Contest Library (Md. Sadman Sakib)

**Running Median (Streaming Median)**

Code:

priority\_queue<**int**, vector<**int**>, greater<**int**> > pqright;

priority\_queue<**int**, vector<**int**>, less<**int**> > pqleft;

**int** val1,val2;

**void** correct()

{

**int** szl,szr;

szl=pqleft.size();

szr=pqright.size();

**if**(szr>szl)

{

pqleft.push(pqright.top());

pqright.pop();

}

**else**

{

pqright.push(pqleft.top());

pqleft.pop();

}

**return**;

}

**double** Running\_Median(**int** idx,**int** point,**double** cur\_med) *// it will return the current\_median,you need to update your current\_median every time*

{

**if**(idx==0)

{

val1=point;

cur\_med=point;

}

**else** **if**(idx==1)

{

val2=point;

**if**(val1>val2) swap(val1,val2);

pqleft.push(val1);

pqright.push(val2);

cur\_med=(val1+val2)/2.0;

}

**else**

{

**if**(cur\_med<point) pqright.push(point);

**else** pqleft.push(point);

**if**(pqleft.size()==pqright.size())

{

cur\_med=(pqleft.top()+pqright.top())\*0.5;

}

**else**

{

**int** szleft,szright;

szleft=pqleft.size();

szright=pqright.size();

**if**(szleft+1==szright)

{

cur\_med=pqright.top();

}

**else** **if**(szright+1==szleft)

{

cur\_med=pqleft.top();

}

**else**

{

correct();

cur\_med=(pqleft.top()+pqright.top())\*0.5;

}

}

}

**return** cur\_med;

}

**Matrix Exponentiation:**

Code:

#define ll long long

ll MOD;

#define SZ 21

**struct** mat

{

ll r, c, g[SZ][SZ];

mat()

{

r = c = SZ;

memset(g,0,**sizeof**(g));

}

mat **operator** \* (mat b)

{

ll p;

mat res;

res.r = r ;

res.c = b.c;

**for**(**int** i=0; i<r; i++)

{

**for**(**int** j=0; j<b.c; j++)

{

p = 0;

**for**(**int** k=0; k<c; k++)

p= (p+(g[i][k]\*b.g[k][j]))%MOD;

res.g[i][j] = (p%MOD);

}

}

**return** res;

}

mat **operator** + (mat b) *// for square matrices*

{

mat res;

res.r = r;

res.c = c;

**for**(**int** i=0; i<r; i++)

{

**for**(**int** j=0; j<c; j++)

{

res.g[i][j]=(g[i][j]%MOD+b.g[i][j]%MOD)%MOD;

}

}

**return** res;

}

mat **operator** - (mat b) *// for square matrices*

{

mat res;

res.r = r;

res.c = c;

**for**(**int** i=0; i<r; i++)

{

**for**(**int** j=0; j<c; j++)

{

res.g[i][j]=(g[i][j]%MOD-b.g[i][j]%MOD)%MOD;

}

}

**return** res;

}

**void** Iden()

{

**for**(**int** i=0; i<20; i++)

{

**for**(**int** j=0; j<20; j++)

{

g[i][j] = (i==j);

}

}

}

};

mat bigMod(mat a, **int** n )

{

mat r,p;

r.Iden() ;

p = a;

**while**(n)

{

**if**(( n & 1 )==1) r = r \* p;

p = p \* p;

n >>= 1;

}

**return** r;

}

Sum of Geometric Series in O((log(n))^2)

Code:

#define ll long long

#define MOD 1000000000

*/// it calculates (base^1+base^2+base^3+base^4+.........+base^n)%MOD in O((log(n))^2) time.*

*/// if n==0 , you have to manually return/add 1 for that*

ll bigMod(ll a,ll b)

{

**if**(b==0) **return** 1LL;

**if**(b%2==0)

{

ll temp=bigMod(a,b/2)%MOD;

**return** (temp\*temp)%MOD;

}

**return** (a%MOD \* bigMod(a,b-1)%MOD)%MOD;

}

ll func(ll base,ll n)

{

**if**(n==1) **return** bigMod(base,n);

**if**(n%2==0)

{

ll temp=func(base,n/2);

**return** (temp+(bigMod(base,n/2)\*temp));

}

**return** (bigMod(base,n)+func(base,n-1));

}

**Chinese Remainder Theorem:**

Code:

#define ll long long

#define no\_eqn 20

ll modPow(ll a,ll b,ll MOD)

{

**if**(b==0) **return** 1LL;

**if**(b%2==0)

{

ll temp=modPow(a,b/2,MOD)%MOD;

**return** (temp\*temp)%MOD;

}

**return** (a\*modPow(a,b-1,MOD))%MOD;

}

ll modInv(ll a,ll b)

{

**return** modPow(a,b-2,b);

}

**struct** point

{

**int** val,m;

};

point ara[no\_eqn];

ll CRT(ll sz)

{

ll x=0; *// x=val[i](mod m[i])*

ll M=1;

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

{

M\*=ara[i].m;

}

ll Midx[14];

ll MInv[14];

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

{

Midx[i]=M/ara[i].m;

MInv[i]=modInv(Midx[i],ara[i].m);

}

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

{

x=(x+(ara[i].val\*Midx[i]\*MInv[i])%M)%M; *// x=val[i]\*Inv(Mi)\*(Mi) where Mi means (m0\*m1\*m2\*...\*mn)/mi;*

}

**return** x;

}

**LCA:**

1. K-th node from node to node .
2. K-th node from node
3. Lowest Common Ancestor of two given nodes and
4. Max Edge Value in a weighted tree from node to node .

