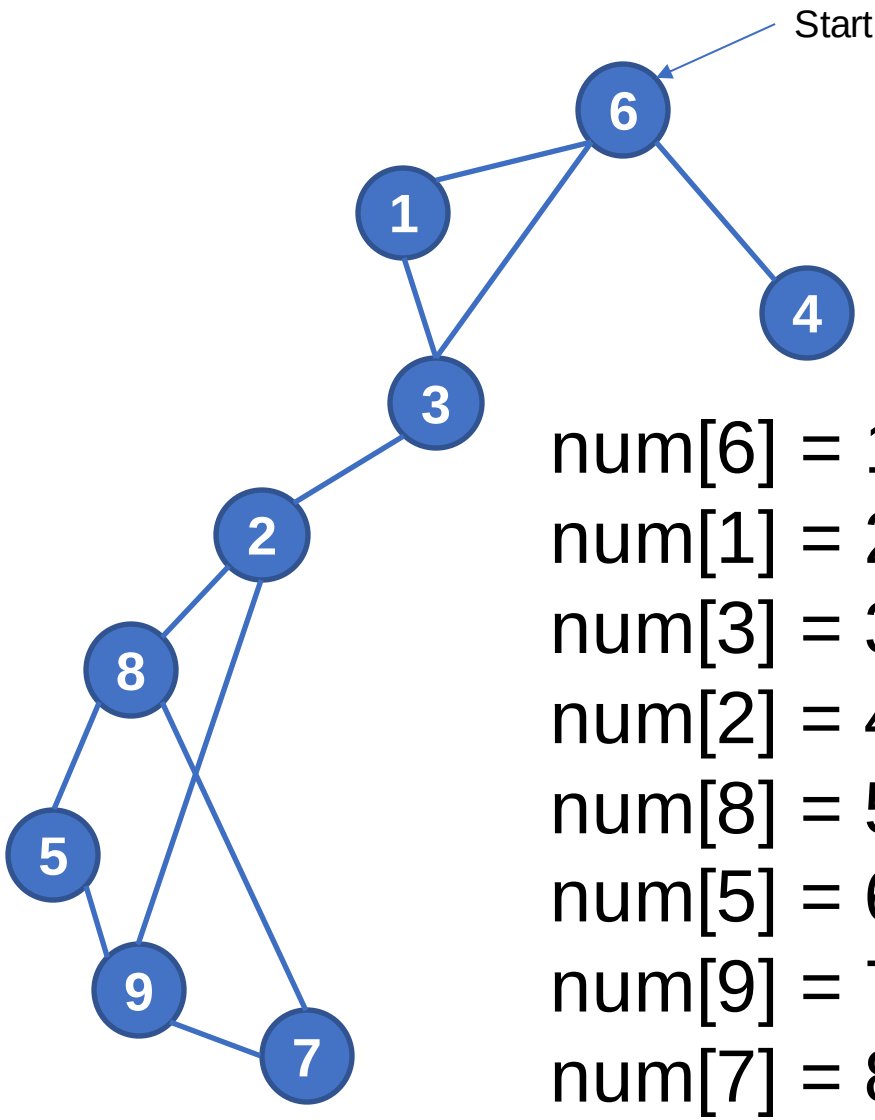
# Bridges

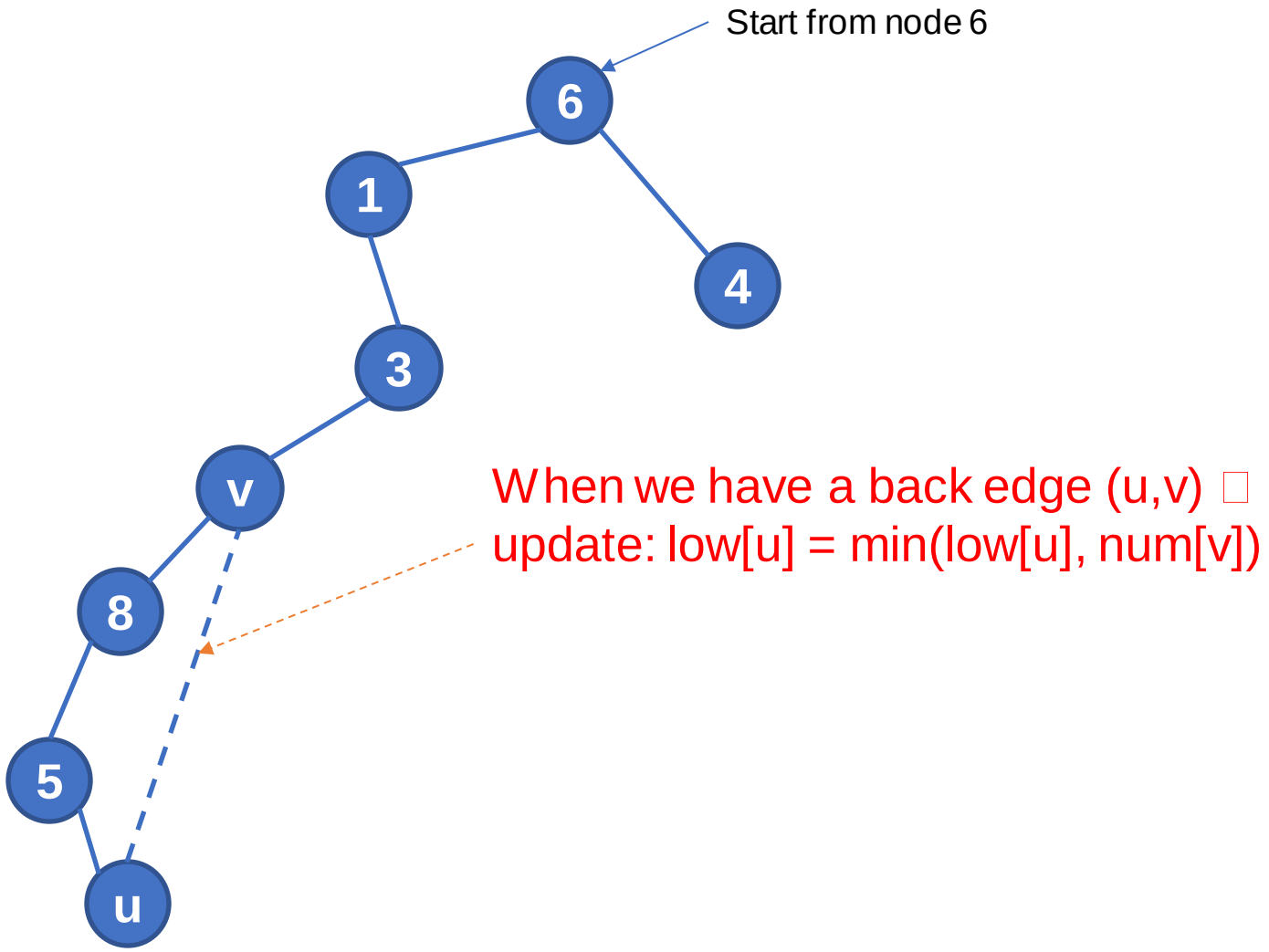
- DFS tree
  - DFS starts from a node u visits a descendants of u on the DFS tree
- Maintain data structures:
  - num[v]: time point node v is visited
  - low[v]: đỉnh con có số thứ tự Num[] thấp nhất khi bắt đầu định chiều từ v




num[6] = 1, low[6] = 1
num[1] = 2, low[1] = 1
num[3] = 3, low[3] = 1
num[2] = 4, low[2] = 4
num[8] = 5, low[8] = 4
num[5] = 6, low[5] = 4
num[9] = 7, low[9] = 4
num[7] = 8, low[7] = 4
num[4] = 9, low[4] = 9

