# 图遍历与拓扑排序

## 三种存图、DFS/BFS 遍历判环、Kahn 拓扑排序与 DAG DP

HGNU-ACM 实验室算法组

配套：邻接表与链式前向星 + 三色判环 + Kahn 入度剥洋葱 + 《查找文献》《家谱树》《奖金》






# 本章目录

图的表示与三种存图方式

图的 DFS 与 BFS 遍历及判环

有向无环图 DAG 与 Kahn 拓扑排序

前三节对比三种存图方式，学习 DFS/BFS 遍历与三色判环，并用 Kahn 算法解决有向无环图（DAG）的先后排队问题。

# 本章目录（续）

拓扑序判环与 DAG 动态规划

三大经典真题精讲

常见错误、检查清单与练习安排

后三节深入拓扑序判环与 DAG 动态规划，精讲《查找文献》《家谱树》《奖金》三道真题，并总结高频易错清单。

# 承前启后：从“线性序列/DP 表”走进“网状关系图”

- 在刚结束的《05_ 动态规划》篇里，状态几乎都排在一条线（$1 \ldots n$）或二维格子上，谁在谁前面一目了然。
- 但在现实世界里，很多关系是网状的：
  - 微信好友关系（双向相连）；
  - 高铁城市航线（单向或双向，带里程）；
  - 大学课程先修依赖（必须先学 A，才能学 B）。
- 把“对象”画成圆圈，把“关系”画成连线，就得到了计算机科学中最重要的模型——图（**Graph**）。

## 本页要点

图论第一步：先把题目里的人物、城市、课程抽象成“点”，把关系抽象成“边”！

# 新术语大白话：点、边、有向/无向、度

- 顶点（点）与边：圆圈叫顶点（常用 $n$ 表示点数），连线叫边（常用 $m$ 表示边数）。
- 有向边与无向边：带箭头的边 $u \to v$ 叫有向边（只能从 $u$ 到 $v$）；不带箭头的叫无向边（等价于两条反向的有向边）。
- 入度与出度：在有向图里，入度（`indeg`）是箭头指向这个点的数量；出度是从这个点出发的箭头数量。
- 稀疏图与稠密图：边数 $m$ 远小于 $n^2$（比如 $n = 10^5, m = 2 \times 10^5$）叫稀疏图；几乎每对点都有边叫稠密图。

## 本页要点

做图论题先看 $n$ 和 $m$ 的范围：绝大多数竞赛题都是 $n, m \sim 10^5$ 的稀疏图！

# 暴力存图（邻接矩阵）为什么在 $n = 10^5$ 时会 MLE？

- 最直观的存图方式是开一个二维数组 `int g[N][N]`，用 `g[u][v] = 1` 表示 $u$ 到 $v$ 有边，这叫邻接矩阵。
- 来算一笔内存账：当 $n = 10^5$ 时，矩阵有 $10^5 \times 10^5 = 10^{10}$ 个 `int`。
- 每个 `int` 占 $4$ 字节，总内存 $= 4 \times 10^{10}$ B $\approx$ **40** GB！
- 竞赛评测机一般只给 $128$ MB $\sim 512$ MB 内存，二维数组直接爆内存 **MLE**（或编译报错）！

## 本页要点

邻接矩阵空间复杂度是 $O(n^2)$，只有 $n \le 2000$ 且需要 $O(1)$ 查两点是否有边时才用！

# 三种存图方式全景对比表

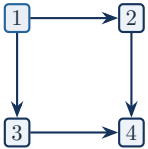
| 存图方式 | 空间复杂度 | 加边 | 遍历 $u$ 的出边 | 适用场景 |
|---|---|---|---|---|
| 邻接矩阵 `g[u][v]` | $O(n^2)$ | $O(1)$ | $O(n)$ | $n \le 10^3$ 稠密图、Floyd |
| `vector` 邻接表 | $O(n+m)$ | $O(1)$ | $O(\deg(u))$ | **90%** 竞赛题首选，支持边排序 |
| 链式前向星 | $O(n+m)$ | $O(1)$ | $O(\deg(u))$ | 极卡常或网络流（需存反向边下标） |

- `vector<vector<int>> g(n + 1)` 只给真正存在的 $m$ 条边分配空间，完全不存“没边”的 $0$。
- 链式前向星用三个一维数组 `head[]`，`to[]`，`nxt[]` 模拟链表，没有动态扩容开销，速度最快。

## 本页要点

平时刷题首选 `vector` 邻接表（好写又不容易越界）；极度卡时间或写网络流再用链式前向星！

# TikZ 图解：同一个 4 点 4 边图在三种存图结构中的形态




邻接表（vect
g[1]: [2, 3
g[2]: [4]
g[3]: [4]
g[4]: []

链式前向星（头插法）
head[1]=2 → 边 2(to=3,nxt=1) → 边 1(to=2,nxt=0)
head[2]=3 → 边 3(to=4,nxt=0)
head[3]=4 → 边 4(to=4,nxt=0)

## 本页要点

链式前向星每次加边都是“插到表头”，所以遍历出边的顺序和读入顺序刚好相反！

# 链式前向星手算跟踪表（4 点 4 边头插法）

**依次加入有向边：$1\to2$、$1\to3$、$2\to4$、$3\to4$，初始 `head = [0,0,0,0,0]`，`cnt = 0`**

| 加边操作 | ++cnt | to[cnt] | nxt[cnt] = head[u] | head[u] = cnt | 当前 head[1..4] |
|---|---|---|---|---|---|
| 加 $1\to2$ | 1 | 2 | 0 | head[1]=1 | [1, 0, 0, 0] |
| 加 $1\to3$ | 2 | 3 | 1 | head[1]=2 | [2, 0, 0, 0] |
| 加 $2\to4$ | 3 | 4 | 0 | head[2]=3 | [2, 3, 0, 0] |
| 加 $3\to4$ | 4 | 4 | 0 | head[3]=4 | [2, 3, 4, 0] |
| 遍历 1 | 从 i=head[1]=2 出发得 to[2]=3，再跳 i=nxt[2]=1 得 to[1]=2，再跳 i=nxt[1]=0 停止！ | | | | |

## 本页要点

用 0 作为链表结尾哨兵，边编号 cnt 从 1 开始（++cnt），遍历写成 `for(int i=head[u]; i; i=nxt[i])`！

# 模板代码（1/2）：vector 邻接表加边与遍历

```
int n, m; cin >> n >> m;
vector<vector<int>> g(n + 1);                    // 下标 1..n，每行是一个动态数组
for (int i = 0, u, v; i < m; ++i) {
    cin >> u >> v;
    g[u].push_back(v);                           // 有向图加单向边 u -> v
    // g[v].push_back(u);                        // 若是无向图，取消此行注释连反
    向边
}
for (int u = 1; u <= n; ++u) {
    for (int v : g[u]) {                         // 依次访问 u 的所有邻居 v
        // 处理边 u -> v
    }
}
```

## 本页要点

带边权时改成 vector<vector<pair<int, int>>>（存 {v, w}）即可！

# 模板代码（2/2）：链式前向星加边与遍历

```cpp
const int MAXN = 1e5 + 5, MAXM = 2e5 + 5;          // 无向图边数组必须开 2 倍！
int head[MAXN], to[MAXM], nxt[MAXM], cnt = 0;      // cnt 从 0 起，++cnt 后第一条
    边为 1
void add_edge(int u, int v) {
    to[++cnt] = v;                                  // 新边指向 v
    nxt[cnt] = head[u];                             // 新边的下一条指向原来的表头
    head[u] = cnt;                                  // 更新表头为当前新边
}
// 遍历点 u 的所有出边：
for (int i = head[u]; i; i = nxt[i]) {              // i == 0 表示链表结束
    int v = to[i];
}
```

