图论（1）：基本概念

图的概念、表示方法和遍历：

图论是数学的一个分支，主要研究图的性质以及图之间的关系。在与数据结构和算法相关的内容中，图论涵盖了以下几个方面：

[1] 图的表示：图可以用不同的数据结构来表示，包括邻接矩阵、邻接表等。这些表示方法影响着对图进行操作和算法实现的效率

[2] 图的遍历：图的遍历是指从图中的某个顶点出发，访问图中所有顶点且不重复的过程。常见的图遍历算法包括深度优先搜索dfs和广度优先搜索bfs

[3] 最短路径：最短路径算法用于找出两个顶点之间的最短路径，如Dijkstra算法和Floyd-Warshall算法。这些算法在网络路由、路径规划等领域有广泛的应用。

[4] 最小生成树：最小生成树算法用于在一个连通加权图中找出一个权值最小的生成树，常见算法包括Prim算法和Kruskal算法

[5] 拓扑排序：拓扑排序算法用于对有向无环图进行排序，使得所有的顶点按照一定的顺序排列，并且保证图中的边的方向符合顺序关系。拓扑排序在任务调度、依赖关系分析等领域有重要的应用。

[6] 图的连通性：图的连通性算法用于判断图中的顶点是否连通，以及找出图中的连通分量。这对于网络分析、社交网络分析等具有重要意义。

一、术语和定义

图是更通用的结构。事实上，可以把树看作是一种特殊的图。图一旦有了很好的表示方法，就可以用一些标准的图算法来解决那些看起来非常困难的问题。

（一）图的元素

1.顶点Vertex

顶点又称节点，是图的基础部分。它可以有自己的名字，我们称作“键”。顶点也可以带有附加信息，我们称作“有效载荷”

2.边Edge

边是图的另一个基础部分。两个顶点通过一条边相连，表示它们之间存在关系。边既可以是单向的，也可以是双向的。如果图中所有边都是单向的，我们称之为有向图。

注：最基本的抽象，图（Graph）是由顶点和边组成的。每条边的两端都必须是图的两个顶点（可以是相同的顶点）。记号G(V,E)表示图G的顶点集为V，边集为E。每一条边都是一个二元组（v，w），w，v均属于V。可以向边的二元组中再添加一个元素，就可以表示权重。

3.度Degree

顶点的度是指和该顶点相连的边的条数。特别是对于有向图来说，顶点的出边条数称为该顶点的出度，顶点的入边条数称为该顶点的入度。

4.权值Weight

顶点和边都可以有一定属性，而量化的属性称为权值，顶点的权值和边值分别称为点权和边权。权值可以根据问题的实际背景设定。

5.路径Path

路径是由边连接的顶点组成的序列。路径的正式定义为：w1,w2,...,wn,且有对1<=i<=n-1,有(wi,wi+1)∈E。

无权重路径的长度是路径上的边数，有权重路径的长度是**路径上的边**权重之和

6.环Cycle

环是有向图中的一条起点和终点为同一个顶点的路径。

没有环的图被称为无环图，没有环的有向图被称为有向无环图（DAG，Directed Acyclic Graph）

（二）Practice

1.无向图的度

现有一个共n个顶点、m条边的无向图（假设顶点编号为从 `0` 到 `n-1` ），求每个顶点的度。

输入

第一行两个整数n、m，分别表示顶点数和边数；

接下来m行，每行两个整数u、v，表示一条边的两个端点的编号。数据保证不会有重边。

输出

在一行中输出n个整数，表示编号为从 `0` 到 `n-1` 的顶点的度。整数之间用空格隔开，行末不允许有多余的空格。

代码实现：

```
n, m = map(int, input().split())
degrees = [0] * n
for _ in range(m):
    u, v = map(int, input().split())
    degrees[u] += 1
```

```
        degrees[v] += 1

    print(' '.join(map(str, degrees)))
```

2.有向图的度

现有一个共n个顶点、m条边的有向图（假设顶点编号为从 `0` 到 `n-1` ），求每个顶点的入度和出度。

输入

第一行两个整数n、m，分别表示顶点数和边数；

接下来m行，每行两个整数u、v，表示一条边的两个端点的编号。数据保证不会有重边。

输出

输出行，每行为编号从 `0` 到 `n-1` 的一个顶点的入度和出度，中间用空格隔开。

代码实现：

```
n, m = map(int, input().split())
in_degrees = [0] * n
out_degrees = [0] * n
for _ in range(m):
    u, v = map(int, input().split())
    out_degrees[u] += 1
    in_degrees[v] += 1

for i in range(n):
    print(in_degrees[i], out_degrees[i])
```

