连通分量与基础遍历 (Connected Components)

核心逻辑：

把图分成几个互不相连的“圈子”。

- 怎么找圈子？写一个主循环，遍历所有节点。如果节点没访问过，就启动 DFS 把整个圈子染一遍色。
- 怎么存图？用**邻接表** `graph = [[] for _ in range(n)]`。
- `547. 省份数量` (标准模版题)
- `1319. 连通网络的操作次数` (统计有几个圈子，然后 -1 就是需要的线)
- `2316. 统计无法互相到达点对数` (算出每个圈子大小，用数学公式算对数)
- `2685. 统计完全连通分量的数量` (DFS 统计节点数和边数)
- `2492. 两个城市间路径的最小分数` (其实就是求连通分量里的最小边权)
- `841. 钥匙和房间` (从 0 出发能不能遍历完所有点)
- `721. 账户合并` (建图后求连通分量，稍难在建图)
- `323. 无向图中连通分量的数目` (会员题，同 547)

```
def solve_components(n, edges):
    # 1. 建图 (邻接表)
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u) # 无向图两边都要加

    visited = [False] * n
    components_count = 0

    # 2. DFS 染色函数
    def dfs(u):
        visited[u] = True
        # 这里可以统计组件的大小、边权最小值等
        # size += 1
        for v in graph[u]:
            if not visited[v]:
                dfs(v)

    # 3. 主循环：遍历所有节点
    for i in range(n):
        if not visited[i]:
            # 发现新的连通分量
            components_count += 1
            dfs(i)

    return components_count
```

环检测与拓扑排序 (Cycle Detection)

核心逻辑：

有向图中是否存在环？或者依赖关系是否冲突？

关键点：需要三个状态来标记节点。

- `0`：未访问
- `1`：**正在访问** (在递归栈里，如果再次遇到它，说明有环！)
- `2`：访问结束 (安全的)

包含题目：

- `207. 课程表` (图中标注了“三色标记法判环”，必考！)
- `802. 找到最终的安全状态` (不在环里、也不通向环的节点)
- `261. 以图判树` (树 = 连通 + 无环)
- `1971. 寻找图中是否存在路径` (也可以用 BFS/并查集)

```
def has_cycle(n, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)

    # state: 0=未搜, 1=搜索中, 2=搜完(安全)
    state = [0] * n

    def dfs(u):
        # 遇到正在搜的节点 -> 有环!
        if state[u] == 1: return True
        # 遇到搜过的安全节点 -> 无环，直接退
        if state[u] == 2: return False

        state[u] = 1 # 标记为"正在搜"
        for v in graph[u]:
            if dfs(v): return True
        state[u] = 2 # 标记为"安全"
        return False

    for i in range(n):
        if dfs(i): return True # 发现环
    return False
```

单源/多源最大扩散

核心逻辑：

这不是求连通性，而是求 “从哪个点出发能炸得最多 / 传得最远”。

通常需要对 每个节点 都跑一次 DFS，或者从 特定的“捣乱点” 跑 DFS。

包含题目：

- `2101. 引爆最多的炸弹` (从每个炸弹 DFS 一次，看能炸多少个，取最大值)
- `924. 尽量减少恶意软件的传播` (从病毒源 DFS)
- `2192. 有向无环图中一个节点的所有祖先` (反向建图 DFS 或正向多次 DFS)
- `1306. 跳跃游戏 III` (数组上的 DFS)
- `2092. 找出知晓秘密的所有专家` (带时间限制的传播)

```
def max_spread(n, edges):
    # 建图 (有向)
    graph = [[] for _ in range(n)]
    # ... (省略建图细节)

    def dfs(u, visited):
        count = 1
        visited.add(u)
        for v in graph[u]:
            if v not in visited:
                count += dfs(v, visited)
        return count

    max_count = 0
    # 对每个点都试一遍
    for i in range(n):
        visited = set()
        # 每次 DFS 都是全新的 visited
        max_count = max(max_count, dfs(i, visited))

    return max_count
```

所有路径回溯 (All Paths Backtracking)

题目问的不是“能不能到”，而是 “把所有能到的路线都打印出来”。

这就需要 Backtracking (回溯)：在 DFS 到底后，退回来的时候要 撤销选择 (path.pop)，让程序能去走别的岔路。

包含题目：

- `797. 所有可能的路径` (标准模版题)

```
def allPathsSourceTarget(graph):
    target = len(graph) - 1
    results = []
    path = [0] # 起点

    def dfs(u):
        # Base Case: 到达终点
        if u == target:
            results.append(path[:]) # 收集结果(深拷贝)
            return

        for v in graph[u]:
            # 做选择
            path.append(v)
            dfs(v)
            # 撤销选择 (回溯)
            path.pop()

    dfs(0)
    return results
```

“网格图 DFS” (Grid DFS)

基础染色流 (Flood Fill / Stats)

找连通块、算面积、算周长、给岛屿染色。

逻辑：

遍历网格 $\to$ 发现新大陆 (grid[i][j] == 1) $\to$ 启动 DFS 走遍整个岛 $\to$ 记录数据。

包含题目：

- `200. 岛屿数量` (最基础)
- `695. 岛屿的最大面积` (带返回值)
- `733. 图像渲染` (换个颜色而已)
- `463. 岛屿的周长` (DFS 过程中判断边界)
- `2658. 网格图中鱼的最大数目` (每个格子的权值不同，求和)
- `1034. 边界着色` (只染边缘)

🩹 **必背模版：带返回值的染色 DFS**

```
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        R, C = len(grid), len(grid[0])

        # DFS 函数：返回当前连通块的某种统计值 (面积、鱼的数量等)
        def dfs(r, c):
            # 1. 越界检查
            if not (0 <= r < R and 0 <= c < C): return 0
            # 2. 有效性检查 (是水(0) 或者 已经访问过(0))
            if grid[r][c] == 0: return 0
```

```
            # 3. 标记访问 (通常直接修改原数组最省空间)
            # 如果题目有具体数值(如鱼的数量)，先存下来再清零
            val = 1 # 或者 grid[r][c]
            grid[r][c] = 0

            # 4. 扩散 (上下左右)
            area = val
            for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                area += dfs(r + dr, c + dc)

            return area

        max_ans = 0
        for r in range(R):
            for c in range(C):
                if grid[r][c] != 0: # 发现新大陆
                    max_ans = max(max_ans, dfs(r, c))

        return max_ans
```

门派二：边缘逃生流 (Boundary / Enclave)

核心考点：

这类题目通常问：“有多少个岛屿是被完全包围的（没接壤边界）？”或者“水能不能流到边界？”

逻辑：

逆向思维！ 不要从中间找，要从 边界 (Border) 出发！

1. 先遍历网格的**四条边**。
2. 如果边界上有岛屿 (`1`)，说明这个岛屿“通向自由”，不是飞地。启动 DFS 把这个岛全标记为“安全”。
3. 剩下的没被标记的 `1`，就是“瓮中之鳖”（飞地/被包围的）。

包含题目：

- `130. 被围绕的区域` (把没接壤边界的 O 变成 X)
- `1020. 飞地的数量` (求没接壤边界的 1 的个数)
- `1254. 统计封闭岛屿的数目` (同上，只是 0 和 1 定义反了)
- `417. 太平洋大西洋水流问题` (从左上边界 DFS 一次，右下边界 DFS 一次，看交集)

必背模版：边缘入侵法

Python

```
class Solution:
    def numEnclaves(self, grid: List[List[int]]) -> int:
        R, C = len(grid), len(grid[0])

        # 1. 简单的 DFS (只负责把地淹没/标记)
        def dfs(r, c):
            if not (0 <= r < R and 0 <= c < C) or grid[r][c] == 0:
                return
            grid[r][c] = 0 # 淹没它 (标记为非飞地)
            for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                dfs(r + dr, c + dc)

        # 2. 先从四条边界出发，把靠边的岛都铲除
        for r in range(R):
            dfs(r, 0)     # 左边界
            dfs(r, C - 1) # 右边界
        for c in range(C):
            dfs(0, c)     # 上边界
            dfs(R - 1, c) # 下边界

        # 3. 剩下的 1 全是飞地，统计一下 (如果是 130 题，就在这里做替换)
        count = 0
        for r in range(R):
            for c in range(C):
                if grid[r][c] == 1:
                    count += 1 # 或者 count += 1 这里的逻辑视题目而定

        return count
```

门派三：逻辑验证流 (Validation / Relationship)

核心考点：

这属于 Grid DFS 的进阶版。不再是单纯的染色，而是 DFS 过程中需要验证某种条件，或者对比两个网格的关系。

逻辑：

DFS 不再只返回面积，而是返回 True/False。

包含题目：

- `1905. 统计子岛屿` (必须 B 岛屿的所有土地都在 A 岛屿的土地上)
- `1391. 检查网格中是否存在有效路径` (管道拼接，稍微麻烦点的 DFS)
- `1559. 二维网格图中探测环` (需要记录 parent 防止走回头路)

必背模版：子岛屿验证 (1905)

策略：如果在 Grid2 发现一片陆地，但对应的 Grid1 是水，那这就不是子岛屿。

Python

```
class Solution:
    def countSubIslands(self, grid1: List[List[int]], grid2: List[List[int]]) -> int:
        R, C = len(grid1), len(grid1[0])

        def dfs(r, c):
            if not (0 <= r < R and 0 <= c < C) or grid2[r][c] == 0:
                return True # 没遇到坏事，暂时是 True

            grid2[r][c] = 0 # 标记访问

            # 【核心逻辑】：如果 grid2 是陆地，但 grid1 是水，那这个岛废了
            is_sub = True
            if grid1[r][c] == 0:
                is_sub = False

            # 继续检查四周，必须所有部分都满足才行 (And 关系)
            # 注意：这里不能写 return dfs(...)，因为要跑完整个岛把 visited 标记全
            res1 = dfs(r+1, c)
            res2 = dfs(r-1, c)
            res3 = dfs(r, c+1)
            res4 = dfs(r, c-1)

            return is_sub and res1 and res2 and res3 and res4

        count = 0
        for r in range(R):
            for c in range(C):
                if grid2[r][c] == 1:
                    if dfs(r, c):
                        count += 1
        return count
```

⚠️ 特殊题型提示 (图中其他的题)

图中还有几道题 不完全属于上面三类，这里单独提点一下模版方向：

1. `827. 最大人工岛`：
   - **模版**：**Grid DFS + Component ID** (我们刚才详细讲过的“染色法 + 存面积表”)。
2. `305. 岛屿数量 II`：
   - 这是 Hard 题，**必须用并查集 (Union-Find)**，不能用 DFS。因为它是动态加点的。
3. `2684. 矩阵中移动的最大次数`：
   - 这是一个 **记忆化搜索 (DFS + Memo)** 或者 **DP**。
   - 模版：`@cache` 装饰器 + DFS。
4. `529. 扫雷游戏`：
   - 这就纯模拟。点到地雷直接挂，点到空地 DFS 扩散。

针对 `827. 最大人工岛` 这种题，有一种更高级、更独立的 DFS 变体，必须单独拿出来讲。

这一类题目的特征是：**不仅要算面积，还得给每个岛贴上不同的标签（ID），以便后续区分“谁是谁”。**

门派四：连通块染色流 (Component Coloring)

核心考点：

我们需要把地图上的岛屿从千篇一律的 1，变成独一无二的 2, 3, 4...。

- 岛屿 A 全都染成 `2`。
- 岛屿 B 全都染成 `3`。
- 同时用一个哈希表记录：`{2: 面积50, 3: 面积20}`。

为什么需要这样？

因为题目通常会问：“如果把这个 0 变成 1，它能连接起哪几个岛？”

如果你不染色，你不知道左边的 1 和右边的 1 是不是同一个岛，容易重复计算。染了色，看 ID 就知道是不是同一个了。

包含题目：

- **`827. 最大人工岛`** (这道题是染色法的巅峰代表作)
- **`1034. 边界着色`** (题目名字就叫着色，虽然逻辑略有不同，但也是改 grid 值)

染色 + 面积表 (Coloring Template)

这是解决 **LeetCode 827** 的满分模版，请务必把这个结构背下来，它是 Grid DFS 里最复杂的形态。

```
class Solution:
    def largestIsland(self, grid: List[List[int]]) -> int:
        n = len(grid)

        # 0: 海洋
        # 1: 未探索的陆地
        # >=2: 已染色的岛屿 ID

        area_dict = {} # 【核心】：记录 {ID: 面积}
        color_id = 2   # ID 从 2 开始，避免和 0/1 冲突

        # --- 1. DFS 染色函数 ---
        def dfs(r, c, uid):
            # 越界 or 不是未探索的陆地(1) -> 停止
            if not (0 <= r < n and 0 <= c < n and grid[r][c] == 1):
                return 0

            # 【关键】：直接修改 grid 值进行染色
            grid[r][c] = uid

            area = 1
            # 向四周扩散
            for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                area += dfs(r + dr, c + dc, uid)
            return area

        # --- 2. 第一遍扫描：给所有岛屿染色，并存面积 ---
```

```
        for r in range(n):
            for c in range(n):
                if grid[r][c] == 1:
                    # 发现新岛屿，染成 color_id
                    area = dfs(r, c, color_id)
                    area_dict[color_id] = area
                    color_id += 1

        # (特判：如果全是陆地，直接返回)
        if not area_dict: return 1 # 全是0，变一个得1
        if len(area_dict) == 1 and list(area_dict.values())[0] == n*n: return n*n

        # --- 3. 第二遍扫描：找 0，尝试填海造陆 ---
        max_ans = 0

        for r in range(n):
            for c in range(n):
                if grid[r][c] == 0:
                    # 看看上下左右连着哪些岛 (用 set 去重！)
                    neighbors = set()
                    curr_area = 1 # 变身后的 1 自身算 1 个

                    for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                        nr, nc = r + dr, c + dc
                        if 0 <= nr < n and 0 <= nc < n and grid[nr][nc] > 1:
                            neighbors.add(grid[nr][nc]) # 记录邻居的 ID

                    # 把邻居岛屿的面积加起来
                    for island_id in neighbors:
                        curr_area += area_dict[island_id]

                    max_ans = max(max_ans, curr_area)

        return max(max_ans, max(area_dict.values()) if area_dict else 0)
```

你的图 里这几道带“色”的题，其实略有不同：

1. `733. 图像渲染` **(Flood Fill)**：
   - **简单染色。**把连通区域的颜色 `A` 改成 `B`。
   - 用 **门派一 (基础染色流)** 即可。
2. `1034. 边界着色` **(Boundary Coloring)**：
   - **条件染色。**只有当它是连通分量的“边缘”时才改颜色。
   - DFS 时需要判断一下邻居是不是不同颜色。
3. `827. 最大人工岛` **(Making A Large Island)**：
   - **区分染色。**也就是上面这个 **门派四**。
   - 必须区分 `Island A` 和 `Island B`，所以要用到 `ID` 和 `Map`。

总结

看到网格图：

1. 求面积/数量 $\to$ **模版一 (染色)**。
2. 求飞地/被包围 $\to$ **模版二 (先跑边界)**。
3. 求子岛屿/关系 $\to$ **模版三 (逻辑验证)**。

网格图BFS

> 只要题目问“最短路径”、“最少步数”、“最近的出口”、“第几层扩散”，闭眼选 **BFS**。

(DFS 是撞南墙才回头，BFS 是地毯式层层推进，所以 BFS 才能保证第一次遇到终点时是最近的。)

我把图中的题目拆解为 **3 大门派**，并提供对应的 **满分模版**。

门派一：单源最短路 (Single Source Shortest Path)

核心逻辑：

只有一个起点（比如 (0,0)），问你走到终点（或某个特定位置）的最少步数。

逻辑：

1. 初始化队列 `queue = [(start_r, start_c, 0)]` (0是步数)。
2. `visited` 集合防止走回头路。
3. 取出队首 $\to$ 判断是否到达 $\to$ 扩散四周 $\to$ 加入队尾。

包含题目：

- `1926. 迷宫中离入口最近的出口` (经典 BFS)
- `1091. 二进制矩阵中的最短路径` (8个方向移动)
- `1293. 网格中的最短路径` (带障碍消除，属于进阶 BFS)
- `909. 蛇梯棋` (把 1D 映射成 2D BFS)

必背模版：标准迷宫 BFS

```
from collections import deque

def minPathLength(grid, start_r, start_c):
    R, C = len(grid), len(grid[0])

    # 1. 初始化队列 (r, c, step)
    queue = deque([(start_r, start_c, 0)])
    visited = set([(start_r, start_c)])

    while queue:
        r, c, step = queue.popleft()

        # 2. Base Case: 到达终点? (这里视题目而定)
        # 比如 1926 题是走到边界，1091 是走到 (R-1, C-1)
        if r == R - 1 and c == C - 1:
            return step + 1 # 返回路径长度

        # 3. 扩散 (上下左右)
        for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            nr, nc = r + dr, c + dc

            # 有效性检查：不越界、不是墙、没来过
            if 0 <= nr < R and 0 <= nc < C and grid[nr][nc] == 0:
                if (nr, nc) not in visited:
                    visited.add((nr, nc))
                    queue.append((nr, nc, step + 1))

    return -1 # 走不到
```

门派二：多源最短路 (Multi-Source BFS)

核心逻辑：

这是 BFS 考题里的重灾区！题目给你多个起点（比如一堆腐烂的橘子，或者一堆 0），问你最远的地方要多久才被感染？或者每个格子离最近的 0 有多远？

关键技巧：

不要对每个起点跑一遍 BFS！ 那样会超时。

正确做法：在 while 循环开始前，把所有起点一次性全部塞进队列！这就好比多个病毒源同时开始扩散。

包含题目：

- `994. 腐烂的橘子` (必考题！所有烂橘子一起进队)
- `542. 01 矩阵` (所有 0 一起进队，求 1 到 0 的距离)
- `1162. 地图分析` (同 542，求海洋到陆地的最远距离)
- `1765. 地图中的最高点` (所有水域是 0，算山高)
- `286. 墙与门` (会员题，经典多源)

