HHU ACM Template

Writter **Luo Longjun, Tang ShengYang**

# -------Datastructure-------

## 1--BIT

#include <iostream>  
#include <cstdio>  
#include <cstring>  
using namespace std;  
  
int a[1010];  
int c[1010];  
int n;  
inline int lowbit(int x) {  
 return x & -x;  
}  
int sum(int x){  
 return x ? c[x]+ sum( x - lowbit(x)):0;  
}  
/\*  
int getx(int x)  
{  
 int ans=0;  
 while(x>0)  
 {  
 ans+=c[x];  
 x-=lowbit(x);  
 }  
 return ans;  
}  
\*/  
void add(int x,int v){  
 for(int i = x; i <= n; i += lowbit(i)){  
 c[i] += v;  
 }  
}  
/\*  
void add(int x,int v){  
 if(x <= n){  
 C[x]+= v, add( x + lowbit(i), v );  
 }  
}  
\*/  
int main() {  
 cin>>n;  
 memset(c,0,sizeof(c));  
 for(int i = 1;i<=n;i++){  
 cin>>a[i];  
 add(i,a[i]);  
 }  
 for(int i = 1;i<=n;i++)  
 cout<<sum(i)<<endl;  
 return 0;  
}

## 2--heap

#include <iostream>  
using namespace std;  
  
#define MAXN 10000  
#define \_cp(a,b) ((a)<(b))  
typedef int elem\_t;  
elem\_t h[MAXN];  
int n;  
void init()  
{  
 n=0;  
}  
void exchange(int a,int b){  
 int t = h[a];  
 h[a]=h[b];  
 h[b]=t;  
}  
//由下到上的有序化  
void swim(int k){  
 while( k > 1 && \_cp(h[k/2],h[k]) ){  
 exchange(k/2,k);  
 k=k/2;  
 }  
}  
//由上到下的有序化  
void sink(int k){  
 while(2\*k<=n){  
 int j = 2\*k;  
 if( j < n && \_cp(h[j],h[j+1])) j++;  
 if(!\_cp(h[k],h[j])) break;  
 exchange(k,j);  
 k=j;  
 }  
}  
  
void ins(elem\_t e){  
 h[++n] = e;  
 swim(n);  
}  
  
elem\_t delmax(){  
 elem\_t ma = h[1];  
 exchange(1,n--);  
 h[n+1]= 0 ;  
 sink(1);  
 return ma;  
}  
  
int del(elem\_t& e){  
 if (!n)  
 return 0;  
 int p,c;  
 for (e=h[p=1] , c=2 ; c<n && \_cp( h[c+=(c<n-1&&\_cp(h[c+1],h[c]))] , h[n]) ; h[p]=h[c] ,p=c , c<<=1) ;  
 h[p]=h[n--];return 1;  
}  
int main()  
{  
 int asd;  
 cin>>asd;  
 for(int i = 0;i<asd;i++)  
 {  
 int qq ;  
 cin>>qq;  
 ins(qq);  
 }  
 for(int i = 1;i<=n;i++)  
 cout<<" "<<h[i];  
 cout<<endl;  
 return 0;  
}

## 3--RMQ

#include<cstdio>  
#include<cstring>  
#include<algorithm>  
using namespace std;  
const int MAXN=50000+100;  
/\*  
 以nlogn的速度初始化，以1的速度查找某区间最大最小值  
\*/  
int dmax[MAXN][20];  
int dmin[MAXN][20];  
  
void initmax(int n,int d[])//初始化最大值查询  
{  
 for(int i=1; i<=n; i++)  
 dmax[i][0]=d[i];  
 for(int j=1 ; (1<<j)<=n ; j++)  
 for(int i=1; i+(1<<j)-1 <=n; i++)  
 dmax[i][j]=max(dmax[i][j-1],dmax[i+(1<<(j-1))][j-1]);  
}  
int getmax(int L,int R)//查询最大值  
{  
 int k=0;  
 while((1<<(k+1))<=R-L+1)k++;  
 return max(dmax[L][k] , dmax[R-(1<<k)+1][k]);  
}  
  
void initmin(int n,int d[])//初始化最小值查询  
{  
{  
 for(int i=1; i<=n; i++)  
 dmin[i][0]=d[i];  
 for(int j=1; (1<<j)<=n; j++)  
 for(int i=1;i+(1<<j)-1<=n;i++)  
 dmin[i][j]= min( dmin[i][j-1],dmin[i+(1<<(j-1))][j-1] );  
}  
int getmin(int L,int R)//查询最小值  
{  
 int k=0;  
 while( (1<<(k+1)) <=R-L+1)k++;  
 return min(dmin[L][k],dmin[R-(1<<k)+1][k]);  
}

## 4--SegTree

#include <iostream>  
#include <cstdio>  
using namespace std;  
//线段树，求最小  
const int INFINITE = 0x7fffffff;  
const int MAXNUM = 1000;  
struct SegTreeNode  
{  
 int val;  
 int addMark;//延迟标记  
}segTree[MAXNUM];//定义线段树  
  
/\*  
功能：构建线段树  
root：当前线段树的根节点下标  
arr: 用来构造线段树的数组  
istart：数组的起始位置  
iend：数组的结束位置  
\*/  
void build(int root, int arr[], int istart, int iend)  
{  
 segTree[root].addMark = 0;//----设置标延迟记域  
 if(istart == iend)//叶子节点  
 segTree[root].val = arr[istart];  
 else  
 {  
 int mid = (istart + iend) / 2;  
 build(root\*2+1, arr, istart, mid);//递归构造左子树  
 build(root\*2+2, arr, mid+1, iend);//递归构造右子树  
 //根据左右子树根节点的值，更新当前根节点的值  
 segTree[root].val = min(segTree[root\*2+1].val, segTree[root\*2+2].val);  
 }  
}  
  
/\*  
功能：当前节点的标志域向孩子节点传递  
root: 当前线段树的根节点下标  
addMark是因为更新的时候并非直接更新所有的，而是标记了一下说他更新了。查询的时候再更新。  
\*/  
void pushDown(int root)  
{  
 if(segTree[root].addMark != 0)  
 {  
 //设置左右孩子节点的标志域，因为孩子节点可能被多次延迟标记又没有向下传递  
 //所以是 “+=”  
 segTree[root\*2+1].addMark += segTree[root].addMark;  
 segTree[root\*2+2].addMark += segTree[root].addMark;  
 //根据标志域设置孩子节点的值。因为我们是求区间最小值，因此当区间内每个元  
 //素加上一个值时，区间的最小值也加上这个值  
 segTree[root\*2+1].val += segTree[root].addMark;  
 segTree[root\*2+2].val += segTree[root].addMark;  
 //传递后，当前节点标记域清空  
 segTree[root].addMark = 0;  
 }  
}  
  
/\*  
功能：线段树的区间查询  
root：当前线段树的根节点下标  
[nstart, nend]: 当前节点所表示的区间  
[qstart, qend]: 此次查询的区间  
\*/  
int query(int root, int nstart, int nend, int qstart, int qend)  
{  
 //查询区间和当前节点区间没有交集  
 if(qstart > nend || qend < nstart)  
 return INFINITE;  
 //当前节点区间包含在查询区间内  
 if(qstart <= nstart && qend >= nend)  
 return segTree[root].val;  
 //分别从左右子树查询，返回两者查询结果的较小值  
 pushDown(root); //----延迟标志域向下传递  
 int mid = (nstart + nend) / 2;  
 return min(query(root\*2+1, nstart, mid, qstart, qend),  
 query(root\*2+2, mid + 1, nend, qstart, qend));  
  
}  
  
/\*  
功能：更新线段树中某个区间内叶子节点的值  
root：当前线段树的根节点下标  
[nstart, nend]: 当前节点所表示的区间  
[ustart, uend]: 待更新的区间  
addVal: 更新的值（原来的值加上addVal）  
\*/  
void update(int root, int nstart, int nend, int ustart, int uend, int addVal)  
{  
 //更新区间和当前节点区间没有交集  
 if(ustart > nend || uend < nstart)  
 return ;  
 //当前节点区间包含在更新区间内  
 if(ustart <= nstart && uend >= nend)  
 {  
 segTree[root].addMark += addVal;  
 segTree[root].val += addVal;  
 return ;  
 }  
 pushDown(root); //延迟标记向下传递  
 //更新左右孩子节点  
 int mid = (nstart + nend) / 2;  
 update(root\*2+1, nstart, mid, ustart, uend, addVal);  
 update(root\*2+2, mid+1, nend, ustart, uend, addVal);  
 //根据左右子树的值回溯更新当前节点的值  
 segTree[root].val = min(segTree[root\*2+1].val, segTree[root\*2+2].val);  
}  
  
int main(){  
 int n;  
 int a[1010];  
 cin>>n;  
 for(int i = 0;i<n;i++)  
 {  
 cin>>a[i];  
 }  
 build(0,a,0,n-1);  
 cout<<query(0,0,n-1,2,4)<<endl;  
 return 0;  
}

## 5--treap

#include<cstdio>  
#include<cstring>  
#include<cstdlib>  
using namespace std;  
// TREAP基本  
struct Node  
{  
 Node \*ch[2];  
 int r;//优先级,构成大顶堆  
 int v;//键值,构成排序二叉树  
  
 int cmp(int x)//比较键值大小  
 {  
 if(x==v) return -1;  
 return x<v?0:1;  
 }  
};  
  
