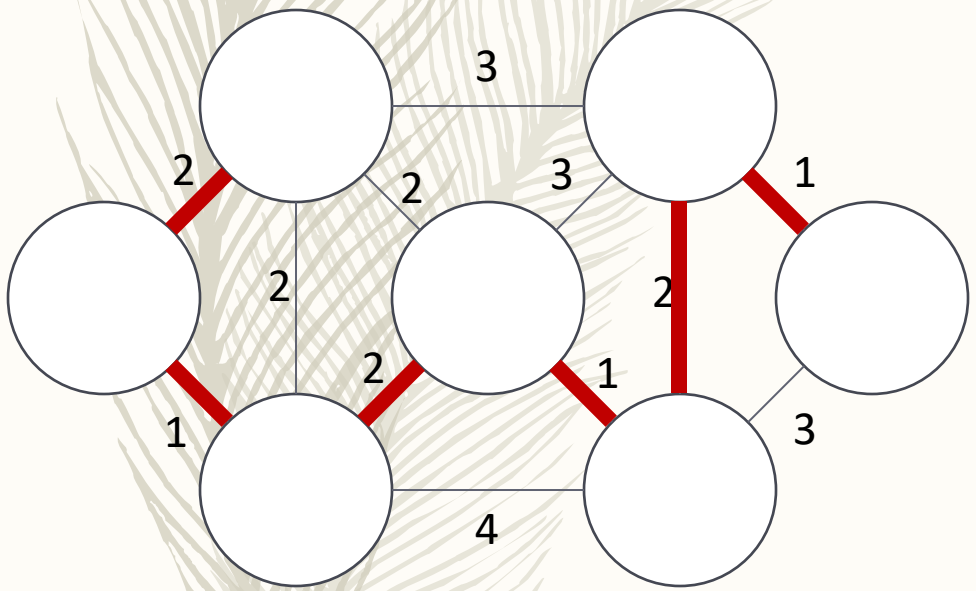
Implementing Kruskal's Algorithm

Kruskal's Algorithm



Given a connected, undirected graph G, a **spanning tree** is a subset of the edges which form a tree on the original nodes.

Given a weighted, connected, undirected graph G, a **minimum spanning tree** is the spanning tree which minimizes the sum of the edge weights.

Kruskal's Algorithm adds edges from smallest to largest value, unless adding an edge creates a cycle. It provably finds the minimum spanning tree.

Implementing Kruskal's Algorithm

What would be the first step in the implementation of Kruskal's Algorithm?

- Sort the edges.

What is the runtime of this step?

- $\Theta(m \log m)$

How do we determine if an edge creates a cycle?

- When adding edge $\langle u, v \rangle$, run BFS or DFS from u on the MST-so-far to see if there is already a path from u to v. If so, don't add the edge.

What's the runtime of this implementation?

- $\Theta(mn)$. We can do better, but we're going to need a new data structure.

The Union-Find ADT

We will maintain one set for each connected component of the MST-so-far.

- We need a Find(v) function, which returns which component node v is in. If Find(u) = Find(v), then we don't want to add edge $\langle u, v \rangle$.
- We need a Union(u, v) function, which combines the components of node u and node v into the same component. We use this when we add edge $\langle u, v \rangle$.

How we implement this will dictate the runtime of these functions, but we can figure out how to use them in Kruskal's Algorithm first.

Kruskal's Pseudocode

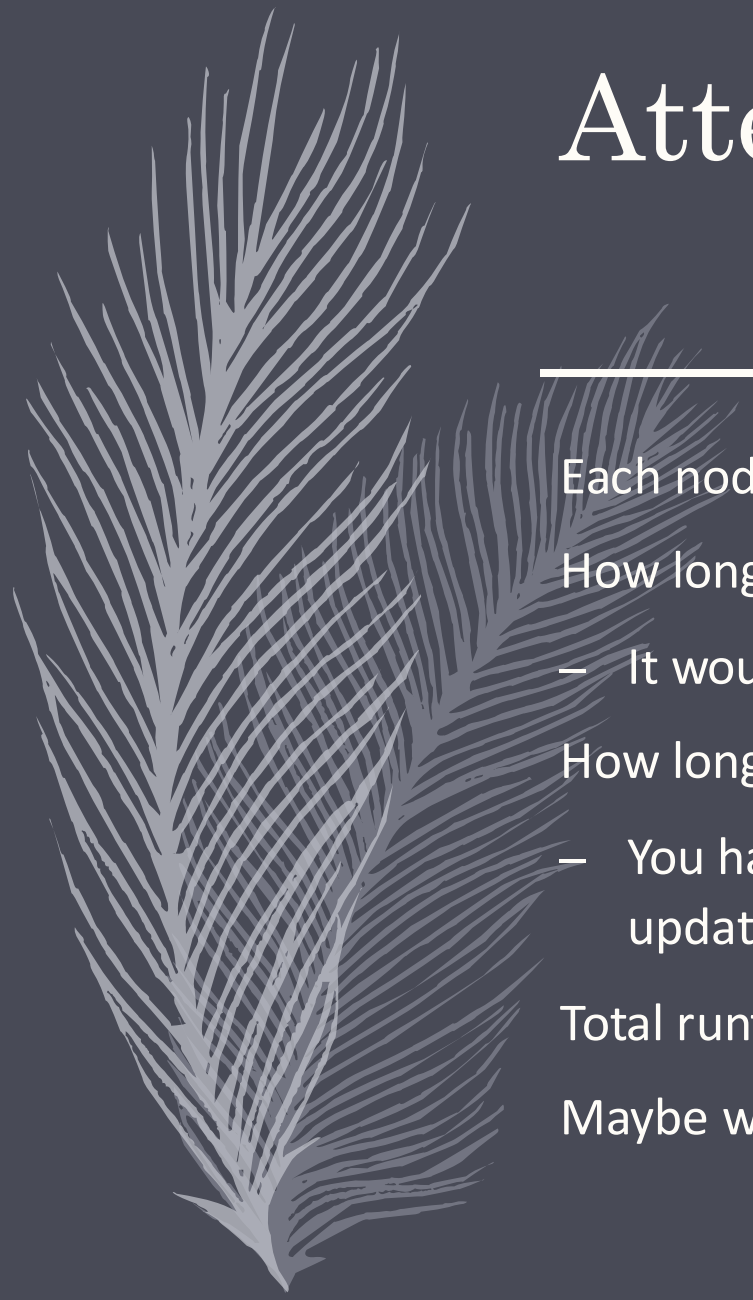
```
Kruskal(V, E)
    Sort E in ascending order of weight.
    For each edge ⟨u, v⟩, in sorted order
        a = Find(u)
        b = Find(v)
        if (a ≠ b) then
            Union(a, b)
            Add edge ⟨u, v⟩ to the MST
```

Θ(m log m)

Find called 2m times

Union called n-1 times

Runtime = Θ(m log m + m · Find + n · Union)



Attempt #1

Each node has a variable which stores the name of the set which it is in.

How long would Find take?

- It would take $\Theta(1)$

How long would Union take?

- You have to walk through all the nodes to check which ones need to be updated. $\Theta(n)$.

Total runtime = $\Theta(m \log m + n^2)$. Better, but still room for improvement.

Maybe we can use pointers to avoid updating all of the nodes somehow?

Attempt #2

Each component is an object, and each node directly points to the component it is in.

Problem: how do we do Union?

