**ACM简陋模板**

**2018.5.22**

1.1头文件

2.1 最短路

2.2最小生成树

2.3网络流

2.4 其他的图算法

3.1线段树

4.1 DP二分

5.1 数论

5.2素数

6.1计算几何

7.1字符串

8.1 STL及骚操作

**1.1头文件**

#include<cstdio>

#include<iostream>

#include<queue>

#include<cmath>

#include<cstring>

#include<string>

#include<map>

#include<sstream>

#include<algorithm>

#define pb(x) push\_back(x)

#define fir first

#define sec second

#define mem(a,x) memset(a,x,sizeof(a))

typedef long long ll;

using namespace std;

const int INF=0x3f3f3f3f;

//freopen("data.in","r",stdin);

//freopen("data.out","w",stdout);

//ios\_base::sync\_with\_stdio(false);cin.tie(NULL);cout.tie(NULL);

**2.1最短路**

/\*起点标号均已改为从1开始\*/

/\*\*\*\*\*Dijkstra POJ - 2387\*\*\*\*\*/

/\*注意重复边的输入情况\*/

int cost[maxn][maxn];

int dis[maxn];

int vis[maxn];

void dij(int x){

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

vis[x]=1;

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

dis[i]=cost[x][i];

dis[x]=0;

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

int min=INF;int u=-1;

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

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

min=dis[j];u=j;

}

}

vis[u]=1;

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

if(!vis[j]&&cost[u][j]+dis[u]<dis[j])

dis[j]=cost[u][j]+dis[u];

}

}

}

/\* Dijkstar 算法+堆优化 使用优先队列优化，复杂度 O (E log E) \*/

const int INF=0x3f3f3f3f;

const int maxn=1110;

int d[maxn];

int n;

typedef pair<int,int>P;

struct node{

int to;int cost;

};

vector<node>g[maxn];

void dij(){

memset(d,0x3f,sizeof(d));

d[1]=0;

priority\_queue< P,vector<P>,greater<P> >que;

que.push(P(0,1));

while(!que.empty()){

P a=que.top();que.pop();

int u=a.second;

//if(d[u]<a.first) continue; optimize in cf 938d

for(int j=0;j<g[u].size();j++){

node e=g[u][j];

if(d[e.to]>e.cost+d[u]) {

d[e.to]=e.cost+d[u];que.push(P(d[e.to],e.to));

}

}

}}

/\*\*\*\*Bellman-Ford\*\*\*\*/

/\*单源负环\*/

/\*o(nm)\*/

struct edge{

int s,e,w

}edges[maxm];

int dis[maxn];

void bell(int x){

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

dis[i]=INF;

dis[x]=0;

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

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

if(edges[i].w!=INF&&dis[edges[i].e]>dis[edges[i].s]+edges[i].w)

dis[edges[i].e]=dis[edges[i].s]+edges[i].w;

}

}

}

/\*再判断有无负环回路\*/

/\*返回1则存在负环回路\*/

struct edge{

int s,e,w

}edges[maxm];

int dis[maxn];

int bell(int x){

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

dis[i]=INF;

dis[x]=0;

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

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

if(edges[i].w!=INF&&dis[edges[i].e]>dis[edges[i].s]+edges[i].w){

dis[edges[i].e]=dis[edges[i].s]+edges[i].w;

if(k==n) return 1;/\*若第n次循环的时候路径还能在缩短，说明存在负环回路\*/

}

}

}

return 0;

}

/\*\*\*\*SPFA\*\*\*\*/

/\*若存在负环回路则返回1\*/

struct edge{

int to,cost;

edge(int To,int Cost):to(To),cost(Cost){};

};

vector<edge>edges[maxn];

int dis[maxn];

int vis[maxn]//标记每个点是否在队列里

int cnt[maxn]//判断是否存在负环回路;若有点更新超过n次，则存在负环

int spfa(int x){

memset(dis,0x3f,sizeof(dis));

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

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

dis[x]=0;

queue<int>que;

que.push(x);

vis[x]=1;cnt[x]=1;

while(!que.empty()){

int t=que.top;que.pop();vis[t]=0;

for(int i=0;i<edges[t].size;i++){

node e=edges[t][i];

if(dis[e.to]>dis[t]+e.cost){

dis[e.to]=dis[t]+e.cost;

if(!vis[e.to]){

que.push(e.to);

vis[e.to]=1;

if(++cnt[e.to]>n) return 1;

}

}

}

}

return 0;}

/\*\*\*\*floyd\*\*\*\*/

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

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

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

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

**2.2最小生成树**

/\*顶点均已改为从1开始\*/

/\*\*\*并查集\*\*\*\*/

void init(){

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

p[i]=i;/\*根节点初始化为自己\*/

}

int find(int x){

return x==p[x]?x:find(p[x]);

}

void unin(int x,int y){

x=find(x);y=find(y);

if(x!=y) p[x]=y;

}

/\*\*\*\*最小生成树Kruskal poj2421\*\*\*\*/

/\*时间复杂度取决于边数\*/

#define maxm 55//最多边数

#define maxn 11//最多顶点数

struct edge{

int u,v,w;

}edges[maxm];

int p[maxn];

int n,m;

void init(){

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

p[i]=i;

}

int find(int x){

return x==p[x]?x:find(p[x]);

}

void unin(int x,int y){

x=find(x);y=find(y);

if(x!=y) p[x]=y;