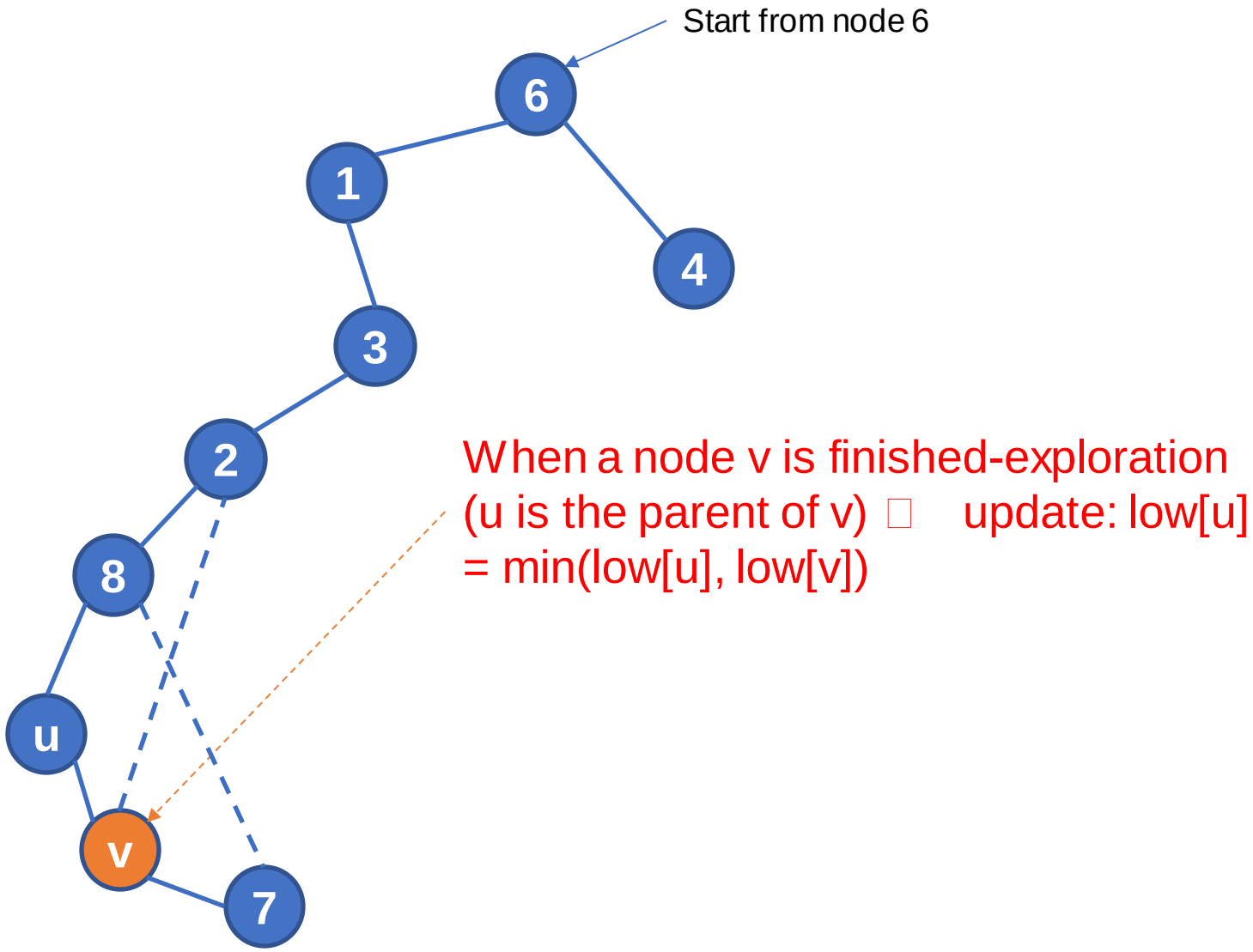
# Bridges

- DFS tree
  - DFS starts from a node u visits a descendants of u on the DFS tree
- Maintain data structures:
  - num[v]: time point node v is visited
  - low[v]: minimal num of some node x such that v is equal to x or there is a back end (u,x) in which u is the node v or some descendant of v



