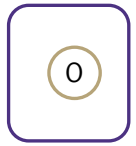
Lecture 22: Graph Modeling

CSE 373: Data Structures and Algorithms

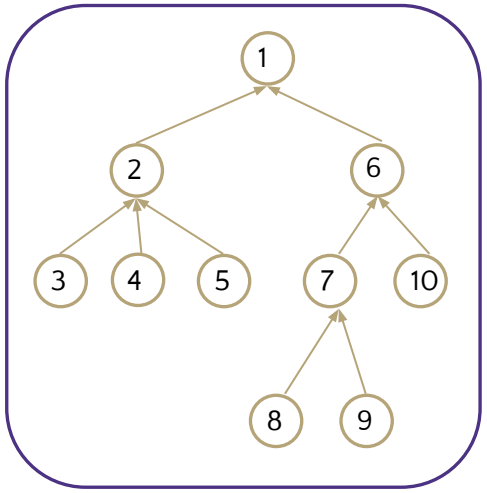
Warm Up

Fill in the array with the correct values representing this Disjoint Set forest
Use the indices that correspond with the values

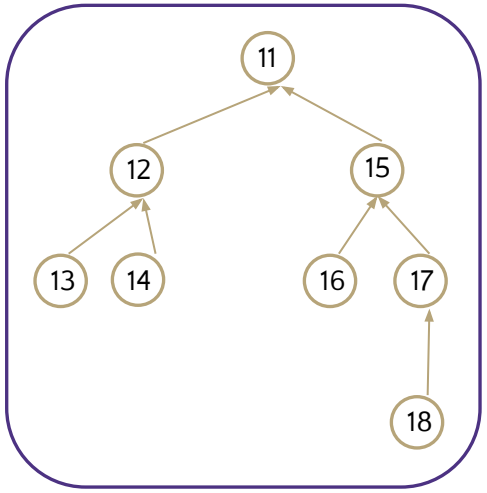
weight = 1



weight = 10



weight = 8

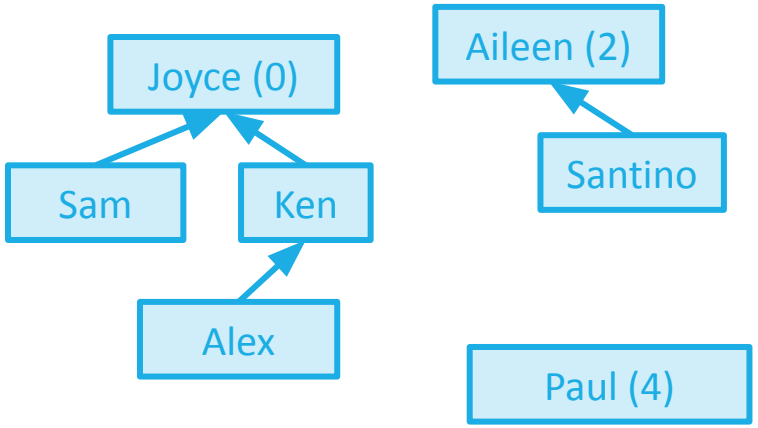


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-1	-10	1	2	2	2	1	6	7	7	6	-8	11	12	12	11	15	15	17

Store (weight * -1)

Each “node” now only takes 4 bytes of memory instead of 32

Using Arrays for Up-Trees



Since every node can have at most one parent, what if we use an array to store the parent relationships?

Proposal: each node corresponds to an index, where we store the index of the parent (or –1 for roots). Use the root index as the representative ID!

Just like with heaps, tree picture still conceptually correct, but exists in our minds!

0	1	2	3	4	5	6
-1	0	-1	6	-1	2	0
Joyce	*Sam*	*Aileen*	*Alex*	*Paul*	*Santino*	*Ken*

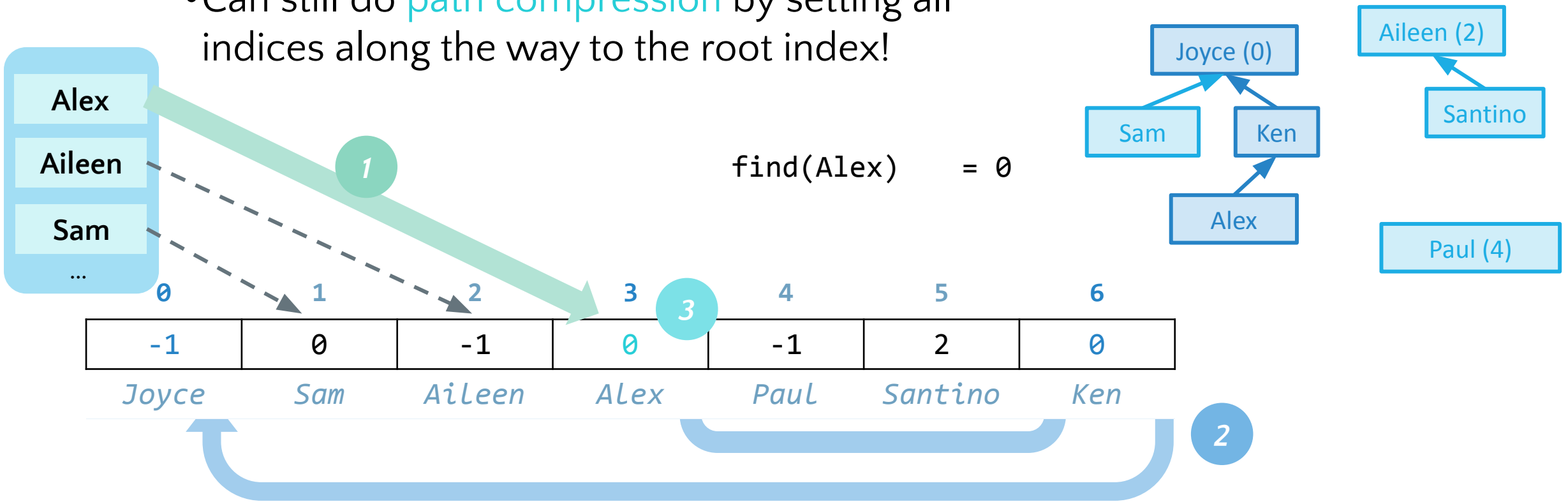
Using Arrays: Find

Initial jump to element still done with extra Map

But traversing up the tree can be done purely within the array!

```
find(A):
1   index = jump to A node's index
2   while array[index] > 0:
      index = array[index]
3   path compression
    return index
```

- Can still do path compression by setting all indices along the way to the root index!

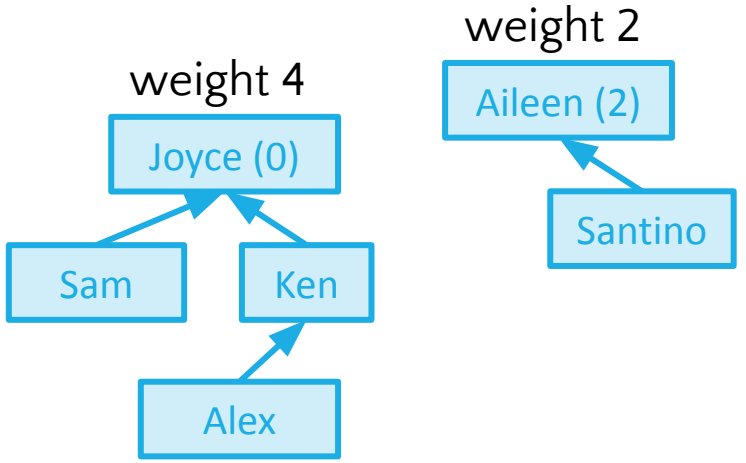


Using Arrays: Union

For WeightedQuickUnion, we need to store the number of nodes in each tree (the weight)

Instead of just storing -1 to indicate a root, we can store -1 * weight!

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  use -1 * array[rootA] and -1 *
    array[rootB] to determine weights
  put lighter root under heavier root
```



union(Ken, Santino)

0	1	2	3	4	5	6
-4	0	-2	6	-1	2	0
Joyce	*Sam*	*Aileen*	*Alex*	*Paul*	*Santino*	*Ken*

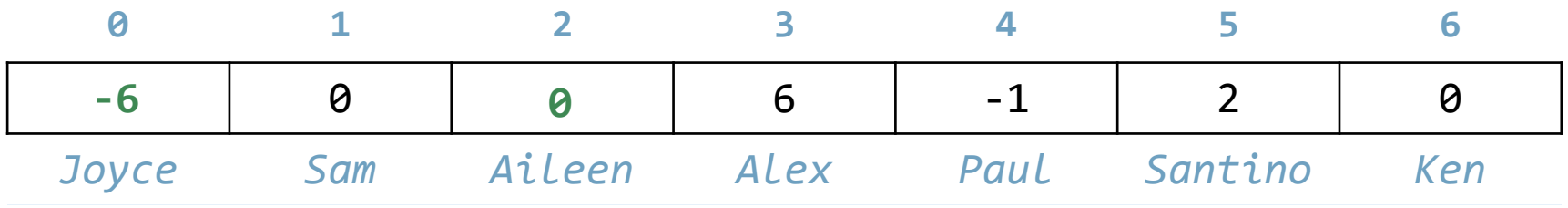
Using Arrays: Union

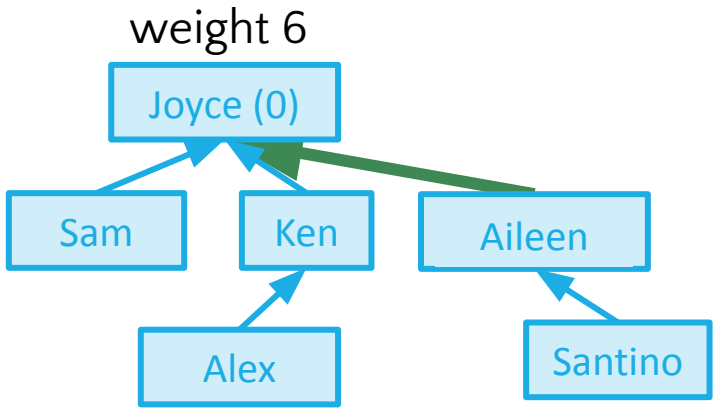
For WeightedQuickUnion, we need to store the number of nodes in each tree (the weight)

Instead of just storing -1 to indicate a root, we can store -1 * weight!

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  use -1 * array[rootA] and -1 *
    array[rootB] to determine weights
  put lighter root under heavier root
```

