import java.util.\*;

class Region

{

int x;//row number

int y;//column number

int h;//height

boolean isDrain;//true if it has a direct path for water to the sea. It is similar to a group of regions marked as 1

boolean isDone;//true if the point is already computed and checked

boolean isFlood;//true if the region is flooded

boolean isFloodBorder;//true if the region is adjacent to a group of flooded regions

int state;//state of the region

Region(int x,int y,int h,boolean isDrain,boolean isDone,int state)//constructor

{

this.x=x;

this.y=y;

this.h=h;

this.isDrain=isDrain;

this.isDone=isDone;

this.state=state;

}

}

public class Q8

{

static Region r[][]=new Region[10][10];//array of regions

static Scanner sc=new Scanner(System.in);//for input

static Stack<Region> stk=new Stack<Region>();//stack representing group of regions that are flooded

static Stack<Region> minimas=new Stack<Region>();//stack of all local minimas as they must be present in a flooded region

static Comparator<Region> cmp=new Comparator<Region>(){//comparator for priority que

public int compare(Region r1,Region r2)

{

return r1.h-r2.h;

}

};

static PriorityQueue<Region> pq=new PriorityQueue<Region>(100,cmp);//priority que of all regions that are adjacent to a flooded region

public static void main(String args[])

{

for(int i=0;i<10;i++)

{

for(int j=0;j<10;j++)

{

r[i][j]=new Region(i,j,sc.nextInt(),false,false,2);

}

}

setDrains();//sets all elements that can possibly be drains

findMinimas();//finds all local minimas

setFloods();//sets all regions that are flooded

setOthers();//set other non flooded regions as isolated or disconnected with state 0

display();//prints the output matrix

}

static void setDrains()

{

//traverses through all edges, sets them as drains and marks them as done

//then it recursively calls for all the adjacent elements and checks whether they can act as drains for all other elements

int i;

//traversing through the coastline

for(i=0;i<10;i++)

//checking isDone to check if the element is not already processed

if(!r[i][0].isDone)

makeDrains(r[i][0],0);

for(i=1;i<10;i++)

if(!r[9][i].isDone)

makeDrains(r[9][i],0);

for(i=0;i<9;i++)

if(!r[i][9].isDone)

makeDrains(r[i][9],0);

for(i=1;i<9;i++)

if(!r[0][i].isDone)

makeDrains(r[0][i],0);

}

static void findMinimas()

{

//finds all local minimas

//a point is a local minima if it is less than atleast one of its neighbours and less than or equal to the rest of its neighbours

//such points shall always be in a group of flooded regions or will be a part of a drain hence will initiate the search for flooded regions

//the above technique will work as drains shall already be marked as done leaving only drains

for(int i=1;i<9;i++)

{

for(int j=1;j<9;j++)

{

if((r[i][j].h<=r[i-1][j-1].h &&

r[i][j].h<=r[i][j-1].h &&

r[i][j].h<=r[i+1][j-1].h &&

r[i][j].h<=r[i-1][j].h &&

r[i][j].h<=r[i+1][j].h &&

r[i][j].h<=r[i-1][j+1].h &&

r[i][j].h<=r[i][j+1].h &&

r[i][j].h<=r[i+1][j+1].h

) &&

(r[i][j].h<r[i-1][j-1].h ||

r[i][j].h<r[i][j-1].h ||

r[i][j].h<r[i+1][j-1].h ||

r[i][j].h<r[i-1][j].h ||

r[i][j].h<r[i+1][j].h ||

r[i][j].h<r[i-1][j+1].h ||

r[i][j].h<r[i][j+1].h ||

r[i][j].h<r[i+1][j+1].h

) && !r[i][j].isDone

)

minimas.push(r[i][j]);

}

}

}

static void setFloods()

{

Region rr;//region under consideration

while(!minimas.empty())

{

rr=minimas.pop();//taking a minima which must be in a flooded region

if(!rr.isDone)

{

//clearing stack and que

pq.clear();

stk.clear();

//marking as flooded

rr.isFlood=true;