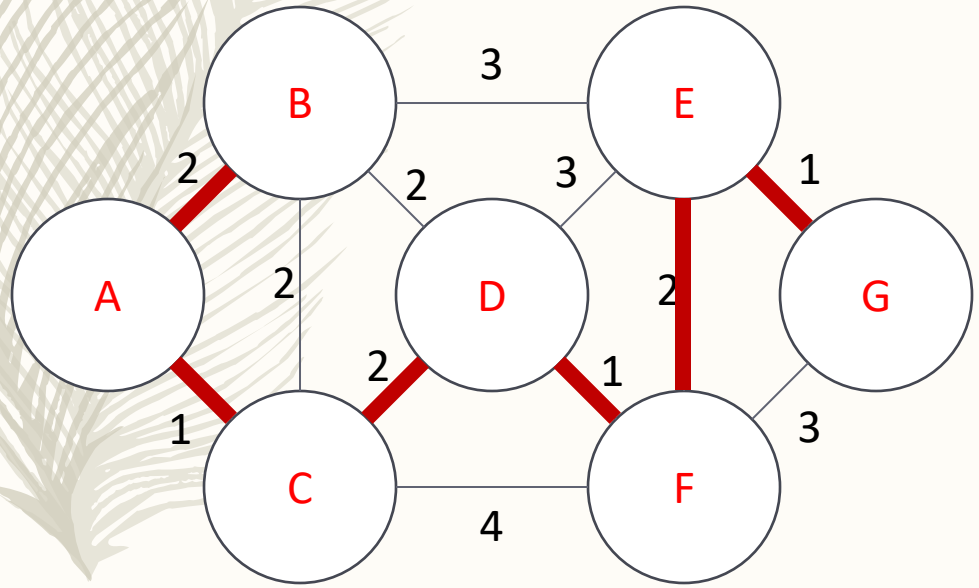
- We could rename the component, but now there are two objects with the same name. Eventually this would devolve back into Attempt #1 where we have to find all of the objects with a given name and update it.
- We could redirect what the node points to, but this also devolves back into Attempt #1 where we have to walk through all of the nodes and update their pointers.

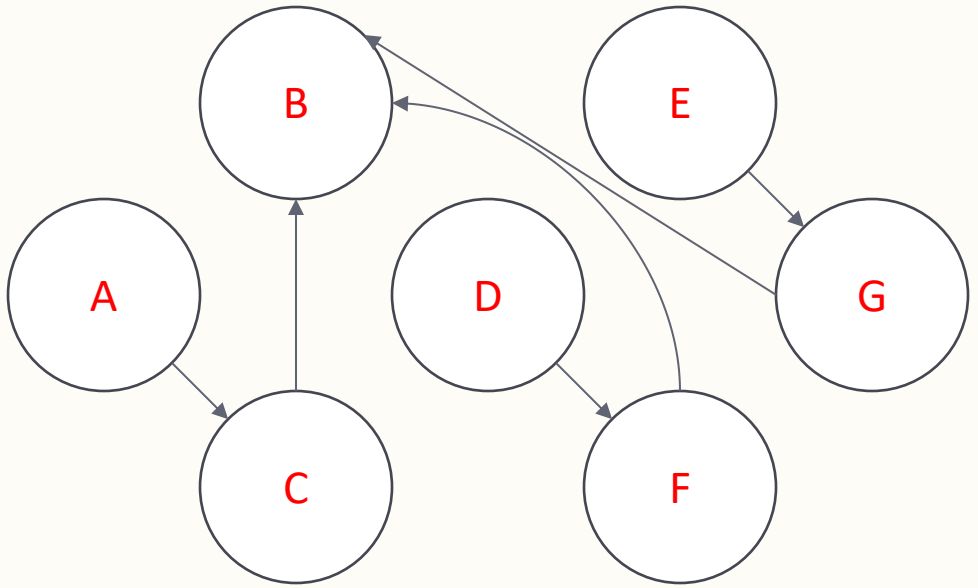
We're not using pointers well. We need to use them more creatively.

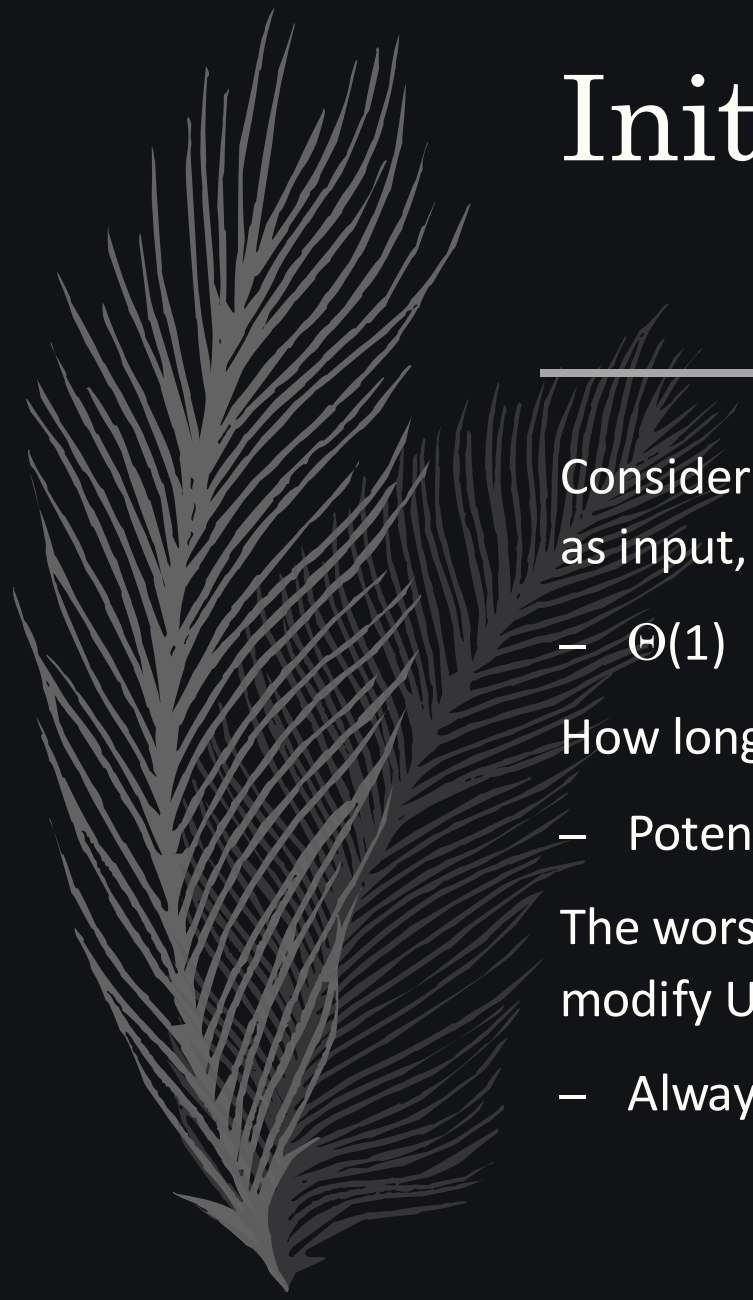
Attempt #3

Every node points to a “parent” node that is in the same component.

- The root of the component has no parent, and is the “captain”, or “identifier”, of that component.







Initial Union-Find Analysis

Considering that Find(x) finds the captain of x's set, and Union only takes captains as input, what is the runtime for Union?

- $\Theta(1)$

How long does Find take?

- Potentially $\Theta(n)$, if the parent pointers form a linked list.

The worst-case analysis of Find requires a rather stupid decision. How could we modify Union to ensure the runtime of Find is minimized?

- Always point the smaller-depth tree to the larger depth tree.

Improving the Union-Find Analysis

If we do a union between trees of depth x and y, with x < y, what is the depth?

- It is y

When would a union produce a tree of greater depth?

- When x = y

What is the depth of a 1, 2, or 3-node tree?

- 1, 2, and 2. The only way to make a 3-node tree is to combine a 2-node tree (2 depth) with a 1-node tree (1 depth), for a total of 2 depth.

Union-Find Analysis

What is the max possible depth for a 4-node tree?

- It's 3, if you combine two 2-node trees.

What is the minimum number of nodes needed to produce a depth of 4?

- It's 8, since you must combine two 3-depth trees.

