# Topological sort & SCC

2020 Winter 중급

20141284 이기현

## 도입

- 자료구조를 듣기 전에 C언어를 들어야 한다.
- 수영을 하기 전에는 준비 운동을 해야 한다.
- 자기 전에는 양치를 해야한다.

# 위상 정렬

- 선/후수 관계를 갖는 그래프를 정렬하기 위한 방법

# 위상 정렬

- 선/후수 관계를 갖는 그래프를 정렬하기 위한 방법
- 선/후수 관계 → 방향성 존재 → directed graph
- 두 노드가 서로 우선해야 한다면 → 아무것도 X → cycle 없어야 함 → acyclic graph

# 위상 정렬

- 선/후수 관계를 갖는 그래프를 정렬하기 위한 방법
- 선/후수 관계 → 방향성 존재 → directed graph
- 두 노드가 서로 우선해야 한다면 → 아무것도 X → cycle 없어야 함 → acyclic graph

⇒ DAG (directed acyclic graph)

(cycle 존재 유무를 판별하기 위해 쓸 수도 있겠다)

# Graph 용어 정리

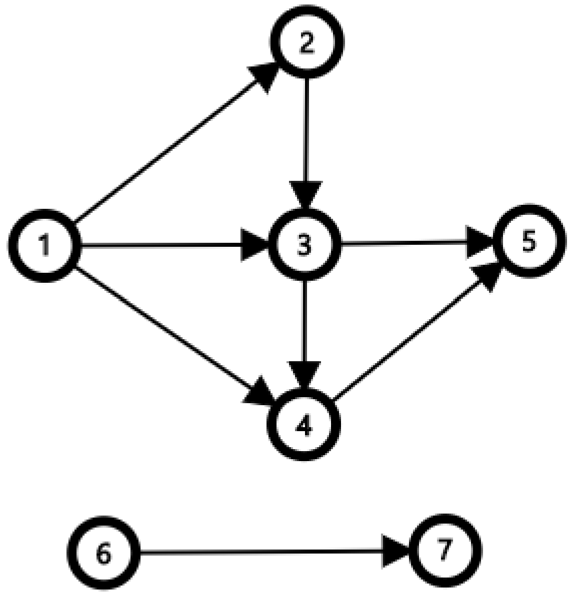
- degree (차수) : 한 노드에 연결된 edge의 수
- outdegree (출력 차수) : directed graph에서 노드에서 나가는 edge의 수
- indegree (입력 차수) : directed graph에서 노드로 들어오는 edge의 수

# Topological sort with indegree

- 수행 가능한 작업
  = 선수 작업이 모두 해결된 상태
  = indegree가 0인 노드

- 수행 가능한 작업을 하나씩 수행
- 작업을 수행하고 다음 작업이 더 이상 선수 작업이 필요 없다면 작업 큐에 넣어주자

# Topological sort with indegree




## indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 1 | 2 | 2 | 2 | 0 | 1 |

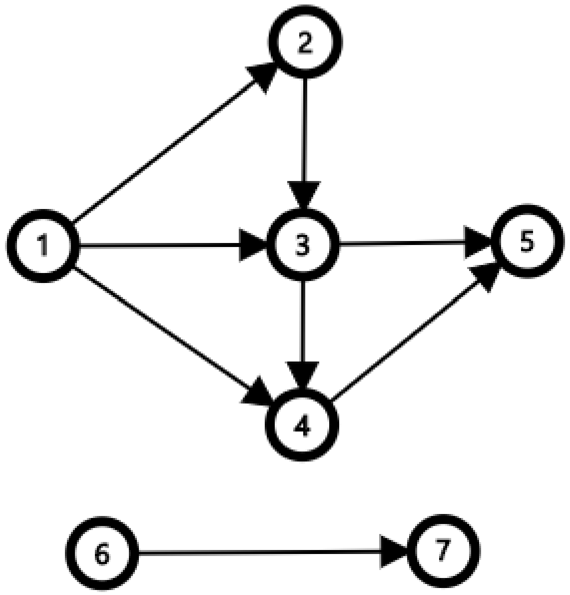
## 작업 queue

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | | | | | | | |

# Topological sort with indegree




indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 1 | 2 | 2 | 2 | 0 | 1 |

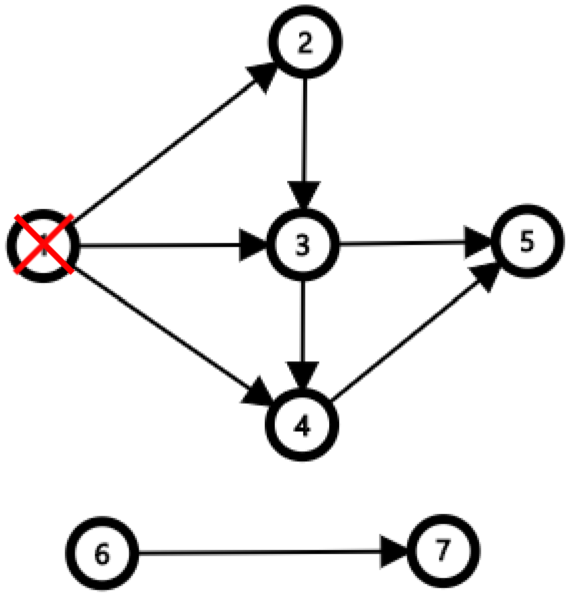
작업 queue

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 6 | | | | | | | |

작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | | | | | | | |

# Topological sort with indegree




indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 0 | 1 | 1 | 2 | 0 | 1 |

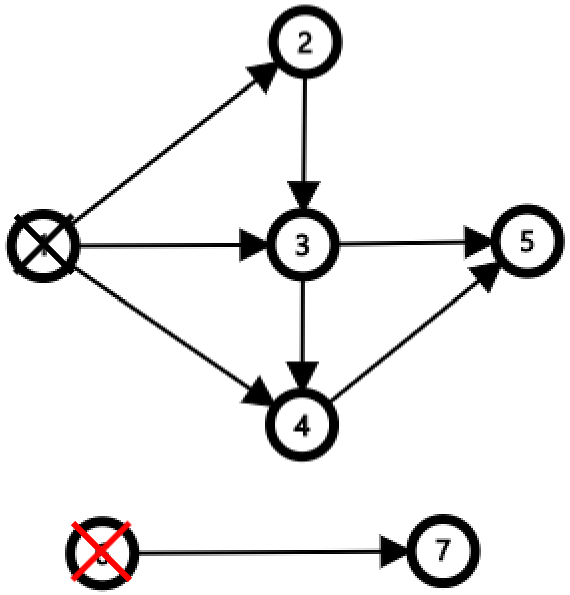
작업 queue

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ~~1~~ | 6 | 2 | | | | | | |

작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 1 | | | | | | |

# Topological sort with indegree




## indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 0 | 1 | 1 | 2 | 0 | 0 |

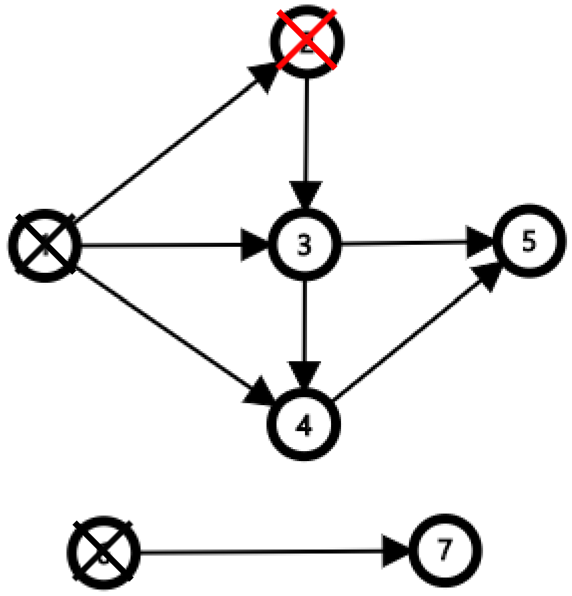
## 작업 queue

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~6~~ | 2 | 7 | | | | | |

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 1 | | | | | 2 | |

# Topological sort with indegree




indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 0 | 0 | 1 | 2 | 0 | 0 |

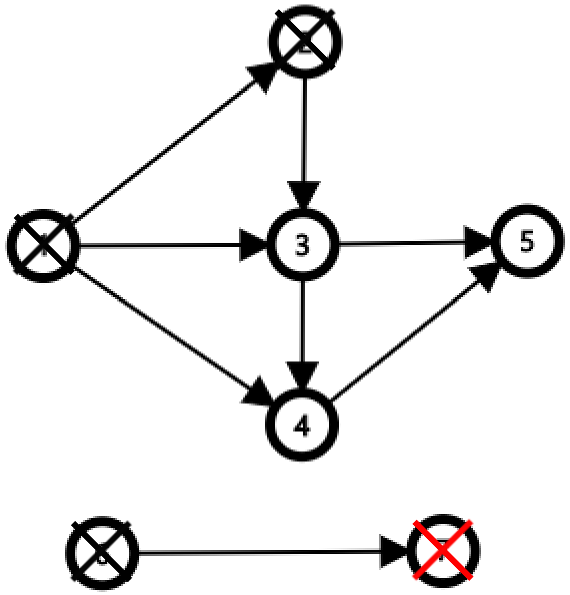
작업 queue

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~6~~ | ~~2~~ | 7 | 3 | | | | |

작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 1 | 3 | | | | 2 | |

# Topological sort with indegree




## indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 0 | 0 | 1 | 2 | 0 | 0 |

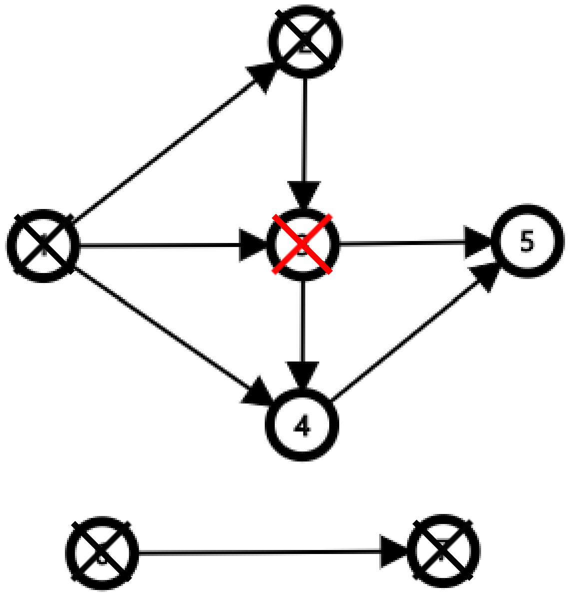
## 작업 queue

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~6~~ | ~~2~~ | ~~7~~ | 3 | | | | |

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 1 | 3 | | | | 2 | 4 |

# Topological sort with indegree




## indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

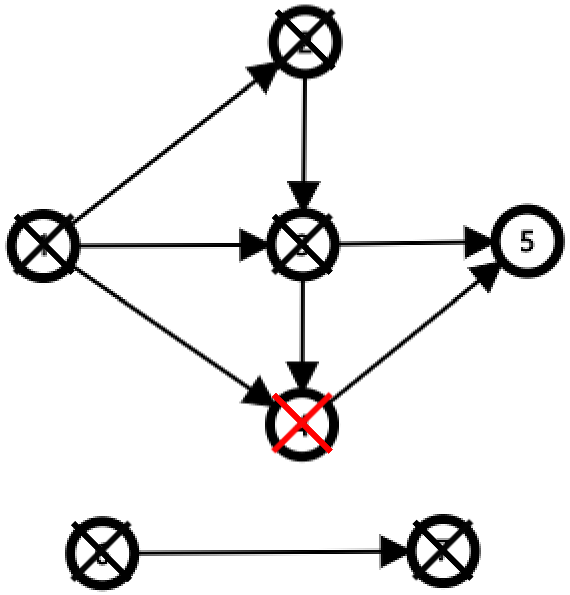
## 작업 queue

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~6~~ | ~~2~~ | ~~7~~ | ~~3~~ | 4 | | | |

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 1 | 3 | 5 | | | 2 | 4 |

# Topological sort with indegree




## indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

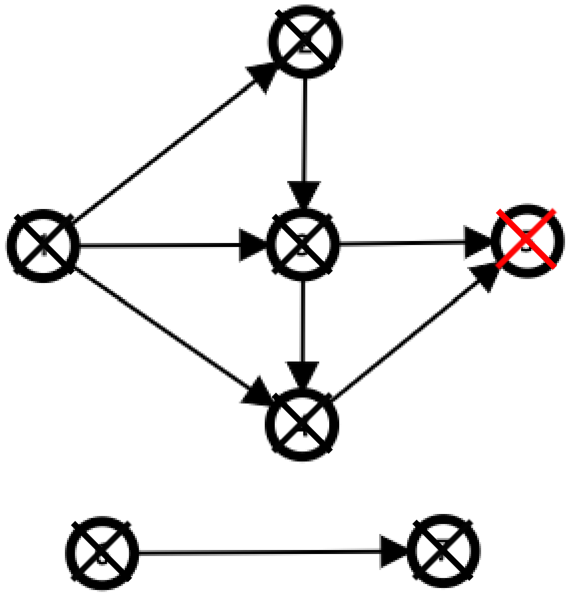
## 작업 queue

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~6~~ | ~~2~~ | ~~7~~ | ~~3~~ | ~~4~~ | 5 | | |

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 1 | 3 | 5 | 6 | | 2 | 4 |

# Topological sort with indegree



## indegree table

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| indegree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## 작업 queue

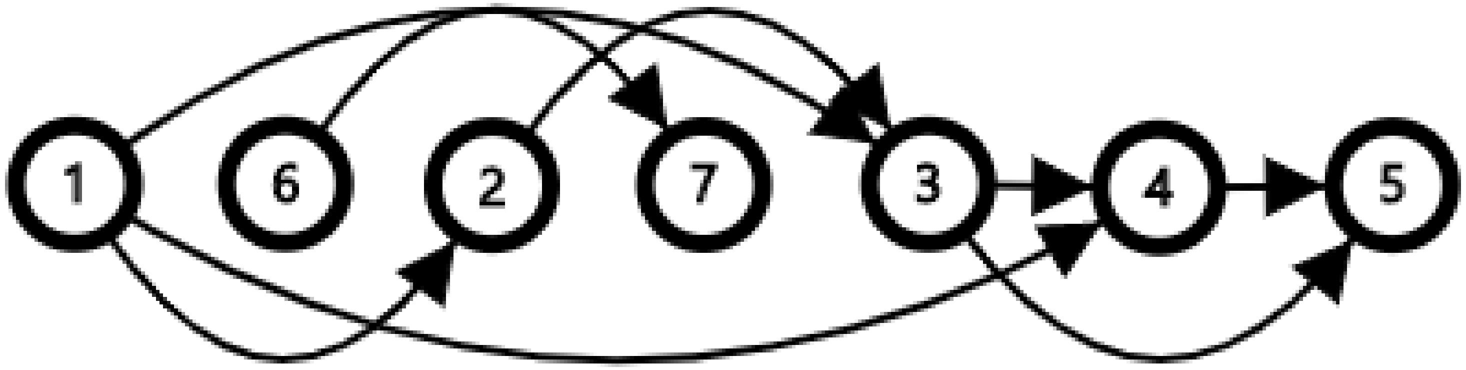
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~6~~ | ~~2~~ | ~~7~~ | ~~3~~ | ~~4~~ | ~~5~~ | | |

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 1 | 3 | 5 | 6 | 7 | 2 | 4 |

# Topological sort with indegree

- 정렬 결과




- edge의 방향이 오른쪽으로만 향한다.

