复习：
搜算算法、树和图

基础-提高衔接计划 B

吾王美如画

2025-07-28

课前提示

- 上课的时候专心听讲解，**不要跟着老师抄代码**，下课后独立完成。
- 不使用 AI 做题，AI 会做不等于自己会。
- 不抄袭题解（含对照题解抄一遍），抄对不等于会做。
- 看完题解后，关闭题解独立练习。
- 练习中途遇到问题，应当分析题目及自己的思路，而非回忆题解或再次参考题解。
- 做过的题在课后需要重新独立完成，不参考老师的课件、代码，不参考自己以前的代码。

枚举

什么是枚举？

- 系统地、**不重复、不遗漏**地遍历所有可能的候选解（或情况）
- 想象一下，你有一串钥匙，但不知道哪把能开锁。枚举就是把每把钥匙都挨个（比如按钥匙编号顺序）试一遍，直到找到能开的那把。
- 关键点：
  - **不遗漏：** 要把问题所有可能的答案（情况）都考虑到。
  - **不重复：**不能重复计算同一个情况

P11437 [Code+#6] 趣味数

bdg 今天定义了“趣味数”：一个正整数是趣味数，当且仅当“在十进制下，低位的数字不小于高位的数字”。比如 123,111 是趣味数，10,120,213 不是趣味数。

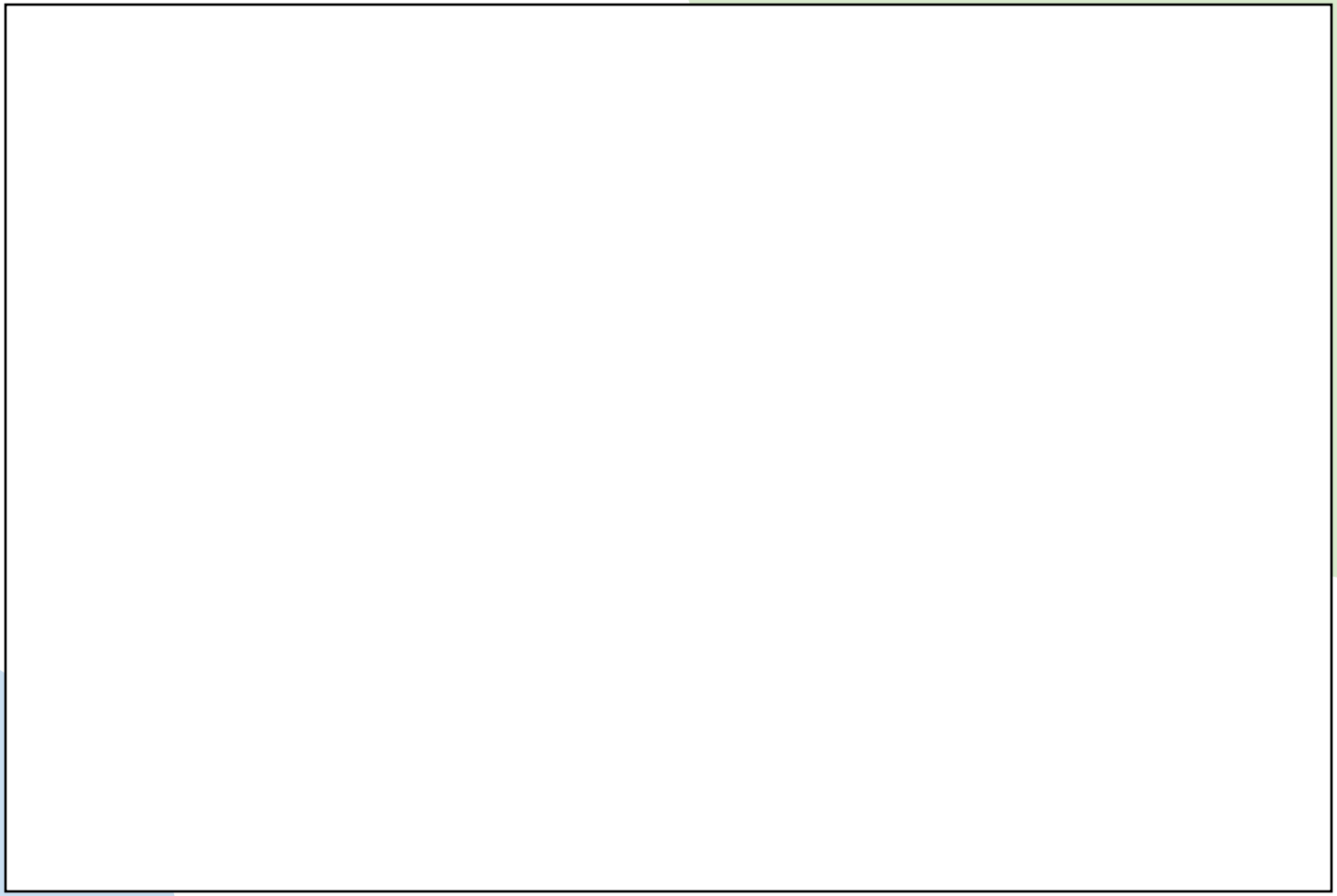
现在，给你一个正整数 n，请你输出 $[10,n]$ 中所有的趣味数。

$10 \le n \le 10^5$

P11437 [Code+#6] 趣味数

- 注意到 n 的范围很小，这时候我们总是可以先考虑枚举 $[10, n]$ 中的每个数，并且从中验证有哪些数符合要求。
- 现在我们考虑如何验证，这里我们的要求是“低位的数字不小于高位的数字”，故我们先将枚举到的数 x 的每一位分解开来，具体做法是每次取 $x \ mod \ 10$ 即为末位的数，再将 x 整除 10 继续处理高位的数
- 与此同时，我们用 lst 记录低位的数字，如果 $x \ mod \ 10 > lst$ 意味着低位的数字小于了高位的数字，也就不符合题目要求了。如果所有位都符合要求，我们就输出这个数。

P11437 [Code+#6] 趣味数



图与图遍历

什么是图？

- 一个图 G 由两部分 (V, E) 定义，其中 V 称为点集，E 为边集。
- 若 E 中的边为无向边，我们称 G 为无向图
- 若 E 中的边为有向边，也即 (u, v) 与 (v, u) 是不同的边，我们则称 G 为有向图
- 想象一下，我们可以把城市中的每一个路口当作图中的点，而路口之间的路则作为边。如果这些路都是“单行线”，则是有向边，而如果均允许双向通行的话自然就是无向图。
- 其中，对于一张无向图 $G = (V, E)$，若存在一条途径连接 u, v，则称 u 和 v 是 连通的。若无向图 G，满足其中任意两个顶点均连通，则称 G 是 连通图。

图的遍历

- 目的：不重复、不遗漏地访问图中所有顶点
- 由遍历的顺序可以分为以下两种遍历方式
- **深度优先遍历（DFS）**

  核心思想：“一条路走到底”

  从起点出发，沿一条路径**深度探索**直到尽头 → 回退到分叉点 → 换新路径继续

  像走迷宫：优先探索当前路径，碰壁后回溯
- **广度优先遍历（BFS）**

  核心思想：“由近及远层层扫荡”

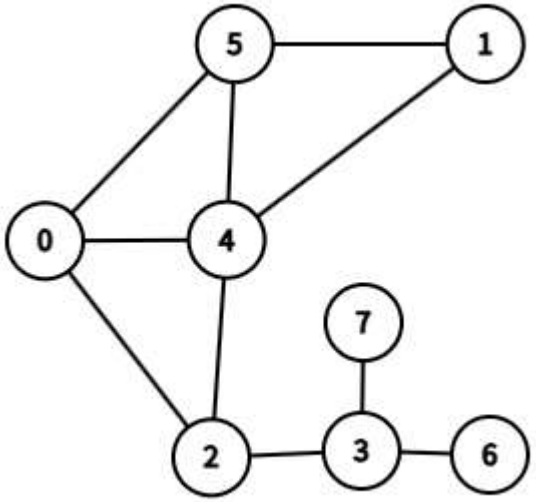
  从起点开始，**先访问所有直接邻居** → 再访问邻居的邻居（逐层扩散）

  像水波纹：从中心一圈圈向外蔓延

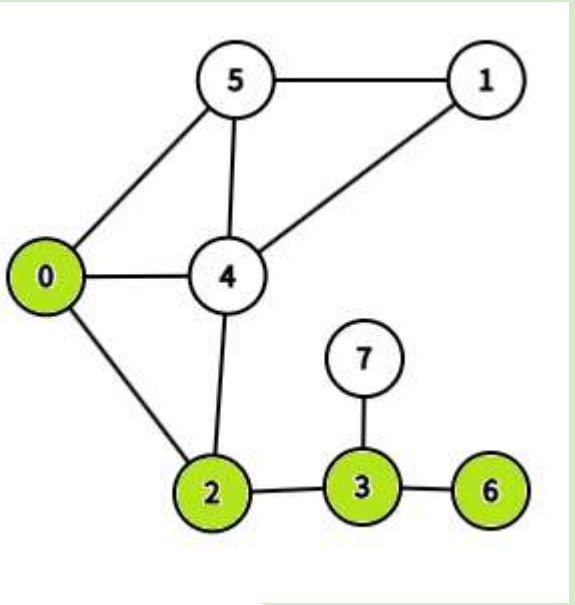
  值得注意的是，如果每一步的代价都一致，由于BFS是一层一层向外扩张的，所以第一次到达某点时的距离必然是其**最短距离**

