# 四川师范大学 实验报告

学期： 2024 至 2025 第一学期 实验成绩：

课程名字：《程序设计基础——数据结构(C语言版)》 专业： 信息与计算科学

班级： 2023 级 9 班 实验编号： 03

实验项目： 实验三 指导老师： 冯山

姓名： 刘智恒 学号： 2023060522

**一、实验题目**

堆栈和队列及其应用

**二、实验目的及要求**

1.掌握堆栈和队列的ADT及其存储结构和基本操作算法实现特点。

2.运用堆栈和队列的逻辑结构、存储结构和基本操作，进行问题求解算法设计和实现。

**三、实验内容：(类C算法的程序实现，任选其二)**

1.设计并实现堆栈和队列的各种操作算法。(必做)

2.设计并实现算术表达式计算算法。

3.设计并实现迷宫问题的搜索求解算法。

**四、实验准备**

1.计算机设备;

2.程序调试环境的准备，本实验采用**Microsoft Visual Studio**环境;

3.实验内容的算法分析与代码设计与分析准备；

4.实验源程序**Exp\_3(1)，Exp\_3(3)**准备。

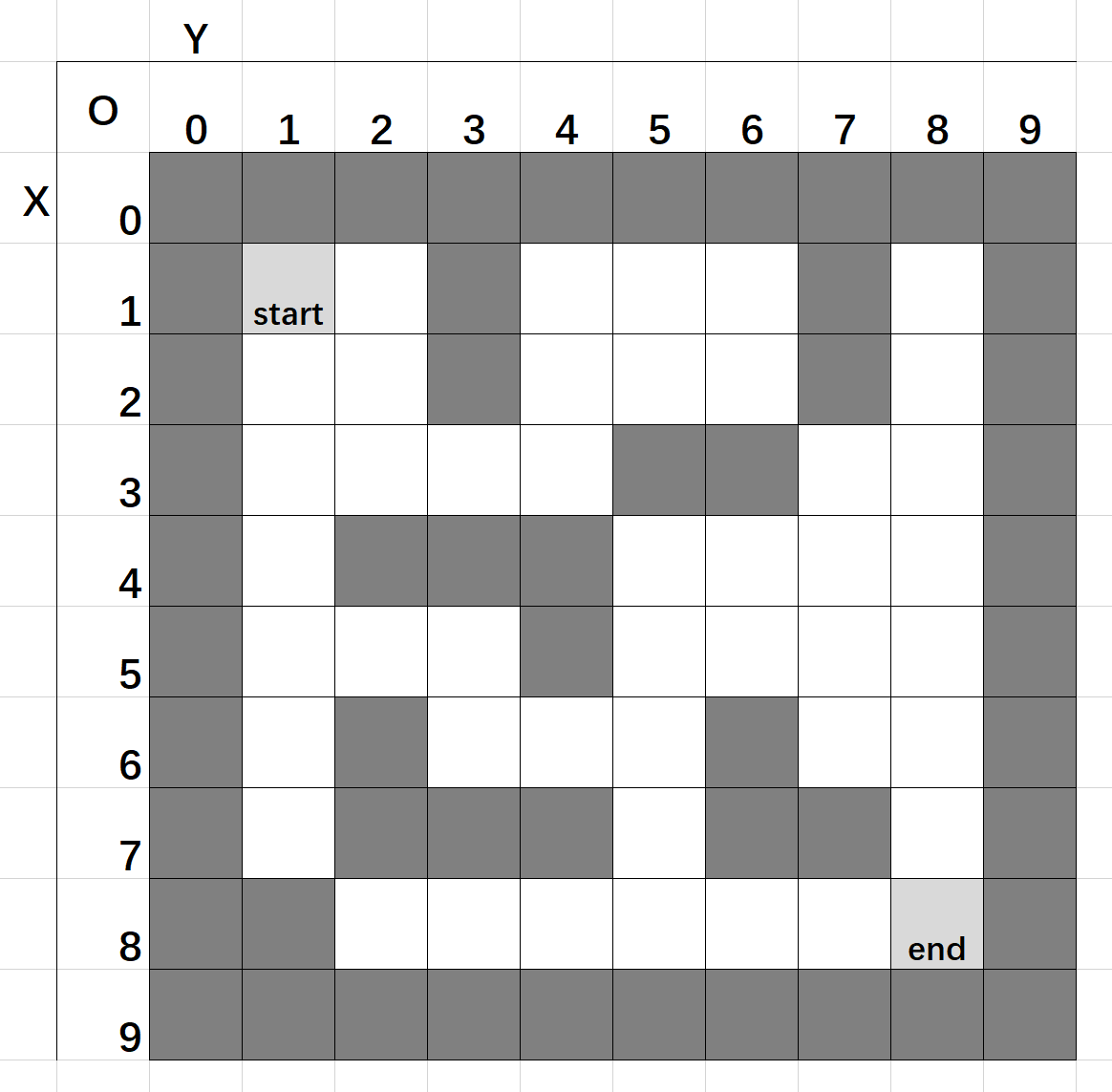
5.实验测试用例：

实验内容1：

（1）**入堆栈数据：1 2 3 4 5 6**

（2）**入队列数据：1 2 3 4 5**

实验内容3：测试所用的迷宫地图如下：

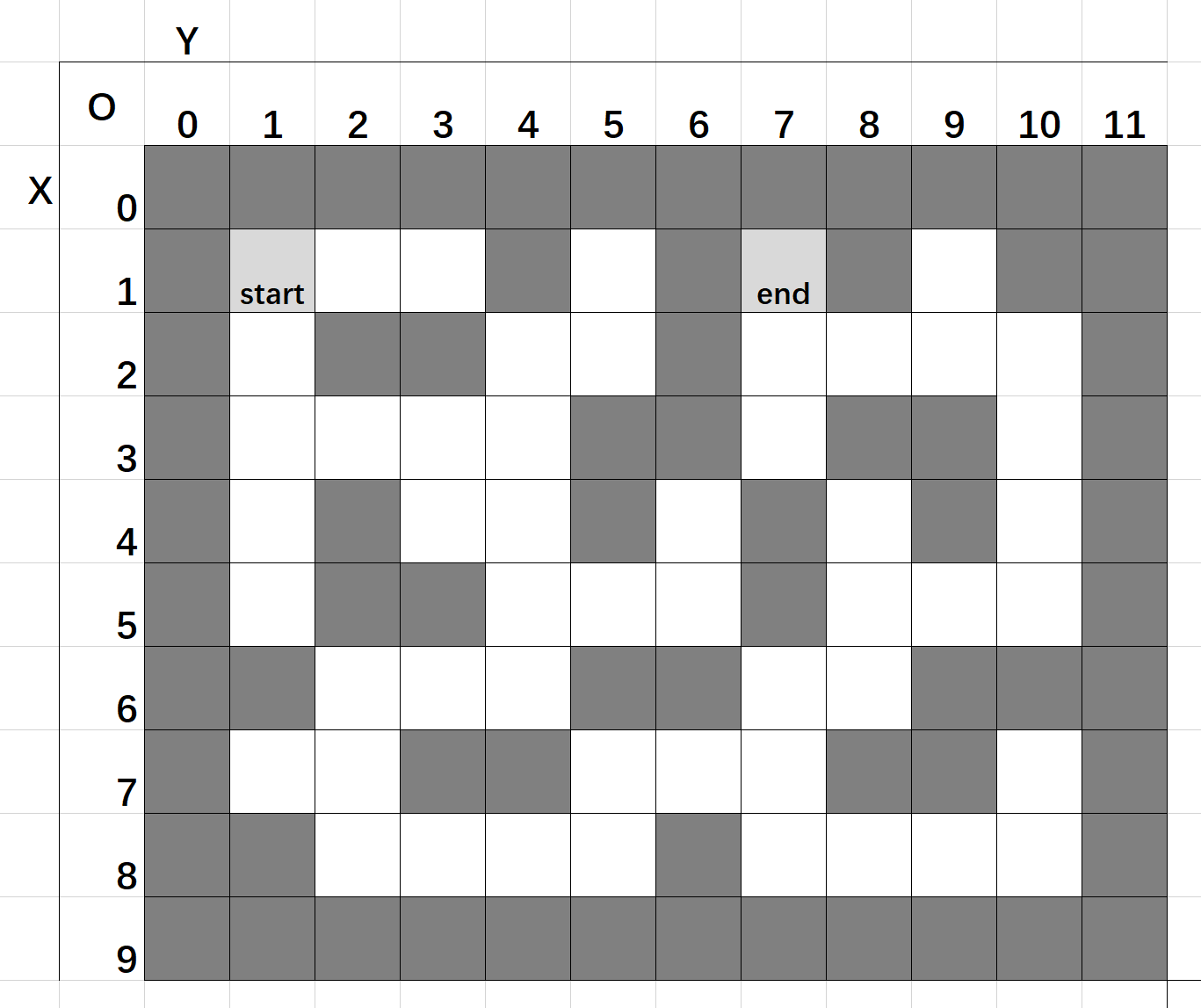


**图4.1迷宫地图1**

（1）**迷宫大小设置：M = N = 10;**

**起点（start）设置：（1，1）;**

**终点（end）设置：（8，8）.**



**图4.2迷宫地图2**

（2）**迷宫大小设置：M = 10，N = 12;**

**起点（start）设置：（1，1）;**

**终点（end）设置：（1，7）.**

**五、实验过程**

**（一）问题分析**

**1.实验内容1：**

（1）基本分析：

[1] 堆栈结构符合“**后进先出**”原则，即**后入堆栈的元素最先出堆栈**。故可以设定**base**指针和**top**指针分别指向**栈底元素**和**栈顶元素**，同时设定其最大容量。当堆栈中没有元素时，设定**base**指针和**top**指针同时指向栈底；随着元素的不断入栈，**top**指针不断**向上移动**直到到达栈顶，而**base**指针则**保持不动**，用于动态分配存储空间。

[2] 队列结构符合“**先进先出**”原则，即**先入队列的元素最先出队列**。故可设定**front**指针和**rear**指针分别指向队列**头部元素**和**尾部元素**。当队列中没有元素时，设定**front**指针和**rear**指针**指向同一处**，此时约定**front = rear = 0**；随着元素不断入队列，**front**指针和**rear**指针进行相应移动。当队列空间被占满时，凭借**front = rear**无法进行判断，因为此时也**可能为队空**状态。故此处的处理方法为**采用循环队列结构**：即**少用一个元素空间，约定以front指针在rear指针的下一位置上作为队列的占满状态**。

（2）本问题中，以**顺序表结构**的堆栈和队列为准，即以**数组**的方式对元素进行存储。链式结构的堆栈和队列操作类似，在此不作赘述。需要注意的是，本问题中的队列实际上采用了**循环队列**的结构。堆栈和队列的数据结构类型定义如下：

