



Undirected and Directed Graphs

Graph의 표현 방법, 탐색 방법 및 이들의 활용도에 대해 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)

02. Digraph(Directed Graph)의 표현(저장) 방법

03. Digraph에 대한 탐색

04. Digraph 탐색의 활용 예: 프로그램 흐름 분석, Garbage Collection, 웹 크롤링

05. Digraph 탐색의 활용 예: Topological Sort

06. Digraph 탐색의 활용 예: Strongly-Connected Component의 탐지

07. 실습: Kosaraju-Sharir 알고리즘 구현

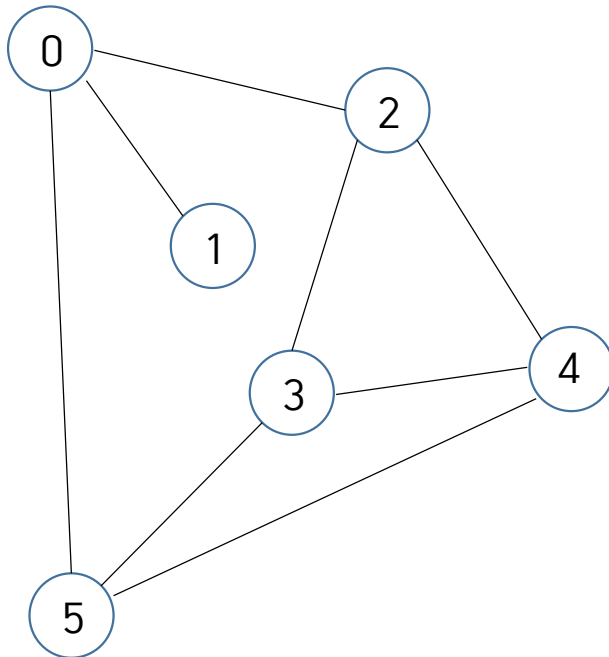
이번 시간 수업 자료에 첨부된 코드를 실행할 수 있도록 준비해 두세요.
이론 수업 중에 실행해 볼 예정입니다.

오늘 보는 탐색법이 앞으로 볼 많은 그래프 알고리즘의 기본이 됨
(이들을 그대로 활용하거나, 조합해 활용하거나, 변형해 활용)



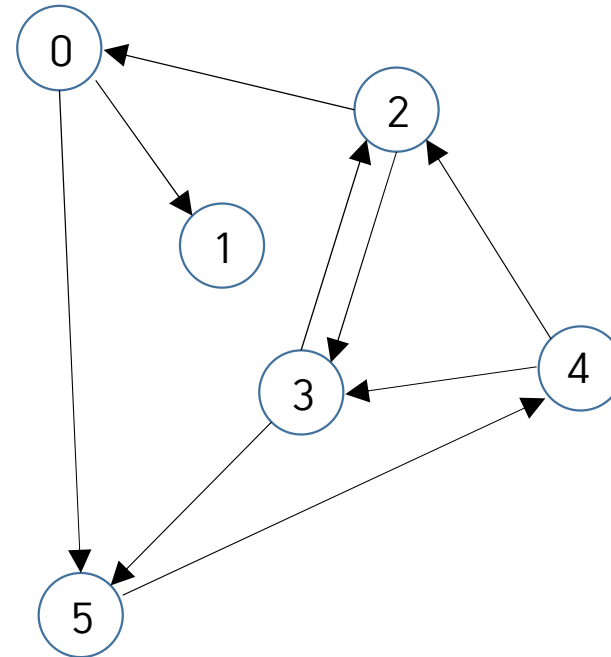
Undirected Graph

간선에 방향성 없는 그래프



Digraph (Directed Graph)

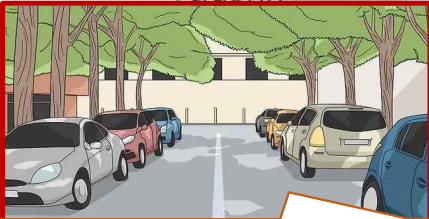
간선에 방향성 있는 그래프





Undirected Graph

간선에 방향성 없는 그래프



대부분의 도로가 양방향 통행 가능한 경우



인터넷 통신: 대부분 양방향 통신

Digraph (Directed Graph)

간선에 방향성 있는 그래프

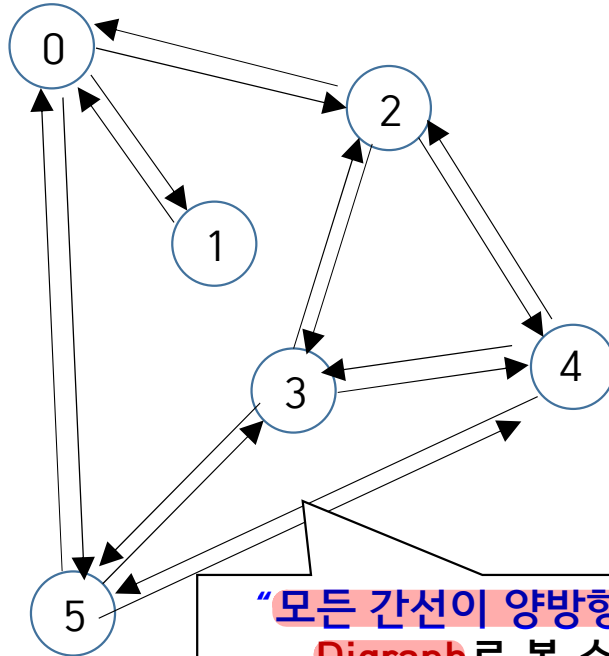


일방통행인 도로도 존재하는 경우



Undirected Graph

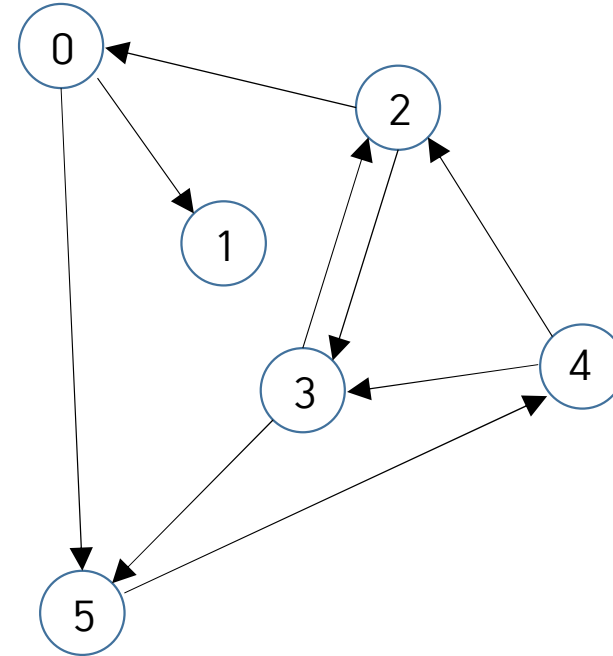
간선에 방향성 없는 그래프



“모든 간선이 양방향으로 있는”
Digraph로 볼 수 있으며,
undirected graph로 단순화해 표현

Digraph (Directed Graph)

간선에 방향성 있는 그래프



- 특정 조건 만족하는 (모든 간선이 양방향) Digraph의 집합. 따라서 Digraph의 **부분집합**
- Undirected graph **알고리즘**: 조건에 맞는 일부 경우에만 동작하는 알고리즘이므로 상대적으로 **간단** (예: connected components 구하는데 DFS 그대로 사용)

- Undirected/directed 모두 포함하므로, 더 **일반적, 포괄적 집합**
- Digraph **알고리즘**: 더 다양한 경우에 대해 모두 동작해야 하므로 상대적으로 더 **복잡** (예: connected components 구하는 Kosaraju-Sharir 알고리즘)

Undirected Graph

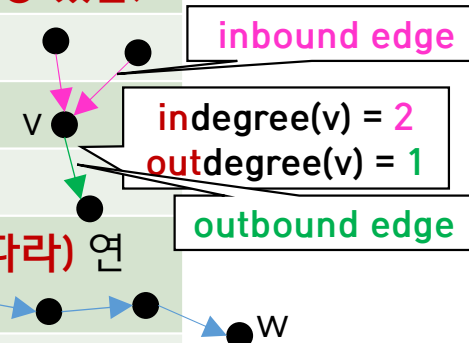
간선에 방향성 없는 그래프

용어	정의
정점	그래프 구성하는 점
V	정점 수
간선, 절선	정점을 연결하는 선(방향 없음)
E	간선 수
$\text{degree}(v)$, 차수	v 에 연결된 간선 수
v - w path	v 에서 w 까지 연결하는 경로
cycle	시작점 = 끝점인 경로
length of path	경로 따라 거쳐가는 간선 수
$\text{connected}(v, w)$	v 와 w 연결하는 경로 존재
connected component	서로 간 모두 연결된 정점의 최대 집합

Digraph (Directed Graph)

간선에 방향성 있는 그래프

용어	정의
정점	그래프 구성하는 점
V	정점 수
간선, 절선	정점을 연결하는 선(방향 있음)
E	간선 수
$\text{indegree}(v)$	v 로 들어오는 간선 수
$\text{outdegree}(v)$	v 에서 나가는 간선 수
$v \rightarrow w$ path	v 에서 w 까지 (방향에 따라) 연결하는 경로
cycle	시작점 = 끝점인 경로
length of path	경로 따라 거쳐가는 간선 수
strongly-connected (v, w)	v 에서 w 까지 경로 존재 and w 에서 v 까지 경로 존재
strongly-connected component	서로 간 모두 연결된 정점의 최대 집합



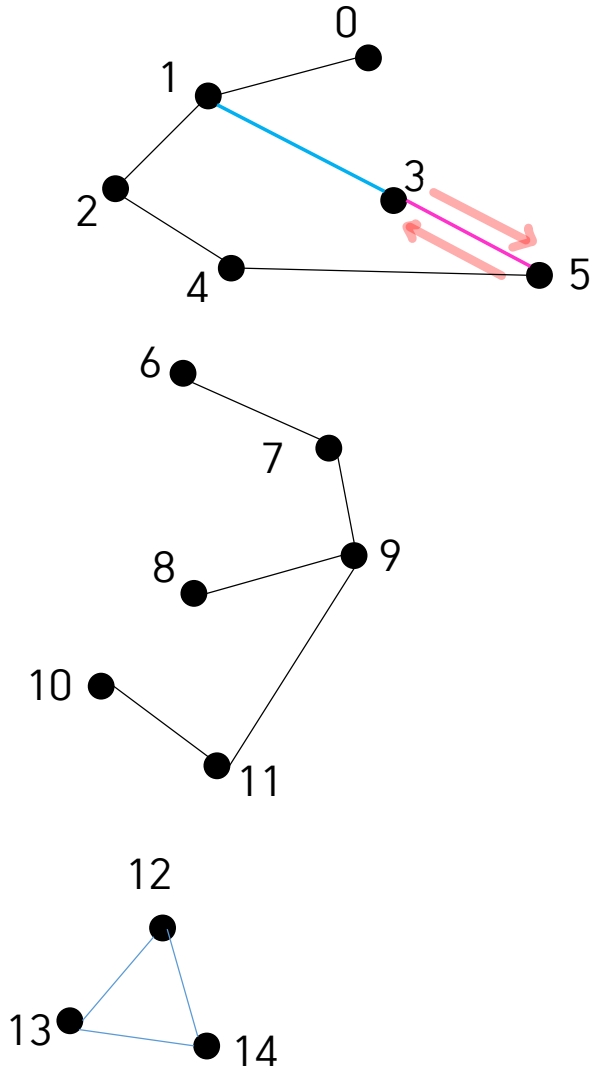


(반응이 있는 그래프)

Undirected Graph의 표현(저장) 방식

Adjacency-list: 각 점에 **인접한 정점**의 목록 저장

6



adj[]

0	→ [1]
1	→ [2, 3]
2	→ [1, 4]
3	→ [1, 5]
4	→ [2, 5]
5	→ [3, 4]
6	→ [7]
7	→ [6, 9]
8	→ [9]
9	→ [7, 8, 11]
10	→ [11]
11	→ [9, 10]
12	→ [13, 14]
13	→ [12, 14]
14	→ [12, 13]

왼쪽 그래프를 adjacency-list
방식으로 저장한 결과

같은 간선이 두 번씩 목록에 저장됨

“모든 간선이 양방향으로 있는”
Digraph로 볼 수 있기 때문



(방향성이 있는 그래프)

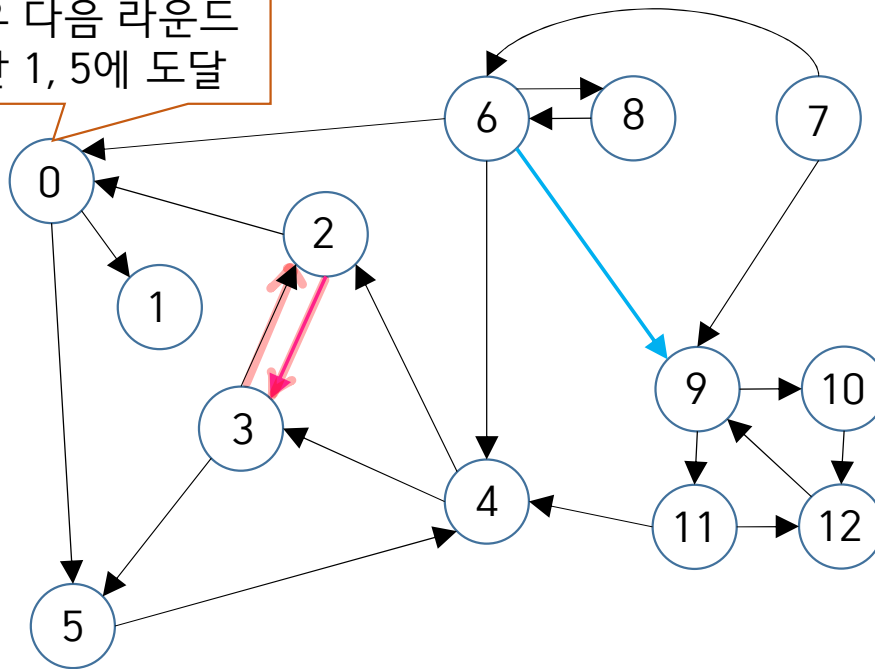
Digraph의 표현(저장) 방식

Adjacency-list: 각 점에서 **나가는 정점**의 목록 저장

왼쪽 그래프를 adjacency-list
방식으로 저장한 결과

7

정점 0에서 그래프 탐색
시작한 경우 다음 라운드
에는 인접한 1, 5에 도달



adj[]

0	→ [1,5]
1	→ []
2	→ [0,3]
3	→ [2,5]
4	→ [2,3]
5	→ [4]
6	→ [0,4,8,9]
7	→ [6,9]
8	→ [6]
9	→ [10,11]
10	→ [12]
11	→ [4,12]
12	→ [9]

각 정점에서 나가는 정점의 목록
저장: 각 간선은 **한 번씩만**
목록에 저장됨

왜 adjacency-matrix나 edge-list 방식 아닌 **adjacency-list 방식 사용**하나?

(1) 각 정점에 인접한 정점을 iterate하는 기능 많이 필요

(2) 많은 실제 문제의 경우 그래프의 정점은 매우 많으나 sparse하므로

효율



Undirected Graph

모든 간선이 양방향인 Digraph

```
class Graph:
```

```
    def __init__(self, V):  
        self.V = V  
        self.E = 0  
        self.adj = [[] for _ in range(V)]
```

```
    def addEdge(self, v, w):  
        self.adj[v].append(w)  
        self.adj[w].append(v)  
        self.E += 1
```

v, w 사이에 양방향
간선 있음 의미

```
    def degree(self, v):  
        return len(self.adj[v])
```

Digraph (Directed Graph)

간선에 방향성 있는 그래프

```
class Digraph:
```

```
    def __init__(self, V):  
        self.V = V  
        self.E = 0  
        self.adj = [[] for _ in range(V)]
```

```
    def addEdge(self, v, w):  
        self.adj[v].append(w)  
        self.E += 1
```

$v \rightarrow w$ 방향 간선
있음 의미

```
    def outDegree(self, v):  
        return len(self.adj[v])
```




Undirected Graph

모든 간선이 양방향인 Digraph

Digraph (Directed Graph)

간선에 **방향성** 있는 그래프

9

```
class Graph:
```

```
    def __init__(self, V):
        self.V = V
        self.E = 0
        self.adj = [[] for _ in range(V)]
```

```
    def addEdge(self, v, w):
        self.adj[v].append(w)
        self.adj[w].append(v)
        self.E += 1
```

```
    def degree(self, v):
        return len(self.adj[v])
```

```
class Digraph:
```

```
    def __init__(self, V):
        self.V = V
        self.E = 0
        self.adj = [[] for _ in range(V)]
```

```
    def addEdge(self, v, w):
        self.adj[v].append(w)
        self.E += 1
```

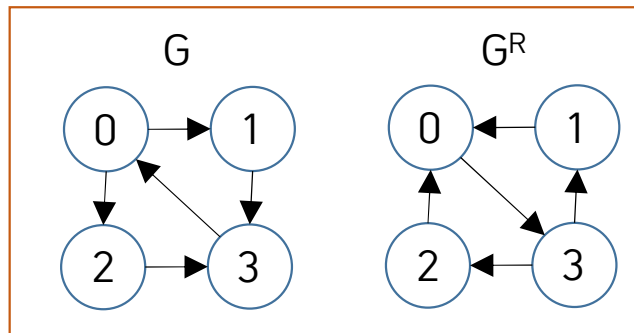
Strongly-connected
component 찾을 때 필요

```
    def outDegree(self, v):
        return len(self.adj[v])
```

모든 간선을 반대 방향으로 뒤집은 그래프
(reverse graph, G^R) 생성해 반환
(52면)

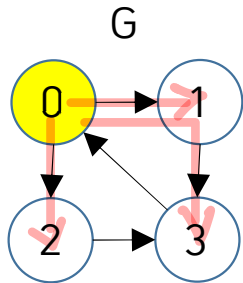
```
    def reverse(self):
        g = Digraph(self.V)
        for v in range(self.V):
            for w in self.adj[v]: g.addEdge(w, v)
        return g
```

G 에 $v \rightarrow w$ 있으면
 G^R 에는 $w \rightarrow v$ 더함

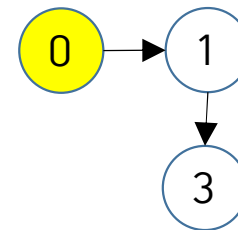
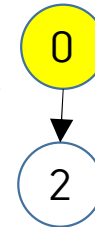
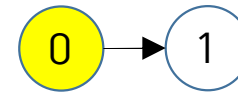




[Reverse Graph 활용 예] 그래프 G와 출발 정점 s 를 입력으로 받아 s 로부터 다른 모든 정점까지의 경로를 찾아주는 프로그램(알고리즘)이 있다. 같은 프로그램을 활용해 임의의 도착지 t 까지의 모든 경로를 찾을 수 있을까?

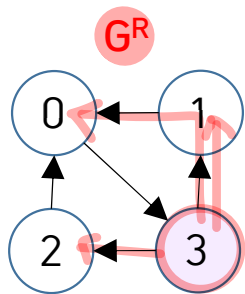


출발지 s ~ 다른 모든 정점까지
최단 경로 찾아주는
프로그램(알고리즘)

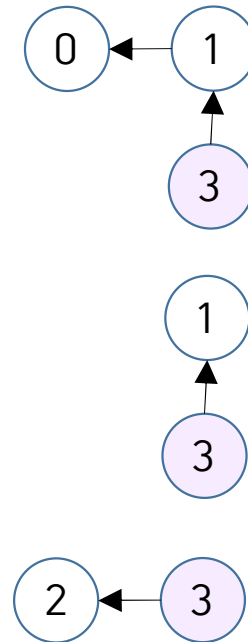




[Reverse Graph 활용 예] 그래프 G 와 출발 정점 s 를 입력으로 받아 s 로부터 다른 모든 정점까지의 경로를 찾아주는 프로그램(알고리즘)이 있다. 같은 프로그램을 활용해 임의의 도착지 t 까지의 모든 경로를 찾을 수 있을까?



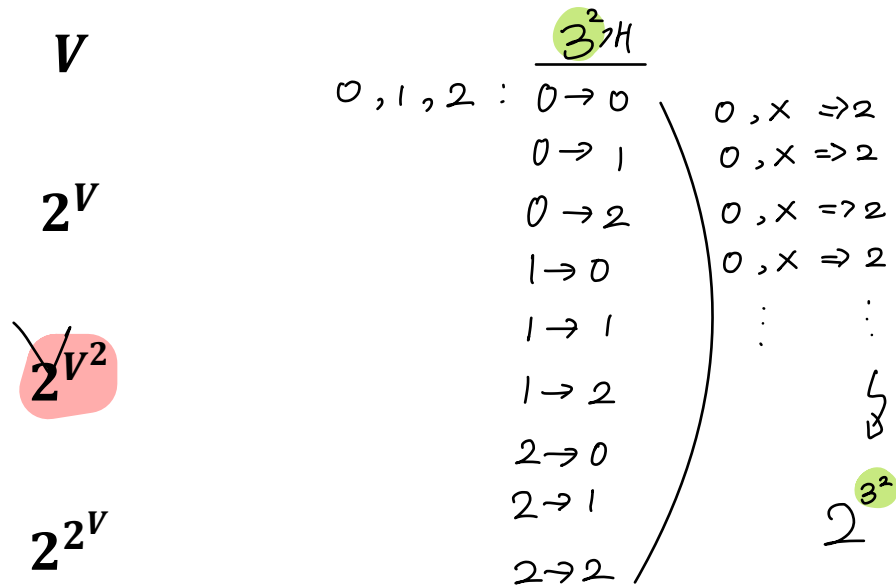
출발지 s ~ 다른 모든 정점까지
최단 경로 찾아주는
프로그램(알고리즘)





[Q] V 개 정점 가진 서로 다른 digraph 개수는?

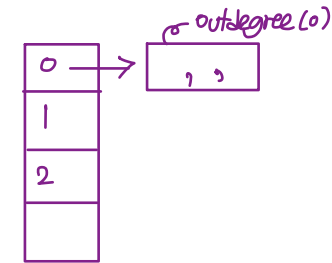
(자기 자신에게 간선 $v \rightarrow v$ 도 가능하나 두 정점 간에 간선은 최대 1개이며 2개 이상인 경우는 없다고 가정)



[Q] Digraph를 adjacency-list 방식으로 표현했다.

