Dynamic Programming (II)

Fuzen Ng {yfng}

2023-03-11

Why DP?

- DP is a very common technique in OI
- Some tasks may divide subtasks into different levels of DP
- Some subtasks could be done by DP even the full solution is not DP

How to DP?

- Solve subproblems
- Memorize and reuse the results of the subproblems

Table of Contents

Related Tasks on HKOJ for today

M1739 How to Run Fast

M1862 Little Patterns, Big Canvas

T094 Medical Laboratories

I1022 Traffic Congestion

M0712 Maximum Sum II

M2136 Guardian

M1830 Lazy Tutor

If you are too strong

Dreaming (IOI 2013)

Alice and Her Lost Cat (Codeforces gym 104053 A)

Training (IOI 2007, you may view/solve the problem on oj.uz)

Things you should know

- Bases cases

  Subproblems that cannot be reduced
- States

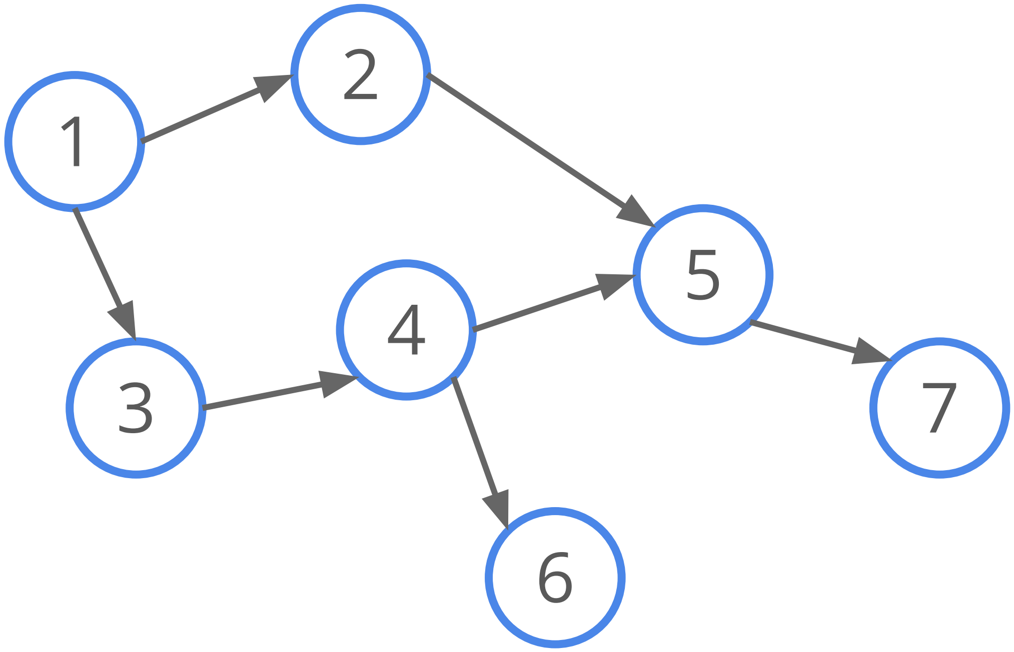
  IDs of subproblems
- Transition formula

  Using results of subproblems to find the answer

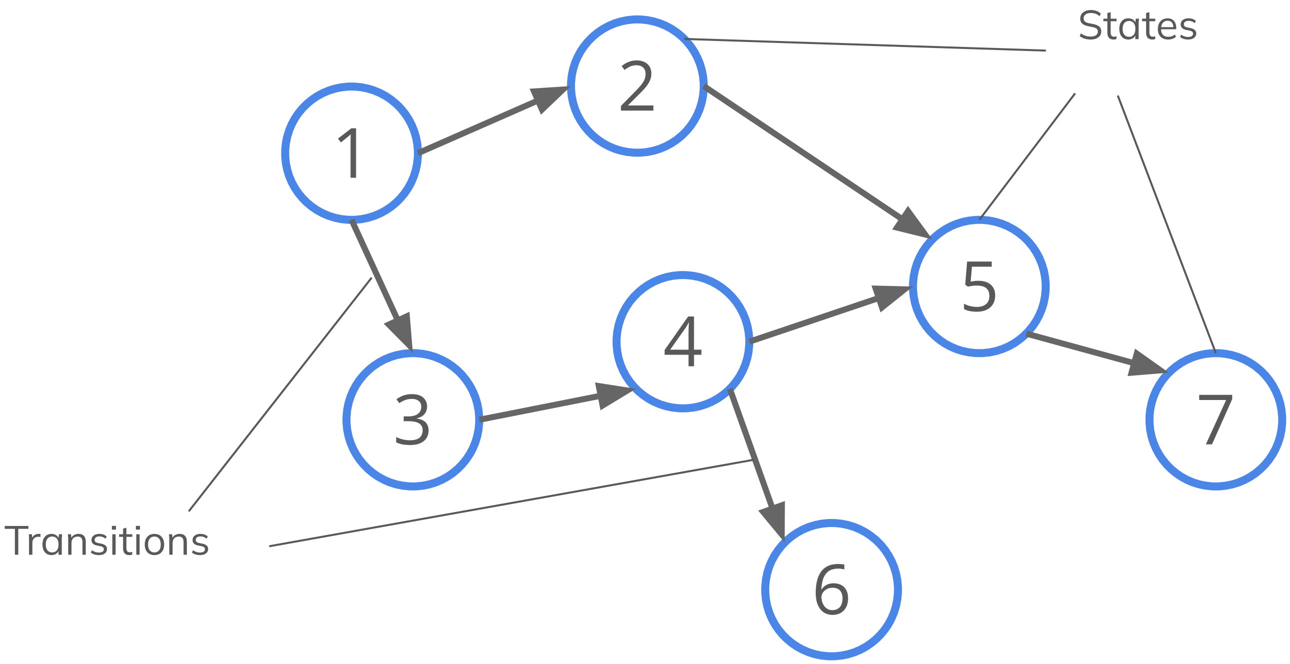
DAG

- **D**irected **A**cyclic **G**raph
- Node = State
- Edge = Transition
- Can be used as a tool to visualize DP transitions

DAG

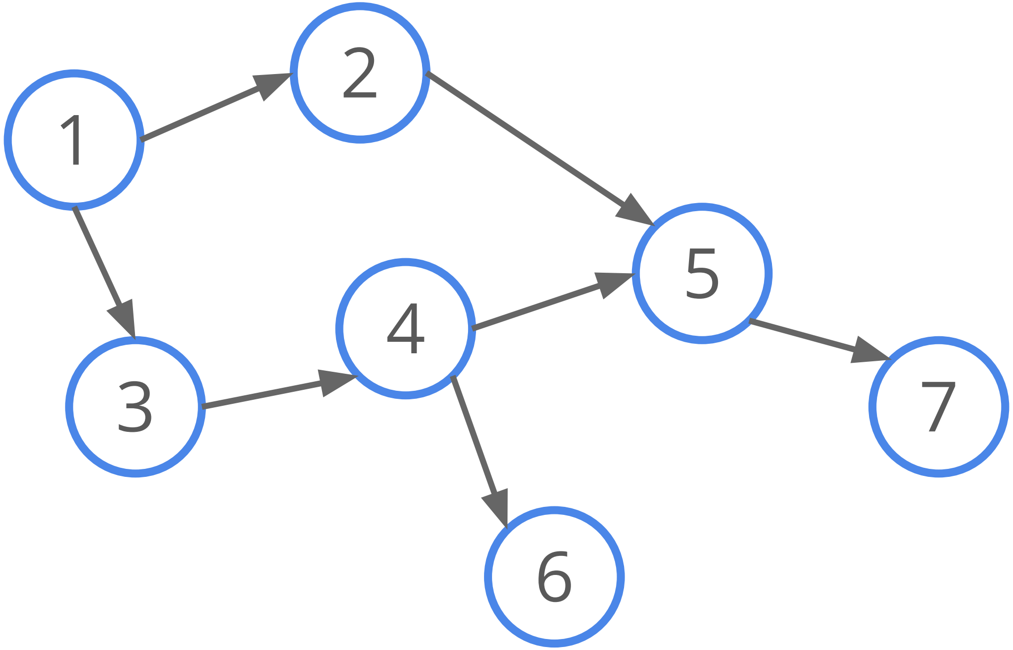


DAG



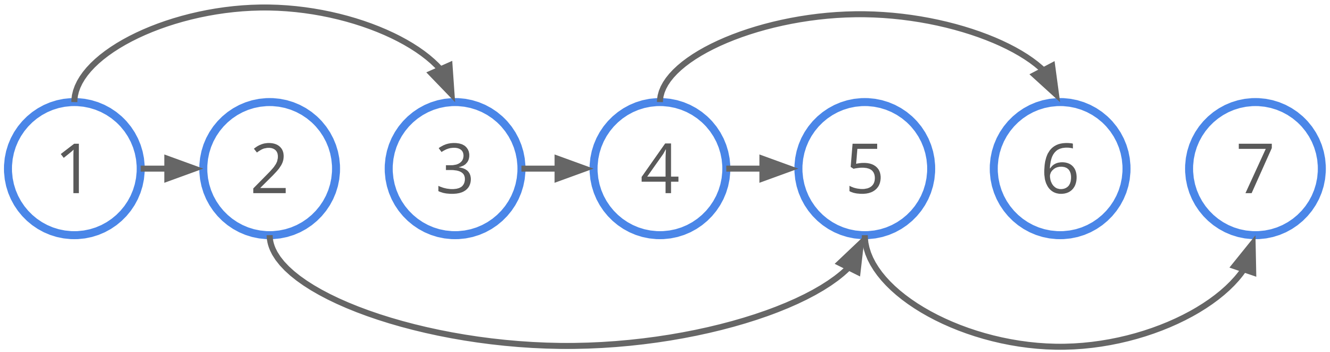
DAG

- **DAG** is usually applied with topological sort to determine the DP order



DAG

- **DAG** is usually applied with topological sort to determine the DP order



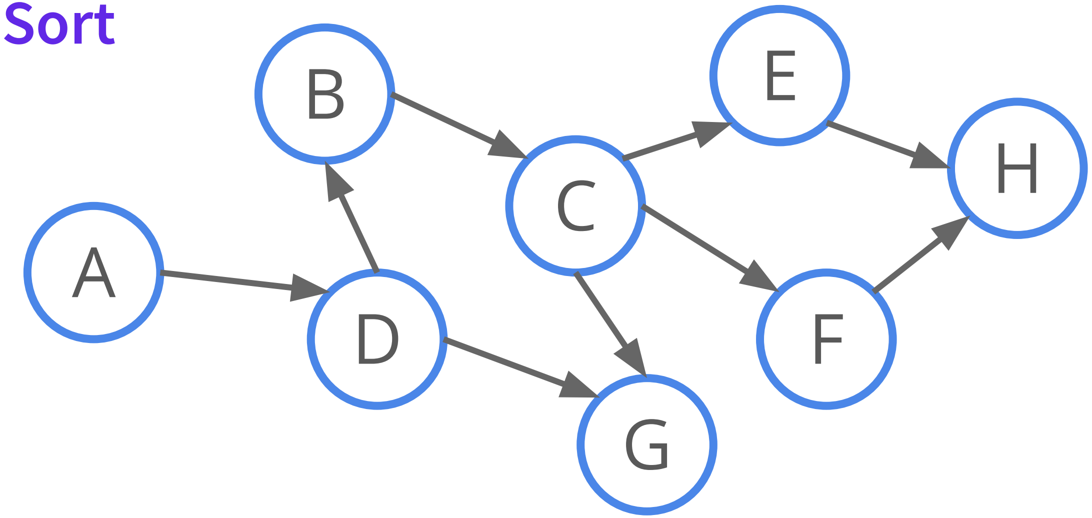
DAG - Topological Sort

- Obtain an order of nodes so that if there exist a directed edge from node A to node B ⇒ node A appears before node B in the order

```
while some nodes are unvisited
    choose any unvisited node
    DFS from the node
        push the node into a stack
        recur to visit an unvisited node
        pop from the stack when there are no more unvisited nodes
            insert the node into the topological order in reverse order
```

