## 向量（vector）

vector是表示可变大小数组的序列容器。

1. 头文件

#include<vector>

2、vector声明及初始化

vector<int> vec; 声明一个int型向量

vector<int> vec(5); 声明一个初始大小为5的int向量

vector<int> vec(10, 1);声明一个初始大小为10且值都是1的向量

vector<int> vec(tmp); 声明并用tmp向量初始化vec向量

vector<int> tmp(vec.begin(),vec.begin()+3);用向量vec的第0个到第2个值初始tmp

int arr[5] = {1, 2, 3, 4, 5};

vector<int> vec(arr, arr + 5); 将arr数组的元素用于初始化vec向量

3、vector基本操作

(1). 容量

向量大小： vec.size();

向量最大容量： vec.max\_size();

更改向量大小： vec.resize();

向量判空： vec.empty();

减少向量大小到满足元素所占存储空间的大小： vec.shrink\_to\_fit(); //shrink\_to\_fit

(2). 修改

多个元素赋值： vec.assign(); 类似于初始化时用数组进行赋值

末尾添加元素： vec.push\_back();

末尾删除元素： vec.pop\_back();

任意位置插入元素:vec.insert(vec.begin()+i,x);第i+1个元素前面插入x

任意位置删除元素： vec.erase(vec.begin()+i);删除第i+1个元素

任意区间删除:vec.erase(vec.begin()+i,vec.begin()+j);区间[i，j-1]

交换两个向量的元素： vec.swap();

清空向量元素： vec.clear();

(3)迭代器

vector<int>::iterator it;

开始指针：vec.begin();

末尾指针：vec.end(); //指向最后一个元素的下一个位置

(4)元素的访问

下标访问： vec[1]; //并不会检查是否越界

访问第一个元素： vec.front();

访问最后一个元素： vec.back();

返回一个指针： int\* p = vec.data();

（5）算法

遍历元素

vector<int>::iterator it;

for (it = vec.begin(); it != vec.end(); it++)

cout << \*it << endl;

元素翻转

#include<algorithm>

reverse(vec.begin(), vec.end());

元素排序

sort(vec.begin(), vec.end()); 默认从小到大

自定义优先级

bool cmp(const int& a, const int& b) { return a > b; }

sort(vec.begin(), vec.end(), cmp);

(6) 特别注意

vector存放结构体类型时，结构体要定义为全局变量 结构体内部可定义优先级

## 列表（list）

list是由节点组成的双向链表，每个结点包含着一个元素

1. 头文件

#include<list>

1. list的声明及初始化

list<int>a{1,2,3}

list<int>a(n) 声明一个n个元素的列表，每个元素都是0

list<int>a(n, m) 声明一个n个元素的列表，每个元素都是m

list<int>a(first, last) 声明一个列表，其元素的初始值来源于由区间所指定的序列中的元素，first和last是迭代器

3、list基本操作

begin()和end()

通过调用list容器的成员函数begin()得到一个指向容器起始位置的iterator，可以调用list容器的end()函数来得到list末端下一位置

push\_back()和push\_front()

使用list的成员函数push\_back和push\_front插入一个元素到list中。其中push\_back()是从list的末端插入，而push\_front()是从list的头部插入。

empty() 判断list是否为空

resize()

调用resize(n)将list的长度改为只容纳n个元素，超出的元素将被删除。如果n比list原来的长度长，那么默认超出的部分元素置为0。也可以用resize(n, m)的方式将超出的部分赋值为m。

list<int>b{1, 2, 3, 4};

b.resize(2);

list中输出元素：1,2

list<int>b{1, 2, 3, 4};

b.resize(6);

list中输出元素：1,2,3,4,0,0

list<int>b{1, 2, 3, 4};

b.resize(6,9);

list中输出元素：1,2,3,4,9,9

**clear()** 清空list中的所有元素

front()和back()

通过front()可以获得list容器中的头部元素，通过back()可以获得list容器的最后一个元素。注意：当list元素为空时，这时候调用front()和back()不会报错。因此在编写程序时，最好先调用empty()函数判断list是否为空，再调用front()和back()函数。

