香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Dynamic Programming (II)

Kodi Siu {bedrockfake}

2026-02-21

Why Dynamic Programming?

- Dynamic Programming is designed to solve problems that ask for the **best** solution from many possibilities.
  - e.g. Finding min cost / max value / best arrangement

"DP is the art of breaking down a complex problem into smaller subproblems."

- Key Characteristics of DP Problems:
  - **Overlapping Subproblems**: The problem can be broken down into smaller pieces that are reused multiple times.
  - **Optimal Substructure**: The optimal solution to the whole problem depends on the optimal solutions to its smaller subproblems.

Why Dynamic Programming? (In OI)

- DP is a very common technique in OI.

- Many problems are designed to be solved (or partially solved) by DP.
  - There are many tricks and optimization of DP, making it a great topic to be tested.
  - Mastering DP unlocks solutions to a wide range of challenges.

How to DP?

1. Divide the main problem into smaller, self-similar subproblems.
2. Identify overlapping subproblems that may be encountered multiple times.
3. Store the solutions to subproblems, usually in an array.
4. When you encounter the same subproblem again, simply look up the stored solution instead of recomputing it.

Table of Contents

Things You SHOULD Know - DP Building Blocks

- **Base Cases**
  - The simplest scenarios where the answer is immediately known.
- **States**
  - Store all essential information to define a subproblem.
- **Transition Formula**
  - How do you solve a subproblem using the solutions to smaller subproblems?
  - Defines the relationship between subproblems.

Things You SHOULD Know - Concepts of DP

- Some concepts that you better understand before getting into today's topic.
- **Optimal Substructure**
  - The optimal solution to the whole problem can be constructed from optimal solutions to its subproblems.
    - In other perspective, the solution of the larger problem is optimal as it depends on an optimal solution of a smaller problem.
  - DP works because of this property.
- **Calculation Order**
  - To use DP effectively, calculate subproblem solutions in an order where you have the needed sub-solutions ready when you need them.
  - Often trivial: solve smaller subproblems first.

Example task for Revision

Problem: N people want to cross a bridge. It costs a[i] time for i people to cross together (i=1 to N). Find minimum total time that all N people cross the bridge.

- State: dp[i] = minimum time for i people to cross.
- Base Case: dp[0] = 0 (0 people, 0 time)
- Transition Formula: dp[i] = min(dp[i], dp[i-j] + a[j]) for all j in 1..i
- Optimal Substructure: Yes
- Calculation Order: dp[0], dp[1], dp[2], ..., dp[N] in increasing order.
- Final Answer: dp[N]

Table of Contents

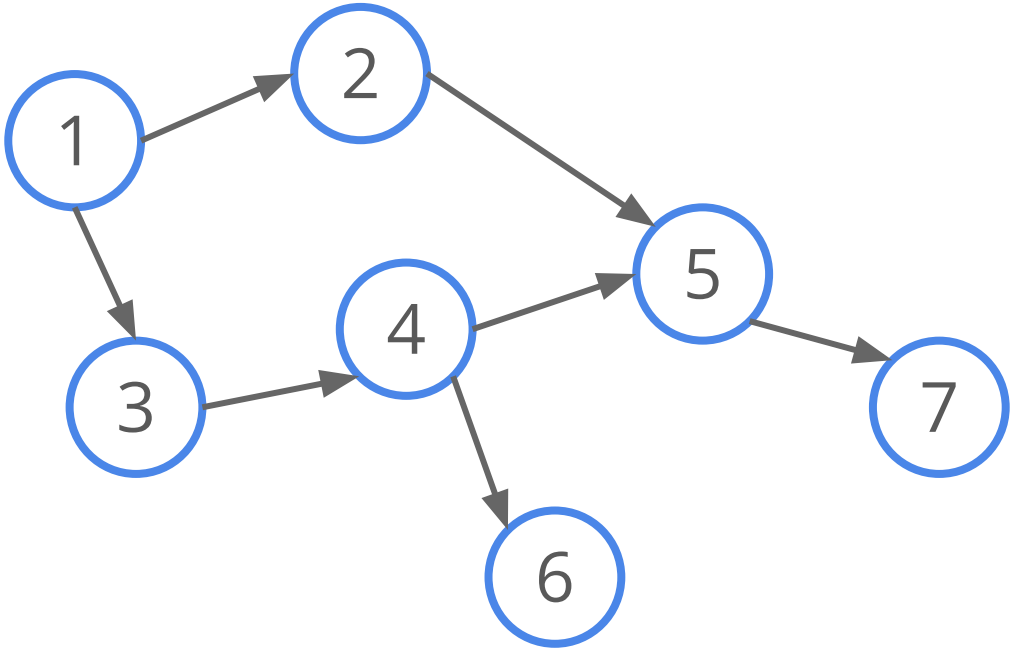
DAG: Visualizing and Solving with Dependencies

- **D**irected **A**cyclic **G**raph

- In DP, subproblems have dependencies: solving one part requires solutions to other parts. Directed Acyclic Graphs are perfect for modeling these dependencies visually.

- **Nodes** = Subproblems (States): Each node in a DAG represents a subproblem.
- **Edges** = Dependencies: A directed edge A -> B means subproblem B depends on subproblem A. (You must solve A first then solve B!)

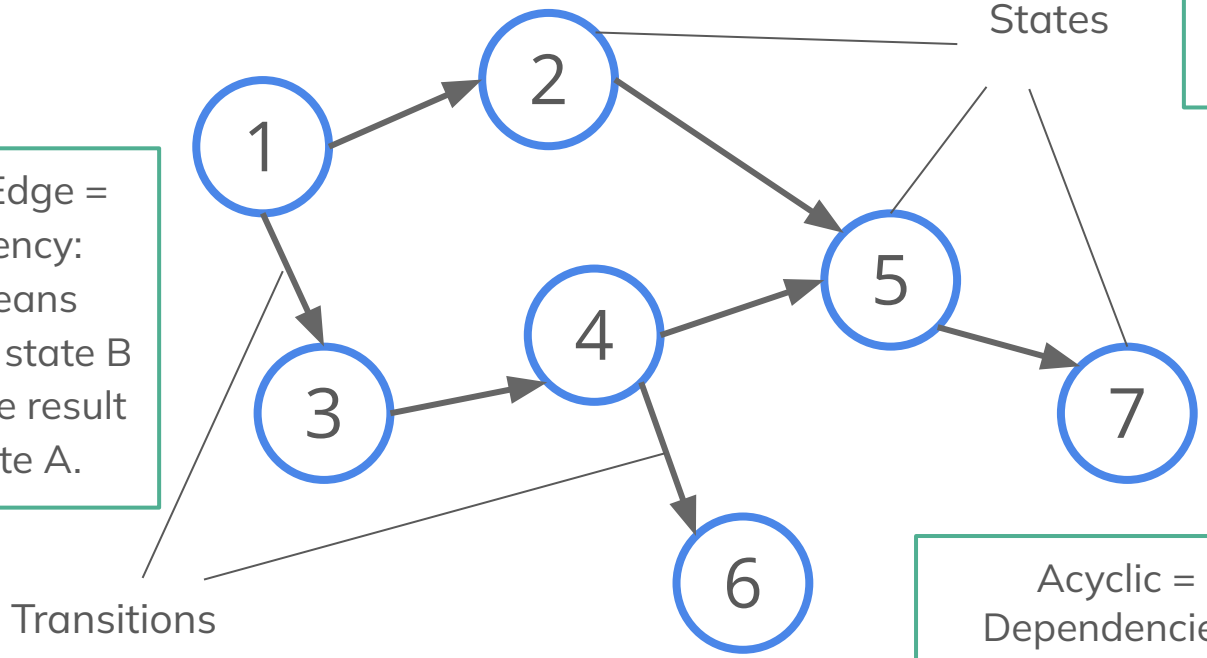
DAG



DAG

Node = State: Each node is a subproblem.

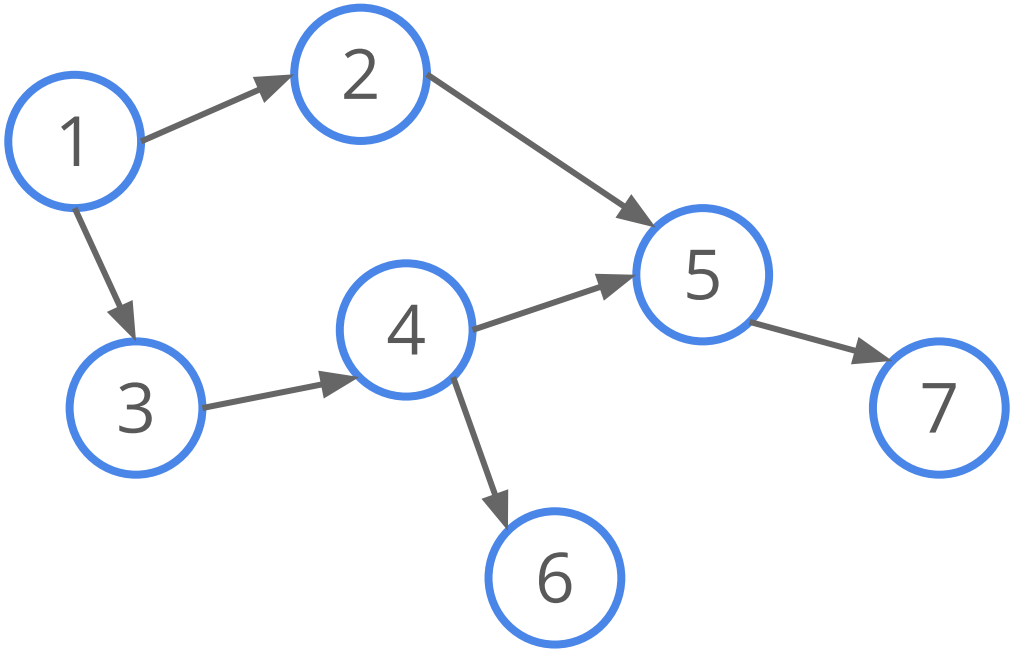
Directed Edge = Dependency: A→B means calculating state B requires the result from state A.



Acyclic = No Circular Dependencies: No loops of dependencies, ensuring a solvable order.

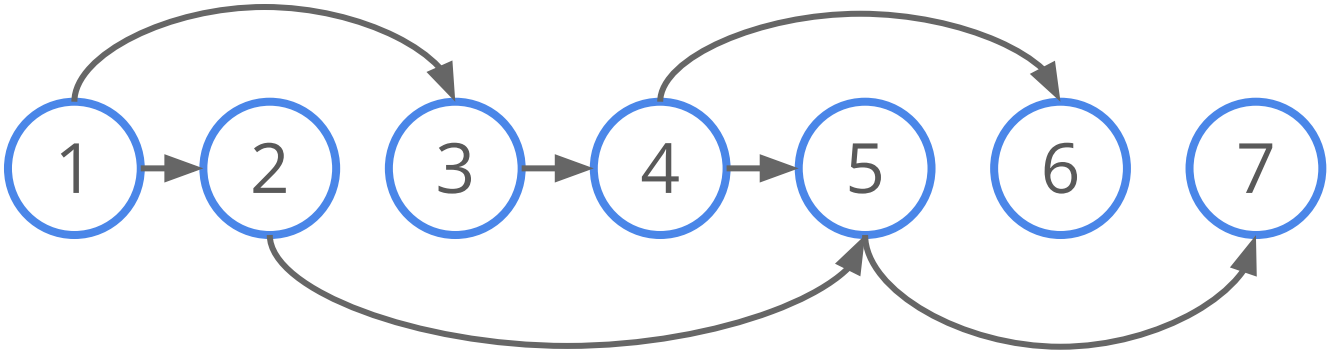
DAG

- **Topological sort** is usually applied on DAG to determine the DP order



DAG

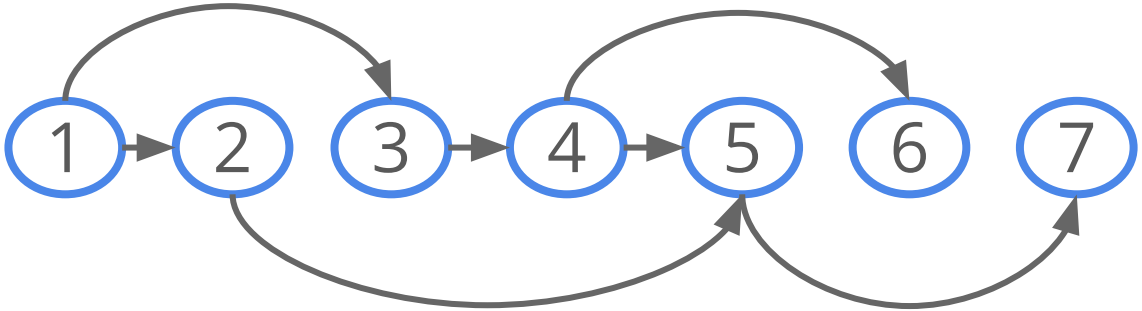
- **Topological sort** is usually applied on DAG to determine the DP order



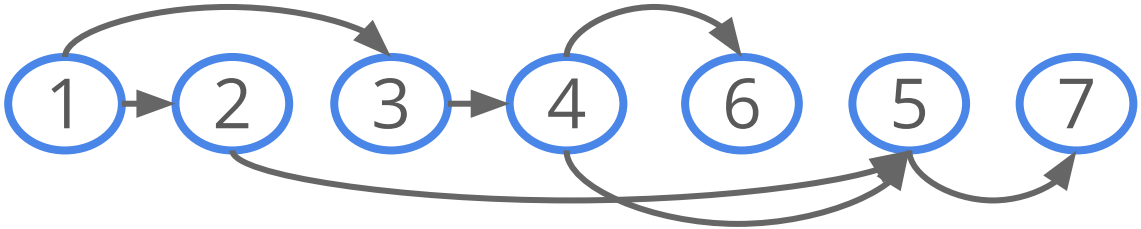
- After sort: a linear ordering of nodes such that for every directed edge from node A to node B, node A comes before node B in the ordering.

DAG

- The results of **Topological sort** is not necessarily unique, unlike normal sorting algorithms.



- 1234567



- 1234657

DAG - Topological Sort

Problem: Given a DAG, find a linear ordering of nodes such that for every directed edge from node A to node B, node A comes before node B in the ordering.

- Topological Sort Algorithm:
  - Algorithm 1: DFS Approach (Explore deeply, order on backtrack).
  - Algorithm 2: Kahn's Algorithm (In-degree based, process nodes with no incoming dependencies first).

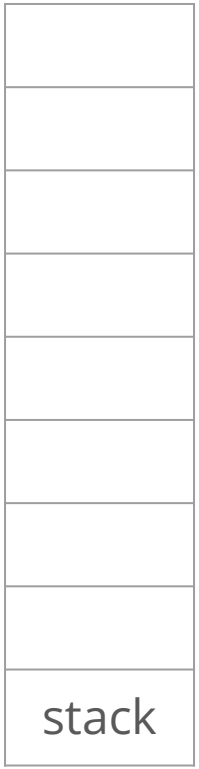
- Result: A topological sort provides a valid order to calculate DP values in DAG DP.

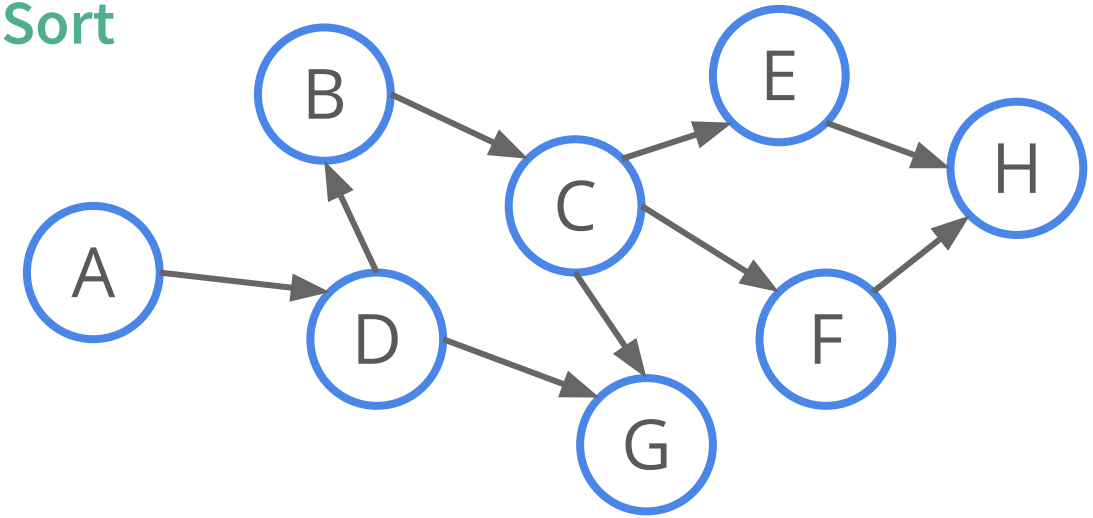
DAG - Topological Sort

- Algorithm 1: DFS Approach

1. Mark all nodes as unvisited.
2. For each unvisited node, start DFS.
3. **DFS(node u)**:
   a. Mark u as visited.
   b. For each neighbor v of u:
      If v is unvisited, DFS(v).
   c. After exploring all neighbors of u, push u onto a stack.
4. Topological order: pop nodes from the stack in order.