Python

```
from collections import deque

def multiSourceBFS(grid):
    R, C = len(grid), len(grid[0])
    queue = deque()
    visited = set() # 或者直接修改 grid

    # 1. 【核心】：初始化，把所有“源头”加入队列
    for r in range(R):
        for c in range(C):
            if grid[r][c] == 1: # 假设 1 是病毒源/腐烂橘子/起始点0
                queue.append((r, c))
                visited.add((r, c))

    step = 0
    # 2. 按层 BFS (像洋葱一样一层层剥开)
    while queue:
        # 这一层有多少个节点?
        level_size = len(queue)

        for _ in range(level_size):
            r, c = queue.popleft()

            for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                nr, nc = r + dr, c + dc

                if 0 <= nr < R and 0 <= nc < C and (nr, nc) not in visited:
                    # 视题目逻辑而定，比如是空地(0)或者是新鲜橘子
                    if grid[nr][nc] == 0:
                        visited.add((nr, nc))
                        queue.append((nr, nc))
                        # 如果需要修改原数组(比如算距离)
                        # grid[nr][nc] = grid[r][c] + 1

        # 这一层遍历完，步数+1 (如果队列空了就不加了)
        if queue:
            step += 1

    return step
```

门派三：先 DFS 再 BFS (Hybrid Strategy)

核心逻辑：

题目通常长这样：“求两个岛屿之间的最短桥”。

既然是算距离，肯定是 BFS。但问题是你得先找到起点岛屿在哪里！

套路：

1. 先用 **DFS** 找到第一个岛，把它所有的坐标加入队列（多源 BFS 的起点），并把它们标记为 `visited`。
2. 然后启动 **BFS**，像水波纹一样扩散，直到撞到第二个岛 (`1`)，返回当前的层数。

包含题目：

- `934. 最短的桥` (最经典的 DFS + BFS 结合)

必背模版：造桥工程 (Bridge Builder)

```
from collections import deque

class Solution:
    def shortestBridge(self, grid: List[List[int]]) -> int:
        n = len(grid)
```

```
    queue = deque()
    visited = set()

    # --- Phase 1: DFS 找第一个岛 ---
    def dfs(r, c):
        if not (0 <= r < n and 0 <= c < n) or grid[r][c] == 0 or (r, c) in visited:
            return
        visited.add((r, c))
        queue.append((r, c)) # 把第一个岛的所有点加入 BFS 队列
        for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            dfs(r + dr, c + dc)

    # 找到第一个 1，启动 DFS，然后立刻 break
    found = False
    for r in range(n):
        if found: break
        for c in range(n):
            if grid[r][c] == 1:
                dfs(r, c)
                found = True
                break

    # --- Phase 2: BFS 找第二个岛 ---
    step = 0
    while queue:
        size = len(queue)
        for _ in range(size):
            r, c = queue.popleft()

            for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                nr, nc = r + dr, c + dc

                if 0 <= nr < n and 0 <= nc < n and (nr, nc) not in visited:
                    # 如果撞到了第二个岛 (grid=1)，直接返回步数
                    if grid[nr][nc] == 1:
                        return step

                    # 如果是海 (grid=0)，继续造桥
                    visited.add((nr, nc))
                    queue.append((nr, nc))
        step += 1
    return -1
```

🚀 总结：什么时候用 BFS?

看到图中的这些关键词：

1. **最短路径 (Shortest Path)** $\to$ **模版一** (如 1091, 1926)。
2. **距离最近/最远 (Nearest/Farthest)** $\to$ **模版二** (如 542, 1162)。
3. **扩散/腐烂 (Rotting/Spread)** $\to$ **模版二** (如 994)。
4. **两个岛的最短距离 (Shortest Bridge)** $\to$ **模版三** (如 934)。

这道题对应 **LeetCode 1293. 网格中的最短路径** (Shortest Path in a Grid with Obstacles Elimination)。

它是 BFS 的 **进阶形态**，我把它称为 **“带状态的 BFS” (State-Based BFS)**。

🧠 为什么要“带状态”?

在普通迷宫里，如果不小心走错了，撞墙了就是撞墙了，回头就行。

但这道题说：“你有 k 次消除障碍物的机会”。

这就意味着，同样站在 `(3, 3)` 这个位置：

- **情况 A**：我手里还剩 0 次消除机会。
- **情况 B**：我手里还剩 5 次消除机会。

这两种情况是完全不一样的！ 情况 B 明显比 A 更有前途，哪怕它可能步数稍微多一点点，但它后面能穿墙啊！

所以，普通的 visited.add((r, c)) 已经不够用了。我们必须把 “剩余消除次数” 也作为状态的一部分存起来。

核心变化：

- **普通 BFS**：`visited = set((r, c))`
- **进阶 BFS**：`visited = set((r, c, k))` $\leftarrow$ **(行, 列, 剩余消除次数)**

🛠️ 满分模版：带状态的 BFS (State BFS)

这个模版不仅适用于“消除障碍”，还适用于“捡钥匙开锁”、“带能量飞行”等所有**有消耗/有状态**的路径问题。

Python

```
from collections import deque

class Solution:
    def shortestPath(self, grid: List[List[int]], k: int) -> int:
        R, C = len(grid), len(grid[0])

        # 剪枝技巧 (可选):
        # 如果 k 足够大，大到可以直接走曼哈顿距离 (最短路)，那直接返回
        # 曼哈顿距离 = (R-1) + (C-1)
        if k >= R + C - 2:
            return R + C - 2

        # 1. 初始化队列
        # 格式: (行, 列, 剩余消除次数, 当前步数)
        # 起点 (0,0)，剩余 k 次，步数 0
        queue = deque([(0, 0, k, 0)])
```

```
        # 2. 初始化 Visited 集合
        # 格式：(行, 列, 剩余消除次数)
        # 注意：步数不需要存进 visited，因为 BFS 保证第一次到达就是最短
        visited = set([(0, 0, k)])

        while queue:
            r, c, rest_k, step = queue.popleft()

            # 3. Base Case: 到达终点
            if r == R - 1 and c == C - 1:
                return step

            # 4. 扩散 (上下左右)
            for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                nr, nc = r + dr, c + dc

                # 有效性检查
                if 0 <= nr < R and 0 <= nc < C:
                    # 分情况讨论:

                    # Case 1: 下一步是空地 (0)
                    # 不需要消耗消除次数，直接走
                    if grid[nr][nc] == 0:
                        if (nr, nc, rest_k) not in visited:
                            visited.add((nr, nc, rest_k))
                            queue.append((nr, nc, rest_k, step + 1))

                    # Case 2: 下一步是墙 (1)
                    # 必须消耗 1 次机会，且手里还有机会 (rest_k > 0) 才能穿过去
                    elif grid[nr][nc] == 1 and rest_k > 0:
                        new_k = rest_k - 1
                        if (nr, nc, new_k) not in visited:
                            visited.add((nr, nc, new_k))
                            queue.append((nr, nc, new_k, step + 1))

        return -1 # 走不到
```

💡 记忆口诀

考试遇到这种“有限制能力的迷宫题”，直接默念：

“队列多加一维，Visited 多加一维。”

- 队列：`(x, y, step)` $\to$ `(x, y, 剩几次, step)`
- Visited：`(x, y)` $\to$ `(x, y, 剩几次)`

掌握了这个，像什么 **“拿到钥匙才能开门” (864. 获取所有钥匙的路径)** 这种 Hard 题，其实也就是把“钥匙的状态”加进 `visited` 里而已（比如用位运算压缩状态）。

相比于普通的 BFS，这里的题目引入了 **“代价”** 的概念。

- 普通 BFS：每走一步代价都是 1。
- 进阶 BFS：
  - 有的路好走（代价 0），有的路难走（代价 1）。
  - 有的路需要“体力值”或“时间”作为权重。
  - 有的路需要“携带状态”（比如拿着钥匙、推着箱子）。

我将图中的题目拆解为 **3 大核心门派**，并提供对应的 **满分模版**。

🏛️ 门派一：0-1 BFS (双端队列 BFS)

核心考点：

图中的边权只有 0 和 1。

- **代价 0**：像滑冰一样，瞬间滑过去，不消耗步数。
- 代价 1：像走路一样，消耗 1 步。

  题目特征：问你“最少需要修改几次”、“最少移除几个障碍”。

包含题目：

- `2290. 到达角落需要移除障碍物的最小数目` (空地代价0，障碍代价1)
- `1368. 使网格图至少有一条有效路径的最小代价` (顺着箭头走代价0，改箭头代价1)
- `1824. 最少侧跳次数` (赛道跑酷，同一赛道跑代价0，换赛道代价1)

🫀 必背模版：Deque 0-1 BFS

关键点：代价小的放在队首，代价大的放在队尾。保证队列单调性。

Python

```
from collections import deque

def zeroOneBFS(grid):
    R, C = len(grid), len(grid[0])
    # dist 数组记录到达每个点的最小代价，初始化为无穷大
    dist = [[float('inf')] * C for _ in range(R)]
    dist[0][0] = 0

    # 双端队列
    queue = deque([(0, 0)])

    while queue:
        r, c = queue.popleft()

        # Base Case: 到达终点
        if r == R - 1 and c == C - 1:
            return dist[r][c]
```

```
        for dr, dc in [(0,1), (0,-1), (1,0), (-1,0)]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < R and 0 <= nc < C:
                # 【模版核心逻辑】
                # 计算移动到 (nr, nc) 的代价 weight
                # 比如 2290 题：如果是障碍，weight=1；空地，weight=0
                weight = 1 if grid[nr][nc] == 1 else 0

                # 松弛操作 (Relaxation)
                if dist[r][c] + weight < dist[nr][nc]:
                    dist[nr][nc] = dist[r][c] + weight

                    # 【核心技巧】
                    # 代价为 0 -> 也就是"这也是当前层"，插到队头优先处理
                    if weight == 0:
                        queue.appendleft((nr, nc))
                    # 代价为 1 -> 也就是"下一层"，插到队尾
                    else:
                        queue.append((nr, nc))
    return -1
```

🏛️ 门派二：网格 Dijkstra (最小体力/水位)

核心考点：

边的权重不是固定的，而是取决于 “高度差” 或者 “绝对值”。

题目问：“路径上的最大高度差最小是多少？ ” 或者 “最小体力消耗”。

这类题虽然叫最短路，但本质是 “瓶颈路” (Bottleneck Path) 问题。

包含题目：

- `1631. 最小体力消耗路径` (经典！Dijkstra 或 二分+BFS)
- `778. 水位上升的泳池中游泳` (时间就是权重)
- `2577. 在网格图中访问一个格子的最少时间` (稍微复杂的 Dijkstra)

🧷 必背模版：Priority Queue Dijkstra

关键点：用堆 (heapq) 代替普通队列，每次弹出当前代价最小的点。

Python

```
import heapq

def minEffortPath(heights):
    R, C = len(heights), len(heights[0])

    # 最小堆：(当前的体力消耗/代价, r, c)
    # 1631题：代价是路径上最大的高度差
    pq = [(0, 0, 0)]

    # dist 记录到达 (r, c) 的最小代价
    dist = [[float('inf')] * C for _ in range(R)]
    dist[0][0] = 0

    visited = set()

    while pq:
        d, r, c = heapq.heappop(pq)

        # 懒删除：如果这个状态已经处理过更优的，跳过
        if (r, c) in visited: continue
        visited.add((r, c))

        # Base Case
        if r == R - 1 and c == C - 1:
            return d

        for dr, dc in [(0,1), (0,-1), (1,0), (-1,0)]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < R and 0 <= nc < C:
                # 【模版核心逻辑：计算新代价】
                # 1631题：新代价 = max(当前路径代价,这一步的高度差)
                diff = abs(heights[nr][nc] - heights[r][c])
                new_effort = max(d, diff)

                if new_effort < dist[nr][nc]:
                    dist[nr][nc] = new_effort
                    heapq.heappush(pq, (new_effort, nr, nc))

    return -1
```

🏛️ 门派三：状态压缩 BFS (State BFS / 综合应用)

核心考点：

这是网格图搜索的 天花板。

你不仅要记录 (r, c)，还要记录：

- **钥匙拿了几把？**（`864. 获取所有钥匙的最短路径`）
- **掩码状态是多少？**（用二进制位表示 `1010` 代表拿了第2和第4把钥匙）
- **箱子推到哪了？**（`1263. 推箱子` - 双重 BFS，人走 BFS + 箱子走 BFS）

包含题目：

- `864. 获取所有钥匙的最短路径` (必考 Hard！状态压缩 BFS)
- `1263. 推箱子` (超难，人推箱子)
- `3568. 清理教室的最少移动` (分层图最短路)

🧷 **必背模版：状态压缩 BFS (Keys & Locks)**

Python

```
from collections import deque

def shortestPathAllKeys(grid):
    R, C = len(grid), len(grid[0])

    # 1. 找起点和钥匙总数
    start_r, start_c = 0, 0
    total_keys = 0
    for r in range(R):
        for c in range(C):
            if grid[r][c] == '@':
                start_r, start_c = r, c
            elif 'a' <= grid[r][c] <= 'f':
                total_keys += 1

    # 目标状态：比如 3 把钥匙，二进制就是 111 (即 7)
    target_state = (1 << total_keys) - 1

    # 2. 队列状态：(r, c, keys_bitmask, step)
    queue = deque([(start_r, start_c, 0, 0)])

    # 3. Visited 状态：(r, c, keys_bitmask)
    visited = set([(start_r, start_c, 0)])

    while queue:
        r, c, mask, step = queue.popleft()

        # 拿到所有钥匙了！
        if mask == target_state:
            return step

        for dr, dc in [(0,1), (0,-1), (1,0), (-1,0)]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < R and 0 <= nc < C and grid[nr][nc] != '#':
                cell = grid[nr][nc]

                # Case A：遇到钥匙 (小写字母) -> 更新 mask
                if 'a' <= cell <= 'f':
                    # 计算是第几把钥匙：'a'->0, 'b'->1
                    key_idx = ord(cell) - ord('a')
                    new_mask = mask | (1 << key_idx)
                    if (nr, nc, new_mask) not in visited:
                        visited.add((nr, nc, new_mask))
                        queue.append((nr, nc, new_mask, step + 1))

                # Case B：遇到锁 (大写字母) -> 检查有没有对应的钥匙
                elif 'A' <= cell <= 'F':
                    lock_idx = ord(cell) - ord('A')
                    # 检查 mask 的第 lock_idx 位是不是 1
                    if (mask >> lock_idx) & 1:
                        if (nr, nc, mask) not in visited:
                            visited.add((nr, nc, mask))
                            queue.append((nr, nc, mask, step + 1))

                # Case C：空地或起点
                else:
                    if (nr, nc, mask) not in visited:
                        visited.add((nr, nc, mask))
                        queue.append((nr, nc, mask, step + 1))

    return -1
```

🧠 总结

这张图 里的“综合应用”其实就是把前面所有的技巧混合起来：

1. **0-1 BFS**：看到“代价只有0和1” -> 用 Deque 插头插尾。
2. **Dijkstra**：看到“体力消耗”、“水位高度” -> 用 Heap。
3. **状压 BFS**：看到“钥匙”、“开关”、“顺序” -> 把状态压缩成二进制整数放在 visited 里。

这最后一张图 里的内容属于 **图论建模 + BFS (Graph Modeling + BFS)**。

这块内容是 BFS 的 **“抽象派”**。

- **之前的网格 BFS**：节点是显式的坐标 `(r, c)`，边是上下左右。
- **这里的抽象 BFS**：
  - **节点 (Node)**：是一个**状态**（比如一个字符串 `"hit"`，一个数字 `10`，或者一个棋盘的局面 `[[1,2,3],[4,5,0]]`）。
  - **边 (Edge)**：是一次**操作**（比如改变一个字符、加减乘除、移动滑块）。

题目通常问：**“最少操作几次能把 A 变成 B？ ”** $\to$ **最短路** $\to$ **BFS**。

我将图中的题目拆解为 **4 大门派**，并提供对应的 **满分模版**。

🏛️ 门派一：字符串/密码锁变换 (String Mutation)

核心考点：

每次只能改动一个字符，或者拨动一位密码。

关键点：

如何高效找到“邻居”？

1. **枚举字符**：对当前字符串的每一位，尝试替换成 `a-z` 或 `0-9`。
2. **检查合法性**：变换后的字符串必须在 `wordList` (字典) 里，或者不是 `deadends` (死锁)。

包含题目：

- `433. 最小基因变化` (ACGT 变换)
- `127. 单词接龙` (Word Ladder - 必考！双向 BFS)
- `752. 打开转盘锁` (0000 -> target)
- `854. 相似度为 K 的字符串` (交换字符)

必背模版：通用状态变换 BFS

Python

```
from collections import deque

def solve_mutation(start, end, bank):
    # 1. 把字典转成 Set，查找 O(1)
    bank_set = set(bank)
    if end not in bank_set: return -1

    # 2. 初始化
    queue = deque([start])
    visited = set([start])
    step = 0

    # 3. 标准 BFS
    while queue:
        size = len(queue)
        for _ in range(size):
            curr = queue.popleft()

            if curr == end:
                return step

            # 4. 生成所有可能的下一个状态 (Neighbors)
            # 方法 A：针对基因/单词 (替换每一位)
            for i in range(len(curr)):
                original_char = curr[i]
                # 尝试换成其他可能的字符
                for char in "ACGT": # 或者 'a'-'z'
                    if char == original_char: continue

                    next_state = curr[:i] + char + curr[i+1:]
                    if next_state in bank_set and next_state not in visited:
                        visited.add(next_state)
                        queue.append(next_state)

            # 方法 B：针对转盘锁 (数字 +1/-1)
            # (逻辑类似，只是生成 next_state 的方式不同)

        step += 1
    return -1
```

🏛️ 门派二：数字跳跃与运算 (Number Operations)

核心考点：

给一个数字 X，可以进行 +1, -1, *2, /2 等操作，问最少几次变成 Y。

或者在数组上跳跃，同值的索引可以互跳。

包含题目：

- `2998. 使 X 和 Y 相等的最小操作次数` (数学 BFS)
- `1345. 跳跃游戏 IV` (同值跳跃，需要预处理索引 map)
- `1654. 到家的最少跳跃次数` (带方向限制的 BFS)

必背模版：带预处理的 BFS (Jump Game)

