搜索

湖南师大附中 许力

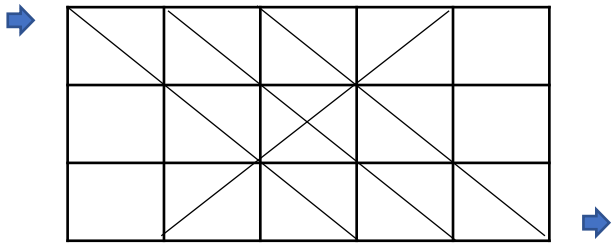
双端队列BFS

- deque是STL中的一种特殊队列：和普通队列queue在队首删除，队尾插入不同的是，它的队首、队尾均支持插入/删除操作

- 那么BFS中使用deque取代queue有什么妙用呢？

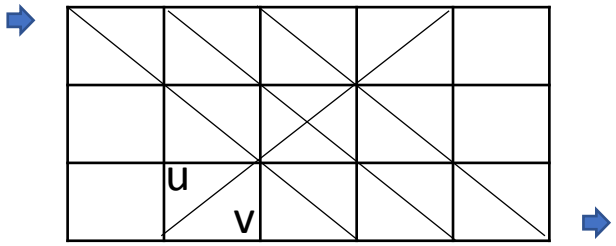
电路维修

- 电路板是一个i行j列的网格，每个交叉点都是电线接入点。每个接入点都有一个电子设备用于把线路接通，同时它是可以旋转的。
- 现在左上角是接入电源，右下角是输出装置。现在电路板处于断路状态，想要知道最少需要旋转多少个接入点才能使整个电路板被接通。



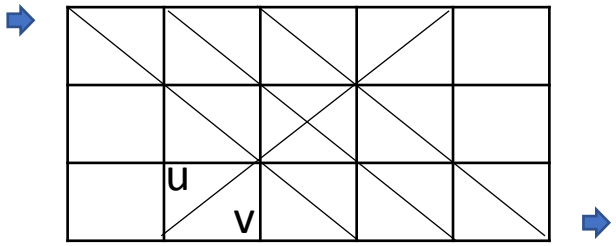
电路维修

- 我们把电路板上的每个点看作是图中的节点，那我们对u点和v点连边，其中u、v点是某个方格的对角
- 如果方格间的对角线与u-v连边重合，意味着u-v边权为0，即不需要花费代价旋转。反之则边权为1



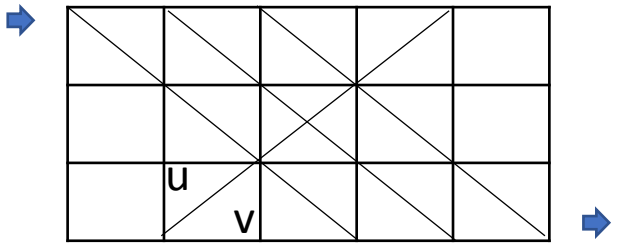
电路维修

- 于是问题转换为在一张边权有0有1的无向图上寻找最短路
- 如果遇到边权为1的边，和普通BFS一样插入队尾
- 遇到边权为0的边，就不能插入队尾了
- 因为这样我们就无法保证在目标节点进入队列后的答案是最优的



