CODE LIBRARY

MD. SHAHADAT HOSSAIN SHAHIN

UNIVERSITY OF DHAKA

Table of Contents

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

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

Template

```cpp
#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 %lld",&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 un(x)           x.erase(unique(all(x)), x.end())

#define D(x)            cerr << #x " = " << (x) << '\n'
#define DBG             cerr << "In" << '\n'

#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             300010
#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


//int dx[] = {+0, +0, +1, -1, -1, +1, -1, +1};
//int dy[] = {-1, +1, +0, +0, +1, +1, -1, -1}; //Four & Eight Direction
```

Disjoint Set Union

```
//Time complexity = 5*number of operations
struct DisjointSet{
    int *root,*rnk,n;
    DisjointSet(){}
    DisjointSet(int sz){
        root = new int[sz+1];
        rnk = new int[sz+1];
        n = sz;
    }
    ~DisjointSet(){
        delete[] root;
        delete[] rnk;
    }
    void init(){
        for(int i=1;i<=n;i++){
            root[i] = i;
            rnk[i] = 0;
        }
    }
    int findRoot(int u){
        if(u!=root[u]) root[u] = findRoot(root[u]);
        return root[u];
    }
    void Merge(int u,int v){
        int ru = findRoot(u); int rv = findRoot(v);
        if(rnk[ru]>rnk[rv]) root[rv] = ru;
        else root[ru] = rv;
        if(rnk[ru]==rnk[rv]) rnk[rv]++;
    }
};

int main(){
    DisjointSet *S;
    S = new DisjointSet(n);
    S->init();
    int ru = S->findRoot(u);
    S->Merge(u,v);
    delete S;

    //or

    DisjointSet S(n);
    S.init();
    int ru = S.findRoot(u);
    S.Merge(u,v);
    return 0;
}
```

Minimum Spanning Tree (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 = {s,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 = {v,dis[v]};
                    q.push(b);
                }
            }
        }
    }
}
```

Bellman Ford

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

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

Floyd Warshal

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

```
                else P[i][j] = -1;
            }
        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]; P[i][j] = k;
                    }
                }
            }
        }
}
void printPath(int s,int d)
{
    if(P[s][d]==-1) puts("No Path!");
    else if(P[s][d]==s) printf("%d\n",s);
    else{
        printPath(s,P[s][d]);
        printPath(P[s][d],d);
    }
}

/* Print d when the function returns */
```

Strongly Connected Component

```
/*
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;

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 n){
    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);
        }
    }
}
```

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(){
    Time = 0;
    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(){
    Time = 0;
    for(int i=1;i<=n;i++) if(!vis[i]) findBridge(i,-1);
}
```

Bridge Tree

```
struct edges{
    int u,v;
}ara[EDGES];

vector <int> ed[NODES]; // actual graph
vector <int> isBridge[NODES]; // if the edge is a bridge, the entry will be 1
vector <int> brTree[NODES]; // edges of the bridge tree

bool vis[MAX];
int st[MAX] , low[MAX] , Time = 0;
int n , m;
int cnum;
int comp[MAX];

void findBridge(int s,int par)
{
    int i,x,child = 0,j;
    vis[s] = 1;
    Time++;
    st[s] = low[s] = Time;
    for(i=0;i<ed[s].size();i++){
        x = ed[s][i];
        if(!vis[x]){
```

```
                child++;
                findBridge(x,s);
                low[s] = min(low[s],low[x]);
                if(low[x]>st[s]){
                    isBridge[s][i] = 1;
                    j = lower_bound(ed[x].begin(),ed[x].end(),s)-ed[x].begin();
                    isBridge[x][j] = 1;
                }
            }
            else{
                if(par!=x) low[s] = min(low[s],st[x]);
            }
        }
}

void processBridge(){
    CLR(vis);
    Time = 0;
    for(int i=1;i<=n;i++) if(!vis[i]) findBridge(i,-1);
}

void dfs(int s){
    int i,x;
    vis[s] = 1;
    comp[s] = cnum;
    for(i=0;i<ed[s].size();i++){
        if(!isBridge[s][i]){
            x = ed[s][i];
            if(!vis[x]) dfs(x);
        }
    }
}

int main(){
    int i,u,v,ans,k;
    scanf("%d %d",&n,&m);
    for(i=1;i<=m;i++){
        sii(u,v);

        ed[u].pb(v);
        ed[v].pb(u);

        isBridge[u].pb(0);
        isBridge[v].pb(0);

        ara[i].u = u;
        ara[i].v = v;
    }
    for(i=1;i<=n;i++) sort(all(ed[i]));
    processBridge();

    cnum = 0;
    CLR(vis);
    for(i=1;i<=n;i++){
        if(!vis[i]){
            cnum++;
            dfs(i);
        }
    }

    n = cnum; //number of nodes in the bridge tree
```

```
    for(i=1;i<=m;i++){
        if(comp[ara[i].u]!=comp[ara[i].v]){
            brTree[comp[ara[i].u]].pb(comp[ara[i].v]);
            brTree[comp[ara[i].v]].pb(comp[ara[i].u]);
        }
    }
    return 0;
}
```

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 popBCC(int s,int x){
    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++;
}

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]) popBCC(s,x);
            else if(par==-1) if(child>1) popBCC(s,x);
        }
        else if(par!=x) low[s] = min(low[s],st[x]);
    }
    if(par==-1 && child>1) isArt[s] = 1;
}

void processBCC(){
    // might need to clear S in every case
    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++;
        }
    }
}
```

```
/*
    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(){
    // might need to clear S in every case
    bcc_counter = 0;
    for(int i=0; i<n; i++){
        if(!vis[i]){
            Time = 0;
            findBCC(i,-1);
        }
    }
}
```

Euler Trail/Circuit

Directed Graph

```
/*
    This code finds Euler path/circuit in a DIRECTED graph
    1. Choose any vertex v and push it onto a stack. Initially all edges are unmarked.
    2. While the stack is nonempty, look at the top vertex, u, on the stack. If u has an unmarked incident edge,
    say, to a vertex w, then push w onto the stack and mark the edge uw. On the other hand, if u has no unmarked
    incident edge, then pop u off the stack and print it.
    When the stack is empty, you will have printed a sequence of vertices that correspond to an Eulerian circuit/Path
*/

vector <int> ed[MAX];
int in[MAX],out[MAX],work[MAX];

stack <int> tp,fp;
// return 0 if no euler path or circuit exists
// return 1 if a euler trail exists
// return 2 if a euler circuit exists
int findEulerTrail(int n){
    int start=-1,cnt = 0,ret;
    for(int i=0;i<n;i++){
        if(in[i]==out[i]);
        else if(in[i]+1==out[i]){start = i;cnt++;}
        else if(in[i]==out[i]+1) cnt++;
        else return 0;
    }
    if(start==-1){
```