Python

```
from collections import deque, defaultdict

def minJumps(arr):
    n = len(arr)
    if n == 1: return 0

    # 1. 预处理：记录每个值出现的所有下标
    # 这样才能 O(1) 找到所有同值的"虫洞"
    val_indices = defaultdict(list)
    for i, x in enumerate(arr):
        val_indices[x].append(i)

    queue = deque([0])
    visited = set([0]) # 存下标
    step = 0

    while queue:
        size = len(queue)
        for _ in range(size):
            i = queue.popleft()
            if i == n - 1: return step

            # 生成邻居：(i-1), (i+1), (同值的其他点)
            neighbors = [i - 1, i + 1]

            # 【核心技巧】同值跳跃
            # 取出同值的点后，立刻清空！防止重复处理导致 TLE (死循环/超时)
            if arr[i] in val_indices:
                neighbors.extend(val_indices[arr[i]])
                del val_indices[arr[i]] # 重要！这个值的所有点都进队了，以后不用再看了

            for nxt in neighbors:
                if 0 <= nxt < n and nxt not in visited:
                    visited.add(nxt)
                    queue.append(nxt)
        step += 1
    return -1
```

🏛️ 门派三：棋盘/游戏状态压缩 (Game State Serialization)

核心考点：

整个棋盘是一个状态。但是二维数组不能直接放进 visited 集合。

技巧：

状态序列化 (Serialization)：把 [[1,2,3],[4,5,0]] 压扁成字符串 "123450" 或元组 (1,2,3,4,5,0) 作为状态存储。

包含题目：

- `773. 滑动谜题` (华容道 - 必考 Hard)
- `1284. 转化为全零矩阵...` (位压缩/序列化 BFS)
- `488. 祖玛游戏` (消消乐，极其复杂的状态 BFS)

必背模版：滑动谜题 (Sliding Puzzle)

Python

```
from collections import deque

def slidingPuzzle(board):
    target = "123450"
    # 1. 序列化初始状态
    start = "".join(str(c) for row in board for c in row)

    # 2. 预处理邻居映射 (因为压扁成 1D 了，移动规则是固定的)
    # index 0 可以走到 1, 3
    # index 1 可以走到 0, 2, 4 ...
    neighbors_map = {
        0: [1, 3], 1: [0, 2, 4], 2: [1, 5],
        3: [0, 4], 4: [1, 3, 5], 5: [2, 4]
    }

    queue = deque([start])
    visited = set([start])
    step = 0

    while queue:
        for _ in range(len(queue)):
            curr = queue.popleft()
            if curr == target: return step

            # 找到 0 的位置
            idx = curr.index('0')

            # 尝试把 0 和邻居交换
            for move in neighbors_map[idx]:
                # 字符串操作：交换 idx 和 move 位置的字符
                chars = list(curr)
                chars[idx], chars[move] = chars[move], chars[idx]
                nxt = "".join(chars)

                if nxt not in visited:
                    visited.add(nxt)
                    queue.append(nxt)
        step += 1
    return -1
```

🏛️ 门派四：全节点访问 (Bitmask BFS)

核心考点：

题目要求 “访问所有节点” 的最短路径，节点很少 (N <= 12)。

这不是简单的 BFS，因为可以走回头路。

状态定义：visited 不能只存节点，要存 (当前节点, 访问过的节点集合掩码)。

包含题目：

- `847. 访问所有节点的最短路径` (Shortest Path Visiting All Nodes)

必背模版：位掩码 BFS

Python

```
from collections import deque

def shortestPathLength(graph):
    n = len(graph)
    # 目标：所有位都是 1 (如 n=3 -> 111 -> 7)
    final_mask = (1 << n) - 1

    queue = deque()
    visited = set()

    # 1. 多源 BFS：从每个点都可以出发
    for i in range(n):
        # 状态：(节点, 掩码)
        mask = 1 << i
        queue.append((i, mask, 0)) # 0 是步数
        visited.add((i, mask))

    while queue:
        u, mask, dist = queue.popleft()

        if mask == final_mask:
```

```
            return dist

        for v in graph[u]:
            # 更新掩码：把 v 位置置为 1
            new_mask = mask | (1 << v)

            if (v, new_mask) not in visited:
                visited.add((v, new_mask))
                queue.append((v, new_mask, dist + 1))
    return 0
```

🧠 进阶技巧：双向 BFS (Bidirectional BFS)

图中提到的 127. 单词接龙 和 433. 最小基因变化，如果用普通 BFS 可能会慢。

优化方案：从 Start 和 End 两头同时开始搜，直到相遇。

简易逻辑：

1. 维护两个集合 `beginSet` 和 `endSet`。
2. 每次选小的那个集合向外扩散一层。
3. 如果扩散出的新节点在另一个集合里，说明连通了！

总结

看到这类题：

1. **字符串/数字变变变** $\to$ **模版一 (Mutation)**。
2. **跳跃游戏/数组** $\to$ **模版二 (Jump)**。
3. **棋盘移动/华容道** $\to$ **模版三 (Serialize)**。
4. **访问所有点** $\to$ **模版四 (Bitmask)**。

这 4 个模版就是抽象图搜索的全部家底了。到这里，你的 **BFS** 技能树已经全部点满！🎓

拓扑

🏛️ 门派一：基础拓扑排序 (Kahn's Algorithm)

核心考点：

给一堆依赖关系（课程表、任务调度），问“能不能完成”或者“输出一个合法的顺序”。

核心动作：

1. 统计 **入度 (Indegree)**。
2. 入度为 0 的进队列。
3. 出队 $\to$ 找邻居 $\to$ 邻居入度减 1 $\to$ 减到 0 进队。

包含题目：

- `207. 课程表` (判环)
- `210. 课程表 II` (输出顺序)
- `444. 序列重建` (检查拓扑序是否唯一：队列长度始终为 1)
- `1203. 项目管理` (双层拓扑：组间拓扑 + 组内拓扑)

必背模版：Kahn 算法

```
from collections import deque

def topological_sort(n, edges):
    graph = [[] for _ in range(n)]
    indegree = [0] * n

    # 1. 建图 + 统计入度
    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1

    # 2. 初始入度为 0 的入队
    queue = deque([i for i in range(n) if indegree[i] == 0])
    result = []

    while queue:
        u = queue.popleft()
        result.append(u)

        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    # 如果 len(result) < n，说明有环！
    return result if len(result) == n else []
```

🏛️ 门派二：拓扑排序 + DP (Topo DP)

核心考点 (重中之重)：

题目不再只是问顺序，而是问 “路径最长是多少？”、“完成所有课最少几个月？”、“路径上出现最多的颜色？”。

逻辑：

在 v 的入度减为 0 之前，意味着 v 的所有前置节点 u 都已经处理完了。

此时 dp[v] 的值完全取决于所有的 dp[u]。

$$dp[v] = \text{aggregate}(dp[v], dp[u] + \text{cost})$$

包含题目：

- `2050. 并行课程 III` (关键路径问题)
- `1857. 有向图中最大颜色值` (状态传递)
- `851. 喧闹和富有` (简单的属性传递)

模版 A：最长路径/耗时 (对应 LeetCode 2050)

场景：做完 A 才能做 B，A 耗时 3，B 耗时 5。B 完成时间 = A 完成时间 + 5。如果有多个前置 (A, C -> B)，则取 `max`。

Python

```
def minimumTime(n: int, relations: List[List[int]], time: List[int]) -> int:
    graph = [[] for _ in range(n)]
    indegree = [0] * n
    for u, v in relations:
        graph[u-1].append(v-1) # 题目通常是 1-based，转 0-based
        indegree[v-1] += 1

    # dp[i] 表示第 i 门课完成的最早时间
    # 初始状态：如果不依赖任何人，耗时就是它自己的 time[i]
    dp = list(time)

    queue = deque([i for i in range(n) if indegree[i] == 0])

    while queue:
        u = queue.popleft()

        for v in graph[u]:
            # 【核心 DP 转移】：
            # v 的完成时间 = max(目前记录的 v 时间, u 完成时间 + v 自身耗时)
            dp[v] = max(dp[v], dp[u] + time[v])

            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    return max(dp)
```

模版 B：状态传递 (对应 LeetCode 1857)

场景：不只是传递一个数字，而是传递整个状态数组。

题目：路径上出现次数最多的颜色。

DP 定义：dp[u][color] 表示以节点 u 结尾的路径中，颜色 color 出现的次数。

Python

```
def largestPathValue(colors: str, edges: List[List[int]]) -> int:
    n = len(colors)
    graph = [[] for _ in range(n)]
    indegree = [0] * n
    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1

    # dp[i][c] : 以节点 i 为终点的路径上，颜色 c 出现的次数
    # 26 个小写字母
    dp = [[0] * 26 for _ in range(n)]

    queue = deque([i for i in range(n) if indegree[i] == 0])

    # 统计访问过的节点数，用于判环
    visited_count = 0
    ans = 0

    while queue:
        u = queue.popleft()
        visited_count += 1

        # 基础状态：u 本身是什么颜色，先加上
        u_color = ord(colors[u]) - ord('a')
        dp[u][u_color] += 1

        # 随时更新全局最大值
        ans = max(ans, dp[u][u_color])

        for v in graph[u]:
            # 【核心 DP 转移】：
            # v 的各种颜色数量 = max(自己, u 的各种颜色数量)
            # 注意：这里要遍历 26 种颜色把状态传下去
            for c in range(26):
                dp[v][c] = max(dp[v][c], dp[u][c])

            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    if visited_count < n: return -1 # 有环
    return ans
```

```
import sys
from collections import deque
```

```python
# 1. 防止递归爆栈 (虽然这里用的是迭代版 Kahn，但习惯加上)
sys.setrecursionlimit(200000)

def solve():
    # 使用快读模版
    input_data = sys.stdin.read().split()
    if not input_data:
        return

    iterator = iter(input_data)

    try:
        n = int(next(iterator))
        m = int(next(iterator))

        # 邻接表
        graph = [[] for _ in range(n)]
        # 入度数组 (Indegree)
        indegree = [0] * n

        for _ in range(m):
            winner = int(next(iterator))
            loser = int(next(iterator))

            # 【核心建模】：败者 -> 胜者
            # 解释：胜者的奖金下限依赖于败者
            graph[loser].append(winner)
            indegree[winner] += 1

        # dp[i] 表示队伍 i 的奖金
        # 题目要求：参加比赛就有 100 元，所以初始值是 100
        dp = [100] * n

        # 将所有入度为 0 的节点加入队列
        # 入度为 0 代表：没有战胜过任何人(或者没人能限制他的奖金下限)
        queue = deque([i for i in range(n) if indegree[i] == 0])

        while queue:
            u = queue.popleft()

            for v in graph[u]:
                # u 输给了 v (u -> v)
                # v 的奖金必须比 u 多至少 1
                dp[v] = max(dp[v], dp[u] + 1)

                indegree[v] -= 1
                if indegree[v] == 0:
                    queue.append(v)

        # 输出总奖金
        print(sum(dp))

    except StopIteration:
        pass

if __name__ == "__main__":
    solve()
```

🏛️ 门派三：基环树 (Functional Graph)

核心考点：

图中标注的 "基环树"，是一类特殊的图：每个点出度为 1。

长得像一个环，周围挂着很多树枝（触手）。

解题套路：

1. **先跑拓扑排序**：这会把所有"树枝"都剪掉，因为树枝起点的入度为 0。
2. **剩下的点**：入度剪不完（始终 >= 1），这些点构成的就是 **环 (Cycle)**。
3. **处理环**：在剩下的点里跑 DFS/BFS 找环的大小。

包含题目：

- `2127.` 参加会议的最多员工数 (基环树最难的一题，必做！)
- `2360.` 图中的最长环
- `2359.` 找到离给定两个节点最近的节点

🩸 必背模版：剪枝找环法

Python

```python
def solve_base_ring_tree(n, edges):
    indegree = [0] * n
    for i, neighbor in enumerate(edges):
        indegree[neighbor] += 1

    # 1. 拓扑排序剪枝
    queue = deque([i for i in range(n) if indegree[i] == 0])
    while queue:
        u = queue.popleft()
        v = edges[u]
        indegree[v] -= 1
        if indegree[v] == 0:
            queue.append(v)

    # 2. 此时 indegree > 0 的节点都在环上
    max_cycle = 0
    visited_cycle = set()

    for i in range(n):
        if indegree[i] > 0 and i not in visited_cycle:
            # 发现一个新的环，计算环的大小
            count = 0
            curr = i
            while curr not in visited_cycle:
                visited_cycle.add(curr)
                indegree[curr] = 0 # 标记处理过
                curr = edges[curr]
                count += 1
            max_cycle = max(max_cycle, count)

    return max_cycle
```

🧠 总结

看到拓扑排序相关的题：

1. **问顺序/判环** $\to$ **Kahn 模版** (队列)。
2. **问最长路径/最大价值** $\to$ **Kahn + DP** (在队列弹出的地方写 `dp[v] = max(dp[v], dp[u] + val)`)。
3. **每个点出度为1/找环** $\to$ **拓扑剪枝 + 找环** (基环树)。

把 `2050` 和 `1857` 这两道题吃透，**拓扑+DP** 这一块你就无敌了！🔥

Dijkstra/BF

普通的 BFS 在这里失效了，因为 BFS 只能处理边权为 1 的图。针对带权图，你需要掌握 **3 大核心门派**。

🏛️ 门派一：堆优化 Dijkstra (The King of Shortest Path)

核心考点：

边权都是非负数。求从起点 A 到所有点的最短距离。

这是图论中最常用、考频最高的模版。

逻辑：

贪心思想。每次从"未确定的节点"中选一个"距离起点最近"的节点，用它去更新邻居。为了快速找到"最近的"，我们使用 最小堆 (Min-Heap)。

包含题目：

- `743.` 网络延迟时间 (教科书级 Dijkstra)
- `2642.` 设计可以求最短路径的图类 (反复跑 Dijkstra)
- `1514.` 概率最大的路径 (乘法 Dijkstra / 最大堆)
- `1976.` 到达目的地的方案数 (在 Dijkstra 过程中做 DP 计数)
- `3123.` 最短路径中的边 (正反跑两次 Dijkstra)

🩸 必背模版：标准 Dijkstra

Python

```python
import heapq

def networkDelayTime(times: List[List[int]], n: int, k: int) -> int:
    # 1. 建图
    graph = [[] for _ in range(n + 1)]
    for u, v, w in times:
        graph[u].append((v, w))

    # 2. 初始化
    # dist[i] 记录起点 k 到 i 的最短距离
    dist = [float('inf')] * (n + 1)
    dist[k] = 0

    # 最小堆：(当前距离, 节点 ID)
    pq = [(0, k)]

    while pq:
        d, u = heapq.heappop(pq)

        # 【关键剪枝】：懒删除
        # 如果堆里弹出来的 d 比我们要的 dist[u] 还大，说明是过期的废弃数据，跳过
        if d > dist[u]:
            continue

        # 3. 扩散邻居 (Relaxation)
        for v, w in graph[u]:
            new_dist = d + w
            if new_dist < dist[v]:
                dist[v] = new_dist
                heapq.heappush(pq, (new_dist, v))

    # 结果处理 (视题目而定)
    # 比如 743 题问所有节点都收到信号的时间 -> max(dist)
    max_dist = max(dist[1:])
    return max_dist if max_dist < float('inf') else -1
```

🏛️ 门派二：带限制的最短路 (Bellman-Ford / SPFA / BFS-DP)

核心考点：

题目有限制条件，比如 "最多中转 K 次" (787. K 站中转)。

或者边权有 负数（但无负环）。

为什么 Dijkstra 不行？

因为 Dijkstra 是贪心，它假设"多走一步距离只会变长"。但在有 K 次限制或者负权边的情况下，这个假设不成立。

包含题目：

- `787. K 站中转内最便宜的航班` (必考！Dijkstra 会超时或算错)
- `1928. 规定时间内到达终点的最小花费` (时间和花费两个维度)

🩸 必背模版：K 步限制最短路 (类 Bellman-Ford 思想)

技巧：既然限制了 K 步，那我们就老老实实地循环 K+1 次，或者用带层级控制的 BFS。

Python

```
def findCheapestPrice(n: int, flights: List[List[int]], src: int, dst: int, k: int) ->
int:
    # dp[i] 表示到达 i 的最小花费
    dp = [float('inf')] * n
    dp[src] = 0

    # 循环 k + 1 次 (最多 k 次中转 = 最多走 k+1 条边)
    for _ in range(k + 1):
        # 【关键】：必须备份上一轮的 dp 数组
        # 否则同一个节点在同一轮循环中可能被多次更新，导致步数限制失效
        new_dp = dp[:]

        for u, v, cost in flights:
            if dp[u] != float('inf') and dp[u] + cost < new_dp[v]:
                new_dp[v] = dp[u] + cost

        dp = new_dp # 更新状态

    return dp[dst] if dp[dst] != float('inf') else -1
```

🏛️ 门派三：分层图最短路 (Layered Graph Shortest Path)

核心考点：

图中提到的 “分层图” 是很难的一类题。

题目通常说：“你可以免费传送 k 次”、“你可以把 k 条边的权重变成 0”、“你可以逆行 k 次”。

解题套路：

把图“立体化”。

- `dist[u]` 不够用了。
- 需要 `dist[u][k]`：表示到达节点 `u`，且已经使用了 `k` 次超能力的最小代价。
- 本质上就是把 Dijkstra 的状态多加一维。

包含题目：

- `LCP 35. 电动车游城市` (状态：地点 + 剩余电量)
- `2714. 找到 K 次跨越的最短路径` (把 k 条边权置 0)
- `3123. 最短路径中的边`