二、图的表示方法

（一）图的抽象数据类型的定义

Graph()：新建一个空图

addVertex(vert)：向图中添加一个顶点实例

addEdge(fromVert,toVert)：向图中添加一条有向边，用于连接顶点fromVert和toVert

addEdge(fromVert,toVert,weight)：向图中添加一条带权重weight的有向边，用于连接顶点fromVert和toVert

getVertex(vertKey)：在图中找到名为vertKey的顶点

getVertices()：以列表形式返回图中所有顶点

in：通过vertex in Graph语句，在顶点存在时返回True，否则为False

在不同的表达方式实现图的ADT时，有两种非常著名的图实现，分别是邻接矩阵adjacency Matrix和邻接表adjacency list

（二）图的视线1：邻接矩阵

1.基本定义

要实现图，最简单的方式就是使用二维矩阵。在矩阵实现中，每一行和每一列都表示图中的一个顶点，第v行和第w列交叉的格子中的值表示**从顶点v到顶点w**的边的权重。如果两个顶点被一条边连接起来，就称它们是相邻的。

邻接矩阵的优点是简单。邻接矩阵适用于表示有很多条边的图。然而，对于稀疏矩阵（存储稀疏数据）而言，矩阵并不高效。

2.practice1：sy376-无向图的邻接矩阵

现有一个共n个顶点、m条边的无向图（假设顶点编号为从 `0` 到 `n-1` ），将其按邻接矩阵的方式存储（存在边的位置填充 `1` ，不存在边的位置填充 `0` ），然后输出整个邻接矩阵。

输入

第一行两个整数n、m，分别表示顶点数和边数；

接下来m行，每行两个整数u、v，表示一条边的两个端点的编号。数据保证不会有重边。

输出

输出n行n列，表示邻接矩阵。整数之间用空格隔开，行末不允许有多余的空格。

```
n, m = map(int, input().split())
adjacency_matrix = [[0]*n for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_matrix[u][v] = 1
    adjacency_matrix[v][u] = 1

for row in adjacency_matrix:
    print(' '.join(map(str, row)))
```

3.practice2：sy377-有向图的邻接矩阵

现有一个共n个顶点、m条边的有向图（假设顶点编号为从 `0` 到 `n-1` ），将其按邻接矩阵的方式存储（存在边的位置填充 `1` ，不存在边的位置填充 `0` ），然后输出整个邻接矩阵。

输入

第一行两个整数n、m，分别表示顶点数和边数；

接下来m行，每行两个整数u、v，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

输出n行n列，表示邻接矩阵。整数之间用空格隔开，行末不允许有多余的空格。

```
n, m = map(int, input().split())
adjacency_matrix = [[0]*n for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_matrix[u][v] = 1

for row in adjacency_matrix:
    print(' '.join(map(str, row)))
```

（三）图的实现2：邻接表

1.基本定义

为了实现稀疏连接的图，更高效的方式是使用邻接表。在邻接表实现中，我们为图对象的所有顶点保存一个主列表，同时为每一个顶点对象都维护一个列表，其中记录了与它相连的顶点。在对Vertex类的实现中，我们使用字典：键表示顶点，值是权重

邻接表的优点是能够紧凑地表示稀疏图，其也有助于方便地找到与某一个顶点相邻的其他所有顶点

2.practice1：sy378-无向图的邻接表

现有一个共n个顶点、m条边的无向图（假设顶点编号为从 `0` 到 `n-1` ），将其按邻接表的方式存储，然后输出整个邻接表。

输入

第一行两个整数n、m，分别表示顶点数和边数；

接下来m行，每行两个整数u、v，表示一条边的两个端点的编号。数据保证不会有重边。

输出

输出行，按顺序给出编号从 `0` 到 `n-1` 的顶点的所有出边，每行格式如下：

```
id(k) v_1 v_2 ... v_k
```

```
n, m = map(int, input().split())
adjacency_list = [[] for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_list[u].append(v)
    adjacency_list[v].append(u)
```

```
for i in range(n):
    num = len(adjacency_list[i])
    if num == 0:
        print(f"{i}({num})")
    else:
        print(f"{i}({num})", ' '.join(map(str, adjacency_list[i])))
```

3.practice2：sy379-有向图的邻接表

现有一个共n个顶点、m条边的有向图（假设顶点编号为从 0 到 n-1 ），将其按邻接表的方式存储，然后输出整个邻接表。

输入

第一行两个整数n、m，分别表示顶点数和边数；

接下来m行，每行两个整数u、v，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

输出行，按顺序给出编号从 0 到 n-1 的顶点的所有出边，每行格式如下：

```
id(k) v_1 v_2 ... v_k
```

```
n, m = map(int, input().split())
adjacency_list = [[] for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_list[u].append(v)

for i in range(n):
    num = len(adjacency_list[i])
    if num == 0:
        print(f"{i}({num})")
    else:
        print(f"{i}({num})", ' '.join(map(str, adjacency_list[i])))
```

（四）图的实现3：关联矩阵

1.基本定义

关联矩阵通常用于表示有向图。在关联矩阵中，行代表顶点，列代表边。如果顶点与边相连，则在对应的位置填上1，否则填上0.