```
        ret = 2;
        for(int i=0;i<n;i++) if(out[i]) start = i;
    }

    if(cnt>2) return 0;

    while(!tp.empty()) tp.pop();
    while(!fp.empty()) fp.pop();

    tp.push(start);
    int u,v,i,j;
    while(!tp.empty()){
        u = tp.top();
        if(work[u]==ed[u].size()){
            fp.push(u);
            tp.pop();
        }
        else{
            v = ed[u][work[u]];
            tp.push(v);
            work[u]++;
        }
    }
    // Have to check if all the edges are visited
    // Counting num of edges in the trail is enough
    return ret;
}

int main(){
    // need to clear ed, in , out , work per case
    int n,m,u,v;
    for(i=0;i<m;i++){
        sii(u,v);
        ed[u].pb(v);
        in[v]++;
        out[u]++;
    }
    int res = findEulerTrail(n);
    //printing trail/circuit
    while(fp.empty()){
        printf("%d\n",S.top());
        S.pop();
    }
    return 0;
}
```

Undirected Graph

```
/*
    This code finds Euler path/circuit in a directed graph
    1. Choose any vertex v and push it onto a stack. Initially all edges are unmarked.
    2. While the stack is nonempty, look at the top vertex, u, on the stack. If u has an unmarked incident edge,
    say, to a vertex w, then push w onto the stack and mark the edge uw. On the other hand, if u has no unmarked
    incident edge, then pop u off the stack and print it.
    When the stack is empty, you will have printed a sequence of vertices that correspond to an Eulerian circuit/Path
*/

```

```
vector <int> ed[MAX];
int deg[MAX],work[MAX];
stack <int> tp,fp;
// return 0 if no euler path or circuit exists
// return 1 if a euler trail exists
// return 2 if a euler circuit exists
int findEulerTrail(int n){
    int start=-1,cnt = 0,ret;
    for(int i=0;i<n;i++) if(deg[i]%2){ start = i; cnt++;}
    if(start==-1){
        ret = 2;
        for(int i=0;i<n;i++) if(deg[i]) start = i;
    }
    if(cnt>2) return 0;

    while(!tp.empty()) tp.pop();
    while(!fp.empty()) fp.pop();

    tp.push(start);
    int u,v,i,j;
    while(!tp.empty()){
        u = tp.top();
        if(work[u]==ed[u].size()){
            fp.push(u);
            tp.pop();
        }
        else{
            // next two lines are not needed for directed graph
            while(work[u] <ed[u].size() && ed[u][work[u]]==-1) work[u]++;
            if(work[u]==ed[u].size()) continue;
            v = ed[u][work[u]];
            // next two lines are not needed for directed graph
            j = lower_bound(ed[v].begin(),ed[v].end(),u)-ed[v].begin();
            ed[v][j] = -1;
            tp.push(v);
            work[u]++;
        }
    }
    // Have to check if all the edges are visited
    // Counting num of edges in the trail is enough
    return ret;
}

int main(){
    // need to clear ed, in , out , work per case
    int n,m,u,v;
    for(i=0;i<m;i++){
        sii(u,v);
        ed[u].pb(v);
        ed[v].pb(u);
        deg[u]++;deg[v]++;
    }
    for(i=0;i<n;i++) sort(all(ed[i]));
    int res = findEulerTrail(n);
    //printing trail/circuit
    while(fp.empty()){
        printf("%d\n",S.top());
        S.pop();
    }
    return 0;
}
```

Centroid Decomposition

```
/*
    Problem : Two kinds of operations in a tree(Initially node 1 is white, others are black)
    Update v : Change the color of node v to white
    Query v : Distance of loset white node from node v(can be node v)
*/

/* Using dis[i][j] array to calculate distance between two node in original tree */

vector <int> ed[MAX]; // edjacency list of the input tree
bool isCentroid[MAX]; // if the node is already a centroid of some part
int sub[MAX]; // subtree size of a node
int cpar[MAX]; // parent of a node in the centroid tree
int clevel[MAX]; // level of a node in centroid tree
int dis[20][MAX]; // dis[i][j] = distance of node j from the root of the i'th level of decomposition

void calcSubTree(int s,int par){
    int i,x;
    sub[s] = 1;
    for(i=0;i<ed[s].size();i++){
        x = ed[s][i];
        if(x!=par && !isCentroid[x]){
            calcSubTree(x,s);
            sub[s] += sub[x];
        }
    }
}

int nn;// number of nodes in the part

int getCentroid(int s,int par){
    int i,x;
    for(i=0;i<ed[s].size();i++){
        x = ed[s][i];
        if(!isCentroid[x] && x!=par && sub[x]>(nn/2)) return getCentroid(x,s);
    }
    return s;
}

void setDis(int s, int from, int par, int l) {
    dis[from][s] = l;
    int i,x;
    for(i=0;i<ed[s].size();i++) {
        x = ed[s][i];
        if(x!=par && !isCentroid[x]) {
            setDis(x, from, s, l+1);
        }
    }
}

//complexity → O(nlog(n))
void decompose(int s,int p,int l){
    calcSubTree(s,p);
    nn = sub[s];
    int c = getCentroid(s,p);
```

```

    // setDis(c,..) calculates the distance of every node from c which
    // are in the subtree of c in centroid tree
    setDis(c,l,p,0);

    isCentroid[c] = true;
    cpar[c] = p;
    clevel[c] = l;

    int i,x;
    for(i=0;i<ed[c].size();i++){
        x = ed[c][i];
        if(!isCentroid[x]) decompose(x,c,l+1);
    }
}

int ans[MAX];

inline void update(int v) {
    int u = v;
    while(u!=-1) {
        ans[u] = min(ans[u], dis[clevel[u]][v]);
        u = cpar[u];
    }
}

inline int query(int v) {
    int ret = INF;
    int u = v;
    while(u != -1) {
        ret = min(ret, dis[clevel[u]][v]+ans[u]);
        u = cpar[u];
    }
    return ret;
}


int main(){
    decompose(1,-1,0);
    for(i=1;i<=n;i++) ans[i] = INF;
    update(v);
    query(v));
    return 0;
}
```

```
/*Using Spares Table and LCA to find distance between two nodes in the original tree*/

vector <int> ed[MAX]; // edjacency list of the input tree
bool isCentroid[MAX]; // if the node is already a centroid of some part
int sub[MAX]; // subtree size of a node
int cpar[MAX]; // parent of a node in the centroid tree
int L[MAX]; // Depth of a node in the original tree
int P[MAX][20]; // P[i][j] denotes (2^j)th parent of node i in the original tree