## 本页要点

多组测试数据用链式前向星时，务必写 `fill(head, head + n + 1, 0); cnt = 0;` 清空！

# 上机验证：3 组测试数据与反事实追问

| 测试场景 | 输入（$n, m$ 与边） | 点 1 的出边邻接表 | 考察目的 |
|---|---|---|---|
| 普通有向图 | `3 2 / 1 2, 1 3` | `2, 3` | 基本加边正确性 |
| 孤立无边图 | `3 0` | 空 | $m = 0$ 时不越界 |
| 重边与自环 | `2 2 / 1 1, 1 1` | `1, 1` | 邻接表自动保留全部边 |

## 反事实追问

- 问：写无向图时用链式前向星，如果题面说 $m \le 10^5$，你把 `MAXM` 也只开成 $10^5 + 5$ 会怎样？
- 答：无向图每条边要调用两次 `add_edge`，总边数达到 $2 \times 10^5$，写入第 $100006$ 条边时数组越界发生 **RE**（或踩坏相邻全局变量导致莫名 **WA**）！

## 本页要点

无向图的边数组大小永远开 $2m + 5$，这是图论最常见的 RE 来源之一！

# 本节易错：无向图漏连反向边与重边/自环处理

## 常见错误写法

- 读无向图只写 `g[u].push_back(v)`，忘了 `g[v].push_back(u)` → **WA**；
- 链式前向星多组数据只清 `cnt = 0` 没清 `head[]` → 访问上组旧边 **RE/WA**；
- 邻接表下标 $1 \ldots n$ 却只开了 `g(n)` → 访问 `g[n]` 越界 **RE**。

## 正确写法

- 读题先圈出“有向图”还是“无向图”；
- 只要点编号从 1 到 $n$，容器大小一律开 `n + 1`；
- 多测清空必清 `head[1..n]` 与 `cnt = 0`。

## 本页要点

点开 `n+1`、无向边开 `2m+5`、多测必清 `head`，三条铁律保你存图零 RE！

# 两种逛图思路：一条路走到黑（DFS）vs 水波纹扩散（BFS）

- 存好图之后，怎样把图上能走到的所有点不重不漏地走一遍？
- 深度优先搜索（**DFS**）：像走迷宫“不撞南墙不回头”，沿着一条路一直往深处走，无路可走再退回上一个路口换条路（用递归或栈实现）。
- 广度优先搜索（**BFS**）：像往水池里扔一颗石子，水波纹一圈一圈往外扩散，先访问距离起点 1 步的点，再访问距离 2 步的点（用队列 queue 实现）。
- 两者都能访问到起点所在的整个连通部分，时间复杂度都是 $O(n+m)$。

## 本页要点

需要“按层求最少步数”用 BFS；需要“连通块计数、判环、树形统计”优先用好写的 DFS！

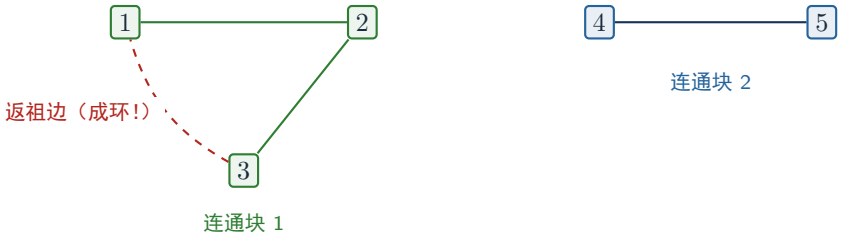
# 为什么遍历图必须加 vis[] 标记？

- 在遍历一棵树时，只要不走回父节点 fa 就绝不会重复；但在图里可能有环（例如 $1 \to 2 \to 3 \to 1$）！
- 如果不记“哪些点已经去过”，程序就会沿着 $1 \to 2 \to 3 \to 1 \to 2 \ldots$ 无限兜圈，直到超时 **TLE** 或递归爆栈 **RE**！
- 于是需要一个布尔数组 vis[u]：
  - **DFS**：刚进入 dfs(u) 就标记 vis[u] = 1，只往 !vis[v] 的邻居走；
  - **BFS**：必须在入队的那一刻就标记 vis[v] = 1，绝不能等出队再标（否则同一个点会被多个邻居重复塞进队列，爆内存/超时）！

## 本页要点

BFS 一定要“入队即标记 vis[v]=1”，千万不要等出队才标！

# TikZ 图解：无向图连通块计数与 DFS 树边/返祖边




- 从 $1$ 开始 DFS 只能染完 $\{1,2,3\}$；主循环继续扫到未访问的 $4$，再次启动 DFS 染完 $\{4,5\}$。
- 每次从主循环里启动一次新的 DFS，就说明发现了一个全新的连通块！

## 本页要点

求无向图连通块个数：主循环 `for (i=1..n) if (!vis[i]) { ++blocks; dfs(i); }`！

# 手算跟踪表：5 点 2 连通块无向图 DFS 过程

**无向图边集：$(1,2),(2,3),(3,1),(4,5)$，初始 `vis = [0,0,0,0,0]`，`blocks = 0`**

| 主循环扫描 $i$ | 当时 `vis[i]` | 触发动作 | 结束后 `vis[1..5]` |
|---|---|---|---|
| $i=1$ | 0 | `++blocks`（$=1$），从 1 DFS 走 $1\to2\to3$ | `[1, 1, 1, 0, 0]` |
| $i=2$ | 1 | 已访问，跳过 | `[1, 1, 1, 0, 0]` |
| $i=3$ | 1 | 已访问，跳过 | `[1, 1, 1, 0, 0]` |
| $i=4$ | 0 | `++blocks`（$=2$），从 4 DFS 走 $4\to5$ | `[1, 1, 1, 1, 1]` |
| $i=5$ | 1 | 已访问，跳过 | `[1, 1, 1, 1, 1]` |
| 终止 | $i=6>n$，循环结束！共找到 `blocks = 2` 个连通块。 | | |

## 本页要点

连通块计数本质上就是统计“一共发动了几次 DFS”！

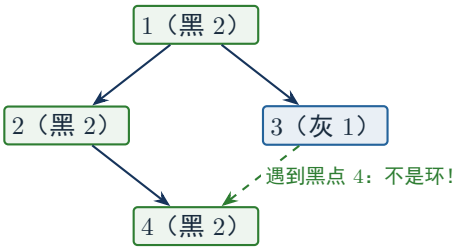
# 无向图 DFS 判环 vs 有向图三色标记（0/1/2）判环

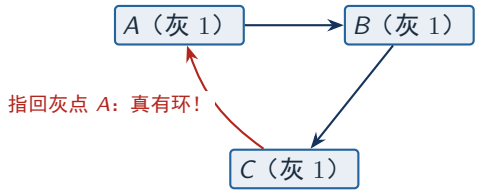
- 无向图判环：在 `dfs(u, fa)` 中，如果遇到一个邻居 $v$ 已经 `vis[v] == 1` 且 $v \neq$ fa（不是刚走过来的那条父边），就说明绕回来了——一定有环！
- 有向图判环陷阱：有向图不能只用 `0/1` 两种状态！因为 $1 \to 2 \to 4$ 和 $1 \to 3 \to 4$ 两条路在 $4$ 汇合时，虽然 $4$ 已经被访问过，但这只是交叉边，并不是环！
- 有向图三色标记法：
  1. `color[u] = 0`（白点）：还没访问过；
  2. `color[u] = 1`（灰点）：正在当前递归栈中（还没退出来）；
  3. `color[u] = 2`（黑点）：它和它的后代已经全部搜完并退栈了。

