

//Binary Indexed Tree

```

#define MAX 1010
//normally 1 based index
int tree[MAX];
int MaxVal;//always should be set(size of the set len)

//cumulative sum
int query(int idx) {
    if(idx<=0) return 0;
    int sum = 0;
    idx =min(idx,MaxVal);
    while (idx > 0) {
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}

void update(int idx,int val) {
    if(idx<=0) return;
    while (idx <= MaxVal) {
        tree[idx] += val;
        idx += (idx & -idx);
    }
}

//single indexed value
int querySingle(int idx) {
    int sum = tree[idx];
    // sum will be decreased
    if (idx > 0) {
        // special case
        int z = idx - (idx & -idx);
        // make z first
        idx--;
        // idx is no important any more, so instead y,
        you can use idx
        while (idx != z) {
            // at some iteration idx (y) will become z
            sum -= tree[idx];
            // substruct tree frequency which is between
            y and "the same path"
            idx -= (idx & -idx);
        }
    }
    return sum;
}

void scale(int c) {
    for (int i = 1 ; i <= MaxVal ; i++)
        tree[i] = tree[i] / c;
}

// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// some of them (we do not know which one)

// bitMask - initially, it is the greatest bit of MaxVal
// bitMask store interval which should be searched
int find(int cumFre) {
    int idx = 0; // this var is result of function

    while ((bitMask != 0) && (idx < MaxVal)) {
        int tIdx = idx + bitMask;
        // we make midpoint of interval
        if (cumFre == tree[tIdx])
            // if it is equal, we just return idx
            return tIdx;
        else if (cumFre > tree[tIdx]) {
            // if tree frequency "can fit" into cumFre,
            // then include it
            idx = tIdx; // update index

```

```

        cumFre -= tree[tIdx];
    }
    // set frequency for next loop
    }
    bitMask >>= 1;
    // half current interval
    }
    if (cumFre != 0)
    // maybe given cumulative frequency doesn't exist
    return -1;
    else
    return idx;
}

// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// the greatest one
int findG(int cumFre) {
    int idx = 0;

    while ((bitMask != 0) && (idx < MaxVal)) {
        int tIdx = idx + bitMask;
        if (cumFre >= tree[tIdx]) {
            // if current cumulative frequency is equal to cumFre,
            // we are still looking for higher index (if exists)
            idx = tIdx;
            cumFre -= tree[tIdx];
        }
        bitMask >>= 1;
    }
    if (cumFre != 0)
        return -1;
    else
        return idx;
}

//for 2D BIT
int query(int x, int y) {
    int y1;
    int sum=0;
    while (x > 0) {
        y1 = y;
        while (y1 > 0) {
            sum += tree[x][y1];
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }
    return sum;
}

void update(int x, int y, int val) {
    int y1;
    while (x <= max_x) {
        y1 = y;
        while (y1 <= max_y) {
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}

//2D BIT Ended

```

//Articulation Point

```
// 1 based, nodes starts from 1
#define MAXN 10005
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];
//tin is here the array for discovery time, and fup for
the discovery time lowest
```

```
set<int> result_of_articulation_points;
void IS_CUTPOINT(int u) {
    result_of_articulation_points.insert(u);
    return;
}
```

```
void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    int children = 0;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (to == p) continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] >= tin[v] && p != -1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if (p == -1 && children > 1)
        IS_CUTPOINT(v);
}
```

```
int main() {
    int n=II,m=II;
    //n=number of vertexes and m= number of the edges.
    for(int i=1; i<=m; i++) {
        int u=II, v=II;
        g[u].pb(v);
        g[v].pb(u);
    }
    timer = 0;
    for (int i=1; i<=n; ++i)
        used[i] = false;
    dfs (1);
    cout<<"Articulation points:\n";
    for(set<int>::iterator
it=result_of_articulation_points.begin();
it!=result_of_articulation_points.end(); it++) {
        cout<<*it<<endl;
    }
    return 0;
}
```

//Bridge

```
//this is a 1 based code
const int MAXN = 10005;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];
set<pi> res_bridge;
void IS_BRIDGE(int v, int to) {
    res_bridge.insert(pi(v,to));
    return;
}
void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
```

```
        if (to == p) continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] > tin[v])
                IS_BRIDGE(v,to);
        }
    }
}
void find_bridges(int n) {
    timer = 0;
    for (int i=1; i<=n; ++i)
        used[i] = false;
    for (int i=1; i<=n; ++i)
        if (!used[i])
            dfs (i);
}
```

```
int main() {
    int n=II,m=II;
    //n=number of vertexes and m= number of the edges.
    for(int i=1; i<=m; i++) {
        int u=II, v=II;
        g[u].pb(v);
        g[v].pb(u);
    }
    timer = 0;
    find_bridges(n);
    cout<<"Bridges:\n";
    for(set<pi>::iterator it=res_bridge.begin();
it!=res_bridge.end(); it++) {
        cout<<it->xx <<" "<<it->yy <<endl;
    }
    return 0;
}
```

