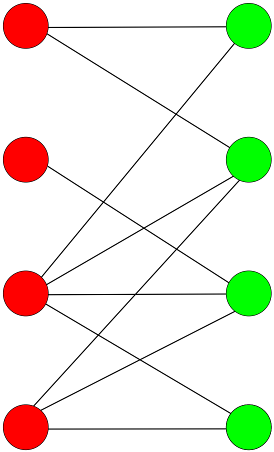
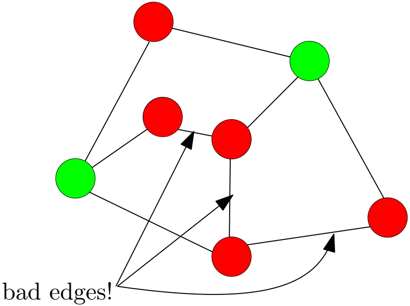
Test Bipartiteness





Testing Bipartiteness using BFS

BFS(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
while $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbors u of v **do**
        if u is "unvisited" **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark u as "visited"

Testing Bipartiteness using BFS

test-bipartiteness(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
$color[s] \leftarrow 0$
while $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbors u of v **do**
        if u is "unvisited" **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark u as "visited"
            $color[u] \leftarrow 1 - color[v]$
        else if $color[u] = color[v]$ **then**
            print("G is not bipartite") and exit

Testing Bipartiteness using BFS

```
mark all vertices as "unvisited"
for each vertex v ∈ V do
    if v is "unvisited" then
        test-bipartiteness(v)
print("G is bipartite")
```

Testing Bipartiteness using BFS

```
mark all vertices as "unvisited"
for each vertex v ∈ V do
    if v is "unvisited" then
        test-bipartiteness(v)
print("G is bipartite")
```

Obs. Running time of algorithm $= O(n + m)$

Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)

1: mark all vertices as "unvisited"
2: recursive-test-DFS(s)

recursive-test-DFS(v)

1: mark v as "visited"
2: **for** all neighbors u of v **do**
3:     **if** u is unvisited **then** , recursive-test-DFS(u)

Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)

mark all vertices as "unvisited"
$color[s] \leftarrow 0$
recursive-test-DFS(s)

recursive-test-DFS(v)

mark v as "visited"
for all neighbors u of v **do**
    if u is unvisited **then**
        $color[u] \leftarrow 1 - color[v]$, recursive-test-DFS($u$)
    else if $color[u] = color[v]$ **then**
        print("G is not bipartite") and exit

Testing Bipartiteness using DFS

mark all vertices as "unvisited"
for each vertex $v \in V$ **do**
    if v is "unvisited" **then**
        test-bipartiteness-DFS(v)
print("G is bipartite")

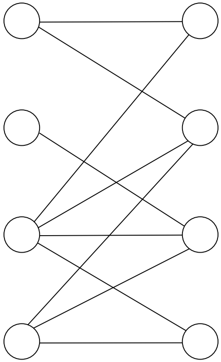
Testing Bipartiteness using DFS

```
mark all vertices as "unvisited"
for each vertex v ∈ V do
    if v is "unvisited" then
        test-bipartiteness-DFS(v)
print("G is bipartite")
```

Obs. Running time of algorithm $= O(n+m)$

Bipartite Graph

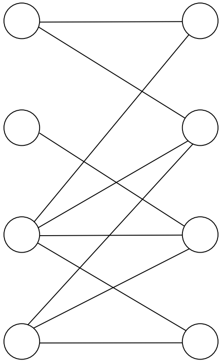
Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.



Bipartite Graph

Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.

Obs. Bipartite graph may contain cycles.

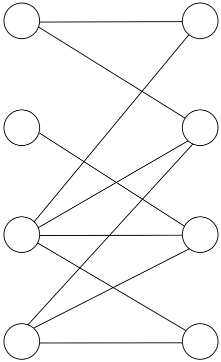


Bipartite Graph

Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.

Obs. Bipartite graph may contain cycles.

Obs. If a graph is a tree, then it is also a bipartite graph.



BFS and DFS

Obs. BFS and DFS naturally induce a tree.