}

int cmp(edge a,edge b){

return a.w<b.w;

}

void kruskal(){

int sumweight=0;//生成树的总权值

int num=0;//已选用的边的数目

init();//初始化

sort(edges,edges+m,cmp);

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

int u=edges[i].u;

int v=edges[i].v;

int w=edges[i].w;

if(find(u)!=find(v)) {

unin(u,v);num++;sumweight+=w;

}

if(num==n-1) break;

}

}

/\*\*\*\*\*最小生成树Prim \*\*\*\*\*/

/\*时间复杂度取决于顶点数\*/

int cost[maxn][maxn];

int lowcost[maxn];

int prim(){

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

lowcost[i]=cost[0][i];

lowcost[1]=-1;

int sum=0;

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

int min=INF;int u;

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

if(lowcost[j]!=-1&&lowcost[j]<min){

min=lowcost[j];u=j;

}}

if(min==INF) return -1;

lowcost[u]=-1;

sum+=min;

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

if(lowcost[j]!=-1&&lowcost[j]>cost[u][j]){

lowcost[j]=cost[u][j];

}

}

}

return sum;

}

**2.3网络流**

/\*\*\*dinic算法 HDU - 1532\*\*\*\*/

/\*复杂度o(n^2\*m)\*/

struct edge{

int to,cap;

int rev;//反向边的序号

};

vector<edge>edges[maxn];

int level[maxn];

void addnode(int u,int v,int cap){

edges[u].push\_back((edge){v,cap,edges[v].size()});

edges[v].push\_back((edge){u,0,edges[u].size()-1});

}

void bfs(int s){

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

level[s]=0;

queue<int>que;

que.push(s);

while(!que.empty()){

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

for(int i=0;i<edges[u].size();i++){

edge e=edges[u][i];

if(e.cap>0&&level[e.to]==-1){

level[e.to]=level[u]+1;

que.push(e.to);}

}

}

}

int dfs(int u,int t,int f){

if(u==t) return f;

for(int i=0;i<edges[u].size();i++){

edge e=edges[u][i];

if(level[e.to]==level[u]+1&&e.cap>0){

int d=dfs(e.to,t,min(f,e.cap));

if(d>0){

edges[u][i].cap-=d;

edges[e.to][e.rev].cap+=d;

return d;

}

}

}

return 0;

}

int max\_flow(int s,int t){

int flow=0;

while(1){

bfs(s);

if(level[t]<0) return flow;

int f;

while((f=dfs(s,t,INF))>0)

flow+=f;

}

return flow;

}

**2.4其他图算法**

/\*\*求割点\*/

vector<int>edge[maxn];

int low[maxn],dfn[maxn],tot;

bool iscut[maxn];//判断是不是割点

void init(){

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

edge[i].clear();

mem(low);

mem(dfn);

mem(iscut);

tot = 0;

}

void dfs(int u,int fa){

low[u] = dfn[u] = ++tot;

int child=0;

for(int i = 0; i < edge[u].size(); i++){

int v = edge[u][i];

if(!dfn[v]){

dfs(v,u);

child++;

low[u] = min(low[u],low[v]);

if(low[v] >= dfn[u])

iscut[u] = true;

}

else if(v != fa){

low[u] = min(low[u],dfn[v]);

}

}

if(fa<0&&child == 1) iscut[u] = 0;

}

/\*强连通分量\*/

/\* Tarjan算法 \* 复杂度O(N+M)\*/

vector<int>edge[maxn];

stack<int>st;

int low[maxn],dfn[maxn];

int instack[maxn];

int tot;

int scc;//强连通分量个数

int belong[maxn];//记录每个点属于哪个连通分量

void init(){

mem(dfn);

mem(low);

mem(instack);

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

edge[i].clear();

while(!st.empty())

st.pop();

tot=scc=0;

}

void tar(int u){

dfn[u]=low[u]=++tot;

st.push(u);

instack[u]=1;

for(int i=0;i<edge[u].size();i++){

int v=edge[u][i];

if(!dfn[v]){

tar(v);

if(low[u]>low[v])

low[u]=low[v];

}

else if(instack[v]&&low[u]>dfn[v])

low[u]=dfn[v];

}

if(low[u]==dfn[u]) {

int v;

scc++;

do{

v=st.top();

st.pop();

belong[v]=scc;

instack[v]=0;

}while(v!=u);

}

}

/\*\*倍增lca\*/

const int maxn=40010;

struct node{

int t,val;

};

int n;

int vis[maxn];

int fa[maxn][40];

int deep[maxn],d[maxn];

vector<node>edge[maxn];

int cnt;

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

// e.pb(node{u,v,val});

edge[u].push\_back(node{v,val});

edge[v].push\_back(node{u,val});

}

void dfs(int u){

vis[u]=1;

for(int i=0;i<edge[u].size();i++){

int v=edge[u][i].t;

if(!vis[v]) {

fa[v][0]=u;//should give fa[v][0] value

d[v]=d[u]+edge[u][i].val;

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

dfs(v);

}

}

}

void bz(){

for(int j=1;j<=30;j++)

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

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

}

int lca(int u,int v){

if(deep[u]<deep[v]) swap(u,v);

int dc=deep[u]-deep[v];

for(int i=0;i<30;i++){

if((1<<i)&dc)//move u to dc+u

u=fa[u][i];