DAG - Topological Sort

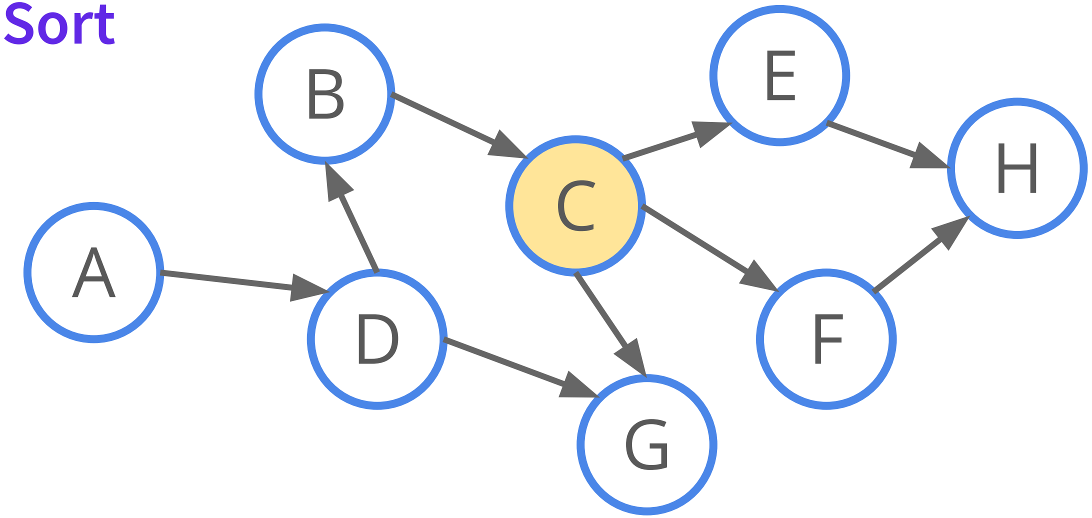
stack



Topological Order							

DAG - Topological Sort

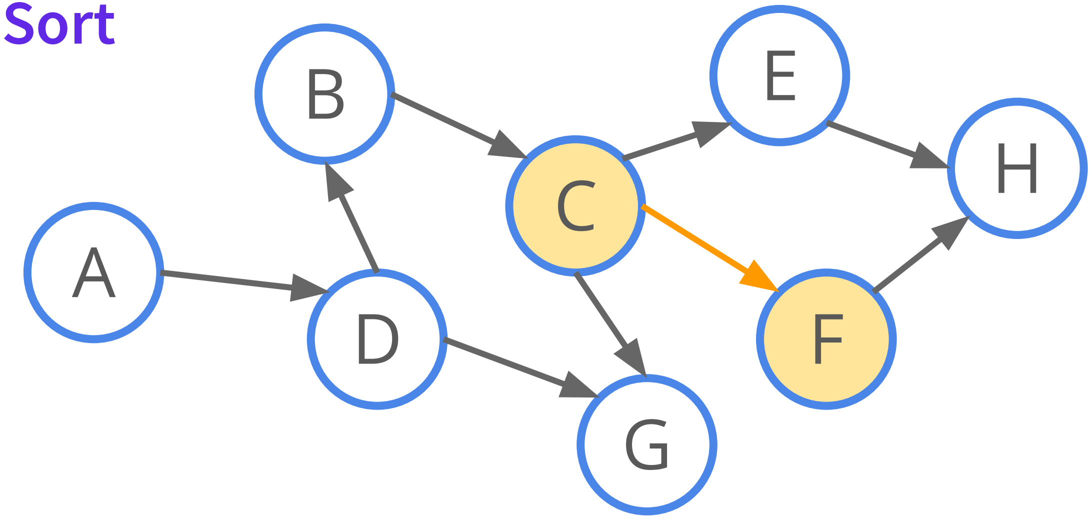
stack
C



Topological Order							

DAG - Topological Sort

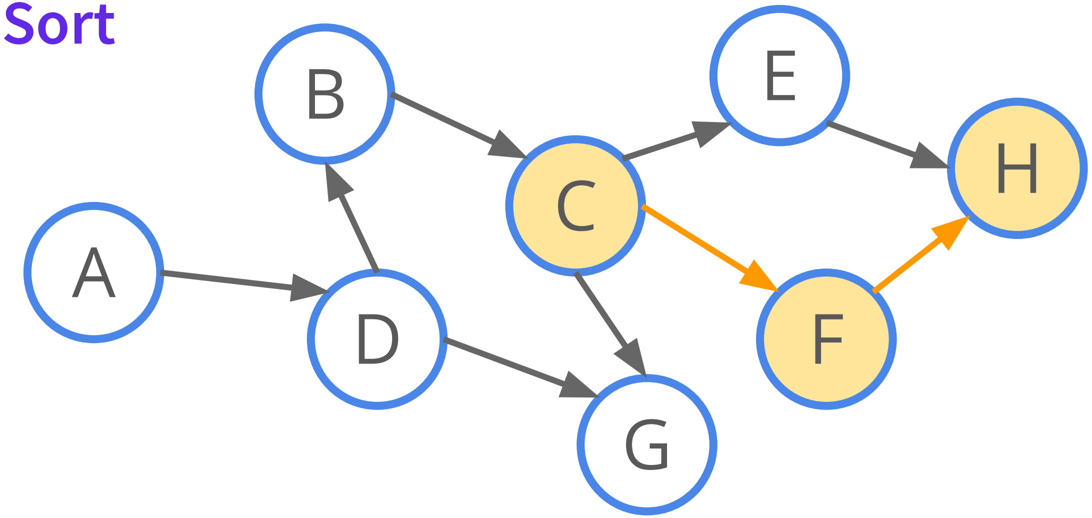
stack
C
F



Topological Order							

DAG - Topological Sort

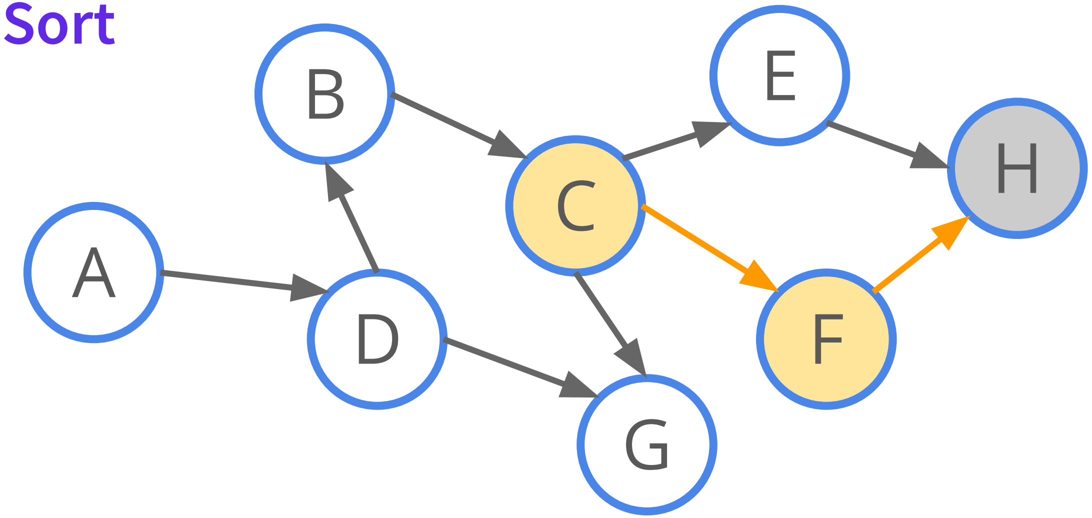
stack
C
F
H



Topological Order							

DAG - Topological Sort

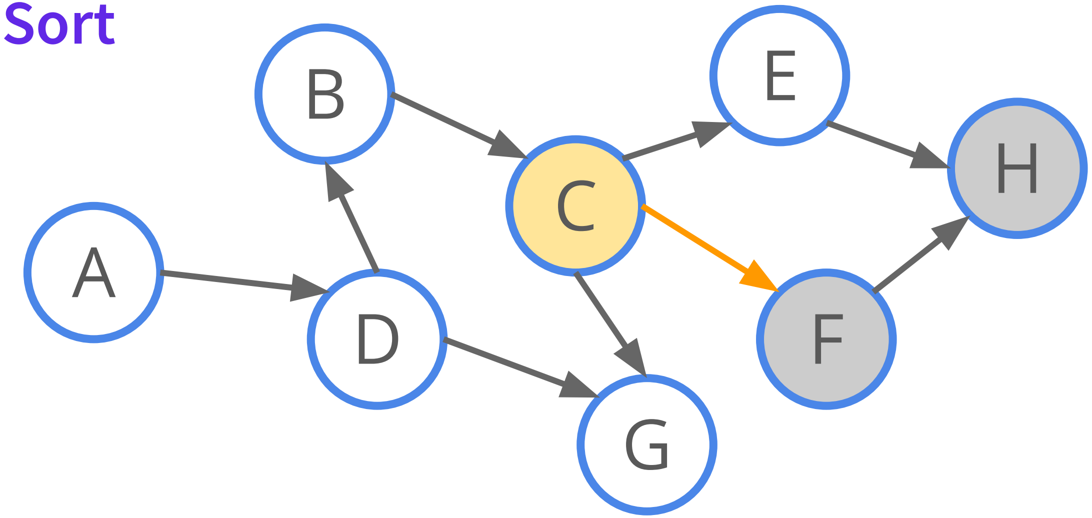
stack
C
F



Topological Order							
							H

DAG - Topological Sort

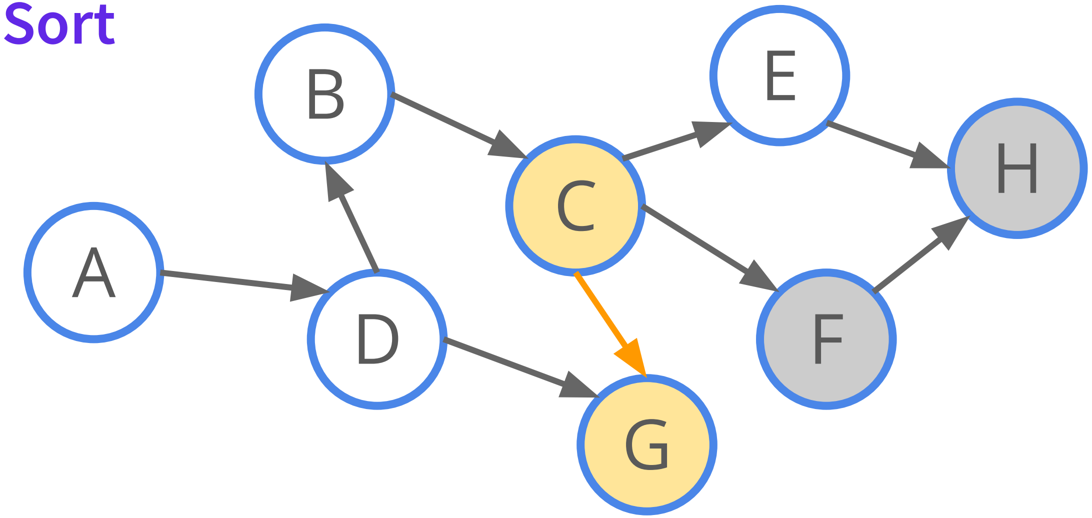
stack
C



Topological Order							
						F	H

DAG - Topological Sort

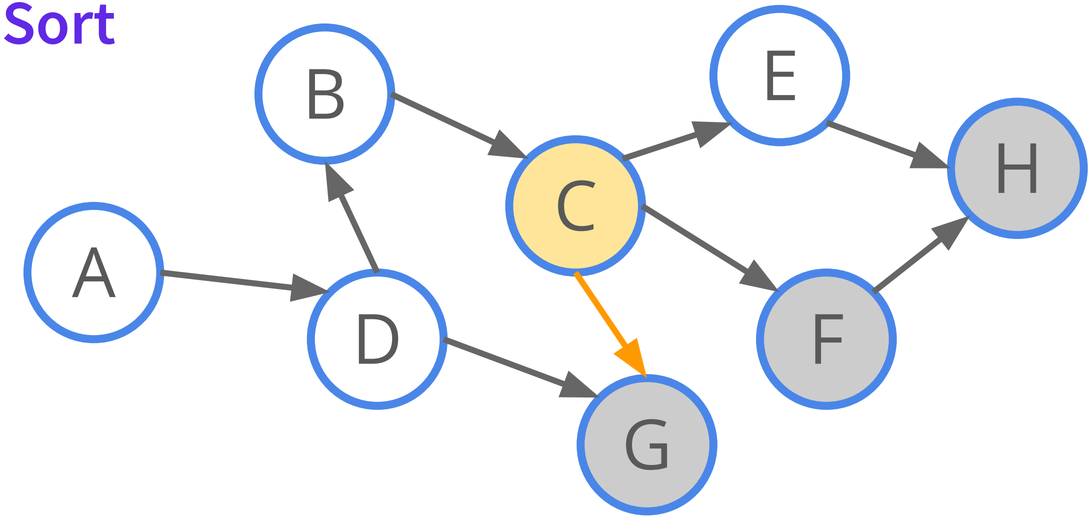
stack
C
G



Topological Order							
						F	H

DAG - Topological Sort

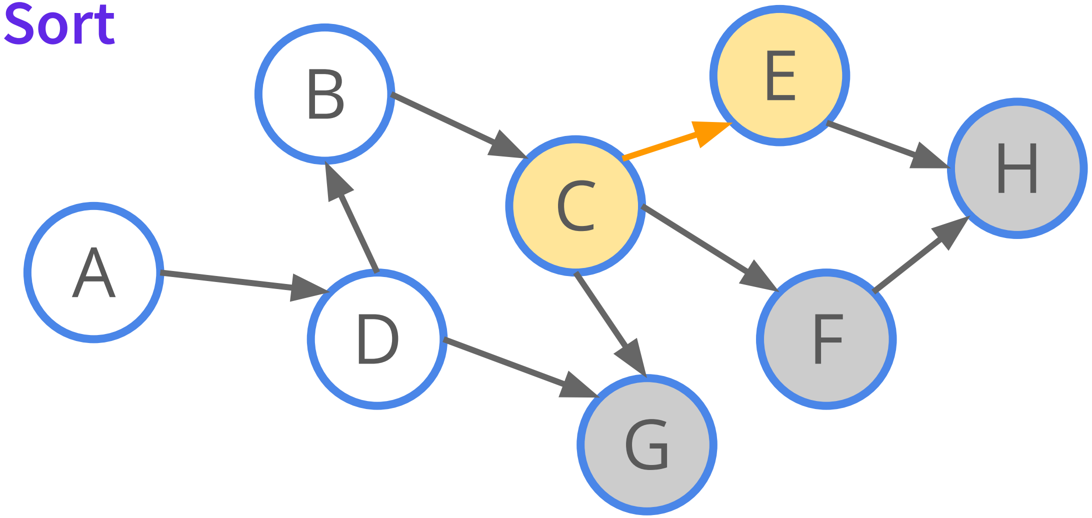
stack
C



Topological Order							
					G	F	H

DAG - Topological Sort

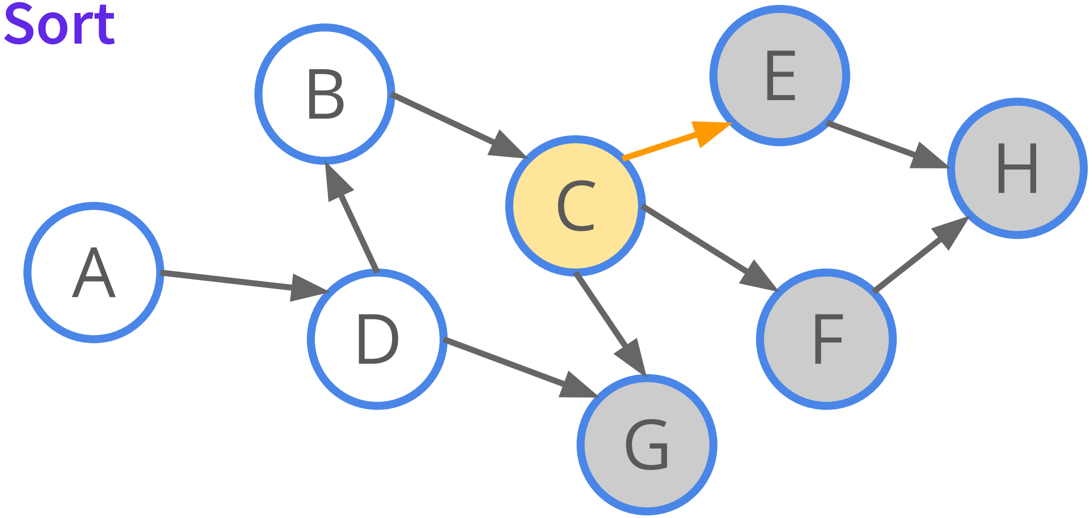
stack
C
E



Topological Order							
					G	F	H

DAG - Topological Sort

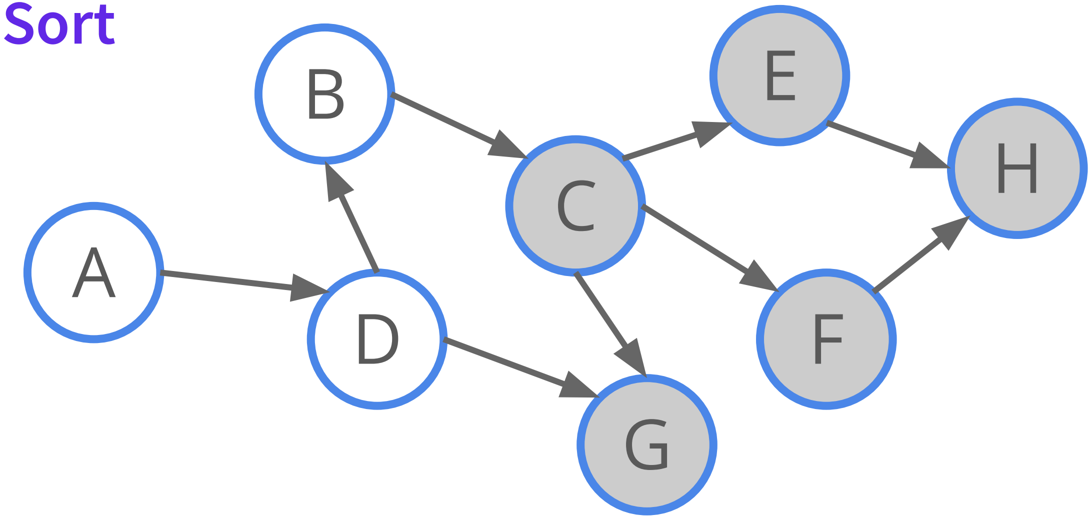
stack
C



Topological Order							
				E	G	F	H

DAG - Topological Sort

