Depth-First Search
and
Topological Sort

Dr. G P Gupta

Depth-First Search

- Graph G=(V,E) directed or undirected
- Adjacency list representation
- **Goal:** Systematically ***explore every vertex and every edge***
- **Idea:** ***search deeper whenever possible***
  - Using a Stack (LIFO); (**Note:** FIFO queue used in BFS)

Depth-First Search

- **Maintains several fields for each** $v \in V$
- Like BFS, colors the vertices to indicate their states. **Each vertex is**
  - Initially **white,**
  - grayed when discovered,
  - blackened when finished
- Like BFS, records discovery of a **white** v during scanning Adj[u] by $\pi[v] \leftarrow u$

Depth-First Search

- Unlike BFS, **predecessor graph** G_π **produced by DFS forms spanning forest**
- $G_\pi=(V,E_\pi)$ where

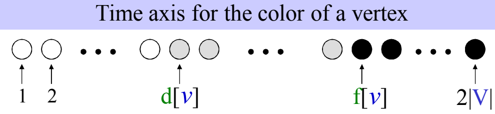
$$E_\pi=\{(\pi[v],v): v \in V \text{ and } \pi[v] \neq \text{NIL}\}$$

- G_π= depth-first forest (DFF) is composed of **disjoint depth-first trees (DFTs)**

Depth-First Search

- DFS also timestamps each vertex with two timestamps
- d[v]: records when v is first discovered and grayed
- f[v]: records when v is finished and blackened
- Since there is only one discovery event and finishing event for each vertex we have $1 \leq d[v] < f[v] \leq 2|V|$

Time axis for the color of a vertex



Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. ***No source vertex given!***
- **Output:**
  - **2 timestamps** on each vertex. Integers between 1 and $2|V|$.
    - $d[v]$ = ***discovery time***
    - $f[v]$ = ***finishing time***

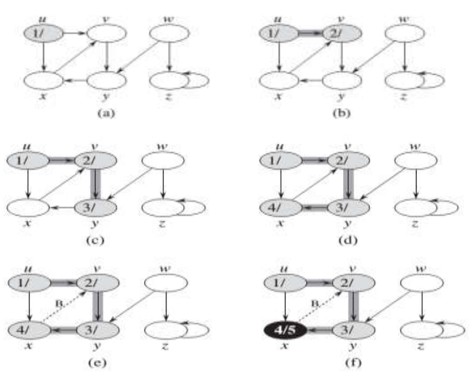
- **Discovery time** - the first time it is encountered during the search.
- **Finishing time** - A vertex is "finished" if it is a leaf node or all vertices adjacent to it have been finished.

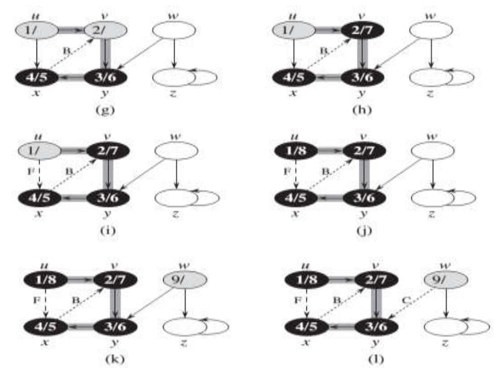
Depth-First Search

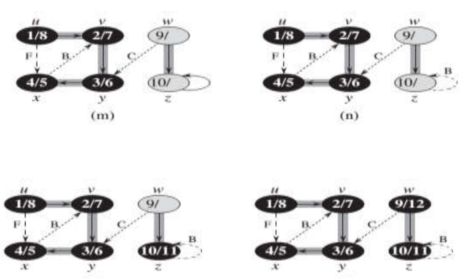
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

Depth-First Search

```
DFS-VISIT(G, u)
time = time + 1                 // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]           // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                 // blacken u; it is finished
time = time + 1
u.f = time
```







Depth-First Search

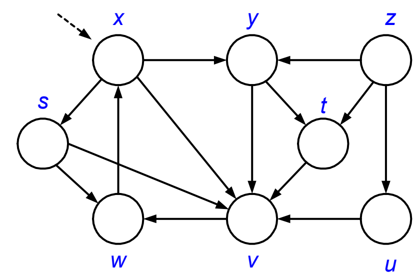
```
DFS(G)
  for each u∈V do
    color[u] ← white
    π[u] ← NIL
  time ← 0
  for each u∈V do
    if color[u] = white then
      DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  color[u] ← gray
  d[u] ← time ← time +1
  for each v∈ Adj[u] do
    if color[v] = white then
      π[v] ← u
      DFS-VISIT(G, v)
  color[u] ← black
  f[u] ← time ← time +1
```

