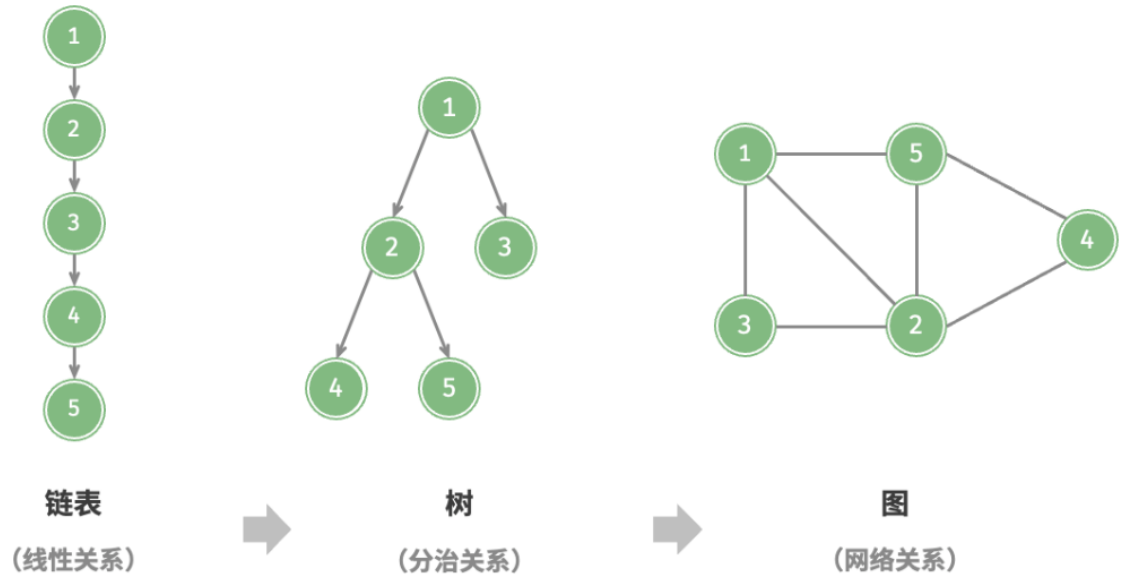
图论

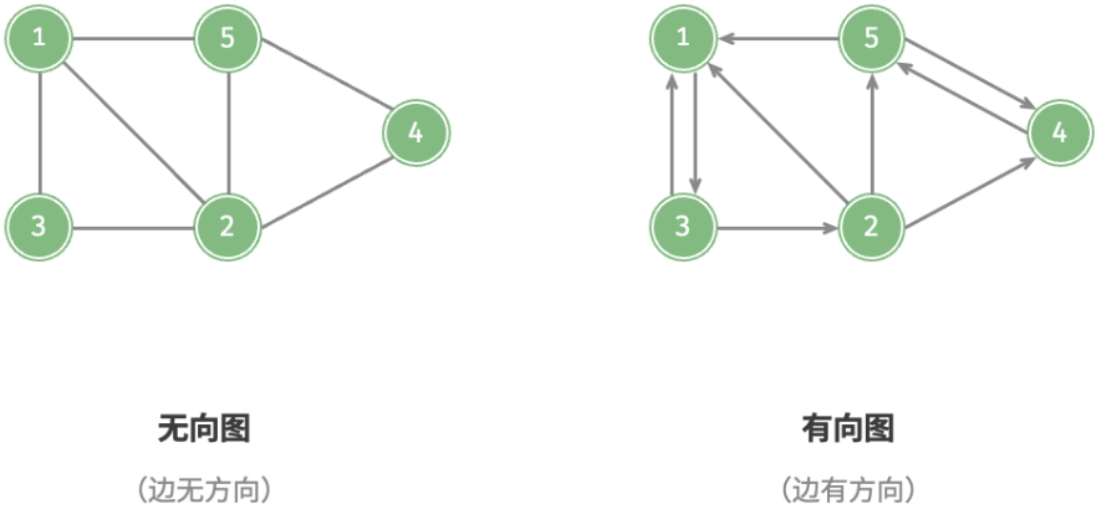
图论基本概念



图的逻辑自由度最高，因而也最为复杂

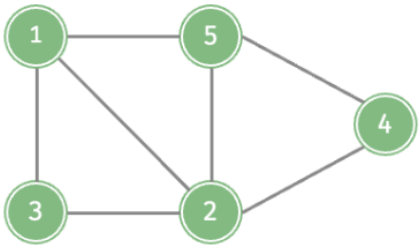
- 图（graph）：点用边连接起来就是图。图是一种数据结构
- G = (V , E) , V 是一个非空有限集合，代表顶点(节点)， E 代表边的集合

图的定义和概念



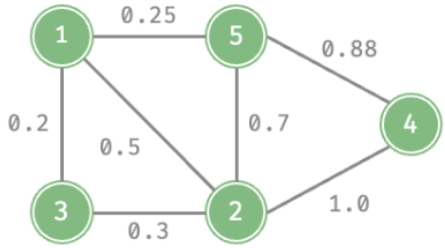
- 有向图：图的边有方向，只能按箭头方向从一个点到另一个点
- 无向图：图的边没有方向，可以双向
- 结点的度：无向图中与结点相连的边的数目，称为结点的度

- 结点的入度：有向图中，以这个结点为终点的有向边的数目