**pop\_back()和pop\_front()**

使用pop\_back()可以删掉尾部第一个元素，pop\_front()可以删掉头部第一个元素。注意：list必须不为空，如果当list为空的时候调用pop\_back()和pop\_front()会使程序崩掉。

**assign()**

有两种使用情况：

（1）a.assign(n, val):将a中的所有元素替换成n个val元素

list<int>b{1,2,3,4,5};

b.assign(5,10);

b中的元素变为10, 10, 10, 10, 10

（2）a.assign(b.begin(), b.end())

list<int>a{6,7,8,9};

list<int>b{1,2,3,4,5};

b.assign(a.begin(),a.end());

b中的元素变为6,7,8,9

**swap()**

交换两个链表。a.swap(b)和swap(a, b)，都可以完成a链表和b链表的交换。

**reverse()**

可以实现list的逆置

**merge()**

a.merge(b) 调用结束后b变为空，a中元素包含原来a和b的元素。

list<int>a{6,7,8,9};

list<int>b{2, 1, 3, 6, 5};

merge(b,greater<int>());

a中元素变为：6,7,8,9,2,1,3,6,5

**insert()**

在指定位置插入一个或多个元素

a.insert(a.begin(),100); //在a的开始位置（即头部）插入100

a.insert(a.begin(),2, 100); //在a的开始位置插入2个100

a.insert(a.begin(),b.begin(), b.end());//在a的开始位置插入b从开始到结束的所有位置的元素

erase() 删除一个元素或一个区域的元素

a.erase(a.begin());//将a的第一个元素删除

a.erase(a.begin(),a.end());//从begin()到end()之间的元素删除。

**remove()函数**

从list中删除元素

list<int>a{6,7,8,9,7,10};

remove(7);

删除了a中所有值为7的元素，此时a中元素为6,8,9,10

## 双端队列（deque）

deque是双端队列

1. 头文件

#include<deque>

1. deque的声明及初始化

deque<type> deq; 声明一个元素类型为type的双端队列que

deque<type> deq(size); 声明一个类型为type、含有size个默认值初始化元素的的双端队列

deque<type> deq(size, value); 声明一个元素类型为type、含有size个value元素的双端队列

deque<type> deq(mydeque); deq是mydeque的一个副本

deque<type> deq(first, last); 使用迭代器first、last范围内的元素初始化deq

1. deque基本操作

deq[ ]：用来访问双向队列中单个的元素。

deq.front()：返回第一个元素的引用。

deq.insert(地址,值);插入到某个地址某个值

deq.erase(地址);删除某个地址的值

deq.back()：返回最后一个元素的引用。

deq.push\_front(x)：把元素x插入到双向队列的头部。

deq.pop\_front()：弹出双向队列的第一个元素。

deq.push\_back(x)：把元素x插入到双向队列的尾部。

deq.pop\_back()：弹出双向队列的最后一个元素。

## 集合（set）

set是一种随机存储的关联式容器，其关键词（key）和元素（value）是同一个值。set之中所有元素互不相同。

1. 头文件

#include<set>

1. set的声明及初始化

set<int> s 创建一个空的set

set<int,greater<int>> s 创建一个带大于比较器的set, 默认是小于比较器less<int>

int a[3] = {1, 2, 3} ;

set<int> s(a, a + 3) ; 用数组初始化一个set

set<int> s1 ;

set<int> s2(s1) ; 用拷贝构造函数初始化set

set<int> s1 ;

set<int> s2(s1.begin(), s1.end()) ; 区间初始化

自定义比较函数

以类为比较器

struct classcmp{

bool operator()(const int& lhs, const int& rhs){

return lhs < rhs ;

}

};

int main(void){

set<int, classcmp> s5 ;

system("pause") ;

return 0 ;

}

以函数指针为比较器

bool fncmp(int lhs, int rhs){

return lhs < rhs ;

}

int main(void){

bool(\*fn\_pt)(int, int) = fncmp ;

set<int, bool(\*)(int, int)> s1(fn\_pt) ;

system("pause") ;

return 0 ;