**typedef struct SqStack{ //堆栈**

**SElemType \*base; //栈底指针**

**SElemType \*top; //栈顶指针**

**int stacksize; //栈可用最大容量**

**}SqStack;**

**typedef struct SqQueue{ //队列**

**ElemType \*base; //初始化动态分配内存空间**

**int front; //头指针**

**int rear; //尾指针**

**}SqQueue;**

几项基本操作如下：

堆栈：[1] **InitStack** —— 堆栈的初始化；

[2] **StackEmpty** —— 判断堆栈是否为空；

[3] **StackLength** —— 求堆栈长度；

[4] **ClearStack** —— 清空堆栈；

[5] **DestroyStack** —— 销毁堆栈；

[6] **Push** —— 入堆栈操作；

[7] **Pop** —— 出堆栈操作；

[8] **PrintStack** —— 打印堆栈里的数据。

队列：[1] **InitQueue** —— 队列的初始化；

[2] **DestroyQueue** —— 销毁队列；

[3] **QueueLength** —— 求队列长度；

[4] **GetHead** —— 取出队头元素；

[4] **EnQueue** —— 入队列操作；

[5] **DeQueue** —— 出队列操作；

[6] **PrintQueue** —— 打印队列里的数据。

**2.实验内容3：**

（1）确定迷宫的存储结构：在多数情况下，迷宫地图均以**二维平面**方式展现。故在本问题中，考虑以**二维数组**的方式对迷宫地图信息进行存储。同时，确定以“**1”表示墙面**，以“**0”表示可通区块**，以“**2”表示已探索过的可通点**，以“**3”表示已探索过的不可通点**。迷宫的数据结构类型定义如下：

**typedef struct //迷宫的数据结构类型**

**{**

**int maze[M][N]; //M:迷宫的宽度，N:迷宫的长度**

**}MazeType;**

（2）确定每个位置的数据结构：在本问题中，同样的，确定以**1表示墙面**，以**0表示可通区块**，以**2表示已探索过的可通点**，以**3表示已探索过的不可通点**。迷宫中的每个点都有如下性质：

[1] 该点**是否可通**，即该位置是可通区块（**0**）还是墙面（**1）**；

[2] 该点**是否已经探索过**，即该位置是可通点（**2**）还是不可通点（**3**）；

[3] 该点**是否已经探索完4个方向**，若是，则标记该点为不可通点（**3**），反之，标记该点为可通点（**2**）。

除此之外，每个可通区块的**位置坐标**应确定。本问题迷宫中每个点的坐标结构类型定义如下：

**typedef struct //可通区块坐标结构类型定义**

**{**

**int x; //可通区块横坐标**

**int y; //可通区块纵坐标**

**}PosType;**

（3）确定迷宫探索过程的数据结构：对于迷宫的探索过程，每次探索的位置均有**4个探索方向（本问题只考虑东南西北四个探索方向）**。若下一位置的某个方向可继续探索，则目标移动到下一位置上，反之则换到下一个方向；若下一位置的**4个方向均不可探索**，**则回退到上一位置**。以上过程遵循一种“**后进先出**”的性质，符合**入栈出栈**的过程。故本问题采用**顺序栈**数据结构，顺序栈结构类型定义如下：