BFS and DFS

Obs. BFS and DFS naturally induce a tree.

Obs. If G is a tree, then BFS tree $=$ DFS tree.

BFS and DFS

Obs. BFS and DFS naturally induce a tree.

Obs. If G is a tree, then BFS tree $=$ DFS tree.

Obs. If BFS tree $=$DFS tree, then G is a tree.

BFS and DFS

Obs. If BFS tree $=$DFS tree, then G is a tree.

- True: simple, undirected graph
- Not True: directed graph

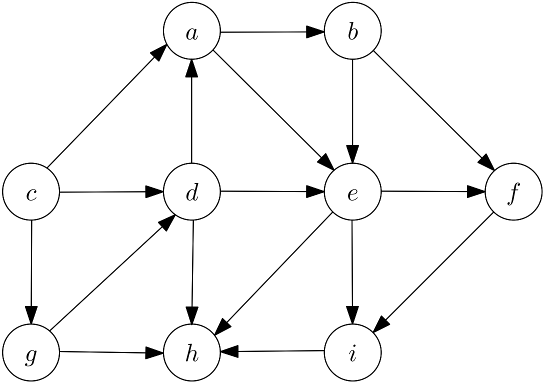
Outline

Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$

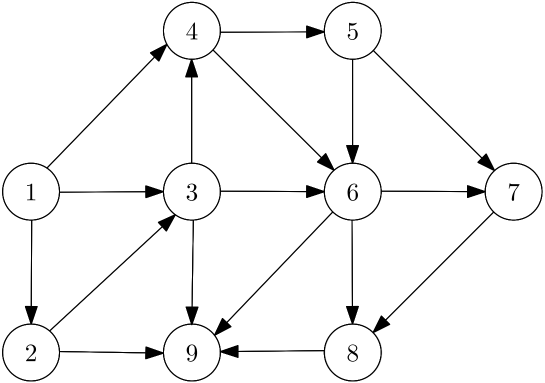


Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

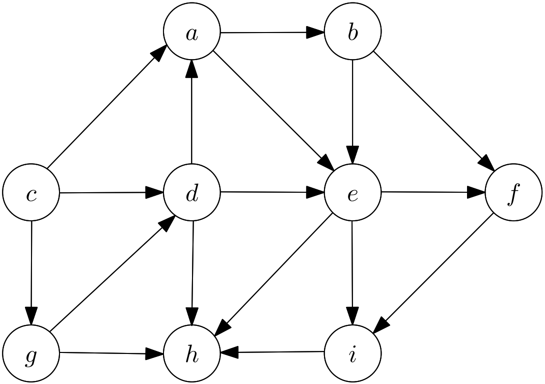
Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$



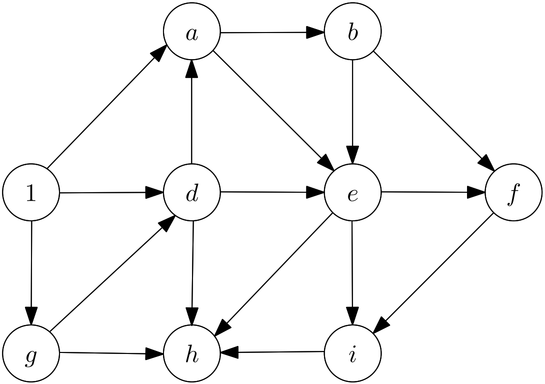
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



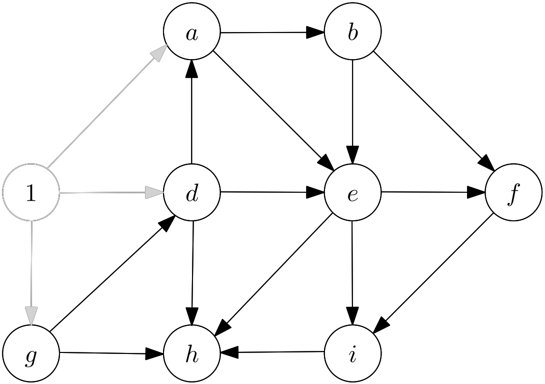
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



