1.题目：编程实现书P96 ADT Graph 基本操作11个，用邻接矩阵存储结构实现；

算法：见代码

代码行数：373

源代码：

//代码已验证

#include<iostream>

#include<queue>

#include<vector>//使用vector数组以避免在加点和删点时产生麻烦

#define MAX\_V 100

#define TYPE char

#define GraphKind bool

//0表示无向图 1表示有向图

using namespace std;

class Vertex

{

public:

int code;//编号

TYPE info;//点的数据

};

class Graph

{

public:

vector<vector<int>> matrix;//邻接矩阵

int vexNum,edgeNum;//点数与边数

vector<Vertex> vdata;//用以储存点信息的数组

GraphKind type;//图的类型 输入类型为无向图时可以辅助输入

};

void createGraph(Graph &G)//ok

{

//输入点和边的个数

cout<<"请依次输入图中点和边的个数"<<endl;

cin>>G.vexNum>>G.edgeNum;

//矩阵初始化

//首先初始化行数

G.matrix.resize(G.vexNum);//注意使用resize方式初始化是可以访问的 使用reserve则不能

//然后初始化列数

for(int i=0;i<G.vexNum;i++)

{

G.matrix[i].resize(G.vexNum);

}

//初始化点信息数组

G.vdata.resize(G.vexNum);

//输入图的类型

cout<<"请输入图的类型"<<endl;

cout<<"0.无向图"<<endl;

cout<<"1.有向图"<<endl;

cin>>G.type;

if(G.type)//有向图 无需考虑对称情况

{

//输入点的信息

for(int i=0;i<G.vexNum;i++)

{

G.vdata[i].code=i;

cout<<"请输入编号为"<<i<<"的点的数据"<<endl;

cin>>G.vdata[i].info;

}

//开始构建矩阵

for(int k=0;k<G.edgeNum;k++)

{

int i,j;

cout<<"请输入第"<<k+1<<"条边"<<endl;

cout<<"依次输入存在邻接关系的两点编号"<<endl;

cin>>i>>j;

G.matrix[i][j]=1;

}

}

else//无向图

{

//输入点的信息

for(int i=0;i<G.vexNum;i++)

{

G.vdata[i].code=i;

cout<<"请输入编号为"<<i<<"的点的数据"<<endl;

cin>>G.vdata[i].info;

}

//开始构建矩阵

for(int k=0;k<G.edgeNum;k++)

{

int i,j;

cout<<"请输入第"<<k+1<<"条边"<<endl;

cout<<"依次输入存在邻接关系的两点编号"<<endl;

cin>>i>>j;

G.matrix[i][j]=1;

G.matrix[j][i]=1;//和有向图的区别是对称位也要考虑

}

}

}

//销毁函数

//算法：销毁邻接矩阵与存储点数据的数组

void destroyGraph(Graph &G)

{

//销毁点信息数组

vector<Vertex>{}.swap(G.vdata);

//销毁矩阵 注意逐维销毁

//先销毁第一维存储的数据

for(int i=0;i<G.vexNum;i++)

{

vector<int>{}.swap(G.matrix[i]);

}

//然后销毁第二维

vector<vector<int>>{}.swap(G.matrix);

//注意将点数和边数置0

G.vexNum=G.edgeNum=0;

}

//返回节点数据函数 v为点的编号

TYPE getVex(Graph &G,int vcode)

{

return G.vdata[vcode].info;

}

//寻找第一个邻接点函数 若没有则返回-1表示没有找到 返回值为点的编号

int findAdjVex(Graph &G,int vcode)

{

for(int i=0;i<G.vexNum;i++)

{

if(G.matrix[vcode][i]==1)

{

return i;

}

}

//若执行完该循环还没有返回 则没有找到

return -1;

}

//寻找相对邻接点函数 若没有则返回-1表示没有找到 返回值为点的编号

int nextAdjVex(Graph &G,int vcode,int wcode)

{

for(int i=wcode+1;i<G.vexNum;i++)//从wcode的下一个点开始找起

{

if(G.matrix[vcode][i]==1)

{

return i;

}

}

//若执行完还没有找到 则w已经是v的最后一个邻接点

return -1;

}

//深度优先搜索函数

//算法：给定起始点的编号 对所有未被访问的子节点进行递归调用 注意需要传入遍历主函数中的状态量数组

void DFS(Graph &G,int index,bool \*ifVisit)