/******************LCA*******************/

int lg;
```

```

void dfs(int s,int par,int l){
    int i,x;
    L[s] = l;
    for(i=0; i<ed[s].size(); i++){
        x = ed[s][i];
        if(x!=par){
            P[x][0] = s;
            dfs(x,s,l+1);
        }
    }
}

void lca_build(int n){
    dfs(1,-1,0);
    lg = (log(n)/log(2.0))+2;
    int i,j;
    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 P[x][0];
}

// returns distance between node u and node v in the original tree
int dist(int u,int v){
    return L[u] + L[v] - 2*L[lca_query(u,v)];
}

/**************Decomposition**************/

void calcSubTree(int s,int par){
    int i,x;
    sub[s] = 1;
    for(i=0;i<ed[s].size();i++){
        x = ed[s][i];
        if(x!=par && !isCentroid[x]){
            calcSubTree(x,s);
            sub[s] += sub[x];
        }
    }
}

int nn;// number of nodes in the part

int getCentroid(int s,int par){
    int i,x;
```

```
    for(i=0;i<ed[s].size();i++){
        x = ed[s][i];
        if(!isCentroid[x] && x!=par && sub[x]>(nn/2)) return getCentroid(x,s);
    }
    return s;
}

//complexity --> O(n*log(n))
void decompose(int s,int p){
    calcSubTree(s,p);
    nn = sub[s];
    int c = getCentroid(s,p);
    isCentroid[c] = true;
    cpar[c] = p;
    int i,x;
    for(i=0;i<ed[c].size();i++){
        x = ed[c][i];
        if(!isCentroid[x]) decompose(x,c);
    }
}
```

```
/*****************Operation*****************/
int ans[MAX];

void update(int v){
    int u = v;
    while(u!=-1){
        ans[u] = min(ans[u],dist(u,v));
        u = cpar[u];
    }
}

int query(int v){
    int u = v,ret = INF;
    while(u!=-1){
        ret = min(ret,ans[u]+dist(u,v));
        u = cpar[u];
    }
    return ret;
}

int main(){
    decompose(1,-1);
    lca_build(n);
    for(i=1;i<=n;i++) ans[i] = INF;
    update(v);
    query(v);
    return 0;
}
```

Maximum Flow

Edmonds Karp

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

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

Dinic

```
// Complexity V*V*E
#define MAXN    5010
int src,snk;
int dist[MAXN],work[MAXN];

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

vector <Edge> ed[MAXN];

int M[MAXN][MAXN]; //Don't forget to SET the array

/* if there are multiple edges between same pair of nodes
   v might appear in the edge list of u multiple times */
void addEdge(int u,int v,int c)  {
    Edge a = {v,ed[v].size(),c,0};
    Edge b = {u,ed[u].size(),0,0}; // cap 0 should be replaced by c for directed graphs
    ed[u].pb(a);
    ed[v].pb(b);
}

/* Makes sure u will appear in the node list of u only once
   but takes extra log(edges) time */
void addEdge(int u,int v,int c)  {
    if(M.find(mp(u,v))==M.end()){
        M[mp(u,v)] = ed[u].size();
        M[mp(v,u)] = ed[v].size();
        Edge a = {v,ed[v].size(),c,0};
        Edge b = {u,ed[u].size(),c,0}; //c should be replaced by 0 for undirected graphs
        ed[u].pb(a);
        ed[v].pb(b);
    }
    else{
        int x = M[mp(u,v)];
        int y = M[mp(v,u)];
        ed[u][x].c += c;
        ed[v][y].c += c; // If the graph is directed, comment it out
    }
}

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<ed[u].size(); i++){
            Edge &e = ed[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] < ed[u].size(); work[u]++){
        Edge &e = ed[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;
                ed[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;
}
```

Minimum Cost Maximum Flow

```
// Maximizes the flow first, then minimizes the cost

// Complexity -->

#define MAXN 105 // maximum number of nodes

int  N;
int dis[MAXN] , par[MAX] , mCap[MAX] , pos[MAX];
bool vis[MAXN];

struct Edge{
   int to, rev_pos, cap, cost, flow;
};

vector <Edge> ed[MAX];

void addEdge(int u,int v,int cap,int cost){
    Edge a = {v,ed[v].size(),cap,cost,0};
    Edge b = {u,ed[u].size(),0,-cost,0};
    ed[u].pb(a);
    ed[v].pb(b);
}
```

```
// 1 based indexing
/*
    SPFA Technique :
    Insert a node in the queue if we get a better distance for
    the node and the node is not already in the queue
    A combination of Dijkstra and BFS
*/

bool SPFA(){
    int i,u,v;
    CLR(vis);
    for(i=0;i<=N;i++) mCap[i] = dis[i] = INF;
    queue <int> q;
    dis[src] = 0;
    vis[src] = true; // src is in the queue now
    q.push(src);

    while(!q.empty()){
        u = q.front();
        q.pop();
        vis[u] = false; // u is not in the queue now
        for(i=0;i<ed[u].size();i++){
            Edge e = ed[u][i];
            v = e.to;
            if(e.cap>e.flow && dis[v]>dis[u]+e.cost){
                dis[v] = dis[u]+e.cost;
                par[v] = u;
                pos[v] = i;
                mCap[v] = min(mCap[u],e.cap-e.flow);
                if(!vis[v]){
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
    }
    return (dis[snk]!=INF);
}

pair <int,int> MCMF() {
    int F = 0, C = 0 , f;
    while(SPFA()){
        int u = snk;
        f = mCap[u];
        F += f;
        while(u!=src){
            int v = par[u];
            ed[v][pos[u]].flow += f; // edge of v-->u increases
            ed[u][ed[v][pos[u]].rev_pos].flow -= f;
            u = v;
        }
        C += dis[snk] * f;
    }
    return mp(F,C);
}

int main(){
    //dont forget to initialize src and snk
    MCMF();
    return 0;
}
```

Maximum Bipartite Matching

```
// worst case complexity V*E

#define L   105
#define R   105

vector <int> G[L];      //The adjacency list for the nodes in the left side
int matchL[L],matchR[R];
bool vis[L];

bool dfs(int s)
{
    int i,x;
    vis[s] = true;
    for(i = 0;i<G[s].size();i++){
        x = G[s][i];
        if(matchR[x]==-1 || (!vis[matchR[x]] && dfs(matchR[x]))){
            matchL[s] = x;
            matchR[x] = s;
            return 1;
        }
    }
    return 0;
}

