2025 겨울방학 알고리즘 스터디

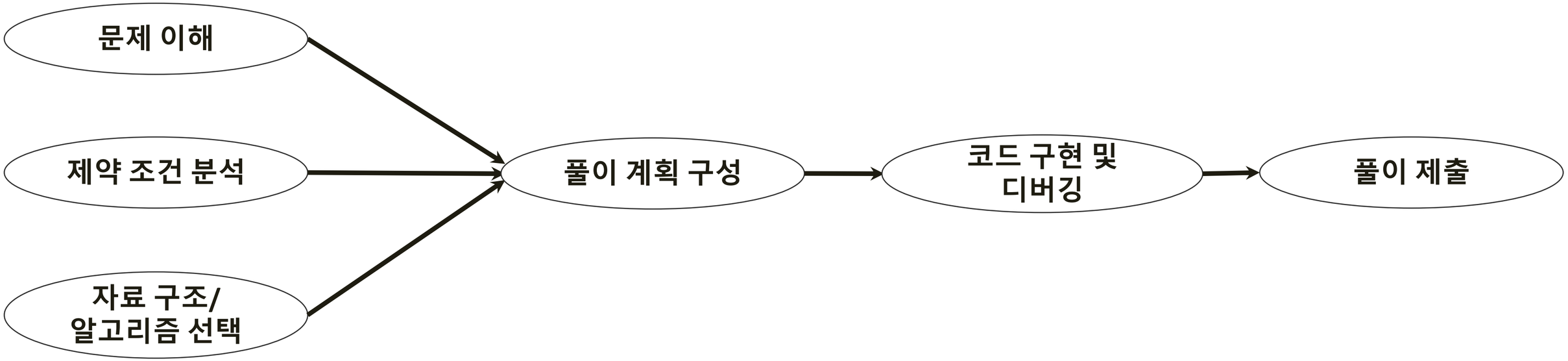
위상 정렬

컴퓨터 공학과 20230546 서보경

목차

유향 비순환 그래프(DAG)란?

유향 비순환 그래프(Directed Acyclic Graph; DAG) 는 방향이 있는 간선으로 이루어진 그래프이다. 사이클이 없는 구조라는 게 가장 큰 특징이다. 간선을 따라 이동할 때 다시 이전 정점으로 돌아오는 일이 절대 없어서, 그래프 안에서 정점 간의 순서가 항상 유지된다. 이런 구조 덕분에 DAG는 작업의 의존성을 표현하거나, 정점의 처리 순서를 결정하는 데 자주 사용된다.



일반적인 알고리즘 문제를 푸는 방식을 DAG형식으로 나타냈을 때

위상 정렬 이란?

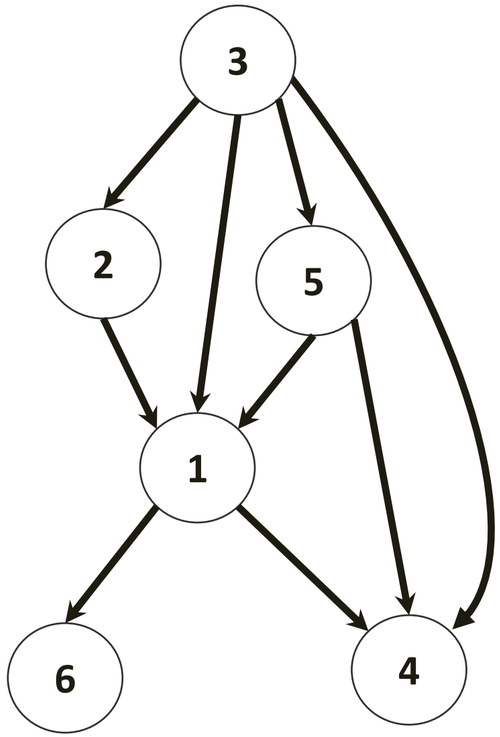
위상 정렬(Topological Sort) 은 **유향 비순환 그래프(DAG)** 의 정점들을 간선의 방향을 거스르지 않고 나열하는 정렬 방식이다. 위상 정렬의 가장 큰 특징은 모든 정점의 순서가 유지되며, 특정 정점이 반드시 먼저 처리되어야 한다는 규칙을 따른다는 점이다. 이러한 구조는 작업의 의존성을 고려한 순서 결정에 사용되며, 작업 스케줄링, 강의 선수 과목 정리, 빌드 의존성 해결 등에서 활용된다.

문제를 풀기 위해선?

1 – 문제를 이해한다.
2 – 제약 조건을 분석 한다.
3 – 자료 구조 및 알고리즘 선택을 한다
4 – 풀이 계획을 구성한다
5 – 코드를 구현하면서 디버깅을 한다
6 – 작성한 코드를 제출한다.

아까 본 DAG를 위상 정렬로 나열했을 때 나올 수 있는 케이스. 다른 방법이 나올 수도 있다.

위상 정렬



방문 순서

Queue

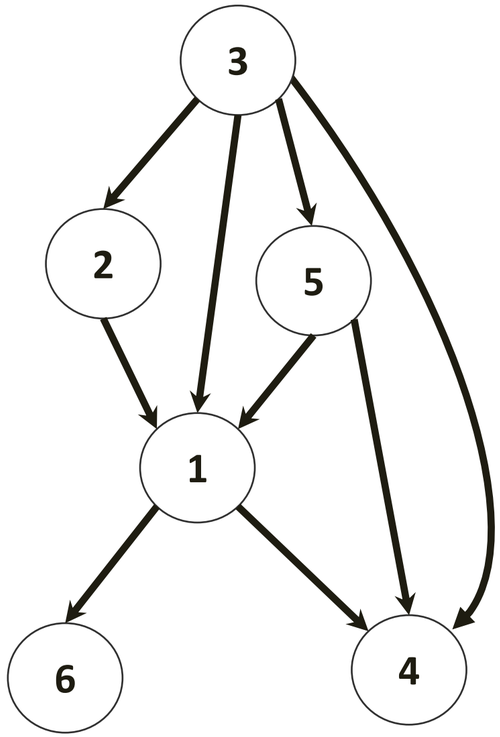
정점	1	2	3	4	5	6
진입 차수	3	1	0	3	1	1

정점들의 차수

일반적으로 위상 정렬은 BFS로 이루어진다. 고로 DAG인 그래프와 Queue가 필요하다.

또한, 각 정점의 진입 정점을 모두 기록 해주어야 한다.
만약 어떤 정점에 대해서 진입 정점이 없다면 선행 정점이 없거나 이미 처리되었다는 뜻이기 때문에 그 정점들을 대상으로 탐색을 진행할 수 있다.

위상 정렬



방문 순서

3						

Queue

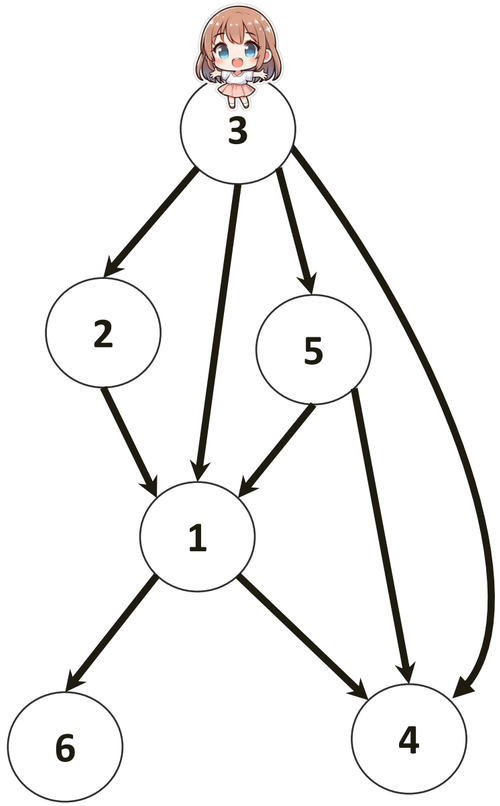
정점	1	2	3	4	5	6
진입 차수	3	1	0	3	1	1

정점들의 진입 차수

먼저, 본격적인 시작전에 진입 차수가 0인 정점을 모조리 큐에 넣어준다.

진입 차수가 0이라는 것은 선행 정점이 이미 처리되었거나 없다는 것을 의미하기 때문에
현재 정점을 처리하는데 모순이 없다는 걸 보장한다.
진입 차수가 0인 정점은 3번 정점밖에 없다. 큐에 3번 정점을 넣어준다.

위상 정렬



3						

방문 순서

Queue

정점	1	2	3	4	5	6
진입 차수	3	1	0	3	1	1

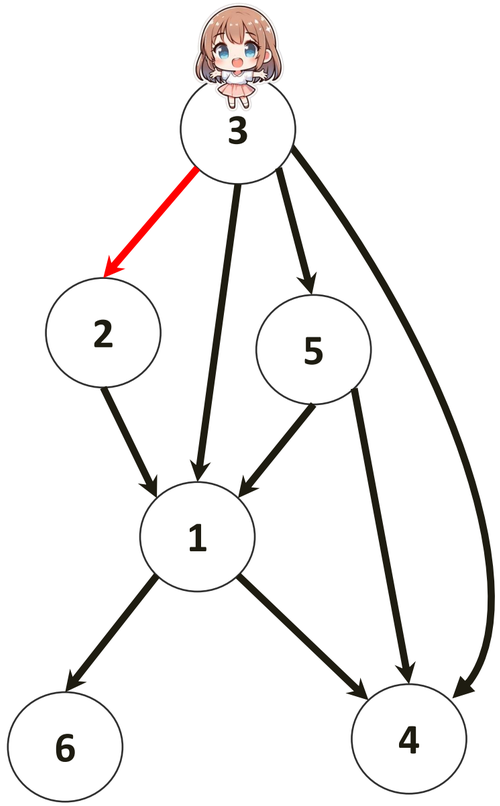
정점들의 진입 차수

큐의 앞쪽에서 원소를 꺼낸다.

꺼낸 정점은 3번이다. 방문 순서에 3을 push 해준다.

이후, 3번 정점과 연결된 정점들의 진입 차수를 하나씩 줄이면서 조건을 확인할 것이다.

위상 정렬



3						

방문 순서

2						

Queue

정점	1	2	3	4	5	6
진입 차수	3	0	0	3	1	1

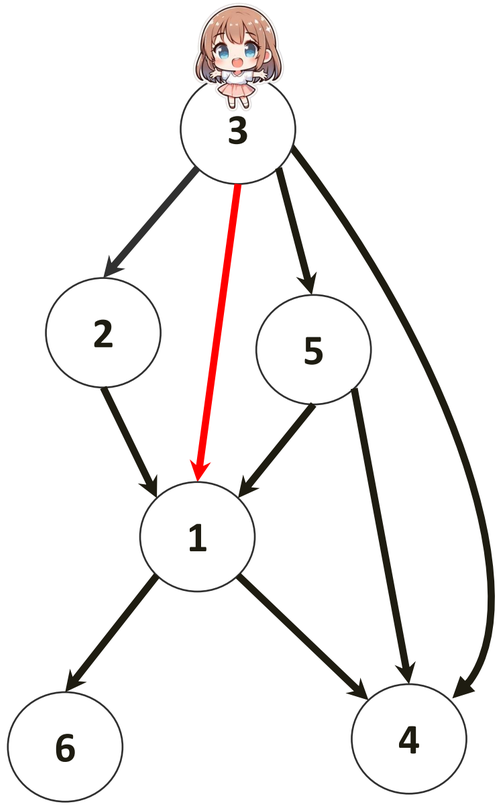
정점들의 진입 차수

3번 정점에서 갈 수 있는 정점인 2번 정점이다.