}

1. set的方法

begin() 返回set容器的第一个元素

end() 　返回set容器的最后一个元素

clear() 删除set容器中的所有的元素

empty() 　判断set容器是否为空

max\_size() 返回set容器可能包含的元素最大个数

size() 返回当前set容器中的元素个数

count() 用来查找set中某个某个键值出现的次数，这里返回值只有0 1。

erase(iterator) 删除定位器iterator指向的值

erase(first,second) 删除定位器first和second之间的值

erase(key\_value) 删除键值key\_value的值

find() 返回给定值值得定位器，如果没找到则返回end()。

insert(key\_value) 将key\_value插入到set中

inset(first,second)将定位器first到second之间的元素插入到set中，返回值是void.

rbegin 返回的值和end()相同

rend() 返回的值和rbegin()相同

lower\_bound(key\_value) 返回第一个大于等于key\_value的定位器

upper\_bound(key\_value)返回最后一个大于等于key\_value的定位器

## 多重集合（multiset）

set和multiset会根据特定的排序准则，自动将元素进行排序。不同的是后者允许元素重复而前者不允许。

multiset的定义、初始化以及基本操作和set都基本相同 要特别注意一下删除操作

a.erase(x);//删除集合中所有的x

multiset<int>::iterator it = a.find(x);

if(it!=a.end()){

a.erase(it);//这里是删除其中的一个x; 删除的是一个位置

}

## 栈（stack）

stack就是栈

1. 头文件

#include<stack>

1. stack的声明及初始化

stack<int> s;

1. 基本操作

empty() 堆栈为空则返回真

pop() 移除栈顶元素

push() 在栈顶增加元素

size() 返回栈中元素数目

top() 返回栈顶元素

## 队列（queue）

## 优先队列（priority\_queue）

priority\_queue是优先队列

1. priority\_queue的声明及初始化

普通声明

priority\_queue<int> Q; 通过操作，按照元素从大到小的顺序出队

自定义优先级

struct cmp{

operator bool ()(int x, int y){

return　x > y; // x小的优先级高

//也可以写成其他方式，如： return p[x] > p[y];表示p[i]小的优先级高

}

};

priority\_queue<int, vector<int>, cmp> Q;//定义方法

//其中，第二个参数为容器类型。第三个参数为比较函数。

结构体声明方式

struct node {

int x, y;

friend bool operator < (node a, node b) {

return a.x > b.x; //结构体中，x小的优先级高

}

};

priority\_queue<node> Q;//定义方法

//在该结构中，y为值, x为优先级。

//通过自定义operator<操作符来比较元素中的优先级。

//在重载”<”时，最好不要重载”>”，可能会发生编译错误

## 映射（map）

1. 基本操作

begin() 返回指向map头部的迭代器

clear() 删除所有元素

count() 返回指定元素出现的次数

empty() 如果map为空则返回true

end() 返回指向map末尾的迭代器

equal\_range() 返回特殊条目的迭代器对

erase() 删除一个元素

find() 查找一个元素

get\_allocator() 返回map的配置器

insert() 插入元素

key\_comp() 返回比较元素key的函数

lower\_bound() 返回键值>=给定元素的第一个位置

max\_size() 返回可以容纳的最大元素个数

rbegin() 返回一个指向map尾部的逆向迭代器

rend() 返回一个指向map头部的逆向迭代器

size() 返回map中元素的个数

swap() 交换两个map

upper\_bound() 返回键值>给定元素的第一个位置

value\_comp() 返回比较元素value的函数

不排序的map(unordered\_map)

多重映射（multimap）

## pair

pair将一对值组合成一个值，这一对值可以具有不同的数据类型（T1和T2），两个值可以分别用pair的两个公有函数first和second访问。

1. 头文件

#include<utility>

2、pair的声明及初始化

pair<int, double> p1; //使用默认构造函数

pair <int ,pair<int ,int > > //也可以这样

pair<int, double> p2(1, 2.4); //用给定值初始化

pair<int, double> p3(p2); //拷贝构造函数

3、基本操作

访问两个元素（通过first和second）

pair<int, double> p1; //使用默认构造函数

p1.first = 1;

p1.second = 2.5;

cout << p1.first << ' ' << p1.second << endl;