정점 v 로부터 ^{outdegree}나가는 모든 간선을 탐색하려면
보기 중 무엇에 비례한 시간이 걸리나?

indegree(v)
~~outdegree(v)~~
 V
 $V+E$



^{어디서 들어오는 간선을 다 찾는다}
[Q] 정점 v 로 들어오는 모든 간선을 탐색하려면
보기 중 무엇에 비례한 시간이 걸리나?

indegree(v)
 outdegree(v)
 V
 ~~$V+E$~~



Undirected and Directed Graphs

Graph의 표현 방법, 탐색 방법 및 이들의 활용도에 대해 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. Digraph(Directed Graph)의 표현(저장) 방법
03. Digraph에 대한 탐색
04. Digraph 탐색의 활용 예: 프로그램 흐름 분석, Garbage Collection, 웹 크롤링
05. Digraph 탐색의 활용 예: Topological Sort
06. Digraph 탐색의 활용 예: Strongly-Connected Component의 탐지
07. 실습: Kosaraju-Sharir 알고리즘 구현

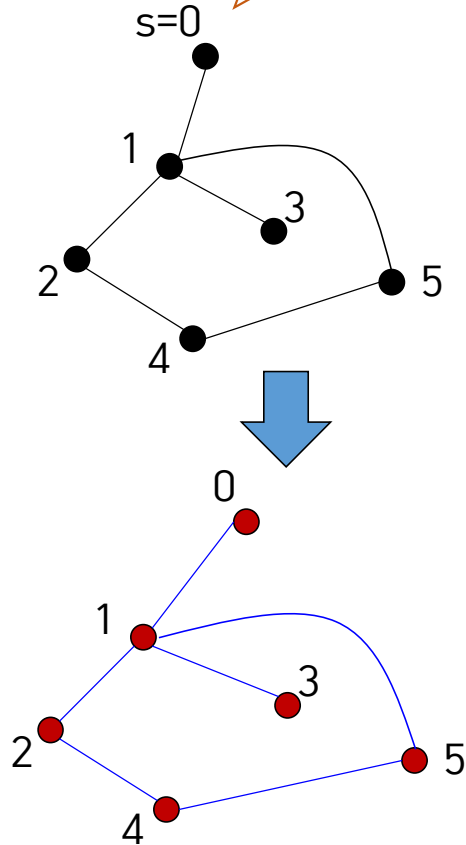


Undirected Graph

모든 간선이 양방향인 Digraph

모든 간선이 양방향이므로
방향 상관 없이 사용하면 사용

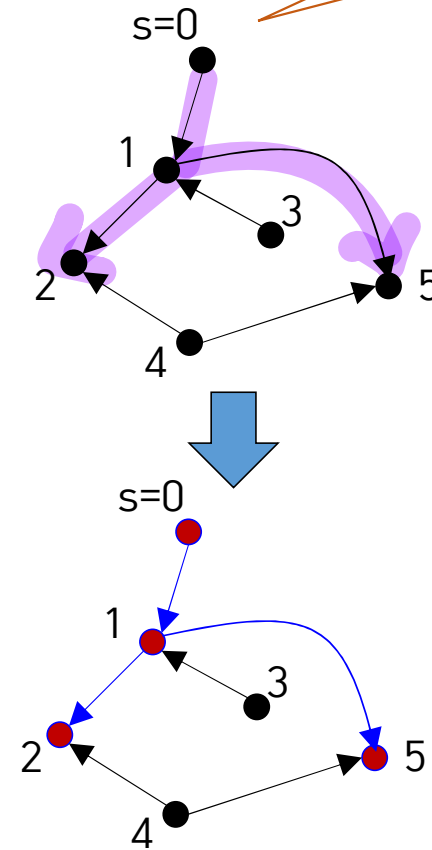
∴ 정답은 2개.



Digraph (Directed Graph)

간선에 방향성 있는 그래프

모든 간선에 방향이 있으므로,
진행 방향에 맞는 간선만 사용





Undirected Graph

모든 간선이 양방향인 Digraph

DFS(to visit vertex v):

v 를 방문한 것으로 표시

v 에 인접한 정점 중 아직 방문 않은 모든 정점 w 를 차례대로 DFS(w) 호출해 재귀적으로 방문

BFS(s):

s 를 queue에 추가하며 방문한 것으로 표시

Queue가 빌 때까지 아래를 반복:

가장 먼저 추가된 정점 v 를 queue에서 pop

v 에 인접한 정점 중 아직 방문 않은 모든 정점을 queue에 추가하며 방문한 것으로 표시

Digraph (Directed Graph)

간선에 방향성 있는 그래프

16

DFS(to visit vertex v):

v 를 방문한 것으로 표시

$v \rightarrow w$ 간선 있는 정점 중 아직 방문 않은 모든 정점 w 를 차례대로 DFS(w) 호출해 재귀적으로 방문

모든 간선에 방향이 있으므로, 진행 방향에 맞는 간선만 사용

BFS(s):

s 를 queue에 추가하며 방문한 것으로 표시

Queue가 빌 때까지 아래를 반복:

가장 먼저 추가된 정점 v 를 queue에서 pop

$v \rightarrow w$ 간선 있는 정점 중 아직 방문 않은 모든 정점을 queue에 추가하며 방문한 것으로 표시

모든 간선에 방향이 있으므로, 진행 방향에 맞는 간선만 사용



Undirected Graph

모든 간선이 양방향인 Digraph

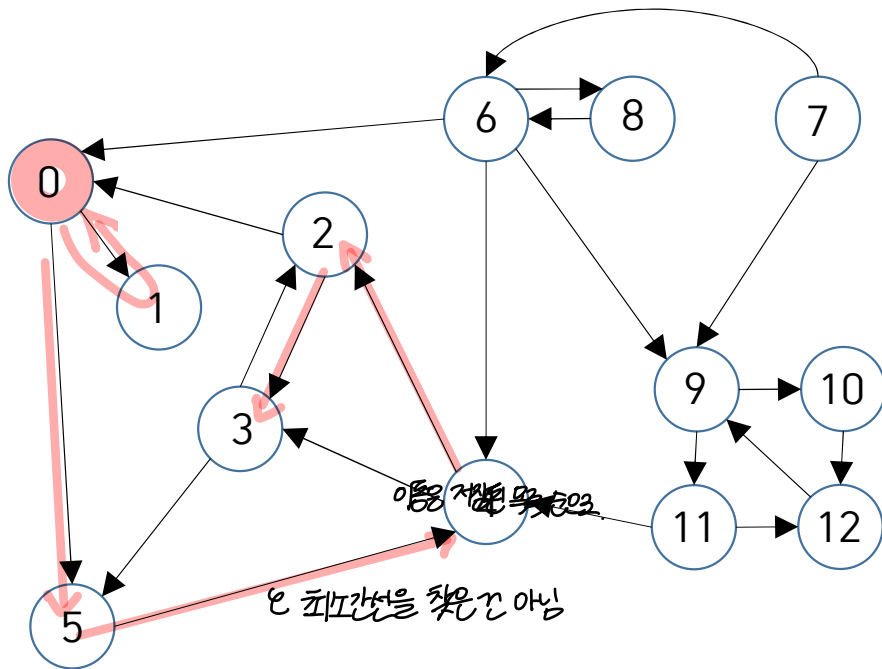
Digraph (Directed Graph)

간선에 방향성 있는 그래프

```
class DFS:
    def __init__(self, g, s):
        def recur(v):
            self.visited[v] = True
            for w in g.adj[v]:
                if not self.visited[w]:
                    recur(w)
                    self.fromVertex[w] = v
        self.g, self.s = g, s
        self.visited = [False for _ in range(g.V)]
        self.fromVertex = [None for _ in range(g.V)]
        recur(s)
```

v에서 adj[v] 목록에 속한 w로 진행하는 것은 같음

BFS도 undirected graph에 대한 코드와 같음



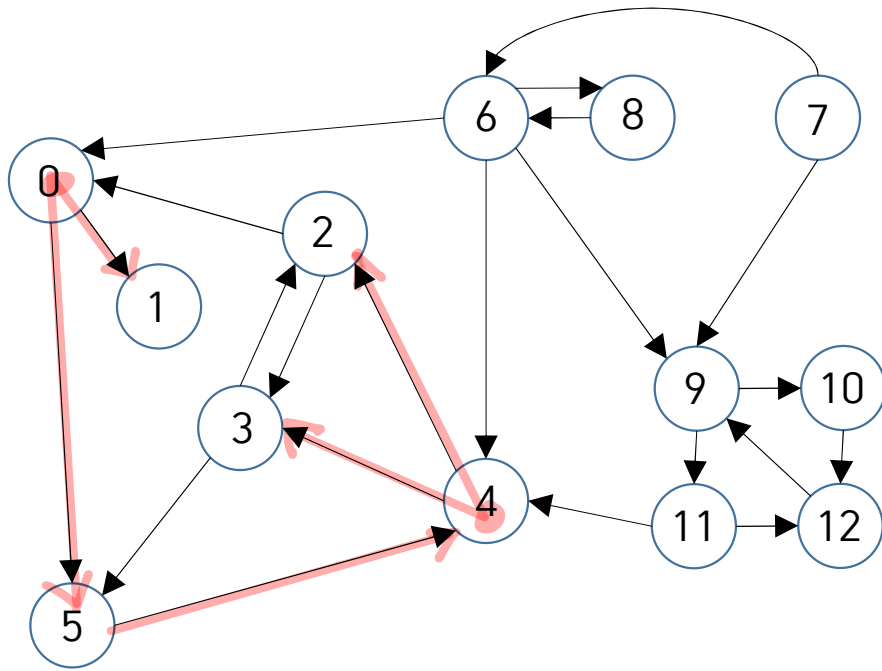
	adj[]
0	→ [1,5]
1	→ []
2	→ [0,3]
3	→ [2,5]
4	→ [2,3]
5	→ [4]
6	→ [0,4,8,9]
7	→ [6,9]
8	→ [6]
9	→ [10,11]
10	→ [12]
11	→ [4,12]
12	→ [9]

[Q] 왼쪽 그래프의 정점 0에서 **DFS**를 시작하면 어떤 경로를 어떤 순서로 찾게 되나?

무조건 최단 경로를 찾은 건 아님

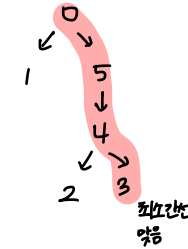


Digraph에서도 BFS는 최소 간선수 경로 찾음



	adj[]
0	•→ [1,5]
1	•→ []
2	•→ [0,3]
3	•→ [2,5]
4	•→ [2,3]
5	•→ [4]
6	•→ [0,4,8,9]
7	•→ [6,9]
8	•→ [6]
9	•→ [10,11]
10	•→ [12]
11	•→ [4,12]
12	•→ [9]

[Q] 왼쪽 그래프의 정점 0에서 **BFS**를 시작하면 어떤 경로를 어떤 순서로 찾게 되나?

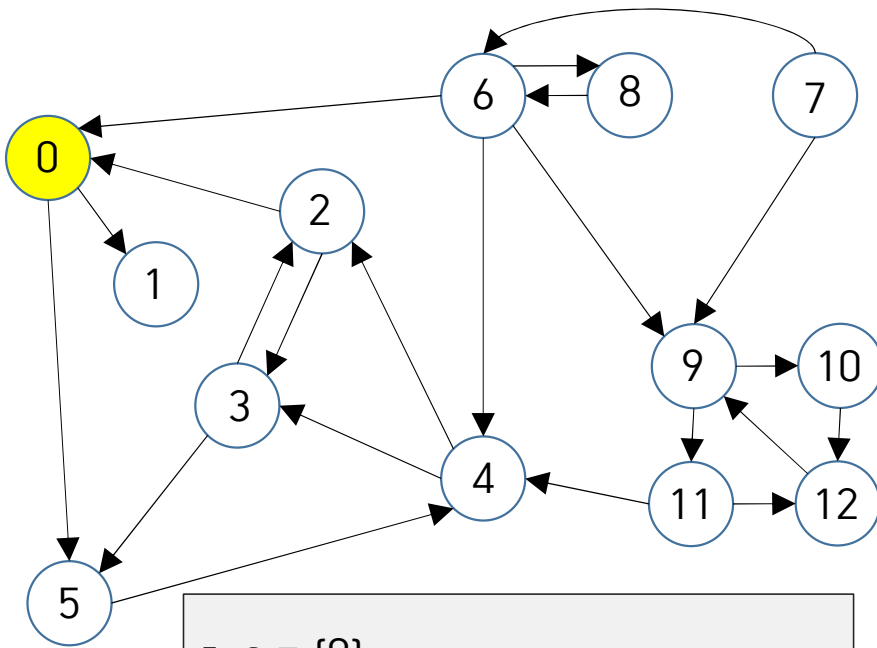


[Q] 정점 0에서 3까지의 경로를 DFS에서 찾은 경로와 비교해 보자. 어느 쪽이 최소 간선수 경로인가? **BFS**



(Single source) BFS

- 한 출발지 s에서 BFS 진행
- s만 queue에 넣고 탐색 시작



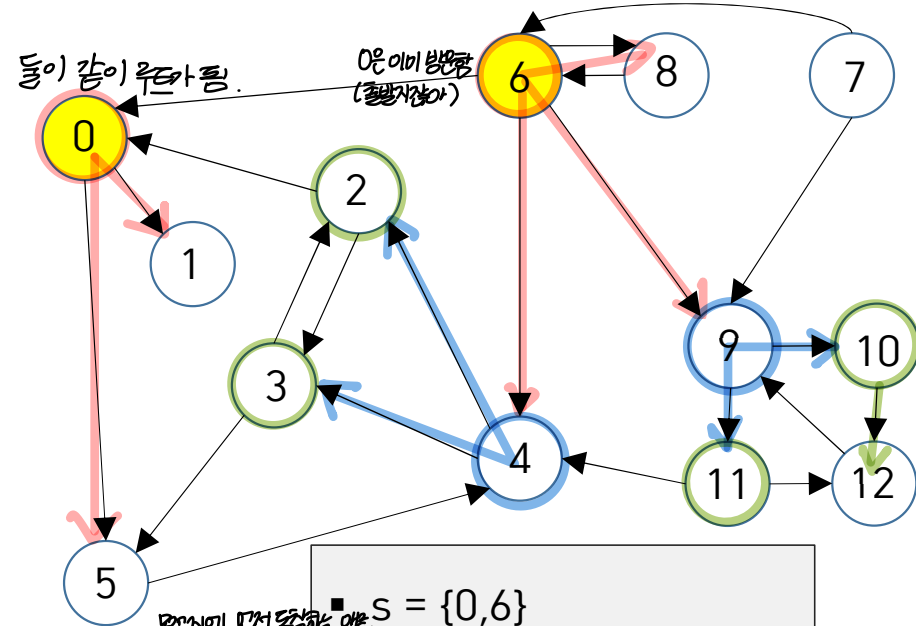
- $s = \{0\}$
- 5까지 최단 경로: $0 \rightarrow 5$
- 4까지 최단 경로: $0 \rightarrow 5 \rightarrow 4$
- 9까지 최단 경로: 존재하지 않음

(출발지 다함 → 여러 출발지)

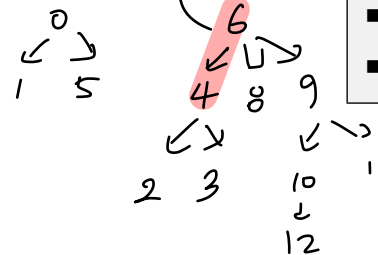
Multiple-source BFS

20

- 여러 출발지에서 BFS 진행 (여러 출발지에서 각각 bfs를 실행)
- 이들 모두를 queue에 넣고 탐색 시작
- 왜 필요한가? ① 정점의 집합에서 목적지까지 최단 경로 찾기 (예: Word Net), ② 같은 시간에 더 다양한 정점과 경로 탐색 등 (예: Web Crawling)



- $s = \{0, 6\}$
- 5까지 최단 경로: $0 \rightarrow 5$
- 4까지 최단 경로: $6 \rightarrow 4$
- 9까지 최단 경로: $6 \rightarrow 9$





(Single source) BFS

```
lass BFS:
```

```
    def __init__(self, g, s):
        assert(isinstance(g, Digraph) and s>=0 and s<g.V)
        self.g, self.s = g, s
        self.visited = [False for _ in range(g.V)]
        self.fromVertex = [None for _ in range(g.V)]
        self.distance = [None for _ in range(g.V)]
        queue = Queue()
        queue.put(s)
        self.visited[s] = True
        self.distance[s] = 0
        while queue.qsize()>0:
            v = queue.get()
            for w in g.adj[v]:
                if not self.visited[w]:
                    queue.put(w)
                    self.visited[w] = True
                    self.fromVertex[w] = v
                    self.distance[w] = self.distance[v] + 1
```

queue에 s 하나만 담고
결과 담을 목록들 초기화한 후
while loop 시작



```
class MultiSourceBFS:
    def __init__(self, g, sList):
        assert(isinstance(g, Digraph) and s >= 0 and s < g.V)
        self.g, self.s = g, s
        self.visited = [False for _ in range(g.V)]
        self.fromVertex = [None for _ in range(g.V)]
        self.distance = [None for _ in range(g.V)]
        queue = Queue()
        for s in sList:
            queue.put(s)
            self.visited[s] = True
            self.distance[s] = 0
        while queue.qsize() > 0:
            v = queue.get()
            for w in g.adj[v]:
                if not self.visited[w]:
                    queue.put(w)
                    self.visited[w] = True
                    self.fromVertex[w] = v
                    self.distance[w] = self.distance[v] + 1
```

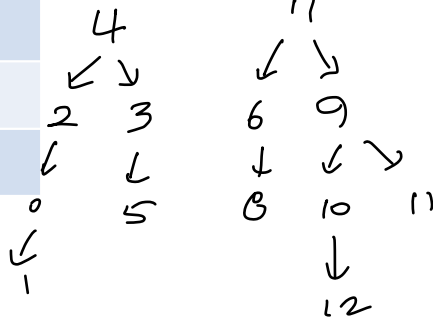
queue에 sList의 모든 정점 담고
while loop 시작



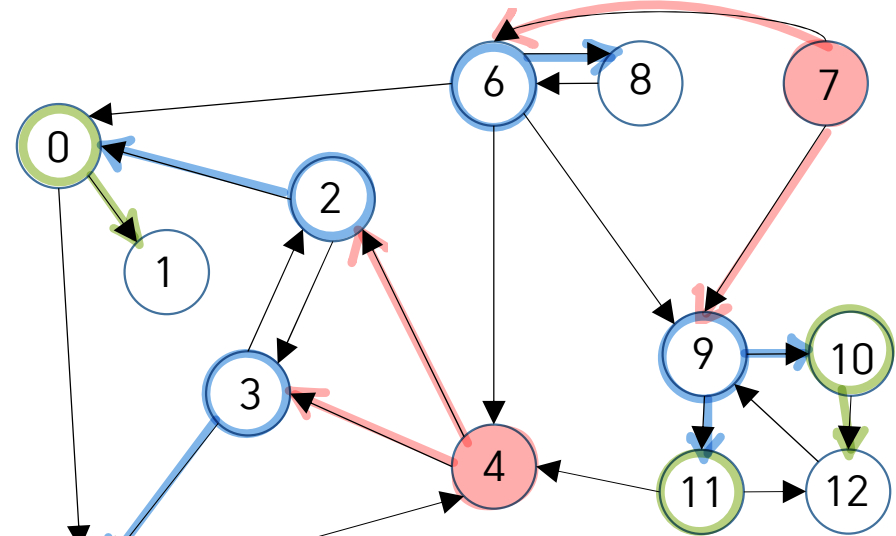
[Q] 다음 그래프에 대해 출발지 $s=\{4,7\}$ 로부터 multi-source BFS를 수행했을 때, 각 목적지에 대해 발견하는 경로를 발견하는 순서대로 써보시오.

※ 여러 정점에 인접할 때는 번호가 작은 정점을 먼저 queue에 넣는다고 가정하시오. 예를 들어 정점 7은 6과 9에 인접한데 [6, 9] 순서로 queue에 넣는다.

목적지	BFS가 발견한 최소 간선 경로
4	4
7	7
2	4→2
3	4→3
6	7→6
9	7→9
0	4→2→0
5	4→3→5
8	7→6→8
10	7→9→10
11	7→9→11
1	4→2→0→1
12	7→9→10→12



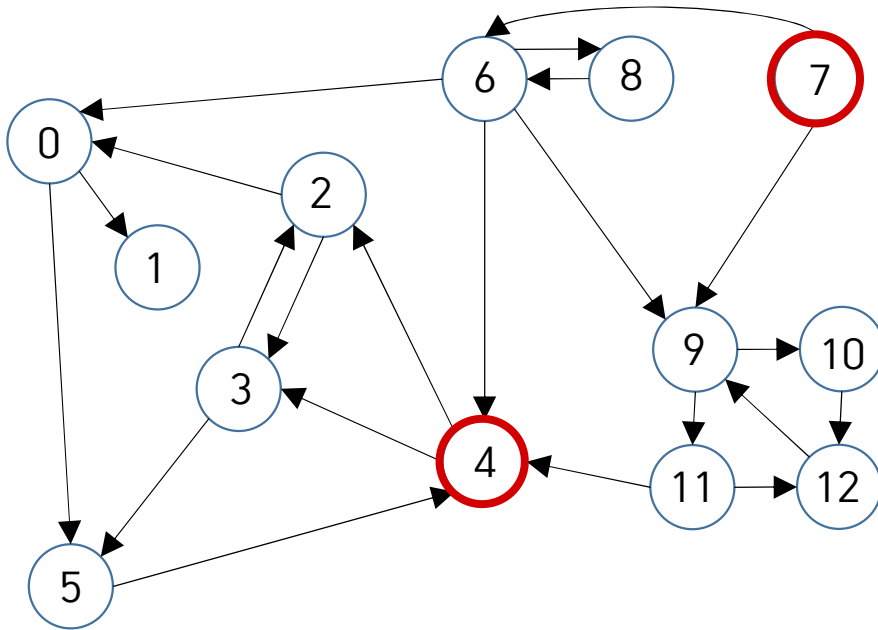
$s=\{4,7\}$ 모두에서 도달 가능한 곳이라면, 더 가까운 쪽이 먼저 찾음





[Q] multi-source BFS가 있다면, multi-source DFS는 없는가? *아니요.*

24



변경
BFS (4, 7)
+
BFS (4)
BFS (7)

변경
DFS (4, 7)
||
DFS (4)
DFS (7)
↳ ∴ *오답*



Undirected and Directed Graphs

Graph의 표현 방법, 탐색 방법 및 이들의 활용도에 대해 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번

02. Digraph(Directed Graph)의 표현

03. Digraph에 대한 탐색

04. Digraph 탐색의 활용 예: 프로그램 흐름 분석, Garbage Collection, 웹 크롤링

05. Digraph 탐색의 활용 예: Topological Sort

06. Digraph 탐색의 활용 예: Strongly-Connected Component의 탐지

07. 실습: Kosaraju-Sharir 알고리즘 구현

목표: Digraph 탐색 기능의 여러 다양한 어플리케이션 보아서, 이후 Graph 관련 기능이 필요할 때 (이번 시간 실습 문제 포함) 문제의 특성에 맞는 적절한 탐색 방법을 잘 적용해 문제 해결하는 능력 기르기

각 탐색 방법의 어떤 특성 때문에 어떤 경우에 활용되는지를 잘 보세요.