{

//将起始点置为已访问

ifVisit[index]=true;

//输出点的信息

cout<<G.vdata[index].info<<" ";

//访问所有未被访问的子节点

int vCodeToCheck=findAdjVex(G, index);//创建辅助变量 存储待检查的子节点编号 初始化为编号为index点的第一个邻接点

while(vCodeToCheck!=-1)//还有子节点没有访问完时

{

if(ifVisit[vCodeToCheck]==false)//子节点尚未访问

{

DFS(G,vCodeToCheck,ifVisit);//对子节点递归调用

}

//无论是否有递归调用 都继续访问下一个子节点

vCodeToCheck=nextAdjVex(G, index, vCodeToCheck);//返回index相对于当前节点的下一个邻接点

}

}

//深度优先遍历函数

//算法：未防止出现本身不连通的图的情况,需要对每个尚未被遍历的点走一次DFS

void DFSTraverse(Graph &G)

{

bool ifVisit[G.vexNum];//创建状态量数组用以表示某点是否已访问

for(int i=0;i<G.vexNum;i++)

{

ifVisit[i]=false;//初始化为均未被访问

}

for(int i=0;i<G.vexNum;i++)

{

//检查所有节点 若当前节点未访问 将当前节点作为起点进行DFS

if(ifVisit[i]==false)

{

DFS(G, i, ifVisit);

}

}

}

//广度优先遍历函数

//算法：与层次遍历类似 注意一旦入队就要修改访问标志 这样操作才能避免重复入队(存在同一子节点双亲有多个的情况）

void BFS(Graph &G,int index,bool \*ifVisit)

{

queue<int> helpQ;//辅助队列

helpQ.push(index);//起始点入队

ifVisit[index]=true;//修改标志

while(!helpQ.empty())

{

int indexOfOutVex=helpQ.front();

cout<<G.vdata[indexOfOutVex].info<<" ";//输出点的信息

helpQ.pop();//删除第一个节点

//下进行子节点入队操作

int vToCheck=findAdjVex(G, index);//辅助变量 初始化为第一个邻接点

while(vToCheck!=-1)//邻接点没有全部访问完

{

if(ifVisit[vToCheck]==false)//对尚未访问的邻接点

{

helpQ.push(vToCheck);//入队

ifVisit[vToCheck]=true;//入队后立刻修改标志

}

vToCheck=nextAdjVex(G, index, vToCheck);//偏移 返回起始点相对于当前节点的下一邻接点

}

//执行完该循环之后将继续执行出队操作

}

}

//广度优先遍历函数

void BFSTraverse(Graph &G)

{

bool ifVisit[G.vexNum];

for(int i=0;i<G.vexNum;i++)

{

ifVisit[i]=false;

}

for(int i=0;i<G.vexNum;i++)

{

if(ifVisit[i]==false)

{

BFS(G, i, ifVisit);

}

}

}

//添加边函数

void insertEdge(Graph &G,int vcode,int wcode)

{

G.edgeNum++;

if(G.type)

{

G.matrix[vcode][wcode]=1;

cout<<"添加完成"<<endl;

}

else

{

G.matrix[vcode][wcode]=1;

G.matrix[wcode][vcode]=1;

cout<<"添加完成"<<endl;

}

}

//添加点函数

void insertVex(Graph &G,Vertex &newVex)

{

//点数+1

G.vexNum++;

//首先根据新点数 重新调整vdata和matrix的大小

G.vdata.resize(G.vexNum);

G.matrix.resize(G.vexNum);

for(int i=0;i<G.vexNum;i++)

{

G.matrix[i].resize(G.vexNum);

}

//输入点的信息

newVex.code=G.vexNum-1;//将点编号为最后一个

cout<<"您所加入的点的编号为"<<newVex.code<<endl;

cout<<"您所加入的点的信息为"<<newVex.info<<endl;

G.vdata[G.vexNum-1]=newVex;//加入该点

//输入邻接情况

cout<<"请输入与该点关联的边数"<<endl;

int plusEdgeNum;

cin>>plusEdgeNum;

//边数增加

G.edgeNum+=plusEdgeNum;

//有向图

if(G.type)

{

for(int k=0;k<plusEdgeNum;k++)