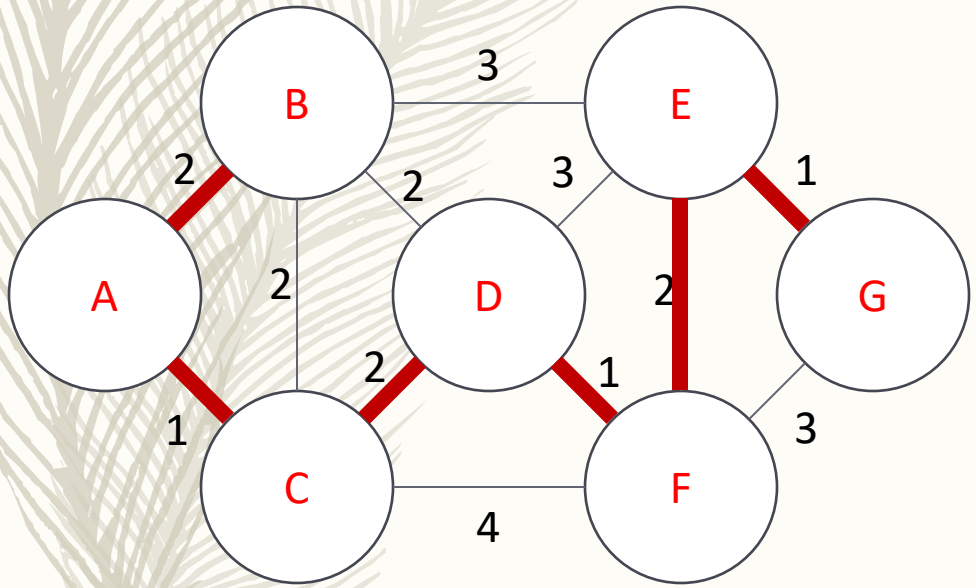
What is the minimum number of nodes needed to produce a depth of 5?

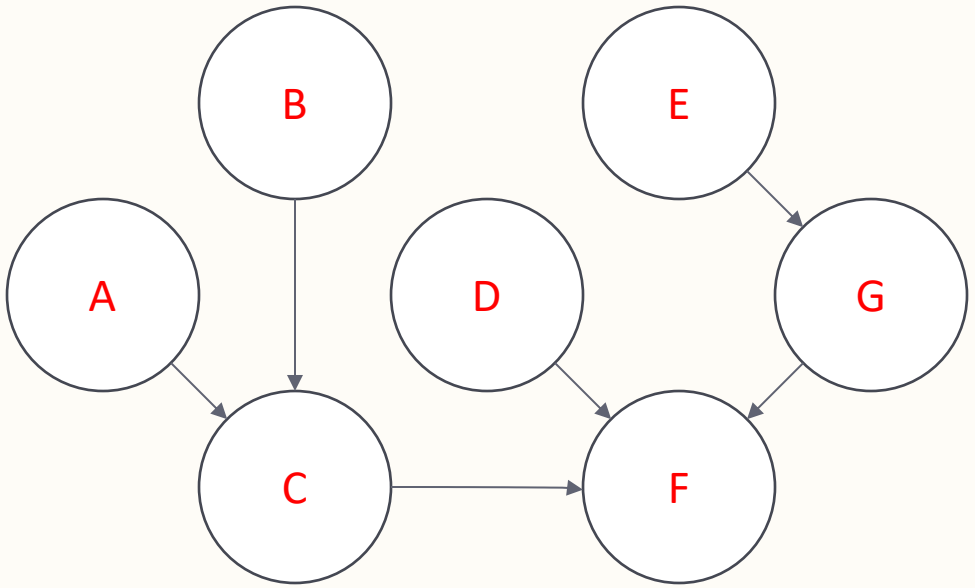
- It's 16, since you must combine two 4-depth trees.

What is the runtime of Find?

- It's $\Theta(\log n)$

A Review of the Example





Kruskal's Algorithm

The runtime of Kruskal's was
$\Theta(m \log m + m \cdot \text{Find} + n \cdot \text{Union})$

- Simplifies to $\Theta(m \log m)$
- The runtime is dominated by sorting!

$m \log m < m \log n^2 = 2m \log n = O(m \log n)$

- It is correct to say that Kruskal's takes $\Theta(m \log n)$
- This is the same runtime as Prim's when using a minheap.
- If we want to do better, we have to find a better sorting algorithm.

A lower bound on sorting

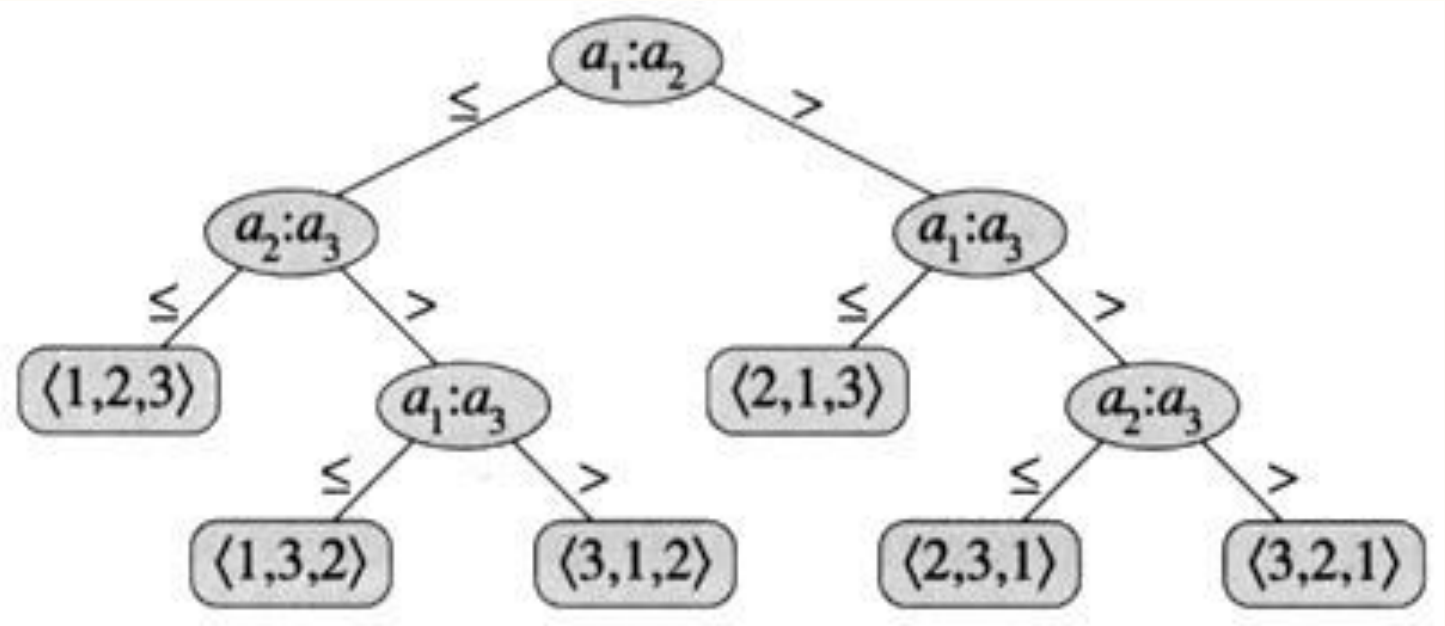
A sorting algorithm is **comparison-based** if the only method of data collection is by making direct comparisons between elements in the list.

- Every sorting algorithm you have studied (Insertion, Selection, Bubble, Merge, and Quick) is comparison-based.

We will prove that it is impossible for a comparison-based sorting algorithm to do better than $\Omega(n \log n)$.

- This is why you didn't study a sorting algorithm that does better than n log n: they don't exist!

A sorting decision tree



Sorts the elements a_1, a_2, and a_3.

Sorting Decision Trees

When we reach a leaf node in a sorting decision tree, what do we know?

– The final sorted order of the list.

How many leaf nodes do you need to sort n elements?

– At least n!, one for every possible permutation of the n elements.

How deep must the tree be to achieve this?