파일 1

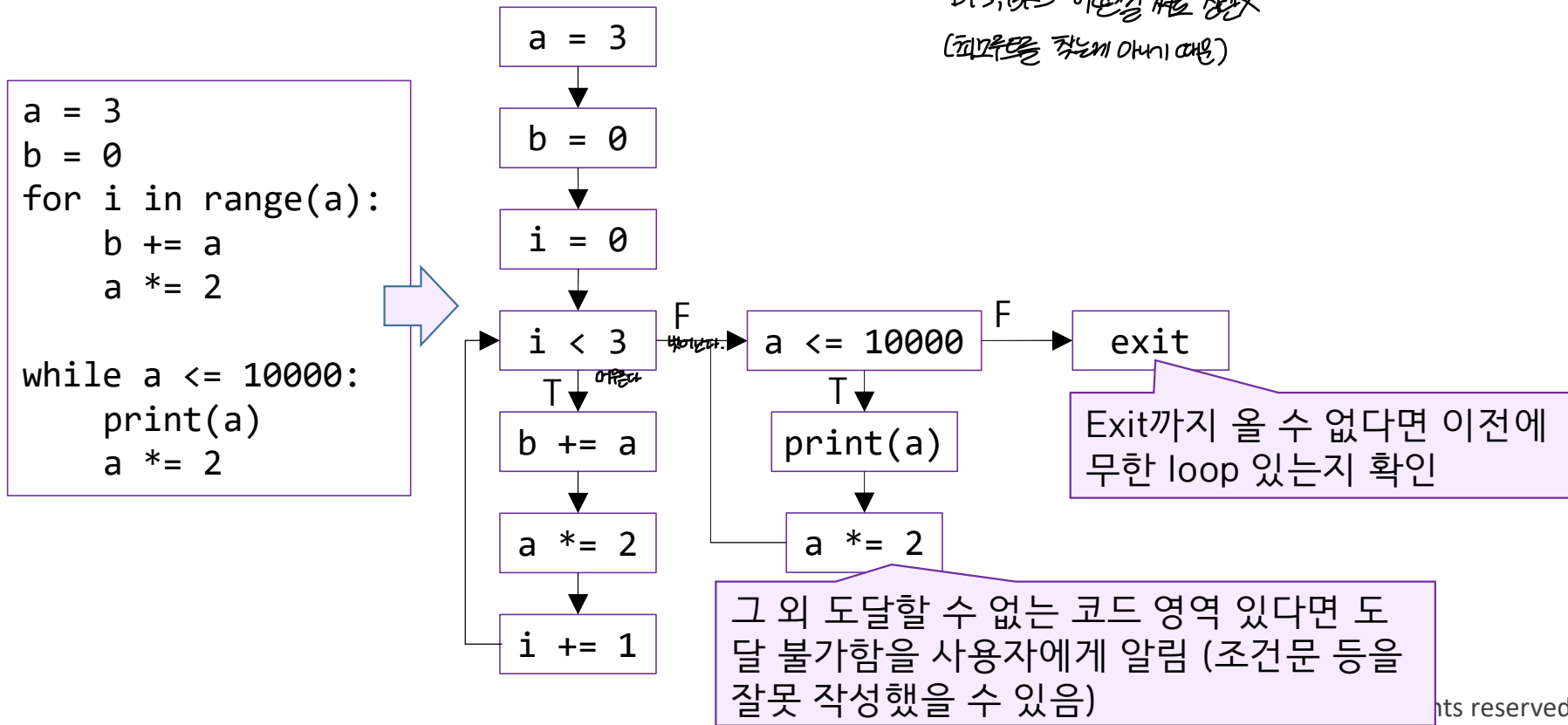
(DFS & BFS) 프로그램의 control-flow 분석해 가능한 오류 탐지

- 모든 프로그램은 digraph로 표현 가능
 - 정점: 명령어
 - 간선 v-w: 실행 순서 (명령어 v 후 w 실행)



- 다양한 오류 탐지해 사용자에게 알림
 - Unreachable code 탐지하고 제거: 시작 명령어로부터 도달 불가능한 명령어
 - 무한 loop 탐지: Exit 가능한지 확인
 - ...

DFS, BFS 어떤걸 사용하든 상관없음
(최소우선 탐색을 찾는게 아니기 때문)





```
a = 3
b = 0
for i in range(a):
    b += a
    a *= 2
```

```
while a <= 10000:
    print(a)
    a *= 2
```

```
while True:
    print(a)
```

```
print(b)
```

대부분의 IDE (Integrated Development Environment, 예: Visual Studio, Eclipse 등)에서 이러한 기능 지원

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Prints the values to a stream, or to sys.stdout by default. Optional arguments: file: a file-like object (stream); defaults to the current sys.stdout. sep: string inserted between values, default a space. end: string appended after the last value, default a newline. flush: whether to forcibly flush the stream.

```
Code is unreachable Pylance
```

```
print(b)
```



Chapter 2

(DFS & BFS) 사용하지 않는 객체 주기적으로 확인해 메모리에서 삭제

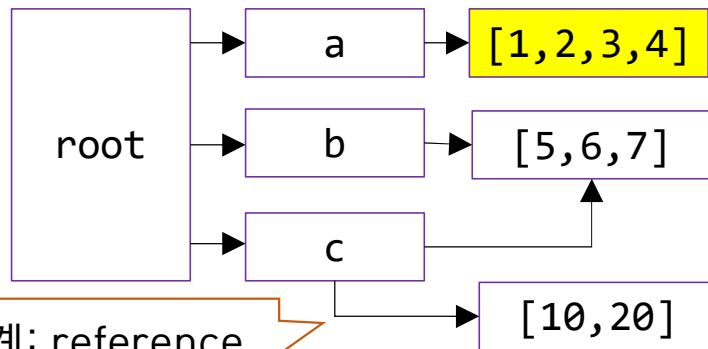
- 동적으로 메모리 할당한 객체들은 digraph로 표현 가능
 - 정점: 객체 (객체가 점유한 메모리)
 - 간선 v-w: 참조 관계 (객체 v가 w를 참조)
 - root에 인접한 정점: 프로그램에서 바로 접근 가능한 변수가 참조하는 객체

- 주기적으로 아래 수행
 - Root에서 도달 가능한 객체 확인
 - Root에서 도달 불가능한 객체는 더는 프로그램에서 사용 불가능. 이러한 객체가 메모리에 쌓이면 메모리가 부족해지며 프로그램 중지됨 (memory leak). 따라서 메모리에서 삭제해 다른 용도로 사용하도록 함 (Garbage Collection)

실행한 코드

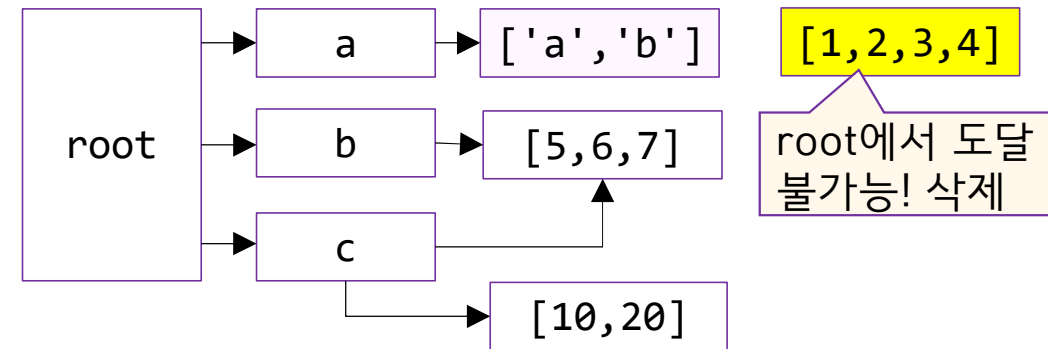
```
a = [1,2,3,4]
b = [5,6,7]
c = [[10,20], b]
```

메모리 상태



참조 관계: reference, pointer 등으로 불림

```
a = ['a', 'b']
```



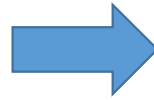


(DFS & BFS) 사용하지 않는 객체 주기적으로 확인해 메모리에서 삭제

- 이러한 기능을 ^{쓰지 않는 객체} Garbage Collection 이라 함
- 많은 프로그래밍 언어의 가상머신에서 이러한 기능 지원 (예: Java, Python 등).
c 계열 언어에서는 사용자가 이를 직접 (코드 작성해) 탐지하고 삭제해야 함

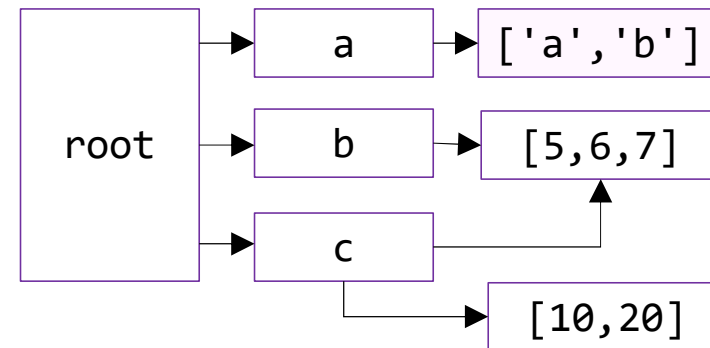
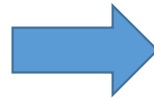
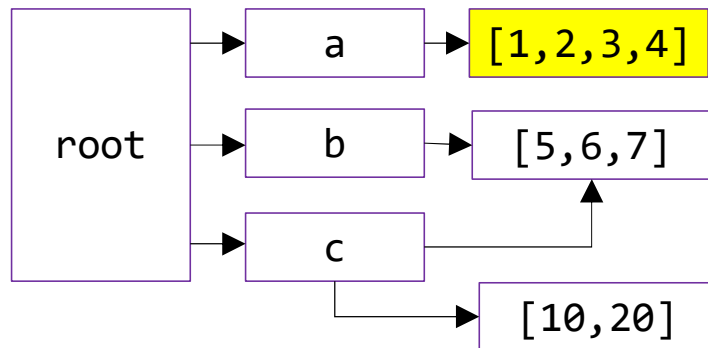
실행한
코드

```
a = [1,2,3,4]
b = [5,6,7]
c = [[10,20], b]
```



```
a = ['a','b']
```

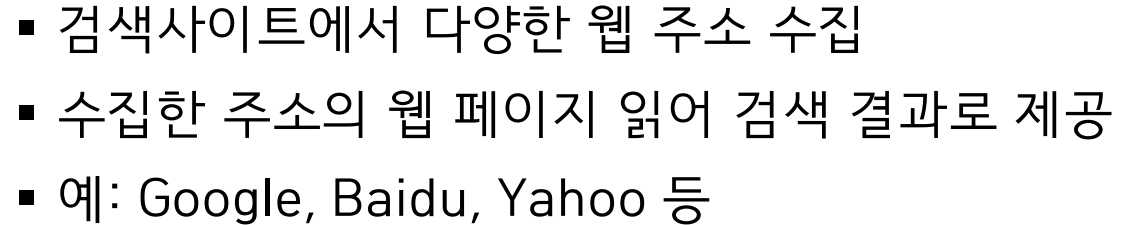
메모리
상태



[1,2,3,4]

root에서 도달
불가능! 삭제

- 출발지 주소 s에서 시작해 링크된 모든 사이트 탐색하는 것 반복



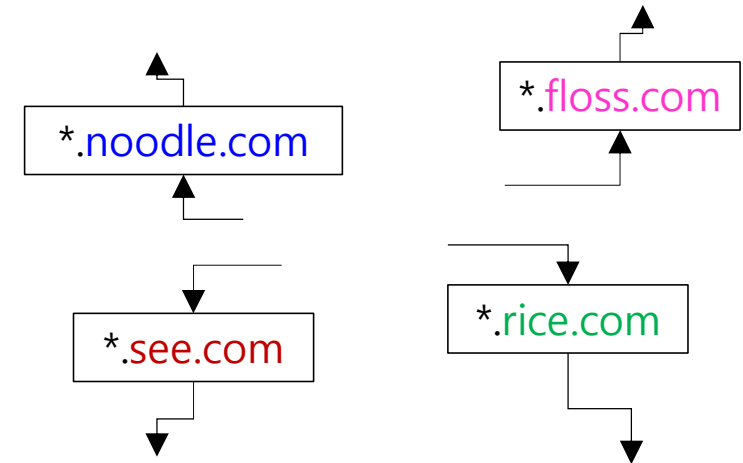
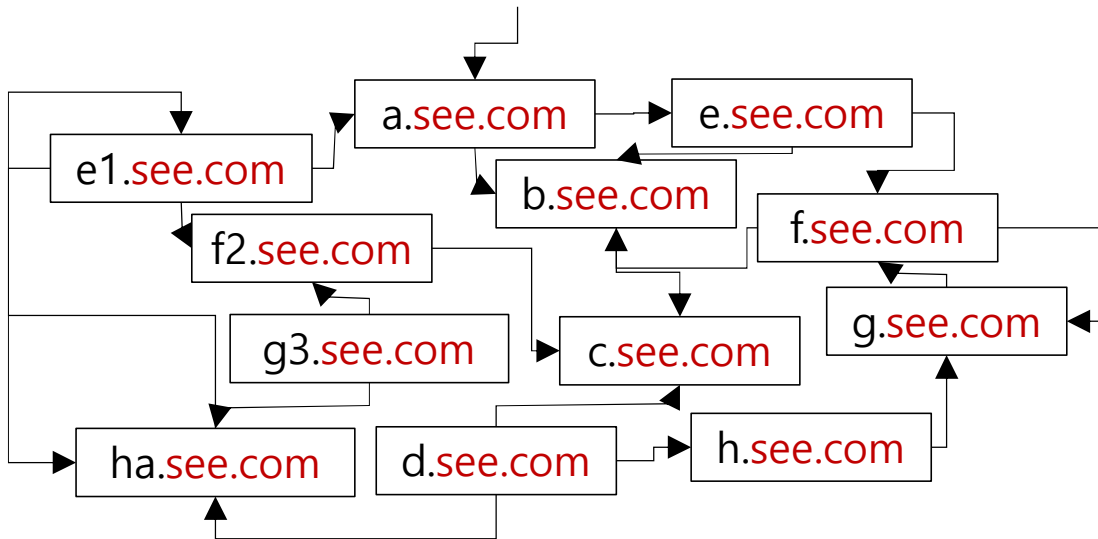


(BFS Preferred) Web Crawling: 웹사이트 찾기 → BFS 사용

31

- DFS 사용 시 내부 주소 많고 상호 참조 많은 도메인 도달한 경우 (예: a.see.com, b.see.com, c.see.com, ...) 새로운 도메인 찾지 못하고 **한 도메인에 오래 빠져 탐색 정체됨**

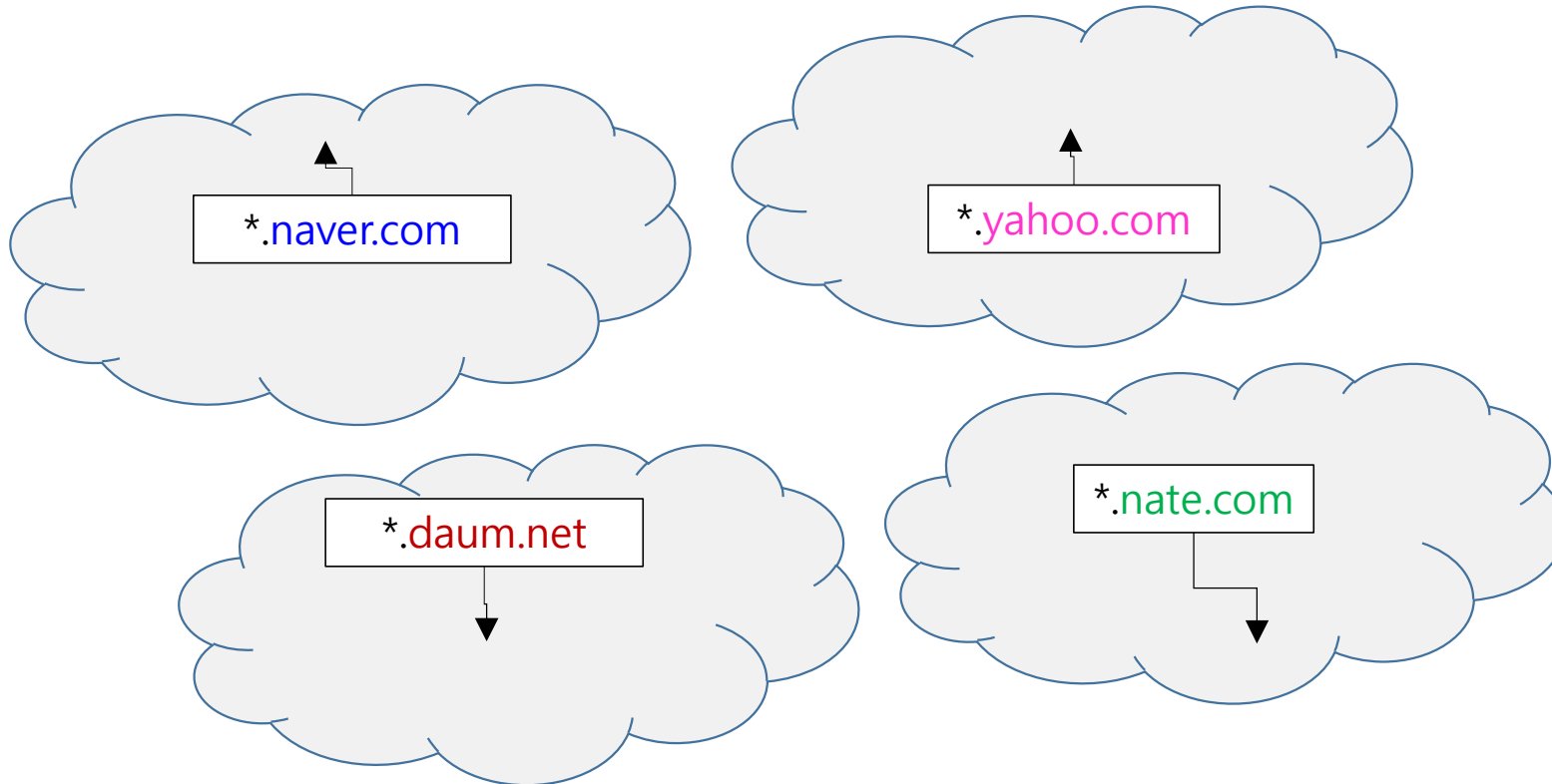
- BFS는 한 branch만 깊이 파지 않고 여러 branch를 돌아가며 탐색하므로
- 같은 시간에 더 다양한 도메인 발견 가능





(Multi-source BFS Preferred) Web Crawling: 웹사이트 찾기

- 여러 source에서 BFS 시작하면 (예: 둘 이상의 popular 웹 사이트)
- 같은 시간에 더 다양한 도메인 발견 가능
- 한 source에서 정체되거나 막다른 길에 다다르더라도 다른 source에서 계속 새로운 주소 탐색 가능





```
webaddrPattern = re.compile("https://(?:\\w+\\.)+(?:\\w+)")
```

```
def webCrawl(roots, maxDepth=1):
```

```
    queue = Queue()
```

```
    discovered = {}
```

```
    for v in roots:
```

```
        queue.put(v)
```

```
        discovered[v] = 0
```

```
    while queue.qsize() > 0:
```

```
        v = queue.get()
```

```
        depth = discovered[v]
```

```
        if depth > maxDepth: break
```

```
    try:
```

```
        resp = requests.get(v)
```

```
        if resp.status_code == 200:
```

```
            print(v, f"(depth={depth})")
```

```
            for w in webaddrPattern.findall(resp.text):
```

```
                if w not in discovered:
```

```
                    discovered[w] = depth + 1
```

```
                    queue.put(w)
```

```
    except requests.exceptions.ConnectionError as error:
```

```
        pass
```

리스트 roots를 출발점의 집합으로 사용해
최대 거리 maxDepth까지 탐색

지금까지 발견한 주소 저장하는 symbol table.
이미 발견한 주소는 다시 탐색 않기 위함

queue를 roots
에 담긴 주소로
초기화

BFS 기본 루틴

웹주소 v 방문해 페이지 소스(html) 받기

방문에 성공했다면

페이지 소스에서 웹주소
패턴 찾기 (https://...)

찾은 웹주소가 아직 방문하지 않
은 주소라면 queue에 넣기



```
# DirectedGraph.py에서 아래 라인 주석 해제해 실행해 보기
webCrawl(["https://www.naver.com/", "https://www.daum.net/"])
=====
...
https://goodtip.co.kr (depth=1)
https://mushroomprincess.tistory.com (depth=1)
https://dazzlehy.tistory.com (depth=1)
https://webtoon.kakao.com (depth=1)
https://game.daum.net (depth=1)
https://poe.game.daum.net (depth=1)
https://pubg.game.daum.net (depth=1)
...
```



Undirected and Directed Graphs

Graph의 표현 방법, 탐색 방법 및 이들의 활용도에 대해 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번

02. Digraph(Directed Graph)의 표현

03. Digraph에 대한 탐색

04. Digraph 탐색의 활용 예: 프로그램 흐름 분석, Garbage Collection, 웹 크롤링

05. Digraph 탐색의 활용 예: Topological Sort (위상 정렬)

06. Digraph 탐색의 활용 예: Strongly-Connected Component의 탐지

07. 실습: Kosaraju-Sharir 알고리즘 구현

목표: Digraph 탐색 기능의 여러 다양한 어플리케이션 보아서, 이후 Graph 관련 기능이 필요할 때 (이번 시간 실습 문제 포함) 문제의 특성에 맞는 적절한 탐색 방법을 잘 적용해 문제 해결하는 능력 기르기

이번 시간 수업 자료에 첨부된 코드를 실행할 수 있도록 준비해 두세요. 이론 수업 중에 실행해 볼 예정입니다.