# Topological sort with indegree

- 16 ~ 22 line

  인접리스트 생성

  indegree 계산

- 25 ~ 28 line

  indegree가 0인 노드들로

  queue 초기화

```cpp
16 vector<vector<int> > adj(n + 1);
17 vector<int> indegree(n + 1);
18 for (int i = 0; i < m; ++i) {
19     int a, b;
20     cin >> a >> b;
21     adj[a].push_back(b);
22     indegree[b]++;
23 }
24 
25 queue<int> Q;
26 for (int i = 1; i <= n; ++i)
27     if (!indegree[i])
28         Q.push(i);
```

# Topological sort with indegree

- 34 line

  현재 노드에서 나가는

  edge 지우기

- 35 line

  indegree가 0인 노드

  queue에 넣기

```
30  while (!Q.empty()) {
31      int cur = Q.front(); Q.pop();
32      cout << cur << ' ';
33      for (int next : adj[cur]) {
34          indegree[next]--;
35          if (!indegree[next]) Q.push(next);
36      }
37  }
```

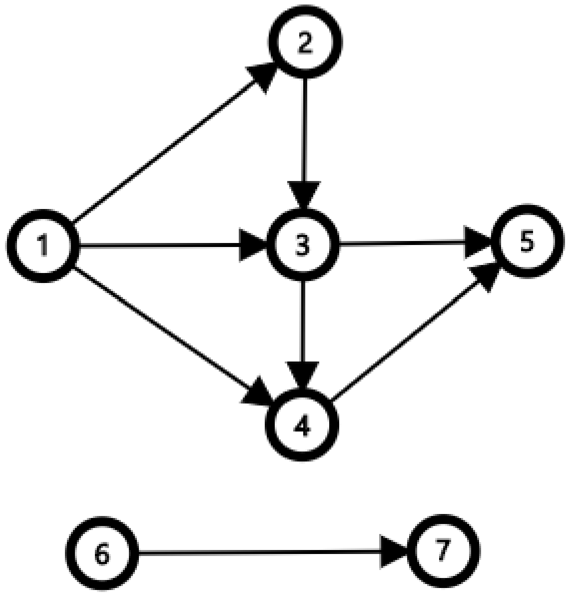
# Topological sort with DFS

- DFS (깊이 우선 탐색)

  = 가장 깊은 곳으로 내려가는 탐색 방법

- 가장 늦게 해야 하는 작업을 판단 가능
- 탐색이 먼저 끝날 수록 늦게 해야 하는 작업

# Topological sort with DFS



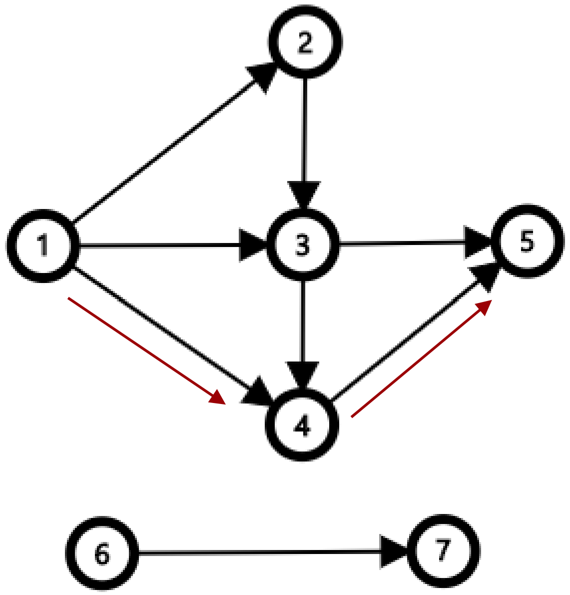

DFS 호출

dfs(1)

작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | | | | | | | |

# Topological sort with DFS



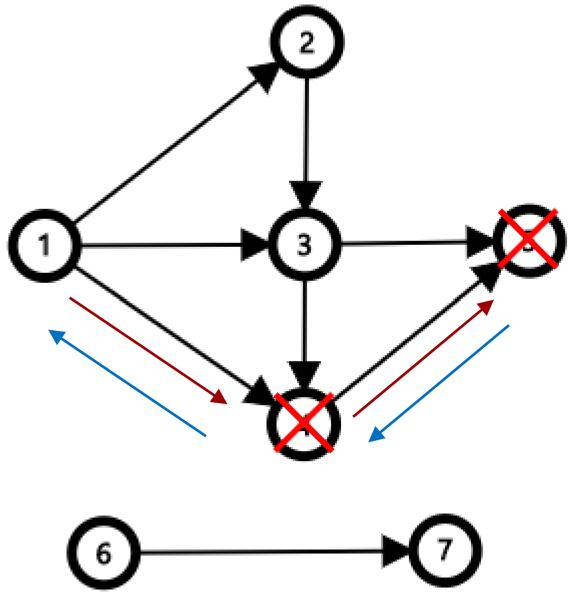

DFS 호출

dfs(1) → dfs(4) → dfs(5)

작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | | | | | | | |

# Topological sort with DFS



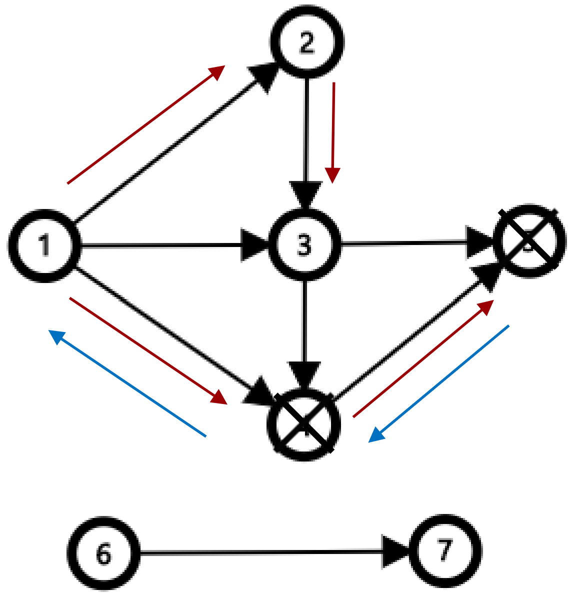

## DFS 호출

dfs(1) → ~~dfs(4)~~ → ~~dfs(5)~~

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | | | | 6 | 7 | | |

# Topological sort with DFS




DFS 호출

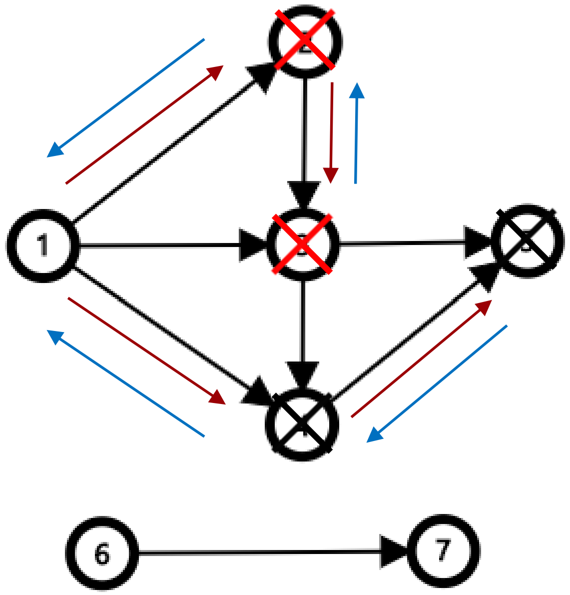
dfs(1) → ~~dfs(4)~~ → ~~dfs(5)~~

→ dfs(2) → dfs(3)

작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | | | | 6 | 7 | | |

# Topological sort with DFS



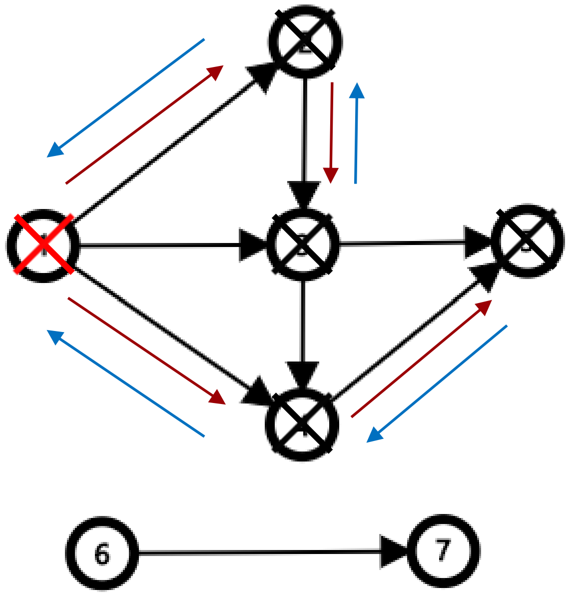

## DFS 호출

dfs(1) → ~~dfs(4)~~ → ~~dfs(5)~~
→ ~~dfs(2)~~ → ~~dfs(3)~~

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order |  | 4 | 5 | 6 | 7 |  |  |

# Topological sort with DFS




## DFS 호출

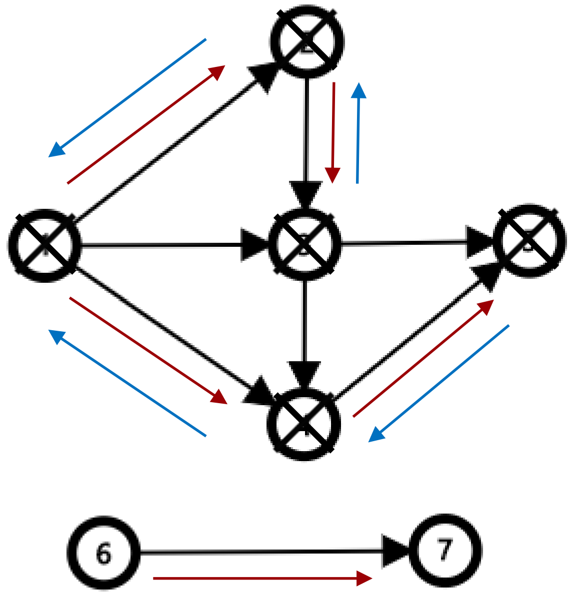
~~dfs(1)~~ → ~~dfs(4)~~ → ~~dfs(5)~~

→ ~~dfs(2)~~ → ~~dfs(3)~~

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 3 | 4 | 5 | 6 | 7 | | |

# Topological sort with DFS




## DFS 호출

~~dfs(1)~~ → ~~dfs(4)~~ → ~~dfs(5)~~

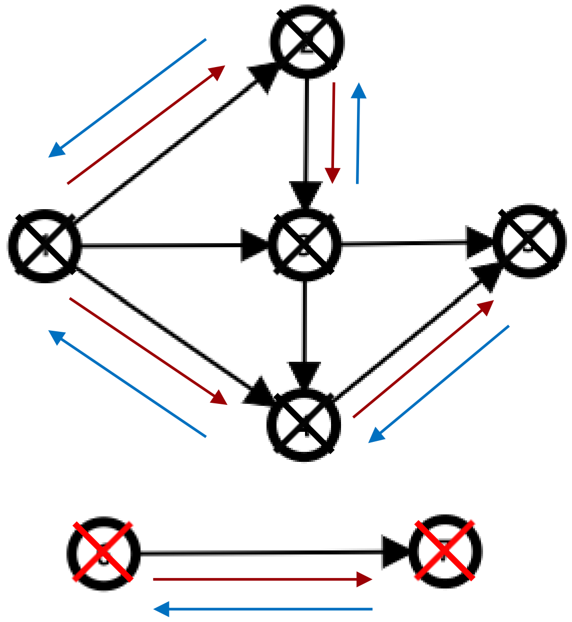
→ ~~dfs(2)~~ → ~~dfs(3)~~

dfs(6) → dfs(7)

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 3 | 4 | 5 | 6 | 7 | | |

# Topological sort with DFS



## DFS 호출

~~dfs(1)~~ → ~~dfs(4)~~ → ~~dfs(5)~~

→ ~~dfs(2)~~ → ~~dfs(3)~~

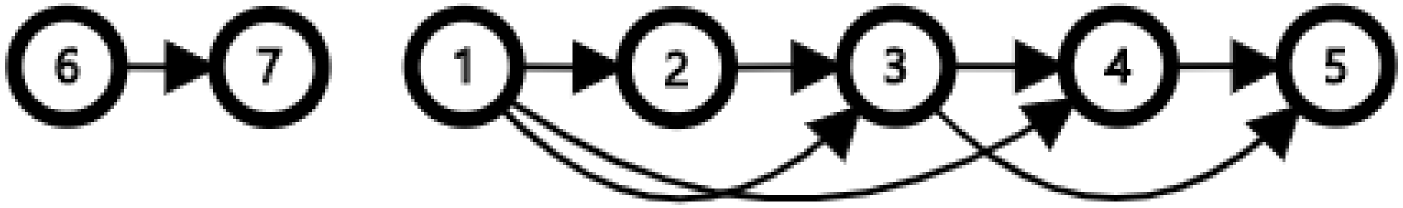
~~dfs(6)~~ → ~~dfs(7)~~

## 작업 순서

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| order | 3 | 4 | 5 | 6 | 7 | 1 | 2 |

# Topological sort with DFS

- 정렬 결과



# Topological sort with DFS

- 60 ~ 62 line

  DFS 실행

- 49 line

  DFS가 끝나는 순서대로 스택 넣기

- 64 ~ 65 line

  스택의 top부터 뽑아내기

```
44 void dfs(int cur) {
45     vis[cur] = true;
46     for (int next : adj[cur])
47         if (!vis[next])
48             dfs(next);
49     ST.push_back(cur);
50 }
51 int main() {
52     int n, m;
53     cin >> n >> m;
54     for (int i = 0; i < m; ++i){ ... }
59
60     for (int i = 1; i <= n; ++i)
61         if (!vis[i])
62             dfs(i);
63
64     for (int i = n - 1; i >= 0; --i)
65         cout << ST[i] << ' ';
66 }
```

# Topological sort

시간 복잡도 : $O(V + E)$

- 위상 정렬의 결과는 한가지가 아닐 수도 있다.
- 선/후수 이외의 우선순위가 있을 때, 우선순위 큐를 이용해도 가능
  (애초에 큐 말고도 스택을 써도 상관은 없다 – 본질은 할 수 있는 작업을 먼저 하는 것)
- 작업 큐가 비었는데 정렬이 안된 노드가 있다
  = cycle 이 있다. = DAG가 아니다.

