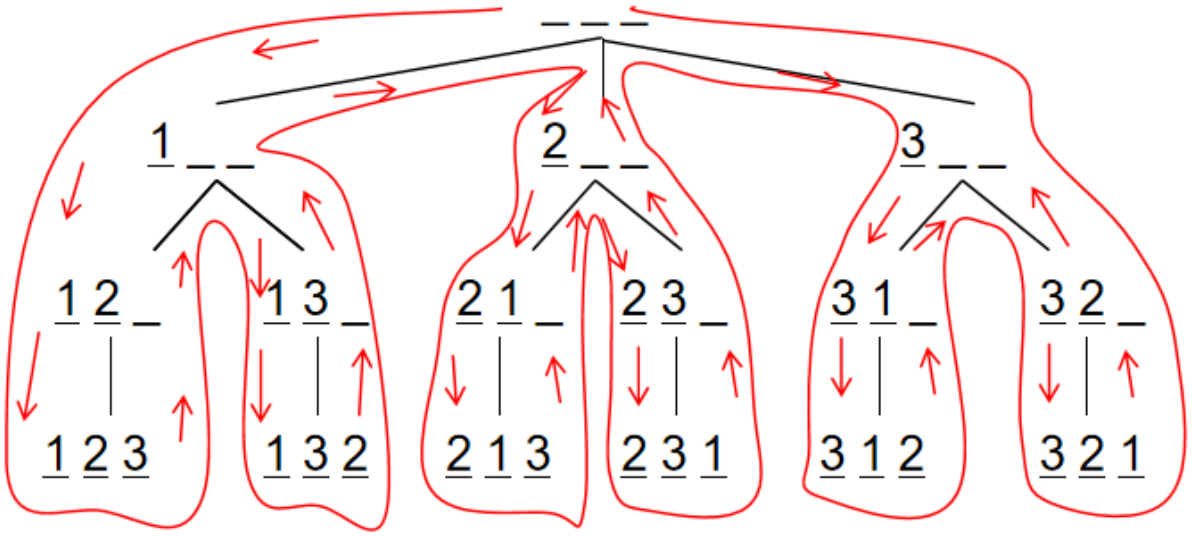
算法基础（九）：DFS与BFS

DFS

全排列问题：

基本思想：

dfs 最重要的是搜索顺序。用什么顺序遍历所有方案。对于全排列问题，以 n = 3 为例，可以这样进行搜索



从根结点一直搜索到最深处，然后返回

代码实现：

给定一个整数 n，将数字 $1 \sim n$ 排成一排，将会有很多种排列方法。

现在，请你按照字典序将所有的排列方法输出。

输入格式

共一行，包含一个整数 n。

输出格式

按字典序输出所有排列方案，每个方案占一行。

数据范围

$1 \leq n \leq 7$

输入样例：

```
3
```

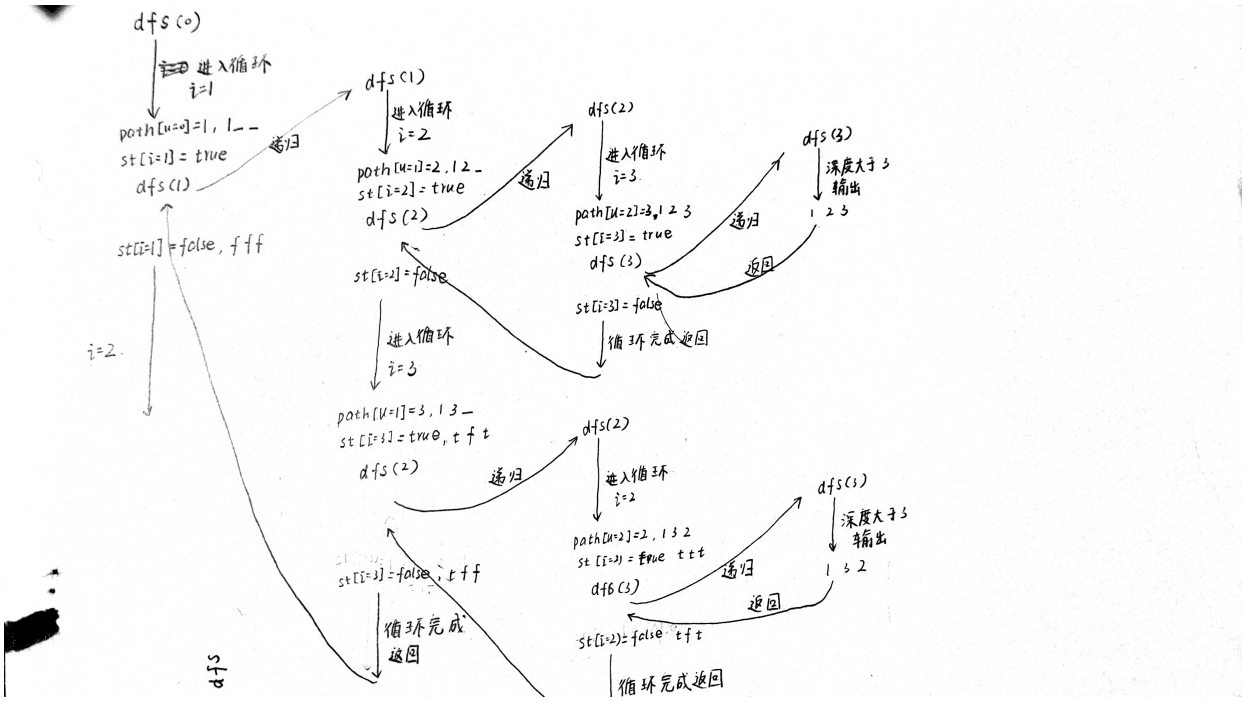
输出样例：

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

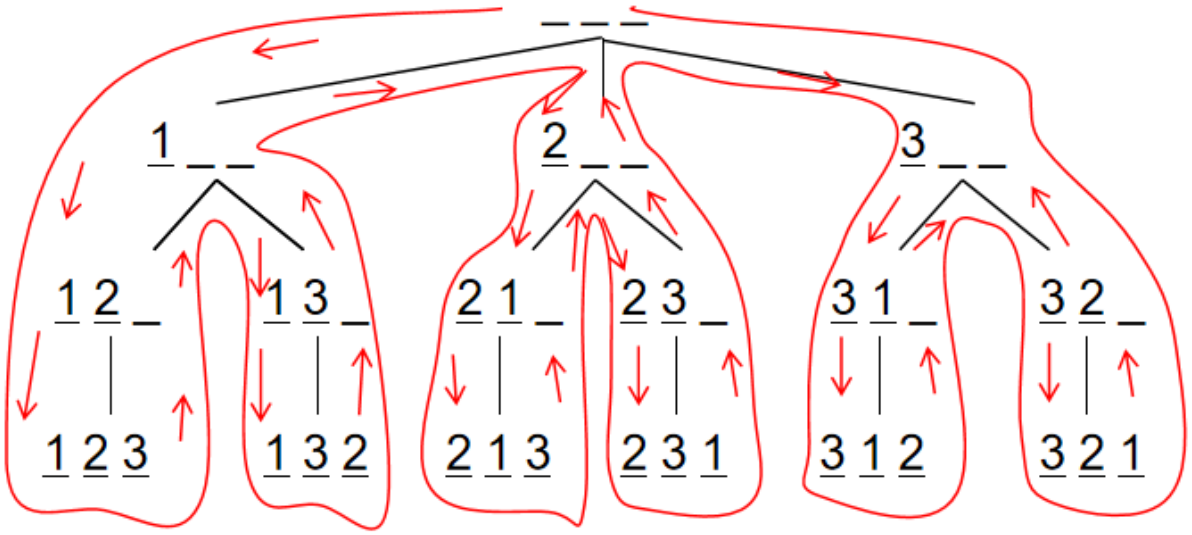
```cpp
#include<iostream>
using namespace std;
const int N = 10;

int n;
int path[N];//path数组用来记录搜索路径上的数值
bool st[N];//st[]数组用来记录那个数被用过了，st[i] = true表示这个数在搜索路径上被用过
//同时在回溯的时候用来恢复数字的状态

void dfs(int u)
{
    //第1次搜索第0层，将数字放到path[0]的位置
    //第n次搜索第n - 1层，将数字放到path[n-1]的位置
    //第n+1次搜索到第n层，这时候就没有数了，需要返回
    if(u == n)
    {
        //因为搜索是从第0层开始的，第0层搜索的结果当然要放在path[0]中
        //当搜索到第n层的时候深度足够需要返回，此时返回的是path[0...n-1]的值
        for(int i = 0; i < n; i ++) printf("%d ", path[i]);
        printf("\n");
        return ;
    }

    //每次搜索，依次看每一个数，看是否被用过
    //这里的i从1开始是因为我们给的全排列的数是1...n，若给的数是2...n+1，那必然是从2开始
    //i从1开始与搜索的层数从0开始没有必然联系
    //st[]的下标也只与给的全排列的数有关，跟搜索的层数从0开始没有必然联系
```

```
    for(int i = 1; i <= n; i++)
    {