**typedef struct //顺序栈结构类型定义**

**{**

**SElemType \*base;**

**SElemType \*top;**

**int stacksize;**

**}SqStack;**

（4）根据以上分析，栈中每个元素的数据类型**（SElemType）**即可确定：

**typedef struct //栈元素的类型**

**{**

**int step; //可通区块在路径上的序号**

**PosType pos; //可通区块在迷宫中的坐标位置**

**int dir; //通道块走向下一块的“方向序号”**

**}SElemType;**

**（二）算法描述**

**1.实验内容1：**

**[1]以下为堆栈的算法描述.**

（1）**Status InitStack(SqStack \*S);**

**//**本算法用于初始化一个顺序栈：为栈分配空间内存并检查是否分配成功，若失败则退出程序；将栈顶指针置于栈底，栈的大小设置为初始预定义值。

（2）**Status StackEmpty(SqStack S);**

**//**本算法用于判断栈是否为空：检查栈顶指针是否等于栈底指针，如果相等，则返回真（栈空）；否则返回假（栈不为空）。

（3）**int StackLength(SqStack S);**

**//**本算法用于获取顺序栈的长度：计算栈中元素的个数，即top - base。最终返回栈的长度

（4）**Status ClearStack(SqStack \*S);**

**//**本算法用于清空顺序栈：如果base不为空，将栈顶指针top重置为栈底指针base，使栈为空。

（5）**Status DestroyStack(SqStack \*S);**

**//**本算法用于销毁顺序栈：如果base不为空，释放之前分配的内存。将stacksize设置为0，并将base和top都设置为NULL。

（6）**Status Push(SqStack \*S, SElemType e);**

**//**本算法用于将元素e入栈：首先检查栈S是否满（即栈顶指针与栈底指针的差是否等于栈大小）。如果栈满，则判断入栈失败；反之将元素e赋值给当前栈顶指针所指向的位置，然后将栈顶指针上移。

（7）**Status Pop(SqStack \*S, SElemType \*e);**

**//**本算法利用元素e将栈顶元素出栈：首先检查栈S是否为空，如果栈空，则判断出栈失败；反之将栈顶指针下移，然后将当前栈顶指针所指元素赋值给e。

（8）**void PrintStack(SqStack S);**

**//**本算法用于打印栈中的所有元素：使用指针遍历从栈底base到栈顶top的所有元素。打印每个元素，最后输出"End"表示结束。

**[2]以下为队列的算法描述.**

（1）**Status InitQueue(SqQueue \*Q)**;

//本算法用于初始化循环队列：为队列的元素分配一块内存，如果内存分配失败，程序退出；否则将队列的头指针front和尾指针rear初始化为0，表示队列为空。

（2）**Status** **DestroyQueue(SqQueue \*Q);**

**//**本算法用于销毁循环队列：释放其占用的内存，同时将base设置为 NULL，避免出现野指针。

（3）**int** **QueueLength(SqQueue Q);**

**//**本算法用于计算循环队列长度：通过(Q.rear - Q.front + MAXQSIZE) % MAXQSIZE计算当前队列中的元素个数。

（4）**Status** **EnQueue(SqQueue \*Q, ElemType e)**;

//本算法用于将元素e入队：首先判断队列是否已满，如果队列已满，返回ERROR，否则将元素e添加到队列的尾部位置，同时更新尾指针rear。

（5）**Status** **DeQueue(SqQueue \*Q, ElemType e);**

**//**本算法用于将队头元素出队：首先判断队列是否为空，如果为空，返回ERROR，否则保存队头元素到变量e，同时更新头指针front。

（6）**Status** **GetHead(SqQueue Q);**

**//**本算法用于获取队头元素的值：判断队列是否为空，如果队列不为空，返回当前队头元素。

（7）**void** **PrintQueue(SqQueue Q);**

**//**本算法用于打印队列中的所有元素：从front开始，循环遍历到rear，打印每个元素，最后输出"End"表示结束。

**2.实验内容3：**

（1）**Status InitStack(SqStack \*S);**

**//**本算法用于初始化一个顺序栈：为栈分配空间内存并检查是否分配成功，若失败则退出程序；将栈顶指针置于栈底，栈的大小设置为初始预定义值。

（2）**Status StackEmpty(SqStack S);**

**//**本算法用于判断栈是否为空：检查栈顶指针是否等于栈底指针，如果相等，则返回真（栈空）；否则返回假（栈不为空）。

（3）**Status Push(SqStack \*S, SElemType e);**

**//**本算法用于将元素e入栈：首先检查栈S是否满（即栈顶指针与栈底指针的差是否等于栈大小）。如果栈满，则判断入栈失败；反之将元素e赋值给当前栈顶指针所指向的位置，然后将栈顶指针上移。

（4）**Status Pop(SqStack \*S, SElemType \*e);**

**//**本算法利用元素e将栈顶元素出栈：首先检查栈S是否为空，如果栈空，则判断出栈失败；反之将栈顶指针下移，然后将当前栈顶指针所指元素赋值给e。

（5）**int Pass(MazeType Maze, PosType pos);**

**//**本算法用于判断当前位置是否可通过：首先检查当前位置坐标是否在迷宫Maze边界内，然后判断当前位置pos的初始标记值（0或1），最终返回对应的真假值。

（6）**void FootPrint(MazeType \*Maze, PosType pos);**

**//**本算法用于将当前探索的位置进行可通过标记：将迷宫中当前位置坐标pos的值标记为2，表示已走过且可通过。

（7）**void MarkPrint(MazeType \*Maze, PosType pos);**

**//**本算法用于将当前探索的位置进行不可通过标记：将迷宫Maze中当前位置坐标pos的值标记为3，表示已走过且不可通过。

（8）**PosType NextPos(PosType pos, int dir);**

**//**本算法用于获取下一个方向的位置坐标：根据传入的方向dir更新当前位置坐标pos，以顺时针方向为准，最终返回新的位置坐标。

（9）**void PrintPath(SqStack S);**

**//**本算法用于输出探索路径的坐标序列：通过遍历栈，从栈底指针到栈顶指针依次输出每个路径位置的坐标。

（10）**Status MazePath(MazeType Maze, PosType start, PosType end);**

**//**本算法用于在迷宫中探索查找从起点到终点的可通路径：其中，Maze表示已知的迷宫，start表示迷宫Maze的起点坐标，end表示迷宫Maze的终点坐标。

**//**首先初始化栈和当前位置坐标，然后进入循环过程：如果当前位置可通过，则留下足迹，并将当前位置信息入栈，同时检查是否到达终点，若是则输出路径并返回真；如果当前位置不可通过，则退回到上一个栈顶元素并更新当前位置信息与状态。当栈为空时，退出循环并返回假表示未找到路径。