2번 정점의 진입 차수를 하나 줄여준다.(왜냐하면 현재 정점은 이미 처리 되었으니 유효 X)

2번 정점의 남은 진입 차수는 0이다. 고로 현재 정점을 처리할 수 있다. 2를 큐에 넣는다.

위상 정렬



3						

방문 순서

2						

Queue

정점	1	2	3	4	5	6
진입 차수	2	0	0	3	1	1

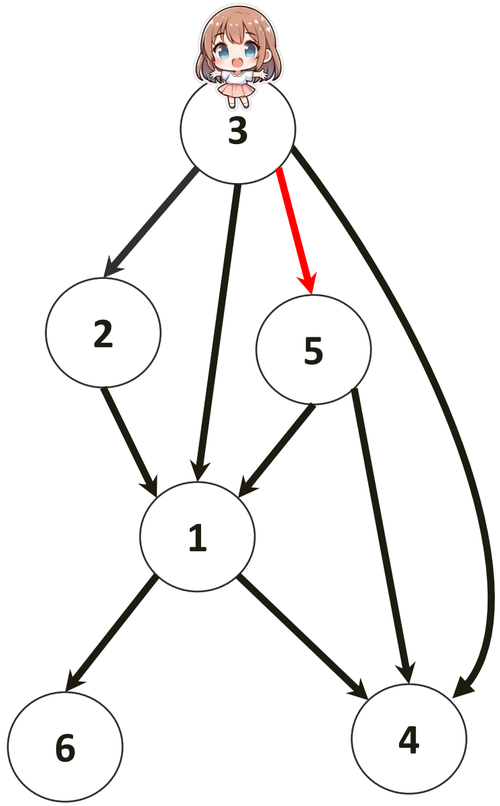
정점들의 진입 차수

3번 정점에서 갈 수 있는 정점인 1번 정점이다.

1번 정점의 진입 차수를 하나 줄여준다.

1번 정점의 남은 차수는 2이다. 차수가 남아 있어 처리할 수 없으므로 다음으로 넘어간다.

위상 정렬



3						

방문 순서

2	5					

Queue

정점	1	2	3	4	5	6
진입 차수	2	0	0	3	0	1

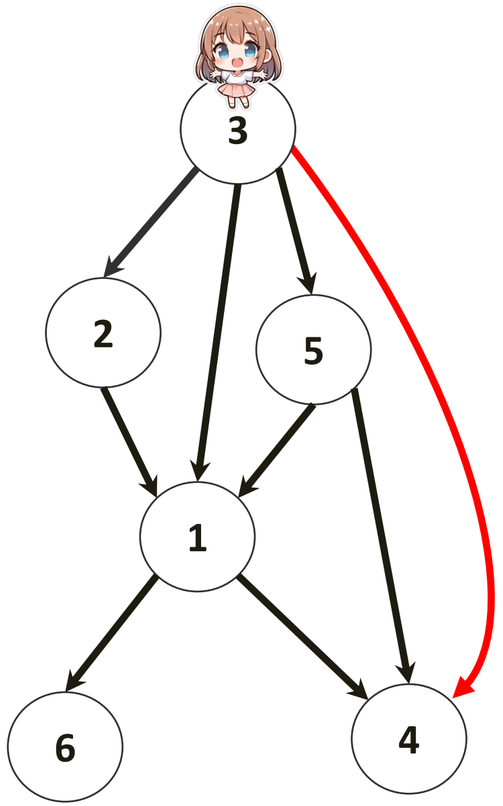
정점들의 진입 차수

3번 정점에서 갈 수 있는 정점인 5번 정점이다.

5번 정점의 진입 차수를 하나 줄여준다.

5번 정점의 남은 진입 차수는 0이다. 고로 현재 정점을 처리할 수 있다. 5를 큐에 넣는다.

위상 정렬



3						

방문 순서

2	5					

Queue

정점	1	2	3	4	5	6
진입 차수	2	0	0	2	0	1

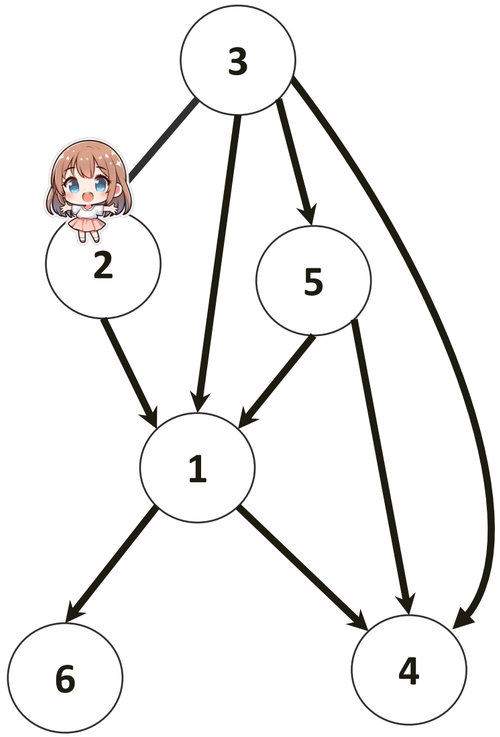
정점들의 진입 차수

3번 정점에서 갈 수 있는 정점인 4번 정점이다.

4번 정점의 진입 차수를 하나 줄여준다.

4번 정점의 남은 차수는 2이다. 차수가 남아 있어 처리할 수 없다. 현재 정점 탐색이 모두 끝났다.

위상 정렬



3	2					

방문 순서

5						

Queue

정점	1	2		4	5	6
진입 차수	2	0		2	0	1

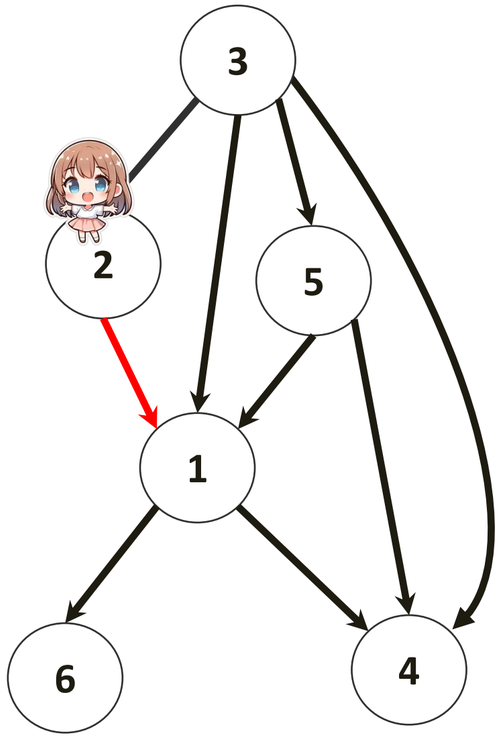
정점들의 진입 차수

큐의 앞쪽에서 원소를 꺼낸다.

꺼낸 정점은 2번이다. 방문 순서에 2을 push 해준다.

이후, 2번 정점과 연결된 정점들의 진입 차수를 하나씩 줄이면서 조건을 확인할 것이다.

위상 정렬



3	2					

방문 순서

5						

Queue

정점	1	2		4	5	6
진입 차수	1	0		2	0	1

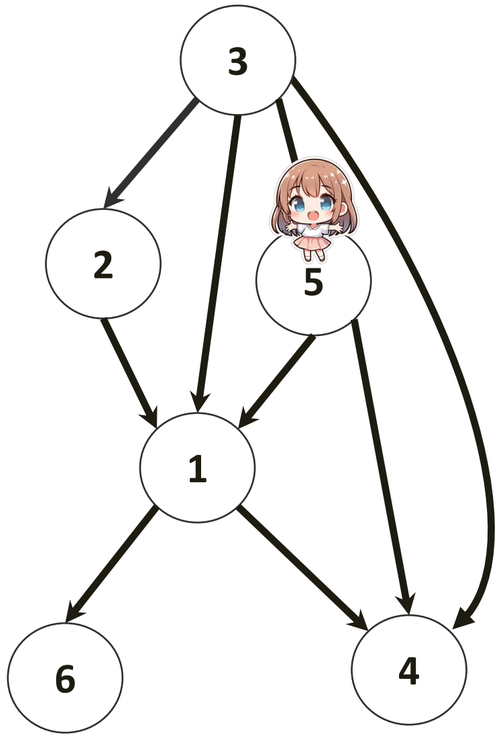
정점들의 진입 차수

2번 정점에서 갈 수 있는 정점인 1번 정점이다.

1번 정점의 진입 차수를 하나 줄여준다.

4번 정점의 남은 차수는 2이다. 차수가 남아 있어 처리할 수 없다. 현재 정점 탐색이 모두 끝났다.

위상 정렬



3	2	5				

방문 순서

Queue

정점	1			4	5	6
진입 차수	1			2	0	1

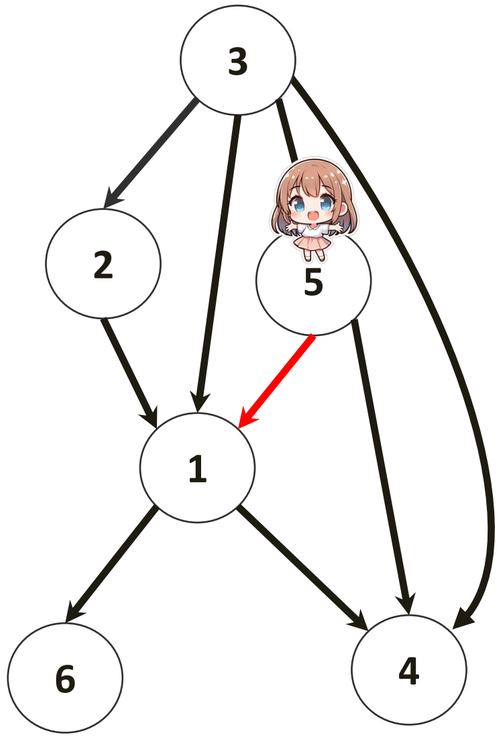
정점들의 진입 차수

큐의 앞쪽에서 원소를 꺼낸다.

꺼낸 정점은 5번이다. 방문 순서에 5을 push 해준다.

이후, 5번 정점과 연결된 정점들의 진입 차수를 하나씩 줄이면서 조건을 확인할 것이다.

위상 정렬



3	2	5				

방문 순서

1						

Queue

정점	1			4	5	6
진입 차수	0			2	0	1

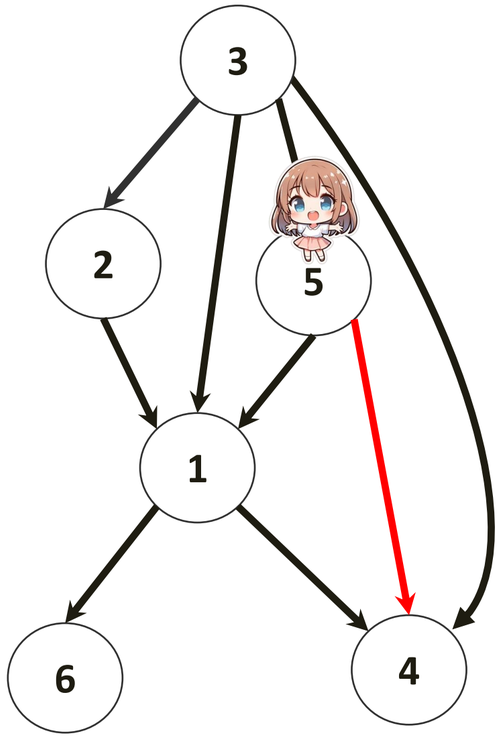
정점들의 진입 차수

5번 정점에서 갈 수 있는 정점인 1번 정점이다.