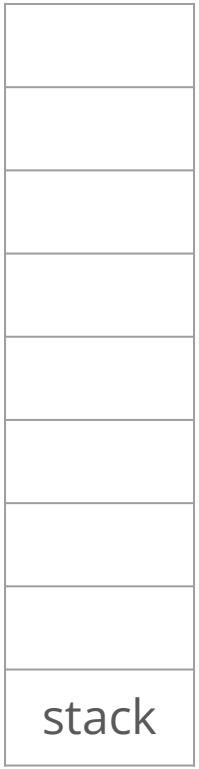
DAG - Topological Sort

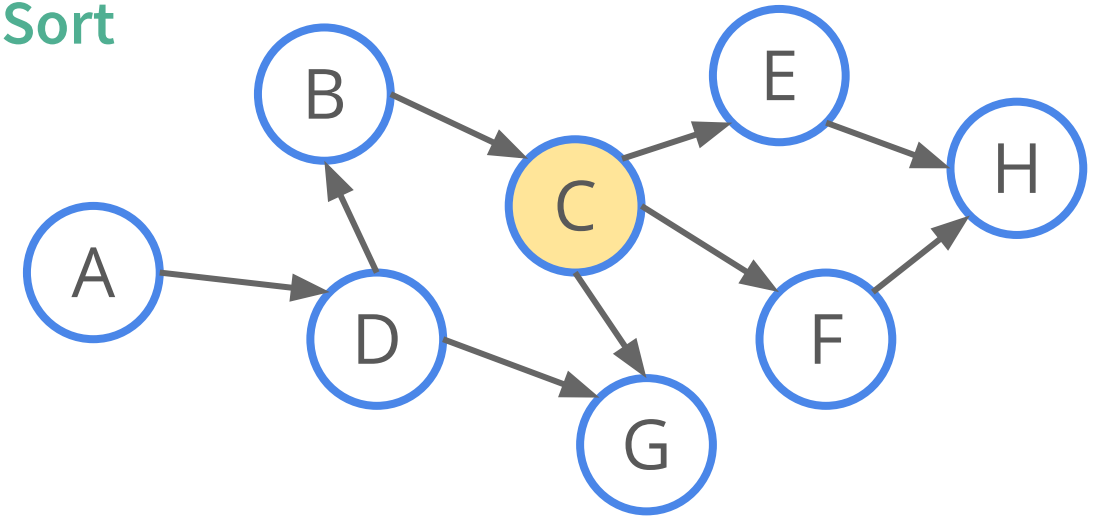




Topological Order							

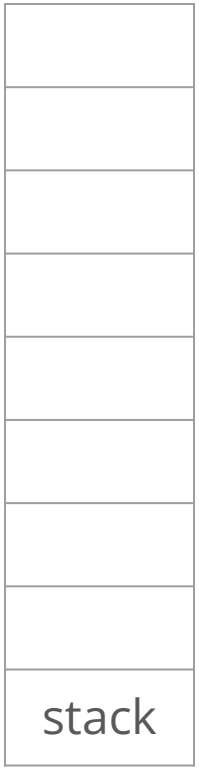
DAG - Topological Sort

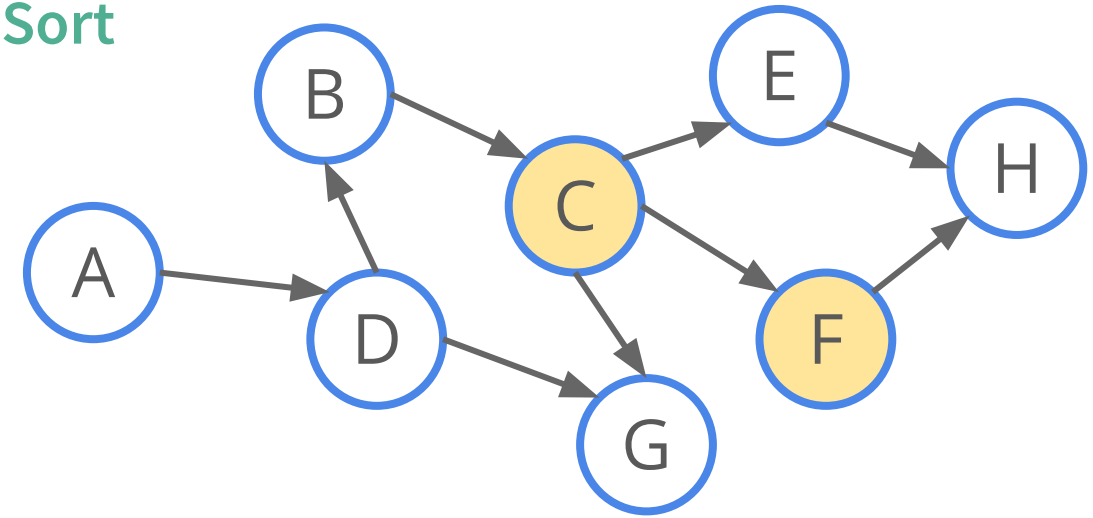




Topological Order							

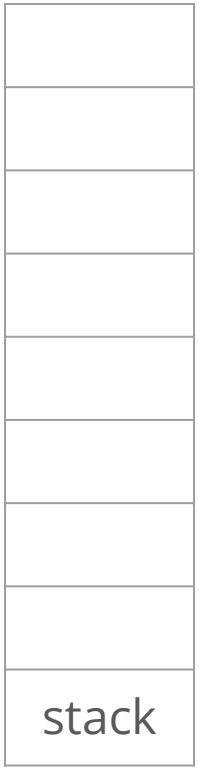
DAG - Topological Sort

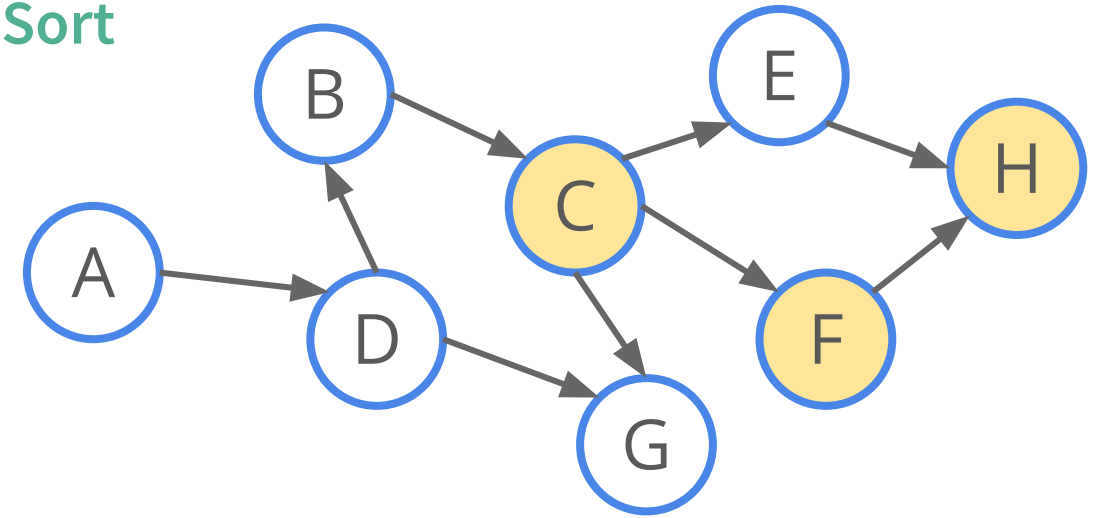




Topological Order							

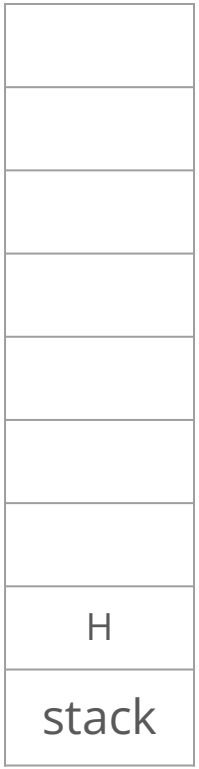
DAG - Topological Sort

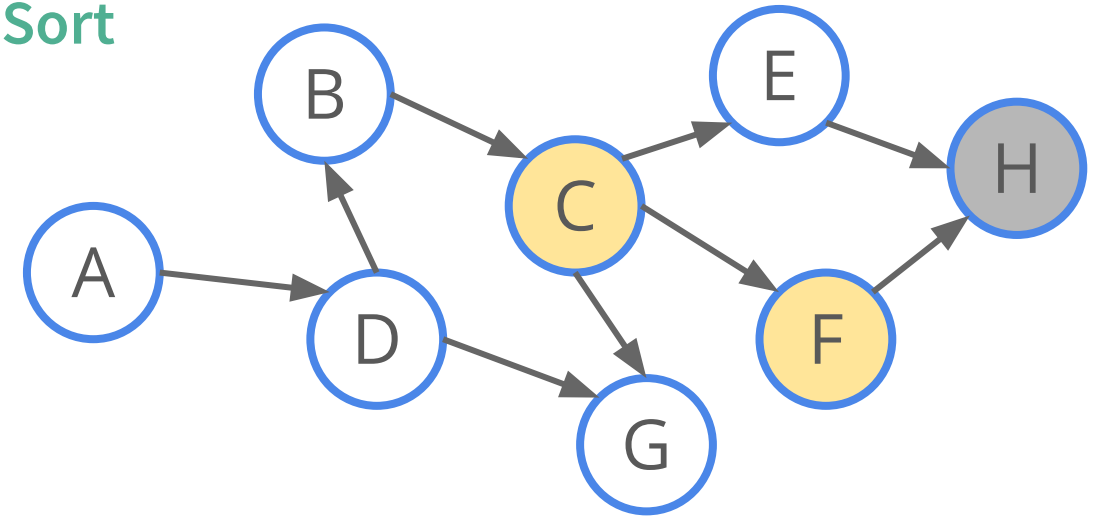




Topological Order							

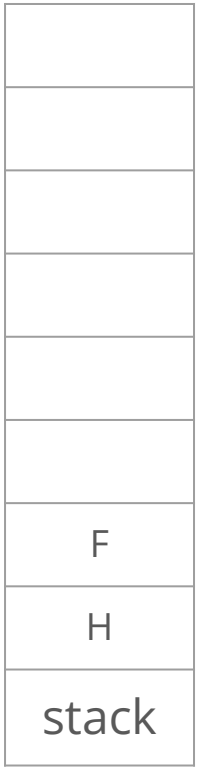
DAG - Topological Sort

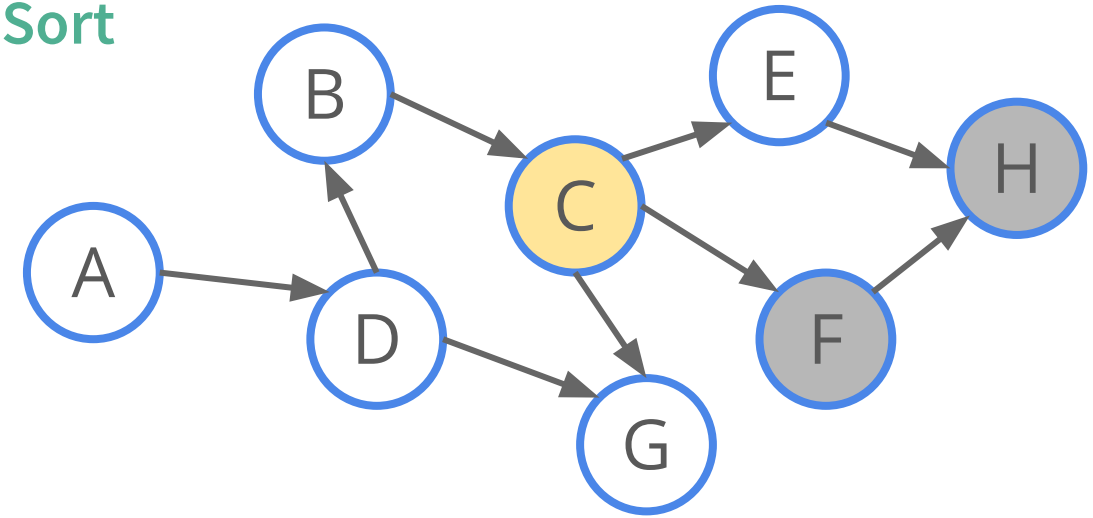




Topological Order							

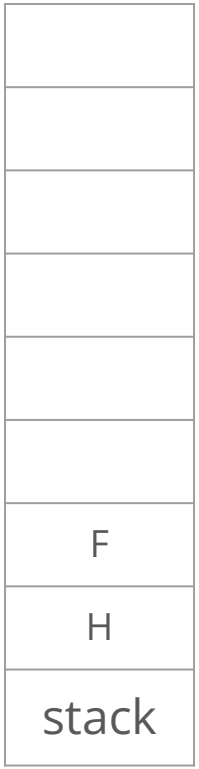
DAG - Topological Sort

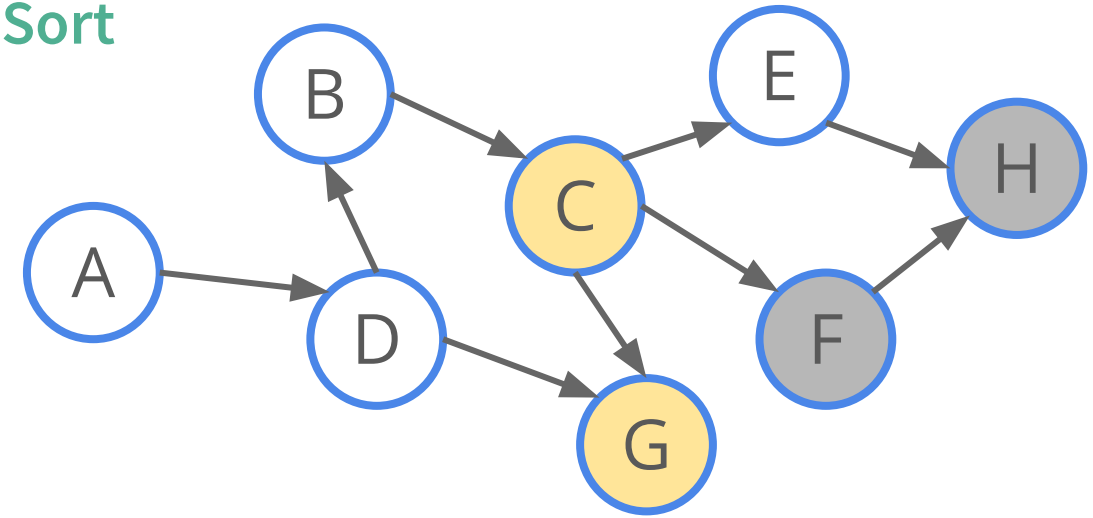




Topological Order							

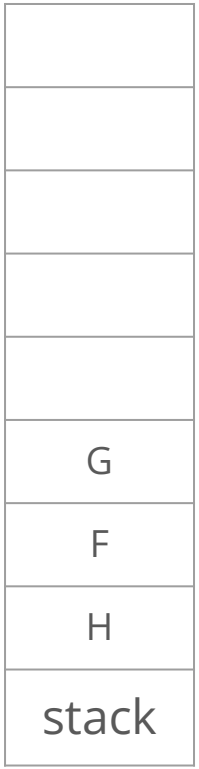
DAG - Topological Sort

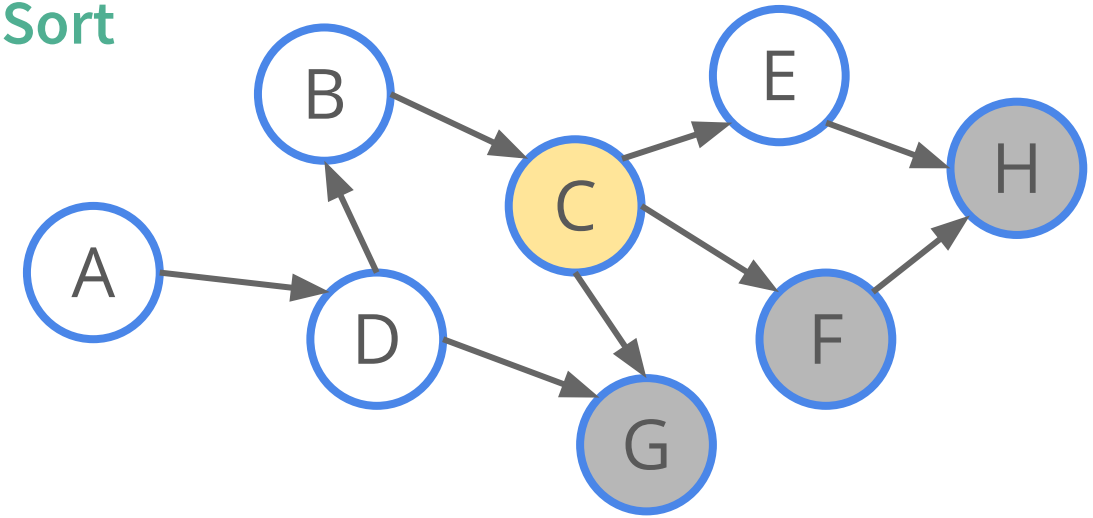




Topological Order							

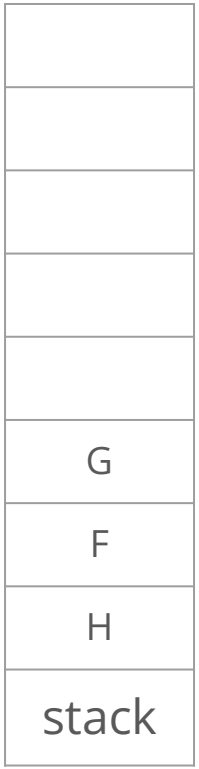
DAG - Topological Sort

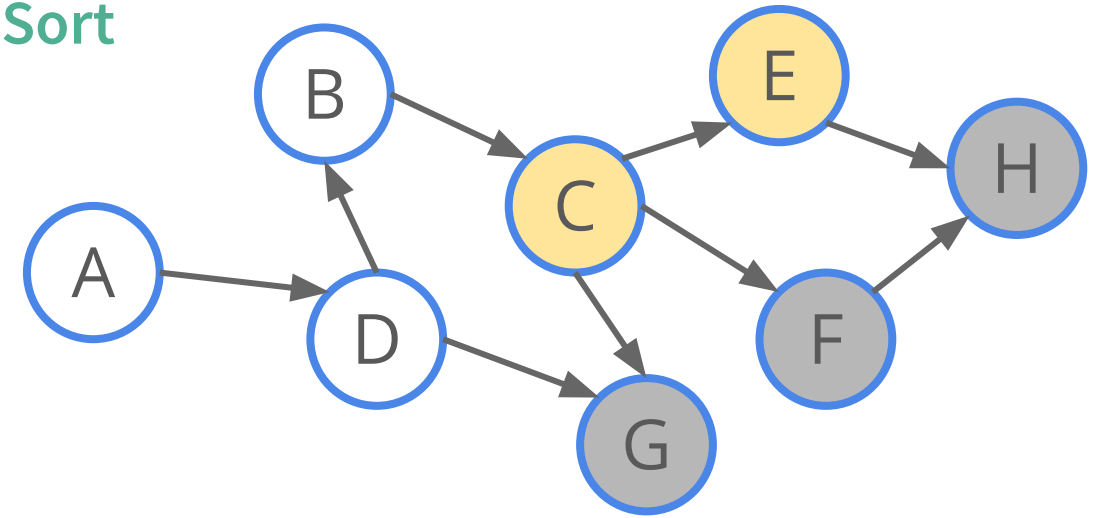




Topological Order							

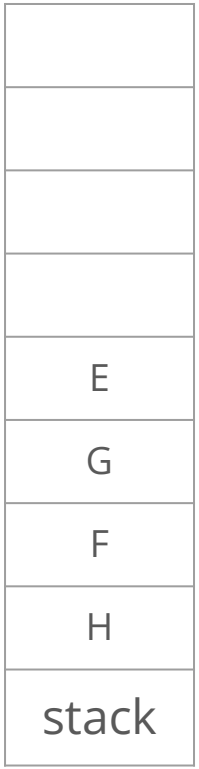
DAG - Topological Sort

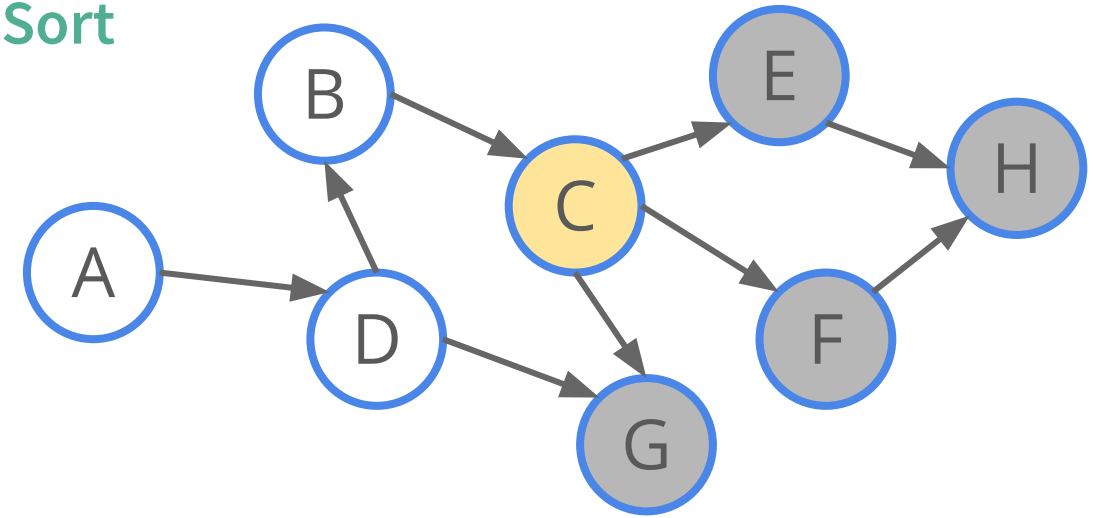




