#include<iostream>

#include<string>

#include<iterator>

using namespace std;

const int MAXSIZE = 10000;//最大的栈空间为1万

template<class ElemType>

class SqStack {

private:

ElemType\* top;//栈顶指针，在栈空的时候和栈底指针一起指向栈的第一个位置，栈为非空时指向栈顶元素的上面的一个位置

ElemType\* base;//栈底指针，始终指向栈的第一个位置，不管栈中有多少个元素

int maxSize;//用于标记栈中最多可以存放的元素数，同时也是判断栈满不满的标志之一，因为这是顺序栈，所以

//在使用它之前要先申请一块空间

public:

//构造函数:

SqStack() {

maxSize = MAXSIZE;

base = new ElemType[MAXSIZE];

top = base;

}

SqStack(int n) {

base = new ElemType[n];

top = base;

maxSize = n;

}

//析构函数:

~SqStack() {

StackDestory();

//cout << "调用了析构函数" << endl;

}

//拷贝构造函数:

SqStack(const SqStack& SqS) {

int L = SqS.getLength();

maxSize = SqS.max\_size();

base = new ElemType[maxSize];

top = base;

ElemType\* bs = SqS.getBase();

int i;

for (i = 0; i < L; ++i) {

\*(base + i) = \*(bs + i);

}

top = base + i;

}

//拷贝赋值运算符:

SqStack<ElemType>& operator=(const SqStack<ElemType>& SqS);

//销毁栈:

void StackDestory() {

top = base;

delete[] base;

}

//得到栈底指针:

ElemType\* getBase() const { return base; }

//得到栈顶指针:

ElemType\* getTop\_pointer()const { return top; }

//返回目前栈的元素个数:

int getLength() const;

//返回栈的最大空间:

int max\_size()const { return maxSize; }

//判断栈是否为空:

bool isEmpty() const;

//判断栈是否为满栈:

bool isFull()const;

//用e返回栈顶元素:

bool getTop(ElemType& e) const;

//入栈:

bool push(ElemType& e);

//出栈:

bool pop(ElemType& e);

//用e返回栈指定位置函数:

bool getElem(int pos, ElemType& e)const;

//遍历栈:

void traverse()const;

// 栈空间加倍:

void doubleSpace();

};

//拷贝赋值运算符:

template<class ElemType>

SqStack<ElemType>& SqStack<ElemType>::operator=(const SqStack<ElemType>& SqS) {

const int L = SqS.getLength();

maxSize = SqS.max\_size();

base = new ElemType[maxSize];

top = base;

ElemType\* bs = SqS.getBase();

int i;

for (i = 0; i < L; ++i) {

\*(base + i) = \*(bs + i);

}

top = base + i;

return \*this;

}

//返回目前栈的元素个数:

template<class ElemType>

int SqStack<ElemType>::getLength() const {

return top - base;

}

//判断栈是否为空:

template<class ElemType>

bool SqStack<ElemType>::isEmpty() const {

if (base == top) return true;

else return false;

}

//判断栈是否为满栈:

template<class ElemType>

bool SqStack<ElemType>::isFull()const {

if (top - base == maxSize) { return true; }

else return false;

}

//用e返回栈顶元素:

template<class ElemType>

bool SqStack<ElemType>::getTop(ElemType& e) const {

if (base == top) { return false; }

else {

ElemType\* t = top;

e = \*(--t);

return true;

}

}

//入栈:

template<class ElemType>

bool SqStack<ElemType>::push(ElemType& e) {

if (top - base == maxSize) { return false; }

else {

\*top++ = e;

return true;

}

}

//出栈:

template<class ElemType>

bool SqStack<ElemType>::pop(ElemType& e) {

if (base == top) { return false; }

else {

e = \*(--top);

if (top - base == 0) {//为了使得当栈中的元素只有一个往出弹后，其实没有真正的弹出，只是指针移位，

//不过底部的指针是不会移位的，所以也就相当于这个值没有被弹出，这里选择采取将空间释放再

//重新申请空间的做法来实现这种特殊情况下的真正的弹出

delete[]base;

base = new ElemType[maxSize];

top = base;

//cout << "栈已被弹空" << endl;

}

return true;

}

}

//用e返回栈指定位置函数:pos从1开始

template<class ElemType>

bool SqStack<ElemType>::getElem(int pos, ElemType& e)const {

return \*(base + pos-1);