//inserting group neighbours (borders)

r[rr.x-1][rr.y-1].isFloodBorder=true;

pq.offer(r[rr.x-1][rr.y-1]);

r[rr.x][rr.y-1].isFloodBorder=true;

pq.offer(r[rr.x][rr.y-1]);

r[rr.x+1][rr.y-1].isFloodBorder=true;

pq.offer(r[rr.x+1][rr.y-1]);

r[rr.x-1][rr.y].isFloodBorder=true;

pq.offer(r[rr.x-1][rr.y]);

r[rr.x+1][rr.y].isFloodBorder=true;

pq.offer(r[rr.x+1][rr.y]);

r[rr.x-1][rr.y+1].isFloodBorder=true;

pq.offer(r[rr.x-1][rr.y+1]);

r[rr.x][rr.y+1].isFloodBorder=true;

pq.offer(r[rr.x][rr.y+1]);

r[rr.x+1][rr.y+1].isFloodBorder=true;

pq.offer(r[rr.x+1][rr.y+1]);

stk.push(rr);

//recursively generating the flooded region

markFloods(stk,rr.h);

}

}

}

static void markFloods(Stack<Region> stk, int maxH)

{

Region rr;//region under consideration

Region rs;

if(pq.isEmpty() == true)

return;

rr=pq.poll();//getting the lowest region adjacet to a group of flooded regions

rr.isFloodBorder=false;

maxH=maxH>rr.h?maxH:rr.h;

if(!rr.isDone)//if this lowest region is not already computed then proceed

{

rr.isFlood=true;//marking as flooded

stk.push(rr);//adding to the stack of flooded regions

//Inserting all regions that are adjacent to the current region and not

//considered yet into the priority que of regions being considered.

//The priority que sorts the regions by height and produces the lowest region on polling.

//If this region is already a part of a flood then it would be marked as done

//the two flooded region groups will be merged into one. As the appended flooded region was

//being drained at the given height, it will cause the current group to also drain hence the

//current group of flooded regions cannot grow further. If the region being polled is a part of a drain then any increase

//in water level beyond its height will be drained causing the rise in water level to cease hence this will also cause the

//flood region to stop growing. If the current region is not processed yet then it can be added to the flooded region groups.

//The code simmulates growing water-level and hence produces the currect answers.

if(!r[rr.x-1][rr.y-1].isFloodBorder && !r[rr.x-1][rr.y-1].isFlood)

pq.offer(r[rr.x-1][rr.y-1]);

if(!r[rr.x][rr.y-1].isFloodBorder && !r[rr.x][rr.y-1].isFlood)

pq.offer(r[rr.x][rr.y-1]);

if(!r[rr.x+1][rr.y-1].isFloodBorder && !r[rr.x+1][rr.y-1].isFlood)

pq.offer(r[rr.x+1][rr.y-1]);

if(!r[rr.x-1][rr.y].isFloodBorder && !r[rr.x-1][rr.y].isFlood)

pq.offer(r[rr.x-1][rr.y]);

if(!r[rr.x+1][rr.y].isFloodBorder && !r[rr.x+1][rr.y].isFlood)

pq.offer(r[rr.x+1][rr.y]);

if(!r[rr.x-1][rr.y+1].isFloodBorder && !r[rr.x-1][rr.y+1].isFlood)

pq.offer(r[rr.x-1][rr.y+1]);

if(!r[rr.x][rr.y+1].isFloodBorder && !r[rr.x][rr.y+1].isFlood)

pq.offer(r[rr.x][rr.y+1]);

if(!r[rr.x+1][rr.y+1].isFloodBorder && !r[rr.x+1][rr.y+1].isFlood)

pq.offer(r[rr.x+1][rr.y+1]);

markFloods(stk,stk.peek().h);

}

else

{

//marking the group of flooded regions as flooded by popping regions one by one from the stack

while(!stk.empty())//looping till the stack is not empty

{

rs=stk.pop();

rs.isDone=true;//marking them as done

rs.isFlood=false;

if(rs.h<maxH)

rs.state=-1;

}

//clearing the que and all its elements

//can be made faster

while(!pq.isEmpty())

{

rr=pq.poll();

rr.isFloodBorder=false;

}

}

}

static void setOthers()

{

//sets all points other than floods and drains

//a group of safe(state 1) regions is the union of drains and non-floodable groups which have atleast one adjacent region

int i,j;

