1.题目：输入稀疏矩阵，建立稀疏矩阵三元组顺序结构，实现矩阵的列序遍历转置和快速转置算法。

算法：见代码

源代码：//列序遍历转制与快速转制

#include<iostream>

using namespace std;

class triple

{

public:

triple(int row,int col,int data)

{

m\_row=row;

m\_col=col;

m\_data=data;

}

triple()

{

}

int m\_row;//行标

int m\_col;//列标

int m\_data;//元素值

};

class matrix

{

public:

matrix(int rownum,int colnum,int elemnum)

{

m\_rownum=rownum;

m\_colnum=colnum;

m\_elemnum=elemnum;

m\_elem=new triple[elemnum];

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

{

int row,col,data;

cout<<"请输入第"<<i+1<<"个点的信息"<<endl;

cin>>row>>col>>data;

triple t(row,col,data);

m\_elem[i]=t;

}

}

matrix(int rownum,int colnum)//未知矩阵的初始化

{

m\_rownum=rownum;

m\_colnum=colnum;

}

triple \*m\_elem;

int m\_rownum;//行数

int m\_colnum;//列数

int m\_elemnum;//元素个数

};

//输出矩阵函数

void printMatrix(matrix &M)

{

for(int i=0;i<M.m\_elemnum;i++)

{

cout<<M.m\_elem[i].m\_row<<" "

<<M.m\_elem[i].m\_col<<" "

<<M.m\_elem[i].m\_data<<endl;

}

cout<<endl;

}

//交换行列

void exchangeRowAndCol(triple &t)

{

int temp=t.m\_row;

t.m\_row=t.m\_col;

t.m\_col=temp;

}

//列序遍历转制 A是被转制矩阵，B用来接收转制后的矩阵

//附带了按列排序功能

//T(O)=col\*elemnum 普通转制为elemnum\*elemnum

void columnTraverseTranspose(matrix A,matrix &B)

{

int count=0;//用以记录B中当前空白存储位的下标

//为B建立存储空间

B.m\_elemnum=A.m\_elemnum;

B.m\_elem=new triple[B.m\_elemnum];

for(int colToCheck=0;colToCheck<A.m\_colnum;colToCheck++)

{

for(int indexOfTripleToCheck=0;indexOfTripleToCheck<A.m\_elemnum;indexOfTripleToCheck++)

{

if(A.m\_elem[indexOfTripleToCheck].m\_col==colToCheck)

{

B.m\_elem[count]=A.m\_elem[indexOfTripleToCheck];

exchangeRowAndCol(B.m\_elem[count]);

count++;

}

}

}

}

//T(O)=col+elemnum

void quickTranspose(matrix A,matrix &B)

{

B.m\_elemnum=A.m\_elemnum;

B.m\_elem=new triple[A.m\_elemnum];

int rowNum[A.m\_colnum];

for(int i=0;i<A.m\_colnum;i++)

{

rowNum[i]=0;

}

int rowStart[A.m\_colnum];

rowStart[0]=0;

for(int i=0;i<A.m\_elemnum;i++)

{

rowNum[A.m\_elem[i].m\_col]++;

}

for(int i=1;i<A.m\_colnum;i++)

{

rowStart[i]=rowStart[i-1]+rowNum[i-1];

}

for(int i=0;i<A.m\_elemnum;i++)

{

B.m\_elem[rowStart[A.m\_elem[i].m\_col]]=A.m\_elem[i];

exchangeRowAndCol(B.m\_elem[rowStart[A.m\_elem[i].m\_col]]);

rowStart[A.m\_elem[i].m\_col]++;

}

}

int main()

{

matrix A(3,4,4);

matrix B(3,4);

quickTranspose(A, B);

printMatrix(B);

return 0;

}

2.题目：求矩阵马鞍点

算法：首先遍历矩阵，求出每一行的最小值所在的列与每一列的最大值所在的行，存入两个数组。然后再次遍历矩阵，判断每一行最小值所在的列与改列最大值所在的行是否相等

源代码：

#include<iostream>

#include<vector>

using namespace std;

int main()

