CODE LIBRARY

MD. SHAHADAT HOSSAIN SHAHIN

UNIVERSITY OF DHAKA

http://www.planetb.ca/syntax-highlight-word

http://remove-line-numbers.ruurtjan.com/

Template

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;

#define si(a)             scanf("%d",&a)
#define sii(a,b)          scanf("%d %d",&a,&b)
#define siii(a,b,c)       scanf("%d %d %d",&a,&b,&c)

#define sl(a)             scanf("%lld",&a)
#define sll(a,b)          scanf("%lld %lld",&a,&b)
#define slll(a,b,c)       scanf("%lld %lld %I64d",&a,&b,&c)

#define pb                push_back
#define PII               pair <int,int>
#define PLL               pair <ll,ll>
#define mp                make_pair
#define xx                first
#define yy                second
#define all(v)            v.begin(),v.end()

#define CLR(a)            memset(a,0,sizeof(a))
#define SET(a)            memset(a,-1,sizeof(a))

#define eps               1e-9
#define PI                acos(-1.0)
#define MAX               100010
#define MOD               1000000007
#define INF               2000000000

int setBit(int n,int pos){ return n = n | (1 << pos); } //sets the pos'th bit to 1
int resetBit(int n,int pos){ return n = n & ~(1 << pos); } //sets the pos'th bit to 0
bool checkBit(int n,int pos){ return (bool)(n & (1 << pos)); } //returns the pos'th bit
```

Graph

MST(Kruskal) :

```
struct edge{
    int u,v,c;
}ara[MAX];

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

int par[MAX];

int findParent(int u){
    while(par[u]!=u) u = par[u];
    return u;
}

/*
int findParent(int u){
    if(par[u]==u) return u;
    else return par[u] = findParent(par[u]);
}
*/

int kruskal(int n,int m){
    sort(ara+1,ara+m+1,cmp);
    int i,mst;
    mst = 0;
    for(i=1;i<=n;i++) par[i] = i;
    for(i=1;i<=m;i++){
        edge x = ara[i];
        par[x.u] = findParent(x.u);
        par[x.v] = findParent(x.v);
        if(par[x.u]!=par[x.v]){
            par[par[x.u]] = par[x.v];
            mst += x.c;
        }
    }
    return mst;
}
```

Dijkstra :

```
vector <int> ed[MAX],co[MAX];
int dis[MAX];
bool vis[MAX];

struct node{
    int city,cost;
};

bool operator < (node a,node b){return a.cost>b.cost;}

void dijkstra(int s,int n)
{
```

```
    CLR(vis);
    int i,x,u,v,c;
    node a,b;
    for(i=1;i<=n;i++) dis[i] = INF;
    dis[s] = 0;
    a.city = s;
    a.cost = 0;
    priority_queue <node> q;
    q.push(a);
    while(!q.empty()){
        a = q.top();
        q.pop();
        u = a.city;
        if(!vis[u]){
            vis[u] = true;
            for(i=0;i<ed[u].size();i++){
                v = ed[u][i];
                c = co[u][i];
                if(dis[v]>dis[u]+c){
                    dis[v] = dis[u]+c;
                    b.city = v;
                    b.cost = dis[v];
                    q.push(b);
                }
            }
        }
    }
}
```

Floyd Warshall :

```
int dis[MAX][MAX];

void warshall(int n){
    int i,j,k;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++) dis[i][j] = INF;

    for(k=0;k<n;k++){
        for(i=0;i<n;i++){
            for(j=0;j<n;j++){
                if(dis[i][k]!=INF && dis[k][j]!=INF && dis[i][k]+dis[k][j]<=dis[i][j])
                    dis[i][j] = dis[i][k]+dis[k][j];
            }
        }
    }
}
```

Bellman Ford :