        if(!st[i])//如果这个数没有被用过就将其加入到搜索路径中
        {
            path[u] = i;
            st[i] = true;
            //继续向下一层搜索
            dfs(u + 1);
            //下一层搜索完返回后将这一层搜索的数值设置为false，在后面的搜中也可以用到
            st[i] = false;
        }
    }

}



int main()
{
    cin >> n;

    //树的搜索从第0层开始
    dfs(0);

    return 0;
}
```

递归过程分析如下：



再看看下面这张图：



以第一个分支为例，当搜索到`dfs(3)`的时候再递归深度足够，输出结果，函数返回到`dfs(2)`，然后将`st[3]`设置为`false`，在第二张图中就是回到状态`1 2 _`

接着循环完成返回，也就是第二张图中状态`1 2 _`没有第二个分支了，返回到`dfs(1)`，并设置`st[2] = false`，在第二张图中就表示回到状态`1 _ _`

然后进入`i = 3`的循环，在第二张图中表示进入到状态`1 3 _`

然后递归进入`dfs(2)`，`i = 2`表示第二张图进入状态`1 3 2`，接着再次递归深度足够返回到`dfs(2)`，并设置`st[2] = false`表示第二张图进入`1 3 _`

然后循环完成返回，表示状态`1 3 _`不再有分支了，返回到`dfs(1)`并设置`st[3] = false`，表示回到状态`1 _ _`

这时`dfs(1)`的循环完成，表示`1 _ _`不再有分支，返回到状态`_ _ _`，然后`dfs(0)`，`i = 2`，进入第二个分支

可以看到这里存在着两种返回方式：

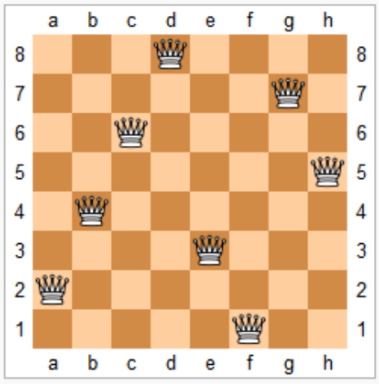
1. 深度足够的返回，上图中，达到`dfs(3)`之后，输出结果，返回到`dfs(2)`，并设置`st[3] = false`表示返回到`1 2 _`
2. 分支尝试完全的返回，上图中，状态`1 2 _`的分支尝试完了之后，也就是`for`循环尝试完了之后，返回到`dfs(1)`，并设置`st[2] = false`，表示返回到状态`1 _ _`

所以，代码中的函数返回并不是回溯到了一个状态，必须同时将状态设置好了才表示回溯到了对应的状态

并且可以看到，第一种深度完全的返回可以看做是第二种返回的特殊方式，进入`dfs(3)`了之后表示直接不存在分支循环完了，然后返回`dfs(2)`，并设置`st[3] = false`

