**Nightmare**

**Time Limit: 2000/1000 MS (Java/Others)    Memory Limit: 65536/32768 K (Java/Others)  
Total Submission(s): 10434    Accepted Submission(s): 5070**

**Problem Description**

Ignatius had a nightmare last night. He found himself in a labyrinth with a time bomb on him. The labyrinth has an exit, Ignatius should get out of the labyrinth before the bomb explodes. The initial exploding time of the bomb is set to 6 minutes. To prevent the bomb from exploding by shake, Ignatius had to move slowly, that is to move from one area to the nearest area(that is, if Ignatius stands on (x,y) now, he could only on (x+1,y), (x-1,y), (x,y+1), or (x,y-1) in the next minute) takes him 1 minute. Some area in the labyrinth contains a Bomb-Reset-Equipment. They could reset the exploding time to 6 minutes.  
  
Given the layout of the labyrinth and Ignatius' start position, please tell Ignatius whether he could get out of the labyrinth, if he could, output the minimum time that he has to use to find the exit of the labyrinth, else output -1.  
  
Here are some rules:  
1. We can assume the labyrinth is a 2 array.  
2. Each minute, Ignatius could only get to one of the nearest area, and he should not walk out of the border, of course he could not walk on a wall, too.  
3. If Ignatius get to the exit when the exploding time turns to 0, he can't get out of the labyrinth.  
4. If Ignatius get to the area which contains Bomb-Rest-Equipment when the exploding time turns to 0, he can't use the equipment to reset the bomb.  
5. A Bomb-Reset-Equipment can be used as many times as you wish, if it is needed, Ignatius can get to any areas in the labyrinth as many times as you wish.  
6. The time to reset the exploding time can be ignore, in other words, if Ignatius get to an area which contain Bomb-Rest-Equipment, and the exploding time is larger than 0, the exploding time would be reset to 6.

**Input**

The input contains several test cases. The first line of the input is a single integer T which is the number of test cases. T test cases follow.  
Each test case starts with two integers N and M(1<=N,Mm=8) which indicate the size of the labyrinth. Then N lines follow, each line contains M integers. The array indicates the layout of the labyrinth.  
There are five integers which indicate the different type of area in the labyrinth:  
0: The area is a wall, Ignatius should not walk on it.  
1: The area contains nothing, Ignatius can walk on it.  
2: Ignatius' start position, Ignatius starts his escape from this position.  
3: The exit of the labyrinth, Ignatius' target position.  
4: The area contains a Bomb-Reset-Equipment, Ignatius can delay the exploding time by walking to these areas.

**Output**

For each test case, if Ignatius can get out of the labyrinth, you should output the minimum time he needs, else you should just output -1.

**Sample Input**

3

3 3

2 1 1

1 1 0

1 1 3

4 8

2 1 1 0 1 1 1 0

1 0 4 1 1 0 4 1

1 0 0 0 0 0 0 1

1 1 1 4 1 1 1 3

5 8

1 2 1 1 1 1 1 4

1 0 0 0 1 0 0 1

1 4 1 0 1 1 0 1

1 0 0 0 0 3 0 1

1 1 4 1 1 1 1 1

**Sample Output**

4

-1

13

**Author**

又是一道好题，然而我还是没解出来（艹）

题目大意：有这么个图，有这个么主人公在2号作为起点，想要走到终点3.图中1为可走道路，0为不能走的道路，4是增加boom时间的点。初始主人公身上有个时间为6的炸弹。走一步时间-1，如果变成了0，即使走到了终点3或者是使炸弹时间初始化的4，也算是死亡了。