// n = number of nodes in the left side
int match(int n) {
    int i;
    bool done;
    SET(matchL); SET(matchR);
    while(true){
        CLR(vis);
        done = true;

        for(i=1;i<=n;i++)
            if(matchL[i]==-1 && !vis[i] && dfs(i)) done = false;

        if(done) break;
    }

    int cnt = 0; // number of matches
    for(i=1; i<=n; i++)
        if(matchL[i]!=-1) cnt++;

    return cnt;
}
```

A minimum vertex cover of a graph is a minimum set of nodes such that each edge of the graph has at least one endpoint in the set.

Minimum Vertex Cover = Maximum Matching

Maximum independent set of a graph is a maximum set of nodes such that no two nodes of the set are connected by an edge.

Maximum Independent Set = Total Nodes – Minimum Vertex Cover

Finding Minimum Vertex Cover :

Run BPM. After that do a dfs from every **unmatched** node in the left side. Go through the edge u to v if and only if :

- u is on the left side and matchL[u] != v or
- u is on the right side and matchR[u] == v

After this, all visited nodes in the left side and the unvisited nodes in the right side will form a maximum independent set. But why? (See the proof)
The rest of the nodes will form MVC then.

Proof :
Size of Maximum independent set = Total nodes - Maximum Matching
In our algorithm, It's easy to observe that the set will be an independent set. And it's size will be equal to **Total nodes - Maximum Matching**. Because, we are keeping exactly one node of every matched edge in the set. EASY!! :P

Dilworth's theorem :
An anti chain is a set of nodes of a graph such that there is no path from any node to another node using the edges of the graph. Dilworth's theorem states that in a directed acyclic graph, the size of a minimum **general** path cover equals the size of a maximum anti chain.

Segment Tree

Point Update, 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));
}
```

Range Update, 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));
}
```

How Many Non Zero Elements in the Tree

```
/*
    Given an array consisting of all zeroes
    Update -> Add some value to all the elements of a range
                (no element ever gets negative value)
    Query -> How many non zero element in a range
*/
```

```
int tree[MAX];  /// how many non zero elements in this segment
int lazy[MAX];  /// how many times a node is fully updated

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

void build(int n,int st,int ed){
    lazy[n] = tree[n] = 0;
    if(st==ed) return;
    int mid = (st+ed)/2;
    build(2*n,st,mid);
    build(2*n+1,mid+1,ed);
}

void update(int n,int st,int ed,int i,int j,int v){
    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);
    lazyUpdate(n,st,ed);
}

int 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 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{
    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;
}
```

Binary Indexed Tree

```
/*
    Initially the tree array is set to zero
    Point Update(Adding v to index p)
    Query returns sum of the range [1,p]
*/

int tree[10];

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

//returns sum of range[1,p]
int query(int p){
    int sum = 0;
    while(p > 0) sum += tree[p], p -= p & (-p);
    return sum;
}

//returns sum of range[l,r]
int range_query(int l,int r){
    if(l>r) return 0;
    return query(r)-query(l-1);
}
```

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,bn; //bn --> block number
} 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);
        while(L<l) Remove(L++);
        while(R>r) Remove(R--);
        while(R<r) Add(++R);

        res[quer[i].id] = ans;
    }
}

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

Lowest Common Ancestor

```
int lg;

int L[MAX]; // Depth of a node
int P[MAX][20]; // P[i][j] denotes (2^j)th parent of node i
vector <int> ed[MAX];

void dfs(int s,int par,int l){
    int i,x;
    L[s] = l;
    for(i=0; i<ed[s].size(); i++){
        x = ed[s][i];
        if(x!=par){
            P[x][0] = s;
            dfs(x,s,l+1);
        }
    }
}

void lca_build(int n)  {
    lg = (log(n)/log(2.0))+2;
    int i,j;
    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 P[x][0];
}

int main(){
    SET(P);
    dfs(1,-1,0); // nodes start from 1
    lca_build(100); // total nodes = n
}
```

Trie

Array Implementation

```
#define MAX     200000
#define SZ      26

int root,now;
int nxt[MAX][SZ] , cnt[MAX][SZ];

void Clear(){
    root = now = 1;
    CLR(nxt),CLR(cnt);
}

int scale(char ch) { return (ch - 'a');}

void Insertu(string s){
    int cur = root, to;
    for(int i=0;i<s.size();i++){
        to = scale(s[i]) ;
        if( nxt[cur][to]==0) nxt[cur][to] = ++now;
        cnt[cur][to]++;
        cur = nxt[cur][to];
    }
}

bool Find(string s){
    int cur = root, to;
    for(int i=0;i<s.size();i++){
        to = scale(s[i]) ;
        if( !nxt[cur][to] || !cnt[cur][to]) return false;
```

```
        cur = nxt[cur][to];
    }
    return true;
}

/// If a string doesn't exist in the trie, Delete function for that string won't work
void Delete(string s){
    int cur = root, to;
    for(int i=0;i<s.size();i++){
        to = scale(s[i]) ;
        cnt[cur][to]--;
        cur = nxt[cur][to];
    }
}
```

Heavy Light Decomposition

```
/*Code of Lightoj-1348 : Aladdin and the Return Journey*/

int n;
int chainNo , chainHead[MAX] , chainId[MAX];
int it , baseArray[MAX] , posInBaseArray[MAX] ;
int subtreeSz[MAX];
int ara[MAX];
vector <int> ed[MAX];

int lg;

int L[MAX]; // Depth/level of a node
int P[MAX][20]; // P[i][j] denotes (2^j)th parent of node i

void dfs(int s,int par){
    int i,x;
    subtreeSz[s] = 1;
    for(i=0;i<ed[s].size();i++){
        x = ed[s][i];
        if(x!=par){
            P[x][0] = s;
            L[x] = L[s]+1;
            dfs(x,s);
            subtreeSz[s] += subtreeSz[x];
        }
    }
}

/*Exact code of segment tree here(lazy propagation added if needed)*/

/*HLD starts*/
void HLD(int s,int p){
    it++;
    posInBaseArray[s] = it;
    baseArray[it] = ara[s];
    if(chainHead[chainNo]==-1) chainHead[chainNo] = s;
    chainId[s] = chainNo;

    int heavyChild = -1, heavyChildSz = 0;
    int i,x;
    for(i=0;i<ed[s].size();i++){
```

```
        x = ed[s][i];
        if(x!=p && subtreeSz[x]>heavyChildSz){
            heavyChildSz = subtreeSz[x];
            heavyChild = x;
        }
    }
    if(heavyChild!=-1) HLD(heavyChild,s);
    for(i=0;i<ed[s].size();i++){
        x = ed[s][i];
        if(x!=p && x!=heavyChild){
            chainNo++; HLD(x,s);
        }
    }
}

void HLDConstruct(){
    it = 0;
    chainNo = 1;
    SET(chainHead);

    // for calculating sub tree size
    // also saves the level of every node for LCA function
    SET(P);
    L[1] = 0;
    dfs(1,-1);
    HLD(1,-1);
    build(1,1,n);
}
/*HLD ends*/
```