深度优先遍历（DFS）

- 我们来对于右图进行dfs，这里我们以 0 号作为起始节点，并且在每次遍历时总是选择编号较小的后继节点。
- 于是我们首先进入 0 号节点，此时后继节点均未被进入，于是我们选择进入 2 号节点
- 进入 2 号节点后，我们选择编号最小的 3 号节点作为后继
- 进入 3 号节点后，我们选择编号最小的 6 号节点作为后继



深度优先遍历（DFS）



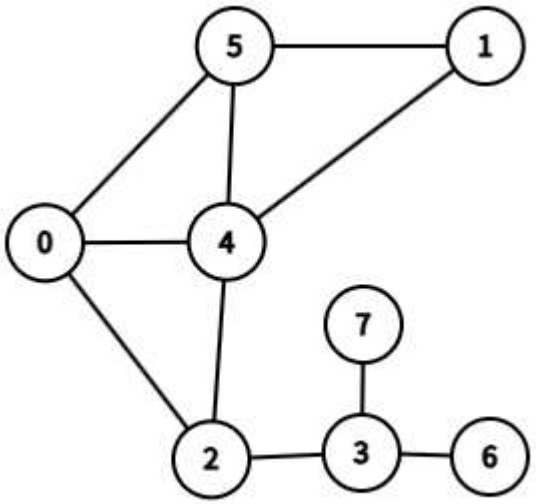
- 当我们访问完 6 号节点后，我们会发现其没有可以访问的后继节点了，于是此时我们应当回到上一个分支，也就是 3 号节点
- 回到 3 号节点后，选择 7 号节点作为后继
- 进入 7 号点后，同样其没有可以访问的后继了，回到上一个分支，也就是 2 号节点
- 回到 2 号节点后，选择 4 号节点作为后继
- 进入 4 号节点后，我们选择 1 号节点作为后继
- 进入 1 号节点后，我们选择 5 号节点作为后继
- 进入 5 号节点之后，没有可以访问的后继，一直回溯均无分支
- 遍历完成，遍历顺序为 0,2,3,6,7,4,1,5

深度优先遍历（DFS）

DFS过程可以简单表达为以下代码

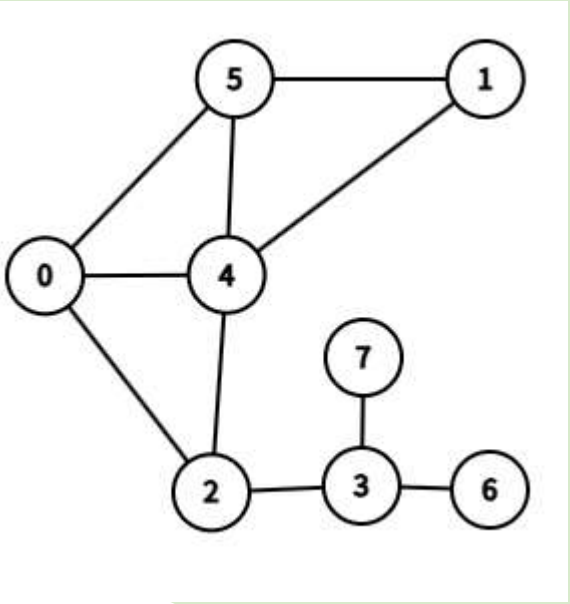
```
vector<int> neighbor[MAXN];
int visited[MAXN];
void dfs(int now) {
    visited[now] = 1;
    for (int i = 0; i < neighbor[now].size(); i++) {
        int nxt= neighbor[now][i];
        if (!visited[nxt]) {
            dfs(nxt);
        }
    }
}
```

广度优先遍历（BFS）



- 现在我们来对于右图进行bfs，同样以 0 号作为起始节点，并且在每次将后继入队时总是按编号从小到大的顺序。
- 于是我们首先进入 0 号节点，此时有三个后继节点均未被进入，将其均加入待访问队列
- 此时待访问队列为 [2,4,5]，故进入 2 号节点并弹出队首，将其未在队列的后继 3 加入队列中，继续下一步
- 此时待访问队列为 [4,5,3]，故进入 4 号节点并弹出队首，将其未在队列的后继 1 加入队列中，继续下一步
- 此时待访问队列为 [5,3,1]，故进入 5 号节点并弹出队首，其无未在队列中的后继

广度优先遍历（BFS）



- 此时待访问队列为 [3,1]，故进入 3 号节点并弹出队首，将其未在队列的后继 6,7 入队
- 此时待访问队列为 [1,6,7]，故进入 1 号节点并弹出队首，其无未在队列的后继
- 此时待访问队列为 [6,7]，故进入 6 号节点并弹出队首，其无未在队列的后继
- 此时待访问队列为 [7]，故进入 7 号节点并弹出队首，其无未在队列的后继
- 遍历完毕，遍历顺序为 0,2,4,5,3,1,6,7

广度优先遍历（BFS）

BFS过程可以简单表达为以下代码

```
vector<int> neighbor[MAXN];
int detected[MAXN];
queue<int> q;
void bfs(int start) {
    detected [start] = 1; //标记入队
    q.push(start); // 初始节点入队
    while (!q.empty()) {
        int now = q.front(); // 取出队首元素
        q.pop();
        for (int nxt : neighbor[now]) {
            if (!detected[nxt]) { // 如果未入队
                detected[nxt] = 1; // 标记已入队
                q.push(nxt); // 入队
            }
        }
    }
}
```

P1746 离开中山路

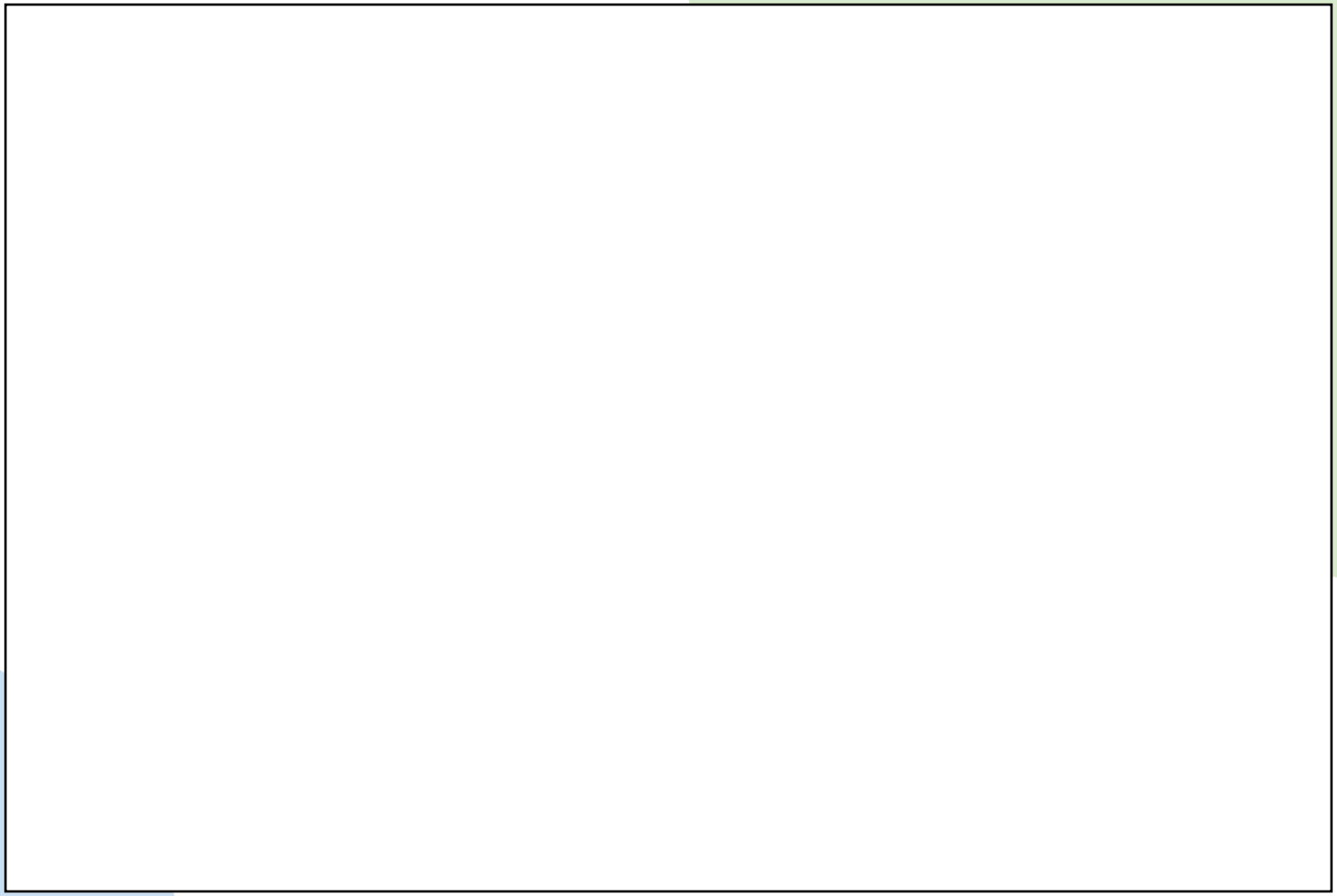
给定一个 $n \times n$ 的地图，其中 0 表示空地，1 表示障碍。每步只能转移到当前位置上下左右中的空地。起始位置在 x_1, y_1，目的地在 x_2, y_2，问从起始位置到目的地的最少步数