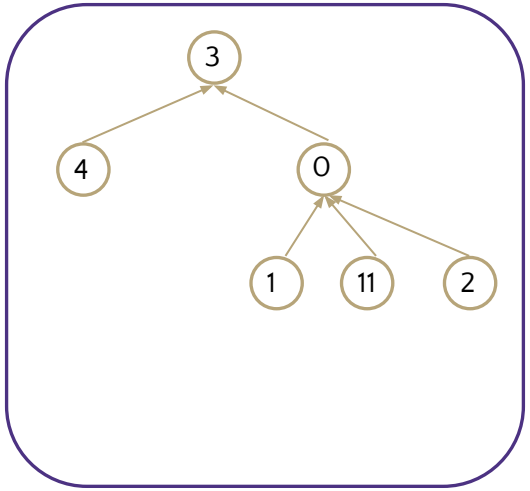
union(Ken, Santino)





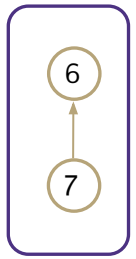
Practice

weight = 6

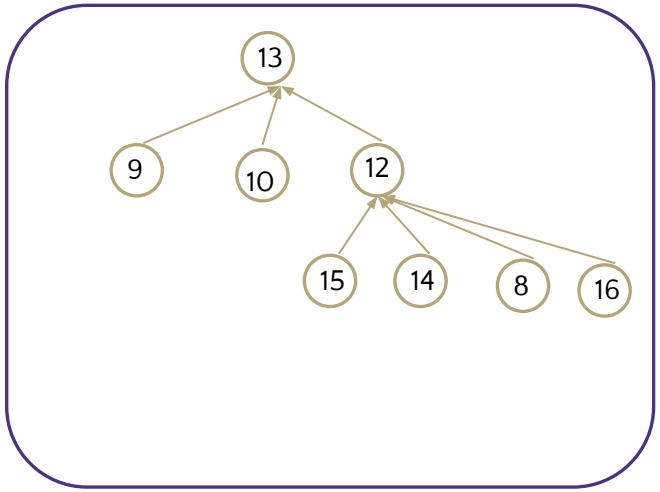


weight = 1

weight = 2



weight = 8

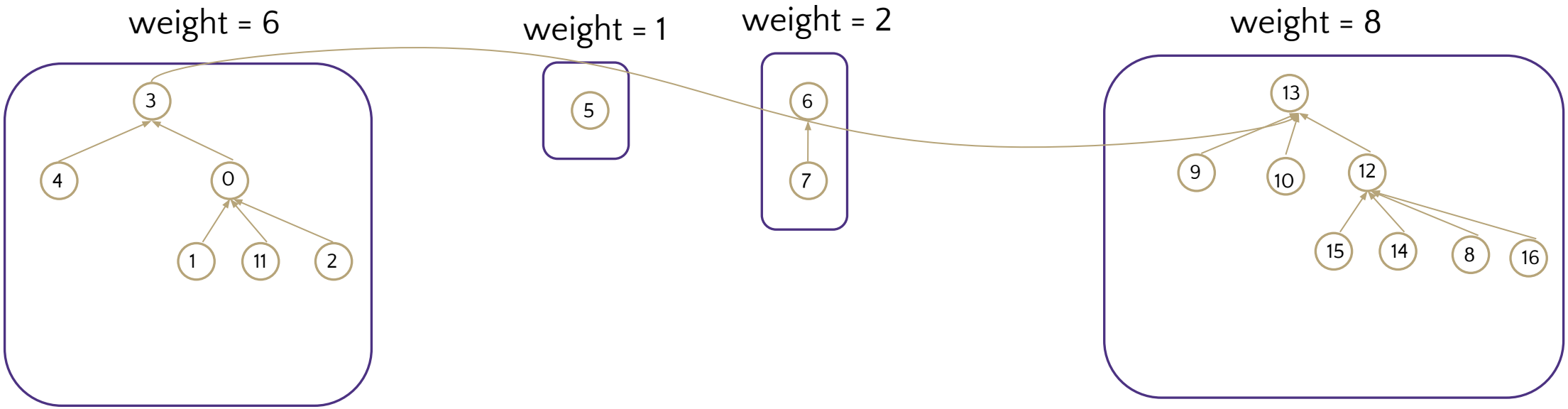


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	0	0	-6	3	-1	-2	6	12	13	13	0	13	-8	12	12	12

union(2, 16)

Practice

union(2, 16)
findSet(2) with path compression
findSet(16) with path compression
union(3,13) by weight



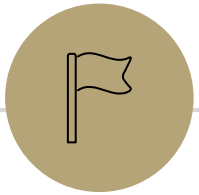
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	0	3	13	3	-1	-2	6	12	13	13	0	13	-8	12	12	13

Using Arrays for WQU+PC

Same asymptotic runtime as using tree nodes, but check out all these other benefits:

- More compact in memory
- Better spatial locality, leading to better constant factors from cache usage
- Simplify the implementation!

	(Baseline)	QuickFind	QuickUnion	WeightedQuickUnion	WQU + Path Compression	ArrayWQU+PC
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log^* n)$	$\Theta(\log^* n)$
`union(x, y)` assuming root args	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log^* n)$	$\Theta(\log^* n)$



Implementing Dijkstra’s

Review Dijkstra's Algorithm: Key Properties

Once a vertex is marked known, its shortest path is known
- Can reconstruct path by following back-pointers (in `edgeTo` map)

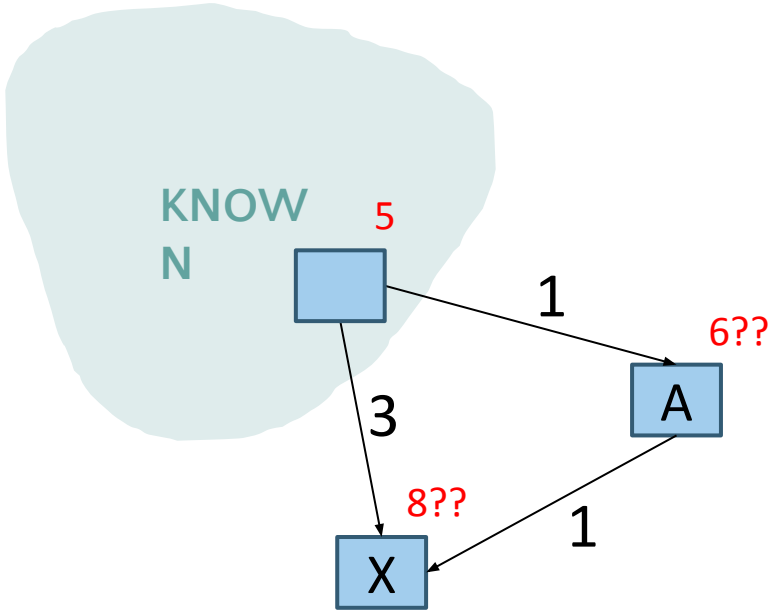
While a vertex is not known, another shorter path might be found
- We call this update **relaxing** the distance because it only ever shortens the current best path

Going through closest vertices first lets us confidently say no shorter path will be found once known
- Because not possible to find a shorter path that uses a farther vertex we'll consider later

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)      // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
```

Review Why Does Dijkstra's Work?



Example:

- We're about to add X to the known set
- But how can we be sure we won't later find a path through some node A that is shorter to X?

INVARIANT

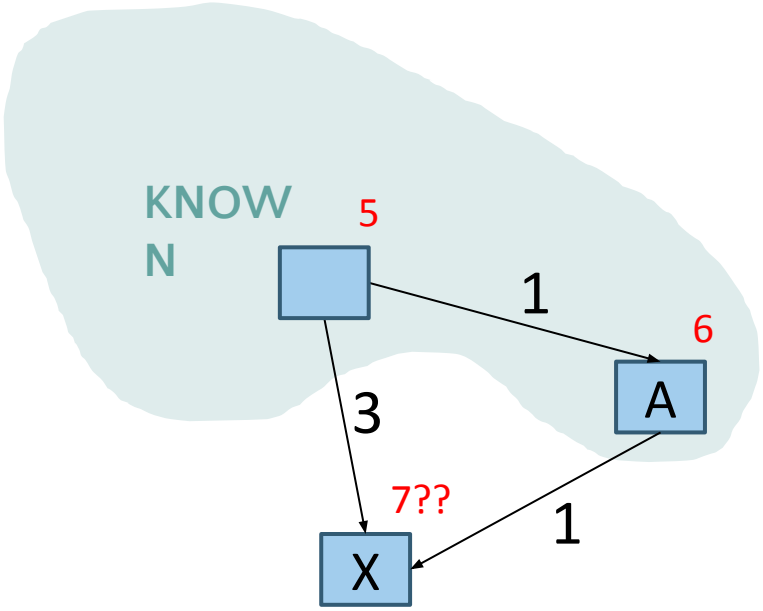
Dijkstra's Algorithm Invariant

All vertices in the "known" set have the correct shortest path

Similar "First Try Phenomenon" to BFS

How can we be sure we won't find a shorter path to X later?

Review Why Does Dijkstra's Work?



Example:

- We're about to add X to the known set
- But how can we be sure we won't later find a path through some node A that is shorter to X?
- Because *if we could, Dijkstra's would explore A first*

INVARIANT

Dijkstra's Algorithm Invariant

All vertices in the "known" set have the correct shortest path

Similar "First Try Phenomenon" to BFS

How can we be sure we won't find a shorter path to X later?

- **Key Intuition**: Dijkstra's works because:
  - IF we always add the closest vertices to "known" first,
  - THEN by the time a vertex is added, any possible relaxing has happened and the path we know is *always the shortest*!

Implementing Dijkstra's

How do we implement "let u be the closest unknown vertex"?

- Would sure be convenient to store vertices in a structure that…
  - Gives them each a distance "priority" value
  - Makes it fast to grab the one with the smallest distance
  - Lets us update that distance as we discover new, better paths

MIN PRIORITY QUEUE ADT

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)      // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(u, newDist)
        edgeTo.put(u, v)
```