- 结点的出度：有向图中，以这个结点为起点的有向边的数目
- 权值：边的“费用”，可以形象理解为边的长度



无权图

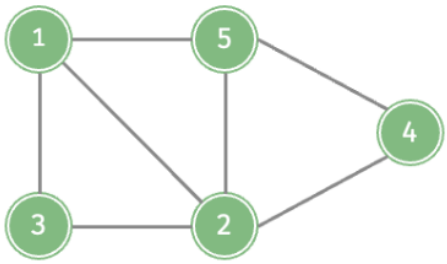
（所有边等价）



有权图

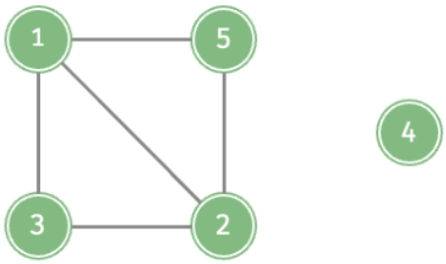
（边具有权重属性）

- 回路 ： 起点和终点相同的路径，称为回路，或“环”
- 完全图：一个 **n** 阶的完全无向图含有 **n*(n-1)/2** 条边，一个 **n** 阶的完全有向图含有 **n*(n-1)** 条边
- 稠密图：一个边数接近完全图的图
- 稀疏图：一个边数远远少于完全图的图
- **连通图和非连通图**：根据所有顶点是否连通，可分为连通图(connected graph)和非连通图(disconnected graph)