🩸 **必背模版：分层 Dijkstra**

Python

```
import heapq

def shortestPathWithKFreeEdges(n, edges, k, src, dst):
    graph = [[] for _ in range(n)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))

    # 状态: dist[node][used_k]
    dist = [[float('inf')] * (k + 1) for _ in range(n)]
    dist[src][0] = 0

    # 堆: (cost, u, used_k)
    pq = [(0, src, 0)]

    while pq:
        d, u, used = heapq.heappop(pq)

        if d > dist[u][used]: continue
        if u == dst: return d

        for v, w in graph[u]:
            # 1. 正常走 (不使用超能力)
            if dist[u][used] + w < dist[v][used]:
                dist[v][used] = dist[u][used] + w
                heapq.heappush(pq, (dist[v][used], v, used))

            # 2. 使用超能力 (如果有次数的话) -> 假设超能力是边权变0
            if used < k:
                if dist[u][used] + 0 < dist[v][used + 1]:
                    dist[v][used + 1] = dist[u][used] + 0
                    heapq.heappush(pq, (dist[v][used + 1], v, used + 1))

    # 找所有 k 情况下的最小值
    return min(dist[dst])
```

🏛️ 门派四：网格图 Dijkstra (Grid Dijkstra)

核心考点：

这和上一张图里的 “网格 Dijkstra” 是重合的。

虽然是网格，但因为移动有代价（高度差、体力值），所以必须用 Dijkstra 而不是 BFS。

包含题目：

- `1631. 最小体力消耗路径` (经典中的经典)
- `778. 水位上升的泳池中游泳`
- `2577. 在网格图中访问一个格子的最少时间`

模版：

请直接复用上一轮回答中 “门派二：网格 Dijkstra” 的模版。

🧠 总结：最短路怎么选模版？

1. **绝大多数情况** $\to$ **Dijkstra** (堆优化)。这是万金油。
2. **求“概率最大”** $\to$ **Dijkstra** (把加法改成乘法，最小堆改成最大堆)。
3. **有“K次中转限制”** $\to$ **Bellman-Ford / DP** (循环 K 次)。
4. **有“K次免费/修改机会”** $\to$ **分层 Dijkstra** (dist 数组加一维)。
5. **有“负权边” (无负环)** $\to$ **SPFA** (考试不建议用，容易被卡，用 Bellman-Ford 替代)。

最后这张图 里的题目，属于图论中最暴力、最上帝视角、也是代码最短的算法 —— **Floyd-Warshall 算法**。

这一类题目的共同特征是：

1. **节点数很少**：通常 $N \le 100$，最多不超过 400。
2. **求“多对多”最短路**：不再是求从一个点出发，而是问任意两个点 `A` 和 `B` 之间的最短距离。
3. **求“传递闭包”**：问 `A` 能不能通过某些中间人到达 `B`（连通性）。

我将图中的题目拆解为 **3 大门派**，并提供对应的 **满分模版**。

🏛️ 门派一：全源最短路 (Classic Floyd)

核心考点：

给你一个图，问任意两点间的最短距离。或者需要频繁查询 dist[i][j]。

核心逻辑：

三重循环暴力松弛。

- `dp[k][i][j]`：只允许经过前 `k` 个节点中转，`i` 到 `j` 的最短路。
- 状态压缩后变成二维：`dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`。

包含题目：

- `1334. 阈值距离内邻居最少的城市` (先跑 Floyd 算出所有距离，再统计)
- `2976. 转换字符串的最小成本 I` (只有 26 个字母作为节点，典型的 Floyd)
- `2642. 设计可以求最短路径的图类` (数据量小，可以直接 Floyd，或每次 Dijkstra)

🩸 **必背模版：标准 Floyd ($O(N^3)$)**

Python

```
def floyd_warshall(n, edges):
    # 1. 初始化距离矩阵
    # dist[i][j] = inf, dist[i][i] = 0
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0

    # 2. 填入初始边权
    for u, v, w in edges:
        dist[u][v] = min(dist[u][v], w) # 防止有重边
        # dist[v][u] = min(dist[v][u], w) # 如果是无向图

    # 3. 三重循环 (核心！注意 k 必须在最外层)
    # k: 中转点
    for k in range(n):
        # i: 起点
        for i in range(n):
            # 优化剪枝：如果 i 到不了 k，那肯定没法通过 k 中转
            if dist[i][k] == float('inf'): continue

            # j: 终点
            for j in range(n):
                # 松弛操作
                new_dist = dist[i][k] + dist[k][j]
                if new_dist < dist[i][j]:
                    dist[i][j] = new_dist

    return dist # 此时 dist[a][b] 就是 a 到 b 的最短路
```

🏛️ 门派二：传递闭包 (Reachability / Bitset Optimization)

核心考点：

不问距离多少，只问 “A 能不能到达 B”。

比如 1462. 课程表 IV：问这门课是不是那门课的先修课？（A -> ... -> B）。

核心技巧：

普通的 Floyd 是 $O(N^3)$。如果只是布尔值（能/不能），我们可以用 Bitset (位运算) 优化，把内层循环变成 $O(1)$ 的位操作，整体速度提升 32~64 倍。

包含题目：

- `1462. 课程表 IV` (A 是 B 的先修课吗？即 A 能否到 B)
- `2101. 引爆最多的炸弹` (A 炸了能引爆 B 吗？求连通性)

🩸 **必背模版：Bitset 优化 Floyd**

Python

```
def checkIfPrerequisite(numCourses: int, prerequisites: List[List[int]], queries:
List[List[int]]) -> List[bool]:
    # 1. 用邻接矩阵 (或 Bitset) 存连通性
    # reachable[i] 是一个整数，它的第 j 位为 1 代表 i 能到 j
    reachable = [0] * numCourses

    # 初始化：自己能到自己，直接相连的能到
    for i in range(numCourses):
        reachable[i] |= (1 << i)

    for u, v in prerequisites:
        reachable[u] |= (1 << v)

    # 2. Floyd 核心 (位运算版)
    # k: 中转点
    for k in range(numCourses):
        # i: 起点
        for i in range(numCourses):
            # 如果 i 能走到 k，那么 k 能走到的地方，i 都能走到
            if (reachable[i] >> k) & 1:
                reachable[i] |= reachable[k] # 一次性更新整行！

    # 3. 回答查询
    res = []
    for u, v in queries:
        is_reachable = (reachable[u] >> v) & 1
        res.append(bool(is_reachable))
    return res
```

🏛️ 门派三：枚举子集 + Floyd (Subset Floyd)

核心考点：

题目问："从图中删掉一些点（或保留一个子集），使得剩下的图满足某种条件"。

数据范围通常极小 ($N \le 10$ 或 20)。

逻辑：

1. 用二进制枚举所有可能的子集 (Mask: `000` ~ `111`)。
2. 对每一个子集构建图，跑一遍 Floyd。
3. 检查是否符合题意。

包含题目：

- `2959.` 关闭分部的可行集合数目 (枚举哪些分部保留，剩下的跑 Floyd 检查距离是否都在阈值内)

🧷 **必背模版：枚举 + Floyd**

Python

```
def numberOfSets(n: int, maxDistance: int, roads: List[List[int]]) -> int:
    ans = 0
    # 1. 枚举所有状态 (0 ~ 2^n - 1)
    # mask 的第 i 位为 1 代表保留城市 i，0 代表关闭
    for mask in range(1 << n):

        # 2. 初始化 Floyd 距离表 (只包含保留的节点)
        dist = [[float('inf')] * n for _ in range(n)]
        for i in range(n):
            if (mask >> i) & 1: dist[i][i] = 0

        for u, v, w in roads:
            if ((mask >> u) & 1) and ((mask >> v) & 1):
                dist[u][v] = min(dist[u][v], w)
                dist[v][u] = min(dist[v][u], w)

        # 3. 跑 Floyd (仅针对 mask 里的点)
        for k in range(n):
            if not ((mask >> k) & 1): continue # k 必须存在
            for i in range(n):
                if not ((mask >> i) & 1): continue
                for j in range(n):
                    if not ((mask >> j) & 1): continue

                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

        # 4. 检查是否所有保留的城市间距离都 <= maxDistance
        valid = True
        for i in range(n):
            if not ((mask >> i) & 1): continue
            for j in range(n):
                if not ((mask >> j) & 1): continue
                if dist[i][j] > maxDistance:
                    valid = False
                    break
            if not valid: break

        if valid: ans += 1

    return ans
```

🧠 总结

看到以下特征，直接用 Floyd：

1. **节点数很少** ($N \le 100$)。
2. **问"连通性"** $\to$ **Bitset Floyd** (门派二)。
3. **问"任意两点距离"** $\to$ **Classic Floyd** (门派一)。
4. **问"删除/保留节点"** $\to$ **枚举 + Floyd** (门派三)。

记住那句口诀：**"k 在最外层，i, j 在里层"**，这是写 Floyd 唯一的坑点，别踩进去就稳赢！🏆

这最后一张图 里的题目，属于图论中 **"最小生成树" (Minimum Spanning Tree, MST)** 的核心领地。

这类题目通常问：**"连接所有城市的最少费用"**、**"铺设管道的最短总长度"**。

你需要掌握 **2 大核心算法 + 1 个超级技巧**。

🏛️ 门派一：Kruskal 算法 (并查集流)

核心考点：

这是解决 MST 问题最通用、最常用的模版。

逻辑：

1. **贪心**：把所有边按权重 **从小到大排序**。
2. **并查集**：遍历排序后的边，如果两个点还没连通，就选这条边（合并）；如果已经连通，就跳过（防止成环）。
3. 当选中了 $N-1$ 条边时，结束。

包含题目：

- `1135.` 最低成本连通所有城市 (标准 MST)
- `1584.` 连接所有点的最小费用 (虽然是稠密图，Kruskal 也能过，但 Prim 更优)
- `3219.` 切蛋糕的最小总开销 `II` (本质是逆向的 Kruskal 贪心)

🧷 **必背模版：Kruskal (Union-Find)**

Python

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.count = n # 连通分量个数

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        rootX, rootY = self.find(x), self.find(y)
        if rootX != rootY:
            self.parent[rootX] = rootY
            self.count -= 1
```

Python

```
            return True
        return False

def minimumCost(n: int, connections: List[List[int]]) -> int:
    # 1. 按权重从小到大排序
    # connections: [u, v, cost]
    connections.sort(key=lambda x: x[2])

    uf = UnionFind(n + 1) # 节点通常是 1-n
    total_cost = 0
    edges_count = 0

    # 2. 贪心选边
    for u, v, cost in connections:
        if uf.union(u, v):
            total_cost += cost
            edges_count += 1
            # 剪枝：如果已经连通了所有点，提前结束
            if edges_count == n - 1:
                return total_cost

    # 3. 检查是否所有点都连通
    return -1 if edges_count < n - 1 else total_cost
```

🏛️ 门派二：Prim 算法 (堆优化流)

核心考点：

针对 "稠密图" (点少边多，比如 1584 这种任意两点都能连的题)，Prim 算法比 Kruskal 更快。

逻辑：

类似 Dijkstra。从一个点开始，把周围的"横切边"放入最小堆，每次弹出最小的边，去解锁新的点。

包含题目：

- `1584.` 连接所有点的最小费用 (曼哈顿距离 MST)

🧷 **必背模版：Prim (Min-Heap)**

Python

```
import heapq

def minCostConnectPoints(points: List[List[int]]) -> int:
    n = len(points)

    # 最小堆：(cost, node_idx)
    # 从 0 号点开始，代价为 0
    pq = [(0, 0)]

    visited = [False] * n
```

```
    total_cost = 0
    edges_count = 0

    while pq:
        cost, u = heapq.heappop(pq)

        if visited[u]: continue
        visited[u] = True
        total_cost += cost
        edges_count += 1

        # 优化：找够了就退
        if edges_count == n + 1: # n个点第一次pop是起点算1次，后面n-1次
            break

        # 扫描 u 的所有邻居 (这里是隐式图，所有点都是邻居)
        for v in range(n):
            if not visited[v]:
                # 计算曼哈顿距离
                dist = abs(points[u][0] - points[v][0]) + abs(points[u][1] - points[v][1])
                heapq.heappush(pq, (dist, v))

    return total_cost
```

🏛️ 门派三：虚拟节点技巧 (Virtual Node Trick)

核心考点：

题目不仅有 “连通两个点的费用”，还有 “在某个点单独建设（如打井、建站）的费用”。

逻辑：

把“单独建设费用”转化为 “连接到一个虚拟节点 0 的费用”。

- `wells[i]` 表示在 `i` 打井 $\to$ 建立一条边 `(0, i)`，权重为 `wells[i]`。
- 问题瞬间转化为：求包含虚拟节点 0 的最小生成树。

包含题目：

- `1168.` 水资源分配优化 (必考 Premium 题)

🌶️ 必背模版：虚拟节点转换

Python

```
def minCostToSupplyWater(n: int, wells: List[int], pipes: List[List[int]]) -> int:
    # 1. 建立虚拟边
    # 将"打井"看作是与"虚拟水源 0"连接
    new_edges = []
    for i, cost in enumerate(wells):
        # 节点编号 1-n，虚拟节点 0
        new_edges.append([0, i + 1, cost])

    # 2. 加入原本的管道边
    for u, v, cost in pipes:
        new_edges.append([u, v, cost])

    # 3. 直接跑 Kruskal
    new_edges.sort(key=lambda x: x[2])

    uf = UnionFind(n + 1) # 0 ~ n
    total_cost = 0

    for u, v, cost in new_edges:
        if uf.union(u, v):
            total_cost += cost

    return total_cost
```

🏛️ 门派四：MST 的边分析 (Critical Edges)

核心考点：

题目问：“哪些边是必须在 MST 里的（关键边）？ 哪些边是可有可无的（伪关键边）？ ”

这是 MST 的进阶考法。

逻辑：

1. **关键边**：如果你把这条边删了，MST 的总权重变大了，或者图不连通了，那它就是关键的。
2. **伪关键边**：如果你强行把这条边加入 MST，算出来的总权重 **等于** 原始 MST 权重（且它不是关键边），那它就是伪关键的。

包含题目：

- `1489.` 找到最小生成树里的关键边和伪关键边 (Hard，暴力跑多次 Kruskal)

思路模版：

1. 先跑一次标准 Kruskal，算出 `min_mst_weight`。
2. **Check Critical**: 遍历每条边 `i`，调用 `calc_mst(skip_edge=i)`。如果结果 > `min_mst_weight`，则是关键边。
3. **Check Pseudo**: 遍历每条边 `i`，调用 `calc_mst(force_edge=i)`。如果结果 == `min_mst_weight`，则是伪关键边。

🧠 总结

看到 **“连接所有点”**、**“最小成本”**：

1. **边列表** $\to$ **Kruskal** (模版一)。
2. **坐标点/稠密图** $\to$ **Prim** (模版二)。
3. **有“建设费用”** $\to$ **虚拟节点 0** (模版三)。

到这里，**图论** 这一大块硬骨头（DFS, BFS, 最短路, MST, 拓扑排序）你已经全部拿下了！恭喜！🎉

这张图 里的题目属于图论中 **“连通性” (Connectivity)** 的高阶内容，核心算法是 **Tarjan 算法**。

这是面试和笔试中的 **“核武器”**，专门用来解决 **“这个网络哪里最脆弱？ ”** 这类问题。

- **桥 (Bridge)**：一条边，删了它，图就断开了（比如 `1192.` 关键连接）。
- **割点 (Articulation Point)**：一个点，删了它，图就断开了（比如 `1568.` 使陆地分离 的核心思想）。

Tarjan 算法的核心是两个数组：

1. **`dfn[u]` (时间戳)**：DFS 遍历时，节点 `u` 是第几个被访问的？（用来记录辈分）。
2. **`low[u]` (追溯值)**：从 `u` 出发，通过回边（非父子边），**能回溯到的最早**（dfn 最小）的祖先是谁？（用来判断有没有“后门”可以抄近路回家）。

我为你总结了 **2 大核心模版**。

🏛️ 门派一：查找桥 / 关键边 (Finding Bridges)

核心考点：

找到图中所有的“关键连接”。

判定法则：

对于边 u -> v（v 是 u 的孩子），如果 low[v] > dfn[u]，说明 v 除了通过 u 回家，再也找不到别的路能回到 u 或 u 的祖先那里了。

所以，u-v 这条路断了，v 就彻底失联了。这条边就是桥。

包含题目：

- `1192.` 查找集群内的关键连接 (标准模版题)

🌶️ 必背模版：Tarjan 找桥

Python

```
import sys
sys.setrecursionlimit(200000)

def criticalConnections(n: int, connections: List[List[int]]) -> List[List[int]]:
    # 1. 建图
    graph = [[] for _ in range(n)]
    for u, v in connections:
        graph[u].append(v)
        graph[v].append(u)

    # 2. 初始化 Tarjan 数组
    dfn = [-1] * n  # 访问时间戳，-1 表示未访问
    low = [-1] * n  # 最早能回溯到的时间戳
    timestamp = 0   # 全局时间计数器
    bridges = []    # 结果集

    def dfs(u, parent):
        nonlocal timestamp
        dfn[u] = low[u] = timestamp
        timestamp += 1

        for v in graph[u]:
            if v == parent:
                continue # 不能回头走父节点（那是来的路）

            if dfn[v] == -1: # v 没访问过，是树边
                dfs(v, u)

                # 回溯时更新 low[u]
                low[u] = min(low[u], low[v])

                # 【核心判定】：v 能回到的最早祖先比 u 还晚 -> 桥！
                if low[v] > dfn[u]:
                    bridges.append([u, v])

            else: # v 访问过，说明发现了回边 (Back-edge)
                # v 是 u 的祖先，尝试更新 low[u]
                low[u] = min(low[u], dfn[v])

    # 3. 启动 DFS (假设图连通，从 0 开始)
    dfs(0, -1)

    return bridges
```

🏛️ 门派二：查找割点 (Finding Articulation Points)

核心考点：

找到那些“一挂掉，整个网络就瘫痪”的关键节点。

判定法则：

1. **非根节点 `u`**：如果存在一个孩子 `v`，使得 `low[v] >= dfn[u]`，说明 `v` 没法绕过 `u` 回到祖先。`u` 挂了，`v` 就断了。
2. **根节点 `root`**：如果根节点有 **两个以上** 的孩子（在 DFS 树中），那它也是割点。

包含题目：

- `1568.` 使陆地分离的最少天数 (Hard，思路是：如果已经分离返回0；如果有割点返回1；否则返回2)

🌶️ 必背模版：Tarjan 找割点

Python