{

cout<<"请按顺序依次输入关联边的起点与终点编号"<<endl;

int i,j;

cin>>i>>j;

insertEdge(G, i, j);

cout<<"第"<<k+1<<"条边加入完成"<<endl;

}

}

//无向图

else

{

for(int k=0;k<plusEdgeNum;k++)

{

cout<<"请输入与加入点邻接的点的编号"<<endl;

int i;

cin>>i;

insertEdge(G, newVex.code, i);

cout<<"第"<<k+1<<"条边加入完成"<<endl;

}

}

cout<<"添加完成"<<endl;

}

//删除边函数

void deleteEdge(Graph &G,int vcode,int wcode)

{

G.edgeNum--;

if(G.type)

{

G.matrix[vcode][wcode]=0;

cout<<"删除完毕"<<endl;

}

else

{

G.matrix[vcode][wcode]=0;

G.matrix[wcode][vcode]=0;

cout<<"删除完毕"<<endl;

}

}

//删除点函数 注意除了删除点 还要删除与之相关的所有边 并且删除完之后需要重置所有点的编号

void deleteVex(Graph &G,int vcode)

{

G.vexNum--;//点数自减

//vdata中删除

vector<Vertex>::iterator it=G.vdata.begin();

G.vdata.erase(it+vcode);

//重置编号

for(int i=0;i<G.vexNum;i++)

{

G.vdata[i].code=i;

}

//矩阵中删除

//无向图和有向图统计去掉边数目的情况不同

if(G.type)

{

int count=0;//计数器

//检查起点边

for(int j=0;j<G.vexNum;j++)

{

if(G.matrix[vcode][j]==1)

{

count++;

}

}

//检查终点边

for(int i=0;i<G.vexNum;i++)

{

if(G.matrix[i][vcode]==1)

{

count++;

}

}

G.edgeNum-=count;

}

else

{

int count=0;

//只需检查一次即可 也无需乘2 因为对称位的算作是同一条边

for(int j=0;j<G.vexNum;j++)

{

if(G.matrix[vcode][j]==1)

{

count++;

}

}

G.edgeNum-=count;

}

//下进行矩阵删除操作

//删除点所在行

vector<vector<int>>::iterator rowIt=G.matrix.begin();

G.matrix.erase(rowIt+vcode);

//删除点所在列

for(int i=0;i<G.vexNum;i++)//由于之前已经处理过vexNum 所以此处的vexNum已经是--过的了

{

vector<int>::iterator colIt=G.matrix[i].begin();

G.matrix[i].erase(colIt+vcode);

}

//之前已经重置过点的编号了

cout<<"删除完毕"<<endl;

}

//遍历函数

void traverse(Graph &G)

{

cout<<"下输出所有点的信息"<<endl;

for(int i=0;i<G.vexNum;i++)

{

cout<<"编号："<<G.vdata[i].code<<" "<<"信息："<<G.vdata[i].info<<endl;

}

cout<<"下输出邻接矩阵"<<endl;

for(int i=0;i<G.vexNum;i++)

{

for(int j=0;j<G.vexNum;j++)

{

cout<<G.matrix[i][j]<<" ";

}

cout<<endl;

}

}

int main()

{

Graph G;

createGraph(G);

traverse(G);

cout<<getVex(G, 2)<<endl;

cout<<findAdjVex(G, 2)<<endl;

cout<<nextAdjVex(G, 2, 3)<<endl;

DFSTraverse(G);

BFSTraverse(G);

insertEdge(G, 0, 4);

traverse(G);

Vertex v;

v.code=5;

v.info='5';

insertVex(G, v);

traverse(G);

deleteEdge(G, 0, 4);

traverse(G);

deleteVex(G, 5);

traverse(G);

destroyGraph(G);

traverse(G);

return 0;

}

2.题目：输入N个权值（1-100正整数），建立哈夫曼树

算法：见代码

代码行数：130

源代码：

//只验证了STL解法

#include<iostream>

#include<vector>

#include<map>

#include<list>

#include<algorithm>

using namespace std;

class HTnode

{

public:

HTnode(int weight,int parent,int lchild,int rchild,int code)

{

this->weight=weight;

this->parent=parent;

this->lchild=lchild;

this->rchild=rchild;

this->code=code;

}

HTnode(int weight,int code)

{

this->weight=weight;

this->code=code;

parent=0;

lchild=0;

rchild=0;