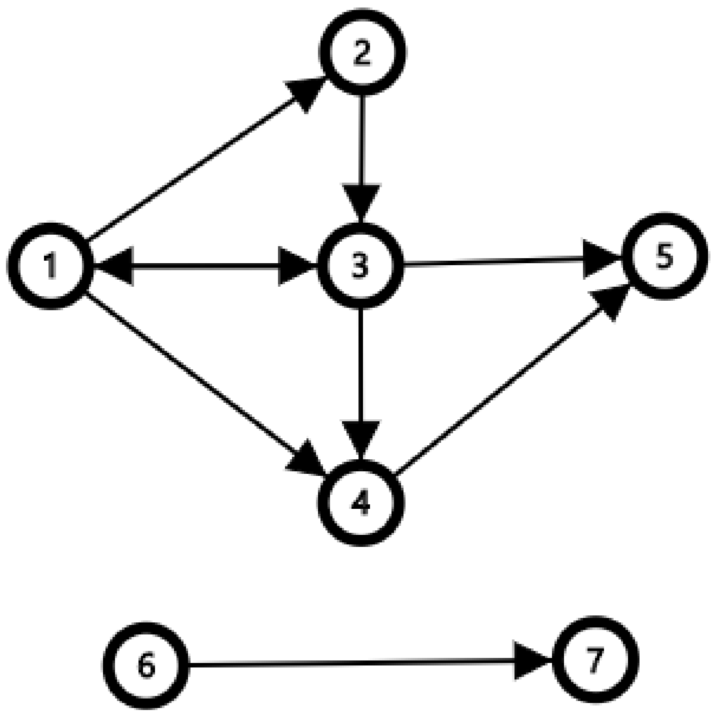
# #1766 문제집 2

1번부터 N번까지의 문제를 푸려고 한다. 1번부터 N번으로 갈수록 난이도는 높아진다.
다음 규칙에 따라 문제의 풀 순서를 결정하여라.

1) N개의 문제는 모두 풀어야 한다.
2) 먼저 푸는게 좋은 문제가 있는 문제는, 먼저 푸는게 좋은 문제를 반드시 먼저 풀어야 한다.
3) 가능하면 쉬운 문제부터 풀어야 한다.

# #1766 문제집

- 대놓고 위상 정렬 하는 문제
- 풀 수 있는 문제가 1개 이상일 때, 쉬운 문제부터 해야 한다.
- 작업 큐로 우선순위 큐를 이용하여 번호가 낮은 문제가 top에 오도록 한다.




정렬하고 싶다

# Strongly Connected Component (SCC, 강한 결합 요소)

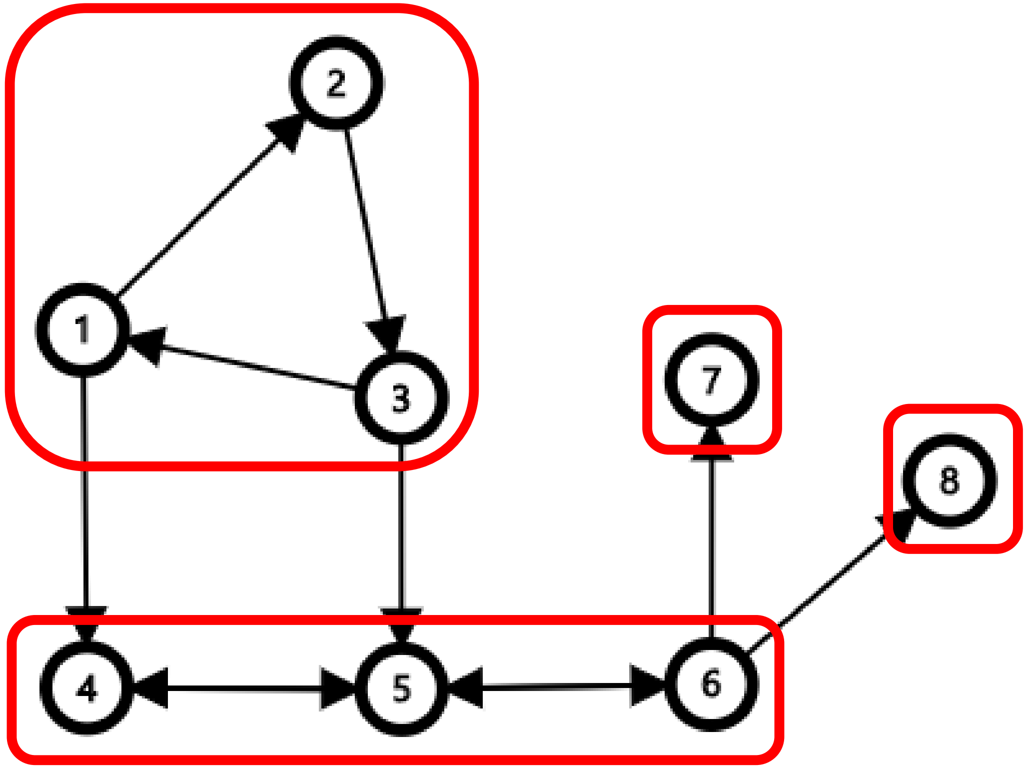
- 한 정점그룹 내에서 임의의 두 정점을 뽑아 u, v 라고 하자.
- u에서 v로 가는 경로가 존재할 때, v에서 u로 가는 또다른 경로가 존재한다.

- 이 때, 이 정점그룹을 SCC라고 부른다.

# Strongly Connected Component (SCC, 강한 결합 요소)

- 같은 SCC 내의 임의의 점 u, v에 대해 u에서 v로 가는 경로와 v에서 u로 가는 경로는 항상 모두 존재한다.
- 다른 SCC 에서 각각 뽑은 임의의 점 u, v에 대해 u에서 v로 가는 경로와 v에서 u로 가는 경로는 동시에 존재하지 않는다.
- Maximal 하기 때문에 가장 큰 집합으로 형성된다.

# Strongly Connected Component (SCC, 강한 결합 요소)



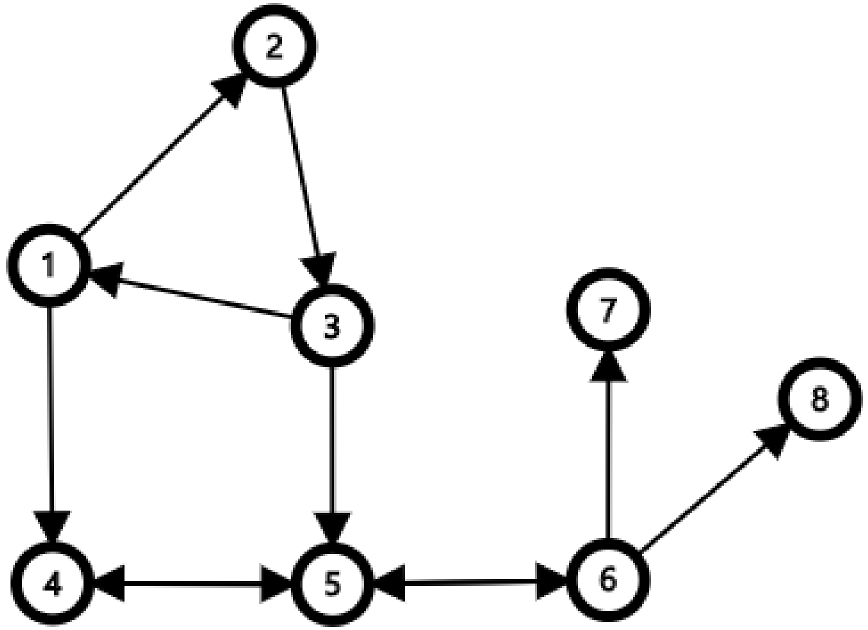
# SCC – Kosaraju

- 정방향 그래프와 역방향 그래프를 이용

- 2번의 DFS를 통해 SCC를 구하기

- 정방향 그래프에 대해서 DFS 수행
  → DFS가 끝나는 순서대로 stack에 담기
  → stack의 top부터 역방향 그래프에 대해 DFS 수행
  → 한 번 역방향 dfs 를 수행할 때, 방문하는 노드들은 한 SCC에 속함

# SCC – Kosaraju

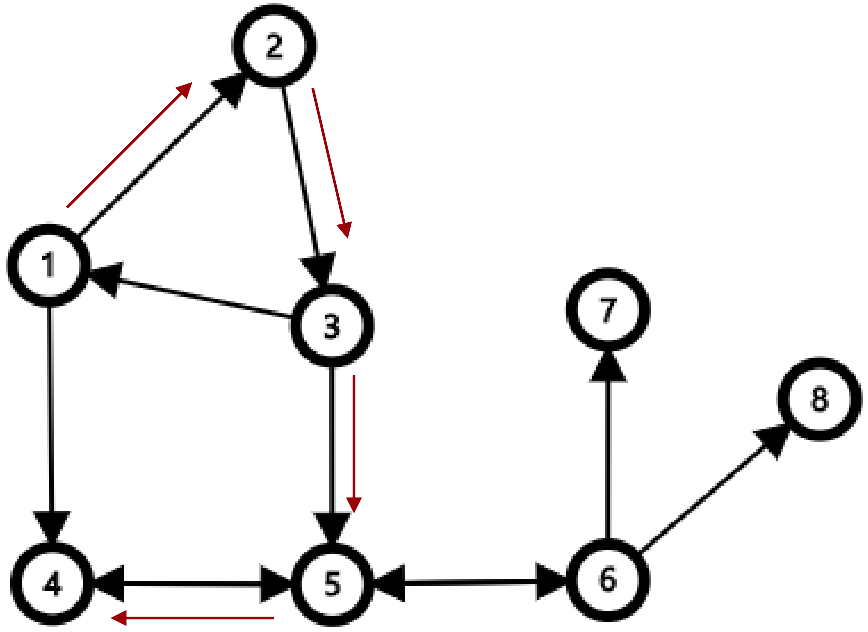



DFS 호출

dfs(1)

stack

# SCC – Kosaraju

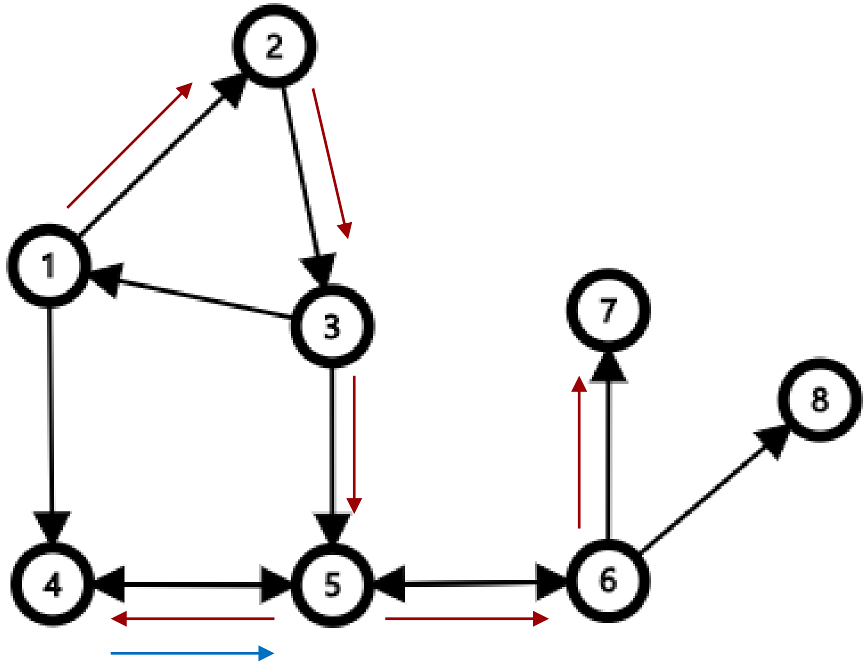



DFS 호출

dfs(1) → dfs(2) → dfs(3) → dfs(5) → dfs(4)

stack

# SCC – Kosaraju




DFS 호출

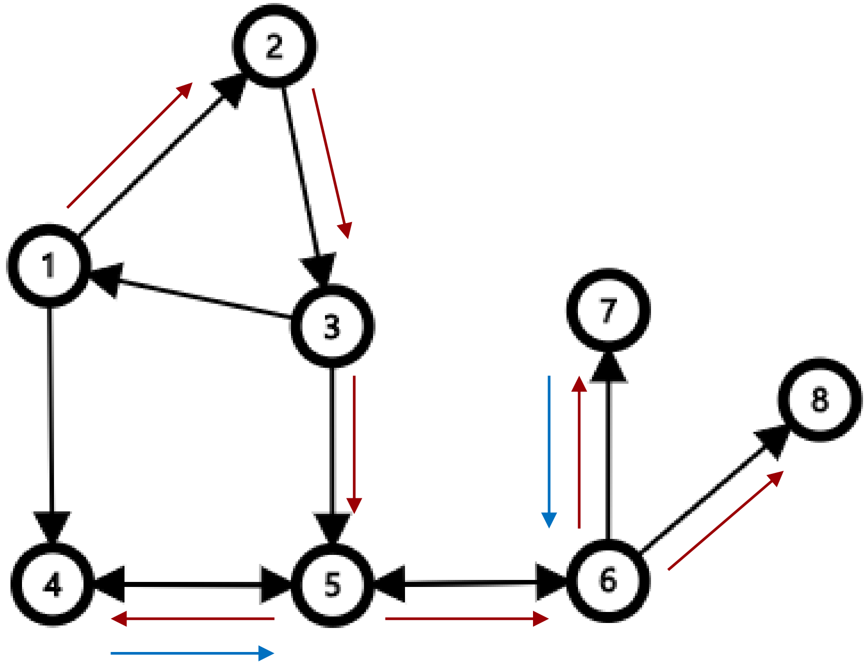
dfs(1) → dfs(2) → dfs(3) → dfs(5) → ~~dfs(4)~~