}

if(u==v) return u;

for(int i=29;i>=0;i--){

if(fa[u][i]!=fa[v][i]){

u=fa[u][i];v=fa[v][i];

}

}

u=fa[u][0];//on the next level of lca,just move up one

return u;

}

/\*\*\*\*二分图最大匹配 匈牙利算法\*\*\*\*/

int line[50][50];

int used[50];//标记这条边有没有用过

int match[50];//标记右侧的点是否被匹配，以及匹配的是左侧哪个点

int nl;

int nr;

bool find(int x){

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

if(line[x][i]&&!used[i]){

used[i]=1;

if(match[i]==0||find(match[i])){

match[i]=x;

return true;

}

}

}

return false;

}

int hungarian()

{

int ans = 0;

memset(match,0,sizeof(match));

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

memset(used,0,sizeof(used));

if(find(i)) ans++;

}

return ans;

}

/\* \* 二分图匹配（Hopcroft-Carp算法）

复杂度O(sqrt(n)\*E) 邻接表存图 vector实现

vector先初始化，然后假如边uN 为左端的顶点数，使用前赋值(点编号0开始)

\*/

const int MAXN = 3000;

vector<int>G[MAXN];

int uN;

int Mx[MAXN],My[MAXN];

int dx[MAXN],dy[MAXN];

int dis;

bool used[MAXN];

bool SearchP() {

queue<int>Q;

dis = INF;

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

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

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

if(Mx[i] == -1) {

Q.push(i);

dx[i] = 0; }

while(!Q.empty()) {

int u = Q.front();

Q.pop();

if(dx[u] > dis)

break;

int sz = G[u].size();

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

int v = G[u][i];

if(dy[v] == -1) {

dy[v] = dx[u] + 1;

if(My[v] == -1)dis = dy[v];

else {

dx[My[v]] = dy[v] + 1;

Q.push(My[v]);

} } } }

return dis != INF;

}

bool DFS(int u) {

int sz = G[u].size();

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

int v = G[u][i];

if(!used[v] && dy[v] == dx[u] + 1) {

used[v] = true;

if(My[v] != -1 && dy[v] == dis)

continue;

if(My[v] == -1 || DFS(My[v])) {

My[v] = u;

Mx[u] = v;

return true;

} } }

return false; }

int MaxMatch() {

int res = 0;

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

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

while(SearchP()) {

memset(used,false,sizeof(used));

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

if(Mx[i] == -1 && DFS(i))

res++;

}

return res; }

/\*\*拓扑排序\*\*/

queue<int>q;

//priority\_queue<int,vector<int>,greater<int>>q;

//优先队列的话，会按照数值大小有顺序的输出

//此处为了理解，暂时就用简单队列

int ind[maxn];//入度

int topo()

{

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

{

if(ind[i]==0)

{

q.push(i);

}

}

int temp;

while(!q.empty())

{

temp=q.front();

q.pop();

//此处可输出排序结果temp

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

{

if(edge[temp][i])

{

ind[i]--;

if(ind[i]==0) q.push(i);

}

}

}

}

/\*带权并查集（食物链) 抄网上的\*/

/\*n个动物 k句话 有一种循环a吃b 吃c c吃a 开始不知道n种动物关系是什么

两种询问：d=1 x y为同类 d=2 x吃y 判断假话条数（关键之违背之前的关系）

\*/

#include<cstdio>

#include<cstring>

#include<algorithm>

using namespace std;

const int Max=50010;

int fat[Max],ran[Max];

void Init(int n)//初始化重要

{

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

{

fat[i]=i;//初始化都是指向（看做）自己

ran[i]=0;//0同类 1吃父节点 2被父节点吃

}

return;

}

int Find(int x)//找寻父节点+路径压缩

{

if(x==fat[x])

return fat[x];

int y=Find(fat[x]);

ran[x]=(ran[x]+ran[fat[x]])%3;//递归后从祖先节点向后到每个孩子来计算

return fat[x]=y;//路径压缩

}

int Union(int typ,int x,int y)//区间并与查询

{

int x1=Find(x);

int y1=Find(y);

if(x1==y1)//共父节点才能判断出关系

{

if((ran[x]-ran[y]+3)%3==typ-1)

return 0;

return 1;

}

fat[x1]=y1;//连接两父节点

ran[x1]=(-ran[x]+typ-1+ran[y]+3)%3;//使用类似向量方法来计算权值,虽然题目只有两个，但是会出现被吃这种情况，所以要变成3种情况，注意一定要处理负数的情况

return 0;

}

int main()

{

int n,k,ans;

int typ,smt1,smt2;

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

Init(n);

ans=0;

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

{

scanf("%d %d %d",&typ,&smt1,&smt2);

if(smt1==smt2&&typ==2)

ans++;

else if(smt1>n||smt2>n)

ans++;

else

ans+=Union(typ,smt1,smt2);

}

printf("%d\n",ans);

return 0;

}

**3.1线段树**

/\*\*\*\*单点更新 区间查询 HDU - 1166\*\*\*\*/

int tree[maxn<<2];

void pushup(int rt){

tree[rt]=tree[rt\*2]+tree[rt\*2+1];

}

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

if(l==r) {scanf("%d",&tree[rt]);return;}

int mid=(l+r)/2;

build(l,mid,rt\*2);

build(mid+1,r,rt\*2+1);

pushup(rt);