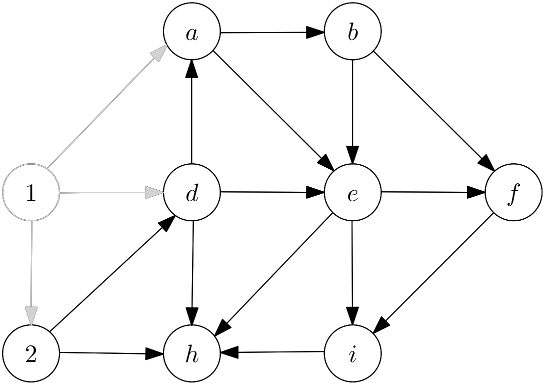
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



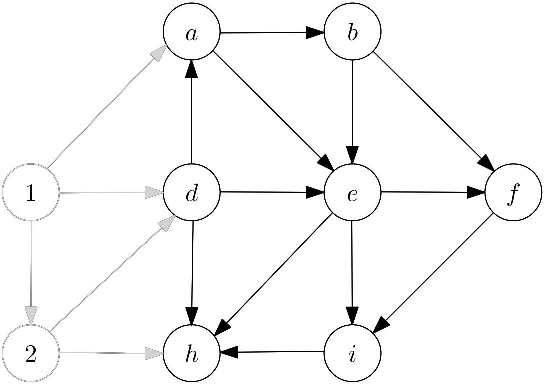
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



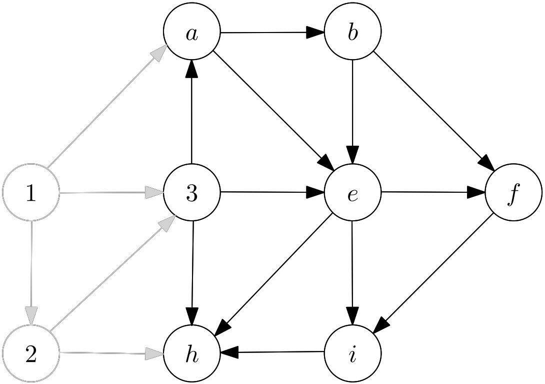
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



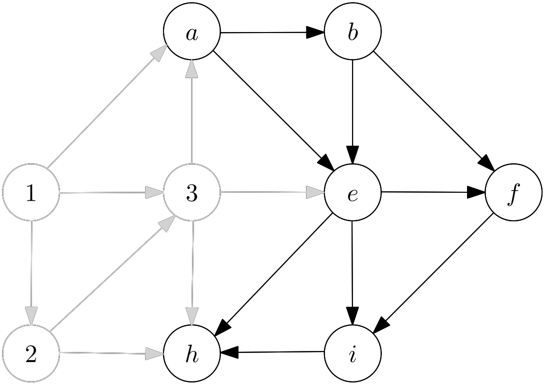
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



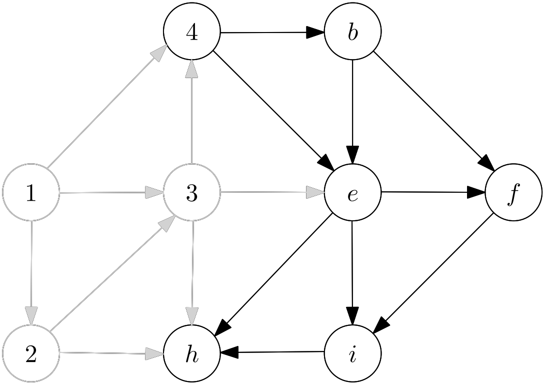
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



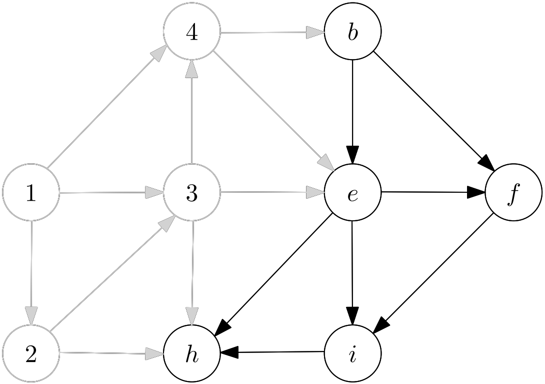
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



