## 算法与复杂性HW02

姓名：林茵 学号：515021910066 班级：F1503030

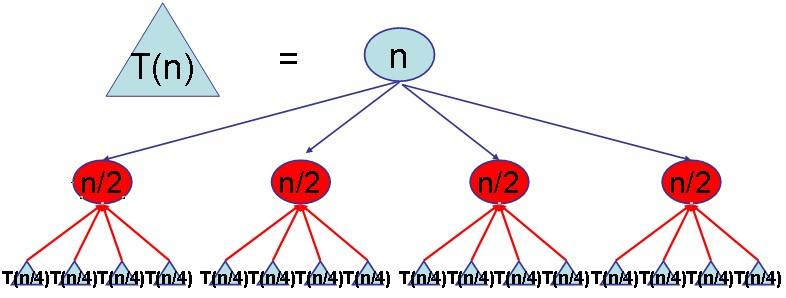
PART1：

城市轮廓线分治问题：

Question：给定城市里几座矩形建筑的外形和位置，利用分治法求出这些建筑的(两维)轮廓，并消去隐藏线。建筑Bi通过三元组(Li，Hi，Ri)来表示。Li和Ri分别表示建筑的左右x坐标，而Hi表示建筑的高度。一个轮廓是一列x坐标以及与它们相连的高度，按照从左到右排列。要求从输入文件in.dat读入建筑数目和各个三元组，结果写入文件out.dat。

1. 算法思想：

采用分治的方法，首先将相邻的两栋楼房归并，然后归并相邻的轮廓线，最后得到总的合并轮廓线。



1. 算法时间复杂度分析：由于分治递推公式T(n)=T(n/2)+n有，T(n)=O(nlogn)
2. 代码实现

#include <iostream>

#include<fstream>

#include<string>

#include<vector>

#include<cmath>

using namespace std;

//储存楼房的结构

struct Building

{

int R;

int H;

int L;

};

//只有两栋楼房的情况

vector<int>MergeTwo(Building &b1,Building &b2)

{

Building first,second;

if(min(b1.L,b2.L)==b1.L)

{

first=b1;

second=b2;

}

else

{

first=b2;

second=b1;

}

vector <int> outline;

if(first.R<second.L)

{

outline.push\_back(first.L);

outline.push\_back(first.H);

outline.push\_back(first.R);

outline.push\_back(0);

outline.push\_back(second.L);

outline.push\_back(second.H);

outline.push\_back(second.R);

outline.push\_back(0);

}

if(first.R==second.L)

{

outline.push\_back(first.L);

outline.push\_back(first.H);

outline.push\_back(first.R);

outline.push\_back(second.H);

outline.push\_back(second.R);

outline.push\_back(0);

}

if(first.R>second.L&&first.R<second.R&&first.H>second.H)

{

outline.push\_back(first.L);

outline.push\_back(first.H);

outline.push\_back(first.R);

outline.push\_back(second.H);

outline.push\_back(second.R);

outline.push\_back(0);

}

if(first.R>second.L&&first.R<second.R&&first.H<second.H)

{

outline.push\_back(first.L);

outline.push\_back(first.H);

outline.push\_back(second.L);

outline.push\_back(second.H);

outline.push\_back(second.R);

outline.push\_back(0);

}

if(first.R>second.L&&first.R<second.R&&first.H==second.H)

{

outline.push\_back(first.L);

outline.push\_back(first.H);

outline.push\_back(second.R);

outline.push\_back(0);

}

if(first.R>second.R&&first.H>=second.H)

{

outline.push\_back(first.L);

outline.push\_back(first.H);

outline.push\_back(first.R);

outline.push\_back(0);

}

if(first.R>second.R&&first.H<second.H)

{

outline.push\_back(first.L);

outline.push\_back(first.H);

outline.push\_back(second.L);

outline.push\_back(second.H);

outline.push\_back(second.R);

outline.push\_back(first.H);

outline.push\_back(first.R);

outline.push\_back(0);

}

return outline;

}

vector<int> SingleBuilding(Building one)

{

vector<int> outline;

outline.push\_back(one.L);

outline.push\_back(one.H);

outline.push\_back(one.R);

outline.push\_back(0);

return outline;

}

//合并两个轮廓线

vector<int> MergeTwoOutline(vector<int> L,vector<int> R)

{

int cursorL=0;

int cursorR=0;

vector <int> outline;

int currentHigh=0;

while(cursorL<L.size()&&cursorR<R.size())

{

if(L[cursorL]<R[cursorR])

{

if(L[cursorL+1]>currentHigh)

{

outline.push\_back(L[cursorL]);

cursorL++;

outline.push\_back(L[cursorL]);

cursorL++;

currentHigh=L[cursorL-1];

if(cursorL==L.size()){cursorL-1;}

}

if (L[cursorL+1]<currentHigh)