```
int dis[MAX];
struct data{
    int u,v,c;
}edge[MAX];

void bellmanFord(int s,int e){
    int i,j;
    dis[s] = 0;
    for(j=1;j<=n-1;j++){
        for(i=1;i<=e;i++){
            if(dis[edge[i].u]!=INF && dis[edge[i].u]+edge[i].c<dis[edge[i].v]){
                dis[edge[i].v] = dis[edge[i].u]+edge[i].c;
            }
        }
    }
    bool cycle = false;
    for(i=1;i<=e;i++){
        if(dis[edge[i].u]!=INF && dis[edge[i].u]+edge[i].c<dis[edge[i].v]){
            cycle = true;
        }
    }
}
```

Articulation Point :

```
vector <int> edges[MAX];
bool vis[MAX] , isArt[MAX];
int st[MAX] , low[MAX] , Time = 0 , n;

void findArt(int s,int par){
    int i,x,child = 0;
    vis[s] = 1;
    Time++;
    st[s] = low[s] = Time;
    for(i=0;i< edges[s].size();i++){
        x = edges[s][i];
        if(!vis[x]){
            child++;
            findArt(x,s);
            low[s] = min(low[s],low[x]);
            if(par!=-1 && low[x]>=st[s]) isArt[s] = 1;
        }
        else{
            if(par!=x) low[s] = min(low[s],st[x]);
        }
    }
    if(par==-1 && child>1) isArt[s] = 1;
}

void processArticulation(){
    for(int i=1;i=<n;i++) if(!vis[i]) findArt(i,-1);
}
```

Bridge :

```
vector <int> ed[MAX];
vector <PII> res;
bool vis[MAX];
int st[MAX] , low[MAX] , Time = 0 , n;

void findBridge(int s,int par){
    int i,x;
    vis[s] = 1;
    Time++;
    st[s] = low[s] = Time;
    for(i=0;i<ed[s].size();i++){
        x = ed[s][i];
        if(!vis[x]){
            findBridge(x,s);
            low[s] = min(low[s],low[x]);
            if(low[x]>st[s]) res.pb(mp(s,x));
        }
        else{
            if(par!=x) low[s] = min(low[s],st[x]);
        }
    }
}

void processBridge(){
    for(int i=1;i<=n;i++) if(!vis[i]) findBridge(i,-1);
}
```

Strongly Connected Component (Kosaraju's Algorithm) :

```
/*
Step 1: Topsort All the nodes
Step 2: Run DFS from the unvisited nodes in topsorted order.
        This will mark the component related to the node.
*/

vector <int> edges[MAX],trans[MAX];
int compNum[MAX];
bool vis[MAX];
int cnum;
stack <int> topSortedNodes;
int n;

void topSort(int s){
    int i,x;
    vis[s] = 1;
    for(i=0;i<edges[s].size();i++){
        x = edges[s][i];
        if(!vis[x]) topSort(x);
    }
    topSortedNodes.push(s);
}

void markComponent(int s){
```

```
    int i,x;
    vis[s] = 1;
    compNum[s] = cnum;
    for(i=0;i<trans[s].size();i++){
        x = trans[s][i];
        if(!vis[x]) markComponent(x);
    }
}

void SCC(){
    int i,x;
    CLR(vis);
    for(int i=1;i<=n;i++){
        if(!vis[i]) topSort(i);
    }
    cnum = 0;
    CLR(vis);
    while(!topSortedNodes.empty()){
        x = topSortedNodes.top();
        topSortedNodes.pop();
        if(!vis[x]){
            cnum++;
            markComponent(x);
        }
    }
}
```

Biconnected Component :

```
/*
A graph is biconnected if every node is reachable from every other node even after removing a single node.
Algorithm of checking Biconnectivity :
    1) The graph is connected.
    2) There is no articulation point in the graph.
*/

/*
In the following code
bcc_counter --> Total number of biconnected components
bcc vector keeps the list of nodes in a single BCC.
*/


vector <int> edges[MAX];
bool vis[MAX] , isArt[MAX];
int Time;
int low[MAX],st[MAX];
vector <int> bcc[MAX];
int bcc_counter , n;
stack <int> S;

void findBCC(int s,int par){
    S.push(s);
    int i,x,child = 0;
    vis[s] = 1;
    Time++;
    st[s] = low[s] = Time;
    for(i=0;i< edges[s].size();i++){
```

