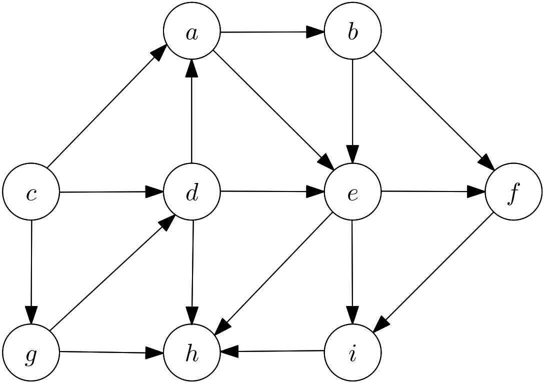
Topological Ordering Problem

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 $\pi(u) < \pi(v)$

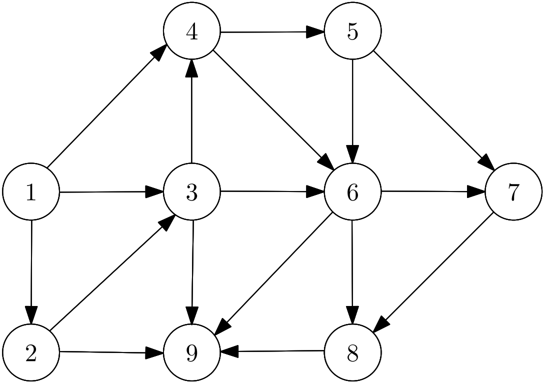


Topological Ordering Problem

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

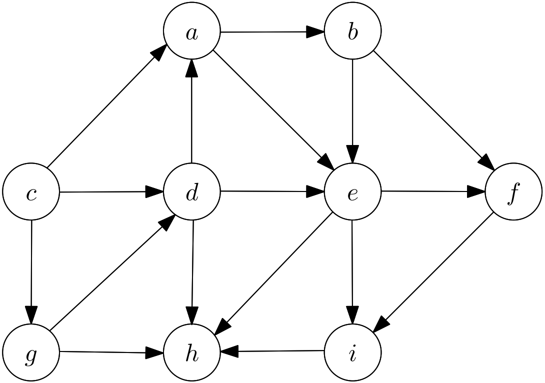
Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