**（三）程序代码：**

**1.实验内容1：**

根据以上对**实验内容1**的算法描述，求解本问题的程序代码如下：



**2.实验内容3：**

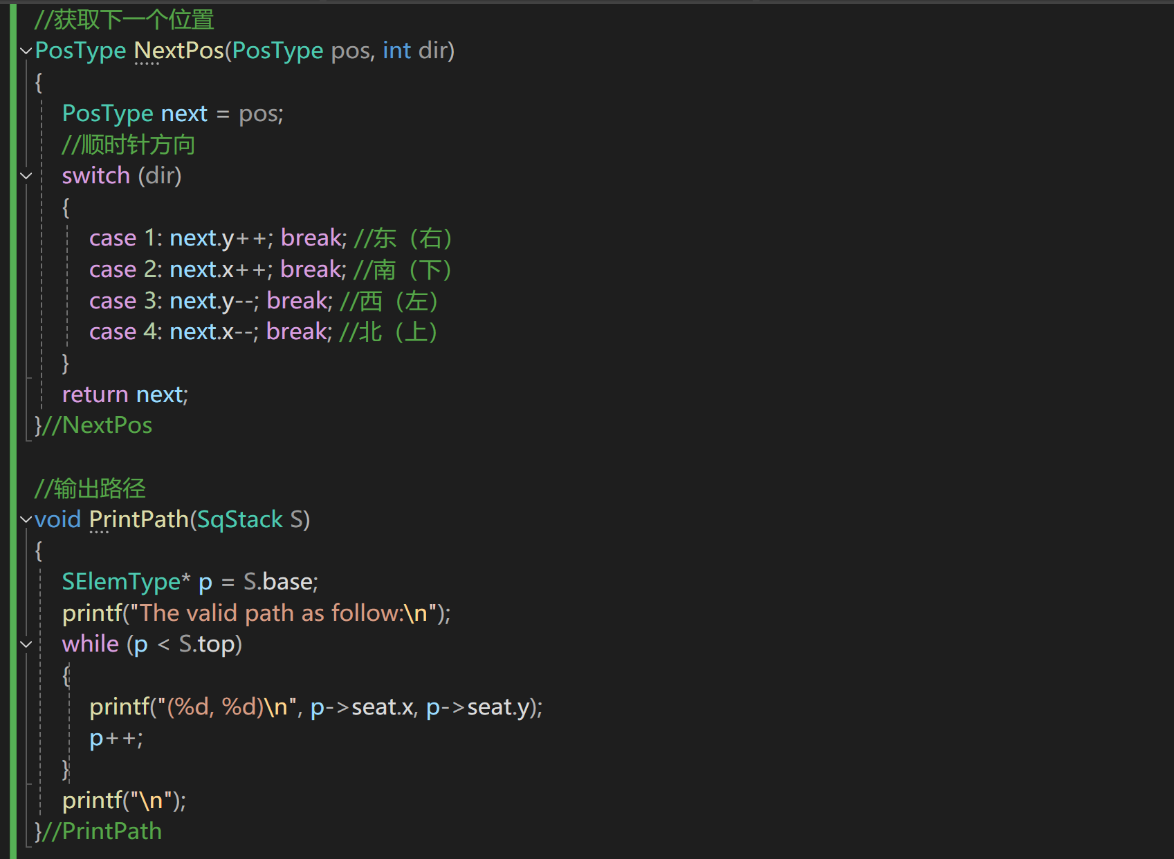
根据以上对**实验内容3**的算法描述，求解本问题的程序代码如下：



**注：此处只展示主要算法的程序代码图。**

****

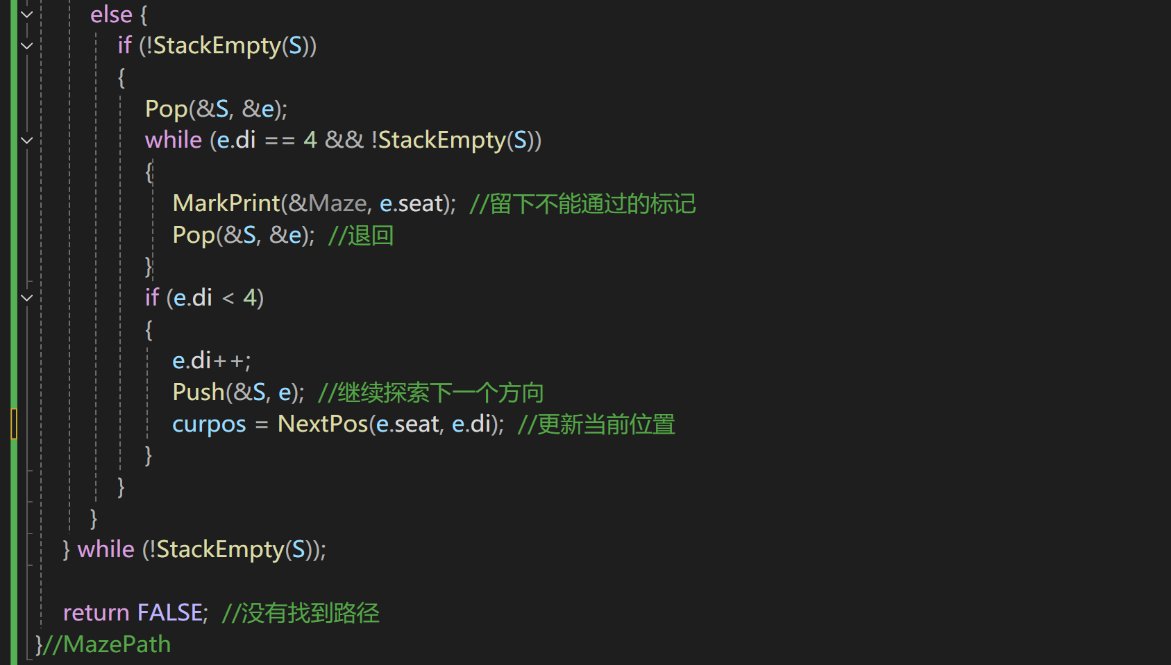