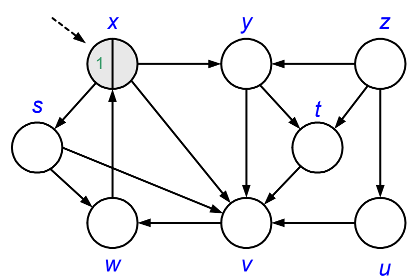
Depth-First Search

- Running time: Θ(V+E)
- Initialization loop in **DFS** : Θ(V)
- Main loop in **DFS**: Θ(V) exclusive of time to execute calls to **DFS-VISIT**
- **DFS-VISIT** is called exactly once for each $v \in V$ since
  - **DFS-VISIT** is invoked only on white vertices and
  - **DFS-VISIT(**G, *u***)** immediately colors u as gray
- For loop of **DFS-VISIT(**G, *u***)** is executed |Adj[*u*]| time
- Since Σ |Adj[*u*]| = E, total cost of executing loop of **DFS-VISIT** is Θ(E)

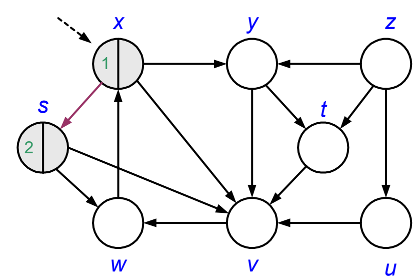
Depth-First Search: Example



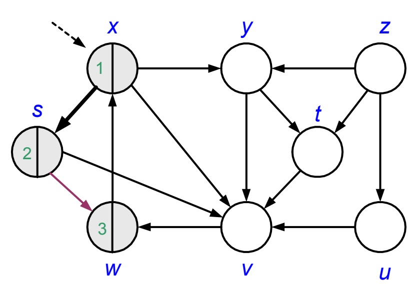
Depth-First Search: Example



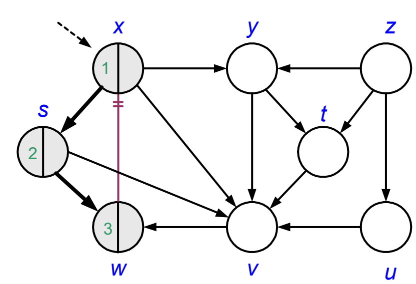
Depth-First Search: Example



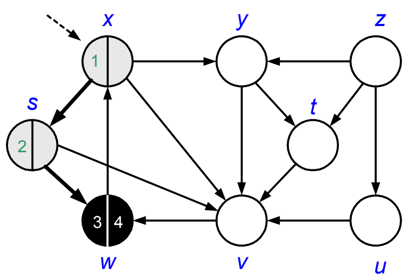
Depth-First Search: Example



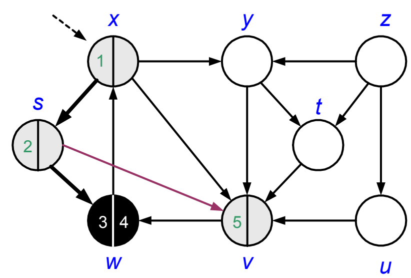
Depth-First Search: Example



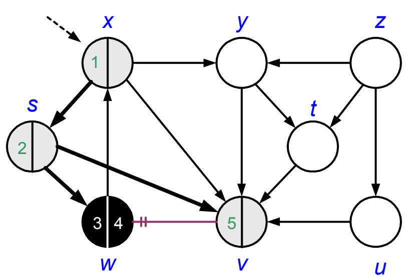
Depth-First Search: Example



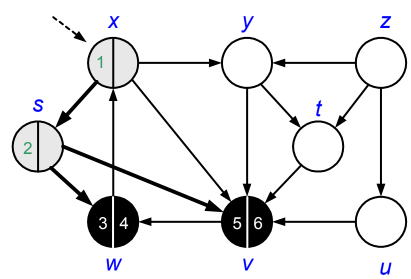
Depth-First Search: Example



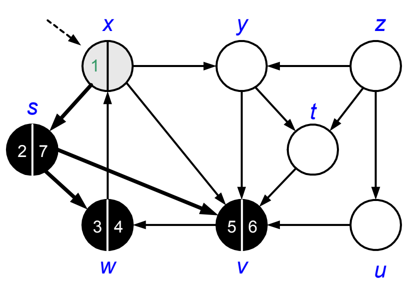
Depth-First Search: Example



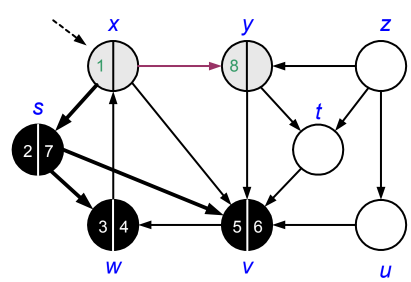
Depth-First Search: Example



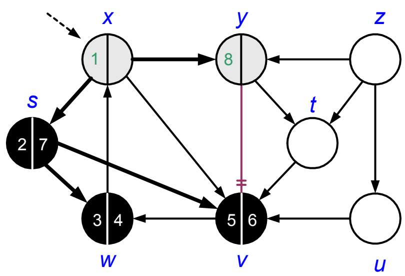
Depth-First Search: Example



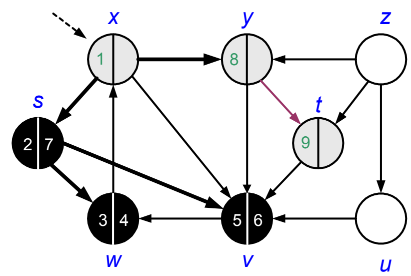
Depth-First Search: Example



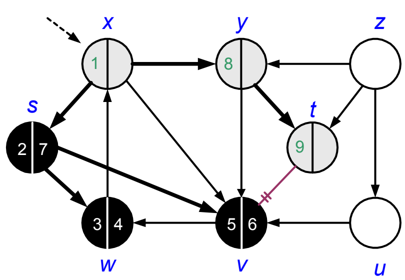
Depth-First Search: Example