//d=0表示左旋,d=1表示右旋  
void rotate(Node\* &o,int d)  
{  
 Node \*k=o->ch[d^1];  
 o->ch[d^1]=k->ch[d];  
 k->ch[d]=o;  
 o=k;  
}  
  
//插入值为x的节点  
void insert(Node\* &o,int x)  
{  
 if(o==NULL)  
 {  
 o=new Node();  
 o->ch[0]=o->ch[1]=NULL;  
 o->v=x;  
 o->r=rand();//在cstdlib头声明  
 }  
 else  
 {  
 //如这里改成int d=o->cmp(x);  
 //就不可以插入相同的值，因为d可能为-1  
 int d=x<(o->v)?0:1;  
 insert(o->ch[d],x);  
 if(o->ch[d]->r > o->r)  
 rotate(o,d^1);  
 }  
}  
  
//删除v值为x的节点  
void remove(Node \*&o,int v)  
{  
 if(o==NULL) return ;//空时返回  
  
 int d=o->cmp(v);  
 if(d==-1)//o就是需要删除的节点  
 {  
 Node \*u=o;  
 if(o->ch[0] && o->ch[1])  
 {  
 int d2 = o->ch[0]->r < o->ch[1]->r ?0:1;  
 rotate(o,d2);  
 remove(o->ch[d2],v);  
 }  
 else  
 {  
 if(o->ch[0]==NULL)o=o->ch[1];  
 else o=o->ch[0];  
 delete u;//记得删除节点  
 }  
 }  
 else remove(o->ch[d],v);  
}  
int find(Node \*o,int x)  
{  
 while(o)  
 {  
 int d=o->cmp(x);  
 if(d==-1)return 1; //存在  
 o=o->ch[d];  
 }  
 return 0; //不存在  
}

## 6--treap\_kth

#include<cstdio>  
#include<cstring>  
#include<cstdlib>  
#include<cassert>  
using namespace std;  
struct Node  
{  
 Node \*ch[2];  
 int r,v,s;//s表示节点数  
  
 Node(int v):v(v)  
 {  
 ch[0]=ch[1]=NULL;  
 r=rand();//在cstdlib头声明  
 s=1;  
 }  
  
 int cmp(int x)  
 {  
 if(x==v)return -1;  
 return x<v?0:1;  
 }  
 void maintain()  
 {  
 s=1;  
 if(ch[0]!=NULL) s+=ch[0]->s;  
 if(ch[1]!=NULL) s+=ch[1]->s;  
 }  
};  
void rotate(Node\* &o,int d)  
{  
 Node \*k=o->ch[d^1];  
 o->ch[d^1]=k->ch[d];  
 k->ch[d]=o;  
 o->maintain();  
 k->maintain();  
 o=k;  
}  
void insert(Node\* &o,int x)//o子树中事先不存在x  
{  
 if(o==NULL) o=new Node(x);  
 else  
 {  
 //如这里改成int d=o->cmp(x);  
 //就不可以插入相同的值，因为d可能为-1  
 int d=x<(o->v)?0:1;  
 insert(o->ch[d],x);  
 if(o->ch[d]->r > o->r)  
 rotate(o,d^1);  
 }  
 o->maintain();  
}  
  
void remove(Node\* &o,int x)  
{  
 if(o==NULL) return ;//空时返回  
  
 int d=o->cmp(x);  
 if(d==-1)  
 {  
 Node \*u=o;  
 if(o->ch[0] && o->ch[1])  
 {  
 int d2=(o->ch[0]->r < o->ch[1]->r)?0:1;  
 rotate(o,d2);  
 remove(o->ch[d2],x);  
 }  
 else  
 {  
 if(o->ch[0]==NULL) o=o->ch[1];  
 else o=o->ch[0];  
 delete u;//这个要放里面  
 }  
 }  
 else remove(o->ch[d],x);  
 if(o) o->maintain();//之前o存在,但是删除节点后o可能就是空NULL了,所以需要先判断o是否为空  
}  
  
//返回关键字从小到大排序时的第k个值  
int kth(Node\* o,int k)  
{  
 assert(o && k>=1 && k<=o->s);//保证输入合法  
 int s=(o->ch[0]==NULL)?0:o->ch[0]->s;  
 if(k==s+1) return o->v;  
 else if(k<=s) return kth(o->ch[0],k);  
 else return kth(o->ch[1],k-s-1);  
}  
  
//返回值x在树中的排名,就算x不在o树中也能返回排名  
//返回值范围在[1,o->s+1]范围内  
int rank(Node\* o,int x)  
{  
 if(o==NULL) return 1;//未找到x;  
  
 int num= o->ch[0]==NULL ? 0:o->ch[0]->s;  
 if(x==o->v) return num+1;  
 else if(x < o->v) return rank(o->ch[0],x);  
 else return rank(o->ch[1],x)+num+1;  
}  
  
  
int main()  
{  
 int n=0;  
 while(scanf("%d",&n)==1 && n)  
 {  
 Node \*root=NULL;  
 for(int i=0;i<n;i++)  
 {  
 int x;  
 scanf("%d",&x);  
 if(root==NULL) root=new Node(x);  
 else insert(root,x);  
 }  
  
 int v;  
 while(scanf("%d",&v)==1)  
 {  
 printf("%d\n",rank(root,v));  
 }  
 }  
 return 0;  
}

## 7--trie

#define MAX 26  
const int maxnode=4000\*100+100;//预计字典树最大节点数目  
const int sigma\_size=26;//每个节点的最多儿子数  
  
struct Trie  
{  
 //这里ch用vector<26元素的数组> ch；实现的话，可以做到动态内存  
 int ch[maxnode][sigma\_size];//ch[i][j]==k表示第i个节点的第j个儿子是节点k  
 int val[maxnode];//val[i]==x表示第i个节点的权值为x  
 int sz;//字典树一共有sz个节点,从0到sz-1标号  
  
 //初始化  
 void clear()  
 {  
 sz=1;  
 memset(ch[0],0,sizeof(ch[0]));//ch值为0表示没有儿子  
 }  
  
 //返回字符c应该对应的儿子编号  
 int idx(char c)  
 {  
 return c-'a';  
 }  
  
 //在字典树中插入单词s,但是如果已经存在s单词会重复插入且覆盖权值  
 //所以执行insert前需要判断一下是否已经存在s单词了  
 void insert(char \*s)  
 {  
 int u=0,n=strlen(s);  
 for(int i=0;i<n;i++)  
 {  
 int id=idx(s[i]);  
 if(ch[u][id]==0)//无该儿子  
 {  
 ch[u][id]=sz;  
 memset(ch[sz],0,sizeof(ch[sz]));  
 val[sz++]=0;  
 }  
 u=ch[u][id];  
 }  
 val[u]=n;  
 }  
  
 //在字典树中查找单词s  
 bool search(char \*s)  
 {  
 int n=strlen(s),u=0;  
 for(int i=0;i<n;i++)  
 {  
 int id=idx(s[i]);  
 if(ch[u][id]==0)  
 return false;  
 u=ch[u][id];  
 }  
 return val[u];  
 }  
};  
Trie trie;//定义一个字典树

## 8--逆序对

#include <iostream>  
#include <string>  
using namespace std;  
#define N 1010  
int c[N];   
int n;  
int lowbit(int i)  
{  
 return i&(-i);  
}  
int insert(int i,int x)  
{  
 while(i<=n){  
 c[i]+=x;  
 i+=lowbit(i);  
 }  
 return 0;  
}  
  
int getsum(int i)  
{  
 int sum=0;  
 while(i>0){  
 sum+=c[i];  
 i-=lowbit(i);  
 }   
 return sum;  
}  
void output()  
{  
 for(int i=1;i<=n;i++) cout<<c[i]<<" ";  
 cout<<endl;  
}  
int main()  
{  
 while(cin>>n){  
 int ans=0;  
 memset(c,0,sizeof(c));  
 for(int i=1;i<=n;i++){  
 int a;  
 cin>>a;  
 insert(a,1);  
 ans+=i-getsum(a);//统计当前序列中大于a的元素的个数  
 }  
 cout<<ans<<endl;  
 }  
 return 0;  
}