}

int query(int l,int r,int ll,int rr,int rt){//l,r为当前边界

if(l>=ll&&r<=rr) return tree[rt];

int ans=0;int mid=(l+r)/2;

if(ll<=mid){

ans+=query(l,mid,ll,rr,rt\*2);

}

if(rr>mid){

ans+=query(mid+1,r,ll,rr,rt\*2+1);

}

return ans;

}

void update(int l,int r,int index,int add,int rt){

if(l==r) {

tree[rt]+=add;return;

}

int mid=(l+r)/2;

if(index<=mid){

update(l,mid,index,val,rt\*2);

}

else update(mid+1,r,index,val,rt\*2+1);

pushup(rt);

}

/\*\*\*\*区间更新 poj-3468\*\*\*\*/

int tree[maxn<<2];

int seg[maxn<<2];

void pushup(int rt){

tree[rt]=tree[rt\*2]+tree[rt\*2+1];

}

void pushdown(int len,int rt){

if(seg[rt]){

seg[rt\*2]+=seg[rt];

seg[rt\*2+1]+=seg[rt];

tree[rt\*2]+=(len-len/2)\*seg[rt];

tree[rt\*2+1]+=len/2\*seg[rt];

seg[rt]=0;

}

}

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

seg[rt]=0;

if(l==r) {

scanf("%d",&tree[rt]);return;}

int mid=(l+r)/2;

build(l,mid,rt\*2);

build(mid+1,r,rt\*2+1);

pushup(rt);

}

void update(int l,int r,int ll,int rr,int add,int rt){//l,r为当前区间，ll,rr为更新的区间

if(l>=ll&&r<=rr) {

seg[rt]+=add;tree[rt]+=(r-l+1)\*add;

return;

}

pushdown(r-l+1,rt);

int mid=(l+r)/2;

if(ll<=mid) {

update(l,mid,ll,rr,add,rt\*2);

}

if(rr>mid){

update(mid+1,r,ll,rr,add,rt\*2+1);

}

pushup(rt);

}

int query(int l,int r,int ll,int rr,int rt){//ll,rr为查询的区间

if(l>=ll&&r<=rr){

return tree[rt];

}

int ans=0;

pushdown(r-l+1,rt);

int mid=(l+r)/2;

if(ll<=mid) ans+=query(l,mid,ll,rr,rt\*2);

if(rr>mid) ans+=query(mid+1,r,ll,rr,rt\*2+1);

return ans;

}

**4.1 DP 二分**

/\*\*lower\_bound 找>=t的第一个元素\*/

int lb=-1,ub=res;

while(ub-lb>1){

int mid=(lb+ub)/2;

if(x[mid]>=t) ub=mid;

else lb=mid;

}

return ub;

/\*最长上升子序列o nlogn\*/

const int MAXN=500010; int a[MAXN],b[MAXN];

//用二分查找的方法找到一个位置，使得num>b[i-1] 并且num<b[i],并用num代替b[i]

/\*二分 upper\_bound\*/

int Search(int num,int low,int high) {

int mid;

while(low<=high) {

mid=(low+high)/2;

if(num>=b[mid]) low=mid+1;

else high=mid-1;

}

return low;

}

int DP(int n) {

int i,len,pos;

b[1]=a[1];

len=1;

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

if(a[i]>=b[len])//如果a[i]比b[]数组中最大还大直接插入到后面即可

{

len=len+1; b[len]=a[i]; }

else//用二分的方法在b[]数组中找出第一个比a[i]大的位置并且让a[i]替代这个位置

{ pos=Search(a[i],1,len);

b[pos]=a[i];

}}

return len; }

/\*0-1背包\*/

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

for(int j=v;j>=c[i];j--)

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

/\*完全背包\*/

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

for(int j=c[i];j<=v;j++)

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

/\*

组合数： c(m,n)=c(m-1,n)+c(m-1,n-1);

\*/

/\*最长公共子序列\*/

void lcs(){

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

int n=strlen(s1);

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

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

{

if(i==0||j==0) {

dp[i][j]=0;}

else if(s1[i-1]==s2[j-1])

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

else {

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

}

}

**5.1数论**

/\*\*\*\*\*欧几里得辗转相除\*\*\*\*/

ll gcd(ll a,ll b){

if(a < b)

swap(a,b);

while(a % b){

ll r = a % b;

a = b;

b = r;

}

return b;

}

/\*\*\*\*拓展欧几里得\*\*\*\*/

/\*求ax+by=c的xy 其中c%gcd(a,b)==0时才有解

解出的为特解x0,y0 \*/

ll ex\_gcd(ll a,ll b,ll &x,ll &y)

{

if(a==0&&b==0) return -1;//无最大公约数

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

ll d=ex\_gcd(b,a%b,y,x);

y-=a/b\*x;

return d;

}

/\*求最小正整数解x;若无解返回false\*/

/\*已求出特解 x0,y0 设t=c/d,则通解为 x=t\*x0+b/d\*k; y=t\*y0-a/d\*k; k为任意整数

最小正整数解 x=x0\*t; x=(x%s+s)%s;\*/

bool min\_x(int a,int b,int &x,int &y,int c)

{

int d=ex\_gcd(a,b,x,y);

if(c%d) return false;

x=x\*c/d;

int s=b/d;

s=s>0?s:-1\*s;

x=(x%s+s)%s;

y=(c-a\*x)/b;

return true;