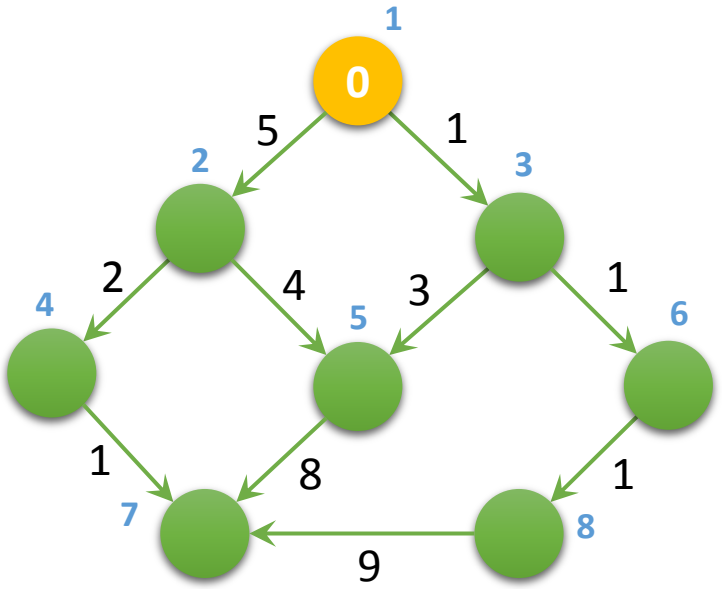
电路维修

- 于是问题转换为在一张边权有0有1的无向图上寻找最短路
- 遇到边权为0的边，就只能从队首插入！



优先队列BFS

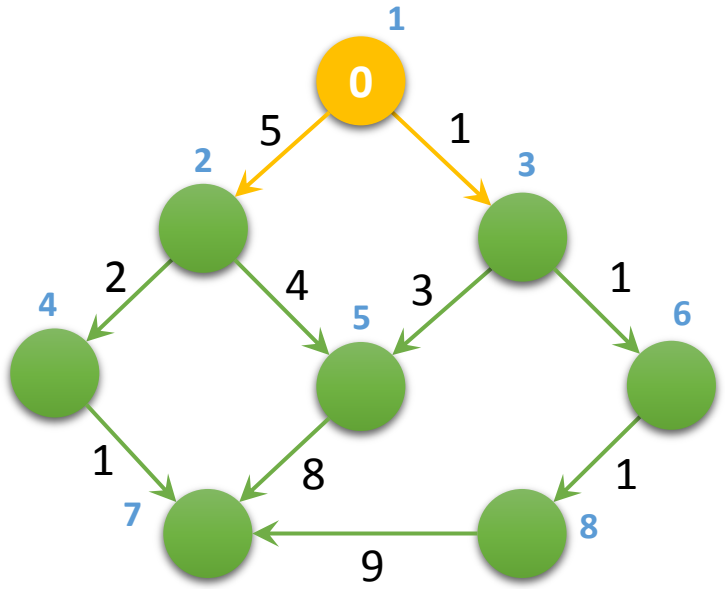
- 我们知道BFS依赖于队列记录访问节点的先后顺序
- 访问节点的更新，是逐个枚举的方式。因此，在搜索最优解的过程中，一个节点有可能多次入队、出队



- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价

优先队列BFS

- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价



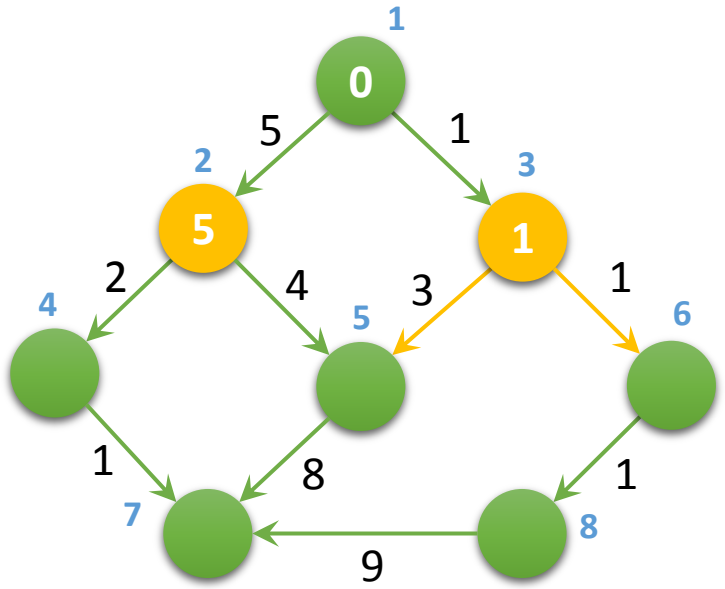
队列：1

队首节点：1

待入队节点：3、2

优先队列BFS

- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价



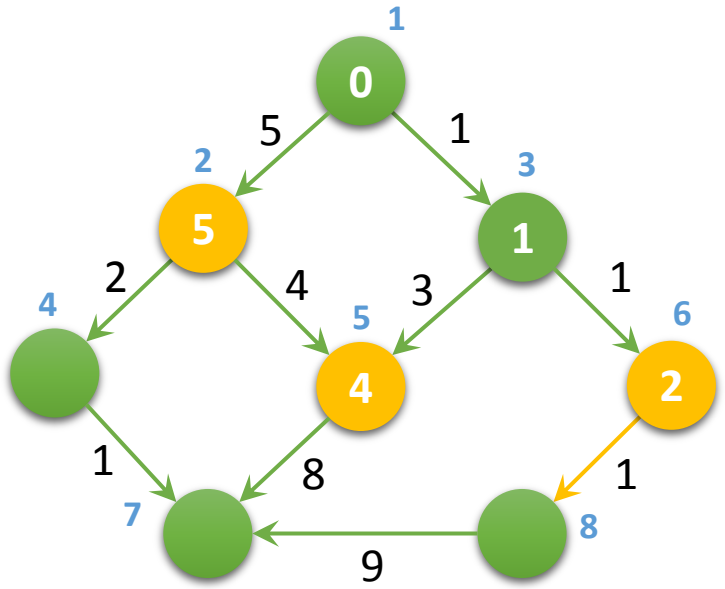
队列：3、2

队首节点：3

待入队节点：6、5

优先队列BFS

- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价



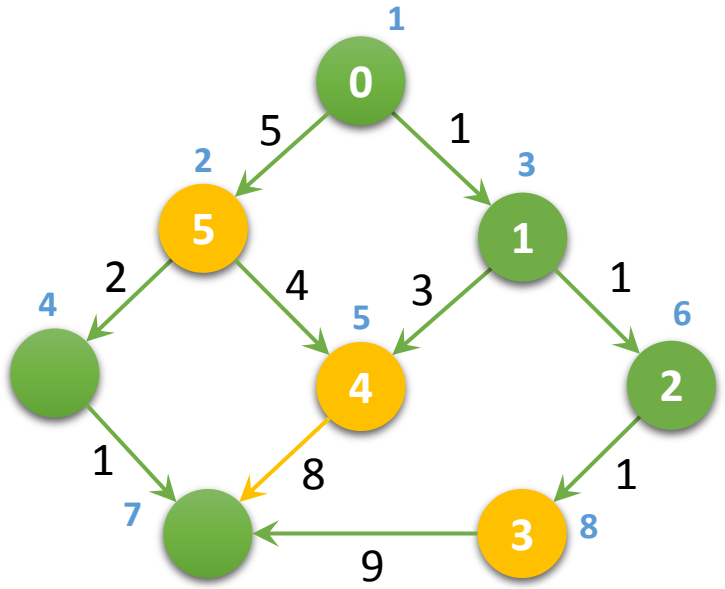
队列：6、5、2

队首节点：6

待入队节点：8

优先队列BFS

- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价



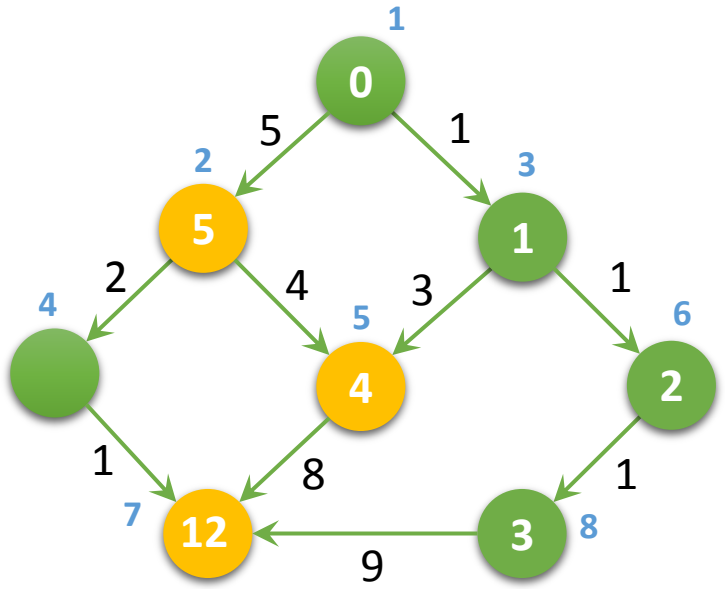
队列：5、2、8

队首：5

待入队节点：7

优先队列BFS

- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价