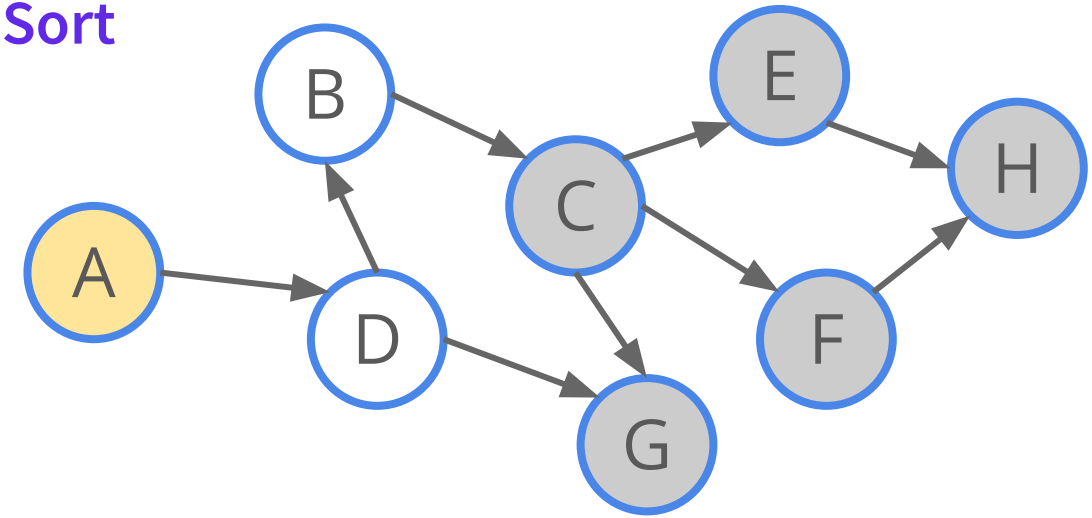
stack



Topological Order							
			C	E	G	F	H

DAG - Topological Sort

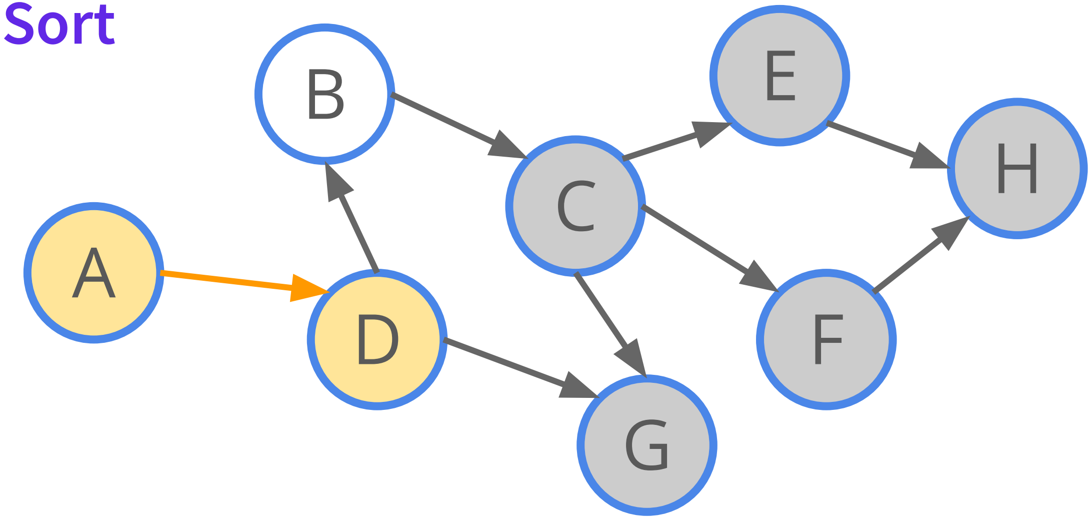
stack
A



Topological Order							
			C	E	G	F	H

DAG - Topological Sort

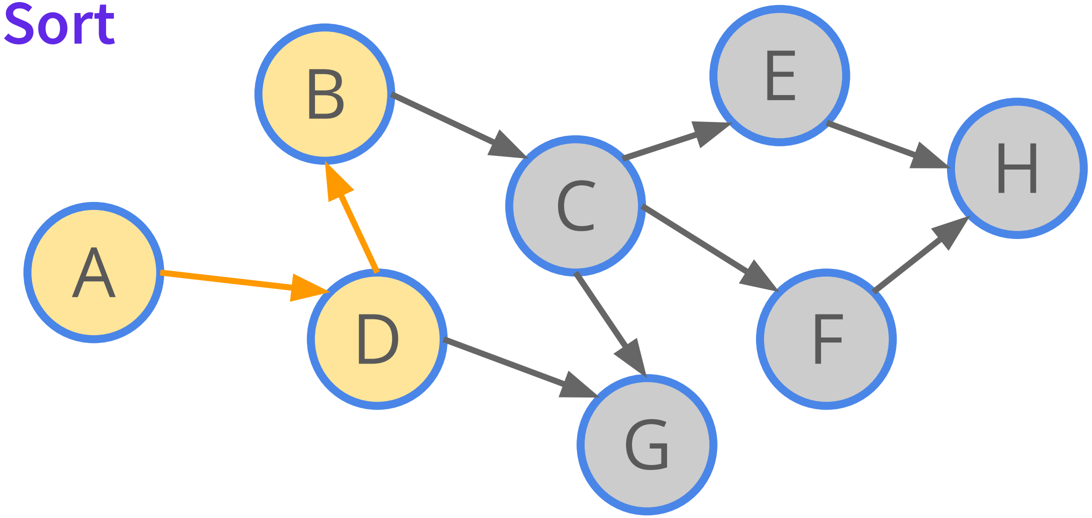
stack
A
D



Topological Order							
			C	E	G	F	H

DAG - Topological Sort

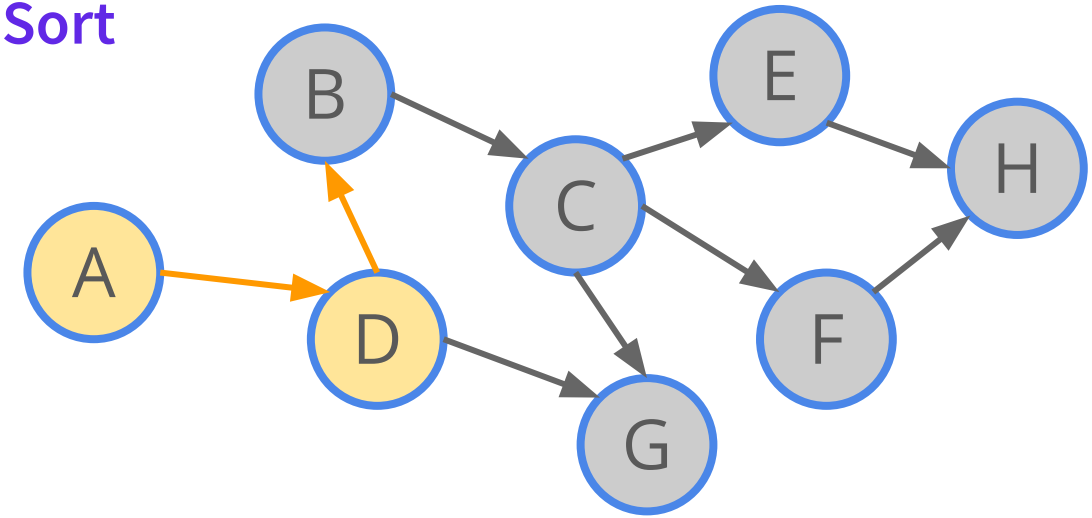
stack
A
D
B



Topological Order							
			C	E	G	F	H

DAG - Topological Sort

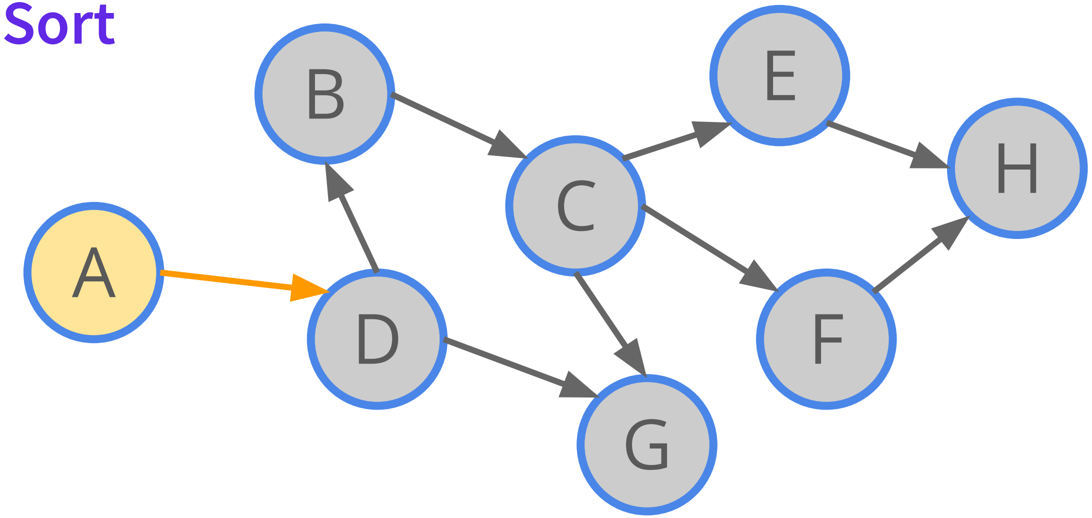
stack
A
D



Topological Order							
		B	C	E	G	F	H

DAG - Topological Sort

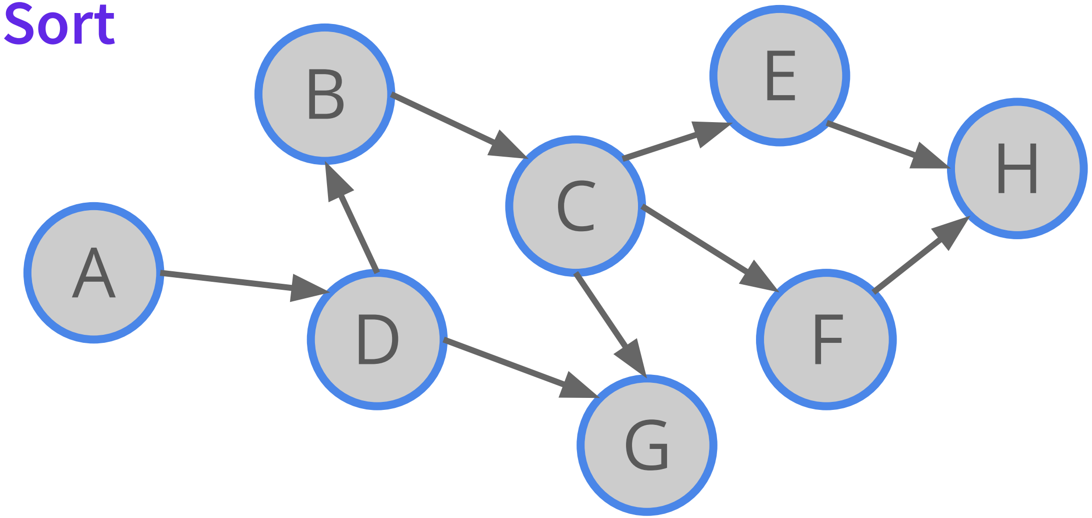
stack
A



Topological Order							
	D	B	C	E	G	F	H

DAG - Topological Sort

stack



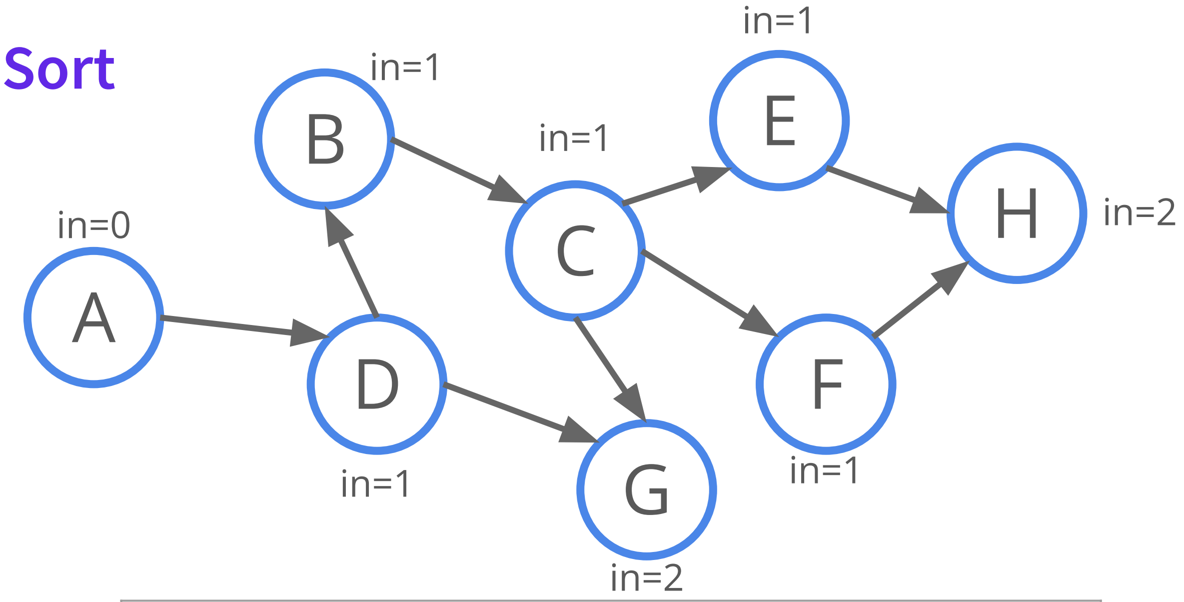
Topological Order							
A	D	B	C	E	G	F	H

DAG - Topological Sort

- Another way

```
maintain a set S of nodes with no incoming edge
while S is not empty
    pop any node u in S
    add u to the topological order
    for each edge from u to v
        delete the edge (one less incoming edge for v)
        if v has no incoming edges
            insert v to S
```