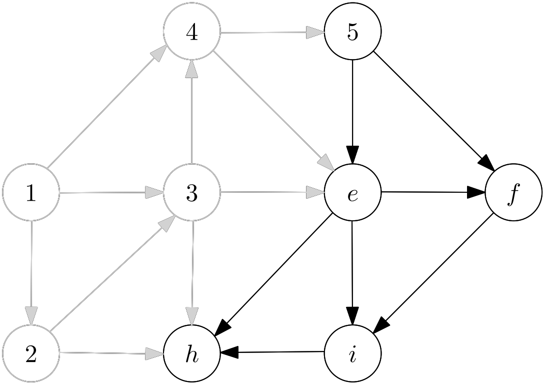
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



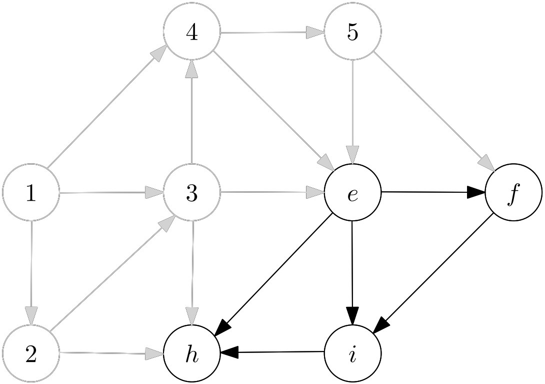
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



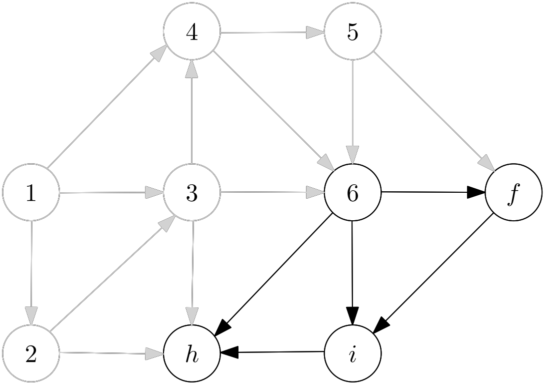
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



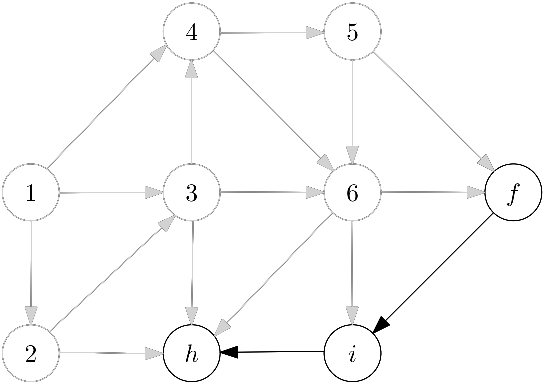
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



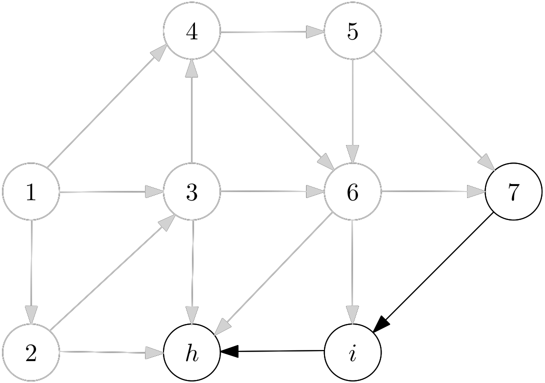
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