```
        x = edges[s][i];
        if(!vis[x]){
            child++;
            findBCC(x,s);
            low[s] = min(low[s],low[x]);
            if(par!=-1 && low[x]>=st[s]){
                isArt[s] = 1;
                bcc[bcc_counter].pb(s);
                while(1){
                    bcc[bcc_counter].pb(S.top());
                    if(S.top()==x){
                        S.pop();
                        break;
                    }
                    S.pop();
                }
                bcc_counter++;
            }
            else if(par==-1){
                if(child>1){
                    isArt[s] = 1;
                    bcc[bcc_counter].pb(s);
                    while(1){
                        bcc[bcc_counter].pb(S.top());
                        if(S.top()==x){
                            S.pop();
                            break;
                        }
                        S.pop();
                    }
                    bcc_counter++;
                }
            }
        }
        else{
            if(par!=x) low[s] = min(low[s],st[x]);
        }
    }
    if(par==-1 && child>1) isArt[s] = 1;
}

void processBCC(){
    for(int i=0;i<n;i++){
        if(!vis[i]){
            Time = 0;
            findBCC(i,-1);
            bool lala = false;
            while(!S.empty()){
                lala = true;
                bcc[bcc_counter].push_back(S.top());
                S.pop();
            }
            if(lala) bcc_counter++;
        }
    }
}
```

```cpp
/*
In the following code
bcc_counter --> Number of BCCs
The code prints the edges of a single bcc serially
*/

vector <int> edges[MAX];
bool vis[MAX] , isArt[MAX];
int st[MAX] , low[MAX] , Time = 0;
stack <PII> S;
int n,bcc_counter;

void findBCC(int s,int par)
{
    int i,x,child = 0;
    vis[s] = 1;
    Time++;
    st[s] = low[s] = Time;
    for(i=0;i<edges[s].size();i++){
        x = edges[s][i];
        if(!vis[x]){
            S.push(mp(s,x));
            child++;
            findBCC(x,s);
            low[s] = min(low[s],low[x]);
            if(/*par!=-1 &&*/ low[x]>=st[s]){
                isArt[s] = 1;
                PII cur,e =  mp(s,x);
                bcc_counter++;
                cout << "Edges of Component " << bcc_counter << ":" << endl;
                do{
                    cur = S.top();
                    S.pop();
                    cout << cur.xx << "--" << cur.yy << endl;
                }while(cur!=e);
            }
        }
        else if(par!=x && st[x]<st[s]){
            S.push(mp(s,x));
            low[s] = min(low[s],st[x]);
        }
    }
    if(par==-1 && child>1) isArt[s] = 1;
}

void processBCC(){
    bcc_counter = 0;
    for(int i=0;i<n;i++){
        if(!vis[i]){
            findBCC(i,-1);
        }
    }
}
```

Data Structures

Segment Tree

```
/*
Segment tree with point update and range query
*/

int ara[MAX];

struct node{
    int sum;
}tree[4*MAX];

node Merge(node a,node b){
    node ret;
    ret.sum = a.sum+b.sum;
    return ret;
}

void build(int n,int st,int ed){
    if(st==ed){
        tree[n].sum = ara[st];
        return;
    }
    int mid = (st+ed)/2;
    build(2*n,st,mid);
    build(2*n+1,mid+1,ed);
    tree[n] = Merge(tree[2*n],tree[2*n+1]);
}

void update(int n,int st,int ed,int id,int v){
    if(id>ed || id<st) return;
    if(st==ed && ed==id){
        tree[n].sum = v;
        return;
    }
    int mid = (st+ed)/2;
    update(2*n,st,mid,id,v);
    update(2*n+1,mid+1,ed,id,v);
    tree[n] = Merge(tree[2*n],tree[2*n+1]);
}

node query(int n,int st,int ed,int i,int j){
    if(st>=i && ed<=j) return tree[n];
    int mid = (st+ed)/2;
    if(mid<i) return query(2*n+1,mid+1,ed,i,j);
    else if(mid>=j) return query(2*n,st,mid,i,j);
    else return Merge(query(2*n,st,mid,i,j),query(2*n+1,mid+1,ed,i,j));
}
```