DAG - Topological Sort

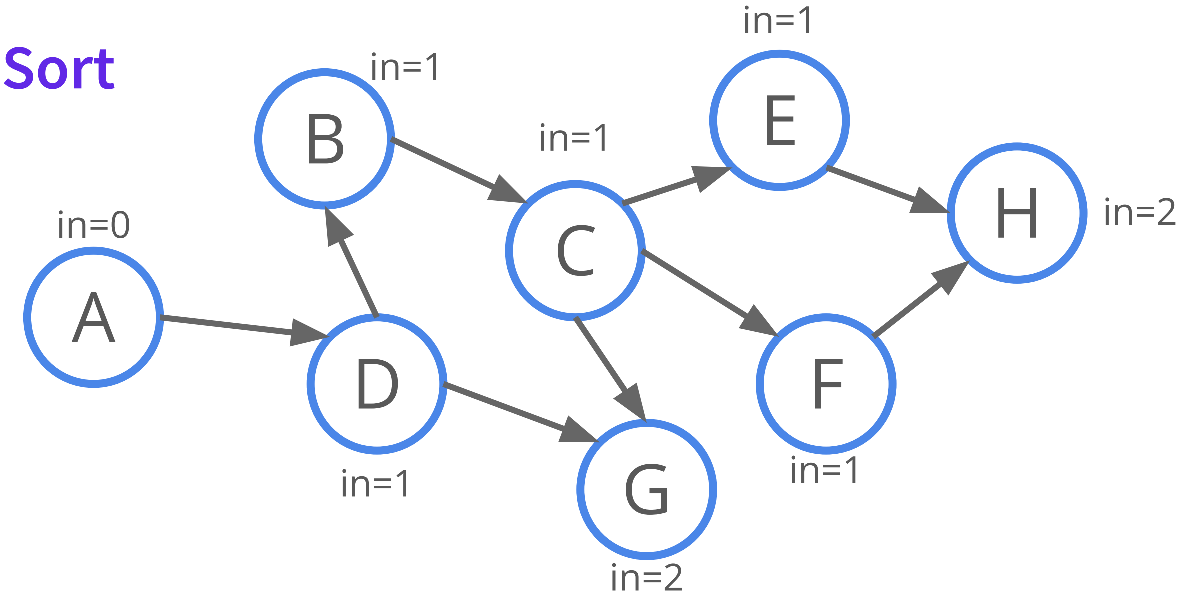


S:

Topological Order							

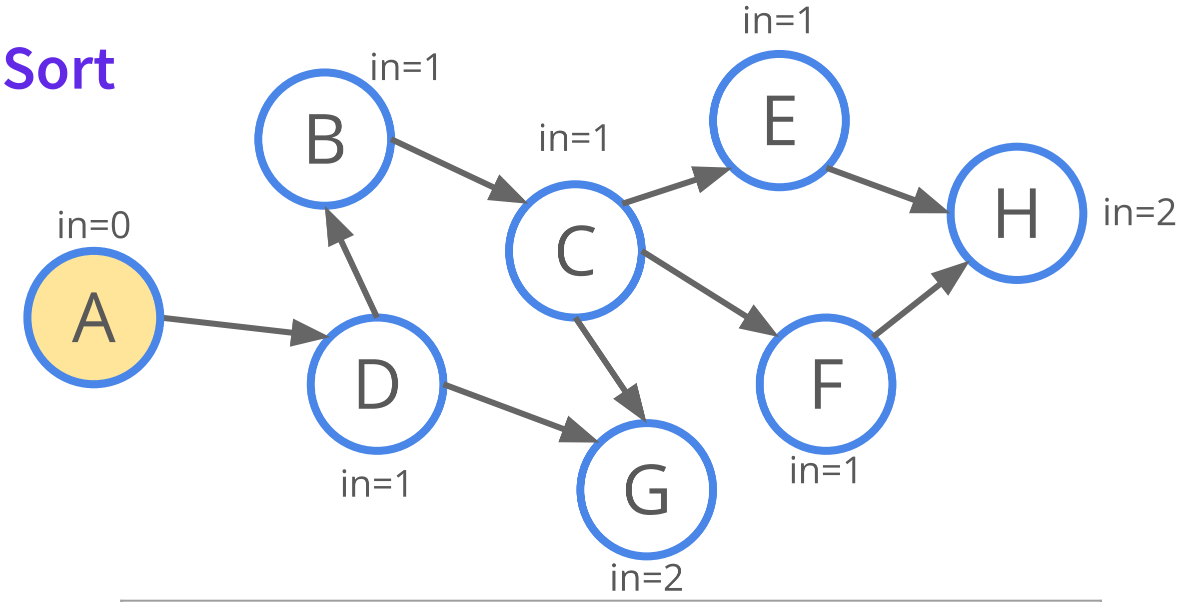
DAG - Topological Sort

S: A



Topological Order							

DAG - Topological Sort

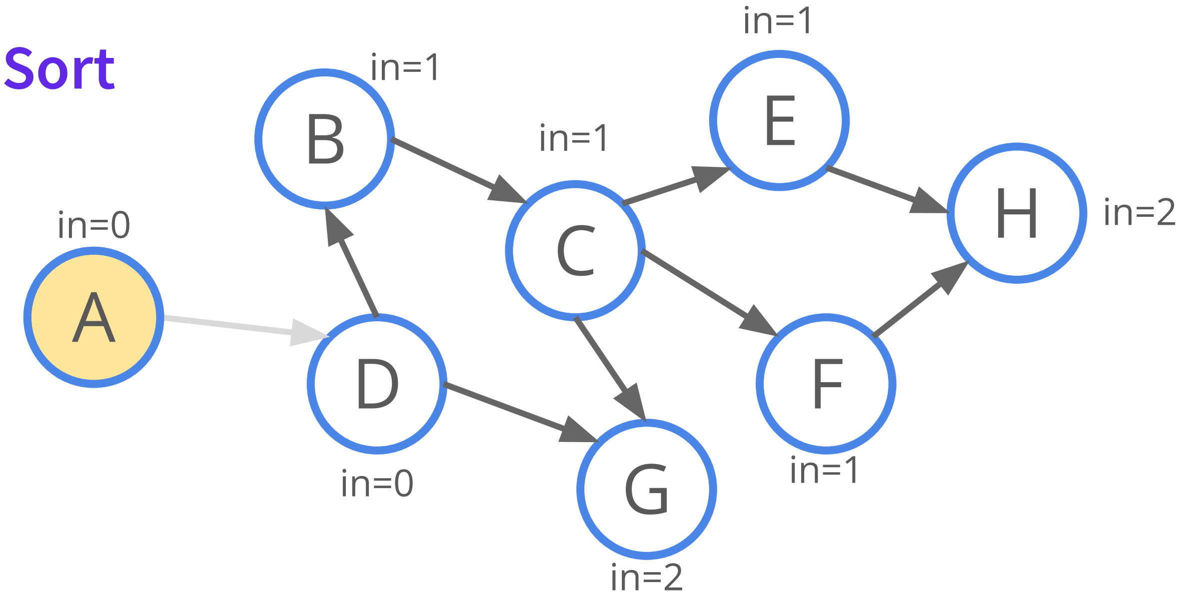


S:

Topological Order							
A							

DAG - Topological Sort

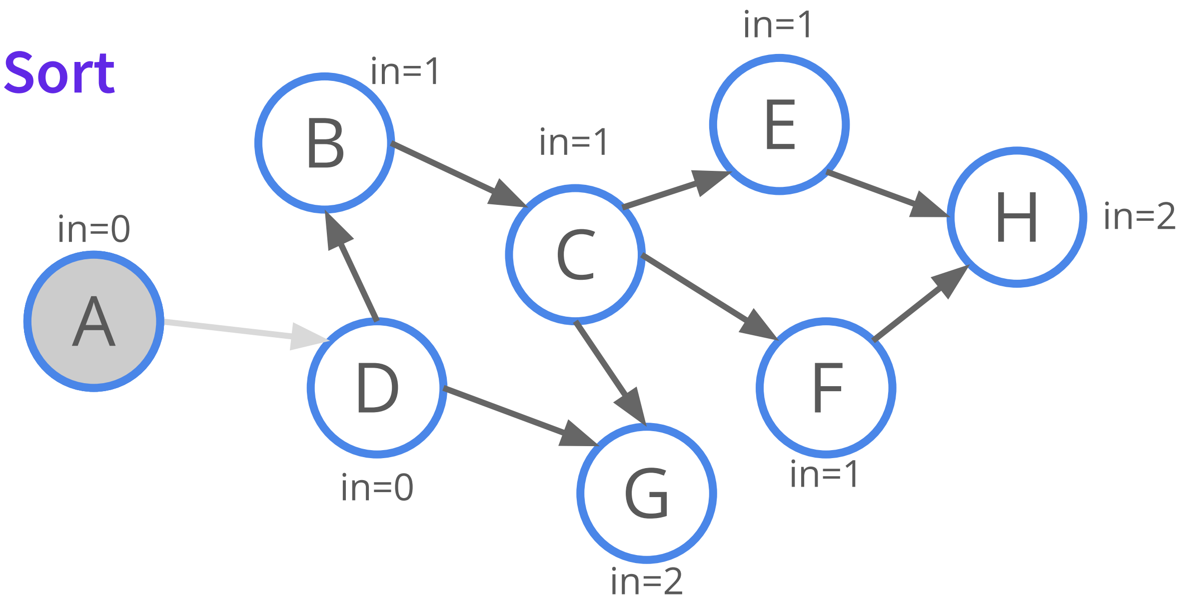
S:



Topological Order							
A							

DAG - Topological Sort

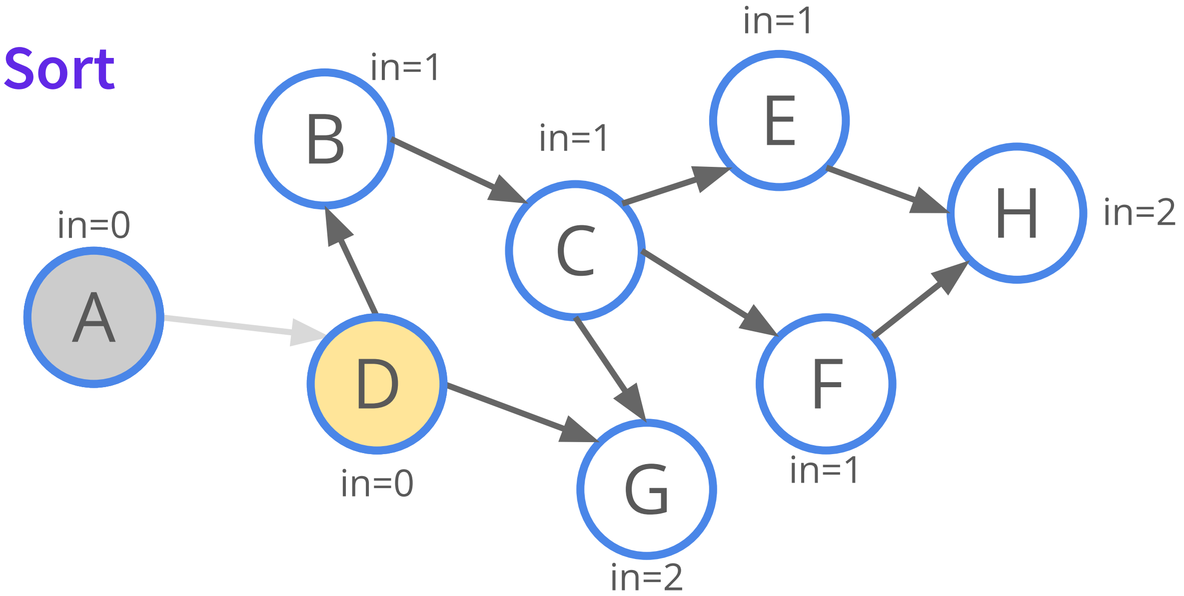
S: D



Topological Order							
A							

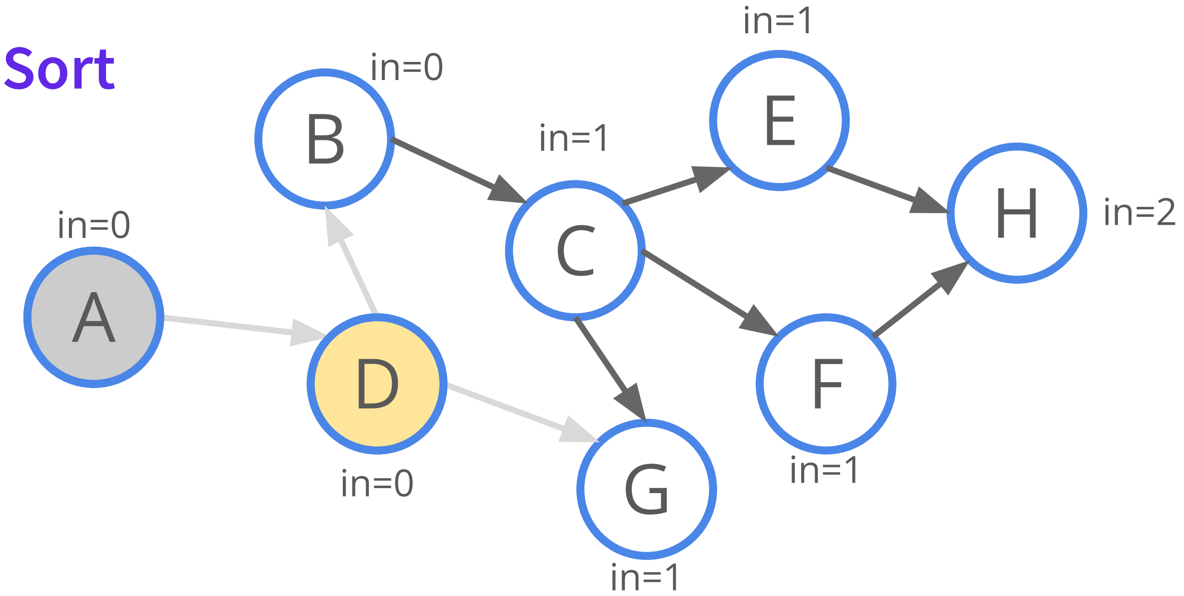
DAG - Topological Sort

S:



Topological Order							
A	D						

DAG - Topological Sort

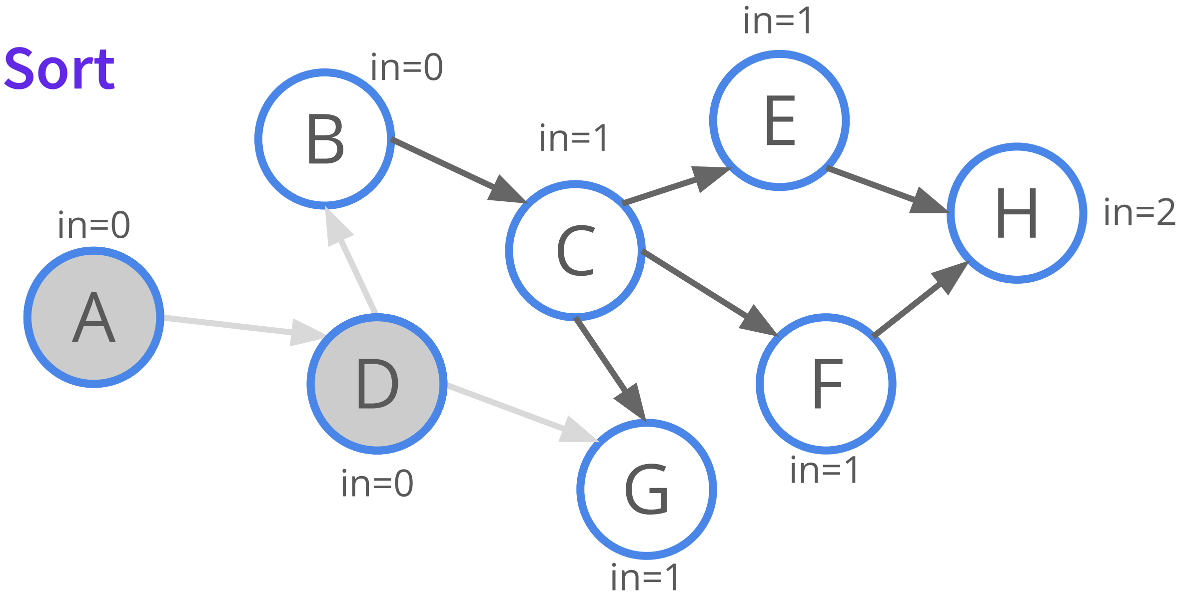


S:

Topological Order							
A	D						

DAG - Topological Sort

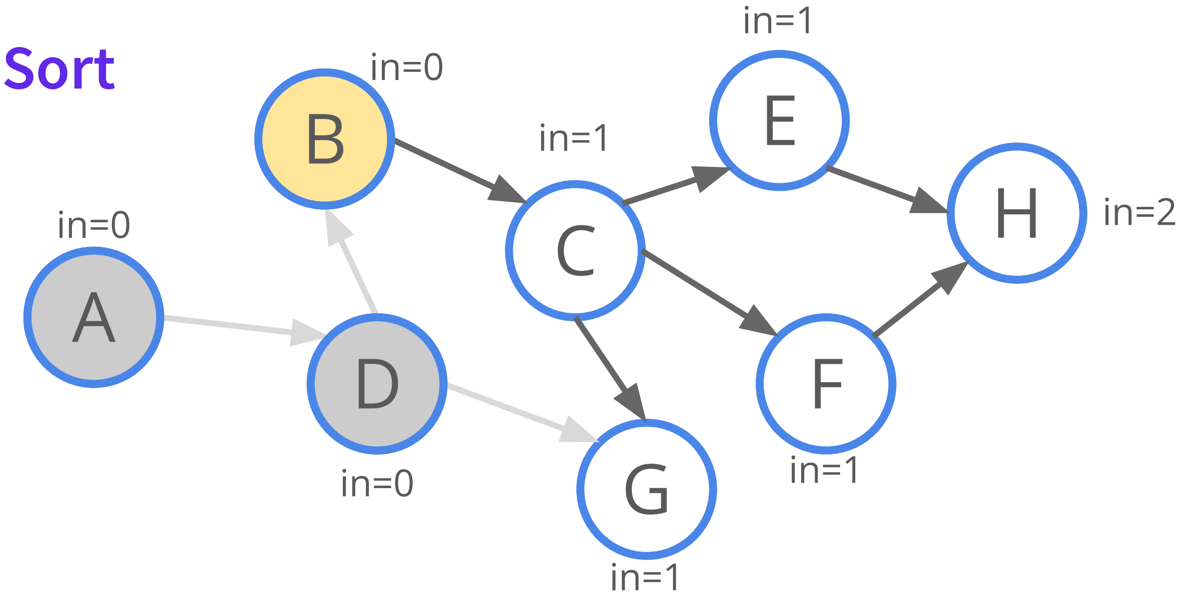
S: B



Topological Order							
A	D						

DAG - Topological Sort

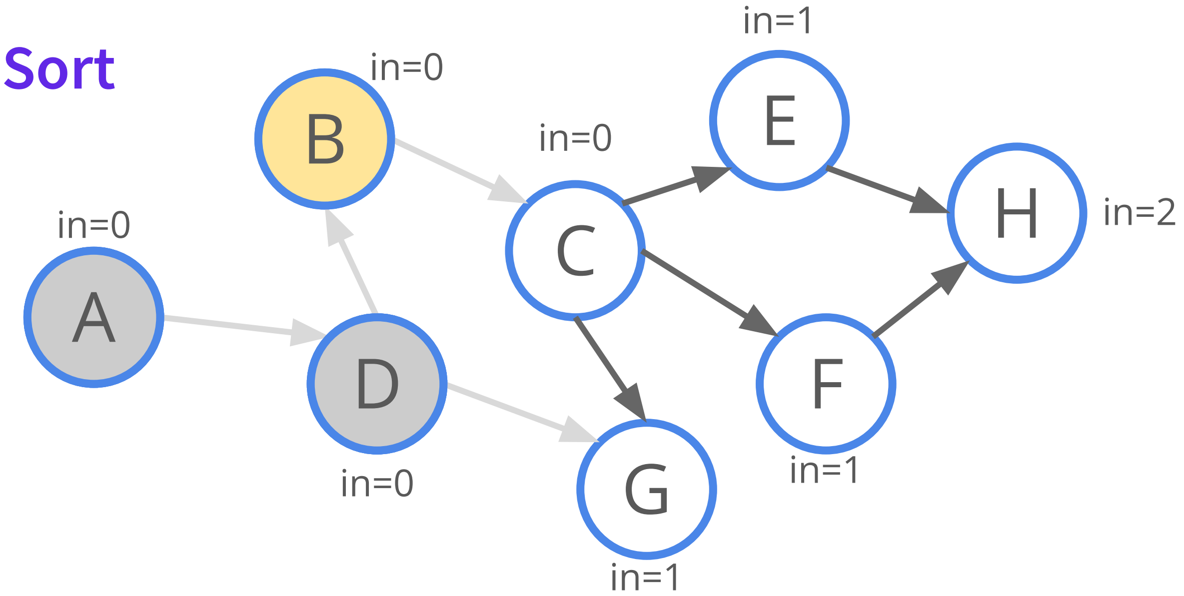
S:



Topological Order							
A	D	B					

DAG - Topological Sort

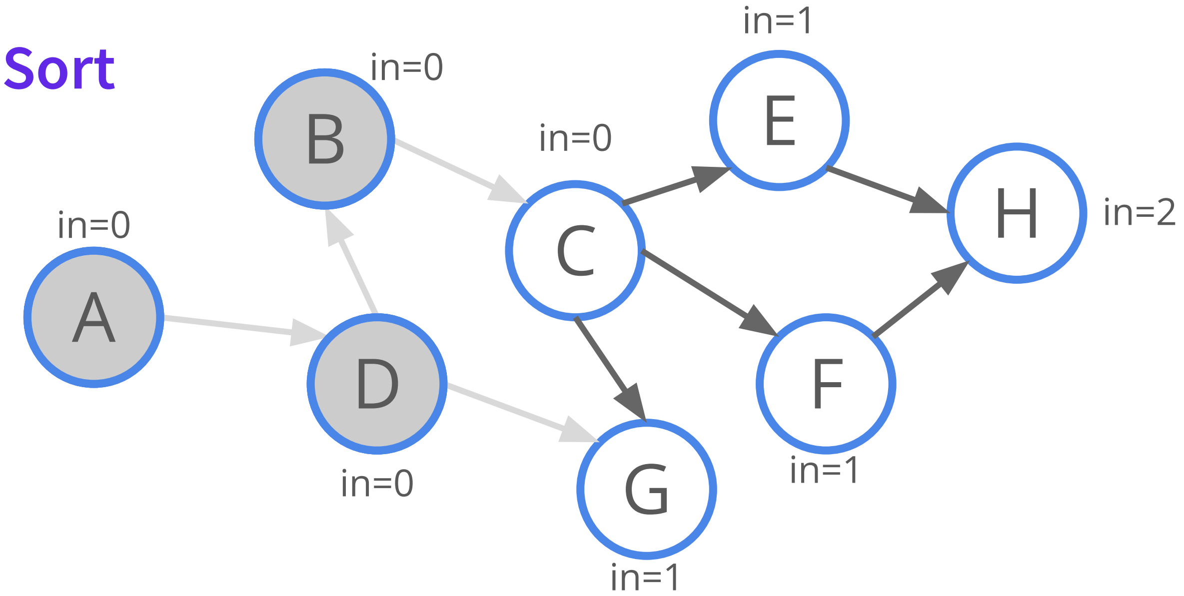
S:



Topological Order							
A	D	B					

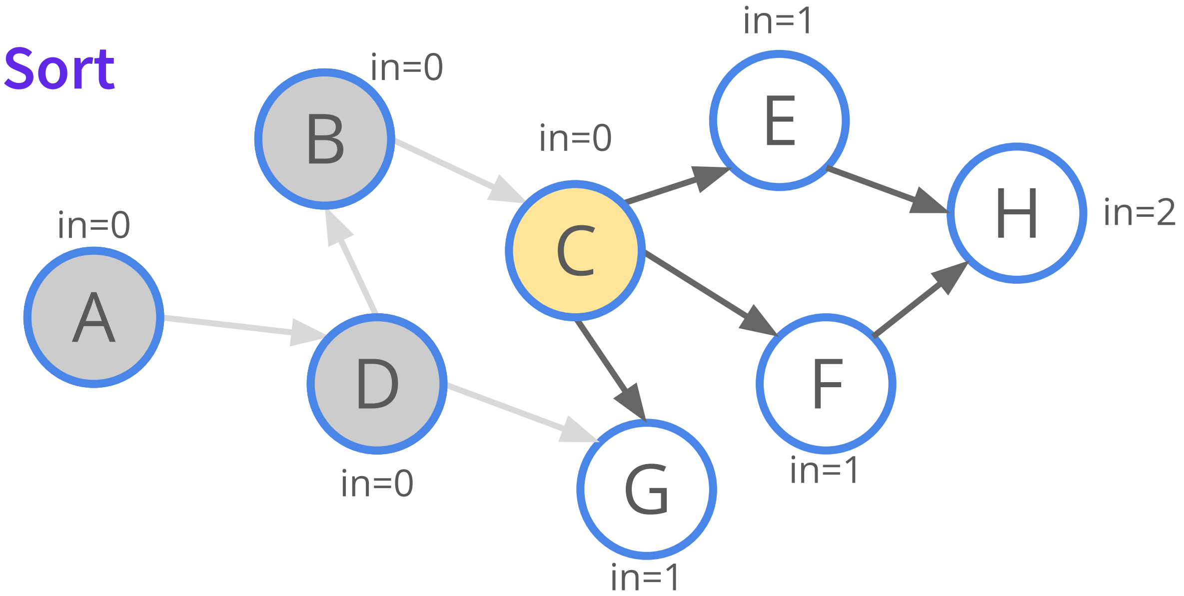
DAG - Topological Sort

S: C



Topological Order							
A	D	B					

DAG - Topological Sort

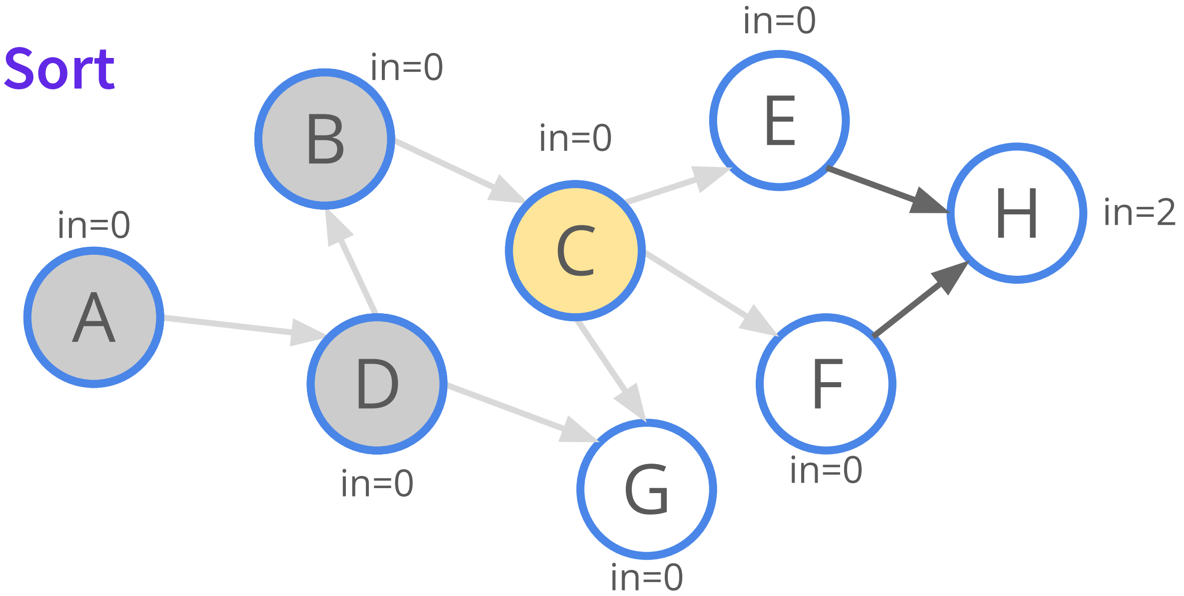


S:

Topological Order							
A	D	B	C				

DAG - Topological Sort

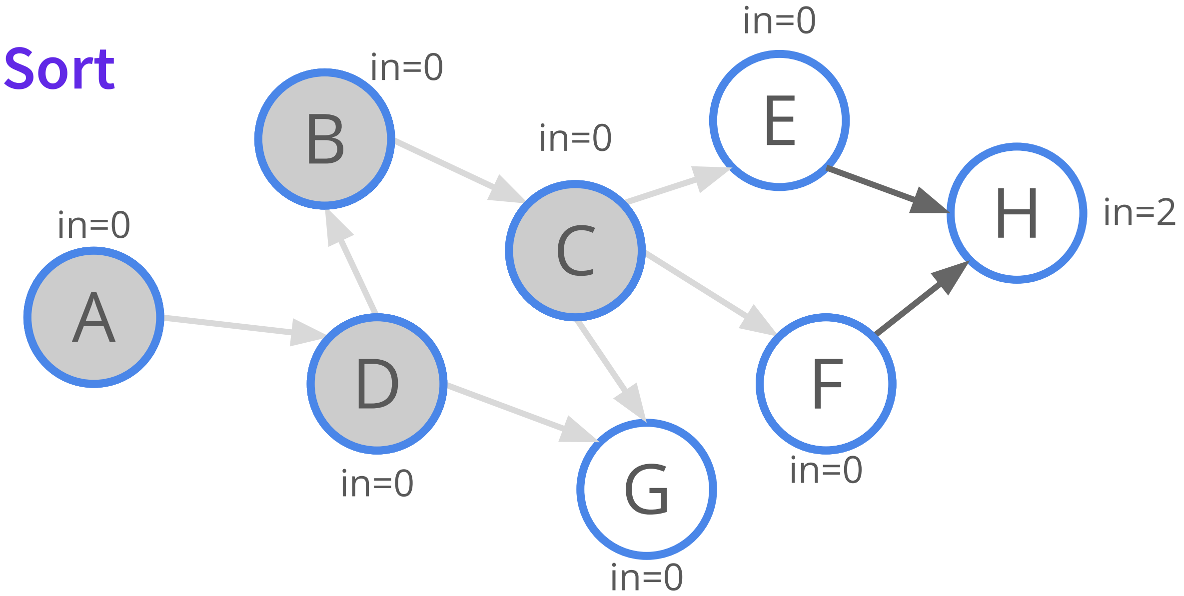
S:



Topological Order							
A	D	B	C				

DAG - Topological Sort

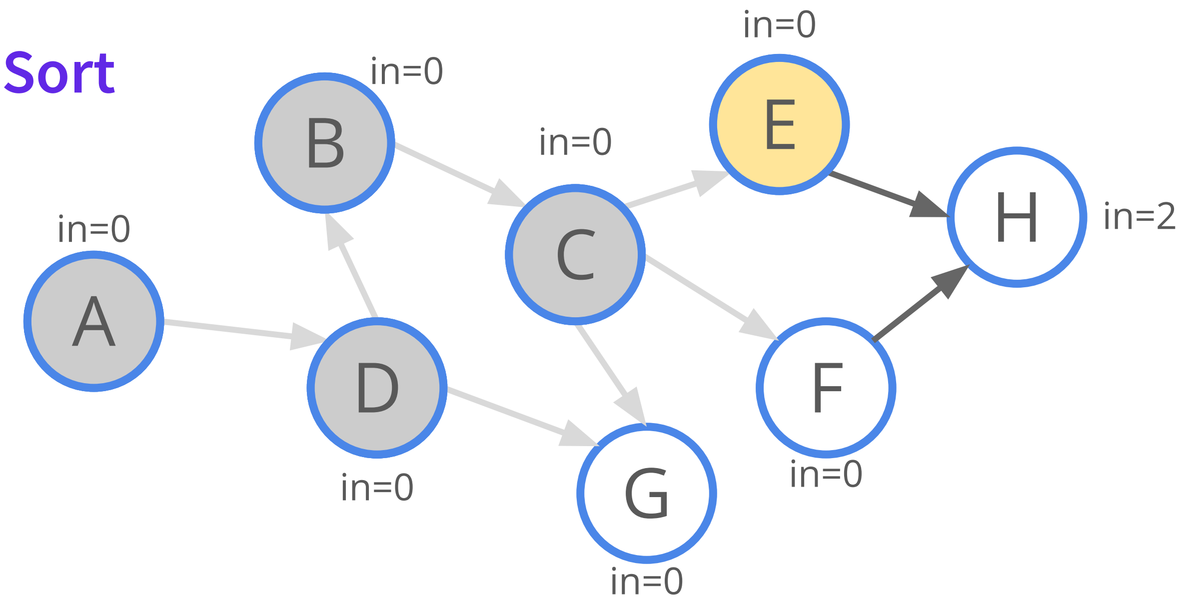
S: E, F, G



Topological Order							
A	D	B	C				

DAG - Topological Sort

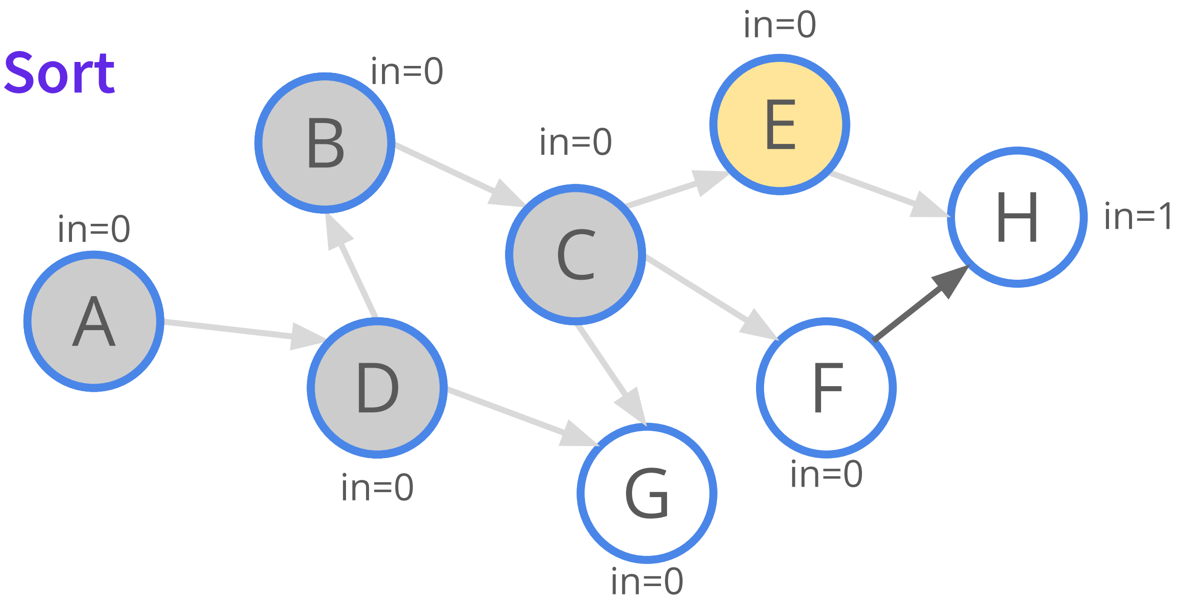
S: F, G



Topological Order							
A	D	B	C	E			

DAG - Topological Sort

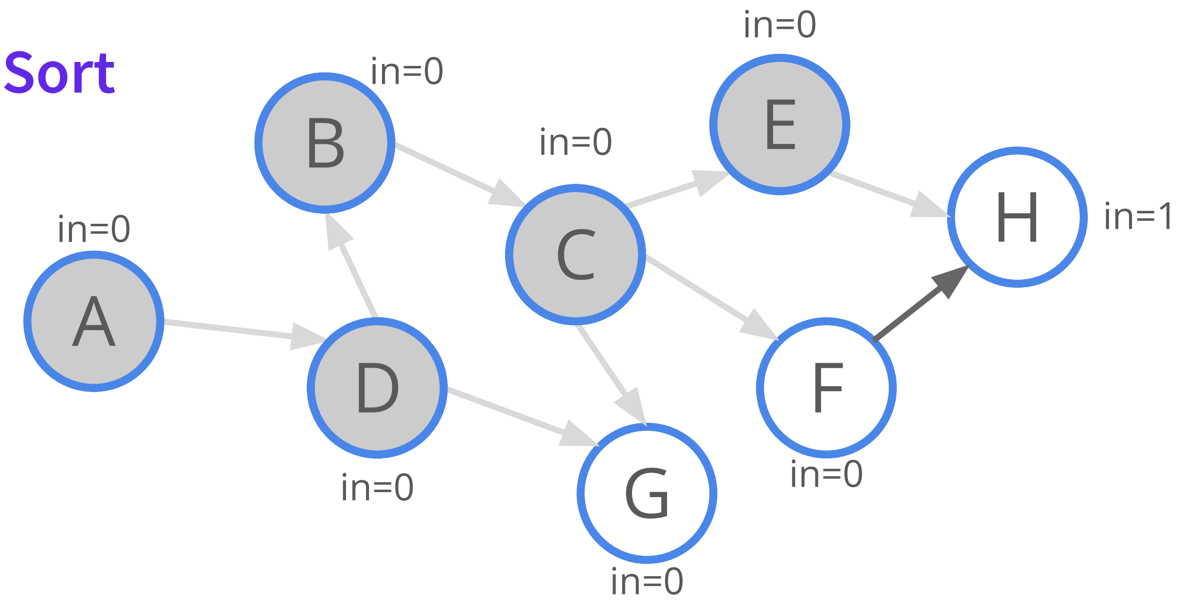
S: F, G



Topological Order							
A	D	B	C	E			

DAG - Topological Sort

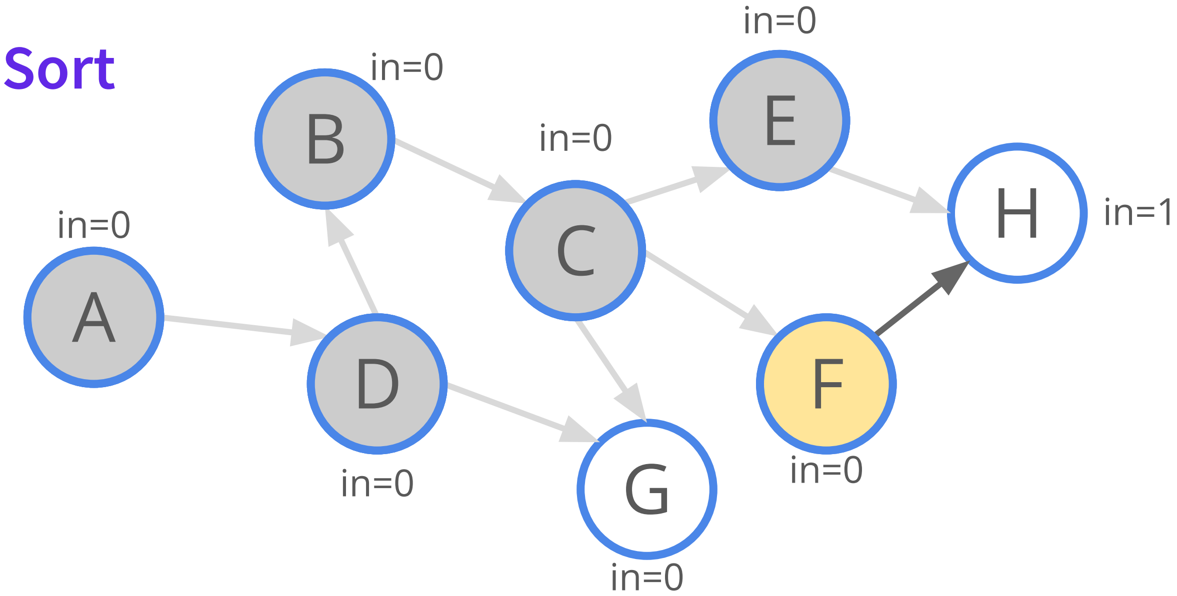
S: F, G



Topological Order							
A	D	B	C	E			

DAG - Topological Sort

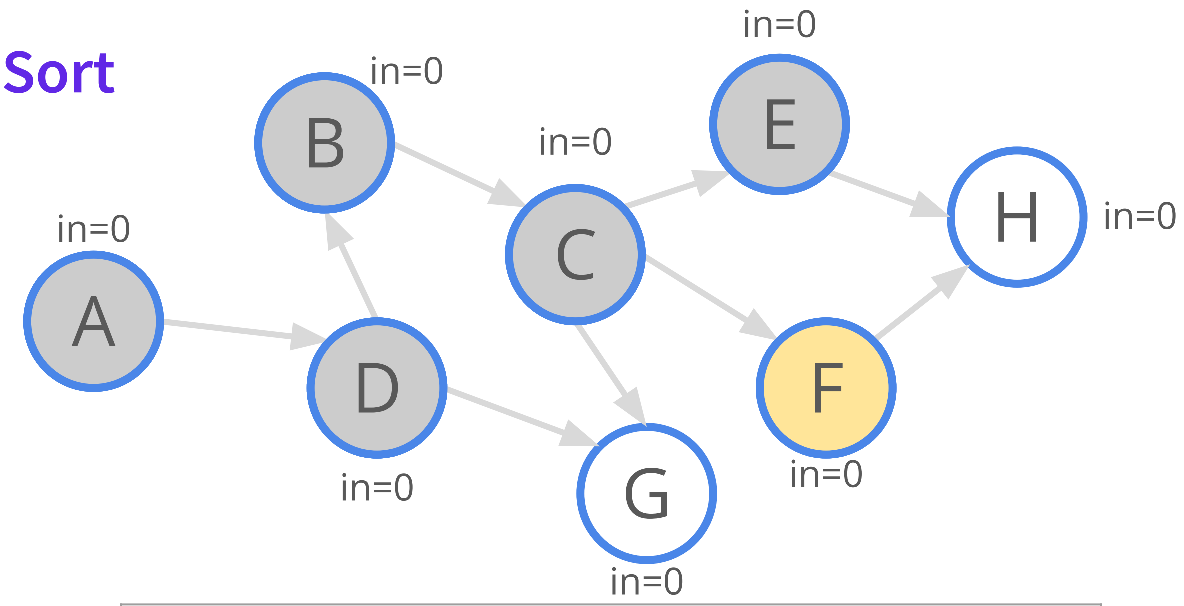
S: G



Topological Order							
A	D	B	C	E	F		

DAG - Topological Sort

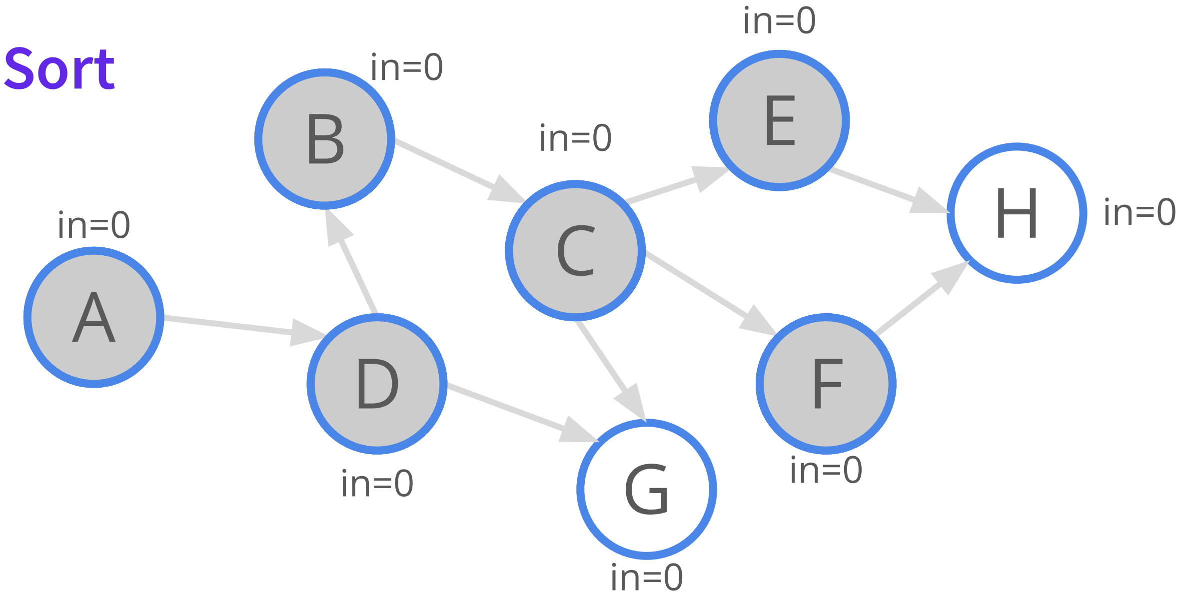
S: G



Topological Order							
A	D	B	C	E	F		

DAG - Topological Sort

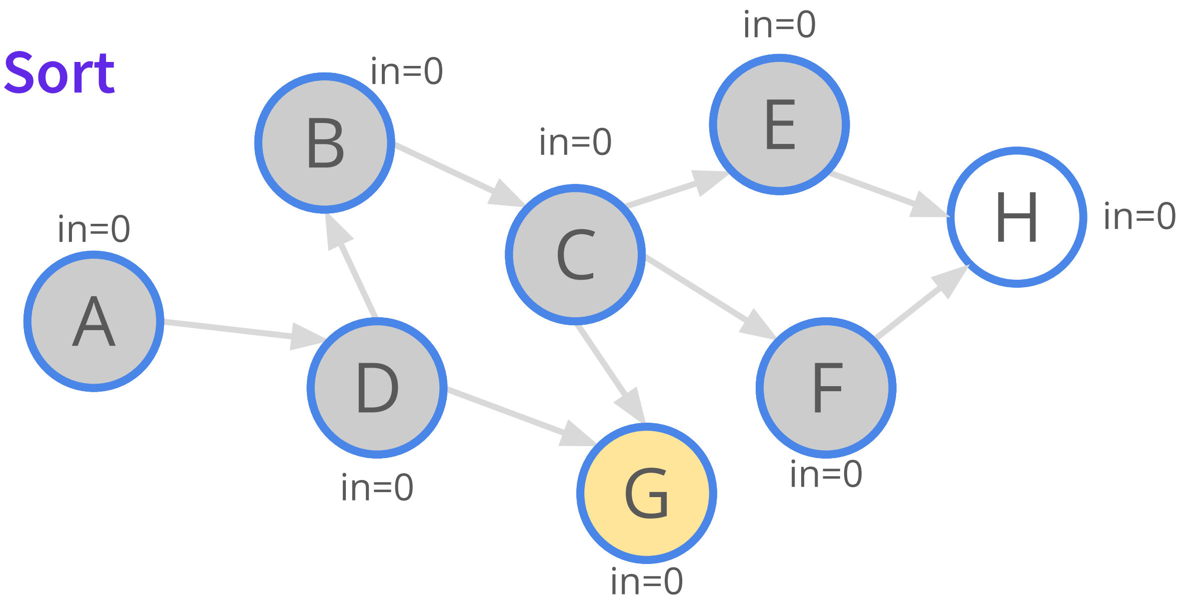
S: G, H



Topological Order							
A	D	B	C	E	F		

DAG - Topological Sort

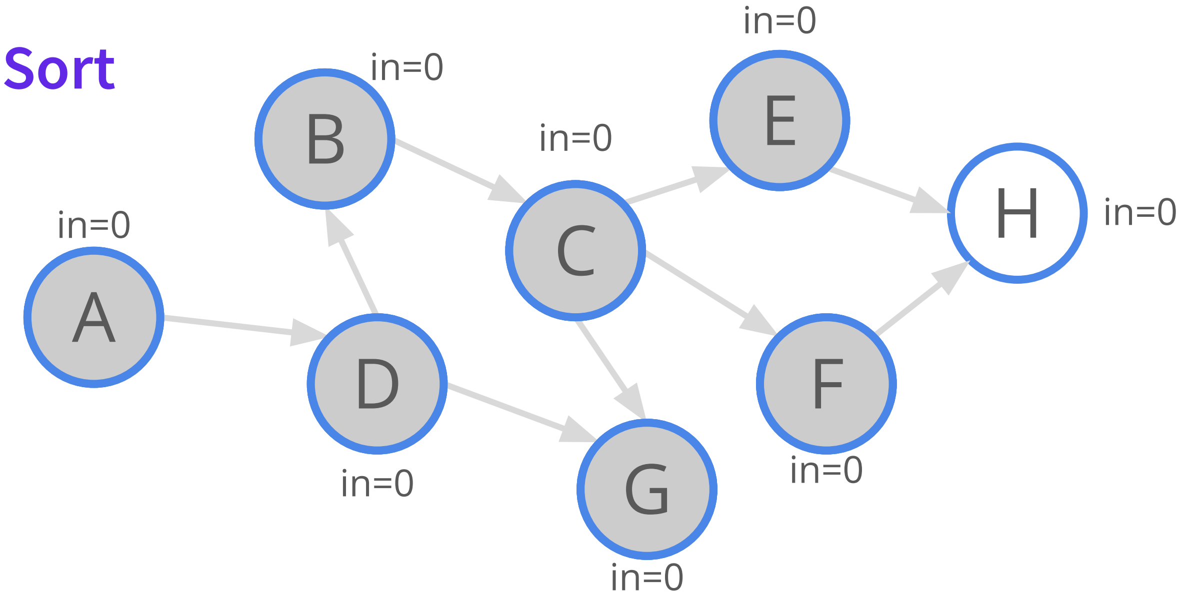
S: H



Topological Order							
A	D	B	C	E	F	G	

DAG - Topological Sort

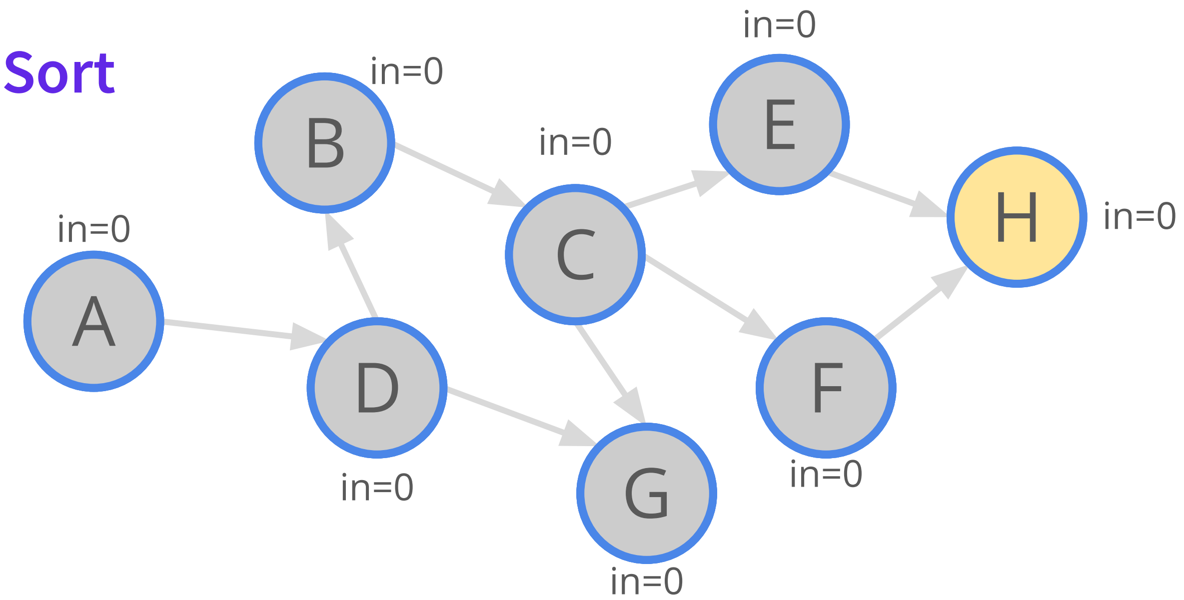
S: H