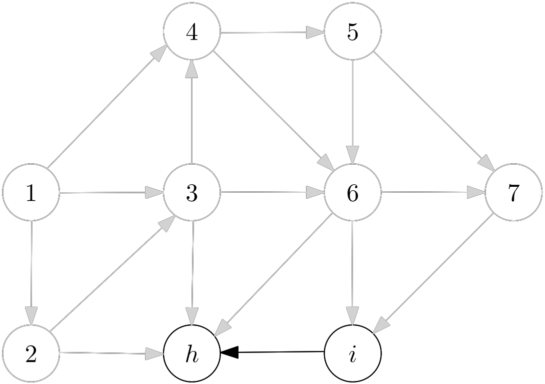


Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

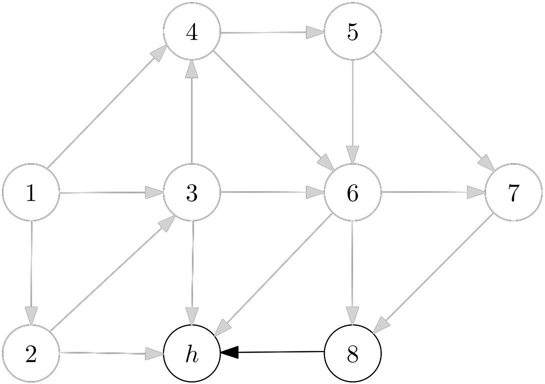


- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



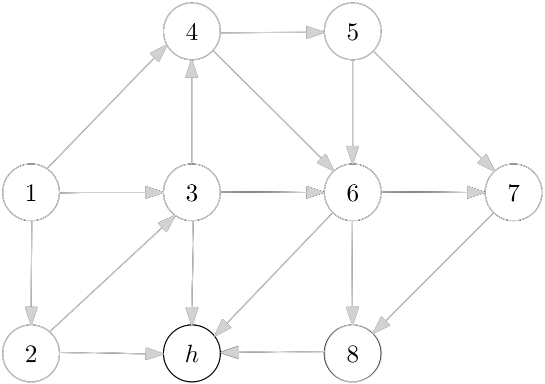
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



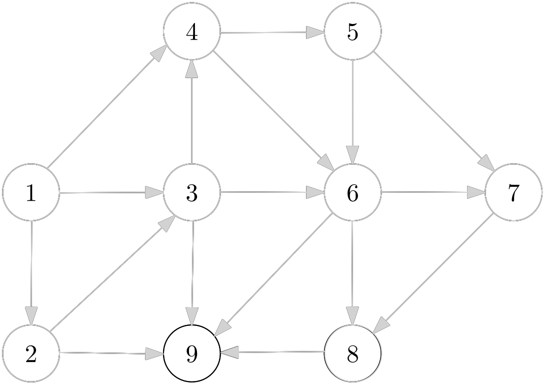
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



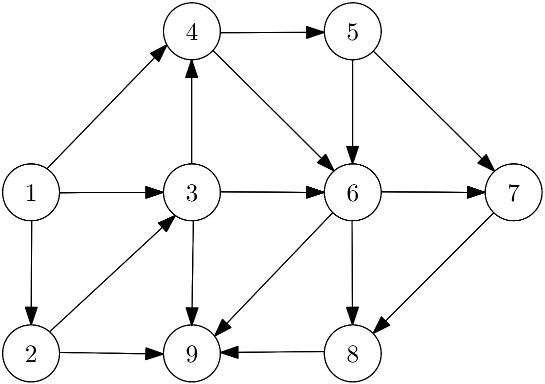
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:

- Use linked-lists of outgoing edges
- Maintain the in-degree d_v of vertices
- Maintain a queue (or stack) of vertices v with $d_v = 0$

topological-sort(G)

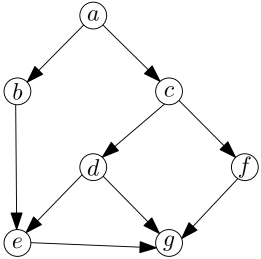
let $d_v \leftarrow 0$ for every $v \in V$
for every $v \in V$ **do**
  for every u such that $(v, u) \in E$ **do**
    $d_u \leftarrow d_u + 1$
$S \leftarrow \{v : d_v = 0\}, i \leftarrow 0$
while $S \neq \emptyset$ **do**
  $v \leftarrow$ arbitrary vertex in S, $S \leftarrow S \setminus \{v\}$
  $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
  for every u such that $(v, u) \in E$ **do**
    $d_u \leftarrow d_u - 1$
    if $d_u = 0$ **then** add u to S
if $i < n$ then output "not a DAG"