//Bellman-Ford Algorithm

```
//complexity VE
#define SIZE 1010
//define INF 2000000000
vi adj[SIZE],cost[SIZE];

//0 based
bool BellmanFord(int source,int nodes) {
    //returns true if it has negative cycle
    vector<ll>dist;
    int i,j,k,w,v;

    for(i=0; i<=nodes; i++) { //distance from source
        dist.push_back(INF);
    }
    dist[source]=0;

    for(i=1; i<=nodes-1; i++) {
        //this iterator for the update
        for(j=1; j<=nodes; j++)
            //current and the next iterator is for edges to go
            for(k=0; k<adj[j].size(); k++) {
                v=adj[j][k];
                w=cost[j][k];
                if(dist[j]!=INF)
                    dist[v]=min(dist[v],dist[j]+w);
            }
    }

    for(i=1; i<=nodes; i++)
        for(j=0; j<adj[i].size(); j++) {
            v=adj[i][j];
            w=cost[i][j];
            if(dist[v]>dist[i]+w)
```

```

        return true;
    }
    return false;
}
// ANOTHER WAY OF BELLMAN-FORD. BECAREFULL DEFINING THE
INFINITY.
//complexity VE
#define SIZE 1010
// #define INF 2000000000
vi adjList[SIZE], RadjList[SIZE];
vector<ll> dist(SIZE);
bool vis[SIZE];
vi res;
//0 based
struct graph {
    int u, v, w;
    graph( int x, int y, int z) {
        u=x, v=y, w=z;
    }
};
vector<graph > gr;
void dfs(int u) {
    vis[u]=1;
    res.pb(u);
    for(int i=0; i<adjList[u].size(); i++) {
        int v=adjList[u][i];
        if(!vis[v])
            dfs(v);
    }
}
bool BellmanFord(int nodes) {
    //returns true if it has negative cycle
    ll i,j,k,w,v,u;
    for(i=0; i<nodes+2; i++)
        dist[i]=INF;
    for(i=1; i<nodes; i++) {
        //number of iteration to do. For bellman ford algorithm
        //it is 1 to n-1
        for(j=0; j<gr.size(); j++) {
            //current and next loop the edges to visit
            u=gr[j].u;
            v=gr[j].v;
            w=gr[j].w;
            dist[v]=min(dist[v], dist[u]+w);
        }
    }
    bool foo=0;
    for(j=0; j<gr.size(); j++) {
        u=gr[j].u;
        v=gr[j].v;
        w=gr[j].w;
        if(dist[v]>dist[u]+w) {
            foo=1;
            dist[v]=dist[u]+w;
            if(!vis[u])
                dfs(u);
        }
    }
    return foo;
}

```

//Finding Euler Path in O(Vertex)

```

int n;
vector< vector<int> > g (n, vector<int> (n));
//Here to Read the graph
vector<int> deg (n);
for (int i=0; i<n; ++i)
    for (int j=0; j<n; ++j)
        deg[i] += g[i][j];

int first = 0;

```

```

while (!deg[first])
    ++first;

int v1 = -1, v2 = -1;
bool bad = false;
for (int i=0; i<n; ++i)
    if (deg[i] & 1)
        if (v1 == -1)
            v1 = i;
        else if (v2 == -1)
            v2 = i;
        else
            bad = true;

if (v1 != -1)
    ++g[v1][v2], ++g[v2][v1];

stack<int> st;
st.push (first);
vector<int> res;
while (!st.empty()) {
    int v = st.top();
    int i;
    for (i=0; i<n; ++i)
        if (g[v][i])
            break;
    if (i == n) {
        res.push_back (v);
        st.pop();
    } else {
        --g[v][i];
        --g[i][v];
        st.push (i);
    }
}

if (v1 != -1)
    for (size_t i=0; i<res.size(); ++i)
        if (res[i] == v1 && res[i+1] == v2 || res[i] ==
v2 && res[i+1] == v1) {
            vector<int> res2;
            for (size_t j=i+1; j<res.size(); ++j)
                res2.push_back (res[j]);
            for (size_t j=1; j<=i; ++j)
                res2.push_back (res[j]);
            res = res2;
            break;
        }

for (int i=0; i<n; ++i)
    for (int j=0; j<n; ++j)
        if (g[i][j])
            bad = true;

if (bad)
    puts ("-1");
else
    for (size_t i=0; i<res.size(); ++i)
        printf ("%d ", res[i]+1);

```

//Floyd Worshell

```

int V, E, u, v, w, AdjMatrix[200][200];
/*
// Graph in Figure 4.30
5 9
0 1 2
0 2 1
0 4 3
1 3 4
2 1 1
2 4 1
3 0 1

```

```

3 2 3
3 4 5
*/
scanf("%d %d", &V, &E);
for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++)
        AdjMatrix[i][j] = INF;
    AdjMatrix[i][i] = 0;
}

for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjMatrix[u][v] = w; // directed graph
}

for (int k = 0; k < V; k++)
// common error: remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            AdjMatrix[i][j] = min(AdjMatrix[i][j],
AdjMatrix[i][k] + AdjMatrix[k][j]);

for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        printf("APSP(%d, %d) = %d\n", i, j,
AdjMatrix[i][j]);

```

//Dijkstra's Algorithm

```

int mx = 20005;
int costMatrix[502][502];
vi adjList[502];
bool vis[502];
int dist[502];
priority_queue<int, vector<pi>, greater<pi> > q;
void Dijkstra(int source) {
    dist[source]=0;
    q.push(pi(0, source));
    while(!q.empty()) {
        pi x=q.top();
        int u=x.yy;
        q.pop();
        if(!vis[u]) {
            for(int i=0; i<adjList[u].size(); i++) {
                int v=adjList[u][i];
                int
max_cost=max(dist[u], costMatrix[u][v]);
                if(max_cost<dist[v]) {
                    dist[v]=max_cost;
                    q.push(pi(dist[v], v));
                }
            }
            vis[u]=1;
        }
    }
}

int main() {
    int t=II;
    for(int cs=1; cs<=t; cs++) {
        for(int i=0; i<502; i++)
            adjList[i].clear();
        for(int i=0; i<502; i++)
            dist[i]=mx;
        clr(vis, 0);
        for(int i=0; i<502; i++)
            for(int j=0; j<502; j++)
                costMatrix[i][j]=mx;
        int nodes=II, edges=II;
        for(int i=1; i<=edges; i++) {
            int u=II, v=II, cost=II;
            if(costMatrix[u][v]==mx) {
                costMatrix[u][v]=cost;
                costMatrix[v][u]=cost;
            }
        }
    }
}

```

```

        adjList[u].pb(v);
        adjList[v].pb(u);
    } else {
        int mini=min(cost, costMatrix[u][v]);
        costMatrix[u][v]=mini;
        costMatrix[v][u]=mini;
    }
}

int source=II;
Dijkstra(source);
pf("Case %d:\n", cs);
for(int i=0; i<nodes; i++) {
    if(dist[i]==mx)
        pf("Impossible\n");
    else
        pf("%d\n", dist[i]);
}
}

return 0;
}