$1 \leq n \leq 10^3$

P1746 离开中山路

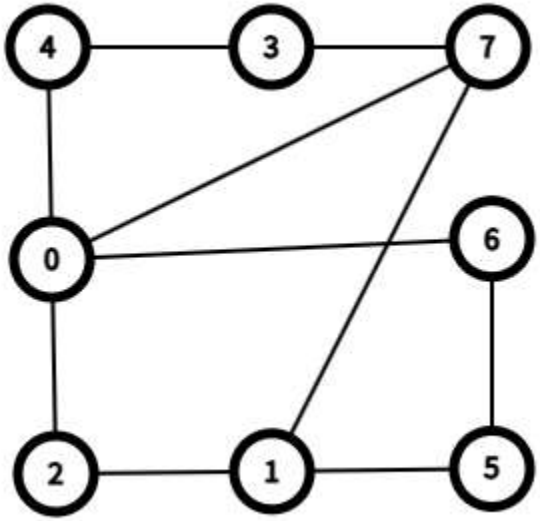
- 在这一道题中我们可以注意到，我们每一步的代价都是一致的（都是1），这启发我们利用BFS。因为BFS在代价一致时第一次到达的距离即为最短距离。
- 具体来说我们只需要在遍历的同时记录距离数组dis，每次在 now 节点将新节点 nxt 入队时同时更新其距离 $dis[nxt] = dis[now] + 1$ 即可。
- 在探索新节点时注意是否为空地，以及是否超出地图范围

P1746 离开中山路



试一试

- 请分别计算出右图的dfs遍历顺序和bfs遍历顺序，其中 0 号节点为初始节点。
- 注意dfs遍历时总是选择当前节点编号较小的后继节点继续访问。
- 同时，bfs遍历将后继入队时总是按编号从小到大的顺序。



试一试

- 请分别计算出右图的dfs遍历顺序和bfs遍历顺序，其中 0 号节点为初始节点。
- 注意dfs遍历时总是选择当前节点编号较小的后继节点继续访问。
- 同时，bfs遍历将后继入队时总是按编号从小到大的顺序。
- dfs遍历顺序：0,2,1,5,6,7,3,4
- bfs遍历顺序：0,2,4,6,7,1,3,5

- 请分别计算出右图的dfs遍历顺序和bfs遍历顺序，其中 0 号节点为初始节点。
- 注意dfs遍历时总是选择当前节点编号较小的后继节点继续访问。
- 同时，bfs遍历将后继入队时总是按编号从小到大的顺序。
- dfs遍历顺序：0,2,1,5,6,7,3,4
- bfs遍历顺序：0,2,4,6,7,1,3,5

树

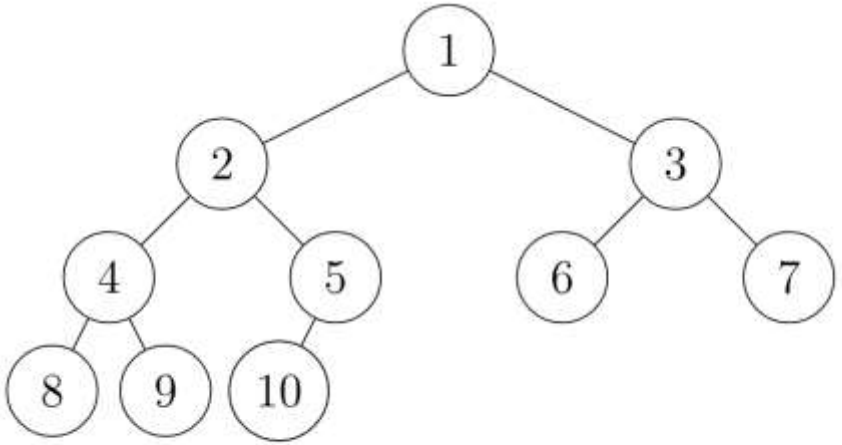
什么是树？

- 树是一种特殊的图，一个没有固定根结点的树称为 **无根树**，其有以下等价定义：
- 有 n 个节点，$n-1$ 条边的无向连通图。（这也表明了一棵树的点数恰好比边数多 1）
- 无环的无向连通图
- 任意两个结点之间有且仅有一条简单路径的无向图

- 在无根树的基础上，指定一个结点称为 **根**，则形成一棵 **有根树**

二叉树

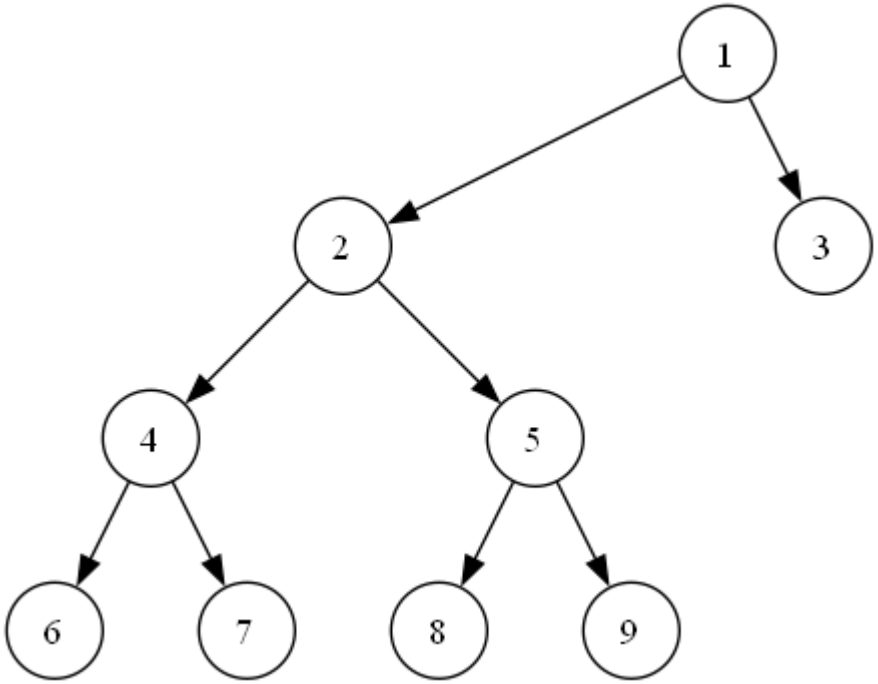
- 每个结点最多只有两个儿子（子结点）的有根树称为二叉树。常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点。
- **完全二叉树（complete binary tree）**：只有最下面两层结点的度数可以小于 2，且最下面一层的结点都集中在该层最左边的连续位置上。