}

/\*\*\*\*求逆元素\*\*\*\*/

/\*ax = 1(mod n) 即求ax+ny=1中x的解\*/

ll mod\_reverse(ll a,ll n) {

ll x,y;

ll d=ex\_gcd(a,n,x,y);

if(d==1) return (x%n+n)%n;

else return -1;

}

/\*简洁写法 求ax = 1( mod m) 的x值 要求 0<a<m 且a,m互质 \*/

ll inv(ll a,ll m){

if(a == 1) return 1;

return inv(m%a,m)\*(m-m/a)%m;

}

/\*\*\*\*费马小定理求逆元\*\*\*\*/

/\*mod为素数,而且a和m互质 a^(p-1)=1(mod p) \*/

ll inv(ll a,ll mod) {

return pow\_m(a,mod-2,mod);

}

/\*中国剩余定理\*/

/\*x=bi(mod m0i)\*/

/\*x0+=biMiyi\*/

ll ChinaRemainder(ll m0[],ll b[],int n){

ll m=1,a=0;s

int i;

for (i=1; i<=n; i++) m=m\*m0[i];

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

ll MM=m/m0[i];

ll x=mod\_reverse(MM,m0[i]);

a=(a+MM\*x\*b[i]) % m;

}

return a;

}

/\*\*\*\*快速幂\*\*\*\*/

ll quickmod(ll a,ll b,ll c){

ll res=1;

while(b){

if(b&1)

res=res\*a%c;

a=a\*a%c;

b>>=1;

}

return res;

}

/\*\*\*\*矩阵快速幂（摘）\*\*\*\*/

typedef vector<int> vec;

typedef vector<vec> mat;

const int MOD=10000;

//计算A\*B

mat mul(mat &A,mat &B){

mat C(A.size(),vec(B[0].size()));

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

for(int k=0;k<B.size();k++)

for(int j=0;j<B[0].size();j++)

C[i][j]=(C[i][j]+A[i][k]\*B[k][j])%MOD;

return C;

}

//计算 A^n

mat pow(mat A,ll n){

mat B(A.size(),vec(A.size()));

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

B[i][i]=1;

while(n>0){

if(n&1) B=mul(B,A);

A=mul(A,A);

n>>=1;

}

return B;

}

//输入

ll n;

void solve(){

mat A(2,vec(2));//行数，列数

A[0][0]=1;A[0][1]=1;

A[1][0]=1;A[1][1]=0;

A=pow(A,n);

cout<<A[1][0]<<endl;

}

/\*\*\*抄邝斌 筛法欧拉函数\*\*\*/

/\*

1. i mod p==0 phi(i \* p) == p \* phi(i)

2. i mod p!=0 phi(i \* p) == phi(i) \* (p-1)

算法就是将 X 中的素数因子pi减1，并且相同的素数因子只有1个减1

\*/

int euler[3000001];

void getEuler() {

memset(euler,0,sizeof(euler));

euler[1] = 1;

for(int i = 2;i <= 3000000;i++)

if(!euler[i])

for(int j = i;j <= 3000000; j += i){

if(!euler[j])

euler[j] = j;

euler[j] = euler[j]/i\*(i-1);

}

}

/\*求单个数的欧拉函数\*/

/\*

phi(n)=n\*((u(d)/d)之和d|n)

\*/

ll eular(ll n) {

ll ans = n;

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

if(n % i == 0) {

ans -= ans/i;

while(n % i == 0)

n /= i; } }

if(n > 1)ans -= ans/n;

return ans;

}

**5.2素数**

/\*\*\*\*一个比较快的素数判断\*\*\*\*/

/\*可以判断一个 < 2^63 的数是不是素数 \*/

/\*用法：若 miller\_rabin(n)为1，则n为素数 \*/

ll mul(ll x,ll y,ll Z)//大数相乘取模

{

ll tmp=x/(long double)Z\*y+1e-3;

return (x\*y+Z-tmp\*Z)%Z;

}

ll MUL(ll x,ll p,ll Z)

{

ll y=1;

while(p)

{

if(p&1)y=mul(y,x,Z);

x=mul(x,x,Z);

p>>=1;

}

return y;

}

bool miller\_rabin(ll n)

{

if(n<=1)return 0;

if(n==2)return 1;

if(n%2==0)return 0;

ll p=n-1;

srand(time(NULL));

int TIMES=8;

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

{

ll x=rand()%(n-1)+1;

if(MUL(x,p,n)!=1)return 0;

}

return 1;

}

/\*\*\*\*素数打表\*\*\*\*/

/\*notprime为true表示不是素数 \*/

const int MAXN=1000010;

bool notprime[MAXN];

void init() {

memset(notprime,false,sizeof(notprime));

notprime[0]=notprime[1]=true;

for(int i=2;i<MAXN;i++)

if(!notprime[i]) {

if(i>MAXN/i)continue;//防止溢出,或者i,j用long long

for(int j=i\*i;j<MAXN;j+=i)

notprime[j]=true;}

}

/\*\*素数筛选打表hash\*\*/

const int MAXN=10000010;

bool has[MAXN];//i是不是素数

int prime[MAXN/10];//求第i个素数

void getPrime()

{

memset(prime,0,sizeof(prime));

memset(has,false,sizeof(has));

for(int i=2;i<=MAXN;i++)

{

if(!has[i]) prime[++prime[0]]=i;

for(int j=1;j<=prime[0]&&prime[j]<=MAXN/i;j++)