连通图

（所有顶点皆可达）

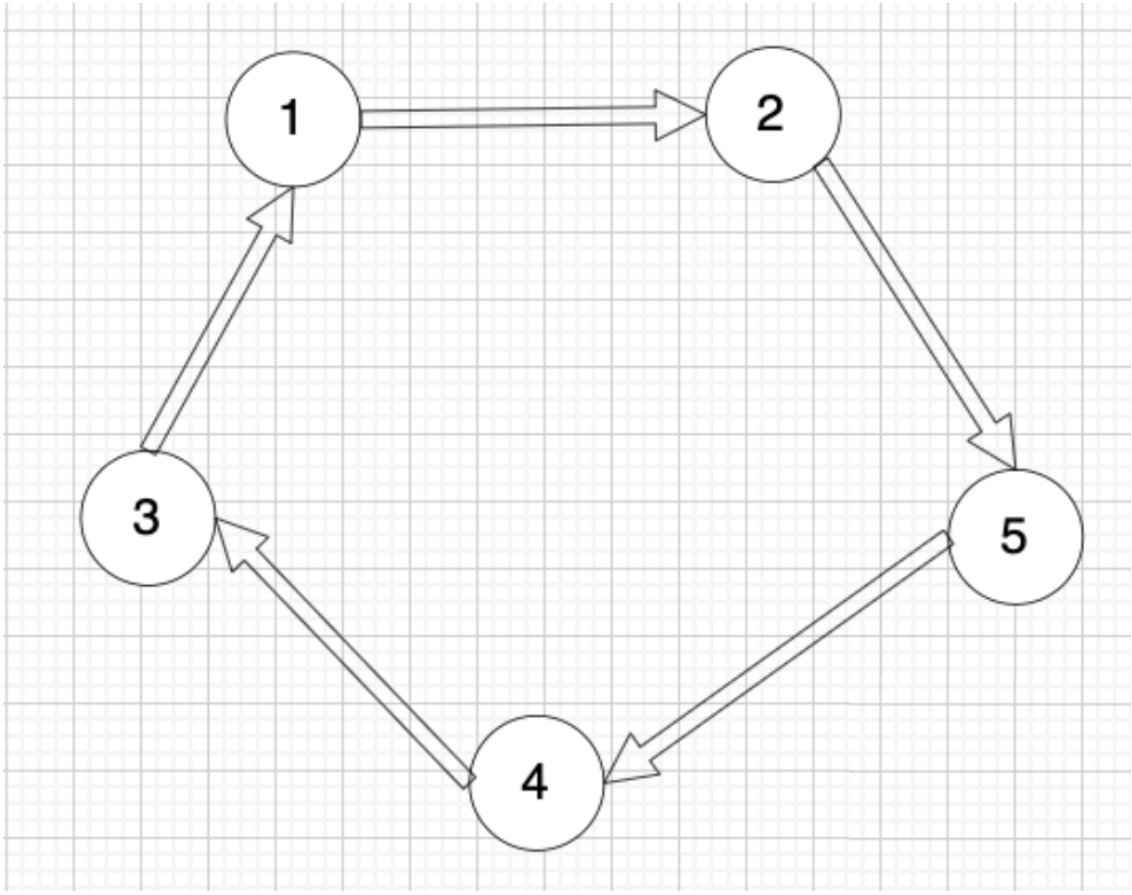


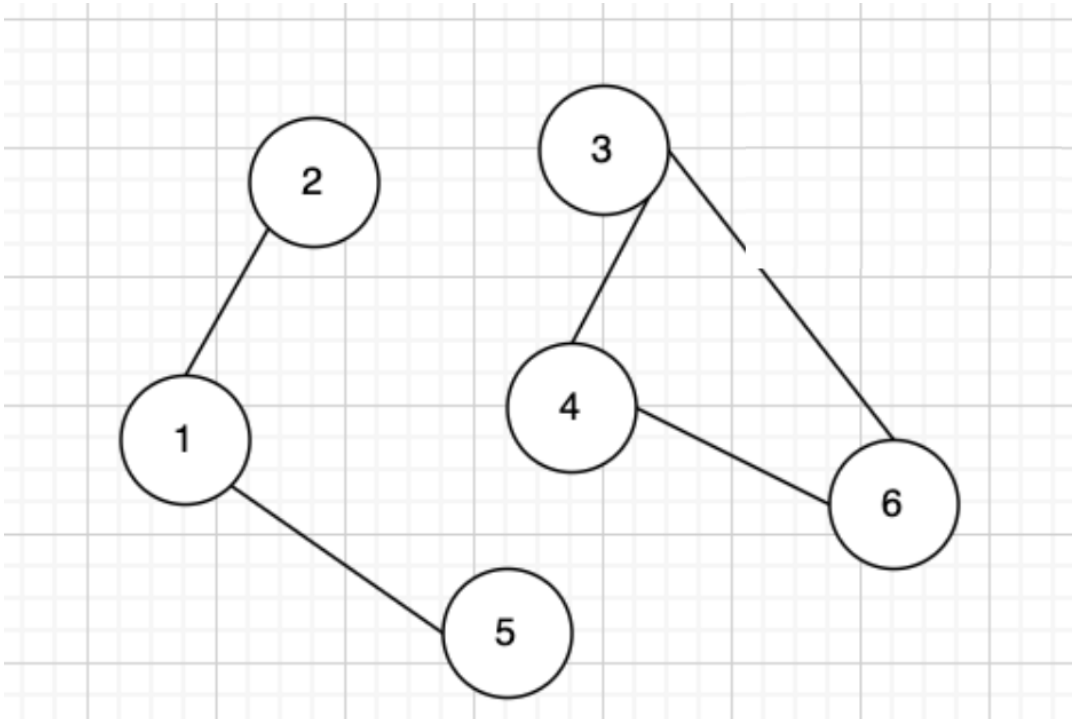
非连通图

（存在顶点不可达）

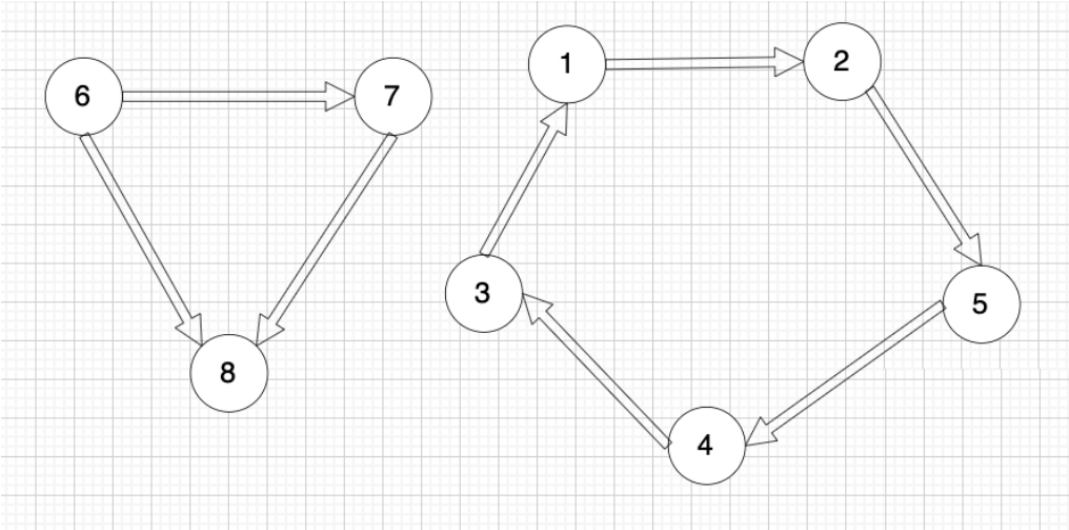
  - 连通图：从某个顶点出发，可以到达其余任意顶点
  - 非连通图：从某个顶点出发，至少有一个顶点无法到达

- **强连通图**：在 **有向图** 中 **任何两个节点是可以相互到达**



- 连通分量： 在 **无向图** 中的极大连通子图称之为该图的一个 **连通分量**
  - 必须是极大连通子图才能是连通分量，所以 节点 3 、4 、6构成的才是连通分量
  - 