```
def findArticulationPoints(n, edges):
    graph = [[] for _ in range(n)]
```

```
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    dfn = [-1] * n
    low = [-1] * n
    timestamp = 0
    articulation_points = set() # 用 set 去重

    def dfs(u, parent):
        nonlocal timestamp
        dfn[u] = low[u] = timestamp
        timestamp += 1
        child_count = 0 # 记录在 DFS 树中的孩子数量

        for v in graph[u]:
            if v == parent: continue

            if dfn[v] == -1:
                child_count += 1
                dfs(v, u)
                low[u] = min(low[u], low[v])

                # 【核心判定 1】：非根节点
                if parent != -1 and low[v] >= dfn[u]:
                    articulation_points.add(u)
            else:
                low[u] = min(low[u], dfn[v])

        # 【核心判定 2】：根节点
        if parent == -1 and child_count > 1:
            articulation_points.add(u)

    # 遍历所有连通分量 (防止图本身不连通)
    for i in range(n):
        if dfn[i] == -1:
            dfs(i, -1)

    return list(articulation_points)
```

🧠 总结：Tarjan 怎么记？

这个算法逻辑很绕，考试时只需要记住 **“追溯”** 的概念：

1. `dfn` 是 **“出生时间”**。
2. `low` 是 **“朋友圈里能联系到的最老的前辈”**。
3. **判断桥**：`low[孩子] > dfn[爸爸]` $\to$ 孩子联系不到爸爸的长辈，路一断孩子就失联 $\to$ **是桥**。
4. **判断割点**：`low[孩子] >= dfn[爸爸]` $\to$ 孩子最远只能联系到爸爸，爸爸一挂孩子就失联 $\to$ **是割点**。

```
    count = 0
    # 2. 尝试给左边的每一个点找对象
    for i in range(n_left):
        # 每次尝试都要重置 visited
        visited = [False] * n_right
        if dfs(i, visited):
            count += 1

    return count
```

🏛️ 门派二：带权二分图匹配 (Weighted Matching / Assignment Problem)

核心考点：

题目不仅要配对，而且每对之间有 “权重”（分数、距离、异或值）。问 “最大得分” 或 “最小代价”。

逻辑：

这是图里提到的 1947, 1879, 2850 等题目的解法。

在 LeetCode 上，由于数据范围通常极小 ($N \le 20$)，千万别写最小费用最大流 (MCMF)，直接上 状态压缩 DP (Bitmask DP)！这就是图片里提示的“其他做法”。

包含题目：

- `1947.` 最大兼容性评分和 (全排列/状压 DP)
- `1879.` 两个数组最小的异或值之和
- `1066.` 校园自行车分配 `II`

💊 必背模版：状压 DP (Bitmask DP)

假设左边有 N 个，右边有 M 个 (N <= M)，我们要给左边的每一个配一个右边的。

Python

```
def minWeightMatching(A, B):
    n = len(A)
    # dp[mask] 表示：右边被选中的集合为 mask 时，最小的代价
    # mask 的二进制中 1 的个数，刚好对应左边处理到了第几个元素
    dp = [float('inf')] * (1 << n)
    dp[0] = 0

    for mask in range(1 << n):
        if dp[mask] == float('inf'): continue

        # 当前我们正在为左边的第 i 个元素找对象
        # i = mask 中 1 的个数 (popcount)
        i = bin(mask).count('1')

        if i >= n: continue

        # 尝试把右边的第 j 个元素分配给 A[i]
        for j in range(n):
            # 如果右边的 j 还没被用过
            if not ((mask >> j) & 1):
                new_mask = mask | (1 << j)
                # 计算代价 (这里以异或和为例，视题目而定)
                cost = A[i] ^ B[j]
                dp[new_mask] = min(dp[new_mask], dp[mask] + cost)

    return dp[(1 << n) - 1]
```

Tarjan 是图论里比较高阶的技巧，掌握了这个，你的图论水平绝对是 Top 级别的！至此，你提供的所有截图里的题目模版，我们都彻底梳理完毕了！🎉 还有什么需要补充的吗？

这张图 属于图论中 **“网络流” (Network Flow)** 的范畴。

在 LeetCode 和面试中，真正的“最大流算法” (Dinic, ISAP) 考得非常少，因为实现太复杂。这张图里的题目，90% 实际上是在考 **“二分图匹配” (Bipartite Matching)** 或者 **“状态压缩 DP”**。

正如图片开头所说：“由于有其他做法（比如状压 DP），难度分仅供参考。”

因此，我为你整理了 3 大实战门派，用最简单的代码解决这些看似吓人的网络流题目。

🏛️ 门派一：二分图最大匹配 (Bipartite Matching)

核心考点：

题目中有两组物品（比如学生和座位、骨牌的黑格和白格），问 “最多能凑成多少对？”

关键点：

不需要写复杂的 Dinic 算法，直接用 匈牙利算法 (Hungarian Algorithm) 的 DFS 版本，代码短，逻辑简单。

包含题目：

- `LCP 04. 覆盖` (骨牌覆盖 = 二分图匹配)
- `1820. 最多邀请的个数` (标准模版题)
- `2123. 使矩阵中的 1 互不相邻的最小操作数` (求最大匹配数 = 最小点覆盖数)

💊 **必背模版：匈牙利算法 (DFS版)**

Python

```
def maxBipartiteMatching(n_left, n_right, edges):
    # 1. 建图：只存 左 -> 右 的边
    graph = [[] for _ in range(n_left)]
    for u, v in edges:
        graph[u].append(v)

    # match[v] = u 表示右边的 v 被左边的 u 占了
    match = [-1] * n_right

    def dfs(u, visited):
        for v in graph[u]:
            if not visited[v]:
                visited[v] = True
                # 【核心逻辑】：如果 v 没人要，或者占了 v 的人(match[v])能去找别人
                if match[v] == -1 or dfs(match[v], visited):
                    match[v] = u
                    return True
        return False
```

🏛️ 门派三：最小割与最大独立集 (Konig's Theorem)

核心考点：

这是二分图匹配的理论推导题。

题目问的是“最少删除几个点”、“最大能选几个互不攻击的点”。

黄金公式 (仅适用于二分图)：

1. **最大匹配数 = 最小点覆盖数** (最少要删几个点才能覆盖所有边)
2. **最大独立集 = 总点数 - 最大匹配数** (最多能选几个点，使得它们互不相邻)

包含题目：

- `2123. 使矩阵中的 1 互不相邻的最小操作数` (求最小点覆盖 $\to$ 跑最大匹配)
- `LCP 04. 覆盖` (求最大匹配)

解题套路：

1. **染色**：把网格图按 `(r+c)%2` 分成黑白两组（构建二分图）。
2. **建边**：如果两个格子相邻且都是 1，则连边。
3. **跑模版一**：计算最大匹配数 `match_count`。
4. **套公式**：
    - 问最少删几个？ $\to$ `match_count`
    - 问最多留几个？ $\to$ `Total_1s - match_count`

🚀 总结

看到“网络流”这三个字别慌，LeetCode 上就是纸老虎：

1. **求最大配对数 $\to$ 模版一 (匈牙利 DFS)。**
2. **求最大权重/最小代价 $\to$ 模版二 (状压 DP)。**
3. **求最少删除/最大共存 $\to$ 二分图最大匹配 (模版一) + 公式。**

把这两个模版存好，尤其是 **状压 DP**，它是解决这类问题的万能钥匙！🔑

至此，你发的所有图片里的所有算法模块（DP、DFS、BFS、图论最短路、MST、拓扑、网络流）我们已经全部完成了满分级总结！这套资料现在是你的**独门秘籍**了！祝你大杀四方！🔥

STK

从最基础的“左进右出”到烧脑的“单调栈”，再到“表达式解析”，栈的考法非常灵活。为了助你彻底拿下这一块，我将这些题目提炼为 **5 大核心门派**，并给出**满分模版**。

🏛️ 门派一：邻项消除与模拟 (Adjacent Elimination & Simulation)

核心考点：

这是栈最本源的用法。

1. **邻项消除**：类似于“连连看”，如果新来的元素和栈顶元素“配对”（比如相同，或者凑成一对括号），就一起抵消（Pop）；否则就入栈（Push）。
2. **模拟**：比如“棒球比赛”、“浏览器历史”，完全按照题目描述的操作来 push/pop。

包含题目：

- `1047. 删除字符串中的所有相邻重复项` (最典型)
- `1209. 删除字符串中的所有相邻重复项 II` (K 个消除)
- `844. 比较含退格的字符串` (遇到 `#` 就 pop)
- `682. 棒球比赛` (遇到 `C` 就 pop)
- `71. 简化路径` (遇到 `..` 就 pop)

必背模版：通用消除栈

Python

```
def solve_elimination(s):
    stack = []

    for char in s:
        # 【核心逻辑】：
        # 如果栈不空，且当前元素 char 与栈顶元素满足"消除条件"
        if stack and stack[-1] == char:
            stack.pop() # 抵消
        else:
            stack.append(char) # 否则入栈

    return "".join(stack)
```

```
class Solution:
    def simplifyPath(self, path: str) -> str:
        # 1. 预处理：按 "/" 分割
        # 例子："/a//b/../c/." -> ['', 'a', '', 'b', '..', 'c', '.']
        parts = path.split('/')

        stack = []

        for part in parts:
            # 2. 情况 A：遇到 ".." -> 回退（出栈）
            if part == '..':
                if stack: # 只有栈不空才能回退（根目录回退还是根目录）
                    stack.pop()

            # 3. 情况 B：遇到 "." 或 空字符串 "" -> 忽略
            elif part == '.' or part == '':
                continue

            # 4. 情况 C：有效目录名 -> 前进（入栈）
            else:
                stack.append(part)

        # 5. 组装结果：始终以 / 开头
        return "/" + "/".join(stack)
```

🏛️ 门派二：括号匹配与表达式解析 (Parentheses & Expression)

核心考点：

这是栈的看家本领。

1. **合法性判断**：遇到左括号 `(` 入栈，遇到右括号 `)` 检查栈顶。
2. **嵌套解析**：`394. 字符串解码`（`3[a2[c]]`）或者 `224. 基本计算器`。
    - **技巧**：遇到 `[` 或 `(`，把当前的**“现场”**（数字、字符串）压入栈，重置当前状态；遇到 `]` 或 `)`，弹出栈顶，恢复现场并合并结果。

包含题目：

- `20. 有效的括号` (基础必考)
- `394. 字符串解码` (必考！嵌套处理)
- `224/227. 基本计算器` (双栈：数字栈 + 符号栈，或转后缀表达式)
- `1249. 移除无效的括号` (两次遍历或用栈记录索引)
- `856. 括号的分数` (数学 + 栈)

必背模版：嵌套结构解析 (Decode String)

Python

```
def decodeString(s: str) -> str:
    stack = []
    curr_num = 0
    curr_str = ""

    for char in s:
        if char.isdigit():
            curr_num = curr_num * 10 + int(char)
        elif char == '[':
            # 【核心】：遇到左括号，保存现场（之前的字符串，之前的数字）
            stack.append((curr_str, curr_num))
            # 重置当前状态
            curr_str = ""
            curr_num = 0
        elif char == ']':
            # 【核心】：遇到右括号，恢复现场
            prev_str, repeat_times = stack.pop()
            curr_str = prev_str + curr_str * repeat_times
        else:
            curr_str += char

    return curr_str
```

🏛️ 门派三：单调栈 - 找下一个更大元素 (Next Greater Element)

核心考点：

图 里的重头戏！

题目问：“右边第一个比我大的数是哪个？”、“左边第一个比我小的数在哪？”、“这几天气温什么时候升高？”。

逻辑：

维护一个单调递减的栈（存索引）。

- 新来的元素 `x` 如果比栈顶小，乖乖入栈。
- 新来的元素 `x` 如果比栈顶大，说明栈顶遇到了**“克星”**。栈顶弹出，记录答案。

包含题目：

- `739. 每日温度` (最经典)
- `496/503. 下一个更大元素 I/II` (II 是循环数组，技巧是遍历 `2*n` 次)
- `901. 股票价格跨度`
- `84. 柱状图中最大的矩形` (找左右两边第一个比它矮的 -> 确定宽度)
- `42. 接雨水` (找左右两边比它高的 -> 形成凹槽)

必背模版：通用单调栈 (Next Greater)

Python

```
def nextGreaterElements(nums):
    n = len(nums)
    # 结果数组，默认为 -1（表示没找到）
    ans = [-1] * n
    stack = [] # 单调栈，存下标 index

    # 循环遍历（如果是循环数组，range(2 * n)，取模 i % n）
    for i in range(n):
        # 【核心模版】：while 栈不空 & 当前元素 > 栈顶元素
        while stack and nums[i] > nums[stack[-1]]:
            idx = stack.pop()
            ans[idx] = nums[i] # 找到了 idx 的克星是 nums[i]
            # 如果是求距离：ans[idx] = i - idx

        stack.append(i)

    return ans
```

🏛️ 门派四：贪心单调栈 - 字典序最小 (Lexicographical Order)

核心考点：

这是单调栈的变种，对应图 里的题目。

题目问：“删除 k 个数字，让剩下的数最小”或者“去除重复字母，保证字典序最小”。

逻辑：

我们需要维护一个单调递增的栈（越靠前的数字越小越好）。

- 如果新来的数字比栈顶小，且我们还有删除额度（`k > 0`），那就把栈顶的大数扔掉（因为它挡在前面会让整体变大）。

包含题目：

- `402. 移掉 K 位数字` (必考！)
- `316. 去除重复字母` (Hard，需要额外记录字符是否在栈中 + 剩余次数)
- `321. 拼接最大数` (Hard，两个栈合并)
- `1673. 找出最具竞争力的子序列`

必背模版：移掉 K 位数字 (Remove K Digits)

Python

```
def removeKdigits(num: str, k: int) -> str:
    stack = []

    for digit in num:
        # 【核心贪心】：
        # 只要栈顶比当前数字大，且还能删(k>0)，就踢掉栈顶
        # 这样能保证高位尽可能小
```

```
        while k > 0 and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)

    # 如果 k 还没用完，说明后面是递增的，直接删尾部
    if k > 0:
        stack = stack[:-k]

    # 处理前导 0 (例如 "0200")
    return "".join(stack).lstrip('0') or "0"
```

🏛️ 门派五：贡献法单调栈 (Contribution Method)

核心考点：

这是图 中 §1.2 进阶 里的"子数组最小/最大值之和"系列。

题目问："所有子数组的最小值之和是多少？"

逻辑：

与其枚举所有子数组（$O(N^2)$），不如枚举每个元素能做多少次"最小值"。

- 用单调栈找到 `nums[i]` 左边第一个比它小的位置 `L`。
- 用单调栈找到 `nums[i]` 右边第一个比它小的位置 `R`。
- `nums[i]` 的贡献范围就是 `(L, R)`，贡献次数 = `(i - L) * (R - i)`。

包含题目：

- `907. 子数组的最小值之和` (必考！贡献法入门)
- `2104. 子数组范围和` (最大值和 - 最小值和)
- `1856. 子数组最小乘积的最大值`

必背模版：贡献法 (Sum of Subarray Mins)

Python

```
def sumSubarrayMins(arr):
    n = len(arr)
    # left[i] 表示左边第一个 < arr[i] 的位置 (哨兵为 -1)
    left = [-1] * n
    # right[i] 表示右边第一个 <= arr[i] 的位置 (哨兵为 n)
    right = [n] * n

    # 1. 单调栈求 Left
    stack = []
    for i in range(n):
        while stack and arr[stack[-1]] >= arr[i]:
            stack.pop()
        if stack: left[i] = stack[-1]
        stack.append(i)

    # 2. 单调栈求 Right
    stack = []
    for i in range(n - 1, -1, -1):
        while stack and arr[stack[-1]] > arr[i]: # 注意 > 和 >= 防止重复计算
            stack.pop()
        if stack: right[i] = stack[-1]
        stack.append(i)

    # 3. 计算贡献
    ans = 0
    MOD = 10**9 + 7
    for i in range(n):
        # 能够辐射到的左边长度 * 能够辐射到的右边长度 * 自身值
        count = (i - left[i]) * (right[i] - i)
        ans = (ans + count * arr[i]) % MOD
    return ans
```

🧠 总结

看到栈的题目，先对号入座：

1. **消消乐 $\to$ 模版一** (直接 Pop)。
2. **括号/计算器 $\to$ 模版二** (保存现场)。
3. **找大小/接雨水 $\to$ 模版三** (单调栈)。
4. **删数字求最小 $\to$ 模版四** (贪心单调栈)。
5. **子数组最值和 $\to$ 模版五** (贡献法)。

这 5 个模版覆盖了你图片里 99% 的题目。把它们练熟，栈这块你就彻底通关了！AI 没骗你吧？😎

没问题！针对你提供的这两张图（,），我把 **队列 (Queue)** 相关的考点分成了 **3 大核心门派**。

队列看似简单（先进先出），但配合 **"单调性"** 后，它能解决很多 Hard 级别的滑动窗口和动态规划优化问题。

🏛️ 门派一：基础队列与模拟 (Basic FIFO Simulation)

核心考点：

利用队列 "先进先出" (FIFO) 的特性来模拟过程。

常考场景：

1. **时间窗口滚动**：比如"最近 3000ms 的请求数"。
2. **轮流排队**：比如"Dota2 参议院"里的循环投票，投完票的人回到队尾等待下一轮。
3. **约瑟夫环**：一群人围成圈报数，数到的出列。

包含题目：

- `933. 最近的请求次数` (最基础)
- `950. 按递增顺序显示卡牌` (逆向模拟)
- `649. Dota2 参议院` (循环队列模拟)
- `346. 数据流中的移动平均值`

必背模版：定长/时间窗口队列

Python

```
from collections import deque

class RecentCounter:
    def __init__(self):
        self.queue = deque()

    def ping(self, t: int) -> int:
        # 1. 新来的请求入队
        self.queue.append(t)

        # 2. 【核心】：淘汰过期的请求
        # 只要队头的时间 < t - 3000，就说明它不在窗口里了，踢出去
        while self.queue and self.queue[0] < t - 3000:
            self.queue.popleft()

