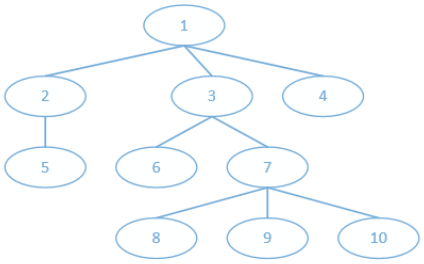
广度优先搜索 1

与深度优先搜索不同，广度优先搜索对于在遍历的树，总是按层次来遍历的，如下图给出的搜索树，它遍历的顺序为 1、2、3、4、5、6、7、8、9、10。



那么，广度优先搜索是如何实现按层次遍历结点的呢？这里要借助一种数据结构——队列。

1、 我们将结点 1 入队。

1									

2、 从队头取出结点，将队头结点的转移状态入队。结点 1 可以转移到 2、3、4 的状态，所以，2、3、4 入队。

	2	3	4						

3、 重复步骤 2，从队头取出结点 2，将 2 的转移状态 5 入队。

		3	4	5					

4、 继续从队头取出结点 3，将结点 3 的拓展状态 6、7 入队。

			4	5	6	7			

5、 重复以上步骤，直到队列为空为止。

通过以上步骤模拟，我们发现，利用队列来存储结点的拓展状态，队头结点的出队顺序就是以层次遍历的顺序出队的。

一般而言，广度优先搜索的基本模板如下：

```
初始状态入队
while(队列不为空){
        取出队头结点;
        for(枚举队头结点的拓展状态){
                if(该拓展状态可用){
                        if(该拓展状态是目标状态){
                                输出答案;
                                return;
                        }
                        锁定该拓展转态;
                        拓展转态入队;
                }
        }
```

oiclass

```
14. }
```

要深刻理会这个模板，我们需要结合以下例题讲解。

家族

【题目描述】
在一个与世隔绝的岛屿上，有一个有趣的现象：同一个家族的人家总是相邻的（这里的相邻是指东南西北四个方向），不同的家族之间总会有河流或是山丘隔绝，但同一个家族的人不一定有相同姓氏。现在给你岛上的地图有 n 行，每行有若干列，每个格子中要么是“”，表示大海，要么是“*”，表示河流或山丘，要么是小写字母，表示一户人家的姓氏。
【输入】
第一行是个数字 N，表示下面信息的行数。接下来是 N 行字符，每行由小写字母和*号组成，有些行的最前面也可能包含若干连续的空格，表示这些区域是大海，每一行最多不超过 200 个字符。
【输出】
一个数字，表示家族数。
【样例输入】
4
*zlw**pxh
l*zlwk*hx*
w*tyy**yyy
zzl
【样例输出】
3
【数据范围】
10%的数据，n≤1。
30%的数据，n≤10。
100%的数据，n≤100 每一行最多不超过 200 个字符。
【分析】
本题其实就是求连通块的数量，题型与前面深度优先搜索里的《细胞数字》类似，不同的地方在于本题需要以字符的形式读入，细节处理的地方较多。
对于求连通块数量，可以用深搜来解，也可以用广搜来解。不过，因为深搜是使用递归栈来实现程序的，极端情况下，连通块的数量较多，比如达到 500*500，那么递归的层数达到 250000，就有栈溢出的危险。相反，使用广搜来求连通块，则不存在栈溢出的顾虑。在时间复杂度上，两种方法的用时差不多。
以下代码 1 用深搜的方式实现，代码 2 用广搜的方式实现。
【代码 1】

```
1. #include<iostream>
2. using namespace std;
3. int xx[4]={-1,0,1,0};
4. int yy[4]={0,1,0,-1};
5. char g[110][210];
```

oiClass

```
int n,ans;
void dfs(int x,int y){  //清除 x，y 所在的连通块
    g[x][y]='*';  //将 x，y 的点标记已走过
    for(int i=0;i<4;i++){  //枚举 x，y 的相邻点
        int nx=x+xx[i];
        int ny=y+yy[i];
        if(g[nx][ny]>='a'&&g[nx][ny]<='z'){//邻接点是字母的话
            dfs(nx,ny);  //递归进去
        }
    }
}
int main(){
    cin>>n;
    cin.getline(g[0],201);  //过滤首行回车
    for(int i=1;i<=n;i++){
        cin.getline(g[i]+1,210);//每次读入一行字符串，从 1 位置开始存
    }
    for(int i=1;i<=n;i++){  //枚举每个点
        for(int j=1;j<=200;j++){
            if(g[i][j]>='a'&&g[i][j]<='z'){//该点为字母
                dfs(i,j);  //将该点所在连通块清除
                ans++;  //连通块数量加 1
            }
        }
    }
    cout<<ans;
}
```

