项目说明文档

数据结构课程设计

——勇闯迷宫游戏

同济大学

Tongji University

目 录

[1 分析 1](#_Toc60005030)

[1.1 背景分析 1](#_Toc60005031)

[1.2 功能分析 1](#_Toc60005032)

[2 设计 2](#_Toc60005033)

[2.1 数据结构设计 2](#_Toc60005034)

[2.2 类结构设计 2](#_Toc60005035)

[2.3 成员与操作设计 2](#_Toc60005036)

[2.4 系统设计 4](#_Toc60005037)

[3 实现 5](#_Toc60005038)

[3.1 插入功能的实现 5](#_Toc60005039)

[3.1.1 插入功能流程图 5](#_Toc60005040)

[3.1.2 插入功能核心代码 6](#_Toc60005041)

[3.2 去顶功能的实现 7](#_Toc60005042)

[3.2.1 去顶功能流程图 7](#_Toc60005043)

[3.2.2 去顶功能核心代码 7](#_Toc60005044)

[3.3 取顶功能的实现 8](#_Toc60005045)

[3.3.1 取顶功能流程图 8](#_Toc60005046)

[3.3.2 查找功能核心代码 8](#_Toc60005047)

[3.4 dfs的实现 9](#_Toc60005048)

[3.4.1 dfs思想： 9](#_Toc60005049)

[3.4.2 dfs核心代码 9](#_Toc60005050)

[3.5 总体系统的实现 11](#_Toc60005051)

[3.5.1 总体系统流程图 11](#_Toc60005052)

[3.6.2 总体系统核心代码 12](#_Toc60005053)

[3.6.3 总体系统截屏示例 13](#_Toc60005054)

[4 测试 14](#_Toc60005055)

[4.1 功能测试 14](#_Toc60005056)

[4.1.1 简单测试1 14](#_Toc60005057)

[4.1.2 简单测试2 14](#_Toc60005058)

[4.1.3 简单测试3 15](#_Toc60005059)

[4.1.4 简单测试4 16](#_Toc60005060)

[4.1.5 大数据测试 17](#_Toc60005061)

[4.2 边界测试 18](#_Toc60005062)

[4.2.1 小数据情况 18](#_Toc60005063)

[4.2.2 无路径情况 19](#_Toc60005064)

[4.3 出错测试 20](#_Toc60005065)

[4.3.1 输入数据错误 20](#_Toc60005066)

[4.3.2 结束时输入有误 20](#_Toc60005067)

# 1 分析

## 1.1 背景分析

最有名的迷宫，当属希腊神话中的米诺斯迷宫，这个迷宫是由希腊克里特国王米诺斯修建的。据说米诺斯在迷宫深处养了一只牛首人身的怪物，国王米诺斯的一个儿子死在雅典，为了替儿子复仇，国王米诺斯恳求宙斯给雅典带去瘟疫惩罚，雅典人为了阻止瘟疫的爆发，答应了米诺斯的要求，就是每年选送7对童男童女去供奉那个怪物。当雅典第三次纳贡时，雅典王子忒修斯自愿充当牺牲品，以入宫伺机杀掉那怪物，为民除害。

王子忒修斯到了米诺斯王宫，公主艾丽阿德涅对他一见钟情,于是暗中帮助王子忒修斯。公主偷偷送他一团线球和一把魔剑，告诉他将线头系在入口处，拖着线进入迷宫。这样，在走迷宫的时候，如果走不通了，也能顺着线返回。

而计算机解决迷宫问题，恰恰是运用了忒修斯的方法，从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止。为了保证在任何位置上都能沿原路退回，忒修斯需要一团线球，而计算机需要后进先出的 “栈”结构。

## 1.2 功能分析

迷宫只有两个门，一个门叫入口，另一个门叫出口。一个骑士骑马从入口进入迷宫，迷宫设置很多障碍，骑士需要在迷宫中寻找通路以到达出口。

根据功能要求，需要能产生迷宫，为了增强用户体验，程序应该能够产生不同的迷宫——根据用户的输入设定迷宫大小，在不同的位置随机设置障碍，出入口。

而程序的重点自然是给出从起点到终点的通路，而这样的通路有很多种情况，考虑到其数量庞大，应该统计其数量，并且把一些通路显示给用户（比如前10条通路以及长度最小的通路）。

同时用户可能需要多次使用程序，应该根据用户的输入(y/n)决定是否结束程序。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，计算机为了回溯，需要使用栈，所以首先需要栈这种数据结构。同时由于迷宫操作较多，在改程序中增置了Labyrinth数据结构。