# -------Dynamic-------

# -------Geometry-------

# -------Graph-------

## 1--bellman\_ford

#include <iostream>  
#include <vector>  
#include <cstring>  
  
using namespace std;  
/\*  
bellman\_ford 算法  
求最短路  
\*/  
struct node  
{  
 int u, v, w;  
};  
node edge[2003];  
int n, m, s, t;  
  
void bellman\_ford()  
{  
 int i, j;  
 bool flag;//用于优化的  
 int dis[203];//保存最短路径  
 //初始化  
 fill(dis,dis+n,MAX);//其他点为+∞  
 dis[s] = 0;//源点初始化为0  
 m = m<<1;//此处和m = 2\*m是一样的，因为建立的无向图  
 for(i=1;i<n;i++)//进行|V|-1次  
 {  
 flag = false;//刚刚开始标记为假  
 for(j=0;j<m;j++)//对每个边  
 {   
 //if (v.d>u.d+w(u,v))  
 if(dis[edge[j].u]>dis[edge[j].v]+edge[j].w){//进行松弛操作  
 dis[edge[j].u] = dis[edge[j].v]+edge[j].w;//松弛成功  
 flag = true;//若松弛成功则标记为真  
 }  
 }  
 if(!flag)//若所有的边i的循环中没有松弛成功的  
 break;//退出循环  
 //此优化可以大大提高效率。  
 }  
 /\*  
 for(i=0;i<m;i++)  
 if(dis[edge[i].u]>dis[edge[i].v]+edge[i].w)//进行|V|-1次操作后 有边还能进行松弛 说明  
 return true;//存在负环  
 \*/  
   
 printf("%d\n",dis[t]==MAX?-1:dis[t]);//输出结果  
}  
  
int main()  
{  
 int i;  
  
 while(scanf("%d %d",&n,&m)==2){//输入点的总数n，边的总数m  
 for(i=0;i<m;i++)  
 {  
 scanf("%d %d %d",&edge[i].u,&edge[i].v,&edge[i].w);//每条边的u,v,w的输入  
 edge[i+m].u = edge[i].v;//因为为无向图所以u—>v和v—>u 是一样的  
 edge[i+m].v = edge[i].u;//So...  
 edge[i+m].w = edge[i].w;//So...  
 }  
 scanf("%d %d",&s,&t);//起点和终点  
 bellman\_ford();//调用算法部分  
 }  
 return 0;  
}

## 2--BFSSP

#include <algorithm>  
#include <iostream>  
#include <cstdio>  
#include <cstring>  
#include <queue>  
  
using namespace std;  
/\*  
BFS求最短路  
  
\*/  
  
  
struct P  
{  
 int v, w;//v 顶点 w 最短距离  
 bool operator <(const P &a)const{  
 return a.w < w;//按w 从小到大排序  
 }  
};  
struct node//前向星  
{  
 int v, w;//v 顶点 w权重  
 int next;//下一个位置  
};  
node edge[2003];  
int head[203];//头指针数组  
int cnt, s, t;// cnt 下标  
  
void add(int u, int v, int w)//加边操作  
{  
 edge[cnt].v = v;  
 edge[cnt].w = w;  
 edge[cnt].next = head[u];  
 head[u] = cnt++;  
}  
  
void BFS()  
{  
 priority\_queue<P>que;//优先队列 按w从小到大  
 bool vis[203];//标记数组， 标记是否被访问过  
 P p, q;  
 int i, v;  
  
 memset(vis,false,sizeof(vis));//初始化  
 p.v = s;//顶点为 s  
 p.w = 0;//距离为 0  
 que.push(p);//放入队列  
 while(que.empty() == false){//队列不为空  
 p = que.top();//取出队列的队首  
 que.pop();//删除  
 if(p.v == t){//若找到终点  
 printf("%d\n",p.w);//输出结果  
 return ;//返回  
 }  
 vis[p.v] = true;//此点标记为访问过  
 for(i=head[p.v];i!=-1;i=edge[i].next)//查找与该点相连的点  
 {  
 v = edge[i].v;  
 if(vis[v] == false){//若点未被访问过  
 q.v = v;//存入结构体  
 q.w = p.w+edge[i].w;//距离更新  
 que.push(q);//放入队列  
 }  
 }  
 }  
 printf("-1\n");//若没有到达终点 输出-1  
}  
  
int main()  
{  
 int m, u, v, w, n;  
  
 while(scanf("%d %d",&n,&m)==2){//获取点的个数 边的个数  
 memset(head,-1,sizeof(head));//初始化  
 cnt = 0;//初始化  
 while(m--){  
 scanf("%d %d %d",&u,&v,&w);//获取u,v,w  
 add(u,v,w);//加边  
 add(v,u,w);//无向图 双向加边  
 }  
 scanf("%d %d",&s,&t);//获取起止点  
 BFS();  
 }  
 return 0;  
}  
  
题解

## 3--dijkstra\_edge

#include <iostream>  
#include <vector>  
#include <cstring>  
  
using namespace std;  
/\*  
dijkstra 算法  
邻接表  
O(Elog|V|)  
\*/  
//pair 的first 保存的为最短距离, second保存的为顶点编号  
  
typedef pair<int, int >P;//对组 不知道请自行百度   
  
struct node  
{  
 int v, w;//v 为到达的点, w为权重  
 int next;//记录下一个结构体的位置 ，就向链表的next功能是一样的  
};  
node edge[2003];//存所有的边，因为是无向图，所以\*2  
int cnt;//结构体的下标  
int n, s, t;//n 点数,s 起点,t止点  
int head[203];//和链表的头指针数组是一样的。只不过此处head[u]记录的为最后加入 edge 的且与u相连的边在 edge 中的位置，即下标  
  
void add(int u, int v, int w)//加边操作  
{  
 edge[cnt].v = v;  
 edge[cnt].w = w;  
 edge[cnt].next = head[u];//获得下一个结构体的位置  
 head[u] = cnt++;//记录头指针的下标  
}  
  
void dijkstra()  
{  
 int dis[203];//最短路径数组  
 int i, v;//v保存从队列中取出的数的第二个数 也就是顶点的编号  
 priority\_queue<P,vector<P>,greater<P> >que;//优先队列 从小到大  
 node e;//保存边的信息，为了书写方便  
 P p;//保存从队列取出的数值  
  
 fill(dis,dis+n,MAX);//初始化，都为无穷大  
 dis[s] = 0;//s—>s 距离为0  
 que.push(P(0,s));//放入距离 为0 点为s  
 while(!que.empty()){  
 p = que.top();//取出队列中最短距离最小的对组  
 que.pop();//删除  
 v = p.second;//获得最短距离最小的顶点编号  
 if(dis[v] < p.first)//若取出的不是最短距离  
 continue;//则进行下一次循环  
 for(i=head[v];i!=-1;i=edge[i].next)//对与此点相连的所有的点进行遍历  
 {  
 e = edge[i];//为了书写的方便。  
 if(dis[e.v]>dis[v]+e.w){//进行松弛  
 dis[e.v]=dis[v]+e.w;//松弛成功  
 que.push(P(dis[e.v],e.v));//讲找到的松弛成功的距离 和顶点放入队列  
 }  
 }  
 }  
 printf("%d\n",dis[t]==MAX?-1:dis[t]);//输出结果  
}  
  
int main()  
{  
 int m, u, v, w;  
  
 while(scanf("%d %d",&n,&m)==2){//获取点数 边数  
 cnt = 0;//结构体下标从0开始  
 memset(head,-1,sizeof(head));//初始化head[N]数组  
 while(m--){  
 scanf("%d %d %d",&u,&v,&w);//获取u,v,w(u,v)  
 add(u,v,w);//加边  
 add(v,u,w);//加边  
 }  
 scanf("%d %d",&s,&t);//获取起止点  
 dijkstra();  
 }  
 return 0;  
}

## 4--dijkstra\_G[][]

#include <cstdio>  
#include <cstring>  
#include <algorithm>  
#include <iostream>  
#include <queue>  
#define MAX 9999999  
  
using namespace std;  
  
int G[203][203];//二维数组 图的存储  
int n, s, t;//n 点的个数 , s 起点 ,t 终点  
  
void dijkstra()  
{  
 bool vis[203];//相当于集合Q的功能， 标记该点是否访问过  
 int dis[203];//保存最短路径  
 int i, j, k;  
  
 for(i=0;i<n;i++)//初始化  
 dis[i] = G[s][i];//s—>各个点的距离  
 memset(vis,false,sizeof(vis));//初始化为假 表示未访问过  
 dis[s] = 0;//s->s 距离为0  
 vis[s] = true;//s点访问过了，标记为真  
 for(i=1;i<n;i++)//G.V-1次操作+上面对s的访问 = G.V次操作  
 {  
 k = -1;  
 for(j=0;j<n;j++)//从尚未访问过的点中选一个距离最小的点  
 if(!vis[j] && (k==-1||dis[k]>dis[j]))//未访问过 && 是距离最小的  
 k = j;  
 if(k == -1)//若图是不连通的则提前结束  
 break;//跳出循环  
 vis[k] = true;//将k点标记为访问过了  
 for(j=0;j<n;j++)//松弛操作  
 if(!vis[j] && dis[j]>dis[k]+G[k][j])//该点为访问过 && 可以进行松弛  
 {  
 dis[j] = dis[k]+G[k][j];//j点的距离 大于当前点的距离+w(k,j) 则松弛成功，进行更新  
 //pre[j]=k; //路径还原用  
 }  
 }  
 printf("%d\n",dis[t]==MAX?-1:dis[t]);//输出结果  
 /\* 路径还原用  
 queue<int> que;//申请一个队列  
 for(t;t!=-1;t=pre[t])//从t 一直寻找到s  
 que.push(t);//放入队列  
 printf("%d",que.front());//输出第一个  
 que.pop();//删除  
 while(!que.empty()){//队列不为空  
 printf("——>%d",que.front());//输出  
 que.pop();//删除  
 }  
 \*/  
}  
  
int main()  
{  
 int m, i, j, u, v, w;  
  
 while(scanf("%d %d",&n,&m)==2){//获取点的个数 边的个数  
 for(i=0;i<n;i++)  
 for(j=0;j<n;j++)  
 G[i][j] = i==j?0:MAX;//初始化，本身到本身的距离为0，其他的为无穷大  
 while(m--){  
 scanf("%d %d %d",&u,&v,&w);//获取u，v，w(u,v);  
 if(G[u][v]>w)//因为初始化的操作 && 若有重边要去最小的权重值  
 G[u][v] = G[v][u] = w;//无向图 双向  
 }  
 scanf("%d %d",&s,&t);//获取起止点  
 dijkstra();  
 }  
 return 0;  
}