**图5.1**

****

**图5.2**

****

**图5.3迷宫路径查找（1）**

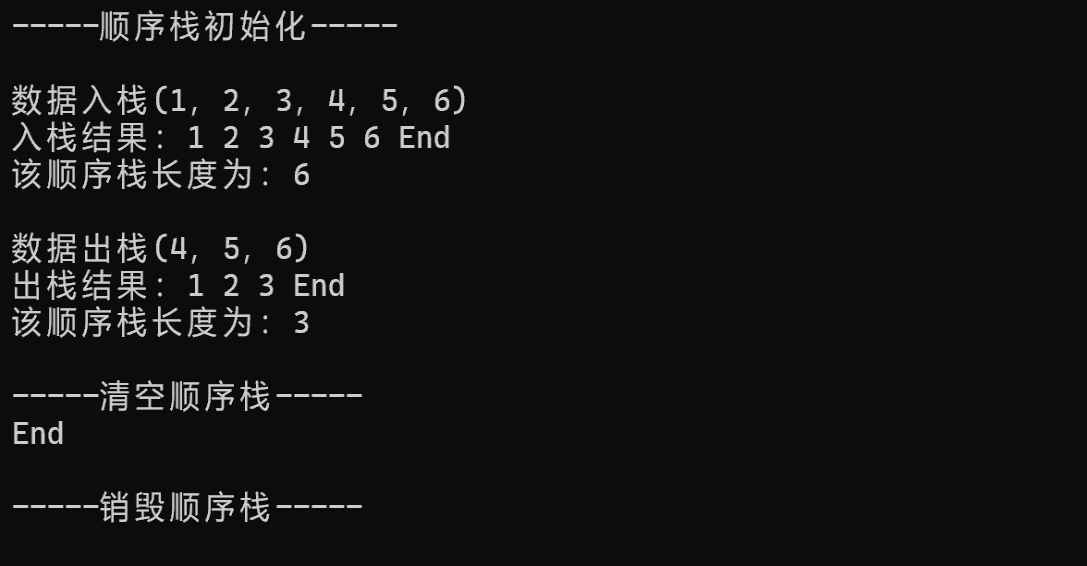
****

**图5.4迷宫路径查找（2）**

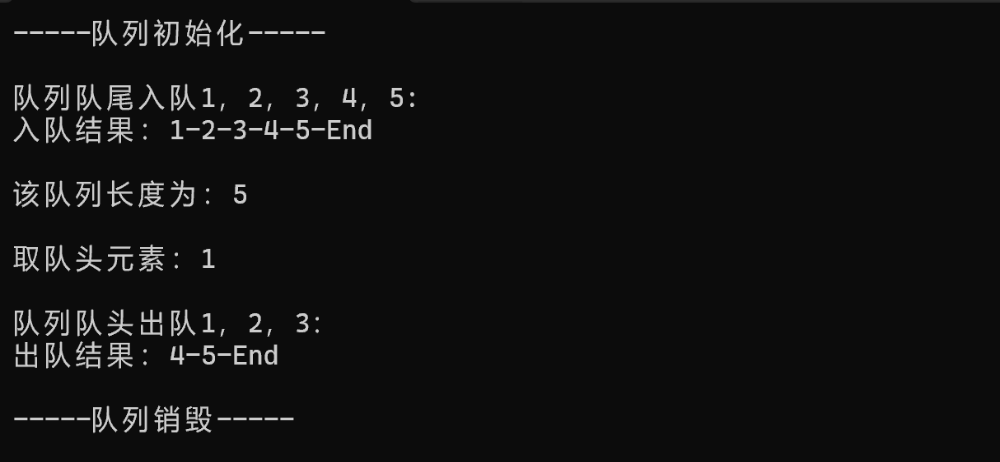