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



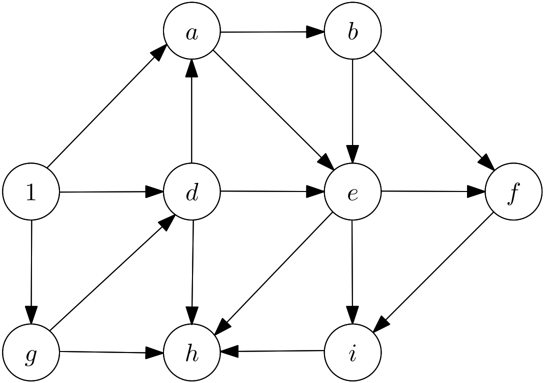
Topological Ordering

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



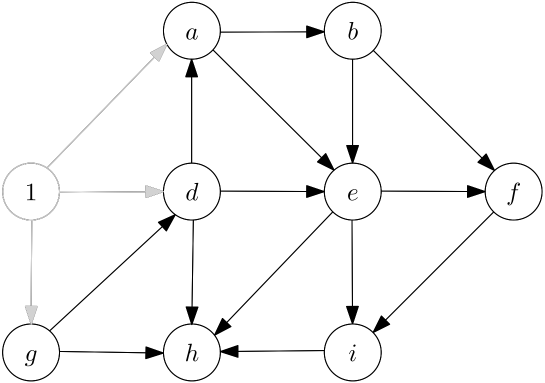
Topological Ordering

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



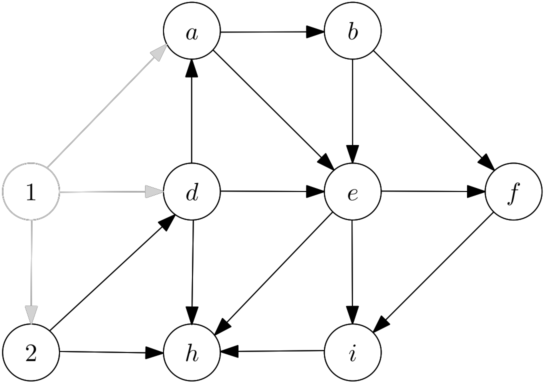
Topological Ordering

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



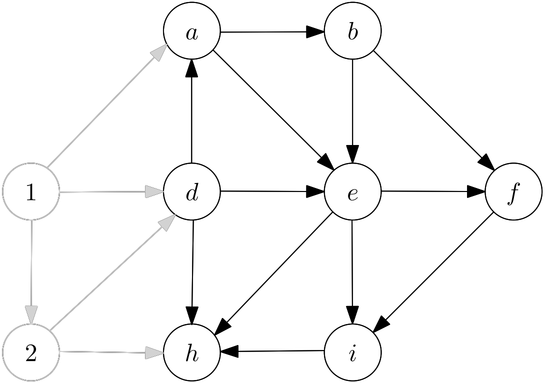
Topological Ordering

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



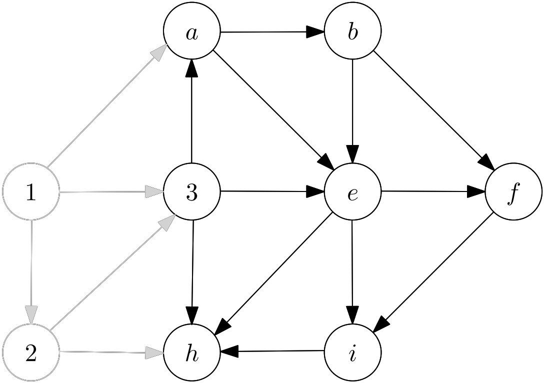
Topological Ordering

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



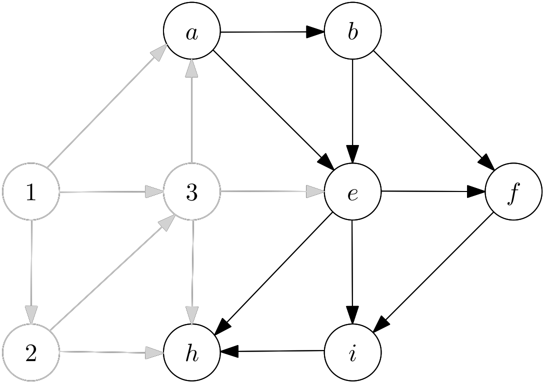
Topological Ordering

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



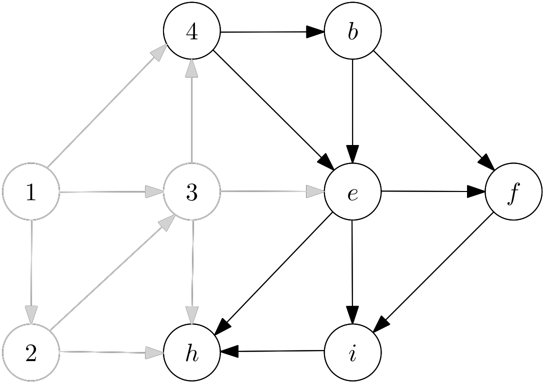
Topological Ordering

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



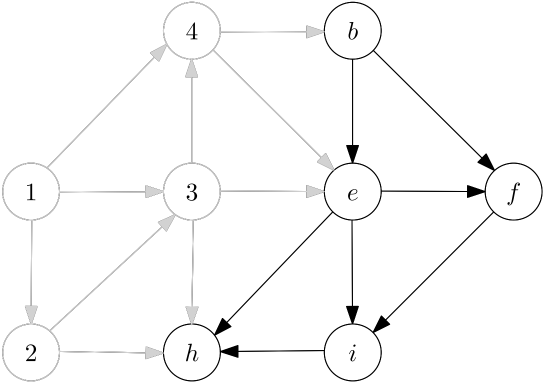
Topological Ordering

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