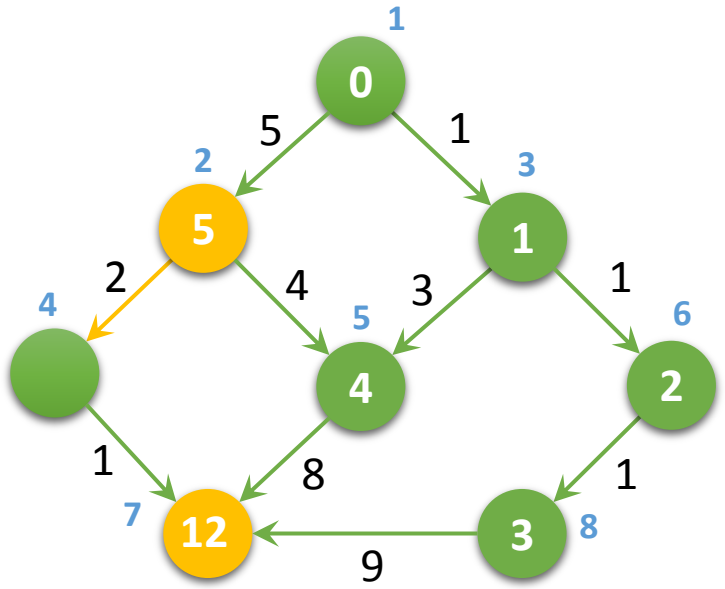
队列：5、2、7

队首：5

待入队节点：7

优先队列BFS

- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价



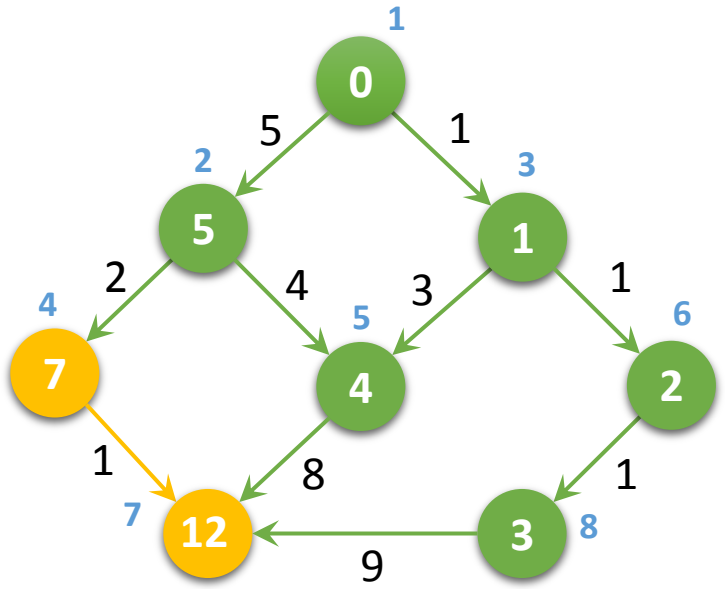
队列：2、7

队首：2

待入队节点：4

优先队列BFS

- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价



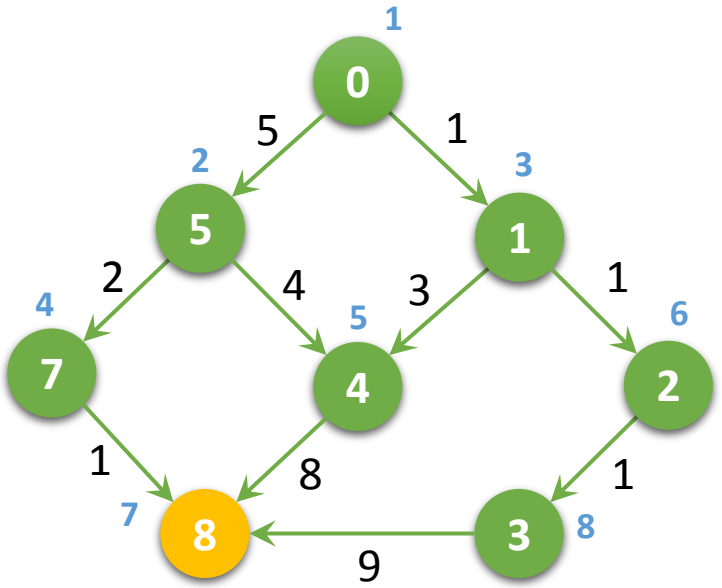
队列：4、7

队首：4

待入队节点：7

优先队列BFS

- 如果改用优先队列来记录先后顺序，会怎样？
- 右图中节点内的数字表示当前累计的代价
- 蓝色数字表示节点编号
- 边上的数字表示遍历该条边的代价



队列：7

队首：7

待入队节点：

最后得到累计代价：8

双向BFS

- 大家应该注意到：如果BFS过程中u可以搜索到v，那么很有可能v也是能搜索到u的，被称之为满足搜索的“可逆性”

- 双向BFS顾名思义，就是从起点状态和终点状态同时BFS，然后借助于判重来判定双向的BFS是否合龙

- 判重数组的值可以多设一组：被逆序访问过。那么一旦某种状态顺序逆序都被访问过，显然就是连接答案的状态