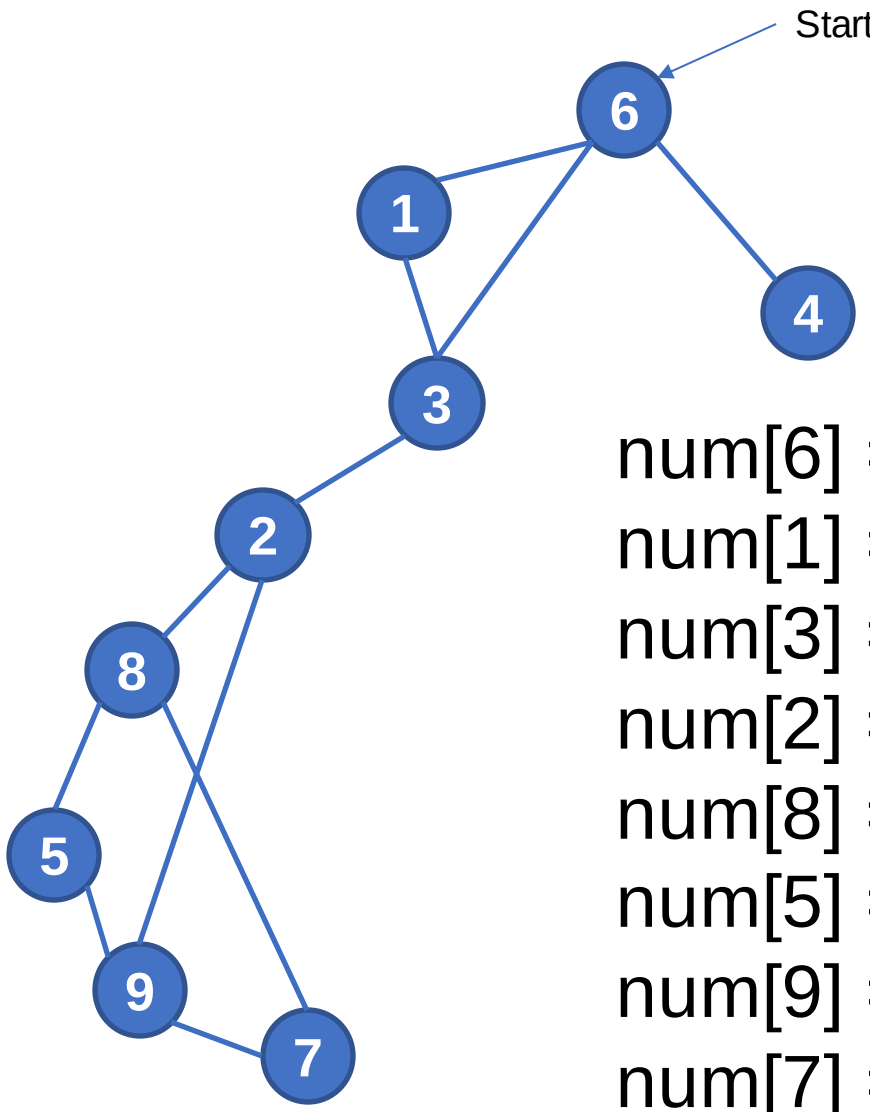
# Bridges

- DFS tree
  - DFS starts from a node u visits a descendants of u on the DFS tree
- Maintain data structures:
  - num[v]: time point node v is visited
  - low[v]: minimal num of some node x such that v is equal to x or there is a back end (u,x) in which u is the node v or some descendant of v



# Bridges

- Bridge:
  - Forward edge (u,v) having low[v] > num[u] is a bridge
- Articulation point:
  - If u is not a root of the DFS tree and forward edge (u,v) having low[v] ≥ num[u] then u is an articulation point
  - If u is the root of a DFS tree, then u is an articulation point if it has more than 1 child




num[6] = 1, low[6] = 1
num[1] = 2, low[1] = 1
num[3] = 3, low[3] = 1
num[2] = 4, low[2] = 4
num[8] = 5, low[8] = 4
num[5] = 6, low[5] = 4
num[9] = 7, low[9] = 4
num[7] = 8, low[7] = 4
num[4] = 9, low[4] = 9

# Bridges

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

using namespace std;

const int N = 1e5 + 2;
int n, m, CriticalEdge;
vector <int> a[N];
bool CriticalNode[N];
int Num[N], Low[N], Time;
//Num[i]: số thứ tự của đỉnh i khi đỉnh chiều đồ thị
//Low[i]: đỉnh con có số thứ tự Num[i] thấp nhất khi bắt đầu đỉnh chiều từ i

void inp()
{
    scanf("%d%d", &n, &m);
    int x, y;
    for(int i = 1; i <= m; i++) {
        scanf("%d%d", &x, &y);
        a[x].push_back(y);
        a[y].push_back(x);
    }
}
```

# Bridges

```
void visit(int u, int p)
{
    int NumChild = 0;
    Low[u] = Num[u] = ++ Time;
    for(int i = 0; i < int(a[u].size()); i++) {
        int v = a[u][i];
        if (v != p) {
            if (Num[v]) Low[u] = min(Low[u], Num[v]);
            else {
                visit(v, u);
                ++ NumChild;
                Low[u] = min(Low[u], Low[v]);

                if (Low[v] >= Num[v]) ++ CriticalEdge;

                if (u == p) {
                    if (NumChild >= 2) CriticalNode[u] = true;
                } else {
                    if (Low[v] >= Num[u]) CriticalNode[u] = true;
                }
            }
        }
    }
}