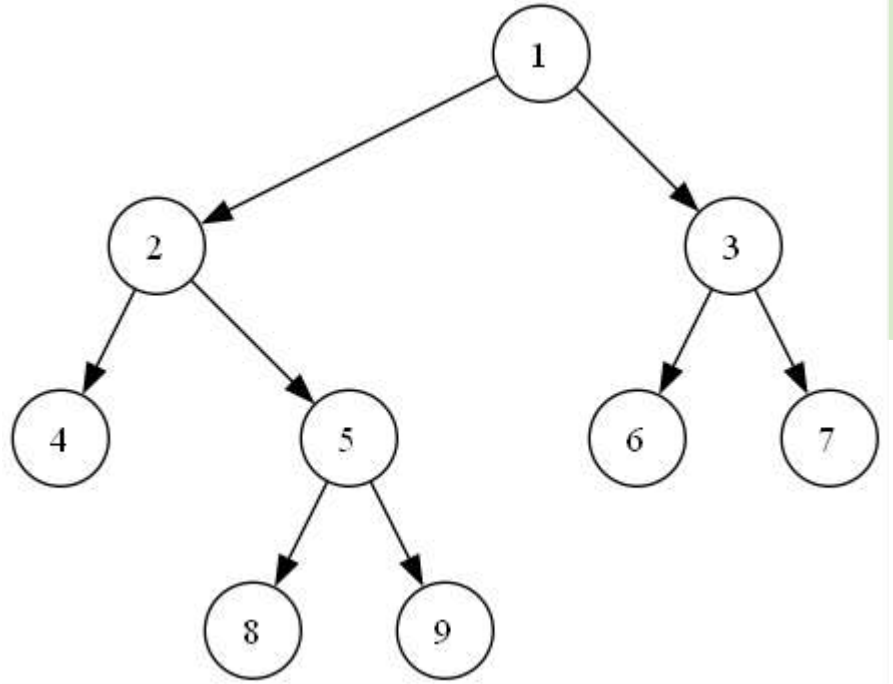


完全二叉树（complete binary tree）

试一试

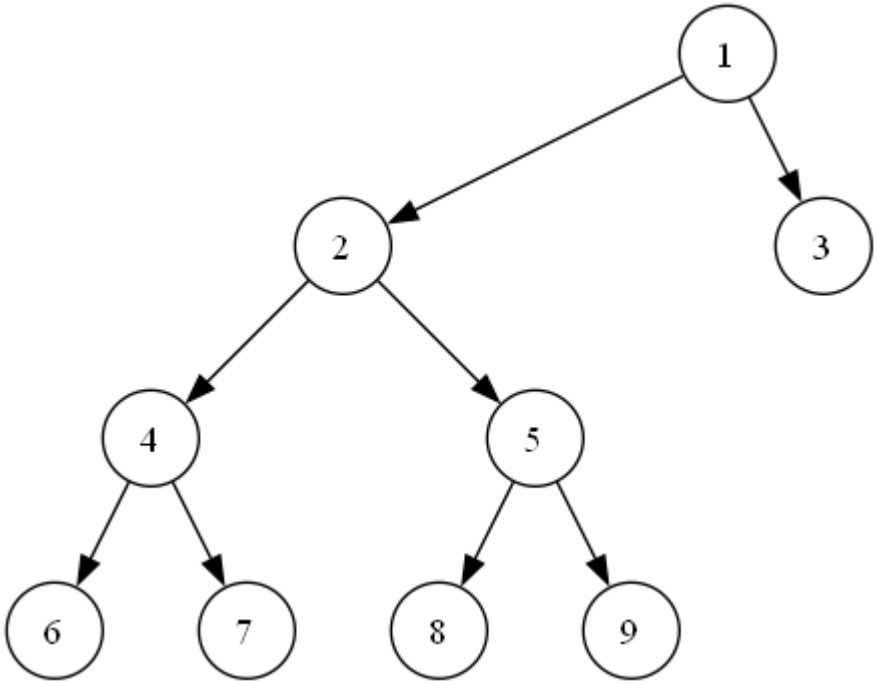
- 请判断以下两棵树是否为完全二叉树

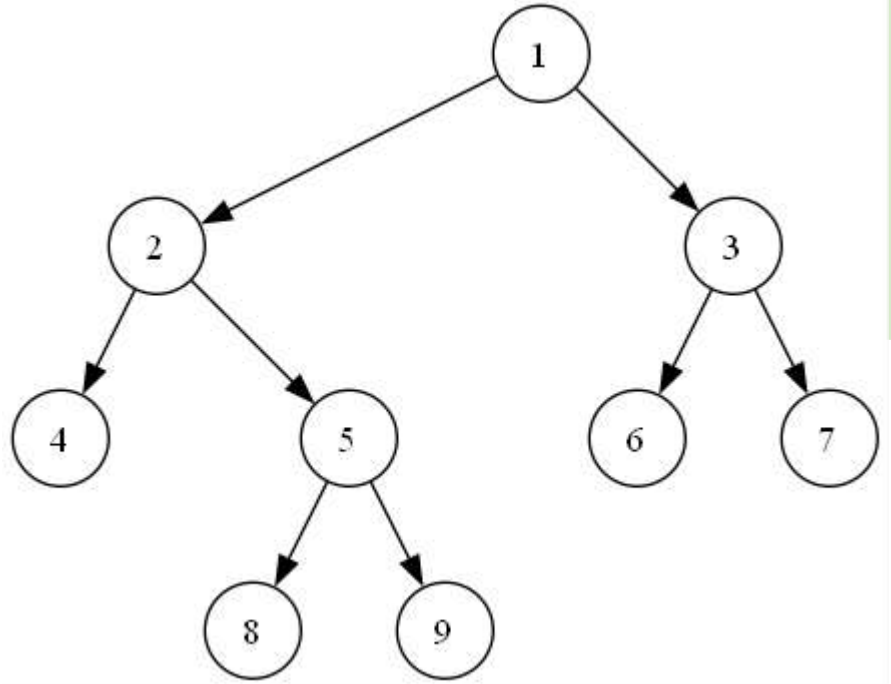




试一试

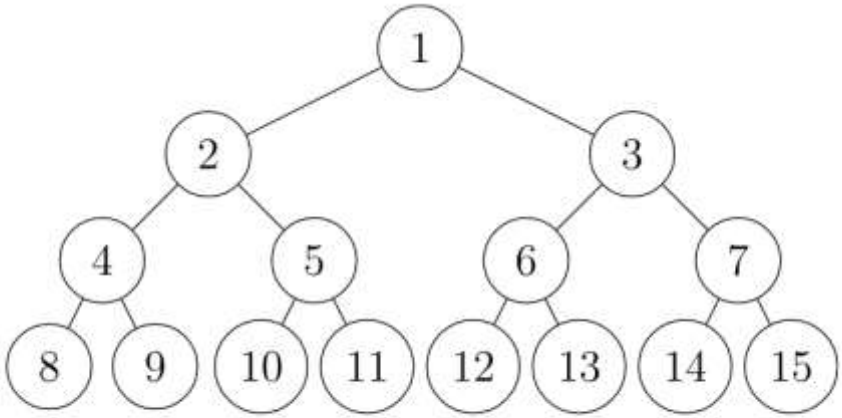
- 都不是!
- 左图中 3 并非最下两层的节点，但是其儿子数为 0
- 右图中最下一层的节点不集中在最左侧





二叉树

- 每个结点最多只有两个儿子（子结点）的有根树称为二叉树。常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点。
- **完美二叉树（perfect binary tree，国内也称为满二叉树）**：所有叶结点的深度均相同，且所有非叶节点的子节点数量均为 2 的二叉树称为完美二叉树。



完美二叉树（perfect binary tree）

二叉搜索树

二叉搜索树（又叫二叉排序树）是一种二叉树的树形数据结构，其定义如下：

- 空树是二叉搜索树。
- 若二叉搜索树的左子树不为空，则其左子树上所有点的附加权值均小于其根节点的值。
- 若二叉搜索树的右子树不为空，则其右子树上所有点的附加权值均大于其根节点的值。
- 二叉搜索树的左右子树均为二叉搜索树。

二叉搜索树

- 考虑在二叉搜索树进行搜索，不妨设搜索 x，当前节点的值为 $root$
- 如果 $x == root$ 说明已找到，结束
- 如果 $x < root$，由定义可知，x 必然在左子树中，递归进入左子树进行搜索
- 如果 $x > root$，则 x 必然在右子树中，递归进入右子树进行搜索
- 如果该节点没有要去的左/右子树，则说明要搜索的值不存在

二叉搜索树上的搜索所花费的时间与这棵树的高度成正比。对于一个有 n 个结点的二叉搜索树中，搜索的最优时间复杂度为 $O(\log n)$ 最坏复杂度为 $O(n)$

哈夫曼树

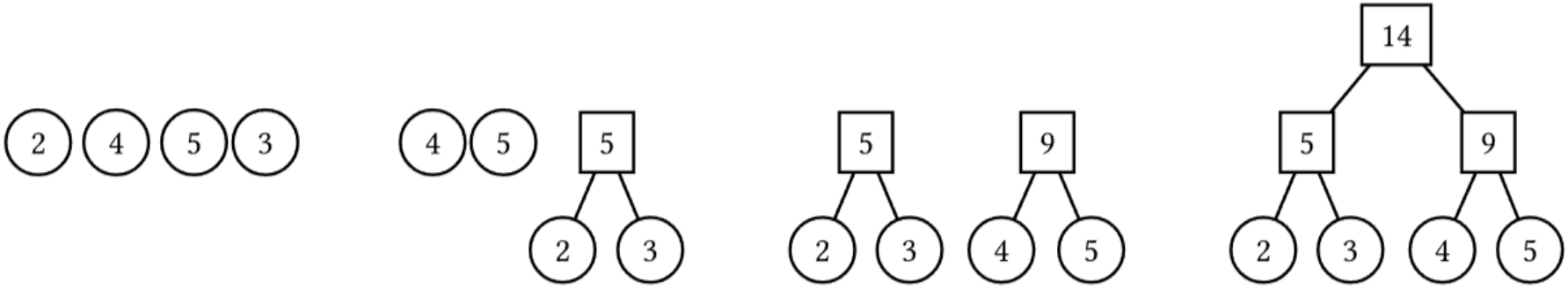
设二叉树具有 n 个带权值的叶节点，从根节点到各叶节点的路径长度与对应叶节点权值的乘积之和为该树的带权路径长度，或者可以表达为以下形式