1번 정점의 진입 차수를 하나 줄여준다.

1번 정점의 남은 차수는 0이다. 고로 현재 정점을 처리할 수 있다. 1를 큐에 넣는다.

위상 정렬



3	2	5				

방문 순서

1						

Queue

정점	1			4	5	6
진입 차수	0			1	0	1

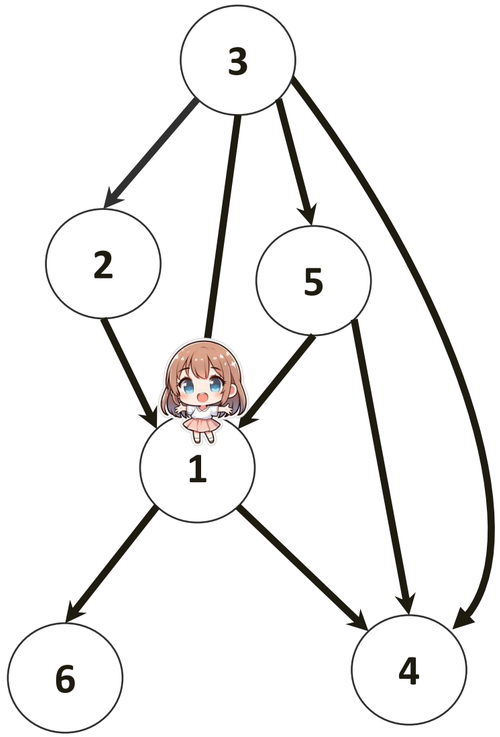
정점들의 진입 차수

5번 정점에서 갈 수 있는 정점인 4번 정점이다.

4번 정점의 진입 차수를 하나 줄여준다.

4번 정점의 남은 차수는 1이다. 차수가 남아 있어 처리할 수 없다. 현재 정점 탐색이 모두 끝났다.

위상 정렬



3	2	5	1			

방문 순서

Queue

정점	1			4		6
진입 차수	0			1		1

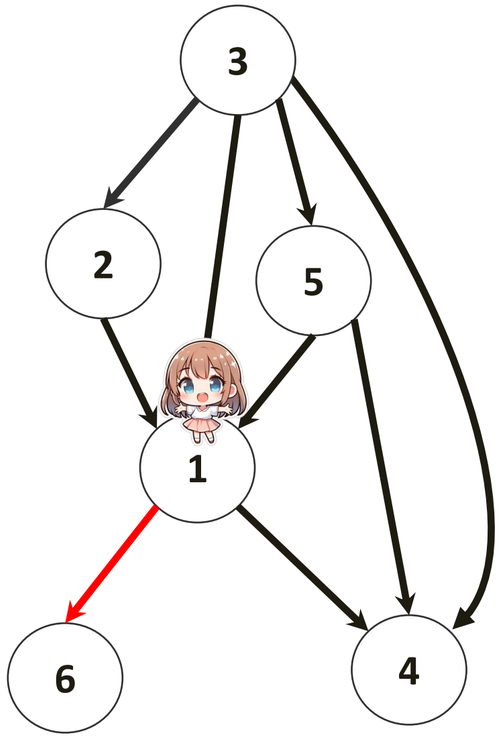
정점들의 진입 차수

큐의 앞쪽에서 원소를 꺼낸다.

꺼낸 정점은 1번이다. 방문 순서에 1을 push 해준다.

이후, 1번 정점과 연결된 정점들의 진입 차수를 하나씩 줄이면서 조건을 확인할 것이다.

위상 정렬



3	2	5	1			

방문 순서

6						

Queue

정점	1			4		6
진입 차수	0			1		0

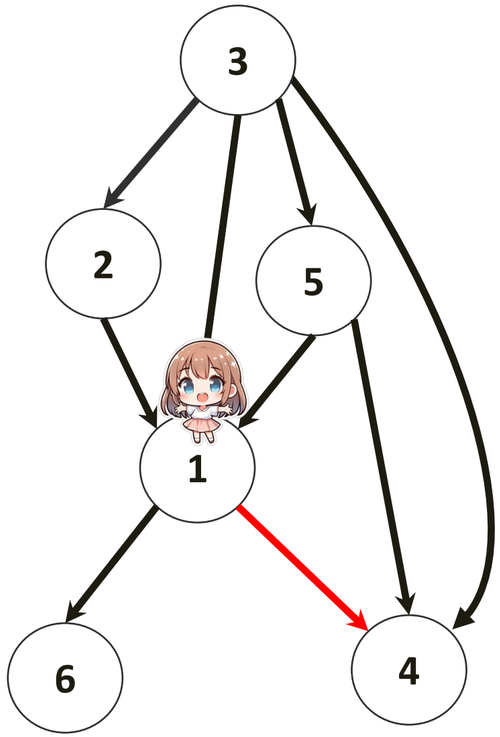
정점들의 진입 차수

1번 정점에서 갈 수 있는 정점인 6번 정점이다.

6번 정점의 진입 차수를 하나 줄여준다.

6번 정점의 남은 차수는 0이다. 고로 현재 정점을 처리할 수 있다. 6를 큐에 넣는다.

위상 정렬



3	2	5	1			

방문 순서

6	4					

Queue

정점	1			4		6
진입 차수	0			0		0

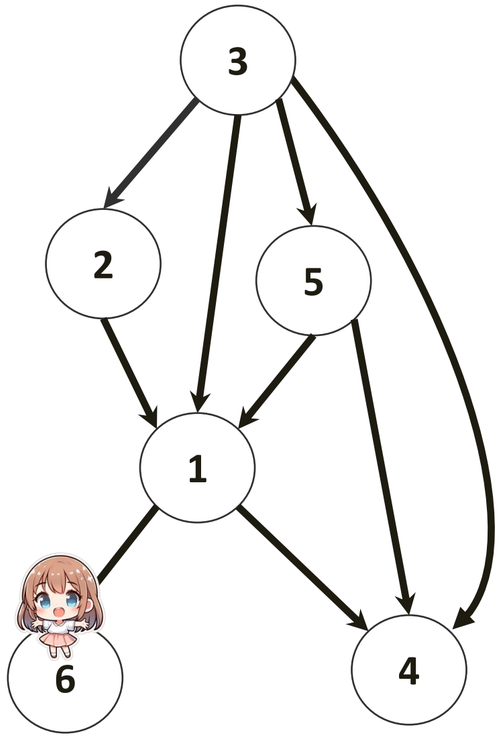
정점들의 진입 차수

1번 정점에서 갈 수 있는 정점인 4번 정점이다.

4번 정점의 진입 차수를 하나 줄여준다.

4번 정점의 남은 차수는 0이다. 고로 현재 정점을 처리할 수 있다. 6를 큐에 넣는다. 정점 탐색 끝.

위상 정렬



3	2	5	1	6		

방문 순서

4						

Queue

정점				4		6
진입 차수				0		0

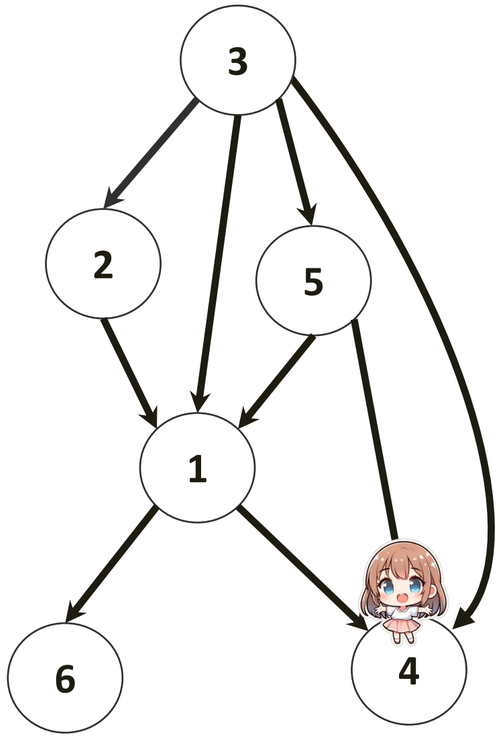
정점들의 진입 차수

큐의 앞쪽에서 원소를 꺼낸다.

꺼낸 정점은 6번이다. 방문 순서에 6을 push 해준다.

연결된 간선이 없으므로 아무 것도 하지 말고 넘어간다.

위상 정렬



3	2	5	1	6	4	

방문 순서

Queue

정점				4		
진입 차수				0		

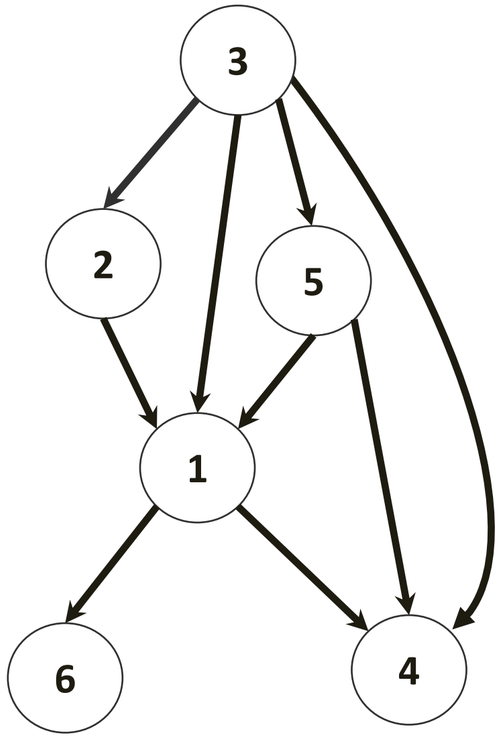
정점들의 진입 차수

큐의 앞쪽에서 원소를 꺼낸다.

꺼낸 정점은 4번이다. 방문 순서에 4을 push 해준다.

연결된 간선이 없으므로 아무 것도 하지 말고 넘어간다. 위상 정렬이 전부 이루어 졌다.

위상 정렬



3	2	5	1	6	4	

방문 순서

Queue

정점						
진입 차수						

정점들의 진입 차수

위상 정렬은 난이도가 높지 않으며 구현 난이도가 단순한 편이다.

또한 위상 정렬을 통해 사이클을 감지할 수 있다.
DAG의 특성을 기억하는가? 사이클이 없는 방향 그래프이다. 그러나 만약 그래프에 사이클이 있으면 어떨까?