{

outline.push\_back(L[cursorL]);

cursorL++;

if(L[cursorL]>R[cursorR-1])

{

outline.push\_back(L[cursorL]);

cursorL++;

currentHigh=L[cursorL-1];

}

else

{

outline.push\_back(R[cursorR-1]);

cursorL++;

currentHigh=R[cursorR-1];

}

}

else

cursorL+=2;

}

if(R[cursorR]<L[cursorL])

{

if(R[cursorR+1]>currentHigh)

{

outline.push\_back(R[cursorR]);

cursorR++;

outline.push\_back(R[cursorR]);

cursorR++;

currentHigh=R[cursorR-1];

if(cursorR==R.size()){cursorR-1;}

}

if (R[cursorR+1]<currentHigh)

{

outline.push\_back(R[cursorR]);

cursorR++;

if(R[cursorR]>L[cursorL-1])

{

outline.push\_back(R[cursorR]);

cursorR++;

currentHigh=R[cursorR-1];

}

else

{

outline.push\_back(L[cursorL-1]);

cursorR++;

currentHigh=L[cursorL-1];

}

}

else

cursorR+=2;

}

if(L[cursorL]==R[cursorR])

{

if(L[cursorL+1]>currentHigh||R[cursorR+1]>currentHigh)

{

outline.push\_back(L[cursorL]);

cursorL++;cursorR++;

if(L[cursorL]>=R[cursorR]) {outline.push\_back(L[cursorL]);currentHigh=L[cursorL];}

else {outline.push\_back(R[cursorR]);currentHigh=R[cursorR];}

cursorL++;cursorR++;

}

else {cursorL+=2;cursorR+=2;}

}

}

if(cursorL==L.size()) for(int i=cursorR;i<R.size();i++) outline.push\_back(R[i]);

else for(int i=cursorL;i<L.size();i++) outline.push\_back(L[cursorL]);

return outline;

}

//总的分治函数

vector<int> MergeBuildings(Building building[],int leftIndex,int rightIndex)

{

if(rightIndex-leftIndex<=1)

{

if(rightIndex-leftIndex==0)

return SingleBuilding(building[rightIndex]);

else

return MergeTwo(building[leftIndex],building[rightIndex]);

//递归终止条件

}

else

{

int middle=(leftIndex+rightIndex)/2;

vector<int> left=MergeBuildings(building,leftIndex,middle);

vector<int> right=MergeBuildings(building,middle+1,rightIndex);

return MergeTwoOutline(left,right);

}

}

int main()

{

//读入并储存数据的部分

ifstream in;

string fileName="in.dat";

in.open(fileName);

if(!in){cout<<"打开文件出错";return 1;}

int temp,number;

in>>number;

Building \*building = new Building [number];

int index=0;

while(in>>temp)

{

building[index].L=temp;cout<<building[index].L;

in>>temp;

building[index].H=temp;cout<<building[index].H;

in>>temp;

building[index].R=temp;cout<<building[index].R;

index++;

}

vector <int> mergeResult;

mergeResult=MergeBuildings(building,0,number-1);

if(mergeResult.empty())

cout<<"It is empty";

else{

vector<int>::const\_iterator itr,itre;

itr=mergeResult.begin();

itre=mergeResult.end();

ofstream outfile("out.dat",ios::out);

if(!outfile){cerr<<"open error"<<endl;return(-1);}

do{

cout<<\*itr<<' ';

outfile<<\*itr<<' ';

itr++;

}while(itr!=itre);

outfile.close();

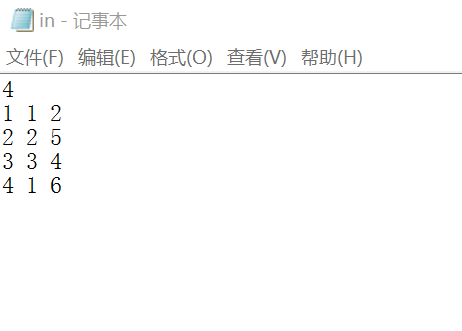
}

cout<<endl;

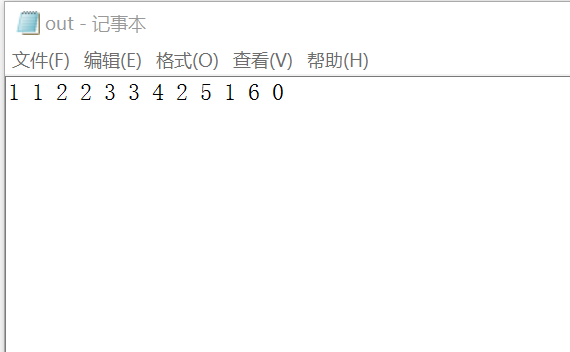
return 0;

