## 图论

### 最短路

#### Dijkstra 优先队列+邻接表

int n, m; //n为顶点数，m为边数

int dis[N]; //dis[i]为起点到i的最短距离

bool vis[N]; //vis[i]表示点i是否已被用来松弛其它点

//使优先队列按dis从小到大排

struct qnode{

int id, dis;

qnode(int id, int dis):id(id),dis(dis){}

bool operator <(const qnode &b)const{

return dis > b.dis;

}

};

struct node{

int to, w;

node(int to, int w):to(to),w(w){}

};

vector<node>v[N];

void dijkstra(int s){

memset(dis,INF,sizeof(dis));

dis[s]=0;

priority\_queue<qnode>q;

q.push(qnode(s,0));

while(!q.empty()){

int k=q.top().id;q.pop();

if(vis[k]) continue;

vis[k] = true;

for(int i=0;i<v[k].size();i++){

node t = v[k][i];

if(dis[k]+t.w<dis[t.to]){

dis[t.to] = dis[k]+t.w;

pre[t.to] = k;

q.push(qnode(t.to,dis[t.to]));

}

}

}

}

**SPFA**

struct node{

int to,next,w;

}edge[M];

int V,E,idx; //V为顶点数，E为边数

int dis[N],head[N],cnt[N]; //dis[i]表示i点到源点的最短距离

bool vis[N]; //vis[i]=true表示点i在队列中

//初始化

void init(){

idx=1;

memset(head,0,sizeof(head));

memset(cnt,0,sizeof(cnt));

}

//添加权值为w的边u->v

void addedge(int u,int v,int w){

edge[idx].to=v;

edge[idx].w=w;

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

head[u]=idx;

idx++;

}

//Spfa算法 return 0 时存在负环

bool Spfa(int s){

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

dis[i]=INF;

vis[i]=false;

}

dis[s]=0;

queue<int>q;

q.push(s);

while(!q.empty()){

int k=q.front();

q.pop();

vis[k]=false;

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

node t=edge[i];

if(dis[t.to]>dis[k]+t.w){

dis[t.to]=dis[k]+t.w;

if(!vis[t.to]){

q.push(t.to);

vis[t.to]=true;

cnt[t.to]++;

if(cnt[t.to]>=V)

return false;

}

}

}

}

return true;

}

int main(){

init();

cin >> V >> E;

int u,v,w;

idx = 1;

while(E--){

cin >> u >> v >> w;

addedge(u,v,w);

addedge(v,u,w);

}

int st,ed;

cin >> st >> ed;

Spfa(st);

cout << dis[ed] << endl;

}

**Floyd 求最小环**

int n,m;

ll g[maxn][maxn],mp[maxn][maxn];

int Floyd(){

ll res = INF;

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

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

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

if(i!=j&&j!=k&&i!=k)

res=min(res,g[i][j]+mp[i][k]+mp[k][j]);

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

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

g[i][j] = min(g[i][j],g[i][k]+g[k][j]);

return res;

}

**MST**

**Kruskal算法**

int n,cnt;//n是节点数,cnt是边数

struct edge{

int from,to,cost;

}g[maxn\*maxn];

int fa[maxn];

/\*

并查集的操作…

\*/

bool cmp(edge a,edge b) {return a.cost<b.cost;}

int Kruskal(){

int ans = 0;

sort(g+1,g+cnt+1,cmp);

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

if(!same(g[i].from,g[i].to)){

ans += g[i].cost;

join(g[i].from,g[i].to);

}

}

return ans;

}

**prim算法**

int n,graph[maxn][maxn],low[maxn],vis[maxn];

int prim(){

memset(low,INF,sizeof(low));

int pos,min,ans=0;

vis[1] = 1;pos = 1;//起始点从1开始

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

if(!vis[i]) low[i] = graph[pos][i];

}

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

min = INF;

for(int j=1;j<=n;j++){//找出连通的最小权值,并记录位置

if(!vis[j]&&low[j]<min){

min = low[j];

pos = j;

}

}

ans += min;

vis[pos] = 1;

for(int j=1;j<=n;j++){//更新权值

if(!vis[j]&&graph[pos][j]<low[j]){

low[j] = graph[pos][j];

}

}

}

return ans;

}

**拓扑排序**

int n,ans[maxn],in[maxn];

vector<int>graph[maxn];

queue<int> Q;

bool topo(){

int cnt = 0;