위상 정렬

정점	A	B
진입 차수	1	1

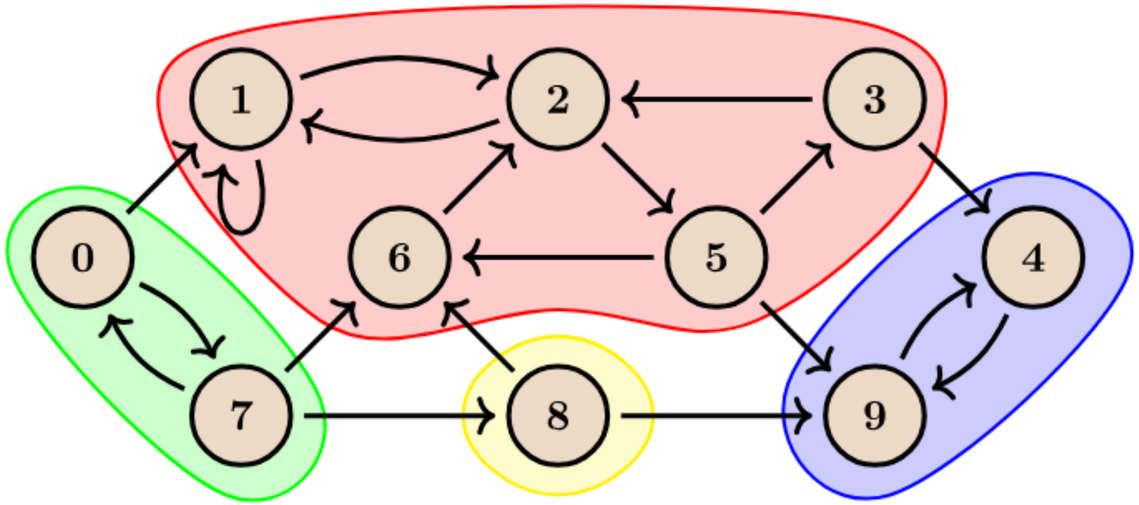
정점들의 진입 차수

A정점과 B정점은 서로 사이클을 이루고 있다.

만약 위상 정렬로 처리를 해야 한다면, 저 두 정점은 처리가 불가능하다. 각 두개의 노드들이 선행관계과 후행관계로 얽혀 있기 때문이다.
만약 어떤 그래프가 연결 그래프인데(컴포넌트가 1개) 위상 정렬을 실시 했을 때 **처리가 되지 않은 정점이 있다면** 그 그래프는 DAG가 아닌 일반 그래프이다.
(혹은 위상 정렬을 잘못 짰거나...?)

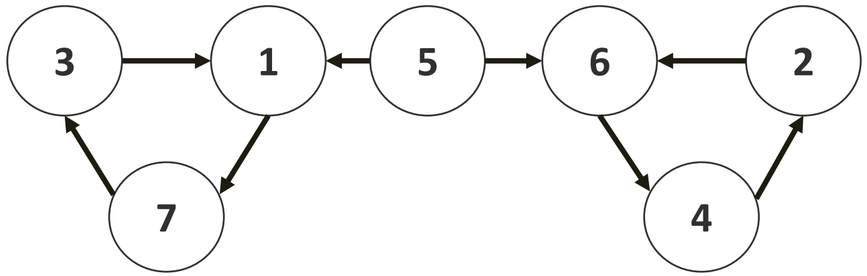
강한 연결 요소란?

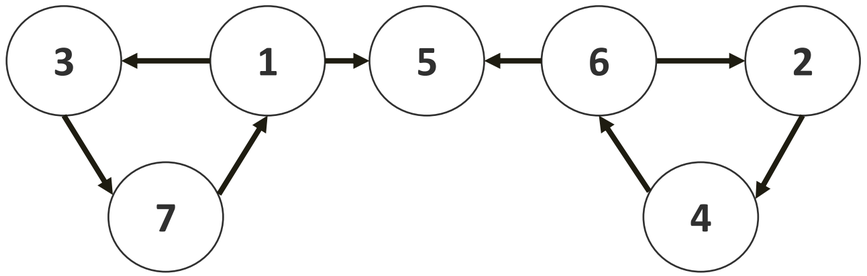
강한 연결 요소(Strongly Connected Component; SCC) 는 유향 그래프에서 모든 정점이 서로 도달 가능한 **최대 크기**의 부분 그래프를 의미한다. SCC의 가장 큰 특징은 한 정점에서 출발하면 같은 SCC 내의 모든 정점으로 이동할 수 있으며, 반대로도 이동할 수 있다는 점이다. 이러한 구조 덕분에 SCC는 그래프의 강한 연결성을 분석하거나, 축소된 DAG 형태로 변환하여 그래프를 단순화하는 데 자주 사용된다.



강한 연결 요소(SCC)는 서로 도달 가능한 정점들의 최대 집합을 의미한다.
위 그래프에선 총 4개의 SCC가 존재한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	X	X	X	X	X	X	X

visited

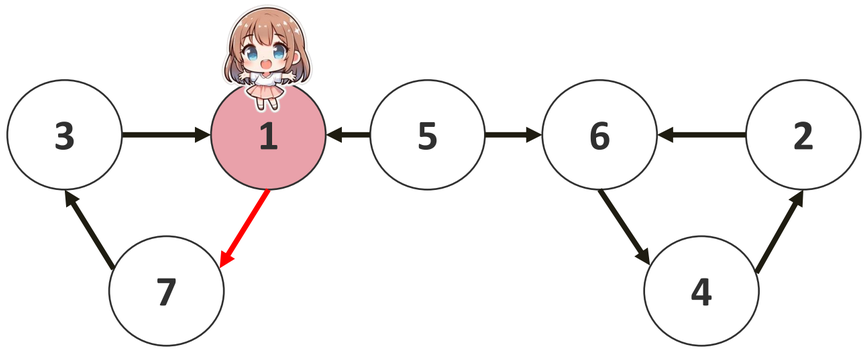
Stack

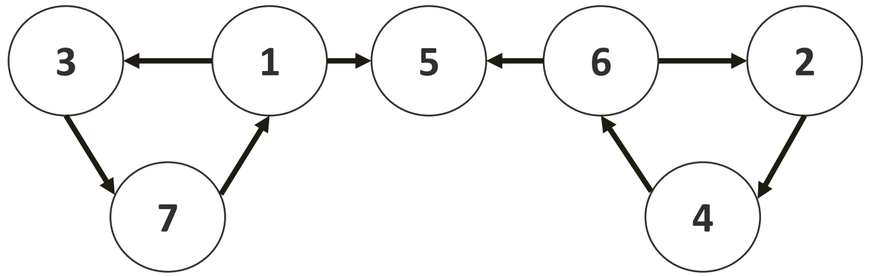
강한 연결 요소를 찾는 코사라주 알고리즘을 소개하겠다.

스택과 깊이 우선 탐색(dfs)을 이용하여 강한 연결 요소를 찾는 방법이다.

이를 위해 정방향 그래프와 역방향 그래프, 스택과 방문처리 배열을 준비한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	X	X	X	X	X	X

visited

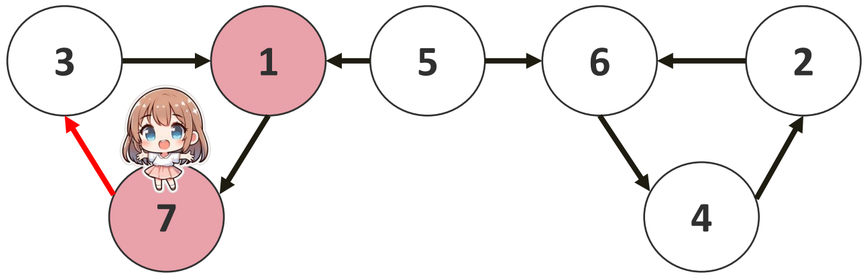
Stack

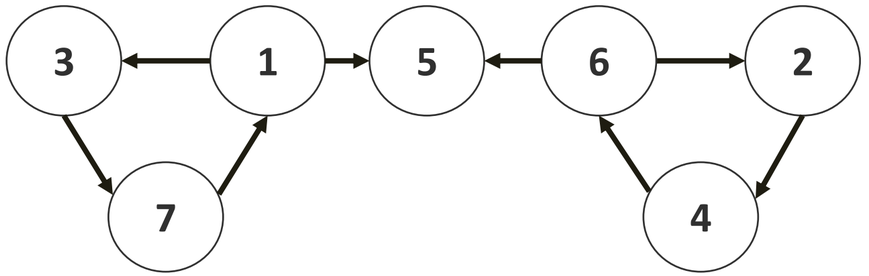
먼저 정방향 그래프를 탐색할 것이다. 1번 정점부터 dfs로 탐색을 하겠다.

현재 정점은 1번 정점이다. 1번 정점에서는 갈 수 있는 정점인 7번이 있다.

현재 노드를 방문 처리하고 7번 정점으로 이동 한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	X	X	X	X	X	O

visited

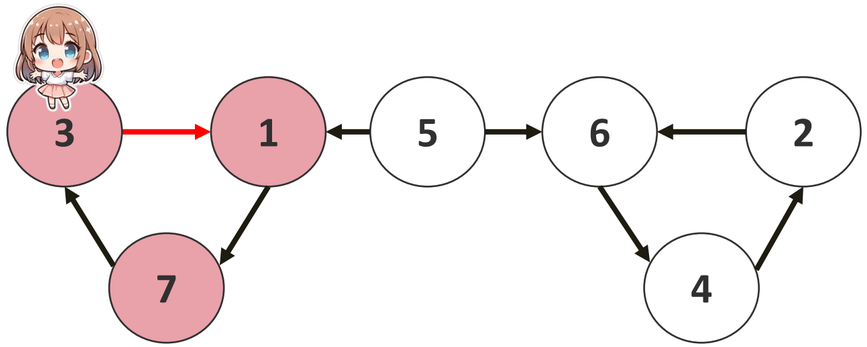
Stack

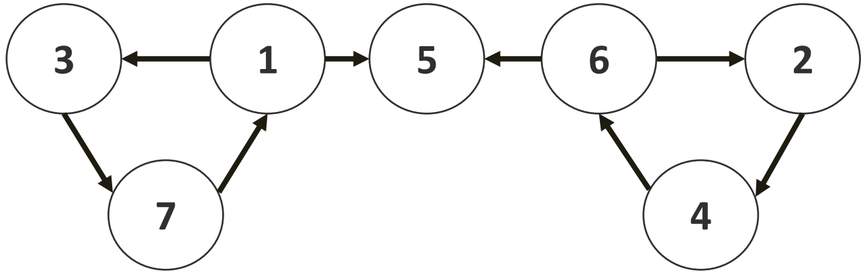
현재 정점은 7번 정점이다.

7번 정점에서는 갈 수 있는 정점인 3번이 있다.

현재 노드를 방문 처리하고 3번 정점으로 이동 한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	X	O	X	X	X	O

visited

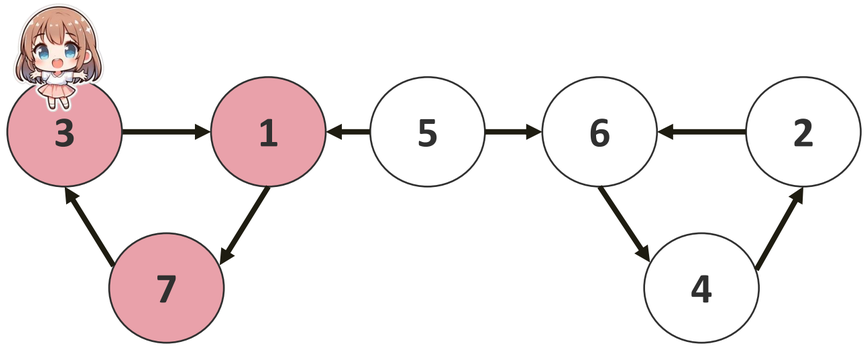
Stack

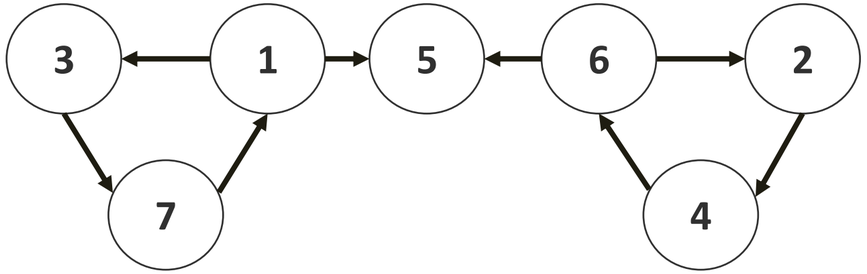
현재 정점은 3번 정점이다.