Implementing Dijkstra's: Pseudocode

Use a MinPriorityQueue to keep track of the perimeter

- Don't need to track entire graph
- Don't need separate "known" set – implicit in PQ (we'll never try to update a "known" vertex)

This pseudocode is much closer to what you'll implement in P4

- However, still some details for you to figure out!
- e.g. how to initialize distTo with all nodes mapped to ∞
- Spec will describe some optimizations for you to make ☺️

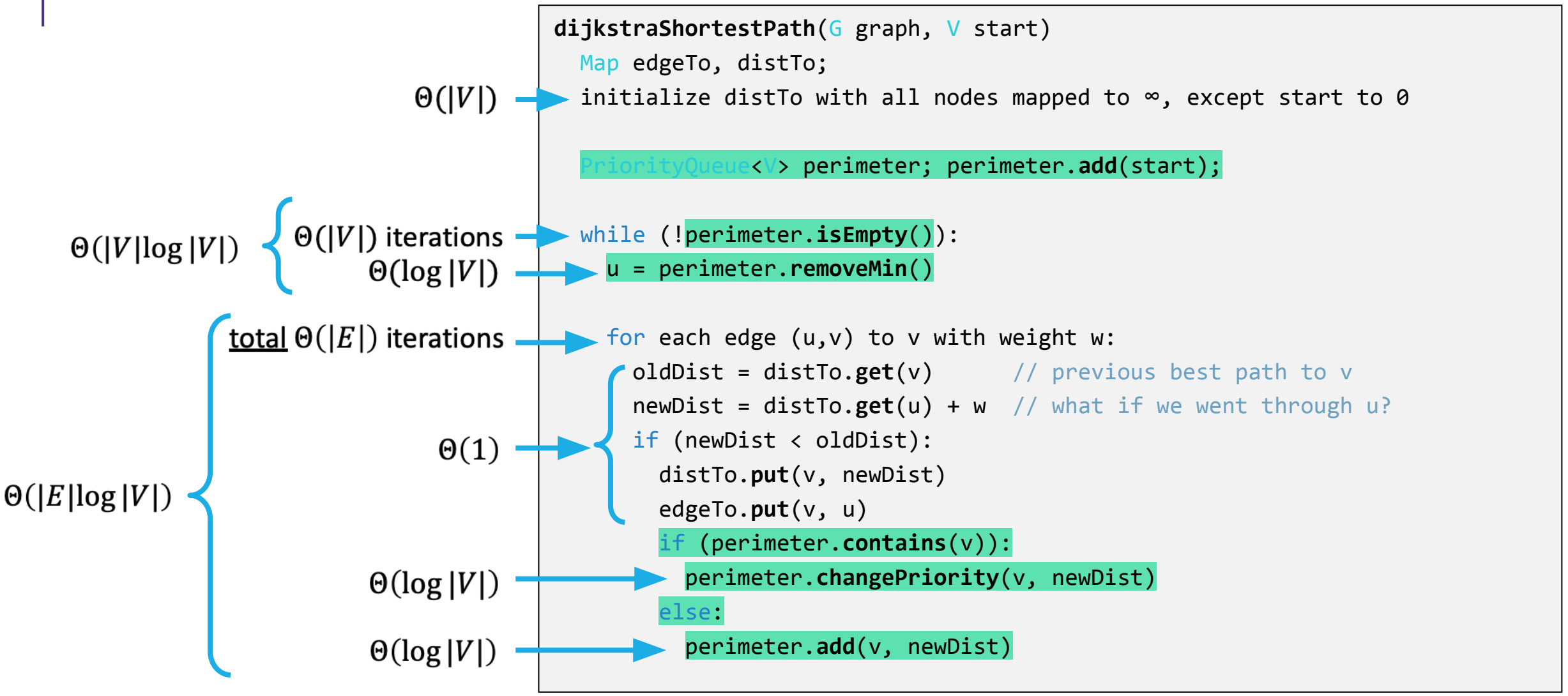
```
dijkstraShortestPath(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  PriorityQueue<V> perimeter; perimeter.add(start);

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()

    for each edge (u,v) to v with weight w:
      oldDist = distTo.get(v)      // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
```

Dijkstra's Runtime



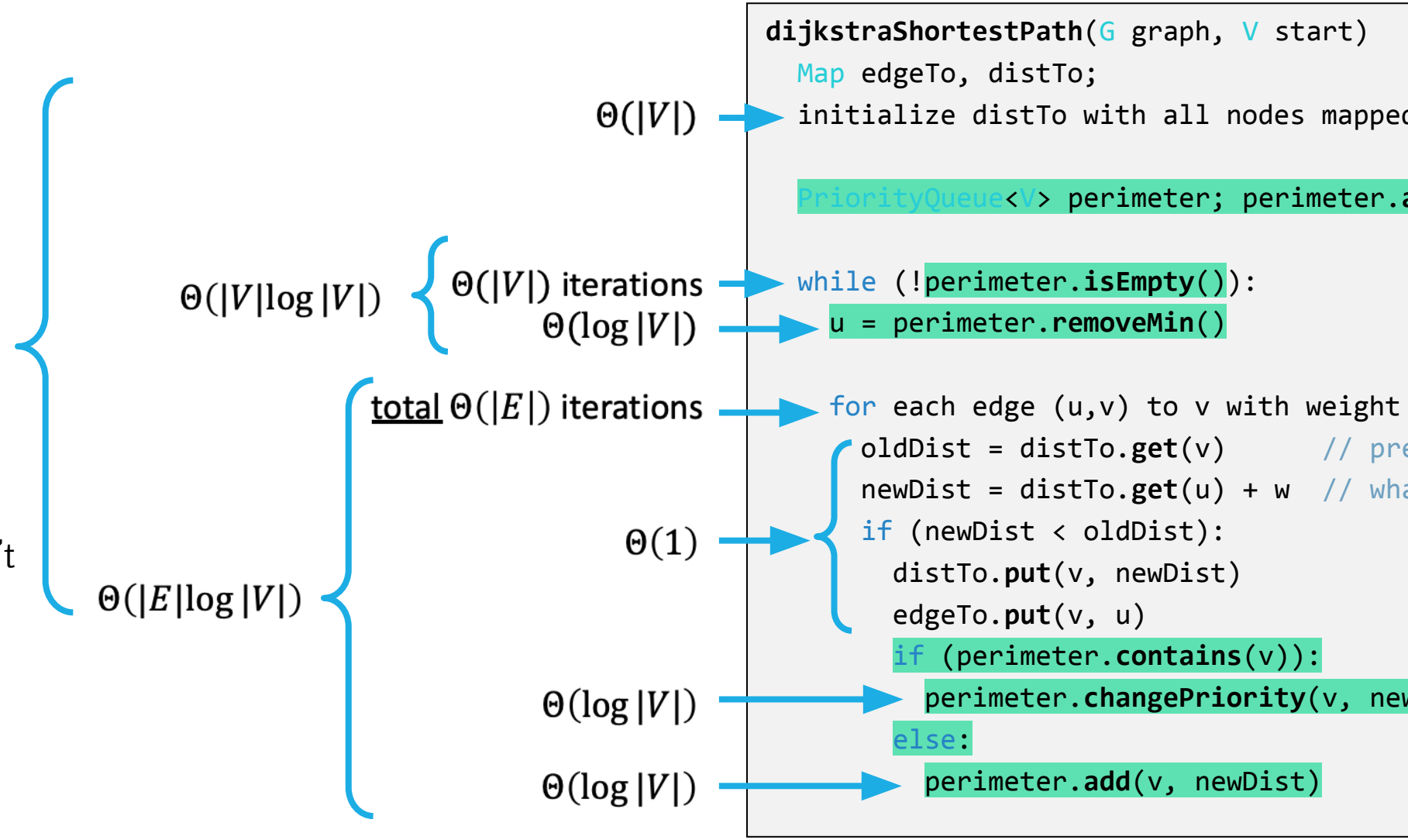
Dijkstra's Runtime

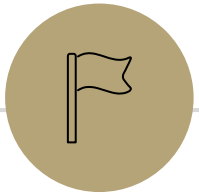
Final result:

$$\Theta(|V|\log|V| + |E|\log|V|)$$

Why can't we simplify further?