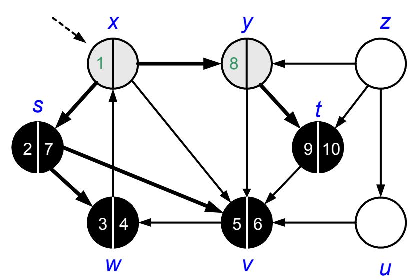
Depth-First Search: Example



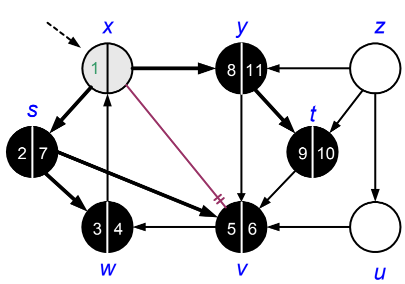
Depth-First Search: Example



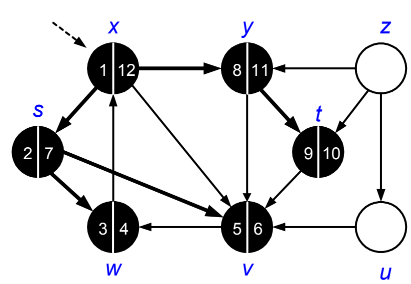
Depth-First Search: Example



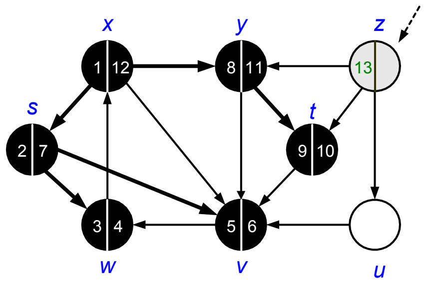
Depth-First Search: Example



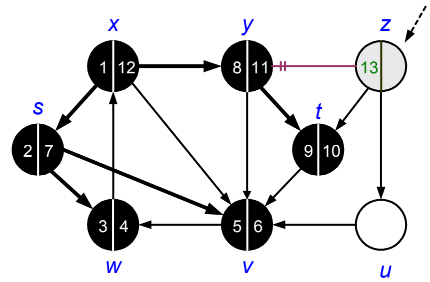
Depth-First Search: Example



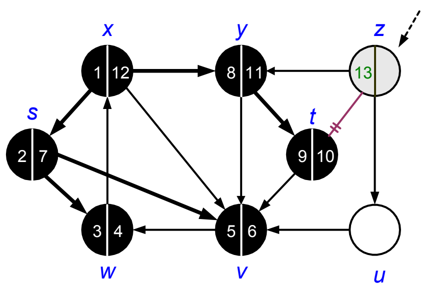
Depth-First Search: Example



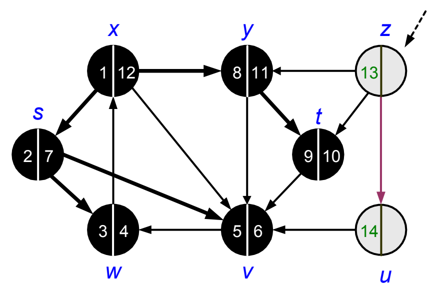
Depth-First Search: Example



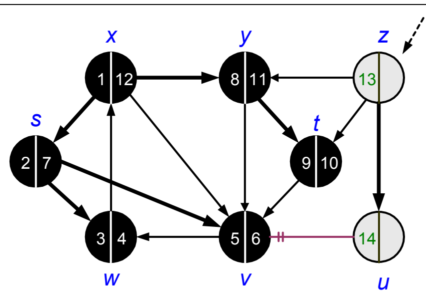
Depth-First Search: Example



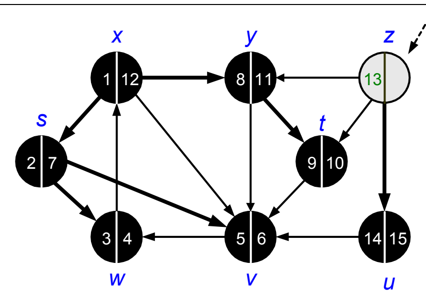
Depth-First Search: Example



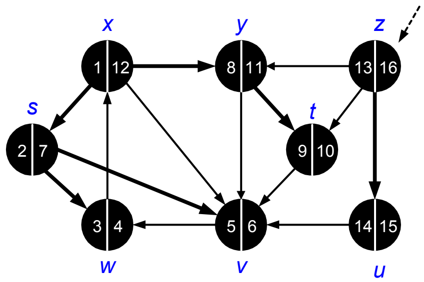
Depth-First Search: Example



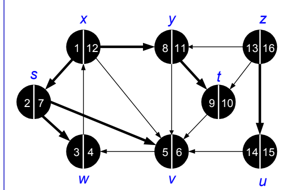
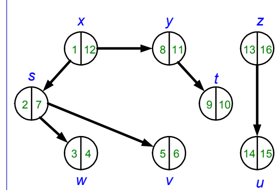
Depth-First Search: Example



Depth-First Search: Example



Depth-First Search: Example

DFS(G) terminated	Depth-first forest (DFF)
	

Topological sort

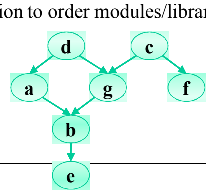
- use depth-first search to perform a topological sort of a directed acyclic graph
- Application
  - for scheduling in project management
  -

Topological sort

- We have a **set of tasks** and a **set of dependencies (precedence constraints)** of form "task A must be done before task B"
- **Topological sort**: An ordering of the tasks that conforms with the given dependencies