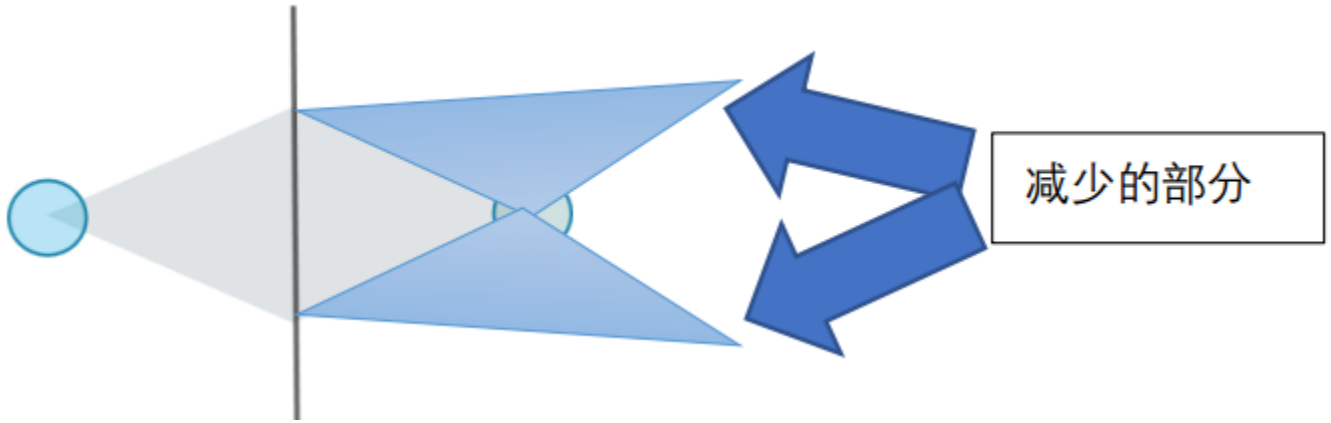
双向BFS

- 要实现双向广搜，就需要开两个队列
- 为了方便我们开二维que[N][0]、que[N][1]
- 也需要两个head[0]/head[1]、tail[0]/tail[1]

- 判重也需要分别判重，而且只和自己队列中的状态判重

双向BFS

- 双向BFS适用的题目：扩展节点较多，目标节点深度较大
- 尤其适合于起始状态和终止状态明确的BFS题型
- 但前提是要求满足搜索可逆性



双向BFS

- 双向BFS的进一步优化：

每次 while 循环时只扩展正反两个方向中节点数目较少的一个，可以使两边的发展速度保持一定的平衡，从而减少总扩展节点的个数，加快搜索速度

选数

- 有n个数，问是否能在这n个数中选出一些数，使得他们的和为指定的k

Sample input	Sample output
4 10 //n、k 2 3 3 3	no
4 9 2 3 3 3	yes

选数

- 首先把这n个数拆成两个部分：n1和n2，然后分别在 $O(2^{n/2})$ 的时间内枚举。
- 设两个部分所得到的和的集合分别为s1和s2，那么我们可以枚举s1的每一个结果s，看k-s是否在s2中出现。
- 如果出现，则说明答案为yes。否则如果对于所有的s，k-s都没有在s2中出现，则说明答案为no
- 时间复杂度$O(n2^{n/2})$