## 本页要点

在有向图里，只有遇到灰点（`color[v]==1`，还在当前递归路径上），才算真正绕回成环！

# TikZ 图解：有向图“交叉边”不是环，只有遇“灰点”才是环！




## 本页要点

看到黑点说明那条岔路早就安全结束了；只有指回灰点，才说明自己踩到了自己的尾巴！

# 正确性论证：为什么三色标记能不漏不误地判有向环？

| 性质 | 论证说明 |
| --- | --- |
| 灰点定义 | 任意时刻，所有 `color == 1` 的点恰好组成当前 DFS 递归栈上的一条有向路径。 |
| 遇灰点必有环 | 若当前点 $u$ 有一条出边指向灰点 $v$，则递归栈上原本就有 $v \rightsquigarrow u$ 的路径，加上 $u \to v$ 立即闭合为一个有向环。 |
| 有环必遇灰点 | 设环上第一个被访问到的点为 $v$；在 $v$ 退栈变黑之前，由于环上所有点都从 $v$ 可达，必有一个环上点顺着环边指回仍为灰色的 $v$！ |

## 本页要点

“第一个进环的点变黑前，环上的最后一条边必然指回它”，这就是三色判环的严密保证！

# 模板代码（1/2）：无向图 DFS 连通块计数

```cpp
void dfs(int u, const vector<vector<int>>& g, vector<int>& vis) {
    vis[u] = 1;                                          // 进门先标记已访问
    for (int v : g[u]) if (!vis[v]) dfs(v, g, vis); // 没去过的邻居才递归
}
// 主函数里统计连通块个数：
int blocks = 0;
for (int i = 1; i <= n; ++i) {
    if (!vis[i]) { ++blocks; dfs(i, g, vis); }      // 发现新连通块并染黑
}
```

## 本页要点

无向图 DFS 连通块计数一共只有 10 来行，时间复杂度严格 $O(n+m)$！

# 模板代码（2/2）：有向图三色 DFS 判环

```cpp
bool has_cycle(int u, const vector<vector<int>>& g, vector<int>& color) {
    color[u] = 1;                                          // 1 = 灰色（正在当前递归栈中）
    for (int v : g[u]) {
        if (color[v] == 1) return true;                    // 指回灰点，发现有向环！
        if (color[v] == 0 && has_cycle(v, g, color)) return true;
    }
    color[u] = 2;                                          // 2 = 黑色（全部后代搜完，安全
    退栈）
    return false;
}
// 主函数对每个 color[i] == 0 的点调用 has_cycle(i, g, color)
```

## 本页要点

记住三色口诀：进门涂灰（1），遇灰有环，出门涂黑（2）！

# 上机验证：3 组测试数据与反事实追问

| 测试场景 | 有向图输入（$n, m$ 与边） | 是否有环 | 考察目的 |
|---|---|---|---|
| 无环交叉边 | `4 4 / 1 2, 2 4, 1 3, 3 4` | `false` | 验证黑点不误报环 |
| 二元自环/对环 | `2 2 / 1 2, 2 1` | `true` | 最小有向环 |
| 不连通带环 | `4 3 / 1 2, 3 4, 4 3` | `true` | 环不在起点 1 所在分量 |

## 反事实追问

- 问：如果把有向图三色判环里的 `color[u] = 2` 删掉（只用 `vis[u] = 1`），上面哪组数据会错？
- 答：第 1 组（无环交叉边）！搜完 $1 \to 2 \to 4$ 后 $4$ 仍是 $1$，当搜到 $1 \to 3 \to 4$ 时误以为遇到环，直接输出 **WA**！

## 本页要点

写有向图判环时，一定要拿“菱形交叉边 $1 \to 2 \to 4,\ 1 \to 3 \to 4$”测一遍！

# 本节易错：有向图用布尔 vis 判环误报与递归爆栈 RE

## 常见错误写法

- 有向图只用 `vis[v]==1` 判环 $\rightarrow$ 交叉边误报 **WA**；
- 无向图判环忘了传 `fa`（父节点）$\rightarrow$ 立刻沿着来路撞回 `fa` 误判有环 **WA**；
- 链状图 $n = 10^5$ 时递归太深在部分系统上爆栈 **RE**（可用 BFS/Kahn 替代判环）。

## 正确写法

- 有向图判环用三色 `0/1/2`，或用下一节的 **Kahn** 拓扑排序（迭代不爆栈!）；
- 无向图判环写成 `if (vis[v] && v != fa)`；
- 若存在重边，无向图需用边编号 `e != (fa_edge ^ 1)` 区分父边。

## 本页要点

在 $n \geq 10^5$ 的有向图判环题里，用下一节的迭代版 **Kahn** 算法既简单又永远不会爆栈!

# 生活引入：穿衣服与大学选课的先后依赖

- 早上出门穿衣服是有严格先后顺序的：必须先穿内衣再穿毛衣，最后穿外套；先穿袜子再穿鞋子。
- 但袜子和内衣之间没有谁先谁后的强制要求，谁先穿都可以。
- 大学选课也一样：必须先修完《C++ 基础》，才能修《数据结构》；先修完《数据结构》，才能修《图论算法》。
- 把“先修课程 $u \to$ 后续课程 $v$”连成有向边，我们要排出一份不违反任何先修顺序的选课清单！

## 本页要点

只要有“干完 A 才能干 B”的先后依赖，本质上就是要在有向图里求一个拓扑序！

# 新术语大白话：DAG、前驱/后继、入度、拓扑序

- 有向无环图（**DAG**）：所有边都有箭头，而且图里没有任何闭环的有向图。
- 前驱与后继：如果有一条边 $u \to v$，那么 $u$ 叫做 $v$ 的前驱（先修课），$v$ 叫做 $u$ 的后继。
- 入度 `indeg[v]`：有多少条箭头直接指向点 $v$，就代表 $v$ 还差多少门先修课没学完。
- 拓扑序：把所有点排成一排，使得每一条有向边 $u \to v$ 里，$u$ 都排在 $v$ 的左边。

## 本页要点

`indeg[v] == 0` 的大白话含义就是：“$v$ 的所有先修条件已经全部满足，现在就能学了！”

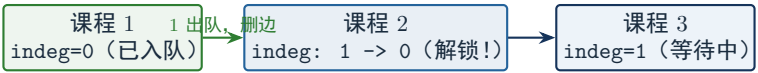
# 暴力找入度 0 点为什么是 $O(n^2+m)$ 会 TLE？

- 最朴素的想法：每轮从 $1\dots n$ 扫一遍所有点，找一个还没输出且 `indeg[u] == 0` 的点，输出它，然后把它指出的边删掉。
- 这样的扫描一共要进行 $n$ 轮，每轮扫 $n$ 个点，光找点就要花 $O(n^2)$ 次操作！
- 当 $n=10^5$ 时，$n^2=10^{10}$ 次操作，远超 1 秒 $10^8$ 次上限，必然 **TLE**！
- 优化洞察：只有当某条边被删掉、导致后继 $v$ 的 `indeg[v]` 减到 0 时，才诞生了新的可选点——根本不用每轮重扫全图，用一个队列记住它们就行！