– $\Omega(\log n!) = \Omega(n \log n)$

If a comparison-based sorting algorithm takes less than $\Omega(n \log n)$ time, it literally doesn't have enough time to distinguish between all the possible answers!

Radix Sort

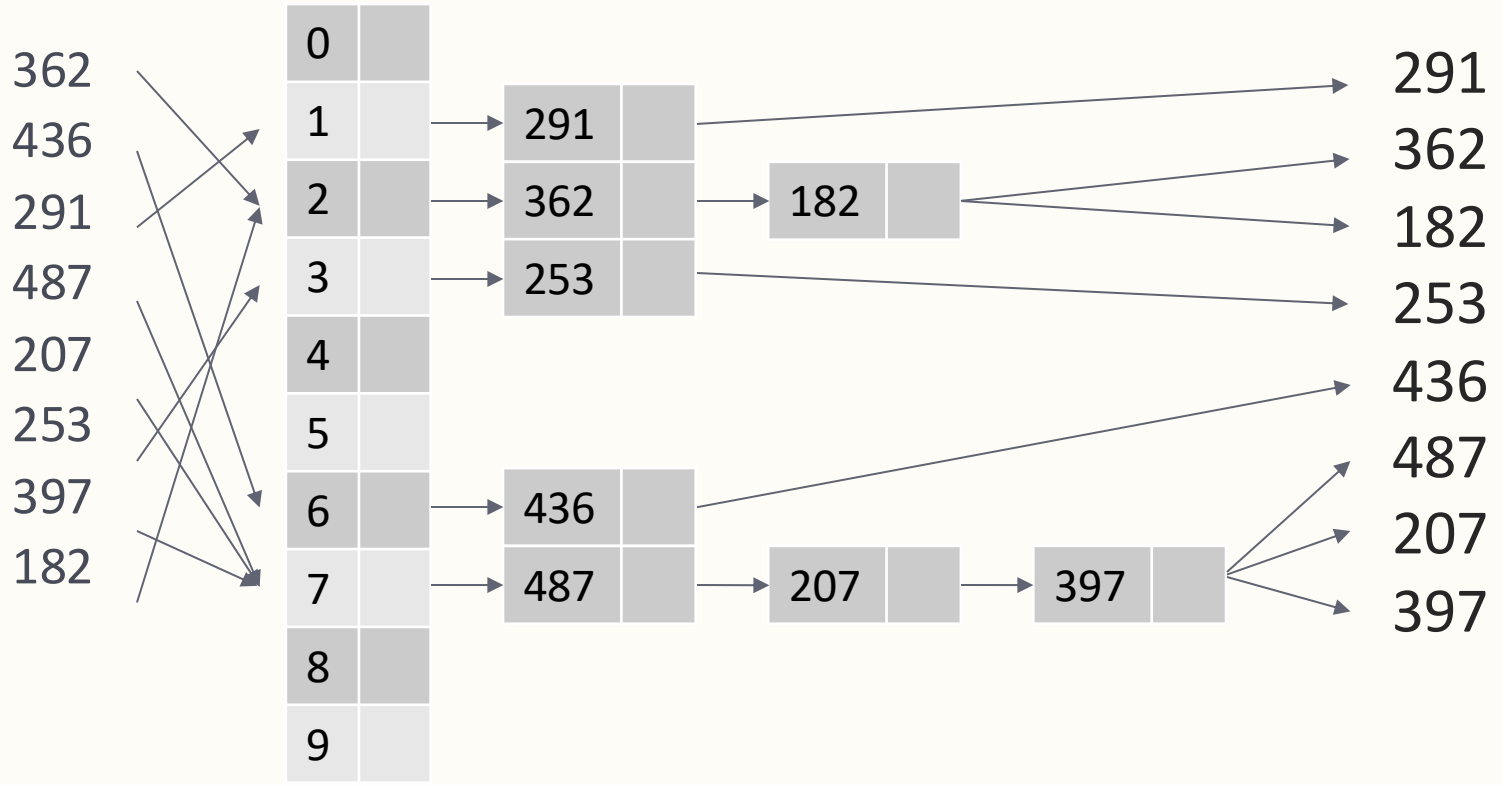
We proved that any comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

- Is there such a thing as a non-comparison-based sorting algorithm?

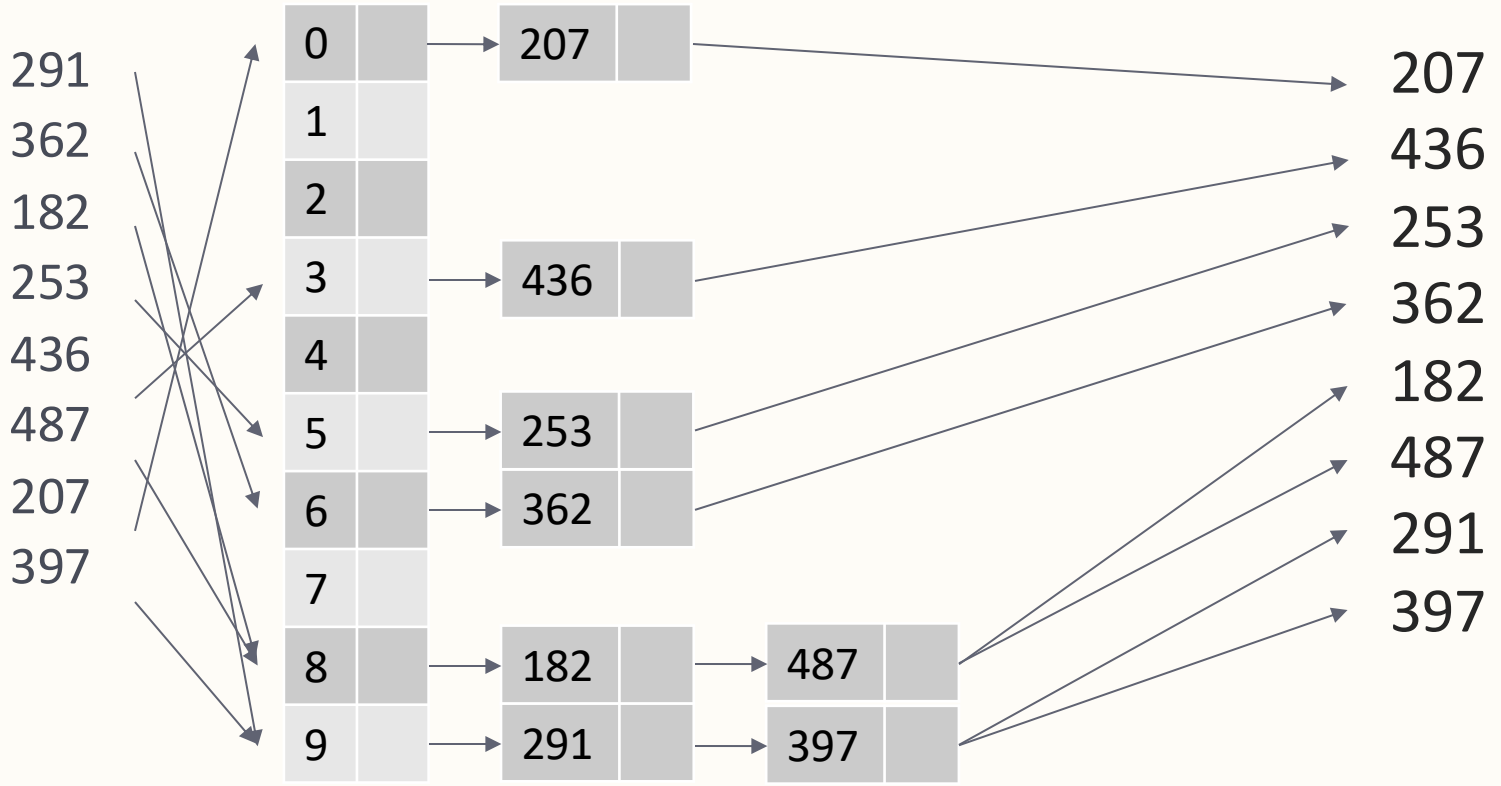
There are, and one of the most famous ones is **Radix Sort**.