N皇后问题：

$n-$皇后问题是指将 n 个皇后放在 $n \times n$ 的国际象棋棋盘上，使得皇后不能相互攻击到，即任意两个皇后都不能处于同一行、同一列或同一斜线上。



One solution to the eight queens puzzle

现在给定整数 n，请你输出所有的满足条件的棋子摆法。

输入格式

共一行，包含整数 n。

输出格式

每个解决方案占 n 行，每行输出一个长度为 n 的字符串，用来表示完整的棋盘状态。

其中 `.` 表示某一个位置的方格状态为空，`Q` 表示某一个位置的方格上摆着皇后。

每个方案输出完成后，输出一个空行。

注意：行末不能有多余空格。

输出方案的顺序任意，只要不重复且没有遗漏即可。

数据范围

$1 \leq n \leq 9$

输入样例：

```
4
```

输出样例：

```
.Q..
...Q
Q...
..Q.

..Q.
Q...
...Q
.Q..
```

第一种解法：

其实`n`皇后问题可以看作第一行该放哪一列，第二行该放哪一列，第三行该放哪一列。。。。。这样的问题

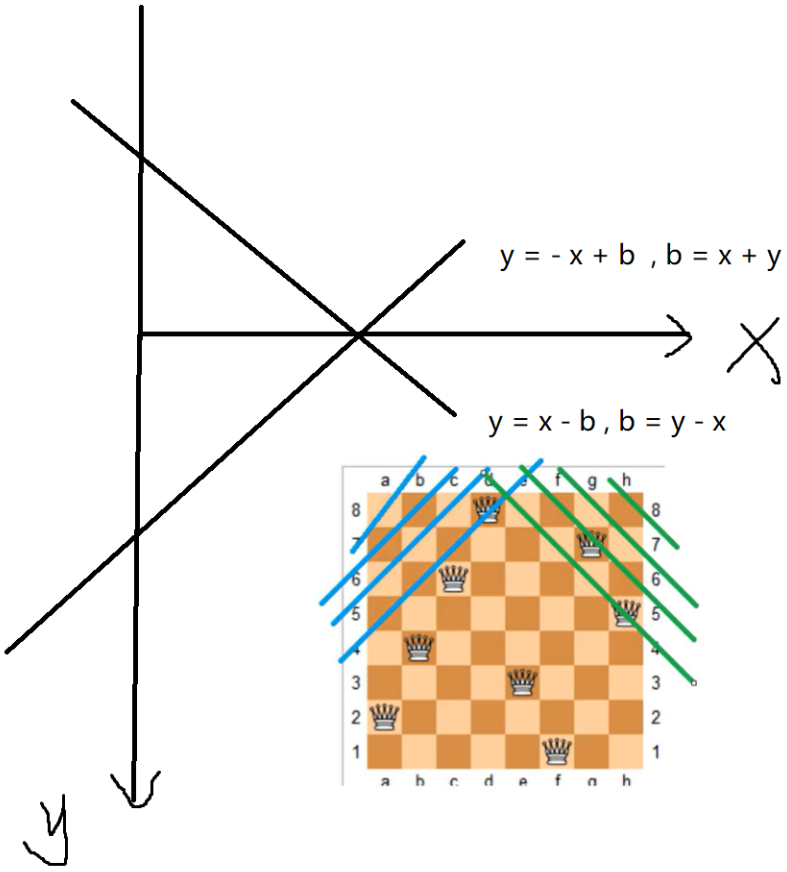
所以他其实可以看作是对应列的全排列，所以代码的结构与全排列很相似

但是跟全排列不同的是，在某一行遍历所有的列的时候会通过`if`语句直接跳过了不行的分支，假如在第`i`行如果第`j`列不行的话，会直接跳过对应代码`if(!col[i] && !dg[u + i] && !udg[u - i + n - 1])`，进行下一列的尝试，从而达到了剪支的目的

代码如下：

```
#include<iostream>
using namespace std;

const int N = 20;
int n;
char g[N][N];//用来记录结果
//col[i] = true表示第i列上存在一个皇后
//dg[i] = true表示第i个正对角线上存在一个皇后
//udg[i] = ture表示第i个反对角线上存在一个皇后
bool col[N], dg[N], udg[N];

//dfs用来拓展行，与全排列里面拓展数的长度一样
void dfs(int u)
{
    if(u == n)
    {
        for(int i = 0; i < n; i ++)
        {
            puts(g[i]);//输出这一行的字符串
        }
        puts("");//输出换行
        return ;
    }

    //for循环用来拓展列，对应于全排列中拓展数的分支
    //i从0开始表示从第0列开始
    //跟全排列中从1开始不一样
    for(int i = 0; i < n; i ++)
    {
        //如果对应的列和对角线上没有元素
        if(!col[i] && !dg[u + i] && !udg[n - 1 + u - i])
        {
            col[i] = dg[u + i] = udg[n - 1 + u - i] = true;
            g[u][i] = 'Q';
            dfs(u + 1);
            col[i] = dg[u + i] = udg[n - 1 + u - i] = false;
            g[u][i] = '.';
        }
    }
}

int main()
{
    //int n;
    cin >> n;

```

```
47      for(int i = 0; i < n; i++)
48      {
49          for(int j = 0; j < n; j ++)
50          {
51              g[i][j] = '.';
52          }
53      }
54
55      dfs(0);
56      return 0;
57
58  }
```



主对角线 `dg[i]` 的下标 `i` 是 `y = -x + b` 的 `b`，从中可以看到第 `u` 行，`i` 列的元素所在的对角线的是第 `u + i` 条，可以将其作为下标

蓝色的主对角线的下标按序增加即可，不需要特殊处理

绿色的反对角线 `udg[i]` 的下标是 `y = x - b` 的 `b`，`b = y - x`

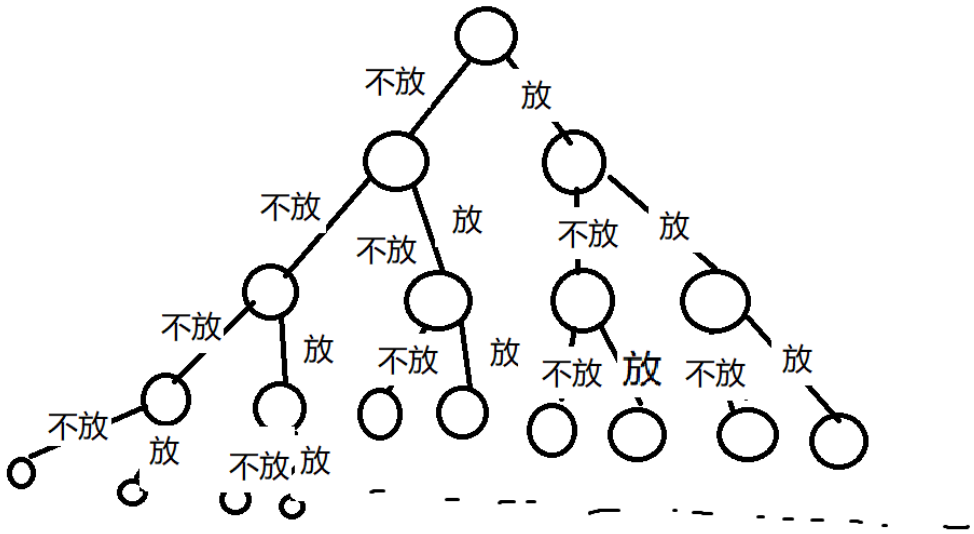
绿色的反对角线，在第 0 条的时候 `y - x` 的结果是 `-(n-1)` 加上 `n-1` 就是 `0`，一直到正中间的对角线，此时是第 `n - 1` 条反对角线，对角线再增加的话就是一个正数加上 `n - 1`

所以从中可以看到第 `u` 行，`i` 列的元素所在的反对角线的条数是第 `u - i + ( n - 1)`，可以将其作为下标

注意我们不需要一定将第几条对角线作为 `dg` 的下标，我们只需要满足对应关系即可，即这条对角线上的元素经过一个计算都可以对应到这条对角线上，然后我们可以判断就行，比如在反对角线中，我们也可以将第 `i` 条对角线的下标设置为 `u - i + n`，后面这条对角线的元素经过 `u - i + n` 的计算都会映射到这条对角线上，对结果没有影响