//setting all other points as isolated

for(i=1;i<9;i++)

{

for(j=1;j<9;j++)

if(r[i][j].state==2)

r[i][j].state=0;

}

//marking isolated group of regions adjacent to a safe group as safe

for(i=1;i<9;i++)

{

for(j=0;j<9;j++)

if(r[i][j].state==0 &&

(r[i-1][j-1].state==1 ||

r[i-1][j].state==1 ||

r[i-1][j+1].state==1 ||

r[i][j-1].state==1 ||

r[i][j+1].state==1 ||

r[i+1][j-1].state==1 ||

r[i+1][j].state==1 ||

r[i+1][j+1].state==1))

markSafe(r[i][j]);

}

}

static void markSafe(Region rr)

{

//setting current region earlies marked as isolated as safe

//marking all its adjacent isolated regions as safe

rr.state=1;

if(r[rr.x-1][rr.y-1].state==0)

markSafe(r[rr.x-1][rr.y-1]);

if(r[rr.x-1][rr.y].state==0)

markSafe(r[rr.x-1][rr.y]);

if(r[rr.x-1][rr.y+1].state==0)

markSafe(r[rr.x-1][rr.y+1]);

if(r[rr.x][rr.y-1].state==0)

markSafe(r[rr.x][rr.y-1]);

if(r[rr.x][rr.y+1].state==0)

markSafe(r[rr.x][rr.y+1]);

if(r[rr.x+1][rr.y-1].state==0)

markSafe(r[rr.x+1][rr.y-1]);

if(r[rr.x+1][rr.y].state==0)

markSafe(r[rr.x+1][rr.y]);

if(r[rr.x+1][rr.y+1].state==0)

markSafe(r[rr.x+1][rr.y+1]);

}

static void makeDrains(Region rr,int h)

{

//this function sets the state of an element as 1 and makes it a drain

//a region can become a part of a drain if and only if it is adjacent to a drain region and has a height greater than that region

//then the region is marked as done

//flooded regions end when a region which is a part of a drain becomes the lowest region adjacent to the flooded region

boolean isDrain=false;//initially not a drain

if(rr.x==0 || rr.y==0 || rr.x==9 || rr.y==9)

{

//coasts are always drains

rr.isDone=true;

rr.isDrain=true;

rr.state=1;

isDrain=true;

}

else

if(rr.h>=h)

{

//checking a region adjacent to a drain which is calling the function

rr.isDrain=true;

rr.isDone=true;

rr.state=1;

isDrain=true;

}

if(isDrain)

{

//checking and calling the function recursively, marking all drains

if(rr.x-1>=0 && rr.y-1>=0 ? !r[rr.x-1][rr.y-1].isDone : false)

makeDrains(r[rr.x-1][rr.y-1],rr.h);

if(rr.y-1>=0 ? !r[rr.x][rr.y-1].isDone : false)

makeDrains(r[rr.x][rr.y-1],rr.h);

if(rr.x+1<=9 && rr.y-1>=0 ? !r[rr.x+1][rr.y-1].isDone : false)

makeDrains(r[rr.x+1][rr.y-1],rr.h);

if(rr.x-1>=0 ? !r[rr.x-1][rr.y].isDone : false)

makeDrains(r[rr.x-1][rr.y],rr.h);

if(rr.x+1<=9 ? !r[rr.x+1][rr.y].isDone : false)

makeDrains(r[rr.x+1][rr.y],rr.h);

if(rr.x-1>=0 && rr.y+1<=9 ? !r[rr.x-1][rr.y+1].isDone : false)

makeDrains(r[rr.x-1][rr.y+1],rr.h);

if(rr.y+1<=9 ? !r[rr.x][rr.y+1].isDone : false)

makeDrains(r[rr.x][rr.y+1],rr.h);

if(rr.x+1<=9 && rr.y+1<=9 ? !r[rr.x+1][rr.y+1].isDone : false)

makeDrains(r[rr.x+1][rr.y+1],rr.h);

}

}

static void display()

{

//for printing the states of all the regions

for(int i=0;i<10;i++)

{

for(int j=0;j<10;j++)

if(j!=0)

System.out.printf("%3d",r[i][j].state);

else

System.out.print("1");

System.out.println();

}

}

}