赋值operator =

（1）利用make\_pair

pair<int, double> p1;

p1 = make\_pair(1, 1.2);

（2）变量间赋值

pair<int, double> p1(1, 1.2);

pair<int, double> p2 = p1;

## 快速全排列

[STL快速全排列函数next\_permutation](http://blog.csdn.net/winter2121/article/details/72581936)()和pre\_permutation()

对于一个大小为n的数组，调用next\_permutation（a，a+n）可以得到数组a的下一个排列 调用pre\_permutation(a，a+n)可以得到上一个全排列

### 01背包

//时间复杂度O(n \* V) 空间复杂度O(n \* V)

//w[] 物品重量 v[] 物品价值 V背包最大容量

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

for (int j = 0; j <= V; j++) {

if (w[i] <= j) {

dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i]);

} else {

dp[i][j] = dp[i - 1][j];

}

}

}

//时间复杂度O(n \* V) 空间复杂度O(n)

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

for (int j = V; j >= w[i]; j--) {

dp[j] = max(dp[j], dp[j - w[i]] + v[i]);

}

}

//01背包记录路径

//path [ i ] [ j ] 表示物品 i 放在了状态 j 的背包中 空间 O(n \* V)

memset(dp, 0,sizeof(dp));

memset(path, 0,sizeof(path));

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

for (int j = V; j >= w[i]; j--) {

dp[j] = max(dp[j], dp[j - w[i]] + v[i]);

path[i][j] = 1;// 把装进去的物品标记一下

}

}

cout << dp[V] << "\n";

int i = n, j = V;// V：背包容量n个物品

while (i >= 1 && j >= 1) {

if (path[i][j]) {// 物品i在j里

cout << i << " ";

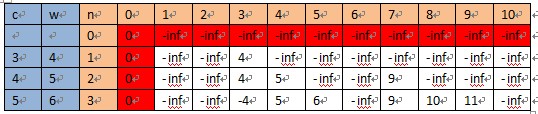
j -= w[i];// 读完了物品i，找下一个背包状态

}

i--;

}

题目要求背包恰好装满时， 只需要把dp数组初始值赋值为-INF，大于0的值为恰好装满的最大价值



### 最长公共子序列

//时间复杂度 O(n\*m)

char a[1100],b[1100];// 字符串a,b

int len[1100][1100];

int count1[1100][1100];

void Print(int i,int j,char x[],int count1[1100][1100]){// 输出最长公共子序列

if (i==0||j==0) return ;

if (count1[i][j]==0){

Print(i-1,j-1,x,count1);

printf ("%c",x[i-1]);

}

else if(count1[i][j]==1) Print(i-1,j,x,count1);

else Print(i,j-1,x,count1);

}

void LCS(){

mset(len,0);

int n=strlen(a),m=strlen(b);

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

for (int j=1;j<=m;j++){

if (a[i-1]==b[j-1]){

len[i][j]=len[i-1][j-1]+1;

count1[i][j]=0;

} else {

len[i][j]=max(len[i-1][j],len[i][j-1]);

if(len[i-1][j]>len[i][j-1]) count1[i][j]=1;

else count1[i][j]=-1;

}

}

}

}

int main (){

while (scanf ("%s %s",a,b)!=EOF){

LCS();

printf ("%d\n",len[strlen(a)][strlen(b)]);

Print(strlen(a),strlen(b),a,count1);

putchar('\n');

}

return 0;

}

### 最长递增子序列

//时间复杂度O(n \* log(n))

b[0] = s[0];

int len = 1, Index;

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

if (s[i] > b[len - 1]){

b[len] = s[i], len++;

} else{

Index = lower\_bound(b, b + len - 1, s[i]) - b;

b[Index] = s[i];

}

}

cout << len << "\n";

### Dijkstra

图中存在负环的情况，Dijkstra算法不可直接用

普通Dijkstra算法

const int maxn=INF;

int dis[110];//已求得的最短距离

int Map[110][110];//用于记录各个点之间的距离

void Dijkstra(int v0,int n){

int vis[110];//标记已经求得最短距离的点

int u;

