Shoal

Problem: Determine if all fish can meet at some puddle by swimming through moats connecting the puddles.

Shoal

Problem: Determine if all fish can meet at some puddle by swimming through moats connecting the puddles.

- Model the problem as a graph! Puddles are the nodes, moats the edges.

Shoal

Problem: Determine if all fish can meet at some puddle by swimming through moats connecting the puddles.

- Model the problem as a graph! Puddles are the nodes, moats the edges.
- They can meet at some puddle if the graph is connected
  - Start BFS or DFS at any node and check if all nodes were visited.

Shoal

- Pitfall for BFS: tracking the set of visited nodes

Shoal

- Pitfall for BFS: tracking the set of visited nodes
  - Best: mark node as visited when pushing into queue

Shoal

- Pitfall for BFS: tracking the set of visited nodes
  - Best: mark node as visited when pushing into queue
  - Still works: when popping out of the queue, check if node was already visited. If yes, skip it, otherwise mark it as visited
    - Nodes can be in the queue at most degree times

Shoal

- Pitfall for BFS: tracking the set of visited nodes
  - Best: mark node as visited when pushing into queue
  - Still works: when popping out of the queue, check if node was already visited. If yes, skip it, otherwise mark it as visited
    - Nodes can be in the queue at most degree times
  - Broken: mark as visited when popping, check for visited when looking at neighbors
    - Nodes are processed multiple times, which can amplify arbitrarily

Kingdoms

Problem: In a 2-colorable graph, which color has a higher weight.

Kingdoms

Problem: In a 2-colorable graph, which color has a higher weight.

- First color the graph

Kingdoms

Problem: In a 2-colorable graph, which color has a higher weight.

- First color the graph
    - Pick arbitrary color for start node

Kingdoms

Problem: In a 2-colorable graph, which color has a higher weight.

- First color the graph
  - Pick arbitrary color for start node
  - Use BFS/DFS to propagate, always coloring neighbors in the opposite color

Kingdoms

Problem: In a 2-colorable graph, which color has a higher weight.

- First color the graph
    - Pick arbitrary color for start node
    - Use BFS/DFS to propagate, always coloring neighbors in the opposite color
    - Graph is connected $\rightarrow$ all nodes will be colored

Kingdoms

Problem: In a 2-colorable graph, which color has a higher weight.

- First color the graph
  - Pick arbitrary color for start node
  - Use BFS/DFS to propagate, always coloring neighbors in the opposite color
  - Graph is connected $\rightarrow$ all nodes will be colored
  - Graph is bipartite $\rightarrow$ no contradictions

Kingdoms

Problem: In a 2-colorable graph, which color has a higher weight.

- First color the graph
  - Pick arbitrary color for start node
  - Use BFS/DFS to propagate, always coloring neighbors in the opposite color
  - Graph is connected $\rightarrow$ all nodes will be colored
  - Graph is bipartite $\rightarrow$ no contradictions
- Iterate over all nodes and sum up node weights

Kingdoms

Problem: In a 2-colorable graph, which color has a higher weight.

- First color the graph
  - Pick arbitrary color for start node
  - Use BFS/DFS to propagate, always coloring neighbors in the opposite color
  - Graph is connected $\rightarrow$ all nodes will be colored
  - Graph is bipartite $\rightarrow$ no contradictions
- Iterate over all nodes and sum up node weights
- Runtime $\mathcal{O}(n + m)$

Boxing

Problem: Given a chess board output the shortest path from a knight to the black king.

Boxing

Problem: Given a chess board output the shortest path from a knight to the black king.

- The board is an undirected, unweighted graph where two squares are connected if they are a knight's move apart and no square is blocked by a white piece.

Boxing

Problem: Given a chess board output the shortest path from a knight to the black king.

- The board is an undirected, unweighted graph where two squares are connected if they are a knight's move apart and no square is blocked by a white piece.
- Running a BFS from every white knight sequentially is $\mathcal{O}(w^2h^2)$

Boxing

Problem: Given a chess board output the shortest path from a knight to the black king.

- The board is an undirected, unweighted graph where two squares are connected if they are a knight's move apart and no square is blocked by a white piece.
- Running a BFS from every white knight sequentially is $\mathcal{O}(w^2h^2)$
- Instead, run a BFS from the black king until a white knight is found

Boxing

Problem: Given a chess board output the shortest path from a knight to the black king.

- The board is an undirected, unweighted graph where two squares are connected if they are a knight's move apart and no square is blocked by a white piece.
- Running a BFS from every white knight sequentially is $\mathcal{O}(w^2h^2)$
- Instead, run a BFS from the black king until a white knight is found
- Maintain the previous square for every visited square

Boxing

Problem: Given a chess board output the shortest path from a knight to the black king.