## 2.2 类结构设计

对于栈，本程序使用顺序栈，即栈的顺序存储结构是利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针top指示栈顶元素在顺序栈中的位置。通常的习惯做法是以top=0表示空栈，鉴于C++中数组的下标约定从0开始,则当以C++作描述语言时,如此设定会带来很大不便；另一方面,由于栈在使用过程中所需最大空间的大小很难估计，因此，一般来说,在初始化设空栈时不应限定栈的最大容量。一个较合理的做法是：先为栈分配一个基本容量,然后在应用过程中,当栈的空间不够使用时再逐段扩大。

而对于迷宫类，主要作用是调用随机函数进行构造，同时还包括申请地图空间，释放内存，展示地图等操作。

## 2.3 成员与操作设计

**栈类（STACK）**

**为了使栈能够泛用，使用类模板T**

**私有成员：**

T\* \_top; //表示栈顶的上一个元素

T\* \_base; //表示栈底元素

int stackSize; //表示栈在内存中申请的大小

const int STACK\_INIT\_SIZE = 100; //规定栈初始化时申请的内存

**公有成员：**

Stack(); //构造函数

Status popStack(T&); //弹出栈，不会报错

T pop(); //弹出栈，会报错

Status push(T); //将T加入栈

Status get\_top(T&); //取栈首元素，不会报错

T top(); //取栈首元素，会报错

int size() //取栈的大小

void print(void(\*)(T )) //自底向上输出栈，形式参数为函数指针

bool empty() //判断栈是否为空

**Labyrinth类：**

**私有成员：**

int\* Map; //0表示不通，1表示通，2表示起点，3表示终点

//由于一维数组在计算机内部实际上以连续的内存存储，所以二维数组和一维数组在存储上是等效的

int \_setup, \_destination, \_size;

**公有成员：**

Labyrinth(const Labyrinth& ) //复制构造函数

Labyrinth(int , int); //普通构造函数

~Labyrinth() //析构函数

posType setup() //返回起点的坐标

int destination() //返回终点值

int query(posType ) //询问p坐标在地图数组里的值

int size() //询问地图的维数（大小）

int revise(posType ,int) //修改地图的值

void display(); //展示地图

SElemType部分：

struct posType

{

int x, y;

};

typedef struct

{

posType pos; //通道快在迷宫中的“坐标位置”

int di; //从此通道走向下一通道的走向

}SElemType;

**主体函数：**

dfs(Labyrinth& , Stack<SElemType>& , posType ,clock\_t&, int = 0)

//通过递归遍历所有的情况，并且使用栈结构加以记录

## 2.4 系统设计

主体部分是一个循环，根据用户输入决定退出与否；而后对于在dfs中会使用到的全局变量（记录情况数的全局变量以及记录最小情况的全局变量）进行赋初值，再调用函数进行打印标语等操作，之后进行dfs函数调用，遍历所有情况，最后根据用户输入来判断是否要继续程序。

# 3 实现

## 3.1 插入功能的实现

### 3.1.1 插入功能流程图

开始

申请空间是否足够

将值插入在top指针所指位置

top指针后移一位

申请新的空间

结束

True

False

### 3.1.2 插入功能核心代码

Status push(T e)

{

if (\_top - \_base >= stackSize)

{

//采取容量扩增的策略，用此方法扩容的平均复杂度为O(1)

\_base = (T\*)realloc(\_base, 2 \* (stackSize) \* sizeof(T));

if (!\_base)return OVERFLOW;

\_top = \_base + stackSize;

stackSize \*= 2;

}

//将e的值赋值给\_top所指，并且\_top指向下一个元素

\*\_top++ = e;

return OK;

}

## 3.2 去顶功能的实现

### 3.2.1 去顶功能流程图

开始

栈空？

\_top指针前移一位

删除失败

结束

True

True

### 3.2.2 去顶功能核心代码

//若栈为空不会报错

Status popStack(T& e)

{

if (\_base == \_top)return ERROR;

e = \*--\_top;

return OK;

}

//若栈为空会报错

T pop()

{

if (\_base == \_top) { cout << "删除失败，因为栈为空！\n"; exit(-1); }

return \*--\_top;

}

## 3.3 取顶功能的实现

### 3.3.1 取顶功能流程图

开始

栈空？

返回\_top前一位所指的元素

取顶失败

结束

True

True

### 3.3.2 查找功能核心代码

//不会报错

Status get\_top(T& e)

