

<알고리즘 실습> - 방향그래프

※ 입출력에 대한 안내

- 특별한 언급이 없으면 문제의 조건에 맞지 않는 입력은 입력되지 않는다고 가정하라.
- 특별한 언급이 없으면, 각 줄의 맨 앞과 맨 뒤에는 공백을 출력하지 않는다.
- 출력 예시에서 □는 각 줄의 맨 앞과 맨 뒤에 출력되는 공백을 의미한다.
- 입출력 예시에서 \mapsto 이 후는 각 입력과 출력에 대한 설명이다.

[문제 1] 위상순서 찾기

주어진 방향그래프 G에 대해 다음과 같이 수행하는 프로그램을 작성하라.

- 1) G가 **방향 비사이클 그래프**(directed acyclic graph: DAG)면 **위상순서**(topological order)를 구해 인쇄.
- 2) G에 **방향 사이클**(directed cycle)이 존재하면 위상순서를 구할 수 없으므로 **0**을 인쇄.

힌트:

1. 그래프를 **인접리스트 구조**로 표현하는 것이 시간 성능 면에서 유리하며 배열로 구현하는 편이 코딩에 용이하다. 아래의 "**알고리즘 설계 팁**" 역시 이 기준으로 제공된다.
2. **위상 정렬** 알고리즘에는 두 가지 버전이 있다. 첫째 **깊이우선탐색(DFS)**을 응용하는 버전, 둘째 **각 정점의 진입차수(in-degree)**를 이용하는 버전이다. 이 가운데 둘째 버전은 G가 DAG면 위상순서를 구하고 그렇지 않으면(즉, 방향 사이클이 존재하면) 일부 정점에 대해 순위를 매기지 않은 채로 정지하므로 DAG가 아님을 알 수 있다. 본 문제 해결을 위해 이 버전을 사용할 것. 상세 내용은 아래의 "**알고리즘 설계 팁**"을 참고할 것.

주의:

1. **방향 사이클의 존재 여부** 검사와 **위상순서 구하기**를 별도 작업으로 수행하면 전체 실행 시간이 늘어나므로, **위상 순서를 구하는 과정에서 방향 사이클의 존재 여부를 확인할 수 있도록** 작성해야 한다.
2. 원래 어떤 그래프에 대한 위상순서는 **여러 개** 있을 수 있다. 하지만 채점 편의 상, 이 문제는 그 가운데 **단 한 개의 위상순서만** 출력하도록 다음 사항을 **준수**해야 한다. (즉, '권고'가 아닌 '요구' 사항으로 받아들여야 제대로 평가받을 수 있고, 아래의 "**알고리즘 설계 팁**"도 이 준수 사항에 따라 작성되어 있다.)
 - 1) 그래프의 **부착리스트** 구축 시, 새로 입력되는 간선에 대한 **노드를 리스트의 맨 앞에 삽입**해야 한다. (이전 실습에서는 정점의 알파벳 순으로 부착 리스트 유지)
 - 2) 위상 정렬 알고리즘에서 **최초로 진입 간선의 개수가 0인 정점을 찾을 때, 알파벳 순서대로 조사**해야 한다.

입출력 형식:

1) main 함수는 아래 형식의 표준입력으로 방향그래프를 입력 받는다.

입력 : 첫 번째 라인 : 정점 수(n)

두 번째 라인 : 정점들의 이름.

각 정점의 이름은 하나의 문자(예: 영문자, 숫자, 기호 등)로 표현

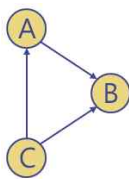
세 번째 라인 : 방향간선 수(m)

이후 m개의 라인 : 방향간선 정보

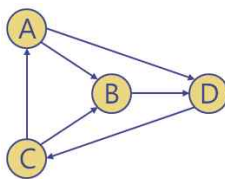
2) main 함수는 다음을 표준출력한다.

출력 : 위상 순서(정점들의 이름을 인쇄)

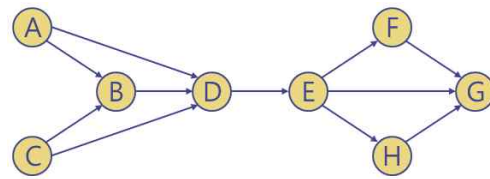
입력 예시 1



입력 예시 2



입력 예시 3



입력 예시 1

3	↳ n = 정점 수	C A B	↳ 위상 순서
A B C	↳ 정점들		
3	↳ m = 간선 수		
A B	↳ 간선 정보		
C A	↳ 간선 정보		
C B	↳ 간선 정보		

출력 예시 1

입력 예시 2

4	↳ n = 정점 수	0	↳ 방향사이클 존재 하므로 위상 순서 없음
A B C D	↳ 정점들		
6	↳ m = 간선 수		
A B	↳ 간선 정보		
C A	↳ 간선 정보		
C B	↳ 간선 정보		
A D	↳ 간선 정보		
B D	↳ 간선 정보		
D C	↳ 간선 정보		