【代码 2】

```
#include<iostream>
#include<cstring>
#include<queue>  //队列所用头文件
using namespace std;
struct point{  //结构体类型，表示一个点的坐标
    int x,y;
};
int xx[4]={-1,0,1,0};
int yy[4]={0,1,0,-1};
char g[110][210];
int n,ans;
void bfs(int x,int y){
    queue<point> q;  //声明队列，队列的类型是 point 结构体类型
    q.push((point){x,y});  //初始状态入队
    g[x][y]='*';  //标记初始状态已被访问过了
```

```
    while(!q.empty()){  //当队列不为空的
        point f=q.front();  //取得队头结点的值
        q.pop();  //队头结点出队
        for(int i=0;i<4;i++){  //枚举队头结点的邻接点（拓展状态）
            int nx=f.x+xx[i];
            int ny=f.y+yy[i];
            if(g[nx][ny]>='a'&&g[nx][ny]<='z'){ //如果邻接点是字母
                g[nx][ny]='*';  //标记邻接点已被访问
                q.push((point){nx,ny}); //邻接点入队
            }
        }
    }
}
int main(){
    cin>>n;
    cin.getline(g[0],201);  //主程序部分与代码 1 一样，这里不做注释
    for(int i=1;i<=n;i++){
        cin.getline(g[i]+1,210);
    }
    for(int i=1;i<=n;i++){
        for(int j=1;j<=200;j++){
            if(g[i][j]>='a'&&g[i][j]<='z'){
                bfs(i,j); //广搜求连通块
                ans++;
            }
        }
    }
    cout<<ans;
}
```

拯救 oiclass

【题目描述】

台风“山竹”从西太平洋呼啸而来，15 级风力带来强降雨和海水倒灌，于是，oiclass 就这样被水淹没了。

还好 oiclass 在某些重要的地方起一些围墙，用*号表示，而一个封闭的*号区域洪水是进不去的……

现在给出 oiclass 的围墙建设图，请计算出 oiclass 没被淹到的重要区域（由"0"表示）有多少。

oiclass 处于危难中，需要你的救援……

【输入】

第一行是两个数，x 和 y（x,y<=500）

第二行及以下是一个由*和 0 组成的 x*y 的图。

【输出】

输出没被水淹没的“0”的数量。

【样例输入】

样例输入 1

```
4 5
00000
00*00
0*0*0
00*00
```

样例输入 2

```
5 5
*****
*0*0*
**0**
*0*0*
*****
```

【样例输出】

样例输出 1

```
1
```

样例输出 2

```
5
```

【分析】

本题也是求连通块，每个被水淹没的区域就是一个连通块，剩下的 0 就是没被水淹的，即所求面积，所以，我们的解题思路可以将总面积-被水淹没的面积=所求的面积。但是，水可以从四面八方进来，我们需要枚举每个边缘上的点，求出每个被水淹的连通块。其实，我们只要在外围加多一圈，则原先边缘上的每个点则和外围形成一个连通块，我们只需要从外围任一点出发求连通块便可以求出整个被水淹的区域。