Sample input	Sample output
4 10 //n、k 2 3 3 3	no
4 9 2 3 3 3	yes

方程的解数

- 已知一个n元高次方程：

$k_1x_1^{p1}+k_2x_2^{p2}+\ldots\ldots+k_nx_n^{pn} = 0$

- 其中：$x_1, x_2, \ldots, x_n$是未知数，$k_1, k_2, \ldots, k_n$是系数，$p_1, p_2, \ldots p_n$是指数。且方程中的所有数均为整数
- 假设未知数$1 \le x_i \le m$，求这个方程的整数解的个数

$1 \le n \le 6$；$1 \le m \le 150$

$|k_1M^{p1}|+|k_2M^{p2}|+\ldots\ldots+|k_nM^{pn}| < 2^{31}$

方程的整数解的个数小于2^{31}

方程的解数

- 暴力的复杂度O(nm)

- 我们转化一下那个方程（假设n=6）

$k_1x_1^{p1}+k_2x_2^{p2}+k_3x_3^{p3} = -(k_4x_4^{p4}+k_5x_5^{p5}+k_6x_6^{p6})$

- 只要将左边的取值和右边的取值分别求出来，再判断一下相等的对数即可，时间复杂度$O(m^3)$

广度优先搜索小结

- BFS自带效果是会优先搜索“距离最近”的解
- 而有些题中，起点到终点的距离是固定的，比如全排列问题，这时候DFS更直观而BFS没有优势

- BFS是同一层节点一起进行
- DFS同一时间只进行当前一种方案，所以状态数很多时，BFS需要对所有节点都开检查数组

深度优先搜索 VS 广度优先搜索

深度优先搜索	广度优先搜索
借助于系统栈，实现回溯	借助于手写队列，实现重放
下一个搜的节点，是当前节点的儿子节点	下一个搜的节点，是队首节点的兄弟节点
常用剪枝、迭代加深等优化	常用判重辅助，以双向BFS、A*等优化
适用于求可行解	适用于求最优解

瓷砖

- 在一个m列n行（2≤n,m≤50）的矩形广场上，每块1×1的地面都铺设了红色或者黑色的瓷砖。你站在某块黑色瓷砖上出发，可以移动到上、下、左、右相邻的且是黑色的瓷砖上。问所能经过的黑色瓷砖总数（包括自己）。

Sample input	Sample output
6 5 //m、n # # • • • # //“•”表示黑瓷砖 //”#”表示红瓷砖 • # • # • • # • • # • • • @ • • • • //“@”表示出发点 • • • • • •	22
输入数据没有空格	

瓷砖

1. 读入初始位置，入队作为队首元素
2. BFS，检查队首元素的上、下、左、右四个位置是否黑色瓷砖，是则入队
3. 不断取出队首元素向四个方向扩展，直到队列为空
4. 为了避免一个位置被重复走到，定义一个二维标记数组判重
5. 最后队尾元素即为答案

- 本题是标准的BFS模型应用，解决一类“连通性问题”

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[51][51], q[2][2510], n, m, x, y;
int next[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
void bfs()
{
    q[0][1] = x; q[1][1] = y;
    int head = 1, tail = 1;
    while (head <= tail)
    {
        for (int k = 0; k < 4; k ++)
        {
            x = q[0][head] + next[k][0];
            y = q[1][head] + next[k][1];
            if (a[x][y])
            {
                tail ++;
                q[0][tail] = x;
                q[1][tail] = y;
                a[x][y] = false;
            }
        }
        head ++;
    }
    printf("%d\n", tail);
}
```

```cpp
int main()
{
    scanf("%d%d", &m, &n);
    char c;
    for (int i = 1; i <= n + 1; i ++)
    {
        for (int j = 1; j <= m + 1; j ++)
        {
            scanf("%c", &c);
            if (c == '.') a[i][j] = true;
            else if (c == '@')
            {
                x = i; y = j;  //记录起点坐标
                a[i][j] = false;
            }
        }
        scanf("*");  //配合前面的+1处理换行符
    }
    bfs();
    return 0;
}
```

后记

- 这段参考代码，教了两个额外的代码细节：

1. 我们可以用一个二维数组来模拟（带横、纵坐标）的队列，而不一定要开结构体数组（如果需要记录三个及以上的参数还是要开结构体数组）