- **强连通分量** ： 在 **有向图** 中极大连通子图称之为该图的强连通分量



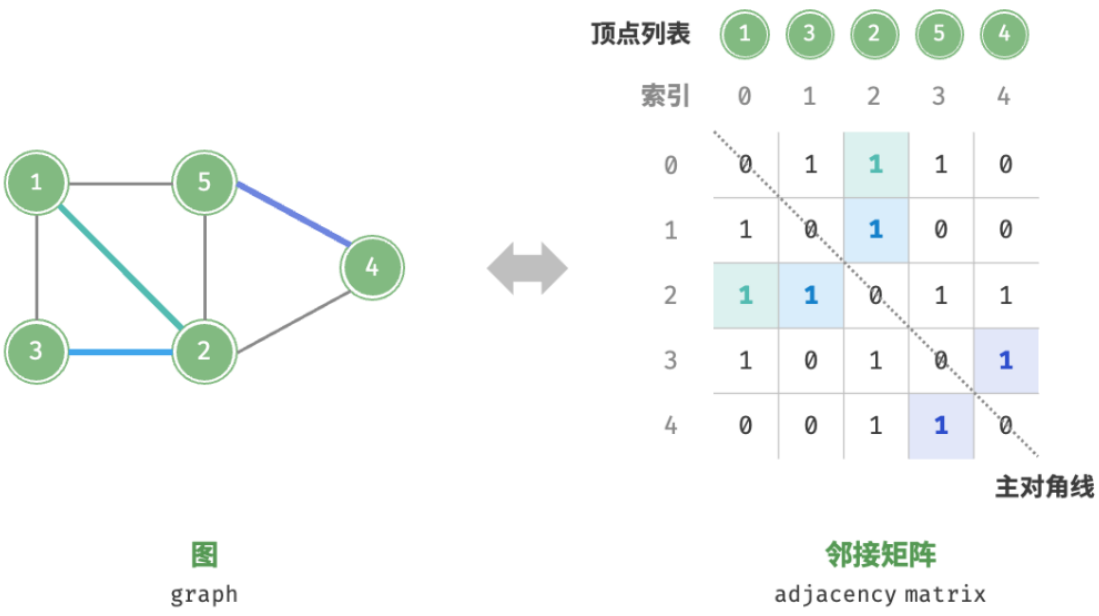
- 节点 1、2 、3、4、5 构成的子图是强连通分量，因为这是强连通图，也是极大图