2.空间复杂度

关联矩阵的存储空间复杂度为O(V*E)，邻接表的存储空间复杂度为O(V+E)

对于某些图的操作，如查找两个顶点之间是否存在边，邻接表的效率更高；而对于计算图的闭包或者判断两个图是否同构等，关联矩阵可能更方便

注： [1] 图的闭包：对于一个有向图或无向图，将所有顶点对之间的可达路径都加入图中的过程。闭包可以用于分析图中的传递性关系

[2] 图的同构：指两个图具有相同的结构，即顶点和边的连接关系是否一致

3.关联矩阵的实现示例

```
class Graph:
    def __init__(self, vertices, edges):
        self.vertices = vertices
        self.edges = edges
        self.adj_matrix = self.create_adj_matrix()

    def create_adj_matrix(self):
        # Create an empty adjacency matrix
        adj_matrix = [[0] * len(self.edges) for _ in range(len(self.vertices))]

        # Fill adjacency matrix based on edges
        for i, vertex in enumerate(self.vertices):
            for j, edge in enumerate(self.edges):
                if vertex in edge:
                    adj_matrix[i][j] = 1
        return adj_matrix

    def display_adj_matrix(self):
        for row in self.adj_matrix:
            print(row)
```

（五）图的类实现

在python中，通过字典可以轻松实现邻接表。我们创建两个类：Graph类存储包含所有顶点的主列表，Vertex类表示图中的每一个顶点。

1.Vertex类

Vertex使用字典connectedTo来记录与其相连的顶点，以及每一条边的权重。

其构造方法简单地初始化id，以及字典connectedTo。addNeighbor方法添加从一个顶点到另一个顶点的连接。getConnections方法返回邻接表的所有顶点，由ConnectedTo来表示。getWeight方法返回从当前顶点到以参数传入的顶点之间的边的权重。

Vertex类实现：

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}
```

```
    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

2.Graph类

Graph类包含一个将顶点名映射到顶点对象的字典。Graph类也提供了向图中添加和连接不同顶点的方法。getVertices方法返回图中所有顶点的名字。此外，我们还实现了iter方法，从而使遍历图中的所有顶点对象更加方便。总之，iter和contain这两个方法使我们能够根据顶点名或者顶点对象本身遍历图中的所有顶点。

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,weight=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], weight)
        #这里的程序是有向图

    def getVertices(self):
        return self.vertList.keys()
```

```
    def __iter__(self):
        return iter(self.vertList.values())
```

注：机考中，可以直接使用二维列表或者字典来显示邻接表

3.例题：OJ19943-图的拉普拉斯矩阵

（1）问题描述

在图论中，度数矩阵是一个对角矩阵 ，其中包含的信息为的每一个顶点的度数，也就是说，每个顶点相邻的边数。邻接矩阵是图的一种常用存储方式。如果一个图一共有编号为0,1,2，…n-1的n个节点，那么邻接矩阵A的大小为n*n，对其中任一元素Aij，如果节点i，j直接有边，那么Aij=1；否则Aij=0。

将度数矩阵与邻接矩阵逐位相减，可以求得图的拉普拉斯矩阵。

现给出一个图中的所有边的信息，需要你输出该图的拉普拉斯矩阵。

输入

第一行2个整数，代表该图的顶点数n和边数m。

接下m行，每行为空格分隔的2个整数a和b，代表顶点a和顶点b之间有一条无向边相连，a和b均为大小范围在0到n-1之间的整数。输入保证每条无向边仅出现一次（如1 2和2 1是同一条边，并不会在数据中同时出现）。

输出

共n行，每行为以空格分隔的n个整数，代表该图的拉普拉斯矩阵。

（2）代码实现

```
class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self, nbr):
        return self.connectedTo[nbr]

class Graph:
```

```
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self, n):
        return n in self.vertList

    def addEdge(self, f, t, weight=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], weight)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())

def constructLaplacianMatrix(n, edges):
    graph = Graph()
    for i in range(n):  # 添加顶点
        graph.addVertex(i)

    for edge in edges:  # 添加边
        a, b = edge
        graph.addEdge(a, b)
        graph.addEdge(b, a)

    laplacianMatrix = []        # 构建拉普拉斯矩阵
    for vertex in graph:
        row = [0] * n
        row[vertex.getId()] = len(vertex.getConnections())
        for neighbor in vertex.getConnections():
            row[neighbor.getId()] = -1
        laplacianMatrix.append(row)

    return laplacianMatrix


n, m = map(int, input().split())        # 解析输入
edges = []
for i in range(m):
```

```
    a, b = map(int, input().split())
    edges.append((a, b))

laplacianMatrix = constructLaplacianMatrix(n, edges)    # 构建拉普拉斯矩阵

for row in laplacianMatrix:     # 输出结果
    print(' '.join(map(str, row)))
```