【代码】

```cpp
#include<iostream>
#include<queue>
using namespace std;
struct point{
    int x,y;
};
int n,m,cnt;
int xx[4]={-1,0,1,0},yy[4]={0,1,0,-1};
bool g[502][502];
char ch;
queue<point> q;
bool check(int x,int y){
    if(x<0||y<0||x>n+1||y>m+1)return false;
    if(g[x][y])return false;
    return true;
}
```

```
int bfs(int x,int y){
    int sum=0;
    g[x][y]=true;
    q.push((point){x,y});  //初始状态入队
    sum++;
    while(!q.empty()){
        point f=q.front();  //取出队头结点
        q.pop();
        for(int i=0;i<4;i++){  //枚举队头结点的邻接点
            int nx=f.x+xx[i];
            int ny=f.y+yy[i];
            if(check(nx,ny)){ //如果该点可访问
                g[nx][ny]=true;  //标记该点访问过了
                q.push((point){nx,ny}); //该点入队
                sum++;  //连通块数量加 1
            }
        }
    }
    return sum;
}
int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            cin>>ch;
            if(ch=='*'){
                g[i][j]=true;
                cnt++; //求出*号的个数
            }
        }
    }
    cout<<(n+2)*(m+2)-bfs(0,0)-cnt;//总面积减去被水淹的面积减去*号的面积
}
```

广搜求无权最短路

在上例中，广搜也可以像深搜一样求连通块，广搜有没有什么功能是相对深搜而言别具优势的呢？那就是求无权最短路。这里所谓的无权，可以理解为权值为 1，或者说每一次状态转移的权值都是一样的无差别的。以下图为例，我们要求一条从 s 走到 t 的最短路。

S	1	2
3	4	5
6	T	7

如果用深搜的方式，它可能会走出一条路径 s>1>2>5>7>t，这条路径需要 5 步，还不能确定是最优的，在回溯的时候，会继续走出 s>1>2>5>4>t,甚至还会走出 s>1>2>5>4>3>6>t 路径，总之，利用深搜的方式求最短路，它需要枚举出所有的路径，然后在所有的路径中比较哪条路径最短。

而使用广搜求最短路则具有天生的优势，因为广搜是按层次遍历的。上例中，从 s 出发，第一层拓展了 1，3，第二层拓展了 2、4、6，第三层拓展了 5，t，在第三层的时候拓展到了 t，则这个时候的 t 是距离起点 s 最近的，一旦找到就可以马上确定 s 到 t 的路径长度，不需要枚举所有路径。这便是广搜在找最短路的优势。当然，前提是状态的每一次转移的权值是等价的，也即是无权最短路。

以下例题，可以便于你理解广搜求最短路。

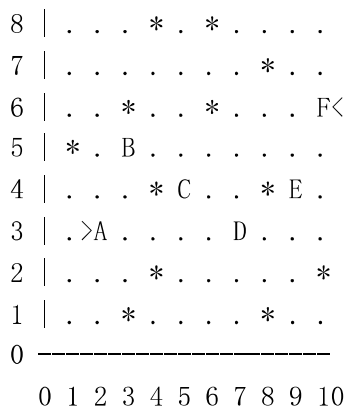
武士风度的牛

【题目描述】
农民 John 有很多牛，他想交易其中一头被 Don 称为 The Knight 的牛。这头牛有一个独一无二的超能力，在农场里像 Knight 一样地跳（就是我们熟悉的象棋中马的走法）。虽然这头神奇的牛不能跳到树上和石头上，但是它可以在牧场上随意跳，我们把牧场用一个 x，y 的坐标图来表示。
这头神奇的牛像其它牛一样喜欢吃草，给你一张地图，上面标注了 The Knight 的开始位置，树、灌木、石头以及其它障碍的位置，除此之外还有一捆草。现在你的任务是，确定 The Knight 要想吃到草，至少需要跳多少次。The Knight 的位置用'K'来标记，障碍的位置用'*'来标记，草的位置用'H'来标记。
这里有一个地图的例子：

```
11 | . . . . . . . . . .
10 | . . . . * . . . . .
 9 | . . . . . . . . . .
 8 | . . . * . * . . . .
 7 | . . . . . . . * . .
 6 | . . * . . * . . . H
 5 | * . . . . . . . . .
 4 | . . . * . . . * . .
 3 | . K . . . . . . . .
 2 | . . . * . . . . . *
 1 | . . * . . . . * . .
 0 ---------------------
   0 1 2 3 4 5 6 7 8 9 10
```

The Knight 可以按照下图中的 A,B,C,D...这条路径用 5 次跳到草的地方（有可能其它路线的长度也是 5）：

```
11 | . . . . . . . . . .
10 | . . . . * . . . . .
 9 | . . . . . . . . . .
```



【输入】

第一行： 两个数，表示农场的列数(<=150)和行数(<=150)