```
int que[2][2510];

tail ++;
que[0][tail] = x2;
que[1][tail] = y2;
```

```
struct node
{
    int x, y;
} que[2510];

tail ++;
que[tail].x = x2;
que[tail].y = y2;
```

后记

- 这段参考代码，教了两个额外的代码细节：

2. 在读入字符矩阵时，如何处理换行符的问题

```
for (行 + 1)
{
    for (列 + 1)
    {
        读入一行;
        为需要的数据打上标记;
    }
    读入一个任意字符代替换行符;
}
```

```
# # • • • #
• # • # • •
# • • # • •
• @ • • • •
• • • • • •
```

而且，在后续的BFS中，居然没有常见的“判出界”。为什么？

黑色图像

- 二维图像是由黑和白两种像素组成的n×m（1≤n,m≤100）矩形点阵，图像识别的一个操作是求出图像中最大的黑色区域面积。一个黑色区域中的每个像素至少与该区域中的另一像素相邻，规定仅与其上、下、左、右的像素相邻。而两个黑色区域没有相邻像素。现在求最大黑色区域面积（像素个数）

Sample input	Sample output
5 6 //n、m 0 1 1 0 0 1 //0表示白像素 1 1 0 1 0 1 //1表示黑像素 0 1 0 0 1 0 0 0 0 1 1 1 1 0 1 1 1 0	7

黑色图像

1. 如果先找到一个黑色点，那么问题就和前一题一样了
2. 从左上角开始，利用一个两层循环找到一个黑点，然后BFS
3. 记录下得到的连通块大小
4. 依此类推，维护连通块大小的最大值即可

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[101][101], q[2][10010], n, m, ans, head, tail;
int next[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
void bfs(int i, int j)
{
    q[0][1] = i; q[1][1] = j; a[i][j] = 0;
    head = 1; tail = 1;
    while (head <= tail)
    {
        for (int k = 0; k < 4; k ++)
        {
            int x = q[0][head] + next[k][0];
            int y = q[1][head] + next[k][1];
            if (a[x][y])
            {
                tail ++;
                q[0][tail] = x;
                q[1][tail] = y;
                a[x][y] = false;
             }
        }
        head ++;
    }
}
```

```cpp
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++)
    for (int j = 1; j <= m; j ++)
        scanf("%d", &a[i][j]);
    for (int i = 1; i <= n; i ++)
    for (int j = 1; j <= m; j ++)
        if (a[i][j])
        {
            bfs(i, j);
            ans = max(ans, tail);
        }
    printf ("%d", ans);
    return 0;
}
```

后记

- 这段参考代码，教了一个额外的代码细节：
- 我们的BFS，什么时候需要带参数，什么时候不需要？

1. 一般情况，因为BFS是单线程不存在回头的问题，所以无需带参数，这点和DFS需要大量参数记录中间过程/状态不同
2. 但是像本题这样需要多次从不同出发点开始BFS，势必每次需要把不同的出发点放入一个新队列的队首位置，这时就需要带参数，参数一般就是每次新的出发点坐标

关系

- 编号1~n的n（n≤100）个人，其中一些人相互认识。现在a想要认识b，可以通过他所认识的人来认识更多的人（如果a认识b，b认识c，那么a可以通过b认识c）。现在想要知道x最少需要通过多少人才能认识y（只算中间人）。

Sample input	Sample output
5 1 5 //n、x、y 0 1 0 0 0 //0表示不认识 1 0 1 0 1 //1表示认识 0 1 0 1 0 0 1 1 0 1 0 0 0 1 0	1
1认识2，2认识5	

课外加练

- luogu 1379 八数码难题
- luogu 1451 求细胞数量
- luogu 1126 机器人搬重物
- luogu 1135 奇怪的电梯
- luogu 1162 填涂颜色
- luogu 1332 血色先锋队
- luogu 1443 马的遍历
- luogu 1747 奇怪的游戏
- luogu 3395 路障
- luogu 2895 流星雨
- luogu 1144 最短路计数
- luogu 1032 字串变换
- luogu 1330 封锁阳光大学
- luogu 1747 好奇怪的游戏
- luogu 1476 离开中山路