- **Goal**: Find a topological sort of the tasks or decide that there is no such ordering

Examples

- **Scheduling**: When **scheduling *task graphs* in distributed systems**,
  - usually we first need to sort the tasks topologically ...and then
  - assign them to resources (the most efficient scheduling is an NP-complete problem)
- Or during compilation to order modules/libraries



Examples

- **Resolving dependencies**:
  - ***apt-get*** uses topological sorting to obtain the admissible sequence in which a set of Debian packages can be installed/removed

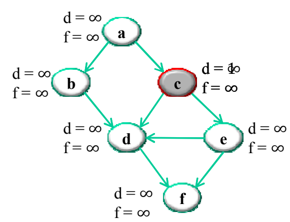
topological sort

- TOPOLOGICAL-SORT(**G**):
  1) call DFS(G) to compute **finishing** times **f**[**v**] for each vertex **v**
  2) as each vertex is finished, insert it onto the **front** of a linked list
  3) return the linked list of vertices

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 2



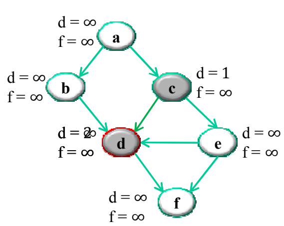
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 3



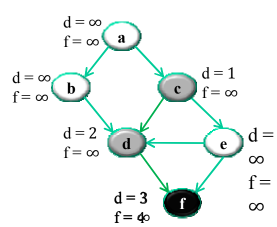
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 4



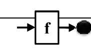
2) as each vertex is finished, insert it onto the **front** of a linked list

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

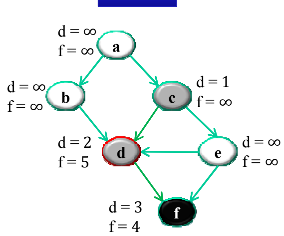
f is done, move back to **d**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 5