{

has[prime[j]\*i]=true;

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

break;

}

}

}

/\*抄邝斌\*/

/\*\*\*\*素数筛选和合数分解 \*\*\*\*/

/\* 素数筛选

prime数组存小于等于MAXN的素数 prime[0] 存的是素数的个数 \*/

const int MAXN=10000;

int prime[MAXN+1];

void getPrime() {

memset(prime,0,sizeof(prime));

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

if(!prime[i])

prime[++prime[0]]=i;

for(int j=1;j<=prime[0]&&prime[j]<=MAXN/i;j++) {

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

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

} } }

//factor[0]中存质因数 factor[1]中存质因数的个数

long long factor[100][2];

int fatCnt;

int getFactors(long long x) {

fatCnt=0;

long long tmp=x;

for(int i=1;prime[i]<=tmp/prime[i];i++) {

factor[fatCnt][1]=0;

if(tmp%prime[i]==0){

factor[fatCnt][0]=prime[i];

while(tmp%prime[i]==0){

factor[fatCnt][1]++;

tmp/=prime[i]; }

fatCnt++;

} }

if(tmp!=1) {

factor[fatCnt][0]=tmp;

factor[fatCnt++][1]=1; }

return fatCnt;

}

**6.1计算几何**

const double eps = 1e-8;

const double PI = acos(-1.0);

int sgn(double x) {/\*判断x的正负性\*/

if(fabs(x) < eps) return 0;

if(x < 0)return -1;

else return 1;

}

/\*\*\*\*Point 定义\*\*\*\*\*/

struct Point {

double x,y;

Point(){};

Point(double \_x,double \_y){x=\_x;y=\_y;}

Point operator- (const Point &b){

return Point(x-b.x,y-b.y);

}

double operator^ (const Point &b){

return x\*b.y-y\*b.x;

}

double operator\*(const Point &b){

return x\*b.x+y\*b.y;

}

};

/\*判断点c是否在线段ab上 已验证\*/

bool onseg(Point a,Point b,Point c){

if(sgn((b-a)^(c-a))==0&&

sgn((c.x-a.x)\*(c.x-b.x))<= 0

&& sgn((c.y-a.y)\*(c.y-b.y))<= 0)

return 1;

return 0;

}

/\*\*\*\*Line 定义\*\*\*\*/

struct Line{

Point s,t;

Line(){};

Line(Point \_s,Point \_t){s=\_s;t=\_t;}

};

/\*判断线段是否相交 51Nod - 1264 已验证\*/

/\*为1则相交\*/

bool inter(Line l1,Line l2){

if(sgn((l1.s-l2.s)^(l1.s-l1.t))\*sgn((l1.s-l1.t)^(l1.s-l2.t))<0)

return 0;

if(sgn((l2.s-l1.s)^(l2.s-l2.t))\*sgn((l2.s-l2.t)^(l2.s-l1.t))<0)

return 0;

if(min(l1.s.x,l1.t.x)<=max(l2.s.x,l2.t.x)&&min(l1.s.y,l1.t.y)<=max(l2.s.y,l2.t.y)

&&min(l2.s.x,l2.t.x)<=max(l1.s.x,l1.t.x)&&min(l2.s.y,l2.t.y)<=max(l1.s.y,l1.t.y))

return 1;

return 0;

}

/\*计算多边形面积\*/

double calarea(Point p[],int n){

double res=0;

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

res+=(p[i]^p[(i+1)%n])/2;

}

return fabs(res);

}

int gcd ( int a,int b){

if(b==0) return a;

else return gcd(b,a%b);

}

/\*多边形边上整点数\*/

int onedge(Point p[],int n){

int ans=0;

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

ans+=gcd(fabs(p[(i+1)%n].x-p[i].x),fabs(p[(i+1)%n].y-p[i].y));

}

return ans;

}

/\*多边形内部整点数 s（面积）=l/2（边上整点数/2）+n(内部整点数）-1\*/

int inside(Point p[],int n){

return calarea(p,n)+1-onedge(p,n)/2;

}

/\*点在多边形内 ZOJ - 1081 已验证\*/

/\*射线法，poly[]的顶点数要大于等于3,点的编号0~n-1\*/

/\*返回值 -1:点在凸多边形外 0:点在凸多边形边界上 1:点在凸多边形内\*/

int inPoly(Point p,Point poly[],int n){

Point p1(-INF+100,p.y);

Line ray(p,p1);

Line side;

int cnt=0;

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

side.s=poly[i];

side.t=poly[(i+1)%n];

if(onseg(side.s,side.t,p)) return 0;

if(sgn(side.s.y-side.t.y)==0) continue;//平行轴不考虑

if(onseg(ray.s,ray.t,side.s)){

if(sgn(side.s.y-side.t.y)>0) cnt++;}

else if(onseg(ray.s,ray.t,side.t)){

if(sgn(side.t.y-side.s.y)>0) cnt++;}

else if(inter(side,ray)) cnt++;

}

if(cnt%2) return 1;

return -1;

}

/\*判断凸多边形\*/

/\*点的编号1~n-1 凸边形为true\*/

bool isconvex(Point poly[],int n) {

bool s[3];

memset(s,false,sizeof(s));

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

s[sgn( (poly[(i+1)%n]-poly[i])^(poly[(i+2)%n]-poly[i]) )+1] = true;

if(s[0] && s[2])

return false; }

return true; }

double dissq(Point a,Point b){

return (a.x-b.x)\*(a.x-b.x)+(a.y-b.y)\*(a.y-b.y);