출력 예시 2

입력 예시 3

8	↳ n = 정점 수	A C B D E H F G	↳ 위상 순서
A B C D E F G H	↳ 정점들		
11	↳ m = 간선 수		
A B	↳ 간선 정보		
C B	↳ 간선 정보		
A D	↳ 간선 정보		
C D	↳ 간선 정보		
B D	↳ 간선 정보		
D E	↳ 간선 정보		
E F	↳ 간선 정보		
E H	↳ 간선 정보		
E G	↳ 간선 정보		
F G	↳ 간선 정보		
H G	↳ 간선 정보		

출력 예시 3

필요 데이터구조:

- **G** 방향그래프
 - 크기: $n < 100, m < 1,000$
 - 내용: 방향그래프
 - 범위: 전역
- **n, m** 변수
 - 내용: n = 정점 수, m = 간선 수
 - 데이터 형: 정수
 - 범위: 전역
- **in** 배열
 - 크기: n
 - 데이터 형: 정수
 - 내용: $in[i]$ = 정점 i 의 진입차수
 - 범위: topologicalSort 함수
- **topOrder** 배열
 - 크기: $n + 1$
 - 데이터 형: 정수
 - 내용: $topOrder[0] = 1$ (G가 DAG인 경우), 0 (G가 non-DAG인 경우)
 $topOrder[1:n] =$ 정점들의 위상순서 (G가 DAG인 경우 유효)
 - 범위: 전역
- **Q** 큐
 - 크기: 최대 n
 - 데이터 형: 정수
 - 내용: 정점들의 대기열
 - 범위: 전역

필요 함수:

- **main()** 함수
 - 인자: 없음
 - 반환값: 없음
 - 내용: 방향그래프 정보로부터 그래프를 구축한 후 위상순서를 구하거나, 방향 싸이클 존재를 보고한 후 종료.
- **buildGraph()** 함수
 - 인자: 방향그래프 G (전역)
 - 반환값: 방향그래프 G (전역)
 - 내용: 표준 입력으로부터 방향그래프 정보를 읽어 들여 그래프 G에 저장
- **insertVertex(vName, i)** 함수
 - 인자: 정점 이름 vName, 인덱스 i, 방향그래프 G (전역)
 - 반환값: 방향그래프 G (전역)

- 내용: vName 정점 i를 G의 정점리스트에 삽입하고 i의 진입차수를 초기화
- 시간 성능: $O(1)$
- **insertDirectedEdge(uName, wName, i)** 함수
 - 인자: 정점 이름 uName, wName, 인덱스 i, 방향그래프 G (전역)
 - 반환값: 방향그래프 G (전역)
 - 내용: uName 정점 u를 시점으로, wName 정점 w를 종점으로 하는 방향간선 i를, G의 간선리스트, u의 진출간선리스트, 그리고 w의 진입간선리스트에 각각 삽입하고 w의 진입차수를 갱신
 - 시간 성능: $O(n)$
- **index(vName)** 함수
 - 인자: 식별자 vName, 방향그래프 G (전역)
 - 반환값: 정수
 - 내용 : vName에 해당하는 정점의 인덱스를 찾아 반환
 - 시간 성능: $O(n)$
- **insertFirst(H, i)** 함수
 - 인자: 헤더연결리스트 H, 인덱스 i
 - 반환값: 없음
 - 내용 : H의 첫 노드 위치에 정수 i를 원소로 하는 노드를 삽입
 - 시간 성능: $O(1)$
- **topologicalSort()** 함수
 - 인자: 배열 topSort (전역), 방향그래프 G (전역)
 - 반환값: 배열 topSort (전역)
 - 내용: G로부터 위상순서를 구하거나 방향 사이클이 존재함을 보고
 - 시간 성능: $O(n + m)$
- **isEmpty()** 함수
 - 인자: 큐 Q (전역)
 - 반환값: True/False
 - 내용 : Q가 비어 있으면 True, 아니면 False를 반환
 - 시간 성능: $O(1)$
- **enqueue(v)** 함수
 - 인자: 큐 Q (전역), 정점 v
 - 반환값: 없음
 - 내용 : v를 Q에 삽입
 - 시간 성능: $O(1)$
- **dequeue()** 함수
 - 인자: 큐 Q (전역)
 - 반환값: 정점
 - 내용 : Q로부터 정점을 삭제하여 반환
 - 시간 성능: $O(1)$

알고리즘 설계 팁:

<pre> Alg main() input none output none 1. buildGraph() 2. topologicalSort() 3. if (topOrder[0] = 0) write(0) else for i ← 1 to n write(G.vertices[topOrder[i]].name) 4. return </pre>	<pre> {입력으로부터 G 구축} {G를 위상 정렬} {G는 non-DAG, 즉 방향 싸이클 존재} {G는 DAG} </pre>
--	---