void proc()
{
    for(int i = 1; i <= n; i++) if (!Num[i]) visit(i, i);
    int Count = 0;
    for(int i = 1; i <= n; i++) if (CriticalNode[i]) ++ Count;
    printf("%d %d\n", Count, CriticalEdge);
}
```

## Inter-City Bus

Algorithm

- Run BFS for computing the cost for traveling with one bus from a city i to another reachable city j → Cost graph G
- Apply the Dijkstra algorithm for finding the shortest path from city 1 to city n in G

# Intercity_bus

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

using namespace std;

typedef pair <int, int> ii;
const int N = 5 * 1e3 + 2;
int n, k, m, dist[N], d[N], c[N];
vector <ii> New[N];
vector <int> Old[N];
priority_queue <ii, vector <ii>, greater <ii> > pq;
int dd[N];

void inp()
{
    int x, y;
    scanf("%d %d", &n, &k);
    for(int i = 1; i <= n; i++) scanf("%d %d", &c[i], &d[i]);
    for(int i = 1; i <= k; i++) scanf("%d %d", &x, &y), Old[x].push_back(y), Old[y].push_back(x);
}
```

# Intercity_bus

```
queue <int> hd;

void bfs(int root)
{
    memset(dd, 0, sizeof(dd));
    int X, v;
    hd.push(root);
    while(!hd.empty()) {
        X = hd.front();
        hd.pop();
        if(dd[X] == d[root]) continue;
        for(int i = 0; i < int(Old[X].size()); i ++) {
            v = Old[X][i];
            if(dd[v] == 0 && v != root) {
                dd[v] = dd[X] + 1;
                New[root].push_back(ii(c[root], v));
                hd.push(v);
            }
        }
    }
}
```

# Intercity_bus

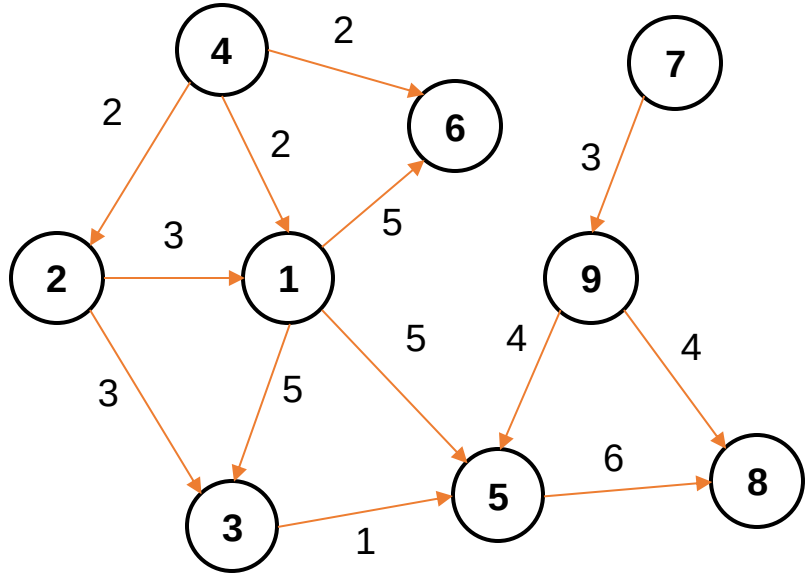
```
void dijkstra()
{
    int u, du, v;
    for(int i = 2; i <= n; i ++) dist[i] = 1e9;
    pq.push(ii(0, 1));
    while(!pq.empty()) {
        u = pq.top().second;
        du = pq.top().first;
        pq.pop();
        if(du != dist[u]) continue;
        for(int i = 0; i < int(New[u].size()); i++) {
            v = New[u][i].second;
            if(dist[v] > dist[u] + New[u][i].first) {
                dist[v] = dist[u] + New[u][i].first;
                pq.push(ii(dist[v], v));
            }
        }
    }
}

void proc()
{
    for(int i = 1; i <= n; i++) bfs(i);
    dijkstra();
    printf("%d\n", dist[n]);
}

int main()
{
    inp();
    proc();
}
```

# Make Span Schedule

- Algorithm
  - L is the TOPO list of nodes of G
  - F[u]: earlest time point the task u can start
  - Explore L from left to right, for each node u:
    - makespan = max(makespan, F[u] + d[u])
    - For each arc (u,v), update F[v] = max(F[v], F[u] + d[u])



# Make Span Schedule

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6;
struct Arc{
    int v;
    int w;
    Arc(int _v, int _w): v(_v), w(_w){}
};
int n,m;
int duration[N];
vector<Arc> A[N];// A[v] set of outgoing arc of v
int d[N];// incoming degree
vector<int> L;
int F[N];// F[v] earliest possible starting time-point
int ans;
```

# Make Span Schedule