}

1. 运行结果截图
2. 输入的in.dat如图



1. 输出的out.dat如图

 输入楼房如图（不同的颜色表示不同的楼房）

经检验，输出的结果正确。

PART2:

Question：从n个数中选出前k个小的数

解决方法与复杂性的估计

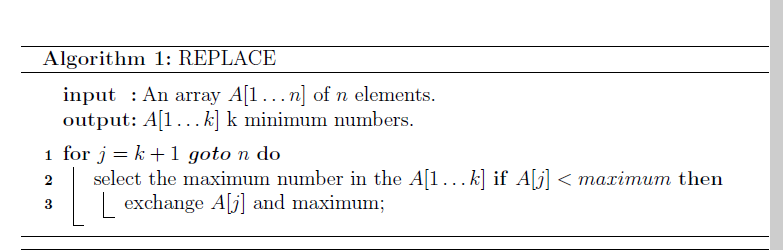
1. 最普通的方法就是将这几个数先排序，然后通过选出前n 个比较小的数来得到所求。以快速排序算法为例，算法的时间复杂度为O(nlogn).

若通过堆排序法来得到前k个小的数，首先建堆的时间复杂度为O(n)，然后加上k次pop的操作，时间复杂度约为O(klogn).

1. 基于交换的算法：该算法的思想为，假装前k个数为满足要求的数，通过对k+1~n的扫描以及与k中最大数的比较以及交换得到前k个最小的数。

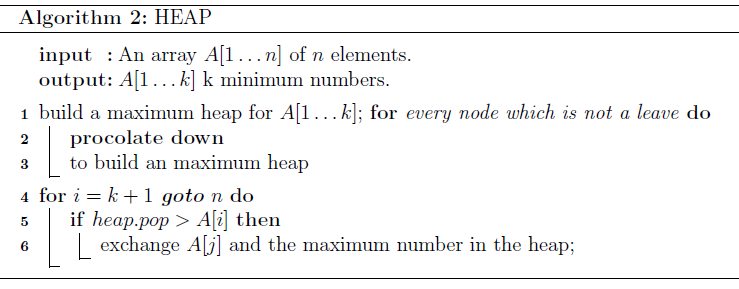
由于扫描的时间为O(n-k)求最大数的操作时间复杂度为O(k)，因此该算法的时间复杂度为O(n\*k)，需要有一个另外的储存空间用于元素的交换。

用latex编辑伪代码如下



1. 对交换算法的优化，其实在上述算法的基础上，将求最大数的算法用一个最大化堆来实现。时间复杂度为O(n\*logk)，需要有一个另外的储存空间用于储存每次出队的元素

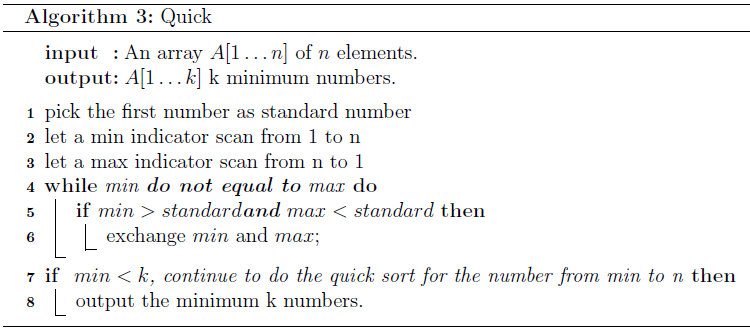
用latex编辑伪代码如下



1. 基于快速排序的优化算法

多次快排一直到中间元素的坐标大于k,时间复杂度为O(nlogk)，需要有一个另外的储存空间用于元素的交换。

用latex编辑伪代码如下



PART3

Question：实现判断连连看中两个位置是否能够连接

算法思想：采用BFS的方法，将起点可以直接到达的位置入队对应的turn标记为0，然后一一出队，每次出队的时候判断出队的元素是否为所想要到达的终点，然后将出队的元素可以直接到达的位置入队，对应的turn为出队元素的turn+1，当出队元素的turn的值为3时，不再将其能够直接到达的元素入队。当队列为空时，则返回false。

算法实现：

#include <iostream>

#include<fstream>

#include<string>

#include<queue>

using namespace std;

class P

{

public:

int x,y,turn;

void operator=(P & p)

{

x=p.x;

y=p.y;

turn=p.turn;

}

};//用于储存点

bool bfs(int \*\*Map,P start,P terminal,int x,int y)

{

queue <P> q;

P pre,temp;

q.push(start);

while(!q.empty())

{

pre=q.front();

q.pop();

if (pre.x==terminal.x&&pre.y==terminal.y)return true;