- We don't know if |V| or |E| is going to be larger, so we don't know which term will dominate.
- *Sometimes we assume |E| is larger than |V|, so |E|log|V| dominates. But not always true!*





Topological Sort

Topological Sort

A **topological sort** of a directed graph G is an ordering of the nodes, where for every edge in the graph, the origin appears before the destination in the ordering

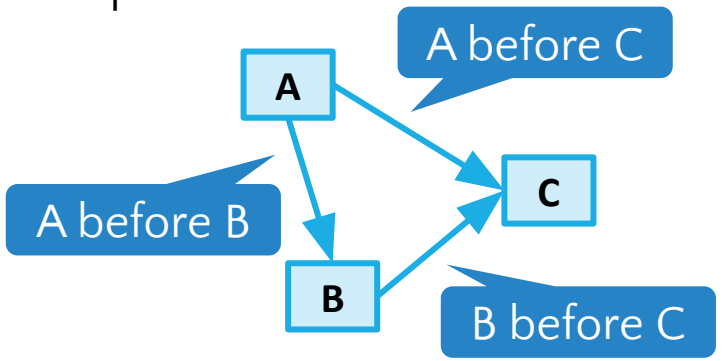
Intuition: a "dependency graph"

- An edge (u, v) means u must happen before v
- A topological sort of a dependency graph gives an ordering that **respects dependencies**

Applications:

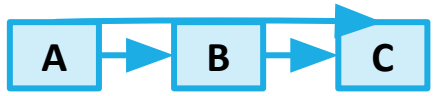
- Graduating
- Compiling multiple Java files
- Multi-job Workflows

Input:



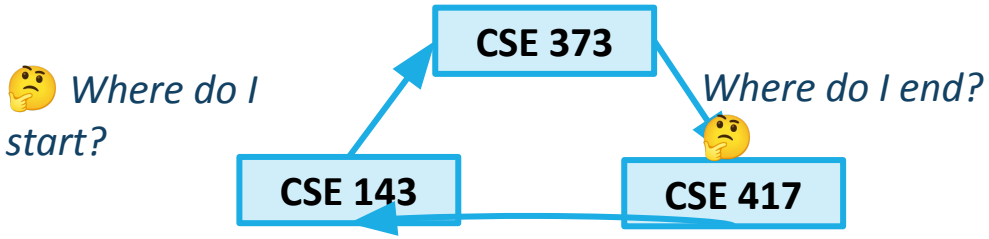
Topological Sort:

With original edges for reference:



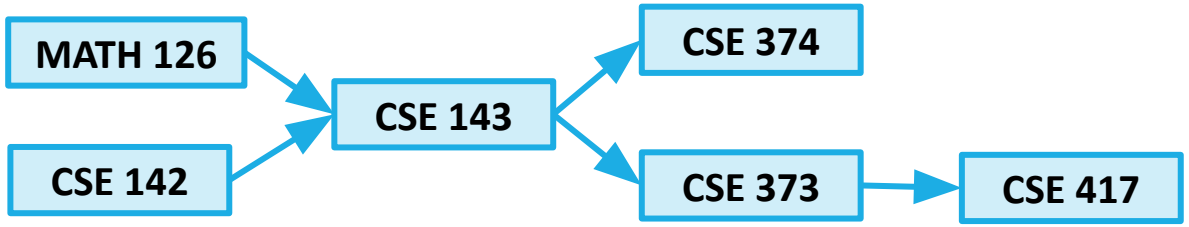
Can We Always Topo Sort a Graph?

Can you topologically sort this graph?



No 😭

What's the difference between this graph and our first graph?



A graph has a topological ordering if it is a DAG

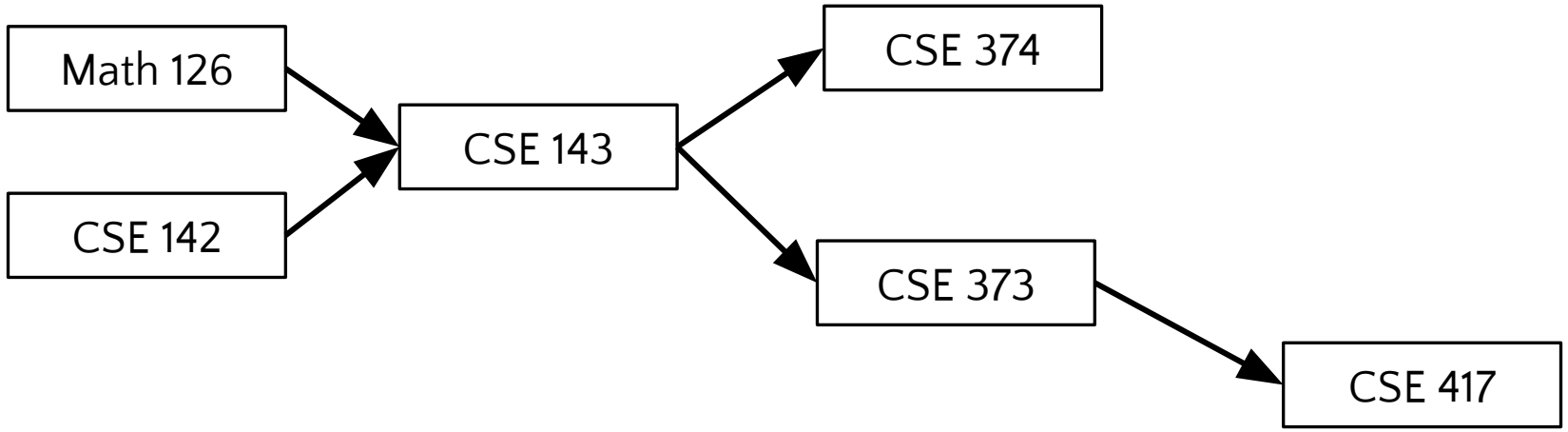
- But a DAG can have multiple orderings

DIRECTED ACYCLIC GRAPH

- A **directed graph** without any **cycles**
- Edges may or may not be weighted

Problem 1: Ordering Dependencies

Today's (first) problem: Given a bunch of courses with prerequisites, find an order to take the courses in.



Problem 1: Ordering Dependencies

Given a directed graph G, where we have an edge from u to v if u must happen before v.

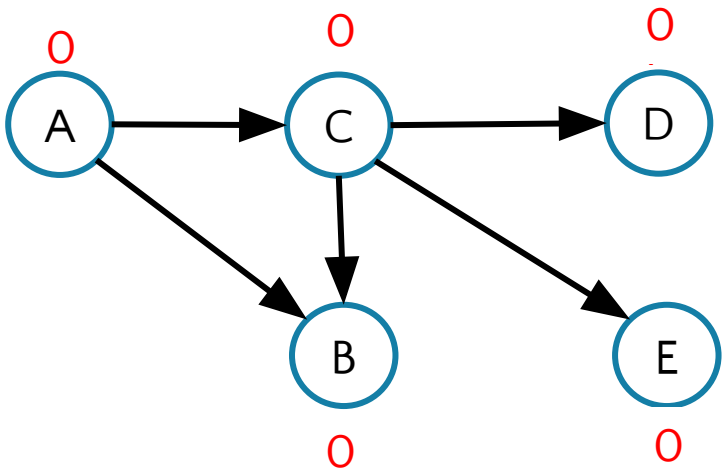
We can only do things one at a time, can we find an order that **respects dependencies**?

Topological Sort (aka Topological Ordering)

Given: a directed graph G
Find: an ordering of the vertices so all edges go from left to right (all the dependency arrows are satisfied and the vertices can be processed left to right with no problems) .

Ordering a DAG

Does this graph have a topological ordering? If so find one.

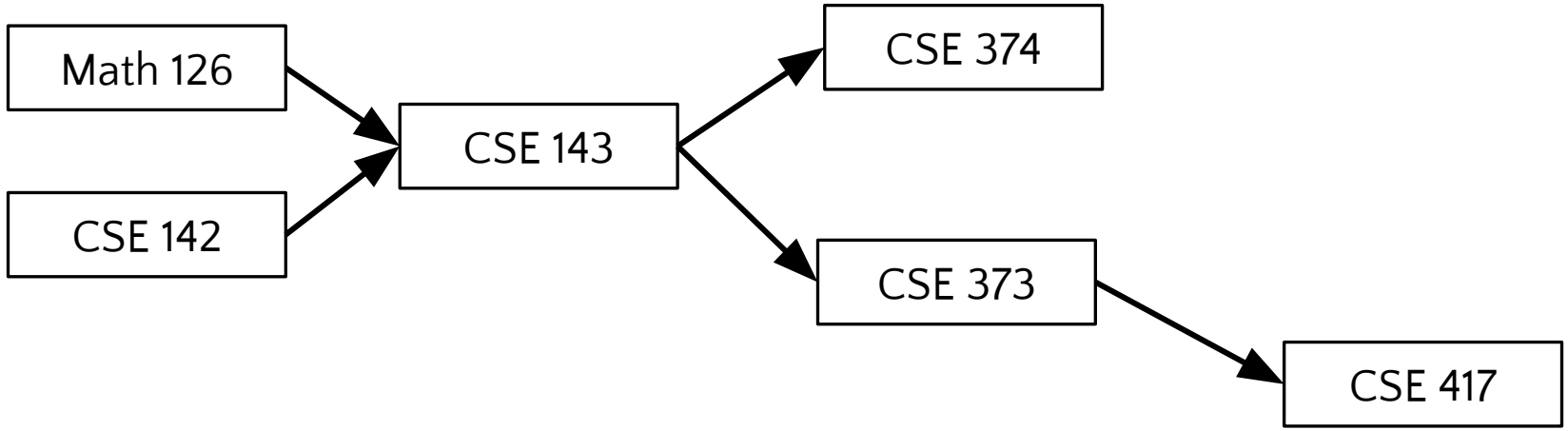


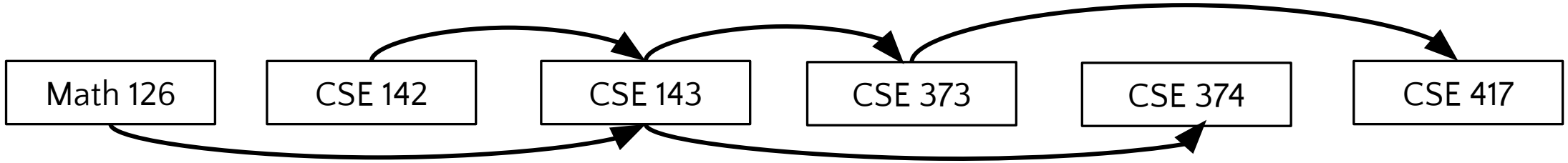
A C B D E

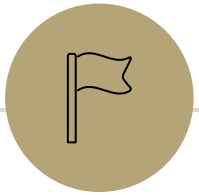
If a vertex doesn't have any edges going into it, we can add it to the ordering.
More generally, if the only incoming edges are from vertices already in the ordering, it's safe to add.