{

int m;//行数

int n;//列数

cin>>m>>n;

int matrix[m][n];

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

{

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

{

cin>>matrix[i][j];

}

}

int minOfRow[m];//记录每一行的最小值 下标即为行号

int maxOfCol[n];//记录每一列的最大值 下标即为列号

int tempMin;

int placeOfTempMin;//记录临时最小值的列

int tempMax;

int placeOfTempMax;//记录临时最大值的行

//查找每一行的最小值

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

{

tempMin=matrix[i][0];

placeOfTempMin=0;

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

{

if(matrix[i][j]<tempMin)

{

tempMin=matrix[i][j];

placeOfTempMin=j;

}

}

//每行查找完后 保存当前行最小值的列;

minOfRow[i]=placeOfTempMin;

}

//查找每一列的最大值

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

{

tempMax=matrix[0][i];

placeOfTempMax=0;

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

{

if(matrix[j][i]>tempMax)

{

tempMax=matrix[j][i];

placeOfTempMax=j;//记录最大值所在的行

}

}

//每列查找完后 保存当前列最大值的行;

maxOfCol[i]=placeOfTempMax;

}

//查找完成 下进行比对

//以行为基础遍历 检查每一行最小值所在的列

//然后查看这一列最大值所在的行是否与最小值的行号相同

int count=0;//计数器 统计马鞍点个数

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

{

if(maxOfCol[minOfRow[i]]==i)

{

count++;

cout<<"找到第"<<count<<"个马鞍点索引为:"<<endl;

cout<<i<<" "<<minOfRow[i]<<endl;

}

}

if(count==0)

{

cout<<"未找到马鞍点"<<endl;

}

return 0;

}

3.题目：

问题描述：

一次放学的时候，小明已经规划好了自己回家的路线，并且能够预测经过各个路段的时间。同时，小明通过学校里安装的“智慧光明”终端，看到了出发时刻路上经过的所有红绿灯的指示状态。请帮忙计算小明此次回家所需要的时间。

输入格式：

输入的第一行包含空格分隔的三个正整数 r、y、g，表示红绿灯的设置。这三个数均不超过 106。输入的第二行包含一个正整数 n，表示小明总共经过的道路段数和路过的红绿灯数目。接下来的 n 行，每行包含空格分隔的两个整数 k、t。k=0 表示经过了一段道路，将会耗时 t 秒，此处 t 不超过 106；k=1、2、3 时，分别表示出发时刻，此处的红绿灯状态是红灯、黄灯、绿灯，且倒计时显示牌上显示的数字是 t，此处 t 分别不会超过 r、y、g。

算法：核心是检查当前是什么灯。具体做法是一个周期是红 绿 黄 先找到开始时对应周期的起始时间（红灯开始的时间）然后根据周期对当前是什么灯进行判断

源代码：

#include<iostream>

#include<vector>

using namespace std;

//检查当前是什么灯的函数 返回需要等待的时间

//算法思路：一个周期是红 绿 黄 先找到开始时对应周期的起始时间（红灯开始的时间）

//然后根据周期对当前是什么灯进行判断

int checkWhatLight(pair<int,int> &currentLight,int currentTime,int r,int y,int g)

{

int waitTime=0;

int pastTime=0;//到周期开始的时间

int cycleTime=0;//加上currentTime后整体循环的总时间

if(currentLight.first==1)//红灯

{

pastTime=r-currentLight.second;

cycleTime=pastTime+currentTime;

}

if(currentLight.first==2)//黄灯

{

pastTime=r+g+y-currentLight.second;

cycleTime=pastTime+currentTime;

}

if(currentLight.first==3)//绿灯

{

pastTime=r+g-currentLight.second;

cycleTime=pastTime+currentTime;