第二种解法：

第二种解法以每一个方格作为对象，严格来说第一种解法是第二种的一个简化版，第一种解法直接对行进行递归，默认了一行中不能存放一个皇后，第二种解法以方格作为递归对象，在方格放上皇后的时候都需要检查是否满足行，列，对角线，反对角线是否满足，以 `2 * 2` 的方格为例，其搜索树如下：



需要注意的是，这种做法的时间复杂度非常高，是最原始的暴力解法

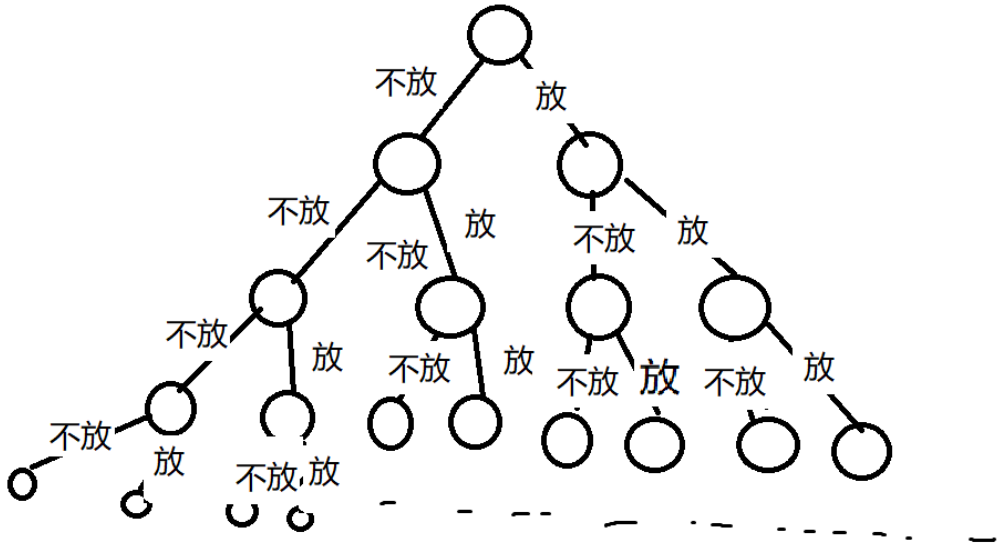
代码如下：

```
#include<iostream>
using namespace std;
const int N = 20;
int n;
char g[N][N];

bool col[N], row[N], dg[N], udg[N];


void dfs(int x, int y, int s)
{
    //方格的下标是从0开始的，当第一行的方格搜索完了之后相当于越界了一次
    //此时将越界了的方格下标调整为正确的下标
    if(y == n) y = 0, x ++;
```

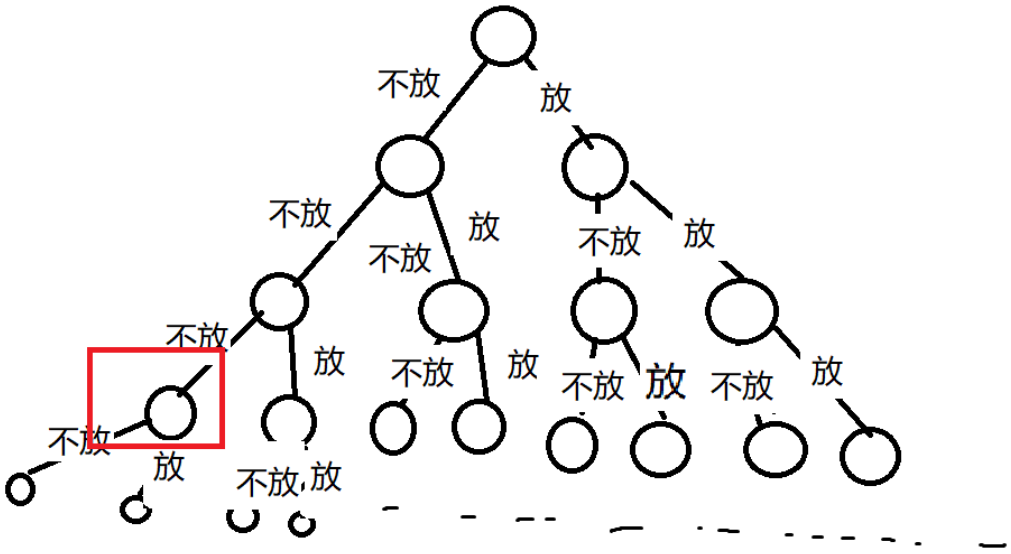
```
15
16      if(x == n)//方格的下标是从0开始的，x = 0相当于行越界，行越界那必然每个方格都递归完了，此时需要返回
17      {
18          //如果已经放上了n个皇后，表示此时递归完所有的方格是一种情况，需要输出
19          //需要注意的是s的值不会大于n
20          //由于if的剪枝条件，每行最多一个，所以递归完所有的格子时，s不可能大于n
21          //对于边界情况，当最后一个格子递归完，x越界时，由于每行最多一个，并且放一个就加一，所以满足的情况下，必然s = n
22          if( s == n)
23          {
24              for(int i = 0; i < n; i ++)
25              {
26                  puts(g[i]);
27              }
28              puts("");
29          }
30
31          return ;//递归到叶结点，返回
32
33      }
34      //接下来进行分支递归，相当于全排列中的while循环
35      dfs(x, y + 1, s);//这个格子不放，递归到下一个格子
36
37      //递归到下一个格子，放的情况，if语句进行剪枝
38      if(!row[x] && !col[y] && !dg[x + y] && !udg[n - 1 - y + x])
39      {
40          row[x] = col[y] = dg[x + y] = udg[n - 1 - y + x] = true;
41          g[x][y] = 'Q';
42          dfs(x, y + 1, s + 1);//递归到下一个格子
43          row[x] = col[y] = dg[x + y] = udg[n - 1 - y + x] = false;
44          g[x][y] = '.';
45      }
46  }
47
48
49  int main()
50  {
51      cin >> n;
52
53      for(int i = 0; i < n ; i++)
54          for(int j = 0; j < n; j ++)
55              g[i][j] = '.';
56
57      dfs(0 ,0 ,0);
58
59
60      return 0;
61  }
```