정점을 정렬할 수 있는가?

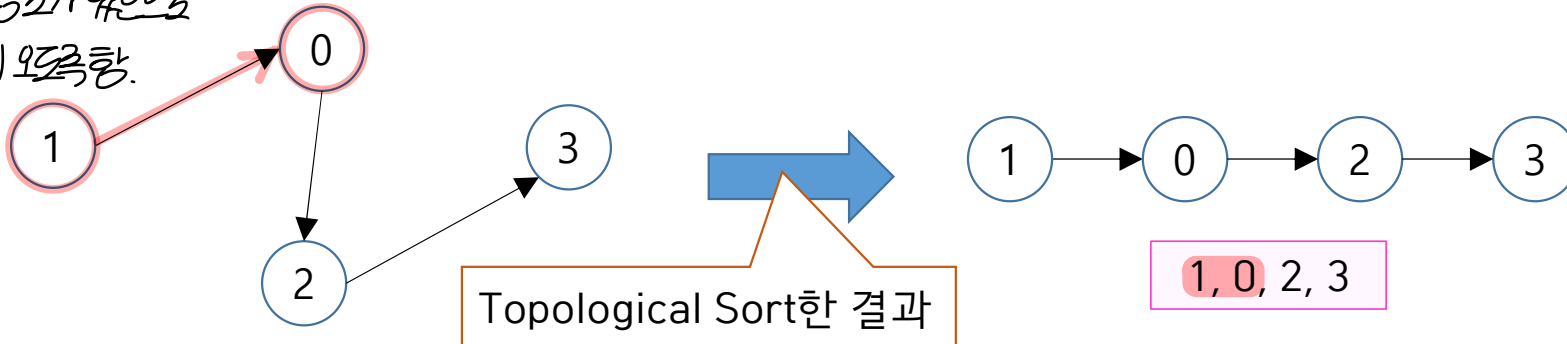
Topological Sort(위상 정렬): 간선 향하는 방향 순으로 정점의 순서 정하기
(간선이 한 방향으로만 향하도록 정점의 순서 정하기)

- Cycle 없는 digraph에서
- Topological order: $v \rightarrow w$ 경로 있다면 v 후에 w 오도록 하는 순서
- Topological order 순으로 정점 나열하는 것을 topological sort라 함

DFS

: 더 내려갈 수 없을 때까지
내려가는 방법

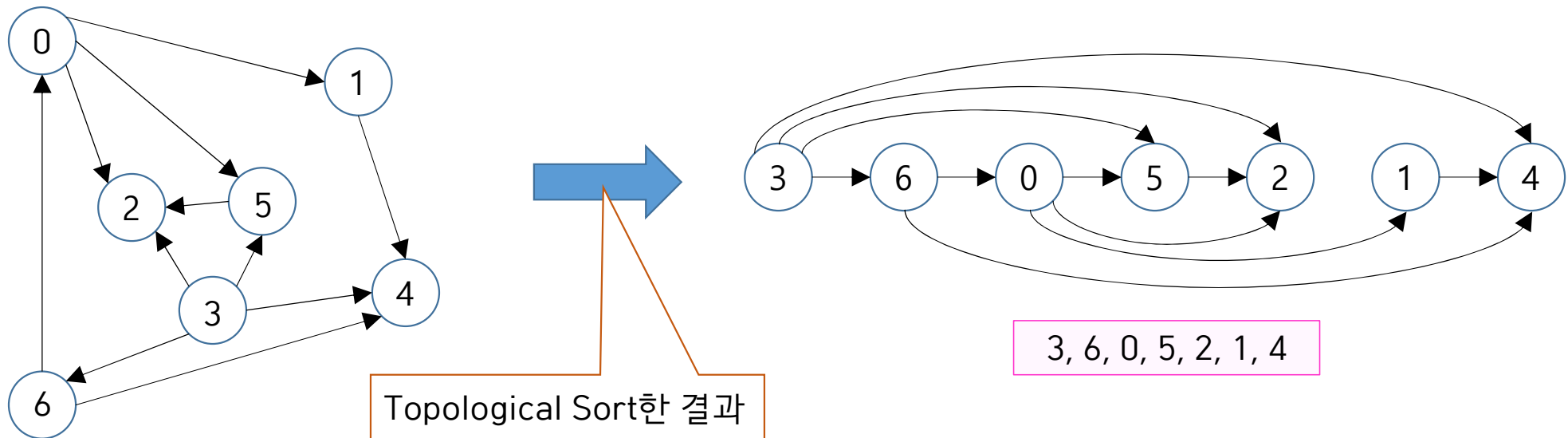
1 → 0의 경로는 없음
(이후에 0이 1으로 향함.)





Topological Sort(위상 정렬): 간선 향하는 방향 순으로 정점의 순서 정하기 (간선이 한 방향으로만 향하도록 정점의 순서 정하기)

- Cycle 없는 digraph에서
- Topological order: $v \rightarrow w$ 경로 있다면 v 후에 w 오도록 하는 순서
- Topological order 순으로 정점 나열하는 것을 topological sort라 함

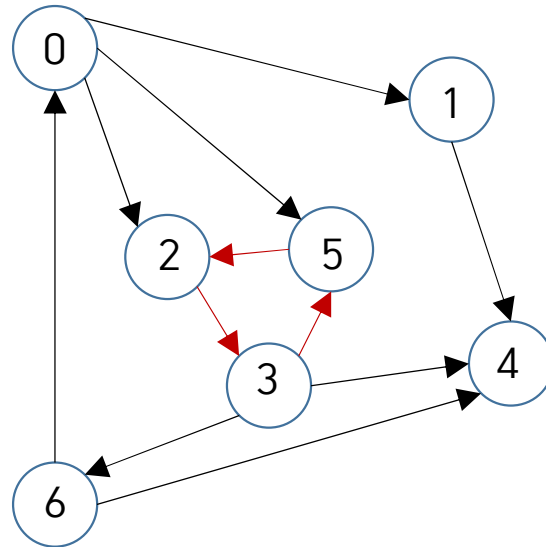




'Cycle 없는' 조건 있는 이유? Cycle 있으면 topological order 존재할 수 없나?

(Cycle이 있으면 topological sort 불가능)

- **Cycle 없는** digraph에서
- Topological order: $v \rightarrow w$ 경로 있다면 v 후에 w 오도록 하는 순서
- Topological order 순으로 정점 나열하는 것을 topological sort라 함



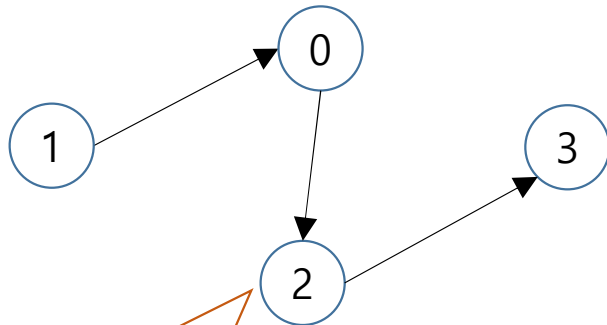
하지만 cycle 있는 그래프에도
topological sort 방법을 적용할 수는 있으며,
그 결과를 활용하는 알고리즘도 있음



Topological Sort, Topological Ordering 이라 부르는 이유?

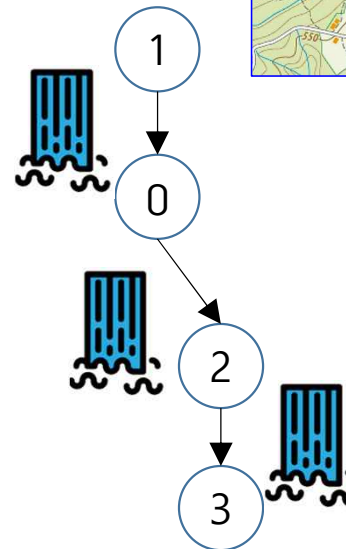
(물 흐르는 방향으로 정렬한 후)

- **Topology**: 연결 상태, 지형
- 정점이 지역, 지대 의미하고
- $v \rightarrow w$ 가 $v \rightarrow w$ 방향으로 물 흐른다는 뜻일 때
- “이들 지역을 높이 순으로 정렬하면?” 에 대한 답

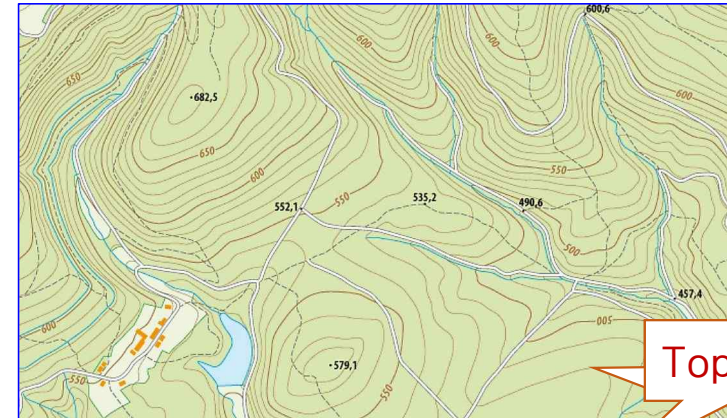


물 흐르는 관계
나타낸 digraph

Topological Sort한 결과



1, 0, 2, 3



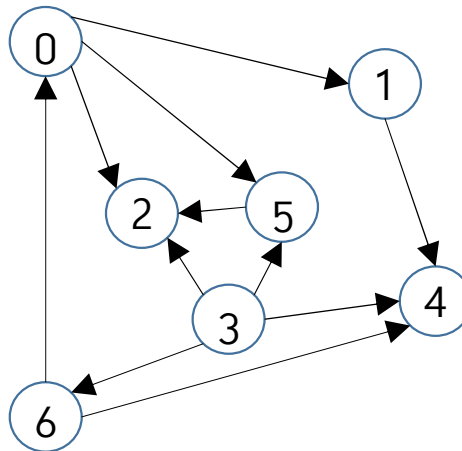
Topology





Topological Sort는 어디에 활용하나?

- Task (Job) Scheduling: 여러 작업 간 **선후관계** 있을 때, 이를 **따르도록 수행하는 순서 정하기**
- Digraph 내 cycle 탐지
- Strongly-connected component 찾기
- Cycle 없는 digraph에서 최단경로 찾기
- ...



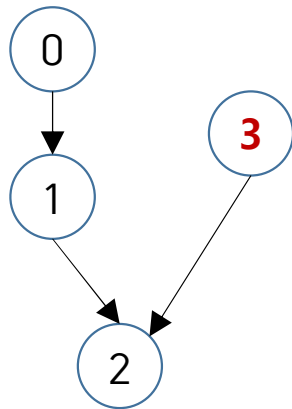
수업 간 선후관계가 왼쪽 그래프와 같을 때 수강할 순서 정하기

0. Algorithms
1. Complexity Theory
2. Artificial Intelligence
3. Intro. to CSE
4. Cryptography
5. Scientific Computing
6. Advanced Programming



같은 그래프에 대해 여러 topological order 존재 가능

41

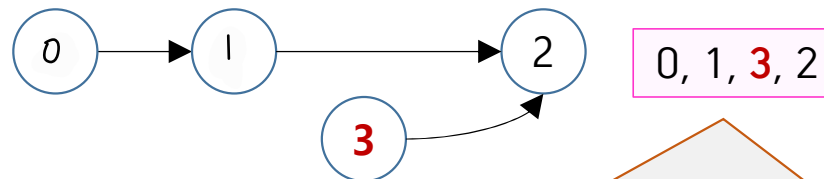
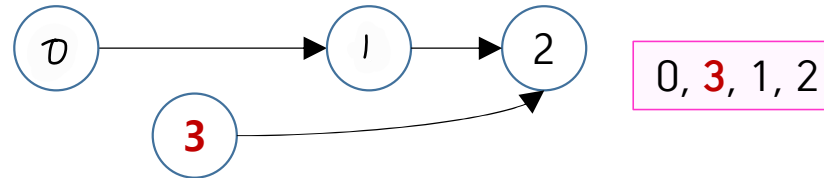
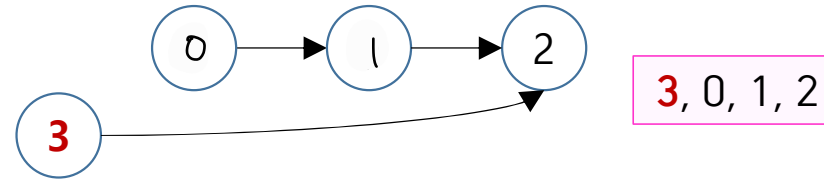


Topological Sort한 결과

(정답은 여러 개 나올 수 있음)

예) 0, 1, 3, 2

ex) 1, 3

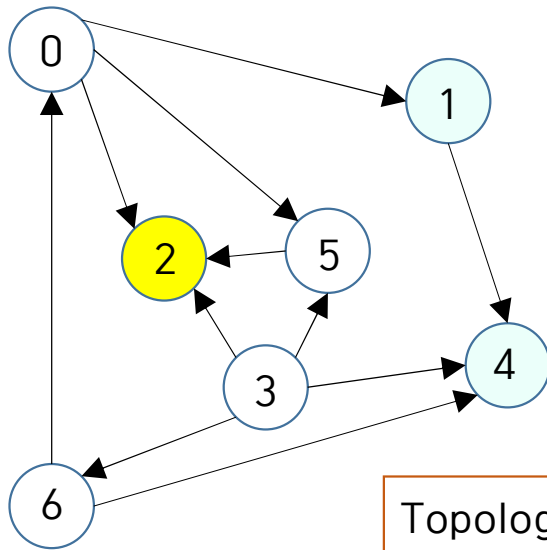


[Q] 세 가지 순서 모두 $0 \rightarrow 1 \rightarrow 2$ 와 $3 \rightarrow 2$ 순서 만족함을 확인하시오.



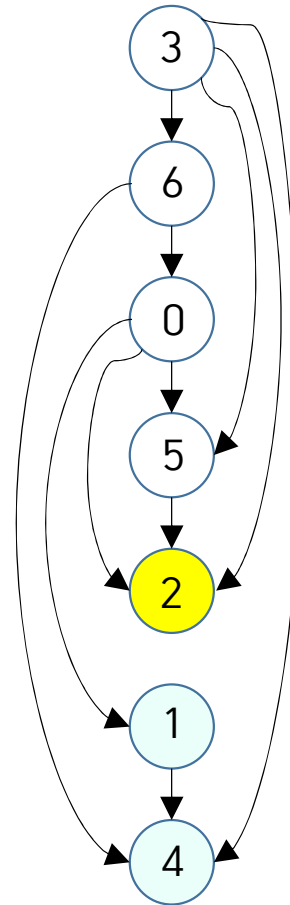
같은 그래프에 대해 여러 topological order 존재 가능

42

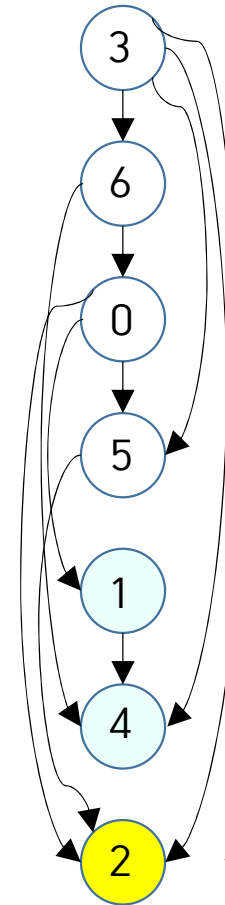


Topological Sort한 결과

[Q] 두 가지 순서 모두
topological order 정의 만족함 확인하시오.



3, 6, 0, 5, 2, 1, 4



3, 6, 0, 5, 1, 4, 2

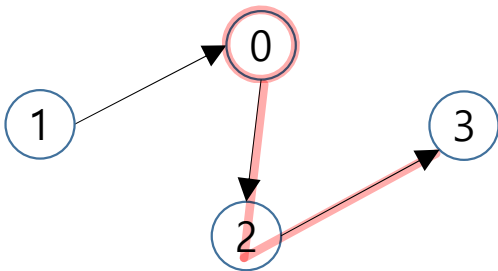
[Q] 1과 4의 순서
는 바뀌어도 되나?

[Q] 2와 4의 순서
는 바뀌어도 되나?



Topological Order는 어떻게 얻을까? DFS!

- 각 정점을 시작점으로 DFS 수행해
- 더는 방문할 곳 없는 정점을 역순으로 기록하면 topological order



$0 \rightarrow 2 \rightarrow 3$
 \therefore 0 2 3

result = [] # 결과 저장할 리스트 초기화

Digraph의 각 정점 v에 대해:

아직 v를 방문하지 않았다면 DFS(v)

DFS(to visit vertex v):

v를 방문한 것으로 표시

v → w 간선 있는 정점 중 아직 방문 않은 모든 정점 w를 차례대로 DFS(w) 호출해 재귀적으로 방문

(v로부터 더는 방문할 곳 없으므로)

result.prepend(v) # v를 목록 앞에 추가

DFS 시작하는 정점	result 리스트
DFS(0)	3rd 2nd 1st 0 2 3
DFS(1)	1 0 2 3
DFS(2) ×	
DFS(3) ×	

이미 방문했기 때문

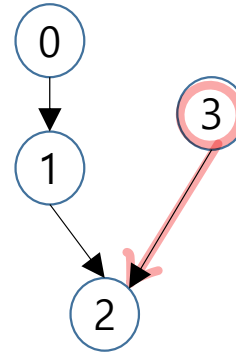
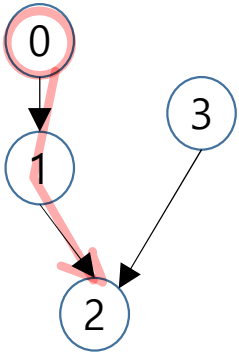
→ 모든 정점 방문 후

→ topological result임



Topological Order는 어떻게 얻을까? DFS! (답여2개 내놓도)

- 각 정점을 시작점으로 DFS 수행해
- 더는 방문할 곳 없는 정점을 역순으로 기록하면 topological order



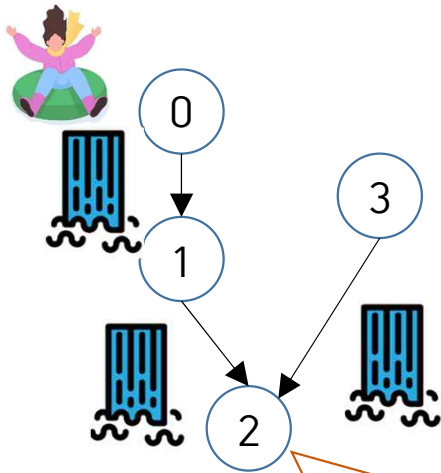
DFS 시작 정점	result 리스트
DFS(0)	0 1 2
DFS(1) x	
DFS(2) x	
DFS(3)	3 0 1 2

DFS 시작 정점	result 리스트
DFS(3)	3 2
DFS(2) x	
DFS(1)	1 3 2
DFS(0)	0 1 3 2

DFS하는 순서에 따라
여러 다른 topological order 나올 수 있으나
모두 0→1→2와 3→2 순서 만족



왜 DFS 해서 더는 방문할 곳 없는 점 역순으로 기록하면 topological order 얻나?



어느 정점에서든 시작해

더 내려갈 수 없는 곳까지 계속 내려가 보는 방법임
더 내려갈 수 없어지는 곳은 역순으로 기록하고
아직 방문하지 않은 정점에 대해 위 과정 반복하면
 $v \rightarrow \dots \rightarrow w$ 선후관계 있다면
 w 가 먼저 기록 되어야만, v 가 기록될 수 있으므로
반드시 w 가 v 보다 리스트 뒤에 오게 됨

result = [] # 결과 저장할 리스트 초기화

Digraph의 각 정점 v 에 대해:

아직 v 를 방문하지 않았다면 DFS(v)

DFS(to visit vertex v):

v 를 방문한 것으로 표시

$v \rightarrow w$ 간선 있는 정점 중 아직 방문 않은 모든 정점 w 를
차례대로 DFS(w) 호출해 재귀적으로 방문

(v 로부터 더는 방문할 곳 없으므로)

result.prepend(v) # v 를 목록 앞에 추가

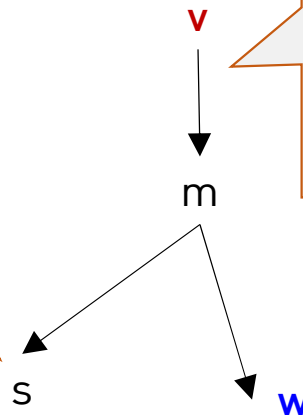


왜 DFS 해서 더는 방문할 곳 없는 점 순으로 기록하면 topological order 역순 얻나?

46

- $v \rightarrow \dots \rightarrow m \rightarrow \dots \rightarrow w$ 와 같은 선후관계 있다면
- DFS를 w, m, v 중 어느 점에서 시작했더라도
- w 가 v 보다 먼저 (더는 방문할 곳 없어져) 목록에 기록되어 선후관계 지킴

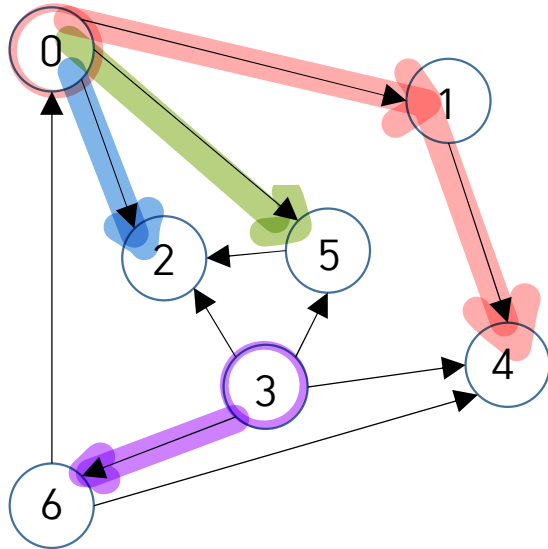
s쪽 가지 먼저 탐색해 s가 w보다 목록에 먼저 기록되어도 상관 없음. s와 w 간에는 선후관계 없으며, v와 w간 선후관계는 지키기 때문



[Q] w, m, v 각각에서 DFS를 시작했을 때 어디를 어떤 순서로 방문하게 될지 생각해 보고, 항상 w 가 v 보다 먼저 목록에 기록됨을 확인해 보시오.