## 本页要点

把“每轮重扫找 0”换成“队列随减随入队”，复杂度立刻从 $O(n^2+m)$ 降到 $O(n+m)$！

# TikZ 图解：Kahn 算法“剥洋葱”（入度归零即解锁入队）




1. 统计所有点的入度 `indeg[1..n]`，把一开始所有 `indeg[i] == 0` 的点塞进队列 `q`。
2. 队头 $u$ 出队，放进答案序列 `order`。
3. 遍历 $u$ 的所有后继 $v$，把 `--indeg[v]`；一旦归 $0$，立刻把 $v$ 塞进 `q`！

## 本页要点

Kahn 算法就像剥洋葱：每次把最外层“没前置依赖”的点剥掉，里面一层就暴露出来了！

# 手算跟踪表：5 点 5 边 DAG 的 Kahn 拓扑排序过程

**有向边：$1 \to 2$, $1 \to 3$, $2 \to 4$, $3 \to 4$, $4 \to 5$；初始 `indeg = [-, 0, 1, 1, 2, 1]`**

| 轮次 | 弹出队头 $u$ | 扣减后继入度 | 新入队点 | 当前队列 q（操作后） |
|---|---|---|---|---|
| 初始 | — | — | 1 | [1] |
| 1 | 1 | indeg[2]=0, indeg[3]=0 | 2, 3 | [2, 3] |
| 2 | 2 | indeg[4]=1（未归 0） | 无 | [3] |
| 3 | 3 | indeg[4]=0（归 0!） | 4 | [4] |
| 4 | 4 | indeg[5]=0（归 0!） | 5 | [5] |
| 5 | 5 | 无出边 | 无 | [] |
| 终止 | 队列为空，循环结束！输出 order = [1, 2, 3, 4, 5]，共 $5 = n$ 个点。 | | | |

## 本页要点

注意第 2 轮时 4 的入度从 2 减到 1，必须等第 3 轮 3 也出队后减到 0 才入队！

# 正确性论证：Kahn 算法循环不变量

| 阶段 | 循环不变量（Loop Invariant） |
|---|---|
| 初始化 | 入队前，q 恰好包含全部没有前驱（indeg == 0）的点，它们前面没有任何约束。 |
| 保持 | 每弹出点 $u$，说明 $u$ 的所有前驱早就出队排在 $u$ 左边了；把 $u$ 的出边删去后，新归 0 的后继 $v$ 也满足“前驱全部已出队”。 |
| 终止 | 队列为空时，若 order.size() == n，则每条边 $u \to v$ 的起点 $u$ 都在 $v$ 之前输出，必为合法拓扑序！ |

## 本页要点

每个点入队、出队各最多一次，每条边被扫一次，总时间复杂度严格 $O(n+m)$！

# 模板代码：Kahn 拓扑排序

```
queue<int> q;
for (int i = 1; i <= n; ++i)
    if (indeg[i] == 0) q.push(i);                    // 所有无前驱的起点同时入队
vector<int> order;
while (!q.empty()) {
    int u = q.front(); q.pop();
    order.push_back(u);                              // u 的所有前驱已排完，确定 u 的
    位置
    for (int v : g[u]) {
        if (--indeg[v] == 0) q.push(v);              // 前置全部完成，立刻解锁后继 v
    }
}
```

## 本页要点

加边时千万别忘了写 `++indeg[v]`，否则全图所有点一开始都以为自己入度为 $0$！

# 上机验证：3 组测试数据与反事实追问

| 测试场景 | 输入（$n, m$ 与边） | 合法拓扑序之一 | 考察目的 |
|---|---|---|---|
| 单链 | `3 2 / 3 1, 1 2` | `3 1 2` | 边方向不按编号大小排 |
| 多起点 | `4 2 / 3 4, 1 2` | `1 3 2 4` | 两个独立起点的交错出队 |
| 无边孤立点 | `3 0` | `1 2 3` | 全图入度均为 $0$ |

## 反事实追问

- 问：如果题目要求“在有多个点同时可选时，优先输出编号最小的（字典序最小拓扑序）”，普通 queue 还能保证吗？
- 答：不能！例如第 2 组 `1 2`，`3 4`，若 $3$ 先入队或 $2$ 后解锁，普通队列先入先出无法保证每次挑出当前最小编号，必须把 queue 换成小根堆！

## 本页要点

“任意合法拓扑序”用 queue；“字典序最小拓扑序”用小根堆 `priority_queue`！

# 字典序最小拓扑序：把 queue 换成小根堆

```cpp
priority_queue<int, vector<int>, greater<int>> pq;  // 小根堆：每次弹出编号最小的可选点
for (int i = 1; i <= n; ++i)
    if (indeg[i] == 0) pq.push(i);
vector<int> order;
while (!pq.empty()) {
    int u = pq.top(); pq.pop();                     // 注意堆取队头是 .top() 不是 .front()
    order.push_back(u);
    for (int v : g[u])
        if (--indeg[v] == 0) pq.push(v);
}
```

## 本页要点

换成小根堆后复杂度从 $O(n+m)$ 变为 $O((n+m)\log n)$，取堆顶记得用 `.top()`！

# 本节易错：加边漏写 ++indeg[v] 与有向边方向建反

## 常见错误写法

- 写了 `g[u].push_back(v)` 却忘了写 `++indeg[v]` → 全图入度全为 0 **WA**；
- 看到输入 `a b` 就无脑连 `a -> b`，没搞清“谁依赖谁” → 顺序全反 **WA**；
- 堆版拓扑排序把 `pq.top()` 写成 `pq.front()` → 编译报错 **CE**。

## 正确写法

- 养成肌肉记忆：连边 `u -> v` 同一行紧跟 `++indeg[v]`；
- 建边前默念一句：“必须先有 $u$，才能有 $v$”，就一定连 `u -> v`；
- 有重边时照样连边并累加 `indeg[v]`，Kahn 扣减次数刚好一一对应。

## 本页要点

“谁必须先确定，谁就是起点 $u$；谁要等它，谁就是终点 $v$ 并加 `indeg[v]`”！

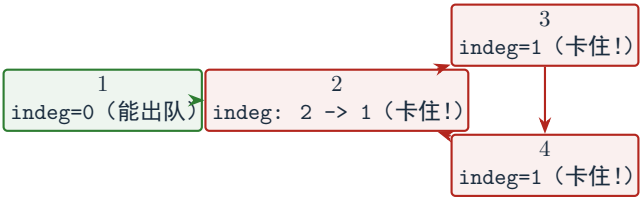
# 为什么有环图会让 Kahn 算法“卡住”？

- 假设出现矛盾要求：员工 $A$ 奖金必须比 $B$ 高，同时员工 $B$ 奖金也必须比 $A$ 高。
- 这在图上就是一个有向环：$A \to B$ 且 $B \to A$，此时 `indeg[A] = 1`，`indeg[B] = 1`。
- 运行 Kahn 算法时，没有任何一个点的入度是 $0$，队列一开始就是空的！
- 就算环外还有其他“尾巴”点能顺利出队，环上的点以及依赖环的点入度永远不可能减到 $0$，它们全都会被卡在图里出不来！