## 5--floyd

#include <iostream>  
#include <vector>  
#include <cstring>  
  
using namespace std;  
/\*  
bellman\_ford 算法  
求最短路  
\*/  
  
int G[100][100];  
void floyd()  
{  
 int i, j, k;  
  
 for(k=0;k<n;k++)  
 for(i=0;i<n;i++)  
 for(j=0;j<n;j++)  
 G[i][j] = min(G[i][j],G[i][k]+G[k][j]);  
 printf("%d\n",G[s][t]==MAX?-1:G[s][t]);  
}  
int main()  
{  
 int i, j, m, u, v, w;  
  
 while(scanf("%d %d",&n,&m)==2){  
 for(i=0;i<n;i++)  
 for(j=0;j<n;j++)  
 G[i][j] = i==j?0:MAX;  
 while(m--){  
 scanf("%d %d %d",&u,&v,&w);  
 if(G[u][v]>w)  
 G[u][v] = G[v][u] = w;  
 }  
 scanf("%d %d",&s,&t);  
 floyd();  
 }  
 return 0;  
}

## 6--isUBG

//无向图的二分图判断  
#include<cstdio>  
#include<vector>  
#include<cstring>  
using namespace std;  
const int maxn=1000+5;  
  
int n;//图节点数  
vector<int> G[maxn];//G[i]表示i节点邻接的点  
int color[maxn];//color[i]=0,1,2 表i节点 不涂颜色 涂白色 涂黑色  
  
//判断无向图是否可二分  
bool bipartite(int u)  
{  
 for(int i=0;i<G[u].size();i++)  
 {  
 int v=G[u][i];  
 if(color[v]==color[u]) return false;  
 if(!color[v])  
 {  
 color[v]=3-color[u];  
 if(!bipartite(v)) return false;  
 }  
 }  
 return true;  
}

## 7--kruskal

/\*  
kruskal算法求最小生成树  
\*/  
int grond[n][n];  
int n,e;  
   
typedef struct node   
{   
 int u; //边的起始顶点   
 int v; //边的终止顶点   
 int w; //边的权值   
}Edge;   
  
void kruskal()   
{   
 int i,j,k;   
 int vset[n]; //辅助数组，判定两个顶点是否连通   
 int E[e]; //存放所有的边   
 k=0;//E数组的下标从0开始   
 for (i=0;i<n;i++)   
 {   
 for (j=0;j<n;j++)   
 {   
 if (grond[i][j]!=0 && grond[i][j]!=INF)   
 {   
 E[k].u=i;   
 E[k].v=j;   
 E[k].w=grond[i][j];   
 k++;   
 }   
 }   
 }   
 sort(E,E+k); //堆排序，按权值从小到大排列   
 for (i=0;i<G.n;i++) //初始化辅助数组   
 {   
 vset[i]=i;   
 }   
 k=1; //生成的边数，最后要刚好为总边数   
 j=0; //E中的下标   
 while (k<n)   
 {   
 sn1=vset[E[j].u];   
 sn2=vset[E[j].v]; //得到两顶点属于的集合编号   
 if (sn1!=sn2) //不在同一集合编号内的话，把边加入最小生成树   
 {  
 //printf("%d ---> %d, %d",E[j].u,E[j].v,E[j].w);   
 k++;   
 for (i=0;i<G.n;i++)   
 {   
 if (vset[i]==sn2)   
 {   
 vset[i]=sn1;   
 }   
 }   
 }   
 j++;   
 }   
}

## 8--maxflow

//最大流模板,可处理重边  
//且节点编号从1到n,边编号从0到m-1  
#include<cstdio>  
#include<cstring>  
#include<queue>  
#include<vector>  
#define INF 1e9  
using namespace std;  
const int maxn=15+5;//之前这里只写10+5,一直TLE,真是悲剧  
  
struct Edge  
{  
 Edge(){}  
 Edge(int from,int to,int cap,int flow):from(from),to(to),cap(cap),flow(flow){}  
 int from,to,cap,flow;  
};  
  
struct Dinic  
{  
 int n,m,s,t; //结点数,边数(包括反向弧),源点与汇点编号  
 vector<Edge> edges; //边表 edges[e]和edges[e^1]互为反向弧  
 vector<int> G[maxn]; //邻接表,G[i][j]表示结点i的第j条边在e数组中的序号  
 bool vis[maxn]; //BFS使用,标记一个节点是否被遍历过  
 int d[maxn]; //d[i]表从起点s到i点的距离(层次)  
 int cur[maxn]; //cur[i]表当前正访问i节点的第cur[i]条弧  
  
 void init(int n,int s,int t)  
 {  
 this->n=n,this->s=s,this->t=t;  
 for(int i=1;i<=n;i++) G[i].clear();  
 edges.clear();  
 }  
  
 void AddEdge(int from,int to,int cap)  
 {  
 edges.push\_back( Edge(from,to,cap,0) );  
 edges.push\_back( Edge(to,from,0,0) );  
 m = edges.size();  
 G[from].push\_back(m-2);  
 G[to].push\_back(m-1);  
 }  
  
 bool BFS()  
 {  
 memset(vis,0,sizeof(vis));  
 queue<int> Q;//用来保存节点编号的  
 Q.push(s);  
 d[s]=0;  
 vis[s]=true;  
 while(!Q.empty())  
 {  
 int x=Q.front(); Q.pop();  
 for(int i=0; i<G[x].size(); i++)  
 {  
 Edge& e=edges[G[x][i]];  
 if(!vis[e.to] && e.cap>e.flow)  
 {  
 vis[e.to]=true;  
 d[e.to] = d[x]+1;  
 Q.push(e.to);  
 }  
 }  
 }  
 return vis[t];  
 }  
  
 //a表示从s到x目前为止所有弧的最小残量  
 //flow表示从x到t的最小残量  
 int DFS(int x,int a)  
 {  
 if(x==t || a==0)return a;  
 int flow=0,f;//flow用来记录从x到t的最小残量  
 for(int& i=cur[x]; i<G[x].size(); i++)  
 {  
 Edge& e=edges[G[x][i]];  
 if(d[x]+1==d[e.to] && (f=DFS( e.to,min(a,e.cap-e.flow) ) )>0 )  
 {  
 e.flow +=f;  
 edges[G[x][i]^1].flow -=f;  
 flow += f;  
 a -= f;  
 if(a==0) break;  
 }  
 }  
 return flow;  
 }  
  
 int Maxflow()  
 {  
 int flow=0;  
 while(BFS())  
 {  
 memset(cur,0,sizeof(cur));  
 flow += DFS(s,INF);  
 }  
 return flow;  
 }  
}DC;  
  
  
int main()  
{  
 int T; scanf("%d",&T);  
 for(int kase=1; kase<=T; ++kase)  
 {  
 int n,m;  
 scanf("%d%d",&n,&m);  
 DC.init(n,1,n);  
 while(m--)  
 {  
 int u,v,w;  
 scanf("%d%d%d",&u,&v,&w);  
 DC.AddEdge(u,v,w);  
 }  
 printf("Case %d: %d\n",kase,DC.Maxflow());  
 }  
 return 0;  
}

## 9--minfee\_maxflow

#include<cstdio>  
#include<cstring>  
#include<queue>  
#include<algorithm>  
#include<vector>  
#define INF 1e9  
using namespace std;  
const int maxn=200+10;  
  
struct Edge  
{  
 int from,to,cap,flow,cost;  
 Edge(){}  
 Edge(int f,int t,int c,int fl,int co):from(f),to(t),cap(c),flow(fl),cost(co){}  
};  
  
struct MCMF  
{  
 int n,m,s,t;  
 vector<Edge> edges;  
 vector<int> G[maxn];  
 bool inq[maxn]; //是否在队列  
 int d[maxn]; //Bellman\_ford单源最短路径  
 int p[maxn]; //p[i]表从s到i的最小费用路径上的最后一条弧编号  
 int a[maxn]; //a[i]表示从s到i的最小残量  
  
 //初始化  
 void init(int n,int s,int t)  
 {  
 this->n=n, this->s=s, this->t=t;  
 edges.clear();  
 for(int i=0;i<n;++i) G[i].clear();  
 }  
  
 //添加一条有向边  
 void AddEdge(int from,int to,int cap,int cost)  
 {  
 edges.push\_back(Edge(from,to,cap,0,cost));  
 edges.push\_back(Edge(to,from,0,0,-cost));  
 m=edges.size();  
 G[from].push\_back(m-2);  
 G[to].push\_back(m-1);  
 }  
  