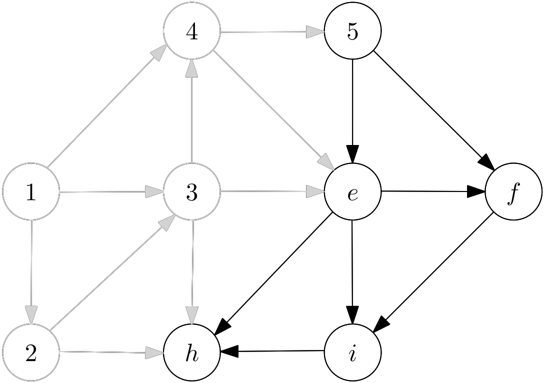
Topological Ordering

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



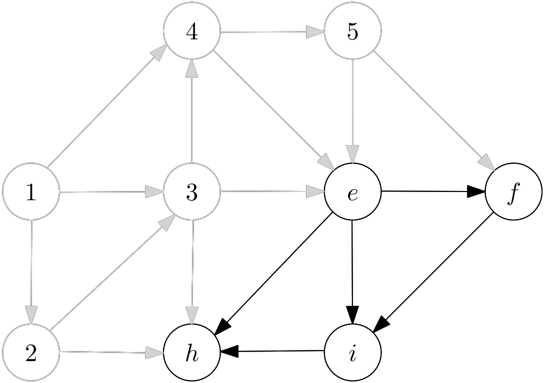
Topological Ordering

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



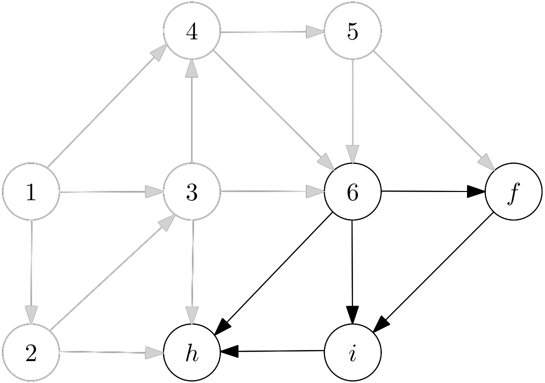
Topological Ordering

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



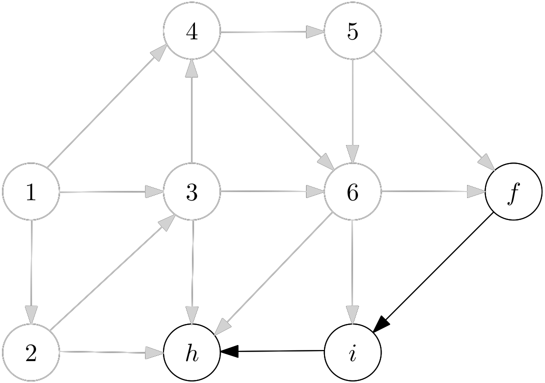
Topological Ordering

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



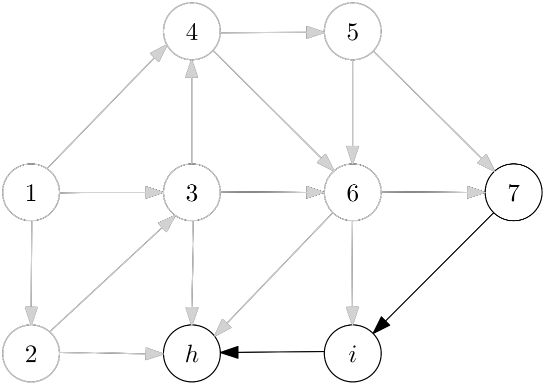
Topological Ordering

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



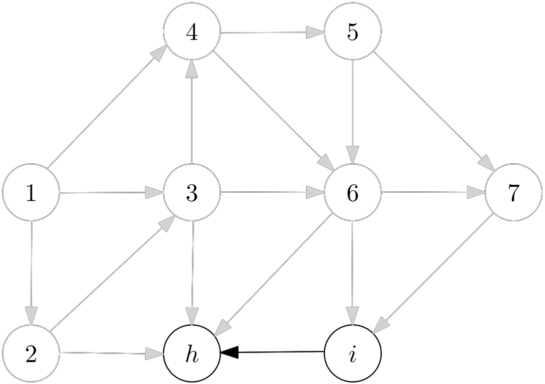
Topological Ordering

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



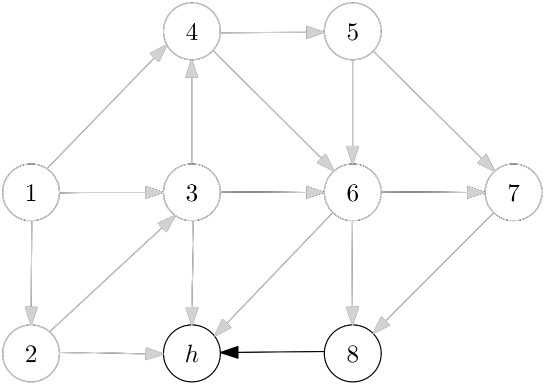
Topological Ordering

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



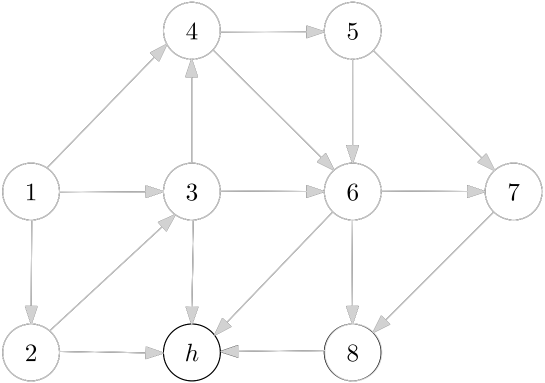
Topological Ordering

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