$\text{weight} = \Sigma_{x \in leaves} w_x \times len_x$，其中 w_x 为叶子的权值，len_x 为该叶子到该根节点的距离

而对于固定的叶节点权值，哈夫曼树是所有带权二叉树中带权路径长度最小的

哈夫曼树

- 哈夫曼树的构造方式如下：
- 最初时分别为 n 个独立的只有一个节点的二叉树，每个二叉树的根的权值即为给定的叶子权值
- 每次选择根节点权值最小的两棵二叉树，将其分别作为左右子树连接到一个新根节点从而形成一个新的二叉树，新根节点的权值为其左右两儿子的和。（并把原本的两树删除）
- 重复上一步，直到只剩一棵树，此时根的权值即为最小带权路径和。

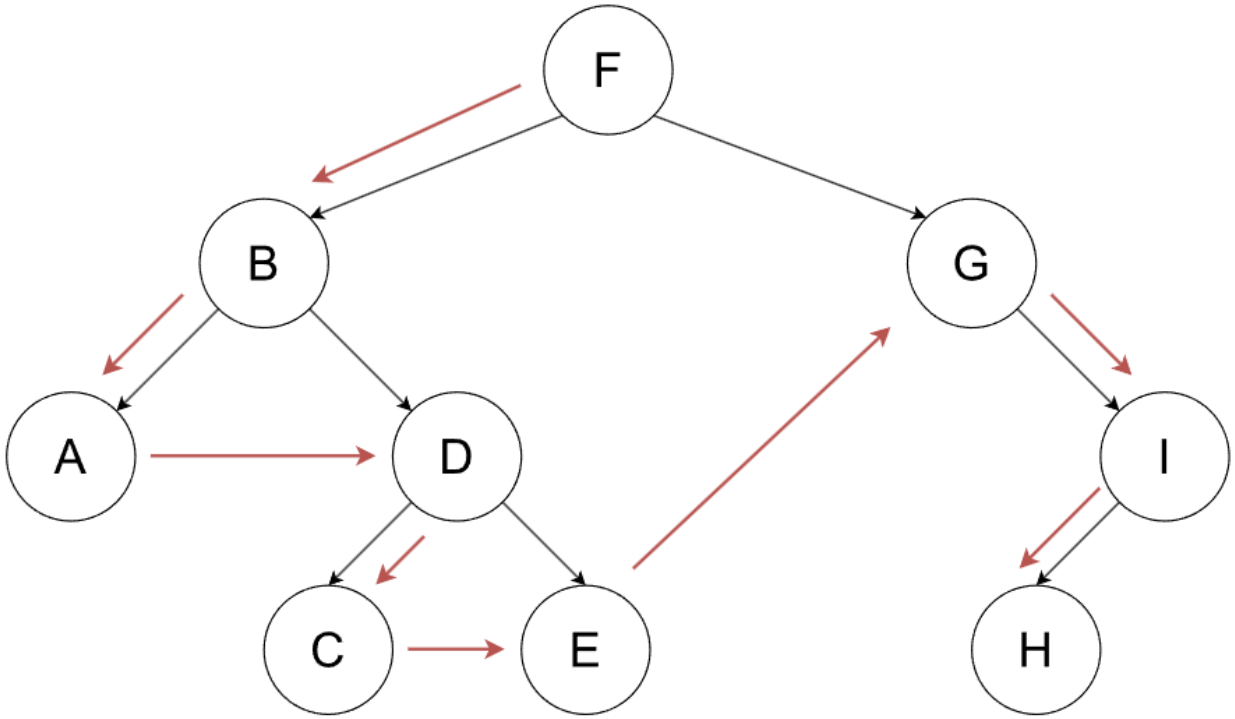


二叉树 DFS 遍历

- 既然二叉树本质是一个图，我们自然可以在二叉树上进行DFS。但是值得注意的是，我们要区分开进入一个节点，和访问一个节点的信息。我们可以进入一个点后不立即访问它的信息，而是进入其儿子并返回之后（遍历完左子树/右子树）再访问。
- 于是我们可以按访问该节点信息相对进入左儿子、进入右儿子的顺序来分为先序遍历、中序遍历、后序遍历。同样，将访问节点信息的顺序得到先序遍历序列、中序遍历序列、后序遍历序列