<pre> Alg buildGraph() input graph G output graph G 1. initializeGraph() 2. n ← readline() 3. for i ← 0 to n - 1 vName ← readchar() insertVertex(vName, i) 4. m ← readline() 5. for i ← 0 to m - 1 (uName, wName) ← readline() insertDirectedEdge(uName, wName, i) 5. return </pre>	<pre> {그래프 구축 알고리즘} {G : 전역, 방향그래프} {G : 전역, 방향그래프} {빈 그래프 G 초기화} {n : 정점 수} {vName : 정점 이름} {그래프에 정점 삽입} {m : 간선 수} {방향간선 입력} {그래프에 방향간선 삽입} </pre>
---	---

<pre> Alg insertVertex(vName, i) input identifier vName integer I graph G output graph G 1. G.vertices[i].name ← vName 2. G.vertices[i].outEdges ← empty list 3. G.vertices[i].inEdges ← empty list 4. G.vertices[i].inDegree ← 0 5. return </pre>	<pre> {그래프에 정점 삽입} {G : 전역, 방향그래프} {G : 전역, 방향그래프} {정점 i의 이름 저장} {진출부착간선리스트 초기화} {진입부착간선리스트 초기화} {정점 i의 진입차수 초기화} </pre>
---	---

Alg insertDirectedEdge(uName, wName, i) input identifier uName, wName integer i graph G output graph G	{그래프에 방향간선 삽입} {G : 전역, 방향그래프} {G : 전역, 방향그래프}
1. $u \leftarrow \text{index}(uName)$	{u : uName 정점의 배열 인덱스}
2. $w \leftarrow \text{index}(wName)$	{w : wName 정점의 배열 인덱스}
3. $G.edges[i].origin \leftarrow u$	{간선 i의 시점으로 u를 저장}
4. $G.edges[i].destination \leftarrow w$	{간선 i의 종점으로 w를 저장}
5. insertFirst (G.vertices[u].outEdges, i)	{정점 u의 진출부착간선리스트에 i 삽입}
6. insertFirst (G.vertices[w].inEdges, i)	{정점 w의 진입부착간선리스트에 i 삽입}
7. $G.vertices[w].inDegree++$	{정점 w의 진입차수 갱신}
8. return	

Alg index(vName) input identifier vName graph G output integer	{정점 vName의 인덱스 반환} {G : 전역, 방향그래프}
1. for $i \leftarrow 0$ to $n - 1$	
if $G.vertices[i].name = vName$	
return i	

Alg insertFirst(H, i) input linked-list H integer i output none	{헤더연결리스트의 첫 노드로 i를 삽입 - 주의 사항 2-1) } {H를 헤더로 하는 연결리스트}
1. $node \leftarrow \text{getnode}()$	{동적메모리 노드 node 할당}
2. $node.element \leftarrow i$	{node 원소로 i를 저장}
3. $node.next \leftarrow H.next$	{기존 연결리스트를 node 뒤에 연결}
4. $H.next \leftarrow node$	{node를 H의 첫 노드로 설정}
5. return	

Alg topologicalSort()	{위상 정렬 알고리즘}
input array topOrder[0:n]	{topOrder : 전역, 위상순서}
graph G	{G : 전역, 방향그래프}
output array topOrder[0:n]	{topOrder : 전역, 위상순서}
1. Q ← empty queue	{Q 초기화}
2. for i ← 0 to n - 1	{G의 모든 정점에 대해 정점 번호 순으로 반복 - 주의 사항 2-2 }
in[i] ← G.vertices[i].inDegree	{정점 i의 진입차수를 in[i]에 저장}
if (in[i] = 0)	
Q.enqueue(u)	{진입차수 0인 정점들을 Q에 삽입}
3. t ← 1	{t : 위상순위}
4. while (!Q.isEmpty())	{Q가 비지 않은 동안 반복}
u ← Q.dequeue()	{Q 삭제}
topOrder[t] ← u	{위상순위 t 정점 저장}
t ← t + 1	{위상순위 t 증가}
for each e in G.vertices[u].outEdges	{u의 모든 진출간선 e에 대해 반복}
w ← G.edges[e].destination	{w : 간선 e의 종점}
in[w] ← in[w] - 1	{in[u] 감소}
if (in[w] = 0)	
Q.enqueue(w)	{정점 w의 진입차수가 0이면 Q에 삽입}
5. if (t ≤ n)	{아직 위상순위가 매겨지지 않은 정점이 존재하면}
topOrder[0] ← 0	{G는 non-DAG, 즉 방향 싸이클 존재}
else	
topOrder[0] ← 1	{G는 DAG}
6. return	{위상순서 반환}

isEmpty, enqueue, dequeue 등 큐 관련 알고리즘 설계는 데이터구조 교재를 참고할 것.