\{\infty,5+2\}=7$	$\pi(e)=g$

Note como a ordem de retirada dos vértices da fila é orientada ao destino.

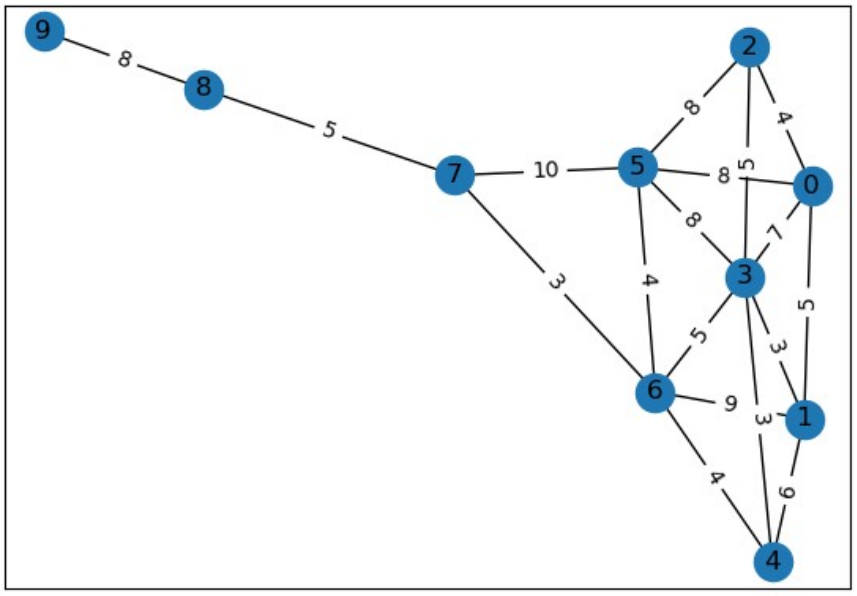
Há um algoritmo que resolve exclusivamente o problema de calcular as distâncias geodésicas entre todos os pares de vértices do grafo: o algoritmo de Floyd-Warshall, cuja complexidade computacional é $O(n^3)$. Em termos de custo computacional é equivalente a executar N vezes o algoritmo de Dijkstra. O código em Python a seguir mostra uma implementação do algoritmo de Djikstra com o auxílio do biblioteca NetworkX.

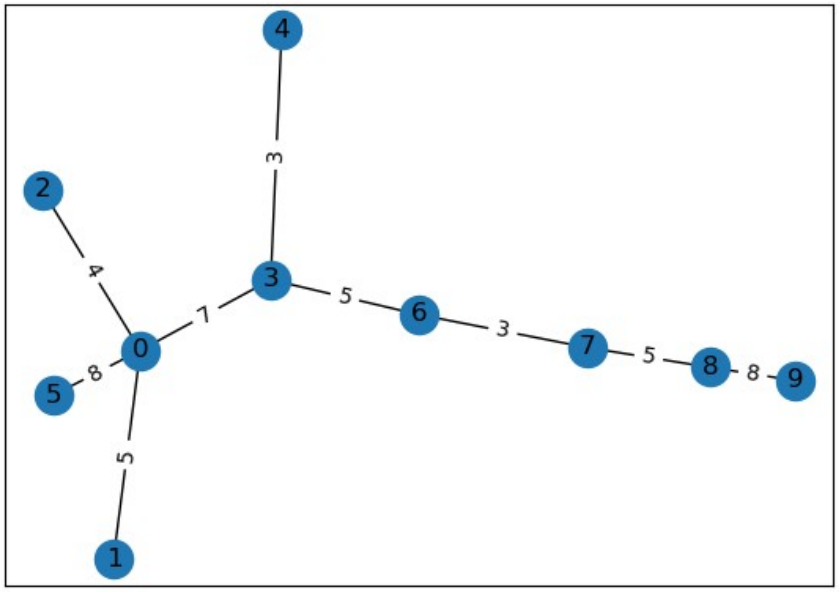
```
# Adiciona bibliotecas auxiliares
import random
import networkx as nx
import matplotlib.pyplot as plt

# Extrai o vértice de menor lambda da fila Q
def extractMin(Q, H):
    # Enconrta o menor vértice de menor lambda
    u = Q[0]
    for node in Q:
        # O vértice u armazena o vértice de menor lambda até o momento
        if H.nodes[node]['lambda'] < H.nodes[u]['lambda']:
            u = node
    # Remove u da fila
    del Q[Q.index(u)]
    return u
```

```
def Dijkstra(G, s):
    # dicionário para armazenar o mapa de predecessores
    P = {} # estrutura de dados que mapeia uma chave a um valor
    # inicialização do algoritmo
    for v in G.nodes():
        G.nodes[v]['lambda'] = float('inf')
    # custo para a raiz é 0
    G.nodes[s]['lambda'] = 0
    # iniciar fila Q com todos os vértices
    Q = list(G.nodes())
    # enquanto fila não estiver vazia
    while (len(Q) > 0):
        # obter o primeiro elemento da fila
        u = extractMin(Q, G)
        # para cada vertice adjacênte a u
        for v in G.neighbors(u):
            if (v in Q and G.nodes[v]['lambda'] > G.nodes[u]['lambda']
                                                  + G[u][v]['weight']):
                # atualizar custo de v
                G.nodes[v]['lambda']=G.nodes[u]['lambda']+G[u][v]['weight']
                # adicionar u como antecessor de v
                P[v] = u

    # retorna a lista de antecessores
    return P

if __name__ == '__main__':
    # Cria um grafo de exemplo
    G = nx.krackhardt_kite_graph()
    # Adicionando peso nas arestas
    for u, v in G.edges():
        G[u][v]['weight'] = random.randint(1, 10)

    print('Plotando grafo...')
    # Cria figura para plotagem
    plt.figure(1)
    # Há vários layouts, mas spring é um dos mais bonitos
    pos = nx.spring_layout(G)
    nx.draw_networkx(G, pos, with_labels=True)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    # Exibir figura
    plt.show()

    # Aplicando Busca em Largura
    P = Dijkstra(G, 0)      # retorna as arestas que compõem a BFS_tree
    # Cria um grafo vaxio para árvore da busca em largura
    T = nx.Graph()
    # Inserir arestas em T
    T.add_edges_from([ (u, v) for u, v in P.items() ])
    # Obtém os pesos das arestas da árvore a partir de G
    for u, v in T.edges():
        T[u][v]['weight'] = G[u][v]['weight']

    # Cria figura para plotagem
    plt.figure(2)
    # Define o layout
    pos = nx.spring_layout(T)
    # Plotar vertices de T
```

```
    nx.draw_networkx(T, pos, with_labels=True)
    # Plotar pesos das arestas
    labels = nx.get_edge_attributes(T, 'weight')
    nx.draw_networkx_edge_labels(T, pos, edge_labels=labels)
    # Exibir figura
    plt.show()
```