{

if (\_base == \_top)return ERROR;

e = \*(\_top - 1);

return OK;

}

//会报错

T top()

{

if (\_base == \_top) { cout << "取栈顶失败，因为栈为空！\n"; exit(-1); }

return \*(\_top - 1);

}

## 3.4 dfs的实现

### 3.4.1 dfs思想：

从起点开始递归，递归边界（结束条件）是到达终点。而到达每一个点，都假设它在通路上，将其置为障碍，而后对它的相邻四个方向进行搜索，寻找是否有可以到达终点的通路；如果一个点的四周经过检验，没有通路，则将其取出（或者是已经到达了递归边界，继续递归其余的情况），并且将因为将这一点放入通路而造成的变量改变复原，同时对栈中前一个元素的其他方向进行递归。

### 3.4.2 dfs核心代码

void dfs(Labyrinth& L, Stack<SElemType>& S, posType cur,clock\_t&lastTime, int num = 0)

{

int judge = L.query(cur);//根据地图在该处的位置判断该处是否可行

if (judge==-2||!judge)return;

if (judge == 3)

{

S.push(SElemType{ cur,-1 });

...//展示打印部分

if (MIN > num)

{

MIN = num;

MIN\_S = S;

}

S.pop();

return;

}

L.revise(cur, 0);

//将cur放入通路，设置为障碍

const int mov\_x[] = { 0,0,-1,1 }, mov\_y[] = { -1,1,0,0 };

//移动方向

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

{

//加入通路，并且修改当前节点

S.push(SElemType{cur,i});

cur.x += mov\_x[i], cur.y += mov\_y[i];

dfs(L, S, cur, lastTime,num + 1);//递归部分

//复原

cur.x -= mov\_x[i], cur.y -= mov\_y[i];

S.pop();

}

L.revise(cur, judge);

//将cur所指位置的值复原

}

## 3.5 总体系统的实现

### 3.5.1 总体系统流程图

继续？

打印标语

读入（读入检查）

dfs

输出结果

开始

结束

y

n

### 3.6.2 总体系统核心代码

while (1)

{

tot = 0, MIN = (1 << 31) - 1; //全局变量初始化

printHead(); //打印标语

int n, pass;

readIn(n, pass); //读入数据检查

Labyrinth L(n, pass); //构造迷宫类

L.display(); //展示迷宫

Stack<SElemType>S; //SElemType栈，该结构体中储存地图坐标以及由此通道快走向下一通道块的方向

clock\_t startTime = clock(); //记录时间，用以打印提示信息（如果dfs搜索了很长时间，需要定时输出一些内容）

dfs(L, S, L.setup(),startTime);

...打印搜索结果部分

int flag;

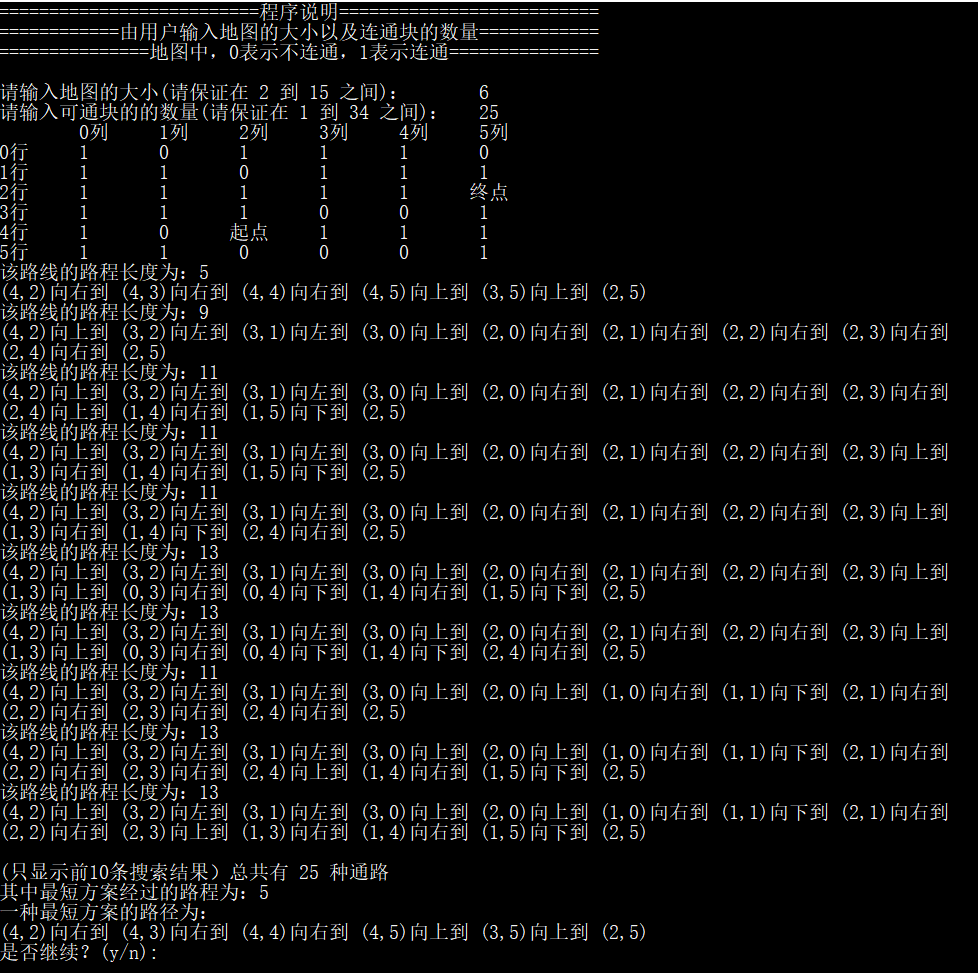
checkInput(flag);