}

/\*求凸包，Graham算法\*/

/\*返回凸包结果st[0~top-1]为凸包边上的点编号 top为凸包边上的点的总数\*/

Point lis[maxn];

int st[maxn],top;

bool cmp(Point a,Point b){

int flag=sgn((a-lis[0])^(b-lis[0]));

if(flag>0) return 1;

if(flag==0&&sgn(dissq(a,lis[0])-dissq(b,lis[0]))<=0) return 1;

return 0;

}

void graham(int n){

Point p0;

p0=lis[0];int k=0;

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

if((p0.y>lis[i].y)||(p0.y==lis[i].y&&p0.x>lis[i].x)){

p0=lis[i];

k=i;

}

}

lis[k]=lis[0];

lis[0]=p0;

sort(lis+1,lis+n,cmp);

if(n==1){

st[0]=0;

top=1;

return;

}

if(n==2){

st[0]=0;

st[1]=1;

top=2;

return;

}

st[0]=0;

st[1]=1;

top=2;

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

while(top>1&&sgn((lis[st[top-1]]-lis[st[top-2]])^(lis[i]-lis[st[top-2]]))<=0)

top--;

st[top++]=i;

}

}

**7.1字符串**

/\*\*\*\*kmp(手码未验证版）\*\*\*\*/

const int maxn=100;

int f[maxn];

void getft(const string &a){

memset(f,0,sizeof(f));

f[0]=-1;

int i=0,k=-1;

while(i<a.size()-1){

if(k==-1||a[i]==a[k])

f[++i]=f[++k];

else k=f[k];

}

}

int kmp(const string &t,const string &pat,int pos=0){

getft(pat);

int i=0,j=0;

while(i<t.size()&&j<pat.size()

&&pat.size()-j<=t.size()-i){

if(j==-1||t[i]==pat[j]){

i++;j++;

}

else{

j=f[j];

}

}

if(j<pat.size())

return -1;

else return i-j;

}

**8.1 STL及骚操作**

/\*c++格式化 long double的输出\*/

long double t1 = (l - t2 \* v2) / v1;

cout << fixed << setprecision(10) << t1 + t2 << endl;//保留10位小数

/\*大小写转换函数\*/

a=tolower(a);

b=toupper(b);

/\*\*sstream\*\*/

string sstream

string str;

getline(cin,str);

stringstream ss(str);//对string对象进行读写

while(ss>>x)

s.compare(b);

ss.clear();//多次使用stringstream，要先清空下

/\*\*

stringstream可以用来把string类型的字符串转换成int

\*\*/

string s("12345");

int x;

stringstream ss(s);

ss>>x;

/\*\*法二\*\*/

string s("12345");

int x;

stringstream ss;

ss<<s;

ss>>x;

/\*\*将多种数值转换成字符串\*\*/

typename x=5.222;

cin>>x;

stringstream ss;

ss<<x;

string s;

s=ss.str(); //s="5.222"

/\*\*字符串处理\*/

char s[] = "a,b\*c,d";

const char \*sep = ",\*"; //可按多个字符来分割

char \*p;

p = strtok(s, sep);// 在第一次被调用的时间str是传入需要被切割字符串的首地址；在后面调用的时间传入NULL

while(p){

printf("%s ", p);

p = strtok(NULL, sep);

}

/\*取子串\*/

string sub1 = s.substr(5); //从下标为5开始一直到结尾

string sub2 = s.substr(5, 3); //从下标为5开始截取长度为3位

/\*algorithm\*/

sort(a,a+n);//n为数列长度，即a+n为最后一个数的位置+1

lower\_bound(a,a+n,x)-a;//在已排序数组a中寻找x ,查找"大于或者等于x的第一个位置”。

upper\_bound(a,a+n,x)-a;//返回第一个大于x的位置

vector<int>a a.push\_back(); //尾部添入元素

a.pop\_back();

a.clear(); a.empty(); a.size() a.resize();

/\*set\*/

set<string>dict; //自动从小到大排好序，每个元素至多出现一次

dict.insert(buf);

dict.erase(buf);

set<string>::iterator it;

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

cout<<\*it<<endl;

/\*\*map\*\*/

map<string,int>cnt;

cnt.size() cnt.count(r)//是否存在r

map<int, string>::iterator it;

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

{

cout<<it->first<<" "<<it->second<<endl;

}

it=mp.find(1);

if(it!=mp.end()){//它返回的一个迭代器;如果map中没有要查找的数据，它返回的迭代器等于end函数返回的迭代器

cout<<it->second()<<endl;

}

mp.count(1);//出现过则返回 1，未出现则返回0

/\*\*queue and stack\*\*/

queue<int>q;

priority\_queue<int>s;

priority\_queue<int，vector<int>，greater<int> >pq;//越小的整数优先级越大的优先队列

stack<int>st;

st.push();

/\*去重

unique(num,num+n)返回的是num去重后的尾地址，之所以说比不真正把重复的元素删除，

其实是，该函数把重复的元素一到后面去了，然后依然保存到了原数组中，

然后返回去重后最后一个元素的地址，因为unique去除的是相邻的重复元素，

所以一般用之前都会要排一下序。 \*/

sort(num,num+n);

n=unique(num,num+n)-num;