```

//Minimum Spanning Tree

```

//Kruskal's Algorithm O(m lgn)
#define node_sz 100
#define edge_sz 1000
vector<pair<int, pair<int, int> > > AdjList; //
G(COST, (U, V))
vector<pair<int, int> > res;
int n, m, cost;
vector<int> p(node_sz);
int dsu_get(int v) {
    return (v == p[v]) ? v : (p[v]=dsu_get(p[v]));
}

void dsu_unite(int a, int b) {
    a = dsu_get(a);
    b = dsu_get(b);
    if (rand() & 1)
        swap(a, b);
    if (a != b)
        p[a] = b;
}

void KruskalsAlgorithm() {
    cost = 0;
    sort(all(AdjList));
    p.resize(node_sz);
    for (int i=0; i<node_sz; ++i)
        p[i]=i;
    for(int i=0; i<m; ++i) {
        int a = AdjList[i].yy.xx, b = AdjList[i].yy.yy, 1
= AdjList[i].xx;
        if (dsu_get(a) != dsu_get(b)) {
            cost += 1;
            res.pb(AdjList[i].second);
            dsu_unite(a, b);
        }
    }
}

int main() {
    n=II, m=II;
    AdjList.clear();
    p.clear();
    for(int i=1; i<=m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        AdjList.pb(mp(w, mp(u, v)));
        AdjList.pb(mp(w, mp(v, u)));
    }

    KruskalsAlgorithm();
    cout << cost << endl;
}

```

```

    return 0;
}

//Kruskal's Algorithm O(m^2 + mlg n)
#define edge_sz 1000
#define node_sz 100
int m,n;
vector<pair<int, pair<int, int> > > AdjList;
// G(COST, (U,V))
vector<int> tree_id(node_sz);
int cost=0;
vector<pair<int, int> > res;
void Kruskals_Algorithm() {
    sort(all(AdjList));
    cost=0;
    for (int i=0; i<node_sz; ++i)
        tree_id[i]=i;
    for (int i=0; i<m; i++) {
        int a=AdjList[i].yy.xx, b=AdjList[i].yy.yy,
        l=AdjList[i].xx;
        if (tree_id[a]!=tree_id[b]) {
            cost+=l;
            res.pb(mp(a, b));
            int old_id=tree_id[b], new_id=tree_id[a];
            for (int j=0; j<n; ++j)
                if (tree_id[j] == old_id)
                    tree_id[j] = new_id;
        }
    }
    return;
}

int main() {
    n=II, m=II;
    AdjList.clear();
    tree_id.clear();
    for(int i=1; i<=m; i++) {
        int u,v,w;
        cin>>u>>v>>w;
        AdjList.pb(mp(w, mp(u, v)));
        AdjList.pb(mp(w, mp(v, u)));
    }

    Kruskals_Algorithm();
    cout<<cost<<endl;
    for(int i=0; i<res.size(); i++)
        cout<<res[i].xx<<" "<<res[i].yy<<endl;
    return 0;
}

//Prim's Algorithm for Sparse Graph
int n;
int mst_cost;
vector< vector< pair <int, int> > > AdjList;
vector<bool> taken;
// global boolean flag to avoid cycle
priority_queue<pi> pq;
// priority queue to help choose shorter edges

void process(int vtx) {
    // so, we use -ve sign to reverse the sort order
    taken[vtx] = 1;
    for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
        pi v = AdjList[vtx][j];
        if (!taken[v.xx])
            pq.push(mp(-v.yy, -v.xx));
    }
}

int main() {
    int m;
    n=II, m=II;
    AdjList.resize(n+3);
    for(int i=1; i<=m; i++) {

```

```

        int u=II, v=II, w=II;
        u--,v--; //making the zero based vertex
        AdjList[u].pb(mp(v,w));
        AdjList[v].pb(mp(u,w));
    }
    taken.assign(n+1, 0);
    // no vertex is taken at the beginning
    process(0);
    //take vertex 0 and process all edges incident to vertex
    0
    mst_cost = 0;
    while (!pq.empty()) {
        // repeat until V vertices (E=V-1 edges) are taken
        pi front = pq.top();
        pq.pop();
        int u = -front.second, w = -front.first;
        // negate the id and weight again
        if (!taken[u])
            // we have not connected this vertex yet
            mst_cost += w, process(u);
        // take u, process all edges incident to u
    }
    // each edge is in pq only once!
    printf("MST cost = %d (Prim's)\n", mst_cost);
    return 0;
}

```

//Strongly Connected Component

//Tarjan SCC

```

#define lim 1005 //number of nodes
//No problem in multiple edges and self loop
//0 based
VI adj[lim]; //only adj should be cleared
int col[lim], low[lim], tim[lim], timer;
int group_id[lim], n, m, components; //components=number of
components group_id = which node belongs to which node
stack<int> S;

void scc(int u) {
    int i, v, tem;
    col[u]=1;
    low[u]=tim[u]=timer++;
    S.push(u);
    fr(i, 0, SZ(adj[u])-1) {
        v=adj[u][i];
        if (col[v]==1)
            low[u]=min(low[u], tim[v]);
        else if (col[v]==0) {
            scc(v);
            low[u]=min(low[u], low[v]);
        }
    }
}

//SCC checking...
if (low[u]==tim[u]) {
    do {
        tem=S.top();
        S.pop();
        group_id[tem]=components;
        col[tem]=2; //Completed...
    } while (tem!=u);
    components++;
}
}

int TarjanSCC(int n) { //some change may be required here
    int i;
    timer=components=0;
    mem(col, 0);
    while (!S.empty()) S.pop();
    fr(i, 0, n-1) if (col[i]==0) scc(i);
    return components;
}

```

```

}

VI nadj[lim]; //new adjacency list after SCC(DAG)
void MakeNewDAG_Graph(int n) {
    int i,j,u,v;

    fr(i,0,components-1) nadj[i].clear();

    fr(i,0,n-1) {
        fr(j,0,SZ(adj[i])-1) {
            u=group_id[i];
            v=group_id[adj[i][j]];
            if(u!=v)
                nadj[u].pb(v);
        }
    }
}

void add(int ina,int inb) {
    adj[ina].pb(inb);
}

int main() {
    int i,j,t,cas=0,u,v,ans;

    while(scanf("%d %d",&n,&m)==2) {
        fr(i,0,n-1) adj[i].clear();
        fr(i,1,m) {
            scanf("%d %d",&u,&v);
            adj[u].pb(v);
        }
        TarjanSCC(n);
        printf("Total Groups: %d\n",components);
        MakeNewDAG_Graph(n);
        printf("NewGraphLinkUsingSCC: this graph is
directed acyclic graph:\n");
        //this link between groups no.....
        fr(i,0,components-1) {
            fr(j,0,SZ(nadj[i])-1) {
                u=i;
                v=nadj[i][j];
                print2(u,v);
            }
        }
        return 0;
    }
}
/*
Input:
8 14
0 1
1 2
1 5
1 4
2 6
2 3
3 2
3 7
4 5
5 6
7 6
7 3
6 5
4 0
Output:
Total Groups: 3
NewGraphLinkUsingSCC: this graph is directed acyclic
graph:
1 0
1 0
2 1
2 0
2 0