[Q] 번호 순서에 따라 정점 방문해 topological order 얻어보기



DFS 시작 정점	result 리스트
DFS(0)	0 5 2 1 4
① DFS(1) x	
DFS(2) x	
DFS(3)	3 6 0 5 2 1 4
DFS(4) x	
DFS(5) x	
DFS(6) x	

: 방문 순서대로



Digraph g의 정점을 topological order 순으로
나열한 목록을 반환하는 함수

```
140 def topologicalSort(g):
```

```
141     def recur(v):
```

DFS로 v 방문하는 함수

```
142     ② visited[v] = True
```

```
143         for w in g.adj[v]:
```

```
144             if not visited[w]: recur(w)
```

```
145         reverseList.append(v)
```

v로부터 더는 방문할 곳 없을 때 v를 목록에 추가

```
146
```

```
147     assert(isinstance(g, Digraph))
```

함수 호출하면 이 라인부터 시작. g가 Digraph 객체임 확인

```
148     visited = [False for _ in range(g.V)]
```

정점의 방문 여부 기록하는 리스트 초기화

```
149     reverseList = []
```

Topological order 기록하는 리스트 초기화

```
150     ① for v in range(g.V):
```

```
151         if not visited[v]: recur(v)
```

아직 방문하지 않은 정점 v를
시작점으로 DFS 수행

```
152
```

```
153     reverseList.reverse()
```

```
154     return reverseList
```

<공부>

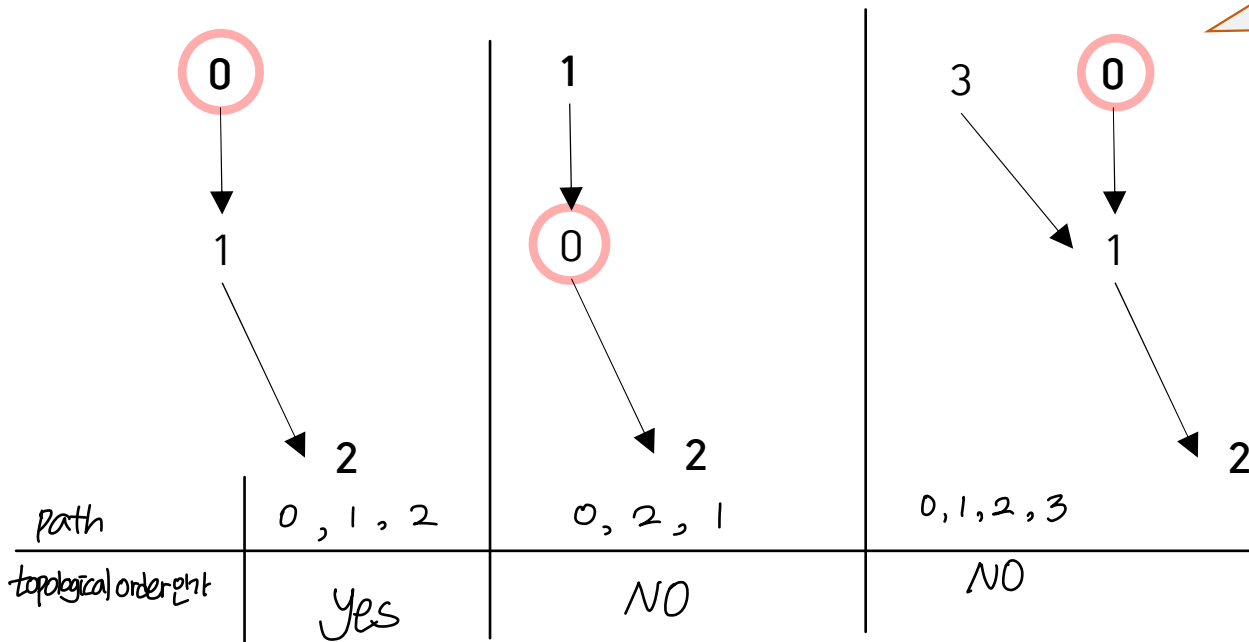
#왜 DFS를 쓸때 역으로 불러 (그냥 순서대로 보면 안될까)

#왜 DFS를 안쓸까



왜 정점 v로부터 더는 방문할 곳이 없어질 때 역순으로 기록하나?
DFS 해서 방문하는 순으로 기록한다면 topological order 얻을 수 없나?

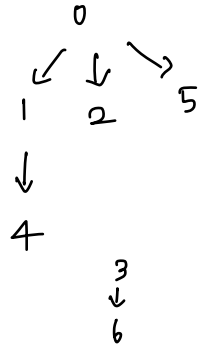
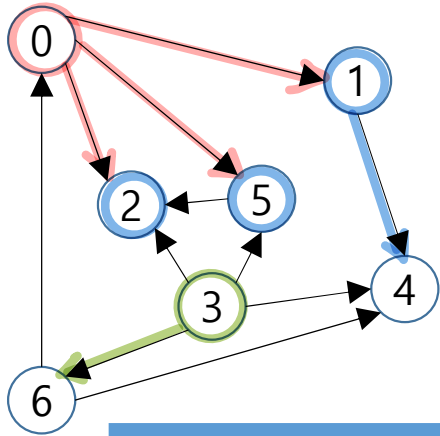
- DFS를 어느 점에서 시작하는가에 따라 방문 순서는 선후관계와 다를 수 있음



[Q] 번호 순으로 DFS를 한다면 어떤 순서로 방문하게 될지 생각해 보고, 항상 topological order대로 방문하지는 않음을 확인해 보시오.



[Q] DFS 대신 **BFS**를 사용해서 마지막에 방문한 정점부터 역순으로 기록하면 역시 topological order 얻을 수 있을까? 아래 그래프에 BFS를 사용해 보자.



result = [] # 정점 저장할 목록 초기화

Digraph의 각 정점 v에 대해:

아직 v를 방문하지 않았다면 **BFS(v)**

BFS(s):

s로부터 depth 순으로 방문하되

마지막에 방문한 정점부터 역순으로 result에 기록

BFS 시작 정점	result 리스트
BFS(0)	0 1 2 5 4
BFS(1) ×	
BFS(2) ×	
BFS(3)	3 6 0 1 2 5 4
BFS(4) ×	
BFS(5) ×	
BFS(6) ×	

← 위치가 반전되어야 함

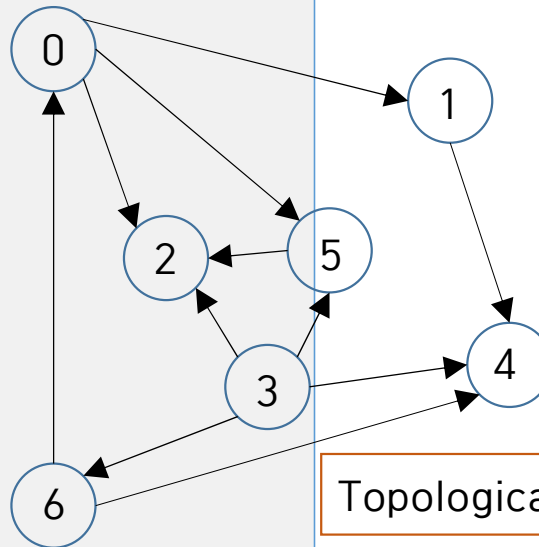
→ Depth만큼 topological order가 잡히지 않음

같은 Depth에서는 안 지켜지니 때문

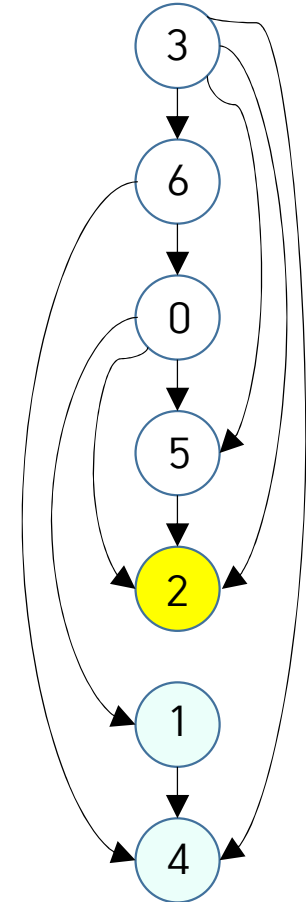


DirectedGraph.py에서 아래 라인 실행해 보기

```
tasks = Digraph(7)
tasks.addEdge(0,1)
tasks.addEdge(0,2)
tasks.addEdge(0,5)
tasks.addEdge(1,4)
tasks.addEdge(3,2)
tasks.addEdge(3,4)
tasks.addEdge(3,5)
tasks.addEdge(5,2)
tasks.addEdge(6,0)
tasks.addEdge(6,4)
print(topologicalSort(tasks))
```

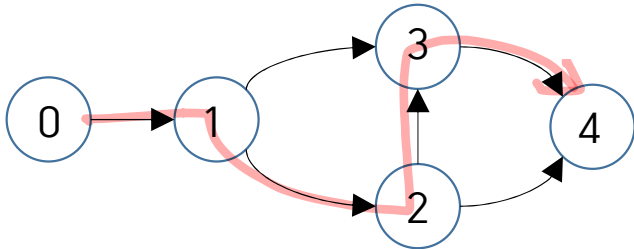


Topological Sort한 결과





[Q] 다음 그래프의 객체를 만들고 topological order를 출력해 보시오. 출력한 결과가 topological order의 정의에 맞는지 그림과 대비해 확인해 보시오.



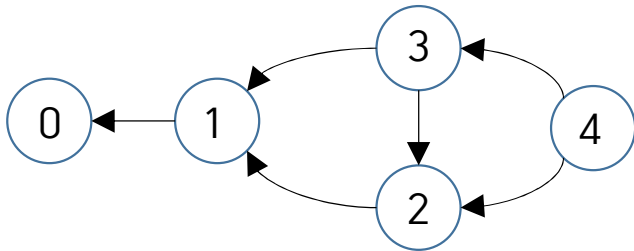
```
g5 = Digraph(5)
g5.addEdge(0,1)
g5.addEdge(1,3)
g5.addEdge(1,2)
g5.addEdge(2,3)
g5.addEdge(3,4)
g5.addEdge(2,4)
print("g5, topological order", topologicalSort(g5))
```



[Q] 다음 그래프의 객체를 만들고 topological order를 출력해 보시오. 출력한 결과가 topological order의 정의에 맞는지 그림과 대비해 확인해 보시오.

이 그래프는 앞 페이지 그래프의 **reverse 그래프**이므로 앞 페이지 그래프 객체가 **g5**라면 (새로운 그래프 객체를 만들고 간선을 더할 필요 없이) 다음과 같이 reverse graph를 얻을 수 있음:

```
gr = g5.reverse()
```





[Q] Topological Sort의 수행 시간은 다음 중 무엇에 비례하나? DFS

V

E

$V+E$

$V \times (V+E)$

DFS



Undirected and Directed Graphs

Graph의 표현 방법, 탐색 방법 및 이들의 활용도에 대해 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번

02. Digraph(Directed Graph)의 표현

03. Digraph에 대한 탐색

04. Digraph 탐색의 활용 예: 프로그램 흐름 분석, Garbage Collection, 웹 크롤링

05. Digraph 탐색의 활용 예: Topological Sort

06. Digraph 탐색의 활용 예: ^{scc} Strongly-Connected Component의 탐지

07. 실습: Kosaraju-Sharir 알고리즘 구현

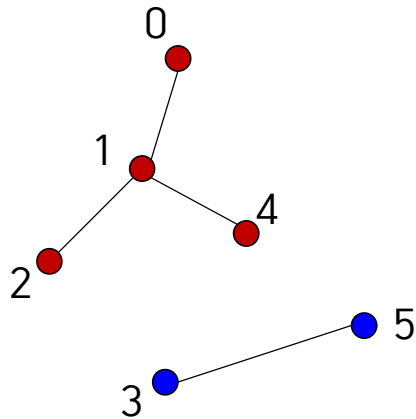
목표: Digraph 탐색 기능의 여러 다양한 어플리케이션 보아서, 이후 Graph 관련 기능이 필요할 때 (이번 시간 실습 문제 포함) 문제의 특성에 맞는 적절한 탐색 방법을 잘 적용해 문제 해결하는 능력 기르기



Undirected Graph

모든 간선이 양방향인 Digraph

- connected(v, w): $v-w$ 경로 존재
- connected component: 서로 간에 모두 연결된 (도달 가능한) 정점의 최대 집합

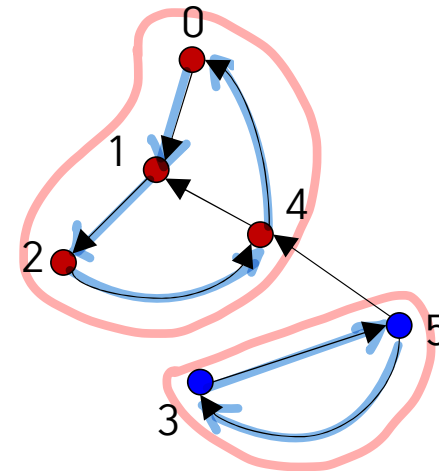


Digraph (Directed Graph)

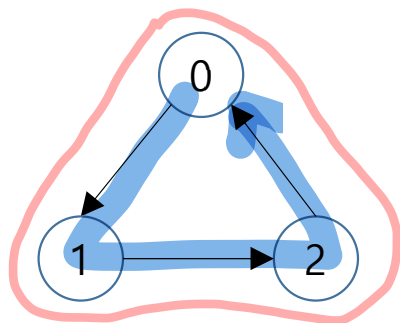
간선에 방향성 있는 그래프

56

- strongly-connected(v, w): $v \rightarrow w$, $w \rightarrow v$ 경로 모두 존재
- Strongly-Connected Component (SCC): 서로 간에 모두 strongly-connected인 (양방향으로 도달 가능한, 즉 갔다가 돌아올 수 있는) 정점의 최대 집합

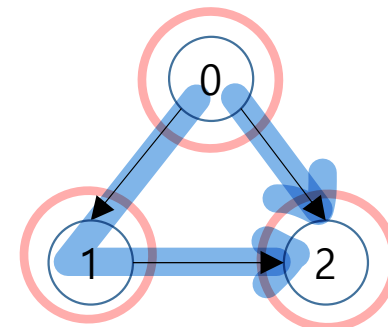
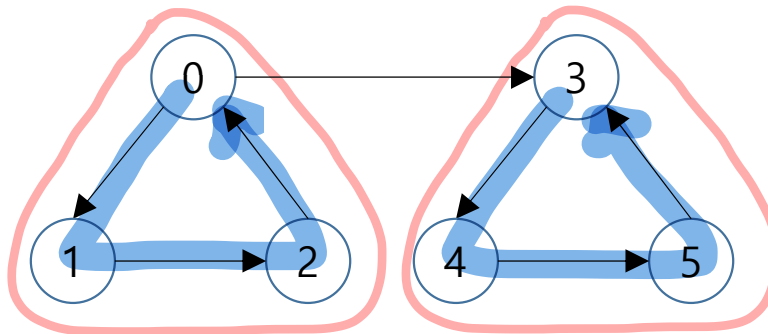
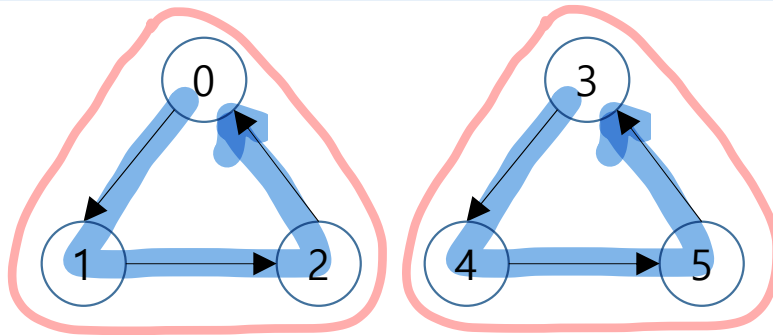


[Q] 아래 그래프에서 같은 SCC에 속하는 정점끼리 묶어 보시오.



$0 \rightarrow 1$ $1 \rightarrow 0$
 $0 \rightarrow 2$ $2 \rightarrow 0$
 $1 \rightarrow 2$ $2 \rightarrow 1$


\therefore 같은 컴포넌트



$0 \rightarrow 1$
 $0 \rightarrow 2$
 $1 \rightarrow 2$

- strongly-connected(v, w): $v \rightarrow w$, $w \rightarrow v$ 경로 모두 존재
- Strongly-Connected Component (SCC): 서로 간에 모두 strongly-connected인 (양방향으로 도달 가능한, 즉 갔다가 돌아올 수 있는) 정점의 최대 집합

[Q] 앞 문제의 해에서 SCC는 다음 특성 만족함을 확인하시오.

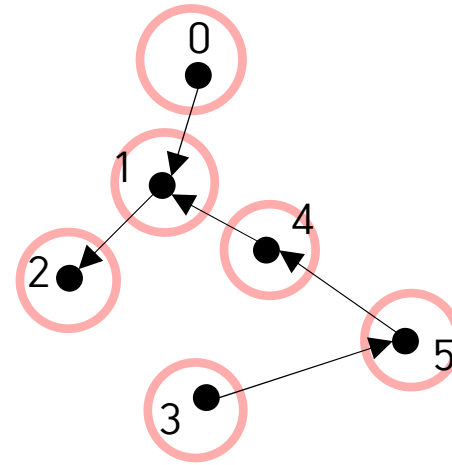
- (1) 같은 SCC 내부에는 반드시 cycle 존재 (즉 양방향 경로 존재) 
- (2) 서로 다른 SCC 간에는 (i) 간선이 없거나 (ii) 간선이 한 방향으로만 존재. 즉 서로 다른 SCC 간에는 양방향으로 간선이 존재하지 않음 (cycle 없음)



[Q] Cycle 없는 digraph에는 몇 개의 SCC(Strongly-Connected Component)가 있나?

→ 정점개수(V)만큼의 SCC 有

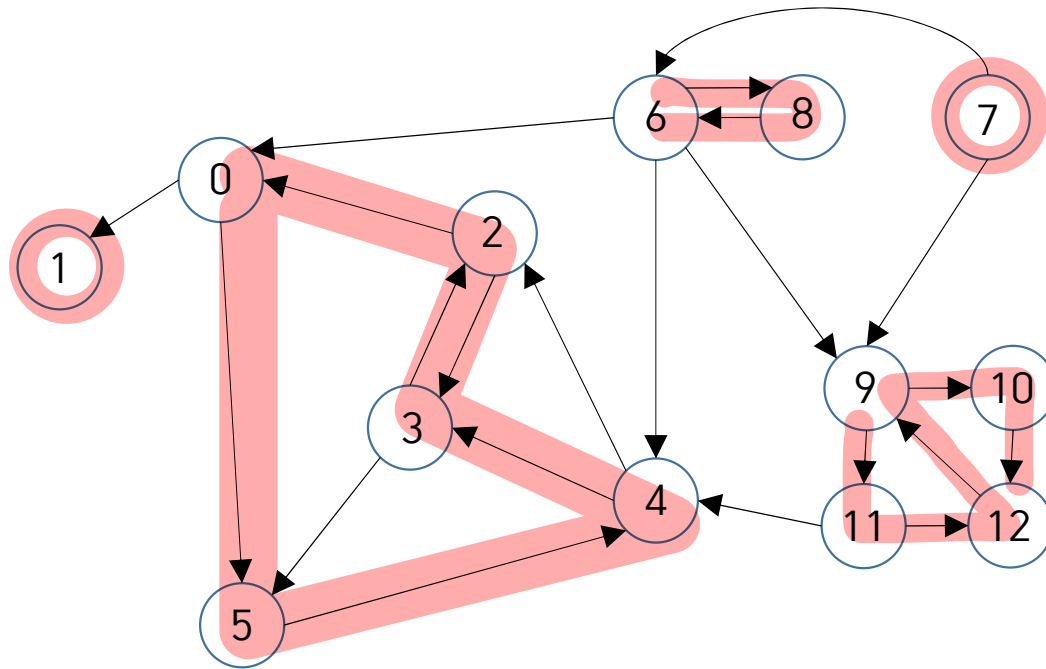
- 0
- 1
- V
- E



<cycle 없는 digraph 예>

[Q] 다음 (1)~(2)와 같은 SCC의 특성을 활용해 아래 그래프에서 같은 SCC에 속하는 정점끼리 묶어 보시오.

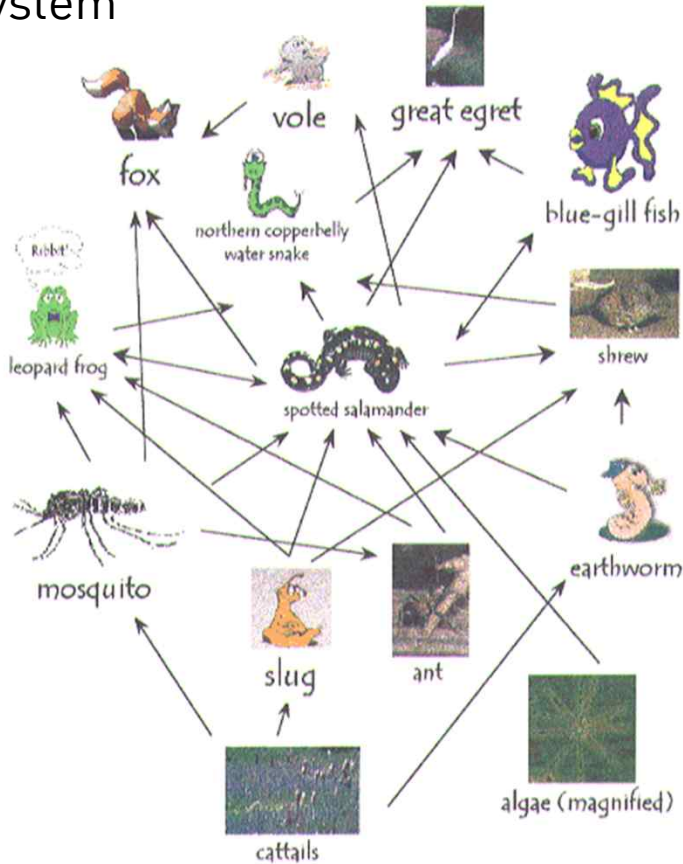
- (1) 같은 SCC 내부에는 cycle 있음
 - (2) 서로 다른 SCC 간에는 cycle 없음
- (Hint: 5개의 component 있음.)