3번 정점에서는 갈 수 있는 정점이 1번이지만 이미 방문 된 상태이다. 고로 갈 수 있는 정점이 없다.

이 때, dfs가 종료되게 되는데 복귀 시 현재 방문한 노드들을 거슬러 올라간다는 특징이 있다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	X	O	X	X	X	O

visited

3						

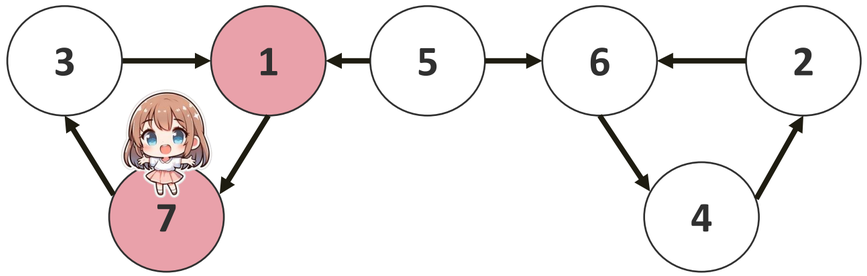
Stack

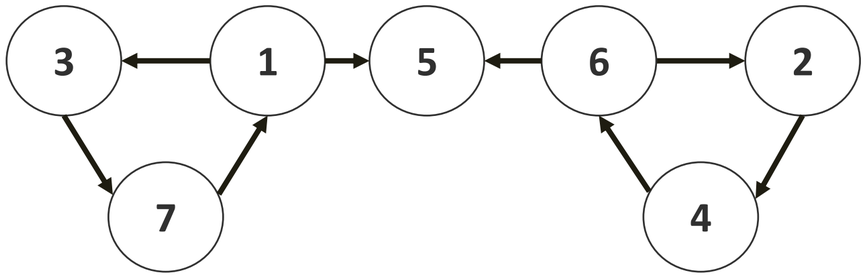
복귀하기 전에 현재 정점인 3번을 스택에 넣어준다.

그 다음 return을 하여 이전 노드로 복귀 한다.

(재귀 함수 기준이며, 스택 기준이면 pop 해서 이전 노드로 복귀한다.)

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	X	O	X	X	X	O

visited

3	7					

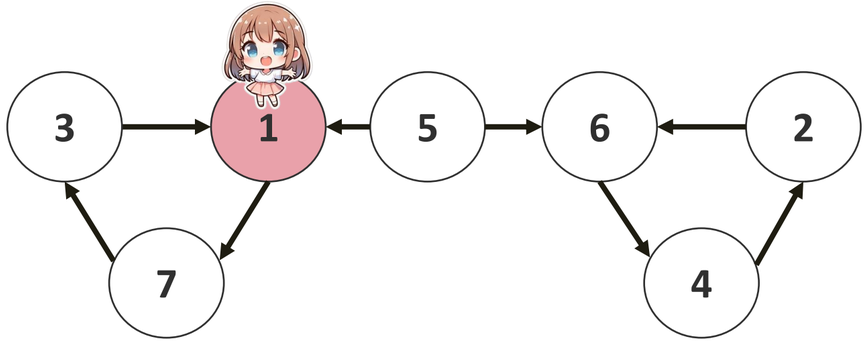
Stack

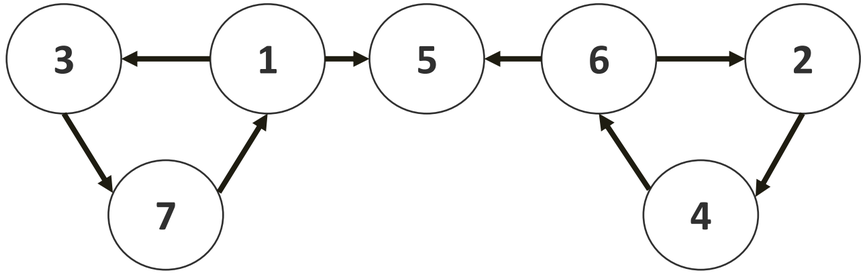
현재 정점은 7번이다.

7번 정점을 스택에 넣어준다.

return 하여 이전 노드로 복귀 한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	X	O	X	X	X	O

visited

3	7	1				

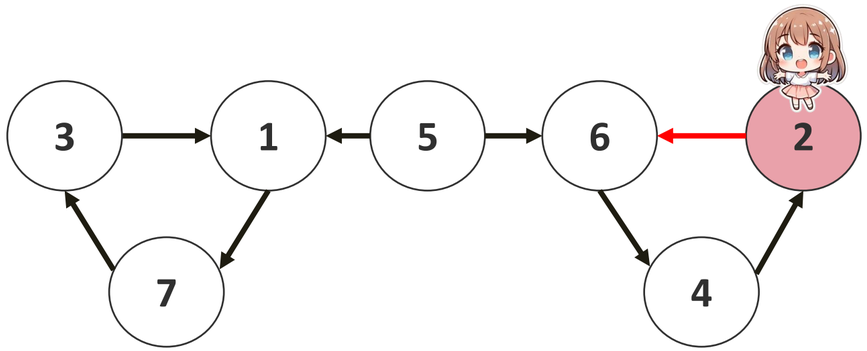
Stack

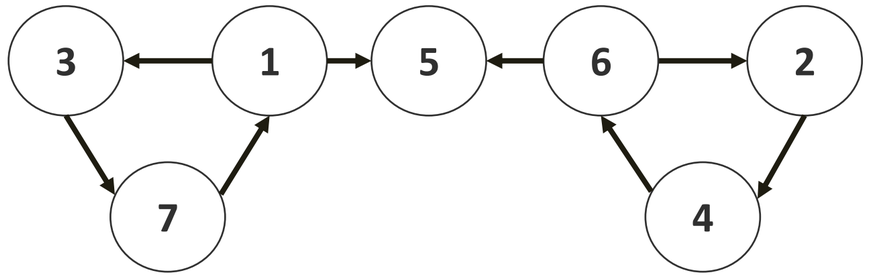
현재 정점은 1번이다.

1번 정점을 스택에 넣어준다.

원점으로 돌아 왔으므로 다음 방문 되지 않은 노드에 대해서도 탐색을 시작하자.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	X	X	X	O

visited

3	7	1				

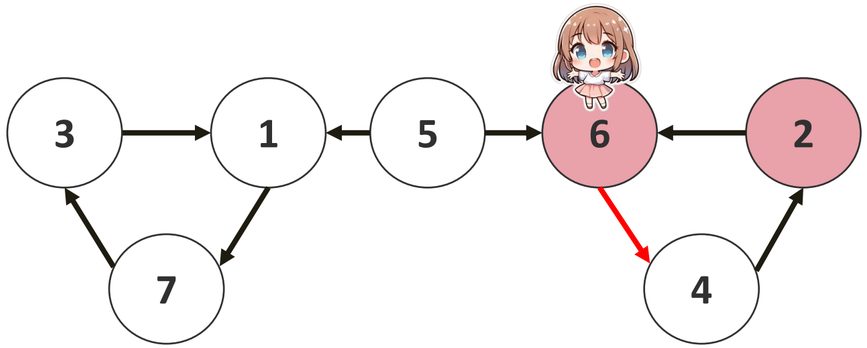
Stack

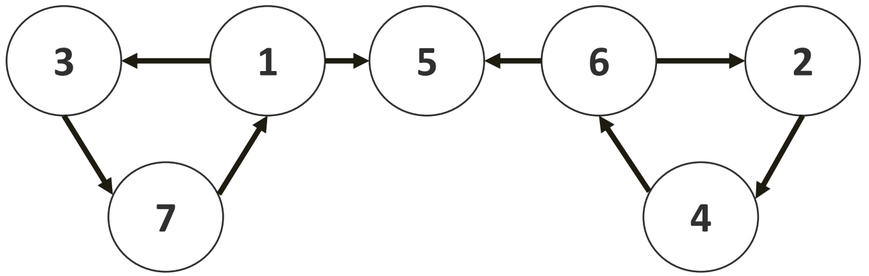
이번에는 2번 정점부터 dfs로 탐색을 하겠다.

현재 정점은 2번 정점이다. 2번 정점에서는 갈 수 있는 정점인 6번이 있다.

현재 노드를 방문 처리하고 6번 정점으로 이동 한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	X	X	O	O

visited

3	7	1				

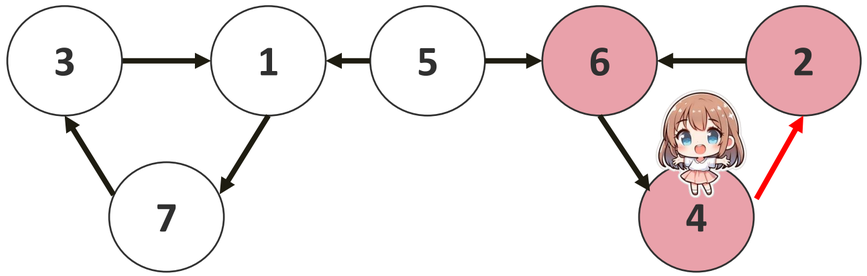
Stack

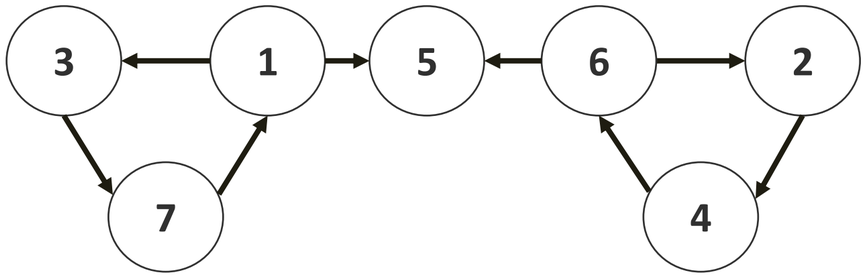
현재 정점은 6번 정점이다.

6번 정점에서는 갈 수 있는 정점인 4번이 있다.

현재 노드를 방문 처리하고 4번 정점으로 이동 한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	X	O	O

visited

3	7	1				

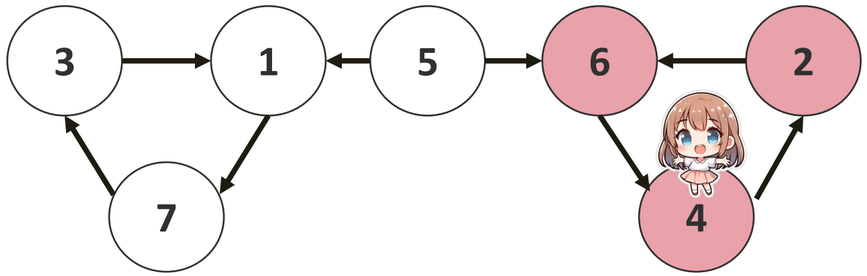
Stack

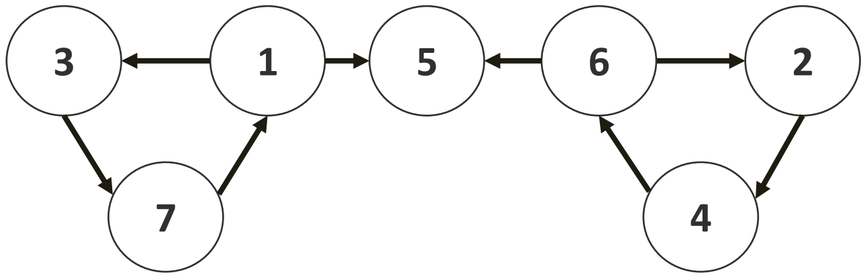
현재 정점은 4번 정점이다.