我们只简单分析其回溯的情况，其他的递归过程类似于全排列的分析

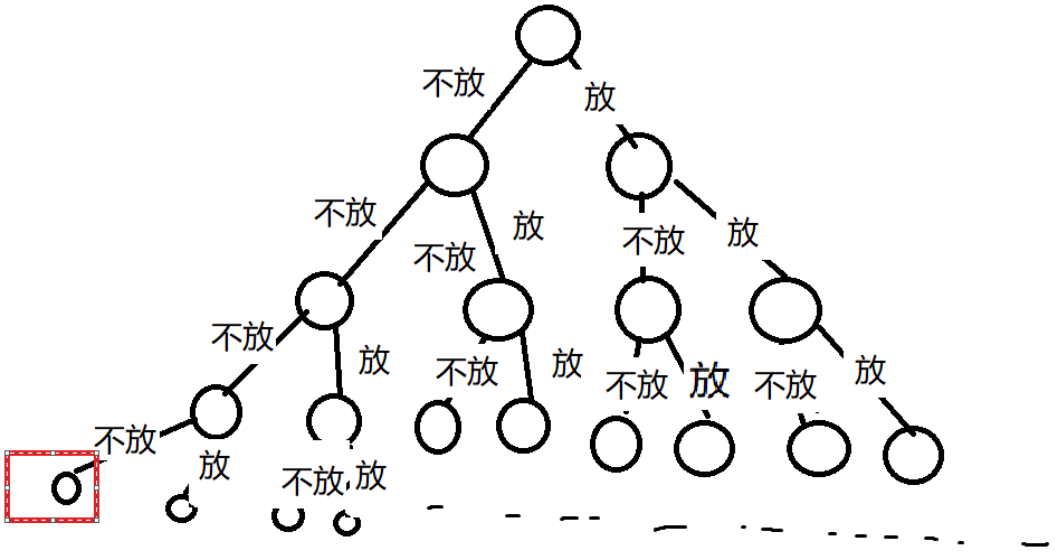
以下面这个图为例：



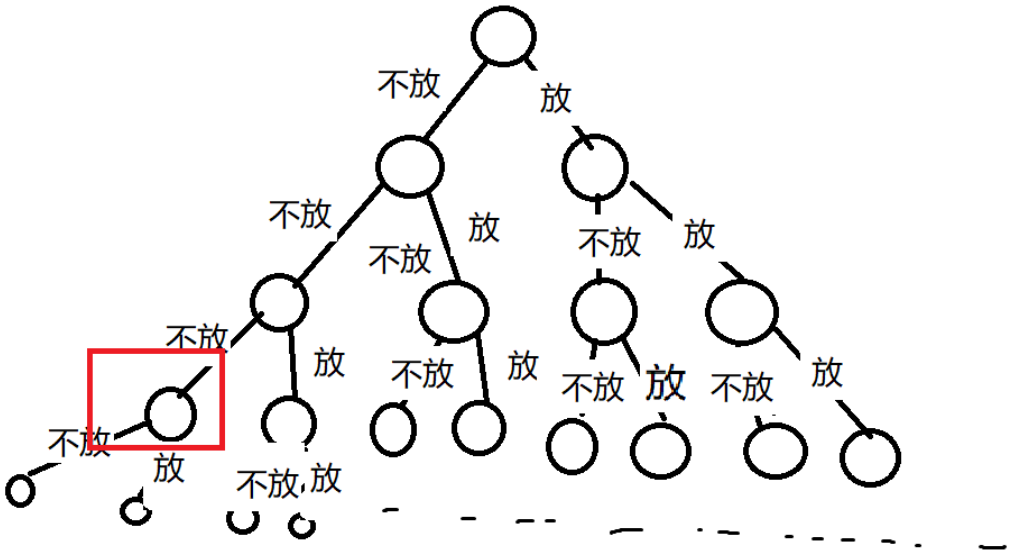
从`dfs(0, 0, 0)`开始，一路递归到`dfs(1, 1, 0)`也就是下图这个点，代表下标为`(1, 1)`的这个格子：



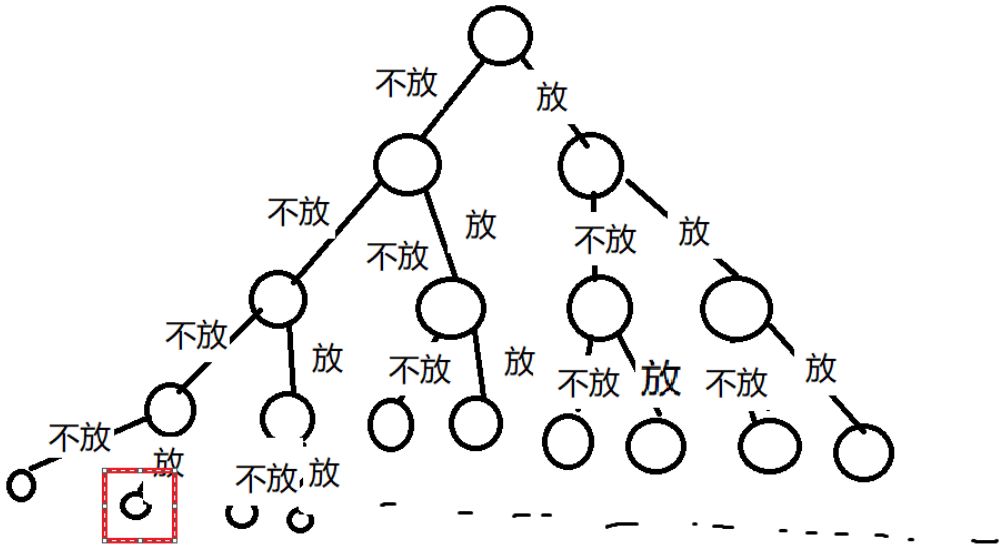
然后这个格子不放，继续递归进入函数`dfs(1, 2, 0)`，也就是下面这个结点



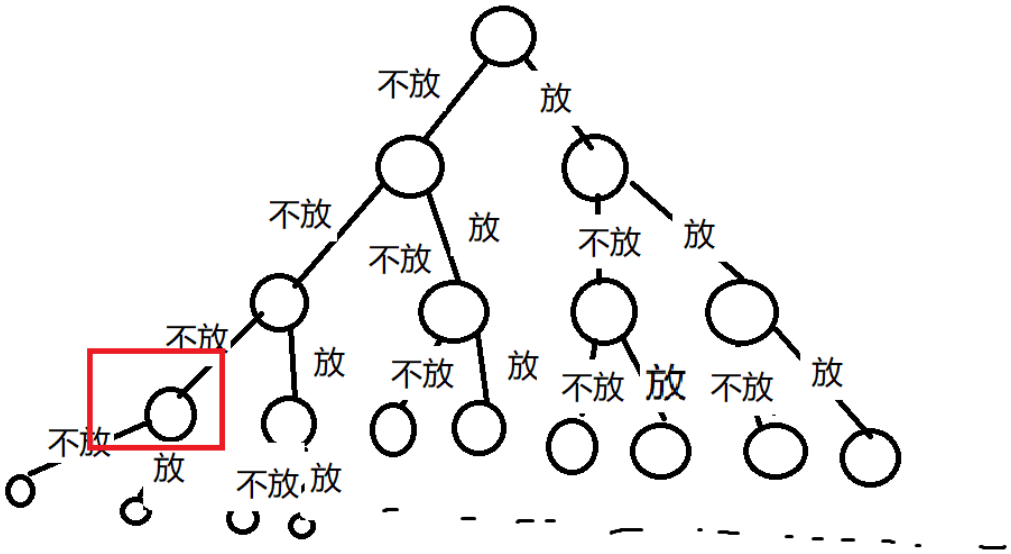
然后调整坐标发现此时 `x` 越界，返回到函数 `dfs(1, 1, 0)`，也就返回到下面这个结点