}

HTnode()

{

}

int weight,parent,lchild,rchild;

int code;//为便于操作 再存储一下节点的编号

//对每个点都需要知道双亲和子节点 注意要分清左节点与右节点

//最小点放左边 次小点放右边 以此保证建立出的哈夫曼树唯一

};

//选择最小两数的函数 分别用非STL的方法和STL的方法来实现

pair<int,int> selectTwoSmallNumbers\_STL(vector<HTnode> &Nodes,map<int,HTnode> &nodesAbleToSelectFrom)

{

//算法：

//first\_min和second\_min两个量 注意要分别初始化成不同的值 遍历数组 先跟first\_min比较 如果比first\_min小 替换之 如果比first\_min大 就与second\_min比较 如果比second\_min小 替换之

int firstMinCode=0;

int secondMinCode=0;

firstMinCode=secondMinCode=(\*nodesAbleToSelectFrom.begin()).first;

for(auto it=++nodesAbleToSelectFrom.begin();it!=nodesAbleToSelectFrom.end();it++)//从第二位开始比较

{

//下面三行测试用

cout<<(\*it).first<<endl;

cout<<firstMinCode<<endl;

cout<<secondMinCode<<endl;

if((\*it).second.weight<nodesAbleToSelectFrom[firstMinCode].weight)

{

//现将firstMinCode的值赋给secondMinCode

secondMinCode=firstMinCode;

//再将新的最小值赋给firstMinCode

firstMinCode=(\*it).first;

}

else if((\*it).second.weight<nodesAbleToSelectFrom[secondMinCode].weight)

{

secondMinCode=(\*it).first;

}

}

//如果查找完之后 firstMinCode==secondMinCode 则表示后面所有数都比第一个数大 此时应确定了firstMinCode为 将secondMinCode赋值成第二位后再找一次最小值即可

if(firstMinCode==secondMinCode){

secondMinCode=(\*(++nodesAbleToSelectFrom.begin())).first;//将secondMinCode赋值成第二位 再查找一遍最小值

cout<<secondMinCode<<endl;

for(auto it=++nodesAbleToSelectFrom.begin();it!=nodesAbleToSelectFrom.end();it++){

cout<<(\*it).first<<endl;

if((\*it).second.weight<nodesAbleToSelectFrom[secondMinCode].weight){

secondMinCode=(\*it).first;

}

}

}

//查找完后记得把两个最小权值点从可比较数组中删除

//注意！ 删除一个元素之后 有可能导致另一元素的迭代器失效 此处采用字典直接删除

nodesAbleToSelectFrom.erase(firstMinCode);

nodesAbleToSelectFrom.erase(secondMinCode);

return pair<int,int>(firstMinCode,secondMinCode);

}

//遍历输出函数

void printNodes(vector<HTnode> &Nodes)

{

cout<<"code\t"<<"weight\t"<<"parent\t"<<"lchild\t"<<"rchild"<<endl;

for(vector<HTnode>::iterator it=Nodes.begin();it!=Nodes.end();it++)

{

cout<<(\*it).code<<"\t"<<(\*it).weight<<"\t"<<(\*it).parent<<"\t"<<(\*it).lchild<<"\t"<<(\*it).rchild<<endl;

}

}

int main()

{

vector<HTnode> Nodes;//存储所有节点

map<int,HTnode> nodesAbleToSelectFrom;//创建一个数组存储所有能够用来比较的点 从而避免反复的重复查找

//输入权值

cout<<"请输入带有权值的点数"<<endl;

int n;

cin>>n;

Nodes.reserve(2\*n);//预留空间 注意多留一个 第一个空间是不用的 此数组中下标即为编号

HTnode noUseNode(-1,-1,-1,-1,0);

Nodes.push\_back(noUseNode);

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

{

cout<<"请输入编号为"<<i<<"的带权点的权值"<<endl;

int weight;

cin>>weight;

HTnode newNode(weight,i);

Nodes.push\_back(newNode);

nodesAbleToSelectFrom.emplace(i,newNode);

cout<<"第"<<i<<"个点输入完毕"<<endl;