 //求一次增广路  
 bool BellmanFord(int &flow, int &cost)  
 {  
 for(int i=0;i<n;++i) d[i]=INF;  
 memset(inq,0,sizeof(inq));  
 d[s]=0, a[s]=INF, inq[s]=true, p[s]=0;  
 queue<int> Q;  
 Q.push(s);  
 while(!Q.empty())  
 {  
 int u=Q.front(); Q.pop();  
 inq[u]=false;  
 for(int i=0;i<G[u].size();++i)  
 {  
 Edge &e=edges[G[u][i]];  
 if(e.cap>e.flow && d[e.to]>d[u]+e.cost)  
 {  
 d[e.to]= d[u]+e.cost;  
 p[e.to]=G[u][i];  
 a[e.to]= min(a[u],e.cap-e.flow);  
 if(!inq[e.to]){ Q.push(e.to); inq[e.to]=true; }  
 }  
 }  
 }  
 if(d[t]==INF) return false;  
 flow +=a[t];  
 cost +=a[t]\*d[t];  
 int u=t;  
 while(u!=s)  
 {  
 edges[p[u]].flow += a[t];  
 edges[p[u]^1].flow -=a[t];  
 u = edges[p[u]].from;  
 }  
 return true;  
 }  
  
 //求出最小费用最大流  
 int Min\_cost()  
 {  
 int flow=0,cost=0;  
 while(BellmanFord(flow,cost));  
 return cost;  
 }  
}MM;

## 10--prim

/\*  
prim算法求最小生成树  
\*/  
const int n = 1000;  
const inf = 1e9;  
int grond[n][n];  
int vis[n],dis[n];  
int prim(int start){  
 int ma = 0;  
 for(int i =0 ;i<n;i++){  
 dis[i]=grond[0][i];  
 vis[i]=0;  
 }  
 vis[start] = 1;  
 dis[start] = inf;  
 for(int i = 1;i<n;i++){  
 int b = 0;  
 for(int j = 1;j<n;j++){ //找到当前范围最短路  
 if(!vis[j] && dis[j] < dis[b])  
 b=j;  
 }  
 if(b){  
 ma += dis[b];//总路径长度  
 vis[b]=1;  
 for(int j = 0;j<n;j++){ //更新其他点最短路  
 if(!vis[j] && dis[j]>grond[j][b])  
 dis[j]=grond[j][b];  
 }  
 }  
 }  
 return ma;  
 /\*  
 求最长路径  
 for(int i = 1;i<n;i++)  
 if(dis[i]>ma)  
 ma = dis[i];  
 return ma;  
 \*/  
}

## 11--SPFA

/\*  
SPFA算法  
求最短路  
SPFA() :  
  
　　　　1 对图的建立和处理，dis[N]数组的初始化等等操作  
　　　　2 Q += s //Q 为一个队列 s为源点  
　　　　3 while Q ≠ ∅//队列不为空  
　　　　4　 　 u = Q中的点//从Q中取出一个点u  
　　　　5　　　把u点标记为为访问过的  
　　　　6　　　 for each vertex v∈ G.Adj[u]//对所有的边  
　　　　7 　　　　　　relax(u,v,w)//进行松弛  
　　　　8 　　　　　if(v 未被访问过)//若v未被访问过  
　　　　9　　　　　　　　　　　　Q += v;//加入队列  
稀疏图最好用  
\*/  
#include <cstdio>  
#include <cstring>  
#include <algorithm>  
#include <iostream>  
#include <queue>  
#define MAX 9999999  
  
using namespace std;  
  
struct node//前向星  
{  
 int v,w;//v 终点，w 权值  
 int next;//下一个  
};  
node edge[5203];//前向星  
int head[503];//头指针式的数组  
int cnt;//下标  
int n;//点的个数  
  
void add(int u, int v, int w)  
{  
 edge[cnt].v = v;  
 edge[cnt].w = w;  
 edge[cnt].next = head[u];  
 head[u] = cnt++;  
}  
  
bool SPFA()  
{  
 int i, u, v;//u 从Q中取出的点 v找到的点  
 int dis[503];//保存最短路径  
 int flag[503];//保存某点加入队列的次数  
 bool vis[503];//标记数组  
 deque<int>que;//双向队列  
  
 fill(dis,dis+n+1,MAX);  
 memset(flag,0,sizeof(flag));  
 memset(vis,false,sizeof(vis));  
 dis[1] = 0;// s为1  
 que.push\_back(1);//将s = 1 加入队列  
 while(!que.empty()){  
 u = que.front();  
 que.pop\_front();  
 vis[u] = false;//标记为未访问  
 for(i=head[u];i!=-1;i=edge[i].next)//对所有与该边相连的边进行查找  
 {  
 v = edge[i].v;//保存点 便于操作  
 if(dis[v]>dis[u]+edge[i].w){//进行松弛操作  
 dis[v] = dis[u]+edge[i].w;//松弛成功  
 if(!vis[v]){//若该点未被标记  
 vis[v] = true;//标记为真  
 flag[v]++;//该点入队列次数++  
 if(flag[v]>=n)//若该点进入队列次数超过n次 说明有负环  
 return true;//返回有负环  
 //一下为SLF优化  
 if(!que.empty() && dis[v]<dis[que.front()])//若为队列不为空 && 队列第一个点的最短距离大于当前点的最短距离  
 que.push\_front(v);//将该点放到队首  
 else//不然  
 que.push\_back(v);//放入队尾  
 }  
 }  
 }  
 }  
 return false;//没有负环  
}  
  
int main()  
{  
 int u, v, w, m, k, t;  
   
 scanf("%d",&t);//获取测试数据  
 while(t--){  
 memset(head,-1,sizeof(head));//初始化  
 cnt = 0;//下标为0 初始化  
 scanf("%d %d %d",&n,&m,&k);//获取点的个数 ，正边的个数， 负边的个数  
 while(m--){  
 scanf("%d %d %d",&u,&v,&w);//正边获取  
 add(u,v,w);//无向图  
 add(v,u,w);//双向建图  
 }  
 while(k--){  
 scanf("%d %d %d",&u,&v,&w);  
 add(u,v,-w);//单向图  
 }  
 printf("%s\n",SPFA()?"YES":"NO");//输出结果  
 }  
 return 0;  
}

## 12--UBG\_bestmatch

#include<cstdio>  
#include<cstring>  
#include<algorithm>  
using namespace std;  
const int maxn=100+5;  
  
//把W[maxn][maxn]数组的内容读进去之后，调用solve(n)即可计算出二分图最优匹配  
//不过需要保证该图肯定有完美匹配  
//因为本图用的W[][]来表示一个完全图，所以一定存在完美匹配的  
struct Max\_Match  
{  
 int W[maxn][maxn],n; //W是权值矩阵,n为左右点集大小  
 int Lx[maxn],Ly[maxn];//左右点集的可行顶标值  
 bool S[maxn],T[maxn]; //标记左右点集是否已被访问过  
 int left[maxn]; //left[i]=j表右i与左j匹配,为-1时表无匹配  
  
 bool match(int i)  
 {  
 S[i]=true;  
 for(int j=1;j<=n;j++)if(Lx[i]+Ly[j]==W[i][j] && !T[j])  
 {  
 T[j]=true;  
 if(left[j]==-1 || match(left[j]))  
 {  
 left[j]=i;  
 return true;  
 }  
 }  
 return false;  
 }  
  
 //更新可行顶标,纳入更多的边进来  
 void update()  
 {  
 int a=1<<30;  
 for(int i=1;i<=n;i++)if(S[i])  
 for(int j=1;j<=n;j++)if(!T[j])  
 {  
 a = min(a,Lx[i]+Ly[j]-W[i][j]);  
 }  
 for(int i=1;i<=n;i++)  
 {  
 if(S[i]) Lx[i]-=a;  
 if(T[i]) Ly[i]+=a;  
 }  
 }  
  
 int solve(int n)  
 {  
 this->n=n;  
 memset(left,-1,sizeof(left));  
 for(int i=1;i<=n;i++)//初始化可行顶标值  
 {  
 Lx[i]=Ly[i]=0;  
 for(int j=1;j<=n;j++)  
 Lx[i]=max(Lx[i], W[i][j]);  
 }  
  
 for(int i=1;i<=n;i++)  
 {  
 while(true)  
 {  
 for(int j=1;j<=n;j++) S[j]=T[j]=false;  
 if(match(i)) break;  
 else update();  
 }  
 }  
  
 int ans=0;//最优完美匹配的权值  
 for(int i=1;i<=n;i++) ans+= W[left[i]][i];  
 return ans;  
 }  
}KM;

## 13--UBG\_maxmatch

//二分图最大匹配模板,二分图都是无向图  
//调用下面算法前，保证本图是二分图  
/\*\*\*\*\*\*\*\*\*\*\*\*\*vecotr模板\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
#include<cstdio>  
#include<cstring>  
#include<vector>  
using namespace std;  
const int maxn=100+5;  
  
struct Max\_Match  
{  
 int n,m;//左右点集大小,点从1开始编号  
 vector<int> g[maxn];//g[i]表示左边第i个点邻接的右边点的集合  
 bool vis[maxn];//vis[i]表示右边第i个点是否在本次match中被访问过  
 int left[maxn];//left[i]==j表右边第i个点与左边第j个点匹配,为-1表无点匹配  
  
 void init(int n,int m)  
 {  
 this->n=n;  
 this->m=m;  
 for(int i=1;i<=n;i++) g[i].clear();  
 memset(left,-1,sizeof(left));  
 }  
  