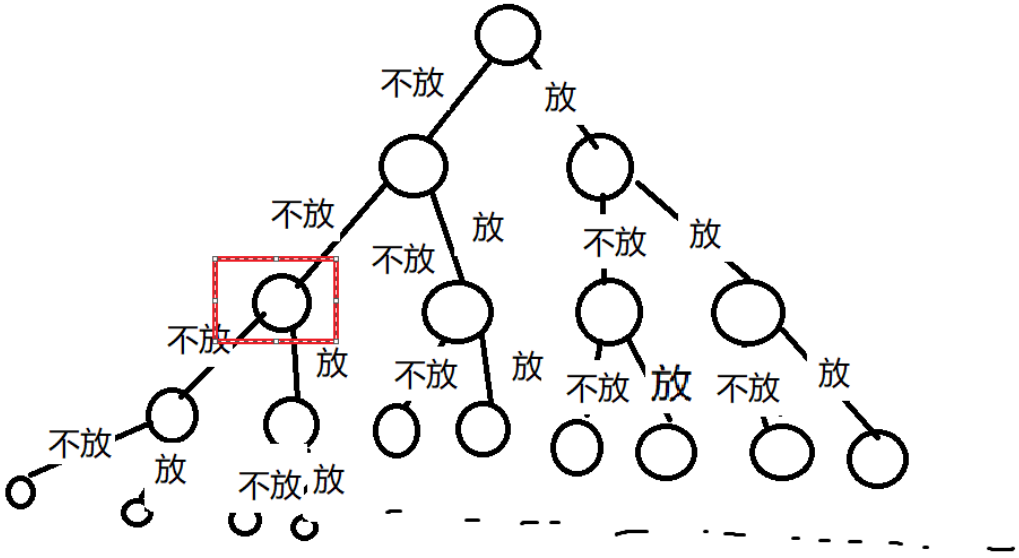
接着这个结点对应的格子放上一个皇后，调整对应的行，列，对角线，反对角线的状态，递归进入函数 `dfs(1, 2, 1)` 进入下面这个结点：



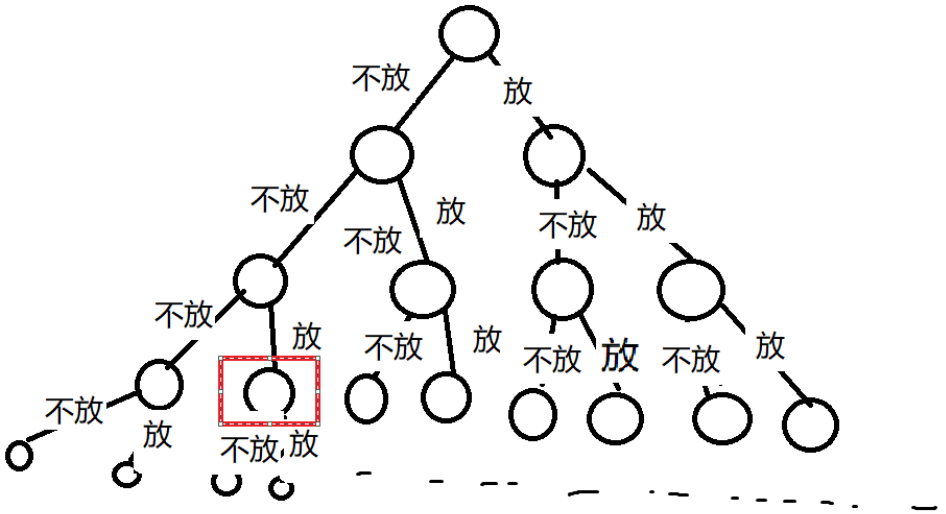
然后调整坐标发现此时 `x` 越界，返回到函数 `dfs(1, 1, 0)` 并调整对应的行，列，对角线，反对角线的状态为原来的状态，表示回溯到原来这个结点：



这个结点的所有分支递归完成，类似于全排列中的 `for` 循环结束，回溯到函数 `dfs(1, 0, 0)` 回溯到上一个结点：



这个结点对应的格子，不放的情况已经递归完了，然后继续递归放的情况，调整对应的行，列，对角线，反对角线的状态，进入函数`dfs(1, 1, 1)`，也就是进入下面这个结点：