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

dis[i]=Map[v0][i];//将各点到源点得距离初始化 有边即为距离 无边为 maxn

vis[i]=0;//将所有得点初始化为未求得最短距离

}

dis[v0]=0; //源点到源点得距离为 0

vis[v0]=1;//标记源点被计算过

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

int mindis=maxn;

for (int j=1;j<=n;j++){

if (!vis[j]&&dis[j]<mindis){// 找到未被计算过并且距离源点距离最短的点

u=j;

mindis=dis[j];

}

}

vis[u]=1;//将该点标记为已被计算

for (int j=1;j<=n;j++){

if (!vis[j]&&Map[u][j]+dis[u]<dis[j]){//从u的基础上 找到一个未被访问 距离源点最近的点

dis[j]=dis[u]+Map[u][j];//保存该点最短距离

}

}

}

}

int main (){

int N,M;

int a,b,c;

while (scanf ("%d%d",&N,&M)&&(N||M)){

for (int i=1;i<=N;i++){

for (int j=1;j<=N;j++){

Map[i][j]=maxn;//将各个点之间的距离初始化为 无穷大（表示为 maxn）

}

}

for (int i=1;i<=M;i++){

scanf ("%d%d%d",&a,&b,&c);

//为两点之间付距离

Map[a][b]=c;

Map[b][a]=c;

}

Dijkstra(1,N);

printf ("%d\n",dis[N]);

}

return 0;

}

优先队列优化的Dijkstra

#define mset(a,b) memset(a,b,sizeof(a))

int n,m;

int cnt=0;

int head[1000005];

int dis[1000005],vis[1000005];

struct node{

int s,w;

bool operator < (const node& u)const{ return w>u.w;}

};

struct Edge{

int to,next,w;

}edge[1000005];

inline void add(int u,int v,int w){

edge[cnt].to=v;

edge[cnt].w=w;

edge[cnt].next=head[u];

head[u]=cnt++;

}

void init(){

cnt=0;

mset(head,-1);

}

void Dijkstra(int s){

mset(dis,INF);mset(vis,0);

priority\_queue<node> Q;

dis[s]=0;

Q.push((node){s,0});

while(!Q.empty()){

node temp=Q.top();

Q.pop();

if(vis[temp.s])continue;

vis[temp.s]=1;

for(int i=head[temp.s];i!=-1;i=edge[i].next){

if(dis[edge[i].to]>dis[temp.s]+edge[i].w){

dis[edge[i].to]=dis[temp.s]+edge[i].w;

Q.push((node){edge[i].to,dis[edge[i].to]});

}

}

}

}

int main(){

int i,j,k;

int s;

while (scanf("%d%d",&n,&m)&&n&&m){

init();

int u,v,w;

for(i=1;i<=m;i++){

scanf("%d%d%d",&u,&v,&w);

add(u,v,w); add(v,u,w);

}

Dijkstra(1);

printf ("%d\n",dis[n]);

}

return 0;

}

### Floyd

图中存在负环的情况，Floyd算法不可直接用

int n,m;

int Map[110][110];//邻接矩阵存储数据

int dis[110][110];//用于记录最短路

void floyd(){

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

for (int j=1;j<=n;j++){

dis[i][j]=Map[i][j];

}

}

for (int k=1;k<=n;k++){

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

for (int j=1;j<=n;j++){

if (dis[i][j]>dis[i][k]+dis[k][j]){//在点 i 和 j 之间插入点 k 取之前从i到j的距离 和 插入点k后经过k点 的i到j的距离 小的那个

dis[i][j]=dis[i][k]+dis[k][j];

}

}

}

}

}

int main (){

while (scanf ("%d%d",&n,&m)&&(n||m)){

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

for (int j=1;j<=n;j++){

if (i!=j) Map[i][j]=INF;

else Map[i][j]=0;//本身到本身为 0

}

}

int u,v,w;

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

scanf ("%d%d%d",&u,&v,&w);