**六、结果分析**

**（一）结果呈现：**

1.根据以上**实验内容1**程序代码运行后进行多次测试，以下是根据**实验内容1的测试用例**进行测试后的结果示意图：

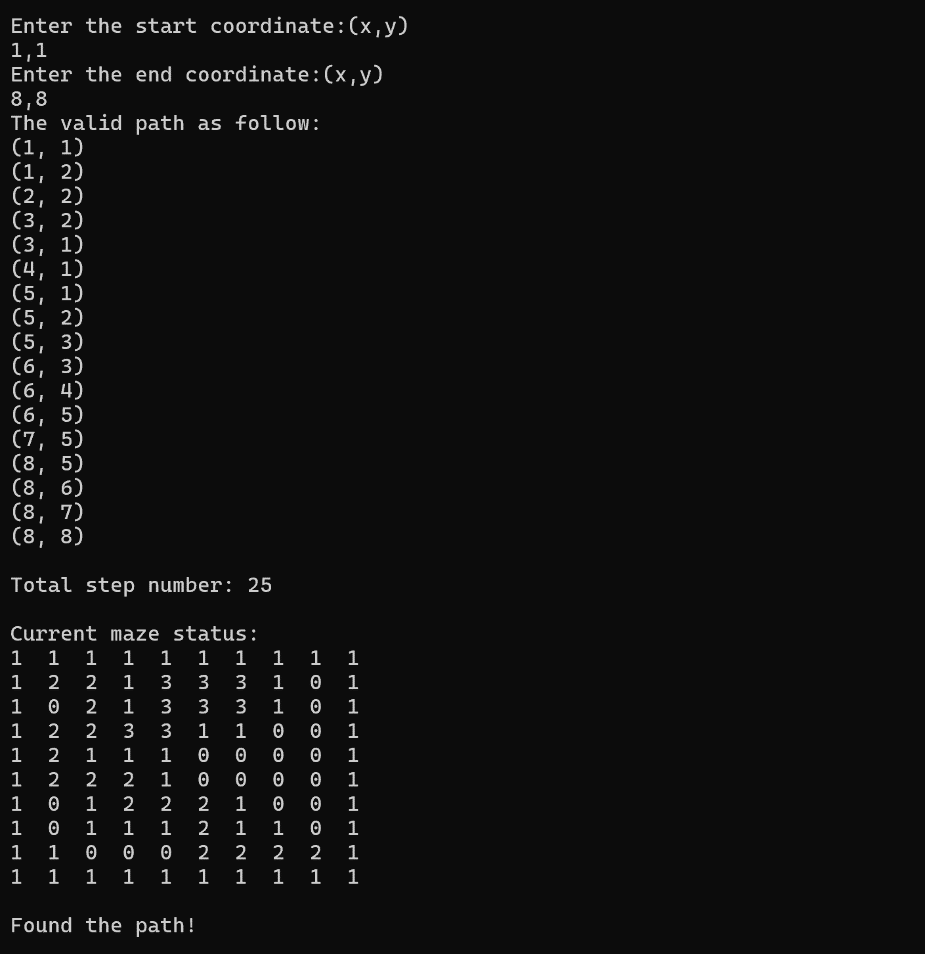
****

**图6.1堆栈操作结果**

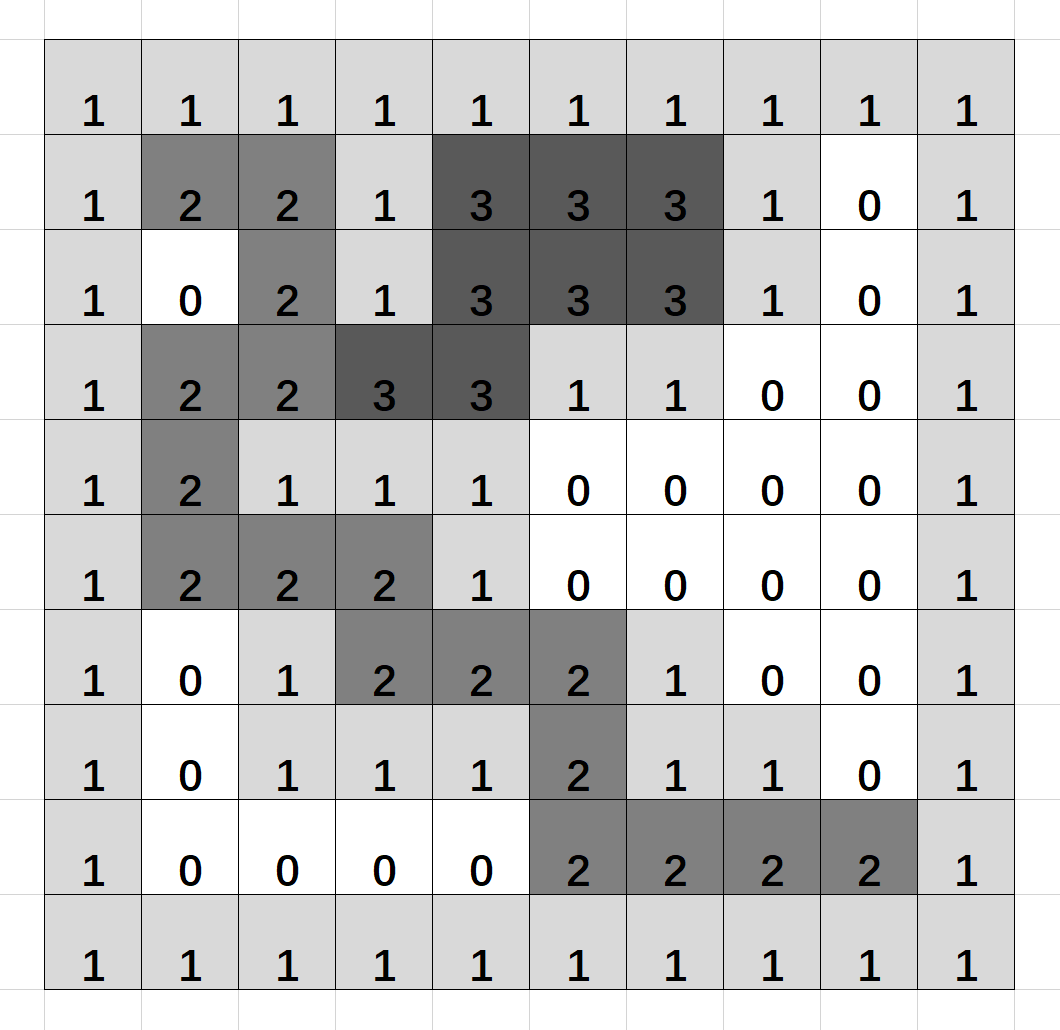
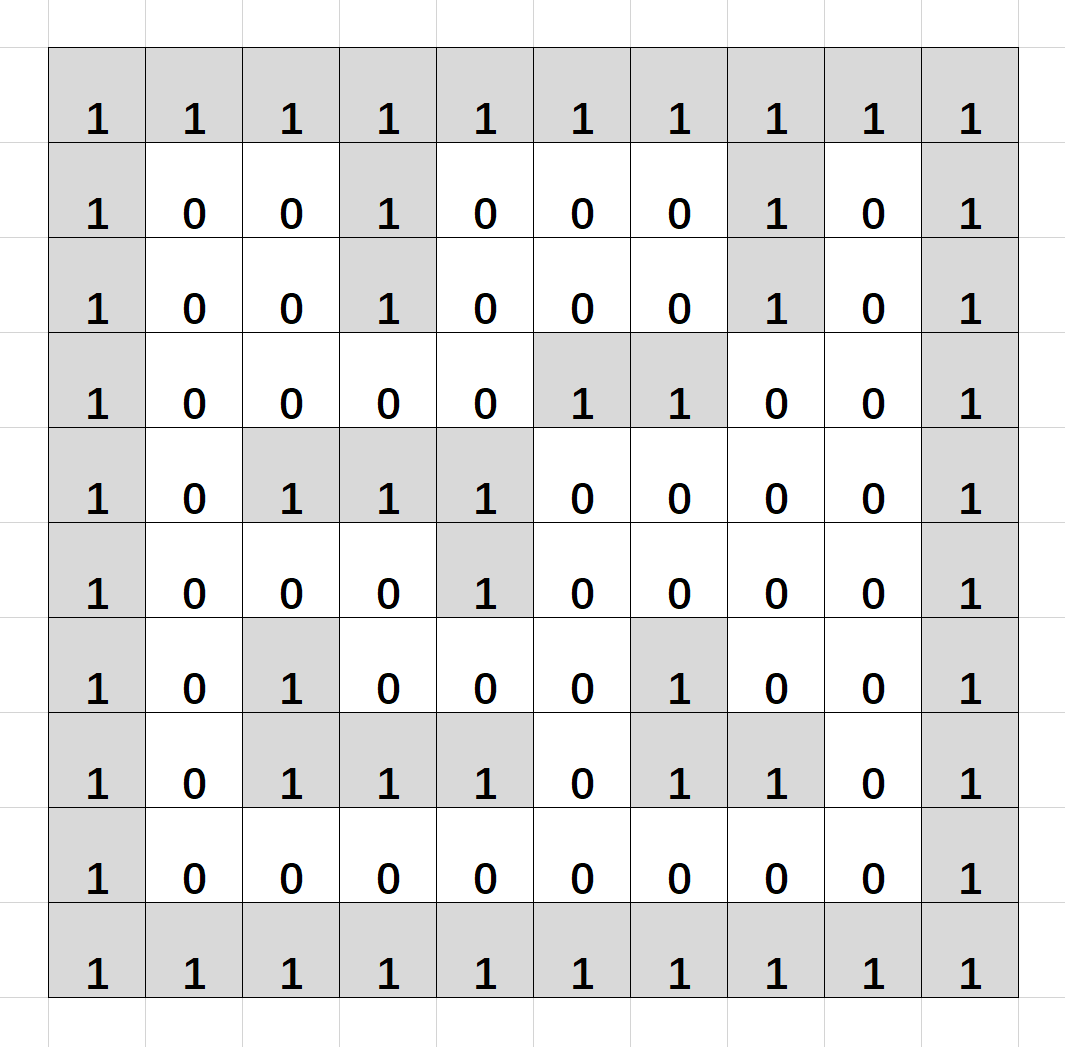
****