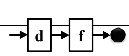
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

f is done, move back to **d**

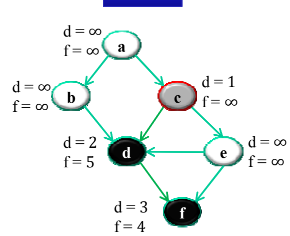
d is done, move back to **c**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 6



Let's say we start the DFS from the vertex **c**

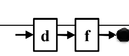
Next we discover the vertex **d**

Next we discover the vertex **f**

f is done, move back to **d**

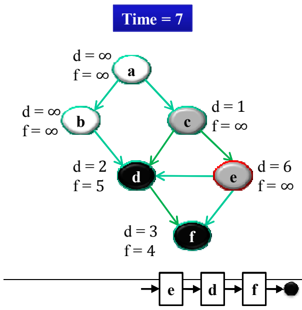
d is done, move back to **c**

Next we discover the vertex **e**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[v]



Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Both edges from **e** are **cross edges**

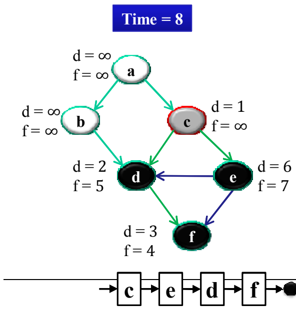
d is done, move back to **c**

Next we discover the vertex **e**

e is done, move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[v]



Let's say we start the DFS from the vertex **c**

Just a note: If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge** (in this particular DFS run)

d is done, move back to **c**

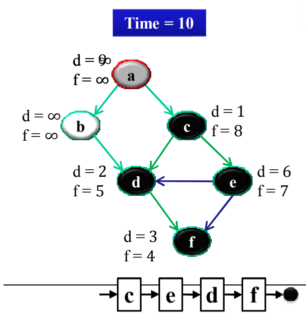
Next we discover the vertex **e**

e is done, move back to **c**

c is done as well

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[v]



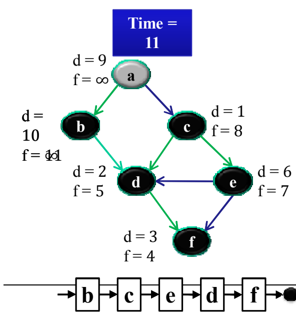
Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[v]



Let's now call DFS visit from the vertex **a**

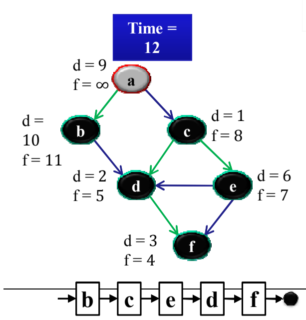
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[v]



Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

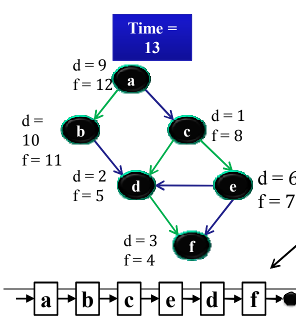
Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[v]



WE HAVE THE RESULT!

3) return the linked list of vertices

Let's now call DFS visit from the vertex **a**

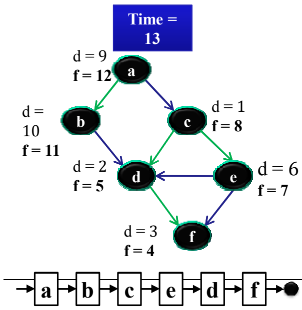
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort



The linked list is sorted in **decreasing** order of finishing times **f[]**

Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point „**from left to right**“

Time complexity of TS(G)

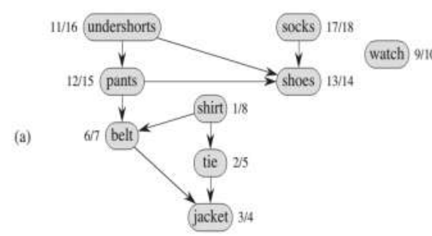
- Running time of topological sort:

  $\mathbf{\Theta(n + m)}$

  where $\mathbf{n}=|\mathbf{V}|$ and $\mathbf{m}=|\mathbf{E}|$
- Why? Depth first search takes $\mathbf{\Theta(n + m)}$ time in the worst case, and inserting into the front of a linked list takes $\mathbf{\Theta(1)}$ time

Example 2

Topological sort



Topological sort