Topological Order							
A	D	B	C	E	F	G	

DAG - Topological Sort

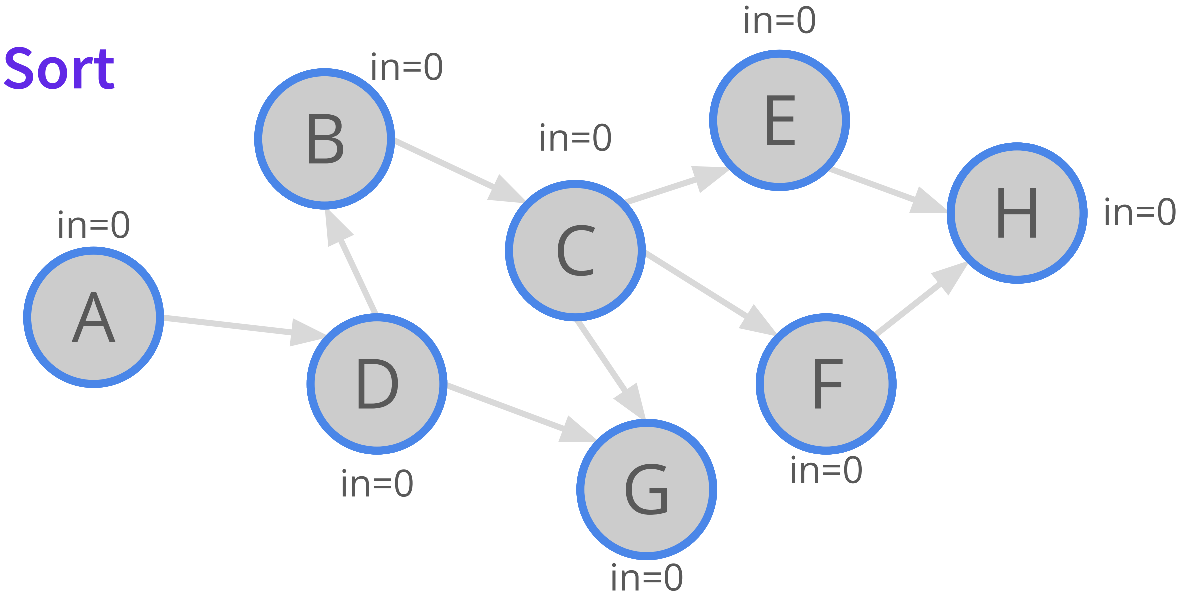
S:



Topological Order							
A	D	B	C	E	F	G	H

DAG - Topological Sort

S:



Topological Order							
A	D	B	C	E	F	G	H

DAG - Topological Sort

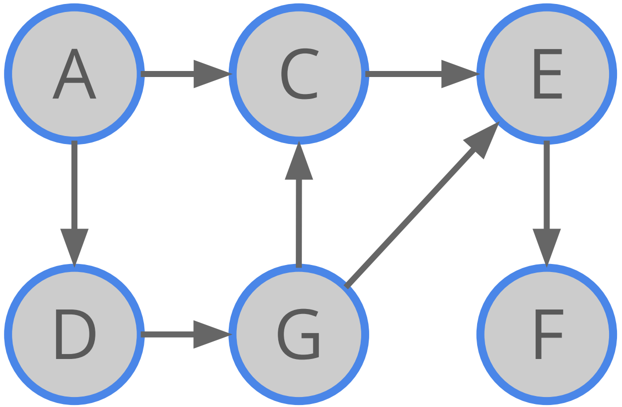
- S can be implemented by a queue, a stack, anything you like
- Time complexity: $O(V + E)$

DAG - Number of paths

- How can we use dp to count number of paths of a DAG?

Eg: 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

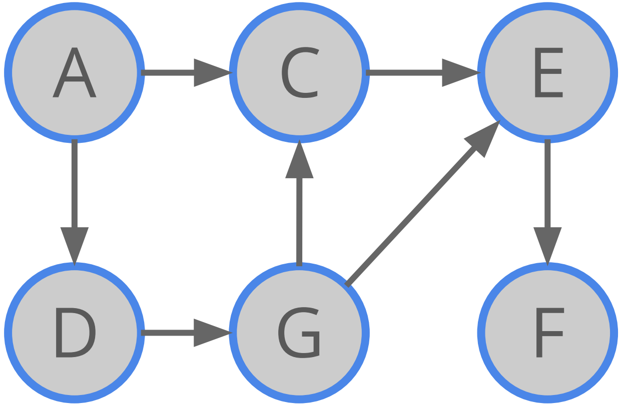
DAG - Counting number of paths

- Let paths(x) denote the number of paths from node A to node x

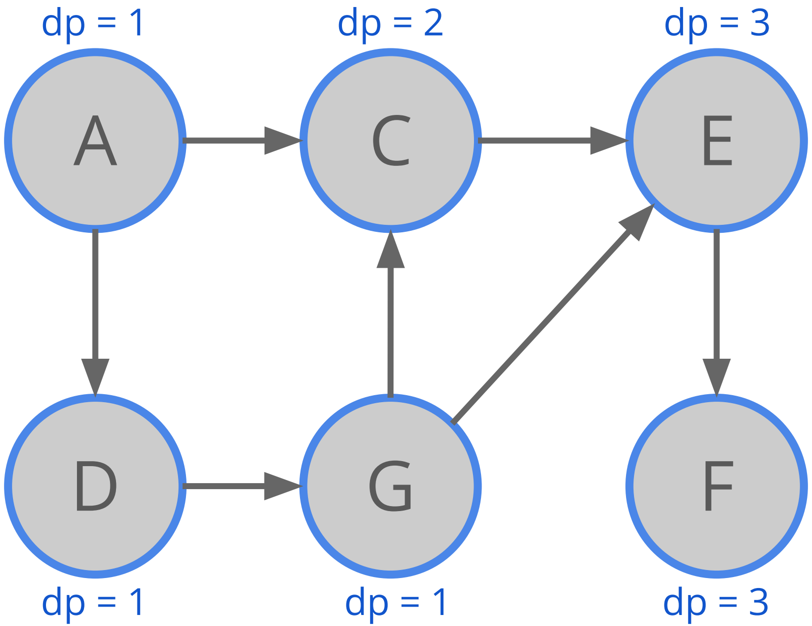
- Since the graph is acyclic, it can be counted as follows:

  $\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \ldots + \text{paths}(a_k)$

  Where a_i are the nodes from which there is an edge to x



DAG - Counting number of paths



DAG - Practice problem

- M1739 How to Run Fast
- Related to shortest path
- Learned last week!