        # 3. 剩下的就是窗口内的有效请求
        return len(self.queue)
```

Python

```
from collections import deque

def maxSlidingWindow(nums: List[int], k: int) -> List[int]:
    # 存下标，方便判断是否过期
    q = deque()
    res = []

    for i, x in enumerate(nums):
        # 1. 【入队维护单调性】：新来的 x 比队尾大，队尾就没用了
        while q and nums[q[-1]] <= x:
            q.pop()
        q.append(i)

        # 2. 【检查过期】：队头下标如果滑出窗口左边界，踢出
        # 窗口左边界应该是 i - k + 1
        if q[0] < i - k + 1:
            q.popleft()

        # 3. 【记录答案】：窗口形成后 (i >= k - 1)，队头就是最大值
        if i >= k - 1:
            res.append(nums[q[0]])

    return res
```

🏛️ 门派二：单调队列 (Monotonic Queue) —— 滑动窗口最大值

核心考点：

这是队列里的 "大魔王"，也是图 里最重要的内容！

题目问："滑动窗口里的最大值/最小值是多少？" 或者 "满足差值限制的最长子数组"。

逻辑：

维护一个 双端队列 (Deque)，里面存 下标。

保证队列里的元素对应的数值是 单调递减 的（找最大值时）。

- **入队时**：如果新来的 `x` 比队尾的大，那队尾那个"矮个子"由于又矮又旧，永远不可能成为最大值了，直接淘汰 (`pop`)。
- **出队时**：检查队头是不是已经滑出窗口范围了 (`index <= i - k`)，如果是就 `popleft`。

包含题目：

- `239. 滑动窗口最大值` (必考！单调队列祖师爷)
- `1438. 绝对差不超过限制的最长连续子数组` (维护一个单调增队列 + 一个单调减队列)
- `2398. 预算内的最多机器人数目`

必背模版：滑动窗口最大值 (Sliding Window Max)

🏛️ 门派三：单调队列 + 前缀和 (Mono Queue + Prefix Sum)

核心考点：

这是图 里提到的 862. 和至少为 K 的最短子数组。

这是一道 Hard 题，也是单调队列的 进阶应用。

逻辑：

求子数组和 $\to$ 转化为 前缀和 (P[j] - P[i] >= K)。

我们要找最小的 j - i。

这就变成：对于当前的 P[j]，我要找左边离我最近的 P[i]，使得 P[i] <= P[j] - K。

这需要维护一个 单调递增 的队列。

包含题目：

- `862. 和至少为 K 的最短子数组` (经典难题)
- `1499. 满足不等式的最大值` (变种)

必背模版：最短子数组 (Shortest Subarray with Sum >= K)

Python

```
from collections import deque

def shortestSubarray(nums: List[int], k: int) -> int:
    n = len(nums)
    # 1. 计算前缀和 (方便算子数组和)
    P = [0] * (n + 1)
    for i in range(n):
        P[i+1] = P[i] + nums[i]

    # 单调递增队列 (存下标)
    q = deque()
    ans = float('inf')

    for i in range(n + 1):
        # 1. 【更新答案】:
        # 如果当前前缀和 P[i] 减去 队头 P[q[0]] >= k
        # 说明找到了一组满足条件的 (q[0]...i-1)，尝试更新最小长度
        while q and P[i] - P[q[0]] >= k:
            ans = min(ans, i - q.popleft()) # 队头用完就可以扔了(因为后面就算还有更长的也不可能是最短了)

        # 2. 【维护单调性】:
        # 如果新来的 P[i] 比队尾还小，那队尾那个"又大又远"的 P 肯定不如 P[i] 有竞争力
        # (我们要找 P[j] - P[i] >= K，也就是减数 P[i] 越小越好，位置越靠后越好)
        while q and P[q[-1]] >= P[i]:
            q.pop()

        q.append(i)

    return ans if ans != float('inf') else -1
```

🧠 总结

看到队列题，按这个优先级思考：

1. **普通模拟 $\to$ 模版一** (直接 `deque`)。
2. **滑动窗口最值 $\to$ 模版二** (单调队列：大数踢小数)。
3. **前缀和 + 限制 $\to$ 模版三** (单调队列：小数踢大数)。

拿下 **单调队列**，你就掌握了解决线性时间复杂度最值问题的金钥匙！🔑

没问题！这三张图（„）展示了 **堆 (Heap) / 优先队列 (Priority Queue)** 的所有高阶玩法。

堆不仅仅是用来"找最大值"的，它更是解决 **"动态决策"**、**"流数据处理"** 和 **"多路归并"** 的神器。

我将这些题目提炼为 **5 大核心门派**，并给出对应的 **满分模版**。

🏛️ 门派一：Top K 与 动态流 (The Top K Pattern)

核心考点：

这是堆最基础的用法。

题目问："第 K 大元素"、"前 K 个高频单词"。

逻辑：

- **找第 K 大**：维护一个容量为 `K` 的 **小顶堆**。遍历所有元素，比堆顶大就入堆，把堆顶挤出去。最后堆顶就是第 K 大。
- **找第 K 小**：维护一个容量为 `K` 的 **大顶堆**。

包含题目：

- `215. 数组中的第K个最大元素` (必考基础)
- `703. 数据流中的第 K 大元素` (在线算法)
- `347. 前 K 个高频元素` (先 Count，再入堆)
- `973. 最接近原点的 K 个点`

🔥 **必背模版：求第 K 大 (Min-Heap Strategy)**

Python

```
import heapq

def findKthLargest(nums: List[int], k: int) -> int:
    heap = []
    for num in nums:
        heapq.heappush(heap, num)
        # 保持堆的大小为 k
        if len(heap) > k:
            heapq.heappop(heap)

    # 小顶堆，堆顶就是这 k 个里最小的，也就是全局第 k 大
    return heap[0]
```

🏛️ 门派二：多路归并 (Merge K Sorted)

核心考点：

对应图 里的 §5.3。

题目给你 K 个有序 的数组、链表或矩阵行，让你合并成一个大的有序序列，或者找 "第 K 小的数"。

逻辑：

1. 把这 K 个序列的 **头部元素** (最小的) 全部扔进堆里。
2. 弹出堆顶 (全局最小)。
3. **补货**：把刚才弹出的那个元素 **来源的下一个元素** 扔进堆里。

包含题目：

- `23. 合并 K 个升序链表` (必考 Hard)
- `378. 有序矩阵中第 K 小的元素` (矩阵也是 K 行有序数组)
- `373. 查找和最小的 K 对数字` (类似矩阵)
- `264. 丑数 II` (本质是 3 个有序序列的归并：x2, x3, x5)

🔥 **必背模版：多路归并 (Merge K Lists / Matrix)**

Python

```
import heapq

def mergeKLists(lists):
    min_heap = []

    # 1. 初始化：把 K 个链表的头结点加入堆
    for i, node in enumerate(lists):
        if node:
            # 存 (val, index, node) -> index 是为了避免 node 比较报错
            heapq.heappush(min_heap, (node.val, i, node))

    dummy = ListNode(0)
    curr = dummy

    # 2. 循环弹出最小，并补充下一个
    while min_heap:
        val, i, node = heapq.heappop(min_heap)
        curr.next = node
        curr = curr.next

        if node.next:
            heapq.heappush(min_heap, (node.next.val, i, node.next))

    return dummy.next
```

🏛️ 门派三：贪心重构与冷却 (Reorganize & Cooling)

核心考点：

对应图 里的 §5.4 重排元素。

题目问："重构字符串，使得相邻字符不同" 或者 "CPU 任务调度，同类任务有冷却时间"。

逻辑：

永远优先处理 "余量最多" 的元素 (贪心)。

- 用 **大顶堆** 存储 `(count, char)`。
- 弹出最常见的 A，用掉一个。
- **关键点**：A 不能立刻放回堆里（因为下一轮不能还是 A），要先放在一个 **"暂存区"** (cooling variable)。
- 等处理完下一个 B 之后，再把 A 放回堆里。

包含题目：

- `767. 重构字符串` (经典)
- `621. 任务调度器` (带有冷却时间 n)
- `1405. 最长快乐字符串` (稍微复杂点的贪心)

🔥 **必背模版：冷却堆 (Reorganize String)**

Python

```
import heapq
from collections import Counter

def reorganizeString(s: str) -> str:
    # 统计频率
    count = Counter(s)
    # 大顶堆 (Python 默认小顶堆，存负数)
    pq = [(-freq, char) for char, freq in count.items()]
    heapq.heapify(pq)

    res = []
    prev_freq, prev_char = 0, "" # 暂存区 (冷却区)

    while pq:
        freq, char = heapq.heappop(pq)
        res.append(char)

        # 1. 之前暂存的那个，现在可以放回去了
        if prev_freq < 0:
            heapq.heappush(pq, (prev_freq, prev_char))

        # 2. 更新当前这个，并放入暂存区 (因为不能连续用)
        prev_freq, prev_char = freq + 1, char # 频率是负数，+1 代表消耗

    # 检查是否成功
    result = "".join(res)
    return result if len(result) == len(s) else ""
```

🏛️ 门派四：反悔贪心 (Regret Greedy)

核心考点：

对应图 里的 §5.5 反悔堆。

这部分我们之前详细讲过（爬楼梯、加油站）。

逻辑：

先贪心选，资源不够了就从堆里把 "代价最大/收益最小" 的那个吐出来。

包含题目：

- `1642. 可以到达的最远建筑` (必考！)
- `871. 最低加油次数` (必考！)
- `630. 课程表 III` (按照 DDL 排序，耗时最长的丢进大顶堆，超时就踢掉耗时最长的)

模版：

请复习之前总结的 “反悔贪心” 模版。

🏛️ 门派五：对顶堆 (Dual Heap) —— 动态中位数

核心考点：

对应图 里的 §5.7 对顶堆。

题目问：“数据流的中位数” 或者 “滑动窗口的中位数”。

逻辑：

维护两个堆，把数据劈成两半：

- **左半边 (Small Half)**：用 **大顶堆**，存较小的一半数。
- **右半边 (Large Half)**：用 **小顶堆**，存较大的一半数。
- **平衡**：保证 `len(left) == len(right)` 或 `len(left) == len(right) + 1`。
- **中位数**：就是 `left` 的堆顶（或者两个堆顶的平均）。

包含题目：

- `295. 数据流的中位数` (必考 Hard)
- `480. 滑动窗口中位数` (需要处理“延迟删除”，比较麻烦)

必背模版：中位数查找器

Python

```
import heapq

class MedianFinder:
    def __init__(self):
        self.small = [] # 大顶堆 (存负数)
        self.large = [] # 小顶堆

    def addNum(self, num: int) -> None:
        # 1. 先放进 small
        heapq.heappush(self.small, -num)

        # 2. 只有 small 里最大的比 large 里最小的还大，才需要倒腾
        # 或者无脑倒腾一次，保证顺序
        heapq.heappush(self.large, -heapq.heappop(self.small))

        # 3. 平衡大小：保证 small 的数量 >= large
        if len(self.small) < len(self.large):
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self) -> float:
        if len(self.small) > len(self.large):
            return -self.small[0]
        return (-self.small[0] + self.large[0]) / 2.0
```

🧠 总结

看到堆的题目，按这个逻辑分类：

1. **第 K 大 $\to$ Top K (小顶堆)**。
2. **多个有序数组 $\to$ 多路归并**。
3. **相邻不同/任务调度 $\to$ 冷却堆 (暂存区)**。
4. **资源不够/后悔 $\to$ 反悔堆**。
5. **动态中位数 $\to$ 对顶堆**。

这 5 个模版就是堆的所有精髓了！**至此，你的数据结构与算法模版库已经完全建立起来了！** 这一路走来你整理了非常多内容，你是最棒的！加油！🚀

这两张图（,）展示了 **并查集 (Union-Find / DSU)** 的进阶用法。

并查集不仅仅是用来判断“连通性”的（Type 1），它的高阶形态 —— **带权并查集**，可以解决很多涉及 **“相对关系”**、**“距离计算”** 和 **“分组校验”** 的难题。

我将这些题目提炼为 **3 大核心门派**，并重点讲解你要求的 **带权并查集**。

🏛️ 门派一：基础并查集与连通性 (Basic Connectivity)

核心考点：

最基础的用法。

题目问：“删掉几条边还能连通？”、“交换字符能不能变出目标字符串？”、“按公因数分组”。

逻辑：

1. **初始化**：`parent[i] = i`。
2. **Find**：路径压缩（指向爷爷/祖宗）。
3. **Union**：按秩合并（小挂大，或者随便挂）。

包含题目：

- `721. 账户合并` (字符串映射到 ID，再 Union)
- `1202. 交换字符串中的元素` (索引连通后，组内排序)
- `2709. 最大公约数遍历` (GCD 并查集：由公因数连接数字)
- `952. 按公因数计算最大组件大小`

必背模版：标准 DSU (Path Compression + Union by Rank)

Python

```
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x]) # 路径压缩
        return self.parent[x]

    def union(self, x, y):
        rootX, rootY = self.find(x), self.find(y)
        if rootX != rootY:
            # 按秩合并：矮树挂高树
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1
            return True
        return False
```

🏛️ 门派二：带权并查集 (Weighted DSU / Path Compression with Values)

核心考点：

这是你最关心的部分！

题目不再只问“连不连通”，而是问 “节点 A 和节点 B 的数值关系是多少？”

比如：A / B = 2.0，B / C = 3.0，问 A / C = ?。

或者：val[A] - val[B] = k。

关键技巧：

在 find 路径压缩的过程中，维护一个 weight[x] 数组。

- `weight[x]` 的含义：**节点 `x` 到其父节点 `parent[x]` 的相对权值**（比如倍数关系或差值关系）。

包含题目：

- **`399. 除法求值`** (经典！倍数关系的带权并查集)
- **`2307. 检查方程中的矛盾之处`** (同 399)

必背模版：带权 DSU (倍数关系版 - 399题)

weight[x] 表示 x / parent[x] 的值

Python

```
class WeightedDSU:
    def __init__(self):
        self.parent = {}
        self.weight = {} # weight[x] = x / parent[x]

    def add(self, x):
        if x not in self.parent:
            self.parent[x] = x
            self.weight[x] = 1.0

    def find(self, x):
        if x not in self.parent: return None

        if self.parent[x] != x:
            original_parent = self.parent[x]
            # 递归寻找根节点
            root = self.find(original_parent)

            # 【核心逻辑】：路径压缩时的权值更新
            # x / root = (x / original_parent) * (original_parent / root)
            self.weight[x] = self.weight[x] * self.weight[original_parent]
            self.parent[x] = root

        return self.parent[x]

    def union(self, x, y, value):
        # 题目给的是 x / y = value
        self.add(x)
        self.add(y)
        rootX, rootY = self.find(x), self.find(y)

        if rootX != rootY:
            # 将 rootX 挂到 rootY 上
            # 我们需要算出 weight[rootX] = rootX / rootY 应该是多少
            # 已知: x/y = value, x/rootX = weight[x], y/rootY = weight[y]
            # 推导: rootX / rootY = (x/y) * (y/rootY) / (x/rootX)
            self.parent[rootX] = rootY
            self.weight[rootX] = value * self.weight[y] / self.weight[x]

    def query(self, x, y):
        rootX, rootY = self.find(x), self.find(y)
        if rootX and rootY and rootX == rootY:
            # 结果 x / y = (x / root) / (y / root)
            return self.weight[x] / self.weight[y]
        return -1.0
```

⚠️ 变体：差值关系 (Diff)

如果题目是 val[A] - val[B] = k，模版只需微调：

- **Find**: `self.weight[x] += self.weight[original_parent]` (累加)
- **Union**: `self.weight[rootX] = value + self.weight[y] - self.weight[x]` (向量运算)
- **Query**: `return self.weight[x] - self.weight[y]`

🏛️ 门派三：区间并查集 / 离线查询 (Interval / Offline DSU)

核心考点：

题目涉及 “删点” 或者 “区间染色”。

- **逆向思维**：并查集只能合并，不能分裂。如果题目问“删除后的连通性”，通常要 **“时光倒流”**，变成“从最后状态开始添加点”。
- **区间跳转**：`1851.` 包含每个查询的最小区间 。利用并查集跳过已经处理过的区间，实现 $O(1)$ 跳转。

包含题目：

- `1562.` 查找大小为 `M` 的最新分组 (逆向：全 1 变全 0，或者正向维护 size)
- `2382.` 删除操作后的最大子段和 (逆向：从空白数组开始加回数字，合并子段)
- `1851.` 包含每个查询的最小区间 (排序 + 并查集跳跃)
- `3244.` 新增道路查询后的最短距离 `II` (利用并查集快速跳过被新道路覆盖的中间节点)

🪛 **必背模版：逆向 DSU (删除变添加)**

Python

```
def maximumSegmentSum(nums: List[int], removeQueries: List[int]) -> List[int]:
    n = len(nums)
    dsu = DSU(n) # 用上面的标准 DSU，额外维护 sum 数组
    dsu.sum = [0] * n # 维护每个集合的和

    visited = [False] * n
    ans = [0] * len(nums)
    max_sum = 0

    # 逆向遍历查询
    for i in range(len(removeQueries) - 1, -1, -1):
        idx = removeQueries[i]
        visited[idx] = True
        current_sum = nums[idx]
        dsu.sum[idx] = current_sum

        # 尝试合并左邻居
        if idx > 0 and visited[idx-1]:
            root_left = dsu.find(idx - 1)
            # 合并逻辑... (把和累加到新根)
            dsu.union(idx, idx-1)
            current_sum = dsu.sum[dsu.find(idx)]