 //判断从左u点是否可以找到一条增广路  
 bool match(int u)  
 {  
 for(int i=0;i<g[u].size();i++)  
 {  
 int v=g[u][i];  
 if(!vis[v])  
 {  
 vis[v]=true;  
 if(left[v]==-1 || match(left[v]))//找到增广路  
 {  
 left[v]=u;  
 return true;  
 }  
 }  
 }  
 return false;  
 }  
  
 //返回当前二分图的最大匹配数  
 int solve()  
 {  
 int ans=0;//最大匹配数  
 for(int i=1;i<=n;i++)//每个左边的节点找一次增广路  
 {  
 memset(vis,0,sizeof(vis));  
 if(match(i)) ans++;//找到一条增广路,形成一个新匹配  
 }  
 return ans;  
 }  
}MM;  
/\*\*\*\*\*\*\*\*\*\*\*\*\*vecotr模板\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# -------Math-------

## 1--ExtendedEuclid

//如果 GCD(a,b) = d, 则存在 x, y, 使 d = ax + by  
//extended\_euclid(a, b) = ax + by  
  
#include <iostream>  
using namespace std;  
int extended\_euclid(int a, int b, int &x, int & y)  
{  
 int d;  
 if(b == 0) {  
 x = 1;  
 y = 0;  
 return a;  
 }  
 d = extended\_euclid(b, a % b, y, x);  
 y -= a / b \* x;  
 return d;  
}  
  
int main()  
{  
 int a = 10, b = 3;  
 int x, y;  
 cout << extended\_euclid(a, b, x, y) << endl;  
 cout << x << " " << y << endl;  
 return 0;  
}

## 2--Greatest Common Divisor

#include <iostream>  
using namespace std;  
  
int GCD(int x, int y)  
{  
 int t;  
 while(y > 0){  
 t = x % y;  
 x = y;  
 y = t;  
 }  
 return x;  
}  
int main()  
{  
 cout << GCD(99, 3) << endl;  
 return 0;  
}

## 3--ModularLinearEquation

//如果 GCD(a, b)不能整除 c, 则 ax + by = c 没有整数解  
// ax ≡ b (mod n) n > 0  
//上式等价于二元一次方程 ax – ny = b  
  
void modular\_linear\_equation(int a, int b, int n)  
{  
 int d, x, y, x0, gcd;  
 // 可以减少扩展欧几里德溢出的可能  
 gcd = GCD(a, n);  
 if(b % gcd != 0){  
 cout << "No Solution" << endl;  
 return;  
 }  
 a /= gcd;  
 b /= gcd;  
 n /= gcd;  
 d = extended\_euclid(a, n, x, y);  
 if(b % d == 0){  
 x0 = (x\*(b/d)) % n;  
 int ans = n;  
 for (int i = 0; i < d; i ++){  
 ans = (x0 + i\*(n/d)) % n;  
 cout << ans << endl;  
 }  
 }  
 else{  
 cout << "No Solution" << endl;  
 }  
}

## 4--ModuleInverse

// ax ≡ 1 (mod n)  
int Inv(int a, int n)  
{  
 int d, x, y;  
 d = extended\_euclid(a, n, x, y);   
 if(d == 1) return (x%n + n) % n;  
 else return -1; // no solution  
}

## 5--Prime

#include <iostream>  
#include <cmath>  
using namespace std;  
  
bool is\_prime(int u)  
{  
 if(u == 0 || u == 1) return false;  
 if(u == 2) return true;  
 if(u % 2 == 0) return false;  
 for (int i = 3; i <= sqrt(u); i += 2)  
 if(u % i == 0) return false;  
 return true;  
}  
int main()  
{  
 cout << is\_prime(23) << endl;  
 return 0;  
}

## 6--SG

//f[]：可以取走的石子个数  
//sg[]:0~n的SG函数值  
//hash[]:mex{}  
/\*  
如果sg[0]^sg[1]^....^sg[n]=1的话，先手赢  
否则后手赢。  
\*/  
int f[N],sg[N],hash[N];   
void getSG(int n)  
{  
 int i,j;  
 memset(sg,0,sizeof(sg));  
 for(i=1;i<=n;i++)  
 {  
 memset(hash,0,sizeof(hash));  
 for(j=1;f[j]<=i;j++)  
 hash[sg[i-f[j]]]=1;  
 for(j=0;j<=n;j++) //求mes{}中未出现的最小的非负整数  
 {  
 if(hash[j]==0)  
 {  
 sg[i]=j;  
 break;  
 }  
 }  
 }  
}  
  
//注意 S数组要按从小到大排序 SG函数要初始化为-1 对于每个集合只需初始化1遍  
//n是集合s的大小 S[i]是定义的特殊取法规则的数组  
int s[110],sg[10010],n;  
int SG\_dfs(int x)  
{  
 int i;  
 if(sg[x]!=-1)  
 return sg[x];  
 bool vis[110];  
 memset(vis,0,sizeof(vis));  
 for(i=0;i<n;i++)  
 {  
 if(x>=s[i])  
 {  
 SG\_dfs(x-s[i]);  
 vis[sg[x-s[i]]]=1;  
 }  
 }  
 int e;  
 for(i=0;;i++)  
 if(!vis[i])  
 {  
 e=i;  
 break;  
 }  
 return sg[x]=e;  
}

## 7--SlevePrime

#include <cstdlib>  
#include <cstdio>   
#include <cstring>  
#include <cmath>  
  
const int M = 1000; // M : size  
bool mark[M]; // true : prime number  
void sieve\_prime()  
{  
 memset(mark, true, sizeof(mark));  
 mark[0] = mark[1] = false;  
 for(int i=2; i <= sqrt(M) ;i++) {  
 if(mark[i]) {  
 for(int j=i\*i; j < M ;j+=i)  
 mark[j] = false;  
 }  
 }  
}  
  
int main()  
{  
   
}

## 8--Subset

#include <iostream>  
#include <vector>  
#include <set>  
using namespace std;  
  
const int Size = 25;  
set<int> GetSubset(int K)  
{  
 set<int> res;  
 int comb = (1 << K) - 1;  
 while(comb < 1 << Size)  
 {  
 res.insert(comb);  
 int x = comb & -comb, y = comb + x;  
 comb = ((comb & ~y) / x >> 1) | y;  
 }  
 return res;  
}  
int main()  
{  
 cout << GetSubset(2).size() << endl;  
 return 0;  
}

# -------Others-------

## 1--discrete

## 2--radixsort

#include<iostream>  
using namespace std;  
  
#define MAXSIZE 10000  
//获取一个数的长度  
int length(int a){  
 int num = 0;  
 while(a){  
 a/=10;  
 num++;  
 }  
 return num;  
}  
//获取数组最长的数的长度  
int maxLength(int ar[],int n) {  
 int malen = 0;  
 for (int i = 0; i < n; i++) {  
 int currentLength = length(ar[i]);  
 if (malen < currentLength) {  
 malen = currentLength;  
 }  
 }  
 return malen;  
 }  
//获取x右往左从0开始的第d位数字  
int getdigit(int x, int d) {  
 int a[] = { 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000, 1000000000 };  
 return ((x / a[d]) % 10);  
}  
  
void lsdradix\_sort(int arr[],int d,int n)  
{  
 const int radix = 10;  
 int count[radix], i, j;  
 int \*bucket = new int[n];  
  
 //按照分配标准依次进行排序过程  
 for(int k = 1; k <= d; ++k)  
 {  
 //置空  
 for(i = 0; i < radix; i++)  
 {  
 count[i] = 0;  
 }  
 //统计各个桶中所盛数据个数  
 for(i = 0; i < n; i++)  
 {  
 count[getdigit(arr[i], k)]++;  
 }  
 //count[i]表示第i个桶的右边界索引  
 for(i = 1; i < radix; i++)  
 {  
 count[i] = count[i] + count[i-1];  
 }  
 for(i = n-1;i >= 0; --i) //这里要从右向左扫描，保证排序稳定性  
 {  
 j = getdigit(arr[i], k); //例如：576的第3位是5  
 bucket[count[j]-1] = arr[i];  
 /\*\*  
 这里要注意的是，count的内容是不会重复的  
 因为在上一个循环中就已经规定了各位占多少位数了。  
 \*/  
 --count[j]; //对应桶的装入数据索引减一  
 }  
 //注意：此时count[i]为第i个桶左边界  
 //从各个桶中收集数据  
 for(i = 0,j = 0; i < n; ++i, ++j)  
 {  
 arr[i] = bucket[j];  
 }  
 }  
 delete [] bucket;  
}  
  
int main()  
{  
 int br[10] = {789, 80, 90, 589, 998, 965, 852, 123, 456, 20};  
 int len = maxLength(br,10);  
 lsdradix\_sort(br,len,3);  
 for(int i=0;i<10;i++)  
 cout<<br[i]<<" ";  
 cout<<endl;  
 return 0;  
}