As figuras a seguir ilustram o grafo original e a árvore de caminhos mínimos obtida.





Bibliografia

MILLER, B. e RANUM, D. Problem Solving with Algorithms and Data Structures using Python, 2011. Disponível em online em:
https://runestone.academy/runestone/books/published/pythonds3/index.html

MENEZES, N. N. C. Introdução à programação com Python: algoritmos e lógica de programação para iniciantes. 2. ed. São Paulo: Novatec, 2014.

BORGES, L. E.; Python para Desenvolvedores. 2. ed., 2010. Disponível em
https://ark4n.files.wordpress.com/2010/01/python_para_desenvolvedores_2ed.pdf

MILLER, B.; RANUM, D.; Como Pensar como um Cientista da Computação: Versão Interativa. Disponível em: https://panda.ime.usp.br/pensepy/static/pensepy/index.html

GOODRICH, M., TAMASSIA, R. e GOLDWASSER, M. Data Structures and Algorithms in Python.

LEE, K. D., HUBBARD, S., Data structures and algorithms with Python, Undergraduate Topics in Computer Science, Springer, 2015.

NECAISE, R. D., Data structures and algorithms using Python, John Wiley & Sons, 2011.

CORMEN, T. H. et al. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, 2012.

MEDINA, M., FERTIG, C. Algoritmos e programação: teoria e prática. 2. ed. São Paulo: Novatec Editora, 2006.

WOLFRAM, S. A New Kind of Science, Wolfram Research, 2002.

Game of Life in Python
https://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/

Autômatos celulares
https://en.wikipedia.org/wiki/Rule_110,
http://www.complex-systems.com/pdf/15-1-1.pdf

MAY, R. M. "Simple mathematical models with very complicated dynamics". *Nature*. **261** (5560): 459–467, 1976. Available at: *http://abel.harvard.edu/archive/118r_spring_05/docs/may.pdf*

Sobre o autor

Alexandre L. M. Levada é bacharel em Ciências da Computação pela Universidade Estadual Paulista “Júlio de Mesquita Filho” (UNESP), mestre em Ciências da Computação pela Universidade Federal de São Carlos (UFSCar) e doutor em Física Computacional pela Universidade de São Paulo (USP). Atualmente é professor adjunto no Departamento de Computação da Universidade Federal de São Carlos e seus interesses em pesquisa são: filtragem de ruído em imagens e aprendizado de métricas via redução de dimensionalidade para problemas de classificação de padrões.