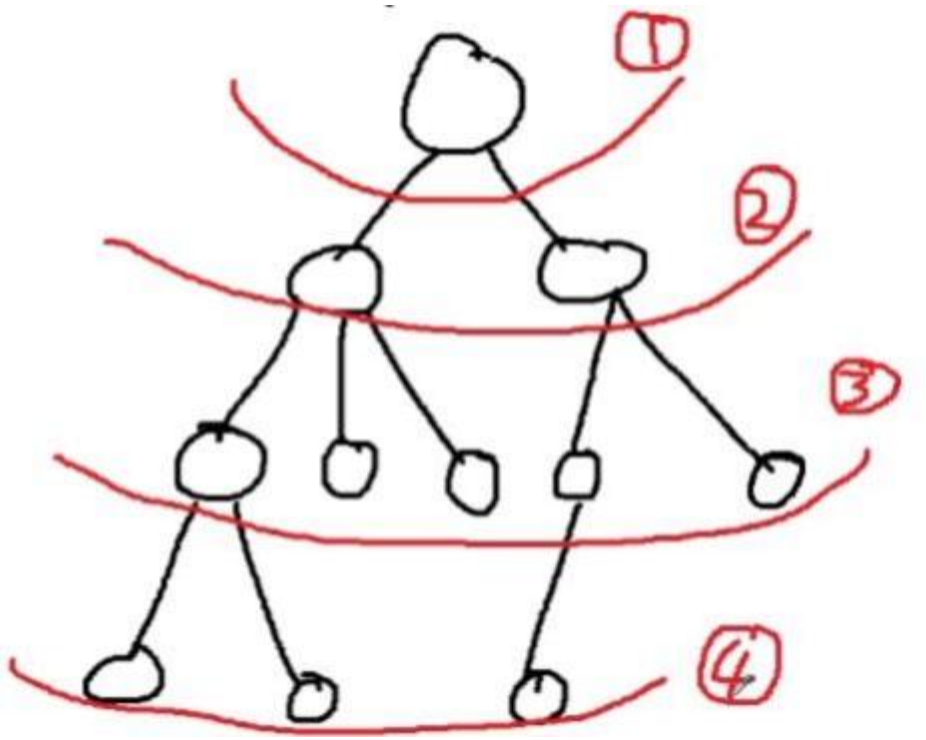
INT10 2W4 _ BFS - DFS

BFS——一石激起千层浪

全称： 广度优先搜索或者宽度优先搜索（使用图的查找算法，要求可以用图表现问题的关联性）

可解决的问题：
1、 从A出发到B的最短距离问题
2、 A是否存在到B的路径

整体思路： 其思路为从图上一个节点出发，访问先访问其直接相连的子节点，若子节点不符合，再问其子节点的子节点，按级别顺序(一层一层)依次访问，直到访问到目标节点。起始:将起点(源点，树的根节点)放入队列中。扩散:从队列中取出队头的结点，将它的相邻结点放入队列，不断重复这一步。终止:当队列为空时，说明我们遍历了所有的结点，整个图都被搜索了一遍，用简笔图表示则如下所示



图中我们可以看到我们是先搜索第一层再第二层以此类推。

注意点： 数的层数就是递归的层数， bfs通常使用递归实现

对比DFS的优缺点：用时间换空间，在时间上落后但是空间上极度简洁。

这里我们可以用一个走迷宫的例子来体现BFS的应用

给定一个nxm的二维整数数组，用来表示一个迷宫，数组中只包含0或1,其中0表示可以走的路，1表示不可通过的墙壁。

最初，有一个人位于左上角(1,1)处，已知该人每次可以向上、下、左、右任意一个方向移动一个位置。请问，该人从左上角移动至右下角(n,m)处，至少需要移动多少次。

数据保证(1,1)处和(n,m)处的数字为0,且一定至少存在一条通路。

输入格式

第一行包含两个整数n和m。接下来n行，每行包含m个整数(0或1),表示完整的二维数组迷宫。

输出格式

输出一个整数，表示从左上角移动至右下角的最少移动次数。

数据范围

1≤n,m≤100

那么我们来逐层分析如何实现他

首先0代表通路1代表闭路且一次只可通过一个单位的距离，目标是最少的移动次数，由于是从上至下，逐层筛选通路，我们可以直接应用BFS。我们先用人的逻辑来演示一下计算机在对应BFS的时候是如何进行最短路径的搜索的