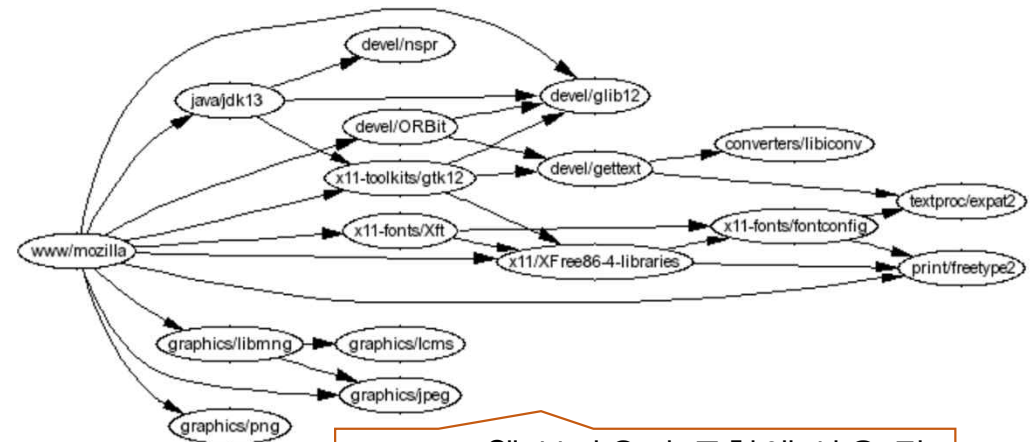


Strongly-connected component는 어디에 활용하나?

- 서로 먹고 먹히는 생물 관계에서 strongly-connected component는 하나의 독립적인 ecosystem



- SW 모듈 간 dependency 관계에서 (한 모듈이 다른 모듈의 API 호출) strongly-connected component 파악하면 하나의 모듈로 묶어 함께 관리하거나 혹은 하나의 패키지로 묶어서 한 번에 더 빠르게 메모리로 읽어오도록 할 수 있음



Firefox 웹 브라우저 구현에 사용된
모듈 간 dependency 예



Naïve한 방법: 모든 정점의 쌍 v, w 에 대해
DFS(v) 해서 w 에 도달 가능한지 확인하고
DFS(w) 해서 v 에 도달 가능한지 확인

[Q] 위와 같은 방법의 성능은 무엇에 비례하나?

$$VC_2 \times 2 (V + E) = \sim V^3 + V^2 E$$

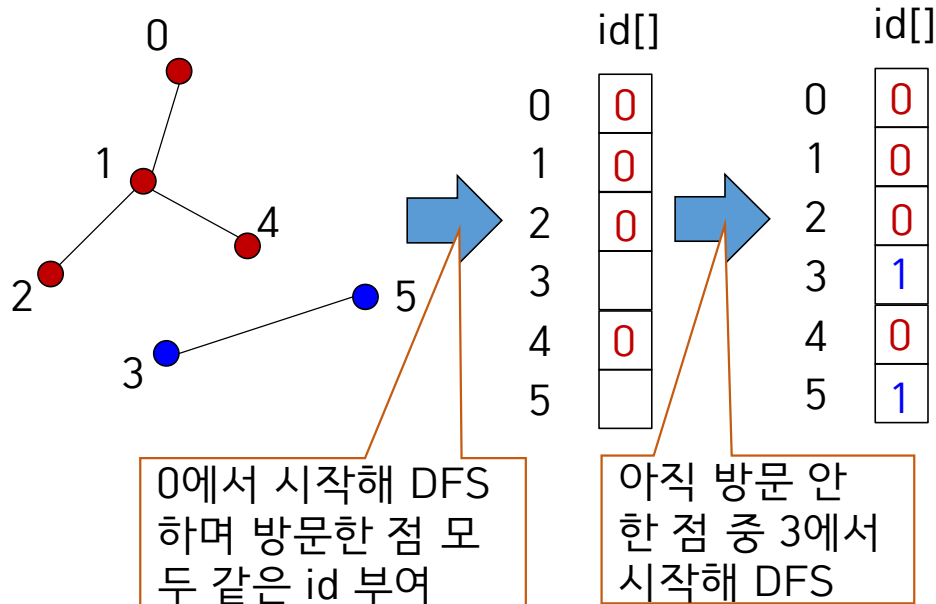


Undirected Graph

모든 간선이 양방향인 Digraph

63

- connected(v,w): v-w 경로 존재
- connected component: 서로 간에 모두 연결된 (도달 가능한) 정점의 최대 집합
- 아직 방문하지 않은 정점 v에 대해 DFS(v) or BFS(v) 해서 방문한 모든 점을 같은 component로 표시



Pseudo-code for Connected-component Detection

00 그래프의 각 정점 v에 대해 아래를 수행:

01 v를 아직 방문하지 않았다면:

02 v를 시작점으로 DFS 혹은 BFS 수행하며

03 새로 방문한 모든 정점 w를

04 v와 같은 component에 속하는 것으로 기록

DFS (or BFS) 시작 정점	발견한 connected component
DFS(0)	0 1 2 4
DFS(1) x	
DFS(2) x	
DFS(3)	3 5
DFS(4) x	
DFS(5) x	

두개의
connected
component

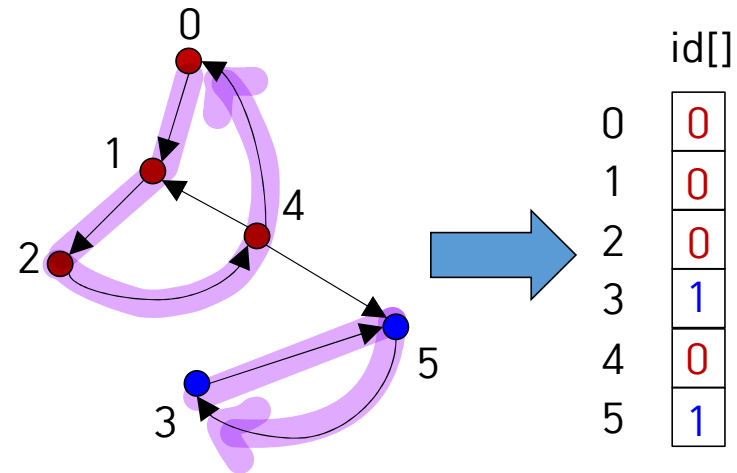


Digraph (Directed Graph)

간선에 **방향성** 있는 그래프

- strongly-connected(v, w): $v \rightarrow w$, $w \rightarrow v$ 경로 모두 존재
- Strongly-Connected Component (SCC): 서로 간에 모두 strongly-connected인 (양방향으로 도달 가능한, 즉 갔다가 돌아올 수 있는) 정점의 최대 집합

DFS (or BFS) 시작 정점	발견한 SCC
DFS(0)	0 1 2 3 4 5
DFS(1)	x
DFS(2)	x
DFS(3)	x
DFS(4)	x
DFS(5)	x

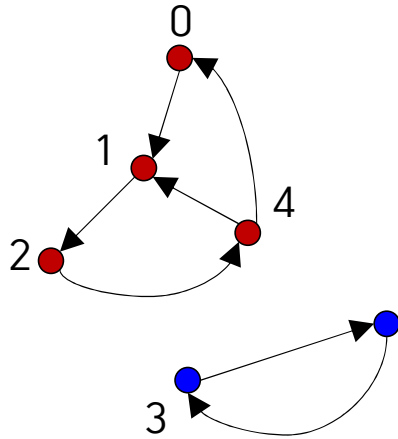


[Q] 어떻게 찾을까? Undirected graph와 마찬가지로 DFS or BFS 하면 되나?

20 이상에 나옴 BFS x

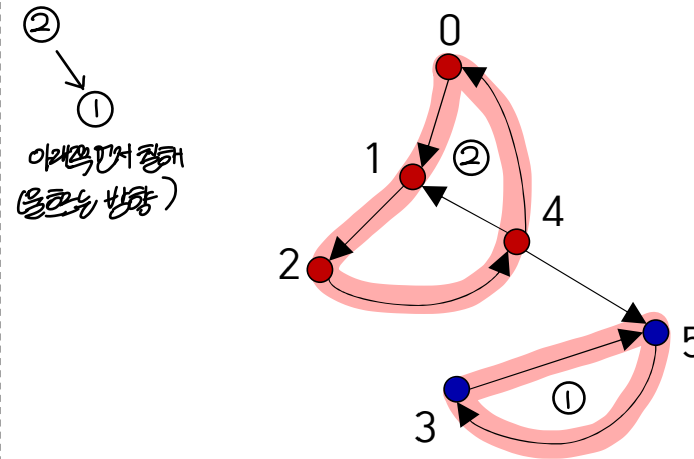


[Q] 서로 다른 SCC 간 간선 없다면
undirected 경우처럼
DFS or BFS 적용해 SCC 찾을 수 있나?



한 component 내 어느 정점에서 DFS or BFS
시작하더라도 그 component 내부의 모든 정점
찾을 수 있음. Strongly-connected 이므로

[Q] 서로 다른 SCC 간 간선 있다면
어떤 순서로 DFS or BFS 수행하면
서로 다른 SCC 분리해 찾을 수 있을까?



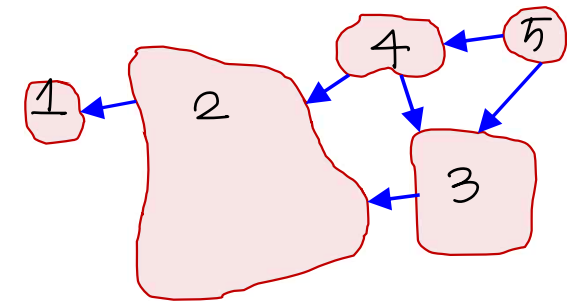
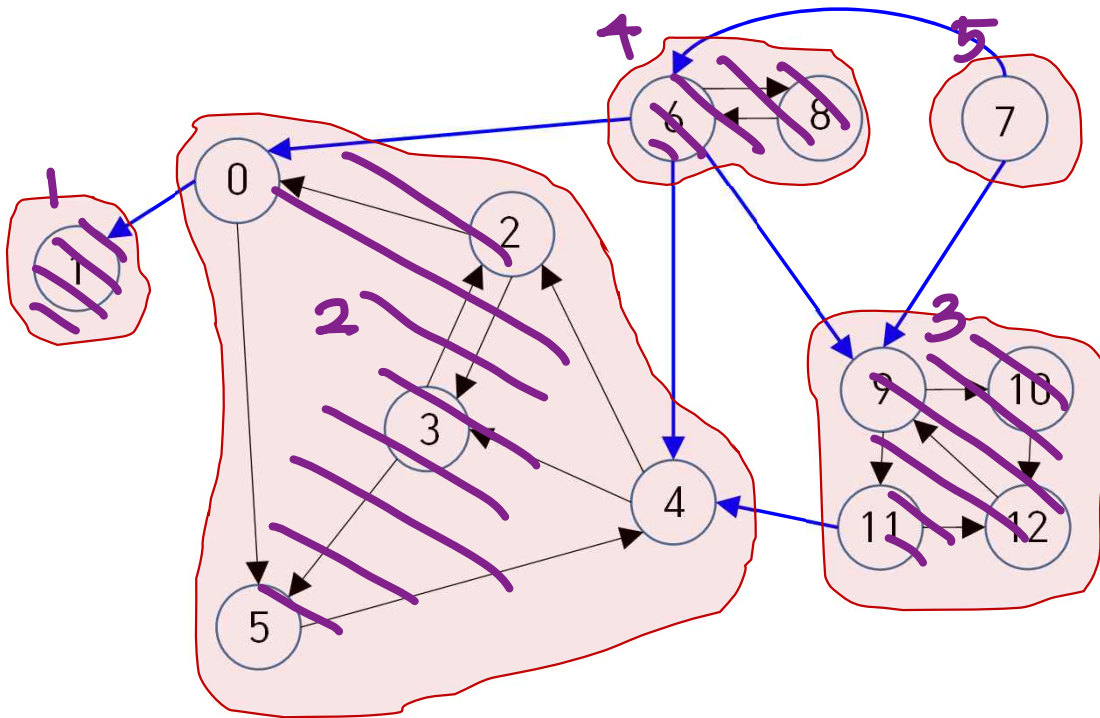
DFS (or BFS) 시작 정점	발견한 SCC
1st DFS(5)	3 5
DFS(4)	0, 1, 2, 4
DFS(3) ✕	
DFS(2) ✕	
DFS(1) ✕	
DFS(0) ✕	

reserved.



[Q] 아래 digraph와 같이 서로 다른 SCC 간 (단방향) 간선이 있다면
어느 SCC에 속한 정점부터 순서대로 DFS or BFS를 적용해야
서로 다른 SCC들을 잘 분리해서 찾을 수 있을까?

한 component 내 임의의 정점에서 DFS or BFS를
시작해 새로 방문한 모든 정점을 같은 component
에 속하는 것으로 기록한다고 가정



[Q] 각 SCC를 하나의 큰 정점으로 생각해 보자.
Cycle 없는 digraph를 볼 수 있다. 방금 DFS를 적
용한 순서는 이 그래프에 대한 어떤 순서인가?

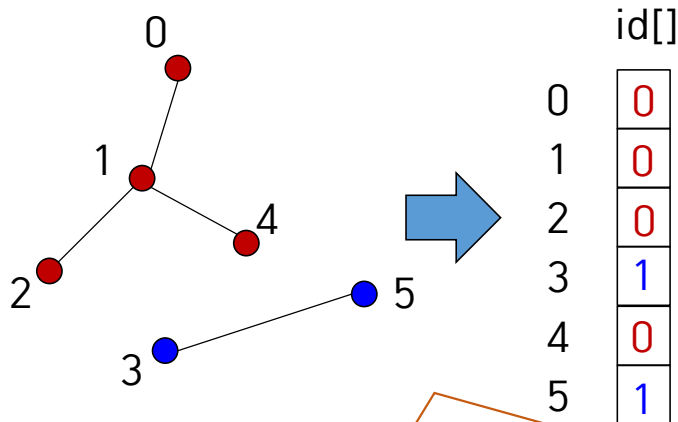
topological order라고 할 수 있음.



Undirected Graph

모든 간선이 양방향인 Digraph

- connected(v,w): v-w 경로 존재
- connected component: 서로 간에 모두 연결된 (도달 가능한) 정점의 최대 집합
- 아직 방문하지 않은 정점 v에 대해 DFS(v) or BFS(v) 해서 방문한 모든 점을 같은 component로 표시



0, 1, 2, .. 순으로

DFS하며 방문한 점 모두 같은 id 부여

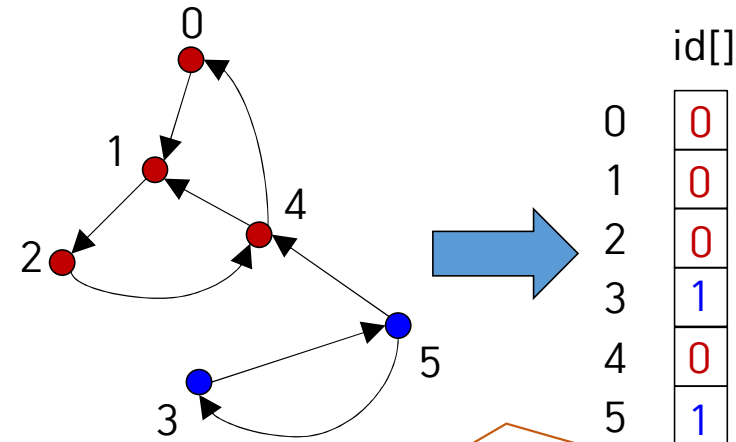
Component 간에 분리되어 있으므로
어떤 순서라도 OK

Digraph (Directed Graph)

간선에 방향성 있는 그래프

67

- strongly-connected(v,w): $v \rightarrow w$, $w \rightarrow v$ 경로 모두 존재
- strongly-connected component: 서로 간에 양방향으로 도달 가능한 정점의 최대 집합
- (SCC를 한 정점으로 본 그래프에서) topological order 역순에 따라 DFS (or BFS) 해서 방문한 모든 점을 같은 component로 표시



SCC 기준 topological order 역순으로

DFS하며 방문한 점 모두 같은 id 부여

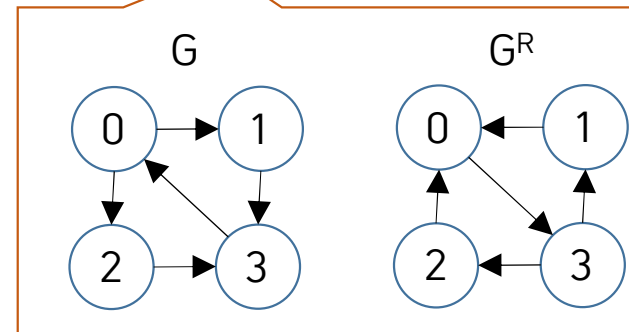
Component 간에 간선 있어 순서 잘못 정하면
여러 component를 하나로 잘못 인식하므로



각 SCC를 하나의 정점으로 본 그래프에서
topological order의 **역순(reverse)** 찾는 방법:

- Topological order는 **DFS**로 얻음
- Topological order의 **역순** 얻어야 하므로 **reverse graph**에 DFS 적용

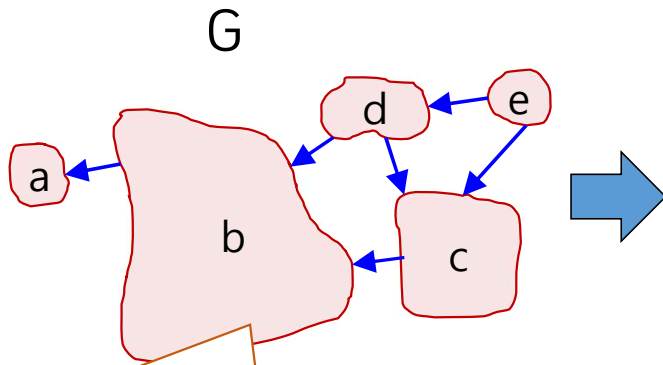
Reverse graph, G^R : 모든 간선을 반대 방향으로 뒤집은 그래프



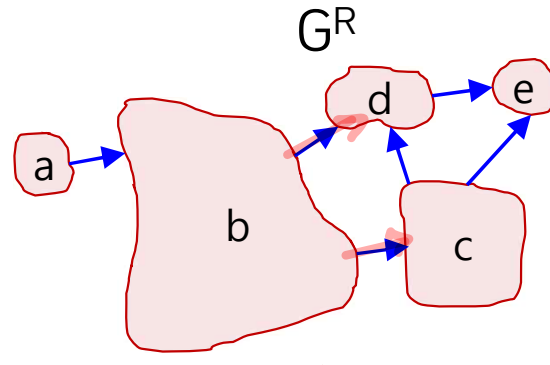


각 SCC를 하나의 정점으로 본 그래프에서
topological order의 **역순(reverse)** 찾는 방법:

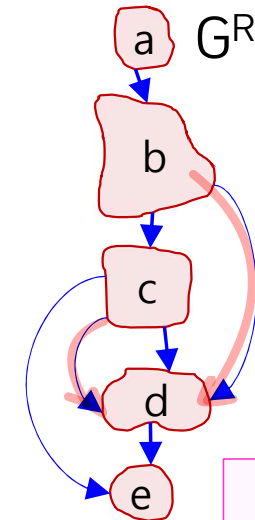
- Topological order는 **DFS**로 얻음
- Topological order의 **역순** 얻어야 하므로 **reverse graph**에 DFS 적용



a, b, c, d, e 순서 알고자 함 (이 순서로 DFS or BFS 하면 strongly-connected component 발견할 수 있으므로)



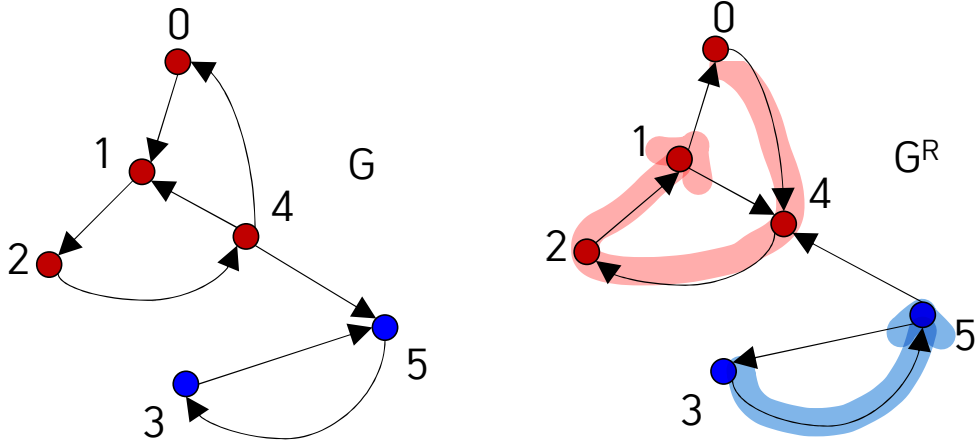
Reverse Graph



a, b, c, d, e

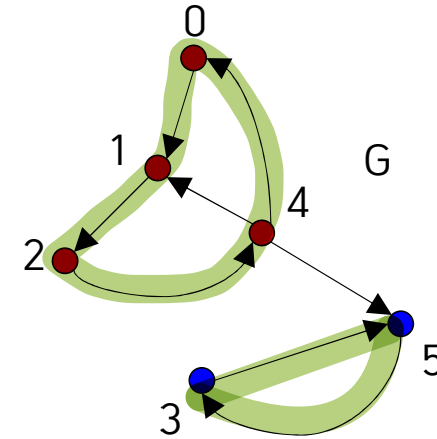
Reverse Graph에 대한 topological order. 우리가 원래 그래프 G에 대해 DFS (or BFS) 수행하고자 하는 순서와 같음

[Q] 다음 그래프 G 의 reverse graph G^R 의 topological order를 찾으시오.