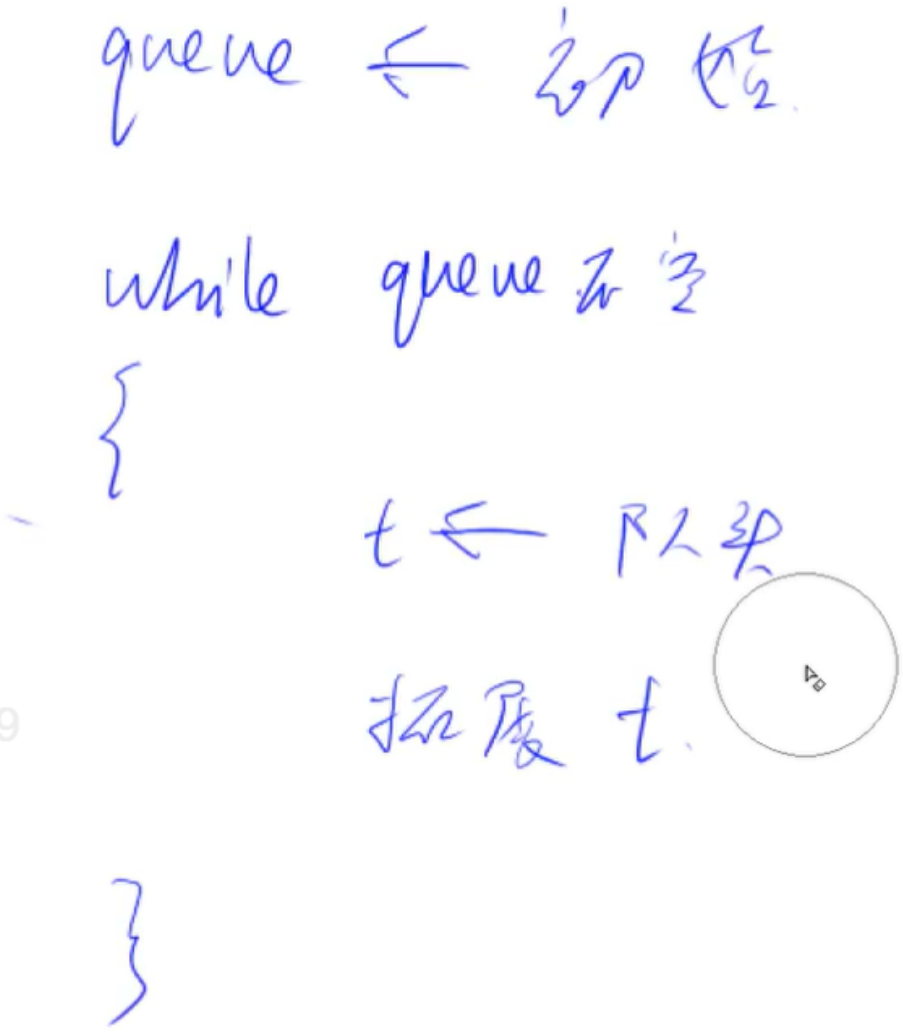
然后重复上述的递归，回溯过程，不再赘述

BFS

广度优先搜索（也称宽度优先搜索，缩写BFS，以下采用广度来描述）是连通图的一种遍历策略。因为它的思想是从一个顶点`v0`开始，辐射状地优先遍历其周围较广的区域，因此得名。

一般可以用它做什么呢？一个最直观经典的例子就是走迷宫，我们从起点开始，找出到终点的最短路程，很多最短路径算法就是基于广度优先的思想成立的。

BFS一般的模板如下：



统一用一个队列来实现宽度优先搜索

走迷宫问题:

给定一个 $n \times m$ 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

$1 \leq n, m \leq 100$

输入样例：

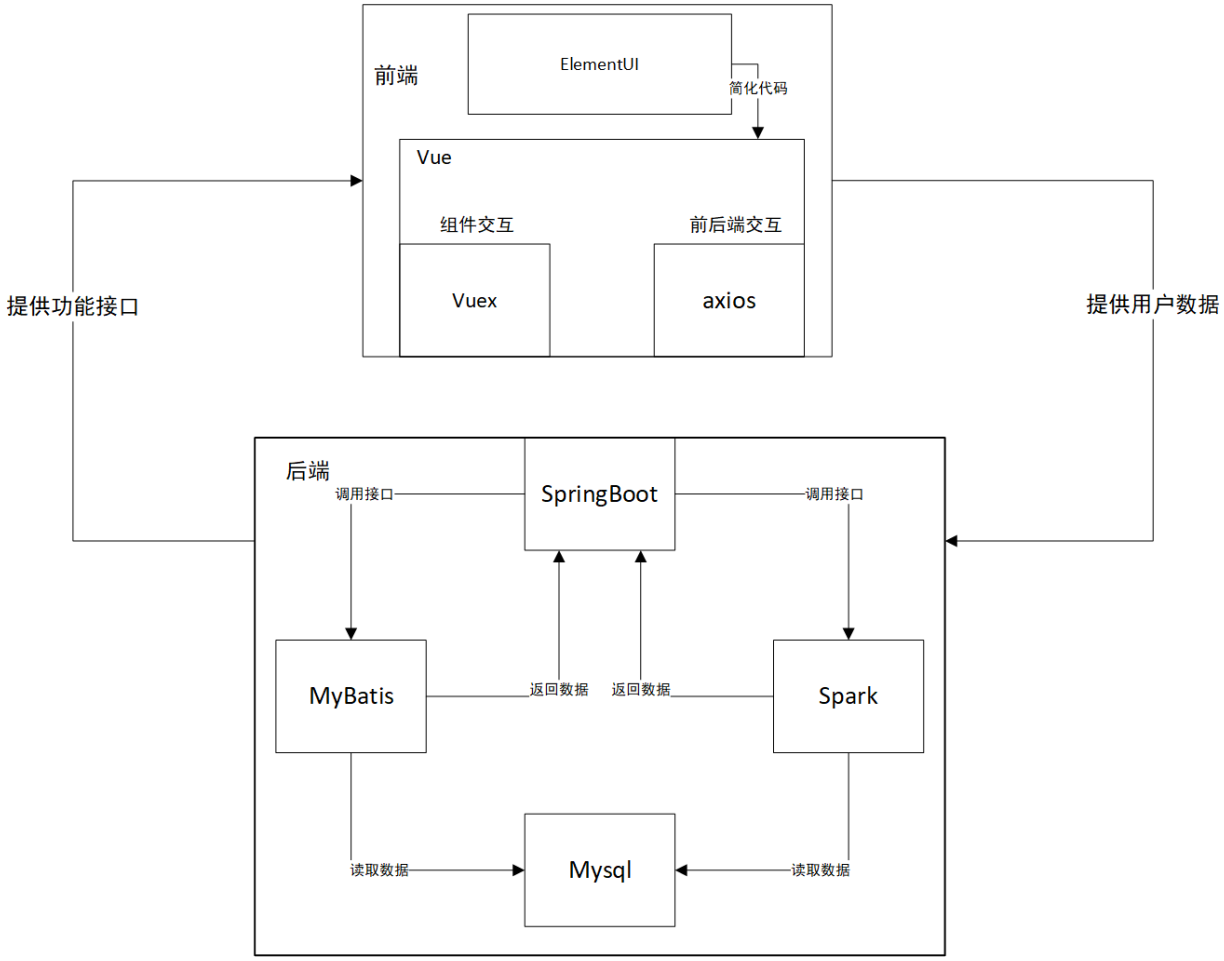
```
5 5
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

输出样例：