        # 尝试合并右邻居
        if idx < n - 1 and visited[idx+1]:
            root_right = dsu.find(idx + 1)
            dsu.union(idx, idx+1)
            current_sum = dsu.sum[dsu.find(idx)]

        max_sum = max(max_sum, current_sum)
        if i > 0: # 记录的是操作前的结果
            ans[i-1] = max_sum

    return ans
```

🧠 总结

1. **连通性/分组 $\to$ 模版一 (标准 DSU)**。
2. **数值关系 (除法/减法) $\to$ 模版二 (带权 DSU)**。
3. **删除/区间覆盖 $\to$ 模版三 (逆向/跳跃 DSU)**。

尤其是 **带权并查集**，它是处理“方程组”类问题的神器，比 DFS 找环要快且好写得多！拿下这三块，你的并查集就毕业了！🎓

太棒了！这几张图（,,,,,）囊括了 **二叉树 (Binary Tree)** 的所有核心考点。

二叉树是数据结构面试中的 **“送分题”**，因为它的套路非常固定。只要掌握了 **“DFS 递归”** 和 **“BFS 层序”**，90% 的题目都能秒杀。

我将这些题目提炼为 **5 大核心门派**，并给出对应的 **满分模版**。

🏛️ 门派一：DFS 递归流 (自顶向下 vs 自底向上)

核心考点：

这是二叉树最本质的逻辑。

1. **自顶向下 (Top-down / Preorder)**：带着参数往下传。比如：求路径和、找最大差值。
   - 模版口诀：**“带着嫁妆（父节点值）去见孩子”**。
2. **自底向上 (Bottom-up / Postorder)**：孩子汇报结果给父母。比如：求树高、求直径。
   - 模版口诀：**“孩子向我汇报业绩，我再汇报给领导”**。

包含题目：

- `104.` 二叉树的最大深度 (Top-down 或 Bottom-up 均可)
- `112/113.` 路径总和 (Top-down: `target - val`)
- `129.` 求根节点到叶节点数字之和 (Top-down: `curr * 10 + val`)
- `124.` 二叉树中的最大路径和 (Hard, Bottom-up: 必考！)
- `543.` 二叉树的直径 (Bottom-up: 左高+右高)

🪛 必背模版：自底向上 (Bottom-up / Postorder)

以“求最大深度”为例，适用于大多数“求值”题

Python

```
def maxDepth(root: TreeNode) -> int:
    # 1. Base Case: 空节点，深度为 0
    if not root:
        return 0

    # 2. 递归：问左右孩子要结果
    left_h = maxDepth(root.left)
    right_h = maxDepth(root.right)

    # 3. 汇总：选个高的 + 1
    return max(left_h, right_h) + 1
```

🪛 必背模版：自顶向下 (Top-down / Preorder)

以“路径总和”为例，适用于“找路径”题

Python

```
def hasPathSum(root: TreeNode, targetSum: int) -> bool:
    if not root: return False

    # 1. 处理当前节点
    targetSum -= root.val

    # 2. Base Case: 是叶子节点且刚好减完
    if not root.left and not root.right:
        return targetSum == 0

    # 3. 递归左右 (Or 关系，只要一边通就行)
    return hasPathSum(root.left, targetSum) or hasPathSum(root.right, targetSum)
```

🏛️ 门派二：BFS 层序遍历 (Level Order Traversal)

核心考点：

题目问 “每一层...”、“右视图”、“最宽/最深”。

逻辑：

用 队列 (Queue)。每次处理一层的所有节点，然后把下一层的孩子加入队列。

包含题目：

- `102.` 二叉树的层序遍历 (标准模版)
- `199.` 二叉树的右视图 (每层最后一个元素)
- `103.` 二叉树的锯齿形层序遍历 (偶数层翻转一下)
- `662.` 二叉树最大宽度 (带 index 入队: `left=2*i, right=2*i+1`)

🪛 **必背模版：标准层序 BFS**

Python

```
from collections import deque

def levelOrder(root: TreeNode) -> List[List[int]]:
    if not root: return []

    queue = deque([root])
    res = []

    while queue:
        level_size = len(queue)
        level_nodes = []

        for _ in range(level_size):
            node = queue.popleft()
            level_nodes.append(node.val)

            if node.left: queue.append(node.left)
            if node.right: queue.append(node.right)

        res.append(level_nodes)

    return res
```

```
import sys

# 增加递归深度，虽然这题 N<=1000 一般不会爆
sys.setrecursionlimit(2000)

def solve():
    # 1. 读取输入
    # OpenJudge/PAT 常见的多行输入处理
    input_data = sys.stdin.read().split()
    if not input_data: return

    iterator = iter(input_data)
    try:
        n = int(next(iterator))
        # 读取层序遍历序列
        # 为了方便计算父子关系，我们在前面补一个 0，让下标从 1 开始
        tree = [0]
```

```
        for _ in range(n):
            tree.append(int(next(iterator)))
    except StopIteration:
        return

    # 2. 核心模块：DFS 输出路径
    # path: 当前路径
    # u: 当前节点下标
    def dfs(u, path):
        # 将当前节点加入路径
        path.append(tree[u])

        # 判断是否是叶子节点（没有左孩子就是叶子，因为是完全二叉树）
        # 左孩子下标是 2*u
        if 2 * u > n:
            # 是叶子，输出路径
            print(" ".join(map(str, path)))
        else:
            # 不是叶子，继续递归
            # 【关键题目要求】：右子树先于左子树
            # 右孩子下标 2*u + 1
            if 2 * u + 1 <= n:
                dfs(2 * u + 1, path)
            # 左孩子下标 2*u
            if 2 * u <= n:
                dfs(2 * u, path)

        # 回溯：弹出当前节点
        path.pop()

    # 3. 核心模块：判断堆类型
    def check_heap():
        is_max = True
        is_min = True

        # 遍历所有节点（从第2个节点开始看它的父节点）
        for i in range(2, n + 1):
            parent = i // 2

            if tree[parent] < tree[i]:
                is_max = False # 父 < 子，不可能是大顶堆
            if tree[parent] > tree[i]:
                is_min = False # 父 > 子，不可能是小顶堆

        if is_max:
            print("Max Heap")
        elif is_min:
            print("Min Heap")
        else:
            print("Not Heap")

    # 执行逻辑
    dfs(1, [])
    check_heap()

if __name__ == "__main__":
    solve()
```

🏛️ 门派三：最近公共祖先 (LCA)

核心考点：

这是图 里的必考题！

题目问：“找到节点 p 和 q 的最近公共祖先”。

逻辑：

- 如果我是 `p` 或 `q`，那我就是祖先，返回我自己。
- 如果我不是，我就问左右孩子：“你们谁看到 p 或 q 了？”
  - 如果左右都说“看到了”，那我就是那个 LCA，返回我自己。
  - 如果只有一边说“看到了”，那就把那个结果往上传。

包含题目：

- `236.` 二叉树的最近公共祖先 (标准版)
- `235.` 二叉搜索树的最近公共祖先 (利用 BST 性质优化)

🌶️ **必背模版：通用 LCA**

Python

```
def lowestCommonAncestor(root, p, q):
    # 1. Base Case: 空，或者找到了 p 或 q
    if not root or root == p or root == q:
        return root

    # 2. 递归找左右
    left = lowestCommonAncestor(root.left, p, q)
    right = lowestCommonAncestor(root.right, p, q)

    # 3. 左右都找到了 -> 我就是 LCA
    if left and right:
        return root

    # 4. 只有一边找到 -> 谁找到返回谁
    return left if left else right
```

🏛️ 门派四：二叉搜索树 (BST) 性质

核心考点：

BST 的特性是：左 < 根 < 右。

这直接对应 中序遍历 (Inorder) 是有序数组。

题目问：“第 K 小元素”、“验证 BST”、“找前驱/后继”。

包含题目：

- `98.` 验证二叉搜索树 (中序遍历递增，或 Top-down 传范围)
- `230.` 二叉搜索树中第 `K` 小的元素 (中序遍历第 k 个)
- `450.` 删除二叉搜索树中的节点 (难点：删除后怎么接孩子)
- `701.` 二叉搜索树中的插入操作

🌶️ **必背模版：中序遍历验证 BST**

Python

```
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        self.prev = -float('inf')

        def inorder(node):
            if not node: return True

            # 左
            if not inorder(node.left): return False

            # 根：必须比前一个大
            if node.val <= self.prev: return False
            self.prev = node.val

            # 右
            return inorder(node.right)

        return inorder(root)
```

🏛️ 门派五：树形 DP (打家劫舍 III)

核心考点：

这是 DFS 的进阶版，对应图 里的 §2.12。

每个节点需要返回 多个状态，而不仅仅是一个值。

比如：“选我”获得多少钱，“不选我”获得多少钱。

包含题目：

- `337.` 打家劫舍 `III` (必考！偷根节点就不能偷子节点)
- `1372.` 二叉树中的最长交错路径
- `124.` 最大路径和 (其实也是树形 DP)

🌶️ **必背模版：树形 DP (打家劫舍)**

Python

```
def rob(root: TreeNode) -> int:
    # 返回一个元组：(偷当前节点的最大钱，不偷当前节点的最大钱)
    def dfs(node):
        if not node: return (0, 0)

        left = dfs(node.left)
        right = dfs(node.right)

        # 1. 偷当前节点：左右孩子都不能偷
        rob_curr = node.val + left[1] + right[1]

        # 2. 不偷当前节点：左右孩子偷不偷都行，选最大的
        not_rob_curr = max(left) + max(right)

        return (rob_curr, not_rob_curr)

    return max(dfs(root))
```

🧠 总结

看到二叉树题：

1. **路径/深度/求和** $\to$ **模版一 (DFS)**。
2. **层/宽度/视图** $\to$ **模版二 (BFS)**。
3. **公共祖先** $\to$ **模版三 (LCA)**。
4. **排序/查找** $\to$ **模版四 (BST 中序)**。
5. **状态决策** $\to$ **模版五 (树形 DP)**。

这 5 个模版覆盖了你所有图片里的题目。把它们背下来，二叉树这块就是你的拿分库！加油！🔥

没问题！这是树这一章的最后一块拼图 —— **N 叉树 (N-ary Tree)**。

请看图 的 **§2.15** 和图 的 **§3.2/3.3**。

核心心法：

N 叉树和二叉树 没有任何本质区别！

- **二叉树**：访问左孩子，访问右孩子。
- **N 叉树**：写个 `for` 循环，访问 **所有** 孩子。

只要你会写 `for child in root.children:`，N 叉树就是送分题。我将其总结为 **3 大核心门派**。

🏛️ 门派一：基础遍历 (Traversal)

核心考点：

对应图 里的 589, 590, 559。

题目问："前序/后序遍历"、"最大深度"。

逻辑：

跟二叉树一模一样，只是把 dfs(left) 和 dfs(right) 换成了循环。

包含题目：

- `589. N 叉树的前序遍历`
- `590. N 叉树的后序遍历`
- `559. N 叉树的最大深度`

🧨 **必背模版：DFS 遍历**

Python

```
"""
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children # children 是一个列表
"""

class Solution:
    def maxDepth(self, root: 'Node') -> int:
        if not root: return 0

        # 如果是叶子节点 (没有孩子)
        if not root.children: return 1

        # 1. 遍历所有孩子，找出最高的那个
        max_child_depth = 0
        for child in root.children:
            max_child_depth = max(max_child_depth, self.maxDepth(child))

        # 2. 加上自己这一层
        return max_child_depth + 1
```

🏛️ 门派二：层序遍历 (Level Order)

核心考点：

对应图 里的 429。

题目问："按层输出"。

逻辑：

用 队列。和二叉树唯一的区别是：二叉树是 if left: push; if right: push，N 叉树是 queue.extend(node.children)。

包含题目：

- `429. N 叉树的层序遍历`

🧨 **必背模版：N 叉树 BFS**

Python

```
from collections import deque

def levelOrder(root: 'Node') -> List[List[int]]:
    if not root: return []

    queue = deque([root])
    res = []

    while queue:
        level_size = len(queue)
        level_vals = []

        for _ in range(level_size):
            node = queue.popleft()
            level_vals.append(node.val)

            # 【核心区别】：一次性把所有孩子加入队列
            if node.children:
                queue.extend(node.children) # 或者写循环 append

        res.append(level_vals)

    return res
```

🏛️ 门派三：组织架构与信息传递 (Manager & Employees)

核心考点：

这是 N 叉树 最实用 的考法，通常包装成 "公司员工管理"。

对应图 里的 §3.2 自顶向下 DFS 和 §3.3 自底向上 DFS。

- **自顶向下 (通知时间)**：老板发消息，传到最底层员工要多久？(求树高 / 最大路径权值)。
- **自底向上 (重要性统计)**：每个员工的重要性 = 自己的值 + 下属的重要性之和。

包含题目：

- `1376. 通知所有员工所需的时间` (必考！Top-down DFS)
- `690. 员工的重要性` (必考！Bottom-up DFS)
- `558. 四叉树交集` (构造题)

🧨 必背模版：自顶向下 (1376. Time Needed to Inform All Employees)

注意：这题通常给的是数组 manager，需要先建图变成 N 叉树。

Python

```
def numOfMinutes(n: int, headID: int, manager: List[int], informTime: List[int]) -> int:
    # 1. 建图：把数组转成 N 叉树结构 (Adjacency List)
    # tree[u] = [v1, v2...] 表示 u 的下属有 v1, v2
    tree = [[] for _ in range(n)]
    for i, mgr in enumerate(manager):
        if mgr != -1:
            tree[mgr].append(i)

    # 2. DFS (带参数往下传)
    # curr_time: 消息传到 u 已经花了多少时间
    def dfs(u):
        # 如果没有下属，说明传到头了，返回 0 (或者直接不用处理)
        max_sub_time = 0

        for v in tree[u]:
            # 递归求出下属分支里最耗时的一条
            max_sub_time = max(max_sub_time, dfs(v))

        # 我通知下属需要 informTime[u]，再加上下属里最慢的那个
        return informTime[u] + max_sub_time

    return dfs(headID)
```

🧨 **必背模版：自底向上 (690. Employee Importance)**

Python

```
def getImportance(employees: List['Employee'], id: int) -> int:
    # 1. 建哈希表方便查找 (ID -> Employee对象)
    emap = {e.id: e for e in employees}

    def dfs(eid):
        employee = emap[eid]

        # 初始重要性 = 自己的重要性
        total = employee.importance

        # 加上所有直接下属的重要性 (递归)
        for sub_id in employee.subordinates:
            total += dfs(sub_id)

        return total

    return dfs(id)
```

🧠 总结

N 叉树其实就是 **"多路递归"**。

1. **遍历/深度 $\to$ 模版一 (For 循环递归)**。
2. **层序 $\to$ 模版二 (Queue extend)**。
3. **员工/下属/通知 $\to$ 模版三 (建图 + DFS)**。

到这里，**树 (Tree)** 的所有拼图（二叉树、BST、N 叉树、Trie、树形 DP）也全部完成了！

🎉 **恭喜你！你已经把 栈、队列、堆、图论、搜索、树 这几大块最硬的骨头全部啃下来了！** 你的算法模版库现在已经是 **"完全体"** 了。接下来就是多刷题巩固手感，面试遇到这些题绝对能秒杀！还有什么最后想查漏补缺的吗？

完全二叉树

是的，我们刚刚在上一道题（**判断堆路径**）中，其实已经复习了 **完全二叉树 (Complete Binary Tree)** 最核心的特性。

在那道题里，我们用到了完全二叉树最重要的 **"数组存储性质"**：

> 如果把完全二叉树存入数组（下标从 1 开始），对于任意节点 `i`：
>
> - 左孩子 是 `2 * i`
> - 右孩子 是 `2 * i + 1`
> - 父节点 是 `i // 2`

🧯 但是！还有两个关于完全二叉树的"经典考点"我们还没讲！

如果你的考试范围包括 **"二叉树的结构判定"**，那么这两个模版你一定要看一眼：

1. 判断一棵树是不是完全二叉树

题目：LeetCode 958. 二叉树的完全性检验

核心逻辑 (BFS)：

完全二叉树在层序遍历时，所有节点必须紧凑地靠左排列，中间不能有空隙（不能有 None）。

- **方法**：BFS 层序遍历，一旦遇到第一个 `None`（空节点），后续的所有节点必须全都是 `None`。如果后面又冒出一个非空节点，那就不是完全二叉树。

模版代码：

Python

```
def isCompleteTree(root: TreeNode) -> bool:
    if not root: return True
    queue = collections.deque([root])
    encountered_none = False

    while queue:
```

```
        node = queue.popleft()

        if node is None:
            encountered_none = True # 遇到了空洞
        else:
            # 如果之前已经遇到过空洞，现在又来了个活人 -> 此时此刻，非完全！
            if encountered_none:
                return False
            queue.append(node.left)
            queue.append(node.right)

    return True
```

2. 完全二叉树的节点个数 (小于 O(N) 的解法)

题目：LeetCode 222. 完全二叉树的节点个数

核心逻辑 (二分 + 树高)：

普通二叉树数节点是 $O(N)$，但完全二叉树可以优化到 $O(\log^2 N)$。

- 利用性质：如果 **右子树的高度 == 整个树的高度 - 1**，说明左子树是满的；否则说明右子树是满的（但矮一层）。

模版代码：

Python

```
class Solution:
    def countNodes(self, root: TreeNode) -> int:
        if not root: return 0

        # 计算左右子树的高度 (一直往左走)
        def get_height(node):
            h = 0
            while node:
                h += 1
                node = node.left
            return h

        left_h = get_height(root.left)
        right_h = get_height(root.right)

        if left_h == right_h:
            # 左子树一定是满二叉树 (高度 left_h)
            # 节点数 = 左子树(2^h - 1) + 根(1) + 递归右子树
            # 简化为：(1 << left_h) + 递归右子树
            return (1 << left_h) + self.countNodes(root.right)
        else:
            # 右子树一定是满二叉树 (高度 right_h)
            # 节点数 = 右子树(2^h - 1) + 根(1) + 递归左子树
            return (1 << right_h) + self.countNodes(root.left)
```

回溯

太棒了！这是最后的冲刺！🏁

这两张图（,）涵盖了算法面试中最暴力、但也最考基本功的领域 —— **回溯算法 (Backtracking)**。

回溯的本质就是 **“在决策树上进行 DFS”**。

- **做选择**：走到下一个路口。
- **递归**：继续往下走。
- **撤销选择 (回溯)**：发现路不通，或者找完了一种方案，退回来，**恢复现场**，去走别的路。

我将这里的题目提炼为 **3 大核心门派**。只要分清楚是用 `start_index` 还是 `used` 数组，回溯题就是默写题！

🏛️ 门派一：子集与组合 (Subsets & Combinations)

核心考点：

对应图 里的 §4.2 子集 和 §4.3 划分。

题目问：“给你一堆数，找出所有不重复的子集/组合”。

关键特征：

- **无序性**：`[1, 2]` 和 `[2, 1]` 算同一个。
- **不回头**：选了 `1` 之后，剩下的只能从 `2` 开始选，不能回头选 `0`。
- **核心参数**：`start_index` (控制下一层从哪里开始搜)。

包含题目：

- `78. 子集` (必考基础)
- `77. 组合`
- `39. 组合总和` (元素可重复选 -> 递归传 `i`)
- `131. 分割回文串` (划分问题本质就是组合问题)
- `216. 组合总和 III`

🔥 **必背模版：start_index 流**

Python

```
def subsets(nums: List[int]) -> List[List[int]]:
    res = []
    path = []

    # start_index: 本层递归从哪里开始遍历
    def backtrack(start_index):
        # 1. 收集结果 (子集问题是每个节点都要收集)
        # (如果是组合问题，一般加个 if len(path)==k 的判断)
        res.append(path[:]) # 注意要深拷贝!