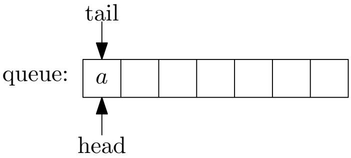
- S can be represented using a queue or a stack
- Running time $= O(n + m)$

S as a Queue or a Stack

DS	Queue	Stack
Initialization	$head \leftarrow 1$, $tail \leftarrow 0$	$top \leftarrow 0$
Non-Empty?	$head \leq tail$	$top > 0$
Add(v)	$tail \leftarrow tail + 1$ $S[tail] \leftarrow v$	$top \leftarrow top + 1$ $S[top] \leftarrow v$
Retrieve v	$v \leftarrow S[head]$ $head \leftarrow head + 1$	$v \leftarrow S[top]$ $top \leftarrow top - 1$

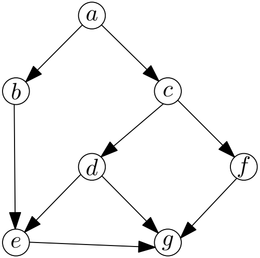
Example

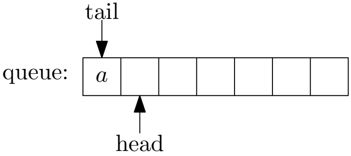




	a	b	c	d	e	f	g
degree	0	1	1	1	2	1	3

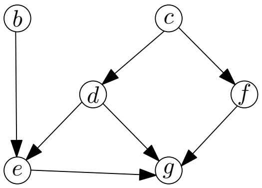
Example

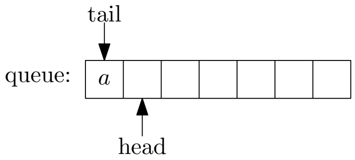




	a	b	c	d	e	f	g
degree	0	1	1	1	2	1	3

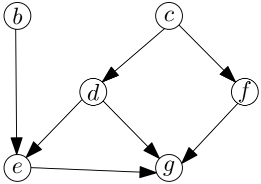
Example

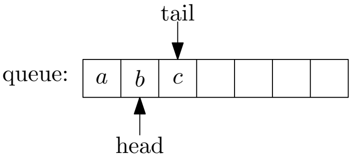




	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

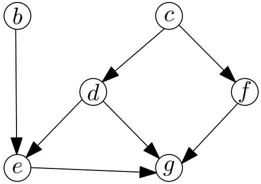
Example

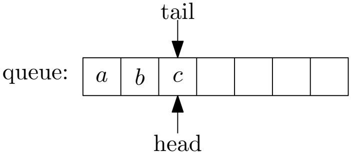




	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

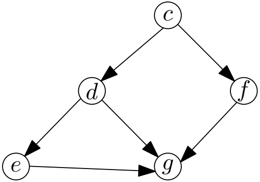
Example

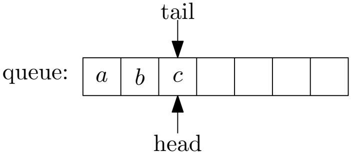




	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

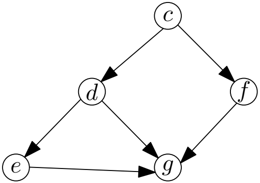
Example

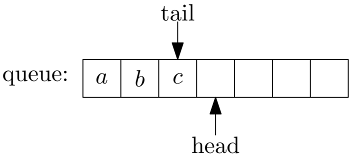




	a	b	c	d	e	f	g
degree	0	0	0	1	1	1	3

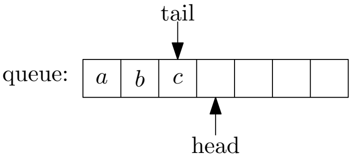
Example

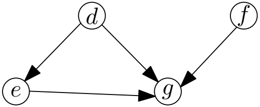




	a	b	c	d	e	f	g
degree	0	0	0	1	1	1	3

Example





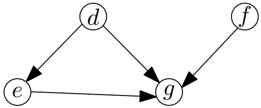
	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