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

if(in[i]==0) Q.push(i);

while(Q.size()){

int now = Q.front();Q.pop();

ans[++cnt] = now;

for(int i=0;i<graph[now].size();i++){

int tmp = graph[now][i];

in[tmp]--;

if(in[tmp]==0) Q.push(tmp);

}

}

if(cnt==n) return true;

return false;

}

**二分图匹配**

int n,m,k;

int graph[maxn][maxn],vis[maxn],nxt[maxn];

int Find(int x){

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

if(graph[x][i] && !vis[i]){

vis[i] = 1;

if(nxt[i] == -1 || Find(nxt[i])){

nxt[i] = x;

return true;

}

}

}

return false;

}

int Match(){

int ans = 0;

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

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

if(Find(i)) ans++;

}

return ans;

}

二分图

最小路径覆盖 = |G| - 最大匹配数

最小点覆盖 = 最大匹配数

**欧拉路**

欧拉回路: 每条边恰好只走一次, 并能回到出发点的路径

欧拉路径: 经过每一条边一次, 但是不要求回到起始点

欧拉回路存在性的判定:

无向图: 每个顶点的度数都是偶数,则存在欧拉回路

有向图: 每个节顶点的入度都等于出度, 则存在欧拉回路

欧拉路径存在性的判定:

无向图: 一个无向图存在欧拉路径, 当且仅当该图所有顶点的度数为偶数或者除了两个度数为奇数外其余的全是偶数

有向图: 一个有向图存在欧拉路径, 当且仅当该图所有顶点的度数为零或者一个顶点的度数为1, 另一个度数为-1, 其他顶点的度数为0

**最小树形图**

//最小树形图

int n,m,u,v,pos;

struct node{

int u,v; ll w;

}edge[M];

ll w,in[N];

int vis[N],id[N],pre[N];

ll D\_MST(int root, int V, int E){

ll ans = 0;

const ll INF = 1e18;

while(1){

for(int i=0;i<V;i++) in[i] = i==root ? 0 : INF;

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

u = edge[i].u;

v = edge[i].v;

if(edge[i].w<in[v] && u!=v){

pre[v] = u;

in[v] = edge[i].w;

if(u==root) pos = i; //无定根

}

}

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

if(i==root) continue;

if(in[i]==INF) return -1;

}

clr(id,-1);

clr(vis,-1);

int cnt = 0;

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

ans += in[i];

int v = i;

while(vis[v]!=i && id[v] == -1 && v!=root){

vis[v] = i;

v = pre[v];

}

if(v!=root&&id[v]==-1){

for(int u=pre[v];u!=v;u = pre[u])

id[u] = cnt;

id[v] = cnt++;

}

}

if(cnt==0) break;

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

if(id[i]==-1) id[i] = cnt++;

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

u = edge[i].u;

v = edge[i].v;

edge[i].u = id[u];

edge[i].v = id[v];

if(id[u]!=id[v])

edge[i].w -= in[v];

}

V = cnt;

root = id[root];

}

return ans;

}

**网络流Dicnic**

bool bfs(){

while(que.size()) que.pop();

clr(dis,INF);

que.push(s);

dis[s] = 0;

while(que.size()){

int u = que.front(); que.pop();

for(int i=head[u];i+1;i=edge[i].nxt){

int v = edge[i].v;

if(dis[v]==INF&&edge[i].w){

dis[v] = dis[u] + 1;

que.push(v);

}

}

}

return dis[t]!=INF;

}

int dfs(int u,int low){

if(u==t) return low;

int res = 0,flow;

for(int &i=cur[u];i+1;i=edge[i].nxt){

int v = edge[i].v;

if(dis[v] == dis[u] + 1 && edge[i].w

&& (flow = dfs(v,min(edge[i].w,low - res)))){

edge[i].w -= flow;

edge[i^1].w += flow;

res += flow;

if(res == low) return res;

}

}

if(!res) dis[u] = INF;

return res;

}

int Dicnic(){

int res = 0;

while(bfs()){

memcpy(cur,head,sizeof(head));

res += dfs(s,INF);

}

return res;

}

## 数据结构

### 并查集

struct Dsu{

int size,set\_num;

int fa[N],num[N];//集合个数，father

Dsu(int n){

size = set\_num = n;

for(int i=0;i<=n;i++) fa[i] = i,num[i] = 1;

}

int fd(int x){

return x == fa[x] ? x : fa[x] = fd(fa[x]);

}

bool join(int x,int y){

int u = fd(x), v = fd(y);

if(u==v) return 0;

fa[u] = v;

num[v] += num[u];

set\_num--;

return 1;

}

bool same(int x,int y){

return fd(x) == fd(y);

}

};