```

Another Input:

```

6 6
0 1
1 2
2 1
3 4
4 5
5 4
Total Groups: 4
NewGraphLinkUsingSCC: this graph is directed acyclic
graph:
1 0
3 2
*/

```

//Kosaraju's Algorithm for SCC by 2DFS

//Kosaraju's Algorithm Learned from <http://emaxx.ru> - Hard copy

```

#define sz 101
vi adjList[sz], RadjList[sz];
vector<bool> used;
vi order,component;
void dfs1(int u) {
    used[u]=1;
    for(int i=0; i<adjList[u].size(); i++) {
        int v=adjList[u][i];
        if(!used[v])
            dfs1(v);
    }
    order.pb(u);
}

void dfs2(int u) {
    used[u]=1;
    component.pb(u);
    //here we can denote the graph node belongs to which
    //strongly connected components
    for(int i=0; i<RadjList[u].size(); i++) {
        int v=RadjList[u][i];
        if(!used[v])
            dfs2(v);
    }
}

int main() {
    int n=II,m=II; //number of nodes, edges
    for(int i=1; i<=m; i++) {
        int u=II, v=II;
        adjList[u].pb(v);
        RadjList[v].pb(u);
    }
    used.assign(n+2, false);
    for(int i=1; i<=n; i++) {
        if(!used[i])
            dfs1(i);
    }
    used.assign(n+2, false);
    reverse(all(order));
    for(int i=0; i<order.size(); i++) {
        int v=order[i];
        if(!used[v]) {
            cout<<v<<"=";
            dfs2(v);
            for(int j=0; j<component.size(); j++)
                pf(" %d",component[j]);
            pf("\n");
            component.clear();
        }
    }
    return 0;
}

```

}

****Flow Related****//BPM by Ford-Fulkerson**

// A C++ program to find maximal Bipartite matching.

#include <iostream>

#include <string.h>

using namespace std;

// M is number of applicants and N is number of jobs

#define M 6

#define N 6

// A DFS based recursive function that returns true if a
// matching for vertex u is possiblebool bpm(bool bpGraph[M][N], int u, bool seen[], int
matchR[]) {

// Try every job one by one

for (int v = 0; v < N; v++) {

 // If applicant u is interested in job v and v is
 // not visited

if (bpGraph[u][v] && !seen[v]) {

seen[v] = true; // Mark v as visited

 // If job 'v' is not assigned to an applicant OR
 // previously assigned applicant for job v

(which is matchR[v])

// has an alternate job available.

// Since v is marked as visited in the above

line, matchR[v]

// in the following recursive call will not

get job 'v' again

if (matchR[v] < 0 || bpm(bpGraph, matchR[v],

seen, matchR)) {

matchR[v] = u;

return true;

}

}

}

return false;

}

// Returns maximum number of matching from M to N

int maxBPM(bool bpGraph[M][N]) {

 // An array to keep track of the applicants assigned
 to // jobs. The value of matchR[i] is the applicant
 number // assigned to job i, the value -1 indicates nobody
 is

// assigned.

int matchR[N];

// Initially all jobs are available

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

int result = 0;

// Count of jobs assigned to applicants

for (int u = 0; u < M; u++) {

// Mark all jobs as not seen for next applicant.

bool seen[N];

memset(seen, 0, sizeof(seen));

// Find if the applicant 'u' can get a job

if (bpm(bpGraph, u, seen, matchR))

result++;

}

return result;

}

// Driver program to test above functions

int main() {

// Let us create a bpGraph shown in the above example

bool bpGraph[M][N] = { {0, 1, 1, 0, 0, 0},

{1, 0, 0, 1, 0, 0},

{0, 0, 1, 0, 0, 0},

{0, 0, 1, 1, 0, 0},

{0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 1}

};

 cout << "Maximum number of applicants that can get
job is "

<< maxBPM(bpGraph);

return 0;

}

//BPM by Blossom Algo, BPM by HopCraft

See in CookBook Palindrome, SUST

//Max Flow Min Cut//Edmond's Karp//This Algorithm takes $O(VE^2)$ (actually $O(V^3E)$)

#define MAX_V 40

int res[MAX_V][MAX_V], mf, f, s, t;

// global variables

vi p;

vector<vi> AdjList;

void augment(int v, int minEdge) {

// traverse BFS spanning tree from s to t

if (v == s) {

f = minEdge;

// record minEdge in a global variable f

return;

} else if (p[v] != -1) {

augment(p[v], min(minEdge, res[p[v]][v]));

// recursive

res[p[v]][v] -= f;

res[v][p[v]] += f;

} // update

}

int main() {

int V, k, vertex, weight;

