**CPS740 - Algoritmos e Grafos - Lista 1**

**Thiago Guimarães Rebello Mendonca de Alcantara - DRE: 118053123**

Tudo esta no repositorio <https://github.com/guim4dev/CPS740>

***Questão 1)***

* **Algoritmo de Ordenação 1:**

def **insertion\_sort**(list):

for i in range(1, len(list)):

current\_item = list[i]

j = i - 1

while current\_item < list[j] and j >= 0:

list[j + 1] = list[j]

j -= 1

list[j + 1] = current\_item

print(list)

return list

Passo a passo:  
[2, 7, 5, 6, 9, 0, 1, 4, 8, 5, 3]

[2, 5, 7, 6, 9, 0, 1, 4, 8, 5, 3]

[2, 5, 6, 7, 9, 0, 1, 4, 8, 5, 3]

[0, 2, 5, 6, 7, 9, 1, 4, 8, 5, 3]

[0, 1, 2, 5, 6, 7, 9, 4, 8, 5, 3]

[0, 1, 2, 4, 5, 6, 7, 9, 8, 5, 3]

[0, 1, 2, 4, 5, 6, 7, 8, 9, 5, 3]

[0, 1, 2, 4, 5, 5, 6, 7, 8, 9, 3]

[0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9]

* **Algoritmo de Ordenação 2:**

def **quick\_sort**(list):

if len(list) <= 1:

return list

pivot\_index = len(list)//2

pivot = list[pivot\_index]

del list[pivot\_index]

left = []

right = []

for item in list:

if item <= pivot:

left.append(item)

else:

right.append(item)

print(left + [pivot] + right)

return **quick\_sort**(left) + [pivot] + **quick\_sort**(right)

Passo a passo:

[2, 7, 5, 6, 9, 0, 1, 4, 8, 5, 3]

[0, 2, 7, 5, 6, 9, 1, 4, 8, 5, 3]

[1, 2, 7, 5, 6, 9, 4, 8, 5, 3]

[2, 7, 5, 6, 4, 8, 5, 3, 9]

[2, 3, 4, 7, 5, 6, 8, 5]

[2, 3]

[5, 5, 6, 7, 8]

[5, 5]

[7, 8]

[0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9]

***Questão 2)*** Matrizes de adjacência possuem complexidade espacial O(n^2), enquanto Estruturas de adjacência possuem O(n + m). Portanto, a Estrutura de adjacências é preferível quando n + m < n^2 -> m < n^2 - n. Logo, Estruturas de adjacências são preferíveis em questão de armazenamento quando o número de arestas é menor que a diferença entre o quadrado do numero de vertices e o numero de vertices.

***Questão 3)*** O Algoritmo é O(m).

def **possui\_ciclo\_euleriano**(arestas):

memo = {}

for aresta in arestas:

for vertice in aresta:

memo[vertice] = (memo.get(vertice) or 0) + 1

for value in memo.values():

if (value % 2) != 0:

return False

return True

***OBS: Função útil que vai ser utilizada a partir de agora:***

def **get\_adjacents**(vertice, edges):

adjacents = []

for edge in edges:

if vertice in edges:

edge.remove(vertice)

adjacents += edge

return adjacents

***Questão 4)*** Utilizando o algoritmo de bi-coloração para definir se um grafo é bipartido:

Complexidade: O(m+n)

colors = {}

def **bipartido\_color**(V, E):

for vertice in V:

colors[vertice] = -1 # sem cor

for vertice in V:

if colors[vertice] == -1: # tentar pintar

if not(**can\_colorize\_vertice**(vertice, E, 0)):

return False

return True

# colors: 1, 0 e -1 (azul, vermelho e sem cor)

def **can\_colorize\_vertice**(vertice, edges, color):

colors[vertice] = color

adjacents = **get\_adjacents**(vertice, edges) # funcao dos adjacentes

for adjacent in adjacents:

if colors[adjacent] == -1:

if not(**can\_colorize\_vertice**(adjacent, edges, 1-color)):

return False

else:

if colors[adjacent] == color:

return False

return True

Também fiz uma tentativa de um algoritmo de DFS por condição contrária para avaliar se eh bipartido ou nao, este encontra-se aqui: <https://github.com/guim4dev/CPS740/blob/master/Lista1/questao_4.py> , especificamente na linha 4.