```
/*LCA starts*/

void lca_build(){
    lg = (log(n)/log(2))+1;
    int i,j;
    //P is set to -1 in HLDConstruct
    //P[i][0] is updated in dfs
    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 P[x][0];
}
/*LCA ends*/
```

```

// path from u to v, L[u] >= L[v]
int call(int u,int v)
{
    int ret = 0,l,r,head;
    while(true){
        l = posInBaseArray[v];
        if(chainId[u]!=chainId[v]){
            head = chainHead[chainId[u]];
            l = posInBaseArray[head];
        }
        r = posInBaseArray[u];
        if(l<=r) ret += query(1,1,n,l,r).sum;
        if(chainId[u]==chainId[v]) return ret;
        u = P[head][0];
    }
}

int getResult(int u,int v){
    int lca = lca_query(u,v);
    int ret = 0;
    return ret = call(u,lca) + call(v,lca) - baseArray[posInBaseArray[lca]];
}

void updateNode(int id, int v){
    id = posInBaseArray[id];
    update(1,1,n,id,v);
}

int main()    {
    HLDConstruct();
    lca_build();
    return 0;
}
```

Treap

Normal Treap

```
struct node{
    int prior; // Heap value
    int val; // BST value
    int sz; // Subtree Size
    int sum; // This bst maintains the sum of it's child nodes
    struct node *l,*r,*p;
};

typedef node* pnode;

pnode Treap;

int get_sz(pnode t) { return t?t->sz:0; }
int get_sum(pnode t) { return t?t->sum:0; }

void update(pnode t){
    if(!t) return;
    if(t->l) t->l->p = t;
```

```
    if(t->r) t->r->p = t;
    t->sz = get_sz(t->l) + 1 + get_sz(t->r);
    t->sum = get_sum(t->l) + 1 + get_sum(t->r);
}

pnode init(int val){
    pnode ret = (pnode)malloc(sizeof(node));
    ret->val = val;
    ret->sz = 1;
    ret->prior=rand();
    ret->p = ret->l = ret->r = NULL;
    return ret;
}

// l contains the nodes having BST value less than val, r contains the rest
void split(pnode t,pnode &l,pnode &r,int val){
    if(!t) l = r = NULL;
    else if(t->val<=val) split(t->r,t->r,r,val) , l = t ;
    else split(t->l,l,t->l,val) , r = t ;
    update(t);
}

void Merge(pnode &t,pnode l,pnode r){
    if(!l || !r) t = l ? l : r;
    else if(l->prior > r->prior) Merge(l->r,l->r,r), t = l ;
    else Merge(r->l,l,r->l), t = r ;
    update(t);
}

// Inserting a new node into BST
void Insert(pnode &t,pnode it){
    if(!t) t = it ;
    else if(it->prior>t->prior) split(t,it->l,it->r,it->val) , t = it ;
    else if(t->val<=it->val) Insert(t->r,it);
    else Insert(t->l,it);
    update(t);
}
//or
void Insert(pnode &t,pnode it){
    if(!t) t = it;
    pnode l,r;
    split(t,l,r,it->val);
    Merge(t,l,it);
    Merge(t,t,r);
}

// Removing a node having BST value = val
void Remove(pnode &t,int val){
    if(!t)return;
    else if(t->val==val){
        pnode temp=t;
        Merge(t,t->l,t->r);
        free(temp);
    }
    else if(t->val<val) Remove(t->r,val);
    else Remove(t->l,val);
    update(t);
}

void Delete(pnode &t){
    if(!t) return;
```

```
    if(t->l) Delete(t->l);
    if(t->r) Delete(t->r);
    delete(t);
    t = NULL;
}
```

Implicit Treap

```
/**Treap as Interval Tree(1 based) With Insert and Remove Operation**/

typedef struct node{
    int prior,sz;
    int val; //value stored in the array
    int sum; //whatever info you want to maintain in segtree for each node
    int lazy; //whatever lazy update you want to do
    struct node *l,*r,*p;
} node;

typedef node* pnode;
pnode Treap;
int get_sz(pnode t){ return t?t->sz:0; }
int get_sum(pnode t){ return t?t->sum:0; }

void lazyUpdate(pnode t){
    if(!t || !t->lazy)return;
    t->val += t->lazy;
    t->sum += t->lazy*get_sz(t);
    if(t->l) t->l->lazy += t->lazy;
    if(t->r) t->r->lazy += t->lazy;
    t->lazy=0;
}

//operation of segtree and size,parent update
void operation(pnode t) {
    if(!t)return;
    lazyUpdate(t->l); lazyUpdate(t->r); //imp:propagate lazy before combining t->l,t->r;
    t->sz=get_sz(t->l)+1+get_sz(t->r);
    t->sum = get_sum(t->l) + t->val + get_sum(t->r); // updateing sum
    if(t->l) t->l->p = t;
    if(t->r) t->r->p = t;
}

// The subarray[1:pos] is saved in node l, the rest in r
void split(pnode t,pnode &l,pnode &r,int pos,int add=0){
    if(!t) return void( l = r = NULL) ;
    lazyUpdate(t);
    int curr_pos = add + get_sz(t->l)+1;
    if(curr_pos<=pos) split(t->r,t->r,r,pos,curr_pos),l=t;
    else split(t->l,l,t->l,pos,add),r=t;
    operation(t);
}

void Merge(pnode &t,pnode l,pnode r){
    lazyUpdate(l); lazyUpdate(r);
    if(!l || !r) t = l?l:r;
    else if(l->prior>r->prior) Merge(l->r,l->r,r) , t = l ;
    else    Merge(r->l,l,r->l) , t = r ;
    operation(t);
}
```

```

pnode init(int val){
    pnode ret = (pnode)malloc(sizeof(node));
    ret->prior = rand();
    ret->sz = 1;
    ret->val = ret->sum = val;
    ret->lazy = 0;
    ret->p = ret->l = ret->r = NULL;
    return ret;
}
```

```
//changes the value of the node at position id to val
void point_update(pnode &t,int id,int val){
    pnode L,mid,R;
    split(t,L,mid,id-1);
    split(mid,t,R,1);
    t->val = val;
    Merge(mid,L,t);
    Merge(t,mid,R);
}

//deletes the node at position id
void Remove(pnode &t,int id){
    pnode L,mid,R,X;
    split(t,L,mid,id-1);
    split(mid,X,R,1);
    delete X;
    Merge(t,L,R);
}