```
void input(){
    memset(d,0,sizeof d);
    cin >>n >> m;
    for(int i = 1; i <= n; i++)
        cin >> duration[i];
    for(int k = 1; k <= m; k++){
        int u,v;
        cin >> u >> v;
        A[u].push_back(Arc(v,duration[u]));
        d[v]++;
    }
}
```

# Make Span Schedule

```
void topoSort(){
    queue<int> Q;
    for(int v = 1; v <= n; v++) if(d[v] == 0)
        Q.push(v);
    while(!Q.empty()){
        int x = Q.front(); Q.pop();
        L.push_back(x);
        for(int i = 0; i < A[x].size(); i++){
            int y = A[x][i].v;
            int w = A[x][i].w;
            d[y] -= 1;
            if(d[y] == 0) Q.push(y);
        }
    }
}
```

# Make Span Schedule

```
void solve(){
    memset(F,0,sizeof F);
    ans = 0;
    for(int i = 0; i < L.size(); i++){
        int u = L[i];
        ans = max(ans,F[u] + duration[u]);
        for(int j = 0; j < A[u].size(); j++){
            int v = A[u][j].v;
            int w = A[u][j].w;
            F[v]  = max(F[v],F[u] + w);
        }
    }
    cout << ans << endl;
}
int main(){
    input();
    topoSort();
    solve();
}
```

# Strongly Connected Component

Algorithm

- Run DFS on G → compute the finishing time f(v) of each node v of G
- Build residual graph $G^T$ of G
- Run DFS on $G^T$: the nodes are considered in a decreasing order of f
  - Each run DFS(u) will visit all nodes of the strongly connected component containing u

# Implementation – Strongly Connected Component

```
#include <stdio.h>
#include <bits/stdc++.h>
#include <vector>
#include <iostream>
using namespace std;
#define MAX_N 100001

int n;
vector<int> A[MAX_N];
vector<int> A1[MAX_N];// residual graph

// data structure for DFS
int f[MAX_N];// finishing time
char color[MAX_N];
int t;
int icc[MAX_N];// icc[v] index of the strongly connected component containing v
int ncc;
int x[MAX_N];// sorted-list (decreasing of finishing time) of nodes visited by DFS
int idx;
```

# Implementation – Strongly Connected Component

```
void buildResidualGraph(){// xay dung do thi bu
    for(int u = 1; u <= n; u++){
        for(int j = 0; j < A[u].size(); j++){
            int v = A[u][j];
            A1[v].push_back(u);
        }
    }
}
void init(){
    for(int v = 1; v <= n; v++){
        color[v] = 'W';
    }
    t = 0;
}
```

# Implementation – Strongly Connected Component

```
// DFS on the original graph
void dfsA(int s){
    t++;    color[s] = 'G';
    for(int j = 0; j < A[s].size(); j++){
        int v = A[s][j];
        if(color[v] == 'W'){   dfsA(v);    }
    }
    t++;
    f[s] = t;
    color[s] = 'B';
    idx++;
    x[idx] = s;
}
void dfsA(){
    init();
    idx = 0;
    for(int v = 1; v <= n; v++){
        if(color[v] == 'W'){
            dfsA(v);
        }
    }
}
```

# Implementation – Strongly Connected Component

```
// DFS on the residual graph
void dfsA1(int s){
    t++;    color[s] = 'G';    icc[s] = ncc;
    //for(set<int>::iterator it = A1[s].begin(); it != A1[s].end(); it++){
    for(int j = 0; j < A1[s].size(); j++){
        int v= A1[s][j];
        if(color[v] == 'W'){    dfsA1(v);   }
    }
    color[s] = 'B';
}

void dfsA1(){
    init();
    ncc = 0;
    for(int i = n; i >= 1; i--){
        int v = x[i];
        if(color[v] == 'W'){
            ncc++;
            dfsA1(v);
        }
    }
}
```

# Implementation – Strongly Connected Component

```
void solve(){
    dfsA();
    buildResidualGraph();
    dfsA1();
    cout << ncc;
}
void input(){
    int m;
    cin >> n >> m;
    for(int k = 1; k <= m; k++){
        int u,v;
        cin >> u >> v;
        A[u].push_back(v);
    }
}

int main(){
    input();
    solve();
}
```