Code:

#define MAX 10001

#define LOGN 14 // be VERY VERY CAREFUL while setting this!

#define pb push\_back

vector<**int**>gr[MAX+10];

vector<**int**>cost[MAX+10]; *// omit this for unweighted graph*

**bool** vis[MAX+10];

**int** level[MAX+10],khoroch[MAX+10];

**int** par[MAX+10][20],weight[MAX+10][20];

**int** node,a,b,c;

**void** dfs(**int** u)

{

vis[u]=**true**;

**for**(**int** i=0; i<gr[u].size(); i++)

{

**int** v=gr[u][i];

**if**(!vis[v])

{

par[v][0]=u;

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

khoroch[v]=khoroch[u]+cost[u][i]; *// omit this for unweighted graph*

dfs(v);

}

}

**return**;

}

**void** reset()

{

memset(par,-1,**sizeof**(par));

memset(vis,**false**,**sizeof**(vis));

memset(level,**false**,**sizeof**(level));

memset(khoroch,**false**,**sizeof**(khoroch)); *// omit this for unweighted graph*

**for**(**int** i=0; i<MAX; i++)

{

gr[i].clear();

cost[i].clear(); *// omit this for unweighted graph*

}

}

**void** makeSparseTable()

{

dfs(1);

**for**(**int** j=1; (1<<j)<node; j++)

{

**for**(**int** i=1; i<=node; i++)

{

**if**(par[i][j-1]!=-1)

{par[i][j]=par[par[i][j-1]][j-1]; *// par[i][j] is defined by : (2^j)-th parent of node i*

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

}

}

}

**return**;

}

**int** LCA(**int** a,**int** b)

{

**if**(a==b) **return** a;

**if**(par[a][0]==par[b][0]) **return** par[a][0];

**if**(level[b]>level[a]) swap(b,a);

**for**(**int** i=LOGN; i>=0; i--)

{

**if**(par[a][i]!=-1)

{

**if**(level[a]-(1<<i)>=level[b])

{

a=par[a][i];

}

}

}

**if**(a==b) **return** a;

**for**(**int** i=LOGN; i>=0; i--)

{

**if**(par[b][i]!=-1 && par[a][i]!=-1 && par[b][i]!=par[a][i])

{

b=par[b][i];

a=par[a][i];

}

}

**return** par[a][0];

}

**int** kthNode(**int** a1,**int** an,**int** kth)

{

**int** w = LCA(a1,an);

**int** d1,dn;

**int** from;

d1 = level[a1]-level[w]+1;

dn = level[an]-level[w]+1;

**if**(d1==kth) **return** w;

**else** **if**(d1>kth)

{

from = a1;

kth--;

}

**else**

{

from = an;

kth=d1+dn-kth-1;

}

**int** lg=LOGN;

**while**(kth>0 && lg>=0)

{

**if**((1<<lg)<=kth)

{

from = par[from][lg];

kth-=(1<<lg);

}

--lg;

}

**return** from;

}

**int** maxEdgeValue(**int** p, **int** q)

{

**int** tmp, logn,i, mx = 0;

**if**(level[p] < level[q]) swap(p, q);

**for**(logn = 1; (1 << logn) <= level[p]; logn++);

logn--;

**for**(i = logn; i >= 0; i--)

{

**if**(level[p] - (1 << i) >= level[q])

{

mx = max2(mx, weight[p][i]);

p = par[p][i];

}

}

**if**(p == q) **return** mx;

**for**(i = logn; i >= 0; i--)

{

**if**(par[p][i] != -1 && par[p][i] != par[q][i])

{

mx = max2(mx, max2(weight[p][i], weight[q][i]));

p = par[p][i];

q = par[q][i];

}

}

mx = max2(mx, max2(weight[p][0], weight[q][0]));

**return** mx;

}

**Shank’s Baby step – Giant step Algorithm:**

#include <tr1/unordered\_map>

**using** **namespace** std;

#define ll long long

ll modPow(ll a,ll b,ll p)

{

ll ans=1LL;

**while** (b > 0)

{

**if** (b & 1)

ans = (ans \* a) % p;

a = (a\*a) % p;

b >>= 1;

}

**return** ans;

}

ll modInvPow(ll a,ll b,ll p)

{

**return** modPow(modPow(a,p-2,p),b,p)%p;

}

ll ShanksAlgo(ll a,ll r,ll p)

{

tr1::unordered\_map<ll,**int**>mp;

ll sqr=sqrt(p);

ll tmp=modPow(a,0,p);

mp[tmp]=0;

**for**(**int** i=1; i<sqr; i++)

{

tmp=(tmp\*a)%p;

mp[tmp]=i;

}

tmp=r;

ll coef = modInvPow(a,sqr,p);

**for**(**int** i=0; i<sqr; i++)

{

**if**(mp.count(tmp))

{

**return** mp[tmp]+(i\*sqr);

}

**else**

{

tmp=(tmp\*coef)%p;

}

}

}