→ dfs(6) → dfs(7)

stack

4

# SCC – Kosaraju




DFS 호출

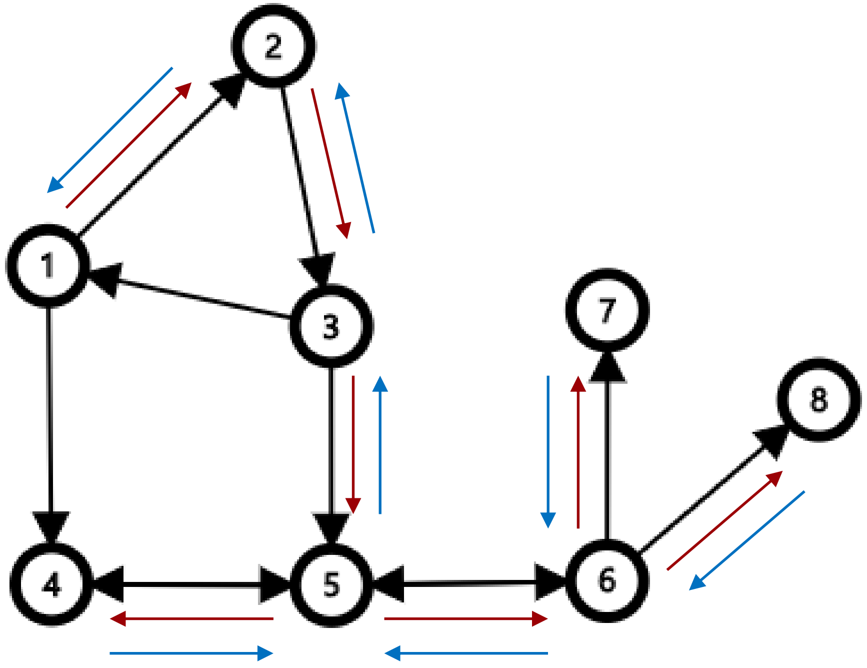
dfs(1) → dfs(2) → dfs(3) → dfs(5) → ~~dfs(4)~~

→ dfs(6) → ~~dfs(7)~~ → dfs(8)

stack

4 – 7

# SCC – Kosaraju



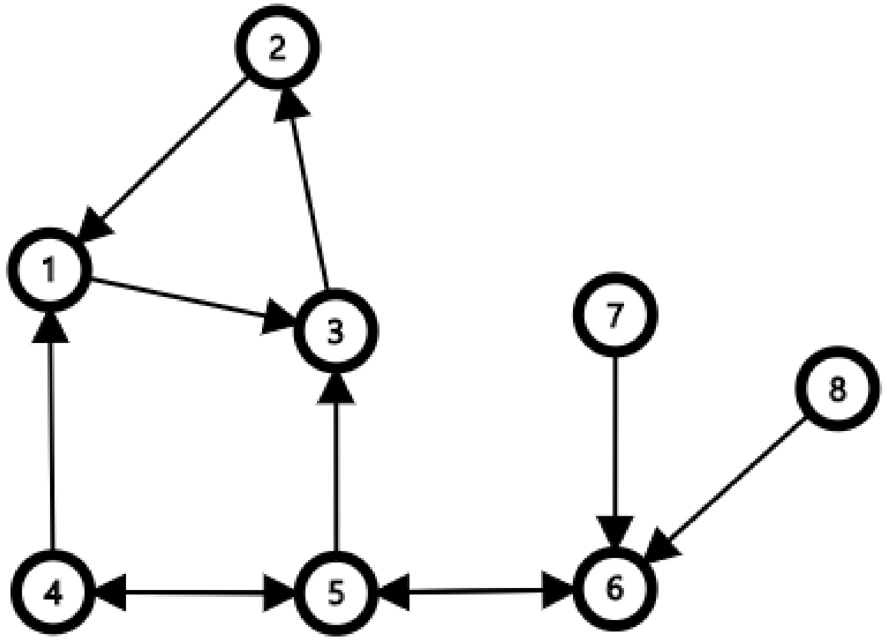

DFS 호출

~~dfs(1)~~ → ~~dfs(2)~~ → ~~dfs(3)~~ → ~~dfs(5)~~ → ~~dfs(4)~~
→ ~~dfs(6)~~ → ~~dfs(7)~~ → ~~dfs(8)~~

stack

4 – 7 – 8 – 6 – 5 – 3 – 2 – 1

# SCC – Kosaraju




reverse graph

stack

4 – 7 – 8 – 6 – 5 – 3 – 2 – 1

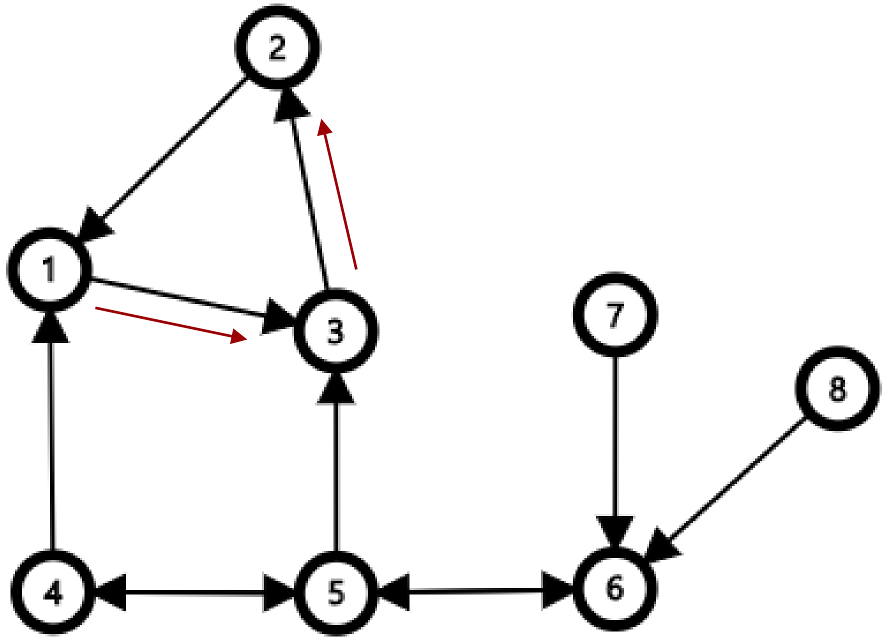
reverse-DFS 호출

rdfs(1)

SCC

{1}

# SCC – Kosaraju




reverse graph

stack

4 – 7 – 8 – 6 – 5 – 3 – 2 – ~~1~~

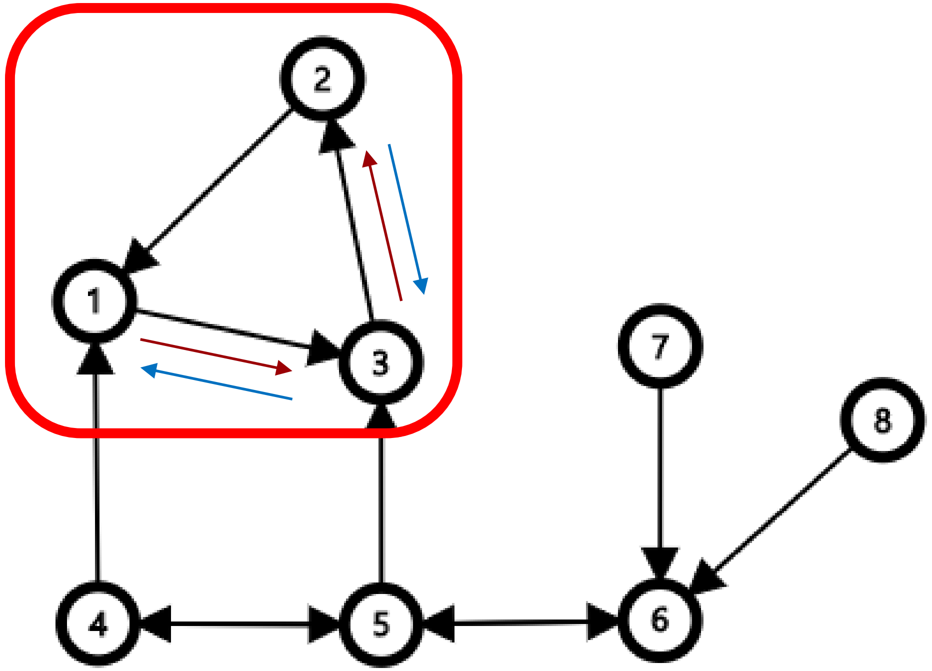
reverse-DFS 호출

rdfs(1) → rdfs(3) → rdfs(2)

SCC

{1, 2, 3}

# SCC – Kosaraju




reverse graph

stack

4 – 7 – 8 – 6 – 5 – 3 – 2 – ~~1~~

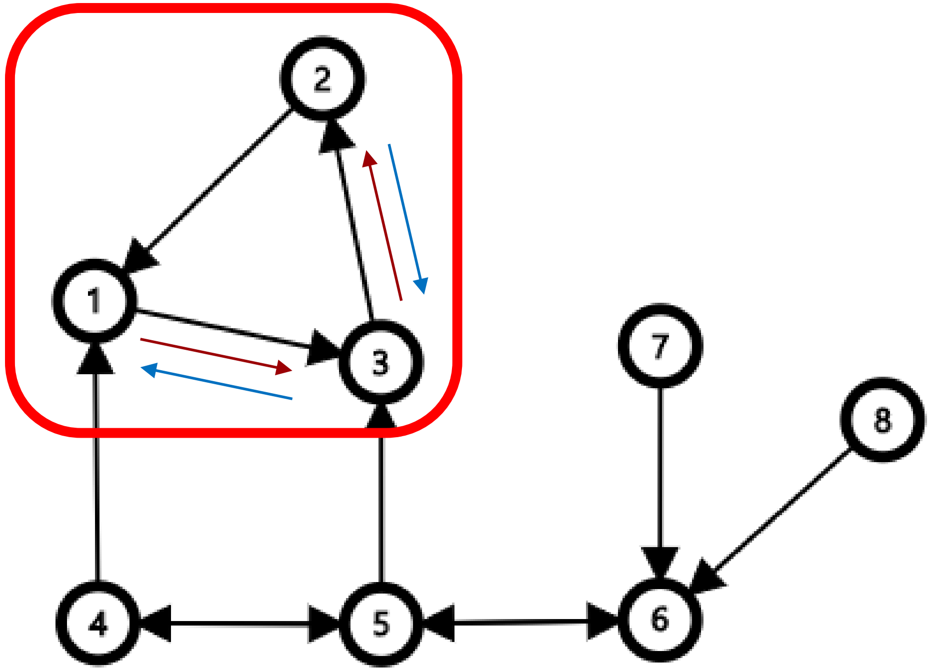
reverse-DFS 호출

~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

SCC

{1, 2, 3}

# SCC – Kosaraju




reverse graph

stack

4 – 7 – 8 – 6 – 5 – ~~3~~ – ~~2~~ – ~~1~~

reverse-DFS 호출

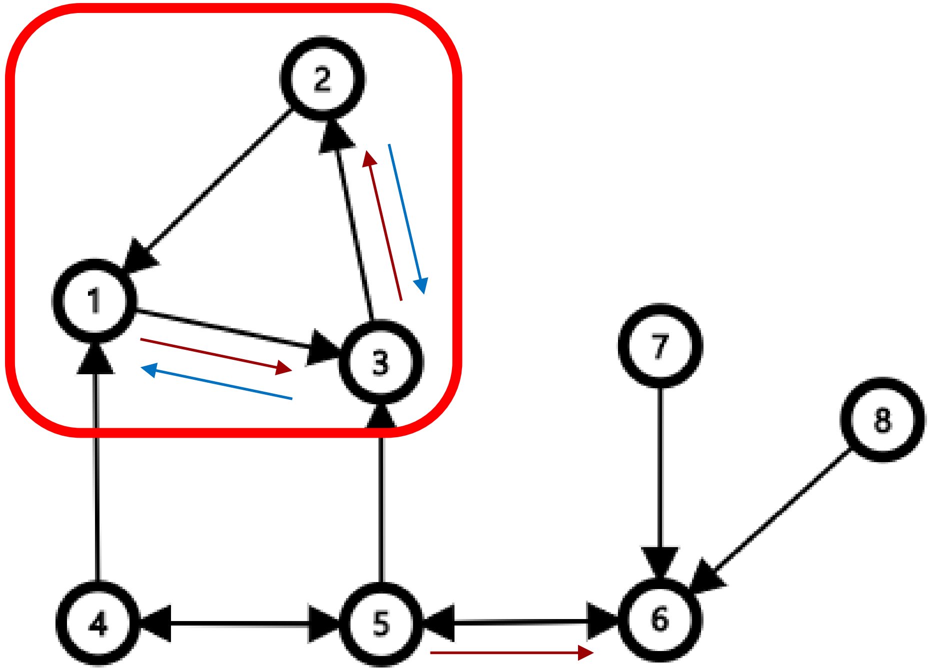
~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

rdfs(5)

SCC

{1, 2, 3} , {5}

# SCC – Kosaraju




reverse graph

stack

4 – 7 – 8 – 6 – ~~5~~ – ~~3~~ – ~~2~~ – ~~1~~

reverse-DFS 호출

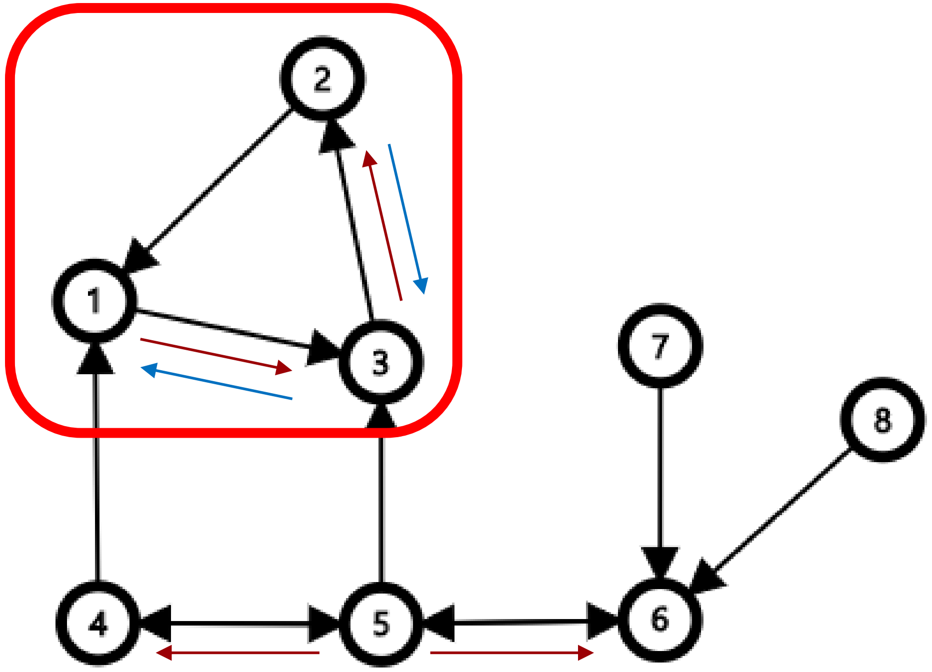
~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

rdfs(5) → rdfs(6)