/*

// Graph in Figure 4.24

4 0 1

2 2 70 3 30

2 2 25 3 70

3 0 70 3 5 1 25

3 0 30 2 5 1 70

// Graph in Figure 4.25

4 0 3

2 1 100 3 100

2 2 1 3 100

1 3 100

0

// Graph in Figure 4.26.A

5 1 0

0

2 2 100 3 50

3 3 50 4 50 0 50

1 4 100

1 0 125

// Graph in Figure 4.26.B

5 1 0

0

2 2 100 3 50

3 3 50 4 50 0 50

1 4 100

```

1 0 75

// Graph in Figure 4.26.C
5 1 0
0
2 2 100 3 50
2 4 5 0 5
1 4 100
1 0 125
*/
scanf("%d %d %d", &V, &s, &t);

memset(res, 0, sizeof res);
AdjList.assign(V, vi());
for (int i = 0; i < V; i++) {
    scanf("%d", &k);
    for (int j = 0; j < k; j++) {
        scanf("%d %d", &vertex, &weight);
        res[i][vertex] = weight;
        AdjList[i].push_back(vertex);
    }
}

mf = 0;
while (1) {
// now a true O(VE^2) Edmonds Karp's algorithm
    f = 0;
    bitset<MAX_V> vis;
    vis[s] = true;
// we change vi dist to bitset!
    queue<int> q;
    q.push(s);
    p.assign(MAX_V, -1);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        if (u == t) break;
        for (int j = 0; j < (int)AdjList[u].size();
j++) { // we use AdjList here!
            int v = AdjList[u][j];
            if (res[u][v] > 0 && !vis[v])
                vis[v] = true, q.push(v), p[v] = u;
        }
        augment(t, INF);
        if (f == 0) break;
        mf += f;
    }

    printf("%d\n", mf);
// this is the max flow value

    return 0;
}

```

//Dinic's Max Flow

See at COOKBOOK Palindrome, SUST

//Longest Common Subsequence

```

string a,b,ans_str;
ll len1,len2;
bool visit[mod][mod];
ll dp[mod][mod];
ll calLCS(ll i, ll j) {
    if(a[i]=='\0' || b[j]=='\0')
        return 0;
    if(visit[i][j]==true)
        return dp[i][j];
    ll ans=0;
    if(a[i]==b[j])
        ans = 1 + calLCS(i+1,j+1);
    else {

```

```

        ll v1 = calLCS(i+1,j);
        ll v2 = calLCS(i,j+1);
        ans =max (v1,v2);
    }
    dp[i][j]=ans;
    visit[i][j]=true;
    return dp[i][j];
}

void printLCS(ll i, ll j) {
    if(a[i]=='\0' || b[j]=='\0') {
        cout<<ans_str<<endl;
        return;
    }
    if(a[i]==b[j]) {
        ans_str+=a[i];
        printLCS(i+1,j+1);
        ans_str.erase(ans_str.end()-1);
    } else {
        if(dp[i+1][j]>dp[i][j+1])
            printLCS(i+1,j);
        else if(dp[i+1][j]<dp[i][j+1])
            printLCS(i,j+1);
        else {
            printLCS(i,j+1);
            printLCS(i+1,j);
        }
    }
}

int main() {
    cin>>a>>b;
    len1 = a.length();
    len2 = b.length();
    cout<<calLCS(0,0)<<"\n";
    printLCS(0,0);
    return 0;
}

```

//Longest Increasing Subsequence

//n^2

```

int n; // the number of items in the sequence
int Sequence[32]; // the sequence of integers
int L[32]; // L[] as described in the algorithm

```

```

void takeInput() {
    scanf("%d", &n); // how many numbers in the sequence?

    // take the sequence
    for( int i = 0; i < n; i++ )
        scanf("%d", &Sequence[i]);
}

```

```

int LIS() {
// which runs the LIS algorithm and returns the result
    int i, j; // auxiliary variables for iteration

    // initialize L[] with 1
    for( i = 0; i < n; i++ )
        L[i] = 1;

    // start from the left most item and iterate right
    for( i = 0; i < n; i++ ) {
// for the ith item find all items that are in right
        for( j = i + 1; j < n; j++ ) {

            if( Sequence[j] > Sequence[i] ) {
// the item is greater than the ith item
// so, L[j] = L[i] + 1, since jth item can be added after
// ith
// item. if L[j] is already greater than or equal to L[i]
+ 1

                // then ignore it
                if( L[j] < L[i] + 1 )

```



```

        L[j] = L[i] + 1;
    }
}
// now find the item whose L[] value is maximum
int maxLength = 0;
for( i = 0; i < n; i++ ) {
    if( maxLength < L[i] )
        maxLength = L[i];
}
// return the result
return maxLength;
}
int LisSequence[32]; // for storing the sequence

void findSequence( int maxLength ) {
// finds a valid sequence
    int i, j; // variable used for iteration

    // at first find the position of the item whose L[]
is maximum
    i = 0;
    for( j = 1; j < n; j++ ) {
        if( L[j] > L[i] )
            i = j;
    }

    // initialize the position in LisSequence where the
items can be added.
    // observe that the data are saving from right to
left!
    int top = L[i] - 1;

    // insert the item in i-th position to LisSequence
    LisSequence[top] = Sequence[i];
    top--;
// is decreasing such that a new item can be added in a
new place

    // now find the other valid numbers to form the
sequence
    for( j = i - 1; j >= 0; j-- ) {
        if( Sequence[j] < Sequence[i] && L[j] == L[i] - 1
) {
            // we have found a valid item so, we will save it
            i = j; // as in our algorithm
            LisSequence[top] = Sequence[i]; // stored
            top--; // decreased for new items
        }
    }

    // so, we have got the sequence, now we want to print it
    printf("LIS is");
    for( i = 0; i < maxLength; i++ ) {
        printf(" %d", LisSequence[i]);
    }
    puts("");
}

int main() {
    takeInput();
    int result = LIS();
    printf("The LIS length is %d\n", result);
    findSequence( result );
    return 0;
}

```

//nLgK

```

const int inf = 2000000000; // a large value as infinity
int n; // the number of items in the sequence
int Sequence[32]; // the sequence of integers
int L[32]; // L[] as described in the algorithm
int I[32]; // I[] as described in the algorithm