### 线段树

struct segt{

struct seg{

int l,r;

ll sum,lz;

void update(int v){

sum += v\*(r-l+1);

lz += v;

}

};

int\* a;

seg t[N<<2];

#define lc x<<1

#define rc x<<1|1

#define mid ((t[x].l+t[x].r)>>1)

inline void modify(int \*arr){

a = arr;

}

inline void push\_up(int x){

t[x].sum = t[lc].sum + t[rc].sum;

}

inline void push\_down(int x){

ll lz = t[x].lz;

t[lc].update(lz);

t[rc].update(lz);

t[x].lz = 0;

}

void build(int x,int l,int r){

t[x].l = l, t[x].r = r, t[x].sum = t[x].lz = 0;

if(l==r){

t[x].sum = a[l];

return;

}

build(lc,l,mid); build(rc,mid+1,r);

push\_up(x);

}

void update(int x,int l,int r,int v){

int L = t[x].l, R = t[x].r;

if(l<=L && R<=r){

t[x].update(v);

return;

}

if(t[x].lz) push\_down(x);

if(l<=mid) update(lc,l,r,v);

if(r>mid) update(rc,l,r,v);

push\_up(x);

}

ll query(int x,int l,int r){

int L = t[x].l, R = t[x].r;

if(l<=L && R<=r) return t[x].sum;

if(t[x].lz) push\_down(x);

int res = 0;

if(l<=mid) res += query(lc,l,r);

if(r>mid) res += query(rc,l,r);

push\_up(x);

return res;

}

};

### 树状数组

struct BIT{

int size,t[N];

void init(int n){

size = n;

clr(t,0);

}

void add(int x,int v){

for(int i=x;i<=size;i+=(i&-i)) t[i] += v;

}

int sum(int x,int ans = 0){

for(int i=x;i;i-=(i&-i)) ans += t[i];

return ans;

}

int Kth(int k){

int ans = 0, cnt = 0;

for(int i=20;i>=0;i--){

ans += (1<<i);

if(ans>size||cnt+t[ans]>=k) ans -= (1<<i);

else cnt += t[ans];

}

return ans+1;

}

};

### 二叉树

struct node { //节点

int val;

node\* lc;

node\* rc;

node(int val): val(val),lc(NULL),rc(NULL){}

};

vector<int> pre;//前序

vector<int> level;//层次遍历

//前序遍历

void preorder(node\* root){

if(!root) return;

pre.push\_back(root -> val);

preorder(root -> lc);

preorder(root -> rc);

}

//BFS 层次遍历

void bfs(node\* root){

queue<node\* > Q;

node\* now = root;

Q.push(now);

int size;

while(Q.size()){

size = Q.size();//每一层的size

while(size--){

now = Q.front();Q.pop();

level.push\_back(now -> val);

if(now -> lc) Q.push(now -> lc);

if(now -> rc) Q.push(now -> rc);

}

}

}

//先序中序还原二叉树

node\* build(int preL, int preR, int inL, int inR){

if(preL > preR) return NULL;

node\* root = new node(pre[preL]);

int index;

for(index = inL;index <= inR;index++){

if(in[index] == pre[preL]) break;

}

int numleft = index - inL;

root -> lc = build(preL+1, preL+numleft, inL, index-1);

root -> rc = build(preL+numleft+1, preR, index+1, inR);

return root;

}

//中序后序还原二叉树

node\* build(int inL, int inR, int postL, int postR){

if(postL > postR) return NULL;

node\* root = new node(post[postR]);

int index;

for(index = inL;index <= inR;index++){

if(in[index] == post[postR]) break;

}

int numleft = index - inL;

root -> lc = build(inL, index-1, postL, postL+numleft-1);

root -> rc = build(index+1, inR, postL+numleft, postR-1);

return root;

}

### 二叉搜索树

struct node{

int val;

node\* lc;

node\* rc;

node(int val):val(val),lc(NULL),rc(NULL){}

};

//插入节点

node\* Insert(node\* root, int val) {

if(!root){

root = new node(val);

return root;

}

node\* now = root;

while(1){

if(root -> val > val){

if(root -> lc == NULL){

root -> lc = new node(val);

break;

}

root = root -> lc;

}

else{

if(root -> rc == NULL){

root -> rc = new node(val);

break;

}

root = root -> rc;

}

}

return now;

}

//删除节点

node\* Delete(node\* root, int key) {

if(!root) return NULL;

if(root -> val > key){

root -> lc = Delete(root -> lc,key);

}

else if(root -> val < key){

root -> rc = Delete(root -> rc,key);

