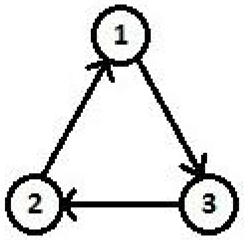
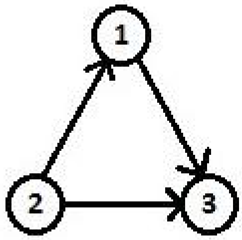
স্ট্রংলি কানেক্টেড কম্পোনেন্ট (Kosaraju’s Algorithm)

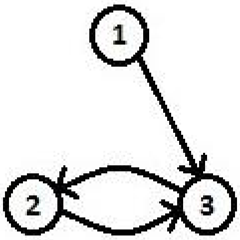
একটি ডাইরেক্টেড গ্রাফ কে স্ট্রংলি কানেক্টেড বলা হয় যখন তার প্রত্যেকটি নোড থেকে ঐ গ্রাফের অন্যান্য প্রত্যেকটি নোডে যাওয়ার জন্য পাথ থাকে। আর যখন কোন একটি গ্রাফের কোন সাব-গ্রাফ স্ট্রংলি কানেক্টেড থাকে তখন তাদেরকে স্ট্রংলি কানেক্টেড কম্পোনেন্ট বা সংক্ষেপে SCC বলা হয়। এই SCC গুলো গ্রাফে একটি ভার্চুয়াল পার্টিশন তৈরি করে যাতে তাদের কে মুল গ্রাফ থেকে পৃথক করা যায়।



(A)



(B)



(C)

উপরের গ্রাফ গুলো খেয়াল করে দেখো তাহলে বিষয় টি ভালো ভাবে বুঝতে পারবে। আমাদের প্রথম গ্রাফে SCC আছে ১ টি {(1, 2, 3)}। কিন্তু দ্বিতীয় গ্রাফে SCC আছে ৩ টি {(1), (2), (3)}। এবং তৃতীয় গ্রাফে মোট SCC আছে ২ টি {(1), (2, 3)}।

গ্রাফের প্রত্যেকটি নোড কোন না কোন SCC এর অন্তর্ভুক্ত থাকে এবং একটি নোড কখনো একাধিক SCC তে থাকতে পারে না। গ্রাফের যেকোনো দুটি নোড যদি স্ট্রংলি কানেক্টেড থাকে, এবং তাদের যেকোনো একটি নোডের সাথে যদি অপর কোন তৃতীয় নোড স্ট্রংলি কানেক্টেড থাকে তাহলে সেই তিনটি নোড একই SCC এর অন্তর্ভুক্ত হবে। কারণ তখন তাদের প্রত্যেকটি নোড থেকে অন্য সব গুলো নোডে যাওয়ার জন্য পাথ থাকবে।

ডাইরেক্টেড গ্রাফে স্ট্রংলি কানেক্টেড কম্পোনেন্ট বের করার জন্য Kosaraju’s Algorithm খুবই সহজ এবং জনপ্রিয় একটি অ্যালগরিদম। এটি শেখার জন্য শুধু DFS জানা থাকলেই হবে। তবে অ্যালগরিদমে যাওয়ার আগে কিছু প্রয়োজনীয় জিনিস আগে বলে নেওয়া উচিত। তাহলে অ্যালগরিদম টা বুঝতে সুবিধা হবে।

DFS স্টার্টিং টাইম এবং ফিনিশিং টাইমঃ কোন গ্রাফে যখন DFS চালানো হয়, তখন তা সবসময় এক বা একাধিক ট্রি তৈরি করে। এই ট্রি গুলো কে depth-first-tree, এবং অনেক গুলো ট্রি কে একত্রে depth-first-forest বলা হয়। এই ট্রি গুলোর কোন একটি নোড কে যখন ভিজিট করা হয়, তখন সেটা হয় তার স্টার্টিং টাইম। সেই নোডের সব গুলো চাইল্ড কে ভিজিট করে আবার ঐ নোডে আসতে যে টাইম (স্টেপ) লাগে, সেটা হলো ঐ নোডের ফিনিশিং টাইম।