//判断是否为终点

if(pre.turn<3)//只有转弯次数小于3的点才可以将接下来的点入队

{

for(int i=pre.x+1;i<x;i++)//水平方向

{

if(Map[i][pre.y]==0||Map[i][pre.y]==Map[terminal.x][terminal.y])

{

temp.x=i;

temp.y=pre.y;

temp.turn=pre.turn+1;

//Map[i][pre.y]=-1;//防止访问过的再次入队,但是这样会引发错误，无法避免的时间复杂度的上升

q.push(temp);

}

else break;

}

for(int i=pre.x-1;i>0;i--)//水平方向

{

if(Map[i][pre.y]==0||Map[i][pre.y]==Map[terminal.x][terminal.y])

{

temp.x=i;

temp.y=pre.y;

temp.turn=pre.turn+1;

//Map[i][pre.y]=-1;//防止访问过的再次入队,但是这样会引发错误，无法避免的时间复杂度的上升

q.push(temp);

}

else break;

}

for(int j=pre.y+1;j<y;j++)

{

if(Map[pre.x][j]==0||Map[pre.x][j]==Map[terminal.x][terminal.y])

{

temp.x=pre.x;

temp.y=j;

temp.turn=pre.turn+1;

//Map[pre.x][j]=-1;//防止访问过的再次入队,但是这样会引发错误，无法避免的时间复杂度的上升

q.push(temp);

}

else break;

}

for(int j=pre.y-1;j>0;j--)

{

if(Map[pre.x][j]==0||Map[pre.x][j]==Map[terminal.x][terminal.y])

{

temp.x=pre.x;

temp.y=j;

temp.turn=pre.turn+1;

//Map[pre.x][j]=-1;//防止访问过的再次入队,但是这样会引发错误，无法避免的时间复杂度的上升

q.push(temp);

}

else break;

}

}

}

return false;

}

int main()

{

//地图的生成部分

ifstream in;

string fileName="in.dat";

in.open(fileName);

if(!in){cout<<"打开文件出错";return 1;}

int temp,x,y;

in>>x;

in>>y;//输入x行y列

int\*\* Map;

Map=new int\*[x];

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

Map[i]=new int[y];

P start,terminal;

in>>start.x;

in>>start.y;

in>>terminal.x;

in>>terminal.y;

start.turn=0;

terminal.turn=0;

while(in>>temp)

{

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

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

{

Map[i][j]=temp;

in>>temp;

}

}

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

{

for(int j=0;j<y;j++) cout<<Map[i][j]<<' ';

cout<<endl;

}

cout<<"The start point is:("<<start.x<<","<<start.y<<")->"<<Map[start.x][start.y]<<endl;

cout<<"The terminal point is:("<<terminal.x<<","<<terminal.y<<")->"<<Map[terminal.x][terminal.y]<<endl;

bool flag=bfs(Map,start,terminal,x,y);

if(flag)

cout<<"Yes we find it!"<<endl;

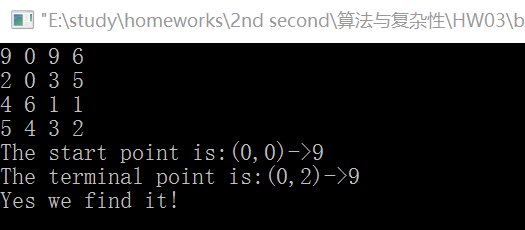
else cout<<"Sorry for not finding it"<<endl;

return 0;

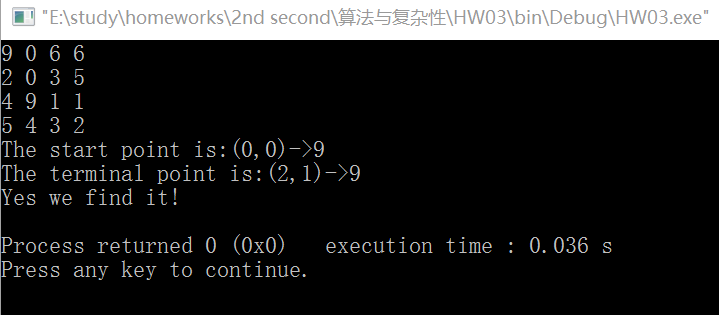
}

运行结果：

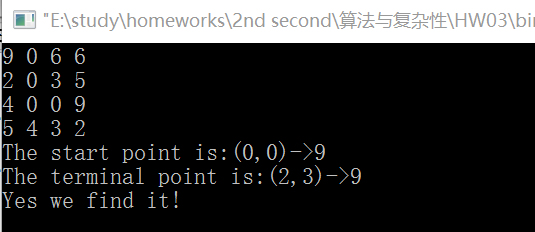
1. 不需要拐弯



1. 只拐一个弯



1. 拐两个弯



1. 拐弯次数超过2，无法到达