- Radix Sort manages to sort a list without ever taking two elements and directly asking which is larger (it of course figures this out by the time the list is sorted, indirectly).

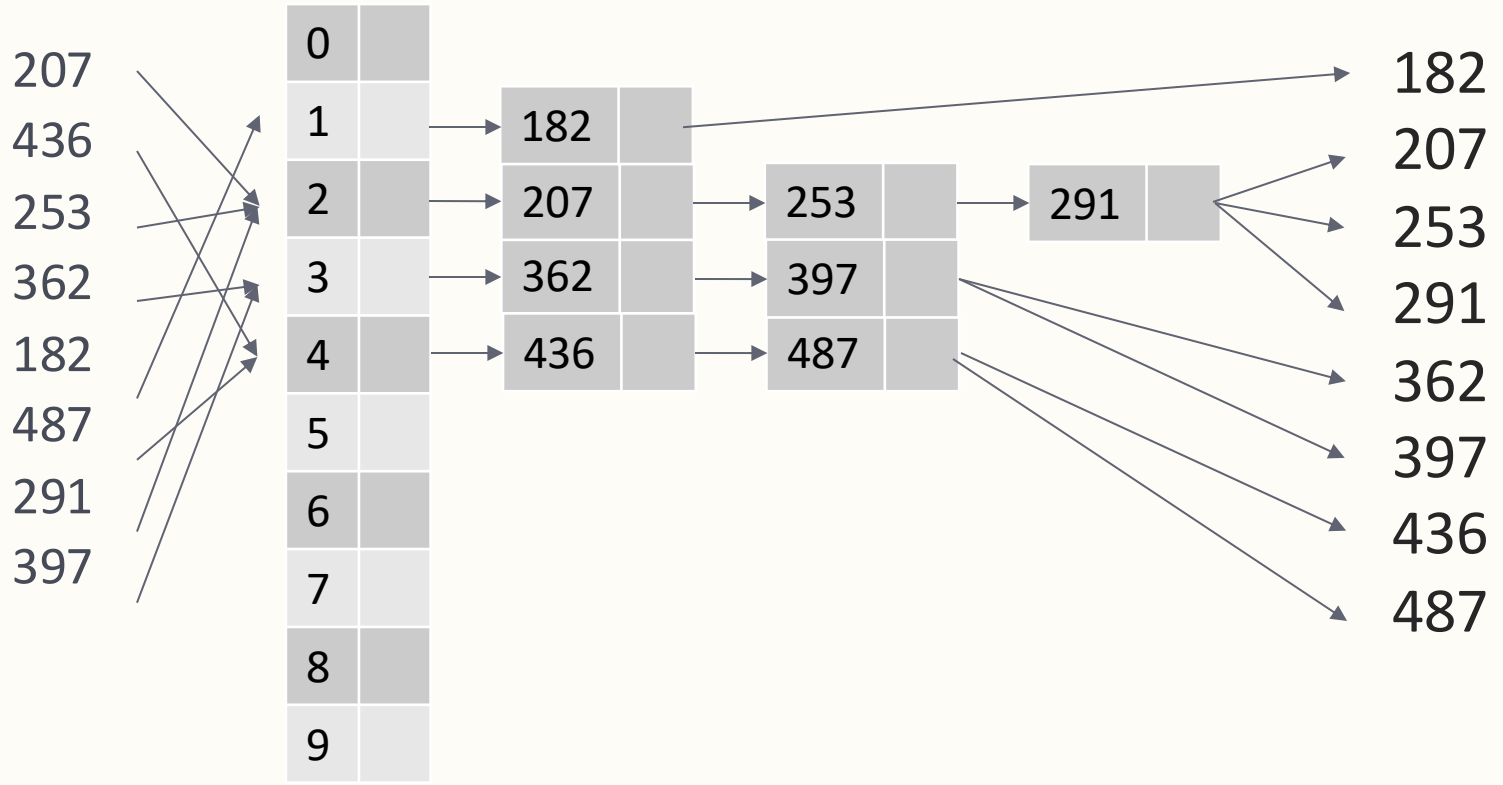
Radix Sort, Phase 1



Radix Sort, Phase 2



Radix Sort, Phase 3



Analysis

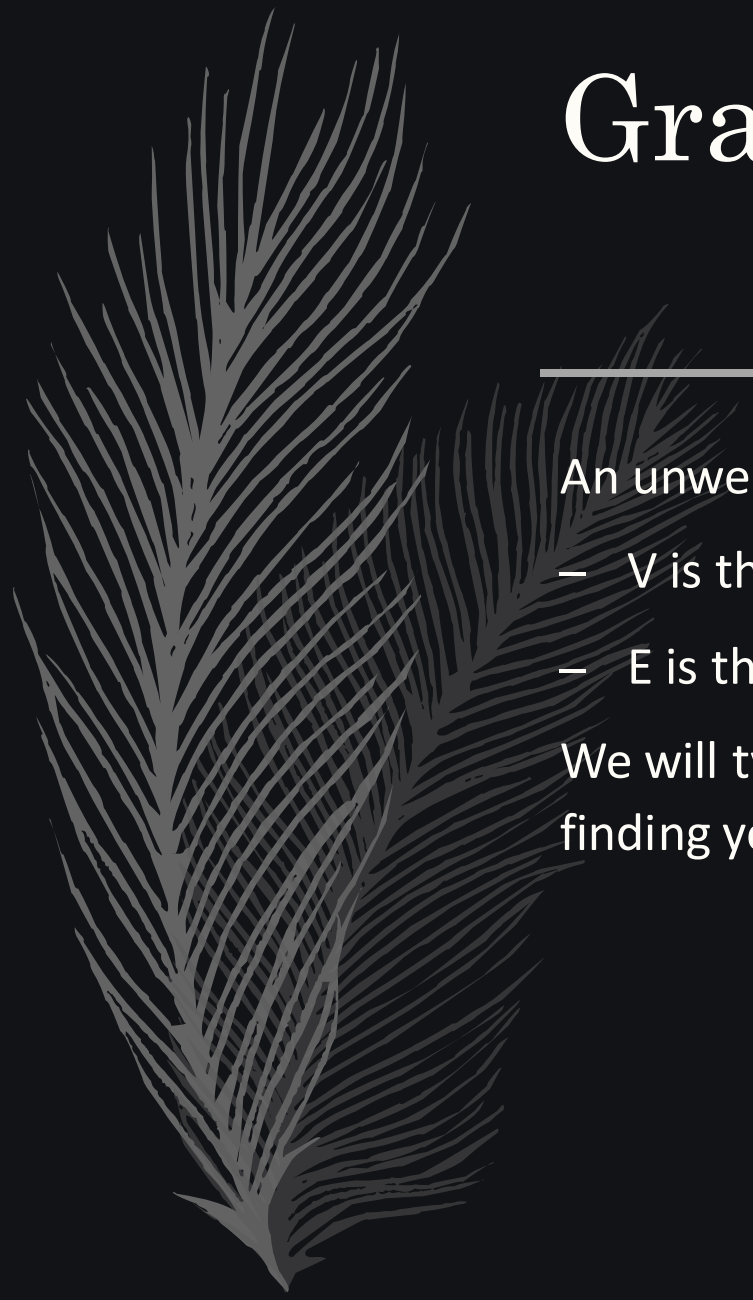
What is the runtime of Radix Sort?

– $\Theta(nd)$, where d is the maximum number of digits in any input.

Is this better than Merge Sort?

– Depends on the value of d!

If you can make assumptions about the data (such as d will be small), then you can do better than n log n. If you can't, then n log n is the best possible.



Graphs

An unweighted graph G is defined by two sets $G = \langle V, E \rangle$.