```
/*
Segment tree with range update and range query
*/
int ara[MAX];

struct node{
    int sum;
}tree[4*MAX];

int lazy[4*MAX];

node Merge(node a,node b){
    node ret;
    ret.sum = a.sum+b.sum;
    return ret;
}

void lazyUpdate(int n,int st,int ed){
    if(lazy[n]!=0){
        tree[n].sum += ((ed-st+1)*lazy[n]);
        if(st!=ed){
            lazy[2*n] += lazy[n];
            lazy[2*n+1] += lazy[n];
        }
        lazy[n] = 0;
    }
}

void build(int n,int st,int ed){
    lazy[n] = 0;
    if(st==ed){
        tree[n].sum = ara[st];
        return;
    }
    int mid = (st+ed)/2;
    build(2*n,st,mid);
    build(2*n+1,mid+1,ed);
    tree[n] = Merge(tree[2*n],tree[2*n+1]);
}
void update(int n,int st,int ed,int i,int j,int v){
    lazyUpdate(n,st,ed);
    if(st>j || ed<i) return;
    if(st>=i && ed<=j){
        lazy[n] += v;
        lazyUpdate(n,st,ed);
        return;
    }
    int mid = (st+ed)/2;
    update(2*n,st,mid,i,j,v);
    update(2*n+1,mid+1,ed,i,j,v);
    tree[n] = Merge(tree[2*n],tree[2*n+1]);
}

node query(int n,int st,int ed,int i,int j){
    lazyUpdate(n,st,ed);
    if(st>=i && ed<=j) return tree[n];
    int mid = (st+ed)/2;
    if(mid<i) return query(2*n+1,mid+1,ed,i,j);
    else if(mid>=j) return query(2*n,st,mid,i,j);
    else return Merge(query(2*n,st,mid,i,j),query(2*n+1,mid+1,ed,i,j));
}
```

Implicit Segment Tree

```
/*
Point Update, Range Query
*/

struct node{
    int sum;
    node *left,*right;
    node(){}
    node(int value){
        sum = value;
        left = right = NULL;
    }
};

void update(node *cur,int st,int ed,int id,int v)
{
    if(id<st || id>ed) return;
    if(id==st && id==ed){
        cur->sum = v;
        return;
    }
    int mid = (st+ed)/2;
    if(cur->left==NULL) cur->left = new node(0);
    if(cur->right==NULL) cur->right = new node(0);
    update(cur->left,st,mid,id,v);
    update(cur->right,mid+1,ed,id,v);
    cur->sum = cur->left->sum + cur->right->sum;
}

int query(node *cur,int st,int ed,int i,int j)
{
    if(st>=i && ed<=j) return cur->sum;
    int mid = (st+ed)/2;
    if(cur->left==NULL) cur->left = new node(0);
    if(cur->right==NULL) cur->right = new node(0);
    if(mid<i) return query(cur->right,mid+1,ed,i,j);
    else if(mid>=j) return query(cur->left,st,mid,i,j);
    else return query(cur->right,mid+1,ed,i,j)+query(cur->left,st,mid,i,j);;
}

int main()
{
    int n = 1000000000;
    node *root = new node(0);
    update(root,1,n,5,1);
    update(root,1,n,3,1);
    cout << query(root,1,n,1,5) << endl;
    return 0;
}
```