if (flag == 2)break;

system("cls");

}

### 3.6.3 总体系统截屏示例



# 4 测试

## 4.1 功能测试

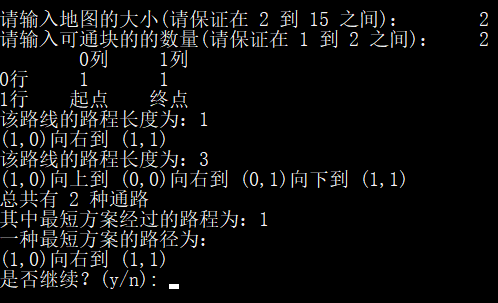
### 4.1.1 简单测试1

**测试用例**：

2

2

**实验结果：**



容易检验，结果为正确的

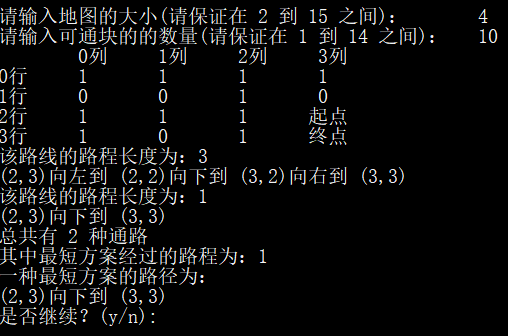
### 4.1.2 简单测试2

**测试用例：**

4

10

**实验结果：**



也容易检验，结果为正确的

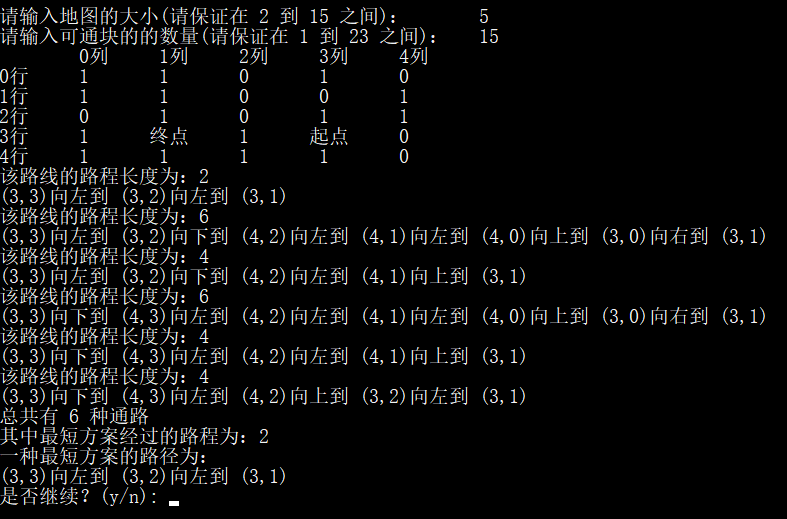
### 4.1.3 简单测试3

**测试用例：**

5

15

**实验结果：**



这种情况下也不难通过演算推的结果为正确的

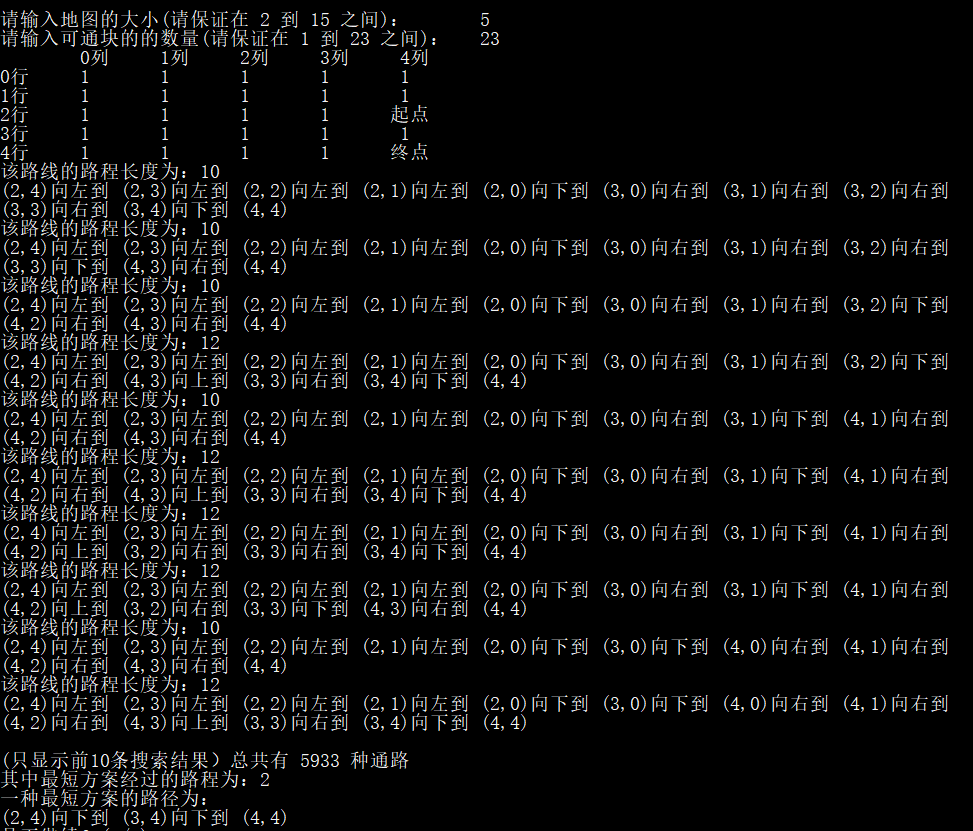