通过读题，我立刻想到要求最短路径，肯定用BFS，这样先确定了题目的方向。

然后，在做的过程中发现了此题的难点就是：一般的迷宫BFS每个点只能走一次，可以通过设置vis[][]来限制走回头路从而求出最短路径。而这道题恰恰说出了每个点可以走无限次，这就造成了一个问题：当你遇到一个点时是继续走没走过的，还是折回走走过的点？因为答案都可能存在于两种情形中。我开始就想到不对每个点设置vis[][]，这样就可以走走过的点了，然而由于没想好怎么避免死循环放弃了这种思路，事实证明，设置vis[][]还是有需要的，只是不是用来记录该点走了还是没走。这时，我灵光一现，想到迪杰斯特拉算法，就是把本题当作最短路径去求，利用输入的矩阵构建一个联通图，来求出最短路径(就是把起点终点以及恢复点取出来作为点集，然后搜索找出长度小于等于5的所有边做成图，再迪杰斯特拉一下，可能比较麻烦)，但是由于想用BFS做所以放弃了这种思路，看了discuss，发现有人果然这么做(开熏)。继续自己的BFS，发现难点在于4这个位置和是否走回头路的处理。对于到了4这个点，我是不确定该怎么办的，因为可以无限来到这个点恢复6，所以造成路径是否应该包含这个点的问题。如果包含，可能会在某种情况下增加路径长度，得不到正确的最小解，如果不包含，可能因为time没有及时补充，而失去了可能唯一的解。这点思考了好久，没想出来。然后就是重复的问题，我想到了比较重复的点和当前的点的time信息，并如果当前time大于马上要重复走的点，就走该点，但是我不确定这样为什么能求出答案。看了题解，发现题解也是这么写的，并作出了解释。下面给出具体的题解。

一个比较好的题解：

这一题与之前的搜索题有一定的区别，最大的区别就是这一题的迷宫可以走“回头路”。我认为这也是本题的最大难点，换句话就是你解决了这个问题也就相当于完成了90%。所以这一题不能像之前搜索一样的去标记走过的路，因为可能还要再走一趟。这样就引发了一个问题，如果不做处理的话，那么就永远走不出来了。所以不是不做处理而是做特殊的处理：利用前后两次踏上同一位置距离炸弹爆炸的时间来判断，如果第二次走到该点，使得该点剩的时间比第一次长的话，那么就入队；反之，入队就没有什么意义了(这一步避免了死循环，也就解释了我的疑惑，因为如果时间还短的话，说明只是在走无用的回头路，既浪费了时间又增加了路径长度，而时间更长，说明很重要的一点就是当前位置进行过了4的资源补充，不然只会比以前走过的点的time短，这样也推出了下面的结论，因为4如果走多次每次时间也都是6，不仅会增加路径长度，也无意义。这里实际上是**贪心**的思想)。

从以上分析来看4的所在的位置无论怎么走，都只走一次，所以可以访问过的4标记为0.这样下一次就不会访问到了（这一步也避免了死循环）。所以也可以用这个作为另一种解法。

当然如果将以上两种方式结合起来的话，那么可以提高搜索的效率。

另一种题解思路：

规则很容易理解，但是题目中一句：如果你需要，可以重复到4这个点补充炸弹时间。最初是让我犯糊涂的一个点，其实广搜的题目中有类似的很多这样的题目，我们用一个贪心思想来解答这个困难点：如果我回头了，步数就多了，而且如果从当前补充点继续走，走不到终点或者是下一个补充点，我们无论多少次补充时间都是徒劳，该走不出去还是走不出去，所以我们设置这个点只能走一次。如果走过了，图改4为0。

第一个难点我们用贪心的思想来解决了，然后我们还有一个困难点如题中样例输入3：

5 8

1 2 1 1 1 1 1 4

1 0 0 0 1 0 0 1

1 4 1 0 1 1 0 1

1 0 0 0 0 3 0 1

1 1 4 1 1 1 1 1

这里我们很容易看出来，从2起点开始出发，我们只能向左走，向右走是走不到终点的。然后向左下走的时候，我们需要进入4然后再出4，这个时候我们就涉及到一个重复走点的问题。通常情况下，入门的搜索题中我们设置的二维数组vis[][]用来标记这个点是否走过。用0/1来表示是否走过，以避免走回头路（多余）.而且很可能进入死循环。但是这里如果这样标记的话 我们走到左边第一个4的时候，我们就出不去了，这里我们要解决重路的问题。这个题，涉及到的变量并不多，无非x,y(坐标)step(步数).time(主角身上boom的剩余时间.)我们想再x.y上边做文章是很不容易的。在step上做标记也是很难实现的.这个时候我们还是直接想到了再boom的剩余时间上做文章，我们可以标记走到这步的时候boom的剩余时间是多少.如果走到这步的时候的boom剩余时间和上次标记的boom剩余时间不一样，那就说明我得到了补充。然而我们另开数组是不合算的，我们直接在vis数组上做文章就可以了：

具体代码：

#include <iostream>

#include <queue>

#include <string.h>

using namespace std;

typedef struct{

int x,y,step,time; //step表示所走步数，time表示离爆炸的时间

}coordinate;

int n,m; //表示迷宫的规格

int labyrinth[10][10],mark[10][10]; //labyrinth表示地图，mark标记访问的点

const int dx[4]={0,0,1,-1},dy[4]={1,-1,0,0}; //位移参量

coordinate start;

queue<coordinate>Q;

void Init(){

memset(labyrinth,0,sizeof(labyrinth));

memset(mark,0,sizeof(mark));