```
8
```

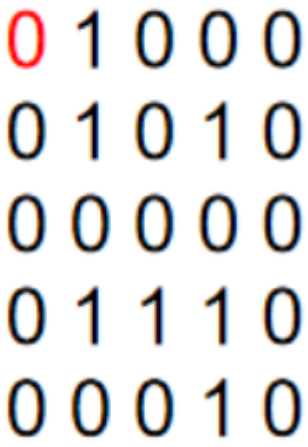
基本思想：

BFS，每一步都保证搜索的点是距离当前点最近的点

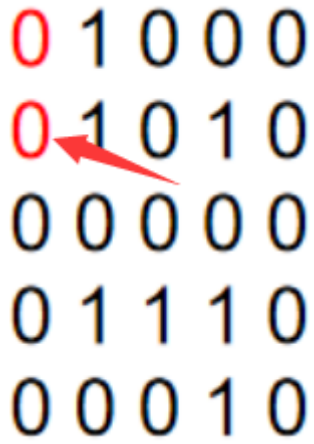


我们用一个队列来实现

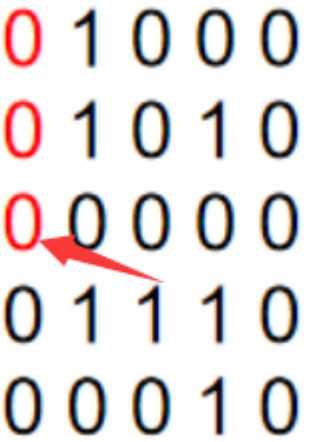
第一次是第`0`个点，跟自己的距离是`0`，我们把它加入队列：



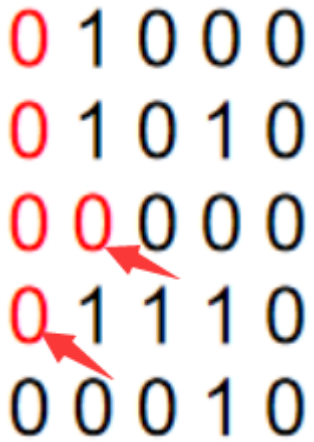
第二次，我们把这个点从队列中取出来，然后看他的前后左右合适的点，计算距离，然后压入队列，如箭头所指的点压入队列：



第三次我们将上次压入队列的点取出来，然后看他的前后左右合适的点，计算距离，然后压入队列，如箭头所指的点压入队列：



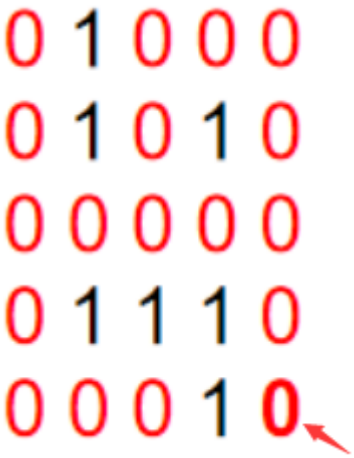
第四次我们将上次压入队列的点取出来，然后看他的前后左右合适的点，计算距离，然后压入队列，如箭头所指的点压入队列：



第五次我们将箭头所指的其中一个点取出队列然后找合适的点，计算距离然后压入队列，由于我们是压入队尾，所以下一次再取出一个点找合适点计算距离的时候我们取的是上图中箭头所指的第二个点，所以我们利用队列实现了搜索的时候始终是从距离小的开始搜索，直到将距离小的一层搜索完再去搜索下一层，队列中的各点按照到初始点的距离从小到大排序的顺序没有发生改变，在下一次取出队头元素然后计算合适点的距离再压入队列仍然是广度优先搜索的。

我们利用队列就实现了广度优先搜索。

一直到最后一次：



我们将这个点放入队列，然后将前面的所有的点遍历完取出后将这个点取出，然后找不到点了，队列为空，跳出循环。

接着输出这个点的距离，就是答案了。

代码实现：

```
#include<iostream>
#include<algorithm>
#include<queue>
#include<cstring>
using namespace std;

typedef pair<int, int> PII;
const int N = 110;
int n, m;
//g数组存放的是地图
int g[N][N];
//d数组存放的是每一个点到起点的距离
int d[N][N];
PII q[N * N];//定义一个队列,由于队列中需要存放地图中的所有元素所以大小必须是N*N
//队列中的每一个元素是一个pair对，表示点的下标


int bfs()
{
    //定义队头和队尾指针，一开始队列不为空，已经放入了第一个点
    int hh = 0, tt = 0;

    //队列中的第一个元素，起始点的下标
    q[0] = {0, 0};

    //定义两个方向向量，用来查找四个方向
    int dx[4] = {-1, 0, 1, 0};
    int dy[4] = {0, 1, 0, -1};

```

```
    memset(d, -1, sizeof d);//将所有点的距离初始化为-1表示这个点没有走过

    d[0][0] = 0;//表示第一个点已经走过了

    //只要队列不空，指队头小于等于队尾，队头等于队尾的时候队列中有一个元素
    while(hh <= tt)
    {
        //每次取出一个队头，并把这个队头弹出
        auto t = q[hh ++];

        //遍历四个方向
        for(int i = 0; i < 4; i ++)
        {
            //i=0的时候点横坐标-1， 纵坐标+0，表示往左走
            int x = t.first + dx[i], y = t.second + dy[i];
            //如果遍历的这个点在地图的范围内，并且这个点可以走，并且这个点还没有走过
            //只有当第一次搜的的时候才是最短距离，才会加一
            //如果是前一次已经搜索过的点，本次有一次搜索到它，前一次的距离必然会比这一次的小，所以这一次必然不能将其加入到队列中
            if(x >= 0 && x < n && y >= 0 && y < m && g[x][y] == 0 && d[x][y] == -1)
            {
                //计算这个点的距离
                d[x][y] = d[t.first][t.second] + 1;
                //将这个点加进队尾
                q[++ tt] = {x, y};
            }
        }
    }

    return d[n - 1][m - 1];//输出右下角的点的距离
}


int main()
{
    cin >> n >> m;
    //读入整个地图
    for(int i = 0; i < n; i ++)
        for(int j = 0; j < m; j ++)
            cin >> g[i][j];

    cout << bfs() << endl;

    return 0;
}
```

另外，如果想要记录路径的话可以开一个数组，记录每个点的前一个点是多少。

声明数组：

```
PII prev[N][N];//prev[x][y] = {x1, y1}表示x,y这个点是从x1，y1这个点搜索过来的
```

修改下面这块代码：

```
if(x >= 0 && x < n && y >= 0 && y < m && g[x][y] == 0 && d[x][y] == -1)
{
    //计算这个点的距离
    d[x][y] = d[t.first][t.second] + 1;
    //将这个点加进队尾
    q[++ tt] = {x, y};
    prev[x][y] = {t.first, t.second};//记录这个点的前一个点
}
```

然后在函数的结尾输出路径：

```
int x = n - 1;
int y = m - 1;
while(x || y)//当x = 0并且y = 0的时候结束循环，也就是从后往前输出到初始点
{
    cout << x << " " << y << endl;
    auto t = prev[x][y];
    x = t.first;
    y = t.second;
    //注意此处不能写下面的代码：
    //x = prev[x][y].first;
    //y = prev[x][y].second;
    //上面错误代码中给y赋值的时候x发生了变化不再是原来的x了
}
```