- The board is an undirected, unweighted graph where two squares are connected if they are a knight's move apart and no square is blocked by a white piece.
- Running a BFS from every white knight sequentially is $\mathcal{O}(w^2h^2)$
- Instead, run a BFS from the black king until a white knight is found
- Maintain the previous square for every visited square
- Alternatively run a BFS from every white knight at once

Topics

Problem: Find any topological sorting of a DAG

Topics

Problem: Find any topological sorting of a DAG

- track in-degree of vertices (use `vector<int>`)

Topics

Problem: Find any topological sorting of a DAG

- track in-degree of vertices (use `vector<int>`)
- track nodes with in-degree 0

Topics

Problem: Find any topological sorting of a DAG

- track in-degree of vertices (use `vector<int>`)
- track nodes with in-degree 0
  - use `vector<int>` or `queue<int>`

Topics

Problem: Find any topological sorting of a DAG

- track in-degree of vertices (use `vector<int>`)
- track nodes with in-degree 0
  - use `vector<int>` or `queue<int>`
- take one of the in-degree 0 nodes next

Topics

Problem: Find any topological sorting of a DAG

- track in-degree of vertices (use `vector<int>`)
- track nodes with in-degree 0
  - use `vector<int>` or `queue<int>`
- take one of the in-degree 0 nodes next
  - update in degree on its neighbors

Topics

Problem: Find any topological sorting of a DAG

- track in-degree of vertices (use `vector<int>`)
- track nodes with in-degree 0
  - use `vector<int>` or `queue<int>`
- take one of the in-degree 0 nodes next
  - update in degree on its neighbors
- runtime: $\mathcal{O}(n+m)$

Topics

Problem: Find any topological sorting of a DAG

- track in-degree of vertices (use `vector<int>`)
- track nodes with in-degree 0
  - use `vector<int>` or `queue<int>`
- take one of the in-degree 0 nodes next
  - update in degree on its neighbors
- runtime: $\mathcal{O}(n+m)$
- there's also a DFS-based algorithm

Topics2

Topics2

- Topological sort with tie-breaking by difficulty

Topics2

- Topological sort with tie-breaking by difficulty
- Tie breaker implementation:

Topics2

- Topological sort with tie-breaking by difficulty
- Tie breaker implementation:
  - Linear search: $\mathcal{O}(n^2 + m)$

Topics2

- Topological sort with tie-breaking by difficulty
- Tie breaker implementation:
  - Linear search: $\mathcal{O}(n^2 + m)$
  - Priority queue: $\mathcal{O}(n \log n + m)$

Topics2

- Topological sort with tie-breaking by difficulty
- Tie breaker implementation:
  - Linear search: $\mathcal{O}(n^2 + m)$
  - Priority queue: $\mathcal{O}(n \log n + m)$
    - Use `std::priority_queue`!

Topics2

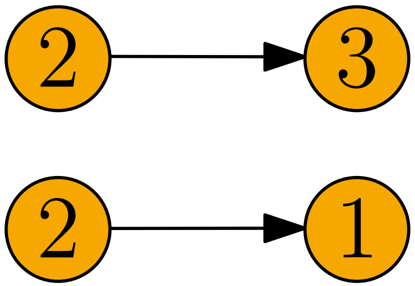
- Topological sort with tie-breaking by difficulty
- Tie breaker implementation:
  - Linear search: $\mathcal{O}(n^2 + m)$
  - Priority queue: $\mathcal{O}(n \log n + m)$
    - Use `std::priority_queue`!
  - Alternative: Search tree (`std::set`)

Topics2

- Topological sort with tie-breaking by difficulty
- Tie breaker implementation:
  - Linear search: $\mathcal{O}(n^2 + m)$
  - Priority queue: $\mathcal{O}(n \log n + m)$
    - Use `std::priority_queue`!
  - Alternative: Search tree (`std::set`)
- Distinct difficulties are important:

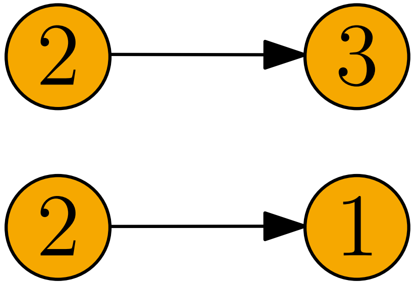
Topics2

- Topological sort with tie-breaking by difficulty
- Tie breaker implementation:
  - Linear search: $\mathcal{O}(n^2 + m)$
  - Priority queue: $\mathcal{O}(n \log n + m)$
    - Use `std::priority_queue`!
  - Alternative: Search tree (`std::set`)
- Distinct difficulties are important:



Topics2

- Topological sort with tie-breaking by difficulty
- Tie breaker implementation:
  - Linear search: $\mathcal{O}(n^2 + m)$
  - Priority queue: $\mathcal{O}(n \log n + m)$
    - Use `std::priority_queue`!
  - Alternative: Search tree (`std::set`)
- Distinct difficulties are important:



$[2, 2, 1, 3]$ vs. $[2, 1, 2, 3]$

Battery

Problem: Find the minimum battery capacity to reach 75% of the *other* cities

Battery

Problem: Find the minimum battery capacity to reach 75% of the *other* cities

- Model as undirected graph with weighted edges

Battery

Problem: Find the minimum battery capacity to reach 75% of the *other* cities

- Model as undirected graph with weighted edges
- For a given battery capacity you can check the size of the reachable subgraph

Battery

Problem: Find the minimum battery capacity to reach 75% of the *other* cities

- Model as undirected graph with weighted edges
- For a given battery capacity you can check the size of the reachable subgraph
  - Can be done using BFS or DFS and ignoring edges having a higher weight

Battery

Problem: Find the minimum battery capacity to reach 75% of the *other* cities

- Model as undirected graph with weighted edges
- For a given battery capacity you can check the size of the reachable subgraph
  - Can be done using BFS or DFS and ignoring edges having a higher weight
- Now do a binary search to find the minimum needed capacity