}

bool Inborder(int x,int y){ //判断是否出界

if(x<0||x>=n||y<0||y>=m)

return false;

return true;

}

void BFS(){

coordinate now,next;

while(!Q.empty()) //队列初始化

Q.pop();

Q.push(start);

mark[start.x][start.y] = 6; //将起始点的时间记为6

while( !Q.empty() ){

now=Q.front();

Q.pop();

for(int i=0 ; i<4 ; i++ ){

next.x = now.x + dx[i]; //进行四个方向的遍历

next.y = now.y + dy[i];

if( labyrinth[next.x][next.y]!=0 && Inborder(next.x,next.y) ){

//该点不是0且不出界则访问该点

next.step = now.step + 1;

next.time = now.time - 1;

if( labyrinth[next.x][next.y] == 4 ){ //这个贪心是个关键

labyrinth[next.x][next.y] = 0;

next.time = 6;

}

else if( labyrinth[next.x][next.y] == 3 ){

if(next.time>0)

cout<<next.step<<endl; //直接输出是因为还是按照类似迷宫的标准bfs步骤进行的搜索，只是加了些限制条件而已，所以到达终点时，一定是最小解。

else

cout<<-1<<endl;

return ;

}

if( next.time>1 && mark[next.x][next.y]<next.time ){

//满足条件则入列

mark[next.x][next.y] = next.time;

Q.push(next);

}

}

}

}

cout<<-1<<endl;

return ;

}

int main()

{

int T;

cin>>T;

while(T--){

cin>>n>>m;

Init();

for(int i=0;i<n;i++){

for(int j=0;j<m;j++){

cin>>labyrinth[i][j];

if(labyrinth[i][j]==2){

//查找起点位置

start.x=i;

start.y=j;

start.step=0;

start.time=6;

}

}

}

BFS();

}

return 0;

}

DFS版：(难点就是如何在不加vis情况下进行dfs)

#include<iostream>

#include<cstdio>

#include<cstring>

#include<climits>

using namespace std;

const int INF=0x7fffffff;

#define maxn INF

int n,m;

int puzz[10][10];//迷宫数组

int sx,sy;

int time[10][10];//时间记录

int road[10][10];//路径长度记录

int res;//结果

void dfs(int x,int y,int tim,int step)

{

if(x<0||y<0||x>=n||y>=m)return;//越界

if(tim<=0||step>=res)return;//炸死或者当前解差于最优解。

if(puzz[x][y]==0)return;//不能走

if(puzz[x][y]==3)//到达3

{

if(res>step)

res=step;//更新最优解

}

if(puzz[x][y]==4)//dfs中4位置可以多次访问,因为bfs只是全路径扫描一遍得出最优结果，所以必须去掉4，不然就会出现无意义的重复路径。而dfs需要回溯完成路径构造，所以如果访问过4就去掉，就会造成回溯失败，此解答树路断裂造成递归无法正常走完所有解空间。所以不能去，所以只有通过标记的方法去环，正是下面的那句话。

tim=6;

if(time[x][y]>=tim&&step>=road[x][y])//通过数组来判断一个点是否经过了多次。什么才算第2次经过呢？很显然应该是当前路径经过该点的长度比该点之前的路径的数组记录长，并且剩余时间也应该大于第一次才不剪掉。去环(不走无意义回头路)操作，可以对照蓝字思路的例子看下去环是怎么实现的，关键就在于time[x][y]>=tim这句话。极其巧妙

return;

time[x][y]=tim;

road[x][y]=step;

dfs(x,y+1,tim-1,step+1);

dfs(x,y-1,tim-1,step+1);

dfs(x+1,y,tim-1,step+1);

dfs(x-1,y,tim-1,step+1);

return;

}

int main(void)

{

int t;

cin>>t;

while(t--)

{

cin>>n>>m;

for(int i=0;i<n;i++)

for(int j=0;j<m;j++)

{

cin>>puzz[i][j];

if(puzz[i][j]==2)

{

sx=i;

sy=j;

}

road[i][j]=INT\_MAX;

}

memset(time,0,sizeof(time));

res=maxn;

dfs(sx,sy,6,0);

if(res==maxn)

cout<<-1<<endl;

else

cout<<res<<endl;

}

return 0;

}