SCC

{1, 2, 3} , {5, 6}

# SCC – Kosaraju



reverse graph

stack

4 – 7 – 8 – 6 – ~~5~~ – ~~3~~ – ~~2~~ – ~~1~~

reverse-DFS 호출

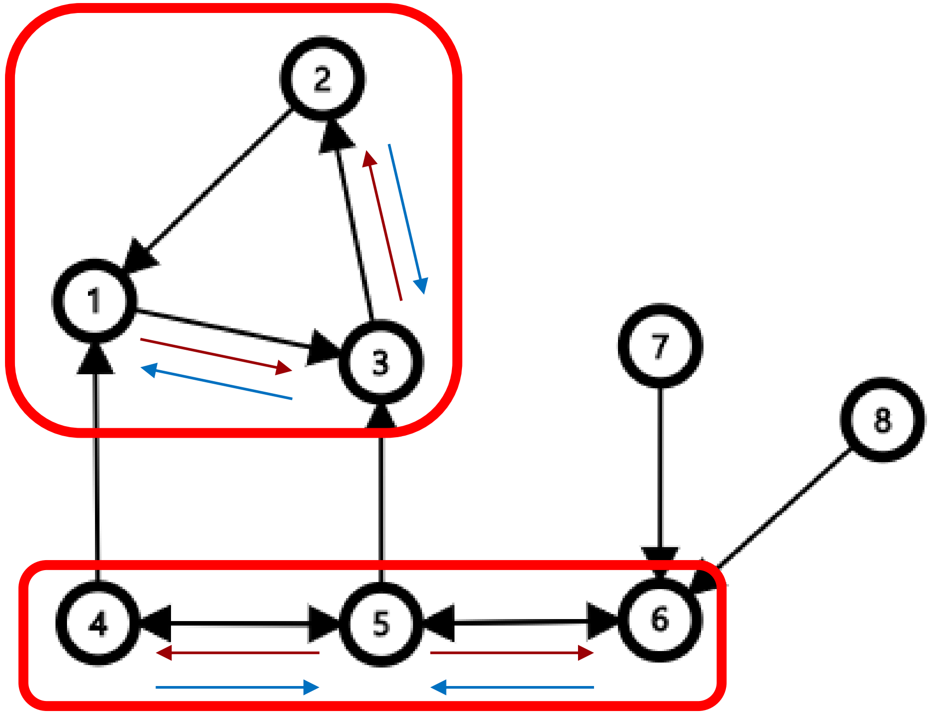
~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

rdfs(5) → ~~rdfs(6)~~ → rdfs(4)

SCC

{1, 2, 3} , {5, 6, 4}

# SCC – Kosaraju




reverse graph

stack

4 – 7 – 8 – 6 – ~~5~~ – ~~3~~ – ~~2~~ – ~~1~~

reverse-DFS 호출

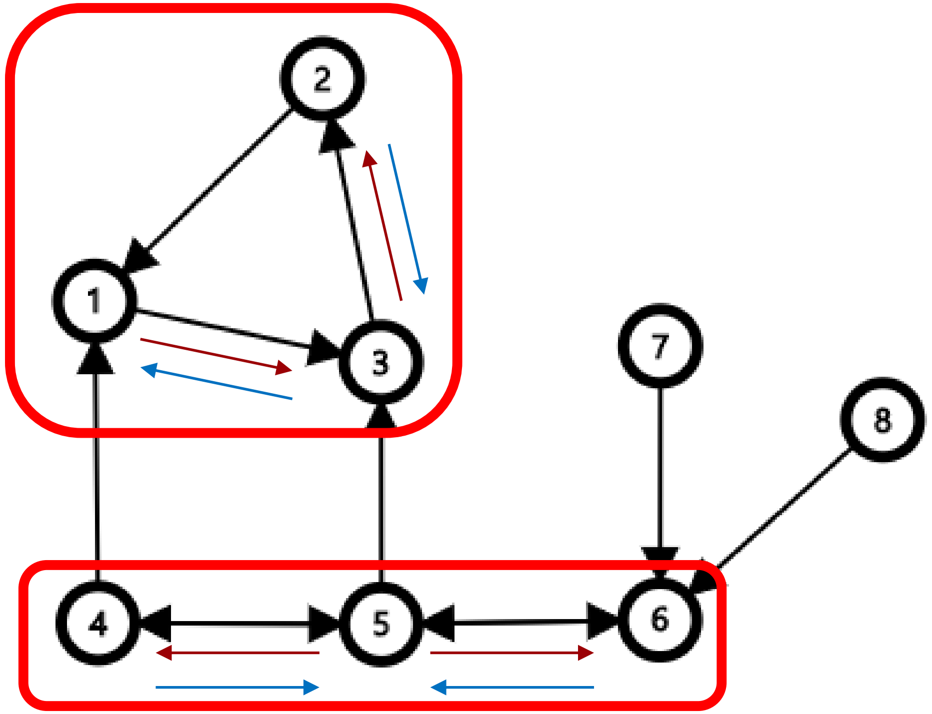
~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

~~rdfs(5)~~ → ~~rdfs(6)~~ → ~~rdfs(4)~~

SCC

{1, 2, 3} , {5, 6, 4}

# SCC – Kosaraju




reverse graph

stack

4 – 7 – 8 – ~~6~~ – ~~5~~ – ~~3~~ – ~~2~~ – ~~1~~

reverse-DFS 호출

~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

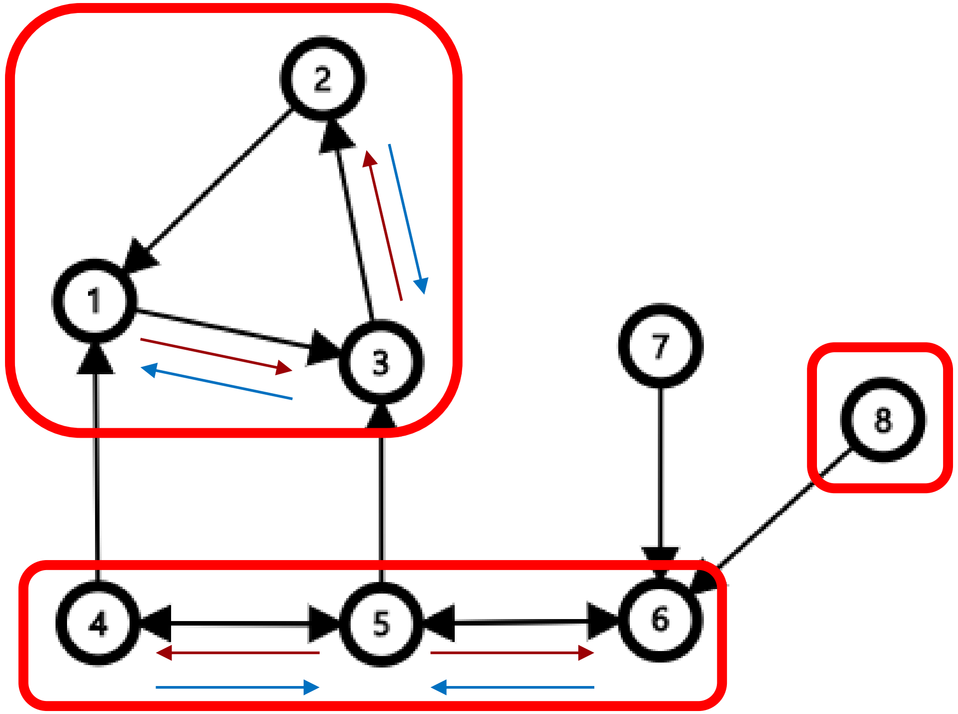
~~rdfs(5)~~ → ~~rdfs(6)~~ → ~~rdfs(4)~~

rdfs(8)

SCC

{1, 2, 3} , {5, 6, 4}, {8}

# SCC – Kosaraju




reverse graph

stack

4 – 7 – ~~8~~ – ~~6~~ – ~~5~~ – ~~3~~ – ~~2~~ – ~~1~~

reverse-DFS 호출

~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

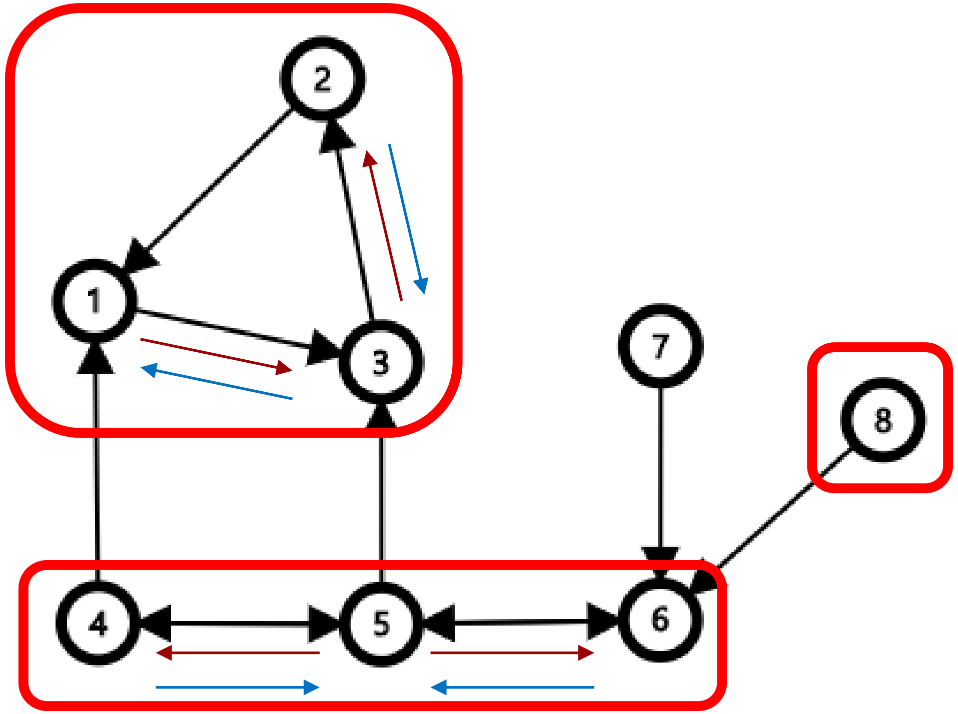
~~rdfs(5)~~ → ~~rdfs(6)~~ → ~~rdfs(4)~~

~~rdfs(8)~~

SCC

{1, 2, 3} , {5, 6, 4}, {8}

# SCC – Kosaraju



reverse graph

stack

4 – 7 – ~~8~~ – ~~6~~ – ~~5~~ – ~~3~~ – ~~2~~ – ~~1~~

reverse-DFS 호출

~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

~~rdfs(5)~~ → ~~rdfs(6)~~ → ~~rdfs(4)~~

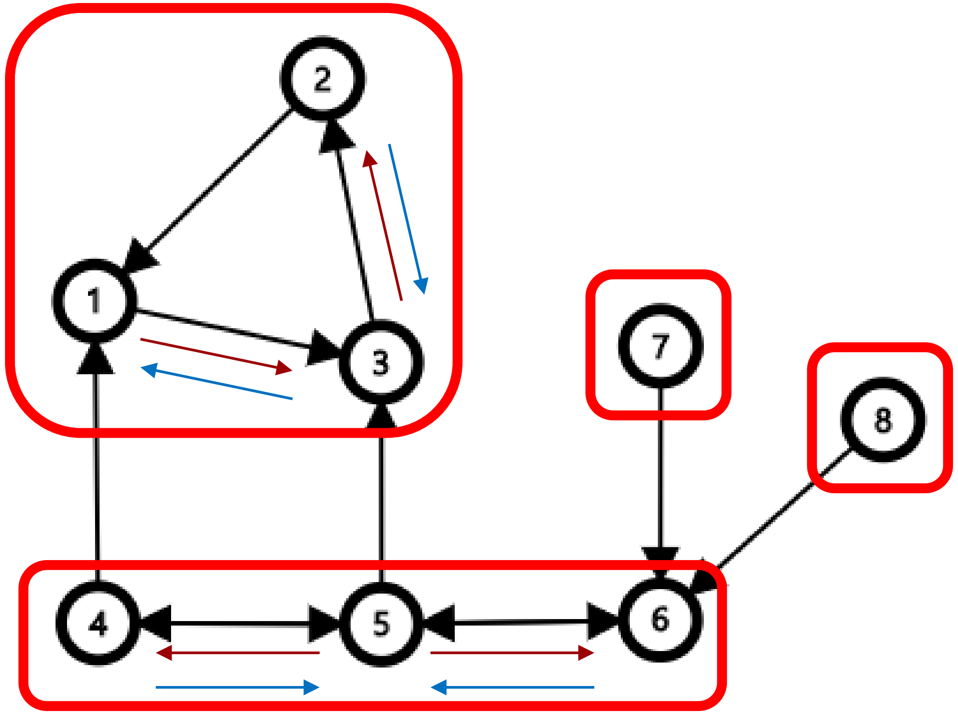
~~rdfs(8)~~

rdfs(7)

SCC

{1, 2, 3} , {5, 6, 4}, {8}, {7}

# SCC – Kosaraju




reverse graph

stack

4 – ~~7~~ – ~~8~~ – ~~6~~ – ~~5~~ – ~~3~~ – ~~2~~ – ~~1~~

reverse-DFS 호출

~~rdfs(1)~~ → ~~rdfs(3)~~ → ~~rdfs(2)~~

~~rdfs(5)~~ → ~~rdfs(6)~~ → ~~rdfs(4)~~

~~rdfs(8)~~

~~rdfs(7)~~

SCC

{1, 2, 3} , {5, 6, 4}, {8}, {7}

# SCC – Kosaraju 증명

1. 같은 SCC의 점들은 같은 역방향 dfs에서 나온다.

2. 같은 역방향 dfs에서 등장한 점들은 같은 SCC에 속한다.

# SCC – Kosaraju 증명

1. 같은 SCC의 점들은 같은 역방향 dfs에서 나온다.

정방향 그래프의 같은 SCC 의 임의의 두 점 u, v 에 대해 u → v 경로는 항상 존재

⇒ 역방향 그래프에서 v → u 경로 역시 자명하게 존재

# SCC – Kosaraju 증명

2. 같은 역방향 dfs에서 등장한 점들은 같은 SCC에 속한다.

x에서 호출한 역방향 dfs에 u, v가 등장

⇒ 역방향(정방향) 그래프에서 x → u (u → x) 경로는 항상 존재

⇒ x가 u보다 정방향 dfs가 늦게 끝났음

# SCC – Kosaraju 증명

2. 같은 역방향 dfs에서 등장한 점들은 같은 SCC에 속한다.