4번 정점에서는 갈 수 있는 정점이 2번이지만 이미 방문 된 상태이다. 고로 갈 수 있는 정점이 없다.

dfs를 완료 했으니 복귀하면서 정점을 스택에 담을 차례이다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	X	O	O

visited

3	7	1	4			

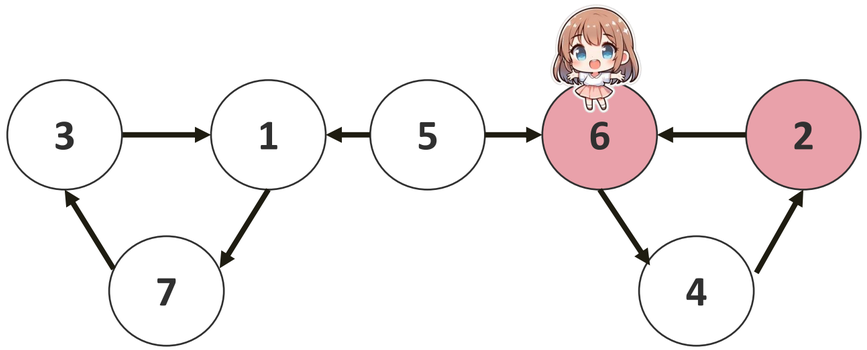
Stack

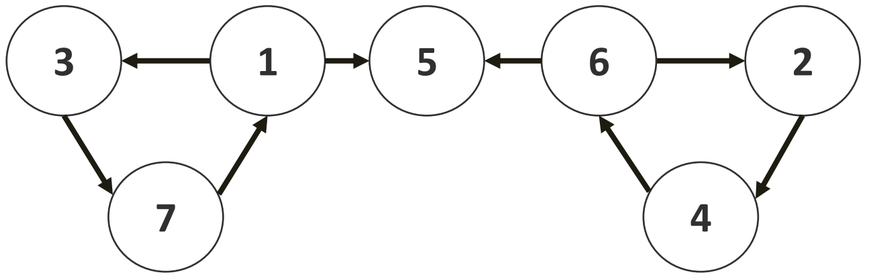
현재 정점은 4번이다.

4번 정점을 스택에 넣어준다.

return 하여 이전 노드로 복귀 한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	X	O	O

visited

3	7	1	4	6		

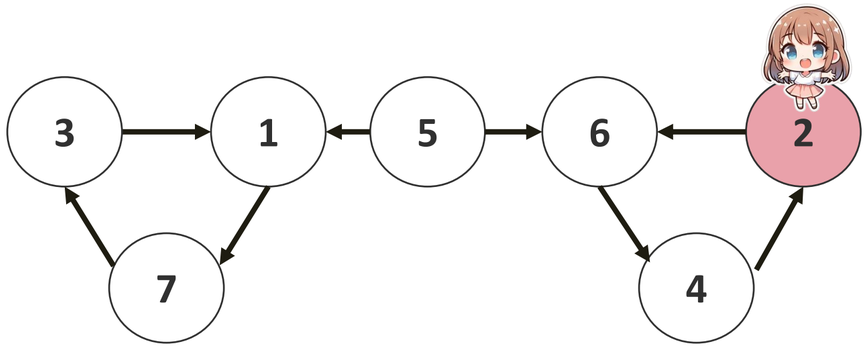
Stack

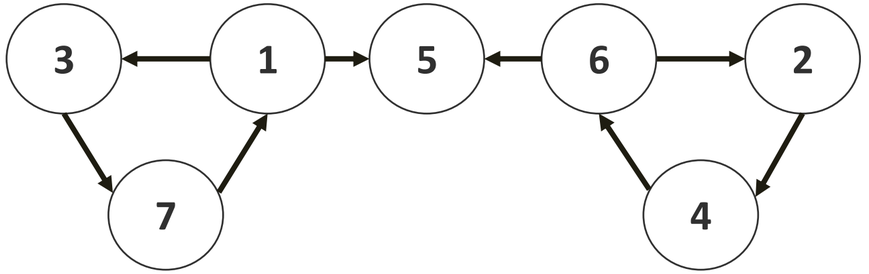
현재 정점은 6번이다.

6번 정점을 스택에 넣어준다.

return 하여 이전 노드로 복귀 한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	X	O	O

visited

3	7	1	4	6	2	

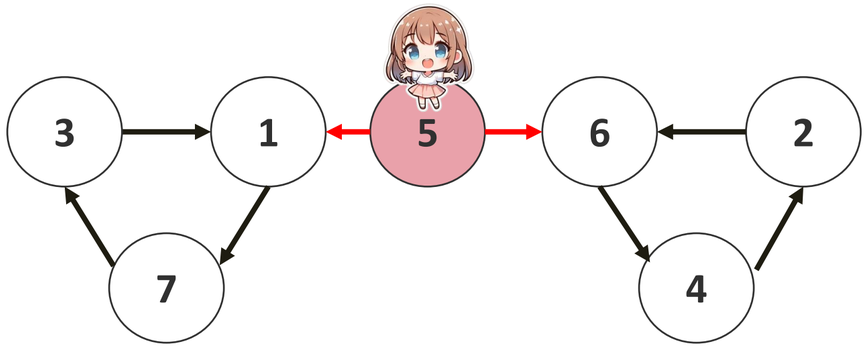
Stack

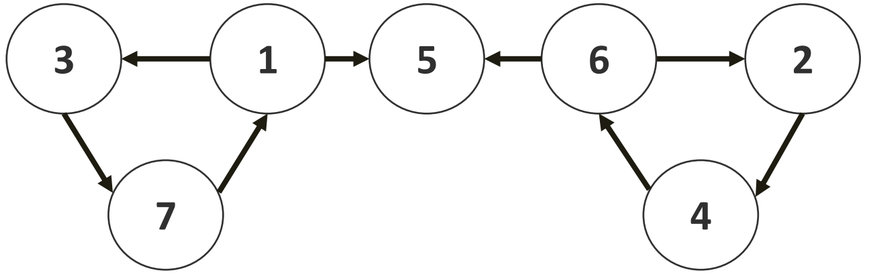
현재 정점은 2번이다.

2번 정점을 스택에 넣어준다.

원점으로 돌아 왔으므로 다음 방문 되지 않은 노드에 대해서도 탐색을 시작하자.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	O	O	O

visited

3	7	1	4	6	2	

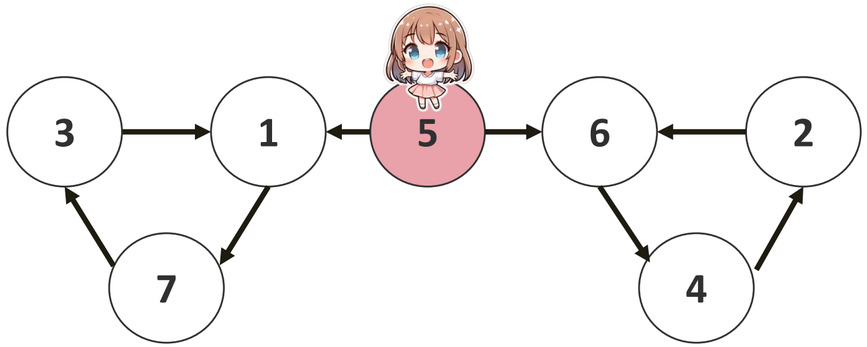
Stack

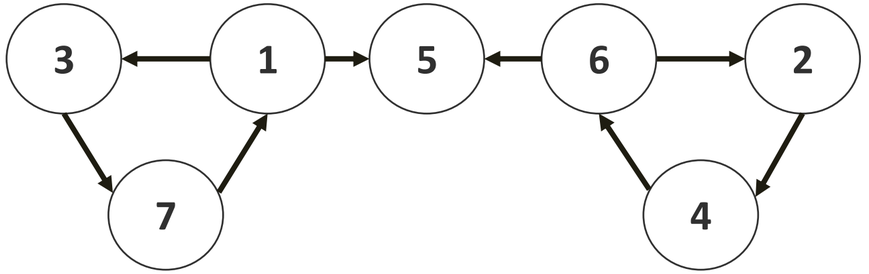
이번에는 5번 정점부터 dfs로 탐색을 하겠다.

현재 정점은 5번 정점이다. 5번 정점에서는 1번, 6번이 있지만 두 정점 모두 방문 상태이다.

현재 노드를 방문 처리했지만 갈 수 있는 노드가 없으므로 dfs가 종료 되었다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	X	X	X	X	X	X	X

visited

3	7	1	4	6	2	5

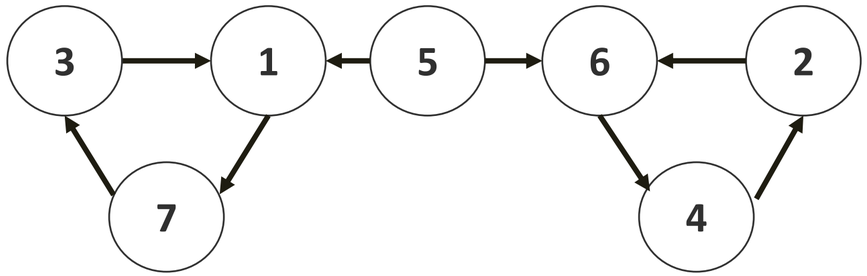
Stack

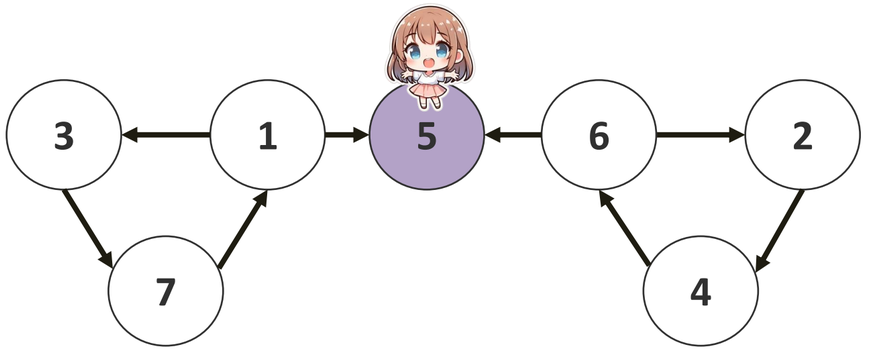
5번 정점을 스택에 넣어준 후에 원점으로 복귀한다.

이제 스택을 이용하여 역방향 그래프를 탐색 할 것이다.