- V is the set of vertices/nodes. $|V| = n$
- E is the set of edges/arcs. $|E| = m$

We will typically disallow self-loops in this class, as they do not contribute towards finding your way through a graph.

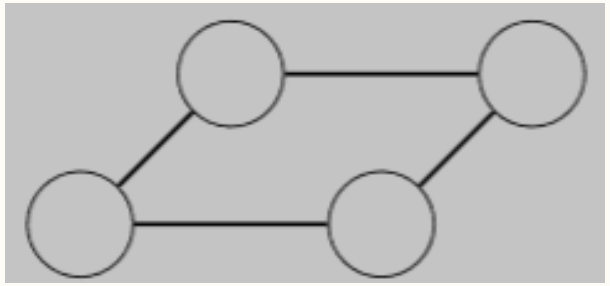
Bipartite Graphs

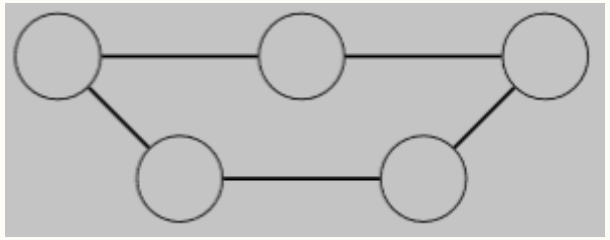
A graph is **bipartite** if it is 2-colorable.

Which graph is bipartite?

What is it about the other graph that causes it to not be bipartite?

The right is not bipartite, because it has an odd-length cycle.





Odd-Length Cycles

If G has an odd-length cycle, it cannot be bipartite.

- Not even worrying about the rest of the graph, you can't 2-color this cycle.

Is the converse true?

Is it true that if G is not bipartite, then it has an odd-length cycle?

- Let's figure it out!

Describe an algorithm that determines whether a graph is bipartite.

The Algorithm

Run BFS.

- Assign the starting node Cardinal. It doesn't matter what color we choose.
- Assign all nodes in level 1 Gold, since they have to be.
- Assign all nodes in level 2 Cardinal, since they have to be.
- Assign all nodes in an odd level Gold, and all nodes in an even level Cardinal.

If any coloring will work, this one will work. Under what conditions would this coloring not work?

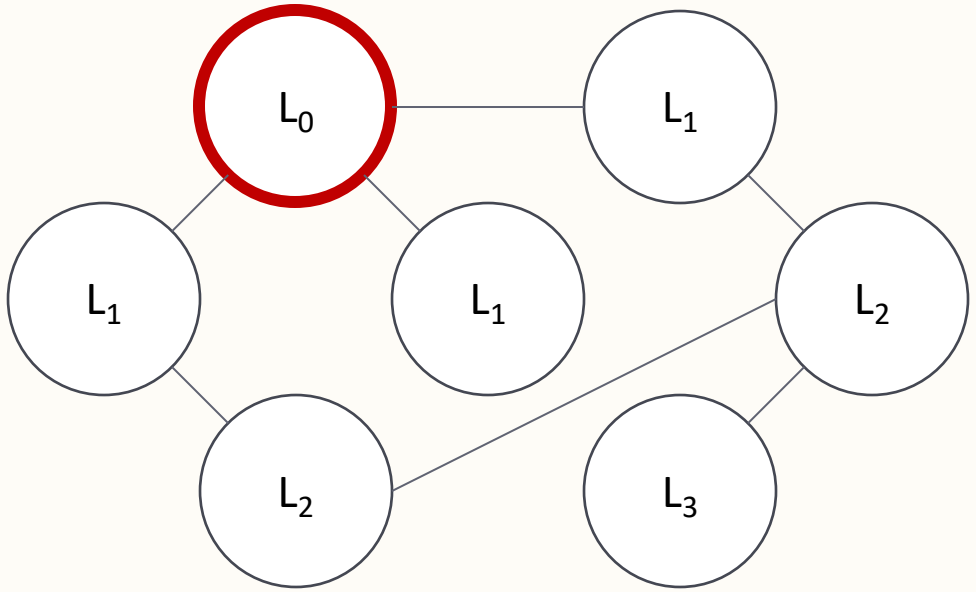
Properties of Bipartite Graphs

If there is an edge between two nodes at the same level, then the coloring will fail. Otherwise it will succeed.

- Such an edge implies an odd-length cycle!

If the nodes are k levels below their common ancestor, then the length of the cycle is 2k+1

- There is a cycle of length 5 in this example.



Connectivity

A directed graph is **weakly connected** if every pair of nodes can reach each other if you ignore edge directions.

A directed graph is **connected** if, for every pair of nodes $\langle u, v \rangle$, there is a path from u to v, or a path from v to u (possibly both).

A directed graph is **strongly connected** if, for every pair of nodes $\langle u, v \rangle$, there is a path from u to v, and a path from v to u.

Strong Connectivity

True or false?

– Let s be an arbitrary node in G. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

If G is strongly connected, then by definition, everything can reach s and s can reach everything.

If every node can reach s and s can reach everything, then every node can first go to s, and then go to its destination.

True!

Testing Strong Connectivity

How could we use this property to test if a graph is strongly connected?

Choose an arbitrary node s
Run BFS from s. If you don't reach every node, reject.
For every node $x \neq s$:

Run BFS from x. If you don't reach s, reject.

Accept.

How could we improve our algorithm?

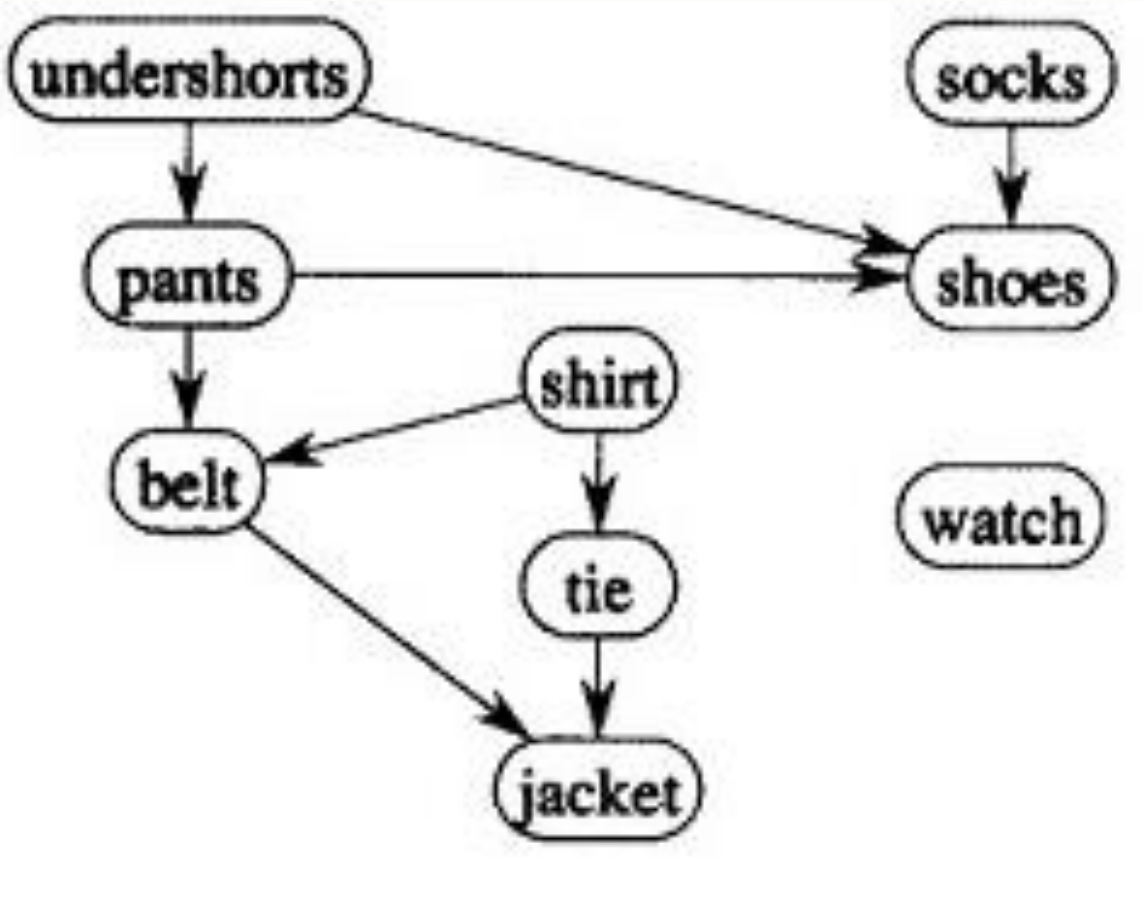
Improving the Algorithm

Given $G = \langle V, E \rangle$, define $G' = \langle V, E' \rangle$, where $\langle u, v \rangle \in V$ iff $\langle v, u \rangle \in V'$.