Map[u][v]=w; Map[v][u]=w;

}

floyd();

printf ("%d\n",dis[1][n]);

}

return 0;

}

## 最小生成树

### Prim

int map[1010][1100];

int dis[1100];//记录实时路长

int vis[1100];//标记访问

int n,m;

void prim(ints) //ans=dis之和

{

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

{

dis[i]=map[s][i];

vis[i]=0;

}

vis[s]=1;

for(inti=2;i<=n;i++)

{

intmin=INF;

intk=0;

for(intj=1;j<=n;j++)

{

if(!vis[j]&&min>dis[j])//找一条以j结尾的未连线的最短线段

{

min=dis[j];

k=j;

}

}

vis[k]=1;

for(intj=1;j<=n;j++)

if(!vis[j]&&dis[j]>map[k][j])//k到j的距离要更新

dis[j]=map[k][j];

}

}

### Kruskal

#include<bits/stdc++.h>

using namespace std;

const int maxn = 100005;

struct node{int s,y,w;} edge[maxn];

int pre[maxn],n,m, sum=0, cnt=0;

void init(){ for(int i=0;i<=n;i++) pre[i]=i;}

int Find(int x){

int r = x;

while(pre[r] != r) r=pre[r];

int i = x, j;

while(i!=r){

j=pre[i], pre[i]=r; i = j;

}

return r;

}

void mix(int x,int y) {

int fx=Find(x),f y=Find(y);

if(fx != fy) pre[fx]=fy;

}

int cmp(node a,node b){ return a.w<b.w; }

void kruskal(){

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

if(cnt==n-1) break;

if(Find(edge[i].s)!=Find(edge[i].y)){

mix(edge[i].s,edge[i].y);

sum+=edge[i].w; cnt++;

}

}

}

int main() {

scanf("%d%d",&n,&m);

init();

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

scanf("%d%d%d",&edge[i].s,&edge[i].y,&edge[i].w);

sort(edge,edge+m,cmp);

kruskal();

if(cnt!=n-1) printf("?\n");

else printf("%d\n",sum);

return 0;

}

## 拓扑排序

#include<iostream>

#include<queue>

#include<cstring>

#include<cstdio>

#include<vector>

using namespace std;

typedef long long ll;

const int INF=0x3f3f3f3f;

struct Edge{

int next;

int to;

int w;

}edge[25005];

int head[25005],cnt=0;

void add(int u,int v){

edge[cnt].to=v;

edge[cnt].next=head[u];

head[u]=cnt++;

}

int sum[200005],a[200005],n,m,s;

void topo(){

queue<int>q;

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

if(sum[i]==0)

q.push(i);

while(!q.empty()){

int t=q.front();

q.pop();

a[s++]=t;

for(int i=head[t];~i;i=edge[i].next){

sum[edge[i].to]--;

if(sum[edge[i].to]==0)

q.push(edge[i].to);

}

}

}

/\*

void topo(){

priority\_queue<int,vector<int>,greater<int> >q;//字典序最小输出

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

if(sum[i]==0)

q.push(i);

while(!q.empty()){

int t=q.top();

q.pop();

a[s++]=t;

for(int i=head[t];~i;i=edge[i].next){

sum[edge[i].to]--;

if(sum[edge[i].to]==0)

q.push(edge[i].to);

}

}

}\*/

int main()

{

memset(sum,0,sizeof(sum));

memset(head,-1,sizeof(head));

scanf("%d%d",&n,&m);

int t,x,y;

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

scanf("%d%d",&x,&y);

add(x,y);

sum[y]++;

}

s=0;

topo();

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

printf("%d ",a[i]);

puts("");

return 0;

}

### 素数打表

int prime[maxn], vis[maxn];

void init1(){

int cnt = 0;

memset(vis,0,sizeof(vis));

vis[0] = vis[1] = 1;

for(int i = 2; i < maxn; i++){

if(!vis[i]) prime[cnt++] = i;

for(int j = 0; j < cnt && i \* prime[j] < maxn; j++){

vis[i\*prime[j]] = 1;

if(i % prime[j] == 0) break;

}

}

}