Topological Ordering

A course prerequisite chart and a possible topological ordering.







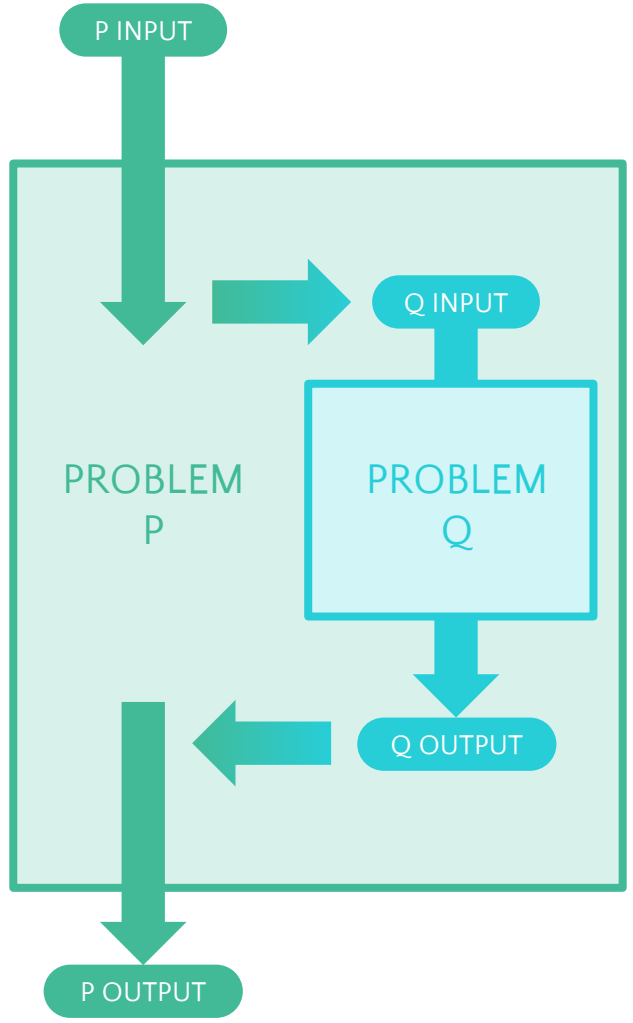
Reductions

Reductions

A **reduction** is a problem-solving strategy that involves using an algorithm for problem Q to solve a different problem P

- Rather than modifying the algorithm for Q, we **modify the inputs/outputs** to make them compatible with Q!
- "P reduces to Q"

1. Convert input for P into input for Q
2. Solve using algorithm for Q
3. Convert output from Q into output from P



Reductions

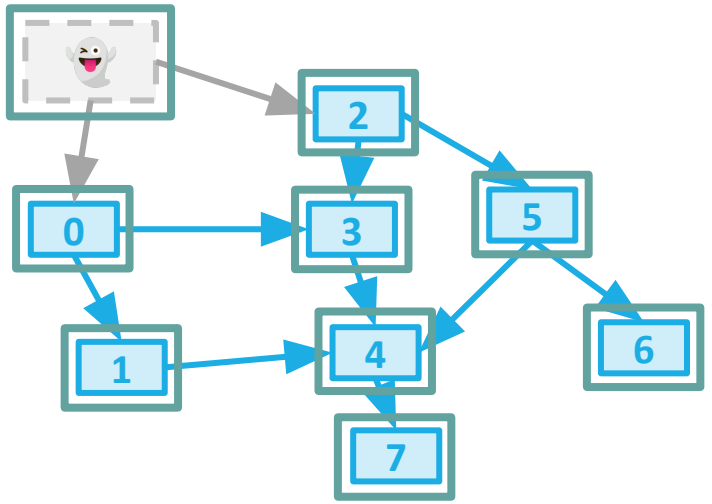
Example: I want to get a note to my friend in Chicago, but walking all the way there is a difficult problem to solve ☹

- Instead, **reduce** the "get a note to Chicago" problem to the "mail a letter" problem!

1. Place note inside of envelope
2. Mail using US Postal Service
3. Take note out of envelope

How To Perform Topo Sort?

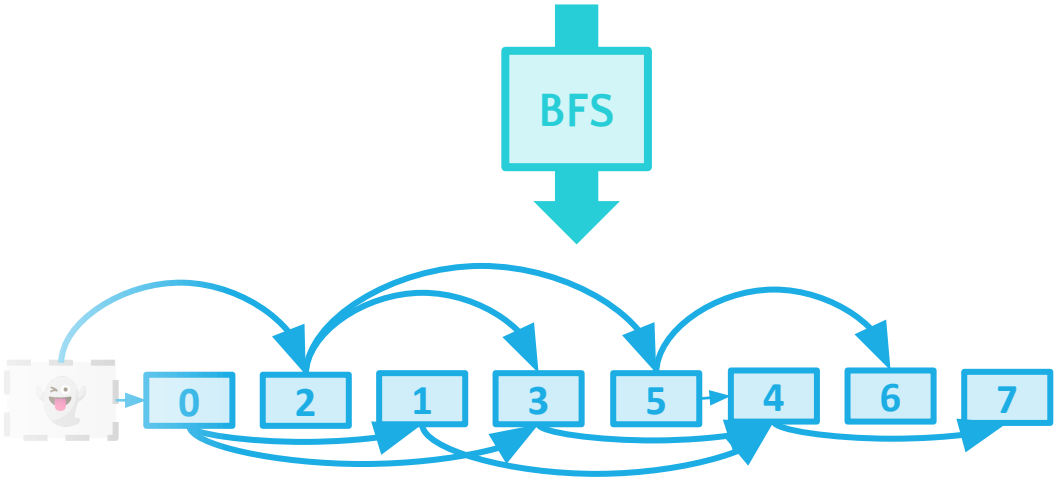
If we add a phantom “start” vertex pointing to other starts, we could use BFS!



Performing Topo Sort

Reduce topo sort to BFS by modifying graph, running BFS, then modifying output back

Sweet sweet victory 😎

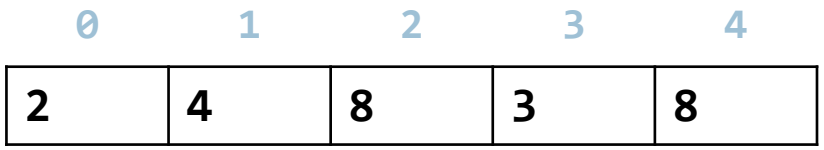


Checking for Duplicates

Problem: We want to determine whether an array contains duplicate elements.

Initial idea: Compare every element to every other element!

- Runtime: $\theta(n^2)$




```
containsDuplicates(array) {
  for (int i = 0; i < array.length; i++):
    for (int j = i; j < array.length; j++):
      if (array[i] == array[j]):
        return true
  return false
}
```

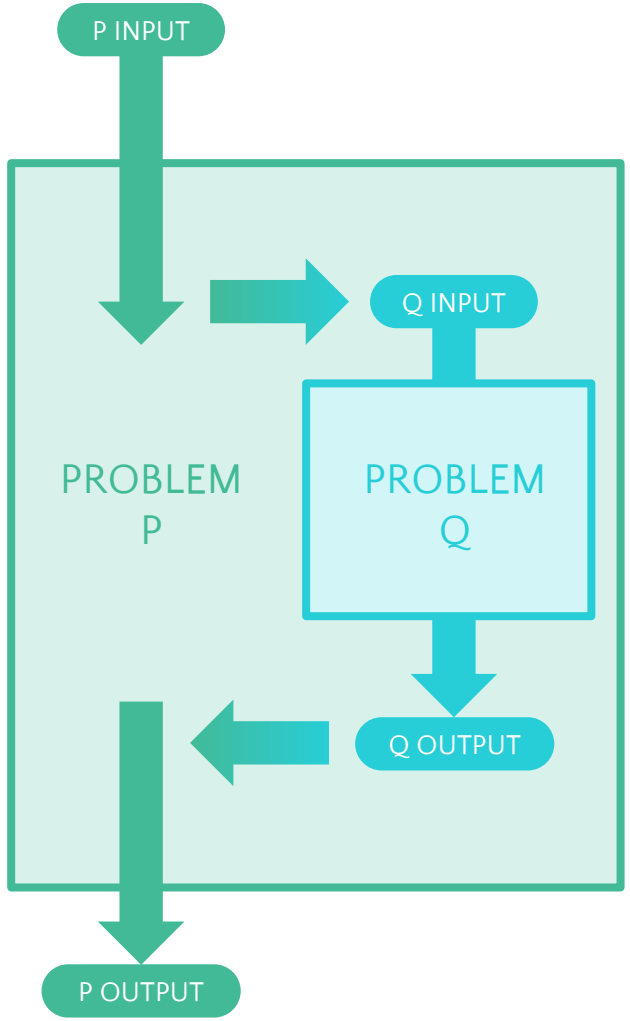
Goal of a Reduction

Goal: Reduce the problem of "Contains Duplicates?" to another problem we have an algorithm for.

Try to identify each of the following:

1. How will you convert the "Contains Duplicates?" input?
2. What algorithm will you apply?
3. How will you convert the algorithm's output?