Topological Order							

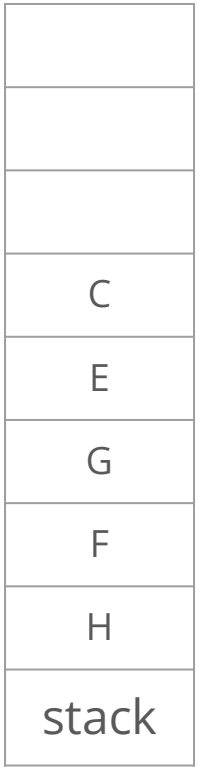
DAG - Topological Sort

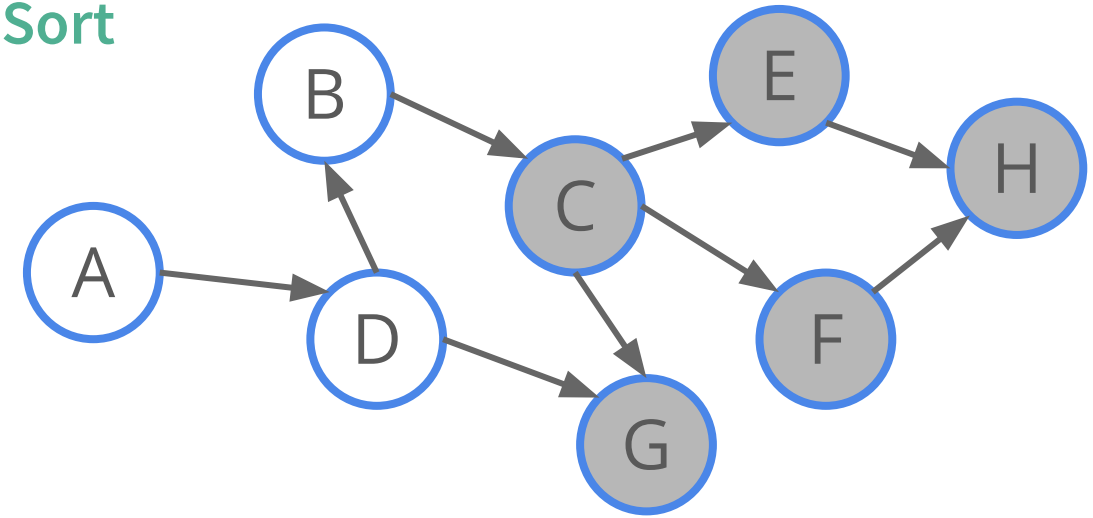




Topological Order							

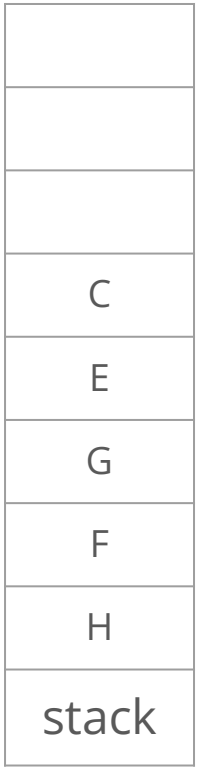
DAG - Topological Sort

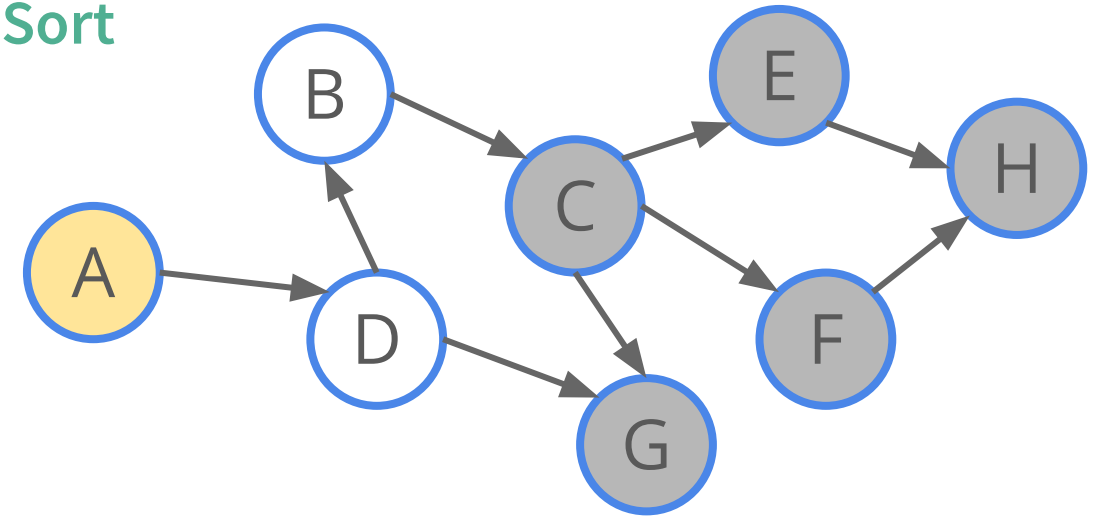




Topological Order							

DAG - Topological Sort

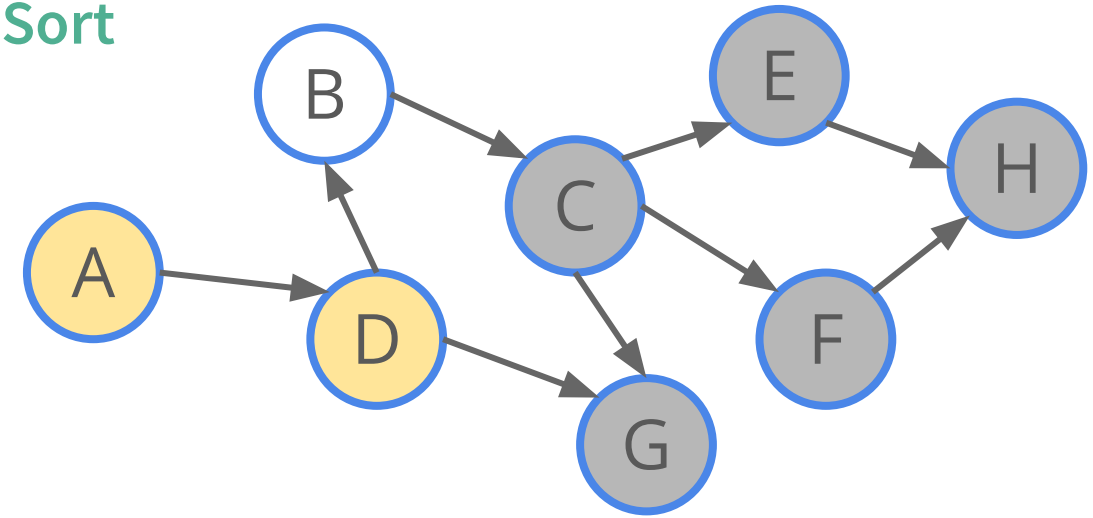




Topological Order							

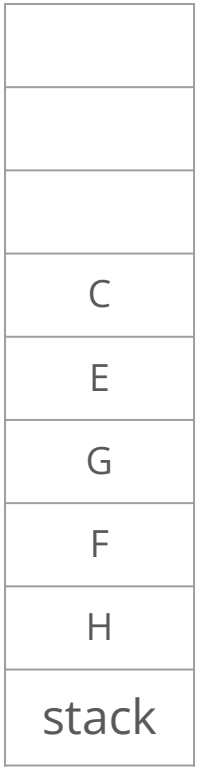
DAG - Topological Sort

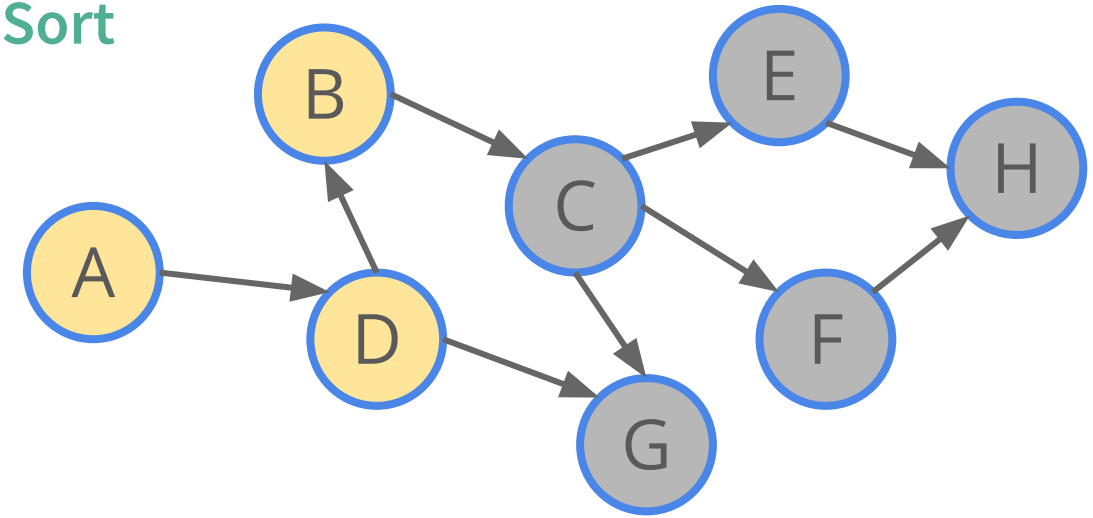
C
E
G
F
H
stack



Topological Order							

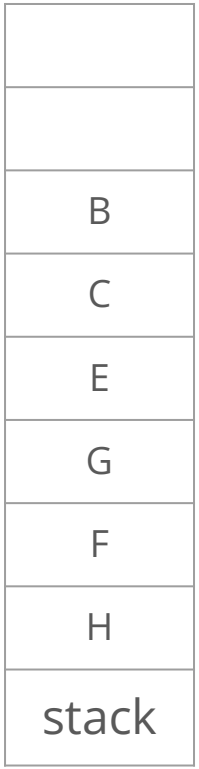
DAG - Topological Sort

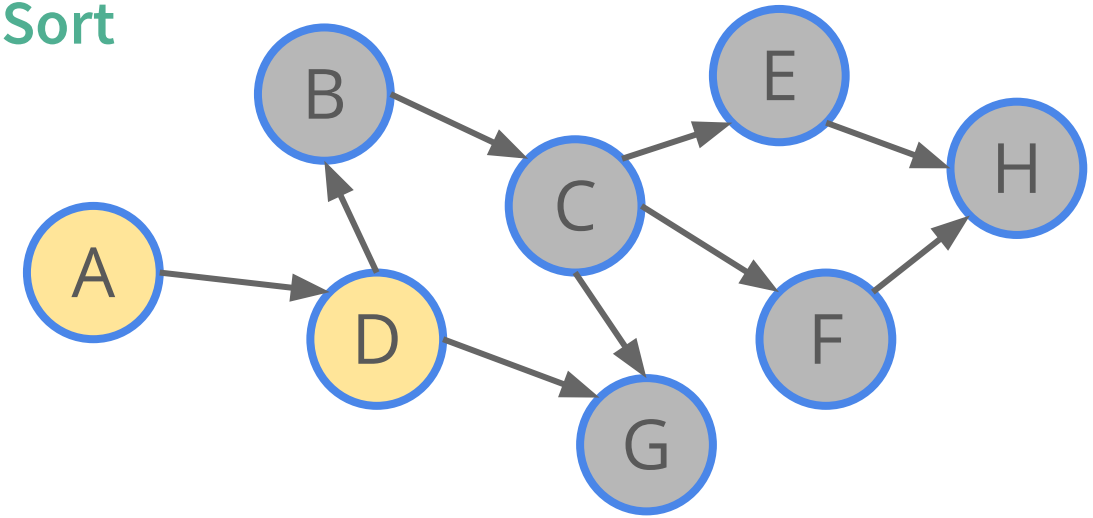




Topological Order							

DAG - Topological Sort

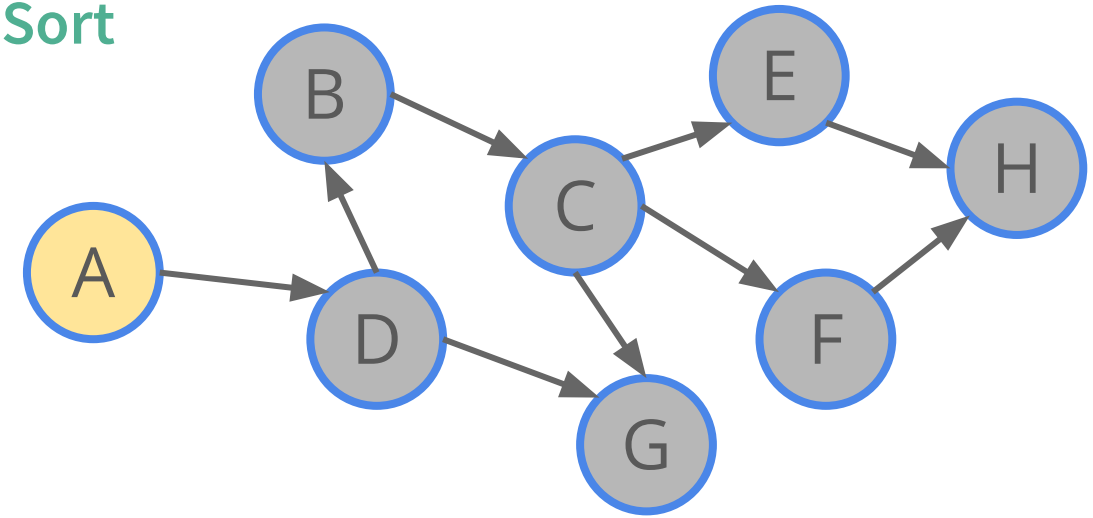




Topological Order							

DAG - Topological Sort

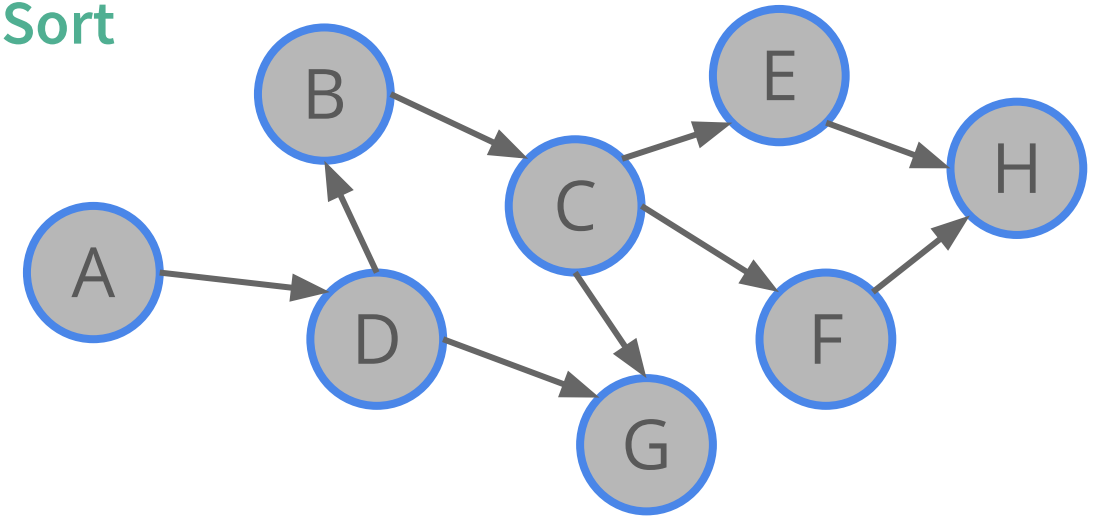
D
B
C
E
G
F
H
stack



Topological Order							

DAG - Topological Sort

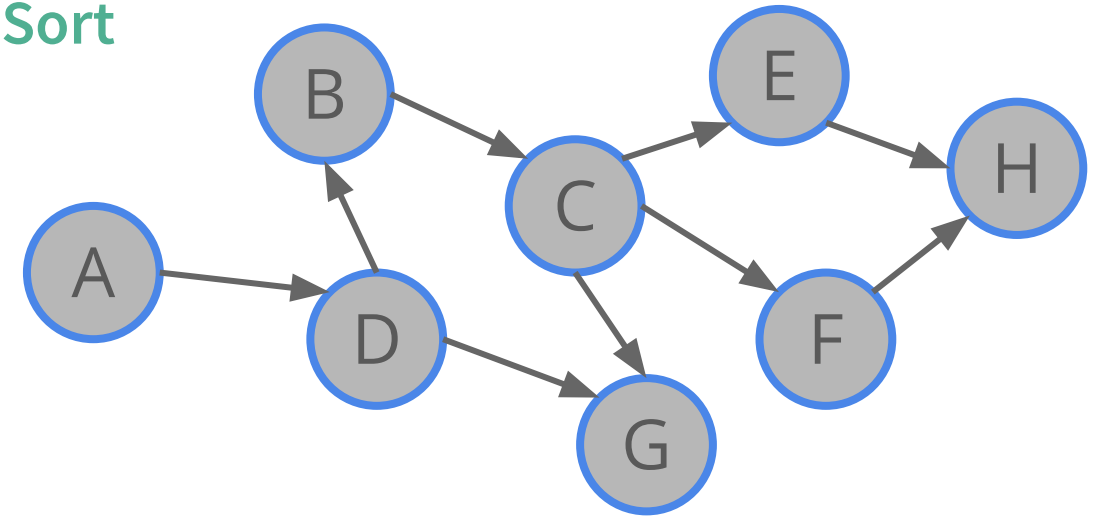
A
D
B
C
E
G
F
H
stack



Topological Order							

DAG - Topological Sort

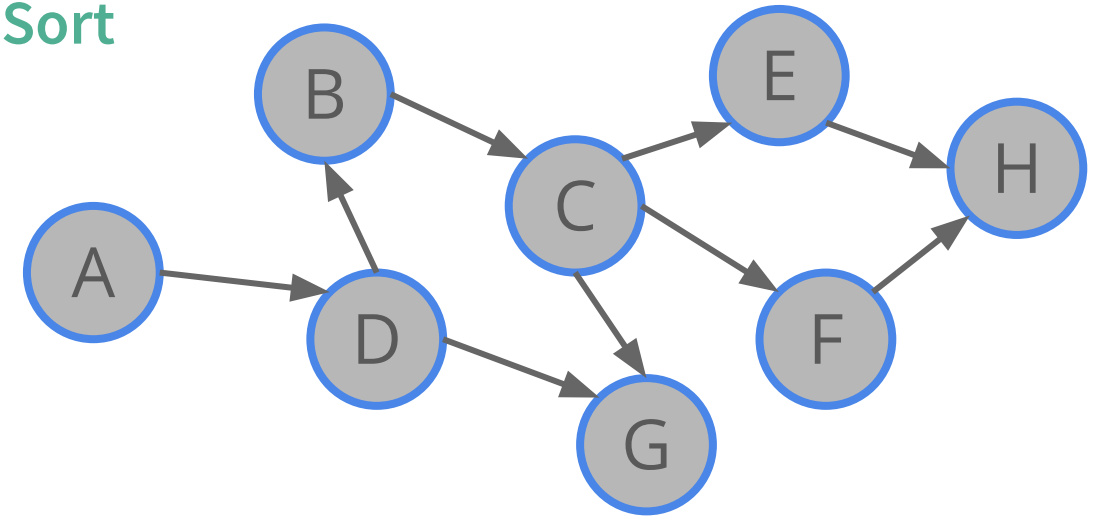
D
B
C
E
G
F
H
stack



Topological Order							
A							

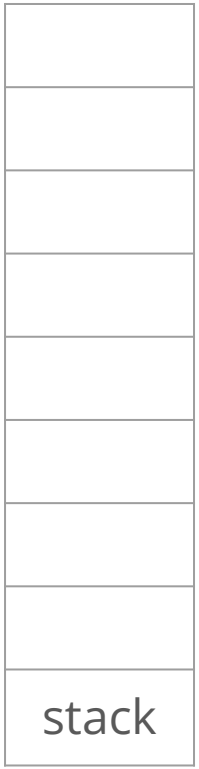
DAG - Topological Sort

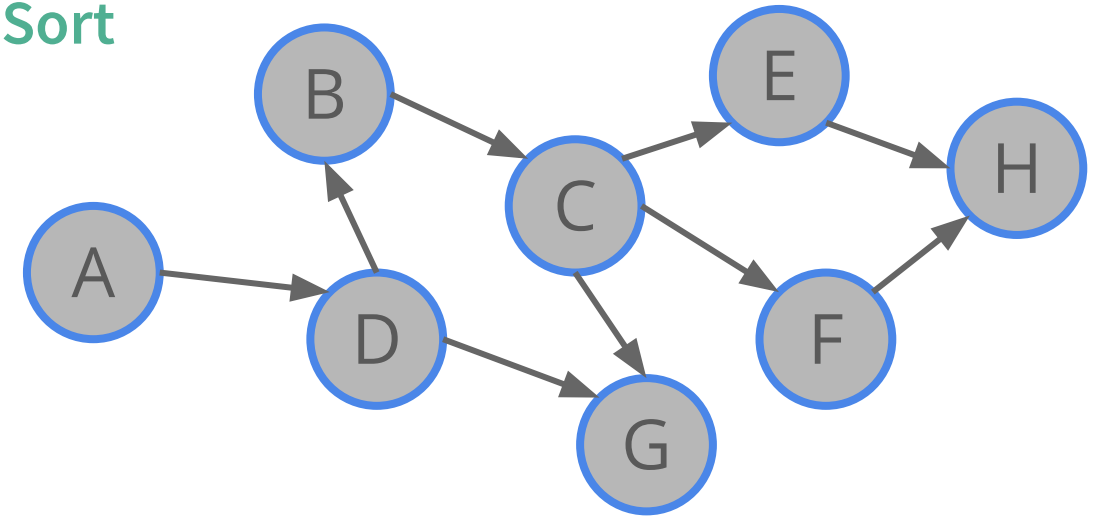