***Questão 5) a) É um algoritmo polinomial.***

def **possui\_conjunto\_independente\_k**(vertices, arestas, k):

vertices\_colors = **colorized\_vertices**(vertices, arestas)

independent\_one\_color = []

independent\_zero\_color = []

for key, value in vertices\_colors:

if value == 1:

independent\_one\_color.append(key)

elif value == 0:

independent\_zero\_color.append(key)

if (len(independent\_one\_color) >= k) or (len(independent\_zero\_color) >= k):

return True

return False

colors = {}

def **colorized\_vertices**(V, E): # funcao para colorir os vertices

for vertice in V:

colors[vertice] = -1 # sem cor

for vertice in V:

if colors[vertice] == -1: # tentar pintar

**colorize\_vertice**(vertice, E, 0)

return colors

# colors: 1, 0 e -1 (azul, vermelho e sem cor)

def **colorize\_vertice**(vertice, edges, color):

colors[vertice] = color

adjacents = **get\_adjacents**(vertice, edges)

for adjacent in adjacents:

if colors[adjacent] == -1:

**colorize\_vertice**(adjacent, edges, 1-color)

return True

***Questão 5) b) Não é um algoritmo polinomial.***

def **possui\_ciclo\_hamiltoniano**(V,E):

posible\_starts = **get\_posible\_starts**(E)#vertices com 2 ou mais arestas

for start in posible\_starts:

adjacents = **get\_adjacents**(start, E)

if **can\_cycle**(start, adjacents, V, E):

return True

return False

def **can\_cycle**(start, start\_adjacents, V, E):

edges = **edges\_without\_vertice**(E, start)

vertices = V - [start]

posible\_pairs = [(start\_adjacents[i], start\_adjacents[j]) for i in range(len(start\_adjacents)) for j in range(i+1, len(start\_adjacents))]

for pair in posible\_pairs:

adjacent\_start = pair[0]

adjacent\_end = pair[1]

if **has\_hamilton\_path**(adjacent\_start, adjacent\_end, vertices, edges):

return True

return False

def **has\_hamilton\_path**(current, end, V, E, visited = []):

adjacents = **get\_adjacents**(current, V)

visited.append(current)

ways = adjacents - visited

if ways == [end]:

return True

ways -= [end]

for way in ways:

if **has\_hamilton\_path**(way, end, V, E, visited):

return True

return False

def **edges\_without\_vertice**(E, vertice):

edges = []

for edge in E:

if vertice in edge:

next

edges.append(edge)

return edges

def **get\_posible\_starts**(edges):

posibles = []

memo = {}

for edge in edges:

for vertice in edge:

memo[vertice] = (memo.get(vertice) or 0) + 1

for key, value in memo:

if value > 1:

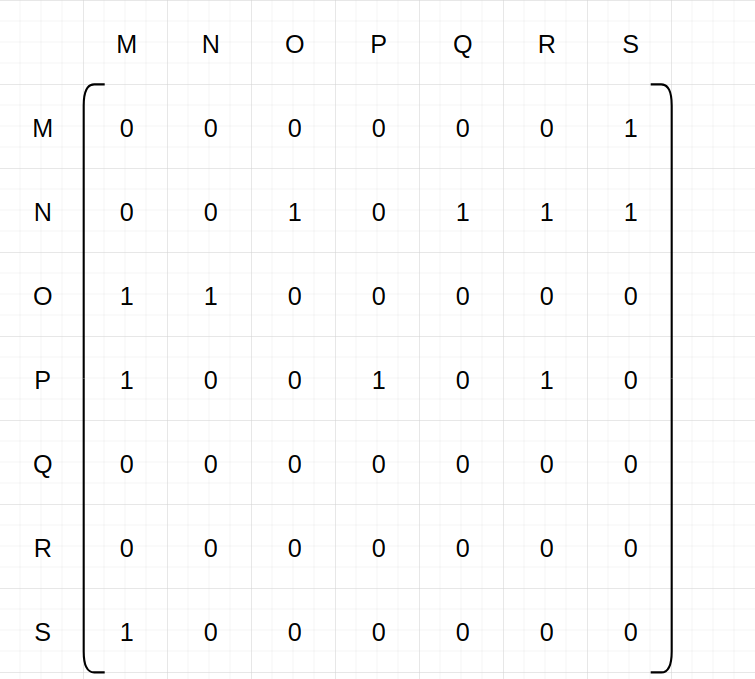
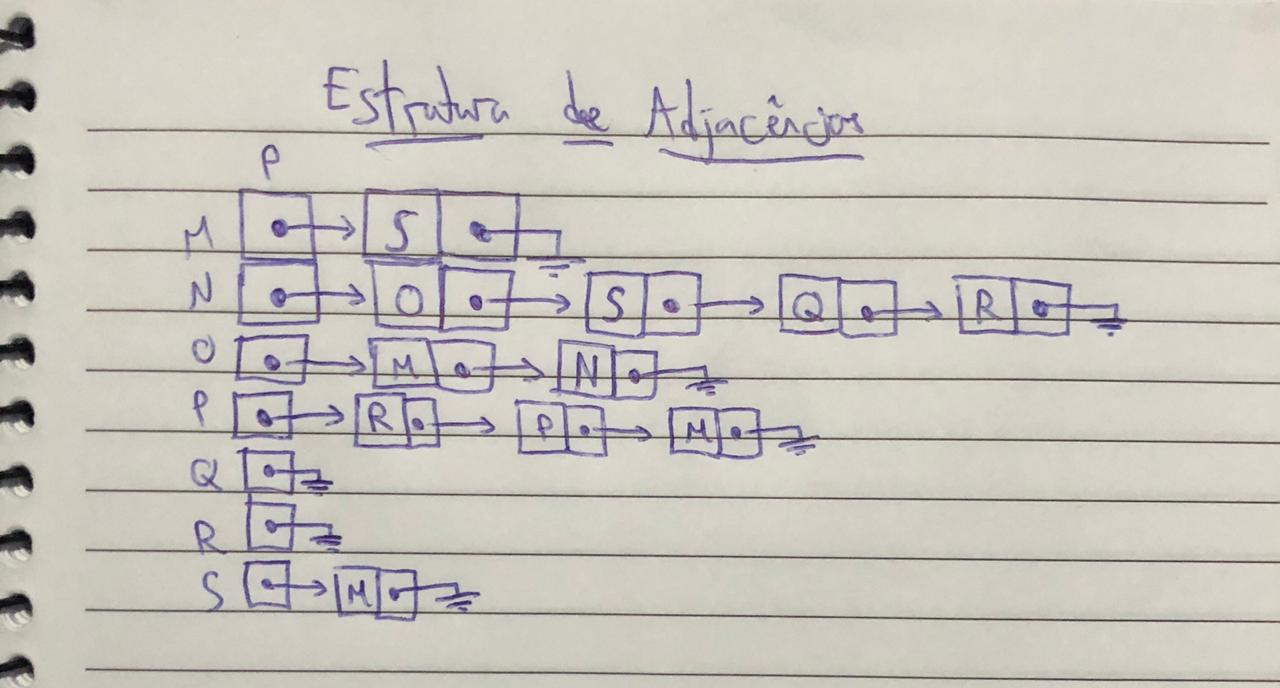
posibles.append(key)

return posibles

***Questão 6)***

1. ***(M,S)***
2. ***Não existe***
3. ***Fracamente Conexo***
4. ***N: Grau de entrada - 1; Grau de saída - 2***

***R: Grau de entrada - 2 ; Grau de saída - 0***

***e)***

***Questão 7)***

1. ***O algoritmo calcula a soma de todos os quadrados de números naturais até n***
2. ***O(n)***
3. ***Sugestão O(1) abaixo.***

def soma\_dos\_quadrados(n):

return n\*\*3/3 + n\*\*2/2 + n/6