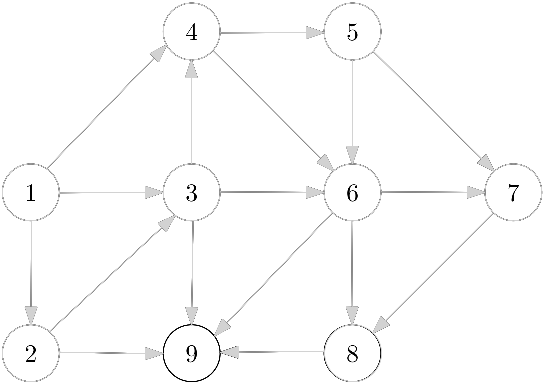
Topological Ordering

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



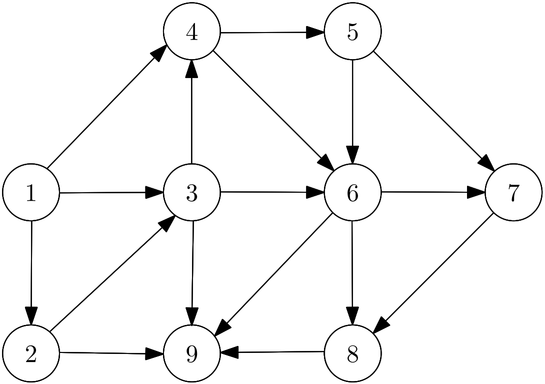
Topological Ordering

- 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.

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)

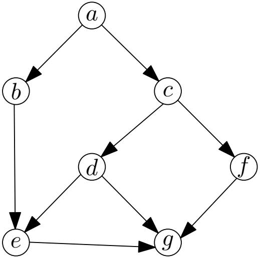
```
let $d_v \leftarrow 0$ for every $v \in V$
for every $v \in V$ do
    for every $u$ such that $(v, u) \in E$ do
        $d_u \leftarrow d_u + 1$
$S \leftarrow \{v : d_v = 0\}, i \leftarrow 0$
while $S \neq \emptyset$ do
    $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$
    $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
    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”
```

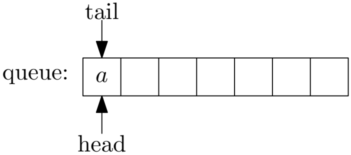
- S can be represented using a queue or a stack
- Running time $= O(n + m)$

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	$v \leftarrow S[head]$ $head \leftarrow head + 1$	$v \leftarrow S[top]$ $top \leftarrow top - 1$