## 本页要点

Kahn 算法结束后，只要检查出队总数 `order.size() < n`，就 100% 证明图里存在有向环！

# TikZ 图解：带尾巴的环图中，环上点入度永不归 0




- 点 1 入度为 0，能顺利出队并把 2 的入度从 2 减到 1。
- 但 2 还在等 4，4 在等 3，3 在等 2，三人互相死锁，谁也进不去队列！

## 本页要点

部分点能出队不代表全图无环！必须核对出队点数是不是恰好等于 $n$！

# 手算跟踪表：4 点带环图运行 Kahn 算法

**有向边：$1 \to 2,\ 2 \to 3,\ 3 \to 4,\ 4 \to 2$；初始 `indeg = [-, 0, 2, 1, 1]`**

| 轮次 | 弹出队头 $u$ | 扣减后继入度 | 当前 `order.size()` | 队列 `q` |
|---|---|---|---|---|
| 初始 | — | — | 0 | [1] |
| 1 | 1 | `indeg[2]` 从 2 减为 1（仍 $> 0$） | 1 | [] |
| 终止 | 队列已空，循环提前结束！此时 `order.size() = 1 < 4`，判定存在有向环！ | | | |

## 本页要点

用 Kahn 拓扑排序判环是迭代写法，绝对不会像递归 DFS 那样在 $n = 10^5$ 时爆栈！

# 为什么能在拓扑序上做 DAG 动态规划？

- 回想《05_ 动态规划》篇：写 DP 最怕“算 $f[v]$ 的时候，它依赖的 $f[u]$ 还没算完”。
- 在线性序列里，我们按 $i = 1 \dots n$ 的顺序就能保证前面都算完了；但在任意一张有向无环图（DAG）里，点编号可能是乱的（比如 $5 \to 2$）。
- 而拓扑序的本质就是把这张网状 DAG “拉直”成一条从左到右的线！
- 当点 $u$ 从 Kahn 队列里弹出来时，它的所有前驱都已经出队并向 $u$ 转移完毕——也就是说，$dp[u]$ 在此刻已经彻底定型，绝不会再变！

## 本页要点

拓扑序 = 给任意 DAG 找到了合法的 DP 填表顺序（无后效性保证）！

# DAG 最长路 / 递推转移怎么写？

- 以“求 DAG 上以各点结尾的最长路径长度 `dp[v]`”为例（每条边权为 $w$，或点权）：
  - 初始化：所有点 `dp[i] = 0`（或题目给的保底值，如《奖金》的 $100$）；
  - 当 $u$ 出队时，枚举它的每条出边 $u \to v$，用已经定型的 `dp[u]` 去更新后继 `dp[v]`：

$$dp[v] = \max(dp[v],\ dp[u] + w)$$

  - 最后再把 `--indeg[v]`，若归 $0$ 则入队。
- 注意这里用的是刷表法（由当前已算好的 $u$ 去推未来的 $v$），非常自然！

## 本页要点

在拓扑序里边出队边刷表，一定要写 `dp[v] = max(dp[v], dp[u] + w)` 取最大值！

# 模板代码：拓扑序判环 + DAG 最长路递推

```
queue<int> q;
vector<long long> dp(n + 1, 0);                    // 初值按题意设定（如保底值）
for (int i = 1; i <= n; ++i) if (indeg[i] == 0) q.push(i);
int vis_cnt = 0;
while (!q.empty()) {
    int u = q.front(); q.pop(); ++vis_cnt;          // 此时 dp[u] 已彻底定型
    for (auto [v, w] : g[u]) {
        dp[v] = max(dp[v], dp[u] + w);              // 用定型的 dp[u] 去刷后继 dp[v
    ]
        if (--indeg[v] == 0) q.push(v);
    }
}
bool has_cycle = (vis_cnt < n);                    // 出队人数不足 n，必有环！
```

## 本页要点

既完成了判环，又顺手做完了整张 DAG 的 DP 递推，一趟 $O(n+m)$ 搞定！

# 正确性论证：为什么算 $v$ 时所有前驱 $u$ 都已定型？

| 步骤 | 论证说明 |
| --- | --- |
| 入队前提 | 点 $v$ 只有在 `indeg[v]` 减到 $0$ 的那一刻才会进入队列 `q`。 |
| 前驱已全出队 | `indeg[v]` 归 $0$ 意味着指向 $v$ 的每一条入边 $u \to v$ 都已经在 $u$ 出队时被处理并更新过 `dp[v]` 了。 |
| 不会再被修改 | 此后队列里再也没有 $v$ 的前驱，所以当 $v$ 自己出队去刷别人时，`dp[v]` 一定是所有前驱取 `max` 后的最终定型值！ |

## 本页要点

“入度归零才入队”天然保证了“所有前驱先算完”，这就是 DAG 拓扑 DP 绝对不会算漏的原因！

# 上机验证：3 组测试数据与反事实追问

| 测试场景 | 输入（$n, m$ 与边，边权均为 1） | 结果 | 考察目的 |
|---|---|---|---|
| 自环 | `2 1 / 1 1` | 有环（`vis_cnt=1<2`） | 自己指自己入度为 1 出不去 |
| 孤立环 | `4 3 / 1 2, 2 1, 3 4` | 有环（`vis_cnt=2<4`） | 尾巴 $3, 4$ 能出队但环出不去 |
| 汇合最长路 | `4 4 / 1 2, 2 4, 1 3, 3 4` | `dp[4] = 2` | 多条前驱汇合取 `max` |

## 反事实追问

- 问：如果把 `dp[v] = max(dp[v], dp[u] + w);` 直接写成 `dp[v] = dp[u] + w;` 会怎样？
- 答：假设 $1 \to 2 \to 4$（长 2）和 $1 \to 3 \to 4$（长 2），或者两条路径一长一短（长 5 和长 1）先后更新 4，后面较短的那条路径会把前面已经算好的较优值直接覆盖改小，导致 **WA**！

## 本页要点

刷表法更新后继时，永远要用 `max`（或 `min`），绝不能直接用 `=` 覆盖！

# 本节易错：只看队列空不判 cnt < n、DAG DP 未取 max

## 常见错误写法

- 只写 `while (!q.empty())` 却忘了统计 `vis_cnt < n` → 漏判有环 **WA**；
- 刷表更新写 `dp[v] = dp[u] + w` → 被后来的较差前驱覆盖 **WA**；
- 求方案数（如食物链计数）时忘了取模或没用 `long long` → 溢出 **WA**。

## 正确写法

- 每次 `q.pop()` 时紧跟 `++vis_cnt`，循环后判断 `if (vis_cnt < n)`；
- 更新后继写成 `dp[v] = max(dp[v], dp[u] + w)`；
- 累加和/方案数开 `long long` 并按题面要求取模。

## 本页要点

出队计数 `vis_cnt < n` 判环、更新后继取 `max`、累加开 `long long`，一个都不能少！

# 例 1 | 查找文献（P2125）：先读懂要求 [入门]

- 输入：第一行 $n, m$（$n \le 10^5, m \le 10^6$）；接下来 $m$ 行每行 $X, Y$，表示文章 $X$ 引用了文章 $Y$（有向边 $X \to Y$）。
- 处理：从编号 $1$ 的文章出发，分别用 DFS 和 BFS 不重复地访问所有能看到的文章；有多篇可选时，优先看编号较小的那篇。
- 输出：两行，第一行为 DFS 遍历结果，第二行为 BFS 遍历结果。