⇒ x가 u보다 정방향 dfs가 늦게 끝났음

u의 정방향 dfs가 먼저 호출 됐다면 ?

⇒ x가 재귀적으로 호출 됐을 것이므로 가정 모순

⇒ x가 u보다 먼저 정방향 dfs 호출 됐다.

⇒ 정방향 그래프에서 x → u 경로 존재

# SCC – Kosaraju 증명

2. 같은 역방향 dfs에서 등장한 점들은 같은 SCC에 속한다.

역방향(정방향) 그래프에서 x → u (u → x) 경로는 항상 존재
정방향 그래프에서 x → u 경로 존재

⇒ 정방향 그래프에서 u, x는 같은 SCC

⇒ v도 마찬가지이므로 u, v는 같은 SCC

# SCC – Kosaraju

- adj – 정방향 인접 리스트
- rev_adj – 역방향 인접 리스트
- SCC – SCC 그룹
- vis – dfs 시, 방문 체크
- ST – 정방향 dfs의 종료 순대로 push
- sz – SCC 의 개수

```
vector<vector<int> > adj(MXV), rev_adj(MXV), SCC;
vector<int> vis(MXV), ST;
int sz;
```

```
int v, e;
cin >> v >> e;

for (int i = 0; i < e; ++i) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    rev_adj[v].push_back(u);
}
```

# SCC – Kosaraju

- 400 ~ 402 line

  모든 정점에 대해 정방향 dfs 수행

```
399     fill(vis.begin(), vis.end(), 0);
400     for (int i = 1; i <= v; ++i)
401         if (!vis[i])
402             dfs(i);
```

- 376 line

  정방향 dfs가 끝나는 순서대로

  stack에 담기

```
370 void dfs(int cur) {
371     vis[cur] = 1;
372     for (int next : adj[cur]) {
373         if (vis[next]) continue;
374         dfs(next);
375     }
376     ST.push_back(cur);
377 }
```

# SCC – Kosaraju

- 406 line
  stack의 top을 꺼내기
- 408 ~ 409 line
  SCC 집합 생성 후, 역방향 dfs 수행
- 385 line
  역방향 dfs가 끝날 때 마다
  현재 SCC 집합에 추가해주기
- 410 line
  SCC 집합 번호 증가

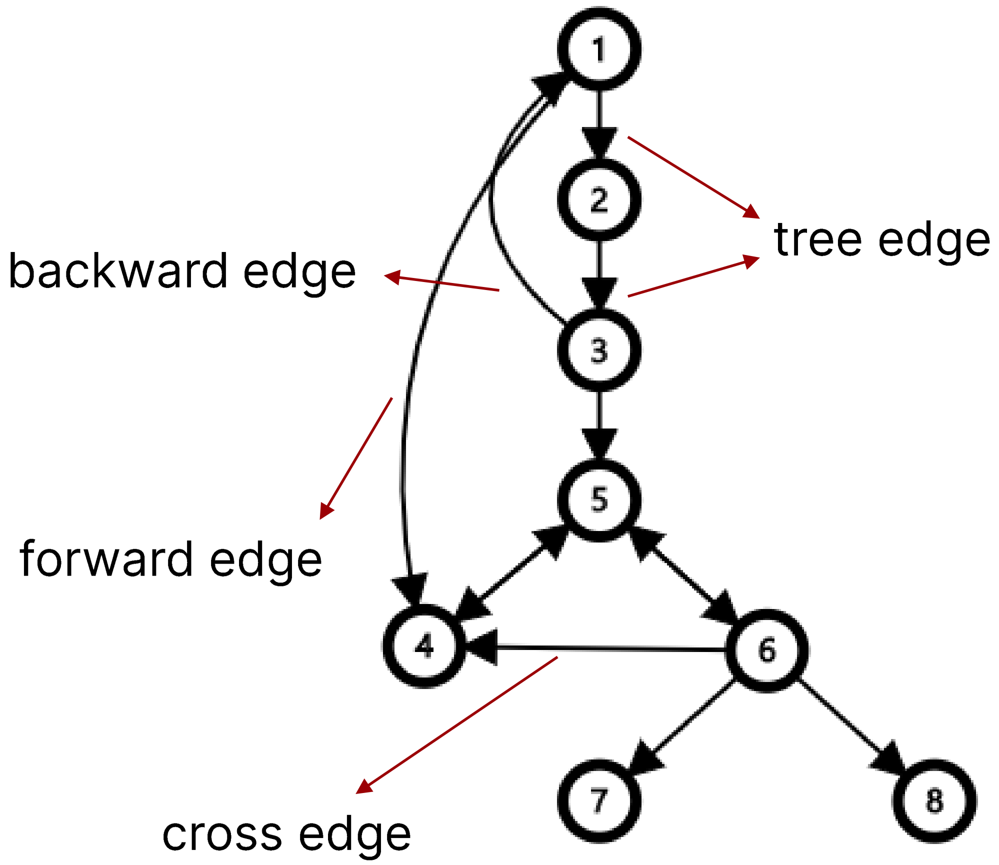
```cpp
404     fill(vis.begin(), vis.end(), 0);
405     while (!ST.empty()) {
406         int cur = ST.back(); ST.pop_back();
407         if (vis[cur]) continue;
408         SCC.push_back({});
409         rev_dfs(cur);
410         sz++;
411     }
```

```cpp
379 void rev_dfs(int cur) {
380     vis[cur] = true;
381     for (int next : rev_adj[cur]) {
382         if (vis[next]) continue;
383         rev_dfs(next);
384     }
385     SCC[sz].push_back(cur);
386 }
```

# Graph 용어 정리 – in DFS spanning tree

- tree edge(트리간선) : 스패닝 트리에 포함된 간선
- forward edge(순방향 간선) : 스패닝 트리에 포함되지 않지만 자손으로 향하는 간선
- backward edge(역방향 간선) : 선조로 향하는 간선
- cross edge(교차 간선) : 자손-선조 관계가 아닌 간선

# Graph 용어 정리 – in DFS spanning tree



# SCC – Tarjan

- dfs number 와 low값을 이용

- low := dfs spanning tree 생성시, 현재 노드가 갈 수 있는 가장 높은 자손의 dfs number

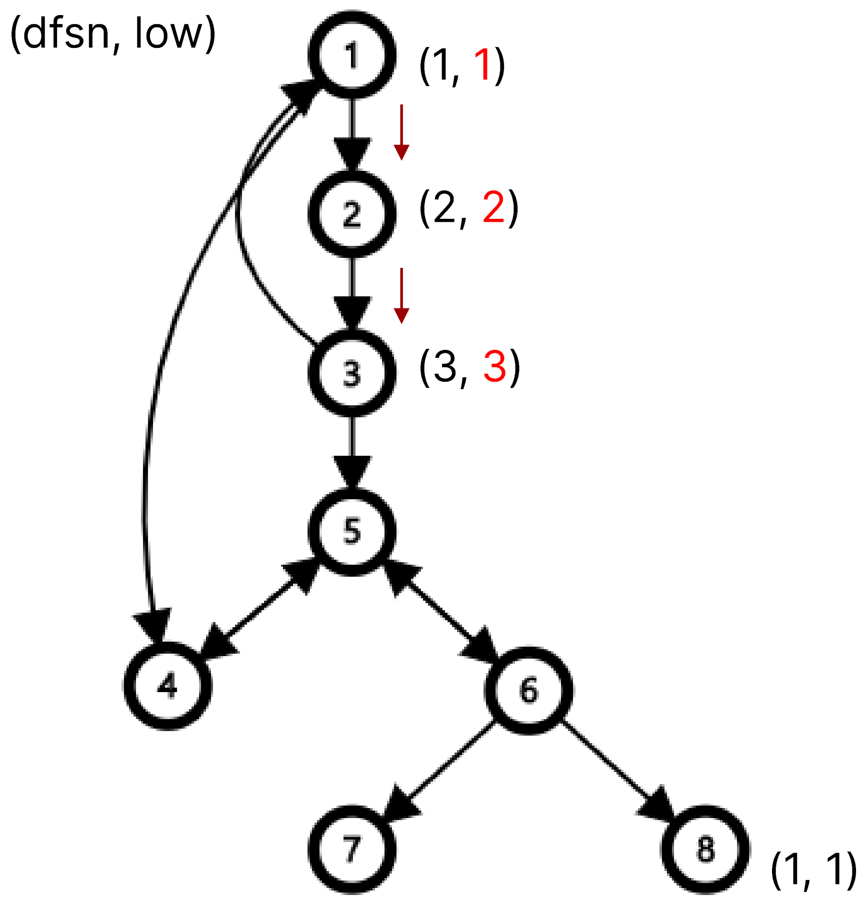
- DFS 수행하면서 방문 노드를 stack에 보관

  → low값은 현재 dfs number로 초기화 / low 값 갱신

  → low == 현재 노드 dfs number → 더 위의 선조로 올라갈 수 없다

  → stack에서 현재 노드 나올 때 까지 빼서 SCC 생성

# SCC – Tarjan




DFS 호출

dfs(1) → dfs(2) → dfs(3)

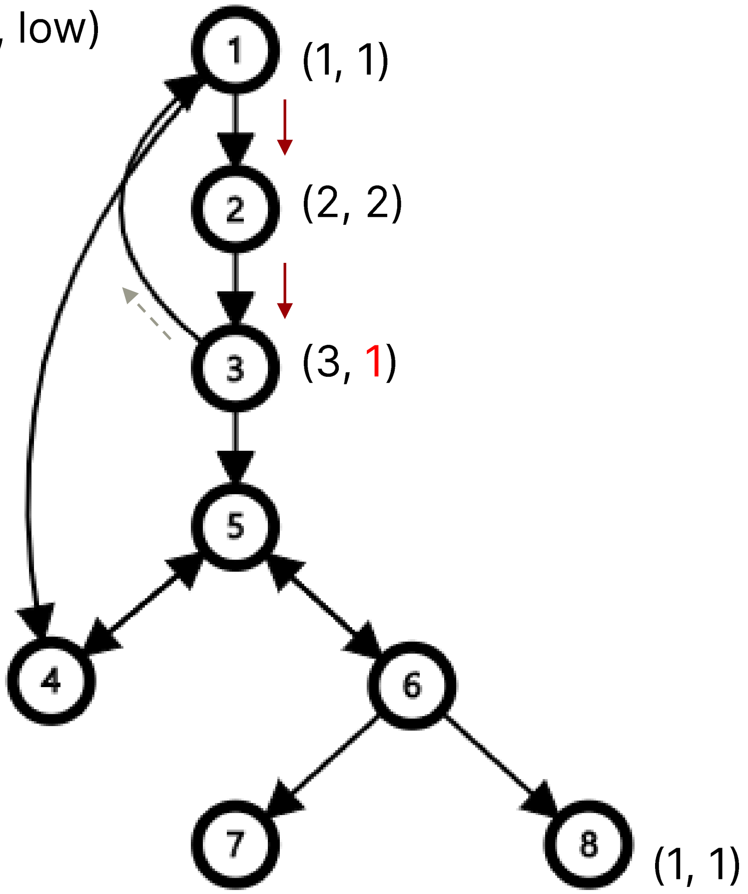
stack

1 – 2 – 3

SCC

# SCC – Tarjan

(dfsn, low)




DFS 호출

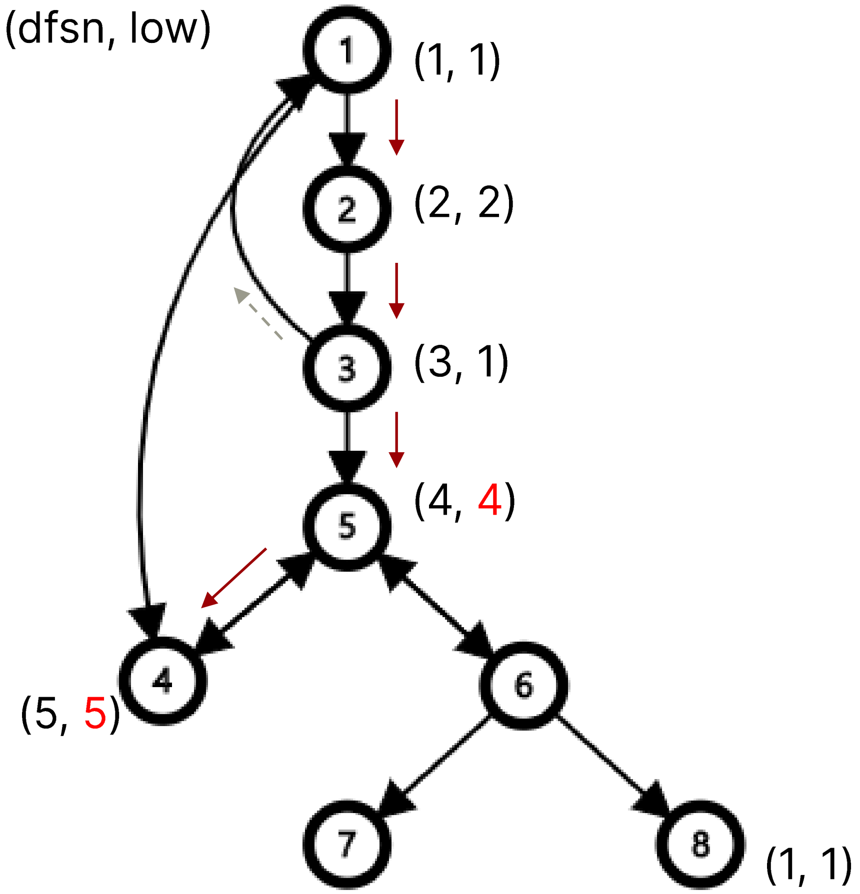
dfs(1) → dfs(2) → dfs(3)