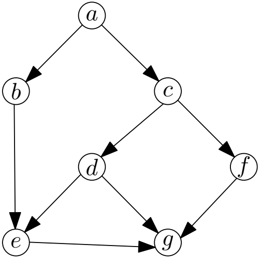
Example

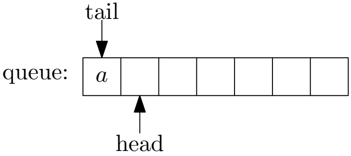




	a	b	c	d	e	f	g
degree	0	1	1	1	2	1	3

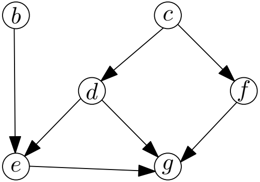
Example

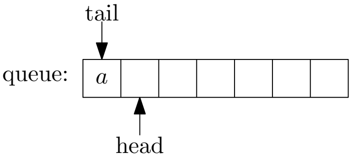




	a	b	c	d	e	f	g
degree	0	1	1	1	2	1	3

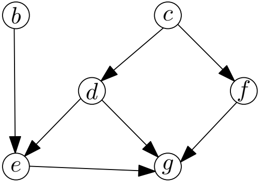
Example

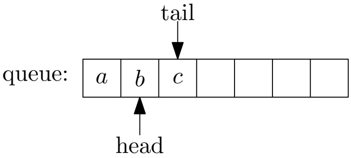




	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

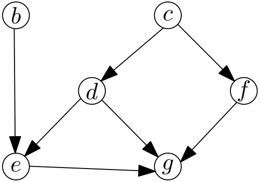
Example

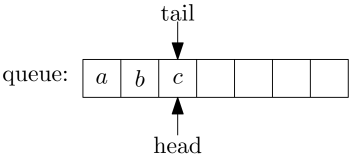




	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

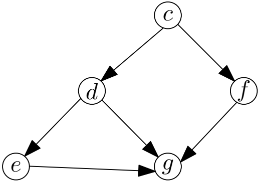
Example

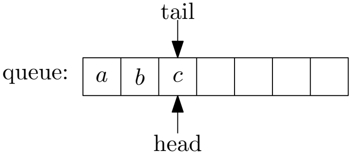




	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

Example





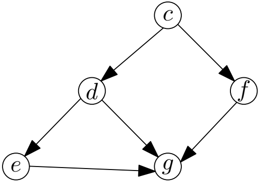
tail

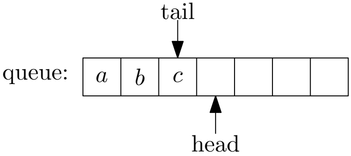
queue:	a	b	c				

head

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

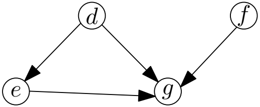
Example

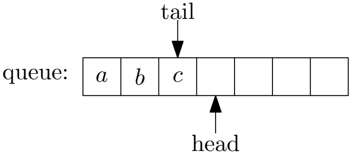




	a	b	c	d	e	f	g
degree	0	0	0	1	1	1	3

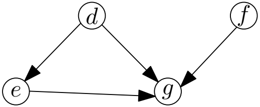
Example

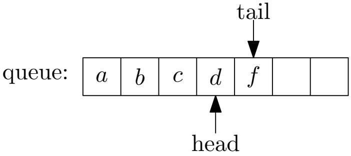




	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

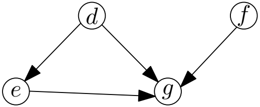
Example

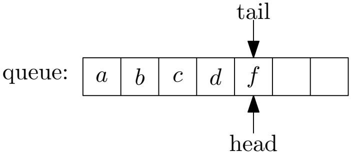




	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

Example





tail

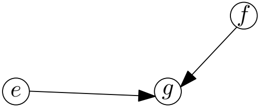
queue:

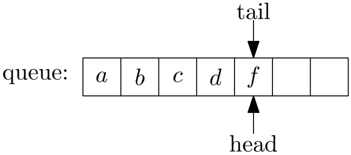
a	b	c	d	f		

head

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

Example





tail

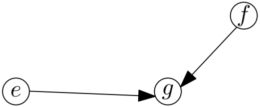
queue:

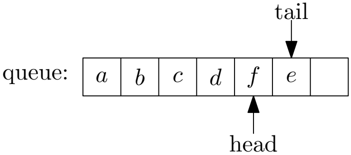
a	b	c	d	f		

head

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

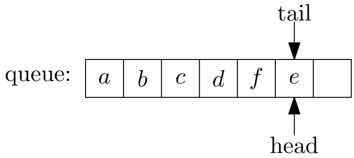
Example





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

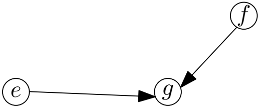
Example



tail

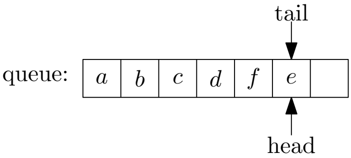
queue:	a	b	c	d	f	e	

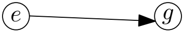
head



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

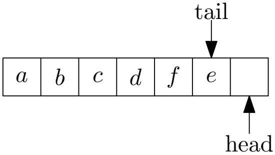
Example





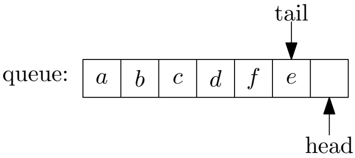
	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	1