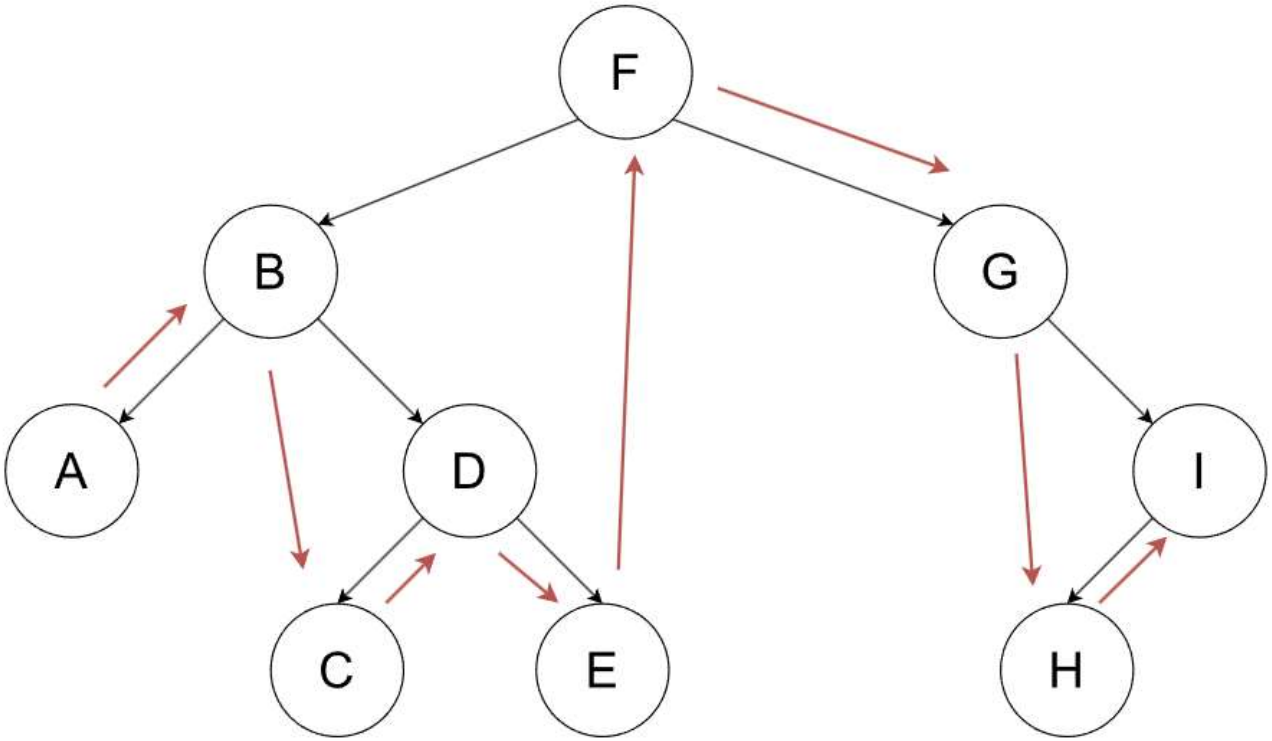
先序遍历

- 先访问节点信息，再进入左儿子，回到该节点之后进入右儿子
- 例如可得如下遍历，先序遍历序列为 $F, B, A, D, C, E, G, I, H$



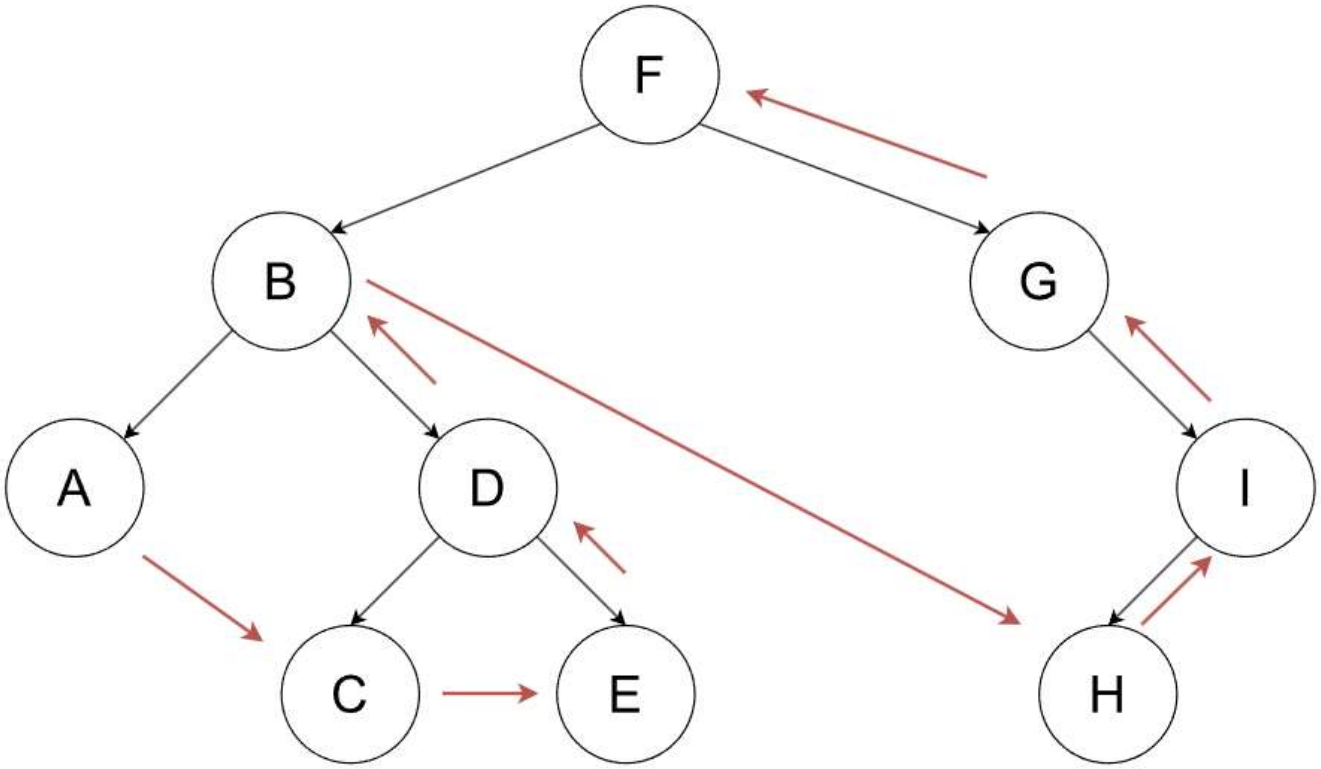
中序遍历

- 先进入左儿子，回到该节点之后（遍历完左子树）再访问节点信息，最后进入右儿子
- 例如可得如下遍历，中序遍历序列为 $A, B, C, D, E, F, G, H, I$



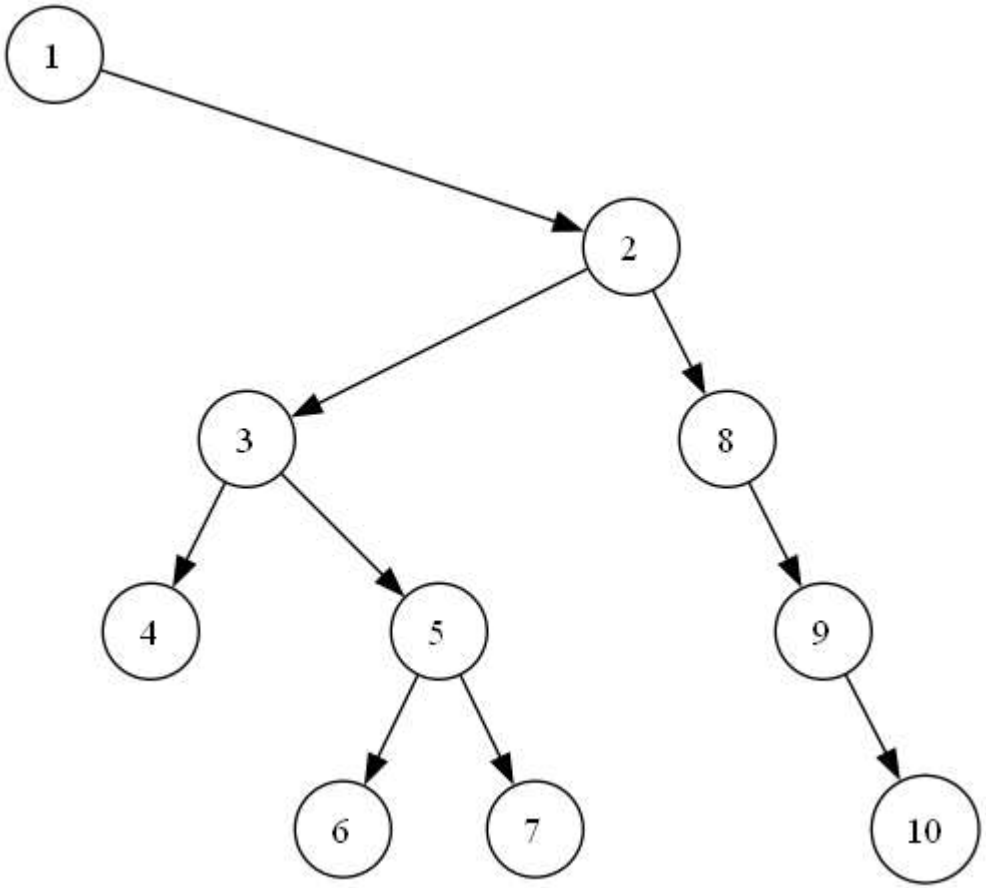
后序遍历

- 先进入左儿子，回到该节点之后进入右儿子，最后回到该节点（遍历完了左右子树）再访问节点信息
- 例如可得如下遍历，后序遍历序列为 $A, C, E, D, B, H, I, G, F$



试一试

- 请在评论区写出以下树的先序遍历/中序遍历/后续遍历序列



试一试

- 先序遍历：1,2,3,4,5,6,7,8,9,10
- 中序遍历：1,4,3,6,5,7,2,8,9,10
- 后序遍历：4,6,7,5,3,10,9,8,2,1

- 先序遍历：1,2,3,4,5,6,7,8,9,10
- 中序遍历：1,4,3,6,5,7,2,8,9,10
- 后序遍历：4,6,7,5,3,10,9,8,2,1

B3642 二叉树的遍历

有一个 n $(n \leq 10^6)$ 个结点的二叉树。给出每个结点的两个子结点编号（均不超过 n），建立一棵二叉树（根节点的编号为 1），如果是叶子结点，则输入 0 0。

建好这棵二叉树之后，依次求出它的前序、中序、后序列遍历。

B3642 二叉树的遍历

- 我们可以首先写出一个普通的dfs，遍历这棵二叉树（注意先遍历左子树再遍历右子树）
- 这时我们来考虑如何求出先序/中序/后序。正如我们前面的定义，先序/中序/后序的区别只在于访问当前节点与遍历左子树、遍历右子树之间的顺序。(我们只需在所谓的访问该节点时将该节点的编号加入对应的序列即可
- 对于先序，我们有：访问now（preorder.push_back(now))、遍历左子树、遍历右子树
- 对于中序，我们有：遍历左子树、访问now、遍历右子树
- 对于后序，我们有：遍历左子树、遍历右子树、访问now

B3642 二叉树的遍历

反推

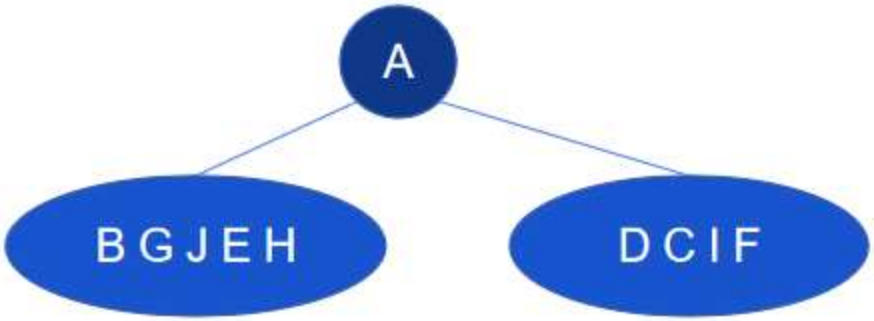
- 如果我们已知中序遍历序列和另外一个序列，如何反推这棵树呢
- 我们注意到先序遍历时的第一个点必然是根。同样的，后序遍历的最后一个点也必然是根，这样我们就确定了根的编号，不妨设为 rt 。返回到中序遍历序列中，在序列中位于 rt 左侧的必然是根的左子树，同样的，在 rt 右侧的必然是右子树。这样我们就将问题划分到了子树中，继续上述过程即可。

反推

- 比如先序遍历的顺序为：A G B E J H C D F I 。

  中序遍历的顺序为：B G J E H A D C I F 。

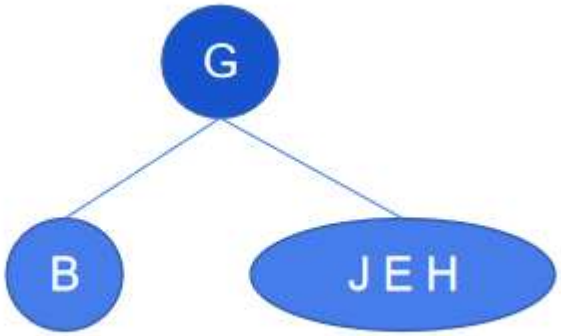
- 我们首先可以确认根为 A，回到中序遍历中可得 B G J E H 为左子树，D C I F 为右子树