第二行..结尾：如题目描述的图。

【输出】

一个数，表示跳跃的最小次数。

【样例输入】

```
10 11
..........
....*.....
..........
...*.*....
.......*..
..*..*...H
*.........
...*...*..
.K........
...*.....*
..*....*..
```

【样例输出】

```
5
```

【分析】

本题从起点出发，每次按层次遍历出来，最先到达终点，则所需步骤即起点到终点的最短步骤。一旦发现终点，即可输出步骤然后终止程序。

【代码】

```
#include<iostream>
#include<queue>
using namespace std;
struct point{
    int x,y,s;  //x，y 表示坐标，s 表示从起点走到 x，y 坐标所需的步骤
};
queue<point> q;
int n,m,sx,sy,tx,ty;
int xx[8]={-2,-2,-1,1,2,2,1,-1};  //牛的 8 个遍历点
int yy[8]={-1,1,2,2,1,-1,-2,-2};
```

oiClass

```
bool g[152][152];
char ch;
bool check(int x,int y){
    if(x<1||y<1||x>n||y>m)return false; //该点越界不可走
    if(g[x][y]==false)return false; //该点已走过不可走
    return true;
}
int main(){
    cin>>m>>n;
    for(int i=1;i<=n;i++){  //输入地图
        for(int j=1;j<=m;j++){
            cin>>ch;
            if(ch=='*')continue;//不可走为 false
            if(ch=='K'){  //记录起点
                sx=i;
                sy=j;
            }
            if(ch=='H'){  //记录终点
                tx=i;
                ty=j;
            }
            g[i][j]=true;  //可以走为 true
        }
    }
    q.push((point){sx,sy,0});  //初始状态入队
    g[sx][sy]=false;  //标记该点走过
    while(!q.empty()){
        point f=q.front();  //取出队头结点
        q.pop();
        for(int i=0;i<8;i++){  //枚举八个邻接点
            int nx=f.x+xx[i];
            int ny=f.y+yy[i];
            if(check(nx,ny)){ //如果该邻接点可走
                if(nx==tx&&ny==ty){  //如果该邻接点为终点
                    cout<<f.s+1<<endl; //输出步骤
                    return 0;  //结束程序
                }
                q.push((point){nx,ny,f.s+1}); //邻接点入队
                g[nx][ny]=false;  //标记邻接点走过
            }
        }
    }
}
```

三维地牢

【题目描述】
你正身陷一个 3 维的地牢中，需要找到最快的路径离开。地牢由立方体单元组成，这些单元或者是岩石，或者不是岩石。你可以用一分钟的时间向东，向西，向南，向北，向上，或者向下，走到下一个单元中。你不能走对角线，迷宫的周围是坚硬的岩石。
可能逃离地牢吗？如果可能，要多长时间？
【输入】
输入由许多地牢组成。每个地牢的描述的第一行是 3 个整数 L，R 和 C（大小限制在 30 以内）。
L 表示地牢的层数。
R 和 C 表示地牢的行和列。
后面跟着 L 块，每块包含 R 行，每行包含 C 个字符，每个字符表示地牢的一个单元。一个单元如果由岩石构成，用‘#’表示；空的单元用‘.’表示；你所在的起始位置用‘S’表示；出口用‘E’表示。每层地牢后跟一空行。输入以为 L，R 和 C 赋 0 结束。
【输出】
每个迷宫产生一行输出，如果可以到达出口，输出形式为
Escaped in x minute(s).
其中 x 是逃离的最短时间。
如果不可能逃离，输出
Trapped!
【样例输入】

```
3 4 5
S....
.###.
.##..
###.#

#####
#####
##.##
##...

#####
#####
#.###
####E

1 3 3
S##
#E#
###
```