**图6.1队列操作结果**

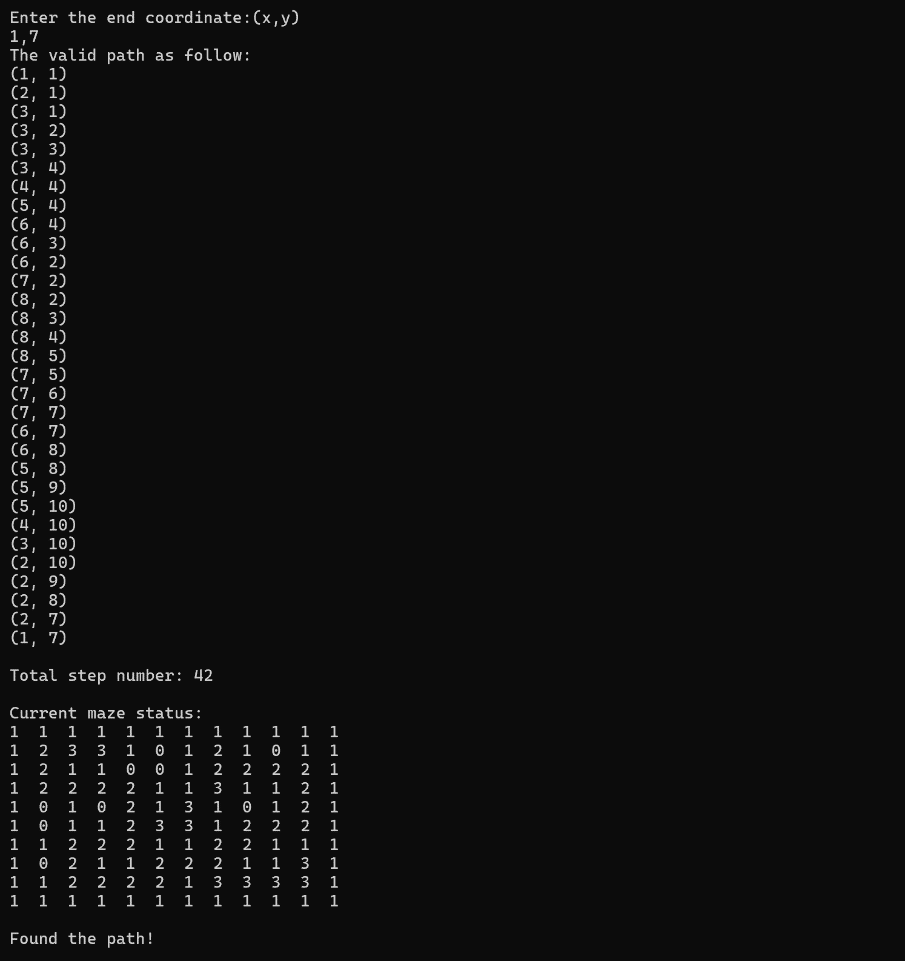
1.根据以上**实验内容3**程序代码运行后进行多次测试，以下是根据**实验内容3的测试用例**进行测试后的结果示意图：

****

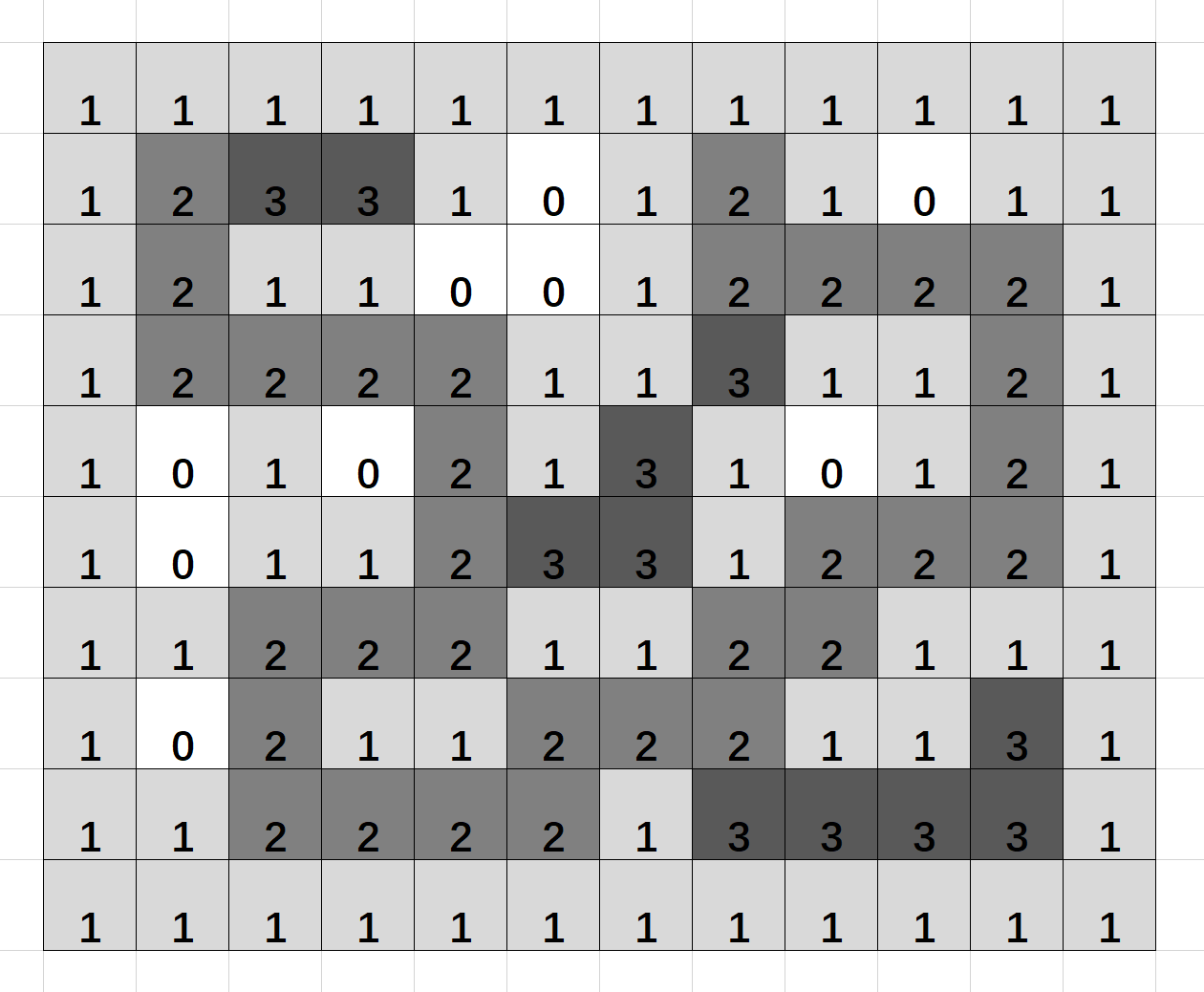
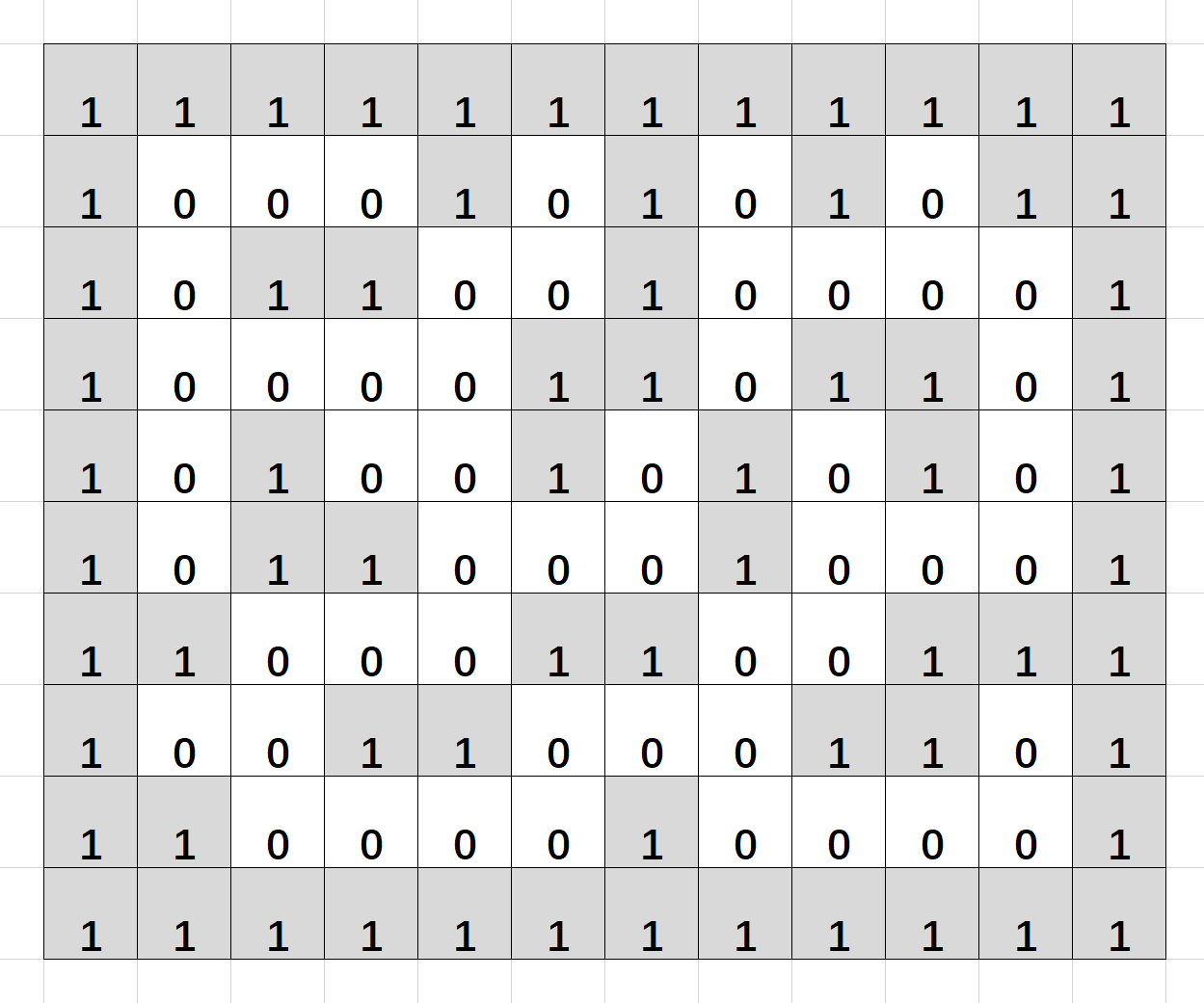
**图6.1测试用例1相关结果**