B
C
E
G
F
H
stack



Topological Order							
A	D						

DAG - Topological Sort





Topological Order							
A	D	B	C	E	G	F	H

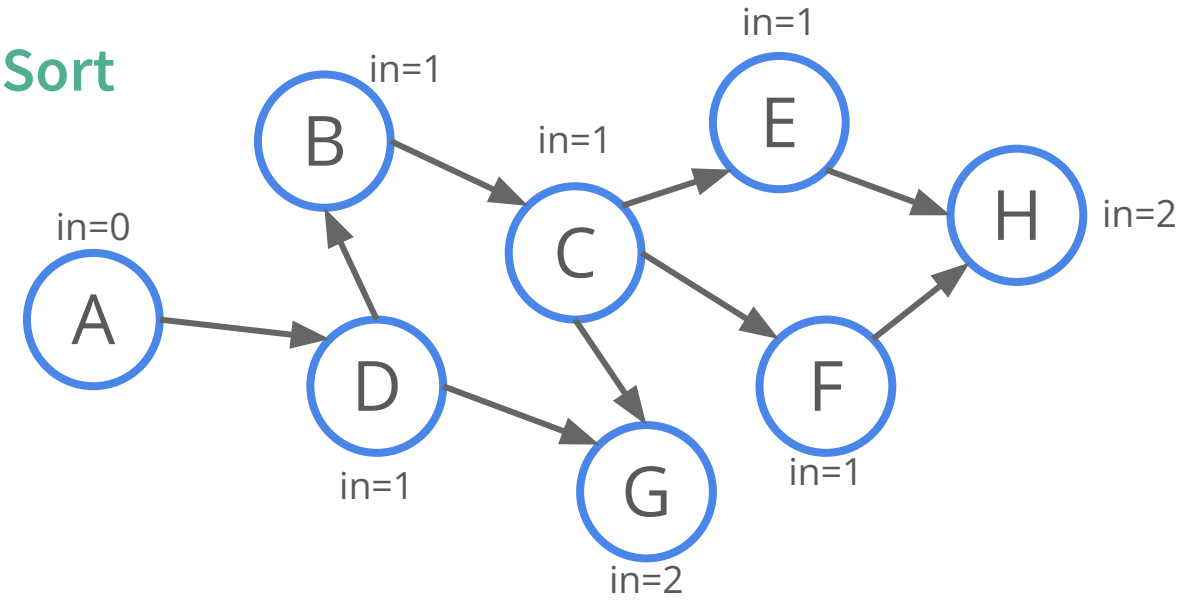
DAG - Topological Sort

- Algorithm 2: Kahn's Algorithm

```
1.  Calculate in-degree for each node (incoming edges).
2.  Create a queue Q with nodes having in-degree 0.
3.  Initialize answer as empty.
4.  While Q is not empty:
    a.  Take a node u from Q.
    b.  Add u to answer.
    c.  For each neighbor v of u:
            Decrease in-degree of v.
            If in-degree of v becomes 0, add v to Q.
5.  answer is the result.
```

DAG - Topological Sort

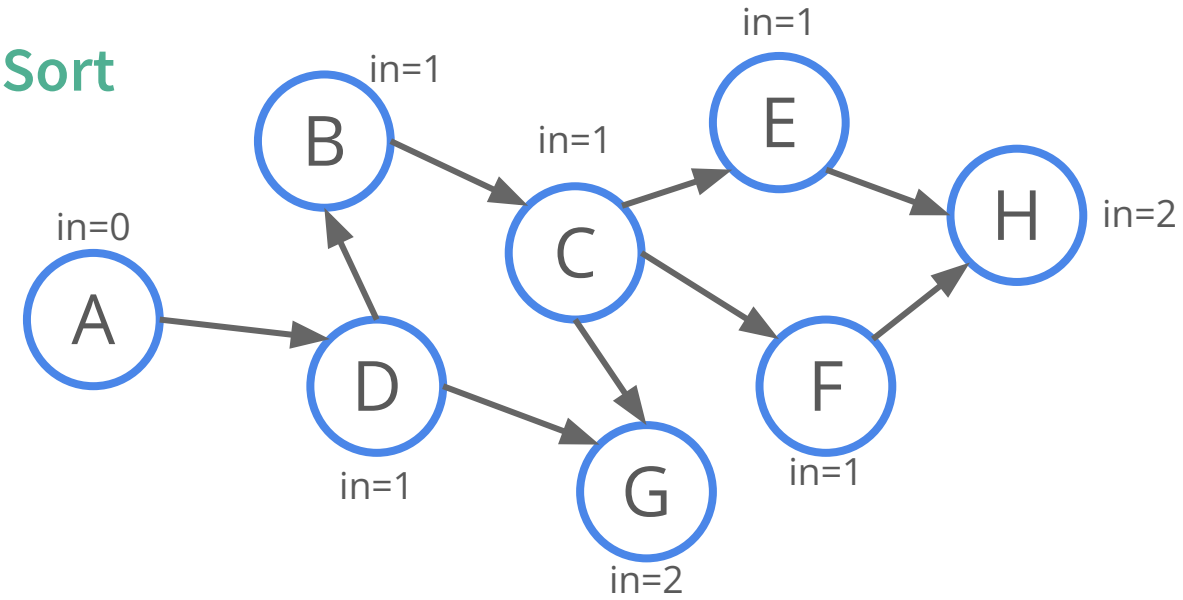
Q:



Topological Order							

DAG - Topological Sort

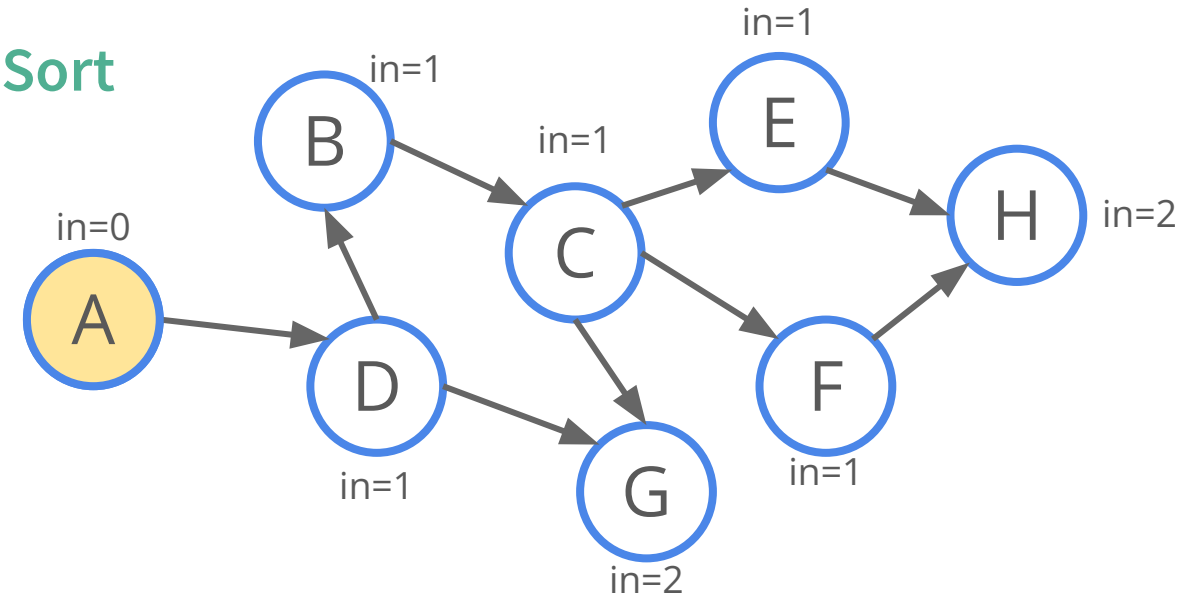
Q: A



Topological Order							

DAG - Topological Sort

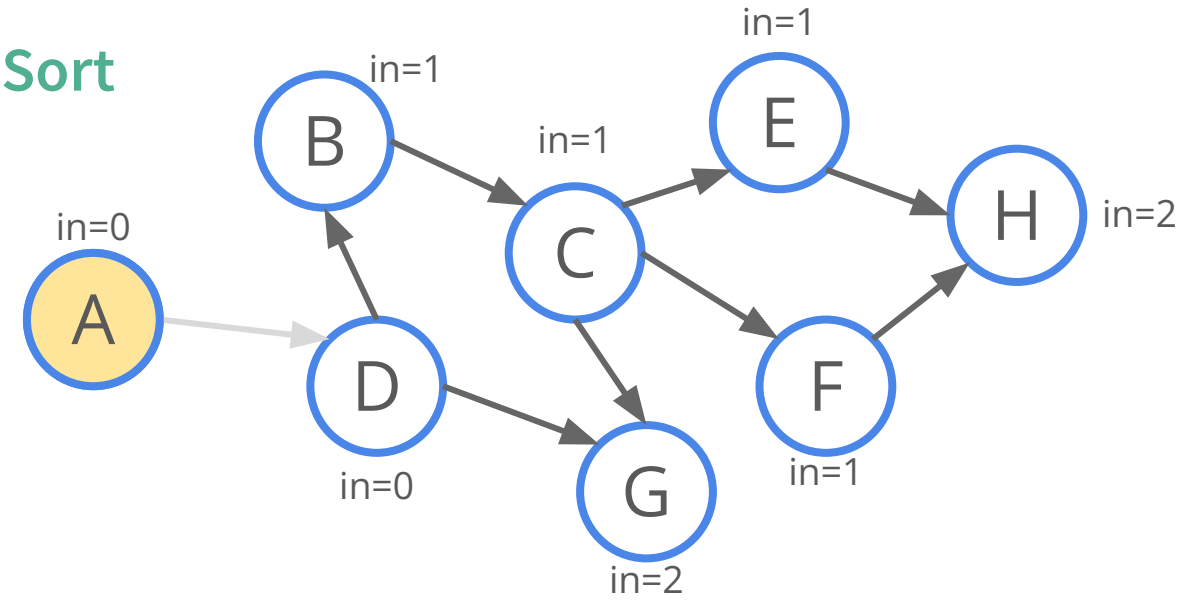
Q:



Topological Order							
A							

DAG - Topological Sort

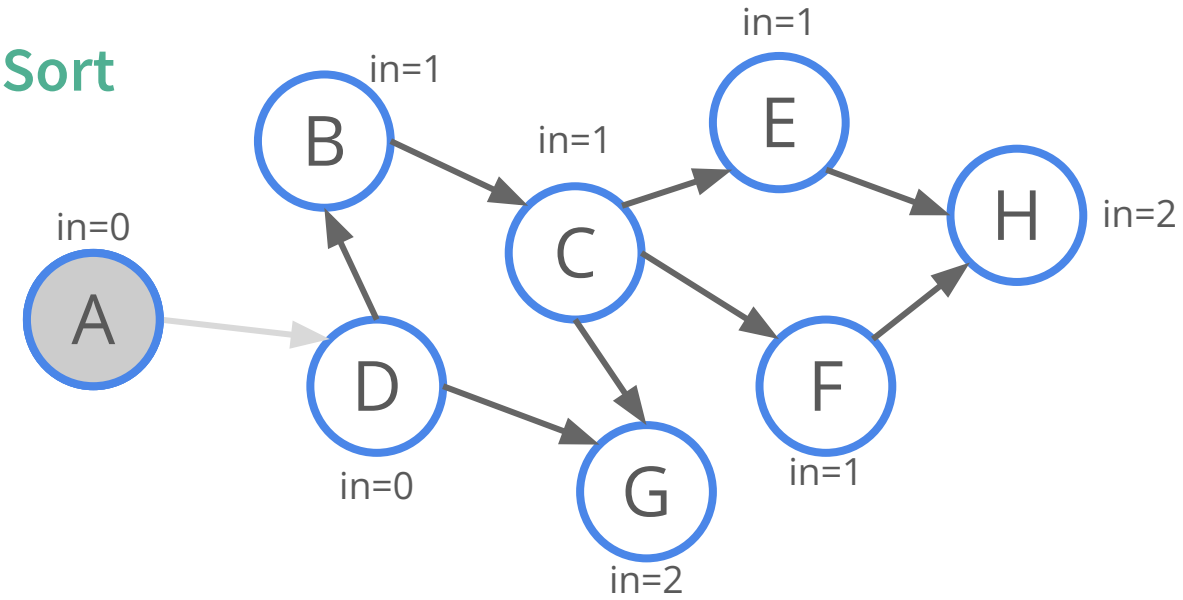
Q:



Topological Order							
A							

DAG - Topological Sort

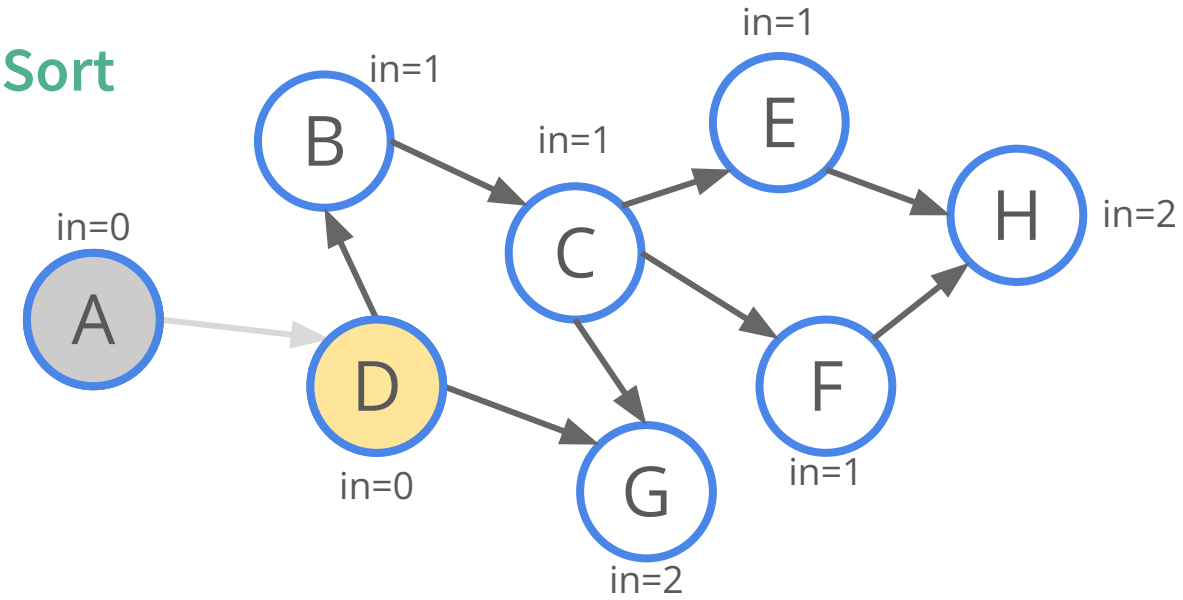
Q: D



Topological Order							
A							

DAG - Topological Sort

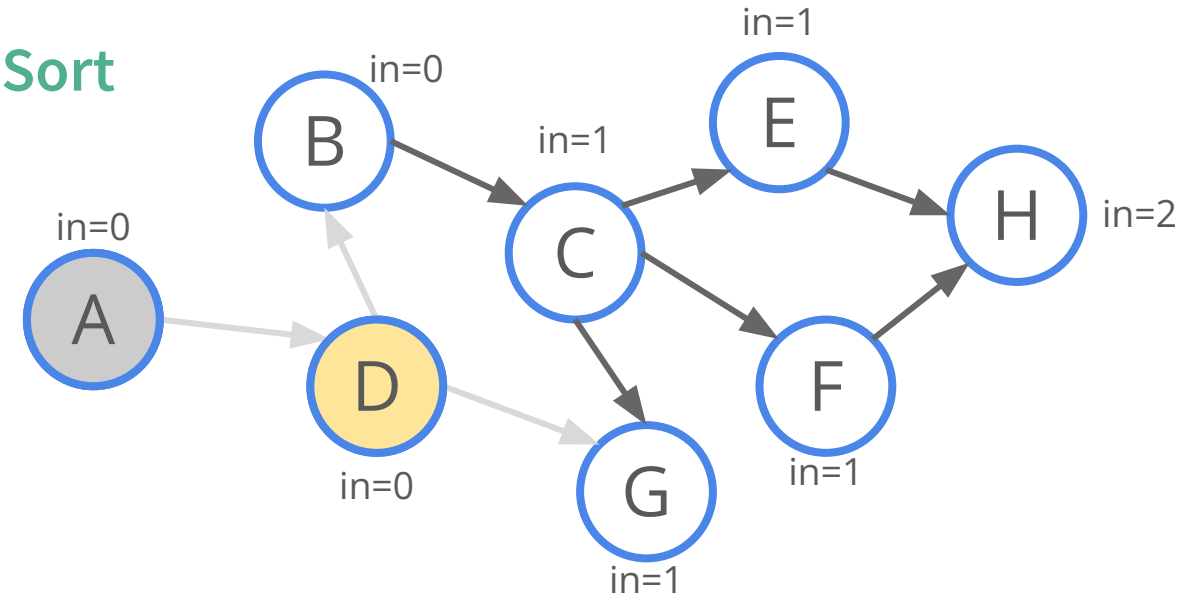
Q:



Topological Order							
A	D						

DAG - Topological Sort

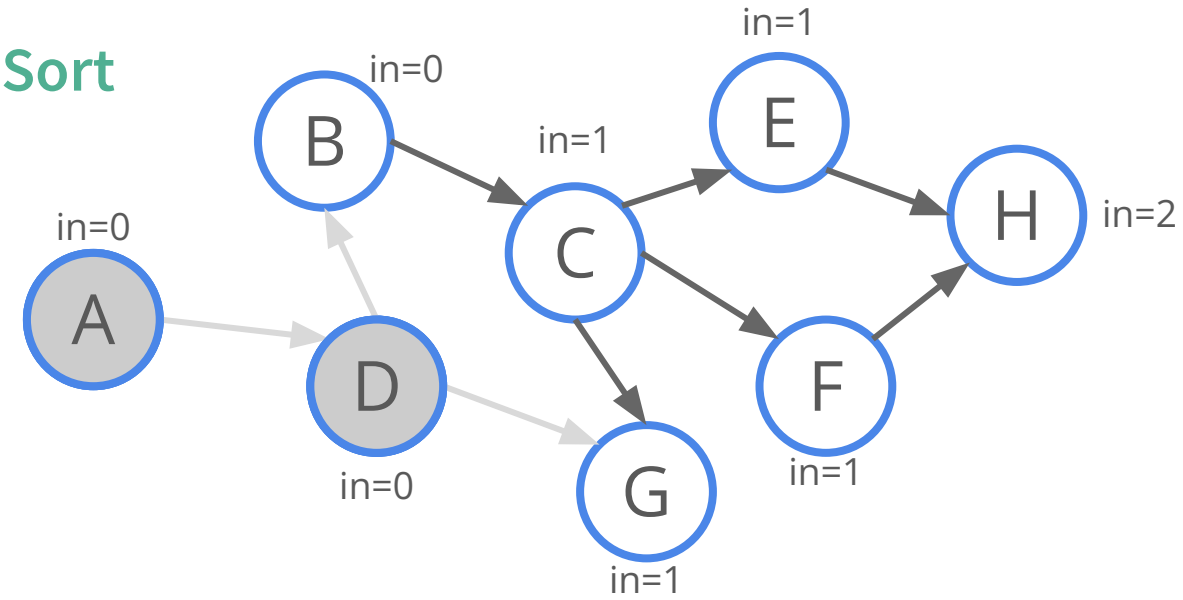
Q:



Topological Order							
A	D						

DAG - Topological Sort

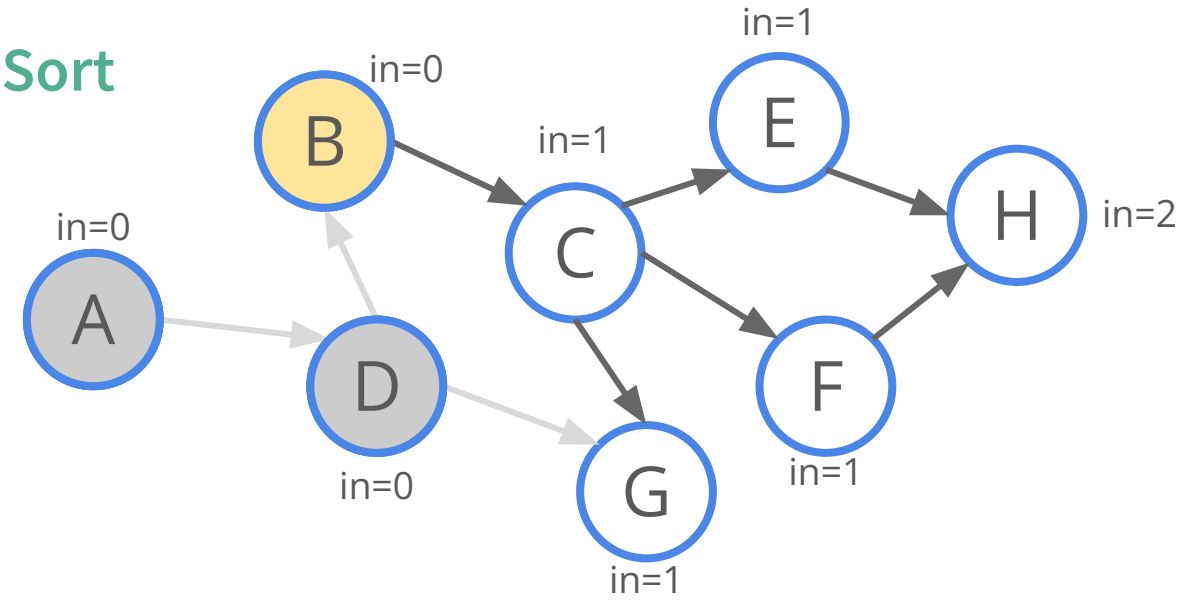
Q: B



Topological Order							
A	D						

DAG - Topological Sort

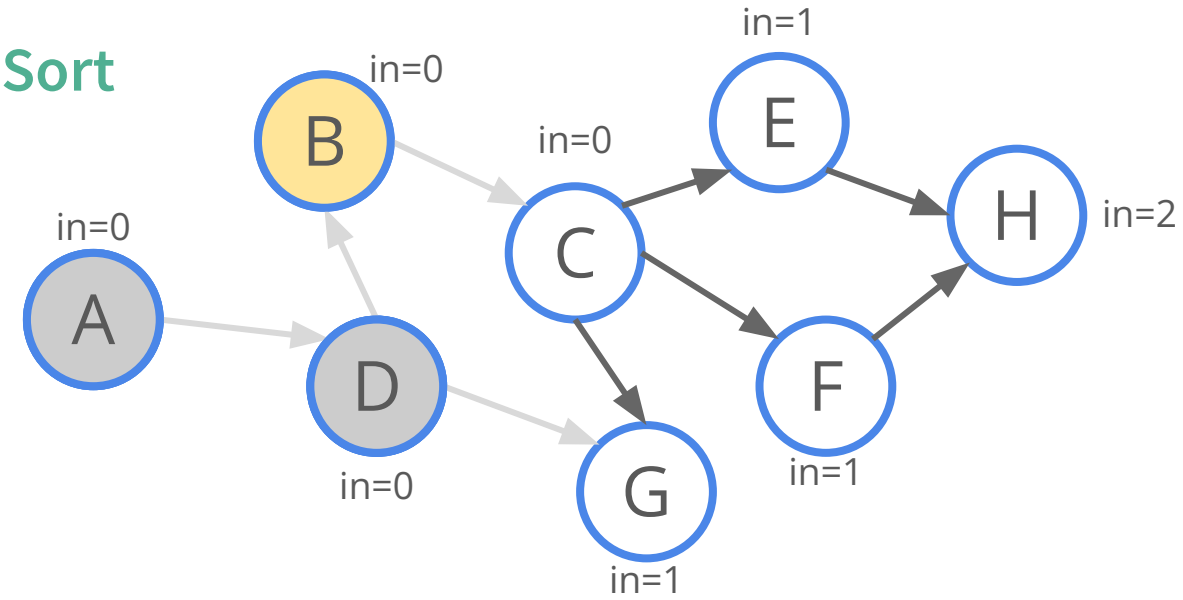
Q:



Topological Order							
A	D	B					

DAG - Topological Sort

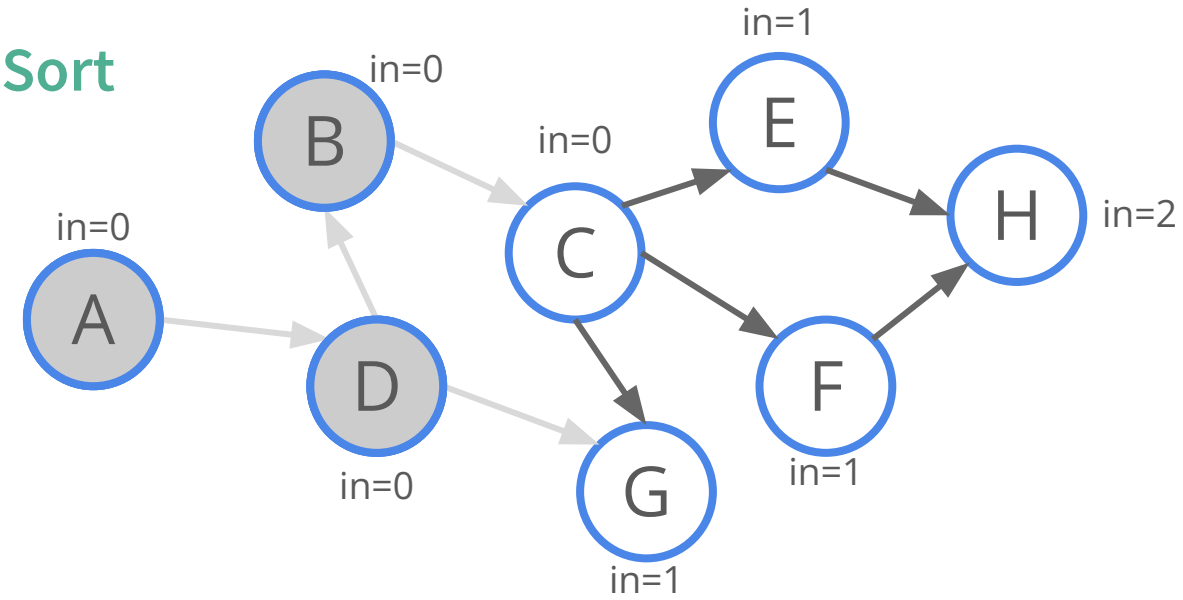
Q:



Topological Order							
A	D	B					

DAG - Topological Sort

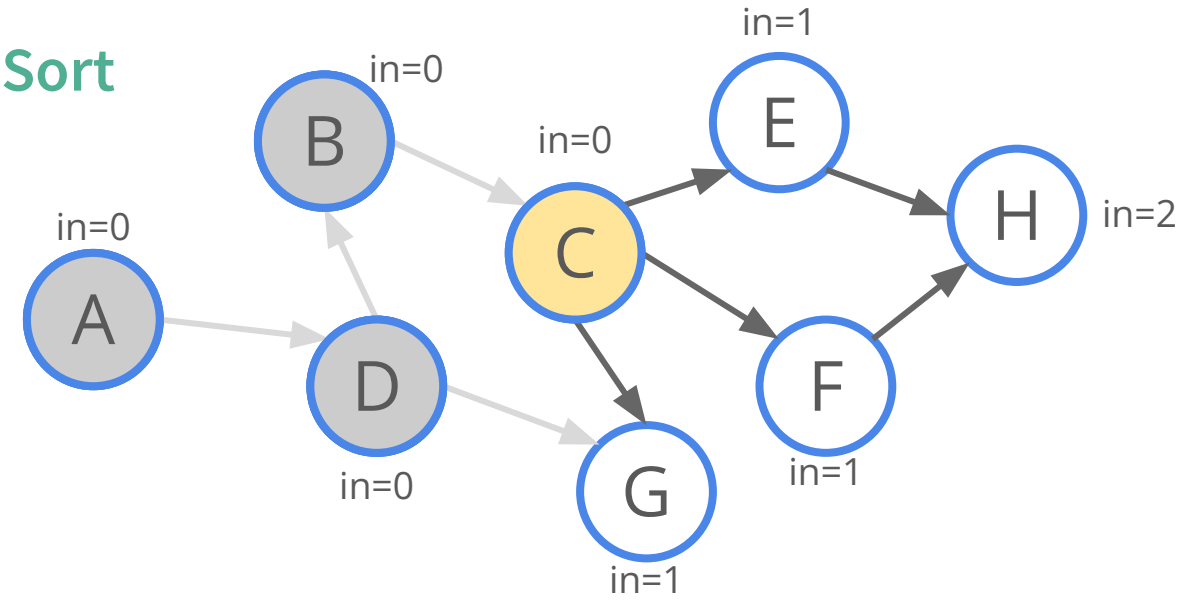
Q: C



Topological Order							
A	D	B					

DAG - Topological Sort

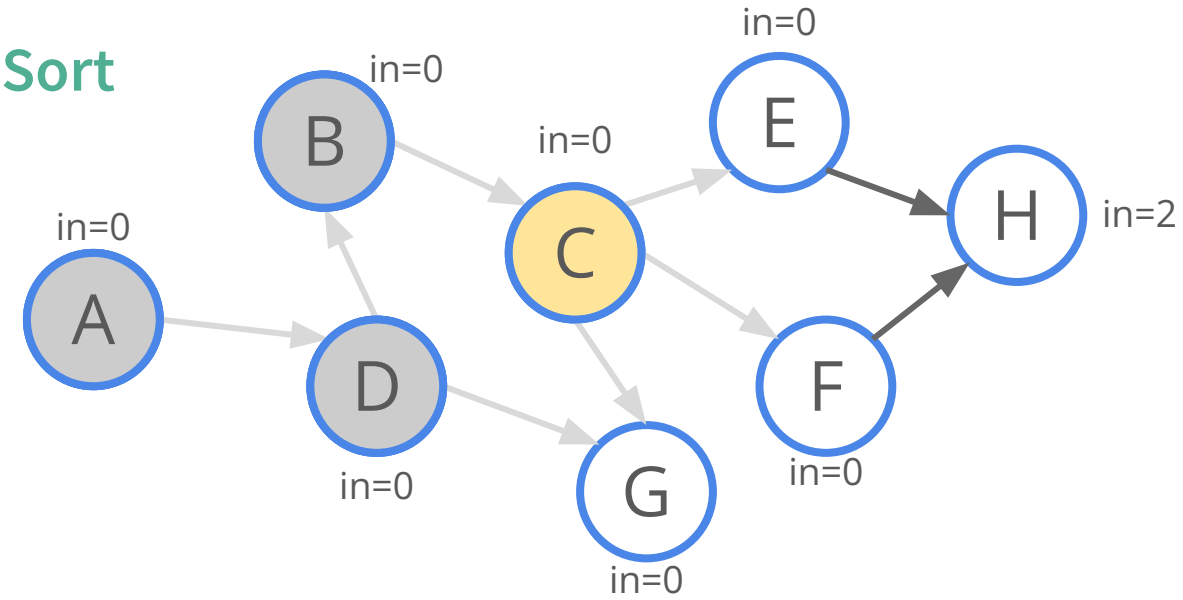
Q:



Topological Order							
A	D	B	C				

DAG - Topological Sort

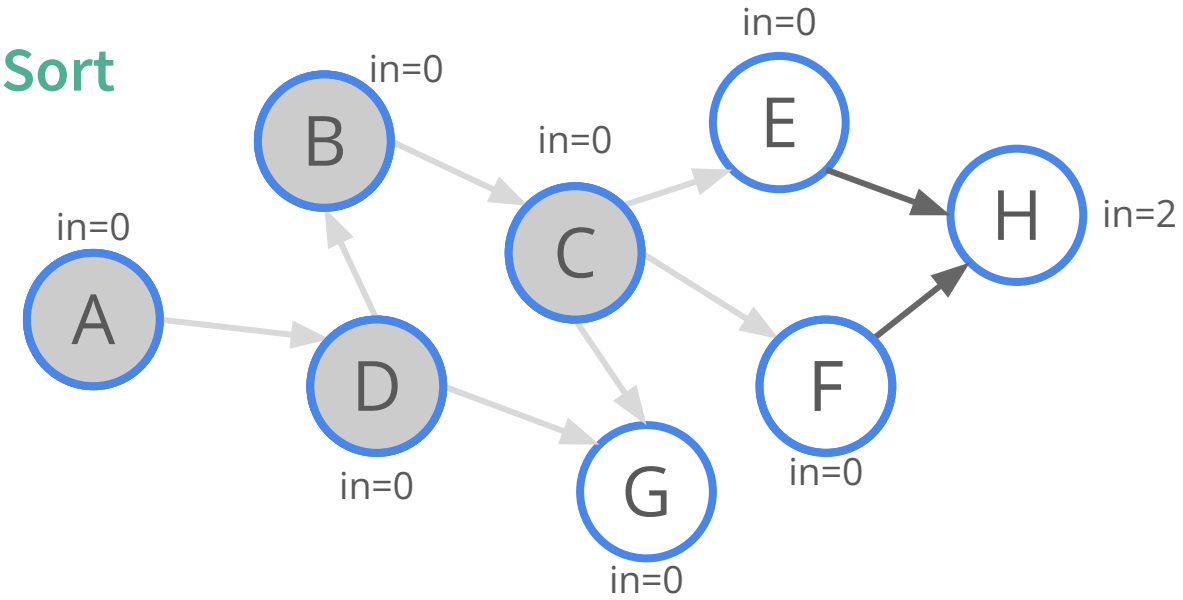
Q:



Topological Order							
A	D	B	C				

DAG - Topological Sort

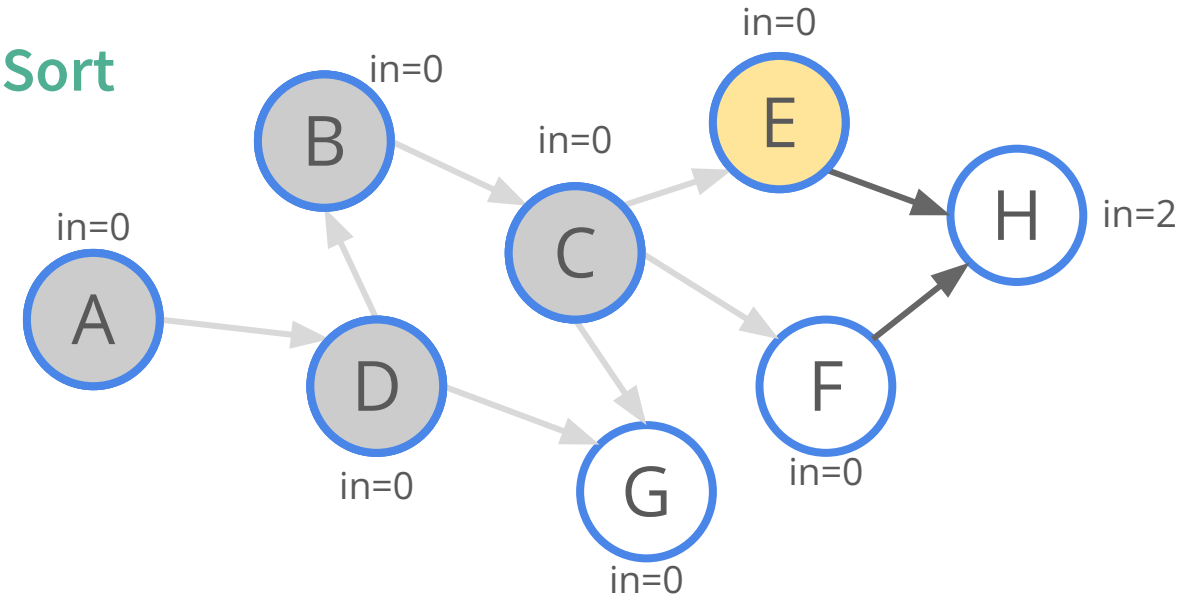
Q: E, F, G



Topological Order							
A	D	B	C				

DAG - Topological Sort

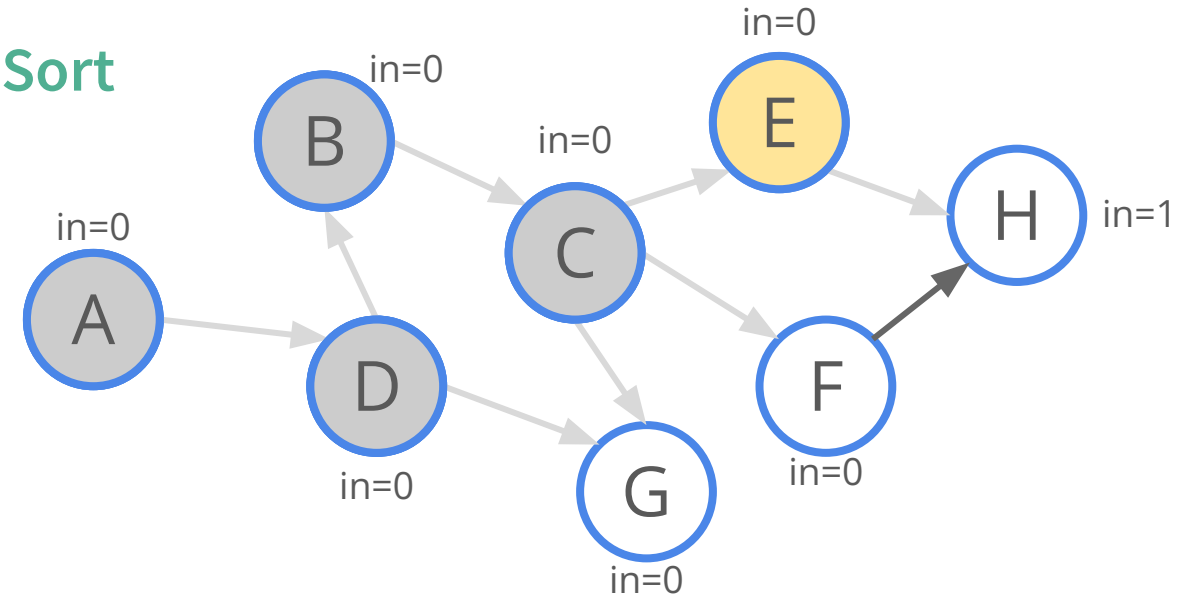
Q: F, G



Topological Order							
A	D	B	C	E			

DAG - Topological Sort

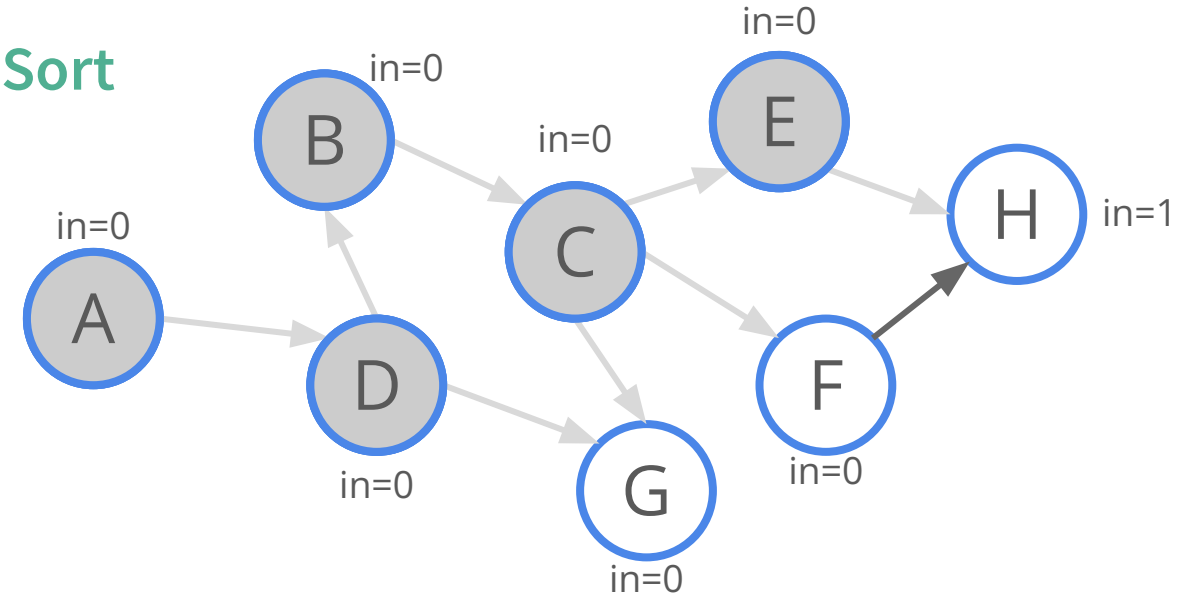
Q: F, G



Topological Order							
A	D	B	C	E			

DAG - Topological Sort

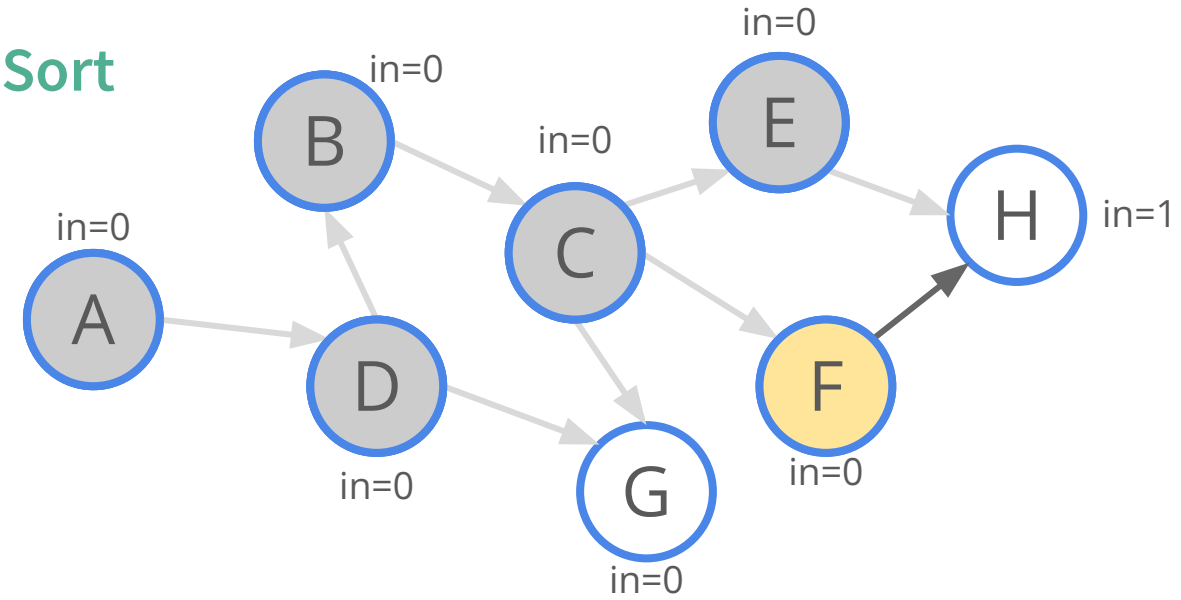
Q: F, G



Topological Order							
A	D	B	C	E			

DAG - Topological Sort

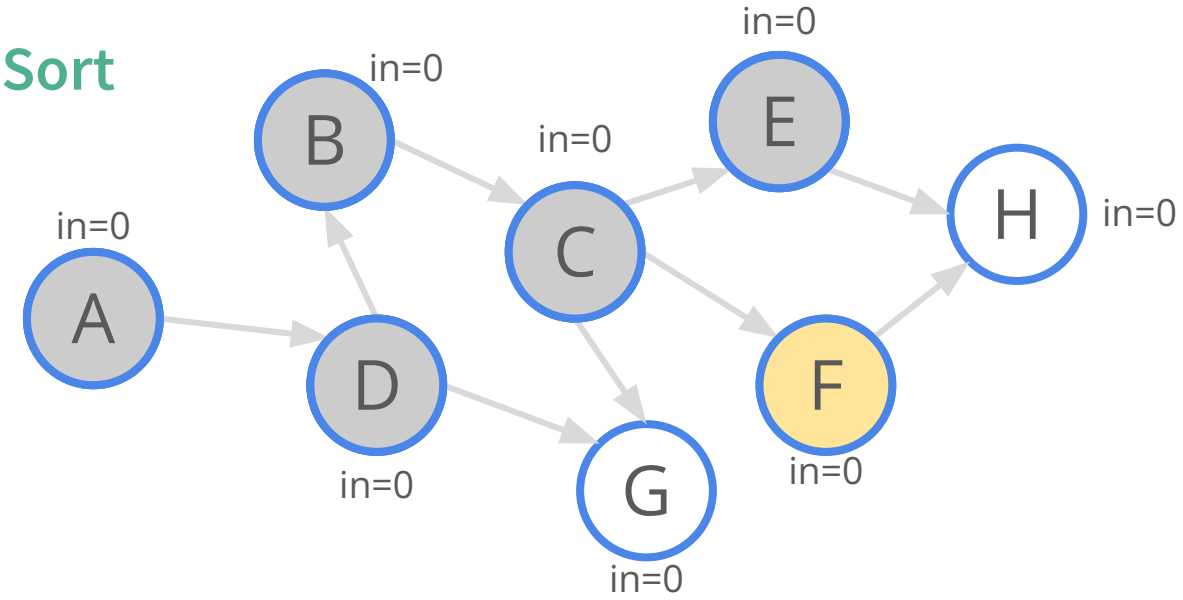
Q: G



Topological Order							
A	D	B	C	E	F		

DAG - Topological Sort

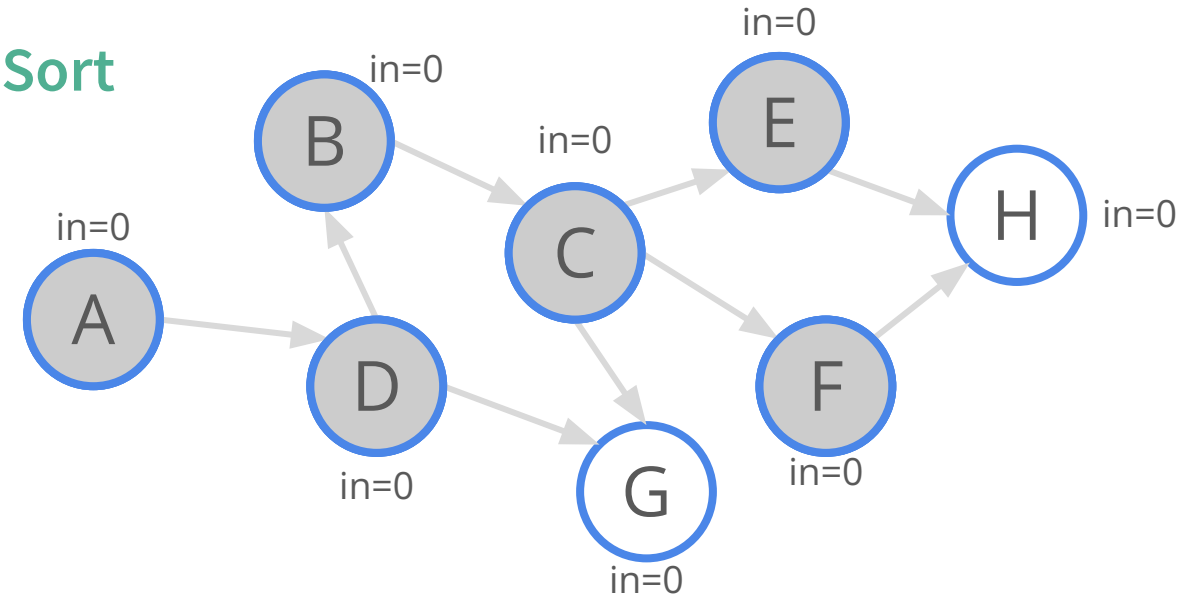
Q: G



Topological Order							
A	D	B	C	E	F		

DAG - Topological Sort

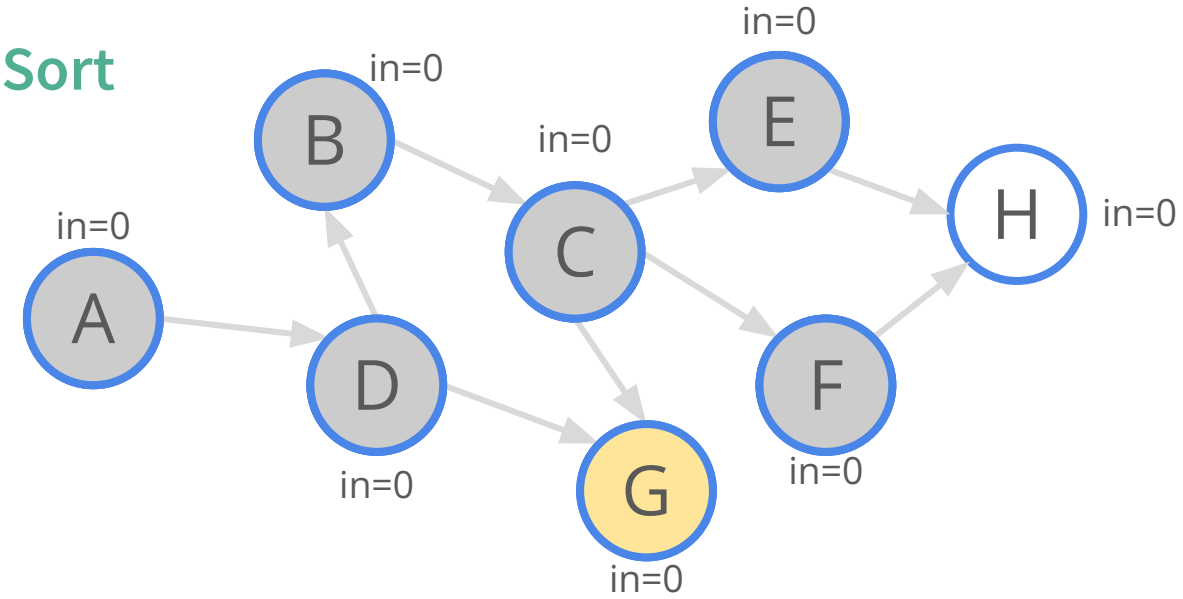
Q: G, H



Topological Order							
A	D	B	C	E	F		

DAG - Topological Sort

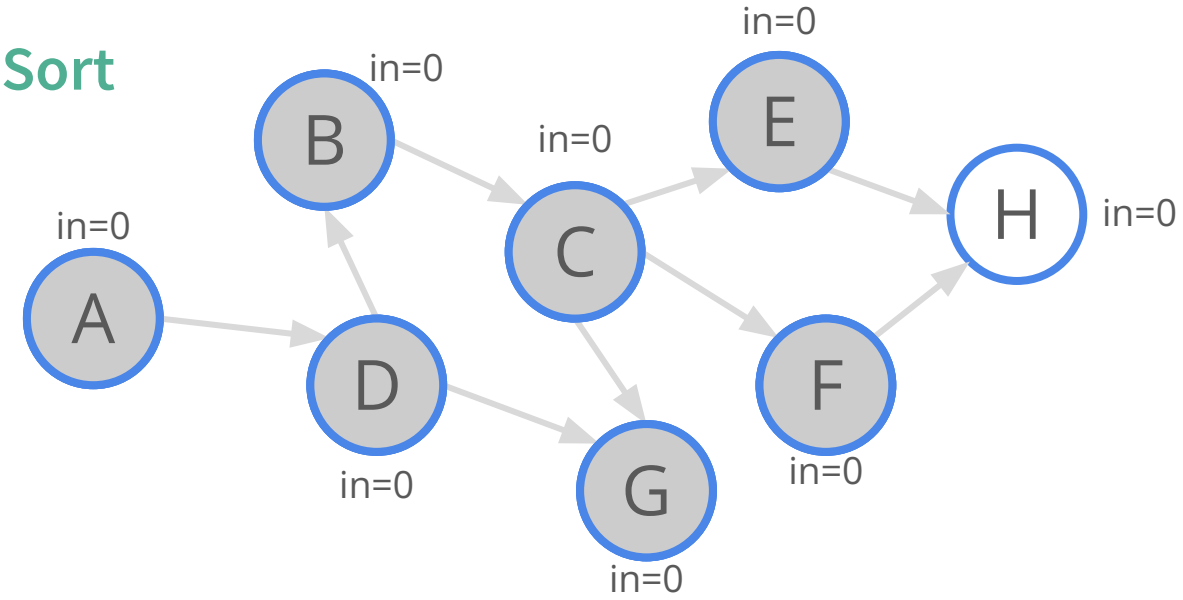
Q: H



Topological Order							
A	D	B	C	E	F	G	

DAG - Topological Sort

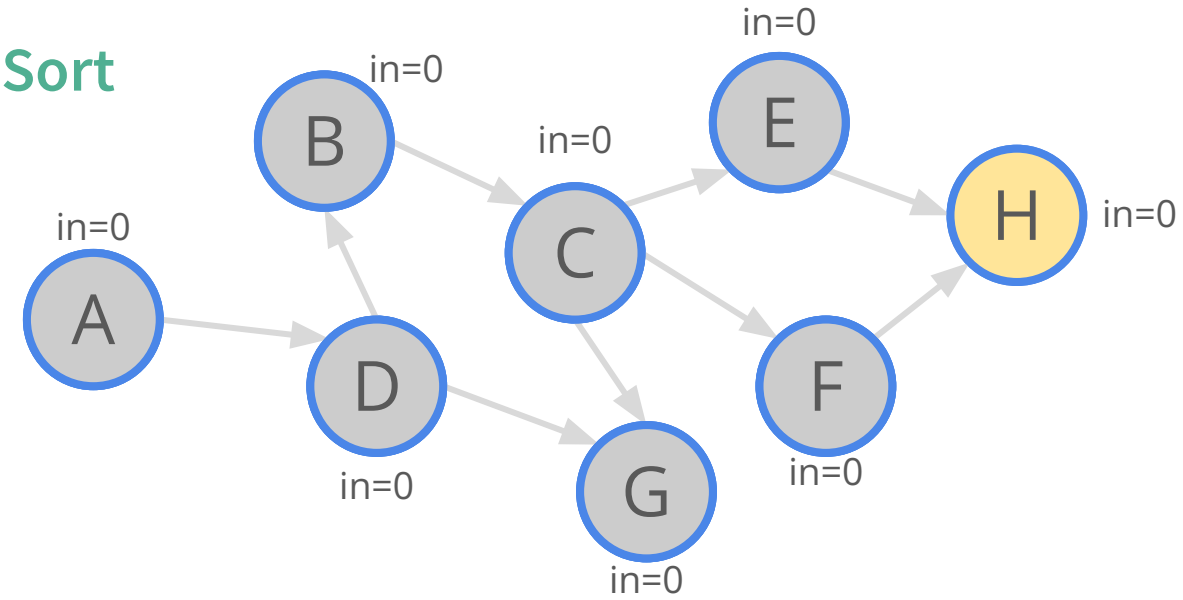
Q: H



Topological Order							
A	D	B	C	E	F	G	

DAG - Topological Sort

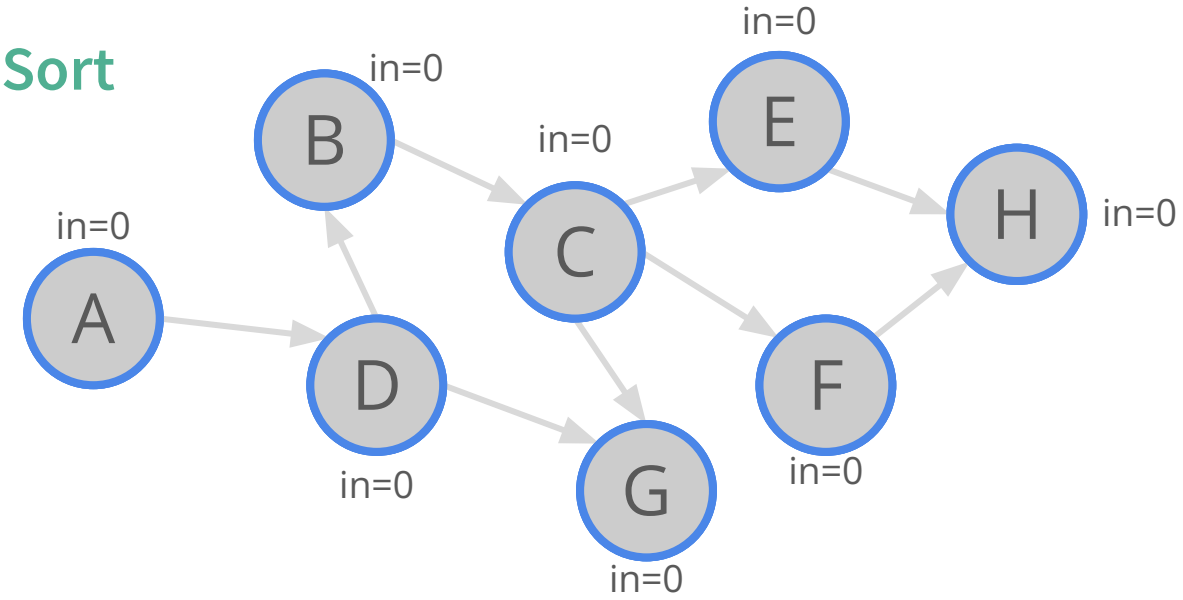
Q:



Topological Order							
A	D	B	C	E	F	G	H

DAG - Topological Sort

Q:



Topological Order							
A	D	B	C	E	F	G	H

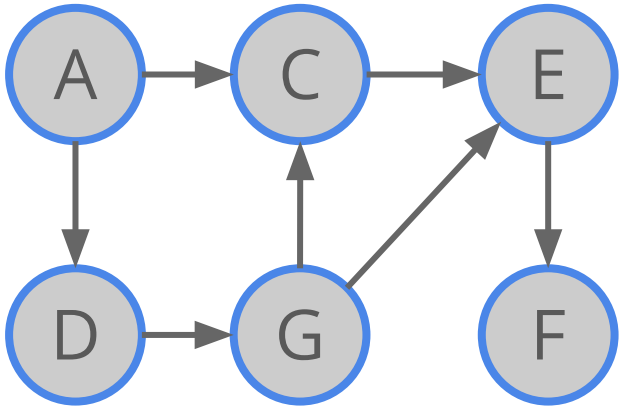
DAG DP Example Task - Counting Paths

Problem: Given a DAG. Count the number of paths from node A to any node x.

e.g.: count number of paths from A to F

1. A → C → E → F
2. A → D → G → E → F
3. A → D → G → C → E → F

- There are three paths in total

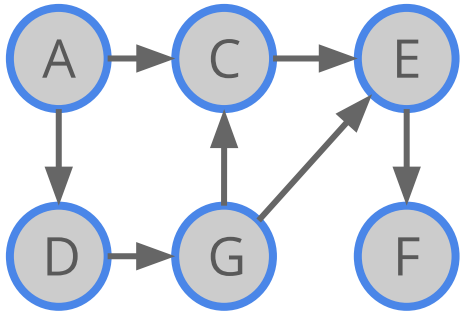


DAG DP Example Task - Counting Paths

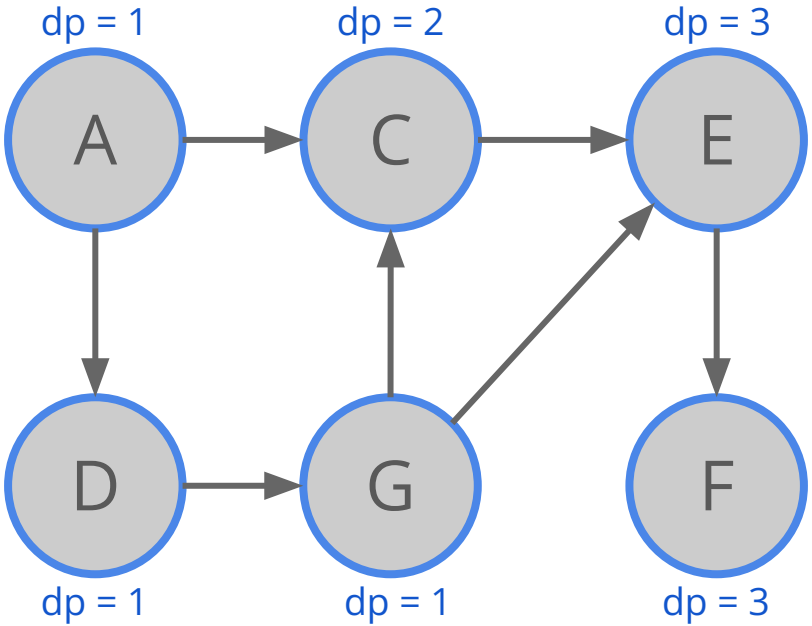
Problem: Given a DAG. Count the number of paths from node A to any node x.

- **DP State:** paths[x] = number of paths from node A to node x.
- **Base Case:** path[A] = 1. paths[x] = 0 initially for other nodes.

- Since the graph is acyclic, the transition formula can be:
  $paths[x] = paths[v_1] + paths[v_2] + \ldots + paths[v_k]$,
  where $v_{1..k}$ are the nodes from which there is a direct edge to x
- To ensure we already have the answer to the subproblem, use **topological sort** to get a valid order.



DAG DP Example Task - Counting Paths



DAG DP Practice Task - HKOJ A300 DAG DP

- https://judge.hkoi.org/task/A300
- Let's spend some time to code this task together.

DAG DP More Practice Task

- You should be aware that DAG is not always given to you easily. It is sometimes hidden.
- M1739 How to Run Fast
  - Similar settings as A300 DAG DP.
  - Edges are bidirectional and have weight.
  - Alice will only use the shortest path from 1 to N. Count the number of paths.

DAG DP More Practice Task

- You should be aware that DAG is not always given to you easily. It is sometimes hidden.
- M1739 How to Run Fast
  - Similar settings as A300 DAG DP.
  - Edges are bidirectional and have weight.
  - Alice will only use the shortest path from 1 to N. Count the number of paths.
  - **Run shortest path algorithm from node 1.**
  - Suppose dist[i] stored shortest path distance from node 1 to node i.
  - Alice can only use edge u->v if dist[u] + weight == dist[v].
  - By the property of shortest path, these valid edges can only form a DAG.
- DAG can also hide in grids, e.g., a grid that only allow going right and down.

Table of Contents

Tree DP - Why Tree DP?

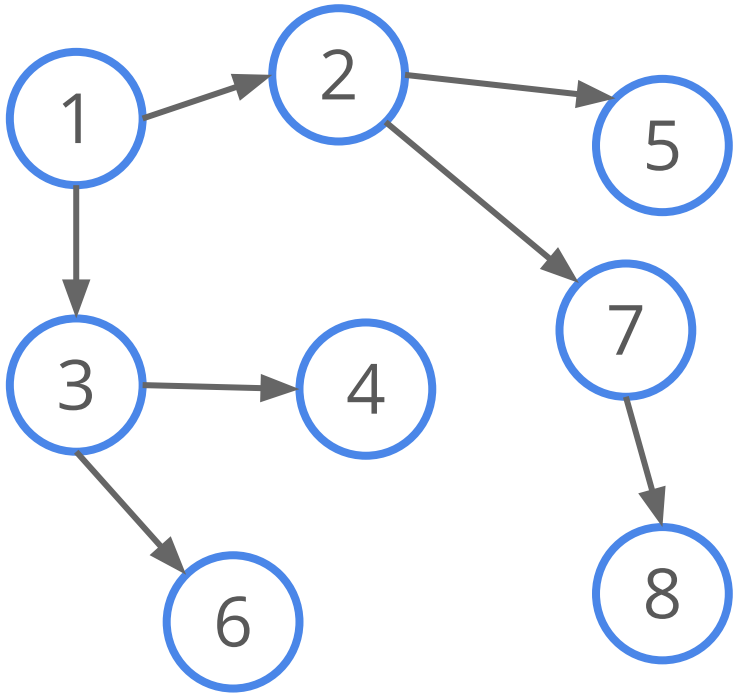
- **Trees are Hierarchical:** Represent relationships in a parent-child hierarchy (organizations, categories, etc.).
- **Recursive Nature:** Trees are naturally recursive (a tree is a root and subtrees), which fits well with DP recursion.

- Think Tree DP when:
  - Your problem is defined on a tree or tree-like structure.
  - You can break down the problem into subproblems on subtrees.
  - Optimal solutions for subtrees can be combined to find the optimal solution for the larger tree.

Tree Basics - Things you should know

- A special case of DAG (if it is rooted)
- It has N nodes and N-1 edges
- Exists an unique simple path from a node to any other nodes.

Tree rooted at node 1



Tree Basics - Things you should know

- **Root:** Top node, the starting point of the hierarchy.
- **Leaf:** Nodes at the bottom, with no children.
- **Parent, Child:** Nodes directly connected in the hierarchy.
- **Ancestor, Descendant**: Have path connecting up or down the tree
- **Height, Depth:** Vertical levels in the tree. (With respect to leaf or to root)
- **Subtree:** A node and all its descendants.

Root = 1

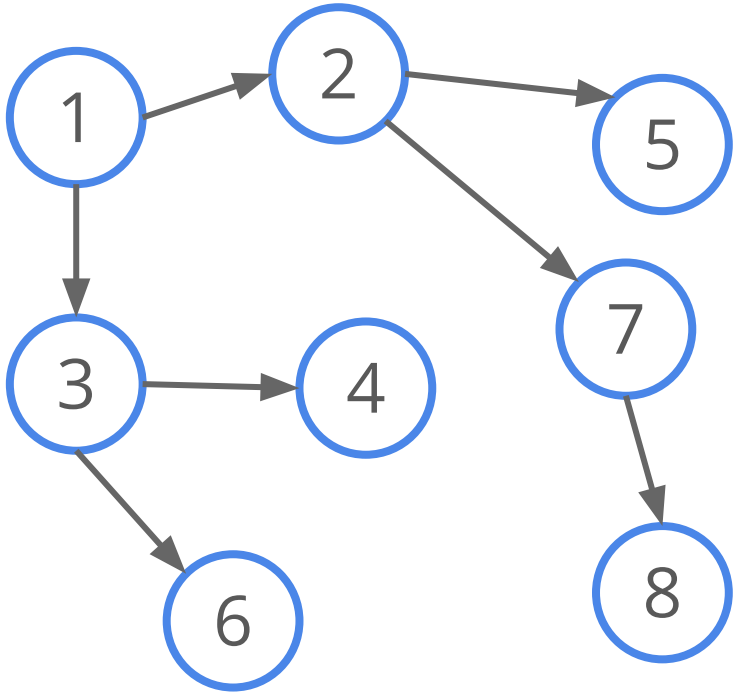
Leaf = 4, 5, 6, 8

2 is parent of 7
7 is child of 2

2 is ancestor of 8
8 is descendant of 2

Height of 2 = 3
Depth of 2 = 2

Subtree of 2
contains 2, 5, 7, 8



Tree DP Approach - Recursive Subtree Solutions

- Sometimes, the given tree is not rooted. You have to choose a root (according to task info or randomly pick one) to define parent-child relationships.

- **DP states:** dp[node] = optimal solution for the subtree rooted at 'node'.
- **Recursive Transitions:** Calculate dp[node] using dp values of its children.
- **Base Cases:** Define DP values for leaf nodes (simplest subtrees).
- **DFS Traversal:** Use Depth First Search to traverse the tree and calculate DP values in a bottom-up manner (from leaves to root).

Tree DP Example Task 0 - Subtree size

Problem: Given a rooted tree of size N. For each node in a rooted tree, calculate the size of its subtree (number of nodes in the subtree, including itself).

Naive Solution

- For each node, recursively count number of nodes in the subtree.
- Time complexity = O(N^2)
  - Can you think of a case that will hit this worst time complexity?

Tree DP Example Task 0 - Subtree size

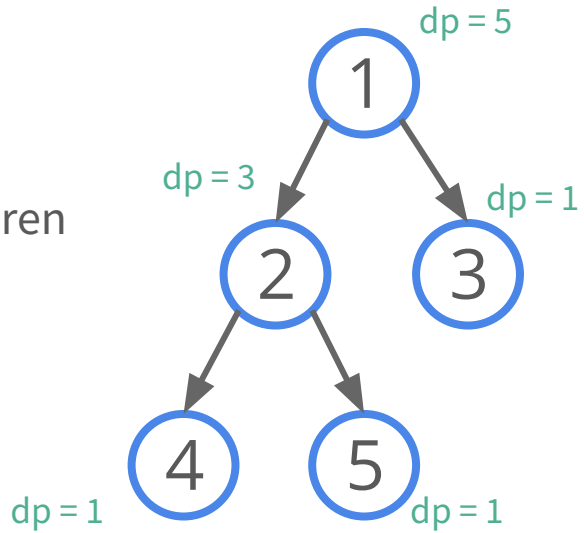
Problem: Given a rooted tree of size N. For each node in a rooted tree, calculate the size of its subtree (number of nodes in the subtree, including itself).

DP State: dp[i] = size of subtree i

Base Case: If node i is a leaf, dp[i] = 1.

Transition Formula: dp[i] = 1 + sum(dp[j]) where j is i's children

- Time complexity = $O(N)$
  - Each node is visited once only



Tree DP Example Task 1 - Subtree max

Problem: Given a rooted tree of size N. Each node i has a value v[i]. Find the maximum value in the subtree of each node.

- Similar settings with the previous task.

Tree DP Example Task 1 - Subtree max

Problem: Given a rooted tree of size N. Each node i has a value v[i]. Find the maximum value in the subtree of each node.

DP State: dp[i] = maximum value in subtree of node i

Base Case: If node i is a leaf, dp[i] = v[i].

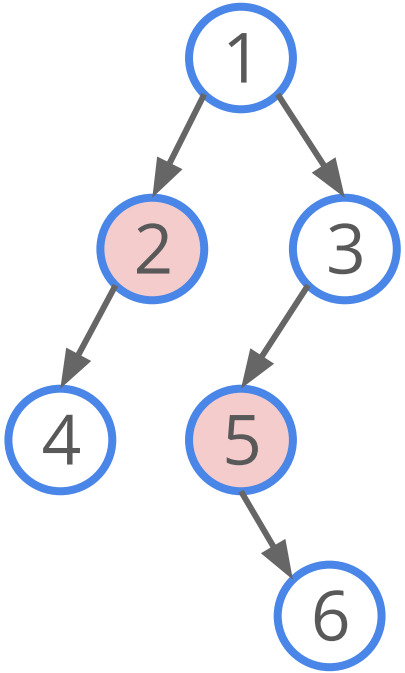
Transition Formula: dp[i] = max(v[i], max(dp[j]) where j is i's child)

- Time complexity = $O(N)$
  - Each node is visited once only

Tree DP Example Task 2 - Painter

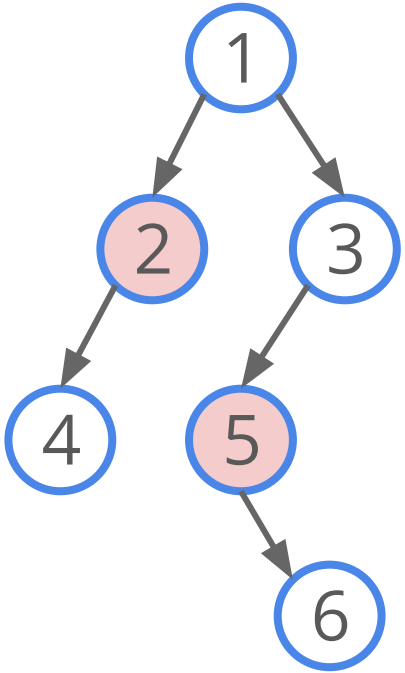
Problem: Given a tree, minimize the number of painters needed to paint all nodes. A painter at a node paints the node itself, its parent, and its immediate children.

How do we minimize painters while ensuring every node is painted?



Tree DP Example Task 2 - Painter

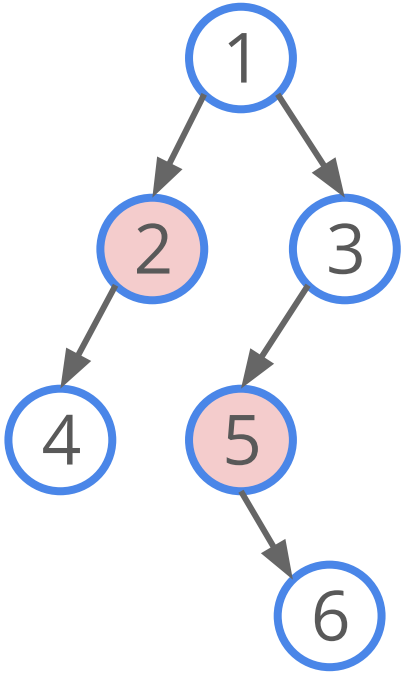
- One state per node is not enough. We also want to know some information about whether it is painted / covered.
- For each node i, we define 3 DP states to cover all possibilities:
  - **dp[i][0]**: Minimum painters in subtree of i if node i is painted by a painter at i.
  - **dp[i][1]**: Minimum painters in subtree of i if node i is painted by at least one of its children.
  - **dp[i][2]**: Minimum painters in subtree of i if node i **will be** painted by its parent.
- We need to ensure all nodes will be covered and we can build up the solution from children to parent.



Tree DP Example Task 2 - Painter

- Transition for **dp[i][0]:**
  - We place a painter at i.
  - Children can be in any state. Minimize painters in subtrees independently.

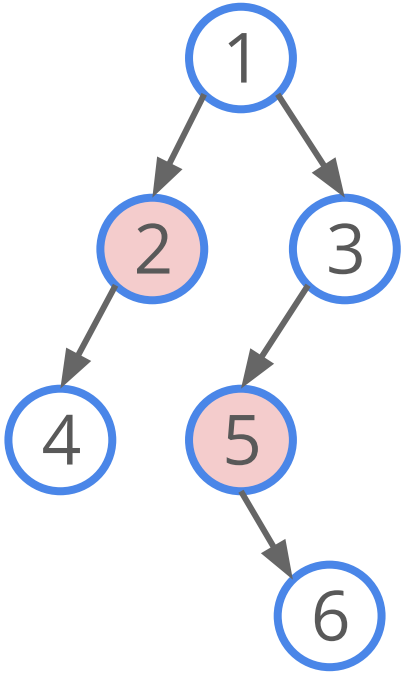
- `dp[i][0] = 1 + sum(`
  `min(dp[j][0], dp[j][1], dp[j][2])`
  ), where j is i's child



Tree DP Example Task 2 - Painter

- Transition for **dp[i][1]:**
  - Node i is painted by one of its children or itself.
  - Since it is not guaranteed that it has a painter. At least one children need to be in state 0.
  - Children cannot rely on node i, so state 2 cannot transition.