oiClass

0 0 0

【样例输出】

```
Escaped in 11 minute(s).
Trapped!
```

【分析】

本题与上题类似，都是给出起点和终点，找无权最短路。不同的是，所给地图是三维的，我们用一个三维数组来模拟就可以了，详见代码。

【代码】

```
#include<iostream>
#include<queue>
#include<cstring>
using namespace std;
struct point{
    int x,y,z,s;  //x,y,z 表示坐标，s 表示步骤
};
queue<point> q;
int xx[6]={0,0,0,0,1,-1};  //一个点能走的坐标偏移量，分别是前后左右上下六个
int yy[6]={-1,0,1,0,0,0};
int zz[6]={0,1,0,-1,0,0};
int l,r,c,sx,sy,sz,tx,ty,tz;
bool g[35][35][35];  //三维地图
char ch;
int bfs(int x,int y,int z){
    q.push((point){x,y,z,0});
    g[x][y][z]=false;
    while(!q.empty()){
        point f=q.front();
        q.pop();
        for(int i=0;i<6;i++){
            int nx=f.x+xx[i];
            int ny=f.y+yy[i];
            int nz=f.z+zz[i];
            if(g[nx][ny][nz]){
                if(nx==tx&&ny==ty&&nz==tz){
                    return f.s+1;
                }
                q.push((point){nx,ny,nz,f.s+1});
                g[nx][ny][nz]=false;
            }
        }
    }
    return -1;
}
int main(){
```

```
    while(cin>>l>>r>>c){  //循环输入每个样例
        if(l==0&&r==0&&c==0)break;
        memset(g,false,sizeof(g));  //初始化地图
        while(!q.empty())q.pop();  //初始化队列，忽略则队列会因塞满出错
        for(int i=1;i<=l;i++){  //读入地图
            for(int j=1;j<=r;j++){
                for(int k=1;k<=c;k++){
                    cin>>ch;
                    if(ch=='#')continue;
                    if(ch=='S'){ //标记起点
                        sx=i,sy=j,sz=k;
                    }
                    if(ch=='E'){  //标记终点
                        tx=i,ty=j,tz=k;
                    }
                    g[i][j][k]=true;
                }
            }
        }
        int ans=bfs(sx,sy,sz); //bfs()从起点找最短路，返回所需步骤
        if(ans==-1){
            cout<<"Trapped!"<<endl;
        }else{
            cout<<"Escaped in "<<ans<<" minute(s)."<<endl;
        }
    }
}
```

乳草的入侵

【题目描述】

Farmer John 一直努力让他的草地充满鲜美多汁的而又健康的牧草。可惜天不从人愿，他在植物大战人类中败下阵来。邪恶的乳草已经在他的农场的西北部份佔领了一片立足之地。

草地像往常一样，被分割成一个高度为 Y(1 <= y <= 100)，宽度为 X(1 <= x <= 100)的直角网格。(1,1)是左下角的格（也就是说坐标排布跟一般的 X,Y 坐标相同）。乳草一开始佔领了格(Mx,My)。每个星期，乳草传播到已被乳草佔领的格子四面八方的每一个没有很多石头的格（包括垂直与水平相邻的和对角线上相邻的格）。1 周之后，这些新佔领的格又可以把乳草传播到更多的格裡面了。

Bessie 想要在草地被乳草完全佔领之前尽可能的享用所有的牧草。她很好奇到底乳草要多久才能佔领整个草地。如果乳草在 0 时刻处於格(Mx,My)，那麼还在那个时刻它们可以完全佔领入侵整片草地呢（对给定的数据总是会发生）？

草地由一个图片表示。"."表示草，而"*"表示大石。比如这个 X=4, Y=3 的例子。

```
....
..*.
```

.**.

如果乳草一开始在左下角（第 1 排，第 1 列），那麼草地的地图将会以如下态势发展：

```
....  ....  MMM.  MMMM  MMMM
..*.  MM*.  MM*.  MM*M  MM*M
M**.  M**.  M**.  M**.  M**M
```

星期数 0 1 2 3 4

乳草会在 4 星期后佔领整片土地。

【输入】

* 第一行：四个由空格隔开的整数：X, Y, Mx, My
* 第 2 到第 Y+1 行：数据的第 y+1 行由 X 个字符（"."表示草地，"*"表示大石），描述草地的第(Y+2-y)行。