### 4.1.4 简单测试4

**测试用例：**

5

23

**实验结果：**



在这种情况下，也不能发现前10条数据以及最短通路是正确的

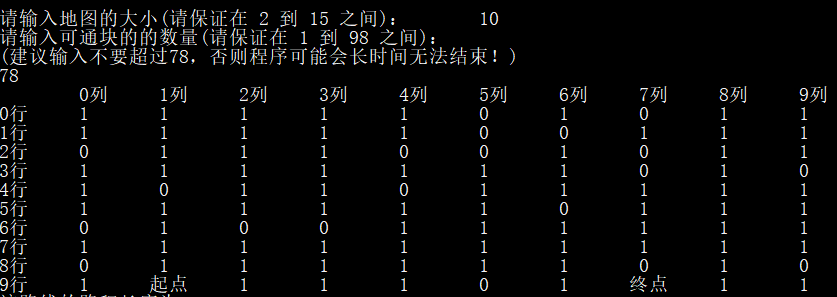
### 4.1.5 大数据测试

**测试用例：**

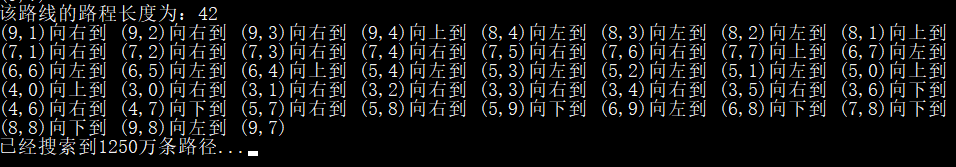
10

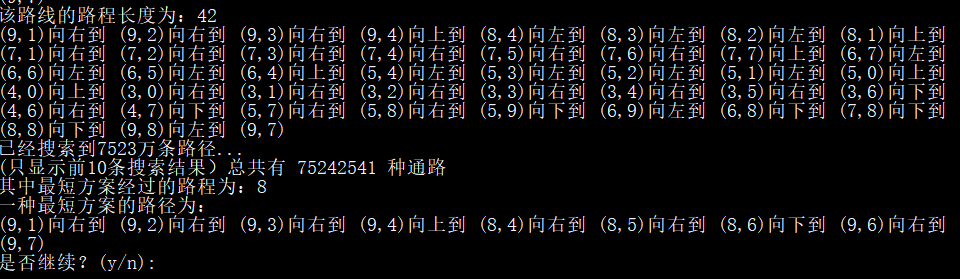
78

**实验结果：**



在搜索过程中，会对搜索到的路径数量进行实时更新





## 4.2 边界测试

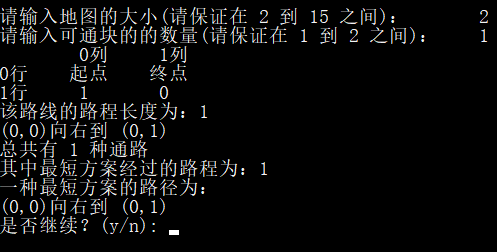
### 4.2.1 小数据情况

**测试用例：**

2

1