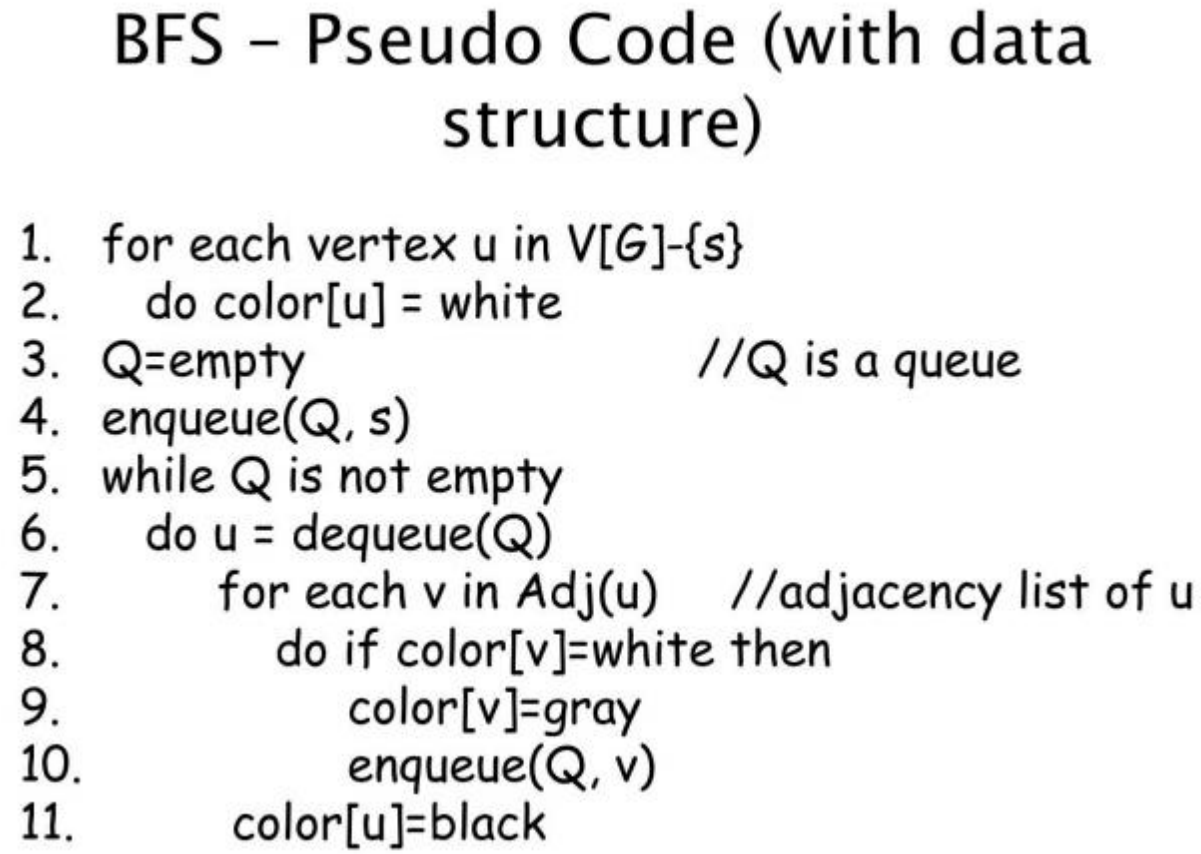
我们可以看到我这里是逐次标记距离原点距离最短的可用点，直到到达目标点为止。不过这个说法只能用在每条边权重相等的情况下，因为BFS找到的只是经过的路径数量而不是长度，存在加权则需要考虑路的长度的问题，可以用Dijkstra算法来解决，这里暂不讨论，我还是先讲这个例题，

现在我们拿代码来进行一下解释：

```
def bfs(): # 定义广度优先搜索函数
    d[0][0] = 0 # 将起点的距离设为0
    queue = [(0, 0)] # 创建一个队列，并将起点加入队列
    dx = [-1, 0, 1, 0] # 定义四个方向的x坐标偏移
    dy = [0, 1, 0, -1] # 定义四个方向的y坐标偏移


    while queue : # 当队列不为空时
        x, y = queue.pop(0) # 取出队列中的第一个元素
        for i in range(4): # 遍历四个方向
            a = x + dx[i]; # 计算新的x坐标
            b = y + dy[i]; # 计算新的y坐标
            # 如果新的坐标在迷宫内，且该位置是可达的，且该位置还未被访问过
            if a >= 0 and a < n and b >= 0 and b < m and g[a][b] == 0 and d[a][b] ==
-1:
                queue.append((a,b))# 将该位置加入队列
                d[a][b] = d[x][y] + 1 # 更新该位置的最短距离
print(d[n - 1][m - 1]) # 打印从起点到终点的最短距
```