【输出】

* 第一行：一个单独的整数表示最后一个不是大石块的格子被乳草佔领的星期数。

【样例输入】

```
4 3 1 1
....
..*.
.**.
```

【样例输出】

```
4
```

【分析】

本题虽然不是求最短路径，而是求最短时间，按照乳草的扩展规则，以广搜的方式一圈一圈地往外扩展，直到队列为空，则最后一个被扩展到的点所需的时间，即最少时间。

【代码】

```
#include<iostream>
#include<queue>
using namespace std;
struct point{
    int x,y,s;
};
queue<point> q;
int n,m,x,y,sx,sy,ans,k;
int xx[8]={-1,-1,-1,0,0,1,1,1};
int yy[8]={-1,0,1,-1,1,-1,0,1};
bool g[102][102];
char ch;
int main(){
    cin>>m>>n>>x>>y;
    for(int i=1;i<=n;i++){  //读入地图
        for(int j=1;j<=m;j++){
            cin>>ch;
            if(ch=='.')g[i][j]=true;
        }
    }
```

```
    sy=x;  //转换坐标系
    sx=n-y+1;
    q.push((point){sx,sy,0});  //初始状态入队
    g[sx][sy]=false;
    while(!q.empty()){
        point f=q.front();  //取出队头结点
        q.pop();
        ans=max(ans,f.s);  //统计最后的时间
        for(int i=0;i<8;i++){  //拓展 8 个邻接点
            int nx=f.x+xx[i];
            int ny=f.y+yy[i];
            if(g[nx][ny]){ //如果该点可走
                q.push((point){nx,ny,f.s+1});  //入队
                g[nx][ny]=false;
            }
        }
    }
    cout<<ans;  //最后一个出队的时间即最晚时间
}
```

流星雨

【题目描述】

贝茜听说了一个骇人听闻的消息：一场流星雨即将袭击整个农场，由于流星体积过大，它们无法在撞击到地面前燃烧殆尽，届时将会对它撞到的一切东西造成毁灭性的打击。很自然地，贝茜开始担心自己的安全问题。以 FJ 牧场中最聪明的奶牛的名誉起誓，她一定要在被流星砸到前，到达一个安全的地方（也就是说，一块不会被任何流星砸到的土地）。如果将牧场放入一个直角坐标系中，贝茜现在的位置是原点，并且，贝茜不能踏上一块被流星砸过的土地。

根据预报，一共有 M 颗流星(1 <= M <= 50,000)会坠落在农场上，其中第 i 颗流星会在时刻 T_i (0 <= T_i <= 1,000)砸在坐标为(X_i, Y_i) (0 <= X_i <= 300; 0 <= Y_i <= 300)的格子里。流星的力量会将它所在的格子，以及周围 4 个相邻的格子都化为焦土，当然贝茜也无法再在这些格子上行走。

贝茜在时刻 0 开始行动，它只能在第一象限中，平行于坐标轴行动，每 1 个时刻中，，她能移动到相邻的（一般是 4 个）格子中的任意一个，当然目标格子要没有被烧焦才行。如果一个格子在时刻 t 被流星撞击或烧焦，那么贝茜只能在 t 之前的时刻在这个格子里出现。