Choose an arbitrary node s
Run BFS from s on G. If you don't reach every node, reject.
Run BFS from s on G'. If you don't reach every node, reject.
Accept.

If s can reach every node by following edges in the wrong direction, then every node can reach s!

Runtime: $\Theta(m+n)$



DAGs

A **directed acyclic graph**, or **DAG**, is a directed graph with no cycles.

A **topological order** of a directed graph is an ordering of its nodes $v_1, \ldots, v_n$, such that for every edge $\langle v_i, v_j \rangle$, $i < j$.

One possible topological order is: undershorts, shirt, pants, socks, belt, tie, watch, jacket, shoes

How are these two concepts related?

DAGS and Topological ORDERS

If a graph has a topological order, it must be a DAG

- Or, equivalently, if it is not a DAG, it doesn't have a topological order.

Is the converse true?

- Is it true that if a graph is a DAG, it must have a topological order?

Let's figure it out!

Properties of DAGS

True or False? If G is a DAG, then it has a node with no incoming edges.

- **True**. Why?
- Assume G is a DAG, and every node has an incoming edge.
- Choose an arbitrary node v.
- v has an incoming edge from some node u. Go to u.
- u has an incoming edge from some node w. Go to w.
- We will never hit a dead-end, since everything has an incoming edge. Therefore we will eventually return to a node we've visited. Contradiction!

An algorithm to find a topological ordering

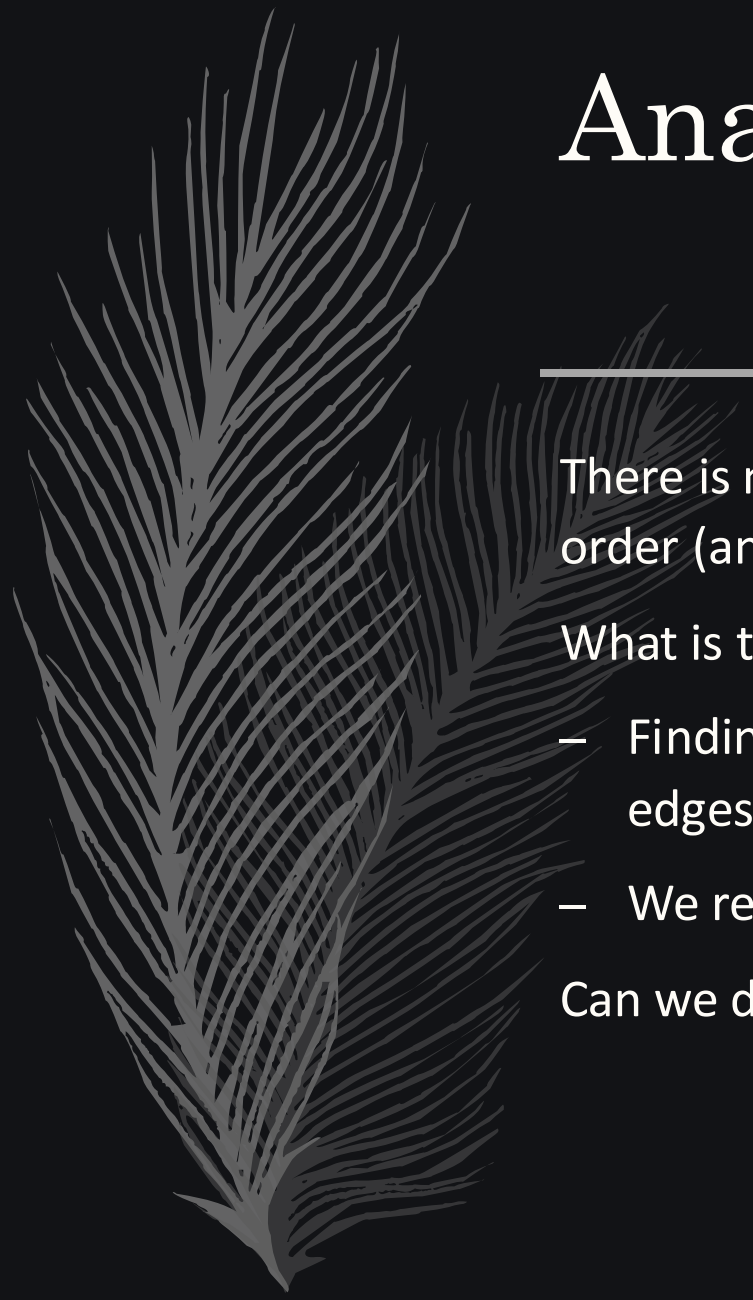
How can we use this property of DAGs to identify a topological ordering of a DAG?

– The node v with no incoming edges is the first node in the topological order. Now what?

Remove v from the graph including its edges (G – v). What kind of graph is this now?

– It's still a DAG, since we clearly can't have created a cycle by removing things. Therefore, it has a node with no incoming edges!

That node is node 2. Repeat the process until you have a topological ordering!