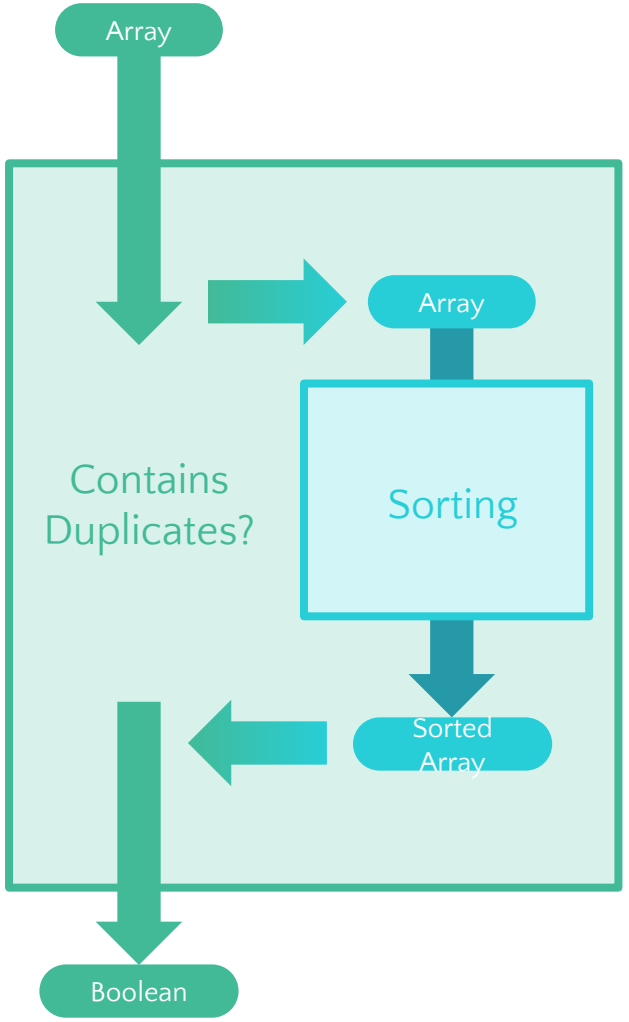
0	1	2	3	4
2	4	8	3	8

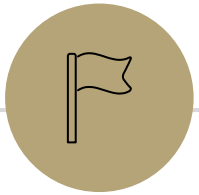


One Solution: Sorting!

One Solution: Reduce "Contains Duplicates?" to the problem of *sorting an array*

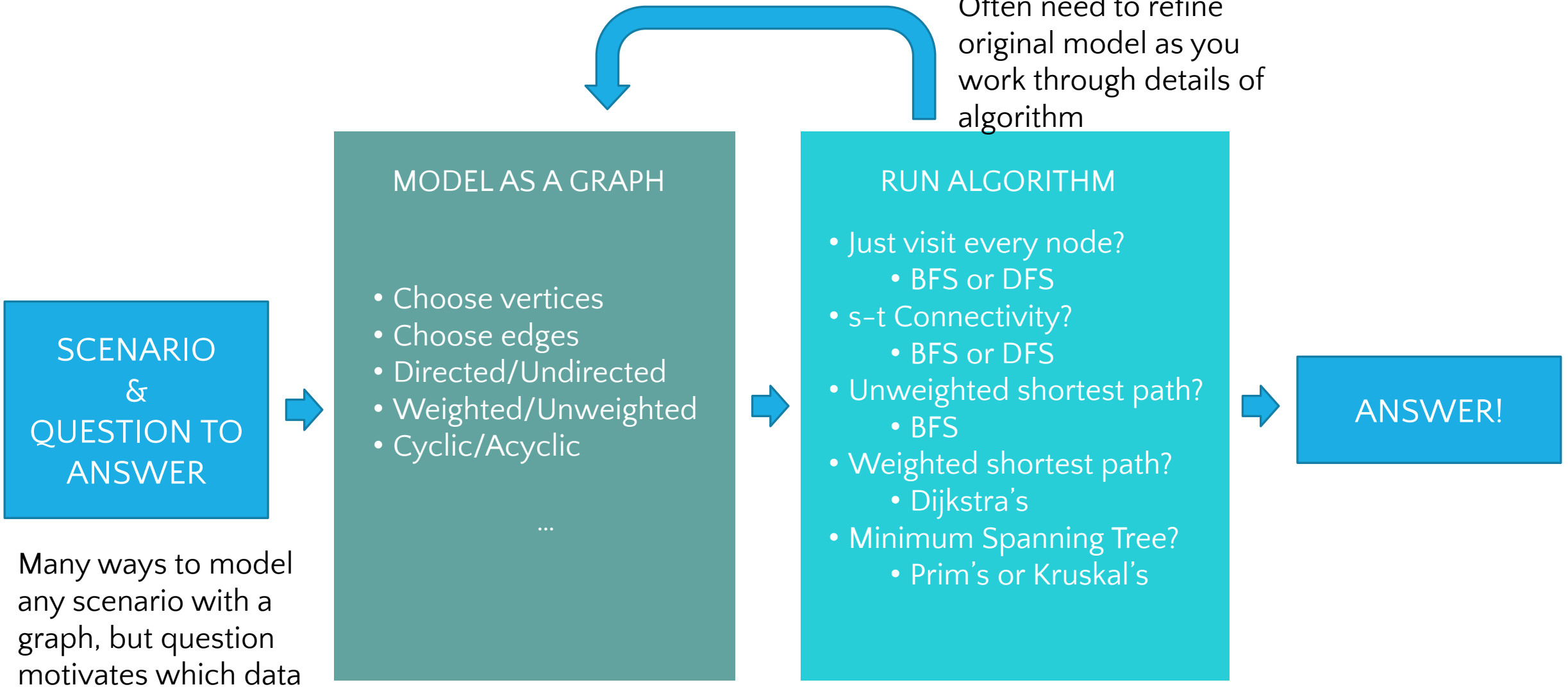
- We know several algorithms that solve this problem quickly!
  1. Simply pass array input to "Sorting"
  2. Use Heap Sort, Merge Sort, or Quick Sort to sort
  3. Scan through sorted array: check for duplicates now *next to each other*, a $\theta(n)$ operation!
- Totally okay to do work in input/output conversion! Even with this pass, runtime is $\theta(n \log n + n)$, so just $\theta(n \log n)$. Reduction helped us avoid quadratic runtime!





Graph Modeling Review

Recap: Graph Modeling



Graph Modeling Activity

Note Passing – Part I

Imagine you are an American High School student. You have a very important note to pass to your crush, but the two of you do not share a class so you need to rely on a chain of friends to pass the note along for you. A note can only be passed from one student to another when they share a class, meaning when two students have the same teacher during the same class period.

Unfortunately, the school administration is not as romantic as you, and passing notes is against the rules. If a teacher sees a note, they will take it and destroy it. Figure out if there is a sequence of handoffs to enable you to get your note to your crush.

How could you model this situation as a graph?

	Period 1	Period 2	Period 3	Period 4
You	Smith	Patel	Lee	Brown
Anika	Smith	Lee	Martinez	Brown
Bao	Brown	Patel	Martinez	Smith
Carla	Martinez	Jones	Brown	Smith
Dan	Lee	Lee	Brown	Patel
Crush	Martinez	Brown	Smith	Patel

Possible Design

Algorithm

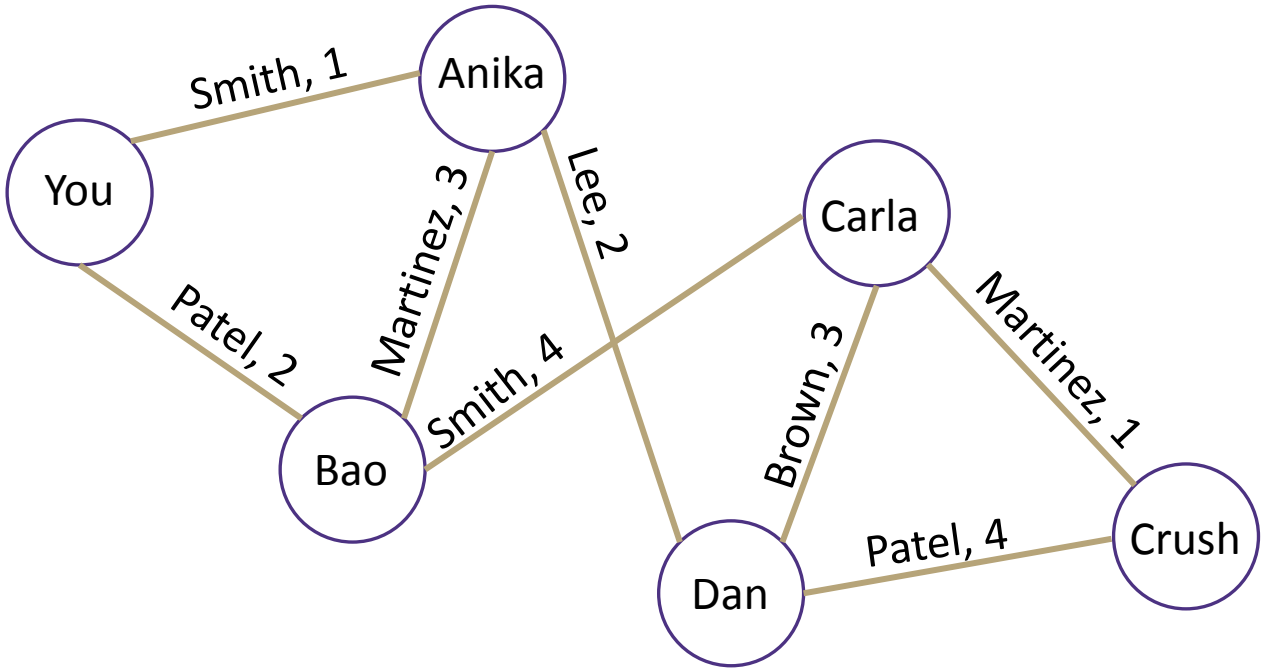
BFS or DFS to see if you and your Crush are connected