```

```

void takeInput() {
    scanf("%d", &n); // how many numbers in the sequence
?
    for( int i = 0; i < n; i++ ) // take the sequence
        scanf("%d", &Sequence[i]);
}

int LisNlogK() { // which runs the NlogK LIS algorithm
    int i; // auxiliary variable for iteration

    I[0] = -inf; // I[0] = -infinite
    for( i = 1; i <= n; i++ ) // observe that i <= n are
given
        I[i] = inf; // I[1 to n] = infinite

    int LisLength = 0; // keeps the maximum position
where a data is inserted

    for( i = 0; i < n; i++ ) { // iterate left to right
        int low, high, mid; // variables to perform
binary search
        low = 0; // minimum position where we to put data
in I[]
        high = LisLength; // the maximum position

        while( low <= high ) {
            // binary search to find the correct position
            mid = ( low + high ) / 2;
            if( I[mid] < Sequence[i] )
                low = mid + 1;
            else
                high = mid - 1;
        }
        // observe the binary search carefully, when the
binary search ends
        // low > high and we put our item in I[low]
        I[low] = Sequence[i];
        L[i]=low;
        if( LisLength < low )
            // LisLength contains the maximum position
            LisLength = low;
    }
    return LisLength; // return the result
}

int LisSequence[32]; // for storing the sequence
void findSequence( int maxLength ) {
// finds a valid sequence
    int i, j; // variable used for iteration
    // at first find the position of the item whose L[] is
maximum
    i = 0;
    for( j = 1; j < n; j++ ) {
        if( L[j] > L[i] )
            i = j;
    }

    // initialize the position in LisSequence where the
items can be added.
    // observe that the data are saving from right to
left!
    int top = L[i] - 1;
    // insert the item in i-th position to LisSequence
    LisSequence[top] = Sequence[i];
    top--; // is decreasing such that a new item can be
added in a new place

    // now find the other valid numbers to form the
sequence
    for( j = i - 1; j >= 0; j-- ) {
        if( Sequence[j] < Sequence[i] && L[j] == L[i] - 1
) {

```

```

        // we have found a valid item so, we will save it
        i = j; // as in our algorithm
        LisSequence[top] = Sequence[i]; // stored
        top--; // decreased for new items
    }
}
// so, we have got the sequence, now we want to print
it
printf("LIS is");
for( i = 0; i < maxLength; i++ ) {
    printf(" %d", LisSequence[i]);
}
puts("");
}
int main() {
    takeInput();
    int result = LisNlogK();
    printf("The LIS length is %d\n", result);
    findSequence(result);
    return 0;
}

```

//Longest Palindrome

```

//you are given a string, you can delete no, one or more
char. not insert to get longest palindrome.
string S;
#define MAX_V 10000
ll Len[MAX_V][MAX_V];
//in c++ initially zero '0' is assigned
ll longest_palindrome(ll l, ll r) {
    //l for left and r for right
    if(Len[l][r]!=0) return Len[l][r];
    // previously calculate
    if(l==r) return Len[l][r]=1; //base case for length 1
    if(l+1==r) { //base case for length 2
        if(S[l]==S[r])
            return Len[l][r]=2;
    }
    //if two character stay beside, and same then two
    return Len[l][r]=1;
}
if(S[l]==S[r])
    return Len[l][r]=2+longest_palindrome(l+1,r-1);
return Len[l][r]=max(longest_palindrome(l,r-1),longest_palindrome(l+1,r));
}

```

//Longest Common Substring

(Multiple String Using Delimeter)

```

//Link = Longest Proper suffix in suffix automata(already
exist). (next clear can be needed)
//Depth means the highest substring attainable towards
these node. Some strings are already attained by the link
of the node(the total depth of the link)
//Preprocess complexity nlogk (k=number of child)
struct state {
    int depth, link ;
    map < char, int > next ;
};
//by sacrificing memory we can make it linear
};
const int MAXLEN = 15010 ;
state st [MAXLEN * 2];
int global[MAXLEN * 2];
int sz, last ;

void sa_init( ) {
    sz=last =0 ;
    st[0].depth = 0 ;
    st[0].link = - 1 ;
    ++sz;

    //It's not always needed

```

```

    for (int i=0; i<MAXLEN*2; ++i)
        st[i].next.clear();
}

void sa_extend(char c ) {
    int cur = sz ++ ;
    st [ cur ] . depth = st [ last ] . depth + 1 ;
    int p;

    for ( p = last; p!= -1 && ! st [ p ] . next . count (
c ) ; p = st [ p ] . link )
        st [ p ] . next [ c ] = cur ;
    if ( p == - 1 )
        st [ cur ] . link = 0 ;
    else {
        int q = st [ p ] . next [ c ] ;
        if ( st [ p ] . depth + 1 == st [ q ] . depth )
            st [ cur ] . link = q ;
        else {
            int clone = sz ++ ;
            st[clone].depth = st [ p ] . depth + 1 ;
            st[clone].next = st [ q ] . next ;
            st[clone].link = st [ q ] . link ;
            for ( ; p!= -1 && st [ p ] . next [ c ] == q
; p = st [ p ] . link )
                st [ p ] . next [ c ] = clone ;
            st [ q ] . link = st [ cur ] . link = clone ;
        }
    }
    last = cur ;
}

string a,b,C,s;
int col[MAXLEN*2],dp[MAXLEN*2];

int getcol(int in,char ch) {
    if(col[in]!=-1) return col[in];
    if(ch<='2') return col[in]=1<<(ch-'0');

    col[in]=0;
    FORE(it,st[in].next)
        col[in]=getcol(it->second,it->first);
    return col[in];
}

int dprec(int in) {
    int &ret=dp[in];
    if(ret!=-1) return ret;
    ret=0;

    FORE(it,st[in].next)
        if(col[it->second]==7) //2^3-1
            ret=max(ret,dprec(it->second)+1);
    return ret;
}

int main() {
    int t,cas=0;
    cin>>t;
    while(t--) {
        cin>>a>>b>>c;
        s=a+"0"+b+"1"+C+"2";
        printf("Case %d: ",++cas);
        sa_init();
        int i;
        fr(i,0,SZ(s)-1)
            sa_extend(s[i]);

        mem(col,-1);
        mem(dp,-1);
        getcol(0,'3'); //3 doesn't mean anything
        printl(dprec(0));
    }
    return 0;
}

```

```

}

//Normal
string lcs (string s, string t) {
//longest common substring with length
    sa_init();
    for (int i=0; i<(int)s.length(); ++i)
        sa_extend (s[i]);

    int v = 0, l = 0,
        best = 0, bestpos = 0;
    for (int i=0; i<(int)t.length(); ++i) {
        while (v && ! st[v].next.count(t[i])) {
            v = st[v].link;
            l = st[v].depth;
        }
        if (st[v].next.count(t[i])) {
            v = st[v].next[t[i]];
            ++l;
        }
        if (l > best)
            best = l, bestpos = i;
    }
    return t.substr (bestpos-best+1, best);
}

//Suffix Tree
// A C program to implement Ukkonen's Suffix Tree
Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will connect
    two nodes, one parent and one child, and (start, end)
    interval of a given edge will be stored in the child node.
    Lets say there are two nodes A and B connected by an edge
    with indices (5, 8) then this indices (5, 8) will be stored
    in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for the
    path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get a
new suffix link (other than root) in next extension of same
phase. lastNewNode will be set to NULL when last newly
created internal node (if there is any) got it's suffix
link reset to new internal node created in next extension
of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;