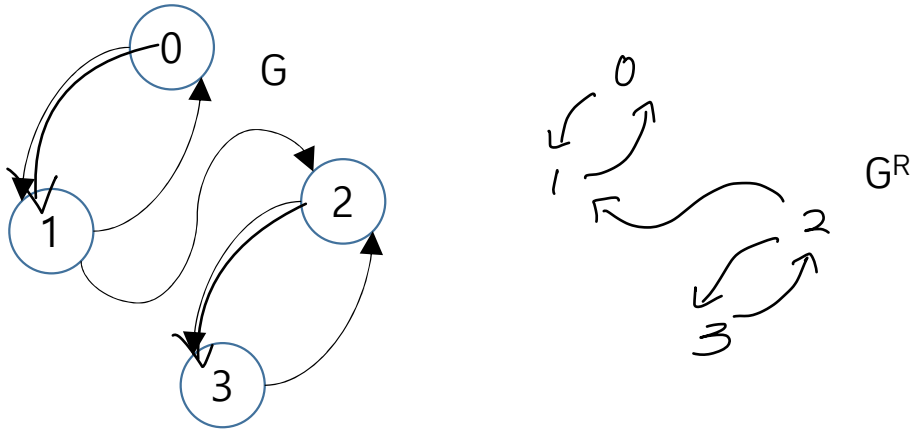
DFS 시작 정점	result 리스트
DFS(0) (walk DFS)	0 4 2 1
DFS(1) ✕	
DFS(2) ✕	
DFS(3)	3 5 0 4 2 1
DFS(4) ✕	
DFS(5) ✕	

[Q] G^R 의 topological order 순으로 DFS를 수행해 SCC를 찾으시오.



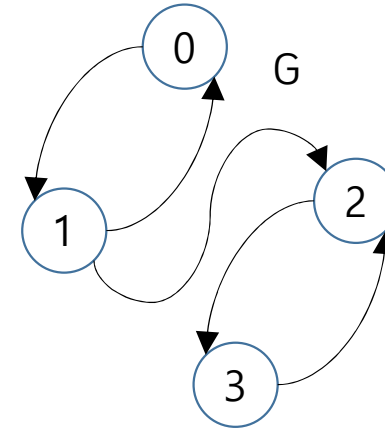
DFS 시작 정점	발견한 SCC
3	3, 5
5 ✕	
0	0, 1, 2, 4
4 ✕	
2 ✕	
1 ✕	

[Q] 다음 그래프 G 의 reverse graph G^R 의 topological order를 찾으시오.



DFS 시작 정점	result 리스트
DFS(0)	0 1
DFS(1) ✕	
DFS(2)	2 3 0 1
DFS(3) ✕	

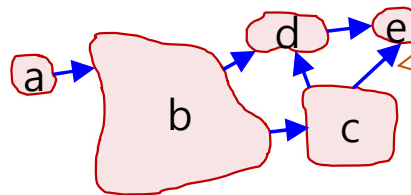
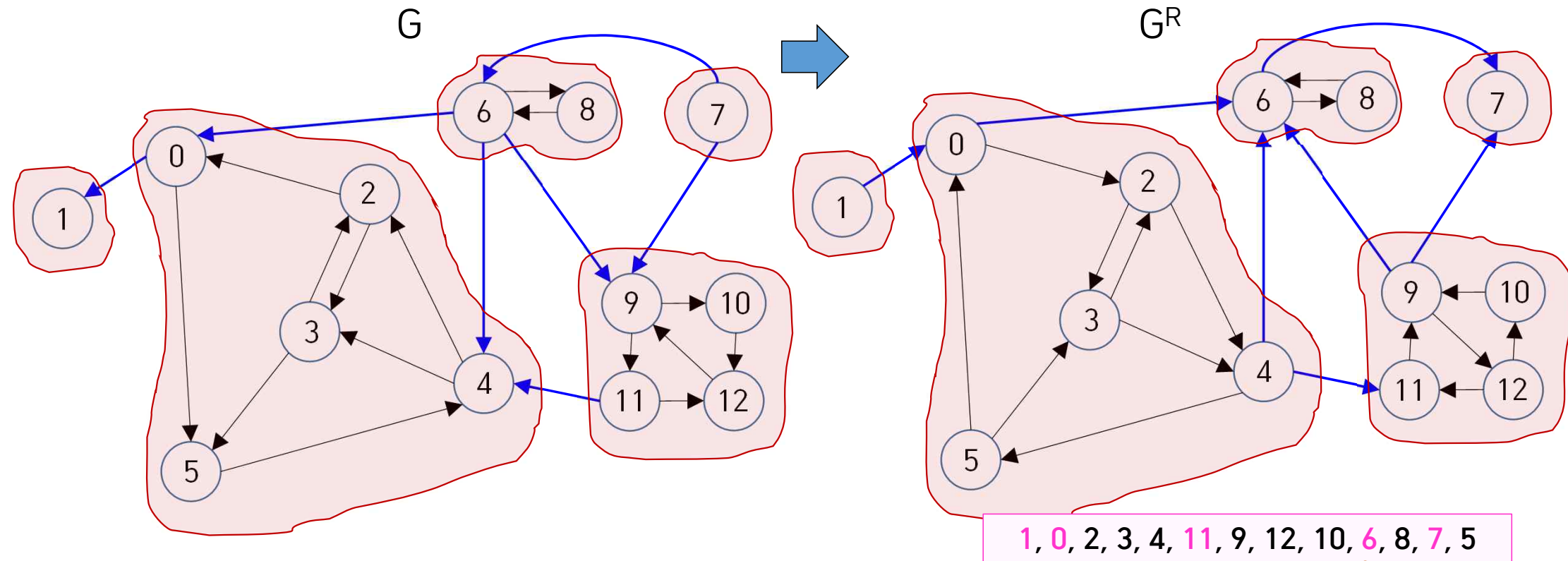
[Q] G^R 의 topological order 순으로 DFS를 수행해 SCC를 찾으시오.



DFS 시작 정점	발견한 SCC
2	2 3
3 ✕	
0	0 1
1 ✕	



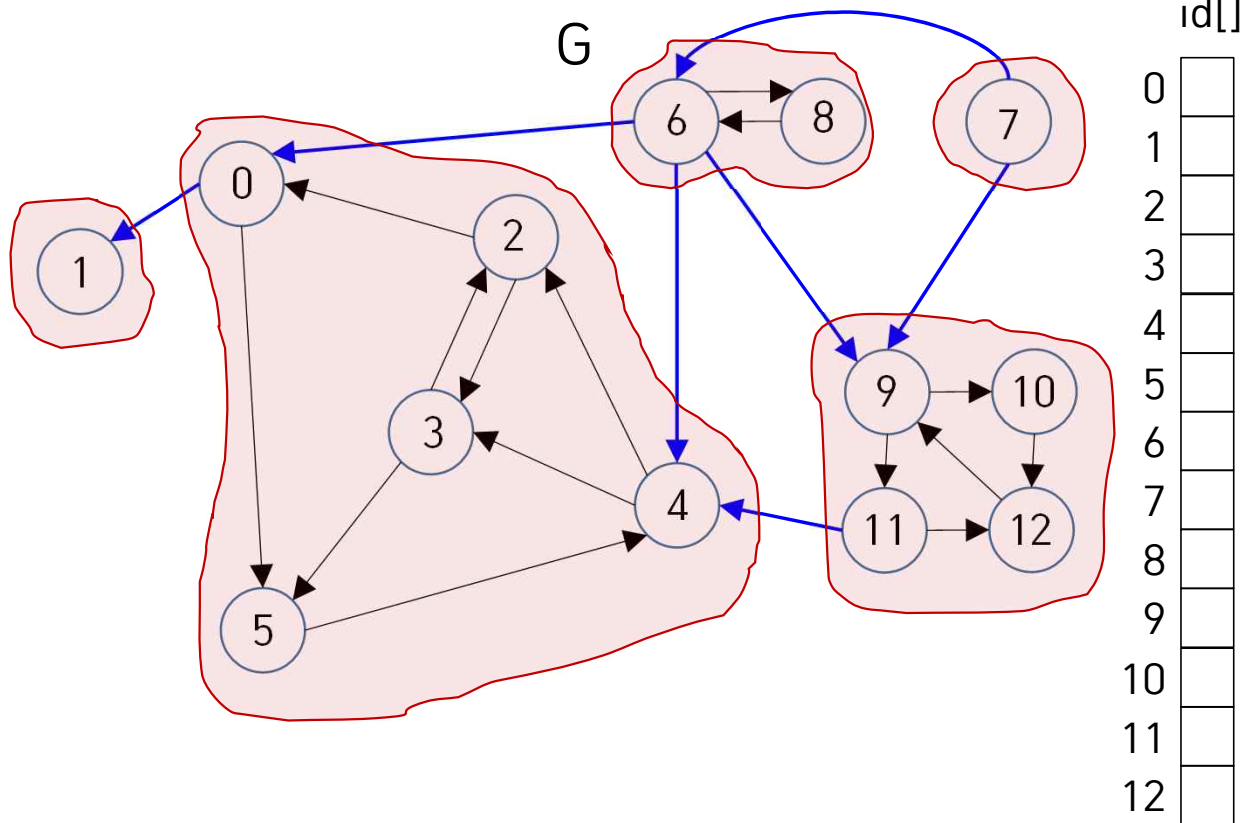
- (Phase 1) **Reverse graph**의 (DFS 적용해) **topological order** 얻기
- (Phase 2) 그 순서로 원래 그래프에 DFS (or BFS) 수행해 SCC 찾음



Topological order에서 **각 component에 대해 처음 나오는 정점**만 보면 component-level에서 우리가 원하는 순서와 같음



- (Phase 1) Reverse graph의 (DFS 적용해) topological order 얻기
- (Phase 2) 그 순서로 원래 그래프에 DFS (or BFS) 수행해 SCC 찾음



1, 0, 2, 3, 4, 11, 9, 12, 10, 6, 8, 7, 5

그래프의 각 정점 v 에 대해 (위 순서로) 아래 수행:
 v 를 아직 방문하지 않았다면:
 v 를 시작점으로 DFS 혹은 BFS 수행하며
 새로 방문한 모든 정점 w 를
 v 와 같은 component에 속하는 것으로 기록

[Q] Reverse graph의 topological order 순서로 DFS 적용하며 각 정점의 component id를 기록해 보자. 5개 component를 잘 구분해 내는가?



Kosaraju-Sharir (코사라쥬-샤이르) 알고리즘

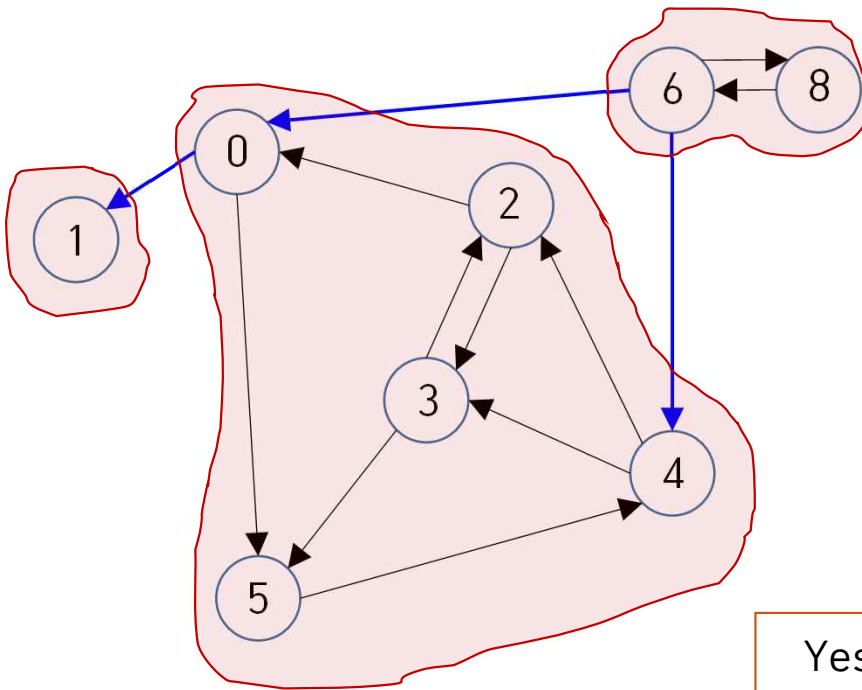
- (Phase 1) Reverse graph의 (DFS 적용해) topological order 얻기
- (Phase 2) 그 순서로 원래 그래프에 DFS (or BFS) 수행해 SCC 찾음

[Q] Kosaraju-Sharir 알고리즘이 digraph에서 SCC를 찾는 시간은 다음 중 무엇에 비례하나?

$$\begin{array}{l}
 V \\
 E \\
 \textcircled{V+E} \\
 V \times (V+E)
 \end{array}
 \quad
 \begin{array}{l}
 G^R = \sim V + E \\
 \downarrow \\
 \text{topological sort (DFS)} = \sim V + E \\
 + \quad \text{DFS on } G = \sim V + E
 \end{array}$$



[FAQ 1] Strong component 간에는 cycle 없지만,
각 strong component 내부에는 cycle 있다.
그래도 topological sort 사용해도 괜찮은가? *NO*



reverseList = [] # 정점 저장할 목록 초기화
Digraph의 각 정점 v 에 대해:

아직 v 를 방문하지 않았다면 DFS(v)
reverseList.reverse()

DFS(to visit vertex v):

v 를 방문한 것으로 표시

$v \rightarrow w$ 간선 있는 정점 중 **아직 방문 않은** 모든 정점
 w 를 차례대로 DFS(w) 호출해 재귀적으로 방문

reverseList.append(v) # v 를 목록에 추가

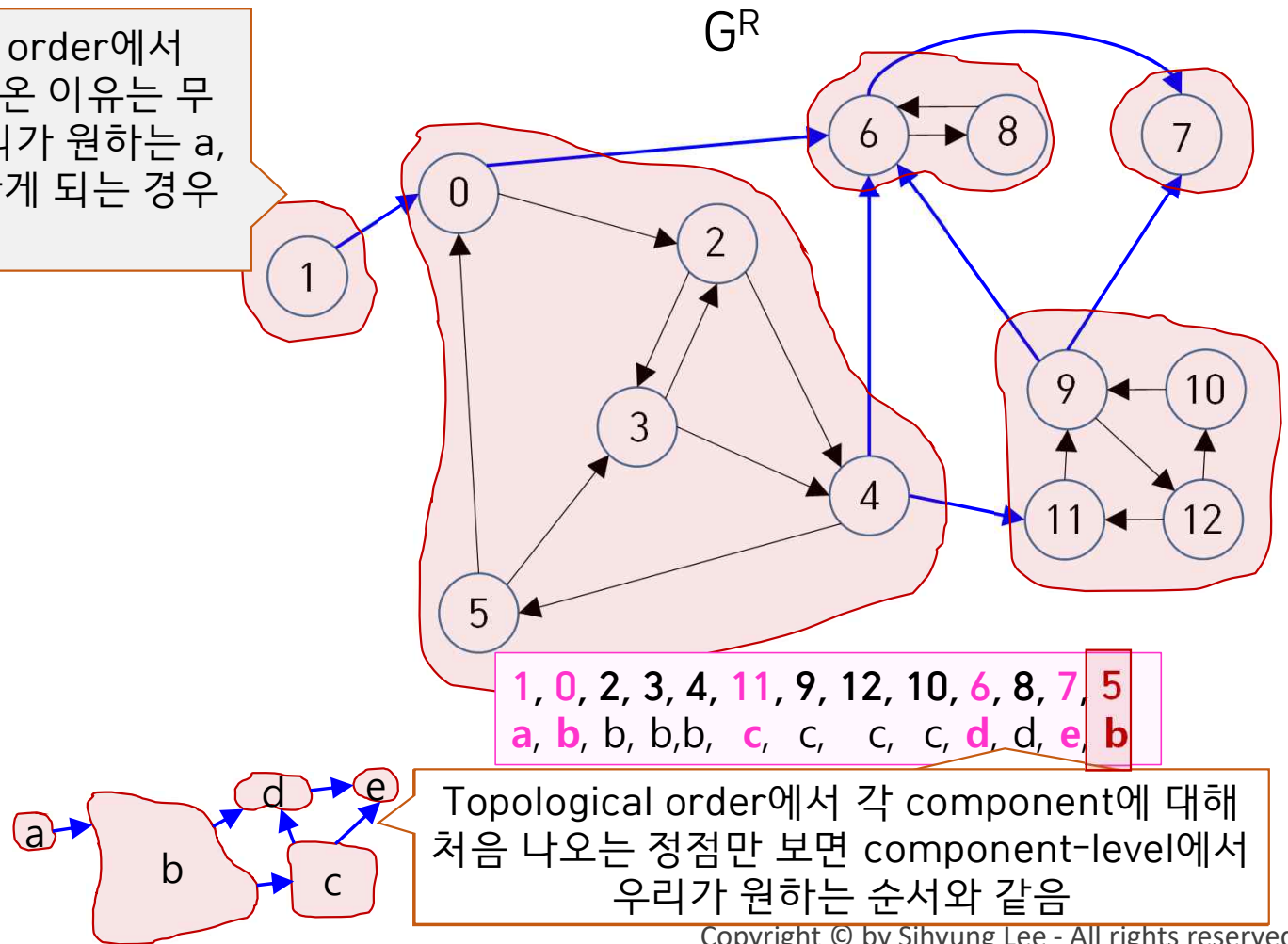
Yes. 방문했던 곳은 다시 방문 않는다는 원래
topological order 규칙만 잘 지키면 문제 없음

같은 component 내 정점 간 순서는 상관 없음
서로 다른 component의 정점 간 순서만
topological order에 맞게 나오면 됨



- (Phase 1) **Reverse graph**의 (DFS 적용해) **topological order** 얻기
- (Phase 2) 그 순서로 원래 그래프에 DFS (or BFS) 수행해 strongly-connected components 찾음

[FAQ2] Reverse graph의 topological order에서 **component b**에 속하는 **5**가 마지막에 온 이유는 무엇일까? 이렇게 되면 phase 2에서 우리가 원하는 a, b, c, d, e 순서로 DFS를 수행하지 못하게 되는 경우가 발생하지 않는가?





- (Phase 1) **Reverse graph**의 (DFS 적용해) **topological order** 얻기
- (Phase 2) 그 순서로 원래 그래프에 DFS (or BFS) 수행해 strongly-connected components 찾음

G

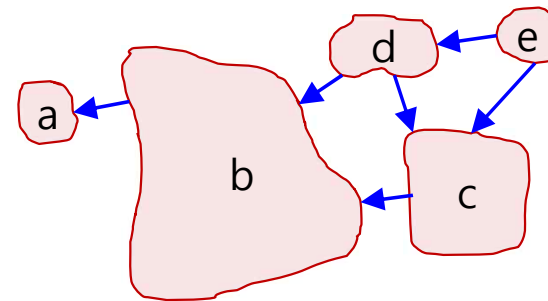
[FAQ3] Phase 1에서 reverse graph의 topological order 얻는 대신 원래 graph의 topological order 얻은 후 뒤집으면 되지 않을까?

G의 topological order

7, 6, 9, 11, 10, 12, 8, 0, 5, 4, 2, 3, 1

G의 topological order 역순

1, 3, 2, 4, 5, 0, 8, 12, 10, 11, 9, 6, 7
a, b, b, b, b, b, d, c, c, c, c, d, e

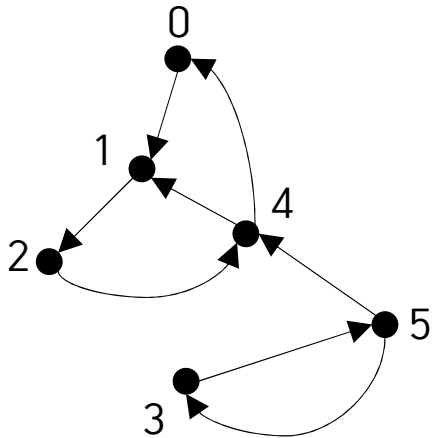




Kosaraju-Sharir (코사라쥬-샤이르) 알고리즘: Digraph의 Strongly-Connected Components 찾기

- (Phase 1) Reverse graph의 (DFS 적용해) topological order 얻기
- (Phase 2) 그 순서로 원래 그래프에 DFS (or BFS) 수행해 strongly-connected components 찾음

[Q] 다음 digraph에 Kosaraju-Sharir 알고리즘 적용해 SCC 찾으시오. *답은 2개*

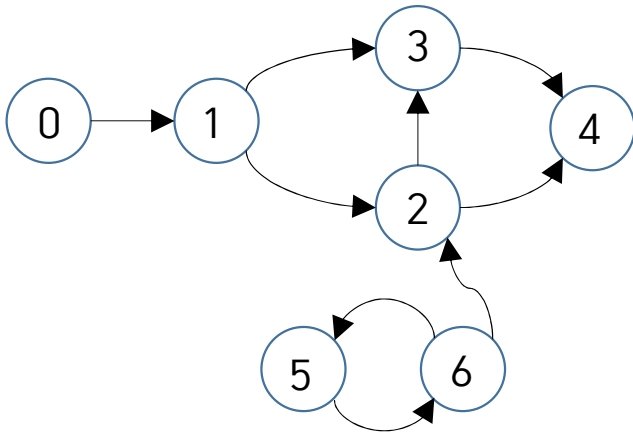




Kosaraju-Sharir (코사라쥬-샤이르) 알고리즘: Digraph의 Strongly-Connected Components 찾기

- (Phase 1) **Reverse graph**의 (DFS 적용해) **topological order** 얻기
- (Phase 2) **그 순서로 원래 그래프에 DFS** (or BFS) 수행해 strongly-connected components 찾음

[Q] 다음 digraph에 Kosaraju-Sharir 알고리즘 적용해 SCC 찾으시오. *동양혁 정답*





[Q] 다음 중 digraph에서 SCC를 찾는 방법으로 잘못된 것은?