**小垃圾的补充部分：**

**树状数组**

/\*树状数组

只能用于求和，不能求最大／小值

不能动态插入

\*/

int lowbit(int t)

{

return t&(-t);

}

void add(int x,int y)

{

for(int i=x;i<=n;i+=lowbit(i))

tree[i]+=y;

}

int getsum(int x)

{

int ans=0;

for(int i=x;i>0;i-=lowbit(i))

ans+=tree[i];

return ans;

}

**二维线段树基本操作**

1、单点修改+矩阵查询

单次访问一个位置，查询一个矩阵的总访问次数

访问位置（x，y）

所有第一维包含x的都包含位置（x，y），访问次数都要加1

所以从根到找到第一维包含x所经过的所有节点，在里面找到第二维包含y的节点，访问次数加1

sum[i][j] 表示的是这个节点所包含的矩阵的所有位置的访问次数之和

第一维：

void changex(int kx,int l,int r)

{

changey(kx,1,1,h);

if(l==r) return;

int mid=l+r>>1;

if(x<=mid) changex(kx<<1,l,mid);

else changex(kx<<1|1,mid+1,r);

}

对于第二维的修改有两种写法

①、从根往下找经过的节点 f访问次数全部+1

因为此时已经确定了第一维包含，从根往下找y，所经过的区间一定包含y

void changey(int kx,int ky,int l,int r)

{

sum[kx][ky]++;

if(l==r) return;

int mid=l+r>>1;

if(y<=mid) changey(kx,ky<<1,l,mid);

else changey(kx,ky<<1|1,mid+1,r);

}

②、在第二维中找到y，访问次数+1，祖先节点通过左右子节点的合并修改

void changey(int kx,int ky,int l,int r)

{

if(l==r)

{

sum[kx][ky]++;

return;

}

int mid=l+r>>1;

if(y<=mid) changey(kx,ky<<1,l,mid);

else changey(kx,ky<<1|1,mid+1,r);

sum[kx][ky]=sum[kx][ky<<1]+sum[kx][ky<<1|1];

}

查询：

查询左上角为（xl，yl），右下角为（xr，yr）的矩阵的所有位置的访问次数之和

void queryy(int kx,int ky,int l,int r)

{

if(l>=yl && r<=yr)

{

cnt+=sum[kx][ky];

return;

}

int mid=l+r>>1;

if(yl<=mid) queryy(kx,ky<<1,l,mid);

if(yr>mid) queryy(kx,ky<<1|1,mid+1,r);

}

void queryx(int kx,int l,int r)

{

if(l>=xl && r<=xr)

{

queryy(kx,1,1,h);

return;

}

int mid=l+r>>1;

if(xl<=mid) queryx(kx<<1,l,mid);

if(xr>mid) queryx(kx<<1|1,mid+1,r);

}

2、矩阵修改+单点查询

修改

访问左上角为（x1，y1），右下角为（x2，y2）的矩阵

sum[i][j] 表示的是这个节点所代表的矩阵的整体访问次数

即这个矩阵的每一个位置都会有一个sum[i][j]的访问次数

void changey(int kx,int ky,int l,int r)

{

if(y1<=l&&r<=y2)

{

sum[kx][ky]++;

return;

}

int mid=l+r>>1;

if(y1<=mid) changey(kx,ky<<1,l,mid);

if(y2>mid) changey(kx,ky<<1|1,mid+1,r);

}

void changex(int kx,int l,int r)

{

if(x1<=l&&r<=x2)

{

changey(kx,1,1,n);

return;

}

int mid=l+r>>1;

if(x1<=mid) changex(kx<<1,l,mid);

if(x2>mid) changex(kx<<1|1,mid+1,r);

}

查询

查询位置（x，y）的访问次数

因为sum[i][j] 表示的是一个矩阵的整体访问次数，所以找（x，y）经过的所有点都要累计其访问次数

void queryy(int kx,int ky,int l,int r)

{

ans+=sum[kx][ky];

if(l==r) return;

int mid=ly+ry>>1;

if(y<=mid) queryy(kx,ky<<1,l,mid);

else queryy(kx,ky<<1|1,mid+1,r);

}

void queryx(int kx,int l,int r)

{

queryy(kx,1,1,n);

if(l==r) return;

int mid=l+r>>1;

if(x<=mid) queryx(kx<<1,l,mid);

else queryx(kx<<1|1,mid+1,r);

}

这两个sum[][] 含义不同，是为了适应不同的修改方式

一个是单点修改，一个是矩阵修改

**Java大数高精度亲情讲义**

**java输入输出架构（注意主类为Main）**

import java.util.Scanner;//输入架构

public class Main {

public static void main(String[] args) {

Scanner s=new Scanner(System.in);

int a,b;

a=s.nextInt();

b=s.nextInt();

System.out.println((a+b));

}

}

**java函数调用**

import java.util.Scanner;

public class Main {

public static int gcd(int a,int b)

{

if(b==0)

return a;

else

return gcd(b,a%b);

}

public static void main(String[] args) {

Scanner s=new Scanner(System.in);

int a,b,sum,i;

while(s.hasNext())

{

a=s.nextInt();

b=s.nextInt();

sum=1;

i=b;

while(i!=0)

{

int temp=gcd(a,b);

if(temp==1)

break;

else

{

sum\*=temp;

a/=temp;

}

i--;

}

System.out.print(sum+"\n");

}

}

}