图的存储

- 练习：图的存储

邻接矩阵

- 设图的顶点数量为 **n** ，邻接矩阵使用一个 **n*n** 大小的矩阵来表示图，每一个行(列)代表一个顶点，矩阵元素代表边，用 **1** 或 **0** 表示两个顶点之间是否存在边
- ```
  int e[N][N];
  // 输入一条边
  void add(int u, int v) {
      e[u][v] = 1;
      e[v][u] = 1;
  }
  ```
-
- 设邻接矩阵为 **G**、顶点列表为 **V** ， 那么矩阵元素 **G[i][j]=1** 表示顶点 **i** 到顶点 **j** 之间存在边，反之 **G[i][j]=0** 表示顶点 **i** 到顶点 **j** 之间存在边


  ```

- 邻接矩阵的特性
  - 在简单图中，顶点不能与自身相连，此时矩阵主对角线元素没有意义
  - 对于无向图中，两个方向的边等价，此时邻接矩阵关于主对角线对称
  - 将邻接矩阵的元素从 **1** 和 **0** 替换为权重，则可表示为有权图
- 复杂度
  - 查询是否存在某条边：**O(1)**
  - 遍历一个点的所有出边：**O(n)**
  - 遍历整张图：$O(n^2)$
  - 空间复杂度：$O(n^2)$
- 技巧：
  - int 数组： **memset(g , 0x7f , sizeof(g))** , 可全部初始为一个很大的数(略小于 **0x7fffffff**)
  - int 数组： **memset(g , 0 , sizeof(g))** , 可全部清为 0