}

int overTime=cycleTime%(r+y+g);//超出循环的时间

if(overTime>=0&&overTime<r)//在红灯

{

waitTime=r-overTime;//等完红灯即可通行

}

if(overTime>=r&&overTime<r+g)//在绿灯

{

waitTime=0;//直接通行

}

if(overTime>=r+g&&overTime<r+g+y)//在黄灯

{

waitTime=y-(overTime-r-g)+r;//等完黄灯还需要等一个红灯

}

return waitTime;

}

int main()

{

int r,y,g;

cin>>r>>y>>g;//三种灯的持续时间

int n;

cin>>n;//小明将会经过的路和灯的总数量

vector<pair<int,int>> roadsAndLights;

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

{

int k;//状态指示量

int t;//通过时间或出发时各种灯的剩余时间

cin>>k>>t;

pair<int,int> p=make\_pair(k,t);

roadsAndLights.push\_back(p);

}

int totalTime=0;

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

{

if(roadsAndLights[i].first==0)

{

totalTime+=roadsAndLights[i].second;

}

else

{

int waitTime=0;

waitTime=checkWhatLight(roadsAndLights[i],totalTime,r,y,g);

totalTime+=waitTime;

}

}

cout<<totalTime<<endl;

return 0;

}

4.题目：西西艾弗岛的购物中心里店铺林立，商品琳琅满目。为了帮助游客根据自己的预算快速选择心仪的商品，IT 部门决定研发一套商品检索系统，支持对任意给定的预算x，查询在该预算范围内（≤x）价格最高的商品。如果没有商品符合该预算要求，便向游客推荐可以免费领取的西西艾弗岛定制纪念品。

假设购物中心里有n件商品，价格从低到高依次为A1,A2,…,An，则根据预算x检索商品的过程可以抽象为如下序列查询问题。

A=[A0,A1,A2,…,An]是一个由n+1个[0,N)范围内整数组成的序列，满足0= A0＜A1＜A2＜…＜An＜N。（这个定义中蕴含了n一定小于N。）

基于序列A，对于[0,N)范围内任意的整数x，查询f(x)定义为：序列A中小于等于x的整数里最大的数的下标。具体来说有以下两种情况：

(1)存在下标0≤i＜n满足Ai≤x＜Ai+1

此时序列A中从A0到Ai均小于等于x，其中最大的数为Ai，其下标为i，故f(x)=i。

(2) An≤x

此时序列A中所有的数都小于等于x，其中最大的数为An，故f(x)=n。

令sum(A)表示f(0)到f(N-1)的总和，即：



对于给定的序列A，试计算sum(A)。

算法：核心是要降低算法的时间复杂度。灵活运用乘法：对任意的A[i]，最终结果中需要加入i\*A[i+1]-A[i]，而最后一项乘的次数是N-A[n]

源代码：

#include<iostream>

#include<vector>

using namespace std;

//创建

vector<pair<int,int>> CreateSubArray(int \*A,int N,int n)

{

vector<pair<int,int>> subArray;

for(int i=0;i<n;i++)//由于要使用后一项 所以只需循环到n-1

{

//创建对组 第一个值存储纳入值的下标，第二个值存储计算区间

pair<int,int> p;

int indexOfPriceToAdd=i;

int timesToAdd=A[i+1]-A[i];

p=make\_pair(indexOfPriceToAdd,timesToAdd);

subArray.push\_back(p);

}

//创建完前面的之后 最后一项要根据N来进行创建

pair<int,int> p;

int lastIndexOfPriceToAdd=n;

int lastTimesToAdd=N-A[n];

p=make\_pair(lastIndexOfPriceToAdd,lastTimesToAdd);

subArray.push\_back(p);

return subArray;

}

int getSum(int \*A,int N,int n)

{

int sum=0;

vector<pair<int,int>> subArray;

subArray=CreateSubArray(A, N, n);

for(int i=0;i<subArray.size();i++)

{

sum+=(subArray[i].first)\*(subArray[i].second);

}

return sum;

}

int main()

{

int n,N;//n表示商品数量 N表示商品价格上界（开区间）

cin>>n>>N;

int A[n+1];

A[0]=0;

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

{

cin>>A[i];

}

cout<<getSum(A,N,n)<<endl;

return 0;

}