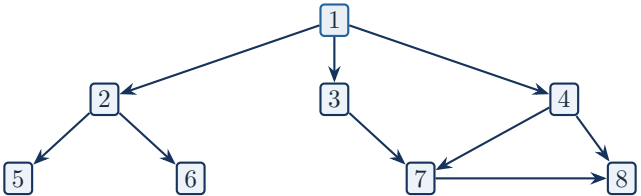
## 原题样例

输入：`8 9 / 1 2, 1 3, 1 4, 2 5, 2 6, 3 7, 4 7, 4 8, 7 8`
输出：第 1 行 `1 2 5 6 3 7 8 4`，第 2 行 `1 2 3 4 5 6 7 8`

## 本页要点

题目要求“优先看编号较小的”，所以建好 `vector` 邻接表后，必须对每个点的出边先 `sort`！

# 例 1 | 怎么想到的？邻接表排序 + 独立两次 vis 遍历




- 因为 $n \le 10^5, m \le 10^6$，且需要对每个点的出边按编号升序排序，使用 `vector<vector<int>> g(n+1)` 最方便！
- 先跑完 DFS，记得用 `fill(vis.begin(), vis.end(), 0)` 清空 vis，再跑 BFS。

## 本页要点

注意题面提醒“不保证 1 没被引用”，但我们只从 1 出发访问能到的点，千万别多跑全图循环！

# 例 1 | 样例手推跟踪表（DFS 与 BFS 对比）

| 步骤 | DFS 访问顺序（一条路扎到底） | BFS 队列与访问（一圈一圈扩散） |
|---|---|---|
| 起点 | 访问 1 | 1 入队并标记，出队 1 |
| 第 1 层 | 进 2 → 进 5 → 退回进 6 | 1 的邻居 2，3，4 依次入队并出队 |
| 第 2 层 | 退回 1 → 进 3 → 进 7 → 进 8 | 弹出 2 时入队 5，6；弹出 3 时入队 7 |
| 第 3 层 | 退回 1 → 进 4（7,8 已标记跳过） | 弹出 4 时 7 已标，只入队 8 |
| 终止 | 得 1 2 5 6 3 7 8 4 | 得 1 2 3 4 5 6 7 8 |

## 本页要点

看第 3 层：7 和 8 都已经被前面的点标记过 `vis=1`，所以后续边指过去时直接跳过！

# 例 1 | 核心步骤表与模板改动

| 按顺序计算 | 纸上得到 | 核对问题 |
| --- | --- | --- |
| 1. 用 `vector` 读入 $m$ 条有向边 | 每个点的出边列表 | 是否只连了单向边 $u \to v$? |
| 2. 对每个 `g[i]` 升序排序 | 保证先访问编号小的邻居 | 别忘了 `sort(g[i].begin(), g[i].end())` |
| 3. 先从 1 跑 DFS，清空 `vis` 后再从 1 跑 BFS | 两个遍历结果数组 | BFS 是否在入队时立刻标 `vis[v]=1`? |

## 相对模板改了哪里

在标准 DFS/BFS 模板前多了一步“对每个点的邻接表排序”；因为 $m \le 10^6$，读入输出务必加 `ios::sync_with_stdio(false); cin.tie(nullptr);`。

## 本页要点

只从点 1 出发即可，不可到达的孤立点不要输出！

# 例 1 | 核心代码（1/2）：建图排序与 DFS

```
vector<vector<int>> g(n + 1);
for (int i = 0, u, v; i < m; ++i) { cin >> u >> v; g[u].push_back(v); }
for (int i = 1; i <= n; ++i) sort(g[i].begin(), g[i].end());
vector<int> vis(n + 1, 0), dfs_ans, bfs_ans;
auto dfs = [&](auto&& self, int u) -> void {
    vis[u] = 1; dfs_ans.push_back(u);
    for (int v : g[u]) if (!vis[v]) self(self, v);
};
dfs(dfs, 1);
```

## 本页要点

完整源码：`code/literature_search.cpp`；C++14/17/20 均可用泛型 Lambda `auto&& self` 实现递归。

# 例 1 | 核心代码（2/2）：清空标记与 BFS 输出

```
fill(vis.begin(), vis.end(), 0);                    // 跑 BFS 前必须清空 vis!
queue<int> q; q.push(1); vis[1] = 1;                // 起点入队立刻标记
while (!q.empty()) {
    int u = q.front(); q.pop(); bfs_ans.push_back(u);
    for (int v : g[u]) if (!vis[v]) { vis[v] = 1; q.push(v); }
}
for (int i = 0; i < (int)dfs_ans.size(); ++i)
    cout << dfs_ans[i] << (i + 1 == (int)dfs_ans.size() ? '\n' : ' ');
for (int i = 0; i < (int)bfs_ans.size(); ++i)
    cout << bfs_ans[i] << (i + 1 == (int)bfs_ans.size() ? '\n' : ' ');
```

## 本页要点

BFS 入队时标记 `vis[v] = 1`，保证每个点最多入队一次，时间复杂度 $O(n + m \log m)$！

# 例 1 | 易错与 3 组自测边界

## 常见易错点

- 跑完 DFS 忘了 `fill` 清空 `vis` → BFS 只输出 1 或空 **WA**；
- 忘了对 `g[i]` 排序 → 按输入顺序访问 **WA**；
- 错连成无向图 → 沿着反向边乱跑 **WA**。

## 3 组自测边界数据

1. 无边图：`3 0` ⇒ 两行都是 `1`；
2. 含重边与环：`2 3 / 1 2, 1 2, 2 1` ⇒ 两行都是 `1 2`；
3. 倒序输入：`4 3 / 1 4, 1 3, 1 2` ⇒ 两行都是 `1 2 3 4`。

## 本页要点

有环图上跑 DFS/BFS，用 `2 3 / 1 2, 1 2, 2 1` 测一遍就能验证 `vis` 没写漏！

# 例 2 | 家谱树（1351）：先读懂要求 [基础]

- 输入：第 1 行 $N$（$1 \le N \le 100$），表示家族人数；接下来 $N$ 行，第 $i$ 行给出第 $i$ 个人的所有儿子编号，每行以 0 结束。
- 处理：排一个序列，使每个人的所有后辈都排在自己后面（任意合法解均可）。
- 输出：一行 $N$ 个整数，空格隔开。

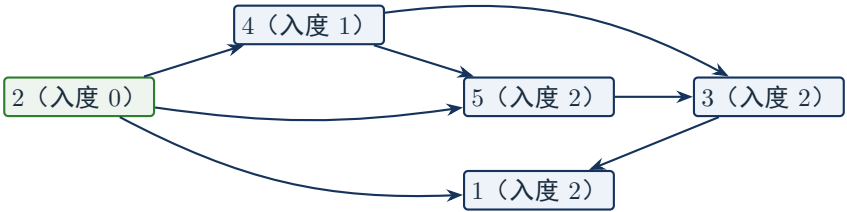
## 原题样例

输入：5 / 0 / 4 5 1 0 / 1 0 / 5 3 0 / 3 0 ⇒ 输出：2 4 5 3 1

## 本页要点

注意输入格式不是给出总边数 $m$，而是每行读到 0 为止！

# 例 2 | 怎么想到的？长辈 → 晚辈连边求拓扑序




- 题目要求“每个人的后辈都比那个人后列出”，也就是长辈在前、晚辈在后。
- 第 $i$ 行读到儿子 $v$ 时，连一条有向边 $i \to v$，并让 `++indeg[v]`。
- 家族关系天然无环（不可能互为长辈），直接跑标准 Kahn 队列拓扑排序即可！