}

//下进行哈夫曼树的建立

for(int i=n+1;i<2\*n;i++)//从第一个非权值点开始到容器末尾

{

//首先需要找出从1到i-1的编号中最小的两个数

pair<int,int> twoSmallestNumbers;

twoSmallestNumbers=selectTwoSmallNumbers\_unSTL(i-1,Nodes);

cout<<twoSmallestNumbers.first<<" "<<twoSmallestNumbers.second<<endl;

//创建新节点

int weightOfNewNode=Nodes[twoSmallestNumbers.first].weight+Nodes[twoSmallestNumbers.second].weight;

HTnode newNode(weightOfNewNode,0,twoSmallestNumbers.first,twoSmallestNumbers.second,i);

//将新节点加入点组

Nodes.push\_back(newNode);

//注意将选出两个点的parent修改为该新节点

Nodes[twoSmallestNumbers.first].parent=Nodes[twoSmallestNumbers.second].parent=newNode.code;

printNodes(Nodes);

//创建出新的节点之后记得要将新节点加入到nodesAbleToSelectFrom中

nodesAbleToSelectFrom.emplace(i,newNode);

}

//建立完成 遍历Nodes输出

printNodes(Nodes);

return 0;

}

3.题目：编写函数，对二叉链表结构的二叉树，求宽度。（书P94 4）

算法：采用BFS的基础思路 对节点进行编号 用每一层的最大编号减去最小编号+1得宽度 然后求出最大值

代码行数：67

源代码：

#include<iostream>

#include<vector>

#include<queue>

#include<algorithm>

using namespace std;

class BiNode

{

public:

char data;

BiNode \*lchild;

BiNode \*rchild;

};

void createBiNodeTree(BiNode\* &T)

{

char ch;

cin>>ch;

if(ch=='#')

{

T=NULL;

}

else

{

T=new BiNode;

T->data=ch;

createBiNodeTree(T->lchild);

createBiNodeTree(T->rchild);

}

}

unsigned long long getWidthOfBiNodeTree(BiNode\* &T)

{

//采用两个数组 一个用来记录当前层的节点 一个用来记录下一层的节点

vector<pair<BiNode\*,unsigned long long>> currentLayer;

//首先将根节点加入current 编号为1

currentLayer.emplace\_back(T,1);

//计数器

unsigned long long ans=0;

if(T==NULL)

{

return 0;

}

while(!currentLayer.empty())

{

vector<pair<BiNode\*,unsigned long long>> nextLayer;//创建临时数组

//先将子代点加入nextLayer中

for(auto &[node,code]:currentLayer)

{

if(node->lchild)

{

nextLayer.emplace\_back(node->lchild,2\*code);

}

if(node->rchild)

{

nextLayer.emplace\_back(node->rchild,2\*code+1);

}

}

//创建完成后 求出当前层的宽度 并更新ans

ans=max(ans,currentLayer.back().second-currentLayer.front().second);

//将current更新为next 继续前面的过程 由于nextLayer不再使用 可以使用move函数

currentLayer=move(nextLayer);

}

return ans;

}

int main()

{

return 0;

}

4.题目：编写函数，对一棵以孩子-兄弟链表表示的树，输出第i层的所有元素。

算法：先按照孩子找到第i层最左侧的点 然后从该点向右遍历所有兄弟

代码行数：51

源代码：

#include<iostream>

using namespace std;

class TreeNode

{

public:

char data;

TreeNode \*child;//第一个左子节点

TreeNode \*brother;

};

void createTree(TreeNode\* &T)

{

char ch;

cin>>ch;

if(ch=='#')

{

T=NULL;

}

else

{

T=new TreeNode;

T->data=ch;

createTree(T->child);

createTree(T->brother);

}

}

void printElementsOfIthLayer(TreeNode\* &T,int i)

{

//首先找到第i层最左侧的点

if(T==NULL)

{

cout<<"空树"<<endl;

return;

}

TreeNode \*startNode=T;

for(int k=0;k<i-1;k++)//对第i层的节点 要找i-1次

{

startNode=startNode->child;

}

//然后从startNode开始遍历兄弟节点

TreeNode \*currentNode=startNode;//创建辅助节点

while(currentNode)

{

cout<<currentNode->data<<" ";

//偏移

currentNode=currentNode->brother;

}

}

int main()

{

return 0;