反推

- 比如先序遍历的顺序为：A G B E J H C D F I 。

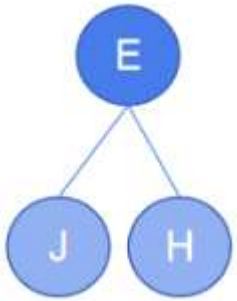
  中序遍历的顺序为：B G J E H A D C I F 。
- 我们首先可以确认根为 A，回到中序遍历中可得 B G J E H 为左子树，D C I F 为右子树
- 再下一步，B G J E H 中再先序序列中出现最早的为 G，故 G 为该子树的根，回到中序序列中 B 在 G 左侧，J E H 在 G 右侧，故该子树应该形如：



反推

- 比如先序遍历的顺序为：A G B E J H C D F I 。

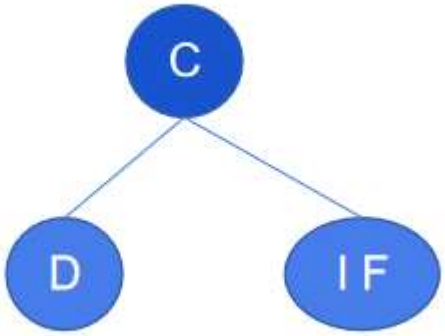
  中序遍历的顺序为：B G J E H A D C I F 。
- 我们首先可以确认根为 A，回到中序遍历中可得 B G J E H 为左子树，D C I F 为右子树
- 再下一步，B G J E H 中再先序序列中出现最早的为 G，故 G 为该子树的根，回到中序序列中 B 在 G 左侧，J E H 在 G 右侧。
- 进一步确认 J E H 的关系，发现 E 在先序中出现最早，故 E 为根。回到中序中 J 在 E 左侧，H 在 E 右侧，故有如图形式：



反推

- 比如先序遍历的顺序为：A G B E J H C D F I 。

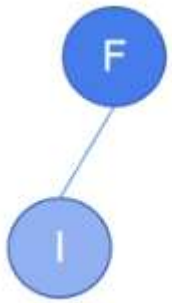
  中序遍历的顺序为：B G J E H A D C I F 。
- 我们首先可以确认根为 A，回到中序遍历中可得 B G J E H 为左子树，D C I F 为右子树
- 同样的我们来处理右子树 D C I F，其中 C 在先序遍历中出现最早，故 C 为该子树的根，D 在左子树，I F 在右子树，如下



反推

- 比如先序遍历的顺序为：A G B E J H C D F I 。

  中序遍历的顺序为：B G J E H A D C I F 。
- 我们首先可以确认根为 A，回到中序遍历中可得 B G J E H 为左子树，D C I F 为右子树
- 同样的我们来处理右子树 D C I F，其中 C 在先序遍历中出现最早，故 C 为该子树的根，D 在左子树。
- 最后处理 I F，F 在先序遍历出现最早，故 F 为根，I在其左子树，如下：

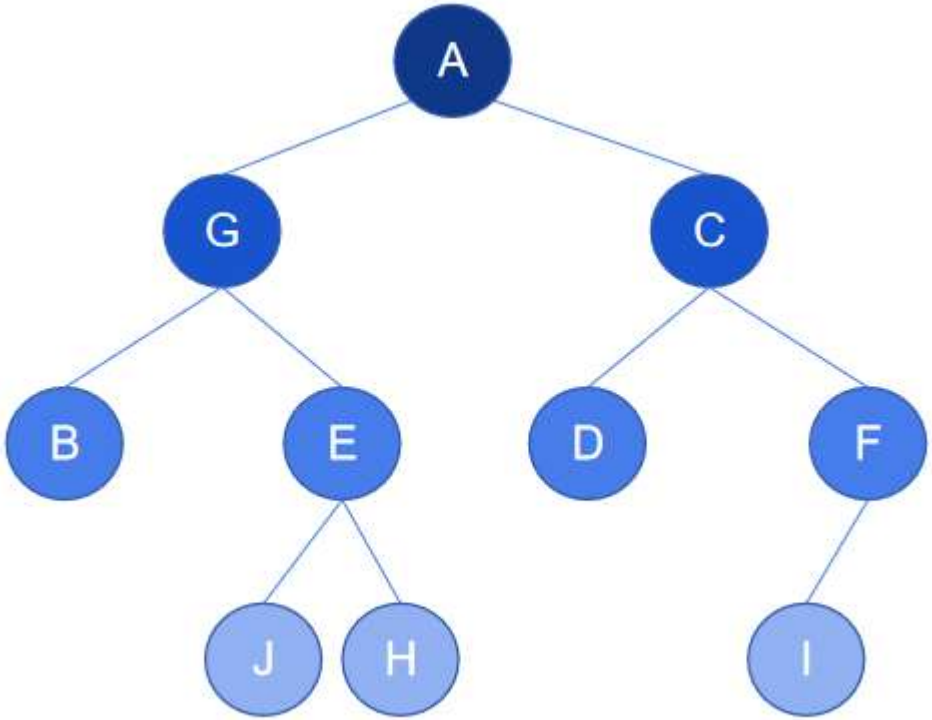


反推

- 比如先序遍历的顺序为：A G B E J H C D F I 。

  中序遍历的顺序为：B G J E H A D C I F 。

- 于是得到这棵树即为：



试一试

1. 若一棵二叉树的先序遍历为：A, B, D, E, C, F、中序遍历为：D, B, E, A, F, C，它的后序遍历为（ ）。

a) D, E, B, F, C, A

b) E, D, B, F, C, A

c) D, E, B, C, F, A

d) E, D, B, C, F, A

试一试

1. 若一棵二叉树的先序遍历为：A, B, D, E, C, F、中序遍历为：D, B, E, A, F, C，它的后序遍历为（ ）。

a) D, E, B, F, C, A

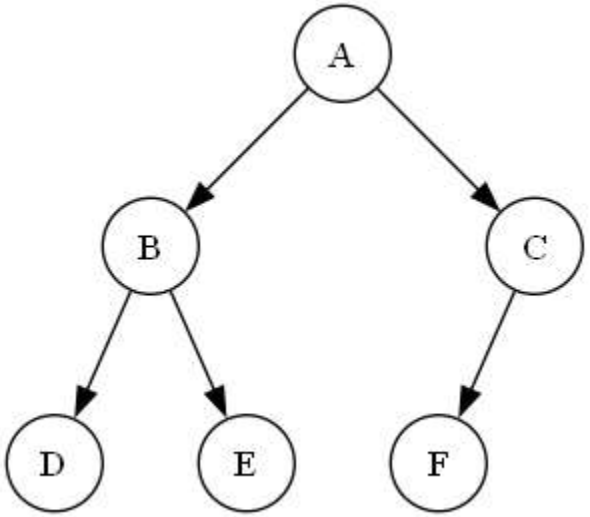
b) E, D, B, F, C, A

c) D, E, B, C, F, A

d) E, D, B, C, F, A

Answer：a

可反推二叉树如右图



试一试

2. 以下代码实现了树的哪种遍历方式?

A. 前序遍历

B. 中序遍历

C. 后序遍历

D. 层次遍历

```
void traverse(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";
    traverse(root->left);
    traverse(root->right);
}
```

注解：这里的 root 类型为 TreeNode* 即为一个结构体指针，可简单理解为一个图书馆中书籍的位置编号。我们取结构体中的元素，比如 TreeNode tmp，时使用 tmp.val 。而当我们使用指针取元素时就把 . 替换成 ->，也即代码中的 root->val

试一试

2. 以下代码实现了树的哪种遍历方式?