**实验结果：**

****

### 4.2.2 无路径情况

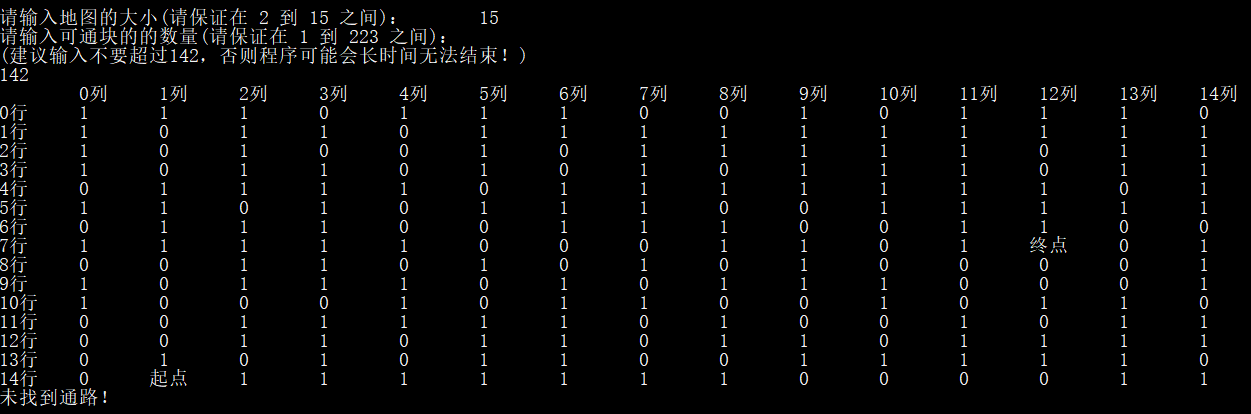
**测试用例：**

15

142

**预期结果：**程序给出无通路提示

**实验结果：**



## 4.3 出错测试

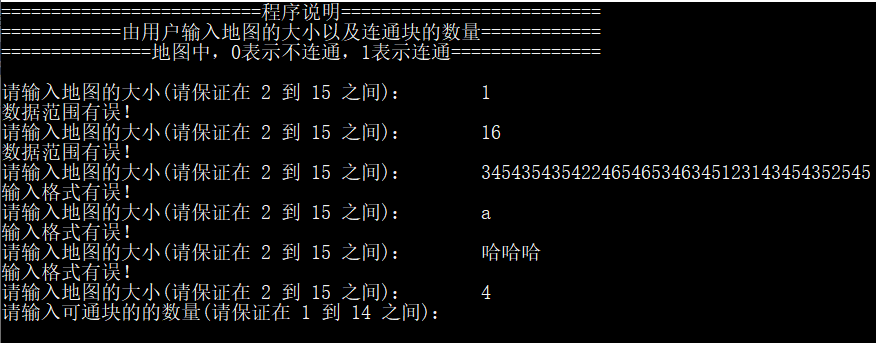
### 4.3.1 输入数据错误

**测试用例：**输入不在范围内的数字，超过int 最大值的数字，

乱码字符

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

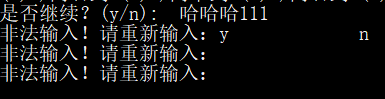
****

### 4.3.2 结束时输入有误

**测试用例：**输入乱码 在正确输入后用空格隔开乱码字符 不输入

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

****