## 本页要点

只要分清“长辈 $i$ 在前 → 儿子 $v$ 在后”，加边方向就一目了然！

# 例 2 | 样例手推跟踪表（输入 $N = 5$）

## 初始 indeg = [-, 2, 0, 2, 1, 2]（只有 2 没有长辈，入度为 0）

| 轮次 | 弹出 $u$ | 扣减后继入度 | 新入队 | 当前 ans |
|---|---|---|---|---|
| 1 | 2 | 4->0, 5->1, 1->1 | 4 | [2] |
| 2 | 4 | 5->0, 3->1 | 5 | [2, 4] |
| 3 | 5 | 3->0 | 3 | [2, 4, 5] |
| 4 | 3 | 1->0 | 1 | [2, 4, 5, 3] |
| 5 | 1 | 无出边 | 无 | [2, 4, 5, 3, 1] |
| 终止 | 队列为空，输出 2 4 5 3 1，与原题样例完全一致！ | | | |

## 本页要点

第 1 行输入是 0，代表 1 号没有任何儿子（他是晚辈），而不是代表 1 号入度为 0！

# 例 2 | 核心步骤表与模板改动

| 按顺序计算 | 纸上得到 | 核对问题 |
| --- | --- | --- |
| 1. 循环 $i = 1 \dots n$，每行 `while (cin >> v && v)` 读入 | 建边 $i \to v$，`++indeg[v]` | 遇到 0 是否正确跳出内层循环？ |
| 2. 把所有 `indeg[i]==0` 入队 | 队列初始化完成 | 别漏了有多个祖先的情况！ |
| 3. 标准 Kahn 循环弹出并减入度 | 得到完整序列 `ans` | 行末是否输出换行且无多余尾空格？ |

## 相对模板改了哪里

唯一改动是读入部分用 `while (cin >> v && v)` 处理以 $0$ 结尾的邻接输入，其余与标准 Kahn 模板完全一致。

## 本页要点

用 `while (cin >> v && v)` 读到 $0$ 自动停止，简洁又不容易死循环！

# 例 2 | 核心代码（1/2）：以 0 结尾建图与入队

```
int n; if (!(cin >> n)) return 0;
vector<vector<int>> g(n + 1);
vector<int> indeg(n + 1, 0);
for (int i = 1; i <= n; ++i) {
    int v; while (cin >> v && v) { g[i].push_back(v); ++indeg[v]; }
}
queue<int> q;
for (int i = 1; i <= n; ++i) if (indeg[i] == 0) q.push(i);
```

## 本页要点

完整源码：code/family_tree.cpp；注意 ++indeg[v] 写在加边同一行，绝不遗漏。

# 例 2 | 核心代码（2/2）：Kahn 出队与格式输出

```
vector<int> ans;
while (!q.empty()) {
    int u = q.front(); q.pop(); ans.push_back(u);
    for (int v : g[u]) if (--indeg[v] == 0) q.push(v); // 先辈全处理完才解锁晚辈
}
for (int i = 0; i < (int)ans.size(); ++i)
    cout << ans[i] << (i + 1 == (int)ans.size() ? '\n' : ' ');
return 0;
```

## 本页要点

输出时用三目运算符控制空格与换行，避免行末多出空格被部分严谨 OJ 判 PE！

# 例 2 | 易错与 3 组自测边界

## 常见易错点

- 把儿子连向长辈（$v \to i$）$\to$ 辈分完全颠倒 **WA**；
- 忘了 `&& v` 把结束符 0 也当成儿子加进边里 $\to$ 破坏入度 **WA**；
- 只从 1 号入队而没扫全图 `1..n` $\to$ 找不到真正祖先 **WA**。

## 3 组自测边界数据

1. 只有 1 人：`1 / 0` $\Rightarrow$ `1`；
2. 全是同辈：`3 / 0 / 0 / 0` $\Rightarrow$ `1 2 3`；
3. 多祖先链：`4 / 3 0 / 4 0 / 0 / 0` $\Rightarrow$ `1 2 3 4`。

## 本页要点

自测“全是同辈（每行只有一个 0）”能立刻检查多个入度 0 起点是否全部入队！

# 例 3 | 奖金（1352）：先读懂要求 [基础]

- 输入：第一行 $n, m$（$n \le 10^4, m \le 2 \times 10^4$）；接下来 $m$ 行每行 $a, b$，表示代表认为员工 $a$ 的奖金应严格高于员工 $b$。
- 处理：每人保底 $100$ 元，求满足所有意见的最小总奖金；若存在矛盾无法满足，输出 `Poor Xed`。
- 输出：最少总奖金，或 `Poor Xed`。

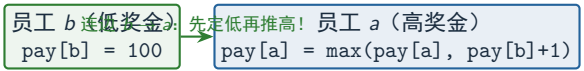
## 原题样例

输入：`2 1 / 1 2` ⇒ 输出：`201`

## 本页要点

看到“无法找到合理方案输出特定字符串”，立刻想到 Kahn 拓扑排序判环 `vis_cnt < n`！

# 例 3 | 怎么想到的？反向建边 $b \to a$ + 拓扑递推




- 要使总奖金最少，没有被要求比别人高的员工（最低层）当然只发保底的 100 元。
- 而 $a$ 的奖金必须等所有不如他的 $b$ 都算完之后才能确定：$pay[a] \geq \max(pay[b]+1)$。
- 所以必须反向建边 $b \to a$（谁低谁先出队，再去推高别人），并让 `++indeg[a]`！

## 本页要点

这是拓扑排序最经典陷阱：输入说“$a$ 比 $b$ 高”，建边必须连 `b -> a` 才能从低往高递推！

# 例 3 | 小数据手推跟踪表（构造 3 人 2 约束）

**输入 3 2 / 1 2，1 3（1 比 2 高，且 1 比 3 高）⇒ 连边 $2 \to 1, 3 \to 1$**

初始 `indeg = [-, 2, 0, 0]`，`pay = [-, 100, 100, 100]`，`sum = 0`。

| 轮次 | 弹出 $u$ | 累加 sum | 更新后继 pay 与入度 | 队列 q |
|---|---|---|---|---|
| 1 | 2 | 100 | `pay[1]=max(100,101)=101, indeg[1]=1` | [3] |
| 2 | 3 | 200 | `pay[1]=max(101,101)=101, indeg[1]=0` → 入队 1 | [1] |
| 3 | 1 | 301 | 无出边 | [] |
| 终止 | 队列空，`vis_cnt = 3 == n`，无环！输出总奖金 301。 | | | |

## 本页要点

如果输入再加一条 `2 1`，则 $1, 2$ 互相卡死，`vis_cnt = 1 < 3`，输出 `Poor Xed!`

# 例 3 | 核心步骤表与模板改动

| 按顺序计算 | 纸上得到 | 核对问题 |
|---|---|---|
| 1. 读入 $a, b$，连边 $b \to a$ 并 `++indeg[a]` | 低奖金指向高奖金 | 方向有没有连成 $a \to b$? |
| 2. 初始化 `pay[1..n] = 100`，入度 0 入队 | 底层员工保底 100 | `sum` 是否开了 `long long`? |
| 3. 出队累加 `sum += pay[u]` 并刷后继 | 更新 `max(pay[v], pay[u]+1)` | 结尾是否检查 `vis_cnt < n`? |

## 相对模板改了哪里

读入 `a b` 时反向连边 `g[b].push_back(a);` `pay` 数组初始值设为 100； 边权固定为 1，即 `pay[v] = max(pay[v], pay[u] + 1);` 若 `vis_cnt < n` 输出 `Poor Xed`。

## 本页要点

最大总奖金最坏可达 $100n + \frac{n(n-1)}{2} \approx 5 \times 10^7$，但养成习惯 `sum` 一律开 `long long`!