//inserts a node at position id having array value = val
void Insert(pnode &t,int id,int val){
    pnode L,R,mid;
    pnode it = init(val);
    split(t,L,R,id-1);
    Merge(mid,L,it);
    Merge(t,mid,R);
}

int range_query(pnode t,int l,int r){
    pnode L,mid,R;
    split(t,L,mid,l-1);
    split(mid,t,R,r-l+1);
    int ans = t->sum;
    Merge(mid,L,t);
    Merge(t,mid,R);
    return ans;
}

void range_update(pnode t,int l,int r,int val){
    pnode L,mid,R;
    split(t,L,mid,l-1);
    split(mid,t,R,r-l+1);
    t->lazy += val;
    Merge(mid,L,t);
    Merge(t,mid,R);
}

```

```
void Delete(pnode &t){
    if(!t) return;
    if(t->l) Delete(t->l);
    if(t->r) Delete(t->r);
    delete(t);
    t = NULL;
}

int ara[10];

int main(){
    //creating a treap to use it as an interval tree of ara (1 based)
    int n = 10;
    for(int i=1; i<=n; i++){
        if(i==1) Treap = init(ara[i]);
        else Merge(Treap,Treap,init(ara[i]));
    }
    Delete(Treap);  //Deleting when work done
    return 0;
}
```

Maximum Contiguous Sum Merging

```
// Maximum contiguous sum merging
void operation(pnode t){
    if(!t)return;
    t->sum = get_sum(t->l) + t->val + get_sum(t->r);
    t->res = max( max(get_res(t->l), get_res(t->r)), max(0, get_rsum(t->l)) + t->val + max(0, get_lsum(t->r)));
    t->lsum = max(max(0,get_lsum(t->r)) + t->val + get_sum(t->l),get_lsum(t->l));
    t->rsum = max(get_sum(t->r) + t->val + max(0,get_rsum(t->l)),get_rsum(t->r));
}
```

Matrix Exponentiation

```
struct matrix{
    ll mat[100][100];
    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 ;
    }
};
```

Extended Euclid (Solving Linear Diophantine Equation)

```
/*
    c = gcd(a,b);

    ax + by = c;
    (bq + r)x + by = c;
    bqx + rx + by = c;
    b(qx + y) + rx = c;
    bx' + ry' = c;      [r = a % b]

    We get,
    x' = qx + y;
    y' = x

    So,
    y = x' - qx;
    y = x' - qy';       [y' = x]
    and
    x = y'

    If c is not the gcd then,

    actual x = x * (c/gcd)
    actual y = y * (c/gcd)
    But if gcd doesn't divide c, there is no solution.
*/


// returns (x,y) for ax + by = gcd(a,b)
// keep in mind that if a or b or both are negative,  gcd(a,b) will be negative

PLL extEuclid(ll a,ll b)
{
    if(b==0LL) return mp(1LL,0LL);
    PLL ret,got;
    got = extEuclid(b,a%b);
    ret = mp(got.yy,got.xx-(a/b)*got.yy);
    return ret;
}

/*
    From one solution (x0,y0), we can obtain all the solutions of the given equation.
    Let g = gcd(a,b) and let x0,y0 be integers which satisfy the following:
    a*x0+b*y0 = c
    Now, we should see that adding b/g to x0 and at the same time subtracting a/g
    from y0 will not break the equality:

      a*(x0 + b/g) + b*(y0 - a/g)
    = a*x0 + b*y0 + (a*b)/g - (b*a)/g
    = c
    Obviously, this process can be repeated again, so all the numbers of the form:

    x = x0 + k * (b/g)
    y = y0 - k * (a/g)
```

```
    are solutions of the given Diophantine equation.

    Solution with minimum (x+y):
    x + y = x0 + y0 + k*(b/g - a/g)
    x + y = x0 + y0 + k*((b-a)/g)

    If b>a, we need to find the k with the minimum value
    else we need to find the k with the maximum value
*/

// Iterative Implementation
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);
}
```

```
// returns number of solutions for the equation ax + by = c
// where minx <= x <= maxx and miny <= y <= maxy
ll numberOfSolutions(ll a,ll b,ll c,ll minx,ll maxx,ll miny,ll maxy)
{
    if(a==0 && b==0){
        if(c!=0) return 0;
        else return (long long)(maxx-minx+1)*(maxy-miny+1);
    }

    ll gcd = __gcd(a,b);
    if(c%gcd!=0) return 0;

    if(b==0){
        c /= a;
        if(c>=minx && c<=maxx) return maxy-miny+1;
        else return 0;
    }

    if(a==0){
        c /= b;
        if(c>=miny && c<=maxy) return maxx-minx+1;
        else return 0;
    }

    PLL sol = extEuclid(a,b);

    a /= gcd;
    b /= gcd;
    c /= gcd;

    ll x,y;
    x = sol.xx*c;
    y = sol.yy*c;

```

```
    ll lx,ly,rx,ry;

    if(x<minx) lx = getCeil(minx-x,abs(b));
    else lx = -getFloor(x-minx,abs(b));

    if(x<maxx) rx = getFloor(maxx-x,abs(b));
    else rx = -getCeil(x-maxx,abs(b));

    if(b<0){
        lx *= -1;
        rx *= -1;
        swap(lx,rx);
    }
    if(lx>rx) return 0;


    if(y<miny) ly = getCeil(miny-y,abs(a));
    else ly = -getFloor(y-miny,abs(a));

    if(y<maxy) ry = getFloor(maxy-y,abs(a));
    else ry = -getCeil(y-maxy,abs(a));

    if(a<0){
        ly *= -1;
        ry *= -1;
        swap(ly,ry);
    }
    if(ly>ry) return 0;

    ly *= -1;
    ry *= -1;
    swap(ly,ry);

    lx = max(lx,ly);
    rx = min(rx,ry);

    return max(rx-lx+1,0LL);
}

// works for negatibe numbers as well
ll getFloor(ll a,ll b)
{
    double x = a/(double)b;
    ll f = floor(x);
    while(f>x) f--;
    while(f+1<x) f++;
    return f;
}

ll getCeil(ll a,ll b)
{
    double x = a/(double)b;
    ll c = ceil(x);
    while(c<x) c++;
    while(c-1>x) c--;
    return c;
}
```

Fraction Class

```
//a --> numerator
//b --> denominator

/*
    Every fraction must be reduced before passing
    Fraction f2(-3,7); // declaration process
*/

struct Fraction{
    ll a,b;
    Fraction(){}
    Fraction(ll _a,ll _b){
        a = _a;
        b = _b;
    }
};

ll GCD(ll a,ll b){
    if(b==0) return a;
    else return GCD(b,a%b);
}

ll LCM(ll a,ll b){
    ll ret;
    ll gcd = GCD(a,b);
    ret = a/gcd;
    ret *= b;
    return ret;
}

Fraction Reduce(Fraction f){
    ll gcd = GCD(f.a,f.b);
    f.a /= gcd;
    f.b /= gcd;
    return f;
}