****

**图6.2迷宫地图1初始状态 图6.3迷宫地图1标记状态**

****

**图6.4测试用例2结果**

****

**图6.5迷宫地图2初始状态 图6.6迷宫地图2标记状态**

**（二）结果分析：**

**1.时间复杂度分析：**

**（1）实验内容1：**

**[堆栈]**

**【InitStack】—— 初始化栈**

其中分配内存的时间复杂度为**O(n)**，其中**n**是栈的最大容量,在这里是固定为**MAXSIZE**。但实际的时间复杂度为**O(1)**，因为这个操作不依赖于栈的当前状态

【**PrintStack**】**—— 打印栈的内容**

需要遍历栈中的所有元素，时间复杂度为**O(n)**，其中**n**是当前栈中的元素数量。

**【Other Functions】—— 其余函数**

其余操作均为常量级时间，故时间复杂度均为**O(1)**。

**综上所述：总时间复杂度为O(1)。**

**[队列]**

**【InitQueue】—— 初始化队列**

动态分配内存的操作时间复杂度为**O(n)**，这里**n**是队列最大容量**（MAXQSIZE）**。但实际的时间复杂度为**O(1)**，因为这个操作不依赖于队列当前状态。

【**PrintQueue**】**—— 打印队列的内容**

需要遍历队列中的所有元素，时间复杂度为**O(n)**，其中**n**是当前队列中的元素数量。

**【Other Functions】—— 其余函数**

其余操作均为常量级时间，故时间复杂度均为**O(1)**。

**综上所述：总时间复杂度为O(1)。**

**（2）实验内容3：**

**【MazePath】—— 寻找迷宫路径**

整个程序中，主要时间复杂度集中在该函数中。该段代码中使用了回溯方法来寻找迷宫路径。每次移动的位置有四个方向（东、南、西、北），但是在每个点上，只有一些特定的方向可以继续移动。

在最坏情况下，算法可能会**遍历整个迷宫的每个位置**，对于每个位置，都需要检查四个方向的可通行性。因此，最坏情况下的时间复杂度为**O(M\*N)**，其中**M**和**N**分别是迷宫的行数和列数。

在实际操作的循环遍历过程中，会标记已经走过的路径，从而避免重复访问。因此实际时间复杂度在很多情况下会**小于O(M\*N)**。

**综上所述：总时间复杂度为O(M\*N)，其中M和N分别是迷宫的行数和列数。**

**2.空间复杂度分析：**

**（1）实验内容1：**

**[堆栈]**

主要的内存消耗来自于栈的存储空间。初始化时分配的数组大小是**n**（其中**n=MAXSIZE**），即栈的最大容量。其他变量的空间消耗是常数级别**O(1)**，因为它们的存储需求不随输入规模变化。

**综上所述：整个顺序栈的空间复杂度为O(n)。**

**[队列]**

队列使用一个数组来存储数据，数组的大小为**MAXQSIZE**，因此空间复杂度为**O(n)**，这里**n**为队列的最大容量。

除了这个数组外，队列结构体中只包含固定的整型变量（如**front**和**rear**），这些的空间占用是**O(1)**。

**综上所述：整个顺序队列的空间复杂度为O(n)。**

**（2）实验内容3：**

**【堆栈存储】**

本问题中使用了堆栈来保存路径，以便回溯。最坏情况下，栈的深度可能达到**O(M\*N)**，因为在最坏的情况下，可能会将整个迷宫的路径点都保存到栈中。

此外，栈的空间使用是基于**SqStack**结构体，其中**base**的大小为**MAXSIZE**，但在这个特定问题中，**实际使用的空间取决于找到的路径的长短**。

【**迷宫状态**】

迷宫本身的存储是一个二维数组，所占用的空间是**O(M\*N)**，这是固定的，不受输入影响。

**综上所述：总空间复杂度为O(M\*N)，主要是由栈和迷宫数据结构所占用的空间决定的。**

**七、感悟体会**

1.**实验内容3**中利用堆栈的方式对迷宫问题进行了求解，同时也利用堆栈完成了回溯的过程，实现了对路径的求解，但无法求得最短路径。而实际上，这也是利用堆栈对**DFS(Deep Fast Search)——深度优先搜索算法**的一种实现。

2.**DFS**是一种用于遍历或搜索树或图的算法。它从一个起始节点开始，尽可能深入一个分支，直到到达目标节点或没有更多的节点可供访问，然后回溯，并继续在其他分支中搜索。问题三同样也可以使用**DFS**的**递归**方法对路径进行求解。