A. 前序遍历

B. 中序遍历

C. 后序遍历

D. 层次遍历

```
void traverse(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";
    traverse(root->left);
    traverse(root->right);
}
```

Answer: A

重点关注 cout << root->val << " "; 位置，其先于遍历左子树故为前序遍历

试一试

3. 以下关于树的说法，（ ）是正确的。

A. 在一棵二叉树中，叶子结点的度一定是2。

B. 满二叉树中每一层的结点数等于 $O(2^{层数-1})$。

C. 在一棵树中，所有结点的度之和等于所有叶子结点的度之和。

D. 一棵二叉树的先序遍历结果和中序遍历结果一定相同。

试一试

3. 以下关于树的说法，（）是正确的。

A. 在一棵二叉树中，叶子结点的度一定是2。

B. 满二叉树中每一层的结点数等于 $O(2^{层数-1})$。

C. 在一棵树中，所有结点的度之和等于所有叶子结点的度之和。

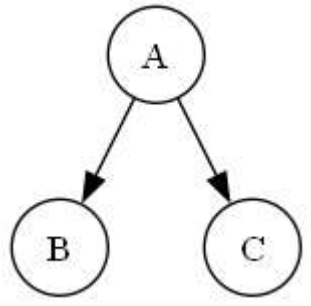
D. 一棵二叉树的先序遍历结果和中序遍历结果一定相同。

Answe：B

A:叶子节点的度一定是0

C:易举反例 $1 \to 2 \to 3 \to 4 \to 5$

D:易举反例如右图



试一试

判断题

1. 完全二叉树的任意一层都可以不满。

2. 二叉搜索树的查找操作的时间复杂度是 $\log n$。

3. 在树的深度优先搜索（DFS）中，使用队列作为辅助数据结构以实现“先进后出”的访问顺序。

4. 二叉排序树（BST）中，若某节点的左子树为空，则该节点一定是树中的最小值节点。

试一试

判断题

1. 完全二叉树的任意一层都可以不满。

错，只有最下一层可以不满

2. 二叉搜索树的查找操作的时间复杂度是 $O(\log n)$。

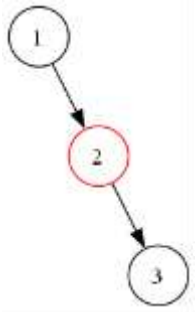
错，比如对于一个退化为链的二叉搜索树，搜索复杂度为 $O(n)$

3. 在树的深度优先搜索（DFS）中，使用队列作为辅助数据结构以实现“先进后出”的访问顺序。

错，使用队列辅助的为BFS

4. 二叉排序树（BST）中，若某节点的左子树为空，则该节点一定是树中的最小值节点。

错误，只能说明其是该子树内的最小值



B3622 枚举子集

今有 n 位同学，可以从中选出任意名同学参加合唱。

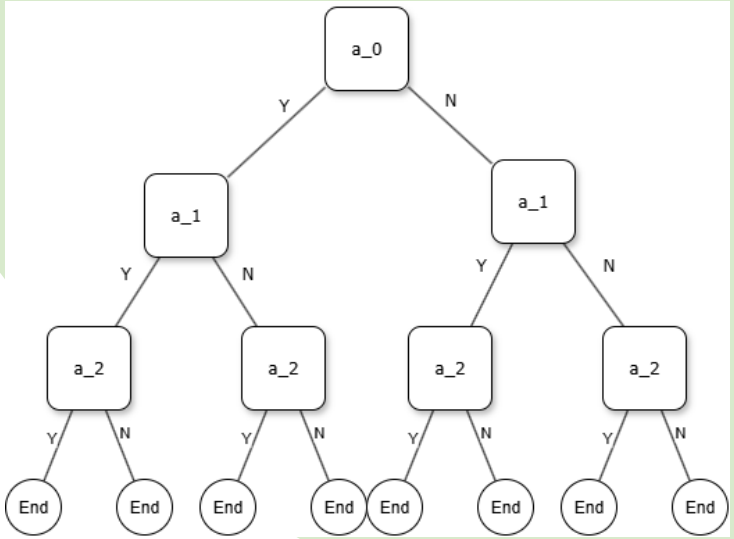
请输出所有可能的选择方案。

每一种选择方案用一个字符串表示，其中第 i 位为 Y 则表示第 i 名同学参加合唱；为 N 则表示不参加。

$1 \leq n \leq 10$

B3622 枚举子集

- 这里我们使用dfs来实现枚举子集。
- 对于枚举子集，我们只需将DFS过程中枚举下一条边的步骤替换成枚举这一个元素选或不选（N/Y)
- 直到枚举完所有的位置（这时候我们就已经得到了一个完整的安排，可以输出了



- 在输出一个位置之后，我们自然的回溯到上一个没有探索完的分支节点，继续枚举该位置的下一个选择

B3622 枚举子集

B3623 枚举排列（递归实现排列型枚举）

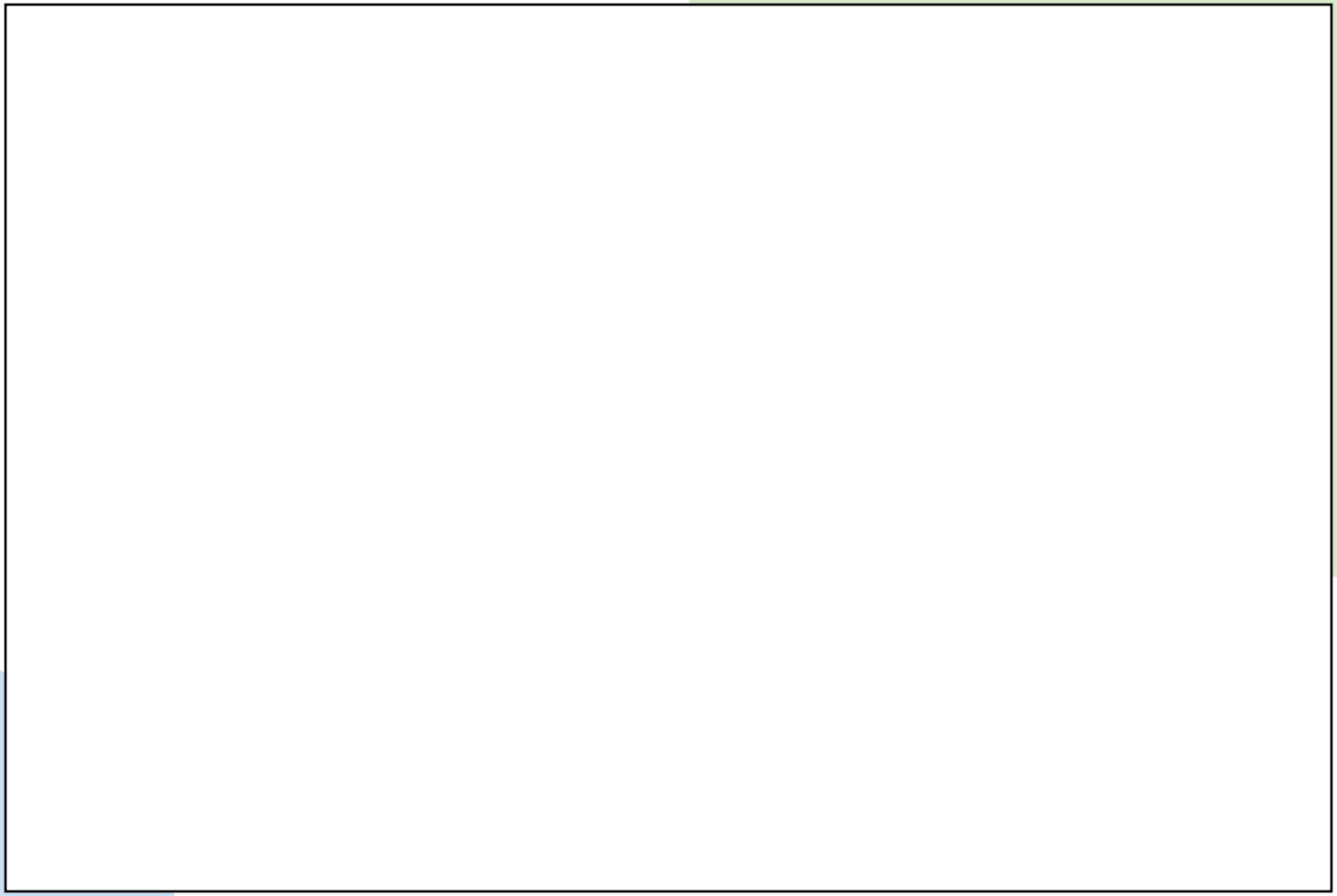
今有 n 名学生，要从中选出 k 人排成一列拍照。

请按字典序输出所有可能的排列方式。

B3623 枚举排列（递归实现排列型枚举）

- 同样使用DFS的方法，和枚举子集的过程一样，我们在每次枚举一个位置选择的人，然后递归枚举下一个位置的选择，直到所有位置都确定，我们就可以输出，之后自然回溯到之前的位置枚举下一个选择。
- 值得注意的是，这里我们多了一个限制，也即不能重复选择同一个人。于是我们用一个vis数组记录选择了哪些人，注意我们在选择时将vis数组中对应位置标为 1 。同时在选择下一个人/回溯的时候一定记得把对应vis数组重置。

B3623 枚举排列（递归实现排列型枚举）



P10448 组合型枚举

从 $1 \sim n$ 这 n 个整数中随机选出 m 个

输出所有可能的选择方案。

P10448 组合型枚举

- 同样和枚举排列一样，我们用一个vis辅助DFS枚举。但此时注意到组合其实不关心相对位置的关系，比如说在枚举排列中 1,2,3 和 1,3,2 不是相同的排列，但在此处 1,2,3 和 1,3,2 显然是同一种组合。
- 于是我们可以强制选择出的排序必须是升序的，比如 1,2,3 或者 1,3,4，这样我们选出的一种方案必然一一对应一种组合。
- 具体来说，我们每次记录上一个位置放的数 lst，并且每次都从 $lst+1$ 开始枚举该位置的数即可。

P10448 组合型枚举