```
/*
Range Update, Range Query
*/

struct node{
```

```cpp
    int sum,lazy;
    node *left,*right;
    node(){}
    node(int value){
        sum = value;
        lazy = 0;
        left = right = NULL;
    }
};

void lazyUpdate(node *cur,int st,int ed)
{
    if(cur->lazy!=0){
        cur->sum += ((ed-st+1)*cur->lazy);
        if(st!=ed){
            if(cur->left==NULL) cur->left = new node(0);
            if(cur->right==NULL) cur->right = new node(0);
            cur->left->lazy += cur->lazy;
            cur->right->lazy += cur->lazy;
        }
        cur->lazy = 0;
    }
}

void update(node *cur,int st,int ed,int i,int j,int v){
    lazyUpdate(cur,st,ed);
    if(st>j || ed<i) return;
    if(st>=i && ed<=j){
        cur->lazy += v;
        lazyUpdate(cur,st,ed);
        return;
    }
    int mid = (st+ed)/2;
    if(cur->left==NULL) cur->left = new node(0);
    if(cur->right==NULL) cur->right = new node(0);
    update(cur->left,st,mid,i,j,v);
    update(cur->right,mid+1,ed,i,j,v);
    cur->sum = cur->left->sum + cur->right->sum;
}

int query(node *cur,int st,int ed,int i,int j){
    lazyUpdate(cur,st,ed);
    if(st>=i && ed<=j) return cur->sum;
    int mid = (st+ed)/2;
    if(cur->left==NULL) cur->left = new node(0);
    if(cur->right==NULL) cur->right = new node(0);
    if(mid<i) return query(cur->right,mid+1,ed,i,j);
    else if(mid>=j) return query(cur->left,st,mid,i,j);
    else return query(cur->right,mid+1,ed,i,j)+query(cur->left,st,mid,i,j);;
}

int main()
{
    int n = 1000000000;
    node *root = new node(0);
    update(root,1,n,1,5,1);
    update(root,1,n,4,10,1);
    update(root,1,n,9,14,1);
    cout << query(root,1,n,1,20) << endl;
    return 0;
}
```

BIT

```
/*
Initially All the array elemets are zero
Point Update(Adding xx to index i)
Query returns sum from index 1 to index i
*/

int tree[MAX];

//n --> size
//x --> value to be added to index idx
void update(int idx, int x, int n){
    while(idx<=n){
        tree[idx]+=x;
        idx += idx & (-idx);
    }
}

int query(int idx){
    int sum=0;
    while(idx>0){
        sum += tree[idx];
        idx -= idx & (-idx);
    }
    return sum;
}
```

Mo's Algorithm

```
/*
Better to keep Query array 0 based
*/

#define MAX_SZ 100010
#define MAX_VAL 100010

int bs;//block size
int ara[MAX];
int cnt[MAX_VAL];
int res[SZ];
int ans;

struct data{
    int l,r,id,b;
    //b--> block size
}quer[MAX_SZ];

bool cmp(data a,data b){
    if(a.seg==b.seg) return a.r<b.r;
    return a.seg<b.seg;
}

void Add(int id){
    cnt[ara[id]]++;
    ///update ans
```

```
}

void Remove(int id){
    cnt[ara[id]]--;
    ///update ans
}

void Mo(int q)
{
    int L = 0, R = 0,l,r;
    ans = 0;
    Add(0);
    for(int i=0;i<q;i++){
        l = quer[i].l;
        r = quer[i].r;
        while(L>l){
            Add(L-1); L--;
        }
        while(L<l){
            Remove(L); L++;
        }
        while(R>r){
            Remove(R); R--;
        }
        while(R<r){
            Add(R+1); R++;
        }
        res[quer[i].id] = ans;
    }
}

int main()
{
    int q;
    sort(quer,quer+q,cmp);
    Mo(q);
    return 0;
}
```

Lowest Common Ancestor

```
#define lg 14

int L[MAX]; // Depth of a node
int T[MAX]; // Immediate Parent of a node
int P[MAX][lg+2]; // P[i][j] denotes (2^j)th parent of node i

void lca_build(int n){
    SET(P);
    int i,j;
    for(i=1;i<=n;i++) P[i][0] = T[i];
    for(j=1;(1<<j)<=n;j++)
        for(i=1;i<=n;i++)
            if(P[i][j-1]!=-1) P[i][j] = P[P[i][j-1]][j-1];
}

int lca_query(int x,int y){
```

```
    if(L[x]<L[y]) swap(x,y);

    int i,j;
    for(i=lg;i>=0;i--){
        if(L[x] - (1<<i) >= L[y]) x = P[x][i];
    }
    if(x==y) return x;
    for(i=lg;i>=0;i--){
        if(P[x][i]!=-1 && P[x][i]!=P[y][i]){
            x = P[x][i];
            y = P[y][i];
        }
    }
    return T[x];
}
```

Trie