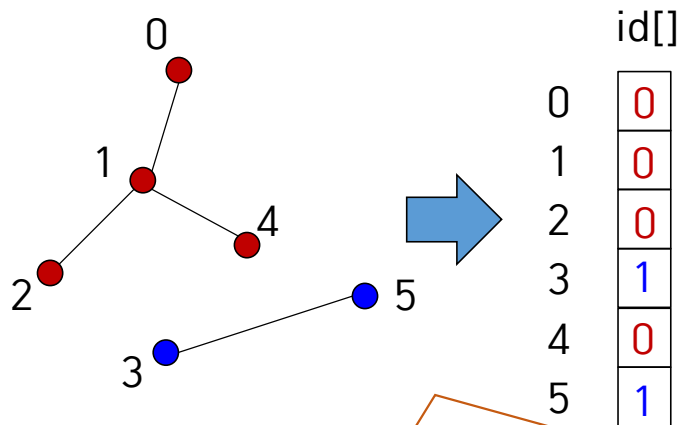
- 1 ■ DFS로 G^R 의 topological order 얻고, 그 순서로 G 에 DFS 수행
- 2 ■ DFS로 G^R 의 topological order 얻고, 그 순서로 G 에 BFS 수행
- 3 ~~■ BFS로 G^R 의 topological order 얻고, 그 순서로 G 에 DFS 수행~~



Undirected Graph

모든 간선이 양방향인 Digraph

- connected(v,w): v-w 경로 존재
- connected component: 서로 간에 모두 연결된 (도달 가능한) 정점의 최대 집합
- 아직 방문하지 않은 정점 v에 대해 DFS(v) or BFS(v) 해서 방문한 모든 점을 같은 component로 표시



0, 1, 2, .. 순으로
DFS하며 방문한 점 모두 같은 id 부여

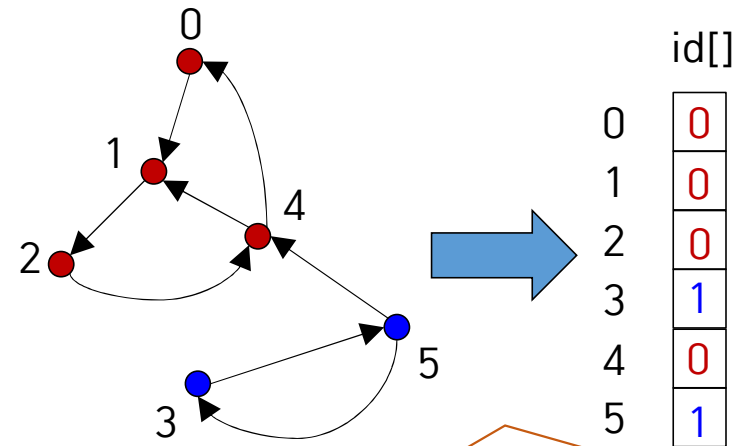
Component 간에 분리되어 있으므로
어떤 순서라도 OK

Digraph (Directed Graph)

간선에 방향성 있는 그래프

81

- strongly-connected(v,w): $v \rightarrow w$, $w \rightarrow v$ 경로 모두 존재
- strongly-connected component: 서로 간에 양방향으로 도달 가능한 정점의 최대 집합
- Phase 1: Reverse 그래프의 topological order 찾기**
- Phase 2: Phase 1 순서에 따라 DFS (or BFS) 해서 방문한 모든 점을 같은 component로 표시**



Reverse 그래프의 topological order 순으로
DFS하며 방문한 점 모두 같은 id 부여

Component 간에 간선 있어 순서 잘못 정
하면 여러 component를 하나로 잘못 인식

|| rights reserved.



Undirected and Directed Graphs

Graph의 표현 방법, 탐색 방법 및 이들의 활용도에 대해 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. Digraph(Directed Graph)의 표현(저장) 방법
03. Digraph에 대한 탐색
04. Digraph 탐색의 활용 예: 프로그램 흐름 분석, Garbage Collection, 웹 크롤링
05. Digraph 탐색의 활용 예: Topological Sort
06. Digraph 탐색의 활용 예: Strongly-Connected Component의 탐지
07. 실습: Kosaraju-Sharir 알고리즘 구현



실습 목표: DirectedGraph 코드에 SCC(Strongly-Connected Component) 구하는 기능 추가

- 이번 시간에 배운 DFS, BFS, Topological Sort의 특성 이해
- 이들은 적재적소에 활용해 보기

프로그램 구현 조건

- SCC(Strongly-Connected Component) 구해 결과 저장하는 클래스 구현

class SCC:

def __init__(self, g): # 생성자. 입력 그래프 g에 대한 SCC 구해 저장

def connected(self, v, w): # 그래프 g의 두 정점 v와 w가 같은 SCC에 속하면 True 반환

- 입력 **g**: directed graph 나타내는 **Digraph 클래스 객체**
- 입력 **v, w**: directed graph **g**에 속하는 정점 번호로 0 ~ (g.V-1) 범위의 정수
- 이번 시간에 제공한 코드 DirectedGraph.py에 위 함수에 대한 코드 작성해 제출
 - 위 파일에 포함된 topologicalSort() 함수는 반드시 사용해야 함
 - 위 파일에 포함된 Digraph 클래스도 반드시 사용해야 함 (__init__() 함수의 입력이므로)
 - 위 파일에 포함된 코드 중 SCC 클래스 외의 코드는 변경하면 안 됨
 - 이미 import된 모듈 외 추가로 import할 수 없음

구현된 API 정리

이미 구현된 기능

`class Digraph:` # Digraph 객체를 저장하는 클래스

`def reverse(self):` # 이 객체가 나타내는 그래프의 reverse graph 객체 만들어 반환

`class DFS:` # Digraph 객체에 DFS를 수행하고 결과를 저장하는 클래스

`class BFS:` # Digraph 객체에 BFS를 수행하고 결과를 저장하는 클래스

`def webCrawl(roots, maxDepth=1):` # 웹주소의 리스트 roots에서 시작해 링크를 따라 웹주소를 수집하는 함수

`def topologicalSort(g):` # Digraph 객체 g에 대한 topological order를 구해 반환하는 함수

구현할 API 정리: class SCC

구현해야 하는 기능

class SCC:

def __init__(self, g):

SCC 생성자. Digraph 객체 g에 대한 strongly-connected component 구해

멤버 변수 self.id[]와 self.count에 결과 저장

self.id[v]: v가 속한 component id를 저장하며, component id는 0부터 시작해 1씩 증가

self.count: component의 수를 저장

#

(1) g의 reverse 그래프에 대한 topological order 얻기

(2) 앞에서 얻은 순서에서 아직 방문하지 않은 각 정점 v에 대해:

v에서 시작해 DFS 혹은 BFS 수행하면서

아직 방문하지 않은 모든 정점 w를 방문하되

이들을 모두 같은 component로 표시 (즉 self.id[w]에 같은 번호 저장)

def connected(self, v, w):

v와 w가 서로 연결된 경우 (같은 SCC에 속하는 경우) True 반환하고

그렇지 않으면 False를 반환

#

__init__()를 수행한 결과가 저장된 self.id[v]와 self.id[w] 활용해 결과값 반환

Kosaraju-Sharir
알고리즘 사용

Hint: UndirectedGraph.py 파일에서
undirected graph의 connected component를 구하는
class CC도 참조해 작성해 보세요.

프로그램 입출력 예: `__main__` 아래 테스트 코드에 있음

```
g1 = Digraph(6)
```

```
g1.addEdge(0,1)
```

```
g1.addEdge(1,2)
```

```
g1.addEdge(2,4)
```

```
g1.addEdge(3,5)
```

```
g1.addEdge(4,0)
```

```
g1.addEdge(4,1)
```

```
g1.addEdge(5,3)
```

```
g1.addEdge(5,4)
```

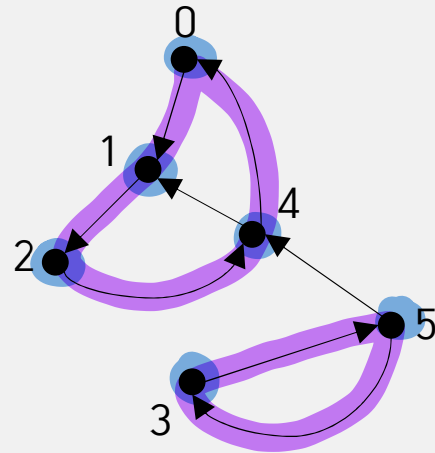
```
scc1 = SCC(g1)
```

```
print(scc1.count, scc1.id)
```

```
print(scc1.connected(1,4))
```

```
print(scc1.connected(2,5))
```

```
print(scc1.connected(3,5))
```



실행 결과

2 [0, 0, 0, 1, 0, 1]

True

False

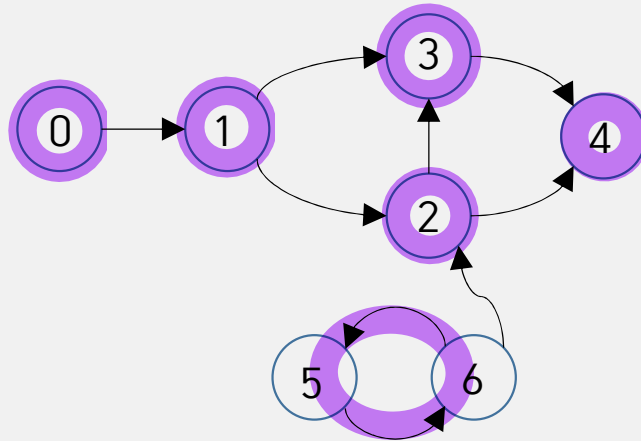
True

2개 component {0,1,2,4}, {3,5} 있음 의미

{0,1,2,4}의 id가 같고
{3,5}의 id가 같으면 됨
(예: [1,1,1,0,1,0]도 괜찮음)

프로그램 입출력 예: `__main__` 아래 테스트 코드에 있음

```
g2 = Digraph(7)
g2.addEdge(0,1)
g2.addEdge(1,2)
g2.addEdge(1,3)
g2.addEdge(2,3)
g2.addEdge(2,4)
g2.addEdge(3,4)
g2.addEdge(5,6)
g2.addEdge(6,2)
g2.addEdge(6,5)
scc2 = SCC(g2)
print(scc2.count, scc2.id)
print(scc2.connected(0,2))
print(scc2.connected(2,4))
print(scc2.connected(4,5))
print(scc2.connected(5,6))
```



실행 결과

6 [5, 4, 2, 1, 0, 3, 3]

False

False

False

True

6개 component {0}, {1}, {2},
{3}, {4}, {5,6} 있음 의미

{5,6}의 id 같고
나머지의 id는 모두 서로 다르면 됨 (단 id
는 0 ~ self.count-1 사이의 값 사용)

프로그램 입출력 예: __main__ 아래 테스트 코드에 있음

```
g3 = Digraph(13)
g3.addEdge(0,1)
g3.addEdge(0,5)
g3.addEdge(2,0)
g3.addEdge(2,3)
g3.addEdge(3,2)
g3.addEdge(3,5)
g3.addEdge(4,2)
g3.addEdge(4,3)
g3.addEdge(5,4)
g3.addEdge(6,0)
g3.addEdge(6,4)
g3.addEdge(6,8)
g3.addEdge(6,9)
g3.addEdge(7,6)
g3.addEdge(7,9)
g3.addEdge(8,6)
g3.addEdge(9,10)
g3.addEdge(9,11)
g3.addEdge(10,12)
g3.addEdge(11,4)
g3.addEdge(11,12)
g3.addEdge(12,9)
```

```
scc3 = SCC(g3)
print(scc3.count, scc3.id)
print(scc3.connected(0,3))
print(scc3.connected(0,7))
print(scc3.connected(0,9))
print(scc3.connected(7,8))
print(scc3.connected(7,11))
print(scc3.connected(10,12))
```

실행 결과

5 [1, 0, 1, 1, 1, 1, 3, 4, 3, 2, 2, 2, 2]

True

False

False

False

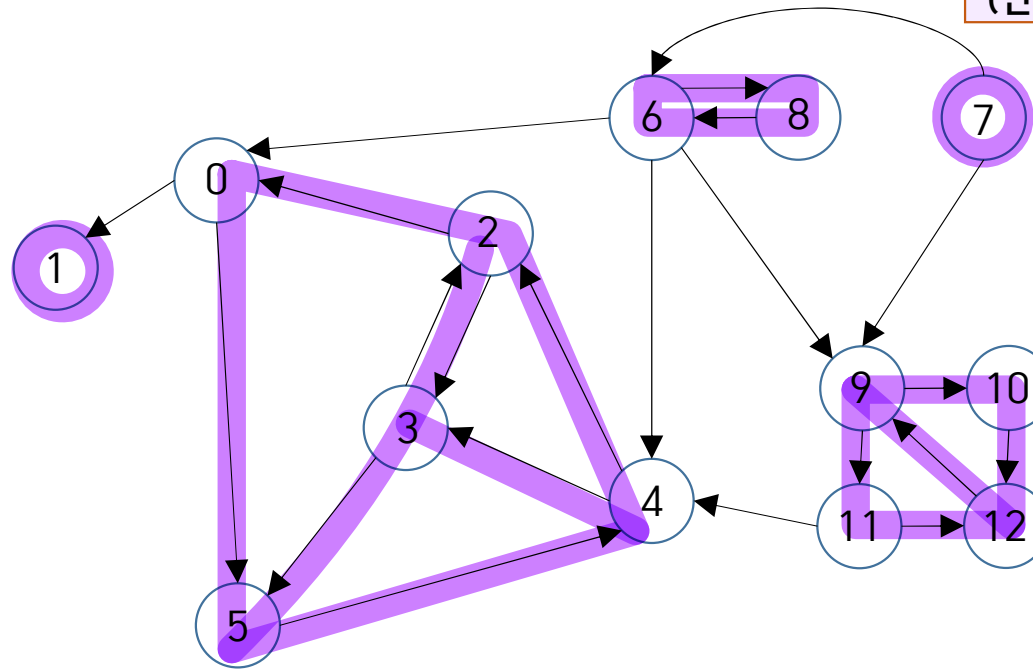
False

True

5개 component

{0,2,3,4,5},{1},{6,8},{7},{9,10,11,12}
있음 의미

같은 component끼리 id 같으면 되며,
위 예제와 똑같이 지정될 필요는 없음
(단 id는 0 ~ self.count-1 사이의 값 사용)

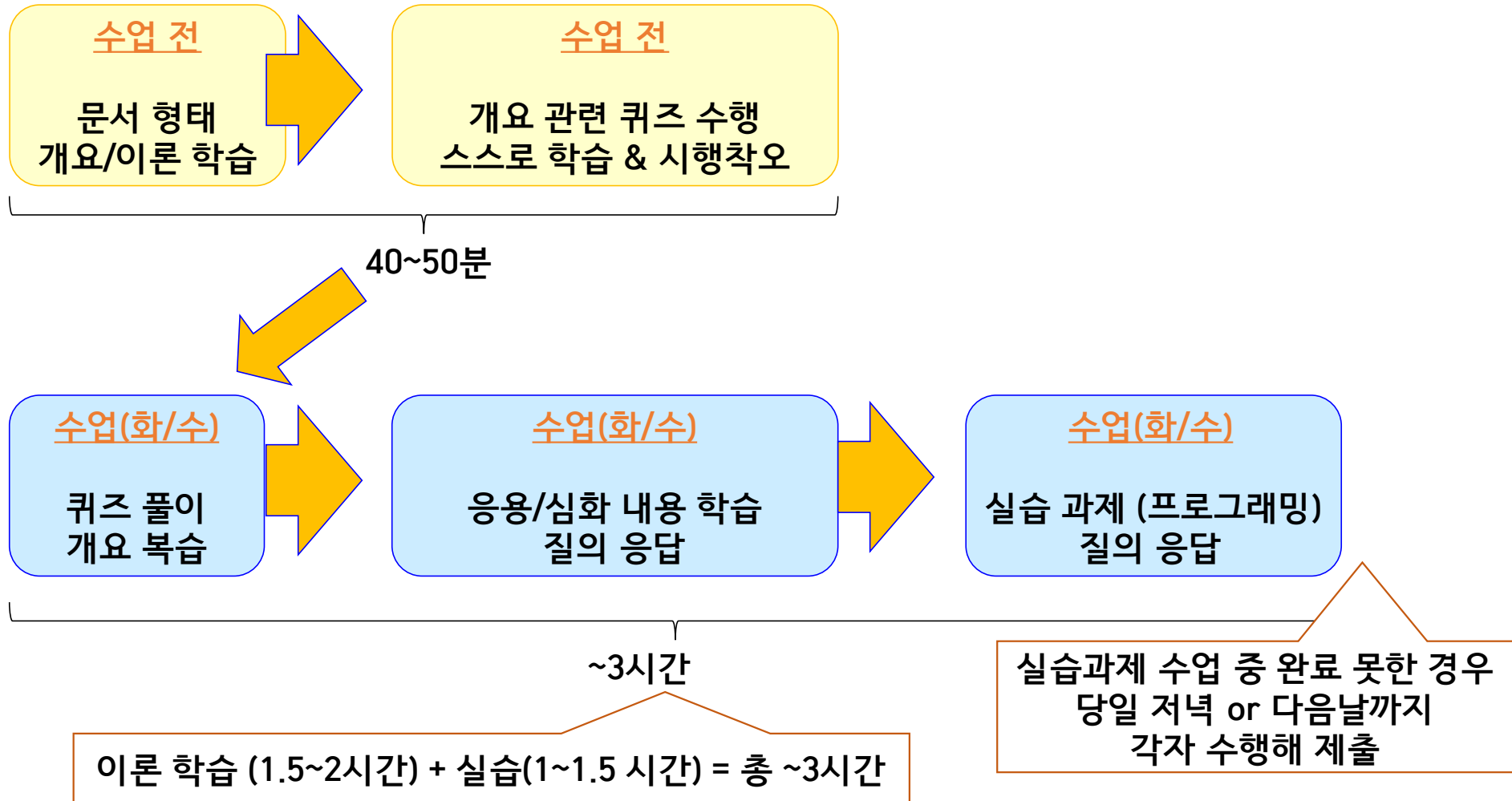


그 외 유의사항

- SCC 클래스의 멤버 변수에 대한 아래 조건을 지켜야 함
- `self.count`는 component의 수를 저장
- `self.id[v]`는 정점 `v`가 속한 component의 id를 저장하며
- component id는 $0 \sim (\text{self.count}-1)$ 범위에 속한 정수여야 함
- 그때까지 찾은 SCC의 count를 id로 사용하면 위 조건 만족함 ([UndirectedGraph.py의 CC 클래스 참조](#))
- 또한 수행 시간에 대한 다음 조건을 지켜야 함
- `connected(v, w)` 함수의 수행 시간은 상수 시간이어야 하며, 정점 개수 V 나 간선 개수 E 에 비례해서는 안 됨
- 즉 SCC 탐색은 SCC 클래스 객체 생성 시 (생성자 함수에서) 한 번 수행해야 하며
- 그 후에 `connected()`를 호출할 때마다 수행하면 안 됨



스마트 출결





중간고사 점수 확인

- 시험 점수 관련 문의는 오늘 수업 마칠 때 까지만 받습니다.
- 친구와 점수 비교하지 말고 **앞으로 계속 열심히 해주세요.**

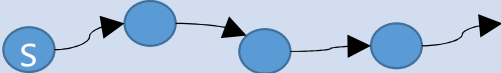
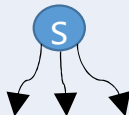
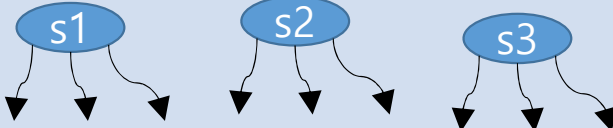


12:00까지 실습 & 질의응답

- 작성한 코드는 lms > 강의 콘텐츠 > 오늘 수업 > 실습 과제 제출함에 제출
- 시간 내 제출 못한 경우 내일 11:59pm까지 제출 마감
- 마감 시간 후에는 제출 불가하므로 그때까지 작성한 코드 꼭 제출하세요.

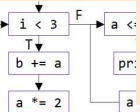
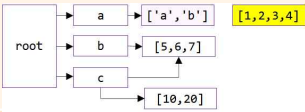
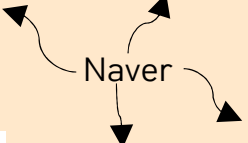
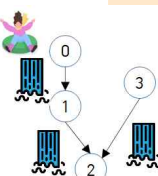
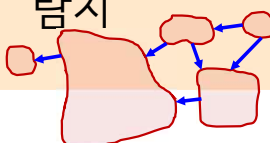


Digraph 기본 탐색 알고리즘 정리

알고리즘	방법	특성
DFS	<ul style="list-style-type: none">- 출발지 s로부터 갈 수 있는 곳을 최대한 깊이까지 가본 후 돌아 나오며 같은 방식으로 다른 길 탐색 	<ul style="list-style-type: none">- 출발지 s에서 도달 가능한 곳 모두 찾음- 도달 가능한 곳까지 임의의 경로 찾음- Topological order 항상 지키며 탐색- 탐색 시간 (V+E)에 비례
BFS	<ul style="list-style-type: none">- 출발지 s에서 같은 거리에 갈 수 있는 모든 곳을 함께 병렬로 탐색 	<ul style="list-style-type: none">- 출발지 s에서 도달 가능한 곳 모두 찾음- 도달 가능한 곳까지 최소 간선 경로 찾음- Topological order 여기는 경우도 있음- 탐색 시간 (V+E)에 비례
Multi-source BFS	<ul style="list-style-type: none">- 둘 이상의 출발지 s에서 같은 거리에 갈 수 있는 모든 곳을 함께 병렬로 탐색 	<ul style="list-style-type: none">- 동시 출발지가 여럿이므로 같은 시간에 더 다양한 목적지 탐색 가능- 탐색 시간 (V+E)에 비례



Digraph 기본 탐색 알고리즘 활용한 어플리케이션 정리

어플리케이션	방법	활용한 기본 탐색 알고리즘
프로그램 흐름 분석 	명령어를 정점으로, 명령어간 흐름을 간선으로 모델링. Digraph 탐색법 사용해 도달 불가능한 코드나 무한 루프 탐지	DFS or BFS 수행 시간 (V+E)에 비례
Garbage Collection 	동적 할당 객체를 정점으로, 이들 간 참조를 간선으로 모델링. Digraph 탐색법 사용해 참조되지 않은 객체 확인, 메모리에서 삭제	DFS or BFS 수행 시간 (V+E)에 비례
Web Crawling 	웹 사이트를 정점으로, 이들 간 링크를 간선으로 모델링. Root 사이트를 시작점으로 BFS 사용해 링크 따라 도달 가능한 사이트 발견하고 수집해 검색 엔진에 활용	BFS or Multi-source BFS 수행 시간 (V+E)에 비례
Topological Sort 	$v \rightarrow w$ 간선이 v 수행 후 w 수행하는 관계로 보고, DFS 사용해 먼저 수행해야 하는 순서대로 정점을 정렬	DFS 수행 시간 (V+E)에 비례
SCC(Strongly-Connected Component) 탐색 	양방향 경로 있는 정점 집합 찾기 위해 ① DFS로 G^R 의 topological order 찾은 후 ② 찾은 순서대로 DFS or BFS 수행해 도달 가능한 정점 모두를 같은 component로 확인	DFS → (DFS or BFS) 수행 시간 (V+E)에 비례 Kosaraju-Sharir 알고리즘

간단하지 않아 보이는 많은 그래프 탐색 문제가 linear time에 해결 가능