}

else{

if(!root -> lc||!root -> rc){

root = (root -> lc) ? root -> lc : root -> rc;

}

else{

node\* now = root -> rc;

while(now -> lc)

now = now -> lc;

root -> val = now -> val;

root -> rc = Delete(root -> rc,now -> val);

}

}

return root;

}

### 线段树

ll n;

ll a[maxn];

struct node{

ll l,r;

ll sum,lazy;

void Update(ll val){

sum += (r-l+1)\*val;

lazy += val;

}

}tr[maxn<<2];

void Push\_up(ll pos){

ll tmp = pos<<1;

tr[pos].sum = tr[tmp].sum+tr[tmp+1].sum;

}

void Push\_down(ll pos){

ll tmp = pos<<1;

ll lazy = tr[pos].lazy;

tr[tmp].Update(lazy);

tr[tmp+1].Update(lazy);

tr[pos].lazy = 0;

}

void Build(ll pos,ll l,ll r){

tr[pos].l = l;tr[pos].r = r;

tr[pos].lazy = 0;

if(l==r){

tr[pos].sum = a[l];

return;

}

ll mid = (l+r)>>1,tmp = pos<<1;

Build(tmp,l,mid);

Build(tmp+1,mid+1,r);

Push\_up(pos);

}

void Update(ll pos,ll l,ll r,ll val){

ll L = tr[pos].l,R = tr[pos].r;

if(l<=L&&R<=r){

tr[pos].Update(val);

return;

}

if(tr[pos].lazy) Push\_down(pos);

ll mid = (L+R)>>1,tmp = pos<<1;

if(l<=mid) Update(tmp,l,r,val);

if(r>mid) Update(tmp+1,l,r,val);

Push\_up(pos);

}

ll Query(ll pos,ll l,ll r){

ll L = tr[pos].l,R = tr[pos].r;

if(l<=L&&R<=r){

return tr[pos].sum;

}

if(tr[pos].lazy) Push\_down(pos);

ll mid = (L+R)>>1,tmp = pos<<1;

ll ans = 0;

if(l<=mid) ans += Query(tmp,l,r);

if(r>mid) ans += Query(tmp+1,l,r);

return ans;

}

### 离散化

int n,a[maxn],b[maxn];

void dis(){

memcpy(b,a,sizeof(a));

sort(b+1,b+n+1);

int size = unique(b+1,b+n+1)-(b+1);

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

a[i] = lower\_bound(b+1,b+size+1,a[i])-(b+1)+1;

}

}

### 树状数组

int n,tr[maxn];

int lowbit(int i){

return i&-i;

}

//单点更新

void add(int i,int val){

while(i<=n){

tr[i] += val;

i += lowbit(i);

}

}

//前缀和

int sum(int i){

int ans = 0;

while(i>0){

ans += tr[i];

i -= lowbit(i);

}

return ans;

}

### 归并排序(逆序对)

ll a[N],t[N];

ll ans=0;//统计逆序对数

void Merge\_sort(int l,int r){

if(l>=r)return ;

int mid=(l+r)>>1,i=l,j=mid+1,k=l;

Merge\_sort(l,mid);Merge\_sort(mid+1,r);

while(i<=mid && j<=r){

if(a[i]<=a[j])t[k++]=a[i++];

else t[k++]=a[j++],ans+=mid-i+1;

}

while(i<=mid)t[k++]=a[i++];

while(j<=r)t[k++]=a[j++];

for(i=l;i<=r;i++)a[i]=t[i];

}

###### 字符串

**KMP**

//在字符串s中找p

string s, p;

int nxt[maxn];

void get\_next(string s){

int len = s.size();

nxt[0] = -1;

int k = -1,j = 0;

while(j<len){

if(k==-1 || s[j]==s[k]) nxt[++j] = ++k;

else k = nxt[k];

}

}

//KMP ,从位置s开始匹配,返回第一个正确的位置

int kmp\_Index(string s, string p, int st){

int n = s.size(), m = p.size();

int i = st, j = 0;

while(i<n){

if(j==m-1 && s[i]==p[j]) return i-j;

if(j==-1 || s[i]==p[j]) i++,j++;

else j = nxt[j];

}

return -1;

}

//KMP数个数，数s串中有多少个p

int kmp\_Count(string s, string p){

int n = s.size(), m = p.size();

int i = 0,j = 0,ans = 0;

while(i<n){

if(j==m-1 && s[i]==p[j]){

ans++,j++;

j = nxt[j];