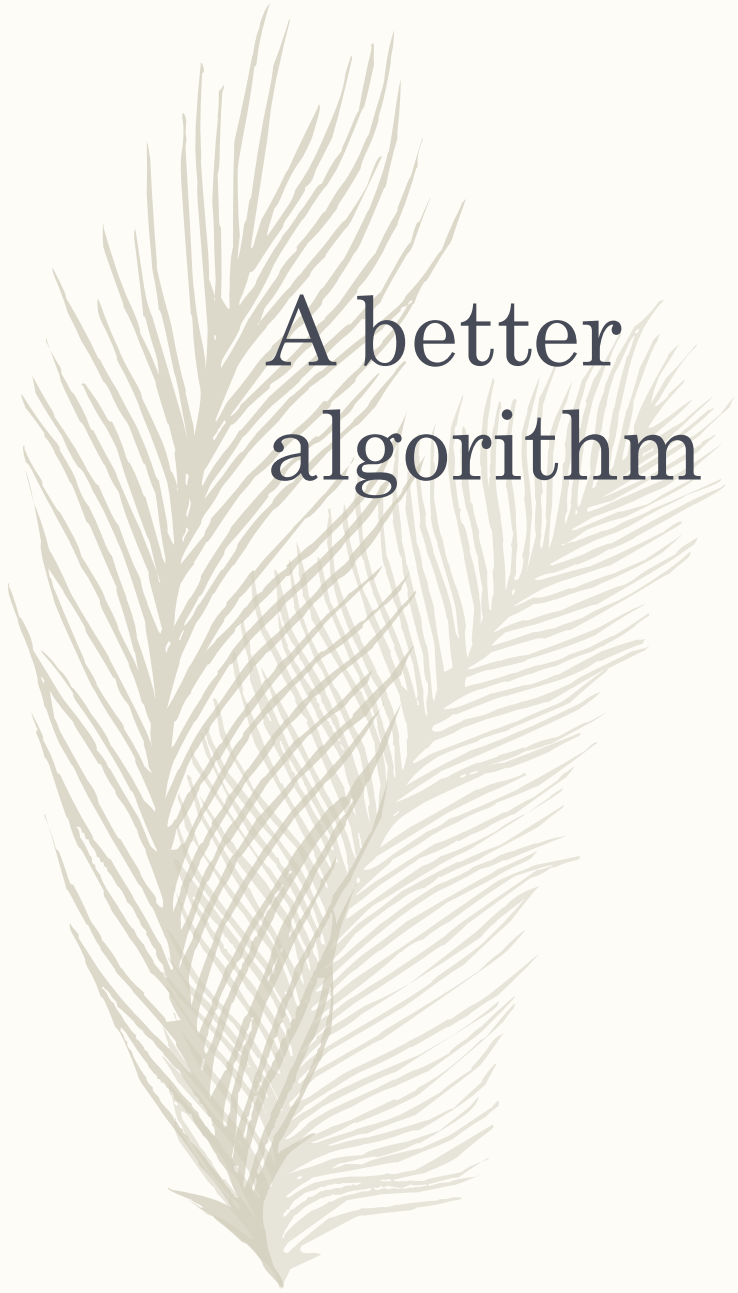
Analysis of the algorithm

There is no failure condition for our algorithm. If G is a DAG, it has a topological order (and our algorithm finds it).

What is the runtime of our algorithm?

- Finding the node with no incoming edges will require looping over all nodes and edges: $\Theta(m+n)$
- We repeat this n times, for a total of $\Theta(n^2+mn)$

Can we do better?



A better algorithm

Find the in-degree of all nodes: this takes $\Theta(m+n)$.

Find all nodes with in-degree 0, and add them to a queue: $\Theta(n)$

Repeat until the queue is empty: (n repeats)

- Pop u from the queue: this is your next node in the order: $\Theta(1)$
- For all outgoing edges $\langle u, v \rangle$: ($\deg^+(u)$ repeats)
  - *Decrement v's in-degree. If it is now 0, add it to the queue: $\Theta(1)$*

Runtime: $\Theta(m+n)$

A different algorithm

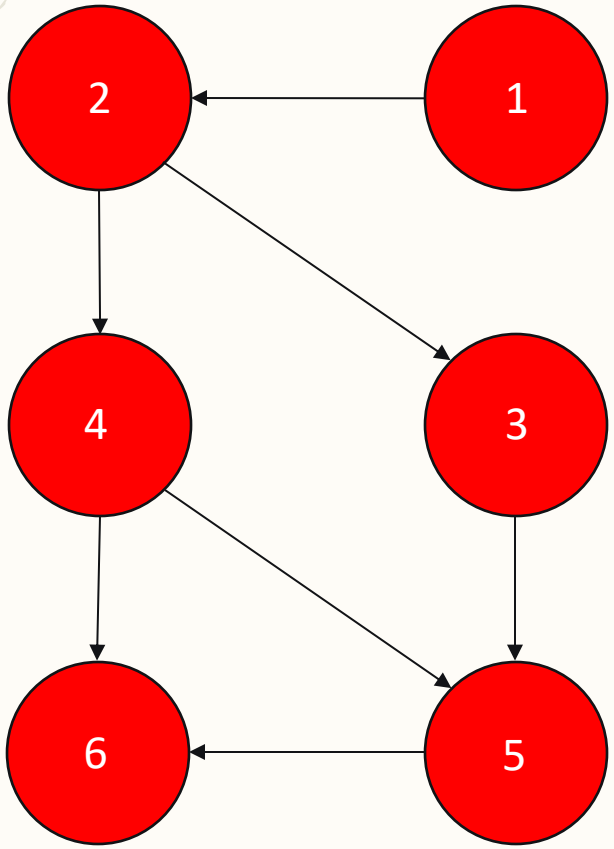
What would it look like if we ran this algorithm in reverse?

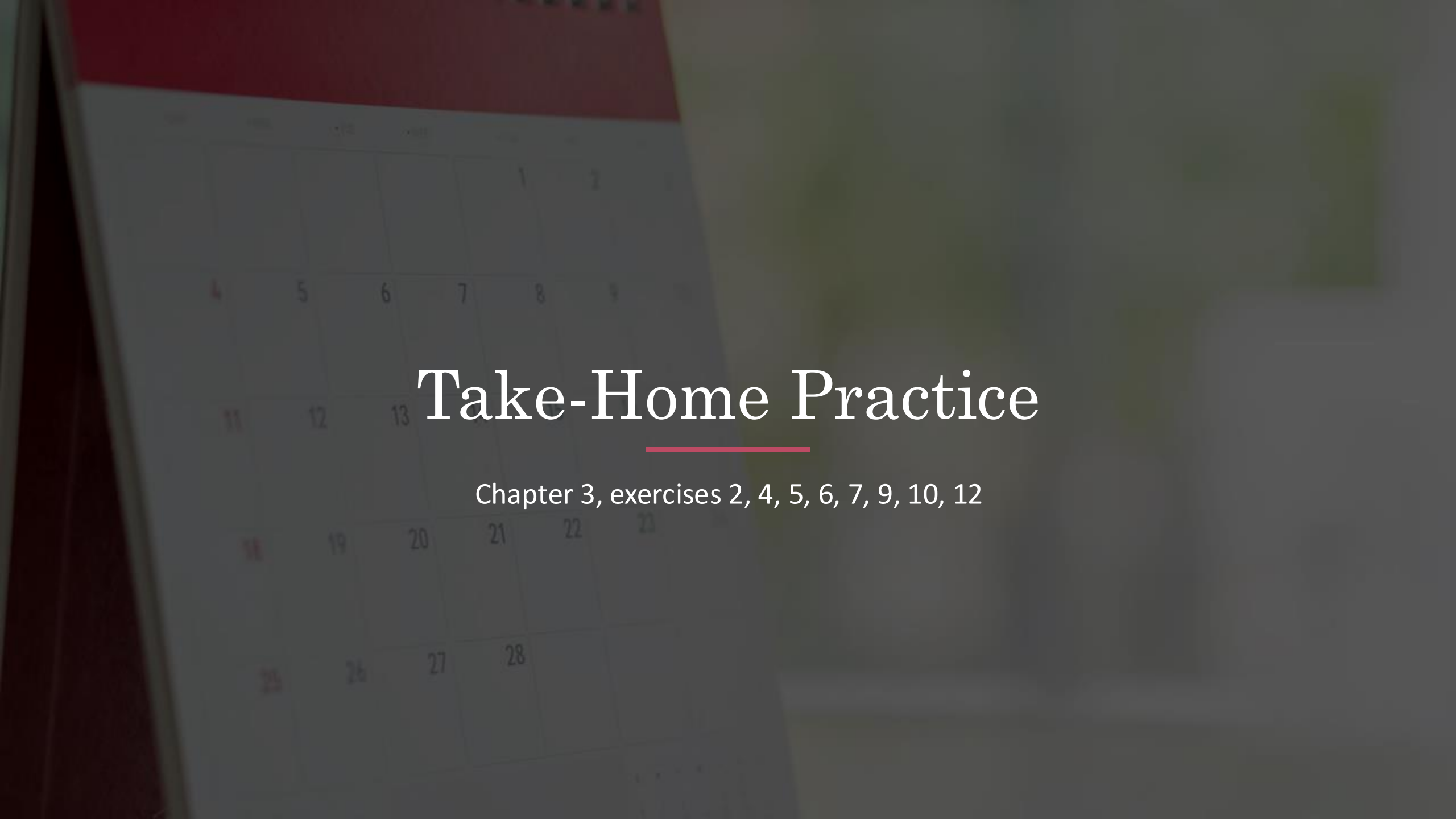
- Find a node with no outgoing edges, which is our last node.
- Remove that node and repeat.

A variant of DFS will do that for us!

- Whenever you leave a node, never to return (going back to its parent), label that as our last node.

An example





Take-Home Practice

Chapter 3, exercises 2, 4, 5, 6, 7, 9, 10, 12