Example



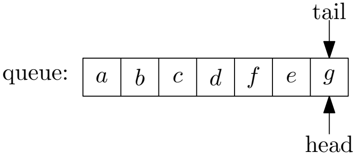
	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	1

Example



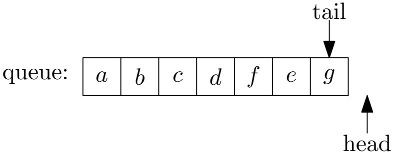
	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Example



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

Example



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

Outline

1. Graphs
2. Connectivity and Graph Traversal
    - Types of Graphs
3. Bipartite Graphs
    - Testing Bipartiteness
4. Topological Ordering
    - Applications: Word Ladder

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 $A = \{W_1, W_2, ..., W_k\}$.

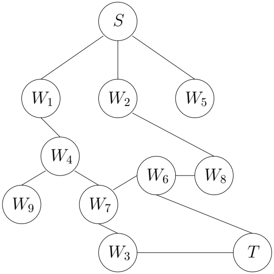
Output: " The smallest word ladder" if we can change S to T by moving between adjacency words in $A \cup \{S, T\}$; Otherwise, "No word ladder".

Example:

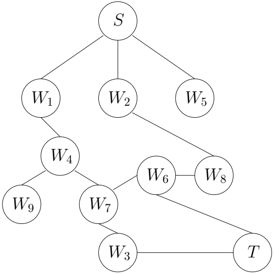
- S="a e f g h", T = "d l m i h"
- W_1="a e f i h", W_2 = "a e m g h", W_3="d l f i h"
  W_4 = "s e f i h", W_5="a d f g h", W_6 = "d e m i h"
  W_7="d e f i h", W_8 = "d e m g h", W_9 = "s e m i h"

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.

CSE 431/531B: Algorithm Analysis and Design (Fall 2023)

Greedy Algorithms

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time = efficient

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time $=$ efficient

Goals of algorithm design

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time $=$ efficient

Goals of algorithm design

1. Design efficient algorithms to solve problems

Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time = efficient

Goals of algorithm design

1. Design efficient algorithms to solve problems
2. Design more efficient algorithms to solve problems

Common Paradigms for Algorithm Design

- Greedy Algorithms: shortest path problem
- Divide and Conquer: merge-sort, binary search
- Dynamic Programming: Fibonacci number

Greedy algorithm properties

Greedy algorithm properties

- Greedy algorithms are often for optimization problems.

Greedy algorithm properties

- Greedy algorithms are often for optimization problems.
- They often run in polynomial time due to their simplicity: easy to come up with, easy to analyze running time.

Greedy algorithm properties

- Greedy algorithms are often for optimization problems.
- They often run in polynomial time due to their simplicity: easy to come up with, easy to analyze running time.
- Hard to see correctness. Mostly, it is not correct. E.g. $\min f(x)$

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is "safe"
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is “safe” (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is "safe" (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Def. A strategy is safe: there is always an optimum solution that agrees with the decision made according to the strategy.

Outline

1. Toy Example: Box Packing
2. Interval Scheduling
3. Offline Caching
    - Heap: Concrete Data Structure for Priority Queue
4. Data Compression and Huffman Code
5. Summary

Box Packing

Input: n boxes of capacities $c_1, c_2, \cdots, c_n$

m items of sizes $s_1, s_2, \cdots, s_m$

Can put at most 1 item in a box

Item j can be put into box i if $s_j \leq c_i$

Output: A way to put as many items as possible in the boxes.

Box Packing

Input: n boxes of capacities $c_1, c_2, \cdots, c_n$

m items of sizes $s_1, s_2, \cdots, s_m$

Can put at most 1 item in a box

Item j can be put into box i if $s_j \leq c_i$

Output: A way to put as many items as possible in the boxes.

Example:

- Box capacities: 60, 40, 25, 15, 12
- Item sizes: 45, 42, 20, 19, 16
- Can put 3 items in boxes: $45 \to 60, 20 \to 40, 19 \to 25$

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy

Designing a Reasonable Strategy for Box Packing

- Q: Take box 1. Which item should we put in box 1?