  - int 数组： **memset(g , 0xaf , sizeof(g))** , 全部初始化为一个很小的数字
  - double 数组： **memset(g , 127 , sizeof(g))** , 可全部初始为一个很大的数($1.38 * 10^{306}$)
  - double 数组： **memset(g , 0 , sizeof(g))** , 可全部清为 0
- 练习题： 图的存储

邻接表

- 使用一个支持动态增加元素的数据结构构成的数组，如 **vector<int> e[n+1]**
- ```
  vector<int> e[N];
  // 输入一条边
  void add(int u, int v) {
      e[u].push_back(v);
      e[v].push_back(u);
  }
  ```
-
- 其中 **e[u]** 存储的是点 **u** 的所有出边的相关信息(终点 、 边权等)

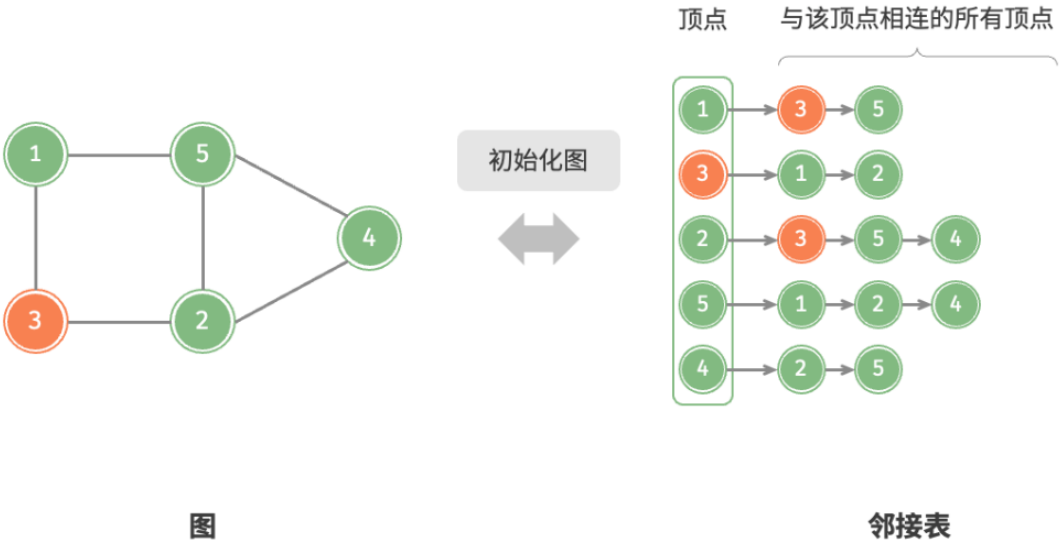


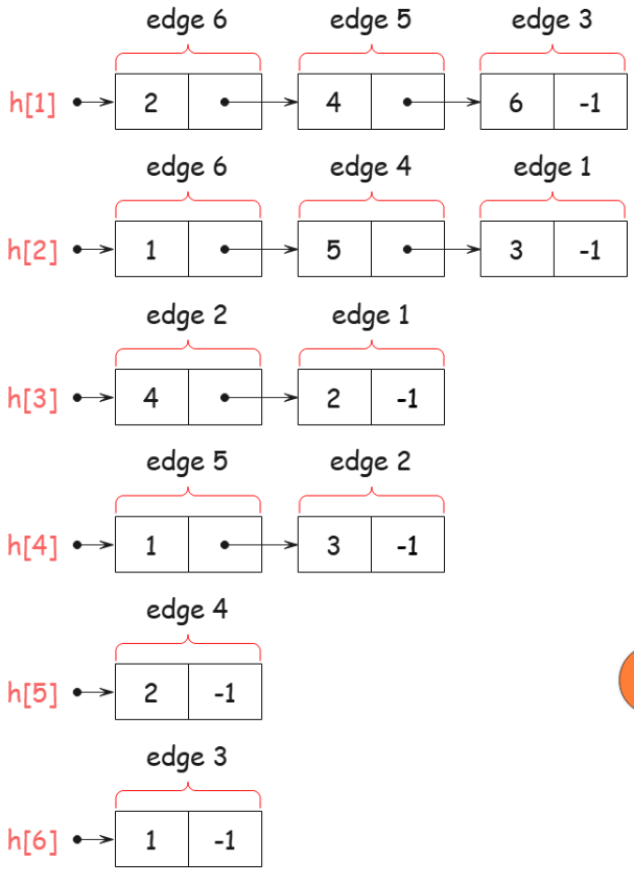
图 **邻接表**

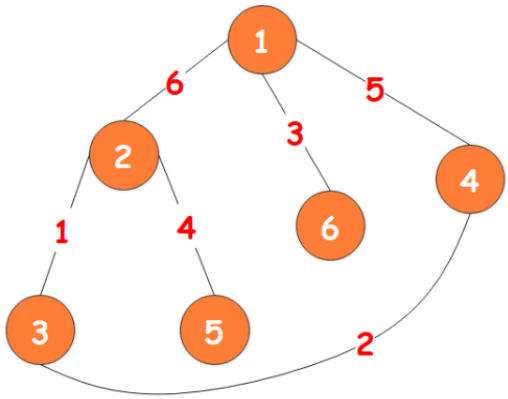
- 复杂度
  - 查询是否存在 **u** 到 **v** 的边： $O(点u的出度)$
  - 遍历点 **u** 的所有出边： $O(点u的出度)$
  ```