Fraction Add(Fraction x,Fraction y){
    Fraction sum;

    sum.b = LCM(x.b,y.b);
    sum.a = 0;
    sum.a += (sum.b/x.b)*x.a;
    sum.a += (sum.b/y.b)*y.a;

    return Reduce(sum);
}

Fraction Sub(Fraction x,Fraction y){
    y.a *= -1;
    return Add(x,y);
}



Fraction Mul(Fraction x,Fraction y){
```

```
    Fraction prod;

    ll gcd = GCD(x.a,y.b);
    x.a /= gcd;
    y.b /= gcd;

    gcd = GCD(y.a,x.b);
    y.a /= gcd;
    x.b /= gcd;

    prod.a = x.a*y.a;
    prod.b = x.b*y.b;

    return Reduce(prod);
}

Fraction Div(Fraction x,Fraction y){
    Fraction ret;
    swap(y.a,y.b);
    return Mul(x,y);
}
```

Sieve of Eratosthenes

```
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);
}
```

Gaussian Elimination

```
#define SZ                      105
#define EPS                     1e-8

double mat[SZ][SZ]; // Augmented Matrix
int where[SZ]; // where[i] denotes the row index of the pivot element of column i
double ans[SZ];

/// n for row, m for column
int Gauss(int n,int m)
{
    SET(where);
    for(int row=0,col=0;col<m && row<n;col++){
        int max_row = row;
        for(int i=row;i<n;i++)
            if( abs(mat[i][col]) > abs(mat[max_row][col]) ) max_row = i;

        if( abs(mat[max_row][col]) < EPS ) continue;

        for(int i=col;i<=m;i++) swap(mat[row][i],mat[max_row][i]);

        where[col] = row;

        double mul;
        for(int i=row+1;i<n;i++){
            if(abs(mat[i][col])>EPS){
                mul = mat[i][col]/mat[row][col];
                for(int j=col;j<=m;j++) mat[i][j] -= mul*mat[row][j];
            }
        }
        row++;
    }

    // checking 0 row
    double sum;
    for(int i=0;i<n;i++){
        sum = 0;
        for(int j=0;j<m;j++) sum += abs(mat[i][j]);
        if( abs(sum) < EPS && abs(mat[i][m]) > EPS ) return 0; //no solution
    }

    // back substitution
    double sltn;
    int cur;
    for(int i=m-1;i>=0;i--){
        //if(where[i] == -1)  return INF; // infinitely many solutions
        sltn = mat[where[i]][m];
        cur = where[i];
        for(int j = i+1; j<m; j++)
            sltn -= mat[cur][j]*ans[j];
        ans[i] = sltn/mat[cur][i];
    }

    return 1; // unique solution
}
```

Gauss-Jordan Elimination

```
#define SZ 105

double mat[SZ][SZ]; // Augmented Matrix
int where[SZ]; // where[i] denotes the row index of the pivot element of column i
double ans[SZ];

// n for row, m for column
int GaussJordan(int n,int m)
{
    SET(where);
    for(int row=0,col=0; col<m && row<n; col++){
        int max_row = row;
        for(int i=row; i<n; i++)
            if( abs(mat[i][col]) > abs(mat[max_row][col]) ) max_row = i;

        if( abs(mat[max_row][col]) < EPS ) continue;

        for(int i=col; i<=m; i++) swap(mat[row][i],mat[max_row][i]);

        where[col] = row;

        double mul;
        for(int i=0; i<n; i++)
            if( i!=row && abs(mat[i][col])>EPS){
                mul = mat[i][col]/mat[row][col];
                for(int j=col; j<=m; j++) mat[i][j] -= mul*mat[row][j];
            }
        row++;
    }
    for(int i=0; i<m; i++)
        if(where[i]!=-1)
            ans[i] = mat[where[i]][m]/mat[where[i]][i];

    double sum;
    for(int i=0; i<n; i++){
        sum = 0;
        for(int j=0; j<m; j++) sum += ans[j] * mat[i][j];
        if( abs(sum - mat[i][m]) > EPS ) return 0;  // no solution
    }

    for(int i=0; i<m; i++)
        if (where[i]==-1) return INF;  // Infinitely many solutions

    return 1;  // unique solution
}
```

Gauss Related Problem

```
/*
    Problem :   Given a set of numbers, Find a subset such that the xor of the elements
                of the subset is as large as possible
    Idea    :   We will try to make the MSB of the result 1 first, then the next bit
```

```
                There will be an equation for every bit
                if the numbers are 1101,0010,1010 then the equation form MSB is
                1*ans[0] + 0*ans[1] + 1*ans[2] = 1
*/

#define MAX         105
ll ara[MAX];
bitset <MAX> mat[70];

int row,ans[MAX],where[MAX];

// bn = bit number
// n = number of columns
// val = the target value of the bn'th bit of the result
void add(int bn,int val,int n){
    ++row;
    mat[row][MAX-1] = val;
    //Stores the bn'th bit of every number in the matrix
    for(int col=0; col<n; col++) mat[row][col]=( (ara[col]>>bn) & 1 );
    // If this column has a pivot entry, we will xor the row with the row containg pivot entry
    for(int col=0; col<n; col++){
        if(mat[row][col]){
            if(where[col]) mat[row] ^= mat[where[col]];
            else break;
        }
    }
    // Setting the pivot
    for(int col=0; col<n; col++){
        if(mat[row][col]){
            where[col]=row;
            return;
        }
    }
    // If no pivot element in the row, the equation is not added
    --row;
}

void solve(int n,int m){
    CLR(where);
    row = 0;
    for(int i=m;i>=0;i--){
        add(i,1,n); // Trying to keep the i'th bit 1 of theresult
    }
    // Back Substitution
    for(int i=n-1;i>=0;i--){
        if(mat[where[i]][MAX-1]){
            ans[i] = 1;
            for(int j=1;j<=row;j++){
                if(mat[j][i]) mat[j].flip(MAX-1);
            }
        }
        else ans[i] = 0;
    }

}

int main(){
    int n;
    ll tot;

```

```
    scanf("%d",&n);
    tot = 0;
    for(int i=0;i<n;i++){
        scanf("%lld",&ara[i]);
        tot ^= ara[i];
    }
    solve(n,63);
    ll res = 0;
    for(int i=0; i<n; i++) res ^= (ara[i]*ans[i]);
    printf("%lld\n",res);
}
```

Chinese Remainder Theorem

```
/*
    X = a_1 % m_1
    X = a_2 % m_2
    X = a_3 % m_3

    m_1,m_2,m_3 are pair wise co-prime

    M = m_1*m_2*m_3

    u_i = Modular inverse of (M/m_i) with respect m_i

    X = ( a_1 * (M/m_1) * u_1 + a_2 * (M/m_2) * u_2 + a_3 * (M/m_3) * u_3 ) % M
*/
```

Lucas Theorem & Modulo Operation with Composite Number

```
/*
    If we need to find nCr % P where P is a prime but P can be
    less than n or r, we can use Lucas Theorem.

    nCm = ((n_0 C r_0) * (n_1 C r_1) * (n_2 C r_2) * ... * (n_k C r_k)) % P

    Where n_i is the i'th digit in P based representation of n
    and r_i is the i'th digit in P based representation of r

    ** What if P is a composite number? **

    P = (p_0 ^ a_0) * (p_1 ^ a_1) * ... * (p_k ^ a_k)
    where all p_i are prime numbers.

    nCr = (n!)/((r)!*(n-r)!)