- M1862 Little Patterns, Big Canvas
- You may refer to slides last year

DAG - M1739 How to Run Fast

- Number of shortest paths from a source in an undirected graph
- Perform Dijkstra to find shortest distance from source to each node (or any shortest path algorithm you like)
- If (dist[A] + edge_cost == dist[B]) then we add a directed edge from A to B
- Transformed into counting number of paths in DAG

break;

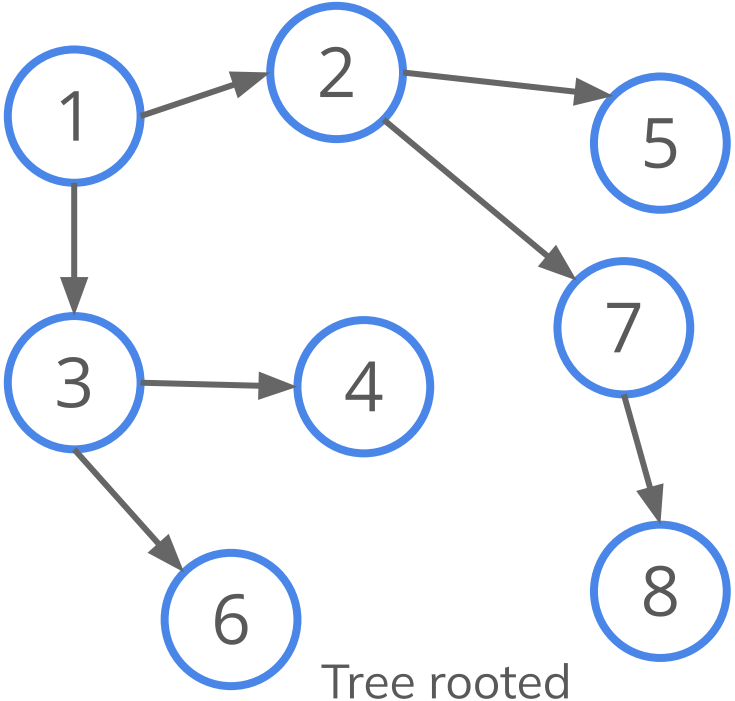
Tree - For those who feel bored

Tree (Codeforces gym 104077 L)

Group Homework (Codeforces gym 104008 G)

Tree

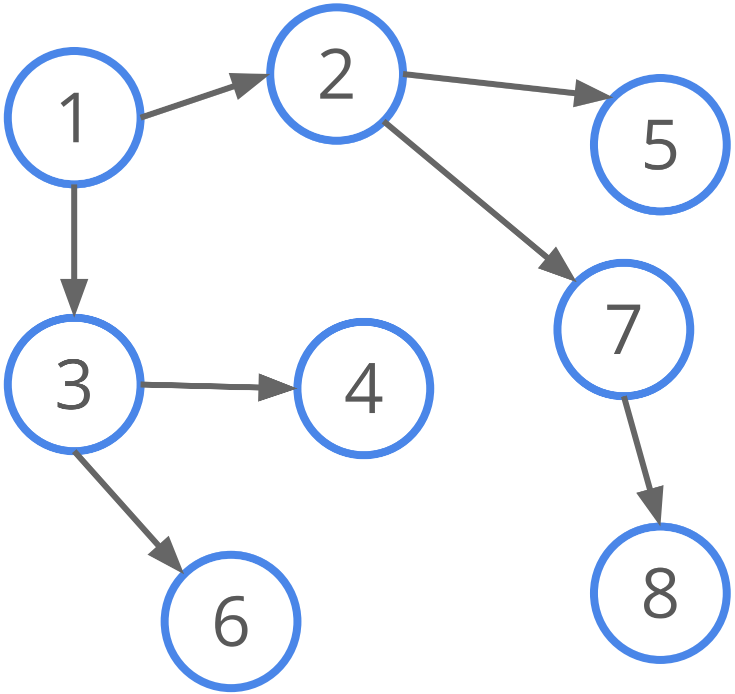
- A special case of DAG
  (if it is rooted)
- N nodes and N-1 edges
- Exist an unique path from a node to any other nodes



Tree rooted at node 1

Tree - Things you should know

- Root, Leaf
- Parent, Child
- Ancestor, Descendant
- Height, Depth
- Subtree



Tree DP

- Given a rooted tree - easier

- Given an unrooted tree with bidirectional edges
- You may need to root it yourself
- e.g. by assigning a random node as the root

Tree DP

- Assume a tree is rooted
- Use nodes as DP states
- Use nodes’ children as transition formula reference

Tree DP - Example 0 - Subtree size

- Given a rooted tree of size **N**
- Calculate the size of each subtree

- For each node, recursively count number of nodes in the subtree

- Time complexity = $O(N^2)$

Tree DP - Example 0 - Subtree size

- Given a rooted tree of size **N**
- Calculate the size of each subtree

- `dp[i] = size of subtree i`
- `dp[i] = 1 + sum(dp[j]) where j is i's children`

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

Tree DP - Example 1 - Subtree max

- Given a rooted tree of size **N**
- Each node is labeled by a value
  - Node `i` has the value `v[i]`
- **Q** queries
  - Find the greatest value in a subtree

Tree DP - Example 1 - Subtree max

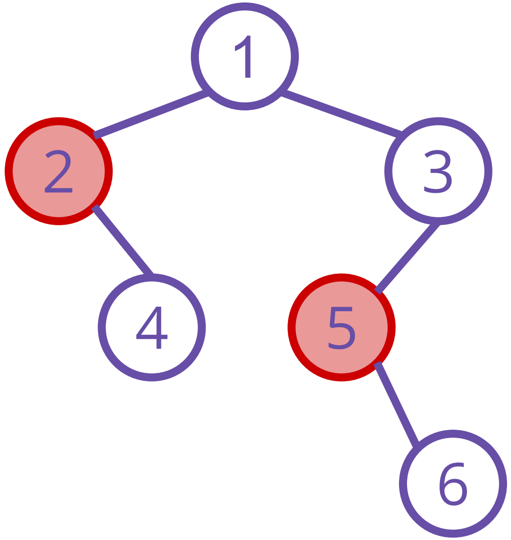
- Given a rooted tree of size **N**
- Each node is labeled by a value
  - Node `i` has the value `v[i]`
- **Q** queries
  - Find the greatest value in a subtree

- `dp[i] = answer for subtree i`
- `dp[i] = max(v[i], dp[j]) where j is i’s children`

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

Tree DP - Example 2 - Painter

- Given a rooted binary tree with size **N**
- You have to paint all nodes by assigning painter to nodes
- A painter at a node can paint the node itself, its parent and its immediate children
- Find the minimum number of painters required



Time complexity required : $O(N)$

Tree DP - Example 2 - Painter

- For each node define 3 dp states:

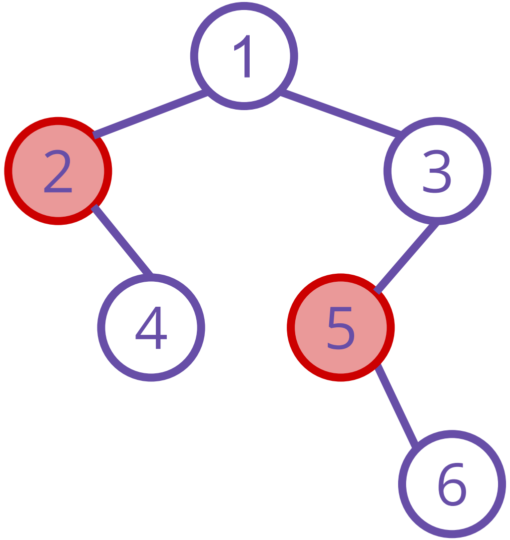
  `dp[i][0] = {assign a painter to this node}`

  `dp[i][1] = {this node is covered by its children node's painter}`

  `dp[i][2] = {this node is not covered by any other node}`

  Take min value in the transition

Time complexity = O(N)



Tree DP - Example 2 - Painter

- Base case(leaf): dp[i][0] = 1, dp[i][1] = 1, dp[i][2] = 0
- Observation: dp[i][2] <= dp[i][1] <= dp[i][0]
  - Not necessary

- consider node i with children j and k:
- dp[i][0] = {assign a painter to node i}
  - 1 + dp[j][2] + dp[k][2]
- dp[i][1] = {node i is covered by its children node's painter}
  - min(dp[j][0] + dp[k][1], dp[j][1] + dp[k][0])
- dp[i][2] = {node i is not covered by any other node}
  - dp[j][1] + dp[k][1]

Tree DP - Example 3 - Paths passing through

- Given a rooted tree of size **N**
- Calculate number of simple paths passing through each node

Tree DP - Example 3 - Paths passing through

- Calculate number of simple paths passing through each node

- `sz[i] = size of subtree i` (Example 0)
- Answer for node x can be calculated with `sz[j]` where `j` is x's children
- Treat x as the root
    - `(N - sz[x])` as one of the subtrees
- For each subtree of x,

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

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

Tree DP - Practice problem

- T094 Medical Laboratories
- Need to backtrack...

- I1022 Traffic Congestion
- You may refer to slides last year

Tree DP - T094 Medical Laboratories

- `dp[i][j] = cost of selecting j leaves in subtree of node i`
- `if x is a leaf:`
  - `dp[x][0] = dp[x][1] = 0`
- `if x has 1 child c:`
  - `dp[x][j] = dp[c][j]`
- `if x has 2 children lc and rc:`
  - `dp[x][i + j] = min(dp[lc][i] + dp[rc][j] + i * j * w[x])`

- Record how to reach the minimum answer for backtracking

Tree DP - Summary

- Rooted tree
- DFS from root
- Recursively calculate answers of the children
- Calculate answer for this node
- May require traveling several times to precompute different values
- Subtree size / sum of subtree / height / ...

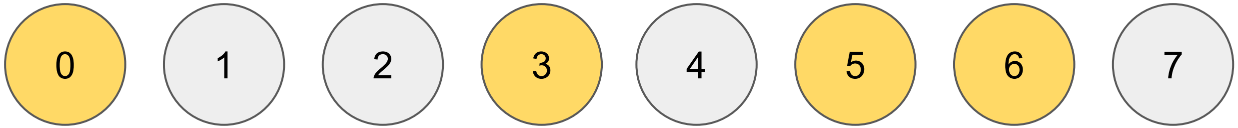
break;

Bitwise DP

- Using bitmask as some states of the dp
- E.g. `dp[3]`[01101001_2] (`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

Bitwise DP - State

- Example 1 (assume the followings are light bulbs):



- Treating the lit bulbs as 1, unlit bulbs as 0, this state can be represented by bitmask **`01101001`**$_2$ = $2^0 + 2^3 + 2^5 + 2^6$
- 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

- Example 2 (0-1 Knapsack Problem):
- Given **N** items with weight $\mathbf{w_i}$ and value $\mathbf{v_i}$, you may pick a subset of items such that their total weight ≤ **K**. Find the maximum total value of items picked
- As with the previous example, each subset 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

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))`
- Get `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 - Transition

- Given **N** light bulbs ($N \le 15$), **M** buttons, each toggles (on → off, off → on) a set of light bulbs (**B**$_i$ - in bitmask form) when pressed ($M \le 30$)
- Find minimum number of times of pressing the buttons to achieve a given state (**K**) Or output ”impossible”

Bitwise DP - Transition

- Given **N** light bulbs (N ≤ 15), **M** buttons, each toggles (on → off, off → on) a set of light bulbs (**B**$_i$ - in bitmask form) when pressed (M ≤ 30)
- Find minimum number of times of pressing the buttons to achieve a given state (**K**) Or output "impossible"

- `dp[i][bitmask]`: Considering only button 1 to `i`, the minimum number of presses needed to achieve the light bulb state in bitmask
- Base case: `dp[0][0] = 0`
- Answer: `dp[M][K]`
- Transition: `dp[i][bitmask] = min(____)`

Bitwise DP - Transition

- For each button, either choose to press it, or not press it
- `dp[i][bitmask]`: Considering only button 1 to `i`, the minimum number of presses needed to achieve the light bulb state in bitmask

- Transition (assume unachievable states are handled):
  `dp[i][bitmask]`
  `= min(dp[i - 1][bitmask ^ B[i]] + 1, dp[i - 1][bitmask])`
- Time complexity: $O(M * 2^N)$

Bitwise DP - M0712 Maximum Sum II

- Given N×N positive integers
- Find the maximum sum of N numbers
- No two numbers are 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 - M0712 Maximum Sum II

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

- Transition:
  for each column j
  ```
  if (bitmask & (1 << j) == 0)
  dp[i][bitmask + (1 << j)] = max(dp[i][bitmask + (1 << j)],
  dp[i - 1][bitmask] + a[i][j])
  ```

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

Bitwise DP - M0712 Maximum Sum II

- Transition:
  for each column j
  `if (bitmask & (1 << j) == 0)`
  `dp[i][bitmask + (1 << j)] = max(dp[i][bitmask + (1 << j)], dp[i - 1][bitmask] + a[i][j])`

- precompute number of 1s in all bitmasks
  - __builtin_popcount(bitmask)
- For each i, only consider bitmasks that number of 1s equals i - 1
- Time complexity: $O(N * 2^N)$

Bitwise DP - M2136 Guardian

- N witches, each with strength S_i and after effect E_i
- Need to fight all witches one by one
- Choosing witch x as the first one to fight against costs S_x % M energy
- For $2 \le j \le N$, choosing witch x as the j^{th} one to fight against and witch y as the
  $j - 1^{th}$ one to fight against costs $(j \times S_x \times E_y)$ % M energy
- Find minimum sum of energy to fight N witches with optimal order

Bitwise DP - M2136 Guardian

The bit (1 << i) in the bitmask represent whether witch i is already chosen

- dp[i][j] = the minimum cost to choose the witches represented by the bitmask j while the most recently chosen one is witch i
- Base case: dp[i][1 << i] = s_i % M, other states = inf
- Calculate the dp states with an increasing order of witches chosen, which is the number of 1s in the bitmasks

Bitwise DP - M2136 Guardian

- For each bitmask j and some i such that (j & (1 << i) != 0)
  - (i is already chosen in j)
- Try all k that (j & (1 << k) == 0)
  - (k is not chosen in j)
- Update dp[k][j ^ (1 << k)] with dp[i][j] + cnt * S_k * E_i % M
  - cnt = (number of 1s in j) + 1

- Answer = minimum of dp[i][(1 << N) - 1]

- Time complexity: $O(N^2 * 2^N)$

Bitwise DP - Practice Problem

- M1830 Lazy Tutor
- You may refer to 2018 minicomp 3 editorial

Memory optimization - Rolling Array

- Optimization on memory usage
- Avoid saving data that is no longer useful
- Tiny improvement on runtime

Memory optimization - Rolling Array

- Solve the following problem with DP
- N * M grid
- Calculate the number of ways to move from (1, 1) to (N, M) with right and down movement only

Memory optimization - Rolling Array

- Solve the following problem with DP
- N * M grid
- Calculate the number of ways to move from (1, 1) to (N, M) with right and down movement only
- `dp[i][j] = number of ways to move from (1,1) to (i, j)`
- `dp[1][1] = 1`
- `dp[i][j] = dp[i - 1][j] + dp[i][j - 1]`

- Memory complexity = `O(NM)`

Memory optimization - Rolling Array

- `dp[i][j] = dp[i - 1][j] + dp[i][j - 1]`
- Only `dp[i - 1][1..M]` is needed
- `dp[i][1..M]` can be calculated without referring `dp[1..i-2][1..M]`

- Keeping two rows of dp states is enough
- Alternatively use `dp[0][1..M]` and `dp[1][1..M]` for 1 to N

- Memory complexity = `O(M)`

- Swapping N and M if `M > N ⇒ O(min(N, M))`

Memory optimization - Rolling Array

- `dp[i][j] = dp[i - 1][j] + dp[i][j - 1]`

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

More Practice Problems

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

I0011 Palindrome
T003 Scheduling Lectures
I0721 Miners

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

Additional Readings

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