## 3--sort

#include <iostream>  
#include <map>  
using namespace std;  
void exch(int &a,int &b)  
{  
 int t=a;  
 a=b;  
 b=t;  
}  
void charu(int a[],int n){  
 for(int i=1;i<n;i++){  
 for(int j=i;j>0&&a[j]<a[j-1];j--)  
 exch(a[j],a[j-1]);  
 }  
}  
void maopao(int a[],int n){  
 for(int i=0;i<n;i++)  
 for(int j=0;j<n;j++)  
 if(a[i]>a[j]) exch(a[i],a[j]);  
}  
void xuanze(int a[],int n){  
 for(int i=0;i<n;i++)  
 {  
 int mi=a[i];  
 int t = -1;  
 for(int j=i+1;j<n;j++)  
 if(a[j]<mi)  
 {mi=a[j];t = j;}  
 if(t!=-1)  
 exch(a[i],a[t]);  
 }  
}  
void xir(int a[],int n)  
{  
 int h=1;  
 while(h<n/3) h=3\*h+1;//1,4,13,40,121,364,1093...  
 while(h>=1)  
 {  
 for(int i=h;i<n;i++)  
 {  
 for(int j=i;j>=h&&a[j]<a[j-h];j-=h)  
 exch(a[j],a[j-h]);  
  
 }  
 h/=3;  
 }  
}  
int b[1000];  
void guibing(int a[],int lo,int mid,int hi)  
{  
 int i=lo;  
 int j=mid+1;  
 for(int k=lo;k<=hi;k++)  
 b[k]=a[k];  
 for(int k=lo;k<=hi;k++)  
 if(i>mid) a[k]=b[j++];  
 else if(j>hi) a[k]=b[i++];  
 else if(b[j]<b[i]) a[k]=b[j++];  
 else a[k]=b[i++];  
}  
void guibingsort(int a[],int lo,int hi)  
{  
 if(hi<=lo) return ;  
 int mid=lo+(hi-lo)/2;  
 guibingsort(a,lo,mid);  
 guibingsort(a,mid+1,hi);  
 guibing(a,lo,mid,hi);  
}  
void guibingxiadaoshang(int a[],int n)  
{  
 for(int sz=1;sz<n;sz+=sz)  
 for(int lo=0;lo<n-sz;lo+=sz+sz)  
 guibing(a,lo,lo+sz-1,min(lo+sz+sz-1,n-1));  
}  
int qiefen(int a[],int lo,int hi)  
{  
 int i=lo;int j=hi+1;  
 if(j<=i) return i;  
 int v = a[lo];  
 while(1)  
 {  
 while(i<j && v<a[--j]);  
 while(i<j && a[++i]<v);  
 if(i>=j) break;  
 exch(a[i],a[j]);  
 }  
 exch(a[lo],a[j]);  
 return j;  
}  
void quicksort(int a[],int lo,int hi)  
{  
 if(hi<=lo) return ;  
 int j=qiefen(a,lo,hi);  
 quicksort(a,lo,j-1);  
 quicksort(a,j+1,hi);  
}  
void sink(int a[],int k,int n)  
{  
 while(2\*k<=n)  
 {  
 int j=2\*k;  
 if(j<n && a[j]<a[j+1])j++;  
 if(a[k]>=a[j]) break;  
 exch(a[k],a[j]);  
 k=j;  
 }  
}  
void duipaixu(int a[],int n)  
{  
 for(int k=n/2;k>=1;k--)  
 sink(a,k,n);  
 while(n>1){  
 exch(a[1],a[n--]);//左节点2k所以是a[1]，若左节点是2k-1这里就是a[0]  
 sink(a,1,n);  
 }  
}  
  
int main()  
{  
 int aaa[]={7,5,3,2,1,4,8,9};  
 int bbb[]={7,5,3,2,1,4,8,9};  
 int ccc[]={7,5,3,2,1,4,8,9};  
 int ddd[]={7,5,3,2,1,4,8,9};  
 int fff[]={0,7,5,3,2,1,4,8,9};  
 xuanze(ddd,9);  
 guibingsort(aaa,0,7);  
 guibingxiadaoshang(bbb,8);  
 duipaixu(fff,8);  
 quicksort(ccc,0,7);  
 for(int i=0;i<8;i++)  
 cout<<aaa[i];  
 cout<<endl;  
 for(int i=0;i<8;i++)  
 cout<<bbb[i];  
 cout<<endl;  
 for(int i=0;i<8;i++)  
 cout<<ccc[i];  
 cout<<endl;  
 for(int i=0;i<8;i++)  
 cout<<ddd[i];  
 cout<<endl;  
 for(int i=1;i<9;i++)  
 cout<<fff[i];  
 cout<<endl;  
 return 0;  
}

# -------String-------

## 1--dictionary

/\*  
先输入一大堆字典，然后再输入一大堆查询，问是否在字典里面，或者增加、删除或替换一个字符后是否在字典里面。  
其中有几个比较坑的地方，比如输出是要按照输入的顺序的，输入的单词可能有前几个是重复的等等。  
\*/  
  
  
#include <iostream>  
#include <cstdio>  
#include <vector>  
#include <algorithm>  
#include <cstring>  
#include <map>  
  
  
using namespace std;  
  
  
struct node{  
 char c;  
 bool ed;  
 vector<node\*> son;  
 node (){  
 ed = false;  
 }  
 node (char d){  
 c = d;  
 ed = false;  
 }  
};  
map <string,int> order;<//这个是存字符串顺序的  
bool cmp(string a,string b){//对答案进行排序  
 return order[a]<order[b];  
}  
int number;  
node top;  
node \*now;  
string tem;  
  
  
vector<string> ans;  
bool found(string a){//在答案集里找重复  
 for(int i = 0;i<ans.size();i++)  
 {  
 if(ans[i]==a)  
 return true;  
 }  
 return false;  
}  
void buildtree(node \*p,string s,int pos){//建立字典树  
 if(pos >= s.length()){  
 p->ed = true;  
 return;  
 }  
 bool yes = false;  
 for(int i = 0 ; i < p->son.size() ; i++){  
 node \*ww = p->son[i];  
 if(ww->c == s[pos]){  
 buildtree(ww,s,pos+1);  
 yes = true;  
 break;  
 }  
 }  
 if(!yes)  
 {  
 node\* ww = new node;  
 ww -> c = s[pos];  
 p->son.push\_back(ww);  
 buildtree(p->son.back(),s,pos+1);  
 }  
}  
  
  
bool exis(node \*p,string s,int pos){//单词是否在树中  
 // cout<<s<<endl;  
 if(pos >= s.length() && p->ed)  
 return true;  
 for(int i = 0 ; i < p->son.size() ; i++){  
 node \*ww = p->son[i];  
 //cout<<tem->c<<endl;  
 if(ww->c == s[pos])  
 {  
 return exis(ww,s,pos+1);  
 }  
 }  
 return false;  
}  
void rep(node \*p,string s,int pos){//替换  
 if(pos >= s.length())  
 return ;  
 int len = p->son.size();  
 for(int i = 0 ; i < len ; i++){  
 node \*ttt = p->son[i];  
 string a = s.substr(0,pos)+ttt->c+s.substr(pos+1);  
 //printf("a[%d] has be %c\n",pos,ttt->c);  
 if(exis(now,a,0) && !found(a)){  
 tem+=" "+a;  
 ans.push\_back(a);  
 }  
 }  
 for(int i = 0 ; i < len ; i++){  
 node \*ttt = p->son[i];  
 if(ttt -> c == s[pos])  
 {  
 rep(ttt,s,pos+1);  
 break;  
 }  
 }  
}  
void inc(node \*p,string s,int pos){//增加  
 if(pos>=s.length())  
 return ;  
 int len = s.length()-pos;  
 for(int i = 0 ; i < p->son.size() ; i++){  
 node \*rrr = p->son[i];  
 if(rrr -> c == s[pos])  
 {  
 inc(rrr,s,pos+1);//\_\_\_\_\_\_\_\_\_\_\_  
 break;  
 }  
 }  
 for(int i = 0 ; i < p->son.size() ; i++){  
 node \*ttt = p->son[i];  
 string a = s.substr(0,pos) + ttt->c + s.substr(pos);  
 // printf("now a = ");  
 //cout<<a<<endl;  
 if(exis(now,a,0) && !found(a))  
 {  
 ans.push\_back(a);  
 tem+=" "+a;  
 }  
 }  
}  
void add(node \*p,string s,int pos){//末尾增加  
 int len = p->son.size();  
 if(pos >= s.length()){  
 for(int i = 0 ;i<len;i++){  
 node \* ttt = p->son[i];  
 string a = s + ttt->c;  
 if(exis(now,a,0)&& !found(a)){  
 tem+=" " + a;  
 ans.push\_back(a);  
 }  
 }  
 }  
 for(int i=0;i<len;i++){  
 node \*ttt = p->son[i];  
 if(ttt->c==s[pos])  
 {  
 add(ttt,s,pos+1);  
 return;  
 }  
 }  
}  
void dic(string s,int pos){//减少一个  
 int length = s.length();  
 string a = s.substr(1,length-1);  
 if( exis(now,a,0)&&!found(a))  
 {  
 ans.push\_back(a);  
 tem += " " +a;  
 }  
 a=s.substr(0,length-1);  
 if(exis(now,a,0) &&!found(a))  
 {  
 tem += " " +a;  
 ans.push\_back(a);  
 }  
 for(int i = 1; i < s.length()-1;i++){  
 string k = s.substr(0,i)+s.substr(i+1,length-i-1);  
  
  
 if(exis(now,k,0)&&!found(k))  
 {  
 ans.push\_back(k);  
 tem += " " + k;  
 }  
 }  
}  
  
  
int n,m;  
  
  
void dele(node \*p){//释放内存  
 for(int i = 0;i< p->son.size();i++){  
 node \*rrr = p->son[i];  
 delete(rrr);  
 }  
 delete p;  
}  
  
  
int main(){  
 top.c=' ';  
 n = 0;  
 number=0;  
 now = &top;  
 string s;  
 while(1){  
 cin>>s;  
 if(s=="#") break;  
 buildtree(now,s,0);  
 order[s]=number;  
 number++;  
 }  
 while(1){  
 cin>>s;  
 if(s=="#")  
 break;  
 tem = "";  
 ans.clear();  
 if(exis(now,s,0))  
 cout<<s<<" "<<"is"<<" "<<"correct"<<endl;  
 else{  
 cout<<s<<":";  
 rep(now,s,0);  
 inc(now,s,0);  
 add(now,s,0);  
 dic(s,0);//大概这里就是超时的地方，因为必须四个操作都要试一试，所以不能减掉。无法加个if然后不去做它。  
 sort(ans.begin(),ans.end(),cmp);  
 for(int i =0 ;i<ans.size();i++)  
 cout<<" "<<ans[i];  
 cout<<" "<<endl;  
 }  
 }  
 dele(now);  
 return 0;  
}  
  