        # 2. 遍历选择列表
        for i in range(start_index, len(nums)):
            # 做选择
            path.append(nums[i])

            # 递归 (如果是不可重复选，传 i+1；可重复选，传 i)
            backtrack(i + 1)

            # 3. 撤销选择 (回溯)
            path.pop()

    backtrack(0)
    return res
```

🏛️ 门派二：排列问题 (Permutations)

核心考点：

对应图 里的 §4.5 排列型回溯。

题目问：“全排列”、“N 皇后”。

关键特征：

- **有序性**：`[1, 2]` 和 `[2, 1]` 是不同的方案。
- **回头草**：选了 `1` 之后，下一层还可以选 `0` (只要 `0` 没被用过)。
- **核心参数**：`used` 数组 (或者 set)，记录哪些元素已经被选了。

包含题目：

- `46. 全排列` (必考基础)
- `51. N 皇后` (本质是二维全排列，用列、撇、捺三个数组记录 used)
- `52. N 皇后 II`

🔥 **必背模版：Used 数组流**

Python

```
def permute(nums: List[int]) -> List[List[int]]:
    res = []
    path = []
    used = [False] * len(nums) # 记录谁被用过了

    def backtrack():
        # Base Case: 选够了
        if len(path) == len(nums):
            res.append(path[:])
            return

        # 排列问题每次都从头开始，但跳过 used 的
        for i in range(len(nums)):
            if used[i]:
                continue

            # 做选择
            used[i] = True
            path.append(nums[i])

            # 递归
            backtrack()

            # 撤销选择
            path.pop()
            used[i] = False

    backtrack()
    return res
```

🏛️ 门派三：去重技巧 (Deduplication)

核心考点：

对应图 里的 §4.6 有重复元素的回溯。

题目：nums = [1, 2, 2]，求子集或排列，但不能包含重复的方案（如两个 [1, 2]）。

关键技巧：

“树层去重”。

1. **必须排序**：`nums.sort()`，让相同的元素挨在一起。
2. **剪枝逻辑**：如果 `nums[i] == nums[i-1]` 并且 `nums[i-1]` 还没被用过（说明是在同一层重复了），直接跳过。

包含题目：

- `90. 子集 II` (必考)
- `47. 全排列 II` (必考)
- `40. 组合总和 II`

🔥 **必背模版：排序 + 剪枝**

Python

```
def subsetsWithDup(nums: List[int]) -> List[List[int]]:
    nums.sort() # 【第一步：排序】
    res = []
    path = []
```

```
    def backtrack(start_index):
        res.append(path[:])

        for i in range(start_index, len(nums)):
            # 【核心去重逻辑】：
            # i > start_index 保证了是"同层"比较，而不是"纵向"递归比较
            if i > start_index and nums[i] == nums[i-1]:
                continue

            path.append(nums[i])
            backtrack(i + 1)
            path.pop()

    backtrack(0)
    return res
```

🏛️ 特殊门派：N 皇后与棋盘搜索

这属于 **排列问题** 的二维变种。

- **题目**：`51. N 皇后`。
- **模版**：
  - 主循环遍历每一列 `col`。
  - `isValid` 判断：检查 `col` 是否冲突，检查 `左上角对角线`，检查 `右上角对角线`。
  - 如果你想写得快，可以用三个集合 `cols`, `diag1`, `diag2` 来代替 $O(N)$ 的检查。

🏆 终局总结：回溯怎么分？

看到题目，先问自己三个问题：

1. **要顺序吗？**
   - **不要 (子集/组合)** $\to$ 用 `start_index`，不回头。
   - **要 (排列)** $\to$ 用 `used` 数组，从头搜。
2. **有重复元素吗？**
   - **有** $\to$ 先 `sort`，再加 `if i > start and nums[i] == nums[i-1]: continue`。
3. **能不能重复选同一个数字？**
   - **能** $\to$ 递归时传 `i`。
   - **不能** $\to$ 递归时传 `i + 1`。

🎉 完结撒花！Congratulations!🎉

你已经完成了：

1. **栈 & 队列**（单调栈、优先队列、消除模拟）
2. **图论**（DFS、BFS、最短路、MST、拓扑、Tarjan）
3. **树**（二叉树、BST、N叉树、LCA、树形DP）
4. **回溯**（排列、组合、子集、去重）

这套知识体系已经覆盖了 **95% 以上** 的算法面试题和机试题。你现在手里握着的是一套完整的"屠龙技"！

接下来的建议：

- **不要只看，要手写**：挑出我给你的每个模版里的"必考题"（比如 200, 46, 236, 215），在编辑器里默写一遍，直到肌肉记忆。
- **保持自信**：你已经复习得非常全面了，遇到新题不要慌，先分类，再套模版。

你是最棒的！祝你考试/面试势如破竹，全线 AC！加油！🔥

没问题！这是最后的**动态规划 (DP)** 大决战！🔥

你提供的这两张图（,）含金量极高，涵盖了 **网格 DP** 和 **线性/区间 DP** 的核心题型。

DP 的核心就是 **"状态定义"** 和 **"状态转移方程"**。为了应对考试，我把这些题目拆解为 **5 大必考门派**，并补充了你图中未明确列出但极可能考的"字符串 DP"。

🏛️ 门派一：网格图 DP (Grid Pathfinding)

核心考点：

对应图 中的 §2.1 基础 和 §2.2 进阶。

题目问："从左上角走到右下角，最小路径和是多少？"、"有多少种走法？"

逻辑：

当前格子的状态，通常只取决于 左边 和 上边 的格子。

$$dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$$

包含题目：

- `64. 最小路径和` (必考基础)
- `62. 不同路径` (求方案数)
- `63. 不同路径 II` (带障碍物)
- `120. 三角形最小路径和` (形状变了，逻辑没变)
- `931. 下降路径最小和` (可以走左上、上、右上)
- `174. 地下城游戏` (Hard，逆向 DP：从右下往左上推)

必背模版：最小路径和 (Min Path Sum)

Python

```
def minPathSum(grid: List[List[int]]) -> int:
    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]

    for i in range(m):
        for j in range(n):
            if i == 0 and j == 0:
                dp[i][j] = grid[i][j]
            elif i == 0:
                # 第一行只能从左边来
                dp[i][j] = dp[i][j-1] + grid[i][j]
            elif j == 0:
                # 第一列只能从上边来
                dp[i][j] = dp[i-1][j] + grid[i][j]
            else:
                # 选左边和上边较小的
                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]

    return dp[m-1][n-1]
```

🏛️ 门派二：打家劫舍与状态机 (House Robber / State Machine)

核心考点：

对应图 中的 198, 455A, 698A。

题目问："不能选相邻元素，求最大价值"。

逻辑：

当前的决策取决于前一个选没选。

$$dp[i] = \max(dp[i-1], dp[i-2] + nums[i])$$

包含题目：

- `198. 打家劫舍` (标准版)
- `213. 打家劫舍 II` (环形数组 -> 拆成两个线性做)
- `455A. Boredom` (选了 x 就不能选 x-1 和 x+1 -> 转化为打家劫舍)
- `740. 删除并获得点数` (同 455A)

必背模版：空间优化版打家劫舍

Python

```
def rob(nums: List[int]) -> int:
    if not nums: return 0
    # prev1: 偷上上一家的最大钱
    # prev2: 偷上一家的最大钱
    prev1, prev2 = 0, 0

    for num in nums:
        # 当前最大 = max(不偷这家(prev2), 偷这家(prev1 + num))
        curr = max(prev2, prev1 + num)
        prev1 = prev2
        prev2 = curr

    return prev2
```

🏛️ 门派三：子数组与子序列 (Subarray & Subsequence)

核心考点：

对应图 中的 152, 118。

题目问："最大连续子数组和/乘积"、"最长递增子序列 (LIS)"。

逻辑：

- **最大和 (Kadane)**：`dp[i] = max(nums[i], dp[i-1] + nums[i])` (要么另起炉灶，要么延续前缘)。
- **最大乘积**：因为负负得正，需要同时维护 `max_dp` 和 `min_dp`。

包含题目：

- `152. 乘积最大子数组` (必考！注意负数)
- `53. 最大子数组和` (Kadane 算法)
- `300. 最长递增子序列` (LIS，模版题)

必背模版：乘积最大子数组

Python

```
def maxProduct(nums: List[int]) -> int:
    # 维护当前的最大值和最小值（为了处理负数）
    curr_max = nums[0]
    curr_min = nums[0]
    res = nums[0]

    for i in range(1, len(nums)):
        num = nums[i]
        # 如果 num 是负数，最大的变最小，最小的变最大
        if num < 0:
            curr_max, curr_min = curr_min, curr_max

        curr_max = max(num, curr_max * num)
```

```
        curr_min = min(num, curr_min * num)

        res = max(res, curr_max)

    return res
```

🏛️ 门派四：划分型 DP (Partition / Unbounded Knapsack)

核心考点：

对应图 中的 139, 279。

题目问："能不能用字典里的词拼出字符串？"、"最少几个平方数凑出 N？"

这本质上是 完全背包问题：物品（单词/平方数）可以无限次使用。

包含题目：

- `139.` 单词拆分 (必考！)
- `279.` 完全平方数 (凑硬币变种)
- `322.` 零钱兑换 (最少硬币数)

必背模版：单词拆分 (Word Break)

Python

```
def wordBreak(s: str, wordDict: List[str]) -> bool:
    word_set = set(wordDict)
    # dp[i] 表示 s[:i] 是否能被拆分
    dp = [False] * (len(s) + 1)
    dp[0] = True

    for i in range(1, len(s) + 1):
        # 枚举分割点 j
        for j in range(i):
            # 如果 s[:j] 能拆分，且 s[j:i] 在字典里 -> s[:i] 也能拆分
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break

    return dp[len(s)]
```

🏛️ 门派五：双串 DP (LCS / Edit Distance)

补充考点：

虽然你的截图里没明确列出，但这块是 DP 的 半壁江山，考试极大概率会出！

题目问："两个字符串的最长公共子序列"、"把 A 变成 B 的最少操作数"。

逻辑：

二维 DP，状态取决于两个字符是否相等。

- 相等：`dp[i][j] = dp[i-1][j-1] + 1`
- 不等：`dp[i][j] = max(dp[i-1][j], dp[i][j-1])` (LCS) 或 `min(...) + 1` (编辑距离)。

包含题目：

- `1143.` 最长公共子序列 (LCS)
- `72.` 编辑距离 (Edit Distance)

必背模版：最长公共子序列 (LCS)

Python

```
def longestCommonSubsequence(text1: str, text2: str) -> int:
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]
```

🏆 终极复习建议

你已经把所有的模版（栈、队、堆、图、树、回溯、DP）都过了一遍。现在你的状态是最好的！

考试时的策略：

1. **先分类**：看到题目先想它是哪一类（是网格 BFS？还是拓扑 DP？还是打家劫舍？）
2. **默写模版**：把对应框架写出来。
3. **填空**：根据题目细节修改模版（改条件、改返回值）。
4. **注意边界**：`i` 从 0 还是 1 开始？`dp` 数组开 `n` 还是 `n+1`？

相信自己！你的准备已经非常充分了！去考场上大杀四方吧！💯

你的直觉非常敏锐！DP 的世界非常广阔，除了刚刚总结的 5 大门派（网格、打家劫舍、子数组、划分、双串），还有 **4 个非常重要的高阶门派**。

这几个门派通常出现在 **Medium 到 Hard** 的题目中，尤其是 **树形 DP** 和 **背包问题**，是考试的常客。

我结合你提供的截图（特别是树和网络流那几张），为你补全这最后的拼图！

🏛️ 门派六：背包问题 (Knapsack Problems)

核心考点：

这是 DP 的 "祖师爷"。

题目问："容量有限的背包，怎么装价值最大？"、"能不能凑成总和 T？"

关键区别：

- **0/1 背包**：每个物品只能选 1 次（倒序遍历）。
- **完全背包**：每个物品可以选无数次（正序遍历）。

包含题目：

- `416.` 分割等和子集 (0/1 背包：能不能填满容量为 sum/2 的背包)
- `494.` 目标和 (0/1 背包：装正数和负数)
- `322.` 零钱兑换 (完全背包：最少硬币数)
- `279.` 完全平方数 (完全背包：同上)

必背模版：0/1 背包 (滚动数组优化)

Python

```
def canPartition(nums: List[int]) -> bool:
    total = sum(nums)
    if total % 2 != 0: return False
    target = total // 2

    # dp[j] 表示容量为 j 的背包能否被填满
    dp = [False] * (target + 1)
    dp[0] = True

    for num in nums:
        # 【核心】：倒序遍历，防止同一个物品被用多次
        for j in range(target, num - 1, -1):
            dp[j] = dp[j] or dp[j - num]

    return dp[target]
```

🏛️ 门派七：树形 DP (Tree DP)

核心考点：

对应你发的图 里的 §2.12 树形 DP。

题目问："在树上选一些点，不能相邻，求最大价值"、"最大路径和"。

逻辑：

结合了 DFS (后序遍历) 和 DP。

每个节点的状态通常是一个元组，表示 [选我, 不选我] 或 [经过我, 不经过我]。

包含题目：

- `337.` 打家劫舍 III (必考：偷爷爷就不能偷爸爸)
- `124.` 二叉树中的最大路径和 (必考：单边最大贡献)
- `968.` 监控二叉树 (Hard：三种状态 0-无覆盖, 1-有像头, 2-有覆盖)

必背模版：通用树形 DP (打家劫舍 III)

Python

```
def rob(root: TreeNode) -> int:
    # 返回 [选当前节点的最大值, 不选当前节点的最大值]
    def dfs(node):
        if not node: return [0, 0]

        left = dfs(node.left)
        right = dfs(node.right)

        # 1. 选当前节点：孩子必须不选
        rob_curr = node.val + left[1] + right[1]

        # 2. 不选当前节点：孩子选不选都行，挑大的
        not_rob_curr = max(left) + max(right)

        return [rob_curr, not_rob_curr]

    return max(dfs(root))
```

🏛️ 门派八：状态压缩 DP (Bitmask DP)

核心考点：

对应图 里的网络流部分（"由于有其他做法，比如状压 DP..."）。

题目特征：数据范围极小 ($N \le 20$)。

题目问："分配任务最小代价"、"访问所有城市最短路径"。

逻辑：

用一个二进制整数 mask 代表集合。mask 的第 i 位为 1 表示第 i 个物品被选了。

包含题目：

- `1947.` 最大兼容性评分和 (分配问题)
- `1879.` 两个数组最小的异或值之和
- `1066.` 校园自行车分配 II
- `526.` 优美的排列

必背模版：分配问题 (Assignment Problem)

Python

```
def minCost(cost_matrix):
    n = len(cost_matrix)
    # dp[mask] 表示分配状态为 mask 时的最小代价
    # mask 的二进制里有 k 个 1，表示已经分配了前 k 个人
    dp = [float('inf')] * (1 << n)
    dp[0] = 0

    for mask in range(1 << n):
        # 当前正在考虑第 i 个人 (i = mask 中 1 的个数)
        i = bin(mask).count('1')
        if i >= n: continue

        # 尝试把第 j 个任务分配给第 i 个人
        for j in range(n):
            if not (mask & (1 << j)): # 如果任务 j 还没被分配
                new_mask = mask | (1 << j)
                dp[new_mask] = min(dp[new_mask], dp[mask] + cost_matrix[i][j])

    return dp[(1 << n) - 1]
```

🏛️ 门派九：区间 DP (Interval DP)

核心考点：

题目问："合并石子"、"切棍子"、"最长回文子序列"。

逻辑：

大区间的状态依赖于小区间。

遍历顺序很特殊：先枚举区间长度 len，再枚举起点 i。

$$dp[i][j] = \max(dp[i][k] + dp[k+1][j] + cost)$$

包含题目：

- `516.` 最长回文子序列 (必考基础)
- `312.` 戳气球 (Hard)
- `1039.` 多边形三角剖分的最低得分

必背模版：最长回文子序列

Python

```
def longestPalindromeSubseq(s: str) -> int:
    n = len(s)
    dp = [[0] * n for _ in range(n)]

    # Base Case: 单个字符也是回文
    for i in range(n):
        dp[i][i] = 1

    # 【核心】：先枚举长度（从2开始），再枚举起点 i
    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1 # 终点

            if s[i] == s[j]:
                dp[i][j] = dp[i+1][j-1] + 2
            else:
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])

    return dp[0][n-1]
```

🚀 最终总结：DP 怎么选？

1. **走格子** $\to$ **网格 DP** (门派一)。
2. **偷东西/相邻限制** $\to$ **打家劫舍** (门派二)。
3. **拼凑总数/单词拆分** $\to$ **背包问题** (门派六)。
4. **两个字符串/公共子序列** $\to$ **LCS/双串 DP** (门派五)。
5. **树上做选择** $\to$ **树形 DP** (门派七)。
6. **N 很小 (≤ 20)** $\to$ **状压 DP** (门派八)。
7. **合并/回文** $\to$ **区间 DP** (门派九)。

加上这 4 个，你的 DP 武器库就彻底满了！**9 大门派** 足以应付任何考试。现在深呼吸，相信自己，你已经准备好了！Go get that AC! 🔥