그전에 방문 배열을 전부 X로 초기화 한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	X	X	X	X	O	X	X

visited

3	7	1	4	6	2	

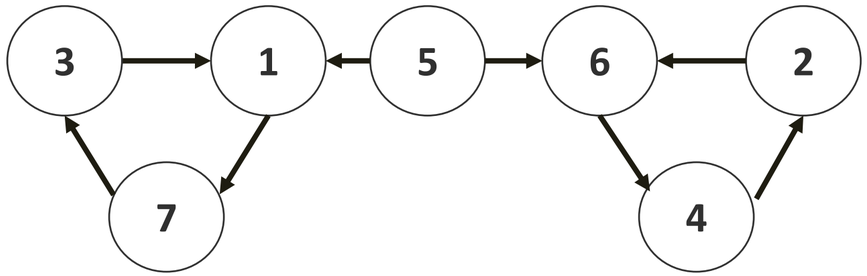
Stack

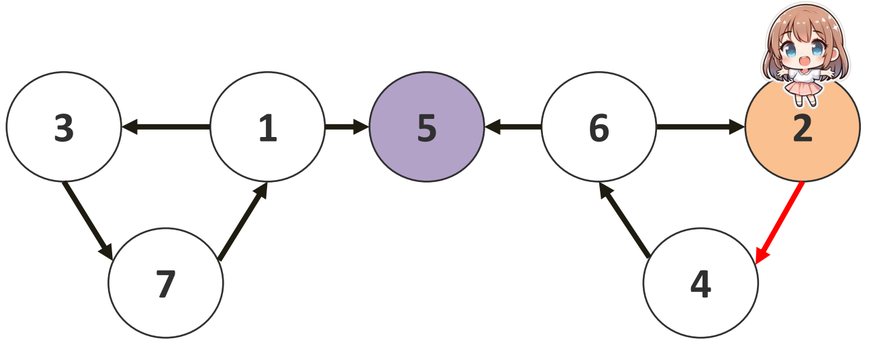
이번에는 스택에서 하나씩 빼서 역방향 그래프를 탐색 할 것이다.

스택의 제일 뒤 원소를 뽑아오자. 이는 5번 정점이며 아직 방문이 되지 않은 상태이다!

그러나 갈 수 있는 노드가 없다. 고로 5번 정점은 자신이 단독인 강한 연결요소이다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	X	O	X	X	O	X	X

visited

3	7	1	4	6		

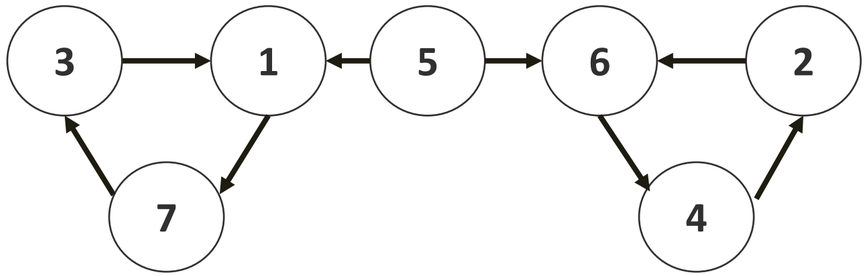
Stack

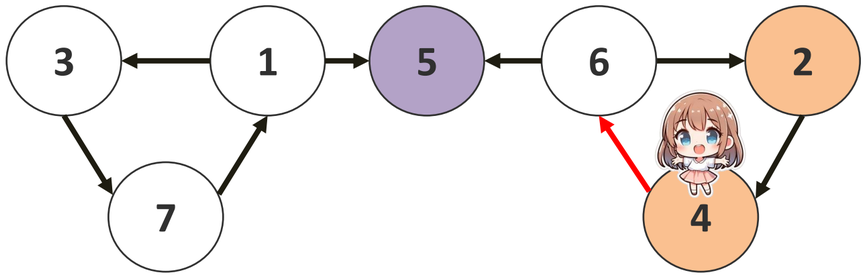
다시 스택의 제일 뒤 원소를 빼준다. 이는 2번 정점이며 아직 방문이 되지 않은 상태이다!

2번 정점을 방문 처리 하고 갈 수 있는 노드를 확인한다.

4번 정점으로 갈 수 있으며, 또한 방문 처리가 안 되어있다. 4번 정점으로 이동한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	X	O	X	O	O	X	X

visited

3	7	1	4	6		

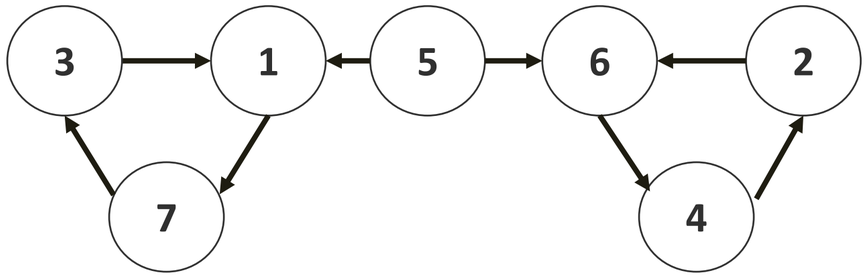
Stack

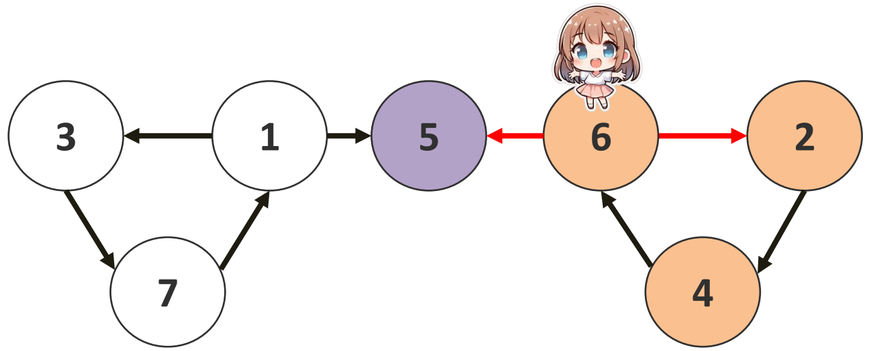
4번 정점에 도착했다. 4번 정점을 방문 처리하고 갈 수 있는 노드를 확인한다.

6번 정점으로 갈 수 있으며, 또한 방문 처리가 안 되어있다.

6번 정점으로 이동한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	X	O	X	O	O	O	X

visited

3	7	1	4	6		

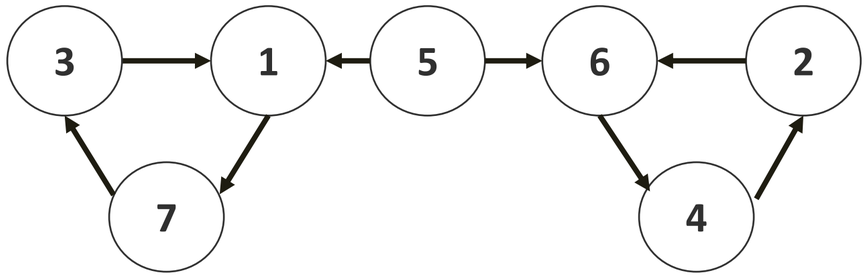
Stack

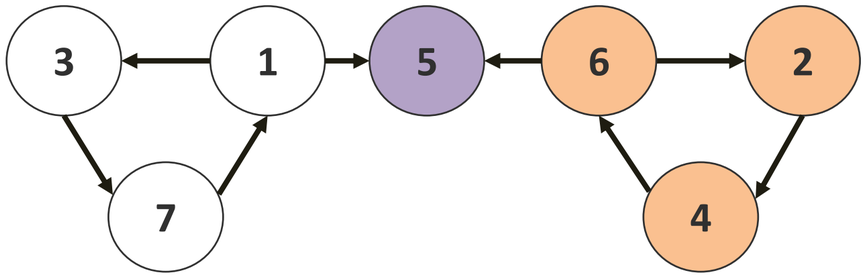
6번 정점에 도착했다. 6번 정점을 방문 처리하고 갈 수 있는 노드를 확인한다.

2번, 5번 정점으로 갈 수 있지만 두 정점 전부 방문 처리가 되어있다. 고로 dfs를 끝내고 복귀한다.

2번, 4번, 6번 정점이 하나의 강한 연결 요소를 이룬다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	X	O	X	O	O	O	X

visited

3	7	1	4			

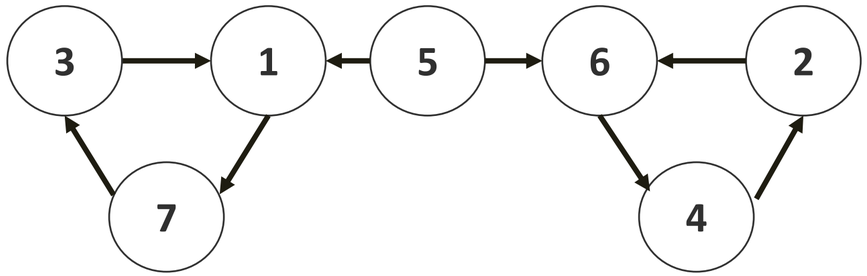
Stack

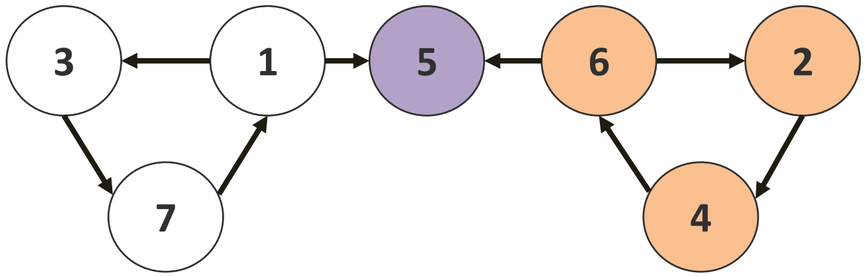
다시 스택의 제일 뒤 원소를 빼준다. 이는 6번 정점이다.

그러나 6번 정점은 이미 방문 된 상태이다.

고로 아무 것도 하지 않는다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	X	O	X	O	O	O	X

visited

3	7	1				

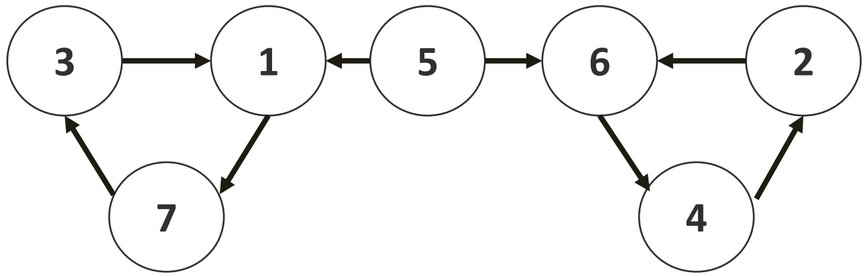
Stack

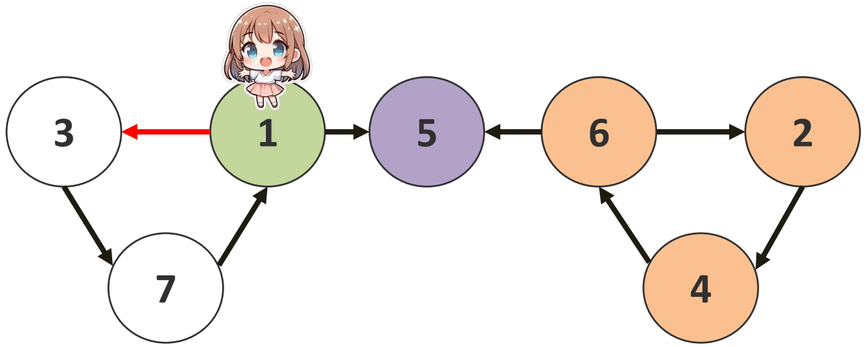
다시 스택의 제일 뒤 원소를 빼준다. 이는 4번 정점이다.

그러나 4번 정점은 이미 방문 된 상태이다.