stack

1 – 2 – 3

SCC

# SCC – Tarjan




DFS 호출

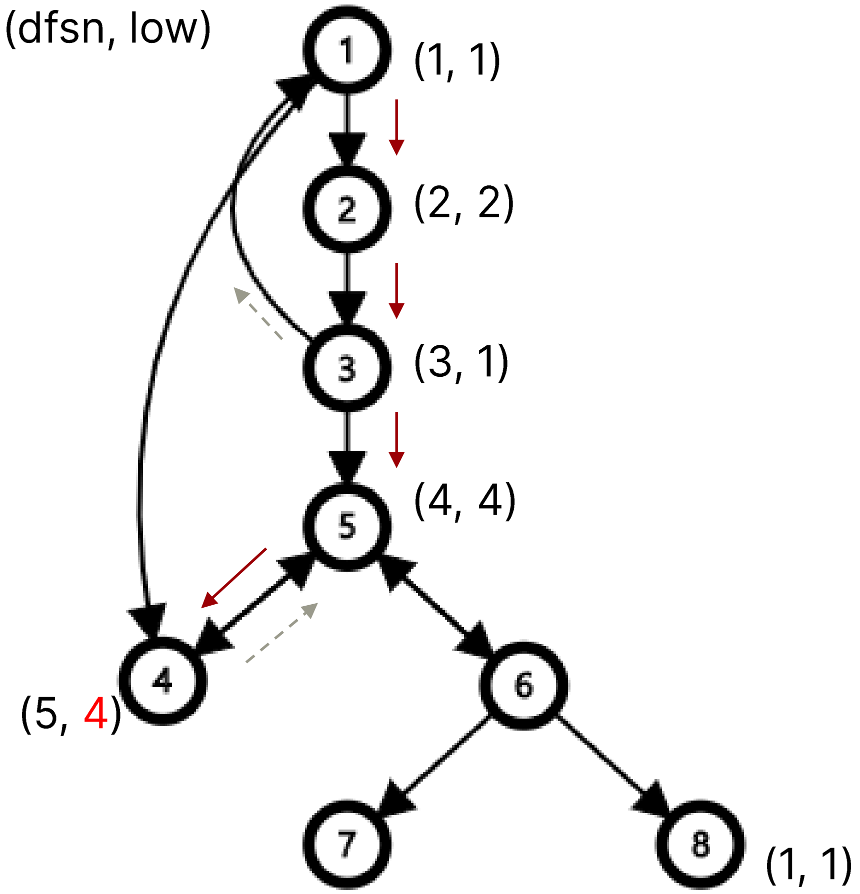
dfs(1) → dfs(2) → dfs(3) → dfs(5) → dfs(4)

stack

1 – 2 – 3 – 5 – 4

SCC

# SCC – Tarjan




DFS 호출

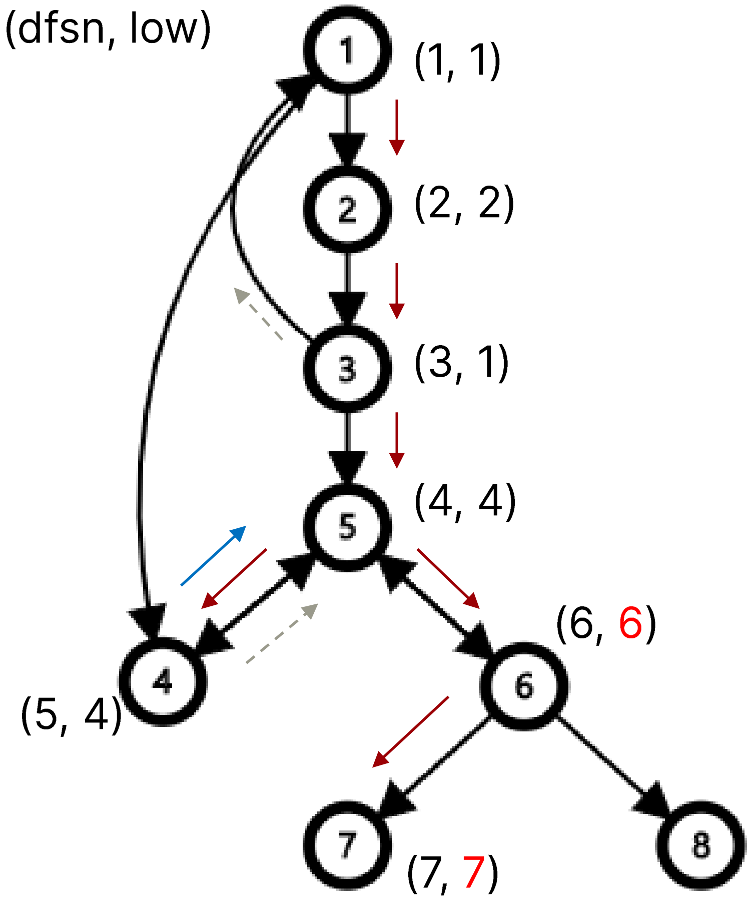
dfs(1) → dfs(2) → dfs(3) → dfs(5) → dfs(4)

stack

1 – 2 – 3 – 5 – 4

SCC

# SCC – Tarjan




DFS 호출

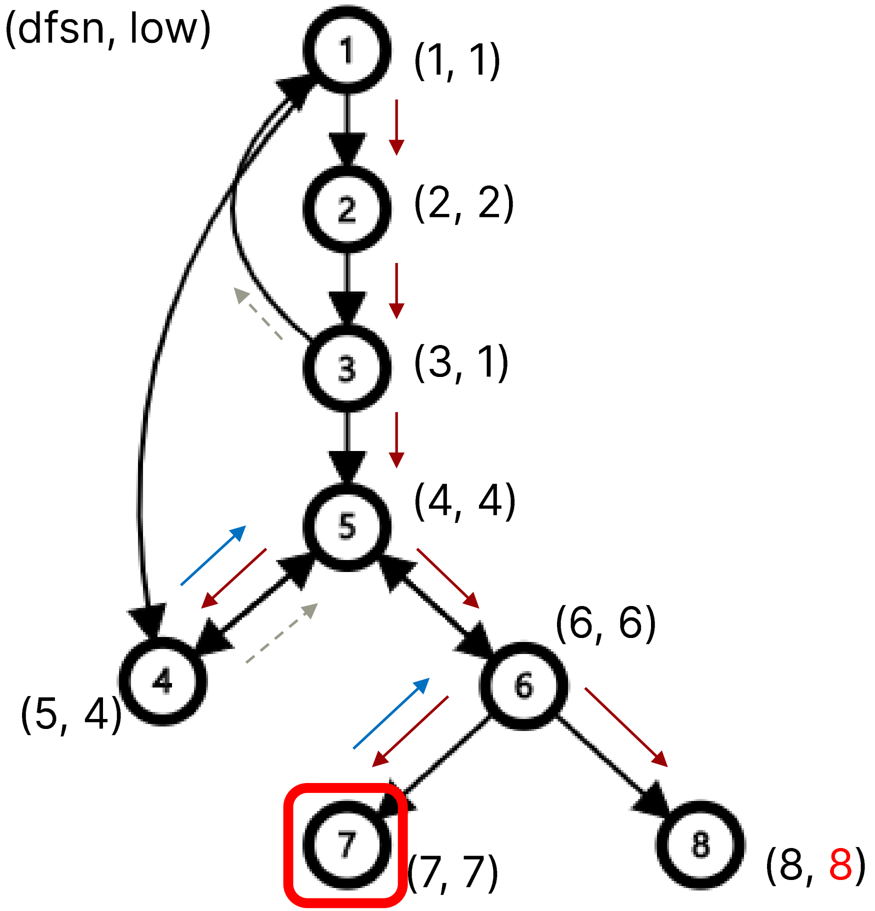
dfs(1) → dfs(2) → dfs(3) → dfs(5) → ~~dfs(4)~~ → dfs(6) → dfs(7)

stack

1 – 2 – 3 – 5 – 4 – 6 - 7

SCC

# SCC – Tarjan




DFS 호출

dfs(1) → dfs(2) → dfs(3) → dfs(5) → ~~dfs(4)~~ → dfs(6) → ~~dfs(7)~~ → dfs(8)

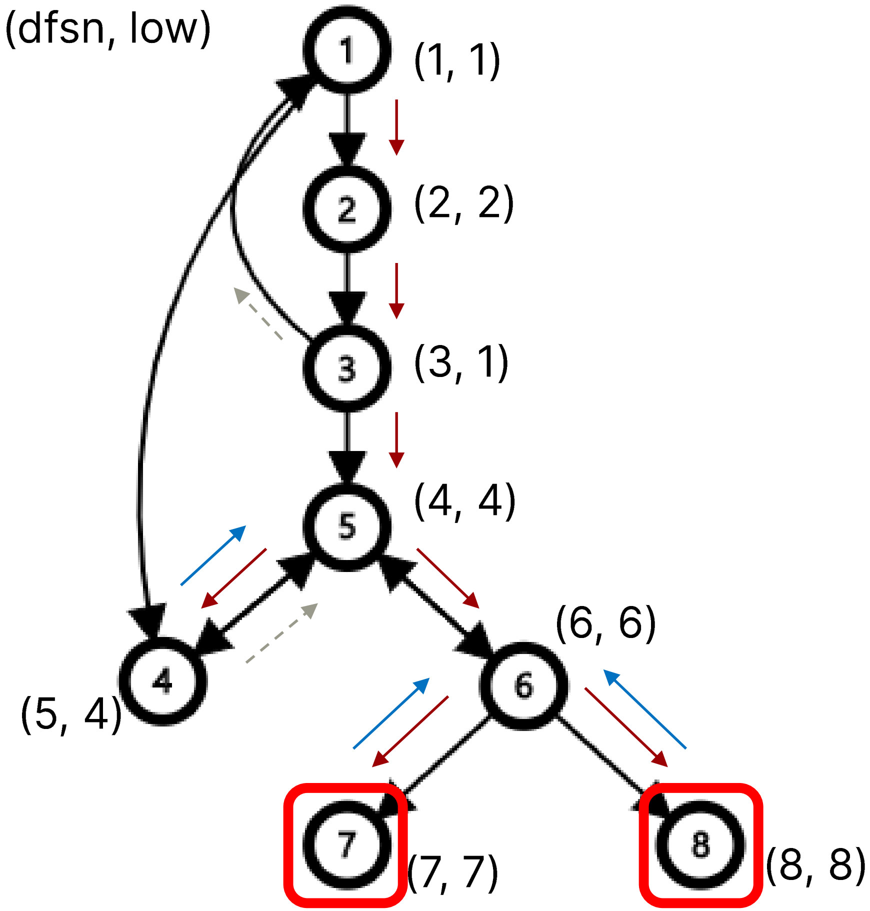
stack

1 – 2 – 3 – 5 – 4 – 6 – 7 – 8

SCC

{7}

# SCC – Tarjan




DFS 호출

dfs(1) → dfs(2) → dfs(3) → dfs(5) → ~~dfs(4)~~
→ dfs(6) → ~~dfs(7)~~ → ~~dfs(8)~~

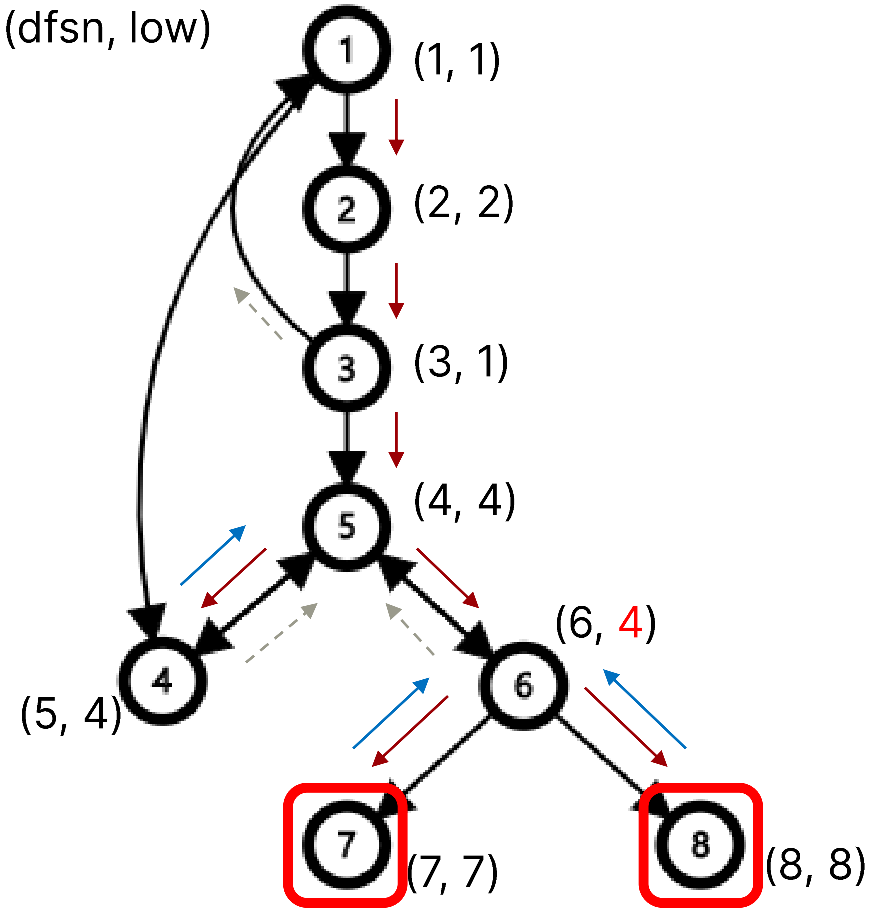
stack

1 – 2 – 3 – 5 – 4 – 6 – 7 – 8

SCC

{7} , {8}

# SCC – Tarjan




DFS 호출

dfs(1) → dfs(2) → dfs(3) → dfs(5) → ~~dfs(4)~~ → dfs(6) → ~~dfs(7)~~ → ~~dfs(8)~~

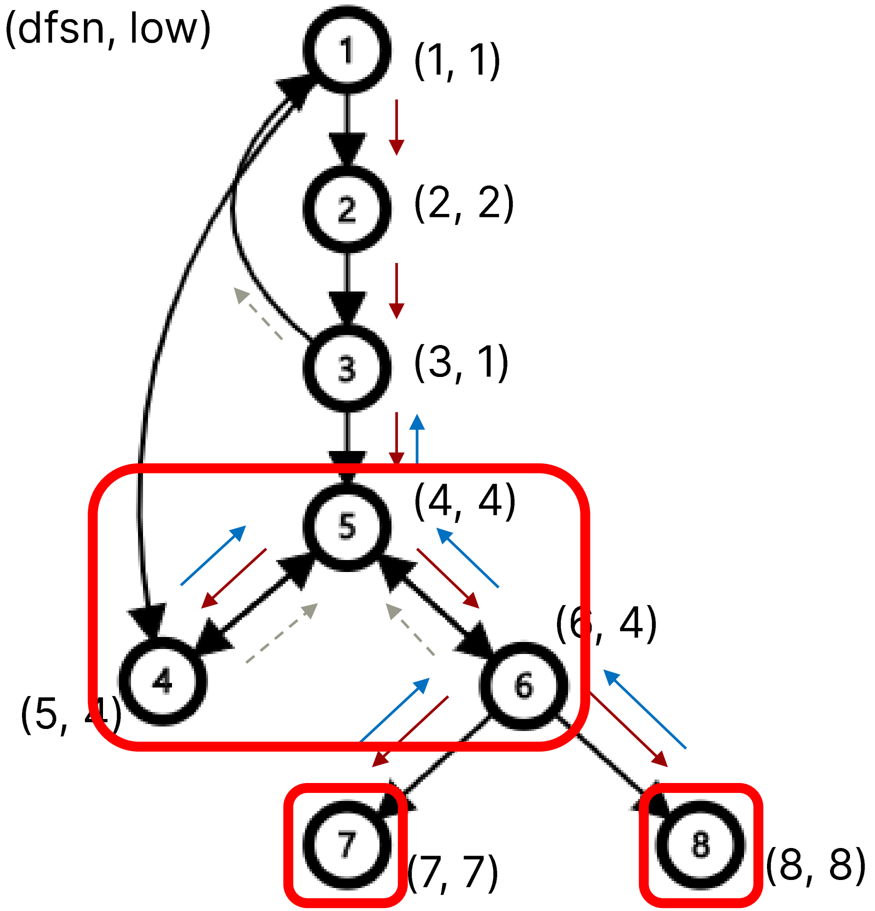
stack

1 – 2 – 3 – 5 – 4 – 6 – 7 – 8

SCC

{7} , {8}

# SCC – Tarjan




DFS 호출

dfs(1) → dfs(2) → dfs(3) → ~~dfs(5)~~ → ~~dfs(4)~~ → ~~dfs(6)~~ → ~~dfs(7)~~ → ~~dfs(8)~~