请你计算一下，贝茜最少需要多少时间才能到达一个安全的格子。

【输入】

* 第 1 行：1 个正整数：M

* 第 2..M+1 行：第 i+1 行为 3 个用空格隔开的整数：X_i，Y_i，以及 T_i

【输出】

* 第 1 行：输出 1 个整数，即贝茜逃生所花的最少时间。如果贝茜无论如何都无法在流星雨中存活下来，输出-1

【样例输入】

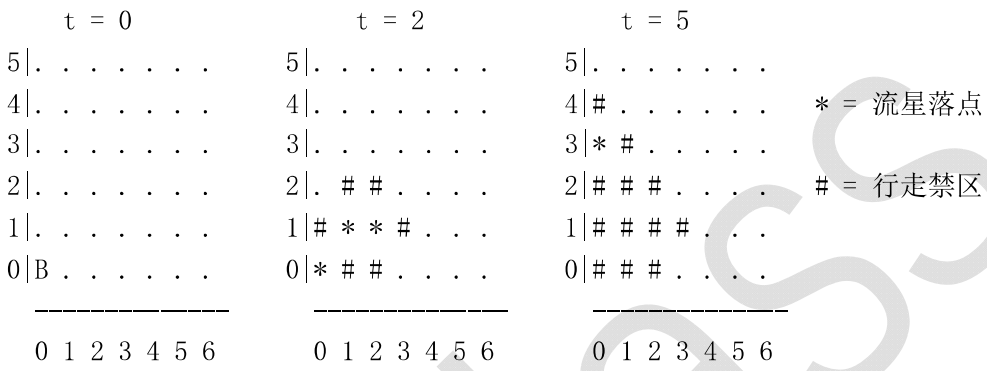
4
0 0 2
2 1 2
1 1 2
0 3 5

【样例输出】

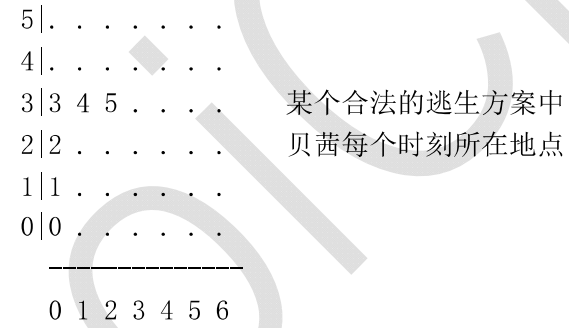
5

【提示】

输入说明：一共有 4 颗流星将坠落在农场，它们落地点的坐标分别是(0, 0)，(2, 1)，(1, 1)以及(0, 3)，时刻分别为 2，2，2，5。



输出说明:如果我们观察在 t=5 时的牧场，可以发现离贝茜最近的安全的格子是(3,0)——不过由于早在第二颗流星落地时，贝茜直接跑去(3,0)的路线就被封死了。离贝茜第二近的安全格子为(4,0)，但它的情况也跟(3,0)一样。再接下来的格子就是在(0,5)-(5,0)这条直线上。在这些格子中，(0,5)，(1,4)以及(2,3)都能在 5 个单位时间内到达。



【分析】

本题是一道很有意思的广搜题。我们可以把陨石坠地的时间统计出来，如果 a[x][y]的值表示陨石落到这个位置的时间，如果 a[x][y]为-1 表示该点不会被陨石砸到。为什么要用-1 表示呢？因为陨石可以在 0 时刻落地。那么，本题则变成从起点出发，找到一个 a[x][y]为-1 的最短时间，要求经过点 x,y 的时间小于 a[x][y]。

【代码】

```
#include<iostream>
#include<queue>
#include<cstring>
using namespace std;
struct point{
    int x,y,t;
};
```

```
int g[500][500],m,x,y,t;
int xx[4]={-1,0,1,0},yy[4]={0,1,0,-1};
queue<point> q;
bool bound(int x,int y){
    if(x<0||y<0)return false;
    return true;
}
int main(){
    memset(g,-1,sizeof(g));
    cin>>m;
    for(int i=1;i<=m;i++){
        cin>>x>>y>>t;  //读入陨石坠地的坐标和时间
        if(bound(x,y)){ //边界判断
            if(g[x][y]==-1)g[x][y]=t; //如果没有陨石砸过则直接赋值
            else g[x][y]=min(g[x][y],t); //如果砸过则比较时间
        }
        for(int j=0;j<4;j++){  //扩展陨石的周边时间
            int nx=x+xx[j];
            int ny=y+yy[j];
            if(bound(nx,ny)){  //边界判断
                if(g[nx][ny]==-1)g[nx][ny]=t;//同样判断是否被砸过
                else g[nx][ny]=min(g[nx][ny],t);
            }
        }
    }
    q.push((point){0,0,0});  //初始状态入队
    g[0][0]=-2;  //标记该点走过
    while(!q.empty()){
        point f=q.front();  //取出队头结点
        q.pop();
        for(int i=0;i<4;i++){  //枚举四个邻接点
            int nx=f.x+xx[i];
            int ny=f.y+yy[i];
            if(bound(nx,ny)&&g[nx][ny]!=-2){ //该点在边界内且没走过
                if(g[nx][ny]==-1){ //该点不会被陨石砸到，即为目标点
                    cout<<f.t+1<<endl;  //输出时间
                    return 0;  //结束程序
                }
                if(f.t+1<g[nx][ny]){  //判断该点能够走
                    q.push((point){nx,ny,f.t+1});  //入队
                    g[nx][ny]=-2;
                }
            }
        }
```

```
52.     }
53.     cout<<-1<<endl;  //如果没在中途退出，则表示找不到安全点
54. }
```

oiClass