如下是老师课件中的伪代码

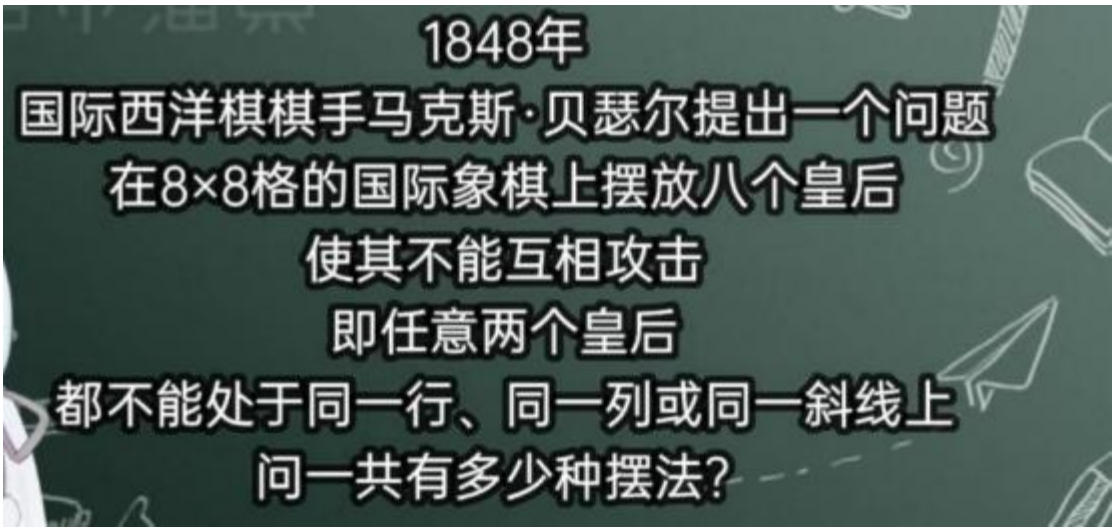


dfs---不撞南墙不回头

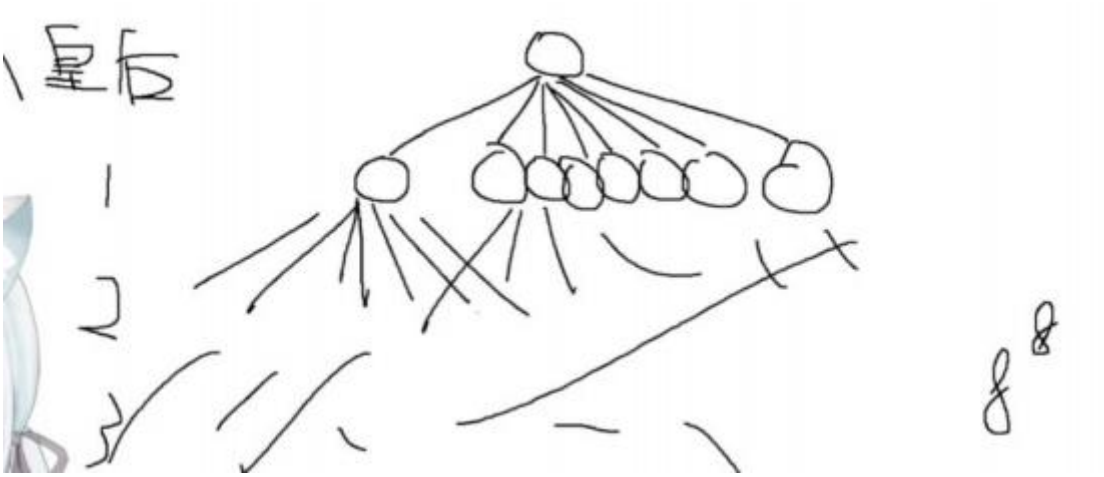
定义：深度优先搜索（DFS，Depth-First Search）是一种用于遍历或搜索树或图 的算法。这个算法会尽可能深地搜索树的分支，看上去或许只是和BFS存在方向的 差距，但是它对比BFS是多了一个回溯的步骤在里面。也就是每次当我们对树的某 一分支探查到底的时候返回到上一个存在多项选择的分支进行另一分支的探索。它 的时间复杂度也很高，在无剪枝的情况下通常被叫做暴力搜索，需要遍历整张图。 同样也是用递归的方法来实现（其实不如说深度搜索就是递归的另一种应用）