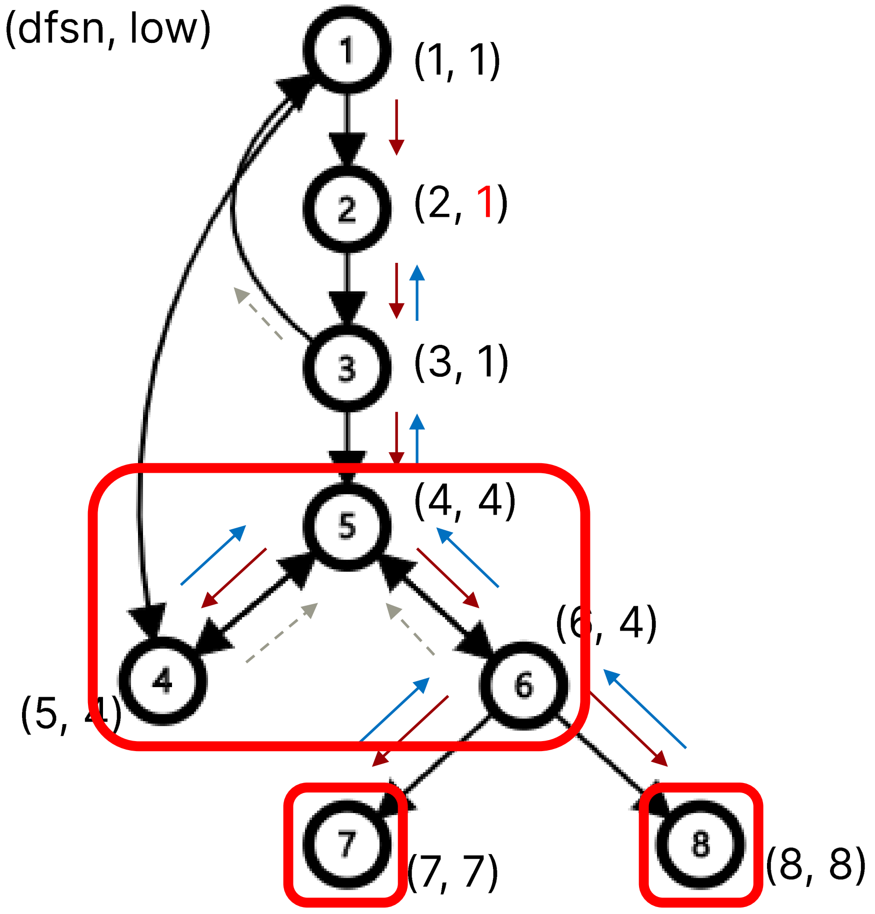
stack

1 – 2 – 3 – 5 – 4 – 6 – 7 – 8

SCC

{7} , {8} , {6, 4, 5}

# SCC – Tarjan




DFS 호출

dfs(1) → dfs(2) → dfs(3) → ~~dfs(5)~~ → ~~dfs(4)~~
→ ~~dfs(6)~~ → ~~dfs(7)~~ → ~~dfs(8)~~

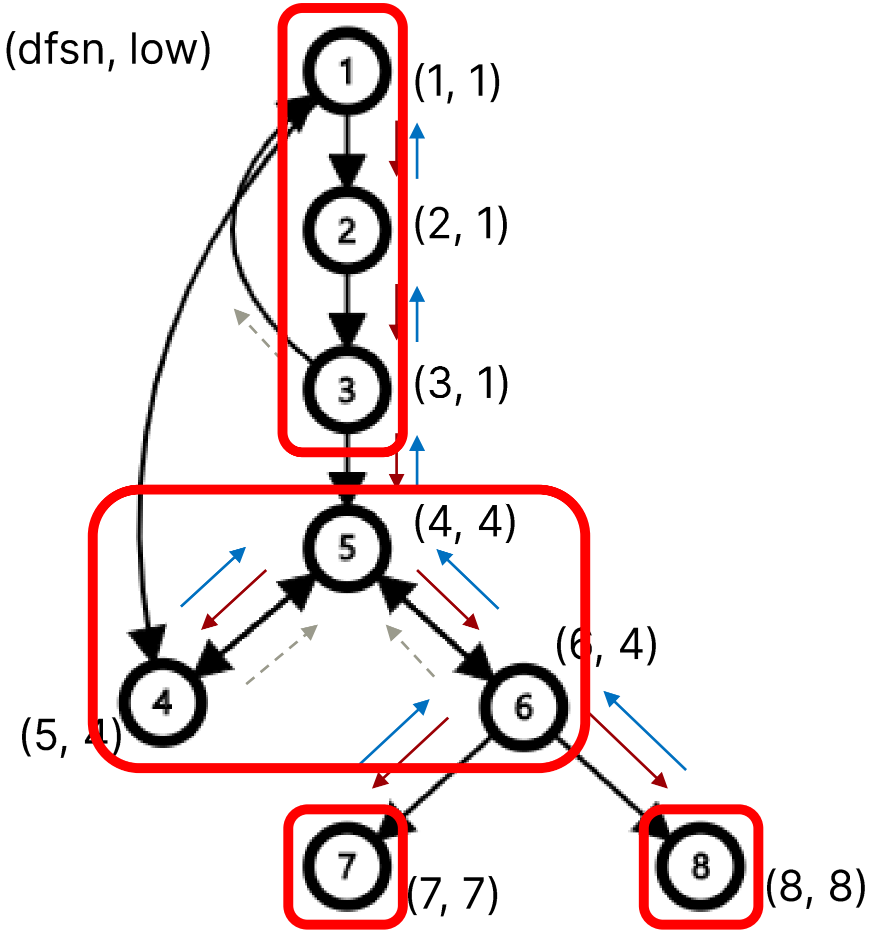
stack

1 – 2 – 3 – 5 – 4 – 6 – 7 – 8

SCC

{7} , {8} , {6, 4, 5}

# SCC – Tarjan




DFS 호출

~~dfs(1)~~ → ~~dfs(2)~~ → ~~dfs(3)~~ → ~~dfs(5)~~ → ~~dfs(4)~~ → ~~dfs(6)~~ → ~~dfs(7)~~ → ~~dfs(8)~~

stack

1 – 2 – 3 – 5 – 4 – 6 – 7 – 8

SCC

{7} , {8} , {6, 4, 5} , {3, 2, 1}

# SCC – Tarjan

- adj – 인접 리스트
- SCC – SCC 그룹
- dfsn – dfs number 기록
- fin – dfs 종료 체크 (역간선을 위함)
- ST – stack
- sz – SCC의 개수
- cnt – dfs number

```cpp
    vector<vector<int> > adj(MXV), SCC;
    vector<int> dfsn(MXV), fin(MXV), ST;
    int sz, cnt;
```

```cpp
int main() {
    int v, e;
    cin >> v >> e;
    for (int i = 0; i < e; ++i) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
    }

    for (int i = 1; i <= v; ++i)
        if (!dfsn[i])
            dfs(i);
}
```

# SCC – Tarjan

- 424 ~ 425 line

  dfs number 기록 , stack 에 push
- 429 line

  tree 간선 → 자식의 low를 받음
- 430 line

  역방향 간선 → low 값 갱신
- 433 ~ 438 line

  더 이상 못 올라갈 시, SCC 생성

```
423 int dfs(int cur) {
424     dfsn[cur] = ++cnt;
425     ST.push_back(cur);
426 
427     int low = dfsn[cur];
428     for (int next : adj[cur]) {
429         if (!dfsn[next]) low = min(low, dfs(next));
430         else if (!fin[next]) low = min(low, dfsn[next]);
431     }
432     if (low == dfsn[cur]) {
433         SCC.push_back({});
434         while (!ST.empty()) {
435             int elem = ST.back(); ST.pop_back();
436             SCC[sz].push_back(elem);
437             fin[elem] = 1;
438             if (elem == cur) break;
439         }
440         sz++;
441     }
442     return low;
443 }
```

# #4196 도미노

- 도미노 블록의 배치가 주어진다.
- 최소 몇 개의 블록을 넘어뜨려야 모두 넘어뜨릴 수 있는지 구하여라.

# #4196 도미노

- 더 앞에 있는 블록을 넘어뜨리면 뒤의 블록은 당연히 넘어진다.
- 위상 정렬을 통해 당장 넘어뜨릴 수 있는 블록이 몇 개인지 세자!
- 블록이 사이클 형태로 배치 되어 있다면?
  ⇒ SCC 로 묶어서 하나의 노드로 생각하자

# 어떻게 써먹을 수 있을까?

- SCC → 위상 정렬
- SCC → dynamic programming
- 2 SAT problem

# 용어 정리

- CNF(Conjunctive normal form) : literal의 논리합으로 된 clause들을 논리곱으로 나타낸 식
- literal : Boolean 값을 갖는 명제 변수 또는 그 부정
- clause : 유한한 literal 의 집합
- 2 SAT : 한 clause 내의 literal이 2개로 제한된 CNF식

# 2 SAT Problem

- $(\neg x1 \lor x2) \land (\neg x2 \lor x3) \land (x1 \lor \neg x2)$
- 위 Boolean 식이 참일 될 수 있는가?

- $x1 = F$ , $x2 = F$ , $x3 = F$ $\Rightarrow$ True

# 2 SAT Problem

| p | q | p ⇒ q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

$p \Rightarrow q$

$\Leftrightarrow \quad \neg p \vee q$

$(\because$ 가정이 거짓 $\Rightarrow \quad \neg p$

가정과 결론이 참 $\Rightarrow \quad p \wedge q$

$\Rightarrow \neg p \vee (p \wedge q) = (\neg p \vee p) \wedge (\neg p \vee q)$

$= \neg p \vee q\,)$

# 2 SAT Problem

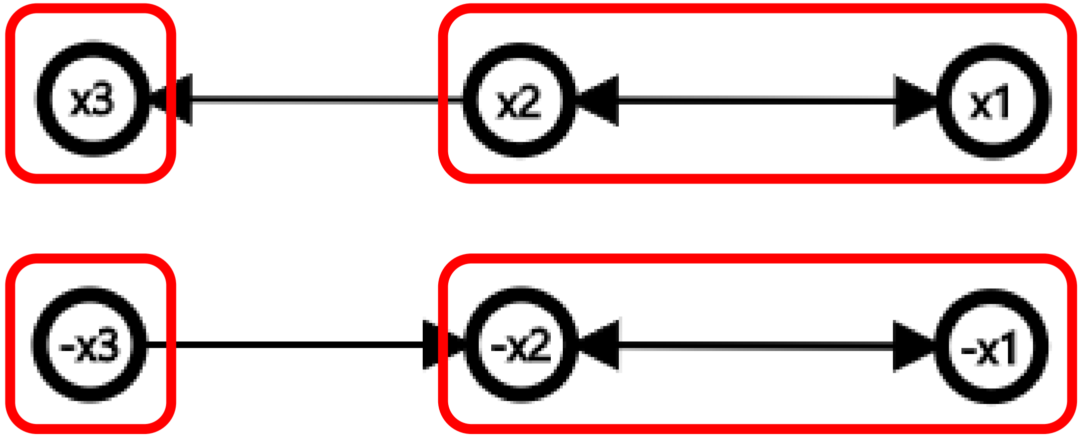
- $(\neg x1 \vee x2) \wedge (\neg x2 \vee x3) \wedge (x1 \vee \neg x2)$

- $p \vee q \Leftrightarrow \neg p \Rightarrow q \Leftrightarrow \neg q \Rightarrow p$

- $(\neg x1 \vee x2) \wedge (\neg x2 \vee x3) \wedge (x1 \vee \neg x2)$
  $\Leftrightarrow (\mathrm{x1} \Rightarrow \mathrm{x2}) \wedge (\neg x2 \Rightarrow \neg x1) \wedge (x2 \Rightarrow x3) \wedge (\neg x3 \Rightarrow \neg \mathrm{x2}) \wedge (\neg x1 \Rightarrow \neg \mathrm{x2}) \wedge (x2 \Rightarrow x1)$

# 2 SAT Problem

- $(\neg x1 \lor x2) \land (\neg x2 \lor x3) \land (x1 \lor \neg x2)$
  $\Leftrightarrow (x1 \Rightarrow x2) \land (\neg x2 \Rightarrow \neg x1) \land (x2 \Rightarrow x3) \land (\neg x3 \Rightarrow \neg x2) \land (\neg x1 \Rightarrow \neg x2) \land (x2 \Rightarrow x1)$



# 2 SAT Problem

| p | -p | p ⇒ -p | -p ⇒ p |
|---|---|---|---|
| T | F | F | T |
| F | T | T | F |

- 같은 SCC 그룹 내에 $x_i$와 $-x_i$가 같이 들어있으면
  $(x_i \Rightarrow \neg x_i) \wedge (\neg x_i \Rightarrow x_i)$ 의 식을 끌어낼 수 있는데 이는 항상 거짓이므로 모순이 일어난다.

⇒ SCC 그룹을 통해 2 SAT formula의 참/거짓을 판별할 수 있음.

각각의 Boolean 값을 설정할 수도 있는데 시간상 패스..
kks227 블로그 들어가보자 (링크)

## 문제 추천

- 2 2252 줄세우기
- 2 1766 문제집
- 3 5847 Milk Scheduling
- 5 14567 선수과목(Prerequisite)
- 5 2848 알고스팟어
- 5 2150 SCC
- 5 4196 도미노
- 4 3977 축구 전술
- 4 2152 여행계획 세우기
- 3 1108 검색 엔진
- 4 11280 2-SAT-3