고로 아무 것도 하지 않는다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	X	O	O	O	X

visited

3	7					

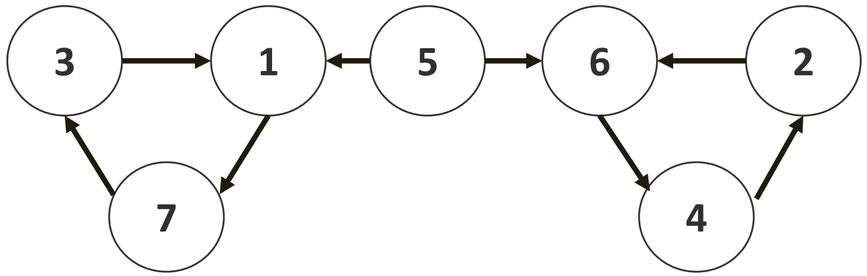
Stack

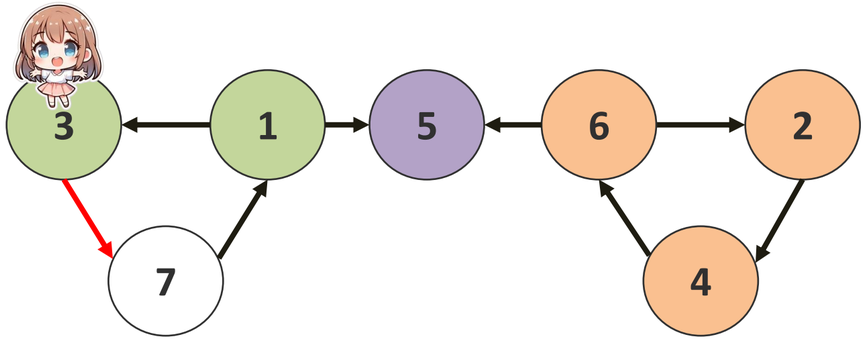
다시 스택의 제일 뒤 원소를 빼준다. 이는 1번 정점이며 아직 방문이 되지 않은 상태이다!

1번 정점을 방문 처리 하고 갈 수 있는 노드를 확인한다.

3번, 5번 정점으로 갈 수 있으나 5번 정점은 이미 방문 된 상태이다. 3번 정점으로 이동한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	O	O	X

visited

3	7					

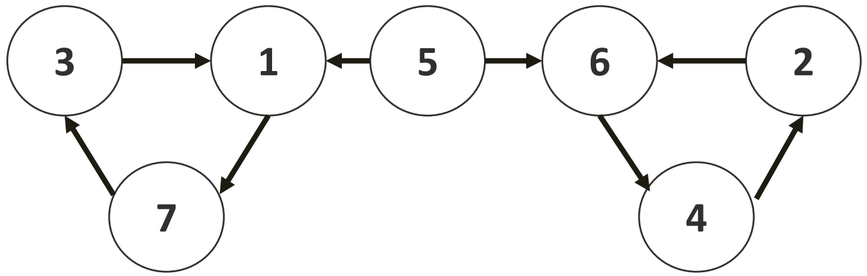
Stack

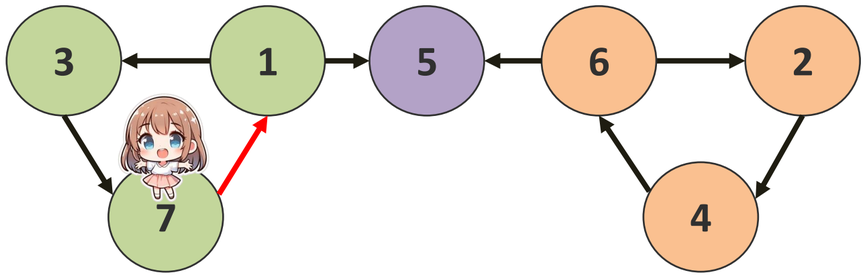
3번 정점에 도착했다. 3번 정점을 방문 처리하고 갈 수 있는 노드를 확인한다.

7번 정점으로 갈 수 있으며, 또한 방문 처리가 안 되어있다.

7번 정점으로 이동한다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	O	O	O

visited

3	7					

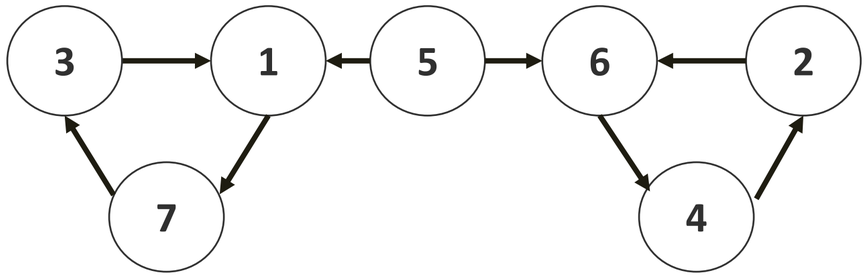
Stack

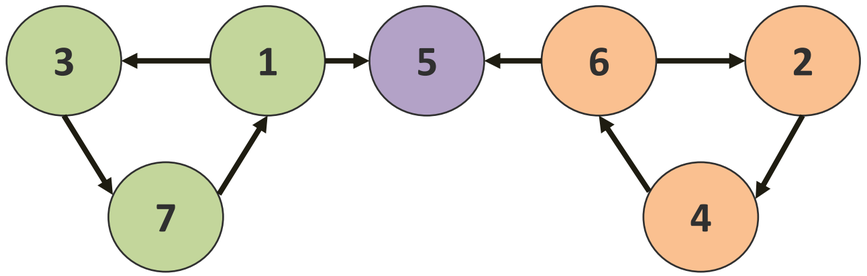
7번 정점에 도착했다. 7번 정점을 방문 처리하고 갈 수 있는 노드를 확인한다.

1번 정점으로 갈 수 있지만 이미 방문 처리가 되어있다. 고로 dfs를 끝내고 복귀한다.

1번, 3번, 7번 정점이 하나의 강한 연결 요소를 이룬다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	O	O	O

visited

3						

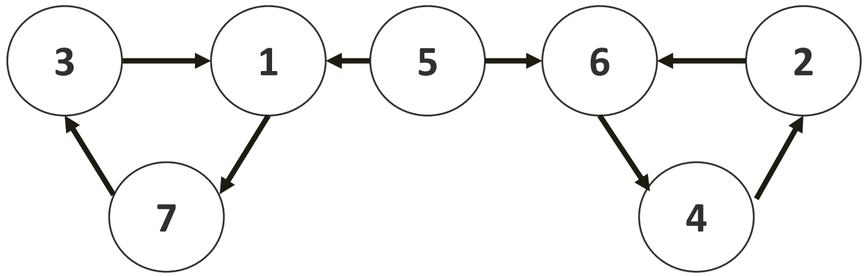
Stack

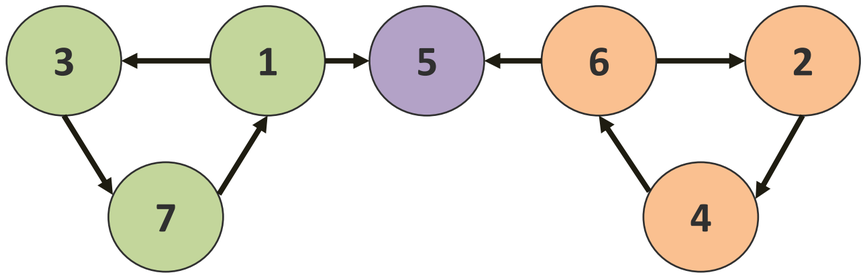
다시 스택의 제일 뒤 원소를 빼준다. 이는 7번 정점이다.

그러나 7번 정점은 이미 방문 된 상태이다.

고로 아무 것도 하지 않는다.

코사라주 알고리즘





정점	1	2	3	4	5	6	7
방문	O	O	O	O	O	O	O

visited

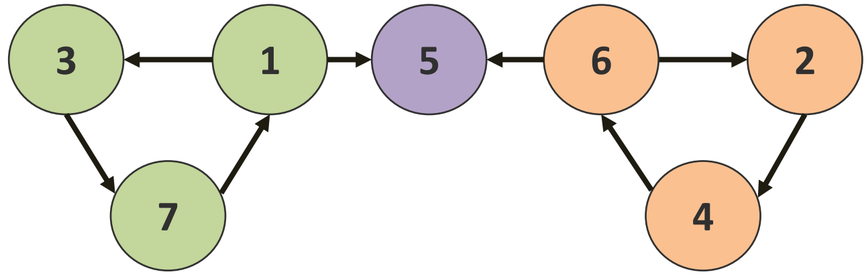
Stack

다시 스택의 제일 뒤 원소를 빼준다. 이는 3번 정점이다.

그러나 3번 정점은 이미 방문 된 상태이다.

고로 아무 것도 하지 않는다. 스택이 비었으므로 모든 강한 연결 요소를 찾는 작업이 끝났다.

코사라주 알고리즘



결론적으로 제시된 그래프에선 강한 연결 요소가 총 3개 있다.

1번째 : {5}

2번째 : {2, 4, 6}

3번째 : {1, 3, 7}

왜 dfs 2번으로 scc를 찾을 수 있을까?

정리 1 : SCC 그래프의 위상 정렬

- SCC들 끼리는 DAG구조를 형성한다. 즉, SCC 전체 집합을 하나의 정점으로 간주하면 전체 그래프가 DAG가 된다.
- 위상 정렬의 성질에 의해 DAG에서는 가장 나중에 방문된 정점이 반드시 SCC의 가장 최상위 SCC중 하나에 속하게 된다.
- DAG에서 위상 정렬을 수행하면 항상 하위 노드부터 방문을 마친 후에 상위 노드를 방문하게 된다. 즉, 나중에 방문이 끝난 정점은 DAG에서 위쪽에 있는 노드일 가능성이 높다. SCC 역시 DAG를 형성 하므로 가장 나중에 탐색이 끝난 정점이 상위 SCC에 속할 가능성이 크다.

-> 따라서 dfs에서 탐색 종료 순서대로 SCC를 찾으면 DAG의 위상 순서를 보장할 수 있다.
(나중에 탐색 완료된 정점부터 SCC를 찾으면 항상 올바른 순서로 SCC를 분리할 수 있다.

정리 2 : 정방향 그래프 G에서 dfs 수행 후 역방향 그래프 T에서 SCC를 찾는 과정의 타당성

- DAG의 성질상 나중에 방문된 정점이 SCC 계층에서 가장 높은 위치에 있다. 즉 이 정점이 속한 SCC는 이후의 SCC에서 절대 도달할 수 없다.
- 그래프 G에서 dfs를 수행할 때, 탐색이 종료된 순서대로 스택에 저장하는 것을 보았을 것이다. SCC 간의 위상 순서에 의해 최상위 SCC의 정점이 가장 늦게 스택에 들어가게 되는데, 스택에서 정점을 꺼내어 처리하면 항상 DAG의 최상위 SCC부터 처리하게 된다.
- 역방향 그래프 T에서는 모든 간선이 반대로 설정되어 있으므로 SCC 내부의 연결은 유지된다. 따라서 T 그래프내에서 dfs를 수행했을때 SCC 내부에서만 탐색이 진행될 수 밖에 없다. 고로 한번의 dfs가 끝나는 순간 해당 SCC 내부의 모든 정점을 방문했음을 보장한다.

-> 따라서 역방향 그래프 T에서 dfs를 수행할 때 SCC가 올바르게 분리됨이 보장된다.