```
#define TC 26

struct node{
    bool endmark;
    node *next[TC];
    node(){
        endmark = false;
        for(int i=0;i<TC;i++) next[i] = NULL;
    }
}*root;

void Insert(char *str,int len){
    node* cur = root;
    for(int i=0;i<len;i++){
        int id = str[i]-'a';
        if(cur->next[id]==NULL) cur->next[id] = new node();
        cur = cur->next[id];
    }
    cur->endmark = true;
}

bool Search(char *str,int len){
    node *cur = root;
    for(int i=0;i<len;i++){
        int id = str[i]-'a';
        if(cur->next[id]==NULL) return false;
        cur = cur->next[id];
    }
    return cur->endmark;
}

void Delete(node* cur){
    for(int i=0;i<TC;i++) if(cur->next[i]!=NULL) Delete(cur->next[i]);
    delete(cur);
}
```

Matrix Exponentiation

```
struct matrix{
    int mat[2][2];
    int dim;
    matrix(){};
    matrix(int d){
        dim = d;
        for(int i=0;i<dim;i++)
            for(int j=0;j<dim;j++)
                mat[i][j] = 0;
    }
    // mat = mat * mul
    matrix operator *(const matrix &mul){
        matrix ret = matrix(dim);
        for(int i=0;i<dim;i++){
            for(int j=0;j<dim;j++){
                for(int k=0;k<dim;k++){
                    ret.mat[i][j] += (mat[i][k])*(mul.mat[k][j]) ;
                    ret.mat[i][j] %= MOD ;
                }
            }
        }
        return ret ;
    }
    matrix operator + (const matrix &add){
        matrix ret = matrix(dim);
        for(int i=0;i<dim;i++){
            for(int j=0;j<dim;j++){
                ret.mat[i][j] = mat[i][j] + add.mat[i][j] ;
                ret.mat[i][j] %= MOD ;
            }
        }
        return ret ;
    }
    matrix operator ^(int p){
        matrix ret = matrix(dim);
        matrix m = *this ;
        for(int i=0;i<dim;i++) ret.mat[i][i] = 1 ; //identity matrix
        while(p){
            if( p&1 ) ret = ret * m ;
            m = m * m ;
            p >>= 1 ;
        }
        return ret ;
    }
    void print(){
        for(int i=0;i<dim;i++){
            for(int j=0;j<dim;j++){
                printf("%d ",mat[i][j]);
            }
            printf("\n");
        }
    }
};
```

LCS(n*log(n))

```
/*
The size of the vector after each iteration denotes the size of the LCS of the sub array starting at 1 and ending at i
*/

int ara[MAX];
vector <int> v;
int max_lcs = 0;
for(i=1;i<=n;i++){
    x = lower_bound(all(v),ara[i])-v.begin();
    if(x==0){
        if(v.size()==0) v.pb(ara[i]);
        else v[0] = ara[i];
    }
    else if(x==v.size()) v.pb(ara[i]);
    else if(ara[i]<v[x]) v[x] = ara[i];
    max_lcs = max(max_lcs,(int)v.size());
}
cout << "The size of the lcs is : " << max_lcs << endl;
```

Max Flow(Edmonds Carp)