}

if(j==-1 || s[i]==p[j]) i++,j++;

else j = nxt[j];

}

return ans;

}

**Manacher**

int len[maxn<<1];

char p[maxn];

int Manacher(){

char s[maxn<<1];

int n = strlen(p),l = 0;

s[l++] = '@';

s[l++] = '#';

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

s[l++] = p[i];

s[l++] = '#';

}

s[l++] = '~';s[l] = 0;

int mx = 0,pos = 0,ans = 0;

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

if(mx>i){

len[i] = min(len[2\*pos-i],mx-i);

}

else

len[i] = 1;

while(s[i+len[i]]==s[i-len[i]])

len[i]++;

ans = max(ans,len[i]);

if(len[i]+i > mx){

mx = len[i]+i;

pos = i;

}

}

return ans-1;

}

###### DP

**01背包**

v[maxn],w[maxn],dp[maxn]

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

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

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

}

}

**完全背包**

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

for(j=w[i];j<=m;j++){

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

}

}

**多重背包**

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

for(j=1;j<=c[i];j=j<<1){

w0=j\*w[i];

v0=j\*v[i];

for(k=W;k>=w0;k--){

bp[k]=max(bp[k],bp[k-w0]+v0);

}

c[i]=c[i]-j;

}

w0=c[i]\*w[i];

v0=c[i]\*v[i];

for(k=W;k>=w0;k--){

bp[k]=max(bp[k],bp[k-w0]+v0);

}

}

###### 数学

**线性筛求欧拉函数**

int phi[N], pri[N], isp[N];

void getphi(){

int tot = 0;

phi[1] = 1;

for(int i=1;i<N;i++) isp[i] = 1;

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

if(isp[i]) pri[++tot] = i, phi[i] = i-1;

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

int x = pri[j];

if(i\*x>=N) break;

isp[i\*x] = 0;

if(i%x==0) {phi[i\*x] = phi[i] \* x; break;}

else phi[i\*x] = phi[i] \* phi[x];

}

}

}

**扩展欧几里德**

void exgcd(ll a,ll b,ll& d,ll& x,ll& y)

{

if(!b) { d = a; x = 1; y = 0; }

else{ exgcd(b, a%b, d, y, x); y -= x\*(a/b); }

}

**乘法逆元**

ll inv(ll a, ll p)

{

ll d, x, y;

exgcd(a, p, d, x, y);

return d == 1 ? (x+p)%p : -1;

}

**快速幂**

ll pow\_mod(ll a,ll n,ll m){

if(n==0) return 1;

ll x = pow\_mod(a,n/2,m);

ll ans = x\*x%m;

if(n%2==1) ans = ans\*a%m;

return ans;

}

**GCD&LCM**

ll gcd(ll a,ll b){return b?gcd(b,a%b):a;}

ll lcm(ll a,ll b){return a/gcd(a,b)\*b;}

**等比数列求和**

ll Geo(ll head, ll comm, ll ns, ll mod){

if(ns==1) return head;

if(comm==1) return ((head%mod)\*(ns%mod))%mod;

return head\*((pow\_mod(comm,ns,mod)-1)\*(inv(comm-1,mod)%mod))%mod;

}

**素数筛**

int prime[maxn];

bool is\_prime[maxn];

int sieve(int n){

int p = 0;

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

is\_prime[i] = true;

is\_prime[0] = is\_prime[1] = false;

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

if(is\_prime[i]){

prime[p++] = i;

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

is\_prime[j] = false;

}

}

return p; // 返回素数个数

}

**约数个数定理**

ll factor\_num(ll n){

if(is\_prime[n]) return 2;

ll ans = 1, t = n, ic, tmp;

for(ll pos = 0;prime[pos]<=t;pos++){

ic = 0,tmp = prime[pos];

while(n%tmp==0){

n/=tmp;

ic++;

}

ans\*=ic+1;

}

return ans;

}

**约数和**

ll factor\_sum(ll n){

ll ans = 1, t = n, ic, tmp;

for(ll pos = 0;prime[pos]<=t;pos++){

ic = 0,tmp = prime[pos];

while(n%tmp==0){

n/=tmp;

ic++;

}

if(ic>0) ans = ans \* Geo(1,tmp,ic+1,mod) % mod;

}

return ans;

}

**组合数** Cmn n>=m m在上,n在下

ll C(ll n,ll m){

if(m>n-m) m = n-m;

ll res = 1;

for(ll i=1,j=n;i<=k;i++,j--){

res = ((res\*j)%mod)\*inv(i,mod)%mod;

}

return res;

}