- 遍历整张图：$O(n+m)$
- 空间复杂度：$O(m)$

链式前向星

- 本质上是用链表实现的邻接表
- ```
  // head[u] 和 cnt 的初始值都为 -1
  void add(int u, int v) {
    nxt[++cnt] = head[u];  // 当前边的后继
    head[u] = cnt;         // 起点 u 的第一条边
    to[cnt] = v;           // 当前边的终点
  }

  // 遍历 u 的出边
  for (int i = head[u]; ~i; i = nxt[i]) {  // ~i 表示 i != -1
    int v = to[i];
  }
  ```
- 



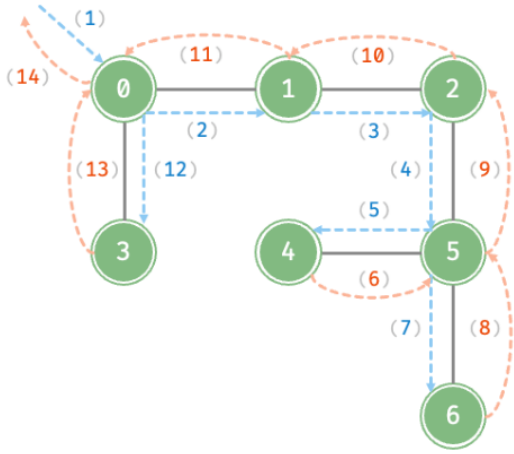
- 复杂度
  - 查询是否存在 u 到 v 的边：$O(u的出度)$
  - 遍历点 u 的所有出边：$O(u的出度)$
  - 遍历整张图：$O(n+m)$
  - 空间复杂度：$O(m)$
  ```

图的遍历

- 练习：图的遍历

深度优先遍历

- 深度优先遍历是一种优先走到底，无路可走再回头的遍历方式
- 从左上角顶点出发，访问当前顶点的某个邻接顶点，直到走到尽头时返回，再继续走到尽头并返回，以此类推，直到所有顶点遍历完成



图的深度优先遍历（DFS）

以顶点 0 为起始，
走到头才返回，再走到头才返回，
以此类推 … 直至完成遍历

遍历序列为

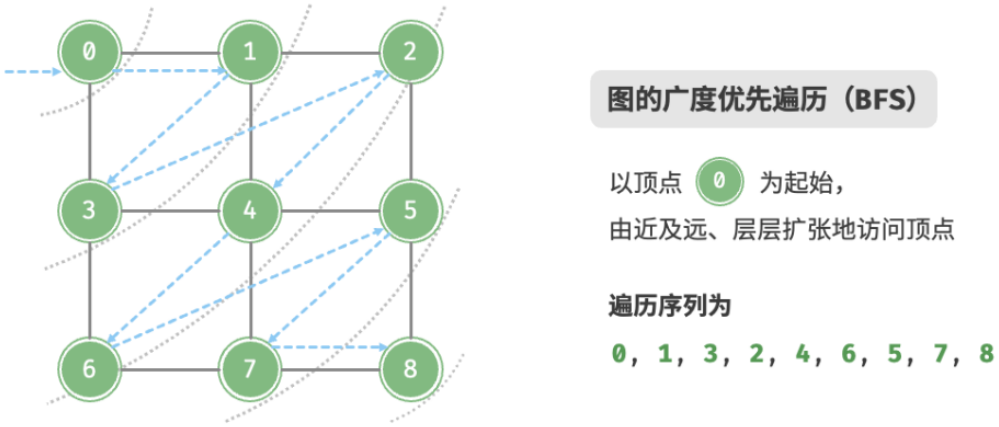
0, 1, 2, 5, 4, 6, 3

- 复杂度分析：
  - 时间复杂度分析：所有顶点都会被访问一次，使用 $O(V)$ 的时间
    - 所有边都会被访问两次： 使用 $O(2|E|)$ 的时间
    - 总的时间复杂度为： $O(V+E)$
  - 空间复杂度：$O(V)$ 空间

广度优先遍历

- 广度优先遍历是一种由近及远的遍历方式，从某个节点出发，始终优先访问距离最近的顶点，并一层层向外扩张

- 如图：从左上角顶点出发，首先遍历该顶点的所有邻接顶点，然后遍历下一个顶点的所有邻接顶点，以此类推，直到所有顶点访问完毕



- 算法实现
  - 将遍历起点 start 加入队列，并开启循环
  - 在循环的每轮迭代中，弹出队首顶点并记录访问，然后将该顶点的所有邻接点加入到队列尾部
  - 循环上一个步骤， 直到所有顶点被访问完毕后结束
  - 为了防止重复遍历顶点，需要借助一个 哈希 集合来记录哪些节点已被访问

拓扑排序

- 给定一张有向无环图，若一个由图中所有点构成的序列 A 满足：对于图中的每条边 (x, y) ，x 在 A 中都出现在 y 之前，则称 A 是该有向无环图顶点的一个拓扑序
- 拓扑排序：不断选择图中入度为 0 的节点 x，然后把 x 连向的点的入度减 1
  1. 建立空的拓扑序列 A
  2. 预处理出所有点的入度 $deg[i]$，起初把所有入度为 0 的点入队
  3. 取出队头节点 x，把 x 加入拓扑序列 A 的末尾
  4. 对于从 x 出发的每条边 (x, y)，把 $deg[y]$ 减 1 。若被减为 0，则把 y 入队
  5. 重复第 3~ 4 步直到队列为空，此时 A 即为所求
- **判断是否存在环**：如果序列 A 的长度小于图中点的数量，则说明某些节点未被遍历，进而说明图中存在环

拓扑排序代码

-
```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 10;
const int M = 1e5 + 10;
int n, m;
int head[N], ver[M], nxt[M], tot, deg[N];
int ans[N], cnt;
void add(int x, int y) {
    tot++;
    ver[tot] = y;
    nxt[tot] = head[x];
```

```
    head[x] = tot;
    deg[y]++; // 入度增加
}

void topsort() {
    queue<int> q;
    for (int i = 1; i <= n; i++)
        if (deg[i] == 0)
            q.push(i);
    while (q.size()) {
        int x = q.front();
        q.pop();
        ans[++cnt] = x;
        for (int i = head[x] ; i ; i = nxt[i]) {
            int y = ver[i];
            if (--deg[y] == 0) q.push(y);
        }
    }
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        add(x, y);
    }
    topsort();
    for (int i = 1; i <= n; i++)
        cout << ans[i] << " ";
    return 0;
}
```