কোন গ্রাফের নোড গুলোর স্টার্টিং এবং ফিনিশিং টাইম বের করতে চাইলে প্রথমে একটি ইন্টিজার ভেরিয়েবল ধরে নিতে হবে, মনেকরি তার নাম clock, এর মান প্রথমেই 0 ধরে নিতে হবে। এবার সব গুলো নোডের স্টার্টিং আর ফিনিশিং টাইম রাখার জন্য ২টি আলাদা অ্যারে নিতে হবে, মনে করি তাদের নাম start এবং finish । এখন কাজ হলো, কোন নোড থেকে DFS চালানোর সময় start[node] এ clock এর মান রাখতে হবে, এবং ঐ নোডের সব গুলো অ্যাডজাসেন্ট নোড ভিজিট হয়ে গেলে finish[node] এ আবার clock এর মান রাখতে হবে। সকল ক্ষেত্রে clock এর মান স্টোর করার পর তার মান ১ বাড়িয়ে দিতে হবে। DFS শেষ হলে আমরা আমাদের গ্রাফের সব গুলো নোডের স্টার্টিং এবং ফিনিশিং টাইম পেয়ে যাবো। তবে SCC বের করার জন্য আমাদের এতো ঝামেলার দরকার হবে না ☺ ।

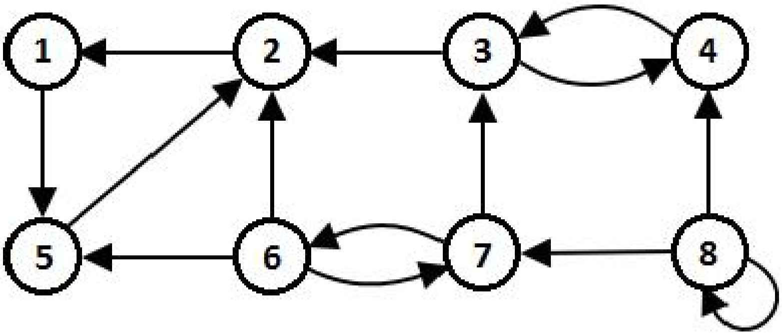
ট্রান্সপোজ/ রিভার্স গ্রাফঃ কোন একটি ডাইরেক্টেড গ্রাফ এর সব গুলো এজ্ কে উল্টো করে দেয়া হলে যে গ্রাফ পাওয়া যাবে তাকে ঐ গ্রাফের ট্রান্সপোজ বা রিভার্স গ্রাফ বলে। কোড ইমপ্লিমেন্টেশনের ক্ষেত্রে, মুল গ্রাফ স্টোর করার সময় এজ্ গুলো উল্টো করে স্টোর করলেই রিভার্স গ্রাফ পাওয়া যাবে।

এখন তাহলে দেখা যাক Kosaraju’s Algorithm ব্যাবহার করে আমরা কিভাবে SCC বের করতে পারি। এজন্য আমাদেরকে মূলত দুটি কাজ করতে হবে।

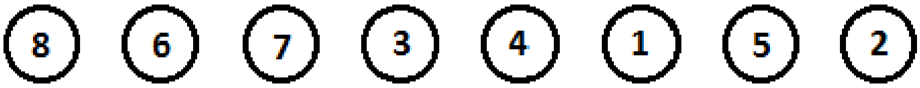
১। মুল গ্রাফে একবার DFS চালিয়ে সব গুলো নোড কে তাদের ফিনিশিং টাইম অনুযায়ী ডিসেন্ডিং অর্ডারে সাজিয়ে একটি লিস্ট তৈরি করতে হবে।

২। লিস্টের শুরু থেকে রিভার্স গ্রাফে DFS চালাতে হবে। প্রতিটি নন-ভিজিটেড নোড থেকে DFS কল করতে হবে এবং প্রতিবার DFS চলার সময় যে সব নোড গুলো ভিজিট হবে তাদের কে আলাদা করে সেট হিসেবে রাখতে হবে, কারণ তারা সবাই একটি SCC এর অন্তর্ভুক্ত হবে।