/\*  
下面这个代码是利用指针操作来完成上面的题目，打开了新世界的大门。  
\*/  
/\*  
#include<iostream>  
#include<string.h>  
using namespace std;  
  
  
char dict[10001][17];  
char word[51][17];  
int n1=0;  
int n2=0;  
  
  
bool Change(char\* word,char\* dict)   
{  
 int dif=0; //记录word与dict中在相同位置出现不同字符的个数  
   
 while(\*word)  
 {  
 if(\*(word++) != \*(dict++))  
 {  
 dif++;  
 if(dif>1)  
 return false;  
 }  
 }  
 return true;  
}  
  
  
bool Del(char\* word,char\* dict)  
{  
 int dif=0; //记录word与dict中在对应位置出现不同字符的个数  
 while(\*word)  
 {  
 if(\*word != \*dict)  
 {  
 word++; //word后移一位再匹配  
 dif++;  
 if(dif>1)  
 return false;  
 }  
 else  
 {  
 word++;  
 dict++;  
 }  
 }  
 return true;  
}  
  
  
bool Add(char\* word,char\* dict)  
{   
 int dif=0;  
  
  
 while(\*dict)  
 {  
 if(\*word != \*dict)  
 {  
 dict++;  
 dif++;  
 if(dif>1)  
 return false;  
 }  
 else  
 {  
 word++;  
 dict++;  
 }  
 }  
 return true;  
}  
int main(void)  
{  
 while(cin>>dict[n1] && dict[n1++][0]!='#');  
 while(cin>>word[n2] && word[n2++][0]!='#');  
 n1--;  
 n2--;  
 int\* DictLen=new int[n1]; //记计算字典中各个单词的长度  
 for(int n=0;n<n1;n++)  
 DictLen[n]=strlen(dict[n]);  
  
  
 for(int i=0;i<n2;i++)  
 {  
 int\* address=new int[n1]; //记录word[i]通过变化得到的单词在dict中的下标  
 int pa=0; //address指针  
  
  
 bool flag=false; //标记字典中是否含有单词word[i]  
 int len=strlen(word[i]);  
  
  
 for(int k=0;k<n1;k++) //遍历字典  
 {  
 if(DictLen[k]==len) //Change or Equal  
 {  
 if(!strcmp(word[i],dict[k]))  
 {  
 flag=true;  
 break;  
 }  
 else if(Change(word[i],dict[k]))  
 address[pa++]=k;  
 }  
 else if(len-DictLen[k]==1) //Delete  
 {  
 if(Del(word[i],dict[k]))  
 address[pa++]=k;  
 }  
 else if(DictLen[k]-len==1) //Add  
 {  
 if(Add(word[i],dict[k]))  
 address[pa++]=k;  
 }  
 }  
 if(flag)  
 cout<<word[i]<<" is correct"<<endl;  
 else  
 {  
 cout<<word[i]<<": ";  
 for(int j=0;j<pa;j++)  
 cout<<dict[ address[j] ]<<' ';  
 cout<<endl;  
 }  
  
  
 delete address;  
 }  
 return 0;  
}  
\*/

## 2--KMP

#include <iostream>  
#include <cstring>  
#include <cstdio>  
using namespace std;  
const int N=100;  
char str[100],ptr[100];  
int next[100];  
string ans;  
void getnext()  
{  
 int i,n,k;  
 n=strlen(ptr);  
 memset(next,0,sizeof(next));  
 k=0;  
 for(i=1;i<n;i++)  
 {  
 while(k>0 && ptr[k]!=ptr[i])  
 k=next[k];  
 if(ptr[k]==ptr[i]) k++;  
 next[i+1]=k;  
 //next表示的是匹配长度  
 }  
}  
int kmp(char \*a,char \*b)  
{  
 int i=0,j=0;  
 int len1=strlen(a);  
 int len2=strlen(b);  
 getnext();  
 while(i<len1&&j<len2)  
 {  
 if(j==0||a[i]==b[j])  
 { i++;j++; }  
 else j=next[j];//到前一个匹配点  
 }  
 if(j>=len2)  
 return i-j;  
 else return 0;  
}  
int main(){  
 while( scanf( "%s%s", str, ptr ) )  
 {  
 int ans = kmp(str,ptr);  
 if(ans)  
 printf( "%d\n", kmp( str,ptr ));  
 else  
 printf("404 Not find\n");  
 }  
 return 0;  
}

## 3--SA

/\*  
 Problem: JZOJ1598(询问一个字符串中有多少至少出现两次的子串)  
 Content: SA's Code and Explanation  
 Author : YxuanwKeith  
\*/  
  
#include <cstdio>  
#include <cstring>  
#include <algorithm>  
  
using namespace std;  
  
const int MAXN = 100005;  
  
char ch[MAXN], All[MAXN];  
int SA[MAXN], rank[MAXN], Height[MAXN], tax[MAXN], tp[MAXN], a[MAXN], n, m;  
char str[MAXN];  
//rank[i] 第i个后缀的排名; SA[i] 排名为i的后缀位置; Height[i] 排名为i的后缀与排名为(i-1)的后缀的LCP  
//tax[i] 计数排序辅助数组; tp[i] rank的辅助数组(计数排序中的第二关键字),与SA意义一样。  
//a为原串  
void RSort() {  
 //rank第一关键字,tp第二关键字。  
 for (int i = 0; i <= m; i ++) tax[i] = 0;  
 for (int i = 1; i <= n; i ++) tax[rank[tp[i]]] ++;  
 for (int i = 1; i <= m; i ++) tax[i] += tax[i-1];  
 for (int i = n; i >= 1; i --) SA[tax[rank[tp[i]]] --] = tp[i]; //确保满足第一关键字的同时，再满足第二关键字的要求  
} //计数排序,把新的二元组排序。  
  
int cmp(int \*f, int x, int y, int w) { return f[x] == f[y] && f[x + w] == f[y + w]; }  
//通过二元组两个下标的比较，确定两个子串是否相同  
  
void Suffix() {  
 //SA  
 for (int i = 1; i <= n; i ++) rank[i] = a[i], tp[i] = i;  
 m = 127 ,RSort(); //一开始是以单个字符为单位，所以(m = 127)  
  
 for (int w = 1, p = 1, i; p < n; w += w, m = p) { //把子串长度翻倍,更新rank  
  
 //w 当前一个子串的长度; m 当前离散后的排名种类数  
 //当前的tp(第二关键字)可直接由上一次的SA的得到  
 for (p = 0, i = n - w + 1; i <= n; i ++) tp[++ p] = i; //长度越界,第二关键字为0  
 for (i = 1; i <= n; i ++) if (SA[i] > w) tp[++ p] = SA[i] - w;  
  
 //更新SA值,并用tp暂时存下上一轮的rank(用于cmp比较)  
 RSort(), swap(rank, tp), rank[SA[1]] = p = 1;  
  
 //用已经完成的SA来更新与它互逆的rank,并离散rank  
 for (i = 2; i <= n; i ++)   
 rank[SA[i]] = cmp(tp, SA[i], SA[i - 1], w) ? p : ++ p;  
 }  
 //至此后缀数组排名完毕  
   
 //离散：把相等的字符串的rank设为相同。  
 //LCP最大公共前缀  
 int j, k = 0;  
 for(int i = 1; i <= n; Height[rank[i ++]] = k)  
 for( k = k ? k - 1 : k, j = SA[rank[i] - 1]; a[i + k] == a[j + k]; ++ k);  
 //这个知道原理后就比较好理解程序  
}  
  
void Init() {  
 scanf("%s", str);  
 n = strlen(str);  
 for (int i = 0; i < n; i ++) a[i + 1] = str[i];  
}  
  
int main() {  
 Init();  
 Suffix();  
  
 int ans = Height[2];  
 for (int i = 3; i <= n; i ++) ans += max(Height[i] - Height[i - 1], 0);  
 printf("%d\n", ans);  
}