```
//Edmonds Carp Algorithm
//Finds Max Flow using ford fulkerson method
//Finds path from source to sink using bfs
//Complexity V*E*E

int cap[MAX][MAX];
int par[MAX]; //keeps track of the parent in a path from s to d
int mCap[MAX]; //mCap[i] keeps track edge that have minimum cost on the shortest path from s to i

bool getPath(int s,int d,int n)
{
    SET(par);
    for(int i=1;i<=n;i++) mCap[i] = INF;
    queue <int> q;
    q.push(s);
    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(int i=1;i<=n;i++){
            if(cap[u][i]!=0 && par[i]==-1){
                par[i] = u;
                mCap[i] = min(mCap[u],cap[u][i]);
                if(i==d) return true;
                q.push(i);
            }
        }
    }
    return false;
}

int getFlow(int s,int d,int n)
{
    int F = 0;
    while(getPath(s,d,n)){
        int f = mCap[d];
        F += f;
        int u = d;
        while(u!=s){
            int v = par[u];
            cap[u][v] += f;
            cap[v][u] -= f;
            u = v;
        }
    }
    return F;
}

int main()
{
    int maxFlow = getFlow(s,d,n);
    return 0;
}
```

Max Flow(Dinic)

```
//Complexity V*V*E

#define MAX_NODES    5000

int src,snk;
int dist[MAX_NODES],work[MAX_NODES];

struct Edge{
    int to , rev_pos , c , f;
};

vector <Edge> G[MAX_NODES];

//This function is written for undirected graph
//The graph doesn't have multiple edge between two nodes
void addEdge(int u,int v,int c){
    Edge a = {v,(int)G[v].size(),c,0};
    Edge b = {u,(int)G[u].size(),c,0};
    G[u].pb(a);
    G[v].pb(b);
}

bool dinic_bfs(){
    SET(dist);
    dist[src] = 0;
    queue <int> q;
    q.push(src);
    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(int i=0;i<G[u].size();i++){
            Edge &e = G[u][i];
            int v = e.to;
            if(dist[v]==-1 && e.f<e.c){
                dist[v] = dist[u]+1;
                q.push(v);
            }
        }
    }
    return (dist[snk]>=0);
}

int dinic_dfs(int u, int fl){
    if (u == snk) return fl;
    for (; work[u] < G[u].size(); work[u]++){
        Edge &e = G[u][work[u]];
        if (e.c <= e.f) continue;
        int v = e.to;
        if (dist[v] == dist[u] + 1){
            int df = dinic_dfs(v, min(fl, e.c - e.f));
            if (df > 0){
                e.f += df;
                G[v][e.rev_pos].f -= df;
                return df;
            }
        }
    }
    return 0;
```

```
}


int maxFlow(int _src, int _snk){
    src = _src;
    snk = _snk;
    int result = 0;
    while (dinic_bfs()){
        CLR(work);
        while (int delta = dinic_dfs(src, INF)) result += delta;
    }
    return result;
}

int main()
{
    cout << maxFlow(src,snk) << endl;
    return 0;
}
```

Number Theory

Sieve

```
bool isComp[MAX+5];
vector <int> primes;

void Sieve(){
    int  i,j;
    for(i=4;i<=MAX;i+=2) isComp[i] = true;
    for(i=3;i<=sqrt(MAX);i+=2){
        if(!isComp[i]){
            for(j=i*i;j<=MAX;j+=i+i) isComp[j] = 1;
        }
    }
    for(i=2;i<=MAX;i++) if(!isComp[i]) primes.pb(i);
}
```

Euler Phi

```
int phi[MAX+10];

void calcPhi() {
    int i,p,k;
    for (i = 1; i <= MAX; i++) phi[i] = i;
    for (p = 2; p <=MAX; p++) {
        if (phi[p] == p) {
            for (k = p; k <= MAX; k += p) {
                phi[k] /= p;
                phi[k] *= (p-1);
            }
        }
    }
}
```

Extended Euclid

```
// ax+by = gcd(a,b)
// returns (x,y)

PLL extEuclid(ll a,ll b)
{
    ll s = 1,t = 0,st = 0,tt = 1;
    while(b){
        s = s - (a/b)*st;
        swap(s,st);
        t = t - (a/b)*tt;
        swap(t,tt);
        a = a%b;
        swap(a,b);
    }
    return mp(s,t);
}
```