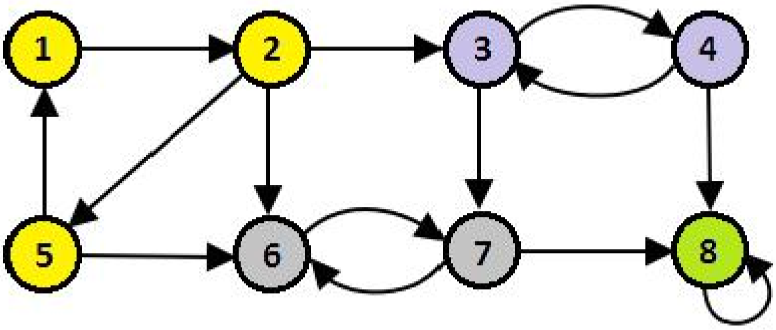
এখন অনেকের মনেই প্রশ্ন আসতে পারে যে এটা কিভাবে কাজ করে? চলো তাহলে নিচের গ্রাফ টায় অ্যালগরিদম অ্যাপ্লাই করে দেখা যাক।



এই গ্রাফ টি তে DFS চালিয়ে নোড গুলো কে তদের ফিনিশিং টাইম অনুযায়ী সাজালে নিচের মত একটি লিস্ট পাওয়া যাবে। এই লিস্ট টা একই গ্রাফে ভিন্ন রকম হতে পারে, সেটা নির্ভর করে গ্রাফ ইনপুট নেওয়ার সময় তাদের এজ্ গুলোর অর্ডারের উপর। তবে সামান্য পার্থক্য হলেও তা আমাদের কোন সমস্যার সৃষ্টি করবে না।



এখন এই লিস্টের শুরু থেকে রিভার্স গ্রাফে DFS চালানো শুরু করতে হবে। লিস্টে কোন নন–ভিজিটেড নোড পেলেই সেটা থেকে DFS কল হবে। আর ঐ সময় যে সব নোড গুলো ভিজিট হবে, তাদের কে আলাদা করে রাখতে হবে। এই নোডের সেট গুলোই হবে এক এক টি স্ট্রংলি কানেক্টেড কম্পোনেন্ট। নিচের ছবিতে রিভার্স গ্রাফে SCC গুলো আলাদা করে দেয়া আছে। এখানে গ্রাফের স্ট্রংলি কানেক্টেড নোড গুলো কে আলাদা আলাদা রং দিয়ে চিহ্নিত করা হয়েছে।



আমাদের এই লিস্টের কোন নন-ভিজিটেড নোড থেকে যখন রিভার্স গ্রাফে DFS চালানো হবে, তখন ঐ নোড থেকে যে সব নোডে যাওয়া যাবে, তাদের সব গুলো নোড একই SCC এর অন্তর্ভুক্ত হবে। গ্রাফে যদি এমন কোন নোড থাকে, যারা অ্যাডজাসেন্ট কিন্তু স্ট্রংলি কানেক্টেড নয় অর্থাৎ তারা ভিন্ন SCC তে, তাহলেও আমাদের কোন সমস্যা হবে না। কারণ ঐ নোড টি অবশ্যই আগে থেকে অন্য কোন SCC এর অন্তর্ভুক্ত হয়ে যাবে। অর্থাৎ সেটা আগে থেকেই ভিজিটেড থাকবে বলে DFS কল করার সময় ঐ নোড টি কখনো দ্বিতীয় বার ভিজিট হবে না। এখন খাতা কলম নিয়ে স্টেপ গুলো হাতে কলমে করে দেখো, তাহলে বিষয় টি আরো ভাল ভাবে বুঝতে পারবে।

এখন আসি কোড ইমপ্লিমেন্টেশনে। গ্রাফ এবং রিভার্স গ্রাফ স্টোর করার জন্য আমাদের দুটি 2D ভেক্টরের প্রয়োজন হবে আর নোড গুলো কে তাদের ফিনিশিং টাইম অনুযায়ী সাজিয়ে একটি লিস্ট তৈরি করতে আরেকটি ভেক্টর লাগবে।