- Consider fixing child j to be state 0:
- `dp[i][1] = min(`
  `dp[j][0] + sum( min(dp[k][0], dp[k][1]) )`
  where k is i's child **except** j
) where j is i's child

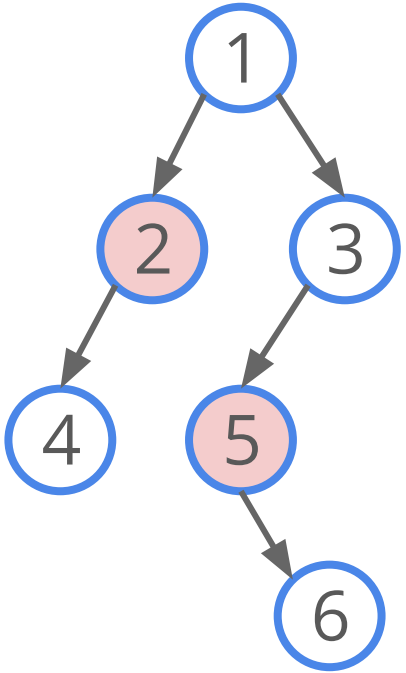


Tree DP Example Task 2 - Painter

`sum( min(dp[k][0], dp[k][1]) )` where k is i's child **except** j =
`sum( min(dp[k][0], dp[k][1]) ) - min(dp[j][0], dp[j][1])`
where k is **any of** i's child

- For each node i we may precompute `sum( min(dp[k][0], dp[k][1]) )` (denote this sum as `f(i)`) before computing the transition

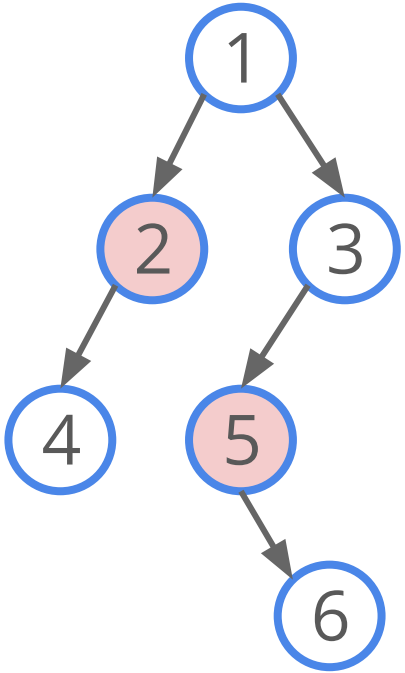
- Consider fixing child j to be state 0:
- `dp[i][1] = min(`
  `dp[j][0] + f(i) - min(dp[j][0], dp[j][1])`
) where j is i's child



Tree DP Example Task 2 - Painter

- Transition for **dp[i][2]:**
  - All children of node i must be covered already.
  - -> all children must be painted by their own children to cover themselves and not rely on i. So, children must not be in state 2.

- `dp[i][2] = 1 + sum(`
  `min(dp[j][0], dp[j][1])`
  ), where j is i's child

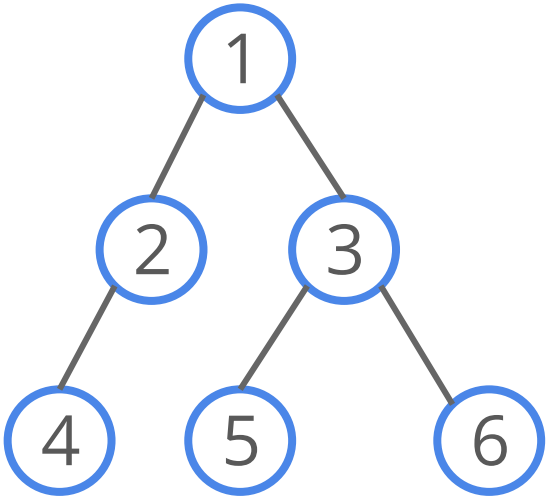


Tree DP Example Task 2 - Painter

- **Base case**(leaf):
  - dp[i][0] = 1 (Painter at leaf)
  - dp[i][1] = ∞ (Leaf has no children so i cannot be painted and this case is impossible, in reality we can set it as a huge number e.g. 10^9 or 10^{18})
  - dp[i][2] = 0 (Rely on its parent to be covered, 'deferring' the painting responsibility to its parent without painting itself)
- **Final Answer:** min(dp[root][0], dp[root][1])
- **Time Complexity:** O(N)

Tree DP Example Task 3 - Paths passing through

Problem: Given a tree of size N. Calculate number of simple paths passing through each node.



e.g. simple paths passing through node 3

$1 \leftrightarrow 5$, $1 \leftrightarrow 6$

$2 \leftrightarrow 5$, $2 \leftrightarrow 6$

$4 \leftrightarrow 5$, $4 \leftrightarrow 6$

$5 \leftrightarrow 6$

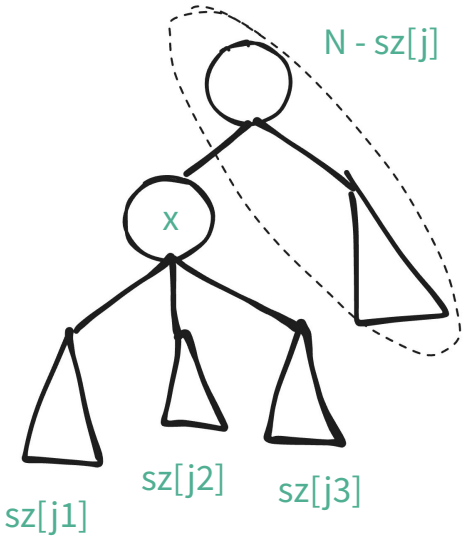
Total 7 paths

Tree DP Example Task 3 - Paths passing through

Problem: Given a tree of size N. Calculate number of simple paths passing through each node.

- A simple path u -> v that passes through some node x can be broken down into two subpaths u -> x and x -> v.
- All these subpaths must pass through either a child of x or the parent of x

- Answer for node x can be calculated with no. of paths that pass through all children j of node x and the no. of paths that pass through the parent of x
- Equal to **sz[j]** and **N - sz[x]**



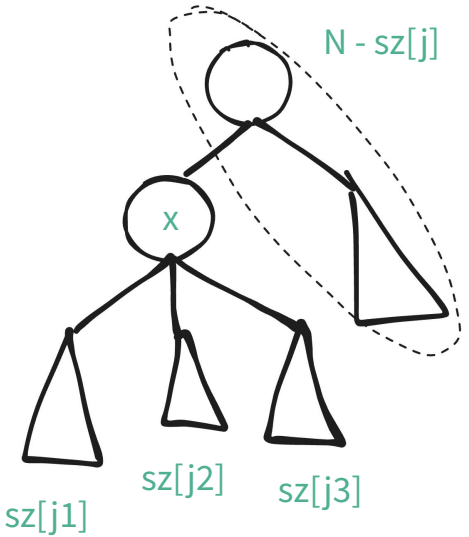
Tree DP Example Task 3 - Paths passing through

Problem: Given a tree of size N. Calculate number of simple paths passing through each node.

- Consider the nodes through parent of x as a "subtree"
- For each node x,

  ans += sz[this subtree] * sum(sz[all other subtrees])

  Final answer equals to **ans / 2**
- Time complexity = O(N)



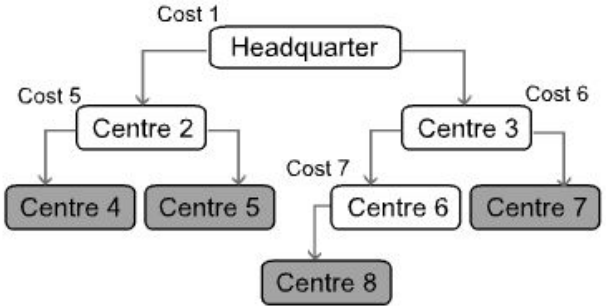
Tree DP Practice Task - HKOJ A301 Tree DP

- https://judge.hkoi.org/task/A301
- Let's spend some time to code this task together.

Problem: Given a tree of size N rooted at vertex 1. Each vertex i has an associated value A[i]. You may select any subset of vertices, but: if you select a vertex, you cannot select any of its children. Maximize the sum of value of selected vertices.

Tree DP More Practice Task

- T094 Medical Laboratories

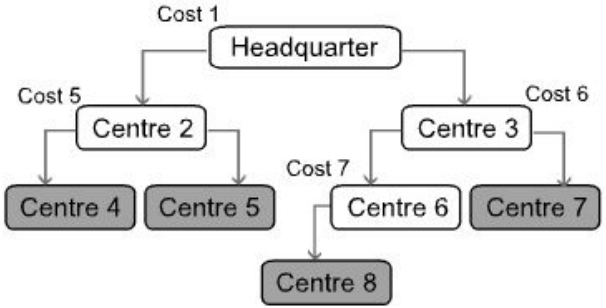


Tree DP More Practice Task

- T094 Medical Laboratories

Problem: Given a tree of size N rooted at vertex 1. Each vertex can have at most 2 children. Each vertex i has an associated weight W[i]. You need to choose K leaves to minimize the following cost:

sum(for pairs of leaf x, y chosen: W[LCA(x,y)])



e.g. 4, 5, 7 is chosen
LCA(4, 5) = 2, LCA(4, 7) = 1, LCA(5, 7) = 1
Cost = W[2] + W[1] + W[1]
Hint: dp[i][j] = cost of selecting j leaf in subtree i

Table of Contents

Bitwise DP

- Using **bitmask** as some states of the dp

- e.g. `dp[3][01101001]` (`dp[3][105]`)

- Bitmask: a sequence of bits, usually an integer written in binary notation
- Each bit can take on the value of 0 or 1, usually used to represented state of on / off or being chosen / not being chosen

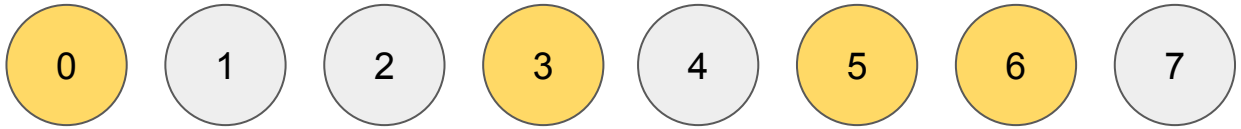
Why Bitwise DP?

- **Compact** for representing sets or combinations when the number of items is small.
- **Fast bitwise operations** for manipulating these compact states.
- Bitmasks allow you to represent complex set-based states using simple integers, reducing memory and potentially runtime.

- Think Bitwise DP when:
  - You need to track subsets or combinations of items.
  - The number of items is small (N <= 20).

Bitwise DP State - Representing Sets with Bitmasks

- Assume the followings are light bulbs. We can represent any combination of on/off states using an 8-bit bitmask.



- Treating the lit bulbs as 1, unlit bulbs as 0, this state can be represented by bitmask $\mathbf{01101001}_2 = 2^0 + 2^3 + 2^5 + 2^6 = 105$

- We corresponds the i-th bit (counting from right to left) with the i-th light bulb. In this order, the bitmask can be calculated by $\mathbf{\Sigma 2^i}$ for i-th bulb being lit

Bitwise DP State - Representing Sets with Bitmasks

Problem (0-1 Knapsack Problem): Given N items with weight w_i and value v_i, pick a subset of items such that their total weight <= K. Find the maximum total value of items picked.

- Like the previous example, the picked item can be represented by a bitmask (i-th item ↔ i-th bit), and can be fitted into a dp state.

- Although there exist better solution, coming up with state of bitmask dp is usually easy and can earn you some basic marks.
  - The transition is often trivial and not much observations is needed.

Bitwise DP - Bitwise Tricks

- Bit manipulation tricks are useful in bitwise dp

- Bitwise AND (`&`)
- Bitwise OR (`|`)
- Bitwise XOR (`^`)
- Bitwise SHIFT (`<<`, `>>`)
  - `x << y`: Shift x left by y bits
  - `5 << 4 = 80` ($5_{10} = 101_2$, $80_{10} = 1010000_2$)

Bitwise DP - Bitwise Tricks

- Test `j-th` bit on `i`
  ```
  if (i & (1 << j))
  ```
- Set `j-th` bit to 1
  ```
  i = i | (1 << j)
  ```
- Set `j-th` bit to 0
  ```
  i = i & ~(1 << j)
  ```
- Toggle `j-th` bit
  ```
  i = i ^ (1 << j)
  ```

Bitwise DP - Bitwise Tricks

- Get masks with `i` ones from the least significant bit
  `(1 << i) - 1`

- Is `i` a submask of `j`?
  `(i & j) == i`

- Enumerate non-empty submasks of `j` (from large to small)
  `for (int i = j; i > 0; i = (i - 1) & j)`

Bitwise DP Example Task - Switching Light bulbs

Problem: You have **N light bulbs** (N <= 15). You start with all light bulbs OFF.
There are **M buttons** (M <= 100) — button i toggles a specific set of light bulbs B_i0, B_i1, ..., B_ik respectively (on → off, off → on). Find the minimum button presses to turn all light bulbs at the same time, or output “impossible”.

Bitwise DP Example Task - Switching Light bulbs

Problem: You have **N light bulbs** (N <= 15). You start with all light bulbs OFF.
There are **M buttons** (M <= 100) — button i toggles a specific set of light bulbs B_i0, B_i1, ..., B_ik respectively (on → off, off → on). Find the minimum button presses to turn all light bulbs at the same time, or output “impossible”.

- State: dp[i][mask] = only considering button 1..i, minimum presses to reach state mask.
- Base Case: dp[0][0] = 0 (0 presses to start), other initialize as infinity.

Bitwise DP Example Task - Switching Light bulbs

Problem: You have **N light bulbs** (N <= 15). You start with all light bulbs OFF.
There are **M buttons** (M <= 100) — button i toggles a specific set of light bulbs B_i0, B_i1, ..., B_ik respectively (on → off, off → on). Find the minimum button presses to turn all light bulbs at the same time, or output “impossible”.

- State: dp[i][mask] = consider button 1..i, minimum presses to reach state mask.
- Transition Formula:
  - Suppose Bi is the set of light bulbs button i toggles, represented in bitmask form.
  - dp[i][mask] = min(…… , ……)

Hint: you either press or not press button i

Bitwise DP Example Task - Switching Light bulbs

Problem: You have **N light bulbs** (N <= 15). You start with all light bulbs OFF.
There are **M buttons** (M <= 100) — button i toggles a specific set of light bulbs B_i0, B_i1, ..., B_ik respectively (on → off, off → on). Find the minimum button presses to turn all light bulbs at the same time, or output “impossible”.

- State: dp[i][mask] = consider button 1..i, minimum presses to reach state mask.
- Transition Formula:
  - Suppose Bi is the set of light bulbs button i toggles, represented in bitmask form.
  - dp[i][mask] = min(dp[i-1][mask ^ Bi] + 1 , dp[i-1][mask])

Bitwise DP Example Task - Switching Light bulbs

Problem: You have **N light bulbs** (N <= 15). You start with all light bulbs OFF. There are **M buttons** (M <= 100) — button i toggles a specific set of light bulbs B_i0, B_i1, ..., B_ik respectively (on → off, off → on). Find the minimum button presses to turn all light bulbs at the same time, or output “impossible”.

- State: dp[i][mask] = consider button 1..i, minimum presses to reach state mask.
- Transition Formula: dp[i][mask] = min(dp[i-1][mask ^ Bi] + 1 , dp[i-1][mask])
- Optimal Substructure: Yes
- Calculation Order: dp[1][0], dp[1][1], ..., dp[1][2^N - 1], dp[2][0], dp[2][1], …, dp[2][2^N - 1], …, dp[M][0], dp[M][1], … **(WHY?)**
- Final Answer: dp[M][2^N - 1]

Bitwise DP Practice Task - HKOJ A302 Bitwise DP

- https://judge.hkoi.org/task/A302
- Let's spend some time to code this task together.

Bitwise DP Example Task 2 - M0712 Maximum Sum II

Problem: Given N x N grid of positive integers. Find the maximum sum of N numbers. No two numbers can be on the same row or the same column.

- $1 \le N \le 16$

SAMPLE TESTS

	Input	Output
1	3 1 1 10 2 5 10 1 10 3	22

Bitwise DP Example Task 2 - M0712 Maximum Sum II

Problem: Given N x N grid of positive integers. Find the maximum sum of N numbers. No two numbers can be on the same row or the same column.

- State: dp[i][bitmask] = the maximum sum of i numbers from the first i rows, by choosing columns represented by the bitmask.
  - Both dimension seems to be able to be a bitmask.
  - In here, only one as bitmask is enough.

SAMPLE TESTS

	Input	Output
1	3 1 1 10 2 5 10 1 10 3	22

Bitwise DP Example Task 2 - M0712 Maximum Sum II

Problem: Given N x N grid of positive integers. Find the maximum sum of N numbers. No two numbers can be on the same row or the same column.

- State: dp[i][bitmask] = the maximum sum of i numbers from the first i rows, by choosing columns represented by the bitmask.

- Transition:

```
for j in 1 .. N, if (bitmask & (1 << j) == 1)
    dp[i][bitmask] = max(
        dp[i][bitmask], dp[i - 1][bitmask ^ (1 << j)] + A[i][j]
    )
```

Bitwise DP Example Task 2 - M0712 Maximum Sum II

Problem: Given N x N grid of positive integers. Find the maximum sum of N numbers. No two numbers can be on the same row or the same column.

- State: dp[i][bitmask]
- Transition: dp[i][bitmask] = max(dp[i][bitmask], dp[i - 1][bitmask ^ (1 << j)] + A[i][j])

- Answer: dp[N][2^N - 1]
- Time complexity: O(N^2 * 2^N)

Bitwise DP Example Task 2 - M0712 Maximum Sum II

- State: dp[i][bitmask]
- It doesn't make sense for consider state where number of 1s in bitmask is not equal to i.

- We can precompute number of 1s in all bitmasks.
  - __builtin_popcount(bitmask)
  - A function that will return the number of 1 in a mask.
- Only consider state satisfying the above observation.
- Time complexity: O(N * 2^N)

Bitwise DP More Practice Tasks

- M1830 Lazy Tutor
  - Note that small constraints on N (N <= 20), items have binary state (on/off, pick/not pick) is a really large hint for bitwise DP.
  - This task is a bit tricky with the way of state design and transition.

- M2136 Guardian

Table of Contents

Memory Optimization

- Reducing Space Complexity
  - **Problem:** DP tables can consume significant memory, especially for large state spaces.
  - You may reach memory limit but not runtime limit, e.g. for N = 5000 with N^2 states.
  - **Techniques:** Rolling Arrays, State Compression, etc.
- Key Idea: If your DP transition only depends on the previous row (or a few previous rows), you don't need to store the entire DP table.

Memory Optimization Example Task - Grid Walk

Problem: Given N x M grid. Count number of paths from (1, 1) to (N, M) using only right and down moves.

- Naive DP
  - DP state: dp[i][j] = number of ways to move from (1,1) to (i, j)
  - Transition: dp[i][j] = dp[i-1][j] + dp[i][j-1]
  - Base state: dp[1][1] = 1
- Memory Complexity: O(NM)
- Observation: To calculate dp[i][*], we only need values from dp[i-1][*]. We don't need rows before i-1.

Memory Optimization - Rolling Array

Transition: dp[i][j] = dp[i-1][j] + dp[i][j-1]

- **Rolling Array** Implementation:
  - Use only two rows to store DP values.
  - Let dp[0][j] represent the values for the "previous" row and dp[1][j] for the "current" row.
  - **Iteration:** Iterate through rows i = 1 to N. In each row iteration:
    - Calculate dp[1][j] using dp[0][j] and dp[1][j-1].
    - **After each row, "roll" the arrays:** Swap dp[0] and dp[1] (or use modulo indexing). The "current" row becomes the "previous" row for the next iteration.
- Memory complexity = $O(M)$

Memory Optimization - Rolling Array

Transition: dp[i][j] = dp[i-1][j] + dp[i][j-1]

- In this particular task, keeping one row of dp states is also enough
- Use dp[1..M] and compute N times

More Practice Tasks

M0422 Christmas Tree
T153 Congressman Lee Sin
CF839C Journey
CF gym 103470H Crystalfly
Atcoder abc246G Game on Tree 3

NP1722 寶藏
CSES Elevator Rides
CF1285D Dr. Evil Underscores
CF1391D 505
CF1103D Professional layer

I0011 Palindrome
T003 Scheduling Lectures
I0721 Miners

Additional Readings

- SOS Dynamic Programming [Tutorial] - Codeforces
- [Tutorial] Non-trivial DP Tricks and Techniques - Codeforces