    If all a_i are 1, then we can use lucas to find individual mods for
    each p_i and combine those using CRT

    If any a_i is greater than 1,

    Let's Suppose,  n! = (p_i ^ u) * x
                    (n-r)! = (p_i ^ v) * y
                    (r)! = (p_i ^ w) * z
                    (See the code for calculation of x,y,z when
                     n or r has large value)

    Let's suppose p_i ^ a_i = t,
    gcd(t,x) = gcd(t,y) = gcd(t,z) = 1, so, x,y,z will have modular inverse
    with respect to t (see Note 1)

    So, we will find ( x /(y*z) ) % (p_i^a_i) and then multiply the
    result by (p_i ^ s) where s = u - v - w;
    If, s is not smaller than a_i, then the result is 0.
```

```

    Then, we will use CRT to combine the result.
    Actually, we don't need Lucas theorem anymore. This technique
    will work for a_i = 1 also.


    ****************Note 1******************
    phi(p^a) = (p^a) - (p^(a-1)) if p is prime

    a ^ phi(p^x) = 1 (mod p^x)  if gcd(a,p) = 1
    modular inverse of a with respect to p^a is
    a ^ ( phi(p^x) - 1 ) % (p^x)
    *******************************************
*/



// returns factorail(n) % (p^a) ignoring prime number p
// can be done using a loop if n is small

ll fact[MAX]; // size at least p^a

ll call(ll n,ll p,ll a)
{
    ll ret = 1LL;
    ll x,m = 1;

    //m = p^a
    for(int i=1;i<=a;i++) m *= p;

    fact[0] = 1;
    for(ll i=1;i<=m;i++){
        if(i%p==0) fact[i] = fact[i-1];
        else fact[i] = (fact[i-1]*i)%m;
    }

    while(true){
        if(n==0) break;

        y = n/m;
        ret *= bm(fact[m],y,m);
        ret %= m;

        y = n%m;

        ret *= fact[y];
        ret %= m;
        n /= p;
    }
    return ret;
}
```

Fibonacci Numbers

$F_{n+1}F_{n-1} - F_n^2 = (-1)^n.$

$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.$

$F_{2n} = F_n(F_{n+1} + F_{n-1}).$

F_{nk} is always a multiple of F_n

$\gcd(F_m, F_n) = F_{\gcd(m,n)}.$

$F_n = \frac{\sqrt{5}}{5}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$

Knuth Morris Pratt

```
/* Complexity = O(P+S) */
/* Searches pat in str */

char str[MAX],pat[MAX];
int pref[MAX];
int match[MAX];

//pref[i] =   length of the longest suffix which is also
            //a proper prefix of theoriginal string
void prefixFunction(int P){
    int j=0;
    for(int i=1; i<P; i++){
        while(true){
            if(pat[i]==pat[j]) {
                j = pref[i] = j+1;
                break;
            }
            else {
                if(j==0) {
                    pref[i] = 0;
                    break;
                }
                else j = pref[j-1];
            }
        }
    }
}

void KMPMatcher(int S) {
    int j = 0;
    for(int i=0; i<S; i++) {
        while(true) {
            if(str[i]==pat[j]) {
                j = match[i] = j+1;
                break;
            }
            else {
                if(j==0) {
                    match[i] = 0;
                    break;
                }
                else j = pref[j-1];
            }
        }
    }
}

int main(){
    scanf("%s",str);
    scanf("%s",pat);
    int S = strlen(str);
    int P = strlen(pat);

    prefixFunction(P);
    KMPMatcher(S);
    return 0;
```

```
}

/*
    NOTES

    We can avoid the matcher by calculating
    the prefix function for the string = P + "#" + S
*/
```

Suffix Array

```
/*
    Suffix Array contains the lexicographically sorted order of the
    suffixes of string(stores the starting index of the suffix only)
*/

#define MAXL 10100 // max length of the string
#define MAXLG 15

char str[MAXL];

struct entry{
    int pr[2]; // parameters for sorting
    int id; // starting index of the suffix
}suf[MAXL];

int P[MAXLG+5][MAXL+5];
// P[i][j] = position of the suffix starting at character j after sorting
// on the basis of 2^i characters

bool cmp(entry a,entry b){
    if(a.pr[0]==b.pr[0]) return a.pr[1]<b.pr[1];
    else return a.pr[0]<b.pr[0];
}


// complexity n * lg n * lg n
void generateSA(int L){
    // suf[i].id will contain the starting index of the i'th suffix in suffix array
    int stp,now,i;

    for(i=0;i<L;i++) P[0][i] = str[i]-'a'+1;

    for(now=1,stp=1 ; now<L ; stp++,now *= 2){
        for(i=0;i<L;i++){
            suf[i].pr[0] = P[stp-1][i];
            if(i+now<L) suf[i].pr[1] = P[stp-1][i+now];
            else suf[i].pr[1] = -1;
            suf[i].id = i;
        }
        sort(suf,suf+L,cmp);
        for(i=0;i<L;i++)
            if(i>0 && suf[i].pr[0]==suf[i-1].pr[0] && suf[i].pr[1]==suf[i-1].pr[1])
                P[stp][suf[i].id] = P[stp][suf[i-1].id];
            else
                P[stp][suf[i].id] = i+1;
    }
}
```

```

// complexity lg n
int getLCP(int x,int y,int L)
{
    int ret = 0,add,i;
    for(i=MAXLG;i>=0;i--){
        if(P[i][x]==P[i][y] && P[i][x]!=0){
            add = (1<<i);

            ret += add;

            x += add;
            y += add;
        }
    }
    return ret;
}


int main(){
    CLR(P); //don't forget
    gets(str);
    L = strlen(str);
    generateSA(L);
    return 0;
}
```

Longest Increasing Subsequence (nlog(n))

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

int ara[MAX];
vector <int> v;
int max_lis = 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_lis = max(max_lis,(int)v.size());
}
cout << "The size of the lis is : " << max_lis << endl;
```