```

```

int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end) {
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode) {
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater than
    current edge length, set next internal node as activeNode
    and adjust activeEdge and activeLength accordingly to
    represent same activePoint*/
    if (activeLength >= edgeLength(currNode)) {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos) {
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in
    tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
    }
}

```

```

// activeEdge from activeNode
if (activeNode->children[text[activeEdge]] ==
NULL) {
    //Extension Rule 2 (A new leaf edge gets
created)
    activeNode->children[text[activeEdge]] =
        newNode(pos, &leafEnd);

/*A new leaf edge is created in above line starting from
an existng node (the current activeNode), and if there is
any internal node waiting for it's suffix link get reset,
point the suffix link from that last internal node to
current activeNode. Then set lastNewNode to NULL indicating
no more node waiting for suffix link reset.*/
    if (lastNewNode != NULL) {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }
}

// There is an outgoing edge starting with activeEdge
// from activeNode
else {

// Get the next node at the end of edge starting
// with activeEdge
Node *next = activeNode-
>children[text[activeEdge]];
    if (walkDown(next)) { //Do walkdown
        //Start from next node (the new activeNode)
        continue;
    }

/*Extension Rule 3 (current character being processed is
already on the edge)*/
    if (text[next->start + activeLength] ==
text[pos]) {
        //If a newly created node waiting for it's
//suffix link to be set, then set suffix link
//of that waiting node to curent active node
        if (lastNewNode != NULL && activeNode !=
root) {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;

/*STOP all further processing in this phase and move on
to next phase*/
        break;
    }

/*We will be here when activePoint is in middle of the edge
being traversed and current character being processed is
not on the edge (we fall off the tree). In this case, we
add a new internal node and a new leaf edge going out of
that new node. This is Extension Rule 2, where a new leaf
edge and a new internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] =
split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos,
&leafEnd);
    next->start += activeLength;
    split->children[text[next->start]] = next;

```

```

/*We got a new internal node here. If there is any internal
node created in last extensions of same phase which is
still waiting for it's suffix link reset, do it now.*/
    if (lastNewNode != NULL) {
        /*suffixLink of lastNewNode points to current newly
        created internal node*/
        lastNewNode->suffixLink = split;
    }

/*Make the current newly created internal node waiting for
it's suffix link reset (which is pointing to root at
present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when Extension
Rule 2 applies is any of the next extension of same phase)
at that point, suffixLink of this node will point to that
internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
    remainingSuffixCount--;
    if (activeNode == root && activeLength > 0) {
//APCFER2C1
        activeLength--;
        activeEdge = pos - remainingSuffixCount + 1;
    } else if (activeNode != root) { //APCFER2C2
        activeNode = activeNode->suffixLink;
    }
}

void print(int i, int j) {
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix
index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight) {
    if (n == NULL) return;

    if (n->start != -1) { //A non-root node
//Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++) {
        if (n->children[i] != NULL) {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);

//Current node is not a leaf as it has outgoing
//edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i],
labelHeight +
                                edgeLength(n-
>children[i]));
        }
    }
    if (leaf == 1) {
        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n) {

```

```

    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++) {
        if (n->children[i] != NULL) {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along
with suffixIndex. suffixIndex for leaf edges will be >= 0
and for non-leaf edges will be -1*/
void buildSuffixTree() {
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[]) {
    strcpy(text, "abcdabcd");
    buildSuffixTree();
    // strcpy(text, "xabxac#"); buildSuffixTree();
    // strcpy(text, "xabxa"); buildSuffixTree();
    // strcpy(text, "xabxa$"); buildSuffixTree();
    // strcpy(text, "abcabxabcd$"); buildSuffixTree();
    // strcpy(text, "geeksforgeeks$"); buildSuffixTree();
    // strcpy(text, "THIS IS A TEST TEXT$");
    buildSuffixTree();
    // strcpy(text, "AABAACAADAABAAABAA$");
    buildSuffixTree();
    return 0;
}

//String Alignment
//Minimum number of steps required
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
#define MIN(x,y,z) min(min((x),(y)),(z))
char A[20] = "TRAILS", B[20] = "ZEIL";
int n = (int)strlen(A), m = (int)strlen(B);
int i, j, table[20][20]; // Needleman Wunschn's algorithm
int dif(int i,int j) {
    if(A[i-1]!=B[j-1])
        return 1; // mismatch then 1 step needed
    return 0; //match then 0 step needed
}

int main() {
    memset(table, 0, sizeof table);
    // insert/delete / Replace char. to change A into B
    for (i = 1; i <= n; i++)
        table[i][0] = i ; //Steps to make i from 0 char.