}//未经测试，慎用！！！

template<class ElemType>

void SqStack<ElemType>::traverse()const {//遍历顺序栈

if (top == base) { return; }

else {

const int L = this->getLength();

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

cout << \*(base + i) << ' ';

}

}

}

// 栈空间加倍:

template<class ElemType>

void SqStack<ElemType>::doubleSpace() {

ElemType\* tmp\_base = base;

base = new ElemType[maxSize \* 2];

int L = this->getLength();

int i;

for (i = 0; i < L; ++i) {

\*(base + i) = \*(tmp\_base + i);

}

top = base + i;

maxSize \*= 2;

}

bool make2DArray(int row, int col, int\*\*& maze){ //二位数组空间申请

maze = new int\* [row];

if (!maze) return false;//表明没有成功申请

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

maze[i] = new int[col];

if (!maze[i]) return false;//表明成功申请

}

return true;

}//注意这里创建的动态数组程序结束以后是要释放的，一定要记住释放它

int direction[4][2] = { -1,0,0,1,1,0,0,-1 };//用于四个方向的遍历

struct node {

int x, y;//x代表所在的行数，y代表所在的列数

int dir=-1;//记录已经试探过的方向的最大坐标,初始化为-1，表明其任何方向都没有试探过

};

//判断起点和终点是否合法的函数

bool judge\_start\_end(const node& S, const node& E,int\*\* &maze) {

if (maze[S.x][S.y] == 1 || maze[E.x][E.y] == 1) {

return false;//起点和终点不合法

}

else return true;

}

bool InputMaze(int row, int col, int\*\*& maze) { //迷宫矩阵输入

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

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

cin >> maze[i][j];

}

}

return true;

}

bool OutputMaze(const int& row, const int& col, int\*\* &maze) {//迷宫矩阵输出函数

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

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

cout << maze[i][j];

if (j < col - 1)cout << ' ';

}

cout << endl;

}

return true;

}

//判断当前的点有没有出界:

bool judge\_point(node& n,const int& row,const int& col) {

if (n.x < 0 || n.y < 0 || n.x >= row || n.y >= col) {

return false;

}

else return true;

}

//最终的找路径的函数://已经走过的点处的值被置为2

template<class ElemType>

void maze\_path(SqStack<ElemType>& S, int row, int col, node enter, node outer, int\*\* maze) {

//S这个栈用于存放已经选择过的路径坐标

if (S.isEmpty()) {

S.push(enter);//当S为空时，先把起点的坐标放进去

maze[enter.x][enter.y] = 2;//并把起点出的maze数组的值置为2，表示已经走过起点

}

node ing,next;

while (!S.isEmpty()) {

S.pop(ing);

while (ing.dir < 3) {

ing.dir += 1;

next.x = ing.x + direction[ing.dir][0];

next.y = ing.y + direction[ing.dir][1];

if (judge\_point(next, row, col)) {

if (next.x == outer.x&&next.y==outer.y) {

S.push(ing);

S.push(next);

int L = S.getLength();

node e;

int i = L - 1;

node \*r=new node[L];

while (!S.isEmpty()) {

S.pop(e);

r[i] = e;

--i;

}

for (i = 0; i < L; ++i) {

cout << '(' << r[i].x << ',' << r[i].y << ')';

if (i < L - 1)cout << "->";

if ((i + 1) % 4 == 0 && i != L - 1) cout << endl;

}

delete[] r;

return;

}

if (maze[next.x][next.y] == 0) {

S.push(ing);

S.push(next);

maze[next.x][next.y] = 2;

break;

}

else continue;

}

}

}

cout << "No Path";

}

//注意：本程序是要求输出一条简单的路径

int main() {

int\*\* maze;//迷宫二维数组

int row, col;//行数、列数

node Start, End;

cin >> row >> col;

cin >> Start.x >> Start.y;

cin >> End.x >> End.y;

make2DArray(row, col, maze);

InputMaze(row, col, maze);

//OutputMaze(row, col, maze);

if (judge\_point(Start,row,col)&&judge\_point(End,row,col)&&judge\_start\_end(Start, End, maze) == false)

cout << "No Path";

else {//起点和终点已经合法，才有了找路的必要

SqStack<node> S;

maze\_path(S, row, col, Start,End, maze);

}

//程序结束以后销毁二维数组，暂时表明没有问题

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

delete [] maze[i];

delete[] maze;

return 0;

}