নোড গুলো কে সাজানোর জন্য আমাদের খুব একটা কষ্ট করতে হবে না। কোন নোড থেকে DFS কল হবার পর তার অ্যাডজাসেন্ট নোড গুলোর কাজ শেষ হলে আমাদের লিস্টের শেষে ঐ নোড টিকে পুশ করে দিলেই কাজ হয়ে যাবে। DFS শেষে আমাদের লিস্টে সব গুলো নোড থাকবে এবং তারা তাদের ফিনিশিং টাইম অনুযায়ী অ্যাসেন্ডিং অর্ডারে সাজানো থাকবে। কিন্তু আমাদের যেহেতু ডিসেন্ডিং অর্ডারে সাজানো প্রয়োজন, তাই লিস্ট টা ব্যাবহারের সময় উল্টো দিক থেকে ব্যাবহার করলেই হবে।

```
dfs1(source)
    source.visited = true
    for i = 0 to i < G[source].size
        if G[source][i] is not visited
            dfs1(G[source][i])
    push source to the list
```

আমাদের অর্ধেক কাজ শেষ। এখন আমাদের কাজ হচ্ছে এই লিস্ট থেকে রিভার্স গ্রাফে DFS চালানো এবং SCC গুলো কে আলাদা করে স্টোর করে রাখা। সব গুলো SCC কে স্টোর করে রাখার জন্য আমরা একটি 2D ভেক্টর ব্যাবহার করতে পারি এবং প্রতিবার DFS কলের সময় যে নোড গুলো ভিজিট হবে তাদের লিস্ট তৈরি করার জন্য আরেকটি ভেক্টর ব্যাবহার করতে পারি। DFS কলের আগে এই লিস্ট টা ক্লিয়ার করে নিতে হবে এবং শেষে লিস্ট টা কে স্টোর করে রাখতে হবে।

```
dfs2(source)
    source.visited = true
    push source to the list
    for i = 0 to i < R[source].size
        if R[source][i] is not visited
            dfs2(R[source][i])
```

আমাদের কাজ শেষ। এখন আমাদের গ্রাফে কয়টা SCC আছে এবং গ্রাফের কোন কোন নোড কোন SCC এর অন্তর্ভুক্ত তা পেয়ে যাবো। সম্পুর্ন ইমপ্লিমেন্টশনের পর কোড টা অনেকটা এরকম হবে-

```
#include <bits/stdc++.h>
using namespace std;

#define CLR(a) memset(a, 0, sizeof(a))
#define MAX 1005

vector<int> G[MAX],R[MAX],SCC[MAX],myList,nodes;
bool visited[MAX];

void dfs1(int u)
{
    visited[u] = 1;
    int l = G[u].size();
    for(int i = 0; i < l; i++) {
        int v = G[u][i];
        if(!visited[v]) dfs1(v);
    }
    myList.push_back(u);
}

void dfs2(int u)
{
    visited[u] = 1;
    nodes.push_back(u);
    int l = R[u].size();
    for(int i = 0; i < l; i++) {
        int v = R[u][i];
        if(!visited[v]) dfs2(v);
    }
}

int main()
{
    int i, j, n, m, a, b, totalSCC;

    scanf("%d %d", &n, &m);
```

```cpp

    for(i = 1; i <= m; i++) {
        scanf("%d %d", &a, &b);
        G[a].push_back(b);
        R[b].push_back(a);
    }

    CLR(visited); myList.clear();
    for(i = 1; i <= n; i++) {
        if(!visited[i]) dfs1(i);
    }

    CLR(visited); totalSCC = 0;
    for(i = n-1; i >= 0; i--) {
        if(!visited[myList[i]]) {
            nodes.clear();
            dfs2(myList[i]);
            SCC[totalSCC] = nodes;
            totalSCC++;
        }
    }

    printf("%d\n", totalSCC);
    for(i = 0; i < totalSCC; i++) {
        for(j = 0; j < SCC[i].size(); j++) {
            printf("%d ", SCC[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

কমপ্লেক্সিটিঃ Kosaraju’s Algorithm এ আমাদের মূলত দুই বার DFS চালাতে হয়েছে। তাই এর কমপ্লেক্সিটি হবে দুই বার DFS চালানোর কমপ্লেক্সিটির সমান, অর্থাৎ O(2*(V+E)) । এখানে V হচ্ছে গ্রাফের মোট ভার্টেক্স সংখ্যা এবং E গ্রাফের এজ্ এর সংখ্যা।

এখন অ্যালগরিদম টা নিজে নিজে ইমপ্লিমেন্ট করার চেষ্টা করো। আর নিচের প্রবলেম টা ঝটপট সমাধান করে ফেলো।

- Lignt OJ 1003 – Drunk

হ্যাপি কোডিং ☺