}

5.题目：

俄罗斯方块是俄罗斯人阿列克谢·帕基特诺夫发明的一款休闲游戏。

游戏在一个15行10列的方格图上进行，方格图上的每一个格子可能已经放置了方块，或者没有放置方块。每一轮，都会有一个新的由4个小方块组成的板块从方格图的上方落下，玩家可以操作板块左右移动放到合适的位置，当板块中某一个方块的下边缘与方格图上的方块上边缘重合或者达到下边界时，板块不再移动，如果此时方格图的某一行全放满了方块，则该行被消除并得分。

在这个问题中，你需要写一个程序来模拟板块下落，你不需要处理玩家的操作，也不需要处理消行和得分。

具体的，给定一个初始的方格图，以及一个板块的形状和它下落的初始位置，你要给出最终的方格图。

算法：递归 分析当前图案位置下移一格的位置 如果下一位置已经有方格 在当前停止 如果没有 递归下移

代码行数：125

源代码：

//算法：模拟一个下移一格的函数 如果下一格位置有1则放置在当前位置 直到放置到结尾

#include<iostream>

#include<vector>

using namespace std;

void printBlocks(vector<vector<int>> &blocks)

{

for(int i=0;i<15;i++)

{

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

{

cout<<blocks[i][j]<<" ";

}

cout<<endl;

}

}

//找到板块图案在游戏界面中的初始坐标

void findIndexsInBlocks(vector<pair<int,int>> &indexsOfPatterns,int startCol)

{

//行坐标无需变更 只需变更列坐标

for(vector<pair<int,int>>::iterator it=indexsOfPatterns.begin();it!=indexsOfPatterns.end();it++)

{

(\*it).second+=startCol;

}

}

//检查冲突函数

bool isAttacking(int row,int col,vector<vector<int>> &blocks)

{

if(blocks[row][col]==1)

{

return true;

}

else

{

return false;

}

}

//下移函数 检查下移后的坐标中是否有1 如果有 将当前位的格子赋值为1 如果没有 继续下移 直到达到界面底端

//递归思想

void downOneBlock(vector<pair<int,int>> &indexOfPatterns,vector<vector<int>> &blocks,int bottomOfPatterns)

{

//如果已经到达了界面底部 直接返回

if(bottomOfPatterns==14)

{

return;

}

else//如果尚未达到底部

{

//检查

for(vector<pair<int,int>>::iterator it=indexOfPatterns.begin();it!=indexOfPatterns.end();it++)

{

if(isAttacking((\*it).first+1, (\*it).second, blocks))

{

return;//如果有冲突 直接返回

}

}

//如果前面没有返回 意味着没有发生冲突的情况 即图案可以下移

//下移

for(vector<pair<int,int>>::iterator it=indexOfPatterns.begin();it!=indexOfPatterns.end();it++)

{

(\*it).first+=1;

}

//底部下移

bottomOfPatterns++;

//递归

downOneBlock(indexOfPatterns, blocks, bottomOfPatterns);

}

}

int main()

{

//游戏界面

vector<vector<int>> blocks;

//输入图形 直接存储所有1块在板块图形中的坐标

vector<pair<int,int>> indexsOfPatterns;

//记录图案底部的坐标

int bottomOfPatterns=-1;

//初始列 注意是整个板块的初始列

int startCol;

//预设空间

blocks.resize(15);

for(int i=0;i<15;i++)

{

blocks[i].reserve(10);

}

//输入

for(int i=0;i<15;i++)

{

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

{

int x;

cin>>x;

blocks[i].push\_back(x);

}

}

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

{

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

{

int x;

cin>>x;

if(x==1)

{

indexsOfPatterns.emplace\_back(i,j);//存储有效方块在图案中的坐标

if(i>bottomOfPatterns)

{

bottomOfPatterns=i;//更新底部坐标

}

}

}

}

cin>>startCol;

//注意到此处的startCol是实际列数 需要在此基础上自减

startCol--;

//输入完成

//找到在界面中的坐标

findIndexsInBlocks(indexsOfPatterns, startCol);

downOneBlock(indexsOfPatterns, blocks, bottomOfPatterns);

//此时的indexOfPatterns存储有最终的添加点坐标

//修改原界面

for(vector<pair<int,int>>::iterator it=indexsOfPatterns.begin();it!=indexsOfPatterns.end();it++)

{

blocks[(\*it).first][(\*it).second]=1;

}

//输出答案

printBlocks(blocks);

return 0;

}

总代码行数：746