Vertices

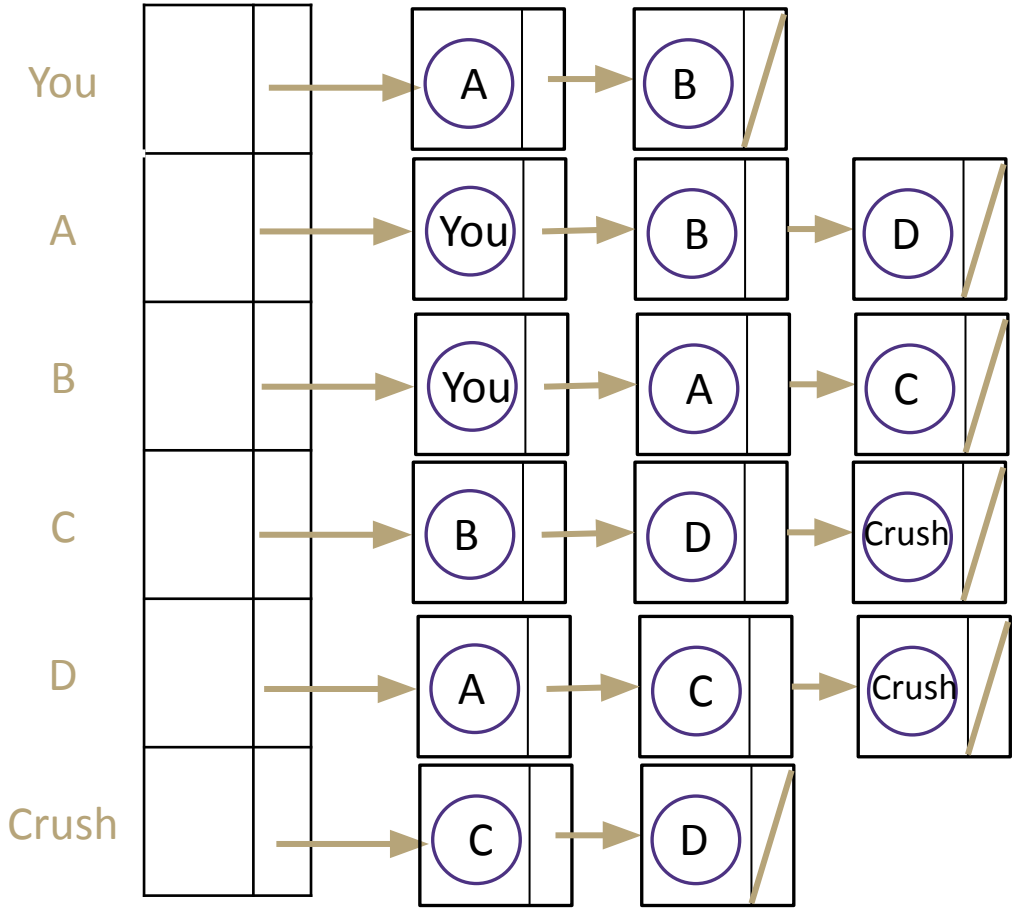
- Students
- Fields: Name, have note

Edges

- Classes shared by students
- Not directed
- Could be left without weights
- Fields: vertex 1, vertex 2, teacher, period



Adjacency List



More Design

Note Passing – Part II

Now that you know there exists a way to get your note to your crush, we can work on picking the best hand off path possible.

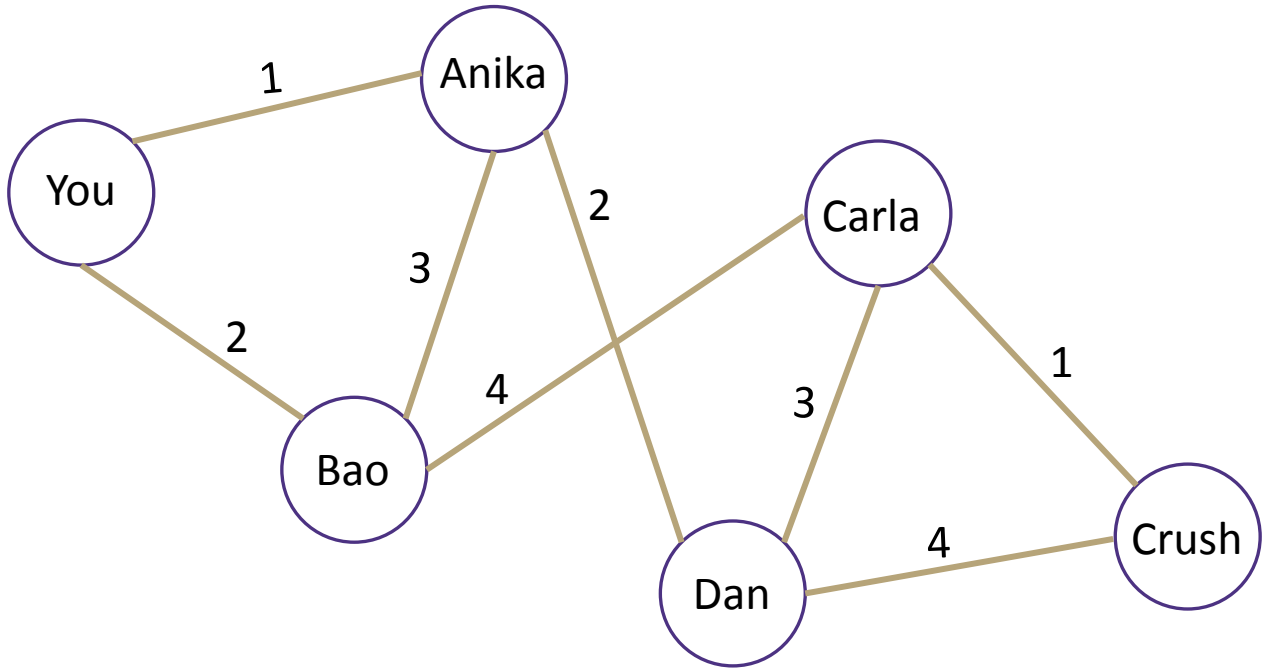
Thought Experiments:

1. What if you want to optimize for time to get your crush the note as early in the day as possible?
   - How can we use our knowledge of which period students share to calculate for time knowing that period 1 is earliest in the day and period 4 is later in the day?
   - How can we account for the possibility that it might take more than a single school day to deliver the note?

2. What if you want to optimize for rick avoidance to make sure your note only gets passed in classes least likely for it to get intercepted?
   - Some teachers are better at intercepting notes than others. The more notes a teacher has intercepted, the more likely it is they will take yours and it will never get to your crush. If we knew how many notes each teacher has intercepted how might we incorporate that into our graph to find the least risky route?

Optimize for Time

"Distance" will represent the sum of which periods the note is passed in, because smaller period values are earlier in the day the smaller the sum the earlier the note gets there except in the case of a "wrap around"



1. Add the period number to each edge as its weight
2. Run Dijkstra's from You to Crush

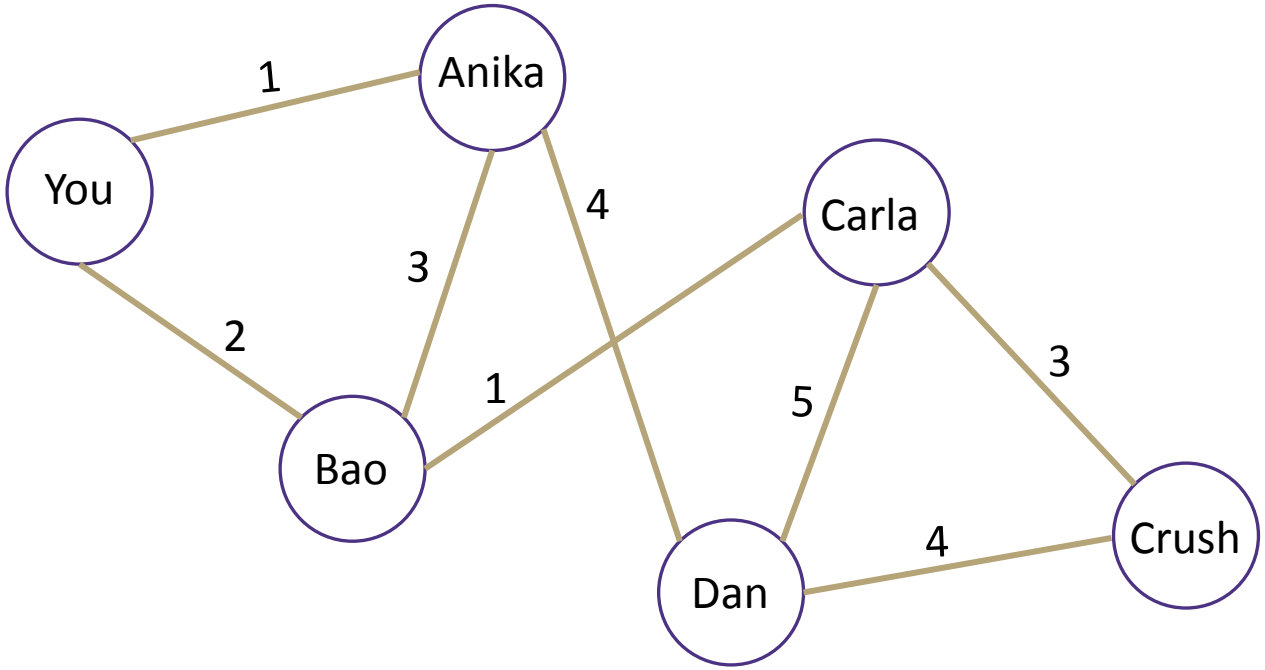
Vertex	Distance	Predecessor	Process Order
You	0	--	0
Anika	1	You	1
Bao	2	You	5
Carla	6	Dan	3
Dan	3	Anika	2
Crush	7	Carla	4*

*The path found wraps around to a new school day because the path moves from a later period to an earlier one

- We can change our algorithm to check for wrap arounds and try other routes

Optimize for Risk

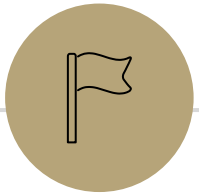
"Distance" will represent the sum of notes intercepted across the teachers in your passing route. The smaller the sum of notes the "safer" the path.