# 例 3 | 核心代码（1/2）：反向建边与初始化

```
int n, m; if (!(cin >> n >> m)) return 0;
vector<vector<int>> g(n + 1);
vector<int> indeg(n + 1, 0), pay(n + 1, 100);
for (int i = 0, a, b; i < m; ++i) {
    cin >> a >> b;                                  // a 比 b 奖金高 -> 连边 b -> a
 递推加钱
    g[b].push_back(a); ++indeg[a];
}
queue<int> q;
for (int i = 1; i <= n; ++i) if (indeg[i] == 0) q.push(i);
```

## 本页要点

完整源码：code/bonus_topo.cpp；注意 g[b].push_back(a) 与 ++indeg[a] 的对应关系！

# 例 3 | 核心代码（2/2）：拓扑递推与判环输出

```
    int vis_cnt = 0; long long sum = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop(); ++vis_cnt; sum += pay[u];
        for (int v : g[u]) {
            pay[v] = max(pay[v], pay[u] + 1);        // 拓扑递推更新至少多 1 元
            if (--indeg[v] == 0) q.push(v);
        }
    }
    if (vis_cnt < n) cout << "Poor Xed\n";           // 有员工没进过队列说明存在矛盾环
    else cout << sum << '\n';
    return 0;
}
```

## 本页要点

输出字符串 `Poor Xed` 区分大小写与空格，务必逐字核对原题！

# 例 3 | 易错与 3 组自测边界

## 常见易错点

- 建边连成 $a \to b \to$ 高奖金先出队发 100 元 **WA**；
- 更新写 `pay[v] = pay[u] + 1` 没取 max $\to$ 被短链覆盖 **WA**；
- 忘了写 `vis_cnt < n` 判环 $\to$ 有矛盾时输出部分奖金 **WA**。

## 3 组自测边界数据

1. 矛盾环：`3 3 / 1 2, 2 3, 3 1` $\Rightarrow$ `Poor Xed`；
2. 无要求：`3 0` $\Rightarrow$ `300`；
3. 长短链汇合：`3 3 / 1 2, 2 3, 1 3` $\Rightarrow$ `303`（$3:100, 2:101, 1:102$）。

## 本页要点

交题前一定拿 `3 3 / 1 2, 2 3, 3 1` 测一次 `Poor Xed`，再拿 `1 2, 2 3, 1 3` 测一次 `max`！

# 图遍历与拓扑排序高频错误对照表

| 题目 / 场景 | 常见错误写法 | 正确做法与原因 |
|---|---|---|
| 有向边方向建反 | 见 `a b` 就无脑连 `a -> b` | 先问谁先定再推谁：《奖金》必须连 `b -> a` |
| 忘记累计入度 | 加了边没写 `++indeg[v]` | 全图所有点一开始都当入度 0 涌进队列全错 |
| 有向图判环用布尔 | 只用 `vis[v]` 判重就报有环 | 交叉边不是环！用三色 `0/1/2` 或 Kahn 判环 |
| 拓扑 DP 忘了 max | 写 `pay[v] = pay[u] + 1` | 后来的低奖金前驱会把已算好的高奖金改小 |
| 判环漏看连通总数 | 只看队列空不比 `vis_cnt < n` | 孤立闭环进不去队列，必须核对出队人数 $== n$ |

## 本页要点

“谁被依赖谁当起点，加边同步加 `indeg`，出队不足 $n$ 个就是有环”！

# 评测结果定位指南（CE / WA / RE / TLE）

| 评测结果 | 本章优先排查方向 |
|---|---|
| **CE** | 小根堆是否写成 `pq.front()`（应为 `pq.top()`），或漏了 `<queue>` / `<algorithm>`？ |
| **WA** | 查四处： 边方向是否建反； `++indeg[v]` 是否漏写； 拓扑 DP 是否取了 `max`； 判环是否用了 `vis_cnt < n`？ |
| **RE** | 查两处： 容器大小是否开了 `n + 1`； 链式前向星无向图边数组是否开了 `2m + 5`？ |
| **TLE** | 查两处： BFS 是否在入队时就标了 `vis[v] = 1`； $m = 10^6$ 时是否加了 `ios::sync_with_stdio(false)`？ |

## 本页要点

图论题出现 TLE，80% 是 BFS 出队才标 `vis` 导致重复入队，或者忘了关同步流！

# 提交前检查清单（六必查）

1. 邻接表加边时，是不是明确区分了“谁先发生、谁后依赖”（如 $b \to a$），并在同一行同步写了 `++indeg[a]`？
2. 存在“矛盾/不可能”情况的题，结尾是否用出队计数 `vis_cnt < n` 判定并按原题大小写输出？
3. 拓扑序同时更新节点权值时，是否取了 `max`（或 `min`）防止后来的较差前驱覆盖掉较优值？
4. 如果题面要求“字典序最小”的拓扑序，是否已把 `queue` 换成了小根堆 `priority_queue`？
5. 边权与人数累加（如总奖金、方案数）是否使用了 `long long` 防溢出？
6. 多组测试数据或先 DFS 再 BFS 时，`vis[]`、`indeg[]` 与邻接表有没有每次彻底清空？

## 最后一步

造一组 3 点环状依赖 1>2，2>3，3>1 自测确认输出 `Poor Xed` 后再提交！

# 本章全部练习与完成标准

1. 学校 OJ：P2125 | 查找文献（[入门]，邻接表排序 + DFS/BFS 遍历，源码 `code/literature_search.cpp`）
2. 学校 OJ：P2124 | 图的遍历（[基础]，反向建边 + 从大到小 DFS/BFS 染色）
3. 学校 OJ：P1351 | 家谱树（[基础]，Kahn 拓扑排序，源码 `code/family_tree.cpp`）
4. 学校 OJ：P1352 | 奖金（[基础]，反向建边 + DAG 拓扑 DP + 判环，源码 `code/bonus_topo.cpp`）

## 完成标准

能独立写出 `vector` 邻接表与链式前向星存图；能写出 DFS/BFS、三色判环与 Kahn 拓扑排序；理解 DAG 拓扑递推为什么不会受遍历顺序干扰，并独立 AC 以上 4 题。

# 课后安排与下一章预告

- 复习：合上代码再走一遍《奖金》里从底薪 100 逐层向上推最大值的过程，试着自己造一个 4 点小 DAG 验算。
- 练习：独立复写 code/ 下三份源码，尝试把《家谱树》改写成输出字典序最小版本，并独立 AC《图的遍历》（P2124）。
- 下一章预告：第 2 章《欧拉回路与欧拉路》——小学生都会玩的“一笔画图形”，怎样写代码不超时地一笔走完每一条边且保证字典序最小！

# 本章一句话

## 本章一句话

邻接表省内存、BFS 入队先标记；入度为零解锁入队，出队删边减后继入度，不足全员即现身环路。

谢谢！