```

```

        for (j = 1; j <= m; j++)
            table[0][j] = j; //Steps to make j from 0 char.

        for (j = 1; j <= m; j++)
            for (i = 1; i <= n; i++) {
                // minimum number of steps required for replace or match
                // delete / insert
                table[i][j] = MIN(table[i - 1][j - 1] +
                    dif(i,j), table[i-1][j]+1, table[i][j-1]+1);
                // steps for match or mismatches
            }

        printf("DP table:\n");
        for (j = 0; j <= m; j++) {
            for (i = 0; i <= n; i++)
                printf("%3d", table[i][j]);
            printf("\n");
        }
        printf("Minimum Number of steps Needed: %d\n",
            table[n][m]);

        return 0;
    }

    //Maximum Score or Needleman Wunschn's algorithm
    #include <algorithm>
    #include <cstdio>
    #include <cstring>
    using namespace std;

    int main() {
        char A[20] = "ACAATCC", B[20] = "AGCATGC";
        int n = (int)strlen(A), m = (int)strlen(B);
        int i, j, table[20][20]; // Needleman Wunschn's
        algorithm

        memset(table, 0, sizeof table);
        // insert/delete = -1 point
        for (i = 1; i <= n; i++)
            table[i][0] = i * -1;
        for (j = 1; j <= m; j++)
            table[0][j] = j * -1;

        for (i = 1; i <= n; i++)
            for (j = 1; j <= m; j++) {
                // match = 2 points, mismatch = -1 point
                table[i][j] = table[i - 1][j - 1] + (A[i - 1]
                    == B[j - 1] ? 2 : -1);
                // cost for match or mismatches
                // insert/delete = -1 point
                table[i][j] = max(table[i][j], table[i -
                    1][j] - 1); // delete
                table[i][j] = max(table[i][j], table[i][j -
                    1] - 1); // insert
            }

        printf("DP table:\n");
        for (i = 0; i <= n; i++) {
            for (j = 0; j <= m; j++)
                printf("%3d", table[i][j]);
            printf("\n");
        }
        printf("Maximum Alignment Score: %d\n", table[n][m]);

        return 0;
    }

    //Knuth - Morris Pratt Algorithm
    #include <cstdio>
    #include <cstring>
    #include <time.h>
    using namespace std;

```

```

#define MAX_N 100010

char T[MAX_N], P[MAX_N]; // T = text, P = pattern
int b[MAX_N], n, m;
// b = back table, n = length of T, m = length of P

void naiveMatching() {
    for (int i = 0; i < n; i++) {
        // try all potential starting indices
        bool found = true;
        for (int j = 0; j < m && found; j++)
            // use boolean flag 'found'
            if (i + j >= n || P[j] != T[i + j])
                // if mismatch found
                found = false;
        // abort this, shift starting index i by +1
        if (found)
            // if P[0 .. m - 1] == T[i .. i + m - 1]
            printf("P is found at index %d in T\n", i);
    }
}

void kmpPreprocess() {
    // call this before calling kmpSearch()
    int i = 0, j = -1;
    b[0] = -1; // starting values
    while (i < m) { // pre-process the pattern string P
        while (j >= 0 && P[i] != P[j]) j = b[j];
        // if different, reset j using b
        i++;
        j++; // if same, advance both pointers
        b[i] = j;
    }
    // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4
    // in the example of P = "SEVENTY SEVEN" above

void kmpSearch() {
    // this is similar as kmpPreprocess(), but on string T
    int i = 0, j = 0; // starting values
    while (i < n) { // search through string T
        while (j >= 0 && T[i] != P[j]) j = b[j];
        // if different, reset j using b
        i++;
        j++; // if same, advance both pointers
        if (j == m) { // a match found when j == m
            printf("P is found at index %d in T\n", i - j);
            j = b[j];
        }
        // prepare j for the next possible match
    }
}

int main() {
    strcpy(T, "ababacbababc");
    strcpy(P, "aba");
    n = (int)strlen(T);
    m = (int)strlen(P);

    //if the end of line character is read too, uncomment
    the line below
    //T[n-1] = 0; n--; P[m-1] = 0; m--;

    printf("T = '%s'\n", T);
    printf("P = '%s'\n", P);
    printf("\n");

    clock_t t0 = clock();
    printf("Naive Matching\n");
    naiveMatching();
    clock_t t1 = clock();

```

```

        printf("Runtime = %.10lf s\n\n", (t1 - t0) / (double)
        CLOCKS_PER_SEC);

        printf("KMP\n");
        kmpPreprocess();
        kmpSearch();
        clock_t t2 = clock();
        printf("Runtime = %.10lf s\n\n", (t2 - t1) / (double)
        CLOCKS_PER_SEC);

        printf("String Library\n");
        char *pos = strstr(T, P);
        while (pos != NULL) {
            printf("P is found at index %d in T\n", pos - T);
            pos = strstr(pos + 1, P);
        }
        clock_t t3 = clock();
        printf("Runtime = %.10lf s\n\n", (t3 - t2) / (double)
        CLOCKS_PER_SEC);

        return 0;
    }
}

```

//Suffix Array & Suffix Automata

See on COOKBOOK Palindrome, SUST

//Binomial Coefficient

```

const int maxn = 4050;
long long C[maxn+1][maxn+1];
void func() {
    for (long long n=0; n<=maxn; ++n) {
        C[n][0] = C[n][n] = 1;
        for (long long k=1; k<n; ++k)
            C[n][k] = (C[n-1][k-1] + C[n-1][k]);
    }
    return;
}

```

//Chinese Remainder Theorem (Garner's)

```

//a=x0+x1*p0+x2*p0*p1+x3*p0*p1*p2+...+x(k-
1)*p0*p1*p2*...*p(k-2) (mod p0*p1*p2*...*p(k-1))
void chineseRemainderTheorem(LL x[], LL a[], LL r[][100], LL
p[], LL k) {
    //a=remainder, r[j][i]=p[j]^i (mod p[i]), p=primes(0 based)
    for(LL i=0; i<k; ++i) {
        x[i] = a[i];
        for (LL j = 0; j < i; ++j) {
            x[i] = r[j][i] * (x[i] - x[j])
        }
        x[i] = x[i] % p[i];
    }
    //mod value to avoid overflow
    if (x[i] < 0) x[i] += p[i];
}
}

```

//Relative Prime (Co-Prime)

```

int relPrime(int n) { //relative prime up to n
    int i;
    int ans=n;
    for(i=1; prime[i]*prime[i]<=n; i++)
        if(n%prime[i]==0) {
            while(n%prime[i]==0) n/=prime[i];
            ans/=prime[i];
            ans*=(prime[i]-1);
        }

    if(n>1) {
        ans/=n;
        ans*=(n-1);
    }
    return ans;
}

```

//Prime Generator

```
#define p_z 1000000005 //prime size need
ll prime[p_z/10];
bool is_prime[p_z+10];
ll total=0;
void prime_generate() {
    is_prime[0]=is_prime[1]=1;
    for(ll i=4; i<=p_z; i+=2)
        is_prime[i]=1;
    for(ll i=2; i<=p_z; i++) {
        if(!is_prime[i]) {
            prime[total++]=i;
            for(ll j=i*i; j<=p_z; j+=i)
                is_prime[j]=1;
        }
    }
    return;
}
```