1. Add the number of letters intercepted by the teacher to each edge as its weight
2. Run Dijkstra's from You to Crush

Teacher	Notes Intercepted
Smith	1
Martinez	3
Lee	4
Brown	5
Patel	2

Vertex	Distance	Predecessor	Process Order
You	0	--	0
Anika	1	You	1
Bao	4	Anika	2
Carla	5	Bao	3
Dan	10	Carla	5
Crush	8	Carla	4



Seam Carving

Content-Aware Image Resizing

Seam carving: A distortion-free technique for resizing an image by removing "unimportant seams"

Original Photo

Horizontally-Scaled

(castle and person are distorted)

Seam-Carved

(castle and person are undistorted; "unimportant" sky removed instead)

Seam carving for content-aware image resizing (Avidan, Shamir/ACM); Broadway Tower (Newton2, Yummifruitbat/Wikimedia)

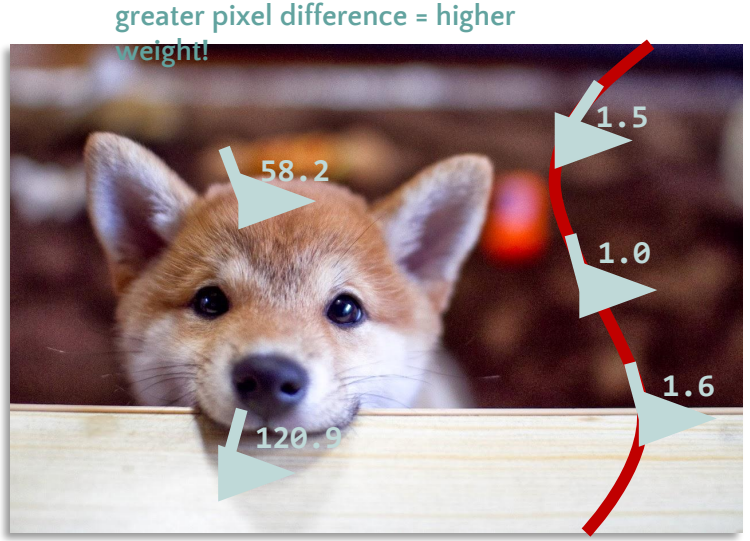
Demo:
https://www.youtube.com/watch?v=vIFCV2spKtg

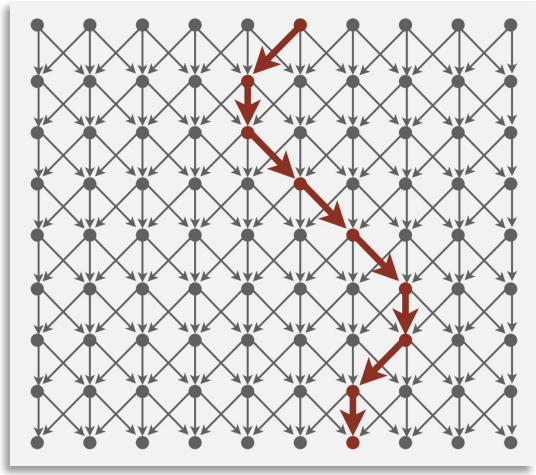
Seam Carving Reduces to Dijkstra's!

1. *Transform the input so that it can be solved by the standard algorithm*
   - Formulate the image as a graph
     - Vertices: pixel in the image
     - Edges: connects a pixel to its 3 downward neighbors
     - Edge Weights: the "energy" (visual difference) between adjacent pixels

2. *Run the standard algorithm as-is on the transformed input*
   - Run Dijkstra's to find the shortest path (sum of weights) from top row to bottom row

3. *Transform the output of the algorithm to solve the original problem*
   - Interpret the path as a removable "seam" of unimportant pixels





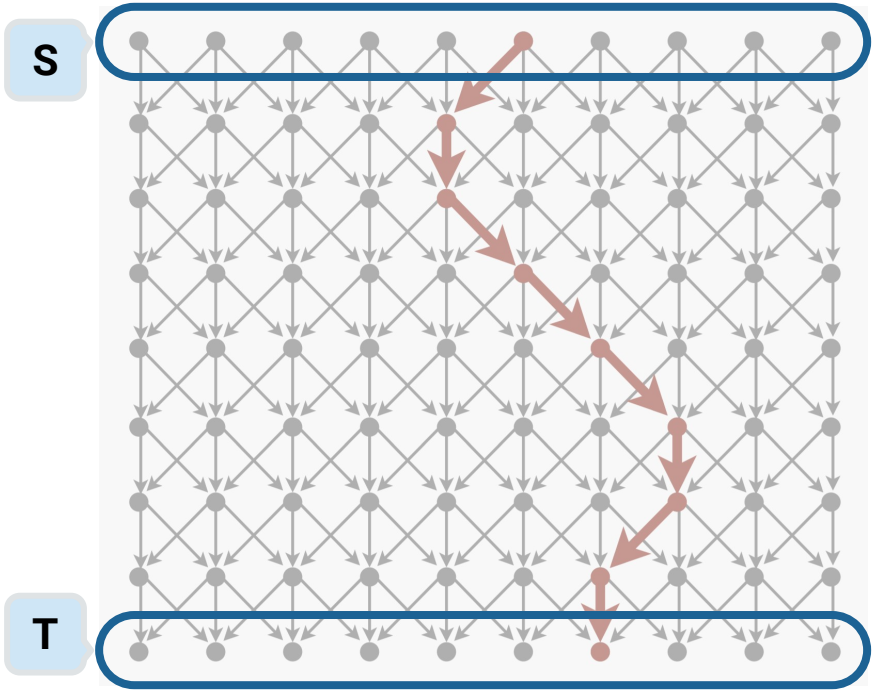
Shortest Paths (Robert Sedgewick, Kevin Wayne/Princeton)

An Incomplete Reduction

Complication:

- Dijkstra's starts with a single vertex S and ends with a single vertex T
- This problem specifies *sets of vertices* for the start and end

Question to think about: how would you transform this graph into something Dijkstra's knows how to operate on?



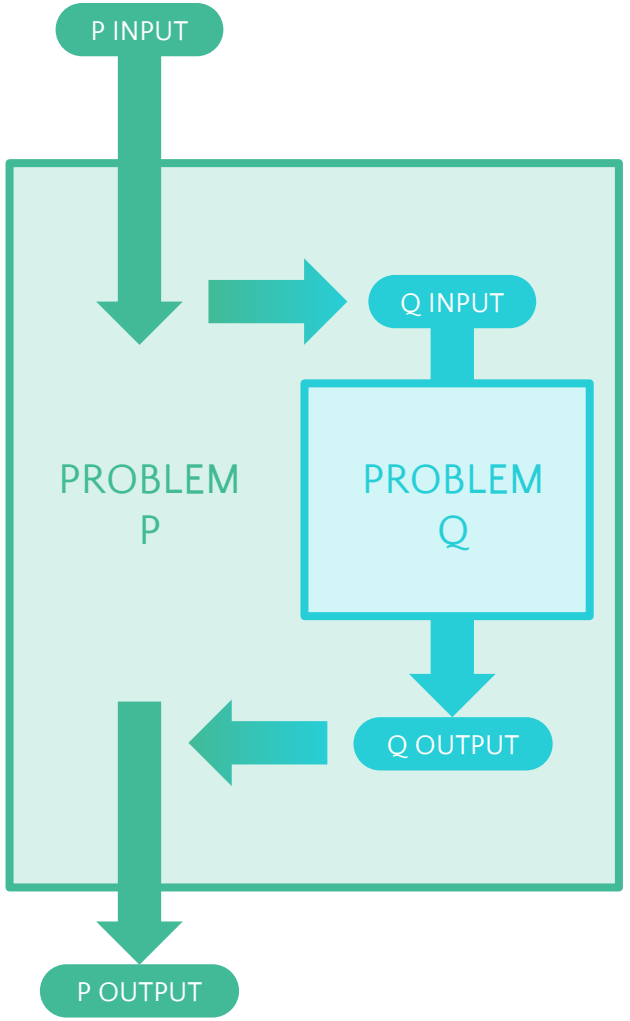
In Conclusion

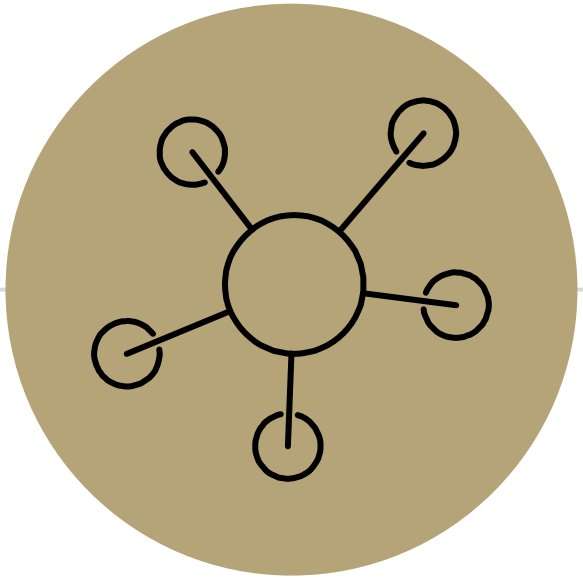
Topo Sort is a widely applicable “sorting” algorithm

Reductions are an essential tool in your CS toolbox -- you’re probably already doing them without putting a name to it

Many more reductions than we can cover!

- Shortest Path in DAG with Negative Edges *reduces to* Topological Sort! (Link)
- 2-Color Graph Coloring *reduces to* 2-SAT (Link)
- …
- Staying on top of the end of the quarter in this course *reduces to* starting early on P4 and EX4/5





Appendix