作用范围：解的个数，最多路径数量，最大路径长度等（可以理解为宽搜求下限深搜找上限？大概“ “）

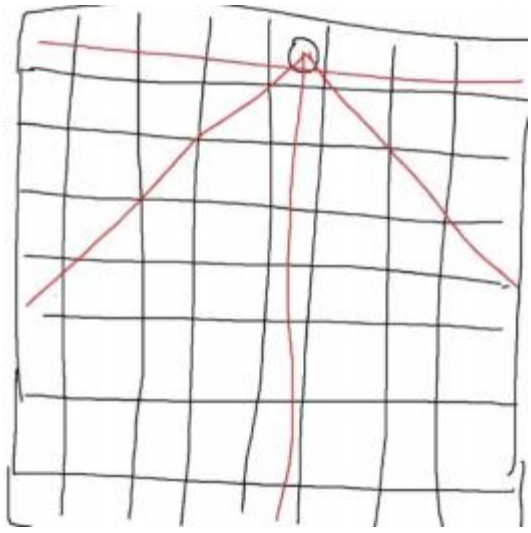
同样我们用一个例子来帮助大家理解深度搜索，就是传统的“八皇后”问题



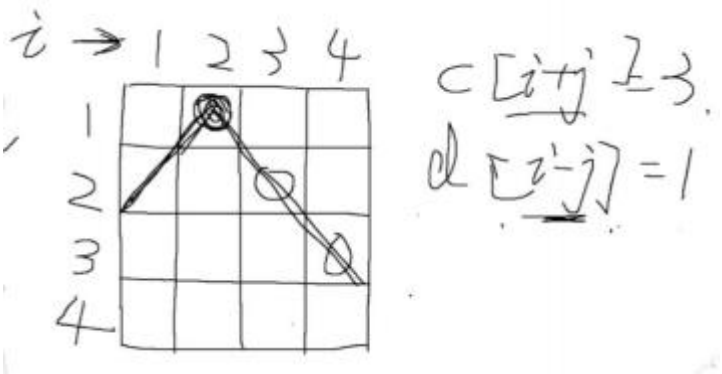
我们一样按照思路先进行，首先是没有限制的情况下会是什么样子呢？他会是一个八叉树，如下所示



最后体现出来的就是一个八叉树的样子，总共就会是8的八次方，但是我们有限制条件所以我们开始考虑限制条件对结果的影响。题中的限制条件是横斜纵不能相见，那我们就可以对他进行剪枝（一样用图来表示这个限制的条件）（剪枝的意思就是根据限制条件的内容来减少我们树的分叉，比如这里我们处理后八叉图就会变成4叉乃至二叉）



在这之后，我们可以设立一个数组a[i]表示第i行的皇后在a[i]列，设立一个数组b【i】来表示该列是否可用（布尔数组，ture表示通路）因为我们的皇后是按照行进行排列所以不存在行上面的冲突，在这之后加入c、d两个数组来表示对角线（i+j)和(i—j），理由如下



在二维平面上，我们可以通过坐标(i, j)来表示一个点，其中i表示行，j表示列。对于任何一个点(i, j)，它的对角线 上的点满足以下两个条件之一：

对于主对角线（左上到右下的对角线），所有的点满足条件：行坐标+列坐标相等，即i+j相等。这是因为在主对角线上，当我们向右下方移动时，行坐标和列坐标都会增加，所以它们的和保持不变。

对于副对角线（左下到右上的对角线），所有的点满足条件：行坐标-列坐标相等，即i-j相等。这是因为在副对角线上，当我们向右上方移动时，行坐标会减少，而列坐标会增加，所以它们的差保持不变。因此，我们可以通过(i+j)和(i-j) 来表示一个点是否在某条对角线上。在八皇后问题中，我们需要确保任何两个皇后都不在同一条对角线上，所以我们可 以使用这两个公式来检查对角线上是否已经有皇后。

自此，我们可以开始写我们的代码了

```
public class EightQueens {
    int N = 8;
    int[] a = new int[N]; // 表示第i行的皇后在a[i]列
    boolean[] b = new boolean[N]; // 表示该列是否可用
    boolean[] c = new boolean[2 * N]; // 表示对角线（i+j)是否可用
    boolean[] d = new boolean[2 * N]; // 表示对角线（i-j)是否可用
    int count = 0; // 记录解决方案的总数

    void search(int i) {
        for (int j = 0; j < N; j++) {
            if (!b[j] && !c[i + j] && !d[i - j + N]) { // 检查列和对角线是否可用 a[i] = j;
                // 放置皇后
                if (i == N - 1) count++; // 如果所有皇后都放置好了，增加解决方案的
总数
                else {
                    b[j] = c[i + j] = d[i - j + N] = true; // 标记列和对角线为不可 用
                    search(i + 1); // 继续放置下一行的皇后
                    b[j] = c[i + j] = d[i - j + N] = false; // 回溯，取消标记 }
            }
        }
    }
```