Example

tail

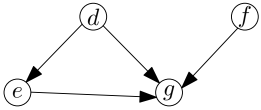
queue:	a	b	c	d	f		

head



	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

Example



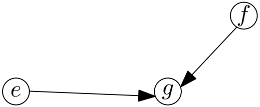
tail ↓

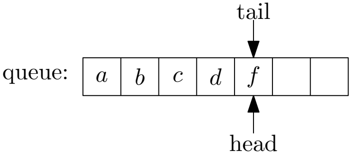
queue:	a	b	c	d	f		

↑ head

	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

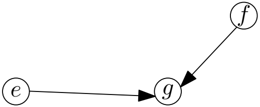
Example





	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	2

Example



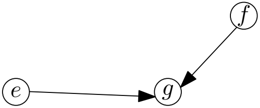
tail

queue:	a	b	c	d	f	e	

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	2

Example



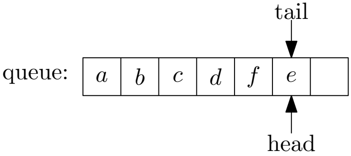
tail

queue:	a	b	c	d	f	e	

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	2

Example



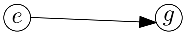
	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	1

Example

tail

queue:	a	b	c	d	f	e	

head



	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	1

Example

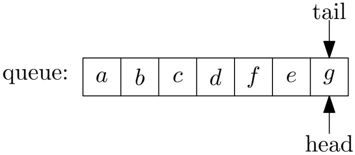
tail

queue:	a	b	c	d	f	e	

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Example



	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Example

tail

queue:	a	b	c	d	f	e	g

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Outline

Def. Word: A string formed by letters.

Def. Adjacency words: Word A and B are adjacent if they differ in exactly one letter.

e.g. wor$d$ and wor$k$; t$e$ll and t$a$ll; ask$b$e and ask$e$e.

Def. Word Ladder: Players start with one word, and in a series of steps, change or transform that word into another word.

Def. Word Ladder: Players start with one word, and in a series of steps, change or transform that word into another word.

- The objective is to make the change in the smallest number of steps, with each step involving changing a **single letter** of the word to create a new valid word.

Word Ladder Problem

Input: Two words S and T, a list of words $A = \{W_1, W_2, ..., W_k\}$.

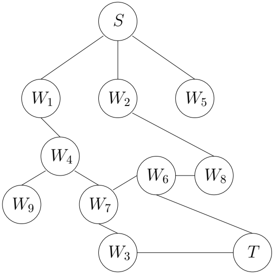
Output: " The smallest word ladder" if we can change S to T by moving between adjacency words in $A \cup \{S, T\}$; Otherwise, "No word ladder".

Example:

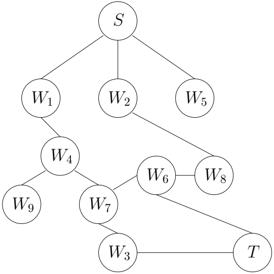
- S="a e f g h", T = "d l m i h"
- W_1="a e f i h", W_2 = "a e m g h", W_3="d l f i h" W_4 = "s e f i h", W_5="a d f g h", W_6 = "d e m i h" W_7="d e f i h", W_8 = "d e m g h", W_9 = "s e m i h"

Example:

- S="a e f g h", T = "d l m i h"
- W_1="a e f i h", W_2 = "a e m g h", W_3="d l f i h"
  W_4 = "s e f i h", W_5="a d f g h", W_6 = "d e m i h"
  W_7="d e f i h", W_8 = "d e m g h", W_9 = "s e m i h"



- Each vertex corresponds to a word.
- Two vertices are adjacent if the corresponding words are adjacent.



- Each vertex corresponds to a word.
- Two vertices are adjacent if the corresponding words are adjacent.
- Hints: Given vertex v, check its nearest neighbor.